**DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING**

**DATA STRUCTURES AND ALGORITHM**



**CS 104**

**LAB FILE**

**SUBMITTED TO**

**SUBMITTED BY:**

**MR. JATIN SHARMA**

**NIKUNJ SHARMA**

**23/CS/491**

**INDEX**

|  |  |  |  |
| --- | --- | --- | --- |
| **SNO.** | **OBJECTIVE** | **DATE** | **SIGN** |
|  |  |  |  |
| **1.** | **Write a program to Implement Linear Search in the** | **23.02.24** |  |
|  | **C/C++ programming language.** |  |  |
|  |  |  |  |
| **2.** | **Write a program to Implement Binary Search in the** | **23.02.24** |  |
|  | **C/C++ programming language. Assume the list is** |  |  |
|  | **already sorted.** |  |  |
|  |  |  |  |
| **3.** | **Write a program to insert an element at the** | **23.02.24** |  |
|  | **mid-position in the One-dimensional array.** |  |  |
|  |  |  |  |
| **4.** | **Write a program to delete a given row in the** | **23.02.24** |  |
|  | **two-dimensional array.** |  |  |
|  |  |  |  |
| **5.** | **Write a program to implement a stack data** | **23.02.24** |  |
|  | **structure and perform its operations.** |  |  |
|  |  |  |  |
| **6.** | **Write a program to implement two stacks using a** | **23.02.24** |  |
|  | **single array.** |  |  |
|  |  |  |  |
| **7.** | **Write a program to reverse a 5-digit number** | **01.03.24** |  |
|  |  |  |  |
| **8.** | **Write a program to convert decimal to binary and** | **01.03.24** |  |
|  | **vice versa.** |  |  |
|  |  |  |  |
| **9.** | **Write a program to find the minimum element of** | **05.04.24** |  |
|  | **the stack in constant time using extra space.** |  |  |
|  |  |  |  |
| **10.** | **Write a program to find the minimum element of** | **05.04.24** |  |
|  | **the stack in constant time without using extra space.** |  |  |
|  |  |  |  |
| **11.** | **Write a program to implement Queue Data** | **05.04.24** |  |
|  | **Structure.** |  |  |
|  |  |  |  |
| **12.** | **Write a program to reverse the first k elements of a** | **05.04.24** |  |
|  | **given Queue.** |  |  |
|  |  |  |  |

|  |  |  |  |
| --- | --- | --- | --- |
| **13.** | **Write a program to implement the Linked List Data** | **05.04.24** |  |
|  | **structure and insert a new node at the beginning,** |  |  |
|  | **and at a given position.** |  |  |
|  |  |  |  |
| **14.** | **Write a Program to check whether the given tree is** | **05.04.24** |  |
|  | **a Binary Search Tree or not.** |  |  |
|  |  |  |  |
| **15.** | **Write a Program to check whether the given tree is** | **05.04.24** |  |
|  | **a Binary Search Tree or not.** |  |  |
|  |  |  |  |

**EXPERIMENT-1**

**AIM -** Write a program to Implement Linear Search in the C/C++ programming language.

**ALGORITHM -**

1. Start from the first element of the array.
2. Compare the target element with each element of the array sequentially.
3. If the target element is found, return the index.
4. If the target element is not found after iterating through the entire array, return -1.

**CODE -**

#include <stdio.h>

int linearSearch(int arr[], int n, int target) { for (int i = 0; i < n; i++) {

if (arr[i] == target)

return i;}

return -1; }

int main() {

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

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

int target = 30;

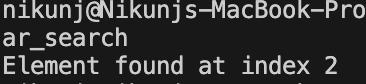
int result = linearSearch(arr, n, target); if (result != -1)

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

printf("Element not found\n");

return 0;}

**OUTPUT -**



**EXPERIMENT-2**

**AIM -** Write a program to Implement Binary Search in the C/C++ programming language. Assume the list is already sorted.

