School of Computing

Department of Computer Science and Engineering

UNIT - I

Fundamental of Data Structure – SBS1201

UNIT-1

Arrays- Linked List - Single Linked List - Insertion and Deletion - Doubly Linked List.- Circular Linked List – Stack- Queues- Array implementation of a Stack and queue - Linked List implementation of a Stack and Queue- Priority Queues

1. INDRODUCTION

ABSTRACT DATA TYPE

In programming each program is breakdown into modules, so that no routine should ever exceed a page. Each module is a logical unit and does specific job modules which in turn will call another module.

Modularity has several advantages

1. Modules can be compiled separately which makes debugging process easier.

2. Several modules can be implemented and executed simultaneously.

3. Modules can be easily enhanced.

Abstract Data type is an extension of modular design.

An abstract data type is a set of operations such as Union, Intersection, Complement, Find etc.,

The basic idea of implementing ADT is that the operations are written once in program and can be called by any part of the program.

1.1 THE LIST ADT

List is an ordered set of elements.

The general form of the list is

$A_1, A_2, A_3, ... ,A_N$

A_1 - First element of the list

A_N - Last element of the list

N - Size of the list

If the element at position i is A_i then its successor is A_{i+1} and its predecessor is A_{i-1}.

Various operations performed on List

1. Insert (X, 5) - Insert the element X after the position 5.

2. Delete (X) - The element X is deleted

3. Find (X) - Returns the position of X.

4. Next (i) - Returns the position of its successor element i+1.

5. Previous (i) - Returns the position of its predecessor i-1.

6. Print list - Contents of the list is displayed.

7. Makeempty - Makes the list empty.

1.1.1 Implementation of List ADT

1. Array Implementation

2. Linked List Implementation

3. Cursor Implementation.

Array Implementation of List

Array is a collection of specific number of data stored in a consecutive memory locations.

* Insertion and Deletion operation are expensive as it requires more data movement

* Find and Printlist operations takes constant time.

* Even if the array is dynamically allocated, an estimate of the maximum size of the

list is required which considerably wastes the memory space.

Linked List Implementation

Linked list consists of series of nodes. Each node contains the element and a pointer to its successor node. The pointer of the last node points to NULL.

Insertion and deletion operations are easily performed using linked list.

Types of Linked List

1. Singly Linked List

2. Doubly Linked List

3. Circular Linked List.

1.1.2 Singly Linked List

A singly linked list is a linked list in which each node contains only one link field pointing to the next node in the list.

DECLARATION FOR LINKED LIST

```
Struct node ;

typedef struct Node *List ;

typedef struct Node *Position ;

int IsLast (List L) ;

int IsEmpty (List L) ;

position Find(int X, List L) ;

void Delete(int X, List L) ;

position FindPrevious(int X, List L) ;

position FindNext(int X, List L) ;

void Insert(int X, List L, Position P) ;

void DeleteList(List L) ;

Struct Node

{

int element ;

position Next ;

};
```

ROUTINE TO INSERT AN ELEMENT IN THE LIST

```
void Insert (int X, List L, Position P)
/* Insert after the position P*/
{
position Newnode;
Newnode = malloc (size of (Struct Node));
If (Newnode! = NULL)
{
Newnode ->Element = X;
Newnode ->Next = P->    Next;
P->    Next = Newnode;
}
}
```

INSERT (25, P, L)

ROUTINE TO CHECK WHETHER THE LIST IS EMPTY

```
int IsEmpty (List L) /*Returns 1 if L is empty */
{
if (L ->    Next = = NULL)
return (1);
}
```

ROUTINE TO CHECK WHETHER THE CURRENT POSITION IS LAST

```
int IsLast (position P, List L) /* Returns 1 is P is the last position in L */
{
```

```
if (P->Next = = NULL)
return
}
```

FIND ROUTINE

```
position Find (int X, List L)
{
/*Returns the position of X in L; NULL if X is not found */
position P;
P = L->     Next;
while (P! = NULL && P    Element ! = X)
P = P->Next;
return P;
}
}
```

FIND PREVIOUS ROUTINE

```
position FindPrevious (int X, List L)
{
/* Returns the position of the predecessor */
position P;
P = L;
while (P ->    Next ! = Null && P ->Next     Element ! = X)
P = P ->Next;
return P;
```

```
}
```

FINDNEXT ROUTINE

```
position FindNext (int X, List L)
{
/*Returns the position of its successor */
P = L ->Next;
while (P    Next! = NULL && P    Element ! = X)
P = P → Next;
return P → Next;
}
```

ROUTINE TO DELETE AN ELEMENT FROM THE LIST

```
void Delete(int X, List L)
{
/* Delete the first occurence of X from the List */
position P, Temp;
P = Findprevious (X,L);
If (!IsLast(P,L))
{
Temp = P → Next;
P  → Next = Temp → Next;
Free (Temp);
}
}
```

ROUTINE TO DELETE THE LIST

```
void DeleteList (List L)

{

position P, Temp;

P = L →Next;

L→Next = NULL;

while (P! = NULL)

{

Temp = P→Next

free (P);

P = Temp;

}

}
```

1.1.3 Doubly Linked List

A Doubly linked list is a linked list in which each node has three fields namely data field, forward link (FLINK) and Backward Link (BLINK). FLINK points to the successor node in the list whereas BLINK points to the predecessor node.

STRUCTURE DECLARATION : -

```
Struct Node

{

int Element;

Struct Node *FLINK;

Struct Node *BLINK

};
```

ROUTINE TO INSERT AN ELEMENT IN A DOUBLY LINKED LIST

```
void Insert (int X, list L, position P)
{
Struct Node * Newnode;
Newnode = malloc (size of (Struct Node));
If (Newnode ! = NULL)
{
Newnode →Element = X;
Newnode →Flink = P     Flink;
P →Flink →Blink = Newnode;
P →Flink = Newnode ;
Newnode →Blink = P;
}
}
```

ROUTINE TO DELETE AN ELEMENT

```
void Delete (int X, List L)
{
position P;
P = Find (X, L);
If ( IsLast (P, L))
{
Temp = P;
P →Blink →Flink = NULL;
```

free (Temp);

}

else

{

Temp = P;

P →Blink→ Flink = P→Flink;

P →Flink →Blink = P→Blink;

free (Temp);

}

}

Advantage

* Deletion operation is easier.

* Finding the predecessor & Successor of a node is easier.

Disadvantage

* More Memory Space is required since it has two pointers.

1.1.4 Circular Linked List

In circular linked list the pointer of the last node points to the first node. Circular linked list can be implemented as Singly linked list and Doubly linked list with or without headers.

Singly Linked Circular List

A singly linked circular list is a linked list in which the last node of the list points to the first node.

Doubly Linked Circular List

A doubly linked circular list is a Doubly linked list in which the forward link of the last node points to the first node and backward link of the first node points to the last node of the list.

Advantages of Circular Linked List

• It allows to traverse the list starting at any point.

• It allows quick access to the first and last records.

1.1.5 Applications of Linked List

1. Polynomial ADT

2. Radix Sort

3. Multilist

1.2 THE STACK ADT

1.2.1 Stack Model :

A stack is a linear data structure which follows Last In First Out (LIFO) principle, in which both insertion and deletion occur at only one end of the list called the Top.

Example : -

Pile of coins., a stack of trays in cafeteria.

2.2.2 Operations On Stack

The fundamental operations performed on a stack are

1. Push

2. Pop

PUSH :

The process of inserting a new element to the top of the stack. For every push operation the top is incremented by 1.

POP :

The process of deleting an element from the top of stack is called pop operation. After every pop operation the top pointer is decremented by 1.

EXCEPTIONAL CONDITIONS

OverFlow

Attempt to insert an element when the stack is full is said to be overflow.

UnderFlow

Attempt to delete an element, when the stack is empty is said to be underflow.

1.2.3 Implementation of Stack

Stack can be implemented using arrays and pointers.

Array Implementation

In this implementation each stack is associated with a pop pointer, which is -1 for an empty stack.

• To push an element X onto the stack, Top Pointer is incremented and then set Stack [Top] = X.

• To pop an element, the stack [Top] value is returned and the top pointer is decremented.

• pop on an empty stack or push on a full stack will exceed the array bounds.

ROUTINE TO PUSH AN ELEMENT ONTO A STACK

```
void push (int x, Stack S)

{

if (IsFull (S))

Error ("Full Stack");

else

{

Top = Top + 1;

S[Top] = X;

}

}

int IsFull (Stack S)

{
```

```
if (Top = = Arraysize)
return (1);
}
```

ROUTINE TO POP AN ELEMENT FROM THE STACK

```
void pop (Stack S)
{
if (IsEmpty (S))
Error ("Empty Stack");
else
{
X = S [Top];
Top = Top - 1;
}
}
int IsEmpty (Stack S)
{
if (S    Top = = -1)
return (1);
}
```

ROUTINE TO RETURN TOP ELEMENT OF THE STACK

```
int TopElement (Stack S)
{
if (! IsEmpty (s))
```

```
return S[Top];
else
Error ("Empty Stack");
return 0;
}
```

1.3 LINKED LIST IMPLEMENTATION OF STACK

• Push operation is performed by inserting an element at the front of the list.

• Pop operation is performed by deleting at the front of the list.

• Top operation returns the element at the front of the list.

DECLARATION FOR LINKED LIST IMPLEMENTATION

```
Struct Node;
typedef Struct Node *Stack;
int IsEmpty (Stack S);
Stack CreateStack (void);
void MakeEmpty (Stack S);
void push (int X, Stack S);
int Top (Stack S);
void pop (Stack S);
Struct Node
{
int Element ;
Struct Node *Next;
};
```

ROUTINE TO CHECK WHETHER THE STACK IS EMPTY

```
int IsEmpty (Stack S)
{
if (S→Next = = NULL)
return (1);
}
```

ROUTINE TO CREATE AN EMPTY STACK

```
Stack CreateStack ( )
{
Stack S;
S = malloc (Sizeof (Struct Node));
if (S = = NULL)
Error (" Outof Space");
MakeEmpty (s);
return S;
}
void MakeEmpty (Stack S)
{
if (S = = NULL)
Error (" Create Stack First");
else
while (! IsEmpty (s))
pop (s);
```

```
}
```

ROUTINE TO PUSH AN ELEMENT ONTO A STACK

```
void push (int X, Stack S)
{
Struct Node * Tempcell;
Tempcell = malloc (sizeof (Struct Node));
If (Tempcell = = NULL)
Error ("Out of Space");
else
{
Tempcell    Element = X;
Tempcell    Next = S    Next;
S→Next = Tempcell;
}
}
```

ROUTINE TO RETURN TOP ELEMENT IN A STACK

```
int Top (Stack S)
{
If (! IsEmpty (s))
return S→Next→Element;
Error ("Empty Stack");
return 0;
}
```

ROUTINE TO POP FROM A STACK

```
void pop (Stack S)

{

Struct Node *Tempcell;

If (IsEmpty (S))

Error ("Empty Stack");

else

{

Tempcell = S → Next;

S → Next = S → Next → Next;

Free (Tempcell);

}

}
```

1.4 The Queue ADT

1.4.1 Queue Model

A Queue is a linear data structure which follows First In First Out (FIFO) principle, in which insertion is performed at rear end and deletion is performed at front end.

Example : Waiting Line in Reservation Counter,

2.3.2 Operations on Queue

The fundamental operations performed on queue are

1. Enqueue

2. Dequeue

Enqueue :

The process of inserting an element in the queue.

Dequeue :

The process of deleting an element from the queue.

Exception Conditions

Overflow : Attempt to insert an element, when the queue is full is said to be overflow condition.

Underflow : Attempt to delete an element from the queue, when the queue is empty is said to be underflow.

1.4.3 Implementation of Queue

Queue can be implemented using arrays and pointers.

Array Implementation

In this implementation queue Q is associated with two pointers namely rear pointer and front pointer.

To insert an element X onto the Queue Q, the rear pointer is incremented by 1 and then set

Queue [Rear] = X

To delete an element, the Queue [Front] is returned and the Front Pointer is incremented by 1.

ROUTINE TO ENQUEUE

```
void Enqueue (int X)
{
if (rear > = max _ Arraysize)
print (" Queue overflow");
else
{
Rear = Rear + 1;
Queue [Rear] = X;
}
```

```
}
```

ROUTINE FOR DEQUEUE

```
void delete ( )
{
if (Front < 0)
print (" Queue Underflow");
else
{
X = Queue [Front];
if (Front = = Rear)
{
Front = 0;
Rear = -1;
}
else
Front = Front + 1 ;
}
}
```

In Dequeue operation, if Front = Rear, then reset both

the pointers to their initial values. (i.e. F = 0, R = -1)

Linked List Implementation of Queue

Enqueue operation is performed at the end of the list.

Dequeue operation is performed at the front of the list.

Queue ADT

DECLARATION FOR LINKED LIST IMPLEMENTATION OF QUEUE ADT

```
Struct Node;
typedef Struct Node * Queue;
int IsEmpty (Queue Q);
Queue CreateQueue (void);
void MakeEmpty (Queue Q);
void Enqueue (int X, Queue Q);
void Dequeue (Queue Q);
Struct Node
{
int Element;
Struct Node *Next;
}* Front = NULL, *Rear = NULL;
```

ROUTINE TO CHECK WHETHER THE QUEUE IS EMPTY

```
int IsEmpty (Queue Q) // returns boolean value /
{ // if Q is empty
if (Q→Next = = NULL) // else returns 0
return (1);
}
```

ROUTINE TO CHECK AN EMPTY QUEUE

```
Struct CreateQueue ( )
{
```

```
Queue Q;
Q = Malloc (Sizeof (Struct Node));
if (Q = = NULL)
Error ("Out of Space");
MakeEmpty (Q);
return Q;
}
void MakeEmpty (Queue Q)
{
if (Q = = NULL)
Error ("Create Queue First");
else
while (! IsEmpty (Q)
Dequeue (Q);
}
```

ROUTINE TO ENQUEUE AN ELEMENT IN QUEUE

```
void Enqueue (int X)
{
Struct node *newnode;
newnode = Malloc (sizeof (Struct node));
if (Rear = = NULL)
{
newnode →data = X;
```

```
newnode →Next = NULL;
Front = newnode;
Rear = newnode;
}
else
{
newnode →   data = X;
newnode →   Next = NULL;
Rear →next = newnode;
Rear = newnode;
}
}
```

ROUTINE TO DEQUEUE AN ELEMENT FROM THE QUEUE

```
void Dequeue ( )
{
Struct node *temp;
if (Front = = NULL)
Error("Queue is underflow");
else
{
temp = Front;
if (Front = = Rear)
{
```

```
Front = NULL;

Rear = NULL;

}

else

Front = Front →Next;

Print (temp→data);

free (temp);

}

}
```

1.4.4 Double Ended Queue (DEQUE)

In Double Ended Queue, insertion and deletion operations are performed at both the ends.

1.4.5 Circular Queue

In Circular Queue, the insertion of a new element is performed at the very first location of the queue if the last location of the queue is full, in which the first element comes just after the last element.

Advantages

It overcomes the problem of unutilized space in linear queues, when it is implemented as arrays.

To perform the insertion of an element to the queue, the position of the element is calculated by the relation as

Rear = (Rear + 1) % Maxsize.

and then set

Queue [Rear] = value.

ROUTINE TO INSERT AN ELEMENT IN CIRCULAR QUEUE

```
void CEnqueue (int X)

{
```

```
if (Front = = (rear + 1) % Maxsize)
print ("Queue is overflow");
else
{
if (front = = -1)
front = rear = 0;
else
rear = (rear + 1)% Maxsize;
CQueue [rear] = X;
}
}
```

To perform the deletion, the position of the Front printer is calculated by the relation

Value = CQueue [Front]

Front = (Front + 1) % maxsize.

ROUTINE TO DELETE AN ELEMENT FROM CIRCULAR QUEUE

```
int CDequeue ( )
{
if (front = = -1)
print ("Queue is underflow");
else
{
X = CQueue [Front];
if (Front = = Rear)
```

```
Front = Rear = -1;

else

Front = (Front + 1)% maxsize;

}

return (X);

}
```

1.5 Priority Queues

Priority Queue is a Queue in which inserting an item or removing an item can be performed from any position based on some priority.

1.5.1 Applications of Queue

* Batch processing in an operating system

* To implement Priority Queues.

* Priority Queues can be used to sort the elements using Heap Sort.

* Simulation.

* Mathematics user Queueing theory.

* Computer networks where the server takes the jobs of the client as per the queue strategy.

SATHYABAMA
INSTITUTE OF SCIENCE AND TECHNOLOGY
(DEEMED TO BE UNIVERSITY)
Accredited "A" Grade by NAAC | 12B Status by UGC | Approved by AICTE
www.sathyabama.ac.in

School of Computing

Department of Computer Science and Engineering

UNIT - II

Fundamental of Data Structure – SBS1201

UNIT-II

Evaluation of arithmetic expression using stack- Prefix –Infix-Postfix-notations, Converting infix expressions to postfix-Evaluation of postfix expression,-Towers of Hanoi problem.

2.1 APPLICATIONS OF STACK

Some of the applications of stack are :

(i) Evaluating arithmetic expression

(ii) Balancing the symbols

(iii) Towers of Hannoi

(iv) Function Calls.

(v) 8 Queen Problem.

Different Types of Notations To Represent Arithmetic Expression

There are 3 different ways of representing the algebraic expression.

They are

* INFIX NOTATION

* POSTFIX NOTATION

* PREFIX NOTATION

INFIX

In Infix notation, The arithmetic operator appears between the two operands to which it is being applied.

For example : - A / B + C

POSTFIX

The arithmetic operator appears directly after the two operands to which it applies. Also called reverse polish notation. ((A/B) + C)

For example : - AB / C +

PREFIX

The arithmetic operator is placed before the two operands to which it applies. Also called as polish notation. ((A/B) + C)

For example : - +/ABC

INFIX PREFIX (or) POLISH POSTFIX (or) REVERSE

POLISH

1. (A + B) / (C - D) /+AB - CD AB + CD - /

2. A + B*(C - D) +A*B - CD ABCD - * +

3. X * A / B - D - / * XABD X A*B/D-

4. X + Y * (A - B) / +X/*Y - AB - CD XYAB - *CD - / +

(C - D)

5. A * B/C + D + / * ABCD AB * C / D +

2.2. Evaluating Arithmetic Expression

To evaluate an arithmetic expressions, first convert the given infix expression to postfix expression and then evaluate the postfix expression using stack.

Infix to Postfix Conversion

Read the infix expression one character at a time until it encounters the delimiter. "#"

Step 1 : If the character is an operand, place it on to the output.

Step 2 : If the character is an operator, push it onto the stack. If the stack operator has a higher

or equal priority than input operator then pop that operator from the stack and place it onto the output.

Step 3 : If the character is a left paraenthesis, push it onto the stack.

Step 4 : If the character is a right paraenthesis, pop all the operators from the stack till it encounters left parenthesis, discard both the parenthesis in the output.

2.3 Evaluating Postfix Expression

Read the postfix expression one character at a time until it encounters the delimiter `#'.

Step 1 : - If the character is an operand, push its associated value onto the stack.

Step 2 : - If the character is an operator, POP two values from the stack, apply the operator to them and push the result onto the stack.

Recursive Solution

N - represents the number of disks.

Step 1. If N = 1, move the disk from A to C.

Step 2. If N = 2, move the 1^{st} disk from A to B.

Then move the 2^{nd} disk from A to C,

The move the 1^{st} disk from B to C.

Step 3. If N = 3, Repeat the step (2) to more the first 2 disks from A to B using C as intermediate.

Then the 3^{rd} disk is moved from A to C. Then repeat the step (2) to move 2 disks from B to C using A as intermediate.

In general, to move N disks. Apply the recursive technique to move N - 1 disks from A to B using C as an intermediate. Then move the N^{th} disk from A to C. Then again apply the recursive technique to move N - 1 disks from B to C using A as an intermediate.

2.3.1 Evaluation of Postfix Expressions Using Stack [with C program]

How to evaluate postfix expression using stack in C language program?

The compiler finds it convenient to evaluate an expression in its postfix form. The virtues of postfix form include elimination of parentheses which signify priority of evaluation and the elimination of the need to observe rules of hierarchy, precedence and associativity during evaluation of the expression.

As **Postfix expression** is without parenthesis and can be evaluated as two operands and an operator at a time, this becomes easier for the compiler and the computer to handle.

Evaluation rule of a Postfix Expression states:

1. While reading the expression from left to right, push the element in the stack if it is an operand.
2. Pop the two operands from the stack, if the element is an operator and then evaluate it.
3. Push back the result of the evaluation. Repeat it till the end of the expression.

Algorithm

1) Add) to postfix expression.
2) Read postfix expression Left to Right until) encountered
3) If operand is encountered, push it onto Stack
[End If]
4) If operator is encountered, Pop two elements
i) A -> Top element
ii) B-> Next to Top element
iii) Evaluate B operator A
push B operator A onto Stack
5) Set result = pop
6) END

Let's see an example to better understand the algorithm:

Expression: 456*+

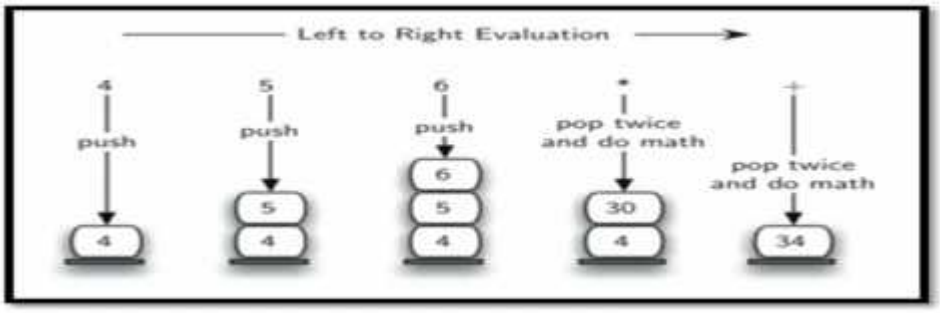


Fig.2.3.1 Evaluating Postfix Expression

Step	Input Symbol	Operation	Stack	Calculation
1.	4	Push	4	
2.	5	Push	4,5	
3.	6	Push	4,5,6	
4.	*	Pop(2 elements) & Evaluate	4	5*6=30
5.		Push result(30)	4,30	
6.	+	Pop(2 elements) & Evaluate	Empty	4+30=34
7.		Push result(34)	34	
8.		No-more elements(pop)	Empty	34(Result)

Result: 34

Evaluation of Postfix Expressions Using Stack

/* This program is for evaluation of postfix expression

* This program assume that there are only four operators

* (*, /, +, -) in an expression and operand is single digit only

* Further this program does not do any error handling e.g.

* it does not check that entered postfix expression is valid

* or not.

