### LIST OF EXPRIMENTS:

Week 1

a) Write a Program to Implement Stack Operations by using Array and Linked Lists.

b) Write a Program to Implement the Operations of Double Linked Lists

Week 2

a) Write a C program that uses stack operations to convert a given infix expression into its postfix

b) Write a Program to Implement Queue Operations by using Array and Linked Lists.

Week 3

Write a Program to Implement Circular Queue Operations by using Array and Linked Lists.

Week 4

Write a Program to Sort the set of elements by using

i) Quick Sort ii) Heap Sort. iii) Merge Sort

Week 5

Write a Program to Implement the Binary Search Tree Operations.

Week 6

Write a Program to Perform the Tree Traversal Techniques by using the Iterative Method

Week 7

Write C programs for implementing the following graph traversal algorithms:

a)Depth first traversal b)Breadth first traversal

Week 8

Write a Program to Implement All functions of a Dictionary by using Hashing

Week 9

Write a Program to Implement Skip List Operations.

Week 10

Write a Program to Implement Insertion, Deletion and Search Operations on SPLAY Trees.

Week 11

Write a program to Implement Insertion and Deletion Operations on AVL Trees

Week 12

Write a Program to Implement Insertion and Deletion Operations on B – Trees

**Note:** Use Classes and Objects to implement the above programs.

***Laboratory Objective:***

*To develop skills to design and analyze linear and non linear data structures.*

*Develop algorithms and programs for manipulating linked lists, stacks, queues, trees and graphs.*

*Develop recursive algorithms as they apply to trees and graphs.*

*To develop a base for advanced computer science study.*

***Learning Outcome:***

*At the end of the course students will be assessed to determine whether they are able to*

*Study variety of advanced abstract data type (ADT) and data structures and their*

*Implementations.*

*Identify and apply the suitable data structure for the given real world problem*

*Software Used: Turbo C++*

**Week 1**

1. Write a Program to Implement Stack Operations by using Array and Linked Lists.

**Aim:**

A c++ program to implement a stack and perform the operations like push,

Pop, peep and Change using arrays.

**Description:**

**Abstract Data Type:**

1. Stack is a data structure which allows insertion and deletion at the same end.
2. Inserting elements into a stack is called push.
3. Deleting an element from the stack is called pop.
4. Retrieving an element by given position number is called peep.
5. Updating an old element by new element by given position number is called Change.
6. In stack each and every operation can do the Changes by using top values.
7. In stack first inserted element is popped out last and last inserted element is popped out first.
8. Stack is last in First out Data Structure (i.e. LIFO).

**Member functions:**

1. **push(d):** this function insert given element into the stack.if push is successful then it

returns true else it return false.

1. **pop(d):** this function delete an element which is inserted last into the stack.if pop is

successful then it returns true else it return false.

1. **peep():** this function searches the element from the top and show the element of the

stack in the given position. if peep is successful then it returns true else it

return “wrong position”.

1. **Change():** this function updates the old element in the stack with new element of the

stack in the given position.if the Change is successful

**Algorithm:**

**1.Push(d):**

1. start
2. read n
3. if top greater than (n-1) then
4. print “ overflow”
5. else
6. top <- top +1
7. data[top] <- d
8. print “element is pushed into stack
9. stop.

**2.pop():**

1. Start
2. if top less than zero then
3. print “underflow”
4. else
5. d <- data [top]
6. top <- top -1
7. print “element is poped from the stack.
8. stop.

**3.Peep(k):**

1. start
2. if (top-p+1) is less than zero then
3. print “wrong position”
4. else
5. print p
6. print “element is”
7. d <- data[top-p+1]
8. print d
9. stop.

**4.Change(k,l):**

1. start

2. if (top-p+1) is less than zero

3. print “wrong position”

4. else

5. d <- data[top-p+1]

6. print “ Changed element is”

7. print data[top-p+1]

1. stop.

**/\* Program::1: to perform stack operations using arrays. \*/**

#include<iostream.h>

#include<process.h>

#include<conio.h>

class stack

{

int \*a,n,top;

public:

stack() // it is a default constructor which is called

// when the class variable is declared.

{

clrscr();

cout<<”\n Enter the size of the stack::”;

cin>>n;

a=new int [n];

top=-1;

}

void push(int);

int pop();

void print();

void peep(int);

void Change(int,int);

};

void stack :: push(int x) // inserts the elements into the stack.

{

if(top= = n-1)

cout<<”\n stack overflow \n”;

else

{

a[++top]=x;

cout<<x<<”is pushed into stack \n”;

}

}

int stack :: pop() // deletes the elements from the stack.

{

if(top= = -1)

{

cout<<”\n stack underflow \n”;

return -1;

}

else

return a[top--];

}

void stack :: print() // prints the elements of the stack.

{

for(int i=top;i>=0;i--)

cout<<”\n”<<a[i]<<”\n”;

}

void stack :: peep(int p) // searches the element of the stack for the given position.

// p is a integer variable which takes positon.

// it check the element from top of the stack.

{

if( top-p+1<0)

cout<<”\n wrong position \n”;

else

cout<<p<<” position element is “<<a[top-p+1];

}

void stack :: Change(int p , int x) // Changes the old value with the new value of stack

// p is a integer variable which takes position

//x is the new value that is to be replaced .

{

if(top-p+1<0)

cout<<”\n wrong position \n”;

else

{

a[top-p+1] = x;

cout<<”Changed element is”<<a[top-p+1];

}

}

void main()

{

stack s; // s is a stack variable, where stack is a class (i.e s is a class variable)

int op, k, l;

while(1)

{

cout<<”\n stack elements are:: \n”;

s.print(); // calls the print function to print the values of stack.

cout<<”\n 1.push\n 2.pop \n 3.Peep \n

4.Change \n

5. Exit \n “;

cout<<”\n Enter your option \n”;

cin>>op;

switch(op)

{

case 1: cout <<”\n Enter a number :: \n”;

cin>>k;

s.push(k); // calls the push function of stack.

break;

case 2: k=s.pop(); // calls the pop function of stack.

cout<<k<<”is poped element \n”;

break;

case 3: cout<<”\n Enter position:”;

cin>>k;

s.Peep(k); // calls the peep function of stack.

break;

case 4: cout<<”\n Enter a number::”;

cin>>l;

cout<<”\n Enter position:”;

cin>>k;

s.Change(k,l);//calls the Change function of stack.

break;

case 5: exit(0);

}// end of switch case;

getch();

}//end of while loop;

}//end of main loop;

**Output of the program:**

Enter the size of the stack::5

Stack is empty

1. Push

2. Pop

3. Peep

4. Change

5. Exit

Enter your choice: 1

Enter a number56

56 is pushed into stack

STACK ELEMENTS ARE...

56

1. Push

2. Pop

3. Peep

4. Change

5. Exit

Enter your choice: 1

Enter a number78

78 is pushed into stack

STACK ELEMENTS ARE....

78

56

1. Push

2. Pop

3. Peep

4. Change

5. Exit

Enter your choice: 1

Enter a number90

90 is pushed into stack

STACK ELEMENTS ARE....

90

78

56

1. Push

2. Pop

3. Peep

4. Change

5. Exit

Enter your choice: 1

Enter a number 54

54 is pushed into stack

STACK ELEMENTS ARE....

54

90

78

56

1. Push

2. Pop

3. Peep

4. Change

5. Exit

Enter your choice: 2

54 is popped element

STACK ELEMENTS ARE....

90

78

56

1. Push

2. Pop

3. Peep

4. Change

5. Exit

Enter your choice: 1

Enter a number34

34 is pushed into stack

STACK ELEMENTS ARE....

34

90

78

56

1. Push

2. Pop

3. Peep

4. Change

5. Exit

Enter your choice: 3

Enter position::2

2 position elements is 90

STACK ELEMENTS ARE....

34

90

78

56

1. Push

2. Pop

3. Peep

4. Change

5. Exit

Enter your choice: 4

Enter a number: 45

Enter position: 3

Changed element is 45

STACK ELEMENTS ARE....

34

90

45

56

1. Push

2. Pop

3. Peep

4. Change

5. Exit

Enter your choice:5 // exit from the operations

**Write a c++ program to implement Stack ADT using singly linked list.**

**Aim:** A c++ program to implement a stack and perform the operations like push, Pop, peep and Change using singly linked list.

**Abstract Data Type:**

1. Stack is a data structure which allows insertion and deletion at the same end.

1. Inserting elements into a stack is called push.
2. Deleting an element from the stack is called pop.
3. Retrieving an element by given position number is called peep.
4. Updating an old element by new element by given position number is called

Change.

1. In stack each and every operation can do the Changes by using top values.
2. In stack first inserted element is popped out last and last inserted element is

Popped out first.

1. Stack is last in First out Data Structure (i.e. LIFO).
2. For each push and pop operations the value of top will change.
3. In stacks using linked list the top most element will points to header pointer.
4. Stacks using linked list is a dynamic data structure. There is no limit for maximum

Number of elements in stack.

1. NODE is a data object,it has two fields ,one field is used to store data and another

field is used to store address of the nextnode.

**Member functions:**

**1.** **Push (d):** this function insert given element into the stack. If push is successful then it

Returns true else it return false.

1. **Pop (d):** this function deletes an element which is inserted last into the stack. if pop is

Successful then it returns true else it return false.

**3. Peep ():** this function searches the element from the top and show the element of the

Stack in the given position. If peep is successful then it returns true else it

Return “wrong position”.

**4. Change ():** this function updates the old element in the stack with new element of the

Stack in the given position. If the Change is successful

**5. Size ():** this function shows the size of the stack.

**Algorithm:**

**1. Push (d):**

1. start

2. \*n is a new node

3. read num

4. n -> d = num

5. n -> next = top

6. top = n

7. print num

8. print “ is pushed into stack “

1. stop.

**2.pop():**

1. start

2. if top = = NULL then

3. print “underflow”

4. num = top -> d

5. \*d is the new node

6. d = top

7. top = top -> next;

8. delete d

9. stop.

**3.Peep ():**

1. start

2. read pos // position

3. c = call the size function

4. if pos is less than 1 or pos is less than c then

5. print “wrong position”

6. \*a is the new node

7. a = top

8. for i = 0 to pos-1

9. a = a -> next

10. i = i + 1

11. print pos

12. print “element is “

13. print a - > d

14. stop

**4.Change ():**

1. start

2. \*a is new node ;

3. a = top

4. read pos // position

5. c = call the size function

6. if pos is less than 1 or pos is less than c then

7. print “wrong position”

8. read num

9. for i = 0 to pos-1

10. a = a -> next

11. a -> d = num

12. i = i + 1

13. print “number is changed “

14. stop

**/\* Program::3: to perform Stack Operations using Linked List. \*/**

#include<iostream.h>

#include<process.h>

#include<conio.h>

class node

{

int d;

node \*next;

friend class stack;

};

class stack

{

node \*top;

public:

stack() // it is a default constructor which is called

// when the class variable is declared.

{

top = NULL;

}

void push();

int pop();

void print();

void peep();

void change();

int size ();

};

void stack :: push() // inserts the elements into the stack.

{

int num;

node \*n;

cout<<”\n enter a number::”;

cin>>num;

n = new node;

n -> d = num;

n - > next = top;

top = n;

cout<<num<<” is pushed into the stack \ n”;  
}

int stack :: pop() // deletes the elements from the stack.

{

if(top= = NULL)

{

cout<<”\n stack underflow \n”;

return 0;

}

int num = top -> d ;

node \*d = top;

top = top -> next;

delete d;

return num;

}

void stack :: print() // prints the elements of the stack.

{

node \*a = top;

while ( a ! = NULL)

{

cout<<”\n “<<a -> d;

a = a -> next;

}

}

void stack :: peep() // searches the element of the stack for the given position.

// p is a integer variable which takes positon.

// it check the element from top of the stack.

{

int pos,i,c=size();

cout<<”\n enter position::”;

cin>> pos;

if(pos < 1|| pos > c)

cout<<”\n wrong position \n”;

node \*a = top;

for( i= 0;i<pos-1;i++)

a = a->next;

cout<<pos<<” position element is “<<a ->d;

}

void stack :: Change() // Changes the old value with the new value of stack

// at the given position ‘p’ is a integer variable

{

int pos,i,num,c = size();

node \*a = top;

cout<<”\n enter the position::”;

cin >>pos

if( pos < 1 || pos > c)

cout<<”\n wrong position \n”;

cout<<”\n enter a number:”;

cin>>num;

for(i=0;i<pos-1;i++)

a = a -> next;

a -> d = num;

cout<<”\n number is changed\n”;

}

void stack :: size() // returns the stack size

{

node \*a = top;

int i=0;

while(a!=NULL)

{

i++;

a = a -> next;

}

return i

}

void main()

{

stack s; // s is a stack variable, where stack is a class (i.e s is a class variable)

int op, num;

while(1);

{

cout<<”\n stack elements are:: \n”;

s.print(); // calls the print function to print the values of stack.

cout<<”\n 1.push\n

2.pop \n

3.Peep \n

4.Change \n

5. size \n “;

cout<<”\n Enter your option \n”;

cin>>op;

switch(op)

{

case 1: s.push(); // calls the push function of stack.

break;

case 2: num=s.pop(); // calls the pop function of stack.

cout<<num<<”is poped element \n”;

break;

case 3: s.peep(); // calls the peep function of stack.

break;

case 4: s.Change();//calls the Change function of stack.

break;

case 5: cout<<”\n Size of the stack:”<<s.size();

break;

default: exit(0);

}// end of switch case;

getch();

}//end of while loop;

}//end of main loop;s

**Output of the program:**

stack::

1.push

2.pop

3.peep

4.change

5.size

enter ur option:: 1

enter a number: 25

25is pushed into the stack

stack:: 25

1.push

2.pop

3.peep

4.change

5.size

enter ur option::1

enter a number: 69

69is pushed into the stack

stack::

69

25

1.push

2.pop

3.peep

4.change

5.size

enter ur option::1

enter a number: 79

79is pushed into the stack

stack::

79

69

25

1.push

2.pop

3.peep

4.change

5.size

enter ur option::1

enter a number: 98

98is pushed into the stack

stack::

98

79

69

25

1.push

2.pop

3.peep

4.change

5.size

enter ur option:: 3

enter position: 4

4 element is 25

stack::

98

79

69

25

1.push

2.pop

3.peep

4.change

5.size

enter ur option:: 4

enter position:: 4

enter a number: 87

25 number is changed

stack::

98

79

69

87

1.push

2.pop

3.peep

4.change

5.size

enter ur option:: 5

size of the stack :: 4.