**ALGORITHM -**

1. Start with the middle element of the sorted array.
2. If the middle element is the target, return its index.
3. If the target is less than the middle element, repeat the search on the left half of the array.
4. If the target is greater than the middle element, repeat the search on the right half of the array.
5. Continue this process until the target is found or the search space is exhausted.

**CODE -**

#include <stdio.h>

int binarySearch(int arr[], int left, int right, int target) { while (left <= right) {

int mid = left + (right - left) / 2;

if (arr[mid] == target)

return mid;

else if (arr[mid] < target)

left = mid + 1;

else

right = mid - 1;

}

return -1;

}

int main() {

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

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

int target = 40;

int result = binarySearch(arr, 0, n - 1, target); if (result != -1)

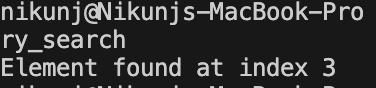
printf("Element found at index %d\n", result); else

printf("Element not found\n");

return 0;

}

**OUTPUT -**



**EXPERIMENT-3**

**AIM -** Write a program to insert an element at the mid-position in the One-dimensional array.

**ALGORITHM -**

1. Calculate the mid-position of the array.
2. Shift elements from the mid-position to the end of the array one position to the right.
3. Insert the new element at the mid-position.

**CODE -** #include <stdio.h>

void insertAtMid(int arr[], int \*size, int element) { int mid = \*size / 2;

for (int i = \*size - 1; i >= mid; i--) { arr[i + 1] = arr[i];}

arr[mid] = element;

(\*size)++;}

int main() {

int arr[10] = {1, 2, 3, 4, 5};

int size = 5;

int element = 10;

printf("Array before insertion: ");

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

printf("%d ", arr[i]);}

printf("\n");

insertAtMid(arr, &size, element);

printf("Array after insertion: ");

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

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

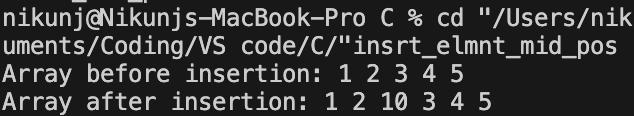
}

printf("\n");

return 0;

}

**OUTPUT -**



**EXPERIMENT-4**

**AIM -** Write a program to delete a given row in the two-dimensional array.

**ALGORITHM -**

1. Move all rows below the deleted row one position up.
2. Decrement the total number of rows.

**CODE -**

#include <stdio.h>

void deleteRow(int arr[][3], int \*rows, int rowToDelete) { for (int i = rowToDelete; i < \*rows - 1; i++) {

for (int j = 0; j < 3; j++) {

arr[i][j] = arr[i + 1][j];

}

}

(\*rows)--;

}

int main() {

int arr[5][3] = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}, {10, 11, 12}, {13, 14, 15}}; int rows = 5;

int rowToDelete = 2;

printf("Array before deletion:\n");

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

for (int j = 0; j < 3; j++) {

printf("%d ", arr[i][j]);

}

printf("\n");

}

deleteRow(arr, &rows, rowToDelete);

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

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

for (int j = 0; j < 3; j++) {

printf("%d ", arr[i][j]);

}

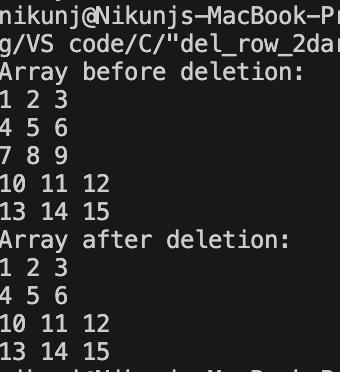
printf("\n");

}

return 0;

}

**OUTPUT -**



**EXPERIMENT-5**

**AIM -** Write a program to implement a stack data structure and perform its operations.

**ALGORITHM -**

1. Initialize a stack data structure.
2. Implement push operation to insert elements onto the stack.
3. Implement pop operation to remove elements from the stack.
4. Implement peek operation to view the top element of the stack.
5. Implement isEmpty operation to check if the stack is empty.
6. Implement isFull operation to check if the stack is full.

