Table of Contentss

[Singly Linked List: 3](#_Toc170584574)

[Description: 3](#_Toc170584575)

[Code: 3](#_Toc170584576)

[Output: 6](#_Toc170584577)

[Doubly Linked List: 7](#_Toc170584578)

[Description: 7](#_Toc170584579)

[Code: 7](#_Toc170584580)

[Output 10](#_Toc170584581)

[Circular Linked List: 11](#_Toc170584582)

[Description: 11](#_Toc170584583)

[Code: 11](#_Toc170584584)

[Output: 16](#_Toc170584585)

[Stack: 17](#_Toc170584586)

[Description: 17](#_Toc170584587)

[Code: 17](#_Toc170584588)

[Header file: 17](#_Toc170584589)

[Cpp file: 17](#_Toc170584590)

[Output 19](#_Toc170584591)

[Queue: 20](#_Toc170584592)

[Description: 20](#_Toc170584593)

[Code: 20](#_Toc170584594)

[Output: 23](#_Toc170584595)

[Binary Tree: 25](#_Toc170584596)

[Description: 25](#_Toc170584597)

[Code: 25](#_Toc170584598)

[Output: 27](#_Toc170584599)

[Binary Search Tree: 28](#_Toc170584600)

[Description: 28](#_Toc170584601)

[Code: 28](#_Toc170584602)

[Header file: 28](#_Toc170584603)

[Cpp file: 29](#_Toc170584604)

[Output: 38](#_Toc170584605)

[Avl Tree: 39](#_Toc170584606)

[Description: 39](#_Toc170584607)

[Code: 39](#_Toc170584608)

[AVL.H: 39](#_Toc170584609)

[AVL.cpp: 39](#_Toc170584610)

[Output: 46](#_Toc170584611)

[Graphs: 47](#_Toc170584612)

[Representation: Adjacency Matrix, Adjacency List 47](#_Toc170584613)

[Description: 47](#_Toc170584614)

[Adjacency Matrix 47](#_Toc170584615)

[Code: 47](#_Toc170584616)

[Output: 49](#_Toc170584617)

[Adjacency List: 49](#_Toc170584618)

[Code: 49](#_Toc170584619)

[Output: 50](#_Toc170584620)

[Graph Traversals: Depth-First Search (DFS), Breadth-First Search (BFS) 51](#_Toc170584621)

[Description: 51](#_Toc170584622)

[Code: 51](#_Toc170584623)

[Output: 52](#_Toc170584624)

[Binary Heap: 53](#_Toc170584625)

[Description: 53](#_Toc170584626)

[Code: 53](#_Toc170584627)

[Output: 55](#_Toc170584628)

[Min-heap: 56](#_Toc170584629)

[Description: 56](#_Toc170584630)

[Code: 56](#_Toc170584631)

[Output: 59](#_Toc170584632)

[Max-Heap: 60](#_Toc170584633)

[Description: 60](#_Toc170584634)

[Code: 60](#_Toc170584635)

[Output: 63](#_Toc170584636)

[Open hashing / separate chaining / closed addressing 64](#_Toc170584637)

[Description: 64](#_Toc170584638)

[Code: 64](#_Toc170584639)

[Output: 67](#_Toc170584640)

[Closed hashing / open addressing: 68](#_Toc170584641)

[Linear Probing: 68](#_Toc170584642)

[Description: 68](#_Toc170584643)

[Code: 68](#_Toc170584644)

[Output: 71](#_Toc170584645)

[Quadratic Probing: 72](#_Toc170584646)

[Description: 72](#_Toc170584647)

[Code: 72](#_Toc170584648)

[Output: 74](#_Toc170584649)

# Singly Linked List:

## Description:

A singly linked list is a linear data structure where each element, known as a node, contains data and a pointer to the next node in the sequence. Visually, it appears as a series of connected nodes, with each node containing data and a directional arrow indicating the link to the next node. The list begins with a head pointer that points to the first node, and the last node points to `NULL`, indicating the end of the list. This structure allows efficient insertion and deletion at the beginning and end of the list but requires sequential traversal for accessing elements.

## Code:

#include<iostream>

using namespace std;

struct node

{

string name;

int cnic;

string disease;

string date;

int roomno;

node\* next;

};

struct node\* head = NULL;

void admitpatient(string name1, int cnic1, string disease1, string date1, int roomno1)

{

node\* newnode = new node;

if (head == NULL)

{

newnode->name = name1;

newnode->cnic = cnic1;

newnode->disease = disease1;

newnode->date = date1;

newnode->roomno = roomno1;

newnode->next = NULL;

head = newnode;

}

else

{

newnode = head;

while (newnode->next != NULL)

{

newnode = newnode->next;

}

node\* temp2 = new node;

temp2 = head;

node\* temp1 = new node;

temp1->name = name1;

temp1->cnic = cnic1;

temp1->disease = disease1;

temp1->date = date1;

temp1->roomno = roomno1;

newnode->next = temp1;

temp1->next = NULL;

if (roomno1 > 20 || roomno1 < 1)

{

temp1->roomno = NULL;

cout << "Room not exit." << endl;

}

}

}

void deleteelement(node\*\* head\_ref, int key)

{

node\* temp = \*head\_ref;

node\* prev = NULL;

if (temp != NULL && temp->cnic == key)

{

\*head\_ref = temp->next;

delete temp;

return;

}

else

{

while (temp != NULL && temp->cnic != key)

{

prev = temp;

temp = temp->next;

}

if (temp == NULL)

return;

prev->next = temp->next;

delete temp;

}

}

void display()

{

node\* temp = head;

while (temp != NULL)

{

cout << "\t" << temp->name << "\t" << temp->cnic << "\t" << temp->disease << "\t" << temp->date << "\t" << temp->roomno << endl;

temp = temp->next;

}

}

int main()

{

string name;

int cnic, cnic1;

string disease;

string date;

int roomno;

int choice;

cout << "\t\t\t List of Patient's." << endl << endl;

cout << "\t Name:" << "\t CNIC:" << "\t\t Disease:" << "\tDate: " << "\tRoom No:" << endl << endl;

admitpatient("Ali", 333028565, "Allergies", "23-10-2021", 17);

admitpatient("David", 333015678, "Conjunctivitis", "20-10-2021", 19);

admitpatient("Laiba", 333104589, "Diarrhea", "19-10-2021", 14);

admitpatient("Ibrahim", 333013578, "Headaches", "24-10-2021", 10);

admitpatient("Mehdi", 333023212, "StomachAches", "14-10-2021", 3);

display();

cout << endl;

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

cout << "1. Enter New Patient." << endl;

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

cout << "3. display" << endl;

do

{

cout << endl;

cout << "Enter choice : ";

cin >> choice;

cout << endl;

switch (choice)

{

case 1:

cout << "Enter name : ";

cin >> name;

cout << "Enter Cnic : ";

cin >> cnic;

cout << "Enter Disease : ";

cin >> disease;

cout << "Enter date : ";

cin >> date;

cout << "Enter Room No : ";

cin >> roomno;

admitpatient(name, cnic, disease, date, roomno);

break;

case 2:

cout << "Enter Cnic : ";

cin >> cnic1;

deleteelement(&head, cnic1);

break;

case 3:

display();

break;

case 4:

break;

}

} while (choice != 4);

}

## Output:

Text

Description automatically generated

Text

Description automatically generated

Text

Description automatically generated

# Doubly Linked List:

## Description:

The code demonstrates a \*\*doubly linked list\*\* implementation. Each node contains an integer data (`int d`) and pointers (`next` and `prev`) to the next and previous nodes respectively. It supports operations such as insertion at the front, after a given node, or at the end (`InsertAtFront`, `InsertAfter`, `InsertAtEnd`). Deletion methods include removing nodes from the front, before a specified node, or at the end (`DeleteAtFront`, `DeleteBefore`, `DeleteAtEnd`). The `printlist` method traverses and displays the list, showcasing bidirectional traversal capability and dynamic data management.

