Introduction
to
Data Structures

Definition

- Data structure is representation of the logical relationship existing between individual elements of data.
- In other words, a data structure is a way of organizing all data items that considers not only the elements stored but also their relationship to each other.

Introduction

- Data structure affects the design of both structural & functional aspects of a program.

  Program=algorithm + Data Structure
- You know that a algorithm is a step by step procedure to solve a particular function.

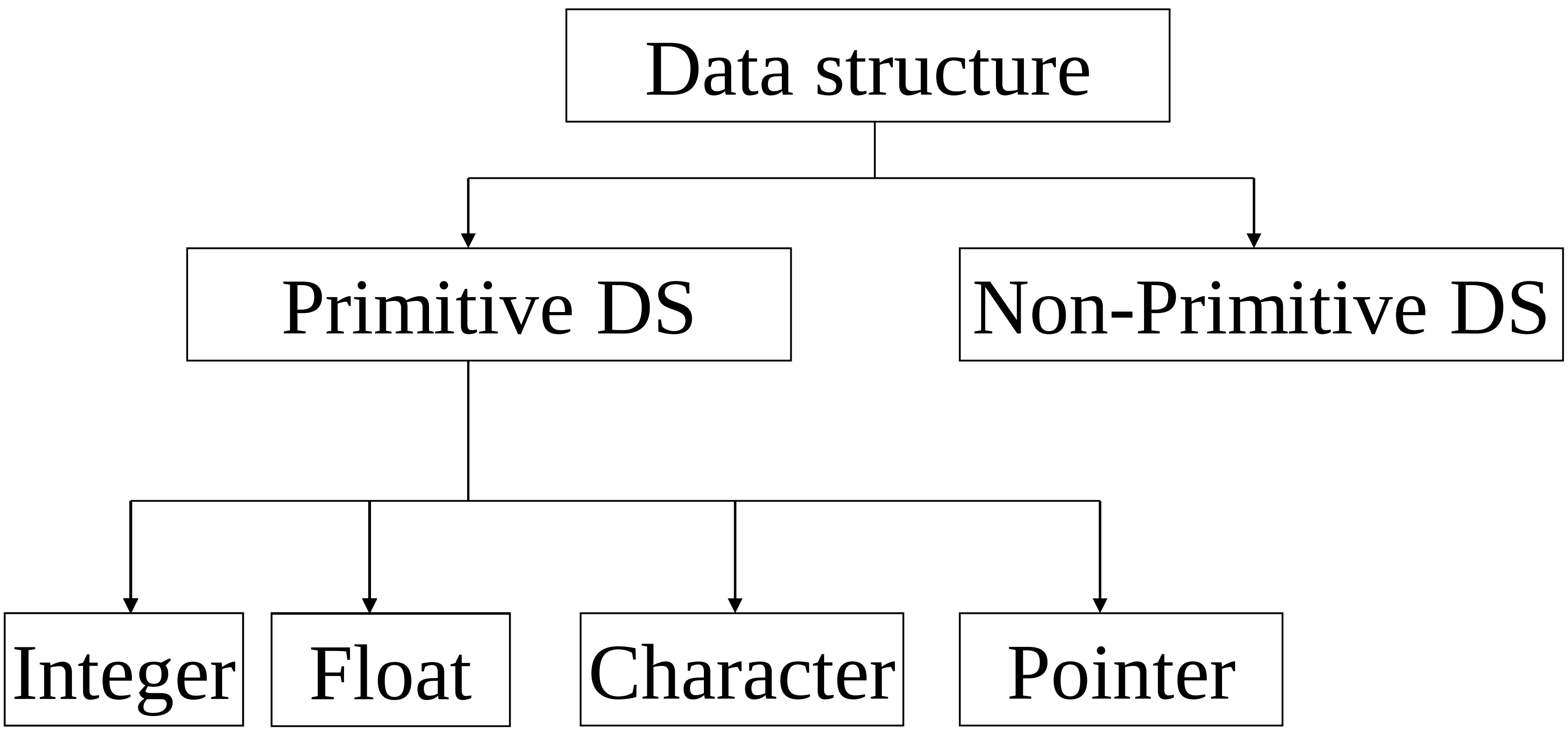
Introduction

- 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.

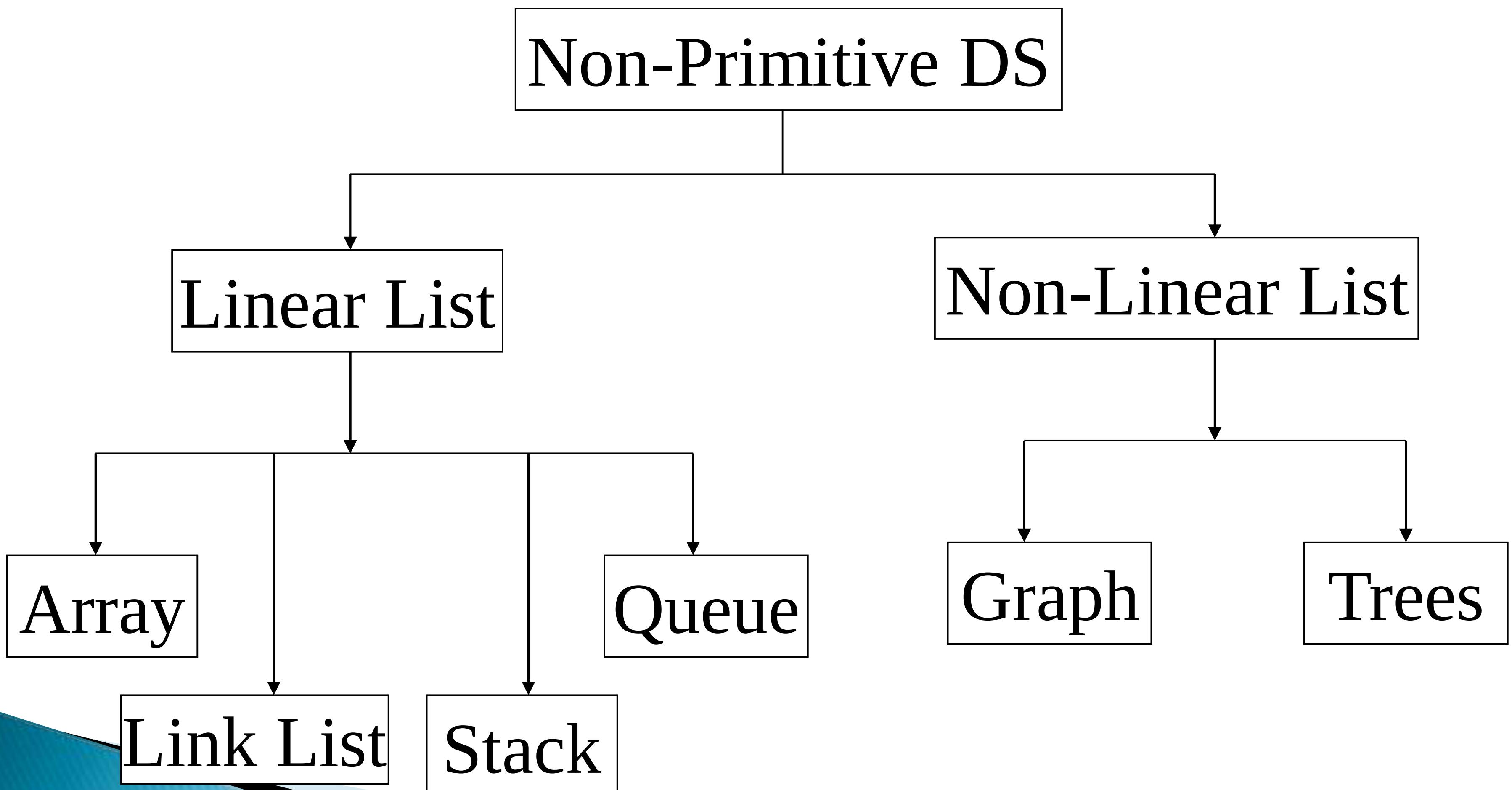
Classification of Data Structure

- Data structure are normally divided into two broad categories:
  - Primitive Data Structure
  - Non-Primitive Data Structure

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

- Lists, Stack, Queue, Tree, Graph are example of non-primitive data structures.
- The design of an efficient data structure must take operations to be performed on the data structure.

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, AVL Tree, graph etc.

Description of various Data Structures : Arrays

- An array is defined as a set of finite number of homogeneous elements or same data items.
- It means an array can contain one type of data only, either all integer, all float-point number or all character.

Arrays

- Simply, declaration of array is as follows:

  int arr[10]

- Where int specifies the data type or type of elements arrays stores.
- “arr” is the name of array & the number specified inside the square brackets is the number of elements an array can store, this is also called sized or length of array.

Arrays

- Following are some of the concepts to be remembered about arrays:
  - The individual element of an array can be accessed by specifying name of the array, following by index or subscript inside square brackets.
  - The first element of the array has index zero[0]. It means the first element and last element will be specified as:arr[0] & arr[9] Respectively.

Arrays

- The elements of array will always be stored in the consecutive (continues) memory location.
- The number of elements that can be stored in an array, that is the size of array or its length is given by the following equation:

  (Upperbound-lowerbound)+1

Arrays

- For the above array it would be
  (9-0)+1=10,where 0 is the lower bound of array and 9 is the upper bound of array.
- Array can always be read or written through loop. If we read a one-dimensional array it require one loop for reading and other for writing the array.

Arrays

- For example: Reading an array

```
For(i=0;i<=9;i++)
    scanf("%d",&arr[i]);
```

- For example: Writing an array

```
For(i=0;i<=9;i++)
    printf("%d",arr[i]);
```

Arrays

- If we are reading or writing two-dimensional array it would require two loops. And similarly the array of a N dimension would required N loops.
- Some common operation performed on array are:
  - Creation of an array
  - Traversing an array

Arrays

- Insertion of new element
- Deletion of required element
- Modification of an element
- Merging of arrays

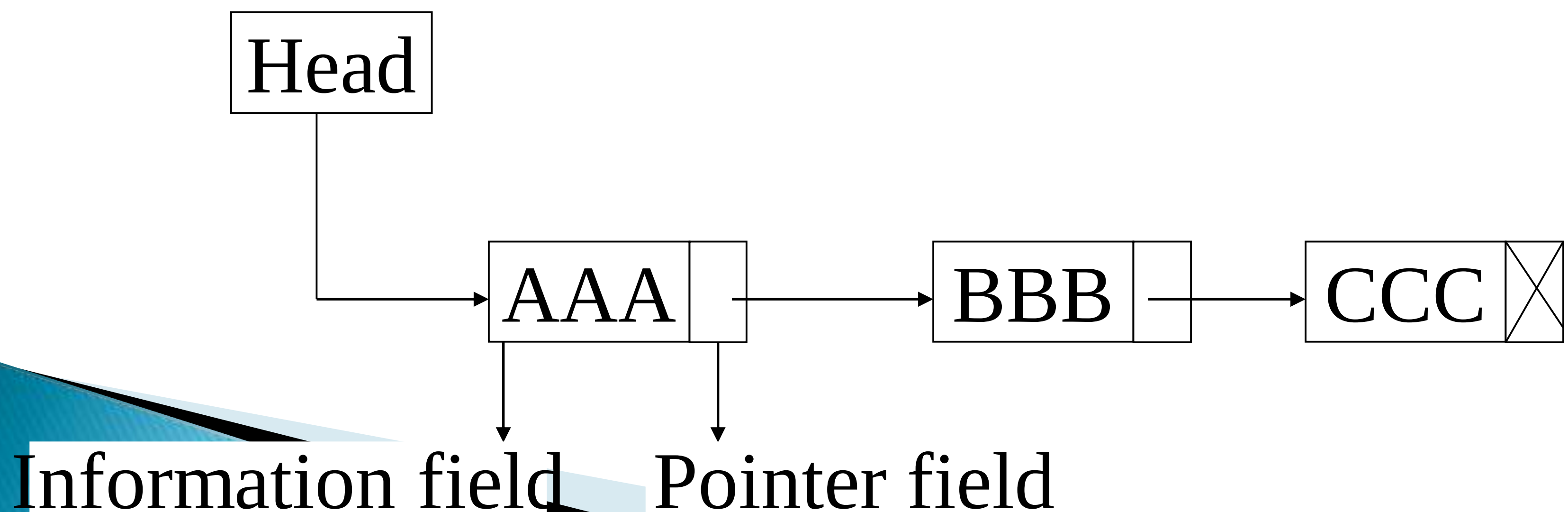
Lists

- A lists (Linear linked list) can be defined as a collection of variable number of data items.
- Lists are the most commonly used non-primitive data structures.
- An element of list must contain at least two fields, one for storing data or information and other for storing address of next element.
- As you know for storing address we have a special data structure of list the address must be pointer type.

Lists

- Technically each such element is referred to as a node, therefore a list can be defined as a collection of nodes as show bellow:

[Linear Liked List]



Lists

- Types of linked lists:
  - Single linked list
  - Doubly linked list
  - Single circular linked list
  - Doubly circular linked list

Stack

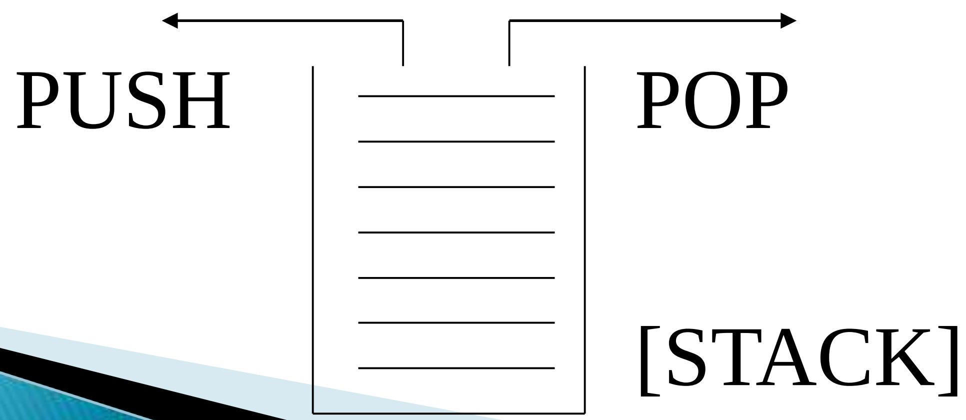
- A stack is also an ordered collection of elements like arrays, but it has a special feature that deletion and insertion of elements can be done only from one end called the top of the stack (TOP)
- Due to this property it is also called as last in first out type of data structure (LIFO).

Stack

- It could be through of just like a stack of plates placed on table in a party, a guest always takes off a fresh plate from the top and the new plates are placed on to the stack at the top.
- It is a non-primitive data structure.
- When an element is inserted into a stack or removed from the stack, its base remains fixed where the top of stack changes.

Stack

- Insertion of element into stack is called PUSH and deletion of element from stack is called POP.
- The bellow show figure how the operations take place on a stack:



Stack

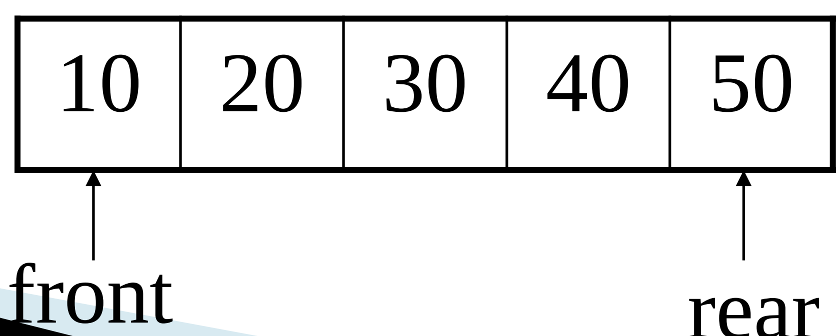
- The stack can be implemented into two ways:
  - Using arrays (Static implementation)
  - Using pointer (Dynamic implementation)

Queue

- Queue are first in first out type of data structure (i.e. FIFO)
- In a queue new elements are added to the queue from one end called REAR end and the element are always removed from other end called the FRONT end.
- The people standing in a railway reservation row are an example of queue.

Queue

- Each new person comes and stands at the end of the row and person getting their reservation confirmed get out of the row from the front end.
- The bellow show figure how the operations take place on a stack:



Queue

- The queue can be implemented into two ways:
  - Using arrays (Static implementation)
  - Using pointer (Dynamic implementation)

Trees

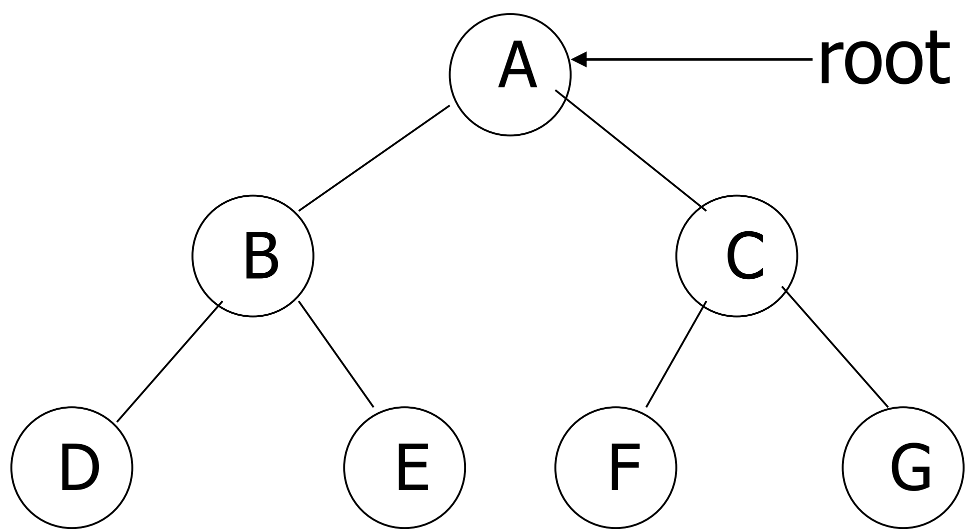
- A tree can be defined as finite set of data items (nodes).
- Tree is non-linear type of data structure in which data items are arranged or stored in a sorted sequence.
- Tree represent the hierarchical relationship between various elements.

Trees

- In trees:
- There is a special data item at the top of hierarchy called the Root of the tree.
- The remaining data items are partitioned into number of mutually exclusive subset, each of which is itself, a tree which is called the sub tree.
- The tree always grows in length towards bottom in data structures, unlike natural trees which grows upwards.

Trees

- The tree structure organizes the data into branches, which related the information.



Graph

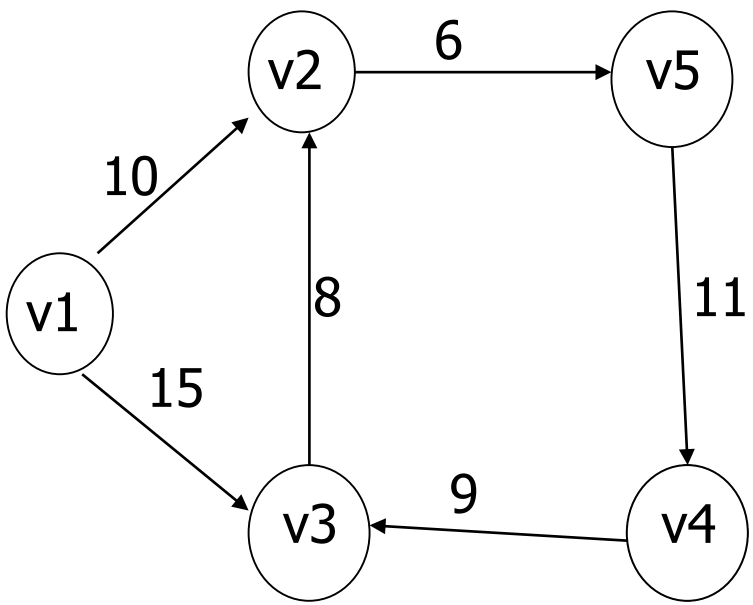
- Graph is a mathematical non-linear data structure capable of representing many kind of physical structures.
- It has found application in Geography, Chemistry and Engineering sciences.
- Definition: A graph G(V,E) is a set of vertices V and a set of edges E.

Graph

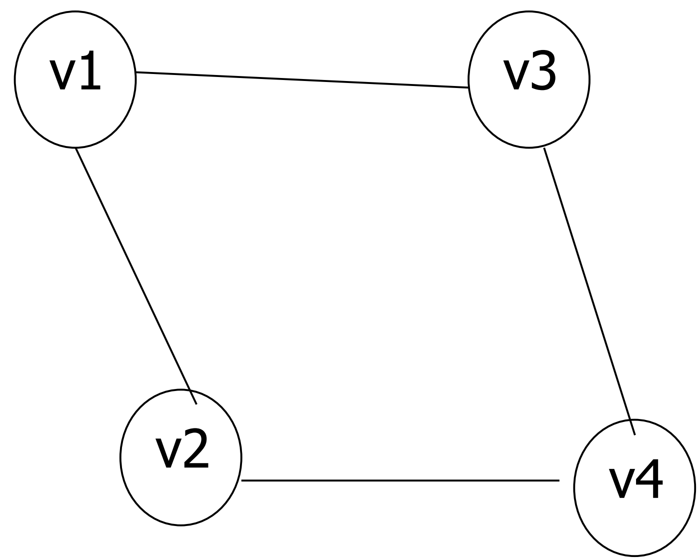
- An edge connects a pair of vertices and many have weight such as length, cost and another measuring instrument for according the graph.
- Vertices on the graph are shown as point or circles and edges are drawn as arcs or line segment.

Graph

- Example of graph:



[a] Directed & Weighted Graph



[b] Undirected Graph

Graph

- Types of Graphs:
  - Directed graph
  - Undirected graph
  - Simple graph
  - Weighted graph
  - Connected graph
  - Non-connected graph

Arrays and Structures

- The array as an abstract data type
- Structures and Unions
- The polynomial Abstract Data Type
- The Sparse Matrix Abstract Data Type
- The Representation of Multidimensional Arrays

2.1 The array as an ADT

- Arrays
  - Array: a set of pairs, <index, value>
  - data structure
    - For each index, there is a value associated with that index.
  - representation (possible)
    - Implemented by using consecutive memory.
    - In mathematical terms, we call this a *correspondence* or a *mapping*.

2.1 The array as an ADT

- When considering an ADT we are more concerned with the operations that can be performed on an array.
  - Aside from creating a new array, most languages provide only two standard operations for arrays, one that retrieves a value, and a second that stores a value.
  - Structure 2.1 shows a definition of the array ADT
  - The advantage of this ADT definition is that it clearly points out the fact that the array is a more general structure than “a consecutive set of memory locations.”

2.1 The array as an ADT

structure *Array* is

objects: A set of pairs <*index*, *value*> where for each value of *index* there is a value from the set *item*. *Index* is a finite ordered set of one or more dimensions, for example, $\{0, \cdots, n-1\}$ 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.

functions:

for all $A \in Array$, $i \in index$, $x \in item$, j, $size \in$ integer

Array Create(j, *list*)	::=	**return** an array of j dimensions where *list* is a j-tuple whose ith element is the the size of the ith dimension. *Items* are undefined.
Item Retrieve(A, i)	::=	**if** ($i \in index$) **return** the item associated with index value i in array A **else return** error
Array Store(A,i,x)	::=	**if** (i in *index*) **return** an array that is identical to array A except the new pair <i, x> has been inserted **else return** error.

end *Array*

Structure 2.1: Abstract Data Type *Array*

2.1 The array as an ADT

- Arrays in C
  - int list[5], *plist[5];
  - list[5]: (five integers) list[0], list[1], list[2], list[3], list[4]
  - *plist[5]: (five pointers to integers)
    - plist[0], plist[1], plist[2], plist[3], plist[4]
  - implementation of 1-D array

list[0]	base address = α
list[1]	α + sizeof(int)
list[2]	α + 2*sizeof(int)
list[3]	α + 3*sizeof(int)
list[4]	α + 4*sizeof(int)

2.1 The array as an ADT

- Arrays in C (cont'd)
  - Compare int *list1 and int list2[5] in C.
    Same: list1 and list2 are pointers.
    Difference: list2 reserves five locations.
  - Notations:
    list2 — a pointer to list2[0]
    (list2 + i) — a pointer to list2[i] (&list2[i])
    *(list2 + i) — list2[i]

2.1 The array

- Example:
  1-dimension array addressing
  - int one[] = {0, 1, 2, 3, 4};
  - Goal: print out address and value
    - void print1(int *ptr, int rows){

```
/* print out a one-dimensional array using a pointer */
 int i;
 printf("Address Contents\n");
 for (i=0; i < rows; i++)
          printf("%8u%5d\n", ptr+i, *(ptr+i));
 printf("\n");
}
```

2.2 Structures and Unions

- 2.2.1 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;
```

2.2 Structures and Unions

- Create structure data type
  - We can create our own structure data types by using the **typedef** statement as below:

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

- This says that human_being is the name of the type defined by the structure definition, and we may follow this definition with declarations of variables such as:

  human_being person1, person2;

2.2 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;
```

2.2 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;
```

2.2 Structures and Unions

- 2.2.3 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.

2.2 Structures and Unions

- 2.2.4 Self-Referential Structures
  - One or more of its components is a pointer to itself.
  - 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;

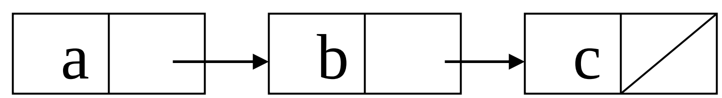
Construct a list with three nodes

item1.link=&item2;

item2.link=&item3;

malloc: obtain a node (memory)

free: release memory



2.3 The polynomial ADT

- Ordered or Linear List Examples
  - ordered (linear) list: (item1, item2, item3, …, item*n*)
    - (Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday)
    - (Ace, 2, 3, 4, 5, 6, 7, 8, 9, 10, Jack, Queen, King)
    - (basement, lobby, mezzanine, first, second)
    - (1941, 1942, 1943, 1944, 1945)
    - $(a_1, a_2, a_3, \ldots, a_{n-1}, a_n)$

2.3 The polynomial ADT

- Operations on Ordered List
  - Finding the length, n , of the list.
  - Reading the items from left to right (or right to left).
  - Retrieving the i'th element.
  - Storing a new value into the i'th position.
  - Inserting a new element at the position i , causing elements numbered i, $i+1$, …, n to become numbered $i+1$, $i+2$, …, $n+1$
  - Deleting the element at position i , causing elements numbered $i+1$, …, n to become numbered i, $i+1$, …, $n-1$
- Implementation
  - sequential mapping (1)~(4)
  - non-sequential mapping (5)~(6)

2.3 The polynomial ADT

- Polynomial examples:
  - Two example polynomials are:
    - $A(x) = 3x^{20}+2x^5+4$ and $B(x) = x^4+10x^3+3x^2+1$
  - Assume that we have two polynomials, $A(\mathrm{x}) = \Sigma a_i x^i$ and $B(x) = \Sigma b_i x^i$ where x is the variable, a_i is the coefficient, and i is the exponent, then:
    - $A(x) + B(x) = \Sigma(a^i + b^i)x^i$
    - $A(x) \cdot B(x) = \Sigma(a^i x^i \cdot \Sigma(b^j x^j))$
    - Similarly, we can define subtraction and division on polynomials, as well as many other operations.

2.3 The polynomial ADT

- An ADT definition of a polynomial

---

structure *Polynomial* is

objects: $p(x) = a_1x^{e_1} + \cdots + a_nx^{e_n}$; a set of ordered pairs of $<e_i, a_i>$ where a_i in *Coefficients* and e_i in *Exponents*, e_i are integers >= 0

functions:

for all *poly*, *poly*1, *poly*2 $\in$ *Polynomial*, *coef* $\in$ *Coefficients*, *expon* $\in$ *Exponents*

Polynomial Zero()	::=	**return** the polynomial, $p(x) = 0$
Boolean IsZero(*poly*)	::=	**if** (*poly*) **return** *FALSE* **else return** *TRUE*
Coefficient Coef(*poly*,*expon*)	::=	**if** (*expon* $\in$ *poly*) **return** its coefficient **else return** zero
Exponent Lead_Exp(*poly*)	::=	**return** the largest exponent in *poly*
Polynomial Attach(*poly*, *coef*, *expon*)	::=	**if** (*expon* $\in$ *poly*) **return** error **else return** the polynomial *poly* with the term <*coef*, *expon*> inserted
Polynomial Remove(*poly*, *expon*)	::=	**if** (*expon* $\in$ *poly*) **return** the polynomial *poly* with the term whose exponent is *expon* deleted **else return** error
Polynomial SingleMult(*poly*, *coef*, *expon*)	::=	**return** the polynomial $poly \cdot coef \cdot x^{expon}$
Polynomial Add(*poly*1, *poly*2)	::=	**return** the polynomial *poly*1 + *poly*2
Polynomial Mult(*poly*1, *poly*2)	::=	**return** the polynomial *poly*1 $\cdot$ *poly*2

end *Polynomial*

Structure 2.2: Abstract data type *Polynomial*

2.3 The polynomial ADT

- There are two ways to create the type *polynomial* in C
- Representation I
  - define MAX_degree 101 /*MAX degree of polynomial+1*/

```
typedef struct{
    int degree;
    float coef [MAX_degree];
}polynomial;
```

drawback: the first representation may waste space.

Polynomial Addition

```
/* d =a + b, where a, b, and d are polynomials */
d = Zero( )
while (! IsZero(a) && ! IsZero(b)) do {
   switch COMPARE (Lead_Exp(a), Lead_Exp(b))  {
       case -1: d =
           Attach(d, Coef (b, Lead_Exp(b)), Lead_Exp(b));
           b = Remove(b, Lead_Exp(b));
           break;
      case  0: sum = Coef (a, Lead_Exp (a)) + Coef ( b, Lead_Exp(b));
          if (sum) {
             Attach (d, sum, Lead_Exp(a));
           }
           a = Remove(a , Lead_Exp(a));
           b = Remove(b , Lead_Exp(b));
           break;
      case 1: d =
           Attach(d, Coef (a, Lead_Exp(a)), Lead_Exp(a));
           a = Remove(a, Lead_Exp(a));
   }
}
insert any remaining terms of a or b into d
```

advantage: easy implementation

disadvantage: waste space when sparse

<u>***Program 2.4** :Initial version of *padd* function(p.62)</u>

2.3 The polynomial ADT

- Representation II
  - MAX_TERMS 100 /*size of terms array*/

```
typedef struct{
    float coef;
    int expon;
}polynomial;
polynomial terms [MAX_TERMS];
int avail = 0;
```

2.3 The polynomial ADT

- Use one global array to store all polynomials
  - Figure 2.2 shows how these polynomials are stored in the array *terms*.

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

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

specification	representation
poly	<start, finish>
A	<0,1>
B	<2,5>

storage requirements: start, finish, 2*(finish-start+1)

non-sparse: twice as much as Representation I when all the items are nonzero

	starta ↓	*finisha* ↓	*startb* ↓			*finishb* ↓	*avail* ↓
coef	2	1	1	10	3	1	
exp	1000	0	4	3	2	0	
	0	1	2	3	4	5	6

Figure 2.2: Array representation of two polynomials

2.3 The polynomial ADT

- We would now like to write a C function that adds two polynomials, *A* and *B*, represented as above to obtain *D* = *A* + *B*.
  - To produce *D*(*x*), *padd* (Program 2.5) adds *A*(*x*) and *B*(*x*) term by term.

Analysis: $O(n+m)$ where n (m) is the number of nonzeros in A (B).

```
void padd(int starta,int finisha,int startb, int finishb,
                                int *startd,int *finishd)
{
/* add A(x) and B(x) to obtain D(x) */
  float coefficient;
  *startd = avail;
  while (starta <= finisha && startb <= finishb)
    switch(COMPARE(terms[starta].expon,
                   terms[startb].expon)) {
      case -1: /* a expon < b expon */
           attach(terms[startb].coef,terms[startb].expon)
           startb++;
           break;
      case 0: /* equal exponents */
           coefficient = terms[starta].coef +
                         terms[startb].coef;
           if (coefficient)
             attach(coefficient,terms[starta].expon);
           starta++;
           startb++;
           break;
      case 1: /* a expon > b expon */
           attach(terms[starta].coef,terms[starta].expon)
           starta++;
    }
  /* add in remaining terms of A(x) */
  for(; starta <= finisha; starta++)
    attach(terms[starta].coef,terms[starta].expon);
  /* add in remaining terms of B(x) */
  for( ; startb <= finishb; startb++)
    attach(terms[startb].coef, terms[startb].expon);
  *finishd = avail-1;
}
```

Program 2.5: Function to add two polynomials

2.3 The polynomial ADT

```
void attach(float coefficient, int exponent)
{
/* add a new term to the polynomial */
   if (avail >= MAX_TERMS) {
      fprintf(stderr,"Too many terms in the polynomial\n");
      exit(1);
   }
   terms[avail].coef = coefficient;
   terms[avail++].expon = exponent;
}
```

Program 2.6: Function to add a new term

Problem: Compaction is required
when polynomials that are no longer needed.
(data movement takes time.)

2.4 The sparse matrix ADT

2.4.1 Introduction

- In mathematics, a matrix contains m rows and n columns of elements, we write $m \times n$ to designate a matrix with m rows and n columns.

$$
\begin{array}{c}
 \\ \text{row 0} \\ \text{row 1} \\ \text{row 2} \\ \text{row 3} \\ \text{row 4}
\end{array}
\begin{array}{c}
\begin{array}{ccc} \text{col 0} & \text{col 1} & \text{col 2} \end{array} \\
\begin{bmatrix}
-27 & 3 & 4 \\
6 & 82 & -2 \\
109 & -64 & 11 \\
12 & 8 & 9 \\
48 & 27 & 47
\end{bmatrix}
\end{array}
$$

5*3

(a) 15/15

$$
\begin{array}{c}
 \\ \text{row 0} \\ \text{row 1} \\ \text{row 2} \\ \text{row 3} \\ \text{row 4} \\ \text{row 5}
\end{array}
\begin{array}{c}
\begin{array}{cccccc} \text{col 0} & \text{col 1} & \text{col 2} & \text{col 3} & \text{col 4} & \text{col 5} \end{array} \\
\begin{bmatrix}
15 & 0 & 0 & 22 & 0 & -15 \\
0 & 11 & 3 & 0 & 0 & 0 \\
0 & 0 & 0 & -6 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
91 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 28 & 0 & 0 & 0
\end{bmatrix}
\end{array}
$$

sparse matrix

data structure?

6*6

(b) 8/36

Figure 2.3: Two matrices

2.4 The sparse matrix ADT

- The standard representation of a matrix is a two dimensional array defined as *a*[*MAX_ROWS*][*MAX_COLS*].
  - We can locate quickly any element by writing *a*[*i*][*j*]
- Sparse matrix wastes space
  - We must consider alternate forms of representation.
  - Our representation of sparse matrices should store only nonzero elements.
  - Each element is characterized by <row, col, value>.

2.4 The sparse matrix ADT

- Structure 2.3 contains our specification of the matrix ADT.
  - A minimal set of operations includes matrix creation, addition, multiplication, and transpose.

---

structure *Sparse_Matrix* is

objects: a set of triples, <*row*, *column*, *value*>, where *row* and *column* are integers and form a unique combination, and *value* comes from the set *item*.

functions:

for all $a, b \in$ *Sparse_Matrix*, $x \in$ *item*, *i*, *j*, *max_col*, *max_row* $\in$ *index*

Sparse_Matrix Create(*max_row*, *max_col*) ::=
: **return** a *Sparse_Matrix* that can hold up to *max_items* = *max_row* × *max_col* and whose maximum row size is *max_row* and whose maximum column size is *max_col*.

Sparse_Matrix Transpose(*a*) ::=
: **return** the matrix produced by interchanging the row and column value of every triple.

Sparse_Matrix Add(*a*, *b*) ::=
: **if** the dimensions of *a* and *b* are the same **return** the matrix produced by adding corresponding items, namely those with identical *row* and *column* values.
: **else return** error

Sparse_Matrix Multiply(*a*, *b*) ::=
: **if** number of columns in *a* equals number of rows in *b*
: **return** the matrix *d* produced by multiplying *a* by *b* according to the formula: $d[i][j] = \sum(a[i][k] \cdot b[k][j])$ where $d(i, j)$ is the (i, j)th element
: **else return** error.

Structure 2.3: Abstract data type *Sparse_Matrix*

2.4 The sparse matrix ADT

- We implement the *Create* operation as below:

Sparse_Matrix Create(*max_row*, *max_col*) ::=

```
#define MAX_TERMS 101 /* maximum number of terms +1*/
typedef struct {
        int col;
        int row;
        int value;
        } term;
term a[MAX_TERMS];
```

2.4 The sparse matrix ADT

- Figure 2.4(a) shows how the sparse matrix of Figure 2.3(b) is represented in the array *a*.
  - Represented by a two-dimensional array.
  - Each element is characterized by <row, col, value>.

# of rows (columns)

# of nonzero terms

	row	col	value
a[0]	6	6	8
[1]	0	0	15
[2]	0	3	22
[3]	0	5	−15
[4]	1	1	11
[5]	1	2	3
[6]	2	3	−6
[7]	4	0	91
[8]	5	2	28

(a)

transpose

	row	col	value
b[0]	6	6	8
[1]	0	0	15
[2]	0	4	91
[3]	1	1	11
[4]	2	1	3
[5]	2	5	28
[6]	3	0	22
[7]	3	2	−6
[8]	5	0	−15

(b)

row, column in ascending order

Figure 2.4: Sparse matrix and its transpose stored as triples

2.4 The sparse matrix ADT

- 2.4.2 Transpose a Matrix
  - For each row i
    - take element <i, j, value> and store it in element <j, i, value> of the transpose.
    - difficulty: where to put <j, i, value>
      (0, 0, 15) ====> (0, 0, 15)
      (0, 3, 22) ====> (3, 0, 22)
      (0, 5, -15) ====> (5, 0, -15)
      (1, 1, 11) ====> (1, 1, 11)
      Move elements down very often.
  - For all elements in column j,
    place element <i, j, value> in element <j, i, value>

2.4 The sparse matrix ADT

- This algorithm is incorporated in transpose (Program 2.7)

```
void transpose(term a[], term b[])
/* b is set to the transpose of a */
{
   int n,i,j, currentb;
   n = a[0].value;          /* total number of elements */
   b[0].row = a[0].col; /* rows in b = columns in a */
   b[0].col = a[0].row; /* columns in b = rows in a */
   b[0].value = n;
   if (n > 0 )  { /* non zero matrix */
      currentb = 1;
      for (i = 0; i < a[0].col; i++)
      /* transpose by the columns in a */
         for (j = 1; j <= n; j++)
         /* find elements from the current column */
            if (a[j].col == i) {
            /* element is in current column, add it to b */
               b[currentb].row = a[j].col;
               b[currentb].col = a[j].row;
               b[currentb].value = a[j].value;
               currentb++;
            }
   }
}
```

Program 2.7: Transpose of a sparse matrix

columns

elements

Scan the array "columns" times.
The array has "elements" elements.

==> O(columns*elements)

2.4 The sparse matrix ADT

- Discussion: compared with 2-D array representation
  - O(columns*elements) vs. O(columns*rows)
  - elements --> columns * rows when non-sparse, O(columns^2*rows)
- Problem: Scan the array “columns” times.
  - In fact, we can transpose a matrix represented as a sequence of triples in O(columns + elements) time.
- Solution:
  - First, determine the number of elements in each column of the original matrix.
  - Second, determine the starting positions of each row in the transpose matrix.

2.4 The sparse matrix ADT

- Compared with 2-D array representation:
  O(columns+elements) vs. O(columns*rows)
  elements --> columns * rows O(columns*rows)

Cost:
Additional row_terms and starting_pos arrays are required.
Let the two arrays row_terms and starting_pos be shared.