**CODE -**

#include <stdio.h>

#include <stdbool.h>

#define MAX\_SIZE 100

typedef struct {

int data[MAX\_SIZE];

int top;

} Stack;

void initStack(Stack \*s) {

s->top = -1;}

bool isEmpty(Stack \*s) {

return s->top == -1;}

bool isFull(Stack \*s) {

return s->top == MAX\_SIZE - 1;}

void push(Stack \*s, int element) {

if (!isFull(s)) {

s->data[++s->top] = element;

printf("%d pushed to stack.\n", element);

} else {

printf("Stack overflow! Unable to push %d\n", element);

}

}

int pop(Stack \*s) {

if (!isEmpty(s)) {

return s->data[s->top--];

} else {

printf("Stack underflow! Unable to pop.\n"); return -1;

}

}

int peek(Stack \*s) {

if (!isEmpty(s)) {

return s->data[s->top];

} else {

printf("Stack is empty!\n");

return -1;

}

}

int main() {

Stack s;

initStack(&s);

push(&s, 10);

push(&s, 20);

push(&s, 30);

printf("Top element: %d\n", peek(&s));

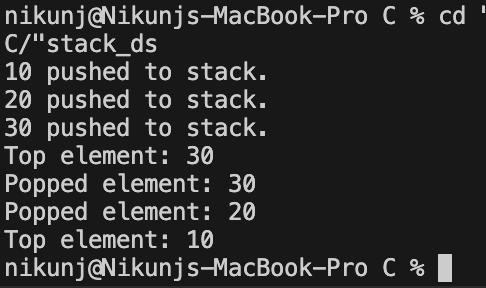
printf("Popped element: %d\n", pop(&s));

printf("Popped element: %d\n", pop(&s));

printf("Top element: %d\n", peek(&s)); return 0;

}

**OUTPUT -**



**EXPERIMENT-6**

**AIM -** Write a program to implement two stacks using a single array.

**ALGORITHM -**

1. Divide the array into two halves to represent two stacks.
2. Implement push and pop operations for each stack separately.
3. Keep track of the top index of each stack

**CODE -**

#include <stdio.h>

#include <stdbool.h>

#define MAX\_SIZE 100

typedef struct {

int data[MAX\_SIZE];

int top1;

int top2;

} TwoStacks;

void initTwoStacks(TwoStacks \*ts) { ts->top1 = -1; // Top of stack 1 ts->top2 = MAX\_SIZE; // Top of stack 2

}

bool isFull(TwoStacks \*ts) {

return ts->top1 == ts->top2 - 1;

}

bool isEmpty1(TwoStacks \*ts) {

return ts->top1 == -1;

}

bool isEmpty2(TwoStacks \*ts) {

return ts->top2 == MAX\_SIZE;

}

void push1(TwoStacks \*ts, int element) { if (!isFull(ts)) {

ts->data[++ts->top1] = element; printf("%d pushed to stack 1.\n", element);

} else {

printf("Stack 1 overflow! Unable to push %d\n", element);

}

}

void push2(TwoStacks \*ts, int element) { if (!isFull(ts)) {

ts->data[--ts->top2] = element; printf("%d pushed to stack 2.\n", element);

} else {

printf("Stack 2 overflow! Unable to push %d\n", element);

}

}\

int pop1(TwoStacks \*ts) {

if (!isEmpty1(ts)) {

return ts->data[ts->top1--];

} else {

printf("Stack 1 underflow! Unable to pop.\n"); return -1;

}

}

int pop2(TwoStacks \*ts) {

if (!isEmpty2(ts)) {

return ts->data[ts->top2++];

} else {

printf("Stack 2 underflow! Unable to pop.\n"); return -1;

}

}

int main() {

TwoStacks ts;

initTwoStacks(&ts);

push1(&ts, 10);

