< DATA STRUCTURE >

QUICK REVISION POINTS

MADE BY: ROHNAK AGARWAL, 2nd YEAR, 3rd SEM, 2019.

Made on: 01 – 01 - 2020
Last updated on: 03 – 01 - 2020

ARRAY[] AND MATRIX[][]

1. Determining memory address of an element in a matrix:

Matrix:	a[m][n]
Base address:	B
Size of each element:	s

   a. Row-Major Order
      Address of a[i][j] = B + [i*n + j]*s

   b. Column-Major Order
      Address of a[i][j] = B + [i + j*m]*s

2. 3-Tuple format:
   First row format:

total rows	total cols	total non-zero elements

   Successive rows format:

i	j	non-zero value (a[i][j])

STACK ⇄

FIFO

1. Infix-Postfix-Prefix important points
   prefix: polish notation
   postfix: reverse polish notation

	Postfix	Prefix
Conversion (operators to be sent to final expression)	>=	>
Evaluation A : Top element . B : Next to top element. Δ : operator encountered	B Δ A	A Δ B

2. Programs:

a. Simple stack operations:

```
#include<stdio.h>

int stack[100],choice,n,top,x,i;
void push(void);
void pop(void);
void display(void);
void main()
{
    top=-1;
    printf("\n Enter the size of STACK[MAX=100]:");
    scanf("%d",&n);
    printf("\n\t STACK OPERATIONS USING ARRAY");
    printf("\n\t--------------------------------");
    printf("\n\t 1.PUSH\n\t 2.POP\n\t 3.DISPLAY\n\t 4.EXIT");
    do
    {
        printf("\n Enter the Choice:");
        scanf("%d",&choice);
        switch(choice)
        {
            case 1:
            {push();break;}
            case 2:
            {pop();break;}
            case 3:
            {display();break;}
            case 4:
            {printf("\n\t EXIT POINT ");break;}
            default:
            {printf ("\n\t Please Enter a Valid Choice(1/2/3/4)");}
        }
    }while(choice!=4);
}
void push()
{
    if(top>=n-1)
    {printf("\n\tSTACK is over flow");}
    else
    {   printf(" Enter a value to be pushed:");
        scanf("%d",&x);
        top++;
        stack[top]=x; }
}
void pop()
{
    if(top<=-1)
    {printf("\n\t Stack is under flow");}
    else
    {printf("\n\t The popped elements is %d",stack[top]); top--;}
}
void display()
{   if(top>=0)
    {printf("\n The elements in STACK \n");
        for(i=top; i>=0; i--)
            printf("\n%d",stack[i]);
        printf("\n Press Next Choice");}
    else
    {printf("\n The STACK is empty");}
}
```

b. Infix to postfix using stack

```
#include<stdio.h>
char stack[20];
int top = -1;
void push(char x)
{
    stack[++top] = x;
}
char pop()
{
    if(top == -1)
        return -1;
    else
        return stack[top--];
}
int priority(char x)
{
    if(x == '(')                    return 0;
    if(x == '+' || x == '-')        return 1;
    if(x == '*' || x == '/')        return 2;
}
void main()
{
    char exp[20];
    char *e, x;
    printf("Enter the expression :: ");
    scanf("%s",exp);
    e = exp;
    while(*e != '\0')
    {
        if(isalnum(*e))
            printf("%c",*e);
        else if(*e == '(')
            push(*e);
        else if(*e == ')')
        {
            while((x = pop()) != '(')
                printf("%c", x);
        }
        else
        {
            while(priority(stack[top]) >= priority(*e))  printf("%c",pop());
            push(*e);
        }
        e++;
    }
    while(top != -1)
    {
        printf("%c",pop());
    }
}
```

c. Evaluation of postfix

```
#include <stdio.h>
#include <ctype.h>
#define MAXSTACK 100
#define POSTFIXSIZE 100
int stack[MAXSTACK];
int top = -1;
void push(int item)
{
    if (top >= MAXSTACK - 1) {printf("stack over flow");return;}
    else                     {top = top + 1;stack[top] = item;}
}
int pop()
{
    int item;
    if (top < 0) {printf("stack under flow");}
    else {item = stack[top]; top = top - 1;return item;}
    return -1;
}
void EvalPostfix(char postfix[])
{
    int i;
    char ch;
    int val;
    int A, B;
    for (i = 0; postfix[i] != ')'; i++) {
        ch = postfix[i];
        if (isdigit(ch)) {
            push(ch - '0');
        }
        else if (ch == '+' || ch == '-' || ch == '*' || ch == '/') {
            A = pop();
            B = pop();
            switch (ch)
            {
            case '*':
                val = B * A;
                break;
            case '/':
                val = B / A;
                break;
            case '+':
                val = B + A;
                break;
            case '-':
                val = B - A;
                break;
            }
            push(val);
        }
    }
    printf(" \n Result of expression evaluation : %d \n", pop());
}
void main()
{
    int i;
    char postfix[POSTFIXSIZE];
    for (i = 0; i <= POSTFIXSIZE - 1; i++) {
        scanf("%c", &postfix[i]);
        if (postfix[i] == ')') break;
    }
}
```

d. Infix to Prefix

```
#define SIZE 50            /* Size of Stack */
#include<string.h>
#include <ctype.h>
char s[SIZE];
int top=-1;
push(char elem)        {s[++top]=elem;}
char pop()             {return(s[top--]);}
int pr(char elem)
{    switch(elem)
     {
     case '#': return 0;
     case ')': return 1;
     case '+':
     case '-': return 2;
     case '*':
     case '/': return 3;
     }
}

void main()
{
     char infx[50],prfx[50],ch,elem;
     int i=0,k=0;
     printf("\n\nRead the Infix Expression ? ");
     scanf("%s",infx);
     push('#');
     strrev(infx);
     while( (ch=infx[i++]) != '\0')
     {
          if( ch == ')') push(ch);
          else
               if(isalnum(ch)) prfx[k++]=ch;
               else
                    if( ch == '(')
                    {
                         while( s[top] != ')') prfx[k++]=pop();
                         elem=pop();
                    }
                    else
                    {
                         while( pr(s[top]) >= pr(ch) ) prfx[k++]=pop();
                         push(ch);
                    }
     }
     while( s[top] != '#')     prfx[k++]=pop();
     prfx[k]='\0';
     strrev(prfx);
     strrev(infx);
     printf("\n\nGiven Infix Expn: %s  Prefix Expn: %s\n",infx,prfx);
}
```

e. Evaluation of prefix

```
#include <stdio.h>
#include <ctype.h>
#define MAXSTACK 100
#define POSTFIXSIZE 100
int stack[MAXSTACK];
int top = -1;
void push(int item)
{
    if (top >= MAXSTACK - 1) {printf("stack over flow");return;}
    else {top = top + 1;stack[top] = item;}
}
int pop()
{
    int item;
    if (top < 0) {printf("stack under flow");}
    else {item = stack[top];top = top - 1;return item;}
}
void EvalPostfix(char postfix[])
{
    int i;
    char ch;
    int val;
    int A, B;
    for (i = 0; postfix[i] != ')'; i++) {
        ch = postfix[i];
        if (isdigit(ch)) {push(ch - '0');}
        else if (ch == '+' || ch == '-' || ch == '*' || ch == '/') {
            A = pop();
            B = pop();
            switch (ch)
            {
            case '*':
                val = B * A;
                break;
            case '/':
                val = B / A;
                break;
            case '+':
                val = B + A;
                break;
            case '-':
                val = B - A;
                break;
            }
            push(val);
        }
    }
    printf(" \n Result of expression evaluation : %d \n", pop());
}
void main()
{
    int i;
    char postfix[POSTFIXSIZE];
    for (i = 0; i <= POSTFIXSIZE - 1; i++) {
        scanf("%c", &postfix[i]);
        if (postfix[i] == ')') {break;}
    }
    EvalPostfix(postfix);
}
```