```
void fast_transpose(term a[], term b[])
{
/* the transpose of a is placed in b */
   int row_terms[MAX_COL], starting_pos[MAX_COL];
   int i,j, num_cols = a[0].col, num_terms = a[0].value;
   b[0].row = num_cols;  b[0].col = a[0].row;
   b[0].value = num_terms;
   if (num_terms > 0) { /* nonzero matrix */
      for (i = 0; i < num_cols; i++)
         row_terms[i] = 0;
      for (i = 1; i <= num_terms; i++)
         row_terms[a[i].col]++;
      starting_pos[0] = 1;
      for (i = 1; i < num_cols; i++)
         starting_pos[i] =
                     starting_pos[i-1] + row_terms[i-1];
      for (i = 1; i <= num_terms; i++) {
         j = starting_pos[a[i].col]++;
         b[j].row = a[i].col;   b[j].col = a[i].row;
         b[j].value = a[i].value;
      }
   }
}
```

columns
elements
columns
elements

Program 2.8: Fast transpose of a sparse matrix

2.4 The sparse matrix ADT

- After the execution of the third **for** loop, the values of *row_terms* and *starting_pos* are:

	[0]	[1]	[2]	[3]	[4]	[5]
row_terms =	2	1	2	2	0	1
starting_pos =	1	3	4	6	8	8

	row	col	value
a[0]	6	6	8
[1]	0	0	15
[2]	0	3	22
[3]	0	5	−15
[4]	1	1	11
[5]	1	2	3
[6]	2	3	−6
[7]	4	0	91
[8]	5	2	28

(a)

transpose

	row	col	value
b[0]	6	6	8
[1]	0	0	15
[2]	0	4	91
[3]	1	1	11
[4]	2	1	3
[5]	2	5	28
[6]	3	0	22
[7]	3	2	−6
[8]	5	0	−15

(b)

Figure 2.4: Sparse matrix and its transpose stored as triples

2.4 The sparse matrix ADT

- 2.4.3 Matrix multiplication
  - Definition:

    Given A and B where A is $m \times n$ and B is $n \times p$, the product matrix D has dimension $m \times p$. Its $< i, j >$ element is

    for $0 \leq i < m$ $d_{ij} = \sum_{k=0}^{n-1} a_{ik} b_{kj}$ p.
  - Example:

$$\begin{bmatrix} 1 & 0 & 0 \\ 1 & 0 & 0 \\ 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

Figure 2.5: Multiplication of two sparse matrices

2.4 The sparse matrix ADT

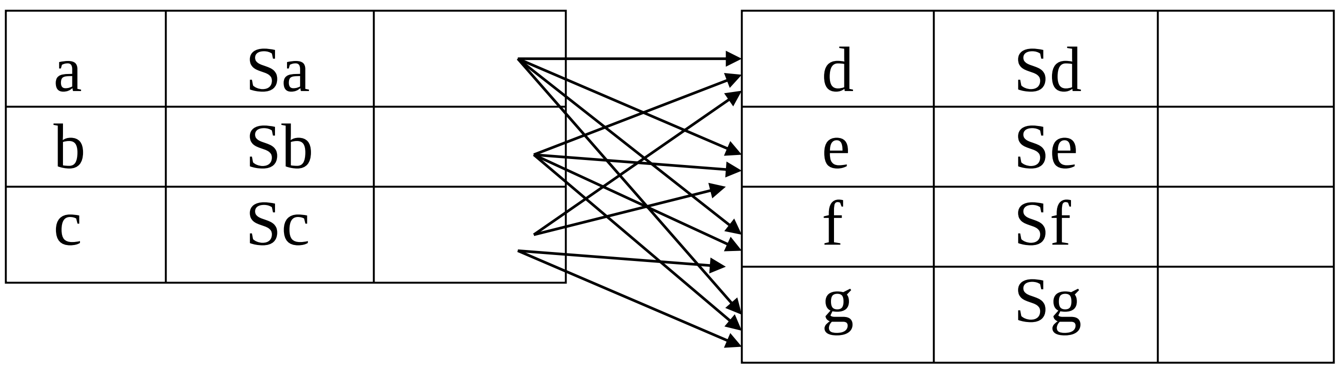
- Sparse Matrix Multiplication
  - Definition: $[D]_{m*p}=[A]_{m*n}*[B]_{n*p}$
  - Procedure: Fix a row of A and find all elements in column j of B for $j=0, 1, \ldots, p-1$.
  - Alternative 1.
    Scan all of B to find all elements in j.
  - Alternative 2.
    Compute the transpose of B.
    (Put all column elements consecutively)
    - Once we have located the elements of row i of A and column j of B we just do a merge operation similar to that used in the polynomial addition of 2.2

2.4 The sparse matrix ADT

- General case:

$$d_{ij}=a_{i0}*b_{0j}+a_{i1}*b_{1j}+\ldots+a_{i(n-1)}*b_{(n-1)j}$$

  - Array A is grouped by i, and after transpose, array B is also grouped by j



The generation at most:
entries ad, ae, af, ag, bd, be, bf, bg, cd, ce, cf, cg

The sparse matrix ADT

- An Example

$$A = \begin{bmatrix} 1 & 0 & 2 \\ -1 & 4 & 6 \end{bmatrix} \quad B^T = \begin{bmatrix} 3 & -1 & 0 \\ 0 & 0 & 0 \\ 2 & 0 & 5 \end{bmatrix} \quad B = \begin{bmatrix} 3 & 0 & 2 \\ -1 & 0 & 0 \\ 0 & 0 & 5 \end{bmatrix}$$

	row	col	value
a[0]	2	3	5
[1]	0	0	1
[2]	0	2	2
[3]	1	0	-1
[4]	1	1	4
[5]	1	2	6

	row	col	value
b_t[0]	3	3	4
b_t[1]	0	0	3
b_t[2]	0	1	-1
b_t[3]	2	0	2
b_t[4]	2	2	5

	row	col	value
b[0]	3	3	4
b[1]	0	0	3
b[2]	0	2	2
b[3]	1	0	-1
b[4]	2	2	5

2.4 The sparse matrix ADT

- The programs 2.9 and 2.10 can obtain the product matrix *D* which multiplies matrices *A* and *B*.

```
void mmult(term a[], term b[], term d[])
/* multiply two sparse matrices */
{
   int i, j, column, totalb = b[0].value, totald = 0;
   int rows_a = a[0].row, cols_a = a[0].col,
   totala = a[0].value; int  cols_b = b[0].col,
   int row_begin = 1, row = a[1].row, sum = 0;
   int new_b[MAX_TERMS][3];
   if (cols_a != b[0].row) {
      fprintf(stderr,"Incompatible matrices\n");
      exit(1);
   }
   fast_transpose(b,new_b);
   /* set boundary condition */
   a[totala+1].row = rows_a;
   new_b[totalb+1].row = cols_b;
   new_b[totalb+1].col = 0;
   for (i = 1; i <= totala; ) {
      column = new_b[1].row;
      for (j = 1; j <= totalb+1;) {
      /* multiply row of a by column of b */
         if (a[i].row != row) {
            storesum(d,&totald,row,column,&sum);
            i = row_begin;
            for (; new_b[j].row == column; j++)
               ;
            column = new_b[j].row;
         }
         else if (new_b[j].row != column) {
            storesum(d, &totald, row, column, &sum);
            i = row_begin;
            column = new_b[j].row;
         }
         else switch (COMPARE(a[i].col, new_b[j].col)) {
            case -1: /* go to next term in a */
                  i++;  break;
            case 0: /* add terms, go to next term in a and b*/
                  sum += ( a[i++].value * new_b[j++].value);
                  break;
            case 1 : /* advance to next term in b */
                  j++;
         }
      } /* end of for j <= totalb+1 */
      for (; a[i].row == row; i++)
         ;
      row_begin = i; row = a[i].row;
   } /* end of for i<=totala */
   d[0].row = rows_a;
   d[0].col = cols_b; d[0].value =  totald;
}
```

a × b

Program 2.9: Sparse matrix multiplication

2.4 The sparse matrix ADT

```
void storesum(term d[], int *totald, int row, int column,
                                              int *sum)
{
/* if *sum != 0, then it along with its row and column
position is stored as the *totald+1 entry in d */
   if (*sum)
      if (*totald < MAX_TERMS) {
         d[++*totald].row = row;
         d[*totald].col = column;
         d[*totald].value = *sum;
         *sum = 0;
      }
      else {
         fprintf(stderr,"Numbers of terms in product
                                    exceeds %d\n",MAX_TERMS);
         exit(1);
      }
}
```

Program 2.10: *storesum* function

2.4 The sparse matrix ADT

- Analyzing the algorithm
  - cols_b * termsrow1 + totalb +
    cols_b * termsrow2 + totalb +
    … +
    cols_b * termsrowp + totalb
    = cols_b * (termsrow1 + termsrow2 + … + termsrowp)+
    rows_a * totalb
    = cols_b * totala + row_a * totalb

    O(cols_b * totala + rows_a * totalb)

2.4 The sparse matrix ADT

- Compared with matrix multiplication using array
  - ```
    for (i =0; i < rows_a; i++)
        for (j=0; j < cols_b; j++) {
            sum =0;
            for (k=0; k < cols_a; k++)
                sum += (a[i][k] *b[k][j]);
            d[i][j] =sum;
        }
    ```
  - O(rows_a * cols_a * cols_b) vs.
    O(cols_b * total_a + rows_a * total_b)
  - optimal case:
    total_a < rows_a * cols_a total_b < cols_a * cols_b
  - worse case:
    total_a --> rows_a * cols_a, or
    total_b --> cols_a * cols_b
    ```

2.5 Representation of multidimensional array

- The internal representation of multidimensional arrays requires more complex addressing formula.
  - If an array is declared $a[upper_0][upper_1]\ldots[upper_n]$, then it is easy to see that the number of elements in $\prod_{i=0}^{n-1} upper_i$ is:

    Where Π is the product of the $upper_i$'s.
  - Example:
    - If we declare a as $a[10][10][10]$, then we require 10*10*10 = 1000 units of storage to hold the array.

2.5 Representation of multidimensional array

- Represent multidimensional arrays: *row major order* and *column major order*.
  - Row major order stores multidimensional arrays by rows.
    - A[$upper_0$][$upper_1$] as $upper_0$ rows, row_0, row_1, …, $row_{upper0-1}$, each row containing *upper1* elements.

2.5 Representation of multidimensional array

- Row major order: A[i][j] : $\alpha + i * upper_1 + j$
- Column major order: A[i][j] : $\alpha + j * upper_0 + i$

	col_0	col_1		col_{u1-1}
row_0	A[0][0] α	A[0][1] $\alpha + u_0$	. . .	A[0][u1-1] $\alpha+(u_1-1)* u_0$
row_1	A[1][0] $\alpha + u_1$	A[1][1]	. . .	A[1][u1-1]
		. . .		
row_{u0-1}	A[u0-1][0] $\alpha+(u_0-1)*u_1$	A[u0-1][1]	. . .	A[u0-1][u1-1]

2.5 Representation of multidimensional array

- To represent a three-dimensional array, $A[upper_0][upper_1][upper_2]$, we interpret the array as $upper_0$ two-dimensional arrays of dimension $upper_1 \times upper_2$.
  - To locate $a[i][j][k]$, we first obtain $\alpha + i*upper_1*upper_2$ as the address of $a[i][0][0]$ because there are i two dimensional arrays of size $upper_1*upper_2$ preceding this element.
  - $\alpha + i*upper_1*upper_2 + j*upper_2 + k$ as the address of $a[i][j][k]$.

2.5 Representation of multidimensional array

- Generalizing on the preceding discussion, we can obtain the addressing formula for any element $A[i_0][i_1]\ldots[i_{n-1}]$ in an n–dimensional array declared as: $A[upper_0][upper_1]\ldots[upper_{n-1}]$
  - The address for $A[i_0][i_1]\ldots[i_{n-1}]$ is:

$$
\begin{aligned}
&\alpha + i_0 upper_1 upper_2 \ldots upper_{n-1} \\
&\quad + i_1 upper_2 upper_3 \ldots upper_{n-1} \\
&\quad + i_2 upper_3 upper_4 \ldots upper_{n-1} \\
&\quad \cdot \\
&\quad \cdot \\
&\quad \cdot \\
&\quad + i_{n-2} upper_{n-1} \qquad = \alpha + \sum_{j=0}^{n-1} i_j a_j \\
&\quad + i_{n-1}
\end{aligned}
$$

$$
\text{where: } \begin{cases} a_j = \prod_{k=j+1}^{n-1} upper_k & 0 \le j < n-1 \\ a_{n-1} = 1 \end{cases}
$$

2.6 The String Abstract data type

2.6.1 Introduction

- The String: component elements are characters.
  - A string to have the form, $S = s_0, \ldots, s_{n-1}$, where s_i are characters taken from the character set of the programming language.
  - If $n = 0$, then S is an empty or null string.
  - Operations in ADT 2.4, p. 81

2.6 The String Abstract data type

- ADT *String*:

structure *String* **is**

objects: a finite set of zero or more characters.

functions:

for all $s, t \in$ *String*, $i, j, m \in$ non-negative integers

String Null(m)	::=	**return** a string whose maximum length is m characters, but is initially set to *NULL* We write *NULL* as "".
Integer Compare(s, t)	::=	**if** s equals t **return** 0 **else if** s precedes t **return** -1 **else return** $+1$
Boolean IsNull(s)	::=	**if** (Compare(s, *NULL*)) **return** *FALSE* **else return** *TRUE*
Integer Length(s)	::=	**if** (Compare(s, *NULL*)) **return** the number of characters in s **else return** 0.
String Concat(s, t)	::=	**if** (Compare(t, *NULL*)) **return** a string whose elements are those of s followed by those of t **else return** s.
String Substr(s, i, j)	::=	**if** $((j > 0)$ && $(i+j-1) <$ Length$(s))$ **return** the string containing the characters of s at positions $i, i+1, \cdots, i+j-1$. **else return** *NULL*.

Structure 2.4: Abstract data type *String*

2.6 The String Abstract data type

- In C, we represent strings as character arrays terminated with the null character \0.

For instance, suppose we had the strings:

```
#define MAX_SIZE 100 /*maximum size of string */
char s[MAX_SIZE] = {"dog"};
char t[MAX_SIZE] = {"house"};
```

- Figure 2.8 shows how these strings would be represented internally in memory.

s[0]	s[1]	s[2]	s[3]
d	o	g	\0

t[0]	t[1]	t[2]	t[3]	t[4]	t[5]
h	o	u	s	e	\0

Figure 2.8: String representation in C

2.6 The String Abstract data type

- Now suppose we want to concatenate these strings together to produce the new string:
  - Two strings are joined together by ***strcat(s, t),*** which stores the result in ***s***. Although s has increased in length by five, we have no additional space in ***s*** to store the extra five characters. Our compiler handled this problem inelegantly: it simply overwrote the memory to fit in the extra five characters. Since we declared ***t*** immediately after ***s***, this meant that part of the word "house" disappeared.

2.6 The String Abstract data type

- C string functior

Function	Description
*char *strcat(char *dest, char *src)*	concatenate *dest* and *src* strings; return result in *dest*
*char *strncat(char *dest, char *src, int n)*	concatenate *dest* and *n* characters from *src*; return result in *dest*
*char *strcmp(char *str1, char *str2)*	compare two strings; return < 0 if $str1 < str2$; 0 if $str1 = str2$; > 0 if $str1 > str2$
*char *strncmp(char *str1, char *str2, int n)*	compare first *n* characters return < 0 if $str1 < str2$; 0 if $str1 = str2$; > 1 if $str1 > str2$
*char *strcpy(char *dest, char *src)*	copy *src* into *dest*; return *dest*
*char *strncpy(char *dest, char *src, int n)*	copy *n* characters from *src* string into *dest*; return *dest*;
*size_t strlen(char *s)*	return the length of a *s*
*char *strchr(char *s, int c)*	return pointer to the first occurrence of *c* in *s*; return *NULL* if not present
*char *strrchr(char *s, int c)*	return pointer to last occurrence of *c* in *s*; return *NULL* if not present
*char *strtok(char *s, char *delimiters)*	return a token from *s*; token is surrounded by *delimiters*
*char *strstr(char *s, char *pat)*	return pointer to start of *pat* in *s*
*size_t strspn(char *s, char *spanset)*	scan *s* for characters in *spanset*; return length of span
*size_t strcspn(char *s, char *spanset)*	scan *s* for characters not in *spanset*; return length of span
*char *strpbrk(char *s, char *spanset)*	scan *s* for characters in *spanset*; return pointer to first occurrence of a character from *spanset*

Figure 2.7: C string functions

2.6 The String Abstract data type

- Example 2.2[String insertion]:
  - Assume that we have two strings, say *string* 1 and *string* 2, and that we want to insert *string* 2 into *string* 1 starting at the *i* th position of *string* 1. We begin with the declarations:
  - In addition to creating the two strings, we also have created a pointer for each string.

```
#include <string.h>
#define MAX_SIZE 100 /*size of largest string*/
char string1[MAX_SIZE], *s = string1;
char string2[MAX_SIZE],  *t = string2;
```

2.6 The String Abstract data type

- Now suppose that the first string contains "amobile" and the second contains "uto".
  - we want to insert "uto" starting at position 1 of the first string, thereb producing the word "automobile.'

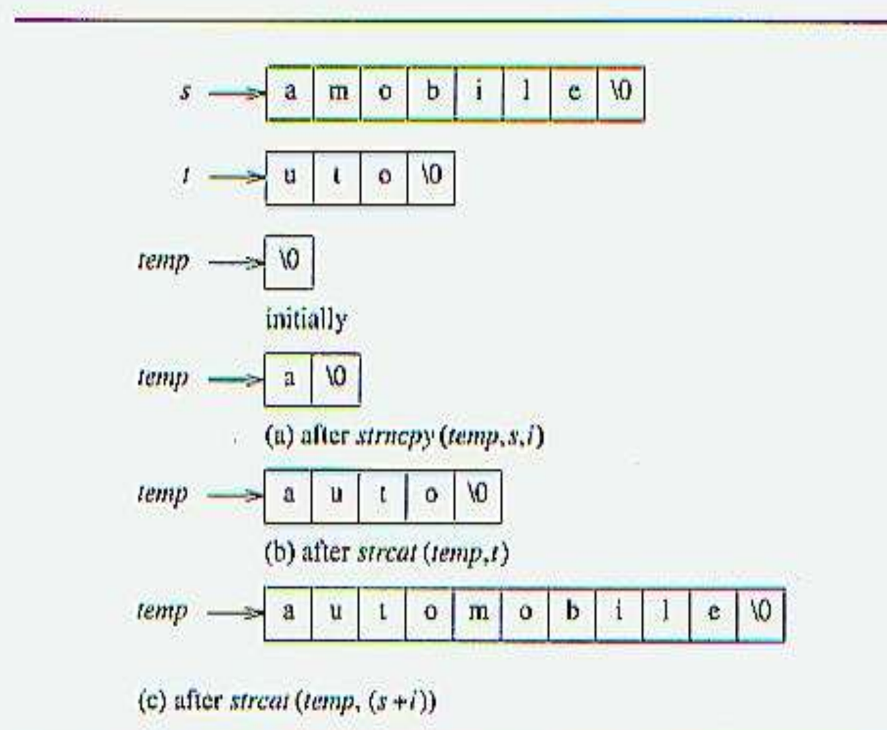


Figure 2.9: String insertion example

2.6 The String Abstract data type

- String insertion function:
  - It should never be used in practice as it is wasteful in its use of time and space.

```
void strnins(char *s, char *t, int i)
{
/* insert string t into string s at position i */
  char string[MAX_SIZE], *temp = string;

  if (i < 0 && i > strlen(s)) {
     fprintf(stderr,"Position is out of bounds \n");
     exit(1);
  }
  if (!strlen(s))
     strcpy(s,t);
  else if (strlen(t)) {
     strncpy(temp, s,i);
     strcat(temp,t);
     strcat(temp, (s+i));
     strcpy(s, temp);
  }
}
```

Program 2.11: String insertion function

2.6 The String Abstract data type

- 2.6.2 Pattern Matching:
  - Assume that we have two strings, *string* and *pat* where *pat* is a pattern to be searched for in *string*.
  - If we have the following declarations:

```
char pat[MAX_SIZE], string[MAX_SIZE], *t;
```

  - Then we use the following statements to determine if *pat* is in *string*:

```
if (t = strstr(string,pat))
  printf("The string from strstr is: %s\n",t);
else
  printf("The pattern was not found with strstr\n");
```

  - If *pat* is not in *string*, this method has a computing time of O(*n***m*) where *n* is the length of *pat* and *m* is the length of *string*.

2.6 The String Abstract data type

- We can improve on an exhaustive pattern matching technique by quitting when *strlen*(*pat*) is greater than the number of remaining characters in the string.

```
int nfind(char *string, char *pat)
{
/* match the last character of pattern first, and
then match from the beginning */
   int i,j,start = 0;
   int lasts = strlen(string)-1;
   int lastp = strlen(pat)-1;
   int endmatch = lastp;

   for (i = 0; endmatch <= lasts; endmatch++, start++) {
      if (string[endmatch] == pat[lastp])
         for (j = 0, i = start; j < lastp &&
                        string[i] == pat[j]; i++,j++)
         ;
      if (j == lastp)
         return start; /* successful */
   }
   return -1;
}
```

Program 2.12: Pattern matching by checking end indices first

2.6 The String Abstract data type

- Example 2.3 [Simulation of *nfind*]
  - Suppose *pat*="aab" and *string*="ababbaabaa."
  - Analysis of *nfind*: The computing time fo these string is linear in the length of the string O(*m*), but the Worst case is still O(*n.m*).

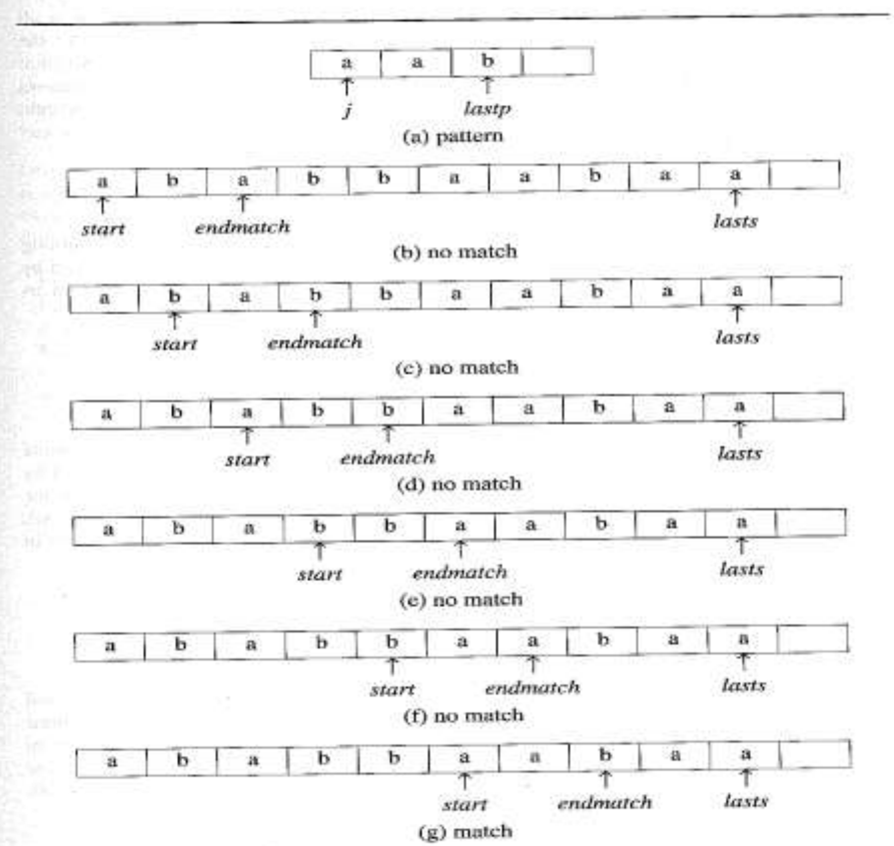


Figure 2.10: Simulation of *nfind*

2.6 The String Abstract data type

- Ideally, we would like an algorithm that works in

  $O(strlen(string)+strlen(pat))$ time.This is optimal for this problem as in the worst case it is necessary to look at all characters in the pattern and string at least once.
- Knuth,Morris, and Pratt have developed a pattern matching algorithm that works in this way and has linear complexity.

2.6 The String Abstract data type

- Suppose pat = "a b c a b c a c a b"

Let $s = s_0 s_2 \cdots s_{m-1}$ be the string and assume that we are currently determining whether or not there is a match beginning at s_i. If $s_i \neq a$ then, clearly, we may proceed by comparing s_{i+1} and a. Similarly if $s_i = a$ and $s_{i+1} \neq b$ then we may proceed by comparing s_{i+1} and a. If $s_i s_{i+1} = ab$ and $s_{i+2} \neq c$ then we have the situation:

$s =$	'-	a	b	?	?	?	.	.	.	.	?'
$pat =$		'a	b	c	a	b	c	a	c	a	b'

The ? implies that we do not know what the character in s is. The first ? in s represents s_{i+2} and $s_{i+2} \neq c$. At this point we know that we may continue the search for a match by comparing the first character in *pat* with s_{i+2}. There is no need to compare this character of *pat* with s_{i+1} as we already know that s_{i+1} is the same as the second character of *pat*, b, and so $s_{i+1} \neq a$. Let us try this again assuming a match of the first four characters in *pat* followed by a nonmatch, i.e., $s_{i+4} \neq b$. We now have the situation:

$s =$	'-	a	b	c	a	?	?	.	.	.	?'
$pat =$		'a	b	c	a	b	c	a	c	a	b'

We observe that the search for a match can proceed by comparing s_{i+4} and the second character in *pat*, b. This is the first place a partial match can occur by sliding the pattern *pat* towards the right. Thus, by knowing the characters in the pattern and the position in the pattern where a mismatch occurs with a character in s we can determine where in the pattern to continue the search for a match without moving backwards in s. To formalize this, we define a failure function for a pattern.

2.6 The String Abstract data type(14/19)

Definition: If $p = p_0 p_1 \cdots p_{n-1}$ is a pattern, then its *failure function*, f, is defined as:

$$f(j) = \begin{cases} \text{largest } i < j \text{ such that } p_0 p_1 \cdots p_i = p_{j-i} p_{j-i+2} \cdots p_j & \text{if such an } i \geq 0 \text{ exists} \\ -1 & \text{otherwise} \end{cases} \quad \square$$

For the example pattern, $pat = abcabcacab$, we have:

j	0	1	2	3	4	5	6	7	8	9
pat	a	b	c	a	b	c	a	c	a	b
f	−1	−1	−1	0	1	2	3	−1	0	1

- From the definition of the failure function, we arrive at the following rule for pattern matching: if a partial match is found such that *Si−j…Si−1=P0P1…Pj−1* and S*i != Pj then matching may be resumed by comparing Si and Pf(j−1)+1 if j != 0 .If j= 0, then we may continue by comparing Si+1 and P0.*

2.6 The String Abstract data type

- This pattern matching rule translates into function *pmatch*.

```
#include <stdio.h>
#include <string.h>
#define max_string_size 100
#define max_pattern_size 100
int pmatch();
void fail();
int failure[max_pattern_size];
char string[max_string_size];
char pat[max_pattern_size];
```

```
int pmatch(char *string, char *pat)
{
/* Knuth, Morris, Pratt string matching algorithm */
  int i = 0, j = 0;
  int lens = strlen(string);
  int lenp = strlen(pat);
  while ( i < lens && j < lenp ) {
    if (string[i] == pat[j]) {
      i++; j++; }
    else if (j == 0) i++;
         else j = failure[j-1]+1;
  }
  return ( (j == lenp) ? (i-lenp) : -1);
}
```

Program 2.13: Knuth, Morris, Pratt pattern matching algorithm

2.6 The String Abstract data type

- Analysis of *pmatch*:
  - The **while** loop is iterated until the end of either the string or the pattern is reached. Since *i* is never decreased, the lines that increase *i* cannot be executed more than $m = strlen(string)$ times. The resetting of *j* to *failure*[j-1]+1 decreases *j++* as otherwise, *j* falls off the pattern. Each time the statement *j++* is executed, *i* is also incremented. So *j* cannot be incremented more than m times. Hence the complexity of function *pmatch* is $O(m) = O(strlen(string))$.

2.6 The String Abstract data type

- If we can compute the failure function in O(*strlen*(*pat*)) time, then the entire pattern matching process will have a computing time proportional to the sum of the lengths of the string and pattern. Fortunately, there is a fast way to compute the failure function. This is based upon the following restatement of the failure function:

$$f(j) = \begin{cases} -1 & \text{if } j = 0 \\ f^m(j-1)+1 & \text{where } m \text{ is the least integer } k \text{ for which } p_{f^k(j-1)+1} = p_j \\ -1 & \text{if there is no } k \text{ satisfying the above} \end{cases}$$

(note that $f^1(j) = f(j)$ and $f^m(j) = f(f^{m-1}(j))$).

2.6 The String Abstract data type

```
void fail(char *pat)
{
/* compute the pattern's failure function */
  int n = strlen(pat);
  failure[0] = -1;
  for (j=1; j < n; j++) {
    i = failure[j-1];
    while ((pat[j] != pat[i+1]) && (i >= 0))
      i = failure[i];
    if (pat[j] == pat[i+1])
      failure[j] = i+1;
    else failure[j] = -1;
  }
}
```

Program 2.14: Computing the failure function

MODULE 2
Stacks and Queues

Abstract Data Type

- Abstract Data Type as a design tool
- Concerns only on the important concept or model
- No concern on implementation details.
- Stack & Queue is an example of ADT
- An array is not ADT.

What is the difference?

- Stack & Queue vs. Array
  - Arrays are data storage structures while stacks and queues are specialized DS and used as programmer's tools.
- Stack – a container that allows push and pop
- Queue – a container that allows enqueue and dequeue
- No concern on implementation details.
- In an array any item can be accessed, while in these data structures access is restricted.
- They are more abstract than arrays.

Questions?

- Array is not ADT
- Is Linked list ADT?
- Is Binary-tree ADT?
- Is Hash table ADT?
- What about graph?

Stacks

- Allows access to only the last item inserted.
- An item is inserted or removed from the stack from one end called the “top” of the stack.
- This mechanism is called Last-In-First-Out (LIFO).

A Stack Applet example

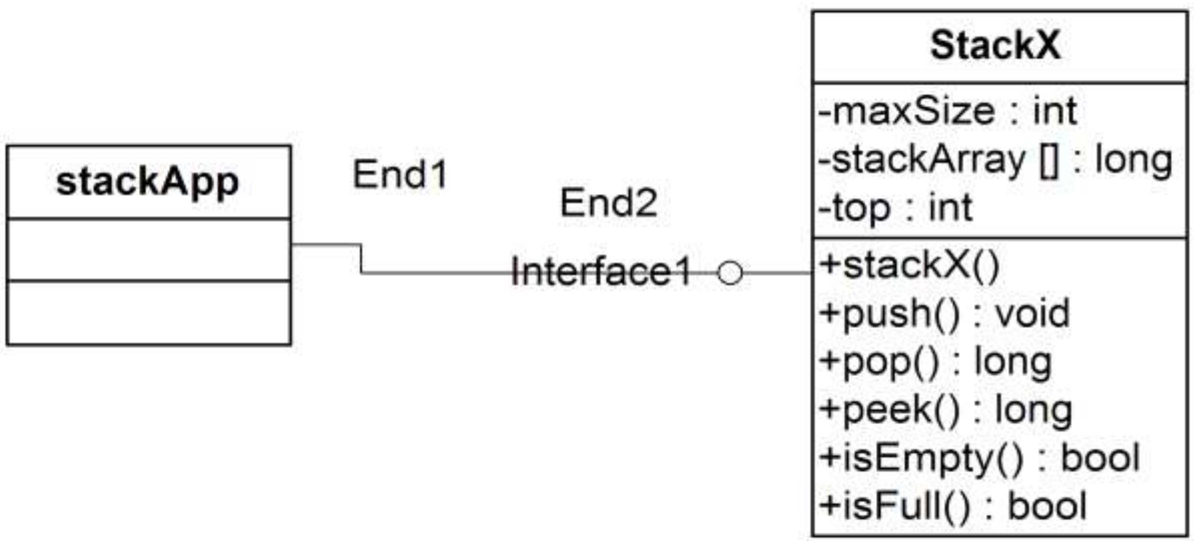
Stack operations

- Placing a data item on the top is called "pushing", while removing an item from the top is called "popping" it.
- *push* and *pop* are the primary stack operations.
- Some of the applications are : microprocessors, some older calculators etc.

Example of Stack codes

- First example stack ADT and implementation

C:\Documents and Settings\box\My Documents\CS\CSC\220\ReaderPrograms\ReaderFiles\Chap04\Stack\stack.java



- *push* and *pop* operations are performed in O(1) time.

Example of Stack codes

- Reversed word
- What is it?
- ABC -> CBA

  C:\Documents and Settings\box\My Documents\CS\CSC\220\ReaderPrograms\ReaderFiles\Chap04\Reverse\reverse.java

Example of Stack codes

- BracketChecker (balancer)
- A syntax checker (compiler) that understands a language containing any strings with balanced brackets ‘{‘ ‘[‘ ‘(‘ and ‘)’, ‘]’, ‘}’
  - S -> Bl S1 Br
  - S1 -> Bl string Br
  - Bl -> ‘{‘ | ‘[‘ | ‘(‘
  - Br -> ‘)’, | ‘]’, | ‘}’

  C:\Documents and Settings\box\My Documents\CS\CSC\220\ReaderPrograms\ReaderFiles\Chap04\Brackets\brackets.java

Queues

- Queue is an ADT data structure similar to stack, except that the first item to be inserted is the first one to be removed.
- This mechanism is called First-In-First-Out (FIFO).
- Placing an item in a queue is called “insertion or enqueue”, which is done at the end of the queue called “rear”.
- Removing an item from a queue is called “deletion or dequeue”, which is done at the other end of the queue called “front”.
- Some of the applications are : printer queue, keystroke queue, etc.

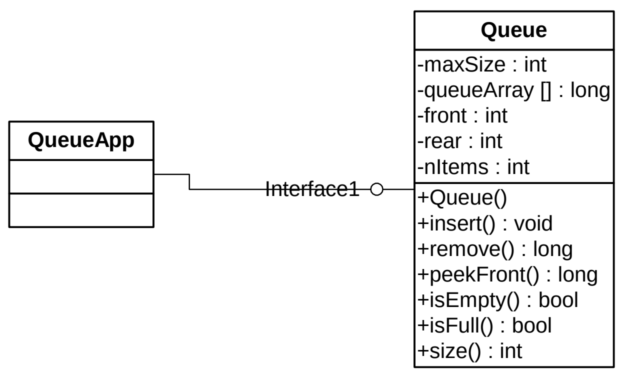
Circular Queue

- When a new item is inserted at the rear, the pointer to rear moves upwards.
- Similarly, when an item is deleted from the queue the front arrow moves downwards.
- After a few insert and delete operations the rear might reach the end of the queue and no more items can be inserted although the items from the front of the queue have been deleted and there is space in the queue.

Circular Queue

- To solve this problem, queues implement wrapping around. Such queues are called Circular Queues.
- Both the front and the rear pointers wrap around to the beginning of the array.
- It is also called as “Ring buffer”.
- Items can inserted and deleted from a queue in O(1) time.

Queue Example



Queue sample code

- C:\Documents and Settings\box\My Documents\CS\CSC\220\ReaderPrograms\ReaderFiles\Chap04\Queue\queue.java

Various Queues

- Normal queue (FIFO)
- Circular Queue (Normal Queue)
- Double-ended Queue (Deque)
- Priority Queue

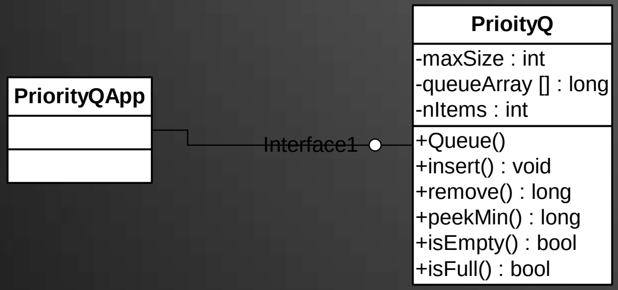
Deque

- It is a double-ended queue.
- Items can be inserted and deleted from either ends.
- More versatile data structure than stack or queue.
- E.g. policy-based application (e.g. low priority go to the end, high go to the front)
- In a case where you want to sort the queue once in a while, What sorting algorithm will you use?

Priority Queues

- More specialized data structure.
- Similar to Queue, having front and rear.
- Items are removed from the front.
- Items are ordered by key value so that the item with the lowest key (or highest) is always at the front.
- Items are inserted in proper position to maintain the order.
- Let’s discuss complexity

Priority Queue Example



Priority Queues

- Used in multitasking operating system.
- They are generally represented using "heap" data structure.
- Insertion runs in O(n) time, deletion in O(1) time.

- C:\Documents and Settings\box\My Documents\CS\CSC\220\ReaderPrograms\ReaderFiles\Chap04\PriorityQ\priorityQ.java

Parsing Arithmetic Expressions

- 2 + 3
- 2 + 4 * 5
- ((2 + 4) * 7) + 3* (9 – 5))

- 2 3 +
- 2 4 5 * +
- 2 4 + 7 * 3 9 5 - * +

- Infix vs postfix
- Why do we want to do this transformation?

Infix to postfix

- Read ch from input until empty
  - If ch is arg , output = output + arg
  - If ch is “(“, push ‘(‘;
  - If ch is op and higher than top push ch
  - If ch is “)” or end of input,
    - output = output + pop() until empty or top is “(“
  - Read next input
- C:\Documents and Settings\box\My Documents\CS\CSC\220\ReaderPrograms\ReaderFiles\Chap04\Postfix\postfix.java

Postfix eval

- 5 + 2 * 3 -> 5 2 3 * +
- Algorithm
  - While input is not empty
  - If ch is number , push (ch)
  - Else
    - Pop (a)
    - Pop(b)
    - Eval (ch, a, b)
- C:\Documents and Settings\box\My Documents\CS\CSC\220\ReaderPrograms\ReaderFiles\Chap04\Postfix\postfix.java

Recursive Thinking

- ***Recursion*** is:
  - A *problem-solving* ***approach***, that can ...
  - Generate *simple solutions* to ...
  - *Certain kinds* of problems that ...
  - Would be *difficult to solve in other ways*
- Recursion *splits a problem*:
  - Into one or more *simpler versions of* ***itself***

Recursive Thinking: Another Example

Strategy for searching a sorted array:

1. if the array is empty
2. return -1 as the search result (not present)
3. else if the middle element == target
4. return subscript of the middle element
5. else if target < middle element
6. recursively search elements before middle
7. else
8. recursively search elements after the middle

Recursive Thinking: The General Approach

1. if problem is "*small enough*"
2. solve it *directly*
3. else
4. break into one or more *smaller subproblems*
5. solve each subproblem *recursively*
6. *combine* results into solution to whole problem

Requirements for Recursive Solution

- At least one "*small*" case that you can solve directly
- A way of *breaking* a larger problem down into:
  - One or more *smaller* subproblems
  - Each of the *same kind* as the original
- A way of *combining* subproblem results into an overall solution to the larger problem

General Recursive Design Strategy

- Identify the ***base case(s)*** (for direct solution)
- Devise a problem ***splitting strategy***
  - Subproblems must be smaller
  - Subproblems must work towards a base case
- Devise a solution ***combining strategy***

Recursive Design Example

Recursive algorithm for finding length of a string:

1. if string is empty (no characters)
2. return 0 ← *base case*
3. else ← *recursive case*
4. compute length of string without first character
5. return 1 + that length

Note: Not best technique for this problem; illustrates the approach.

Recursive Design Example: Code

Recursive algorithm for finding length of a string:

```
public static int length (String str) {
  if (str == null ||
        str.equals(“”))
     return 0;
  else
     return length(str.substring(1)) + 1;
}
```

Recursive Design Example: `printChars`

Recursive algorithm for printing a string:

```
public static void printChars
    (String str) {
  if (str == null ||
      str.equals(""))
    return;
  else
    System.out.println(str.charAt(0));
    printChars(str.substring(1));
}
```

Recursive Design Example: printChars2

Recursive algorithm for printing a string?

```
public static void printChars2
    (String str) {
  if (str == null ||
      str.equals(“”))
    return;
  else
    printChars2(str.substring(1));
    System.out.println(str.charAt(0));
}
```

Recursive Design Example: `mystery`

What does this do?

```
public static int mystery (int n) {
   if (n == 0)
      return 0;
   else
      return n + mystery(n-1);
}
```

Proving a Recursive Method Correct

Recall ***Proof by Induction:***

1. Prove the theorem for the *base case(s):* ***n=0***
2. Show that:
   - ***If*** the theorem is *assumed true* for ***n***,
   - ***Then*** it *must be true* for ***n+1***

Result: Theorem true for all $n \geq 0$.