* */

Example

```
#include <stdio.h>

#include <ctype.h>


#define MAXSTACK 100 /* for max size of stack */
```

```
#define POSTFIXSIZE 100 /* define max number of charcters in postfix expression */

/* declare stack and its top pointer to be used during postfix expression
evaluation*/
int stack[MAXSTACK];
int top = -1; /* because array index in C begins at 0 */
/* can be do this initialization somewhere else */

/* define push operation */
void push(int item)
{

   if (top >= MAXSTACK - 1) {
      printf("stack over flow");
      return;
   }
   else {
      top = top + 1;
      stack[top] = item;
   }
}

/* define pop operation */
```

```
int pop()
{
   int item;
   if (top < 0) {
      printf("stack under flow");
   }
   else {
      item = stack[top];
      top = top - 1;
      return item;
   }
}

/* define function that is used to input postfix expression and to evaluate it */
void EvalPostfix(char postfix[])
{
   int i;
   char ch;
   int val;
   int A, B;
   /* evaluate postfix expression */
   for (i = 0; postfix[i] != ')'; i++) {
      ch = postfix[i];
```

```
        if (isdigit(ch)) {
            /* we saw an operand,push the digit onto stack
         ch - '0' is used for getting digit rather than ASCII code of digit */
            push(ch - '0');
        }
        else if (ch == '+' || ch == '-' || ch == '*' || ch == '/') {
            /* we saw an operator
* pop top element A and next-to-top elemnet B
* from stack and compute B operator A
*/
            A = pop();
            B = pop();

            switch (ch) /* ch is an operator */
            {
            case '*':
                val = B * A;
                break;
            case '/':
                val = B / A;
                break;
            case '+':
                val = B + A;
```

```
            break;
        case '-':
            val = B - A;
            break;
        }
        /* push the value obtained above onto the stack */
        push(val);
      }
   }
   printf(" \n Result of expression evaluation : %d \n", pop());
}

int main()
{

   int i;

   /* declare character array to store postfix expression */
   char postfix[POSTFIXSIZE];
   printf("ASSUMPTION: There are only four operators(*, /, +, -) in an expression and operand is single digit only.\n");
   printf(" \nEnter postfix expression,\npress right parenthesis ')' for end expression : ");
   /* take input of postfix expression from user */
   for (i = 0; i <= POSTFIXSIZE - 1; i++) {
```

```
        scanf("%c", &postfix[i]);
        if (postfix[i] == ')') /* is there any way to eliminate this if */
        {
            break;
        } /* and break statement */
    }
    /* call function to evaluate postfix expression */
    EvalPostfix(postfix);
    return 0;
}
```

Output

First Run:

Enter postfix expression, press right parenthesis ')' for end expression : 456*+)

Result of expression evaluation : 34

Second Run:

Enter postfix expression, press right parenthesis ')' for end expression: 12345*+*+)

Result of expression evaluation: 47

2.4 Infix to Postfix Conversion

This problem requires you to write a program to convert an infix expression to a postfix expression. The evaluation of an infix expression such as A + B * C requires knowledge of which of the two operations, + and *, should be performed first. In general, A + B * C is to be interpreted as A + (B * C) unless otherwise specified. We say that multiplication takes *precedence* over addition. Suppose that we would now like to convert A + B * C to postfix. Applying the rules of precedence, *we first convert the portion of the expression that is evaluated first*, namely the multiplication. Doing this conversion in stages, we obtain

A	+ B	* C	Given infix form
A	+ B C *		Postfix
A B	C * +		Convert the addition to posfix

The major rules to remember during the conversion process are that the operations with highest precedence are converted first and that after a portion of an expression has been converted to postfix, it is to be treated as a single operand. Let us now consider the same example with the precedence of operators reversed by the deliberate insertion of parentheses.

(A + B) * C	Given infix form
A B + * C	Convert the addition
A B + C *	Convert the multiplication

Note that in the conversion from AB + * C to AB + C *, AB+ was treated as a single operand. The rules for converting from infix to postfix are simple, provided that you know the order of precedence.

We consider five binary operations: addition, subtraction, multiplication, division, and exponentiation. These operations are denoted by the usual operators, +, –, *, /, and ^, respectively. There are three levels of operator precedence. Both * and / have higher precedence than + and –. ^ has higher precedence than * and /. Furthermore, when operators of the same precedence are scanned, +, –, * and / are left associative, but ^ is right associative. Parentheses may be used in infix expressions to override the default precedence.

The postfix form requires no parentheses. The order of the operators in the postfix expressions determines the actual order of operations in evaluating the expression, making the use of parentheses unnecessary.

Input

A collection of *error-free* simple arithmetic expressions. Expressions are presented one per line. The input has an arbitrary number of blanks between any two symbols. A symbol may be a letter (A – Z), an operator (+, – , *, or /), a left parenthesis, or a right parenthesis. Each operand is composed of a single letter. The input expressions are in infix notation.

Example

A + B – C A + B
* C
(A + B) / (C – D)
((A + B) * (C – D) + E) / (F + G)

Output

Your output will consist of the input expression, followed by its corresponding postfix expression. All output (including the original infix expression) must be clearly formatted (or reformatted) and also clearly labeled.

Example

(Only the four postfix expressions corresponding to the above sample input are shown here.) A B + C –
A B C * +
A B + C D - /
A B + C D - * E + F G + /

In converting infix expressions to postfix notation, the following fact should be taken into consideration: In infix form, the order of applying operators is governed by the possible appearance of parentheses and the operator precedence relations; however, in postfix form, the order is simply the "natural" order – i.e., the order of appearance from left to right.

Accordingly, subexpressions within innermost parentheses must first be converted to postfix, so that they can then be treated as single operands. In this fashion, parentheses can be successively eliminated until the entire expression has been converted. The *last* pair of parentheses to be opened within a group of nested parentheses encloses the *first* subexpression within the group to be transformed. This last-in, first-out behavior should immediately suggest the use of a stack.

Your program should utilize the basic stack methods. You will need to PUSH certain symbols on the stack, POP symbols, test to see if the stack is EMPTY, look at the TOP element of the stack, etc.

In addition, you must devise a boolean method that takes two operators and tells you which has higher precedence. This will be helpful, because in Rule 3 below, you need to compare the next symbol to the one on the top of the stack. [Question: what precedence do you assign to '('? You need to answer this question since '(' may be on top of the stack.]

You should formulate the conversion algorithm using the following six rules:

1. Scan the input string (infix notation) from left to right. One pass is sufficient.
2. If the next symbol scanned is an operand, it may be immediately appended to the

postfix string.

3. If the next symbol is an operator,

i. Pop and append to the postfix string every operator on the stack that a. is above the most recently scanned left parenthesis, and has precedence higher than or is a right-associative operator of equal precedence to that of the new operator symbol.

ii. Push the new operator onto the stack.

4. When a left parenthesis is seen, it must be pushed onto the stack.

5. When a right parenthesis is seen, all operators down to the most recently scanned left parenthesis must be popped and appended to the postfix string. Furthermore, this pair of parentheses must be discarded.

6. When the infix string is completely scanned, the stack may still contain some operators. [Why are there no parentheses on the stack at this point?] All the remaining operators should be popped and appended to the postfix string.

Examples

Here are two examples to help you understand how the algorithm works. Each line below demonstrates the state of the postfix string and the stack when the corresponding next infix symbol is scanned. The rightmost symbol of the stack is the top symbol. The rule number corresponding to each line demonstrates which of the six rules was used to reach the current state from that of the previous line.

Example 1

Input expression: A + B * C / D - E

A	A		2
+	A	+	3
B	A B	+	2
*	A B	+ *	3
C	A B C	+ *	2
/	A B C *	+ /	3
D	A B C * D	+ /	2
-	A B C * D / +	-	3
E	A B C * D / + E	-	2
	A B C * D / + E -		6

Example 2

Input expression: (A + B * (C - D)) / E.

Next Symbol	*Postfix String*	*Stack*	*Rule*
(		(	4
A	A	(	2
+	A	(+	3
B	A B	(+	2
*	A B	(+ *	3
(	A B	(+ * (	4
C	A B C	(+ * (	2
-	A B C	(+ * (-	3
D	A B C D	(+ * (-	2
)	A B C D -	(+ *	5
)	A B C D - * +		5
/	A B C D - * +	/	3
E	A B C D - * + E	/	2
	A B C D - * + E /		6

2.5 RECURSIVE ROUTINE FOR TOWERS OF HANOI

Tower of Hanoi, is a mathematical puzzle which consists of three towers (pegs) and more than one rings is as depicted −

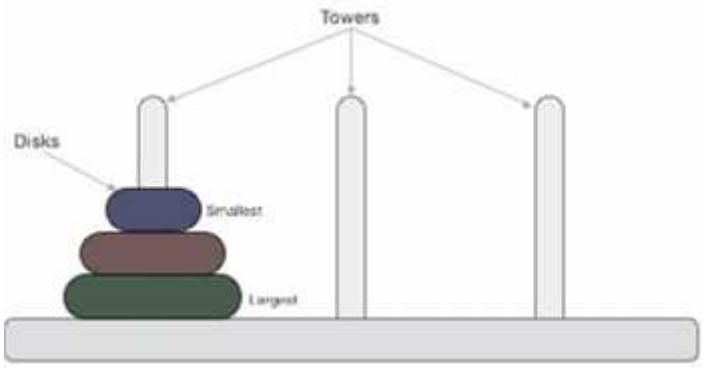


Fig.2.5.1 Tower of Hanoi

These rings are of different sizes and stacked upon in an ascending order, i.e. the smaller one sits over the larger one. There are other variations of the puzzle where the number of disks increase, but the tower count remains the same.

Rules

The mission is to move all the disks to some another tower without violating the sequence of arrangement. A few rules to be followed for Tower of Hanoi are −

- Only one disk can be moved among the towers at any given time.
- Only the "top" disk can be removed.

- No large disk can sit over a small disk.

Following is an animated representation of solving a Tower of Hanoi puzzle with three disks.

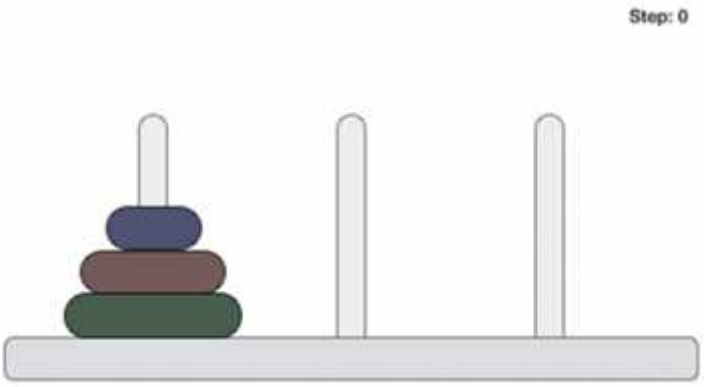


Fig.2.5.2 Tower of Hanoi with 3 Disks

Tower of Hanoi puzzle with n disks can be solved in minimum $\mathbf{2^n-1}$ steps. This presentation shows that a puzzle with 3 disks has taken $\mathbf{2^3 - 1 = 7}$ steps.

Algorithm

To write an algorithm for Tower of Hanoi, first we need to learn how to solve this problem with lesser amount of disks, say → 1 or 2. We mark three towers with name, **source**, **destination** and **aux** (only to help moving the disks). If we have only one disk, then it can easily be moved from source to destination peg.

If we have 2 disks −

- First, we move the smaller (top) disk to aux peg.
- Then, we move the larger (bottom) disk to destination peg.
- And finally, we move the smaller disk from aux to destination peg.

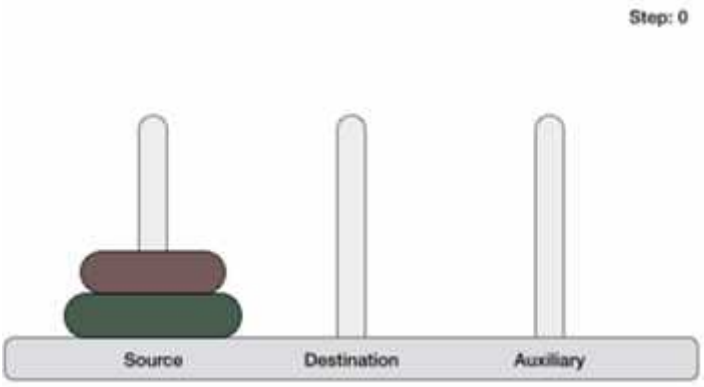


Fig.2.5.3 Tower of Hanoi with 2 Disks

So now, we are in a position to design an algorithm for Tower of Hanoi with more than two disks. We divide the stack of disks in two parts. The largest disk (n^{th} disk) is in one part and all other (n-1) disks are in the second part.

Our ultimate aim is to move disk **n** from source to destination and then put all other (n1) disks onto it. We can imagine to apply the same in a recursive way for all given set of disks.

The steps to follow are −

Step 1 − Move n-1 disks from **source** to **aux**
Step 2 − Move n^{th} disk from **source** to **dest**
Step 3 − Move n-1 disks from **aux** to **dest**

A recursive algorithm for Tower of Hanoi can be driven as follows −

```
START
Procedure Hanoi(disk, source, dest, aux)

  IF disk == 1, THEN
    move disk from source to dest
  ELSE
    Hanoi(disk - 1, source, aux, dest)    // Step 1
    move disk from source to dest         // Step 2
    Hanoi(disk - 1, aux, dest, source)    // Step 3
  END IF

END Procedure
STOP
```

Example:

```
void hanoi (int n, char s, char d, char i)

{

/* n     no. of disks, s    source, d    destination i    intermediate */

if (n = = 1)

{

print (s, d);

return;

}

else

{

hanoi (n - 1, s, i, d);
```

```
print (s, d)

hanoi (n-1, i, d, s);

return;

}

}
```

Function Calls

When a call is made to a new function all the variables local to the calling routine need to be saved, otherwise the new function will overwrite the calling routine variables. Similarly the current location address in the routine must be saved so that the new function knows where to go after it is completed.

RECURSIVE FUNCTION TO FIND FACTORIAL : -

```
int fact (int n)

{

int s;

if (n = = 1)

return (1);

else

s = n * fact (n - 1);

return (s);

}
```

SATHYABAMA
INSTITUTE OF SCIENCE AND TECHNOLOGY
(DEEMED TO BE UNIVERSITY)
Accredited "A" Grade by NAAC | 12B Status by UGC | Approved by AICTE
www.sathyabama.ac.in

School of Computing

Department of Computer Science and Engineering

UNIT - III

Fundamental of Data Structure – SBS1201

UNIT-III

Tree Structures: Binary Trees- Implementation of Binary Trees- Linear Representation of Binary Tree-Linked representation of a Binary Tree. Binary Tree Traversal: Pre order – In order - Post order.

3. TREES

3.1 PRELIMINARIES:

TREE: A tree is a finite set of one or more nodes such that there is a specially designated

node called the Root, and zero or more non empty sub trees T_1, T_2....T_k, each of

whose roots are connected by a directed edge from Root R.

The ADT tree

A tree is a finite set of elements or nodes. If the set is non-empty, one of the nodes is distinguished as the root node, while the remaining (possibly empty) set of nodes are grouped into subsets, each of which is itself a tree. This hierarchical relationship is described by referring to each such subtree as a child of the root, while the root is referred to as the parent of each subtree. If a tree consists of a single node, that node is called a leaf node.

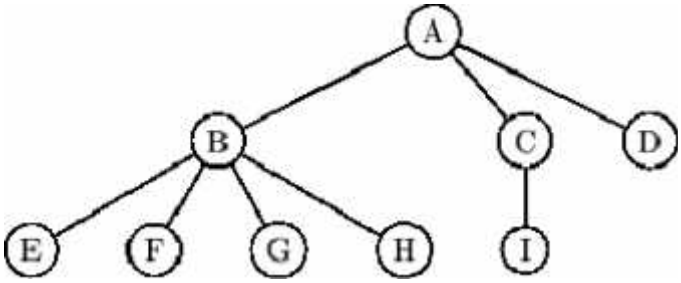


Figure 3.1.1 A simple tree.

It is a notational convenience to allow an empty tree. It is usual to represent a tree using a picture such as Fig. 3.1.1, in which the root node is *A*, and there are three subtrees rooted at *B*, *C* and *D*. The root of the subtree *D* is a leaf node, as are the remaining nodes, *E*, *F*, *G*, *H* and *I*. The node *C* has a single child *I*, while each of *E*, *F*, *G* and *H* have the same parent *B*. The subtrees rooted at a given node are taken to be *ordered*, so the tree in Fig. 3.1.1 is different from the one in which nodes *E* and *F* are interchanged. Thus it makes sense to say that the *first* subtree at *A* has 4 leaf nodes.

Example 3.1.1 Show how to implement the Abstract Data Type tree using lists.

Solution We write [A B C] for the list containing three elements, and distinguish A from [A]. We can represent a tree as a list consisting of the root and a list of the subtrees in order. Thus the list-based representation of the tree in Fig 3.1.1 is

[A [[B [[E] [F] [G] [H]]] [C [I]] [D]]].

ROOT : A node which doesn't have a parent. In the above tree.

NODE : Item of Information.

LEAF : A node which doesn't have children is called leaf or Terminal node.

SIBLINGS : Children of the same parents are said to be siblings,. F, G are siblings. **PATH :** A path from node n, to n_k is defined as a sequence of nodes n_1, n_2,n_3....n_k such that n_i is the parent of n_{i+1}. for . There is exactly only one path from each node to root.

LENGTH : The length is defined as the number of edges on the path.

DEGREE : The number of subtrees of a node is called its degree.

3.2 BINARY TREE

Definition :-

Binary Tree is a tree in which no node can have more than two children.

Maximum number of nodes at level i of a binary tree is 2^{i-1}.

A **binary tree** is a tree which is either empty, or one in which every node:

- has no children; or
- has just a left child; or
- has just a right child; or
- has both a left and a right child.

A **complete** binary tree is a special case of a binary tree, in which all the levels, except perhaps the last, are full; while on the last level, any missing nodes are to the right of all the nodes that are present. An example is shown in Fig. 3.5.

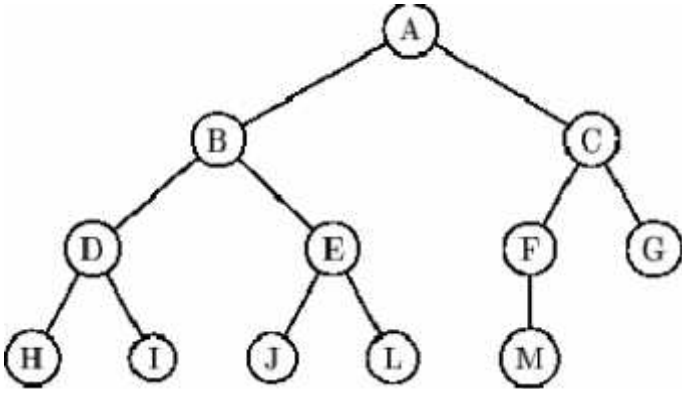


Fig. 3.2.1 A complete binary tree: the only ``missing'' entries can be on the last row.

Example 3.2.1 Give a space - efficient implementation of a complete binary tree in terms of an array A. Describe how to pass from a parent to its two children, and vice-versa.

Solution An obvious one, in which no space is wasted, stores the root of the tree in A[1]; the two children in A[2] and A[3], the next generation at A[4] up to A[7] and so on. An element A[k] has children at A[2k] and A[2k+1], providing they both exists, while the parent of node A[k] is at A[k div 2]. Thus traversing the tree can be done very efficiently.

BINARY TREE NODE DECLARATIONS

Struct TreeNode

{

int Element;

Struct TreeNode *Left ;

Struct TreeNode *Right;

};

COMPARISON BETWEEN

GENERAL TREE & BINARY TREE

General Tree Binary Tree

* General Tree has any * A Binary Tree has not

number of children. more than two children.

FULL BINARY TREE :-

A full binary tree of height h has 2^{h+1} - 1 nodes.

Here height is 3 No. of nodes in full

binary tree is = 2^{3+1} -1

= 15 nodes.

COMPLETE BINARY TREE :

A complete binary tree of height h has between 2^h and 2^{h+1} - 1 nodes. In the bottom level the elements should be filled from left to right.

3.2.1 REPRESENTATION OF A BINARY TREE

There are two ways for representing binary tree, they are

* Linear Representation

* Linked Representation

Linear Representation

The elements are represented using arrays. For any element in position i, the left child is in position 2i, the right child is in position (2i + 1), and the parent is in position (i/2).

Linked Representation

The elements are represented using pointers. Each node in linked representation has three fields, namely,

* Pointer to the left subtree

* Data field

* Pointer to the right subtree

In leaf nodes, both the pointer fields are assigned as NULL.

3.2.2 EXPRESSION TREE

Expression Tree is a binary tree in which the leaf nodes are operands and the interior nodes are operators. Like binary tree, expression tree can also be travesed by inorder, preorder and postorder traversal.

Constructing an Expression Tree

Let us consider postfix expression given as an input for constructing an expression tree by performing the following steps :

1. Read one symbol at a time from the postfix expression.

2. Check whether the symbol is an operand or operator.

(a) If the symbol is an operand, create a one - node tree and push a pointer on to the stack.

(b) If the symbol is an operator pop two pointers from the stack namely T_1 and T_2 and

form a new tree with root as the operator and T_2 as a left child and T_1 as a right child.

A pointer to this new tree is then pushed onto the stack.

3.3 The Search Tree ADT : - Binary Search Tree

Definition : -

Binary search tree is a binary tree in which for every node X in the tree, the values of all the keys in its left subtree are smaller than the key value in X, and the values of all the keys in its right subtree are larger than the key value in X.

Comparision Between Binary Tree & Binary Search Tree

Binary Tree Binary Search Tree

* A tree is said to be a binary * A binary search tree is a binary tree in which

tree if it has atmost two childrens. the key values in the left node is less than the

root and the keyvalues in the right node is greater than the root.

* It doesn't have any order.

Note : * Every binary search tree is a binary tree.

* All binary trees need not be a binary search tree.

DECLARATION ROUTINE FOR BINARY SEARCH TREE

```
Struct TreeNode;

typedef struct TreeNode * SearchTree;
```

```
SearchTree Insert (int X, SearchTree T);

SearchTree Delete (int X, SearchTree T);

int Find (int X, SearchTree T);

int FindMin (Search Tree T);

int FindMax (SearchTree T);

SearchTree MakeEmpty (SearchTree T);

Struct TreeNode

{

int Element ;

SearchTree Left;

SearchTree Right;

};
```

Make Empty :-

This operation is mainly for initialization when the programmer prefer to initialize the first element as a one - node tree.

ROUTINE TO MAKE AN EMPTY TREE :-

```
SearchTree MakeEmpty (SearchTree T)

{

if (T! = NULL)

{

MakeEmpty (T     left);

MakeEmpty (T     Right);

free (T);

}
```

return NULL ;

}

Insert : -

To insert the element X into the tree,

* Check with the root node T

* If it is less than the root,

Traverse the left subtree recursively until it reaches

the T left equals to NULL. Then X is placed in

T left.

* If X is greater than the root.

Traverse the right subtree recursively until it reaches

the T right equals to NULL. Then x is placed in

T Right.

ROUTINE TO INSERT INTO A BINARY SEARCH TREE

SearchTree Insert (int X, searchTree T)

{

if (T = = NULL)

{

T = malloc (size of (Struct TreeNode));

if (T! = NULL) // First element is placed in the root.

{

T →Element = X;

T→ left = NULL;

T →Right = NULL;

}

}

else

if (X < T →Element)

T left = Insert (X, T →left);

else

if (X > T →Element)

T Right = Insert (X, T →Right);

// Else X is in the tree already.

return T;

}

Example : -

To insert 8, 5, 10, 15, 20, 18, 3

* First element 8 is considered as Root.

As 5 < 8, Traverse towards left

10 > 8, Traverse towards Right.

Similarly the rest of the elements are traversed.

Find : -

* Check whether the root is NULL if so then return NULL.

* Otherwise, Check the value X with the root node value (i.e. T data)

(1) If X is equal to T data, return T.

(2) If X is less than T data, Traverse the left of T recursively.

(3) If X is greater than T data, traverse the right of T recursively.

