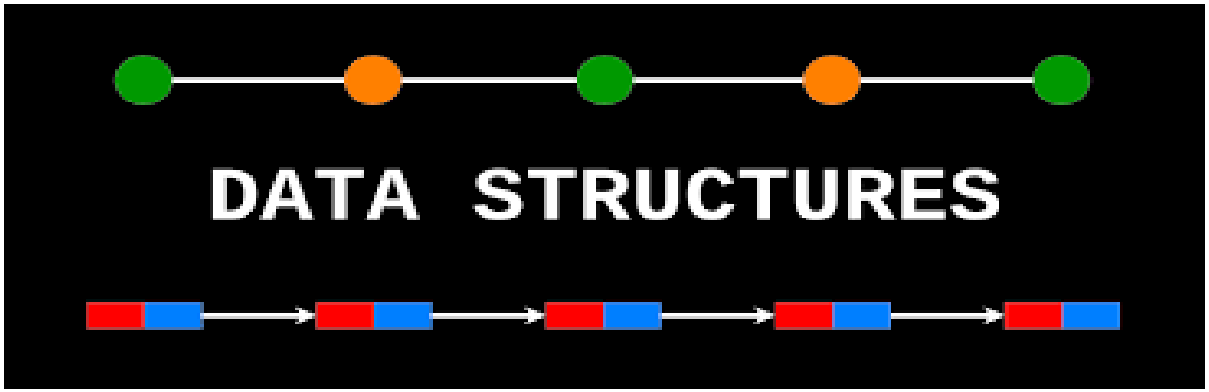
Department of Computer Science and Engineering

DATA STRUCTURE AND APPLICATIONS
BCS304



MODULE 1: INTRODUCTION TO DATASTRUCTURES

DATA STRUCTURE

A data structure is a class of data that can be characterized by its organization and the operations that are defined on it.

Data Structure = Organized Data + Allowed Operations

In other words, the organized collection of data is called data structure. A Data structure is a set of values along with the set of operations permitted on them. It is also required to specify the semantics of operations permitted on the data values, and this is done by using set of axioms, which describes how these operations work.

- A set of data values
- A set of functions specifying the operations permitted on the data values
- A set of axioms describing how these operations work.

Classification of Data Structure

There are various ways to classify data structure :

- Primitive and Non-Primitive Data Structure
- Linear and Non-Linear Data Structure
- Homogenous and Non-Homogeneous Data Structure
- Static and Dynamic Data Structure

Primitive and Non-Primitive Data Structure

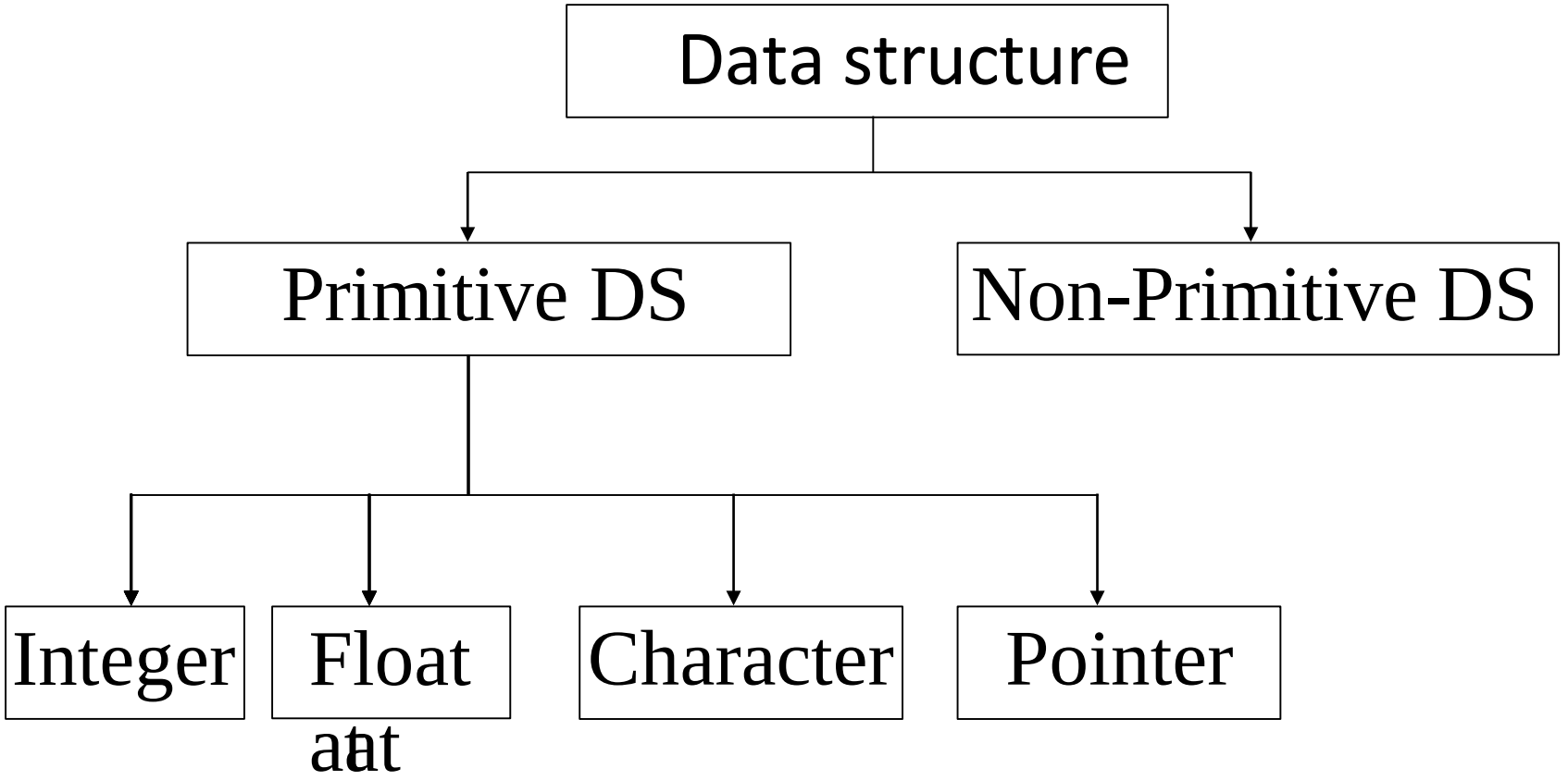
The data structure that are atomic (indivisible) are called primitive. Example are integer, real, Boolean and characters.

The data structure that are not atomic are called non-primitive or composite. Example are records, array and string.

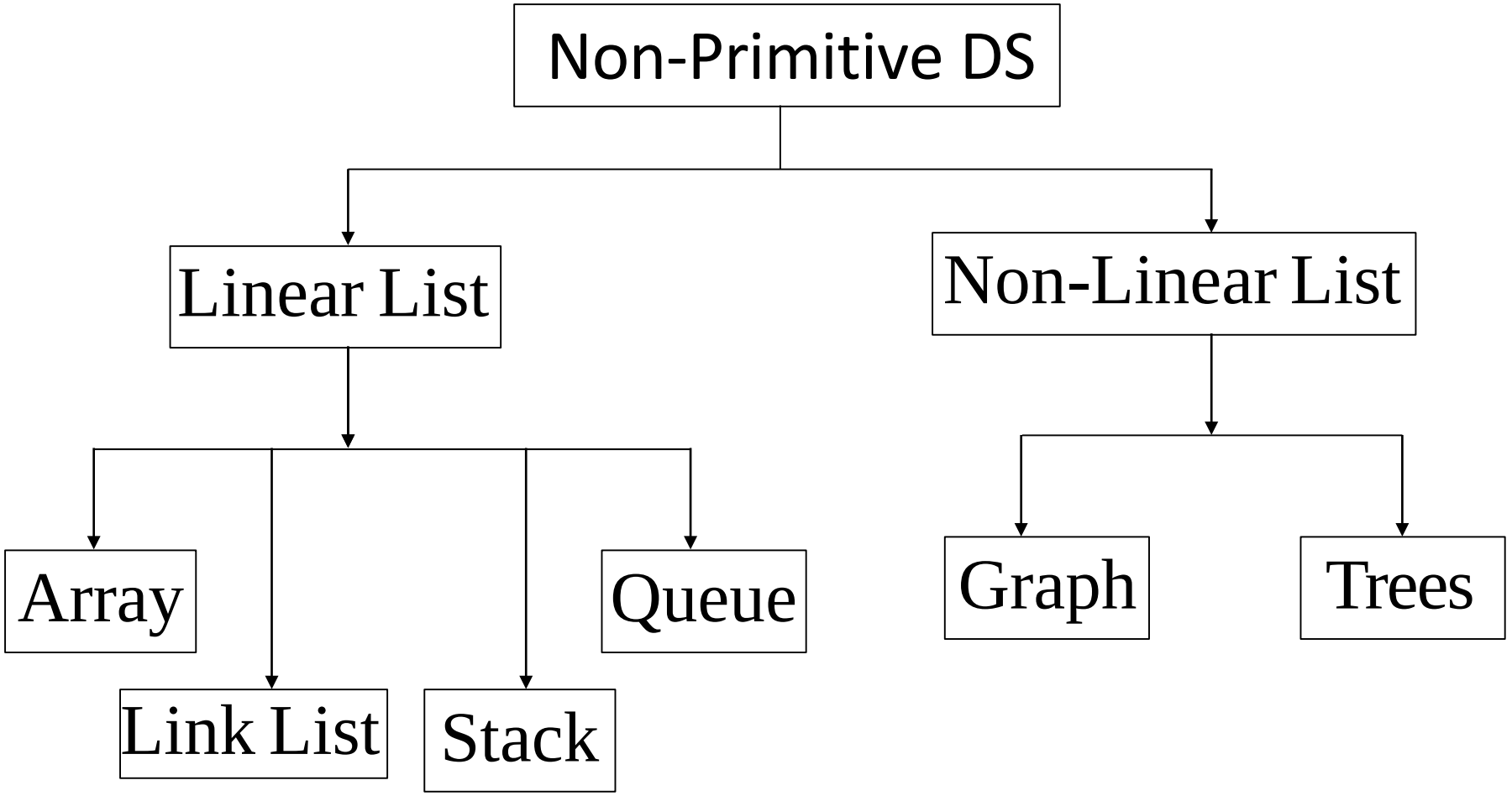
Introduction

- That means, algorithm is a set of instruction written to carry out certain tasks & the data structure is the way of organizing the data with their logical relationship retained.
- To develop a program of an algorithm, we should select an appropriate data structure for that algorithm.
- Therefore algorithm and its associated data structures from a program.

Classification of Data Structure



Classification of Data Structure



Primitive Data Structure

- There are basic structures and directly operated upon by the machine instructions.
- In general, there are different representation on different computers.
- Integer, Floating-point number, Character constants, string constants, pointers etc, fall in this category.

Non-Primitive Data Structure

- There are more sophisticated data structures.
- These are derived from the primitive data structures.
- The non-primitive data structures emphasize on structuring of a group of homogeneous (same type) or heterogeneous (different type) data items.

Non-Primitive Data Structure

The most commonly used operation on data structure are broadly categorized into following types:

- Create
- Selection
- Updating
- Searching
- Sorting
- Merging
- Destroy or Delete

Different between them

- A primitive data structure is generally a basic structure that is usually built into the language, such as an integer, a float.

- A non-primitive data structure is built out of primitive data structures linked together in meaningful ways, such as a or a linked-list, binary search tree,AVLTree, graph etc.

Linear and Non- Linear Data Structure

In a linear data structure, the data items are arranged in a linear sequence. Example is array.

In a non-Linear data structure, the data items are not in a sequence. Example is tree.

Homogeneous and Non- Homogenous Data Structure

In Homogeneous Structure, all the elements are of same type. Example is arrays.

In Non-homogeneous structure, the elements may or may not be of same type. Example is records.

Static and Dynamic Data Structure

Static structures are ones whose sizes and structures, associated memory location are fixed at compile time.

Dynamic structures are ones which expand or shrink as required during the program execution and there associated memory location change.

Data Structure Operations

There are six basic operations that can be performed on data structure:-

- Traversing
- Searching
- Sorting
- Inserting
- Deleting
- Merging

(a) Traversing

Traversing means accessing and processing each element in the data structure exactly once. This operation is used for counting the number of elements, printing the contents of the elements etc.

b) Searching

Searching is finding out the location of a given element from a set of numbers.

c) Sorting

Sorting is the process of arranging a list of elements in a sequential order.

The sequential order may be descending order or an ascending order according to the requirements of the data structure.

(d) Inserting

Inserting an element is adding an element in the data structure at any position. After insert operation the number of elements are increased by one.

e) Deleting

Deleting an element is removing an element in the data structure at any position. After deletion operation the number of elements are decreased by one.

(f) Merging

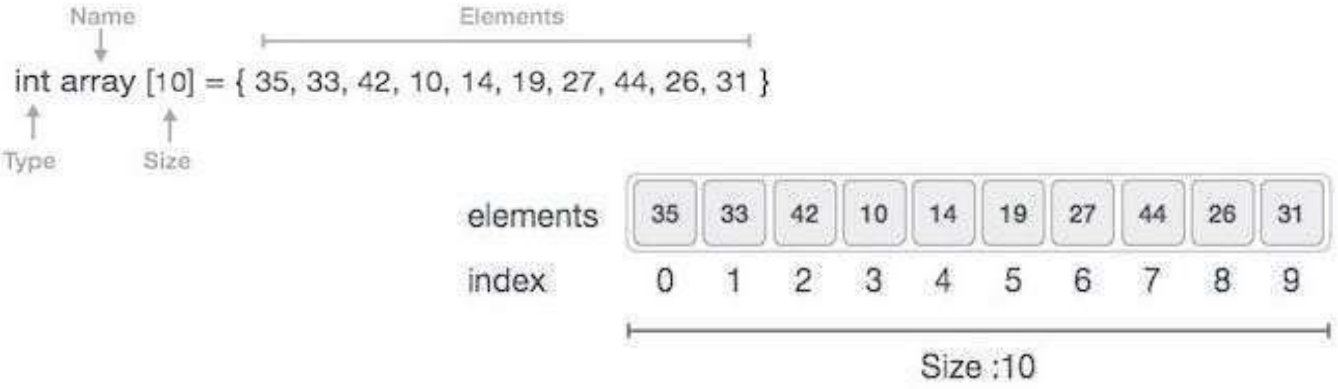
The process of combining the elements of two data structures into a single data structure is called merging.

DATA STRUCTURES AND APPLICATIONS

Arrays and Its Operation

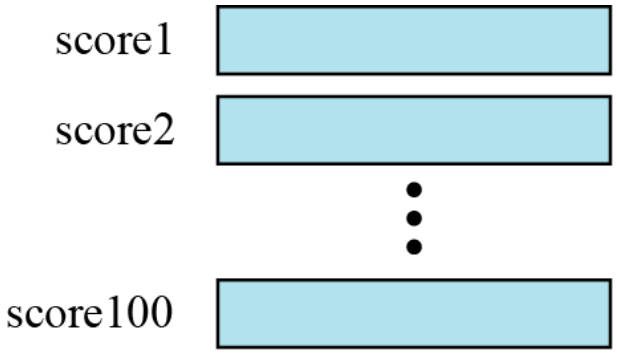
- **Array** is a container which can hold a fix number of items and these items should be of the same type. Most of the data structures make use of arrays to implement their algorithms. Following are the important terms to understand the concept of Array.
- **Element** − Each item stored in an array is called an element.
- **Index** − Each location of an element in an array has a numerical index, which is used to identify the element.
- Array Representation
- Arrays can be declared in various ways in different languages. For illustration, let's take C array declaration.

- Arrays can be declared in various ways in different languages. For illustration, let's take C array declaration.
- Arrays can be declared in various ways in different languages. For illustration, let's take C array declaration.
- As per the above illustration, following are the important points to be considered.
- Index starts with 0.
- Array length is 10 which means it can store 10 elements.
- Each element can be accessed via its index. For example, we can fetch an element at index 6 as 9.

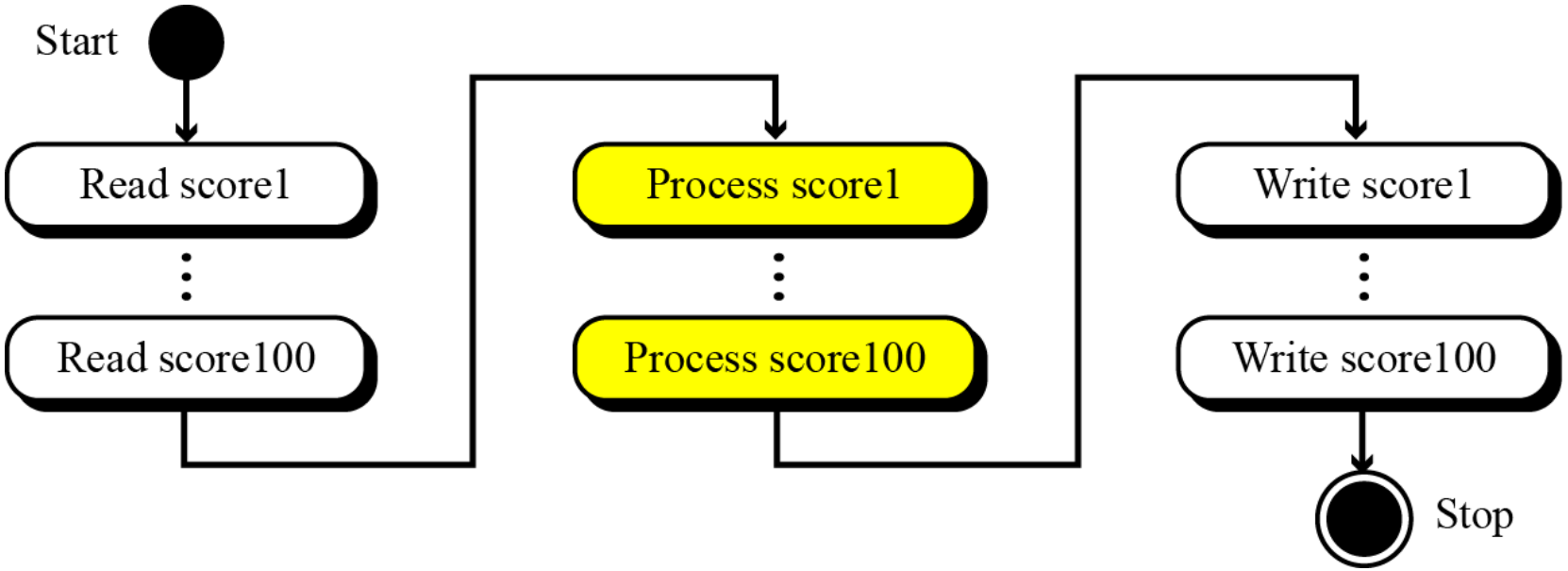


ARRAYS

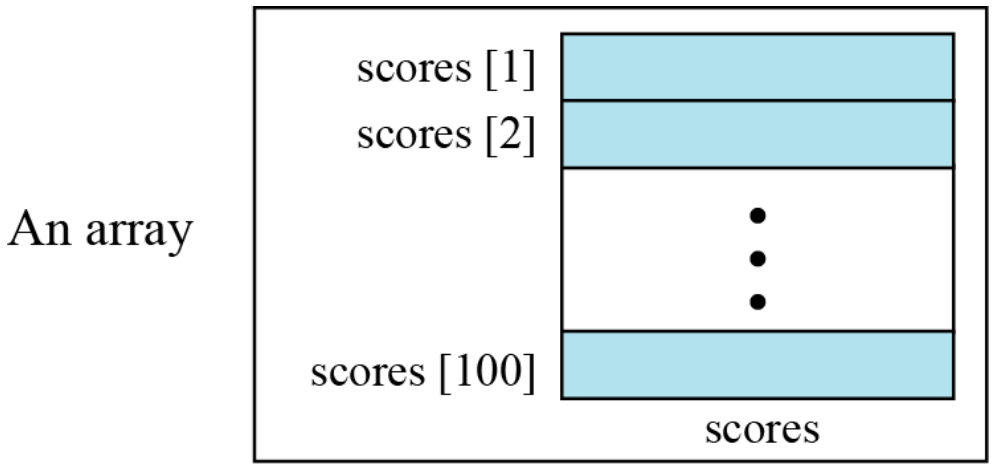
Imagine that we have 100 scores. We need to read them, process them and print them. We must also keep these 100 scores in memory for the duration of the program. We can define a hundred variables, each with a different name.



But having 100 different names creates other problems. We need 100 references to read them, 100 references to process them and 100 references to write them.



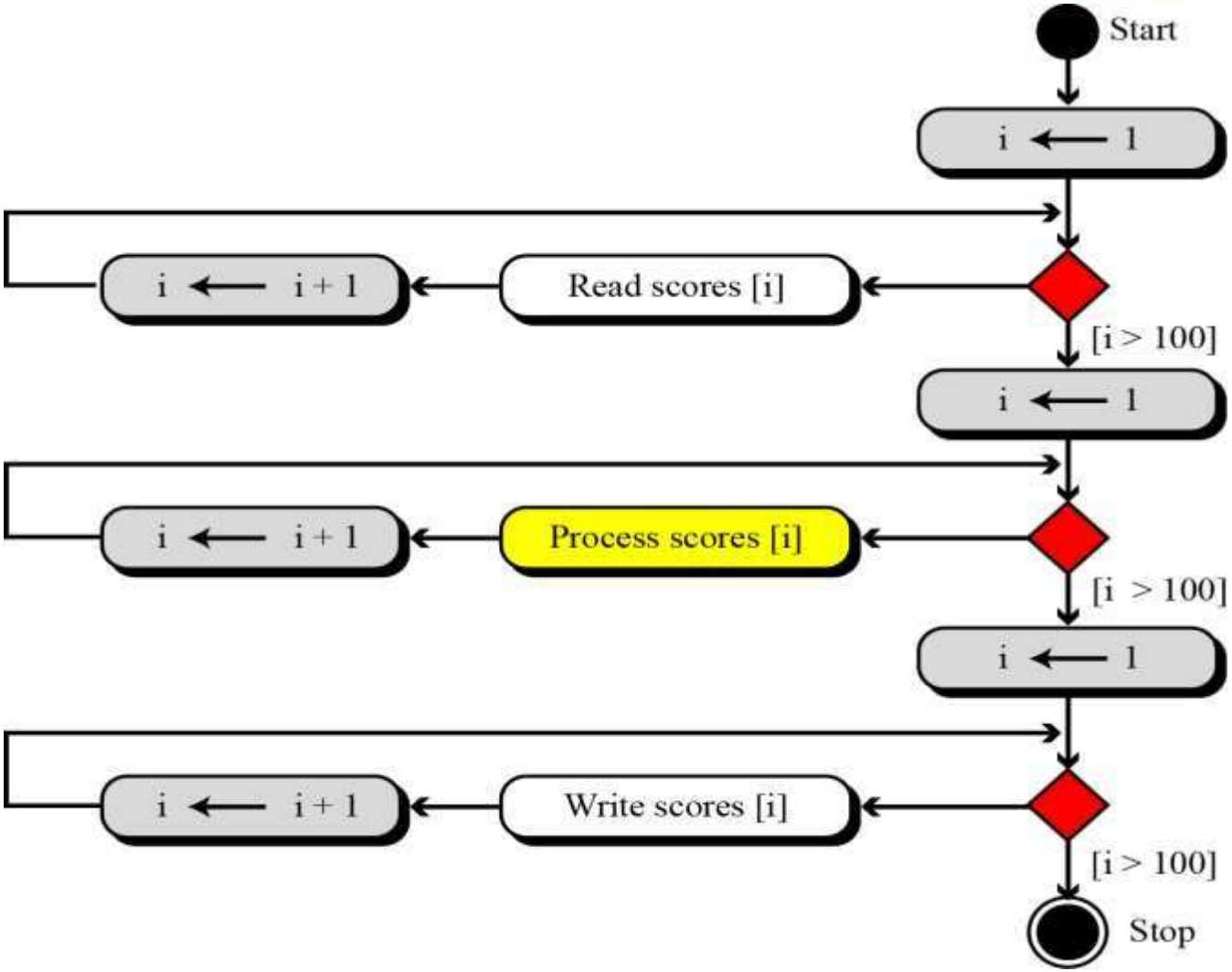
An array is a sequenced collection of elements, normally of the same data type, although some programming languages accept arrays in which elements are of different types. We can refer to the elements in the array as the first element, the second element and so forth, until we get to the last element.



Arrays with indexes

We can use loops to read and write the elements in an array. We can also use loops to process elements. Now it does not matter if there are 100, 1000 or 10,000 elements to be processed—loops make it easy to handle them all. We can use an integer variable to control the loop and remain in the loop as long as the value of this variable is less than the total number of elements in the array

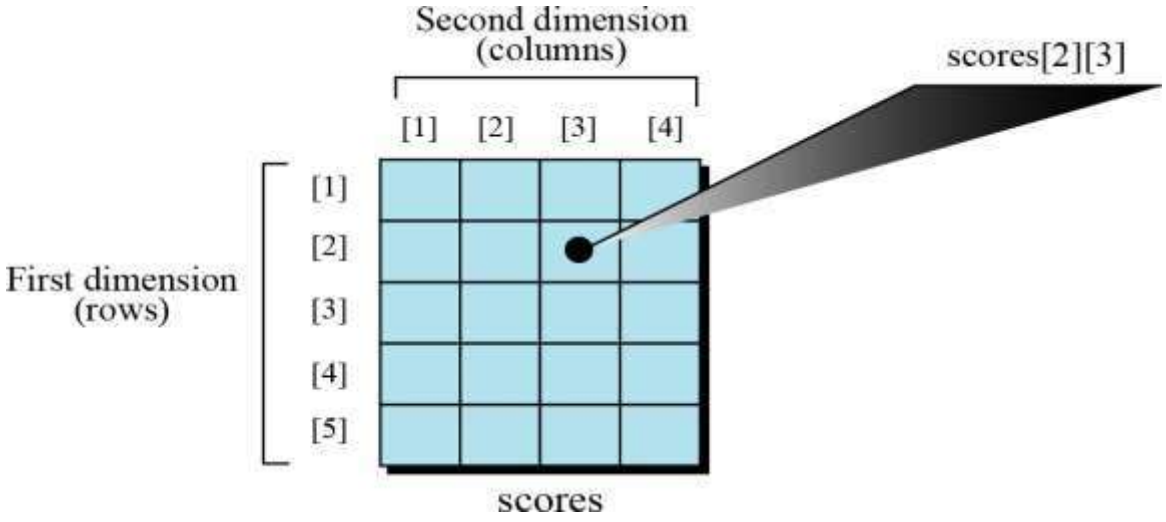
We have used indexes that start from 1;
some modern languages such as C,
C++ and Java start indexes from 0.



Processing an array

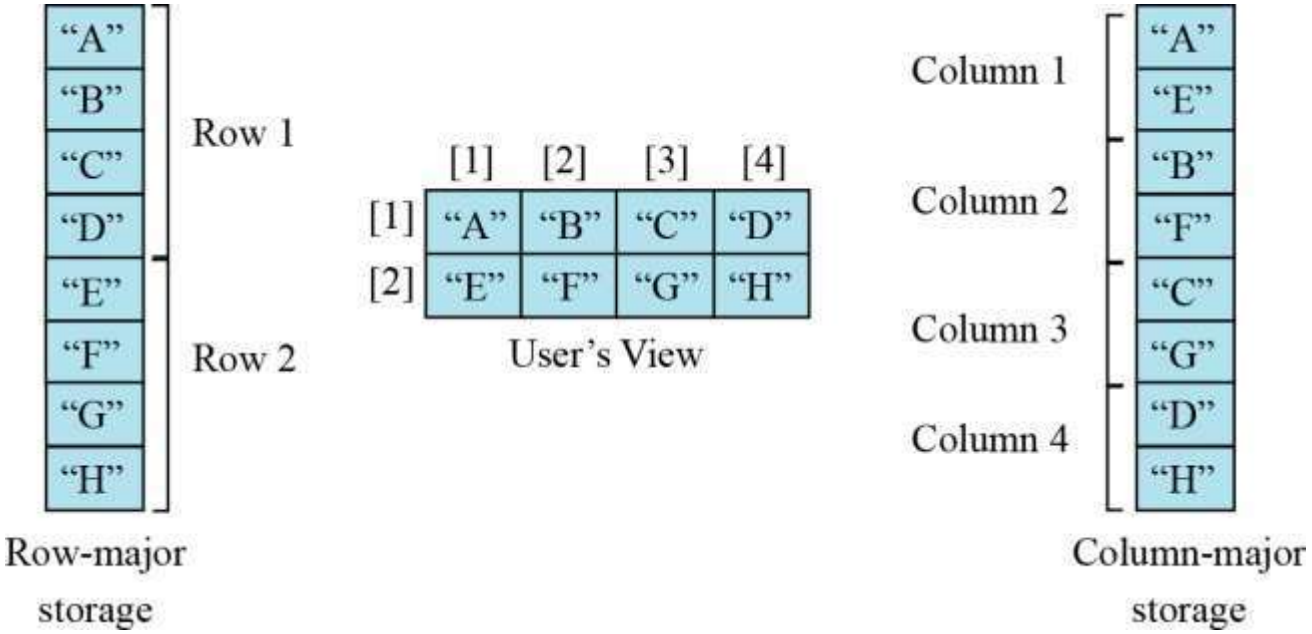
Multi-dimensional arrays

The arrays discussed so far are known as one-dimensional arrays because the data is organized linearly in only one direction. Many applications require that data be stored in more than one dimension. Figure shows a table, which is commonly called a two-dimensional array.



A two-dimensional array

The indexes in a one-dimensional array directly define the relative positions of the element in actual memory. Figure shows a two-dimensional array and how it is stored in memory using row-major or column-major storage. Row-major storage is more common.



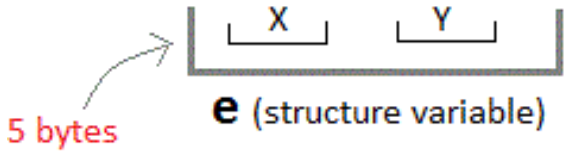
Memory layout of arrays

Data Structures and Applications Module 1

Structures , Unions and Pointers

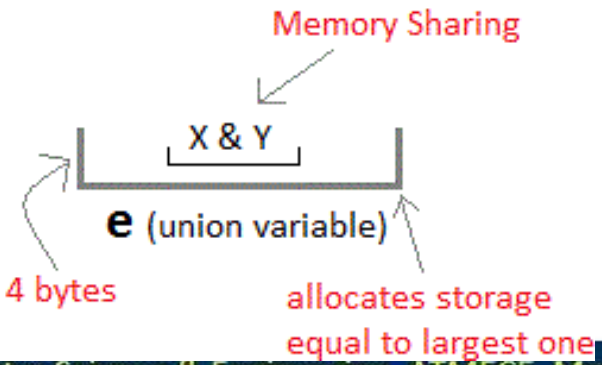
Structure

```
struct Emp
{
 char X ;     // size 1 byte
 float Y ;    // size 4 byte
} e ;
```



Unions

```
union Emp
{
 char X ;
 float Y ;
} e ;
```



Generic Pointers

type cast: tells the compiler to "change" an object's type (for type checking purposes – does not modify the object in any way)

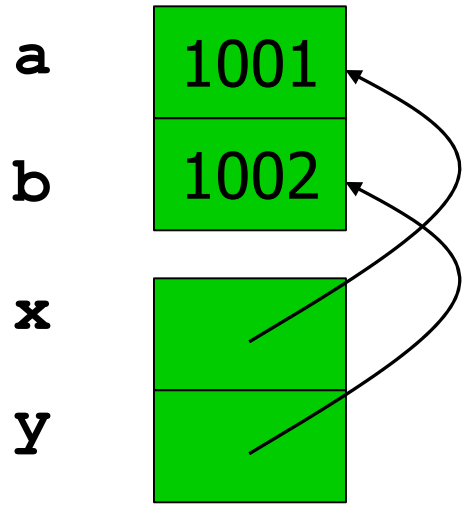
Dangerous! Sometimes necessary…

```
void   *p;
int     i;
char    c;
p = &i;
p = &c;
putchar(*(char *)p);
```

- void *: a "pointer to anything"
- Lose all information about what type of thing is pointed to
  - Reduces effectiveness of compiler's type-checking
  - Can't use pointer arithmetic

Pass-by-Reference

```
void
set_x_and_y(int *x, int *y)
{

    *x = 1001;
    *y = 1002;
}

void
f(void)
{
    int a = 1;
    int b = 2;

    set_x_and_y(&a, &b);
}
```



Arrays and Pointers

- Array name ≈ a to the initial (0th) array element

```
a[i]  ≡  *(a + i)
```

- An array is passed to a function as a pointer
  - The array size is lost!

- Usually bad style to interchange arrays and pointers
  - Avoid pointer arithmetic!

Passing arrays:

Really `int *array`

Must explicitly pass the size

```
int
foo(int array[],
    unsigned int size)
{
   … array[size - 1] …
}

int
main(void)
{
   int a[10], b[5];
   … foo(a, 10)… foo(b, 5) …
}
```

Arrays and Pointers

```
int
foo(int array[],
    unsigned int size)
{
   …
   printf("%d\n", sizeof(array));
}

int
main(void)
{
   int a[10], b[5];
   … foo(a, 10)… foo(b, 5) …
   printf("%d\n", sizeof(a));
}
```

What does this print? **8**

… because **`array`** is really a pointer

What does this print? **40**

Arrays and Pointers

```
int  i;
int  array[10];

for (i = 0; i < 10; i++) {
  …
  array[i] = …;
  …
}
```

```
int *p;
int  array[10];

for (p = array; p < &array[10]; p++) {
  …
  *p = …;
  …
}
```

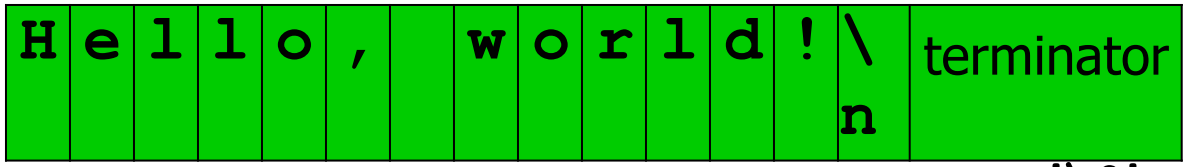
These two blocks of code are functionally equivalent

Strings

- In C, strings are just an array of characters
  - Terminated with '\0' character
  - Arrays for bounded-length strings
  - Pointer for constant strings (or unknown length)

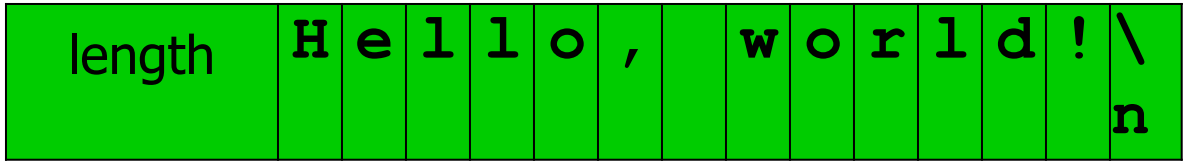
```
char  str1[15] = "Hello, world!\n";
char *str2     = "Hello, world!\n";
```

C, …



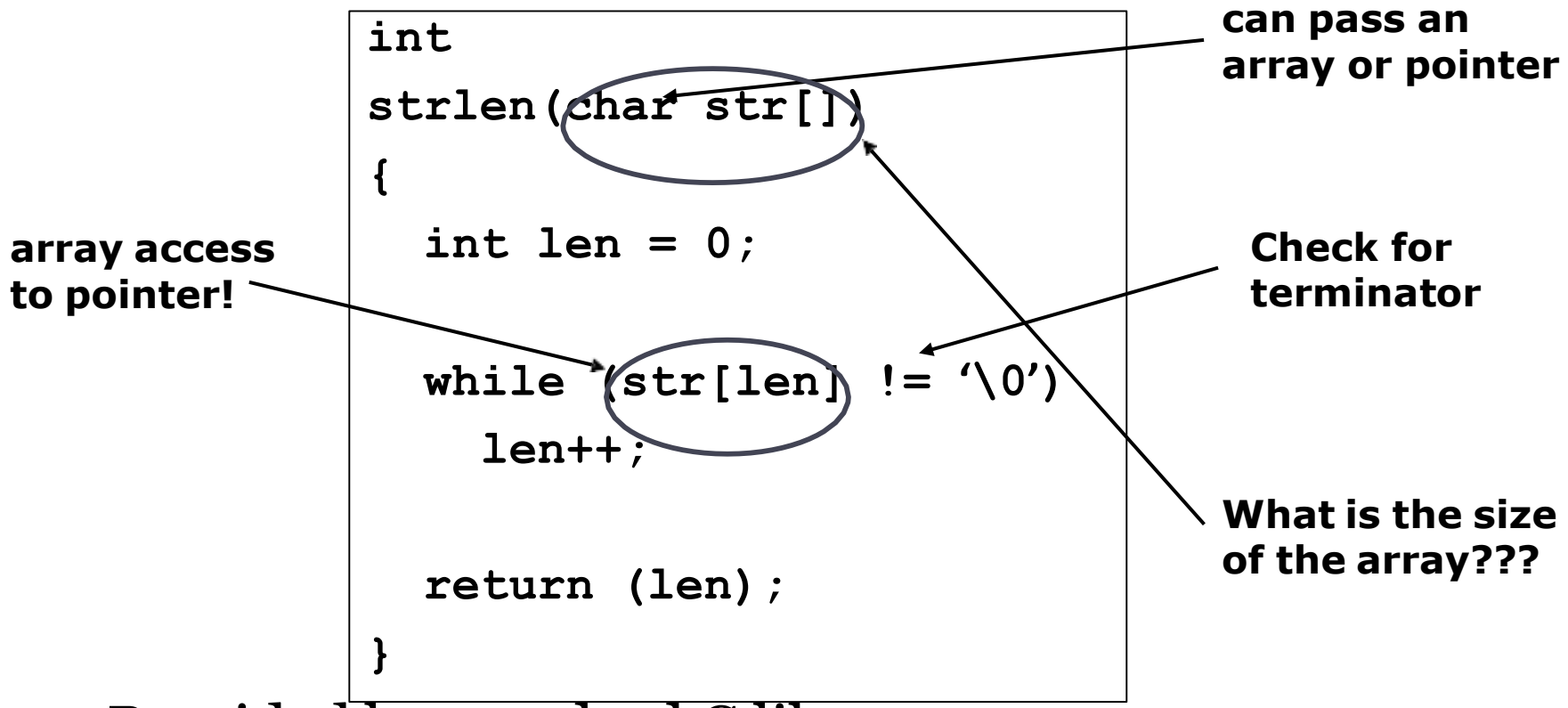
C terminator: '\0'

Pascal, Java, …



String length

- Must calculate length:



- Provided by standard C library: `#include <string.h>`

Pointer to Pointer (char **argv)

Passing arguments to main:

```
int
main(int argc, char **argv)
{
   ...
}
```

size of the argv array/vector

an array/vector of char *

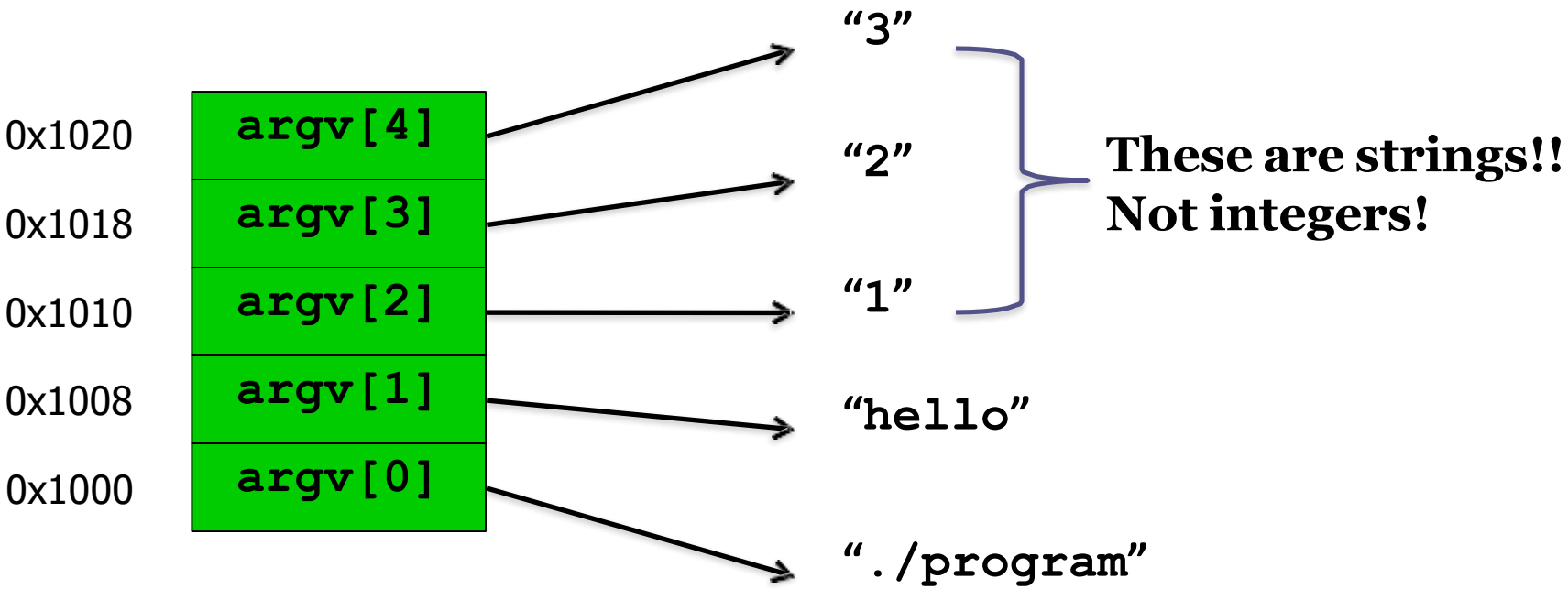
Recall when passing an array, a pointer to the first element is passed

Suppose you run the program this way

```
UNIX% ./program hello 1 2 3
```

`argc == 5` **(five strings on the command line)**

char **argv


0x1020
0x1018
0x1010
0x1008
0x1000
argv[4]
argv[3]
argv[2]
argv[1]
argv[0]
"3"
"2"
"1"
"hello"
"./program"
These are strings!!
Not integers!

Structures and Unions in C
Objectives

- Be able to use compound data structures in programs
- Be able to pass compound data structures as function arguments, either by value or by reference
- Be able to do simple bit-vector manipulations

Structures

- Compound data:
- A date is
  - an `int month` and
  - an `int day` and
  - an `int year`

```
struct ADate {
   int  month;
   int  day;
   int  year;
};

struct ADate date;

date.month = 1;
date.day = 18;
date.year = 2018;
```

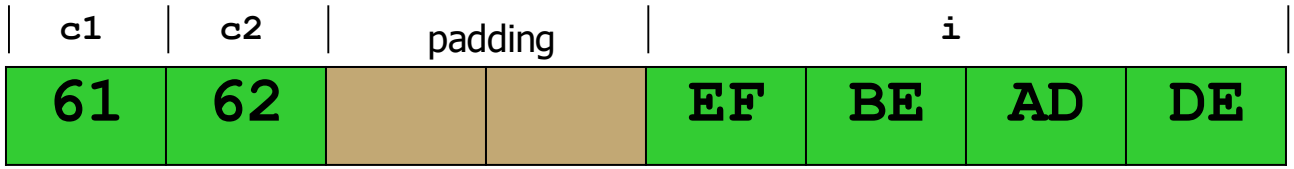
- Unlike Java, C doesn't automatically define functions for initializing and printing …

Structure Representation & Size

- `sizeof(struct …)` =
- sum of `sizeof(field)`
- alignment padding

+ Processor- and compiler-specific

```
struct CharCharInt {
    cha   c1;
    r     c2;
    cha   i;
    r
    int
} foo;

foo.c1 = 'a';
foo.c2 = 'b';
foo.i  = 0xDEADBEEF;
```



x86 uses “little-endian” representation

Typedef

- Mechanism for creating new type names
  - New names are an alias for some other type
  - *May* improve clarity and/or portability of the program

```
typedef long int64_t;
typedef struct ADate {
    int month;
    int day;
    int year;
} Date;

int64_t i = 100000000000;
Date d = { 1, 18, 2018 };
```

Overload existing type names for clarity and portability

Simplify complex type names

Constants

- Allow consistent use of the same constant throughout the program
  - Improves clarity of the program
  - Reduces likelihood of simple errors
  - Easier to update constants in the program

```
int array[10];

for (i=0; i<10; i++) {
   …
}
```

Preprocessor directive

Constant names are capitalized by convention

```
#define SIZE 10

int array[SIZE];

for (i=0; i<SIZE; i++) {
   …
}
```

Define once, use throughout the program

Arrays of Structures

Array declaration

Constant

```
Date birthdays[NFRIENDS];

bool
check_birthday(Date today)
{
  int i;

  for (i = 0; i < NFRIENDS; i++) {
    if ((today.month == birthdays[i].month) &&
        (today.day == birthdays[i].day))
      return (true);

  return (false);
}
```

Array index, then structure field

Pointers to Structures

```
Date
create_date1(int month,
             int day,
             int year)
{
  Date d;

  d.month = month;
  d.day   = day;
  d.year  = year;

  return (d);
}
```

Copies date

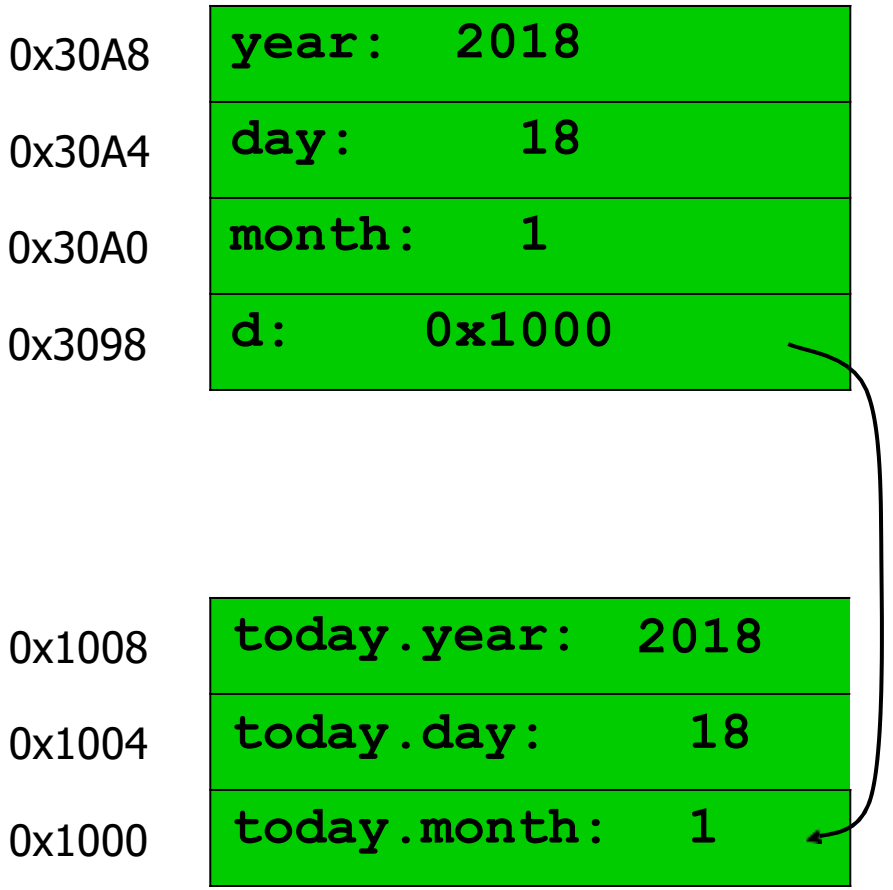
Pass-by-reference

```
void
create_date2(Date *d,
             int month,
             int day,
             int year)
{
  d->month = month;
  d->day   = day;
  d->year  = year;
}
```

```
Date today;

today = create_date1(1, 18, 2018);
create_date2(&today, 1, 18, 2018);
```

Pointers to Structures (cont.)

```
void
create_date2(Date *d,
             int month,
             int day,
             int year)
{
  d->month = month;
  d->day   = day;
  d->year  = year;
}

void
fun_with_dates(void)
{
  Date today;
  create_date2(&today, 1, 18, 2018);
}
```



Pointers to Structures (cont.)

```
Date *
create_date3(int month,
             int day,
             int year)
{
  Date *d;

  d->month = month;
  d->day   = day;
  d->year  = year;

  return (d);
}
```

What is d pointing to?!?!
(more on this later)

Unions

- Up to programmer to determine how to interpret a union (i.e. which member to access)
- Often used in conjunction with a "type" variable that indicates how to interpret the union value

```
enum TYPE { INT, FLOAT, STRING };
struct VARIABLE {
  enum TYPE type;
  union VALUE value;
};
```

Access `type` to determine how to interpret `value`

Unions

- Storage
  - size of union is the size of its largest member
  - avoid unions with widely varying member sizes; for the larger data types, consider using pointers instead
- Initialization
  - Union may only be initialized to a value appropriate for the type of its first member

Structures

- Reminder… the C struct declaration creates a data type that groups objects of possibly different types into a single object
- Implementation similar to arrays
  - All components are stored in a contiguous region of memory
  - A pointer to a structure is the address of its first byte
- The compiler maintains information about each structure type indicating the byte offset of each field
  - Generates references to structure elements using these offsets as displacements in memory referencing instructions

Structure allocation

Concept

- Contiguously-allocated region of memory
- Refer to members within structure by names
- Members may be of different types

```
struct rec {
  int i;
  int a[3];
  int *p;
};
```

Memory Layout

Accessing Structure Member

```
void
set_i(struct rec *r,
      int val)
{
  r->i = val;
}
```

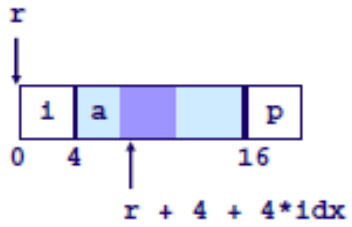
Assembly

```
# %eax = val
# %edx = r
movl %eax,(%edx)    # Mem[r] = val
```

Structure Access

Generating Ptr to Structure Member

```
struct rec {
  int i;
  int a[3];
  int *p;
};
```



Generating Pointer to Array Element

- Offset of each structure member determined at compile time

```
int *
find_a
 (struct rec *r, int idx)
{
  return &r->a[idx];
}
```

```
# %ecx = idx
# %edx = r
leal 0(,%ecx,4),%eax    # 4*idx
leal 4(%eax,%edx),%eax # r+4*idx+4
```

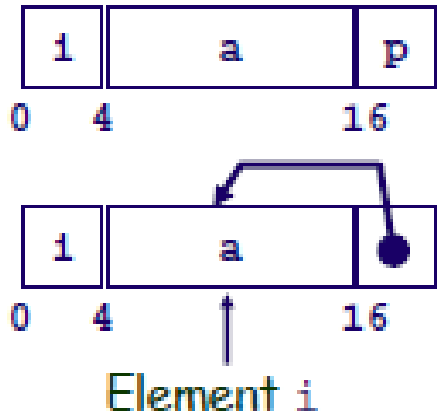
```
find_a:
  pushl   %ebp
  movl    %esp, %ebp
  movl    12(%ebp), %eax      // idx (2nd arg)
  sall    $2, %eax            // mult by 4
  addl    8(%ebp), %eax       // ptr to struct (1st arg)
  addl    $4, %eax
  popl    %ebp
  ret
```

Structure referencing (cont)

C Code

```
struct rec {
  int i;
  int a[3];
  int *p;
};
```

```
void
set_p(struct rec *r)
{
  r->p =
   &r->a[r->i];
}
```



"i" represents the element of "a" that I want "p" to point to

```
# %edx = r
movl (%edx),%ecx        # r->i
leal 0(,%ecx,4),%eax    # 4*(r->i)
leal 4(%edx,%eax),%eax  # r+4+4*(r->i)
movl %eax,16(%edx)      # Update r->p
```

Structures and Unions

- Structures (records)
  - Arrays are collections of data of the same type. In C there is an alternate way of grouping data that permit the data to vary in type.
    - This mechanism is called the **struct**, short for structure.
  - A structure is a collection of data items, where each item is identified as to its type and name.

```
struct {
        char name[10];
        int age;
        float salary;
        } person;
```

```
strcpy(person.name,"james");
person.age = 10;
person.salary = 35000;
```

Structures and Unions

- We can also embed a structure within a structure.

```
typedef struct {
        int month;
        int day;
        int year;
        } date;

typedef struct human_being {
        char name[10];
        int age;
        float salary;
        date dob;
        };
```

- A person born on February 11, 1994, would have have values for the *date* **struct** set as

```
person1.dob.month = 2;
person1.dob.day = 11;
person1.dob.year = 1944;
```

Structures and Unions

- A **union** declaration is similar to a structure.
- The fields of a **union** must share their memory space.
- Only one field of the **union** is "active" at any given time
  - Example: Add fields for male and female.

```
typedef struct sex_type {
   enum tag_field {female, male} sex;
   union {
      int children;
      int beard ;
      } u;
   };
typedef struct human_being {
   char name[10];
   int age;
   float salary;
   date dob;
   sex_type sex_info;
   };
human_being person1, person2;
```

person1.sex_info.sex = male;
person1.sex_info.u.beard = FALSE;
and
person2.sex_info.sex = female;
person2.sex_info.u.children = 4;

Structures and Unions

- Internal implementation of structures
  - The fields of a structure in memory will be stored in the same way using increasing address locations in the order specified in the structure definition.
  - Holes or padding may actually occur
    - Within a structure to permit two consecutive components to be properly aligned within memory
  - The size of an object of a struct or union type is the amount of storage necessary to represent the largest component, including any padding that may be required.

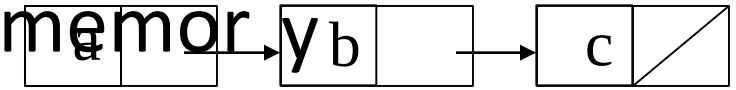
Self Referential Structures

- typedef struct list {
  char data;
  list *link;
  }

- list item1, item2, item3;
  item1.data='a';
  item2.data='b';
  item3.data='c';
  item1.link=item2.link=item3.link=NULL;

three nodes
item1.link=&item2;
item2.link=&item3;

- malloc: obtain a node (memory) free: release memor y



DATA STRUCTURES AND APPLICATIONS

Dynamic Memory Allocation

Problem with Arrays

- Sometimes
  - Amount of data cannot be predicted beforehand
  - Number of data items keeps changing during program execution
- Example: Search for an element in an array of N elements
- One solution: find the maximum possible value of N and allocate an array of N elements
  - Wasteful of memory space, as N may be much smaller in some executions
  - Example: maximum value of N may be 10,000, but a particular run may need to search only among 100 elements
    - Using array of size 10,000 always wastes memory in most cases

Better Solution

- Dynamic memory allocation
  - Know how much memory is needed after the program is run
    - Example: ask the user to enter from keyboard
  - Dynamically allocate only the amount of memory needed
- C provides functions to dynamically allocate memory
  - malloc, calloc, realloc

Memory Allocation Functions

- malloc
  - Allocates requested number of bytes and returns a pointer to the first byte of the allocated space
- calloc
  - Allocates space for an array of elements, initializes them to zero and then returns a pointer to the memory.
- free
  - Frees previously allocated space.
- realloc
  - Modifies the size of previously allocated space.
- We will only do malloc and free

Allocating a Block of Memory

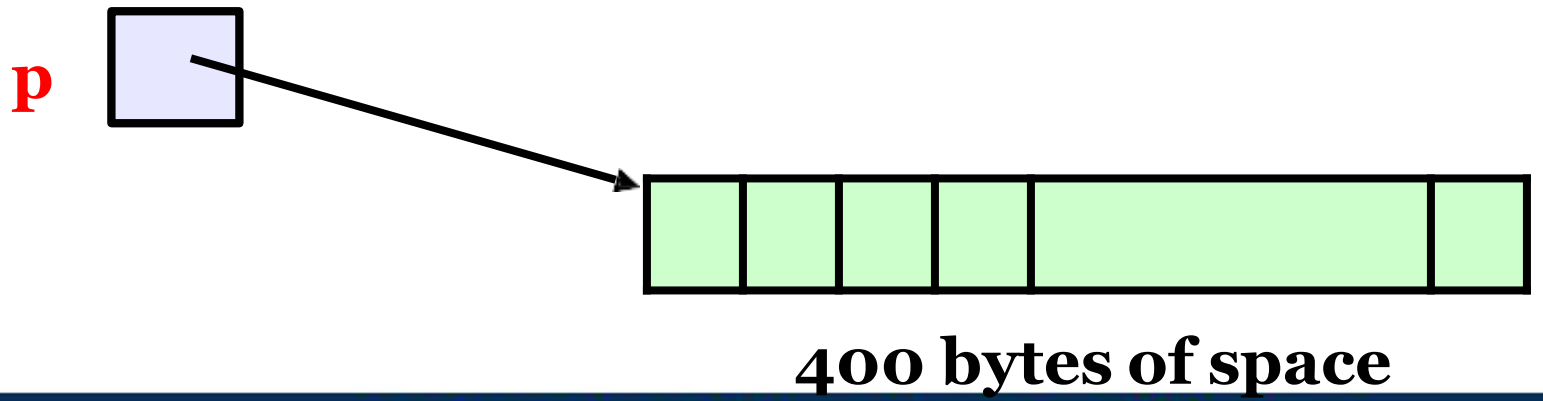
- A block of memory can be allocated using the function malloc
  - Reserves a block of memory of specified size and returns a pointer of type void
  - The return pointer can be type-casted to any pointer type
- General format:

```
type *p;
   p =  (type *) malloc (byte_size);
```

Example

```
p = (int *) malloc(100 * sizeof(int));
```

- A memory space equivalent to 100 times the size of an int bytes is reserved
- The address of the first byte of the allocated memory is assigned to the pointer p of type int



Contd.

- cptr = (char *) malloc (20);

  Allocates 20 bytes of space for the pointer cptr of type char

- sptr = (struct stud *) malloc(10*sizeof(struct stud));

  Allocates space for a structure array of 10 elements. sptr points to a structure element of type struct stud

Always use sizeof operator to find number of bytes for a data type, as it can vary from machine to machine

Points to Note

- malloc always allocates a block of contiguous bytes
  - The allocation can fail if sufficient contiguous memory space is not available
  - If it fails, malloc returns NULL

```
if  ((p = (int *) malloc(100 * sizeof(int))) == NULL)
  {
        printf ("\n Memory cannot be allocated");
        exit();
   }
```

Using the malloc'd Array

- Once the memory is allocated, it can be used with pointers, or with array notation
- Example:

```
int *p, n, i;
scanf("%d", &n);
p = (int *) malloc (n * sizeof(int));
for (i=0; i<n; ++i)
    scanf("%d", &p[i]);
```

The n integers allocated can be accessed as *p, *(p+1), *(p+2),…, *(p+n-1) or just as p[0], p[1], p[2], …,p[n-1]

Example

```
int main()
{
  int i,N;
  float *height;
  float sum=0,avg;

  printf("Input no. of students\n");
  scanf("%d", &N);

  height = (float *)
       malloc(N * sizeof(float));
```

```
printf("Input heights for %d
students \n",N);
  for (i=0; i<N; i++)
   scanf ("%f", &height[i]);

  for(i=0;i<N;i++)
    sum += height[i];

  avg = sum / (float) N;

  printf("Average height = %f \n",
                  avg);
  free (height);
  return 0;
}
```

Releasing the Allocated Space: free

- An allocated block can be returned to the system for future use by using the free function
- General syntax:

  free (ptr);

  where ptr is a pointer to a memory block which has been previously created using malloc
- Note that no size needs to be mentioned for the allocated block, the system remembers it for each pointer returned

Can we allocate only arrays?

- malloc can be used to allocate memory for single variables also
  - p = (int *) malloc (sizeof(int));
  - Allocates space for a single int, which can be accessed as *p
- Single variable allocations are just special case of array allocations
  - Array with only one element

malloc( )-ing array of structures

```
typedef struct{
        char name[20];
        int roll;
        float SGPA[8], CGPA;
    } person;
void main()
{
    person *student;
    int i,j,n;
    scanf("%d", &n);
    student = (person *)malloc(n*sizeof(person));
    for (i=0; i<n; i++) {
       scanf("%s", student[i].name);
       scanf("%d", &student[i].roll);
       for(j=0;j<8;j++) scanf("%f", &student[i].SGPA[j]);
       scanf("%f", &student[i].CGPA);
    }
}
```

Static array of pointers

```
#define N  20
#define M  10
int main()
{
    char word[N], *w[M];
    int i, n;
    scanf("%d",&n);
    for (i=0; i<n; ++i) {
        scanf("%s", word);
        w[i] = (char *) malloc ((strlen(word)+1)*sizeof(char));
        strcpy (w[i], word) ;
    }
    for (i=0; i<n; i++) printf("w[%d] = %s \n",i,w[i]);
    return 0;
}
```

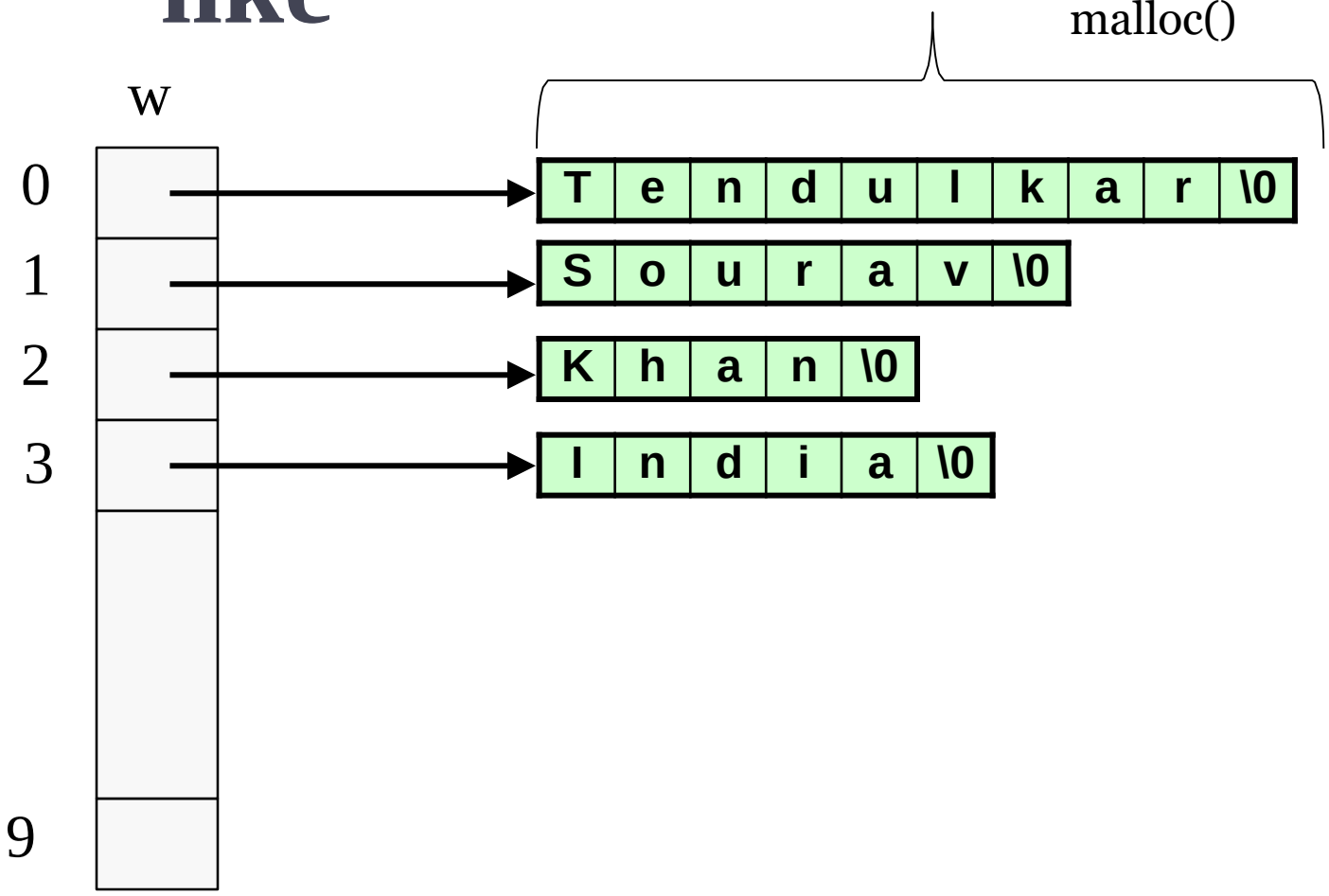
Static array of pointers

```
#define N   20
#define M  10
int main()
{
    char word[N], *w[M];
    int i, n;
    scanf("%d",&n);
    for (i=0; i<n; ++i) {
        scanf("%s", word);
        w[i] = (char *) malloc ((strlen(word)+1)*sizeof(ch
        strcpy (w[i], word) ;
    }
    for (i=0; i<n; i++) printf("w[%d] = %s \n",i,w[i]);
    return 0;
}
```

Output

```
4
Tendulkar
Sourav
Khan
India
w[0] = Tendulkar
w[1] = Sourav
w[2] = Khan
w[3] = India
```

How it will look like



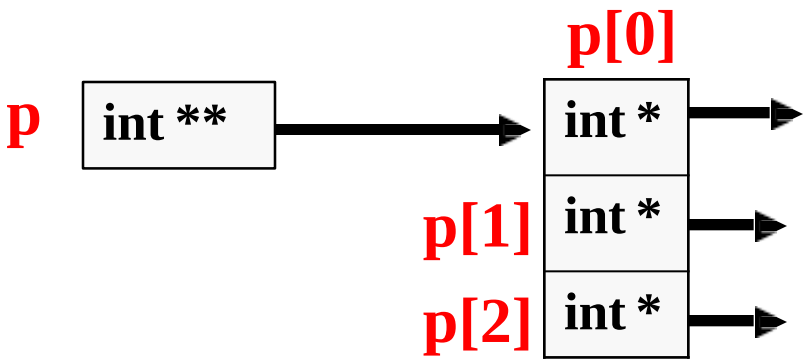
Pointers to Pointers

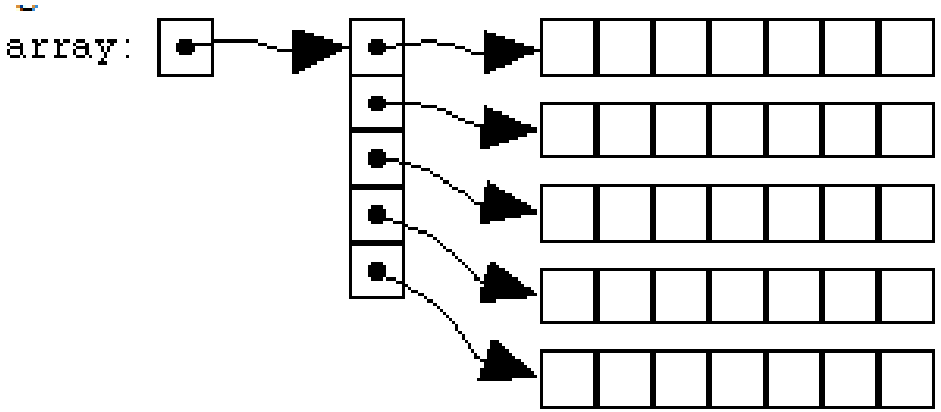
- Pointers are also variables (storing addresses), so they have a memory location, so they also have an address
- Pointer to pointer – stores the address of a pointer variable

```
int x = 10, *p, **q;
p = &x;
q = &p;
printf("%d  %d  %d", x, *p, *(*q));

will print 10  10  10 (since *q = p)
```

Allocating Pointer to Pointer

```
int **p;
p = (int **) malloc(3 * sizeof(int *));
```



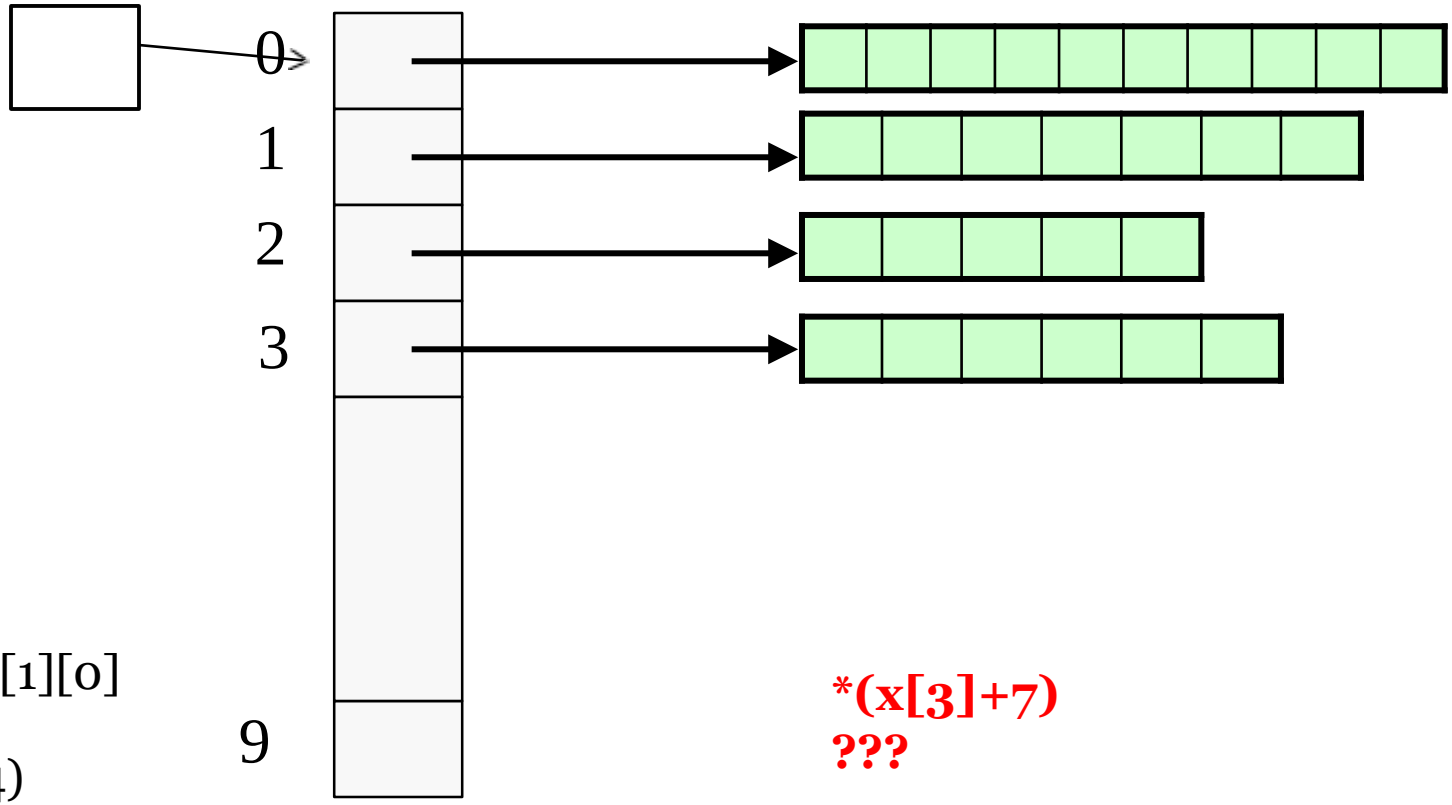


2D array

```
#include <stdlib.h>

int main()
{
        int **array;
       array = (int**) malloc(nrows * sizeof(int *));

       for(i = 0; i < nrows; i++)
      {

          array[i] = (int*)malloc(ncolumns *
sizeof(int));

      }

{
```

2D array

```
Int main()
{

        x

    *x;

    *(x+1);

    **(x+1);  x[1][0]

    *(*(x+1)+4)
}                   x[1][4]

    *((*x)+8)   x[0][8]
```



*(x[3]+7)
???

Dynamic Allocation of 2-d Arrays

- Recall that address of [i][j]-th element is found by first finding the address of first element of i-th row, then adding j to it
- Now think of a 2-d array of dimension [M][N] as M 1-d arrays, each with N elements, such that the starting address of the M arrays are contiguous (so the starting address of k-th row can be found by adding 1 to the starting address of (k-1)-th row)
- This is done by allocating an array p of M pointers, the pointer p[k] to store the starting address of the k-th row

Contd.

- Now, allocate the M arrays, each of N elements, with p[k] holding the pointer for the k-th row array
- Now p can be subscripted and used as a 2-d array
- Address of p[i][j] = *(p+i) + j (note that *(p+i) is a pointer itself, and p is a pointer to a pointer)

Dynamic Allocation of 2-d Arrays

```
int **allocate (int h, int w)
  {
    int **p;
    int i, j;

    p = (int **) malloc(h*sizeof (int *) );
    for (i=0;i<h;i++)
      p[i] = (int *) malloc(w * sizeof (int));
    return(p);
  }
```

Allocate array of pointers

Allocate array of integers for each row

```
• void read_data (int **p, int h, int w
)  {
•  int i, j;
   • for (i=0;i<h;i++)
     for (j=0;j<w;j++)
   •  scanf ("%d", &p[i][j]);
•  }
```

•Elements accessed like 2-D array elements.

