Thadomal Shahani Engineering College

Bandra (W.), Mumbai - 400050

ACADEMIC YEAR	2021-2022
ACADEMIC SESSION	JULY-DEC 2021
YEAR/SEMESTER	SE/III
COURSE CODE	CSL301
COURSE NAME	Data Structure Lab
ROLL NO	2003085
NAME OF STUDENT	Kasliwal Harsh Nitin
DATE OF SUBMISSION	16.12.21

Index

Experiment 1.

Aim: Implement stack ADT using array.

Theory:

Stack ADT: A list with the restriction that insertion and deletion can be performed only from one end. ~~called~~ according to the Last in First out (LIFO) principle.

1. PUSH: The push operation is used to insert an element into the stack at the topmost position of stack.

```
step 1 : If (TOP = MAX-1) {
           print "Overflow"          → {If this is the case, then stack
           Goto step 4 }               is full & no more insertions
step 2 : Set TOP = TOP+1.              can be done }
step 3 : Set STACK[TOP] = VALUE
step 4 : END.
```

2. POP: The pop operation is used to delete the topmost element from the stack.

```
step 1 : If TOP = NULL {              → {If this is the case, it means
           Print "UNDERFLOW"             the stack is empty & no
           Goto step 4 }                more deletions can be done}
step 2 : Set VAL = STACK[TOP]
step 3 : Set TOP = TOP-1.
step 4 : END.
```

3. PEEK : The peek operation returns the value of the topmost element of the stack without deleting it from the stack

```
Step 1 : If TOP = NULL                    { checks if the stack
           print "STACK IS EMPTY"           is empty }
           Goto step 3
Step 2 : Return STACK[TOP]
Step 3 : END.
```

4. isEmpty() : The isEmpty() operation is to test for whether or not stack is Empty.

```
step 1 : If TOP < 1
           return true
         else
           return false
step 2 : END.
```

5. isFull() : The isFull() operation is used to check is the stack is full or not.

```
Step 1 : If TOP = MAXSIZE
           return true
         else
           return false
step 2 : END.
```

6. Display : The Display operation is used to display any element from the stack

1. Display (TOP, i, a[i])
2. If TOP = 0 then
   Print "STACK EMPTY"
   exit
3. ELSE.
   For i = TOP to 0
   print a[i]
4. END.

→ If this the case, it means the stack is empty.

Conclusion : Stack ADT allows all data operations at one end only. At any time, we can only access the top element of a stack.

Program:

```
#include <iostream>
using namespace std;
int stack[10], n=10, top=-1;
void push(int val) {
  if(top>=n-1){
     cout<<"Stack Overflow"<<endl;
  }
  else {
    top++;
    stack[top]=val;
  }}
void pop() {
  if(top<=-1){
     cout<<"Stack Underflow"<<endl;
  }
  else {
    cout<<"The popped element is "<< stack[top] <<endl;
    top--;
  }}
void peek(){
   if(top<=-1){
      cout<<"Stack Underflow"<<endl;
   }
   else {
      cout<<"The top most element is : "<<stack[top]<<endl;
   }}
void size(){
   int count = 0;
    if(top<=-1){
```

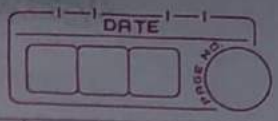
```
        cout<<"Stack Underflow"<<endl;
    }
    else {
        for (int i= top; i>=0; i--){
            count++;
        }
        cout<<"Size of the stack is:"<<count<<endl;
    }}

void display() {
   if(top>=0) {
      cout<<"Stack elements are:";
      for(int i=top; i>=0; i--)
      cout<<stack[i]<<" ";
      cout<<endl;
   } else
   cout<<"Stack is empty";
}
int main() {
   int ch, val;
   cout<<"1) Push in stack"<<endl;
   cout<<"2) Pop from stack"<<endl;
   cout<<"3) Peek in stack"<<endl;
   cout<<"4) Size of the element"<<endl;
   cout<<"5) Display stack"<<endl;
   cout<<"6) Exit"<<endl;
   do {
      cout<<"Enter choice: "<<endl;
      cin>>ch;
      switch(ch) {
```

```
        case 1: {
           cout<<"Enter value to be pushed:"<<endl;
           cin>>val;
           push(val);
           break;}
        case 2: {
           pop();
           break; }
        case 3: {
            peek();
            break;}
        case 4: {
            size();
            break;}
        case 5: {
            display();
            break;}
        case 6: {
           cout<<"Exit"<<endl;
           break; }
        default: {
           cout<<"Invalid Choice"<<endl;
        }
      }}
 while(ch!=6);
   return 0;
}
```

Output:

```
PS C:\Users\Harsh\OneDrive\Desktop\C++> cd "
1) Push in stack
2) Pop from stack
3) Peek in stack
4) Size of the element
5) Display stack
6) Exit
Enter choice:
1
Enter value to be pushed:
2
Enter choice:
2
The popped element is 2
Enter choice:
3
Stack Underflow
Enter choice:
4
Stack Underflow
Enter choice:
5
Stack is emptyEnter choice:
```

Harsh Kaliwal
85 / C21



Experiment 2

Aim: Convert an infix expression to postfix expression using stack ADT

Theory:

1)

Infix: For arithmetic expression such as x+y, 6*3, we know the variable 'x' is being added to the variable 'y'. Since the In this way in which operands surround the operator is called infix notation. E.g: 6*3, x+y etc.

Postfix: Postfix notation are also known as Reverse Polish Notation. They are different from infix and prefix notation in the sense that in the postfix notation, operator comes after the operands, e.g. xy+, xyz+* etc

Prefix: Also known as Polish notation. The name only suggests operator comes before the operands.
E.g: +xy, *+xyz etc.

2) Converting the following expression from infix to postfix.

5*3^2+(2*(7/11)^9)

5*23^+(2*(711/)^9)
5*23^+(2711/9^*)
523^*2711/9^*+

Harsh Kashiwal

Postfix to infix:

1 4 3 ^ - 7 9 1 ^ 5 / * + 2 -

1 (4 ^ 3) - 7 (9 ^ 1) 5 / * + 2 -
1 - (4 ^ 3) 7 (9 ^ 1 / 5) * + 2 -
1 - (4 ^ 3) 7 * (9 ^ 1 / 5) + 2 -
1 - (4 ^ 3) + 7 * (9 ^ 1 / 5) 2 -
1 - (4 ^ 3) + 7 * (9 ^ 1 / 5) + 2

=

2) Infix to Postfix (Algorithm).

Infix to Postfix (Exp)
{
1. Create an empty stack (s)
2. Create an empty string for storing result.
3. for i=0 to len(Exp)-1 {
// case-1:
4. if (Exp[i] is operand)
5. { result = result + E[i] }
// case-2:
6. Else if (E[i] is "(") {
7. Push (E[i]) }
// case 3:
8. Else if (E[i] is ")") {
9. while ((s != Empty) && (top != "(")) {
10. result = result + top.
11. POP() }
12. POP() }
// Case 4
13. Else {
write

Harsh Kaliwal
85

14. while ((S!=Empty) && (top!="(") && (top>=precedence(Exp[i])))
15. result = result + top
16. POP()
}
17. while (S!=Empty)
18. 17. Push(Exp[i]) }
19. while (S!=Empty) {
20. result = result + Top
POP() }.

4) Example showing each step using stack

Infix - A+B*C/(E-F)

	Input string	Output stack	Operator stack
0	A+B*C/(E-F)	A	
1	A+B*C/(E-F)	A	+
2	A+B*C/(E-F)	AB	+
3	A+B*C/(E-F)	AB	+*
4	-11-	ABC	+*
5	-11-	ABC*	+/
6	-11-	ABC*	+/(
7	-11-	ABC*E	+/(
8	-11-	ABC*E	+/(-
9	-11-	ABC*EF	+/(-
10	-11-	ABC*EF-	+/
11	A+B*C/(E-F)	ABC*EF-/+	

Harsh Kashiwar
C21/85

Conclusion :-

1) Infix expressions are human readable but not efficient for computer or machine reading.
2) Prefix and Postfix do not need the concept of precedence & associativity hence it becomes highly efficient to parse expressions in prefix or postfix formats.

Program:

```
#include <iostream>
#include <cstring>
using namespace std;
#define SIZE 100
char stack[SIZE];
int top = -1;
void push(char item)
{
    if (top >= SIZE - 1)
    {
        cout << "Stack Overflow." << endl;
    }
    else
    {
        top = top + 1;
        stack[top] = item;
    }
}

char pop()
{
    char item;

    if (top < 0)
    {
        cout << "stack under flow: invalid infix expression" << endl;
        getchar();
        exit(1);
    }
    else
    {
        item = stack[top];
        top = top - 1;
        return (item);
    }
}

int is_operator(char symbol)
{
    if (symbol == '^' || symbol == '*' || symbol == '/' || symbol == '+' || sy
mbol == '-')
    {
        return 1;
    }
    else
    {
        return 0;
```

```
    }
}

int precedence(char symbol)
{
    if (symbol == '^')
    {
        return (3);
    }
    else if (symbol == '*' || symbol == '/')
    {
        return (2);
    }
    else if (symbol == '+' || symbol == '-')
    {
        return (1);
    }
    else
    {
        return (0);
    }
}

void InfixToPostfix(char infix_exp[], char postfix_exp[])
{
    cout<<"Sr.\tStack\tPostfix"<<endl;
    cout<<"0\t(\t"<<endl;
    int i, j;
    char item;
    char x;

    push('(');
    strcat(infix_exp, ")");

    i = 0;
    j = 0;
    item = infix_exp[i];
    int counter =1;
    while (item != '\0')
    {
        if (item == '(')
        {
            push(item);
        }
        else if (isdigit(item) || isalpha(item))
        {
            postfix_exp[j] = item;
            j++;
```

```
            postfix_exp[j] = '\0';
        }
        else if (is_operator(item) == 1)
        {
            x = pop();
            while (is_operator(x) == 1 && precedence(x) >= precedence(item))
            {
                postfix_exp[j] = x;
                j++;
                postfix_exp[j] = '\0';
                x = pop();
            }
            push(x);

            push(item);
        }
        else if (item == ')')
        {
            x = pop();
            while (x != '(')
            {
                postfix_exp[j] = x;
                j++;
                postfix_exp[j] = '\0';
                x = pop();
            }
        }
        else
        {
            cout << "Invalid infix Expression." << endl;
            getchar();
            exit(1);
        }
        i++;
        cout<<counter<<"\t"<<stack<<"\t"<<postfix_exp<<"\t"<<endl;
        counter++;
        item = infix_exp[i];
    }
    if (top > 0)
    {
        cout << "Invalid infix Expression." << endl;
        getchar();
        exit(1);
    }


    postfix_exp[j] = '\0';
}
```

```
int main()
{
    char infix[SIZE], postfix[SIZE];
    cout << "ASSUMPTION: The infix expression contains single letter variables
 and single digit constants only." << endl;
    cout << "Enter Infix expression : " << endl;
    gets(infix);

    InfixToPostfix(infix, postfix);
    cout << "Postfix Expression: " << endl;
    puts(postfix);

    return 0;
}
```

Output:

```
PS C:\Users\Harsh\OneDrive\Desktop\C++> cd "c:\Users\Harsh\OneDrive\Desktop\C++\" ; if ($?) { g++ D
ASSUMPTION: The infix expression contains single letter variables and single digit constants only.
Enter Infix expression :
A+B*C/(E-F)
Sr.     Stack   Postfix
0       (
1       (       A
2       (+      A
3       (+      AB
4       (+*     AB
5       (+*     ABC
6       (+/     ABC*
7       (+/(    ABC*
8       (+/(    ABC*E
9       (+/(-   ABC*E
10      (+/(-   ABC*EF
11      (+/(-   ABC*EF-
12      (+/(-   ABC*EF-/+
Postfix Expression:
ABC*EF-/+
PS C:\Users\Harsh\OneDrive\Desktop\C++>
```

Harsh Kathuria
2003055
C21

DATE 22/09/21

Experiment 3

Aim: Evaluate Postfix Expression using Stack ADT.

Theory:

Algorithm:

```
Evaluate Postfix(E)
{
1. Create an Empty stack(S)
2. for i=0 to len(E)-1
   {
3. Case 1:
     if (E[i] is operand)    // if operand is encountered, push it onto
4.   {push(E[i])}                 stack.

5. case 2:
     Else if (E[i] is operator)  // if operator is encountered, pop2
   {                                 elements.
6.   op2 = pop()      // Top element
7.   op1 = pop()      // Next to top element.
8.   res = perform(op1, op2, E[i])   // result = op1 operator op2
9.   push(res)
     }
   }  // End of for.
10. return top of stack.
  }
```

Harsh Kashiwal
2003035

Example. 6821 +-684 /+*

Token	Action	Stack
-	-	Empty.
6	push	6
2	push	6,2
1	push	6,2,1
+	Pop 1,2 ; 1+2=3 push (3)	6,3
-	Pop 3,6 ; 6-3=3 push (3)	3
6	push	3,6
8	push	3,6,8
4	push	3,6,8,4
/	Pop 4,8 ; 8/4=2 push (2)	3,6,2
+	Pop 2,6 ; 6+2=8 pus(8)	3,8
*	Pop 8,3 ; 3*8=24 push (24)	24

Result = 24

Conclusion :- The expressions written in postfix form are evaluated faster compared to infix notation as parenthesis are not required in postfix.

