

**NMIMS Mumbai**

# Data Structure and Algorithm

**Submitted By Name: Ashmit Jain SAP ID: 70322100206**

**Mukesh Patel School of Technology Management & Engineering Bhakthi Vendanth Swami Marg, Near Cooper Hospital, JVPD Scheme, Vile Parle**

**West, Mumbai, Maharashtra 400056**



# Certificate

Name: Ashmit Jain

Class: B1

Roll Number: C013

SAP ID: 70322100206

Program: B-tech Integrated

This is to certify the work of the student in Data Structure and Algorithms subject during the academic year 2024-25 is successfully completed.

**Examiner’s Sign**

**Date: 4/11/2024**

|  |  |  |  |
| --- | --- | --- | --- |
| Sr. No. | Title | CO | Date |
| 1 | Implementation of various array operations like traversal, insertion and deletion using any real life application. | CO1 |  |
| 2. | To study and implement concept of Stack data structure and use in recursion | CO2 |  |
| 3. | Application of stack (Expression conversion/evaluation of postfix expression /Parenthesis matching ) | CO2 |  |
| 4 | Implementation of simple queues and circular queue using arrays and apply them in real life computer applications -  Railway reservation system | CO 2 |  |
| 5 | To study and implement the concept of Linked list data structure. | CO2 |  |
| 6 | To study and implement stack and queue data structure using concept of linked list | CO2 |  |
| 7 | Representation of Binary trees in memory and implementation of Binary Tree traversal algorithms | CO 3 |  |
| 8 | Implementation of Binary Search Tree: Insertion, deletion and Search operation on tree data structure | CO3 |  |
| 9 | Implementation of Graph Traversal: Breadth first search, Depth first search | CO3 |  |
| 10 | Implementation of Linear search and binary search, Insertion Sort and Merge Sort understand their difference | CO4 |  |
| 11 | Mini Project Documentation | CO1,2,3,4 |  |
| 12 | Mid Test 1 | CO1,2 |  |
| 13 | Mid Test II | CO2,3 |  |

**Experiment 1**

(PART B: TO BE COMPLETED BY STUDENTS)

**(Students must submit the soft copy as per following segments within two hours of the practical. The soft copy must be uploaded on the Blackboard or emailed to the concerned lab in charge faculties at the end of the practical in case the there is no Black board access available)**

|  |  |
| --- | --- |
| Roll No. C013 | Name: Ashmit Jain |
| Program : BTI | Division: B1 |
| Batch: 1 | Date of Experiment: |
| Date of Submission: 8/6/24 | Grade : |

### ●.1 Tasks given in PART A to be completed here

*(****Students must write the answers of the task(s) given in the PART A )***

**#include <iostream> #include <string> using namespace std;**

**class Library { private:**

**string bookName; float bookCost; bool issuedStatus; int serialNumber;**

**public:**

**// Default constructor to initialize values**

**Library() : bookName(""), bookCost(0.0), issuedStatus(false), serialNumber(0) {}**

**// Parameterized constructor**

**Library(string name, float cost, bool status, int serial)**

**: bookName(name), bookCost(cost), issuedStatus(status), serialNumber(serial) {}**

**// Function to input book details void inputBookDetails() {**

**cout << "Enter book name: "; cin.ignore();**

**getline(cin, bookName); cout << "Enter book cost: "; cin >> bookCost;**

**cout << "Is the book issued? (1 for Yes, 0 for No): "; cin >> issuedStatus;**

**cout << "Enter serial number: "; cin >> serialNumber;**

**}**

**// Function to display book details void displayBookDetails() const {**

**cout << "Book Name: " << bookName << endl; cout << "Book Cost: " << bookCost << endl;**

**cout << "Issued Status: " << (issuedStatus ? "Issued" : "Not Issued") << endl;**

**cout << "Serial Number: " << serialNumber << endl;**

**}**

**// Function to display all books**

**static void displayAllBooks(Library books[], int size) { for (int i = 0; i < size; ++i) {**

**cout << "\nBook " << i + 1 << " details:\n"; books[i].displayBookDetails();**

**}**

**}**

**// Function to display a specific book by serial number static void displaySpecificBook(Library books[], int size) {**

**int serial;**

**cout << "Enter serial number: "; cin >> serial;**

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

**if (books[i].serialNumber == serial) { books[i].displayBookDetails(); return;**

**}**

**}**

**cout << "Book not found.\n";**

**}**

**// Function to insert a book at a specified position**

**static void insertAtPosition(Library books[], int &size, int pos, Library newBook) {**

**for (int i = size; i > pos; --i) { books[i] = books[i - 1];**

**}**

**books[pos] = newBook;**

**++size;**

**}**

**// Function to insert a book at the start**

**static void insertAtStart(Library books[], int &size, Library newBook) {**

**insertAtPosition(books, size, 0, newBook);**

**}**

**// Function to insert a book at the end**

**static void insertAtEnd(Library books[], int &size, Library newBook)**

**{**

**books[size] = newBook;**

**++size;**

**}**

**// Function to delete a book at the start**

**static void deleteAtStart(Library books[], int &size) { for (int i = 0; i < size - 1; ++i) {**

**books[i] = books[i + 1];**

**}**

**--size;**

**}**

**// Function to delete a book at the end**

**static void deleteAtEnd(Library books[], int &size) {**

**--size;**

**}**

**// Function to delete a book at a specified position**

**static void deleteAtPosition(Library books[], int &size, int pos) { for (int i = pos; i < size - 1; ++i) {**

**books[i] = books[i + 1];**

**}**

**--size;**

**}**

**};**

**int main() {**

**int n, choice;**

**cout << "Enter the number of books: "; cin >> n;**

**// Create an array of Book objects**

**Library books[100]; // Assuming a maximum of 100 books int size = n;**

**// Input details for each book for (int i = 0; i < n; ++i) {**

**cout << "\nEntering details for book number " << i + 1 << ":\n"; books[i].inputBookDetails();**

**}**

**do {**

**cout << "\nEnter your choice:\n";**

|  |  |  |  |
| --- | --- | --- | --- |
| **cout** | **<<** | **"1.** | **Display all the books\n";** |
| **cout** | **<<** | **"2.** | **Display specific book\n";** |
| **cout** | **<<** | **"3.** | **Insert at position\n";** |
| **cout** | **<<** | **"4.** | **Insert at start\n";** |
| **cout** | **<<** | **"5.** | **Insert at end\n";** |
| **cout** | **<<** | **"6.** | **Delete at start\n";** |
| **cout** | **<<** | **"7.** | **Delete at end\n";** |
| **cout** | **<<** | **"8.** | **Delete at position\n";** |
| **cout** | **<<** | **"9.** | **Exit\n";** |

**cin >> choice;**

**switch (choice) { case 1:**

**Library::displayAllBooks(books, size); break;**

**case 2:**

**Library::displaySpecificBook(books, size); break;**

**case 3: {**

**Library newBook; newBook.inputBookDetails(); int pos;**

**cout << "Enter position to insert: "; cin >> pos;**

**Library::insertAtPosition(books, size, pos - 1, newBook); break;**

**}**

**case 4: {**

**Library newBook; newBook.inputBookDetails();**

**Library::insertAtStart(books, size, newBook); break;**

**}**

**case 5: {**

**Library newBook; newBook.inputBookDetails(); Library::insertAtEnd(books, size, newBook); break;**

**}**

**case 6:**

**Library::deleteAtStart(books, size);**

**break; case 7:**

**Library::deleteAtEnd(books, size); break;**

**case 8: {**

**int pos;**

**cout << "Enter position to delete: "; cin >> pos;**

**Library::deleteAtPosition(books, size, pos - 1); break;**

**}**

**case 9:**

**cout << "Exiting...\n"; break;**

**default:**

**cout << "Invalid Input\n"; break;**

**}**

**} while (choice != 9);**

**return 0;**

**}**

### Observations and Learning:

*(****Students must write the observations and learning based on their understanding built about the subject matter and inferences drawn)***

### Class Definition:

The Library class contains private members: bookName, bookCost, issuedStatus, and serialNumber.

It has a default constructor and a parameterized constructor for initialization.

Public methods include functions for inputting and displaying book details, and various static methods for handling operations on an array of Library objects.

### Book Management:

The inputBookDetails function gathers details of a book from the user. The displayBookDetails function prints the details of a book.

displayAllBooks and displaySpecificBook functions allow for viewing all books or a specific book based on serial number.

### Array Operations:

Static methods insertAtPosition, insertAtStart, insertAtEnd, deleteAtStart, deleteAtEnd, and deleteAtPosition manage the array of books, allowing insertion and deletion of books at specified positions.

### Main Function:

The main function initializes an array of Library objects and collects book details from the user.

A menu-driven interface allows the user to choose various operations on the books array, including displaying all books, displaying a specific book, inserting books at various positions, and deleting books from various positions.

### Menu and Input Handling:

The program uses a do-while loop to repeatedly present a menu to the user until they choose to exit.

The user inputs their choice, and the program executes the corresponding operation.

### Conclusion:

(Students must write the conclusive statements as per the attainment of individual outcomes listed above and learning/observation noted in section B.2)

This program effectively demonstrates how to manage a collection of books using an object-oriented approach in C++. By encapsulating book properties and operations within the Library class, the program maintains a clean and modular structure. Static methods facilitate operations on an array of Library objects, showcasing how class functions can be used to manipulate class instances collectively. The menu-driven interface allows users to interact with the program intuitively, performing various operations such as insertion, deletion, and display of books. Overall, the program provides a practical example of object- oriented programming concepts, including encapsulation, constructors, and static methods, applied to a real-world scenario of library management.

### Question of curiosity:

**What is the difference between int array[] and int[] array?**

##### C++ Array Declarations Int array[]:

* **Usage:** This form is used when the array is being declared and its size is either inferred from initialization or not explicitly specified.

##### Example:

int array[] = {1, 2, 3, 4, 5}; // The size is inferred to be 5 based on the number of initializers

##### Notes:

* \\When initializing an array this way, the compiler determines the size of the array based on the number of elements in the initializer list.

##### Int array[5]:

* **Usage:** This form explicitly specifies the size of the array.

##### Example:

* int array[5]; // Declares an array of 5 integers
* array[0] = 10; // Assigning a value to the first element

##### Key Points: Initialization and Size:

int array[] can be used with an initializer list where the size is automatically determined by the number of elements provided.

int array[5] explicitly defines the size of the array, and you must ensure that you do not exceed this size when accessing or modifying the array.

##### Flexibility:

int array[] is useful when you want the compiler to automatically handle the size of the array based on the number of initial values provided.

int array[5] is used when you know the exact size of the array in advance and do not need initialization or want to define the size explicitly without initial values.

### Can we declare array size as negative? Justify

No, in C++, you cannot declare an array with a negative size. The reasons are both logical and technical:

##### Logical Inconsistency:

* An array size represents the number of elements it can hold.
* A negative number of elements is nonsensical as you cannot have a collection with fewer than zero elements.

##### Memory Allocation:

* Arrays require a specific amount of memory to be allocated at compile time.
* A negative size would lead to undefined behavior because the memory allocation routines expect a non-negative size.

##### Compiler Enforcement:

* Most C++ compilers will flag the use of a negative array size as a compile-time error.
* For example: int arr[-5]; // This will cause a compile-time error

##### Standard Compliance:

* The C++ Standard explicitly defines array sizes to be of type `size\_t`, which is an unsigned integral type. This inherently prevents negative sizes since `size\_t` can only represent non-

negative values.

