A Lab report on

## Data Structures & Algorithms

### Faculty of Humanities and Social Science

### Tribhuvan University

**Submitted by:**

**Rijan Manandhar**

**Roll No.:**

**T.U. Registration No.:**

**Southwestern State College**

**Submitted to:**

### Swastika Banskota

**Department of Bachelor of Computer Application (BCA)**

### Southwestern State College

Basundhara, Kathmandu

In the partial fulfilment of the requirements of the degree of

Bachelor of Computer Application (BCA)

# Supervisor’s Recommendation

This is to certify that lab report, is an academic work done by **Rijan Manandhar** submitted in the partial fulfillment of the requirement for the degree of Bachelor in Computer Application (BCA) at Southwestern State College (SWSC) under my guidance and supervision. To the best of my knowledge, the work performed by his own creation.

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

**(Supervisor)**

Name: **Swastika Banskota**

Date:

# Acknowledgement

This is an attempt to present report entitled, A Lab Report On “Data Structures & Algorithms” prepared for partial fulfillment of the requirement for the Degree of Bachelor in Computer Application (BCA) is an outcome of immeasurable support.The writing of this project has been of the significant academic challenges we have faced and without the support and guidance of the people involved, this task wouldn't have been completed. It is to them we own our deepest gratitude.

I express my sincere honor and special sense of gratitude to my academic supervisor, Ms. Swastika Banskota. I would also like to thank academic head Mr. Ravi Chapagain and respected campus chief Mr. Rajendra KC of Southwestern State College for their generous guidance and brilliant insight throughout this work.

**Rijan Manandhar**

Date:

## Table of Contents

[Supervisor’s Recommendation ii](#_Toc167794379)

[Acknowledgement iii](#_Toc167794380)

[LAB WORK 1](#_Toc167794381)

[1. WAP to implement to implement dynamic memory allocation and de-allocation. 1](#_Toc167794382)

[2. WAP using stack to convert infix expression to postfix expression. 3](#_Toc167794383)

[3. WAP using stack to convert infix expression to prefix expression. 6](#_Toc167794384)

[4. WAP to evaluate postfix/prefix expression. 13](#_Toc167794385)

[5. WAP to implement queue operations for linear, circular, and priority queue. 18](#_Toc167794386)

[6. WAP to implement Fibonacci sequence and Tower of Hanoi algorithms using Recursion. 32](#_Toc167794387)

[7. WAP to implement different operations related to linked list: singly and doubly linked lists. 35](#_Toc167794388)

[8. WAP to implement Binary Search Trees and AVL Trees. 40](#_Toc167794389)

[9. WAP to implement Merge Sort. 48](#_Toc167794390)

[10. WAP to implement searching and hashing algorithms. 51](#_Toc167794391)

[11. WAP to implement spanning tree and shortest path. 55](#_Toc167794392)

…………………..

Lecturer

Swastika Banskota

# LAB WORK

# WAP to implement to implement dynamic memory allocation and de-allocation.

**Theory:-**

Dynamic memory allocation refers to the process of allocating memory for variables and data structures at runtime, as opposed to static allocation which happens at compile time. This is typically done using a portion of memory called the heap.

Here's a breakdown of the concept:

* **Allocation:** When your program needs memory for data that isn't known beforehand (like the size of an array entered by the user), it can request a block of memory from the heap during program execution. This is done using functions like malloc and calloc in C, or the new operator in C++. These functions return a pointer to the allocated memory block.
* **Deallocation**: When you're finished using the dynamically allocated memory, it's crucial to release it back to the heap to prevent memory leaks. This is done using functions like free in C or the delete operator in C++. Failing to release unused memory can lead to your program consuming more and more memory over time.

**Code:-**

#include <stdio.h>

#include <stdlib.h>

int main() {

int num\_elements, i;

int \*ptr;

// Get the number of elements from the user

printf("Enter the number of elements: ");

scanf("%d", &num\_elements);

// Allocate memory for the array using malloc

ptr = (int\*)malloc(num\_elements \* sizeof(int));

// Check if memory allocation was successful

if (ptr == NULL) {

printf("Memory allocation failed!\n");

return 1;

}

// Access and potentially modify elements

printf("Enter %d integers:\n", num\_elements);

for (i = 0; i < num\_elements; i++) {

scanf("%d", ptr + i);

}

// Print the elements (optional)

printf("The entered elements are:\n");

for (i = 0; i < num\_elements; i++) {

printf("%d ", \*(ptr + i));

}

printf("\n");

// Deallocate the memory using free

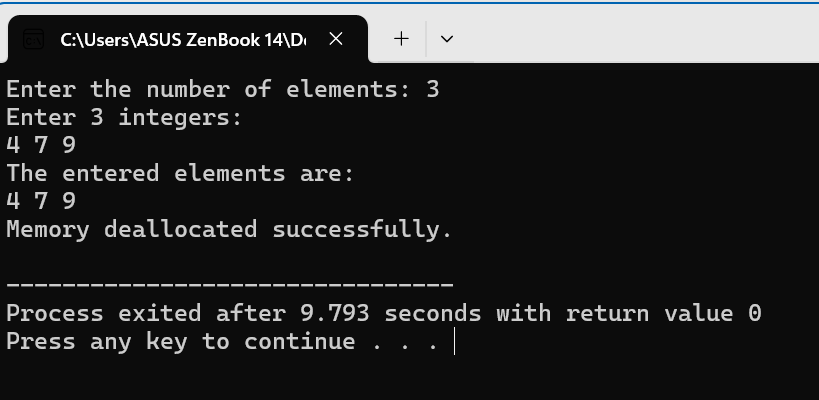
free(ptr);

printf("Memory deallocated successfully.\n");

return 0;

}

**Output**



# WAP using stack to convert infix expression to postfix expression.

**Theory:-**

Infix notation is the standard way we write mathematical expressions, with operands (numbers or variables) placed between operators. Postfix notation, on the other hand, puts the operator after its operands. Here's how to convert infix to postfix using a stack:

**i. Utilize a Stack:**

A stack is a data structure that follows the "Last In, First Out" (LIFO) principle. We'll use it to store operators temporarily.

**ii. Scan the Infix Expression:**

Read the infix expression from left to right one symbol at a time.

**iii. Process Each Symbol:**

* **Operand:** If you encounter an operand (number or variable), simply append it to the postfix expression.
* **Left Parenthesis ("("):** Push the left parenthesis onto the stack.
* **Right Parenthesis (")"):** Pop operators from the stack and append them to the postfix expression until you encounter the matching left parenthesis. Discard the parentheses.
* **Operator:**
  + If the stack is empty or the top element is a left parenthesis, push the current operator onto the stack.
  + Otherwise, compare the precedence of the current operator with the one on top of the stack.
  + If the current operator has higher precedence, push it onto the stack.
  + If the current operator has lower precedence or equal precedence with left associativity (e.g., +, -), pop operators from the stack and append them to the postfix expression until you find an operator with lower precedence or a left parenthesis. Then, push the current operator onto the stack.

**iv. Precedence:** Operators have different precedence levels, which determine the order of operations. Higher precedence operators are evaluated first. Common precedence levels (from highest to lowest):

1. Exponentiation (^)
2. Multiplication (\*) and Division (/)
3. Addition (+) and Subtraction (-)

**Code:-**

#include<stdio.h>

#include<ctype.h>

char stack[100];

int top = -1;

void push(char x)

{

stack[++top] = x;

}

char pop()

{

if(top == -1)

return -1;

else

return stack[top--];

}

int priority(char x)

{ if(x == '(')

return 0;

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

return 1;

if(x == '\*' || x == '/')

return 2;

return 0;

}

int main()

{ char exp[100];

char \*e, x;

printf("Enter the expression : ");

scanf("%s",exp);

printf("\n");

e = exp;

while(\*e != '\0')

{

if(isalnum(\*e))

printf("%c ",\*e);

else if(\*e == '(')

push(\*e);

else if(\*e == ')')

{

while((x = pop()) != '(')

printf("%c ", x);

} else {

while(priority(stack[top]) >= priority(\*e))

printf("%c ",pop());

push(\*e);

}

e++;

} while(top != -1)

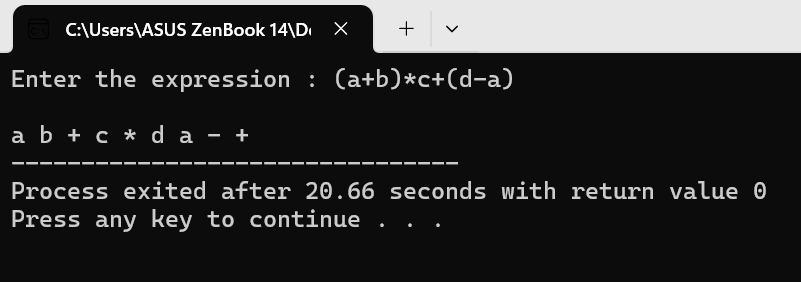
{

printf("%c ",pop());

}

return 0;

}

**Output**

# WAP using stack to convert infix expression to prefix expression.

**Theory:-**

Infix expressions have operators placed between operands, while prefix expressions put the operator before the operands. Converting infix to prefix involves a similar approach to postfix conversion, but with some key differences. Here's how to do it:

**1. Utilize a Stack:**

Similar to postfix conversion, we'll use a stack to store operators temporarily.