ROUTINE FOR FIND OPERATION

```
Int Find (int X, SearchTree T)

{

If T = = NULL)

Return NULL ;

If (X < T Element)

return Find (X, T →left);

else

If (X > T→ Element)

return Find (X, T →Right);

else

return T; // returns the position of the search element.

}
```

Example : - To Find an element 10 (consider, X = 10)

10 is checked with the Root 10 > 8, Go to the right child of 8

10 is checked with Root 15 10 < 15, Go to the left child of 15.

10 is checked with root 10 (Found)

Find Min :

This operation returns the position of the smallest element in the tree.

To perform FindMin, start at the root and go left as long as there is a left child. The stopping point is the smallest element.

RECURISVE ROUTINE FOR FINDMIN

```
int FindMin (SearchTree T)
```

{

if (T = = NULL);

return NULL ;

else if (T →left = = NULL)

return T;

else

return FindMin (T → left);

Example : -

Root T

T

(a) T! = NULL and T→left!=NULL, (b) T! = NULL and T→left!=NULL,

Traverse left Traverse left

Min T

(c) Since T left is Null, return T as a minimum element.

NON - RECURSIVE ROUTINE FOR FINDMIN

int FindMin (SearchTree T)

{

if (T! = NULL)

while (T →Left ! = NULL)

T = T →Left ;

return T;

}

FindMax

FindMax routine return the position of largest elements in the tree. To perform a FindMax, start at the root and go right as long as there is a right child. The stopping point is the largest element.

RECURSIVE ROUTINE FOR FINDMAX

```
int FindMax (SearchTree T)

{

if (T = = NULL)

return NULL ;

else if (T →Right = = NULL)

return T;

else FindMax (T →Right);

}
```

Example :-

Root T

(a) T! = NULL and T→Right!=NULL, (b) T! = NULL and T→Right!=NULL,

Traverse Right Traverse Right

Max

(c) Since T Right is NULL, return T as a Maximum element.

NON - RECURSIVE ROUTINE FOR FINDMAX

```
int FindMax (SearchTree T)

{

if (T! = NULL)

while (T Right ! = NULL)

T = T →Right ;
```

return T ;

}

Delete :

Deletion operation is the complex operation in the Binary search tree. To delete an element, consider the following three possibilities.

CASE 1 Node to be deleted is a leaf node (ie) No children.

CASE 2 Node with one child.

CASE 3 Node with two children.

CASE 1 Node with no children (Leaf node)

If the node is a leaf node, it can be deleted immediately.

Delete : 8

After the deletion

CASE 2 : - Node with one child

If the node has one child, it can be deleted by adjusting its parent pointer that points to its child node.

To Delete 5

before deletion After deletion

To delete 5, the pointer currently pointing the node 5 is now made to to its child node 6.

Case 3 : Node with two children

It is difficult to delete a node which has two children. The general strategy is to replace the data of the node to be deleted with its smallest data of the right subtree and recursively delete that node.

DELETION ROUTINE FOR BINARY SEARCH TREES

SearchTree Delete (int X, searchTree T)

{

```
int Tmpcell ;
if (T = = NULL)
Error ("Element not found");
else
if (X < T →Element) // Traverse towards left
T →Left = Delete (X, T  Left);
else
if (X > T  Element) // Traverse towards right
T →Right = Delete (X, T →Right);
// Found Element tobe deleted
else
// Two children
if (T→  Left && T→  Right)
{ // Replace with smallest data in right subtree
Tmpcell = FindMin (T→  Right);
T →Element = Tmpcell  Element ;
T →Right = Delete (T →  Element; T →Right);
}
else // one or zero children
{
Tmpcell = T;
if (T →Left = = NULL)
T = T→  Right;
```

```
else if (T→     Right = = NULL)

T = T →Left ;

free (TmpCell);

}

return T;

}
```

3.4 Tree Representation

Tree represents the nodes connected by edges. We will discuss binary tree or binary search tree specifically.

Binary Tree is a special datastructure used for data storage purposes. A binary tree has a special condition that each node can have a maximum of two children. A binary tree has the benefits of both an ordered array and a linked list as search is as quick as in a sorted array and insertion or deletion operation are as fast as in linked list.

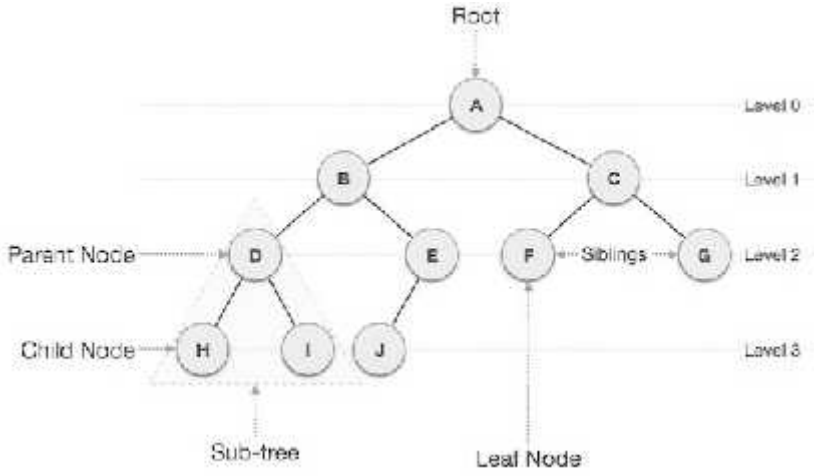


Fig.3.4.1 Tree Representation with Levels

Important Terms

Following are the important terms with respect to tree.

- **Path** − Path refers to the sequence of nodes along the edges of a tree.
- **Root** − The node at the top of the tree is called root. There is only one root per tree and one path from the root node to any node.
- **Parent** − Any node except the root node has one edge upward to a node called parent.

- **Child** − The node below a given node connected by its edge downward is called its child node.
- **Leaf** − The node which does not have any child node is called the leaf node.
- **Subtree** − Subtree represents the descendants of a node.
- **Visiting** − Visiting refers to checking the value of a node when control is on the node.
- **Traversing** − Traversing means passing through nodes in a specific order.
- **Levels** − Level of a node represents the generation of a node. If the root node is at level 0, then its next child node is at level 1, its grandchild is at level 2, and so on.
- **keys** − Key represents a value of a node based on which a search operation is to be carried out for a node.

Binary Search Tree Representation

Binary Search tree exhibits a special behavior. A node's left child must have a value less than its parent's value and the node's right child must have a value greater than its parent value.

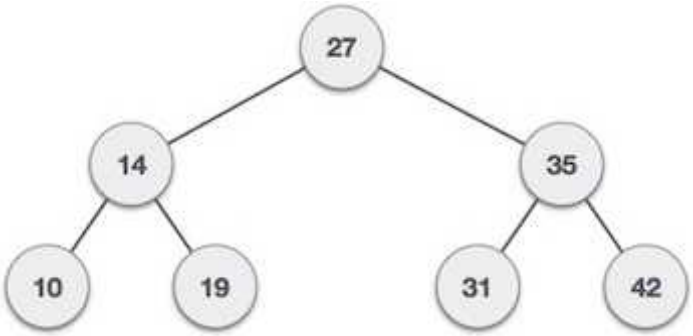


Fig.3.4.2 Binary search Tree

We're going to implement tree using node object and connecting them through references.

Tree Node

The code to write a tree node would be similar to what is given below. It has a data part and references to its left and right child nodes.

```
struct node {
   int data;
   struct node *leftChild;
   struct node *rightChild;
};
```

In a tree, all nodes share common construct.

BST Basic Operations

The basic operations that can be performed on a binary search tree data structure, are the following −

- **Insert** − Inserts an element in a tree/create a tree.
- **Search** − Searches an element in a tree.
- **Preorder Traversal** − Traverses a tree in a pre-order manner.
- **Inorder Traversal** − Traverses a tree in an in-order manner.
- **Postorder Traversal** − Traverses a tree in a post-order manner.

We shall learn creating (inserting into) a tree structure and searching a data item in a tree in this chapter. We shall learn about tree traversing methods in the coming chapter.

Insert Operation

The very first insertion creates the tree. Afterwards, whenever an element is to be inserted, first locate its proper location. Start searching from the root node, then if the data is less than the key value, search for the empty location in the left subtree and insert the data. Otherwise, search for the empty location in the right subtree and insert the data.

Algorithm

```
If root is NULL
   then create root node
return

If root exists then
   compare the data with node.data

   while until insertion position is located

      If data is greater than node.data
         goto right subtree
      else
         goto left subtree

   endwhile

   insert data

end If
```

Implementation

The implementation of insert function should look like this −

```
void insert(int data) {
  struct node *tempNode = (struct node*) malloc(sizeof(struct node));
  struct node *current;
  struct node *parent;

  tempNode->data = data;
  tempNode->leftChild = NULL;
  tempNode->rightChild = NULL;

  //if tree is empty, create root node
  if(root == NULL) {
    root = tempNode;
  } else {
    current = root;
    parent  = NULL;

    while(1) {
      parent = current;

      //go to left of the tree
      if(data < parent->data) {
        current = current->leftChild;

        //insert to the left
        if(current == NULL) {
          parent->leftChild = tempNode;
          return;
        }
      }

      //go to right of the tree
      else {
        current = current->rightChild;

        //insert to the right
        if(current == NULL) {
          parent->rightChild = tempNode;
          return;
        }
      }
    }
  }
}
```

Search Operation

Whenever an element is to be searched, start searching from the root node, then if the data is less than the key value, search for the element in the left subtree. Otherwise, search for the element in the right subtree. Follow the same algorithm for each node.

Algorithm

```
If root.data is equal to search.data
  return root
else
  while data not found

    If data is greater than node.data
      goto right subtree
    else
      goto left subtree

    If data found
      return node
  endwhile

  return data not found

end if
```

The implementation of this algorithm should look like this.

```
struct node* search(int data) {
  struct node *current = root;
  printf("Visiting elements: ");

  while(current->data != data) {
    if(current != NULL)
    printf("%d ",current->data);

    //go to left tree

    if(current->data > data) {
      current = current->leftChild;
    }
    //else go to right tree
    else {
      current = current->rightChild;
    }

    //not found
    if(current == NULL) {
      return NULL;
    }
```

```
    return current;
  }
}
```

3.5 TREE TRAVERSAL:

Traversal is a process to visit all the nodes of a tree and may print their values too. Because, all nodes are connected via edges (links) we always start from the root (head) node. That is, we cannot randomly access a node in a tree. There are three ways which we use to traverse a tree −

- In-order Traversal
- Pre-order Traversal
- Post-order Traversal

Generally, we traverse a tree to search or locate a given item or key in the tree or to print all the values it contains.

In-order Traversal

In this traversal method, the left subtree is visited first, then the root and later the right sub-tree. We should always remember that every node may represent a subtree itself.

If a binary tree is traversed **in-order**, the output will produce sorted key values in an ascending order.

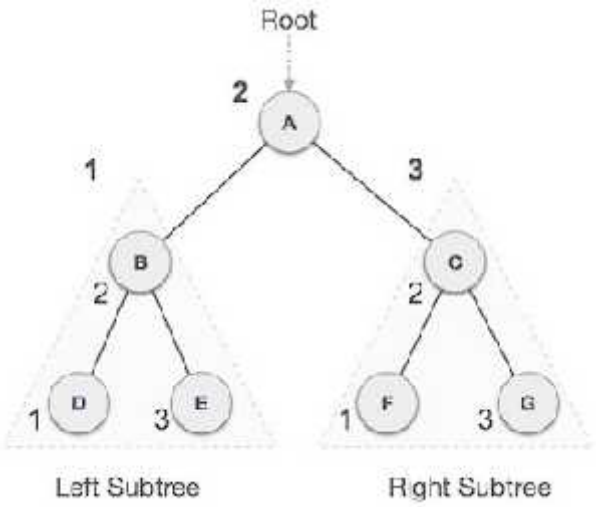


Fig.3.5.1 Inorder Tree

We start from **A**, and following in-order traversal, we move to its left subtree **B**. **B** is also traversed in-order. The process goes on until all the nodes are visited. The output of inorder traversal of this tree will be −

D → B → E → A → F → C → G

Algorithm

Until all nodes are traversed −
Step 1 − Recursively traverse left subtree.
Step 2 − Visit root node.
Step 3 − Recursively traverse right subtree.

Pre-order Traversal

In this traversal method, the root node is visited first, then the left subtree and finally the right subtree.

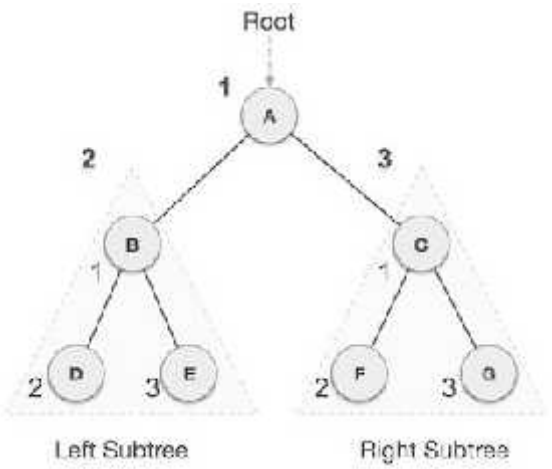


Fig.3.5.2 Pre-order Tree

We start from **A**, and following pre-order traversal, we first visit **A** itself and then move to its left subtree **B**. **B** is also traversed pre-order. The process goes on until all the nodes are visited. The output of pre-order traversal of this tree will be −

A → B → D → E → C → F → G

Algorithm

Until all nodes are traversed −
Step 1 − Visit root node.
Step 2 − Recursively traverse left subtree.
Step 3 − Recursively traverse right subtree.

Post-order Traversal

In this traversal method, the root node is visited last, hence the name. First we traverse the left subtree, then the right subtree and finally the root node.

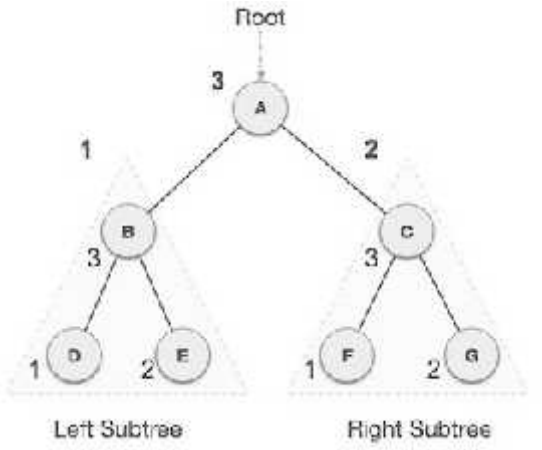


Fig.3.5.3 Post-order Tree

We start from **A**, and following Post-order traversal, we first visit the left subtree **B**. **B** is also traversed post-order. The process goes on until all the nodes are visited. The output of post-order traversal of this tree will be −

D → E → B → F → G → C → A

Algorithm

Until all nodes are traversed −
Step 1 − Recursively traverse left subtree.
Step 2 − Recursively traverse right subtree.
Step 3 − Visit root node.

3.6 BINARY HEAP

The efficient way of implementing priority queue is Binary Heap. Binary heap is merely referred as Heaps, Heap have two properties namely

* Structure property

* Heap order property.

Like AVL trees, an operation on a heap can destroy one of the properties, so a heap operation must not terminate until all heap properties are in order. Both the operations require the average running time as O(log N).

Structure Property

A heap should be complete binary tree, which is a completely filled binary tree with the possible exception of the bottom level, which is filled from left to right.

A complete binary tree of height H has between 2^H and 2^{H+1} -1 nodes.

For example if the height is 3. Then the numer of nodes will be between 8 and 15. (ie) (2^3 and 2^4-1).

For any element in array position i, the left child is in position 2i, the right child is in position 2i + 1, and the parent is in i/2. As it is represented as array it doesn't require pointers and also the operations required to traverse the tree are extremely simple and fast. But the only disadvantage is to specify the maximum heap size in advance.

Heap Order Property

In a heap, for every node X, the key in the parent of X is smaller than (or equal to) the key in X, with the exception of the root (which has no parent).

This property allows the deletemin operations to be performed quickly has the minimum element can always be found at the root. Thus, we get the FindMin operation in constant time.

Declaration for priority queue

```
Struct Heapstruct;

typedef struct Heapstruct * priority queue;

PriorityQueue Initialize (int MaxElements);

void insert (int X, PriorityQueue H);

int DeleteMin (PriorityQueue H);

Struct Heapstruct

{

int capacity;

int size;

int *Elements;

};
```

Initialization

```
PriorityQueue Initialize (int MaxElements)
```

```
{

PriorityQueue H;

H = malloc (sizeof (Struct Heapstruct));

H    Capacity = MaxElements;

H    size = 0;

H     elements [0] = MinData;

return H;
```

BASIC HEAP OPERATIONS

To perform the insert and DeleteMin operations ensure that the heap order property is maintained.

Insert Operation

To insert an element X into the heap, we create a hole in the next available location, otherwise the tree will not be complete. If X can be placed in the hole without violating heap order, then place the element X there itself. Otherewise, we slide the element that is in the hole's parent node into the hole, thus bubbling the hole up toward the root. This process continues until X can be placed in the hole. This general strategy is known as Percolate up, in which the new element is percolated up the heap until the correct location is found.

Routine To Insert Into A Binary Heap

```
void insert (int X, PriorityQueue H)

{

int i;

If (Isfull (H))

{

Error (" priority queue is full");

return;

}
```

```
for (i = ++H→    size; H→    Elements [i/2] > X; i/=2)

/* If the parent value is greater than X, then place the element of parent

node into the hole */.

H→    Elements [i] = H→Elements [i/2];

H→    elements [i] = X; // otherwise, place it in the hole.

}
```

DeleteMin

DeleteMin Operation is deleting the minimum element from the Heap.

In Binary heap the minimum element is found in the root. When this minimum is removed, a hole is created at the root. Since the heap becomes one smaller, makes the last element X in the heap to move somewhere in the heap.

If X can be placed in hole without violating heaporder property place it.

Otherwise, we slide the smaller of the hole's children into the hole, thus pushing the hole down one level. We repeat until X can be placed in the hole. This general strategy is known as perculate down.

ROUTINE TO PERFORM DELETEMIN IN A BINARY HEAP

```
int Deletemin (PriorityQueue H)

{

int i, child;

int MinElement, LastElement;

if (IsEmpty (H))

{

Error ("Priority queue is Empty");

return H →    Elements [0];

}
```

```
MinElement = H →    Elements [1];
LastElement = H→Elements [H→     size - -];
for (i = 1; i * 2 < = H    size; i = child)
{
/* Find Smaller Child */
child = i * 2;
if (child ! = H →    size && H →Elements [child + 1]
< H →Elements [child])
child + +;
// Percolate one level down
if (LastElement > H →Elements [child])
H→     Elements [i] = H →    Elements [child];
else
break ;
}
H →    Elements [i] = LastElement;
return MinElement;
}
```

OTHER HEAP OPERATIONS

The other heap operations are

(i) Decrease - key

(ii) Increase - key

(iii) Delete

(iv) Build Heap

DECREASE KEY

The Decreasekey (P, , H) operation decreases the value of the key at position P by a positive amount . This may violate the heap order property, which can be fixed by percolate up.

Increase - Key

The increase - key (p, , H) operation increases the value of the key at position p by a positive amount . This may violate heap order property, which can be fixed by percolate down.

3.7 HASHING:

Hash Table

The hash table data structure is an array of some fixed size, containing the keys. A key is a value associated with each record.

Hashing Function

A hashing function is a key - to - address transformation, which acts upon a given key to compute the relative position of the key in an array.

A simple Hash function

HASH (KEYVALUE) = KEYVALUE MOD TABLESIZE

Example : - Hash (92)

Hash (92) = 92 mod 10 = 2

The keyvalue `92' is placed in the relative location `2'.

ROUTINE FOR SIMPLE HASH FUNCTION

```
Hash (Char *key, int Table Size)

{

int Hashvalue = 0;

while (* key ! = `\0')
```

Hashval + = * key ++;

return Hashval % Tablesize;

}

Some of the Methods of Hashing Function

1. Module Division

2. Mid - Square Method

3. Folding Method

4. PSEUDO Random Method

5. Digit or Character Extraction Method

6. Radix Transformation.

Collisions

Collision occurs when a hash value of a record being inserted hashes to an address (i.e. Relative position) that already contain a different record. (ie) When two key values hash to the same position.

Collision Resolution

The process of finding another position for the collide record.

Some of the Collision Resolution Techniques

1. Seperate Chaining

2. Open Addressing

3. Multiple Hashing

Seperate Chaining

Seperate chaining is an open hashing technique. A pointer field is added to each record location. When an overflow occurs this pointer is set to point to overflow blocks making a linked list.

In this method, the table can never overflow, since the linked list are only extended upon the arrival of new keys.

Insertion

To perform the insertion of an element, traverse down the appropriate list to check whether the element is already in place.

If the element turns to be a new one, it is insertd either at the front of the list or at the end of the list.

If it is a duplicate element, an extra field is kept and placed.

INSERT 10 :

Hash (k) = k% Tablesize

Hash (10) = 10 % 10

NSERT 11 :

Hash (11) = 11 % 10

Hash (11) = 1

INSERT 81 :

Hash (81) = 81% 10

Hash (81) = 1

The element 81 collides to the same hash value 1. To place the value 81 at this position perform the following.

Traverse the list to check whether it is already present.

Since it is not already present, insert at end of the list. Similarly the rest of the elements are inserted.

ROUTINE TO PERFORM INSERTION

```
void Insert (int key, Hashtable H)

{

Position Pos, Newcell;

List L;
```

```
/* Traverse the list to check whether the key is already present */
Pos = FIND (Key, H);
If (Pos = = NULL) /* Key is not found */
{
Newcell = malloc (size of (struct ListNode));
If (Newcell ! = NULL)
(
L = H    Thelists [Hash (key, H   Tablesize)];
Newcell→    Next = L →   Next;
Newcell →   Element = key;
/* Insert the key at the front of the list */
L →Next = Newcell;
}
}
}
```

FIND ROUTINE

```
Position Find (int key, Hashtable H)
{
Position P;
List L;
L = H→    Thelists [Hash (key, H→Tablesize)];
P = L→    Next;
```

while (P! = NULL && P Element ! = key)

P = p → Next;

return p;

}

Advantage

More number of elements can be inserted as it uses array of linked lists.

Disadvantage of Seperate Chaining

* It requires pointers, which occupies more memory space.

* It takes more effort to perform a search, since it takes time to evaluate the hash function and

also to traverse the list.

OPEN ADDRESSING

Open addressing is also called **closed Hashing**, which is an alternative to resolve the collisions with linked lists.

In this hashing system, if a collision occurs, alternative cells are tried until an empty cell is found. (ie) cells $h_0(x)$, $h_1(x)$, $h_2(x)$.....are tried in succession.

There are three common collision resolution strategies. They are

(i) Linear Probing

(ii) Quadratic probing

(iii) Double Hashing.

LINEAR PROBING

In linear probing, for the i^{th} probe the position to be tried is (h(k) + i) mod tablesize, where F(i) = i, is the linear function.

In linear probing, the position in which a key can be stored is found by sequentially searching all position starting from the position calculated by the hash function until an empty cell is found.

If the end of the table is reached and no empty cells has been found, then the search is continued from the beginning of the table. It has a tendency to create clusters in the table.