## Code:

#include <iostream>

using namespace std;

struct node

{

int d;

struct node\* next;

struct node\* prev;

};

class linklist

{

private:

struct node\* head;

public:

linklist()

{

node\* head = NULL;

}

void InsertAtFront(struct node\*\* head, int data)

{

struct node\* newNode = new node;

newNode->d = data;

newNode->next = (\*head);

newNode->prev = NULL;

if ((\*head) != NULL)

{

(\*head)->prev = newNode;

}

(\*head) = newNode;

}

void InsertAfter(struct node\* prevNode, int data)

{

if (prevNode == NULL)

{

cout << "This is pervious node, it should not be NULL." << endl;

return;

}

struct node\* newNode = new node;

newNode->d = data;

newNode->next = prevNode->next;

prevNode->next = newNode;

newNode->prev = prevNode;

if (newNode->next != NULL)

{

newNode->next->prev = newNode;

}

}

void InsertAtEnd(struct node\*\* head, int data)

{

struct node\* newNode = new node;

struct node\* last = \*head;

newNode->d = data;

newNode->next = NULL;

if (\*head == NULL)

{

newNode->prev = NULL;

\*head = newNode;

return;

}

while (last->next != NULL)

{

last = last->next;

}

last->next = newNode;

newNode->prev = last;

return;

}

void printlist(struct node\* node1)

{

struct node\* last;

while (node1 != NULL)

{

cout << node1->d << "<==>";

last = node1;

node1 = node1->next;

}

if (node1 == NULL)

{

cout << "NULL";

}

}

void DeleteAtFront(struct node\*\* head)

{

if (head != NULL)

{

node\* temp = \*head;

\*head = (\*head)->next;

free(temp);

if (head != NULL)

{

(\*head)->prev = NULL;

}

}

}

void DeleteBefore(struct node\*\* head, node\* del)

{

if (\*head == NULL || del == NULL)

{

return;

}

if (\*head == del)

{

\*head = del->next;

}

if (del->next != NULL)

{

del->next->prev = del->prev;

}

if (del->prev != NULL)

{

del->prev->next = del->next;

}

free(del);

return;

}

void DeleteAtEnd(struct node\* head)

{

if (head != NULL)

{

if (head->next == NULL)

{

head = NULL;

}

else

{

node\* temp = head;

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

{

temp = temp->next;

}

node\* lastNode = temp->next;

temp->next = NULL;

free(lastNode);

}

}

}

void isEmpty(struct node\*\* head)

{

\*head = (\*head)->next;

while (head != NULL)

{

(\*head)->prev = NULL;

}

cout << "List Is Empty." << endl;

}

};

int main()

{

linklist a;

struct node\* head = NULL;

cout << "Insert At Front,After & End." << endl;

a.InsertAtEnd(&head, 40);

a.InsertAtFront(&head, 20);

a.InsertAtFront(&head, 10);

a.InsertAtEnd(&head, 50);

a.InsertAfter(head->next, 30);

a.printlist(head);

cout << endl;

cout << "Delete At Front." << endl;

a.DeleteAtFront(&head);

a.printlist(head);

cout << endl;

cout << "Delete Before." << endl;

a.DeleteBefore(&head, head->next);

a.printlist(head);

cout << endl;

cout << "Delet At End." << endl;

a.DeleteAtEnd(head);

a.printlist(head);

cout << endl;

a.isEmpty(&head);

a.printlist(head);

cout << endl;

return 0;

}

## Output

Text

Description automatically generated

# Circular Linked List:

## Description:

The provided code implements a \*\*circular linked list\*\*. Each node (`struct node`) includes an integer data (`int d`) and pointers (`next`) that link each node to the next in a circular fashion, ensuring the last node points back to the first (`head`). Operations like insertion at the front (`InsertAtFront`), after a specific node (`InsertAfter`), and at the end (`InsertAtEnd`) are supported. It includes methods for deletion from the front (`DeleteAtFront`) and at specific nodes (`DeleteBefore`), along with printing the list (`printlist`) to display the circular structure and data contents efficiently.

## Code:

#include<iostream>

using namespace std;

struct node

{

int data;

node\* next;

};

struct node\* InsertAtEmpty(struct node\* head, int data1)

{

if (head != NULL)

return head;

struct node\* temp = new node;

temp->data = data1;

head = temp;

head->next = head;

return head;

}

struct node\* InsertAtFront(struct node\* head, int data1)

{

if (head == NULL)

return InsertAtEmpty(head, data1);

struct node\* temp = new node;

temp->data = data1;

temp->next = head->next;

head->next = temp;

return head;

}

struct node\* InsertAtEnd(struct node\* head, int data)

{

if (head == NULL)

return InsertAtEmpty(head, data);

struct node\* temp = new node;

temp->data = data;

temp->next = head->next;

head->next = temp;

head = temp;

return head;

}

struct node\* InsertAtPosition(struct node\* head, int data1, int aftervalue)

{

if (head == NULL)

return NULL;

struct node\* temp, \* p;

p = head->next;

do

{

if (p->data == aftervalue)

{

temp = new node;

temp->data = data1;

temp->next = p->next;

p->next = temp;

if (p == head)

head = temp;

return head;

}

p = p->next;

} while (p != head->next);

cout << "it is not present in the list." << endl;

return head;

}

void display(struct node\* head)

{

struct node\* p;

if (head == NULL) {

cout << "Circular linked List is empty." << endl;

return;

}

p = head->next;

do {

cout << p->data << "==>";

p = p->next;

} while (p != head->next);

if (p == head->next)

cout << p->data;

cout << "\n\n";

}

bool emptycheck(struct node\* head)

{

if (head == NULL || head->next == NULL)

{

cout << "the list is empty : " << endl;

return true;

}

else

{

return false;

cout << "the list is not empty " << endl;

}

}

void sort(node\*\* head1)

{

node\* newnode = new node;

node\* current = \*head1;

if (current == NULL)

{

newnode->next = newnode;

\*head1 = newnode;

}

else if (current->data >= newnode->data)

{

while (current->next != \*head1)

current = current->next;

current->next = newnode;

newnode->next = \*head1;

\*head1 = newnode;

}

else

{

while (current->next != \*head1 &&

current->next->data < newnode->data)

current = current->next;

newnode->next = current->next;

current->next = newnode;

}

}

void printlistaftersort(node\* head)

{

node\* temp;

if (head != NULL)

{

temp = head;

do {

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

temp = temp->next;

} while (temp != head);

}

}

void DeleteAll(struct node\* head)

{

if (head != NULL && head->next != NULL)

{

free(head);

cout << "list is empty now : " << endl;;

}

}

void deleteNode(node\*\* head, int data1)

{

if (\*head == NULL)

return;

if ((\*head)->data == data1 && (\*head)->next == \*head)

{

free(\*head);

\*head = NULL;

}

node\* last = \*head, \* d;

if ((\*head)->data == data1) {

while (last->next != \*head)

last = last->next;

last->next = (\*head)->next;

free(\*head);

\*head = last->next;

}

while (last->next != \*head && last->next->data != data1) {

last = last->next;

}

if (last->next->data == data1) {

d = last->next;

last->next = d->next;

cout << "The node with data " << data1 << " deleted from the list" << endl;

free(d);

cout << endl;

cout << "Circular linked list after deleting " << data1 << " is as follows :" << endl;

display(last);

}

else

cout << "The node with data " << data1 << " not found in the list" << endl;

}

int main()

{

struct node\* head = NULL;

int val1, val2, val3, val4, pos, choice;

int arr[] = { 40, 50, 30, 20, 10, 60 };

int list\_size, i;

node\* start = NULL;

node\* temp;

cout << endl;

cout << "1. Insert At front." << endl;

cout << "2. Insert At end." << endl;

cout << "3. Insert At position." << endl;

cout << "4. Insert At Empty list." << endl;

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

cout << "6. Delete Node." << endl;

cout << "7. Empty check." << endl;

cout << "8. Sorting" << endl;

cout << "9. Delete all list." << endl;

cout << "10. EXIT." << endl;

do

{

cout << endl;

cout << "Enter choice : ";

cin >> choice;

cout << endl;

switch (choice)

{

case 1:

cout << "Enter value : ";

cin >> val1;

head = InsertAtFront(head, val1);

break;

case 2:

cout << "Enter value : ";

cin >> val2;

head = InsertAtEnd(head, val2);

break;

case 3:

cout << "Enter value : ";

cin >> val3;

cout << "Enter position : ";

cin >> pos;

head = InsertAtPosition(head, val3, pos);

break;

case 4:

cout << "Enter value : ";

cin >> val4;

head = InsertAtEmpty(head, val4);

break;

case 5:

cout << "The circular linked list : " << endl;

display(head);

break;

case 6:

deleteNode(&head, 10);

break;

case 7:

emptycheck(head);

break;

case 8:

sort(&head);

cout << "sorted circular linked list : " << endl;

printlistaftersort(head);

cout << endl;

break;

case 9:

DeleteAll(head);

break;

case 10:

break;

}

} while (choice != 10);

return 0;

}

## Output:

Text

Description automatically generated Text

Description automatically generated Text

Description automatically generated

# Stack:

## Description:

A stack is implemented using a linked list structure (`struct node`). This stack operates with a Last-In, First-Out (LIFO) principle, where elements (strings representing words) are pushed onto the stack when encountered in the sentence processing. Punctuation marks are selectively removed during the stack operations (`push` and `pop`), ensuring that the modified sentence output excludes these characters. The `display` function outputs the final modified sentence by retrieving and printing words from the stack in their correct order.