Enter your option :: 0 // exit from the operations.

**b) Write a Program to Implement the Operations of Double Linked Lists**

**AIM:**

This C++ Program demonstrates operations on doubly linked list.

Here is source code of the C++ Program to demonstrate doubly single linked list. The C++ program is successfully compiled and run on a Linux system. The program output is also shown below.

**Program:**

/\* C++ Program to Implement Doubly Linked List \*/

#include<iostream>

#include<cstdio>

#include<cstdlib>

/\*

\* Node Declaration

\*/

using namespace std;

struct node

{

int info;

struct node \*next;

struct node \*prev;

}\*start;

/\*

Class Declaration

\*/

class double\_llist

{

public:

void create\_list(int value);

void add\_begin(int value);

void add\_after(int value, int position);

void delete\_element(int value);

void search\_element(int value);

void display\_dlist();

void count();

void reverse();

double\_llist()

{

start = NULL;

}

};

/\*

\* Main: Conatins Menu

\*/

int main()

{

int choice, element, position;

double\_llist dl;

while (1)

{

cout<<endl<<"----------------------------"<<endl;

cout<<endl<<"Operations on Doubly linked list"<<endl;

cout<<endl<<"----------------------------"<<endl;

cout<<"1.Create Node"<<endl;

cout<<"2.Add at begining"<<endl;

cout<<"3.Add after position"<<endl;

cout<<"4.Delete"<<endl;

cout<<"5.Display"<<endl;

cout<<"6.Count"<<endl;

cout<<"7.Reverse"<<endl;

cout<<"8.Quit"<<endl;

cout<<"Enter your choice : ";

cin>>choice;

switch ( choice )

{

case 1:

cout<<"Enter the element: ";

cin>>element;

dl.create\_list(element);

cout<<endl;

break;

case 2:

cout<<"Enter the element: ";

cin>>element;

dl.add\_begin(element);

cout<<endl;

break;

case 3:

cout<<"Enter the element: ";

cin>>element;

cout<<"Insert Element after postion: ";

cin>>position;

dl.add\_after(element, position);

cout<<endl;

break;

case 4:

if (start == NULL)

{

cout<<"List empty,nothing to delete"<<endl;

break;

}

cout<<"Enter the element for deletion: ";

cin>>element;

dl.delete\_element(element);

cout<<endl;

break;

case 5:

dl.display\_dlist();

cout<<endl;

break;

case 6:

dl.count();

break;

case 7:

if (start == NULL)

{

cout<<"List empty,nothing to reverse"<<endl;

break;

}

dl.reverse();

cout<<endl;

break;

case 8:

exit(1);

default:

cout<<"Wrong choice"<<endl;

}

}

return 0;

}

/\*

\* Create Double Link List

\*/

void double\_llist::create\_list(int value)

{

struct node \*s, \*temp;

temp = new(struct node);

temp->info = value;

temp->next = NULL;

if (start == NULL)

{

temp->prev = NULL;

start = temp;

}

else

{

s = start;

while (s->next != NULL)

s = s->next;

s->next = temp;

temp->prev = s;

}

}

/\*

\* Insertion at the beginning

\*/

void double\_llist::add\_begin(int value)

{

if (start == NULL)

{

cout<<"First Create the list."<<endl;

return;

}

struct node \*temp;

temp = new(struct node);

temp->prev = NULL;

temp->info = value;

temp->next = start;

start->prev = temp;

start = temp;

cout<<"Element Inserted"<<endl;

}

/\*

\* Insertion of element at a particular position

\*/

void double\_llist::add\_after(int value, int pos)

{

if (start == NULL)

{

cout<<"First Create the list."<<endl;

return;

}

struct node \*tmp, \*q;

int i;

q = start;

for (i = 0;i < pos - 1;i++)

{

q = q->next;

if (q == NULL)

{

cout<<"There are less than ";

cout<<pos<<" elements."<<endl;

return;

}

}

tmp = new(struct node);

tmp->info = value;

if (q->next == NULL)

{

q->next = tmp;

tmp->next = NULL;

tmp->prev = q;

}

else

{

tmp->next = q->next;

tmp->next->prev = tmp;

q->next = tmp;

tmp->prev = q;

}

cout<<"Element Inserted"<<endl;

}

/\*

\* Deletion of element from the list

\*/

void double\_llist::delete\_element(int value)

{

struct node \*tmp, \*q;

/\*first element deletion\*/

if (start->info == value)

{

tmp = start;

start = start->next;

start->prev = NULL;

cout<<"Element Deleted"<<endl;

free(tmp);

return;

}

q = start;

while (q->next->next != NULL)

{

/\*Element deleted in between\*/

if (q->next->info == value)

{

tmp = q->next;

q->next = tmp->next;

tmp->next->prev = q;

cout<<"Element Deleted"<<endl;

free(tmp);

return;

}

q = q->next;

}

/\*last element deleted\*/

if (q->next->info == value)

{

tmp = q->next;

free(tmp);

q->next = NULL;

cout<<"Element Deleted"<<endl;

return;

}

cout<<"Element "<<value<<" not found"<<endl;

}

/\*

\* Display elements of Doubly Link List

\*/

void double\_llist::display\_dlist()

{

struct node \*q;

if (start == NULL)

{

cout<<"List empty,nothing to display"<<endl;

return;

}

q = start;

cout<<"The Doubly Link List is :"<<endl;

while (q != NULL)

{

cout<<q->info<<" <-> ";

q = q->next;

}

cout<<"NULL"<<endl;

}

/\*

\* Number of elements in Doubly Link List

\*/

void double\_llist::count()

{

struct node \*q = start;

int cnt = 0;

while (q != NULL)

{

q = q->next;

cnt++;

}

cout<<"Number of elements are: "<<cnt<<endl;

}

/\*

\* Reverse Doubly Link List

\*/

void double\_llist::reverse()

{

struct node \*p1, \*p2;

p1 = start;

p2 = p1->next;

p1->next = NULL;

p1->prev = p2;

while (p2 != NULL)

{

p2->prev = p2->next;

p2->next = p1;

p1 = p2;

p2 = p2->prev;

}

start = p1;

cout<<"List Reversed"<<endl;

}

**OUTPUT:**

---------------------------------

Operations on Doubly linked list

---------------------------------

1.Create Node

2.Add at begining

3.Add after

4.Delete

5.Display

6.Count

7.Reverse

8.Quit

Enter your choice : 2

Enter the element: 100

First Create the list.

---------------------------------

Operations on Doubly linked list

---------------------------------

1.Create Node

2.Add at begining

3.Add after

4.Delete

5.Display

6.Count

7.Reverse

8.Quit

Enter your choice : 3

Enter the element: 200

Insert Element after postion: 1

First Create the list.

---------------------------------

Operations on Doubly linked list

---------------------------------

1.Create Node

2.Add at begining

3.Add after

4.Delete

5.Display

6.Count

7.Reverse

8.Quit

Enter your choice : 4

List empty,nothing to delete

---------------------------------

Operations on Doubly linked list

--------------------------------

1.Create Node

2.Add at begining

3.Add after

4.Delete

5.Display

6.Count

7.Reverse

8.Quit

Enter your choice : 5

List empty,nothing to display

---------------------------------

Operations on Doubly linked list

---------------------------------

1.Create Node

2.Add at begining

3.Add after

4.Delete

5.Display

6.Count

7.Reverse

8.Quit

Enter your choice : 6

Number of elements are: 0

---------------------------------

Operations on Doubly linked list

--------------------------------

1.Create Node

2.Add at begining

3.Add after

4.Delete

5.Display

6.Count

7.Reverse

8.Quit

Enter your choice : 7

List empty,nothing to reverse

---------------------------------

Operations on Doubly linked list

--------------------------------

1.Create Node

2.Add at begining

3.Add after

4.Delete

5.Display

6.Count

7.Reverse

8.Quit

Enter your choice : 1

Enter the element: 100

---------------------------------

Operations on Doubly linked list

--------------------------------

1.Create Node

2.Add at begining

3.Add after

4.Delete

5.Display

6.Count

7.Reverse

8.Quit

Enter your choice : 5

The Doubly Link List is :

100 <-> NULL

---------------------------------

Operations on Doubly linked list

---------------------------------

1.Create Node

2.Add at begining

3.Add after

4.Delete

5.Display

6.Count

7.Reverse

8.Quit

Enter your choice : 2

Enter the element: 200

Element Inserted

---------------------------------

Operations on Doubly linked list

--------------------------------

1.Create Node

2.Add at begining

3.Add after

4.Delete

5.Display

6.Count

7.Reverse

8.Quit

Enter your choice : 5

The Doubly Link List is :

200 <-> 100 <-> NULL

---------------------------------

Operations on Doubly linked list

---------------------------------

1.Create Node

2.Add at begining

3.Add after

4.Delete

5.Display

6.Count

7.Reverse

8.Quit

Enter your choice : 3

Enter the element: 50

Insert Element after postion: 2

Element Inserted

---------------------------------

Operations on Doubly linked list

---------------------------------

1.Create Node

2.Add at begining

3.Add after

4.Delete

5.Display

6.Count

7.Reverse

8.Quit

Enter your choice : 5

The Doubly Link List is :

200 <-> 100 <-> 50 <-> NULL

---------------------------------

Operations on Doubly linked list

---------------------------------

1.Create Node

2.Add at begining

3.Add after

4.Delete

5.Display

6.Count

7.Reverse

8.Quit

Enter your choice : 3

Enter the element: 150

Insert Element after postion: 3

Element Inserted

---------------------------------

Operations on Doubly linked list

---------------------------------

1.Create Node

2.Add at begining

3.Add after

4.Delete

5.Display

6.Count

7.Reverse

8.Quit

Enter your choice : 5

The Doubly Link List is :

200 <-> 100 <-> 50 <-> 150 <-> NULL

---------------------------------

Operations on Doubly linked list

---------------------------------

1.Create Node

2.Add at begining

3.Add after

4.Delete

5.Display

6.Count

7.Reverse

8.Quit

Enter your choice : 6

Number of elements are: 4

---------------------------------

Operations on Doubly linked list

---------------------------------

1.Create Node

2.Add at begining

3.Add after

4.Delete

5.Display

6.Count

7.Reverse

8.Quit

Enter your choice : 4

Enter the element for deletion: 50

Element Deleted

---------------------------------

Operations on Doubly linked list

---------------------------------

1.Create Node

2.Add at begining

3.Add after

4.Delete

5.Display

6.Count

7.Reverse

8.Quit

Enter your choice : 5

The Doubly Link List is :

200 <-> 100 <-> 150 <-> NULL

---------------------------------

Operations on Doubly linked list

---------------------------------

1.Create Node

2.Add at begining

3.Add after

4.Delete

5.Display

6.Count

7.Reverse

8.Quit

Enter your choice : 6

Number of elements are: 3

---------------------------------

Operations on Doubly linked list

---------------------------------

1.Create Node

2.Add at begining

3.Add after

4.Delete

5.Display

6.Count

7.Reverse

8.Quit

Enter your choice : 7

List Reversed

---------------------------------

Operations on Doubly linked list

---------------------------------

1.Create Node

2.Add at begining

3.Add after

4.Delete

5.Display

6.Count

7.Reverse

8.Quit

Enter your choice : 5

The Doubly Link List is :

150 <-> 100 <-> 200 <-> NULL

---------------------------------

Operations on Doubly linked list

---------------------------------

1.Create Node

2.Add at begining

3.Add after

4.Delete

5.Display

6.Count

7.Reverse

8.Quit

Enter your choice : 3

Enter the element: 200

Insert Element after postion: 100

There are less than 100 elements.

