Aman Ullah

[COMPANY NAME][Company address]

**1. Linked List**

• Singly

#include<iostream>

#include<string>

using namespace std;

class node {

public:

int data;

node \*next;

};

class Singly\_LL {

private:

node \*head;

public:

Singly\_LL() {

head = NULL;

}

bool isEmpty() {

if (head == NULL) { return true; }

else { return false; }

}

bool InsertAtHead(int Val) {

node \*newNode = new node;

newNode->data = Val;

if (isEmpty()) {

head = newNode;

newNode->next = NULL;

return true;

}

newNode->next = head;

head = newNode;

return true;

}

bool InsertAtTail(int Val) {

node \*newNode = new node;

newNode->data = Val;

if (isEmpty()) {

head = newNode;

newNode->next = NULL;

return true;

}

node \*curr = head;

while (curr->next != NULL) {

curr = curr->next;

}

// newNode->next = curr->next;

curr->next = newNode;

return true;

}

bool Insertion(int Val, int pos) {

node \*newNode = new node;

newNode->data = Val;

if (isEmpty()) {

InsertAtHead(Val);

return true;

}

node \*curr = head;

int currIndex = 2;

while (curr && currIndex < pos) {

curr = curr->next;

currIndex++;

}

if (pos > 1 && curr == NULL) {

return false;

}

if (pos == 1) {

newNode->next = head;

head = newNode;

return true;

}

else {

// newNode->next=curr->next;

curr->next = newNode;

return true;

}

}

bool DeleteAtHead() {

if (isEmpty()) {

cout << " Linked List is Already Empty." << endl;

return false;

}

node \*newNode = head;

head = head->next;

delete newNode;

}

bool DeleteAtTail() {

if (isEmpty()) {

cout << " Linked List is Already Empty." << endl;

return false;

}

node \*curr = head;

while (curr->next->next != NULL) {

curr = curr->next;

}

curr->next = NULL;

delete curr->next;

return true;

}

bool Deletion(int Val) {

if (isEmpty()) {

cout << " Linked List Already Empty." << endl;

return false;

}

node \*curr = head;

node \*prev = NULL;

while (curr->data != Val) {

prev = curr;

curr = curr->next;

}

if (curr) {

if (prev) {

prev->next = curr->next;

delete curr;

}

else {

head = head->next;

delete curr;

}

}

}

void Sort() {

node \*curr\_1 = head;

node \*curr\_2 = NULL;

while (curr\_1 != NULL) {

curr\_2 = curr\_1->next;

while (curr\_2 != NULL) {

if (curr\_1->data > curr\_2->data) {

int temp = curr\_1->data;

curr\_1->data = curr\_2->data;

curr\_2->data = temp;

}

curr\_2 = curr\_2->next;

}

curr\_1 = curr\_1->next;

}

}

node \*getHead() {

return head;

}

void Reverse() {

node \*prev = NULL;

node \*curr = head;

node \*Next = NULL;

while (curr != NULL) {

Next = curr->next;

curr->next = prev;

prev = curr;

curr = Next;

}

head = prev;

}

void Display\_Reverse(node \*Temp) {

if (Temp) {

Display\_Reverse(Temp->next);

cout << " " << Temp->data << " ";

}

}

void Display() {

node \*temp = head;

cout << " Singly Linked List = {";

while (temp != NULL) {

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

temp = temp->next;

}

cout << "}" << endl;

}

};

int main()

{

Singly\_LL ll;

ll.InsertAtHead(1);

ll.InsertAtHead(9);

ll.InsertAtHead(8);

ll.InsertAtHead(7);

ll.InsertAtHead(6);

ll.InsertAtTail(5);

ll.InsertAtTail(4);

ll.InsertAtTail(3);

ll.InsertAtTail(1);

ll.Display();

ll.Deletion(7);

ll.Display();

ll.Insertion(99, 9);

ll.Display();

ll.Sort();

ll.Display();

ll.Display\_Reverse(ll.getHead());

cout << endl;

ll.Reverse();

ll.Display();

system("pause");

return 0;

}

* • Doubly

#include<iostream>

#include<string>

using namespace std;

class node {

public:

int data;

node \*next, \*prev;

};

class Doubly\_LL {

private:

node \*head, \*tail;

public:

Doubly\_LL() {

head = tail = NULL;

}

bool isEmpty() {

if (head == NULL) { return true; }

else { return false; }

}

bool InsertAtHead(int Val) {

node \*newNode = new node;

newNode->data = Val;

newNode->next = NULL;

if (isEmpty()) {

head = tail = newNode;

newNode->next = NULL;

return true;

}

newNode->next = head;

head->prev = newNode;

head = newNode;

return true;

}

bool InsertAtTail(int Val) {

node \*newNode = new node;

newNode->data = Val;

if (isEmpty()) {

head = tail = newNode;

newNode->next = NULL;

return true;

}

while (tail->next != NULL) {

tail = tail->next;

}

tail->next = newNode;

newNode->prev = tail;

tail = tail->next;

return true;

}

bool Insertion(int Val, int pos) {

node \*newNode = new node;

newNode->data = Val;

int currIndex = 2;

node \*curr = head;

while (curr&&currIndex < pos) {

curr = curr->next;

currIndex++;

}

if (curr == NULL && pos > 1) {

return false;

}

if (pos == 1) {

InsertAtHead(Val);

return true;

}

else {

newNode->prev = curr;

newNode->next = curr->next;

newNode->next->prev = newNode;

newNode->prev->next = newNode;

return true;

}

}

bool DeleteATHead() {

if (isEmpty()) {

cout << " Linked List is Already Empty." << endl;

return false;

}

node \*newNode = head;

head = head->next;

head->prev = NULL;

delete newNode;

}

bool DeleteAtTail() {

if (isEmpty()) {

cout << " Linked List is Already Empty." << endl;

return false;

}

node \*curr = head;

while (curr->next->next != NULL) {

curr = curr->next;

}

curr->next = NULL;

tail = tail->prev;

delete curr->next;

}

bool Deletion(int Val) {

if (isEmpty()) {

cout << " Linked List is Already Empty." << endl;

return false;

}

node \*Prev = NULL;

node \*curr = head;

while (curr->data != Val) {

Prev = curr;

curr = curr->next;

}

if (curr) {

if (Prev) {

Prev = curr;