**2. Reverse the Infix Expression:**

The first step in infix to prefix conversion differs from postfix. We need to reverse the entire infix expression before processing it. This ensures the operators are encountered in the correct order for prefix notation.

**3. Scan the Reversed Expression:**

Read the reversed expression from left to right one symbol at a time.

**4. Process Each Symbol:**

* **Operand:** If you encounter an operand (number or variable), simply prepend it to the prefix expression.
* **Right Parenthesis (")"):** Push the right parenthesis onto the stack (similar to a left parenthesis in postfix conversion).
* **Left Parenthesis ("("):** Pop operators from the stack and prepend them to the prefix expression until you encounter the matching right parenthesis. Discard the parentheses.
* **Operator:**
  + If the stack is empty, push the current operator onto the stack.
  + Otherwise, compare the precedence of the current operator with the one on top of the stack.
  + If the current operator has higher precedence, push it onto the stack.
  + If the current operator has lower precedence or equal precedence, pop operators from the stack and prepend them to the prefix expression until you find an operator with lower precedence or a right parenthesis. Then, push the current operator onto the stack.

**5. After Scanning:**

* Pop any remaining operators from the stack and prepend them to the prefix expression.

**Precedence:**

The precedence rules remain the same as in infix to postfix conversion:

1. Exponentiation (^)
2. Multiplication (\*) and Division (/)
3. Addition (+) and Subtraction (-)

**Code:-**

#include<string.h>

#include<limits.h>

#include<stdio.h>

#include<stdlib.h>

#define MAX 100

// A structure to represent a stack

struct Stack

{

int top;

int maxSize;

int \*array;

}; struct Stack \*create (int max)

{

struct Stack \*stack = (struct Stack \*) malloc (sizeof (struct Stack));

stack->maxSize = max;

stack->top = -1;

stack->array = (int \*) malloc (stack->maxSize \* sizeof (int));

return stack;

}

// Checking with this function is stack is full or not

// Will return true is stack is full else false

//Stack is full when top is equal to the last index

int isFull (struct Stack \*stack)

{

if (stack->top == stack->maxSize - 1)

{

printf ("Will not be able to push maxSize reached\n");

}

// Since array starts from 0, and maxSize starts from 1

return stack->top == stack->maxSize - 1;

}

// By definition the Stack is empty when top is equal to -1

// Will return true if top is -1

int isEmpty (struct Stack \*stack)

{

return stack->top == -1;

}

// Push function here, inserts value in stack and increments stack top by 1

void push (struct Stack \*stack, char item)

{

if (isFull (stack))

return;

stack->array[++stack->top] = item; }

// Function to remove an item from stack. It decreases top by 1

int pop (struct Stack \*stack)

{

if (isEmpty (stack))

return INT\_MIN;

return stack->array[stack->top--];

}

// Function to return the top from stack without removing it

int peek (struct Stack \*stack)

{

if (isEmpty (stack))

return INT\_MIN;

return stack->array[stack->top];

}

// A utility function to check if the given character is operand

int checkIfOperand (char ch)

{

return (ch >= 'a' && ch <= 'z') || (ch >= 'A' && ch <= 'Z');

}

// Fucntion to compare precedence

// If we return larger value means higher precedence

int precedence (char ch)

{

switch (ch)

{

case '+':

case '-':

return 1;

case '\*':

case '/':

return 2;

case '^':

return 3;

}

return -1;

}

// The driver function for infix to postfix conversion

int getPostfix (char \*expression)

{

int i, j;

// Stack size should be equal to expression size for safety

struct Stack \*stack = create (strlen (expression));

if (!stack) // just checking is stack was created or not

return -1;

for (i = 0, j = -1; expression[i]; ++i)

{

if (checkIfOperand (expression[i]))

expression[++j] = expression[i];

else if (expression[i] == '(')

push (stack, expression[i]);

else if (expression[i] == ')')

{

while (!isEmpty (stack) && peek (stack) != '(')

expression[++j] = pop (stack);

if (!isEmpty (stack) && peek (stack) != '(')

return -1; // invalid expression

else

pop (stack);

}

else // if an opertor

{

while (!isEmpty (stack)

&& precedence (expression[i]) <= precedence (peek (stack)))

expression[++j] = pop (stack);

push (stack, expression[i]);

}

}

// Once all inital expression characters are traversed

// adding all left elements from stack to exp

while (!isEmpty (stack))

expression[++j] = pop (stack);

expression[++j] = '\0';

}

void reverse (char \*exp)

{

int size = strlen (exp);

int j = size, i = 0;

char temp[size];

temp[j--] = '\0';

while (exp[i] != '\0')

{

temp[j] = exp[i];

j--;

i++;

}

strcpy (exp, temp);

}

void brackets (char \*exp)

{

int i = 0;

while (exp[i] != '\0')

{

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

exp[i] = ')';

else if (exp[i] == ')')

exp[i] = '(';

i++;

}

}

void InfixtoPrefix (char \*exp)

{

int size = strlen (exp);

// reverse string

reverse (exp);

//change brackets

brackets (exp);

//get postfix

getPostfix (exp);

// reverse string again

reverse (exp);

}

int main ()

{

printf ("The infix is: ");

char expression[] = "((a/b)+c)-(d+(e\*f))";

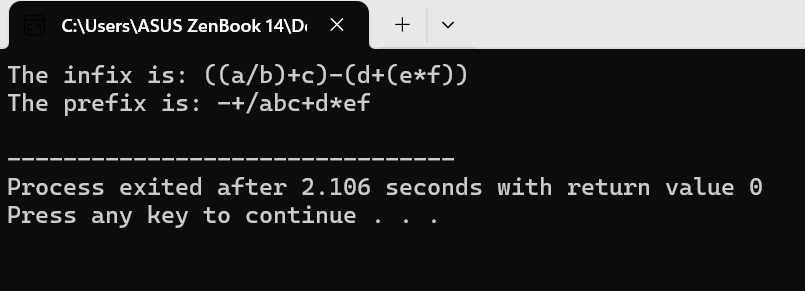
printf ("%s\n", expression);

InfixtoPrefix (expression);

printf ("The prefix is: ");

printf ("%s\n", expression);

return 0; }

**Output**

# WAP to evaluate postfix/prefix expression.

**Theory:-**

Both postfix and prefix expressions can be evaluated using a stack. Here's how:

**Evaluating Postfix Expressions:**

1. **Utilize a Stack:** Create a stack to store operands (numbers or variables) during the evaluation process.
2. **Scan the Postfix Expression:** Read the expression from left to right one symbol at a time.
3. **Process Each Symbol:**
   * **Operand:** If you encounter an operand (number), push it onto the stack.
   * **Operator:**
     + Pop two operands (b and a) from the stack, where b is the top element and a is the element below it.
     + Perform the operation based on the operator (+, -, \*, /) on the operands (b op a).
     + Push the result of the operation back onto the stack.
4. **After Scanning:**
   * The only element remaining in the stack will be the final result of the expression.

**Evaluating Prefix Expressions:**

1. **Utilize a Stack:** Similar to postfix, use a stack to store operands during evaluation.
2. **Reverse the Prefix Expression (Optional):** While not strictly necessary, some algorithms choose to reverse the prefix expression before processing. This can simplify the logic by treating it similarly to postfix evaluation. You can choose to do this step or proceed directly to step 3.
3. **Scan the Expression (Reversed or Original):** Read the expression (reversed or original depending on your approach) from right to left one symbol at a time.
4. **Process Each Symbol:**
   * **Operand:** If you encounter an operand (number), push it onto the stack.
   * **Operator:**
     + Pop two operands (a and b) from the stack, where a is the top element and b is the element below it.
     + Perform the operation based on the operator (op a, b).
     + Push the result of the operation back onto the stack.
5. **After Scanning:**
   * The only element remaining in the stack will be the final result of the expression.