Proving a Recursive Method Correct (2)

Recursive proof is similar to induction:

1. Show *base case* recognized and *solved correctly*
2. Show that
   - ***If*** all *smaller problems* are *solved correctly*,
   - ***Then*** *original problem* is also *solved correctly*

3. Show that each recursive case makes progress towards the base case ← *terminates properly*

Tracing a Recursive Method

Overall result

```
length("ace")
```

3

```
return 1 + length("ce")
```

2

```
return 1 + length("e")
```

1

```
return 1 + length("")
```

0

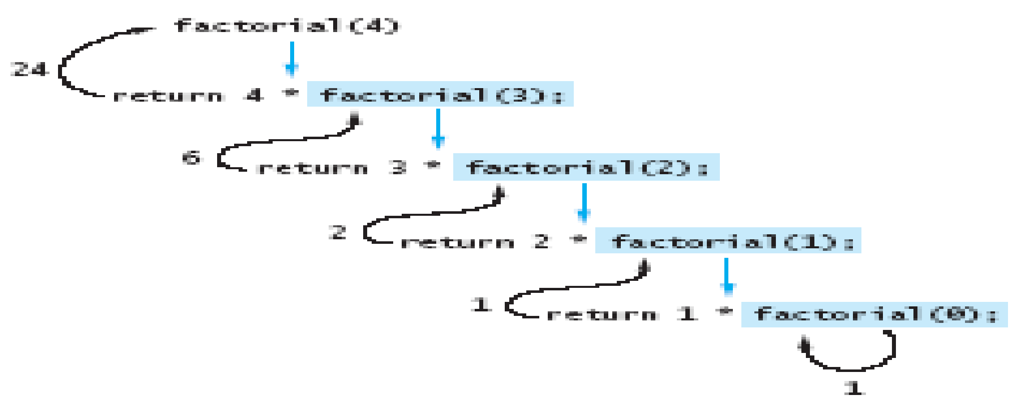
Recursive Definitions of Mathematical Formulas

- Mathematicians often use *recursive definitions*
- These lead very naturally to *recursive algorithms*
- Examples include:
  - Factorial
  - Powers
  - Greatest common divisor

Recursive Definitions: Factorial

- 0! = 1
- n! = n x (n–1)!

FIGURE 7.5
Trace of
factorial(4)



- If a recursive function never reaches its base case, a stack overflow error occurs

Recursive Definitions: Factorial Code

```
public static int factorial (int n) {
  if (n == 0) // or: throw exc. if < 0
    return 1;
  else
    return n * factorial(n-1);
}
```

Recursive Definitions: Power

- $x^0 = 1$
- $x^n = x \times x^{n-1}$

```
public static double power
     (double x, int n) {
   if (n <= 0)  // or: throw exc. if < 0
     return 1;
   else
     return x * power(x, n-1);
}
```

Recursive Definitions: Greatest Common Divisor

Definition of gcd(m, n), for integers m > n > 0:

- gcd(m, n) = n, if n divides m evenly
- gcd(m, n) = gcd(n, m % n), otherwise

```
public static int gcd (int m, int n) {
  if (m < n)
    return gcd(n, m);
  else if (m % n == 0) // could check n>0
    return n;
  else
    return gcd(n, m % n);
}
```

Recursive Definitions: Fibonacci Series

Definition of fib_i, for integer $i > 0$:

- $fib_1 = 1$
- $fib_2 = 1$
- $fib_n = fib_{n-1} + fib_{n-2}$, for $n > 2$

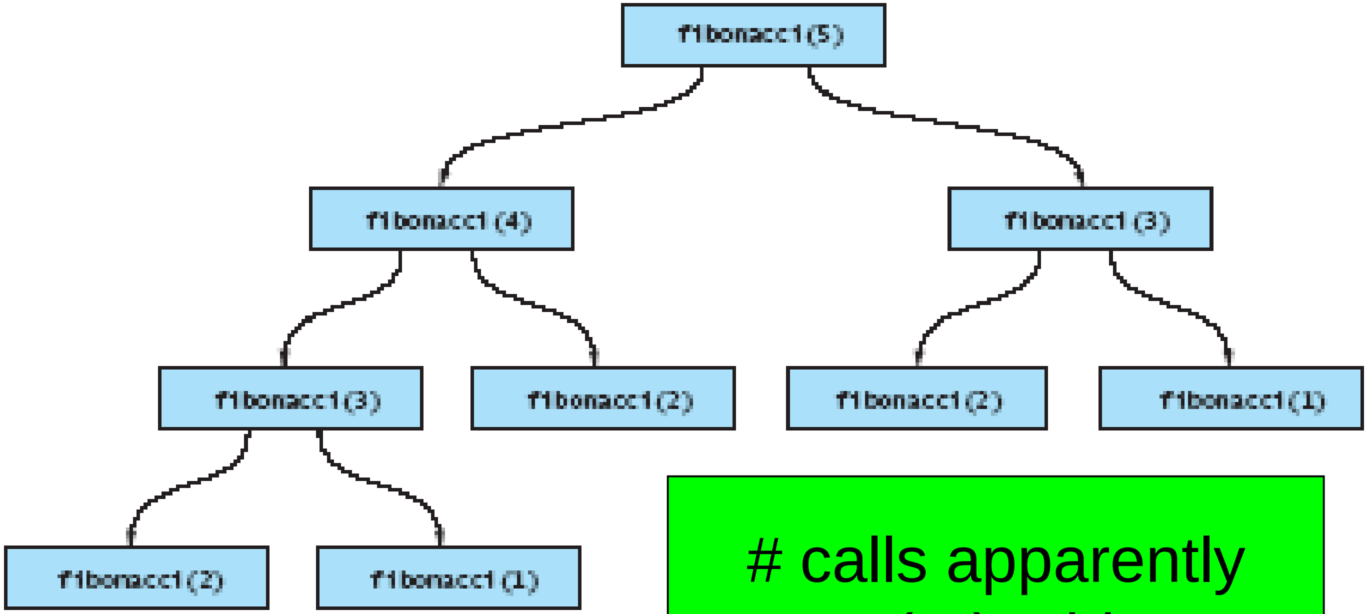
Fibonacci Series Code

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

This is straightforward, but an inefficient recursion ...

Efficiency of Recursion: Inefficient Fibonacci

FIGURE 7.6
Method Calls Resulting from `fibonacci(5)`



calls apparently $O(2^n)$ – big!

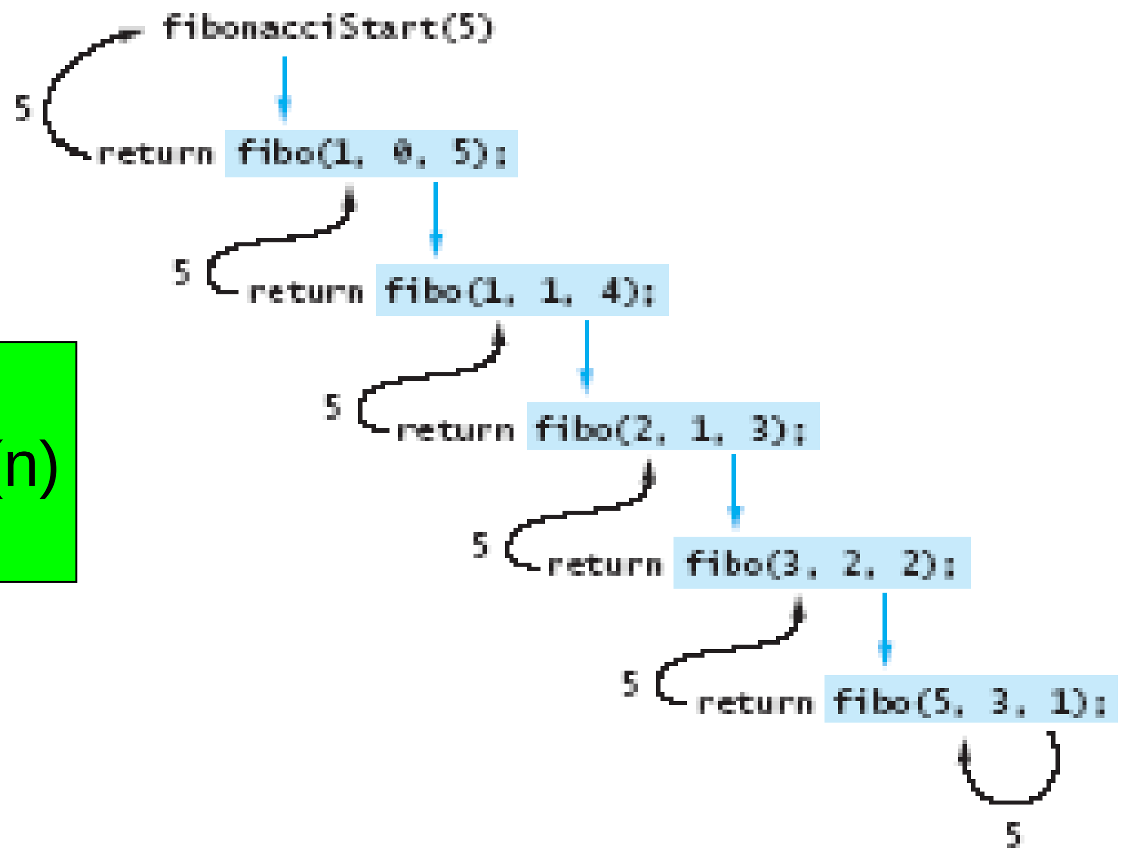
Efficient Fibonacci: Code

```
public static int fibStart (int n) {
  return fibo(1, 0, n);
}

private static int fibo (
    int curr, int prev, int n) {
  if (n <= 1)
    return curr;
  else
    return fibo(curr+prev, curr, n-1);
}
```

Efficient Fibonacci: A Trace

FIGURE 7.7
Trace of
`fibonacciStart(5)`



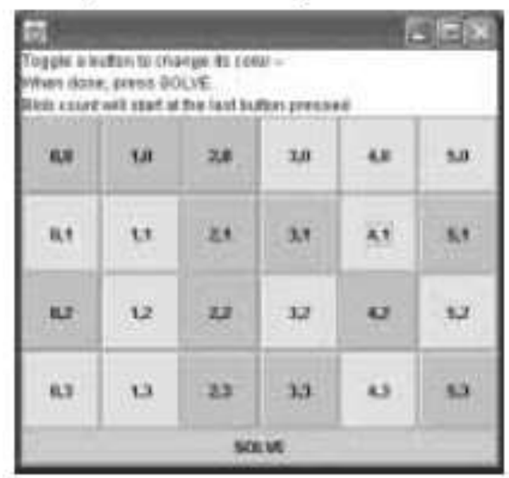
Performance is O(n)

Problem Solving with Recursion

- Towers of Hanoi
- Counting grid squares in a blob
- Backtracking, as in maze search

FIGURE 7.11
Children's Version of Towers of Hanoi

FIGURE 7.16
A Sample Grid for Counting Cells in a Blob



Towers of Hanoi: Description

Goal: Move entire tower to another peg

Rules:

1. You can move only the top disk from a peg.
2. You can only put a smaller on a larger disk (or on an empty peg)

FIGURE 7.11
Children's Version of Towers of Hanoi

Towers of Hanoi: Solution Strategy

FIGURE 7.11
Children's Version of Towers of Hanoi

FIGURE 7.12
Towers of Hanoi After the First Two Steps in Solution of the Three-Disk Problem

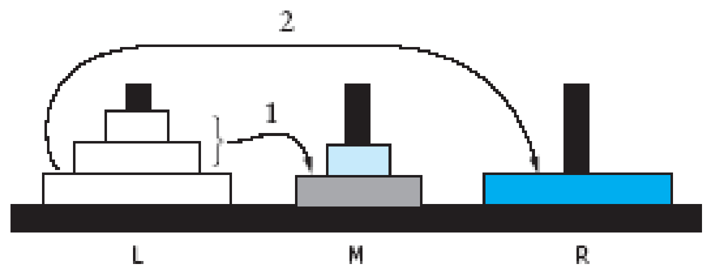
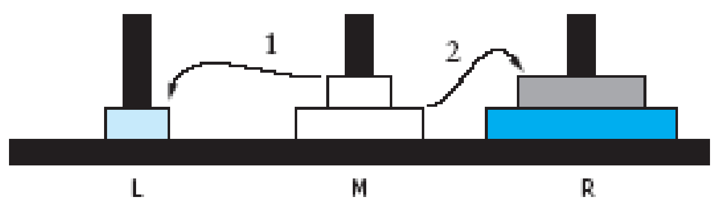


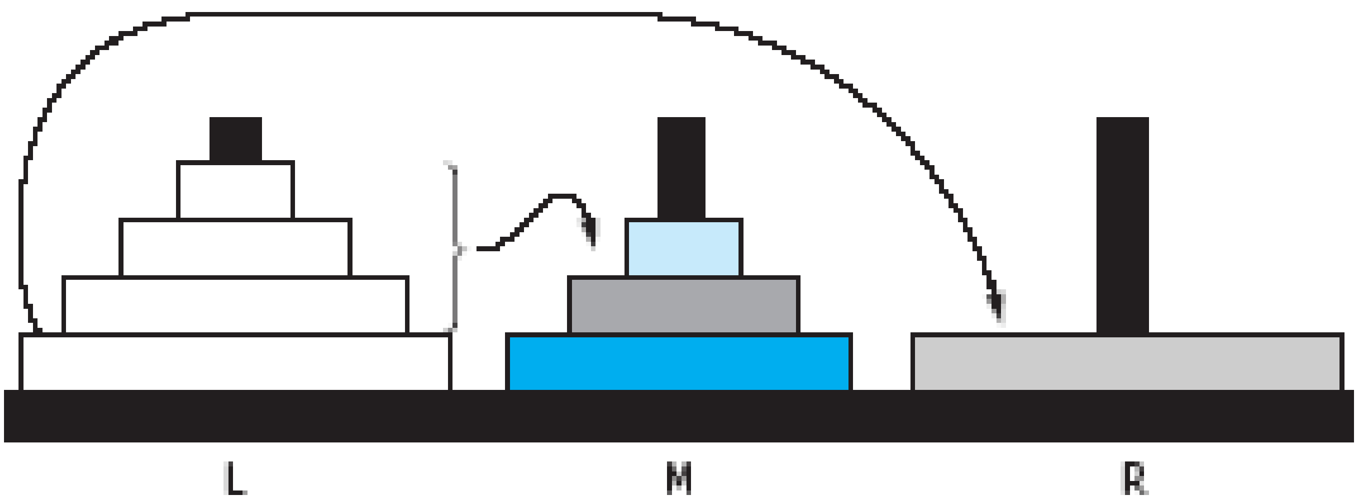
FIGURE 7.13
Towers of Hanoi After First Two Steps in Solution of Two-Disk Problem



Towers of Hanoi: Solution Strategy (2)

FIGURE 7.14

Towers of Hanoi After the First Two Steps in Solution of the Four-Disk Problem



Towers of Hanoi: Program Specification

TABLE 7.1

Inputs and Outputs for Towers of Hanoi Problem

Problem Inputs
Number of disks (an integer)
Letter of starting peg: L (left), M (middle), or R (right)
Letter of destination peg (L, M, or R), but different from starting peg
Letter of temporary peg (L, M, or R), but different from starting peg and destination peg
Problem Outputs
A list of moves

Towers of Hanoi: Program Specification (2)

TABLE 7.2

Class `TowersOfHanoi`

Method	Behavior
`public String showMoves(int n, char startPeg, char destPeg, char tempPeg)`	Builds a string containing all moves for a game with `n` disks on `startPeg` that will be moved to `destPeg` using `tempPeg` for temporary storage of disks being moved.

Towers of Hanoi: Recursion Structure

```
move(n, src, dst, tmp) =
  if n == 1:  move disk 1 from src to dst
  otherwise:
     move(n-1, src, tmp, dst)
     move disk n from src to dst
     move(n-1, tmp, dst, src)
```

Towers of Hanoi: Code

```
public class TowersOfHanoi {
  public static String showMoves(int n,
      char src, char dst, char tmp) {
    if (n == 1)
      return "Move disk 1 from " + src +
             " to " + dst + "\n";
    else return
      showMoves(n-1, src, tmp, dst) +
      "Move disk " + n + " from " + src +
        " to " + dst + "\n" +
      showMoves(n-1, tmp, dst, src);
  }
}
```

Towers of Hanoi: Performance Analysis

How big will the string be for a tower of size n?

We'll just count lines; call this L(n).

- For $n = 1$, one line: $L(1) = 1$
- For $n > 1$, one line plus twice L for next smaller size:

  $L(n+1) = 2 \times L(n) + 1$

Solving this gives $L(n) = 2^n - 1 = O(2^n)$

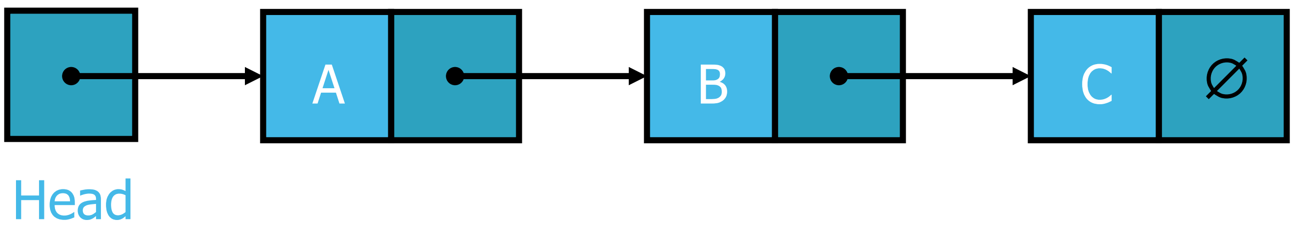
So, don't try this for very large n – you will do a lot of string concatenation and garbage collection, and then run out of heap space and terminate.

MODULE 3
Linked Lists

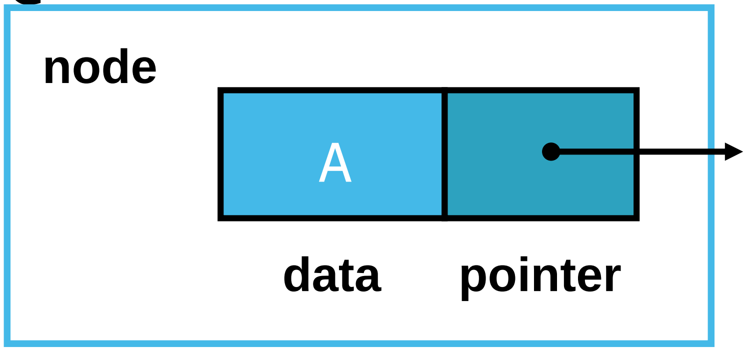
List Overview

- 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;
};
```

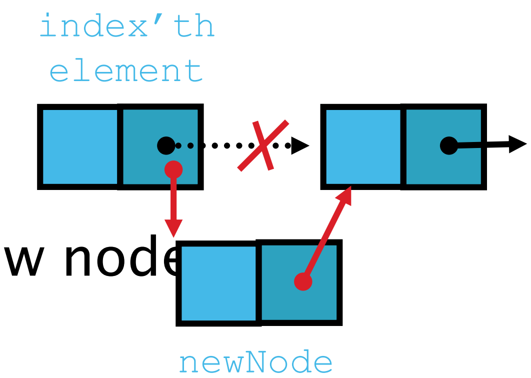
A Simple Linked List Class

- 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

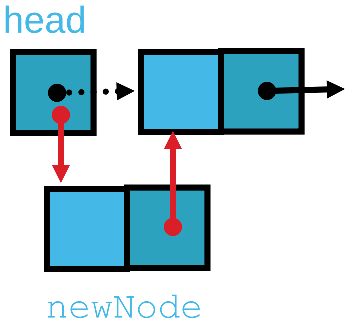
Inserting a new node

```
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



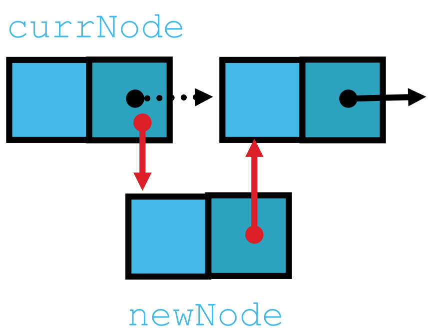
Inserting a new node

```
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

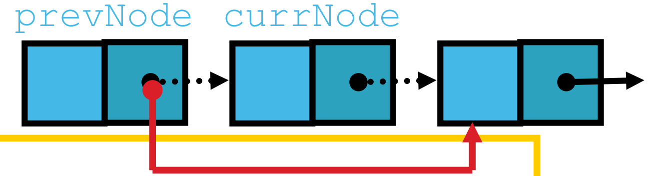
Deleting a node

```
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;
}
```

Try to find the node with its value equal to **x**

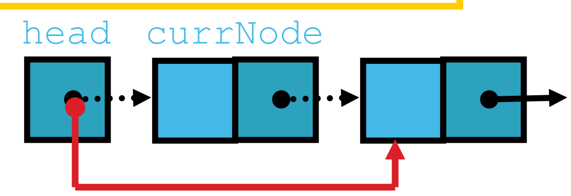
Deleting a node

```
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

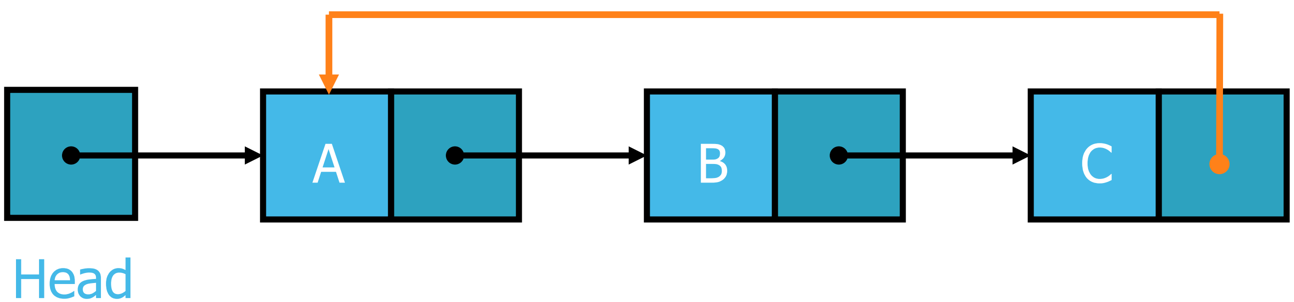
```
int main(void)
{
        List list;
        list.InsertNode(0, 7.0);   // successful
        list.InsertNode(1, 5.0);   // successful
        list.InsertNode(-1, 5.0);  // unsuccessful
        list.InsertNode(0, 6.0);   // successful
        list.InsertNode(8, 4.0);   // unsuccessful
        // print 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

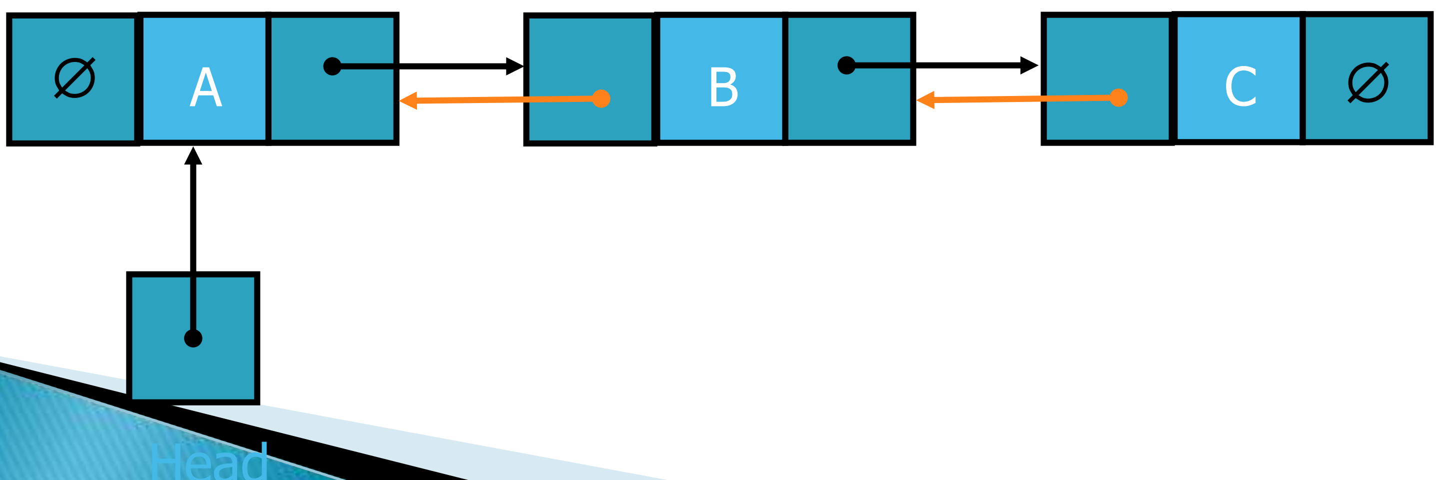
- *Circular linked lists*
  - The last node points to the first node of the list



  - 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

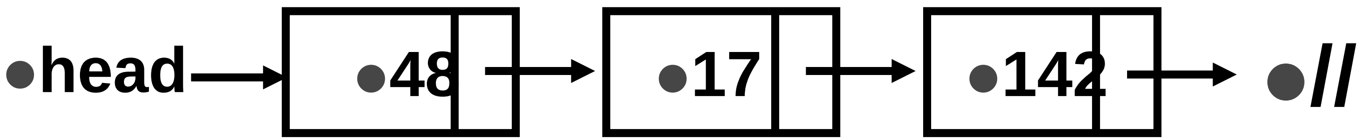
- *Doubly linked lists*
  - Each node points to not only successor but the predecessor
  - There are two `NULL`: at the first and last nodes in the list
  - Advantage: given a node, it is easy to visit its predecessor. Convenient to traverse lists backwards



Array versus Linked Lists

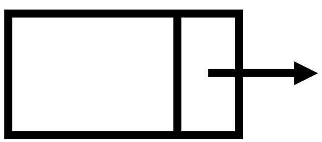
- Linked lists are more complex to code and manage than arrays, but they have some distinct advantages.
  - Dynamic: a linked list can easily grow and shrink in size.
    - We don't need to know how many nodes will be in the list. They are created in memory as needed.
    - In contrast, the size of a 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

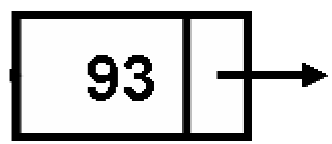


- Follow the previous steps and we get

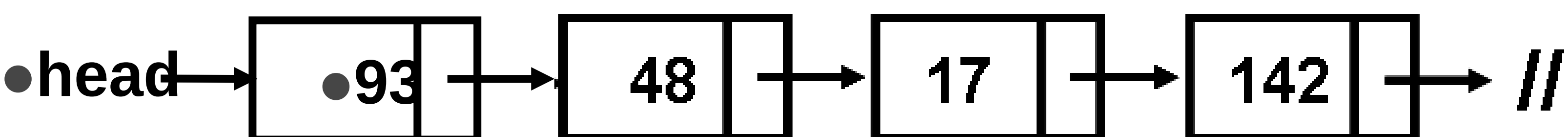
- Step 1



- Step 2



- Step 3



Insertion Description

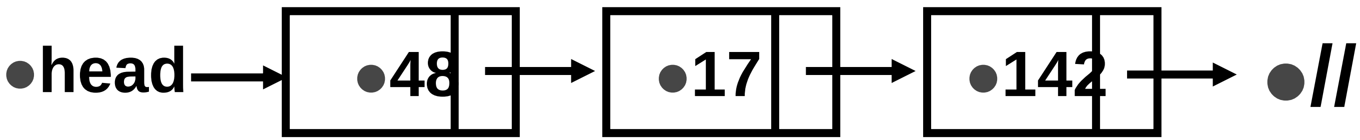
- Insertion at the top of the list
- Insertion at the end of the list
- Insertion in the middle of the list

Insertion at the end

Steps:

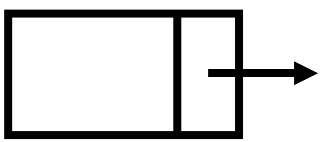
- Create a Node
- Set the node data Values
- Connect the pointers

Insertion Description

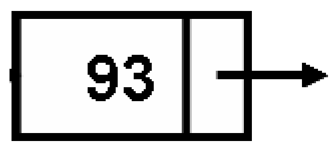


- Follow the previous steps and we get

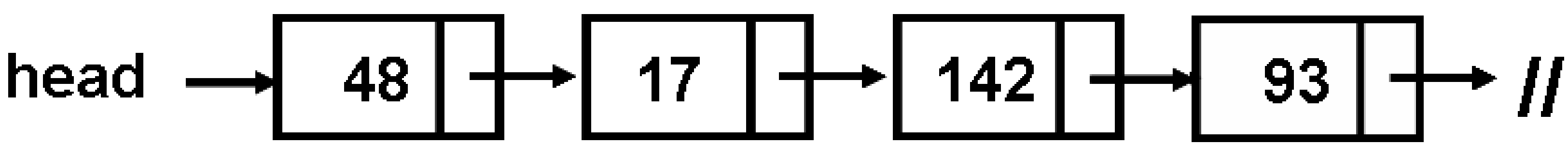
- Step 1



- Step 2



- Step 3



Insertion Description

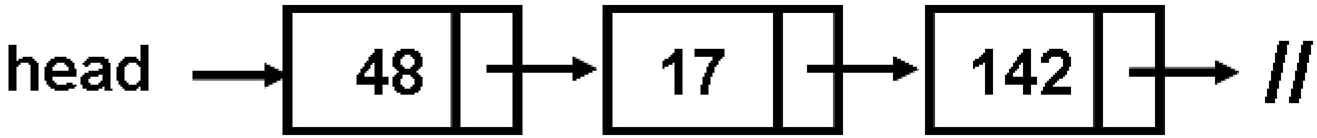
- Insertion at the top of the list
- Insertion at the end of the list
- Insertion in the middle of the list

Insertion in the middle

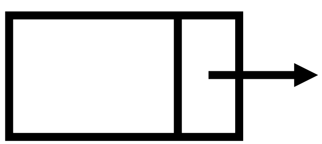
Steps:

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

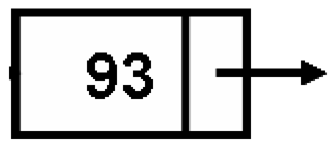
Insertion Description



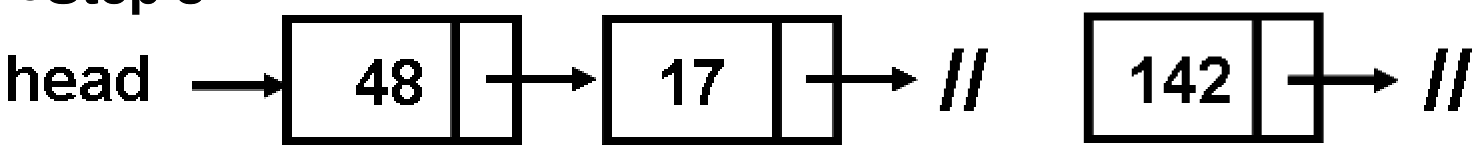
- Step 1



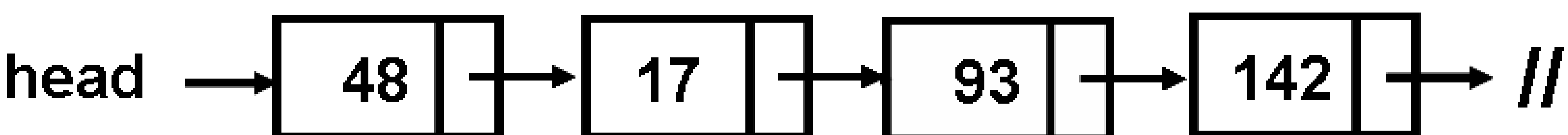
- Step 2



- 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

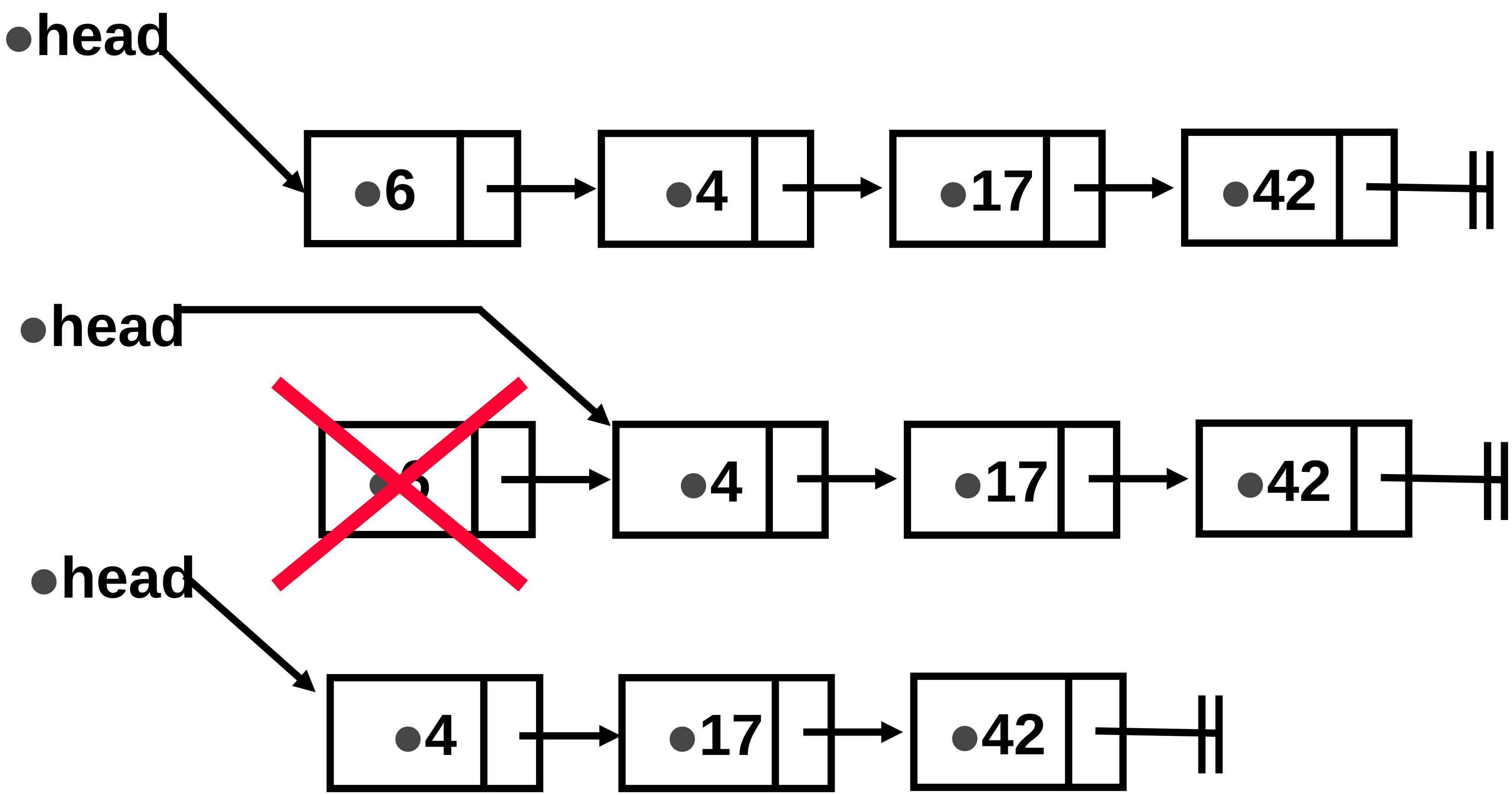
- Deleting from the top of the list
- Deleting from the end of the list
- Deleting from the middle of the list

Deleting from the top

Steps

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

Deletion Description



Deletion Description

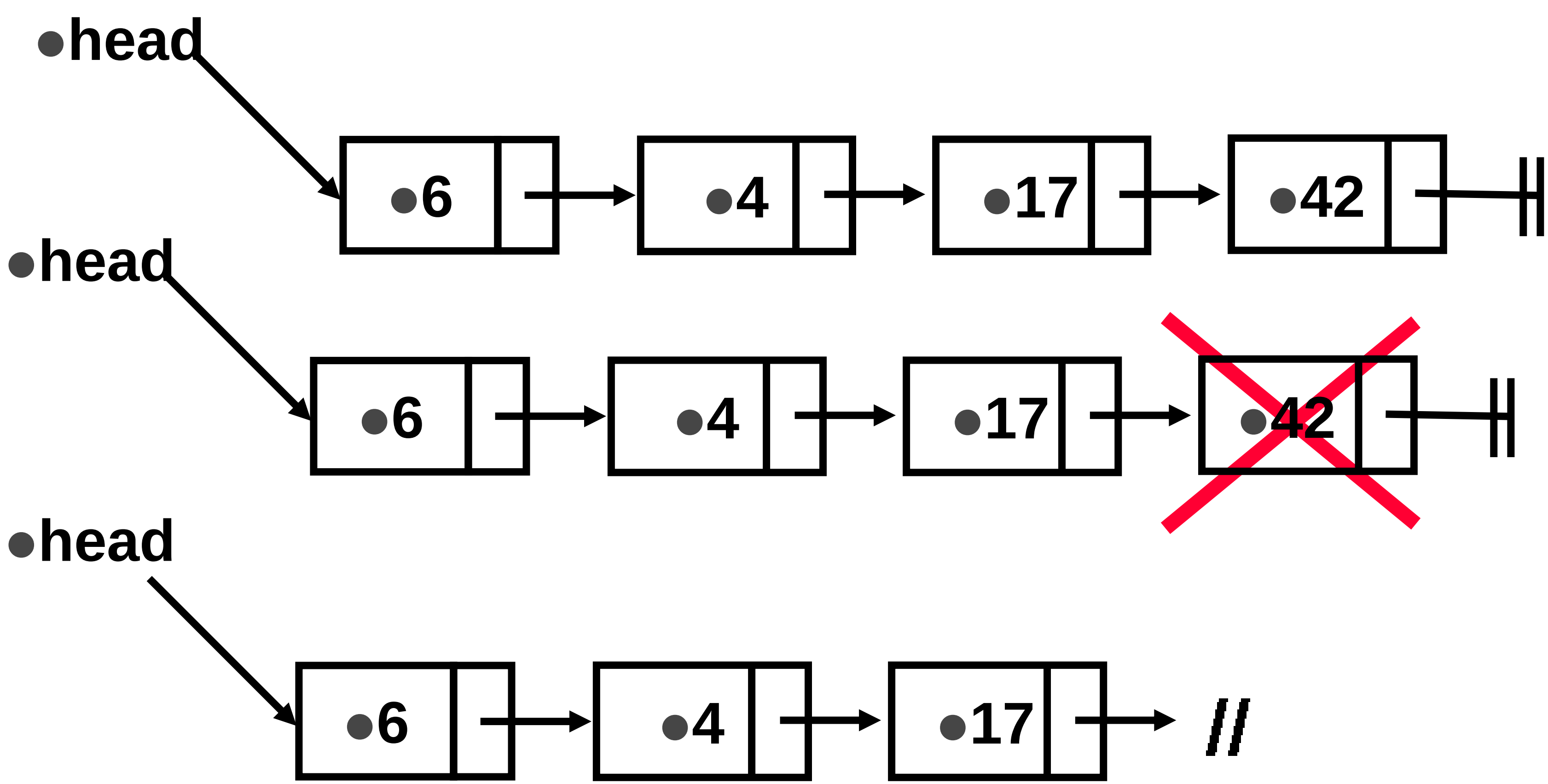
- Deleting from the top of the list
- Deleting from the end of the list
- Deleting from the middle of the list

Deleting from the end

Steps

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

Deletion Description



Deletion Description

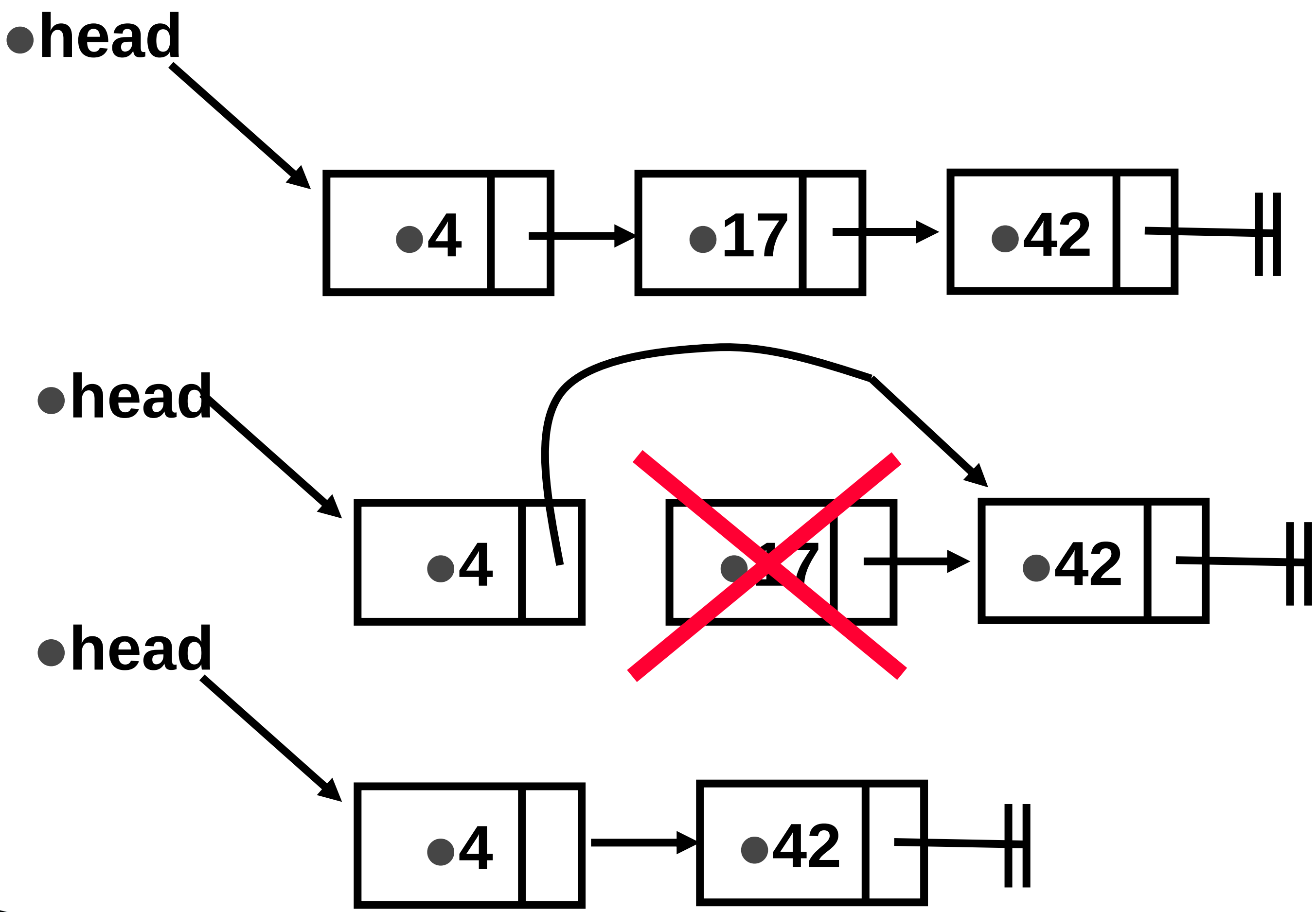
- Deleting from the top of the list
- Deleting from the end of the list
- Deleting from the middle of the list

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"
};
```

