Sorting Project Documentation

Version v1.0 (4/26/2019)

# Introduction

This project demonstrates the insertion and quick sort algorithms, by importing a list of values from an external file, and inserting those values into a doubly-linked list. The linked list header file is the implementation that was built by me earlier this semester.

For more information on the linked list implementation, please see the attached [Linked List Library Documentation](#_wq1tiikrsyih).

# **Analysis**

According to my measurements and testing (see the following charts), both the insertion and quick sort implementations used in this program meet their expected worst-case time complexities, both of which are O(n2).

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Runtime unit** | **10** | **100** | **1000** | **10,000** | **100,000** |
| **CPU Cycles** | 0 | 1.2 | 110.7 | 10751.5 | 9423228 |
| **Seconds** | 0 | 0.0012 | 0.1107 | 10.7515 | 9423.228 |

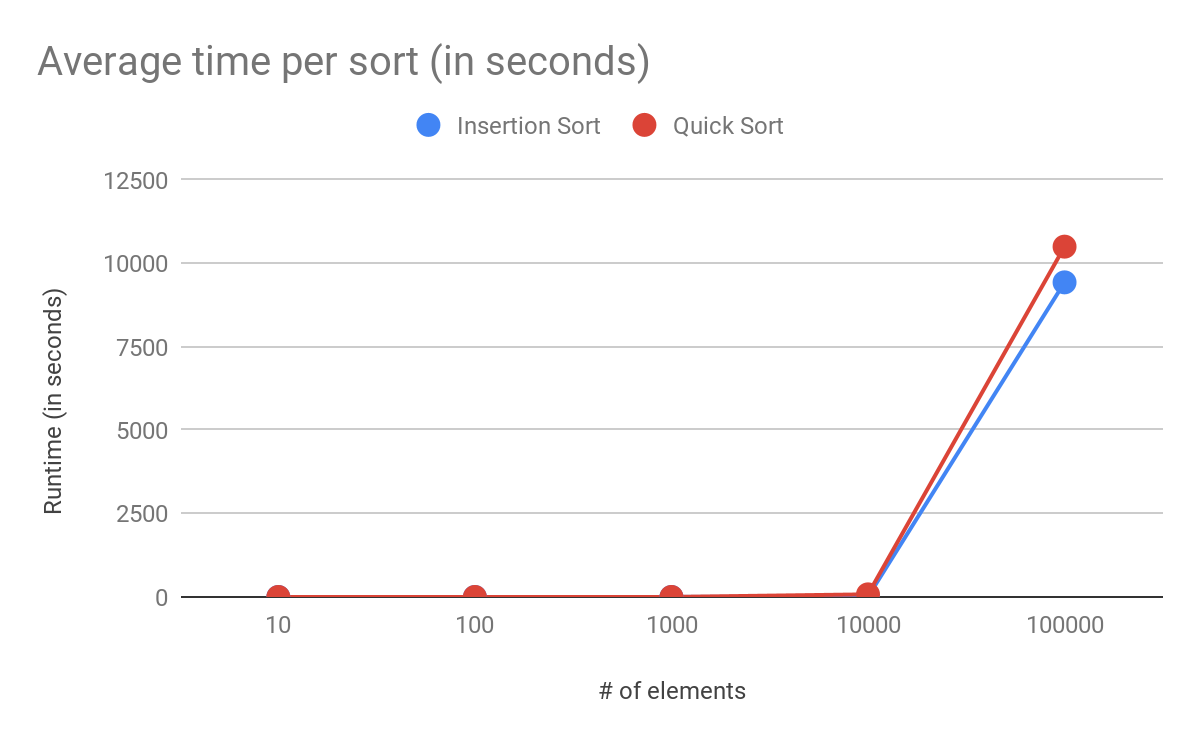
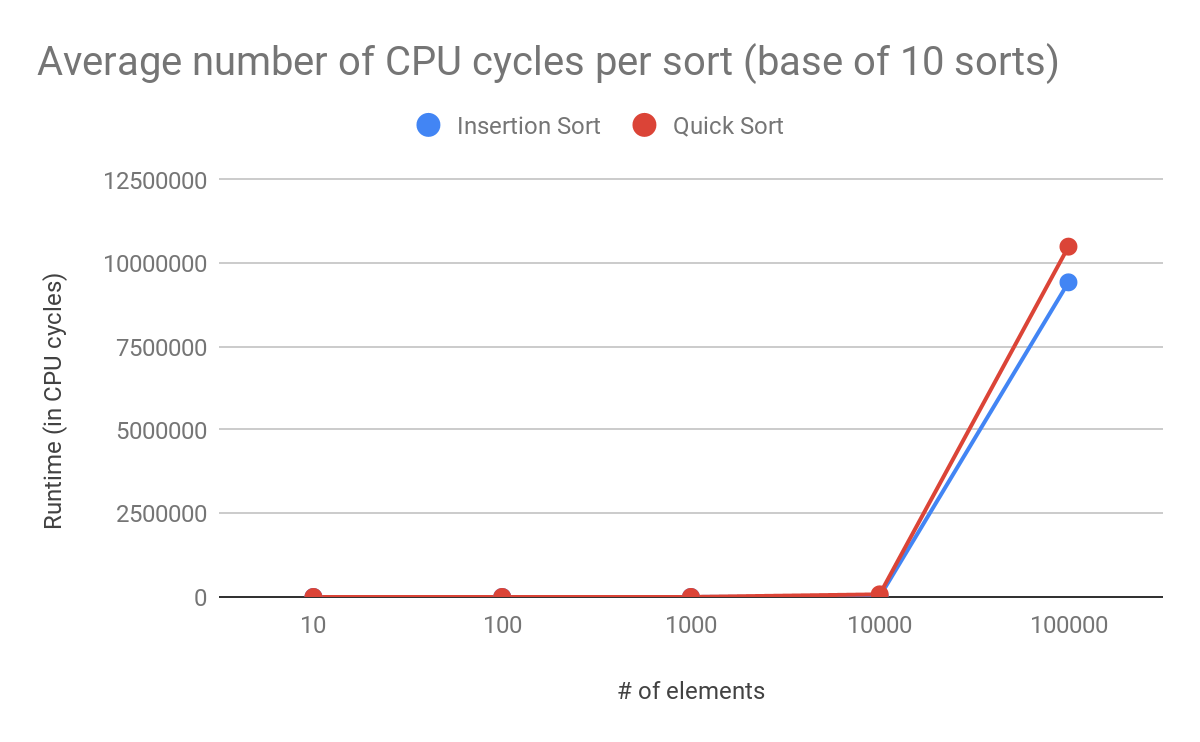
Figure 1: Average runtime for the insertion sort algorithm implementation