Prev->next->prev = curr->prev;

curr->prev->next = curr->next;

}

else {

head = head->next;

delete curr;

}

}

}

void Display() {

node \*temp = head;

cout << " Doubly Linked List = {";

while (temp != NULL) {

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

temp = temp->next;

}

cout << "}" << endl;

}

};

int main()

{

Doubly\_LL ll;

ll.InsertAtHead(1);

ll.InsertAtHead(9);

ll.InsertAtHead(8);

ll.InsertAtHead(7);

ll.InsertAtHead(6);

ll.InsertAtTail(5);

ll.InsertAtTail(4);

ll.InsertAtTail(3);

ll.InsertAtTail(1);

ll.Display();

ll.DeleteATHead();

ll.DeleteAtTail();

ll.Deletion(9);

ll.Display();

ll.Insertion(99, 5);

ll.Display();

system("pause");

return 0;

}

* • Circular Singly

#include<iostream>

#include<string>

using namespace std;

class node {

public:

int data;

node \*next;

};

class Singly\_LL {

private:

node \*head, \*tail;

public:

Singly\_LL() {

head = tail = NULL;

}

bool isEmpty() {

if (head == NULL) { return true; }

else { return false; }

}

bool InsertAtHead(int Val) {

node \*newNode = new node;

newNode->data = Val;

if (isEmpty()) {

head = tail = newNode;

newNode->next = head;

return true;

}

newNode->next = head;

head = newNode;

tail->next = head;

return true;

}

bool InsertAtTail(int Val) {

node \*newNode = new node;

newNode->data = Val;

if (isEmpty()) {

head = newNode;

newNode->next = head;

return true;

}

while (tail->next != head) {

tail = tail->next;

}

tail->next = newNode;

tail = tail->next;

tail->next = head;

return true;

}

bool Insertion(int Val, int pos) {

node \*newNode = new node;

newNode->data = Val;

if (isEmpty()) {

InsertAtHead(Val);

return true;

}

node \*curr = head;

int currIndex = 2;

while (curr && currIndex < pos) {

curr = curr->next;

currIndex++;

}

if (pos > 1 && curr == NULL) {

return false;

}

if (pos == 1) {

newNode->next = head;

head = newNode;

return true;

}

else {

// newNode->next=curr->next;

curr->next = newNode;

return true;

}

}

bool DeleteAtHead() {

if (isEmpty()) {

cout << " Linked List is Already Empty." << endl;

return false;

}

node \*newNode = head;

head = head->next;

tail->next = head;

delete newNode;

}

bool DeleteAtTail() {

if (isEmpty()) {

cout << " Linked List is Already Empty." << endl;

return false;

}

node \*curr = head;

while (curr->next->next != head) {

curr = curr->next;

}

curr->next = NULL;

delete curr->next;

curr->next = head;

return true;

}

bool Deletion(int Val) {

if (isEmpty()) {

cout << " Linked List Already Empty." << endl;

return false;

}

node \*curr = head;

node \*prev = NULL;

while (curr->data != Val) {

prev = curr;

curr = curr->next;

}

if (curr) {

if (prev) {

prev->next = curr->next;

delete curr;

}

else {

head = head->next;

delete curr;

}

}

}

void Sort() {

node \*curr\_1 = head;

node \*curr\_2 = NULL;

while (curr\_1->next != head) {

curr\_2 = curr\_1->next;

while (curr\_2 != head) {

if (curr\_1->data > curr\_2->data) {

int temp = curr\_1->data;

curr\_1->data = curr\_2->data;

curr\_2->data = temp;

}

curr\_2 = curr\_2->next;

}

curr\_1 = curr\_1->next;

}

}

node \*getHead() {

return head;

}

void Display\_Reverse(node \*Temp) {

if (Temp) {

Display\_Reverse(Temp->next);

cout << " " << Temp->data << " ";

}

}

void Display() {

node \*temp = head;

cout << " Singly Linked List = {";

while (temp->next != head) {

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

temp = temp->next;

}

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

cout << "}" << endl;

}

};

int main()

{

Singly\_LL ll;

ll.InsertAtHead(1);

ll.InsertAtHead(9);

ll.InsertAtHead(8);

ll.InsertAtHead(7);

ll.InsertAtHead(6);

ll.InsertAtTail(5);

ll.InsertAtTail(4);

ll.InsertAtTail(3);

ll.InsertAtTail(1);

ll.Display();

ll.DeleteAtHead();

ll.DeleteAtTail();

ll.Display();

ll.Sort();

ll.Display();

system("pause");

return 0;

}

* • Doubly Circular

#include<iostream>

#include<string>

using namespace std;

class node {

public:

int data;

node \*next, \*prev;

};

class Doubly\_LL {

private:

node \*head, \*tail;

public:

Doubly\_LL() {

head = tail = NULL;

}

bool isEmpty() {

if (head == NULL) { return true; }

else { return false; }

}

bool InsertAtHead(int Val) {

node \*newNode = new node;

newNode->data = Val;

newNode->next = NULL;

if (isEmpty()) {

head = tail = newNode;

tail->next = head;

head->prev = tail;

return true;

}

newNode->next = head;

head->prev = newNode;

head = newNode;

head->prev = tail;

tail->next = head;

return true;

}

bool InsertAtTail(int Val) {

node \*newNode = new node;

newNode->data = Val;

if (isEmpty()) {

head = tail = newNode;

tail->next = head;

head->prev = tail;

return true;

}

while (tail->next != head) {

tail = tail->next;

}

tail->next = newNode;

newNode->prev = tail;

tail = tail->next;

tail->next = head;

head->prev = tail;

return true;

}

bool Insertion(int Val, int pos) {

node \*newNode = new node;

newNode->data = Val;

int currIndex = 2;

node \*curr = head;

while (curr&&currIndex < pos) {

curr = curr->next;

currIndex++;

}

if (curr == NULL && pos > 1) {

return false;

}

if (pos == 1) {

InsertAtHead(Val);

return true;

}

else {

newNode->prev = curr;

newNode->next = curr->next;

newNode->next->prev = newNode;

newNode->prev->next = newNode;

return true;

}

}

bool DeleteATHead() {

if (isEmpty()) {

cout << " Linked List is Already Empty." << endl;

return false;

}

node \*newNode = head;

head = head->next;

head->prev = tail;

tail->next = head;

delete newNode;