---------------------------------

Operations on Doubly linked list

---------------------------------

1.Create Node

2.Add at begining

3.Add after

4.Delete

5.Display

6.Count

7.Reverse

8.Quit

Enter your choice : 4

Enter the element for deletion: 150

Element Deleted

---------------------------------

Operations on Doubly linked list

---------------------------------

1.Create Node

2.Add at begining

3.Add after

4.Delete

5.Display

6.Count

7.Reverse

8.Quit

Enter your choice : 5

The Doubly Link List is :

100 <-> 200 <-> NULL

---------------------------------

Operations on Doubly linked list

---------------------------------

1.Create Node

2.Add at begining

3.Add after

4.Delete

5.Display

6.Count

7.Reverse

8.Quit

Enter your choice : 8

**Week 2**

**a) Write a C program that uses stack operations to convert a given infix expression into its**

**postfix**

**b) Write a Program to Implement Queue Operations by using Array and Linked Lists.**

a)**AIM:** C++ program to perform infix to postfix conversion

**Program:**

#include<constream.h>

#include<iostream.h>

#include<process.h>

#include<string.h>

#define size 50

char stack[size];

int tos=0,ele;

void push(int);

char pop();

int prec(char);

char infix[30],output[30];

void main()

{

int i=0,j=0,length;

char temp;

clrscr();

cout<<"\nEnter an infix expression:";

cin>>infix;

length=strlen(infix);

for(i=0;i<length;i++)

{

if(infix[i]!='+'&&infix[i]!='-'&&infix[i]!='/'&&infix[i]!='%'&&infix[i]!='^'

&&infix[i]!='\*'&&infix[i]!=')'&&infix[i]!='(')

{

output[j++]=infix[i];

}

else

{

if(tos==0)

push(infix[i]);

else

{

if(infix[i]!=')'&&infix[i]!='(')

{

if(prec(infix[i])<=prec(stack[tos-1]))

{

temp=pop();

output[j++]=temp;

push(infix[i]);

}

else

push(infix[i]);

}

else

{

if(infix[i]=='(')

push(infix[i]);

if(infix[i]==')')

{

temp=pop();

while(temp!='(')

{

output[j++]=temp;

temp=pop();

}

}

}

}

}

}

while(tos!=0)

{

output[j++]=pop();

}

cout<<"\nPostfix expression is:"<<output;

}

void push(int ele)

{

stack[tos]=ele;

tos++;

}

char pop()

{

tos--;

return (stack[tos]);

}

int prec(char symbol)

{

if(symbol=='(')

return 0;

if(symbol==')')

return 0;

if(symbol=='+'||symbol=='-')

return 1;

if(symbol=='\*'||symbol=='/'||symbol=='%')

return 2;

if(symbol=='^')

return 3;

return 0;

}

**Output:**

Infix Expression: 1+2\*3+4\*6

Postfix: 123\*+46\*+

1. **Write a Program to Implement Queue Operations by using Array and Linked Lists.**

**Aim:**

A c++ program to implement the queue and perform the operations like insertion,

Deletion using arrays.

**Abstract Data Type:**

1. Queue is a data structure where insertion is done through one end and deletion

At another end the end through insertion is called rear and the end through

Deletion is called front.

2. Initially front and rear values are ‘-1’.

3. In Queues first inserted element is deleted first and last element is deleted last.

4. Queue is called first in first out data structure (i.e. FIFO).

5. The element which is to be inserted at the end of the queue is known as the

“REAR” of the queue and other end where as the deletions takes place is called

the “FRONT” of the queue.Front and Rear are the two pointers to represent the queue.

1. Queue is said to be empty when ‘front’ value becomes greater than the ‘rear’ value.

**Member functions:**

**1. Insertion ():** this function insert given element into the queue. If insertion is successful

then it returns true else it return false.

**2. deletion ():** this function deletes an element which is inserted first into the queue. If

deletion is successful then it returns true else it return false.

**Algorithm:**

**1. Insertion ():**

1 .start

2. read n

3. if rear = = n -1 then

4. print “ overflow”

5. print “enter a number:”

6. read num

7. if front = = -1 then

8. front = rear = 0;

9. else

10. rear = rear +1

11. a[rear] = num

12. print “ number is inserted”

13. stop.

**2. Deletion ():**

1. start

2. if front = = -1 then

3. print “underflow”

4. print ”deleted element is”

5. print a[front]

6. if front = = rear then

7. front = rear = -1.

8. else

9. front = front + 1

10.print “ number is deleted”

11. stop

**/\* Program::2: to perform Queue Operations using arrays. \*/**

#include<iostream.h>

#include<process.h>

#include<conio.h>

class queue

{

int \*a, font, rear, n;

public:

queue () // it is a default constructor which is called

// when the class variable is declared.

{

clrscr ();

front = rear = -1;

cout<<”\n Enter the size of the queue:”;

cin>>n;

a=new int [n];

}

void insertion ();

void deletion ();

void print ();

};

void queue:: insertion () // inserts the elements into the queue using rear pointer.

{

int num;

if (rear = = n-1)

{

cout<<”\n queue overflow \n”;

return;

}

cout<<”\n enter a number:”;

cin>>num;

if( front = = -1)

front = rear = 0;

else

rear++;

a[rear] = num;

cout<<num<<”number is inserted into queue \n”;

}

void queue :: deletion () // deletes the elements from the queue using front pointer.

{

if(front = = -1)

{

cout<<”\n queue underflow \n”;

return ;

}

cout<<” \n deleted element is “<<a[front]<<”\n”;

if (front = = rear)

front = rear = -1

else

front ++;

cout<<”\n number is deleted\n”;

}

void queue :: print() // prints the elements of the queue.

{

if(front = = -1)

cout<<”QUEUE IS EMPTY:”;

for(int i=front;i<=rear;i++)

cout<<”\n”<<a[i]<<”\n”;

}

void main()

{

queue q; // q is a queue variable, where queue is a class (i.e s is a class variable)

// q calls the constructor declared in the class defaultly.

int op;

clrscr ();

while(1);

{

cout<<”\n QUEUE ELEMENTS ARE:: \n”;

q.print(); // calls the print function to print the values of queue.

cout<<”\n 1.insertion \n

2.deletion \n

3. Exit \n “;

cout<<”\n Enter your option \n”;

cin>>op;

switch(op)

{

case 1: q.insertion (); // calls the insertion function of queue.

break;

case 2: q.deletion (); // calls the pop function of stack.

break;

case 3: exit(0);

}// end of switch case;

getch();

}//end of while loop;

}//end of main loop;

**Output of the program:**

Enter the size of the queue::5

QUEUE IS EMPTY

1.insertion

2.deletion

3.exit

enter your option::1

enter a number:34

number is inserted

QUEUE ELEMENTS ARE

34

1.insertion

2.deletion

3.exit

enter your option::1

enter a number:56

number is inserted

QUEUE ELEMENTS ARE

34 56

1.insertion

2.deletion

3.exit

enter your option:: 1

enter a number:67

number is inserted

QUEUE ELEMENTS ARE

34 56 67

1.insertion

2.deletion

3.exit

enter your option::1

enter a number:87

number is inserted

QUEUE ELEMENTS ARE

34 56 67 87

1.insertion

2.deletion

3.exit

enter your option::2

deleted element is 34

number is deleted

QUEUE ELEMENTS ARE

56 67 87

1.insertion

2.deletion

3.exit

enter your option:: 3 // Exit from the operations.

**Write a c++ program to implement Queue ADT using linked list.**

**Aim:**

A c++ program to implement the Queue and perform the operations like insertion,

Deletion using linked list.

**Abstract Data Type:**

1. Queue is a data structure where insertion is done at one end and deletion

At other end, the end through insertion is called rear and the end through

Deletion is called front.

2. if an item is inserted into the queue then rear is going to be incremented by one.

3. In Queues first inserted element is deleted first and last element is deleted last.

4. Queue is called first in first out data structure (i.e. FIFO).

5. The element which is to be inserted at the end of the queue is known as the

“REAR” of the queue and other end where as the deletions takes place is called

the “FRONT” of the queue.Front and Rear are the two pointers to represent the queue.

6. Queue is said to be empty when ‘front’ value becomes greater than the ‘rear’ value.

1. NODE id a data object, which contains two parts.One part is used to store the data

and another part is to store the address of the next node.

**Member functions:**

**1. Insertion ():** this function insert given element into the queue. If insertion is successful

then it returns true else it return false.

**2. deletion ():** this function deletes an element which is inserted first into the queue. If

deletion is successful then it returns true else it return false.

**Algorithm:**

**1. Insertion ():**

1. start

2. \*link is a new node

3. read num

4. \*n is a new node

5. n -> data = num //Storing the data

6. n -> link = NULL // storing the address of the next node.

7. if front is equal to NULL then

7.1. front = rear = n

7.2. print “number is inserted”

7.3.return.

8. rear -> link = n;

9. rear = n;

10. print “ number is inserted”.

**2. Deletion ():**

1. start

2. \*d is a new node

3. if front is equal to NULL then

3.1. print”underflow”

3.2. return

4. if front is equal to rear then

4.1. front = rear = NULL

4.2. print “ number is deleted”

4.3. return

5. d = front

6. front = front -> link // assign the address of the next

// node to front pointer.

7.print d->data

8. ptint “is deleted”

9. delete d

10.stop.

**/\* Program::4:to perform Queue Operations using linked list. \*/**

#include<iostream.h>

#include<process.h>

#include<conio.h>

class node

{

int data;

node \*link;

friend class queue;

};

class queue

{

node \*font, \*rear;

public:

queue () // it is a default constructor which is called

// when the class variable is declared.

{

front = rear = NULL;

}

void insertion ();

void deletion ();

void print ();

};

void queue:: insertion () // inserts the elements into the queue using rear pointer.

{

int num;

node \*n;

cout<<”\n enter a number:”;

cin>>num;

n = new node;

n -> data = num;

n -> link = NULL;

if( front = = NULL)

{

front = rear = NULL;

cout<<” number is inserted”;

return;

}

rear -> link = n;

rear = n;

cout<<”number is inserted into queue \n”;

}

void queue :: deletion () // deletes the elements from the queue using front pointer.

{

node \*d;

if(front = = NULL)

{

cout<<”\n queue underflow \n”;

return ;

}

if (front = = rear)

{

front = rear = NULL;

cout<<”\n number is deleted\n”;

return;

}

d = front;

front = front -> likn;

cout<<d - > data<<” is deleted\n”;

delete d;

}

void queue :: print() // prints the elements of the queue.

{

node \*a = front;

while(a!=NULL)

{

cout<<”\n “<<a->data;

a= a-> link;

}

}

void main()

{

queue q; // q is a queue variable, where queue is a class (i.e s is a class variable)

// q calls the constructor declared in the class defaultly.

int op;

clrscr ();

while(1);

{

cout<<”\n QUEUE ELEMENTS ARE:: \n”;

q.print(); // calls the print function to print the values of queue.

cout<<”\n 1.insertion \n

2.deletion \n

3. Exit \n “;

cout<<”\n Enter your option \n”;

cin>>op;

switch(op)

{

case 1: q.insertion (); // calls the insertion function of queue.

break;

case 2: q.deletion (); // calls the pop function of stack.

break;

case 3: exit(0);

}// end of switch case;

getch();

}//end of while loop;

}//end of main loop;

**Output of the program:**

QUEUE ELEMENTS ARE::

1.insertion

2.deletion

3.exit

enter your option::1

enter number:25

number is inserted

QUEUE ELEMENTS ARE::

25

1.insertion

2.deletion

3.exit

enter your option::1

enter number:56

number inserted

QUEUE ELEMENTS ARE::

25 56

1.insertion

2.deletion

3.exit

enter your option::1

enter number:89

number inserted

QUEUE ELEMENTS ARE::

25 56 89

1.insertion

2.deletion

3.exit

enter your option::1

enter number: 56

number inserted

QUEUE ELEMENTS ARE::

25 56 89 56

1.insertion

2.deletion

3.exit

enter your option::1

enter number:89

number inserted

QUEUE ELEMENTS ARE::

25 56 89 56 89

1.insertion

2.deletion

3.exit

enter your option::1

enter number:89

number inserted

QUEUE ELEMENTS ARE::

25 56 89 56 89 89

1.insertion

2.deletion

3.exit

enter your option::2

25is deleted

QUEUE ELEMENTS ARE::

56 89 56 89 89

1.insertion

2.deletion

3.exit

enter your option::2

56is deleted

QUEUE ELEMENTS ARE::

89 56 89 89

1.insertion

2.deletion

3.exit

enter your option:: 3 //exit from the operations or loop.

**Week 3**

**AIM:** Write a Program to Implement Circular Queue Operations by using Array and Linked Lists.

**PROGRAM:**

#include<iostream.h>

#include<conio.h>

#include<stdlib.h>

class cqueue

{

int q[5],front,rare;

public:

cqueue()

{

front=-1;

rare=-1;

}

void push(int x)

{

if(front ==-1 && rare == -1)

{

q[++rare]=x;

front=rare;

return;

}

else if(front == (rare+1)%5 )

{

cout <<" Circular Queue over flow";

return;

}

rare= (rare+1)%5;

q[rare]=x;

}

void pop()

{

if(front==-1 && rare== -1)

{

cout <<"under flow";

return;

}

else if( front== rare )

{

front=rare=-1;

return;

}

front= (front+1)%5;

}

void display()

{

int i;

if( front <= rare)

{

for(i=front; i<=rare;i++)

cout << q[i]<<" ";

}

else

{

for(i=front;i<=4;i++)

{

cout <<q[i] << " ";

}

for(i=0;i<=rare;i++)

{

cout << q[i]<< " ";

}

}

}

};

main()

{

int ch;

cqueue q1;

while( 1)

{

cout<<"\n1.INSERT 2.DELETE 3.DISPLAY 4.EXIT\nEnter ur choice";

cin >> ch;

switch(ch)

{

case 1: cout<<"enter element";

cin >> ch;

q1.push(ch); break;

case 2: q1.pop(); break;

case 3: q1.display(); break;

case 4: exit(0);

}

}

}

**OUTPUT:**

1.INSERT 2.DELETE 3.DISPLAY 4.EXIT

Enter ur choice1

enter element4

1.INSERT 2.DELETE 3.DISPLAY 4.EXIT

Enter ur choice1

enter element5

1.INSERT 2.DELETE 3.DISPLAY 4.EXIT

Enter ur choice1

enter element3

1.INSERT 2.DELETE 3.DISPLAY 4.EXIT

Enter ur choice3

4 5 3

1.INSERT 2.DELETE 3.DISPLAY 4.EXIT

Enter ur choice2

1.INSERT 2.DELETE 3.DISPLAY 4.EXIT

Enter ur choice3

5 3

1.INSERT 2.DELETE 3.DISPLAY 4.EXIT

Enter ur choice4

**AIM:** Write a program In C++ To Implement Circular Queue Using Linked List

A circular queue is a particular implementation of a queue. It is very efficient. It is also quite useful in low level code, because insertion and deletion are totally independant, which means that you don’t have to worry about an interrupt handler trying to do an insertion at the same time as your main code is doing a deletion.

