Fraser International College

CMPT 135: Introduction to Computing Science and Programming II

Linked List Data Structure Supplementary Material

Instructor: Dr. Yonas T. Weldeselassie (Ph.D.)

In this supplementary material, we will describe the linked list data structure and discuss the typical operations that we can perform on linked lists.

Definition: A linked list data structure is a container whose size can expand (by inserting new objects into the container) or shrink (by removing objects from the container) that stores its objects in sequence and only allows sequential access of its elements which is achieved thanks to the fact that each object knows the object in the container that comes after it. Fig 1 below shows a linked list with four objects.

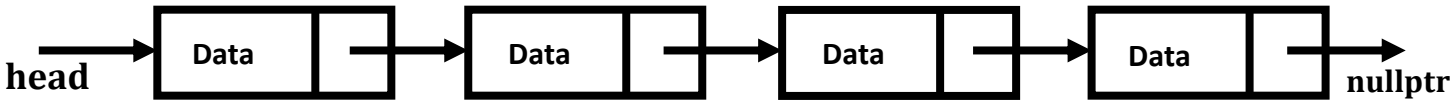


Fig 1. Linked List Data Structure

Terminologies

- Each object in a linked list is called a node. Thus the linked list in fig 1 above has four nodes.
- A node has two parts: a data part where we store data and a pointer part which points to the next node in the linked list.
- The last node in the linked list points to a special pointer value that indicates the end of the linked list. A convenient value that we can easily use will be the nullptr value.
- A linked list also requires a pointer that points to the first node in the linked list. This pointer is usually referred to as the head pointer of the linked list.
- A linked list with no nodes is known as an empty linked list. Thus a linked list is empty if its head pointer has a nullptr value.
- The first node in a linked list is known as the head node.
- The last node in a linked list is known as the tail node.

Representing a Node of a linked list

Since a node is an object, we may use either a C++ struct or a C++ class to represent it. A node will have two member variables; namely a data member variable (where we will store some data of our choice) and a pointer member variable of type node pointer (where we will store the memory address of the next node in the linked list). The data member variable can be any data type depending on what kind of information we are interested to store in our linked list.

The following class demonstrates a C++ class named Node that represents a node of a linked list whose data member variable is an integer data type and whose pointer member variable is a Node* data type.

Observe that it is a good programming habit to have typedef names whenever we work with pointers so that to minimize coding and debugging time. Thus since we will be working quite a lot with Node*, it is a good idea to have a typedef name for it. We see that we can specify a typedef name inside a class right at the beginning in order to use the typedef inside the class block but not outside the class block. We will then need to specify the typedef name again outside the class in order to use it in the remaining part of our program. The best

place to specify the typedef again is right after the class declaration but before the class definition as shown below. We also provide a test main program to demonstrate how to use the Node class in our applications.

```
#include <iostream>
using namespace std;
class Node
{
typedef Node* NodePtr;
private:
	int data;
	NodePtr link;
public:
	Node();//Construct ONE node whose data is 0 and link whose is nullptr
	Node(const int&);//Construct ONE node whose data is given and whose link is nullptr
	Node(const Node&);//Construct ONE node whose data is shallow copied and whose link is
nullptr
	int getData() const;
	NodePtr getLink() const;
	void setData(const int&);
	void setLink(const NodePtr&);
	friend ostream& operator << (ostream&, const Node&);
};

typedef Node* NodePtr;

Node::Node() : data(0), link(nullptr) {}
Node::Node(const int& d) : data(d), link(nullptr){}
Node::Node(const Node& n) : data(n.data), link(nullptr){}
int Node::getData() const { return data; }
NodePtr Node::getLink() const { return link; }
void Node::setData(const int& d) { data = d; }
void Node::setLink(const NodePtr& p) { link = p; }
ostream& operator << (ostream& out, const Node& n)
{
	out << n.data;
	return out;
}
int main()
{
	//Create three nodes and print them
	Node n1; //n1 data is 0 and n1 link is nullptr
	Node n2(5); //n2 data is 5 and n2 link is nullptr
	Node n3(n2); //n3 data is 5 and n3 link is nullptr

	cout << "Node 1 is " << n1 << endl;
	cout << "Node 2 is " << n2 << endl;
	cout << "Node 3 is " << n3 << endl;

	system("Pause");
	return 0;
}
```