* Trying to declare an array with a negative size will result in a compile-time error similar to: error: size of array ‘arr’ is negative
* Therefore, it is both logically invalid and technically unsupported to declare an array with a negative size in C++.

### What are the advantages and disadvantages of an array? Advantages of Arrays

* Fixed Size: The size of an array is determined at the time of its creation and cannot be changed. This makes memory management straightforward.
* Efficient Access: Arrays provide O(1) time complexity for accessing elements by index, making them very efficient for lookups.
* Memory Contiguity: Elements in an array are stored in contiguous memory locations, which can lead to improved cache performance and faster access times.
* Ease of Use: Arrays are simple to understand and use, with straightforward syntax for defining, accessing, and iterating over elements.
* Static Allocation: In languages like C and C++, arrays can be statically allocated, reducing the overhead of dynamic memory allocation and deallocation.
* Efficiency for Iteration: Iterating through an array is typically very efficient due to its contiguous memory layout.

### Disadvantages of Arrays

* Fixed Size: The fixed size of arrays can be a limitation if the required size is not known at compile time or if the array needs to grow or shrink dynamically.
* Wasted Space: If the array size is overestimated, there can be wasted memory space. Conversely, underestimating the size requires resizing, which is not straightforward for statically allocated arrays.
* Insertion and Deletion: Inserting or deleting elements from an array (except at the end) can be inefficient, as it may require shifting multiple elements, leading to O(n) time complexity.
* Lack of Flexibility: Arrays do not provide built-in mechanisms for dynamic resizing, complex data structures, or advanced operations such as sorting or searching.
* Type Homogeneity: Arrays typically require all elements to be of the same type, which can be restrictive if different types need to be stored together (although this can be mitigated with structures or classes in languages like C++).

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

# Experiment 2

(PART B: TO BE COMPLETED BY STUDENTS)

**(Students must submit the soft copy as per following segments within two hours of the practical. The soft copy must be uploaded on the Blackboard or emailed to the concerned lab in charge faculties at the end of the practical in case the there is no Black board access available)**

|  |  |
| --- | --- |
| Roll No. C013 | Name: Ashmit Jain |
| Program : BTI | Division: B |
| Batch: B1 | Date of Experiment: |
| Date of Submission: 7/8/2024 | Grade : |

### B.1 Tasks given in PART A to be completed here

*(****Students must write the answers of the task(s) given in the PART A )***

#include<iostream>

using namespace std;

const int MAX\_SIZE = 100;

class Stack

{

private:

int stack[MAX\_SIZE];

int top;

public:

Stack()

{

top = -1;

}

bool isEmpty()

{

return (top == -1);

}

bool isFull()

{

return (top == MAX\_SIZE - 1);

}

void push(int element)

{

if(isFull())

{

cout<<"Stack overflow. Can't push";

}

else

{

stack[++top] = element;

cout<<"Pushed Element"<<element<<endl;

}

}

void pop()

{

if(isEmpty())

{

cout<<"Stack underflow. Can't pop";

}

else

{

cout<<"Element"<<stack[top--]<<"Is popped."<<endl;

}

}

int peek()

{

if(isEmpty())

{

cout<<"No top element"<<endl;

return -1;

}

else

{

return stack[top];

}

}

void display()

{

if (isEmpty())

{

cout << "Stack is empty. No elements to display." << endl;

}

else

{

cout << "Stack elements: ";

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

{

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

}

cout << endl;

}

}

};

int main()

{

Stack MyStack;

int element;

int choice;

do

{

cout<<"Menu"<<endl;

|  |  |  |  |
| --- | --- | --- | --- |
| cout | << | "1. | Push Element" << endl; |
| cout | << | "2. | Pop Element" << endl; |
| cout | << | "3. | Peek Element" << endl; |
| cout | << | "4. | Display Stack" << endl; |
| cout | << | "5. | Exit" << endl; |

cout<<"Enter choice: ";

cin>>choice;

switch(choice)

{

case 1:

cout<<"Enter the Element which needs to get pushed. "<<endl;

cin>>element;

MyStack.push(element);

break;

case 2:

MyStack.pop();

break;

case 3:

MyStack.peek();

break;

case 4:

MyStack.display();

break;

case 5:

cout<<"GOODBYEEE";

break;

default:

cout<<"Invalid input";

}

} while (choice != 5);

return 0;

}

### Using Recursion:

#include <iostream>

#include <stack>

using namespace std;

void sortedInsert(stack<int> &s, int element) // insert an element in a sorted

stack

{

if (s.empty() || element >= s.top()) // 90>=8

{

s.push(element);

// 1 2 3 8 90

return;

}

int top = s.top(); // 1 2 3 8 90(pop), 9 >= 90

s.pop();

sortedInsert(s, element); //after pushing 1 2 3 8 9

s.push(top); //stored value will get pushed

}

// Function to sort the stack using recursion

void sortStack(stack<int> &s)

{

if (s.empty())

{

return;

}

int top = s.top(); //stores top value in top 1 2 3 8 9

s.pop();

sortStack(s); //it will pop until its empty

sortedInsert(s, top); //stored value will get pushed 9 8 3 2 1

}

// Helper function to print stack

void printStack(stack<int> s)

{

while (!s.empty())

{

cout << s.top() << " ";

s.pop();

}

cout << endl;

}

int main()

{

stack<int> s;

int n;

// Take number of elements input from the user

cout << "Enter the number of elements in the stack: ";

cin >> n; //5

// Take elements input from the user

cout << "Enter " << n << " elements:" << endl;

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

{

int element;

cin >> element;

s.push(element); // 1 2 3 4 5

}

cout << "Stack before sorting: ";

printStack(s); //

sortStack(s);

cout << "Stack after sorting: ";

printStack(s);

return 0;

}

### Observations and Learning:

*(****Students must write the observations and learning based on their understanding built about the subject matter and inferences drawn)***

**Implementing Stacks with Arrays**:

* You created a stack data structure using an array to store the elements.
* You implemented basic stack operations such as push (to add an element to the top of the stack), pop (to remove the top element), peek (to view the top element without removing it), and isEmpty (to check if the stack is empty).

**Sorting a Stack Using Recursion**:

* You used a recursive approach to sort the elements within the stack.
* The process involves removing elements from the stack and using recursion to sort them before placing them back in the correct order.

### Conclusion:

*(****Students must write the conclusive statements as per the attainment of individual outcomes listed above and learning/observation noted in section B.2)***

In conclusion, I learned how to implement stacks and their operations using arrays, and I also explored how to sort stacks using recursion.

# Experiment 3

(PART B: TO BE COMPLETED BY STUDENTS)

**(Students must submit the soft copy as per following segments within two hours of the practical. The soft copy must be uploaded on the Blackboard or emailed to the concerned lab in charge faculties at the end of the practical in case the there is no Black board access available)**

|  |  |
| --- | --- |
| Roll No. C013 | Name: Ashmit Jain |
| Program : BTI | Division: B |
| Batch: B1 | Date of Experiment: 8/9/2024 |
| Date of Submission: 8/9/2024 | Grade : |

### Tasks given in PART A to be completed here

**#include <iostream> #include <string> using namespace std;**

**class Stack { private:**

**int size; int top;**

**char\* stackArray;**

**public:**

**// Constructor Stack(int maxSize) {**

**size = maxSize; top = -1;**

**stackArray = new char[size];**

**}**

**// Destructor**

**~Stack() {**

**delete[] stackArray;**

**cout << "Stack has been destroyed" << endl;**

**}**

**// Check if the stack is empty bool isEmpty() {**

**return top == -1;**

**}**

**// Check if the stack is full bool isFull() {**

**return top == size - 1;**

**}**

**// Push an element onto the stack void push(char ch) {**

**if (isFull()) {**

**cout << "Stack Overflow\n";**

**} else {**

**stackArray[++top] = ch;**

**cout << "Element pushed: " << ch << endl;**

**}**

**}**

**// Pop the topmost element from the stack void pop() {**

**if (isEmpty()) {**

**cout << "Stack Underflow\n";**

**} else {**

**cout << "Element popped: " << stackArray[top--] << endl;**

**}**

**}**

**// Peek at the topmost element without popping it char peek() {**

**if (isEmpty()) {**

**cout << "Stack is empty\n"; return '\0';**

**} else {**

**cout << "Topmost element: " << stackArray[top] << endl; return stackArray[top];**

**}**

**}**

**};**

**// Function to check if the braces in the string are balanced bool balancedBraces(string str) {**

**Stack stk(str.length());**

**for (char ch : str) {**

**if (ch == '(' || ch == '{' || ch == '[') { stk.push(ch);**

**} else if (ch == ')' || ch == '}' || ch == ']') { if (stk.isEmpty()) {**

**cout << "Unbalanced Braces" << endl; return false;**

**}**

**char topElement = stk.peek();**

**if ((ch == ')' && topElement == '(') || (ch == '}' && topElement == '{') || (ch == ']' && topElement == '[')) { stk.pop();**

**} else {**

**cout << "Unbalanced Braces" << endl; return false;**

**}**

**}**

**}**

**if (stk.isEmpty()) {**

**cout << "Balanced Braces" << endl; return true;**

**} else {**

**cout << "Unbalanced Braces" << endl; return false;**

**}**

**}**

**int main() {**

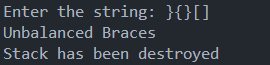
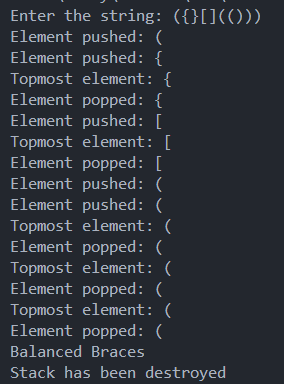
**string str;**

**cout << "Enter the string: "; cin >> str; balancedBraces(str);**

**return 0;**

**}**

### Output /Observations



**Observation:**

### Class Structure and Methods:

* + - * The Stack class is implemented with basic stack operations: push, pop, peek, isEmpty, and isFull.
      * The constructor dynamically allocates memory for the stack array based on the length of the input string.
      * The balancedBraces function is responsible for checking if the braces in a string are balanced using the stack.

### Balanced Brace Check Logic:

* + - * The string is traversed character by character.
      * Opening braces ((, {, [) are pushed onto the stack.
      * For each closing brace (), }, ]), the stack is checked to see if it has the corresponding opening brace at the top. If it does, that brace is popped from the stack.
      * If at any point the closing brace does not match the opening brace or the stack is empty when it shouldn't be, the string is deemed unbalanced.
      * At the end of the traversal, if the stack is empty, the braces are balanced; otherwise, they are unbalanced.

### Output Messages:

* + - * The program prints whether the braces are balanced or not, based on the operations performed during the string traversal.

### Conclusion:

The code effectively demonstrates how to use a stack data structure to solve the problem of checking balanced braces in a string. By pushing opening braces onto the stack and popping them when corresponding closing braces are encountered, the stack helps ensure that every opening brace has a matching closing brace in the correct order.

The separation of the Stack class and the balancedBraces function provides a clear and modular approach to solving this problem. The program handles both underflow (attempting to pop from an empty stack) and overflow (attempting to push onto a full stack) scenarios gracefully, although in this specific context of brace checking, overflow is unlikely to occur given the string's length determines the stack size.