How does it work?

A circular queue consists of an array that contains the items in the queue, two array indexes and an optional length. The indexes are called the head and tail pointers and are labelled H and T on the diagram.

A circular queue

The head pointer points to the first element in the queue, and the tail pointer points just beyond the last element in the queue. If the tail pointer is before the head pointer, the queue wraps around the end of the array.

**PROGRAM:**

/\* C++ To Implement Circular Queue Using Linked List\*/

#include<iostream.h>

#include<conio.h>

#include<process.h>

class circlst //Class Declaration

{

private:

struct node

{ int info; //Node Structure

node \*next;

}\*ptr,\*first,\*last,\*track;

public:

circlst() //Constructor

{ptr=first=last=track=NULL;}

void insrt(int);

void delnde();

void displst(); //Memeber Functions

};

void circlst::insrt(int n)

{

ptr=new node;

if(ptr==NULL)

{ cout<<"\nNode Cannot Be Created!!!";

getch();

exit(0);

}

ptr->info=n;

ptr->next=ptr; //Function To Insert Node

if(first==NULL)

first=last=ptr;

else

{ last->next=ptr;

ptr->next=first;

last=ptr;

}

}

void circlst::delnde()

{ if(first==NULL)

{ cout<<"\nThe List Is Empty!!!";

getch();

exit(0);

} //Function To Delete Node

first=first->next;

last->next=first;

cout<<"\n\nNode Deleted Succesfully!!!";

}

void circlst::displst()

{ track=first;

if(first==NULL)

cout<<"\nThe List Is Empty!!!";

while((track->next)!=first)

{ //Function To Delete Node

track=track->next;

cout<<track->info<<" ";

}

}

void main() //Main

{

circlst ob;

char choice;

int data,ch;

do

{ clrscr();

cout<<"\n\n\t\tCREATION OF A CIRCULAR LINKED LIST";

cout<<"\n\t\t---------------------------------";

cout<<"\n\n1.INSERT ELEMENT";

cout<<"\n2.DISPLAY LIST";

cout<<"\n3.DELETE ELEMENT"; //menu

cout<<"\n4.EXIT";

cout<<"\n\nWhich Operation Do You Want To Perform:";

cin>>ch;

switch(ch)

{

case 1: cout<<"\nEnter the Data To Be Inserted:";

cin>>data;

ob.insrt(data);

break;

case 2: ob.displst();

break; //calling functions

case 3: ob.delnde();

break;

case 4: exit(1);

default: cout<<"\nPleasae Enter A valid Choice(1-4)!!!";

}

cout<<"\n\nDo you Want to Continue(Y/N):";

cin>>choice;

}while(choice=='Y' || choice=='y');

} //end of program

**Week 4**

**Write a Program to Sort the set of elements by using**

**i) Quick Sort ii) Heap Sort. iii) Merge Sort**

Aim:

A c++ program to implement the quick sort program.

Abstract Data Type:

1.quick sort is a recursive function does three things.

* 1. it partitions the array into two groups.
  2. calls itself to sort the left group.
  3. calls itself to sort the right group.

1. Quick sort is a very efficient sorting algorithm invented

by C.A.R hoare.

3. it has two phases: the partition phase(dividing the work).

the sort phase(sorting of two partitions that are divided

by partition phase)

4. Quick sort works on the principle of divide and conquer.

5. in Quick sort the groups are formed based on the key element.

6. the elements that are smaller than key elements are placed in

one group and the elements that are bigger than key elements

are placed on another group.

**Algorithm:**

1.start

[initializes]

flag = 1

2. (start the sorting process)

qsort(q, l, u)

2.1 if l is less than u

2.2 i=l

2.3 j=u

2.4 key = q.a[i]

2.5 repeat while( flag)

2.5.1. i=i+1

2.5.2. repeat while(q.a[i]<key &&i<u)

2.5.2.1. i=i+1

2.5.3. repeat while( q.a[j]>key)

2.5.3.1. j= j-1

2.5.4. if (i<j)

2.5.4.1. int t = q.a[i]

2.5.4.2.q.a[i] = q.a[j]

2.5.4.3. q.a[j]=t

2.5.4.4.else flag = 0

2.6. int t=q.a[l]

2.7. q.a[l]= q.a[j]

2.8. q.a[j] = t

2.9. qsort(q,l,j-1)

2.10. qsort(q,j+1,u)

1. stop

**Program : write a c++ program to implement quick sort.**

#include<iostream.h>

#include<iomanip.h>

#include<conio.h>

class qusort

{

int \*a;

public:

int n;

void read();

void print();

friend void qsort(qusort,int,int);

};

void qusort::read()

{

clrscr();

cout<<"\n enter range::";

cin>>n;

a=new int[n];

cout<<"\n enter elements that you want to sort::";

for(int i=0;i<n;i++)

cin>>a[i];

}

void qusort::print()

{

cout<<"\n sorted order is.....\n";

for(int i=0;i<n;i++)

cout<<a[i]<<setw(5);

}

void qsort(qusort q,int l,int u)

{

int i,j,key;

if(l>=u) return;

i=l;

j=u;

key=q.a[l];

while(1)

{

i++;

while(q.a[i]<=key && i<u)

i++;

while(q.a[j]>key)

j--;

if(i<j)

{

int t = q.a[i];

q.a[i]=q.a[j];

q.a[j]=t;

}

else break;

}

int t = q.a[l];

q.a[l]=q.a[j];

q.a[j]=t;

qsort(q,l,j-1);

qsort(q,j+1,u);

}

void main()

{

qusort q;

clrscr();

cout<<"\nenter the numbers for the sorting::";

q.read();

qsort(q,0,(q.n-1));

q.print();

getch();

}

**Output of the program:**

enter range::5

enter elements that you want to sort::

5

4

3

2

1

sorted order is.....

1 2 3 4 5

output2 of the program:

enter range::6

enter elements that you want to sort::

9

5

2

1

5

6

sorted order is.....

1 2 5 5 6 9

**(b) Write a c++ program to implement the heap sort.**

**Aim:**  A c++ program to implement the sorting technique i,e(heap sort)

**Abstract Data Type:**

1. heap sort is a tree based sorting technique.
2. in heap sort technique the parent node is always bigger than a child node.
3. **Building heap :**

To build heap structure for the given data list,we considererd child nodes as key elements.

1. The elements are compared with their corresponding parent elements to validate the heap conditions.
2. if the heap condition doesn’t satisfy then the parent is denoted to the child position& and the position of parent is considered as the new position for child.
3. Sorting procedure:

During the heap sort we need heap structure.

7. in each insertion first element in the heap structure and last element in the

heap structure is the biggest element in the unsorted list..

8. starting from the root heap validations are performed.

9. if heap conditions doesnot satisfy at any level child node is promoted to

the parent node.

10. after performing all iterations **the heap structures contains all sorted elements.**

**Member functions:**

1. **createheap ():** this function is used to build the heap structure.
2. **heapsort():** this function is used to perform the sorting the elements.
3. **display():** this function is used to print the elements in the sorted order

by using heap technique.

**Algorithm::**

**1.createheap():**

1.start

2.repeat step through I=0 to I<n

3.set val=a[I];

4.set s=I;

5.set f=(s-1)/s

6.repeat step while s>0 and a[I] <val

7. i.set a[s] =a[f]

ii.set s=f

iii.set f=(s-1)/2

8.set a[s]=val

9.end.

**2.heapsort():**

1**.**start.

2.repeat step through I=n-1 to I>1

3.set val1=a[I]

4.set a[I]=a[0]

5.set f=s=0

6.if I=1 then s=-1

else

s=1

7.if I>2 and a[2]>a[1] then s=2

8.repeat step while s>=0 and val1<a[s]

* + 1. set a[f]=a[s]
    2. set f=s
    3. set s=2\*(f+1)
    4. if(s+1<=I-1)and (a[s]<a[s+1]) then
    5. s++
    6. if(s>I-1) then s=-1

9.set a[f]=val1

10.end

**3.display():**

* + 1. start
    2. write “list elements are”
    3. repeat step through I=0 to I<n
    4. write a[I]
    5. end

**Program-: to write a c++ program to perform the heap sort.**

#include<iostream.h>

#include<conio.h>

class heap

{

private:

public:

void heapsort(int\*,int);

void createheap(int\*,int);

void display(int\*,int);

};

void heap::createheap(int list[],int n)

{

for(int k=2;k<=n;++k)

{

int i= k;

int temp =list[k];

int j=i/2;

while((i>1)&&(temp>list[j]))

{

list[i]=list[j];

i=j;

j=i/2;

if(j<1)

j=1;

}

list[i]=temp;

}

}

void heap::heapsort(int list[],int n)

{

for(int k=n;k>=2;--k)

{

int temp = list[1];

list[1]=list[k];

list[k]=temp;

int i=1;

int value=list[1];

int j=2;

if((j+1)<k)

if(list[j+1]>list[j])

j++;

while((j<=(k-1))&&(list[j]>value))

{

list[i] = list[j];

i=j;

j=2\*i;

if((j+1)<k)

if(list[j+1]>list[j])

j++;

else

if(j>n)

j=n;

list[i]=value;

}

cout<<"\n";

for(int p=1;p<=n;p++)

cout<<" "<<list[p];

}

}

void heap::display(int list[], int n)

{

for(int i=1;i<=n;++i)

{

cout<<list[i]<<"\t";

}

}

void main()

{

heap s;

int list[100];

int size;

clrscr();

cout<<"\n enter the input size::\n";

cin>>size;

for(int i=1;i<=size;++i)

{

cout<<"\n intput values::"<<i<<":";

cin>>list[i];

}

cout<<"\n entered list to be sorted are::\n";

s.display(list,size);

s.createheap(list,size);

cout<<"\n\n\n\n sortted list are::\n";

s.heapsort(list,size);

s.display(list,size);

getch();

}

**Output of the program:**

enter the input size::6

input values::1:8

input values::2:6

input values::3:4

input values::4:5

input values::5:2

input values::6:3

entered list to be sorted are::

8 6 4 5 2 3

sortted list are::

2 3 4 5 6 8

**Output2 of the program:**

enter the input size::6

input values::1:9

input values::2:6

input values::3:3

input values::4:2

input values::5:5

input values::6:8

entered list to be sorted are::

9 6 3 2 5 8

sortted list are::

2 3 5 6 8 9

**c) Write a c++ program to implement the mergesort.**

**Aim:**

To implement a c++ program for one of the sorting technique

i.e,(mergesort).

**Abstract Data Type:**

1.An array is divided in half and each half is stored separately.then the halves are merge.the process is carried out recursively for smaller and smaller arrays.

2.merge sort algorithm is based on divide-and-conquer algorithmic design pattern.

3.Divide:the input data is divided into two or more disjoint subsets.

4.Recur: recursively solve the subproblems associated with the subsets.

5.Conquer:take the solutions to the subproblems and “merge” then into a

solution to the original problem.

**Member functions:**

**1.mergesort():** the purpose of this function is to sort the divided list

**2.mergepass():** the purpose of this function is to divide the given list into

as small as we can do.

Algorithms:

**1.merge(k,first,second,third):**

1.initialization of the variables.

f = first

s = second

i = 0

2. repeat while(f<second and s<=third)

if (k[f] <=k[s]) then

{

i = i+1

temp[i] = k[f]

f = f+1

}

else

{

i=i+1

temp[i] = k[s]

s=s+1

}

* + 1. store the elements which are not processed

if(f>=second) then

repeat while(s <= third)

{

i= i+1

temp[i]=k[s]

s=s+1

}

else

repeat while( f<second)

{

i=i+1

temp[i]=k[f]

f=f+1

}

4. get back the elements from the temporary elements

repeat for i=1 to l do

k[first-1+i] = temp[i]

5. return.

Program-:- to write a c++ program to implement the merge sort technique.