Representing a linked list

In order to represent a linked list, all we need is the head pointer of the linked list. Then the head pointer will have information where to find the first node in the linked list. Why? Because it will be assigned the memory address of the first node in the linked list and therefore will be pointing to the first node in the linked list. Similarly the first node will know where to find the second node. Why? Because the link pointer of the first

node will be assigned the memory address of the second node and thus it will be pointing to it. Similarly, the second node also will know where to find the third node; the third node will know where to find the fourth node, and so on so forth. Thus given the Node class shown above, the following program demonstrates the creation of a linked list and traversal of the linked list in order to print the nodes in the order they are arranged in the linked list.

```
int main()
{
	//Create three nodes
	Node n1; //n1 data is 0 and n1 link is nullptr
	Node n2(5); //n2 data is 5 and n2 link is nullptr
	Node n3(n2); //n3 data is 5 and n3 link is nullptr

	//Create a linked list with the sequence head-->n1-->n2-->n3-->nullptr
	NodePtr head = &n1;  //head is pointing to n1 node
	n1.setLink(&n2); //n1 link is pointing to n2 node
	n2.setLink(&n3); //n2 link is pointing to n3 node

	//Traverse the linked list to print the nodes in the linked list
	cout << "Printing the linked list" << endl;
	while (head != nullptr)
	{
		cout << *head << "  ";
		head = head->getLink(); //Now head will point to the next node
	}
	cout << endl;

	system("Pause");
	return 0;
}
```

Observation: Observe that traversal of a linked list is possible only in the forward direction but not in the backward direction. Neither does a linked list data structure allow random access of its nodes. If we want to access a node, then the only way is to start from the head pointer and traverse in a forward direction until we find the node of our interest.

Caution: Observe that after traversing through the linked list during the printing of the nodes, at the end the value of the head pointer will become nullptr; which means it is no more pointing to the first node in the linked list and therefore no more the head of the linked list. This means in essence we have destroyed our linked list. Thus, if we want to perform more traversals of the linked list afterwards then we can't because we have lost the head of the linked list. This shortcoming will be fixed in our next example program.

Constructing linked lists: Empty linked list

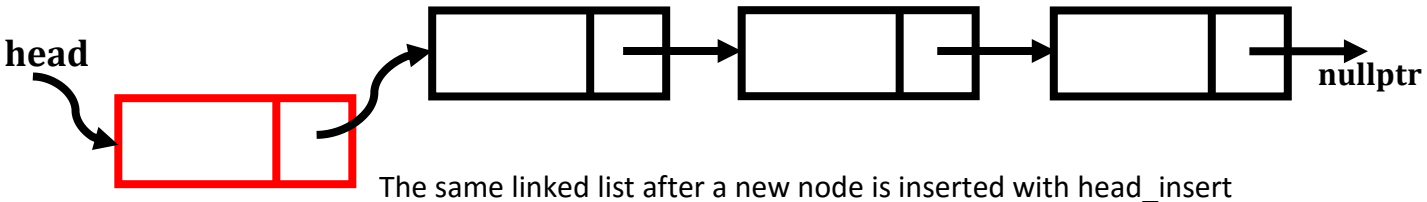
A linked list is constructed starting with an empty linked list; that is where there are no nodes in it. An empty linked list is constructed by assigning its head pointer a nullptr value. Then new nodes are inserted to the linked list either at the beginning of the linked list, in the middle of the linked list, or at the end of the linked list as we desire.

Moreover, typically nodes of a linked list are stored on the heap memory so that to be able to expand or shrink the linked list as we desire.

Inserting nodes in a linked list: head_insert algorithm

The easiest way to insert a new node into a linked list is to insert the new node as a first node in the linked list. Inserting a new node as a first node is usually referred to as head_insert operation. The diagram below shows a linked list before and after a new node (shown in red color) is inserted using head_insert operation.

Given linked list



The same linked list after a new node is inserted with head_insert

The following program demonstrates the creation of an empty linked list, inserting new nodes with head_insert algorithm, traversal of the linked list in order to print the nodes, and finally traversal of the linked list in order to delete the nodes of the linked list.

```
int main()
{
	//Create an empty linked list
	NodePtr head = nullptr;

	//Insert five nodes in the linked list using head_insert operation
	for (int i = 0; i < 5; i++)
	{
		NodePtr n = new Node(i);
		n->setLink(head);
		head = n;
	}

	//Print the linked list
	cout << "Printing the linked list." << endl;
	NodePtr temp = head;
	while(temp != nullptr)
	{
		cout << *temp << endl;
		temp = temp->getLink();
	}

	//Delete the nodes from the heap
	cout << "Deleting the linked list." << endl;
	while(head != nullptr)
	{
		temp = head;
		head = head->getLink();
		delete temp;
	}

	system("Pause");
	return 0;
}
```

The output of the program will be **4 3 2 1 0**. We note that when using the head_insert algorithm, the last node that has been inserted will be printed first. This is known as last in first out (LIFO) ordering.

