Linked Lists

Topics:

- Nodes
- Node Insertion
- Node Removal
- Copying A Linked List
- Destroying A Linked List

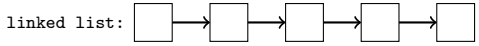
Resources:

- `Node.h`
- `Repository.h`
- `main.cpp`

Introduction

Currently, we have been constructing data structures with arrays; however, we can use an alternative storage method that typcially has a constant Big-O runtime for insertions and removal methods. It is called a *linked list*. An array, as you know is a block of sequential memory cells that can be referenced with an index

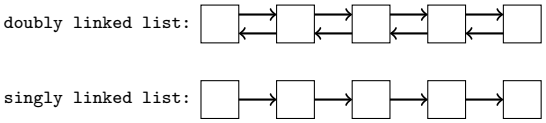
Whereas, a linked list is a collection of individual independent memory cells that links to each other to form a list called *nodes*.



Because a linked list is constructed this way, the list is always the exact size of the collection; this means you will never need to allocate unnecessary additional space to anticipate future data. However, to get to a node in a linked list, you need to traverse through all the nodes that precede it if you do not have a reference to it.

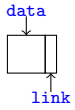
Nodes

A node is a data structure that consist of data and a reference to a node (node pointer) called a *link*. A linked list is typically constructed using one of two types of nodes: *singly-linked nodes* or *doubly-linked nodes*. A singly-linked node is a node that has a single link, and a doubly-linked node is a node that has two links. Linked list constructed from singly-linked nodes (singly linked list) can sequentially move in only one direction; whereas, linked list constructed from doubly-linked nodes (doubly linked list) can sequentially move in both directions.

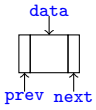


where the actually data structure and a more detailed illustration for each node is

```
template <class T>
struct Node
{
 T data;
 Node<T>* link;
};
```

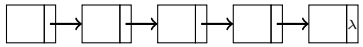



```
template <class T>
struct Node
{
 T data;
 Node<T>* prev;
 Node<T>* next;
};
```

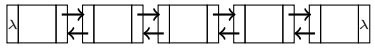


where the left is a singly-linked node and the right is a doubly-linked node. Linked lists are typically a collection of dynamically allocated nodes; hence, links, which are node pointers, are referencing allocated memory. This means you need to be careful whenever you are adding or removing nodes from a linked list to avoid causing memory leaks. Whenever, you add and remove from a linked list, you must make sure that the list is in a consistent state, which means that every node is referencing the correct node and there are no holes between nodes.

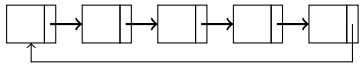
Furthermore, when you construct a linked list, you typically make it either a *null-terminated* linked list or a *circular* linked list. For a null-terminated singly linked list, the link of the last node of the linked list is null.



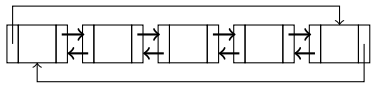
For a null-terminated doubly linked list, the previous link of the first node and the next link of the last node are null.



For a circular singly linked list, the link of the last node is the first node.



And for a circular doubly linked list, the previous link of the first node is the last node and the next link of the last node is the first node.



Last, at stated before, a linked list is typically a collection of dynamic allocated nodes; hence, it is best to define a function that creates new nodes. These functions should accept the data of the node as a parameter and assign NULL to the link(s) of the node.

These functions will be defined as follows

```
template<typename T>
Node<T>* Create(const T& data)
{
 Node<T>* t = new Node<T>;
 t->data = data;
 t->link = NULL;
 return t;
}
```