#include <iostream.h>

#include<conio.h>

class merge

{

public:

void mergesort(float\*,int,int,int);

void mergepass(float\*,int,int);

};

void merge::mergesort(float l[],int top,int size,int bottom)

{

float temp[100];

int f=top;

int s = size+1;

int t = top;

while((f<=size) && (s<=bottom))

{

if(l[f]<=l[s])

{

temp[t]=l[f];

f++;

}

else

{

temp[t]=l[s];

s++;

}

t++;

}

if(f<=size)

{

for(f=f;f<=size;f++)

{

temp[t]=l[f];

t++;

}

}

else

{

for(s=s;s<=bottom;s++)

{

temp[t] = l[s];

t++;

}

}

for(int up=top;up<=bottom;up++)

{

l[up]= temp[up];

}

}

void merge::mergepass(float appened[],int m,int n)

{

if(m!=n)

{

int mid = (m+n)/2;

mergepass(appened,m,mid);

mergepass(appened,mid+1,n);

mergesort(appened,m,mid,n);

}

}

void main()

{

merge s;

float list[100];

int n;

clrscr();

cout<<"\n enter the size of the list::\n ";

cin>>n;

for(int i=0;i<n;i++)

{

cout<<"\n element ::\n"<<i+1<<":";

cin>>list[i];

}

cout<<"\n entered the list::\n";

for(i=0;i<n;i++)

cout<<"\n"<<list[i]<<"\n";

i=0;

s.mergepass(list,i,n-1);

cout<<"\n merge sort:\n";

for(i=0;i<n;i++)

cout<<"\n"<<list[i]<<"\n";

getch();

}

**Output of the program:**

enter the size of the list::5

element ::1:7

element ::2:9

element ::3:8

element ::4:4

element ::5:6

entered the list::

7 9 8 4 6

merge sort:

4 6 7 8 9

**Week 5**

**Write a Program to Implement the Binary Search Tree Operations.**

**Aim:** to write a c++program to implement a binary search tree.

**Abstract data type::**

1. binary search tree is a binary tree that may be empty.
2. a non empty binary search tree satisfies the properties.
3. every element has a key(or value), and no two elements have the same key;
4. the keys in the left subtree of the root are Smaller than the key in the root.
5. the keys in the right subtree of the root are larger than the key in the root.
6. the left and the right subtrees of the root are also binary search trees.

**Member functions::**

* + 1. **insert():** the purpose of this function is to insert the elements according to the properties.
    2. **delete():**the purpose of this function is to delete the elements from the tree.
    3. **inorder():**the purpose of this function is to display the elements in inorder technique.
    4. **search():**the purpose of this function is to search for the given element.

**Algorithm::**

1**.inorder(node \*r) :**

1.start

2.if ptr!=NULL then

3.inorder(ptr->lc)

4.print data(ptr)

5.inorder (ptr->rc)

6.stop

#include<iostream.h>

#include<conio.h>

#include<process.h>

class node

{

node \*lptr,\*rptr;

int data;

friend class tree;

};

class tree

{

node \*root;

public:

tree()

{

root=NULL;

}

void insert();

void del();

void inorder(node \*);

void search();

node \*getroot()

{

return root;

}

};

void tree::inorder(node \*r)

{

if(r==NULL)

{

cout<<"\n empty tree";

return;

}

if(r->lptr!=NULL)

inorder(r->lptr);

cout<<r->data;

if(r->rptr!=NULL)

inorder(r->rptr);

}

void tree::search()

{

int num,found=0;

node \*s=root,\*c;

cout<<" Enter searching number::";

cin>>num;

while(s!=NULL)

{

c=s;

if(found==1)

cout<<c->data;

if(num>s->data)

s=s->rptr;

else if(num<s->data)

s=s->lptr;

else

{

found=1;

break;

}

cout<<"\n parent node is::"<<c->data;

}

if(found==1)

cout<<"\n number is found\n";

else

cout<<"\n number is not found\n";

}

void tree::insert()

{

node \*s,\*c,\*n;

int num;

cout<<"\n enter number for insertion::";

cin>>num;

n=new node;

n->data=num;

n->lptr=n->rptr=NULL;

if(root==NULL)

{

root=n;

return;

}

s=root;

while(s!=NULL)

{

c=s;

if(num>s->data)

s=s->rptr;

else

s=s->lptr;

}

if(num>c->data)

c->rptr=n;

else

c->lptr=n;

}

void tree::del()

{

node \*d=root,\*pd=NULL,\*le,\*ple,\*c;

int dnum;

cout<<"\n enter deletion number::";

cin>>dnum;

while(d!=NULL && d->data!=dnum)

{

pd=d;

if(dnum>d->data)

d=d->rptr;

else

d=d->lptr;

}

if(d==NULL)

{

cout<<"\n number not found";

return;

}

if(d->lptr!=NULL &&d->rptr!=NULL)

{

le=d->lptr;

ple=d;

while(le->rptr!=NULL)

{

ple=le;

le=le->rptr;

}

d->data=le->data;

d=le;

pd=ple;

}

if(d->lptr!=NULL)

c=d->lptr;

else

c=d->rptr;

if(d==root)

root=c;

if(pd->lptr==d)

pd->lptr=c;

else

pd->rptr=c;

}

void main()

{

int op;

clrscr();

tree t;

node \*r;

while(1)

{

clrscr();

r=t.getroot();

cout<<"\n INORDER::\n";

t.inorder(r);

cout<<"\n 1.insert\n2.deletion\n3.search\n enter ur option::";

cin>>op;

switch(op)

{

case 1:t.insert();

break;

case 2:t.del();

break;

case 3: t.search();

break;

default:exit(0);

}

getch();

}

}

**output of the program::**

INORDER::

empty tree

1.insert

2.deletion

3.search

enter ur option::1

enter number to insert: 4

INORDER::

4

1.insert

2.deletion

3.search

enter ur option::1

enter number to insert::5

INORDER::

4

5

1.insert

2.deletion

3.search

enter ur option::1

enter number to insert::6

INORDER::

4

5

6

1.insert

2.deletion

3.search

enter ur option::3

Enter searching number::6

parent node is::5

number is found

INORDER::

4

5

6

1.insert

2.deletion

3.search

enter ur option::2

enter number to delete::5

INORDER::

4

6

1.insert

2.deletion

3.search

enter ur option::6

exit.

**Week 6**

**Aim: To Write a Program to Perform the Tree Traversal Techniques by using the Iterative Method**

**Program:**

#include<iostream.h>

#include<conio.h>

#include<stdlib.h>

class node

{

public:

class node \*left;

class node \*right;

int data;

};

class tree: public node

{

public:

int stk[50],top;

node \*root;

tree()

{

root=NULL;

top=0;

}

void insert(int ch)

{

node \*temp,\*temp1;

if(root== NULL)

{

root=new node;

root->data=ch;

root->left=NULL;

root->right=NULL;

return;

}

temp1=new node;

temp1->data=ch;

temp1->right=temp1->left=NULL;

temp=search(root,ch);

if(temp->data>ch)

temp->left=temp1;

else

temp->right=temp1;

}

node \*search(node \*temp,int ch)

{

if(root== NULL)

{

cout <<"no node present";

return NULL;

}

if(temp->left==NULL && temp->right== NULL)

return temp;

if(temp->data>ch)

{ if(temp->left==NULL) return temp;

search(temp->left,ch);}

else

{ if(temp->right==NULL) return temp;

search(temp->right,ch);

} }

void display(node \*temp)

{

if(temp==NULL)

return ;

display(temp->left);

cout<<temp->data << " ";

display(temp->right);

}

void postorder( node \*root)

{

node \*p;

p=root;

top=0;

while(1)

{

while(p!=NULL)

{

stk[top]=p->data;

top++;

if(p->right!=NULL)

stk[top++]=-p->right->data;

p=p->left;

}

while(stk[top-1] > 0 || top==0)

{

if(top==0) return;

cout << stk[top-1] <<" ";

p=pop(root);

}

if(stk[top-1]<0)

{

stk[top-1]=-stk[top-1];

p=pop(root);

} }

}

node \* pop(node \*p)

{

int ch;

ch=stk[top-1];

if(p->data==ch)

{

top--;

return p;

}

if(p->data>ch)

pop(p->left);

else

pop(p->right);

}

};

void main()

{

tree t1;

int ch,n,i;

clrscr();

while(1)

{

cout <<"\n1.INSERT\n2.DISPLAY 3.POSTORDER TRAVERSE\n4.EXIT\nEnter your choice:";

cin >> ch;

switch(ch)

{

case 1: cout <<"enter no of elements to insert:";

cout<<"\n enter the elements";

cin >> n;

for(i=1;i<=n;i++)

{ cin >> ch;

t1.insert(ch);

}

break;

case 2: t1.display(t1.root);break;

case 3: t1.postorder(t1.root); break;

case 4: exit(1);

}

}

}

**OUTPUT**

1.INSERT  
2.DISPLAY 3.POSTORDER TRAVERSE  
4.EXIT  
Enter your choice:1  
enter no of elements to insert:  
enter the elements7  
5 24 36 11 44 2 21

1.INSERT  
2.DISPLAY 3.POSTORDER TRAVERSE  
4.EXIT  
Enter your choice:2  
2 5 11 21 24 36 44

1.INSERT  
2.DISPLAY 3.POSTORDER TRAVERSE  
4.EXIT  
Enter your choice:3  
2 21 11 44 36 24 5

1.INSERT  
2.DISPLAY 3.POSTORDER TRAVERSE  
4.EXIT  
Enter your choice:4

**Week 7**

**Write C programs for implementing the following graph traversal algorithms:**

**a)Depth first traversal b)Breadth first traversal**

**Aim:**

A c++ program to implement the breadth first search algorithm for the given graph.

**Abstract Data Type:**

* 1. Bfs is implemented by using queues.
  2. In this technique, the search operation starts from any node in the graph structure.
  3. The search operation proceed by visiting level by level node in the graph.
  4. In bfs there is a cycle path in the graph ,a node can be adjacent to mare than one node. Because of this, node can be visited more than one time.to eliminate this problem a flag variable is used to know weather the node is visitedor not.

**Member Functions:**

1. **graph():** it is a constructor which is used to initialize data members.
2. **bfs():**it is a function used to implement bfs algorithm.
3. **addq():**it a function used to add the node to the queue.
4. **deleteq():**it is a function used to delete a node from the queue.

**Algorithm:**

**1. bfs(int):**

1. start.
2. visited[i]=1
3. addq(i)
4. while front != rear

4.1. i=deleteq()

4.2. i=i+1

5. for j=0 to maxsize(where max size=8)

6. if visited[j]==0 && graph[i][j]==1 then

6.1. addq(j)

6.2. visited[j]=1

7. j=j+1

8. stop

**2.addq(i):**

1. start
2. if rear==maxsize1-1 then (where maxsize1=20)
3. print “stack full”
4. print item
5. else
6. queue[++rear=item]
7. stop

**3.deleteq():**

1. start
2. if rear==front then
3. print” queue is empty”
4. return queue[++front]
5. stop

**c++ program to implement the BFS(Beadth First Search) for the given graph.**

#include<iostream.h>

#include<conio.h>

#include<stdlib.h>

#include<string.h>

# define maxsize 8

# define maxsize1 20

void addq(int);

int deleteq(void);

void bfs(int);

int visited[maxsize];

int graph[][maxsize]={{0,1,1,1,0,0,0,0},

{1,0,1,0,1,1,0,0},

{1,1,0,0,0,1,0,0},

{1,0,0,0,0,0,0,1},

{0,1,0,0,0,1,0,0},

{0,1,1,0,1,0,0,0},

{0,0,0,0,0,0,0,1},

{0,0,0,1,0,0,1,0}};

int queue[maxsize],front,rear;

void main()

{

clrscr();

int i;

front=rear=-1;

for(i=0;i<maxsize;i++)

visited[i]=0;

bfs(0);

getch();

}

void bfs(int i)

{

int j;

visited[i]=1;

addq(i);

while( front!=rear)

{

i=deleteq();

cout<<”\n”<<”node visited=”<<i+1<<endl;

for(j=0;j<maxsize;j++)

{

if((visited [j]==0)&&(graph[i][j]==1))

{

addq(j);

visited[j]=1;

}

}

}

void addq(int item)

{

if(rear==maxsize1-1)

cout<<”stack full not adding\n”<<item;

else

queue[++rear]=item;

}

int deleteq()

{

if(rear==front)

{

cout<<”queue is empty\n”;

return -1;

}

return queue[++front];

}

**Output of the program:**

node visited=1

node visited=2

node visited=3

node visited=4

node visited=5

node visited=6

node visited=8

node visited=7

**Write a c++ program to implement the DFS(depth First Search) for the given graph.**

**Aim:**

A c++ program to implement the depth first search algorithm for the given graph.

**Abstract Data Type:**

1. dfs algorithm is implemented by using stacks.

1. In this technique, the search operation starts from any node in the graph structureand continues in a single path until deepest element is found..

3. the search operation process continued until all nodes are visited in graph structure.