Some remarks about the program:

- We observe that the head pointer should never be used for traversal except when deleting the nodes. Otherwise if we had used the head pointer for traversal (say for example for printing the nodes) then after printing the nodes the head pointer would have had a nullptr value which means it would not know where the first node is located in the heap memory. Moreover we wouldn't be able to re-assign the head pointer the memory address of the first node because we don't have a variable name for the first node as it is stored in the heap memory. This means we would have lost the first node. Then that would mean we wouldn't be able to locate the second node either because the only way to locate the second node is by going through the first node. It means we would also lose the third node and so on so forth. That is, we would have lost the entire linked list.
- When deleting a linked list however, it is perfectly fine to use the head pointer for traversal because we don't need to locate the nodes after they are deleted. Once all the nodes are deleted, we see that the head pointer will have a nullptr value which indicates the linked list has become empty which is correct.

Typically, we perform the head_insert, printing, and other operations of a linked list using functions. Thus the program given above modified to use functions for its head_insert, printing, and deleting the linked list operations would look like as follows.

```
void head_insert(NodePtr& head, const int& value)
{
	NodePtr n = new Node(value);
	n->setLink(head);
	head = n;
}
void print_linkedlist(const NodePtr& head)
{
	NodePtr temp = head;
	while(temp != nullptr)
	{
		cout << *temp << "  ";
		temp = temp->getLink();
	}
	cout << endl;
}
void delete_linkedlist(NodePtr& head)
{
	NodePtr temp;
	while(head != nullptr)
	{
		temp = head;
		head = head->getLink();
		delete temp;
	}
}
int main()
{
	//Create an empty linked list
	NodePtr head = nullptr;

	//Insert five nodes in the linked list using head_insert algorithm
	for (int i = 0; i < 5; i++)
```

```
                head_insert(head, i);

        //Print the linked list
        cout << "Printing the linked list." << endl;
        print_linkedlist(head);

        //Delete the nodes of the linked list from the heap memory
        cout << "Deleting the linked list." << endl;
        delete_linkedlist(head);

        system("Pause");
        return 0;
}
```

Some remarks about the functions:

- The head argument must pass by reference to the head_insert and delete_linkedlist functions because we want to modify its value in these functions and have the modification reflected on the main program as well. This means we will be passing a pointer by reference. The typedef name will make the syntax easier to understand.
- However it is not necessary to pass the head argument by reference to the print_linkedlist function; although we still do so for efficiency purposes and use const modifier to make sure the value of the head argument is not modified in the function.

Searching for a node in a linked list

We may also be interested to search if there exists a node in a linked list that satisfies some condition. For example, we could search for a node whose data is equal to a given value.

The search_node function given below takes the head pointer of a linked list and an integer value as its arguments and returns a pointer to a node in the linked list whose data is equal to the integer value argument.

If there is no any node in the linked list that satisfies the condition then the function will return nullptr. Moreover if there are two or more nodes that satisfy the condition then the function will return a pointer to the first node it finds that satisfies the condition.

```
NodePtr search_node(const NodePtr& head, const int& value)
{
        NodePtr temp = head;
        while(temp != nullptr)
        {
                if (temp->getData() == value)
                        return temp;
                else
                        temp = temp->getLink();
        }
        return nullptr;
}
int main()
{
        //Create an empty linked list
        NodePtr head = nullptr;

        //Insert five nodes in the linked list using head_insert algorithm
        for (int i = 0; i < 5; i++)
        {
```