→ QUEUE →

FIFO

Programs

Basic operations (Array)

```
#include <stdio.h>
#include<stdlib.h>
#define MAX 50
void insert();void delete();void display();
int queue_array[MAX];
int rear = - 1, front = - 1;
int main()
{       int choice;
        while (1)
        {
                printf("1.Insert element to queue n");
                printf("2.Delete element from queue n");
                printf("3.Display all elements of queue n");
                printf("4.Quit n");
                printf("Enter your choice : ");
                scanf("%d", &choice);
                switch(choice)
                {
                        case 1: insert();break;
                        case 2: delete();break;
                        case 3: display();break;
                        case 4: exit(1);
                        default: printf("Wrong choice n");
                }
        }
}
void insert()
{       int item;
        if(rear == MAX - 1) printf("Queue Overflow n");
        else
        {
                if(front== - 1) front = 0;
                printf("Inset the element in queue : ");
                scanf("%d", &item); rear = rear + 1;
                queue_array[rear] = item;
        }
}
void delete()
{       if(front == - 1 || front > rear) {printf("Queue Underflow n");return;}
        else
        {
                printf("Element deleted from queue is : %dn", queue_array[front]);
                front = front + 1;
        }
}
void display()
{       int i;
        if(front == - 1) printf("Queue is empty n");
        else
        {
                printf("Queue is : n");
                for(i = front; i <= rear; i++) printf("%d ", queue_array[i]);
                printf("n");
        }
}
```

CIRCULAR QUEUE

Programs
Basic operations (Array)

```
#include<stdio.h>
# define MAX 5
int cqueue_arr[MAX];
int front = -1, rear = -1;
void insert(int item)
{     if((front == 0 && rear == MAX-1) || (front == rear+1))
      {printf("Queue Overflow n");return;}
      if(front == -1)
      {front = 0;rear = 0;}
      else
      {     if(rear == MAX-1) rear = 0;
            else              rear = rear+1;
      }
      cqueue_arr[rear] = item ;
}
void deletion()
{     if(front == -1)
            {printf("Queue Underflown");return;}
      printf("Element deleted from queue is : %dn",cqueue_arr[front]);
      if(front == rear) {front = -1;rear=-1;}
      else
      {     if(front == MAX-1)      front = 0;
            else                    front = front+1;
      }
}
void display()
{     int front_pos = front,rear_pos = rear;
      if(front == -1) {printf("Queue is emptyn");return;}
      printf("Queue elements :n");
      if( front_pos <= rear_pos )
            while(front_pos <= rear_pos)
                  {printf("%d ",cqueue_arr[front_pos]);front_pos++;}
      else
      {     while(front_pos <= MAX-1)
                  {printf("%d ",cqueue_arr[front_pos])front_pos++;}
            front_pos = 0;
            while(front_pos <= rear_pos)
                  {printf("%d ",cqueue_arr[front_pos]);front_pos++;}
      }
      printf("n");
}
```

```
void main()
{
	int choice,item;
	do
	{
		printf("1.Insert\n");
		printf("2.Delete\n");
		printf("3.Display\n");
		printf("4.Quit\n");
		printf("Enter your choice : ");
		scanf("%d",&choice);
		switch(choice)
		{
			case 1 :
				printf("Input the element for insertion in queue : ");
				scanf("%d", &item);
				insert(item);
				break;
			case 2 :
				deletion();
				break;
			case 3:
				display();
				break;
			case 4:
				break;
			default:
				printf("Wrong choicen");
		}
	}while(choice!=4);
}
```

LINKED QUEUE

Program:

```
#include <stdio.h>
#include <conio.h>
#include <malloc.h>
struct node
{     int data;
      struct node *next;
};
struct queue
{
      struct node *front;
      struct node *rear;
};
struct queue *q;
void create_queue(struct queue *);
struct queue *insert(struct queue *,int);
struct queue *delete_element(struct queue *);
struct queue *display(struct queue *);
int peek(struct queue *);
int main()
{
      int val, option;
      create_queue(q);
      clrscr();
      do
      {
            printf("\n *****MAIN MENU*****");
            printf("\n 1. INSERT");
            printf("\n 2. DELETE");
            printf("\n 3. PEEK");
            printf("\n 4. DISPLAY");
            printf("\n 5. EXIT");
            printf("\n Enter your option : ");
            scanf("%d", &option);
            switch(option)
            {
                  case 1:
                        printf("\n Enter the number to insert in the queue:");
                        scanf("%d", &val);
                        q = insert(q,val);
                        break;
                  case 2:
                        q = delete_element(q);
                        break;
                  case 3:
                        val = peek(q);
                        if(val! = -1)
                        printf("\n The value at front of queue is : %d", val);
                        break;
                  case 4:
                        q = display(q);
                        break;
            }
      }while(option != 5);
      getch();
      return 0;
}
```

```
void create_queue(struct queue *q)
{     q -> rear = NULL;q -> front = NULL;       }
struct queue *insert(struct queue *q,int val)
{
      struct node *ptr;
      ptr = (struct node*)malloc(sizeof(struct node));
      ptr -> data = val;
      if(q -> front == NULL)
      {
            q -> front = ptr;
            q -> rear = ptr;
            q -> front -> next = q -> rear -> next = NULL;
      }
      else
      {
            q -> rear -> next = ptr;
            q -> rear = ptr;
            q -> rear -> next = NULL;
      }
      return q;
}
struct queue *display(struct queue *q)
{
      struct node *ptr;
      ptr = q -> front;
      if(ptr == NULL)
            printf("\n QUEUE IS EMPTY");
      else
      {
            printf("\n");
            while(ptr!=q -> rear)
                  {printf("%d\t", ptr -> data);ptr = ptr -> next;}
            printf("%d\t", ptr -> data);
      }
      return q;
}
struct queue *delete_element(struct queue *q)
{
     struct node *ptr;
      ptr = q -> front;
      if(q -> front == NULL)
            printf("\n UNDERFLOW");
      else
      {
           q -> front = q -> front -> next;
            printf("\n The value being deleted is : %d", ptr -> data);
            free(ptr);
      }
      return q;
}
int peek(struct queue *q)
{     if(q->front==NULL)
      {     printf("\n QUEUE IS EMPTY");
            return -1;
      }
      else
            return q->front->data;
}
```