## Code:

### Header file:

#ifndef \_STACK\_H

#define \_STACK\_H

#include<iostream>

using namespace std;

struct node

{

string data;

node\* next;

};

struct node\* top = NULL;

void push(string data);

int isEmpty();

void pop();

void display();

#endif

### Cpp file:

#include "stack.h"

#include<string>

void push(string data)

{

struct node\* temp = new node;

temp->data = data;

temp->next = top;

top = temp;

}

int isEmpty()

{

return top == NULL;

}

void pop()

{

struct node\* temp = top;

string data = temp->data;

if (top == NULL)

{

cout << "\nStack Underflow" << endl;

exit(1);

}

else

{

while (temp != NULL)

{

if (temp->data == "!" || temp->data == "," || temp->data == ";" || temp->data == ":" || temp->data == "?" ||

temp->data == "/" || temp->data == "-" || temp->data == "'")

{

top = top->next;

temp->next = NULL;

free(temp);

}

temp = temp->next;

}

}

}

void display()

{

node\* temp = top;

while (temp != NULL)

{

if (temp == top)

{

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

}

else

{

cout << temp->data;

}

temp = temp->next;

}

}

int main()

{

string sentence;

cout << "Please enter a string : ";

getline(cin, sentence);

string str;

char\* array1 = new char[sentence.length()];

for (int i = 0; i <= sentence.length(); i++)

{

array1[i] = sentence[i];

}

cout << endl;

for (int i = 0; i <= sentence.length(); i++)

{

if (array1[i] == '!' || array1[i] == ',' || array1[i] == ';' || array1[i] == ':' || array1[i] == '?' ||

array1[i] == '/' || array1[i] == '-' || array1[i] == '"' || array1[i] == '(' || array1[i] == ')' )

{

str = str;

}

else

{

str = str + array1[i];

}

if (array1[i] == ' ')

{

push(str);

str.erase();

}

}

push(str);

cout << endl;

cout << "Modified sentence is : ";

display();

cout << endl;

return 0;

}

## Output

Text

Description automatically generated

# Queue:

## Description:

A template-based queue implementation (`Queues`) is demonstrated. This queue follows the FIFO (First-In, First-Out) principle where elements are added (`Enqueue`) to the rear and removed (`Dequeue`) from the front. Multiple queues can be instantiated based on user input (`N` queues), each managing a fixed number of persons (`person`) as elements. The implementation ensures efficient operations using dynamic memory allocation for nodes and provides functionalities to enqueue, dequeue, display the queues visually, and handle multiple queues seamlessly within a console application.

## Code:

#include<iostream>

#include<Windows.h>

#include<stdlib.h>

using namespace std;

template<class numberofqueues>

class Queues

{

class node

{

public:

numberofqueues data;

node\* Next;

};

node\* front;

node\* rear;

public:

Queues()

{

front = NULL;

rear = NULL;

}

bool isempty()

{

if (front == NULL)

return true;

else

return false;

}

void Enqueue(int data1)

{

node\* NewNode = new node;

if (isempty())

{

NewNode->data = data1;

front = rear = NewNode;

front->Next = rear->Next = NULL;

return;

}

NewNode->data = data1;

rear->Next = NewNode;

rear = NewNode;

rear->Next = NULL;

}

numberofqueues Dequeue()

{

numberofqueues d = 0;

if (isempty())

{

cout << "Underflow\n";

return d;

}

d = front->data;

if (front == rear)

rear = NULL;

node\* temp = new node;

temp = front;

front = front->Next;

delete temp;

return d;

}

void display(int person)

{

if (isempty())

return;

node\* temp = front;

HANDLE colors = GetStdHandle(STD\_OUTPUT\_HANDLE);

SetConsoleTextAttribute(colors, 10);

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

{

cout << "\t \_\_\_";

}

cout << endl;

while (temp != NULL)

{

cout << "\t | " << temp->data << " |";

temp = temp->Next;

}

cout << endl;

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

{

cout << "\t |\_\_\_|";

}

SetConsoleTextAttribute(colors, 7);

cout << endl;

}

~Queues()

{

node\* temp = front;

node\* temp2 = NULL;

while (temp2 != NULL)

{

temp = temp2;

temp2 = temp2->Next;

delete temp;

}

delete temp2;

}

};

int main()

{

Queues<int>\* newqueue;

int N;

int person, count = 0;

int temp1, temp2, choice;

cout << "Enter no of Queues you want to : ";

cin >> N;

newqueue = new Queues<int>[N];

cout << "Enter no of persons in each Queue : ";

cin >> person;

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

{

for (int j = 1; j <= person; j++)

{

newqueue[i].Enqueue(j);

}

}

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

{

cout << "\nQueue " << i + 1 << endl;

newqueue[i].display(person);

}

do

{

cout << endl;

cout << "1. Dequeue." << endl;

cout << "2. Display Queues." << endl;

cout << "3. Exit." << endl;

cout << " Enter : ";

cin >> choice;

switch (choice)

{

case 1:

system("cls");

for (int i = count; i < N; i++)

{

if (!newqueue[i].isempty())

{

temp1 = newqueue[i].Dequeue();

if (i < N - 1)

newqueue[i + 1].Enqueue(temp1);

}

else

count++;

}

if (newqueue[N - 1].isempty())

{

cout << "\nAll Queues are empty\n\n";

break;

}

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

{

cout << "Queue " << i + 1 << endl;

newqueue[i].display(person);

}

break;

case 2:

system("cls");

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

{

cout << endl << "Queue " << i + 1;

newqueue[i].display(person);

}

break;

case 3:

return 0;

break;

}

} while (choice != 3);

}

## Output:

A screenshot of a computer screen

Description automatically generated with medium confidence A screenshot of a computer

Description automatically generated with medium confidence

A screenshot of a computer

Description automatically generated with medium confidence 

Text

Description automatically generated

# Binary Tree:

## Description:

A binary tree is a hierarchical data structure composed of nodes, where each node has at most two children: left and right. Nodes store data and are connected according to parent-child relationships. Binary trees are versatile for organizing hierarchical data and support various operations like insertion, deletion, and traversal (inorder, preorder, postorder), making them fundamental in computer science for efficient data organization and retrieval.

## Code:

#include<iostream>

using namespace std;

struct node {

int data;

node\* left;

node\* right;

};

node\* root = NULL;

void insert(int data1) {

node\* newnode = new node;

newnode->data = data1;

newnode->left = NULL;

newnode->right = NULL;

if (root == NULL) {

root = newnode;

} else {

// Inserting as a normal binary tree without considering ordering

node\* temp = root;

while (temp->left != NULL && temp->right != NULL) {

if (temp->left == NULL) {

temp->left = newnode;

break;

} else if (temp->right == NULL) {

temp->right = newnode;

break;

} else {

temp = temp->left;

}

}

}

}

void inorder(node\* temp) {

if (temp != NULL) {

inorder(temp->left);

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

inorder(temp->right);

}

}

int main() {

int choice, val1;

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

cout << "2. Inorder traversal." << endl;

cout << "3. Exit." << endl;

do {

cout << endl;

cout << "Enter choice : ";

cin >> choice;

switch (choice) {

case 1:

cout << "Enter value to insert : ";

cin >> val1;

insert(val1);

break;

case 2:

cout << "Inorder traversal is : ";

inorder(root);

cout << endl;

break;

case 3:

cout << endl << "EXIT" << endl;

break;

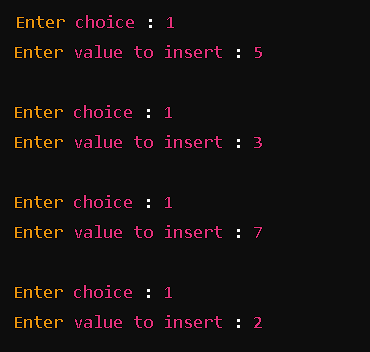
}

} while (choice != 3);

return 0;

}

## Output:



A number on a black background

Description automatically generated

# Binary Search Tree:

## Description:

In the context of the provided code, a Binary Search Tree (BST) is a data structure where each node has at most two children, organized such that for any given node:

* Nodes in the left subtree have values less than the node's value.
* Nodes in the right subtree have values greater than the node's value.