This approach can be expanded to handle more complex scenarios involving other

types of delimiters or more intricate validation rules. 4o

## Experiment 4

(PART B : TO BE COMPLETED BY STUDENTS)

***(Students must submit the soft copy as per following segments within two hours of the practical. The soft copy must be uploaded on the Blackboard or emailed to the concerned lab in charge faculties at the end of the practical in case the there is no Black board access available)***

|  |  |
| --- | --- |
| Roll No. C013 | Name: Ashmit Jain |
| Class : B | Batch : B1 |
| Date of Experiment: 16/08/2024 | Date of Submission: 16/08/2024 |
| Grade : | Time of Submission: |
| Date of Grading: |  |

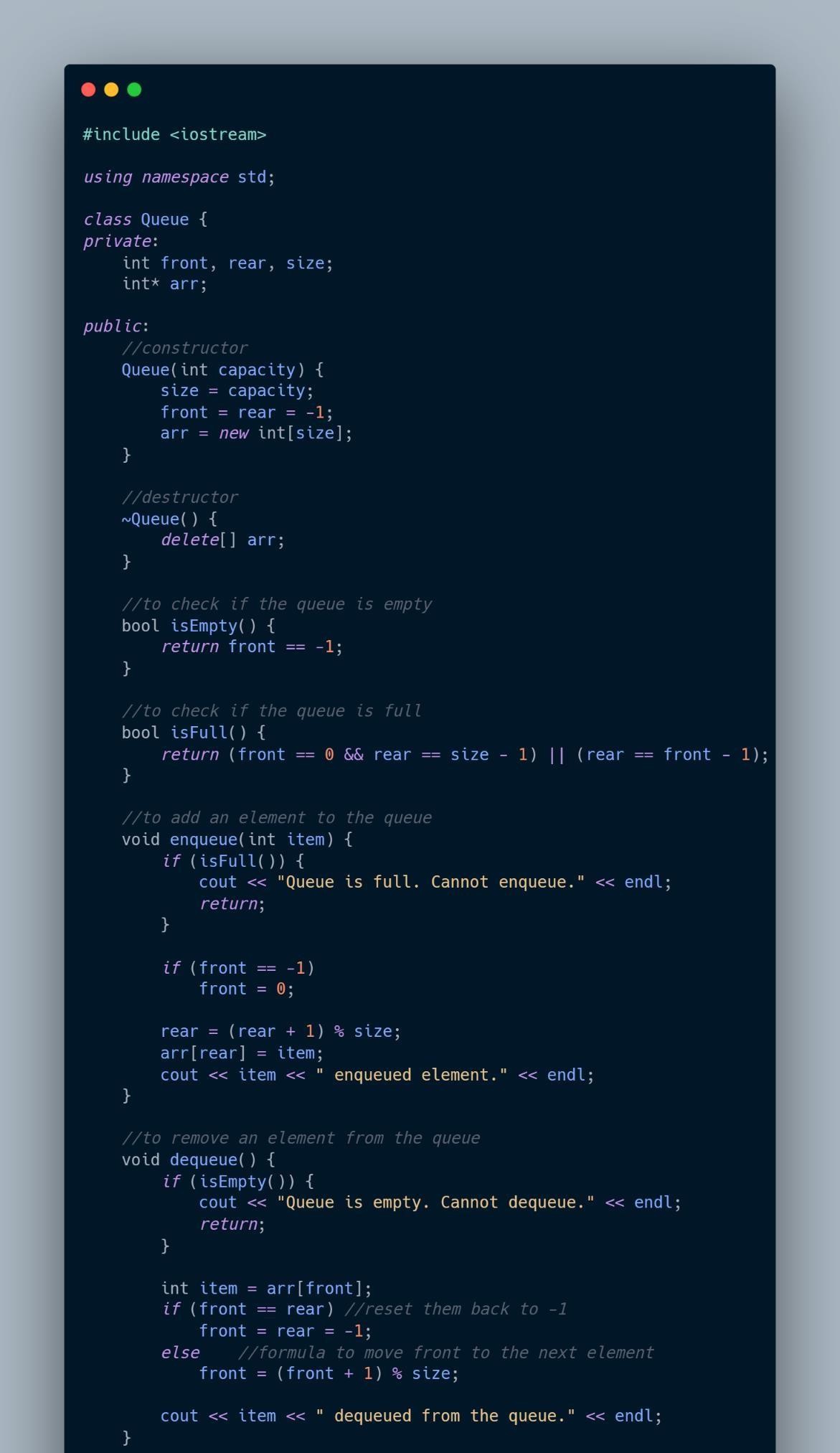
### Software Code written by student: (Task 1)

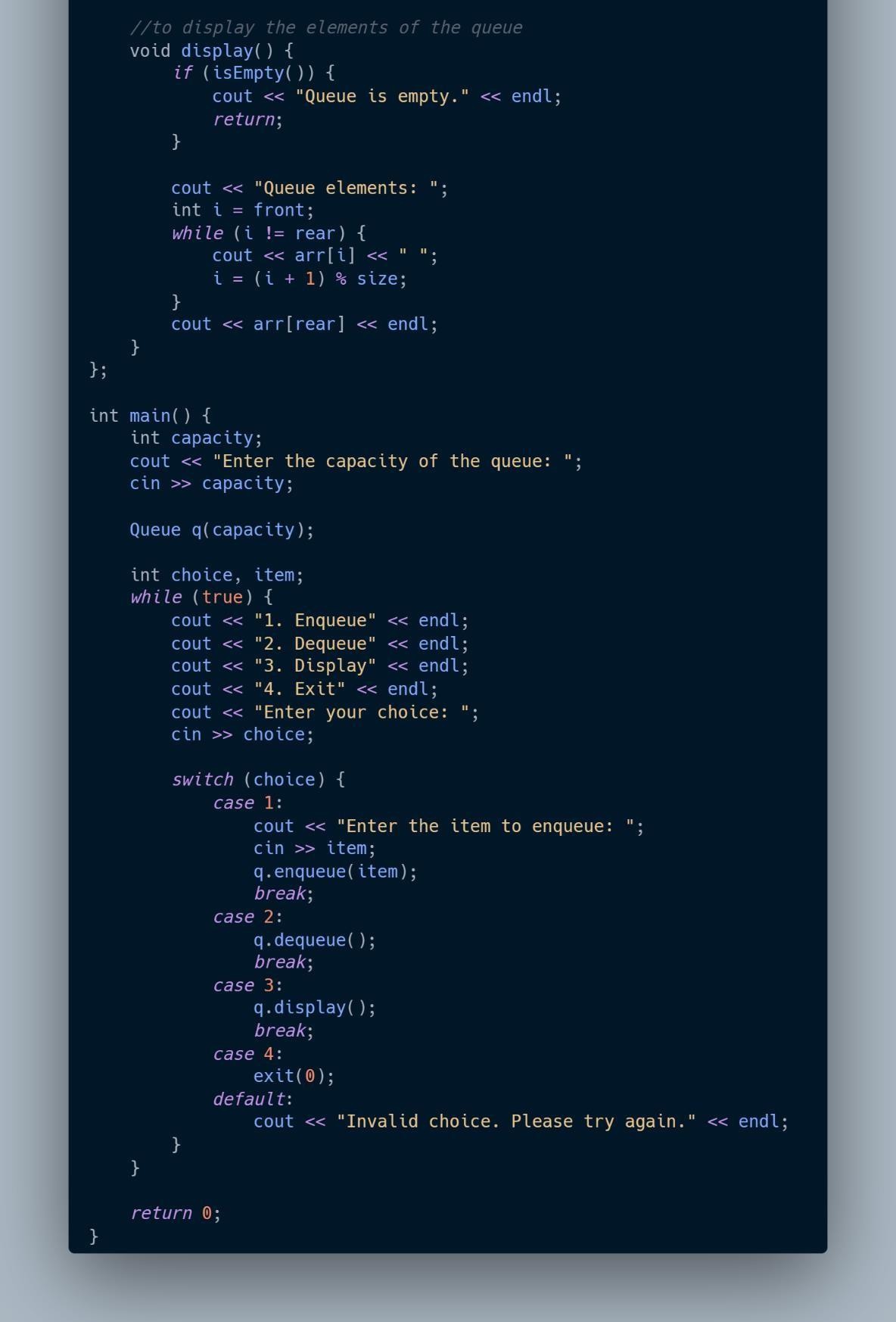
***(Paste your code completed during the 2 hours of practical in the lab here)***

**Task1:**



**Task2:**

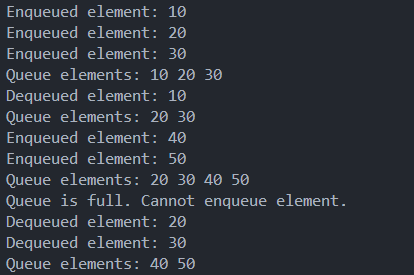




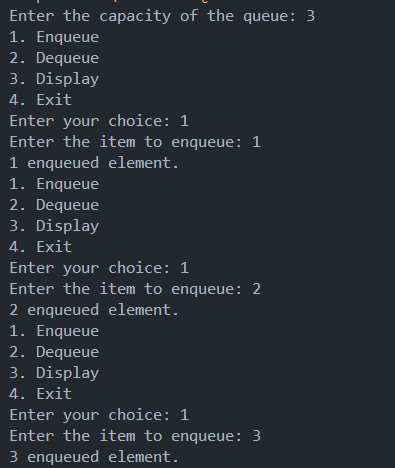
* 1. **Output:**

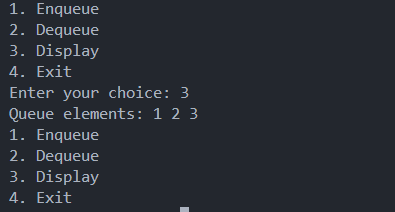
***(Paste your program input and output in following format, If there is error then paste the specific error in the output part. In case of error with due permission of the faculty extension can be given to submit the error free code with output in due course of time. Students will be graded accordingly.)***

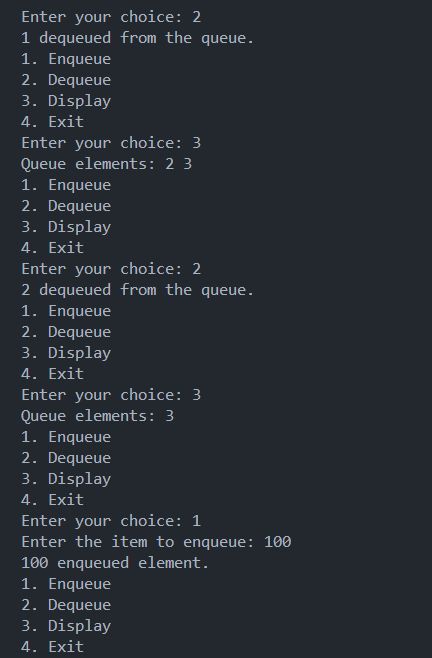
**Task1:**

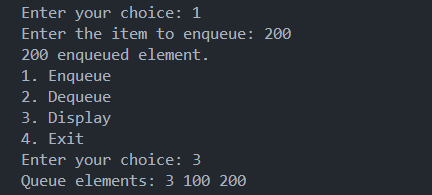


**Task2:**









* 1. **Observations and learning [w.r.t. all tasks]:**

***(Students are expected to comment on the output obtained with clear observations and learning for each task/ sub part assigned)***

##### Data Structures Used:

* + 1. **Array:**
       - Both programs use a dynamically allocated array (int\* arr or int\* queue) as the underlying data structure to implement the queue.
       - The array size is determined by the user or fixed at runtime, and memory is allocated accordingly.