**Code:-**

#include <stdio.h>

#include <stdlib.h>

#define MAX\_SIZE 100

// Stack implementation

int stack[MAX\_SIZE];

int top = -1;

void push(int item) {

if (top >= MAX\_SIZE - 1) {

printf("Stack Overflow\n");

return;

}

top++;

stack[top] = item;

}

int pop() {

if (top < 0) {

printf("Stack Underflow\n");

return -1;

}

int item = stack[top];

top--;

return item;

}

int is\_operator(char symbol) {

if (symbol == '+' || symbol == '-' || symbol == '\*' || symbol == '/') {

return 1;

}

return 0;

}

int evaluate(char\* expression) {

int i = 0;

char symbol = expression[i];

int operand1, operand2, result;

while (symbol != '\0') {

if (symbol >= '0' && symbol <= '9') {

int num = symbol - '0';

push(num);

} else if (is\_operator(symbol)) {

operand2 = pop();

operand1 = pop();

switch(symbol) {

case '+': result = operand1 + operand2; break;

case '-': result = operand1 - operand2; break;

case '\*': result = operand1 \* operand2; break;

case '/': result = operand1 / operand2; break;

} push(result);

} i++;

symbol = expression[i];

} result = pop();

return result;

} int main() {

char expression[] = "5 6 7 + \* 8 -";

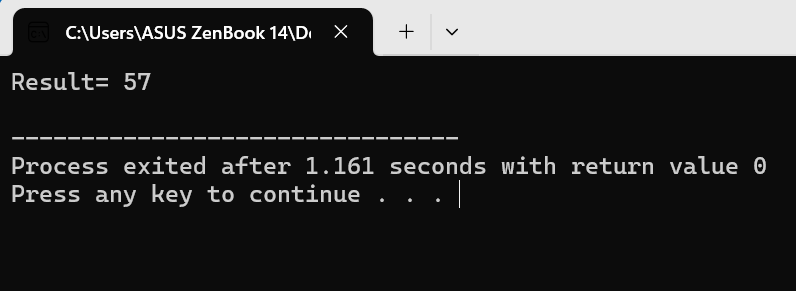
int result = evaluate(expression);

printf("Result= %d\n", result);

return 0;

}

**Output**



**Code:-**

#include <bits/stdc++.h>

using namespace std;

bool isOperand(char c)

{

// If the character is a digit then it must

// be an operand

return isdigit(c);

}

double evaluatePrefix(string exprsn)

{

stack<double> Stack;

for (int j = exprsn.size() - 1; j >= 0; j--) {

// Push operand to Stack

// To convert exprsn[j] to digit subtract

// '0' from exprsn[j].

if (isOperand(exprsn[j]))

Stack.push(exprsn[j] - '0');

else {

// Operator encountered

// Pop two elements from Stack

double o1 = Stack.top();

Stack.pop();

double o2 = Stack.top();

Stack.pop();

// Use switch case to operate on o1

// and o2 and perform o1 Or o2.

switch (exprsn[j]) {

case '+':

Stack.push(o1 + o2);

break;

case '-':

Stack.push(o1 - o2);

break;

case '\*':

Stack.push(o1 \* o2);

break;

case '/':

Stack.push(o1 / o2);

break;

}

}

}

return Stack.top();

}

// Driver code

int main()

{

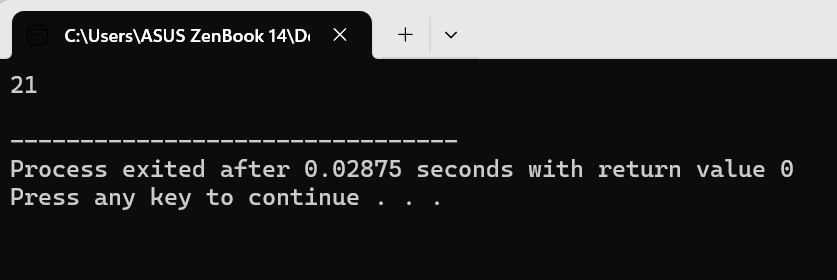
string exprsn = "+9\*26";

cout << evaluatePrefix(exprsn) << endl;

return 0;

}

**Output**



# WAP to implement queue operations for linear, circular, and priority queue.

**Theory:-**

**Linear Queue**

Representation: Typically implemented using an array.

Operations:

enqueue(data): Inserts an element data at the rear of the queue. If the queue is full (rear == size-1), throws an overflow exception.

dequeue(): Removes and returns the element at the front of the queue. If the queue is empty (front == -1), throws an underflow exception.

isEmpty(): Checks if the queue is empty (front == -1).

isFull(): Checks if the queue is full (rear == size-1).

**Circular Queue**

Representation: Implemented using an array with a fixed size. It utilizes modulo operation (%) to handle wrapping around the array.

Operations: Similar to linear queue, but without overflow checks as the queue can never truly be full due to wrapping.

Additional Note: A separate variable count is used to track the number of elements in the queue, as front and rear might have the same value when the queue is empty.

**Priority Queue**

Representation: Often implemented using a heap data structure (min-heap or max-heap) to prioritize elements based on their associated values.

Operations:

enqueue(data, priority): Inserts an element data with a corresponding priority into the queue. Lower values (for min-heap) or higher values (for max-heap) indicate higher priority.

dequeue(): Removes and returns the element with the highest priority (minimum for min-heap, maximum for max-heap).

isEmpty(): Checks if the queue is empty.

**Code:-**

#include <stdio.h>

#define MAX\_SIZE 100

int queue[MAX\_SIZE];

int front = -1, rear = -1;

// Function prototypes

void enqueue(int data);

int dequeue();

int isEmpty();

int isFull();

void display();

int main() {

int choice, data;

while (1) {

printf("\n1. Enqueue\n2. Dequeue\n3. Display\n4. Exit\n");

printf("Enter your choice: ");

scanf("%d", &choice);

switch (choice) {

case 1:

printf("Enter data to enqueue: ");

scanf("%d", &data);

if (isFull()) {

printf("Queue Overflow\n");

} else {

enqueue(data);

printf("Data enqueued successfully\n");

}

break;

case 2:

data = dequeue();

if (isEmpty()) {

printf("Queue Underflow\n");

} else {

printf("Dequeued data: %d\n", data);

}

break;

case 3:

display();

break;

case 4:

printf("Exiting program\n");

return 0;

default:

printf("Invalid choice\n");

}

}

return 0;

}

// Enqueue operation: Adds element to the rear of the queue

void enqueue(int data) {

if (rear == MAX\_SIZE - 1) {

return; // Queue is full

}

if (front == -1) {

front = 0;

}

rear++;

queue[rear] = data;

}

// Dequeue operation: Removes element from the front of the queue

int dequeue() {

if (isEmpty()) {

return -1; // Queue is empty

}

int data = queue[front];

if (front == rear) {

front = rear = -1;

} else {

front++;

}

return data;

}

// Checks if the queue is empty

int isEmpty() {

return front == -1;

}

// Checks if the queue is full

int isFull() {

return rear == MAX\_SIZE - 1;

}

// Displays the elements of the queue

void display() {

if (isEmpty()) {

printf("Queue is empty\n");

return;

}

printf("Queue elements: ");

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

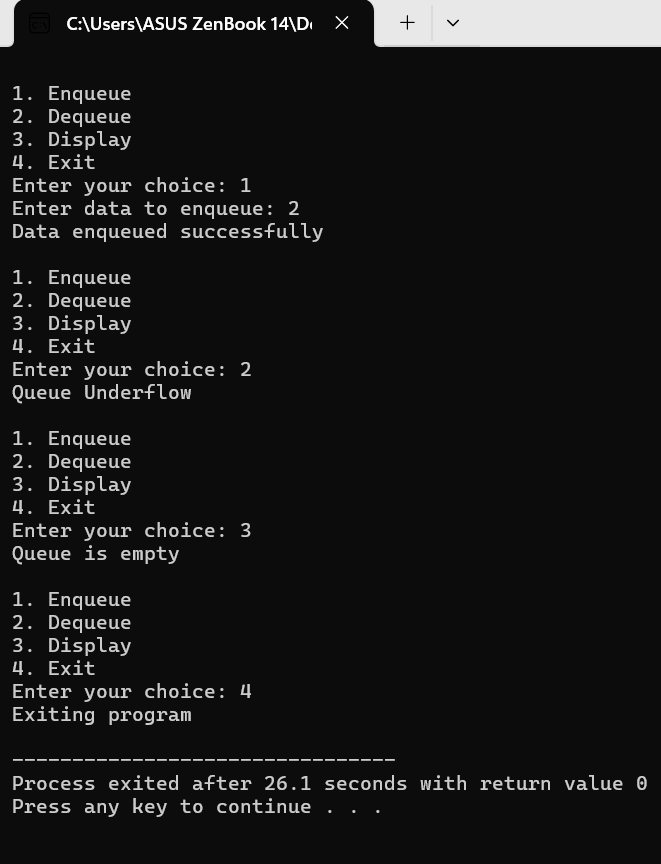
printf("%d ", queue[i]);

}

printf("\n");

}

**Output**

****

**Code:-**

#include <stdio.h>

#define MAX\_SIZE 100

int queue[MAX\_SIZE];

int front = -1, rear = -1;

// Function prototypes

void enqueue(int data);

int dequeue();

int isEmpty();

int isFull();

void display();

int main() {

int choice, data;

while (1) {

printf("\n1. Enqueue\n2. Dequeue\n3. Display\n4. Exit\n");

printf("Enter your choice: ");

scanf("%d", &choice);

switch (choice) {

case 1:

printf("Enter data to enqueue: ");

scanf("%d", &data);

if (isFull()) {

printf("Queue Overflow\n");