Contd.

```
void print_data (int **p, int h, int w)
 {
    int i, j;
     for (i=0;i<h;i++)
     {
     for (j=0;j<w;j++)
       printf ("%5d ", p[i][j]);
      printf ("\n");
     }
}
```

```
int main()
{
  int **p;
  int M, N;
  printf ("Give M and N \n");
  scanf ("%d%d", &M, &N);
  p = allocate (M, N);
  read_data (p, M, N);
  printf ("\nThe array read as \n");
  print_data (p, M, N);
  return 0;
}
```

Contd.

```
void print_data (int **p, int h, int w)
 {
    int i, j;
     for (i=0;i<h;i++)
     {
     for (j=0;j<w;j++)
       printf ("%5d ", p[i][j]);
      printf ("\n");
     }
}
```

```
Give M and N
3 3
1 2 3
4 5 6
7 8 9
 The array read as
    1    2    3
    4    5    6
    7    8    9
```

```
int main()
{
  int **p;
  int M, N;
  printf ("Give M and N \n");
  scanf ("%d%d", &M, &N);
  p = allocate (M, N);
  read_data (p, M, N);
  printf ("\nThe array read as \n");
  print_data (p, M, N);
  return 0;
}
```

Memory Layout in Dynamic Allocation

```
int main()
{
int **p;
int M, N;
printf ("Give M and N \n"); scanf ("%d%d", &M, &N); p = allocate (M, N);
for (i=0;i<M;i++) {
  for (j=0;j<N;j++)
    printf ("%10d", &p[i][j]);
  printf("\n");
}
return 0;
}
```

```
int **allocate (int h, int w)
{
    int **p;
    int i, j;

    p = (int **)malloc(h*sizeof (int *));
    for (i=0; i<h; i++)
    printf("%10d", &p[i]);
    printf("\n\n");
    for (i=0;i<h;i++)
      p[i] = (int *)malloc(w*sizeof(int));
    return(p);
}
```

Output

```
3 3
31535120  31535128  31535136

31535152  31535156  31535160
31535184  31535188  31535192
31535216  31535220  31535224
```

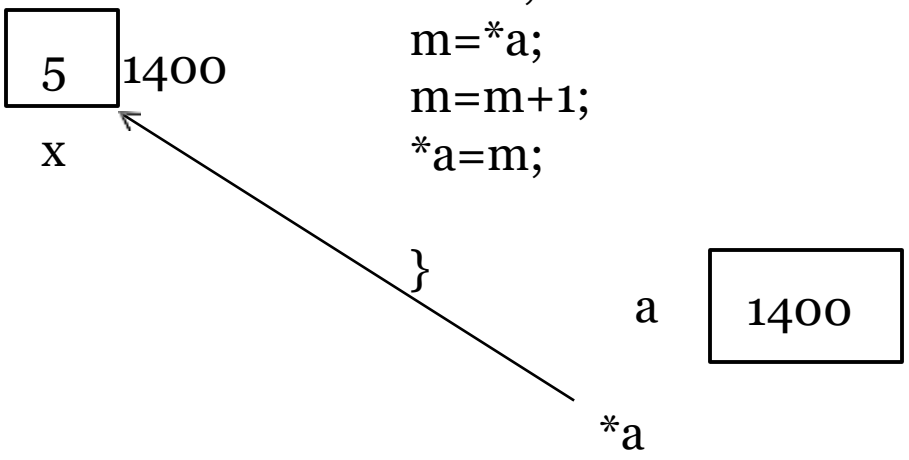
Starting address of each row, contiguous (pointers are 8 bytes long)

Elements in each row are contiguous

Passing pointers to function

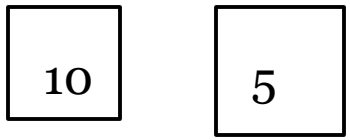
```
Int main()
{
        int x=5;
.....
.....
foo(&x);

}
```

```
void foo(int *a)
{

int m;
m=*a;
m=m+1;
*a=m;

}
```

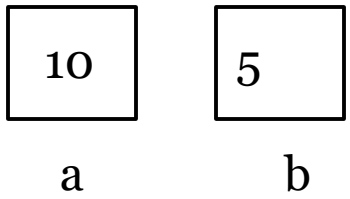


Call by value

```
Int main()
{
    int x=10, y=5;
    swap(x,y);


}
```



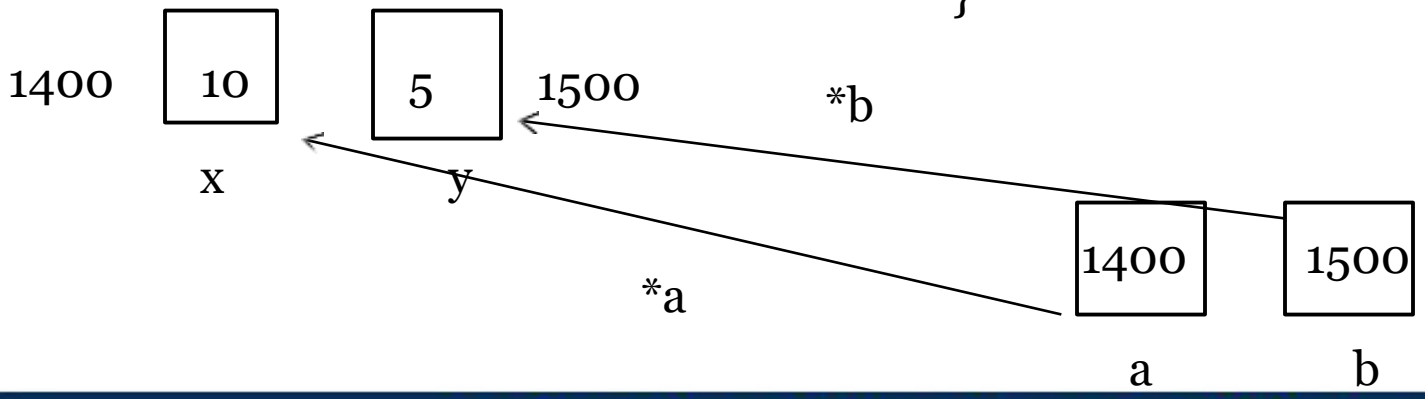

```
swap(int a, int b)
{

    int temp;
    temp=a;
    a=b;
    b=temp;

}
```



Call by Reference

```
Int main()
{
    int x=10, y=5;
    swap(&x,&y);


}
```

```
swap(int *a, int *b)
{

   int temp;
   temp=*a;
   *a=*b;
   *b=temp;

}
```



DATA STRUCTURES AND APPLICATIONS

Arrays- Representation of Linear Arrays in Memory

Overview

- An array
  - a single name for a collection of data values
  - all of the same data type
  - subscript notation to identify one of the values
- A carryover from earlier programming languages
- More than a primitive type, less than an object
  - like objects when used as method parameters and return types
  - do not have or use inheritance
- Accessing each of the values in an array
  - Usually a `for` loop

Creating Arrays

- General syntax for declaring an array:

```
Base_Type[] Array_Name = new Base_Type[Length];
```

- Examples:
  80-element array with base type `char`:

```
char[] symbol = new char[80];
```

  100-element array of `doubles`:

```
double[] reading = new double[100];
```

  70-element array of `Species`:

```
Species[] specimen = new Species[70];
```

Three Ways to Use [ ] (Brackets) with an Array Name

1. Declaring an array: `int[] pressure`
   - creates a name of type "`int` array"
   - types `int` and `int[]` are different
     - `int[]`: type of the array
     - `int` : type of the individual values

1. To create a new array, e.g. `pressure = new int[100];`

2. To refer to a specific element in the array
   - also called *an indexed variable*, e.g.

```
pressure[3] = keyboard.nextInt();
System.out.println("You entered" + pressure[3]);
```

Array Length

- Specified by the number in brackets when created with `new`
  - *maximum* number of elements the array can hold
  - storage is allocated whether or not the elements are assigned values

- the attribute `length`,

```
Species[] entry = new Species[20];
System.out.println(entry.length);
```

- The `length` attribute is established in the declaration and cannot be changed unless the array is redeclared

Subscript Range

- Array subscripts use zero-numbering
  - the first element has subscript 0
  - the second element has subscript 1
  - etc. - the n^{th} element has subscript n-1
  - the last element has subscript `length-1`
- For example: an int array with 4 elements

Subscript:	0	1	2	3
Value:	97	86	92	71

Subscript out of Range Error

- Using a subscript larger than `length-1` causes a *run time* (not a compiler) error
  - an `ArrayOutOfBoundsException` is thrown
    - you do not need to catch it
    - you need to fix the problem and recompile your code
- Other programming languages, e.g. C and C++, do not even cause a run time error!
  - one of the most dangerous characteristics of these languages is that they allow out of bounds array indices.

Array Length Specified at Run-time

```
// array length specified at compile-time
int[] array1 = new int[10];


// array length specified at run-time
// calculate size…
int size = …;
int[] array2 = new int[size];
```

Programming Tip:

Use Singular Array Names

- Using singular rather than plural names for arrays improves readability

- Although the array contains many elements the most common use of the name will be with a subscript, which references a *single* value.
- It is easier to read:
  - `score[3]` than
  - `scores[3]`

Initializing an Array's Values in Its Declaration

- can be initialized by putting a comma-separated list in braces
- Uninitialized elements will be assigned some default value, e.g. 0 for `int` arrays (**explicit initialization** is recommended)
- The length of an array is automatically determined when the values are explicitly initialized in the declaration
- For example:

```
double[] reading = {5.1, 3.02, 9.65};
System.out.println(reading.length);
```

- displays `3`, the length of the array `reading`

Initializing Array Elements in a Loop

- A `for` loop is commonly used to initialize array elements
- For example:

```
int i;//loop counter/array index
int[] a = new int[10];
for(i = 0; i < a.length; i++)
   a[i] = 0;
```

  - note that the loop counter/array index goes from `0` to `length - 1`
  - it counts through `length = 10` iterations/elements using the zero-numbering of the array index

Programming Tip:

Do not count on default initial values for array elements

  - explicitly initialize elements in the declaration or in a loop

Arrays, Classes, and Methods

An array of a class can be declared and the class's methods applied to the elements of the array.

This excerpt from the Sales Report program in the text uses the `SalesAssociate` class to create an array of sales associates:

create an array of `SalesAssociates`

each array element is a `SalesAssociate` variable

use the `readInput` method of `SalesAssociate`

```
public void getFigures()
{
   System.out.println("Enter number of sales associates:");
   numberOfAssociates = SavitchIn.readLineInt();
   SalesAssociate[] record =
            new SalesAssociate[numberOfAssociates];
   for (int i = 0; i < numberOfAssociates; i++)
   {
      record[i] = new SalesAssociate();
      System.out.println("Enter data for associate " + (i + 1));
      record[i].readInput();
      System.out.println();
   }
}
```

Arrays and Array Elements as Method Arguments

- Arrays and array elements can be
  - used with classes and methods just like other objects
  - be an argument in a method
  - returned by methods

Indexed Variables as Method Arguments

`nextScore` is an array of `int`s

an element of `nextScore` is an argument of method `average`

`average` method definition

```
public static void main(String[] arg)
  {
     Scanner keyboard = new Scanner(System.in);a
     System.out.println("Enter your score on exam 1:");
     int firstScore = keyboard.nextInt();
     int[ ] nextScore = new int[3];
     int i;
     double possibleAverage;
     for (i = 0; i < nextScore.length; i++)
        nextScore[i] = 80 + 10*i;
     for (i = 0; i < nextScore.length; i++)
     {
        possibleAverage = average(firstScore, nextScore[i]);
        System.out.println("If your score on exam 2 is "
                    + nextScore[i]);
        System.out.println("your average will be "
                    + possibleAverage);
     }
  }
  public static double average(int n1, int n2)
  {
    return (n1 + n2)/2.0;
  }
```

Excerpt from ArgumentDemo program in text.

Passing Array Elements

```
public static void main(String[] arg)
 {
    SalesAssociate[] record = new SalesAssociate[numberOfAssociates];
    int i;
    for (i = 0; i < numberOfAssociates; i++)
    {
       record[i] = new SalesAssociate();
       System.out.println("Enter data for associate " + (i + 1));
       record[i].readInput();
    }
    m(record[0]);
 }
public static void m(SalesAssociate sa)
{
}
```

When Can a Method Change an Indexed Variable Argument?

- primitive types are "call by value"
  - only a copy of the value is passed as an argument
  - method *cannot* change the value of the indexed variable
- class types are reference types ("call by reference")
  - pass the address of the object
  - the corresponding parameter in the method definition becomes an alias of the object
  - the method has access to the actual object
  - so the method *can* change the value of the indexed variable if it is a class (and not a primitive) type

Passing Array Elements

```
int[] grade = new int[10];
obj.method(grade[i]);   // grade[i] cannot be changed

… method(int grade)     // pass by value; a copy
{
}
```

```
Person[] roster = new Person[10];
obj.method(roster[i]); // roster[i] can be changed

… method(Person p)      // pass by reference; an alias
{
}
```

Array Names as Method Arguments

- Use just the array name and no brackets
- Pass by reference
  - the method has access to the original array and can change the value of the elements
- The length of the array passed can be different for each call
  - when you define the method you do not need to know the length of the array that will be passed
  - use the `length` attribute inside the method to avoid `ArrayIndexOutOfBoundsExceptions`

Example: An Array as an Argument in a Method Call

```
public static void showArray(char[] a)
{
    int i;
    for(i = 0; i < a.length; i++)
        System.out.println(a[i]);
}


-------------
char[] grades = new char[45];
MyClass.showArray(grades);
```

the method's argument is the name of an array of characters

uses the `length` attribute to control the loop
allows different size arrays and avoids index-out-of-bounds exceptions

Arguments for the Method `main`

- The heading for the `main` method shows a parameter that is an array of `String`s:
  `public static void main(`**`String[] arg`**`)`
- When you run a program from the command line, all words after the class name will be passed to the main method in the arg array.
  `java TestProgram` **`Josephine Student`**
- The following `main` method in the class `TestProgram` will print out the first two arguments it receives:

```
Public static void main(String[] arg)
{
   System.out.println("Hello " + arg[0] + " " + arg[1]);
}
```

- In this example, the output from the command line above will be:
  `Hello Josephine Student`

Using = with Array Names: Remember They Are Reference Types

```
int[] a = new int[3];
int[] b = new int[3];
for(int i=0; i < a.length; i++)
   a[i] = i;
b = a;
System.out.println(a[2] + " " + b[2]);
a[2] = 10;
System.out.println(a[2] + " " + b[2]);
```

The output for this code will be:

```
2 2
10 10
```

This does not create a copy of array `a`; it makes `b` another *name* for array `a`.

A value changed in `a` is the same value obtained with `b`

Using == with array names: remember they are reference types

```
int i;
int[] a = new int[3];
int[] b = new int[3];
for(i=0; i < a.length; i++)
   a[i] = 0;
for(i=0; i < b.length; i++)
   b[i] = 0;
if(b == a)
   System.out.println("a equals b");
else
   System.out.println("a does not equal b");
```

`a` and `b` are both 3-element arrays of `int`s

all elements of `a` and `b` are assigned the value `0`

tests if the *addresses* of `a` and `b` are equal, not if the array values are equal

The output for this code will be "`a does not equal b`" because the *addresses* of the arrays are not equal.

Behavior of Three Operations

	Primitive Type	Class Type	Entire Array	Array Element
Assignment (=)	Copy content	Copy address	Copy address	Depends on primitive/ class type
Equality (==)	Compare content	Compare address	Compare address	Depends on primitive/ class type
Parameter Passing	*Pass by value* (content)	*Pass by reference* (address)	*Pass by reference* (address)	Depends on primitive/ class type

Testing Two Arrays for Equality

- To test two arrays for equality you need to define an `equals` method that returns true if and only the arrays have the same length and all corresponding values are equal

```
public static boolean equals(int[] a,
                             int[] b)
  {
      boolean match = false;
      if (a.length == b.length)
      {
          match = true; //tentatively
          int i = 0;
          while (match && (i < a.length))
          {
              if (a[i] != b[i])
                  match = false;
              i++;
          }
      }
      return match;
   }
```

Methods that Return an Array

- the address of the array is passed
- The local array name within the method is just another name for the original array

```
public class returnArrayDemo
  {
      public static void main(String arg[])
      {
          char[] c;
          c = vowels();
          for(int i = 0; i < c.length; i++)
             System.out.println(c[i]);
       }
       public static char[] vowels()
       {
            char[] newArray = new char[5];
            newArray[0] = 'a';
            newArray[1] = 'e';
            newArray[2] = 'i';
            newArray[3] = 'o';
            newArray[4] = 'u';
            return newArray;
       }
  }
```

`c`, `newArray`, and the return type of `vowels` are all the same type: `char []`

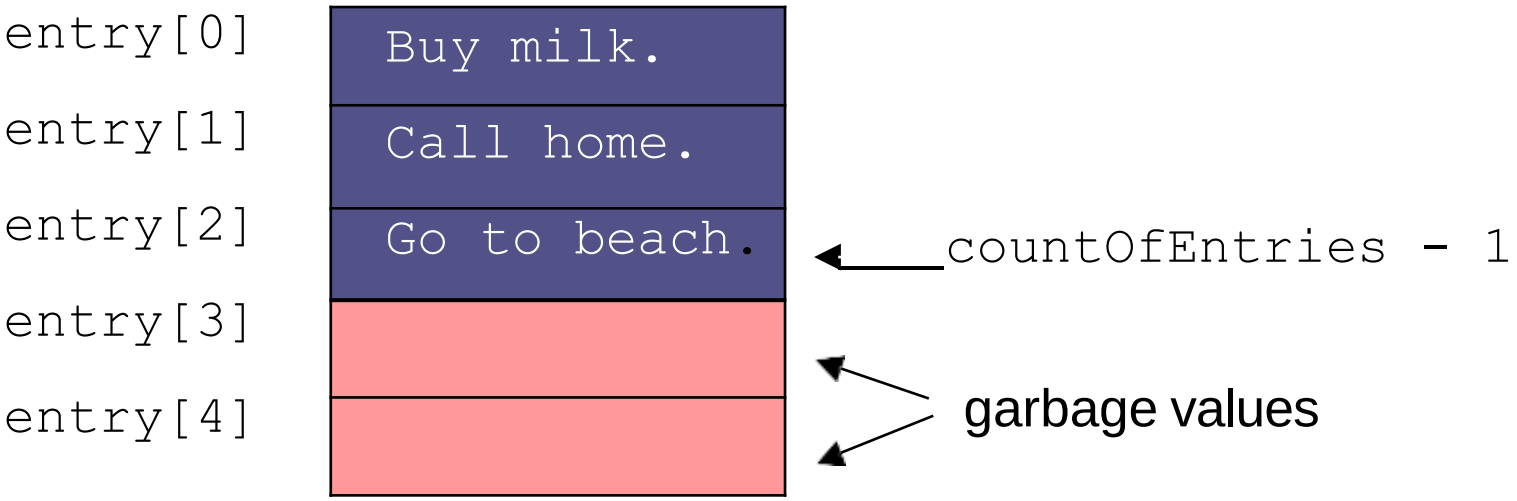
Wrapper Classes for Arrays

- Arrays can be made into objects by creating a wrapper class
  - similar to wrapper classes for primitive types
- In the wrapper class:
  - make an array an attribute
  - define constructors
  - define accessor methods to read and write element values and parameters
- The text shows an example of creating a wrapper class for an array of objects of type `OneWayNoRepeatsList`
  - the wrapper class defines two constructors plus the following methods:
    `addItem`, `full`, `empty`, `entryAt`, `atLastEntry`, `onList`, `maximumNumberOfEntries`, `numberOfEntries`, and `eraseList`

Partially Filled Arrays

- Sometimes only part of an array has been filled with data
- Array elements always contain something
  - elements which have not been written to
    - contain unknown (*garbage*) data so you should avoid reading them
- There is no automatic mechanism to detect how many elements have been filled
  - *you*, the programmer need to keep track!
- An example: the instance variable `countOfEntries` (in the class `OneWayNoRepeatsList`) is incremented every time `addItem` is called (see the text)

Example of a Partially Filled Array



`countOfEntries` has a value of `3`.

`entry.length` has a value of `5`.

DATA STRUCTURES AND APPLICATIONS

Arrays- Operations
Deleting, Searching and Sorting

Searching an Array

- There are many techniques for searching an array for a particular value

- *Sequential search*:
  - start at the beginning of the array and proceed in sequence until either the value is found or the end of the array is reached*
    - if the array is only partially filled, the search stops when the last meaningful value has been checked
  - it is not the most efficient way
  - but it works and is easy to program

* Or, just as easy, start at the end and work backwards toward the beginning

Example: Sequential Search of an Array

The `onList` method of `OneWayNoRepeatsList` sequentially searches the array `entry` to see it the parameter `item` is in the array

```
public boolean onList(String item)
{
      boolean found = false;
      int i = 0;
      while ((! found) &&
              (i < countOfEntries))
      {
          if (item.equals(entry[i]))
              found = true;
          else
              i++;
      }

     return found;
}
```

Returning an Array Attribute (Instance Variable)

- Access methods that return references to array instance variables cause problems for information hiding.
  Example:

```
class …
{
  private String[] entry;
  …
  public String[] getEntryArray()
  {
        return entry;
  }
```

Even though `entries` is declared private, a method outside the class can get full access to it by using `getEntryArray`.

- In most cases this type of method is not necessary anyhow.
- If it is necessary, make the method return a copy of the array instead of returning a reference to the actual array.

Sorting an Array

- Sorting a list of elements is another very common problem (along with searching a list)
  - sort numbers in ascending order
  - sort numbers in descending order
  - sort strings in alphabetic order
  - etc.

- There are many ways to sort a list, just as there are many ways to search a list

- *Selection sort*
  - one of the easiest
  - not the most efficient, but easy to understand and program

Selection Sort Algorithm for an Array of Integers

To sort an array on integers in ascending order:

1. Find the smallest number and record its index
2. swap (interchange) the smallest number with the first element of the array
   - the sorted part of the array is now the first element
   - the unsorted part of the array is the remaining elements
3. repeat Steps 2 and 3 until all elements have been placed
   - each iteration increases the length of the sorted part by one

Selection Sort Example

Key:
- smallest remaining value
- sorted elements

Problem: sort this 10-element array of integers in ascending order:

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]
7	6	11	17	3	15	5	19	30	14

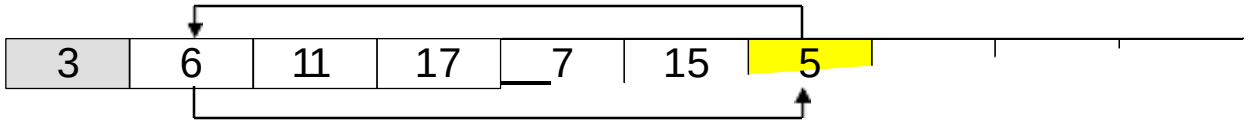
1st iteration: smallest value is 3, its index is 4, swap a[0] with a[4]

before:

after:

3	6	11	17	7	15	5			

2nd iteration: smallest value in remaining list is 5, its index is 6, swap a[1] with a[6]



3	5	11	17	7	15	6	19	30	14

How many iterations are needed?

Example: Selection Sort

- Notice the precondition: every array element has a value
- may have duplicate values
- broken down into smaller tasks
  - "find the index of the smallest value"
  - "interchange two elements"
  - `private` because they are helper methods (users are not expected to call them directly)

Selection Sort Code

```
/**********************************************
*Precondition:
*Every indexed variable of the array a has a value.
*Action: Sorts the array a so that
*a[0] <= a[1] <= ... <= a[a.length - 1].
**********************************************/
public static void sort(int[] a)
{
    int index, indexOfNextSmallest;
    for (index = 0; index < a.length - 1; index++)
    {//Place the correct value in a[index]:
        indexOfNextSmallest = indexOfSmallest(index, a);
        interchange(index,indexOfNextSmallest, a);
        //a[0] <= a[1] <=...<= a[index] and these are
        //the smallest of the original array elements.
        //The remaining positions contain the rest of
        //the original array elements.
    }
}
```

Insertion Sort

- Basic Idea:
  - Keeping expanding the sorted portion by one
  - Insert the next element into the right position in the sorted portion
- Algorithm:
  1. Start with one element [is it sorted?] – sorted portion
  2. While the sorted portion is not the entire array
     1. Find the right position in the sorted portion for the next element
     2. Insert the element
     3. If necessary, move the other elements down
     4. Expand the sorted portion by one

Insertion Sort: An example

- First iteration
  - Before: **[5]**, 3, 4, 9, 2
  - After: **[3, 5]**, 4, 9, 2
- Second iteration
  - Before: **[3, 5]**, 4, 9, 2
  - After: **[3, 4, 5]**, 9, 2
- Third iteration
  - Before: **[3, 4, 5]**, 9, 2
  - After: **[3, 4, 5, 9]**, 2
- Fourth iteration
  - Before: **[3, 4, 5, 9]**, 2
  - After: **[2, 3, 4, 5, 9]**

Bubble Sort

- Basic Idea:
  - Expand the sorted portion one by one
  - "Sink" the largest element to the bottom after comparing adjacent elements
  - The smaller items "bubble" up
- Algorithm:
  - While the unsorted portion has more than one element
    - Compare adjacent elements
    - Swap elements if out of order
    - Largest element at the bottom, reduce the unsorted portion by one

Bubble Sort: An example

- First Iteration:
  - [5, 3], 4, 9, 2 → [3, 5], 4, 9, 2
  - 3, [5, 4], 9, 2 → 3, [4, 5], 9, 2
  - 3, 4, [5, 9], 2 → 3, 4, [5, 9], 2
  - 3, 4, 5, [9, 2] → 3, 4, 5, [2, 9]
- Second Iteration:
  - [3, 4], 5, 2, **9** → [3, 4], 5, 2, **9**
  - 3, [4, 5], 2, **9** → 3, [4, 5], 2, **9**
  - 3, 4, [5, 2], **9** → 3, 4, [2, 5], **9**
- Third Iteration:
  - [3, 4], 2, **5, 9** → [3, 4], 2, **5, 9**
  - 3, [4, 2], **5, 9** → 3, [2, 4], **5, 9**
- Fourth Iteration:
  - [3, 2], **4, 5, 9** → [2, 3], **4, 5, 9**

Multidimensional Arrays

- Arrays with more than one index
  - number of dimensions = number of indexes
- Arrays with more than two dimensions are a simple extension of two-dimensional (2-D) arrays
- A 2-D array corresponds to a table or grid
  - one dimension is the row
  - the other dimension is the column
  - cell: an intersection of a row and column
  - an array element corresponds to a cell in the table

Table as a 2-Dimensional Array

- The table assumes a starting balance of $1000
- First dimension: row identifier - Year
- Second dimension: column identifier - percentage
- Cell contains balance for the year (row) and percentage (column)
- Balance for year 4, rate 7.00% = $1311

Balances for Various Interest Rates Compounded Annually (Rounded to Whole Dollar Amounts)

Year	5.00%	5.50%	6.00%	6.50%	7.00%	7.50%
1	$1050	$1055	$1060	$1065	$1070	$1075
2	$1103	$1113	$1124	$1134	$1145	$1156
3	$1158	$1174	$1191	$1208	$1225	$1242
4	$1216	$1239	$1262	$1286		$1335
5	$1276	$1307	$1338	$1370	$1403	$1436
...	...	...	...	...	...	...

Table as a 2-D Array

Column Index 4
(5th column)

Row Index 3
(4th row)

Indexes	0	1	2	3	4	5
0	$1050	$1055	$1060	$1065	$1070	$1075
1	$1103	$1113	$1124	$1134	$1145	$1156
2	$1158	$1174	$1191	$1208	$1225	$1242
3	$1216	$1239	$1262	$1286		$1335
4	$1276	$1307	$1338	$1370	$1403	$1436
…	…	…	…	…	…	…

- Generalizing to two indexes: [row][column]
- First dimension: row index
- Second dimension: column index
- Cell contains balance for the year/row and percentage/column
- All indexes use zero-numbering
  - Balance[3][4] = cell in 4th row (year = 4) and 5th column (7.50%)
  - Balance[3][4] = $1311 (shown in yellow)

Processing a 2-D Array: for Loops Nested 2-Deep

- To process all elements of an *n*-D array nest *n* for loops
  - each loop has its own counter that corresponds to an index
- For example: calculate and enter balances in the interest table
  - inner loop repeats 6 times (six rates) for every outer loop iteration
  - the outer loop repeats 10 times (10 different values of `years`)
  - so the inner repeats 10 x 6 = 60 times = # cells in `table`

```
int[][] table = new int[10][6];
int row, column;
for (row = 0; row < 10; row++)
      for (column = 0; column < 6; column++)
          table[row][column] = balance(1000.00,
row + 1, (5 + 0.5*column));
```

Excerpt from `main` method of `InterestTable`

Multidimensional Array Parameters and Returned Values

- Methods may have multi-D array parameters
- Methods may return a multi-D array as the value returned
- The situation is similar to 1-D arrays, but with more brackets
- Example: a 2-D `int` array as a method argument

```
public static void showTable(int[][] displayArray)
{
   int row, column;
   for (row = 0; row < displayArray.length; row++)
   {
      System.out.print((row + 1) + "      ");
      for (column = 0; column < displayArray[row].length; column++)
         System.out.print("$" + displayArray[row][column] + "   ");
      System.out.println();
   }
}
```

Notice how the number of rows is obtained

Notice how the number of columns is obtained

`showTable` method from class `InterestTable2`

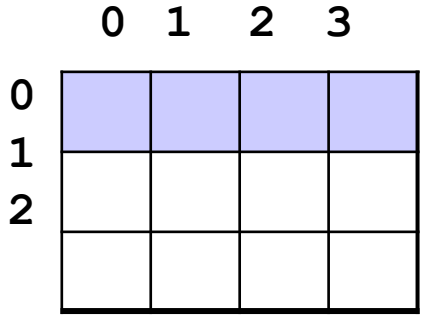
Implementation of Multidimensional Arrays

- Multidimensional arrays are implemented as ***arrays of arrays***.

  Example:

  ```
  int[][] table = new int[3][4];
  ```

  - `table` is a one-dimensional array of length 3
  - Each element in `table` is an array with base type `int`.
- Access a row by only using only one subscript:
  - `table[0].length` gives the length (4) of the first row in the array



`table[0]` refers to the first row in the array, which is a one-dimensional array.

Note: `table.length` (which is 3 in this case) is not the same thing as `table[0].length` (which is 4).

Ragged Arrays

- Ragged arrays have rows of unequal length
  - each row has a different number of columns, or entries
- Ragged arrays are allowed in Java
- Example: create a 2-D `int` array named `b` with 5 elements in the first row, 7 in the second row, and 4 in the third row:

```
int[][] b = new int[3][];
b[0] = new int[5];
b[1] = new int[7];
b[2] = new int[4];
```

Programming Example: Employee Time Records

- The class `TimeBook` uses several arrays to keep track of employee time records:

```
public class TimeBook
{
        private int numberOfEmployees;
        private  int[][]  hours;
        private int[] weekHours;
        private int[] dayHours;
        . . .
}
```

`hours[i][j]` has the hours for employee `j` on day `i`

`dayHours[i]` has the total hours worked by all employees on day `i`

`weekHours[j]` has the week's hours for employee `j+1`

Nested Loops with Multidimensional Arrays

```
for (employeeNumber = 1;
     employeeNumber <= numberOfEmployees; employeeNumber++)
{  // Process one employee
   sum = 0;
   for (dayNumber = 0; dayNumber < 5; dayNumber++)
      sum = sum + hours[dayNumber][employeeNumber - 1];
   weekHours[employeeNumber - 1] = sum;
}
```

hours array

	0	1	2
0	8	0	9
1	8	0	9
2	8	8	8
3	8	8	4
4	8	8	8

weekHours array

0	1	2
40	24	38

- The method `computeWeekHours` uses nested `for` loops to compute the week's total hours for each employee.
- Each time through the outer loop body, the inner loop adds all the numbers in one column of the `hours` array to get the value for one element in the `weekHours` array.

Parallel Arrays

```
public class Course
{
    private String   _name;
    private String[] _studentName;
    private int[]    _studentId;
    private float[]  _studentGrade;
    private String[] _assignmentName;   // parallel array?

    public Course(String name, int numOfStudents)
    {
        _name = name;
        _studentName = new String[numOfStudents];
        _studentId = new int[numOfStudents];
        _studentGrade = new float[numOfStudents];
        for (int i = 0; i < numOfStudents; i++)
        {
            _studentName[i] = "none";
            _studentId[i] = 0;
            _studentGrade[i] = 0.0;
        }
    }
}
```

Array of Objects

```
public class Student
{
    private String _name;
    private int    _id;
    private float  _grade;

    public Student()                          { _name = "none"; _id = 0;
   _grade = .0; }
    public Student(String name, int id, float grade)
                                              { _name = name; _id = id;
   _grade = grade;}
}

public class Course
{
    private String    _name;
    private Student[] _student;

    public Course(String name, int numOfStudents)
    {
       _name = name;
       _student = new Student[numOfStudents];
     for (int i = 0; i < numOfStudents; i++)
       _student[i] = new Student();     // how to init
  name,id,grade for each obj
    }}
```

Array Operations - Polynomials and Sparse Matrices

- Is it necessary to define an array as an ADT?
  - C++ array requires the index set to be a set of consecutive integers starting at 0
  - C++ does not check an array index to ensure that it belongs to the range for which the array is defined.

ADT 2.1 Abstract data type GeneralArray

```
class GeneralArray {
// objects: A set of pairs < index, value> where for each value of index in IndexSet there
// is a value of type float. IndexSet is a finite ordered set of one or more dimensions,
// for example, {0, …, n-1} for one dimension, {(0, 0), (0, 1), (0, 2),(1, 0), (1, 1), (1, 2), (2, 0),
// (2, 1), (2, 2)} for two dimensions, etc.

public:
    GeneralArray(int j; RangeList list, float initValue = defatultValue);
    // The constructor GeneralArray creates a j dimensional array of floats; the range of
    // the kth dimension is given by the kth element of list. For each index i in the index
    // set, insert <i, initValue> into the array.

    float Retrieve(index i);
    // if (i is in the index set of the array) return the float associated with i
    // in the array; else signal an error

    void Store(index i, float x);
    // if (i is in the index set of the array) delete any pair of the form <i, y> present
    // in the array and insert the new pair <i, x>; else signal an error.
}; // end of GeneralArray
```

Polynomial Representations

Representation 1

```
private:
   int degree;                  // degree ≤ MaxDegree
   float coef [MaxDegree + 1];
```

Representation 2

```
private:
   int degree;
   float *coef;

   Polynomial::Polynomial(int d)
{
   degree = d;
   coef = new float [degree+1];
}
```

Representation 3

```
class Polynomial;  // forward delcaration

class term {
    friend Polynomial;
    private:
        float coef;                    // coefficient
        int exp;              // exponent
};

private:
    static term termArray[MaxTerms];
    static int free;
    int Start, Finish;

term Polynomial:: termArray[MaxTerms];
Int Polynomial::free = 0;          // location of next free location in
    temArray
```

Array Representation of two Polynomials

Represent the following two polynomials:

$A(x) = 2x^{1000} + 1$

$B(x) = x^4 + 10x^3 + 3x^2 + 1$

	A.Start	*A.Finish*	*B.Start*			*B.Finish*	*free*
coef	2	1	1	10	3	1	
exp	1000	0	4	3	2	0	
	0	*1*	*2*	*3*	*4*	*5*	*6*

Polynomial Addition

```
Polynomial Polynomial:: Add(Polynomial B)
// return the sum of A(x) ( in *this) and B(x)
{
   Polynomial C; int a = Start; int b = B.Start; C.Start = free; float c;
   while ((a <= Finish) && (b <= B.Finish))
      switch (compare(termArray[a].exp, termArray[b].exp)) {
         case '=':
              c = termArray[a].coef +termArray[b].coef;
              if ( c ) NewTerm(c, termArray[a].exp);
                    a++; b++;
                    break;
         case '<':
              NewTerm(termArray[b].coef, termArray[b].exp);
              b++;
         case '>':
              NewTerm(termArray[a].coef, termArray[a].exp);
              a++;
   }
```

```
      // end of switch and while
    // add in remaining terms of A(x)
    for (; a<= Finish; a++)
      NewTerm(termArray[a].coef, termArray[a].exp);
    // add in remaining terms of B(x)
    for (; b<= B.Finish; b++)
      NewTerm(termArray[b].coef, termArray[b].exp);
    C.Finish = free – 1;
    return C;
} // end of Add
```

Adding a new Term

```
void Polynomial::NewTerm(float c, int e)
// Add a new term to C(x)
{
    if (free >= MaxTerms) {
            cerr << "Too many terms in polynomials"<< endl;
            exit();
    }
    termArray[free].coef = c;
    termArray[free].exp = e;
    free++;
} // end of NewTerm
```

Disadvantages of Representing Polynomials by Arrays

- What should we do when free is going to exceed MaxTerms?
  - Either quit or reused the space of unused polynomials. But costly.
- If use a single array of terms for each polynomial, it may alleviate the above issue but it penalizes the performance of the program due to the need of knowing the size of a polynomial beforehand.

Sparse matrix and its representations

- A **matrix** is a two-dimensional data object made of m rows and n columns, therefore having total m x n values. If most of the elements of the matrix have **0 value**, then it is called a sparse matrix.

- **Why to use Sparse Matrix instead of simple matrix ?**
- **Storage:** There are lesser non-zero elements than zeros and thus lesser memory can be used to store only those elements.
- **Computing time:** Computing time can be saved by logically designing a data structure traversing only non-zero elements..

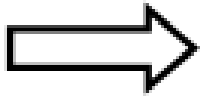
- Example:

- Representing a sparse matrix by a 2D array leads to wastage of lots of memory as zeroes in the matrix are of no use in most of the cases. So, instead of storing zeroes with non-zero elements, we only store non-zero elements. This means storing non-zero elements with **triples- (Row, Column, value).**
- Sparse Matrix Representations can be done in many ways following are two common representations:
- Array representation
- Linked list representation

Method 1: Using Arrays

- 2D array is used to represent a sparse matrix in which there are three rows named as
- **Row:** Index of row, where non-zero element is located
- **Column:** Index of column, where non-zero element is located
- **Value:** Value of the non zero element located at index – (row,column)

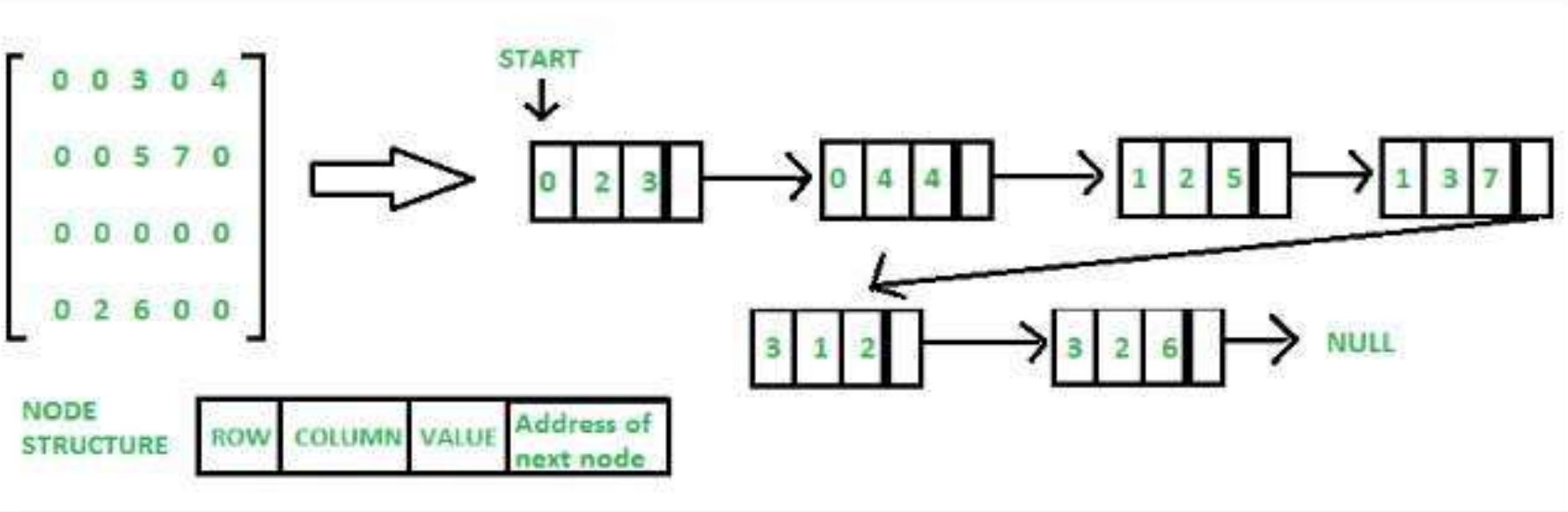
$$\begin{bmatrix} 0 & 0 & 3 & 0 & 4 \\ 0 & 0 & 5 & 7 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 2 & 6 & 0 & 0 \end{bmatrix}$$



Row	0	0	1	1	3	3
Column	2	4	2	3	1	2
Value	3	4	5	7	2	6

Method 2: Using Linked Lists

- In linked list, each node has four fields. These four fields are defined as:
- **Row:** Index of row, where non-zero element is located
- **Column:** Index of column, where non-zero element is located
- **Value:** Value of the non zero element located at index – (row,column)
- **Next node:** Address of the next node



DATA STRUCTURES AND APPLICATIONS

STRINGS – Its Operations

Strings

- A string is a sequence of characters treated as a group
- We have already used some string literals:
  - “filename”
  - “output string”
- Strings are important in many programming contexts:
  - names
  - other objects (numbers, identifiers, etc.)

Outline

- Strings
  - Representation in C
  - String Literals
  - String Variables
  - String Input/Output

    printf, scanf, gets, fgets, puts, fputs
  - String Functions

    strlen, strcpy, strncpy, strcmp, strncmp, strcat, strncat, strchr, strrchr, strstr, strspn, strcspn, strtok
  - Reading from/Printing to Strings

    sprintf, sscanf

Strings in C

- No explicit type, instead strings are maintained as arrays of characters
- Representing strings in C
  - stored in arrays of characters
  - array can be of any length
  - end of string is indicated by a *delimiter*, the zero character '\0'

"A String"

A		S	t	r	i	n	g	\0

String Literals

- String literal values are represented by sequences of characters between double quotes (“)
- Examples
  - “” - empty string
  - “hello”
- “a” versus ‘a’
  - ‘a’is a single character value (stored in 1 byte) as the ASCII value for a
  - “a” is an array with two characters, the first is a, the second is the character value \0

Referring to String Literals

- String literal is an array, can refer to a single character from the literal as a character
- Example:
  printf("%c","hello"[1]);
  outputs the character 'e'
- During compilation, C creates space for each string literal (# of characters in the literal + 1)
  - referring to the literal refers to that space (as if it is an array)

Duplicate String Literals

- Each string literal in a C program is stored at a different location
- So even if the string literals contain the same string, they are not equal (in the == sense)
- Example:
  - char string1[6] = “hello”;
  - char string2[6] = “hello”;
  - but string1 does not equal string2 (they are stored at different locations)

String Variables

- Allocate an array of a size large enough to hold the string (plus 1 extra value for the delimiter)
- Examples (with initialization):

```
char str1[6] = "Hello";
char str2[] = "Hello";
char *str3 = "Hello";
char str4[6] = {'H','e','l','l','o','\0'};
```

- Note, each variable is considered a constant in that the space it is connected to cannot be changed

```
str1 = str2; /* not allowable, but we can copy the contents
                of str2 to str1 (more later) */
```

Changing String Variables

- Cannot change space string variables connected to, but can use pointer variables that can be changed
- Example:

```
char *str1 = “hello”;    /* str1 unchangeable */
char *str2 = “goodbye”; /* str2 unchangeable */
char *str3; /* Not tied to space */
str3 = str1; /* str3 points to same space s1 connected to */
str3 = str2;
```

Changing String Variables (cont)

- Can change parts of a string variable

```
char str1[6] = "hello";
str1[0] = 'y';
/* str1 is now "yello" */
str1[4] = '\0';
/* str1 is now "yell" */
```

- Important to retain delimiter (replacing str1[5] in the original string with something other than '\0' makes a string that does not end)
- Have to stay within limits of array

String Input

- Use %s field specification in scanf to read string
  - ignores leading white space
  - reads characters until next white space encountered
  - C stores null (\0) char after last non-white space char
  - Reads into array (no & before name, array is a pointer)
- Example:

```
char Name[11];
scanf("%s",Name);
```

- Problem: no limit on number of characters read (need one for delimiter), if too many characters for array, problems may occur

String Input (cont)

- Can use the width value in the field specification to limit the number of characters read:

```
char Name[11];
scanf("%10s",Name);
```

- Remember, you need one space for the \0
  - width should be one less than size of array
- Strings shorter than the field specification are read normally, but C always stops after reading 10 characters

String Input (cont)

- Edit set input %[*ListofChars*]
  - ListofChars specifies set of characters (called scan set)
  - Characters read as long as character falls in scan set
  - Stops when first non scan set character encountered
  - Note, does not ignored leading white space
  - Any character may be specified except]
  - Putting ^ at the start to negate the set (any character BUT list is allowed)
- Examples:

```
scanf("%[-+0123456789]",Number);
scanf("%[^\n]",Line); /* read until newline char */
```

String Output

- Use %s field specification in printf:
  characters in string printed until \0 encountered
  char Name[10] = “Rich”;
  printf(“|%s|”,Name); /* outputs |Rich| */
- Can use width value to print string in space:
  printf(“|%10s|”,Name); /* outputs | Rich| */
- Use - flag to left justify:
  printf(“|%-10s|”,Name); /* outputs |Rich | */

Input/Output Example

```
#include <stdio.h>

void main() {
  char LastName[11];
  char FirstName[11];

  printf("Enter your name (last , first): ");
  scanf("%10s%*[^,],%10s",LastName,FirstName);

  printf("Nice to meet you %s %s\n",
        FirstName,LastName);
}
```

Printing a String

Commands:

int puts(char *str)

prints the string pointed to by str to the screen
prints until delimiter reached (string better have a \0)
returns EOF if the puts fails
outputs newline if \n encountered (for strings read with gets or fgets)

int fputs(char *str, FILE *fp)

prints the string pointed to by str to the file connected to fp
fp must be an output connection
returns EOF if the fputs fails
outputs newline if \n encountered

Array of Strings

- Sometimes useful to have an array of string values
- Each string could be of different length (producing a ragged string array)
- Example:

```
char *MonthNames[13]; /* an array of 13 strings */
MonthNames[1] = “January”; /* String with 8 chars */
MonthNames[2] = “February”; /* String with 9 chars */
MonthNames[3] = “March”; /* String with 6 chars */
etc.
```

Array of Strings Example

```
#include <stdio.h>

void main() {
  char *days[7];
  char TheDay[10];
  int day;

  days[0] = "Sunday";
  days[1] = "Monday";
  days[2] = "Tuesday";
  days[3] = "Wednesday";
  days[4] = "Thursday";
  days[5] = "Friday";
  days[6] = "Saturday";
```

Array of Strings Example

```
  printf("Please enter a day: ");
  scanf("%9s",TheDay);

  day = 0;
  while ((day < 7) && (!samestring(TheDay,days[day])))
    day++;

  if (day < 7)
    printf("%s is day %d.\n",TheDay,day);
  else
    printf("No day %s!\n",TheDay);
}
```

Array of Strings Example

```
int samestring(char *s1, char *s2) {
  int i;

  /* Not same if not of same length */
  if (strlen(s1) != strlen(s2))
    return 0;
  /* look at each character in turn */
  for (i = 0; i < strlen(s1); i++)
    /* if a character differs, string not same */
    if (s1[i] != s2[i]) return 0;
  return 1;
}
```

String Functions

- C provides a wide range of string functions for performing different string tasks
- Examples

  strlen(str) - calculate string length

  strcpy(dst,src) - copy string at src to dst

  strcmp(str1,str2) - compare str1 to str2
- Functions come from the utility library string.h
  - #include <string.h> to use

String Length

Syntax: int strlen(char *str)

returns the length (integer) of the string argument

counts the number of characters until an \0 encountered

does not count \0 char

Example:

char str1 = “hello”;

strlen(str1) would return 5

Copying a String

Syntax:

char *strcpy(char *dst, char *src)

copies the characters (including the \0) from the source string (src) to the destination string (dst)

dst should have enough space to receive entire string (if not, other data may get written over)

if the two strings overlap (e.g., copying a string onto itself) the results are unpredictable

return value is the destination string (dst)

char *strncpy(char *dst, char *src, int n)

similar to strcpy, but the copy stops after n characters

if n non-null (not \0) characters are copied, then no \0 is copied

String Comparison

Syntax:

int strcmp(char *str1, char *str2)

compares str1 to str2, returns a value based on the first character they differ at:

less than 0

if ASCII value of the character they differ at is smaller for str1

or if str1 starts the same as str2 (and str2 is longer)

greater than 0

if ASCII value of the character they differ at is larger for str1

or if str2 starts the same as str1 (and str1 is longer)

0 if the two strings do not differ

String Comparison (cont)

strcmp examples:

strcmp(“hello”,”hello”) -- returns 0

strcmp(“yello”,”hello”) -- returns value > 0

strcmp(“Hello”,”hello”) -- returns value < 0

strcmp(“hello”,”hello there”) -- returns value < 0

strcmp(“some diff”,”some dift”) -- returns value < 0

expression for determining if two strings s1,s2 hold the same string value:

!strcmp(s1,s2)

String Comparison (cont)

Sometimes we only want to compare first n chars:

int strncmp(char *s1, char *s2, int n)

Works the same as strcmp except that it stops at the nth character

looks at less than n characters if either string is shorter than n

strcmp("some diff","some DIFF") -- returns value > 0

strncmp("some diff","some DIFF",4) -- returns 0

strcpy/strcmp Example

```
#include <stdio.h>
#include <string.h>

void main() {
  char fname[81];
  char prevline[101] = "";
  char buffer[101];
  FILE *instream;

  printf("Check which file: ");
  scanf("%80s",fname);

  if ((instream = fopen(fname,"r")) == NULL) {
    printf("Unable to open file %s\n",fname);
    exit(-1);
  }
```

strcpy/strcmp Example

```
/* read a line of characters */
while (fgets(buffer,sizeof(buffer)-1,instream) != NULL) {
  /* if current line same as previous */
  if (!strcmp(buffer,prevline))
  printf("Duplicate line: %s",buffer);
  /* otherwise if the first 10 characters of the current
     and previous line are the same */
  else if (!strncmp(buffer,prevline,10))
    printf(”Start the same:\n %s %s",prevline,buffer);
  /* Copy the current line (in buffer) to the previous
     line (in prevline) */
  strcpy(prevline,buffer);
}

fclose(instream);
}
```

String Comparison (ignoring case)

Syntax:

int strcasecmp(char *str1, char *str2)

similar to strcmp except that upper and lower case characters (e.g., ‘a’and ‘A’) are considered to be equal

int strncasecmp(char *str1, char *str2, int n)

version of strncmp that ignores case

String Concatenation

Syntax:

char *strcat(char *dstS, char *addS)

- appends the string at addS to the string dstS (after dstS's delimiter)
- returns the string dstS
- can cause problems if the resulting string is too long to fit in dstS

char *strncat(char *dstS, char *addS, int n)

- appends the first n characters of addS to dstS
- if less than n characters in addS only the characters in addS appended
- always appends a \0 character

strcat Example

```
#include <stdio.h>
#include <string.h>

void main()  {
  char fname[81];
  char buffer[101];
  char curraddress[201] = "";
  FILE *instream;
  int first = 1;

  printf("Address file: ");
  scanf("%80s",fname);

  if ((instream = fopen(fname,"r")) == NULL) {
    printf("Unable to open file %s\n",fname);
    exit(-1);
  }
```

strcat Example

```
/* Read a line */
while (fgets(buffer,sizeof(buffer)-1,instream) != NULL) {
    if (buffer[0] == '*') { /* End of address */
    printf("%s\n",curraddress); /* Print address */
    strcpy(curraddress,""); /* Reset address to “” */
      first = 1;
    }
    else {
      /* Add comma (if not first entry in address) */
      if (first) first = 0; else strcat(curraddress,", ");
      /* Add line (minus newline) to address */
      strncat(curraddress,buffer,strlen(buffer)-1);
    }
  }

  fclose(instream);
}
```

Searching for a Character/String

Syntax:

- char *strchr(char *str, int ch)
  - returns a pointer (a char *) to the first occurrence of ch in str
  - returns NULL if ch does not occur in str
  - can subtract original pointer from result pointer to determine which character in array
- char *strstr(char *str, char *searchstr)
  - similar to strchr, but looks for the first occurrence of the string searchstr in str
- char *strrchr(char *str, int ch)
  - similar to strchr except that the search starts from the end of string str and works backward

String Spans (Searching)

Syntax:

int strspn(char *str, char *cset)

specify a set of characters as a string cset

strspn searches for the first character in str that is not part of cset

returns the number of characters in set found before first non-set character found

int strcspn(char *str, char *cset)

similar to strspn except that it stops when a character that is part of the set is found

Examples:

strspn(“a vowel”,”bvcwl”) returns 2

strcspn(“a vowel”,”@,*e”) returns 5

Printing to a String

The sprintf function allows us to print to a string argument using printf formatting rules

First argument of sprintf is string to print to, remaining arguments are as in printf

Example:

```
char buffer[100];
sprintf(buffer,"%s, %s",LastName,FirstName);
if (strlen(buffer) > 15)
  printf("Long name %s %s\n",FirstName,LastName);
```

Reading from a String

The sscanf function allows us to read from a string argument using scanf rules

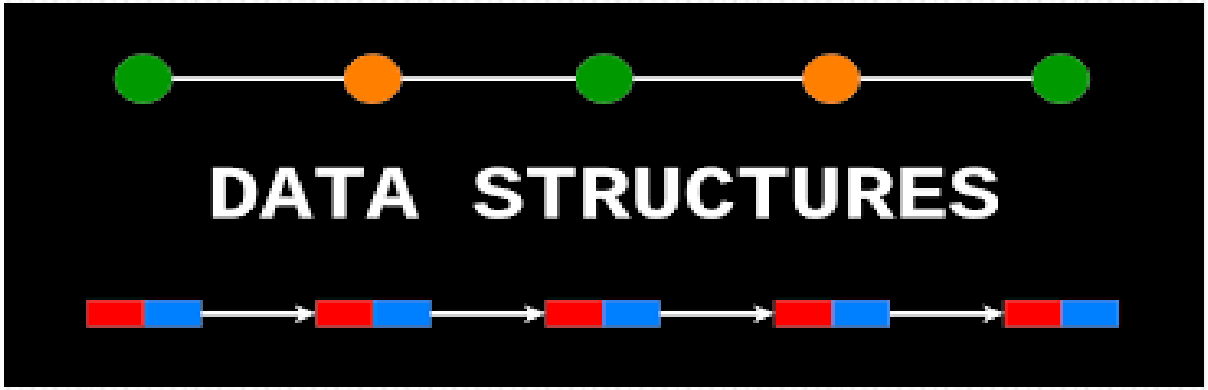
First argument of sscanf is string to read from, remaining arguments are as in scanf

Example:

```
char buffer[100] = "A10 50.0";
sscanf(buffer,"%c%d%f",&ch,&inum,&fnum);
/* puts 'A' in ch, 10 in inum and 50.0 in fnum */
```

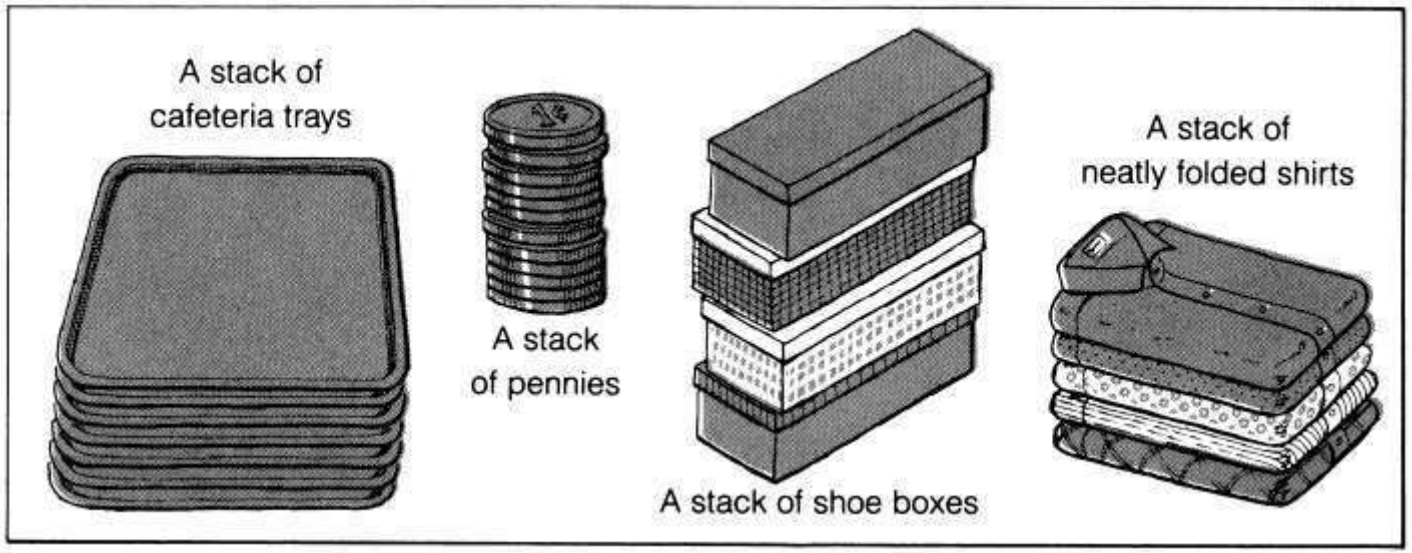
Data Structure and Applications – 18CS32

Module 2:
Stacks , Recursion and Queues



What is a stack?

- It is an ordered group of homogeneous items of elements.
- Elements are added to and removed from the top of the stack (the most recently added items are at the top of the stack).
- The last element to be added is the first to be removed (**LIFO**: Last In, First Out).



Stack Specification

- Definitions: (provided by the user)
  - *MAX_ITEMS*: Max number of items that might be on the stack
  - *ItemType*: Data type of the items on the stack
- Operations
  - MakeEmpty
  - Boolean IsEmpty
  - Boolean IsFull
  - Push (ItemType newItem)
  - Pop (ItemType& item) (or pop and top)

Push (ItemType newItem)

- *Function*: Adds newItem to the top of the stack.
- *Preconditions*: Stack has been initialized and is not full.
- *Postconditions*: newItem is at the top of the stack.

STACK OPERATIONS

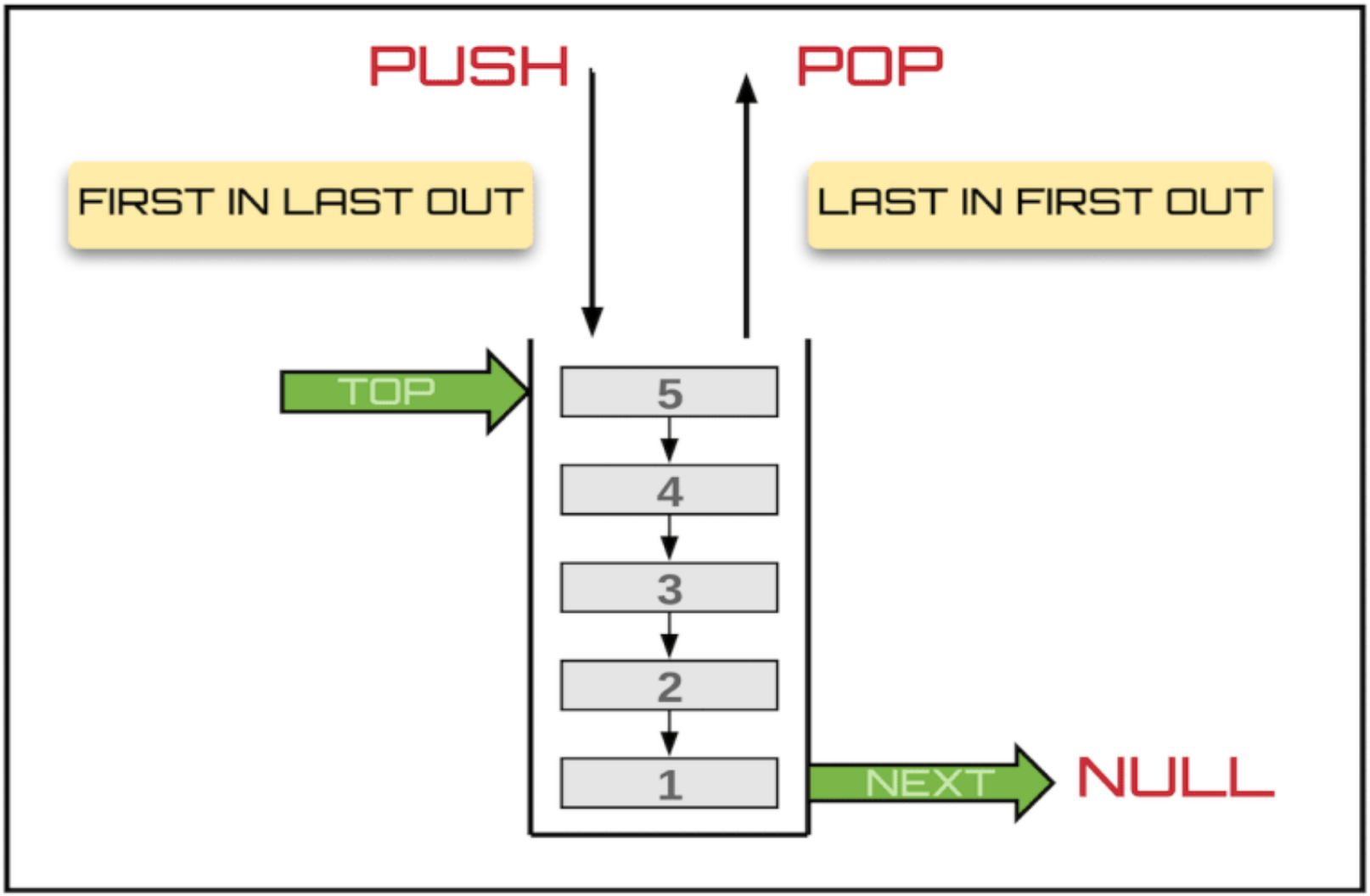
The bottom of a stack is a sealed end. Stack may have a capacity which is a limitation on the number of elements in a stack. The operations on stack are

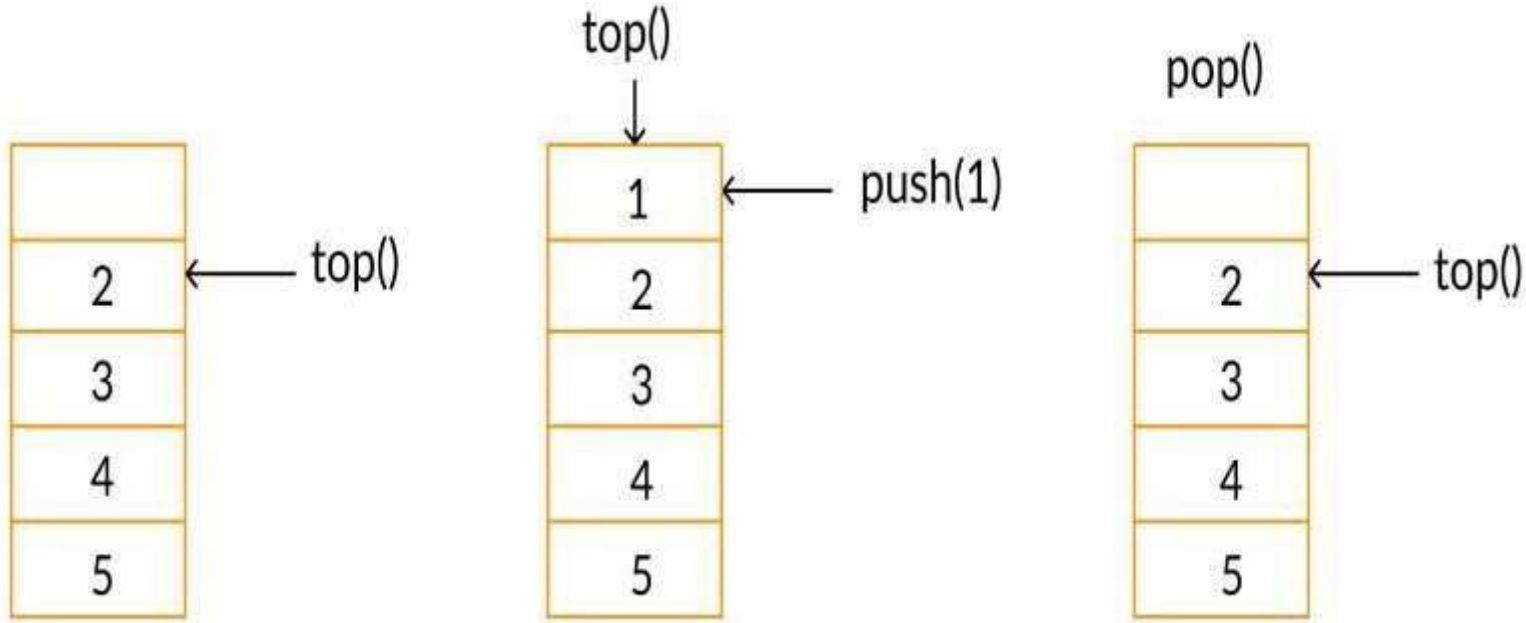
- Push: Places an object on the *top* of the stack.
- Pop: Removes an object from the *top* of the stack.
- IsEmpty: Reports whether the stack is empty or not.
- IsFull: Reports whether the stack exceeds limit or not.

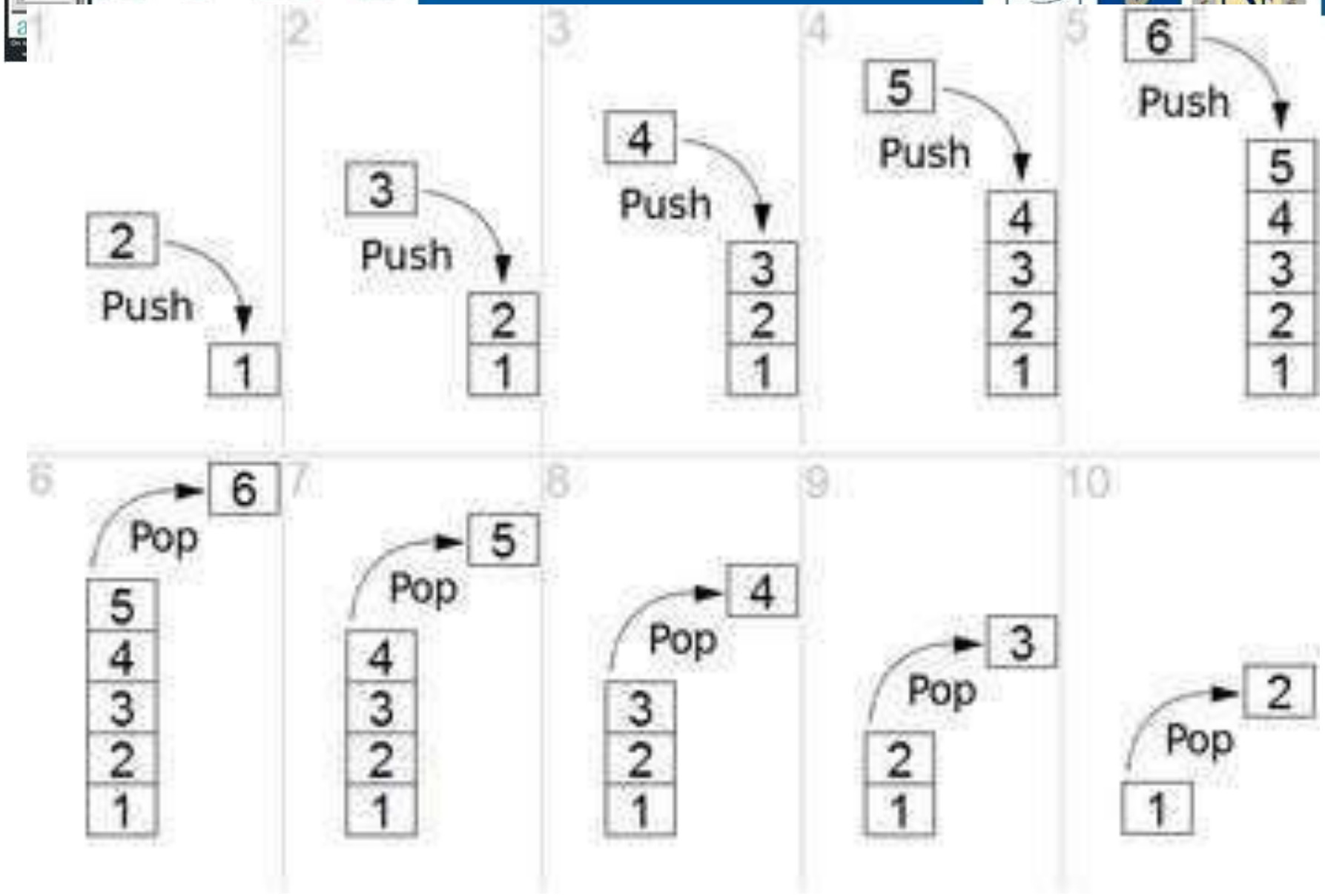
(i)stack (ii)push(s,a) (iii)push(s,d)-stack overflow (iv)pop(s)

(v)pop(s)-stack underflow

STACK

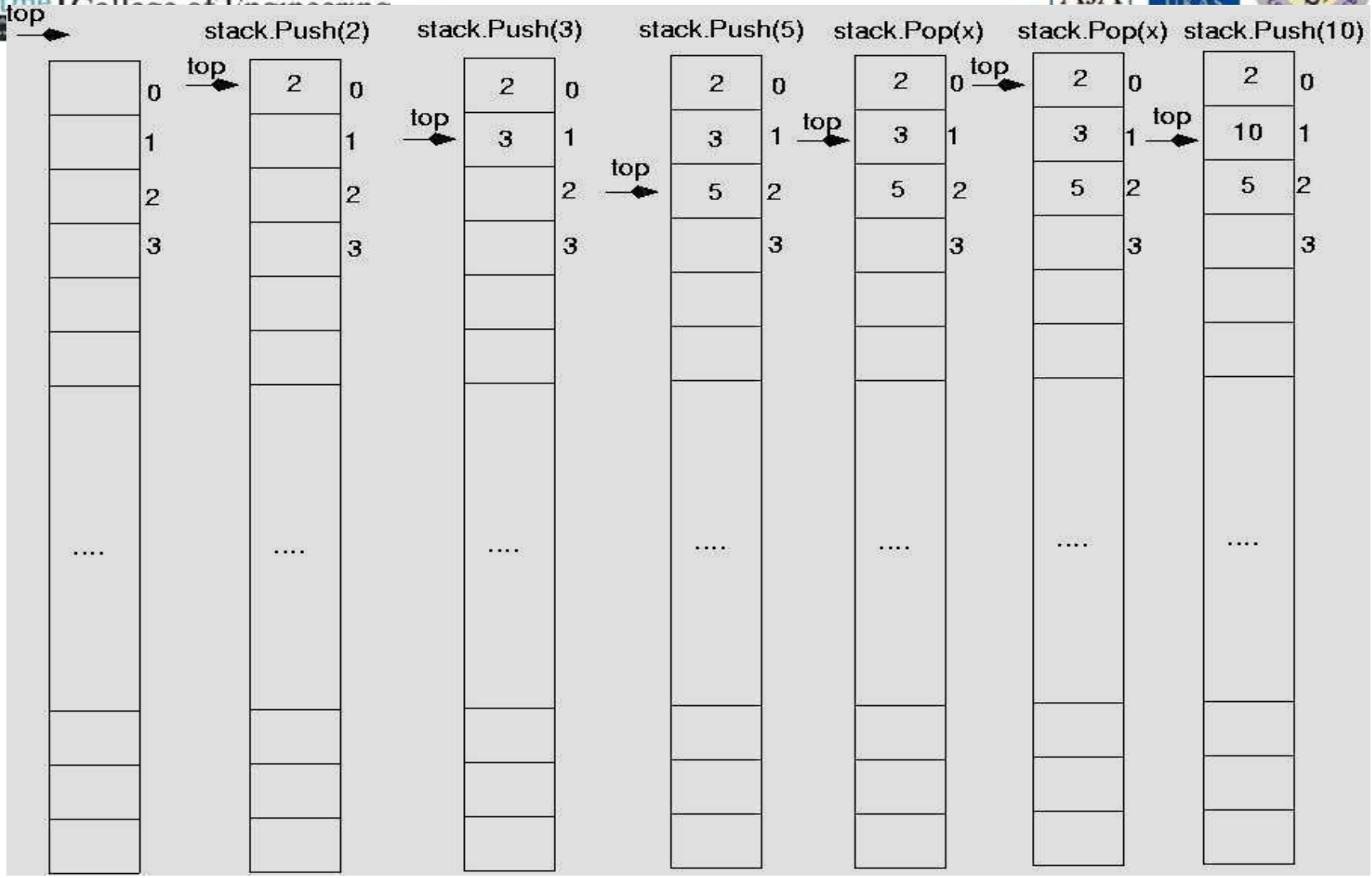

PUSH
POP
FIRST IN LAST OUT
LAST IN FIRST OUT
TOP
5
4
3
2
1
NEXT
NULL


top()
2
3
4
5
top()
top()
1
2
3
4
5
push(1)
pop()
2
3
4
5
top()


1
2
Push
1
2
3
Push
2
1
3
4
Push
3
2
1
4
5
Push
4
3
2
1
5
6
Push
5
4
3
2
1
6
7
8
9
10
Pop
6
5
4
3
2
1
Pop
5
4
3
2
1
Pop
4
3
2
1
Pop
3
2
1
Pop
2
1

Pop (ItemType& item)

- *Function*: Removes topItem from stack and returns it in item.
- *Preconditions*: Stack has been initialized and is not empty.
- *Postconditions*: Top element has been removed from stack and item is a copy of the removed element.


top
stack.Push(2)
stack.Push(3)
stack.Push(5)
stack.Pop(x)
stack.Pop(x)
stack.Push(10)
0
1
2
3
....
2
3
5
10

Implementing a Stack

- At least three different ways to implement a stack
  - array
  - vector
  - linked list
- Which method to use depends on the application
  - what advantages and disadvantages does each implementation have?

Implementing Stacks: Array

- Advantages
  - best performance
- Disadvantage
  - fixed size
- Basic implementation
  - initially empty array
  - field to record where the next data gets placed into
  - if array is full, push() returns false
    - otherwise adds it into the correct spot
  - if array is empty, pop() returns null
    - otherwise removes the next item in the stack

Stack Class (array based)

```
class StackArray {
    private Object[ ] stack;
    private int nextIn;
    public StackArray(int size) {
            stack = new Object[size];
            nextIn = 0;}
    public boolean push(Object data);
    public Object pop();
    public void clear();
    public boolean isEmpty();
    public boolean isFull();
}
```

push() Method (array based)

```
public boolean push(Object data) {
   if(nextIn == stack.length) { return false; }    // stack
   is full

   // add the element and then increment nextIn
   stack[nextIn] = data;
   nextIn++;
   return true;
}
```

pop() Method (array based)

```
public Object pop() {
   if(nextIn == 0) { return null; }    // stack is empty

   // decrement nextIn and return the data
    nextIn--;
   Object data = stack[nextIn];
   return data;
}
```

Stack Implementation

```
#include "ItemType.h"
// Must be provided by the user of the class
// Contains definitions for MAX_ITEMS and ItemType

class StackType {
 public:
          StackType();
          void MakeEmpty();
    bool IsEmpty() const;
          bool IsFull() const;
          void Push(ItemType);
          void Pop(ItemType&);
private:
          int top;
          ItemType items[MAX_ITEMS];
};
```

Stack Implementation (cont.)

```
StackType::StackType()
{
 top = -1;
}

void StackType::MakeEmpty()
{
 top = -1;
}

bool StackType::IsEmpty() const
{
 return (top == -1);
}
```

Stack Implementation (cont.)

```
bool StackType::IsFull() const
{
 return (top == MAX_ITEMS-1);
}

 void StackType::Push(ItemType newItem)
{
 top++;
 items[top] = newItem;
}

 void StackType::Pop(ItemType& item)
{
 item = items[top];
 top--;
}
```

Stack overflow

- The condition resulting from trying to push an element onto a full stack.

```
if(!stack.IsFull())
    stack.Push(item);
```

Stack underflow

- The condition resulting from trying to pop an empty stack.

```
if(!stack.IsEmpty())
    stack.Pop(item);
```

Implementing stacks using templates

- Templates allow the compiler to generate multiple versions of a class type or a function by allowing parameterized types.

Implementing stacks using templates

(cont.)

```
template<class ItemType>
class StackType {
 public:
    StackType();
    void MakeEmpty();
    bool IsEmpty() const;
    bool IsFull() const;
    void Push(ItemType);
    void Pop(ItemType&);
 private:
    int top;
    ItemType items[MAX_ITEMS];
};
```

Example using templates

```
// Client code
StackType<int> myStack;
StackType<float> yourStack;
StackType<StrType> anotherStack;

myStack.Push(35);
yourStack.Push(584.39);
```

The compiler generates distinct class types and gives its own internal name to each of the types.

