November 24, 2020

Evan Delasota

cspc-311 section 03

Implementing A SINGLY LINKED List in c++

a step-by-step tutorial



# Preliminary

This tutorial requires the reader to understand the programming language C++ at a beginner level, as it will assume knowledge of pointers, data types, usage of the standard library, conditional statements, and class creation.

Please look at the links below if you need a refresher on the language or need to learn it!

Resources for learning C++:

<https://www.geeksforgeeks.org/c-plus-plus/>

<https://www.learncpp.com/>

Recommended read from Bjarne Stroustrup, the creator of C++:

<https://www.stroustrup.com/4th.html>

## Tools Needed

An IDE (recommended) or text editor

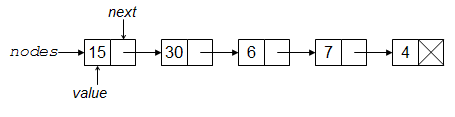
This tutorial will be using Microsoft’s Visual Studio IDE.

Download Here: <https://visualstudio.microsoft.com/downloads/>

# Introduction

A linked list is a type of data structure that represents a sequence, or chaining of nodes where each node contains some value and is placed in a random location in memory.

There are three types of linked lists, a singly linked list, a doubly linked list, and a circular linked list. The list we will be implementing in this tutorial will be a singly linked list. This data structure features a node with a value and a link to the next node in the list.



[This Photo](https://en.wikibooks.org/wiki/Data_Structures/All_Chapters) by Unknown Author is licensed under [CC BY-SA](https://creativecommons.org/licenses/by-sa/3.0/)

Figure - Singly Linked List

## Advantages and Disadvantages

Using this data structure harbors great insertion and deletion speed, but falters slightly when accessing any elements located in the middle of the list, as it cannot access nodes outside of the head and tail nodes without traversing through the list. As you might’ve guessed, this structure is often used for queues and stacks.

Table - Big-O Analysis

|  |  |
| --- | --- |
| Element Access | O(n)  O(1) access to head/tail elements |
| Insertion/Deletion from Beginning | O(1) |
| Insertion/Deletion from middle | O(n) |
| Insertion/Deletion at End | O(n)  Note: O(1) can be achieved for insertion with a tail pointer |

# Implementation

Once you have everything installed and ready, go ahead and startup Visual Studio and create a new project.

## Setup

You may put everything into one cpp file, but for best practices we will be splitting this implementation into three files:

1. LinkedListTutorial.cpp – Your driver cpp file, what you’ll be using to test out your linked list.

2. LinkedList.h – Your class header file, used to declare the LinkedList class and all its respective methods and variables.

3. LinkedList.cpp – Your class cpp file, used to define the declared methods located in the header file.

I highly recommend doing it in this fashion as it increases readability by a considerable amount.

## LinkedList Class Declarations

Go to your LinkedList.h file to begin.

First, we will be creating our node element in the form of a struct consisting of two variables:

*int**val* – Stores the data value

*node\***next* – Stores the address of the next node in the list

Lastly, we will create the inline method *int* *getVal()* to return a node’s data value.

Graphical user interface, text, application

Description automatically generated

*Note:* *We’re including <cstddef> for the NULL type and <iostream> for output statements to the command line.*

Next, we will be creating our LinkedList class.

The *private* section will consist of three variables:

*node\* head* – The pointer pointing to the beginning of the list

*node\* tail* – The pointer pointing to the end of the list

*int**size* – The size of the list

In the *public* section, we will begin by declaring several methods used for this implementation.

**Class Constructors/Destructors**:

*LinkedList()* – Constructor

*~LinkedList()* – Empty Destructor

**Helper Methods:**

*void newNode(int)* – Creates a new node to be added to the list

*int getIndexValue(int)* – Returns the value of the target index if it exists

*void print()* – Prints to console the contents of the linked list

**Insertion Methods:**

*void insertAtPosition(int, int)* – Creates a new node with the given value, then inserts the node at the target index and increments the *size* variable by 1.

*void insertAtHead()* – Inserts a node at the head of the list using *insertAtPosition(int, int)*

*void insertAtTail()* – Inserts a node at the tail of the list using *insertAtPosition(int, int)*

**Deletion Methods:**

*void deleteAtPosition(int)* – Deletes the node at the target index, then decrements the *size* variable by 1.

*void deleteAtHead()* – Deletes the node located at the head of the list, uses *deleteAtPosition(int)*

*void deleteAtTail()* – Deletes the node located at the tail of the list, uses *deleteAtPosition(int)*

A screenshot of a computer screen

Description automatically generated

Your class header file should resemble something like this.

## Linked List Class Definitions

Once the header file is complete, continue by going to your LinkedList.cpp file.

### Class Constructor

In the constructor simply assign the node values to NULL and the size value to 0 since our newly created linked list is empty.

A screenshot of a computer screen

Description automatically generated

Figure - Class constructor

### Helper Methods

Next, we will be writing a few methods that make will make our life easier for testing and bring a bit of versatility to what we can do with our data structure!