##### Queue:

* + - * The programs implement a queue, which is a linear data structure that follows the **First-In-First-Out (FIFO)** principle.
      * The queue operations implemented include enqueue (to add an element to the rear of the queue), dequeue (to remove an element from the front), and display (to show the current elements in the queue).

##### Key Observations and Learning:

1. **Memory Management:**
   * The use of dynamic memory allocation (new and delete) is essential when implementing data structures that require variable sizes. This allows flexibility but also requires careful memory management to avoid leaks.
   * The destructor (~Queue) ensures that the dynamically allocated memory is properly deallocated, preventing memory leaks.

##### Circular Queue Implementation:

* + The first program implements a circular queue. The circular queue solves the problem of unused space that occurs in a linear queue after several dequeue operations by using modulo arithmetic to wrap around the array.
  + This ensures efficient use of the array and allows enqueueing at the beginning of the array once the end is reached, as long as there is free space.

##### Linear Queue Implementation:

* + The second program uses a simple linear queue, where the front and rear pointers are incremented linearly.
  + In this implementation, the queue can fill up and become unusable even if there are empty slots at the beginning of the array (due to the front pointer moving forward), leading to inefficient use of space.

##### Edge Case Handling:

* + Both implementations handle edge cases such as trying to enqueue when the queue is full and dequeue when it is empty, ensuring the program remains robust.

##### Queue Resetting:

* + In both implementations, when all elements are dequeued, the front and rear pointers are reset to -1. This is crucial for reusing the queue without unnecessary memory operations.

##### Efficiency Considerations:

* + The circular queue is generally more efficient in terms of space utilization than the linear queue because it makes better use of the array by reusing spaces freed by dequeue operations.

##### Practical Implementation:

* + These implementations provide a foundational understanding of how queues work in practice, including the trade-offs between different methods of implementation (circular vs. linear).

##### What I Learned:

* **Understanding of Queue Operations:** The fundamental operations of a queue (enqueue, dequeue, and display) and how they can be implemented using arrays.
* **Importance of Edge Case Handling:** Ensuring that the queue behaves correctly when it is full or empty, and understanding how to reset the queue for further use.
* **Circular vs. Linear Queues:** The circular queue is more efficient in terms of space, while the linear queue is simpler but can lead to wasted space.
* **Memory Management in C++:** The importance of dynamically allocating and deallocating memory to manage resources effectively, especially in data structures.
* **C++ Specific Constructs:** Familiarity with C++ syntax and constructs such as classes, constructors, destructors, and dynamic memory management.

These programs provide a practical introduction to implementing basic data structures in C++ and highlight the importance of considering efficiency and proper memory management.

### 4 Conclusion:

*(****Students must write the conclusion as per the attainment of individual outcome listed above and learning/observation noted in section B.3)***

In conclusion, these programs provide a practical understanding of implementing queues using arrays in C++. The comparison between circular and linear queues highlights the importance of efficient space utilization and proper memory management. The circular queue is more efficient in handling space, while the linear queue is simpler but may lead to wasted space. Additionally, the importance of handling edge cases and resetting pointers is emphasized to ensure robust and reusable code. Overall, these implementations offer valuable insights into basic data structure operations and memory management in C++.

### B.5 Question of Curiosity

***(To be answered by student based on the practical performed and learning/observations)***

##### Answer Following Question

1. A linear queue, if implemented using an array of size MAX\_SIZE, gets full when?
   1. Rear == MAX\_SIZE – 1
   2. Front == (rear + 1)mod MAX\_SIZE
   3. Front == rear + 1
   4. Rear == front

##### Answer: A

1. A linear queue, if implemented using an array of size MAX\_SIZE, gets empty when?
   1. Front== MAX\_SIZE – 1
   2. Rear == -1 Or FRONT > REAR
   3. Front == rear + 1
   4. Rear == front

##### Answer: B

1. A circular queue, if implemented using an array of size MAX\_SIZE, gets full when?
   1. Rear == MAX\_SIZE – 1
   2. Front == (rear + 1)mod MAX\_SIZE
   3. FRONT = 0 and Rear = MAX – 1 ) || (Front==Rear +1 )
   4. Rear == front

##### Answer: C

1. A circular queue, if implemented using an array of size MAX\_SIZE, gets empty when?
   1. Front== MAX\_SIZE – 1
   2. Rear == -1
   3. Front == rear + 1
   4. Rear == front

##### Answer: B

Write any two applications of queue data structure.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

## Experiment 5

(PART B : TO BE COMPLETED BY STUDENTS)

***(Students must submit the soft copy as per following segments within two hours of the practical. The soft copy must be uploaded on the Blackboard or emailed to the concerned lab in charge faculties at the end of the practical in case the there is no Black board access available)***

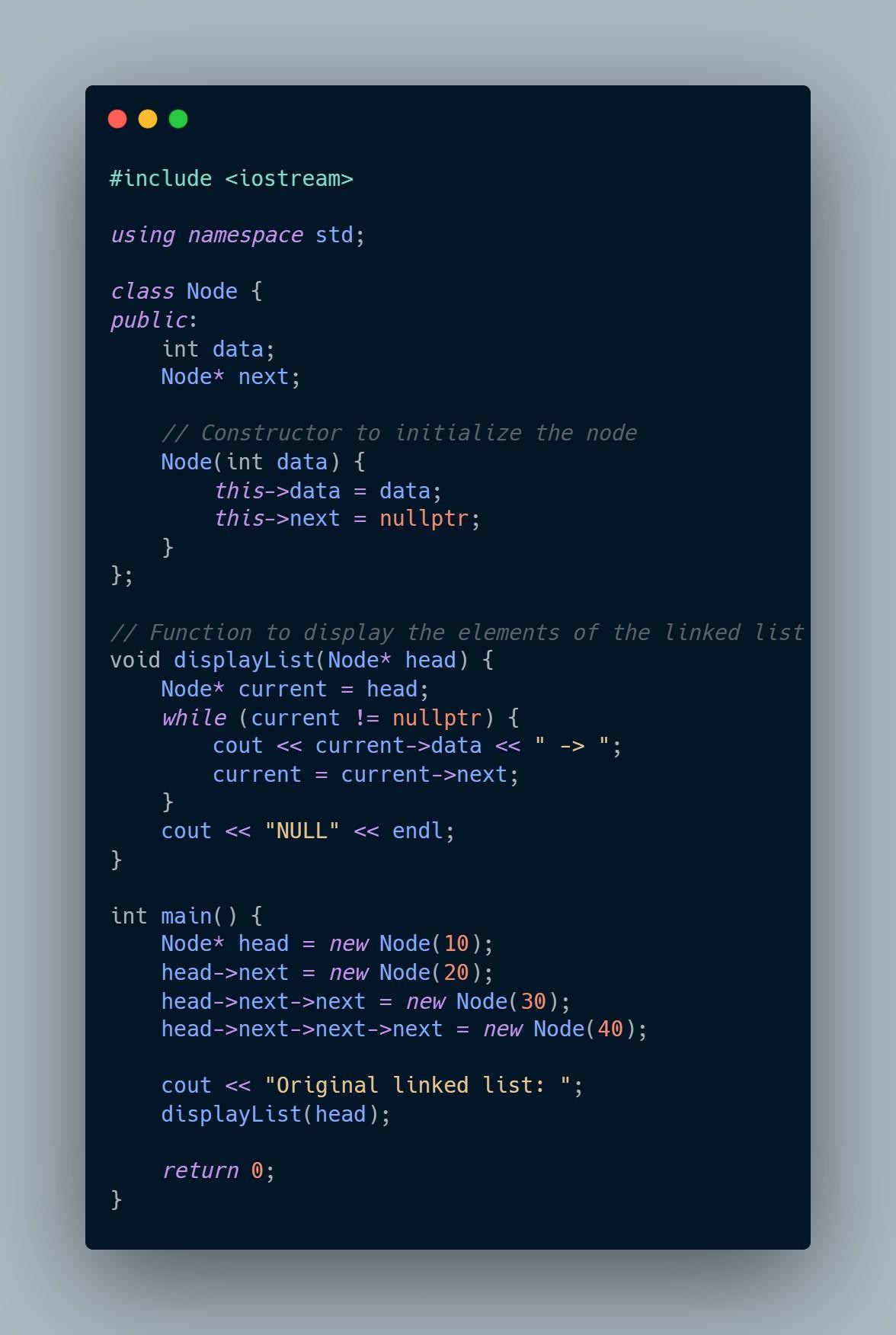
|  |  |
| --- | --- |
| Roll No. C013 | Name: Ashmit Jain |
| Class : B | Batch : B1 |
| Date of Experiment: | Date of Submission: 8/9/24 |
| Grade : | Time of Submission: |
| Date of Grading: |  |

### Software Code written by student: (Task 1)

***(Paste your Matlab code completed during the 2 hours of practical in the lab here)***

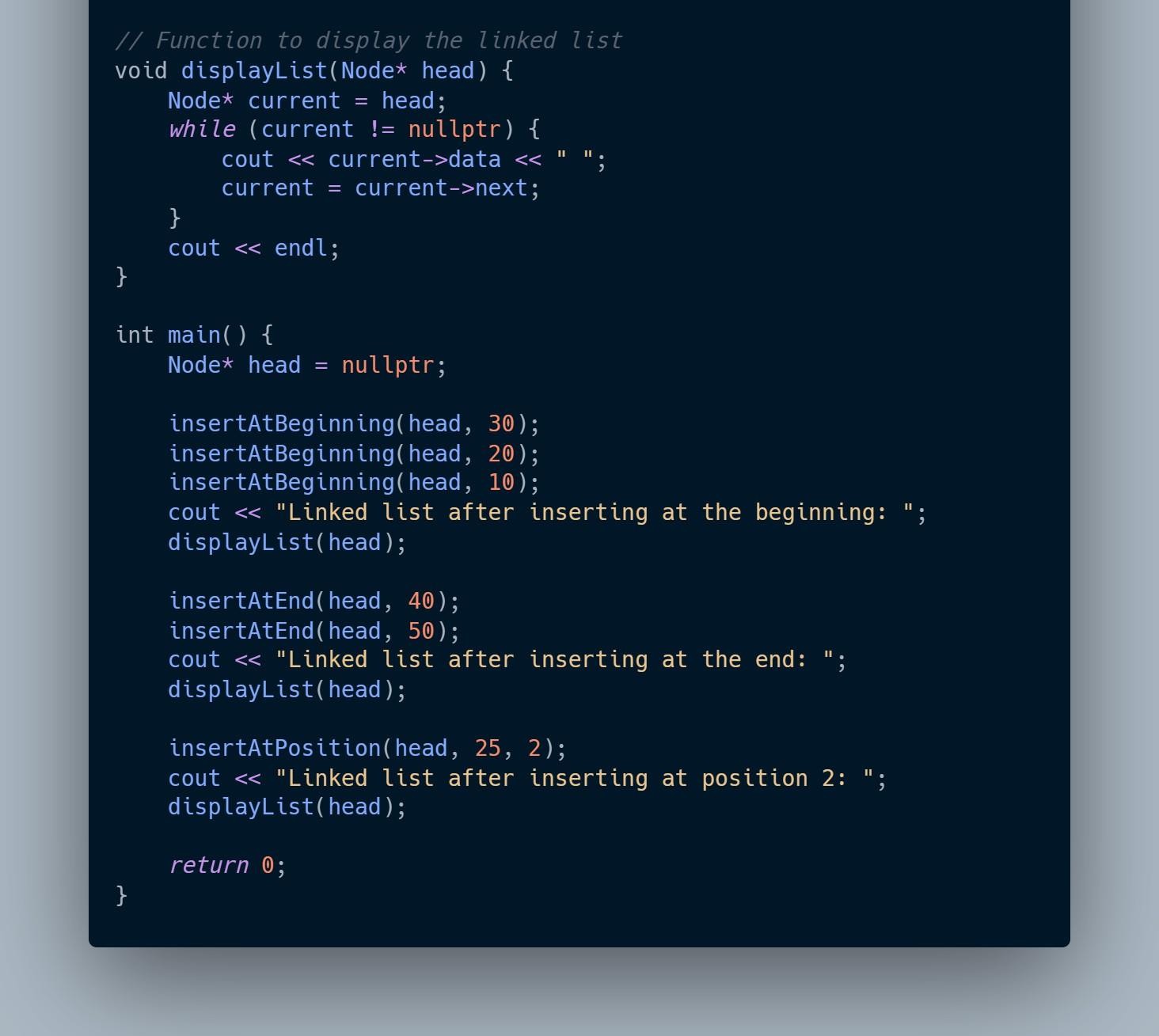
### Task1:

**1.1**

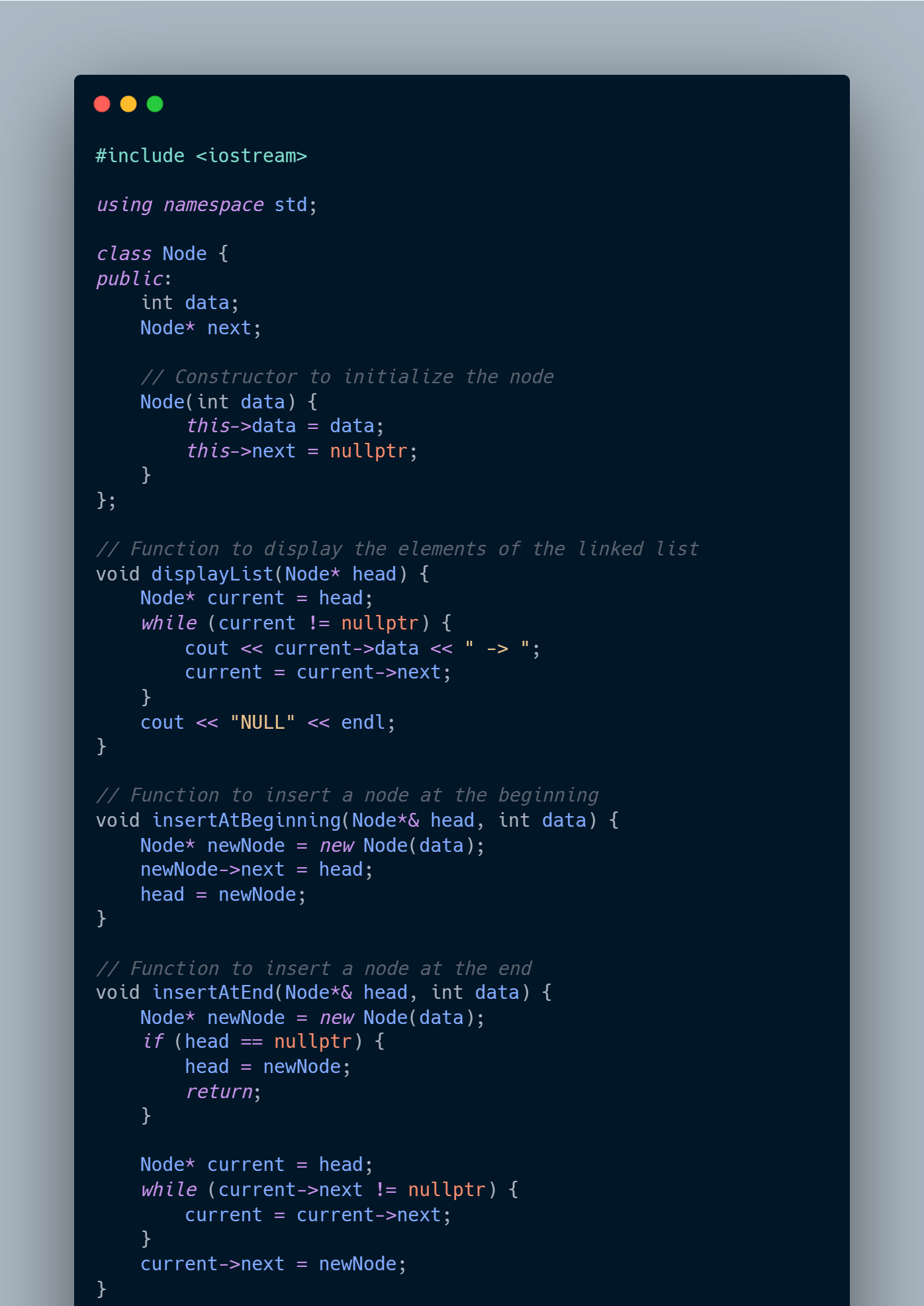


**1.2**





**1.3**







**Task1:**

#### Observations

##### Task 1.1: Creating and Displaying a Linked List

* + - * The linked list structure was successfully created with nodes containing integer data.
      * The displayList function correctly traversed the list and displayed the elements in the order they were linked.
      * The linked list terminates with NULL, indicating the end of the list, which was displayed as NULL at the end of the output.

##### Task 1.2: Inserting Nodes at Different Positions

* + - * **Insertion at the Beginning**:
        + Nodes were successfully added to the beginning of the list, and the head pointer was correctly updated.
        + The list displayed after insertion at the beginning showed that the new node was at the start, pushing previous elements to the right.

##### Insertion at the End:

* + - * + Nodes were successfully appended to the end of the list.
        + The list displayed after insertion at the end showed that the new node appeared at the end of the list.

##### Insertion at a Specific Position:

* + - * + Nodes were inserted at specific positions within the list.
        + The list displayed after insertion at a specific position showed that the new node was correctly placed between the appropriate existing nodes.
        + When attempting to insert at an out-of-bounds position, the program handled the error by displaying an appropriate message.

##### Task 1.3: Deleting Nodes at Different Positions

* + - * **Deletion at the Beginning**:
        + The head node was successfully deleted, and the head pointer was updated to the next node.
        + The list displayed after deletion at the beginning showed that the previous head was removed, and the second node became the new head.

##### Deletion at the End:

* + - * + The last node was successfully removed.
        + The list displayed after deletion at the end confirmed that the final node was deleted, and the second-to-last node became the new end, pointing to NULL.

##### Deletion at a Specific Position:

* + - * + Nodes were correctly removed from specified positions within the list.
        + The list displayed after deletion at a specific position showed that the targeted node was removed and the links between the remaining nodes were correctly maintained.
        + The program correctly handled cases where the position was out of bounds by displaying an appropriate message.

#### Conclusion

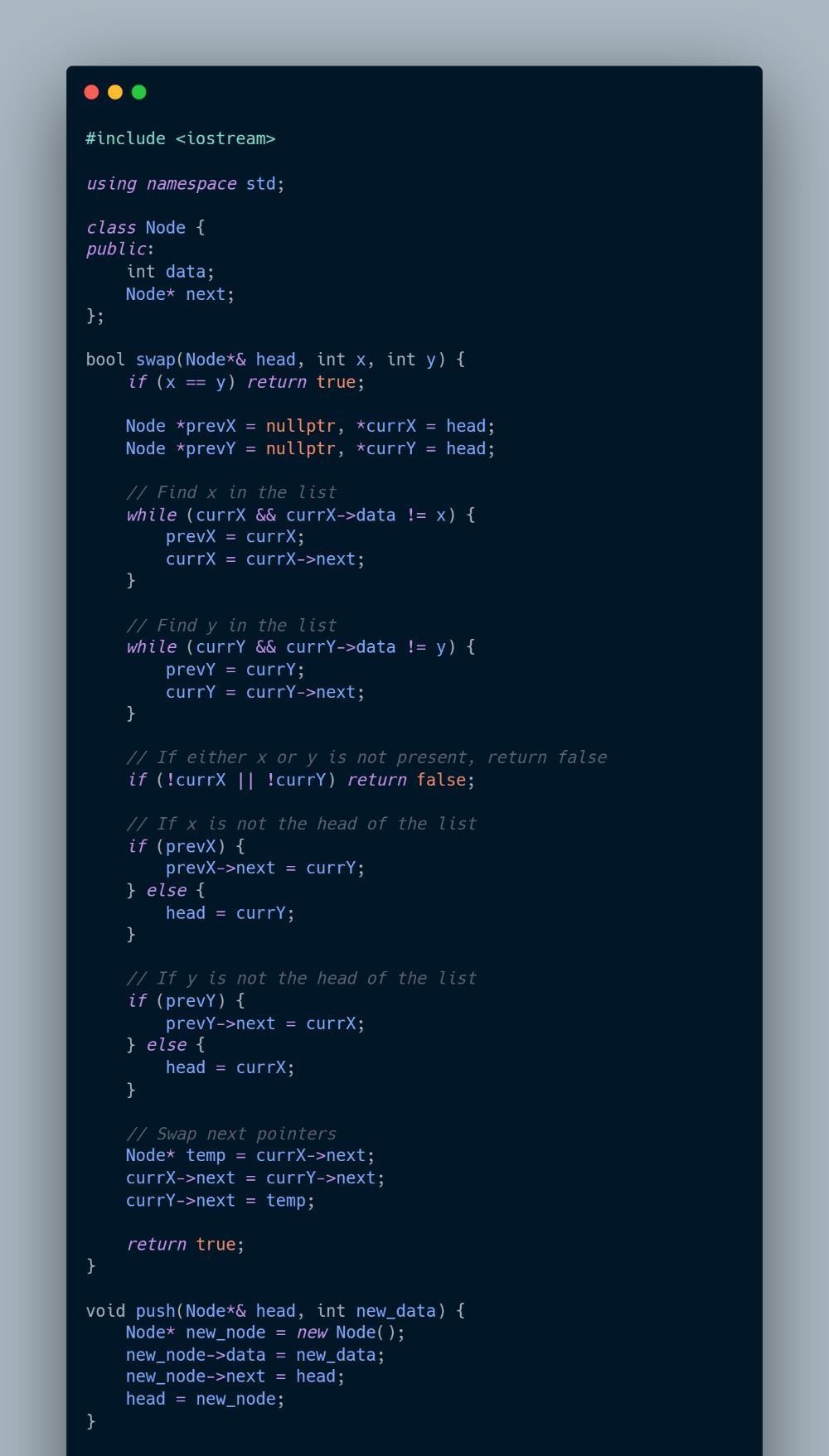
* **Task 1.1** confirmed that a basic linked list can be effectively constructed and traversed. This demonstrated an understanding of the fundamental operations required to manage a singly linked list in C++.
* **Task 1.2** illustrated that insertion operations at the beginning, end, and middle positions of the list can be managed seamlessly. This involved updating the head pointer, handling cases where the list was initially empty, and ensuring that all links between nodes were maintained correctly.
* **Task 1.3** demonstrated the ability to delete nodes from various positions within the list, with careful management of pointers to ensure no memory leaks or dangling pointers occurred. The program also handled edge cases, such as attempting to delete nodes from an empty list or from out-of-bounds positions, by providing clear feedback to the user.