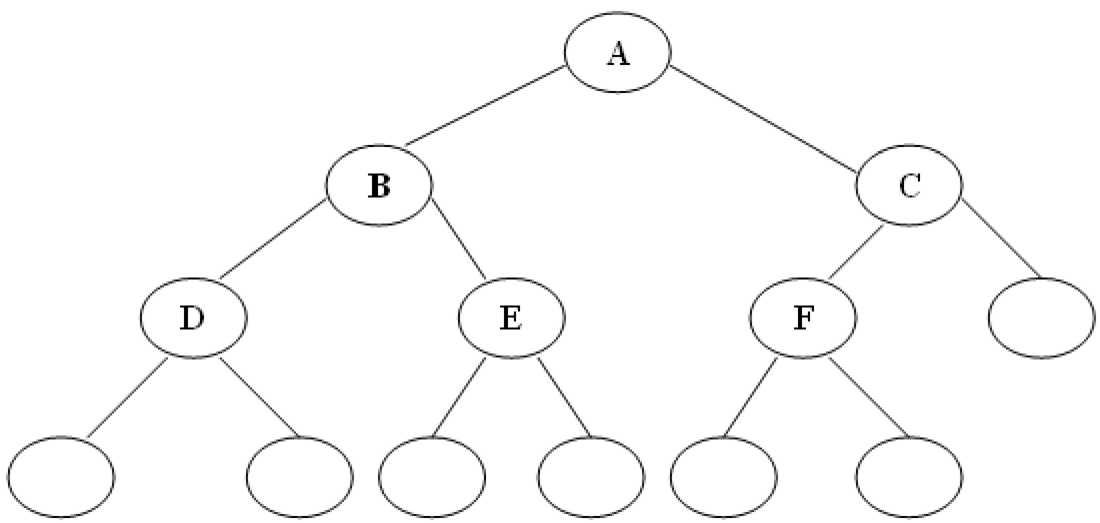
MODULE – 4

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

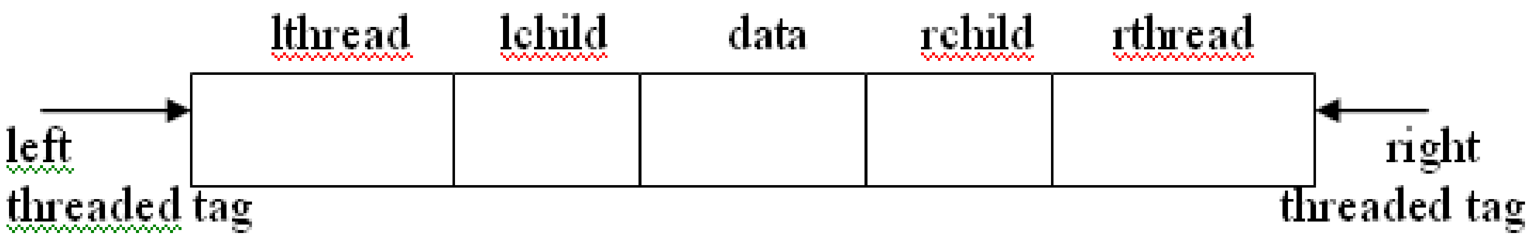
- In above binary tree, there are 7 null pointers & actual 5 pointers.
- In all there are 12 pointers.
- We can generalize it that for any binary tree with n nodes there will be (n+1) null pointers and 2n total pointers.
- The objective here to make effective use of these null pointers.
- A. J. perils & 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

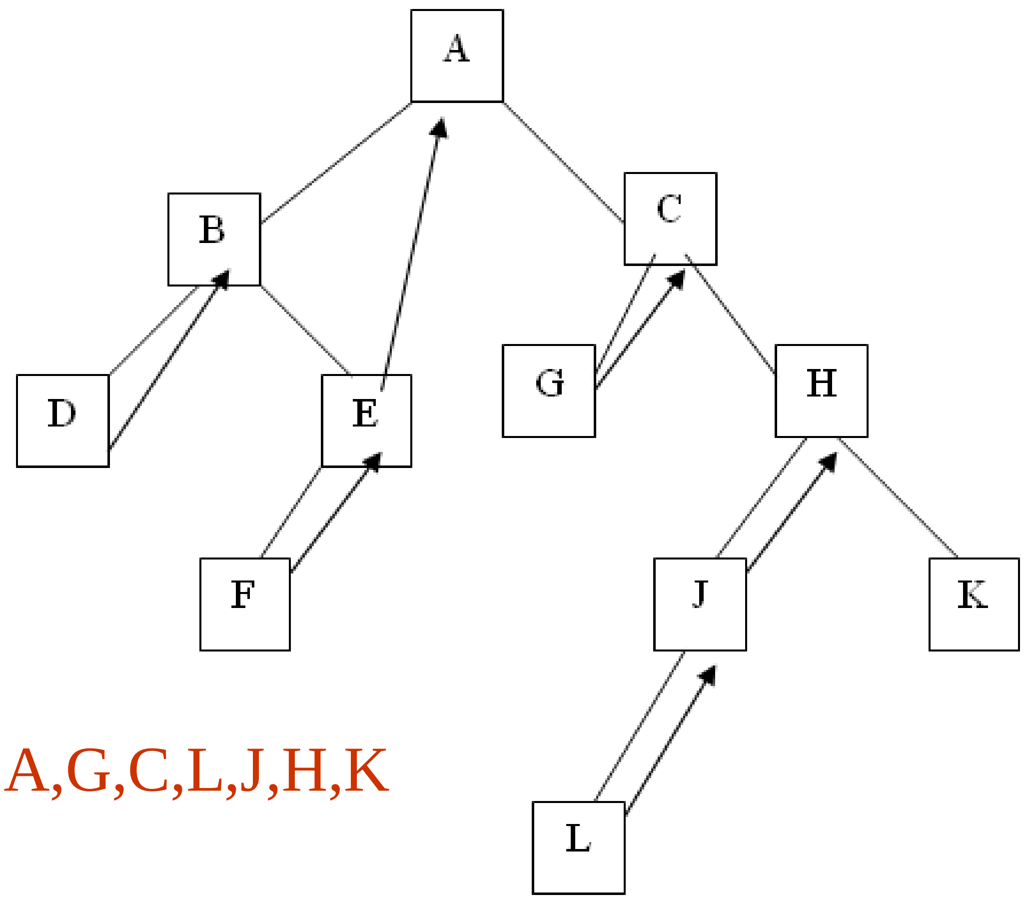
Threaded Binary Tree

- 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

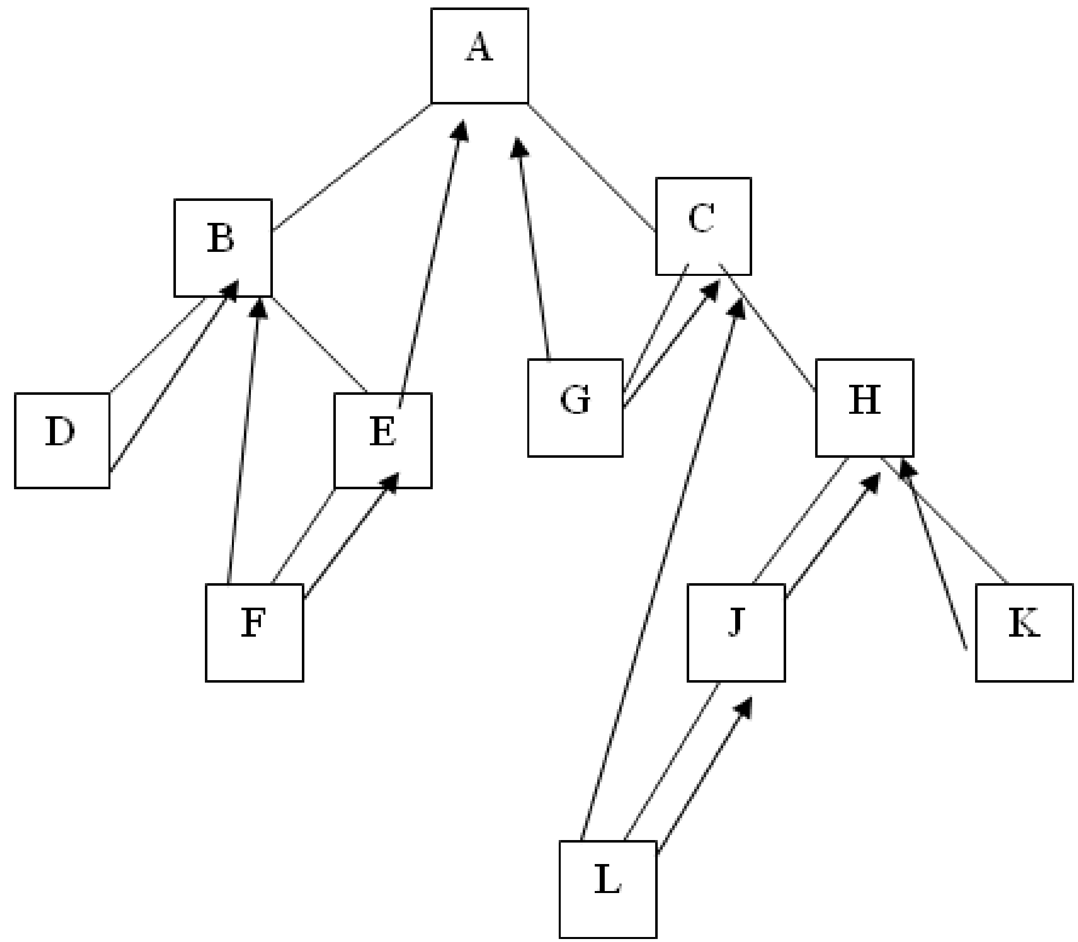
Threaded Binary Tree
Two-Way

- 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



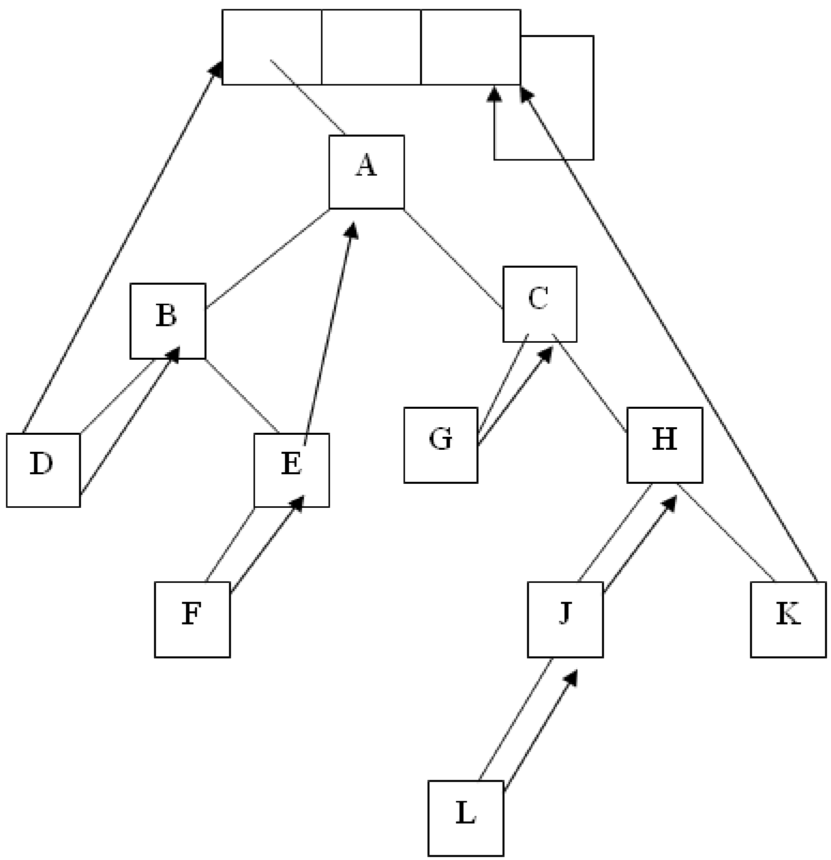
Two-way inorder threading

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

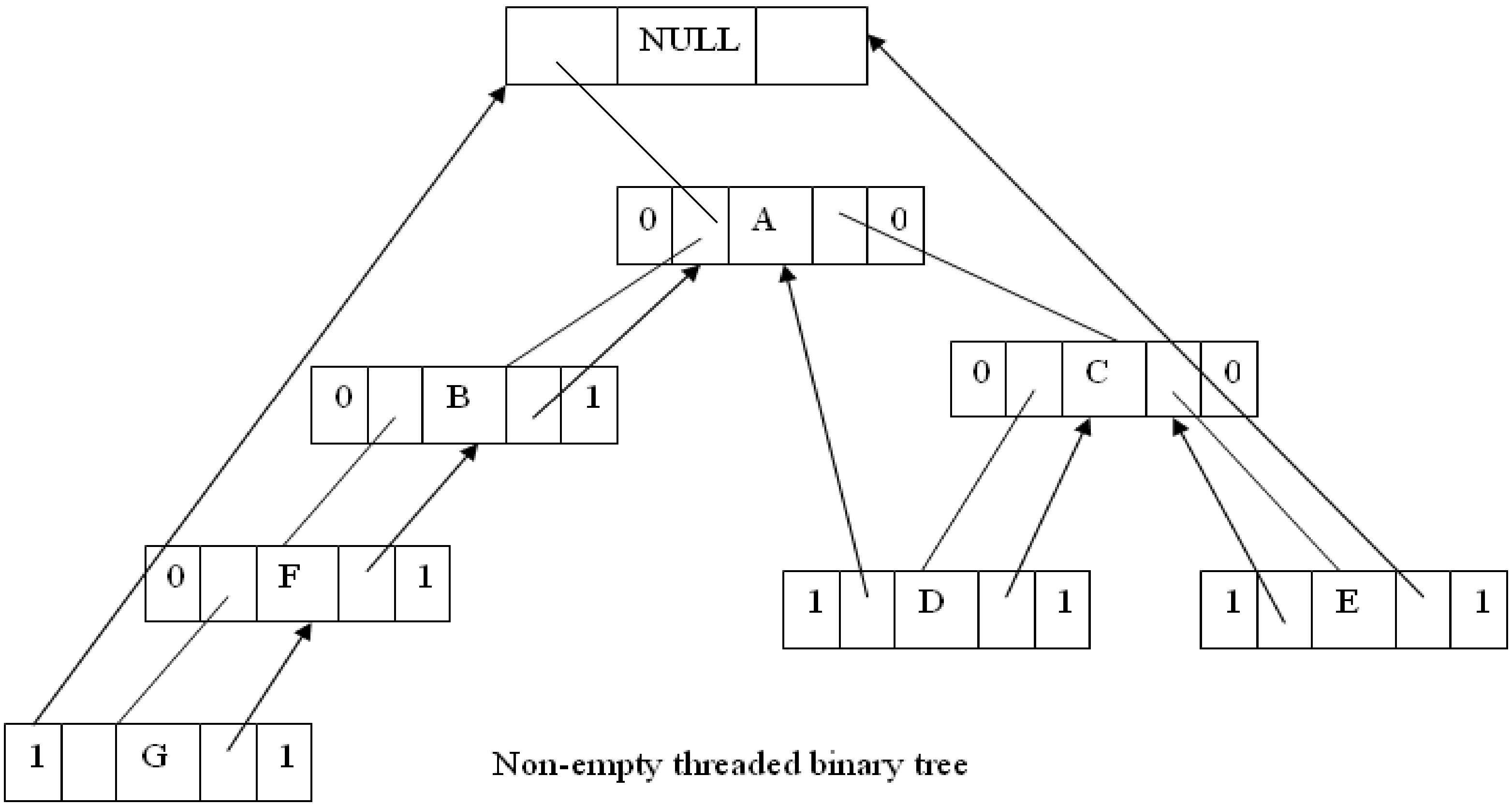
- Bellow figure to explain two-way threading with header node



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 second advantage is more understated 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 be accessible from any other 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.

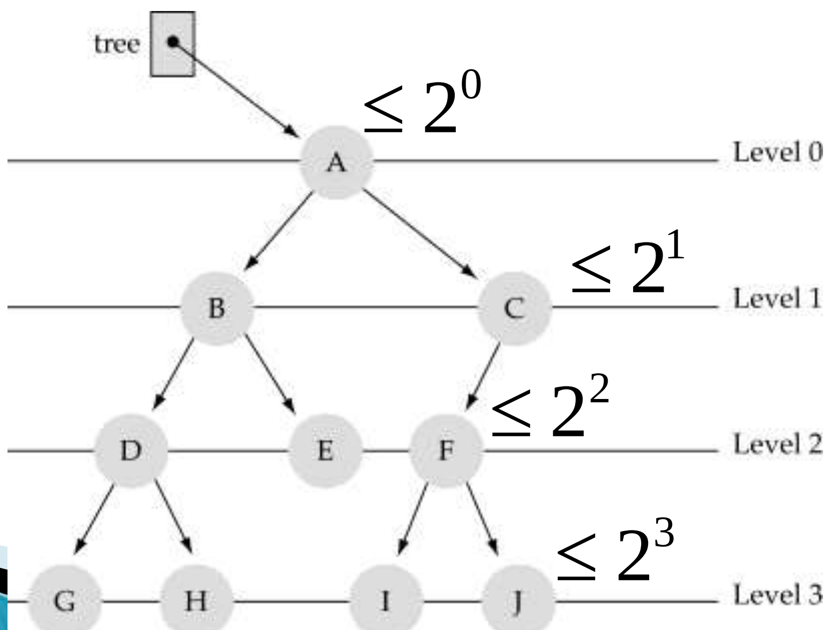
Threaded Binary Tree

- **Disadvantages of 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.

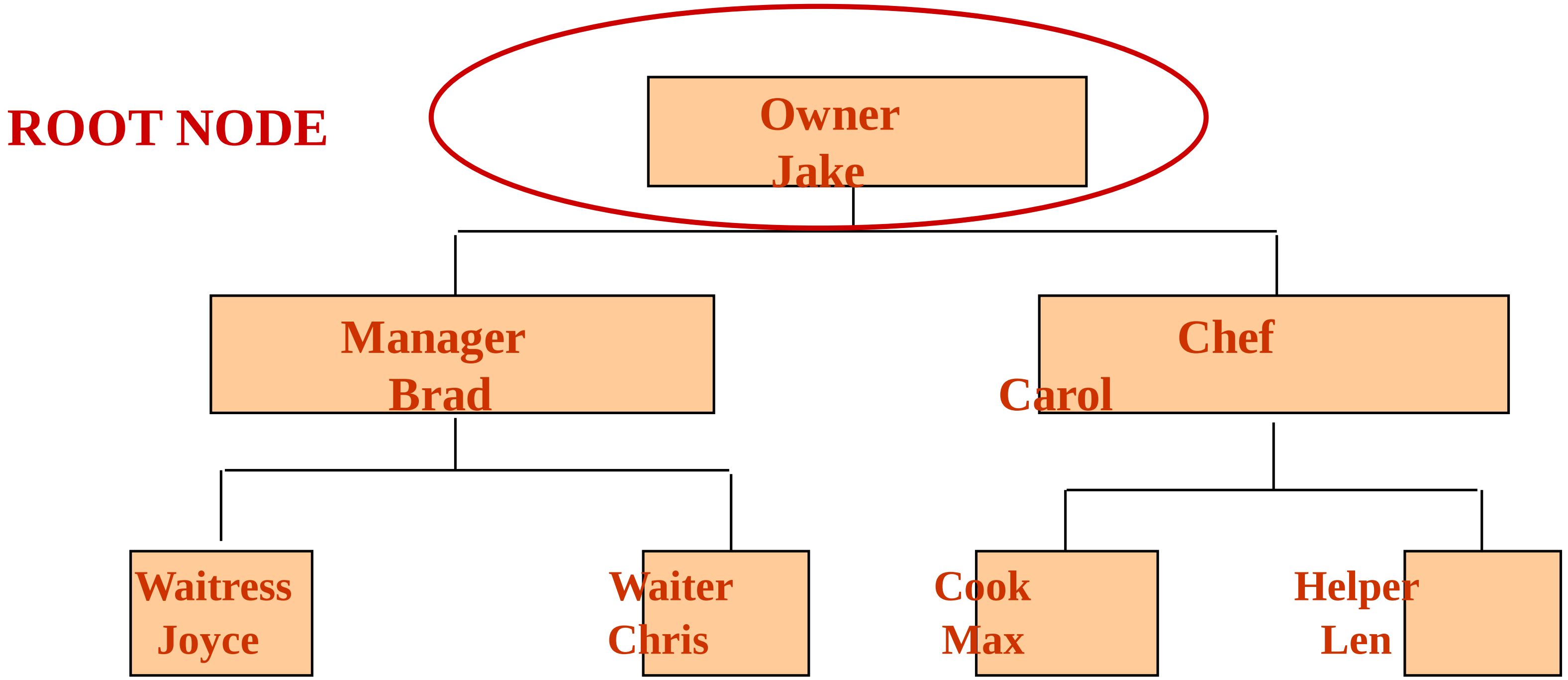
Binary Search Trees

What is a binary tree?

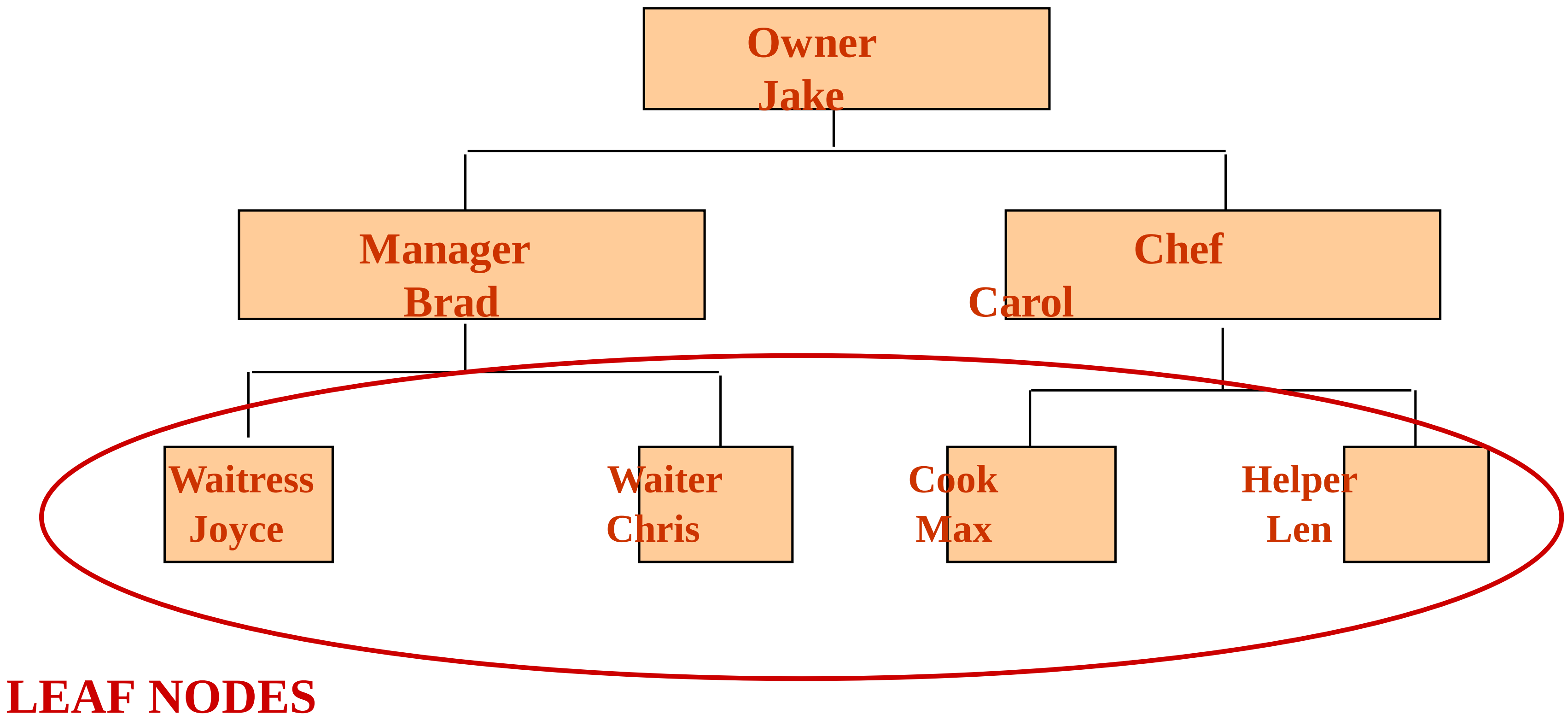
- *Property1*: each node can have up to two successor nodes (*children*)
  - The predecessor node of a node is called its *parent*
  - The "beginning" node is called the *root* (no parent)
  - A node without *children* is called a *leaf*



A Tree Has a Root Node

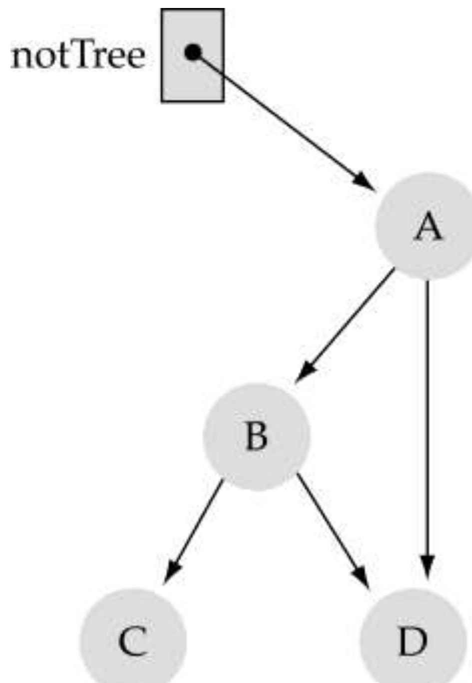


Leaf nodes have no children



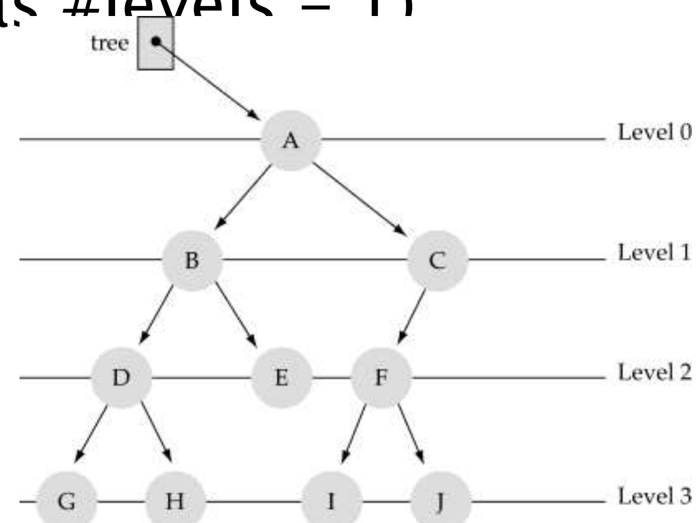
What is a binary tree? (cont.)

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

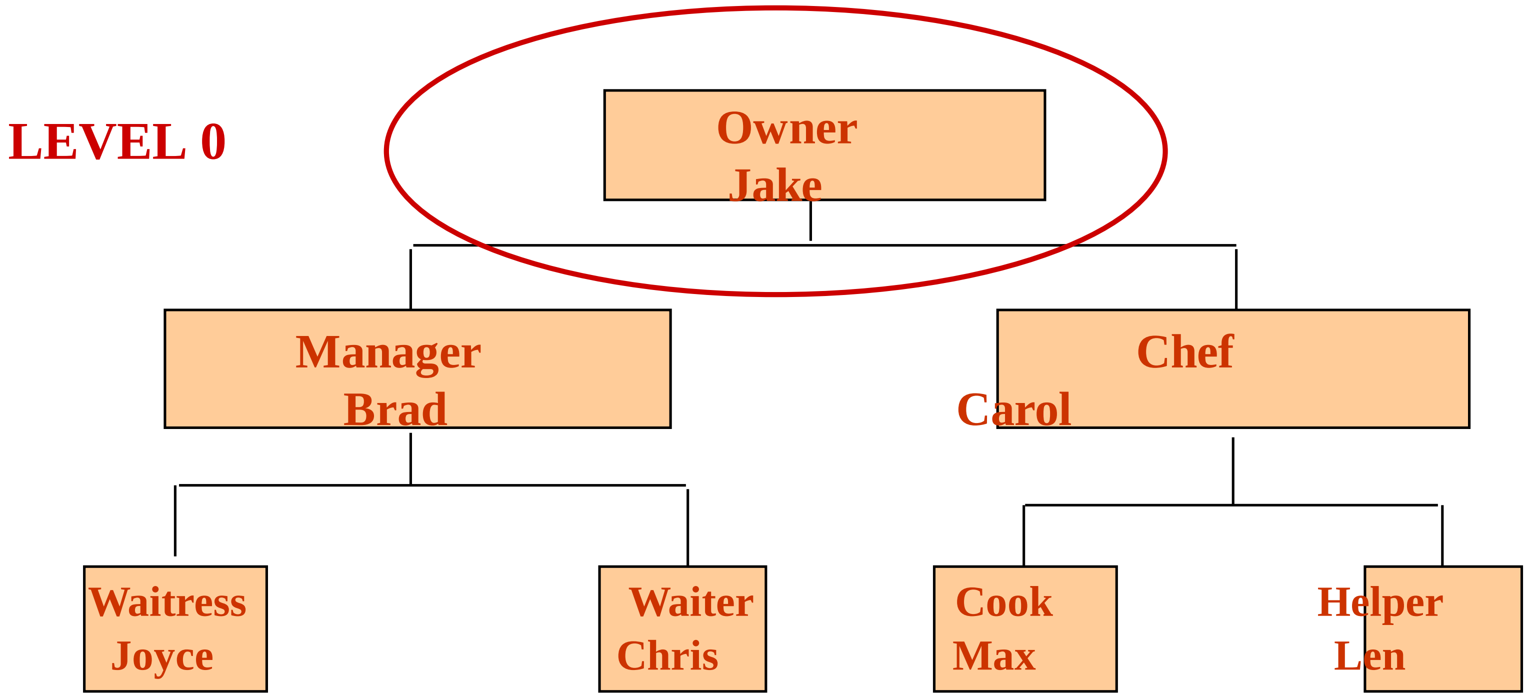


Some terminology

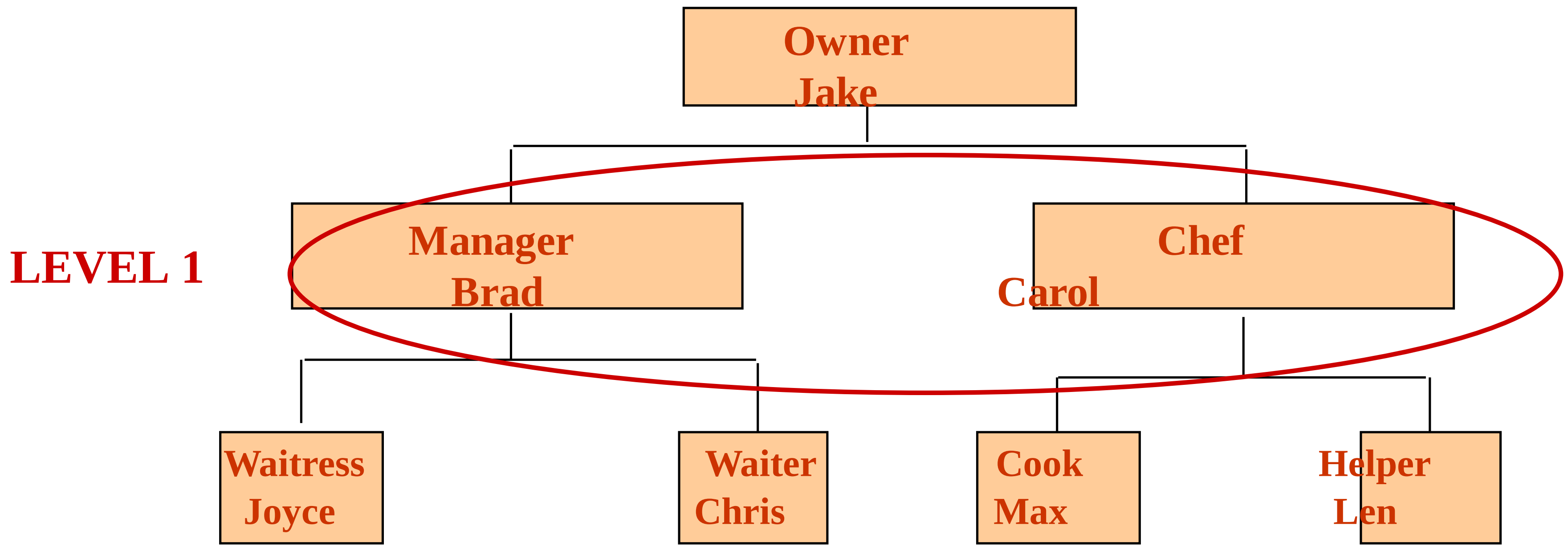
- Ancestor of a node: any node on the path from the root to that node
- Descendant of a node: any node on a path from the node to the last node in the path
- Level (depth) of a node: number of edges in the path from the root to that node
- Height of a tree: number of levels (warning: some books define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



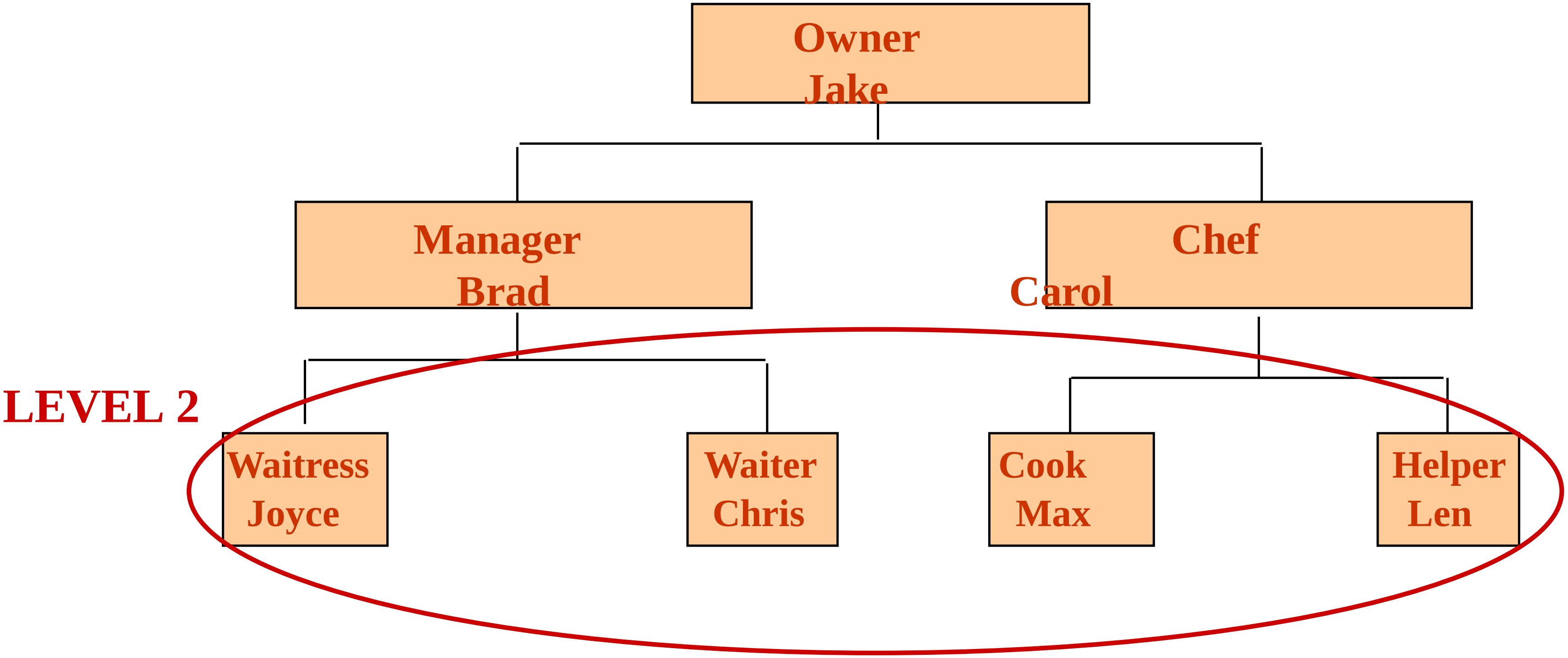
A Tree Has Levels



Level One

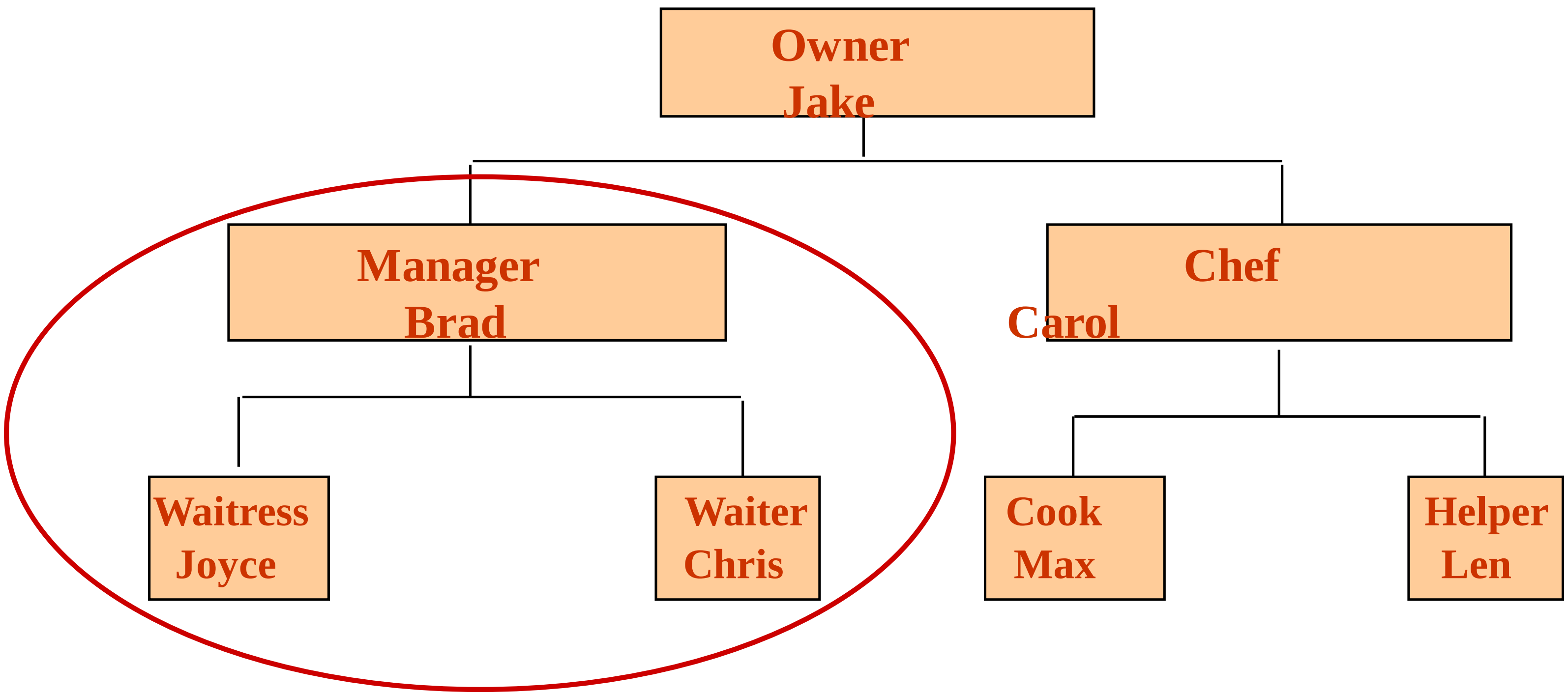

Owner
Jake
LEVEL 1
Manager
Brad
Chef
Carol
Waitress
Joyce
Waiter
Chris
Cook
Max
Helper
Len