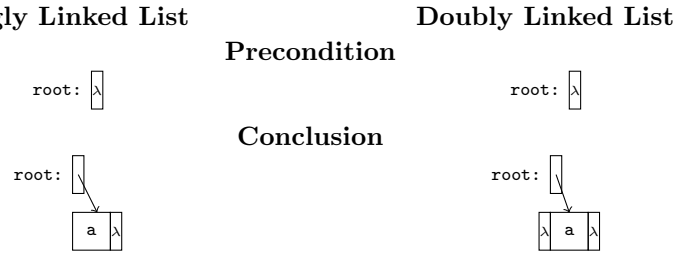
```
template<typename T>
Node<T>* Create(const T& data)
{
 Node<T>* t = new Node<T>;
 t->data = data;
 t->prev = NULL;
 t->next = NULL;
 return t;
}
```

where the left is the singly-linked node definition and the right is the doubly-linked node definition. For the remainder of this lecture, all implementation will be for null-terminated linked list. Before we precede, it is important to note that you should always have a stationary reference of the head (the first) node in a linked list especially for singly linked lists; getting access to previous nodes is not always possible for multiple reasons.

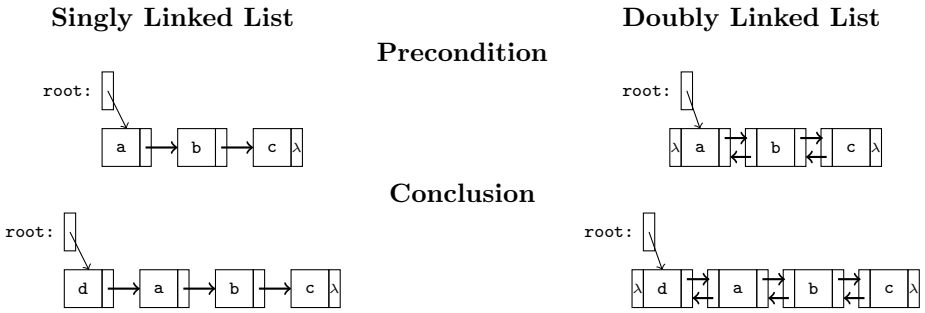
Node Insertion

For each of these insertion functions, we will be focus on inserting a new node after a given node in a linked list. The parameters of the insertion functions will be *root*, *node* and *data* respectively where *root* is the reference to the head of the linked list, *node* is the node of the linked link that the new node will be placed after, and *data* is the data of the new node. Furthermore, when we are inserting a new node to a linked list we need to consider the possible scenarios, which are

Scenario 1: The list is empty. In this scenario, the new node will become the head of the list.



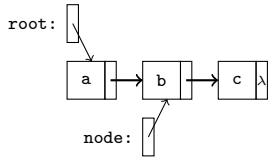
Scenario 2: The list is not empty and the new node has to be inserted at the beginning of the of the list (*node* will be equal to NULL to indicate this). In this scenario, the current linked list head will become the link (or next link) of the new node and *root* will be assigned the new node. Furthermore, for the doubly linked list, the prevoius node of the original head will become the new node.

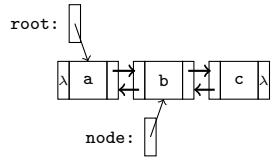


Scenario 3: The list is not empty and *node* is a random node of the list. In this scenario, the link (or next link) of *node* becomes the link (or next link) of the new node; and then, the new node becomes the link (or next link) of *node*. Furthermore, for the doubly linked list, the previous link of the old next link of *node* becomes the new node if it is not null and the previous link of the new node becomes *node*.

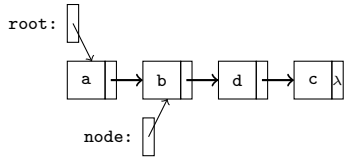
Singly Linked List **Doubly Linked List**

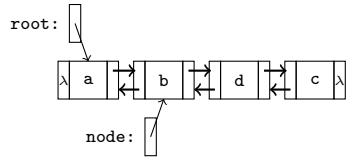
Precondition





Conclusion





Now that we established all of the different scenarios, we can define the insertion functions below

```
template<typename T>
void Insert(Node<T>*& root,Node<T>* node,const T& data)
{
 if(root == NULL)
 {
  root = Create(data);
 }
 else if(node == NULL)
 {
  Node<T>* t = Create(data);
  t->link = root;
  root = t;
 }
 else
 {
  Node<T>* t = Create(data);
  t->link = node->link;
  node->link = t;
 }
}
```