First, we must create some type of method that creates and appends a node to our linked list, let’s call it *newNode(int value)*. First create a new node pointer temp, assign its *val* value from the given argument, and assign its *next* variable to NULL.

Now, through an if-else conditional statement we will consider where we put the newly created node in our list. If the head of the list is pointing to NULL, we will assign the temp node to the head and tail variables and assign the temp variable to NULL for garbage collection.

Otherwise, we will append the newly created node to the end of the list. We can do this by assigning the tail’s next value to temp, and then assigning tail to temp. By doing this we now have our tail correctly pointing to the last node again, which happens to be our newly created node. Lastly, increment the size of the list by 1.

You should have something like the illustration below.

Text

Description automatically generated

Figure – newNode method

Our next method *getIndexValue(int target)* traverses through our linked list and finds the value of the targeted index.

There are two common errors we must safeguard against:

* + If the target index is larger than the size of the list
  + If the target is less than 0

We can implement this by simply using a conditional IF-statement checking for either errors.

If the target index is valid, we declare a node pointer and assign it to the head of the linked list.

Now, we enter a for loop for as long as i is less than the target index – 1 (decrement to account for 0th index) and assign the node pointer to the next pointer address.

At this point the loop should stop, and from here simply return the node pointer’s data value using *getVal()*.

Our last helper method *print()* simply traverses through the linked list printing out the position and value of the node(s).

Do this by assigning a node pointer to the head and use a for loop to traverse through the list, using cout and *getVal* to print the position and values until the node pointer becomes NULL. Make sure to assign the node pointer to the next pointer address much like the method discussed previously.

Text

Description automatically generated

Figure - getIndexValue & print methods

### Insertion

Now that our helper methods are complete and ready to be used, let’s continue to our insertion methods.

Our primary insertion method, *insertAtPosition(int target, int value)* creates a node of a given value, and inserts it at the targeted index.

First, we will consider any errors in user input:

* + A target index greater than the size of the linked list
  + A target index less than the size of the linked list

Again, do this through an IF statement and return if either of these conditionals are true.

Next, we need to create three node pointers for this method:

1. A previous node pointer to keep track of the previous node that the current pointer pointed to.
2. A current node pointer assigned to the head of the list for traversing.
3. A temp node pointer to be declared and assigned, then inserted into the target index.

Then we’ll enter an IF-ELSE statement, depending on if the head of the list is NULL or if the target is equal to the size of the list.

* If the head is empty, simply assign our temporary node the value given and have its next variable point to the empty head pointer. Lastly, assign head to our temp node to complete the process.
* Else, if our target index is equal to the size, assign the given value to temp, assign tail’s next variable to temp and lastly assign tail to temp.
* Otherwise, we’ll begin by entering our for-loop iterating through our list using our current node and assigning the previous node to the current node. After looping through our list, assign our temporary node the given value, assign the next variable for temp to current, and prev node’s next variable to temp.

Once this is complete exit the conditional and increment the size of the list.

For *insertAtHead(int value)* all we really need to do is call *insertAtPosition(0, value)* inside of our method definition.

For *insertAtTail(int value)* again, simply call *insertAtPosition(size, value)* inside of our method definition.

### 

Figure - Insertion Methods

### Deletion

Finally, we’ll work on our deletion methods. Our primary deletion method *deleteAtPosition(int target)* is given a target index and removes it from the list, decrementing the size in the process.

Much like our previous methods, let’s error check the user input:

* If the target index is greater than the size of the list
* If the target index is less than 0

Check these using an IF-statement and return if true.

For this method we need to create two node pointers:

1. A previous node to keep track of the previous node that the current node pointed to, will be used for any middle or tail deletions.
2. A current node to traverse the list until our target index is reached.

Now, enter an IF-ELSE statement, depending on whether the target is at the head of the list or not.

If yes, assign the prev node to the head node, then assign the head node to head’s next variable.

Otherwise, enter a FOR-loop iterating through our list using our current node and assigning the previous node to the current node. Once we reach our target index, assign prev’s next variable to current’s next variable. By doing so you essentially break the chain of the list and re-link the previous node to the next node after current.

Now, exit the conditional and decrement the size by 1.

For *deleteAtHead()*, call *deleteAtPosition(0)* in the definition.

For *deleteAtTail()*, call *deleteAtPosition(size – 1)* in the definition. Make sure to decrement the argument by one to account for the 0th index.

Text

Description automatically generated

Figure - Deletion Methods

# Testing Our Data Structure

Now that we’ve fleshed out all our class functions, let’s go ahead and test out our linked list in our driver file, *LinkedListTutorial.cpp*.

Here is an example of what to test:

A screenshot of a computer screen

Description automatically generated

Figure - Testing in Our Driver File

## Where to Go from Here

Congratulations! We’ve completed our implementation of a singly linked list and can now play around with it. One thing to note is that since this is a very basic implementation, there are many things you can add on to this data structure, such as more robust error checking, building a doubly linked list, or even a circular linked list. Hopefully this was a useful tool in understanding the mechanisms of how a linked list operates.

See the full implementation here: <https://github.com/evan-delasota/Linked-List-Tutorial>