Level Two



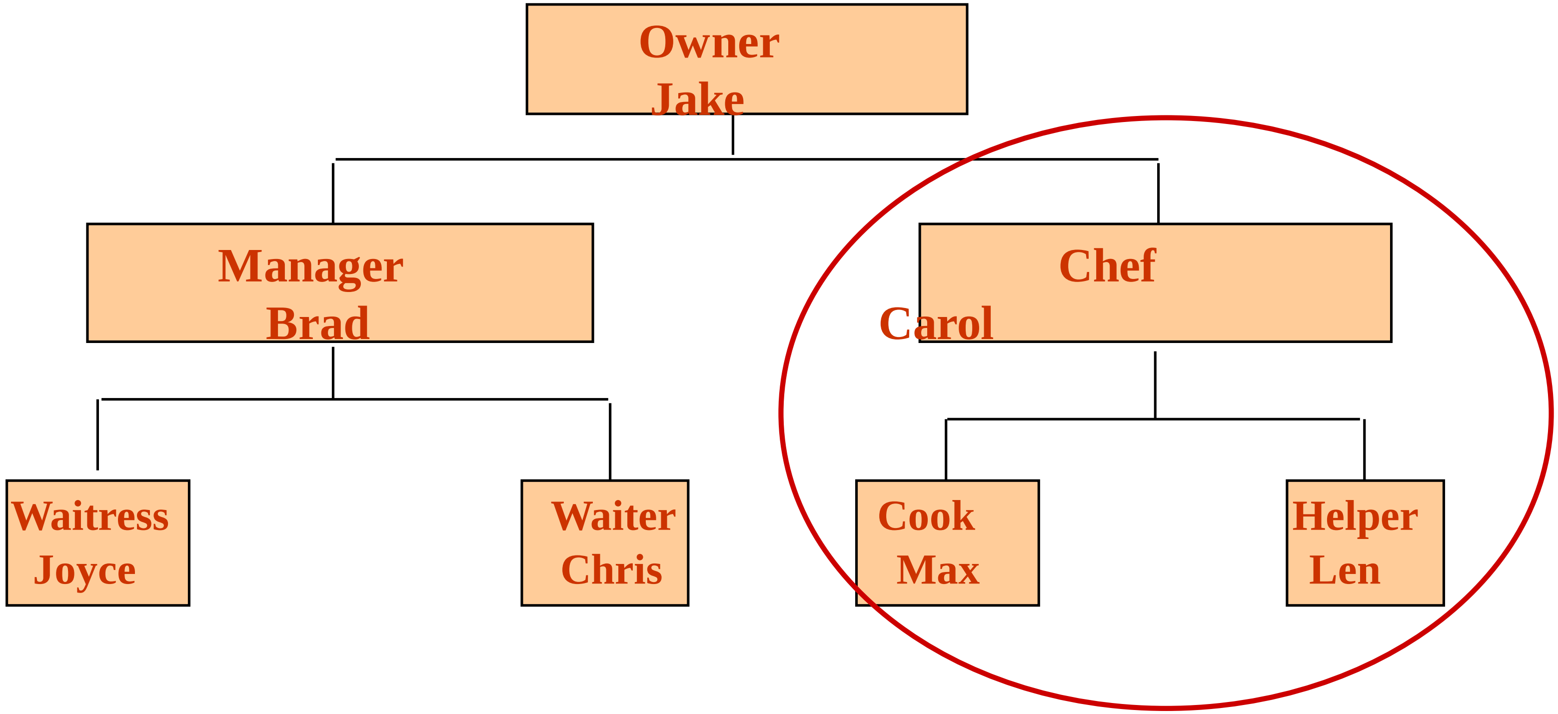
Owner
Jake
Manager
Brad
Chef
Carol
LEVEL 2
Waitress
Joyce
Waiter
Chris
Cook
Max
Helper
Len

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 $\quad 2^l$

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

l=0 l=1 l=h-1

using the geometric series:

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

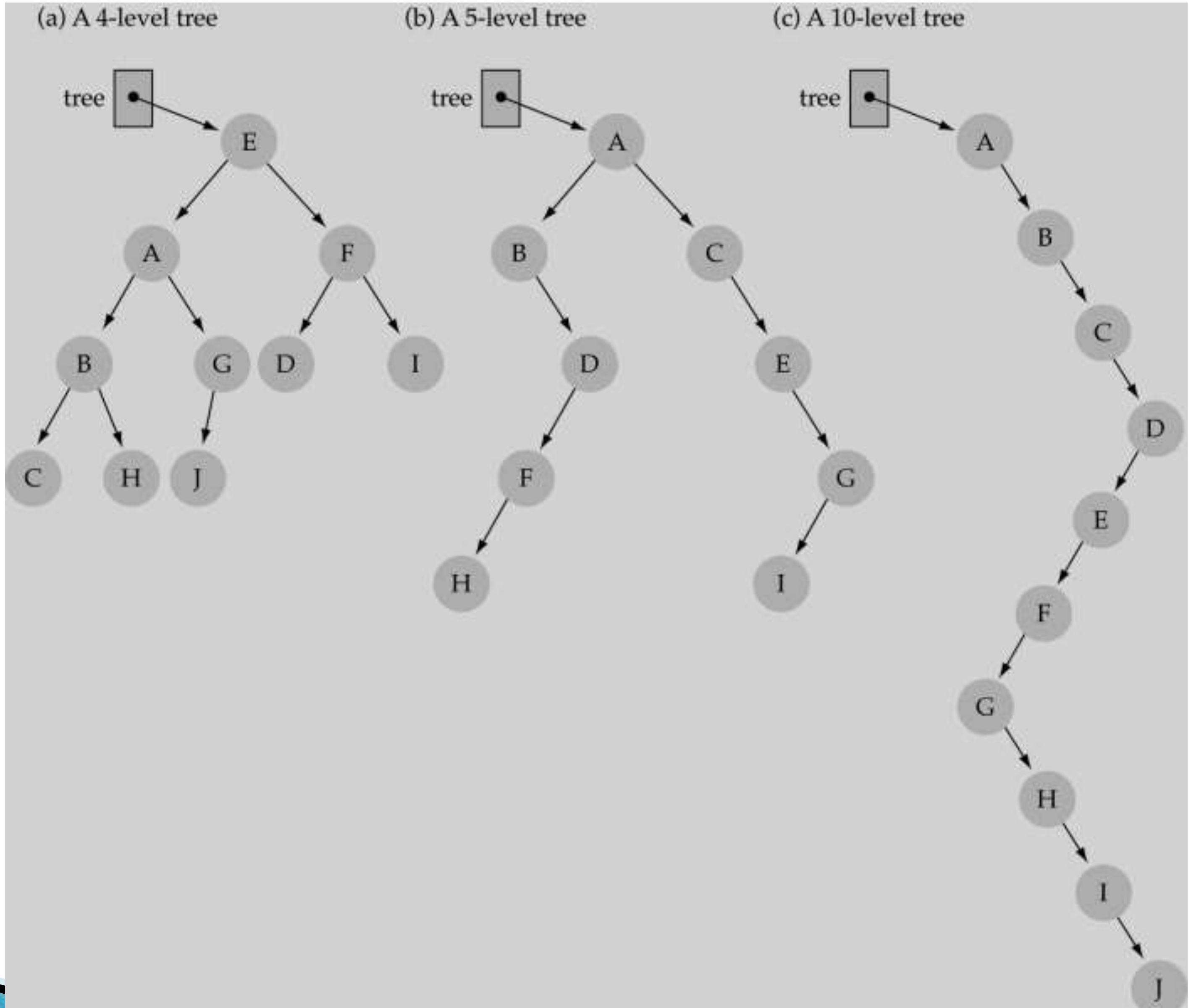
What is the height h of a <u>full tree</u> with N nodes?

$$2^h - 1 = N$$

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

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

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


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

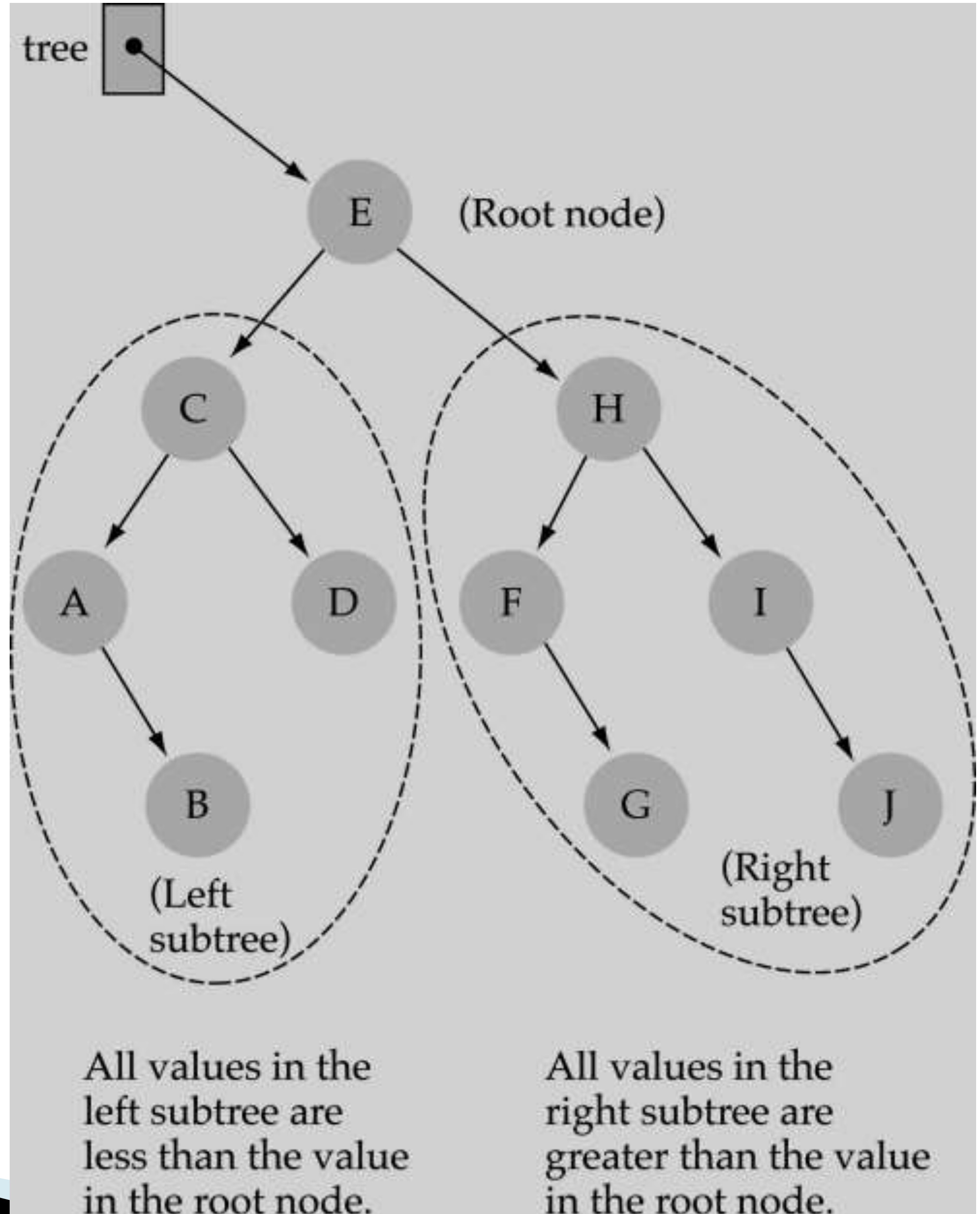
($O(N)$ time in worst case)

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

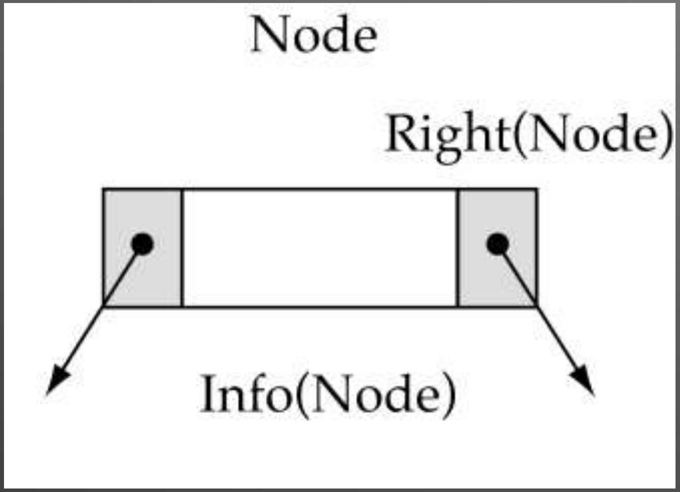
(5) If it is greater than the value stored at the root, then search the right subtree

(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

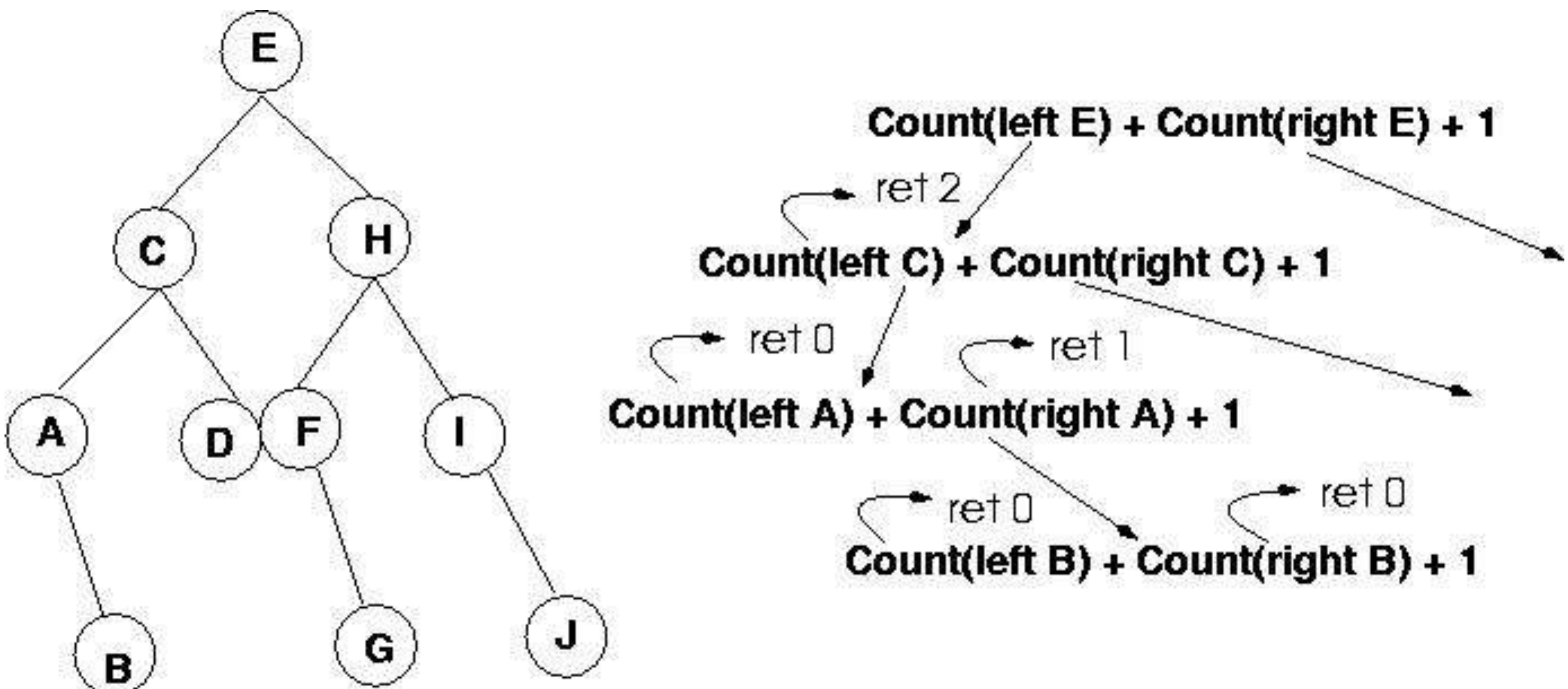
- Recursive implementation
  #nodes in a tree =
  #nodes in left subtree + #nodes in right subtree + 1
- What is the size factor?
  Number of nodes in the tree we are examining
- What is the base case?
  The tree is empty
- What is the general case?
  CountNodes(Left(tree)) + CountNodes(Right(tree)) + 1

Function NumberOfNodes (cont.)

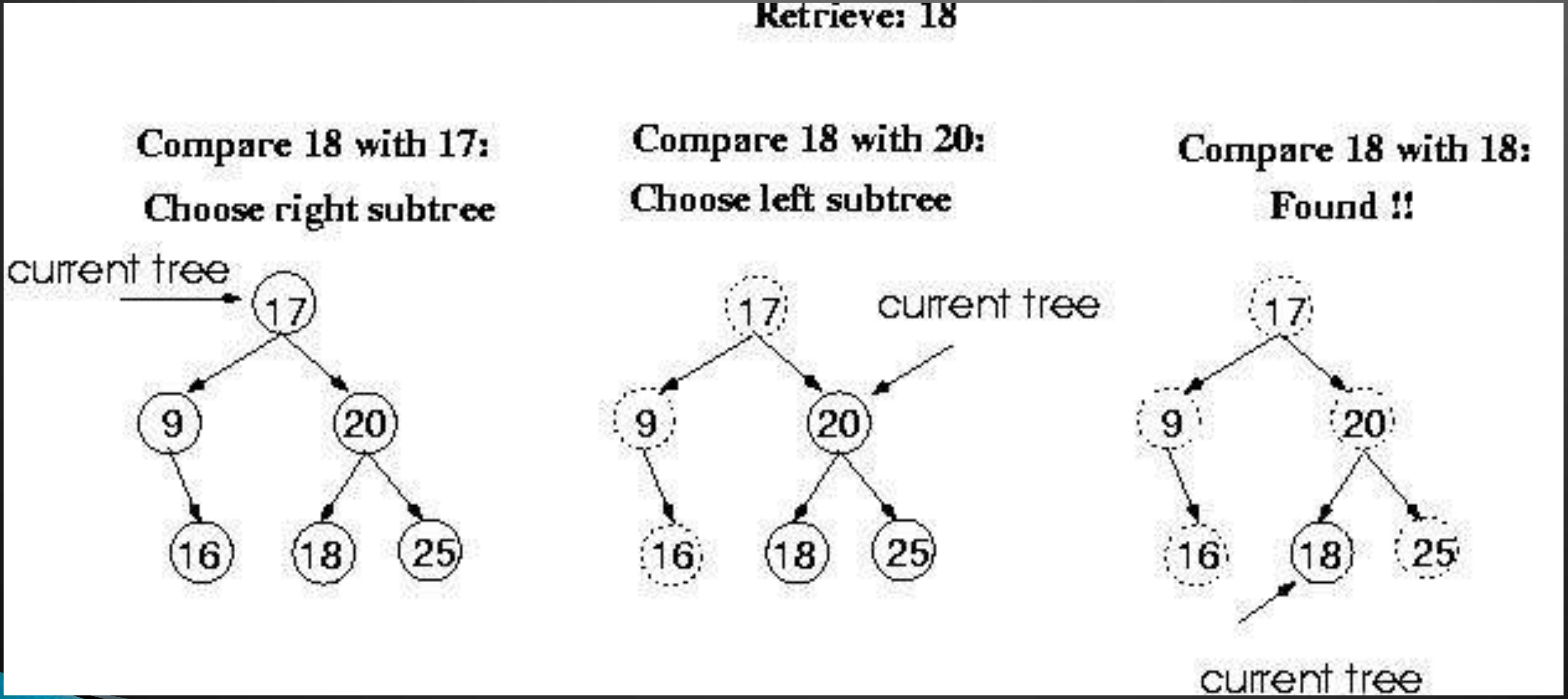
```
template<class ItemType>
int TreeType<ItemType>::NumberOfNodes() const
{
 return CountNodes(root);
}
```

```
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 RetrieveItem



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

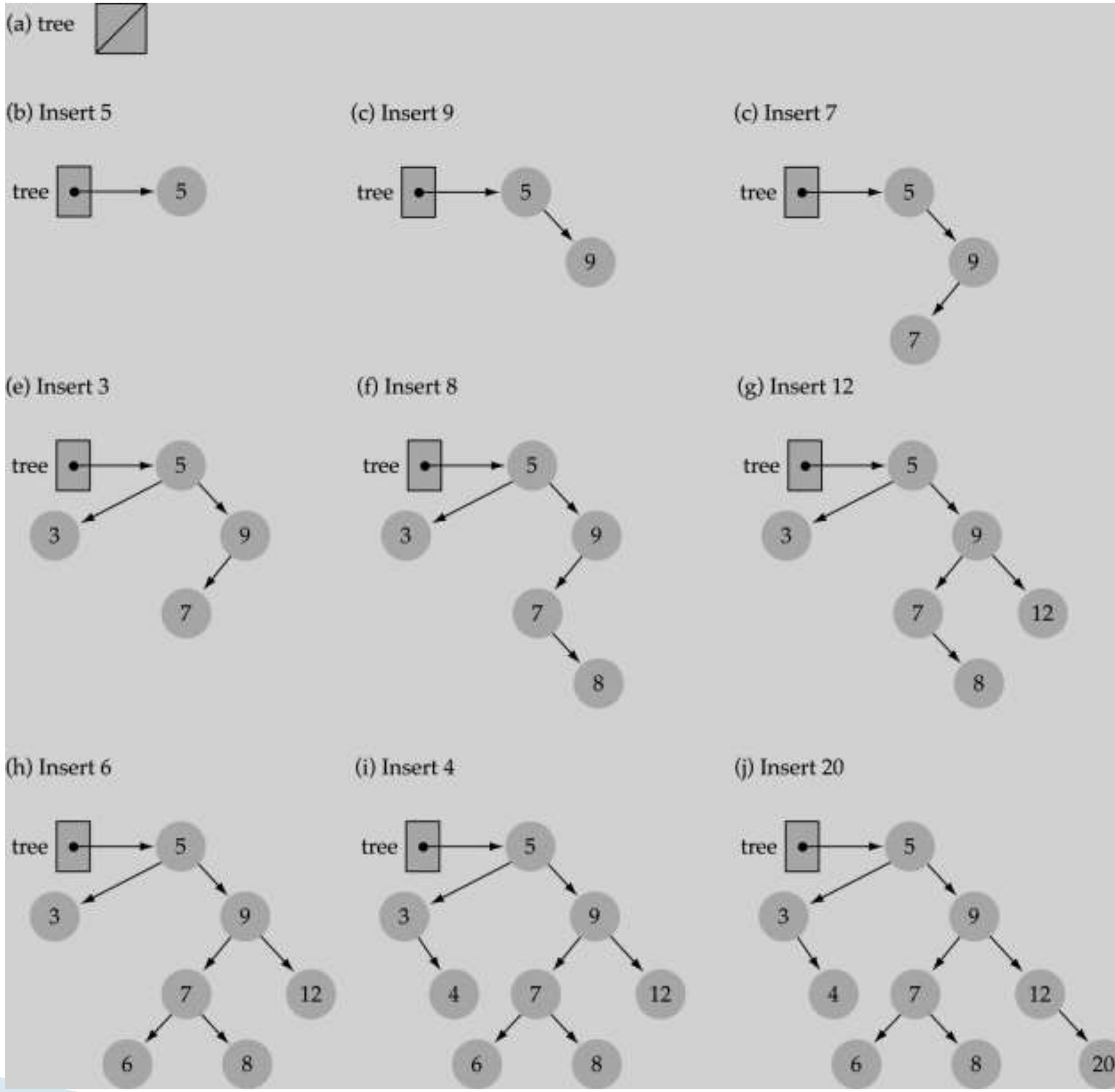
Function RetrieveItem (cont.)

```
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

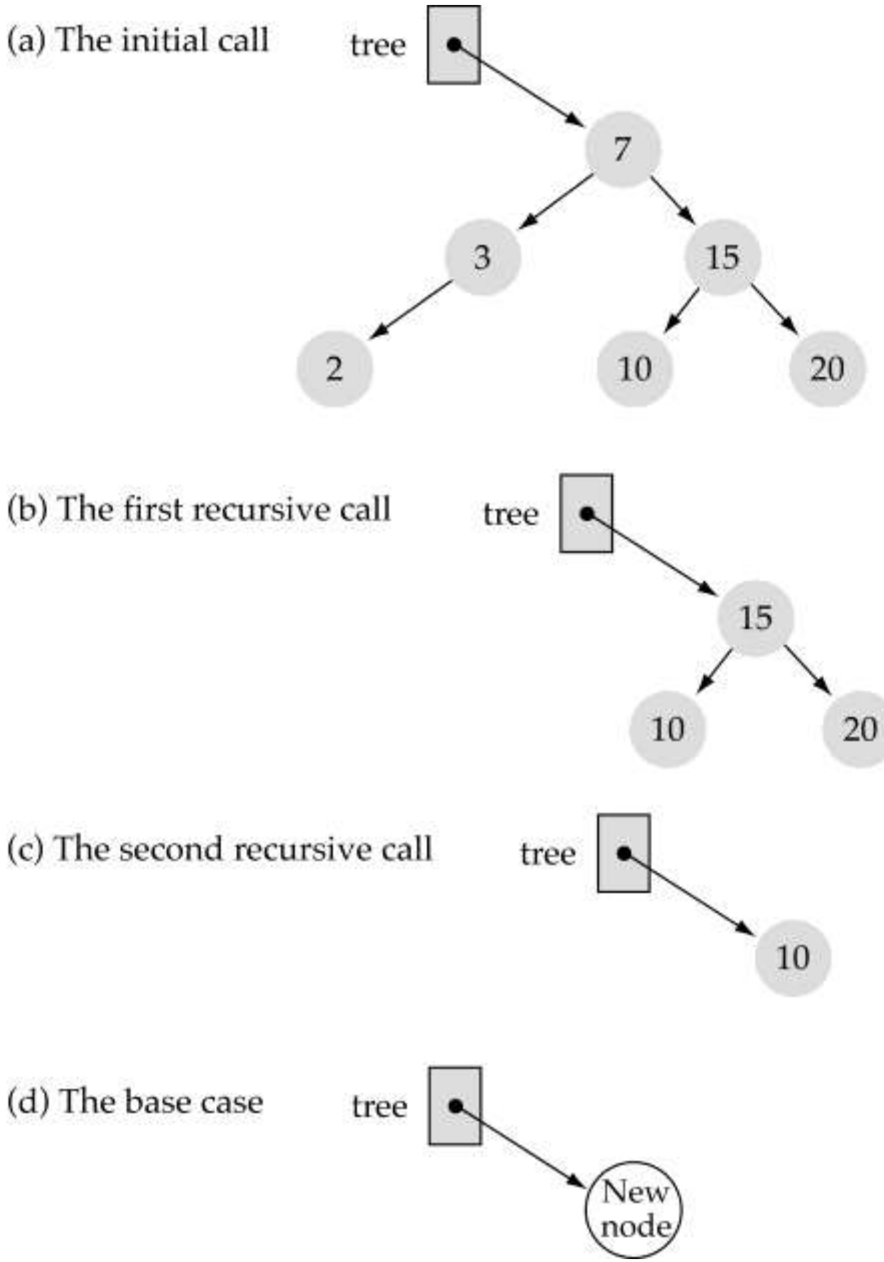
- Use the binary search tree property to insert the new item at the correct place



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

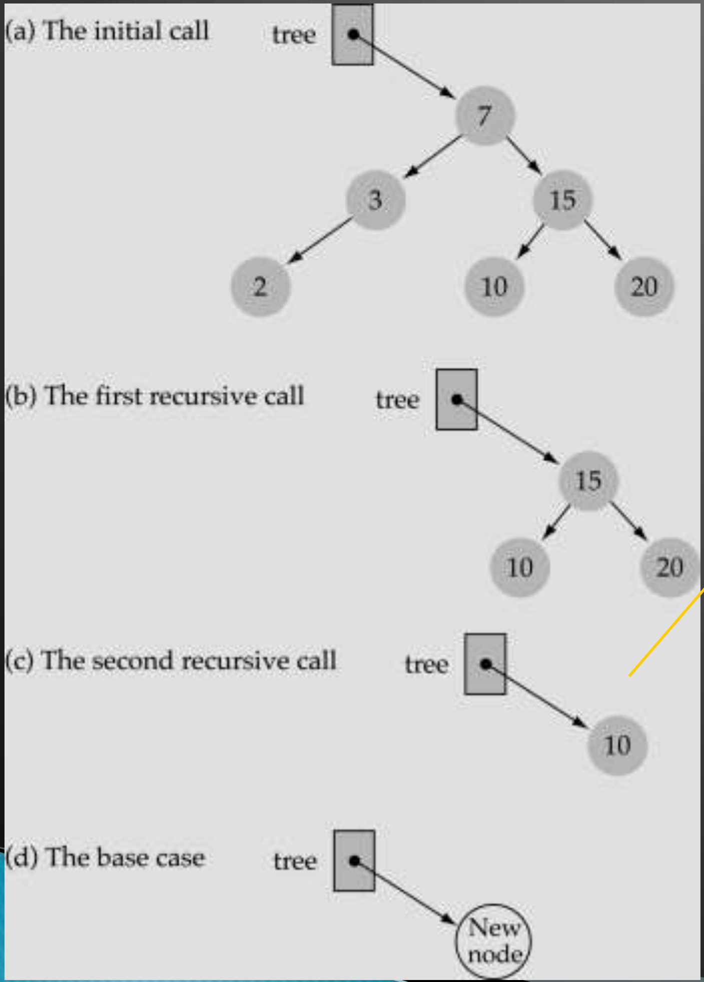
Function InsertItem (cont.)

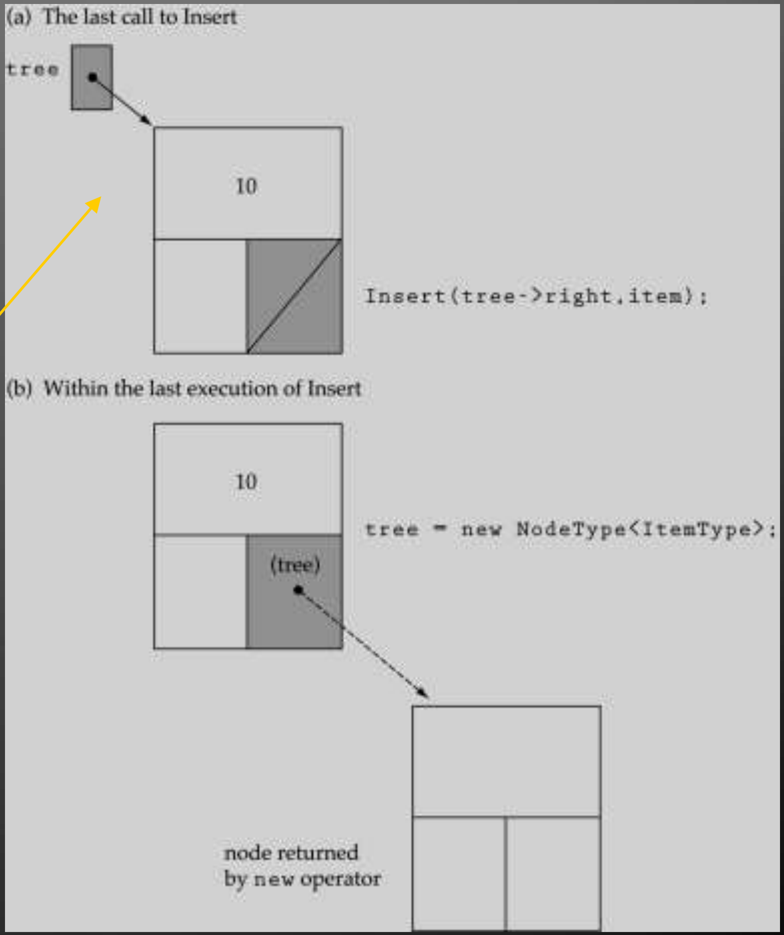
```
template<class ItemType>
void TreeType<ItemType>::InsertItem(ItemType item)
{
 Insert(root, item);
}

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

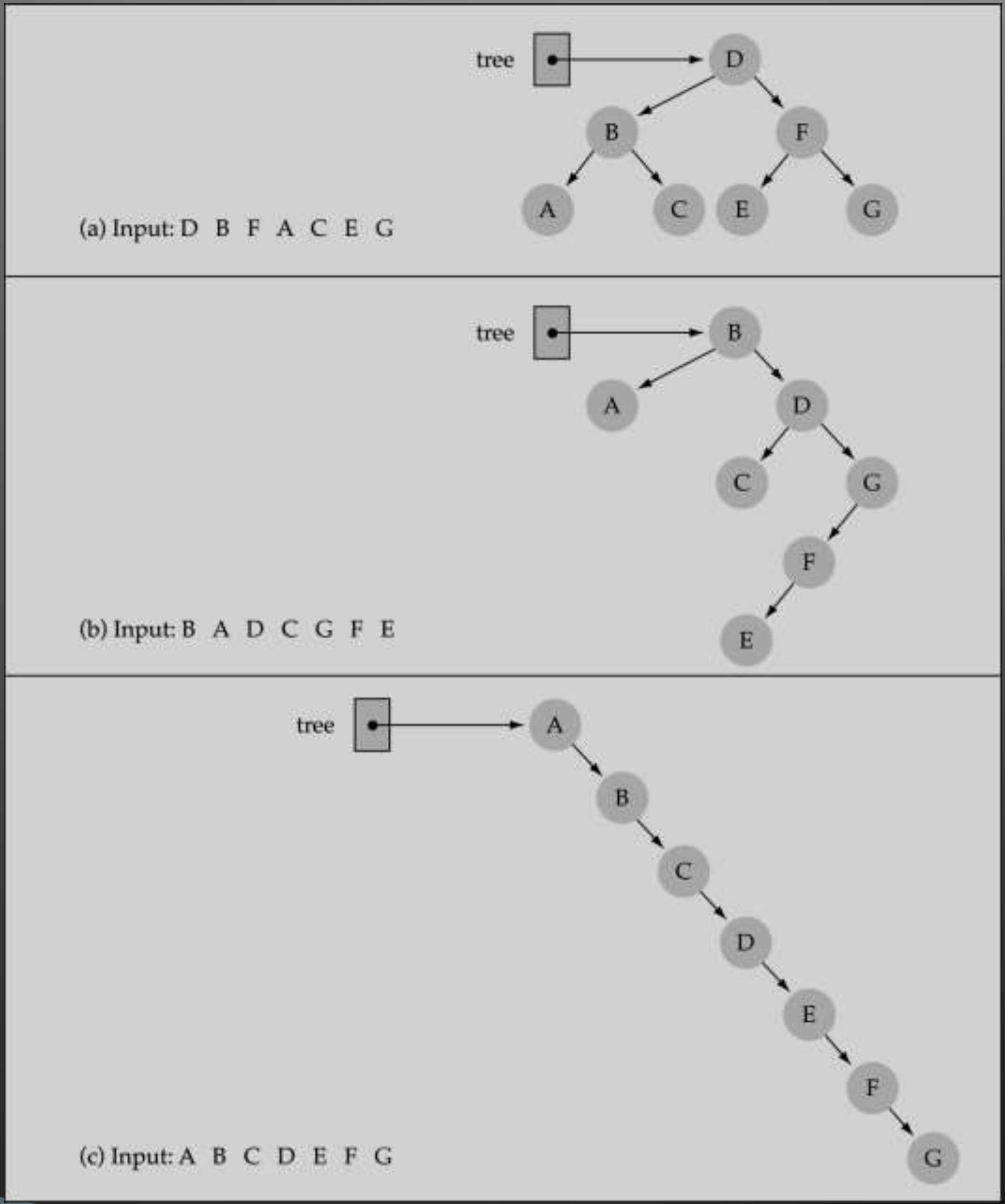




Does the order of inserting elements into a tree matter?

- 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

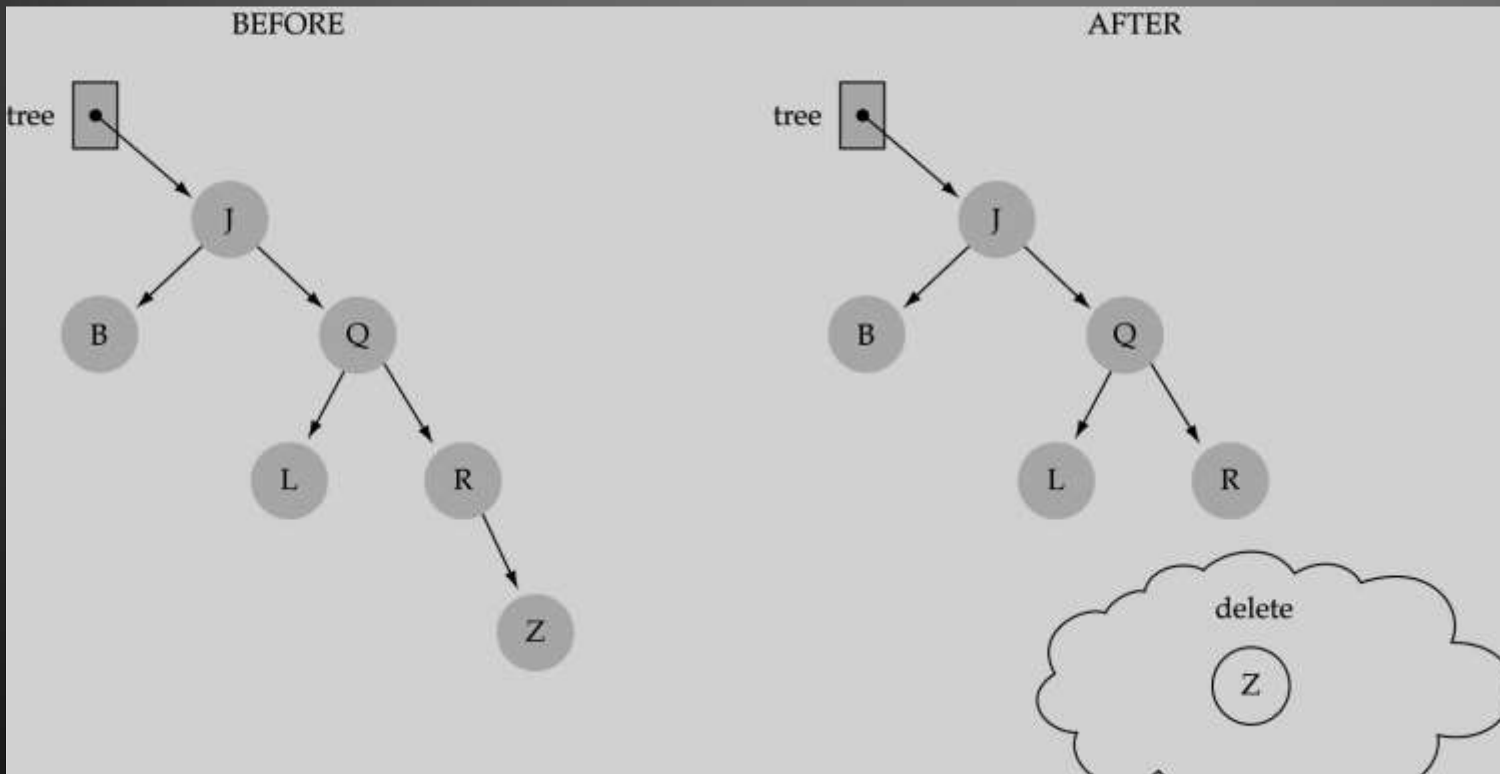
- First, find the item; then, delete it
- Important: binary search tree property must be preserved!!
- We need to consider three different cases:

  (1) Deleting a leaf

  (2) Deleting a node with only one child

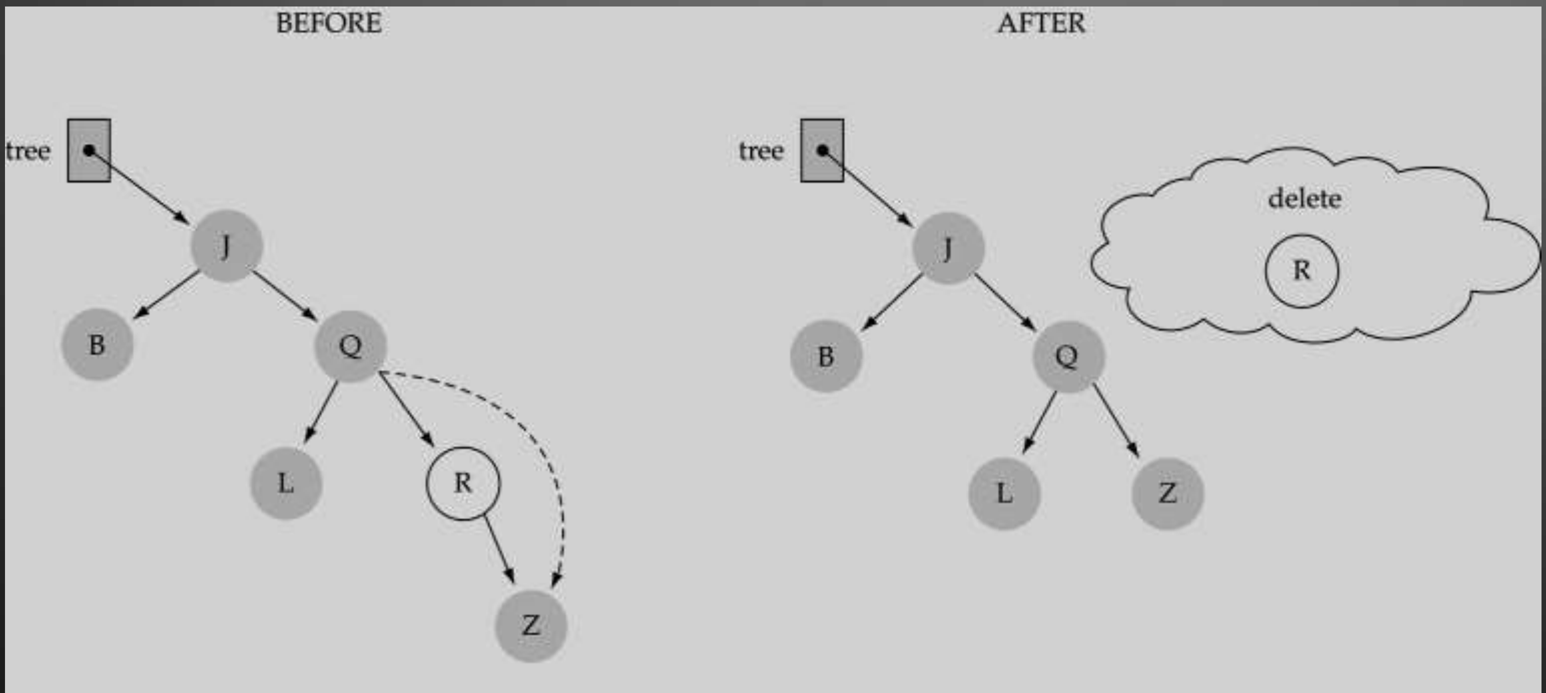
  (3) Deleting a node with two children

(1) Deleting a leaf



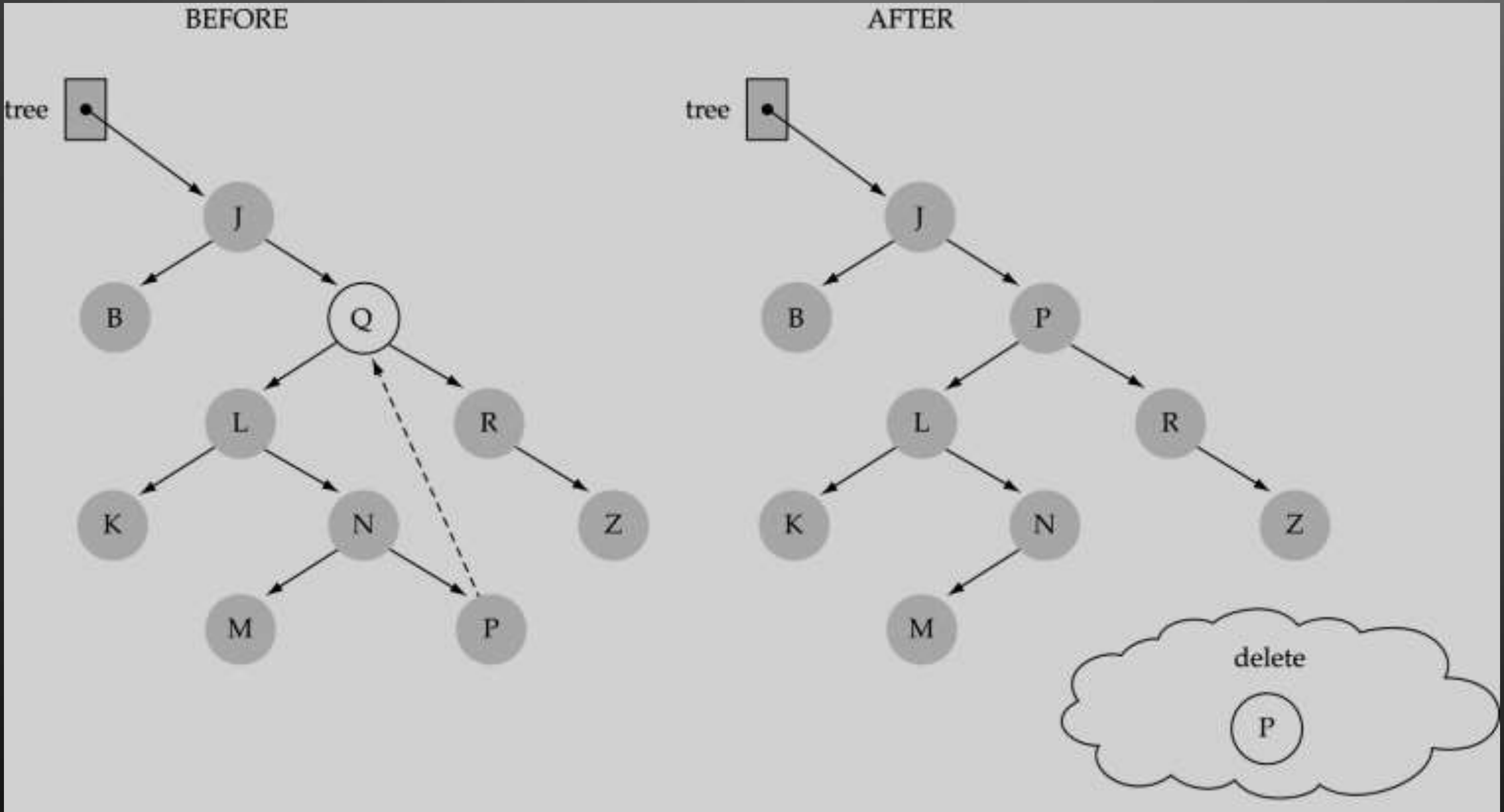
Delete the node containing Z

(2) Deleting a node with only one child



Delete the node containing R

(3) Deleting a node with two children



Delete the node containing Q

(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.)

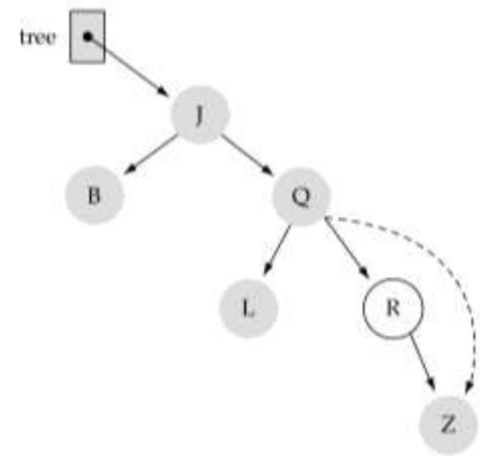
```
template <class ItemType>
void DeleteNode(TreeNode<ItemType>*& tree)
{
 ItemType data;
 TreeNode<ItemType>* tempPtr;