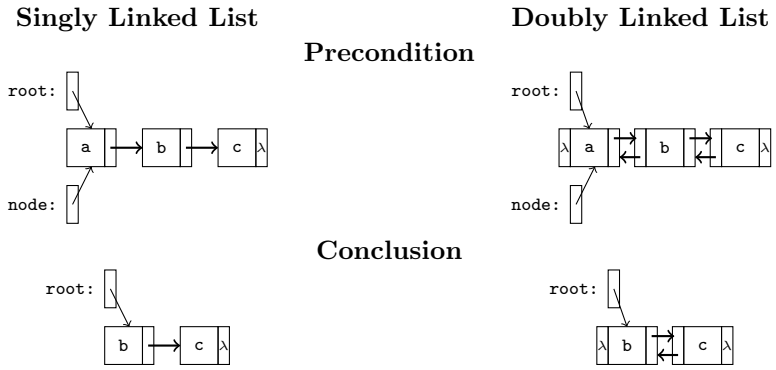
```
template<typename T>
void Insert(Node<T>*& root,Node<T>* node,const T& data)
{
 if(root == NULL)
 {
  root = Create(data);
 }
 else if(node == NULL)
 {
  Node<T>* t = Create(data);
  t->next = root;
  root->prev = t;
  root = t;
 }
 else
 {
  Node<T>* t = Create(data);
  t->next = node->next;
  node->next = t;
  t->prev = node;

  if(t->next != NULL)
  {
   t->next->prev = t;
  }
 }
}
```

Node Removal

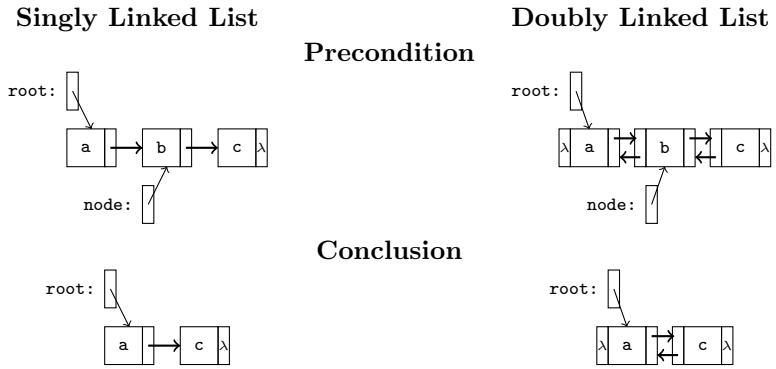
For each of these removal functions, we will be focus on removing the node provided from a given linked list. The parameters of the removal functions will be *root* and *node* respectively where *root* is the reference to the head of the linked list, *node* is the reference to the node from the linked list that will be removed. Like with the insertion functions, we need to consider the possible scenarios, which are

Scenario 1: The list is empty or *node* is null. In this scenario, we do nothing. If the linked list is empty, there is nothing to remove, and if *node* is null, it is not a member of the linked list.

Scenario 2: The list is not empty and *node* is the head of the linked list. In this scenario, *root* will become its link (or next link), and then, *node* will be deallocated. Furthermore, for the doubly linked list, the previous node of the new *root* will become null.



Scenario 3: The list is not empty and *node* is a random node of the list. In this scenario, the link (or next link) of *node* becomes the link (or next link) of the node that precedes *node*; and then, *node* will be deallocated. Furthermore, for the doubly linked list, the previous link of the old next link of *node* becomes the node that precedes *node* if it is not null.