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Runtime unit** | **10** | **100** | **1000** | **10,000** | **100,000** |
| **CPU Cycles** | 0 | 4.7 | 518.9 | 78050 | 10492596 |
| **Seconds** | 0 | 0.0047 | 0.5189 | 78.05 | 10492.596 |

Figure 2: Average runtime for the quick sort algorithm implementation

Notes:

* Both sorting algorithms perform well on the included datasets with sizes of 10, 100, and 1000 elements, with sub-1 second average runtimes.
* The insertion algorithm performs reasonably well up to 10,000 elements, with an average runtime of just over 10 seconds.
* However, the quick sort takes roughly 78 seconds to sort 10,000 elements.
* Both algorithms take an extremely long time to sort 100,000 elements. The insertion took roughly 2 ½ hours, while the quick sort took nearly 3 hours.
* As a result of the incredibly long runtimes for 100,000 nodes, I didn’t even both to test the 1,000,000 element lists. I had better things to do. (Like watch grass grow!)



Linked List Library Documentation

Version v1.3 (4/26/2019)

# Introduction

This program is a sample implementation of a doubly-linked list using generic programming in C++. It will store- a value in a node, and maintain a list of nodes in the order they were created, with each node containing a pointer to the node previous to and immediately following itself.

The user interacts with the program through a simple console-based menu. They are able to perform the following functions:

* **Insert** a new node at the end of the linked list.
* **Delete** a node from the linked list that has a key value matching a value of the user’s choosing. The program will only delete the first instance of the inputted value (it uses the Find function to locate this node).
* **Find** a node containing a key value.matching a value of the user’s choosing. The program will only find the first instance of the inputted value.
* Determine the lowest (**minimum**) value stored in the linked list.
* Determine the highest (**maximum**) value stored in the linked list.
* **Display** the contents of the linked list, in both **forward** and **reverse** orders.
* **Auto-populate** a random number of nodes with a randomized value.
* **Display** a **detailed** list of the contents of the linked list. (Each node’s key value, as well the key value and memory address of that node’s adjacent nodes.
* **Exit** the program.