Advantage :

* It doesn't requires pointers

Disadvantage

* It forms clusters, which degrades the performance of the hash table for storing and retrieving

SATHYABAMA
INSTITUTE OF SCIENCE AND TECHNOLOGY
(DEEMED TO BE UNIVERSITY)
Accredited "A" Grade by NAAC | 12B Status by UGC | Approved by AICTE
www.sathyabama.ac.in

School of Computing

Department of Computer Science and Engineering

UNIT - IV

Fundamental of Data Structure – SBS1201

UNIT-4

Sorting Techniques: Bubble Sort- Merge Sort - Shell Sort- Insertion Sort- Selection Sort-Quick Sort- Heap Sort. Searching Techniques: Sequential Search- Binary Search- Hashing- Indexing.

4.1 SEARCHING AND SORTING

Searching is used to find the location where an element is available. There are two types of search techniques. They are:

1. Linear or sequential search
2. Binary search

Sorting allows an efficient arrangement of elements within a given data structure. It is a way in which the elements are organized systematically for some purpose. For example, a dictionary in which words are arranged in alphabetical order and telephone director in which the subscriber names are listed in alphabetical order. There are many sorting techniques out of which we study the following.

1. Bubble sort
2. Quick sort
3. Selection sort and
4. Heap sort

4.2 LINEAR SEARCH

This is the simplest of all searching techniques. In this technique, an ordered or unordered list will be searched one by one from the beginning until the desired element is found. If the desired element is found in the list then the search is successful otherwise unsuccessful.

Suppose there are 'n' elements organized sequentially on a List. The number of comparisons required to retrieve an element from the list, purely depends on where the element is stored in the list. If it is the first element, one comparison will do; if it is second element two comparisons are necessary and so on. On an average you need [(n+1)/2] comparison's to search an element. If search is not successful, you would need 'n' comparisons.

The time complexity of linear search is O(n).

Algorithm:

Let array a[n] stores n elements. Determine whether element 'x' is present or not.

Example 1:

Suppose we have the following unsorted list: 45, 39, 8, 54, 77, 38, 24, 16, 4, 7, 9, 20

If we are searching for: 45, we'll look at 1 element before success

39, we'll look at 2 elements before success
8, we'll look at 3 elements before success
54, we'll look at 4 elements before success
77, we'll look at 5 elements before success
38 we'll look at 6 elements before success
24, we'll look at 7 elements before success
16, we'll look at 8 elements before success
4, we'll look at 9 elements before success
7, we'll look at 10 elements before success
9, we'll look at 11 elements before success
20, we'll look at 12 elements before success

For any element not in the list, we'll look at 12 elements before failure

Example 2:

Let us illustrate linear search on the following 9 elements:

Index	***0***	***1***	***2***	***3***	***4***	***5***	***6***	***7***	***8***
Elements	**-15**	**-6**	**0**	**7**	**9**	**23**	**54**	**82**	**101**

Searching different elements is as follows:

1. Searching for x = 7 Search successful, data found at 3rd position

2. Searching for x = 82 Search successful, data found at 7th position

3. Searching for x = 42 Search un-successful, data not found

A Recursive program for linear search:

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

void linear_search(int a[], int data, int position, int n)
{
        int mid;
        if(position < n)
        {
                if(a[position] == data)
                        printf("\n Data Found at %d ", position);
                else
                        linear_search(a, data, position + 1, n);
}
else
                printf("\n Data not found");
 }

 void
 main()
 {
```

```
        int a[25], i, n, data;
        clrscr();
        printf("\n Enter the number of elements: ");
        scanf("%d", &n);
        printf("\n Enter the elements: ");
        for(i = 0; i < n; i++)
        {
                scanf("%d", &a[i]);
        }
        printf("\n Enter the element to be seached: ");
        scanf("%d",          &data);
        linear_search(a,   data,   0,
        n); getch();
}
```

4.3 BINARY SEARCH

If we have 'n' records which have been ordered by keys so that $x_1 < x_2 < \ldots < x_n$. When we are given a element 'x', binary search is used to find the corresponding element from the list. In case 'x' is present, we have to determine a value 'j' such that a[j] = x (successful search). If 'x' is not in the list then j is to set to zero (un successful search).

In Binary search we jump into the middle of the file, where we find key a[mid], and compare 'x' with a[mid]. If x = a[mid] then the desired record has been found. If x < a[mid] then 'x' must be in that portion of the file that precedes a[mid]. Similarly, if a[mid] > x, then further search is only necessary in that part of the file which follows a[mid]. If we use recursive procedure of finding the middle key a[mid] of the un-searched portion of a file, then every un-successful comparison of 'x' with a[mid] will eliminate roughly half the un-searched portion from consideration.

Since the array size is roughly halved after each comparison between 'x' and a[mid], and since an array of length 'n' can be halved only about $\log_2 n$ times before reaching a trivial length, the worst case complexity of Binary search is about $\log_2 n$

Algorithm:
Let array a[n] of elements in increasing order, $n \geq 0$, determine whether 'x' is present, and if so, set j such that x = a[j] else return 0.

```
binsrch(a[], n, x)
{
        low = 1; high = n;
        while (low ≤ high) do
        {
                mid = ⌊ (low + high)/2 ⌊
                if (x < a[mid])
                        high = mid – 1;
                else if (x > a[mid])
                        low = mid + 1;
                else return mid;
        }
        return 0;
 }
```

low and *high* are integer variables such that each time through the loop either 'x' is found or

low is increased by at least one or *high* is decreased by at least one. Thus we have two sequences of integers approaching each other and eventually *low* will become greater than *high* causing termination in a finite number of steps if 'x' is not present.

Example 1:

Let us illustrate binary search on the following 12 elements:

Index	***1***	***2***	***3***	***4***	***5***	***6***	***7***	***8***	***9***	***10***	***11***	***12***
Elements	**4**	**7**	**8**	**9**	**16**	**20**	**24**	**38**	**39**	**45**	**54**	**77**

If we are searching for x = 4: (This needs 3 comparisons)
low = 1, high = 12, mid = 13/2 = 6,
check 20 low = 1, high = 5, mid = 6/2 =
3, check 8
low = 1, high = 2, mid = 3/2 = 1, check 4,
found

If we are searching for x = 7: (This needs 4 comparisons)
low = 1, high = 12, mid = 13/2 = 6,
check 20 low = 1, high = 5, mid = 6/2 =
3, check 8
low = 1, high = 2, mid = 3/2 = 1,
check 4
low = 2, high = 2, mid = 4/2 = 2, check 7,
found

If we are searching for x = 8: (This needs 2 comparisons)
low = 1, high = 12, mid = 13/2 = 6,
check 20
low = 1, high = 5, mid = 6/2 = 3, check 8,
found

If we are searching for x = 9: (This needs 3 comparisons)
low = 1, high = 12, mid = 13/2 = 6,
check 20 low = 1, high = 5, mid = 6/2 =
3, check 8
low = 4, high = 5, mid = 9/2 = 4, check 9,
found

If we are searching for x = 16: (This needs 4 comparisons)
low = 1, high = 12, mid = 13/2 = 6,
check 20 low = 1, high = 5, mid = 6/2 =
3, check 8
low = 4, high = 5, mid = 9/2 = 4,
check 9
low = 5, high = 5, mid = 10/2 = 5, check 16,
found

If we are searching for x = 20: (This needs 1

comparison)
low = 1, high = 12, mid = 13/2 = 6, check 20,
found

If we are searching for x = 24: (This needs 4 comparisons)
low = 1, high = 12, mid = 13/2 = 6,
check 20 low = 7, high = 12, mid = 19/2
= 9, check 39 low = 7, high = 10, mid =
17/2 = 8, check 38
low = 7, high = 7, mid = 14/2 = 7, check 24, *found*

If we are searching for x = 38: (This needs 3 comparisons)
low = 1, high = 12, mid = 13/2 = 6,
check 20 low = 7, high = 12, mid = 19/2
= 9, check 39
low = 7, high = 10, mid = 17/2 = 8, check 38, *found*

If we are searching for x = 39: (This needs 2 comparisons)
low = 1, high = 12, mid = 13/2 = 6, check 20
low = 7, high = 12, mid = 19/2 = 9, check 39, *found*

If we are searching for x = 45: (This needs 4 comparisons)
low = 1, high = 12, mid = 13/2 = 6, check
20 low = 7, high = 12, mid = 19/2 = 9,
check 39 low = 10, high = 12, mid = 22/2
= 11, check 54
low = 10, high = 10, mid = 20/2 = 10, check 45, *found*

If we are searching for x = 54: (This needs 3 comparisons)
low = 1, high = 12, mid = 13/2 = 6,
check 20 low = 7, high = 12, mid = 19/2
= 9, check 39
low = 10, high = 12, mid = 22/2 = 11, check 54, *found*

If we are searching for x = 77: (This needs 4 comparisons)
low = 1, high = 12, mid = 13/2 = 6, check
20 low = 7, high = 12, mid = 19/2 = 9,
check 39 low = 10, high = 12, mid = 22/2
= 11, check 54
low = 12, high = 12, mid = 24/2 = 12, check 77, *found*
The number of comparisons necessary by search element:

20 – requires 1 comparison; 8 and 39 – requires 2 comparisons;
4, 9, 38, 54 – requires 3 comparisons; and 7, 16, 24, 45, 77 – requires 4 comparisons

Summing the comparisons, needed to find all twelve items and dividing by 12, yielding 37/12 or approximately 3.08 comparisons per successful search on the average.

Time Complexity:

The time complexity of binary search in a successful search is O(log n) and for an unsuccessful search is O(log n).

A non-recursive program for binary search:

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

main()
{
        int number[25], n, data, i, flag = 0, low, high, mid;
        clrscr();
        printf("\n Enter the number of elements: ");
        scanf("%d", &n);
        printf("\n Enter the elements in ascending order: ");
        for(i = 0; i < n; i++)
                scanf("%d",
                &number[i]);
        printf("\n Enter the element to be searched: ");
        scanf("%d", &data); low = 0; high =  n-1; while(low <= high)
        {
        mid = (low + high)/2;
                if(number[mid] == data)

{
                        flag = 1;
                        break;
}
else
{
                        if(data < number[mid])
                                high = mid - 1;
else
                                low = mid + 1;
                }
        }
        if(flag == 1)
                printf("\n Data found at location: %d", mid + 1);
else
                printf("\n Data Not Found ");
 }
```

4.4 BUBBLE SORT

The bubble sort is easy to understand and program. The basic idea of bubble sort is to pass through the file sequentially several times. In each pass, we compare each element in the file with its successor i.e., X[i] with X[i+1] and interchange two element when they are not in proper order. We will illustrate this sorting technique by taking a specific example. Bubble sort is also called as exchange sort.

Consider the array x[n] which is stored in memory as shown below:

X[0]	X[1]	X[2]	X[3]	X[4]	X[5]
33	44	22	11	66	55

Suppose we want our array to be stored in ascending order. Then we pass through the array 5 times as described below:

Pass 1: (first element is compared with all other elements)

We compare X[i] and X[i+1] for i = 0, 1, 2, 3, and 4, and interchange X[i] and X[i+1] if X[i] > X[i+1]. The process is shown below:

X[0]	X[1]	X[2]	X[3]	X[4]	X[5]	Remarks
33	44	22	11	66	55	
	22	44				
		11	44			
			44	66		
				55	66	
33	22	11	44	55	66	

The biggest number 66 is moved to (bubbled up) the right most position in the array.

Pass 2: (second element is compared)

We repeat the same process, but this time we don't include X[5] into our comparisons. i.e., we compare X[i] with X[i+1] for i=0, 1, 2, and 3 and interchange X[i] and X[i+1] if X[i] > X[i+1]. The process is shown below:

X[0]	X[1]	X[2]	X[3]	X[4]	Remarks

33	22	11	44	55	
22	33				
	11	33			
		33	44		
			44	55	
22	11	33	44	55	

The second biggest number 55 is moved now to X[4].

Pass 3: (third element is compared)

We repeat the same process, but this time we leave both X[4] and X[5]. By doing this, we move the third biggest number 44 to X[3].

X[0]	X[1]	X[2]	X[3]	Remarks
22	11	33	44	
11	22			
	22	33		
		33	44	
11	22	33	44	

Pass 4: (fourth element is compared)

We repeat the process leaving X[3], X[4], and X[5]. By doing this, we move the fourth biggest number
33 to X[2].

X[0]	X[1]	X[2]	Remarks
11	22	33	
11	22		
	22	33	

Pass 5: (fifth element is compared)

We repeat the process leaving X[2], X[3], X[4], and X[5]. By doing this, we move the fifth biggest number 22 to X[1]. At this time, we will have the smallest number 11 in X[0]. Thus, we see that we can sort the array of size 6 in 5 passes.

For an array of size n, we required (n-1)
passes.

Program for Bubble Sort:

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

```
void          bubblesort(int
x[],int n)
{
        int i,  j,  t;
        for (i = 0; i < n; i++)
        {
                for (j = 0; j <n-i; j++)
                {
                        if (x[j] > x[j+1])
                        {
                                t = x[j];
                                x[j] = x[j+1];
                                x[j+1] = t;
                        }
                }
        }
}

main()
{
int i, n, x[25];
clrscr();
printf("\n Enter the number of elements: ");
scanf("%d",&n);       printf("\n
Enter Data:"); for(i = 0; i < n ;
i++)
        scanf("%d", &x[i]);
bubblesort(x,n);
printf ("\nArray Elements after sorting: ");
for (i = 0; i < n; i++)
          printf ("%5d", x[i]);
}
```

Time Complexity:

The bubble sort method of sorting an array of size n requires (n-1) passes and (n-1) comparisons on each pass. Thus the total number of comparisons is (n-1) * (n-1) = n^2 – 2n + 1, which is O(n^2). Therefore bubble sort is very inefficient when there are more elements to sorting.

4.5 SELECTION SORT

Now, you will learn another sorting technique, which is more efficient than bubble sort and the insertion sort. This sort, as you will see, will not require no more than n-1 interchanges. The sort we are talking about is selection sort.

Suppose x is an array of size n stored in memory. The selection sort algorithm first selects the smallest element in the array x and place it at array position 0; then it selects the next smallest element in the array x and place it at array position 1. It simply continues this procedure until it places the biggest element in the last position of the array. We will now present to you an algorithm for selection sort.

The array is passed through (n-1) times and the smallest element is placed in its respective position in the array as detailed below:

Pass 1:
Find the location j of the smallest element in the array x [0], x[1], ... x[n-1], and then interchange x[j]
with x[0]. Then x[0] is
sorted.

Pass 2:
Leave the first element and find the location j of the smallest element in the sub-array x[1], x[2], ... x[n-1], and then interchange x[1] with x[j]. Then x[0], x[1] are sorted.

Pass 3:
Leave the first two elements and find the location j of the smallest element in the sub-array x[2], x[3], .
. . . x[n-1], and then interchange x[2] with x[j]. Then x[0], x[1], x[2]
are sorted.

Pass (n-
1):
Find the location j of the smaller of the elements x[n-2] and x[n-1], and then interchange x[j] and x[n-
2]. Then x[0], x[1], ... x[n-2] are sorted. Of course, during this pass x[n-1] will be the biggest element and so the entire array is sorted.

Time
Complexity:

In general we prefer selection sort in case where the insertion sort or the bubble sort requires exclusive swapping. In spite of superiority of the selection sort over bubble sort and the insertion sort (there is significant decrease in run time), its efficiency is also $O(n^2)$ for n data items.

Example:

Let us consider the following example with 9 elements to analyze
selection Sort:

1	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*Remark*
65	70	75	80	50	60	55	85	45	find the first smallest element
I								j	swap a[i] & a[j]
45	70	75	80	50	60	55	85	65	find the second smallest
	I			j					swap a[i] and a[j]
45	50	75	80	70	60	55	85	65	Find the third smallest
		i				j			swap a[i] and a[j]

45	50	55	80	70	60	75	85	65	Find the fourth smallest
			I		j				swap a[i] and a[j]
45	50	55	60	70	80	75	85	65	Find the fifth smallest
				i				j	swap a[i] and a[j]
45	50	55	60	65	80	75	85	70	Find the sixth smallest
					i			j	swap a[i] and a[j]
45	50	55	60	65	70	75	85	80	Find the seventh smallest
						i j			swap a[i] and a[j]
45	50	55	60	65	70	75	85	80	Find the eighth smallest
							i	J	swap a[i] and a[j]
45	50	55	60	65	70	75	80	85	The outer loop ends.

Non-recursive Program for selection sort:

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

void selectionSort( int low, int high );

int a[25];

int main()
{
        int num,  i= 0;
        clrscr();
        printf( "Enter the number of elements: " );
        scanf("%d", &num);
        printf( "\nEnter the elements:\n" );
        for(i=0; i < num; i++)
                scanf( "%d", &a[i] );
        selectionSort( 0, num - 1 );
        printf( "\nThe elements after sorting are: " );
        for( i=0; i< num; i++ )
                printf( "%d    ", a[i] );
        return 0;
}

void selectionSort( int low, int high )
{
        int i=0, j=0, temp=0, minindex;
        for( i=low; i <= high; i++ )
        {
                minindex = i;
                for( j=i+1; j <= high; j++ )
                        if( a[j] < a[minindex] )
                                minindex = j;
                temp = a[i];
                a[i] = a[minindex];
                a[minindex] = temp;
        }
```

```
}
```

Recursive Program for selection sort:

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

int x[6] = {77, 33, 44, 11, 66};

selectionSort(int);
main()
{
int i, n = 0;
clrscr();
printf (" Array Elements before sorting: ");
for (i=0; i<5; i++)
        printf ("%d  ", x[i]);
selectionSort(n);                       /* call selection sort */
printf ("\n Array Elements after sorting: ");
for (i=0; i<5; i++)
        printf ("%d  ", x[i]);
selectionSort( int n)
{
        int k, p, temp, min;
        if (n== 4)
                return (-1);
        min = x[n];
        p = n;
        for (k = n+1; k<5; k++)
        {
                if (x[k] <min)
                {
                        min = x[k];
                        p = k;
                }
        }
        temp = x[n];            /* interchange x[n] and x[p] */
        x[n]   =
        x[p];
        x[p]   =
        temp;
        n++ ;
        selectionSort(n);
}
```

4.6 INSERTION SORT

The main idea behind the insertion sort is to insert the i^{th} element in its correct place in the i^{th} pass. Suppose an array A with n elements A[1], A[2],...A[N] is in memory. The insertion sort algorithm scans A from A[1] to A[N], inserting each element A[K] into its proper position in the previously sorted subarray A[1], A[2],..A[K-1].

Principle: In Insertion Sort algorithm, each element A[K] in the list is compared with all the elements before it (A[1] to A[K-1]). If any element A[I] is found to be greater than A[K] then A[K] is inserted in the place of A[I}. This process is repeated till all the elements are sorted.

Algorithm:

```
Procedure INSERTIONSORT(A, N)

// A is the array containing the list of data items
// N is the number of data items in the list

Last ⇓ N – 1

Repeat For Pass = 1 to Last Step 1
Repeat For I = 0 to Pass – 1 Step 1
If A[Pass] < A[I] Then
Temp ⇓ A[Pass]
Repeat For J = Pass -1 to I Step -1
A[J +1] ⇓ A[J] End Repeat
A[I] ⇓ Temp
End If
End Repeat
End Repeat

End

 INSERTIONSORT
```

In Insertion Sort algorithm, *Last* is made to point to the last element in the list and *Pass* is made to point to the second element in the list. In every pass the *Pass* is incremented to point to the next element and is continued till it reaches the last element. During each pass A[Pass] is compared all elements before it. If A[Pass] is lesser than A[I] in the list, then A[Pass] is inserted in position I. Finally, a sorted list is obtained.
For performing the insertion operation, a variable temp is used to safely store A[Pass] in it and then shift right elements starting from A[I] to A[Pass- 1].

Example:

N = 10 ◊ Number of elements in the list
L ◊ Last
P ◊ Pass

i = 0 i =1 i = 2 i = 3 i = 4 i = 5 i = 6 i = 7 i = 8 i = 9

42	23	74	11	65	58	94	36	99	87

P=1 *A[P] < A[0] ◇ Insert A[P] at 0* L=9

23	42	74	11	65	58	94	36	99	87

P=2 L=9

A[P] is greater than all elements before it. Hence No Change

23	42	74	11	65	58	94	36	99	87

P=3 *A[P] < A[0] ◇ Insert A[P] at 0* L=9

11	23	42	74	65	58	94	36	99	87

P=4 L=9

A[P] < A[3] ◊ Insert A[P] at 3

11	23	42	65	74	58	94	36	99	87

P=5 L=9

A[P] < A[3] ◊ Insert A[P] at 3

11	23	42	58	65	74	94	36	99	87

P=6 L=9

A[P] is greater than all elements before it. Hence No Change

11	23	42	58	65	74	94	36	99	87

P = 7 L = 9

A[P] < A[2] ◊ Insert A[P] at 2

11	23	36	42	58	65	74	94	99	87

P=8 L=9

A[P] is greater than all elements before it. Hence No Change

11	23	36	42	58	65	74	94	99	87

P, L=9

A[P] < A[7] ◇ Insert A[P] at 7

Sorted List:

11	23	36	42	58	65	74	87	94	99

Program:

```
void array::sort()
{
int temp, last=count-1;
for (int pass=1; pass<=last;pass++)
{
for (int i=0; i<pass; i++)
{
if (a[pass]<a[i])
{
temp=a[pass];
for (int j=pass-1;j>=i;j--)
a[j+1]=a[j];
a[i]=temp;
}
}
}
}
```

In the sort function, the integer variable *last* is used to point to the last element in the list. The first pass starts with the variable *pass* pointing to the second element and continues till *pass* reaches the last element. In each pass, a[pass] is compared with all the elements before it and if a[pass] is lesser than a[i], then it is inserted in position i. Before inserting it, the elements a[i] to a[pass-1] are shifted right using a temporary variable.

Advantages:

1. Sorts the list faster when the list has less number of elements.
2. Efficient in cases where a new element has to be inserted into a sorted list.

Disadvantages:

1. Very slow for large values of n.
2. Poor performance if the list is in almost reverse order.

4.7 QUICK SORT

The quick sort was invented by Prof. C. A. R. Hoare in the early 1960's. It was one of the first more efficient sorting algorithms. It is an example of a class of algorithms that work by what is usually called "divide and conquer".

In essence, the quick sort algorithm partitions the original array by rearranging it into two groups. The first group contains those elements less than some arbitrary chosen value taken from the set, and the second group contains those elements greater than or equal to the chosen value.

The chosen value is known as the *pivot element*. Once the array has been rearranged in this way with respect to the pivot, the very same partitioning is recursively applied to each of the two subsets. When all the subsets have been partitioned and rearranged, the original array is sorted.
The function partition() makes use of two pointers up and down which are moved toward each other in the following fashion:

1. Repeatedly increase the pointer up by one position until a[up] > =pivot.

2. Repeatedly decrease the pointer down by one position until a[down] <=pivot.

3. If down > up, interchange a[down] with a[up]

4. Repeat the steps 1, 2 and 3 till the 'up' pointer crosses the 'down' pointer. If 'up' pointer crosses 'down' pointer, the position for pivot is found and place pivot element in 'down' pointer position.

The program uses a recursive function quicksort(). The algorithm of quick sort function sorts all elements in an array 'a' between positions 'low' and 'high'.

1. It terminates when the condition low >= high is satisfied. This condition will be satisfied only when the array is completely sorted.

2. Here we choose the first element as the 'pivot'. So, pivot = x[low]. Now it calls the partition function to find the proper position j of the element x[low] i.e. pivot. Then we will have two sub-arrays x[low], x[low+1], ... x[j-1] and x[j+1], x[j+2], . . .x[high].