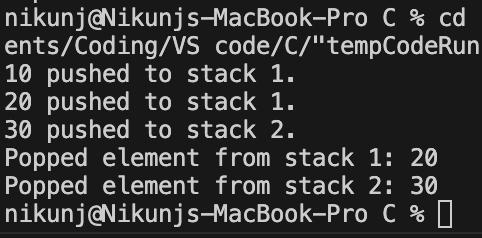
push1(&ts, 20);

push2(&ts, 30);

printf("Popped element from stack 1: %d\n", pop1(&ts)); printf("Popped element from stack 2: %d\n", pop2(&ts)); return 0;

}

**OUTPUT -**



**EXPERIMENT-7**

**AIM -** Write a program to reverse a 5-digit number

**ALGORITHM -**

1. Extract each digit of the 5-digit number iteratively.
2. Reconstruct the reversed number by appending the digits in reverse order.
3. Display the reversed number.

**CODE -**

#include <stdio.h>

int reverseNumber(int num) {

int reversed = 0;

while (num > 0) {

reversed = reversed \* 10 + num % 10; num /= 10;

}

return reversed;

}

int main() {

int number;

printf("Enter a 5-digit number: ");

scanf("%d", &number);

if (number >= 10000 && number <= 99999) { int reversedNumber = reverseNumber(number); printf("Reversed number: %d\n", reversedNumber);

} else {

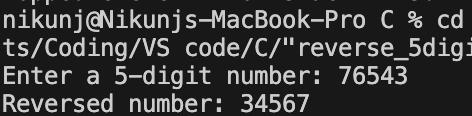
printf("Invalid input! Please enter a 5-digit number.\n");

}

return 0;

}

**OUTPUT -**



**EXPERIMENT-8**

**AIM -** Write a program to convert decimal to binary and vice versa.

**ALGORITHM -**

* Algorithm (Decimal to Binary):

1. Initialize variables to hold binary number and remainder.
2. Perform repeated division of the decimal number by 2.
3. Record the remainders to obtain the binary equivalent.
4. Reverse the binary equivalent to get the final binary number.
5. Display the binary number.

* Algorithm (Binary to Decimal):

1. Initialize variables to hold decimal number, base, and remainder.
2. Perform the conversion by multiplying each binary digit by powers of 2.
3. Sum the results to obtain the decimal equivalent.
4. Display the decimal number.

**CODE -**

* **Decimal to Binary**

#include <stdio.h>

long decimalToBinary(int decimal) {

long binary = 0;

int remainder, base = 1;

while (decimal > 0) {

remainder = decimal % 2;

binary += remainder \* base;

decimal /= 2;

base \*= 10;

}

return binary;

}

int main() {

int decimalNumber;

printf("Enter a decimal number: "); scanf("%d", &decimalNumber);

long binaryNumber = decimalToBinary(decimalNumber); printf("Binary equivalent: %ld\n", binaryNumber);

return 0;

}

* **Binary to Decimal**

#include <stdio.h>

int binaryToDecimal(long binary) { int decimal = 0, base = 1, remainder; while (binary > 0) {

remainder = binary % 10;

decimal += remainder \* base;

binary /= 10;

base \*= 2;

}

return decimal;

}

int main() {

long binaryInput;

printf("Enter a binary number: ");

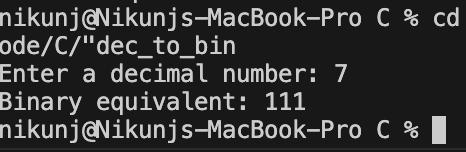
scanf("%ld", &binaryInput);

int decimalResult = binaryToDecimal(binaryInput); printf("Decimal equivalent: %d\n", decimalResult); return 0;

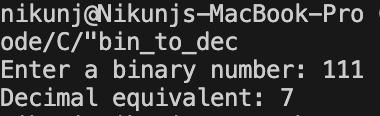
}

**OUTPUT -**

* **Decimal to Binary**



* **Binary to Decimal**



**EXPERIMENT-9**

**AIM -** Write a program to find the minimum element of the stack in constant time using extra space.

**ALGORITHM -**

1. Maintain an auxiliary stack to keep track of the minimum elements.
2. Push each element onto the auxiliary stack while maintaining the minimum element.
3. For each push operation, if the element is smaller than or equal to the current minimum, push it onto the auxiliary stack.
4. For each pop operation, if the popped element is equal to the current minimum, pop from the auxiliary stack as well.
5. The top of the auxiliary stack always contains the minimum element of the stack.