Now that we established all of the different scenarios, we can define the removal functions below

```
template<typename T>
void Remove(Node<T>*& root,Node<T>* node)
{
 if(root != NULL && node != NULL)
 {
  if(root == node)
  {
   root = root->link;
   delete node;
   node = NULL;
  }
  else
  {
   Node<T>* t = root;

   while(t->link != node)
   {
    t = t->link;
   }
   t->link = node->link;
   delete node;
   node = NULL;
  }
 }
}
```

```
template<typename>
void Remove(Node<T>*& root,Node<T>* node)
{
 if(root != NULL && node != NULL)
 {
  if(root == node)
  {
   root = root->next;
   delete node;
   node = NULL;

   if(root != NULL)
   {
    root->prev = NULL;
   }
  }
  else
  {
   node->prev->next = node->next;

   if(node->next != NULL)
   {
    node->next->prev = node->prev;
   }
   delete node;
   node = NULL;
  }
 }
}
```

Copying A Linked List

When a linked list is copied, a deep copy needs to be performed. This implies that a new linked list needs to be created such that the data of its nodes match the data of the nodes of the original linked list in the same position.

Singly Linked List **Doubly Linked List**

Precondition

Conclusion

The accomplish this task,

1. Check if the original linked list is empty. If it is empty, return null.
2. The head of the new linked list needs to be create such that its data equals the data of the head of the original linked list.
3. Create a temporary node pointer for both the original linked list and the new linked list that will be assigned the heads of each list.
4. Loop through the original linked list using the temporary node for the original linked list until the link (or next link) of the temporary node equals null.
5. For each instance of the loop, create a new node for the link (or next link) of the temporary node for the new linked list such that its data equals the data of the link (or next link) of the temporary node for the original linked list.
6. Afterwards, assign both temporary nodes their links (or next links).
7. After the loop terminates, return the head of the new linked list.

For the doubly linked list, the previous link of the next link of the temporary node of the new linked list is assigned to temporary node within the body of the loop as well. The definitions of these copy functions are below

```
template<typename T>
Node<T>* Copy(Node<T>* root)
{
 if(root == NULL)
 {
  return NULL;
 }
 else
 {
  Node<T>* cp = Create(root->data);
  Node<T>* ta = root;
  Node<T>* tb = cp;

  while(ta->link != NULL)
  {
   tb->link = Create(ta->link->data);
   ta = ta->link;
   tb = tb->link;
  }
```

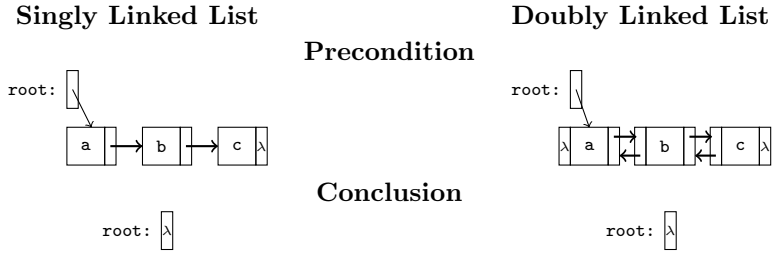
```
template<typename>
Node<T>* Copy(Node<T>* root)
{
 if(root == NULL)
 {
  return NULL;
 }
 else
 {
  Node<T>* cp = Create(root->data);
  Node<T>* ta = root;
  Node<T>* tb = cp;

  while(ta->next != NULL)
  {
   tb->next = Create(ta->next->data);
   tb->next->prev = tb;
   ta = ta->next;
   tb = tb->next;
```

```
  return cp;
 }
}
```

```
  }
  return cp;
 }
}
```

Destroying A Linked List

When a linked list is destroyed, the head does not need to be maintained. Process is repetitively removing the head of the linked list until the linked list is empty.



For the doubly linked list, you do not have to worry about the previous link, so the function look practically identical. The definitions of these clear functions are below

```
template<typename T>
void Clear(Node<T>*& root)
{
 Node<T>* t;

 while(root != NULL)
 {
  t = root;
  root = root->link;
  delete t;
  t = NULL;
 }
}
```

```
template<typename>
void Clear(Node<T>*& root)
{
 Node<T>* t;

 while(root != NULL)
 {
  t = root;
  root = root->next;
  delete t;
  t = NULL;
 }
}
```