Overall, these tasks collectively validated that the linked list implementation was both robust and flexible, capable of handling common operations like insertion and deletion in a variety of scenarios. The code effectively manages memory and maintains the integrity of the list structure during all operations, which is essential in the context of dynamic data structures like linked lists.

### B.5 Question of Curiosity

***(To be answered by student based on the practical performed and learning/observations)***

***1. Write an algorithm and C\C++ program that swaps (exchanges) two nodes in a list. The nodes are identified by number and are passed as parameters. For example, to exchange nodes 5 and 8, you would call swap(5,8). If the exchange is successful, the algorithm is to return true. If it encounters an error, such as an invalid node number, it returns false. Use linked list implementation.***





\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

## Experiment 6

(PART B : TO BE COMPLETED BY STUDENTS)

***(Students must submit the soft copy as per following segments within two hours of the practical. The soft copy must be uploaded on the Blackboard or emailed to the concerned lab in charge faculties at the end of the practical in case the there is no Black board access available)***

|  |  |
| --- | --- |
| Roll No. C013 | Name: Ashmit Jain |
| Class : B | Batch : B1 |
| Date of Experiment: | Date of Submission: |
| Grade : | Time of Submission: |
| Date of Grading: |  |

### Software Code written by student:

***(Paste your code completed during the 2 hours of practical in the lab here)***

### Task1:

Write a C/C++ program to implement operations of stack using linked list.





### Task2:

Write a C/C++ program to implement operations of queue using linked list.

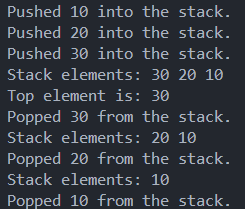




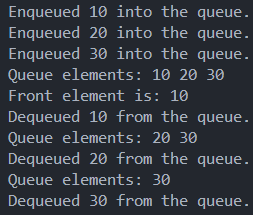
### Input and Output:

***(Paste your program input and output in following format, If there is error then paste the specific error in the output part. In case of error with due permission of the faculty extension can be given to submit the error free code with output in due course of time. Students will be graded accordingly.)***

**Task1:**



**Task2:**



* 1. **Observations and learning [w.r.t. all tasks]:**

***Observations for the given experiment are as follows:***

### Stack (LIFO - Last In First Out) Using Linked List:

##### Push Operation:

* + - * A new node is inserted at the **top** of the stack.
      * The newly inserted node’s next pointer is set to point to the previous top node, making it the new top of the stack.
      * Time complexity: **O(1)** for each push, as insertion is done at the head of the list.

##### Pop Operation:

* + - * Removes the top node from the stack, i.e., the node at the head of the linked list.
      * After removal, the next node becomes the new top.
      * Time complexity: **O(1)** for each pop, as removal happens at the head of the list.

##### Peek Operation:

* + - * Directly returns the value at the top of the stack without removing it.
      * Time complexity: **O(1)**.

##### Advantages:

* + - * Dynamic size: The stack can grow and shrink dynamically as nodes are added or removed, unlike arrays where size must be predefined.
      * No need to shift elements as in array-based implementations.

### Queue (FIFO - First In First Out) Using Linked List:

##### Enqueue Operation:

* + - * A new node is inserted at the **rear** of the queue.
      * The rear pointer is updated to point to this new node.
      * Time complexity: **O(1)** for each enqueue, as insertion is done at the tail of the list.

##### Dequeue Operation:

* + - * The node at the **front** (head) of the queue is removed.
      * The front pointer is updated to point to the next node in the list.
      * Time complexity: **O(1)** for each dequeue, as removal happens at the head of the list.

##### Peek Operation:

* + - * Returns the value of the front node without removing it.
      * Time complexity: **O(1)**.

##### Advantages:

* + - * Dynamic size: Like the stack, the queue can grow and shrink dynamically based on the number of elements.
      * No need for element shifting as required in array-based implementations when dequeuing.

### Common Observations:

##### Dynamic Nature:

* + - * Both stack and queue implementations using linked lists are dynamic, allowing for flexible memory usage.
      * Unlike array-based implementations, you don’t need to allocate a fixed amount of memory upfront.

##### Memory Efficiency:

* + - * Memory is allocated only when needed (during push or enqueue operations).
      * However, linked list implementations do have a small memory overhead for storing pointers (next) in addition to data.

##### No Overflow:

* + - * Since memory allocation is dynamic, there’s no risk of overflow (as with arrays when the predefined capacity is reached), except when system memory is exhausted.

##### Time Complexity:

* + - * Both implementations have constant time complexity, **O(1)**, for key operations (push, pop, enqueue, dequeue, peek).
      * Linked lists avoid the **O(n)** worst-case time complexity of shifting elements in array-based stacks/queues (especially for dequeuing or popping in arrays).

##### Destruction and Cleanup:

* + - * Both the stack and queue implementations require careful cleanup to avoid memory leaks. The destructor is used to free up the memory allocated for nodes when the stack or queue is destroyed.

##### Performance Trade-offs:

* + - * Linked list implementations might have slightly worse cache performance than array-based implementations, as linked list nodes are not stored contiguously in memory.

### 4 Conclusion:

* The linked list approach to implementing both stacks and queues is efficient in terms of dynamic size management and constant-time operations for insertion and removal.
* It provides more flexibility compared to array-based implementations, which may have fixed sizes and require costly element shifting for certain operations.

### B.5 Question of Curiosity

***(To be answered by student based on the practical performed and learning/observations)***

##### Differentiate between static and dynamic implementation of stack and queue.



* + - **Static (Array-based)**:
    - This implementation is ideal for use cases where the maximum size of the stack or queue is known in advance and does not change frequently. It offers faster operations with minimal overhead but is limited in flexibility.

##### Dynamic (Linked List-based):

* + - This implementation is better suited for scenarios where the number of elements in the stack or queue is highly variable. While it may have some performance overhead due to dynamic memory management, it offers greater flexibility and memory efficiency.

## Experiment 7

(PART B : TO BE COMPLETED BY STUDENTS)

***(Students must submit the soft copy as per following segments within two hours of the practical. The soft copy must be uploaded on the Blackboard or emailed to the concerned lab in charge faculties at the end of the practical in case the there is no Black board access available)***

|  |  |
| --- | --- |
| Roll No. C013 | Name: Ashmit Jain |
| Class: B | Batch: B1 |
| Date of Experiment: | Date of Submission: 22/9/2024 |
| Grade : | Time of Submission: |
| Date of Grading: |  |

### Software Code written by student:

***(Paste your code completed during the 2 hours of practical in the lab here)***

### Task1:

#include <iostream> using namespace std;

class Node { public:

int data; // Data stored in the node Node\* left; // Pointer to the left child Node\* right; // Pointer to the right child

// Constructor to initialize a node with a given value Node(int value) {

data = value; left = nullptr; right = nullptr;

}

// Pre-order traversal (root, left, right) void preOrderTraversal(Node\* node) {

if (node == nullptr) return; cout << node->data << " "; preOrderTraversal(node->left); preOrderTraversal(node->right);

}

};

int main() {

// Manually constructing the binary tree Node\* root = new Node(50);

root->left = new Node(30); root->right = new Node(70);

root->left->left = new Node(20); root->left->right = new Node(40); root->right->left = new Node(60); root->right->right = new Node(80);

// Pre-order traversal of the tree cout << "Pre-order traversal: "; root->preOrderTraversal(root); cout << endl;

// Cleaning up memory delete root;

return 0;

}

### Task2:

#include <iostream>

using namespace std;

class Node

{

public:

int data; Node\* left; Node\* right;

// Constructor to initialize the node with data Node(int value = 0)

{

data = value; left = nullptr; right = nullptr;

}

// Destructor to clean up dynamically allocated memory

~Node()

{

delete left; delete right;

}

// Pre-order traversal (root, left, right) void preOrderTraversal(Node\* root)

{

if (root == nullptr) return; cout << root->data << " "; preOrderTraversal(root->left); preOrderTraversal(root->right);

}

// In-order traversal (left, root, right) void inOrderTraversal(Node\* root)

{

if (root == nullptr) return; inOrderTraversal(root->left); cout << root->data << " "; inOrderTraversal(root->right);

}

// Post-order traversal (left, right, root) void postOrderTraversal(Node\* root)

{

if (root == nullptr) return; postOrderTraversal(root->left); postOrderTraversal(root->right); cout << root->data << " ";

}

};

int main()

{

// Initialize the main root of the tree Node\* root = new Node(10);

// Manually build the tree root->left = new Node(2); root->right = new Node(3);

root->left->left = new Node(4); root->left->right = new Node(5); root->right->left = new Node(6); root->right->right = new Node(7);

// Traversals

cout << "Pre-order traversal: "; root->preOrderTraversal(root); cout << endl;

cout << "In-order traversal: "; root->inOrderTraversal(root); cout << endl;

cout << "Post-order traversal: "; root->postOrderTraversal(root); cout << endl;

// Cleanup memory delete root;

return 0;

}

### Input and Output:

***(Paste your program input and output in following format, If there is error then paste the specific error in the output part. In case of error with due permission of the faculty extension can be given to submit the error free code with output in due course of time. Students will be graded accordingly.)***

**Task1:**



**Task2:**



* 1. **Observations and learning [w.r.t. all tasks]:**

##### Representation of Binary Tree:

Binary trees can be represented in memory using either arrays or linked lists.

The linked list representation is more efficient for binary trees because it dynamically allocates memory for each node, allowing non-contiguous storage of nodes in memory.

##### Binary Tree Structure:

Each node in a binary tree contains three components: data, a left pointer, and a right pointer.

The left pointer references the left child, the right pointer references the right child, and if a node has no children, the pointers are set to nullptr.

##### Tree Traversal:

**Preorder Traversal**:

The root is visited first, followed by the left subtree, then the right subtree.

##### Inorder Traversal:

The left subtree is visited first, followed by the root, then the right subtree.

##### Postorder Traversal:

The left subtree is visited first, followed by the right subtree, and the root is visited last.

##### DFS Traversals:

The Depth-First Search (DFS) method allows efficient tree traversal by exploring nodes down a branch before backtracking. All three traversals are forms of DFS, differing only in the order nodes are visited.

### 4 Conclusion:

##### Memory Representation:

The linked list representation of binary trees allows flexible memory allocation and easy manipulation of nodes, making it a preferred method over array representation for binary trees.

##### Traversal Algorithms:

Implementing DFS traversal techniques (preorder, inorder, and postorder) provides a comprehensive method to access and process all nodes in a binary tree. These traversal techniques are foundational in tree operations and are useful in various applications like expression trees, syntax trees, and searching.

##### Binary Tree Analysis:

Understanding tree height, node relationships, and traversal methods is critical for efficient manipulation and navigation of binary tree structures. Traversals provide different views of tree data, and each method is suitable for specific use cases in programming.