BST operations like insertion, deletion, and searching are efficient due to its ordered structure, facilitating fast access and manipulation of data.

## Code:

### Header file:

#ifndef \_BST\_H

#define \_BST\_H

#include<iostream>

#include<windows.h>

using namespace std;

template<class B>

class BST

{

private:

struct node

{

B data;

node\* left = NULL;

node\* right = NULL;

};

node\* root;

void deletenode(node\*& temp, B val);

void deletetree(node\* temp);

int search(struct node\* temp, B key);

void inorder(struct node\* temp);

void preorder(struct node\* temp);

void postorder(struct node\* temp);

int height(node\* temp);

int totalnumberofnodes(node\* temp);

int nodeslevel(node\* temp, B data1, B level);

int totalnumberofleafnodes(node\* temp);

public:

int numberofnodes = 1;

BST();

void insert(B data1);

void makedeletion(node\*& temp);

void deletenode(B data1);

void deletetree();

void search(B key1);

int largestnumber();

int Smallestnumber();

void inorder();

void preorder();

void postorder();

int height();

int totalnumberofnodes();

int totalnumberofleafnodes();

int nodeslevel(B data1, B level);

void average();

};

#endif

### Cpp file:

#include<iostream>

#include "BST.h"

#include<string>

using namespace std;

template<class B>

BST<B>::BST()

{

root = NULL;

}

template<class B>

void BST<B>::insert(B data1)

{

node\* newnode = new node;

newnode->data = data1;

newnode->left = NULL;

newnode->right = NULL;

node\* temp1 = root;

if (root == NULL)

{

root = newnode;

}

else

{

while (temp1 != NULL)

{

if (temp1->data < data1)

{

if (temp1->right == NULL)

{

temp1->right = newnode;

break;

}

temp1 = temp1->right;

}

else if (temp1->data > data1)

{

if (temp1->left == NULL)

{

temp1->left = newnode;

break;

}

temp1 = temp1->left;

}

}

}

}

template<class B>

void BST<B>::makedeletion(node\*& temp)

{

node\* temp1;

if (temp->right == NULL)

{

temp1 = temp;

temp = temp->left;

delete temp1;

}

else if (temp->left == NULL)

{

temp1 = temp;

temp = temp->right;

delete temp1;

}

else

{

temp1 = temp->right;

while (temp1->left)

{

temp1 = temp1->left;

}

temp1->left = temp->left;

temp1 = temp;

temp = temp->right;

delete temp1;

}

}

template<class B>

void BST<B>::deletenode(B data1)

{

deletenode(this->root, data1);

}

template<class B>

void BST<B>::deletenode(node\*& temp, B val)

{

if (temp == NULL)

{

cout << "Value not found" << endl;

}

else if (temp->data < val)

{

deletenode(temp->right, val);

}

else if (temp->data > val)

{

deletenode(temp->left, val);

}

else

{

makedeletion(temp);

}

}

template<class B>

void BST<B>::deletetree()

{

deletetree(this->root);

root = NULL;

}

template<class B>

void BST<B>::deletetree(node\* temp)

{

if (temp == NULL)

{

return;

}

deletetree(temp->left);

deletetree(temp->right);

delete temp;

}

template<class B>

void BST<B>::search(B key1)

{

search(this->root, key1);

}

template<class B>

int BST<B>::search(struct node\* temp, B key)

{

if (temp == NULL)

{

cout << "Value not found. " << endl;

}

else if (temp->data == key)

{

cout << "Value is found." << endl;

}

else if (temp->data < key)

{

return search(temp->right, key);

}

else if (temp->data > key)

{

return search(temp->left, key);

}

return 0;

}

template<class B>

int BST<B>::largestnumber()

{

node\* temp = root;

node\* largest = NULL;

while (temp != NULL)

{

if (temp > temp->right)

{

largest = temp;

}

else

{

largest = temp->right;

}

temp = temp->right;

}

return largest->data;

}

template<class B>

int BST<B>::Smallestnumber()

{

node\* temp = root;

node\* smallest = NULL;

while (temp != NULL)

{

if (temp > temp->left)

{

smallest = temp;

}

else

{

smallest = temp->left;

}

temp = temp->left;

}

return smallest->data;

}

template<class B>

void BST<B>::inorder()

{

inorder(this->root);

cout << endl;

}

template<class B>

void BST<B>::inorder(struct node\* temp)

{

if (temp != NULL)

{

inorder(temp->left);

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

inorder(temp->right);

}

}

template<class B>

void BST<B>::preorder()

{

preorder(this->root);

cout << endl;

}

template<class B>

void BST<B>::preorder(struct node\* temp)

{

if (temp != NULL)

{

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

preorder(temp->left);

preorder(temp->right);

}

}

template<class B>

void BST<B>::postorder()

{

postorder(this->root);

cout << endl;

}

template<class B>

void BST<B>::postorder(struct node\* temp)

{

if (temp != NULL)

{

postorder(temp->left);

postorder(temp->right);

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

}

}

template<class B>

int BST<B>::height()

{

return height(this->root);

cout << endl;

}

template<class B>

int BST<B>::height(node\* temp)

{

if (temp == NULL)

{

return -1;

}

else

{

int leftdepth = height(temp->left);

int rightdepth = height(temp->right);

if (leftdepth > rightdepth)

{

return leftdepth + 1;

}

else

{

return rightdepth + 1;

}

}

}

template<class B>

int BST<B>::totalnumberofnodes()

{

return totalnumberofnodes(this->root);

cout << endl;

}

template<class B>

int BST<B>::totalnumberofnodes(node\* temp)

{

if (temp == NULL)

{

return 0;

}

if (temp->left != NULL)

{

numberofnodes = numberofnodes + 1;

totalnumberofnodes(temp->left);

}

if (temp->right != NULL)

{

numberofnodes = numberofnodes + 1;

totalnumberofnodes(temp->right);

}

return numberofnodes;

}

template<class B>

int BST<B>::totalnumberofleafnodes()

{

return totalnumberofleafnodes(this->root);

cout << endl;

}

template<class B>

int BST<B>::totalnumberofleafnodes(node \* temp)

{

if (temp == NULL)

return 0;

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

return 1;

else

return totalnumberofleafnodes(temp->left) + totalnumberofleafnodes(temp->right);

}

template<class B>

int BST<B>::nodeslevel(B data1, B level)

{

return nodeslevel(this->root, data1, level);

cout << endl;

}

template<class B>

int BST<B>::nodeslevel(node\* temp, B data1, B level)

{

if (temp == NULL)

{

return 0;

}

else

{

if (temp->data == data1)

{

return level;

}

else

{

int level1 = nodeslevel(temp->left, data1, level + 1);

if (level1 != 0)

{

return level1;

}

level1 = nodeslevel(temp->right, data1, level + 1);

return level1;

}

}

}

int main()

{

HANDLE colors = GetStdHandle(STD\_OUTPUT\_HANDLE);

SetConsoleTextAttribute(colors, 7);

int choice;

double value1, key, level;

double average;

BST<int> bst;

int arr[10] = { 65, 55, 22, 54, 61, 19, 90, 10, 78, 52 };

cout << "1. Insert an sorted array." << endl;

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

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

cout << "4. Find maximum node." << endl;

cout << "5. Find minimum node." << endl;

cout << "6. Inorder traversal." << endl;

cout << "7. Pre-order traversal." << endl;

cout << "8. Post-order traversal." << endl;

cout << "9. Tree height." << endl;

cout << "10. Number of nodes." << endl;

cout << "11. Number of leaf nodes." << endl;

cout << "12. Level of node." << endl;

cout << "13. TASK 1 : Convert bst to bst of minimal height." << endl;

do

{

cout << "Enter choice : ";

cin >> choice;

switch (choice)