**CODE -**

#include <stdio.h>

#include <stdlib.h>

#define MAX\_SIZE 100

typedef struct {

int data[MAX\_SIZE];

int top;

int minStack[MAX\_SIZE];

int minTop;

} Stack;

void initStack(Stack \*s) {

s->top = -1;

s->minTop = -1;

}

void push(Stack \*s, int element) {

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

printf("Stack overflow!\n");

return;

}

s->top++;

s->data[s->top] = element;

if (s->minTop == -1 || element <= s->minStack[s->minTop]) { s->minTop++;

s->minStack[s->minTop] = element;

}

}

int pop(Stack \*s) {

if (s->top < 0) {

printf("Stack underflow!\n");

return -1;

}

int popped = s->data[s->top];

s->top--;

if (popped == s->minStack[s->minTop])

s->minTop--;

return popped;

}

int getMin(Stack \*s) {

if (s->minTop >= 0)

return s->minStack[s->minTop];

else {

printf("Stack is empty!\n");

return -1;

}

}

int main() {

Stack s;

initStack(&s);

push(&s, 10);

push(&s, 5);

push(&s, 15);

push(&s, 3);

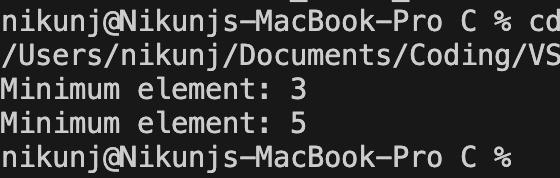
printf("Minimum element: %d\n", getMin(&s)); // Should print 3

pop(&s); // Pop the top element

pop(&s); // Pop the top element

printf("Minimum element: %d\n", getMin(&s)); // Should print 5 return 0;}

**OUTPUT -**



**EXPERIMENT-10**

**AIM -** Write a program to find the minimum element of the stack in constant time without using extra space.

**ALGORITHM -**

1. Maintain a variable to store the current minimum element.
2. For each push operation, compare the incoming element with the current minimum.
3. If the incoming element is smaller than the current minimum, update the current minimum.
4. For each pop operation, if the popped element is equal to the current minimum, update the current minimum by traversing the stack.

**CODE -**

#include <stdio.h>

#include <stdlib.h>

#include <limits.h> // Include this header for INT\_MAX

#define MAX\_SIZE 100

typedef struct {

int data[MAX\_SIZE];

int top;

int minElement;

} Stack;

void initStack(Stack \*s) {

s->top = -1;

s->minElement = INT\_MAX;

}

void push(Stack \*s, int element) {

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

printf("Stack overflow!\n");

return;

}

if (element < s->minElement) { s->data[++s->top] = s->minElement; s->minElement = element;

}

s->data[++s->top] = element;

}

int pop(Stack \*s) {

if (s->top < 0) {

printf("Stack underflow!\n");

return -1;

}

int popped = s->data[s->top--];

if (popped == s->minElement)

s->minElement = s->data[s->top--]; return popped;

}

int getMin(Stack \*s) {

if (s->top >= 0)

return s->minElement;

else {

printf("Stack is empty!\n");

return -1;

}

}

int main() {

Stack s;

initStack(&s);

push(&s, 10);

push(&s, 5);

push(&s, 15);

push(&s, 3);

printf("Minimum element: %d\n", getMin(&s)); // Should print 3

pop(&s); // Pop the top element

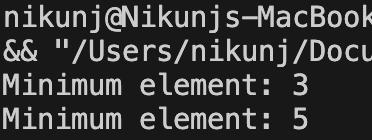
pop(&s); // Pop the top element

printf("Minimum element: %d\n", getMin(&s)); // Should print 5

return 0;