PRIORITY QUEUE

Program (Linked List)

```
#include <stdio.h>
#include <malloc.h>
#include <conio.h>
struct node
{
      int data;
      int priority;
      struct node *next;
};
struct node *start=NULL;
struct node *insert(struct node *);
struct node *delete(struct node *);
void display(struct node *);
int main()
{
      int option;
      clrscr();
      do
      {
            printf("\n *****MAIN MENU*****");
            printf("\n 1. INSERT");
            printf("\n 2. DELETE");
            printf("\n 3. DISPLAY");
            printf("\n 4. EXIT");
            printf("\n Enter your option : ");
            scanf( "%d", &option);
            switch(option)
            {
                  case 1:
                        start=insert(start);
                        break;
                  case 2:
                        start = delete(start);
                        break;
                  case 3:
                        display(start);
                        break;
            }
      }while(option!=4);
}
struct node *insert(struct node *start)
{     int val, pri;
      struct node *ptr, *p;
      ptr = (struct node *)malloc(sizeof(struct node));
      printf("\n Enter the value and its priority : " );
      scanf( "%d %d", &val, &pri);
      ptr->data = val;
      ptr->priority = pri;
      if(start==NULL || pri < start->priority )
            {ptr->next = start;start = ptr;}
      else
      {
            p = start;
            while(p->next != NULL && p->next->priority <= pri)    p = p->next;
            ptr->next = p->next;
            p->next = ptr;
      }
return start;
}
```

```
struct node *delete(struct node *start)
{
	struct node *ptr;
	if(start == NULL)
		{printf("\n UNDERFLOW" );return;}
	else
	{
		ptr = start;
		printf("\n Deleted item is: %d", ptr->data);
		start = start->next;
		free(ptr);
	}
	return start;
}
void display(struct node *start)
{
	struct node *ptr;
	ptr = start;
	if(start == NULL)
		printf("\nQUEUE IS EMPTY" );
	else
	{
		printf("\n PRIORITY QUEUE IS : " );
		while(ptr != NULL)
		{
			printf( "\t%d[priority=%d]", ptr->data, ptr->priority );
			ptr=ptr->next;
		}
	}
}
```

⇌ DEQUE ⇌

Program
Basic operations (Array)

```
# include<stdio.h>
# define MAX 5
int deque_arr[MAX];
int left = -1, right = -1;
void insert_right()
{     int added_item;
      if((left == 0 && right == MAX-1) || (left == right+1))
                                    {printf("Queue Overflow\n");return;}
      if (left == -1)               {left = 0;right = 0;}
      else  if(right == MAX-1)      right = 0;
      else                          right = right+1;
      printf("Input the element for adding in queue : ");
      scanf("%d", &added_item);
      deque_arr[right] = added_item ;
}
void insert_left()
{     int added_item;
      if((left == 0 && right == MAX-1) || (left == right+1))
                              {printf("Queue Overflow \n");return;}
      if (left == -1)         {left = 0;right = 0;}
      else  if(left== 0)      left=MAX-1;
      else                    left=left-1;
      printf("Input the element for adding in queue : ");
      scanf("%d", &added_item);
      deque_arr[left] = added_item ;
}
void delete_left()
{     if (left == -1)         {printf("Queue Underflow\n");return;}
      printf("Element deleted from queue is : %d\n",deque_arr[left]);
      if(left == right)       {left = -1;right=-1;}
      else if(left == MAX-1)  left = 0;
      else                    left = left+1;
}
void delete_right()
{     if (left == -1)         {printf("Queue Underflow\n");return ;}
      printf("Element deleted from queue is : %d\n",deque_arr[right]);
      if(left == right)       {left = -1;right=-1;}
      else  if(right == 0)    right=MAX-1;
      else                    right=right-1;
}
void display_queue()
{     int front_pos = left,rear_pos = right;
      if(left == -1)    {printf("Queue is empty\n");return;}
      printf("Queue elements :\n");
      if( front_pos <= rear_pos )
      {     while(front_pos <= rear_pos)
                  {printf("%d ",deque_arr[front_pos]);front_pos++;}
      }else{
            while(front_pos <= MAX-1)
                  {printf("%d ",deque_arr[front_pos]);front_pos++;}
            front_pos = 0;
            while(front_pos <= rear_pos)
                  {printf("%d ",deque_arr[front_pos]);front_pos++;}
      }
      printf("\n");
}
```

```
void input_que()
{     int choice;
      do
      {     printf("1.Insert at right\n");
            printf("2.Delete from left\n");
            printf("3.Delete from right\n");
            printf("4.Display\n");
            printf("5.Quit\n");
            printf("Enter your choice : ");
            scanf("%d",&choice);
            switch(choice)
            {     case 1:
                        insert_right();
                        break;
                  case 2:
                        delete_left();
                        break;
                  case 3:
                        delete_right();
                        break;
                  case 4:
                        display_queue();
                        break;
                  case 5:
                        break;
                  default:
                        printf("Wrong choice\n");
            }
      }while(choice!=5);
}
void output_que()
{     int choice;
      do
      {     printf("1.Insert at right\n");
            printf("2.Insert at left\n");
            printf("3.Delete from left\n");
            printf("4.Display\n");
            printf("5.Quit\n");
            printf("Enter your choice : ");
            scanf("%d",&choice);
            switch(choice)
            {
             case 1:
                  insert_right();
                  break;
             case 2:
                  insert_left();
                  break;
             case 3:
                  delete_left();
                  break;
             case 4:
                  display_queue();
                  break;
             case 5:
                  break;
             default:
                  printf("Wrong choice\n");
            }
      } while (choice!=5);
}
```

```
void main()
{     int choice;
      printf("1.Input restricted dequeue\n");
      printf("2.Output restricted dequeue\n");
      printf("Enter your choice : ");
      scanf("%d",&choice);
      switch(choice)
      {
       case 1 :
            input_que();
            break;
       case 2:
            output_que();
            break;
       default:
            printf("Wrong choice\n");
      }
}
```