}

bool DeleteAtTail() {

if (isEmpty()) {

cout << " Linked List is Already Empty." << endl;

return false;

}

node \*curr = head;

while (curr->next->next != head) {

curr = curr->next;

}

curr->next = NULL;

delete curr->next;

tail = tail->prev;

tail->next = head;

head->prev = tail;

return true;

}

bool Deletion(int Val) {

if (isEmpty()) {

cout << " Linked List is Already Empty." << endl;

return false;

}

node \*Prev = NULL;

node \*curr = head;

while (curr->data != Val) {

Prev = curr;

curr = curr->next;

}

if (curr) {

if (Prev) {

Prev = curr;

Prev->next->prev = curr->prev;

curr->prev->next = curr->next;

}

else {

head = head->next;

delete curr;

}

}

}

void Display() {

node \*temp = head;

cout << " Doubly Linked List = {";

while (temp->next != head) {

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

temp = temp->next;

}

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

cout << "}" << endl;

}

};

int main()

{

Doubly\_LL ll;

ll.InsertAtHead(1);

ll.InsertAtHead(9);

ll.InsertAtHead(8);

ll.InsertAtHead(7);

ll.InsertAtHead(6);

ll.InsertAtTail(5);

ll.InsertAtTail(4);

ll.InsertAtTail(3);

ll.InsertAtTail(1);

ll.Display();

ll.DeleteATHead();

ll.DeleteAtTail();

ll.Deletion(9);

ll.Display();

ll.Insertion(99, 5);

ll.Display();

system("pause");

return 0;

}

**2. Stack**

• Array Based

#include<iostream>

#include<string>

using namespace std;

class Stack {

int \*stack;

int Size, top;

public:

Stack(int Size) {

this->Size = Size;

stack = new int[Size];

for (int i = 0; i < Size; i++) {

stack[i] = 0;

}

top = -1;

}

bool isEmpty() {

if (top == -1) { return true; }

else { return false; }

}

bool isFull() {

if (top == Size - 1) { return true; }

else { return false; }

}

bool push(int Val) {

if (isFull()) {

cout << " Stack is Full" << endl;

return false;

}

top = top + 1;

stack[top] = Val;

return true;

}

bool pop() {

if (isEmpty()) {

cout << " Stack is Empty" << endl;

return false;

}

top = top - 1;

return true;

}

void Display() {

cout << " Stack = ";

for (int i = 0; i <= top; i++) {

cout << " " << stack[i] << " ";

}

cout << endl;

}

};

int main()

{

Stack s(6);

s.push(1);

s.push(2);

s.push(3);

s.push(4);

s.push(5);

s.push(6);

s.Display();

s.pop();

s.Display();

system("pause");

return 0;

}

* • Linked List Based

#include<iostream>

#include<string>

using namespace std;

class node {

public:

int data;

node \*next;

};

class Stack {

node \*top;

public:

Stack() {

top = NULL;

}

bool isEmpty() {

if (top == NULL) { return true; }

else { return false; }

}

bool push(int Val) {

node \*newNode = new node;

newNode->data = Val;

if (isEmpty()) {

top = newNode;

newNode->next = NULL;

return true;

}

newNode->next = top;

top = newNode;

return true;

}

bool pop() {

if (isEmpty()) {

cout << " Stack is Empty" << endl;

return false;

}

node \*newNode = top;

top = top->next;

delete newNode;

}

void Display() {

cout << " Stack = ";

node \*temp = top;

while (temp != NULL) {

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

temp = temp->next;

}

cout << endl;

}

};

int main()

{

Stack s;

s.push(1);

s.push(2);

s.push(3);

s.push(4);

s.push(5);

s.push(6);

s.Display();

s.pop();

s.Display();

system("pause");

return 0;

}

**3. Queue**

* • Array Based

#include<iostream>

#include<string>

using namespace std;

class Queue {

private:

int \*queue;

int Size, front, rear;

public:

Queue(int Size) {

this->Size = Size;

queue = new int[Size];

for (int i = 0; i < Size; i++) {

queue[i] = 0;

}

front = rear = -1;

}

bool isEmpty() {

if (front == -1) { return true; }

else { return false; }

}

bool isFull() {

if ((rear + 1) % Size == front) { return true; }

else { return false; }

}

bool EnQueue(int Val) {

if (isFull()) {

cout << " Queue is Full" << endl;

return false;

}

if (isEmpty()) {

front = rear = 0;

}

else {

rear = (rear + 1) % Size;

}

queue[rear] = Val;

return true;

}

bool DeQueue() {

if (isEmpty()) {

cout << " Queue is Empty" << endl;

return false;

}

if (front == rear) {

front = rear = -1;

}

else {

front = (front + 1) % Size;

return true;

}

}

void Display() {

cout << " Queue = ";

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

cout << " " << queue[i] << " ";

}

cout << endl;

}

};

int main()

{

Queue q(5);

q.EnQueue(1);

q.EnQueue(2);

q.EnQueue(3);

q.EnQueue(4);

q.EnQueue(5);

q.Display();

q.DeQueue();

q.Display();

system("pause");

return 0;

}

* • Linked List Based

#include<iostream>

#include<string>

using namespace std;

class node {

public:

int data;

node \*next;

};

class Queue {

private:

node \*front, \*rear;

public:

Queue() {

front = rear = NULL;

}

bool isEmpty() {

if (front == NULL) { return true; }

else { return false; }

}

bool EnQueue(int Val) {

node \*newNode = new node;

newNode->data = Val;

if (isEmpty()) {

front = rear = newNode;

newNode->next = NULL;

return true;

}

rear->next = newNode;

rear = rear->next;

rear->next = NULL;

return true;

}

bool DeQueue() {

if (isEmpty()) {

cout << " Queue is Empty" << endl;

return false;

}

node \*newNode = front;

front = front->next;

delete newNode;

}

void Display() {

cout << " Queue = ";

node \*temp = front;

while (temp != NULL) {

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

temp = temp->next;

}

cout << endl;

}

};

int main()

{

Queue q;

q.EnQueue(1);

q.EnQueue(2);

q.EnQueue(3);

q.EnQueue(4);

q.EnQueue(5);

q.Display();

q.DeQueue();

q.Display();

system("pause");

return 0;

