UOW MALAYSIA KDU PENANG UNIVERSITY COLLEGE

PART OF THE UNIVERSITY OF WOLLONGONG AUSTRALIA GLOBAL NETWORK

ASSIGNMENT COVER PAGE

Programme	**Course Code and Title**
Diploma in Computer Studies Diploma in Information Technology	DDA1224 Data Structures and Algorithms DDA1224N Data Structures and Algorithms
Student's name / student's id	**Lecturer's name**
• 0204677 LIM ZHE YUAN • 0205096 THOR WEN ZHENG • 0204144 ADAM JOHN SIMPSON • 0205430 TAN PENG HENG	Rosmah Ismail

Date issued	**Submission Deadline**	**Indicative Weighting**
Week 5 - 30/6/2021	Week 9 - 30/7/2021	20%
Assignment [2]	Assignment 2 – group of 4 members	

This assessment assesses the following course learning outcomes

# as in Course Guide	**UOWM KDU Penang University College Learning Outcome**
CLO2	Apply the use of basic data structures in applications.

Student's declaration

I certify that the work submitted for this assignment is my own and research sources are fully acknowledged.

Student's signature: ZHE YUAN

Submission Date: 30/7/2021

TABLE OF CONTENTS

INTRODUCTION OF DATA STRUCTURES

A data structure is a logical arrangement or collection of data, combined with a set of operations that can be performed on the data. Data structures are used together with their operations to access, store, manage, manipulate, and organize data in an efficient and effective manner. Data structures are important as they make it easier to store and manage large amounts of data, thus saving time and effort, and reducing costs. Data structures are also considered abstract data types (ADT), which means their implementation details are hidden. Hence, the abstraction can be convenient when people use ADTs, as they do not need to know how the ADTs are coded internally, they only need to consider the behaviour and operations of the ADTs.

Data structures can generally be divided into 2 categories, linear and non-linear. Linear data structures are data structures that store elements in sequential order, such as arrays, stacks, queues, and linked lists; non-linear data structures are data structures that store elements in a hierarchical manner, such as graphs and trees.

In this research, the data structures that will be covered are:

- Linked List
- Stack
- Queue
- Binary Trees

1.0 LINKED LIST

1.1 DEFINITION

A linked list is a linear data structure which stores a collection of data elements that are linked to each other. A linked list is also dynamic; thus, its size can freely shrink and grow throughout the runtime of a program. Each element in a linked list is a structure variable called a node. Nodes typically have two components: one that stores information and one which is a pointer that stores the address of the next node in the list. To keep track of the start of the list, a pointer, typically called “head”, is used to point to the first node.

There are three common types of linked lists including singly, doubly, and circular linked list. The singly linked list is the most basic and common, with each node having some data and a pointer to the next node. For doubly linked list, each node has an additional pointer that points to the previous node. In a circular linked list, the last node points to the first node to form a circular loop. In this research, only the **singly linked list** will be covered.

***Figure 1.1.1*:** Graphical representation of a singly linked list (Programiz, 2021)

***Figure 1.1.2*:** Graphical representation of a doubly linked list (Programiz, 2021)

***Figure 1.1.3*:** Graphical representation of a circular linked list (Programiz, 2021)

1.2 OPERATIONS OF LINKED LIST

The basic operations of a linked list include traversal, searching, insertion, and deletion.

i. Traversal

Traversal is the process of going through and accessing every element of a linked list. This is typically used to display all the data in the list. Firstly, a temporary node pointer is declared and points to the first node. While the temporary pointer is not pointing to NULL, the data of the current node can be printed if desired, then the temporary pointer is assigned the address of the next node. This moves the temporary pointer through every node in the list until it reaches the end of the list, when it points to NULL.

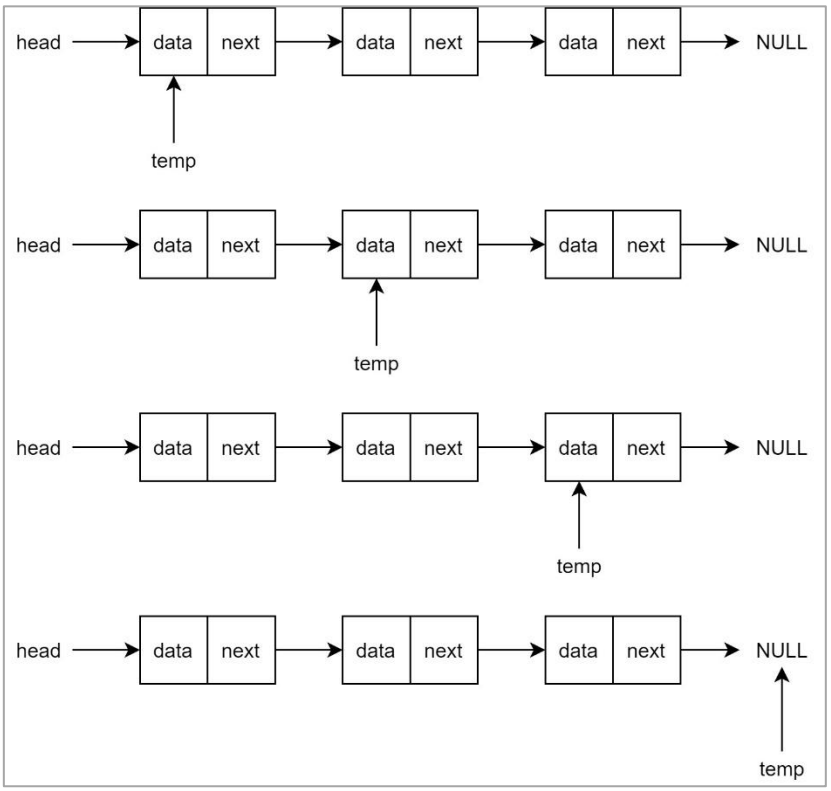

***Figure 1.2.1*:** Illustration of traversal in a linked list

ii. Searching

Searching is the process of finding specific elements in a linked list. It utilizes traversal to access every element in the list and compares the data of every node with the data being searched until the desired data is found or the end of the list is reached.

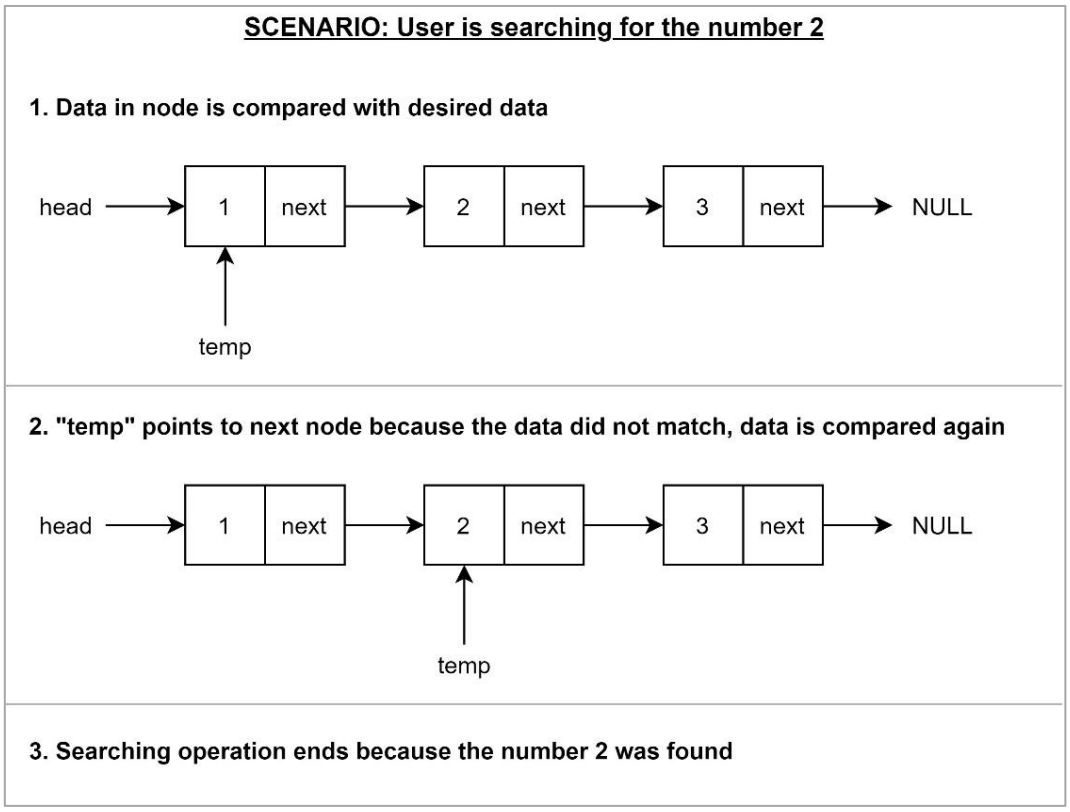

***Figure 1.2.2*:** Illustration of searching operation in a linked list

iii. Insertion

Insertion is the process of adding elements into a linked list, either at the front, end, or middle of the list. In practical use, elements are usually added at the frond or end of a list. Firstly, memory is allocated for a new node and the address is assigned to a node pointer. For insertion at the front, the new node points to the first node and head will point to the new node. For insertion at the end, the new node points to NULL and the current last node will point to the new node.