Function templates

- The definitions of the member functions must be rewritten as function templates.

```
template<class ItemType>
StackType<ItemType>::StackType()
{
 top = -1;
}

template<class ItemType>
void StackType<ItemType>::MakeEmpty()
{
 top = -1;
}
```

Function templates (cont.)

```
template<class ItemType>
bool StackType<ItemType>::IsEmpty() const
{
 return (top == -1);
}

template<class ItemType>
bool StackType<ItemType>::IsFull() const
{
 return (top == MAX_ITEMS-1);
}

template<class ItemType>
void StackType<ItemType>::Push(ItemType newItem)
{
 top++;
 items[top] = newItem;
}
```

Function templates (cont.)

```
template<class ItemType>
void StackType<ItemType>::Pop(ItemType& item)
{
 item = items[top];
 top--;
}
```

Implementing stacks using dynamic array allocation

```
template<class ItemType>
class StackType {
 public:
    StackType(int);
    ~StackType();
   void MakeEmpty();
    bool IsEmpty() const;
    bool IsFull() const;
    void Push(ItemType);
    void Pop(ItemType&);
```

Implementing stacks using dynamic array allocation (cont.)

```
template<class ItemType>
StackType<ItemType>::StackType(int max)
{
 maxStack = max;
 top = -1;
 items = new ItemType[max];
}

template<class ItemType>
StackType<ItemType>::~StackType()
{
 delete [ ] items;
}
```

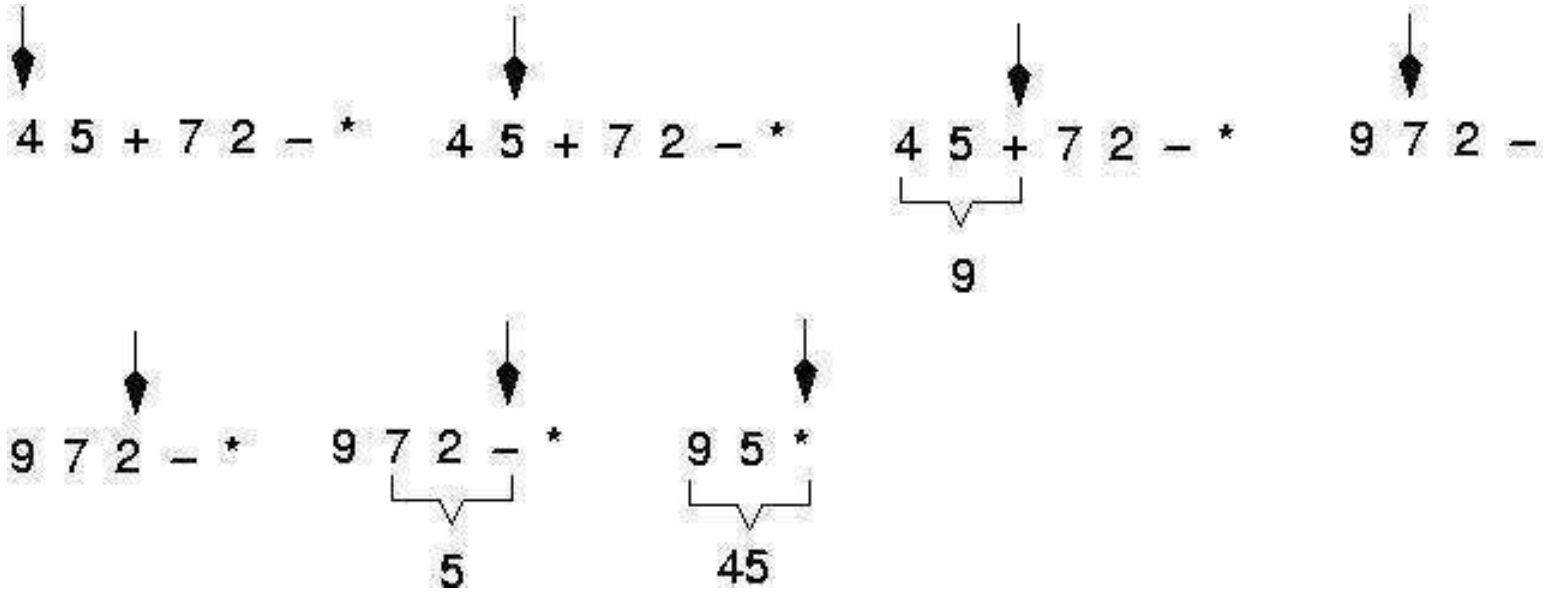
Example: postfix expressions

- Postfix notation is another way of writing arithmetic expressions.
- In postfix notation, the operator is written after the two operands.

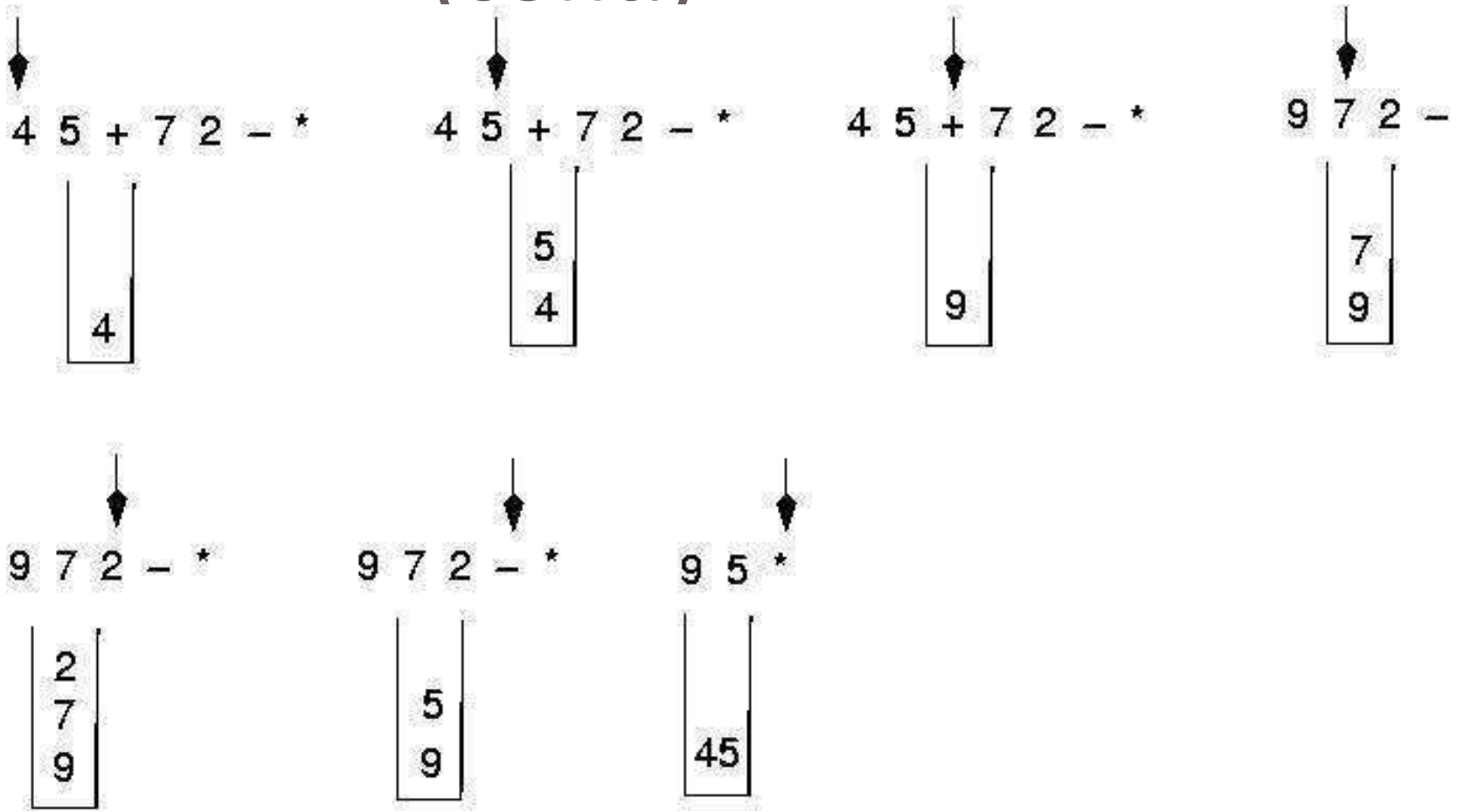
  infix: 2+5 *postfix*: 2 5 +

- Expressions are evaluated from left to right.
- Precedence rules and parentheses are never needed!!

Example: postfix expressions (cont.)



Postfix expressions: Algorithm using stacks (cont.)



expressions : Algorithm using stacks

```
WHILE more input items exist
   Get an item
   IF item is an operand
     stack.Push(item)
   ELSE
     stack.Pop(operand2)
     stack.Pop(operand1)
     Compute result
     stack.Push(result)
stack.Pop(result)
```

Write the body for a function that replaces each copy of an item in a stack with another item. Use the following specification. (this function is a client program).

ReplaceItem(StackType& stack, ItemType oldItem, ItemType newItem)

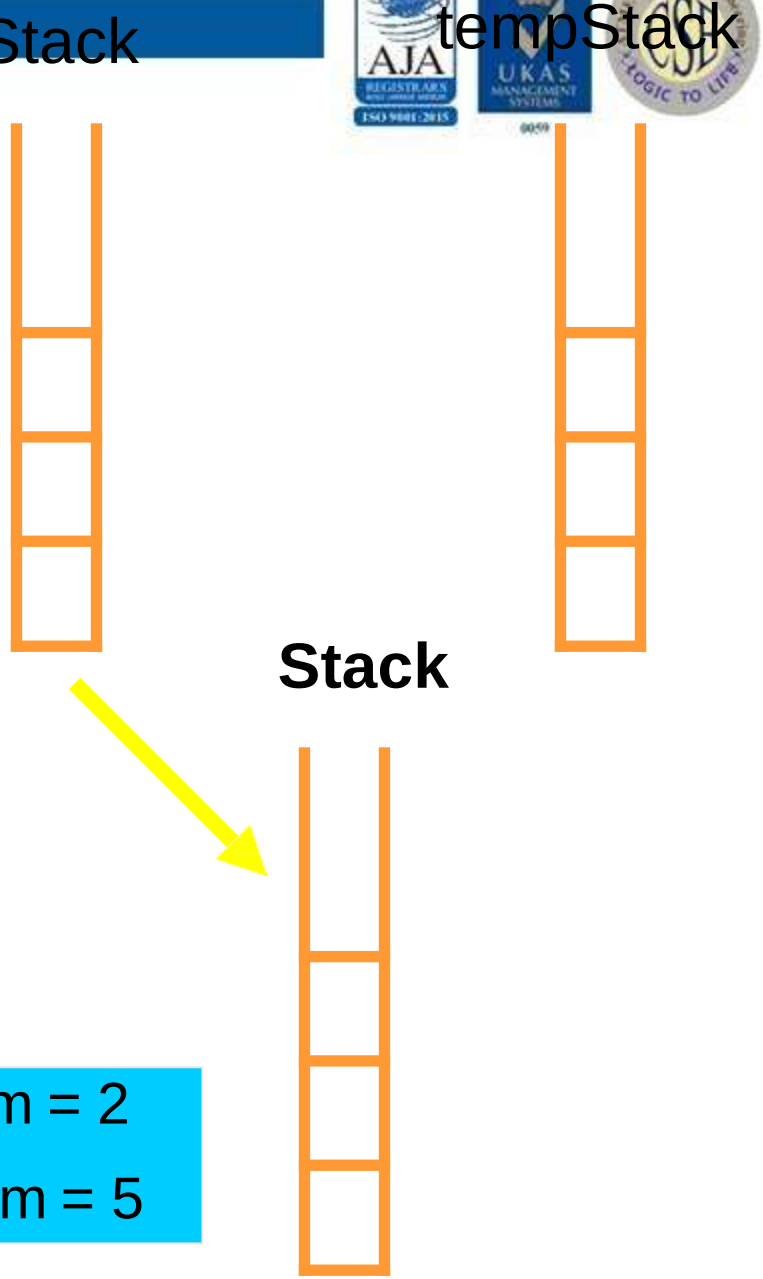
Function: Replaces all occurrences of oldItem with newItem.

Precondition: stack has been initialized.

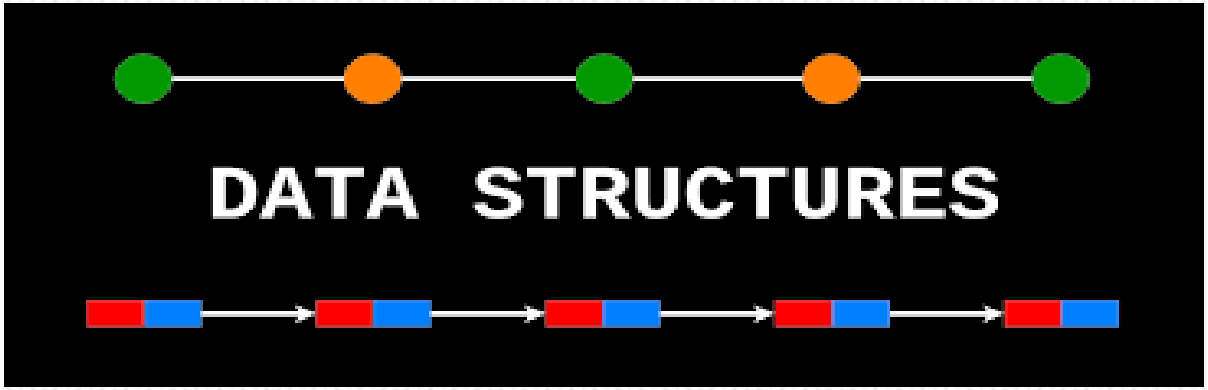
Postconditions: Each occurrence of oldItem in stack has been replaced by newItem.

(You may use any of the member functions of the StackType, but you may not assume any knowledge of how the stack is implemented).

```
ItemType item;
StackType tempStack;

while (!Stack.IsEmpty()) {
        Stack.Pop(item);
        if (item==oldItem)
          tempStack.Push(newItem);
        else
          tempStack.Push(item);
}
while (!tempStack.IsEmpty()) {
        tempStack.Pop(item);
        Stack.Push(item);
}
}
```

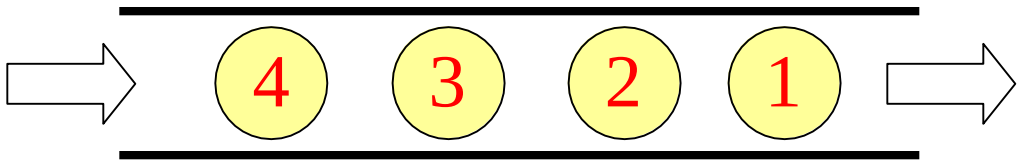


Module 2:
Queues



What is a queue?

- It is an ordered group of homogeneous items of elements.
- Queues have two ends:
  - Elements are added at one end.
  - Elements are removed from the other end.
- The element added first is also removed first (**FIFO**: First In, First Out).



Queue Specification

- Definitions: (provided by the user)
  - *MAX_ITEMS*: Max number of items that might be on the queue
  - *ItemType*: Data type of the items on the queue

- Operations
  - MakeEmpty
  - Boolean IsEmpty
  - Boolean IsFull
  - Enqueue (ItemType newItem)
  - Dequeue (ItemType& item) (

Enqueue (ItemType newItem)

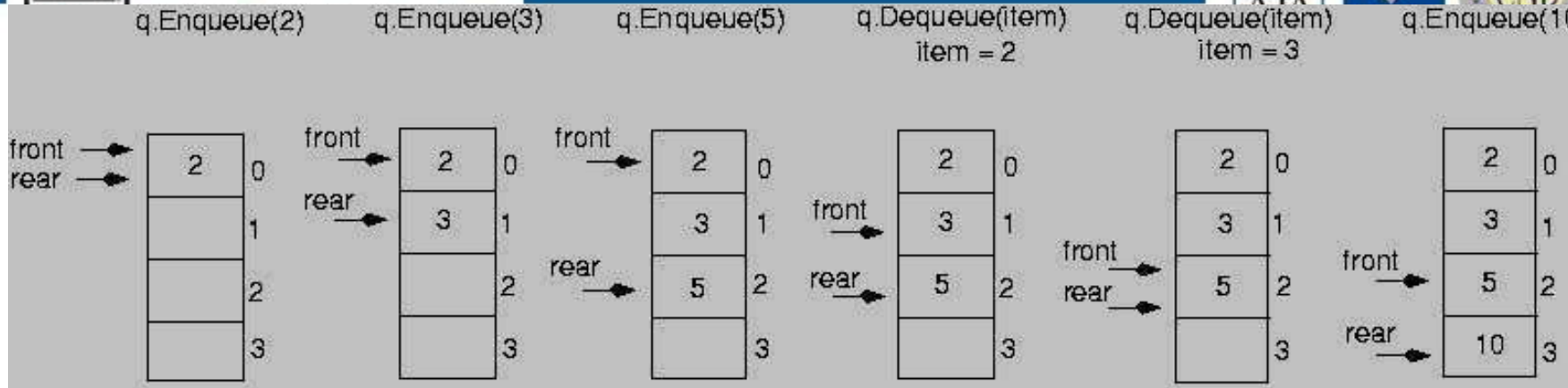
- *Function*: Adds newItem to the rear of the queue.
- *Preconditions*: Queue has been initialized and is not full.
- *Postconditions*: newItem is at rear of queue.

Dequeue (ItemType& item)

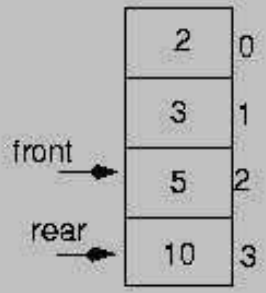
- *Function*: Removes front item from queue and returns it in item.
- *Preconditions*: Queue has been initialized and is not empty.
- *Postconditions*: Front element has been removed from queue and item is a copy of removed element.

Implementation issues

- Implement the queue as a *circular structure*.
- How do we know if a queue is full or empty?
- Initialization of *front* and *rear*.
- Testing for a *full* or *empty* queue.



q.Enqueue(20) ???

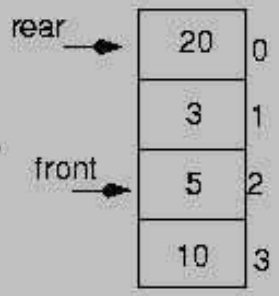


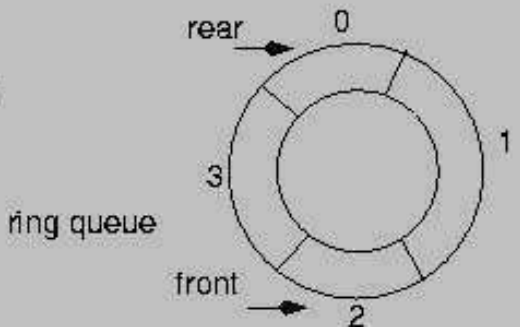
Let the queue elements "wrap around"

```
if(rear == maxQue -1)
  rear = 0;
else
  rear = rear + 1;
```

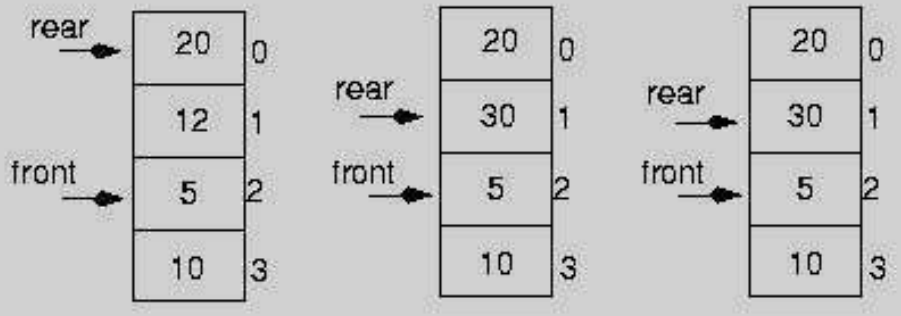
or

```
rear = (rear + 1) % maxQue;
```





q.Enqueue(30) q.Enqueue(50) ???

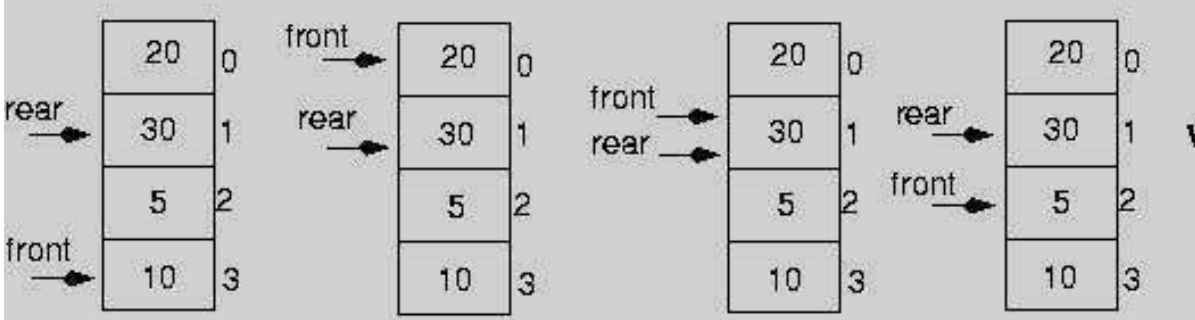


The queue is full !!

What is the condition for a full queue ?

```
rear + 1 == front
```

q.Dequeue(item) item = 5 q.Dequeue(item) item = 10 q.Dequeue(item) item = 20 q.Dequeue(item) item = 30



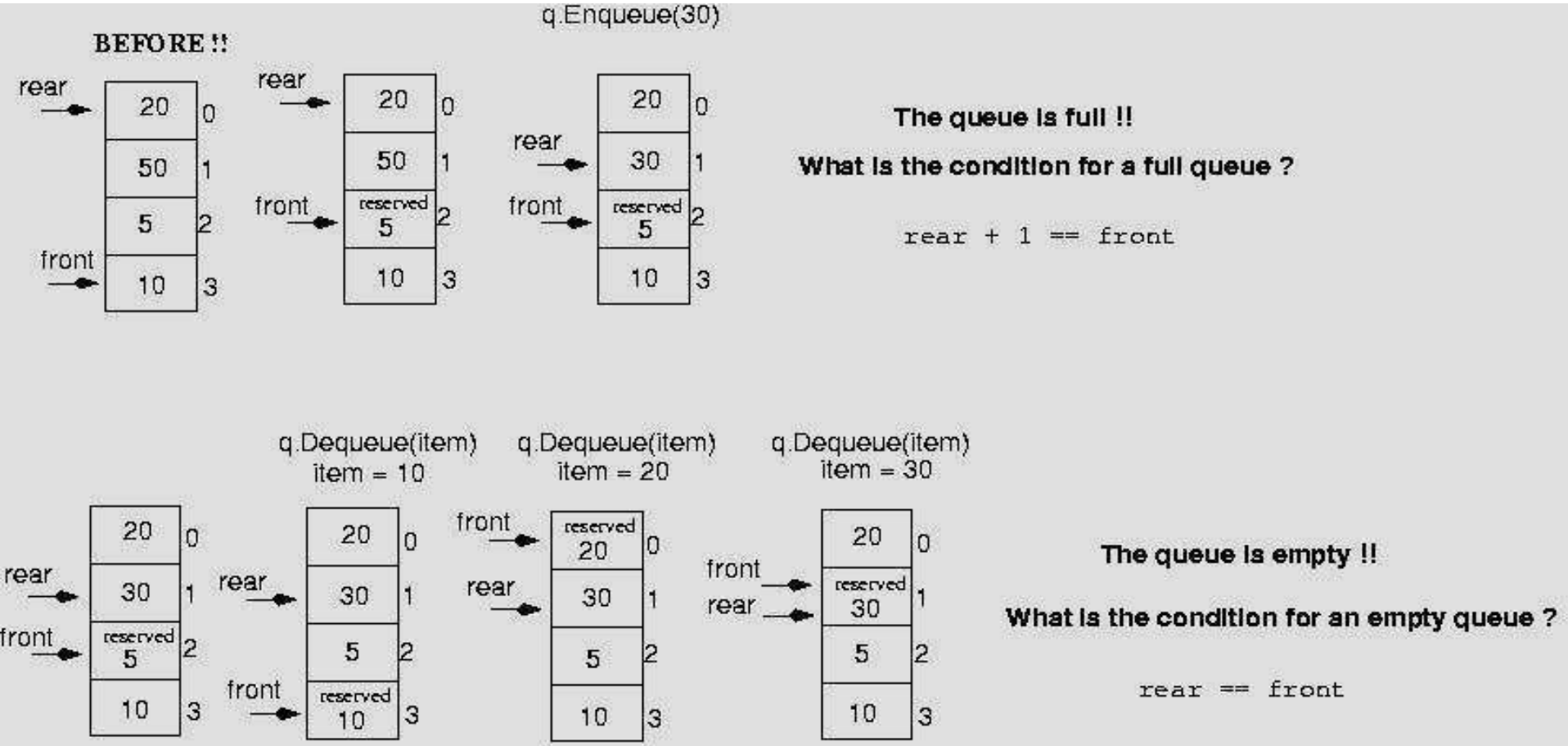
The queue is empty !!

What is the condition for an empty queue ?

```
rear + 1 == front
```

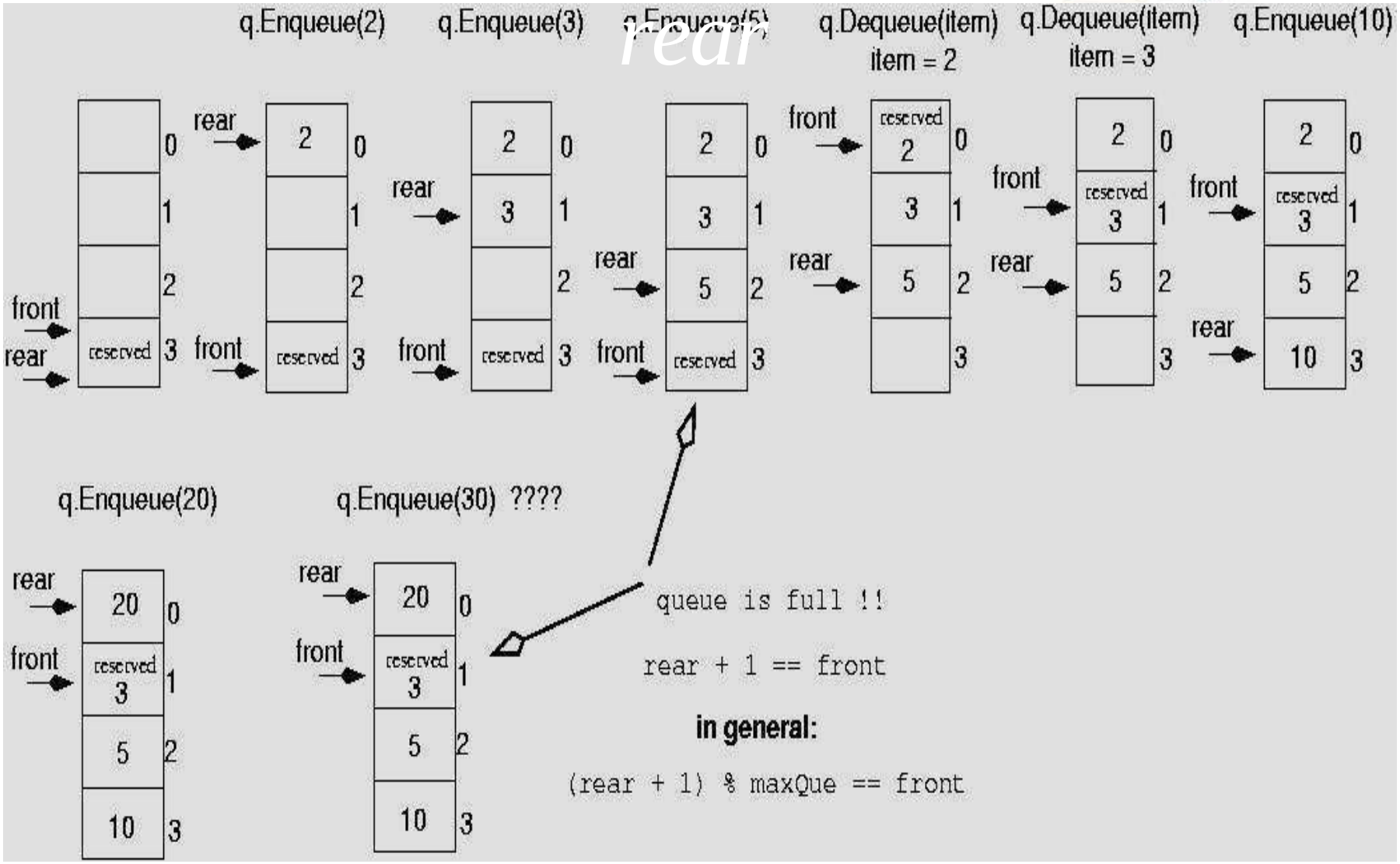
We cannot distinguish between the two cases !!!

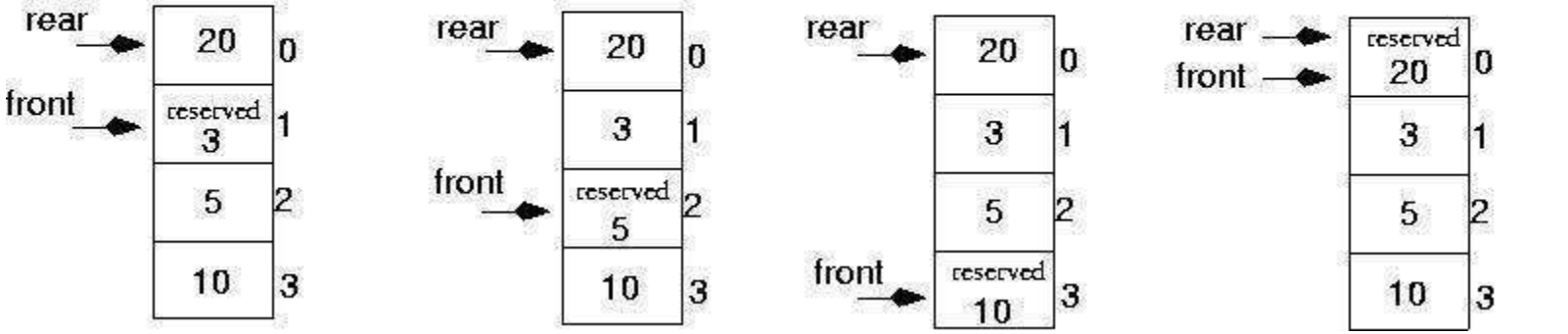
Make *front* point to the element **preceding** the front element in the queue (one memory location will be wasted).



Based on this solution, one memory location is wasted !!!

Initialize *front* and *rear*




rear
front
20
reserved
3
5
10
0
1
2
3
q.Dequeue(item)
item = 5
rear
front
20
3
reserved
5
10
0
1
2
3
q.Dequeue(item)
item = 10
rear
front
20
3
5
reserved
10
0
1
2
3
q.Dequeue(item)
item = 20
rear
front
reserved
20
3
5
10
0
1
2
3

Queue Implementation

```
template<class ItemType>
class  QueueType  {
public:
    QueueType(int);
    QueueType();
    ~QueueType();
    void MakeEmpty();
    bool IsEmpty() const;
    bool IsFull() const;
    void Enqueue(ItemType);
    void Dequeue(ItemType&);
```

Queue Implementation (cont.)

```
template<class ItemType>
QueueType<ItemType>::QueueType(int max)
{
 maxQue = max + 1;
 front = maxQue - 1;
 rear = maxQue - 1;
 items = new ItemType[maxQue];
}
```

Queue Implementation (cont.)

```
template<class ItemType>
QueueType<ItemType>::~QueueType()
{
 delete [] items;
}
```

Queue Implementation (cont.)

```
template<class ItemType>
void QueueType<ItemType>:: MakeEmpty()
{
 front = maxQue - 1;
 rear = maxQue - 1;
}
```

Queue Implementation (cont.)

```
template<class ItemType>
bool QueueType<ItemType>::IsEmpty() const
{
 return (rear == front);
}

template<class ItemType>
bool QueueType<ItemType>::IsFull() const
{
 return ( (rear + 1) % maxQue == front);
}
```

Queue Implementation (cont.)

```
template<class ItemType>
void QueueType<ItemType>::Enqueue (ItemType
   newItem)
{
 rear = (rear + 1) % maxQue;
 items[rear] = newItem;
}
```

Queue Implementation (cont.)

```
template<class ItemType>
void QueueType<ItemType>::Dequeue (ItemType&
   item)
{
 front = (front + 1) % maxQue;
 item = items[front];
}
```

Queue overflow

- The condition resulting from trying to add an element onto a full queue.

```
if(!q.IsFull())
  q.Enqueue(item);
```

Queue underflow

- The condition resulting from trying to remove an element from an empty queue.

```
if(!q.IsEmpty())
  q.Dequeue(item);
```

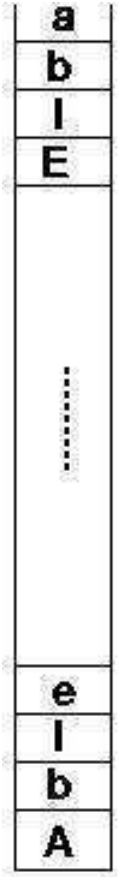
Example: recognizing palindromes

- A *palindrome* is a string that reads the same forward and backward.

  Able was I ere I saw Elba

- We will read the line of text into both a stack and a queue.
- Compare the contents of the stack and the queue character-by-character to see if they would produce the same string of characters.

Example: recognizing palindromes



Stack

Queue

Example: recognizing palindromes

```
#include <iostream.h>
#include <ctype.h>
#include "stack.h"
#include "queue.h“
int main()
{
 StackType<char> s;
 QueType<char> q;
 char ch;
 char sItem, qItem;
 int mismatches = 0;
```

Example: recognizing palindromes

```
while( (!q.IsEmpty()) && (!s.IsEmpty()) ) {

  s.Pop(sItem);

  q.Dequeue(qItem);

  if(sItem != qItem)
    ++mismatches;
}
if (mismatches == 0)
  cout << "That is a palindrome" << endl;
else
 cout << That is not a palindrome" << endl;

return 0;
}
```

Priority Queues

- Review the abstract data type Priority Queues
- Review different implementation options

Abstract Data Type: Priority Queue

- A priority queue is a *collection* of zero or more items,
  - associated with each item is a priority

- A priority queue has at least three operations
  - *insert*(item i) (enqueue) a new item
  - *delete*() (dequeue) the member with the highest priority
  - *find*() the item with the highest priority
  - decreasePriority(item i, p) decrease the priority of ith item to p

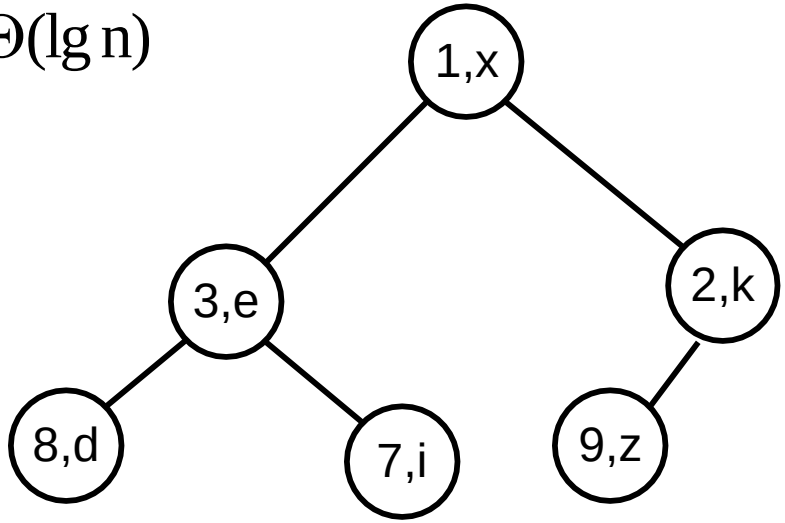
- Note that in a priority queue "*first in first out*" does not apply in general.

Priority Queues: Assumptions

- The highest priority can be either the *minimum* value of all the *items*, or the *maximum*.
  - We will assume the highest priority is the *minimum*.
  - Call the delete operation *deleteMin*().
  - Call the find operation *findMin*().

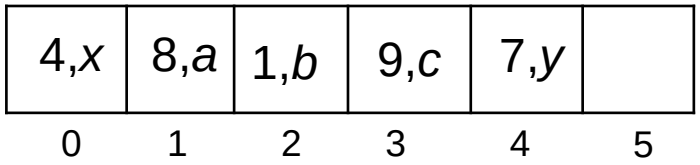
- Assume the priority queue has *n* members

- Heap.
  - In the worst case *insert*() is Θ(lg n) and
  - *deleteMin*() is Θ(lg n)
  - *findMin*() is Θ(1)
  - decreaseKey(i, p) is Θ(lg n)



Unsorted list: Array

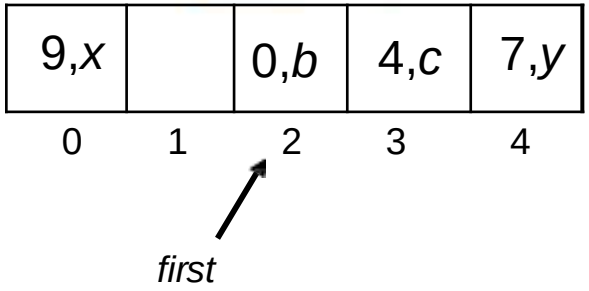
1. Using an array *arr*.
   - *insert*() adds the new *item* into next empty position in *arr*, in Θ(1).
   - *findMin*() is Θ(n) in the worst case
   - *deleteMin*() is Θ(n) in the worst case
     - Θ(n) to find the minimum *item*
     - and Θ (*1*) to move the last *item* to the position of the deleted element.
   - DecreasePriority(i, p) – decrease priority of ith item stored at *arr*[i] in Θ(1)



Unsorted list: Linked List

2. Using a linked list.
   - *insert*() in $\Theta(1)$ with appropriate pointers.
   - *findMin*() is $\Theta(n)$ since we may need to search the whole list.
   - *deleteMin*() is $\Theta(n)$
     - In the worst case we may need to search the whole list, $\Theta(n)$
     - Delete *item*, $\Theta(1)$

Sorted list: Circular Array



1. A circular array A.
   - *insert*() must maintain a sorted list.
     - Θ(n) in the worst case
     - For example:

       The new *item* needs to be inserted after the *item* with the highest priority. So *n*-1 *items* have to be moved to make room.
   - *findMin*() is Θ(1)
   - *deleteMin*() is Θ(1) because the minimum *item* is the first one in the queue, and only the *pointer* to the first *item* needs to be changed.
   - DecreasePriority(i, p) – decrease priority of *i*th item, and reinsert Θ(n)

Sorted list: Linked List

2. A linked list.
   - *insert*() is Θ(n)
     - since in the worst case the whole list must be searched sequentially to find the location for insertion.
   - *findMin*() is Θ(1)
   - *deleteMin* is Θ(1)
     - since with appropriate pointers the first element of a linked list can be deleted in Θ(1).

Priority Queue Implementations

Data Structure	*insert* worst case	*DeleteMin* worst case
Heap	$\Theta(\lg n)$	$\Theta(\lg n)$
Unsorted (array or linked list)	$\Theta(1)$	$\Theta(n)$
Sorted (array or linked list)	$\Theta(n)$	$\Theta(1)$

Backtracking- The Maze Problem

A short list of categories

- Algorithm types we will consider include:
  - Simple recursive algorithms
  - Backtracking algorithms
  - Divide and conquer algorithms
  - Dynamic programming algorithms
  - Greedy algorithms
  - Branch and bound algorithms
  - Brute force algorithms
  - Randomized algorithms

Backtracking

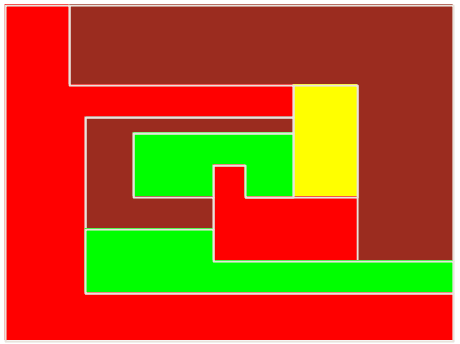
- Suppose you have to make a series of *decisions*, among various *choices*, where
  - You don't have enough information to know what to choose
  - Each decision leads to a new set of choices
  - Some sequence of choices (possibly more than one) may be a solution to your problem
- Backtracking is a methodical way of trying out various sequences of decisions, until you find one that "works"

Solving a maze

- Given a maze, find a path from start to finish
- At each intersection, you have to decide between three or fewer choices:
  - Go straight
  - Go left
  - Go right
- You don't have enough information to choose correctly
- Each choice leads to another set of choices
- One or more sequences of choices may (or may not) lead to a solution
- Many types of maze problem can be solved with backtracking

Coloring a map

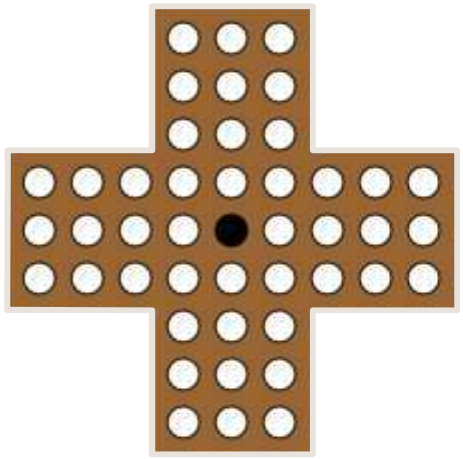
- You wish to color a map with not more than four colors
  - red, yellow green, blue
- Adjacent countries must be in different colors



- You don't have enough information to choose colors
- Each choice leads to another set of choices
- One or more sequences of choices may (or may not) lead to a solution
- Many coloring problems can be solved with backtracking

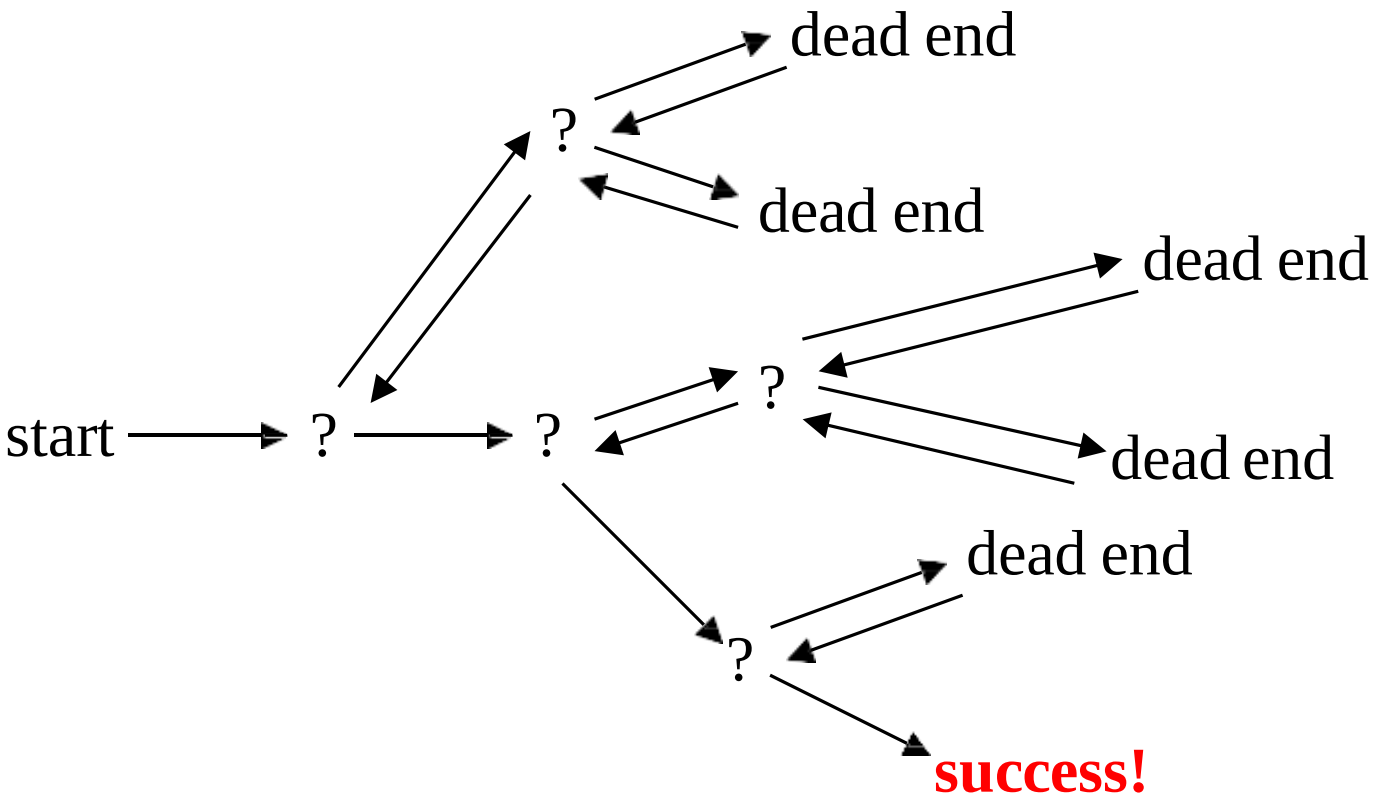
Solving a puzzle

- In this puzzle, all holes but one are filled with white pegs
- You can jump over one peg with another
- Jumped pegs are removed
- The object is to remove all but the last peg



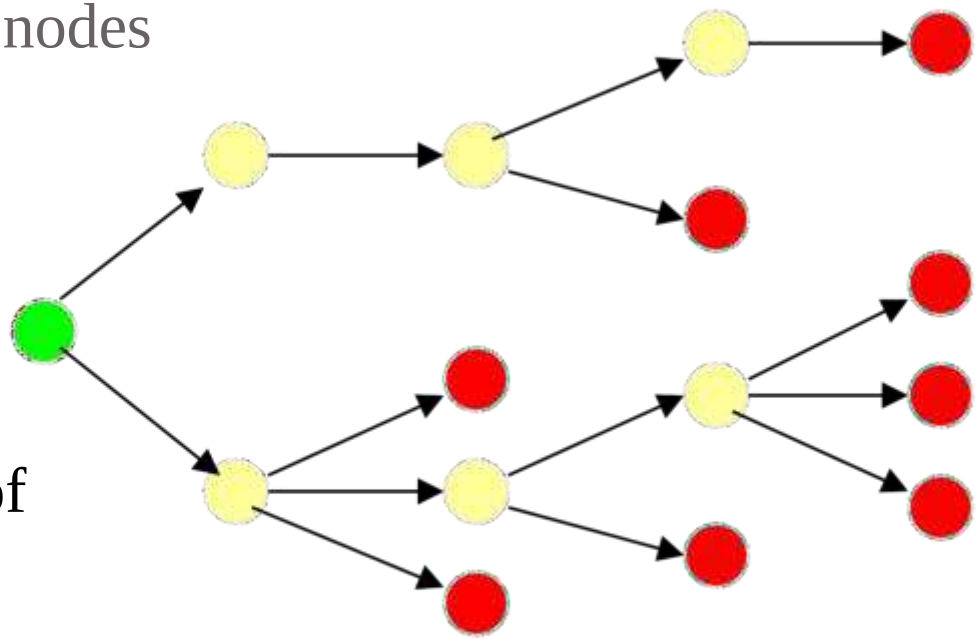
- You don't have enough information to jump correctly
- Each choice leads to another set of choices
- One or more sequences of choices may (or may not) lead to a solution
- Many kinds of puzzle can be solved with backtracking

Backtracking (animation)



Terminology I

A tree is composed of nodes



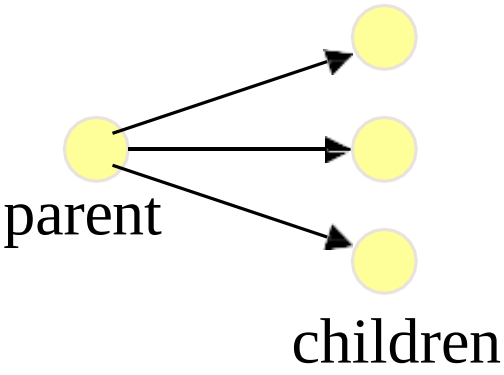
There are three kinds of nodes:

- The (one) root node
- Internal nodes
- Leaf nodes

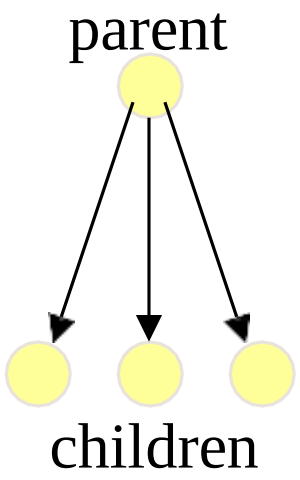
Backtracking can be thought of as searching a tree for a particular "goal" leaf node

Terminology II

- Each non-leaf node in a tree is a parent of one or more other nodes (its children)
- Each node in the tree, other than the root, has exactly one parent



Usually, however, we draw our trees *downward*, with the root at the top



Real and virtual trees

- There is a type of data structure called a tree
  - But we are **not** using it here
- If we diagram the sequence of choices we make, the diagram looks like a tree
  - In fact, we did just this a couple of slides ago
  - Our backtracking algorithm "sweeps out a tree" in "problem space"

The backtracking algorithm

- Backtracking is really quite simple--we "explore" each node, as follows:
- To "explore" node N:
  1. If N is a goal node, return "success"
  2. If N is a leaf node, return "failure"
  3. For each child C of N,
     1. Explore C
        1. If C was successful, return "success"
  4. Return "failure"

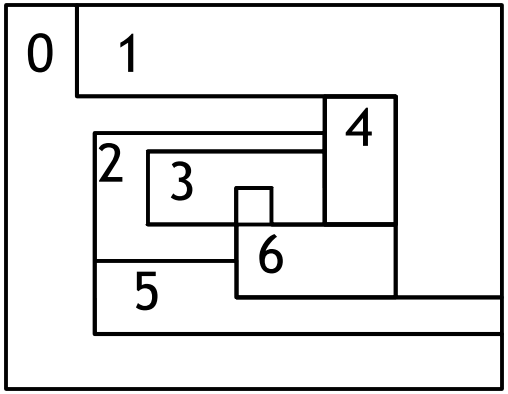
Full example: Map coloring

- The Four Color Theorem states that anymap on aplane can be colored with no more than four colors, so that no two countries with a common border are the same color
- For most maps, finding a legal coloring is easy
- For some maps, it can be fairly difficult to find a legal coloring
- We will develop a complete Java program to solve this problem

Data structures

- We need a data structure that is easy to work with, and supports:
  - Setting a color for each country
  - For each country, finding all adjacent countries
- We can do this with two arrays
  - An array of "colors", where `countryColor[i]` is the color of the i^{th} country
  - A ragged array of adjacent countries, where `map[i][j]` is the j^{th} country adjacent to country i
    - Example: `map[5][3]==8` means the 3^{th} country adjacent to country 5 is country 8

Creating the map




```
int map[][];

void createMap() {
    map = new int[7][];
    map[0] = new int[] { 1, 4, 2, 5 };
    map[1] = new int[] { 0, 4, 6, 5 };
    map[2] = new int[] { 0, 4, 3, 6, 5 };
    map[3] = new int[] { 2, 4, 6 };
    map[4] = new int[] { 0, 1, 6, 3, 2 };
    map[5] = new int[] { 2, 6, 1, 0 };
    map[6] = new int[] { 2, 3, 4, 1, 5 };
}
```

Setting the initial colors

```
static final int NONE = 0;
static final int RED = 1;
static final int YELLOW = 2;
static final int GREEN = 3;
static final int BLUE = 4;

int mapColors[] = { NONE, NONE, NONE, NONE,
                    NONE, NONE, NONE };
```

The main program

(The name of the enclosing class is ColoredMap)

```
public static void main(String args[]) {
    ColoredMap m = new ColoredMap();
    m.createMap();
    boolean result = m.explore(0, RED);
    System.out.println(result);
    m.printMap();
}
```

The backtracking method

```
boolean explore(int country, int color) {
    if (country >= map.length) return true;
    if (okToColor(country, color)) {
        mapColors[country] = color;
        for (int i = RED; i <= BLUE; i++) {
            if (explore(country + 1, i)) return true;
        }
    }
    return false;
}
```

Checking if a color can be used

```
boolean okToColor(int country, int color) {
    for (int i = 0; i < map[country].length; i++) {
        int ithAdjCountry = map[country][i];
        if (mapColors[ithAdjCountry] == color) {
            return false;
        }
    }
    return true;
}
```

Printing the results

```
void printMap() {
    for (int i = 0; i < mapColors.length; i++) {
        System.out.print("map[" + i + "] is ");
        switch (mapColors[i]) {
            case NONE:   System.out.println("none");   break;
            case RED:    System.out.println("red");    break;
            case YELLOW: System.out.println("yellow"); break;
            case GREEN:  System.out.println("green");  break;
            case BLUE:   System.out.println("blue");   break;
        }
    }
}
```

Summary

- We went through all the countries recursively, starting with country zero
- At each country we had to decide a color
  - It had to be different from all adjacent countries
  - If we could not find a legal color, we reported failure
  - If we could find a color, we used it and recurred with the next country
  - If we ran out of countries (colored them all), we reported success
- When we returned from the topmost call, we were done

Recursion

What is recursion?

- Sometimes, the best way to solve a problem is by solving a smaller version of the exact same problem first
- Recursion is a technique that solves a problem by solving a smaller problem of the same type

```
int f(int x)
{
 int y;

 if(x==0)
    return 1;
 else {
    y = 2 * f(x-1);
    return y+1;
 }
}
```

Problems defined recursively

- There are many problems whose solution can be defined recursively

Example: *n factorial*

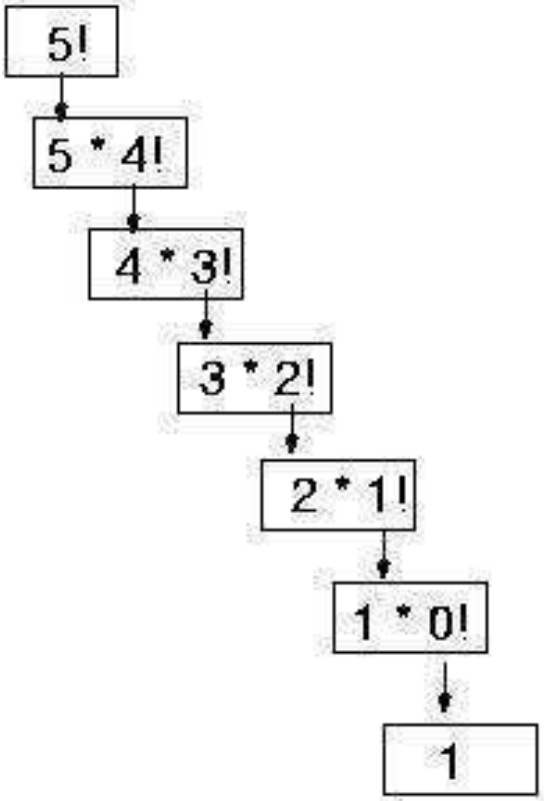
$$n! = \begin{cases} 1 & \text{if } n = 0 \\ (n-1)!*n & \text{if } n > 0 \end{cases} \quad (\textit{recursive} \text{ solution})$$

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ 1*2*3*\ldots*(n-1)*n & \text{if } n > 0 \end{cases} \quad (\textit{closed form} \text{ solution})$$

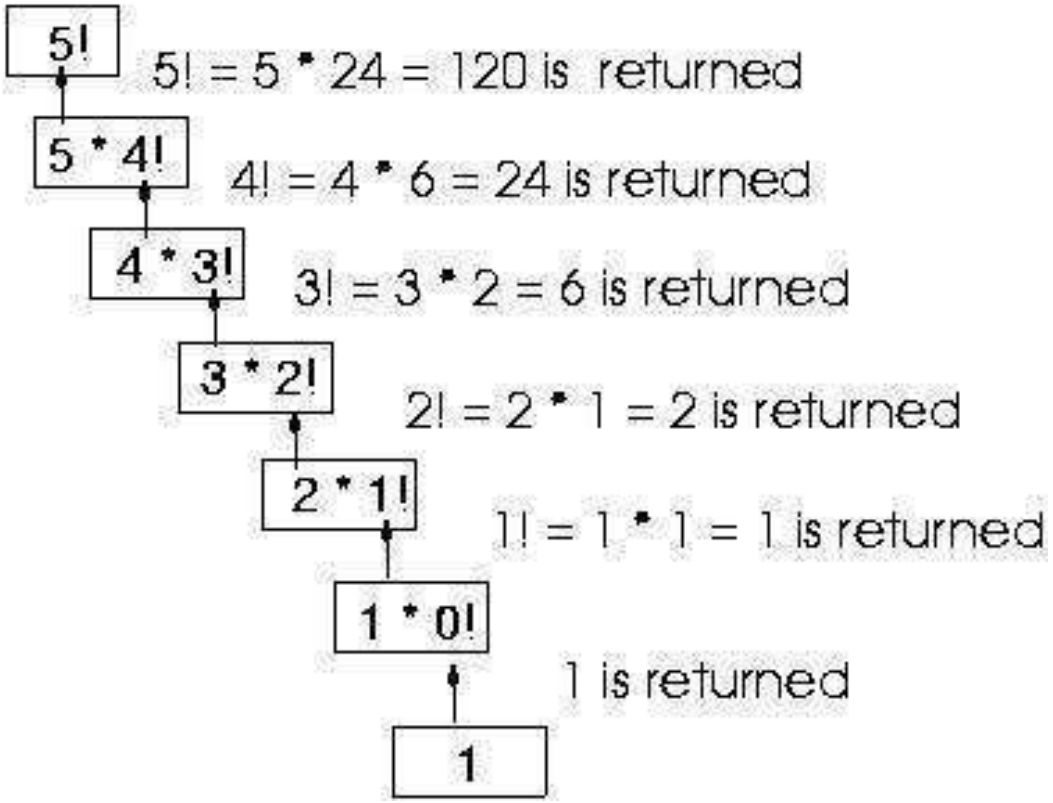
Coding the factorial function

- Recursive implementation

```
int Factorial(int n)
{
 if (n==0)  // base case
   return 1;
 else
   return n * Factorial(n-1);
}
```



Final value = 120



Coding the factorial function (cont.)

- Iterative implementation

```
int Factorial(int n)
{
 int fact = 1;

 for(int count = 2; count <= n; count++)
   fact = fact * count;

 return fact;
}
```

Another example:

n choose *k* (combinations)

- Given *n* things, how many different sets of size *k* can be chosen?

$$\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}, \quad 1 < k < n \qquad \textit{(recursive solution)}$$

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}, \quad 1 < k < n \qquad \textit{(closed-formsolution)}$$

with base cases:

$$\binom{n}{1} = n \ (k = 1), \quad \binom{n}{n} = 1 \ (k = n)$$

n choose *k* (combinations)

```
int Combinations(int n, int k)
{
 if(k == 1)  // base case 1
   return n;
 else if (n == k)  // base case 2
   return 1;
 else
   return(Combinations(n-1, k) + Combinations(n-1, k-1));
}
```

Key Components of a Recursive Algorithm Design

1. What is a smaller ***identical*** problem(s)?
   - Decomposition
2. How are the answers to smaller problems combined to form the answer to the larger problem?
   - Composition
3. Which is the smallest problem that can be solved easily (without further decomposition)?
   - Base/s topping case

Examples in Recursion

- Usually quite confusing the first time
- Start with some simple examples
  - recursive algorithms might not be best
- Later with inherently recursive algorithms
  - harder to implement otherwise

Factorial (*N*!)

- *N*! = (*N*-1)! * N [for *N* > 1]
- 1! = 1
- 3!
  = 2! * 3
  = (1! * 2) * 3
  = 1 * 2 * 3
- Recursive design:
  - Decomposition: (*N*-1)!
  - Composition: * *N*
  - Base case: 1!

factorial Method

```
public static int factorial(int n)
{
  int fact;
  if (n > 1) // recursive case (decomposition)
    fact = factorial(n - 1) * n; // composition
  else // base case
    fact = 1;

  return fact;
}
```

```
public static int factorial(int 3)
{
  int fact;
  if (n > 1)
    fact = factorial(2) * 3;
  else
    fact = 1;
  return fact;
}
```

```
public static int factorial(int 3)
{
  int fact;
  if (n > 1)
    fact = factorial(2) * 3;
  else
    fact = 1;
  return fact;
}
```

```
public static int factorial(int 2)
{
  int fact;
  if (n > 1)
    fact = factorial(1) * 2;
  else
    fact = 1;
  return fact;
}
```

```
public static int factorial(int 3)
{
  int fact;
  if (n > 1)
    fact = factorial(2) * 3;
  else
    fact = 1;
  return fact;
}
```

```
public static int factorial(int 2)
{
  int fact;
  if (n > 1)
    fact = factorial(1) * 2;
  else
    fact = 1;
  return fact;
}
```

```
public static int factorial(int 1)
{
  int fact;
  if (n > 1)
    fact = factorial(n - 1) * n;
  else
    fact = 1;
  return fact;
}
```

```
public static int factorial(int 3)
{
  int fact;
  if (n > 1)
    fact = factorial(2) * 3;
  else
    fact = 1;
  return fact;
}
```

```
public static int factorial(int 2)
{
  int fact;
  if (n > 1)
    fact = factorial(1) * 2;
  else
    fact = 1;
  return fact;
}
```

```
public static int factorial(int 1)
{
  int fact;
  if (n > 1)
    fact = factorial(n - 1) * n;
  else
    fact = 1;
  return 1;
}
```

```
public static int factorial(int 3)
{
  int fact;
  if (n > 1)
    fact = factorial(2) * 3;
  else
    fact = 1;
  return fact;
}
```

```
public static int factorial(int 2)
{
  int fact;
  if (n > 1)
    fact = 1 * 2;
  else
    fact = 1;
  return fact;
}
```

```
public static int factorial(int 1)
{
  int fact;
  if (n > 1)
    fact = factorial(n - 1) * n;
  else
    fact = 1;
  return 1;
}
```

```
public static int factorial(int 3)
{
  int fact;
  if (n > 1)
    fact = factorial(2) * 3;
  else
    fact = 1;
  return fact;
}
```

```
public static int factorial(int 2)
{
  int fact;
  if (n > 1)
    fact = 1 * 2;
  else
    fact = 1;
  return 2;
}
```

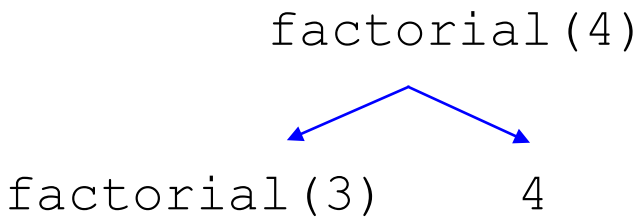
```
public static int factorial(int 3)
{
  int fact;
  if (n >  1)
    fact = 2 * 3;
  else
    fact = 1;
  return fact;
}
```

```
public static int factorial(int 2)
{
  int fact;
  if (n > 1)
    fact = 1 * 2;
  else
    fact = 1;
  return 2;
}
```

```
public static int factorial(int 3)
{
  int fact;
  if (n >  1)
    fact = 2 * 3;
  else
    fact = 1;
  return 6;
}
```

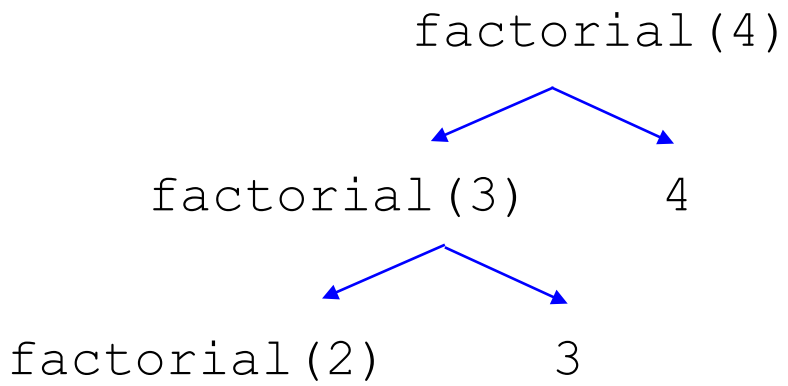
Trace (decomposition)

```
public static int factorial(int n)
{
  int fact;
  if (n > 1) // recursive case (decomposition)
    fact = factorial(n - 1) * n; (composition)
  else // base case
    fact = 1;
  return fact;
}
```



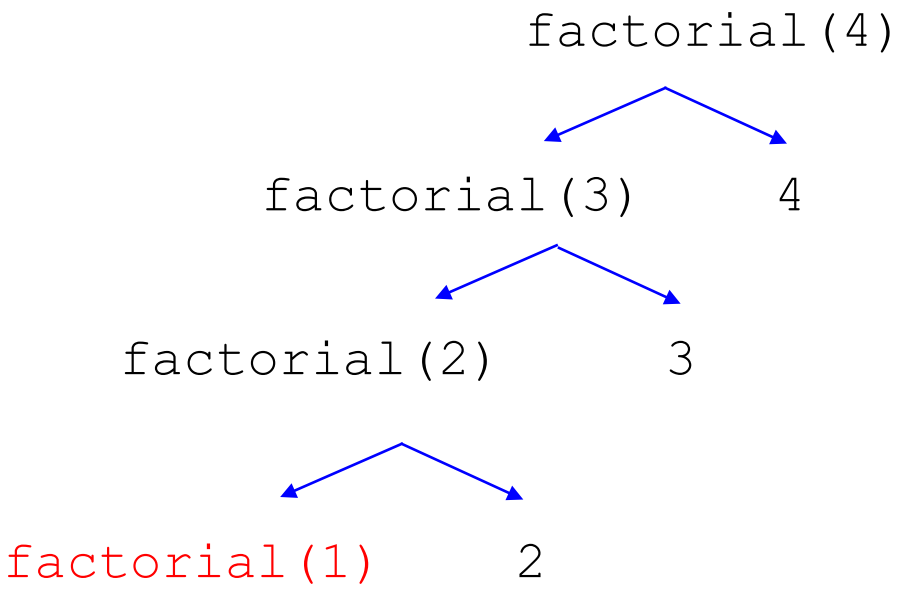
Trace (decomposition)

```
public static int factorial(int n)
{
  int fact;
  if (n > 1) // recursive case (decomposition)
    fact = factorial(n - 1) * n; (composition)
  else // base case
    fact = 1;
  return fact;
}
```



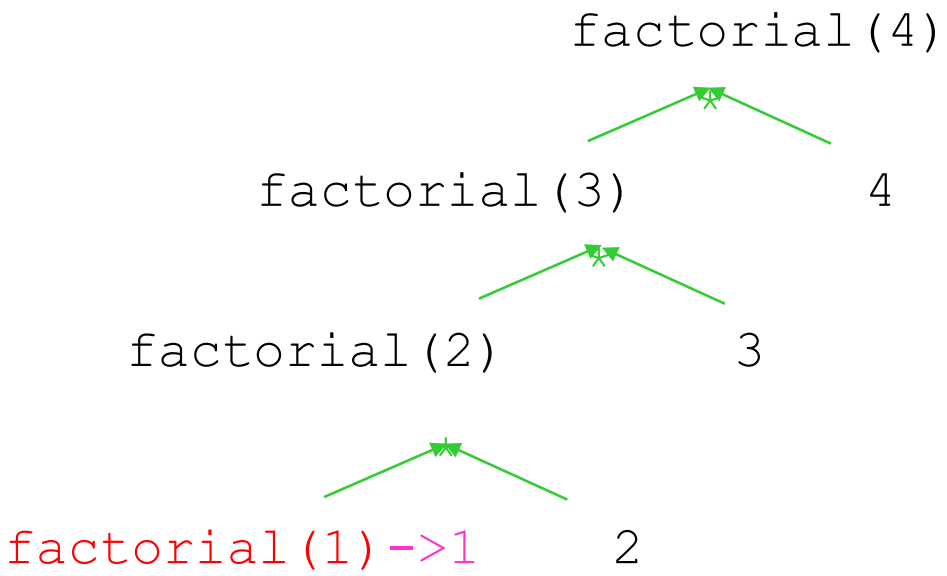
Trace (decomposition)

```
public static int factorial(int n)
{
  int fact;
  if (n > 1) // recursive case (decomposition)
    fact = factorial(n - 1) * n; (composition)
  else // base case
    fact = 1;
  return fact;
}
```



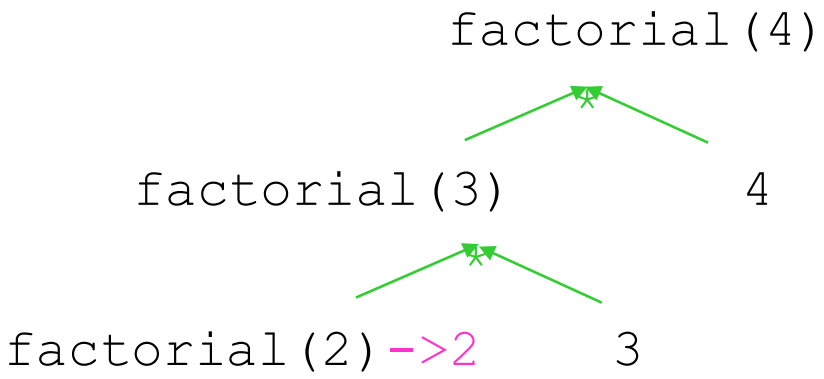
Trace (composition)

```
public static int factorial(int n)
{
  int fact;
  if (n > 1) // recursive case (decomposition)
    fact = factorial(n - 1) * n; (composition)
  else // base case
    fact = 1;
  return fact;
}
```



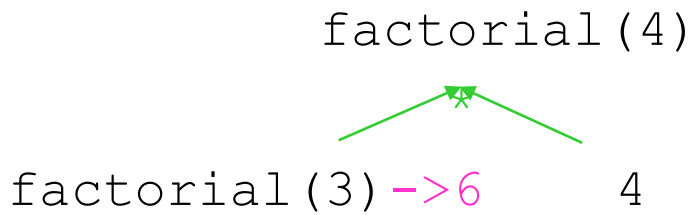
Trace (composition)

```
public static int factorial(int n)
{
  int fact;
  if (n > 1) // recursive case (decomposition)
    fact = factorial(n - 1) * n; (composition)
  else // base case
    fact = 1;
  return fact;
}
```



Trace (composition)

```
public static int factorial(int n)
{
  int fact;
  if (n > 1) // recursive case (decomposition)
    fact = factorial(n - 1) * n; (composition)
  else // base case
    fact = 1;
  return fact;
}
```



Trace (composition)

```
public static int factorial(int n)
{
  int fact;
  if (n > 1) // recursive case (decomposition)
    fact = factorial(n - 1) * n; (composition)
  else // base case
    fact = 1;
  return fact;
}
```

factorial(4)->24

Improved factorial Method

```
public static int factorial(int n)
{
  int fact=1;  // base case value

  if (n > 1)   // recursive case (decomposition)
    fact = factorial(n - 1) * n; // composition
  // else do nothing; base case

  return fact;
}
```

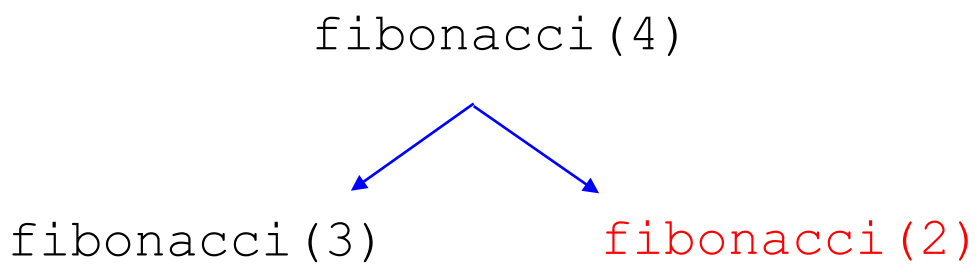
Fibonacci Numbers

- The *N*th Fibonacci number is the sum of the previous two Fibonacci numbers
- 0, 1, 1, 2, 3, 5, 8, 13, …
- Recursive Design:
  - Decomposition & Composition
    - fibonacci(n) = fibonacci(n-1) + fibonacci(n-2)
  - Base case:
    - fibonacci(1) = 0
    - fibonacci(2) = 1

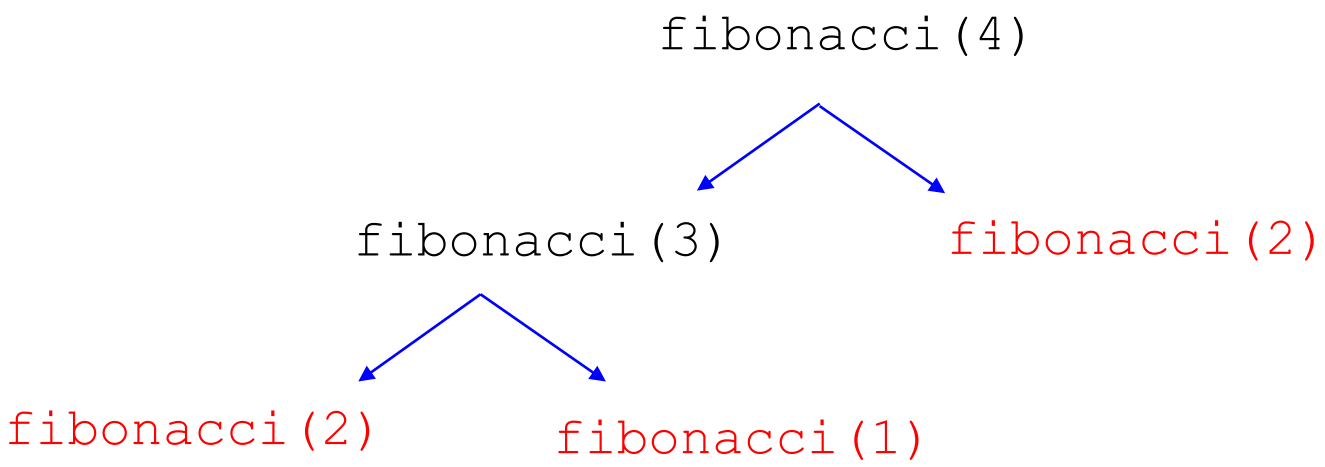
fibonacci Method

```
public static int fibonacci(int n)
{
   int fib;
   if (n > 2)
     fib = fibonacci(n-1) + fibonacci(n-2);
   else if (n == 2)
     fib = 1;
   else
     fib = 0;
   return fib;
}
```

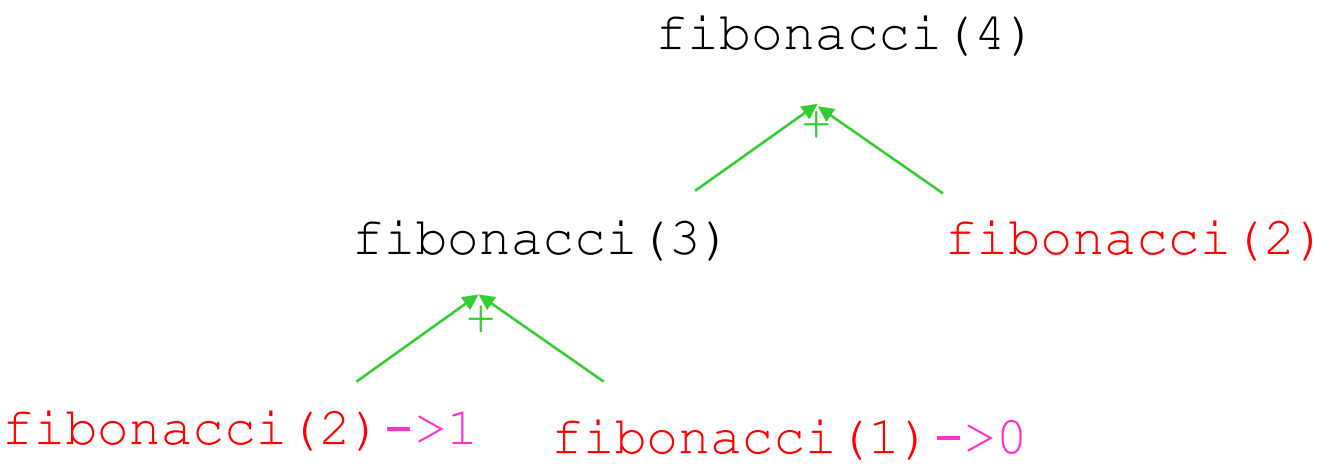
Execution Trace (decomposition)



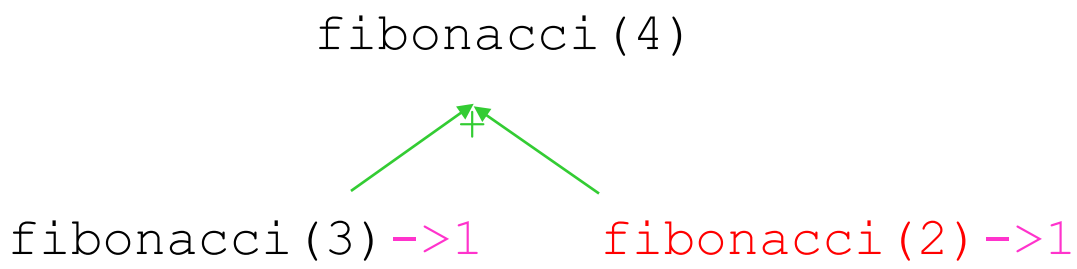
Execution Trace (decomposition)



Execution Trace (composition)



Execution Trace (composition)



Execution Trace (composition)