3. It calls itself recursively to sort the left sub-array x[low], x[low+1], ... x[j-1] between positions low and j-1 (where j is returned by the partition function).

4. It calls itself recursively to sort the right sub-array x[j+1], x[j+2], . . .x[high] between positions j+1 and high.

Time complexity:

There are several choices for choosing the ‘pivot’ element through which we can improve the efficiency of quick sort. For example, one may choose the ‘pivot’ element as median or mean or middle element. Also, a non-recursive method could be developed for execution efficiency. When these improvements are made, experiments indicate the fact that the total number of comparisons for quick sort is of O(n log n).

Example:

Select first element as the pivot element. Move ‘up’ pointer from left to right in search of an element larger than pivot. Move the ‘down’ pointer from right to left in search of an element smaller than pivot. If such elements are found, the elements are swapped. This process continues till the ‘up’ pointer crosses the ‘down’ pointer. If ‘up’ pointer crosses ‘down’ pointer, the position for pivot is found and interchange pivot and element at ‘down’ position.

Let us consider the following example with 13 elements to analyze quick sort:

1	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*Remarks*
38	08	16	06	79	57	24	56	02	58	04	70	45	
pivot				up						dow n			swap up & down
				04						79			
					Up			dow n					swap up & down
					02			57					
						dow n	up						
(24	08	16	06	04	02)	38	(56	57	58	79	70	45)	swap pivot &
pivot					dow n								swap pivot &
(02	08	16	06	04)	24								
pivot, down	up												swap pivot &

													down
02	(08	16	06	04)									
	Pivot	up		dow n									swap up & down
		04		16									
			dow n	Up									
	(06	04)	08										swap pivot &

	pivot, dow n	up											
	(04)	06											swap pivot &
	04 pivot, dow n												
				16 pivot , dow									
(02	04	06	08	16	24)	38							
							(56	57	58	79	70	45)	
							pivot	up				dow n	swap up & down
								45				57	
								dow n	up				
							(45)	56	(58	79	70	57)	swap pivot &
							45 pivot , dow n						swap pivot & down
									(58 pivot	79 up	70	57) dow n	swap up & down
										57		79	
										dow			
									(57)	58	(70	79)	swap pivot &
									57 pivot , dow				
											(70	79)	
											pivot , dow n	up	swap pivot &
											70		
												7 pivot , dow	

												n	
							(45	56	57	58	70	79)	

02	04	06	08	16	24	38	45	56	57	58	70	79	

Program for Quick Sort (Recursive version):

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

void quicksort(int, int);
int partition(int, int);
void interchange(int, int);

int array[25];

int main()
{
int num, i = 0;
clrscr();
printf( "Enter the number of elements: " );
scanf( "%d", &num);
printf( "Enter the elements: " );
for(i=0; i < num; i++)
scanf( "%d", &array[i] );
quicksort(0, num -1);
printf( "\nThe elements after sorting are: " );
for(i=0; i < num; i++)
printf("%d  ", array[i]);
return 0;
}
void quicksort(int low, int high)
{
int pivotpos;
if( low < high )
{
pivotpos = partition(low, high + 1); quicksort(low, pivotpos - 1); quicksort(pivotpos + 1, high);
}
}

int partition(int low, int high)
{
int pivot = array[low];
int up = low, down = high;

do
{
do
up = up + 1;
```

```
while(array[up] < pivot );

do
down = down - 1;
while(array[down] > pivot);

if(up < down)
interchange(up, down);

}while(up < down);
array[low] = array[down]; array[down] = pivot; return down;
}

void interchange(int i, int j)
{
int temp;
temp = array[i]; array[i] = array[j]; array[j] = temp;
}
```

4.8 MERGE SORT

Principle: The given list is divided into two roughly equal parts called the left and the right subfiles. These subfiles are sorted using the algorithm recursively and then the two subfiles are merged together to obtain the sorted file.

Given a sequence of n elements A[1], ...A[N], the general idea is to imagine them split into two sets A[1],...A[N/2] and A[(N/2) + 1],...A[N]. Each set is individually sorted, and the resulting sorted sequences are merged to produce a single sorted sequence of N elements. Thus this sorting method follows Divide and Conquer strategy.

Algorithm:

```
Procedure MERGE(A, low, mid, high)

// A is the array containing the list of data items

I ⇓ low, J ⇓ mid + 1, K ⇓ low
While I ≤ mid and J ≤ high
If A[I] < A[J] Then
        Temp[K] ⇓ A[I] I ⇓ I + 1
        K ⇓ K + 1

        Temp[K] ⇓ A[J] J ⇓ J + 1
        K ⇓ K + 1
Else
End If
End While

If I > mid
Then
    While J ≤ high
    Temp[K] ⇓ A[J] K ⇓ K + 1
    J ⇓ J + 1
    End While

    While I ≤ mid
    Temp[K] ⇓ A[I] K ⇓ K + 1
Else I ⇓ I + 1
    End While
```

End If

Repeat for K = low to high step 1
A[K] ⇓ Temp[K] End Repeat
End MERGE

Procedure MERGESORT(A, low, high)

// A is the array containing the list of data items

If low < high
Then
mid ⇓ (low + high)/2
MERGESORT(low, high)
MERGESORT(mid + 1, high) MERGE(low, mid, high)
End If
End MERGESORT

The first algorithm MERGE can be applied on two sorted lists to merge t hem. Initially, the index variable I points to low and J points to mid + 1. A[I] is compared with A[J] and if A[I] found to be lesser than A[J] then A[I] is stored in a temporary array and I is incremented otherwise A[J] is stored in the temporary array and J is incremented. This comparis on is continued until either I crosses mid or J crosses high. If I crosses the mid first then that implies that all the elements in first list is accommodated in the temporary array and hence the remaining elements in the second list can be put into the temporary array as it is. If J crosses the high first th en the remaining elements of first list is put as it is in the temporary array. After this process we get a single sorted list. Since this method merges 2 lists at a time, this is called 2-way merge sort.

In the MERGESORT algorithm, the given unsorted list is first split into N n umber of lists, each list consisting of only 1 element. Then the MERGE al gorithm is applied for first 2 lists to get a single sorted list. Then the sam e thing is done on the next two lists and so on. This process is continued till a single sorted list is obtained.

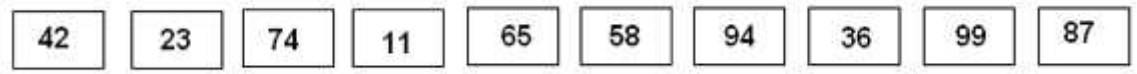
Example:

Le t L ◊ low , M ◊ mi d , H ◊ hi g h

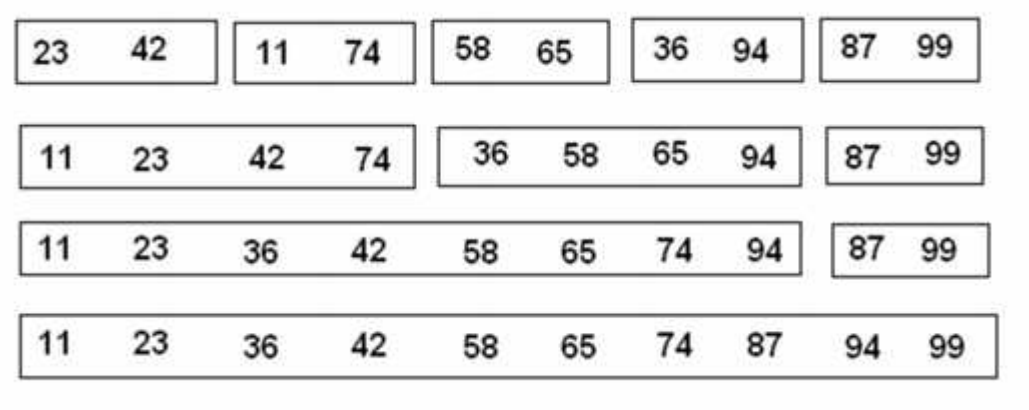
i = 0 i =1 i = 2 i = 3 i = 4 i = 5 i = 6 i = 7 i = 8 i = 9

42	23	74	11	65	58	94	36	99	87
U				M					H

In each pass the mid value is calculated and based on that the list is split into two. This is done recursively and at last N number of lists each having only one element is produced as shown.



Now merging operation is called on first two lists to produce a single sorted list, then the same thing is done on the next two lists and so on. Finally a single sorted list is obtained.



Program:

```
void array::sort(int low, int high)
{
      int mid;
      if (low<high)
      {
            mid=(low+high)/2;
            sort(low,mid);
            sort(mid+1, high);
            merge(low, mid, high);
      }
}

void array::merge(int low, int mid, int high)
{
int i=low, j=mid+1, k=low, temp[MAX];

while (i<=mid && j<=high)
      if (a[i]<a[j])
            temp[k++]=a[i++];
      else
            temp[k++]=a[j++];

if (i>mid)
      while (j<=high)
            temp[k++]=a[j++];
else
      while (i<=mid)
            temp[k++]=a[i++];
```

```
for (k=low; k<=high; k++)
      a[k]=temp[k];
}
```

Advantages:

1. Very useful for sorting bigger lists.
2. Applicable for external sorting also.

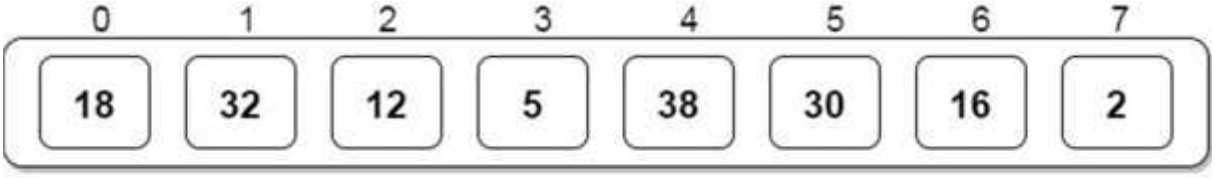
Disadvantages:

1. Needs a temporary array every time, for storing the new sorted list.

4.9 SHELL SORT

The shell sort, sometimes called the "diminishing increment sort," improves on the insertion sort by breaking the original list into a number of smaller sublists, each of which is sorted using an insertion sort. The unique way that these sublists are chosen is the key to the shell sort. Instead of breaking the list into sublists of contiguous items, the shell sort uses an increment i, sometimes called the gap, to create a sublist by choosing all items that are i items apart.

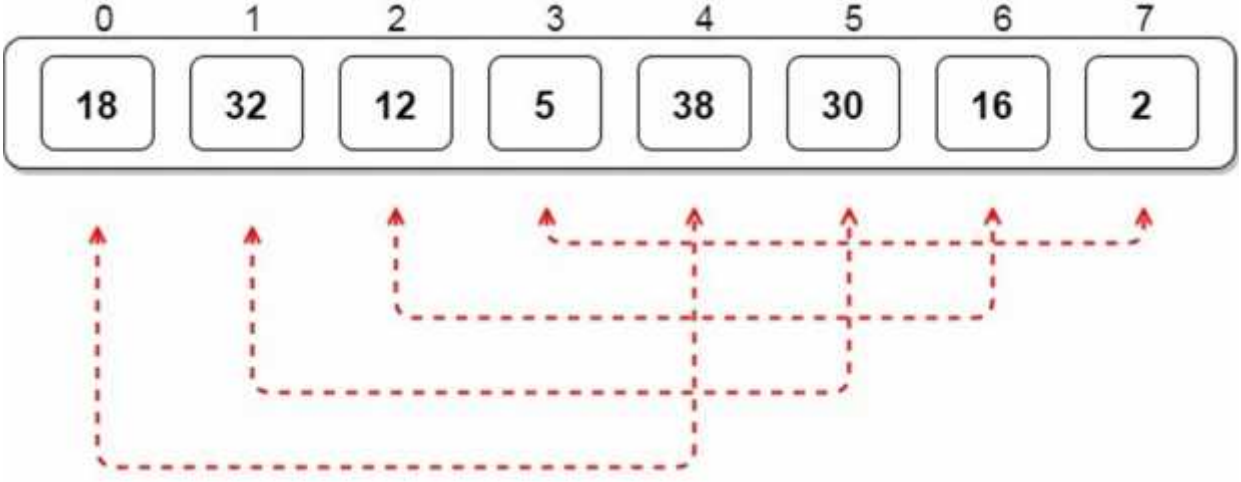
Example of shell Sort : Use Shell sort for the following array : 18, 32, 12, 5, 38, 30, 16, 2



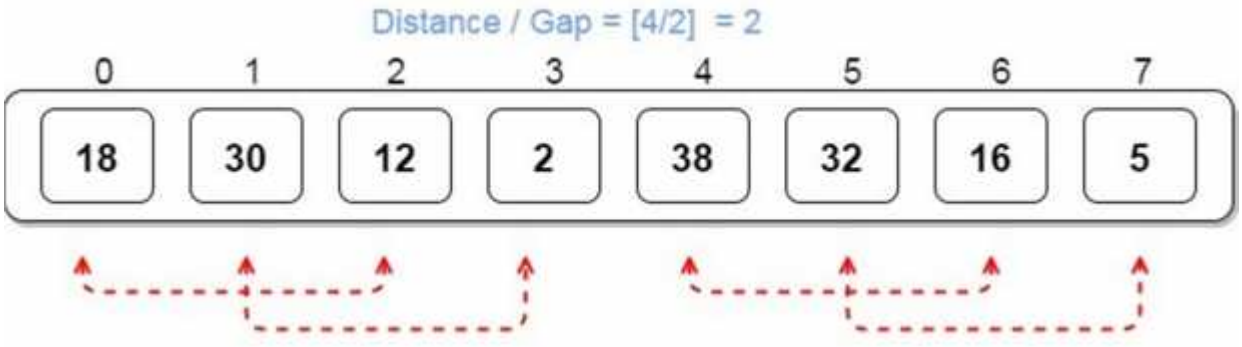
Distance / Gap = [n/2] (floor value)

= 8/2 => 4

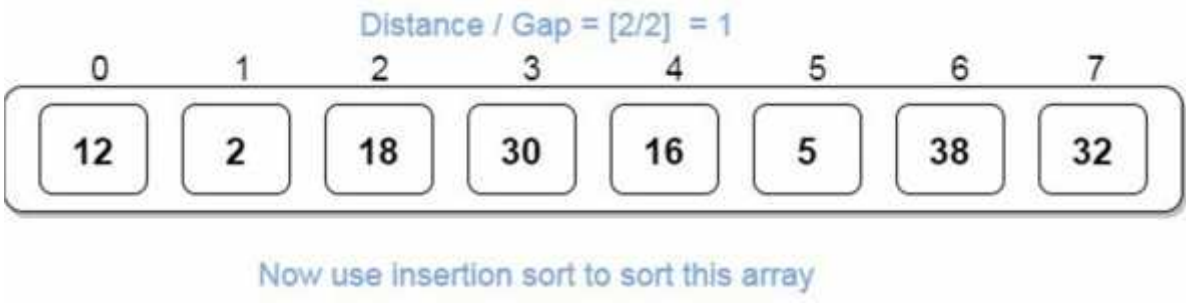
Compare the elements at a gap of 4. i.e 18 with 38 and so on and swap if first number is greater than second.



Compare the elements at a gap of 2 i.e 18 with 12 and so on.



Now the gap is 1. So now use insertion sort to sort this array.



After insertion sort. The final array is sorted.

4.10 HEAP SORT ALGORITHM

Heap Sort is a popular and efficient sorting algorithm in computer programming. Learning how to write the heap sort algorithm requires knowledge of two types of data structures - arrays and trees.

The initial set of numbers that we want to sort is stored in an array e.g. [10, 3, 76, 34, 23, 32] and after sorting, we get a sorted array [3,10,23,32,34,76]

Heap sort works by visualizing the elements of the array as a special kind of complete binary tree called a heap.

As a prerequisite, you must know about a complete binary tree and heap data structure.

Relationship between Array Indexes and Tree Elements

A complete binary tree has an interesting property that we can use to find the children and parents of any node.

If the index of any element in the array is i, the element in the index 2i+1 will become the left child and element in 2i+2 index will become the right child. Also, the parent of any element at index i is given by the lower bound of (i-1)/2.

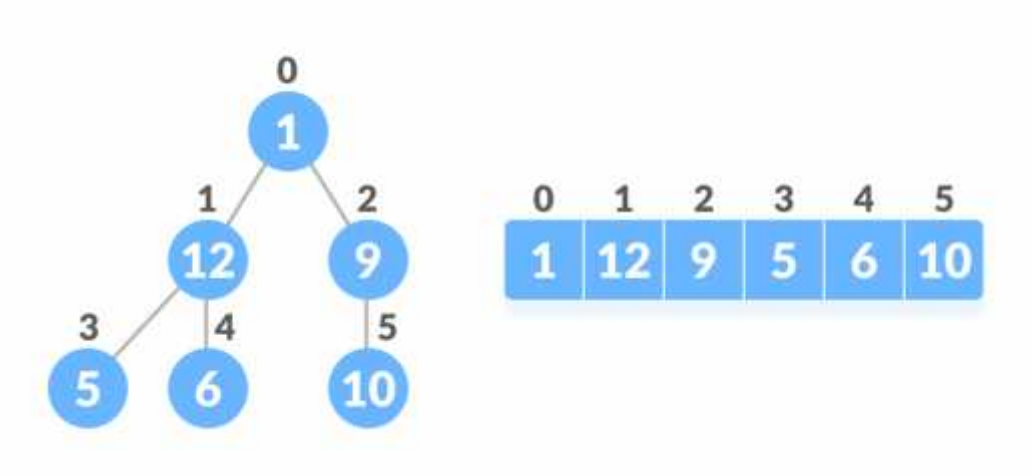


Fig.4.10.1 Relationship between array and heap indices

Let's test it out,
Left child of 1 (index 0)
= element in (2*0+1) index
= element in 1 index
= 12

Right child of 1
= element in (2*0+2) index
= element in 2 index
= 9

Similarly,
Left child of 12 (index 1)
= element in (2*1+1) index
= element in 3 index
= 5

Right child of 12
= element in (2*1+2) index
= element in 4 index
= 6
Let us also confirm that the rules hold for finding parent of any node
Parent of 9 (position 2)
= (2-1)/2
= ½
= 0.5
~ 0 index
= 1

Parent of 12 (position 1)

= (1-1)/2
= 0 index
= 1
Understanding this mapping of array indexes to tree positions is critical to understanding how the Heap Data Structure works and how it is used to implement Heap Sort.
What is Heap Data Structure?
Heap is a special tree-based data structure. A binary tree is said to follow a heap data structure if it is a complete binary tree
All nodes in the tree follow the property that they are greater than their children i.e. the largest element is at the root and both its children and smaller than the root and so on. Such a heap is called a max-heap. If instead, all nodes are smaller than their children, it is called a min-heap
The following example diagram shows Max-Heap and Min-Heap.

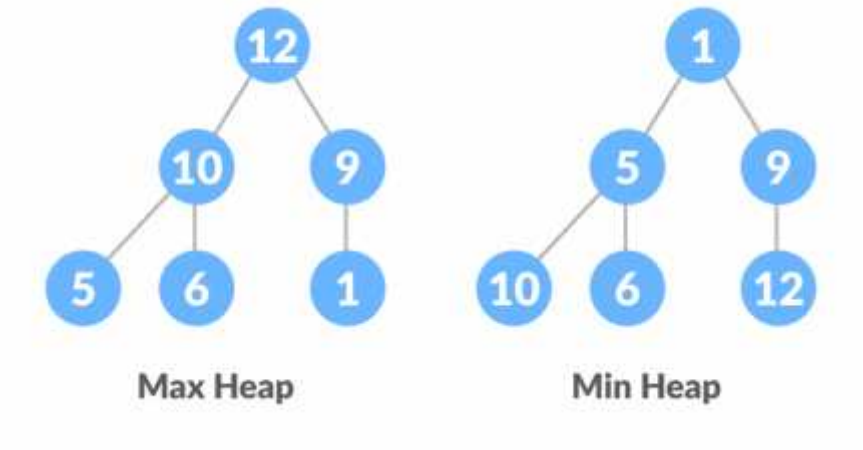


Fig.4.10.2 Max Heap and Min Heap

To learn more about it, please visit Heap Data Structure.
How to "heapify" a tree
Starting from a complete binary tree, we can modify it to become a Max-Heap by running a function called heapify on all the non-leaf elements of the heap.
Since heapify uses recursion, it can be difficult to grasp. So let's first think about how you would heapify a tree with just three elements.