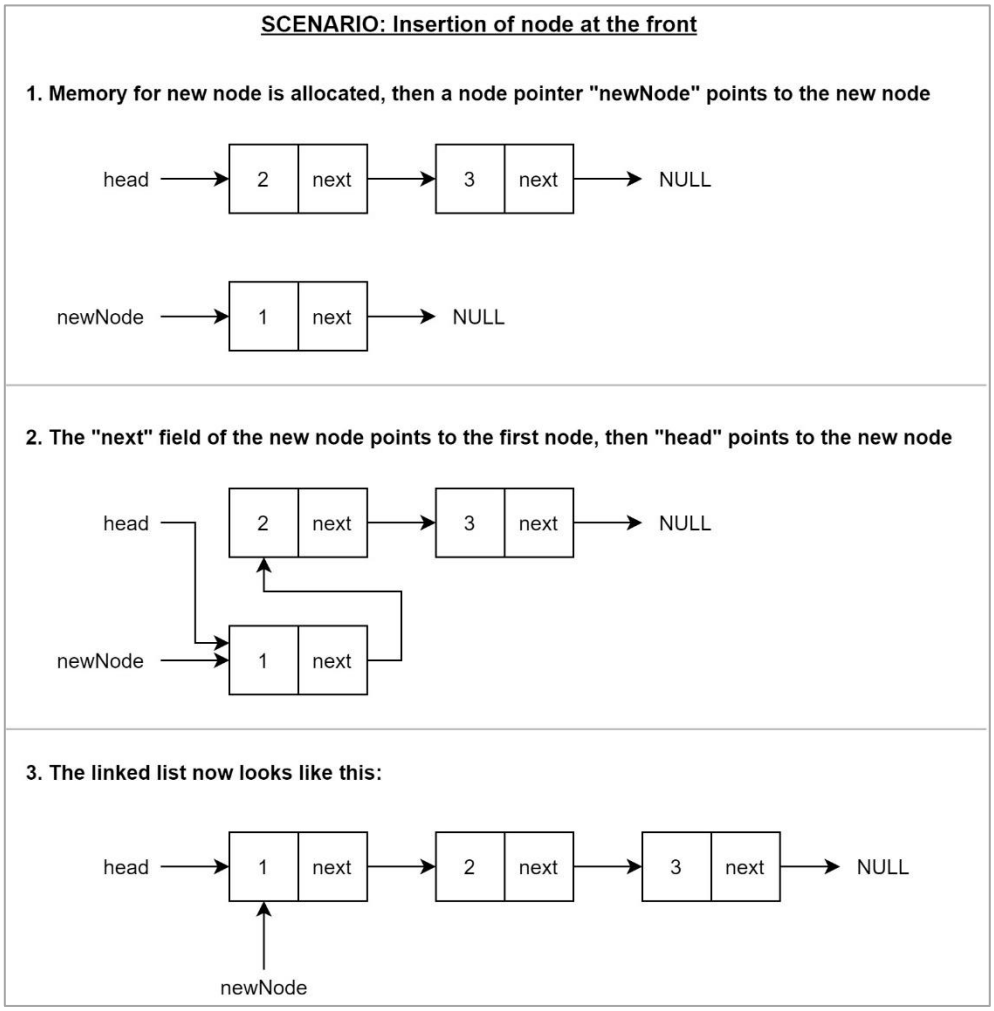

***Figure 1.2.3*:** Illustration of insertion operation at the front of a linked list

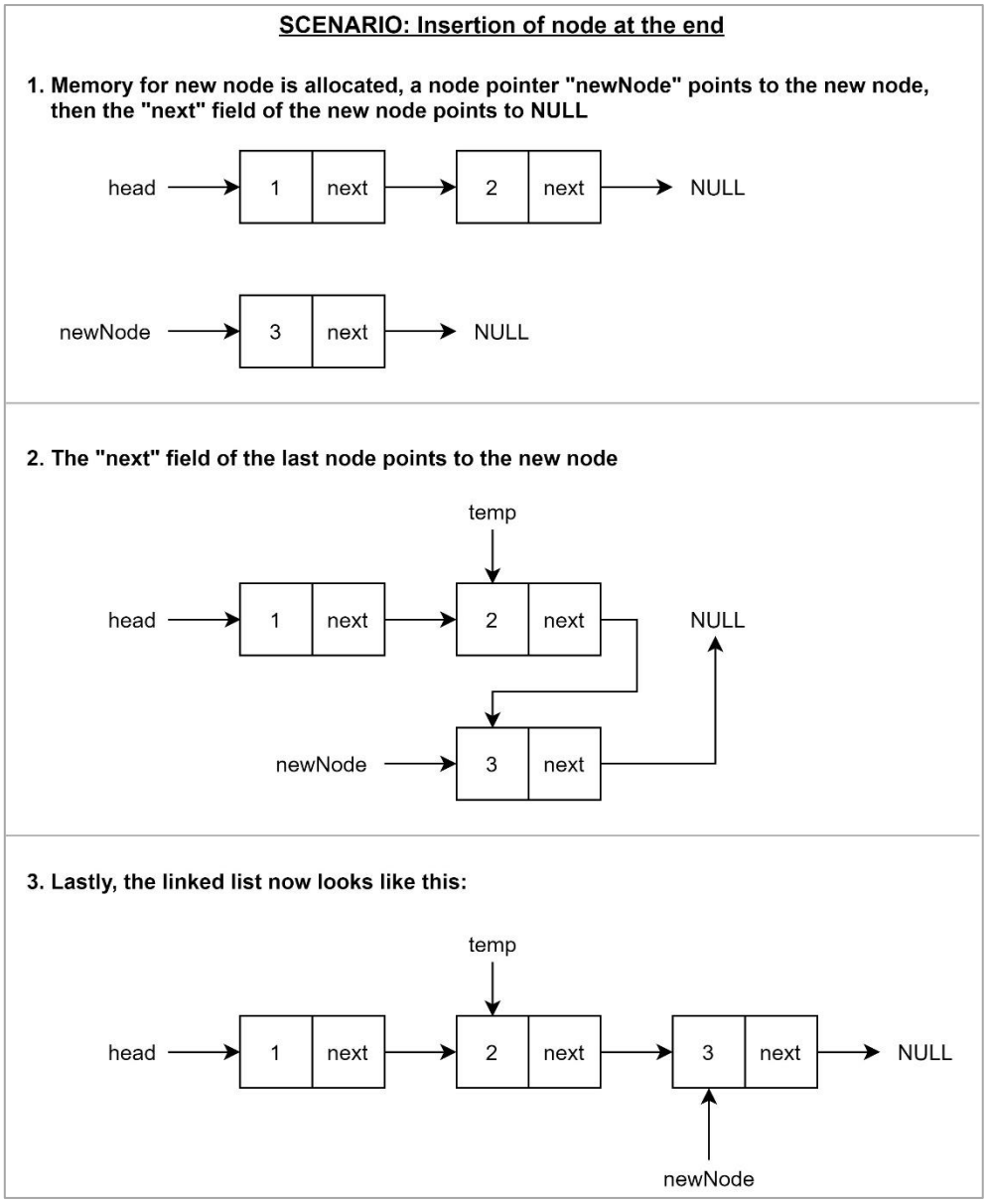

Figure 1.2.4: Illustration of insertion operation at the end of a linked list

iv. Deletion

Deletion is the process of removing specific elements from a linked list. It utilizes the search operation to locate specific elements to delete. Once the element is found, a temporary pointer points at the node to be deleted, the previous node points to the node after the node to be deleted. Then, the memory allocated for the node to be deleted is deallocated using the delete operator.

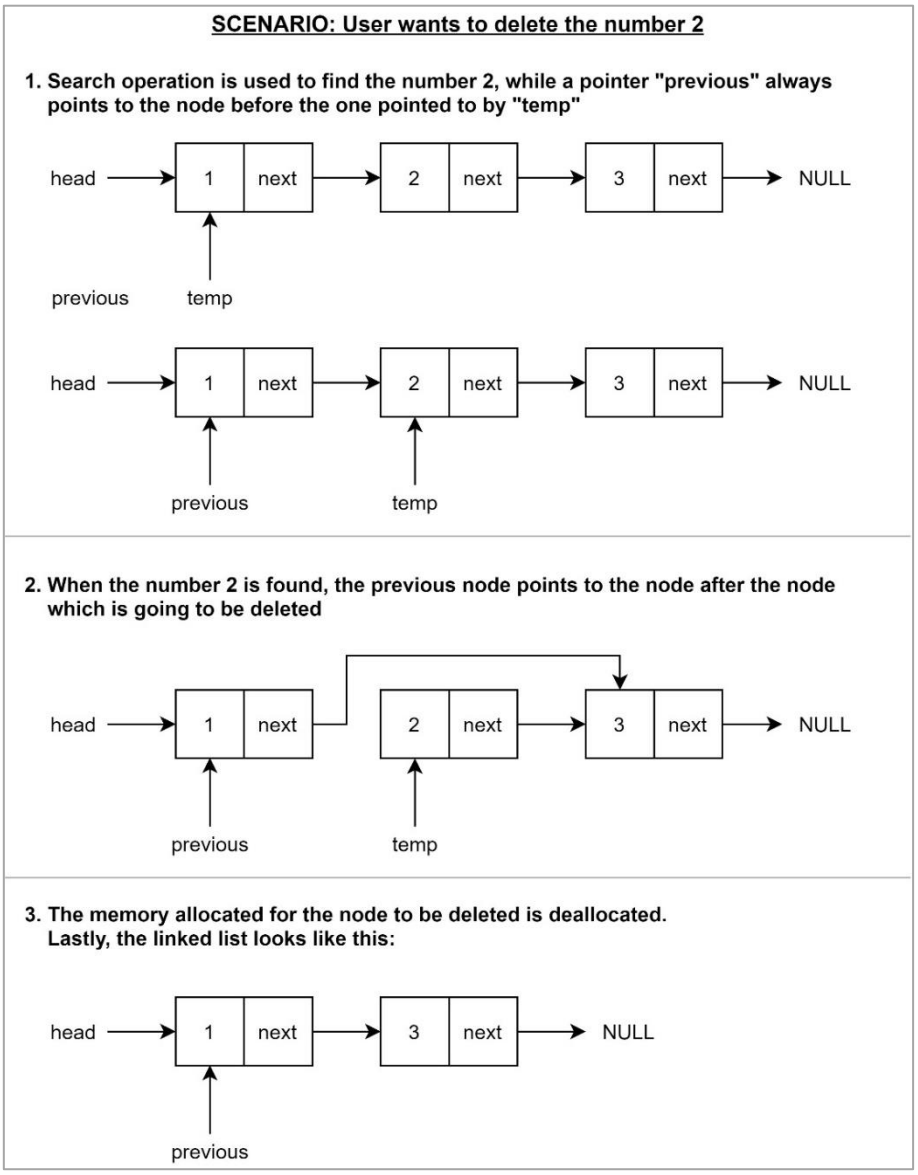

***Figure 1.2.5*:** Illustration of deletion operation in a linked list

1.3 IMPLEMENTATION OF LINKED LIST IN C++

- **Program Code**

i. Node structure definition

```
// Node structure
struct Node {
    int data;
    struct Node *next;
};
```

***Figure 1.3.1*:** Code for a basic node structure in singly linked lists

ii. Display function using Traversal

```
void display(Node *head) {
    // Check if list is empty
    if (isEmpty(head)) {
        cout << "\n* * * The linked list is currently empty * * *\n";
        return;
    }

    // Print out all elements in the list
    Node *current = head;
    cout << "\nElements in linked list: ";
    while (current != NULL) {
        cout << current->data << " ";
        current = current->next;
    }
    cout << endl;
}
```

***Figure 1.3.2*:** Code for display function in linked list using traversal

iii. Search function

```
void searchNode(Node *head, int data) {
    int index = 0;
    Node *current;

    // Check if list is empty
    if (isEmpty(head)) {
        cout << "\n* * * The linked list is currently empty * * *\n";
        return;
    }

    // Traverse list to find entered data
    current = head;
    while (current != NULL && current->data != data) {
        current = current->next;
        index++;
    }

    // Check if matching node was found
    if (current == NULL) {
        cout << "\n* * * Search failed: No nodes with data " << data << " were found * * *\n";
    } else {
        cout << "\n- - - Node with data " << data << " was found at index " << index << " - - -\n";
    }
}
```

***Figure 1.3.3*:** Code for searching operation in linked list

iv. Insertion function

```
void addNode(Node **head, int data) {
    char position;
    Node *current, *newNode;

    do {
        // Determine if user wants to add at the Front or End of list
        cout << "\nDo you want to add at the Front or End of the list? (F/E): ";
        cin >> position;
        position = toupper(position);
        cin.ignore(numeric_limits<streamsize>::max(), '\n');
        if (position != 'F' && position != 'E')
            cout << "\n* * * Invalid input. Please enter 'F' or 'E' * * *\n";
    } while (position != 'F' && position != 'E');

    // Add node at Front or End based on user's choice
    newNode = new Node;
    newNode->data = data;

    if (position == 'F') {
        newNode->next = *head;
        *head = newNode;
    } else {
        // Check if list is empty
        if (isEmpty(*head)) {
            newNode->next = *head;
            *head = newNode;
        } else {
            // Traverse til the end of list
            current = *head;
            while (current->next != NULL) {
                current = current->next;
            }

            // Connect new node
            newNode->next = NULL;
            current->next = newNode;
        }
    }

    cout << "\n+ + + Node successfully added + + +\n";
}
```

***Figure 1.3.4*:** Code for insertion operation in a linked list

v. Deletion function

```
void deleteNode(Node **head, int data) {
    Node *prev, *delPtr;

    // Check if list is empty
    if (isEmpty(*head)) {
        cout << "\n* * * The linked list is currently empty * * *\n";
        return;
    }

    // Traverse list to find entered data
    delPtr = *head;
    while (delPtr != NULL && delPtr->data != data) {
        prev = delPtr;
        delPtr = delPtr->next;
    }

    // Check if matching node was found
    if (delPtr == NULL) {
        cout << "\n* * * Deletion failed: No nodes with data " << data << " were found * * *\n";
    } else {
        if (delPtr == *head) {
            *head = (*head)->next;
        } else {
            prev->next = delPtr->next;
        }

        delete delPtr;
        cout << "\n- - - Node with data " << data << " was deleted - - -\n";
    }
}
```

***Figure 1.3.5*:** Code for deletion operation in a linked list

- **Output Produced by Program**

i. Main Menu

```
WELCOME TO LINKED LIST DEMO PROGRAM

------------------------------
            MENU
------------------------------
        1. Add node
        2. Delete node
        3. Search node
        4. Display all
        5. Exit

Selection:
```

***Figure 1.3.6*:** Main menu of the demo program

ii. Insertion operation

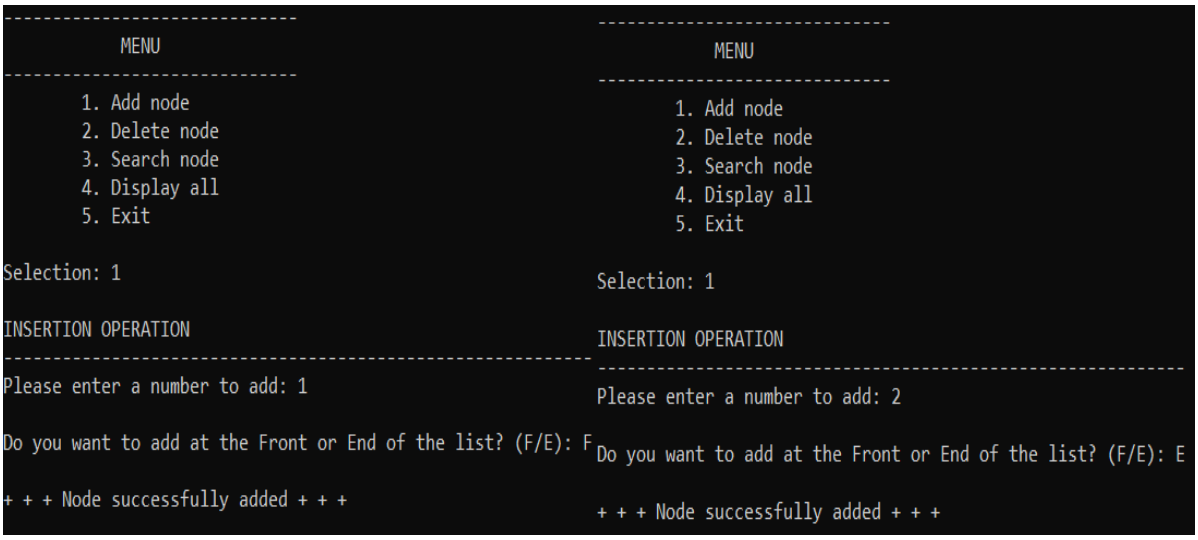

***Figure 1.3.7*:** Process of inserting new data elements into linked list

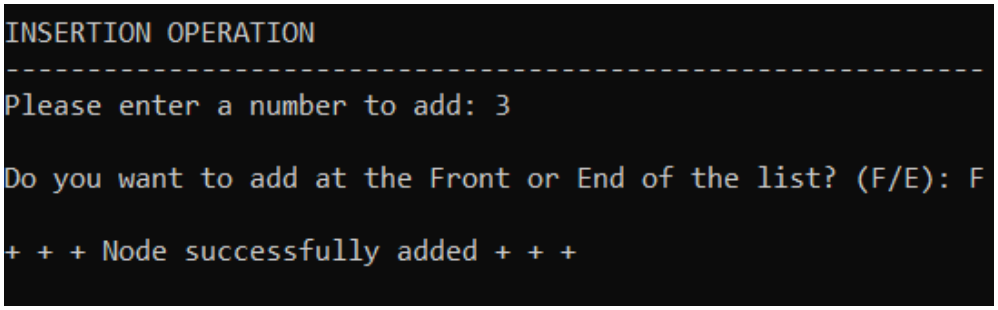

***Figure 1.3.8*:** Inserting new data to the front of a linked list that currently has elements

iii. Display operation

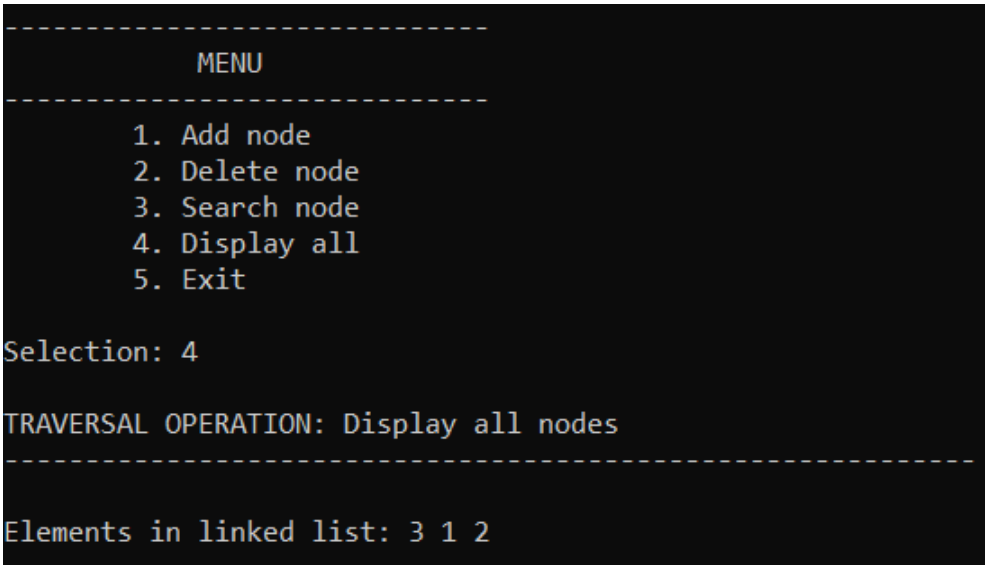

***Figure 1.3.9*:** Display function using traversal operation, the screenshot proves that the elements have been inserted correctly

```
TRAVERSAL OPERATION: Display all nodes
------------------------------------------------------------

* * * The linked list is currently empty * * *
```

