CC-213L

Data Structures and Algorithms

Laboratory 06

Singly LinkedList

Version: 1.0.0

Release Date: 14-10-2023

Department of Information Technology

University of the Punjab

Lahore, Pakistan

Contents:

Learning Objectives:

- Pointers and Dynamic Memory Allocation
- Self-Referential Objects
- Singly LinkedList

Resources Required:

- Desktop Computer or Laptop
- Microsoft ® Visual Studio 2022

General Instructions:

- In this Lab, you are **NOT** allowed to discuss your solution with your colleagues, even not allowed to ask how is s/he doing, this may result in negative marking. You can **ONLY** discuss with your Teaching Assistants (TAs) or Lab Instructor.
- Your TAs will be available in the Lab for your help. Alternatively, you can send your queries via email to one of the followings.

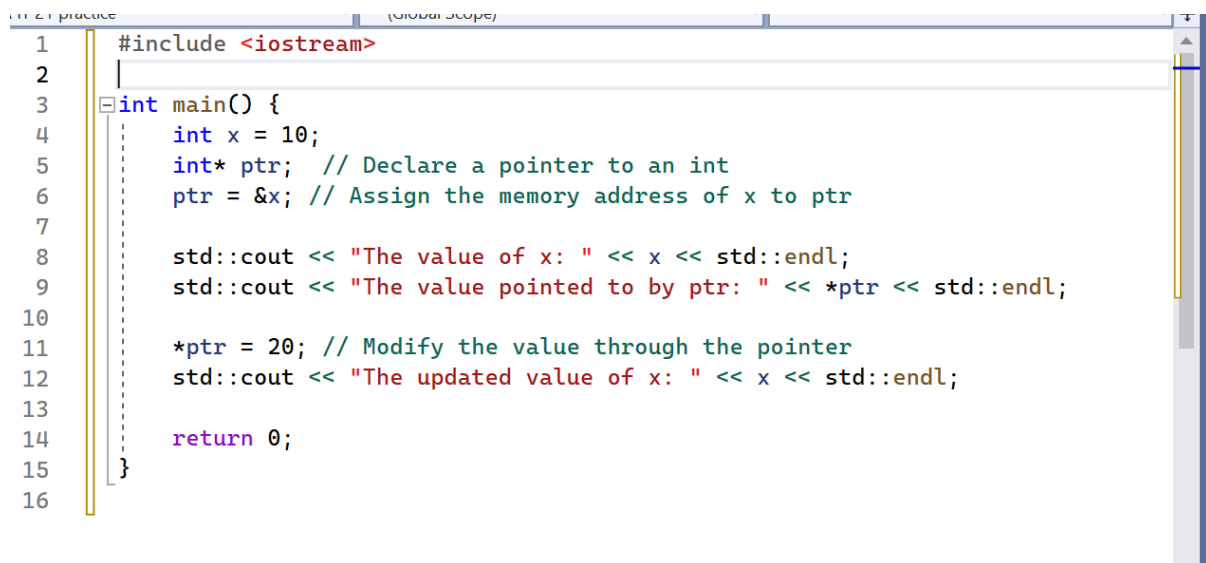| Teachers:		
Course Instructor	Prof. Dr. Syed Waqar ul Qounain	swjaffry@pucit.edu.pk
Lab Instructor	Madiha Khalid	madiha.khalid @pucit.edu.pk
Teacher Assistants	Muhammad Nabeel	bitf20m009@pucit.edu.pk
	Muhammad Subhan	bcsf20a033@pucit.edu.pk

Background and Overview

Pointers and Dynamic Memory Allocation

Pointers and dynamic memory allocation are important concepts in programming, particularly in languages like C and C++. Pointers allow you to work with memory addresses, while dynamic memory allocation allows you to manage memory at runtime.

Pointers:

A pointer is a variable that stores the memory address of another variable. It allows you to indirectly access the value of the variable stored at that address. Pointers are often used for various purposes, such as dynamically allocated memory, working with arrays, and passing functions as arguments.
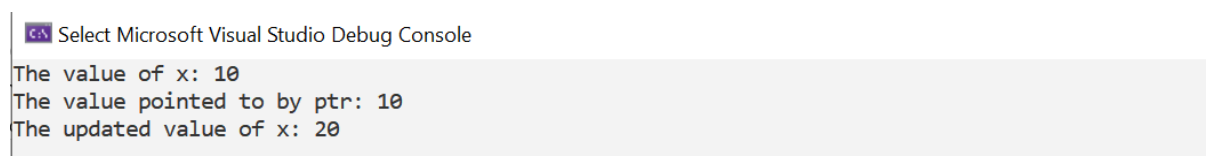
```cpp
#include <iostream>

int main() {
    int x = 10;
    int* ptr;  // Declare a pointer to an int
    ptr = &x; // Assign the memory address of x to ptr

    std::cout << "The value of x: " << x << std::endl;
    std::cout << "The value pointed to by ptr: " << *ptr << std::endl;

    *ptr = 20; // Modify the value through the pointer
    std::cout << "The updated value of x: " << x << std::endl;

    return 0;
}
```