SINGLY LINKED LIST

Program
All operations

```
#include <stdio.h>
#include <malloc.h>
#define ISEMPTY printf("\nEMPTY LIST:");
struct node
{    int value;
     struct node *next;
};

snode* create_node(int);
void insert_node_first();
void insert_node_last();
void insert_node_pos();
void sorted_ascend();
void delete_pos();
void search();
void update_val();
void display();
void rev_display(snode *);
typedef struct node snode;
snode *newnode, *ptr, *prev, *temp;
snode *first = NULL, *last = NULL;

int main()
{   int ch;
    char ans = 'Y';
    while (ans == 'Y'||ans == 'y')
    {   printf("\n---------------------------------\n");
        printf("\nOperations on singly linked list\n");
        printf("\n---------------------------------\n");
        printf("\n1.Insert node at first");
        printf("\n2.Insert node at last");
        printf("\n3.Insert node at position");
        printf("\n4.Sort Linked List in Ascending Order");
        printf("\n5.Delete Node from any Position");
        printf("\n6.Update Node Value");
        printf("\n7.Search Element in the linked list");
        printf("\n8.Display List from Beginning to end");
        printf("\n9.Display List from end using Recursion");
        printf("\n10.Exit\n");
        printf("\n~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~\n");
        printf("\nEnter your choice");
        scanf("%d", &ch);
        switch (ch)
        {
        case 1:
            printf("\n...Inserting node at first...\n");
            insert_node_first();
            break;
        case 2:
            printf("\n...Inserting node at last...\n");
            insert_node_last();
            break;
        case 3:
            printf("\n...Inserting node at position...\n");
            insert_node_pos();
            break;
```

```
        case 4:
            printf("\n...Sorted Linked List in Ascending Order...\n");
            sorted_ascend();
            break;
        case 5:
            printf("\n...Deleting Node from any Position...\n");
            delete_pos();
            break;
        case 6:
            printf("\n...Updating Node Value...\n");
            update_val();
            break;
        case 7:
            printf("\n...Searching Element in the List...\n");
            search();
            break;
        case 8:
            printf("\n...Displaying List From Beginning to End...\n");
            display();
            break;
        case 9:
            printf("\n...Displaying List From End using Recursion...\n");
            rev_display(first);
            break;
        case 10:
            printf("\n...Exiting...\n");
            return 0;
            break;
        default:
            printf("\n...Invalid Choice...\n");
            break;
        }
        printf("\nYOU WANT TO CONTINUE (Y/N)");
        scanf(" %c", &ans);
    }
    return 0;
 }
snode* create_node(int val)
{   newnode = (snode *)malloc(sizeof(snode));
    if (newnode == NULL)
            {printf("\nMemory was not allocated");return 0;}
    else
    {   newnode->value = val;
        newnode->next = NULL;
        return newnode;
    }
}
void insert_node_first()
{
    int val;
    printf("\nEnter the value for the node:");
    scanf("%d", &val);
    newnode = create_node(val);
    if (first == last && first == NULL)
    {   first = last = newnode;
        first->next = NULL;
        last->next = NULL;
    }
    else
    {   temp = first;
        first = newnode;
        first->next = temp;
    }
    printf("\n----INSERTED----");
}
```

```c
void insert_node_last()
{   int val;
    printf("\nEnter the value for the Node:");
    scanf("%d", &val);
    newnode = create_node(val);
    if (first == last && last == NULL)
    {   first = last = newnode;
        first->next = NULL;
        last->next = NULL;
    }
    else
    {   last->next = newnode;
        last = newnode;
        last->next = NULL;
    }
 printf("\n----INSERTED----");
}
void insert_node_pos()
{
    int pos, val, cnt = 0, i;
    printf("\nEnter the value for the Node:");
    scanf("%d", &val);
    newnode = create_node(val);
    printf("\nEnter the position ");
    scanf("%d", &pos);
    ptr = first;
    while (ptr != NULL)   {ptr = ptr->next;cnt++;}
    if (pos == 1)
    {   if (first == last && first == NULL)
        {   first = last = newnode;
            first->next = NULL;
            last->next = NULL;
        }
        else
        {   temp = first;
            first = newnode;
            first->next = temp;
        }
        printf("\nInserted");
    }else if (pos>1 && pos<=cnt) {
        ptr = first;
        for (i = 1;i < pos;i++) {prev = ptr;ptr = ptr->next;}
        prev->next = newnode;
        newnode->next = ptr;
        printf("\n----INSERTED----");
    }else{printf("Position is out of range");}
}
void sorted_ascend()
{   snode *nxt;    int t;
    if (first == NULL) {ISEMPTY; printf(":No elements to sort\n");}
    else
    {   for (ptr = first;ptr != NULL;ptr = ptr->next)
        {   for (nxt = ptr->next;nxt != NULL;nxt = nxt->next)
            {   if (ptr->value > nxt->value)
                {   t = ptr->value;
                    ptr->value = nxt->value;
                    nxt->value = t;
                }
            }
        }
        printf("\n---Sorted List---");
        for (ptr = first;ptr != NULL;ptr = ptr->next)
        {printf("%d\t", ptr->value);}
    }
}
```

```
void delete_pos()
{   int pos, cnt = 0, i;
    if (first == NULL)
    {ISEMPTY;printf(":No node to delete\n");}
    else
    {   printf("\nEnter the position of value to be deleted:");
        scanf(" %d", &pos);
        ptr = first;
        if (pos == 1) {first = ptr->next; printf("\nElement deleted");}
        else
        {   while (ptr != NULL)
                {ptr = ptr->next; cnt = cnt + 1;}
            if (pos > 0 && pos <= cnt)
            {   ptr = first;
                for (i = 1;i < pos;i++)
                  {prev = ptr; ptr = ptr->next; }
                prev->next = ptr->next;
            }
            else
                {printf("Position is out of range");}
        free(ptr);
        printf("\nElement deleted");
        }
    }
}
void update_val()
{   int oldval, newval, flag = 0;
    if (first == NULL)
        {ISEMPTY; printf(":No nodes in the list to update\n");}
    else
    {   printf("\nEnter the value to be updated:");
        scanf("%d", &oldval);
        printf("\nEnter the newvalue:");
        scanf("%d", &newval);
        for (ptr = first;ptr != NULL;ptr = ptr->next)
        {   if (ptr->value == oldval)
            {   ptr->value = newval;
                flag = 1;
                break;
            }
        }
        if (flag == 1)  {printf("\nUpdated Successfully");}
        else            {printf("\nValue not found in List");}
    }
}
void search()
{   int flag = 0, key, pos = 0;
    if (first == NULL) {ISEMPTY;printf(":No nodes in the list\n");}
    else
    {   printf("\nEnter the value to search");
        scanf("%d", &key);
        for (ptr = first;ptr != NULL;ptr = ptr->next)
        {   pos = pos + 1;
            if (ptr->value == key) {flag = 1;break;}
        }
        if (flag == 1)  {printf("\nElement %d found at %d position\n", key, pos);}
        else            {printf("\nElement %d not found in list\n", key);}
    }
}
```

```
void display()
{   if (first == NULL) {ISEMPTY;printf(":No nodes in the list to display\n");}
    else
    {   for (ptr = first;ptr != NULL;ptr = ptr->next)
            {printf("%d\t", ptr->value);}
    }
}
void rev_display(snode *ptr)
{
    int val;

    if (ptr == NULL)
    {
        ISEMPTY;
        printf(":No nodes to display\n");
    }
    else
    {
        if (ptr != NULL)
        {
            val = ptr->value;
            rev_display(ptr->next);
            printf("%d\t", val);
        }

    }
}
```