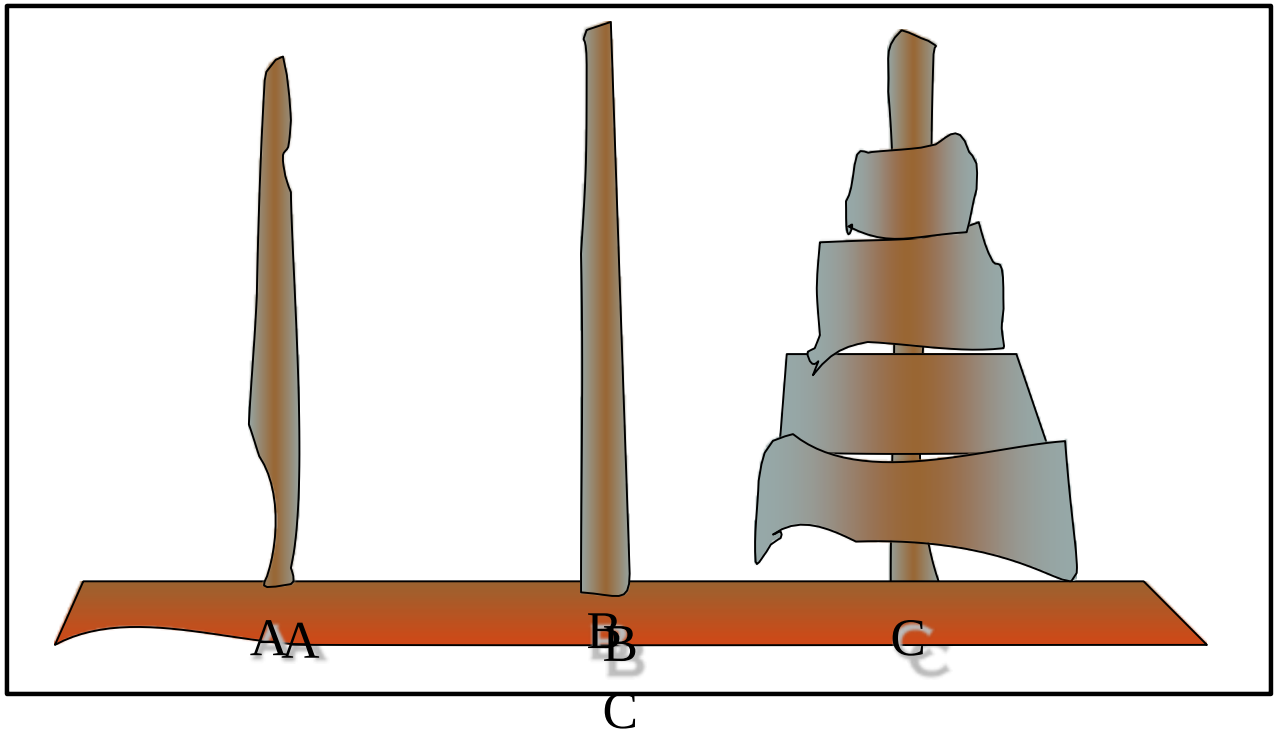
```
fibonacci(4)->2
```

Remember: Key to Successful Recursion

- if-else statement (or some other branching statement)
- Some branches: recursive call
  - "smaller" arguments or solve "smaller" versions of the same task (*decomposition*)
  - Combine the results (*composition*) [if necessary]
- Other branches: no recursive calls
  - stopping cases or base cases

Towers of Hanoi

- Move n (4) disks from pole A to pole C
- such that a disk is never put on a smaller disk



- Move n (4) disks from A to C
  - Move n-1 (3) disks from A to B
  - Move 1 disk from A to C
  - Move n-1 (3) disks from B to C

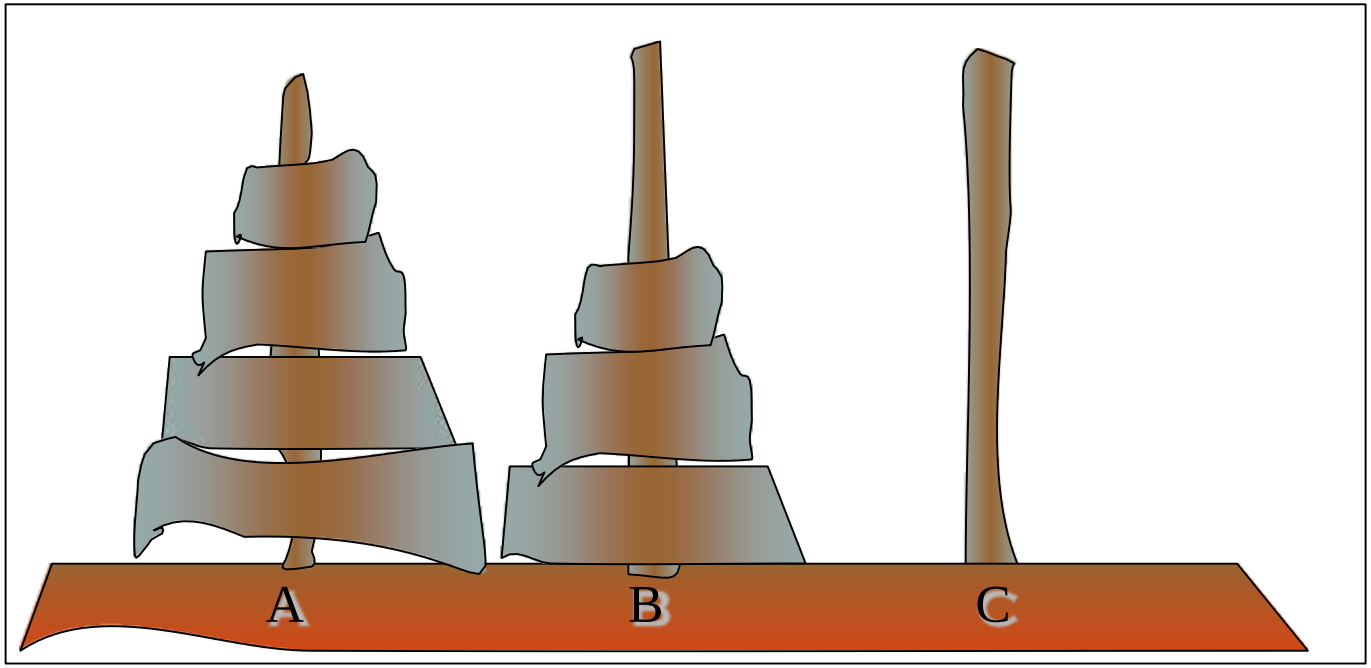


Figure 2.19a and b

a) The initial state; b) move *n* - 1 disks from *A* to *C*

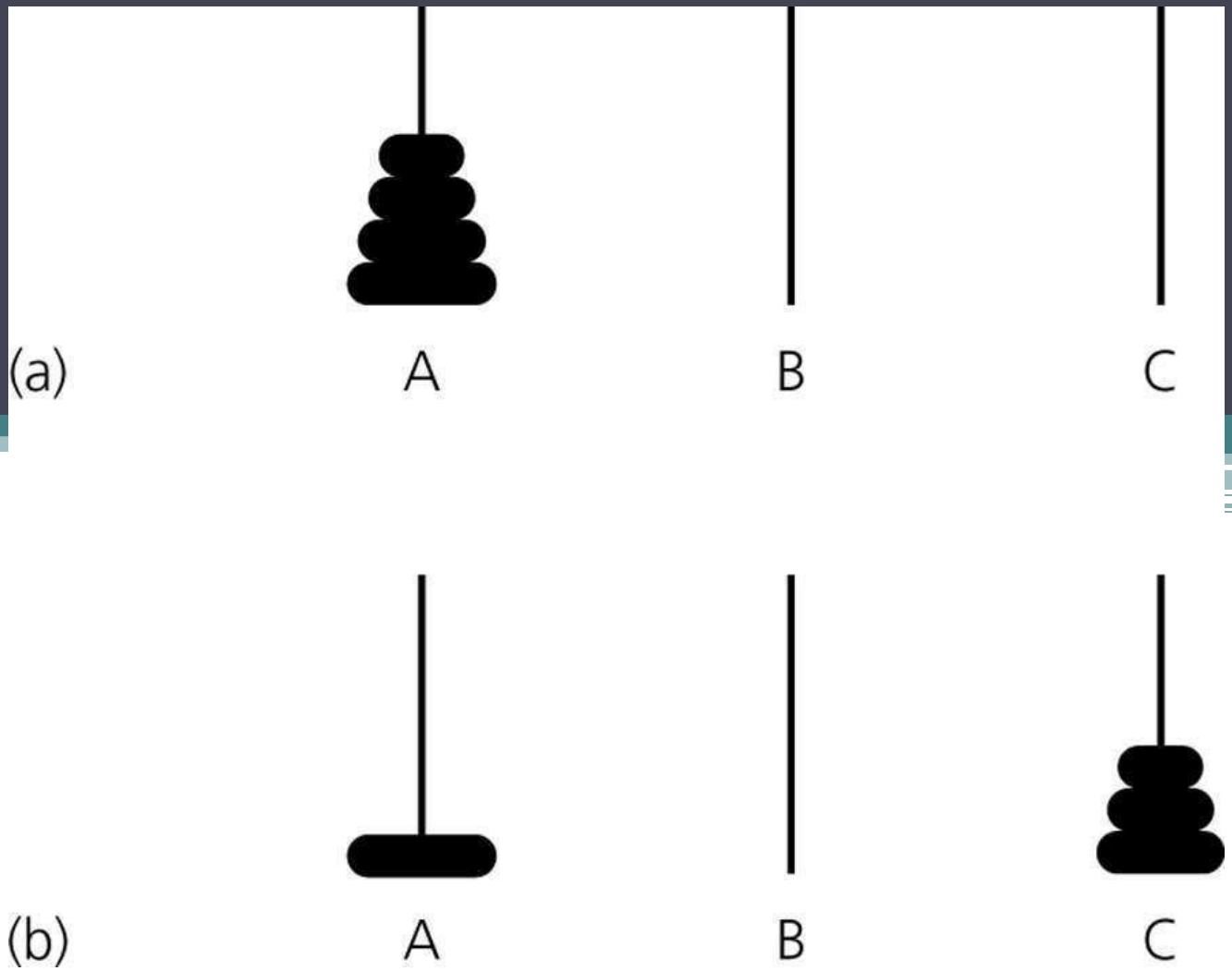
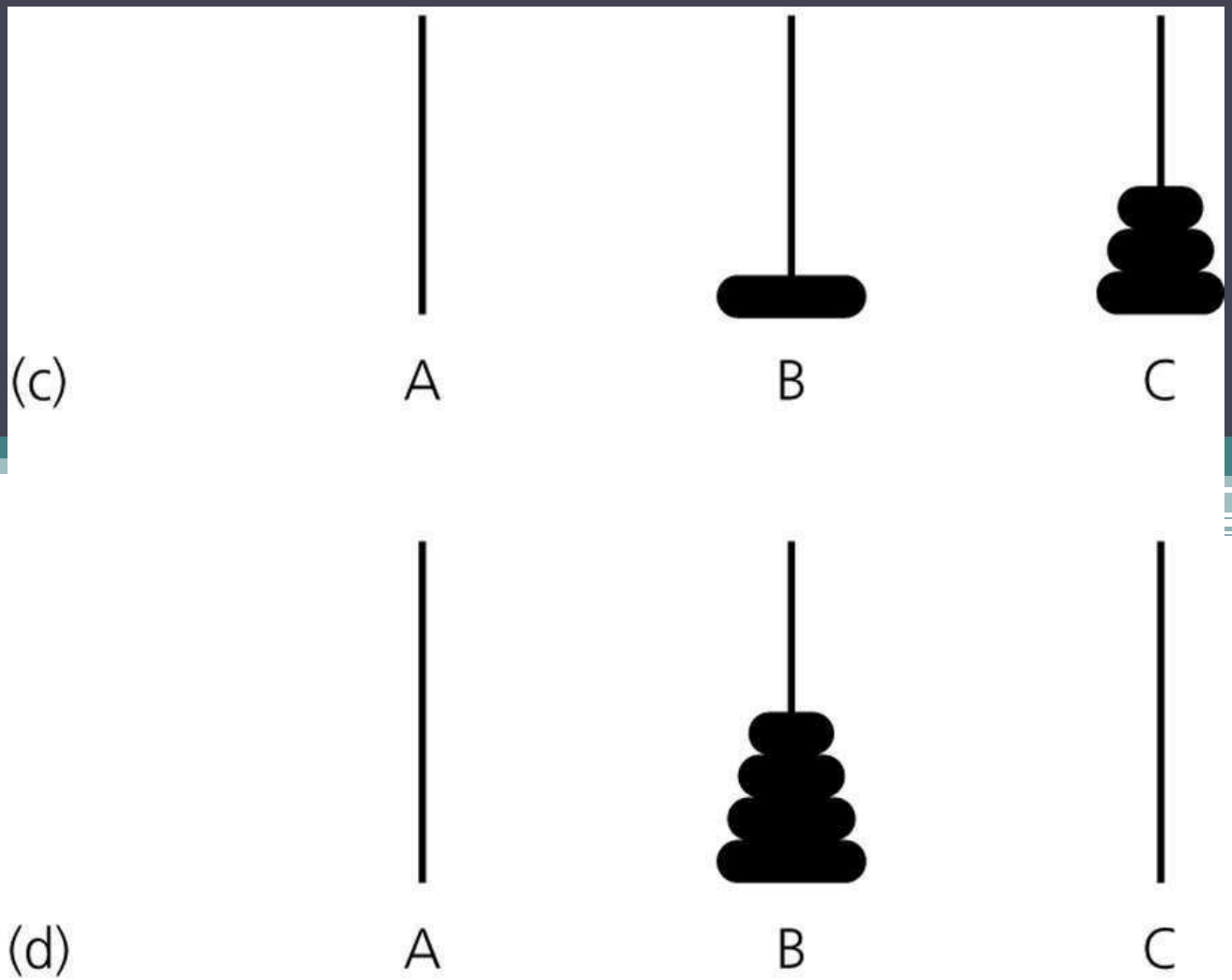


Figure 2.19c and d

c) move one disk from *A* to *B*; d) move *n* - 1 disks from *C* to *B*



Hanoi towers

```
public static void solveTowers(int count, char source,
                               char destination, char spare) {
  if (count == 1) {
    System.out.println("Move top disk from pole " + source +
                       " to pole " + destination);
  }
  else {
    solveTowers(count-1, source, spare, destination); // X
    solveTowers(1, source, destination, spare);       // Y
    solveTowers(count-1, spare, destination, source); // Z
  }  // end if
}  // end solveTowers
```

Recursion tree:

The order of recursive calls that results from ***solveTowers(3,A,B,C)***

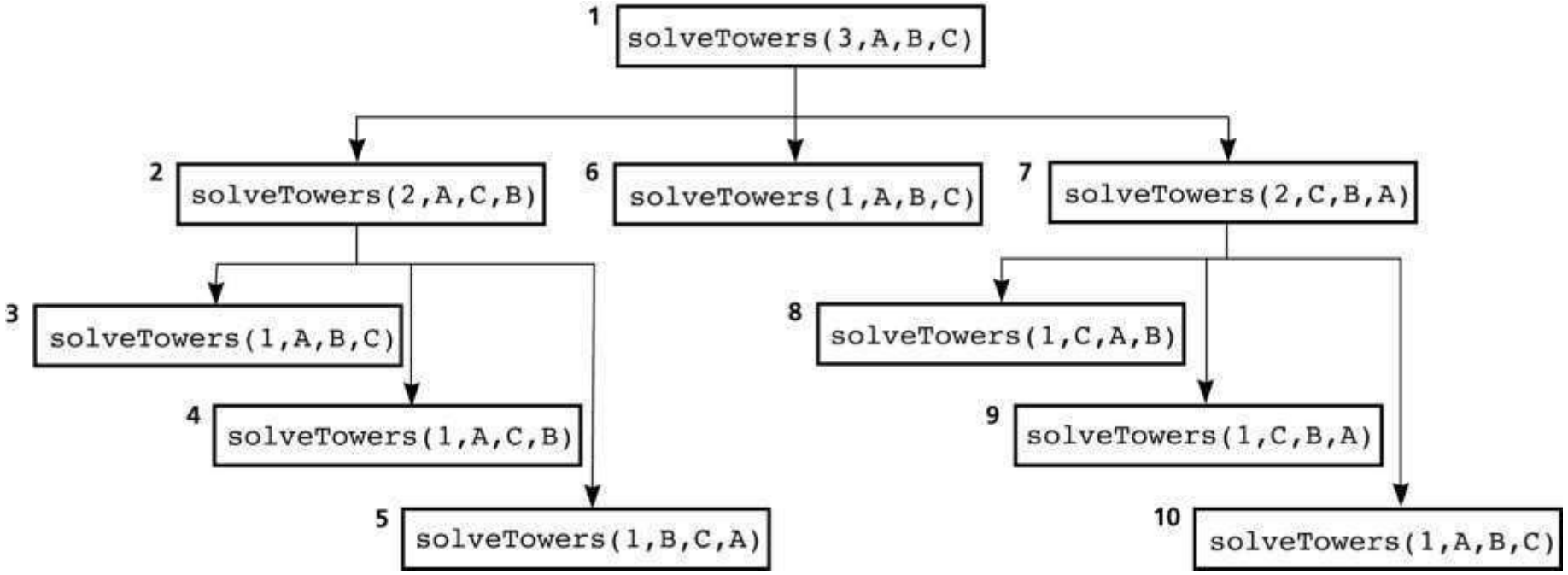
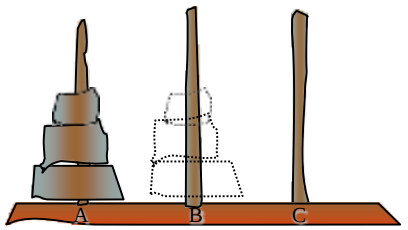


Figure 2.21a

Box trace of ***solveTowers(3, 'A', 'B', 'C')***

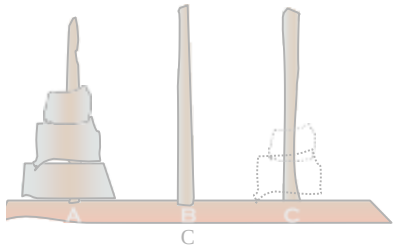
The initial call 1 is made, and `solveTowers` begins execution:

```
count  = 3
source = A
dest   = B
spare  = C
```



At point X, recursive call 2 is made, and the new invocation of the method begins execution:

```
count  = 3        X    count  = 2
source = A             source = A
dest   = B             dest   = C
spare  = C             spare  = B
```



At point X, recursive call 3 is made, and the new invocation of the method begins execution:

```
count  = 3        X    count  = 2        X    count  = 1
source = A             source = A             source = A
dest   = B             dest   = C             dest   = B
spare  = C             spare  = B             spare  = C
```

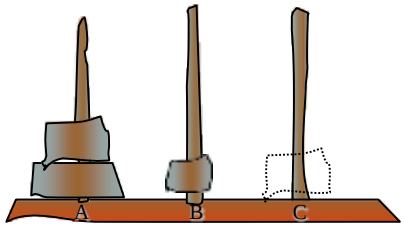

This is the base case, so a disk is moved, the return is made, and the method continues execution.

Figure 2.21b

Box trace of ***solveTowers(3, 'A', 'B', 'C')***

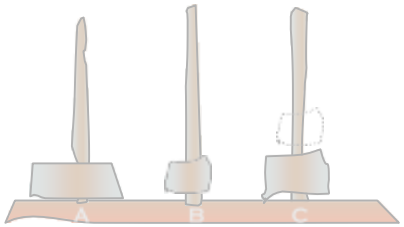
At point Y, recursive call 4 is made, and the new invocation of the method begins execution:

count = 3 source = A dest = B spare = C	X	count = 2 source = A dest = C spare = B	Y	count = 1 source = A dest = C spare = B



This is the base case, so a disk is moved, the return is made, and the method continues execution.

count = 3 source = A dest = B spare = C	X	count = 2 source = A dest = C spare = B	count = 1 source = A dest = C spare = B



At point Z, recursive call 5 is made, and the new invocation of the method begins execution:

count = 3 source = A dest = B spare = C	X	count = 2 source = A dest = C spare = B	Z	count = 1 source = B dest = C spare = A

This is the base case, so a disk is moved, the return is made, and the method continues execution.

count = 3 source = A dest = B spare = C	X	count = 2 source = A dest = C spare = B	count = 1 source = B dest = C spare = A

Figure 2.21c

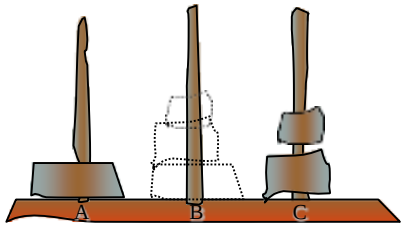
Box trace of ***solveTowers(3, 'A', 'B', 'C')***

This invocation completes, the return is made, and the method continues execution.

count	= 3
source	= A
dest	= B
spare	= C

count	= 2
source	= A
dest	= C
spare	= B

count	= 1
source	= B
dest	= C
spare	= A

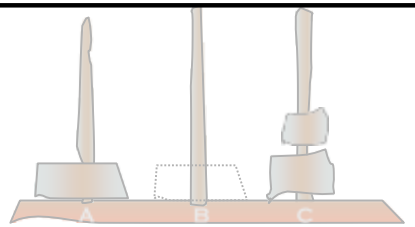


At point Y, recursive call 6 is made, and the new invocation of the method begins execution:

count	= 3
source	= A
dest	= B
spare	= C

Y →

count	= 1
source	= A
dest	= B
spare	= C



This is the base case, so a disk is moved, the return is made, and the method continues execution.

count	= 3
source	= A
dest	= B
spare	= C

count	= 1
source	= A
dest	= B
spare	= C

At point Z, recursive call 7 is made, and the new invocation of the method begins execution:

count	= 3
source	= A
dest	= B
spare	= C

Z →

count	= 2
source	= C
dest	= B
spare	= A

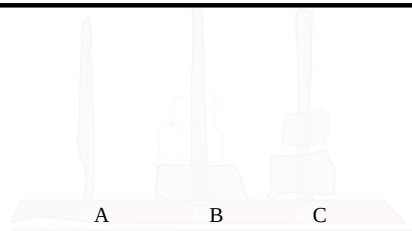


Figure 2.21d

Box trace of `solveTowers(3, 'A', 'B', 'C')`

At point X, recursive call 8 is made, and the new invocation of the method begins execution:

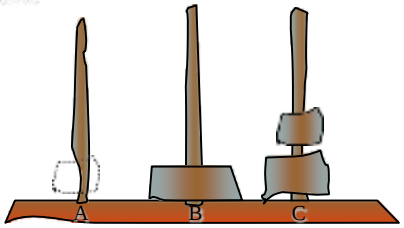
count	= 3
source	= A
dest	= B
spare	= C

Z →

count	= 2
source	= C
dest	= B
spare	= A

X →

count	= 1
source	= C
dest	= A
spare	= B



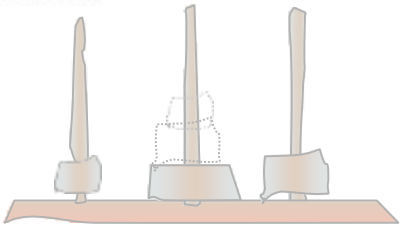
This is the base case, so a disk is moved, the return is made, and the method continues execution.

count	= 3
source	= A
dest	= B
spare	= C

Z →

count	= 2
source	= C
dest	= B
spare	= A

count	= 1
source	= C
dest	= A
spare	= B



At point Y, recursive call 9 is made, and the new invocation of the method begins execution:

count	= 3
source	= A
dest	= B
spare	= C

Z →

count	= 2
source	= C
dest	= B
spare	= A

Y →

count	= 1
source	= C
dest	= B
spare	= A

This is the base case, so a disk is moved, the return is made, and the method continues execution.

count	= 3
source	= A
dest	= B
spare	= C

Z →

count	= 2
source	= C
dest	= B
spare	= A

count	= 1
source	= C
dest	= B
spare	= A

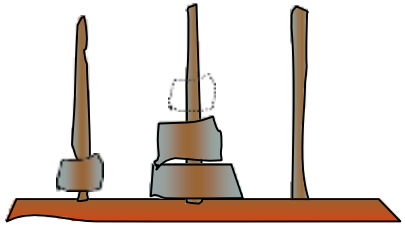
Figure 2.21e

Box trace of ***solveTowers(3, 'A', 'B', 'C')***

At point Z, recursive call 10 is made, and the new invocation of the method begins execution:

count = 3	Z →	count = 2
source = A		source = C
dest = B		dest = B
spare = C		spare = A

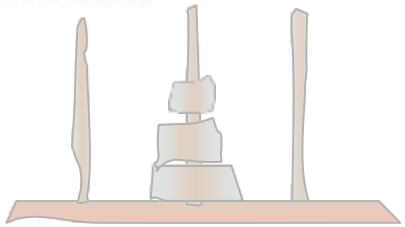
count = 2	Z →	count = 1
source = C		source = A
dest = B		dest = B
spare = A		spare = C



This is the base case, so a disk is moved, the return is made, and the method continues execution.

count = 3	Z →	count = 2
source = A		source = C
dest = B		dest = B
spare = C		spare = A

count = 1
source = A
dest = B
spare = C



This invocation completes, the return is made, and the method continues execution.

count = 3
source = A
dest = B
spare = C

spare = 2
source = C
dest = B
spare = A

count = 1
source = A
dest = B
spare = C

Cost of Hanoi Towers

- How many moves is necessary to solve Hanoi Towers problem for N disks?
- moves(1) = 1
- moves(N) = moves(N-1) + moves(1) + moves(N-1)
- i.e.
  moves(N) = 2*moves(N-1) + 1

- Guess solution and show it's correct with Mathematical Induction!

Ackerman's function

- In computability theory, theAckermann function, named after **WilhelmAckermann**, is one of the simplest and earliest-discovered examples of a total computable function that is not primitive recursive.All primitive recursive functions are total and computable, but theAckermann function illustrates that not all total computable functions are primitive recursive.

- It's a function with two arguments each of which can be assigned any non-negative integer.

- **Ackermann function is defined as:**

$$A(m,n) = \begin{cases} n+1 & \text{if } m=0 \\ A(m-1,1) & \text{if } m>0 \text{ and } n=0 \\ A(m-1,A(m,n-1)) & \text{if } m>0 \text{ and } n>0 \end{cases}$$

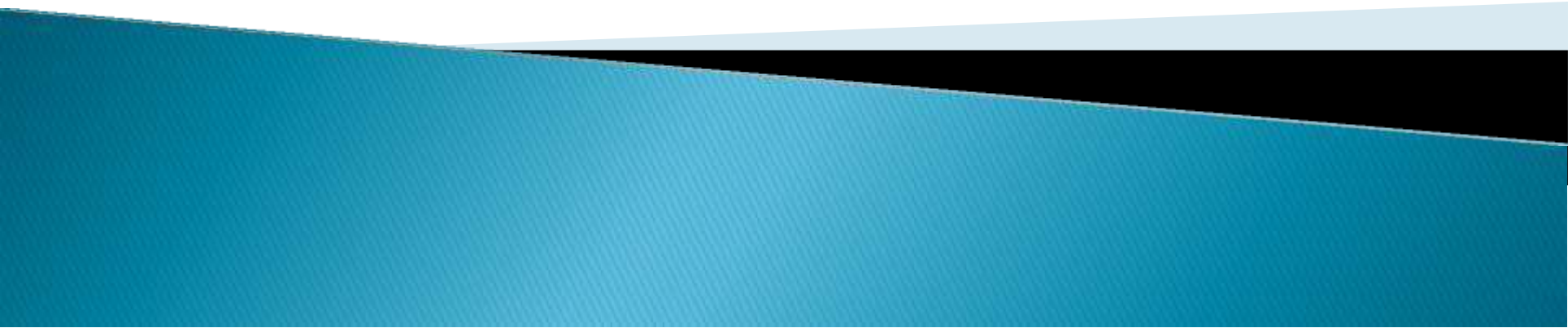
where m and n are non-negative integers

Ackermann algorithm:

```
Ackermann(m, n)
   {next and goal are arrays indexed from 0 to m, initialized so that next[0
    through next[m] are 0, goal[0] through goal[m - 1] are 1, and goal[m] is
repeat
    value <-- next[0] + 1
    transferring <-- true
    current <-- 0
    while transferring do begin
       if next[current] = goal[current] then goal[current] <-- value
                                        else transferring <-- false
       next[current] <-- next[current]+1
       current <-- current + 1
       end while
until next[m] = n + 1
return value {the value of A(m, n)}
end Ackermann
```

Thank you

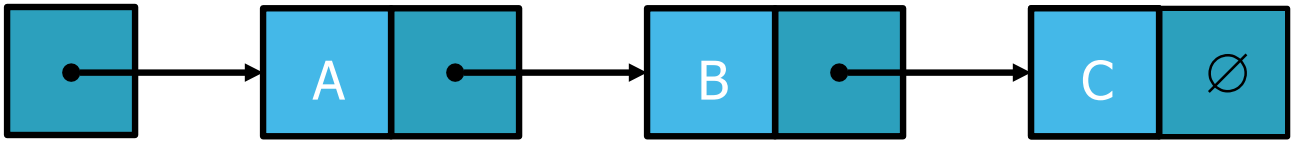
MODULE 3
Linked Lists



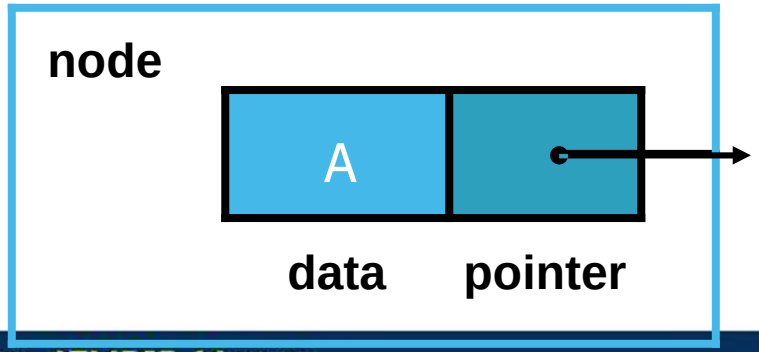
List Overview

- Linked lists
  - Abstract data type (ADT)
- Basic operations of linked lists
  - Insert, find, delete, print, etc.
- Variations of linked lists
  - Circular linked lists
  - Doubly linked lists

Linked Lists



- A *linked list* is a series of connected *nodes*
- Each node contains at least
  - A piece of data (any type)
  - Pointer to the next node in the list
- *Head*: pointer to the first node
- The last node points to `NULL`



A Simple Linked List Class

- We use two classes: `Node` and `List`
- Declare `Node` class for the nodes
  - `data`: `double`-type data in this example
  - `next`: a pointer to the next node in the list

```
class Node {
public:
      double      data;       // data
      Node*       next;       // pointer to next
};
```

A Simple Linked List Class

- Declare `List`, which contains
  - `head`: a pointer to the first node in the list.
    Since the list is empty initially, `head` is set to `NULL`
  - Operations on `List`

```
class List {
public:
        List(void) { head = NULL; }          // constructor
        ~List(void);                         // destructor

        bool IsEmpty() { return head == NULL; }
        Node* InsertNode(int index, double x);
        int FindNode(double x);
        int DeleteNode(double x);
        void DisplayList(void);
private:
        Node* head;
};
```

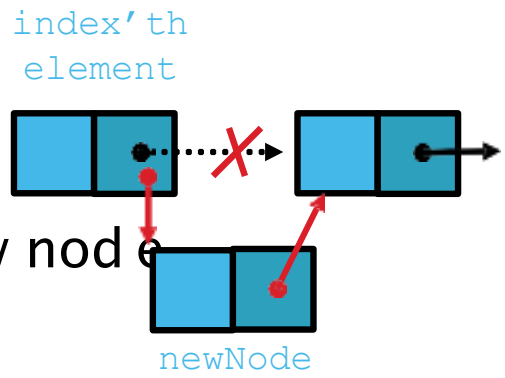
A Simple Linked List Class

- Operations of `List`
  - `IsEmpty`: determine whether or not the list is empty
  - `InsertNode`: insert a new node at a particular position
  - `FindNode`: find a node with a given value
  - `DeleteNode`: delete a node with a given value
  - `DisplayList`: print all the nodes in the list

Inserting a new node

- `Node* InsertNode(int index, double x)`
  - Insert a node with data equal to `x` after the `index`'th elements. (i.e., when `index = 0`, insert the node as the first element; when `index = 1`, insert the node after the first element, and so on)
  - If the insertion is successful, return the inserted node. Otherwise, return `NULL`.

    (If `index` is < 0 or > length of the list, the insertion will fail.)
- Steps
  1. Locate `index`'th element
  2. Allocate memory for the new node
  3. Point the new node to its successor
  4. Point the new node's predecessor to the new node



Inserting a new node

- Possible cases of `InsertNode`
  1. Insert into an empty list
  2. Insert in front
  3. Insert at back
  4. Insert in middle
- But, in fact, only need to handle two cases
  - Insert as the first node (Case 1 and Case 2)
  - Insert in the middle or at the end of the list (Case 3 and Case 4)

Inserting a new node

```
Node* List::InsertNode(int index, double x) {
        if (index < 0) return NULL;

        int currIndex  =       1;
        Node* currNode =       head;
        while (currNode && index > currIndex) {
                currNode       =       currNode->next;
                currIndex++;
        }
        if (index > 0 && currNode == NULL) return NULL;

        Node* newNode  =       new     Node;
        newNode->data  =       x;
        if (index == 0) {
                newNode->next  =       head;
                head           =       newNode;
        }
        else {
                newNode->next  =       currNode->next;
                currNode->next =       newNode;
        }
        return newNode;
}
```

Try to locate `index`'th node. If it doesn't exist, return `NULL`.

Inserting a new node

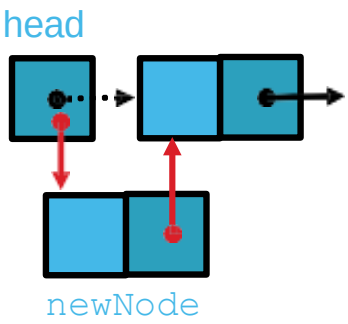
```
Node* List::InsertNode(int index, double x) {
        if (index < 0) return NULL;

        int currIndex  =       1;
        Node* currNode =       head;
        while (currNode && index > currIndex) {
                currNode       =       currNode->next;
                currIndex++;
        }
        if (index > 0 && currNode == NULL) return NULL;

        Node* newNode  =       new     Node;
        newNode->data  =       x;
        if (index == 0) {
                newNode->next  =       head;
                head           =       newNode;
        }
        else {
                newNode->next  =       currNode->next;
                currNode->next =       newNode;
        }
        return newNode;
}
```

Create a new node

Inserting a new node

```
Node* List::InsertNode(int index, double x) {
        if (index < 0) return NULL;

        int currIndex  =       1;
        Node* currNode =       head;
        while (currNode && index > currIndex) {
                currNode       =       currNode->next;
                currIndex++;
        }
        if (index > 0 && currNode == NULL) return NULL;

        Node* newNode  =       new     Node;
        newNode->data  =       x;
        if (index == 0) {
                newNode->next  =       head;
                head           =       newNode;
        }
        else {
                newNode->next  =       currNode->next;
                currNode->next =       newNode;
        }
        return newNode;
}
```

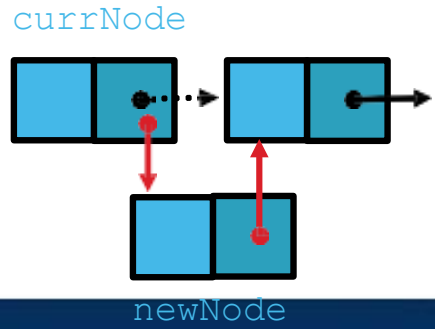
Insert as first element



Inserting a new node

```
Node* List::InsertNode(int index, double x) {
        if (index < 0) return NULL;

        int currIndex  =       1;
        Node* currNode =       head;
        while (currNode && index > currIndex) {
                currNode       =       currNode->next;
                currIndex++;
        }
        if (index > 0 && currNode == NULL) return NULL;

        Node* newNode  =       new     Node;
        newNode->data  =       x;
        if (index == 0) {
                newNode->next  =       head;
                head           =       newNode;
        }
        else {
                newNode->next  =       currNode->next;
                currNode->next =       newNode;
        }
        return newNode;
}
```

Insert after currNode



Finding a node

- `int FindNode(double x)`
  - Search for a node with the value equal to `x` in the list.
  - If such a node is found, return its position. Otherwise, return 0.

```
int List::FindNode(double x) {

        Node* currNode      =       head;
        int currIndex       =       1;
        while (currNode && currNode->data != x) {
                currNode    =       currNode->next;
                currIndex++;
        }
        if (currNode) return currIndex;
        return 0;
}
```

Deleting a node

- `int DeleteNode(double x)`
  - Delete a node with the value equal to `x` from the list.
  - If such a node is found, return its position. Otherwise, return 0.
- Steps
  - Find the desirable node (similar to `FindNode`)
  - Release the memory occupied by the found node
  - Set the pointer of the predecessor of the found node to the successor of the found node
- Like `InsertNode`, there are two special cases
  - Delete first node
  - Delete the node in middle or at the end of the list

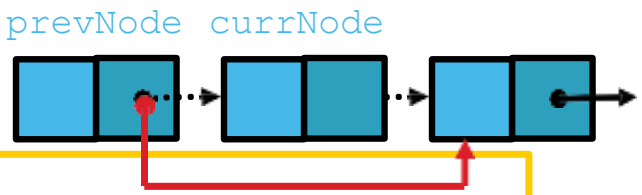
Deleting a node

```
in  List::DeleteNode(double x) {
        Node* prevNode =        NULL;
        Node* currNode =        head;
        int currIndex  =        1;
        while (currNode && currNode->data != x) {
                prevNode        =       currNode;
                currNode        =       currNode->next;
                currIndex++;
        }
        if (currNode) {
                if (prevNode) {
                        prevNode->next =        currNode->next;
                        delete currNode;
                }
                else {
                        head            =       currNode->next;
                        delete currNode;
                }
                return currIndex;
        }
        return 0;
}
```

Try to find the node with its value equal to x

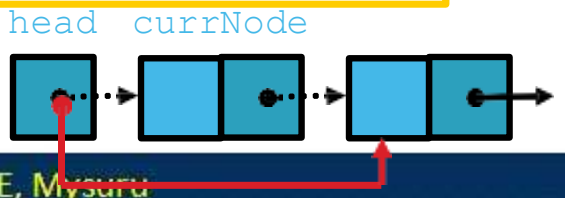
Deleting a node

```
int List::DeleteNode(double x) {
        Node* prevNode =        NULL;
        Node* currNode =        head;
        int currIndex  =        1;
        while (currNode && currNode->data != x) {
                prevNode        =       currNode;
                currNode        =       currNode->next;
                currIndex++;
        }
        if (currNode) {
                if (prevNode) {
                        prevNode->next =        currNode->next;
                        delete currNode;
                }
                else {
                        head            =       currNode->next;
                        delete currNode;
                }
                return currIndex;
        }
        return 0;
}
```



Deleting a node

```
int List::DeleteNode(double x) {
        Node* prevNode =        NULL;
        Node* currNode =        head;
        int currIndex  =        1;
        while (currNode && currNode->data != x) {
                prevNode        =       currNode;
                currNode        =       currNode->next;
                currIndex++;
        }
        if (currNode) {
                if (prevNode) {
                        prevNode->next =        currNode->next;
                        delete currNode;
                }
                else {
                        head            =       currNode->next;
                        delete currNode;
                }
                return currIndex;
        }
        return 0;
}
```



Printing all the elements

- `void DisplayList(void)`
  - Print the data of all the elements
  - Print the number of the nodes in the list

```
void List::DisplayList()
{
    int num          =      0;
    Node* currNode   =      head;
    while (currNode != NULL){
        cout << currNode->data << endl;
        currNode     =      currNode->next;
        num++;
    }
    cout << "Number of nodes in the list: " << num << endl;
}
```

Destroying the list

- ~List(void)
  - Use the destructor to release all the memory used by the list.
  - Step through the list and delete each node one by one.

```
List::~List(void) {
    Node* currNode = head, *nextNode = NULL;
    while (currNode != NULL)
    {
        nextNode    =    currNode->next;
        // destroy the current node
        delete currNode;
        currNode    =    nextNode;
    }
}
```

Using List

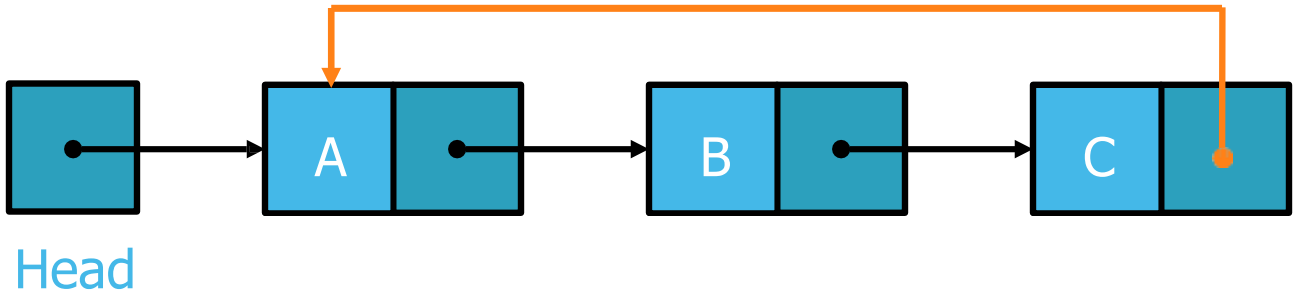
```
int main(void)
{
        List list;
        list.InsertNode(0, 7.0);    // successful
        list.InsertNode(1, 5.0);    // successful
        list.InsertNode(-1, 5.0);   // unsuccessful
        list.InsertNode(0, 6.0);    // successful
        list.InsertNode(8, 4.0);    // unsuccessful
        // print all the elements
        list.DisplayList();
        if(list.FindNode(5.0)  > 0) cout << "5.0 found" << endl;
        else                        cout << "5.0 not found" << endl;
        if(list.FindNode(4.5)  > 0) cout << "4.5 found" << endl;
        else                        cout << "4.5 not found" << endl;
        list.DeleteNode(7.0);
        list.DisplayList();
        return 0;
}
```

result

```
6
7
5
Number of nodes in the list: 3
5.0 found
4.5 not found
6
5
Number of nodes in the list: 2
```

Variations of Linked Lists

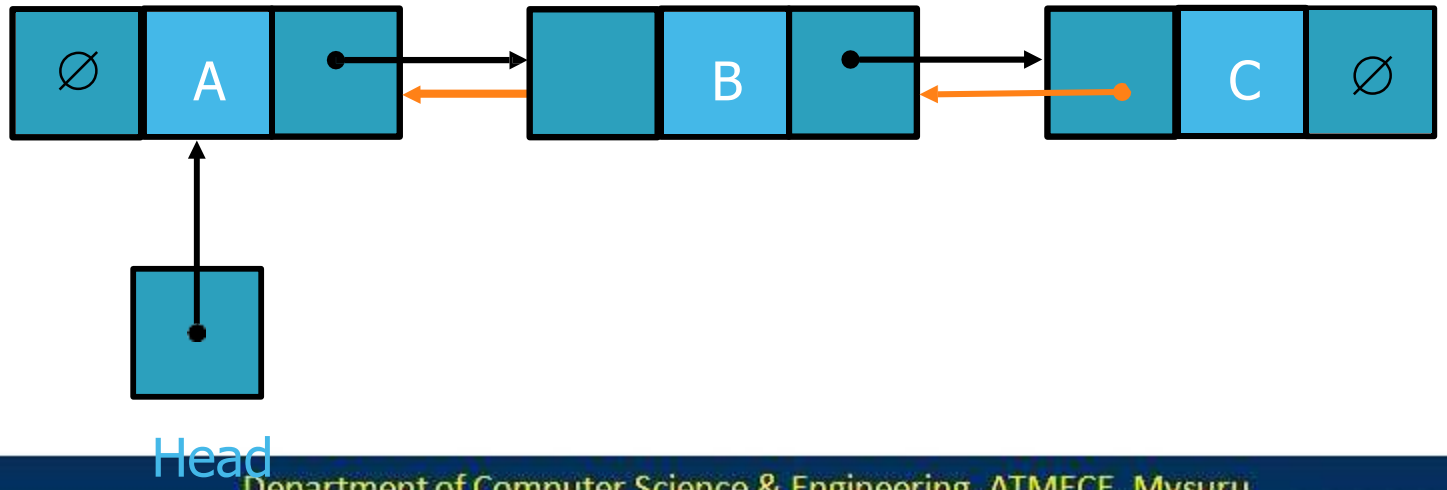
- *Circular linked lists*
  - The last node points to the first node of the list



  - How do we know when we have finished traversing the list? (Tip: check if the pointer of the current node is equal to the head.)

Variations of Linked Lists

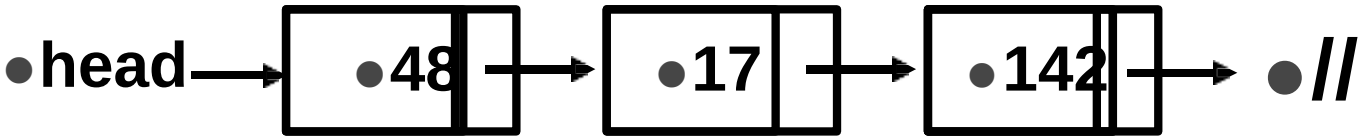
- *Doubly linked lists*
  - Each node points to not only successor but the predecessor
  - There are two `NULL`: at the first and last nodes in the list
  - Advantage: given a node, it is easy to visit its predecessor. Convenient to traverse lists backwards



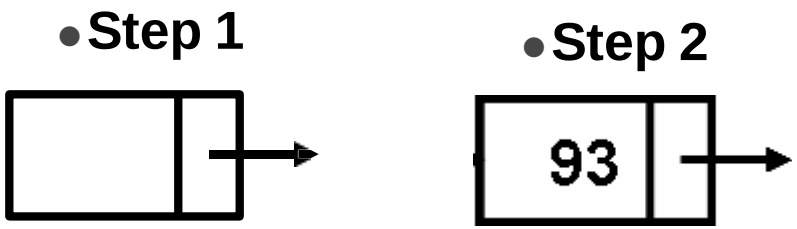
Array versus Linked Lists

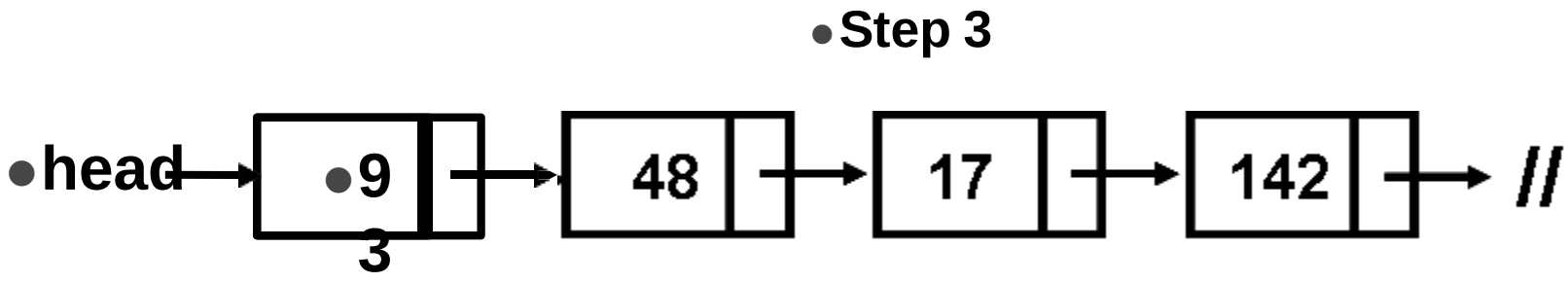
- Linked lists are more complex to code and manage than arrays, but they have some distinct advantages.
  - **Dynamic**: a linked list can easily grow and shrink in size.
    - We don't need to know how many nodes will be in the list. They are created in memory as needed.
    - In contrast, the size of a C++ array is fixed at compilation time.
  - **Easy and fast insertions and deletions**
    - To insert or delete an element in an array, we need to copy to temporary variables to make room for new elements or close the gap caused by deleted elements.
    - With a linked list, no need to move other nodes. Only need to reset some pointers.

Insertion Description



- Follow the previous steps and we get





Insertion Description

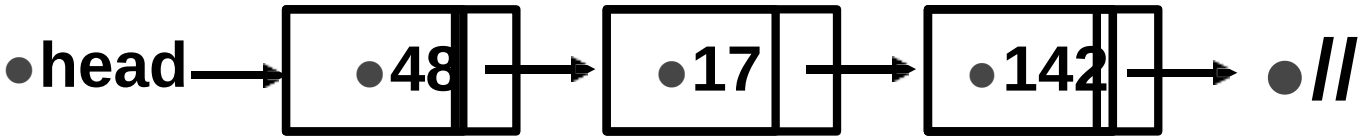
- Insertion at the top of the list
- Insertion at the end of the list
- Insertion in the middle of the list

Insertion at the end

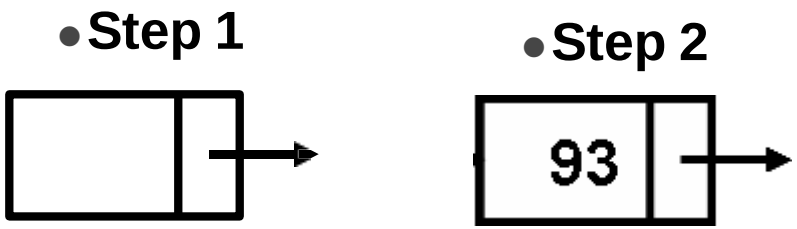
Steps:

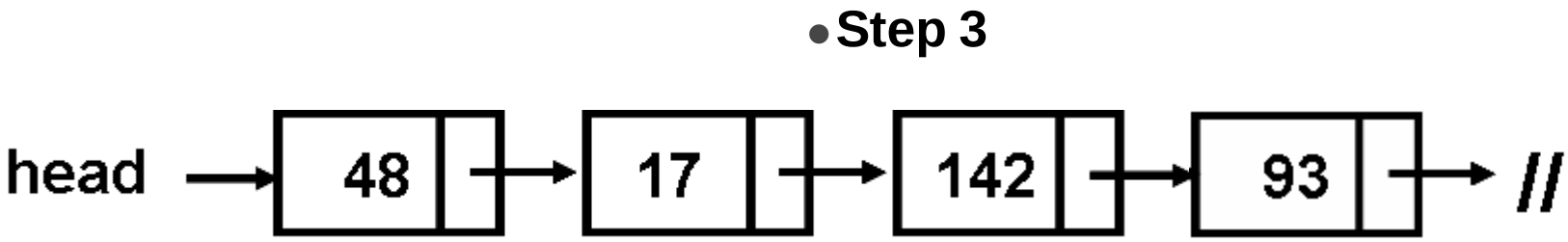
- Create a Node
- Set the node data Values
- Connect the pointers

Insertion Description



- Follow the previous steps and we get





Insertion Description

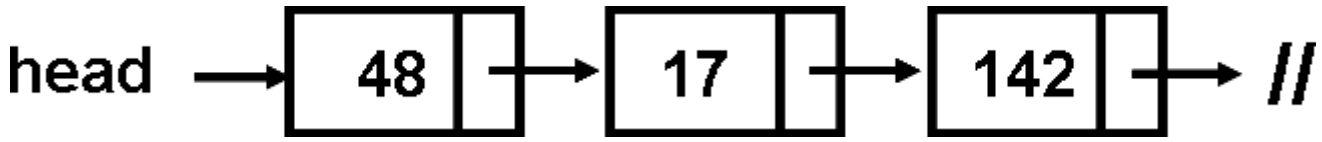
- Insertion at the top of the list
- Insertion at the end of the list
- Insertion in the middle of the list

Insertion in the middle

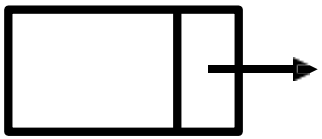
Steps:

- Create a Node
- Set the node data Values
- Break pointer connection
- Re-connect the pointers

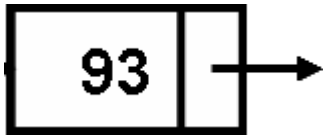
Insertion Description



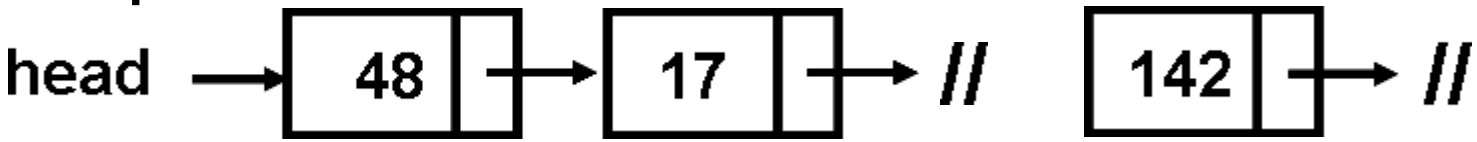
- Step 1



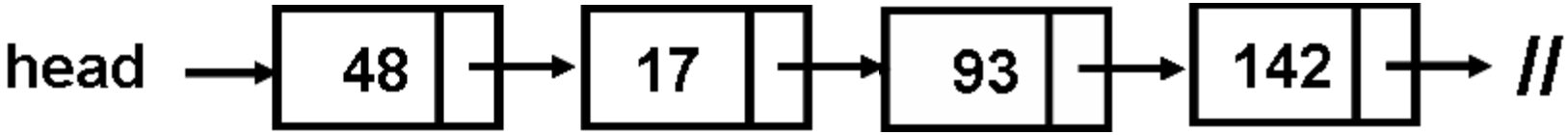
- Step 2



- Step 3



- Step 4



Outline

- Introduction
- Insertion Description
- Deletion Description
- Basic Node Implementation
- Conclusion

Deletion Description

- Deleting from the top of the list
- Deleting from the end of the list
- Deleting from the middle of the list

Deletion Description

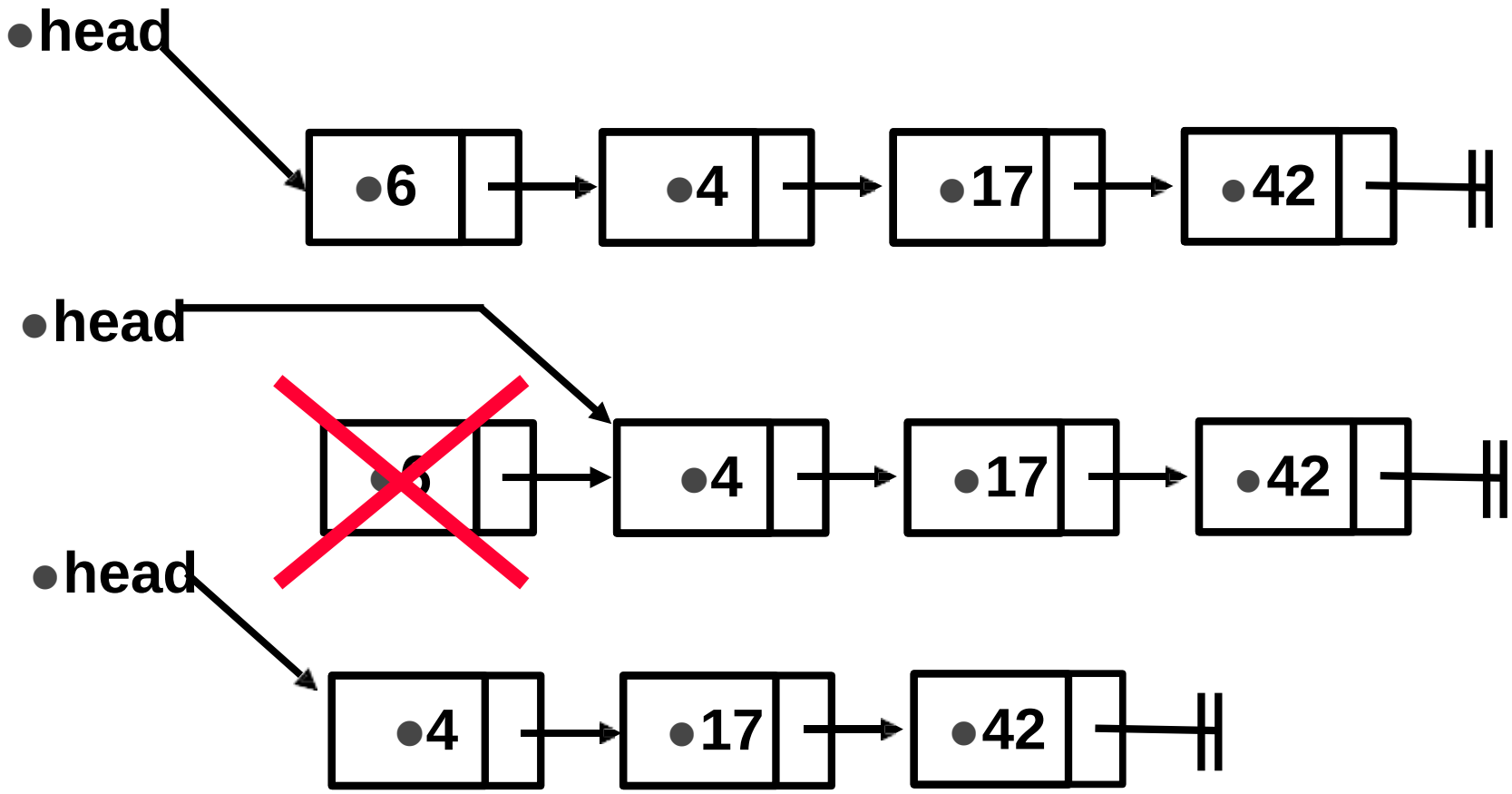
- Deleting from the top of the list
- Deleting from the end of the list
- Deleting from the middle of the list

Deleting from the top

Steps

- Break the pointer connection
- Re-connect the nodes
- Delete the node

Deletion Description



Deletion Description

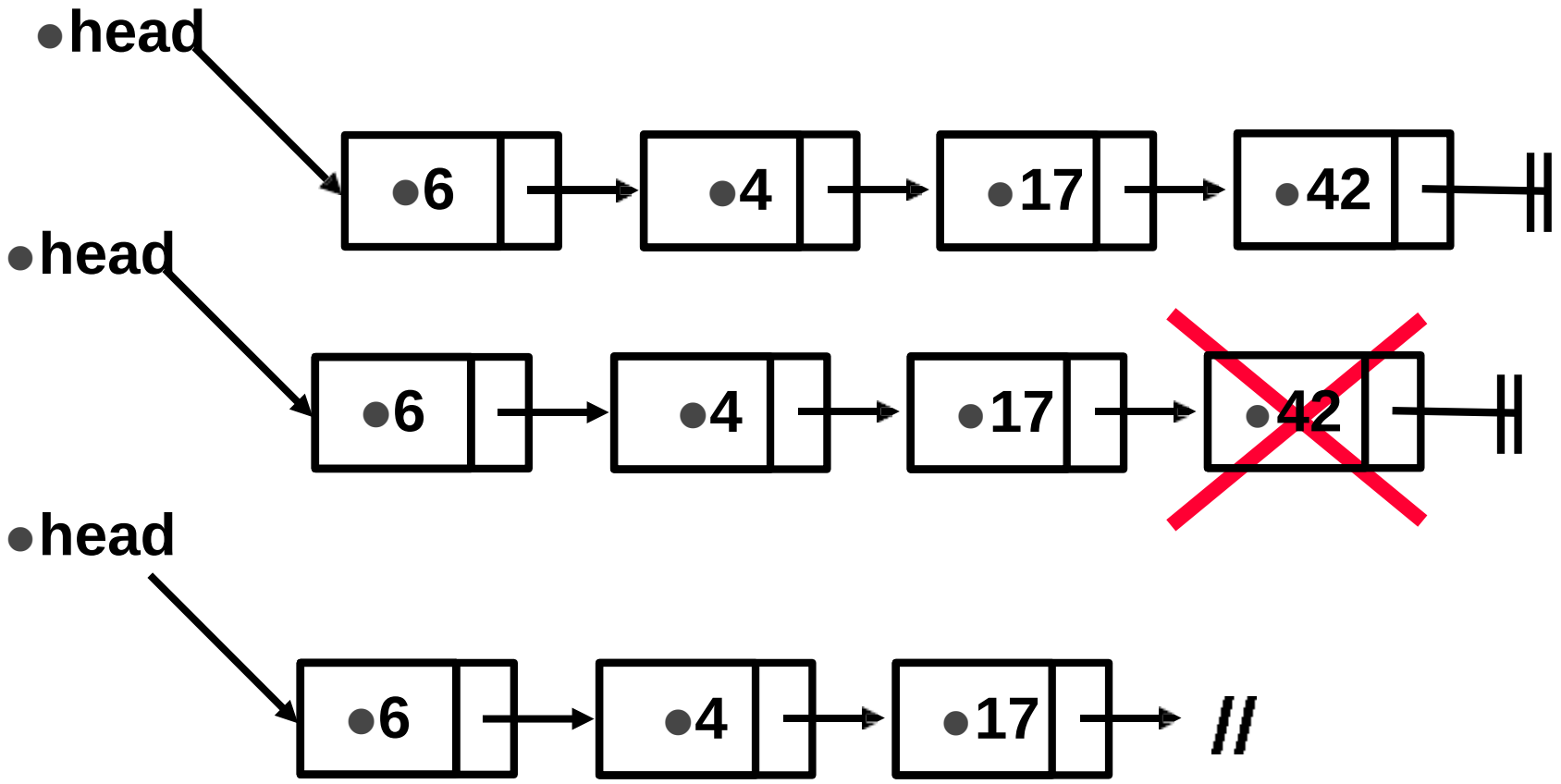
- Deleting from the top of the list
- Deleting from the end of the list
- Deleting from the middle of the list

Deleting from the end

Steps

- Break the pointer connection
- Set previous node pointer to NULL
- Delete the node

Deletion Description



Deletion Description

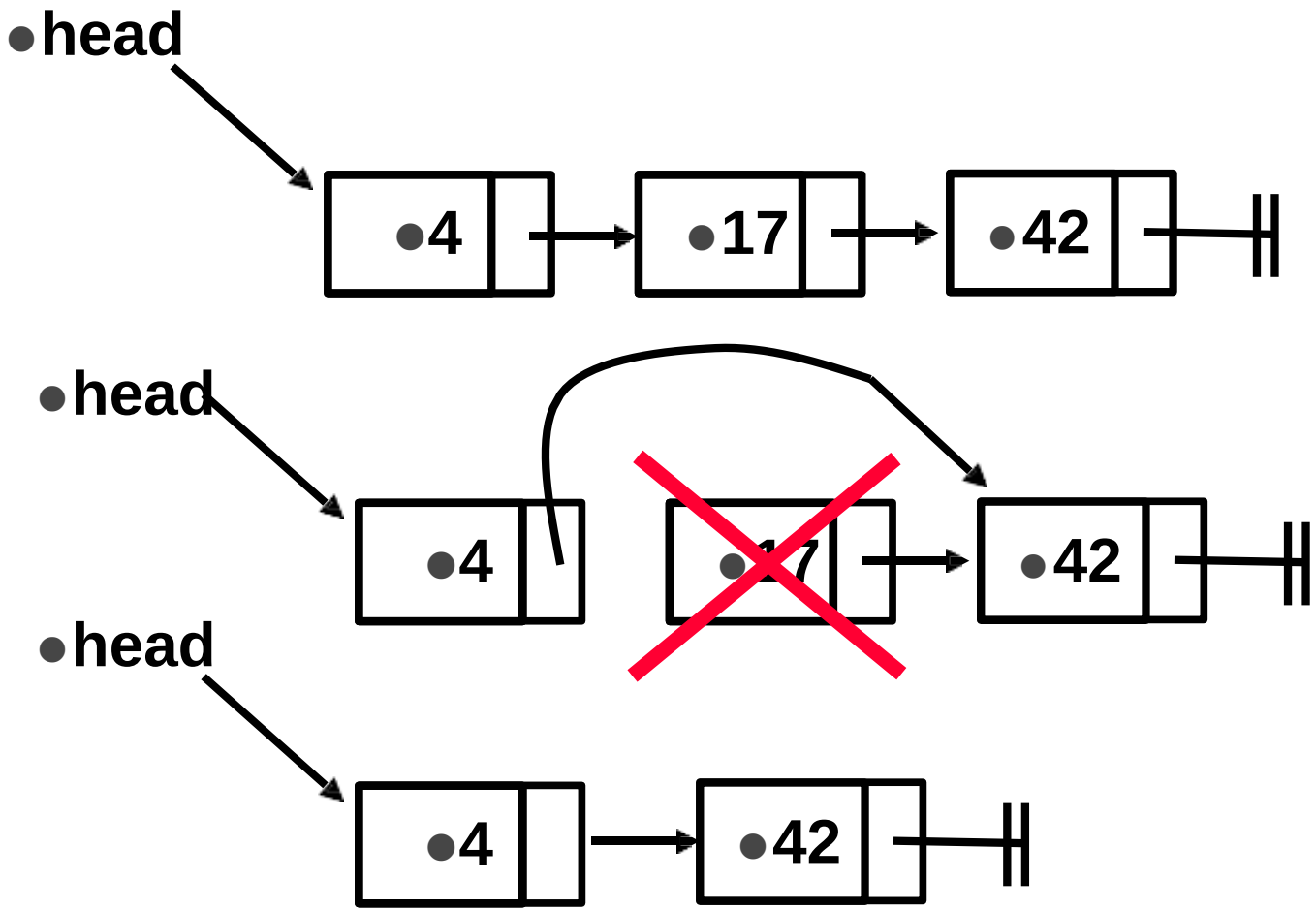
- Deleting from the top of the list
- Deleting from the end of the list
- Deleting from the middle of the list

Deleting from the Middle

Steps

- Set previous Node pointer to next node
- Break Node pointer connection
- Delete the node

Deletion Description



Basic Node Implementation

The following code is written in C++:

```
Struct Node
{
   int data;                    //any type of data could be another
   struct
   Node *next;                  //this is an important piece of code
   "pointer"
};
```

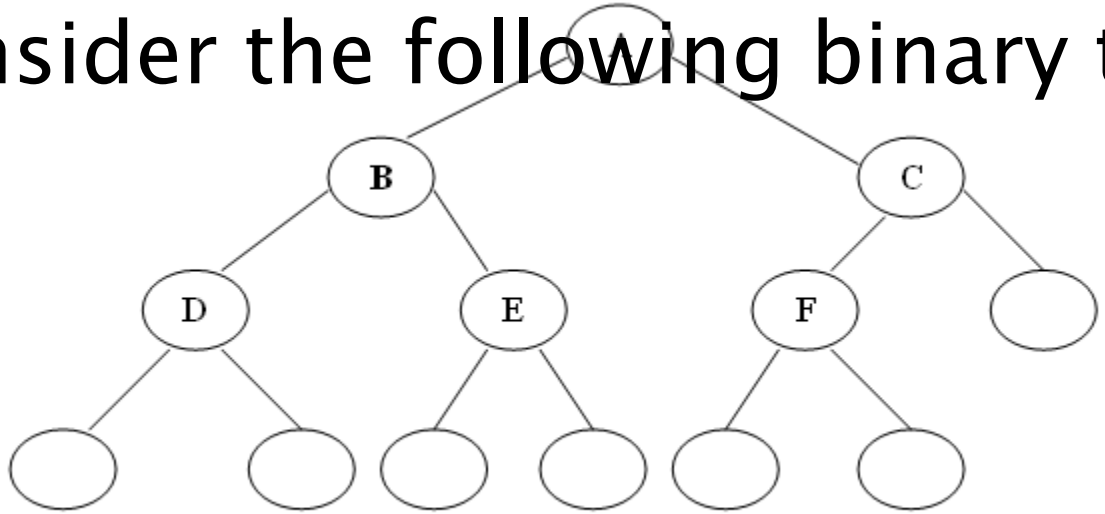
MODULE – 4

TREES

Threaded Binary Tree

Threaded Binary Tree

- In a linked representation of a binary tree, the number of null links (null pointers) are actually more than non-null pointers.
- Consider the following binary tree:



A Binary tree with the null pointers

Threaded Binary Tree

- In above binary tree, there are 7 null pointers & actual 5 pointers.
- In all there are 12 pointers.
- We can generalize it that for any binary tree with n nodes there will be (n+1) null pointers and 2n total pointers.
- The objective here to make effective use of these null pointers.
- A. J. perils & C. Thornton jointly proposed idea to make effective use of these null pointers.
- According to this idea we are going to replace all the null pointers by the appropriate pointer values called threads.

Threaded Binary Tree

- And binary tree with such pointers are called threaded tree.
- In the memory representation of a threaded binary tree, it is necessary to distinguish between a normal pointer and a thread.

Threaded Binary Tree

- Therefore we have an alternate node representation for a threaded binary tree which contains five fields as show bellow:

For any node p, in a threaded binary tree.

lthred(p)=1 indicates lchild (p) is a thread pointer

lthred(p)=0 indicates lchild (p) is a normal

rthred(p)=1 indicates rchild (p) is a thread

rthred(p)=0 indicates rchild (p) is a normal pointer

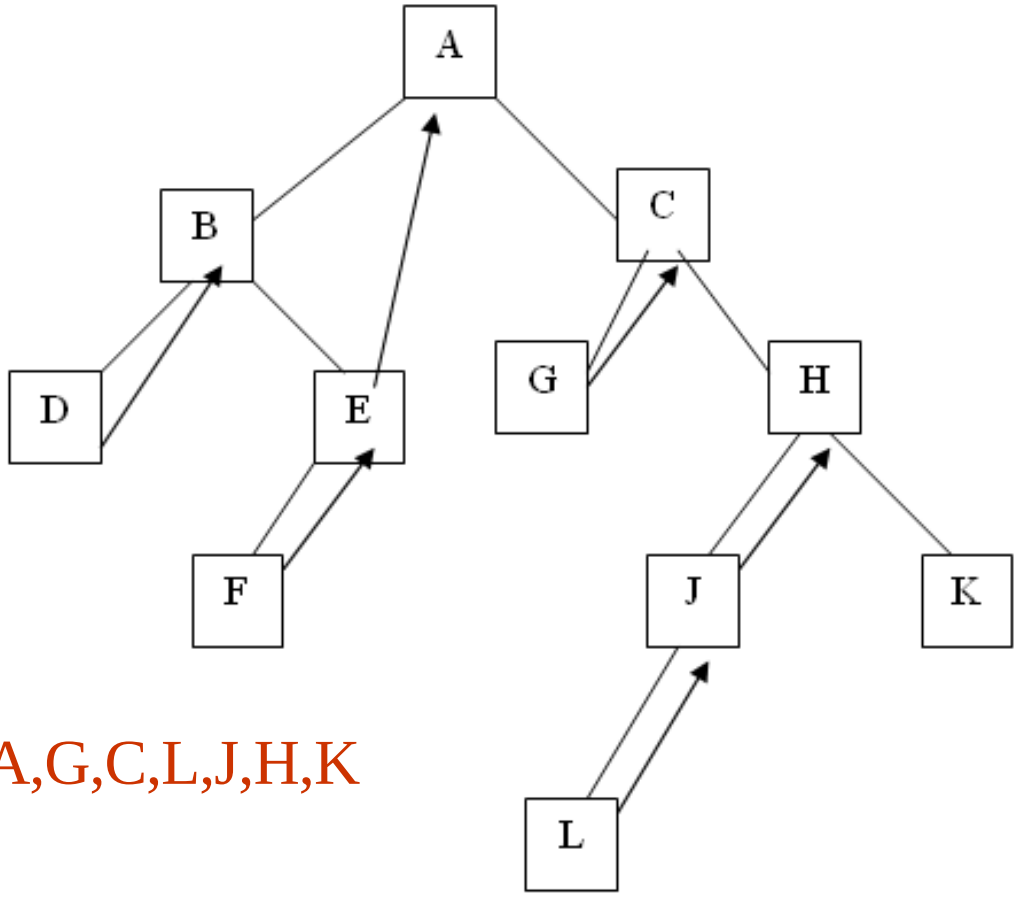
Threaded Binary Tree

- Also one may choose a one-way threading or a two-way threading.
- Here, our threading will correspond to the in order traversal of T.

Threaded Binary Tree One-Way

- Accordingly, in the one way threading of T, a thread will appear in the right field of a node and will point to the next node in the **in-order** traversal of T.
- See the bellow example of one-way **in-order** threading.

Threaded Binary Tree: One-Way



Inorder of bellow tree is: D,B,F,E,A,G,C,L,J,H,K

One-way inorder threading

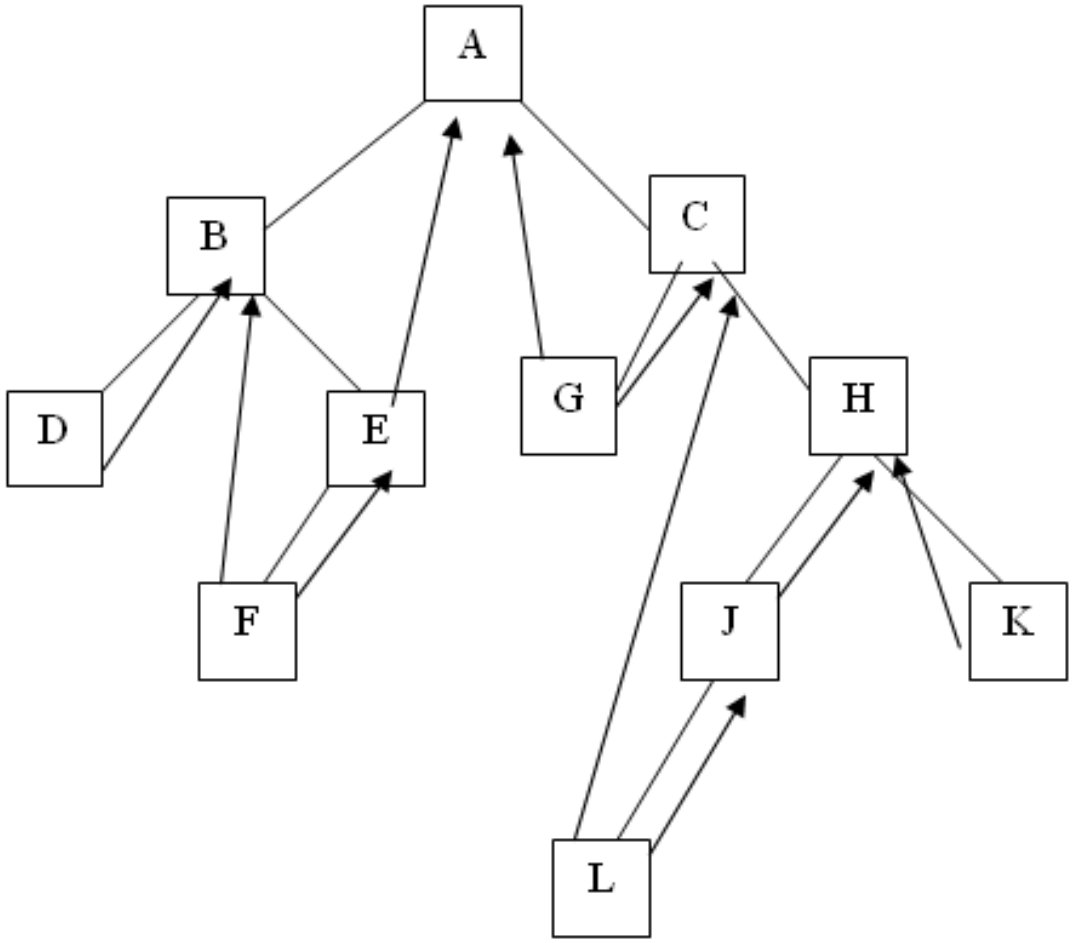
Threaded Binary Tree
Two-Way

- In the two-way threading of T.
- A thread will also appear in the left field of a node and will point to the preceding node in the **in-order** traversal of tree T.
- Furthermore, the left pointer of the first node and the right pointer of the last node (in the **in-order** traversal of T) will contain the null value when T does not have a header node.