***Figure 1.3.10*:** If linked list is empty, the display function will notify the user that the list is empty

iv. Searching operation

```
-----------------------------
           MENU
-----------------------------
       1. Add node
       2. Delete node
       3. Search node
       4. Display all
       5. Exit

Selection: 3

SEARCHING OPERATION
------------------------------------------------------------
Please enter a number to search: 3

- - - Node with data 3 was found at index 0 - - -
```

```
-----------------------------
           MENU
-----------------------------
       1. Add node
       2. Delete node
       3. Search node
       4. Display all
       5. Exit

Selection: 3

SEARCHING OPERATION
------------------------------------------------------------
Please enter a number to search: 2

- - - Node with data 2 was found at index 2 - - -
```

***Figure 1.3.11*:** Process of searching for specific data elements in linked list in a running program

```
SEARCHING OPERATION
------------------------------------------------------------
Please enter a number to search: 6

* * * Search failed: No nodes with data 6 were found * * *
```

***Figure 1.3.12*:** Attempting to search for data that is not in the linked list

v. Deletion operation

```
-----------------------------
           MENU
-----------------------------
       1. Add node
       2. Delete node
       3. Search node
       4. Display all
       5. Exit

Selection: 2

DELETION OPERATION
------------------------------------------------------------
Please enter a number to delete: 2

- - - Node with data 2 was deleted - - -
```

***Figure 1.3.13*:** Process of deleting specific data elements in linked list in a running program

```
Selection: 2

DELETION OPERATION
-----------------------------------------------------------------
Please enter a number to delete: 5

* * * Deletion failed: No nodes with data 5 were found * * *
```

***Figure 1.3.14*:** Attempting to delete data that is not in the linked list

```
Selection: 4

TRAVERSAL OPERATION: Display all nodes
-----------------------------------------------------------------

Elements in linked list: 3 1
```

***Figure 1.3.15*:** Using display function to prove that the data was deleted

2.0 STACK

2.1 DEFINITION

Stack is a linear data structure which follows the Last In First Out (LIFO) order when its operations are performed. This means addition and deletion of data always occurs at the top of the stack, so the last data element that was added to the stack will be deleted first and vice versa. Hence, this works much like managing a stack of plates in real life.

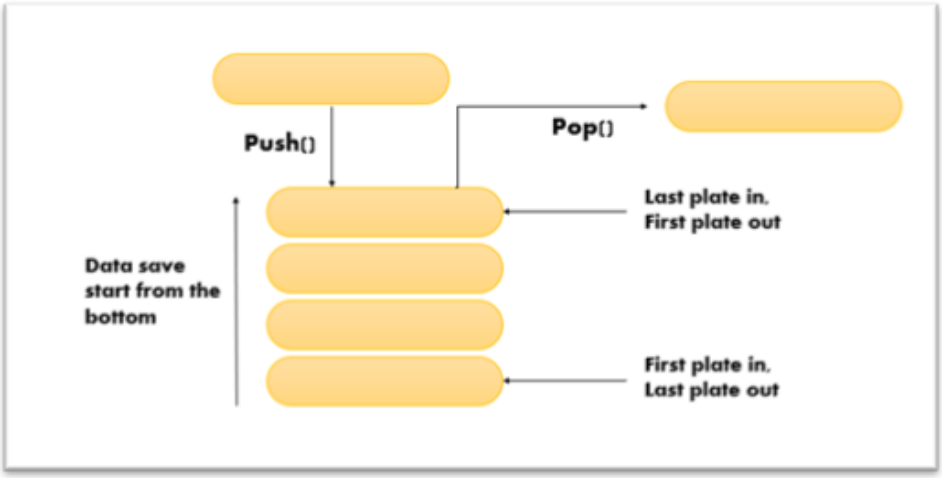

***Figure 2.1.1*:** General concept of a Stack represented by a diagram

There are two ways to implement a stack: in a fixed or dynamic manner. For a fixed stack, **an array** to store data and an integer variable that tracks the index of the next available space in the stack will be used. For a dynamic stack, a data variable and a pointer that points to the address of the next node will be used, and a dedicated top pointer is required to track the topmost node in the stack – much like a linked list. This research will be based on the **fixed stack**.

2.2 OPERATIONS OF STACK

The main operations of a stack include push, pop, isEmpty, isFull, peek, and display.

Terminology:

- top: integer variable that stores the index number of the next available space in the array of a stack

i. Push

Push is used to add a data element into the “top” index of the array in a stack. Each time a new element is added, the value of “top” increments by one. If there is any attempt to push a new element when the stack is full, then it will cause an overflow condition.

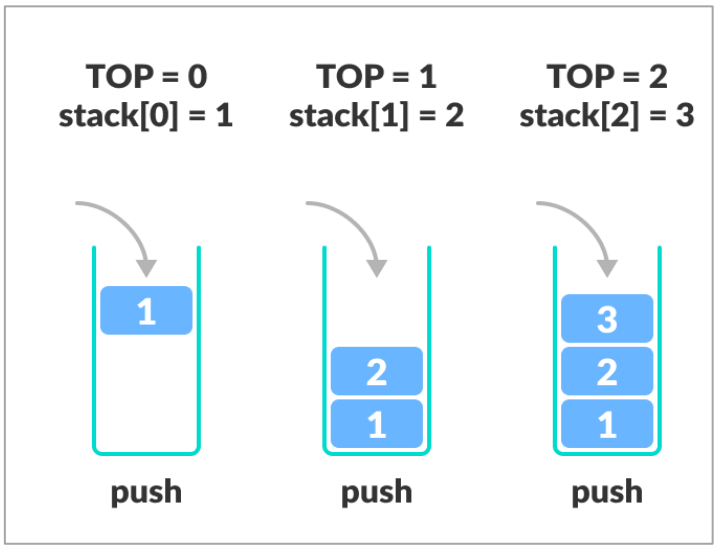

Figure 2.2.1: Push operation in a Stack

ii. Pop

Pop is used to remove a data element from the top of the stack and returns its value. Each time an element is removed, the value of “top” decreases by one. If there is any attempt to pop an element when the stack is empty, then it will cause an underflow condition.

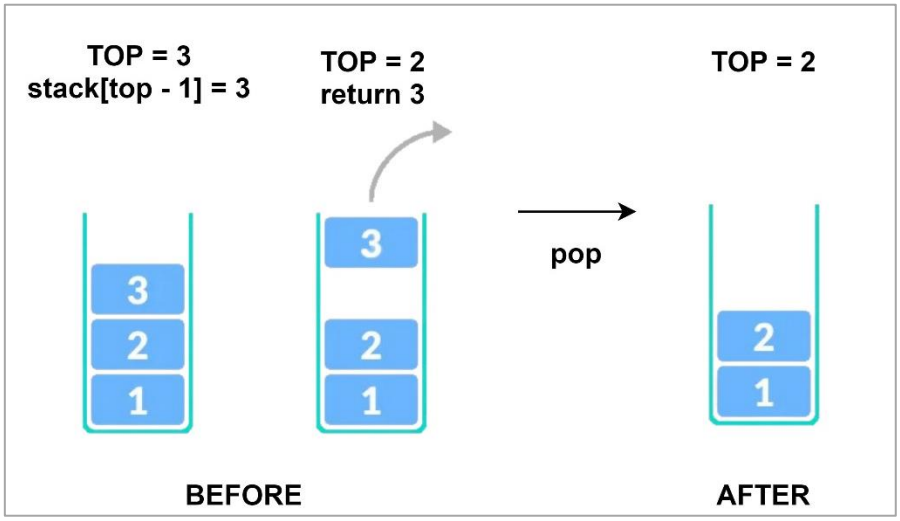

Figure 2.2.2: Pop operation in a Stack

iii. isEmpty

This operation checks whether the stack is empty or not. It returns the boolean value “true’ if the value of “top” is 0, which means the stack is empty. Otherwise, it returns false.

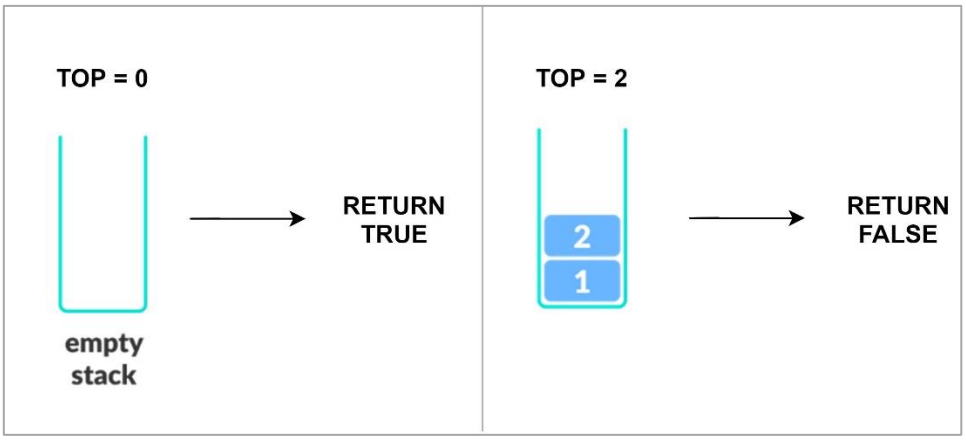

Figure 2.2.3: isEmpty operation in a Stack

iv. isFull

This operation checks whether the stack is full or not. It returns the boolean value “true” if the value of “top” is equal to the capacity of the array, which means the stack is full. Otherwise, it returns “false”.

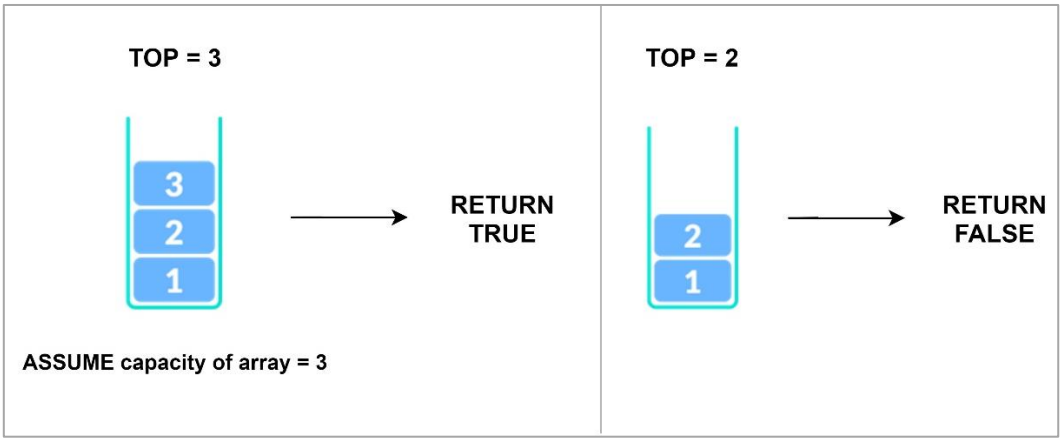

***Figure 2.2.4*:** isFull operation in a Stack

v. Peek

This operation retrieves and returns the value of the top element in the stack if the stack is not empty.

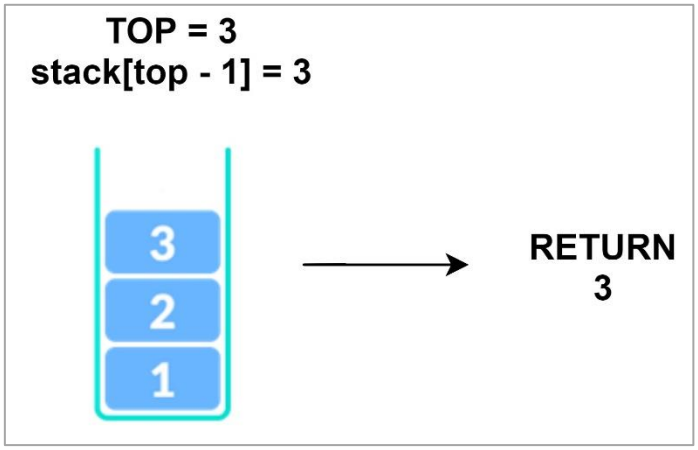

***Figure 2.2.5*: Peek operation**

vi. Display

This operation displays the data stored in every element in the data array of the stack.