DOUBLY LINKED LIST

Program
All operations

```
#include <stdio.h>
#include <stdlib.h>
struct node
{
    struct node *prev;
    int n;
    struct node *next;
}*h,*temp,*temp1,*temp2,*temp4;

void insert1();
void insert2();
void insert3();
void traversebeg();
void traverseend(int);
void sort();
void search();
void update();
void delete();

int count = 0;

void main()
{
    int ch;

    h = NULL;
    temp = temp1 = NULL;

    printf("\n 1 - Insert at beginning");
    printf("\n 2 - Insert at end");
    printf("\n 3 - Insert at position i");
    printf("\n 4 - Delete at i");
    printf("\n 5 - Display from beginning");
    printf("\n 6 - Display from end");
    printf("\n 7 - Search for element");
    printf("\n 8 - Sort the list");
    printf("\n 9 - Update an element");
    printf("\n 10 - Exit");

    while (1)
    {
        printf("\n Enter choice : ");
        scanf("%d", &ch);
        switch (ch)
        {
        case 1:
            insert1();
            break;
        case 2:
            insert2();
            break;
        case 3:
            insert3();
            break;
        case 4:
            delete();
            break;
```

```
        case 5:
            traversebeg();
            break;
        case 6:
            temp2 = h;
            if (temp2 == NULL)
                printf("\n Error : List empty to display ");
            else
            {
                printf("\n Reverse order of linked list is : ");
                traverseend(temp2->n);
            }
            break;
        case 7:
            search();
            break;
        case 8:
            sort();
            break;
        case 9:
            update();
            break;
        case 10:
            exit(0);
        default:
            printf("\n Wrong choice menu");
        }
    }
}

/* TO create an empty node */
void create()
{
    int data;

    temp =(struct node *)malloc(1*sizeof(struct node));
    temp->prev = NULL;
    temp->next = NULL;
    printf("\n Enter value to node : ");
    scanf("%d", &data);
    temp->n = data;
    count++;
}

/*  TO insert at beginning */
void insert1()
{
    if (h == NULL)
    {
        create();
        h = temp;
        temp1 = h;
    }
    else
    {
        create();
        temp->next = h;
        h->prev = temp;
        h = temp;
    }
}

/* To insert at end */
void insert2()
{
```

```
    if (h == NULL)
    {
        create();
        h = temp;
        temp1 = h;
    }
    else
    {
        create();
        temp1->next = temp;
        temp->prev = temp1;
        temp1 = temp;
    }
}

/* To insert at any position */
void insert3()
{
    int pos, i = 2;

    printf("\n Enter position to be inserted : ");
    scanf("%d", &pos);
    temp2 = h;

    if ((pos < 1) || (pos >= count + 1))
    {
        printf("\n Position out of range to insert");
        return;
    }
    if ((h == NULL) && (pos != 1))
    {
        printf("\n Empty list cannot insert other than 1st position");
        return;
    }
    if ((h == NULL) && (pos == 1))
    {
        create();
        h = temp;
        temp1 = h;
        return;
    }
    else
    {
        while (i < pos)
        {
            temp2 = temp2->next;
            i++;
        }
        create();
        temp->prev = temp2;
        temp->next = temp2->next;
        temp2->next->prev = temp;
        temp2->next = temp;
    }
}

/* To delete an element */
void delete()
{
    int i = 1, pos;

    printf("\n Enter position to be deleted : ");
    scanf("%d", &pos);
    temp2 = h;
```

```
    if ((pos < 1) || (pos >= count + 1))
    {
        printf("\n Error : Position out of range to delete");
        return;
    }
    if (h == NULL)
    {
        printf("\n Error : Empty list no elements to delete");
        return;
    }
    else
    {
        while (i < pos)
        {
            temp2 = temp2->next;
            i++;
        }
        if (i == 1)
        {
            if (temp2->next == NULL)
            {
                printf("Node deleted from list");
                free(temp2);
                temp2 = h = NULL;
                return;
            }
        }
        if (temp2->next == NULL)
        {
            temp2->prev->next = NULL;
            free(temp2);
            printf("Node deleted from list");
            return;
        }
        temp2->next->prev = temp2->prev;
        if (i != 1)
            temp2->prev->next = temp2->next;    /* Might not need this statement if i
== 1 check */
        if (i == 1)
            h = temp2->next;
        printf("\n Node deleted");
        free(temp2);
    }
    count--;
}

/* Traverse from beginning */
void traversebeg()
{
    temp2 = h;

    if (temp2 == NULL)
    {
        printf("List empty to display \n");
        return;
    }
    printf("\n Linked list elements from begining : ");

    while (temp2->next != NULL)
    {
        printf(" %d ", temp2->n);
        temp2 = temp2->next;
    }
    printf(" %d ", temp2->n);
}
```

```
/* To traverse from end recursively */
void traverseend(int i)
{
    if (temp2 != NULL)
    {
        i = temp2->n;
        temp2 = temp2->next;
        traverseend(i);
        printf(" %d ", i);
    }
}

/* To search for an element in the list */
void search()
{
    int data, count = 0;
    temp2 = h;

    if (temp2 == NULL)
    {
        printf("\n Error : List empty to search for data");
        return;
    }
    printf("\n Enter value to search : ");
    scanf("%d", &data);
    while (temp2 != NULL)
    {
        if (temp2->n == data)
        {
            printf("\n Data found in %d position",count + 1);
            return;
        }
        else
             temp2 = temp2->next;
            count++;
    }
    printf("\n Error : %d not found in list", data);
}

/* To update a node value in the list */
void update()
{
    int data, data1;

    printf("\n Enter node data to be updated : ");
    scanf("%d", &data);
    printf("\n Enter new data : ");
    scanf("%d", &data1);
    temp2 = h;
    if (temp2 == NULL)
    {
        printf("\n Error : List empty no node to update");
        return;
    }
    while (temp2 != NULL)
    {
        if (temp2->n == data)
        {

            temp2->n = data1;
            traversebeg();
            return;
        }
        else
```

```
            temp2 = temp2->next;
    }

    printf("\n Error : %d not found in list to update", data);
}

/* To sort the linked list */
void sort()
{
    int i, j, x;

    temp2 = h;
    temp4 = h;

    if (temp2 == NULL)
    {
        printf("\n List empty to sort");
        return;
    }

    for (temp2 = h; temp2 != NULL; temp2 = temp2->next)
    {
        for (temp4 = temp2->next; temp4 != NULL; temp4 = temp4->next)
        {
            if (temp2->n > temp4->n)
            {
                x = temp2->n;
                temp2->n = temp4->n;
                temp4->n = x;
            }
        }
    }
    traversebeg();
}
```

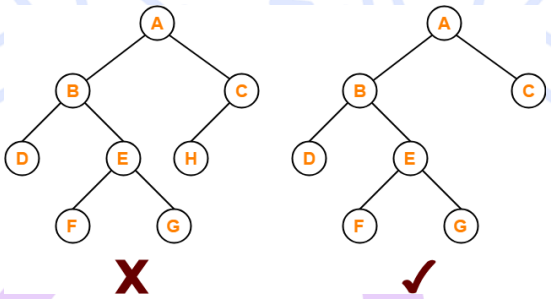
TREE

1. InOrder: Left Root Right
2. PreOrder: Root Left Right
3. PostOrder: Left Right Root

Types of trees:

A. Binary Tree: Each node has Maximum 2 sub-trees.

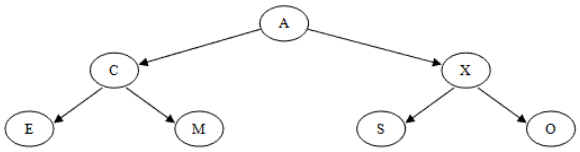
a. Strictly BT: Each non-leaf has both sub-trees.