}

* • Double Ended Queue Array Based

#include<iostream>

#include<string>

using namespace std;

class Queue {

private:

int \*queue;

int Size, front, rear;

public:

Queue(int Size) {

this->Size = Size;

queue = new int[Size];

for (int i = 0; i < Size; i++) {

queue[i] = 0;

}

front = rear = -1;

}

bool isEmpty() {

if (front == -1) { return true; }

else { return false; }

}

bool isFull() {

if ((rear + 1) % Size == front) { return true; }

else { return false; }

}

bool EnQueue\_Rear(int Val) {

if (isFull()) {

cout << " Queue is Full" << endl;

return false;

}

if (isEmpty()) {

front = rear = 0;

}

else {

rear = (rear + 1) % Size;

}

queue[rear] = Val;

return true;

}

bool EnQueue\_Front(int Val) {

if (isFull()) {

cout << " Queue is Full" << endl;

return false;

}

if (isEmpty()) {

front = rear = 0;

}

else if (front == 0) {

front = Size - 1;

}

else {

front = (front - 1) % Size;

}

queue[front] = Val;

return true;

}

bool DeQueue\_Front() {

if (isEmpty()) {

cout << " Queue is Empty" << endl;

return false;

}

if (front == rear) {

front = rear = -1;

}

else {

queue[front] = 0;

front = (front + 1) % Size;

return true;

}

}

bool DeQueue\_Rear() {

if (isEmpty()) {

cout << " DEQUEUE is Empty" << endl;

return false;

}

if (front == rear) {

front = rear = -1;

}

else {

queue[rear] = 0;

rear = (rear - 1) % Size;

return true;

}

}

void Display() {

cout << " Queue = ";

for (int i = 0; i < Size; i++) {

cout << " " << queue[i] << " ";

}

cout << endl;

}

};

int main()

{

Queue q(5);

q.EnQueue\_Rear(1);

q.EnQueue\_Front(2);

q.EnQueue\_Front(3);

q.EnQueue\_Front(4);

q.EnQueue\_Front(5);

q.Display();

q.DeQueue\_Front();

q.DeQueue\_Rear();

q.Display();

system("pause");

return 0;

}

* • Double Ended Queue LL Based

#include<iostream>

#include<string>

using namespace std;

class node {

public:

int data;

node \*next;

};

class Queue {

private:

node \*front, \*rear;

public:

Queue() {

front = rear = NULL;

}

bool isEmpty() {

if (front == NULL) { return true; }

else { return false; }

}

bool EnQueue\_Rear(int Val) {

node \*newNode = new node;

newNode->data = Val;

if (isEmpty()) {

front = rear = newNode;

newNode->next = NULL;

return true;

}

rear->next = newNode;

rear = rear->next;

rear->next = NULL;

return true;

}

bool EnQueue\_Front(int Val) {

node \*newNode = new node;

newNode->data = Val;

if (isEmpty()) {

front = rear = newNode;

newNode->next = NULL;

return true;

}

newNode->next = front;

front = newNode;

return true;

}

bool DeQueue\_Front() {

if (isEmpty()) {

cout << " Queue is Empty" << endl;

return false;

}

node \*newNode = front;

front = front->next;

delete newNode;

}

bool DeQueue\_Rear() {

if (isEmpty()) {

cout << " Queue is Empty" << endl;

return false;

}

node \*Temp = front;

while (Temp->next->next != NULL) {

Temp = Temp->next;

}

rear = Temp;

Temp->next = NULL;

delete Temp->next;

}

void Display() {

cout << " Queue = ";

node \*temp = front;

while (temp != NULL) {

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

temp = temp->next;

}

cout << endl;

}

};

int main()

{

Queue q;

q.EnQueue\_Front(1);

q.EnQueue\_Rear(2);

q.EnQueue\_Rear(3);

q.EnQueue\_Rear(4);

q.EnQueue\_Front(5);

q.Display();

q.DeQueue\_Front();

q.DeQueue\_Rear();

q.Display();

system("pause");

return 0;

}

* • Priority Queue

#include<iostream>

#include<string>

using namespace std;

class node {

public:

int data;

int Priority;

node \*next;

node() {

data = Priority = NULL;

next = NULL;

}

};

class Priority\_Queue {

private:

node \*front, \*rear;

public:

Priority\_Queue() {

front = rear = NULL;

}

bool isEmpty() {

if (front == NULL) { return true; }

else { return false; }

}

bool EnQueue(int Val,int Priority) {

node \*newNode = new node;

newNode->data = Val;

newNode->Priority = Priority;

if (isEmpty()) {

front = newNode;

//newNode->next = NULL;

return true;

}

if (Priority < front->Priority) {

newNode->next = front;

front = newNode;

return true;

}

else {

node \*curr = front;

while (curr && curr->next->Priority < Priority) {

curr = curr->next;

}

newNode->next = curr->next;

curr->next = newNode;

newNode = curr;

return true;

}

}

bool DeQueue() {

if (isEmpty()) {

cout << " Priority Queue is Empty" << endl;

return false;

}

node \*Temp = front;

front = front->next;

delete Temp;

}

void Display() {

node \*Temp = front;

cout << " Priority\_Queue = ";

while (Temp != NULL) {

cout << " " << Temp->data << " ";

Temp = Temp->next;

}

cout << endl;

}

};

int main()

{

Priority\_Queue pq;

pq.EnQueue(1, 50);

pq.EnQueue(2, 40);

pq.EnQueue(3, 30);

pq.EnQueue(4, 20);

pq.EnQueue(5, 10);

pq.EnQueue(99, 25);

pq.Display();

system("pause");

return 0;

}

**4. BST**

#include<iostream>

#include<string>

using namespace std;

class node {

public:

int data;

node \*left, \*right;

};

class AVL {

private:

node \*root;

node \*Insert(node \*Root, int Value) {

if (Root == NULL) {

Root = new node;

Root->data = Value;

Root->left = Root->right = NULL;

}

else if (Value < Root->data) {

Root->left = Insert(Root->left, Value);

}

else if (Value > Root->data) {

Root->right = Insert(Root->right, Value);

}

return Root;

}

node \*Search(node \*Root, int Value) {

if (Root == NULL) {

return NULL;

}

else if (Value < Root->data) {

Root->left = Search(Root->left, Value);

}

else if (Value > Root->data) {

Root->right = Search(Root->right, Value);