## Experiment 8

(PART B : TO BE COMPLETED BY STUDENTS)

***(Students must submit the soft copy as per following segments within two hours of the practical. The soft copy must be uploaded on the Blackboard or emailed to the concerned lab in charge faculties at the end of the practical in case the there is no Black board access available)***

|  |  |
| --- | --- |
| Roll No. C013 | Name: Ashmit Jain |
| Class : B | Batch : B1 |
| Date of Experiment: | Date of Submission: 28/09/2024 |
| Grade : | Time of Submission: |
| Date of Grading: |  |

### Software Code written by student:

***(Paste your code completed during the 2 hours of practical in the lab here)***

### Task1:

Write a C/C++ program to implement insertion and deletion on BST

#*include*<iostream> using namespace std;

class Node { public:

int data; Node\* left; Node\* right;

*Node*(int value = 0) { data = value;

left = nullptr;

right = nullptr;

}

};

class BST { public:

// *Main root node of class Node*

Node\* root;

// *Constructor to initialize the root node BST*() {

root = nullptr;

}

void *insert*(int value) {

root = *insertData*(root, value);

}

bool *search*(int value) {

*return searchRec*(root, value);

}

void *inOrderTraversal*() { *inOrderTraversal*(root); cout << endl;

}

void *deleteNode*(int value) {

root = *deleteNodeRec*(root, value);

}

private:

Node*\* insertData*(Node*\** node, int value) {

// *If the tree is empty, create a new node and return it*

*if* (node == nullptr) {

*return* new *Node*(value);

}

// *Otherwise, recur down the tree if* (value < node->data) {

node->left = *insertData*(node->left, value);

} *else if* (value > node->data) {

node->right = *insertData*(node->right, value);

}

// *Return the (unchanged) node pointer return* node;

}

bool *searchRec*(Node*\** node, int value) {

// *Base case: root is null or value is present at root if* (node == nullptr) {

*return* false;

}

*if* (node->data == value) {

*return* true;

}

// *Value is greater than root's data if* (value < node->data) {

*return searchRec*(node->left, value);

}

// *Value is smaller than root's data return searchRec*(node->right, value);

}

void *inOrderTraversal*(Node*\** node) {

*if* (node != nullptr) { *inOrderTraversal*(node->left); cout << node->data << " -> "; *inOrderTraversal*(node->right);

}

}

// *Helper function to find the minimum value node in a subtree*

Node*\* findMin*(Node*\** node) {

*while* (node && node->left != nullptr) { node = node->left;

}

*return* node;

}

// *Function to delete a node from the BST*

Node*\* deleteNodeRec*(Node*\** node, int value) {

*if* (node == nullptr) {

*return* node;

}

// *Traverse the tree to find the node to be deleted if* (value < node->data) {

node->left = *deleteNodeRec*(node->left, value);

} *else if* (value > node->data) {

node->right = *deleteNodeRec*(node->right, value);

} *else* {

// *Node to be deleted found if* (node->left == nullptr) {

Node\* temp = node->right; delete node;

*return* temp;

} *else if* (node->right == nullptr) { Node\* temp = node->left; delete node;

*return* temp;

}

// *Node with two children: get the inorder successor (smallest in the right subtree)*

Node\* temp = *findMin*(node->right);

// *Copy the inorder successor's data to this node*

node->data = temp->data;

// *Delete the inorder successor*

node->right = *deleteNodeRec*(node->right, temp->data);

}

*return* node;

}

};

int *main*() { BST tree; int inp;

char choice;

// *Inserting nodes into the tree*

tree.*insert*(5); tree.*insert*(3); tree.*insert*(7); tree.*insert*(2); tree.*insert*(4); tree.*insert*(6); tree.*insert*(8);

cout << "Binary Search Tree created with nodes: 5, 3, 7, 2, 4, 6, 8" << endl; tree.*inOrderTraversal*();

cout << endl;

*while* (true) {

cout << "\nOptions:\n"; cout << "S: Search\n"; cout << "I: Insert\n"; cout << "D: Delete\n";

cout << "P: Print in-order traversal\n"; cout << "Q: Quit\n";

cout << "Enter your choice: "; cin >> choice;

*switch* (choice) {

*case* 'S':

*case* 's':

cout << "Input value to search: "; cin >> inp;

*if* (tree.*search*(inp)) {

cout << inp << " found in the tree." << endl;

} *else* {

cout << inp << " not found in the tree." << endl;

}

*break*;

*case* 'I':

*case* 'i':

cout << "Input value to insert: "; cin >> inp;

tree.*insert*(inp);

cout << inp << " inserted into the tree." << endl;

*break*;

*case* 'D':

*case* 'd':

cout << "Input value to delete: "; cin >> inp; tree.*deleteNode*(inp);

cout << inp << " deleted from the tree." << endl;

*break*;

*case* 'P':

*case* 'p':

cout << "In-order Traversal: "; tree.*inOrderTraversal*(); *break*;

*case* 'Q':

*case* 'q':

*return* 0;

*default*:

cout << "Invalid choice. Try again." << endl;

*break*;

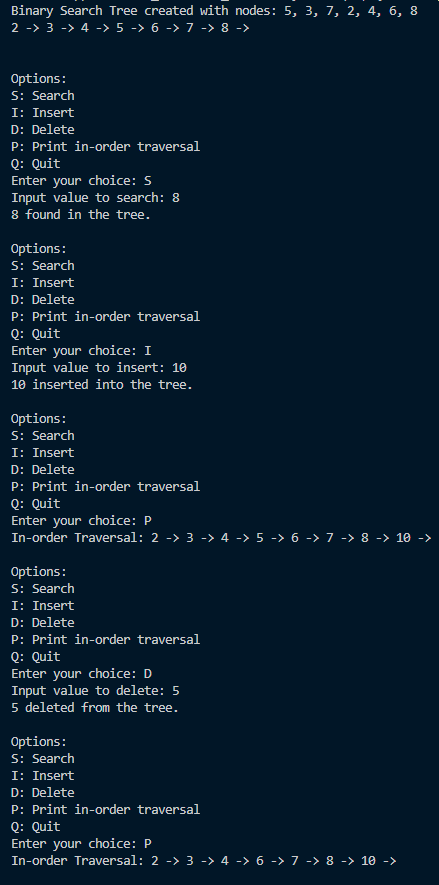
}

}

*return* 0;

}

### Input and Output: Task1:



* 1. **Observations and learning [w.r.t. all tasks]:**

In this experiment, we successfully implemented the insertion and deletion operations on a Binary Search Tree (BST). The program adheres to the basic properties of a BST, where the left subtree contains nodes with keys lesser than the root, and the right subtree contains nodes with keys greater than the root. The insert() and deleteNode() functions were implemented to handle the following cases efficiently:

* + 1. **Insertion**: The insertion function recursively finds the correct position in the tree and inserts a new node as a leaf in its appropriate place.
    2. **Deletion**: The deletion function addresses three distinct cases:
       - **Case 1**: Deletion of a node with no children (a leaf node).
       - **Case 2**: Deletion of a node with one child (either left or right).
       - **Case 3**: Deletion of a node with two children, which involves replacing the node with its in-order successor (the smallest node in its right subtree) and deleting the successor.

The program also provides functionality to perform in-order traversal, which verifies the correctness of the tree's structure after insertion and deletion operations.

### 4 Conclusion:

By completing this experiment, we gained a clear understanding of how to implement and manage the insertion and deletion operations in a Binary Search Tree. The handling of special cases during deletion (such as nodes with one or two children) enhances the robustness of the tree structure. The experiment provided practical insights into tree-based algorithms, strengthening our knowledge of data structures and their operations. After completion, students are equipped to implement efficient binary search trees with insertion and deletion functionalities.

### B.5 Question of Curiosity

Write algorithm to perform insertion and deletion (case 3: Node with 2 child nodes)

#### Algorithm for Insertion in a Binary Search Tree (BST):

##### Steps for Insertion:

1. **Start at the root node** of the BST.
2. **Compare the value** to be inserted with the current node's data.
   * If the value is **less than** the current node's data:
     + Move to the **left child**.
   * If the value is **greater than** the current node's data:
     + Move to the **right child**.
3. **Repeat the comparison** until you reach a nullptr (an empty position in the tree).
4. Insert the new node in that position as a **leaf node**.

#### Algorithm for Deletion in a Binary Search Tree (Case 3: Node with Two Children):

##### Steps for Deleting a Node with Two Children:

1. **Start at the root node** of the BST.
2. **Search for the node** to be deleted by comparing its value with the current node's data.
   * If the value is **less than** the current node’s data, move to the **left child**.
   * If the value is **greater than** the current node’s data, move to the **right child**.
   * If the value matches the current node, this is the node to be deleted.
3. **Check if the node has two children** (i.e., both left and right subtrees are non-null).

##### Find the in-order successor:

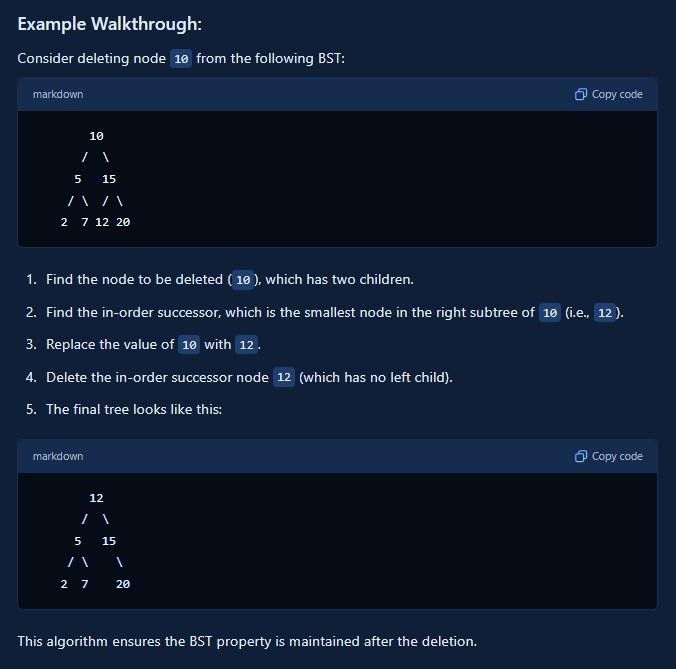
* + The in-order successor is the **smallest node in the right subtree** of the node to be deleted.
  + Start from the node's right child and keep moving to the left child until you find a node with no left child.

1. **Replace the value** of the node to be deleted with the value of the in-order successor.

##### Delete the in-order successor node:

* + Since the in-order successor will either have no children or one child, delete it following the appropriate deletion case (no child or one child).