✗ ✓

if number of leaf nodes = n,
then, total nodes in tree = 2*n-1

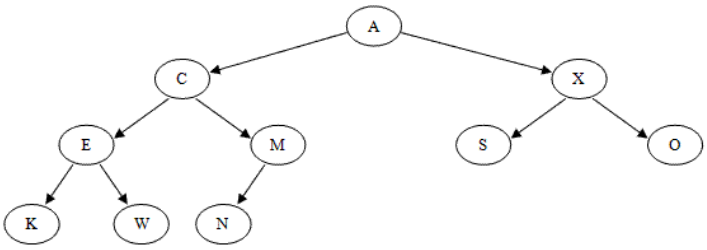
b. Complete BT: Each non-leaf has both sub-trees and all leaves at same level.



if level of tree = l,
then, maximum number of nodes (total) = pow(2, l+1) -1

c. Almost Complete BT:

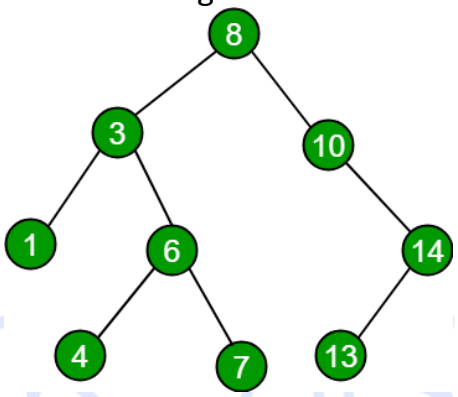
i. Each leaf is either at depth **d** or **d-1**.
ii. For any node with right descendant at depth d, all left descendants of nodes that are leaves are also at depth d.



if number of leaf nodes = n,
then total number of nodes = 2*n-1
if total number of nodes = n,
number of different BTs of different shapes = $\frac{1}{n+1}\ {}^{2n}C_n$

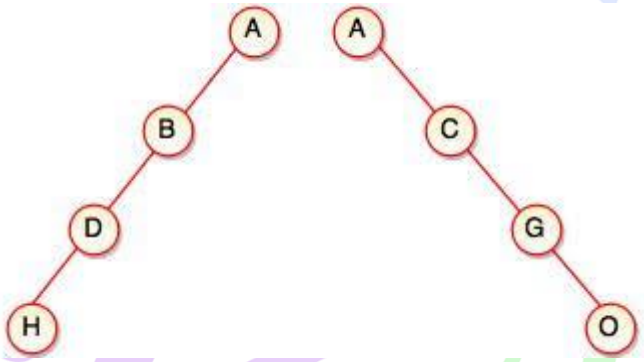
d. Binary Search Tree:

value: left < root < right



e. Skewed Binary Tree:

Left Skewed BT: Right Skewed BT:



Representation of Simple BT in memory:

a. Array:

i. index of main root 0

ii. index of left child of a parent at index i 2*i + 1

iii. index of left child of a parent at index i 2*i + 2

b. Linked List:

Structure of each node:

```
struct node{
      struct node *left, *right;
      int data;
};
```

Preorder traversal:

```
void preorder(struct node *root){
      if(root==null) return;
      printf("%d ", root->data);
      preorder(root->left);
      preorder(root->right);
};
```

Inorder traversal:

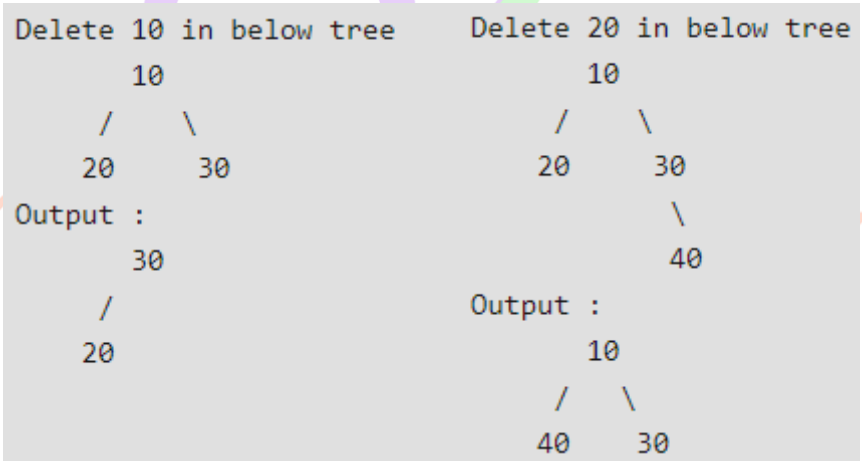
```
void inorder(struct node *root){
      if(root==null) return;
      inorder(root->left);
      printf("%d ", root->data);
      inorder(root->right);
};
```

Postorder traversal:

```
void postorder(struct node *root){
      if(root==null) return;
      postorder(root->left);
      postorder(root->right);
      printf("%d ", root->data);
};
```

Deletion of node in simple Binary Tree:

Given a binary tree, delete a node from it by making sure that tree shrinks from the bottom (i.e. the deleted node is replaced by bottom most and rightmost node).