```
heapify(array)
   Root = array[0]
   Largest = largest( array[0] , array [2*0 + 1]. array[2*0+2])
   if(Root != Largest)
        Swap(Root, Largest)
```

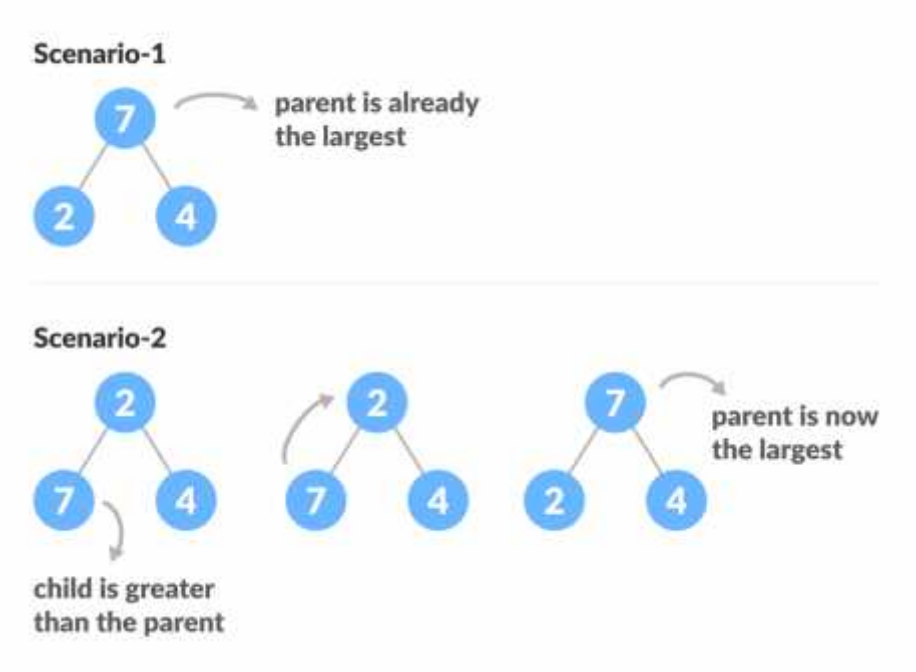


Fig.4.10.3 Heapify base cases

The example above shows two scenarios - one in which the root is the largest element and we don't need to do anything. And another in which the root had a larger element as a child and we needed to swap to maintain max-heap property.

If you're worked with recursive algorithms before, you've probably identified that this must be the base case.

Now let's think of another scenario in which there is more than one level.

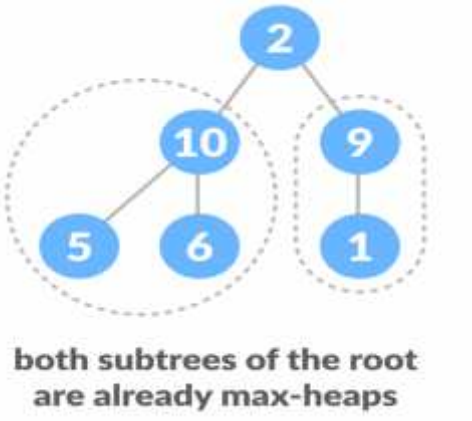


Fig.4.10.4 How to heapify root element when its subtrees are already max heaps

The top element isn't a max-heap but all the sub-trees are max-heaps.

To maintain the max-heap property for the entire tree, we will have to keep pushing 2 downwards until it reaches its correct position.

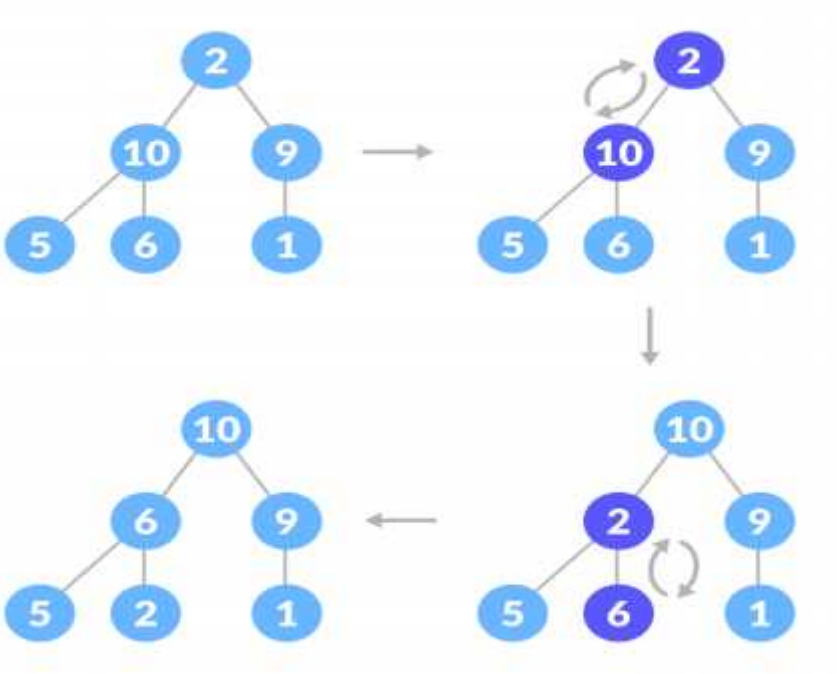


Fig.4.10.5 How to heapify root element when its subtrees are max-heaps

Thus, to maintain the max-heap property in a tree where both sub-trees are max-heaps, we need to run heapify on the root element repeatedly until it is larger than its children or it becomes a leaf node.

We can combine both these conditions in one heapify function as

```
void heapify(int arr[], int n, int i) {
  // Find largest among root, left child and right child
  int largest = i;
  int left = 2 * i + 1;
  int right = 2 * i + 2;

  if (left < n && arr[left] > arr[largest])
    largest = left;

  if (right < n && arr[right] > arr[largest])
    largest = right;

    // Swap and continue heapifying if root is not largest
    if (largest != i) {
      swap(&arr[i], &arr[largest]);
      heapify(arr, n, largest);
  }
}
```

This function works for both the base case and for a tree of any size. We can thus move the root element to the correct position to maintain the max-heap status for any tree size as long as the sub-trees are max-heaps.

Build max-heap

To build a max-heap from any tree, we can thus start heapifying each sub-tree from the bottom up and end up with a max-heap after the function is applied to all the elements including the root element.

In the case of a complete tree, the first index of a non-leaf node is given by n/2 - 1. All other nodes after that are leaf-nodes and thus don't need to be heapified.

So, we can build a maximum heap as

```
// Build heap (rearrange array)
for (int i = n / 2 - 1; i >= 0; i--)
  heapify(arr, n, i);
```

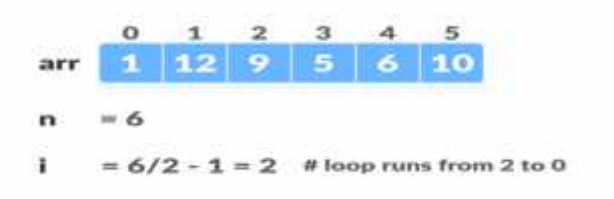


Fig.4.10.6 Create array and calculate i

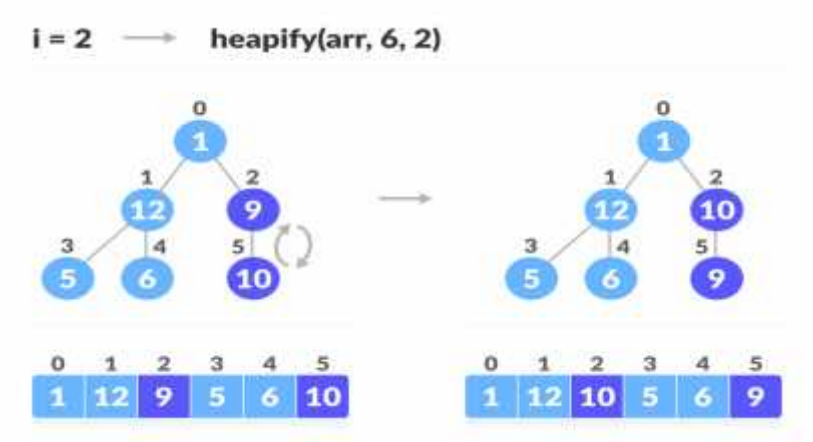


Fig.4.10.7 Steps to build max heap for heap sort

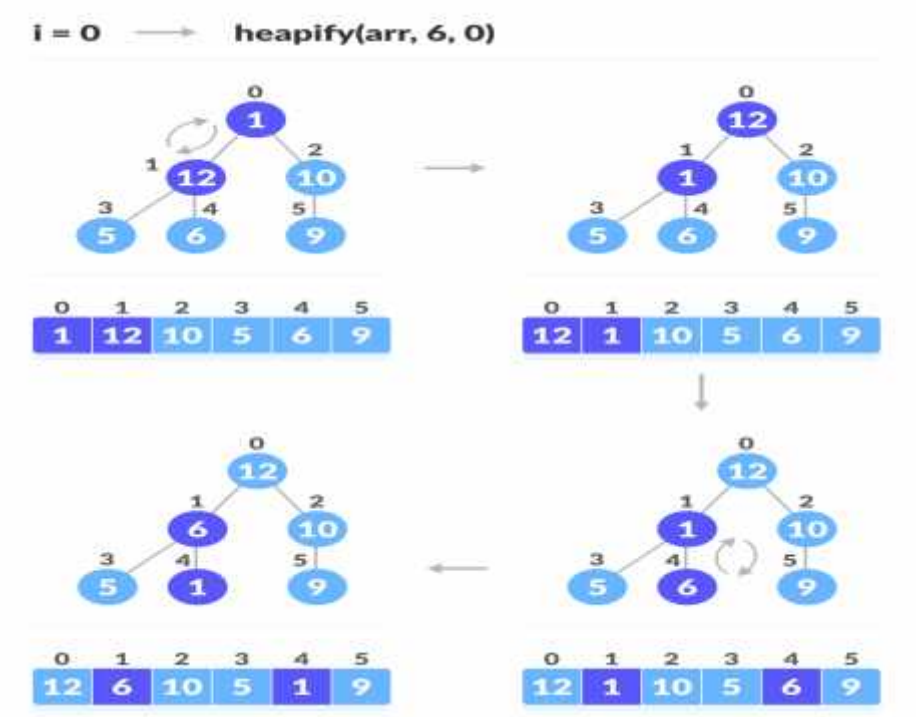


Fig.4.10.8 Steps to build max heap for heap sort

As shown in the above diagram, we start by heapifying the lowest smallest trees and gradually move up until we reach the root element.
If you've understood everything till here, congratulations, you are on your way to mastering the Heap sort.

How Heap Sort Works?
Since the tree satisfies Max-Heap property, then the largest item is stored at the root node.
Swap: Remove the root element and put at the end of the array (nth position) Put the last item of the tree (heap) at the vacant place.
Remove: Reduce the size of the heap by 1.
Heapify: Heapify the root element again so that we have the highest element at root.
The process is repeated until all the items of the list are sorted.

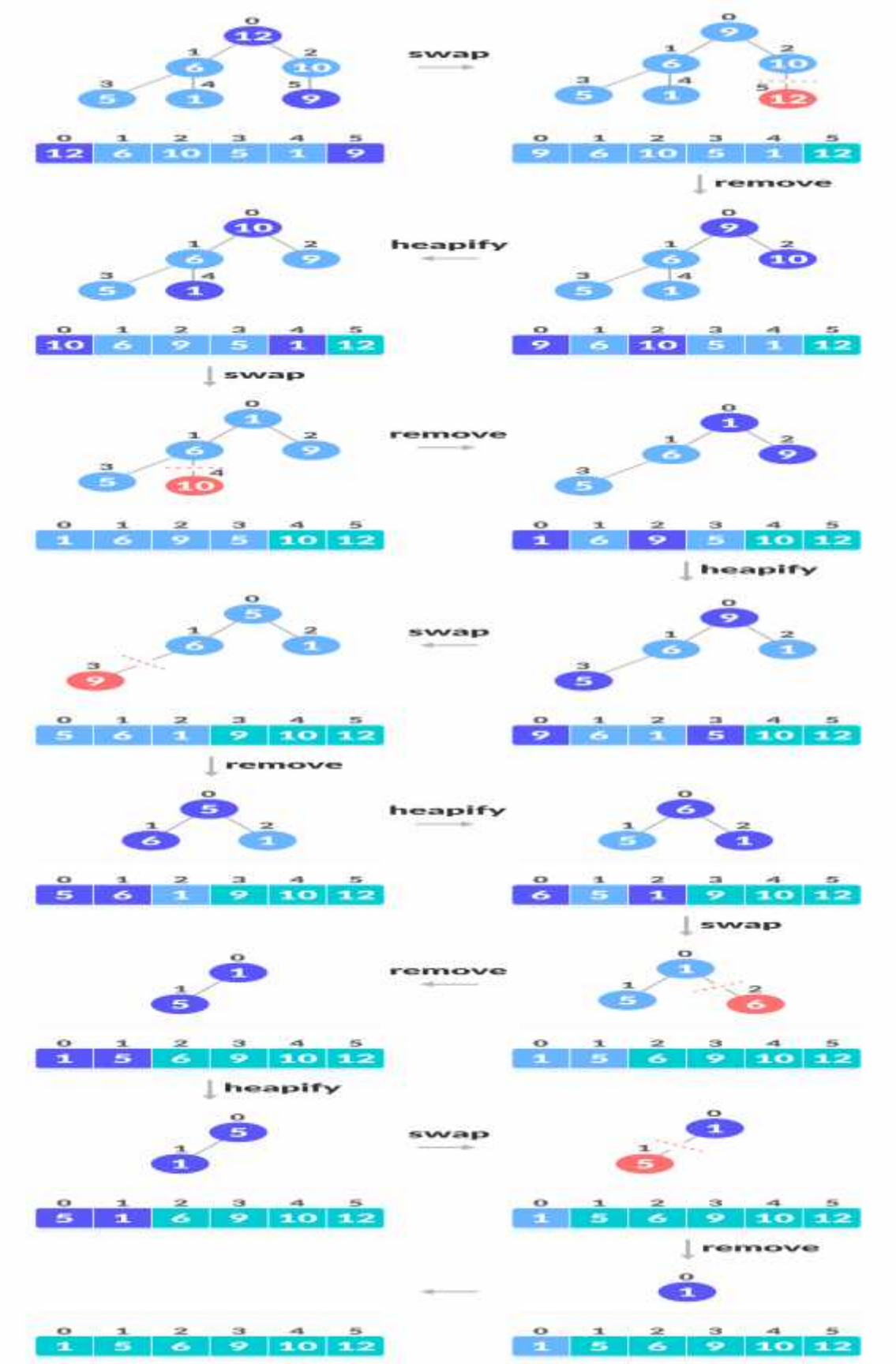


Fig.4.10.9Swap, Remove, and Heapify

The code below shows the operation.

```
  // Heap sort
  for (int i = n - 1; i >= 0; i--) {
    swap(&arr[0], &arr[i]);

    // Heapify root element to get highest element at root again
    heapify(arr, i, 0);
  }C++
```

```
def heapify(arr, n, i):
    # Find largest among root and children
    largest = i
    l = 2 * i + 1
    r = 2 * i + 2

    if l < n and arr[i] < arr[l]:
        largest = l

    if r < n and arr[largest] < arr[r]:
        largest = r

    # If root is not largest, swap with largest and continue heapifying
    if largest != i:
        arr[i], arr[largest] = arr[largest], arr[i]
        heapify(arr, n, largest)


def heapSort(arr):
    n = len(arr)

    # Build max heap
    for i in range(n//2, -1, -1):
        heapify(arr, n, i)

    for i in range(n-1, 0, -1):
        # Swap
        arr[i], arr[0] = arr[0], arr[i]