PROGRAM: Write a program to implement postfix evaluation using stack

```
#include <iostream>
#include <string.h>
using namespace std;
#define SIZE 100

int stack[SIZE];
int Top = -1;
void push(char Item)
{
    if (Top >= SIZE - 1)
    {
        cout << "\nStack Overflow.";
    }
    else
    {
        Top = Top + 1;
        stack[Top] = Item;
    }
}

void push1(int y)
{
    stack[++Top] = y;
}

void get_stack()
{
    int i;
    for (i = 0; i <= Top; i++)
    {
        cout << stack[i];
        cout << " ";
    }
}

char pop()
{
    char Item;
    if (Top < 0)
    {
        cout << "Stack Under Flow: Invalid Infix Expression";
        getchar();
        exit(1);
    }
    else
    {
        Item = stack[Top];
```

```
        Top = Top - 1;
        return (Item);
    }
}
int pop1()
{
    return stack[Top--];
}
int is_operator(char symbol)
{
    if (symbol == '^' || symbol == '*' || symbol == '/' || symbol == '+' || sy
mbol == '-')
    {
        return 1;
    }
    else
    {
        return 0;
    }
}

int precedence(char symbol)
{
    if (symbol == '^')
    {
        return (3);
    }
    else if (symbol == '*' || symbol == '/')
    {
        return (2);
    }
    else if (symbol == '+' || symbol == '-')
    {
        return (1);
    }
    else
    {
        return (0);
    }
}

void InfixToPostfix(char infix_exp[], char postfix_exp[])
{
    int i, k;
    char Item;
    char x;

    push('(');
```

```
    strcat(infix_exp, ")");

    i = 0;
    k = 0;
    Item = infix_exp[i];
    while (Item != '\0')
    {
        if (Item == '(')
        {
            push(Item);
        }
        else if (isalnum(Item))
        {
            postfix_exp[k] = Item;
            k++;
        }
        else if (is_operator(Item) == 1)
        {
            x = pop();
            while (is_operator(x) == 1 && precedence(x) >= precedence(Item))
            {
                postfix_exp[k] = x;
                k++;
                x = pop();
            }
            push(x);

            push(Item);
        }
        else if (Item == ')')
        {
            x = pop();
            while (x != '(')
            {
                postfix_exp[k] = x;
                k++;
                x = pop();
            }
        }
        else
        {
            cout << "\nInvalid Infix Expression.\n";
            getchar();
            exit(1);
        }
        i++;

        Item = infix_exp[i];
```

```
    }
    if (Top > 0)
    {
        cout << "\nInvalid Infix Expression.\n";
        getchar();
        exit(1);
    }
    postfix_exp[k] = '\0';
}

int main()
{
    int i, ch;
    char exp[SIZE];
    char *a;
    int n1, n2, n3, num;
    char infix[SIZE], postfix[SIZE];
    cout << "You can enter infix or postfix expression, choose an option\n";
    cout << "1. Infix expression\n2. Postfix Expression\n\nEnter an Option:
 ";
    cin >> ch;
    switch (ch)
    {
    case 1:
        for (i = 0; i < SIZE; i++)
        {
            postfix[i] = '\0';
        }
        cout << "\nYou have chosen 1, Enter an infix expression:    ";
        cin >> infix;
        InfixToPostfix(infix, postfix);
        cout << "\n";
        cout << "Resultant postfix expression:    ";
        puts(postfix);
        a = postfix;
        break;
    case 2:
        cout << "Enter Postfix Expression : " << endl;
        cin >> postfix;
        a = postfix;
        break;
    }

    cout << "\nToken\tStack\n";
    char token;
    while (*a != '\0')
    {
        if (isdigit(*a))
```

```
        {
            num = *a - '0';
            token = *a;
            push1(num);
        }
        else
        {
            n1 = pop1();
            n2 = pop1();
            switch (*a)
            {
            case '+':
            {
                n3 = n1 + n2;
                token = '+';
                break;
            }
            case '-':
            {
                n3 = n2 - n1;
                token = '-';
                break;
            }
            case '*':
            {
                n3 = n1 * n2;
                token = '*';
                break;
            }
            case '/':
            {
                n3 = n2 / n1;
                token = '/';
                break;
            }
            }
            push1(n3);
        }
        cout << " \n " << token << "\t";
        get_stack();
        cout << " \n ";

        a++;
    }
    cout << "Final Result: " << pop1();
    return 0;
}
```

OUTPUT:

```
PS C:\Users\Harsh\OneDrive\Desktop\DS\CODES> cd "c:\Users\Harsh\OneDrive\Desktop\DS\CODES\"
 { .\Evaluation }
You can enter infix or postfix expression, choose an option
1. Infix expression
2. Postfix Expression

Enter an Option:    1

You have chosen 1, Enter an infix expression:      (5+4)*2

Resultant postfix expression:    54+2*

Token   Stack

 5      5

 4      5 4

 +      9

 2      9 2

 *      18
 Final Result: 18
PS C:\Users\Harsh\OneDrive\Desktop\DS\CODES>
```

Harsh Kaliwal
2003055
C21

Experiment 4.

Aim: Implement linear Queue ADT using array.

Theory:

Queue ADT: Queue data structure is a collection of similar data items in which insertion and deletion operations are performed based on FIFO principle.

1. enqueue(): Adds a new item to the rear of the queue. It needs the item and returns nothing.

```
Enqueue(val){
    if (isFull()){
        return }
    else if (isEmpty())
        front=rear=0
    else rear=rear+1
        queue[rear]=val
```

2. dequeue(): Removes the front item from the queue. It needs no parameters and returns the item.

```
dequeue(){
    if (isEmpty()){
        return }
    else
        val = queue[front]
    if (front==rear){
        front=rear=-1}
    else front=front+1
    return val; }
```

Harsh Kashliwal
2003055

3. Front() : An operation that returns the value of the front element of the queue without deleting it from the front.

```
get-front(){
 if(isEmpty())
    return.
 else return queue[front].
}.
```

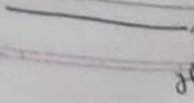
4. isEmpty() : It tests to see whether the queue is empty. It needs no parameters and returns a boolean value.

```
isEmpty (){
 if (front==-1 && rear==-1)
      return True
 else
      return false.
```

5. isFull() : It tests to see whether the queue is full. It needs no parameters and returns a boolean value.

```
isfull (){
 if (rear==Max-1)
    return True.
 else
    return False.
```

Conclusion : Push and pop operation takes place from a different end of the queue.
Based on FIFO (first in first out).



PROGRAM: Write a menu driven code to implement QUEUE ADT using arrays.

```
#include <iostream>
using namespace std;
#define n 100
int queue[n];
int size = 0;
int rear = - 1;
int front = -1;
void enqueue() {
   int val;
   if (rear == n - 1)
   cout<<"Queue Overflow"<<endl;
   else {
      if (front == - 1)
      front = 0;
      cout<<"Insert the element in queue : "<<endl;
      cin>>val;
      rear++;
      size++;
      queue[rear] = val;
   }
}
void dequeue(){
   if (front == - 1 || front > rear) {
      cout<<"Queue Underflow ";
      return ;
   } else {
      cout<<"Element deleted from queue is : "<< queue[front] <<endl;
      front++;
      size--;
   }
}
void get_front(){
    if (front == - 1 && rear == -1)
    cout<<"Queue is Empty"<<endl;
    else{
        cout<<"Front = "<<queue[front]<<endl;
    }
}
void get_rear(){
    if (front == - 1 && rear == -1)
    cout<<"Queue is Empty"<<endl;
    else{
        cout<<"Rear = "<<queue[rear]<<endl;
    }
}
```

```
void Display() {
   if (front == - 1)
   cout<<"Queue is empty"<<endl;
   else {
      cout<<"Queue elements are : ";
      for (int i = front; i <= rear; i++)
      cout<<queue[i]<<" ";
         cout<<endl;
   }
}
int main() {
   int ch;
   cout<<"1) ENQUEUE "<<endl;
   cout<<"2) DEQUEUE"<<endl;
   cout<<"3) GET FRONT"<<endl;
   cout<<"4) GET REAR"<<endl;
   cout<<"5) SIZE"<<endl;
   cout<<"6) DISPLAY"<<endl;
   cout<<"7) EXIT"<<endl;
   do {
      cout<<"Enter your choice : "<<endl;
      cin>>ch;
      switch (ch) {
         case 1: enqueue();
         break;
         case 2: dequeue();
         break;
         case 3: get_front();
         break;
         case 4: get_rear();
         break;
         case 5:  cout<<"Size = "<<size;
                  cout<<endl;
         break;
         case 6:  Display();
         break;
         case 7: cout<<"Exit"<<endl;
         break;
         default: cout<<"Invalid choice"<<endl;
      }
   } while(ch!=7);
   return 0;
}
```

OUTPUT:

```
PS C:\Users\Harsh\OneDrive\Desktop\DS\CODES> cd "c:\Users\Harsh\O
File } ; if ($?) { .\tempCodeRunnerFile }
1) ENQUEUE
2) DEQUEUE
3) GET FRONT
4) GET REAR
5) SIZE
6) DISPLAY
7) EXIT
Enter your choice :
1
Insert the element in queue :
122
Enter your choice :
5
Size = 1
Enter your choice :
6
Queue elements are : 122
Enter your choice :
```

Harsh Kashiwal
2003085

DATE 02 10 21

Experiment 5

Aim: Implement Circular Queue ADT using array.

Theory:

Limitation of Linear Queue

In linear queue, once an element is deleted, we cannot insert another element in its position. This disadvantage of a linear queue is overcome by a circular queue, thus saving memory.
The circular queue connects the last node of a queue to its first by forming a circular link & because of this it resolves the memory wastage problem.

```
1. enqueue(val) {
    if( front==0 and rear=n-1) || (front==rear+1)
        print Queue is full.
    else
        if(front==-1)
            then front=0
        if (front!=0 and rear=n-1
            rear=-1
        rear++
        queue[rear] = val
    }
```

HK
085

2. dequeue(){
```
if (front==-1 && rear==-1)
 print Empty.
else
 if(front==rear)
    queue[rear]
    front=rear=-1
 else if(front=n-1)
    queue[front]
    front 0
 else
    queue[front]
    front++
}
```

3. display(){
```
if(front==-1 and rear==-1) => Empty.
else if(front<=rear)
  for(i=front, i<=rear)
      queue[i]
else{
    for(i=front- i<=n-1)
        queue[i]
    for(i=0; i<=rear)
        queue[i]}
}
```

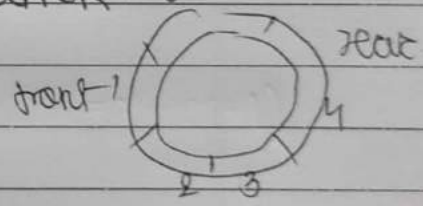
Harsh Hasija
2003085

4. get_front(){
        queue[front]
    }

5. ~~get_rear(){~~
5. isEmpty(){
        if(front==-1 and rear==-1)
        return True

6. isfull(){
        if(front==0 and rear==n-1) || (front==rear+1)
        return True
    }.

Conclusion: Circular queue resolves the memory wastage problem by forming a circular link.



PROGRAM:

Write a menu driven code to implement CIRCULAR QUEUEADT using arrays.

Code:

```
#include <iostream>
using namespace std;
#define n 3
int queue[n];
int size = 0;
int rear = - 1;
int front = -1;

void enqueue(){
   int val;
    if(((front==0)&&(rear==n-1)) || (front==rear+1))
    cout<<"Queue iS FULL";
   else{
   if(front == - 1 )
      front = 0;
   if(front!=0 && (rear==n-1))
       rear=-1;
      cout<<"Insert the element in queue : "<<endl;
      cin>>val;
      rear++;
      queue[rear] = val;
      size++;
   }
}

void dequeue(){
   if (front == - 1 && rear == -1) {
      cout<<"Queue Underflow ";
   }
   else {
   if(front==rear){
       cout<<"1.Element deleted from queue is : "<< queue[front] <<endl;
       front=rear=-1;
       size--;
   }
   else if(front==n-1){
    cout<<"2.Element deleted from queue is : "<< queue[front] <<endl;
    front=0;
```

```
    size--;
    }
    else{
       cout<<"3.Element deleted from queue is : "<< queue[front] <<endl;
       front++;
       size--;
   }
}
}
void display(){
   if (front == - 1 && rear == -1)
   cout<<"Queue is Empty"<<endl;
   else if (front <=rear){
       for(int i=front; i<=rear; i++)
        cout<<queue[i]<<" ";
   }
   else{
       for(int i=front; i<=n-1; i++)
       cout<<queue[i]<<" ";
       for(int i=0; i<=rear; i++)
       cout<<queue[i]<<" ";
   }
   cout<<endl;
}
void get_front(){
    if (front == - 1 && rear == -1)
    cout<<"Queue is Empty"<<endl;
    else{
        cout<<"Your front element is: "<<queue[front]<<endl;
    }
}
void get_rear(){
    if (front == - 1 && rear == -1)
    cout<<"Queue is Empty"<<endl;
    else{
        cout<<"Your rear element is: "<<queue[rear]<<endl;
    }
}
int main() {
   int ch;
   cout<<"1) ENQUEUE "<<endl;
   cout<<"2) DEQUEUE"<<endl;
   cout<<"3) GET FRONT"<<endl;
   cout<<"4) GET REAR"<<endl;
   cout<<"5) SIZE"<<endl;
   cout<<"6) DISPLAY"<<endl;
   cout<<"7) EXIT"<<endl;
   do {
```

```
      cout<<"Enter your choice : "<<endl;
      cin>>ch;
      switch (ch) {
         case 1: enqueue();
         break;
         case 2: dequeue();
         break;
         case 3: get_front();
         break;
         case 4: get_rear();
         break;
         case 5:  cout<<"No. of elements in queue = "<<size;
                  cout<<endl;
         break;
         case 6:  display();
         break;
         case 7: cout<<"Exit"<<endl;
         break;
         default: cout<<"Invalid choice"<<endl;
      }
   } while(ch!=7);
   return 0;
}
```

OUTPUT:

```
PS D:\Harsh\SEM 3\DS\CODES> cd "d:\Harsh\SEM 3\DS\CODES\" ; if (
1) ENQUEUE
2) DEQUEUE
3) GET FRONT
4) GET REAR
5) SIZE
6) DISPLAY
7) EXIT
Enter your choice :
1
Insert the element in queue :
12
Enter your choice :
1
Insert the element in queue :
13
Enter your choice :
1
Insert the element in queue :
14
Enter your choice :
1
Queue iS FULL
Enter your choice :
6
12 13 14
Enter your choice :
2
3.Element deleted from queue is : 12
Enter your choice :
6
13 14
Enter your choice :
1
Insert the element in queue :
15
Enter your choice :
6
13 14 15
Enter your choice :
5
No. of elements in queue = 3
Enter your choice :
7
Exit
PS D:\Harsh\SEM 3\DS\CODES>
```

Harsh Kashikaal
2003085
C21

Experiment 6

Aim: Implement Singly linked list

Theory:

Linked list can be defined as collection of objects called nodes that are randomly stored in the memory. A node contains two fields

→ Data stored at the address.