Figure 1(Pointers)

Explanation:

In this example, ptr is a pointer to an integer, and it is assigned the memory address of the variable x. You can access and modify the value of x through the pointer using the dereference operator (*ptr).

```
Select Microsoft Visual Studio Debug Console
The value of x: 10
The value pointed to by ptr: 10
The updated value of x: 20
```

Figure 2(output)

Dynamic Memory Allocation

Dynamic memory allocation allows you to allocate memory for variables at runtime. In C++, you can use new and delete operators to allocate and deallocate memory for objects on the heap.

```
1    #include <iostream>
2
3    int main() {
4        int* dynamicArray = new int[5]; // Allocate an array of 5 integers
5
6        for (int i = 0; i < 5; i++) {
7            dynamicArray[i] = i * 10;
8        }
9
10       for (int i = 0; i < 5; i++) {
11           std::cout << "dynamicArray[" << i << "] = " << dynamicArray[i] << std::endl;
12       }
13
14       delete[] dynamicArray; // Deallocate the memory
15
16       return 0;
17   }
```

Figure 3(Dynamic Memory)

Explanation:

In this example, dynamicArray is allocated on the heap with space for 5 integers. After using it, it is essential to deallocate the memory using delete[] to prevent memory leaks.

Note: In modern C++ (C++11 and later), it is recommended to use smart pointers like std::unique_ptr and std::shared_ptr for better memory management, as they automatically handle memory deallocation.

```
Select Microsoft Visual Studio Debug Console
dynamicArray[0] = 0
dynamicArray[1] = 10
dynamicArray[2] = 20
dynamicArray[3] = 30
dynamicArray[4] = 40
```

Figure 4(Output)

Self-Referential Objects:

Classes that have capability to refer to their own types of objects are called **Self Referential Classes/Structs.** Objects of such classes are called self-referential Objects.

Self-referential structure in C++ are those structure that contains one or more than one pointer as their member which will be pointing to the structure of the same type. In simple words, a structure that is pointing to the structure of the same type is known as a self-referential structure.

Example in C++

```
1    struct Node
2    {
3        int info;
4        Node* ptr;
5    };
```
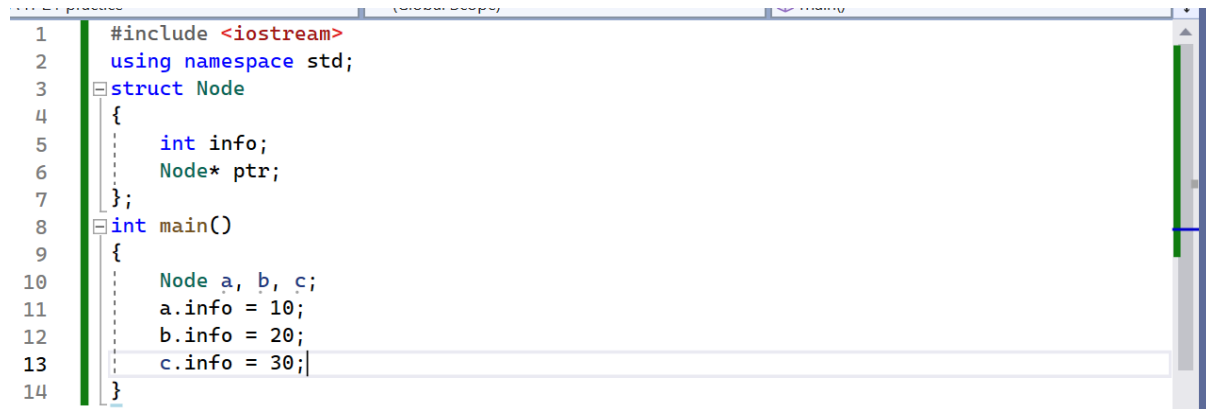
Figure 5(Self Referencing)

Explanation:

In Figure 5 we have declared a struct Node. It has two data members info and ptr.