}

**OUTPUT -**



**EXPERIMENT-11**

**AIM -** Write a program to implement Queue Data Structure.

**ALGORITHM -**

1. Maintain a fixed-size array to store the elements of the queue.
2. Maintain two pointers, 'front' and 'rear', to keep track of the front and rear of the queue.
3. Implement enqueue operation to insert elements at the rear of the queue.
4. Implement dequeue operation to remove elements from the front of the queue.
5. Implement isEmpty operation to check if the queue is empty.
6. Implement isFull operation to check if the queue is full.

**CODE -**

#include <stdio.h>

#include <stdbool.h>

#define MAX\_SIZE 100

typedef struct {

int data[MAX\_SIZE];

int front;

int rear;

} Queue;

void initQueue(Queue \*q) {

q->front = -1;

q->rear = -1;

}

bool isEmpty(Queue \*q) {

return q->front == -1;

}

bool isFull(Queue \*q) {

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

}

void enqueue(Queue \*q, int element) { if (isFull(q)) {

printf("Queue is full! Unable to enqueue %d\n", element); return;

}

if (isEmpty(q))

q->front = 0;

q->rear = (q->rear + 1) % MAX\_SIZE;

q->data[q->rear] = element;

printf("%d enqueued to the queue.\n", element);

}

int dequeue(Queue \*q) {

if (isEmpty(q)) {

printf("Queue is empty! Unable to dequeue.\n"); return -1;

}

int removed = q->data[q->front];

if (q->front == q->rear) {

q->front = -1;

q->rear = -1;

} else {

q->front = (q->front + 1) % MAX\_SIZE;

}

return removed;

}

int main() {

Queue q;

initQueue(&q);

enqueue(&q, 10);

enqueue(&q, 20);

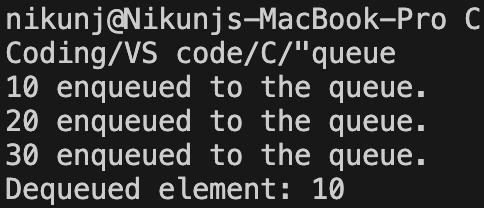
enqueue(&q, 30);

printf("Dequeued element: %d\n", dequeue(&q)); // Should print 10

return 0;

}

**OUTPUT -**



**EXPERIMENT-12**

**AIM -** Write a program to reverse the first k elements of a given Queue.

**ALGORITHM -**

1. Dequeue the first k elements from the queue and push them onto a stack.
2. Pop the elements from the stack and enqueue them back into the queue.
3. Dequeue the remaining elements from the queue and enqueue them back into the queue in their original order.

**CODE -**

#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>

#define MAX\_SIZE 100

typedef struct {

int data[MAX\_SIZE];

int front;

int rear;

} Queue;

void initQueue(Queue \*q) {

q->front = -1;

q->rear = -1;

}

bool isEmpty(Queue \*q) {

return q->front == -1;

}

bool isFull(Queue \*q) {

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

}

void enqueue(Queue \*q, int element) { if (isFull(q)) {

printf("Queue is full! Unable to enqueue %d\n", element); return;

}

if (isEmpty(q))

q->front = 0;

q->rear = (q->rear + 1) % MAX\_SIZE;

q->data[q->rear] = element;

printf("%d enqueued to the queue.\n", element);

}

int dequeue(Queue \*q) {

if (isEmpty(q)) {

printf("Queue is empty! Unable to dequeue.\n"); return -1;

}

int removed = q->data[q->front];

if (q->front == q->rear) {

q->front = -1;

q->rear = -1;

} else {

q->front = (q->front + 1) % MAX\_SIZE;

}

return removed;

}

void reverseFirstK(Queue \*q, int k) {

if (isEmpty(q) || k <= 0) {

printf("Invalid operation!\n");

return;

}

if (k > (q->rear - q->front + 1)) {

printf("Queue does not have enough elements to reverse!\n"); return;