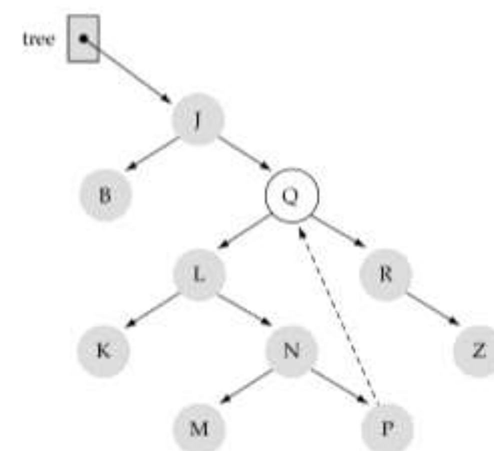
 tempPtr = tree;
 if(tree->left == NULL) { //right child
   tree = tree->right;
   delete tempPtr;
 }
 else if(tree->right == NULL) { // left child
   tree = tree->left;
   delete tempPtr;
 }
 else {
   GetPredecessor(tree->left, data);
   tree->info = data;
   Delete(tree->left, data);
 }
}
```

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

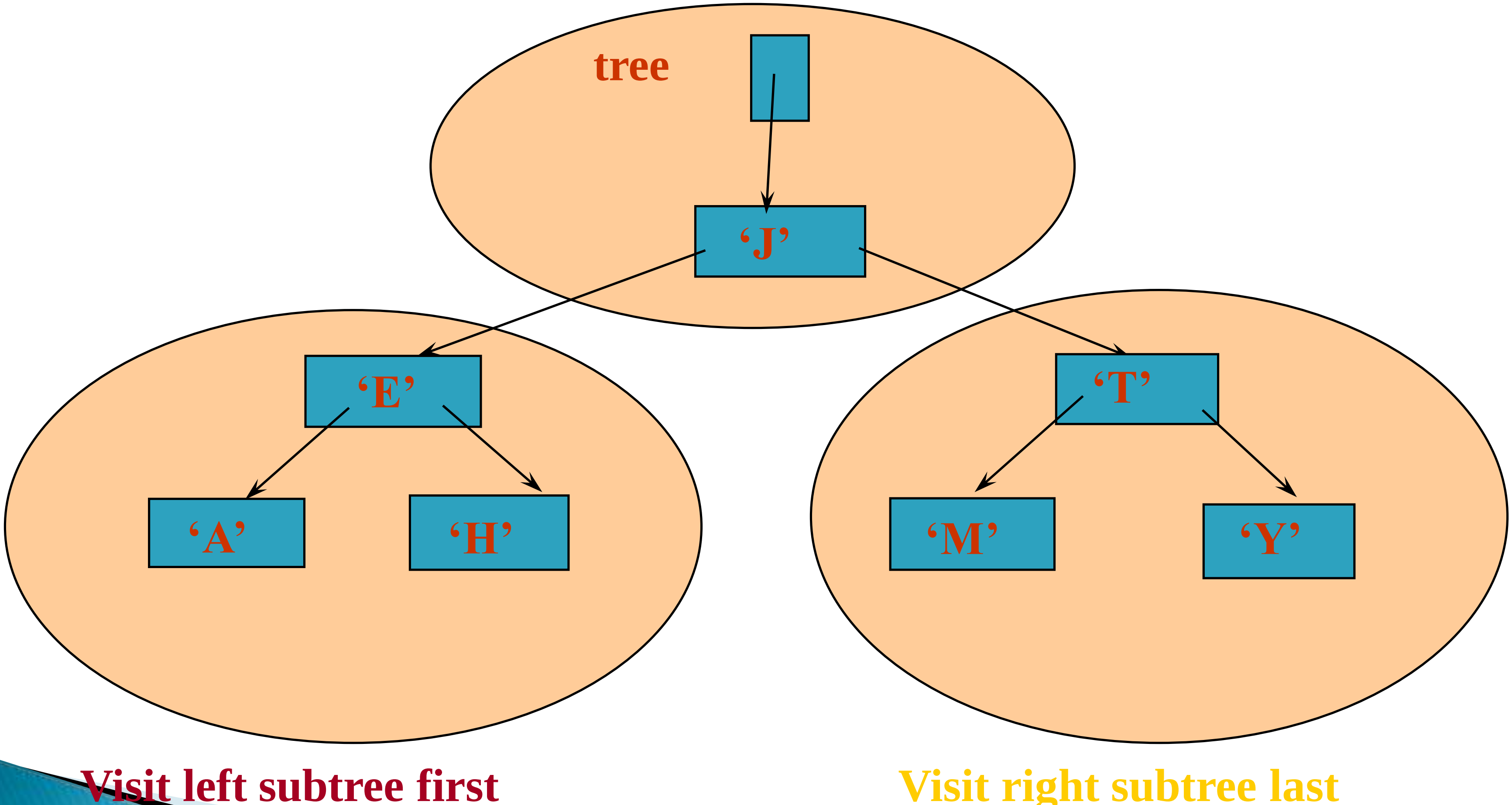
There are mainly three ways to traverse a tree:

Inorder Traversal

Postorder Traversal

Preorder Traversal

Inorder Traversal: A E H J M T Y



Inorder Traversal

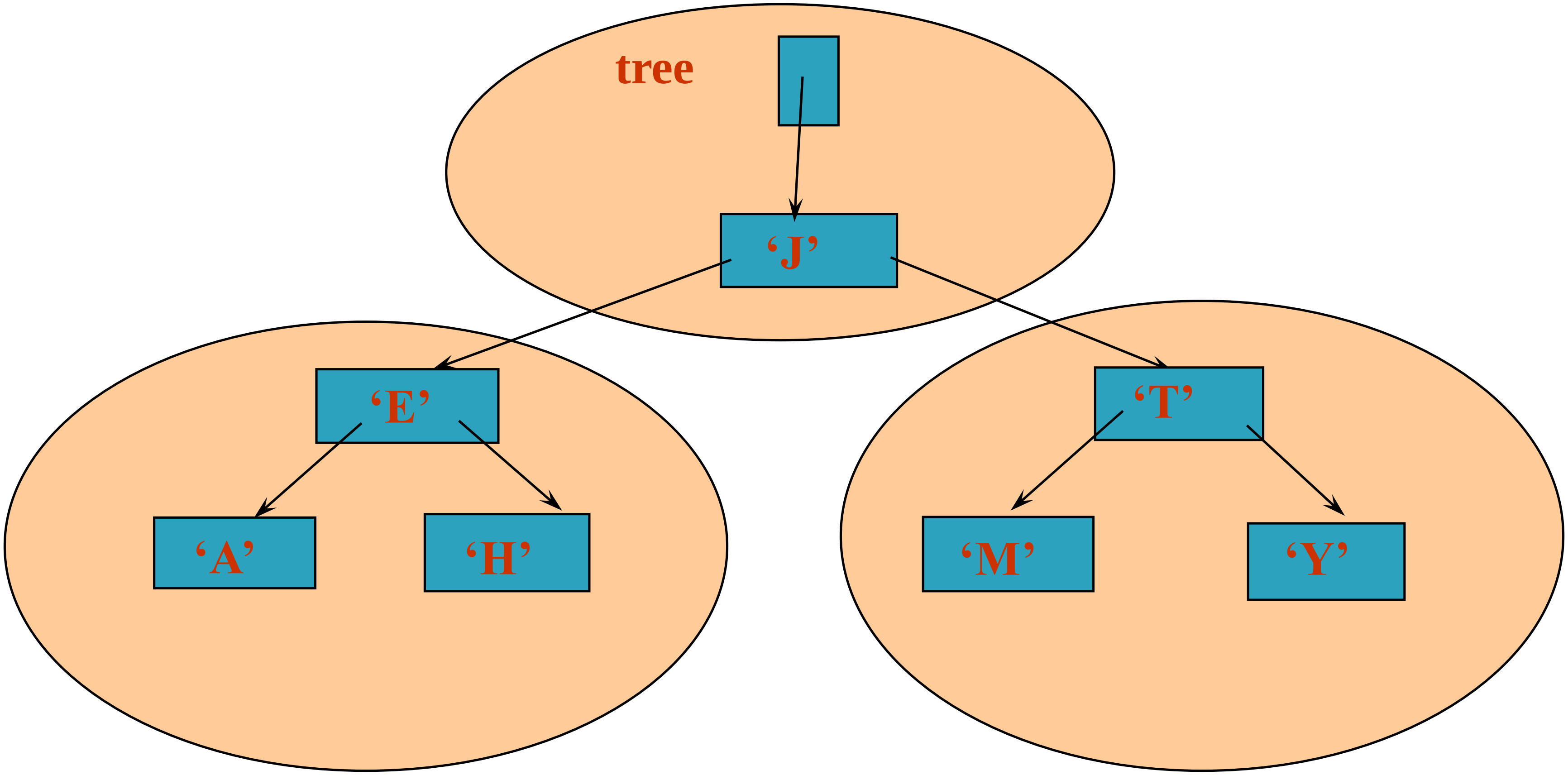
- Visit the nodes in the left subtree, then visit the root of the tree, then visit the nodes in the right subtree

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

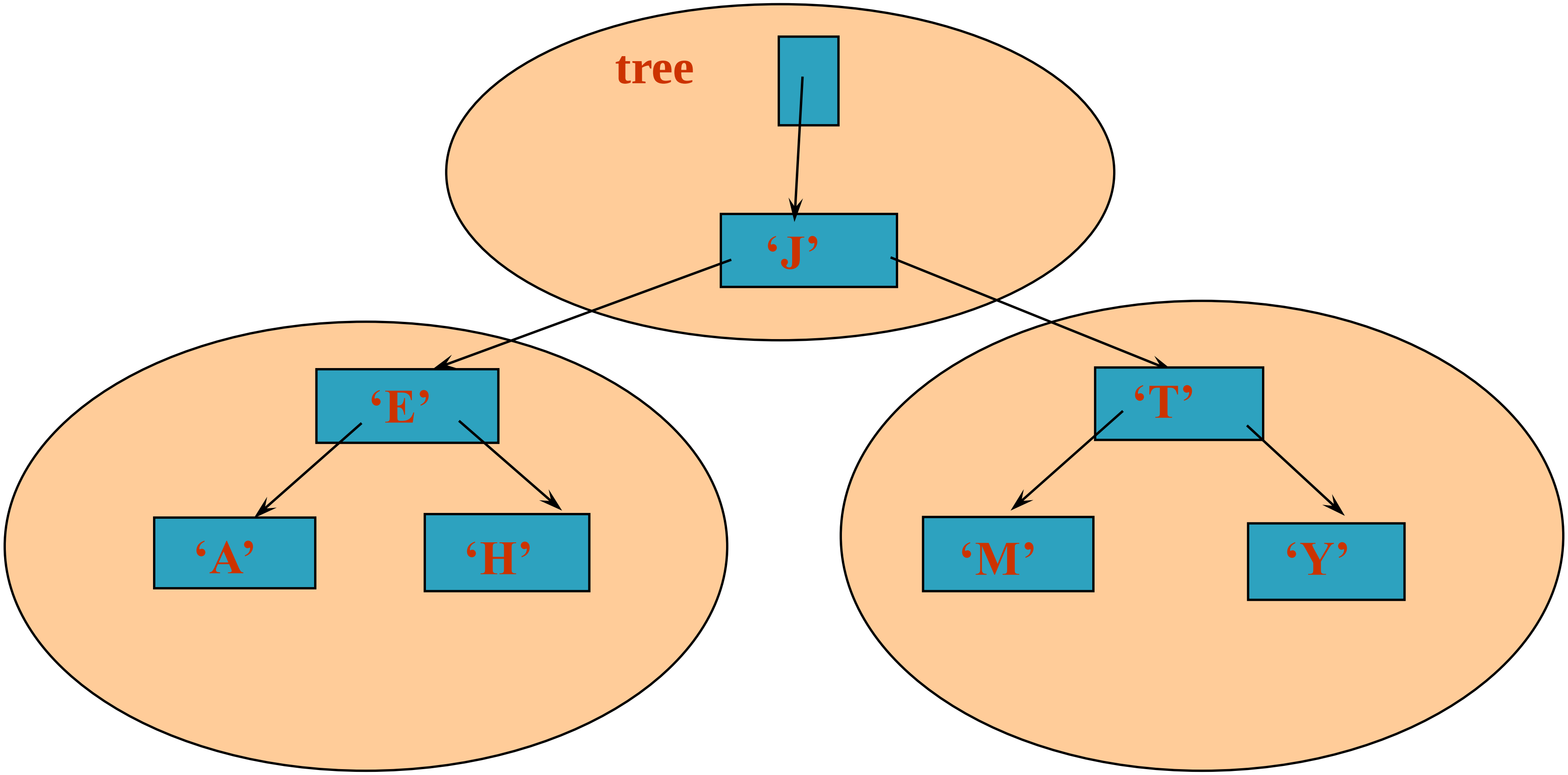
Visit right subtree second

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

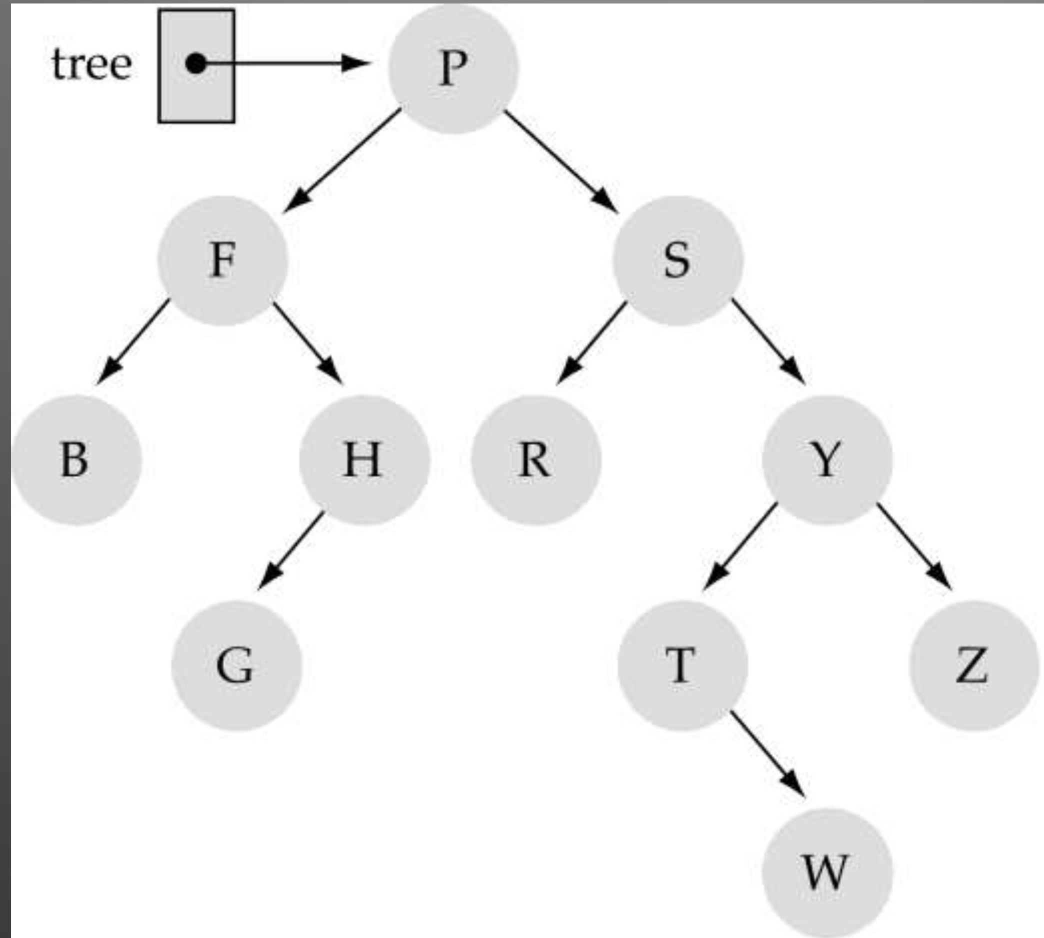
Visit right subtree last

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))
```

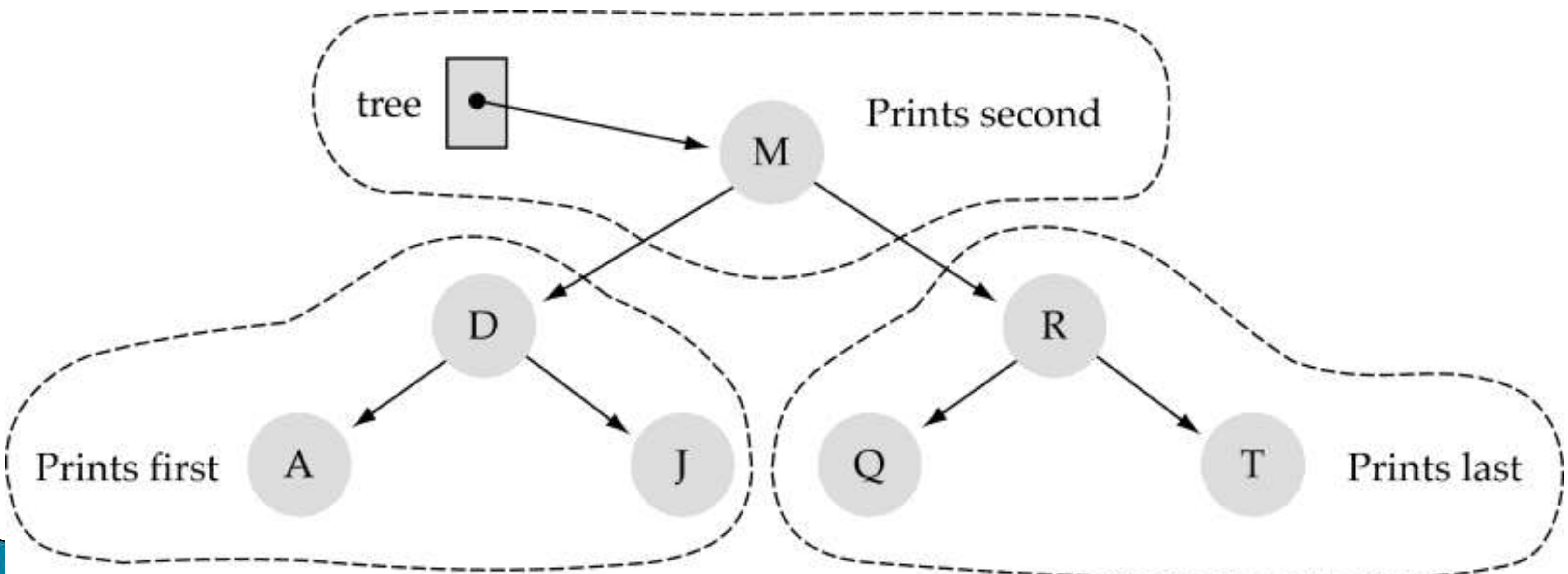
Tree Traversals



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?? (keys are printed out in ascending order!!)
- *Hint*: use binary search trees for sorting !!



Function PrintTree (cont.)

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

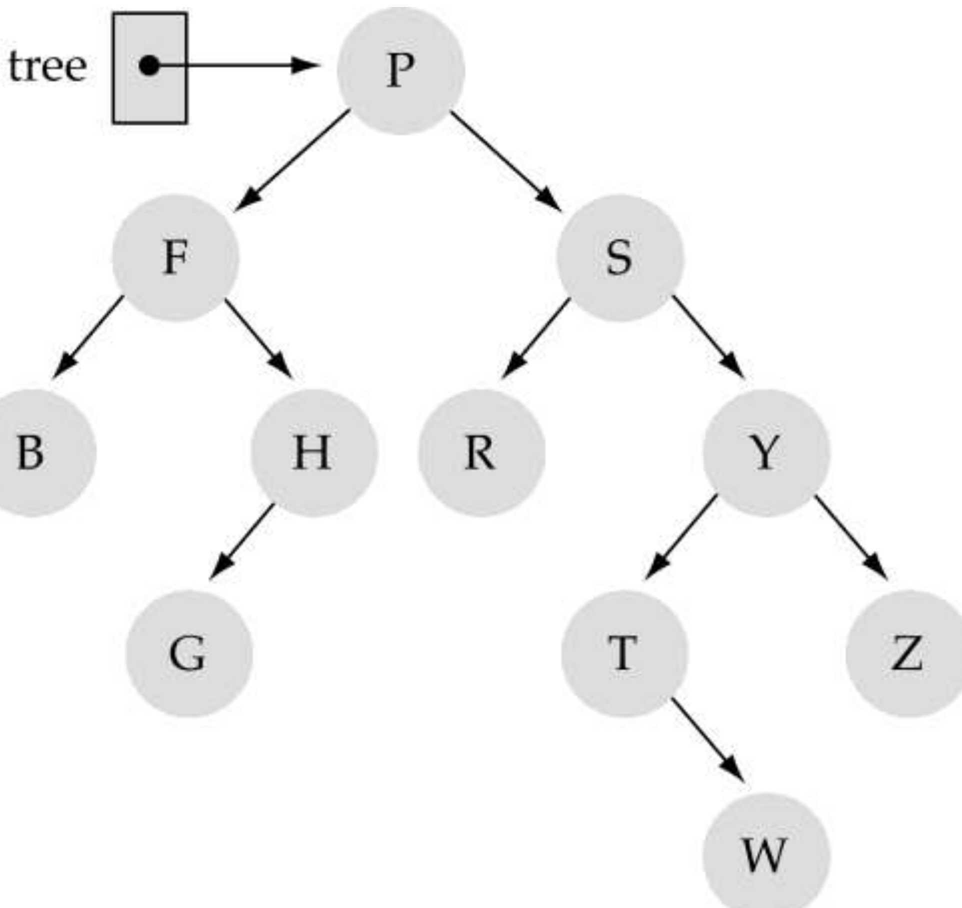
template<class ItemType>
void Print(TreeNode<ItemType>* tree, ofstream& outFile)
{
 if(tree != NULL) {
   Print(tree->left, outFile);
   outFile << tree->info;
   Print(tree->right, outFile);
 }
}
```

(see textbook for overloading << and >>)

Class Constructor

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

Class Destructor



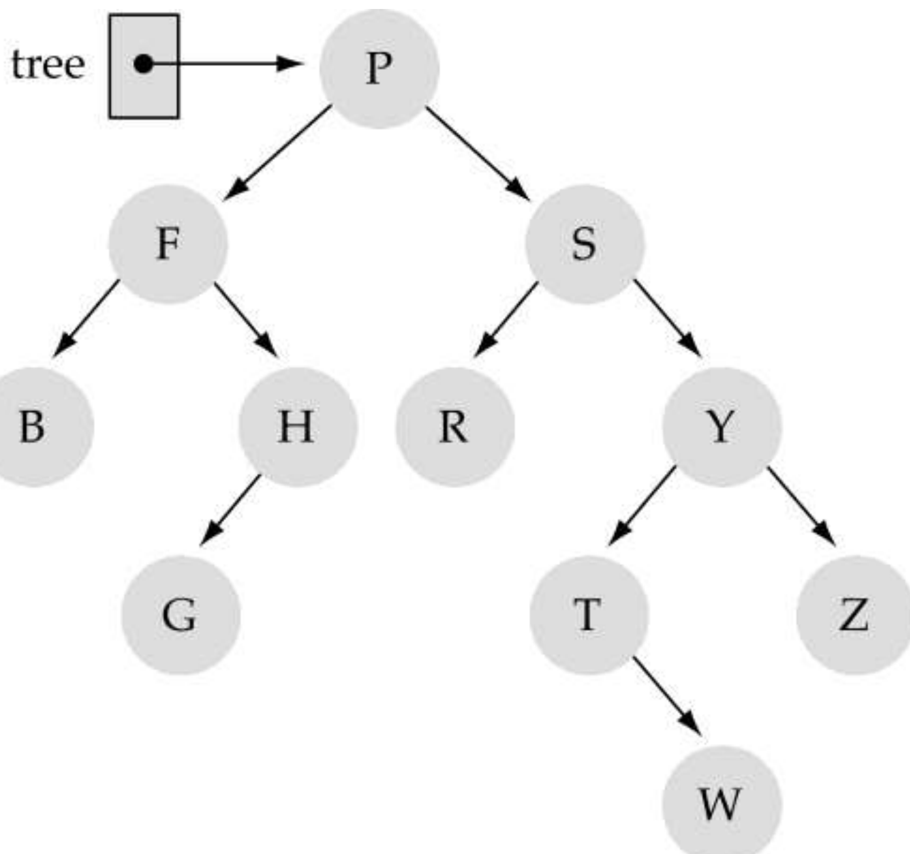
Class Destructor (cont'd)

- 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



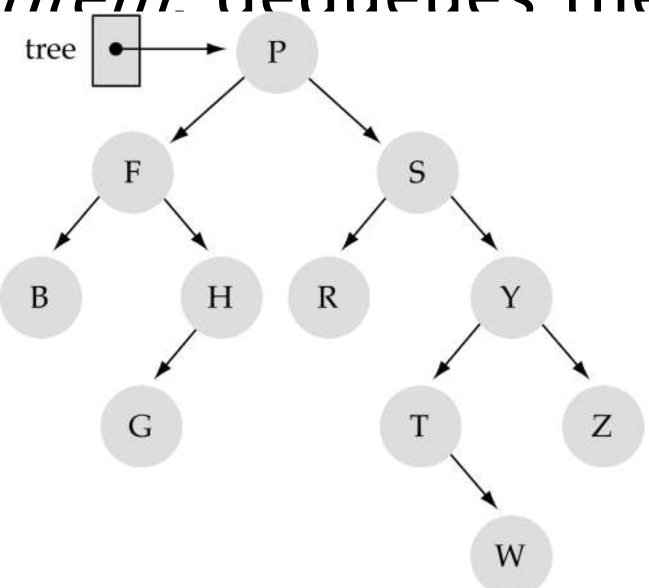
Copy Constructor (cont'd)

```
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* dequeues 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

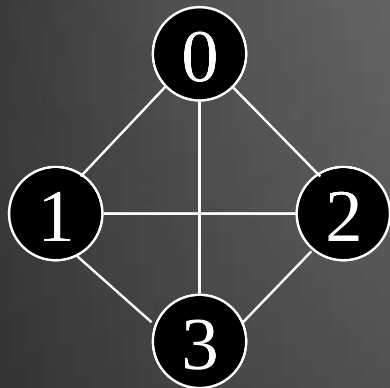
GRAPHS

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, $<v_0, v_1> \; != \; <v_1,v_0>$

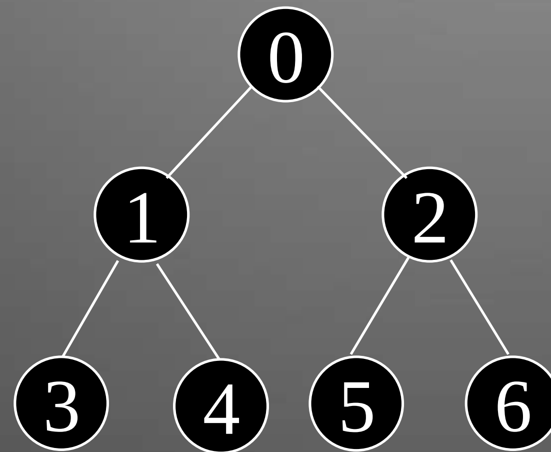
tail $\longrightarrow$ head

Examples for Graph



G_1

complete graph



G_2

incomplete graph

G_3

$V(G_1)=\{0,1,2,3\}$ $E(G_1)=\{(0,1),(0,2),(0,3),(1,2),(1,3),(2,3)\}$

$V(G_2)=\{0,1,2,3,4,5,6\}$ $E(G_2)=\{(0,1),(0,2),(1,3),(1,4),(2,5),(2,6)\}$

$V(G_3)=\{0,1,2\}$ $E(G_3)=\{<0,1>,<1,0>,<1,2>\}$

complete undirected graph: n(n-1)/2 edges

complete directed graph: n(n-1) edges

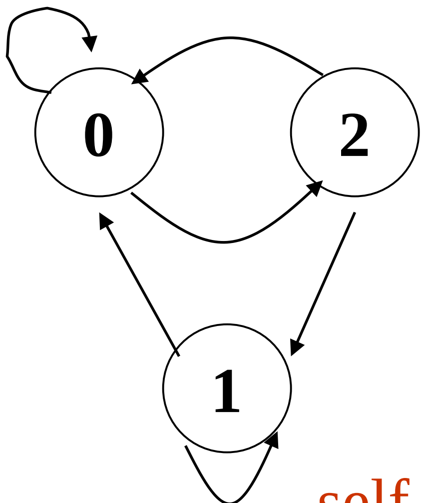
Complete Graph

- 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

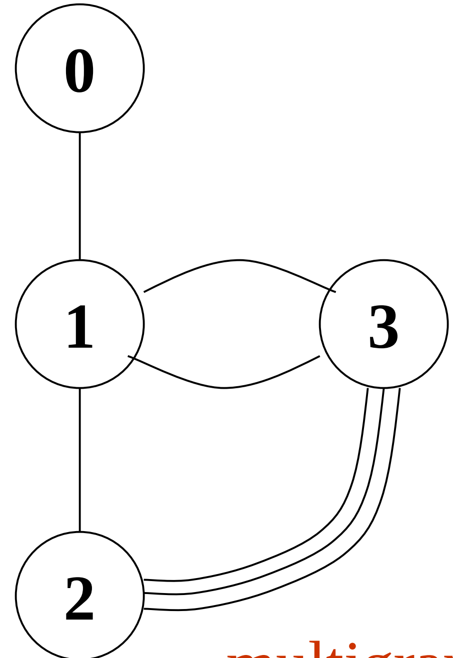
- If (v_0, v_1) is an edge in an undirected graph,
  – v_0 and v_1 are adjacent
  – The edge (v_0, v_1) is incident on vertices v_0 and v_1
- If $<v_0, v_1>$ is an edge in a directed graph
  – v_0 is adjacent to v_1, and v_1 is adjacent from v_0
  – The edge $<v_0, v_1>$ is incident on v_0 and v_1

***Figure 6.3:**Example of a graph with feedback loops and a multigraph (p.260)



self edge

(a)



(b)

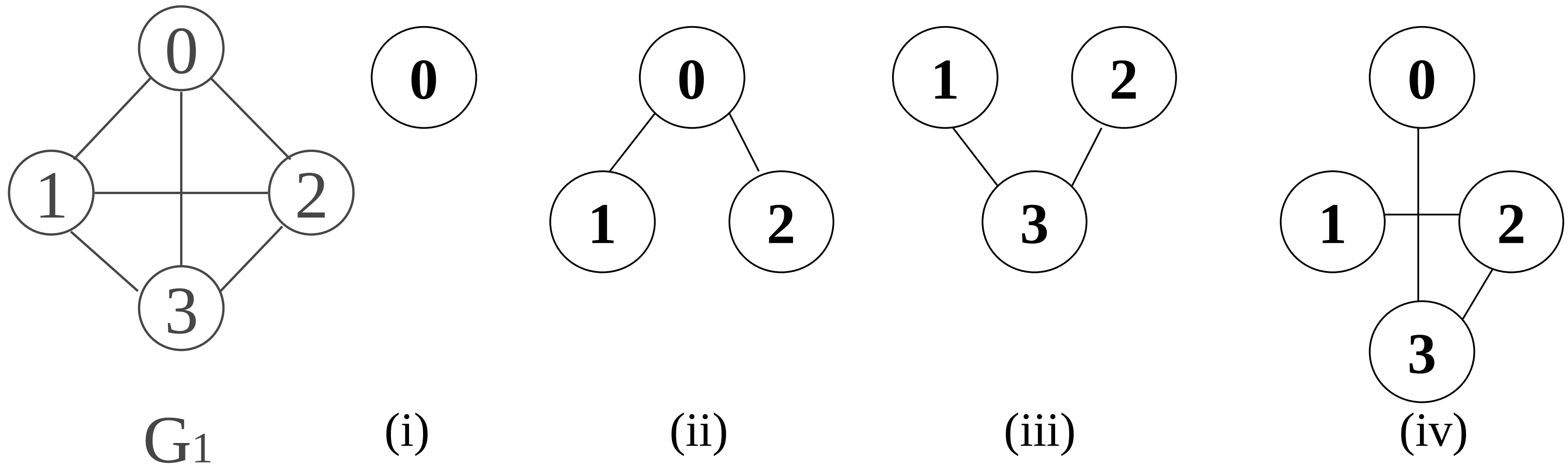
multigraph:
multiple occurrences
of the same edge