Info Represents the information data part. Enables the object to store relevant information in it. There can be more than one identifier of same/different datatypes depending upon the application /situation.

ptr Represents the link part. Enables the object to a self-referential object. There can be more than one such references used for different purposes in different applications /situations.

```cpp
1    #include <iostream>
2    using namespace std;
3    struct Node
4    {
5        int info;
6        Node* ptr;
7    };
8    int main()
9    {
10       Node a, b, c;
11       a.info = 10;
12       b.info = 20;
13       c.info = 30;
14   }
```

Figure 6(Self Referential Objects)

Explanation:

We have declared three Node types variables a, b and stored proper values in their info data member.

```cpp
1    #include <iostream>
2    using namespace std;
3    struct Node
4    {
5        int info;
6        Node* ptr;
7    };
8    int main()
9    {
10       Node a, b, c;
11       a.info = 10;
12       b.info = 20;
13       c.info = 30;
14       a.ptr = &b;
15       b.ptr = &c;
16       c.ptr = nullptr;
17   }
```
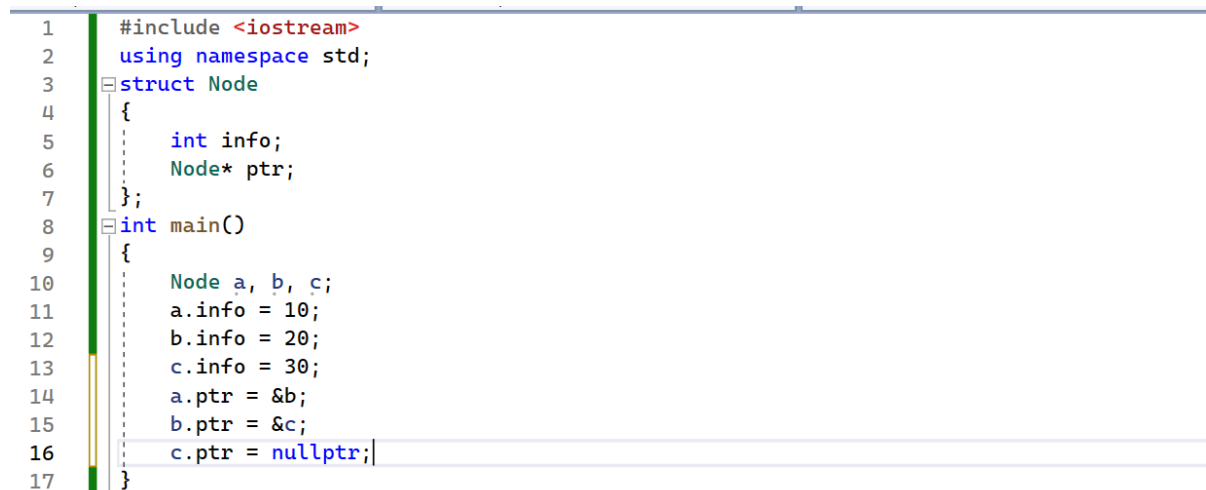
Figure 7(Self Referencing)

Explanation:

We have declared three Node types variables a, b and stored proper values in their info data member.

At line number 14 the ptr variable of type Node* (pointer to Node) is assigned the address of Node b and similarly at line 15 ptr variable of Node b is assigned addresss of Node c. ptr of Node c is pointing to null.

Some Important Operators:

- **_Member Access operators_**   arrow operator (→) and dot operator (**.**) have same precedence with associativity from left to right.
- **_Dereference /indirection operator (*)_** _address_ **_operator (&)_** have same precedence but associativity from right to left.
- Member access operators have high priority than indirection and address operators.

```cpp
8   int main()
9   {
10      Node a, b, c;
11      a.info = 10;
12      b.info = 20;
13      c.info = 30;
14      a.ptr = &b;
15      b.ptr = &c;
16      c.ptr = nullptr;
17      Node* p = &a;
18      cout << p->info << endl;
19      cout << p->ptr->info << endl;
20      cout << p->ptr->ptr->info << endl;
21
22  }
```

Figure 8(Member Access)

Explanation:

On line 17, a pointer of type Node named **p** is declared. This pointer will be used to reference **a** node. With this **p** pointer, it becomes straightforward to traverse through the linked list, as each node contains a **ptr** member that points to the next node in the sequence. We have accessed all the next node as well as information through the → (pointer member access operator).

```
Select Microsoft Visual Studio Debug Console
10
20
30
```