}

else {

return Root;

}

}

node \*Max\_Finder(node \*Root) {

if (Root == NULL) { return NULL; }

else if (Root->right == NULL) { return Root; }

else { return Max\_Finder(Root->right); }

}

node \*Delete(node \*Root, int Value) {

if (Root == NULL) {

return NULL;

}

else if (Value < Root->data) {

Root->left = Delete(Root->left, Value);

}

else if (Value > Root->data) {

Root->right = Delete(Root->right, Value);

}

else if (Root->left && Root->right) {

node \*Temp = Max\_Finder(Root->left);

Root->data = Temp->data;

Root->left = Delete(Root->left, Root->data);

}

else {

node \*Temp = Root;

if (Root->left == NULL) {

Root = Root->right;

}

else if (Root->right == NULL) {

Root = Root->left;

}

delete Temp;

}

return Root;

}

void Display\_InOrder(node \*InOrder) {

if (InOrder) {

Display\_InOrder(InOrder->left);

cout << " " << InOrder->data << " ";

Display\_InOrder(InOrder->right);

}

}

int Maximum(node \*Root) {

if (Root) {

Maximum(Root->right);

}

return Root->data;

}

int Minimum(node \*Root) {

if (Root) {

Minimum(Root->left);

}

return Root->data;

}

int Height\_of\_BST(node \*Root) {

if (Root == NULL) {

return 0;

}

int Left = Height\_of\_BST(Root->left);

int Right = Height\_of\_BST(Root->right);

if (Left > Right) {

return (Left + 1);

}

else {

return (Right + 1);

}

}

int All\_Nodes\_Count(node \*Root) {

int count = 0;

if (Root == NULL) {

return 0;

}

else {

count = count + All\_Nodes\_Count(Root->left) + All\_Nodes\_Count(Root->right);

}

return count;

}

int Identical\_Trees(node \*First, node \*Second) {

if (First == NULL && Second == NULL) {

return 0;

}

if (First != NULL && Second != NULL) {

return (First->data && Second->data && Identical\_Trees(First->left, Second->left) && Identical\_Trees(First->right, Second->right));

}

return 0;

}

int Leaf\_Count(node \*Root) {

if (Root == NULL) {

return 0;

}

if (Root->left == NULL && Root->right == NULL) {

return 1;

}

else {

return Leaf\_Count(Root->left) + Leaf\_Count(Root->right);

}

}

public:

AVL() {

root = NULL;

}

void Insertion(int Value) {

root = Insert(root, Value);

}

void Deletion(int Value) {

root = Delete(root, Value);

}

void Searching(int Value) {

root = Search(root, Value);

}

void Display() {

cout << " InOrder = ";

Display\_InOrder(root);

cout << endl;

}

void Height() {

cout << " Height = " << Height\_of\_BST(root) << endl;

}

void Leaf() {

cout << " Leaf Nodes = " << Leaf\_Count(root) << endl;

}

void Count() {

cout << " All Nodes = " << All\_Nodes\_Count(root) << endl;

}

};

int main()

{

AVL t;

t.Insertion(5);

t.Insertion(2);

t.Insertion(1);

t.Insertion(3);

t.Insertion(7);

t.Insertion(6);

t.Insertion(9);

t.Display();

t.Count();

t.Height();

t.Leaf();

system("pause");

return 0;

}

**5. AVL**

#include<iostream>

#include<string>

using namespace std;

class node {

public:

int data, height;

node \*left, \*right;

};

class AVL {

private:

node \*root;

node \*Insert(node \*Root, int Value) {

if (Root == NULL) {

Root = new node;

Root->data = Value;

Root->left = Root->right = NULL;

}

else if (Value < Root->data) {

Root->left = Insert(Root->left, Value);

}

else if (Value > Root->data) {

Root->right = Insert(Root->right, Value);

}

Root->height = 1 + Max(Height(Root->left), Height(Root->right));

int BF = Balance\_Factor(Root);

if (BF > 1 && Value < Root->left->data) {

return Right\_Rotation(Root);

}

if (BF<-1 && Value>Root->right->data) {

return Left\_Rotation(Root);

}

if (BF > 1 && Value > Root->left->data) {

Root->left = Left\_Rotation(Root->left);

return Right\_Rotation(Root);

}

if (BF < -1 && Value < Root->right->data) {

Root->right = Right\_Rotation(Root->right);

return Left\_Rotation(Root);

}

return Root;

}

node \*Search(node \*Root, int Value) {

if (Root == NULL) {

return NULL;

}

else if (Value < Root->data) {

Root->left = Search(Root->left, Value);

}

else if (Value > Root->data) {

Root->right = Search(Root->right, Value);

}

else {

return Root;

}

}

node \*Delete(node \*Root, int Value) {

if (Root == NULL) {

return Root;

}

else if (Value < Root->data) {

Root->left = Delete(Root->left, Value);

}

else if (Value > Root->data) {

Root->right = Delete(Root->right, Value);

}

else if (Root->left && Root->right) {

node \*Temp = Max\_Finder(Root->left);

Root->data = Temp->data;

Root->left = Delete(Root->left, Root->data);

}

else {

node \*Temp = Root;

if (Root->left == NULL) {

Root = Root->right;

}

else if (Root->right == NULL) {

Root = Root->left;

}

delete Temp;

}

if (Root == NULL) {

return Root;

}

Root->height = 1 + Max(Height(Root->left), Height(Root->right));

int BF = Balance\_Factor(Root);

if (BF > 1 && Balance\_Factor(Root->left) >= 0) {

return Right\_Rotation(Root);

}

if (BF > 1 && Balance\_Factor(Root->left) < 0) {

Root->left = Left\_Rotation(Root->left);

return Right\_Rotation(Root);

}

if (BF < -1 && Balance\_Factor(Root->right) <= 0) {

return Left\_Rotation(Root);

}

if (BF < -1 && Balance\_Factor(Root->right) > 0) {

Root->right = Right\_Rotation(Root->right);

return Left\_Rotation(Root);

}

return Root;

}

node \*Max\_Finder(node \*Root) {

if (Root == NULL) { return NULL; }

else if (Root->right == NULL) { return Root; }

else { return Max\_Finder(Root->right); }

}

int Max(int First\_Num, int Second\_Num) {

if (First\_Num > Second\_Num) {

return First\_Num;