{

case 1:

for (int i = 0; i < 10; i++)

{

bst.insert(arr[i]);

}

break;

case 2:

cout << "Enter value to delete : ";

cin >> value1;

bst.deletenode(value1);

break;

case 3:

cout << "Enter key to search : ";

cin >> key;

bst.search(key);

break;

case 4:

cout << "largest : " << bst.largestnumber() << endl;

break;

case 5:

cout << "Smallest : " << bst.Smallestnumber() << endl;

break;

case 6:

cout << "In-order traversal : ";

bst.inorder();

break;

case 7:

cout << "Pre-order traversal : ";

bst.preorder();

break;

case 8:

cout << "Post-order traversal : ";

bst.postorder();

break;

case 9:

SetConsoleTextAttribute(colors, 12);

cout << " Height of tree is : " << bst.height() << endl;

SetConsoleTextAttribute(colors, 7);

break;

case 10:

cout << "Total number of nodes is : " << bst.totalnumberofnodes() << endl;

break;

case 11:

cout << "Total number of leaf nodes : " << bst.totalnumberofleafnodes() << endl;

break;

case 12:

cout << "Enter a value to find it's level : ";

cin >> level;

cout << "Level is : " << bst.nodeslevel(level, 1) << endl;

break;

case 13:

int average1 = (bst.largestnumber() + bst.Smallestnumber()) / 2;

int max = average1 + 5;

int min = average1 - 5;

int minimum = 0, j = -1;

int arr2[10];

for (int i = 0; i < 10; i++)

{

if (arr[i] >= min && arr[i] <= max)

{

j++;

arr2[j] = arr[i];

}

}

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

{

if (i == 0)

{

minimum = arr2[i];

}

else if (arr2[i] < minimum)

{

minimum = arr2[i];

}

}

bst.deletetree();

bst.insert(minimum);

for (int i = 0; i < 10; i++)

{

if (arr[i] != minimum)

{

bst.insert(arr[i]);

}

}

SetConsoleTextAttribute(colors, 12);

cout << "\t TREE IS UPDATED. KNOW THE ROOT IS [" << minimum << "]" << endl;

SetConsoleTextAttribute(colors, 7);

break;

}

cout << endl;

} while (choice != 14);

}

## Output:

Text

Description automatically generated Text

Description automatically generated

Text

Description automatically generated Text

Description automatically generated

# Avl Tree:

## Description:

## Code:

### AVL.H:

#include<iostream>

#include<windows.h>

using namespace std;

#pragma once

template<class T>

class node

{

public:

T data;

T balancefactor;

node<T>\* left;

node<T>\* right;

node();

};

template<class T>

class BST

{

node<T>\* root;

public:

int numberofnodes = 1;

BST();

node<T>\* newNode(T key);

node<T>\* insert(T data1);

node<T>\* insert1(node<T>\* temp);

void makedeletion(node<T>\*& temp);

void deletenode(T data1);

void deletenode(node<T>\*& temp, T val);

void inorder();

void inorder(struct node<T>\* temp);

void preorder();

void preorder(node<T>\* temp);

void postorder();

void postorder(node<T>\* temp);

int height();

int height(node<T>\* temp);

node<T>\* llrotation(node<T>\* temp);

node<T>\* lrrotation(node<T>\* temp);

node<T>\* rrrotation(node<T>\* temp);

node<T>\* rlrotation(node<T>\* temp);

int balancefactor(node<T>\* temp);

};

### AVL.cpp:

#include "AVL.h"

template<class T>

node<T>::node()

{

right = NULL;

left = NULL;

}

template<class T>

BST<T>::BST()

{

root = NULL;

}

template<class T>

node<T>\* BST<T>::insert(T data1)

{

node<T>\* newnode = new node<T>;

newnode->data = data1;

node<T>\* temp1 = root;

if (temp1 == NULL)

{

temp1 = newnode;

root = temp1;

root->balancefactor = 1;

return temp1;

}

else

{

while (temp1 != NULL)

{

if (temp1->data < data1)

{

if (temp1->right == NULL)

{

temp1->right = newnode;

break;

}

temp1 = temp1->right;

}

else if (temp1->data > data1)

{

if (temp1->left == NULL)

{

temp1->left = newnode;

break;

}

temp1 = temp1->left;

}

}

temp1->balancefactor = height(temp1);

if (balancefactor(temp1) == -2 && balancefactor(temp1->right) == 1)

{

temp1 = lrotation(temp1);

}

else if (balancefactor(temp1) == 2 && balancefactor(temp1->left) == -1)

{

temp1 = lrrotation(temp1);

}

return temp1;

}

}

template<class T>

void BST<T>::makedeletion(node<T>\*& temp)

{

node<T>\* temp1;

if (temp->right == NULL)

{

temp1 = temp;

temp = temp->left;

delete temp1;

}

else if (temp->left == NULL)

{

temp1 = temp;

temp = temp->right;

delete temp1;

}

else

{

temp1 = temp->right;

while (temp1->left)

{

temp1 = temp1->left;

}

temp1->left = temp->left;

temp1 = temp;

temp = temp->right;

delete temp1;

}

if (balancefactor(temp) == 2 && balancefactor(temp->left) == 1)

{

temp = llrotation(temp);

}

else if (balancefactor(temp) == 2 && balancefactor(temp->left) == -1)

{

temp = lrrotation(temp);

}

else if (balancefactor(temp) == 2 && balancefactor(temp->left) == 0)

{

temp = llrotation(temp);

}

else if (balancefactor(temp) == -2 && balancefactor(temp->right) == -1)

{

temp = rrrotation(temp);

}

else if (balancefactor(temp) == -2 && balancefactor(temp->right) == 1)

{

temp = rlrotation(temp);

}

else if (balancefactor(temp) == -2 && balancefactor(temp->right) == 0)

{

temp = llrotation(temp);

}

}

template<class T>

void BST<T>::deletenode(T data1)

{

deletenode(this->root, data1);

}

template<class T>

void BST<T>::deletenode(node<T>\*& temp, T val)

{

if (temp == NULL)

{

cout << "Value not found" << endl;

}

else if (temp->data < val)

{

deletenode(temp->right, val);

}

else if (temp->data > val)

{

deletenode(temp->left, val);

}

else

{

makedeletion(temp);

}

}

template<class T>

void BST<T>::inorder()

{

inorder(this->root);

cout << endl;

}

template<class T>

void BST<T>::inorder(struct node<T>\* temp)

{

if (temp != NULL)

{

inorder(temp->left);

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

inorder(temp->right);

}

}

template<class T>

void BST<T>::preorder()

{

preorder(this->root);

cout << endl;

}

template<class T>

void BST<T>::preorder(node<T>\* temp)

{

if (temp != NULL)

{

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

preorder(temp->left);

preorder(temp->right);

}

}

template<class T>

void BST<T>::postorder()

{

postorder(this->root);

cout << endl;

}

template<class T>

void BST<T>::postorder(node<T>\* temp)

{

if (temp != NULL)

{

postorder(temp->left);

postorder(temp->right);

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

}

}

template<class T>

int BST<T>::height()

{

return height(this->root);

cout << endl;

}

template<class T>

int BST<T>::height(node<T>\* temp)

{

if (temp == NULL)

{

return -1;

}

else

{

int leftdepth = height(temp->left);

int rightdepth = height(temp->right);

if (leftdepth > rightdepth)

{

return leftdepth + 1;

}

else

{

return rightdepth + 1;

}

}

}

template<class T>

int BST<T>::balancefactor(node<T>\* temp)

{

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

{

return temp->left->balancefactor - temp->right->balancefactor;

}

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

{

return temp->left->balancefactor;

}

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

{

return -temp->right->balancefactor;

}

}

template<class T>

node<T>\* BST<T>::llrotation(node<T>\* temp)

{

node<T>\* temp1;

node<T>\* temp2;

temp1 = temp;

temp2 = temp1->left;

temp1->left = temp2->right;

temp2->right = temp1;

return temp2;

}

template<class T>

node<T>\* BST<T>::rrrotation(node<T>\* temp)

{

node<T>\* temp1;

node<T>\* temp2;

temp1 = temp;

temp2 = temp1->right;

temp->right = temp2->left;

temp2->left = temp1;

return temp2;