2.3 IMPLEMENTATION OF STACK IN C++

- **Program Code**

i. Stack structure definition

```
// Stack structure definition
struct Stack {
    int data[5];
    int top;
};
```

***Figure 2.3.1*: Code for a basic Stack structure in array-based stack**

ii. Stack initialization

```
// Initialize the Stack by setting top to 0
void initializeStack(Stack *stkPtr) {
    stkPtr->top = 0;
}
```

***Figure 2.3.2*: Code for initializing a stack**

iii. Push

```
// Push function (to insert elements)
void push(Stack *stkPtr, int pushData) {
    stkPtr->data[stkPtr->top] = pushData; // Assign new data to top element
    (stkPtr->top)++; // Increment top each time new data is pushed
}
```

***Figure 2.3.3*:** Code for push operation in stack

iv. Pop

```
// Pop function (to delete elements)
int pop(Stack *stkPtr) {
    int popData = stkPtr->data[stkPtr->top - 1]; // Get data from element to be popped
    (stkPtr->top)--; // Decrease top each time data is popped
    return popData; // Return the popped data
}
```

***Figure 2.3.4*:** Code for pop operation in stack

v. isEmpty

```
// Checks if stack is empty
bool isEmpty(Stack *stkPtr){
    // Return true if top is 0; else return false
    if (stkPtr->top == 0)
        return true;
    else
        return false;
}
```

***Figure 2.3.5*:** Code for isEmpty operation in a stack

vi. isFull

```
// Checks if stack is full
bool isFull(Stack *stkPtr){
    // Return true if top is 5; else return false
    if (stkPtr->top == 5)
        return true;
    else
        return false;
}
```

***Figure 2.3.6*:** Code for isFull operation in a stack

vii. Peek

```
// Peek function (retrieves data from top element)
int peek(Stack *stkPtr) {
    int peekData = stkPtr->data[stkPtr->top - 1];
    return peekData;
}
```

***Figure 2.3.7*:** Code for peek operation in a stack

- **Output Produced by Program**

i. Main Menu

```
WELCOME TO STACK DEMO PROGRAM

------- MAIN MENU --------
1. Push
2. Pop
3. Check Empty
4. Check Full
5. Peek
6. Display
7. Exit

Please select a function to proceed:
```

***Figure 2.3.8*:** Main menu of the demo program

ii. Menu input validation

```
Please select a function to proceed: 11

* * * Invalid input, please enter 1 - 7 * * *

Please select a function to proceed:
```

***Figure 2.3.9*:** The program rejects invalid menu inputs

iii. Push operation

```
------- MAIN MENU --------
1. Push
2. Pop
3. Check Empty
4. Check Full
5. Peek
6. Display
7. Exit

Please select a function to proceed: 1

Enter data to push: 1

- - - Data was added successfully - - -
```

```
------- MAIN MENU --------
1. Push
2. Pop
3. Check Empty
4. Check Full
5. Peek
6. Display
7. Exit

Please select a function to proceed: 1

Enter data to push: 2

- - - Data was added successfully - - -
```

***Figure 2.3.10*:** Process of pushing new elements into the stack

```
Please select a function to proceed: 6

Elements in the stack:
 2 <-- top
 1
```

***Figure 2.3.11*:** Using display function to prove the elements were added successfully

```
Please select a function to proceed: 1

------ THE STACK IS FULL ------
```

***Figure 2.3.12*:** If the stack is full, user is notified, and new data is rejected

iv. Pop operation

```
------- MAIN MENU --------
1. Push
2. Pop
3. Check Empty
4. Check Full
5. Peek
6. Display
7. Exit

Please select a function to proceed: 2

- - - Popped value: 2 - - -
```

Figure 2.3.13: Pop operation deletes top element, and returns the deleted data

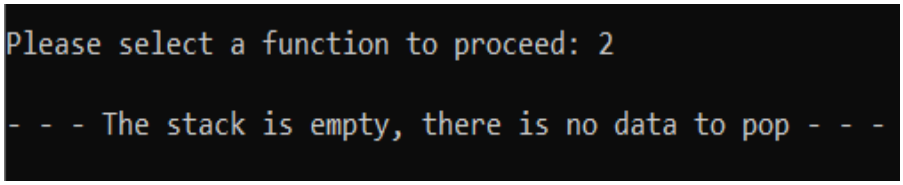

***Figure 2.3.14*:** Attempting to pop data when the stack is empty

v. isEmpty operation

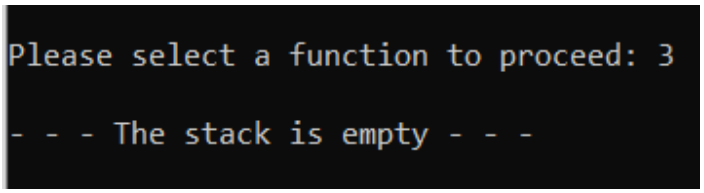

***Figure 2.3.15*:** Output of program when stack is empty

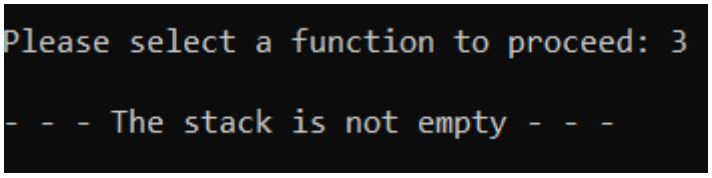

***Figure 2.3.16*:** Output of program when stack is not empty

vi. isFull operation

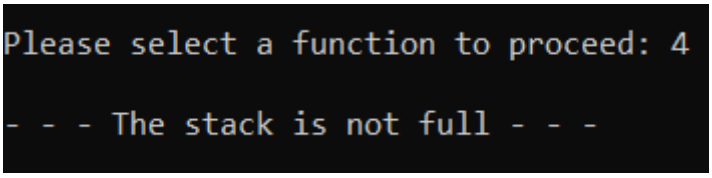

***Figure 2.3.17*:** Output of program when stack is not full

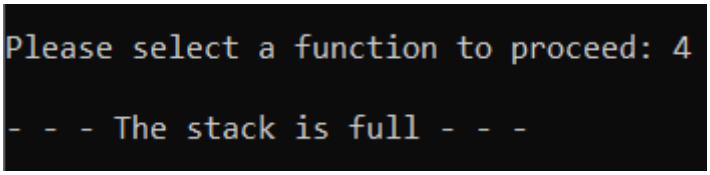

***Figure 2.3.18*:** Output of program when stack is full

vii. Peek operation

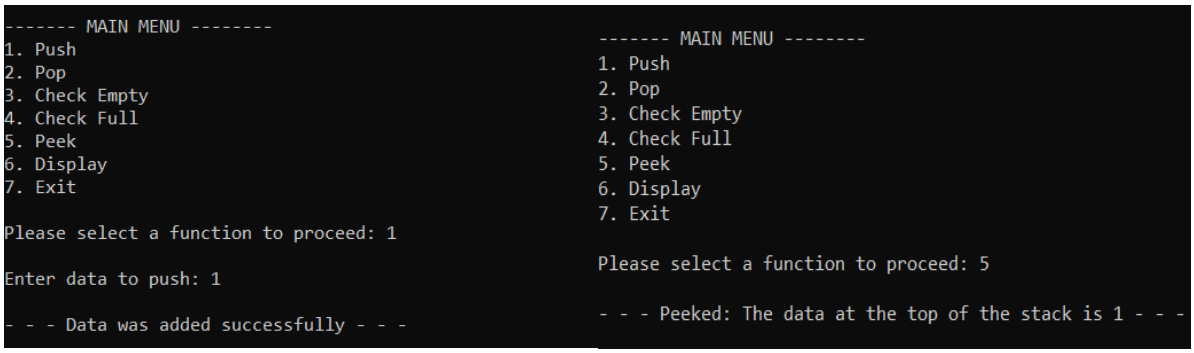

***Figure 2.3.19*:** Peek operation returns the data stored in the topmost element

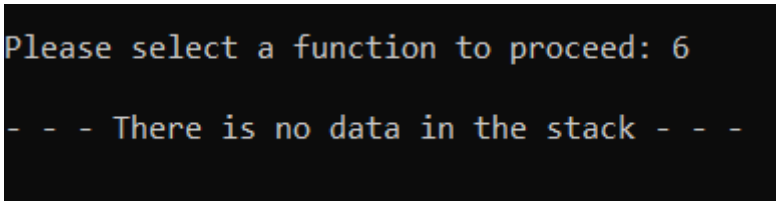

***Figure 2.3.20*:** Output of program if there is no data in the stack to peek at

viii. Display operation

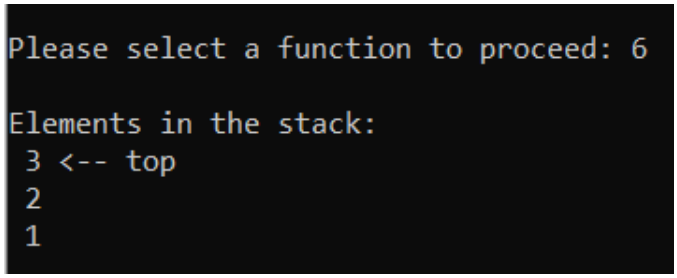

***Figure 2.3.21*:** Display operation prints out all data stored in stack, assuming there are 3 elements

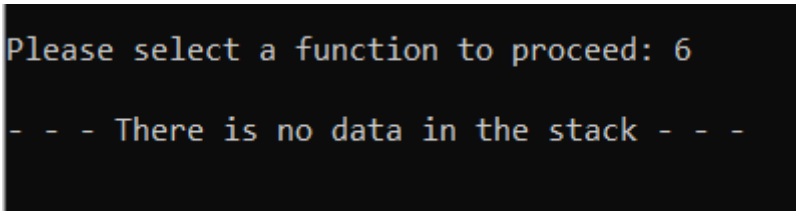

***Figure 2.3.22*:** Output of program if display operation is used when stack is empty

3.0 QUEUE

3.1 DEFINITION

Queue is a linear data structure which follows the First In First Out (FIFO) order when it performs an operation. This means that data elements which were added first are removed first, while elements which were added last are removed last. The concept of queue is applied in everyday-life situations such as lining up to buy food or for checkout at the supermarket cashier counter, both of which apply the concept of first come, first served.

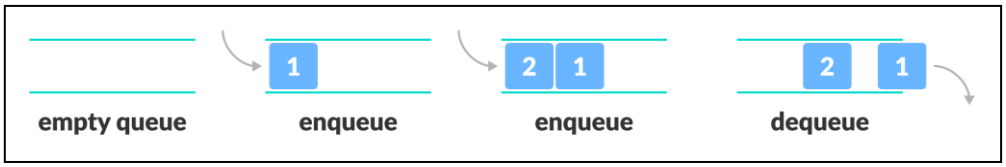

Figure 3.1.1: Graphical representation of FIFO concept in queues (Programiz, 2021)

There are 4 types of queues including simple queue, circular queue, priority queue, and double ended queue. This research will only cover the **simple queue**, and it will be implemented **using a linked list**, which involves the use of node structures as data elements.

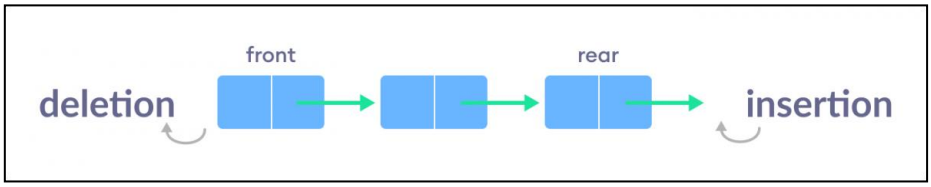

Figure 3.1.2: Graphical representation of a simple queue (Programiz, 2021)

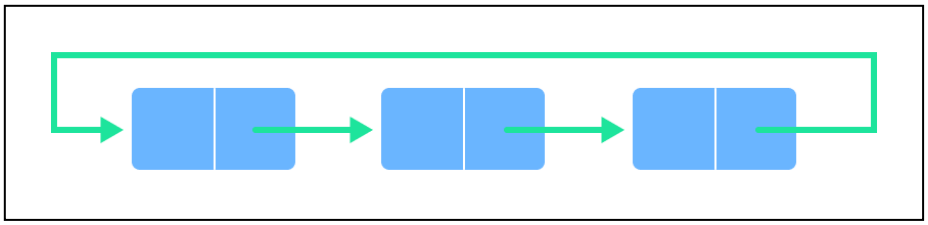

Figure 3.1.3: Graphical representation of a circular queue (Programiz, 2021)

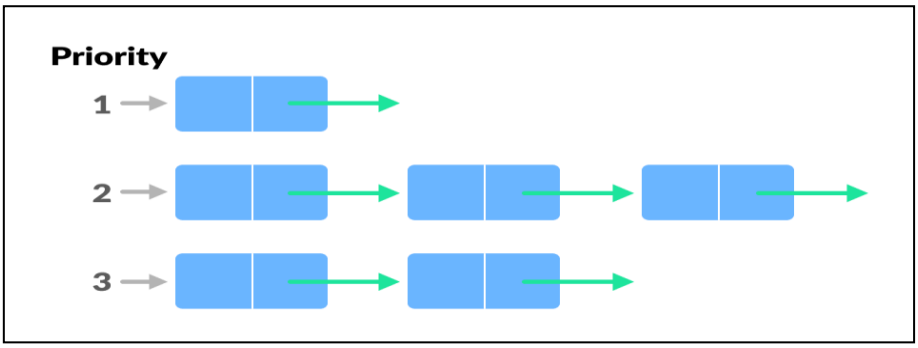

Figure 3.1.4: Graphical representation of a priority queue (Programiz, 2021)

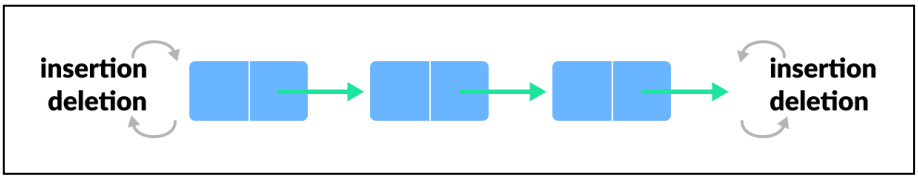

Figure 3.1.5: Graphical representation of a double ended queue (Programiz, 2021)

3.2 OPERATIONS OF QUEUE

In a queue, there are several operations that can be performed. Its basic operations enqueue, dequeue, and other utility operations for including isEmpty, peek and display.

Terminology:

- head/front: the pointer that always points to the first node in the queue
- tail/rear: the pointer that always points to the last node in the queue

i. Enqueue

Enqueue adds an item to the queue by insertion, this is always done at the rear of the queue. When a new node is added, it points to NULL, while “rear” and the last node will both point to the new node. If the queue is empty, both “front” and “rear” will point to the new node.