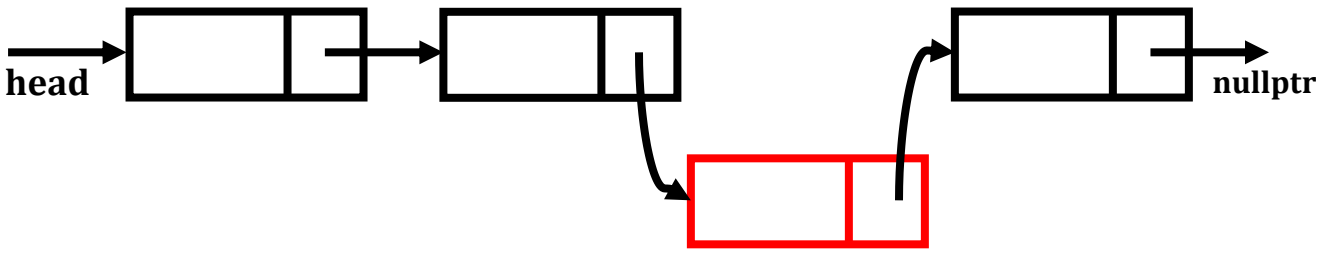
```
                head_insert(head, i);
        }

        //Print the linked list
        cout << "Printing the linked list." << endl;
        print_linkedlist(head);

        int search_value = 2;
        NodePtr n = search_node(head, search_value);
        if (n != nullptr)
                cout << "A node whose data is equal to " << n->getData() << " is found." << endl;
        else
                cout << "A node whose data us equal to " << search_value << " is not found." << endl;

        search_value = 8;
        n = search_node(head, search_value);
        if (n != nullptr)
                cout << "A node whose data is equal to " << n->getData() << " is found." << endl;
        else
                cout << "A node whose data us equal to " << search_value << " is not found." << endl;

        //Delete the linked list
        cout << "Deleting the linked list." << endl;
        delete_linkedlist(head);

        system("Pause");
        return 0;
}
```

Inserting nodes in a linked list: insert_after algorithm

We may also insert a new node after a specified node. For example, we may use the search_node function to search for a specific node and then insert a new node after that specified node.

The diagram below shows a given linked list with three nodes and the same linked list after a new node (shown in red color) has been inserted after the second node (that is between the second the third nodes). Such insertion operation is commonly known as insert_after operation.

Given linked list



The same linked list after inserting a new node after the second node in the linked list

In order to implement the insert_after function, we first note the following subtle points about the operation

- The function takes two arguments; namely a node pointer and an integer value.
- If the node pointer argument is a nullptr value then this function should return without inserting any new node because it doesn't make sense to insert after a nullptr.
- The new node's data will be the integer value argument and it will be inserted after the node pointed by the node pointer argument.
- This function does not require the head pointer of the linked list.

The following program demonstrates the insert_after function. Please note that it is not necessary to pass the node pointer argument to the function by reference because we will not modify its value. Instead we are dereferencing it and therefore even if we use parameter passing by value then it would still be fine.

```
void insert_after(const NodePtr& n, const int& value)
{
	if (n == nullptr)
		return;
	else
	{
		NodePtr temp = new Node(value);
		temp->setLink(n->getLink());
		n->setLink(temp);
	}
}
int main()
{
	//Create an empty linked list
	NodePtr head = nullptr;

	//Insert five nodes in the linked list using head_insert operation
	for (int i = 0; i < 5; i++)
	{
		head_insert(head, i);
	}

	//Print the linked list. This will print 4,3,2,1,0
	cout << "Printing the linked list." << endl;
	print_linkedlist(head);

	//Insert a new node with data equal to -4 after the node whose data is equal to 2
	int search_value = 2;
	NodePtr n = search_node(head, search_value);
	insert_after(n, -4);

	//Print the linked list. This will print 4,3,2,-4,1,0
	cout << "Printing the linked list." << endl;
	print_linkedlist(head);

	//Insert a new node with data equal to -6 after the node whose data is equal to 8
	search_value = 8;
	n = search_node(head, search_value);
	insert_after(n, -6); //This will not insert because n will be nullptr

	//Print the linked list. This will still print 4,3,2,-4,1,0
	cout << "Printing the linked list." << endl;
	print_linkedlist(head);

	//Delete the nodes from the heap
	cout << "Deleting the linked list." << endl;
	delete_linkedlist(head);
```

```
	system("Pause");
	return 0;
}
```

Inserting nodes in a linked list: tail_insert algorithm

We may also be interested to insert a new node as a last node in a given linked list. Then we will have two cases to consider:

- If the linked list is empty then this function will insert the new node as the first node of the linked list (which will also be the last node in the linked). Thus the insertion will be performed using the head_insert function.
- Otherwise we will first find the last node of the linked list and then we will insert after the last node using the insert_after function.

The following program demonstrates the tail_insert function.