}

else {

return Second\_Num;

}

}

int Height(node \*Root) {

if (Root == NULL) {

return NULL;

}

else {

return Root->height;

}

}

int Balance\_Factor(node \*Root) {

if (Root == NULL) {

return NULL;

}

else {

return (Height(Root->left) - Height(Root->right));

}

}

node \*Right\_Rotation(node \*Root) {

node \*Current = Root->left;

node \*Temp = Current->right;

Current->right = Root;

Root->left = Temp;

Root->height = 1 + Max(Height(Root->left), Height(Root->right));

Current->height = 1 + Max(Height(Current->left), Height(Current->right));

return Current;

}

node \*Left\_Rotation(node \*Root) {

node \*Current = Root->right;

node \*Temp = Current->left;

Current->left = Root;

Root->right = Temp;

Root->height = 1 + Max(Height(Root->left), Height(Root->right));

Current->height = 1 + Max(Height(Current->left), Height(Current->right));

return Current;

}

void Display\_InOrder(node \*InOrder) {

if (InOrder) {

Display\_InOrder(InOrder->left);

cout << " " << InOrder->data << " ";

Display\_InOrder(InOrder->right);

}

}

public:

AVL() {

root = NULL;

}

void Insertion(int Value) {

root = Insert(root, Value);

}

void Deletion(int Value) {

root = Delete(root, Value);

}

void Searching(int Value) {

root = Search(root, Value);

}

void Display() {

cout << " InOrder = ";

Display\_InOrder(root);

cout << endl;

}

};

int main()

{

AVL a;

a.Insertion(1);

a.Insertion(2);

a.Insertion(3);

a.Insertion(4);

a.Insertion(5);

a.Display();

a.Deletion(2);

a.Display();

system("pause");

return 0;

}

**6. Heap**

#include<iostream>

#include<string>

using namespace std;

class Min\_Heap {

private:

int \*heap;

int Size, Max\_Size;

bool isEmpty() {

if (Size == 0) { return true; }

else { return false; }

}

bool isFull() {

if (Size == Max\_Size - 1) { return true; }

else { return false; }

}

void Heapify\_Up() {

int Child = Size;

while (Child != 1) {

if (heap[Child] < heap[Child / 2]) { //Child<Parent

int Temp = heap[Child];

heap[Child] = heap[Child / 2];

heap[Child / 2] = Temp;

Child = Child / 2;

}

else {

break;

}

}

}

void Heapify\_Down() {

int Parent = 1;

while (Parent < Size / 2 || Size == 3) {

if (heap[Parent \* 2] < heap[(Parent \* 2) + 1]) {// Left\_Child < Right\_Child

int Temp = heap[Parent];

heap[Parent] = heap[Parent \* 2];

heap[Parent \* 2] = Temp;

Parent = Parent \* 2;

}

else if (heap[(Parent \* 2) + 1] < heap[Parent \* 2]) {//Right\_Child < Left\_Child

int Temp = heap[Parent];

heap[Parent] = heap[(Parent \* 2) + 1];

heap[(Parent \* 2) + 1] = Temp;

Parent = (Parent \* 2) + 1;

}

else {

break;

}

}

}

public:

Min\_Heap(int Max\_Size) {

this->Max\_Size = Max\_Size;

Size = 0;

heap = new int[Max\_Size];

for (int i = 0; i < Max\_Size; i++) {

heap[i] = 0;

}

}

bool Insert(int Value) {

if (isFull()) {

cout << " Insertion is not Possible" << endl;

return false;

}

Size = Size + 1;

heap[Size] = Value;

Heapify\_Up();

return true;

}

bool Delete() {

if (isEmpty()) {

cout << " Deletion is not Possible" << endl;

return false;

}

int Temp = heap[1];

heap[1] = heap[Size];

heap[Size] = Temp;

heap[Size] = NULL;

Size = Size - 1;

Heapify\_Down();

return true;

}

void Heap\_Sort(int \*&Heap\_Sort) {

int Arr\_Size = Size;

for (int i = 0; i < Arr\_Size; i++) {

Heap\_Sort[i] = heap[1];

Delete();

}

}

void Display() {

cout << " Min Heap = {";

for (int i = 1; i <= Size; i++) {

cout << " " << heap[i] << " ";

}

cout << "}" << endl;

}

};

int main()

{

Min\_Heap mh(8);

mh.Insert(7);

mh.Insert(6);

mh.Insert(5);

mh.Insert(4);

mh.Insert(3);

mh.Insert(2);

mh.Insert(1);

mh.Display();

int Size = 7;

int \*Heap = new int[Size];

mh.Heap\_Sort(Heap);

for (int i = 0; i < Size; i++) {

cout << " " << Heap[i] << " ";

}

cout << endl;

system("pause");

return 0;

}

**7. Graphs**

**Adjacency Matrix:**

#include<iostream>

#include<string>

using namespace std;

class Graph

{

private:

int \*\*Array;

int row, col;

int wieght;

public:

Graph() {}

Graph(int row, int col) {

this->row = row;

this->col = col;

Array = new int\*[row];

for (int i = 0; i < row; i++) {

Array[i] = new int[col];

}

for (int i = 0; i < row; i++) {

for (int j = 0; j < col; j++) {

Array[i][j] = 0;

}

}

}

bool Insert(int i, int j, int weight) {

Array[i][j] = weight;

Array[j][i] = weight;

return true;

}

void Display() {

int ch = 97;

cout << " ";

for (int i = 0; i < col; i++) {

cout << " " << char(ch++) << " ";

}

cout << endl << " ";

for (int i = 0; i < col; i++) {

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

}

cout << endl;

ch = 97;

for (int i = 0; i < row; i++) {

cout << " " << char(ch++) << " | ";

for (int j = 0; j < col; j++) {

cout << " " << Array[i][j] << " | ";

}

cout << endl;

}

}

};

int main()

{

Graph g(6, 6);

g.Insert(0, 1, 7);

g.Insert(0, 2, 9);

g.Insert(0, 5, 14);

g.Insert(1, 3, 10);

g.Insert(1, 4, 15);

g.Insert(2, 5, 2);

g.Insert(3, 2, 11);

g.Insert(3, 4, 6);

g.Insert(4, 5, 9);

cout << " Adjacency Matrix of the Given Graph:" << endl << endl;

g.Display();

cout << endl;

system("pause");

return 0;