Figure 9(Output)

```cpp
7       };
8   int main()
9   {
10      Node a, b, c;
11      a.info = 10;
12      b.info = 20;
13      c.info = 30;
14      a.ptr = &b;
15      b.ptr = &c;
16      c.ptr = nullptr;
17      Node* p = &a;
18      cout << (*p).info << endl;
19      cout << (*(*p).ptr).info << endl;
20      cout << (*(*(*p).ptr).ptr).info << endl;
21
22  }
```

Figure 10(Indirection Operator)

Explanation:

At line number 18,19 and 20 members have been accessing through the indirection and dot operator that is object member access operator.

```
8   int main()
9   {
10      Node a, b, c;
11      a.ptr = &b;
12      b.ptr = &c;
13      c.ptr = nullptr;
14      Node* p = &a;
15      p->info = 100;
16      p->ptr->info = 200;
17      p->ptr->ptr->info = 300;
18      while (p != nullptr)
19      {
20          cout << p->info << endl;
21          p = p->ptr;
22      }
23
24      return 0;
25  }
```

Figure 11(Traverse Nodes)

Explanation:

Rather than individually accessing information from each node, a more efficient approach is to employ a loop for traversing the nodes. This way, you can access the information in each node as long as you haven't reached a node with its **ptr** member pointing to **nullptr.**

```
Select Microsoft Visual Studio Debug Console
100
200
300
```

Figure 12(Output)

Singly LinkedList

A linked list is a data structure used in computer science and programming to organize and store a collection of elements. It consists of a sequence of nodes, where each node contains two components:

Data: This component holds the actual value or information you want to store.

Pointer (or reference): This component points to the next node in the sequence, effectively linking the nodes together.
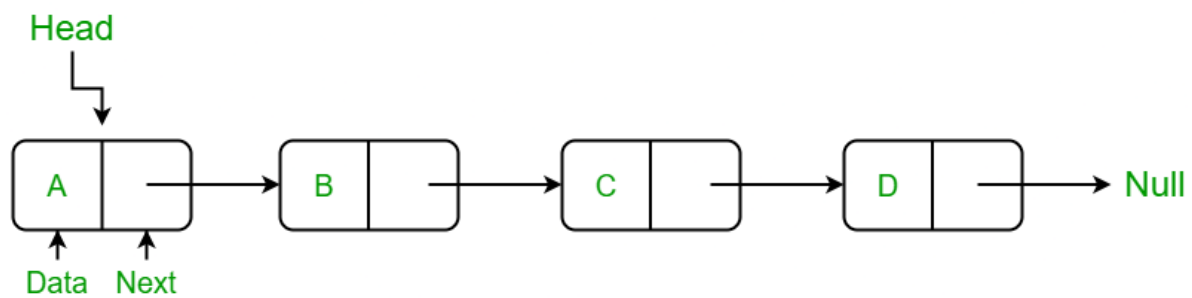
Figure 13(Singly LinkedList)

In a singly linked list, each node points to the next node in the list, forming a linear structure. The last node typically points to a special value like nullptr or NULL to indicate the end of the list.

Linked lists come in various forms, including singly linked lists (described above), doubly linked lists (where each node has a pointer to both the next and the previous node), and circular linked lists (where the last node points back to the first node, forming a closed loop).

Linked lists are particularly useful when you need to efficiently insert or delete elements at arbitrary positions in the list, as these operations typically involve updating a few pointers. They are often contrasted with arrays, which have fixed sizes and require elements to be shifted when inserting or deleting in the middle.

```
3    class Node
4    {
5    public:
6         int data;
7         Node* next;
8
9         Node(int val) : data(val), next(nullptr) {}
10   };
```

Figure 14(class Node)

Explanation:

A class Node have been declared that is self-referential class.

Insert Node:

```
12  int main() {
13      Node* head = new Node(1);
14      head->next = new Node(2);
15      head->next->next = new Node(3);
16
17      // Traversing the linked list and printing its elements
18      Node* current = head;
19      while (current != nullptr) {
20          std::cout << current->data << " -> ";
21          current = current->next;
22      }
23      std::cout << "nullptr" << std::endl;
24
25      return 0;
26  }
```

Figure 15(Linked list)

Explanation:

In this example, a singly linked list with three nodes is created, and then the elements are traversed and printed.

```
Select Microsoft Visual Studio Debug Console
1 -> 2 -> 3 -> nullptr
```

Figure 16(Output)