→ Pointer which contains the address of the next node in the memory.

The last node of the list contains pointer to the null.

→ Differentiate b/w Array & linked list.

	Array	Linked list.
1	Random Access (Fast search)	Fast insertion/deletion time.
2	less memory needed	Dynamic size.
3	Better cache locality	Efficient memory allocation utilization
4	Slow insertion/deletion time	Slow search time
5	Fixed size	More memory needed per node as addition of storage required for pointers.
6	Inefficient memory allocation/utilization	

Harsh
2003085

→ Different operations:

1) Insert in beginning: Inserting element at the front of the list

Algorithm: insert_beg (val){
1. Create a new node
2. new node → data = val
3. new node → next = head
4. head = new node
}

2) Insert at End: Insertion at the last of the linked list

Algorithm: insert at End (val){
1. Create a new node
2. new node → data = val
3. new node → next = null
4. if (head == null)
head = new node
}

3) Insert before a given node: Insertion before the specified node of the linked list

Algorithm: insert_bef (num, val){
Create new node
new nod → data = val
temp1 = temp2 = head
while (true){
if (temp2 → data + 1 == val){
temp 2 = temp1 ; break
temp 1 = tem1 → next }
new node → next = temp1 , temp2 → next = newnode

Harsh
2003035

4> Delete from beginning: Deletation of a node from the start of the list

Algorithm:
```
Del-beg( ){
if (head == null)
    print (underflow)
      return.
temp = head
head = head → next
free (temp)}
```

5> Delete from the End: Deletes the last node of the list.

Algorithm:
```
Del_End ( ){
if (head == null)
  print (underflow)
else if (head → next == null)
    temp = head.
    head = null
    free (temp) }.
else {
    temp1 = temp2 = head.
  while (temp1 → next != null){
      temp2 = temp1
      temp1 = temp1 → next. }
   temp2 → next = null
   free (temp)
}.
```

Harsh
2003085

6) Delete node before a specified location: Deletion of a node before the specified node in the list.

Algorithm:
```
Del-specif (num){
  if (head = null){
  print (underflow)
    return};
  temp1 = temp2 = head.
  while (temp2 → next → data != num){
    temp2 = temp1
    temp1 = temp1 → next}.
  temp2 → next = temp1 → next
  free (temp1)
}.
```

7) Forward traversal: Displaying the contents is very easy.

Algorithm:
```
traversal(){
if (head == null){
  print empty
  return}
temp = head
while (temp != null){
  print (temp → data)
  temp = temp → next
}
}.
```

8) Backward traversal: Displaying the content in reverse order

Algorithm: Backtraversal (head) {
1. if (head == null)
return.
2. backtraversal (head → next)
3. print (head → data);
}

9) Sorting: Sorting the data of the list using bubble sort.

Algorithm: sort (head) {
temp1 = head
while (temp1 → next != null) {
temp2 = temp1 → next;
while (temp2 != null) {
if (temp1 → data > temp2 → data) {
swap = temp1 → data
temp1 = temp2 → data.
temp2 = data == swap. }
temp2 = temp2 → next
}
temp1 = temp1 → next
}.

10) No. of nodes: Counts the number of nodes present in the list

Algorithm: no. of nodes (start) {
if (head == null)
print (0) }

Harsh Kaliwal.
2003085

```
else {
  int count;
  temp = head
 while ( temp != null)
  { count++
     Temp = Temp → next
  }
```

11> Search an Element: Searching a specified element in the list.

Algorithm:
```
Search(val){
   if (head == null)
    print ( empty)
   temp = head
   while (temp != null) {
    if (temp → data == val){
      print ( Element found!)
       break }
   else
       temp → temp → next.
  } }
```

Conclusion: Linked lists are used because of their efficient insertion and deletion. Linked lists are more efficient than arrays in memory allocation. It can grow or shrink at runtime by allocating & deallocating memory.

PROGRAM:

Write a menu driven code to implement Singly LinkedList.

Code:

```
#include <iostream>
#include <conio.h>
using namespace std;
struct node
{
    int data;
    struct node *next;
};
int flag = 0;
int count = 0;
struct node *start = NULL;
struct node *list(struct node *);
struct node *display(struct node *);
struct node *insert_beg(struct node *);
struct node *insert_end(struct node *);
struct node *insert_before(struct node *);
struct node *insert_after(struct node *);
struct node *delete_beg(struct node *);
struct node *delete_end(struct node *);
struct node *delete_node(struct node *);
struct node *delete_after(struct node *);
struct node *delete_list(struct node *);
struct node *sort_list(struct node *);
struct node *backtraversal(struct node *);
struct node *no_of_nodes(struct node *);
struct node *search(struct node *);
struct node *del_specif(struct node *);
int main(int argc, char *argv[])
{
    int option;
    do
    {
        cout << "\n1.LIST";
        cout << "\n2.INSERT IN BEGINNING";
        cout << "\n3.INSERT AT END";
        cout << "\n4.INSERT BEFORE A GIVEN NODE";
        cout << "\n5.DELETE FROM BEGINNING";
        cout << "\n6.DELETE FROM END";
        cout << "\n7.DELETE NODE BEFORE A SPECIFIED LOCATION ";
        cout << "\n8.FORWARD TRAVERSAL";
        cout << "\n9.BACKWARD TRAVERSAL";
        cout << "\n10.SORTING";
        cout << "\n11.COUNT NUMBER OF NODES";
```

```
        cout << "\n12.SEARCH AN ELEMENT";
        cout << "\n13.EXIT";
        cout << "\n\nENTER YOUR OPTION : ";
        cin >> option;
        switch (option)
        {
        case 1:
            start = list(start);
            cout << "\n--LINKED LIST CREATED--";
            break;
        case 2:
            start = insert_beg(start);
            break;
        case 3:
            start = insert_end(start);
            break;
        case 4:
            start = insert_before(start);
            break;
        case 5:
            start = delete_beg(start);
            break;
        case 6:
            start = delete_end(start);
            break;
        case 7:
            start = del_specif(start);
            break;
        case 8:
            start = display(start);
            break;
        case 9:
            start = backtraversal(start);
            break;
        case 10:
            start = sort_list(start);
            break;
        case 11:
            start = no_of_nodes(start);
            cout << "\nNUMBER OF NODES ARE : " << count;
            break;
        case 12:
            start = search(start);
            break;
        }
    } while (option != 13);
    getch();
    return 0;
```

```
}
struct node *list(struct node *start)
{
    struct node *new_node, *ptr;
    int num;
    cout << "ENTER -1 TO END!"<<endl;
    cout << "ENTER THE DATA :";
    cin >> num;
    while (num != -1)
    {
        new_node = (struct node *)malloc(sizeof(struct node));
        new_node->data = num;
        if (start == NULL)
        {
            new_node->next = NULL;
            start = new_node;
        }
        else
        {
            ptr = start;
            while (ptr->next != NULL)
                ptr = ptr->next;
            ptr->next = new_node;
            new_node->next = NULL;
        }
        cout << "\n ENTER THE DATA : ";
        cin >> num;
    }
    return start;
}
struct node *display(struct node *start)
{
    if (flag == 1)
    {
        cout << "\n\n -- EMPTY LIST -- \n\n";
    }
    struct node *ptr;
    ptr = start;
    while (ptr != NULL)
    {
        cout << "\t "<< ptr->data;
        ptr = ptr->next;
    }
    return start;
}
struct node *search(struct node *start)
{
    int num;
```

```
    int flag = 0;
    cout << "ENTERR THE ELEMEMENT TO BE SEARCHED :";
    cin >> num;
    struct node *ptr;
    ptr = start;
    while (ptr != NULL)
    {
        if (ptr->data == num)
        {
            cout << "\n\nNUMBER IS FOUND!\n\n";
            flag = 1;
            break;
        }
        ptr = ptr->next;
    }
    if (flag != 1)
    {
        cout << "\n\nNUMBER NOT FOUND! :(\n\n";
    }
    return start;
}
struct node *no_of_nodes(struct node *start)
{
    struct node *ptr;
    ptr = start;
    while (ptr != NULL)
    {
        count = count + 1;
        ptr = ptr->next;
    }
    return start;
}
struct node *backtraversal(struct node *start)
{
    struct node *prev = NULL;
    struct node *current = start;
    struct node *nextt = NULL;
    while (current != NULL)
    {
        nextt = current->next;
        current->next = prev;
        prev = current;
        current = nextt;
    }
    start = prev;
};
struct node *insert_beg(struct node *start)
{
```

```
    struct node *new_node;
    int num;
    cout << "\n ENTER THE DATA : ";
    cin >> num;
    new_node = (struct node *)malloc(sizeof(struct node));
    new_node->data = num;
    new_node->next = start;
    start = new_node;
    return start;
}
struct node *insert_end(struct node *start)
{
    struct node *ptr, *new_node;
    int num;
    cout << "\n ENTER THE DATA ";
    cin >> num;
    new_node = (struct node *)malloc(sizeof(struct node));
    new_node->data = num;
    new_node->next = NULL;
    ptr = start;
    while (ptr->next != NULL)
        ptr = ptr->next;
    ptr->next = new_node;
    return start;
}
struct node *insert_before(struct node *start)
{
    struct node *new_node, *ptr, *preptr;
    int num, value;
    cout << "ENTER THE DATA ";
    cin >> num;
    cout << "ENTER THE VALUE BEFORE WHICH THE DATA IS TO BE INSERTED ";
    cin >> value;
    new_node = (struct node *)malloc(sizeof(struct node));
    new_node->data = num;
    ptr = start;
    while (ptr->data != value)
    {
        preptr = ptr;
        ptr = ptr->next;
    }
    preptr->next = new_node;
    new_node->next = ptr;
    return start;
}
struct node *delete_beg(struct node *start)
{
    struct node *ptr;
```

```
    ptr = start;
    start = start->next;
    free(ptr);
    return start;
}
struct node *delete_end(struct node *start)
{
    struct node *ptr, *preptr;
    ptr = start;
    while (ptr->next != NULL)
    {
        preptr = ptr;
        ptr = ptr->next;
    }
    preptr->next = NULL;
    free(ptr);
    return start;
}
struct node *delete_node(struct node *start)
{
    struct node *ptr, *preptr;
    int val;
    cout << "ENTER THE VALUE TO BE DELETED";
    cin >> val;
    ptr = start;
    if (ptr->data == val)
    {
        start = delete_beg(start);
        return start;
    }
    else
    {
        while (ptr->data != val)
        {
            preptr = ptr;
            ptr = ptr->next;
        }
        preptr->next = ptr->next;
        free(ptr);
        return start;
    }
}
struct node *del_specif(struct node *start)
{
    start = no_of_nodes(start);
    int loc;
    struct node *ptr, *preptr;
    ptr = start;
```

```
    cout << "ENTER THE LOCATION ";
    cin >> loc;
    while (ptr->next != NULL)
    {
        if (loc + 1 == count)
        {
            preptr = ptr;
        }
        ptr = ptr->next;
    }
    preptr->next = ptr->next;
    free(ptr);
    return start;
}
struct node *sort_list(struct node *start)
{
    struct node *ptr1, *ptr2;
    int temp;
    ptr1 = start;
    while (ptr1->next != NULL)
    {
        ptr2 = ptr1->next;
        while (ptr2 != NULL)
        {
            if (ptr1->data > ptr2->data)
            {
                temp = ptr1->data;
                ptr1->data = ptr2->data;
                ptr2->data = temp;
            }
            ptr2 = ptr2->next;
        }
        ptr1 = ptr1->next;
    }
    return start;
}
```