}

template<class T>

node<T>\* BST<T>::rlrotation(node<T>\* temp)

{

node<T>\* temp1;

node<T>\* temp2;

node<T>\* temp3;

temp1 = temp;

temp2 = temp1->right;

temp3 = temp1->right->left;

temp1->right = temp3->left;

temp2->left = temp3->right;

temp3->left = temp1;

temp3->right = temp2;

return temp3;

}

template<class T>

node<T>\* BST<T>::lrrotation(node<T>\* temp)

{

node<T>\* temp1;

node<T>\* temp2;

node<T>\* temp3;

temp1 = temp;

temp2 = temp1->left;

temp3 = temp1->left->right;

temp1->left = temp3;

temp2->right = temp3->left;

temp3->right = temp1;

temp3->left = temp2;

return temp3;

}

int main()

{

HANDLE colors = GetStdHandle(STD\_OUTPUT\_HANDLE);

SetConsoleTextAttribute(colors, 7);

int choice;

double value1, key, level, sum;

double average;

BST<int> bst;

int arr[10] = { 65, 55, 22, 54, 61, 19, 90, 10, 78, 52 };

cout << "1. Insert an sorted array." << endl;

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

cout << "3. Inorder traversal." << endl;

cout << "4. Pre-order traversal." << endl;

cout << "5. Post-order traversal." << endl;

cout << "6. Tree height." << endl;

do

{

cout << "Enter choice : ";

cin >> choice;

switch (choice)

{

case 1:

for (int i = 0; i < 10; i++)

{

bst.insert(arr[i]);

}

break;

case 2:

cout << "Enter value to delete : ";

cin >> value1;

bst.deletenode(value1);

break;

case 3:

cout << "In-order traversal : ";

bst.inorder();

break;

case 4:

cout << "Pre-order traversal : ";

bst.preorder();

break;

case 5:

cout << "Post-order traversal : ";

bst.postorder();

break;

case 6:

SetConsoleTextAttribute(colors, 12);

cout << " Height of tree is : " << bst.height() << endl;

SetConsoleTextAttribute(colors, 7);

break;

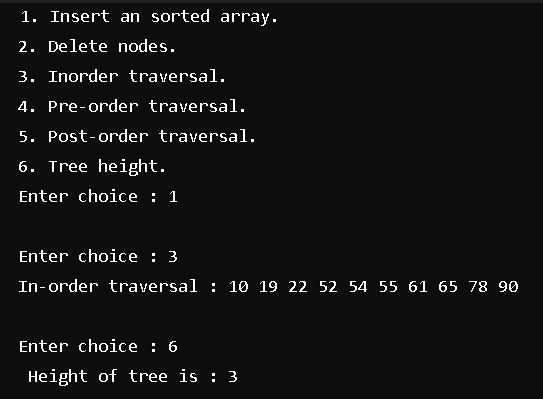
}

cout << endl;

} while (choice != 8);

}

## Output:

 A black screen with white text

Description automatically generated

# Graphs:

## Representation: Adjacency Matrix, Adjacency List

### Description:

Graphs can be represented in two primary ways:

1. **Adjacency Matrix**:
   * Uses a 2D array where matrix[i][j] indicates if there's an edge between vertex i and vertex j.
   * Efficient for dense graphs but can be memory-intensive for sparse graphs due to its V x V size.
2. **Adjacency List**:
   * Uses an array of lists (or arrays) where each list list[i] stores vertices adjacent to vertex i.
   * Efficient for sparse graphs as it saves space and allows quick access to neighbors.

### Adjacency Matrix

#### Code:

#include<iostream>

using namespace std;

void CreatMatrix(int\*\* arr, int vertex)

{

int row, col;

cout << "Enter Row No : ";

cin >> row;

cout << "Enter Col No : ";

cin >> col;

cout << endl;

if (row <= vertex && col <= vertex)

{

arr[row - 1][col - 1] = 1;

}

else

{

cout << "This Index is Invalid." << endl;

}

}

void DisplayMatrix(int\*\* arr, int vertex)

{

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

{

cout << "\t\t";

for (int j = 0; j < vertex; j++)

{

cout << " " << arr[i][j] << " ";

}

cout << endl;

}

cout << endl;

}

int main()

{

int vertex; int choice;

cout << "\t |Graph : Matrix and list representation|" << endl << endl;

cout << " Enter no. of vertices : ";

cin >> vertex;

int\*\* arr = new int\* [vertex];

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

{

arr[i] = new int[vertex];

}

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

{

for (int j = 0; j < vertex; j++)

{

arr[i][j] = 0;

}

}

cout << endl;

do

{

cout << "1. Make Adjacency Matrix." << endl;

cout << "2. Display Matrix." << endl;

cout << "3. Exit." << endl;

cout << "Enter choice : ";

cin >> choice;

cout << endl;

switch (choice)

{

case 1:

CreatMatrix(arr, vertex);

break;

case 2:

DisplayMatrix(arr, vertex);

break;

case 3:

cout << "\t Exit" << endl;

}

} while (choice != 3);

system("pause");

return 0;

}

#### Output:

Text

Description automatically generatedText

Description automatically generatedText

Description automatically generated

### Adjacency List:

#### Code:

#include<iostream>

using namespace std;

void CreatList(int\*\* arr1, int arr[][2])

{

int val1, val2;

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

{

for (int j = 0; j < 2; j++)

{

val1 = arr[i][0];

val2 = arr[i][1];

arr1[val1 - 1][val2 - 1] = 1;

arr1[val2 - 1][val1 - 1] = 1;

}

}

}

void DisplayList(int\*\* arr1, int vertex, int edges)

{

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

{

cout << "\t (" << i + 1 << ")";

for (int j = 0; j < edges; j++)

{

if (arr1[i][j] == 1)

{

cout << "->" << j + 1;

}

}

cout << endl;

}

}

int main()

{

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

int vertex, edges, choice;

cout << "\t |Graph : Matrix and list representation|" << endl << endl;

cout << " Enter no. of vertices : ";

cin >> vertex;

cout << " Enter no. of edges : ";

cin >> edges;

int\*\* arr1 = new int\* [vertex];

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

{

arr1[i] = new int[edges];

}

cout << endl;

do

{

cout << "1. Make Adjacency List." << endl;

cout << "2. Display List." << endl;

cout << "3. Exit." << endl;

cout << "Enter choice : ";

cin >> choice;

cout << endl;

switch (choice)

{

case 1:

CreatList(arr1, arr);

break;

case 2:

DisplayList(arr1, vertex, edges);

cout << endl;

break;

Text

Description automatically generated case 3:

cout << "\t Exit" << endl;

break;

}

} while (choice != 3);

system("pause");

return 0;

}

#### Output:

## Graph Traversals: Depth-First Search (DFS), Breadth-First Search (BFS)

### Description:

### Code:

#include<iostream>

using namespace std;

void DepthFirst(char\* arr, char node, int index)

{

while (arr[index] != node)

{

cout << arr[index] << " ";

index = index \* 2;

}

cout << arr[index];

}

void BreadthFirst(char \*arr, char node, int size)

{

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

{

if (arr[i] != NULL)

{

cout << arr[i] << " ";

if (arr[i] == node)

{

break;

}

}

}

}

int main()

{

int choice, index, Nnodes;

char arr[16][2] = { {0,'A'},{'B','C'},{'D','E'},{0, 'F'},{'G','H'}

,{0,'I'},{0,0},{'J',0},{'K','L'},{0,0},{0,0},{'M',0},{0,0}

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

cout << endl;

char node;

cout << "Enter node to find it's path : ";

cin >> node;

cout << endl;

do

{

cout << endl;

cout << "1. Find Path using DfS." << endl;

cout << "2. Find Path using BFS." << endl;

cout << "3. Exit." << endl;

cout << "Enter choice : ";

cin >> choice;

cout << endl;

switch (choice)

{

case 1:

cout << "Enter start index : ";

cin >> index;

cout << endl;

cout << "DFS : ";

DepthFirst(\*arr, node, index);

cout << endl;

break;

case 2:

cout << "Enter Number of nodes : ";

cin >> Nnodes;

cout << endl;

cout << "BFS : ";

BreadthFirst(\*arr, node, Nnodes);

cout << endl;

break;

case 3:

cout << "\t Exit" << endl;

break;

}

} while (choice != 3);

system("pause");

}

### Output:

Text

Description automatically generatedText

Description automatically generated

# Binary Heap:

## Description:

A binary heap is a complete binary tree where every parent node has a value less than or equal to (min-heap) or greater than or equal to (max-heap) the values of its children. It is commonly implemented using an array, allowing efficient insertion, deletion, and access operations, making it ideal for priority queue applications and heap sort algorithms due to its logarithmic time complexity for essential operations.