4. A flag variable is used to know whether the node is visited or not.

**Member Functions:**

**1.dfs(i):** is used to check whether the node is visited or not.

**Algorithm:**

1.Dfs(int i)

1.start

2.visited[i]=1

3. print “Node visited =”

4. print i=i+1

5. for j=0 to maxsize

5.1.if((visited[j]==0) &&( graph[i][j]==1))

5.2. dfs(j);

6.stop

**Aim: To write a c++ program implement DFS for given graph**

**Program:** #include<iostream.h>

#include<conio.h>

#include<stdlib.h>

#include<string.h>

void dfs(int);

int visited[8];

int graph[][8]={{0,1,1,1,0,0,0,0},

{1,0,1,0,1,1,0,0},

{1,1,0,0,0,1,0,0},

{1,0,0,0,0,0,0,1},

{0,1,0,0,0,1,0,0},

{0,1,1,0,1,0,0,0},

{0,0,0,0,0,0,0,1},

{0,0,0,1,0,0,1,0}};

void main()

{

int i;

clrscr();

for(i=0;i<8;i++)

visited[i]=0;

for(i=0;i<8;i++)

if(visited[i]==0)

dfs(i);

getch();

}

void dfs(int i)

{

int j;

visited[i]=1;

cout<<"\n"<<"node visited="<<i+1<<endl;

for(j=0;j<8;j++)

if((visited [j]==0)&&(graph[i][j]==1))

dfs(j);

}

**Output of the program:**

node visited=1

node visited=2

node visited=3

node visited=6

node visited=5

node visited=4

node visited=8

node visited=7

**Week 8**

AIM: Write a Program to Implement All functions of a Dictionary by using Hashing

**Program:**

#include<iostream>

#include<conio.h>

#include<stdlib.h>

using namespace std;

# define max 10

typedef struct list

{

int data;

struct list \*next;

} node\_type;

node\_type \*ptr[max],\*root[max],\*temp[max];

class Dictionary

{

public:

int index;

Dictionary();

void insert(int);

void search(int);

void delete\_ele(int);

};

Dictionary::Dictionary()

{

index=-1;

for(int i=0; i<max; i++)

{

root[i]=NULL;

ptr[i]=NULL;

temp[i]=NULL;

}

}

void Dictionary::insert(int key)

{

index=int(key%max);

ptr[index]=(node\_type\*)malloc(sizeof(node\_type));

ptr[index]->data=key;

if(root[index]==NULL)

{

root[index]=ptr[index];

root[index]->next=NULL;

temp[index]=ptr[index];

}

else

{

temp[index]=root[index];

while(temp[index]->next!=NULL)

temp[index]=temp[index]->next;

temp[index]->next=ptr[index];

}

}

void Dictionary::search(int key)

{

int flag=0;

index=int(key%max);

temp[index]=root[index];

while(temp[index]!=NULL)

{

if(temp[index]->data==key)

{

cout<<"\nSearch key is found!!";

flag=1;

break;

}

else temp[index]=temp[index]->next;

}

if (flag==0)

cout<<"\nsearch key not found.......";

}

void Dictionary::delete\_ele(int key)

{

index=int(key%max);

temp[index]=root[index];

while(temp[index]->data!=key && temp[index]!=NULL)

{

ptr[index]=temp[index];

temp[index]=temp[index]->next;

}

ptr[index]->next=temp[index]->next;

cout<<"\n"<<temp[index]->data<<" has been deleted.";

temp[index]->data=-1;

temp[index]=NULL;

free(temp[index]);

}

main()

{

int val,ch,n,num;

char c;

Dictionary d;

do

{

cout<<"\nMENU:\n1.Create";

cout<<"\n2.Search for a value\n3.Delete an value";

cout<<"\nEnter your choice:";

cin>>ch;

switch(ch)

{

case 1:

cout<<"\nEnter the number of elements to be inserted:";

cin>>n;

cout<<"\nEnter the elements to be inserted:";

for(int i=0; i<n; i++)

{

cin>>num;

d.insert(num);

}

break;

case 2:

cout<<"\nEnter the element to be searched:";

cin>>n;

d.search(n);

case 3:

cout<<"\nEnter the element to be deleted:";

cin>>n;

d.delete\_ele(n);

break;

default:

cout<<"\nInvalid Choice.";

}

cout<<"\nEnter y to Continue:";

cin>>c;

}

while(c=='y');

getch();

}

OUTPUT:

MENU:

1.Create

2.Search for a value

3.Delete an value

Enter your choice:1

Enter the number of elements to be inserted:3

Enter the elements to be inserted:12

23

56

Enter y to Continue:y

MENU:

1.Create

2.Search for a value

3.Delete an value

Enter your choice:2

Enter the element to be searched:23

Search key is found!!

Enter the element to be deleted:23

23 has been deleted.

**Week 9**

**Write a Program to Implement Skip List Operations.**

Aim: This C++ Program demonstrates the implementation of Splay Tree.

Here is source code of the C++ Program to demonstrate the implementation of Splay Tree. The C++ program is successfully compiled and run on a Linux system. The program output is also shown below.

 Program:

1. /\*

2. \* C++ Program to Implement Splay Tree

3. \*/

4.

5. #include <iostream>

6. #include <cstdio>

7. #include <cstdlib>

8. using namespace std;

9.

10. struct splay

11. {

12. int key;

13. splay\* lchild;

14. splay\* rchild;

15. };

16.

17. class SplayTree

18. {

19. public:

20. SplayTree()

21. {

22. }

23.

24. // RR(Y rotates to the right)

25. splay\* RR\_Rotate(splay\* k2)

26. {

27. splay\* k1 = k2->lchild;

28. k2->lchild = k1->rchild;

29. k1->rchild = k2;

30. return k1;

31. }

32.

33. // LL(Y rotates to the left)

34. splay\* LL\_Rotate(splay\* k2)

35. {

36. splay\* k1 = k2->rchild;

37. k2->rchild = k1->lchild;

38. k1->lchild = k2;

39. return k1;

40. }

41.

42. // An implementation of top-down splay tree

43. splay\* Splay(int key, splay\* root)

44. {

45. if (!root)

46. return NULL;

47. splay header;

48. /\* header.rchild points to L tree;

49. header.lchild points to R Tree \*/

50. header.lchild = header.rchild = NULL;

51. splay\* LeftTreeMax = &header;

52. splay\* RightTreeMin = &header;

53. while (1)

54. {

55. if (key < root->key)

56. {

57. if (!root->lchild)

58. break;

59. if (key < root->lchild->key)

60. {

61. root = RR\_Rotate(root);

62. // only zig-zig mode need to rotate once,

63. if (!root->lchild)

64. break;

65. }

66. /\* Link to R Tree \*/

67. RightTreeMin->lchild = root;

68. RightTreeMin = RightTreeMin->lchild;

69. root = root->lchild;

70. RightTreeMin->lchild = NULL;

71. }

72. else if (key > root->key)

73. {

74. if (!root->rchild)

75. break;

76. if (key > root->rchild->key)

77. {

78. root = LL\_Rotate(root);

79. // only zag-zag mode need to rotate once,

80. if (!root->rchild)

81. break;

82. }

83. /\* Link to L Tree \*/

84. LeftTreeMax->rchild = root;

85. LeftTreeMax = LeftTreeMax->rchild;

86. root = root->rchild;

87. LeftTreeMax->rchild = NULL;

88. }

89. else

90. break;

91. }

92. /\* assemble L Tree, Middle Tree and R tree \*/

93. LeftTreeMax->rchild = root->lchild;

94. RightTreeMin->lchild = root->rchild;

95. root->lchild = header.rchild;

96. root->rchild = header.lchild;

97. return root;

98. }

99.

100. splay\* New\_Node(int key)

101. {

102. splay\* p\_node = new splay;

103. if (!p\_node)

104. {

105. fprintf(stderr, "Out of memory!\n");

106. exit(1);

107. }

108. p\_node->key = key;

109. p\_node->lchild = p\_node->rchild = NULL;

110. return p\_node;

111. }

112.

113. splay\* Insert(int key, splay\* root)

114. {

115. static splay\* p\_node = NULL;

116. if (!p\_node)

117. p\_node = New\_Node(key);

118. else

119. p\_node->key = key;

120. if (!root)

121. {

122. root = p\_node;

123. p\_node = NULL;

124. return root;

125. }

126. root = Splay(key, root);

127. /\* This is BST that, all keys <= root->key is in root->lchild, all keys >

128. root->key is in root->rchild. \*/

129. if (key < root->key)

130. {

131. p\_node->lchild = root->lchild;

132. p\_node->rchild = root;

133. root->lchild = NULL;

134. root = p\_node;

135. }

136. else if (key > root->key)

137. {

138. p\_node->rchild = root->rchild;

139. p\_node->lchild = root;

140. root->rchild = NULL;

141. root = p\_node;

142. }

143. else

144. return root;

145. p\_node = NULL;

146. return root;

147. }

148.

149. splay\* Delete(int key, splay\* root)

150. {

151. splay\* temp;

152. if (!root)

153. return NULL;

154. root = Splay(key, root);

155. if (key != root->key)

156. return root;

157. else

158. {

159. if (!root->lchild)

160. {

161. temp = root;

162. root = root->rchild;

163. }

164. else

165. {

166. temp = root;

167. /\*Note: Since key == root->key,

168. so after Splay(key, root->lchild),

169. the tree we get will have no right child tree.\*/

170. root = Splay(key, root->lchild);

171. root->rchild = temp->rchild;

172. }

173. free(temp);

174. return root;

175. }

176. }

177.

178. splay\* Search(int key, splay\* root)

179. {

180. return Splay(key, root);

181. }

182.

183. void InOrder(splay\* root)

184. {

185. if (root)

186. {

187. InOrder(root->lchild);

188. cout<< "key: " <<root->key;

189. if(root->lchild)

190. cout<< " | left child: "<< root->lchild->key;

191. if(root->rchild)

192. cout << " | right child: " << root->rchild->key;

193. cout<< "\n";

194. InOrder(root->rchild);

195. }

196. }

197. };

198.

199. int main()

200. {

201. SplayTree st;

202. int vector[10] = {9,8,7,6,5,4,3,2,1,0};

203. splay \*root;

204. root = NULL;

205. const int length = 10;

206. int i;

207. for(i = 0; i < length; i++)

208. root = st.Insert(vector[i], root);

209. cout<<"\nInOrder: \n";

210. st.InOrder(root);

211. int input, choice;

212. while(1)

213. {

214. cout<<"\nSplay Tree Operations\n";

215. cout<<"1. Insert "<<endl;

216. cout<<"2. Delete"<<endl;

217. cout<<"3. Search"<<endl;

218. cout<<"4. Exit"<<endl;

219. cout<<"Enter your choice: ";

220. cin>>choice;

221. switch(choice)

222. {

223. case 1:

224. cout<<"Enter value to be inserted: ";

225. cin>>input;

226. root = st.Insert(input, root);

227. cout<<"\nAfter Insert: "<<input<<endl;

228. st.InOrder(root);

229. break;

230. case 2:

231. cout<<"Enter value to be deleted: ";

232. cin>>input;

233. root = st.Delete(input, root);

234. cout<<"\nAfter Delete: "<<input<<endl;

235. st.InOrder(root);

236. break;

237. case 3:

238. cout<<"Enter value to be searched: ";

239. cin>>input;

240. root = st.Search(input, root);

241. cout<<"\nAfter Search "<<input<<endl;

242. st.InOrder(root);

243. break;

244.

245. case 4:

246. exit(1);

247. default:

248. cout<<"\nInvalid type! \n";

249. }

250. }

251. cout<<"\n";

252. return 0;

253. }

**OUTPUT:**

Splay Tree Operations

1. Insert

2. Delete

3. Search

4. Exit

Enter your choice: 1

Enter value to be inserted: 1

After Insert: 1

key: 1

Splay Tree Operations

1. Insert

2. Delete

3. Search

4. Exit

Enter your choice: 1

Enter value to be inserted: 9

After Insert: 9

key: 1

key: 9 | left child: 1

Splay Tree Operations

1. Insert

2. Delete

3. Search

4. Exit

Enter your choice: 1

Enter value to be inserted: 5

After Insert: 5

key: 1

key: 5 | left child: 1 | right child: 9

key: 9

Splay Tree Operations

1. Insert

2. Delete

3. Search

4. Exit

Enter your choice: 1

Enter value to be inserted: 3

After Insert: 3

key: 1

key: 3 | left child: 1 | right child: 5

key: 5 | right child: 9

key: 9

Splay Tree Operations

1. Insert

2. Delete

3. Search

4. Exit

Enter your choice: 1

Enter value to be inserted: 8

After Insert: 8

key: 1

key: 3 | left child: 1

key: 5 | left child: 3

key: 8 | left child: 5 | right child: 9

key: 9

Splay Tree Operations

1. Insert

2. Delete

3. Search

4. Exit

Enter your choice: 1

Enter value to be inserted: 7

After Insert: 7

key: 1

key: 3 | left child: 1

key: 5 | left child: 3

key: 7 | left child: 5 | right child: 8

key: 8 | right child: 9

key: 9

Splay Tree Operations

1. Insert

2. Delete

3. Search

4. Exit

Enter your choice: 1

Enter value to be inserted: 2

After Insert: 2

key: 1

key: 2 | left child: 1 | right child: 5

key: 3

key: 5 | left child: 3 | right child: 7

key: 7 | right child: 8

key: 8 | right child: 9

key: 9

Splay Tree Operations

1. Insert

2. Delete

3. Search

4. Exit

Enter your choice: 1

Enter value to be inserted: 4

After Insert: 4

key: 1

key: 2 | left child: 1

key: 3 | left child: 2

key: 4 | left child: 3 | right child: 5

key: 5 | right child: 7

key: 7 | right child: 8

key: 8 | right child: 9

key: 9

Splay Tree Operations

1. Insert

2. Delete

3. Search

4. Exit

Enter your choice: 1

Enter value to be inserted: 10

After Insert: 10

key: 1

key: 2 | left child: 1

key: 3 | left child: 2

key: 4 | left child: 3

key: 5 | left child: 4 | right child: 8

key: 7

key: 8 | left child: 7

key: 9 | left child: 5

key: 10 | left child: 9

Splay Tree Operations

1. Insert

2. Delete

3. Search

4. Exit

Enter your choice: 1

Enter value to be inserted: 6

After Insert: 6

key: 1

key: 2 | left child: 1

key: 3 | left child: 2

key: 4 | left child: 3

key: 5 | left child: 4

key: 6 | left child: 5 | right child: 7

key: 7 | right child: 9

key: 8

key: 9 | left child: 8 | right child: 10

key: 10

Splay Tree Operations

1. Insert

2. Delete

3. Search

4. Exit

Enter your choice: 2

Enter value to be deleted: 5

After Delete: 5

key: 1

key: 2 | left child: 1

key: 3 | left child: 2

key: 4 | left child: 3 | right child: 6

key: 6 | right child: 7

key: 7 | right child: 9

key: 8

key: 9 | left child: 8 | right child: 10

key: 10

Splay Tree Operations

1. Insert

2. Delete

3. Search

4. Exit

Enter your choice: 2

Enter value to be deleted: 10

After Delete: 10

key: 1

key: 2 | left child: 1

key: 3 | left child: 2

key: 4 | left child: 3

key: 6 | left child: 4 | right child: 7

key: 7 | right child: 8

key: 8

key: 9 | left child: 6

Splay Tree Operations

1. Insert

2. Delete

3. Search

4. Exit

Enter your choice: 3

Enter value to be searched: 9

After Search 9

key: 1

key: 2 | left child: 1

key: 3 | left child: 2

key: 4 | left child: 3

key: 6 | left child: 4 | right child: 7

key: 7 | right child: 8

key: 8

key: 9 | left child: 6

Splay Tree Operations

1. Insert

2. Delete

3. Search

4. Exit

Enter your choice: 4

WEEK 10:

**Aim:** Write a program to Implement Insertion and Deletion Operations on AVL Trees

**Program:**

This C++ Program demonstrates operations on AVL Trees.

Here is source code of the C++ Program to demonstrate AVL Trees. The C++ program is successfully compiled and run on a Linux system. The program output is also shown below.

1. /\*

2. \* C++ program to Implement AVL Tree

3. \*/

4. #include<iostream>

5. #include<cstdio>

6. #include<sstream>

7. #include<algorithm>

8. #define pow2(n) (1 << (n))

9. using namespace std;

10.

11. /\*

12. \* Node Declaration

13. \*/

14. struct avl\_node

15. {

16. int data;

17. struct avl\_node \*left;

18. struct avl\_node \*right;

19. }\*root;

20.

21. /\*

22. \* Class Declaration

23. \*/

24. class avlTree

25. {

26. public:

27. int height(avl\_node \*);

28. int diff(avl\_node \*);

29. avl\_node \*rr\_rotation(avl\_node \*);

30. avl\_node \*ll\_rotation(avl\_node \*);

31. avl\_node \*lr\_rotation(avl\_node \*);

32. avl\_node \*rl\_rotation(avl\_node \*);

33. avl\_node\* balance(avl\_node \*);

34. avl\_node\* insert(avl\_node \*, int );

35. void display(avl\_node \*, int);

36. void inorder(avl\_node \*);

37. void preorder(avl\_node \*);

38. void postorder(avl\_node \*);

39. avlTree()

40. {

41. root = NULL;

42. }

43. };

44.

45. /\*

46. \* Main Contains Menu

47. \*/

48. int main()

49. {

50. int choice, item;

51. avlTree avl;

52. while (1)

53. {

54. cout<<"\n---------------------"<<endl;

55. cout<<"AVL Tree Implementation"<<endl;

56. cout<<"\n---------------------"<<endl;

57. cout<<"1.Insert Element into the tree"<<endl;

58. cout<<"2.Display Balanced AVL Tree"<<endl;

59. cout<<"3.InOrder traversal"<<endl;

60. cout<<"4.PreOrder traversal"<<endl;

61. cout<<"5.PostOrder traversal"<<endl;

62. cout<<"6.Exit"<<endl;

63. cout<<"Enter your Choice: ";

64. cin>>choice;

65. switch(choice)

66. {

67. case 1:

68. cout<<"Enter value to be inserted: ";

69. cin>>item;

70. root = avl.insert(root, item);

71. break;

72. case 2:

73. if (root == NULL)

74. {

75. cout<<"Tree is Empty"<<endl;

76. continue;

77. }

78. cout<<"Balanced AVL Tree:"<<endl;

79. avl.display(root, 1);

80. break;

81. case 3:

82. cout<<"Inorder Traversal:"<<endl;

83. avl.inorder(root);

84. cout<<endl;

85. break;

86. case 4:

87. cout<<"Preorder Traversal:"<<endl;

88. avl.preorder(root);

89. cout<<endl;

90. break;

91. case 5:

92. cout<<"Postorder Traversal:"<<endl;

93. avl.postorder(root);

94. cout<<endl;

95. break;

96. case 6:

97. exit(1);

98. break;

99. default:

100. cout<<"Wrong Choice"<<endl;

101. }

102. }

103. return 0;

104. }

105.

106. /\*

107. \* Height of AVL Tree

108. \*/

109. int avlTree::height(avl\_node \*temp)

110. {

111. int h = 0;

112. if (temp != NULL)

113. {

114. int l\_height = height (temp->left);

115. int r\_height = height (temp->right);

116. int max\_height = max (l\_height, r\_height);

117. h = max\_height + 1;

118. }

119. return h;

120. }

121.

122. /\*

123. \* Height Difference

124. \*/

125. int avlTree::diff(avl\_node \*temp)

126. {

127. int l\_height = height (temp->left);

128. int r\_height = height (temp->right);

129. int b\_factor= l\_height - r\_height;

130. return b\_factor;

131. }

132.

133. /\*

134. \* Right- Right Rotation

135. \*/

136. avl\_node \*avlTree::rr\_rotation(avl\_node \*parent)

137. {

138. avl\_node \*temp;

139. temp = parent->right;

140. parent->right = temp->left;

141. temp->left = parent;

142. return temp;

143. }

144. /\*

145. \* Left- Left Rotation

146. \*/

147. avl\_node \*avlTree::ll\_rotation(avl\_node \*parent)

148. {

149. avl\_node \*temp;

150. temp = parent->left;

151. parent->left = temp->right;

152. temp->right = parent;

153. return temp;

154. }

155.

156. /\*

157. \* Left - Right Rotation

158. \*/

159. avl\_node \*avlTree::lr\_rotation(avl\_node \*parent)

160. {

161. avl\_node \*temp;

162. temp = parent->left;

163. parent->left = rr\_rotation (temp);

164. return ll\_rotation (parent);

165. }

166.

167. /\*

168. \* Right- Left Rotation

169. \*/

170. avl\_node \*avlTree::rl\_rotation(avl\_node \*parent)

171. {

172. avl\_node \*temp;

173. temp = parent->right;

174. parent->right = ll\_rotation (temp);

175. return rr\_rotation (parent);

176. }

177.

178. /\*

179. \* Balancing AVL Tree

180. \*/

181. avl\_node \*avlTree::balance(avl\_node \*temp)

182. {

183. int bal\_factor = diff (temp);

184. if (bal\_factor > 1)

185. {

186. if (diff (temp->left) > 0)

187. temp = ll\_rotation (temp);

188. else

189. temp = lr\_rotation (temp);

190. }

191. else if (bal\_factor < -1)

192. {

193. if (diff (temp->right) > 0)

194. temp = rl\_rotation (temp);

195. else

196. temp = rr\_rotation (temp);

197. }

198. return temp;

199. }

200.

201. /\*

202. \* Insert Element into the tree

203. \*/

204. avl\_node \*avlTree::insert(avl\_node \*root, int value)

205. {

206. if (root == NULL)

207. {

208. root = new avl\_node;

209. root->data = value;

210. root->left = NULL;

211. root->right = NULL;

212. return root;

213. }

214. else if (value < root->data)

215. {

216. root->left = insert(root->left, value);

217. root = balance (root);

218. }

219. else if (value >= root->data)

220. {

221. root->right = insert(root->right, value);

222. root = balance (root);

223. }

224. return root;

225. }

226.

227. /\*

228. \* Display AVL Tree

229. \*/

230. void avlTree::display(avl\_node \*ptr, int level)

231. {

232. int i;

233. if (ptr!=NULL)

234. {

235. display(ptr->right, level + 1);

236. printf("\n");

237. if (ptr == root)

238. cout<<"Root -> ";

239. for (i = 0; i < level && ptr != root; i++)

240. cout<<" ";

241. cout<<ptr->data;

242. display(ptr->left, level + 1);

243. }

244. }

245.

246. /\*

247. \* Inorder Traversal of AVL Tree

248. \*/

249. void avlTree::inorder(avl\_node \*tree)

250. {

251. if (tree == NULL)

252. return;

253. inorder (tree->left);

254. cout<<tree->data<<" ";

255. inorder (tree->right);

256. }

257. /\*

258. \* Preorder Traversal of AVL Tree

259. \*/

260. void avlTree::preorder(avl\_node \*tree)

261. {

262. if (tree == NULL)

263. return;

264. cout<<tree->data<<" ";

265. preorder (tree->left);

266. preorder (tree->right);

267.

268. }

269.

270. /\*

271. \* Postorder Traversal of AVL Tree

272. \*/

273. void avlTree::postorder(avl\_node \*tree)

274. {

275. if (tree == NULL)

276. return;

277. postorder ( tree ->left );

278. postorder ( tree ->right );

279. cout<<tree->data<<" ";

280. }

OUTPUT:

---------------------

AVL Tree Implementation

---------------------

1.Insert Element into the tree

2.Display Balanced AVL Tree

3.InOrder traversal

4.PreOrder traversal

5.PostOrder traversal

6.Exit

Enter your Choice: 2

Tree is Empty

--------------------

AVL Tree Implementation

---------------------

1.Insert Element into the tree

2.Display Balanced AVL Tree

3.InOrder traversal

4.PreOrder traversal

5.PostOrder traversal

6.Exit

Enter your Choice: 1

Enter value to be inserted: 8

---------------------

AVL Tree Implementation

---------------------

1.Insert Element into the tree

2.Display Balanced AVL Tree

3.InOrder traversal

4.PreOrder traversal

5.PostOrder traversal

6.Exit

Enter your Choice: 2

Balanced AVL Tree:

Root -> 8

---------------------

AVL Tree Implementation

---------------------

1.Insert Element into the tree

2.Display Balanced AVL Tree

3.InOrder traversal

4.PreOrder traversal

5.PostOrder traversal

6.Exit

Enter your Choice: 1

Enter value to be inserted: 5

---------------------

AVL Tree Implementation

---------------------

1.Insert Element into the tree

2.Display Balanced AVL Tree

3.InOrder traversal

4.PreOrder traversal

5.PostOrder traversal

6.Exit

Enter your Choice: 2

Balanced AVL Tree:

Root -> 8

5

---------------------

AVL Tree Implementation

---------------------

1.Insert Element into the tree

2.Display Balanced AVL Tree

3.InOrder traversal

4.PreOrder traversal

5.PostOrder traversal

6.Exit

Enter your Choice: 1

Enter value to be inserted: 4

---------------------

AVL Tree Implementation

---------------------

1.Insert Element into the tree

2.Display Balanced AVL Tree

3.InOrder traversal

4.PreOrder traversal

5.PostOrder traversal

6.Exit

Enter your Choice: 2

Balanced AVL Tree:

8

Root -> 5

4

---------------------

AVL Tree Implementation

---------------------

1.Insert Element into the tree

2.Display Balanced AVL Tree

3.InOrder traversal

4.PreOrder traversal

5.PostOrder traversal

6.Exit

Enter your Choice: 1

Enter value to be inserted: 11

---------------------

AVL Tree Implementation

---------------------

1.Insert Element into the tree

2.Display Balanced AVL Tree

3.InOrder traversal

4.PreOrder traversal

5.PostOrder traversal

6.Exit

Enter your Choice: 2

Balanced AVL Tree:

11

8

Root -> 5

4

---------------------

AVL Tree Implementation

---------------------

1.Insert Element into the tree

2.Display Balanced AVL Tree

3.InOrder traversal

4.PreOrder traversal

5.PostOrder traversal

6.Exit

Enter your Choice: 1

Enter value to be inserted: 15

---------------------

AVL Tree Implementation

---------------------

1.Insert Element into the tree

2.Display Balanced AVL Tree

3.InOrder traversal

4.PreOrder traversal

5.PostOrder traversal

6.Exit

Enter your Choice: 2

Balanced AVL Tree:

15

11

8

Root -> 5

4

---------------------

AVL Tree Implementation

---------------------

1.Insert Element into the tree

2.Display Balanced AVL Tree

3.InOrder traversal

4.PreOrder traversal

5.PostOrder traversal

6.Exit

Enter your Choice: 1

Enter value to be inserted: 3

---------------------

AVL Tree Implementation

---------------------

1.Insert Element into the tree

2.Display Balanced AVL Tree

3.InOrder traversal

4.PreOrder traversal

5.PostOrder traversal

6.Exit

Enter your Choice: 2

Balanced AVL Tree:

15

11

8

Root -> 5

4

3

---------------------

AVL Tree Implementation

---------------------

1.Insert Element into the tree

2.Display Balanced AVL Tree

3.InOrder traversal

4.PreOrder traversal

5.PostOrder traversal

6.Exit

Enter your Choice: 1

Enter value to be inserted: 6

---------------------

AVL Tree Implementation

---------------------

1.Insert Element into the tree

2.Display Balanced AVL Tree

3.InOrder traversal

4.PreOrder traversal

5.PostOrder traversal

6.Exit

Enter your Choice: 2

Balanced AVL Tree:

15

11

8

6

Root -> 5

4

3

---------------------

AVL Tree Implementation

---------------------

1.Insert Element into the tree

2.Display Balanced AVL Tree

3.InOrder traversal

4.PreOrder traversal

5.PostOrder traversal

6.Exit

Enter your Choice: 1

Enter value to be inserted: 2

---------------------

AVL Tree Implementation

---------------------

1.Insert Element into the tree

2.Display Balanced AVL Tree

3.InOrder traversal

4.PreOrder traversal

5.PostOrder traversal

6.Exit

Enter your Choice: 2

Balanced AVL Tree:

15

11

8

6

Root -> 5

4

3

2

---------------------

AVL Tree Implementation

---------------------

1.Insert Element into the tree

2.Display Balanced AVL Tree

3.InOrder traversal

4.PreOrder traversal

5.PostOrder traversal

6.Exit

Enter your Choice: 4

Preorder Traversal:

5 3 2 4 11 8 6 15

---------------------

AVL Tree Implementation

---------------------

1.Insert Element into the tree

2.Display Balanced AVL Tree

3.InOrder traversal

4.PreOrder traversal

5.PostOrder traversal

6.Exit

Enter your Choice: 5

Postorder Traversal:

2 4 3 6 8 15 11 5

---------------------

AVL Tree Implementation

---------------------

1.Insert Element into the tree

2.Display Balanced AVL Tree

3.InOrder traversal

4.PreOrder traversal

5.PostOrder traversal

6.Exit

Enter your Choice: 3

Inorder Traversal:

2 3 4 5 6 8 11 15

---------------------

AVL Tree Implementation

---------------------

1.Insert Element into the tree

2.Display Balanced AVL Tree

3.InOrder traversal

4.PreOrder traversal

5.PostOrder traversal

6.Exit

Enter your Choice: 2

Balanced AVL Tree:

15

11

8

6

Root -> 5

4

3

2

---------------------

AVL Tree Implementation

---------------------

1.Insert Element into the tree

2.Display Balanced AVL Tree

3.InOrder traversal

4.PreOrder traversal

5.PostOrder traversal

6.Exit

Enter your Choice: 6

**Week 12**

Write a Program to Implement Insertion and Deletion Operations on B – Trees

**Program ://** B-Tree Operations

#include<iostream.h>

#include<conio.h>

#include<stdio.h>

#include<process.h>

#define max 4

#define min 2

class btnode;

class qnode

{

public:

btnode\* data;

qnode\* next;

qnode(btnode\* t)

{

data=t;

next=NULL;

}

};

class queue

{

qnode\* front;

qnode\* rear;

public:

queue()

{

front=NULL;

rear=NULL;

}

void add(btnode\* t);

btnode\* remove();

int isempty()

{

if(front==NULL)

return 1;

else

return 0;

}

};

void queue:: add(btnode\* t)

{

if(front==NULL)

front=rear=new qnode(t);

else

rear=rear->next=new qnode(t);

}

btnode\* queue:: remove()

{

btnode\* t;

if(isempty())

return 0;

qnode\* x=front;

t=front->data;

front=x->next;

delete x;

return t;

}

class btnode

{

public:

int count;

int data[max+1];

btnode \*child[max+1];

};

class btree

{

int push\_down(int,btnode\*,int\*,btnode\*\*);

void pushin(int,btnode\*,btnode\*,int);

void split\_node(int,btnode\*,btnode\*,int,int\*,btnode\*\*);

void del\_node(int,btnode\*);

void remove\_key(btnode\*,int);

void successor(btnode\*,int);

void restore(btnode\*,int);

void move\_right(btnode\*,int);

void move\_left(btnode\*,int);

void combine\_nodes(btnode\*,int);

int search\_node(int,btnode\*,int\*);

btnode \*search(int,btnode\*,int\*);

public:

btnode\* root;

void display();

btnode\* del(int,btnode\*);

void pre\_rec(btnode\*);

btnode\* insert(int,btnode\*);

};

void btree:: display()

{

queue q;

btnode\* m;

m=root;

while(m)

{

for(int i=0;i<m->count;i++)

cout<<m->data[i]<<" ";

for(i=0;i<5;i++)

{

if(m->child[i])

q.add(m->child[i]);

}

m=q.remove();

cout<<"\n";

}

}

void btree:: pre\_rec(btnode \*n)

{

int i;

if(n!=NULL)

{

cout<<endl<<endl;

for(i=1;i<=n->count;i++)

{

cout<<"\t"<<n->data[i];

}

for(i=0;i<n->count;i=i+2)

{

pre\_rec(n->child[i]);

pre\_rec(n->child[i+1]);

}

if(n->count%2==0)

{

pre\_rec(n->child[n->count]);

}

}

}

btnode \*btree::del(int key,btnode \*root)

{

btnode \*oldroot;

del\_node(key,root);

if(root->count==0)

{

oldroot=root;

root=root->child[0];

delete oldroot;

}

return root;

}

void btree:: del\_node(int key,btnode \*curr)

{

int pos;

if(!curr)

{

cout<<"\n\n target not found";

return;

}

else

{

if(search\_node(key,curr,&pos))

if(curr->child[pos-1])

{

successor(curr,pos);

del\_node(curr->data[pos],curr->child[pos]);

}

else

{

remove\_key(curr,pos);

}

else

del\_node(key,curr->child[pos]);

if(curr->child[pos])

{

if(curr->child[pos]->count<min)

restore(curr,pos);

}

}

}

void btree:: remove\_key(btnode \*curr,int pos)

{

int p;

for(p=pos+1;p<=curr->count;p++)

{

curr->data[p-1]=curr->data[p];

curr->child[p-1]=curr->child[p];

}

curr->count--;

}

void btree::successor(btnode \*curr,int pos)

{

btnode \*leaf;

leaf=curr->child[pos];

while(leaf->child[0])

leaf=leaf->child[0];

curr->data[pos]=leaf->data[1];

}

void btree::restore(btnode \*curr,int pos)

{

if(pos==0)

if(curr->child[1]->count>min)

move\_left(curr,1);

else

combine\_nodes(curr,1);

else if(pos==curr->count)

if(curr->child[pos-1]->count>min)

{

move\_right(curr,pos);

}

else

{

combine\_nodes(curr,pos);

}

else if(curr->child[pos-1]->count>min)

move\_right(curr,pos);

else if(curr->child[pos+1]->count>min)

move\_left(curr,pos+1);

else

combine\_nodes(curr,pos);

}

void btree:: move\_right(btnode \*curr,int pos)

{

int p;

btnode \*temp;

temp=curr->child[pos];

for(p=temp->count;p>0;p--)

{

temp->data[p+1]=temp->data[p];

temp->child[p+1]=temp->child[p];

}

temp->child[1]=temp->child[0];

temp->count++;

temp->data[1]=curr->data[pos];

temp=curr->child[pos-1];

curr->data[pos]=temp->data[temp->count];

curr->child[pos]->child[0]=temp->child[temp->count];

temp->count--;

}

void btree:: move\_left(btnode \*curr,int pos)

{

int p;

btnode \*temp;

temp=curr->child[pos-1];

temp->count++;

temp->data[temp->count]=curr->data[pos];

temp->child[temp->count]=curr->child[pos]->child[0];

temp=curr->child[pos];

curr->data[pos]=temp->data[1];

temp->child[0]=temp->child[1];

temp->count--;

for(p=1;p<=temp->count;p++)

{

temp->data[p]=temp->data[p+1];

temp->child[p]=temp->child[p+1];

}

}

void btree:: combine\_nodes(btnode \*curr,int pos)

{

int p;

btnode \*left,\*right;

left=curr->child[pos-1];

right=curr->child[pos];

left->count++;

left->data[left->count]=curr->data[pos];

left->child[left->count]=right->child[0];

for(p=1;p<=right->count;p++)

{

left->count++;

left->data[left->count]=right->data[p];

left->child[left->count]=right->child[p];

}

for(p=pos;p<curr->count;p++)

{

curr->data[p]=curr->data[p+1];

curr->child[p]=curr->child[p+1];

}

curr->count--;

delete right;

}

int btree::search\_node(int newkey,btnode\* curr,int\* pos)

{

if(newkey<curr->data[1])

{

\*pos=0;

return 0;

}

else

{

\*pos=curr->count;

while((newkey<curr->data[\*pos])&&(\*pos>1))

(\*pos)--;

if(newkey==curr->data[\*pos])

return 1;

else

return 0;

}

}

btnode \*btree:: search(int newkey,btnode \*root,int \*pos)

{

if(!root)

{

return NULL;

}

else if(search\_node(newkey,root,pos))

return root;

else

return search(newkey,root->child[\*pos],pos);

}

btnode \*btree:: insert(int newdata,btnode \*root)

{

int meddata;

btnode \*medright,\*newroot;

if(push\_down(newdata,root,&meddata,&medright))

{

newroot=new btnode;

newroot->count=1;

newroot->data[1]=meddata;

newroot->child[0]=root;

newroot->child[1]=medright;

return newroot;

}

return root;

}

int btree:: push\_down(int newdata,btnode \*curr,int \*meddata,btnode \*\*medright)

{

int pos;

if(curr==NULL)

{

\*meddata=newdata;

\*medright=NULL;

return 1;

}

else

{

if(search\_node(newdata,curr,&pos))

cout<<"\n\n Error Duplicate keys cannot be inserted!";

if(push\_down(newdata,curr->child[pos],meddata,medright))

if(curr->count<max)

{

pushin(\*meddata,\*medright,curr,pos);

return 0;

}

else

{

split\_node(\*meddata,\*medright,curr,pos,meddata,medright);

return 1;

}

return 0;

}

}

void btree:: pushin(int meddata,btnode \*medright,btnode \*curr, int pos)

{

int p;

for(p=curr->count;p>pos;p--)

{

curr->data[p+1]=curr->data[p];

curr->child[p+1]=curr->child[p];

}

curr->data[pos+1]=meddata;

curr->child[pos+1]=medright;

curr->count++;

}

void btree::split\_node(int meddata,btnode \*medright,btnode \*curr,int pos,int \*newmedian,btnode \*\*newright)

{

int p,median;

if(pos<=min)

{

median=min;

}

else

{

median=min+1;

}

\*newright=new btnode;

for(p=median+1;p<=max;p++)

{

(\*newright)->data[p-median]=curr->data[p];

(\*newright)->child[p-median]=curr->child[p];

}

(\*newright)->count=max-median;

curr->count=median;

if(pos<=min)

{

pushin(meddata,medright,curr,pos);

}

else

{

pushin(meddata,medright,\*newright,pos - median);

}

\*newmedian=curr->data[curr->count];

(\*newright)->child[0]=curr->child[curr->count];

curr->count--;

}

void main()

{

int ch,c,n;

char ans;

btree b;

b.root=NULL;

clrscr();

do

{

cout<<"\n\t\t\*\*\*B-Tree Main Menu\*\*\*";

cout<<"\n\n1. Insert a key";

cout<<"\n\n2.Display B-tree";

cout<<"\n\n3.Delete a key";

cout<<"\n\n4.exit"<<"\n\n Enter choice:";

ch=getche();

ch=ch-'0';

switch(ch)

{

case 1:

do

{

cout<<"\n\n\n Enter data:";

cin>>n;

b.root=b.insert(n,b.root);

cout<<"\n\n Do you want to enter more keys?";

ans=getche();

if(ans=='n'||ans=='N')

break;

}

while(1);

getch();

break;

case 2:

b.pre\_rec(b.root);

getch();

break;

case 3:

cout<<"\n\n Enter key to be deleted:";

cin>>n;

b.root=b.del(n,b.root);

getch();

break;

case 4:

exit(0);

default:

cout<<"\n\t You have entered an invalid choice!";

getch();

break;

}

}while(1);

}

\*\*\*B-Tree Main Menu\*\*\*

1. Insert a key

2. Display B-tree

3. Delete a key

4. Exit

Enter choice: 1

Enter Data: 34

Do you want to Enter more keys? y

Enter Data: 87

Do you want to Enter more keys? y

Enter Data: 12

Do you want to Enter more keys? y

Enter Data: 67

Do you want to Enter more keys? y

Enter Data: 5

Do you want to Enter more keys? n

\*\*\*B-Tree Main Menu\*\*\*

1. Insert a key

2. Display B-tree

3. Delete a key

4. Exit

Enter choice: 2

5 12 34 67 87

\*\*\*B-Tree Main Menu\*\*\*

1. Insert a key

2. Display B-tree

3. Delete a key

4. Exit

Enter choice: 3

Enter key to be deleted: 34

\*\*\*B-Tree Main Menu\*\*\*

1. Insert a key

2. Display B-tree

3. Delete a key

4. Exit

Enter choice: 2

5 12 67 87

\*\*\*B-Tree Main Menu\*\*\*

1. Insert a key

2. Display B-tree

3. Delete a key

4. Exit

Enter choice: 4