OUTPUT:

```
PS D:\Harsh\SEM 3\DS\CODES> cd "d:\Harsh\SEM 3\DS\CODES\" ; if ($?) { g++ SinglyList.cpp

1.LIST
2.INSERT IN BEGINNING
3.INSERT AT END
4.INSERT BEFORE A GIVEN NODE
5.DELETE FROM BEGINNING
6.DELETE FROM END
7.DELETE NODE BEFORE A SPECIFIED LOCATION
8.FORWARD TRAVERSAL
9.BACKWARD TRAVERSAL
10.SORTING
11.COUNT NUMBER OF NODES
12.SEARCH AN ELEMENT
13.EXIT

ENTER YOUR OPTION : 1
ENTER -1 TO END!
ENTER THE DATA :12

 ENTER THE DATA : 13

 ENTER THE DATA : 14

 ENTER THE DATA : 15

 ENTER THE DATA : 16

 ENTER THE DATA : -1

--LINKED LIST CREATED--
1.LIST
2.INSERT IN BEGINNING
3.INSERT AT END
4.INSERT BEFORE A GIVEN NODE
5.DELETE FROM BEGINNING
6.DELETE FROM END
7.DELETE NODE BEFORE A SPECIFIED LOCATION
8.FORWARD TRAVERSAL
9.BACKWARD TRAVERSAL
10.SORTING
11.COUNT NUMBER OF NODES
12.SEARCH AN ELEMENT
13.EXIT

ENTER YOUR OPTION : 2
```

```
 ENTER THE DATA : 18

1.LIST
2.INSERT IN BEGINNING
3.INSERT AT END
4.INSERT BEFORE A GIVEN NODE
5.DELETE FROM BEGINNING
6.DELETE FROM END
7.DELETE NODE BEFORE A SPECIFIED LOCATION
8.FORWARD TRAVERSAL
9.BACKWARD TRAVERSAL
10.SORTING
11.COUNT NUMBER OF NODES
12.SEARCH AN ELEMENT
13.EXIT

ENTER YOUR OPTION : 3

 ENTER THE DATA 19

1.LIST
2.INSERT IN BEGINNING
3.INSERT AT END
4.INSERT BEFORE A GIVEN NODE
5.DELETE FROM BEGINNING
6.DELETE FROM END
7.DELETE NODE BEFORE A SPECIFIED LOCATION
8.FORWARD TRAVERSAL
9.BACKWARD TRAVERSAL
10.SORTING
11.COUNT NUMBER OF NODES
12.SEARCH AN ELEMENT
13.EXIT

ENTER YOUR OPTION : 4
ENTER THE DATA 20
ENTER THE VALUE BEFORE WHICH THE DATA IS TO BE INSERTED 15

1.LIST
2.INSERT IN BEGINNING
3.INSERT AT END
4.INSERT BEFORE A GIVEN NODE
5.DELETE FROM BEGINNING
6.DELETE FROM END
7.DELETE NODE BEFORE A SPECIFIED LOCATION
8.FORWARD TRAVERSAL
9.BACKWARD TRAVERSAL
```

```
10.SORTING
11.COUNT NUMBER OF NODES
12.SEARCH AN ELEMENT
13.EXIT

ENTER YOUR OPTION : 8
         18      12      13      14      20      15      16      19
1.LIST
2.INSERT IN BEGINNING
3.INSERT AT END
4.INSERT BEFORE A GIVEN NODE
5.DELETE FROM BEGINNING
6.DELETE FROM END
7.DELETE NODE BEFORE A SPECIFIED LOCATION
8.FORWARD TRAVERSAL
9.BACKWARD TRAVERSAL
10.SORTING
11.COUNT NUMBER OF NODES
12.SEARCH AN ELEMENT
13.EXIT

ENTER YOUR OPTION : 5

1.LIST
2.INSERT IN BEGINNING
3.INSERT AT END
4.INSERT BEFORE A GIVEN NODE
5.DELETE FROM BEGINNING
6.DELETE FROM END
7.DELETE NODE BEFORE A SPECIFIED LOCATION
8.FORWARD TRAVERSAL
9.BACKWARD TRAVERSAL
10.SORTING
11.COUNT NUMBER OF NODES
12.SEARCH AN ELEMENT
13.EXIT

ENTER YOUR OPTION : 6

1.LIST
2.INSERT IN BEGINNING
3.INSERT AT END
4.INSERT BEFORE A GIVEN NODE
5.DELETE FROM BEGINNING
6.DELETE FROM END
7.DELETE NODE BEFORE A SPECIFIED LOCATION
8.FORWARD TRAVERSAL
```

```
9.BACKWARD TRAVERSAL
10.SORTING
11.COUNT NUMBER OF NODES
12.SEARCH AN ELEMENT
13.EXIT

ENTER YOUR OPTION : 8
         12      13      14      20      15      16
1.LIST
2.INSERT IN BEGINNING
3.INSERT AT END
4.INSERT BEFORE A GIVEN NODE
5.DELETE FROM BEGINNING
6.DELETE FROM END
7.DELETE NODE BEFORE A SPECIFIED LOCATION
8.FORWARD TRAVERSAL
9.BACKWARD TRAVERSAL
10.SORTING
11.COUNT NUMBER OF NODES
12.SEARCH AN ELEMENT
13.EXIT

ENTER YOUR OPTION : 9

1.LIST
2.INSERT IN BEGINNING
3.INSERT AT END
4.INSERT BEFORE A GIVEN NODE
5.DELETE FROM BEGINNING
6.DELETE FROM END
7.DELETE NODE BEFORE A SPECIFIED LOCATION
8.FORWARD TRAVERSAL
9.BACKWARD TRAVERSAL
10.SORTING
11.COUNT NUMBER OF NODES
12.SEARCH AN ELEMENT
13.EXIT

ENTER YOUR OPTION : 8
         16      15      20      14      13      12
1.LIST
2.INSERT IN BEGINNING
3.INSERT AT END
4.INSERT BEFORE A GIVEN NODE
5.DELETE FROM BEGINNING
6.DELETE FROM END
7.DELETE NODE BEFORE A SPECIFIED LOCATION
8.FORWARD TRAVERSAL
```

```
9.BACKWARD TRAVERSAL
10.SORTING
11.COUNT NUMBER OF NODES
12.SEARCH AN ELEMENT
13.EXIT

ENTER YOUR OPTION : 10

1.LIST
2.INSERT IN BEGINNING
3.INSERT AT END
4.INSERT BEFORE A GIVEN NODE
5.DELETE FROM BEGINNING
6.DELETE FROM END
7.DELETE NODE BEFORE A SPECIFIED LOCATION
8.FORWARD TRAVERSAL
9.BACKWARD TRAVERSAL
10.SORTING
11.COUNT NUMBER OF NODES
12.SEARCH AN ELEMENT
13.EXIT

ENTER YOUR OPTION : 11

NUMBER OF NODES ARE : 6
1.LIST
2.INSERT IN BEGINNING
3.INSERT AT END
4.INSERT BEFORE A GIVEN NODE
5.DELETE FROM BEGINNING
6.DELETE FROM END
7.DELETE NODE BEFORE A SPECIFIED LOCATION
8.FORWARD TRAVERSAL
9.BACKWARD TRAVERSAL
10.SORTING
11.COUNT NUMBER OF NODES
12.SEARCH AN ELEMENT
13.EXIT

ENTER YOUR OPTION : 12
ENTERR THE ELEMEMENT TO BE SEARCHED :16


NUMBER IS FOUND!


1.LIST
2.INSERT IN BEGINNING
```

```
3.INSERT AT END
4.INSERT BEFORE A GIVEN NODE
5.DELETE FROM BEGINNING
6.DELETE FROM END
7.DELETE NODE BEFORE A SPECIFIED LOCATION
8.FORWARD TRAVERSAL
9.BACKWARD TRAVERSAL
10.SORTING
11.COUNT NUMBER OF NODES
12.SEARCH AN ELEMENT
13.EXIT

ENTER YOUR OPTION : 13
```

Harsh Kaushal
2003085
C21

Experiment 7

Aim: Implement Circular linked list

Theory:

Circular linked list: linked list where all nodes form a circle.

→ Useful for implementation of a queue. We don't need to maintain two pointers for front and rear if we use the linked list.

→ Any node can be starting point. We can traverse the list by starting from any point. We just need to stop when the first visited node is visited again.

→ Different Operations:

1) Insert in beginning:

Algorithm: Insert_beg(val)

1. create new node.
2. newnode → data = val
   if (head == null){
   head = newnode
   newnode → next = head.
   return }.
   temp = head.
   while (temp → next) = null)
   temp = temp → next.
   new node → next = head.
   temp → next = newnode
   head = newnode }.

Harsh Kushwaha
2003085

2> Insert at end:

Algorithm: A node is inserted at the end of the circular linked list

```
insert_end(val){
create newnode
newnode->data=val
if(head==null){
   head=newnode
   newnode->next=head
   return
}
temp=head
while(temp->next!=head)
   temp=temp->next
newnode->next=head
temp->next=newnode
}
```

3> Delete from beginning:

Algorithm:

```
del_beg(){
if(head==null)
print(underflow)
else if(head->next==head){
   temp=head
   head=null
   free(temp)}
else{
   temp=head
```

Harsh Kashikar

```
while (temp→next != head)
    temp = temp→next
temp→next = head→next
free(head)
head = temp→next
}
```

4. Delete from the End:

```
Algorithm: del_End(){
    if (head == null)
        print (underflow)
    else if (head→next == head){
        temp = head
        head = null
        free(temp)
    }
temp1 = temp2 = head
while (temp1→next != head){
    temp2 = temp1
    temp1 = temp1→next
}
temp2→next = temp1→next
    free (temp))
}
```

Harshkaul
2003085

5. Forward traversal (Display):

Algorithm: Display(start){
ptr = start.
while (Ptr → next != start){
print (Ptr → data)
ptr → ptr → next.
}
print (ptr → data);
return start.

6. Reverse traversal

Algorithm: back (start){
if (start → next == start)
print (start → data)
return 0.
back (start → next)
print (start → data).
return 0
}.

Conclusion: If the size of the list is fixed, it is much more efficient to use circular list. We can go to any node from any node in the circular list which was not possible in the singly list if we reached the last node.

PROGRAM:

Write a menu driven code to implement Circular Linked List.

Code:

```
#include <iostream>
#include <conio.h>
using namespace std;
struct node
{
    int data;
    struct node *next;
};
int flag = 0;
int count = 0;
struct node *start = NULL;
struct node *list(struct node *);
struct node *display(struct node *);
struct node *insert_beg(struct node *);
struct node *insert_end(struct node *);
struct node *delete_beg(struct node *);
struct node *delete_end(struct node *);
void backtraversal(struct node *);
struct node *nodes(struct node *);
int main(int argc, char *argv[])
{
    int option;
    do
    {
        cout<<"\n1.LIST ";
        cout<<"\n2.ADD NODE IN THE BEGINING";
        cout<<"\n3.ADD NODE IN THE END";
        cout<<"\n4.DELETE A NODE FROM BEGINNING";
        cout<<"\n5.DELETE A NODE FROM END";
        cout<<"\n6.DISPLAY";
        cout<<"\n7.DISPLAY REVERSE";
        cout<<"\n8.COUNT THE NUMBER OF NODES ";
        cout<<"\n9.EXIT";
        cout<<"\n\nENTER YOUR OPTION : ";
        cin>>option;
        switch (option)
        {
        case 1:
            start = list(start);
            cout<<"\n--LINKED LIST CREATED--";
            break;
        case 2:
            start = insert_beg(start);
```

```
            break;
        case 3:
            start = insert_end(start);
            break;
        case 4:
            start = delete_beg(start);
            break;
        case 5:
            start = delete_end(start);
            break;
        case 6:
            start = display(start);
            break;
        case 7:
            backtraversal(start);
            break;
        case 8:
            start = nodes(start);
            cout<<"THE NUMBER OF NODES IN THE LINKED LIST ARE : "<<count;
            break;
        }
    } while (option != 9);
    getch();
    return 0;
}
struct node *list(struct node *start)
{
    struct node *new_node, *ptr;
    int num;
    cout<<"ENTER -1 TO END!"<<endl;
    cout<<"\nENTER THE DATA :";
    cin>>num;
    while (num != -1)
    {
        new_node = (struct node *)malloc(sizeof(struct node));
        new_node->data = num;
        if (start == NULL)
        {
            new_node->next = new_node;
            start = new_node; // new node ka address
        }
        else
        {
            ptr = start;
            while (ptr->next != start)
                ptr = ptr->next;
            ptr->next = new_node;
            new_node->next = start;
```

```
        }
        cout<<"\nENTER THE DATA : ";
        cin>>num;
    }
    return start;
}
struct node *display(struct node *start)
{
    if (flag == 1)
    {
        cout<<"\n\n  EMPTY LIST  \n\n";
    }
    struct node *ptr;
    ptr = start;
    while (ptr->next != start)
    {
        cout<<"\t "<<ptr->data;
        ptr = ptr->next;
    }
    cout<<"\t "<<ptr->data;
    return start;
}
void backtraversal(struct node *ptr)
{
    if (ptr->next != start)
    {
        backtraversal(ptr->next);
    }
    cout<<" "<< ptr->data<<"\t";
}
struct node *insert_beg(struct node *start)
{
    struct node *new_node, *ptr;
    int num;
    cout<<"\nENTER THE DATA : ";
    cin>>num;
    new_node = (struct node *)malloc(sizeof(struct node));
    new_node->data = num;
    ptr = start;
    while (ptr->next != start)
        ptr = ptr->next;
    ptr->next = new_node;
    new_node->next = start;
    start = new_node;
    return start;
}
struct node *insert_end(struct node *start)
{
```

```
    struct node *ptr, *new_node;
    int num;
    cout<<"\nENTER THE DATA ";
    cin>>num;
    new_node = (struct node *)malloc(sizeof(struct node));
    new_node->data = num;
    new_node->next = start;
    ptr = start;
    while (ptr->next != start)
        ptr = ptr->next;
    ptr->next = new_node;
    return start;
}
struct node *delete_beg(struct node *start)
{
    struct node *ptr;
    ptr = start;
    while (ptr->next != start)
        ptr = ptr->next;
    ptr->next = start->next;
    free(start);
    start = ptr->next;
    return start;
}
struct node *nodes(struct node *start)
{
    struct node *ptr;
    ptr = start->next;
    count = 1;
    while (ptr != start)
    {
        count = count + 1;
        ptr = ptr->next;
    }
    return start;
}
struct node *delete_end(struct node *start)
{
    struct node *ptr, *preptr;
    ptr = start;
    while (ptr->next != start)
    {
        preptr = ptr;
        ptr = ptr->next;
    }
    preptr->next = ptr->next;
    free(ptr);
    return start;
```

```
}
```