BINARY SEARCH TREE

Condition: left < root < right

Program:

Structure of each node:

```
struct node{
      struct node *left, *right;
      int data;
};
```

Insertion:

```
struct node* insert(struct node* root, int data)
{
    if (root == NULL) return createNode(data);
    if (data < root->data)
        root->left  = insert(root->left, data);
    else if (data > root->data)
        root->right = insert(root->right, data);

    return root;
}
```

Traversal: (use only inorder)

```
void inorder(struct node *root){
      if(root==null) return;
      inorder(root->left);
      printf("%d ", root->data);
      inorder(root->right);
};
```

Deletion of a node:

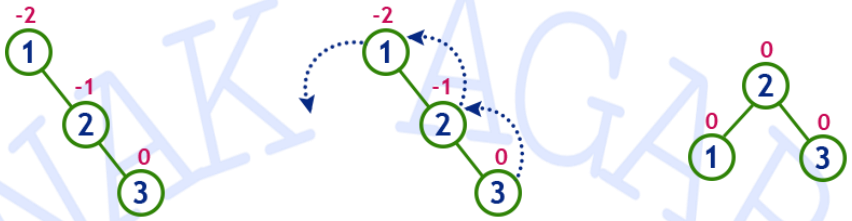
```
node* Delete( node* root,int value)
{     c=Search(root,value);
      if(root==NULL)
            return root;
      else if(value< root->data)
            root->left= Delete(root->left,value);
      else if(value> root->data)
            root->right= Delete(root->right,value);
      else
      {     if(root->left==NULL && root->right==NULL)
                  {delete root; root=NULL; return root;}
            else if(root->left==NULL)
            {     struct node* temp=root;
                  root=root->right;
                  delete temp;
                  return root;
            }
            else if(root->right==NULL)
            {     struct node* temp=root;
                  root=root->left;
                  delete temp; return root;
            }
            else
            {     struct node*temp=findMin(root->right);
                  root->data=temp->data;
                  root->right=Delete(root->right,temp->data);
            }
      }
      return root;
}
```

AVL TREE

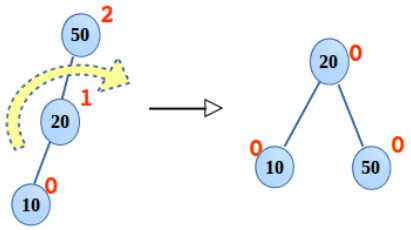
Condition: balance factor = {-1, 0, 1}
if balance factor >1 or <-1
then, rebalance.

Rebalancing techniques:

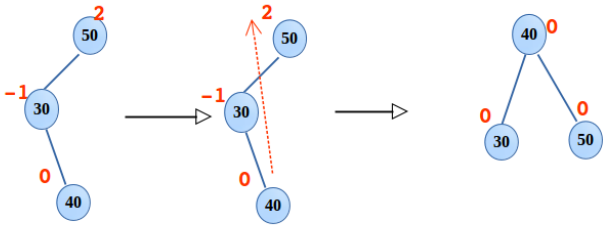
1. RR



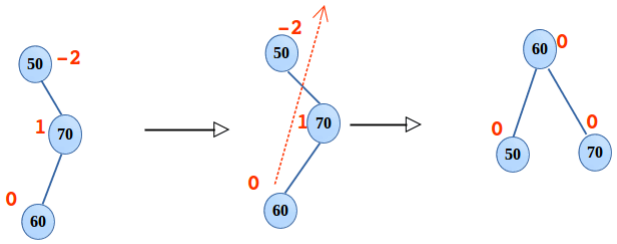
2. LL



3. LR



4. RL



BUBBLE SORT

```
void bubbleSort(int arr[], int n)
{  int i, j, temp;
   for (i = 0; i < n-1; i++)
       // Last i elements are already in place
       for (j = 0; j < n-i-1; j++)
           if (arr[j] > arr[j+1])
              {   temp = arr[j];
                  arr[j] = arr[j+1];
                  arr[j+1] = temp;
              }
}
```

SELECTION SORT

```
void selectionSort(int arr[], int n)
{   int i, j, min_idx, temp;
    // One by one move boundary of unsorted subarray
    for (i = 0; i < n-1; i++)
    {   // Find the minimum element in unsorted array
        min_idx = i;
        for (j = i+1; j < n; j++)
          if (arr[j] < arr[min_idx])
            min_idx = j;
        // Swap the found minimum element with the first element
        temp = arr[min_idx];
        arr[min_idx] = arr[i];
        arr[i] = temp;
    }
}
```

INSERTION SORT

```
void insertionSort(int arr[], int n)
{   int i, key, j;
    for (i = 1; i < n; i++) {
        key = arr[i];
        j = i - 1;
        /* Move elements of arr[0..i-1], that are
          greater than key, to one position ahead
          of their current position */
        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j = j - 1;
        }
        arr[j + 1] = key;
    }
}
```

QUICK SORT

```
/* This function takes last element as pivot, places the pivot element at its correct
position in sorted array, and places all smaller (smaller than pivot) to left of pivot
and all greater elements to right of pivot */
int partition (int arr[], int low, int high)
{    int pivot = arr[high];    // pivot
     int i = (low - 1);  // Index of smaller element
     for (int j = low; j <= high- 1; j++)
     {    // If current element is smaller than the pivot
          if (arr[j] < pivot)
          {    i++;    // increment index of smaller element
               swap(&arr[i], &arr[j]);
          }
     }
     swap(&arr[i + 1], &arr[high]);
     return (i + 1);
}
/* The main function that implements QuickSort
  arr[]  --> Array to be sorted,
  low    --> Starting index,
  high   --> Ending index */
void quickSort(int arr[], int low, int high)
{    if (low < high)
     {    /* pi is partitioning index, arr[p] is now at right place */
          int pi = partition(arr, low, high);
          // Separately sort elements before
          // partition and after partition
          quickSort(arr, low, pi - 1);
          quickSort(arr, pi + 1, high);
     }
}
```

MERGE SORT

```
// Merges two subarrays of arr[].
// First subarray is arr[l..m]
// Second subarray is arr[m+1..r]
void merge(int arr[], int l, int m, int r)
{   int i, j, k;
    int n1 = m - l + 1;
    int n2 =  r - m;
    /* create temp arrays */
    int L[n1], R[n2];
    /* Copy data to temp arrays L[] and R[] */
    for (i = 0; i < n1; i++)          L[i] = arr[l + i];
    for (j = 0; j < n2; j++)          R[j] = arr[m + 1+ j];
    /* Merge the temp arrays back into arr[l..r]*/
    i = 0; // Initial index of first subarray
    j = 0; // Initial index of second subarray
    k = l; // Initial index of merged subarray
    while (i < n1 && j < n2)
    {   if (L[i] <= R[j])          { arr[k] = L[i]; i++; }
        else                       { arr[k] = R[j]; j++; }
        k++;
    }
    /* Copy the remaining elements of L[], if there are any */
    while (i < n1)      {   arr[k] = L[i]; i++; k++; }
    /* Copy the remaining elements of R[], if there are any */
    while (j < n2)      {   arr[k] = R[j]; j++; k++; }
}
/* l is for left index and r is right index of the sub-array of arr to be sorted */
void mergeSort(int arr[], int l, int r)
{   if (l < r)
    {   // Same as (l+r)/2, but avoids overflow for
        // large l and h
        int m = l+(r-l)/2;
        // Sort first and second halves
        mergeSort(arr, l, m);
        mergeSort(arr, m+1, r);
        merge(arr, l, m, r);
    }
}
```