Threaded Binary Tree

- Bellow figure show two-way **in-order** threading.
- Here, right pointer=next node of **in-order** traversal and left pointer=previous node of **in-order** traversal
- Inorder of bellow tree is: D,B,F,E,A,G,C,L,J,H,K

Threaded Binary Tree



Two-way inorder threading

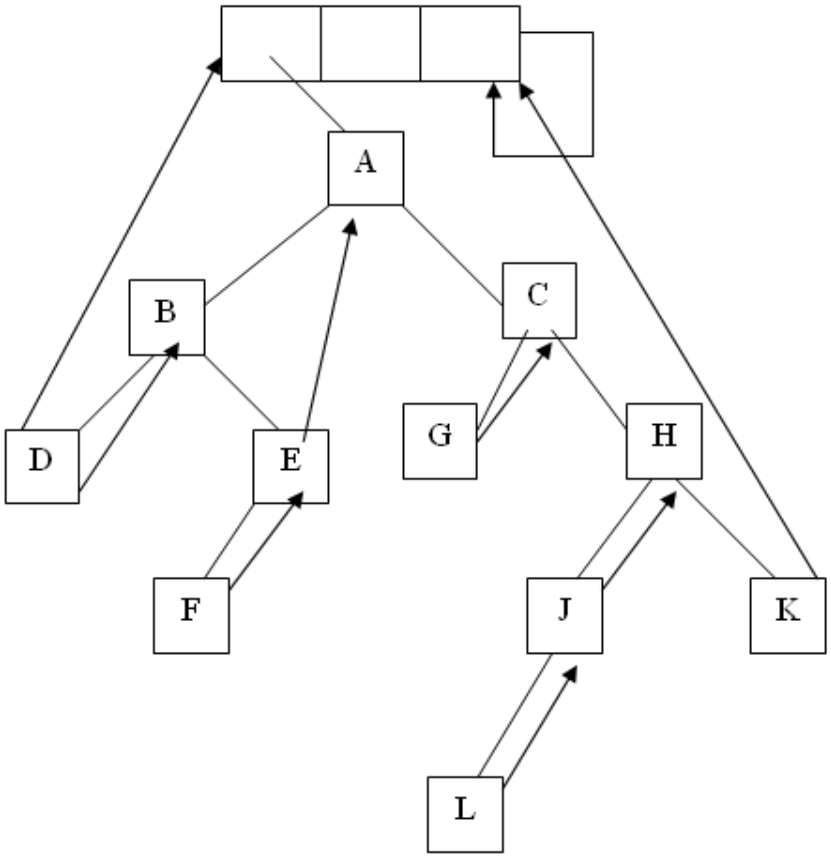
Threaded Binary Tree

Two-way Threading with Header node

- Again two-way threading has left pointer of the first node and right pointer of the last node (in the inorder traversal of T) will contain the null value when T will point to the header nodes is called two-way threading with header node threaded binary tree.

Threaded Binary Tree

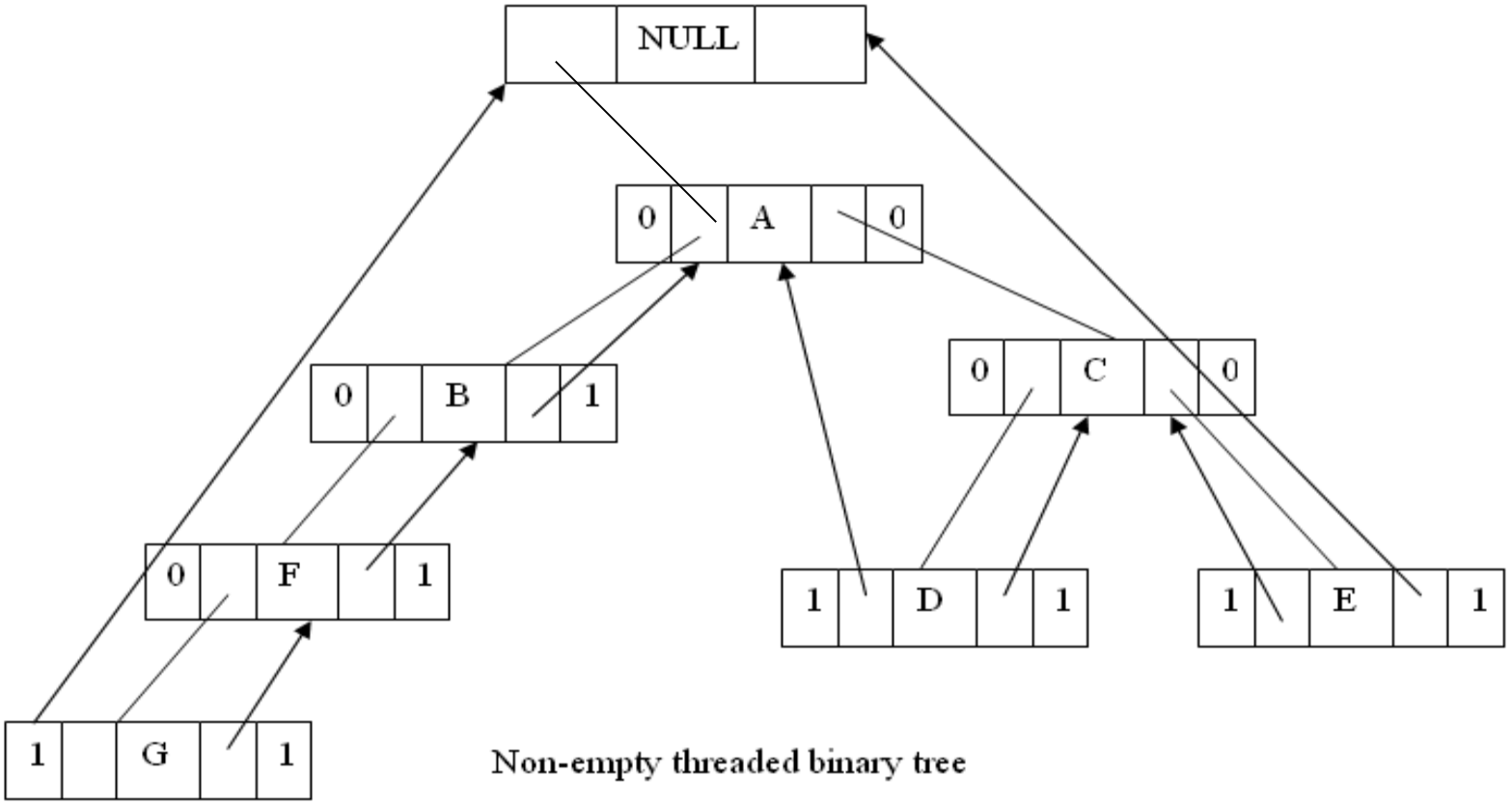
- Bellow figure to explain two-way threading with header node.



Threaded Binary Tree

- Bellow example of link representation of threading binary tree.
- **In-order** traversal of bellow tree: G,F,B,A,D,C,E

Threaded Binary Tree



Non-empty threaded binary tree

Threaded Binary Tree

- **Advantages of threaded binary tree:**
- Threaded binary trees have numerous advantages over non-threaded binary trees listed as below:
  - The traversal operation is more faster than that of its unthreaded version, because with threaded binary tree non-recursive implementation is possible which can run faster and does not require the botheration of stack management.

Threaded Binary Tree

- **Advantages of threaded binary tree:**
  - The advantage is that associated with a threaded binary tree, we can efficiently determine the predecessor and successor nodes starting from any node. In case of unthreaded binary tree, however, this task is more time consuming and difficult. For this case a stack is required to provide upward pointing information in the tree whereas in a threaded binary tree, without having to include the overhead of using a stack mechanism the same can be carried out with the threads.

Threaded Binary Tree

Advantages of threaded binary tree:

- Any node can access its predecessor and successor node.
- Threads are usually more to upward whereas links are downward. Thus in a threaded tree, one can move in their direction and nodes are in fact circularly linked. This is not possible in unthreaded counter part because there we can move only in downward direction starting from root.
- Insertion into and deletions from a threaded tree are although time consuming operations but these are very easy to implement.

Threaded Binary Tree

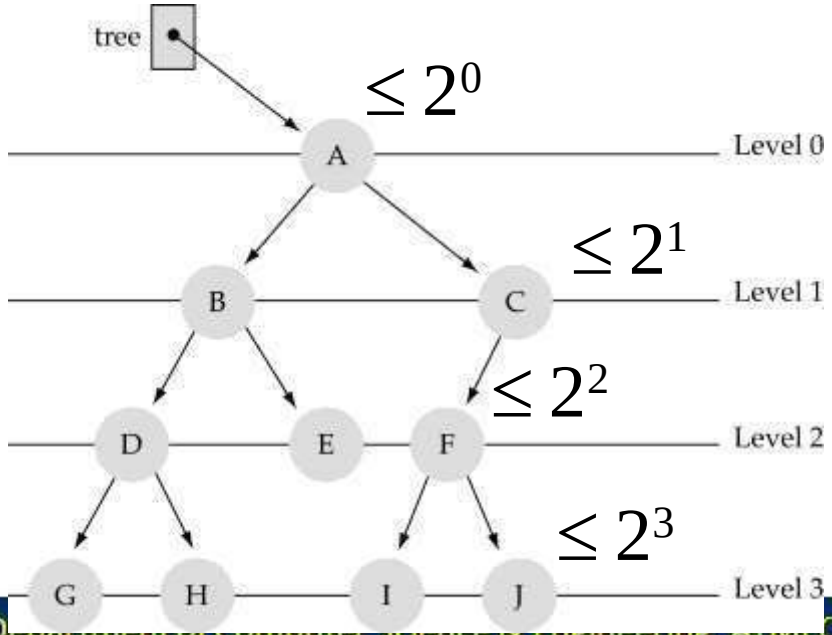
- Insertion and deletion from a threaded tree are very time consuming operation compare to non-threaded binary tree.
- This tree require additional bit to identify the threaded link.

- **Disadvantages of threaded binary tree:**

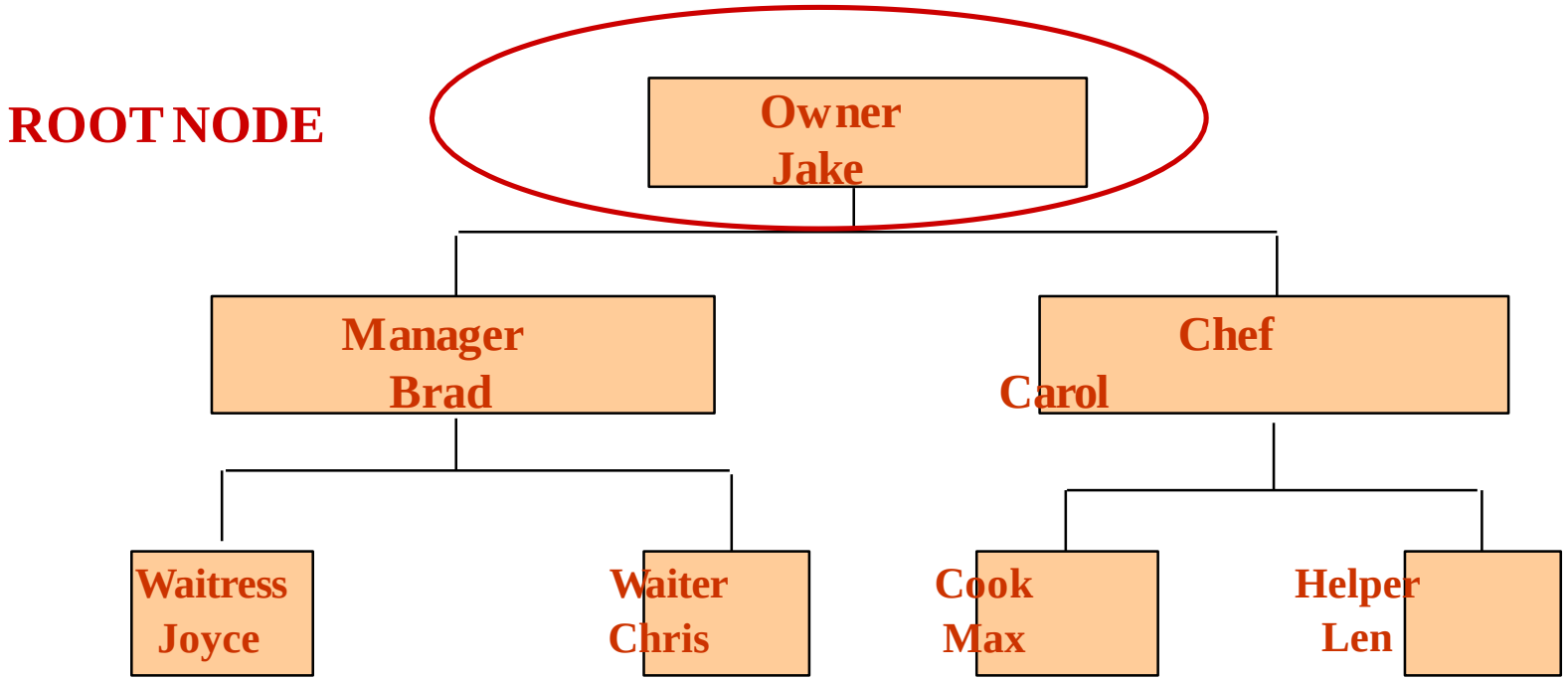
TREES

What is a binary tree?

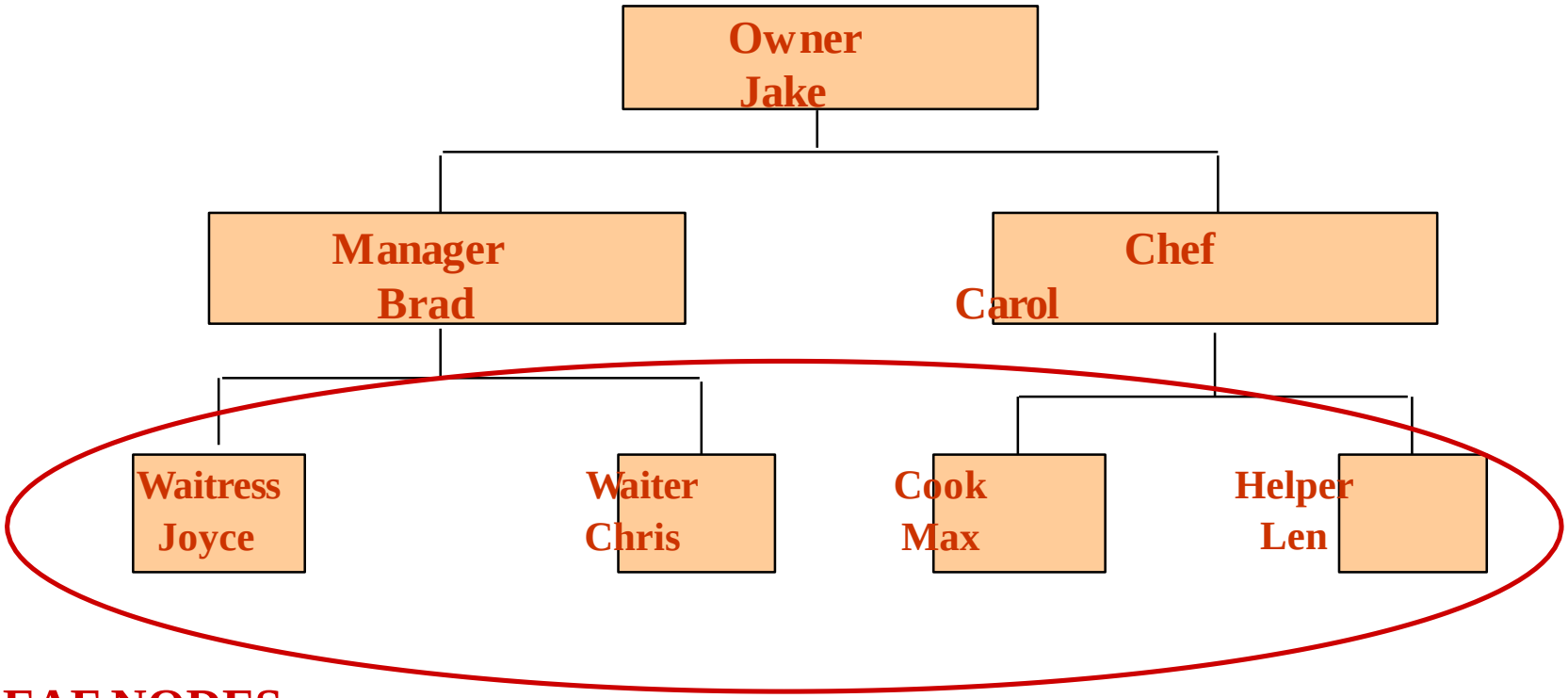
- *Property1*: each node can have up to two successor nodes (*children*)
  - The predecessor node of a node is called its *parent*
  - The "beginning" node is called the *root* (no parent)
  - A node without *children* is called a *leaf*



A Tree Has a Root Node



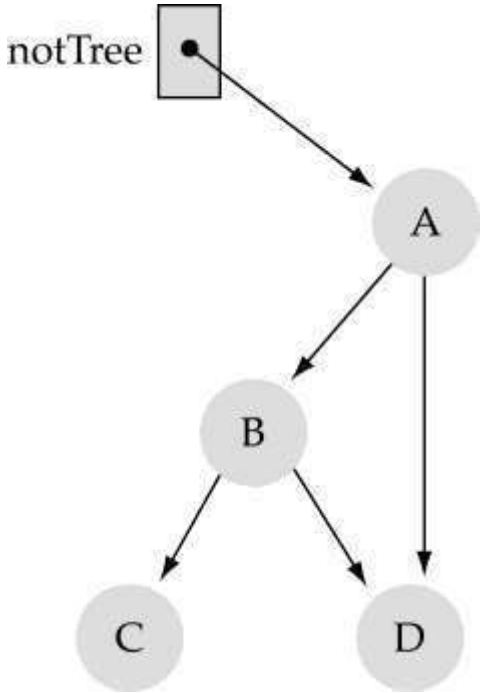
Leaf nodes have no children



LEAF NODES

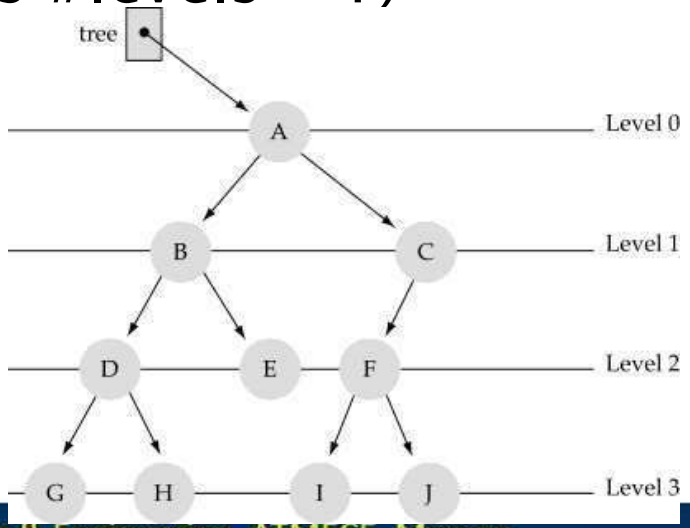
What is a binary tree? (cont.)

- *Property2*: a unique path exists from the root to every other node

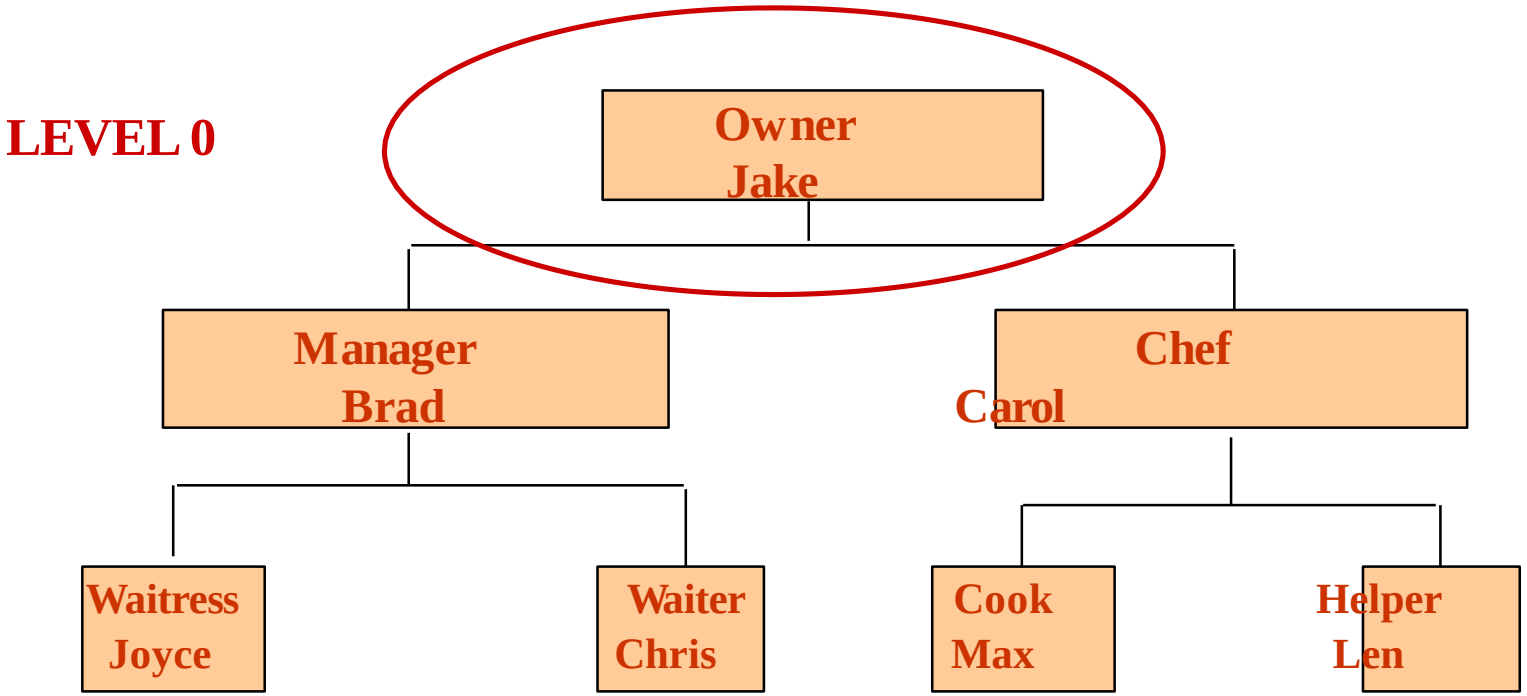


Some terminology

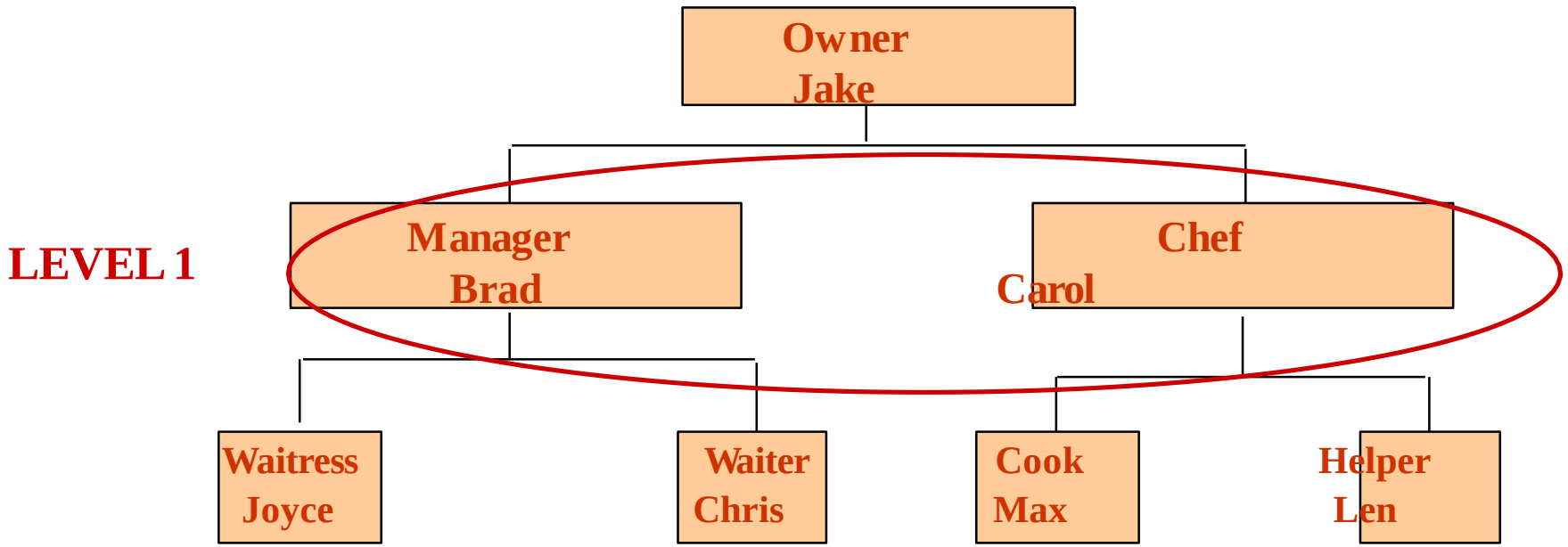
- Ancestor of a node: any node on the path from the root to that node
- Descendant of a node: any node on a path from the node to the last node in the path
- Level (depth) of a node: number of edges in the path from the root to that node
- Height of a tree: number of levels (warning: some books define it as #levels – 1)



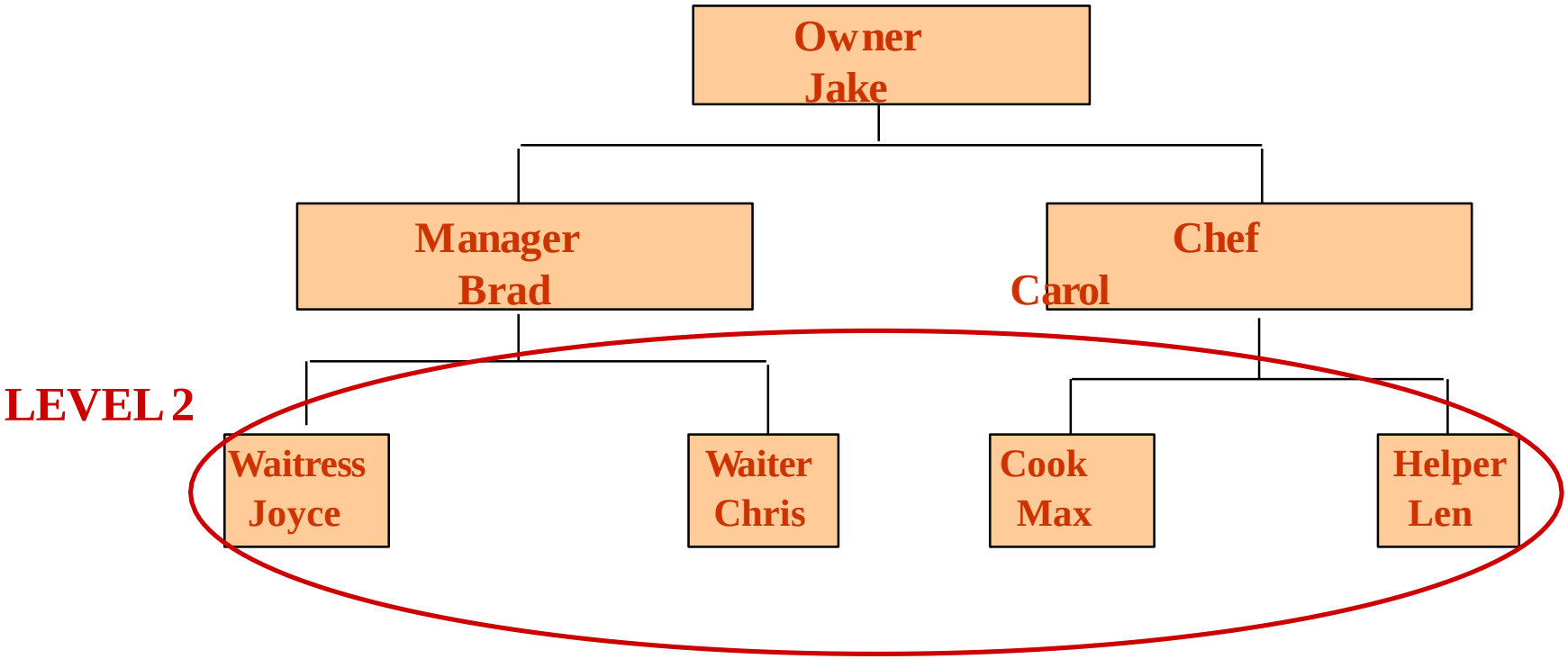
A Tree Has Levels



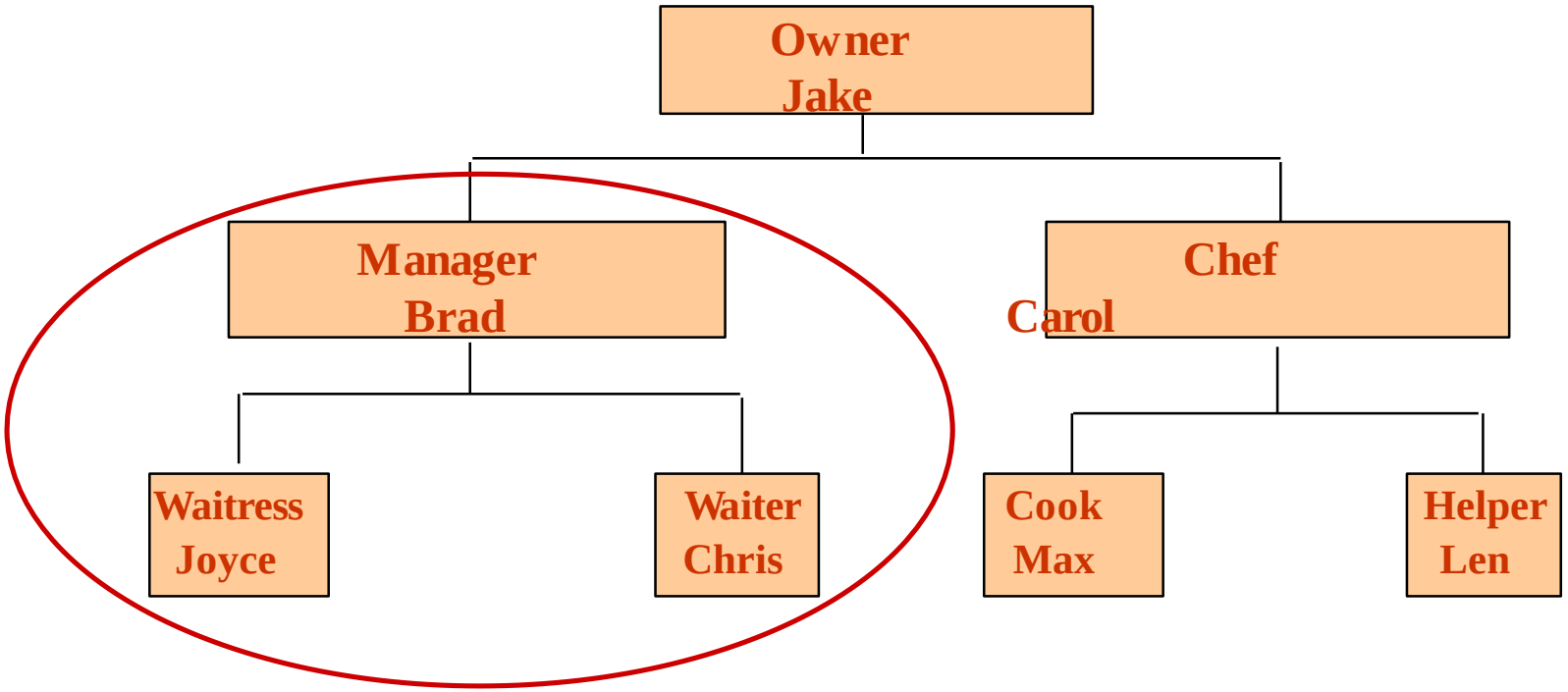
Level One


LEVEL 1
Owner
Jake
Manager
Brad
Chef
Carol
Waitress
Joyce
Waiter
Chris
Cook
Max
Helper
Len

Level Two

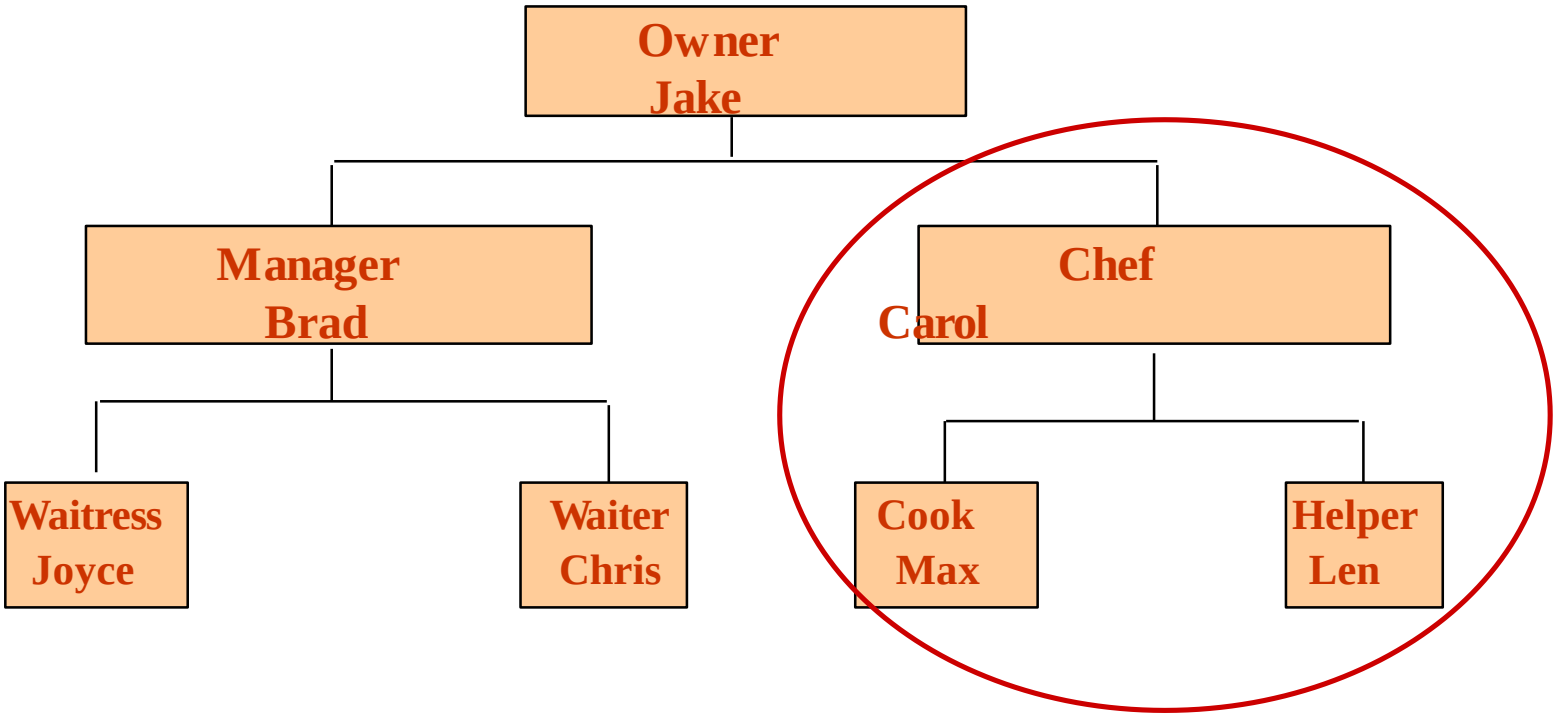

Owner
Jake
Manager
Brad
Chef
Carol
LEVEL 2
Waitress
Joyce
Waiter
Chris
Cook
Max
Helper
Len

A Subtree



LEFT SUBTREE OF ROOT NODE

Another Subtree



RIGHT SUBTREE OF ROOT NODE

What is the # of nodes N of a full tree with height h?

l 2^l

$$N = \underset{l=0}{2^0} + \underset{l=1}{2^1} + \ldots + \underset{l=h-1}{2^{h-1}} = 2^h - 1$$

using the geometric series:

$$x^0 + x^1 + \ldots + x^{n-1} = \sum_{i=0}^{n-1} x^i = \frac{x^n - 1}{x - 1}$$

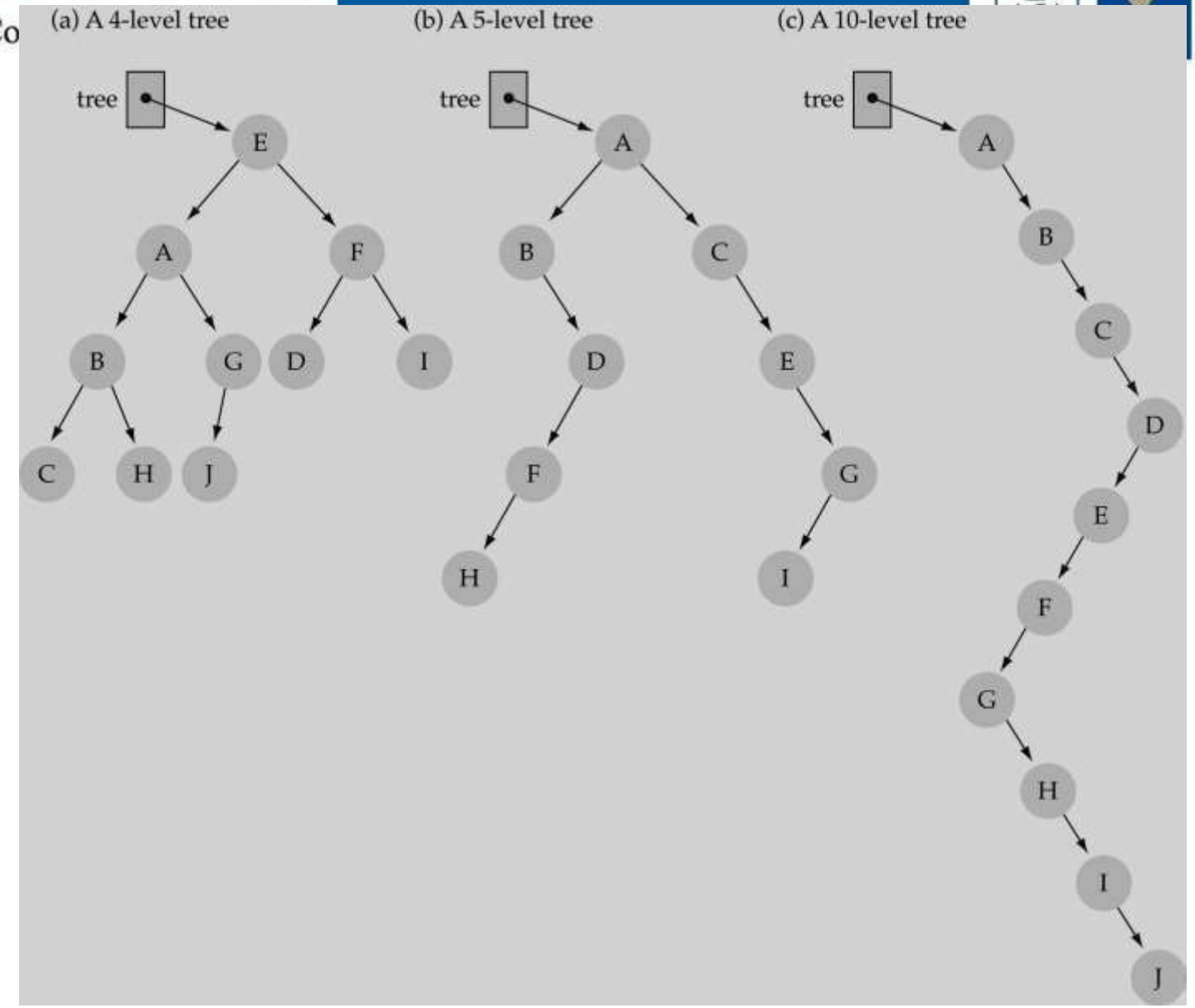
What is the height h of a full tree with N nodes?

$$2^h - 1 = N$$

$$\Rightarrow 2^h = N + 1$$

$$\Rightarrow h = \log(N+1) \rightarrow O(\log N)$$

- The n… is *N* (same…
- The min height of a tree with *N* nodes is *log(N+1)*


(a) A 4-level tree
tree
E
A
F
B
G
D
I
C
H
J
(b) A 5-level tree
tree
A
B
C
D
E
F
G
H
I
(c) A 10-level tree
tree
A
B
C
D
E
F
G
H
I
J

Searching a binary tree

(1) Start at the root

(2) Search the tree level by level, until you find the element you are searching for

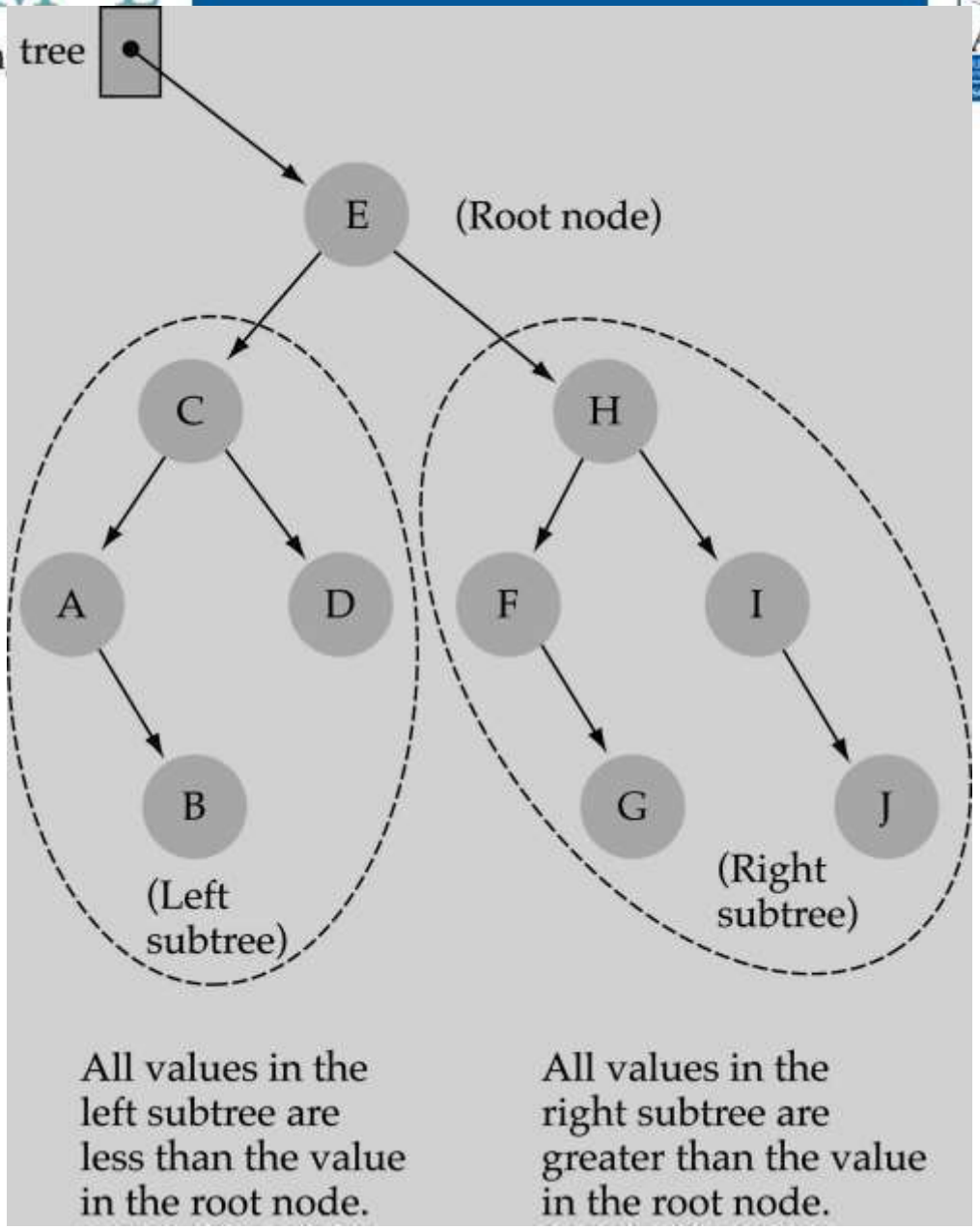
(*O(N)* time in worst case)

Is this better than searching a linked list?

No ---> O(N)

Binary Search Trees

- **Binary Search Tree Property**: The value stored at a node is *greater* than the value stored at its left child and *less* than the value stored at its right child
- Thus, the value stored at the root of a subtree is *greater* than any value in its left subtree and *less* than any value in its right subtree!!


tree
E
(Root node)
C
H
A
D
F
I
B
G
J
(Left subtree)
(Right subtree)
All values in the left subtree are less than the value in the root node.
All values in the right subtree are greater than the value in the root node.

Searching a binary search tree

(1) Start at the root

(2)Compare the value of the item you are searching for with the value stored at the root

(3)If the values are equal, then *item found*; otherwise, if it is a leaf node, then *not found*

Searching a binary search tree (cont.)

(4) If it is less than the value stored at the root, then search the left subtree

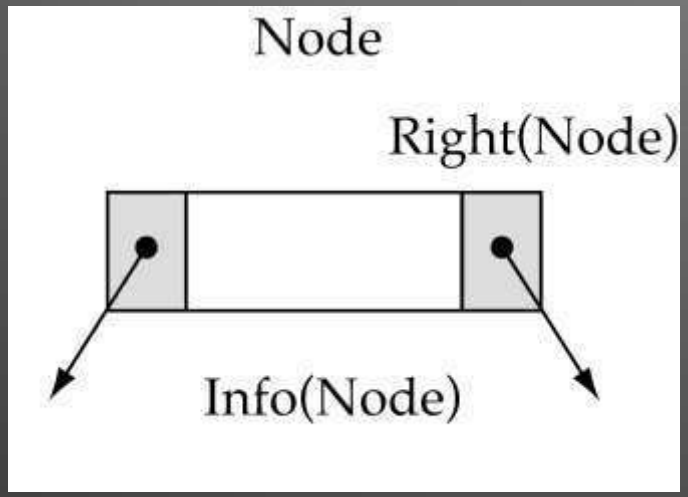
(5)If it is greater than the value stored at the root, then search the right subtree

(6)Repeat steps 2-6 for the root of the subtree chosen in the previous step 4 or 5

Is this better than searching a linked list?

Yes !! ---> O(logN)

Tree node structure




```
template<class ItemType>
struct TreeNode {
    ItemType info;
    TreeNode* left;
    TreeNode* right; };
```

Binary Search Tree Specification

```
#include <fstream.h>

template<class ItemType>
struct TreeNode;

enum OrderType {PRE_ORDER, IN_ORDER, POST_ORDER};

template<class ItemType>
class TreeType {
  public:
  TreeType();
    ~TreeType();
    TreeType(const TreeType<ItemType>&);
    void operator=(const TreeType<ItemType>&);
    void MakeEmpty();
    bool IsEmpty() const;
    bool IsFull() const;
    int NumberOfNodes() const;
```

(continues)

Binary Search Tree Specification

(cont.)

```
    void RetrieveItem(ItemType&, bool& found);
    void InsertItem(ItemType);
    void DeleteItem(ItemType);
    void ResetTree(OrderType);
    void GetNextItem(ItemType&, OrderType, bool&);
    void PrintTree(ofstream&) const;
  private:
    TreeNode<ItemType>* root;
};


};
```

Function NumberOfNodes

- Recursive implementation

  #nodes in a tree ≡
  #nodes in left subtree + #nodes in right subtree + 1

- What is the size factor?

  Number of nodes in the tree we are examining

- What is the base case?

  The tree is empty

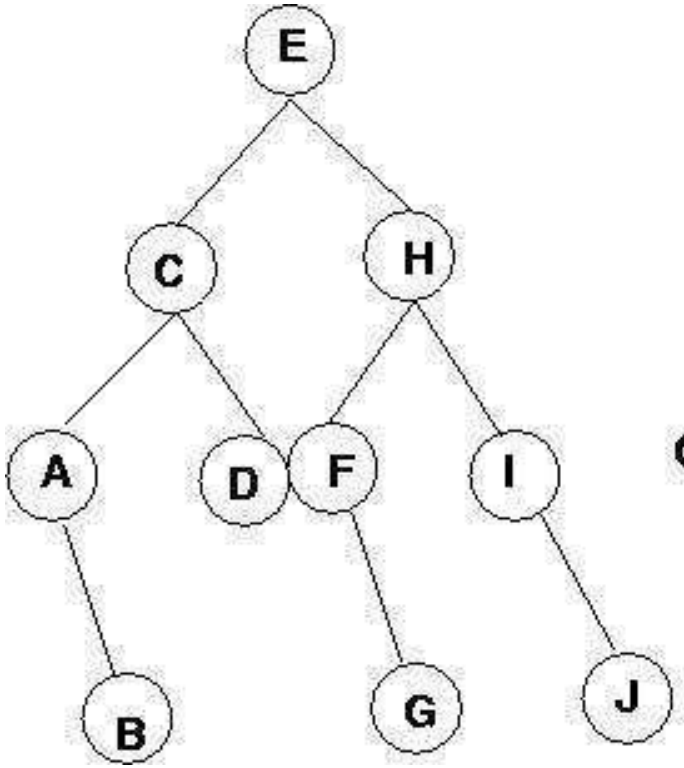
- What is the general case?

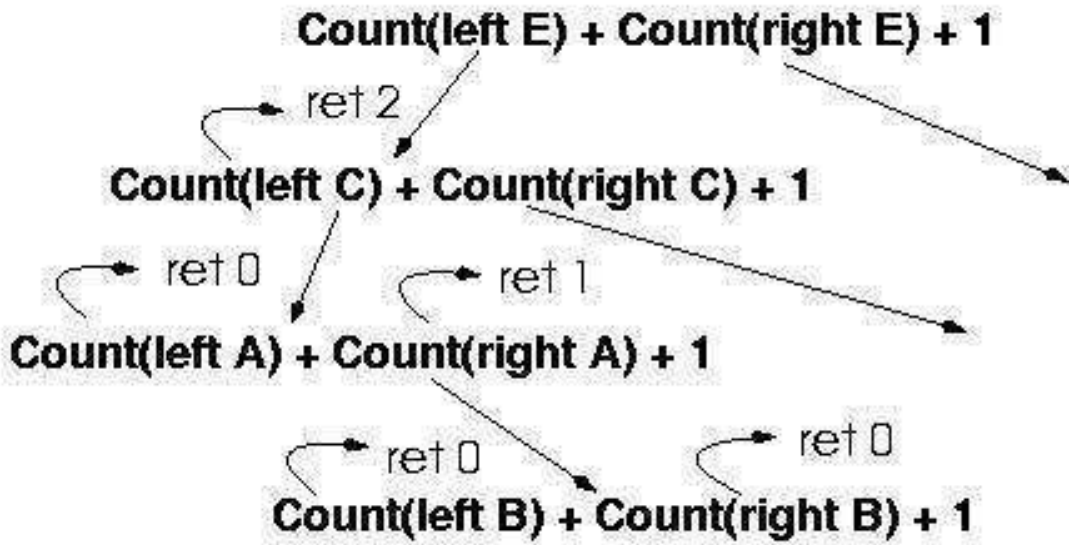
  CountNodes(Left(tree)) + CountNodes(Right(tree)) + 1

Function NumberOfNodes (cont.)

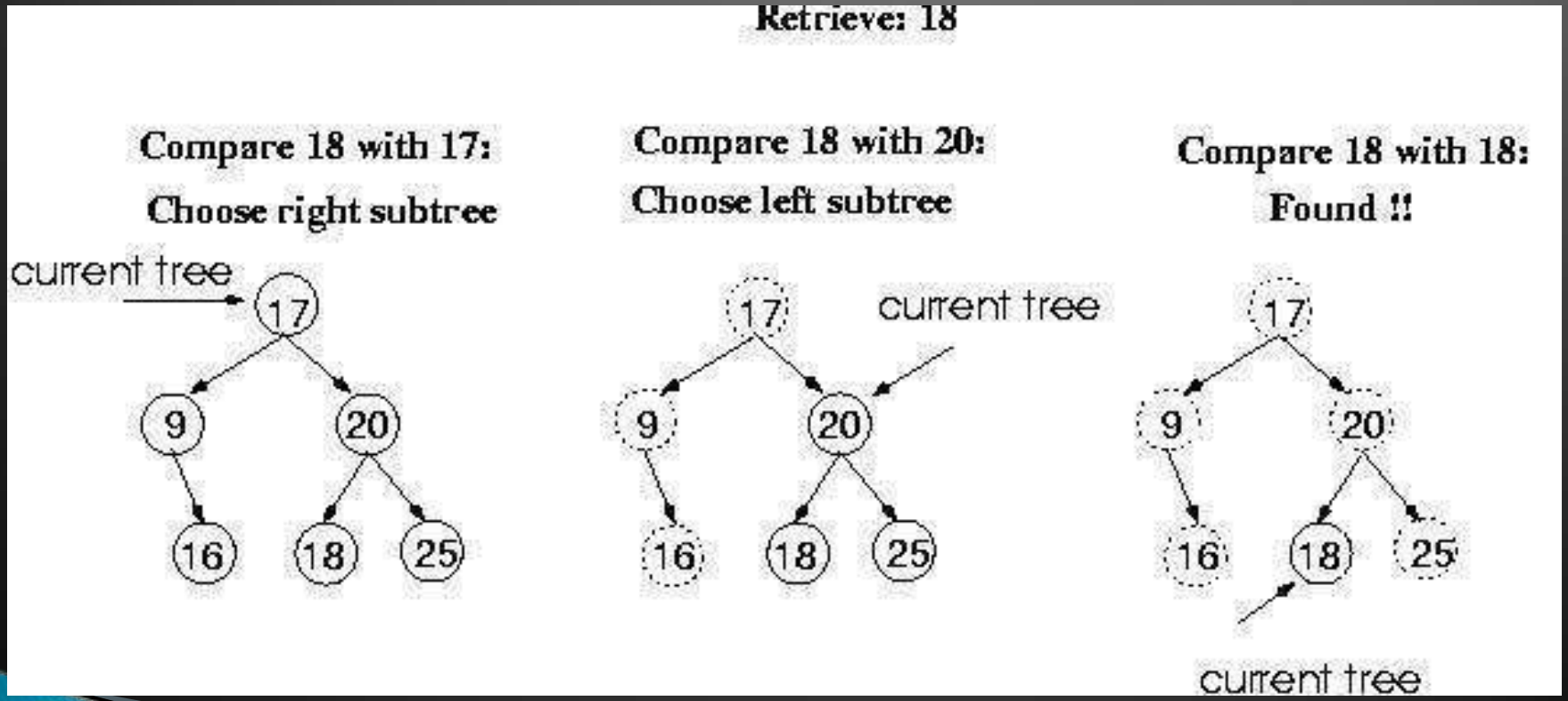
```
template<class ItemType>
int TreeType<ItemType>::NumberOfNodes() const
{
 return CountNodes(root);
}

template<class ItemType>
int CountNodes(TreeNode<ItemType>* tree)
{
 if (tree == NULL)
   return 0;
 else
   return CountNodes(tree->left) + CountNodes(tree->right) + 1;
}
```

Let's consider the first few steps:





Function Retrieveltem



Function RetrieveItem

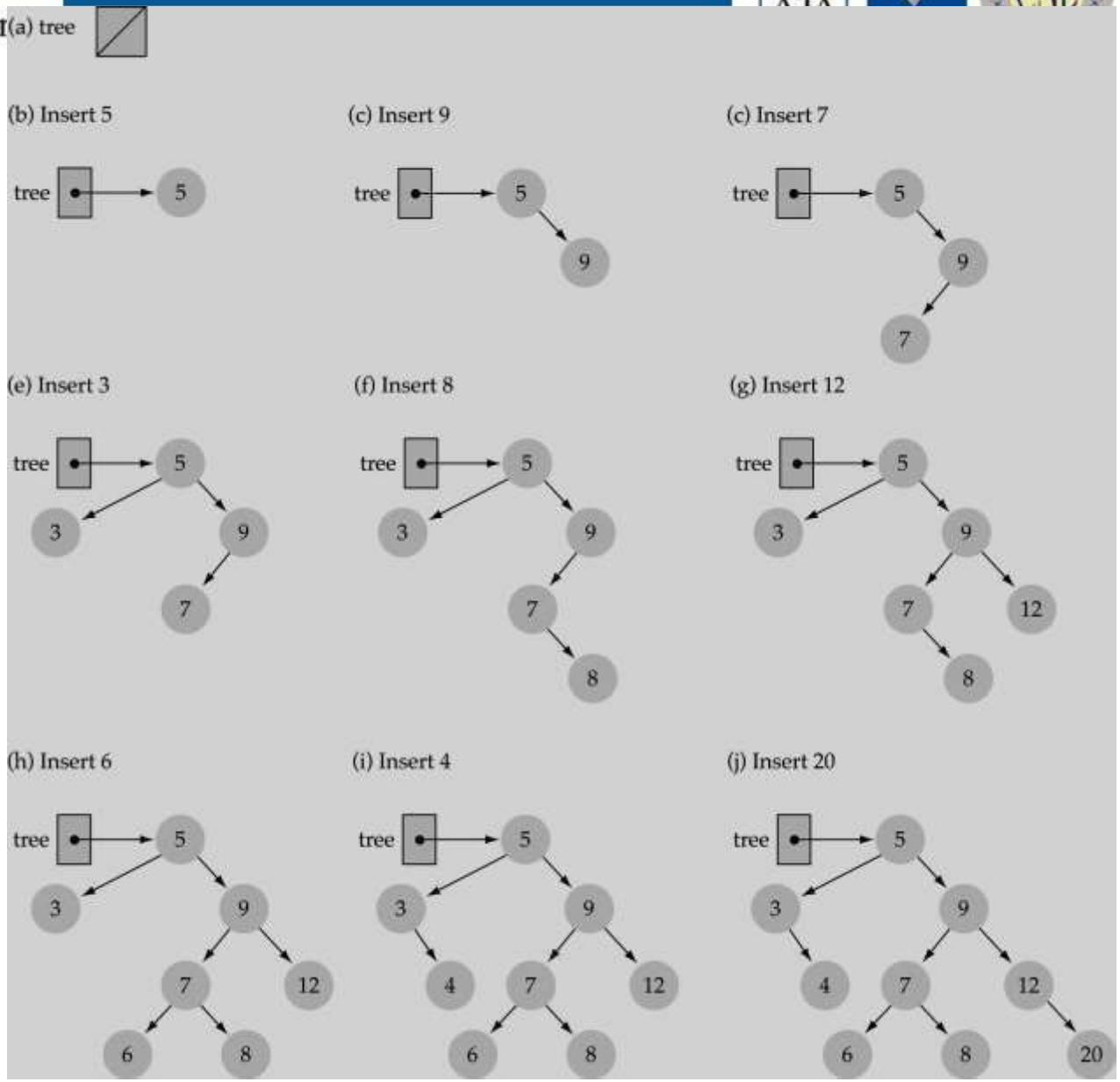
- What is the size of the problem?
  Number of nodes in the tree we are examining
- What is the base case(s)?
  1) When the key is found
  2) The tree is empty (key was not found)
- What is the general case?
  Search in the left or right subtrees

Function RetrieveItem (cont.)

```
template <class ItemType>
void TreeType<ItemType>:: RetrieveItem(ItemType& item,bool& found)
{
 Retrieve(root, item, found);
}
template<class ItemType>
void Retrieve(TreeNode<ItemType>* tree,ItemType& item,bool& found)
{
 if (tree == NULL) // base case 2
   found = false;
 else if(item < tree->info)
 Retrieve(tree->left, item, found);
 else if(item > tree->info)
 Retrieve(tree->right, item, found);
 else {  // base case 1
   item = tree->info;
   found = true;
 }
}
```

Function InsertItem

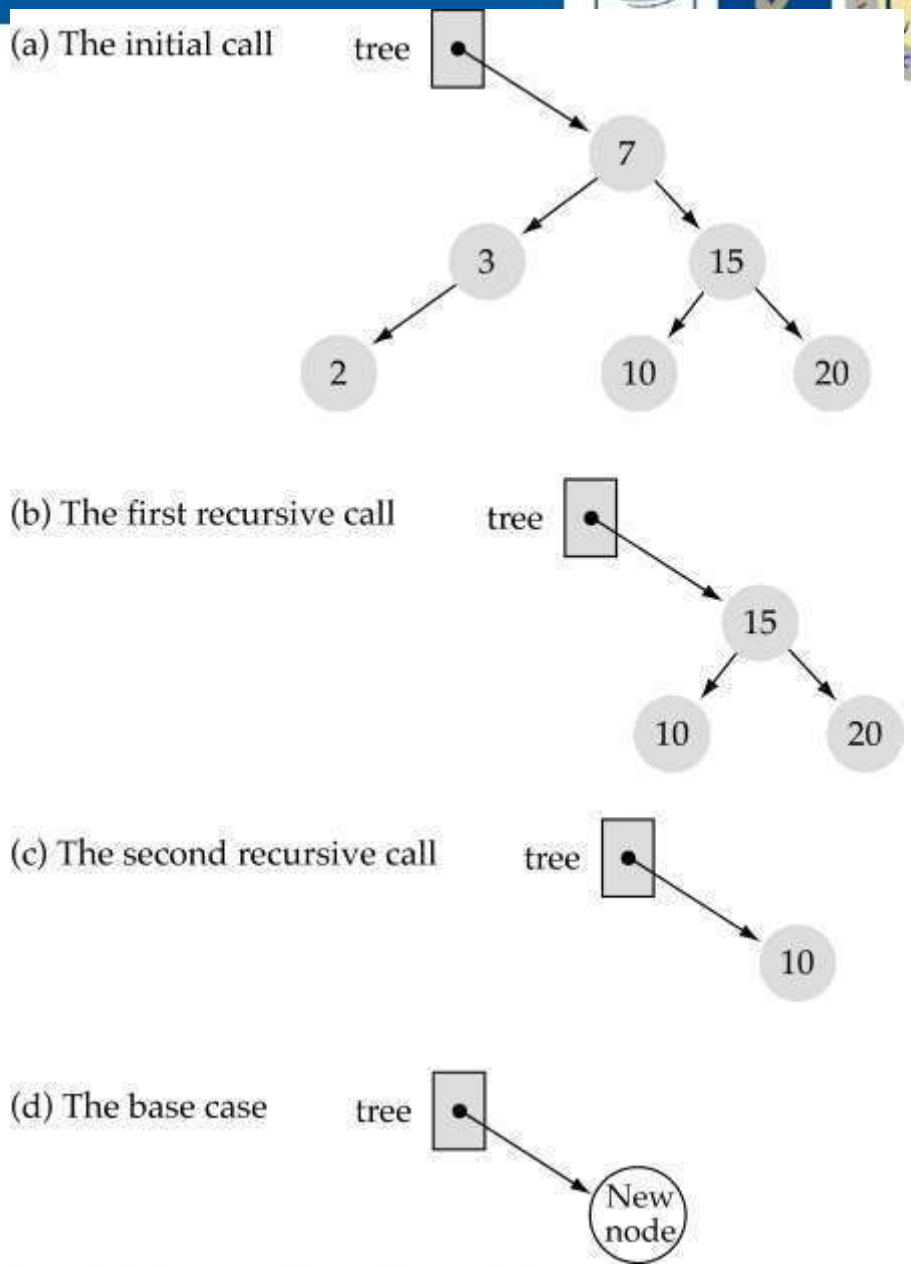
- Use the binary search tree property to insert the new item at the correct place



Function InsertItem (cont.)

-

Insert 11



Function InsertItem (cont.)

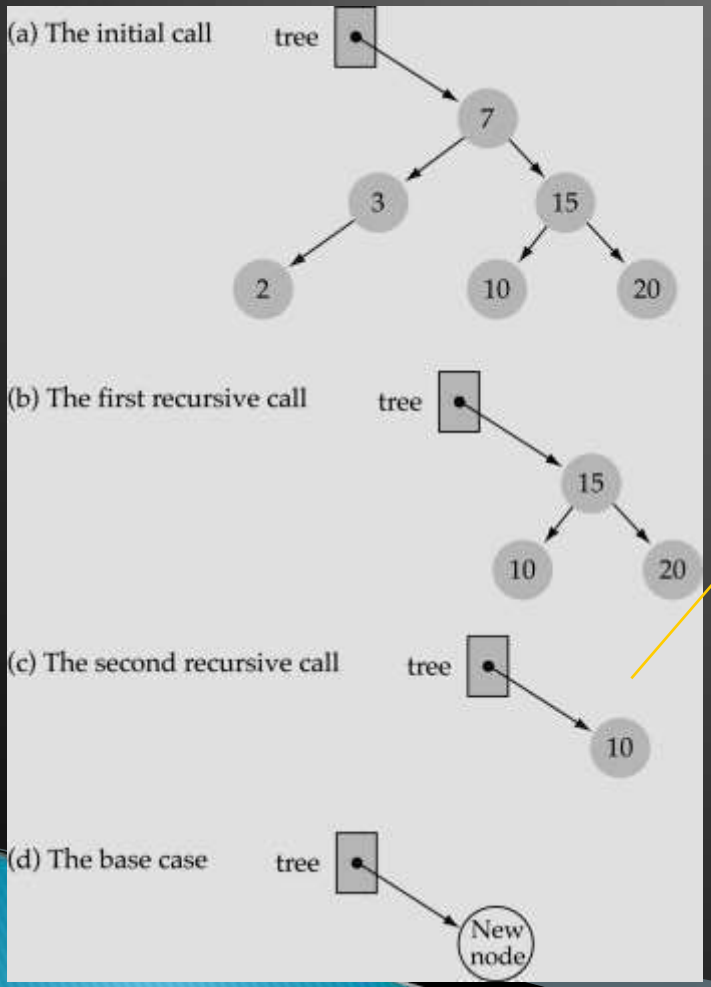
- What is the size of the problem?
  - Number of nodes in the tree we are examining
- What is the base case(s)?
  - The tree is empty
- What is the general case?
  - Choose the left or right subtree

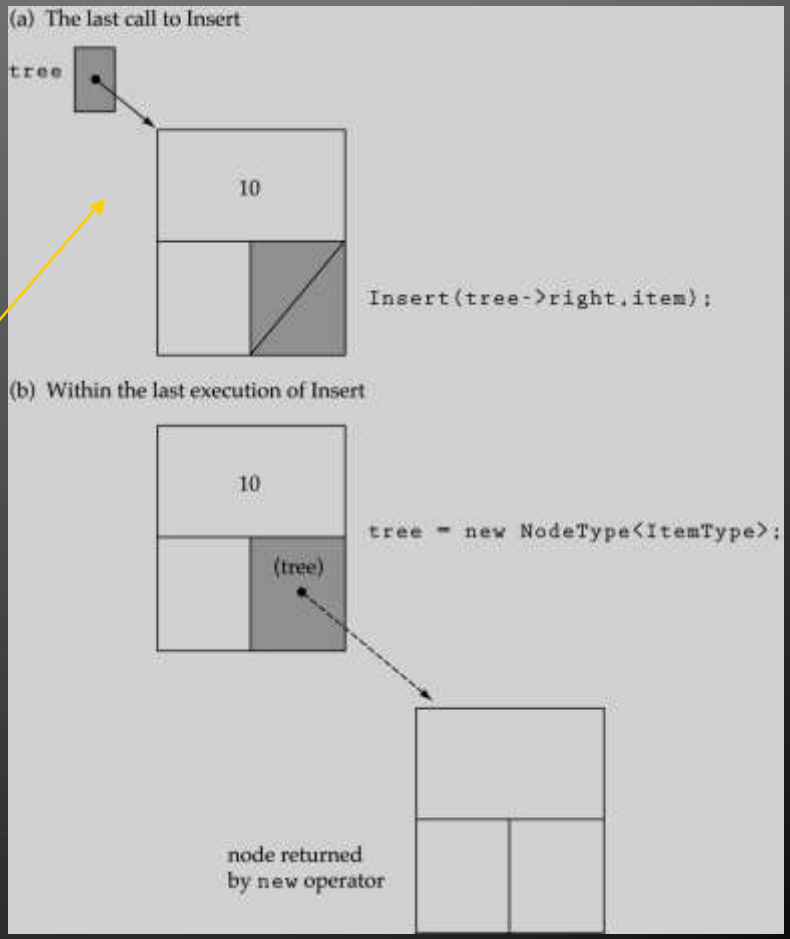
Function InsertItem (cont.)

```
template<class ItemType>
void TreeType<ItemType>::InsertItem(ItemType item)
{
 Insert(root, item);
}

template<class ItemType>
void Insert(TreeNode<ItemType>*& tree, ItemType item)
{
 if(tree == NULL) {  // base case
   tree = new TreeNode<ItemType>;
   tree->right = NULL;
   tree->left = NULL;
   tree->info = item;
 }
 else if(item < tree->info)
 Insert(tree->left,    item);
 else
   Insert(tree->right, item);
}
```

Function InsertItem (cont.)

Insert 11

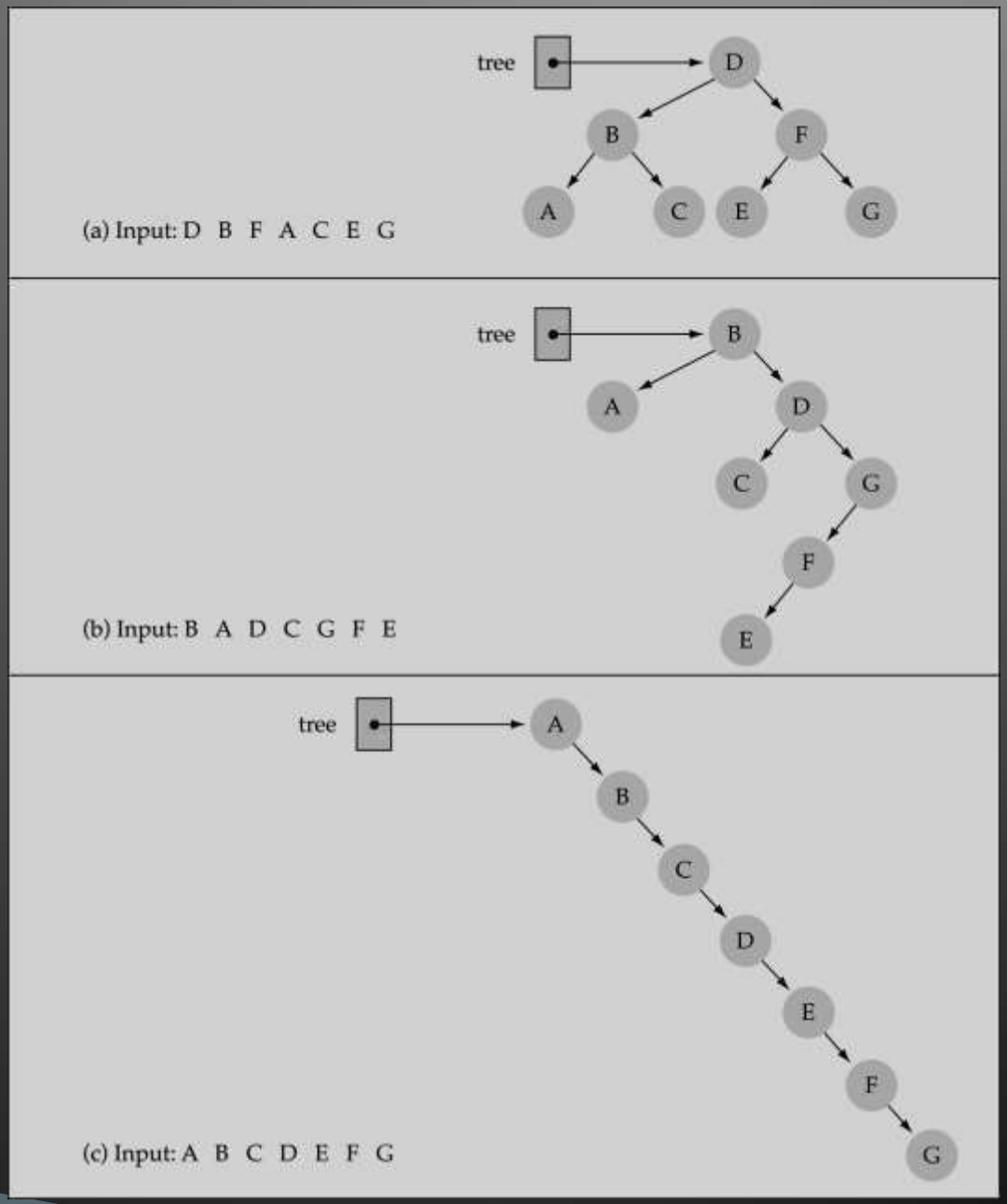




Does the order of inserting elements into a tree matter?

- Yes, certain orders produce very unbalanced trees!!
- Unbalanced trees are not desirable because search time increases!!
- There are advanced tree structures (e.g.,"red-black trees") which guarantee balanced trees

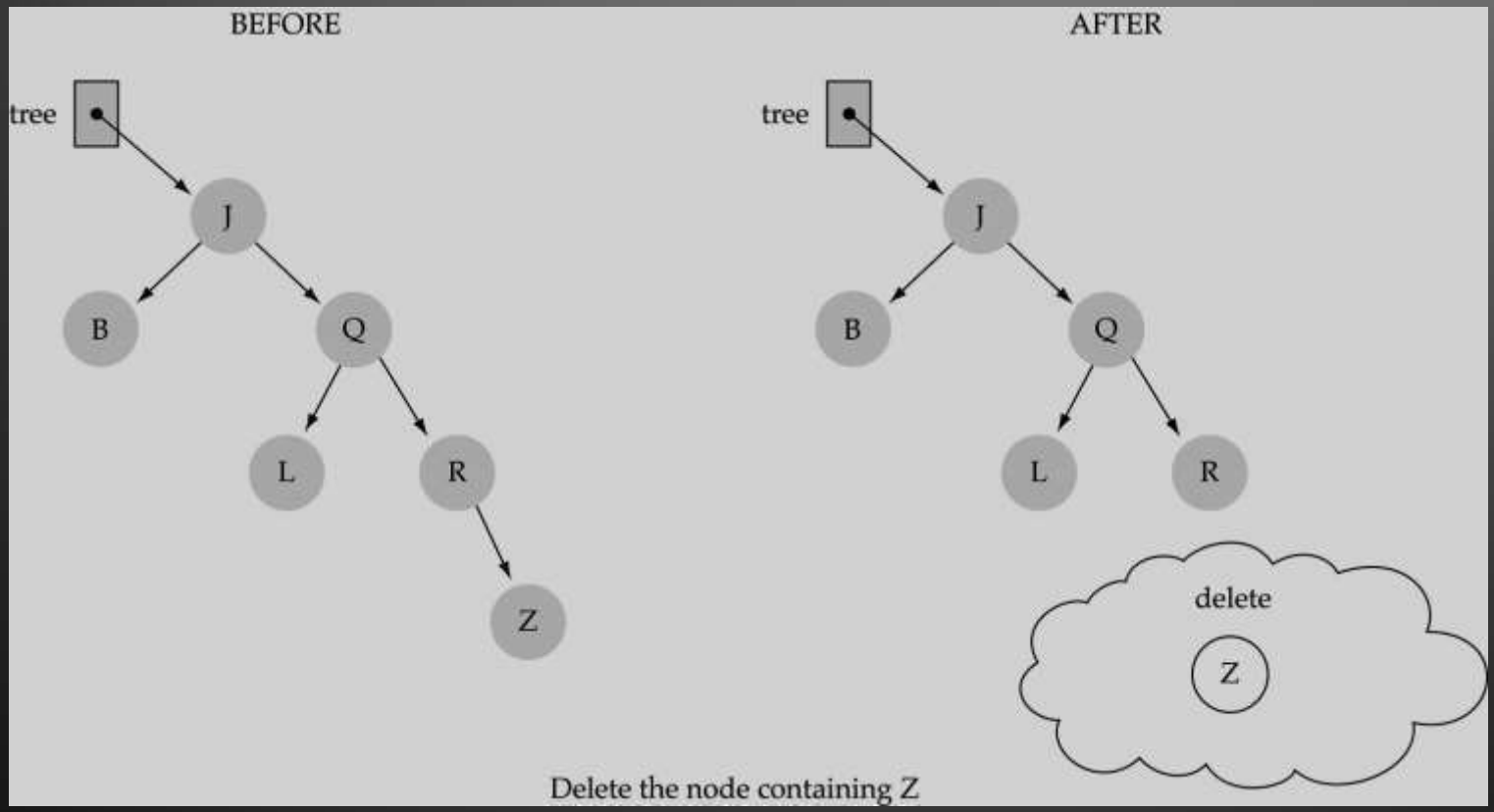
Does the order of inserting elements into a tree matter? (cont.)