OUTPUT:

```
PS D:\Harsh\SEM 3\DS\CODES> cd "d:\Harsh\SEM 3\DS\CODES\" ; if ($?) { g++ tempCodeRun

1.LIST
2.ADD NODE IN THE BEGINING
3.ADD NODE IN THE END
4.DELETE A NODE FROM BEGINNING
5.DELETE A NODE FROM END
6.DISPLAY
7.DISPLAY REVERSE
8.COUNT THE NUMBER OF NODES
9.EXIT

ENTER YOUR OPTION : 1
ENTER -1 TO END!

ENTER THE DATA :12

ENTER THE DATA : 13

ENTER THE DATA : 14

ENTER THE DATA : -1

--LINKED LIST CREATED--
1.LIST
2.ADD NODE IN THE BEGINING
3.ADD NODE IN THE END
4.DELETE A NODE FROM BEGINNING
5.DELETE A NODE FROM END
6.DISPLAY
7.DISPLAY REVERSE
8.COUNT THE NUMBER OF NODES
9.EXIT

ENTER YOUR OPTION : 2

ENTER THE DATA : 15

1.LIST
2.ADD NODE IN THE BEGINING
3.ADD NODE IN THE END
4.DELETE A NODE FROM BEGINNING
5.DELETE A NODE FROM END
```

```
6.DISPLAY
7.DISPLAY REVERSE
8.COUNT THE NUMBER OF NODES
9.EXIT

ENTER YOUR OPTION : 3

ENTER THE DATA 16

1.LIST
2.ADD NODE IN THE BEGINING
3.ADD NODE IN THE END
4.DELETE A NODE FROM BEGINNING
5.DELETE A NODE FROM END
6.DISPLAY
7.DISPLAY REVERSE
8.COUNT THE NUMBER OF NODES
9.EXIT

ENTER YOUR OPTION : 6
        15      12      13      14      16
1.LIST
2.ADD NODE IN THE BEGINING
3.ADD NODE IN THE END
4.DELETE A NODE FROM BEGINNING
5.DELETE A NODE FROM END
6.DISPLAY
7.DISPLAY REVERSE
8.COUNT THE NUMBER OF NODES
9.EXIT

ENTER YOUR OPTION : 4

1.LIST
2.ADD NODE IN THE BEGINING
3.ADD NODE IN THE END
4.DELETE A NODE FROM BEGINNING
5.DELETE A NODE FROM END
6.DISPLAY
7.DISPLAY REVERSE
8.COUNT THE NUMBER OF NODES
9.EXIT

ENTER YOUR OPTION : 5

1.LIST
2.ADD NODE IN THE BEGINING
```

```
3.ADD NODE IN THE END
4.DELETE A NODE FROM BEGINNING
5.DELETE A NODE FROM END
6.DISPLAY
7.DISPLAY REVERSE
8.COUNT THE NUMBER OF NODES
9.EXIT

ENTER YOUR OPTION : 6
         12      13      14
1.LIST
2.ADD NODE IN THE BEGINING
3.ADD NODE IN THE END
4.DELETE A NODE FROM BEGINNING
5.DELETE A NODE FROM END
6.DISPLAY
7.DISPLAY REVERSE
8.COUNT THE NUMBER OF NODES
9.EXIT

ENTER YOUR OPTION : 7
 14      13      12
1.LIST
2.ADD NODE IN THE BEGINING
3.ADD NODE IN THE END
4.DELETE A NODE FROM BEGINNING
5.DELETE A NODE FROM END
6.DISPLAY
7.DISPLAY REVERSE
8.COUNT THE NUMBER OF NODES
9.EXIT

ENTER YOUR OPTION : 8
THE NUMBER OF NODES IN THE LINKED LIST ARE : 3
1.LIST
2.ADD NODE IN THE BEGINING
3.ADD NODE IN THE END
4.DELETE A NODE FROM BEGINNING
5.DELETE A NODE FROM END
6.DISPLAY
7.DISPLAY REVERSE
8.COUNT THE NUMBER OF NODES
9.EXIT

ENTER YOUR OPTION : 9
```

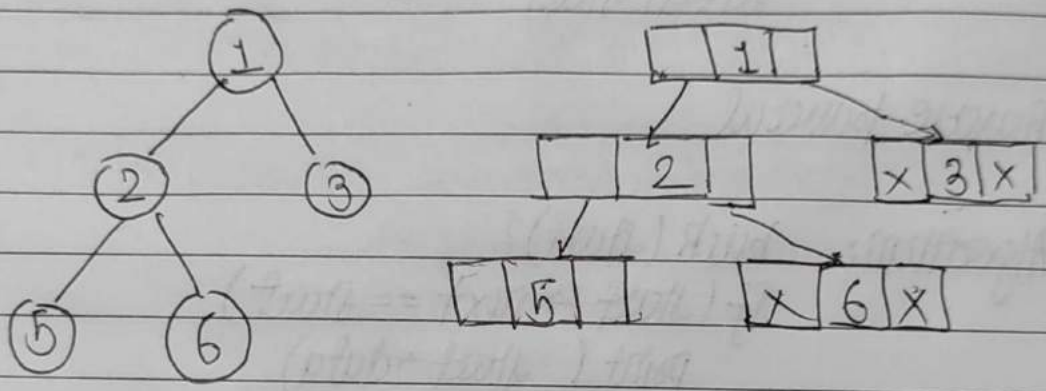
Harsh Kashiwal
2003085
C21

Experiment 8

Aim: Implement Binary search tree

Theory:

Binary tree means that the node can have maximum two children. 'Binary' itself suggests that 'two' therefore each node can have either 0, 1 or 2 children



Properties:

P1: Minimum number of nodes in a binary tree of height H. = H+1.

P2: Maximum number of nodes in a binary tree of height H = $2^{H+1} - 1$.

P3: Maximum number of nodes at any level 'L' in a binary tree = 2^L.

P4: Total number of leaf nodes in a Binary Tree = Total number of nodes with 2 children + 1.

1. Strict Binary Trees: Strict binary trees are those binary trees whose nodes either have 2 or 0 children.

2. Complete binary trees: Complete Binary Trees are those that have all their different levels completely filled. The only exception to this could be their last level, whose keys are predominantly on the left.

3. Perfect Binary tree: ~~Tree~~ Binary tree whose leaves are present at the same level and whose internal nodes carry two children.

4. Balanced Binary tree: It is binary tree in which height of the left and the right sub-trees of every node may differ by at most 1.

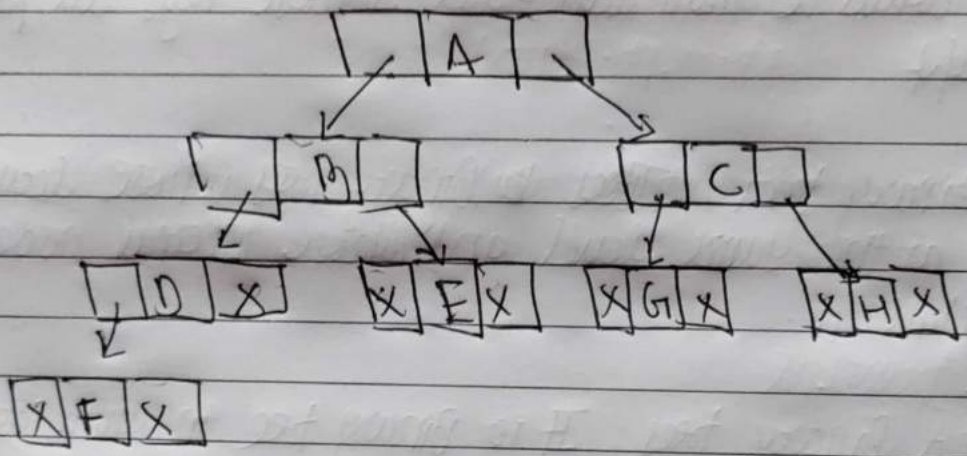
Representation:

1. Using Array: The array representation stores the tree data by scanning elements using level order fashion. So it stores nodes level by level. If some element is missing it left blank spaces for it.

1	2	3	4	5	6	7
5	10	17	-	8	15	25

2.

2. Using linked list: We use a double linked list to represent a binary tree. In this, every node consists of three fields. First field for storing left child address & 2nd for storing actual data & 3rd for storing right child address.



PROGRAM:

Write a menu driven code to implement BinarySearch Tree.

Code:

```
#include <iostream>
#include <conio.h>
using namespace std;

struct Node
{
    int data;
    Node *left;
    Node *right;
};

void display(Node *root)
{
    if (root == NULL)
        return;
    cout<<" "<<root->data;
    display(root->left);
    display(root->right);
}

Node *minValueNode(Node *root)
{
    Node *current = root;
    while (current->left != NULL)
    {
        current = current->left;
    }
    return current;
}

Node *maxValueNode(Node *root)
{
    Node *current = root;
    while (current->right != NULL)
    {
        current = current->right;
    }
    return current;
}

Node *insert(Node *root, int data)
{
    if (root == NULL)
```

```
    {
        root = (Node *)malloc(sizeof(Node));
        root->data = data;
        root->left = NULL;
        root->right = NULL;
    }
    else if (data <= root->data)
    {
        root->left = insert(root->left, data);
    }
    else
    {
        root->right = insert(root->right, data);
    }
    return root;
}

Node *deleteNode(Node *root, int data)
{
    if (root == NULL)
    {
        return root;
    }
    else if (data < root->data)
    {
        root->left = deleteNode(root->left, data);
    }
    else if (data > root->data)
    {
        root->right = deleteNode(root->right, data);
    }
    else
    {
        if (root->left == NULL && root->right == NULL)
        {
            free(root);
            root = NULL;
        }
        else if (root->left == NULL)
        {
            Node *temp = root;
            root = root->right;
            free(temp);
        }
        else if (root->right == NULL)
        {
            Node *temp = root;
            root = root->left;
```

```
            free(temp);
        }
        else
        {
            Node *temp = minValueNode(root->right);
            root->data = temp->data;
            root->right = deleteNode(root->right, temp->data);
        }
    }
    return root;
}

void search(Node *root, int data)
{
    if (root == NULL)
    {
        cout<<"Not found\n";
        return;
    }
    else if (data < root->data)
    {
        search(root->left, data);
    }
    else if (data > root->data)
    {
        search(root->right, data);
    }
    else
    {
        cout<<"Found "<<root->data<<endl;
    }
}

void postorder(Node *root)
{
    if (root != NULL)
    {
        postorder(root->left);
        postorder(root->right);
        cout<<" "<<root->data;
    }
}

void inorder(Node *root)
{
    if (root != NULL)
    {
        inorder(root->left);
```

```
        cout<<" "<<root->data;
        inorder(root->right);
    }
}

void preorder(Node *root)
{
    if (root != NULL)
    {
        cout<<" "<<root->data;
        preorder(root->left);
        preorder(root->right);
    }
}

int height(Node *root)
{
    if (root == NULL)
    {
        return 0;
    }
    else
    {
        int lheight = height(root->left);
        int rheight = height(root->right);
        if (lheight > rheight)
        {
            return (lheight + 1);
        }
        else
        {
            return (rheight + 1);
        }
    }
}

void deleteTree(Node *root)
{
    if (root != NULL)
    {
        deleteTree(root->left);
        deleteTree(root->right);
        free(root);
    }
}

void mirror(Node *root)
{
```

```
    if (root != NULL)
    {
        Node *temp = root->left;
        root->left = root->right;
        root->right = temp;
        mirror(root->left);
        mirror(root->right);
    }
}

int countNodes(Node *root)
{
    if (root == NULL)
    {
        return 0;
    }
    else
    {
        return (countNodes(root->left) + countNodes(root->right) + 1);
    }
}

int main()
{
    char ch;
    Node *root = NULL;
    int choice, data;
    while (1)
    {
        cout<<"\n1. Insertion\n2. Deleting a node\n3. Search\n4. Preorder
Traversal\n5. Inorder Traversal\n6. Postorder Traversal\n7. Height of a
tree\n8. Mirror of BST\n9. Count Total Numbers of Nodes\n10. Delete entire
Tree\n11. Display\n12. Smallest Element in the Tree\n13. Largest Element in
the Tree\n14. Exit\n\nEnter your choice: ";
        cin>>choice;
        switch (choice)
        {
        case 1:
            cout<<"\nEnter the data: ";
            cin>>data;
            root = insert(root, data);
            break;
        case 2:
            cout<<"\nEnter the data: ";
            cin>>data;
            root = deleteNode(root, data);
            break;
        case 3:
```

```
            cout<<"\nEnter the data: ";
            cin>>data;
            search(root, data);
            break;
        case 4:
            cout<<"\nPreorder traversal: ";
            preorder(root);
            break;
        case 5:
            cout<<"\nInorder traversal: ";
            inorder(root);
            break;
        case 6:
            cout<<"\nPostorder traversal: ";
            postorder(root);
            break;
        case 7:
            cout<<"\nHeight: "<<height(root)<<endl;
            break;
        case 8:
            cout<<"\nMirror of tree: ";
            mirror(root);
            inorder(root);
            break;
        case 9:
            cout<<"\nCount Nodes: "<<countNodes(root)<<endl;
            break;
        case 10:
            cout<<"\nAre you sure you want to delete the tree?\n";
            cin>>ch;
            if (ch == 'y' || ch == 'Y')
                deleteTree(root);
            else
                cout<<"\nTree not deleted\n";
            break;
        case 11:
            display(root);
            break;
        case 12:
            cout<<"\nMinimum value: "<<maxValueNode(root)->data<<endl;
            break;
        case 13:
            cout<<"\nMaximum value: "<<minValueNode(root)->data<<endl;
            break;
        case 14:
            exit(0);
        default:
            cout<<"\nWrong choice.\n";
```

```
        }
    }
    return 0;
}
```

OUTPUT:

```
Try the new cross-platform PowerShell https://aka.ms/pscore6
PS D:\Harsh\SEM 3\DS\CODES> cd "d:\Harsh\SEM 3\DS\CODES\" ; if ($?) { g++ Tree.cpp -o

1. Insertion
2. Deleting a node
3. Search
4. Preorder Traversal
5. Inorder Traversal
6. Postorder Traversal
7. Height of a tree
8. Mirror of BST
9. Count Total Numbers of Nodes
10. Delete entire Tree
11. Display
12. Smallest Element in the Tree
13. Largest Element in the Tree
14. Exit

Enter your choice: 1

Enter the data: 12

1. Insertion
2. Deleting a node
3. Search
4. Preorder Traversal
5. Inorder Traversal
6. Postorder Traversal
7. Height of a tree
8. Mirror of BST
9. Count Total Numbers of Nodes
10. Delete entire Tree
11. Display
12. Smallest Element in the Tree
13. Largest Element in the Tree
14. Exit

Enter your choice: 1

Enter the data: 13

1. Insertion
2. Deleting a node
3. Search
4. Preorder Traversal
```

```
5. Inorder Traversal
6. Postorder Traversal
7. Height of a tree
8. Mirror of BST
9. Count Total Numbers of Nodes
10. Delete entire Tree
11. Display
12. Smallest Element in the Tree
13. Largest Element in the Tree
14. Exit

Enter your choice: 1

Enter the data: 14

1. Insertion
2. Deleting a node
3. Search
4. Preorder Traversal
5. Inorder Traversal
6. Postorder Traversal
7. Height of a tree
8. Mirror of BST
9. Count Total Numbers of Nodes
10. Delete entire Tree
11. Display
12. Smallest Element in the Tree
13. Largest Element in the Tree
14. Exit

Enter your choice: 11
 12 13 14
1. Insertion
2. Deleting a node
3. Search
4. Preorder Traversal
5. Inorder Traversal
6. Postorder Traversal
7. Height of a tree
8. Mirror of BST
9. Count Total Numbers of Nodes
10. Delete entire Tree
11. Display
12. Smallest Element in the Tree
13. Largest Element in the Tree
14. Exit
```

```
Enter your choice: 2

Enter the data: 11

1. Insertion
2. Deleting a node
3. Search
4. Preorder Traversal
5. Inorder Traversal
6. Postorder Traversal
7. Height of a tree
8. Mirror of BST
9. Count Total Numbers of Nodes
10. Delete entire Tree
11. Display
12. Smallest Element in the Tree
13. Largest Element in the Tree
14. Exit

Enter your choice: 4

Preorder traversal:  12 13 14
1. Insertion
2. Deleting a node
3. Search
4. Preorder Traversal
5. Inorder Traversal
6. Postorder Traversal
7. Height of a tree
8. Mirror of BST
9. Count Total Numbers of Nodes
10. Delete entire Tree
11. Display
12. Smallest Element in the Tree
13. Largest Element in the Tree
14. Exit

Enter your choice: 5

Inorder traversal:  12 13 14
1. Insertion
2. Deleting a node
3. Search
4. Preorder Traversal
5. Inorder Traversal
6. Postorder Traversal
7. Height of a tree
```

```
7. Height of a tree
8. Mirror of BST
9. Count Total Numbers of Nodes
10. Delete entire Tree
11. Display
12. Smallest Element in the Tree
13. Largest Element in the Tree
14. Exit

Enter your choice: 12

Minimum value: 14

1. Insertion
2. Deleting a node
3. Search
4. Preorder Traversal
5. Inorder Traversal
6. Postorder Traversal
7. Height of a tree
8. Mirror of BST
9. Count Total Numbers of Nodes
10. Delete entire Tree
11. Display
12. Smallest Element in the Tree
13. Largest Element in the Tree
14. Exit

Enter your choice: 13

Maximum value: 12

1. Insertion
2. Deleting a node
3. Search
4. Preorder Traversal
5. Inorder Traversal
6. Postorder Traversal
7. Height of a tree
8. Mirror of BST
9. Count Total Numbers of Nodes
10. Delete entire Tree
11. Display
12. Smallest Element in the Tree
13. Largest Element in the Tree
14. Exit
```