1. **Return the updated tree.**



## Experiment 9

(PART B : TO BE COMPLETED BY STUDENTS)

***(Students must submit the soft copy as per following segments within two hours of the practical. The soft copy must be uploaded on the Blackboard or emailed to the concerned lab in charge faculties at the end of the practical in case the there is no Black board access available)***

|  |  |
| --- | --- |
| Roll No. C013 | Name: Ashmit Jain |
| Class: B | Batch: B1 |
| Date of Experiment: | Date of Submission |
| Grade : | Time of Submission: |
| Date of Grading: |  |

### Software Code written by student:

***(Paste your code completed during the 2 hours of practical in the lab here)***

### Task1:

#include <iostream> #include <queue> #include <stack>

#define MAX\_VERTICES 100 // Define the maximum number of vertices

using namespace std;

class Graph {

int V; // Number of vertices

int adj[MAX\_VERTICES][MAX\_VERTICES]; // Adjacency matrix representation bool visited[MAX\_VERTICES]; // Visited array for traversals

public:

Graph(int V); // Constructor

void addEdge(int v, int w); // Function to add an edge to the graph void BFS(int start); // Function to perform BFS

void DFS(int start); // Function to perform DFS void DFSUtil(int v); // Utility function for DFS

};

// Constructor Graph::Graph(int V) {

this->V = V;

// Initialize the adjacency matrix and visited array for (int i = 0; i < V; i++) {

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

adj[i][j] = 0; // No edges initially

}

visited[i] = false; // Not visited initially

}

}

// Function to add an edge to the graph void Graph::addEdge(int v, int w) {

adj[v][w] = 1; // Add edge v -> w

adj[w][v] = 1; // Add edge w -> v (undirected graph)

}

// Function to perform BFS void Graph::BFS(int start) {

queue<int> q;

fill(begin(visited), begin(visited) + V, false); // Reset visited array

visited[start] = true; q.push(start);

cout << "BFS Traversal: ";

while (!q.empty()) { int v = q.front(); cout << v << " "; q.pop();

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

if (adj[v][i] == 1 && !visited[i]) { // If there's an edge and not visited visited[i] = true;

q.push(i);

}

}

}

cout << endl;

}

// Utility function for DFS void Graph::DFSUtil(int v) {

visited[v] = true; cout << v << " ";

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

if (adj[v][i] == 1 && !visited[i]) {

DFSUtil(i);

}

}

}

// Function to perform DFS void Graph::DFS(int start) {

fill(begin(visited), begin(visited) + V, false); // Reset visited array cout << "DFS Traversal: ";

DFSUtil(start); cout << endl;

}

int main() {

Graph g(5); // Create a graph with 5 vertices

g.addEdge(0, 1);

g.addEdge(0, 2);

g.addEdge(1, 3);

g.addEdge(1, 4);

g.addEdge(2, 4);

g.BFS(0); // Perform BFS starting from vertex 0 g.DFS(0); // Perform DFS starting from vertex 0

return 0;

}

### Input and Output:

***(Paste your program input and output in following format, If there is error then paste the specific error in the output part. In case of error with due permission of the faculty extension can be given to submit the error free code with output in due course of time. Students will be graded accordingly.)***

***Input;***

g.addEdge(0, 1);

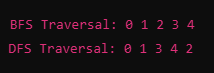
g.addEdge(0, 2);

g.addEdge(1, 3);

g.addEdge(1, 4);

g.addEdge(2, 4);

***Ouput:***



### Observations and learning [w.r.t. all tasks]:

***(Students are expected to comment on the output obtained with clear observations and learning for each task/ sub part assigned)***

##### Traversal Order:

* + - * **BFS** explores layer by layer, while **DFS** goes deep along a branch before backtracking.

##### Output Differences:

* + - * The sequence of visited nodes varies between BFS and DFS due to their distinct exploration strategies.

##### Pathfinding:

* + - * BFS guarantees the shortest path in unweighted graphs, whereas DFS does not.

##### Memory Usage:

* + - * BFS generally uses more memory (queue) than DFS (stack or recursion), especially in dense graphs.

##### Use Cases:

* + - * BFS is suited for shortest path and level-order tasks; DFS is better for complete explorations, topological sorting, and cycle detection.

### 4 Conclusion:

*(****Students must write the conclusion as per the attainment of individual outcome listed above and learning/observation noted in section B.3)***

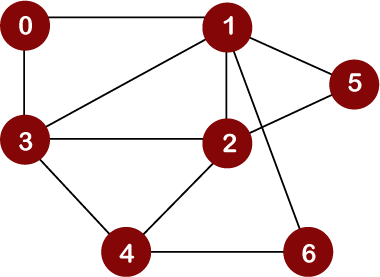
*BFS and DFS are crucial for graph traversal, each with unique strengths. BFS is ideal for shortest path problems, while DFS is useful for deep explorations. Choosing between them depends on the specific needs of the problem, such as path length requirements or memory efficiency.*

4o mini

### B.5 Question of Curiosity

***(To be answered by student based on the practical performed and learning/observations)***

Give BFS and DFS traversals for the following graph



#### Breadth-First Search (BFS)

BFS uses a queue and explores each level of neighbors before going deeper. The BFS traversal of the graph starting from Node 0 would be:

1. Start from Node 0.
2. Visit Nodes connected to 0: [1, 3].
3. Visit Nodes connected to 1 and 3: [2, 4].
4. Visit Nodes connected to 2 and 4: [5, 6].

**BFS Order:** 0 -> 1 -> 3 -> 2 -> 4 -> 5 -> 6

#### Depth-First Search (DFS)

DFS uses a stack (or recursion) and goes as deep as possible along each branch before backtracking. The DFS traversal of the graph starting from Node 0 would be:

1. Start from Node 0.
2. Go to Node 1.
3. Go to Node 2.
4. Go to Node 5.
5. Backtrack to Node 2, then go to Node 3.
6. Go to Node 4.
7. Go to Node 6.

**DFS Order:** 0 -> 1 -> 2 -> 5 -> 3 -> 4 -> 6

## Experiment 10

(PART B : TO BE COMPLETED BY STUDENTS)

|  |  |
| --- | --- |
| Program: BTI CE | Sem: VII |
| Roll No. C013 | Name: Ashmit Jain |
| Division: B | Batch : B1 |
| Date of Experiment: | Date of Submission: 28/10/2024 |
| Grade : |  |

### Software Code written by student:

***(Paste your code completed during the 2 hours of practical in the lab here)***

##### Binary Search Code:

#include <iostream> using namespace std;

bool BinarySearch(int arr[], int low, int high, int key)

{

if(low<=high)

{

int mid = (low + high) / 2; if(arr[mid] == key)

{

return true;

}

else if(arr[mid] > key)

{

return BinarySearch(arr, low, mid-1, key);

}

else

{

return BinarySearch(arr, mid+1, high, key);

}

}

return false;

}

int main()

{

int n;

cout << "Enter the number of elements: "; cin >> n;

int arr[n];

cout << "Enter the elements: "; for(int i=0; i<n; i++)

{

cin >> arr[i];

}

int key;

cout << "Enter the element to be searched: "; cin >> key;

int low = 0, high = n-1;

if ( BinarySearch(arr, low, high, key) == true )

{

cout << "Element found" << endl;

}

else

{

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

}

return 0;

}

**Insertion Sort Code:** #include <iostream> using namespace std;

void insertionSort(int arr[], int n)

{

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

{

int key = arr[i]; int j = i - 1;

// Move elements of arr[0...i-1] that are greater than key

// one position ahead to make space for the key while (j >= 0 && arr[j] > key)

{

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

}

arr[j + 1] = key;

}

}

int main()

{

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

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

// Print the sorted array directly in main cout << "Sorted array: ";

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

{

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

}

cout << endl; return 0;

}

Merge Sort

#include <iostream> using namespace std;

void merge(int arr[], int left, int mid, int right) {

int n1 = mid - left + 1; // Size of the left subarray int n2 = right - mid; // Size of the right subarray

int leftArr[n1], rightArr[n2];

// Copy data to temporary arrays

for (int i = 0; i < n1; i++) leftArr[i] = arr[left + i];

for (int j = 0; j < n2; j++) rightArr[j] = arr[mid + 1 + j];

// Merge the temporary arrays back into arr[left...right] int i = 0, j = 0, k = left;

while (i < n1 && j < n2) {

if (leftArr[i] <= rightArr[j]) { arr[k] = leftArr[i];

i++;

} else {

arr[k] = rightArr[j]; j++;

} k++;

}

// Copy any remaining elements of leftArr, if any while (i < n1) {

arr[k] = leftArr[i]; i++;

k++;

}

// Copy any remaining elements of rightArr, if any

while (j < n2) { arr[k] = rightArr[j]; j++;

k++;

}

}

void mergeSort(int arr[], int left, int right) { if (left < right) {

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

// Sort first and second halves mergeSort(arr, left, mid); mergeSort(arr, mid + 1, right);

// Merge the sorted halves merge(arr, left, mid, right);

}

}

int main() {

int arr[] = {12, 11, 13, 5, 6, 7};

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

mergeSort(arr, 0, n - 1);

// Print the sorted array cout << "Sorted array: "; for (int i = 0; i < n; i++) cout << arr[i] << " ";

cout << endl;

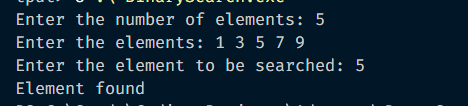
return 0;

}

### Input and Output:

***(Paste your commented program input and output in following format, If there is error then paste the specific error in the output part. In case of error with due permission of the faculty extension can be given to submit the error free code with output in due course of time.)***

**Binary Search**



**Insertion Sort**



* 1. **Observations and learning:**

***(Students are expected to comment on the output obtained with clear observations and learning for each task/ sub part assigned)***

##### Binary Search:

* + - Efficient for large, sorted datasets with O(log⁡N)O(\log N)O(logN) complexity, but requires the array to be pre-sorted.
    - Reduces search space quickly, ideal for static data where minimal changes occur.

##### Insertion Sort:

* + - Simple, intuitive, and works well for small or nearly sorted datasets with O(n2)O(n^2)O(n2) complexity.
    - Stable and adaptive, making it efficient in cases with partially sorted data.

##### Merge Sort:

* + - Consistent O(nlog⁡n)O(n \log n)O(nlogn) performance, suitable for large datasets.
    - Uses additional memory for merging, but is stable and reliable, especially for linked lists or large arrays.

### Conclusion:

*(****Students must write the conclusion as per the attainment of individual outcome listed above and learning/observation noted in section B.3)***

* + - **Binary Search** is optimal for static, sorted arrays due to its efficiency but requires pre-sorted data.
    - **Insertion Sort** is practical for small datasets and partially sorted data due to its simplicity and adaptability.
    - **Merge Sort** is a robust, stable choice for large datasets but uses more memory.

### Question of Curiosity

**Show the working of Mergesort on following numbers**

**-8, 5, 1, 0, 5, 26, 47, 10,99, 23**

#### Merge Sort Steps:

1. **Initial Array**: [-8, 5, 1, 0, 5, 26, 47, 10, 99, 23]
2. **Divide the Array** (Repeatedly split the array in half until each sub-array has one element):

o Split 1: [-8, 5, 1, 0, 5] and [26, 47, 10, 99, 23]

o Split 2: [-8, 5] and [1, 0, 5] | [26, 47] and [10, 99, 23]

o Split 3: [-8] and [5] | [1] and [0, 5] | [26] and [47] | [10] and [99, 23]

o Split 4: [0] and [5] | [99] and [23]

1. **Merge Process** (Sort and merge sub-arrays back together):
   * **Merge Step 1**: Merge [-8] and [5] → [-8, 5]
   * **Merge Step 2**: Merge [0] and [5] → [0, 5]
   * **Merge Step 3**: Merge [1] and [0, 5] → [0, 1, 5]
   * **Merge Step 4**: Merge [-8, 5] and [0, 1, 5] → [-8, 0, 1, 5, 5]
   * **Merge Step 5**: Merge [26] and [47] → [26, 47]
   * **Merge Step 6**: Merge [99] and [23] → [23, 99]
   * **Merge Step 7**: Merge [10] and [23, 99] → [10, 23, 99]
   * **Merge Step 8**: Merge [26, 47] and [10, 23, 99] → [10, 23, 26, 47, 99]
   * **Merge Step 9**: Merge [-8, 0, 1, 5, 5] and [10, 23, 26, 47, 99] → [-8, 0, 1, 5, 5, 10, 23, 26,

47, 99]

##### Final Sorted Array: [-8, 0, 1, 5, 5, 10, 23, 26, 47, 99]

*(To be answered by student base****d on the practical performed and learning/observations)***

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*