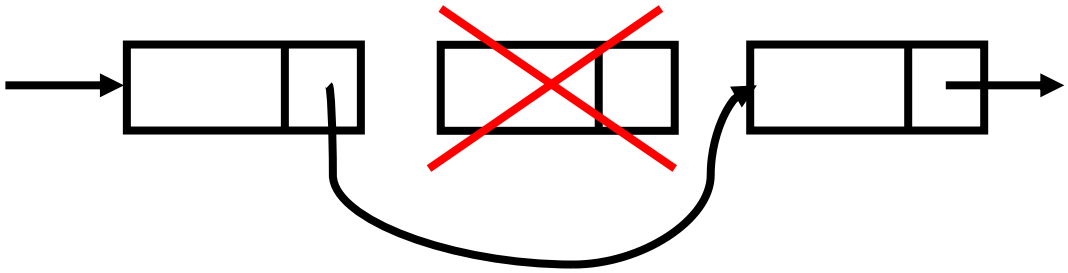
```
void tail_insert(NodePtr& head, const int& value)
{
	if (head == nullptr)
		head_insert(head, value);
	else
	{
		//Find the last node in the linked list
		NodePtr b = head; //Here b is guaranteed to be not nullptr
		while(b->getLink() != nullptr)
			b = b->getLink();
		//Now insert after b
		insert_after(b, value);
	}
}
int main()
{
	srand(time(0));

	//Create an empty linked list
	NodePtr head = nullptr;

	//Insert ten random integers using insert_increasing function
	for (int i = 0; i < 5; i++)
		tail_insert(head, i);

	//Print the linked list. This will still print 0,1,2,3,4
	cout << "Printing the linked list." << endl;
	print_linkedlist(head);

	//Delete the nodes from the heap
	cout << "Deleting the linked list." << endl;
	delete_linkedlist(head);

	system("Pause");
	return 0;
}
```

The output of the program will be **0 1 2 3 4**.

Removing a specified node from a linked list: remove_node

We may also remove a specified node from a linked list. The following diagram shows a given linked list and the same linked list after the second node (crossed in red color) is removed.

Given linked list



The same linked list after the second node has been removed from the linked list

We observe that in order to remove a node from a linked list, we first need to find the node that is just before it in the linked list. The following program demonstrates the remove_node function. The function will take the head of a linked list and a node pointer pointing the node to be removed as arguments and it will remove the node pointed by the node pointer from the linked list. Of course this function will not remove a node if any of the following is true

- The head of the linked list argument is nullptr (that is the linked list is empty), or
- The node pointer argument is nullptr, or
- The node pointer is not found in the linked list.

Moreover the function must pass the head pointer argument by reference because we need to modify the head pointer if the node to be removed happens to be the first node.

```
void remove_node(NodePtr& head, const NodePtr& n)
{
	if (head == nullptr || n == nullptr)//empty linked list or nullptr n
		return;
	else if (head == n)
	{
		head = head->getLink();
		delete n;
		return;
	}
	else
	{
		NodePtr b = head;
		while (b != nullptr) //Here b is guaranteed to be not nullptr
		{
			if (b->getLink() == n) //found the node before n
			{
				b->setLink(n->getLink());
				delete n;
				return;
			}
			b = b->getLink();
		}
	}
}
```

```
int main()
{
	//Create an empty linked list
	NodePtr head = nullptr;

	//Insert five nodes in the linked list using head_insert operation
	for (int i = 0; i < 5; i++)
	{
		head_insert(head, i);
	}

	//Print the linked list. This will print 4,3,2,1,0
	cout << "Printing the linked list." << endl;
	print_linkedlist(head);

	//Remove the node in the linked list whose data is 2
	int search_value = 2;
	NodePtr n = search_node(head, search_value);
	remove_node(head, n);

	//Print the linked list. This will print 4,3,1,0
	cout << "Printing the linked list." << endl;
	print_linkedlist(head);

	//Remove the node in the linked list whose data is 6
	search_value = 6;
	n = search_node(head, search_value);
	remove_node(head, n);

	//Print the linked list. This will still print 4,3,1,0
	cout << "Printing the linked list." << endl;
	print_linkedlist(head);

	//Delete the nodes from the heap
	cout << "Deleting the linked list." << endl;
	delete_linkedlist(head);

	system("Pause");
	return 0;
}
```

We may also be interested to remove a node whose data is equal to a specified integer value. This can easily be done by calling the search_node function in order to first find a node pointer pointing to the node to be removed as follows.

```
void remove_node(NodePtr& head, const int& value)
{
	NodePtr n = search_node(head, value);
	remove_node(head, n);
}
```