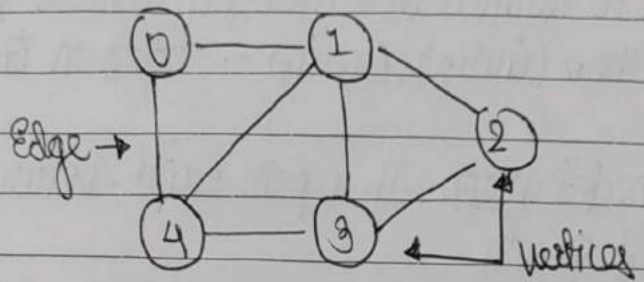
Harsh Kashiwal
2003085
C21

Experiment 9

Aim: Implement Graph traversal methods

Theory:

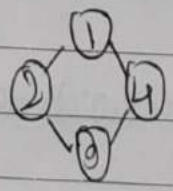
A Graph is a non linear data structure consisting of nodes and edges. The nodes are sometimes also referred to as vertices and the edges are lines or arcs that connect any two nodes in the graph.



Vertices: {0,1,2,3,4}.
Edges: {01, 12, 23, 34, 04, 14, 13}.

- Two nodes are adjacent if they are connected by an edge.
- Graphs can be directed or undirected.

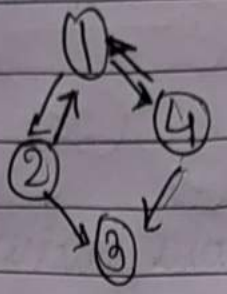
In undirected graph, edges do not have any direction associated with them.
i.e pair (u,v) & (v,u) represents the same edge.
eg.



{12, 23, 34, 41} ✓
{21, 32, 43, 14} ✗ only 4

Harsh Kashivedal
2003085

In directed graph, edges forms an ordered pair i.e (u,v) and (v,u) are two distinct edges.



Terminologies:

1) Path: Can be defined as sequence of vertices ($v_1, v_2 \dots v$) in such a way that there (v_1, v_2), (v_2, v_3) ... edges in G(E).

2) Strongly connected graph: If a path exist between each pair of vertices of graph.

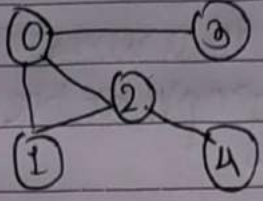
3) Indegree: of a node (u) is no. of edges that terminates at u.
4) Outdegree: of a node (u) is no. of edges that originates at u.
5) Degree: of a node (u), is sum of indegree & outdegree of that node
6) length: No. of edges in the path.
7) cycle: A path that starts and ends at same nodes

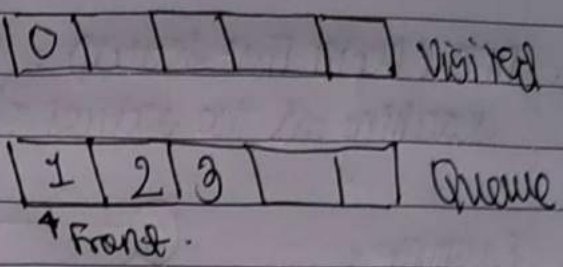
1. BFS (Breadth First Search): A standar BFS implementation puts each vertex of the graph into one of two categories.
   - Visited
   - Not visited

The purpose of the algorithm is to mark each vertex as visited while avoiding cycles.

Harsh Agrawal
2003085

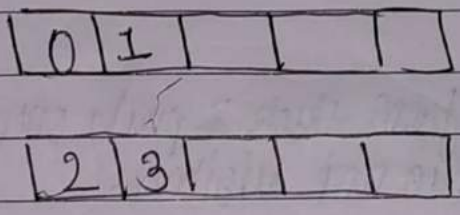
Example:



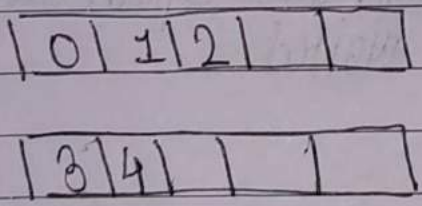


Start from vertex 0, BFS algorithm starts by putting in visited list & putting all its adjacent vertices in the stack.

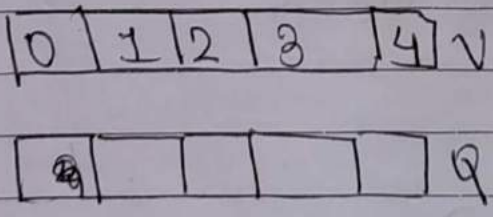
Next, we visit the element at the front of queue i.e 1 & go to its adjacent nodes, ~~now~~ we visit ~~2 instead of 0~~.



V(2) has an unvisited adjacent vertex in 4, so we add that the back of the queue & visit 3



Similarly,

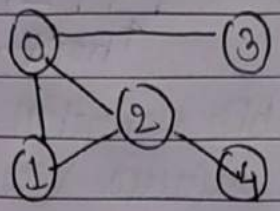


0 is already visited. So queue is empty. BFS completed.

Hash Kashiwal
2003085
C21

DFS (Depth First Search): DFS is a recursive algorithm for searching all the vertices of a graph or tree data structure.

Example:



1> Initially push 0 onto stack as follows

stack: 0

2> Pop top element 0 from stack & push onto stack all adjacent nodes of 0 which is not visited.

stack: 1 2 3

3> Pop top element (1) from stack & push all adjacent nodes of (1) which is not visited.

stack: 2 3

4> Pop top element (2) from stack & push all adjacent nodes which is not visited

stack: 4 3

Similarly:

5> Pop top element (4)

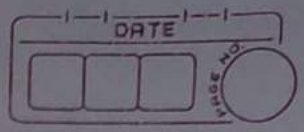
stack: 3

6> Pop top element (3)

stack: Empty!

Visited: 0 1 2 4 3

Harsh Kaliwal
2003085



Conclusion: BFS is more suitable for searching vertices which are closer to the given source while DFS is more suitable when there are solutions away from source.

PROGRAM:

Write a program to implement infix to postfix conversion using stack.

Code:

```
#include <iostream>
#include <conio.h>
using namespace std;

int q[20],top=-1,front=-1,rear=-1,a[20][20],vis[20],stack[20];
int deleteQ();
void add(int item);
void bfs(int s,int n);
void dfs(int s,int n);
void push(int item);
int pop();
int main()
{
int n,i,s,ch,j;
char c,;
cout<<"ENTER THE NUMBER VERTICES ";
cin>>n;
for(i=1;i<=n;i++)
{
for(j=1;j<=n;j++)
{
cout<<"ENTER 1 IF "<<i<<" HAS A NODE WITH "<<j<<" ELSE 0 ";
cin>>a[i][j];
}
}
cout<<"THE ADJACENCY MATRIX IS\n";
for(i=1;i<=n;i++)
{
for(j=1;j<=n;j++)
{
cout<<"\t"<<a[i][j];
}
cout<<"\n";
}

do
{
for(i=1;i<=n;i++)
vis[i]=0;
cout<<"\nENTER YOUR CHOICE";
cout<<"\n1.BFS Traversal";
cout<<"\n2.DFS Traversal\n";
cin>>ch;
```

```
cout<<"ENTER THE SOURCE VERTEX :";
cin>>s;

switch(ch)
{
case 1:bfs(s,n);
break;
case 2:
dfs(s,n);
break;
}
cout<<"\nDO U WANT TO CONTINUE(Y/N) ? ";
cin>>c;
}while((c=='y')||(c=='Y'));
}


//**************BFS**************//
void bfs(int s,int n)
{
int p,i;
add(s);
vis[s]=1;
p=deleteQ();
if(p!=0)
cout<<p<<"\t";
while(p!=0)
{
for(i=1;i<=n;i++)
if((a[p][i]!=0)&&(vis[i]==0))
{
add(i);
vis[i]=1;
}
p=deleteQ();
if(p!=0)
cout<<p<<"\t";
}
for(i=1;i<=n;i++)
if(vis[i]==0)
bfs(i,n);
}


void add(int item)
{
if(rear==19)
cout<<"QUEUE FULL";
else
```

```
{
if(rear==-1)
{
q[++rear]=item;
front++;
}
else
q[++rear]=item;
}
}
int deleteQ()
{
int k;
if((front>rear)||(front==-1))
return(0);
else
{
k=q[front++];
return(k);
}
}


//***************DFS*****************//
void dfs(int s,int n)
{
int i,k;
push(s);
vis[s]=1;
k=pop();
if(k!=0)
cout<<k<<"\t";
while(k!=0)
{
for(i=1;i<=n;i++)
if((a[k][i]!=0)&&(vis[i]==0))
{
push(i);
vis[i]=1;
}
k=pop();
if(k!=0)
cout<<k<<"\t";
}
for(i=1;i<=n;i++)
if(vis[i]==0)
dfs(i,n);
}
void push(int item)
```

```
{
if(top==19)
cout<<"Stack overflow ";
else
stack[++top]=item;
}
int pop()
{
int k;
if(top==-1)
return(0);
else
{
k=stack[top--];
return(k);
}
return 0;
}
```

OUTPUT:

```
PS D:\Harsh\SEM 3\DS\CODES> cd "d:\Harsh\SEM 3\DS\CODES\" ; if ($?)
ENTER THE NUMBER VERTICES 3
ENTER 1 IF 1 HAS A NODE WITH 1 ELSE 0 1
ENTER 1 IF 1 HAS A NODE WITH 2 ELSE 0 1
ENTER 1 IF 1 HAS A NODE WITH 3 ELSE 0 0
ENTER 1 IF 2 HAS A NODE WITH 1 ELSE 0 1
ENTER 1 IF 2 HAS A NODE WITH 2 ELSE 0 0
ENTER 1 IF 2 HAS A NODE WITH 3 ELSE 0 1
ENTER 1 IF 3 HAS A NODE WITH 1 ELSE 0 0
ENTER 1 IF 3 HAS A NODE WITH 2 ELSE 0 1
ENTER 1 IF 3 HAS A NODE WITH 3 ELSE 0 1
THE ADJACENCY MATRIX IS
        1       1       0
        1       0       1
        0       1       1

ENTER YOUR CHOICE
1.BFS Traversal
2.DFS Traversal
1
ENTER THE SOURCE VERTEX :2
2       1       3
DO U WANT TO CONTINUE(Y/N) ? y

ENTER YOUR CHOICE
1.BFS Traversal
2.DFS Traversal
2
ENTER THE SOURCE VERTEX :2
2       3       1
DO U WANT TO CONTINUE(Y/N) ? n
PS D:\Harsh\SEM 3\DS\CODES>
```

Harsh Kashikar
2003085
C21

Experiment 10

AIM: Implement linear and Binary search algorithm

Theory:

⇒ Linear search: The linear search algorithm searches all elements in the array sequentially. When a data is unsorted, a linear search algorithm is preffered.

Space complexity for linear search is O(n) as it does not use any extra space where n is the no. of elements in an array.

Algorithm:

```
LinearSearch (A, n, x) {
1. for i=0 to n-1
2. { if (A[i] == x)
        return i }
3. return -1
}
```

Example :- We are given the following linear array.

Element 15 has to be searched in it using linear search algorithm.

92	87	53	10	15	21	77
0	1	2	3	4	5	6

LSA compares 15 with all the elements of the array one by one. It continues searching until either the element is found or all the elements are searched. It first compares with 1st element '92' since 15 ≠ 92 it moves to next and so on...

Harsh Kashiwal
085

Binary search: Binary is one of the fastest searching algorithms

To find an element;

⇒ Set the range on index in which element is to be searched, there are 3 cases

1. If it matches, element is at middle index.
2. If search value is less than value at middle index, then upper bound is shifted to the first half of the array.
3. If search value is greater than value at middle index, then lower bound is shifted to the second half of the array.

The process terminates when searched element is found or process is repeated.

Algorithm: Iterative

```
BinarySearch (A, n, x) {
  start = 0, end = n-1
  while (start <= end) {
    mid = (start + end)/2
    if (A[mid] == x)
      return mid
    else if (x < A[mid])
      end = mid-1
    else start = mid+1
  }
  return -1
}
```

Harsh Kashivaal
2003085
(2)

Algorithm : Recursive

```
BinarySearch(A, start, end, x){
    if (start > end)
        return -1
    mid = (start + end)/2
    if (x == A[mid])
        return mid
    else if (x < A[mid])
    return BinarySearch(A, start, mid-1, x)
    else,
    return Binary Search(A, mid+1, end, x)
}
```

⇒ In each iteration or in each recursive call, the search gets reduced to half of the array. So for n elements in the array, there are $\log_2 n$ iterations or recursive calls.