Delete Node:

```
12  void deleteNode(Node*& head, int value) {
13      // Handle the case when the node to be deleted is the head
14      while (head != nullptr && head->data == value) {
15          Node* temp = head;
16          head = head->next;
17          delete temp;
18      }
19
20      // Traverse the linked list and delete nodes with the specified value
21      Node* current = head;
22      while (current != nullptr) {
23          while (current->next != nullptr && current->next->data == value) {
24              Node* temp = current->next;
25              current->next = current->next->next;
26              delete temp;
27          }
28          current = current->next;
29      }
30  }
```

Figure 17(Delete Nodes)

Activities

Pre-Lab Activities:

Task 01: Singly LinkedList implementation

Create a linked list with following class definitions.

```cpp
template<class T>
class LinkedList;
template<class T>
class Node
{
public:
        T info;
        Node<T>* next;
        // Methods…
};
template<class T>
class LinkedList
{
        Node<T>* head;

        // Methods…
};
```

Implement following functions for List class.

1. Constructor, destructor, Copy-constructor.

2. void insertAtHead(T value)

3. void insertAtTail(T value)

4. bool deleteAtHead( )

5. bool deleteAtTail( )

6. void printList( )

7. Node* getNode(int n)

This function should return pointer to nth node in the list. Returns last node if n is greater than the number of nodes present in the list.

8. bool insertAfter(T value, T key)

Insert a node after some node whose info equals input parameter key and returns true if node is successfully inserted, false otherwise.

9. bool insertBefore(T value, T key)

Insert a node before some node whose info equals input parameter key and returns true if node is successfully inserted, false otherwise.

10. bool deleteBefore(T key)

Delete a node that is before some node whose info equals input parameter key and returns true if node is successfully inserted, false otherwise. Check boundary cases i.e if node to be deleted is last node or

first node in the list.

11. bool deleteAfter(T value)

Delete a node that is after some node whose info equals input parameter key and returns true if node is successfully inserted, false otherwise. Check boundary cases i.e if node to be deleted is last node or first node in the list.

12. int getLength( ) returns the total number of nodes in the list.

13. Node* search(T x)

Search a node with value "x" from list and return its link. If multiple nodes of same value exist, then return pointer to first node having the value "x".

In-Lab Activities

Task 01 Remove KthNode

Implement the following public member function of the **LinkedList** class:

bool removeKthNode (int k, int& val)

This function will remove the **k**th element (node) from the linked list.

For example, if the linked list object **list** contains these 8 values **{4 2 8 1 9 5 4 6}**, then the function call:

list.removeKthNode(4,val);

should remove the 4th element (node) from the linked list and the resulting **list** should contain these 7 values: **{ 4 2 8 9 5 4 6 }** . Before deallocating the node, this function should store the data present in that node into the reference parameter **val**.

This function should return **false** if the linked list contains fewer than **k** elements; otherwise it should remove the **k**th node from the linked list and return **true.** (*Note:* You are NOT allowed to modify the data of any node in the linked list).

Also write a driver main function to test the working of the above function.

Implement the following public member function of the **LinkedList** class:

Task 02: Combine Lists

void combine (LinkedList& list1, LinkedList& list2)
This function should combine the nodes of the two linked lists (**list1** and **list2**) into one list. All the nodes of the first list (**list1**) will precede (come before) all the nodes of the second list (**list2**).

For example, if **list1** contain **{7 3 4 2}, list2** contains **{5 9}**, and **list3** is Empty, then after the function call:

list3.combine(list1,list2);

list3 should contain **{7 3 4 2 5 9}** and **list1** and **list2** should be Empty now.

*Very Important: You are NOT allowed to create any new node in this function. You are also NOT allowed to modify the "data" field of any node. You can **assume** that the LinkedList object on which this function is called is empty at the start of this function.*

Hint 1: Do a lot of **paperwork** before writing the code. Also, make sure that all the **boundary cases** / **special cases** are properly handled.

Also write a driver main function to test the working of the above function.

Hint 2: Pseudo-code outline of **combine** function:

```
if (Both list1 and list2 are EMPTY)
{
```

```
        }
    else  if (list1  is  EMPTY)

    {

    }
    else  if (list2  is  EMPTY)
    {


    }
    else            // Both list1 and list2 contain at least one node

    {

    }
```

Task 03: Shuffle Merge

Implement the following public member function of the **LinkedList** class:

void shuffleMerge (LinkedList& list1, LinkedList& list2)

This function should shuffle-merge the nodes of the two linked lists (**list1** and **list2**) to make one list, by taking nodes alternately from the two lists.