} else {

enqueue(data);

printf("Data enqueued successfully\n");

}

break;

case 2:

data = dequeue();

if (isEmpty()) {

printf("Queue Underflow\n");

} else {

printf("Dequeued data: %d\n", data);

}

break;

case 3:

display();

break;

case 4:

printf("Exiting program\n");

return 0;

default:

printf("Invalid choice\n");

}

}

return 0;

}

// Enqueue operation: Adds element to the rear of the queue (circular logic)

void enqueue(int data) {

if ((front == 0 && rear == MAX\_SIZE - 1) || (front == rear + 1)) {

printf("Queue Overflow\n");

return;

}

if (front == -1) {

front = 0;

}

rear = (rear + 1) % MAX\_SIZE;

queue[rear] = data;

}

// Dequeue operation: Removes element from the front of the queue (circular logic)

int dequeue() {

if (isEmpty()) {

return -1; // Queue is empty

}

int data = queue[front];

if (front == rear) {

front = rear = -1;

} else {

front = (front + 1) % MAX\_SIZE;

}

return data;

}

// Checks if the queue is empty

int isEmpty() {

return front == -1;

}

// Checks if the queue is full (considering circular property)

int isFull() {

return (rear + 1) % MAX\_SIZE == front;

}

// Displays the elements of the queue

void display() {

if (isEmpty()) { printf("Queue is empty\n");

return;

} printf("Queue elements: ");

if (front <= rear) {

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

printf("%d ", queue[i]);

}

} else {

// Handle wrapping around the array

for (int i = front; i < MAX\_SIZE; i++) {

printf("%d ", queue[i]);

} for (int i = 0; i <= rear; i++) {

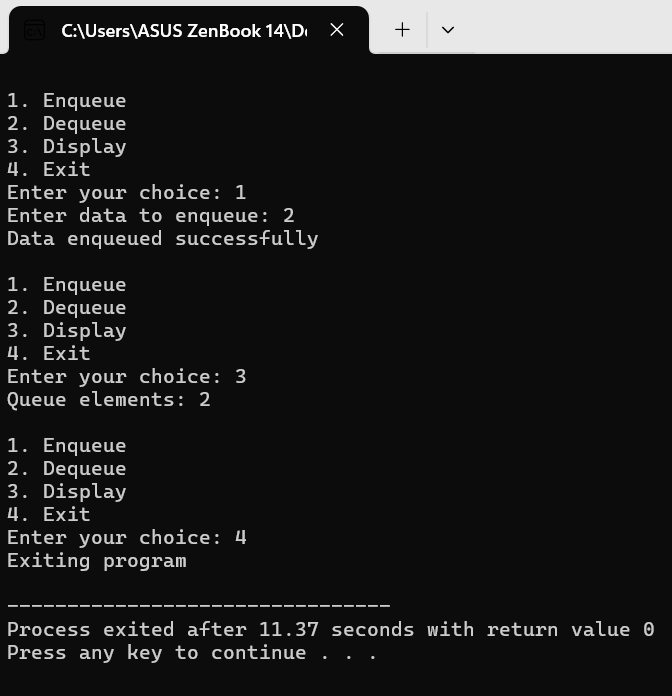
printf("%d ", queue[i]);

}

} printf("\n");

}

**Output**

****

**Code:-**

#include <stdio.h>

#define MAX\_SIZE 10

// Structure to represent a node in the priority queue

typedef struct {

int data;

int priority;

} Node;

// Function prototypes

void swap(Node \*a, Node \*b);

void heapify(Node arr[], int n, int i);

int parent(int i);

int left(int i);

int right(int i);

// Global variables

Node queue[MAX\_SIZE];

int front = -1, rear = -1;

// Function to swap two nodes

void swap(Node \*a, Node \*b) {

Node temp = \*a;

\*a = \*b;

\*b = temp;

}

// Function to maintain max-heap property

void heapify(Node arr[], int n, int i) {

int largest = i;

int l = left(i);

int r = right(i);

// Check if left child is larger than root

if (l < n && arr[l].priority > arr[largest].priority)

largest = l;

// Check if right child is larger than largest so far

if (r < n && arr[r].priority > arr[largest].priority)

largest = r;

// If largest is not root

if (largest != i) {

swap(&arr[i], &arr[largest]);

// Recursively heapify the affected sub-tree

heapify(arr, n, largest);

}

}

// Function to get the parent index

int parent(int i) {

return (i - 1) / 2;

}

// Function to get the left child index

int left(int i) {

return 2 \* i + 1;

}

// Function to get the right child index

int right(int i) {

return 2 \* i + 2;

}

// Function to check if the queue is empty

int isEmpty() {

return front == -1;

}

// Function to check if the queue is full

int isFull() {

return rear == MAX\_SIZE - 1;

}

// Function to insert an element in the priority queue

void enqueue(int data, int priority) {

if (isFull()) {

printf("Queue overflow\n");

return;

}

// Create a new node

Node newNode = {data, priority};

rear++;

queue[rear] = newNode;

// Heapify bottom-up to maintain max-heap property

int i = rear;

while (i > 0 && queue[parent(i)].priority < queue[i].priority) {

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

i = parent(i);

}

}

// Function to remove the element with highest priority

Node dequeue() {

if (isEmpty()) {

printf("Queue underflow\n");

return queue[0]; // Dummy node to avoid errors

}

Node temp = queue[front];

// If there is only one element, simply decrement rear

if (front == rear) {

front = rear = -1;

} else {

// Move the last element to the root

queue[front] = queue[rear];

rear--;

// Heapify top-down to maintain max-heap property

heapify(queue, rear + 1, front);

}

return temp;

}

// Function to display the elements in the queue

void display() {

if (isEmpty()) {

printf("Queue is empty\n");

return;

}

printf("Queue elements: ");

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

printf("%d(%d) ", queue[i].data, queue[i].priority);

printf("\n");

}

int main() {

enqueue(5, 2);

enqueue(3, 1);

enqueue(1, 3);

printf("Queue after insertion:\n");

display();

Node removed = dequeue();

printf("Removed element: %d (%d)\n", removed.data, removed.priority);

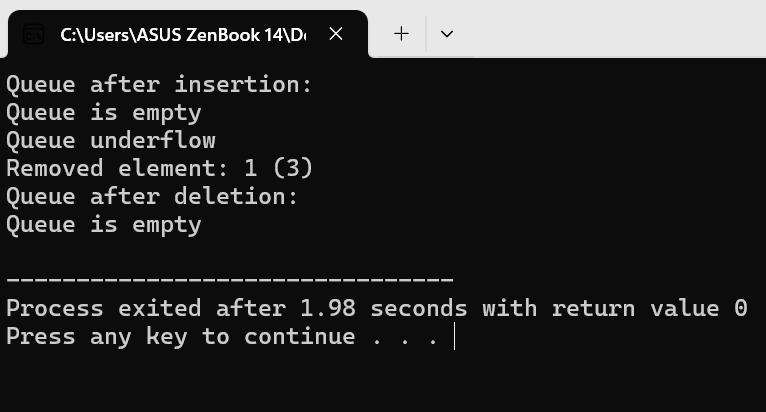
printf("Queue after deletion:\n");

display();

return 0;

}

**Output**



# WAP to implement Fibonacci sequence and Tower of Hanoi algorithms using Recursion.

**Theory:-**

**Fibonacci Sequence**

* **Definition:** A sequence where each number is the sum of the two preceding numbers. Starting from 0 and 1, the sequence goes 0, 1, 1, 2, 3, 5, 8, 13, ...
* **Formula:** F(n) = F(n-1) + F(n-2), where F(n) is the nth Fibonacci number. Alternatively, the closed-form solution involves the golden ratio.
* **Applications:** It finds applications in various fields, including computer science (e.g., recursion practice, generating pseudorandom numbers), mathematics (modeling growth patterns), and even art and music (aesthetics based on the golden ratio).

**Tower of Hanoi**

* **Problem:** A puzzle involving disks of different sizes stacked on three pegs. The objective is to move all disks to a different peg, following these rules:
  + Only one disk can be moved at a time.
  + A larger disk cannot be placed on a smaller disk.
* **Solution:** There's a recursive approach to solve the Tower of Hanoi. The strategy involves breaking down the problem into smaller subproblems:
  + Move the top n-1 disks to a temporary peg (not the destination).
  + Move the largest disk to the destination peg.
  + Move the n-1 disks from the temporary peg to the destination peg, stacking them on top of the largest disk.