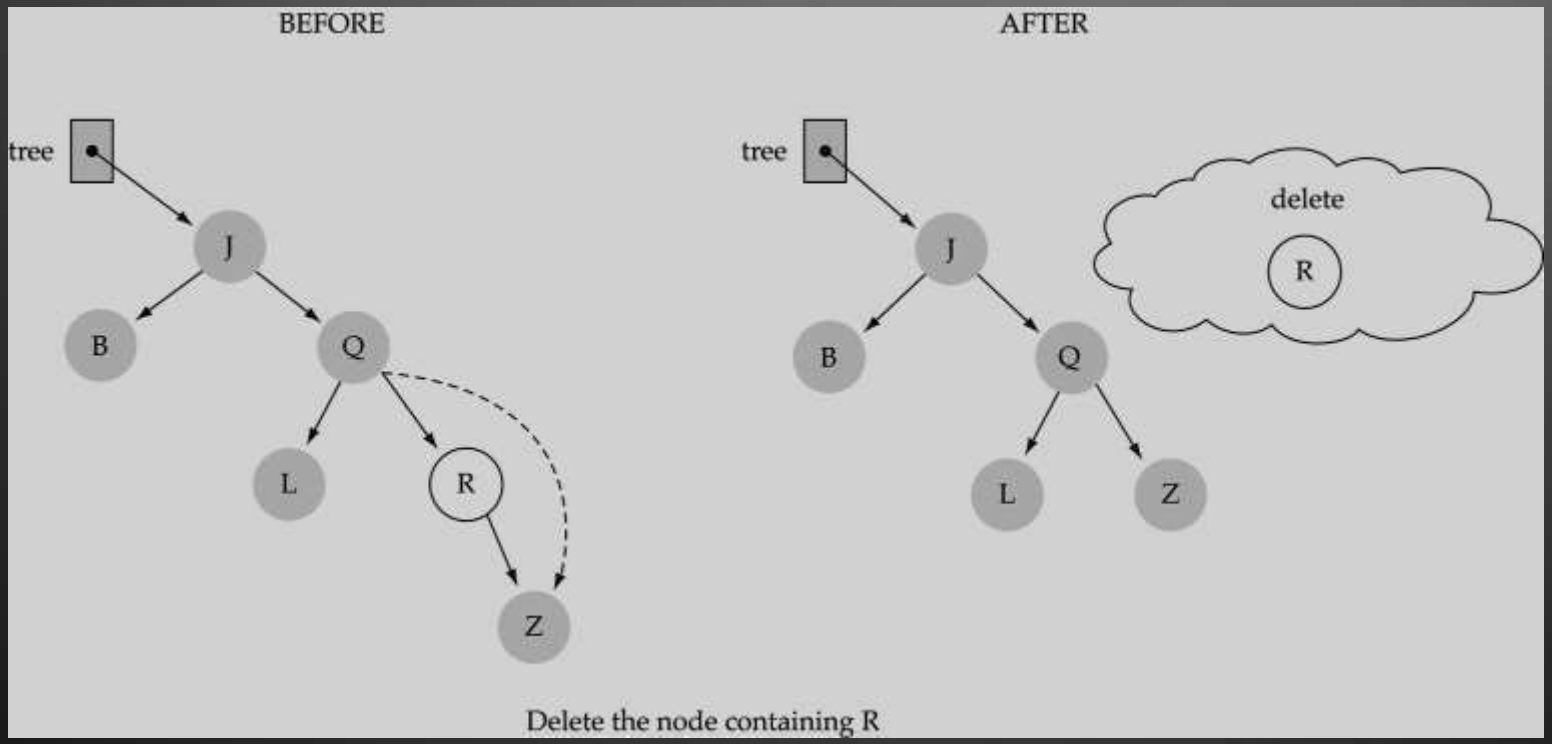
Function DeleteItem

- First, find the item; then, delete it
- Important: binary search tree property must be preserved!!
- We need to consider three different cases:
  (1) Deleting a leaf
  (2) Deleting a node with only one child
  (3) Deleting a node with two children

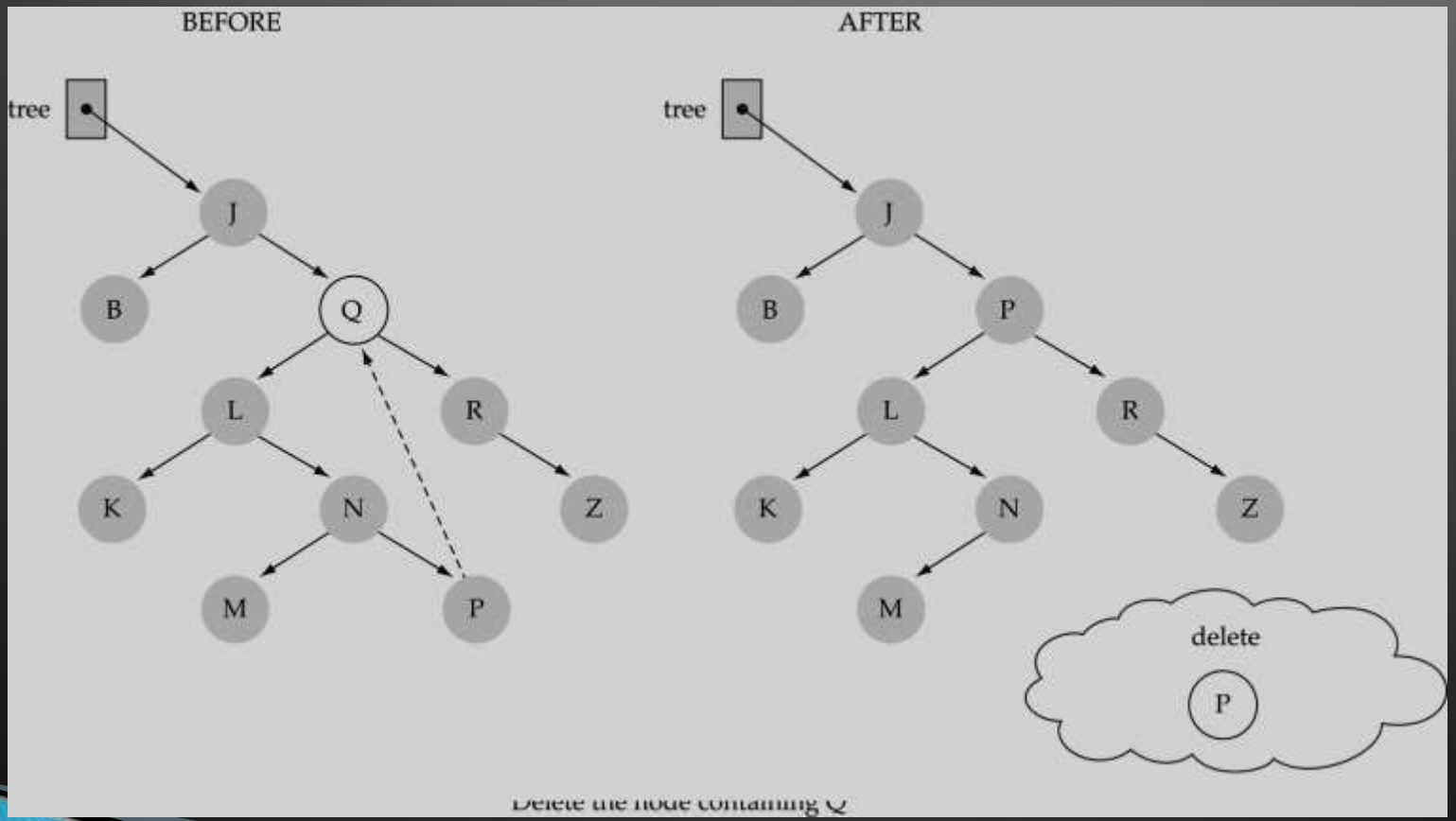
(1) Deleting a leaf



(2) Deleting a node with only one child



(3) Deleting a node with two children



(3) Deleting a node with two children (cont.)

- Find predecessor (it is the rightmost node in the left subtree)
- Replace the data of the node to be deleted with predecessor's data
- Delete predecessor node

Function DeleteItem (cont.)

- What is the size of the problem?
  - Number of nodes in the tree we are examining
- What is the base case(s)?
  - Key to be deleted was found
- What is the general case?
  - Choose the left or right subtree

Function DeleteItem (cont.)

```
template<class ItemType>
void TreeType<ItmeType>::DeleteItem(ItemType  item)
{
 Delete(root, item);
}
```

```
template<class ItemType>
void Delete(TreeNode<ItemType>*& tree, ItemType item)
{
 if(item < tree->info)
 Delete(tree->left, item);
 else if(item > tree->info)
 Delete(tree->right, item);
 else
   DeleteNode(tree);
}
```

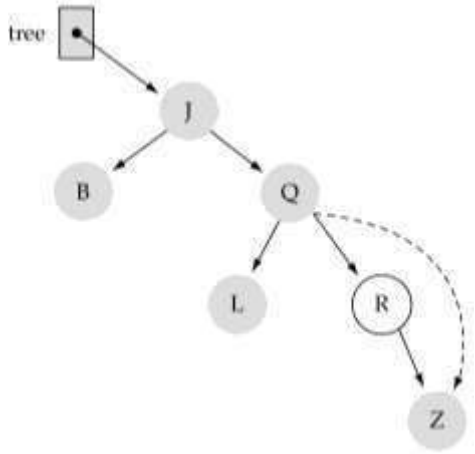
Function DeleteItem (cont.)

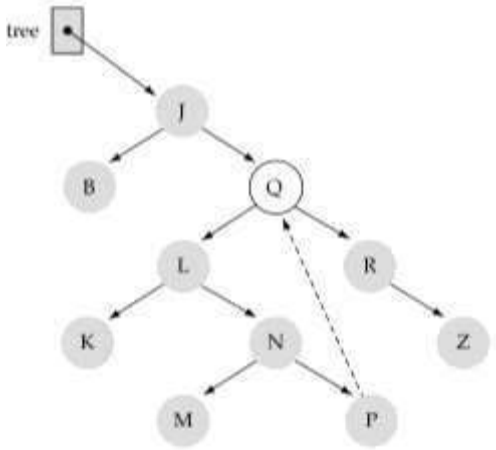
```
template <class ItemType>
void DeleteNode(TreeNode<ItemType>*& tree)
{
 ItemType data;
 TreeNode<ItemType>* tempPtr;

 tempPtr = tree;
 if(tree->left == NULL) { //right child
   tree = tree->right;
   delete tempPtr;
 }
 else if(tree->right == NULL) { // left child
   tree = tree->left;
   delete tempPtr;
 }
 else {
   GetPredecessor(tree->left, data);
   tree->info = data;
   Delete(tree->left, data);
 }
}
```

0 or 1 child

0 or 1 child

2 children





Function DeleteItem (cont.)

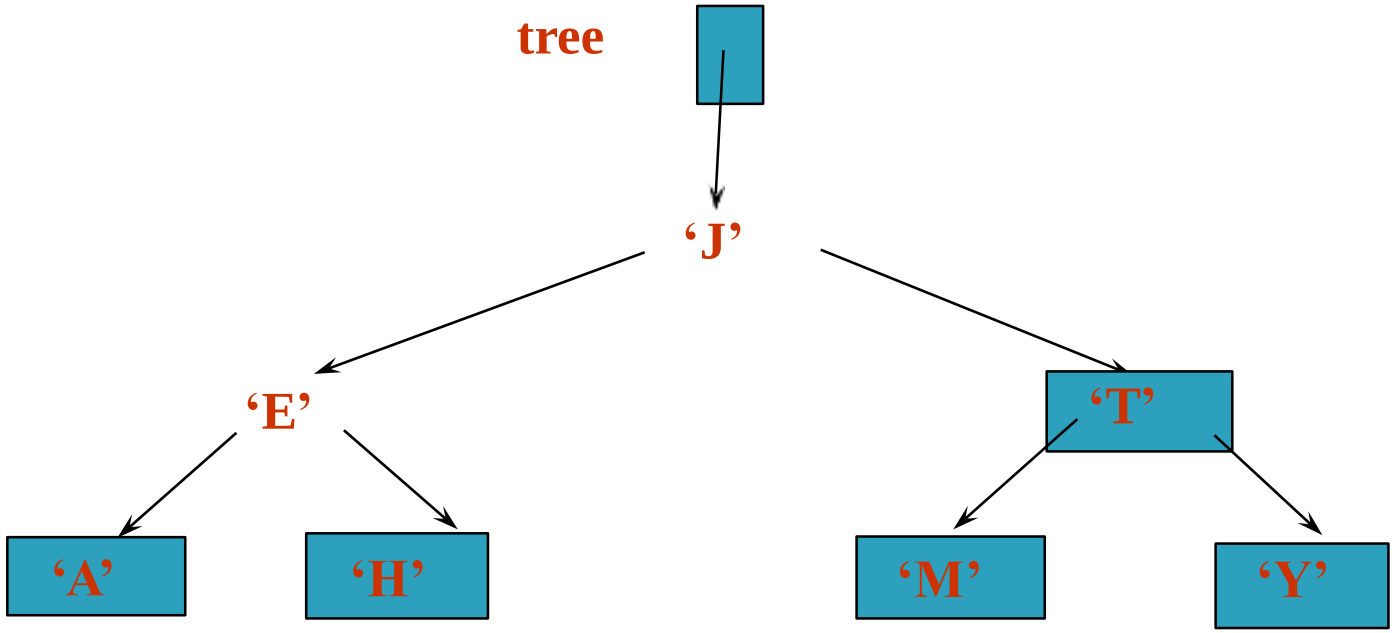
```
template<class ItemType>
void GetPredecessor(TreeNode<ItemType>* tree, ItemType& data)
{
 while(tree->right != NULL)
   tree = tree->right;
 data = tree->info;
}
```

Tree Traversals

There are mainly three ways to traverse a tree:

- Inorder Traversal
- Postorder Traversal
- Preorder Traversal

Inorder Traversal: A E H J M T Y



Visit left subtree first

Visit right subtree last

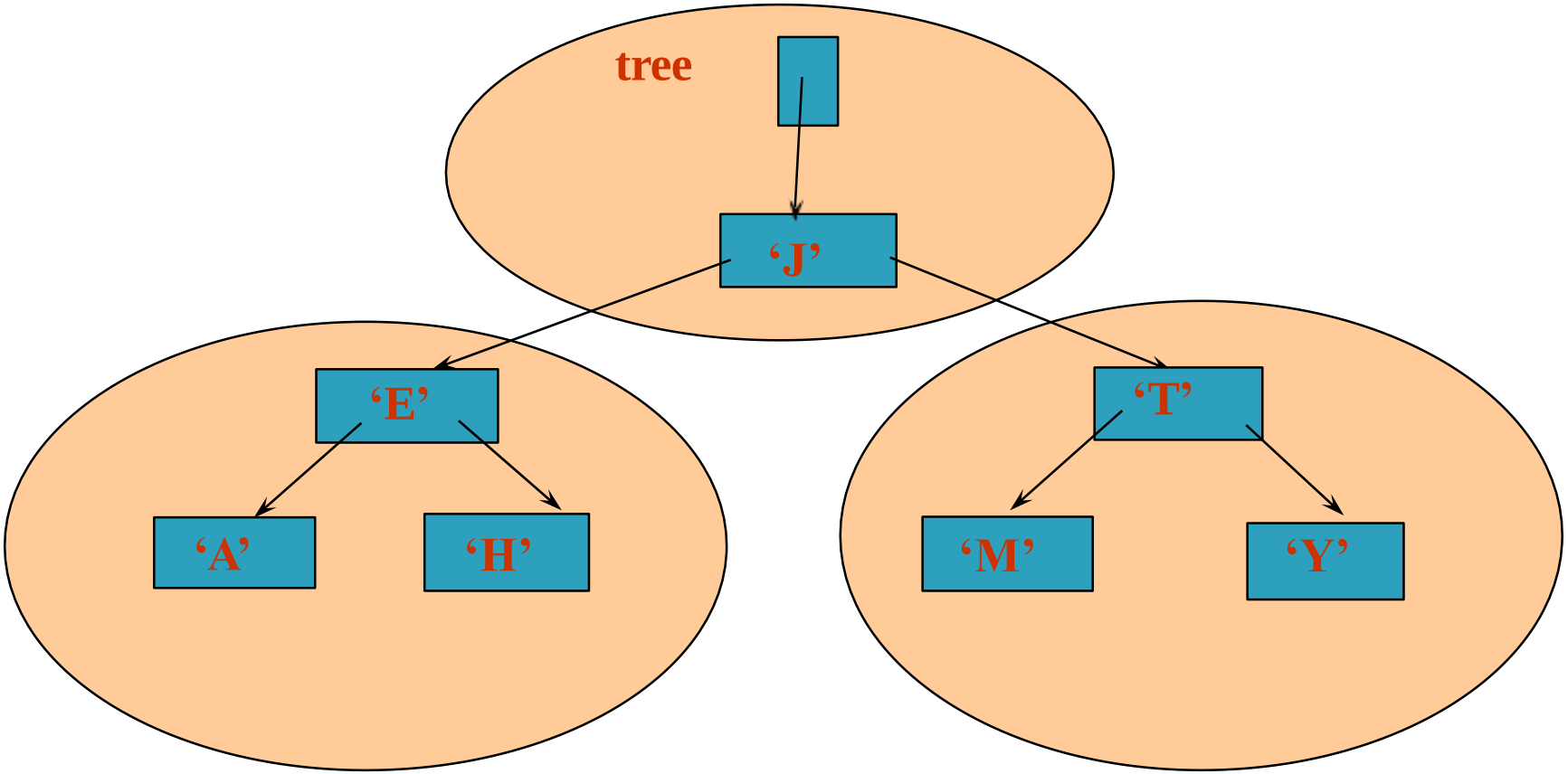
Inorder Traversal

- Visit the nodes in the left subtree, then visit the root of the tree, then visit the nodes in the right subtree

```
Inorder(tree)

If tree is not NULL
  Inorder(Left(tree))
  Visit Info(tree)
  Inorder(Right(tree))
```

(Warning: "visit" means that the algorithm does something with the values in the node, e.g., print the value)

Postorder



Visit left subtree first

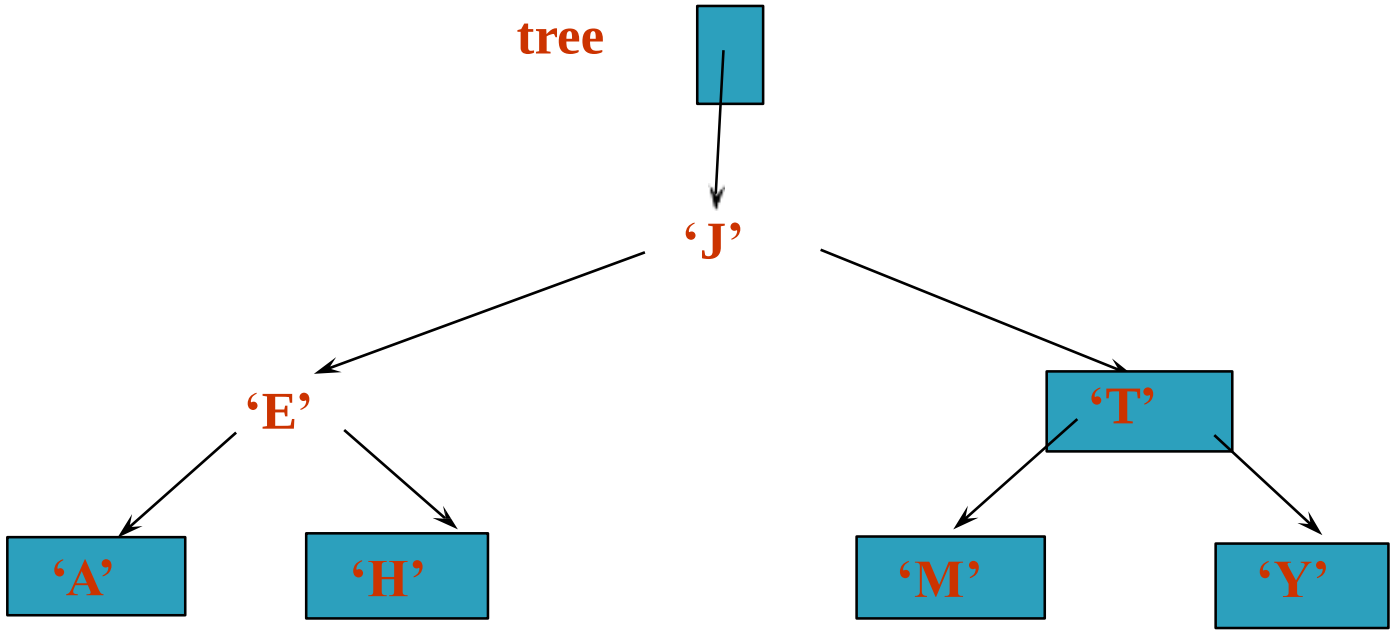
Visit right subtree second

Postorder Traversal

- Visit the nodes in the left subtree first, then visit the nodes in the right subtree, then visit the root of the tree

```
Postorder(tree)
If tree is not NULL
  Postorder(Left(tree))
  Postorder(Right(tree))
  Visit Info(tree)
```

Preorder Traversal: J E A H T M Y



Visit left subtree second

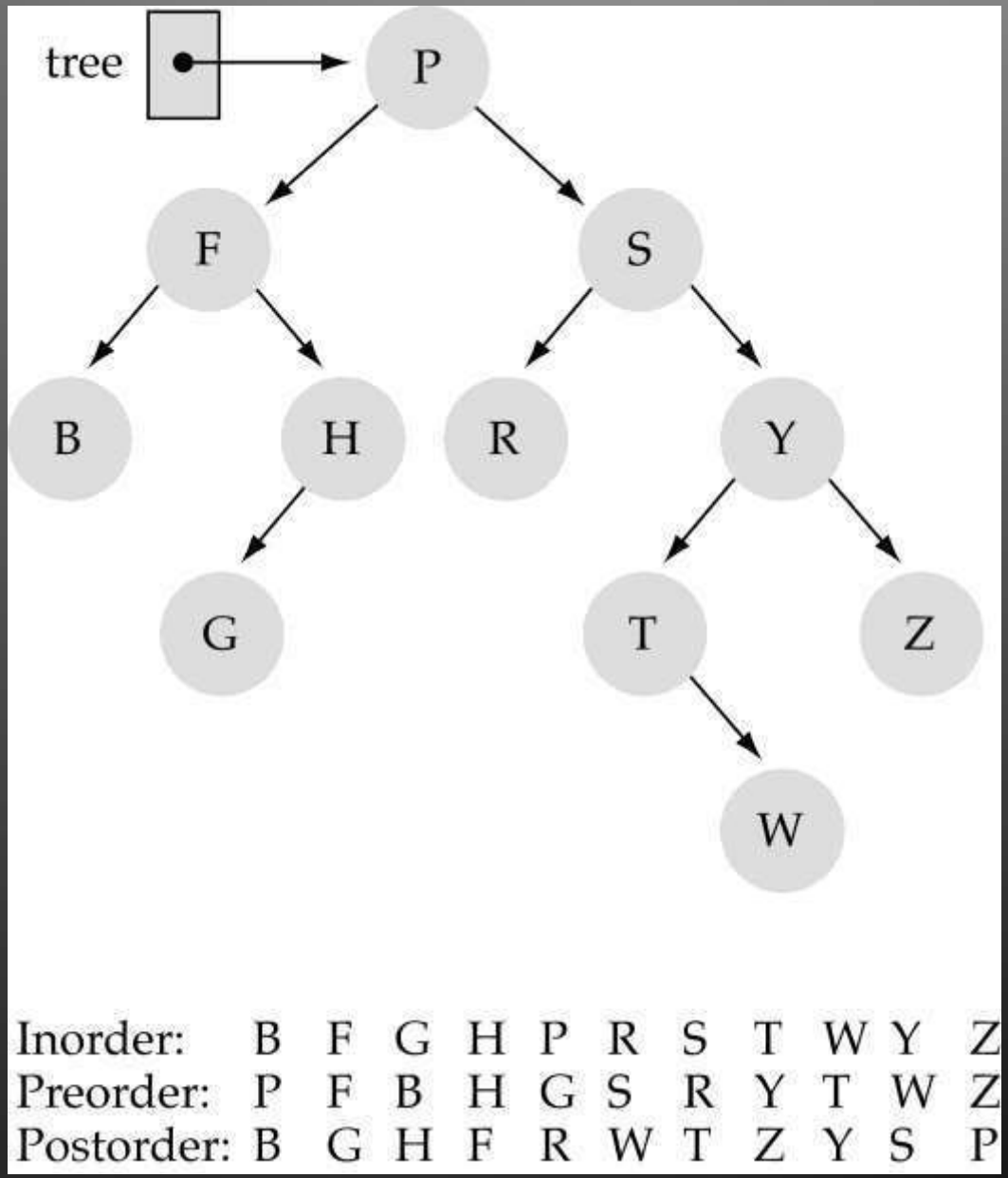
Visit right subtree last

Preorder Traversal

- Visit the root of the tree first, then visit the nodes in the left subtree, then visit the nodes in the right subtree

```
Preorder(tree)
If tree is not NULL
  Visit Info(tree)
  Preorder(Left(tree))
  Preorder(Right(tree))
```

Tree Traversals



Inorder:	B	F	G	H	P	R	S	T	W	Y	Z
Preorder:	P	F	B	H	G	S	R	Y	T	W	Z
Postorder:	B	G	H	F	R	W	T	Z	Y	S	P

Function PrintTree

- We use "inorder" to print out the node values
- Why??
- *Hint*: use binary search trees for sorting !!

Why?? (keys are printed out in ascending

Function PrintTree (cont.)

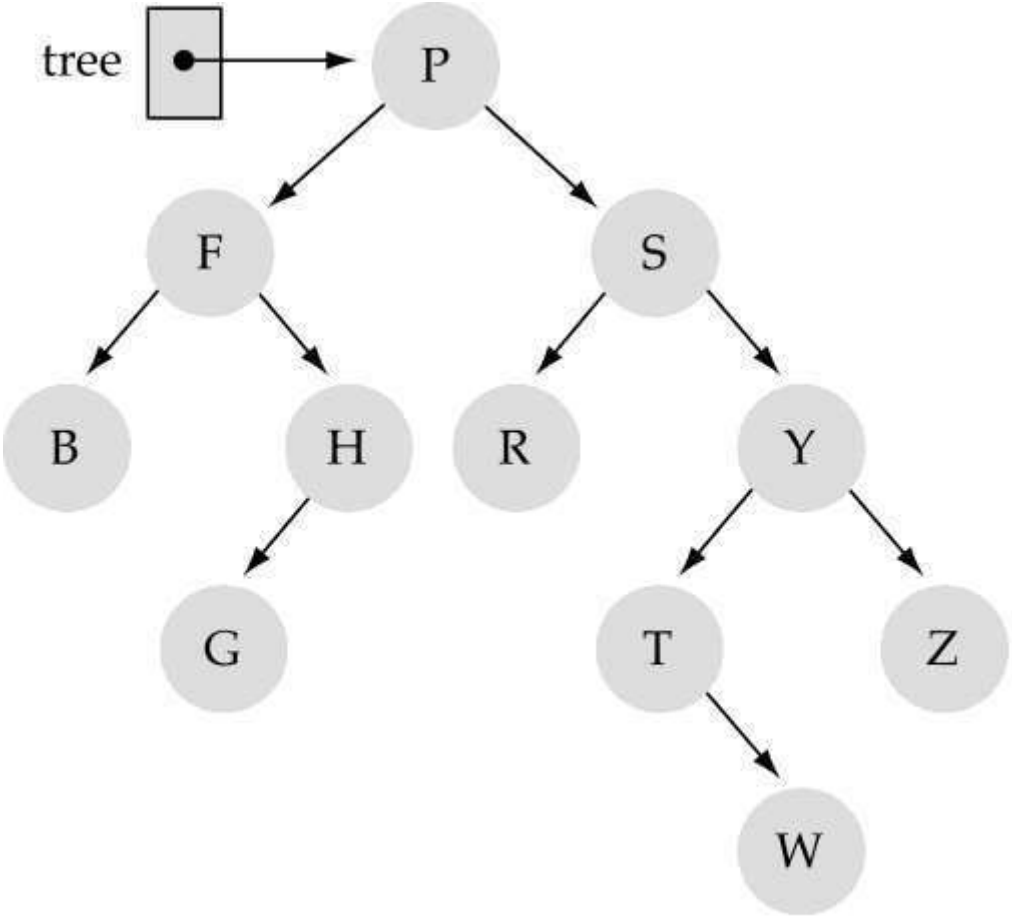
```
void TreeType::PrintTree(ofstream& outFile)
{
 Print(root, outFile);
}

template<class ItemType>
void Print(TreeNode<ItemType>* tree, ofstream& outFile)
{
 if(tree != NULL) {
   Print(tree->left, outFile);
   outFile << tree->info;
   Print(tree->right, outFile);
 }
}
```

(see textbook for overloading << and >>)

Class Constructor

```
template<class ItemType>
TreeType<ItemType>::TreeType()
{
 root = NULL;
}
```

Class Destructor

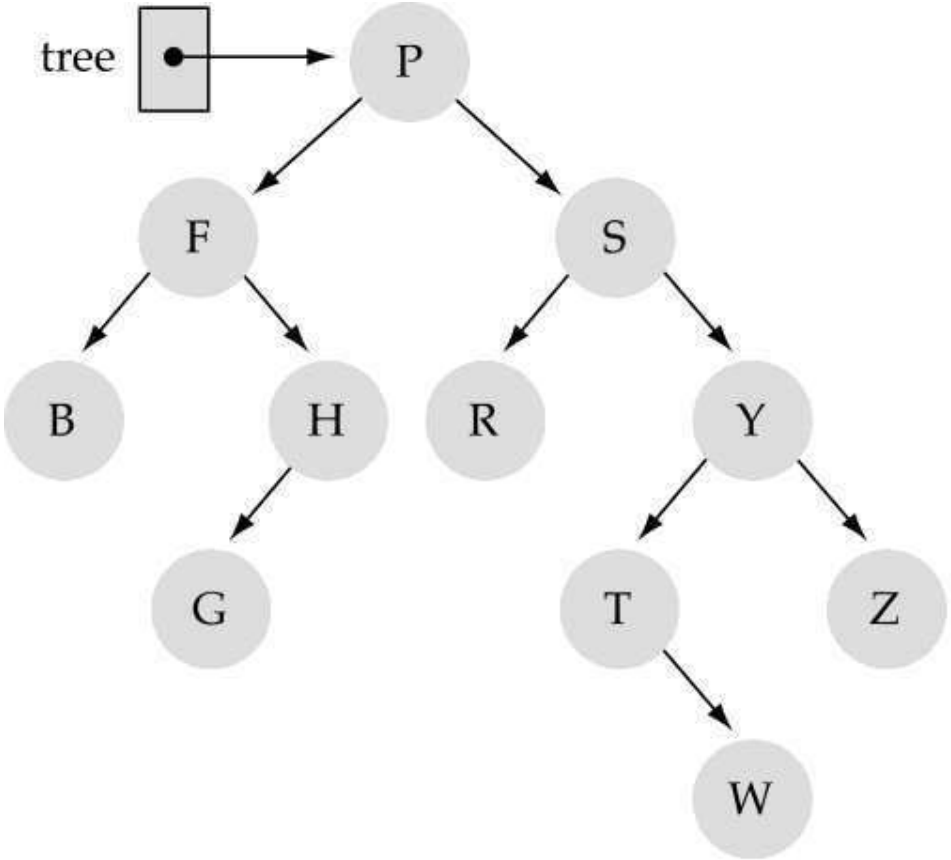


Class Destructor (cont'd)

- Delete the tree in a "bottom-up" fashion
- *Postorder traversal* is appropriate for this !!

```
TreeType::~TreeType()
{
 Destroy(root);
}
void Destroy(TreeNode<ItemType>*& tree)
{if(tree != NULL) {
    Destroy(tree->left);
    Destroy(tree->right);
    delete tree;
  }
}
```

Copy Constructor



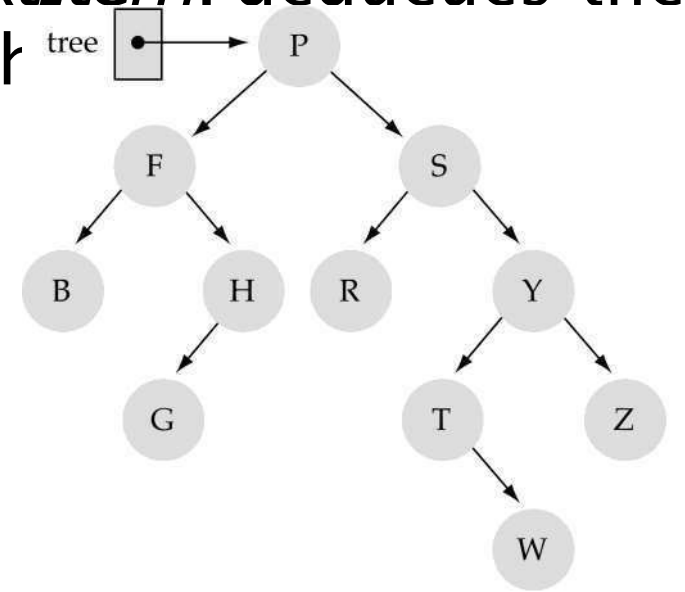
Copy Constructor (cont'd)

```
template<class ItemType>
TreeType<ItemType>::TreeType(const TreeType<ItemType>&
                             originalTree)
{
 CopyTree(root, originalTree.root);
}

template<class ItemType)
void CopyTree(TreeNode<ItemType>*& copy,
    TreeNode<ItemType>* originalTree)
{
if(originalTree == NULL)
   copy = NULL;
 else {
   copy = new TreeNode<ItemType>;
   copy->info = originalTree->info;
   CopyTree(copy->left, originalTree->left);
   CopyTree(copy->right, originalTree->right);
 }
}
```

preorder

ResetTree and GetNextItem

- The user is allowed to specify the tree traversal order
- For efficiency, *ResetTree* stores in a queue the results of the specified tree traversal
- Then, *GetNextItem*. deqaueues the node values from th



ResetTree and GetNextItem (cont.)

(specification file)

```
enum OrderType {PRE_ORDER,
IN_ORDER,
  POST_ORDER};
template<class ItemType>
class TreeType {
  public:
   // same as before
  private:
    TreeNode<ItemType>* root;
    QueType<ItemType> preQue;
    QueType<ItemType> inQue;
    QueType<ItemType> postQue;
};
```

new private data

ResetTree and GetNextItem (cont.)

```
template<class ItemType>
void PreOrder(TreeNode<ItemType>*,
   QueType<ItemType>&);

template<class ItemType>
void InOrder(TreeNode<ItemType>*,
   QueType<ItemType>&);

template<class ItemType>
void PostOrder(TreeNode<ItemType>*,
   QueType<ItemType>&);
```

ResetTree and GetNextItem (cont.)

```
template<class ItemType>
void PreOrder(TreeNode<ItemType>tree,
   QueType<ItemType>& preQue)
{
 if(tree != NULL) {
   preQue.Enqueue(tree->info);
   PreOrder(tree->left, preQue);
   PreOrder(tree->right, preQue);
 }
}
```

ResetTree and GetNextItem (cont.)

```
template<class ItemType>
void InOrder(TreeNode<ItemType>tree,
   QueType<ItemType>& inQue)
{
 if(tree != NULL) {
   InOrder(tree->left, inQue);
   inQue.Enqueue(tree->info);
   InOrder(tree->right, inQue);
 }
}
```

ResetTree and GetNextItem (cont.)

```
template<class ItemType>
void PostOrder(TreeNode<ItemType>tree,
   QueType<ItemType>& postQue)
{
 if(tree != NULL) {
   PostOrder(tree->left, postQue);
   PostOrder(tree->right, postQue);
   postQue.Enqueue(tree->info);
 }
}
```

The function *ResetTree*

```
template<class ItemType>
void TreeType<ItemType>::ResetTree(OrderType order)
{
 switch(order) {
   case PRE_ORDER: PreOrder(root, preQue);
                   break;
   case IN_ORDER:  InOrder(root, inQue);
                   break;
   case POST_ORDER: PostOrder(root, postQue);
                   break;
 }
}
```

The function GetNextItem

```
template<class ItemType>
void TreeType<ItemType>::GetNextItem(ItemType& item,
    OrderType order, bool& finished)
{finished = false;
 switch(order) {
   case PRE_ORDER: preQue.Dequeue(item);
                   if(preQue.IsEmpty())
                   finished = true;
                   break;
   case IN_ORDER: inQue.Dequeue(item);
                   if(inQue.IsEmpty())
                   finished = true;
                   break;
   case POST_ORDER: postQue.Dequeue(item);
                   if(postQue.IsEmpty())
                     finished = true;
                   break;
 }
}
```

Iterative Insertion and Deletion

- See textbook

Comparing Binary Search Trees to Linear Lists

Big-O Comparison			
Operation	Binary Search Tree	Array-based List	Linked List
Constructor	O(1)	O(1)	O(1)
Destructor	O(N)	O(1)	O(N)
IsFull	O(1)	O(1)	O(1)
IsEmpty	O(1)	O(1)	O(1)
RetrieveItem	O(logN)	O(logN)	O(N)
InsertItem	O(logN)	O(N)	O(N)
DeleteItem	O(logN)	O(N)	O(N)

Exercises

- 1–3, 8–18, 21, 22, 29–32

MODULE 5

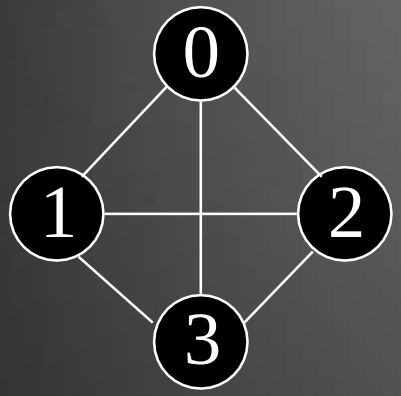
GRAPHS

Definition

- A graph G consists of two sets
  - a finite, nonempty set of vertices V(G)
  - a finite, possible empty set of edges E(G)
  - G(V,E) represents a graph
- An undirected graph is one in which the pair of vertices in a edge is unordered, $(v_0, v_1) = (v_1, v_0)$
- A directed graph is one in which each edge is a directed pair of vertices, $\langle v_0, v_1 \rangle$!= $\langle v_1, v_0 \rangle$

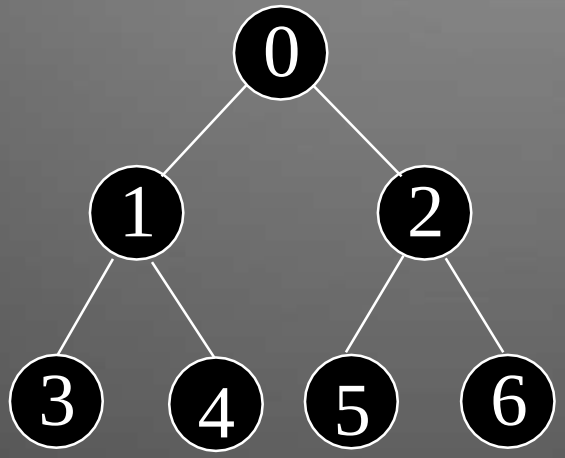
tail ——————→ head

Examples for Graph



G_1

complete graph



G_2

incomplete graph

G_3

$V(G_1)=\{0,1,2,3\}$
$V(G_2)=\{0,1,2,3,4,5,6\}$
$V(G_3)=\{0,1,2\}$

$E(G_1)=\{(0,1),(0,2),(0,3),(1,2),(1,3),(2,3)\}$
$E(G_2)=\{(0,1),(0,2),(1,3),(1,4),(2,5),(2,6)\}$
$E(G_3)=\{<0,1>,<1,0>,<1,2>\}$

complete undirected graph: n(n-1)/2 edges
complete directed graph: n(n-1) edges

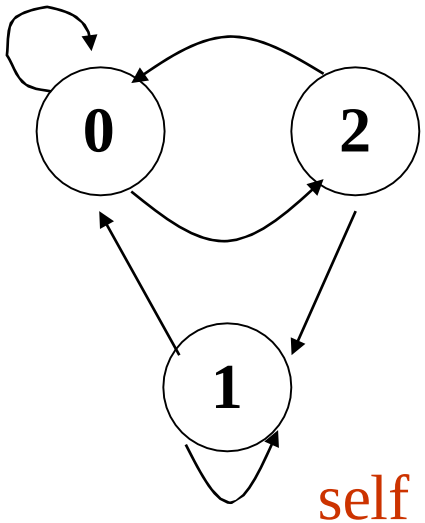
Complete Graph

- A complete graph is a graph that has the maximum number of edges
  - for undirected graph with n vertices, the maximum number of edges is n(n-1)/2
  - for directed graph with n vertices, the maximum number of edges is n(n-1)
  - example: G1 is a complete graph

Adjacent and Incident

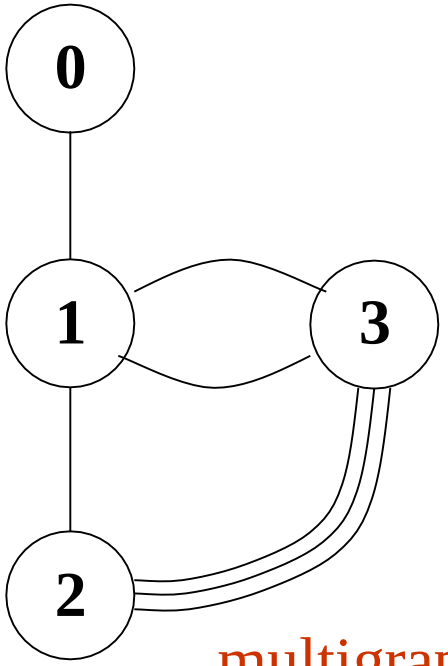
- If (v_0, v_1) is an edge in an undirected graph,
  - v_0 and v_1 are adjacent
  - The edge (v_0, v_1) is incident on vertices v_0 and v_1
- If $<v_0, v_1>$ is an edge in a directed graph
  - v_0 is adjacent to v_1, and v_1 is adjacent from v_0
  - The edge $<v_0, v_1>$ is incident on v_0 and v_1

***Figure 6.3:** Example of a graph with feedback loops and a multigraph (p.260)



self edge

(a)



(b)

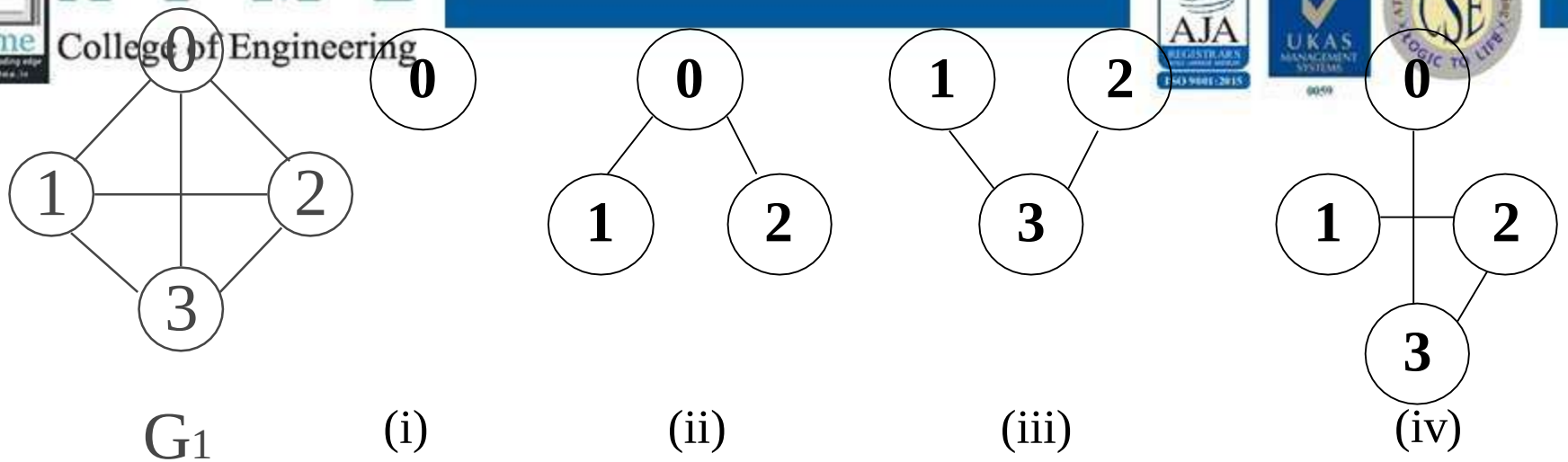
multigraph:
multiple occurrences
of the same edge

Figure 6.3

Subgraph and Path

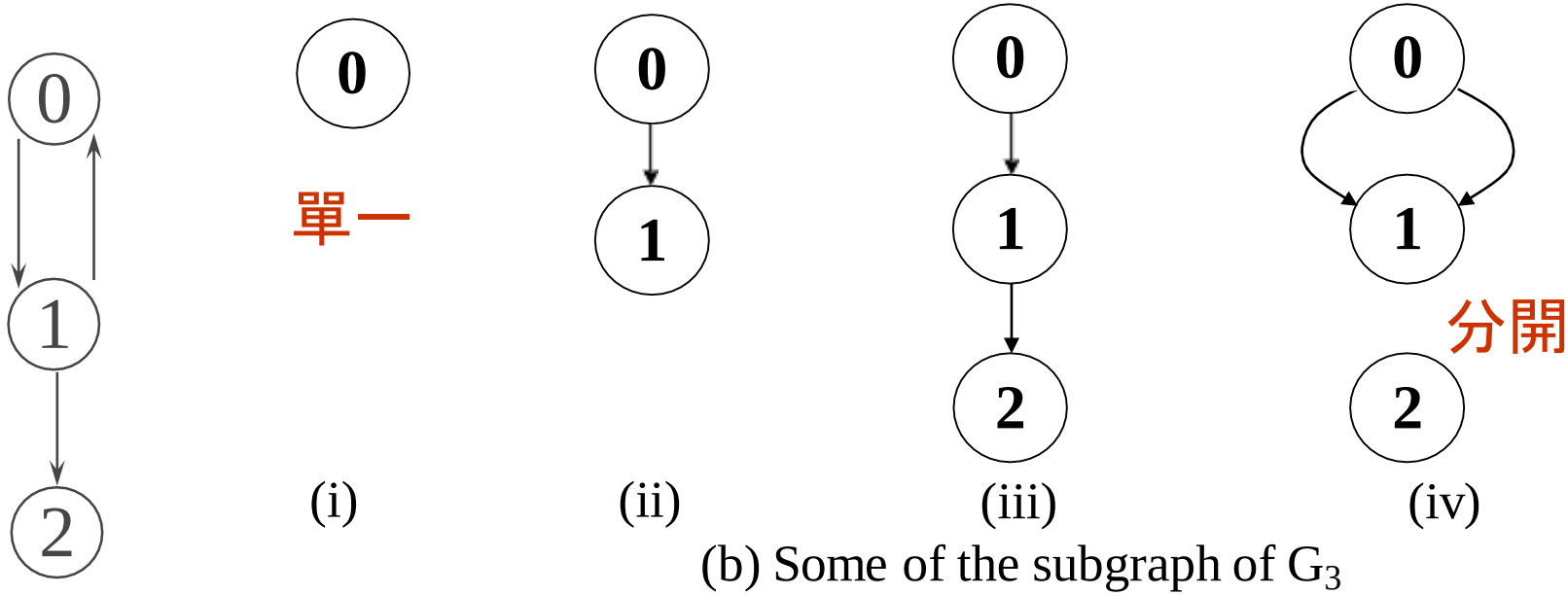
- A subgraph of G is a graph G'such that V(G') is a subset of V(G) and E(G') is a subset of E(G)
- A path from vertex v_p to vertex v_q in a graph G, is a sequence of vertices, v_p, v_{i1}, v_{i2}, ..., v_{in}, v_q, such that (v_p, v_{i1}), (v_{i1}, v_{i2}), ..., (v_{in}, v_q) are edges in an undirected graph
- The length of a path is the number of edges on it

Figure 6.4: subgraphs of G_1 and G_3 (p.261)



G_1 (i) (ii) (iii) (iv)

(a) Some of the subgraph of G_1



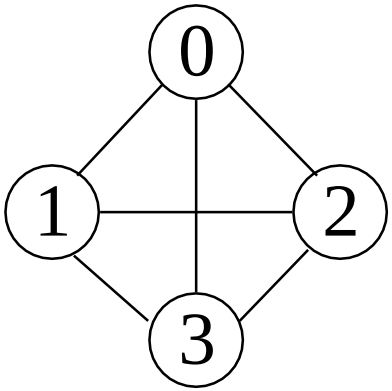
G_3 (i) (ii) (iii) (iv)

(b) Some of the subgraph of G_3

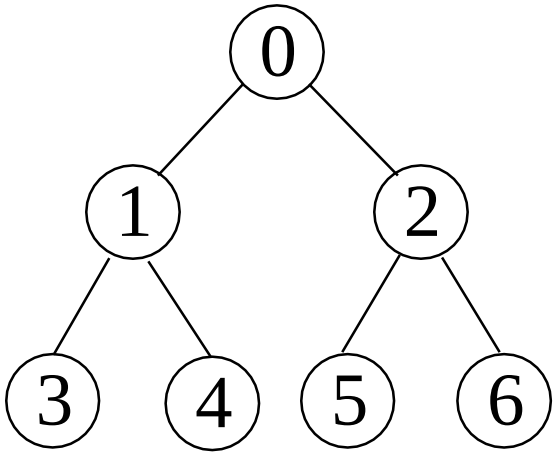
Simple Path and Style

- A simple path is a path in which all vertices, except possibly the first and the last, are distinct
- A cycle is a simple path in which the first and the last vertices are the same
- In an undirected graph G, two vertices, v_0 and v_1 are connected if there is a path in G from v_0 to v_1
- An undirected graph is connected if, for every pair of distinct vertices v_i, v_j, there is a path from v_i to v_j

connected



G_1



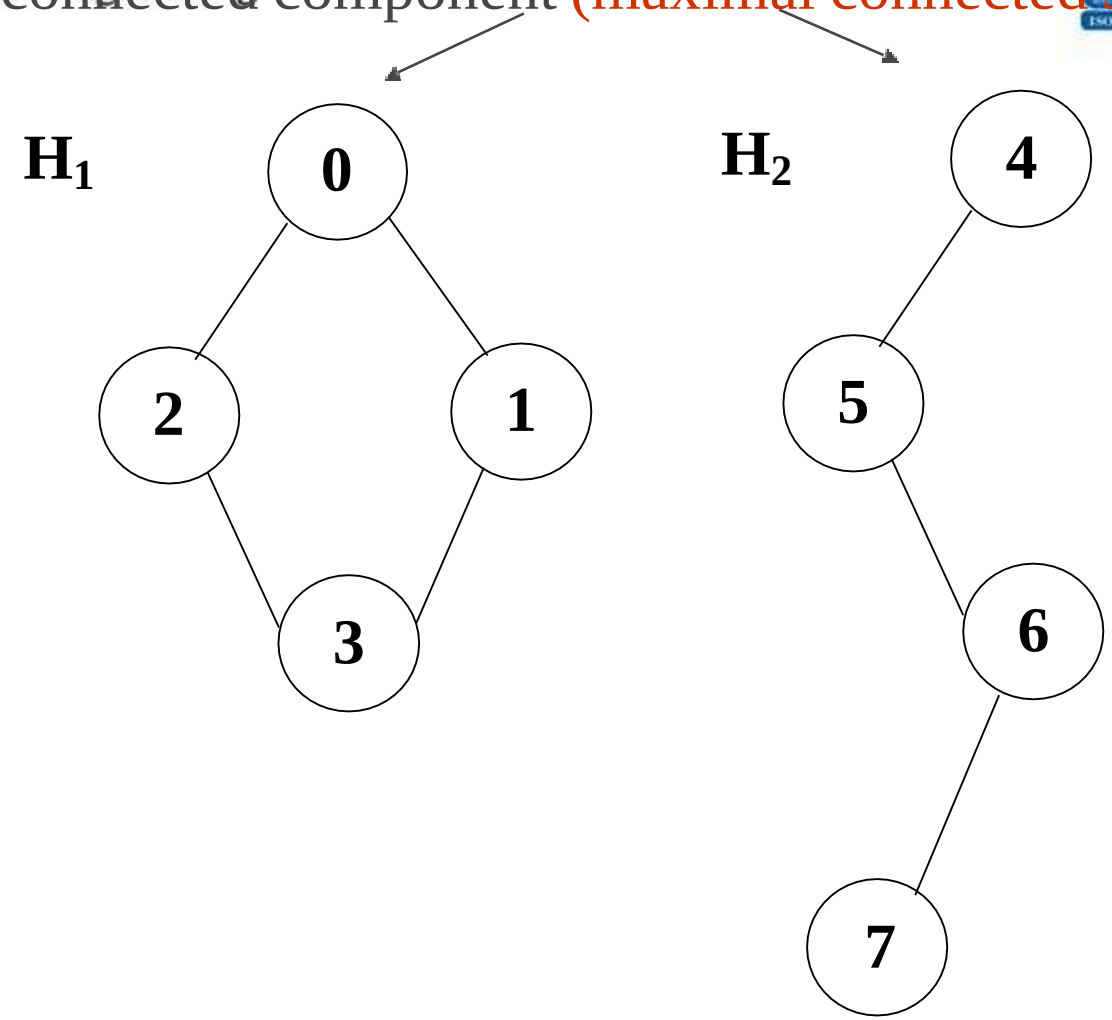
G_2

tree (acyclic graph)

Connected Component

- A connected component of an undirected graph is a maximal connected subgraph.
- A tree is a graph that is connected and acyclic.
- A directed graph is strongly connected if there is a directed path from v_i to v_j and also from v_j to v_i.
- A strongly connected component is a maximal subgraph that is strongly connected.

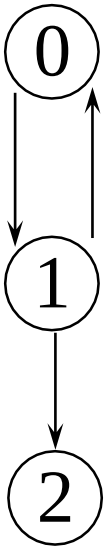
***Figure 6.5:** A graph with two connected components (p.262)



G_4 (not connected)

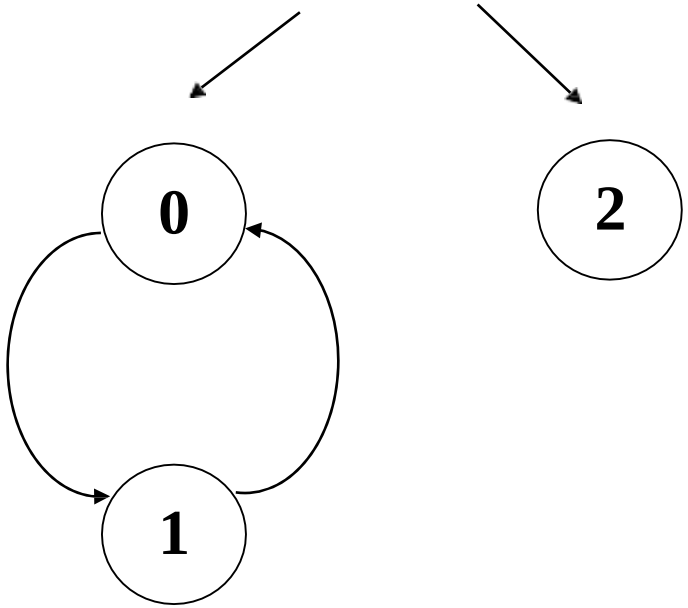
***Figure 6.6:** Strongly connected components of G_3 (p.262)

not strongly connected



G_3

strongly connected component
(maximal strongly connected subgraph)



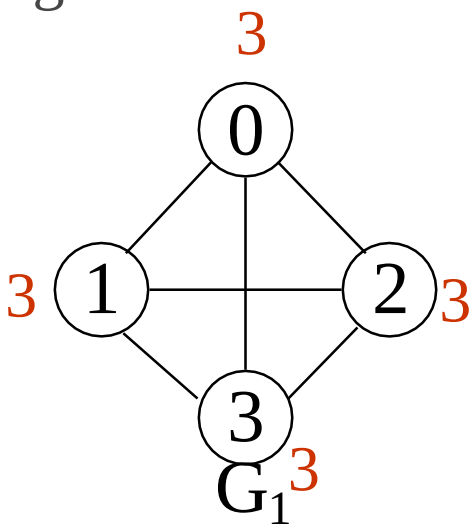
Degree

- The degree of a vertex is the number of edges incident to that vertex
- For directed graph,
  - the in-degree of a vertex *v* is the number of edges that have *v* as the head
  - the out-degree of a vertex *v* is the number of edges that have *v* as the tail
  - if d_i is the degree of a vertex *i* in a graph *G* with *n* vertices and *e* edges, the number of edges is

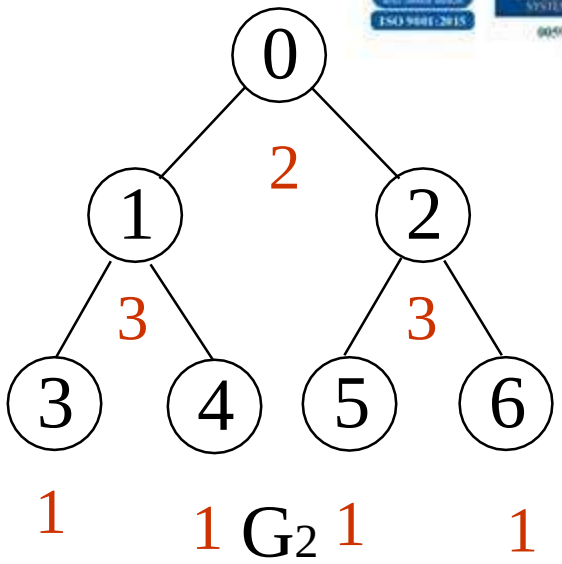
$$e = \left(\sum_{0}^{n-1} d_i\right) / 2$$

undirected graph

degree



G_1

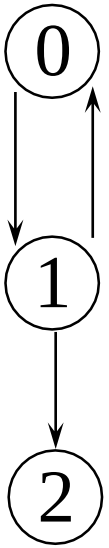


G_2

directed graph

in-degree

out-degree



in:1, out: 1

in: 1, out: 2

in: 1, out: 0

G_3

ADT for Graph

structure Graph is

objects: a nonempty set of vertices and a set of undirected edges, where each edge is a pair of vertices

functions: for all *graph* $\in$ *Graph*, *v*, v_1 and $v_2 \in$ *Vertices*

Graph Create()::=return an empty graph

Graph InsertVertex(*graph*, *v*)::= return a graph with *v* inserted. *v* has no incident edge.

Graph InsertEdge(*graph*, v_1,v_2)::= return a graph with new edge between v_1 and v_2

Graph DeleteVertex(*graph*, *v*)::= return a graph in which *v* and all edges incident to it are removed

Graph DeleteEdge(*graph*, v_1, v_2)::=return a graph in which the edge (v_1, v_2) is removed

Boolean IsEmpty(*graph*)::= if (*graph*==*empty graph*) return TRUE else return FALSE

List Adjacent(*graph*,*v*)::= return a list of all vertices that are adjacent to *v*

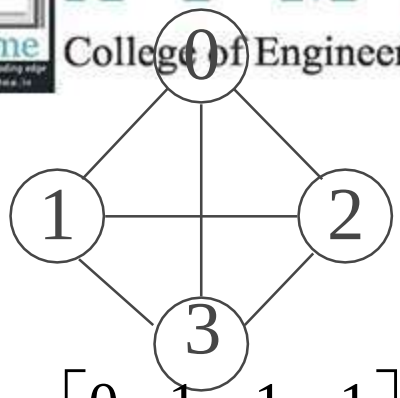
Graph Representations

- Adjacency Matrix
- Adjacency Lists

Adjacency Matrix

- Let G=(V,E) be a graph with n vertices.
- The adjacency matrix of G is a two-dimensional n by n array, say adj_mat
- If the edge (v_i, v_j) is in E(G), adj_mat[i][j]=1
- If there is no such edge in E(G), adj_mat[i][j]=0
- The adjacency matrix for an undirected graph is symmetric; the adjacency matrix for a digraph need not be symmetric

Examples for Adjacency Matrix



$$\begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix}$$

G_1

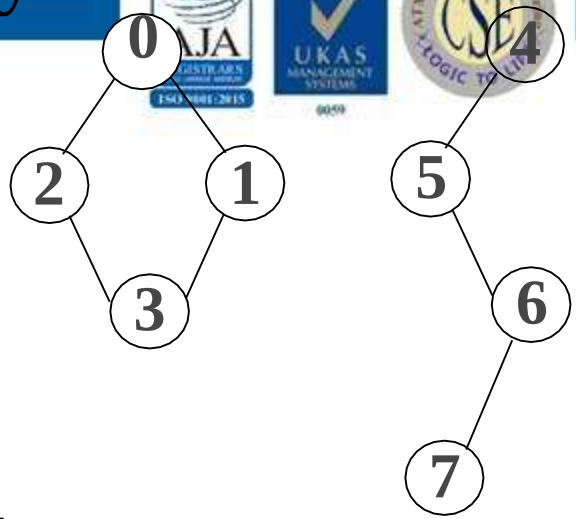
$$\begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix}$$

G_2

symmetric

undirected: $n^2/2$

directed: n^2



$$\begin{bmatrix} 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

G_4

Merits of Adjacency Matrix

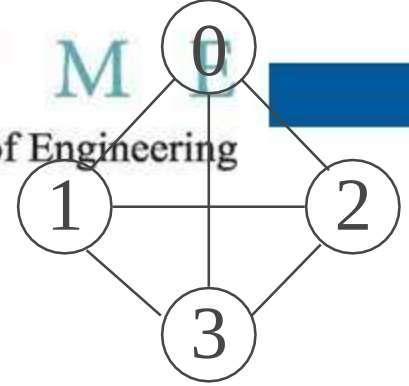
- From the adjacency matrix, to determine the connection of vertices is easy
- The degree of a vertex is $\sum_{j=0}^{n-1} adj\_mat[i][j]$
- For a digraph, the row sum is the out_degree, while the column sum is the in_degree

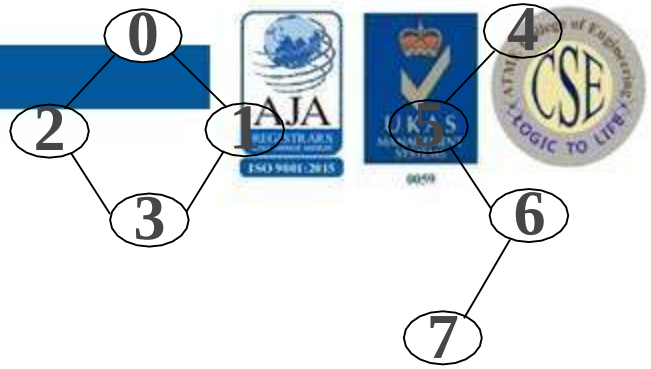
$$ind(vi) = \sum_{j=0}^{n-1} A[j,i] \quad outd(vi) = \sum_{j=0}^{n-1} A[i,j]$$

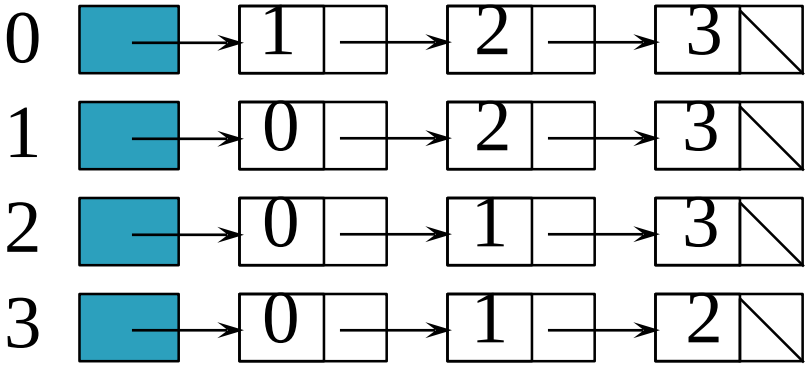
Data Structures for Adjacency Lists

Each row in adjacency matrix is represented as an adjacency list.

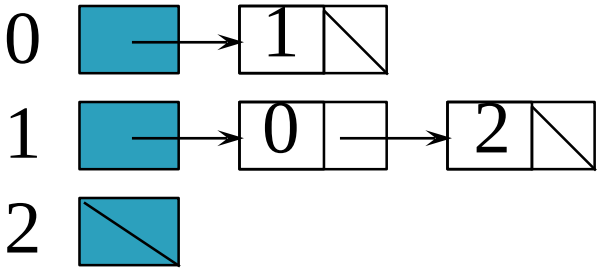
```
#define MAX_VERTICES 50
typedef struct node *node_pointer;
typedef struct node {
    int vertex;
    struct node *link;
};
node_pointer graph[MAX_VERTICES];
int n=0; /* vertices currently in use *
```



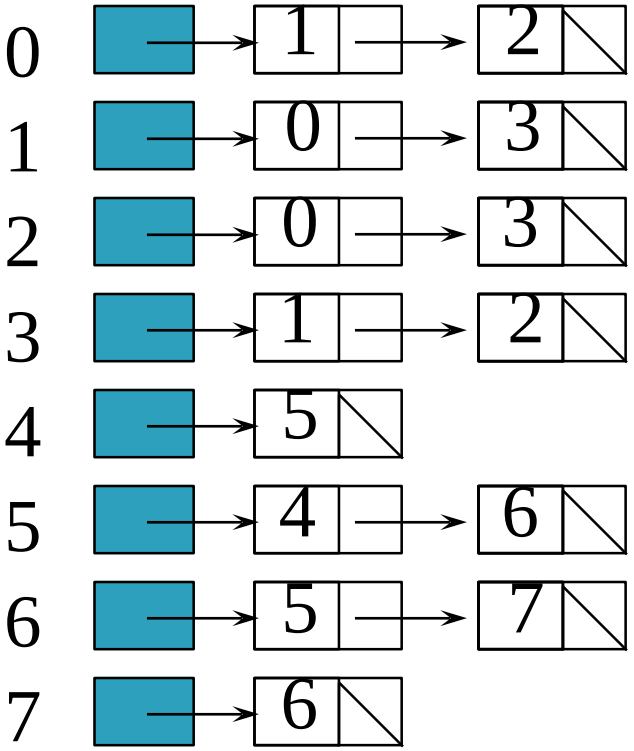




G_1



G_3



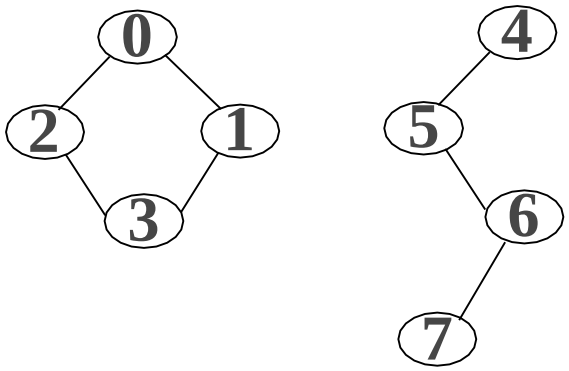
G_4

An undirected graph with e edges ==> e*2 list nodes and 2e list nodes

Interesting Operations

- degree of a vertex in an undirected graph
  - –# of nodes in adjacency list
- # of edges in a graph
  - –determined in O(n+e)
- out-degree of a vertex in a directed graph
  - –# of nodes in its adjacency list
- in-degree of a vertex in a directed graph
  - –traverse the whole data structure

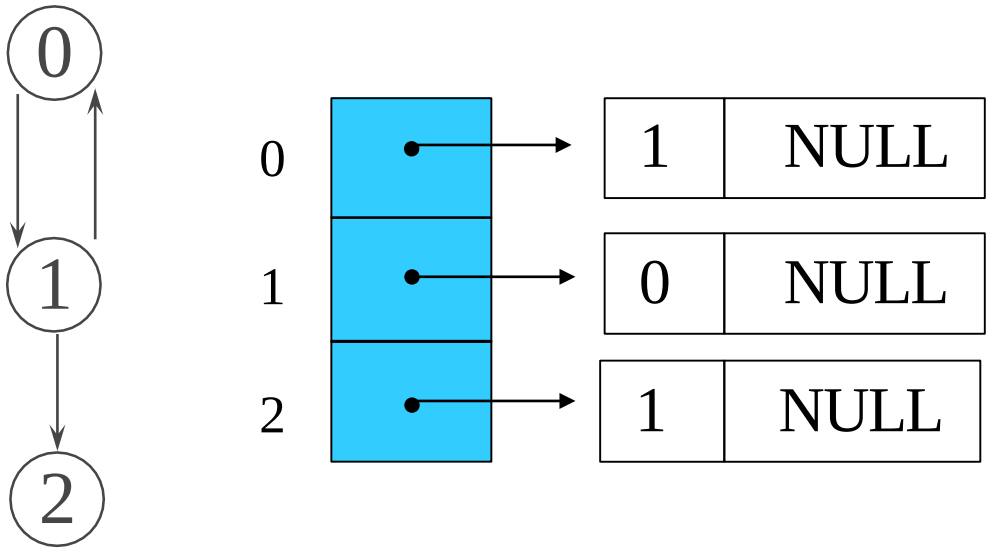
Compact Representation



node[0] … node[n-1]: starting point for vertices
node[n]: n+2e+1
node[n+1] … node[n+2e]: head node of edge

[0]	9		[8]	23		[16]	2	
[1]	11	0	[9]	1	4	[17]	5	
[2]	13		[10]	2	5	[18]	4	
[3]	15	1	[11]	0		[19]	6	
[4]	17		[12]	3	6	[20]	5	
[5]	18	2	[13]	0		[21]	7	
[6]	20		[14]	3	7	[22]	6	
[7]	22	3	[15]	1				

Figure 6.10: Inverse adjacency list for G_3



Determine in-degree of a vertex in a fast way.

Figure 6.11: Alternate node structure for adjacency lists (p.267)

tail	head	column link for head	row link for tail

Figure 6.12: Orthogonal representation for graph G3 (p.268)

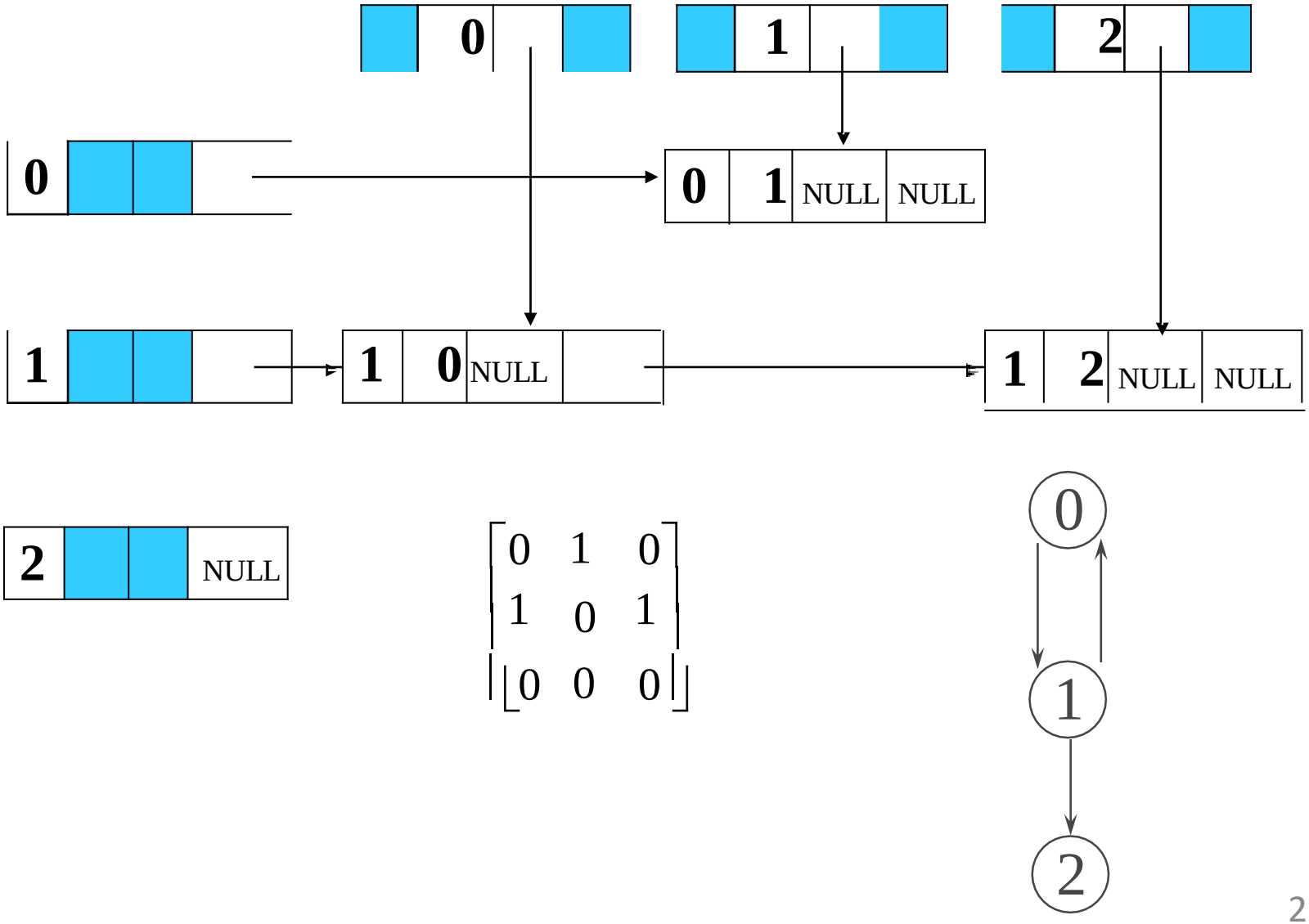
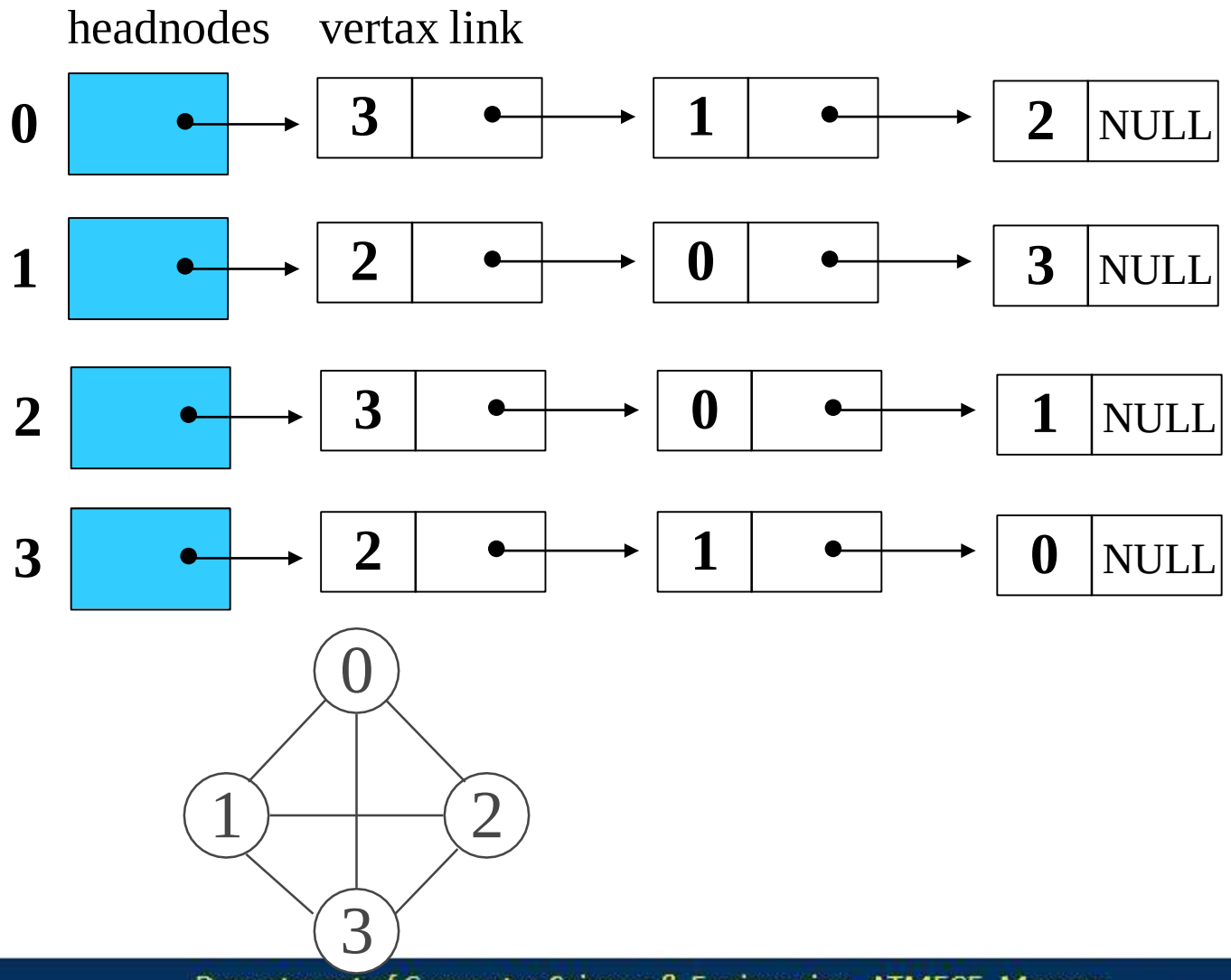


Figure 6.13:Alternate order adjacency list for G_1 (p.268)

Order is of no significance.