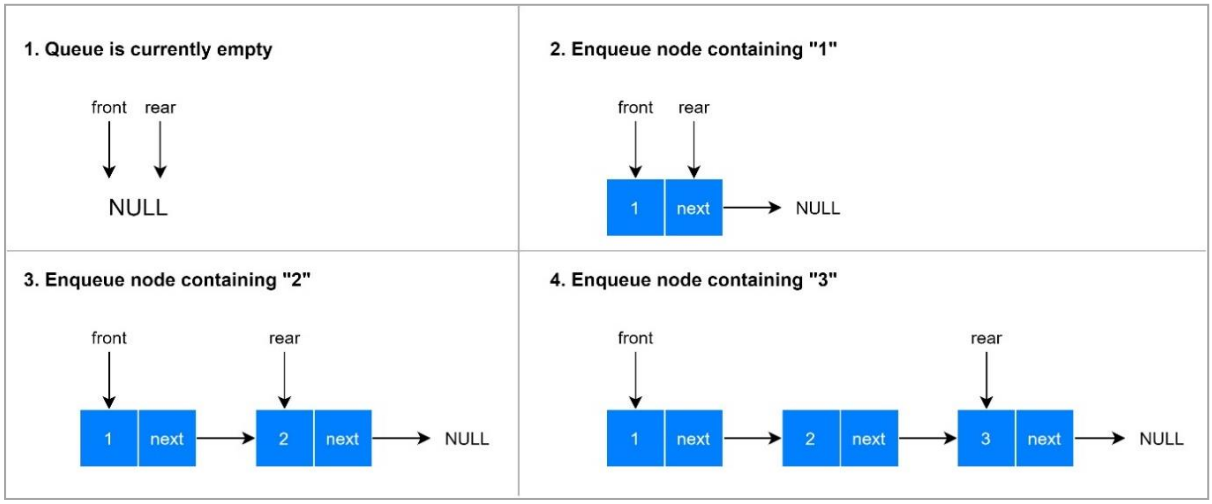

***Figure 3.2.1*:** Enqueue operation

ii. Dequeue

Dequeue removes an item from the queue by deletion, this is always done at the front of the queue. Before deleting a node, a temporary pointer is used to point to the first node and “front” points to the next node in the queue. Then, the memory for the first node is deallocated. If the queue is empty after that, then both “front” and “rear” will be pointing to NULL. If dequeue is used when the queue is empty, it will cause an underflow error.

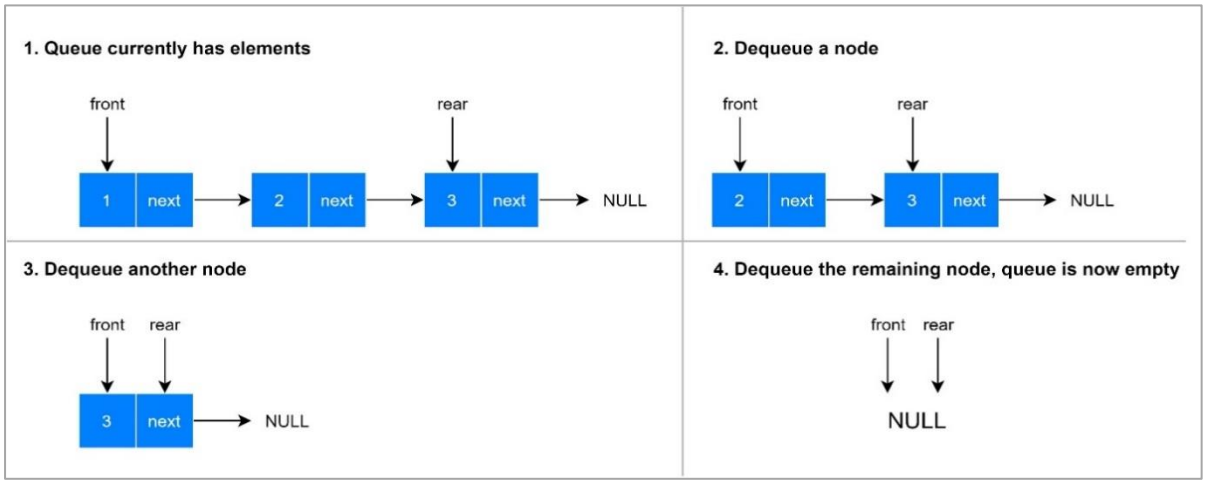

***Figure 3.2.2*:** Dequeue operation

iii. Peek

Peek is the process of checking and retrieving the data currently stored at the front of a queue. The data stored at the front node is simply copied into a variable and set as the return value of this operation.

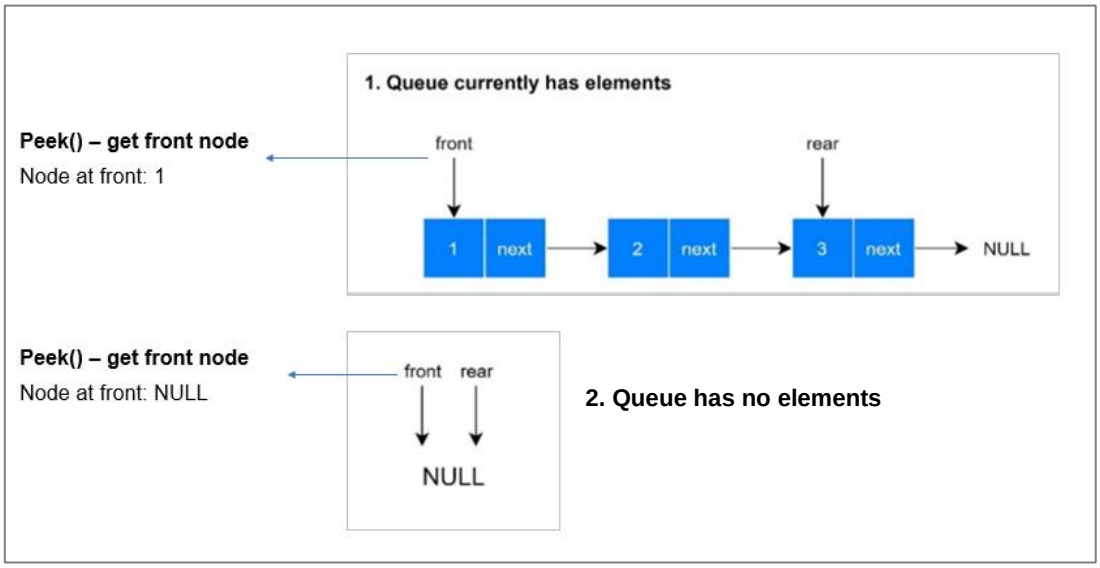

***Figure 3.2.3*:** Peek operation

iv. isEmpty

This operation is used to check if the queue is empty. If both “front” and “rear” are pointing to NULL, the queue is empty. If it is empty, this operation returns the boolean value true; otherwise, it returns false.

v. Display

This operation is used to display the data of all nodes in the queue if it is not empty. Traversal is used to step through every node and print out their data.

3.3 IMPLEMENTATION OF QUEUE IN C++

- **Program Code**

i. Node structure definition

```
// Node structure definition
struct node {
   int data;
   struct node *next;
};
```

***Figure 3.3.1*:** Structure definition for a node in queues

ii. Queue initialization

```
// Queue initialization
struct node* front = NULL;
struct node* back = NULL;
struct node* temporary;
```

***Figure 3.3.2*:** Declaration of front and back pointers that points at each end of the queue, and a temporary pointer to get copied values from the queue

iii. Enqueue

```
// Enqueue operation (adds nodes)
void enqueue() {
   int val;
   cout << "\nInsert an element in queue : ";
   cin >> val;

   if (back == NULL) {
       back = (struct node *)malloc(sizeof(struct node));
       back->next = NULL;
       back->data = val;
       front = back;
   } else {
       temporary = (struct node *)malloc(sizeof(struct node));
       back->next = temporary;
       temporary->data = val;
       temporary->next = NULL;
       back = temporary;
   }

   cout << endl << val << " has been enqueued.\n" << endl;
}
```

***Figure 3.3.3*:** Code for enqueue operation

iv. Dequeue

```
// Dequeue operation (deletes nodes)
void dequeue() {
   temporary = front;

   if (front == NULL) {
      cout << "\nThere is an underflow in the queue\n" << endl;
      return;
   } else if (temporary->next != NULL) {
      temporary = temporary->next;
      cout << "\nElement removed from the queue : " << front->data << endl << endl;
      free(front); // deallocate memory space
      front = temporary;
   } else {
      cout << "\nElement removed from queue : " << front->data << endl << endl;
      free(front);
      front = NULL;
      back = NULL;
   }
}
```

***Figure 3.3.4*:** Code for dequeue operation

v. Display

```
// Display operation (prints all data in the queue)
void display() {
   temporary = front;

   if ((front == NULL) && (back == NULL)) {
      cout << "The queue is empty\n" << endl;
      return;
   }

   cout << "Queue elements are: ";
   while (temporary != NULL) {
      cout << temporary->data << " ";
      temporary = temporary->next;
   }
   cout << endl << endl;
}
```

***Figure 3.3.5*:** Code for display operation

vi. isEmpty

```
// isEmpty operation (checks if queue is empty)
bool isEmpty() {
    // If front points to null, queue is empty, return true; else false
    if (front == NULL)
        return true;
    else
        return false;
}
```

***Figure 3.3.6*:** Code for isEmpty operation

vii. Peek

```
// Peek operation (retrieves data at front node)
void peek() {
    int data;
    if (front != NULL) {
        data = front->data;
        cout << "Data at the front: " << data << endl << endl;
    } else
        cout << "The queue is empty!" << endl << endl;
}
```

***Figure 3.3.7*:** Code for peek operation

- **Output produced by program**

i. Main menu

```
WELCOME TO QUEUE DEMO PROGRAM

1) Insert an element in the queue
2) Remove an element from queue
3) Display all the elements in the queue
4) isEmpty
5) Peek
6) Exit
Enter a choice : _
```

***Figure 3.3.8*:** Main menu of program

ii. Enqueue operation

```
Enter a choice : 1

Insert an element in queue : 3

3 has been enqueued.

Enter a choice : 1

Insert an element in queue : 5

5 has been enqueued.

Enter a choice : 1

Insert an element in queue : 7

7 has been enqueued.

Enter a choice : _
```

***Figure 3.3.9*:** Program asks user for data and enqueue operation is executed

```
Enter a choice : 1

Insert an element in queue : 7

7 has been enqueued.

Enter a choice : 3
Queue elements are: 3 5 7
```

Figure 3.3.10: Enqueue operation results

iii. Display operation

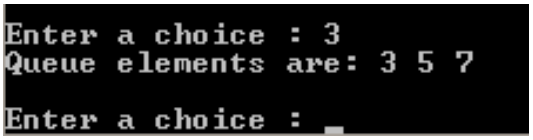

***Figure 3.3.11*:** Output for display operation if queue is not empty

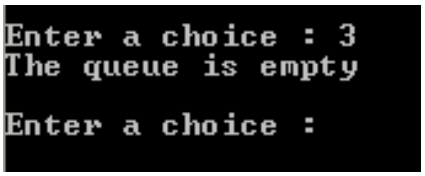

***Figure 3.3.12*:** Output for display operation if queue is empty

iv. isEmpty operation

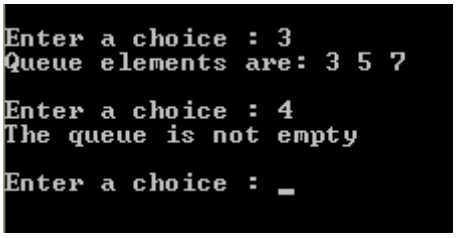

***Figure 3.3.13*:** Output for isEmpty operation if queue is not empty

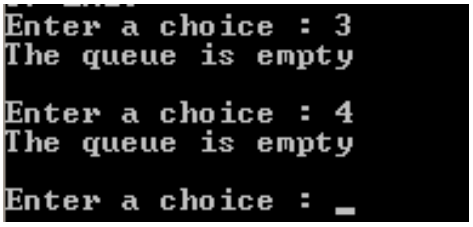

***Figure 3.3.14*:** Output for isEmpty operation if queue is empty

v. Peek operation

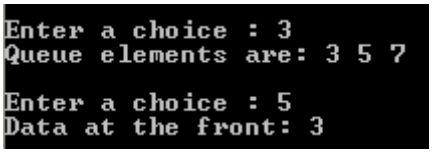

***Figure 3.3.15*:** Output for peek operation if queue is not empty

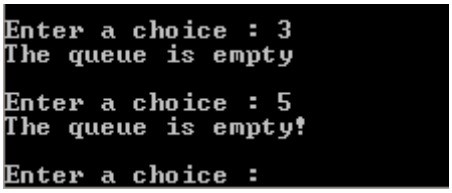

***Figure 3.3.16*:** Output for peek operation if queue is empty

vi. Dequeue operation

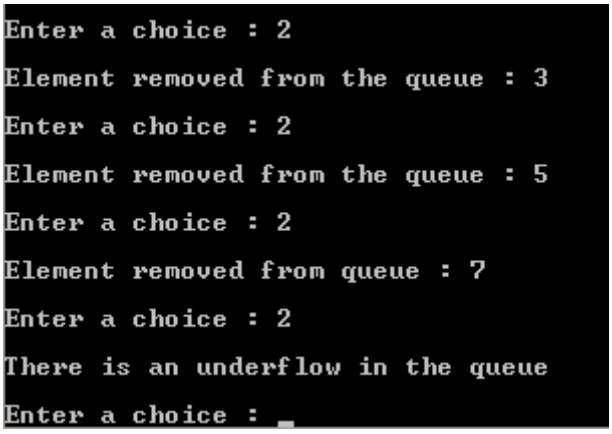

***Figure 3.3.17*:** Program asks user for target data and dequeue operation is executed

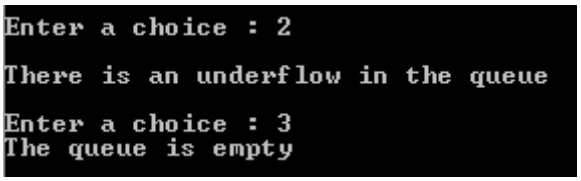

Figure 3.3.18: Dequeue operation terminates if queue is empty

4.0 BINARY TREES

4.1 DEFINITION

A binary tree is a non-linear, hierarchical data structure that has at most 2 children nodes for each parent node in the structure. Binary trees, in some cases, have left children that hold a lower value than its parent node's, while their right children hold a greater or equal value than its parent node's, or vice versa. They are usually used when people consider about efficient data searching and sorting.