}

int stack[MAX\_SIZE];

int top = -1;

* Dequeue
* Dequeue the first k elements and push them onto the stack for (int i = 0; i < k; i++) {

stack[++top] = dequeue(q);

}

* Enqueue the elements from the stack back into the queue for (int i = 0; i < k; i++) {

enqueue(q, stack[top--]);}

* Dequeue the remaining elements and enqueue them back into the queue for (int i = 0; i < (q->rear - q->front + 1 - k); i++) {

enqueue(q, dequeue(q));}}

int main() {

Queue q;

initQueue(&q);

enqueue(&q, 10);

enqueue(&q, 20);

enqueue(&q, 30);

enqueue(&q, 40);

enqueue(&q, 50);

printf("Queue before reversing first 3 elements:\n"); while (!isEmpty(&q)) {

printf("%d ", dequeue(&q));}

printf("\n");

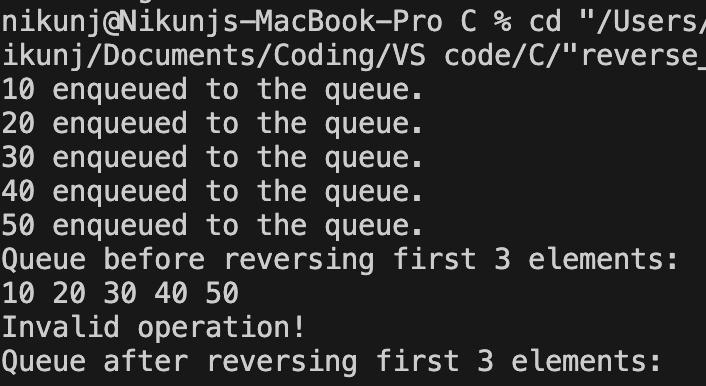
* Reverse the first 3 elements reverseFirstK(&q, 3);

printf("Queue after reversing first 3 elements:\n"); while (!isEmpty(&q)) {

printf("%d ", dequeue(&q));} printf("\n");

return 0;}

**OUTPUT -**



**EXPERIMENT-13**

**AIM -** Write a program to implement the Linked List Data structure and insert a new node at the beginning, and at a given position.

**ALGORITHM -**

1. Define a structure to represent a node in the linked list.
2. Implement functions to create a new node, insert a new node at the beginning of the list, and insert a new node at a given position in the list.

**CODE -**

#include <stdio.h>

#include <stdlib.h>

typedef struct Node {

int data;

struct Node \*next;

} Node;

Node \*createNode(int data) {

Node \*newNode = (Node \*)malloc(sizeof(Node)); if (newNode == NULL) {

printf("Memory allocation failed!\n"); exit(EXIT\_FAILURE);

}

newNode->data = data;

newNode->next = NULL;

return newNode;

}

void insertAtBeginning(Node \*\*head, int data) { Node \*newNode = createNode(data); newNode->next = \*head;

\*head = newNode;

}

void insertAtPosition(Node \*\*head, int data, int position) { if (position < 1) {

printf("Invalid position!\n");

return;

}

if (position == 1) {

insertAtBeginning(head, data);

return;

}

Node \*newNode = createNode(data); Node \*temp = \*head;

for (int i = 1; i < position - 1 && temp != NULL; i++) { temp = temp->next;

}

if (temp == NULL) { printf("Position out of range!\n"); return;

}

newNode->next = temp->next;

temp->next = newNode;

}

void printList(Node \*head) {

Node \*temp = head;

while (temp != NULL) {

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

temp = temp->next;

}

printf("\n");

}

int main() {

Node \*head = NULL;

* Insert at the beginning insertAtBeginning(&head, 10); insertAtBeginning(&head, 20); insertAtBeginning(&head, 30); printf("List after inserting at the beginning: "); printList(head);
* Insert at position

insertAtPosition(&head, 40, 2);

insertAtPosition(&head, 50, 4);