        # Heapify root element
        heapify(arr, i, 0)
```

```
arr = [1, 12, 9, 5, 6, 10]
heapSort(arr)
n = len(arr)
print("Sorted array is")
for i in range(n):
    print("%d " % arr[i], end='')
```

Heap Sort Complexity

Heap Sort has O(nlog n) time complexities for all the cases (best case, average case, and worst case).

Let us understand the reason why. The height of a complete binary tree containing n elements is log n

As we have seen earlier, to fully heapify an element whose subtrees are already max-heaps, we need to keep comparing the element with its left and right children and pushing it downwards until it reaches a point where both its children are smaller than it.

In the worst case scenario, we will need to move an element from the root to the leaf node making a multiple of log(n) comparisons and swaps.

During the build_max_heap stage, we do that for n/2 elements so the worst case complexity of the build_heap step is n/2*log n ~ nlog n.

During the sorting step, we exchange the root element with the last element and heapify the root element. For each element, this again takes log n worst time because we might have to bring the element all the way from the root to the leaf. Since we repeat this n times, the heap_sort step is also nlog n.

Also since the build_max_heap and heap_sort steps are executed one after another, the algorithmic complexity is not multiplied and it remains in the order of nlog n.

Also it performs sorting in O(1) space complexity. Compared with Quick Sort, it has a better worst case (O(nlog n)). Quick Sort has complexity O(n^2) for worst case. But in other cases,

Quick Sort is fast. Introsort is an alternative to heapsort that combines quicksort and heapsort to retain advantages of both: worst case speed of heapsort and average speed of quicksort.

Heap Sort Applications

Systems concerned with security and embedded systems such as Linux Kernel use Heap Sort because of the O(n log n) upper bound on Heapsort's running time and constant O(1) upper bound on its auxiliary storage.

Although Heap Sort has O(n log n) time complexity even for the worst case, it doesn't have more applications (compared to other sorting algorithms like Quick Sort, Merge Sort). However, its underlying data structure, heap, can be efficiently used if we want to extract the smallest (or largest) from the list of items without the overhead of keeping the remaining items in the sorted order. For e.g Priority Queues.

School of Computing

Department of Computer Science and Engineering

UNIT - V

Fundamental of Data Structure – SBS1201

UNIT-V

Graphs and Networks: Implementation of Graphs - Adjacency Matrix- Depth First Search- Breath First Search. Networks: Minimum Spanning Tree - The Shortest path Algorithm.

5.1 GRAPH, HASHING & INDEXING

A graph is an abstract data structure that is used to implement the mathematical concept of graphs. It is basically a collection of vertices (also called nodes) and edges that connect these vertices. A graph is often viewed as a generalization of the tree structure, where instead of having a purely parent-to- child relationship between tree nodes, any kind of complex relationship can exist.

Graphs - Terminology and Representation

Definitions: Graph, Vertices, Edges

- Define a graph G = (V, E) by defining a pair of sets:
  V = a set of vertices
  E = a set of edges
- Edges:
  - Each edge is defined by a pair of vertices
  - An edge connects the vertices that define it
- Vertices:
  - Vertices also called nodes
  - Denote vertices with labels

Representation:

- Represent vertices with circles, perhaps containing a label
- Represent edges with lines between circles

Example:

- V = {A,B,C,D}
- E = {(A,B),(A,C),(A,D),(B,D),(C,D)}

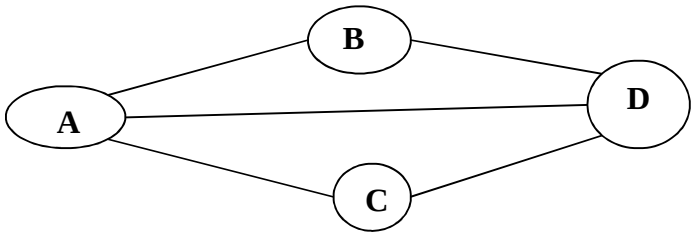


Fig. 5.1.1 Many algorithms use a graph representation to represent data or the problem to be solved

Examples of Graph applications:

- Cities with distances between
- Roads with distances between intersection points
- Course prerequisites
- Network and shortest routes
- Social networks
- Electric circuits, projects planning and many more...

Graph Classifications

- There are several common kinds of graphs
  - Weighted or unweighted
  - Directed or undirected
  - Cyclic or acyclic
  - Multigraphs

Kinds of Graphs: Weighted and Unweighted

- Graphs can be classified by whether or not their edges have weights
- **Weighted graph:** edges have a weight
  Weight typically shows cost of traversing
  Example: weights are distances between cities
- **Unweighted graph:** edges have no weight of Edges simply show connections
  Example: course prerequisites

Kinds of Graphs: Directed and Undirected

Graphs can be classified by whether or their edges are have direction

o Undirected Graphs: each edge can be traversed in either direction
o Directed Graphs: each edge can be traversed only in a specified direction

Undirected Graphs

- Undirected Graph: no implied direction on edge between nodes
- The example from above is an undirected graph

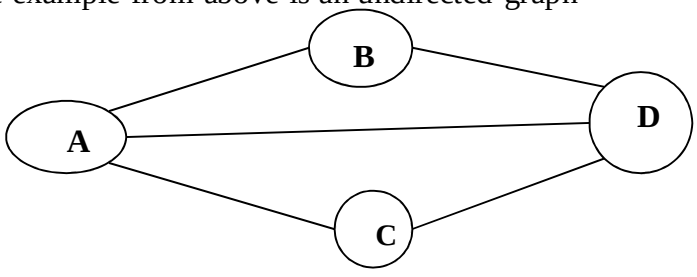


Fig.5.1.2 .In diagrams, edges have no direction (ie there are no arrows) Can traverse edges in either directions

In an undirected graph, an edge is an unordered pair

o Actually, an edge is a set of 2 nodes, but for simplicity we write it with parenthesis

For example, we write (A, B) instead of {A, B}

Thus, (A,B) = (B,A), etc

If (A,B) ∈ E then (B,A) ∈ E

Directed Graphs

Digraph: A graph whose edges are directed (ie have a direction)

- Edge drawn as arrow
- Edge can only be traversed in direction of arrow
- Example: E = {(A,B), (A,C), (A,D), (B,C), (D,C)}

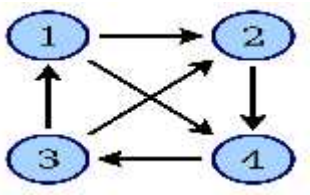


Fig.5.1.3 In a digraph, an edge is an ordered pair

o Thus: (u,v) and (v,u) are not the same edge

o In the example, (D,C) ∈ E, (C,D) ∉ E

Degree of a Node

- The degree of a node is the number of edges incident on it.
- In the example above:
  - Degree 2: B and C
  - Degree 3: A and D
  - A and D have odd degree, and B and C have even degree
- Can also define in-degree and out-degree
  - In-degree: Number of edges pointing to a node
  - Out-degree: Number of edges pointing from a node

Graphs: Terminology Involving Paths

- Path: sequence of vertices in which each pair of successive vertices is connected by an edge
- Cycle: a path that starts and ends on the same vertex
- Simple path: a path that does not cross itself
- That is, no vertex is repeated (except first and last)
- Simple paths cannot contain cycles

- Length of a path: Number of edges in the path
- Examples

Cyclic and Acyclic Graphs

ACyclic graph contains cycles
- Example: roads (normally)

An acyclic graph contains no cycles
- Example: Course prerequisites

Multigraph: A graph with self loops and parallel edges is called a multigraph.

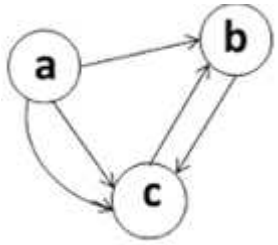


Fig.5.1.4 Connected and Unconnected Graphs and Connected Components

An undirected graph is connected if every pair of vertices has a path between it
- Otherwise it is unconnected
- A directed graph is strongly connected if every pair of vertices has a path between them, in both directions

Data Structures for Representing Graphs

Two common data structures for representing graphs:

- Adjacency lists
- Adjacency matrix

5.2 Adjacency List Representation

An adjacency list is a way in which graphs can be represented in the computer's memory. This structure consists of a list of all nodes in G. Furthermore, every node is in turn linked to its own list that contains the names of all other nodes that are adjacent to it. The key advantages of using an adjacency list are:

- It is easy to follow and clearly shows the adjacent nodes of a particular node.
- It is often used for storing graphs that have a small-to-moderate number of edges. That is, an adjacency list is preferred for representing sparse graphs in the computer's memory; otherwise, an adjacency matrix is a good choice.
- Adding new nodes in G is easy and straightforward when G is represented using an

adjacenc y list. Adding new nodes in an adjacency matrix is a difficult task, as the size of the matrix needs to be changed and existing nodes may have to be reordered. Each node has a list of **adjacent nodes.**

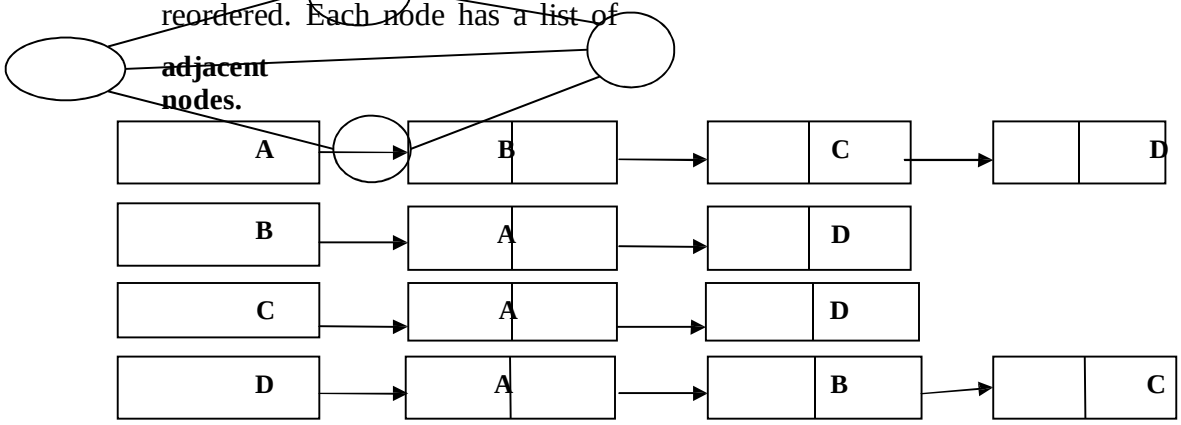


Fig.5.2.1 Adjaceny List Representation

Adjacency Matrix Representation

An adjacency matrix is used to represent which nodes are adjacent to one another. By definition, two nodes are said to be adjacent if there is an edge connecting them. In a directed graph G, if node v is adjacent to node u, then there is definitely an edge from u to v. That is, if v is adjacent to u, we can get from u to v by traversing one edge. For any graph G having n nodes, the adjacency matrix will have the dimension of n * n. In an adjacency matrix, the rows and columns are labelled by graph vertices. An entry aij in the adjacency matrix will contain 1, if vertices vi and vj are adjacent to each other. However, if the nodes are not adjacent, aij will be set to zero. It. Since an adjacency matrix contains only 0s and 1s, it is called a bit matrix or a Boolean matrix. The entries in the matrix depend on the ordering of the nodes in G. Therefore, a change in the order of nodes will result in a different adjacency matrix.

Aij = 1 if there is an edge from Vi to Vj

0 otherwise

Adjacency Matrix: 2D array containing weights on edges

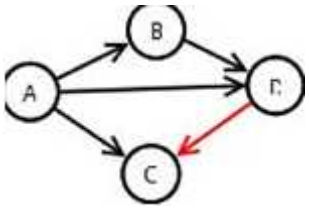
- Row for each vertex
- Column for each vertex
- Entries contain weight of edge from row vertex to column vertex
- Entries contain ∞ if no edge from row vertex to column vertex
- Entries contain 0 on diagonal (if self edges not allowed)

Example undirected graph (assume self-edges not allowed):

	A	B	C	D
A	0	1	1	1
B	1	0	∞	1
C	1	∞	0	1
D	1	1	1	0

Fig.5.2.2 Undirected graph

Example directed graph (assume self-edges allowed):



	A	B	C	D
A	∞	1	1	1
B	∞	∞	∞	1
C	∞	∞	∞	∞
D	∞	∞	1	∞

Fig.5.2.3 Directed Graph

Disadv: Adjacency matrix representation is easy to represent and feasible as long as the graph is small and connected. For a large graph ,whose matrix is sparse, adjacency matrix representation wastes a lot of memory. Hence list representation is preferred over matrix representation.

5.3 Graph traversal algorithms

Traversing a graph, is the method of examining the nodes and edges of the graph. There are two standard methods of graph traversal. These two methods are:

1. Breadth-first search 2. Depth-first search

While breadth-first search uses a queue as an auxiliary data structure to store nodes for further processing, the depth-first search scheme uses a stack.

Breadth-first search algorithm

Breadth-first search (BFS) is a graph search algorithm that begins at the root node and explores all the neighbouring nodes. Then for each of those nearest nodes, the algorithm explores their unexplored neighbour nodes, and so on, until it finds the goal. That is, we start examining the node A and then all the neighbours of A are examined. In the next step, we examine the neighbours of neighbours of A, so on and so forth. This means that we need to track the neighbours of the node and guarantee that every node in the graph is processed and no node is processed more

than once. This is accomplished by using a queue that will hold the nodes that are waiting for further processing.

Algorithm for BFS traversal

Step 1: Define a Queue of size total number of vertices in the graph.

Step 2: Select any vertex as starting point for traversal. Visit that vertex and insert it into the Queue.

Step 3: Visit all the adjacent vertices of the vertex which is at front of the Queue which is not visited and insert them into the Queue.

Step 4: When there is no new vertex to be visit from the vertex at front of the Queue then delete that vertex from the Queue.

Step 5: Repeat step 3 and 4 until queue becomes empty.

Step 6: When queue becomes Empty, then the enqueue or dequeue order gives the BFS traversal order.

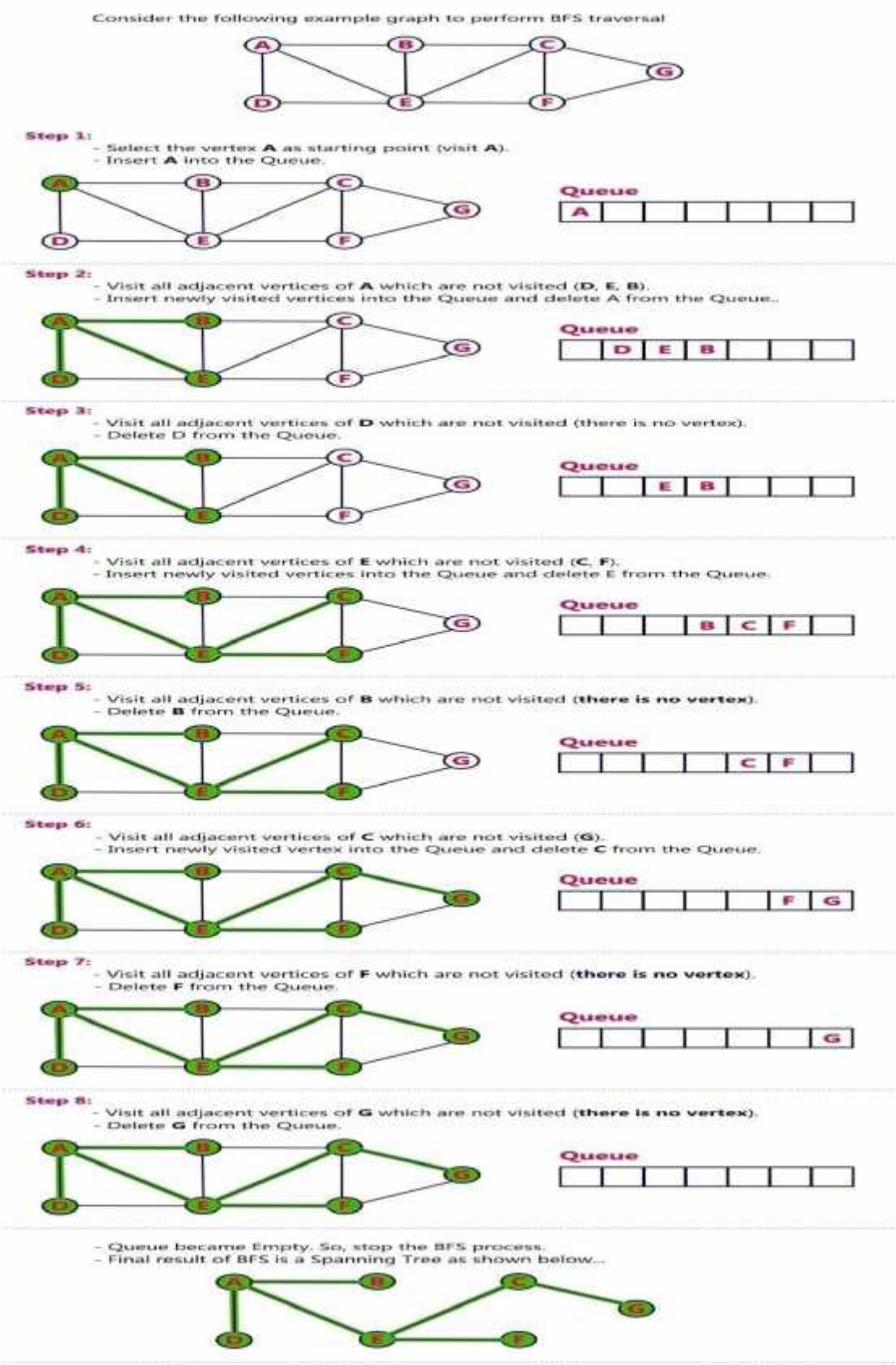


Fig.5.3.1 Example for BFS

Depth-first Search Algorithm

Depth-first search begins at a starting node A which becomes the current node. Then, it examines each node N along a path P which begins at A. That is, we process a neighbour of A, then a neighbour of neighbour of A, and so on.

During the execution of the algorithm, if we reach a path that has a node N that has already been processed, then we backtrack to the current node. Otherwise, the unvisited (unprocessed) node becomes the current node. The algorithm proceeds like this until we reach a dead-end (end of path P). On reaching the deadend, we backtrack to find another path P. The algorithm terminates when backtracking leads back to the starting node A.

In this algorithm, edges that lead to a new vertex are called discovery edges and edges that lead to an already visited vertex are called back edges. Observe that this algorithm is similar to the in- order traversal of a binary tree. Its implementation is similar to that of the breadth-first search algorithm but here we use a stack instead of a queue.

We use the following steps to implement DFS traversal...

Step 1: Define a Stack of size total number of vertices in the graph.

Step 2: Select any vertex as starting point for traversal. Visit that vertex and push it on to the Stack.

Step 3: Visit any one of the adjacent vertex of the verex which is at top of the stack which is not visited and push it on to the stack.

Step 4: Repeat step 3 until there are no new vertex to be visit from the vertex on top of the

stack. Step 5: When there is no new vertex to be visit then use back tracking and pop

one vertex from
the stack.

Step 6: Repeat steps 3, 4 and 5 until stack becomes Empty.

Step 7: When stack becomes Empty, then produce final spanning tree by removing unused edges from the graph

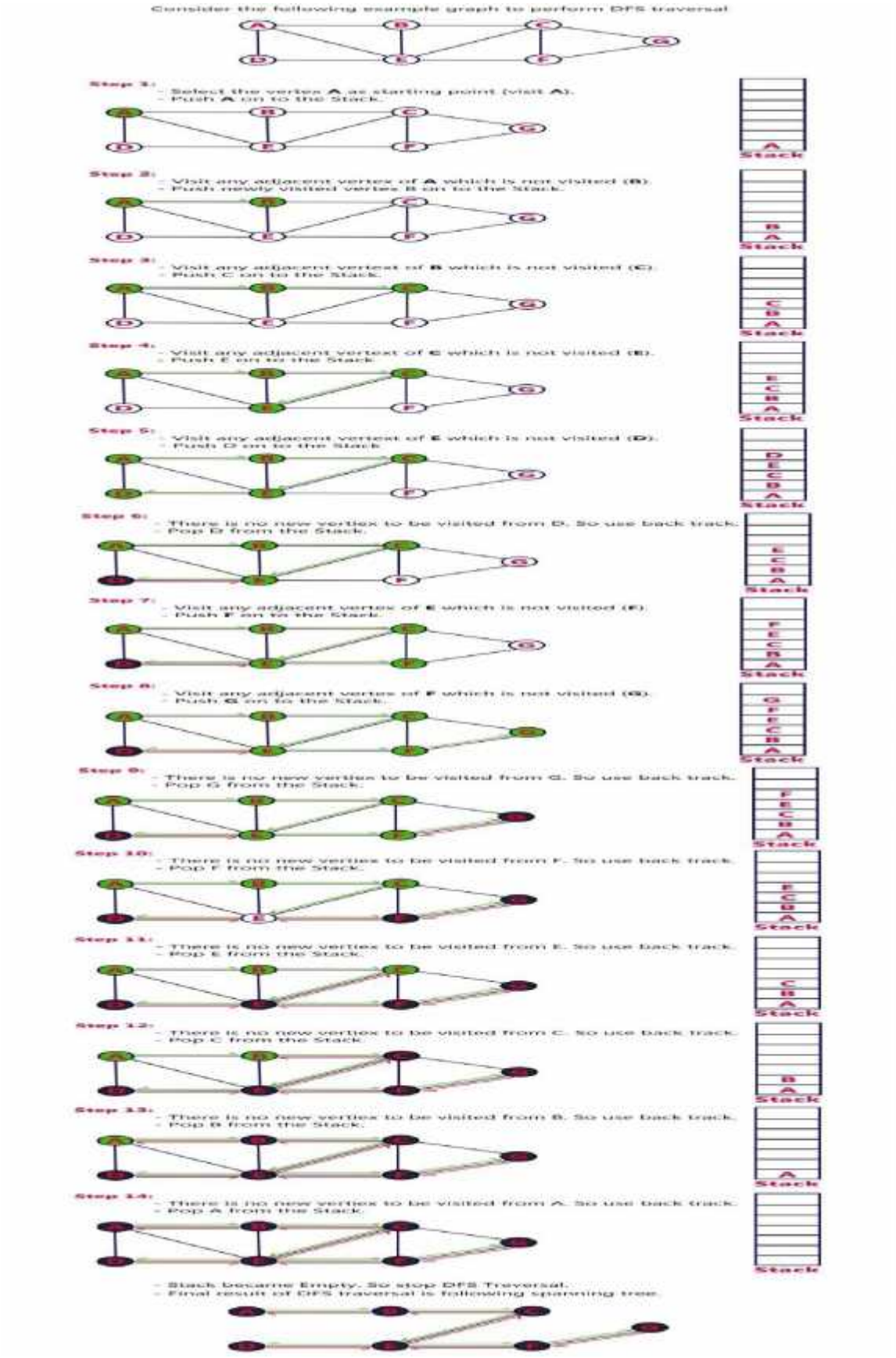


Fig.5.3.2 Example for DFS

Applications OF graphs

- Graphs are constructed for various types of applications such as:
- In circuit networks where points of connection are drawn as vertices and component wires become the edges of the graph.
- In transport networks where stations are drawn as vertices and routes become the edges of the graph.
- In maps that draw cities/states/regions as vertices and adjacency relations as edges.
- In program flow analysis where procedures or modules are treated as vertices and calls to these procedures are drawn as edges of the graph.
- Once we have a graph of a particular concept, they can be easily used for finding shortest paths, project planning, etc.
- In flowcharts or control-flow graphs, the statements and conditions in a program are
    - represented as nodes and the flow of control is represented by the edges.
- In state transition diagrams, the nodes are used to represent states and the edges represent legal moves from one state to the other.
- Graphs are also used to draw activity network diagrams. These diagrams are extensively used as a project management tool to represent the interdependent relationships between groups, steps, and tasks that have a significant impact on the project.

5.4 Hashing

Why Hashing?

Internet has grown to millions of users generating terabytes of content every day. According to internet data tracking services, the amount of content on the internet doubles every six months. With this kind of growth, it is impossible to find anything in the internet, unless we develop new data structures and algorithms for storing and accessing data. So what is wrong with traditional data structures like Arrays and Linked Lists? Suppose we have a very large data set stored in an array. The amount of time required to look up an element in the array is either O(log n) or O(n) based on whether the array is sorted or not. If the array is sorted then a technique such as binary search can be used to search the array. Otherwise, the array must be searched linearly. Either case may not be desirable if we need to process a very large data set. Therefore we discuss a new technique called hashing that allows us to update and retrieve any entry in constant time O(1). The constant time or O(1) performance means, the amount of time to perform the operation does not depend on data size n.

The Map Data Structure(Hash Map)(Hash function)

In a mathematical sense, a map is a relation between two sets. We can define Map M as a set of pairs, where each pair is of the form (key, value), where for given a key, we can find a value using some kind of a "function" that maps keys to values. The key for a given object can be calculated using a function called a hash function. In its simplest form, we can think of an array as a Map where key is the index and value is the value at that index. For example, given an array A, if i is the key, then we can find the value by simply looking up A[i]. The idea of a hash table is more generalized and can be described as follows.

The concept of a hash table is a generalized idea of an array where key does not have to be an integer. We can have a name as a key, or for that matter any object as the key. The trick is to find a hash function to compute an index so that an object can be stored at a specific location in a table such that it can easily be found.

STATIC HASHING

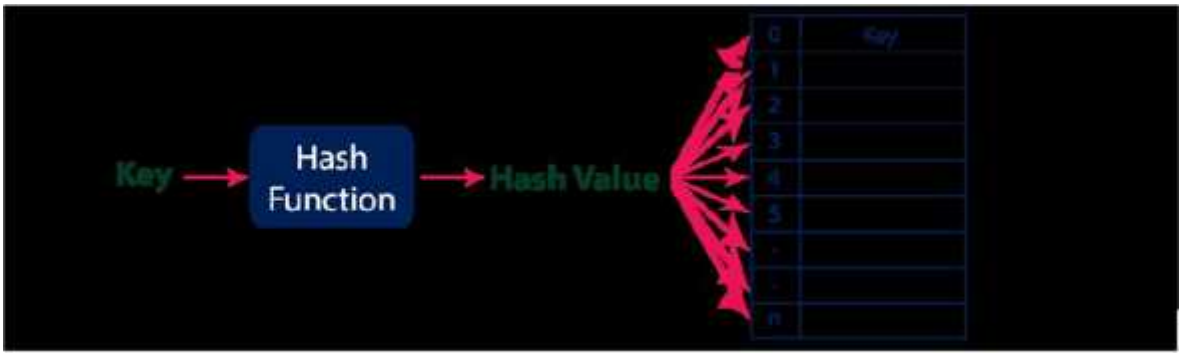


Fig.5.4.1 This kind of hashing is called static hashing since the size of the hash table is fixed.(an array)

Example:

Suppose we have a set of strings {"abc", "def", "ghi"} that we'd like to store in a table. Our objective here is to find or update them quickly from a table, actually in O(1). We are not concerned about ordering them or maintaining any order at all. Let us think of a simple schema to do this. Suppose we assign "a" = 1, "b"=2, … etc to all alphabetical characters. We can then simply compute a number for each of the strings by using the sum of the characters as follows.

"abc" = 1 + 2 + 3=6, "def" = 4 + 5 + 6=15 , "ghi" = 7 + 8 + 9=24

If we assume that we have a table of size 5 to store these strings, we can compute the location of the string by taking the sum mod 5. So we will then store " abc" in 6 mod 5 = 1, "def" in 15 mod 5 = 0,
and "ghi" in 24 mod 5 = 4 in locations 1, 0 and 4 as follows.

Now the idea is that if we are given a string, we can immediately compute the location using a simple hash function, which is sum of the characters mod Table size. Using this hash value, we can search for the string.

Problem with Hashing -collision

The method discussed above seems too good to be true as we begin to think more about the hash function. First of all, the hash function we used, that is the sum of the letters, is a bad one. In case we have permutations of the same letters, "abc", "bac" etc in the set, we will end up with the same value for the sum and hence the key. In this case, the strings would hash into the same location, creating what we call a "collision". This is obviously not a good thing. Secondly, we need to find a good table size, preferably a prime number so that even if the sums are different, then collisions can be avoided, when we take mod of the sum to find the location. So we ask two questions.

Question 1: How do we pick a good hash function?
Question 2: How do we deal with collisions?

The problem of storing and retrieving data in O(1) time comes down to answering the above questions. Picking a "good" hash function is key to successfully implementing a hash table. What we mean by "good" is that the function must be easy to compute and avoid collisions as much as possible. If the function is hard to compute, then we lose the advantage gained for lookups in O(1). Even if we pick a very good hash function, we still will have to deal with "some" collisions.