There are different types of binary tree, such as full binary trees, complete binary trees, perfect binary trees, balanced binary trees and many more. Each binary tree type has their own unique properties that differentiate between themselves and a binary tree can be of multiple types at the same time:

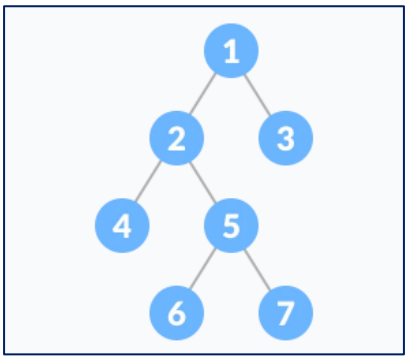

Figure 4.1.1: Full binary trees

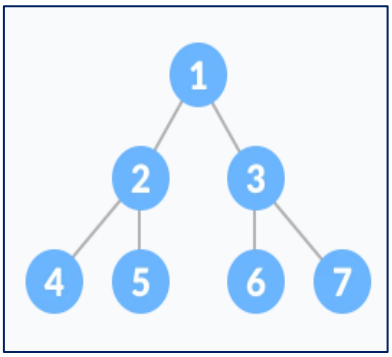

Figure 4.1.2: Perfect binary trees

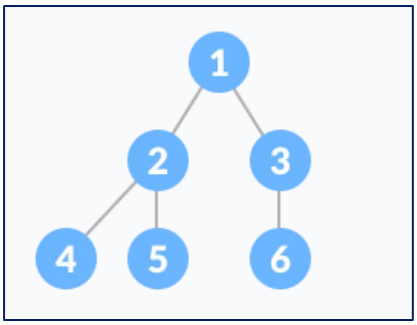

Figure 4.1.3: Complete binary trees

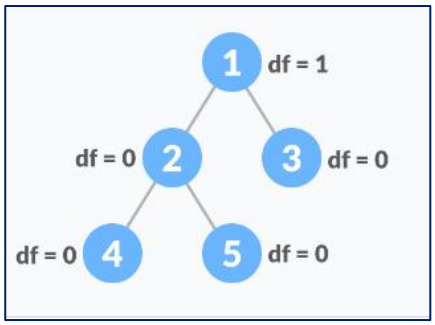

Figure 4.1.4: Balanced binary trees
(Source: Programiz, 2021)

Terminology:

Parent – Ancestor of a node
Child – Descendant of a node, either left or right
Root – Origin node of the tree
Leaf node – Node that has no children
Subtree – Granular tree structure using another node as root.

4.2 OPERATIONS OF BINARY TREES

In this implementation, the binary tree that will be used for demonstration is called a binary search tree. It is also a rarity **that allows duplicates**. There are some operations that can be performed on binary search trees, namely insertion, deletion, traversal, and search.

i. Insertion

This operation recursively checks the value of the new node before inserting it into the tree. Starting from the root, if the new node has a lower value than its parent node's, move the insert position to the left child, otherwise move it to the right child. This process repeats until a position with a NULL value is found. The new node will then be inserted at the determined position in the tree.

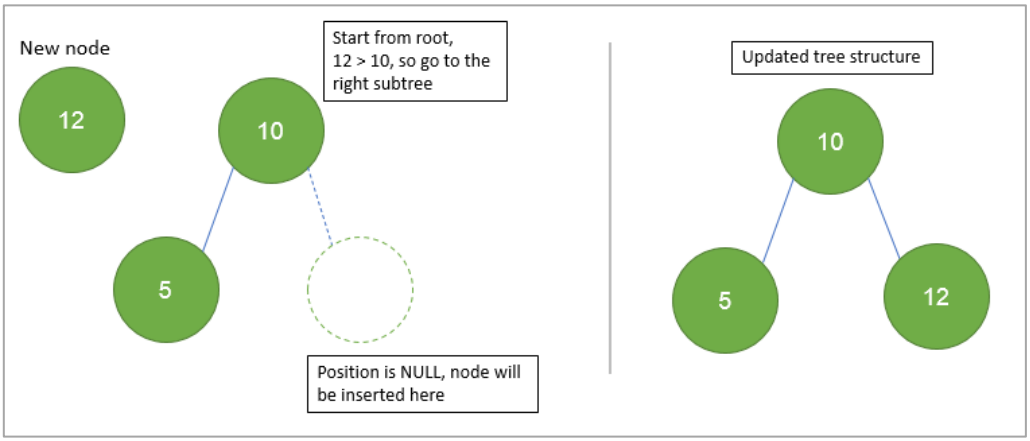

***Figure 4.2.1*:** Insertion of a new node

ii. Deletion

This operation searches the tree structure for the target node recursively and deleting it when found. Before deleting the node, if the target node has 2 children, its value will be replaced by its inorder successor's value, of which is the deepest, leftmost node from the subtree of the right child of the target node; If the target node only has 1 child, the child will replace its parent. Otherwise, if the target node is a leaf node, then the node can just be deleted without any modification to the tree. After deletion, the updated tree structure will be returned.

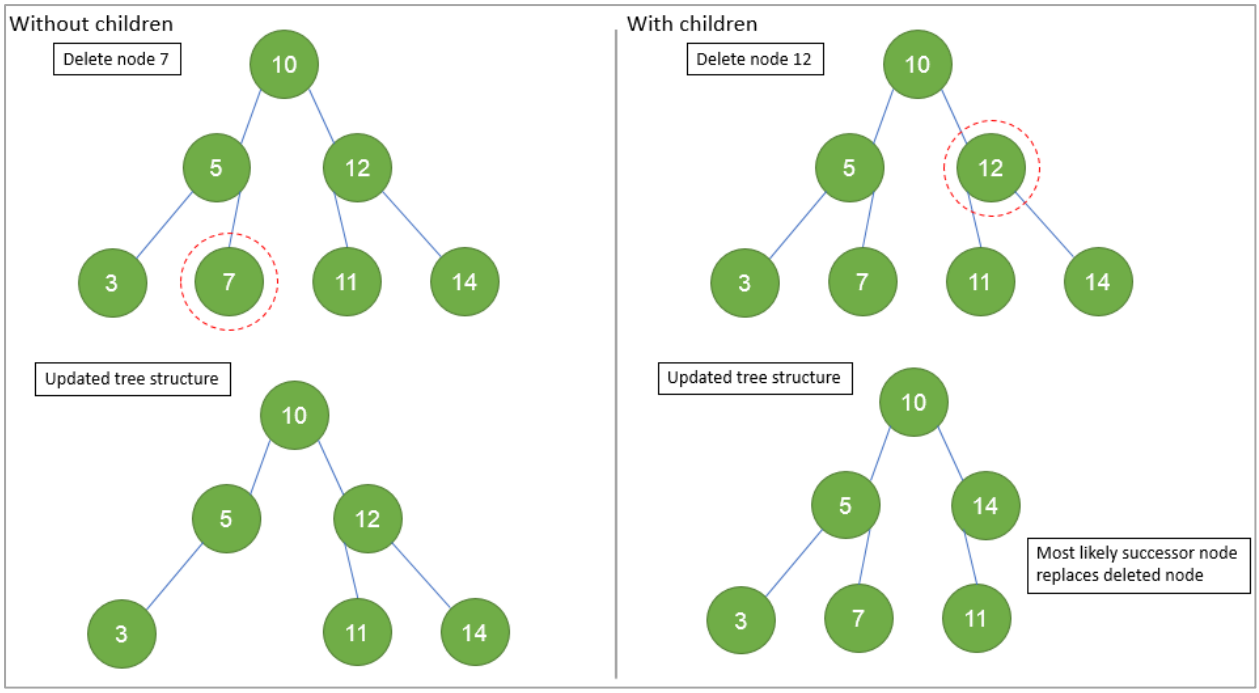

***Figure 4.2.2*:** Deletion of a node, with and without children

iii. Traversal

There are 3 ways to traverse through binary trees: inorder, preorder and postorder traversal. This operation displays the output of these traversal methods by visiting nodes in the following order:

1. **Inorder traversal**: Left subtree of root -> Root -> Right subtree of root
2. **Preorder traversal**: Root -> Left subtree of root -> Right subtree of root
3. **Postorder traversal**: Left subtree of root -> Right subtree of root -> Root

Inorder traversal is preferred since it produces an ordered node list that is more readable.

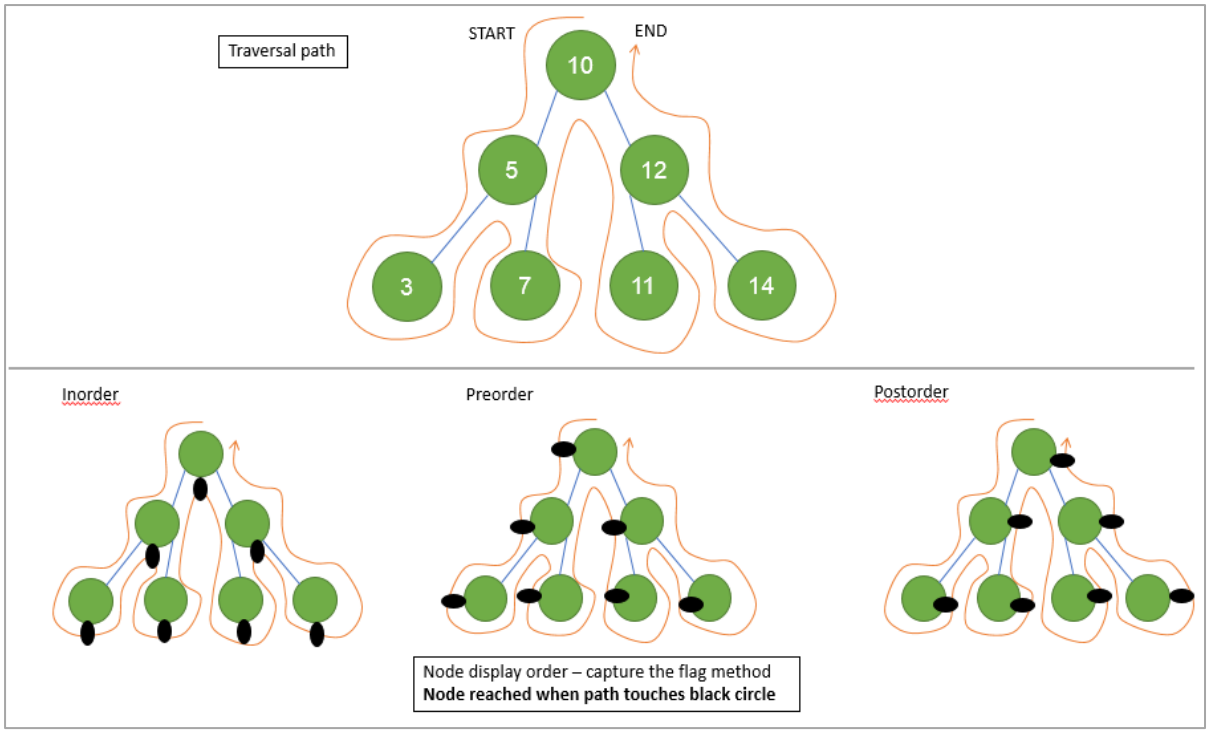

***Figure 4.2.3*:** Binary tree inorder, preorder and postorder traversal

iv. Search

This operation recursively travels down the tree by comparing nodes using the target node's value. The function returns the node if it is found and NULL if it is not. It is popularly known as binary search.