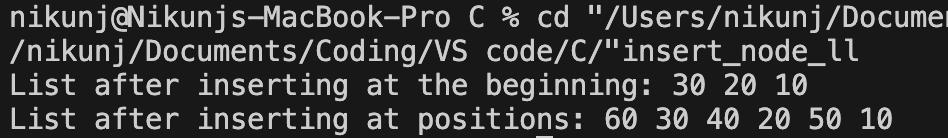
insertAtPosition(&head, 60, 1);

printf("List after inserting at positions: ");

printList(head);

return 0;}

**OUTPUT -**



**EXPERIMENT-14**

**AIM -** Write a Program to check whether the given tree is a Binary Search Tree or not.

**ALGORITHM -**

1. For each node in the tree, check if its left child is less than the node's value and its right child is greater than the node's value.
2. Recursively apply the above condition to all nodes in the tree.
3. If the above condition holds true for all nodes, then the tree is a Binary Search Tree.

**CODE -**

#include <stdio.h>

#include <stdbool.h>

#include <limits.h>

#include <stdlib.h> // Added this line

typedef struct TreeNode {

int data;

struct TreeNode \*left;

struct TreeNode \*right;

} TreeNode;

TreeNode \*createNode(int data) {

TreeNode \*newNode = (TreeNode \*)malloc(sizeof(TreeNode)); if (newNode == NULL) {

printf("Memory allocation failed!\n"); exit(EXIT\_FAILURE);

}

newNode->data = data;

newNode->left = NULL;

newNode->right = NULL;

return newNode;

}

bool isBSTUtil(TreeNode \*root, int min, int max) { if (root == NULL)

return true;

if (root->data < min || root->data > max)

return false;

return isBSTUtil(root->left, min, root->data - 1) && isBSTUtil(root->right, root->data + 1, max);

}

bool isBinarySearchTree(TreeNode \*root) { return isBSTUtil(root, INT\_MIN, INT\_MAX);

}

int main() {

TreeNode \*root = createNode(10); root->left = createNode(5); root->right = createNode(15); root->left->left = createNode(3); root->left->right = createNode(7); root->right->left = createNode(12); root->right->right = createNode(18);

if (isBinarySearchTree(root))

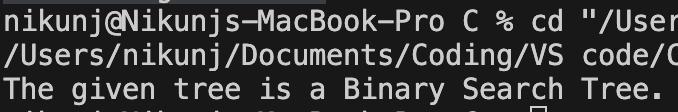
printf("The given tree is a Binary Search Tree.\n"); else

printf("The given tree is not a Binary Search Tree.\n");

return 0;

}

**OUTPUT -**



**EXPERIMENT-15**

**AIM -** Write an Program to count the number of leaf nodes in an AVL tree.

**ALGORITHM -**

1. Traverse the AVL tree in a Depth First Search (DFS) manner.
2. At each node, check if it is a leaf node (i.e., it has no left and right child).
3. Increment the leaf node count whenever a leaf node is encountered.

**CODE -**

#include <stdio.h>

#include <stdlib.h>

typedef struct AVLNode {

int data;

struct AVLNode \*left;

struct AVLNode \*right;

int height;

} AVLNode;

AVLNode \*createNode(int data) {

AVLNode \*newNode = (AVLNode \*)malloc(sizeof(AVLNode)); if (newNode == NULL) {

printf("Memory allocation failed!\n"); exit(EXIT\_FAILURE);

}

newNode->data = data;

newNode->left = NULL;

newNode->right = NULL;

newNode->height = 1;

return newNode;

}

int countLeafNodes(AVLNode \*root) { if (root == NULL)

return 0;

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

return 1;

return countLeafNodes(root->left) + countLeafNodes(root->right);

}

int main() {

AVLNode \*root = createNode(10); root->left = createNode(5); root->right = createNode(15); root->left->left = createNode(3); root->left->right = createNode(7); root->right->left = createNode(12); root->right->right = createNode(18); printf("Number of leaf nodes in the AVL tree: %d\n",

countLeafNodes(root));

return 0;

}

**OUTPUT -**