Figure 6.3

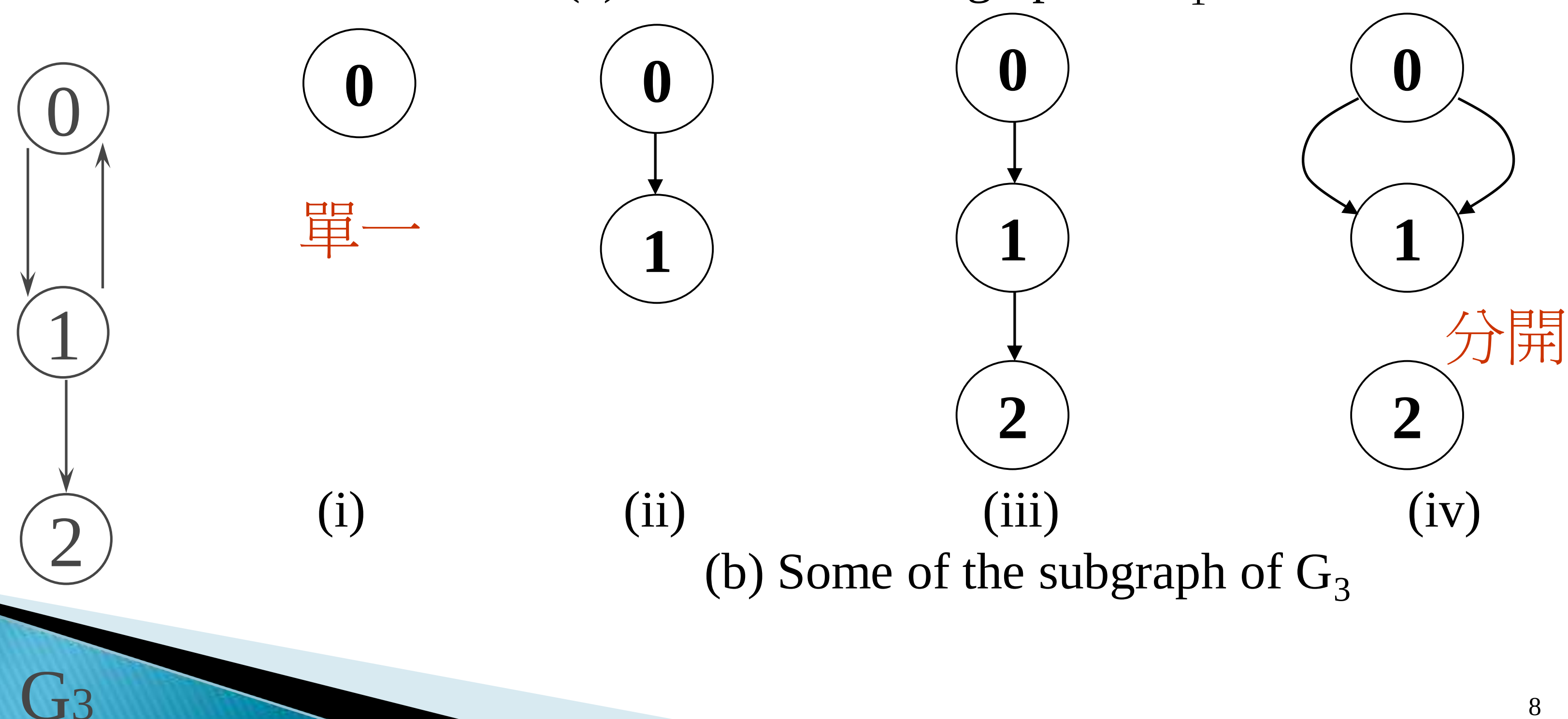
Subgraph and Path

- 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, $v_p, v_{i1}, v_{i2}, \ldots, v_{in}, v_q$, such that $(v_p, v_{i1}), (v_{i1}, v_{i2}), \ldots, (v_{in}, v_q)$ 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)



(a) Some of the subgraph of G_1

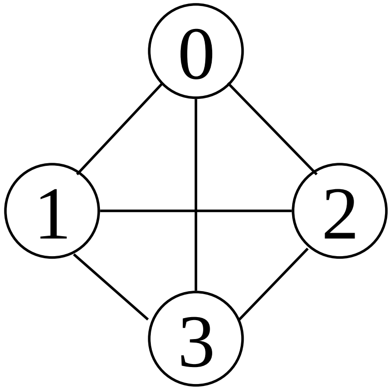


(b) Some of the subgraph of G_3

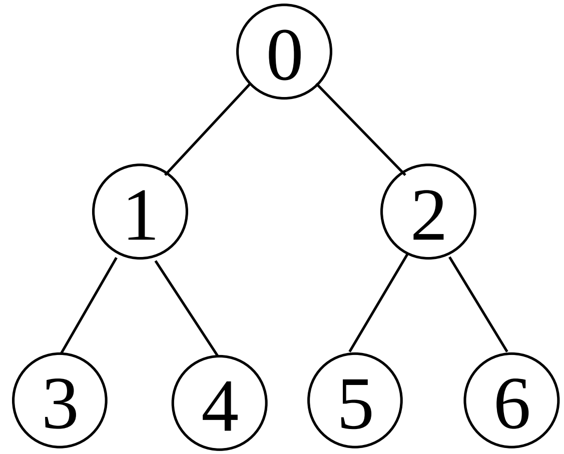
Simple Path and Style

- 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

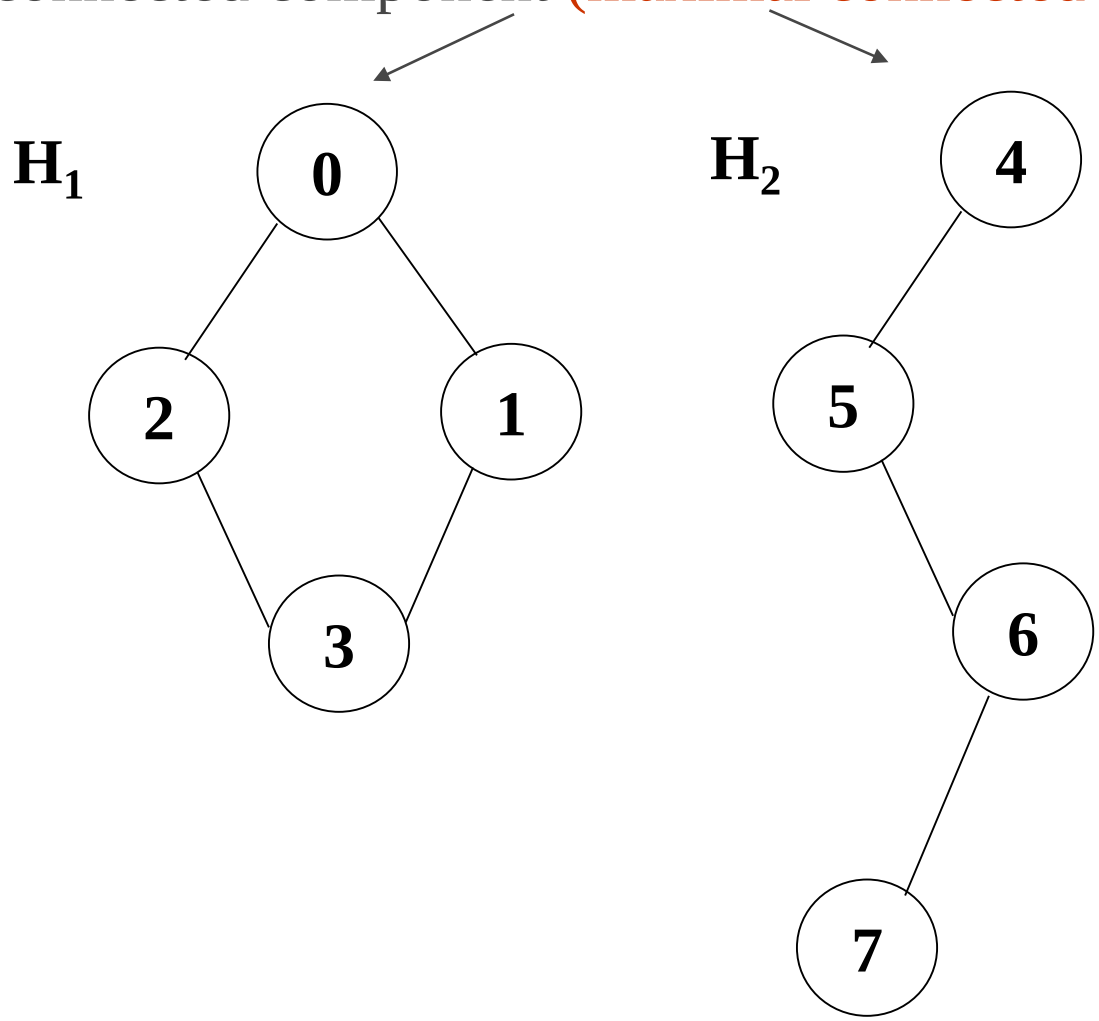
tree (acyclic graph)

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)

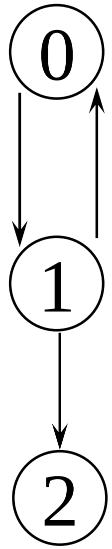
connected component (maximal connected subgraph)



G4 (not connected)

***Figure 6.6:** Strongly connected components of G_3 (p.262)*

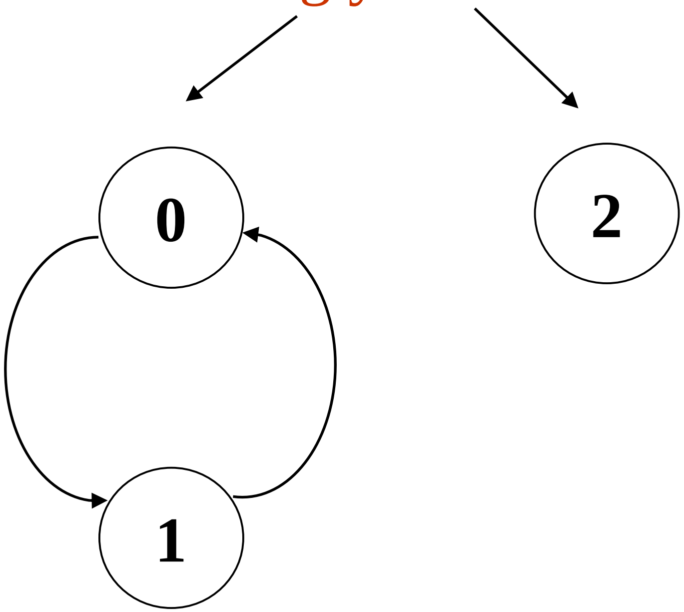
not strongly connected



G_3

strongly connected component

(maximal strongly connected subgraph)



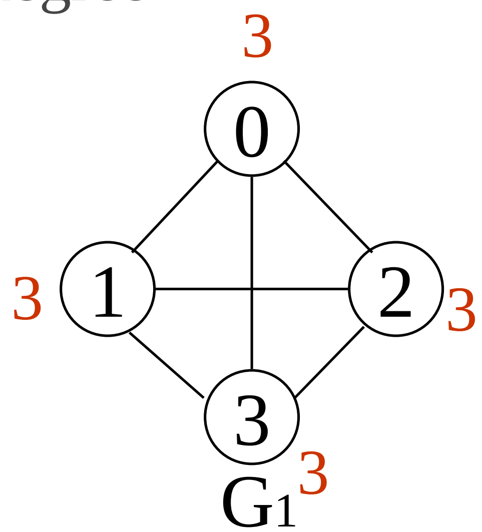
Degree

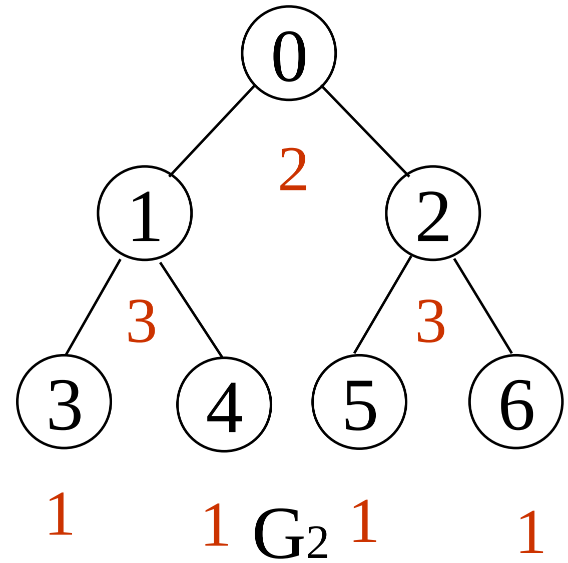
- The degree of a vertex is the number of edges incident to that vertex
- For directed graph,
  - the in-degree of a vertex *v* is the number of edges that have *v* as the head
  - the out-degree of a vertex *v* is the number of edges that have *v* as the tail
  - if d_i is the degree of a vertex *i* in a graph *G* with *n* vertices and *e* edges, the number of edges is

$$e = (\sum_{0}^{n-1} d_i)/2$$

undirected graph

degree

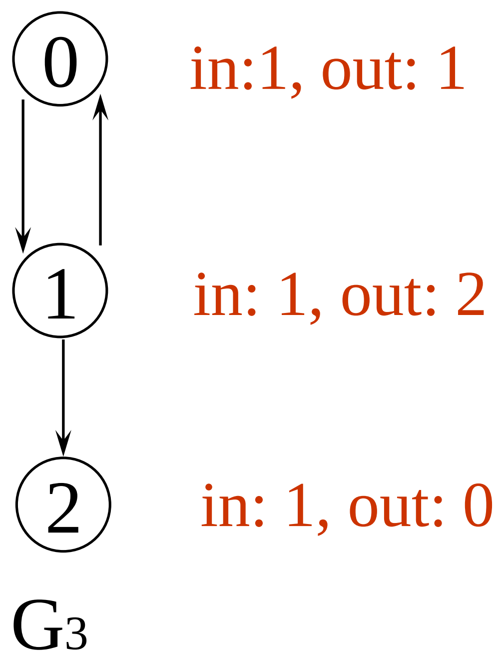




directed graph

in-degree

out-degree



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*

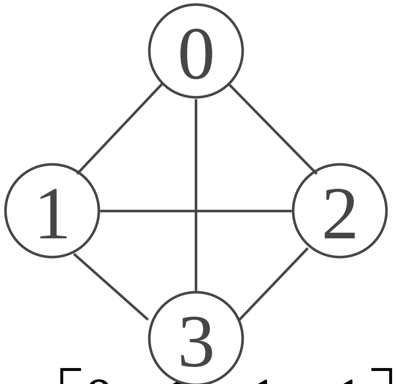
Graph Representations

- Adjacency Matrix
- Adjacency Lists

Adjacency Matrix

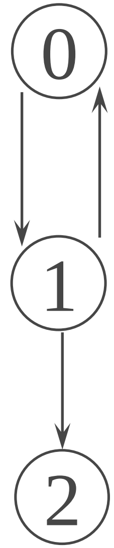
- Let G=(V,E) be a graph with n vertices.
- The adjacency matrix of G is a two-dimensional n by n array, say adj_mat
- If the edge (v_i, v_j) is in E(G), adj_mat[i][j]=1
- If there is no such edge in E(G), adj_mat[i][j]=0
- The adjacency matrix for an undirected graph is symmetric; the adjacency matrix for a digraph need not be symmetric

Examples for Adjacency Matrix



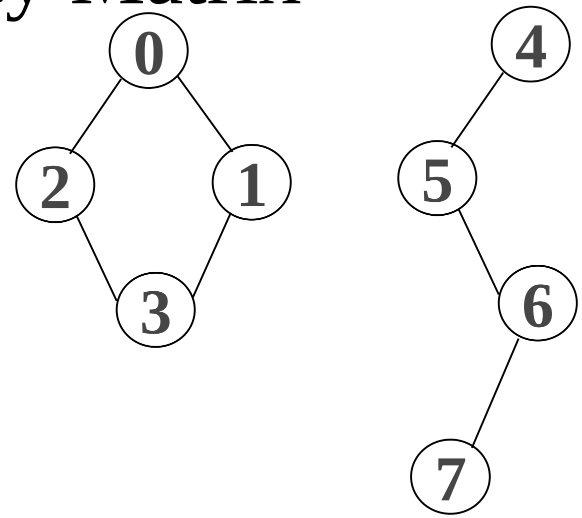
$$\begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix}$$

G_1



$$\begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix}$$

G_2



$$\begin{bmatrix} 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

G_4

symmetric

undirected: $n^2/2$

directed: n^2

Merits of Adjacency Matrix

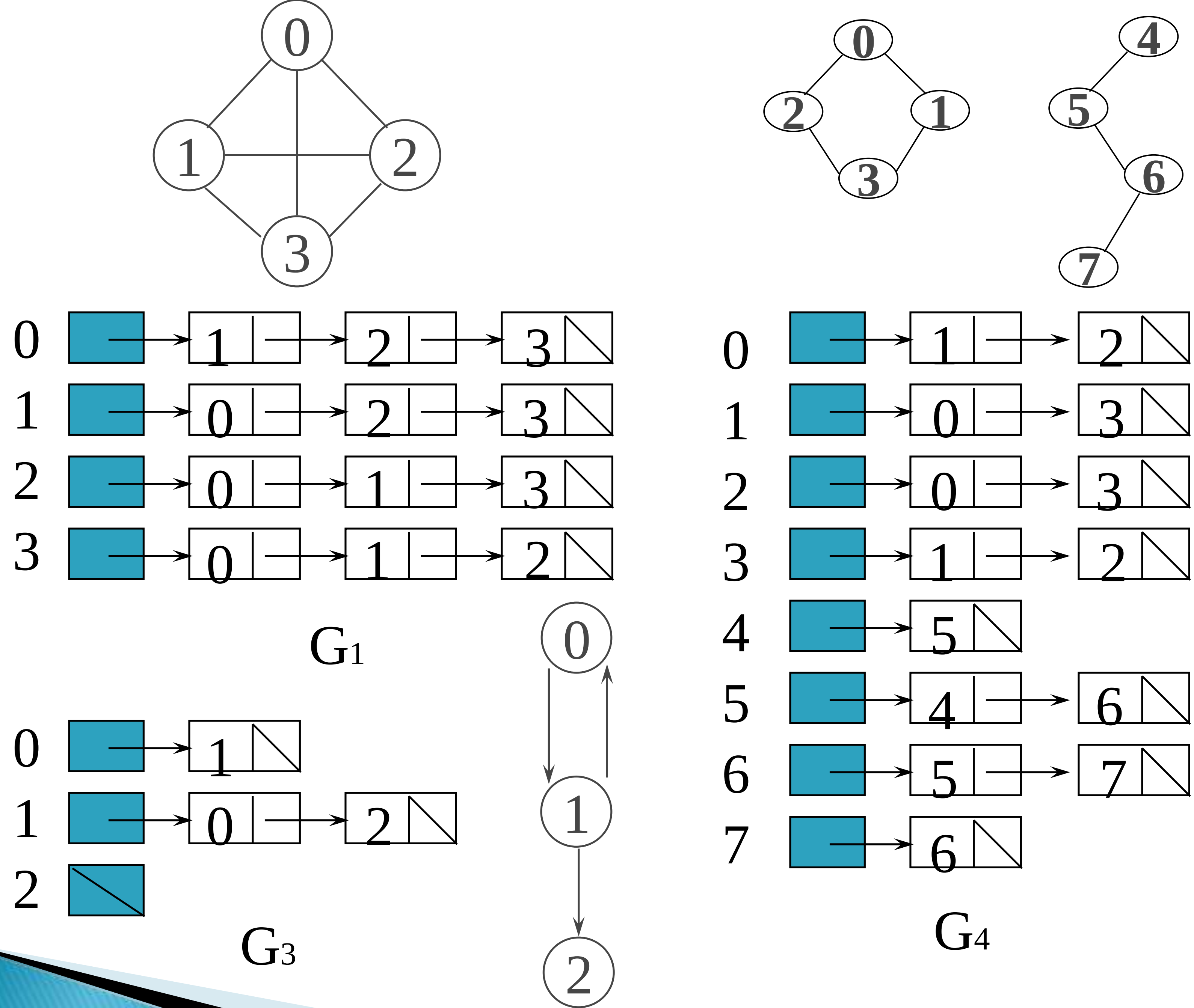
- From the adjacency matrix, to determine the connection of vertices is easy
- The degree of a vertex is $\sum_{j=0}^{n-1} adj\_mat[i][j]$
- For a digraph, the row sum is the out_degree, while the column sum is the in_degree

$$ind(vi) = \sum_{j=0}^{n-1} A[j,i] \quad outd(vi) = \sum_{j=0}^{n-1} A[i,j]$$

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 *
```

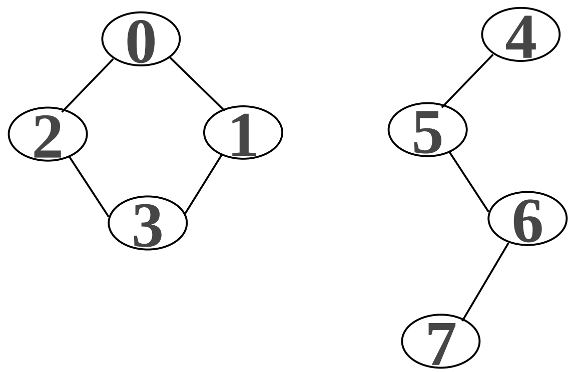


An undirected graph with n vertices and e edges ==> n head nodes and 2e list nodes

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

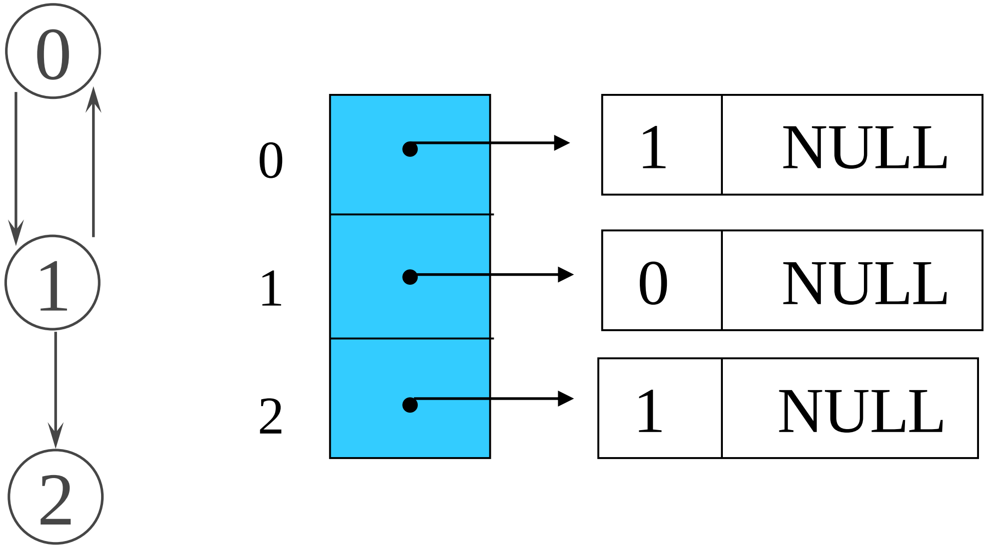
Compact Representation



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

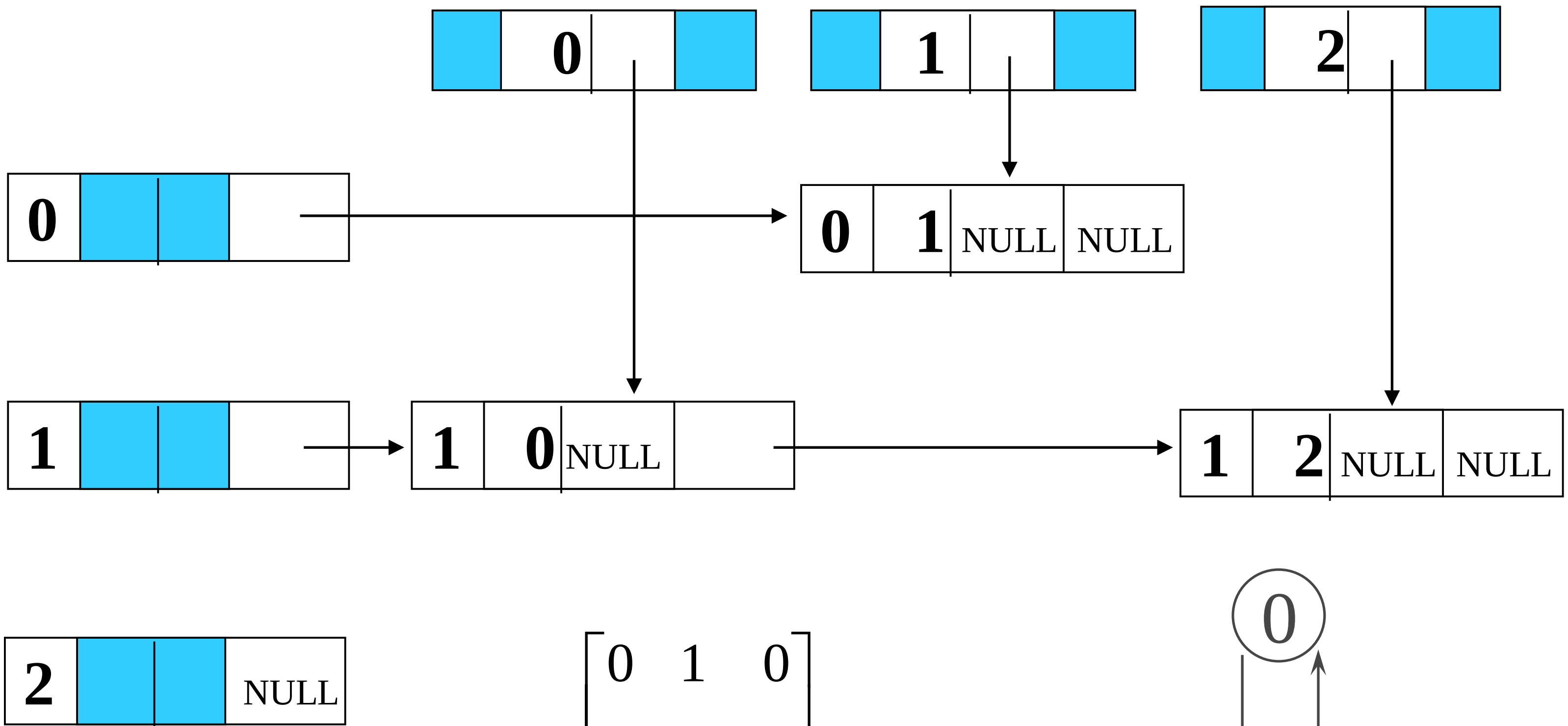


Determine in-degree of a vertex in a fast way.

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 G_3(p.268)



$$\begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix}$$

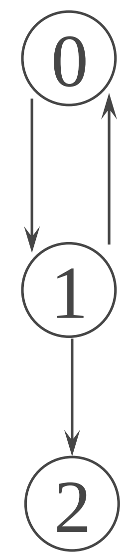
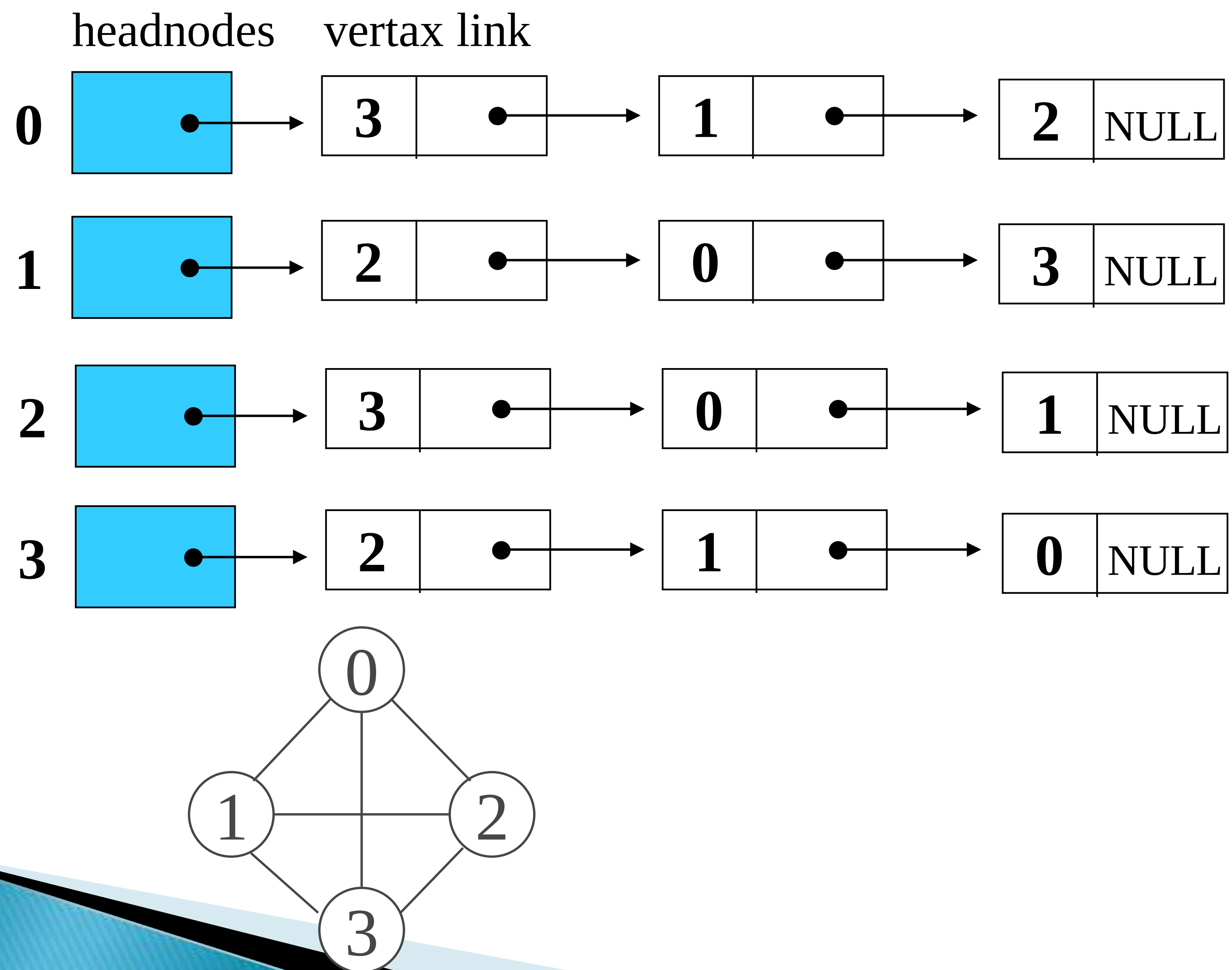


Figure 6.13: Alternate order adjacency list for G_1 (p.268)

Order is of no significance.

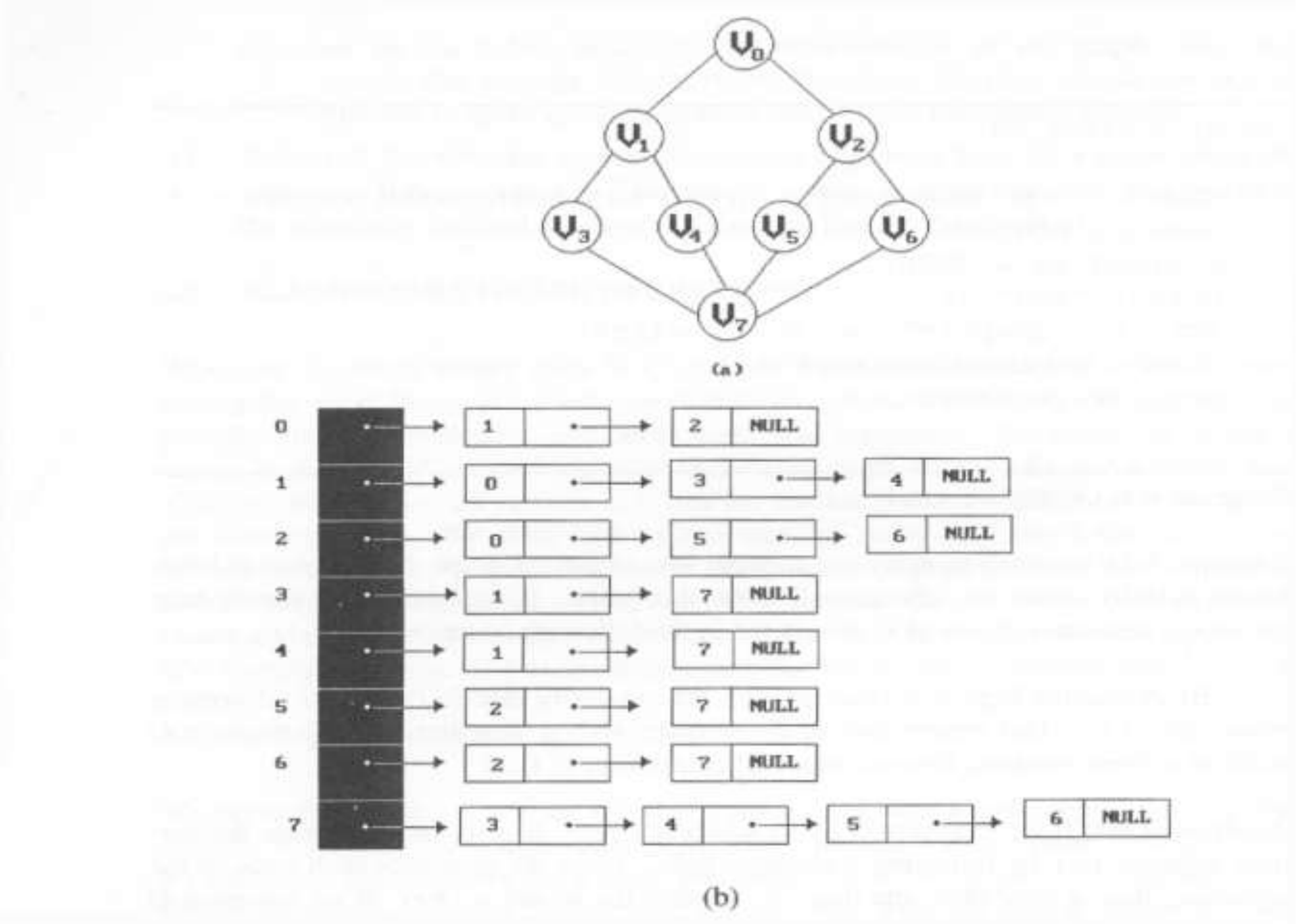


Some Graph Operations

- Traversal
  Given G=(V,E) and vertex v, find all $w \in V$, 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



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 (con't)

- 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?

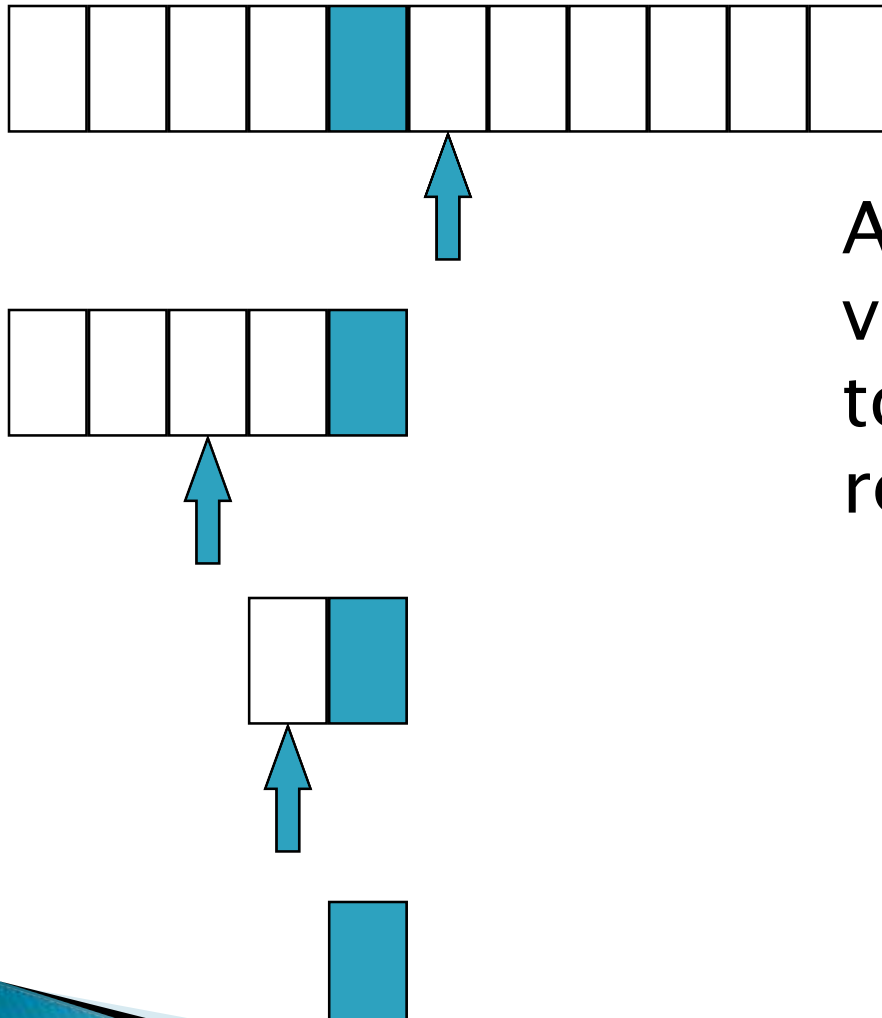
Ordered vs. Unordered (con't)

- 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
- How about the worst case for a list with 32 items ?
  - 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 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 $512 = 2^9$
- $8 < 11 < 16 \qquad 2^3 < 11 < 2^4$
- $128 < 250 < 256 \qquad 2^7 < 250 < 2^8$

A Very Fast Algorithm!

- How long (worst case) will it take to find an item in a list 30,000 items long?

$2^{10} = 1024$	$2^{13} = 8192$
$2^{11} = 2048$	$2^{14} = 16384$
$2^{12} = 4096$	$2^{15} = 32768$

- 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.
- $8 = 2^3$ $\lg 8 = 3$ $16 = 2^4$ $\lg 16 = 4$
- There are no algorithms that run faster than lg n time.

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
Selection Sort
Insertion Sort
Heap Sort
Merge 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

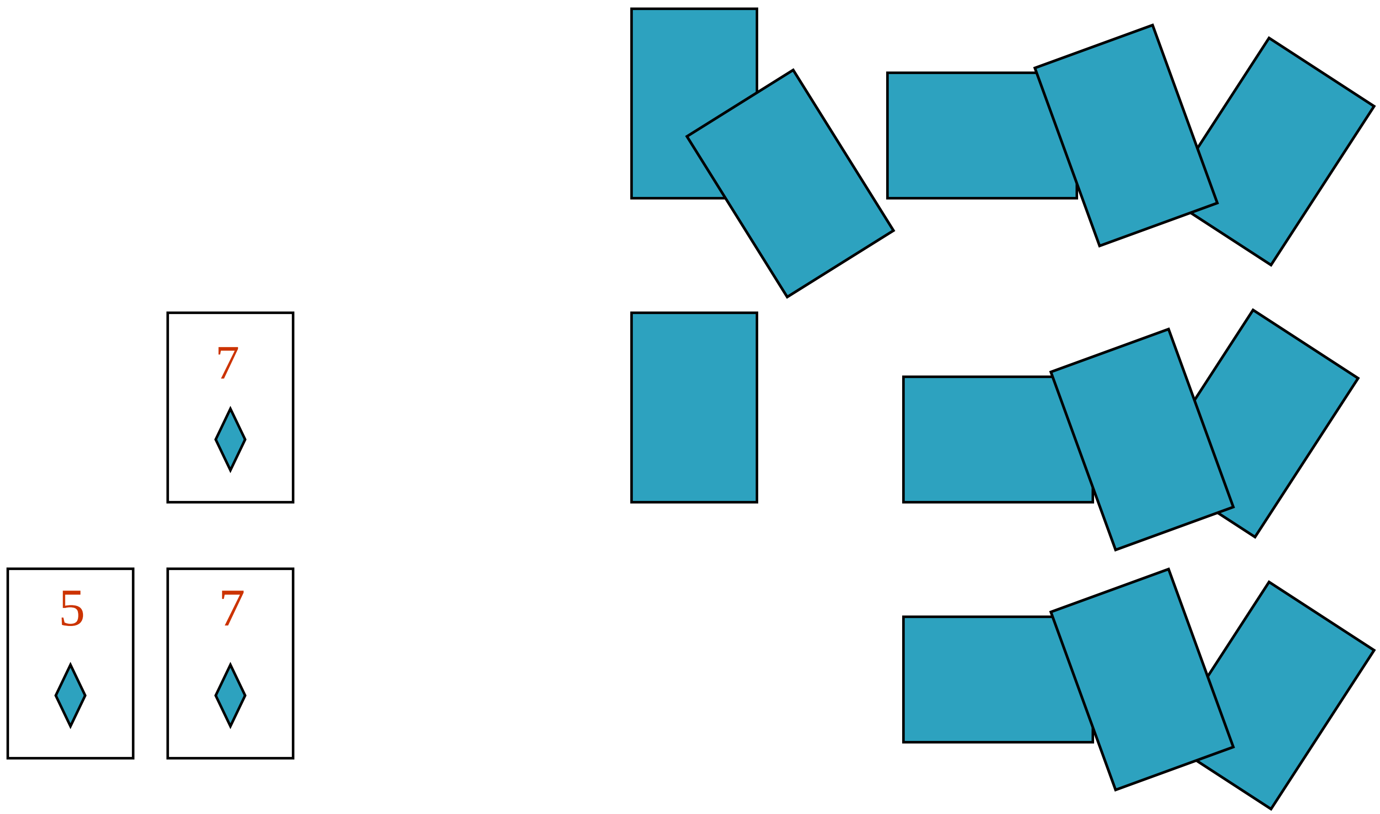
Bubble Sort Code