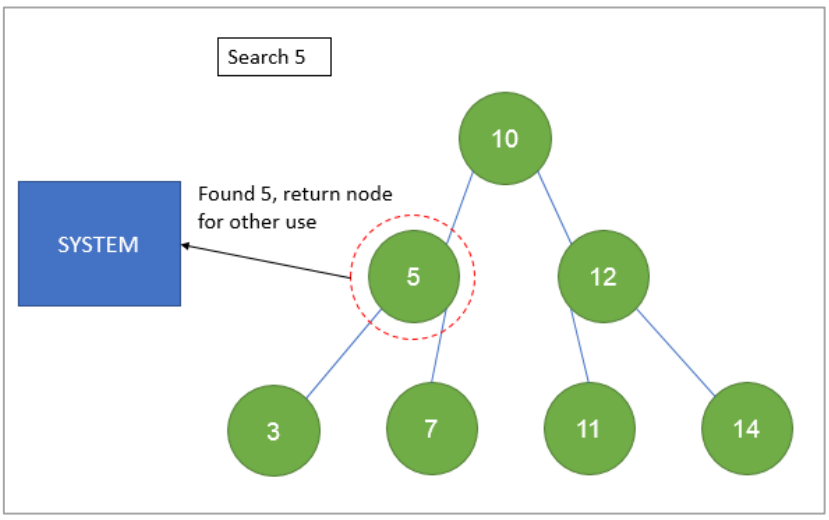

Figure 4.2.4: Searching for a node in a binary tree

4.3 IMPLEMENTATION OF BINARY TREE IN C++

- **Program Code**

i. TreeNode structure definition

```
struct TreeNode {
    int data;
    TreeNode *left = NULL;
    TreeNode *right = NULL;

    /*optional constructor
      If implemented, all initialized nodes must have a data
      If not implemented, data needs to be assigned by using member accessor, node->data */
    TreeNode(int d) {
        data = d;
    }
};
```

***Figure 4.3.1*:** Structure definition for a tree node in binary search trees

```
struct TreeNode *root;
```

***Figure 4.3.2*:** Declaration of root node

ii. Insertion function

```
int memFull = 0;
//recursive insert function
TreeNode* insertNode(TreeNode* current, int data) {
    TreeNode *newNode = new TreeNode(data);
    //if no memory is allocated for newNode, memory is full
    if (!newNode) {
        //sentinel value to check if memory is full
        memFull = 1;
        //return original value
        return current;
    } else if (current == NULL) {
    // recursion base case, return newNode when current = NULL
        return newNode;
    } else if (data < current->data)
    // assign to the left child if lower than parent data
        current->left = insertNode(current->left, data);
    else
    // assign to right child if greater/equal to parent data
        current->right = insertNode(current->right, data);

    //return updated subtree
    return current;
}
```

Figure 4.3.3: Code for insertion operation in binary search trees

iii. Deletion function

```
int dltFound = 0;
//recursive delete function
TreeNode* deleteNode(TreeNode* current, int data) {
    if (current == NULL)
    //if node is not found, return NULL
        return current;
    else if (data < current->data)
    //if target node lower than current node, go to left subtree
        current->left = deleteNode(current->left, data);
    else if (data > current->data)
    //if target node greater than/equal to current node, go to right subtree
        current->right = deleteNode(current->right, data);
```

Figure 4.3.4: Code for deletion operation in binary search trees – part (1)

```
    //Matching node found, processing delete
    else {
        //case 1: no children
        if (current->left == NULL && current->right == NULL) {
            delete current;
            current = NULL;
        //case 2: 1 child
        } else if (current->right == NULL) { //1 left child
            TreeNode* temp = current;
            current = current->left;
            delete temp;
        } else if (current->left == NULL) { //1 right child
            TreeNode* temp = current;
            current = current->right;
            delete temp;
        //case 3: 2 children
        } else {
            //find the inorder successor in the right subtree
            TreeNode* temp = findMinNode(current->right);
            //replace current data with the successor's
            current->data = temp->data;
            //delete the successor(duplicate now) from the right subtree
            //successor is either leaf node/node with right child so we can
            //just call the func recursively
            current->right = deleteNode(current->right,temp->data);
        }

        //sentinel value to detect node is found/not found
        dltFound = 1;
        //return updated subtree after deletion
        return current;
    }
}
```

Figure 4.3.5: Code for deletion operation in binary search trees – part (2)

```
TreeNode* findMinNode(TreeNode* rootNode) {
    //loops until deepest leftmost node is found
    while(rootNode->left != NULL) {
        rootNode = rootNode->left;
    }
    return rootNode;
}
```

Figure 4.3.6: Function to find inorder successor of target node

iv. Traversal function

```
//recursive traversal function
void inorderTraverse(TreeNode* rootNode) {
    if (rootNode == NULL) //base case
        return;
    /*Order of printing:
        1. root's left child's data
        2. root's data
        3. root's right child's data
    Functions will pause when they encounter child nodes that also have children
    and will proceed to work from the deepest, leftmost leaf nodes
    */
    inorderTraverse(rootNode->left);
    cout << rootNode->data << " -> ";
    inorderTraverse(rootNode->right);
}
```

***Figure 4.3.7*:** Code for binary search tree inorder traversal

```
void preorderTraverse(TreeNode* rootNode){
    if (rootNode == NULL)
        return;
    /*similar logic but different order:
        1. root's data
        2. root's left child's data
        3. root's right child's data
    */
    cout << rootNode->data << " -> ";
    preorderTraverse(rootNode->left);
    preorderTraverse(rootNode->right);
}
```

***Figure 4.3.8*:** Code for binary search tree preorder traversal

```
void postorderTraverse(TreeNode* rootNode){
    if (rootNode == NULL)
        return;
    /*similar logic but different order:
        1. root's left child's data
        2. root's right child's data
        3. root's data
    */
    preorderTraverse(rootNode->left);
    preorderTraverse(rootNode->right);
    cout << rootNode->data << " -> ";
}
```

***Figure 4.3.9*:** Code for binary search tree postorder traversal

v. Search function

```
//recursive search function
TreeNode* searchNode(TreeNode* rootNode, int data) {
    //if data not found
    if (rootNode == NULL)
        //return NULL
        return rootNode;
    else if (data < rootNode->data) {
    //if data smaller than parent, go to left subtree
        cout << "left->";
        return searchNode(rootNode->left, data);
    } else if (data > rootNode->data) {
    //if data smaller than parent, go to right subtree
        cout << "right->";
        return searchNode(rootNode->right, data);
    //if data is found
    } else {
        //return the node address
        return rootNode;
    }
}
```

***Figure 4.3.10*:** Code for search operation in binary search tree

- **Output Produced by Program**

i. Main Menu

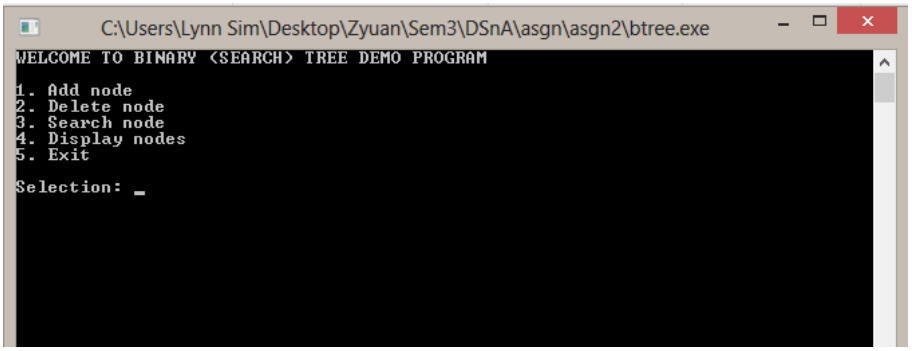

***Figure 4.3.11*:** Output of main menu when the program ran

ii. Insertion operation

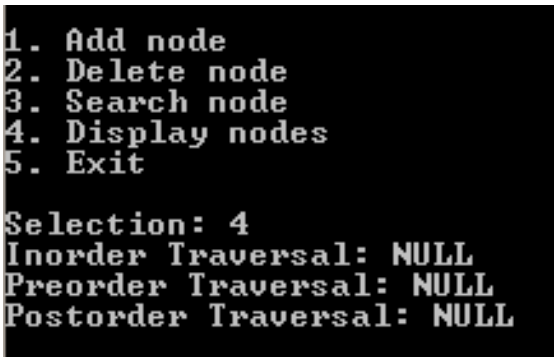

***Figure 4.3.12*:** Binary tree is not initialized yet

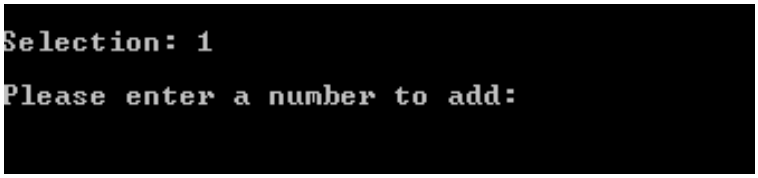

***Figure 4.3.13*:** System prompts user for the new node's value

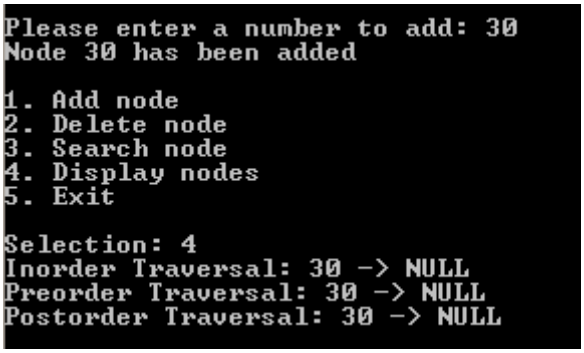

***Figure 4.3.14*:** Result after insertion operation, node is added to the binary tree

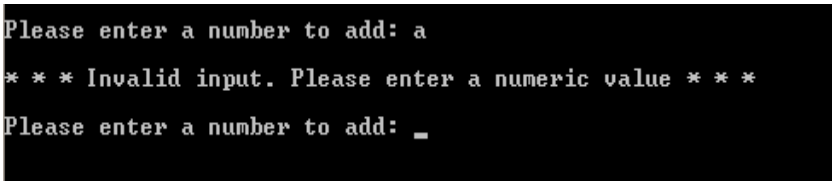

***Figure 4.3.15*:** Invalid input is captured, only numeric values allowed

iii. Deletion operation

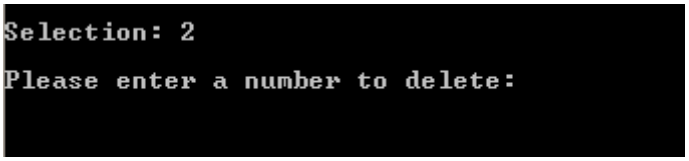

***Figure 4.3.16*:** System prompts user for the target node

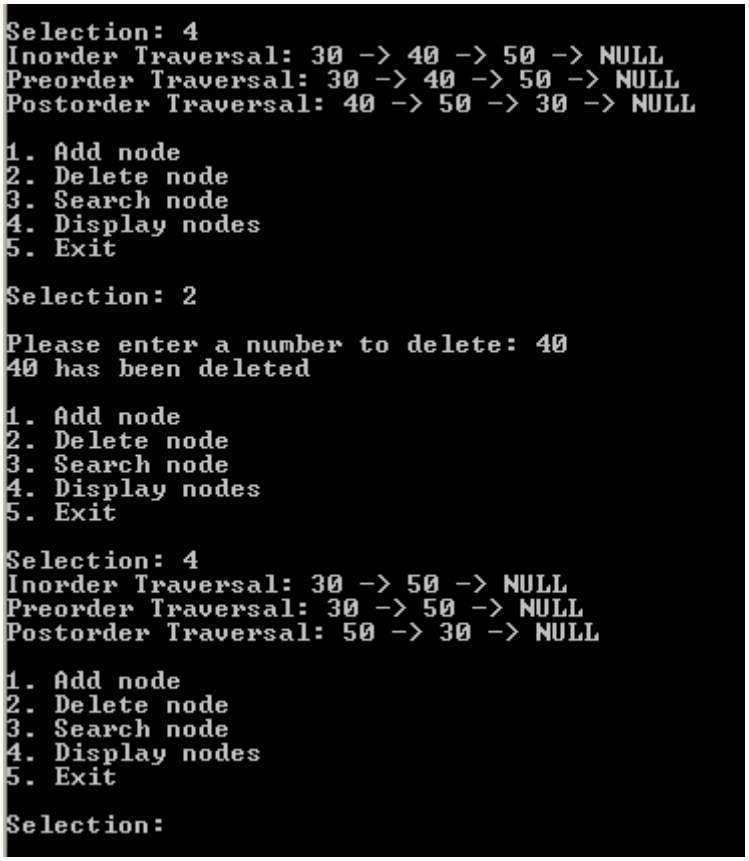

***Figure 4.3.17*:** Result after deleting an existing node, node is deleted from the binary tree

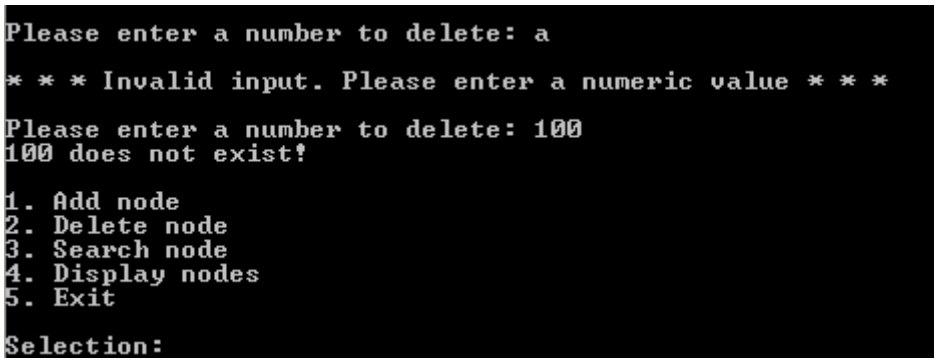

***Figure 4.3.18*:** Invalid input is captured, only numeric values allowed; System notifies user if target node does not exist in the binary tree

iv. Search operation

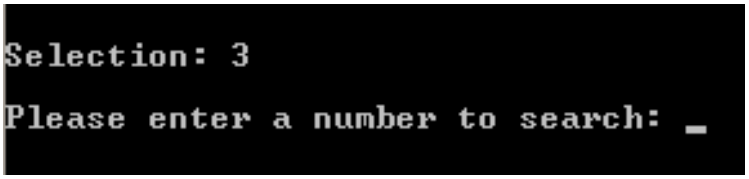

***Figure 4.3.19*:** System prompts user for the target node

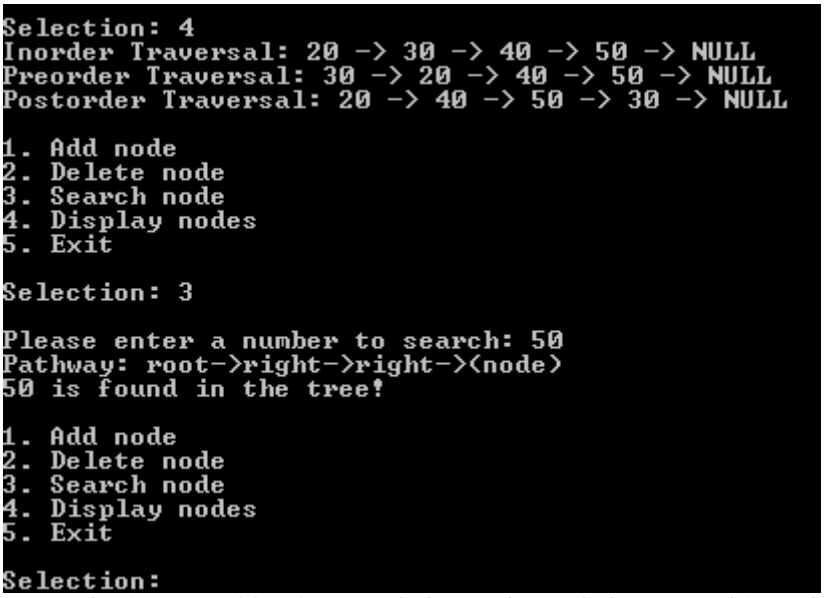

***Figure 4.3.20*:** Result after searching for an existing node, node is returned from the binary tree

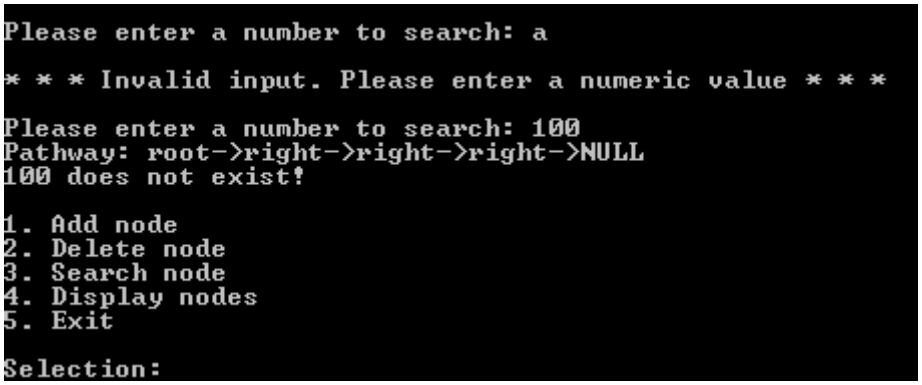

***Figure 4.3.21*:** Invalid input is captured, only numeric values allowed; System notifies user if target node does not exist in the binary tree