**Code:-**

#include <stdio.h>

int fibonacci(int n) {

if (n <= 1) {

return n;

} else {

return fibonacci(n - 1) + fibonacci(n - 2);

}

}

int main() {

int num;

printf("Enter the number of terms for the Fibonacci Series: ");

scanf("%d", &num);

printf("Fibonacci Series: ");

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

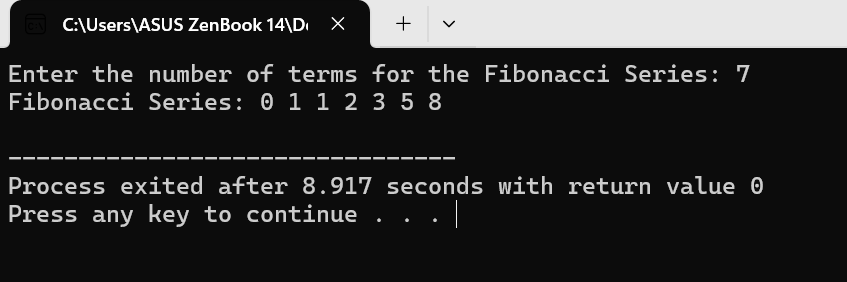
printf("%d ", fibonacci(i));

}

printf("\n");

return 0;

}

**Output**

**Code:-**

#include <stdio.h>

void towerOfHanoi(int n, char from\_rod, char to\_rod, char aux\_rod) {

if (n == 1) {

printf("Move disk 1 from rod %c to rod %c \n", from\_rod, to\_rod);

return;

}

towerOfHanoi(n - 1, from\_rod, aux\_rod, to\_rod);

printf("Move disk %d from rod %c to rod %c \n", n, from\_rod, to\_rod);

towerOfHanoi(n - 1, aux\_rod, to\_rod, from\_rod);

}

int main() {

int num\_disks;

printf("Enter the number of disks: ");

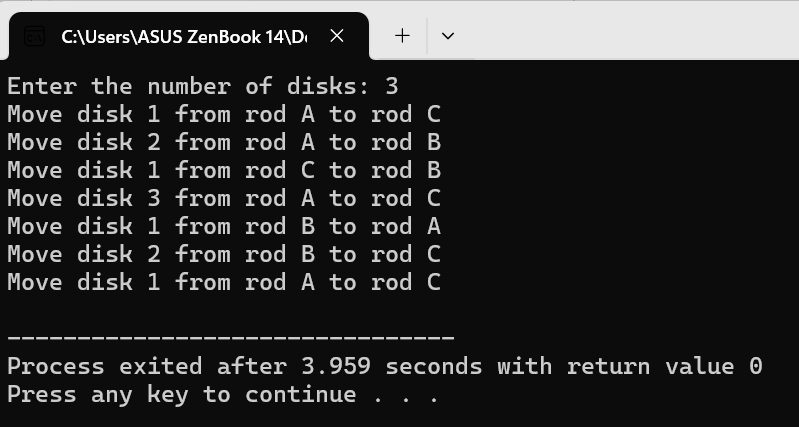
scanf("%d", &num\_disks);

towerOfHanoi(num\_disks, 'A', 'C', 'B'); // Assuming rods are named A, B, and C

return 0;

}

**Output**



# WAP to implement different operations related to linked list: singly and doubly linked lists.

**Theory:-**

**Singly Linked List:**

* **Structure:** Each node in a singly linked list contains two parts:
  + **Data:** The actual information stored in the node (can be any data type).
  + **Next pointer:** A reference to the next node in the sequence.
* **Operations:** Common operations on singly linked lists include:
  + **Traversal:** Visiting each node in the list, typically starting from the head (first node).
  + **Insertion:** Adding a new node at the beginning, end, or a specific position in the list.
  + **Deletion:** Removing a node based on its value or position.
  + **Searching:** Finding a node containing a specific value.
* **Advantages:**
  + Simpler to implement compared to doubly linked lists.
  + Less memory overhead as each node only stores one pointer.
* **Disadvantages:**
  + Less efficient for operations that require traversing backward in the list, as you need to start from the head and iterate through each node.

**Doubly Linked List:**

* **Structure:** Each node in a doubly linked list contains three parts:
  + **Data:** Similar to singly linked lists.
  + **Next pointer:** A reference to the next node in the sequence.
  + **Previous pointer:** A reference to the previous node in the sequence.
* **Operations:** Doubly linked lists support the same operations as singly linked lists, but with the ability to traverse in both directions (forward and backward) due to the presence of the previous pointer.
* **Advantages:**
  + Efficient for operations requiring backward traversal.
  + Useful when you need to modify the previous node during operations.
* **Disadvantages:**
  + More complex to implement compared to singly linked lists.
  + Requires more memory overhead as each node stores two pointers.

**Code:-**

#include <stdio.h>

#include <stdlib.h>

// Define the node structure

struct Node {

int data;

struct Node\* next;

};

// Function prototypes

void insertAtEnd(struct Node\*\* head\_ref, int new\_data);

void printList(struct Node\* node);

void freeList(struct Node\* head);

int main() {

// Initialize an empty list

struct Node\* head = NULL;

// Insert elements at the end

insertAtEnd(&head, 1);

insertAtEnd(&head, 2);

insertAtEnd(&head, 3);

printf("Created Linked List: ");

printList(head);

// Free the allocated memory

freeList(head);

return 0;

}

// Inserts a new node at the end of the linked list

void insertAtEnd(struct Node\*\* head\_ref, int new\_data) {

// Allocate memory for the new node

struct Node\* new\_node = (struct Node\*)malloc(sizeof(struct Node));

// Assign data and set next as NULL

new\_node->data = new\_data;

new\_node->next = NULL;

// If the list is empty, make the new node as head

if (\*head\_ref == NULL) {

\*head\_ref = new\_node;

return;

}

// Traverse to the last node

struct Node\* last = \*head\_ref;

while (last->next != NULL) {

last = last->next;

}

// Change the last node's next to point to the new node

last->next = new\_node;

}

// Prints the contents of the linked list

void printList(struct Node\* node) {

while (node != NULL) {

printf("%d ", node->data);

node = node->next;

} printf("\n");

}

// Frees the allocated memory for the linked list

void freeList(struct Node\* head) {

struct Node\* temp;

while (head != NULL) {

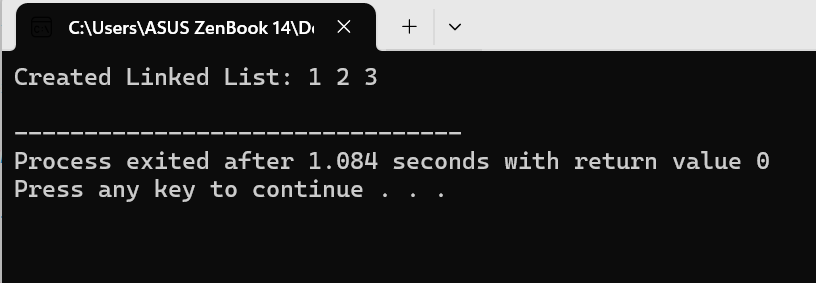
temp = head;

head = head->next;

free(temp); }

}

**Output**



**Code:-**

#include <stdio.h>

#include <stdlib.h>

// Define the node structure

struct Node {

int data;

struct Node\* prev;

struct Node\* next;

};

// Function prototypes

void insertAtEnd(struct Node\*\* head\_ref, int new\_data);

void printList(struct Node\* node);

void freeList(struct Node\* head);

int main() {

// Initialize an empty list

struct Node\* head = NULL;

// Insert elements at the end

insertAtEnd(&head, 1);

insertAtEnd(&head, 2);

insertAtEnd(&head, 3);

printf("Created Doubly Linked List: ");

printList(head);

// Free the allocated memory

freeList(head);

return 0;

}

// Inserts a new node at the end of the doubly linked list

void insertAtEnd(struct Node\*\* head\_ref, int new\_data) {

// Allocate memory for the new node

struct Node\* new\_node = (struct Node\*)malloc(sizeof(struct Node));

// Assign data and set next and prev as NULL

new\_node->data = new\_data;

new\_node->prev = NULL;

new\_node->next = NULL;

// If the list is empty, make the new node as head

if (\*head\_ref == NULL) {

\*head\_ref = new\_node;

return;

}

// Traverse to the last node

struct Node\* last = \*head\_ref;

while (last->next != NULL) {

last = last->next;

}

// Change the last node's next to point to the new node

last->next = new\_node;

// Set the new node's prev to point to the last node

new\_node->prev = last;

}

// Prints the contents of the doubly linked list

void printList(struct Node\* node) {

while (node != NULL) {

printf("%d ", node->data);

node = node->next;

}

printf("\n");

}

// Frees the allocated memory for the doubly linked list

void freeList(struct Node\* head) {

struct Node\* temp;

while (head != NULL) {

temp = head;

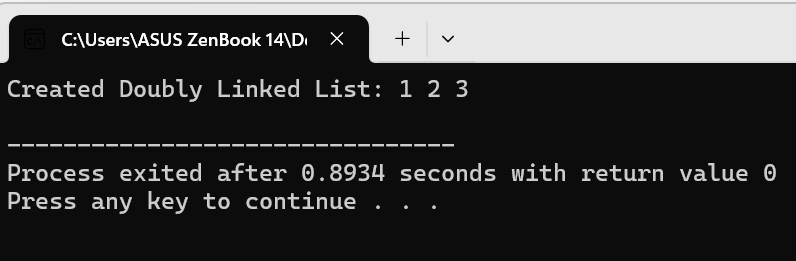
head = head->next;

free(temp);

}

}

**Output**



# WAP to implement Binary Search Trees and AVL Trees.

**Theory:-**

Both Binary Search Trees (BSTs) and AVL Trees are fundamental data structures used for efficient searching and sorting. Here's a breakdown of their characteristics and how they differ:

**Binary Search Trees (BSTs):**

* **Basic principle:** BSTs maintain a property where each node has a value greater than all elements in its left subtree and less than all elements in its right subtree. This allows for efficient searching by traversing the tree based on the target value.
* **Searching:** Start at the root node. If the target value is equal to the current node's value, you've found it. If the target value is less, move to the left subtree; if it's greater, move to the right subtree. Repeat until you find the target value or reach an empty subtree, indicating the target isn't present.
* **Insertion:** New nodes are inserted following the BST property, maintaining the ordering within the tree.
* **Deletion:** Deleting a node involves finding it and then rearranging the tree based on the node's structure (one or two children) to uphold the BST property.
* **Efficiency:** Searching, insertion, and deletion in a balanced BST have an average time complexity of O(log n), where n is the number of nodes. However, BSTs can become unbalanced if insertions or deletions are skewed, leading to a worst-case time complexity of O(n) for these operations.