HEAP SORT

```
// To heapify a subtree rooted with node i which is
// an index in arr[]. n is size of heap
void heapify(int arr[], int n, int i)
{   int largest = i; // Initialize largest as root
    int l = 2*i + 1; // left = 2*i + 1
    int r = 2*i + 2; // right = 2*i + 2
    // If left child is larger than root
    if (l < n && arr[l] > arr[largest])        largest = l;
    // If right child is larger than largest so far
    if (r < n && arr[r] > arr[largest])        largest = r;
    // If largest is not root
    if (largest != i)
    {   swap(arr[i], arr[largest]);
        // Recursively heapify the affected sub-tree
        heapify(arr, n, largest);
    }
}
// main function to do heap sort
void heapSort(int arr[], int n)
{   // Build heap (rearrange array)
    for (int i = n / 2 - 1; i >= 0; i--)         heapify(arr, n, i);
    // One by one extract an element from heap
    for (int i=n-1; i>=0; i--)
    {   // Move current root to end
        swap(arr[0], arr[i]);
        // call max heapify on the reduced heap
        heapify(arr, i, 0);
    }
}
```

RADIX SORT

OR

BUCKET SORT

```
int getMax(int arr[], int n)
{   int mx = arr[0];
    for (int i = 1; i < n; i++)
        if (arr[i] > mx)
            mx = arr[i];
    return mx;
}
// A function to do counting sort of arr[] according to
// the digit represented by exp.
void countSort(int arr[], int n, int exp)
{   int output[n]; // output array
    int i, count[10] = {0};
    // Store count of occurrences in count[]
    for (i = 0; i < n; i++)
        count[ (arr[i]/exp)%10 ]++;
    // Change count[i] so that count[i] now contains actual
    //  position of this digit in output[]
    for (i = 1; i < 10; i++)          count[i] += count[i - 1];
    // Build the output array
    for (i = n - 1; i >= 0; i--)
    {   output[count[ (arr[i]/exp)%10 ] - 1] = arr[i];
        count[ (arr[i]/exp)%10 ]--;
    }
    // Copy the output array to arr[], so that arr[] now
    // contains sorted numbers according to current digit
    for (i = 0; i < n; i++)          arr[i] = output[i];
}
// The main function to that sorts arr[] of size n using
// Radix Sort
void radixsort(int arr[], int n)
{   // Find the maximum number to know number of digits
    int m = getMax(arr, n);
    // Do counting sort for every digit. Note that instead
    // of passing digit number, exp is passed. exp is 10^i
    // where i is current digit number
    for (int exp = 1; m/exp > 0; exp *= 10)        countSort(arr, n, exp);
}
```

LINEAR SEARCH

```
/*
      arr[] :     array
      n     :     size of array
      x     :     element to be searched
*/
int search(int arr[], int n, int x)
{
    int i;
    for (i = 0; i < n; i++)
        if (arr[i] == x)
            return i;
    return -1;
}
```

BINARY SEARCH

```
// A recursive binary search function. It returns
// location of x in given array arr[l..r] is present,
// otherwise -1
int binarySearch(int arr[], int l, int r, int x)
{   if (r >= l) {
        int mid = l + (r - l) / 2;
        // If the element is present at the middle
        // itself
        if (arr[mid] == x)       return mid;
        // If element is smaller than mid, then
        // it can only be present in left subarray
        if (arr[mid] > x)        return binarySearch(arr, l, mid - 1, x);
        // Else the element can only be present
        // in right subarray
        return binarySearch(arr, mid + 1, r, x);
    }
    // We reach here when element is not present in array
    return -1;
}
```

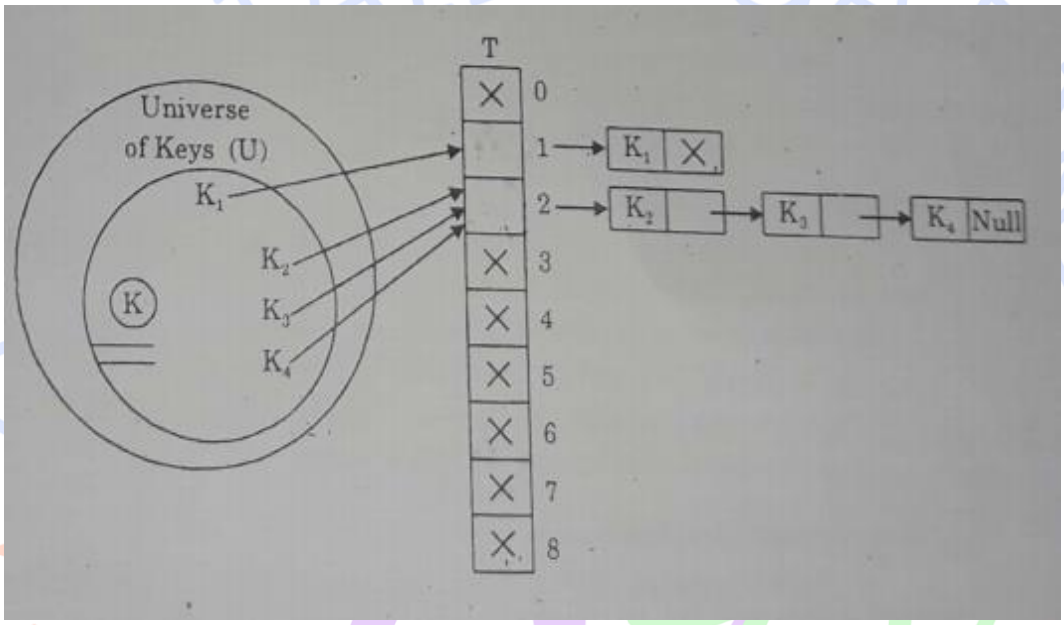
HASHING

DIVISION METHOD:

array	:	arr[]
value to be stored	:	x
size of array	:	n
index where element to be stored	:	x % n

PREVENTING COLLISION:

1. SEPARATE CHAINING



2. OPEN ADDRESSING

A. LINEAR PROBING

value to be stored : k
size of array : m
key : h(k, i)

$h(k, i) = [\ h'(k) + i\ ] \bmod m$
$i = 0 \ldots m-1$
$h'(k) = k \bmod m$

if h(k, i) is full, go for h(k, i+1) till an empty slot is found.

B. QUADRATIC PROBING

value to be stored : k
size of array : m
key : h(k, i)

$h(k, i) = [\ h'(k) + c_1 i + c_2 i^2\ ] \bmod m$
$i = 0 \ldots m-1$
$h'(k) = k \bmod m$

if h(k, i) is full, go for h(k, i+1) till an empty slot is found.

C. DOUBLE HASHING

value to be stored : k
size of array : m
key : h(k, i)

$h(k, i) = [h_1(k) + i*h_2(k)] \bmod m$
$i = 0 \ldots m-1$
$h_1(k) = k \bmod m$
$h_2(k) = k \bmod m'$ $m' = m-1$ or $m-2$

if h(k, i) is full, go for h(k, i+1) till an empty slot is found.

GRAPH

BREADTH FIRST SEARCH

uses : queue

FORMAT

queue : ...

parent : ...

DEPTH FIRST SEARCH

uses : stack

FORMAT

stack : ...