For example, if **list1** contains **{2 6 4}**, **list2** contains **{8 1 3}**, and **list3** is Empty, then after the function call:

list3.shuffleMerge(list1, list2);

list3 should contain **{2 8 6 1 4 3}** and **list1** and **list2** should be Empty now.

*Very Important: You are NOT allowed to create any new node in this function. You are also NOT allowed to modify the "data" field of any node. You can assume that both lists (which are being shuffle-merged) contain the **same number of elements/nodes**. You can also assume that the LinkedList object on which this function is called is empty at the start of this function.*

Hint: Do a lot of **paperwork** before writing the code. Also, make sure that all the **boundary cases**

/ special cases are properly handled.

Also write a driver main function to test the working of the above function.

Task 04: Linked Stack

You have already implemented array-based stack that needs resizing after its array is full. But there is no need of resizing in the linked stack. Implement a linked Stack as provided with the template classes.

```cpp
// forward declaration of template class Stack
template <class T>
class Stack;
template<class T>
class StackNode
{

    T data;
    StackNode* link;
    friend Stack<T>
public:
    // methods
};
template<class T>
class Stack
{
private:
    StackNode* Top;
    int Size;

public:
    Stack()
    {
        Top = nullptr;
        Size = 0;
    }
    ~Stack();
    Stack(const Stack<T>&); // copy constructor
    int getTop();
    int pop();
    void push(int Element);
    int currSize();
    bool isEmpty();
    bool isFull();
    T Peek(int nodeNumber);
};
```

Post-Lab Activities

Task 01: Linked Queue

We studied about how to represent queues using sequential organization. Such a representation is efficient if we have a circular queue of fixed size. However, there are many drawbacks of implementing queues using arrays. The fixed sizes do not give flexibility to the user to dynamically exceed the maximum size. The declaration of arbitrarily maximum size leads to poor utilization of memory. In addition, the major drawback is the updating of front and rear. For correctness of the said implementation, the shifting of the queue to the left is necessary and to be done frequently. Here is a good solution to this problem which uses linked list.

Implement following linked Queue.

```cpp
template <class T>
class Queue;
template<class T>
class QNode
{
    friend Queue<T>;
    T data;
    QNode* link;
public:
    // methods
};
template<class T>
class Queue
{
    QNode* front, * fear;
    int size;
public:
    Queue()
    {
        front = fear = Null;
    }
    Queue(const Queue<T>&);
    void enqueu(int element);
    T dequeue();
    int currentSize();
    int frontElement();
    bool isEmpty();
    bool isFull();
    void display();
    ~Queue();
};
```

Submissions:

- For In-Lab Activity:
    - Save the files on your PC.
    - TA's will evaluate the tasks offline.
- For Pre-Lab & Post-Lab Activity:
    - Submit the .cpp file on Google Classroom and name it to your roll no.

Evaluations Metric:

- All the lab tasks will be evaluated offline by TA's
- **Division of Pre-Lab marks:**   **[40 marks]**
    - Task 01: Singly LinkedList Implementation   [40 marks]
- **Division of In-Lab marks:**   **[60 marks]**
    - Task 01: Remove Kth Node   [10 marks]
    - Task 02: Combine Lists   [10 marks]
    - Task 03: Shuffle Merge   [10marks]
    - Task 04: Linked Stack   [30marks]
- **Division of Post-Lab marks:**   **[40 marks]**
    - Task 01: Linked Queue   [40 marks]

References and Additional Material:

Singly LinkedList

https://www.geeksforgeeks.org/what-is-linked-list/

Lab Time Activity Simulation Log:

- Slot – 01 – 02:00 – 00:15:   Class Settlement
- Slot – 02 – 02:15 – 02:30:   In-Lab Task 01
- Slot – 03 – 02:30 – 02:45:   In-Lab Task 01
- Slot – 04 – 02:45 – 03:00:   In-Lab Task 02
- Slot – 05 – 03:00 – 03:15:   In-Lab Task 02
- Slot – 06 – 03:15 – 03:30:   In-Lab Task 03
- Slot – 07 – 03:30 – 03:45:   In-Lab Task 03
- Slot – 08 – 03:45 – 04:00:   In-Lab Task 04
- Slot – 09 – 04:00 – 04:15:   In-Lab Task 04
- Slot – 10 – 04:15 – 04:30:   In-Lab Task 04
- Slot – 11 – 4:300 – 04:45:   In-Lab Task 04
- Slot – 12 – 04:45 – 05:00:   Discussion on Post-Lab Task