**AVL Trees:**

* **BST with a balance factor:** AVL trees are a special type of BST that enforces an additional property: the balance factor. Each node in an AVL tree has a balance factor, which is the difference between the heights of its left and right subtrees. This factor must be within -1 and 1 for the tree to be considered balanced.
* **Self-balancing:** AVL trees perform rotations (restructuring subtrees) during insertions and deletions to maintain the balance factor. These rotations ensure the tree stays roughly balanced, even with a sequence of skewed insertions or deletions.
* **Guaranteed efficiency:** Due to self-balancing, AVL trees guarantee a time complexity of O(log n) for search, insertion, and deletion operations in most cases, even for unbalanced data.

**Code:-**

#include<stdio.h>

#include<conio.h>

#include<stdlib.h>

struct tree

{

struct tree \*left;

int info;

struct tree \*right;

}\*rt;

void insrt()

{

int x;

struct tree \*temp,\*root;

root=rt;

temp=(struct tree \*)malloc(sizeof(struct tree));

printf("Enter data:");

scanf("%d",&x);

temp->left=NULL;

temp->right=NULL;

temp->info=x;

if(root==NULL)

{

root=temp;

rt=root;

return;

} while(root!=NULL)

{

if(temp->info>root->info)

{

if(root->right==NULL)

{

root->right=temp;

return;

} else {

root=root->right;

continue;

} if(root->left==NULL)

{

root->left=temp;

return;

} else

{

root=root->left;

continue;

}

} else

{

return;

}

}

} void inorder(struct tree \*root)

{

if(root!=NULL)

{

inorder(root->left);

printf("\t%d",root->info);

inorder(root->right);

} return;

} void preorder(struct tree \*root)

{

if(root!=NULL)

{

printf("\t%d",root->info);

preorder(root->left);

preorder(root->right);

} return;

} void postorder(struct tree \*root)

{

if(root!=NULL)

{

postorder(root->left);

postorder(root->right);

printf("\t%d",root->info);

} return;

}

void delnode()

{

int key;

struct tree \*current,\*prev;

int found;

current=rt;

while(current!=NULL)

{

printf("Enter the key:");

scanf("%d",&key);

if(current->info==key)

{

found=1;

break;

}

else

{

prev=current;

if(current->info>=key)

current=current->left;

else

current=current->right;

}

}

if(current->left==NULL && current->right!=NULL)

{

if(prev->left==current)

prev->left=NULL;

else

prev->right=NULL;

free(current);

}

if(current->right==NULL && current->left!=NULL)

{

if(prev->left==current)

prev->left=current->left;

else

prev->right=current->left;

free(current);

}

if(current->left!=NULL && current->right!=NULL)

{

struct tree \*temp=current->right;

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

{

current->info=temp->info;

free(temp);

current->right=NULL;

}

}

if(current->right->left!=NULL)

{

struct tree \*left\_current=current->right;

struct tree \*left\_current\_prev=current->right->left;

while(left\_current->left!=NULL)

{

left\_current\_prev=left\_current;

left\_current=left\_current->left;

}

current->info=left\_current->info;

free(left\_current);

left\_current\_prev->left=NULL;

}

else

{

struct tree \*temp;

temp=current->right;

current->info=temp->info;

current->right=temp->right;

free(temp);

}

}

int main()

{

int ch;

rt=NULL;

while(1)

{

printf("\n\t1.Insert a node\n\t2.Pre-order traversal\n\t3.In-order traversal\n\t4.Post-order traversal\n\t5.Search a node\n\t6.Delete a node\n\t7.Exit\n");

printf("Enter a choice:");

scanf("%d",&ch);

switch(ch)

{

case 1:insrt();break;

case 2:preorder(rt);break;

case 3:inorder(rt);break;

case 4:postorder(rt);break;

case 6:delnode();break;

case 7:exit(1);break;

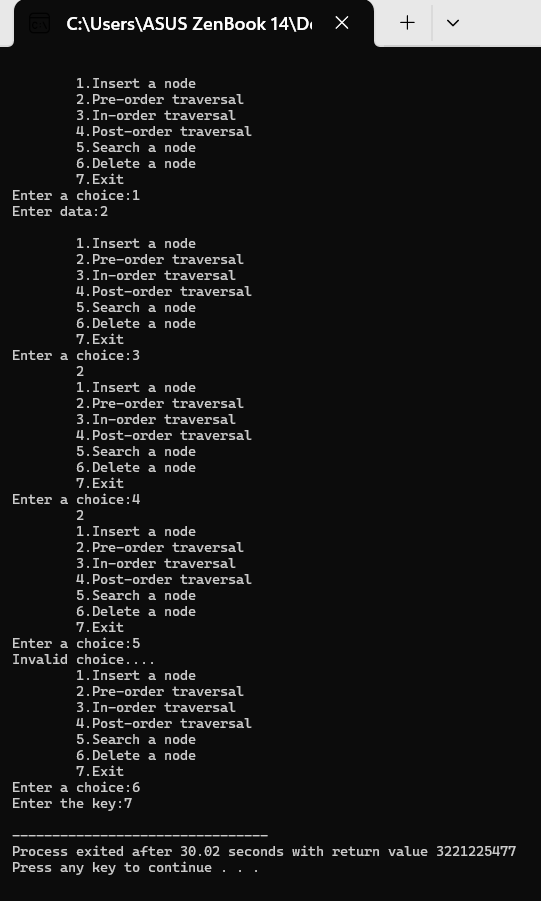
default: printf("Invalid choice....");

}

}getch();

}

**Output**



**Code:-**

#include <stdio.h>

#include <stdlib.h>

struct node {

int data;

struct node \*left;

struct node \*right;

int height;

};

int height(struct node \*N) {

if (N == NULL)

return 0;

return N->height;

}

int max(int a, int b) {

return (a > b) ? a : b;

}

int getBalance(struct node \*N) {

if (N == NULL)

return 0;

return height(N->left) - height(N->right);

}

struct node\* newNode(int data) {

struct node\* node = (struct node\*)malloc(sizeof(struct node));

node->data = data;

node->left = NULL;

node->right = NULL;

node->height = 1;

return node;

}

struct node\* rightRotate(struct node \*y) {

struct node \*x = y->left;

struct node \*T2 = x->right;

// Perform rotation

x->right = y;

y->left = T2;

// Update heights

y->height = max(height(y->left), height(y->right)) + 1;

x->height = max(height(x->left), height(x->right)) + 1;

return x;

}

struct node\* leftRotate(struct node \*x) {

struct node \*y = x->right;

struct node \*T2 = y->left;

// Perform rotation

y->left = x;

x->right = T2;

// Update heights

x->height = max(height(x->left), height(x->right)) + 1;

y->height = max(height(y->left), height(y->right)) + 1;

return y;

}

struct node\* insert(struct node\* node, int data) {

/\* 1. Perform the normal BST insertion \*/

if (node == NULL)

return newNode(data);

if (data < node->data)

node->left = insert(node->left, data);

else if (data > node->data)

node->right = insert(node->right, data);

else // Equal keys are not allowed in AVL tree

return node;

/\* 2. Update height of this ancestor node \*/

node->height = 1 + max(height(node->left), height(node->right));

/\* 3. Get the balance factor of this ancestor node to check whether

this node became unbalanced after insertion \*/

int balance = getBalance(node);

// Left Left Case

if (balance > 1 && data < node->left->data)

return rightRotate(node);

// Right Right Case

if (balance < -1 && data > node->right->data)

return leftRotate(node);

// Left Right Case

if (balance > 1 && data > node->left->data) {

node->left = leftRotate(node->left);

return rightRotate(node);