∴ Time complexity of Binary Search Algorithm is $O(\log_2 n)$.

Example :

3	10	20	15	40	35	60
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]

15 has to be searched

1. To begin with, we take start = 0 and end = 6
2. mid = (start + end)/2
   = (0+6)/2 = 3
3. Here, a[mid] = a[3] = 15 which matches to the element being searched.
4. So, our search terminates and index 3 is returned.

Harsh Kaliwal
2003085

Conclusion: Binary search is more efficient and takes minimum time to search an element than a linear search.
But linearly search is less complex than binary search.

PROGRAM:

Linear Search

```
// Linear Search:
#include <iostream>
#include <conio.h>
using namespace std;
#define size 10
int main()
{
    int arr[size], n, i, num, found = 0;
    cout << "\n Enter the number of elements:";
    cin >> n;
    cout << "Enter elements:";
    for (i = 0; i < n; i++)
        cin >> arr[i];
    cout << "\nEnter the element to be searched:\n";
    cin >> num;
    for (i = 0; i < n; i++)
    {
        if (arr[i] == num)
        {
            found = 1;
            cout << "Found at " << i + 1;
            break;
        }
    }
    if (found == 0)
    {
        cout << "Element not found";
    }
    return 0;
}
```

OUTPUT: Linear

```
PS D:\Harsh\SEM 3\DS\CODES> cd "d:\Harsh\SEM

 Enter the number of elements:3
Enter elements:12
13
14

Enter the element to be searched:
12
Found at 1
PS D:\Harsh\SEM 3\DS\CODES>
```

Binary Search:

```
//Binary Search:
#include <iostream>
#include <conio.h>
using namespace std;
#define size 10
int smallest(int arr[], int k, int n){
        int pos = k, small = arr[k], i;
        for (i = k + 1; i < n; i++)
        {
            if (arr[i] < small)
            {
                small = arr[i];
                pos = i;
            }
        }
        return pos;
    }
 void selection_sort(int arr[], int n)
    {
        int k, pos, temp;
        for (k = 0; k < n; k++)
        {
            pos = smallest(arr, k, n);
            temp = arr[k];
            arr[k] = arr[pos];
            arr[pos] = temp;
        }
    }
int main()
{
    int arr[size], n, i, num, start, end, mid, found = 0;
    cout << "\n Enter the number of elements:";
    cin >> n;
    cout << "Enter elements:";
    for (i = 0; i < n; i++)
        cin >> arr[i];
    selection_sort(arr, n);
    cout << "The sorted array is:\n";
    for (i = 0; i < n; i++)
        cout << arr[i] << "\t";
    cout << "\nEnter the element to be searched:\n";
    cin >> num;
    start = 0;
    end = n - 1;
    while (start <= end)
```

```
    {
        mid = (start + end) / 2;
        if (arr[mid] == num)
        {
            cout << "Found at " << mid + 1;
            found = 1;
            break;
        }
        else if (arr[mid] > num)
            end = mid - 1;
        else
            start = mid + 1;
    }
    if (start > end && found == 0)
    {
        cout << "Element not found";
    }


    return 0;
}
```

OUTPUT: Binary

```
PS D:\Harsh\SEM 3\DS\CODES> cd "d:\Harsh\SEM 3\DS\CODES\"

 Enter the number of elements:5
Enter elements:12
13
14
15
16
The sorted array is:
12      13      14      15      16
Enter the element to be searched:
17
Element not found
PS D:\Harsh\SEM 3\DS\CODES>
```

Harsh Kasliwal
2003085
C21

Page No.
Date

Written Assignment-1

1) Differentiate b/w linear and non-linear data structure.

	Linear data structure	Non linear data structure
i)	Elements are arranged sequentially	Elements are not arranged sequentially
ii)	Each element is connected to its previous and next element	Elements can have multiple connections with other elements or may have none at all.
iii)	Elements can be traversed in a single run since they are connected sequentially	Multiple runs may be required to traverse all elements of this data structure.
iv)	They can be stored in memory easily, as memory itself is accessed sequentially.	Efficient methods of storing these data structures in memory must be developed.
v)	All elements are present on a single level	Elements may be present on multiple levels.
vi)	Eg: Array, list, queue, stack	Eg: Graph, map, tree.

② Explain ADT of Stack

A stack is a ADT (Abstract Data type) which stores a collection of items. In a stack items can only be added and removed from one end, thus a stack follows LIFO principle.
It is based on stacks in real life, for example, a stack of books or plates. The insertion operation is called 'push' and the deletion operation is called 'pop'.
Other operations:
1. peek - Returns the element at the top of stack, without removing it.
2. isFull - Checks if stack is full or not.
3. isEmpty - checks if stack is empty or not.
4. display - Shows all the elements in the stack.

To implement stack ADT, we need to use an array to hold the items and two variables : TOP and MAX. TOP stores the index of the element at top and MAX stores capacity. Then, the operations mentioned above are implemented as follows:

1> push(element) : 1. If stack is full, print "overflow"
2. Otherwise set TOP = TOP+1 & array[TOP] = element.

2> pop() : 1. If stack is empty, print "Underflow"
2. Otherwise return array[TOP] and set TOP = TOP-1

3> isFull() : If TOP is MAX-1 return True otherwise return False.

4> isEmpty() : If TOP = -1 return True otherwise return False.

⑤ WAP to implement priority queue with the following operations:
1> Insert 2> Delete 3> Search 4> Display

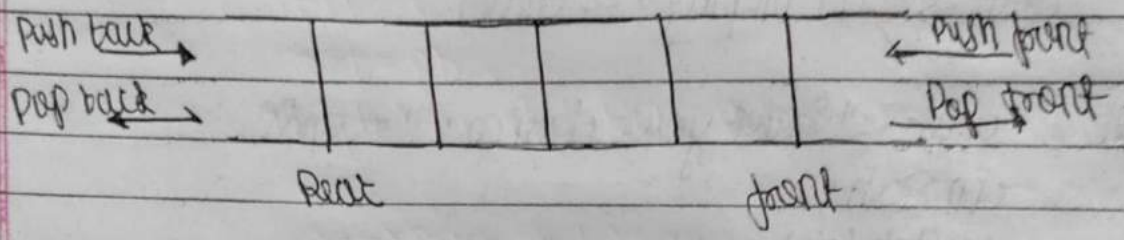
```
#include<iostream>
using namespace std;
#define n 5
void enqueue
int queue[n];
int size=0, rear=-1, front=-1;

void enqueue(){
 int val;
 if(rear==n-1)
 cout<<"queue overflow"<<endl;
 else{
    if(front==-1)
    front=0;
    cout<<"Insert the element in queue:"<<endl;
    cin>>val;
    rear++;
    size++;
   queue[rear]=val;
 }
}
void dequeue(){
  if(front==-1 || front>rear){
      cout<<"queue Underflow";
       return;
  }
```

```
    else{
    cout<<" element deleted from queue is : "<< queue[front];
    front++;
     size--;
    }
}

void search(){
    int i;
    if(front==-1 && rear==-1)
    cout<<"Queue is empty"<<endl;
    else{
        for(i=front; i<=rear; i++)
            if(queue[i]=element)
            cout<<"Element found in queue with pos";
        }
    }
void Display(){
    if(front==-1)
    cout<<"Queue is empty"<<endl;
    else{
      cout<<"Queue elements are:";
      for(int i=front; i<=rear; i++)
      cout<<queue[i]<<" ";
      cout<<endl;
    }
}
```

```
int main() {
  int ch
  cout<<" 1) Insert" <<endl;
  cout <<" 2) Delete" <<endl;
  cout <<" 3) Search" <<endl;
  cout <<" 4) Display" << endl;

do { cout <<"Enter your choice :" <<endl
     cin >> ch;
     switch (ch) {
      case 1 : enqueue ();
      break;
     case 2: dequeue();
     break;
     case 3 : search();
     break;
     case 4 : Display();
      break
     default : cout << "Invalid" <<endl;
   }
 } while (ch!=4);
   return 0;
}
```

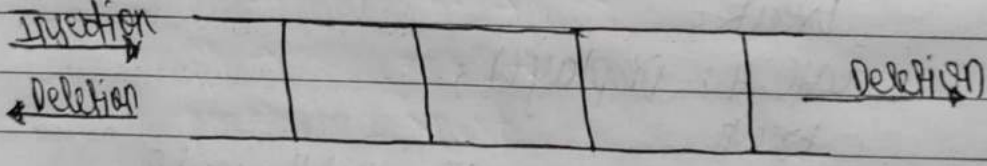
④ Write a short note on Double Ended Queue.

A Double ended queue is an ADT and is a generalized form of a queue in which elements can be added and removed from the front as well as the back.

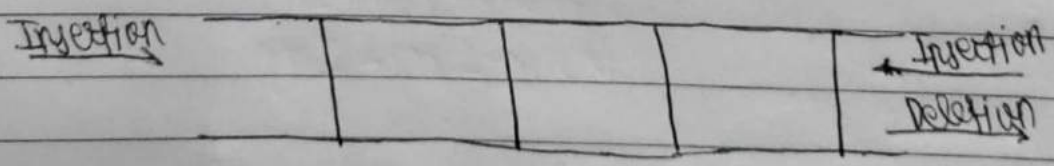


There are two types of deque:

1> Input restricted deque: In an input restricted deque, insertion can only be performed at one end but deletion can be performed at both ends.



2> Output restricted deque: In an output restricted deque, deletion can only be performed at one end, but insertion can be performed at both ends.



Page No.
Date

⑤ WAP to implement singly linked list application - Polynomial representation and addition.

```
#include<iostream>
  using namespace std;
#include<malloc.h>
typedef struct node{
      int coeff, pow;
      struct node *next;
   } NODE;
   NODE* add(NODE*root1, NODE *root2), *input();
         void display(NODE* root);

int main(){
   NODE*root1, *root2, *root;
   cout<<"Enter details of polynomial 1: \n");
   root1 = input();
   cout<<"Enter details of polynomial 2: \n");
   root2 = input();
   cout<<"Polynomial 1: "); display(root1);
   cout<<"\n Polynomial 2: "); display(root2);
   root = add(root1, root2);
   cout<<"\n Addition of polynomials: "); display(root);
   return 0;
  }

NODE *add(NODE*root1, NODE*root2){
   NODE*ptr1=root1, *ptr2=root2, *ptr=NULL, *root=NULL;
   int i=0;
```

```
while(ptr->next!=NULL || ptr2->next!=NULL){
    if(ptr==NULL){
        ptr=(NODE*)(malloc(sizeof(NODE)));
        ptr->next=NULL;
    }
    if(i==0)
    root=ptr;
    if(ptr1->next!=NULL && (ptr2->next==NULL || ptr1->pow
                                >ptr2->pow)){

        ptr->pow=ptr1->pow;
        ptr->coeff=ptr1->coeff;
        ptr1=ptr1->next;
    }
    else if(ptr2->next!=NULL && (ptr->next==NULL || ptr1->pow<
                                ptr2->pow)){

        ptr->pow=ptr2->pow;
        ptr->coeff=ptr2->coeff;
        ptr=ptr2->next
    }
    else{
        ptr->coeff=ptr1->coeff+ptr2->coeff;
        ptr->pow=ptr1->pow;
        ptr1=ptr1->next;
        ptr2=ptr2->next;
    }
    ptr->next=(NODE*)(malloc(sizeof(NODE)));
     ptr=ptr->next;
    ptr->next=NULL;
    i++;
}
return root;}
```

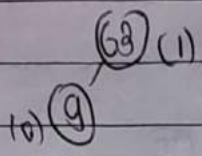
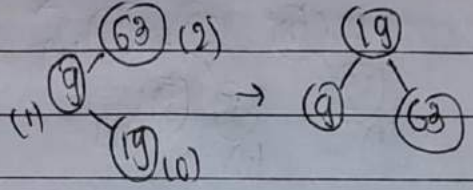
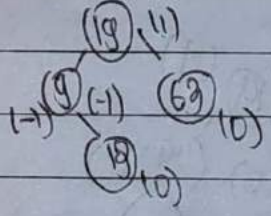
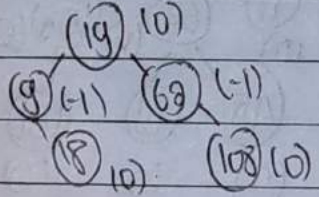
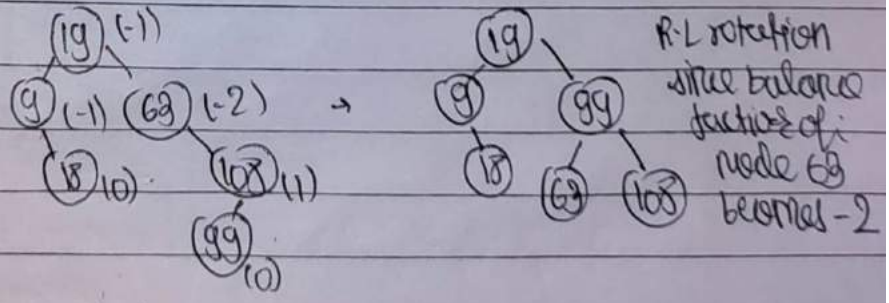
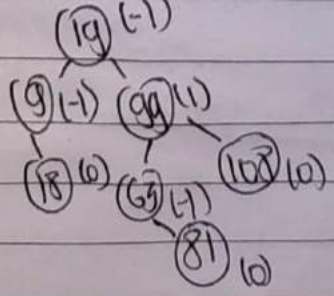
Harsh Kushwaha
2009086 C21

```
NODE *input(){
    NODE *root=NULL, *ptr=NULL;
    int i, n, coeff, pow;
    cout<<"Enter no. of terms in the polynomial:");
    cin>>n;
    for(i=0;i<n;i++){
        cout<<"Enter coefficient & power of term "<<i;
        cin>>coeff>>pow;
        if(ptr==NULL){
          ptr=(NODE*)(malloc(sizeof(NODE)));
          ptr->next=NULL;
        }
    if(i==0)
      root=ptr;
    ptr->coeff=coeff; ptr->pow=pow;
    ptr->next=(NODE*)(malloc(sizeof(NODE)));
    ptr=ptr->next;
    ptr->next=NULL;
}
  return root;
}
void display(NODE *root){
    NODE *ptr=root;
    while(ptr->next!=NULL){
     cout<<" "<<ptr->coeff<<"x^"<<ptr->pow;
     ptr=ptr->next;
     if(ptr->next!=Null)
       cout<<"+";
    }
}
```