}

**Adjacency List:**

#include<iostream>

#include<string>

using namespace std;

class node {

public:

int data;

node \*next;

};

class Adjacency\_List {

private:

node \*head, \*tail;

public:

Adjacency\_List() {

head = NULL;

tail = NULL;

}

bool Insertion\_in\_Adjacency\_List(int n) {

node \*temp = new node;

temp->data = n;

temp->next = NULL;

if (head == NULL) {

head = temp;

tail = temp;

}

else {

tail->next = temp;

tail = tail->next;

return true;

}

}

void Display() {

node \*temp = head;

while (temp != NULL) {

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

temp = temp->next;

}

}

};

class Graph\_by\_List {

private:

Adjacency\_List \*adj;

int n;

public:

Graph\_by\_List(int n) {

this->n = n;

adj = new Adjacency\_List[n];

}

void Insert\_Edge(int s, int d) {

adj[s].Insertion\_in\_Adjacency\_List(d);

adj[d].Insertion\_in\_Adjacency\_List(s);

}

// Print the graph

void Display\_List() {

static int i = 0;

for (int d = 0; d < n; ++d) {

if (d != n) {

cout << i++ << " ";

}

adj[d].Display();

cout << endl;

}

}

};

int main()

{

int No\_of\_Rows = 8, No\_of\_Coloumbs = 7;

Graph\_by\_Matrix g(No\_of\_Rows, No\_of\_Coloumbs);

int Arr[][7] = { {1, 2}, {2, 3}, {4, 5}, {1, 5}, {6, 1}, {7, 4}, {3, 8} };

for (int i = 0; i < 7; i++) {

for (int j = 0; j < 1; j++) {

g.Insert((Arr[i][j]) - 1, Arr[i][j + 1] - 1);

}

}

cout << " i) " << endl;

cout << " Adjacency MAtrix of the Given Graph:" << endl << endl;

g.Display();

cout << endl;

Graph\_by\_List gl(8);

for (int i = 0; i < 7; i++) {

for (int j = 0; j < 1; j++) {

gl.Insert\_Edge((Arr[i][j]) - 1, Arr[i][j + 1] - 1);

}

}

cout << " ii) " << endl;

cout << " Adjacency List of the Given Graph:" << endl << endl;

gl.Display\_List();

system("pause");

return 0;

}

**BFS and DFS by Adjacency List:**

#include<iostream>

#include<string>

using namespace std;

class node {

public:

int data;

node \*next;

};

class Queue {

node \*front, \*rear;

public:

Queue() {

front = rear = NULL;

}

bool isEmpty() {

if (front == NULL) { return true; }

else { return false; }

}

bool EnQueue(int Value) {

node \*newNode = new node;

newNode->data = Value;

if (isEmpty()) {

front = rear = newNode;

newNode->next = NULL;

return true;

}

rear->next = newNode;

rear = rear->next;

rear->next = NULL;

return true;

}

bool DeQueue() {

if (isEmpty()) {

cout << " Queue is Already Empty." << endl;

return false;

}

node \*Temp = front;

front = front->next;

delete Temp;

}

int getFront() {

if (front == NULL) { return NULL; }

return front->data;

}

};

class Stack {

node \*top;;

public:

Stack() {

top = NULL;

}

bool isEmpty() {

if (top == NULL) { return true; }

else { return false; }

}

bool Push(int Value) {

node \*newNode = new node;

newNode->data = Value;

if (isEmpty()) {

top = newNode;

newNode->next = NULL;

return true;

}

newNode->next = top;

top = newNode;

return true;

}

bool Pop() {

if (isEmpty()) {

cout << " Stack is Already Empty." << endl;

return false;

}

node \*Temp = top;

top = top->next;

delete Temp;

}

int getTop() {

if (top == NULL) { return NULL; }

return top->data;

}

};

class Adjacency\_List {

private:

node \*head, \*tail;

public:

Adjacency\_List() {

head = NULL;

tail = NULL;

}

bool Insertion\_in\_Adjacency\_List(int n) {

node \*temp = new node;

temp->data = n;

temp->next = NULL;

if (head == NULL) {

head = temp;

tail = temp;

}

else {

tail->next = temp;

tail = tail->next;

//tail->next = NULL;

return true;

}

}

void Display() {

node \*temp = head;

while (temp != NULL) {

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

temp = temp->next;

}

}

node \*getHead() {

return head;

}

};

class Graph\_by\_List {

private:

Adjacency\_List \*adj;

int n;

public:

Graph\_by\_List(int n) {

this->n = n;

adj = new Adjacency\_List[n];

}

void Insert\_Edge(int s, int d) {

adj[s].Insertion\_in\_Adjacency\_List(d);

adj[d].Insertion\_in\_Adjacency\_List(s);

}

void Bredth\_First\_Search(int Source) {

bool \*Visited = new bool[n];

Queue q;

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

Visited[i] = false;

}

Visited[Source] = true;

q.EnQueue(Source);

while (q.isEmpty() != true) {

int Curr = q.getFront();

cout << " " << Curr << " ";

q.DeQueue();

node \*temp = adj[Curr].getHead();

while (temp != NULL) {

int Adj\_Vertex = temp->data;

if (!Visited[Adj\_Vertex]) {

Visited[Adj\_Vertex] = true;

q.EnQueue(Adj\_Vertex);

}

temp = temp->next;

}

}

}

void Depth\_First\_Search(int Source) {

bool \*Visited = new bool[n];

Stack s;

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

Visited[i] = false;

}

Visited[Source] = true;

s.Push(Source);

while (!s.isEmpty()) {

int Curr = s.getTop();

cout << " " << Curr << " ";

s.Pop();

node \*temp = adj[Curr].getHead();

while (temp != NULL) {

int Adj\_Vertex = temp->data;

if (!Visited[Adj\_Vertex]) {

Visited[Adj\_Vertex] = true;

s.Push(Adj\_Vertex);

}

temp = temp->next;

}

}

}

// Print the graph

void Display\_List() {

static int i = 0;

for (int d = 0; d < n; ++d) {

if (d != n) {

cout << i++ << " ";

}

adj[d].Display();

cout << endl;

}

}

};

int main()

{

int Size = 6;