Removing all nodes satisfying a condition from a linked list: remove_all

We may also be interested to remove all nodes from a linked list whose data is equal to some specified integer value. This can easily be done using a loop and the search_node and remove_node functions as follows.

```
void remove_all(NodePtr& head, const int& value)
{
	NodePtr n;
	do
	{
		n = search_node(head, value);
		remove_node(head, n);
	}while (n != nullptr);
}
int main()
{
	//Create an empty linked list
	NodePtr head = nullptr;

	//Insert ten nodes in the linked list using head_insert operation
	for (int i = 0; i < 10; i++)
	{
		head_insert(head, i/3);
	}

	//Print the linked list. This will print 3,2,2,2,1,1,1,0,0,0
	cout << "Printing the linked list." << endl;
	print_linkedlist(head);

	//Remove all the nodes whose data is 2
	remove_all(head, 2);

	//Print the linked list. This will print 3,1,1,1,0,0,0
	cout << "Printing the linked list." << endl;
	print_linkedlist(head);

	//Delete the nodes from the heap
	cout << "Deleting the linked list." << endl;
	delete_linkedlist(head);

	system("Pause");
	return 0;
}
```

Inserting nodes in a linked list: insert_before algorithm

We may also insert a new node before a specified node in a linked list. In this case, we first need to find the node just before the specified node and then use insert_after function.

In order to mimic the insert member function in STL containers, we will design this function with the following constraints

- If the linked list is empty and the specified node is nullptr then this function should do head_insert.
- If the linked list is empty and the specified node is not nullptr then the specified node cannot be found in the linked list and therefore an error message should be printed and this function should not insert any new node.
- If the linked list is not empty and the specified node is nullptr then this function should do tail_insert.
- If the linked list is not empty and the first node in the linked list is equal to the specified node then this function should do head_insert.

- If the linked list is not empty and the specified node is not nullptr but that it is not found in the linked list then an error message should be printed and this function should not insert any new node.
- Otherwise a node just before the specified node will be found and this function should insert after it using the insert_after function.

The following program demonstrates the insert_before function with a test main program designed to test all the above six different cases.

```
void insert_before(NodePtr& head, const NodePtr& n, const int& value)
{
	//If the linked list is empty then insert only if n is nullptr too
	if (head == nullptr)
	{
		if (n == nullptr)
			head_insert(head, value);
		else
			cout << "Insertion failed. Bad node argument." << endl;
		return;
	}
	//If n is nullptr then tail_insert
	if (n == nullptr)
	{
		tail_insert(head, value);
		return;
	}
	//If head is equal to n then head_insert
	if (head == n)
	{
		head_insert(head, value);
		return;
	}
	//Find the node just before n and insert after it
	NodePtr b = head;
	while (b != nullptr)
	{
		if (b->getLink() == n)
			break;
		b = b->getLink();
	}
	if (b == nullptr)
		cout << "Insertion failed. Bad node argument." << endl;
	else
		insert_after(b, value);
}
int main()
{
	srand(time(0));

	//Create an empty linked list
	NodePtr head = nullptr;

	//Insert a node with data 4 before a non-existing node
	Node n; //Default Node object
	insert_before(head, &n, 4);
	cout << "Printing the linked list." << endl;
	print_linkedlist(head);//Prints empty linked list

	//Insert a node with data 6 before nullptr
	insert_before(head, nullptr, 6);
```

```
        cout << "Printing the linked list." << endl;
        print_linkedlist(head); //Prints 6

        //Insert a node with data 3 before nullptr
        insert_before(head, nullptr, 3);
        cout << "Printing the linked list." << endl;
        print_linkedlist(head); //Prints 6, 3

        //Insert a node with data 8 before 6
        insert_before(head, search_node(head, 6), 8);
        cout << "Printing the linked list." << endl;
        print_linkedlist(head); //Prints 8, 6, 3

        //Insert a node with data 5 before 6
        insert_before(head, search_node(head, 6), 5);
        cout << "Printing the linked list." << endl;
        print_linkedlist(head); //Prints 8, 5, 6, 3

        //Insert a node with data 9 before a non-existing node
        insert_before(head, &n, 9);
        cout << "Printing the linked list." << endl;
        print_linkedlist(head); //Prints Prints 8, 5, 6, 3

        //Delete the nodes from the heap
        cout << "Deleting the linked list." << endl;
        delete_linkedlist(head);

        system("Pause");
        return 0;
}
```