## Code:

#include <iostream>

using namespace std;

class MinHeap {

private:

int\* h; // Pointer to array of elements in heap

int capacity; // Maximum possible size of heap

int size; // Current number of elements in heap

public:

MinHeap(int cap) {

capacity = cap;

size = 0;

h = new int[capacity];

}

int parent(int i) { return (i - 1) / 2; }

int leftChild(int i) { return 2 \* i + 1; }

int rightChild(int i) { return 2 \* i + 2; }

void swap(int\* x, int\* y) {

int temp = \*x;

\*x = \*y;

\*y = temp;

}

void insert(int val) {

if (size == capacity) {

cout << "Heap is full." << endl;

return;

}

size++;

int i = size - 1;

h[i] = val;

// Maintain heap property

while (i != 0 && h[parent(i)] > h[i]) {

swap(&h[parent(i)], &h[i]);

i = parent(i);

}

}

void heapify(int i) {

int l = leftChild(i);

int r = rightChild(i);

int smallest = i;

if (l < size && h[l] < h[smallest])

smallest = l;

if (r < size && h[r] < h[smallest])

smallest = r;

if (smallest != i) {

swap(&h[i], &h[smallest]);

heapify(smallest);

}}

int extractMin() {

if (size <= 0)

return INT\_MAX;

if (size == 1) {

size--;

return h[0];

}

int root = h[0];

h[0] = h[size - 1];

size--;

heapify(0);

return root;

}

void display() {

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

cout << h[i] << " ";

cout << endl;

}};

int main() {

MinHeap heap(10);

heap.insert(3);

heap.insert(2);

heap.insert(1);

heap.insert(15);

heap.insert(5);

heap.insert(4);

cout << "Heap: ";

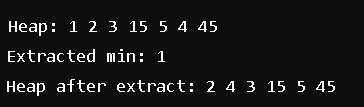
heap.display();

cout << "Extracted min: " << heap.extractMin() << endl;

cout << "Heap after extract: ";

heap.display();

return 0;

}

### Output:

# Min-heap:

## Description:

A min-heap is a specialized binary tree structure represented in an array, where the parent nodes have values smaller than or equal to their children. It supports efficient operations like insertion, extraction of the minimum element, and maintaining the heap property through heapify operations. This structure is pivotal in priority queues and algorithms like heap sort for maintaining order and quick access to the smallest element.

## Code:

#include<iostream>

using namespace std;

class heap

{

private:

int\* h;

int size;

int capacity;

public:

heap(int key)

{

size = 0;

capacity = key;

h = new int[capacity];

}

int parent(int i)

{

return (i - 1) / 2;

}

int leftchild(int i)

{

return 2 \* i + 1;

}

int rightchild(int i)

{

return 2 \* i + 2;

}

void swap(int\* x, int\* y)

{

int temp = \*x;

\*x = \*y;

\*y = temp;

}

void insert(int val)

{

if (size == capacity)

{

cout << "Heap is full." << endl;

return;

}

else

{

size++;

int i = size - 1;

h[i] = val;

while (i != 0 && h[parent(i)] > h[i])

{

swap(&h[parent(i)], &h[i]);

i = parent(i);

}

}

}

void min\_heap(int i)

{

int l = leftchild(i);

int r = rightchild(i);

int smallest = i;

if (l < size && h[l] < h[i])

{

smallest = l;

}

if (l < size && h[r] < h[smallest])

{

smallest = r;

}

if (i != smallest)

{

swap(&h[i], &h[smallest]);

min\_heap(smallest);

}

}

int extract\_min()

{

if (size <= 0)

{

return 0;

}

else if (size == 1)

{

size--;

return h[0];

}

else

{

int parent = h[0];

h[0] = h[size - 1];

size--;

min\_heap(0);

return parent;

}

}

void getmin()

{

cout << h[0] << endl;

}

void display()

{

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

{

cout << h[i] << " ";

}

cout << endl;

}

};

int main()

{

int choice, val;

int size;

cout << endl;

cout << "Enter the size if heap : ";

cin >> size;

heap H(size);

cout << endl;

cout << "1. Insert values into heap." << endl;

cout << "2. Display root." << endl;

cout << "3. Display Tree." << endl;

cout << "4. Extract minimum element." << endl;

do

{

cout << endl;

cout << "Enter choice : ";

cin >> choice;

switch (choice)

{

case 1:

cout << "Enter value to insert : ";

cin >> val;

H.insert(val);

break;

case 2:

cout << "Root : ";

H.getmin();

break;

case 3:

cout << "Tree : ";

H.display();

break;

case 4:

cout << "Delete value : " << H.extract\_min() << endl;

break;

}

} while (choice != 5);

system("pause");

return 0;

}

## Output:

Text

Description automatically generated Text

Description automatically generated Text

Description automatically generated

# Max-Heap:

## Description:

A max-heap is a complete binary tree where the value of each parent node is greater than or equal to the values of its children. This property ensures that the maximum element is always at the root. Max-heaps are typically implemented using arrays, supporting efficient operations such as insertion, deletion of the maximum element, and heapifying to maintain the heap property, all of which operate in logarithmic time complexity relative to the number of elements.

## Code:

#include<iostream>

using namespace std;

class heap

{

private:

int\* h;

int\* arr;

int size;

int capacity;

int k;

public:

heap(int key, int val)

{

size = 0;

capacity = key;

k = val;

h = new int[capacity];

arr = new int[k];

}

int parent(int i)

{

return (i - 1) / 2;

}

int leftchild(int i)

{

return 2 \* i + 1;

}

int rightchild(int i)

{

return 2 \* i + 2;

}

void swap(int\* x, int\* y)

{

int temp = \*x;

\*x = \*y;

\*y = temp;

}

void insert(int val)

{

if (size == capacity)

{

cout << "Heap is full." << endl;

return;

}

else

{

size++;

int i = size - 1;

h[i] = val;

while (i != 0 && h[parent(i)] < h[i])

{

swap(&h[parent(i)], &h[i]);

i = parent(i);

}

}

}

void max\_heap(int i)

{

int l = leftchild(i);

int r = rightchild(i);

int smallest = i;

if (l < size && h[l] > h[i])

{

smallest = l;

}

if (l < size && h[r] > h[smallest])

{

smallest = r;

}

if (i != smallest)

{

swap(&h[i], &h[smallest]);

max\_heap(smallest);

}

}

int Extraxt\_max()

{

int del;

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

{

if (size <= 0)

{

return 0;

}

else if (size == 1)

{

size--;

del = h[0];

}

else

{

del = h[0];

h[0] = h[size - 1];

size--;

max\_heap(0);

}

arr[i] = del;

}

}

void display\_tree()

{

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

{

cout << h[i] << " ";

}

cout << endl;

}

void display\_largest()

{

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

{

cout << arr[i] << " ";

}

cout << endl;

}

};

int main()

{

int choice, val;

int size, k;

cout << endl;

cout << "Enter the size if heap : ";

cin >> size;

cout << endl << "Number of largest element : ";

cin >> k;

heap H(size, k);

cout << endl;

cout << "1. Insert values into heap." << endl;

cout << "2. Display Tree." << endl;

cout << "3. Extract maximum element." << endl;

cout << "4. Display Largest element." << endl;

do

{

cout << endl;

cout << "Enter choice : ";

cin >> choice;

switch (choice)

{

case 1:

cout << "Enter value to insert : ";

cin >> val;

H.insert(val);

break;

case 2:

cout << "Tree : ";

H.display\_tree();

break;

case 3:

H.Extraxt\_max();

break;

case 4:

cout << "Largest element : ";

H.display\_largest();

break;

}

} while (choice != 5);

system("pause");

return 0;

}

## Output:

Text

Description automatically generatedText

Description automatically generatedText

Description automatically generated

# Open hashing / separate chaining / closed addressing

## Description:

Open hashing, also known as separate chaining, handles collisions in hash tables by storing multiple elements in the same slot using linked lists. Each slot in the hash table (array) points to a linked list where elements with the same hash key are stored sequentially. This approach allows for efficient handling of collisions and supports dynamic allocation of memory for hash table entries, making it suitable for scenarios where the number of elements is unpredictable or can vary widely.