}

// Right Left Case

if (balance < -1 && data < node->right->data) {

node->right = rightRotate(node->right);

return leftRotate(node);

}

/\* return the (unchanged) node pointer \*/

return node;

}

// A utility function to print the preorder traversal of the tree.

void preOrder(struct node \*root) {

if (root != NULL) {

printf("%d ", root->data);

preOrder(root->left);

preOrder(root->right);

}

}

int main() {

struct node \*root = NULL;

root = insert(root, 10);

root = insert(root, 20);

root = insert(root, 30);

root = insert(root, 40);

root = insert(root, 50);

root = insert(root, 25);

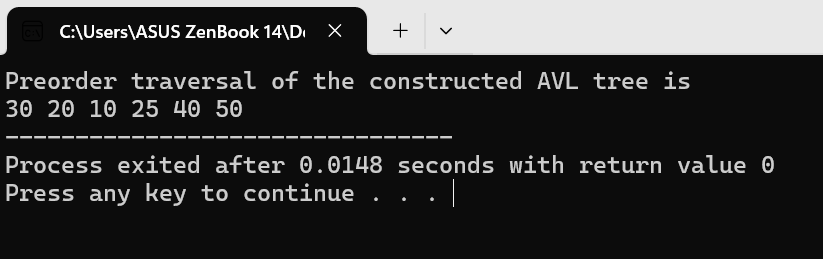
printf("Preorder traversal of the constructed AVL tree is \n");

preOrder(root);

return 0;

}

**Output**



# WAP to implement Merge Sort.

**Theory:-**

Merge sort is a divide-and-conquer sorting algorithm that excels in efficiency and is well-suited for various scenarios. Here's a breakdown of how it works:

**Core principle:**

1. **Divide:** It recursively divides the unsorted list into sub-lists containing a single element each. Since a single element is inherently sorted, this forms the base case for the recursion.
2. **Conquer:** It repeatedly merges these sub-lists to produce new sorted sub-lists until only one sub-list remains. This final sub-list represents the entire sorted list.

**Steps involved:**

1. **Divide:** If the list has more than one element:
   * Find the middle index of the list.
   * Recursively call merge sort on the first half of the list (elements before the middle index).
   * Recursively call merge sort on the second half of the list (elements after the middle index).
2. **Conquer (Merge):** Once both halves are sorted individually, merge them back together into a single sorted list. This merging process compares elements from both halves and inserts the smaller element into the final sorted list. It continues this comparison and insertion until all elements from both halves are incorporated into the final sorted list.

**Code:-**

#include<stdio.h>

#include<conio.h>

#define MAX 100

void merge(int L[],int R[],int A[],int nL,int nR)

{ int i,j,k;

i=j=k=0;

while(i<nL && j<nR)

{ if(L[i]<=R[j]) {

A[k]=L[i];

i++; }

else {

A[k]=R[j];

j++; }

k++; }

while(i<nL) {

A[k]=L[i];

i++;

k++; }

while(j<nR)

{

A[k]=R[j];

j++;

k++; }

} void mergesort(int A[],int n)

{

int i,mid;

if(n<2)

return;

mid=n/2;

int left[mid],right[n-mid];

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

{

left[i]=A[i];

} for(i=mid;i<=(n-1);i++)

{

right[i-mid]=A[i];

} mergesort(left,mid);

mergesort(right,(n-mid));

merge(left,right,A,mid,(n-mid));

} int main()

{

int merge[MAX],i,n;

printf("Enter total no. of elements:");

scanf("%d",&n);

printf("Enter the elements:");

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

{

scanf("%d",&merge[i]);

} mergesort(merge,n);

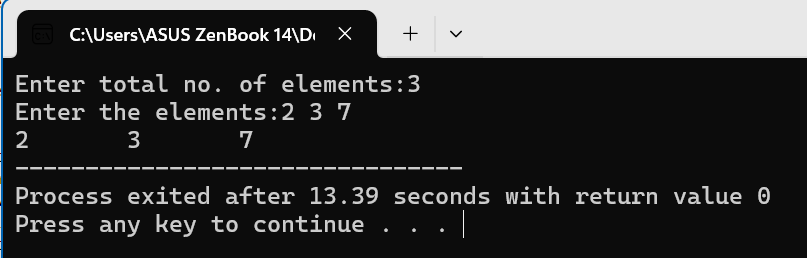
for(i=0;i<n;i++)

{

printf("%d\t",merge[i]);

} return 0; }

**Output**



# WAP to implement searching and hashing algorithms.

**Theory:-**

Searching and hashing algorithms are both fundamental techniques for finding data within a collection. They take different approaches, each with its own strengths and weaknesses.

**Searching Algorithms:**

* **Goal:** Locate a specific item within a dataset.
* **Types:** There are various searching algorithms, each suited for different data structures and scenarios. Here are two common ones:
  + **Linear Search (Sequential Search):** This is a simple approach that iterates through each item in the collection one by one until it finds a match. It's efficient for small datasets but becomes slow for larger ones.
  + **Binary Search:** This method works only on sorted data. It repeatedly halves the search space by comparing the target element with the middle element of the remaining data. It's much faster than linear search for large sorted datasets.

**Hashing Algorithms:**

* **Goal:** Efficiently find (or insert) data based on a key value.
* **Concept:** Hashing uses a data structure called a hash table. This table stores key-value pairs. A hash function takes a key as input and generates a unique index (hash value) within the hash table's range. Ideally, different keys should map to different hash values to avoid collisions (multiple keys ending up at the same index).
* **Benefits:** Hashing offers an average-case time complexity of O(1) for search and insertion operations, making it very fast for large datasets compared to linear search.

**Code:-**

#include <stdio.h>

int sequential\_search(int arr[], int size, int target) {

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

if (arr[i] == target) {

return i; // Return index if found

}

}

return -1; // Return -1 if not found

}

int main() {

int arr[] = {5, 2, 8, 1, 9};

int size = sizeof(arr) / sizeof(arr[0]);

int target = 8;

int index = sequential\_search(arr, size, target);

if (index != -1) {

printf("Element %d found at index %d\n", target, index);

} else {

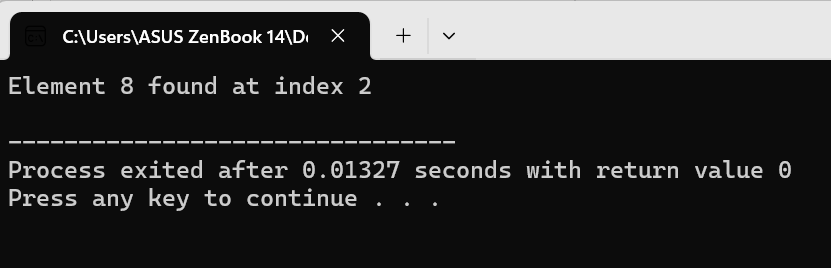
printf("Element %d not found in the array\n", target);

}

return 0;

}

**Output**

****

**Code:-**

#include <stdio.h>

#include <stdlib.h>

#define MAX\_SIZE 10 // Adjust this value for desired hash table size

// Structure to store a key-value pair

struct Item {

int key;

int value;

};

struct Item\* hashTable[MAX\_SIZE]; // Array to store key-value pairs

void initializeHashTable() {

// Initialize all hash table slots to NULL

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

hashTable[i] = NULL;

}

}

int hashFunction(int key) {

// Simple modulo hash function

return key % MAX\_SIZE;

}

void insert(int key, int value) {

int index = hashFunction(key);

// Handle collisions with linear probing

while (hashTable[index] != NULL) {

index = (index + 1) % MAX\_SIZE; // Move to the next slot

}

// Create a new item and insert it at the found index

struct Item\* newItem = (struct Item\*)malloc(sizeof(struct Item));

newItem->key = key;

newItem->value = value;

hashTable[index] = newItem;

}

struct Item\* search(int key) {

int index = hashFunction(key);

int counter = 0; // To prevent infinite loop in case of full table

// Loop until an empty slot or the key is found

while (hashTable[index] != NULL && counter < MAX\_SIZE) {

if (hashTable[index]->key == key) {

return hashTable[index];

}

index = (index + 1) % MAX\_SIZE;

counter++;

}

return NULL; // Not found

}

int main() {

initializeHashTable();

insert(10, 20);

insert(15, 30);

insert(2, 40); // Collision with index of 10, will be placed at 11

struct Item\* item = search(15);

if (item != NULL) {

printf("Item found: Key = %d, Value = %d\n", item->key, item->value);

} else {

printf("Item with key 15 not found\n");

}

item = search(2);

if (item != NULL) {

printf("Item found: Key = %d, Value = %d\n", item->key, item->value);