Some Graph Operations

- Traversal

  Given G=(V,E) and vertex v, find all w∈V, such that w connects v.

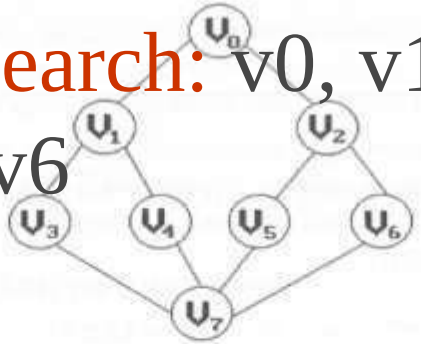
  – Depth First Search (DFS)
  preorder tree traversal

  – Breadth First Search (BFS)
  level order tree traversal

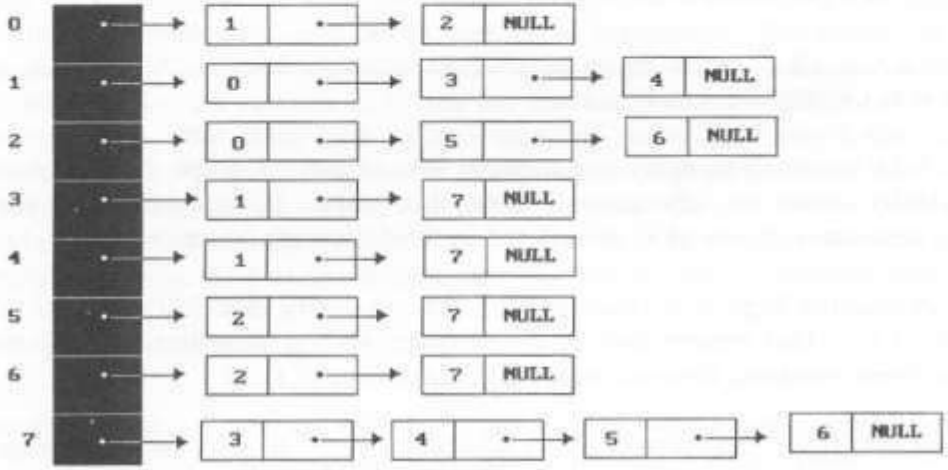
- Connected Components
- Spanning Trees

*Figure 6.19: Graph *G* and its adjacency lists (p.274)

depth first search: v0, v1, v3, v7, v4, v5, v2, v6



(a)



(b)

breadth first search: v0, v1, v2, v3, v4, v5, v6, v7

Depth First Search

```
#define FALSE 0
#define TRUE 1
short int visited[MAX_VERTICES];
```

```
void dfs(int v)
{
  node_pointer w;
  visited[v]= TRUE;
  printf("%5d", v);
  for (w=graph[v]; w; w=w->link)
    if (!visited[w->vertex])
      dfs(w->vertex);
}
```

Data structure
adjacency list: O(e)
adjacency matrix: $O(n^2)$

Breadth First Search

```
typedef struct queue *queue_pointer;
typedef struct queue {
    int vertex;
    queue_pointer link;
};
void addq(queue_pointer *,
          queue_pointer *, int);
int deleteq(queue_pointer *);
```

Breadth First Search (*Continued*)

```
void bfs(int v)
{
   node_pointer w;
   queue_pointer front, rear;
   front = rear = NULL;
   printf("%5d", v);
   visited[v] = TRUE;
   addq(&front, &rear, v);
```

adjacency list: O(e)
adjacency matrix: $O(n^2)$

```
    while (front) {
       v= deleteq(&front);
       for (w=graph[v]; w; w=w->link)
          if (!visited[w->vertex]) {
             printf("%5d", w->vertex);
             addq(&front, &rear, w->vertex);
             visited[w->vertex] = TRUE;
          }
    }
}
```

Connected Components

```
void connected(void)
{
    for (i=0; i<n; i++) {
        if (!visited[i]) {
            dfs(i);
            printf("\n");
        }
    }
}
```

adjacency list: O(n+e)
adjacency matrix: $O(n^2)$

Searching and Sorting

Topics

- Sequential Search on an Unordered File
- Sequential Search on an Ordered File
- Binary Search
- Bubble Sort
- Insertion Sort

Common Problems

- There are some very common problems that we use computers to solve:
  - **Searching** through a lot of records for a specific record or set of records
  - Placing records in order, which we call **sorting**
- There are numerous algorithms to perform searches and sorts. We will briefly explore a few common ones.

Searching

- A question you should always ask when selecting a search algorithm is “How fast does the search have to be?” The reason is that, in general, the faster the algorithm is, the more complex it is.
- Bottom line: you don’t always need to use or should use the fastest algorithm.
- Let’s explore the following search algorithms, keeping speed in mind.
  - Sequential (linear) search
  - Binary search

Sequential Search on an Unordered File

- Basic algorithm:

  Get the search criterion (**key**)
  Get the first record from the file
  While ((record != key) and (still more records))
    Get the next record
  End_while

- When do we know that there wasn't a record in the file that matched the key?

Sequential Search on an Ordered File

- Basic algorithm:

  Get the search criterion (key)
  Get the first record from the file
  While ((record < key) and (still more records))
    Get the next record
  End_while
  If (record = key)
    Then success
    Else there is no match in the file
  End_else

- When do we know that there wasn't a record in the file that matched the key?

Sequential Search of Ordered vs. Unordered List

- Let's do a comparison.
- If the order was ascending alphabetical on customer's last names, how would the search for John Adams on the ordered list compare with the search on the unordered list?
  - Unordered list
    - if John Adams was in the list?
    - if John Adams was not in the list?
  - Ordered list
    - if John Adams was in the list?
    - if John Adams was not in the list?

Ordered vs Unordered (cont.)

- How about George Washington?
  - Unordered
    - if George Washington was in the list?
    - If George Washington was not in the list?
  - Ordered
    - if George Washington was in the list?
    - If George Washington was not in the list?
- How about James Madison?

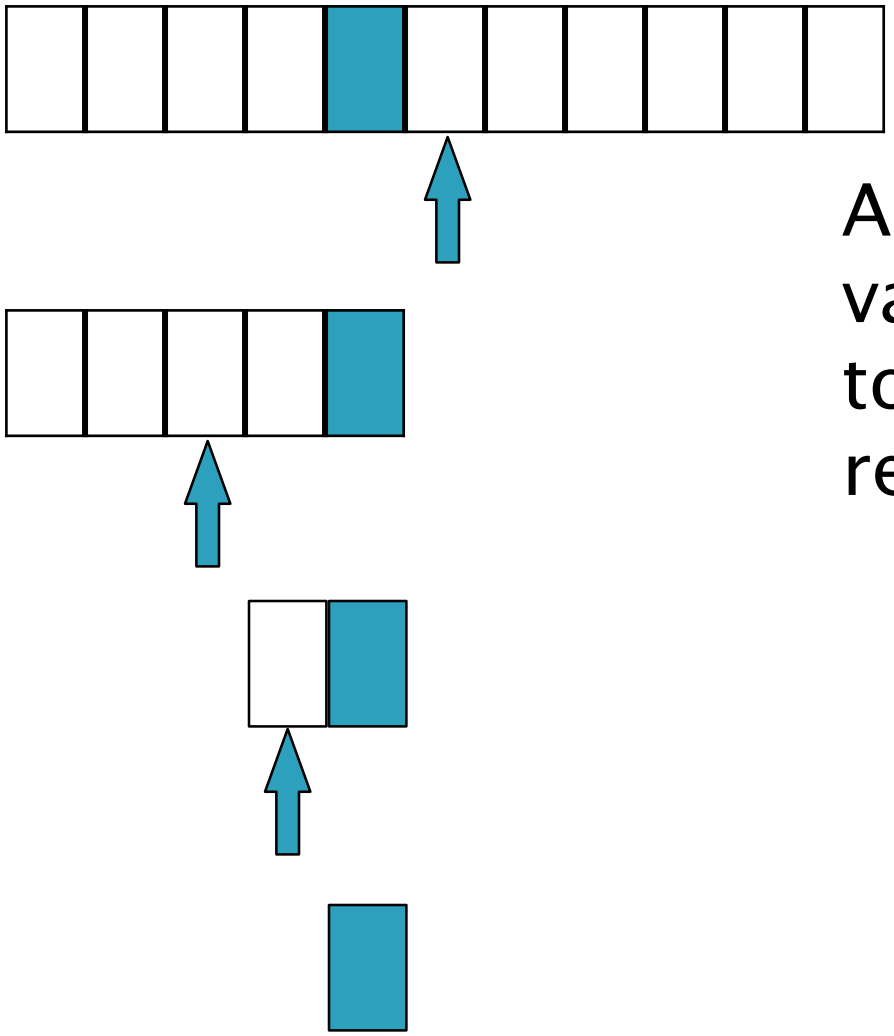
Ordered vs. Unordered (cont.)

- Observation: the search is faster on an ordered list only when the item being searched for is not in the list.
- Also, keep in mind that the list has to first be placed in order for the ordered search.
- Conclusion: the **efficiency** of these algorithms is roughly the same.
- So, if we need a faster search, we need a completely different algorithm.
- How else could we search an ordered file?

Binary Search

- If we have an ordered list and we know how many things are in the list (i.e., number of records in a file), we can use a different strategy.
- The **binary search** gets its name because the algorithm continually divides the list into two parts.

How a Binary Search Works



Always look at the center value. Each time you get to discard half of the remaining list.

Is this fast ?

How Fast is a Binary Search?

- Worst case: 11 items in the list took 4 tries
  - 1st try - list has 16 items
  - 2nd try - list has 8 items
  - 3rd try - list has 4 items
  - 4th try - list has 2 items
  - 5th try - list has 1 item
- How about the worst case for a list with 32 items ?

How Fast is a Binary Search? (con't)

List has 250 items

1st try – 125 items
2nd try – 63 items
3rd try – 32 items
4th try – 16 items
5th try – 8 items
6th try – 4 items
7th try – 2 items
8th try – 1 item

List has 512 items

1st try – 256 items
2nd try – 128 items
3rd try – 64 items
4th try – 32 items
5th try – 16 items
6th try – 8 items
7th try – 4 items
8th try – 2 items
9th try – 1 item

What's the Pattern?

- List of 11 took 4 tries
- List of 32 took 5 tries
- List of 250 took 8 tries
- List of 512 took 9 tries

- $32 = 2^5$ and $512 = 2^9$
- $8 < 11 < 16$ $\quad 2^3 < 11 < 2^4$
- $128 < 250 < 256$ $\quad 2^7 < 250 < 2^8$

A Very Fast Algorithm!

- How long (worst case) will it take to find an item in a list 30,000 items long?

$2^{10} = 1024$
$2^{11} = 2048$
$2^{12} = 4096$
$2^{13} = 8192$
$2^{14} = 16384$
$2^{15} = 32768$

- So, it will take only 15 tries!

Lg n Efficiency

- We say that the binary search algorithm runs in **$\log_2 n$ time.** (Also written as **lg n**)
- Lg n means the log to the base 2 of some value of n.
- $8 = 2^3$ lg 8 = 3 $16 = 2^4$ lg 16 = 4
- <u>There are no algorithms that run faster than lg n time</u>.

Sorting

- So, the binary search is a very fast search algorithm.
- But, the list has to be sorted before we can search it with binary search.
- To be really efficient, we also need a fast sort algorithm.

Common Sort Algorithms

Bubble Sort	Heap Sort
Selection Sort	Merge Sort
Insertion Sort	Quick Sort

- There are many known sorting algorithms. Bubble sort is the slowest, running in **n^2 time.** Quick sort is the fastest, running in **n lg n time.**
- As with searching, the faster the sorting algorithm, the more complex it tends to be.
- We will examine two sorting algorithms:
  - Bubble sort
  - Insertion sort

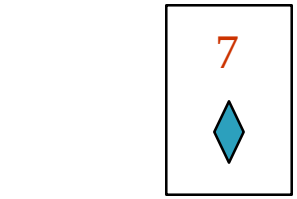
Bubble Sort Code

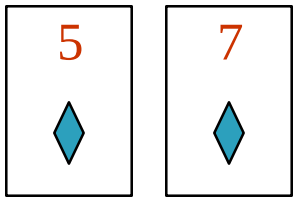
```
void bubbleSort (int a[ ] , int size)
{
    int i, j, temp;
    for ( i =  0; i <  size; i++ )    /* controls passes through the list */
    {
          for ( j =  0; j <  size -  1; j++ )   /* performs adjacent comparisons
    */
          {
                  if ( a[ j ] >  a[ j+1 ] )   /* determines if a swap should
    occur */
                  {
                          temp =  a[ j ];       /* swap is performed */
                          a[ j ] =  a[ j +  1 ];
                          a[ j+1 ] =  temp;
                  }
          }
    }
}
```

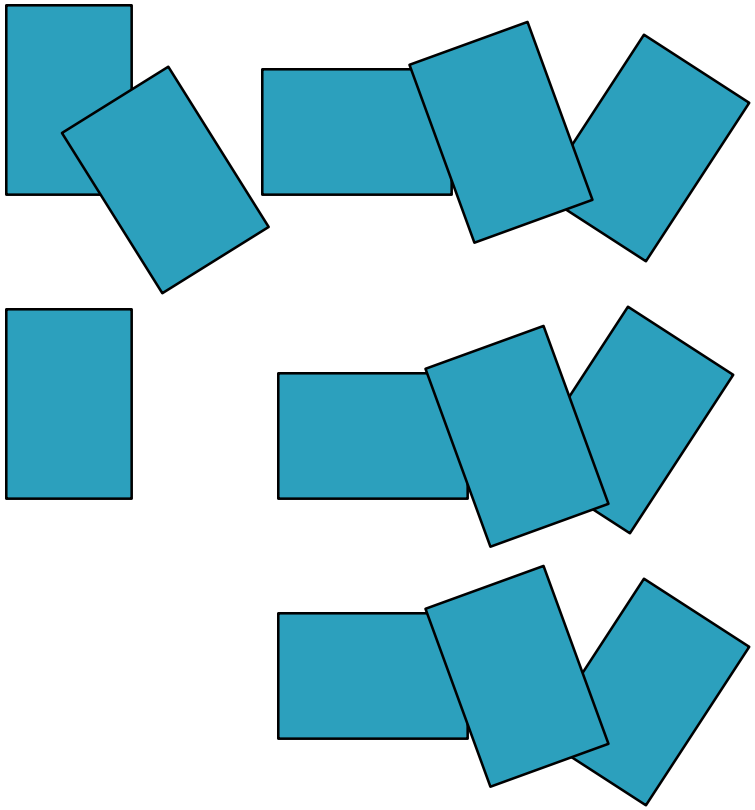
Insertion Sort

- Insertion sort is slower than quick sort, but not as slow as bubble sort, and it is easy to understand.
- Insertion sort works the same way as arranging your hand when playing cards.
  - Out of the pile of unsorted cards that were dealt to you, you pick up a card and place it in your hand in the correct position relative to the cards you're already holding.

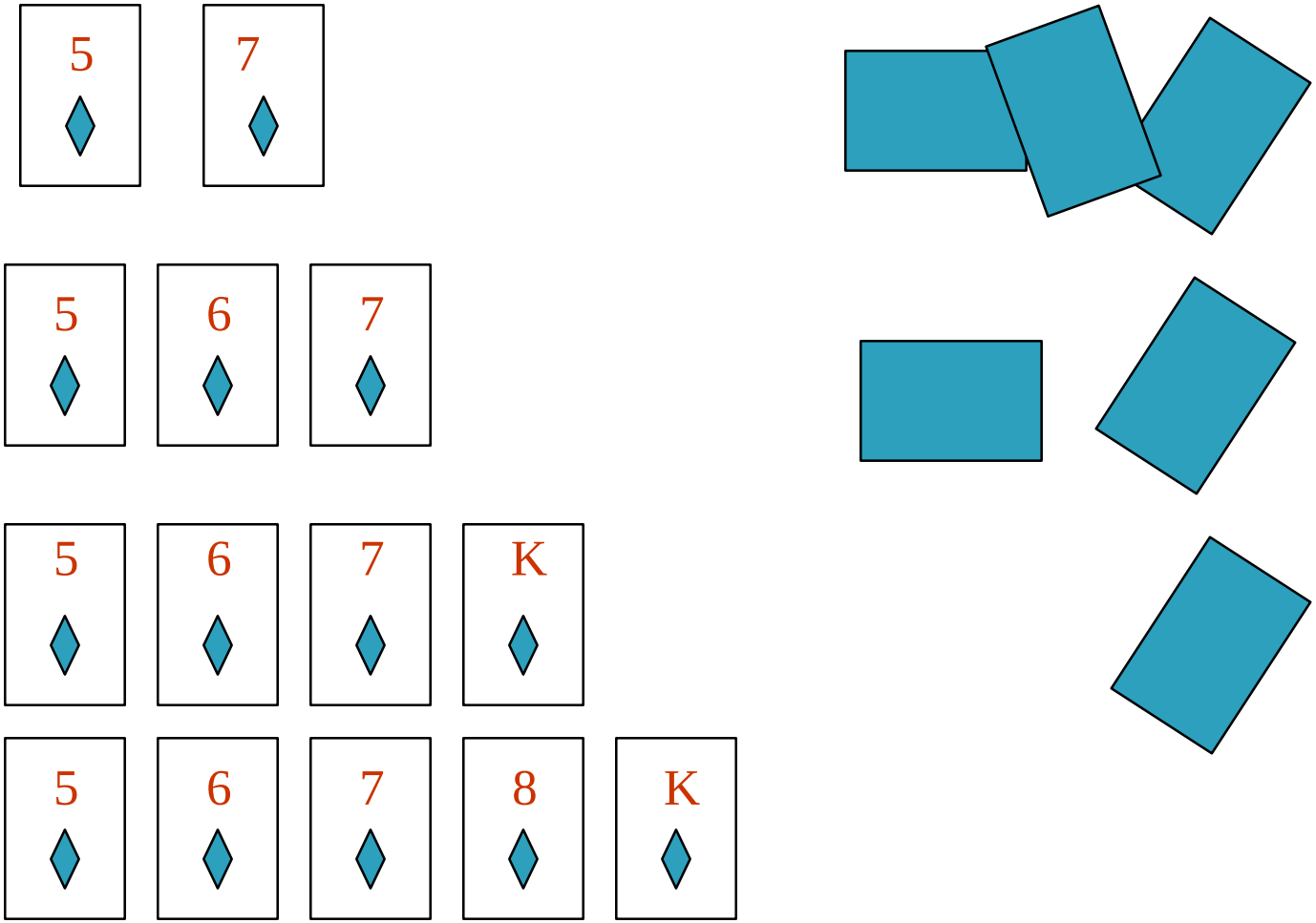
Arranging Your Hand



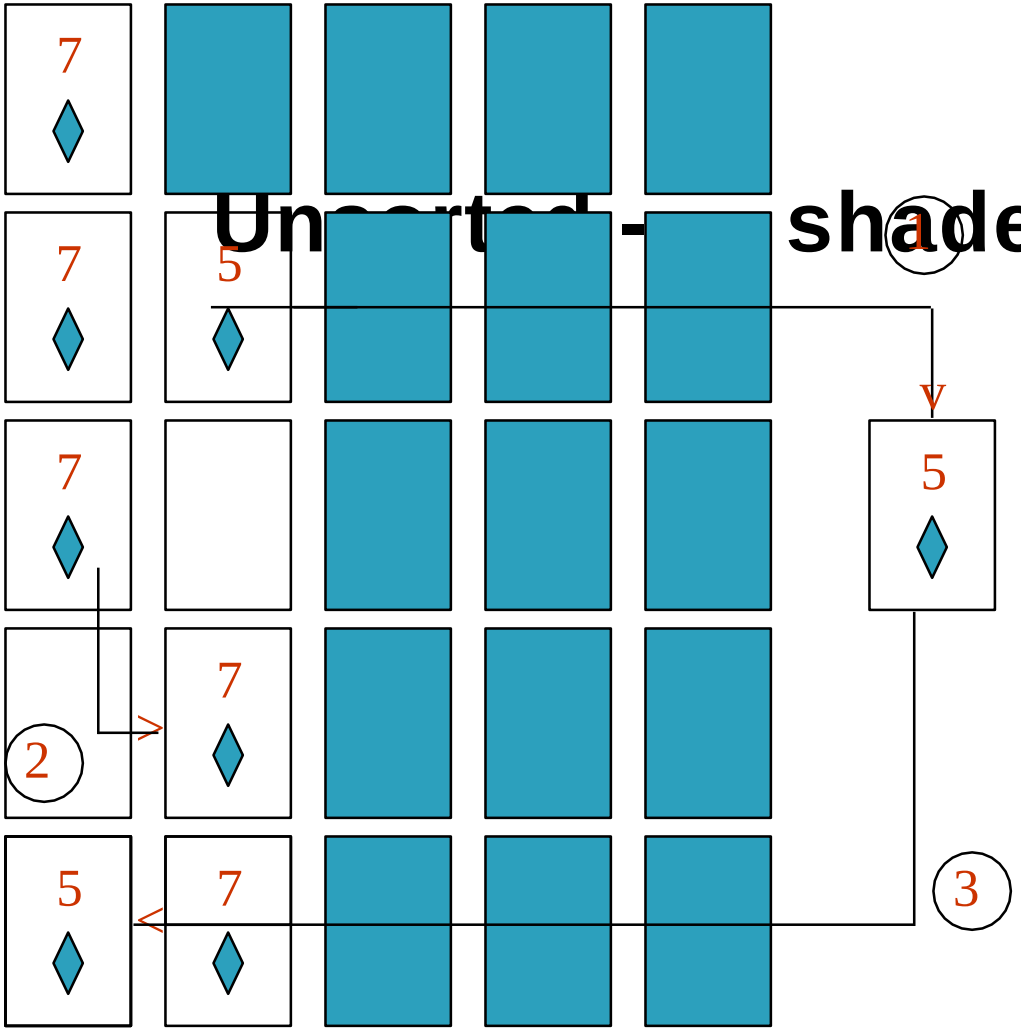




Arranging Your Hand



Insertion Sort



Look at 2nd item – 5.

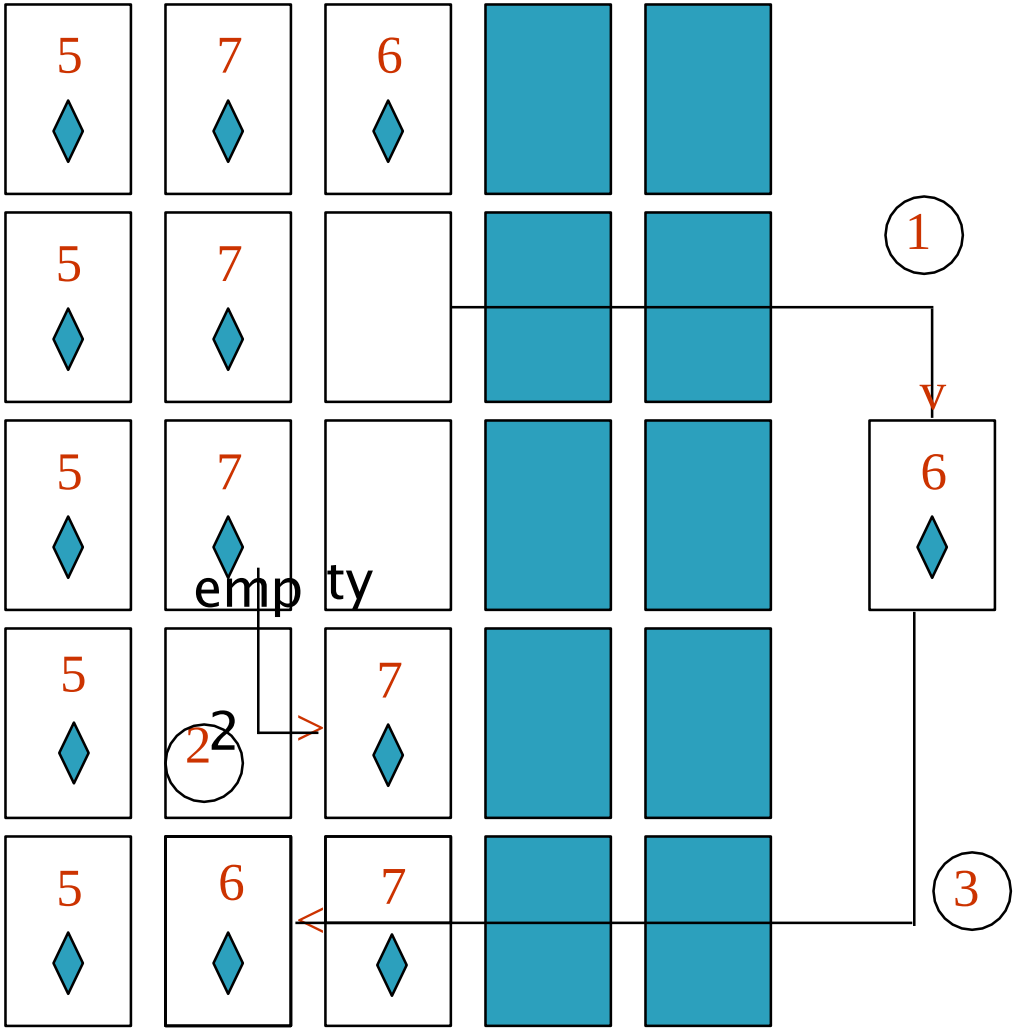
Compare 5 to 7.

5 is smaller, so move 5 to temp, leaving an empty slot in position 2.

Move 7 into the empty slot, leaving position 1 open.

Move 5 into the open position.

Insertion Sort (con't)



Look at next item – 6.

Compare to 1st – 5. 6 is larger, so leave 5.

Compare to next – 7. 6 is smaller, so move 6 to temp, leaving an empty slot.

Move 7 into the

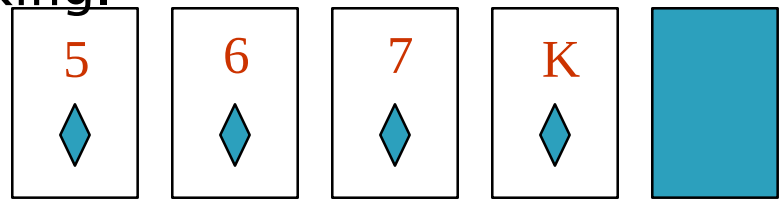
slot, leaving position

open.

Move 6 to the open 2nd position.

Insertion Sort (con't)

King.



it is.

6.

where it is.

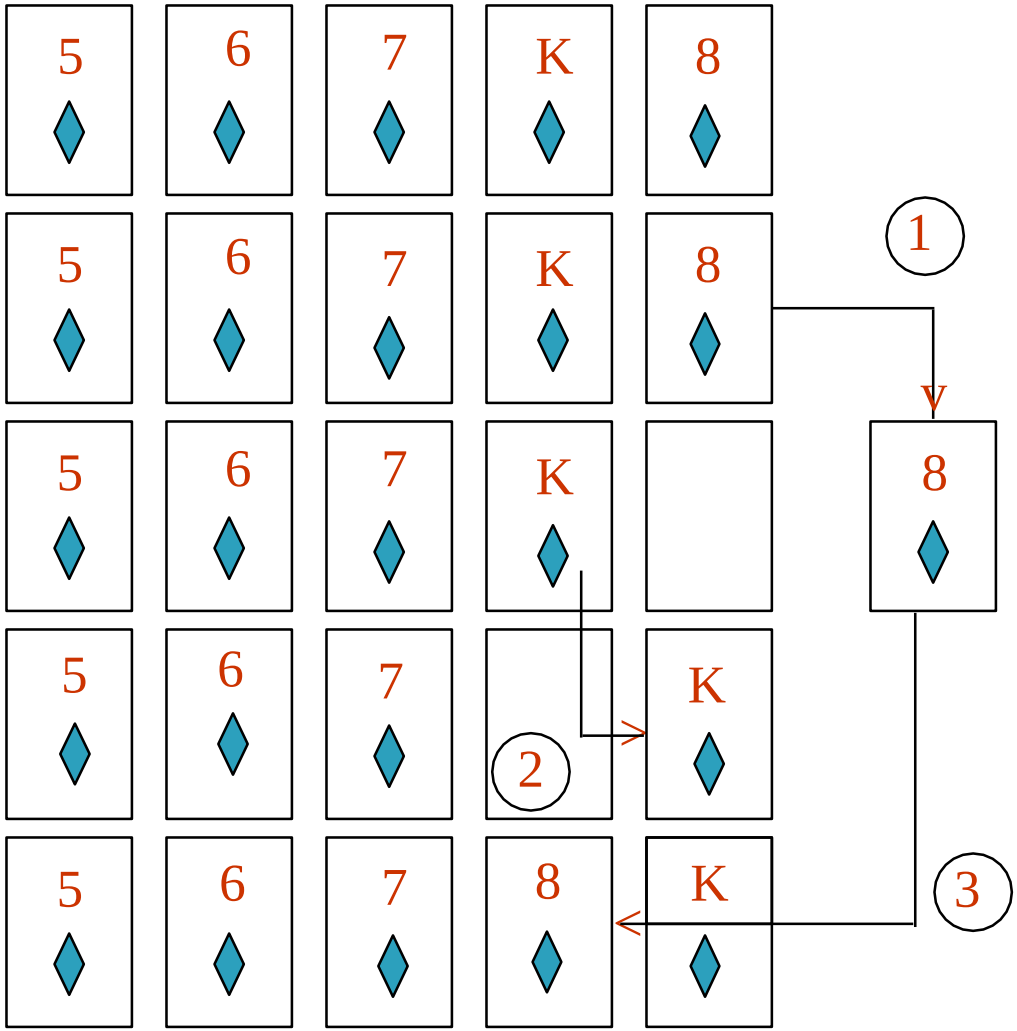
Look at next item –

Compare to 1st – 5.
King is larger, so leave 5 where

Compare to next –
King is larger, so leave 6

Compare to next – 7.
King is larger, so leave 7 where it is.

Insertion Sort (con't)



HASHING

Hashing Techniques:

Hashing:

- Hashing is a technique used to Performing Insertion, deletion & search operations in the constant average time by implementing Hash table Data Structure .
- It is used to Index and Retrieve Items in a Database.

Two Types of Hashing

1. **Static Hashing .**
2. **Dynamic Hashing** .

Static Hashing :

- It is the hash function maps search key value to a fixed set of locations.

Dynamic Hashing :

- The Hash Table can grow to handle more Items.The associated Hash Function must Change as the table grows.

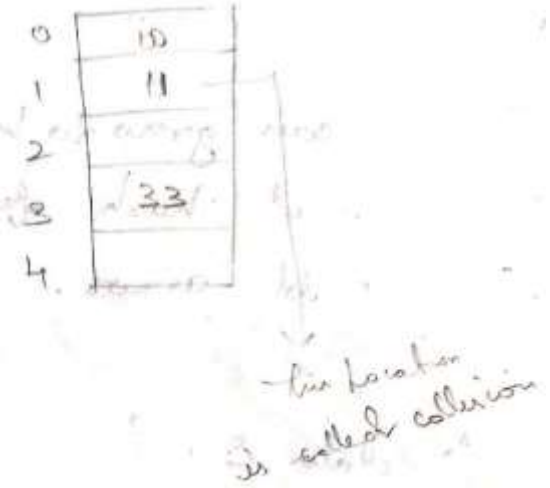
Hash Table :

- The Hash Table data structure is a array of some fixed size table containing the Keys.
- A Key is a values associated with each record.
- A Hash table is partition into array of size.
- Each Bucket has many slots and each slots holds one records.

Hash Function :

- A Hashing Function is a key to address Transformation which acts upon a given Key to complete the Relative Position of the Key in a array

- A Key can be a member of a String etc..
- A Hash Function Formula is
- **Hash (Key Value) =(Key Values % Table Size)**



- Hash (Key Value)=Key Values % Table Size
- Hash(10) =10 % 5=0 Hash(21)= 21 % 5=1
- Hash(33) =33 % 5 =3
- Hash(11)=11 % 5=1

A Good Hashing consist of

- Minimum Collision.
- Be easy and quick to complete.
- Distribute Key Value Every in the Hash Table.
- Use all the Information Provided in the Key.

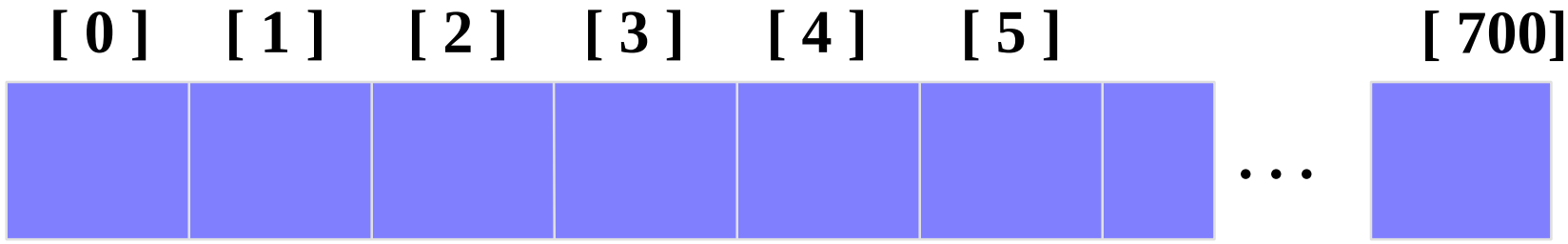
Application of Hash Table:

- Database Systems
- Symbol Tables .
- Data Dictionaries
- Network Processing Algorithm
- Browse Casher.

What is a Hash Table ?

The simplest kind of hash table is an array of records.

This example has 701 records.

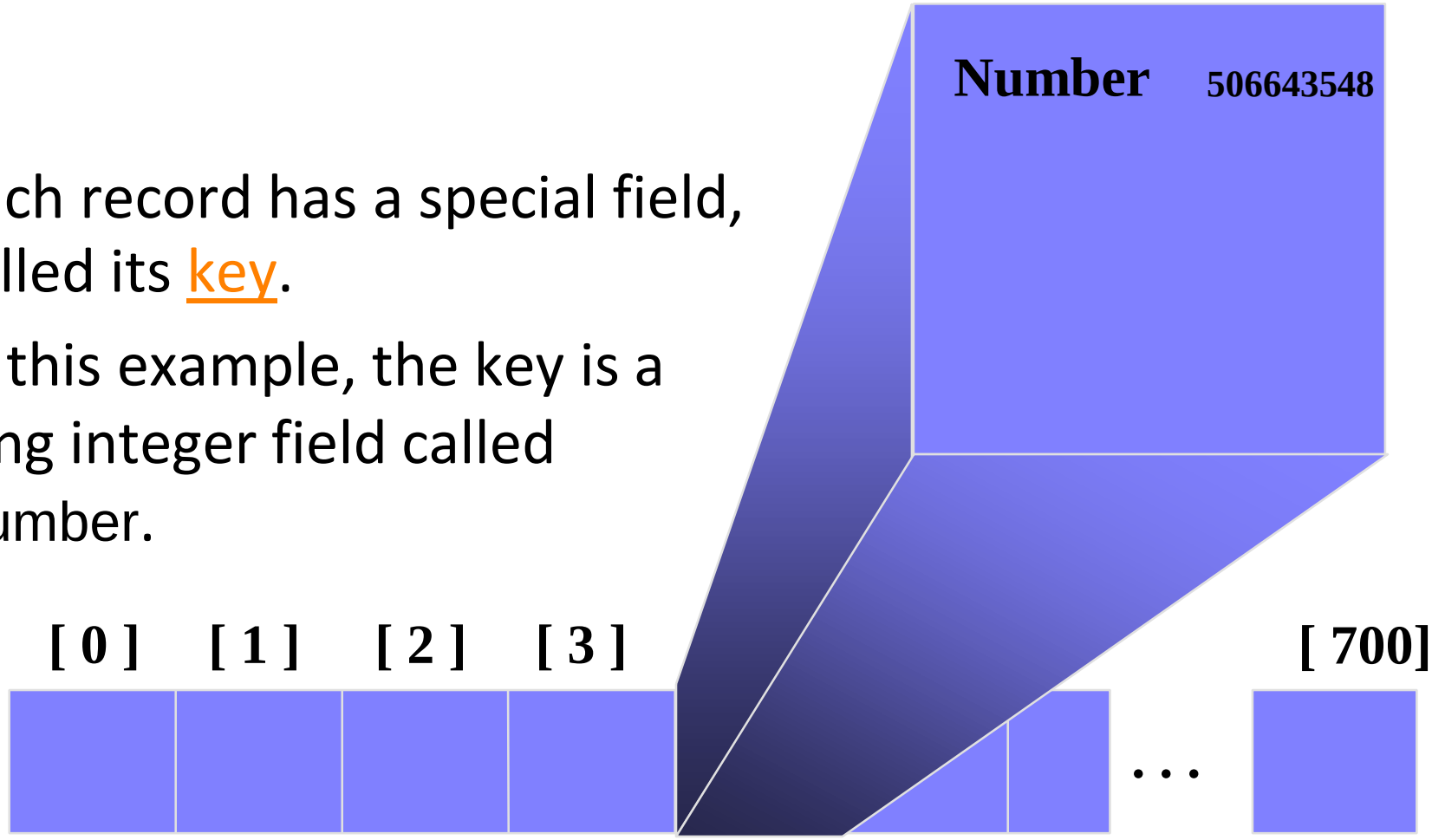


An array of records

What is a Hash Table ?

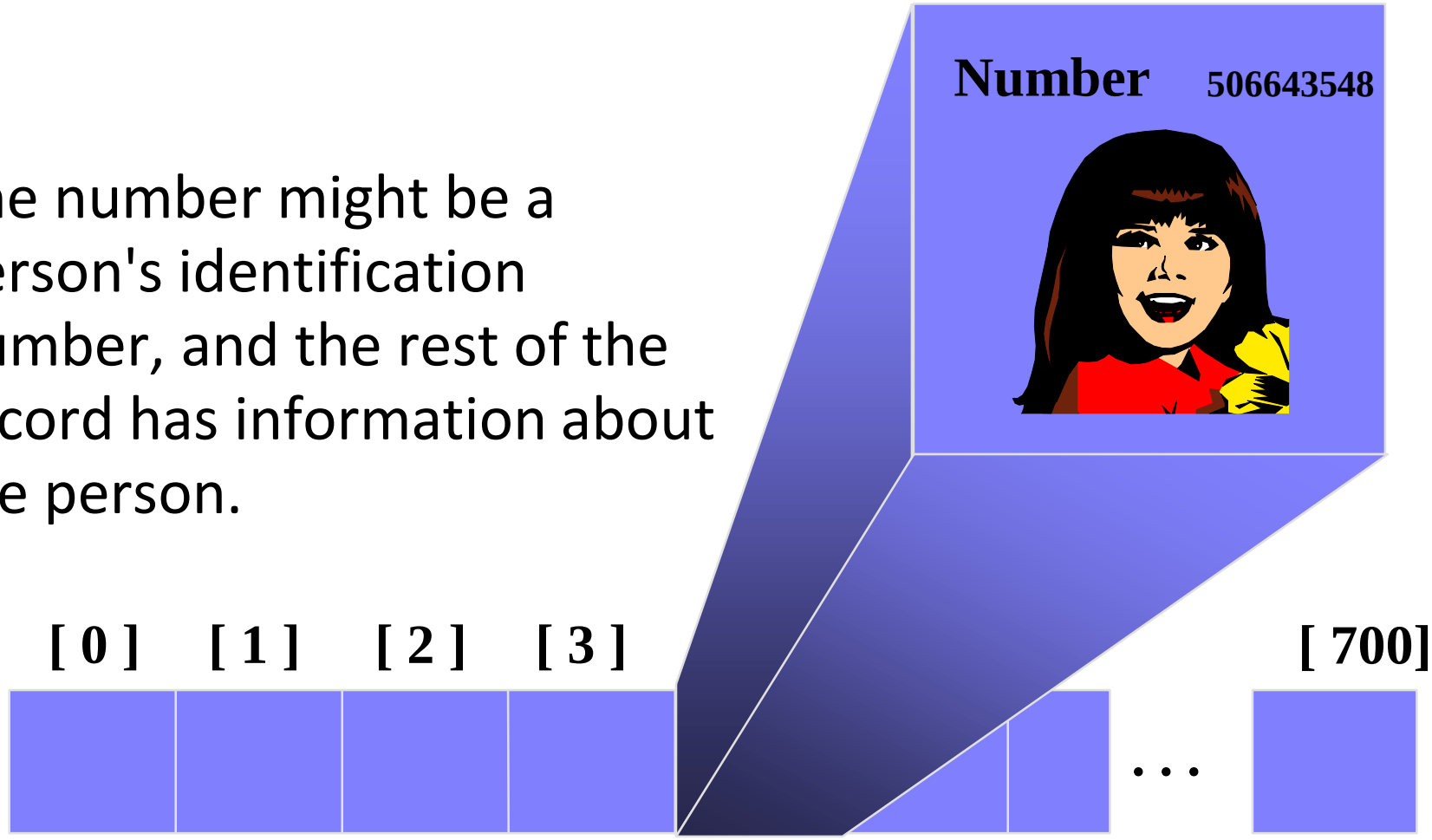
Each record has a special field, called its key.

In this example, the key is a long integer field called Number.



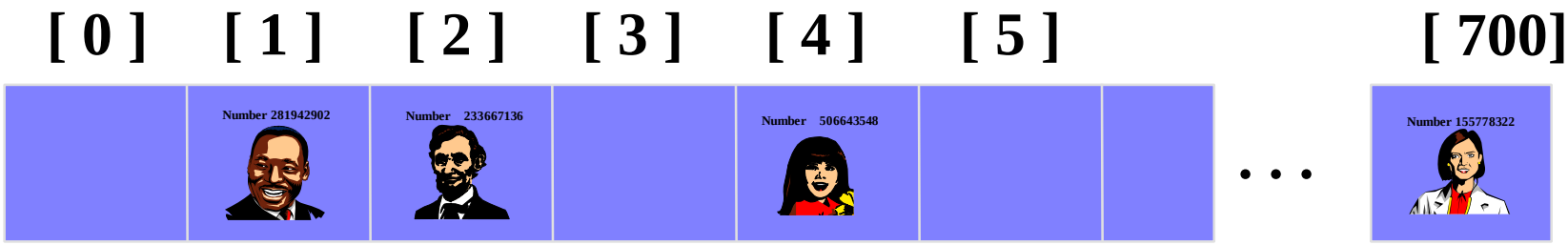
What is a Hash Table ?

The number might be a person's identification number, and the rest of the record has information about the person.



What is a Hash Table ?

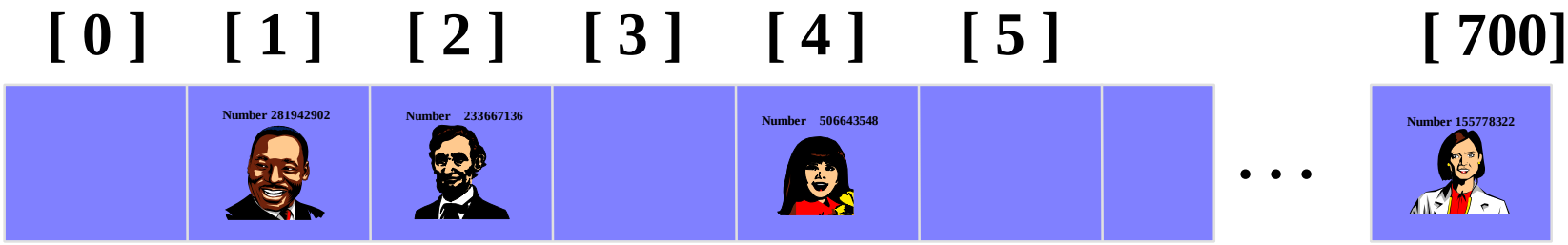
When a hash table is in use, some spots contain valid records, and other spots are "empty".



Inserting a New Record

In order to insert a new record, the **key** must somehow be **converted to** an array **index**.

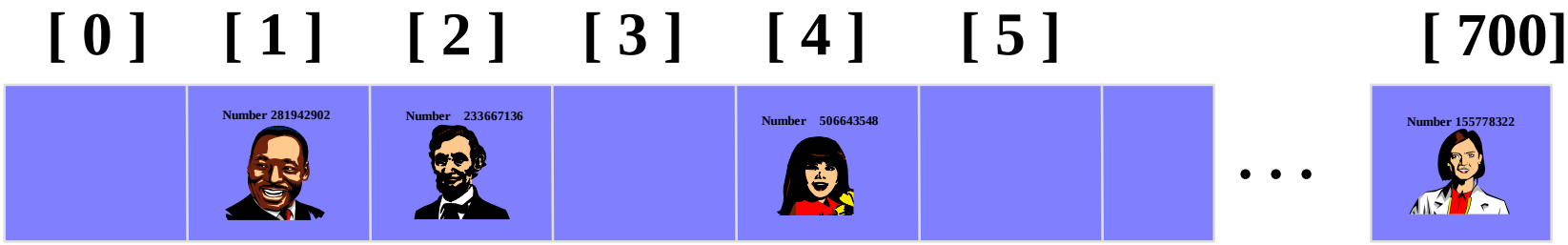
The index is called the **hash value** of the key.



Inserting a New Record

Typical way create a hash value:

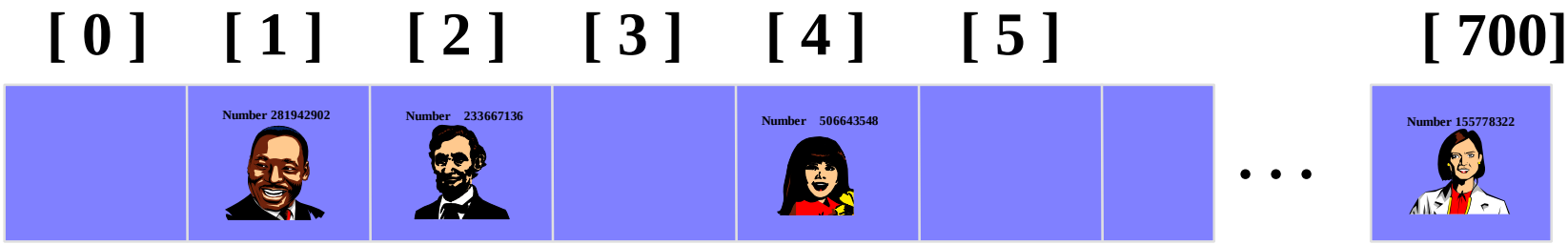
What is (580625685 mod 701) ?



Inserting a New Record

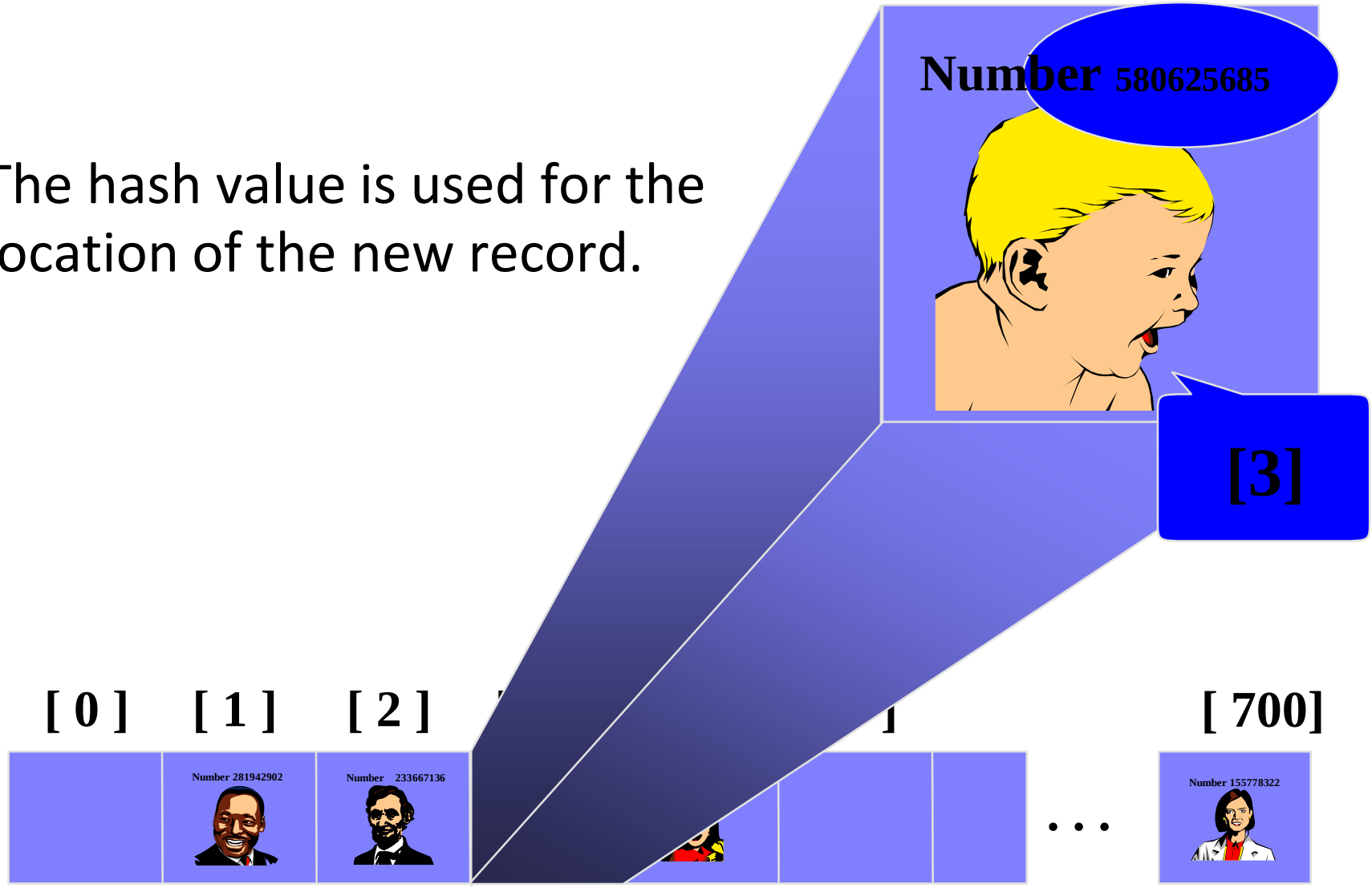
Typical way to create a hash value:

What is (580625685 mod 701) ?



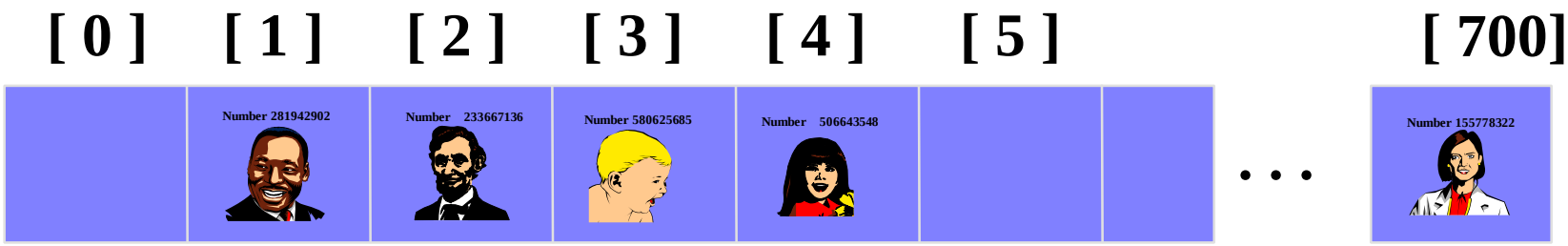
Inserting a New Record

The hash value is used for the location of the new record.



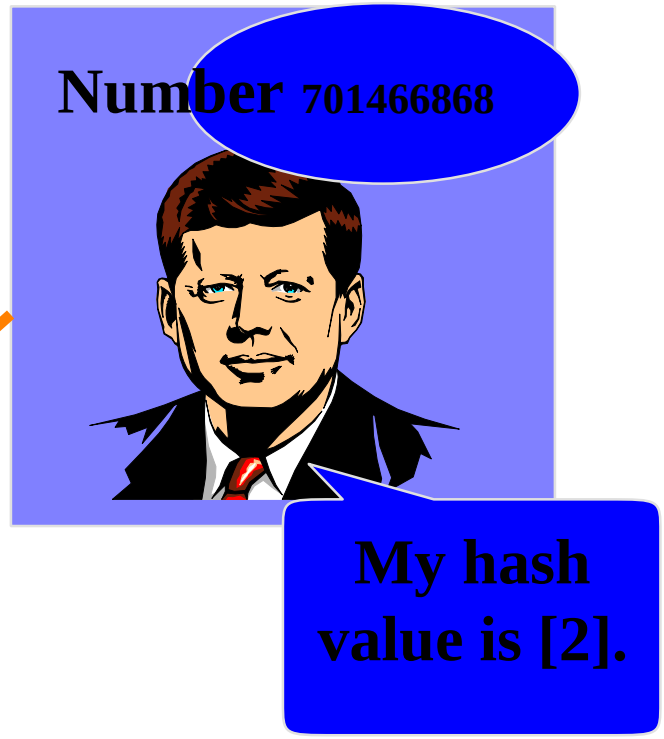
Inserting a New Record

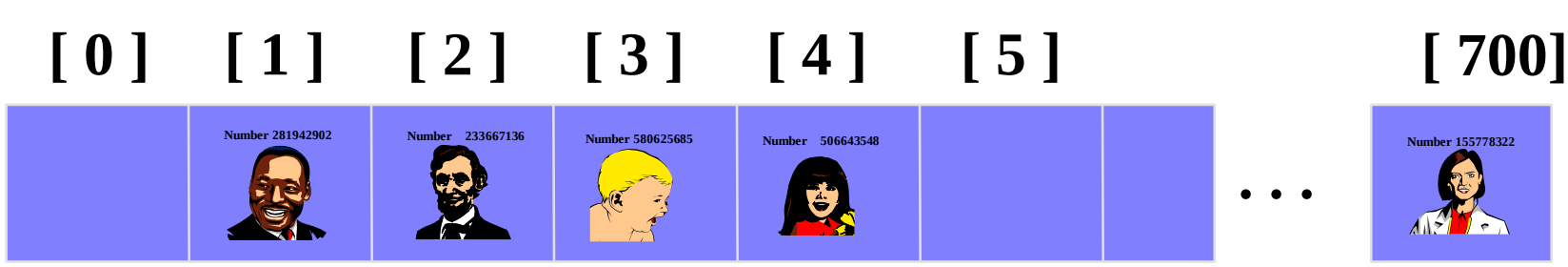
The hash value is used for the location of the new record.



Collisions

Here is another new record to insert, with a hash value of 2.

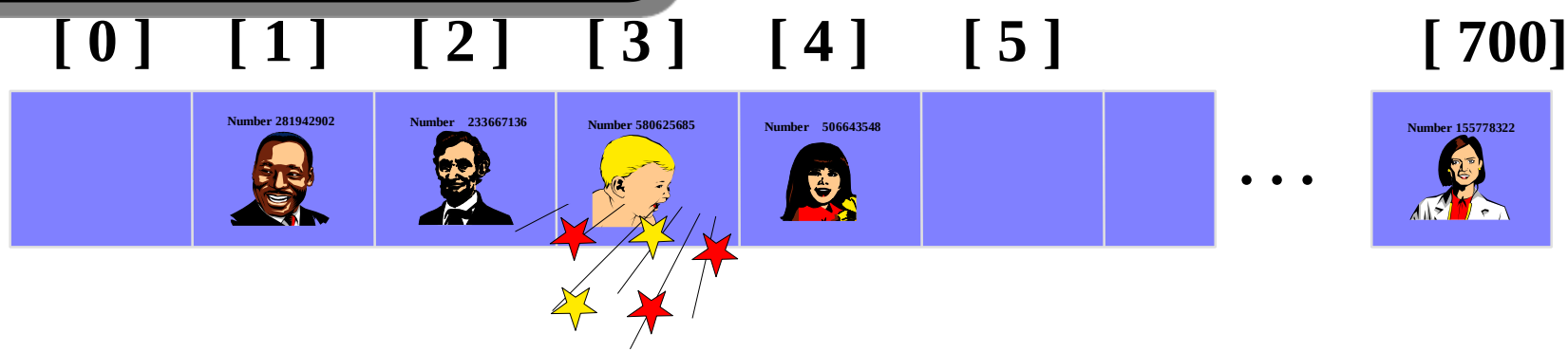




Collisions

This is called a collision, because there is already another valid record at [2].

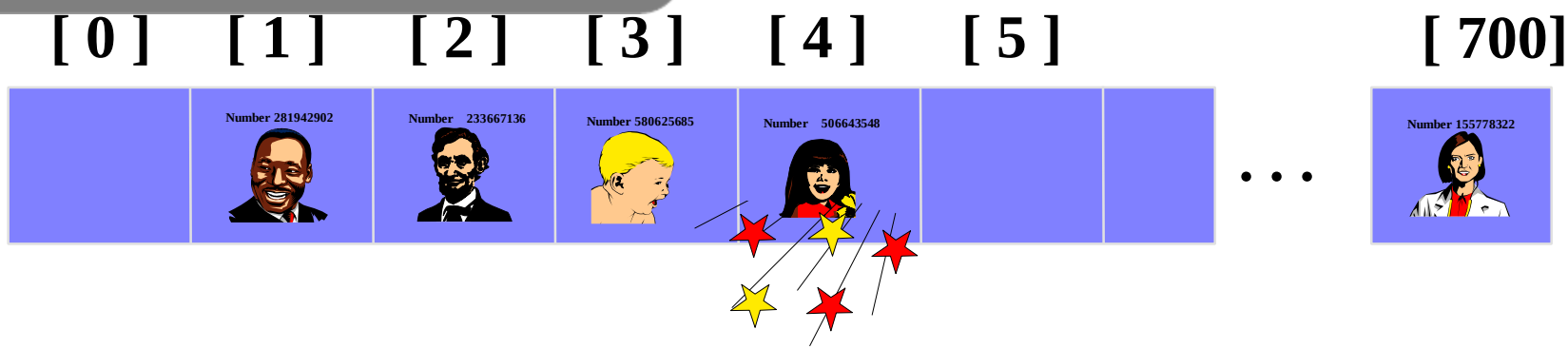
When a collision occurs, move forward until you find an empty spot.



Collisions

This is called a **collision**, because there is already another valid record at [2].

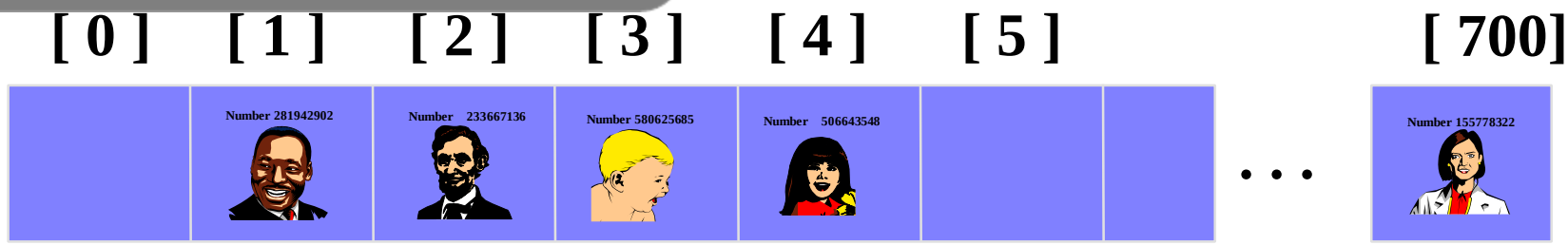
When a collision occurs, move forward until you find an empty spot.



Collisions

This is called a collision, because there is already another valid record at [2].

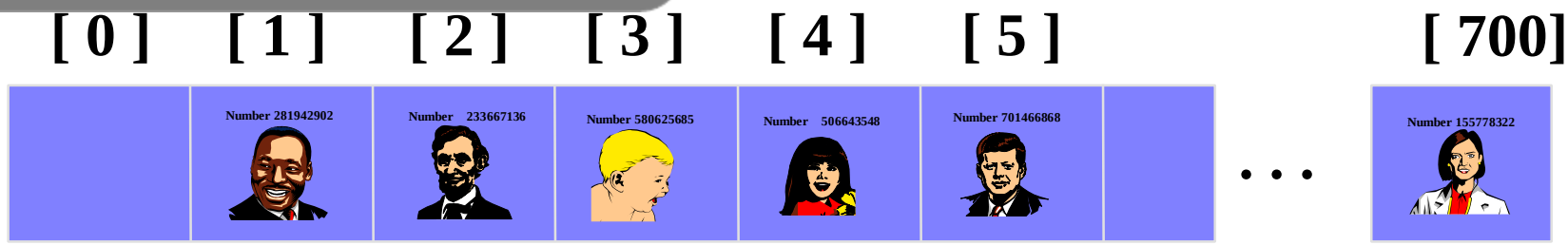
When a collision occurs, move forward until you find an empty spot.



Collisions

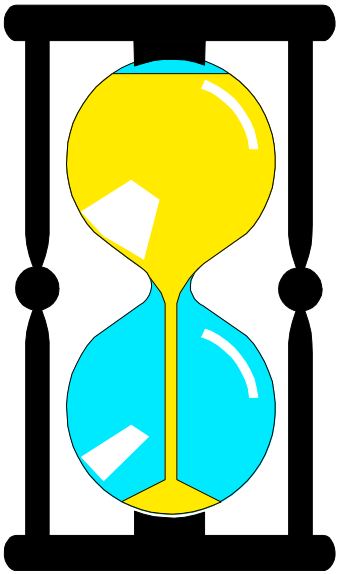
This is called a **collision**, because there is already another valid record at [2].

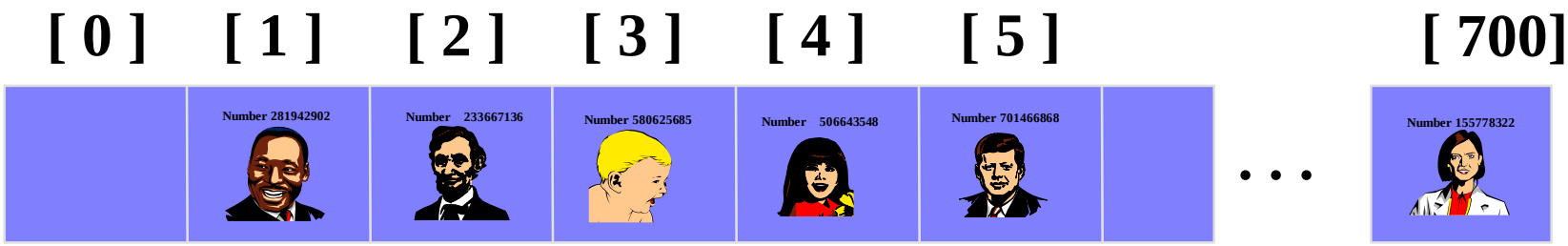
The new record goes in the empty spot.



A Quiz

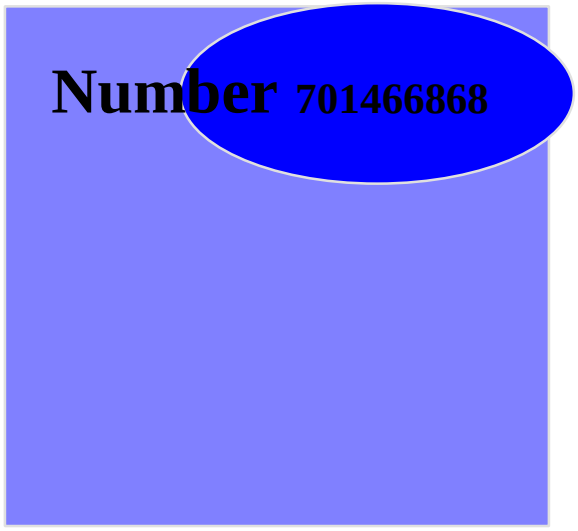
Where would you be placed in this table, if there is no collision? Use your social security number or some other favorite number.

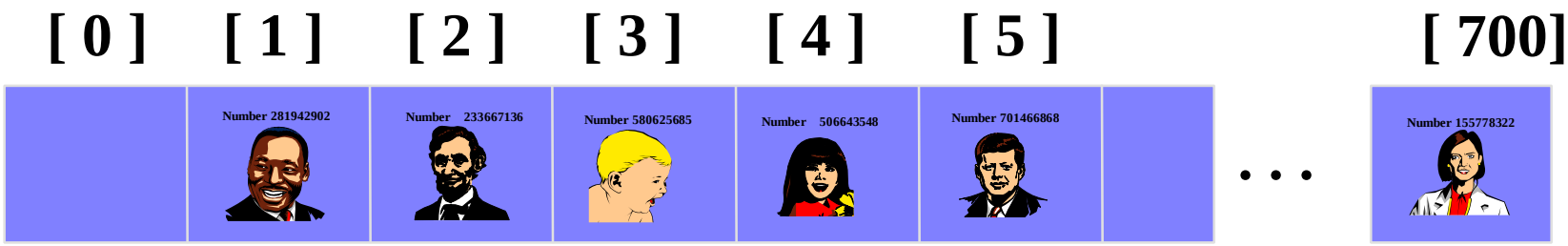




Searching for a Key

The data that's attached to a key can be found fairly quickly.

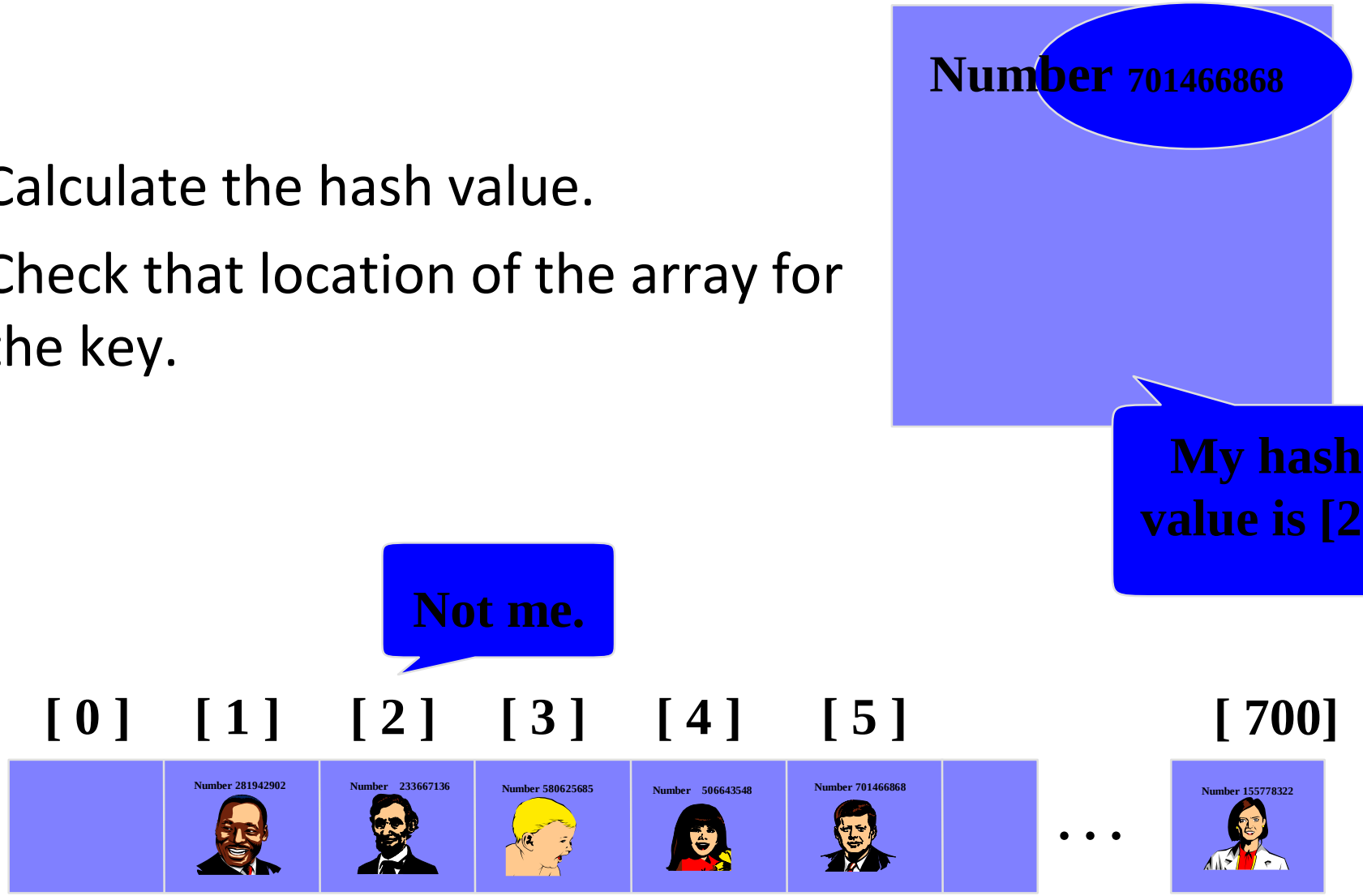




Searching for a Key

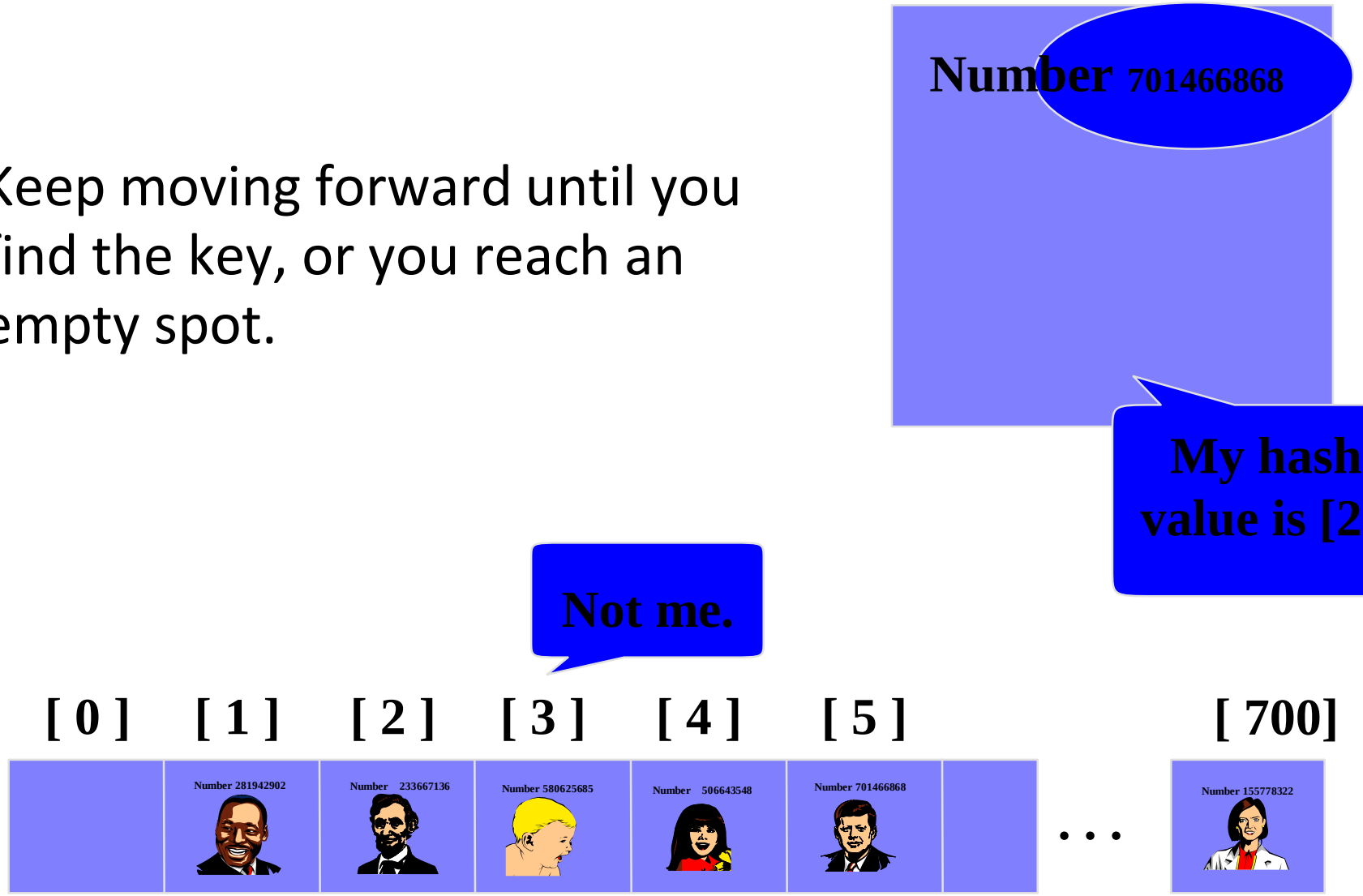
Calculate the hash value.

Check that location of the array for the key.