v. Traversal operation

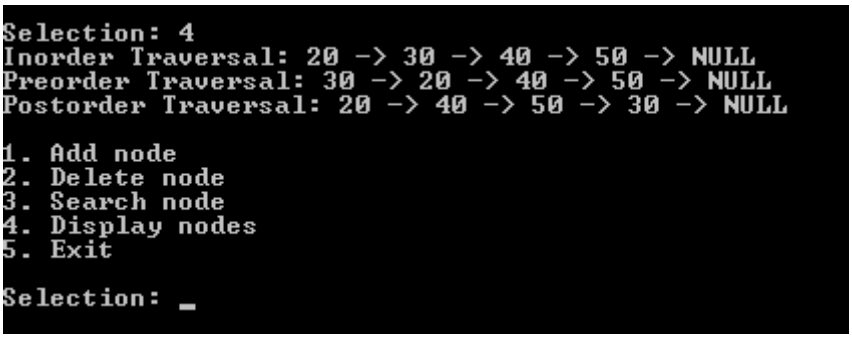

***Figure 4.3.22*:** Output of traversal operation, including inorder, preorder, and postorder traversal. System immediately returns user back to main menu after output

TOTAL WORD COUNT = 2028 *(excluding titles, figure descriptions and citations)*

PRESENTATION VIDEO LINKS

- **Linked List**

  Presenter: Thor Wen Zheng

  Video Link:
  https://drive.google.com/file/d/1PT5IMx1J6dXLpw49wB0sCF4ukpbfNgSH/view?usp=sharing

- **Stack**

  Presenter: Tan Peng Heng

  Video Link:
  https://drive.google.com/file/d/1gYT6AkjW1H2JPO_nE_xkfOQRwRIz8CyN/view?usp=sharing

- **Queue**

  Presenter: Adam John Simpson

  Video Link:
  https://drive.google.com/file/d/1O7QiFJiozOAYmaP8fNAbsoiLrG5Vpdbc/view?usp=sharing

- **Binary Tree**

  Presenter: Lim Zhe Yuan

  Video Link:
  https://drive.google.com/file/d/15YyC9im0DHQJgdJgH5bN2VBJs5o6LK87/view?usp=sharing

BIBLIOGRAPHY

AfterAcademy (2021) *Queue and its basic operations*. Available from https://afteracademy.com/blog/queue-and-its-basic-operations [accessed 21 July 2021]

Bullinaria, J. (2019) *Lecture Notes for Data Structures and Algorithms*. School of Computer Science, University of Birmingham. Birmingham, UK. Available from https://www.cs.bham.ac.uk/~jxb/DSA/dsa.pdf [accessed 23 July 2021].

Cplusplus (2021) *Introduction and concept of Stack*. Available from https://www.cplusplus.com/reference/stack/stack/. [accessed 20 July 2021].

FreeCodeCamp (2021) *Everything you need to know about tree data structures*. Available from https://www.freecodecamp.org/news/all-you-need-to-know-about-tree-data-structures-bceacb85490c/ [accessed 19 July 2021].

GeeksforGeeks (2021a) *Binary Tree Data Structure*. Available from https://www.geeksforgeeks.org/binary-tree-data-structure/ [accessed 19 July 2021].

GeeksforGeeks (2021b) *Data Structure*. Available from https://www.geeksforgeeks.org/data-structures/ [accessed 23 July 2021].

GeeksforGeeks (2021c) *Difference between Linear and Non-linear Data Structures*. Available from https://www.geeksforgeeks.org/difference-between-linear-and-non-linear-data-structures/ [accessed 23 July 2021].

GeeksforGeeks (2021d) *Implementation and Stack Introduction*. Available from https://www.geeksforgeeks.org/stack-in-cpp-stl/. [accessed 20 July 2021].

Joshi, V. (2017) *What's a Linked List, Anyway? [Part 1]*. Medium. Available from https://medium.com/basecs/whats-a-linked-list-anyway-part-1-d8b7e6508b9d [accessed 11 July 2021].

Malik, D.S. (2010) *Data Structures Using C++, 2nd* edition. Cengage Learning. Available from https://bu.edu.eg/portal/uploads/Computers%20and%20Informatics/Computer%20Science/1266/crs-10600/Files/Esam%20Halim%20Houssein%20Abd%20El-Halim_4-%20Data-Structure%20Using%20C++%20Malik.pdf [accessed 11 July 2021].

Mycodeschool (2021a) *Delete a node from Binary Search Tree* [video]. Available from https://www.youtube.com/watch?v=gcULXE7ViZw&list=PL2_aWCzGMAwI3W_JlcBbtYTwiQSsOTa6P&index=38 [accessed 19 July 2021].

Mycodeschool (2021b) *Inorder Successor in a binary search tree* [video]. Available from https://www.youtube.com/watch?v=5cPbNCrdotA&list=PL2_aWCzGMAwI3W_JlcBbtYTwiQSsOTa6P&index=38 [accessed 19 July 2021].

Opengenus (2021) *Explanation and concept of stack initialization.* Available from https://iq.opengenus.org/stack-initialization-cpp-stl/ [accessed 20 July 2021].

Programiz (2021a) *Binary Tree*. Parewa Labs Pvt. Ltd. Available from https://www.programiz.com/dsa/binary-tree [accessed 19 July 2021].

Programiz (2021b) *Data Structure and Types*. Parewa Labs Pvt. Ltd. Available from https://www.programiz.com/dsa/data-structure-types [accessed 23 July 2021].

Programiz (2021c) *Linked list Data Structure*. Parewa Labs Pvt. Ltd. Available from https://www.programiz.com/dsa/linked-list [accessed 11 July 2021].

Programiz (2021d) *Queue Data Structure*. Parewa Labs Pvt. Ltd. Available from https://www.programiz.com/dsa/queue [accessed 21 July 2021]

StackOverflow (2021a) *Base case in a recursive method*. Available from https://stackoverflow.com/questions/2783306/base-case-in-a-recursive-method [accessed 20 July 2021].

StackOverflow (2021b) *How to save the return value of a recursive function in a variable JavaScript*. Available from https://stackoverflow.com/questions/33513358/how-to-save-the-return-value-of-a-recursive-function-in-a-variable-javascript/33513732 [accessed 20 July 2021].

StackOverflow (2021c) *Passing struct pointer to function in c*. Available from https://stackoverflow.com/questions/10066709/passing-struct-pointer-to-function-in-c [accessed 20 July 2021].

StackOverflow (2021d) *Why Binary Tree is called by that name (Binary)?* Available from https://stackoverflow.com/questions/47528580/why-binary-tree-is-called-by-that-name-binary [accessed 19 July 2021].

Studytonight (2021a) *Introduction to Data Structures and Algorithms*. Available from https://www.studytonight.com/data-structures/introduction-to-data-structures [accessed 23 July 2021].

Studytonight (2021b) *What is a Queue Data Structure?* Available from https://www.studytonight.com/data-structures/queue-data-structure [accessed 21 July 2021].

Tutorialspoint (2021a) *Data Structure and Algorithms – Queue*. Available from https://www.tutorialspoint.com/data_structures_algorithms/dsa_queue.htm [accessed 21 July 2021].

Tutorialspoint (2021b) *Data Structure - Binary Search Tree*. Available from https://www.tutorialspoint.com/data_structures_algorithms/binary_search_tree.htm [accessed 19 July 2021].

DDA1224 Data Structures and Algorithms
MARKING RUBRIC
ASSIGNMENT 2
(20%)

Student1(S1): 0204677 LIM ZHE YUAN
Student2(S2): 0205096 THOR WEN ZHENG
Student3(S3): 0204144 ADAM JOHN SIMPSON
Student4(S4): 0205430 TAN PENG HENG

Coding(40%)

Learning Outcome	MARKING CRITERIA	SCALE					
		Not Achieved (0-4.5)	**Just Achieved (5.0 - 5.5)**	**Proficient (6.0 - 6.5)**	**Very Good (7.0 - 7.5)**	**Excellent (8.0-10.0)**	**YOUR MARKS/COMMENTS**
CLO2: Apply the use of basic data structures in applications.	1. Code Quality (5%)	Very poor attempt. Unstructured, no comment/remarks, non meaningful variables used.	A poor attempt; which may be several problems with structure, or very little use has been made of comments, or the naming of classes, methods and variables is unsatisfactory in a significant number of cases.	quality with several omissions of naming and use of comments.	of comments, and where the majority of classes, variables and methods have been appropriately named. However there may be several omissions of Javadoc comments, and the code.	Good use of commenting throughout, including Javadoc comments for the vast majority of classes, methods and variables.	
	2. Modularity (use of classes/functions, with parameters and returned value/ collection) (10%)	Non modular program. No function used.	Demonstrate some limited use of classes/ functions/ collection.	Demonstrates appropriate use of classes/ functions/ collection.	Demonstrates proficiency in use of classes/ functions/ collection.	Demonstrates mastery in the use of classes/ functions/ collection.	
	3.Use of appropriate data structure (10%)	No data structure used	Demonstrates limited use of data structure and algorithm.	Demonstrates reasonable use of data structure and algorithm, but with a few shortcomings.	Demonstrates proficiency in use of data structure and algorithm.	Demonstrates complete and proper use of data structure and algorithm.	

	4. Functionality (Program execution and output quality) (10%)	Non executable program	Program executed with runtime error but achieve partial program requirements.	Program executed error free with limitations to achieve minimum program requirements.	Program executed error free with correct output and achieve all program requirements.	Program executed error free with excellent output with appropriate validation.	
	5. Readability of the program (5%)	Non readable program	readability with poor indentation, variable names, and remarks	readability with poor indentation, but meaningful variable names, and remarks	readability with good indentation, meaningful variable names, but less remarks	Excellent readability with good indentation, meaningful variable names, and remarks	
						Total: 40%	
Report (60 %)							
Learning Outcome	**MARKING CRITERIA**	**SCALE**					
		Not Achieved (0-4.5)	**Just Achieved (5.0 - 5.5)**	**Proficient (6.0 - 6.5)**	**Very Good (7.0 - 7.5)**	**Excellent (8.0-10.0)**	**YOUR MARKS/COMMENTS**
CLO2: Apply the use of basic data structures in applications.	**1. Introduction (5%)**	Little or insufficient definition of a data structure	Basic definition of a data structure.One or two topics is/are left out.	Able to define a data structure and its services but some parts are without adequate explanation/support evidence.	Able to define mobile a data structure with adequate explanation /support but there are only minor parts need further clarification.	Author is able to define a data structure with adequate support of journal papers.	
	2. Quality of information (25%)	Information not clearly relates to the main Topic.	Information clearly relates to the main topic. Points are insufficiently developed. Analysis is weak	Information clearly relates to the main topic. Points are made, but analysis is minimal	Information clearly relates to the main topic. Points and analysis are made and related to the topic.	Information clearly relates to the main topic. Points are clearly made. Analysis is sophisticated	
	5.Style of writing (10%)	The report writing does not meet the criteria for the assignment (too short or incomplete, too long, and/or completely off-	Many ideas require clarification and/or are off-topic or have marginal relevance to the assignment. Many grammatical	Ideas are stated clearly and are related to the topic, with only adequate grammatical and/or spelling errors. Reference	Most ideas are stated clearly and are related to the topic, with only minor grammatical and/or spelling errors. Reference section is in minimal	Writing is clear and relevant, with no grammatical and/or spelling errors – polished and professional. Reference section properly formatted.	

		topic). Reference section is missing.	and/or spellings errors throughout the paper. Improper reference section	section with minor flaws						
	6. Individual presentation(content) (10%)	Student unable to demonstrates accurate, extensive, and deep understanding of the topics, where connection to a wide range of context, and reflection are not evident.	Student strive to demonstrates accurate, extensive, and deep understanding of the topics, where connection to a wide range of context, and reflection are partly evident.	Student strive to demonstrates accurate, extensive, and deep understanding of the topics, where connection to a wide range of context, and reflection are evident.	Student demonstrates accurate, extensive, and deep understanding of the topics, where connection to a wide range of context, and reflection are evident.	Student demonstrates very accurate, extensive, and deep understanding of the topics, where connection to a wide range of context, and reflection are clearly evident.	**S1**	**S2**	**S3**	**S4**
	7. Individual presentation (clarity) (10%)	Monotone; speaker seemed uninterested in material.	Little eye contact; fast speaking rate, little expression, mumbling.	Clear articulation of ideas, but apparently lacks confidence with materials.	Clear articulation of ideas, and confidence with materials.	Exceptional confidence with material displayed through poise, clear articulation, eye contact, and enthusiasm.				
						Total 60%				
						Overall score (100%)				