## Code:

#include<iostream>

using namespace std;

struct node

{

public:

int phone;

string name;

node\* next;

};

class linkedList

{

public:

node\* head;

linkedList();

void InsertList(int number, string name);

void deleteList(string name);

void SearchList(string name);

};

class OpenHashing

{

public:

linkedList arr[10];

int hashvalue(string name);

void insert(string name, int number);

void search(string name);

void deleteV(string name);

};

linkedList::linkedList()

{

head = NULL;

}

void linkedList::InsertList(int number, string name)

{

node\* newnode = new node();

newnode->name = name;

newnode->phone = number;

if (head == NULL)

{

head = newnode;

}

else

{

node\* temp = head;

while (temp->next)

{

temp = temp->next;

}

temp->next = newnode;

}

}

void linkedList::deleteList(string name)

{

node\* temp = head;

if (head == NULL)

{

return;

}

else if (head->name == name)

{

head = head->next;

return;

}

else

{

while (temp->next)

{

if (temp->next->name == name)

{

temp->next = temp->next->next;

}

else

{

temp = temp->next;

}

}

if (temp && temp->name == name)

{

temp->next = NULL;

}

}

cout << "Record was deleted" << endl;

}

void linkedList::SearchList(string name)

{

node\* temp = head;

while (temp)

{

if (temp->name == name)

{

cout << "Value Found." << endl;

break;

}

else

{

temp = temp->next;

}

}

}

int OpenHashing::hashvalue(string name)

{

int key = name[name.length() - 1] % 10;

return key;

}void OpenHashing::insert(string name, int number)

{

int temp\_index = hashvalue(name);

arr[temp\_index].InsertList(number, name);

}

void OpenHashing::search(string name)

{

int temp\_key = name[name.length() - 1] % 10;

arr[temp\_key].SearchList(name);

}

void OpenHashing::deleteV(string name)

{

int temp\_key = name[name.length() - 1] % 10;

arr[temp\_key].deleteList(name);

}

int main()

{

int choice;

string name, name2, name3;

int number;

OpenHashing OH1;

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

cout << "2. Search." << endl;

cout << "3. Delete." << endl;

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

do

{

cout << endl;

cout << "Enter choice : ";

cin >> choice;

switch (choice)

{

case 1:

cout << "Enter name : ";

cin >> name;

cout << "Enter phone number : ";

cin >> number;

OH1.insert(name, number);

break;

case 2:

cout << "Enter Name to Search: ";

cin >> name2;

OH1.search(name2);

break;

case 3:

cout << "Enter Name to Delete: ";

cin >> name3;

OH1.deleteV(name3);

break;

case 4:

cout << "EXIT." << endl;

break;

}

} while (choice != 4);

}

## Output:

Text

Description automatically generated Text

Description automatically generated

# Closed hashing / open addressing:

## ****Linear Probing:****

### Description:

Closed hashing with linear probing is a collision resolution technique where each slot in the hash table can hold only one element. When a collision occurs (hash function maps two keys to the same index), the algorithm linearly searches for the next available slot, wrapping around to the beginning of the table if necessary. This approach ensures that all elements are stored within the fixed-size table, optimizing memory usage but potentially leading to clustering and performance degradation under high load factors.

### Code:

#include<iostream>

#include<string>

using namespace std;

struct details

{

string name;

long long int phone;

};

class ClosedHashing

{

private:

details person[10];

int max\_length;

int length;

public:

ClosedHashing();

int hash\_function(char);

void insert\_item(string name, long long int phone);

void search(string name);

void deleteVal(string name);

};

ClosedHashing::ClosedHashing()

{

length = 0;

max\_length = 10;

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

{

person[i].name = "NULL";

}

}

int ClosedHashing::hash\_function(char val)

{

return val % 10;

}

void ClosedHashing::insert\_item(string name, long long int phone)

{

int val = name.length();

int location = hash\_function(val);

while (person[location].name != "NULL")

{

location++;

}

if (person[location].name == "NULL")

{

person[location].name = name;

person[location].phone = phone;

length++;

}

}

void ClosedHashing::search(string name)

{

int counter = 0;

details\* item = new details;

item->name = name;

int location;

int l = name.length();

location = hash\_function(l);

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

{

if (person[i].name == item->name)

{

cout << "Name : " << person[i].name << " " << "Phone no : " << person[i].phone << endl;

cout << endl << "Record found." << endl;

}

if (counter == 20)

{

cout << endl << "Record not found." << endl;

}

else

{

counter++;

}

}

}

void ClosedHashing::deleteVal(string name)

{

int counter = 0;

details\* item = new details;

item->name = name;

int location;

int l = name.length();

location = hash\_function(l);

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

{

if (person[i].name == item->name)

{

person[i].name = "NULL";

person[i].phone = -1;

cout << endl << "Record deleted." << endl;

}

if (counter == 20)

{

cout << endl << "Record not found." << endl;

}

else

{

counter++;

}

}

}

int main()

{

ClosedHashing C1;

long long int phone;

string name;

int choice;

string name1, name2;

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

cout << "2. Search." << endl;

cout << "3. Delete." << endl;

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

do

{

cout << endl;

cout << "Enter choice : ";

cin >> choice;

switch (choice)

{

case 1:

cout << "Enter name : ";

cin >> name1;

cout << "Enter phone number : ";

cin >> phone;

C1.insert\_item(name1, phone);

break;

case 2:

cout << "Enter Name to Search: ";

cin >> name2;

C1.search(name2);

break;

case 3:

cout << "Enter Name to Delete: ";

cin >> name;

C1.deleteVal(name);

break;

case 4:

cout << "EXIT." << endl;

break;

}

} while (choice != 4);

}

### Output:

Text

Description automatically generatedText

Description automatically generated

## ****Quadratic Probing:****

### Description:

Closed hashing with quadratic probing is a collision resolution technique where collisions in hash tables are resolved by placing the colliding element into the next available slot, incremented quadratically (i.e., probing with \( h(k, i) = (h'(k) + i^2) \mod m \), where \( h'(k) \) is the initial hash function value, \( i \) is the probe sequence, and \( m \) is the table size). This method reduces clustering and improves search performance by spreading out collided entries across the hash table.

### Code:

#include <iostream>

#include <string>

using namespace std;

const int TABLE\_SIZE = 10;

class HashTable {

private:

struct HashNode {

string key;

int value;

bool occupied;

HashNode() : key(""), value(0), occupied(false) {}

};

HashNode table[TABLE\_SIZE];

int hash(string key) {

int hashVal = 0;

for (int i = 0; i < key.length(); i++)

hashVal += key[i];

return hashVal % TABLE\_SIZE;

}

int quadraticProbe(int index, int attempt) {

return (index + attempt \* attempt) % TABLE\_SIZE;

}

public:

void insert(string key, int value) {

int index = hash(key);

int attempt = 0;

while (table[index].occupied) {

attempt++;

index = quadraticProbe(index, attempt);

}

table[index].key = key;

table[index].value = value;

table[index].occupied = true;

}

int search(string key) {

int index = hash(key);

int attempt = 0;

while (table[index].occupied && table[index].key != key) {

attempt++;

index = quadraticProbe(index, attempt);

}

if (table[index].occupied && table[index].key == key)

return table[index].value;

else

return -1; // Not found

}

void display() {

for (int i = 0; i < TABLE\_SIZE; i++) {

if (table[i].occupied)

cout << "Key: " << table[i].key << ", Value: " << table[i].value << endl;

else

cout << "Empty" << endl;

}

}

};

int main() {

HashTable ht;

ht.insert("John", 123456);

ht.insert("Mary", 789012);

ht.insert("Emma", 345678);

ht.insert("Bob", 901234);

cout << "Displaying Hash Table:" << endl;

ht.display();

string searchKey = "Mary";

int result = ht.search(searchKey);

if (result != -1)

cout << "Value of " << searchKey << ": " << result << endl;

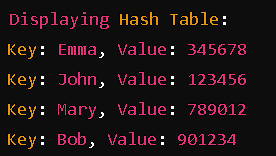
else

cout << "Key " << searchKey << " not found." << endl;

return 0;

}

### Output:

 A black background with pink text

Description automatically generated