} else {

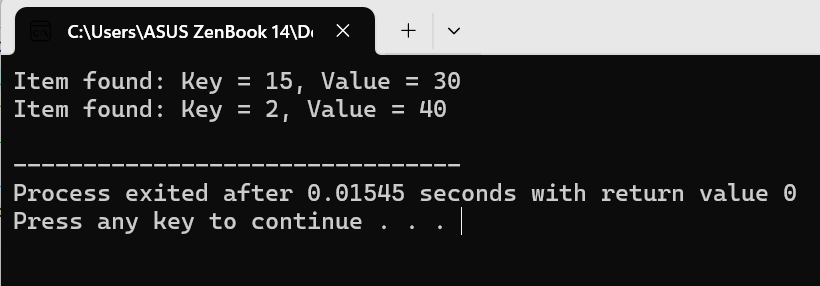
printf("Item with key 2 not found\n");

}

return 0;

}

**Output**



# WAP to implement spanning tree and shortest path.

**Theory:-**

Prim's algorithm is a powerful tool for finding a minimum spanning tree (MST) in a weighted undirected graph. An MST is a subset of edges that connects every vertex in the graph with the total weight of the edges being minimized. Here's a breakdown of how Prim's algorithm works for finding a spanning tree:

1. **Initialization:** Choose any vertex in the graph as the starting point. This can be done randomly. This vertex forms the initial minimum spanning tree.
2. **Iteration:**
   * Identify all edges that connect the existing tree (vertices already included in the MST) to vertices outside the tree. These are called fringe edges.
   * Among the fringe edges, find the one with the minimum weight.
   * Add this edge with the minimum weight to the MST, as long as it doesn't create a cycle (a closed loop) in the tree.
3. **Repeat:** Keep repeating step 2 until all vertices are included in the MST. By then, you'll have explored all possible connections and reached the minimum spanning tree.

Prim's algorithm is a greedy approach, meaning it makes the locally optimal choice at each step (picking the minimum weight edge) with the confidence that it will lead to the globally optimal solution (the MST).

Dijkstra's algorithm is a well-known algorithm for finding the shortest paths between nodes in a weighted graph, particularly for a single source (starting point) to all other nodes. Here's how it works:

1. **Initialization:**
   * Assign a distance value (initially set to positive infinity) to each node in the graph. The distance represents the tentative shortest distance from the source node to that particular node.
   * Mark all nodes as unvisited.
   * Set the distance of the source node to itself as 0 (since the starting point is 0 distance away from itself).
2. **Finding the shortest path:**
   * While there are still unvisited nodes:
     + Identify the unvisited node with the smallest distance value (considered the current node).
     + For all neighbors (adjacent nodes) of the current node:
       - Calculate the tentative distance to the neighbor by adding the weight of the edge connecting them to the current node's distance value.
       - If this tentative distance is less than the neighbor's current distance value, update the neighbor's distance value with this new smaller distance.
     + Mark the current node as visited.
3. **Shortest distances found:**
   * Once all nodes are visited, the algorithm has determined the shortest distances from the source node to all other reachable nodes in the graph.

Dijkstra's algorithm is efficient and guarantees finding the actual shortest paths, making it a valuable tool for various applications like route planning and network analysis.

**Code:-**

#include <stdio.h>

#include <stdbool.h>

#include <stdlib.h>

#define MAX 100

typedef struct Edge {

int dest;

struct Edge \*next; } Edge;

typedef struct Graph {

int V;

Edge \*\*adj;

} Graph;

Graph \*createGraph(int V) {

Graph \*graph = (Graph \*)malloc(sizeof(Graph));

graph->V = V;

graph->adj = (Edge \*\*)malloc(V \* sizeof(Edge \*));

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

graph->adj[i] = NULL; }

return graph; }

void addEdge(Graph \*graph, int src, int dest) {

Edge \*newEdge = (Edge \*)malloc(sizeof(Edge));

newEdge->dest = dest;

newEdge->next = graph->adj[src];

graph->adj[src] = newEdge; }

void DFSUtil(Graph \*graph, int v, bool visited[], int \*parent) {

visited[v] = true;

Edge \*current = graph->adj[v];

while (current!= NULL) {

int i = current->dest;

if (!visited[i]) {

parent[i] = v;

DFSUtil(graph, i, visited, parent); }

current = current->next; } }

int \*DFS(Graph \*graph) {

bool \*visited = (bool \*)calloc(graph->V, sizeof(bool));

int \*parent = (int \*)calloc(graph->V, sizeof(int));

for (int i = 0; i < graph->V; i++) {

if (!visited[i]) {

DFSUtil(graph, i, visited, parent); }

} free(visited);

return parent; }

int main() {

Graph \*graph = createGraph(9);

addEdge(graph, 0, 1);

addEdge(graph, 0, 2);

addEdge(graph, 1, 3);

addEdge(graph, 1, 4);

addEdge(graph, 2, 5);

addEdge(graph, 2, 6);

addEdge(graph, 3, 7);

addEdge(graph, 3, 8);

int \*parent = DFS(graph);

for (int i = 0; i < graph->V; i++) {

printf("Parent of %d is %d\n", i, parent[i]); }

free(parent);

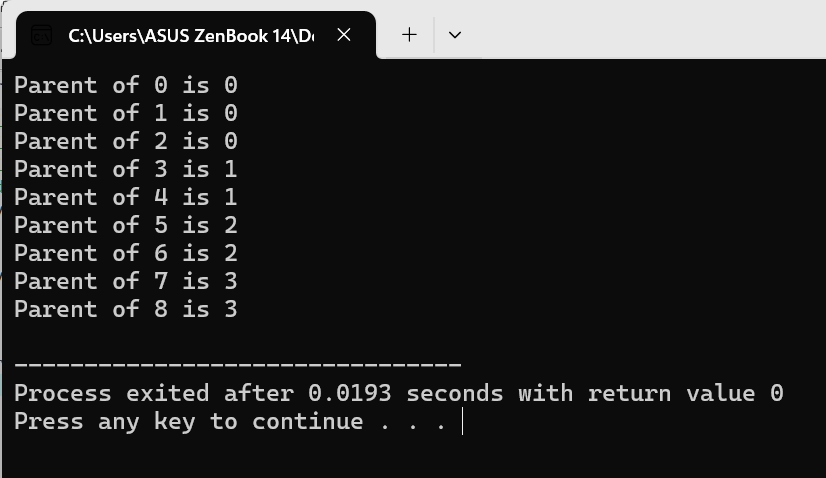
free(graph->adj);

free(graph);

return 0;

}

**Output**



**Code:-**

#include <stdio.h>

#include <stdlib.h>

#include <limits.h>

#define INF INT\_MAX

#define MAX 100

typedef struct Edge {

int dest;

int weight;

struct Edge \*next;

} Edge;

typedef struct Graph {

int V;

Edge \*\*adj;

} Graph;

Graph \*createGraph(int V) {

Graph \*graph = (Graph \*)malloc(sizeof(Graph));

graph->V = V;

graph->adj = (Edge \*\*)malloc(V \* sizeof(Edge \*));

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

graph->adj[i] = NULL; }

return graph; }

void addEdge(Graph \*graph, int src, int dest, int weight) {

Edge \*newEdge = (Edge \*)malloc(sizeof(Edge));

newEdge->dest = dest;

newEdge->weight = weight;

newEdge->next = graph->adj[src];

graph->adj[src] = newEdge; }

void dijkstra(Graph \*graph, int src) {

int \*dist = (int \*)calloc(graph->V, sizeof(int));

int \*parent = (int \*)calloc(graph->V, sizeof(int));

bool \*visited = (bool \*)calloc(graph->V, sizeof(bool));

for (int i = 0; i < graph->V; i++) {

dist[i] = INF;

parent[i] = -1; }

dist[src] = 0;

for (int count = 0; count < graph->V - 1; count++) {

int u = -1;

for (int i = 0; i < graph->V; i++) {

if (!visited[i] && dist[i] < INF) {

u = i;} }

if (u == -1) {

break; }

visited[u] = true;

Edge \*current = graph->adj[u];

while (current!= NULL) {

int v = current->dest;

if (!visited[v] && dist[u] + current->weight < dist[v]) {

dist[v] = dist[u] + current->weight;

parent[v] = u; }

current = current->next; } }

for (int i = 0; i < graph->V; i++) {

if (parent[i]!= -1) {

printf("Shortest path from %d to %d is %d with cost %d\n", src, i, parent[i], dist[i]); }

} free(dist);

free(parent);

free(visited);

} int main() {

Graph \*graph = createGraph(9);

addEdge(graph, 0, 1, 4);

addEdge(graph, 0, 7, 8);

addEdge(graph, 1, 2, 8);

addEdge(graph, 1, 7, 11);

addEdge(graph, 2, 3, 7);

addEdge(graph, 2, 5, 4);

addEdge(graph, 2, 8, 2);

addEdge(graph, 3, 4, 9);

addEdge(graph, 3, 5, 14);

addEdge(graph, 4, 5, 10);

addEdge(graph, 5, 6, 2);

addEdge(graph, 6, 7, 1);

addEdge(graph, 6, 8, 6);

addEdge(graph, 7, 8, 7);

dijkstra(graph, 0);

free(graph->adj);

free(graph);

return 0;

}

**Output**