```
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;
            }
        }
    }
}
```

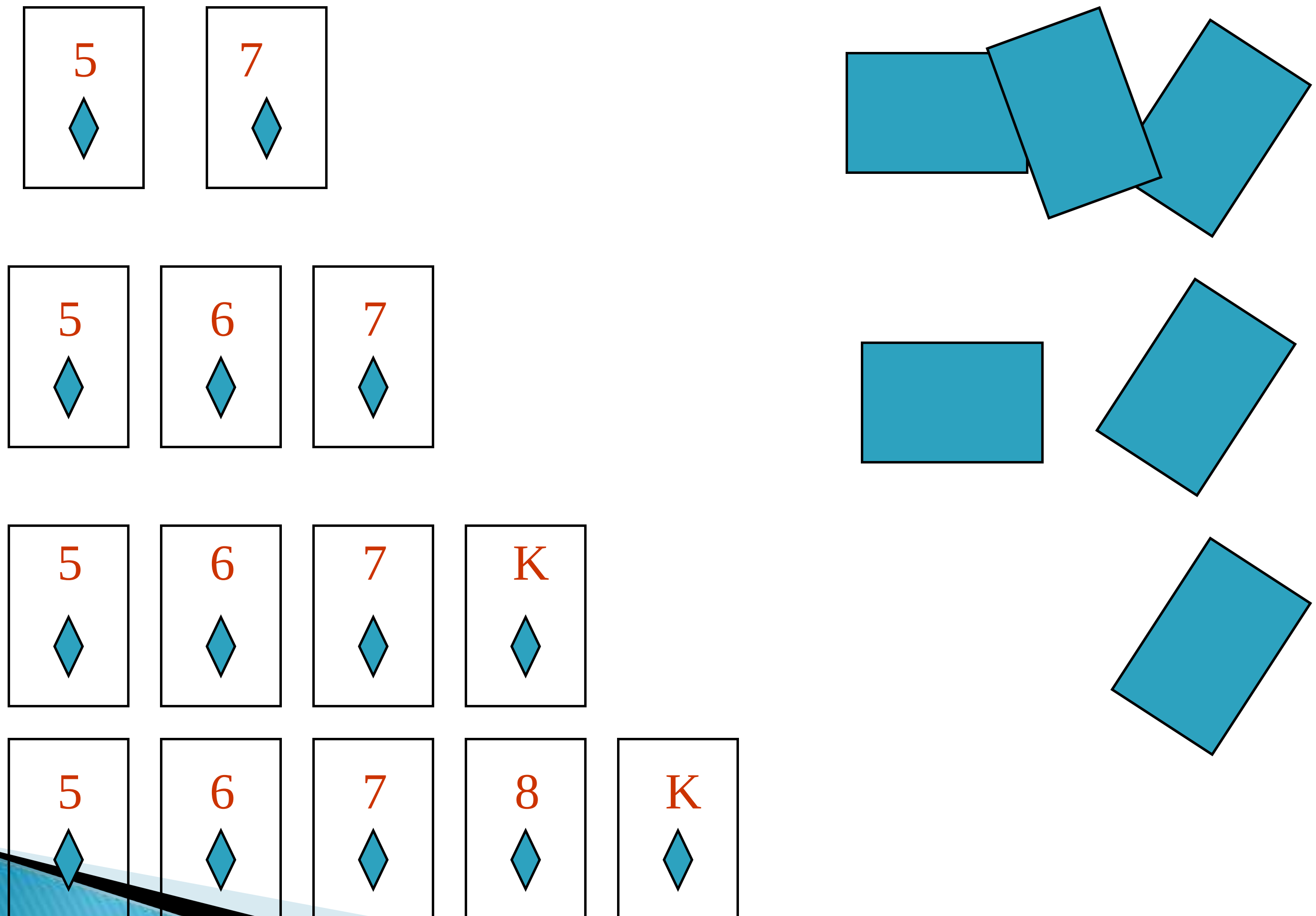
Insertion Sort

- 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.

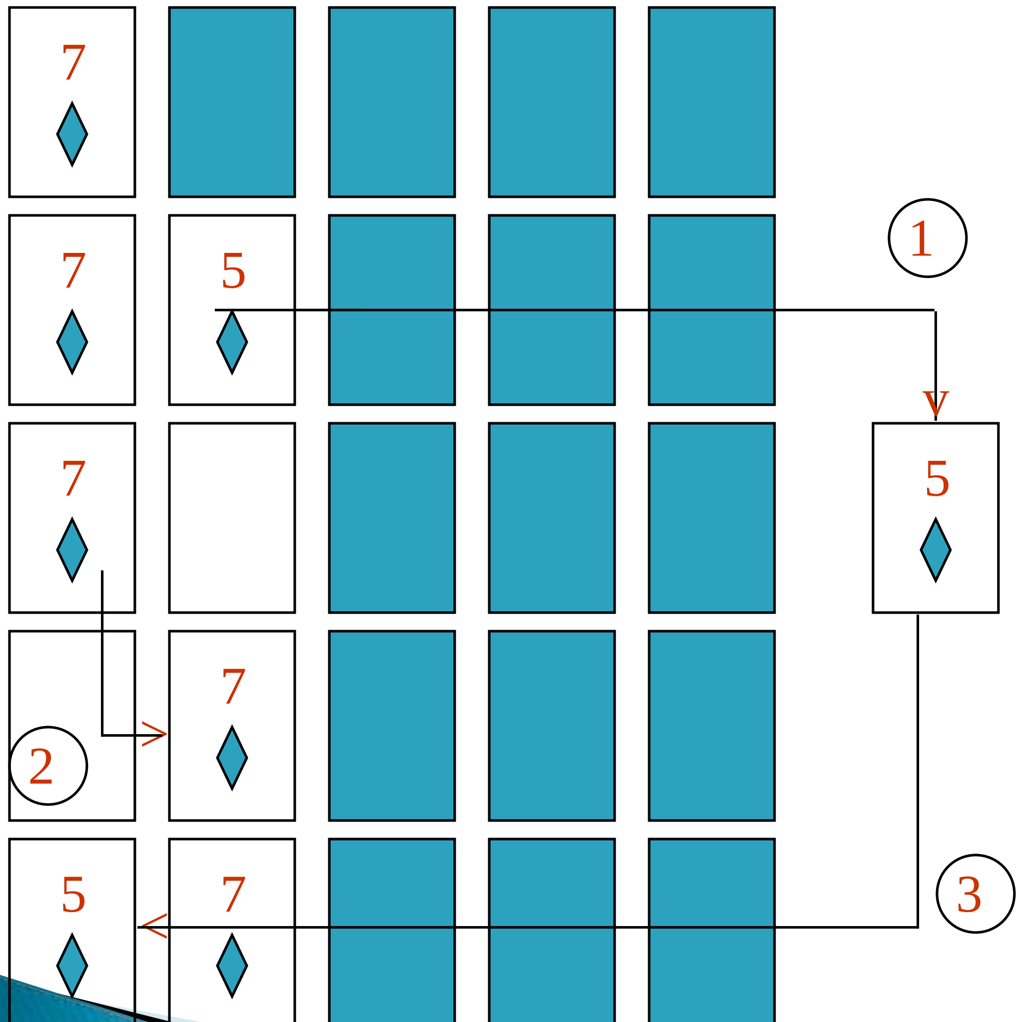
Arranging Your Hand



Arranging Your Hand



Insertion Sort



Unsorted – shaded

Look at 2nd item – 5.

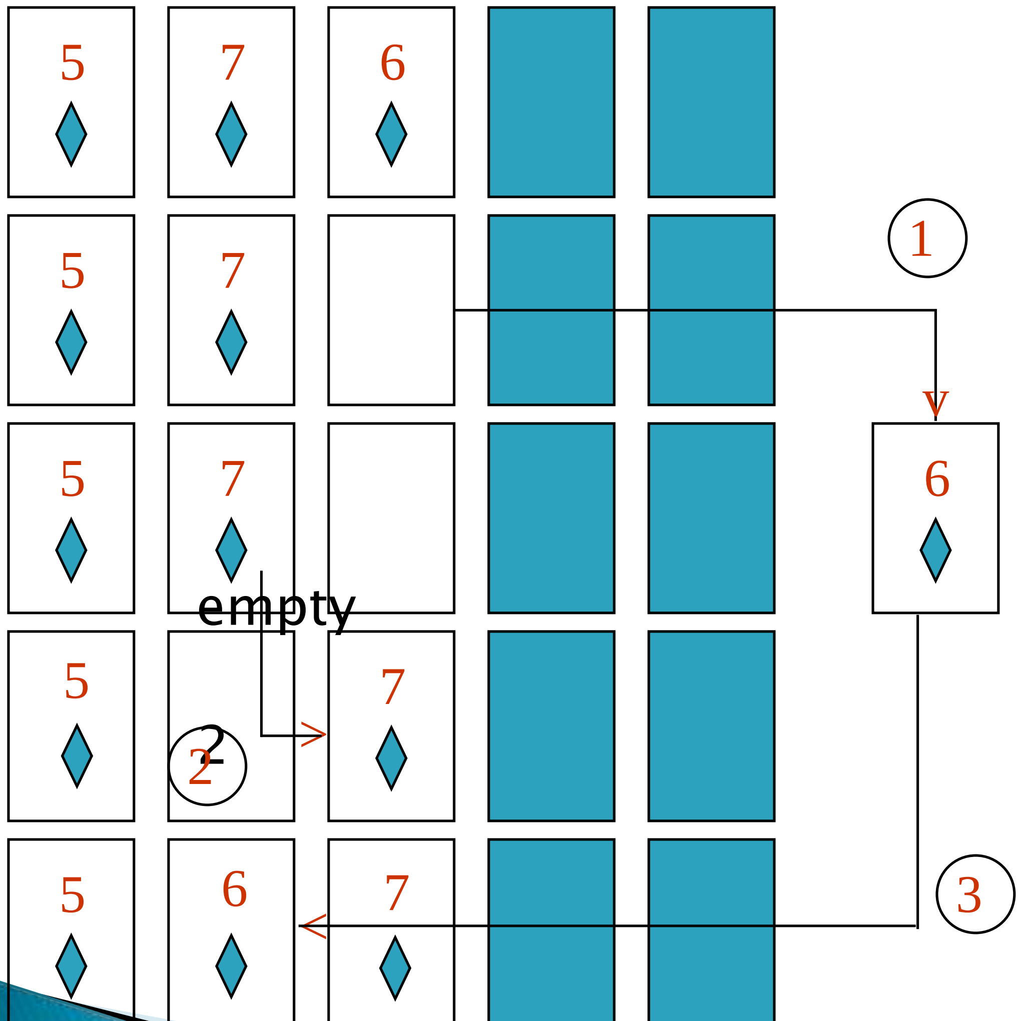
Compare 5 to 7.

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

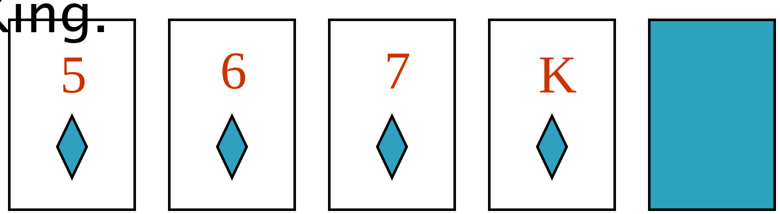
slot, leaving position

open.

Move 6 to the open 2nd position.

Insertion Sort (con't)

Look at next item – King.

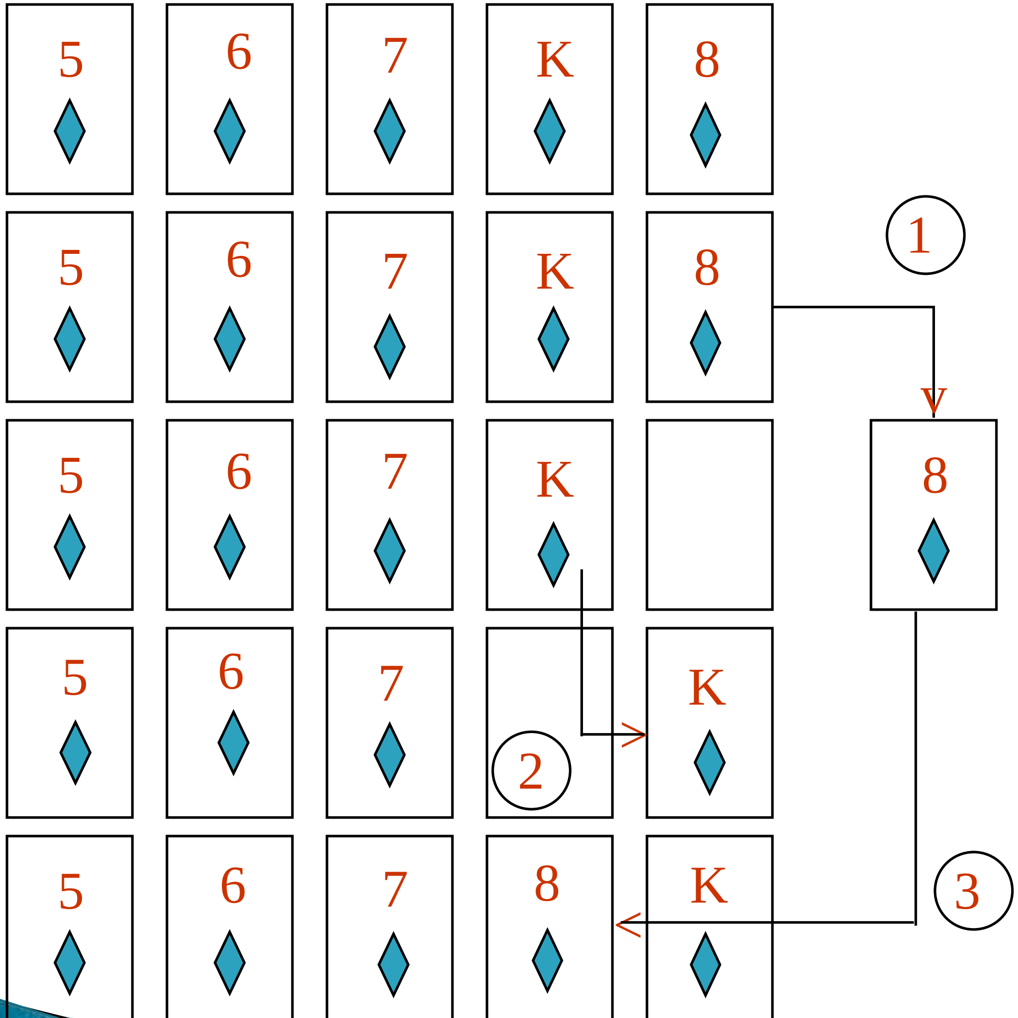


Compare to 1st – 5. King is larger, so leave 5 where it is.

Compare to next – 6. King is larger, so leave 6 where it is.

Compare to next – 7. King is larger, so leave 7 where it is.

Insertion Sort (con't)

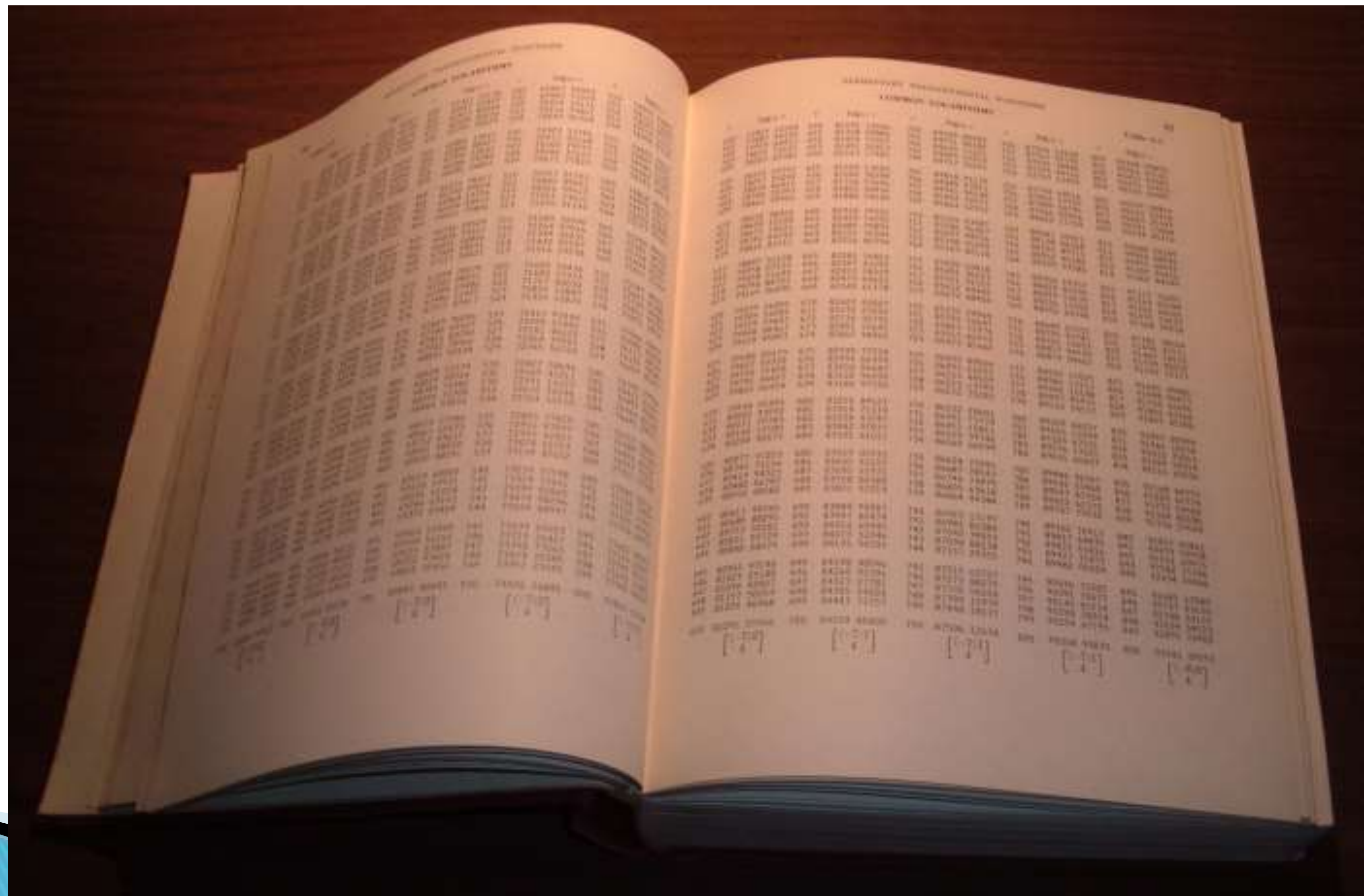


Hashing

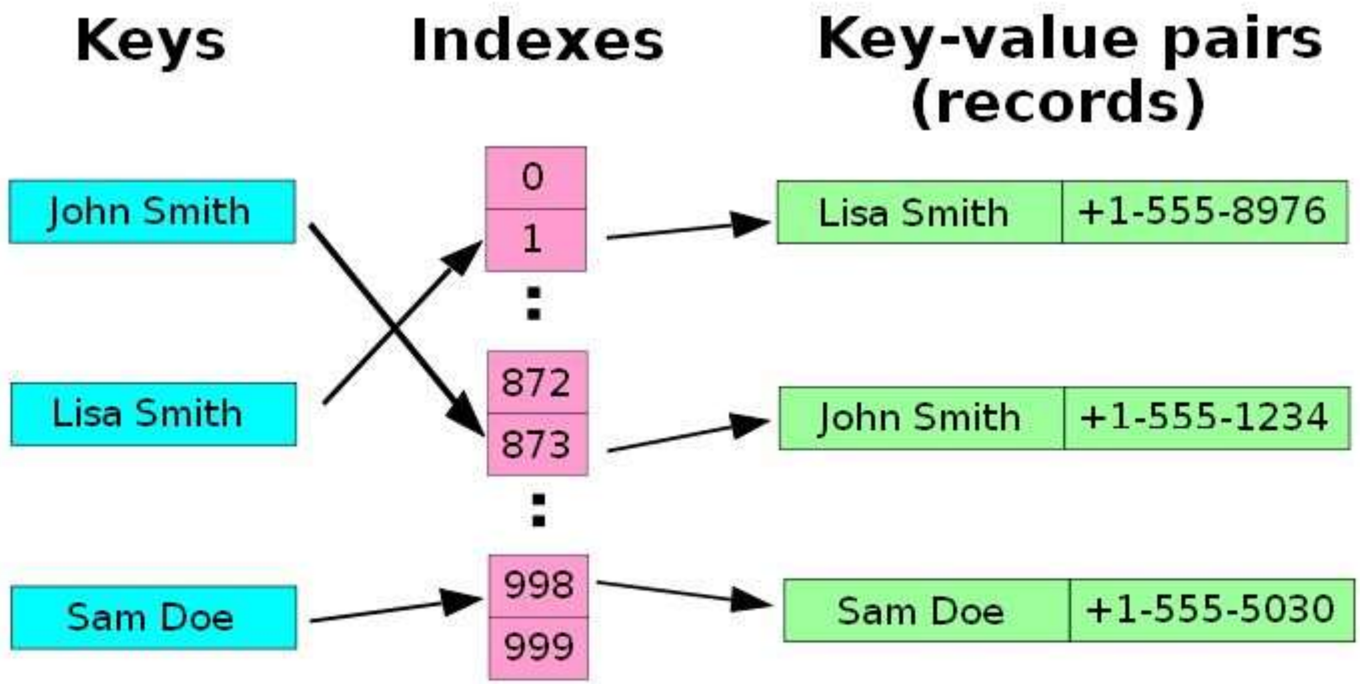
Concept of Hashing

- In CS, a **hash table**, or a **hash map**, is a data structure that associates keys (names) with values (attributes).
  - Look-Up Table
  - Dictionary
  - Cache
  - Extended Array

Tables of logarithms



Example



A small phone book as a hash table.

(Figure is from Wikipedia)

Dictionaries

- Collection of pairs.
  - (key, value)
  - Each pair has a unique key.
- Operations.
  - Get(theKey)
  - Delete(theKey)
  - Insert(theKey, theValue)

Just An Idea

- Hash table :
  - Collection of pairs,
  - Lookup function (Hash function)
- Hash tables are often used to implement associative arrays,
  - Worst-case time for Get, Insert, and Delete is O(size).
  - Expected time is O(1).

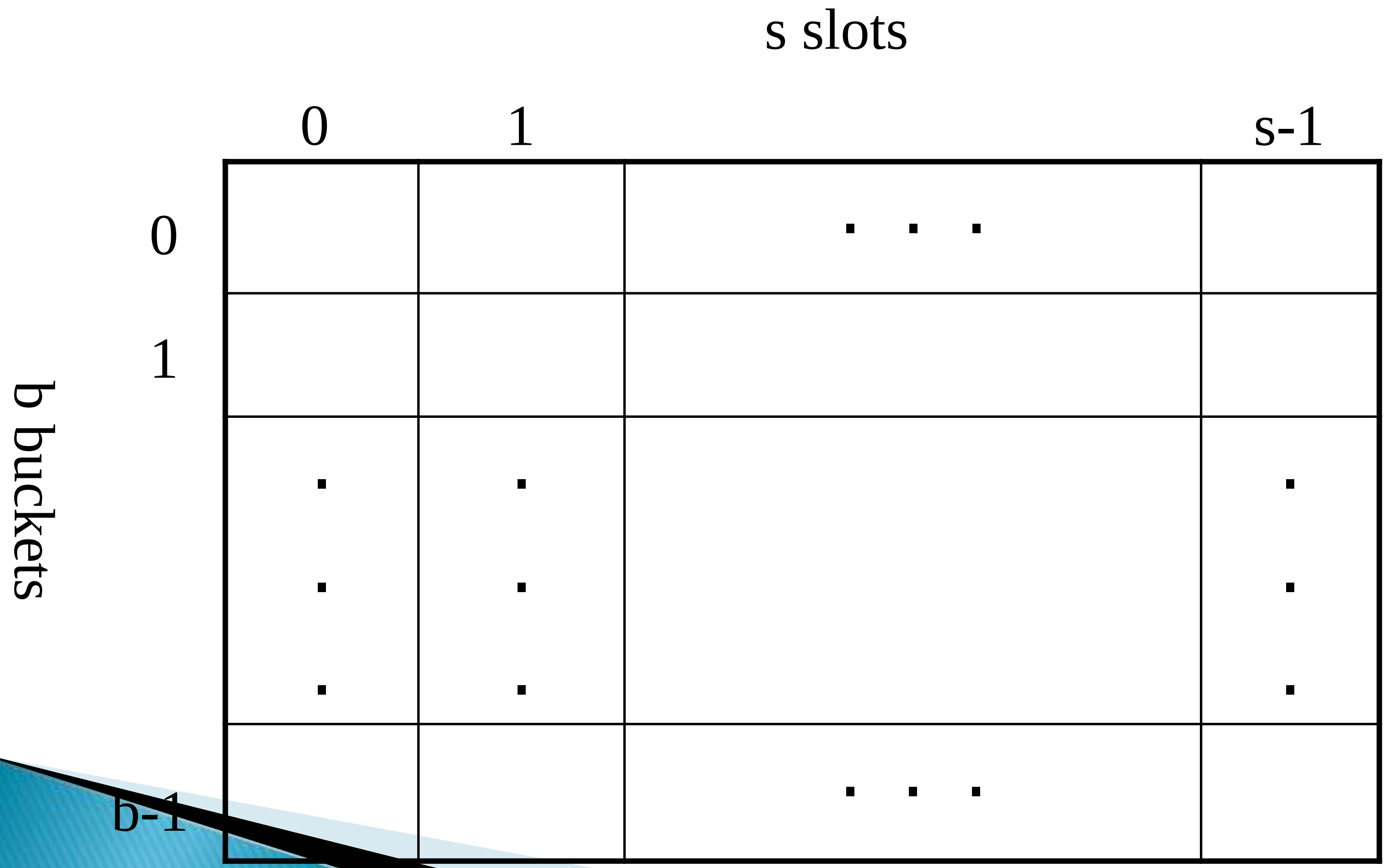
Search vs. Hashing

- Search tree methods: key comparisons
  - Time complexity: O(size) or O(log n)
- Hashing methods: hash functions
  - Expected time: O(1)
- Types
  - Static hashing (section 8.2)
  - Dynamic hashing (section 8.3)

Static Hashing

- Key-value pairs are stored in a fixed size table called a *hash table*.
  - A hash table is partitioned into many ***buckets***.
  - Each bucket has many ***slots***.
  - Each slot holds one record.
  - A hash function f(x) transforms the identifier (key) into an address in the hash table

Hash table



Data Structure for Hash Table

```
#define MAX_CHAR  10
#define TABLE_SIZE  13
typedef struct {
    char key[MAX_CHAR];
    /* other fields */
} element;
element hash_table[TABLE_SIZE];
```

Linear probing (linear open addressing)

- **Open addressing** ensures that all elements are stored directly into the hash table, thus it attempts to resolve collisions using various methods.

- **Linear Probing** resolves collisions by placing the data into the next open slot in the table.

Linear Probing – Get And Insert

- divisor = b (number of buckets) = 17.
- Home bucket = key % 17.

0				4				8				12				16
34	0	45				6	23	7			28	12	29	11	30	33

- Insert pairs whose keys are 6, 12, 34, 29, 28, 11, 23, 7, 0, 33, 30, 45

Linear Probing – Delete

0				4				8				12				16
34	0	45				6	23	7			28	12	29	11	30	33

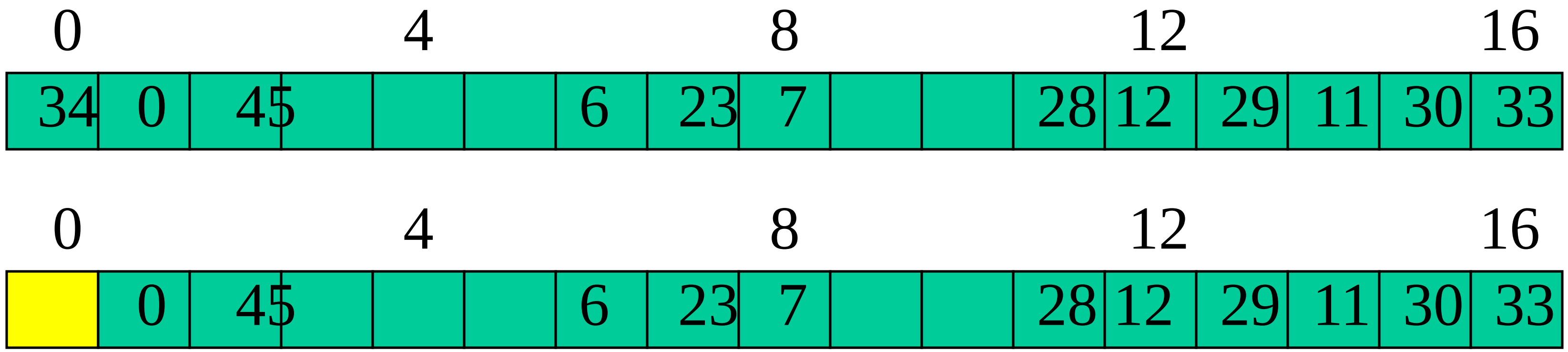
- Delete(0)

0				4				8				12				16
34		45				6	23	7			28	12	29	11	30	33

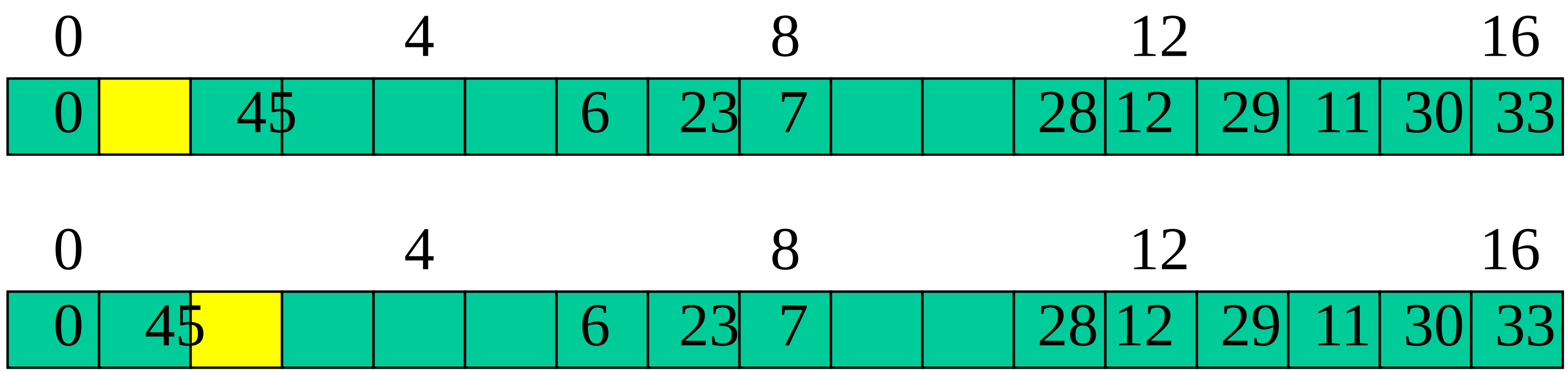
- Search cluster for pair (if any) to fill vacated bucket.

0				4				8				12				16
34	45					6	23	7			28	12	29	11	30	33

Linear Probing – Delete(34)



- Search cluster for pair (if any) to fill vacated bucket.



Linear Probing – Delete(29)

0				4				8				12				16
34	0	45				6	23	7			28	12	29	11	30	33

0				4				8				12				16
34	0	45				6	23	7			28	12		11	30	33

- Search cluster for pair (if any) to fill vacated bucket.

0				4				8				12				16
34	0	45				6	23	7			28	12	11		30	33

0				4				8				12				16
34	0	45				6	23	7			28	12	11	30		33

0				4				8				12				16
34	0					6	23	7			28	12	11	30	45	33

Linear Probing (program 8.3)

```
void linear_insert(element item, element ht[]){
  int i, hash_value;
  i = hash_value = hash(item.key);
  while(strlen(ht[i].key)) {
     if (!strcmp(ht[i].key, item.key)) {
                    fprintf(stderr, “Duplicate entry\n”);  exit(1);
          }
     i = (i+1)%TABLE_SIZE;
     if (i == hash_value) {
               fprintf(stderr, “The table is full\n”); exit(1);
     }  }
  ht[i] = item;
}
```

Problem of Linear Probing

- Identifiers tend to cluster together
- Adjacent cluster tend to coalesce
- Increase the search time

Dynamic Hashing Using Directories

Identifiers	Binary representaiton
a0	100 0<u>00</u>
a1	100 0<u>01</u>
b0	101 0<u>00</u>
b1	101 0<u>01</u>
c0	110 0<u>00</u>
c1	110 0<u>01</u>
c2	110 0<u>10</u>
c3	110 0<u>11</u>

Example:

M (# of pages)=4,

P (page capacity)=2

Allocation: lower order two bits

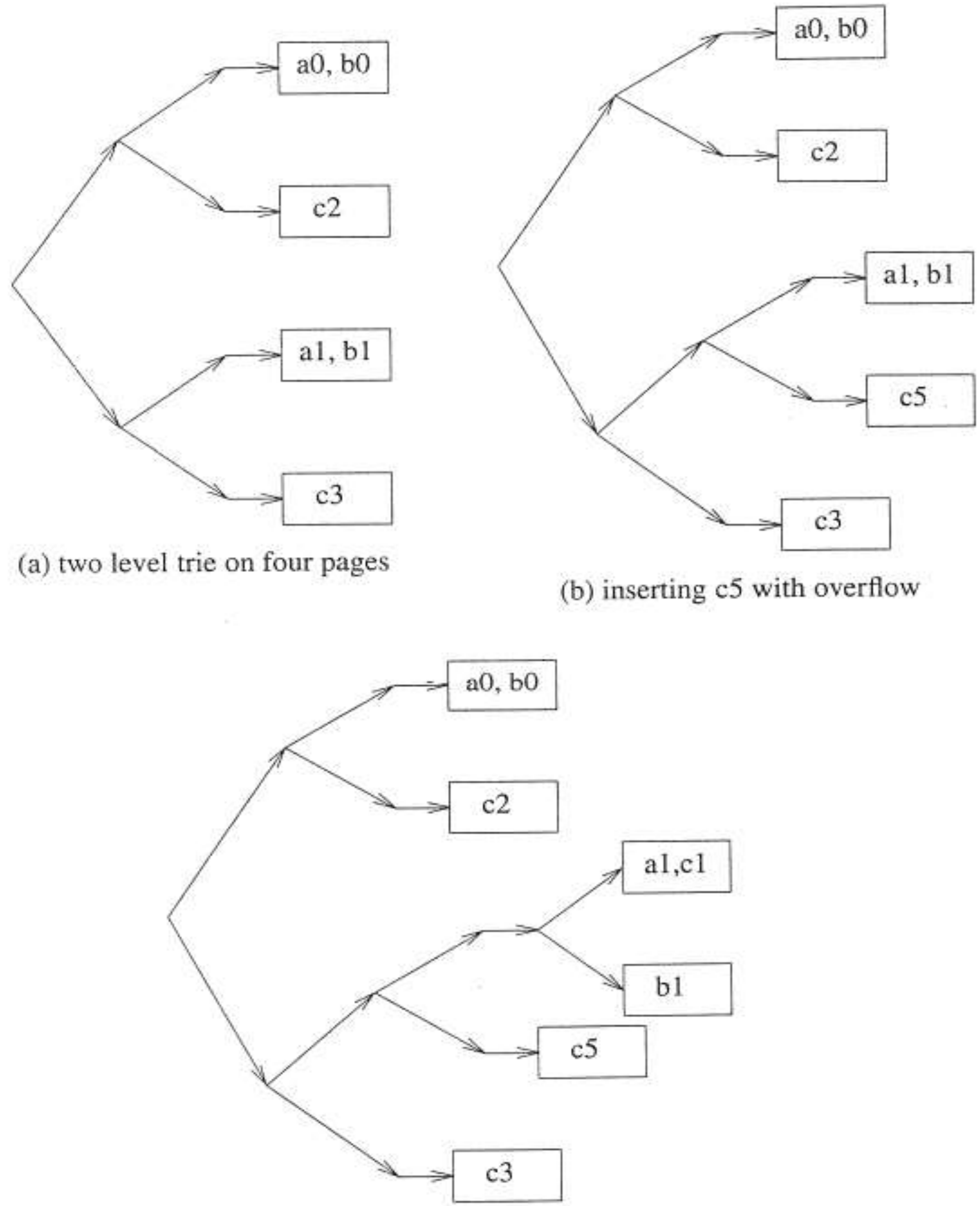
Figure 8.8: Some identifiers requiring 3 bits per character(p.414)

Figure 8.9: A trie to hole identifiers (p.415)

Read it in reverse order.

c5: 110 101

c1: 110 001



(a) two level trie on four pages

(b) inserting c5 with overflow

(c) inserting c1 with overflow

Figure 8.9: A trie to hold identifiers

Dynamic Hashing Using Directories II

- We need to consider some issues!
  - Skewed Tree,
  - Access time increased.
- Fagin et. al. proposed extendible hashing to solve above problems.
  - **Ronald Fagin**, **Jürg Nievergelt**, **Nicholas Pippenger**, and **H. Raymond Strong**, *Extendible Hashing - A Fast Access Method for Dynamic Files*, ACM Transactions on Database Systems, 4(3):315–344, 1979.

Dynamic Hashing Using Directories III

- A directories is a table of pointer of pages.
- The directory has k bits to index 2^k entries.
- We could use a hash function to get the address of entry of directory, and find the page contents at the page.

The directory of the three tries of Figure 8.9

$00 \xrightarrow{a} a0,\ b0$
$01 \xrightarrow{c} a1,\ b1$
$10 \xrightarrow{b} c2$
$11 \xrightarrow{d} c3$

(a) 2 bits

$000 \xrightarrow{a} a0,\ b0$
$001 \xrightarrow{c} a1,\ b1$
$010 \xrightarrow{b} c2$
$011 \xrightarrow{e} c3$
$100 \xrightarrow{a}$
$101 \xrightarrow{d} c5$
$110 \xrightarrow{b}$
$111 \xrightarrow{e}$

(b) 3 bits

$0000 \xrightarrow{a} a0,\ b0$
$0001 \xrightarrow{c} a1,\ c1$
$0010 \xrightarrow{b} c2$
$0011 \xrightarrow{f} c3$
$0100 \xrightarrow{a}$
$0101 \xrightarrow{e} c5$
$0110 \xrightarrow{b}$
$0111 \xrightarrow{f}$
$1000 \xrightarrow{a}$
$1001 \xrightarrow{d} b1$
$1010 \xrightarrow{b}$
$1011 \xrightarrow{f}$
$1100 \xrightarrow{a}$
$1101 \xrightarrow{e}$
$1110 \xrightarrow{b}$
$1111 \xrightarrow{f}$

(c) 4 bits

Figure 8.10: Tries collapsed into directories

Dynamic Hashing Using Directories IV

It is obvious that the directories will grow very large if the hash function is clustering.

Therefore, we need to adopt the uniform hash function to translate the bits sequence of keys to the random bits sequence.

Moreover, we need a family of uniform hash functions, since the directory will grow.