Harsh Kasliwal
2003055
C21

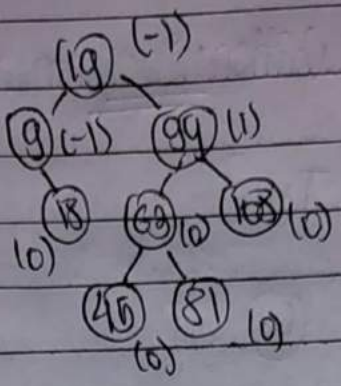
Page No.
Date

Written Assignment 2

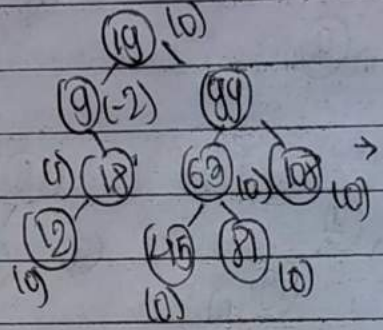
ID	Element	Tree
.	63	
	9	 No rotation required.
	19	 L-R rotation required since balance of node 63 becomes 2.
	18	
	108	
	99	 R-L rotation since balance factor of node 63 becomes -2
	81	

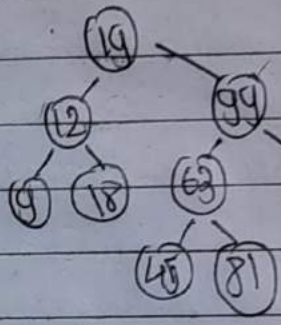
Harsh Kasliwal (2)
2003085

45



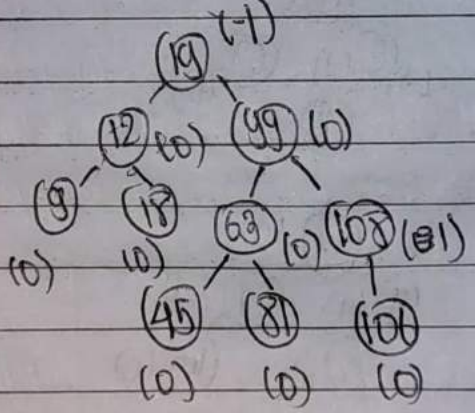
12





R-L rotation
BF of node becomes -2.

106



No rotation required.

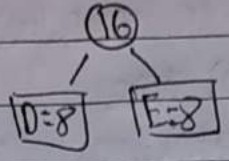
Harsh
085

Page No.
Date

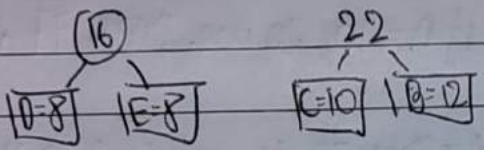
(2) Symbol A B C D E
Frequency 24 12 10 8 8

D=8, E=8, C=10, B=12, A=24

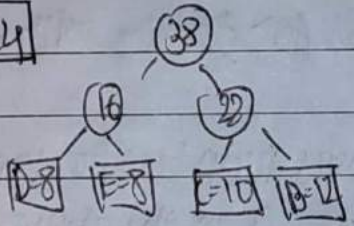
i=0 [C=10] [B=12] (16) [A=24] Adding D & E to the tree as they are lowest



i=1 (16) 22 [A=24] Now (10) & (12)

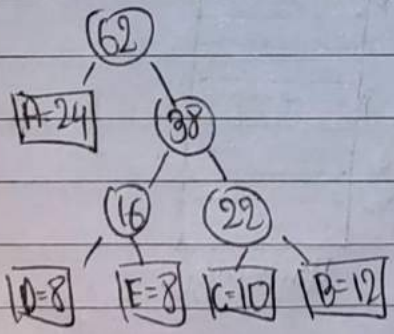


i=2 [A=24]



Now 16 & 22 are the two lowest nodes, so we ~~do~~ add them.

i=3



Combine A(24) & 38 node.

⇒ A=0, B=111, C=110, D=100, E=101;

are the Huffman codes for the given frequency table.

Hash
085

Page No.
Date

(3) Hashing is the process of converting a given key into a code, smaller than the input key using a hash function. The hash function maps data of arbitrary size to a fixed range. The values generated by hash function are called hash codes and are used to index a table called the hash table.

→ Hashing is done in order to have faster access to arbitrary data elements usually in O(1) time

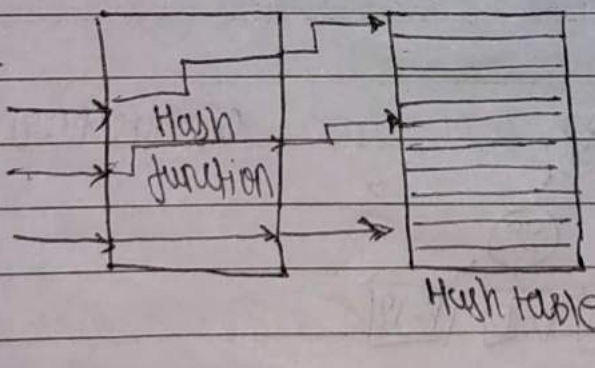
→ There are various hash functions available like modulo division folding, mid-square

→ Hash code of 77 will be 7 if n=10, so we store 77 at index 7 in the hash table

→ Now if we want to check whether 77 is contained in the hash table we just have to check the index 7.

→ There are also many methods for dealing with collisions, like linear probing quadratic probing.

Collisions occur when two keys map to the same index, like 77 and 87 both mapping to index 7 in the above example.



Hash 085

(5)

Page No.
Date

0 : 10
1 : 44
2 : 82
3 : 63
4 : 94
5 : 53
6 : 23
7 : 77
8 : 87
9 : 55

Linear probing.

i> 63 %. 10 = 3
ii> 82 %. 10 = 2
iii> 94 %. 10 = 4
iv> 77 %. 10 = 7
v> 53 %. 10 = 3
vi> 87 %. 10 = 7
vii> 23 %. 10 = 3
viii> 55 %. 10 = 5
ix> 10 %. 10 = 0
x> 44 %. 10 = 4

Probes
collision final location = 5 (3)
Collision final location = 8 (2)
Collision final location = 6 (4)
Collision final location = 9 (5)
Collision final location = 1 (8)
Total collisions = 5

Quadratic probing.

0 10
1
2 82
3 63
4 94
5 55
6
7 77
8 87
9 53

i> 63 %. 10 = 3.
ii> 82 %. 10 = 2
iii> 94 %. 10 = 4.
iv> 77 %. 10 = 7
v> 53 %. 10 = 3. Collision ✓
(3+1²) %. 10 = 4.
(3+2²) %. 10 = 7
(3+3²) %. 10 = 2.
(3+4²) %. 10 = 9 ✓
vi> 87 %. 10 = 7 Collision ✓
(7+1²) %. 10 = 8 ✓
vii> 23 %. 10 = 3 Collision ✓
(3+1²) %. 10 = 4
(3+2²) %. 10 = 7
(3+3²) %. 10 = 2.
(3+4²) %. 10 = 9
(3+5²) %. 10 = 8
(3+6²) %. 10 = 9
(3+7²) %. 10 = 2
(3+8²) %. 10 = 7
(3+9²) %. 10 = 4.
⇒ 23 cannot be placed in table.

Harsh 085 (6)

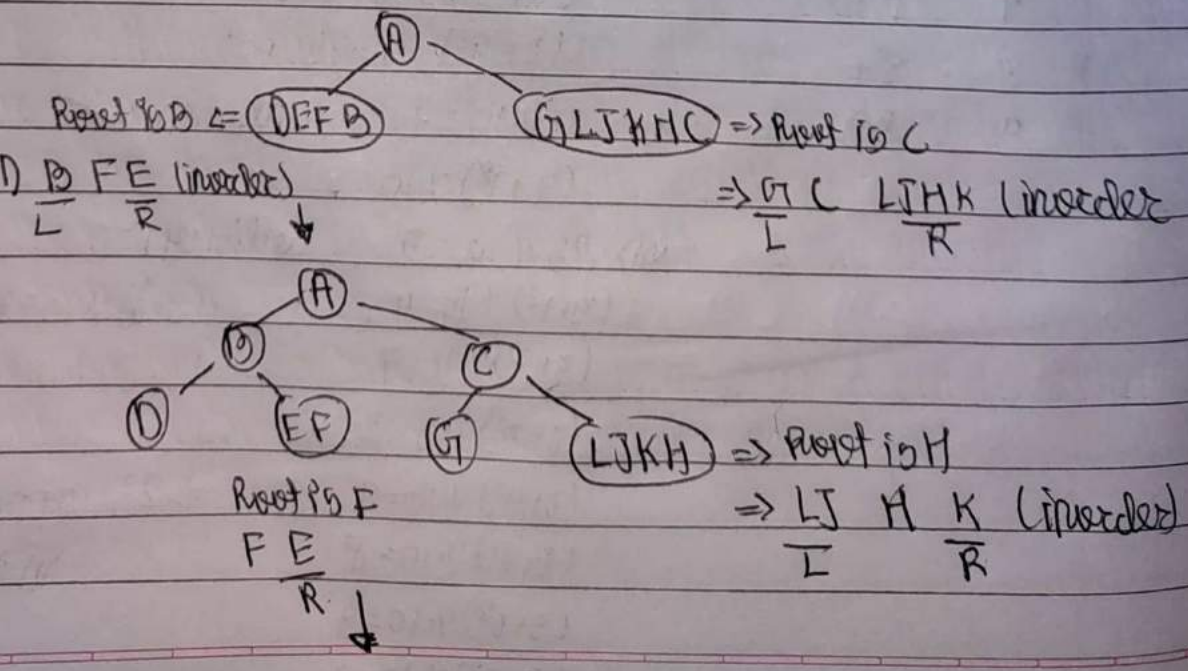
Page No.
Date

Y> 55 %10 = 5
I> 10 %10 = 0
A> 44 %10 = 4 collision ✓
(4+1²) %10 = 5
(4+2²) %10 = 8
(4+3²) %10 = 3
(4+4²) %10 = 0
(4+5²) %10 = 9
(4+6²) %10 = 0
(4+7²) %10 = 9
(4+8²) %10 = 8
(4+9²) %10 = 5
Cannot be placed in table

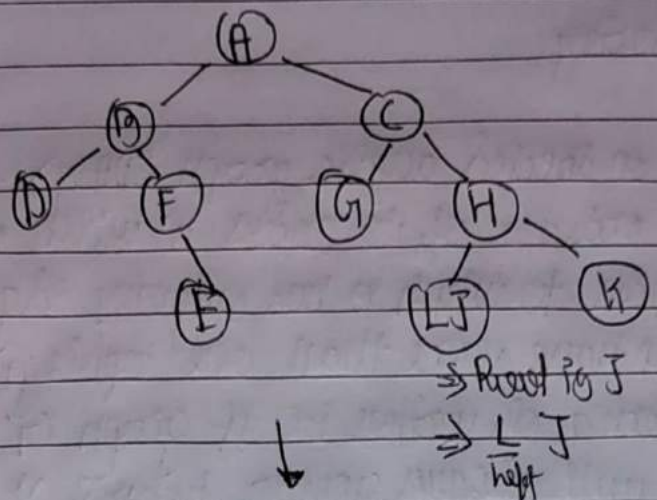
⇒ No. of collisions = 4.

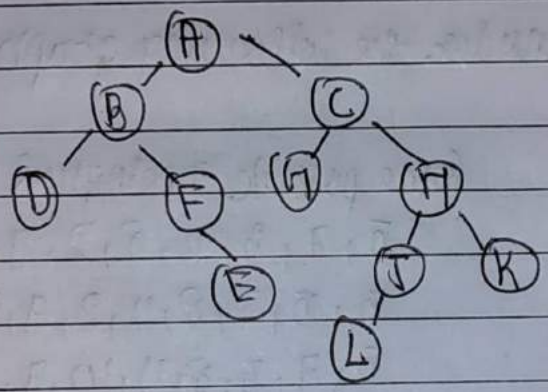
(4) Post : D E F B G L J K H C A → left Right root ⇒ root at end
In : D B F E A G C L J H K → left root Right ⇒ root in middle

DEFBGLJKHCA ⇒ Root is A



(7)





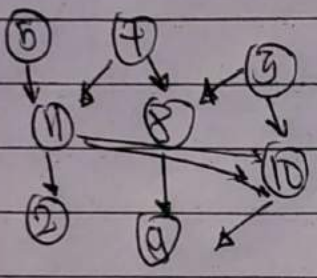
Harsh Kalicut
2003089 (21) (8)

Page No.
Date

(5) Topological Sorting

→ It is applied on directed acyclic graph (DAG).
→ It is linear ordering of its nodes in which nodes comes before all nodes to which it has outgoing edges
→ Every DAG can have more than one topological sort.
→ It is an ordering of vertices i.e if graph G contains an edge (u, v) then u will always appear before v in the ordering.

For example, consider the following graph



Some possible topological sortings are.
5, 7, 3, 11, 8, 2, 9, 10
3, 5, 7, 8, 11, 2, 9, 10
5, 7, 3, 8, 11, 10, 9, 2

→ A simple algorithm to implement topological sorting can be written using stack & recursion.

→ We start from nodes that has no incoming edges and push them into a stack without printing them, and set visited for those nodes as true. Then we pick any node and keep recursively calling the function for all adjacent nodes. Again we do not print any nodes Also, the node is only pushed in stack after all the adjacent nodes have been pushed
[Also set visited = true for any node accessed and do return if an already visited node is visited again.