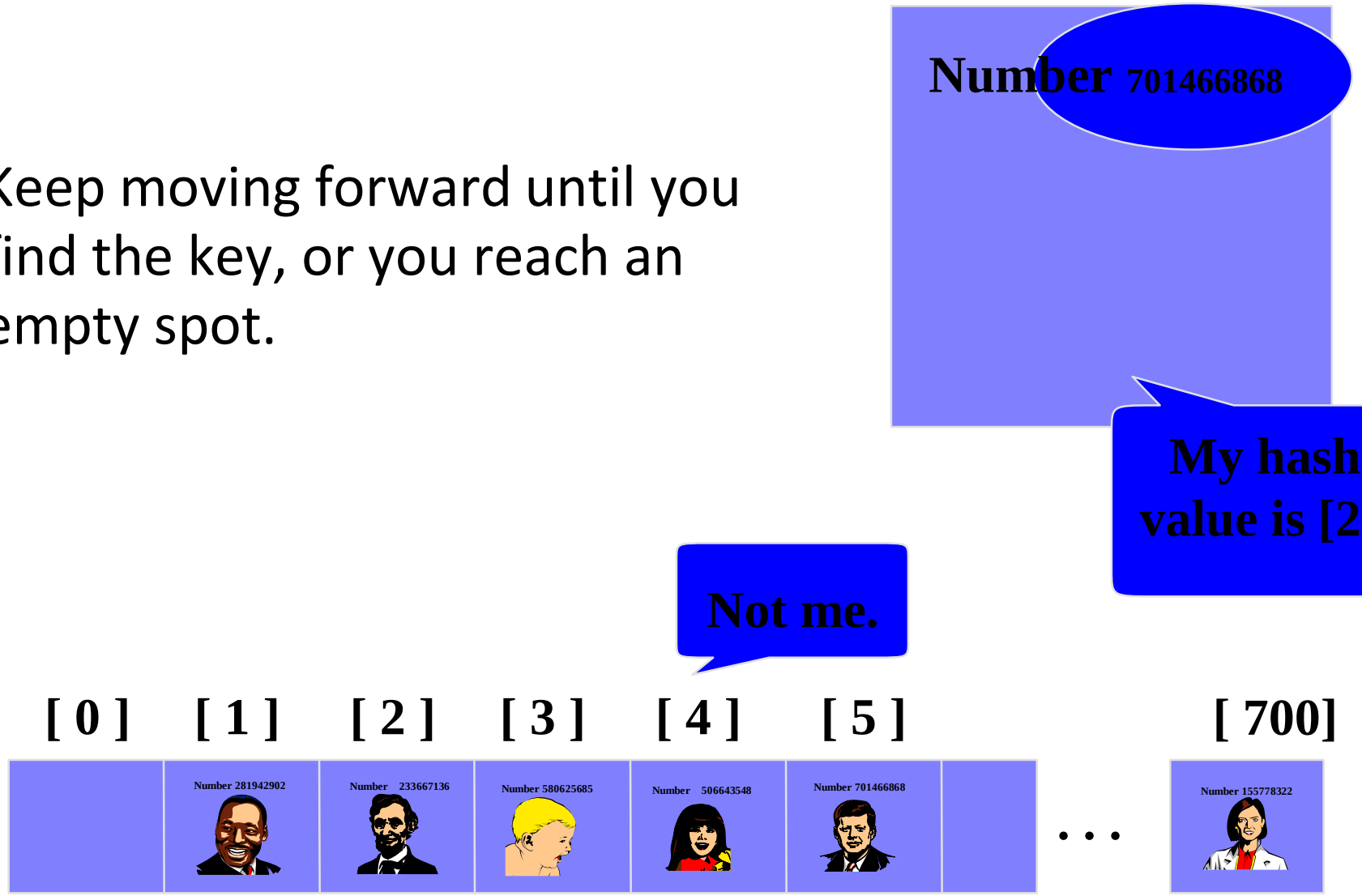
Searching for a Key

Keep moving forward until you find the key, or you reach an empty spot.



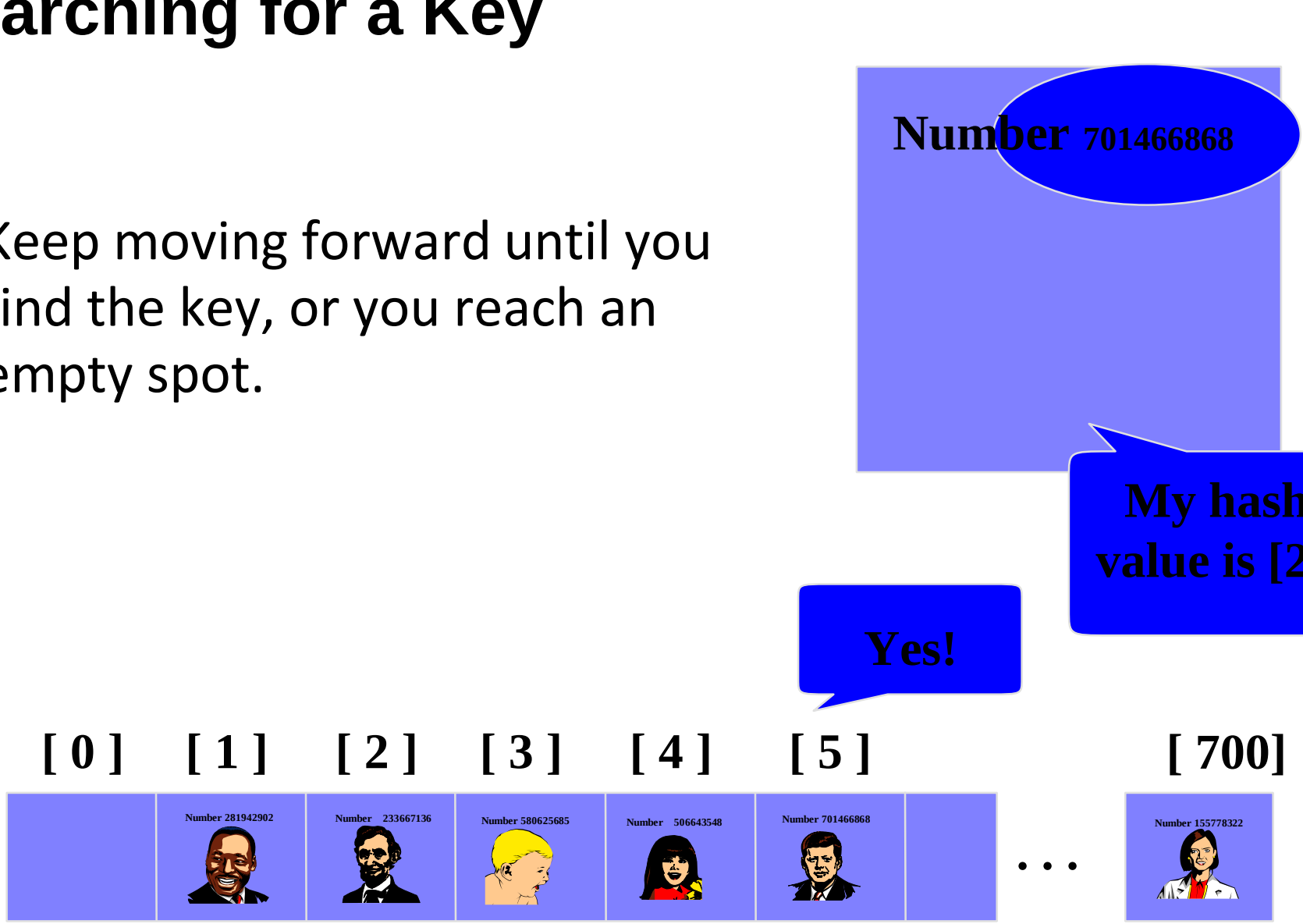
Searching for a Key

Keep moving forward until you find the key, or you reach an empty spot.



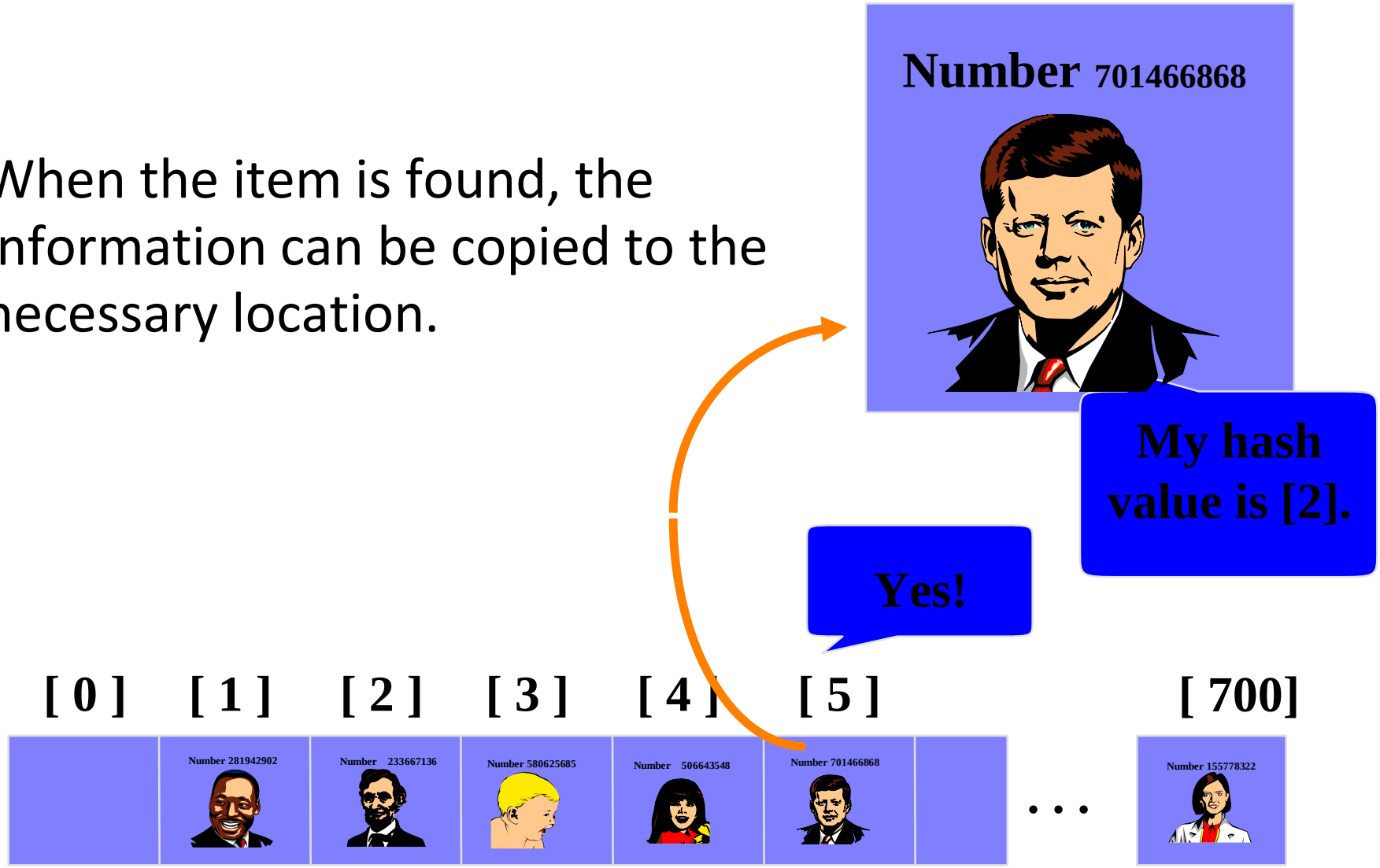
Searching for a Key

Keep moving forward until you find the key, or you reach an empty spot.



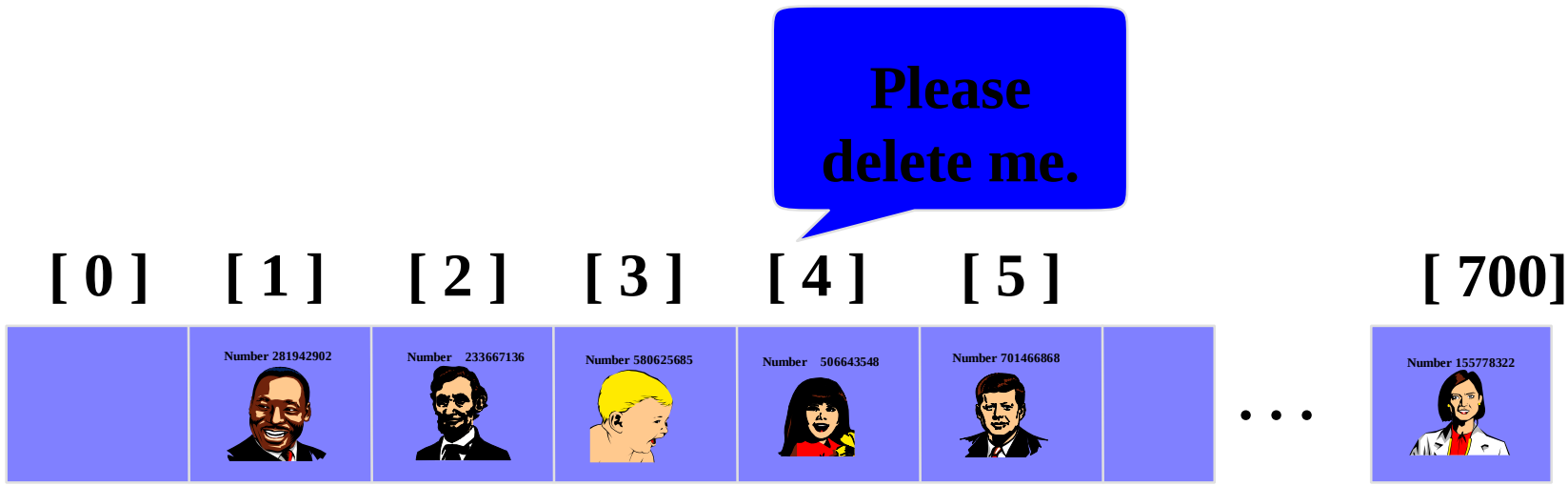
Searching for a Key

When the item is found, the information can be copied to the necessary location.



Deleting a Record

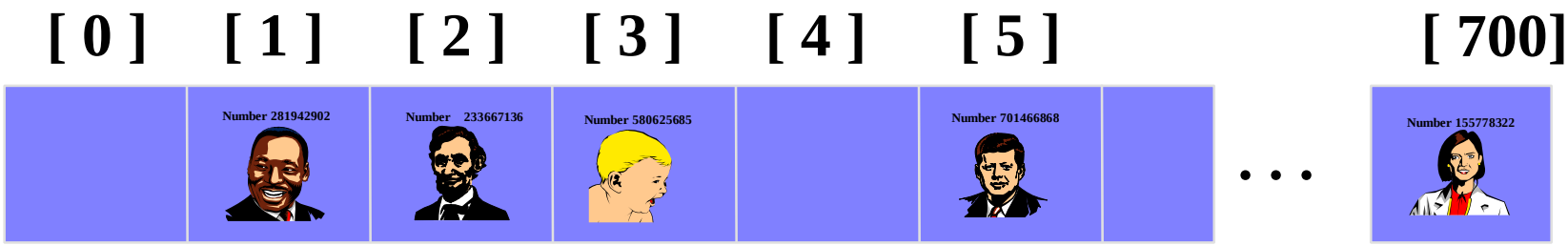
Records may also be deleted from a hash table.



Deleting a Record

Records may also be deleted from a hash table.

But the location must not be left as an ordinary "empty spot" since that could interfere with searches.

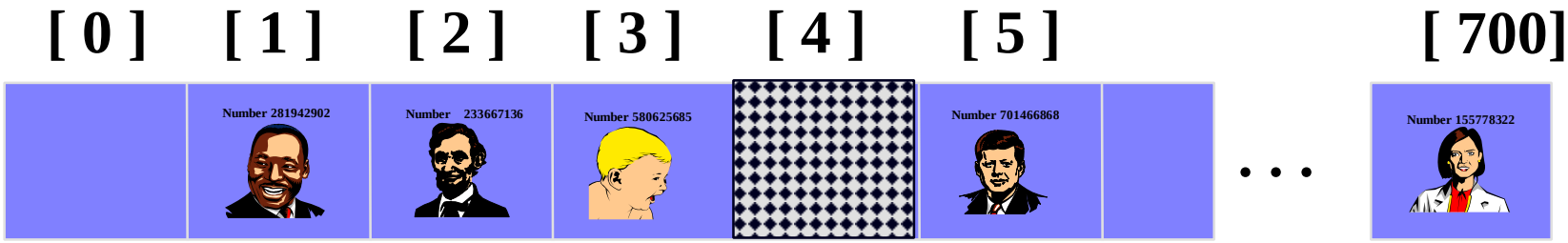


Deleting a Record

Records may also be deleted from a hash table.

But the location must not be left as an ordinary "empty spot" since that could interfere with searches.

The location must be marked in some special way so that a search can tell that the spot used to have something in it.



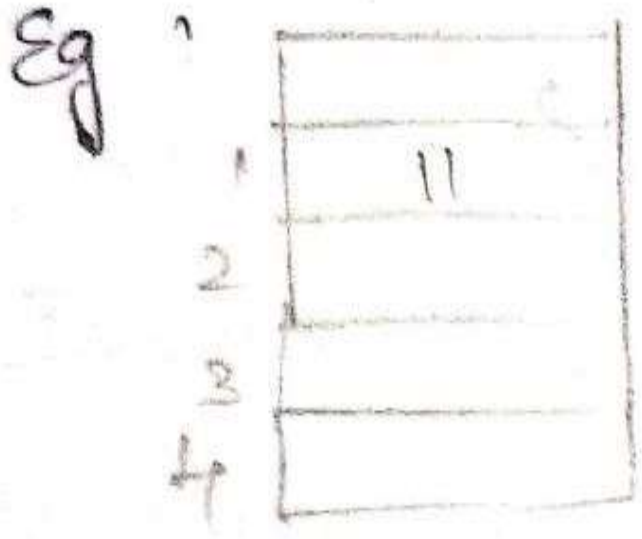
Summary

- Hash tables store a collection of records with keys.
- The location of a record depends on the hash value of the record's key.
- When a collision occurs, the next available location is used.
- Searching for a particular key is generally quick.
- When an item is deleted, the location must be marked in a special way, so that the searches know that the spot used to be used.

Collision :

- Collision occurs when a hash values of a records being Inserted hashes to an Address That already contains a difference Record.

" When Two Key values hash to the position".



Insert 11,21 in hash table

11 =Hash(11) =11%5 =1

21= Hash (21) =21%5 =1

(collision occur)

Collision Resolution :

- The Process of finding another Position for The Collide Record Is said to be collision Resolution Strategy.

Two categories of Hashing .

1. **Open Hashing**

   eg: Separate Chaining.

2. **Closed Hashing .**

   eg : Open Addressing ,Rehashing and Extendable hashing.

Open Hashing :

- Each Bucket in the Hash table is the head of a Linked List.
- All Elements that hash to a Particular Bucket are Placed on the Buckets Linked List .

Closed Hashing:

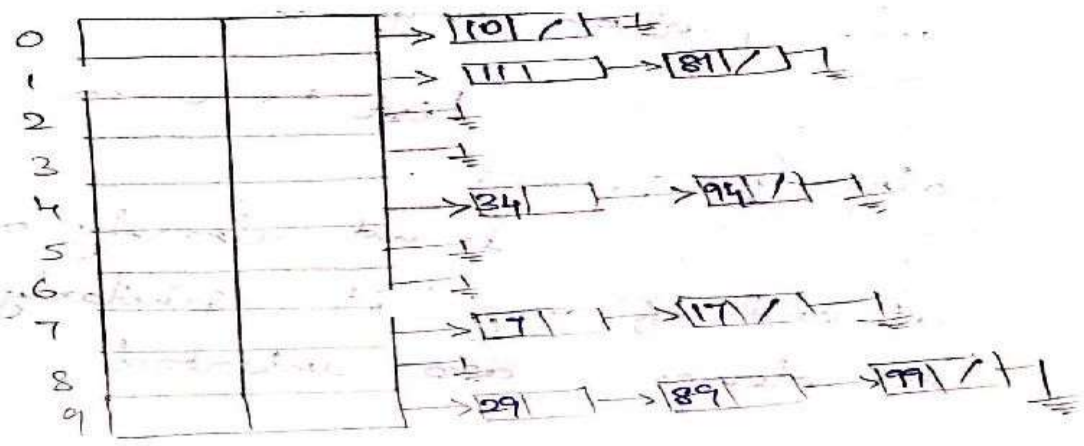
- Ensures that all elements are stored directly in to the Hash Table.

Separate Chaining :

- Separate Chaining is an open hashing Technique
- A Pointer fields is added to each record Location.
- In this method the table can never overflow since the linked are only extended upon or New Keys.

Example:

10 , 11 , 81 , 10 , 7 , 34 , 94 , 17 , 29 , 89 , 99



Hash ('Keyvalue) = keyvalue %. table size

Insert = 10 : Hash(10) = 10 %. 10 = 0

Insert 11 : Hash (11) = 11 %. 10 = 1

Insert 81 : Hash (81) = 81 %. 10 = 1

Insert 10 ⇒ ×

Insert 7 : Hash (7) = 7 %. 10 = 7

Insert 34 : Hash (34) = 34 %. 10 = 4

Insert 94 : Hash (94) = 94 %. 10 = 4

Insert 17 : Hash (17) = 17 %. 10 = 7

Insert 29 : Hash (29) = 29 %. 10 = 9

Insert 89 : Hash (89) = 89 %. 10 = 9

Insert 99 : Hash (99) = 99 %. 10 = 9

The element 81 collides to the same of the hash value to place the value 81 at this position perform the following.

1. Traverse the list to check whether it is already present.
2. Since, it is not already present, insert at end of the list similarly, the rest of the elements are inserted.

Advantages:

- More number of elements can be inserted as it uses of linked list.
- Collision resolution is simple and efficient.

Disadvantages:

It requires pointers, which occupies more memory space.

Closed hashing:

- Collide elements are stored at another slot in the table. Ensures that all elements stored directly in the hash table.

Eg: Open addressing

Rehashing and extendable hashing.

Thank you

MUTHAYAMMAL ENGINEERING COLLEGE

DEPARTMENT OF INFORMATION TECHNOLOGY

Subject Code:19ITC01

Subject Name: Data Structures

Open Addressing:

- Open addressing also called closed hashing which is an attentive to resolve the collision with linked list.
- If a collision occurs, alternative cells are tried until an empty cell is found (i.e) cells ho(x), h1(x), h2(x) are tried in succession.

  There are three common collision strategies, there are

  1. Linear Probing
  2. Quadratic Probing
  3. Double Hashing

1.Linear Probing:

Key value :18,70,65,51,13

Table size : 7

Formula :

$H(k,v) = k.v \,\%\, T.S$

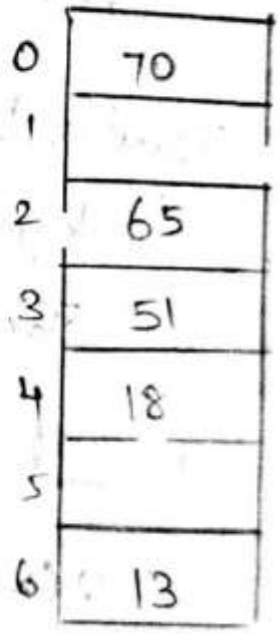
$H(18) = 18 \,\%\, 7 = 4$

$H(70) = 70 \,\%\, 7 = 0$

$H(65) = 65 \,\%\, 7 = 2$

$H(51) = 51 \,\%\, 7 = 2$

$H(13) = 13 \,\%\, 7 = 6$



Quadratic probing

Key Value : 18, 70, 65, 51, 13.
Table Size : 7

Formula :

(i) $H(k.v) = k.v \ \%.T.S$

(ii) $a + i^2$

where

$i = 1, 2, 3, \ldots n$

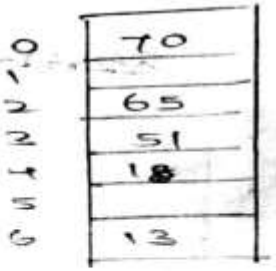
Solution :

$H(k.v) = k.v \ \%.T.S$

$H(18) = 18 \ \%.7 = 4$

$H(70) = 70 \ \%.7 = 0$

$H(65) = 65 \ \%.7 = 2$



$H(51) = 51 \ \%.7 = 2 \rightarrow$ collision.

Formula location of collision:

$a + i^2$

$= 2 + 1^2 = 2 + 1 = 3$

$H(13) = 13 \ \%.7 = 6$

Double hashing

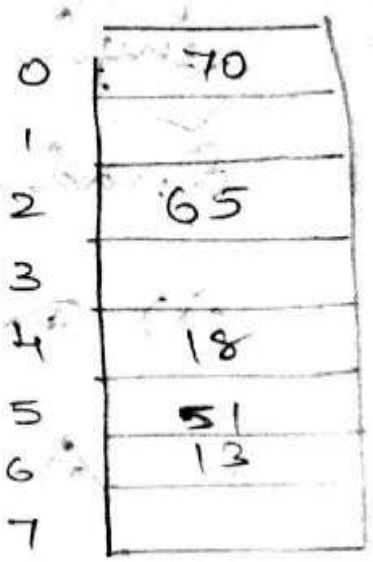
Keyvalue : 18, 70, 65, 51, 13
Table size : 7

Formula :

$H(k.v) = k.v \ \%.Ts \rightarrow a$

$H'(k) = 9 - (k.v \ \%.9) \rightarrow a'$

$(a+a')\ \%\ T.S$



Solution :

$H(K.V) = K.V\ \%\ T_S$

$H(18) = 18\ \%\ 7 = 4$

$H(70) = 70\ \%\ 7 = 0$

$H(65) = 65\ \%\ 7 = 2$

$H(51) = 51\ \%\ 7 = 2 \rightarrow$ collision

$H'(K.V) = 9 - (K.V\ \%\ 9)$

$H'(51) = 9 - (51\ \%\ 9)$

$= 9 - 6 = 3 \rightarrow a'$

$a + a' = 2 + 3 = 5$

$a + a'\ \%\ T.S = 5\ \%\ 7 = 5$

$H(13) = 13\ \%\ 7 = 6.$

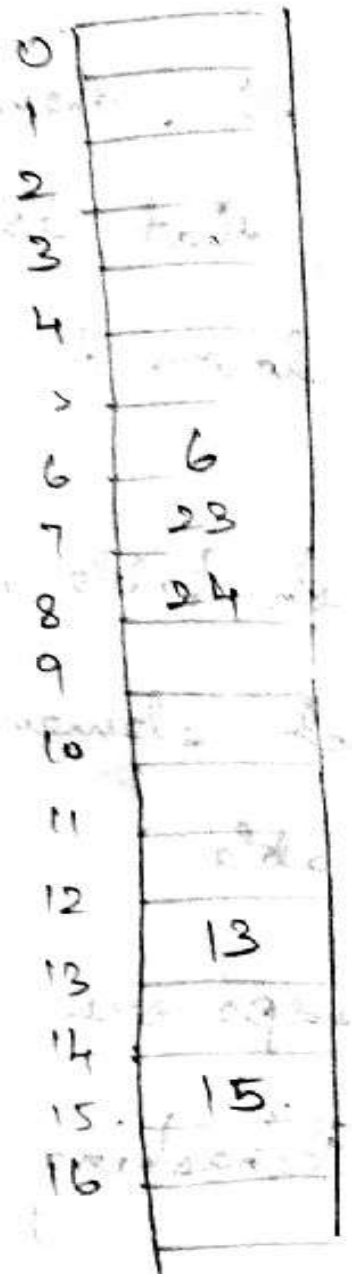
Rehashing:

- If the table get's to full. Then the rehashing method new tables that is about twice as with and scan down the entire original hash table.
- The entire original hash table, computing the new hash value for each element and in sorting it in the new table.
- Rehashing is very expensive operations, the running time is O(N), Rehashing can be implements is several ways with Quadratic probing such as

  i). Rehash as soon as the table is half full.

- A new table is created as table so full. The size of the table so full. The size of the table is 17, as this to the first prime (i.e) a) twice as large as the old table size.

- The new hash function is h(x) = Xmod 17 the old table is scanned and the elements, 6,15,24,23 and 13 are inserted into the new table.



i) insert (6) = 6 %.17 = 6

ii) insert (15) : 15 %.17 = 15

iii) insert (23) : 23 %.17 = 6

$H_1(x) = (6+1) \%.17 = 7 \%.17 = 7$

iv) 24 %.17 = 7

$= (7+1) \%.17 = 8 \%.17$

$= 8$

v) insert 13

$= 13 \%.17$

$= 13.$

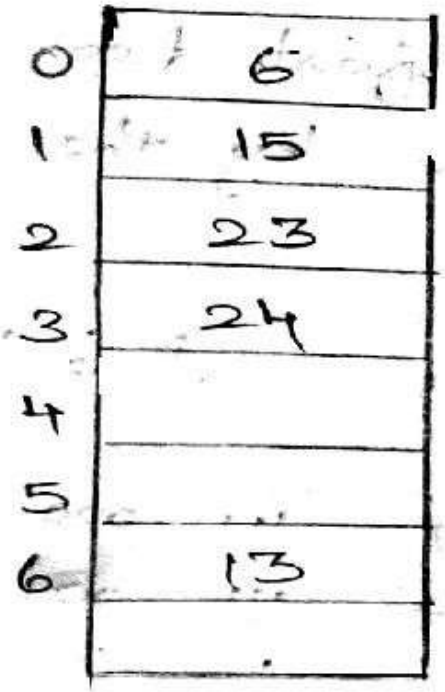
Advantages:

- Programmer does not about the table size.
- Simple to implement.
- It can be used in order data structure as well.

ii) Rehash only when an insertion fails. Suppose the elements 13,15 ,23 ,24 & 6 are insert into an open addressing hash table of size.

Hash function h(x) = x mod 7

Hash Function $h(x) = x \bmod 17$



1) Insert = 13

H(13) = 13 %. 7

= 6

2) Insert = 15

H(15) = 15 %. 7

= 1

3) Insert = 24

H(24) = 24 %. 7

= 3

4) Insert = 6

Hash(6) = 6 %. 7

= 6 (collaid

Hash(6) = (6+1)

= 7 %. 7 =

Linear Probing is used to resolve the collisi

If 23 is inserted into the Table ,the Resulting Table Will be over 70% Full.

1) Insert 23:

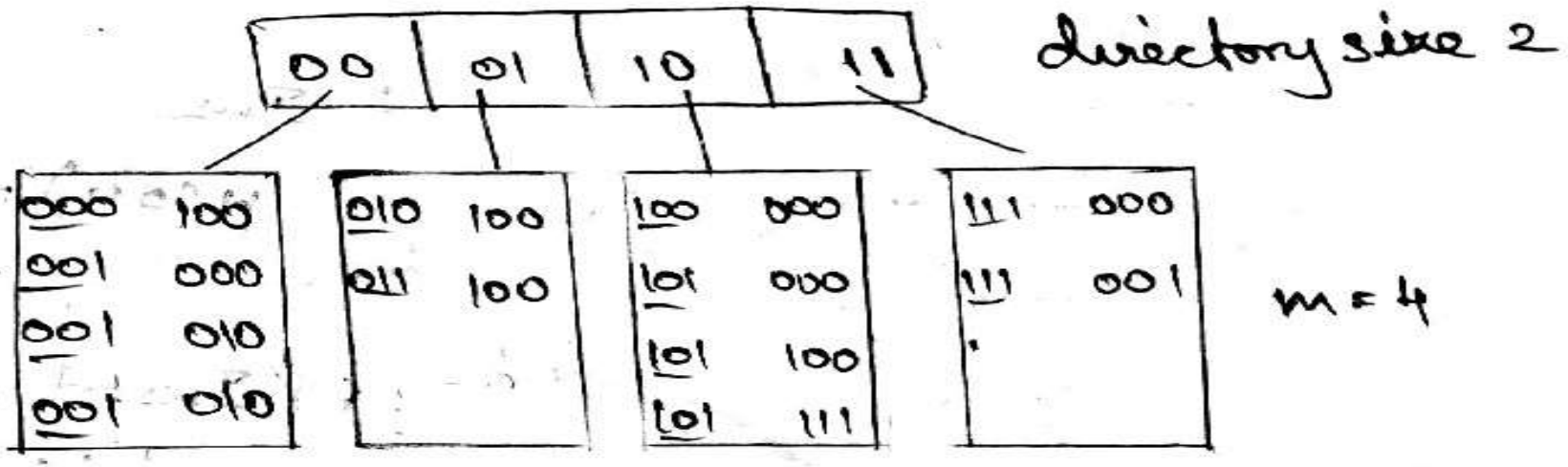
Hash(23)=23%7

=2

Extendable Hashing:

- Extendable Hashing Allows a find to be Performed in two disk accesses. Insertion Also Requires few Disk accesses.
- Let us support consider our data consist of determined by the leading two bits are the data .

- In each leaf has upto m=4 elements ,identified This is indicated by the number in Parenthesis.



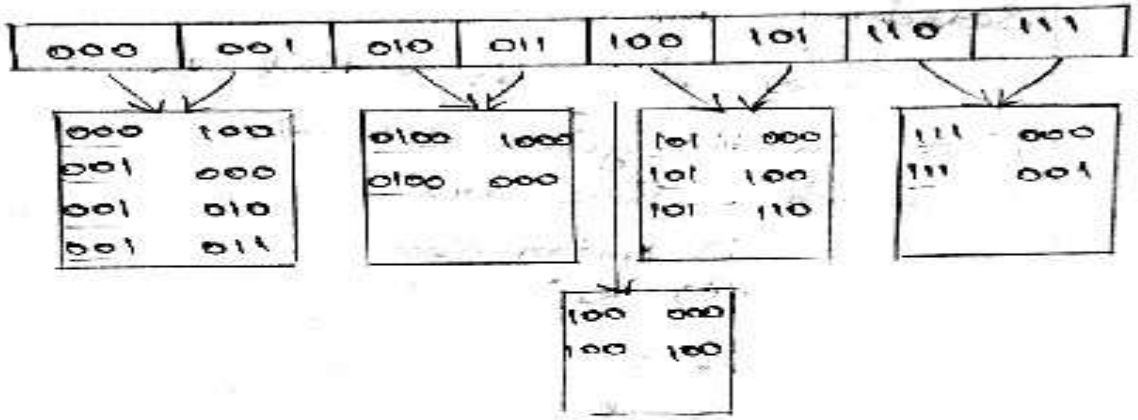
Extendible hashing original data

Suppose that we want to insert the key

100

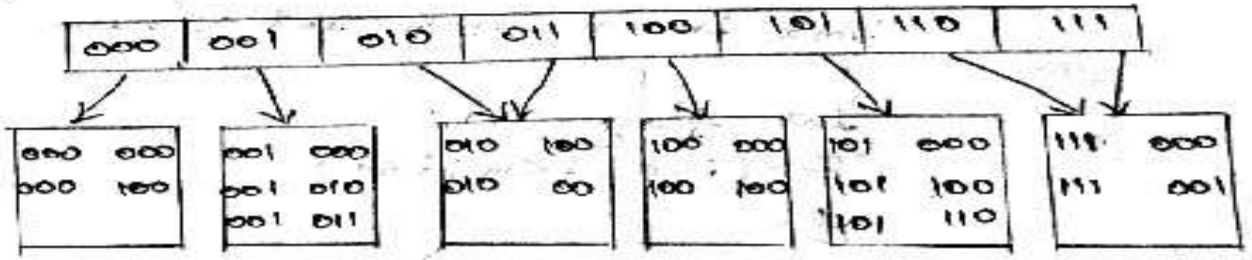
This would go into the third leaf but as the third leaf is already full, that there is no room.

We split this leaf into two leaves, which are now determined by the first three bits now the directory size is increased to three.



extendible hashing after insertion of 100 100 and directory split

Similarly 000 000 is inserted, then first leaf is split



Advantages:

- Provides quick access times for insert an find operation on large database.

Disadvantages:

- This algorithm does not work if there are more then m duplicates.

Thank you

MUTHAYAMMAL ENGINEERING COLLEGE

DEPARTMENT OF INFORMATION TECHNOLOGY

Subject Code:19ITC01

Subject Name: Data Structures

Searching Algorithm:

Searching is method to search a data item in the given set. There are two types of search.

There are

1. **Linear Search**

2. **Binary Search**

Linear Search:

- Linear search is used to search and data item in the given set in the sequential manner Starting from the first element it is also called as Sequential Search.

Routine for Linear Search:

```
void linear search(int x, int a[], int n)
{
int flag=0,i;
for(i=0;i<n;i++)
{
```

```
    if(x==a[i])
{
    flag=1
    break;
}
}
    if(flag==1)
    print f(“the element is found”);
else
    print f(“the element is not found”);
}
```

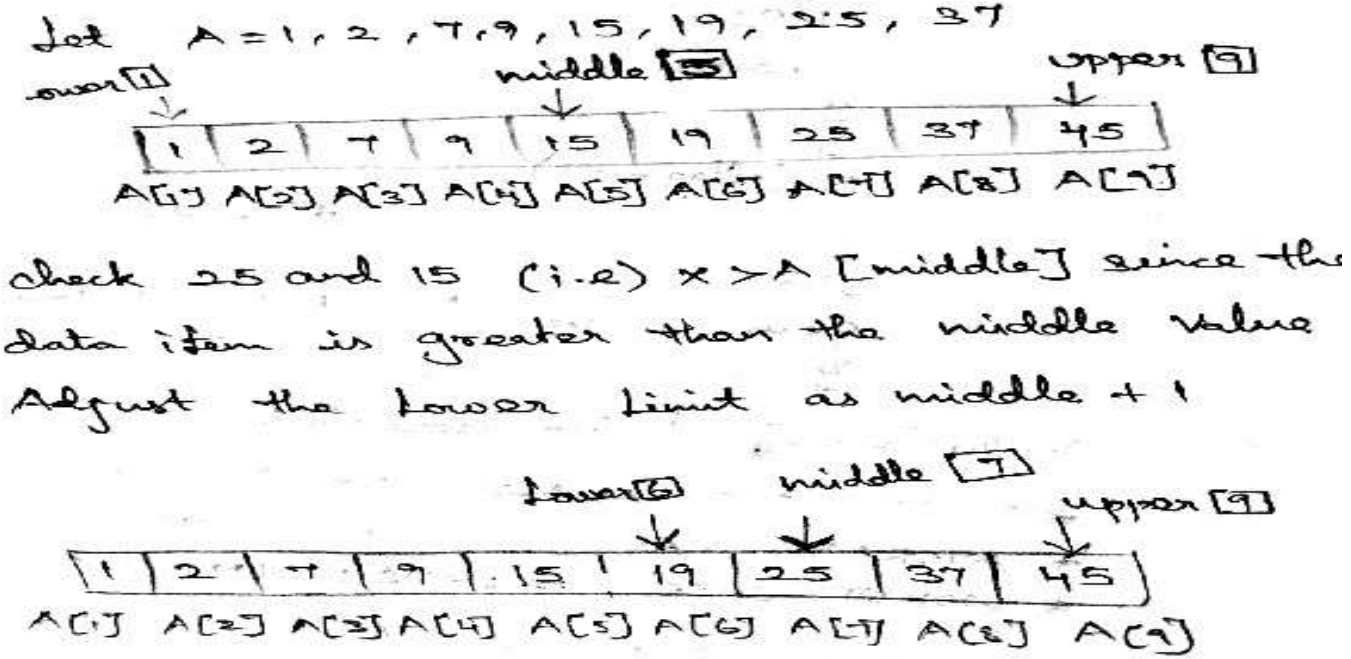
Analysis of Linear Search :

- BEST CASE ANALYSIS:0(1)
- AVERAGE CASE ANALYSIS:0(N)
- WORST CASE ANALYSIS: 0(N)

Binary Search:

- Binary Search is used to Search an Item in a Sorted list . In this Method ,Initialize the lower Limit as 1 And Upper Limit as N(N-1)
- The middle position is computed as (Lower +Upper)/2 and check the element in the Middle Position with the Data item to be Searched.

- If the data item is greater then are Middle Value then the Lower limit Is adjusted to one Greater then the middle value.
- Otherwise , The Upper Limit is adjusted to One Less then the Middle Value Ex: X=25.



Routine For Binary search:

```
   void binary search(int x,int a[],int n);
{
   int lower,upper,mid ;
   lower=1;
   upper=n;
   while(lower<upper)
{
   mid=(lower+upper)/2;
   if(x>a[mid])
   lower=mid+1;
```

```
    else if(x<a[mid])
    upper=mid-1;
else
{
    printf("element is found");
    break;
}
}
}
```

Analysis;

- BEST CASE ANALYSIS: 0(1)
- AVERAGE CASE ANALYSIS:0(logN)
- WORST CASE ANALYSIS: 0(LogN)

Thank you

MUTHAYAMMAL ENGINEERING COLLEGE

DEPARTMENT OF INFORMATION TECHNOLOGY

Subject Code:19ITC01

Subject Name: Data Structures

SORTING

Def: Sorting is a Process (or) Technique of Arranging a group or a Sequence of Data Elements in an order either in ascending or descending.

Two Types of Sorting:

1. Internal Sorting
2. External Sorting.

Internal Sorting:

- A sorting in which all records of the file to be sorted should be within the main memory at the time of sorting.

External Sorting:

- Sorting is which at the time of Sorting Some Record of the file to be Sorted can be in the secondary memory.

Internal Sorting:

1. Insertion Sort
2. Shell sort

3. Heap Sort
4. Quick Sort
5. Radix Sort
6. Bubble Sort
7. Selection Sort

External Sorting

1. Merge Sort
2. Two Way Sort
3. Multiple Way Merge Sort
4. Polyphase merge sort.

Insertion Sort:

- Insertion Sort Works by tasking element from the list one by one and inserting them in their current position into a new sorted list.
- Insertion sort consists of N-1 Passes Where N is the number of element to be sorted .
- The ith Pass of insertion sort will insert the ith
- Element A*1+ , A*2+ ,….A*i-1] .After Doing this Insertion the Record occupying A*1+,….A*i+ are in sorted order.

Insertion and Routine:

```
   void insertion sort (element type a[], int N )
{
   int  j, p ;
   element type tmp ;
   for(p=1; p<N ; p++)
{
   tmp =a[p];
   for (j=p ; j>0&& a[j-1]>tmp ;j--)
   a[j] =a[j-1];
   a[j]=tmp ;
} }
```

Example:

Consider an Unsorted array as Follows :

20, 10, 60, 40, 30, 15

Passes of Insertion Sort

								Position Moved
original		20	10	60	40	30	15	
After	i=1	10	20	60	40	30	15	1
	i=2	10	20	60	40	30	15	0
	i=3	10	20	40	60	30	15	1
	i=4	10	20	30	40	60	15	2
	i=5	10	15	20	30	40	60	4
sorted array		10	15	20	30	40	60	

Analysis of insertion Sort :

- WORST CASE ANALYSIS - O(N^2)
- BEST CASE ANALYSIS - O(N)
- AVERAGE CASE ANALYSIS -O(N^2)

Shell Sort :

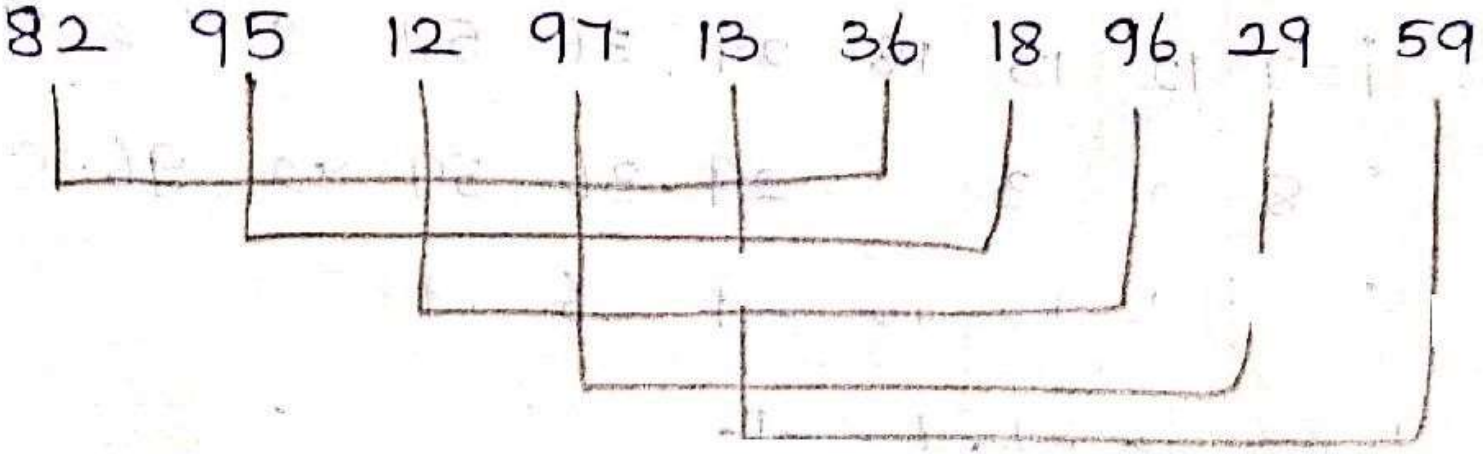
- This method is an Improvement over the Simple Insertion Sort .In this Method The Element at Fixed Distance K (K is Preferably prime Number) or compared .
- The distance will then be decremented by Some fixed amount and again the Comparison will be made . Finally Individuals elements will be compared.

Ex :

82, 95, 12, 97, 13, 36, 18, 96, 29, 59

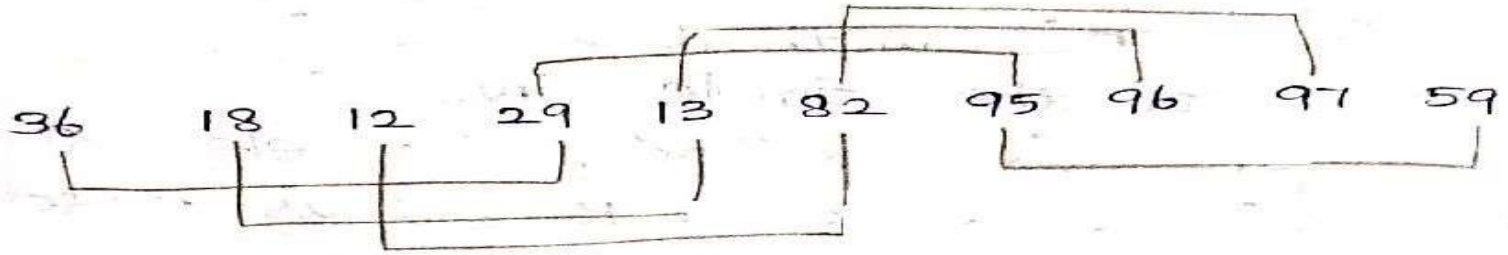
K = 5, 3, 1

Pass 1 : K = 5



36 18 12 29 13 82 95 96 97 59

Pass 2 : K = 3



29: 13 12 36 18 82 59 46 97 95

Pass 3 : K = 1

	29	13	12	36	18	82	59	96	97	95	P.I
i=1	13	29	12	36	18	82	59	96	97	95	1
i=2	12	13	29	36	18	82	59	96	97	95	2
i=3	12	13	29	36	18	82	59	96	97	95	0
i=4	12	13	18	29	36	82	59	96	97	95	2
i=5	12	13	18	29	36	82	59	96	97	95	0
i=6	12	13	18	29	36	59	82	96	97	95	1
i=7	12	13	18	29	36	59	82	96	97	95	0
i=8	12	13	18	29	36	59	82	96	97	95	0
i=9	12	13	18	29	36	59	82	95	96	97	2

The sorted elements are

12, 13, 18, 29, 36, 59, 82, 95, 96, 97

Quick Sort [Partition Exchange Sort]:

- The idea Behind This Sorting is Much easier In the Two Sort List Rather Then One Long list.
- All The Element on the Left Side Of Pivot Should Be Smaller Or Equal To The Pivot.
- All The Element On the Right Side Of Pivot Should Should Be Greater Then Or Equal To Pivot.

The Process for Sorting the Element Quick Sort is as:

I. Take the First Element of List as Pivot .

II. Place At The Proper Place In List So one Element of The List (i.e) Pivot will be at its Proper Place.

III. Create two Sublist's Left Child, Right Child of Pivot.

IV. Repeat the same Process Until all element of List are at Proper Position in List.

For Placing the Pivot at Proper Place we have a Need to do The Following Process:

I. Compare the Pivot Element One by One From Right to Left For Getting The Element Which Has Value Less Then Pivot Element .Interchange The Element With Pivot Element.

II. Now the Comparison will start from the Interchanged element position from left to right for getting the element which has higher Value then Pivot.

III. Repeat the same process Until Process is at its Proper position.

Let us take a list of element and process through quick sorting :

(48), 29, 8, 59, 72, 88, 42, 65, 95, 19, 83, 62

← low

Here we are taking 48 as pivot and we shall as start comparison from right to left.

Now the first element less than 48 is 19 so interchange it with pivot, (i.e) 48.

19 29 8 59 72 98 42 65 95 (48)

high →

19 29 8 (48) 72 88 42 65 95 59 82 6

19 29 8 42 72 88 (48) 65 95 59 82 6

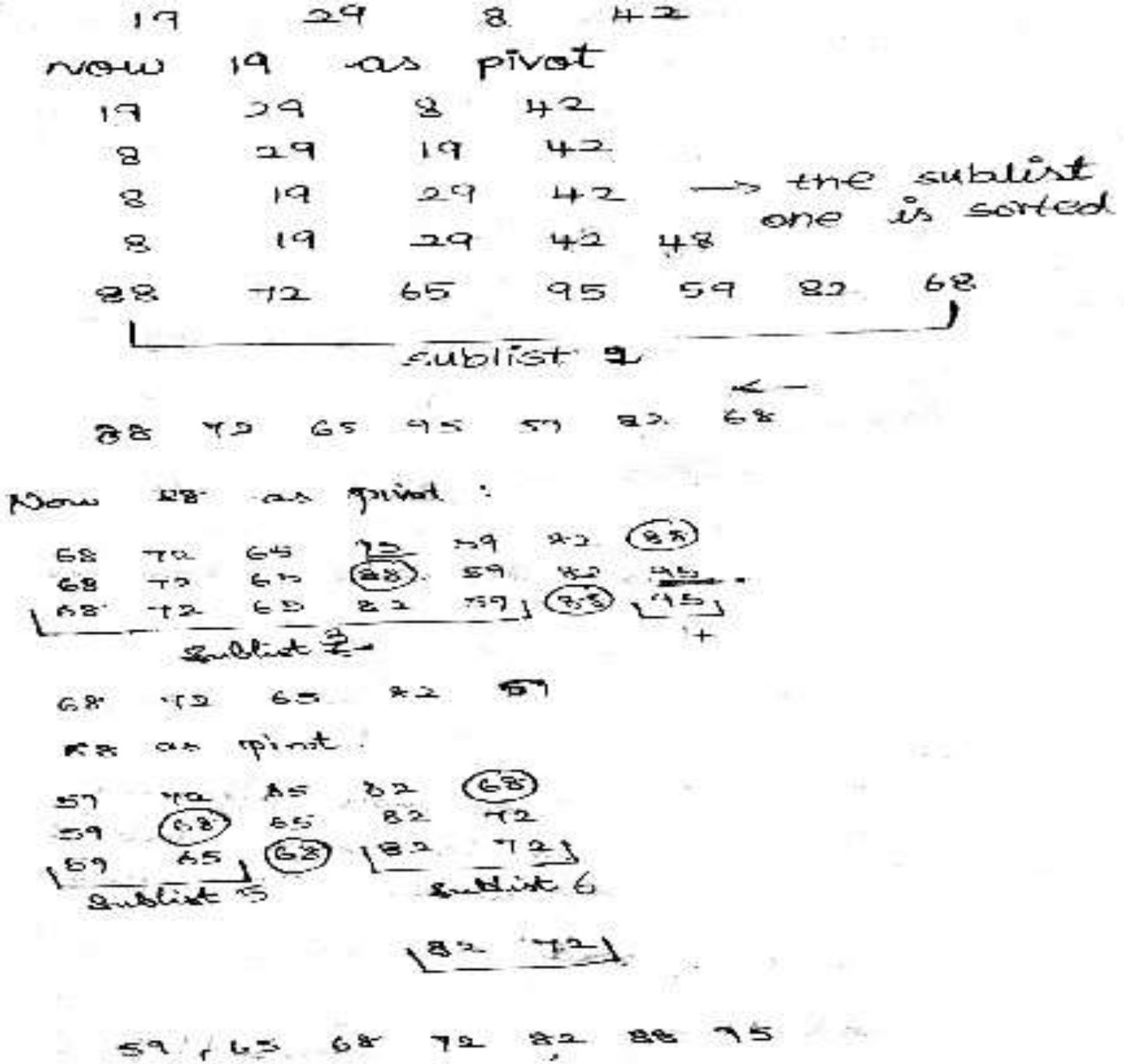
19 29 8 42 (48) 88 72 65 95 59 82

First element greater than 48 is 72.

So, we can divide list into two sublist left and right side first.

19 29 8 42 (48) 88 72 65 95 59 82 6

[Sublist] [Sublist]


19 29 8 42
Now 19 as pivot
19 29 8 42
8 29 19 42
8 19 29 42 → the sublist one is sorted
8 19 29 42 48
28 72 65 95 59 82 68
sublist 2
28 72 65 95 59 82 68
Now 28 as pivot :
68 72 65 72 59 82 (88)
68 72 65 (68) 59 82 85
68 72 65 82 59 (88) 15
sublist 3
68 72 65 82 59
68 as pivot :
59 72 65 82 (68)
59 (68) 65 82 72
59 65 (68) 82 72
sublist 5
sublist 6
82 72
59 65 68 72 82 88 95

Now Combine 2 Sub list

8 19 29 42 48 59 65 68 72 82 88 95

All The Elements Are New Sorted.

Analysis of Quick Sort :

- WORST CASE ANALYSIS - O[N^2]
- BEST CASE ANALYSIS - O[N log N]
- AVERAGE CASE ANALYSIS – O[N log N]

Advantage :

- It is Faster than other O(N log N) algorithm.
- It has better cache Performance and High Speed.

Limitations: Requires More Memory Space

Heap Sort :

- In heap sort the array of Interpret as a binary Tree. This Method pass 2 Phases.
- In Phase 1: Binary heap is Constructed.
- In Phase 2: Delete min Routine is Performed.

Phase 1: Two Properties of Binary Heap :

- Structure Property .
- Heap order Property.

Structure Property :

- For Any Element in Array Position i , The Left child is in 2i+1 (i.e) The Cell after the Left Child

Heap Order Property:

- The key Values in the parent Node is Smaller then or equal to the key Value of any in its Child node.
- To Build the Heap , apply the heap order Property Starting from the Right Most Non – Leaf Node at the Bottom level.

Phase 2:

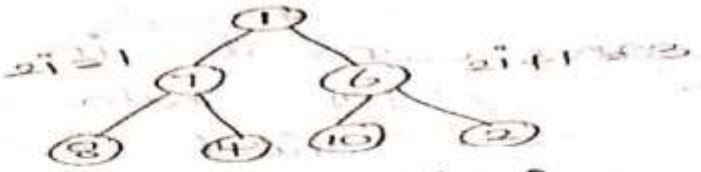
The Array Element are Stored using Deletion Operation.

Example :

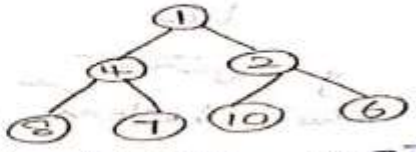
For example :

1	7	6	8	4	10	2
1	2	3	4	5	6	7

Phase 1 :



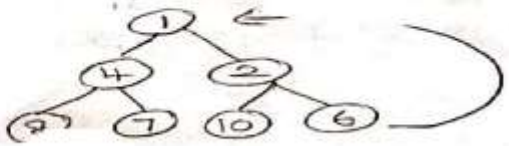
Binary heap satisfying structure property.

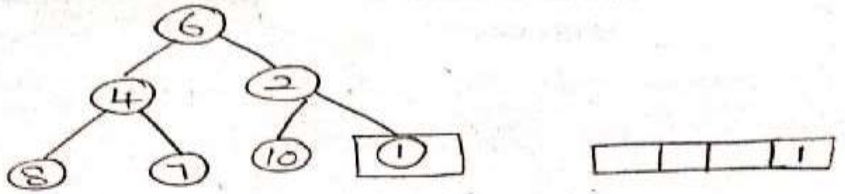


Binary heap satisfying heap order property.

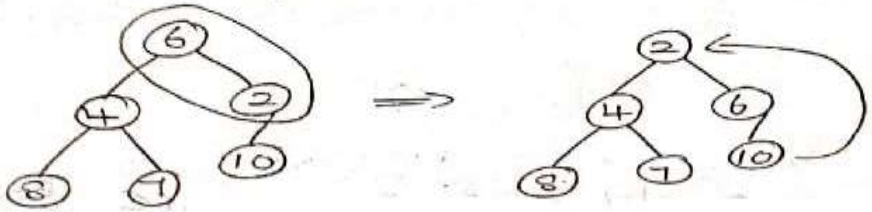
Phase 2 :

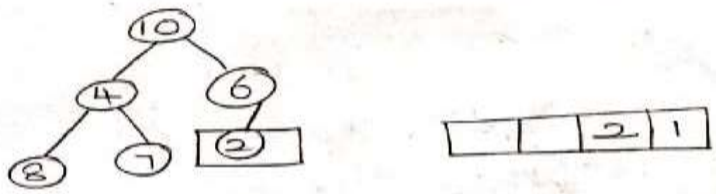
Remove the smallest element from the heap and place it in the array.



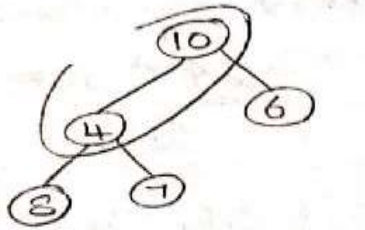


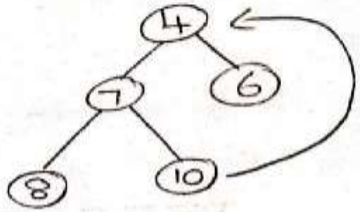
satisfies heap order property

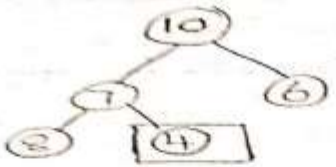


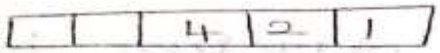


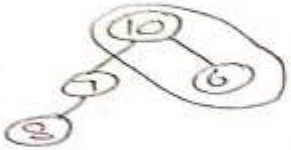
swap (4) :



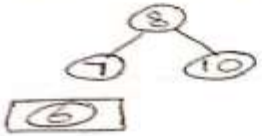


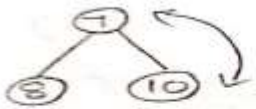


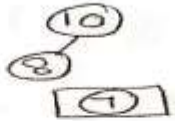


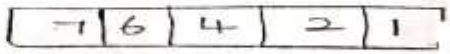


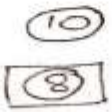
similarly apply the procedure for other element.

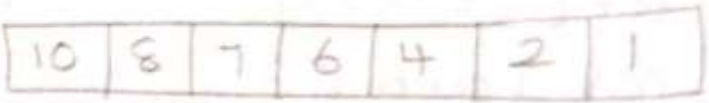












Routine for Heap sort :

Routine for Heap Sort :

```
#define left child [i] (2*(i)+1)
void perdown (element type A[] , int i , int N)
{
int child ;
element type tmp ;
for(tmp =A[i] , left child (i)<N , i=child)
{
child =left child (i);
if (child!=N-1&&A[child +]>A[child])
child ++ ;
```

```
    if (tmp<A[child])
    A[i]=A[child]);
else
Break ;
}
    A[i] =tmp;
}
    void heap sort(element type a[],int N)
{
    int I;
    for(i=N/2;i>=0 ; i--)
```

```
    perdown (A ,i , N);
    for(i=N-1; i>0; i--)
{
    swap (&A[0] ,&A[i]);
    per down (A,o,i);
}
}
}
```

Analysis of Heap Sort :

- Worst Case Analysis =O(N log N)
- Best case Analysis = O(N log N)
- Arg Case Analysis = O(N log N)

Advantages:

- It is efficient for Sorting Large number of Element.
- It has the Add of Worst case.

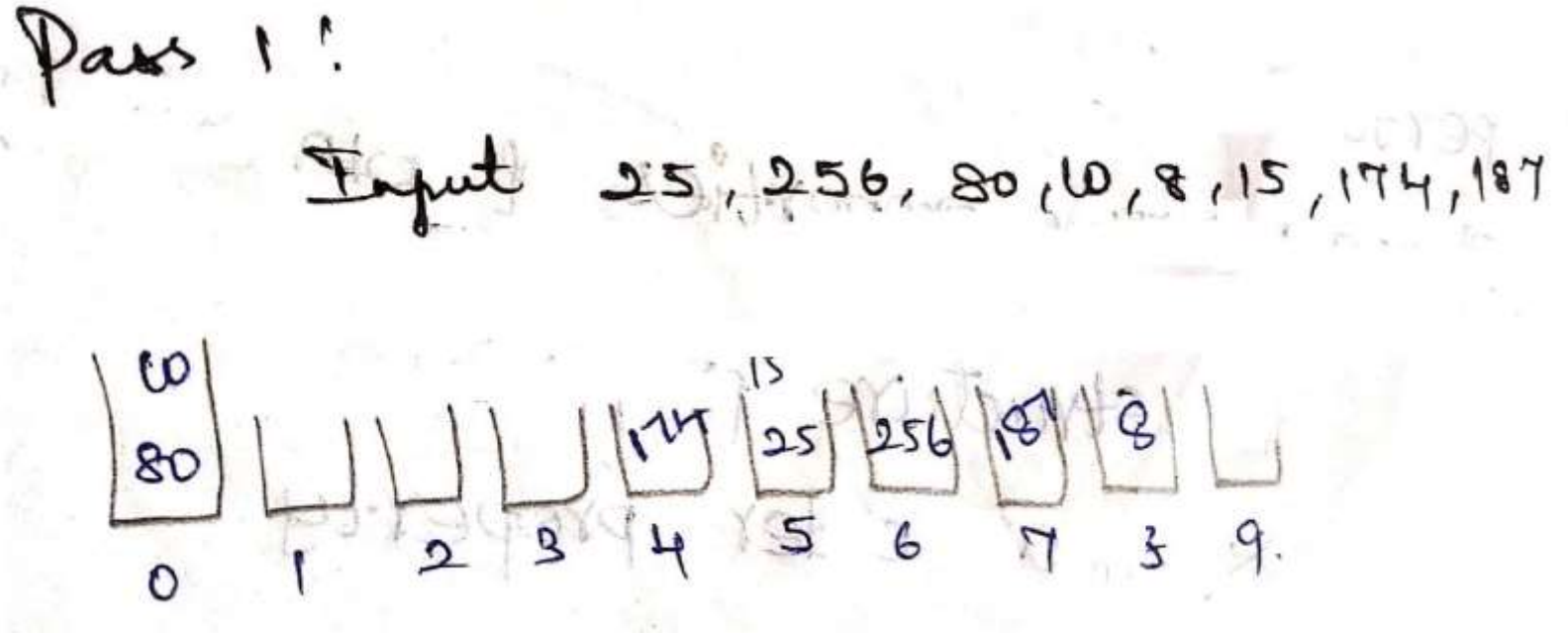
Limitation :

- It is Not a stable Sort .
- It Require Most Processing Time .

Radix Sort :

- Radix Sort is one of the Linear Sorting algorithm for Integers.
- It is generated from radix Sort.
- It can be performed using Bucket 0 to 9.

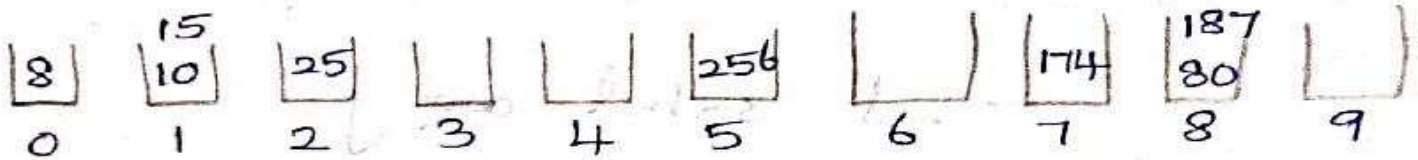
- It is also called as Binsort.
- In First Pass all element arranged according to the least Significant digit. In Second Pass ,the element are arranged according to the next least significant digit and so on.



After pass 1 : 80, 10, 174, 25, 15, 256, 187, 8.

Pass 2 :

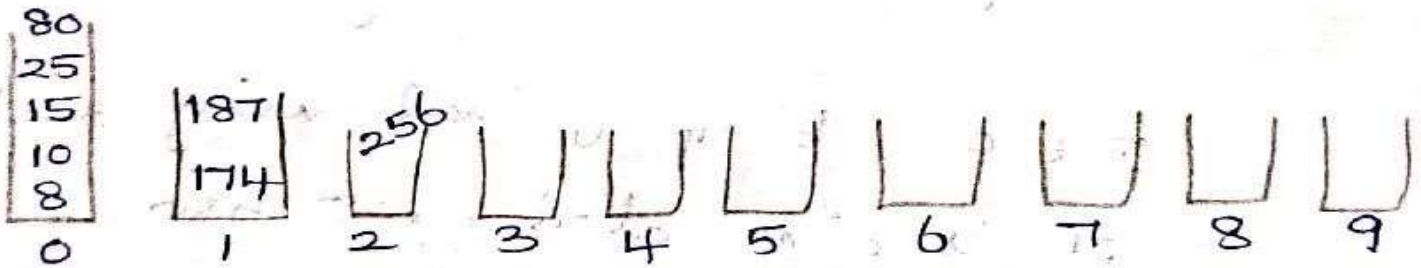
Input : 80, 10, 174, 25, 15, 256, 187, 8.



After pass 2 : 8, 10, 15, 25, 256, 174, 80, 187

Pass 3 :

Input : 8, 10, 15, 25, 256, 174, 80, 187



After pass 3 : 8, 10, 15, 25, 80, 174, 187, 256

External Sorting
Merge Sort :

- The most common algorithm used in external Sorting is the merge sort this algorithm follows **Divide and Conquer strategy**.

Merge Sort Routine :

```
void Msort ( element type A[], element type
tmparray [],int left ,int right)
int center ;
if (left<right)
{
```

```
    center =(left +right)/2;
    msort(A, tmp array, left ,center ) ;
    msort (A,tmp array , center , right ) ;
    merge (A , tmp array ,left ,center+1,right);
}
}
    void merge sort (element type A[],int N))
{
    element type *tmparray ;
    tmparray =malloc(N*sizeof(element type ));
```

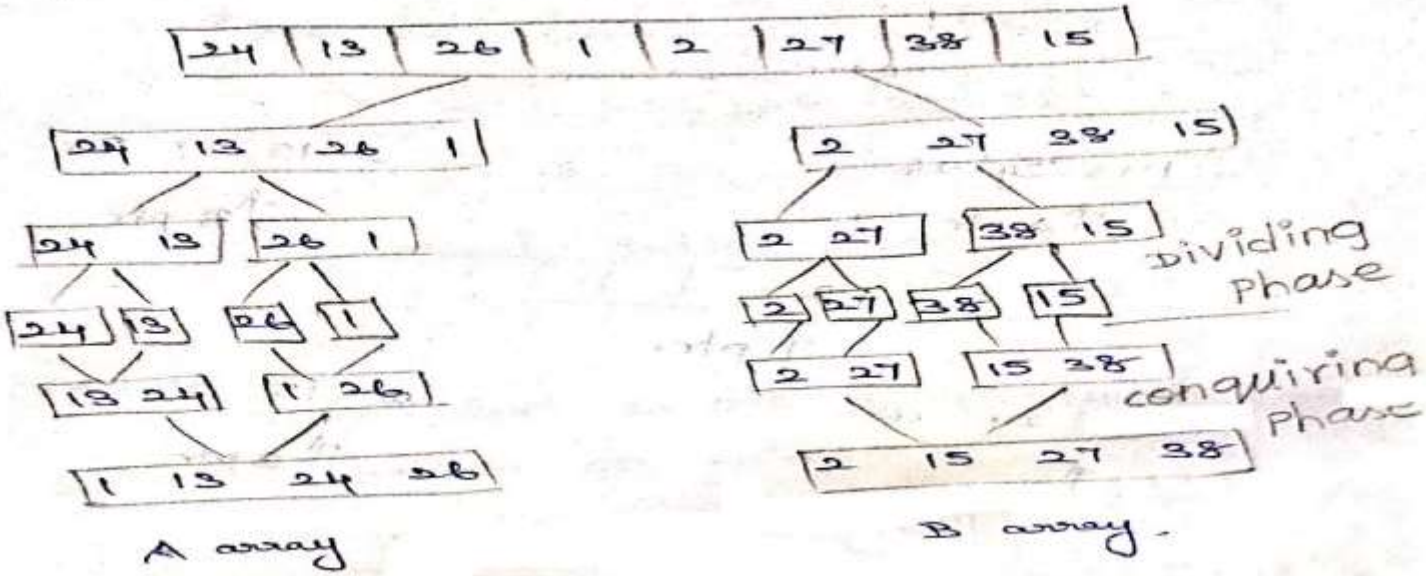
```
   if (tmparray!=NULL)
{
   msort (A,tmparray,0,N++);
   free(tmp array);
}
else
{
   fatal error (“no sapce for tmp array”)
}
}
```

Merge Routine:

```
void merge(element type A[], element type
tmparray [], int LPOS , int RPOS , int rightend)
{
int I, leftend ,numelement , tmppos;
left end =RPOS-1;
TMPPOS =LPOS ;
num element  = rightend –LPOS +1;
while (Lpos<=leftend && RPOS<=rightend)
if (A[LPOS] <=A[RPOS])
tmp array [tmppos++] =A[LPOS++];
```

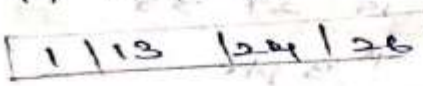
```
else
   tmp array [tmppos++]=A[RPOS++];
   while(LPOS<=leftend)
   tmparray [tmppos++] =A[LPOS++];
   while(RPOS< =rightend)
   tmparray [tmppos++] =A[RPOS++];
   for(i=0; i<numelements; i++; rightend--)
   A[rightend ]=tmparray [rightend];
}
```

Eg: 24, 13, 26, 1, 2, 27, 38, 15.

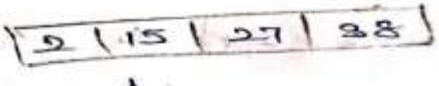


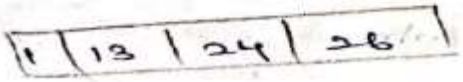
Let us consider first 4 elements.

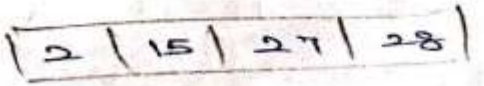
1, 13, 24, 26 as A

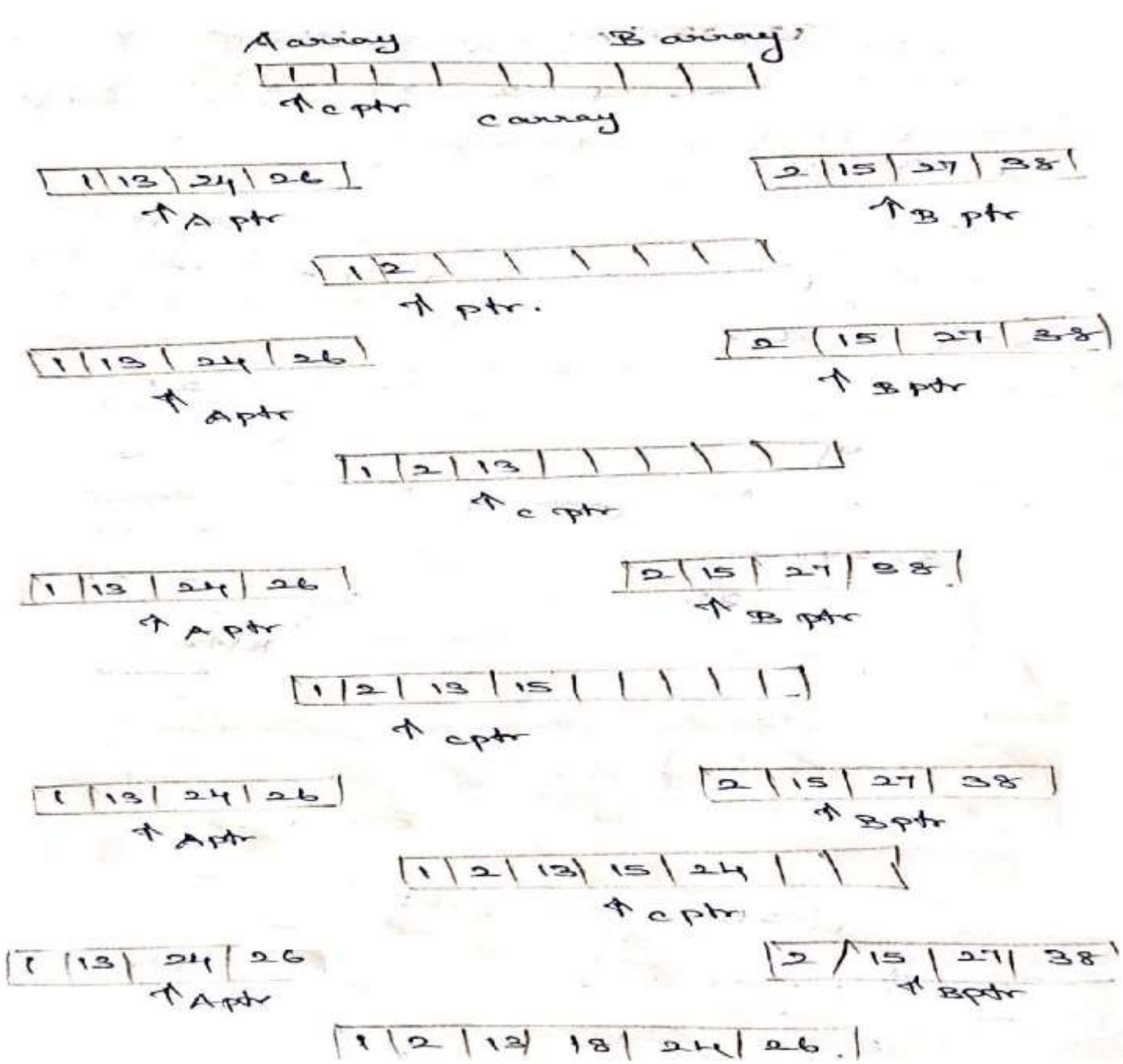


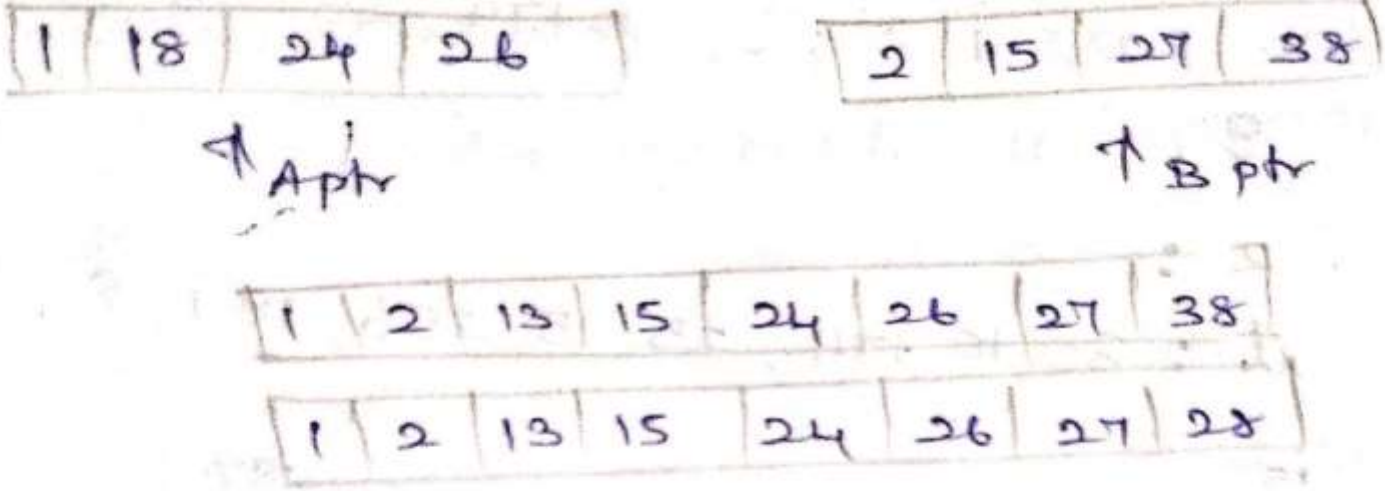
2, 15, 27, 38 as B.








A array
B array
cptr
carray
1 13 24 26
A ptr
2 15 27 38
B ptr
1 2
ptr.
1 13 24 26
A ptr
2 15 27 38
B ptr
1 2 13
c ptr
1 13 24 26
A ptr
2 15 27 38
B ptr
1 2 13 15
cptr
1 13 24 26
A ptr
2 15 27 38
Bptr
1 2 13 15 24
cptr
1 13 24 26
A ptr
2 15 27 38
B ptr
1 2 13 18 24 26


1 18 24 26
2 15 27 38
A ptr
B ptr
1 2 13 15 24 26 27 38
1 2 13 15 24 26 27 28
Final sorted array.

Thank you