Graph\_by\_List gl(Size);

gl.Insert\_Edge(0, 1);

gl.Insert\_Edge(0, 4);

gl.Insert\_Edge(1, 2);

gl.Insert\_Edge(1, 5);

gl.Insert\_Edge(2, 3);

gl.Insert\_Edge(2, 5);

gl.Insert\_Edge(2, 4);

gl.Insert\_Edge(3, 5);

gl.Insert\_Edge(4, 5);

cout << " Adjacency List of the Given Graph:" << endl << endl;

gl.Display\_List();

cout << endl;

gl.Bredth\_First\_Search(0);

cout << endl << endl;

gl.Depth\_First\_Search(1);

cout << endl << endl;

system("pause");

return 0;

}

**8. Hashing**

#include<iostream>

#include<string>

using namespace std;

class Hash\_Table {

int \*Table;

int Size, count;

bool isEmpty() {

if (count == 0) { return true; }

else { return false; }

}

bool isFull() {

if (count == Size - 1) { return true; }

else { return false; }

}

int Linear\_Probing(int Value) {

int i = 0, key = 0;

while (Table[key] != 0) {

key = (Value + i) % Size;

i = i + 1;

}

return (key % Size);

}

int Linear\_Probing\_Step\_Size(int Value) {

int i = 0, key = 0;

while (Table[key] != 0) {

key = (Value + i) % Size;

i = i + 3;

}

return (key % Size);

}

int Quadratic\_Probing(int Value) {

int i = 0, key = 0;

while (Table[key] != 0) {

key = (Value + (i\*i)) % Size;

i = i + 1;

}

return (key % Size);

}

int Hash\_Function(int Value) {

return (Value%Size);

}

public:

Hash\_Table(int Size) {

this->Size = Size;

Table = new int[Size];

for (int i = 0; i < Size; i++) {

Table[i] = 0;

}

count = 0;

}

bool Insert(int Value) {

count++;

if (isFull()) {

cout << " Hash Table if Full" << endl;

return false;

}

int Index = Hash\_Function(Value);

if (Table[Index] == 0) {

Table[Index] = Value;

}

else {

int Linear = Quadratic\_Probing(Value);

Table[Linear] = Value;

}

return true;

}

void Display() {

cout << " Hash Table = ";

for (int i = 0; i < Size; i++) {

cout << " " << Table[i] << " ";

}

cout << endl;

}

};

int main()

{

Hash\_Table h(10);

h.Insert(5);

h.Insert(10);

h.Insert(15);

h.Insert(21);

h.Insert(28);

h.Insert(37);

h.Insert(41);

h.Insert(51);

h.Display();

system("pause");

return 0;

}

**Bucketing**

#include<iostream>

using namespace std;

class Buckets {

private:

int \*\*Bucket;

int Size, folds, counter, no\_of\_buckets;

int Hash\_Function(int Value) {

return (Value % Size);

}

int get\_Folding\_by\_Hash\_Function(int Value) {

int Org\_Value = Value;

int i = 0;

while ((\*(\*(Bucket + Org\_Value) + i)) != 0) {

i = i + 1;

}

return i;

}

void Re\_Hashing() {

cout << " Re\_Hashing applied on the basis of 70% load Factor:" << endl;

int old\_folds = folds;

folds = folds \* 3;

no\_of\_buckets = Size \* folds;

int \*\*Temp\_Arr = new int\*[Size];

for (int i = 0; i < Size; i++) {

\*(Temp\_Arr + i) = new int[old\_folds];

}

for (int i = 0; i < Size; i++) {

for (int j = 0; j < old\_folds; j++) {

(\*(\*(Temp\_Arr + i) + j)) = (\*(\*(Bucket + i) + j));

}

}

for (int i = 0; i < Size; i++) {

Bucket[i] = new int[folds];

}

for (int i = 0; i < Size; i++) {

for (int j = 0; j < folds; j++) {

(\*(\*(Bucket + i) + j)) = 0;

}

}

for (int i = 0; i < Size; i++) {

for (int j = 0; j < old\_folds; j++) {

if ((\*(\*(Temp\_Arr + i) + j)) != 0) {

Insert((\*(\*(Temp\_Arr + i) + j)));

}

}

}

return;

}

int loadFactor() {

return ((counter \* 100) / no\_of\_buckets);

}

public:

Buckets(int Size) {

this->Size = Size;

counter = 0;

folds = 3;

no\_of\_buckets = Size \* folds;

Bucket = new int\*[Size];

for (int i = 0; i < Size; i++) {

Bucket[i] = new int[folds];

}

for (int i = 0; i < Size; i++) {

for (int j = 0; j < folds; j++) {

(\*(\*(Bucket + i) + j)) = 0;

}

}

}

bool isFull() {

if (counter == no\_of\_buckets) { return true; }

else { return false; }

}

bool Insert(int value) {

if (loadFactor() >= 70) {

Re\_Hashing();

}

int curr\_index = 0;

curr\_index = Hash\_Function(value);

if (isFull()) {

cout << " Hash Table Becomes Full Now!" << endl;

return false;

}

if ((\*(\*(Bucket + curr\_index) + 0)) == 0) {

((\*(\*(Bucket + curr\_index)))) = value;

counter = counter + 1;

}

else {

int Folding = get\_Folding\_by\_Hash\_Function(curr\_index);

((\*(\*(Bucket + curr\_index) + Folding))) = value;

counter = counter + 1;

}

return true;

}

void Display() {

for (int i = 0; i < Size; i++) {

for (int j = 0; j < folds; j++) {

if ((\*(\*(Bucket + i) + j)) == 0) {

cout << " - ";

}

else {

cout << " " << (\*(\*(Bucket + i) + j)) << " ";

}

}

cout << endl;

}

}

};

int main()

{

int Size = 15;

Buckets Bucket(Size);

Bucket.Insert(17);

Bucket.Insert(26);

Bucket.Insert(15);

Bucket.Insert(9);

Bucket.Insert(11);

Bucket.Insert(43);

Bucket.Insert(75);

Bucket.Insert(19);

Bucket.Insert(35);

Bucket.Insert(45);

Bucket.Insert(55);

Bucket.Insert(9);

Bucket.Insert(10);

Bucket.Insert(17);

Bucket.Insert(21);

Bucket.Insert(61);

Bucket.Insert(23);

cout << " Buckets of Hash Table:" << endl << endl;

Bucket.Display();

system("pause");

return 0;

}