The process where two records can hash into the same location is called collision. We can deal with collisions using many strategies, such as linear probing (looking for the next available location i+1, i+2, etc. from the hashed value i), quadratic probing (same as linear probing, except we look for available positions i+1 , i + 4, i + 9, etc from the hashed value i and separate chaining, the process of creating a linked list of values if they hashed into the same location.This is called collision resolution.

Popular hash functions

Hash functions that use numeric keys are very popular.. However, there can be cases in real-world applications where we can have alphanumeric keys rather than simple numeric keys. In such cases, the ASCII value of the character can be used to transform it into its equivalent numeric key. Once this transformation is done, any hash function can be applied to generate the hash value.

Division Method

It is the most simple method of hashing an integer x. This method divides x by M and then

uses the remainder obtained. In this case, the hash function can be given as

h(x) = x mod M

The division method is quite good for just about any value of M and since it requires only a single division operation, the method works very fast. However, extra care should be taken to select a suitable value for M. Generally, it is best to choose M to be a prime number because making M a prime number increases the likelihood that the keys are mapped with a uniformity in the output range of values.

A potential drawback of the division method is that while using this method, consecutive keys map to consecutive hash values. On one hand, this is good as it ensures that consecutive keys do not collide, but on the other, it also means that consecutive array locations will be occupied. This may lead to degradation in performance.

Example :

Calculate the hash values of keys 1234 and 5462. Solution Setting M = 97, hash values can be calculated as:

h(1234) = 1234 % 97 = 70

h(5642) = 5642 % 97 = 16

Mid-Square Method

The mid-square method is a good hash function which works in two steps:

Step 1: Square the value of the key. That is, find k2.

Step 2: Extract the middle r digits of the result obtained in Step 1.

The algorithm works well because most or all digits of the key value contribute to the result. This is because all the digits in the original key value contribute to produce the middle digits of the squared value. Therefore, the result is not dominated by the distribution of the bottom digit or the top digit of the original key value. In the mid-square method, the same r digits must be chosen from all the keys. Therefore, the hash function can be given as:

h(k) = s where s is obtained by selecting r digits from k2.

Example Calculate the hash value for keys 1234 and 5642 using the mid-square method. The hash table has 100 memory locations. Solution Note that the hash table has 100 memory locations whose indices vary from 0 to 99.

This means that only two digits are needed to map the key to a location in the hash table,

so r = 2. When k = 1234, k2 = 1522756, h (1234) = 27

When k = 5642, k2 = 31832164, h (5642)
= 21

Observe that the 3rd and 4th digits starting from the right are chosen.

Folding Method

The folding method works in the following two steps:

Step 1: Divide the key value into a number of parts. That is, divide k into parts k1, k2, ..., kn, where each part has the same number of digits except the last part which may have lesser digits than the other parts.

Step 2: Add the individual parts. That is, obtain the sum of k1 + k2 + ... + kn. The hash value is produced by ignoring the last carry, if any. Note that the number of digits in each part of the key will vary depending upon the size of the hash table. .

Example Given a hash table of 100 locations, calculate the hash value using folding method for keys
5678, 321, and 34567. Solution Since there are 100 memory locations to address, we will break the key into parts where each part (except the last) will contain two digits. The hash values can be obtained as shown below:

key	5678	321	34567
Parts	56 and 78	32 and 1	34, 56 and 7
Sum	134	33	97
Hash value	34 (ignore the last carry)	33	97

Collision Resolution Strategies

1. Open Addressing/Closed Hashing
2. Chaining

Once a collision takes place, open addressing or closed hashing computes new positions using a probe sequence and the next record is stored in that position

The process of examining memory locations in the hash table is called probing. Open addressing technique can be implemented using linear probing, quadratic probing, double hashing.

Linear Probing

When using a linear probe to resolve collision, the item will be stored in the next available

slot in the table, assuming that the table is not already full.

This is implemented via a linear search for an empty slot, from the point of collision. If the physical end of table is reached during the linear search, the search will wrap around to the beginning of the table and continue from there.If an empty slot is not found before reaching the point of collision, the table is full.

If h is the point of collision, probe through h+1,h+2,h+3.....h+i. till an empty slot is found

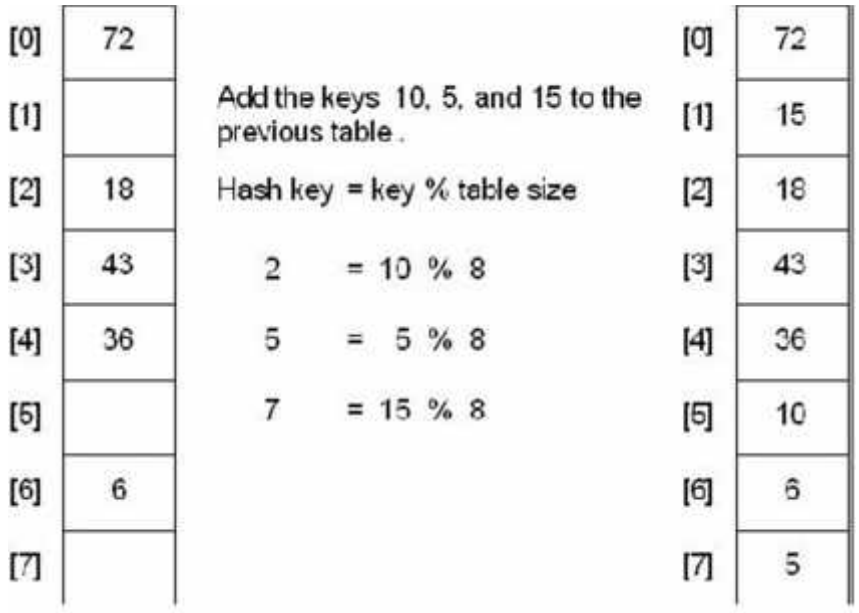


Fig.5.4.2 Searching a Value using Linear Probing

The procedure for searching a value in a hash table is same as for storing a value in a hash table. While searching for a value in a hash table, the array index is re-computed and the key of the elementstored at that location is compared with the value that has to be searched. If a match is found, then the search operation is successful. The search time in this case is given as O(1). If the key does not match, then the search function begins a sequential search of the array that continues until:

- the value is found,or
- the search function encounters a vacant location in the array, indicating that the value is not present, or
- the search function terminates because it reaches the end of the table and the value is not present.

Probe Sequence ::(h+i)%Table size

Disadvantage:

As the hash table fills, clusters of consecutive cells are formed and the time required for a search increases with the size of the cluster. It is possible for blocks of data to form when collisions are resolved. This means that any key that hashes into the cluster will require several attempts to resolve the collision. More the number of collisions, higher the probes that are required to find a free location and lesser is the performance. This phenomenon is called clustering. To avoid clustering, other techniques such as quadratic probing and double hashing are used.

Quadratic Probing

A variation of the linear probing idea is called quadratic probing. Instead of using a constant "skip" value, if the first hash value that has resulted in collision is h, the successive values which are probed are h+1, h+4, h+9, h+16, and so on. In other words, quadratic probing uses a skip consisting of successive perfect squares.

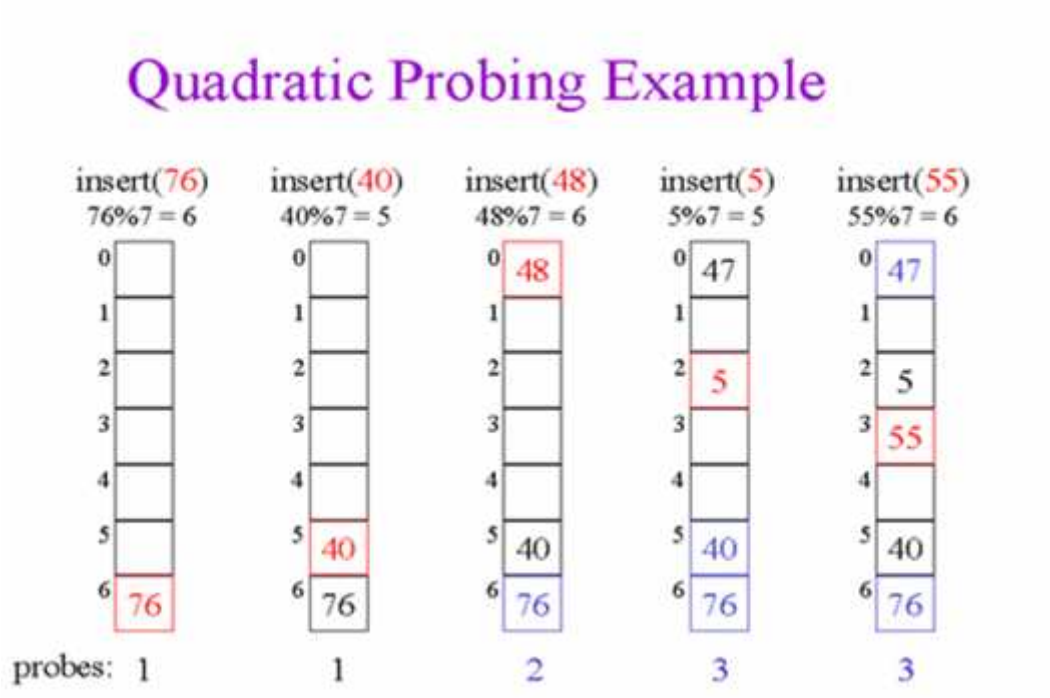


Fig.5.4.3 Quadratic Probing

Probe sequence :h,h+12,h=22,h=32.....h+i2

H(k)=(h+i2)%Tablesize

Double Hashing

In double hashing, we use two hash functions rather than a single function. Double hashing uses the idea of applying a second hash function to the key when a collision occurs. The result of the second hash function will be the number of positions form the point of collision to insert.There are a couple of requirements for the second function:

- it must never evaluate to 0
- must make sure that all cells can be probed

A popular second hash function is: Hash2(key) = R - (key % R) where R is a prime number that is smaller than the size of the table.But any independent hash function may also be used.

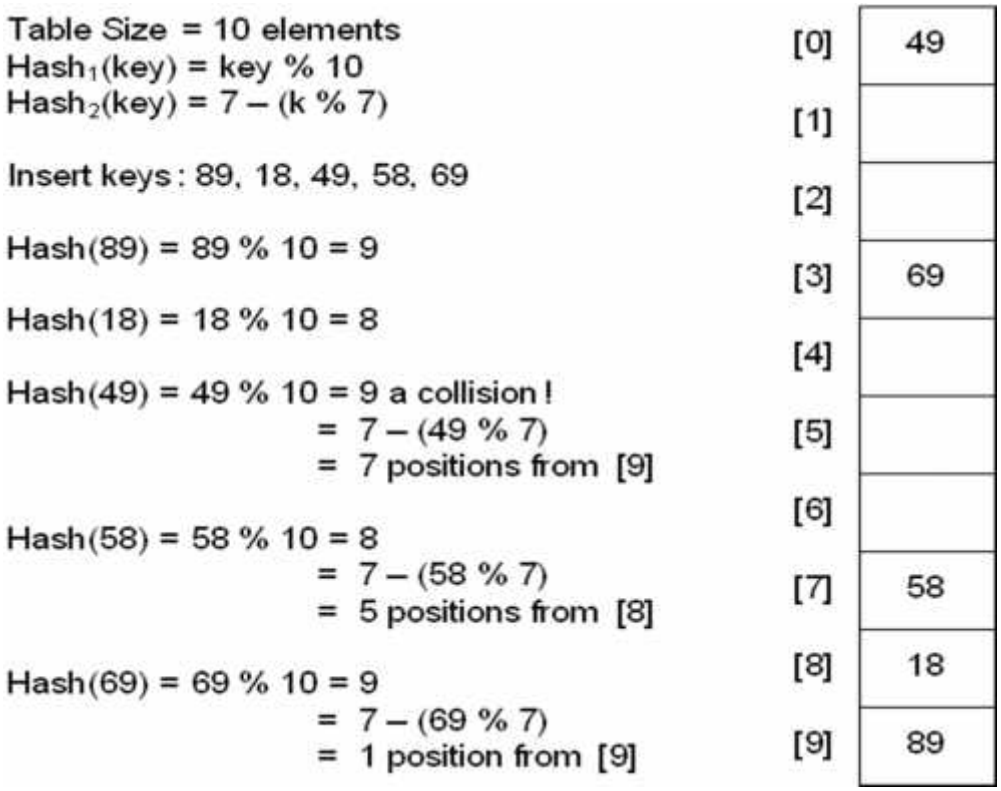
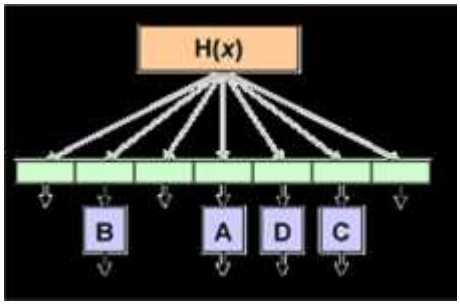


Fig.5.4.4 Double hashing minimizes repeated collisions and the effects of clustering.

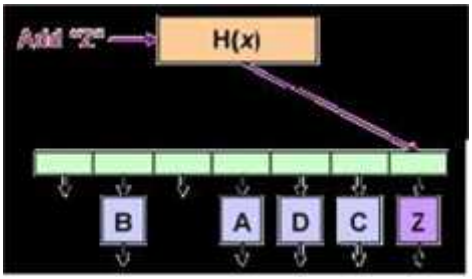
Chaining

Chaining is another approach to implementing a hash table; instead of storing the data directly inside the structure, have a linked list structure at each hash element. That way, all the collision, retrieval and deletion functions can be handled by the list, and the hash function's

role is limited mainly to that of a guide to the algorithms, as to which hash element's list to operate on.



The linked list at each hash element is often called a chain. A chaining hash table gets its name because of the linked list at each element -- each list looks like a 'chain' of data strung together.Operations on the data structure are made far simpler, as all of the data storage issues are handled by the list at the hash element, and not the hash table structure itself.



Drawbacks of static hashing

1. Table size is fixed and hence cannot accommodate data growth.

2. Collsions increases as data size grows.

Avoid the above conditions by doubling the hash table size. This increase in hash table size is taken up, when the number of collisions increase beyond a certain threshold. The threshold limit is decided by the load factor.

Load factor

The load factor α of a hash table with n key elements is given by α= n / hash table size

The probability of a collision increases as the load factor increases. We cannot just double the size of the table and copy the elements from the original table to the new table, since when the table size is doubled from N to 2N+1, the hash function changes. It requires reinserting each element of the old table into the new table (using the modified hash function).This is called Rehashing. Rehashing in large databases is a tedious process and hence dynamic hashing.

Dynamic hashing schemes

Dynamically increases the size of the hash table as collision occurs. There are two types:

Extendible hashing (directory): uses a directory that grows or shrinks depending on the data distribution. No overflow buckets

Linear hashing(directory less): No directory. Splits buckets in linear order, uses overflow

buckets.

Extendible hashing :

- Uses a directory of pointers to buckets/bins which are collections of records
- The number of buckets are doubled by doubling the directory, and splitting just the bin that overflowed.
- Directory much smaller than file, so doubling it is much cheaper. Only one bin of data entries is split and rehashed.

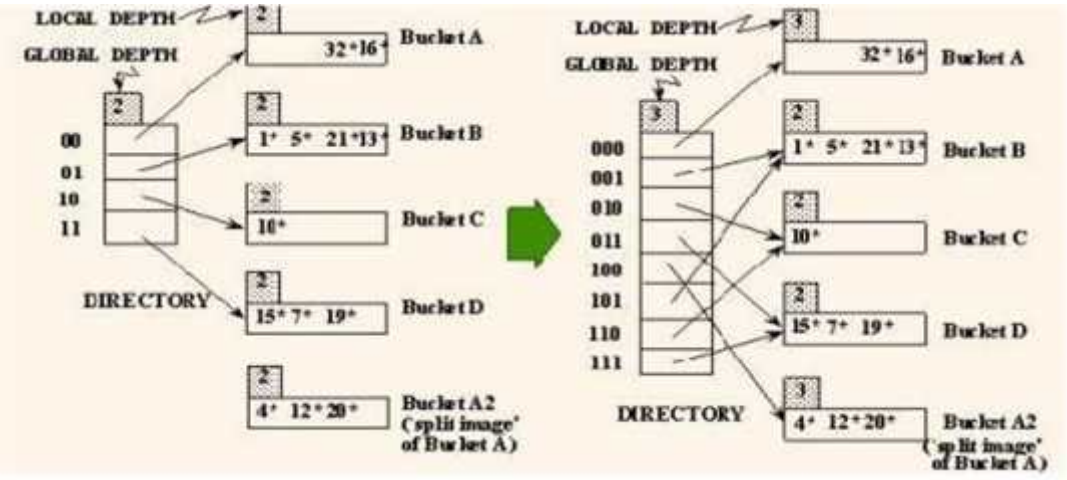


Fig.5.4.5 Extendible Hashing

Global Depth

– Max number of bits needed to tell which bucket an entry

belongs to. Local Depth

- The number of least significant digits that is common for all the numbers sharing the

same bin. On overflow:

If global depth =Local depth

1. Double the hash directory
2. Split the overflowing bin
3. Redistribute elements of the overflowing bin
4. Increment the global and local depth

If global depth >Local depth

1. Split the overflowing bin
2. Redistribute elements of the overflowing bin
3. Increment the local depth

Linear Hashing

Basic Idea:

Pages are split when overflows occur – but not necessarily the page with the overflow.

Directory avoided in LH by using overflow pages. (chaining approach)

Splitting occurs in turn, in a round robin fashion.one by one from the first bucket to the last bucket.

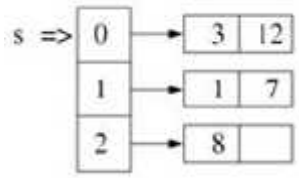
Use a family of hash functions h0, h1, h2, ...

– Each function's range is twice that of its predecessor.

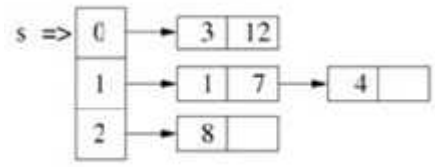
When all the pages at one level (the current hash function) have been split, a new level is applied.

Insert in Order using linear hashing: 1,7,3,8,12,4,11,2,10,13.....

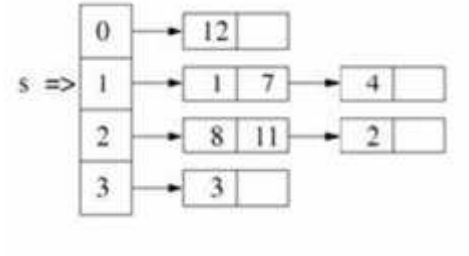
After insertion till 12:



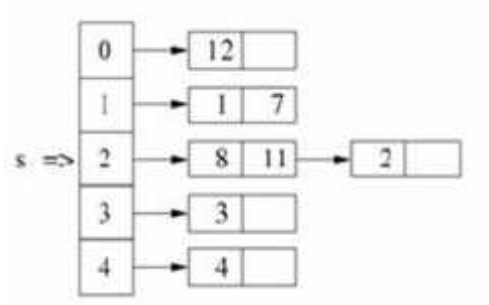
When 4 inserted overflow occurred. So we split the bucket (no matter it is full or partially empty). And increment pointer.



So we split bucket 0 and rehashed all keys in it. Placed 3 to new bucket as (3 mod 6 = 3) and (12 mod 6 = 0). Then 11 and 2 are inserted. And now overflow. s is pointing to bucket 1, hence split bucket 1 by re- hashing it.



After split:



Insertion of 10 and 13: as (10 mod 3 = 1) and bucket 1 < s, we need to hash 10 again using h1(10) =
10 mod 6 = 4th
bucket.

5.5 INDEXING

The indexing technique based on factors such as access type, access time, insertion time, deletion time, and space overhead involved. There are two kinds of indices:

- *Ordered indices* that are sorted based on one or more key values
- *Hash indices* that are based on the values generated by applying a *hash function*

1. Ordered Indices

Indices are used to provide fast random access to records. An index of a file may be a primary index or a secondary index.

Primary Index

In a sequentially ordered file, the index whose search key specifies the sequential order of the file is defined as the primary index.

Example: suppose records of students are stored in a STUDENT file in a sequential order starting from roll number 1 to roll number 60. Now, if we want to search a record for, say, roll number 10, then the student's roll number is the primary index.

Secondary Index

An index whose search key specifies an order different from the sequential order of the file is called as the secondary index.

Example: If the record of a student is searched by his name, then the name is a secondary index. Secondary indices are used to improve the performance of queries on non-primary keys.

2. Dense and Sparse Indices

Dense index

- In a dense index, the index table stores the address of every record in the file.
- Dense index would be more efficient to use than a sparse index if it fits in the memory
- By looking at the dense index, it can be concluded directly whether the record exists in the file or not.

Sparse index

- In a sparse index, the index table stores the address of only some of the records in the file.
- Sparse indices are easy to fit in the main memory,
- In a sparse index, to locate a record, first find an entry in the index table with the largest search key value that is either less than or equal to the search key value of the desired record. Then, start at that record pointed to by that entry in the index table and then proceed searching the record using the sequential pointers in the file, until the desired record is obtained.

Example: If we need to access record number 40, then record number 30 is the largest key value that is less than 40. So jump to the record pointed by record number 30 and move along the sequential pointer to reach record number 40.

Below figure shows a dense index and a sparse index for an indexed sequential file.

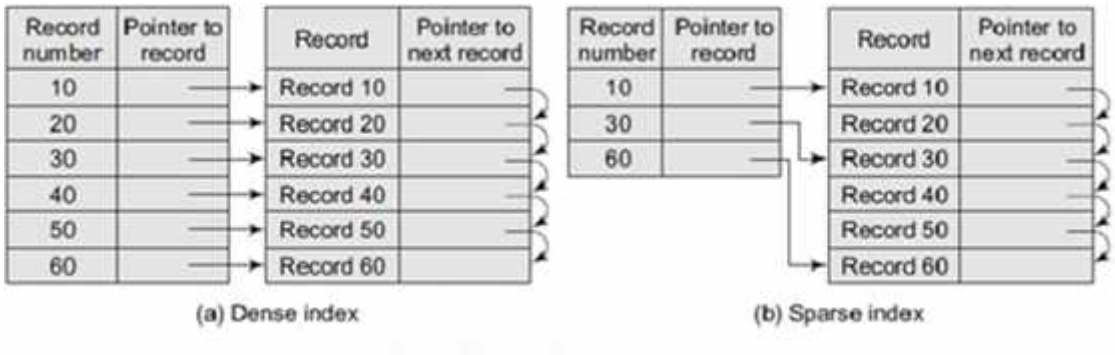


Fig.5.5.1 Dense index and sparse index

3. Cylinder Surface Indexing

Cylinder surface indexing is a very simple technique used only for the primary key index of a sequentially ordered file.

The index file will contain two fields—cylinder index and several surface indices. There are multiple cylinders, and each cylinder has multiple surfaces. If the file needs m cylinders for storage then the cylinder index will contain m entries.

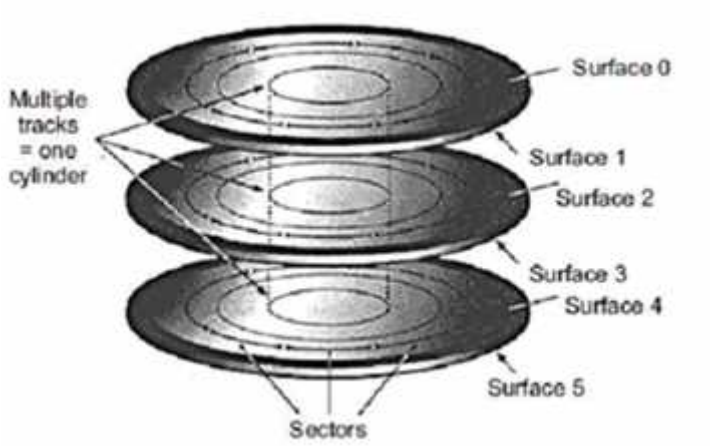


Fig.5.5.2 Physical and logical organization of disk

When a record with a particular key value has to be searched, then the following steps are performed:

- First the cylinder index of the file is read into memory.
- Second, the cylinder index is searched to determine which cylinder holds the desired record. For this, either the binary search technique can be used or the cylinder index can be made to store an array of pointers to the starting of individual key values. In either case the search will takeO (log m) time. After
- the cylinder index is searched, appropriate cylinder is determined.
- Depending on the cylinder, the surface index corresponding to the cylinder is then retrieved from the disk

- Once the cylinder and the surface are determined, the corresponding track is read and searched for the record with the desired key.

Hence, the total number of disk accesses is three—first, for accessing the cylinder index, second for accessing the surface index, and third for getting the track address.

4. Multi-level Indices

Consider very large files that may contain millions of records. For such files, a simple indexing technique will not suffice. In such a situation, we use multi-level indices.

Below figure shows a two-level multi-indexing. Three-level indexing and so, can also be used In the figure, the main index table stores pointers to three inner index tables. The inner index tables are sparse index tables that in turn store pointers to the records.

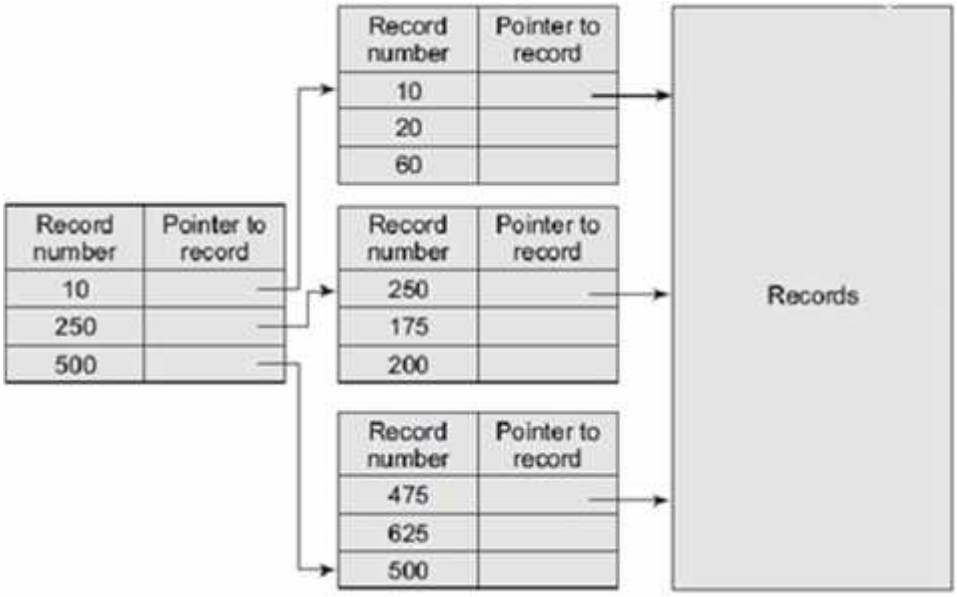


Fig.5.5.3 Multi-level indices

5. Inverted Indices

- Inverted files are used in document retrieval systems for large textual databases.
- An inverted file reorganizes the structure of an existing data file in order to provide fast access to all records having one field falling within the set limits.
- When a term or keyword specified in the inverted file is identified, the record number is given and a set of records corresponding to the search criteria are created.
- For each keyword, an inverted file contains an inverted list that stores a list of pointers to all occurrences of that term in the main text. Therefore, given a keyword, the addresses of all the documents containing that keyword can easily be located.

There are two main variants of inverted indices:

- A record-level inverted index (inverted file index or inverted file) stores a list of references to documents for each word
- A word-level inverted index (full inverted index or inverted list) in addition to a list of references to documents for each word also contains the positions of each word within a document.

7. Hashed Indices

Hashing is used to compute the address of a record by using a hash function on the search key value. The hashed values map to the same address, then collision occurs and schemes to resolve these collisions are applied to generate a new address

Choosing a good hash function is critical to the success of this technique. By a good hash function, it mean two things.

1. First, a good hash function, irrespective of the number of search keys, gives an average-case lookup that is a small constant.
2. Second, the function distributes records uniformly and randomly among the buckets, where a bucket is defined as a unit of one or more records

The worst hash function is one that maps all the keys to the same bucket.

The drawback of using hashed indices includes:

Though the number of buckets is fixed, the number of files may grow with time.
If the number of buckets is too large, storage space is wasted.
If the number of buckets is too small, there may be too many collisions.

The following operations are performed in a hashed file organization.

1. Insertion

To insert a record that has ki as its search value, use the hash function h(ki) to compute the address of the bucket for that record.
If the bucket is free, store the record else use chaining to store the record.

2. Search

To search a record having the key value ki, use h(ki) to compute the address of the bucket where the record is stored.
The bucket may contain one or several records, so check for every record in the bucket to retrieve the desired record with the given key value.

3. Deletion

To delete a record with key value ki, use h(ki) to compute the address of the bucket where the record is stored. The bucket may contain one or several records so check for every record in the bucket, and then delete the record.