# 

# Programmer’s Guide

## <LinkedList.h>

template <typename T> class LinkedList

A custom-built doubly-linked list class. The class is templated as to allow for use with a wide variety of data types.

|  |  |  |
| --- | --- | --- |
| Attributes/Data Members (private) | | |
| Name | Type | Description |
| nodeCount | int | Stores a count of the number of nodes in the list. Initializes to 0. |
| headNode | LL\_Node<T>\* | Stores a pointer to the first node in the list. Initializes to nullptr. |
| tailNode | LL\_Node<T>\* | Stores a pointer to the last node in the list. Initializes to nullptr. |

|  |  |  |  |
| --- | --- | --- | --- |
| Member Functions (private) | | | |
| Signature | |  | |
| Name | Type | Description & Parameters | |
| DecreaseNodeCount | void | Decreases the count of the number of nodes stored in the linked list. | |
| IncreaseNodeCount | void | Increases the count of the number of nodes stored in the linked list. | |
| PrintDetails | void | Outputs the key values and addresses of the current node and its adjacent nodes.  LL\_Node<T>\* targetNode - A pointer to the node object that will have its details printed to the console. | |

|  |  |  |  |
| --- | --- | --- | --- |
| Member Functions (public) | | | |
| Signature | |  | |
| Name | Type | Description & Parameters | |
| LinkedList | (Constructor) | Default constructor. | |
| ~LinkedList | (Destructor) | Default destructor. | |
| Clear | void | Deletes all of the nodes contained in this linked list. | |
| Delete | void | Deletes a node from the linked list.  LL\_Node<T>\* delNode - The node to look for in the linked. If the node is found, it will be deleted.  bool verbose - Whether or not to output status messages to the console. Defaults to true. | |
| Delete | void | Deletes a node with the given value from the linked list.  T dataToFind - The key value to look for in the linked list. If this value is found in a node, that node will be deleted. If the value appears multiple times in the list, the first node that contains the value will be deleted.  bool verbose - Whether or not to output status messages to the console. Defaults to true. | |
| Find | LL\_Node<T>\* | Finds the first instance of a key value in the linked list. Returns a pointer to the node containing the key value. If no match is found, returns nullptr.  T dataToFind - The key value to look for in the linked list.  bool verbose - Whether or not to output status messages to the console. Defaults to true.. | |
| GetHeadNode | LLNode<T>\* | Returns the pointer to the head node of this linked list. | |
| GetNodeCount | int | Returns the current count of the number of nodes in the linked list. | |
| GetTailNode | LLNode<T>\* | Returns the pointer to the tail node of this linked list. | |
| Insert | void | Creates a new node, and inserts it at the end the linked list.  T newData - The key value to be stored in the new node.  bool verbose - Whether or not to output status messages to the console. Defaults to true. | |
| IsEmpty | bool | Determines if the linked list is empty. Returns true if empty, returns false if not. | |
| Maximum | T | Finds and returns the largest value stored in the nodes contained in the linked list. | |
| Minimum | T | Finds and returns the smallest value stored in the nodes contained in the linked list. | |
| PrintList | void | Outputs the key values of each node to the console.  bool reverse - Determines the order in which to print the list. false prints from head to tail; true prints from tail to head (reverse). Defaults to false.  bool detailed - Detailed mode will display each node’s value, as well as the value and address of the current node’s adjacent nodes, if desired. true prints the extra data; false only prints the key values. Defaults to true. | |
| SwapValues | void | Swaps two key values between two different nodes.  LL\_Node<Type>\* node1 - A node that will be switching key values with node2.  LL\_Node<Type>\* node2 - A node that will be switching key values with node1. | |

## 

## <LL\_Iterator.h>

template <typename T> class LL\_Iterator

A custom iterator for LL\_Node objects stored in an LinkedList object. Uses its member functions GetPrevNode() and GetNextNode() to traverse the linked list. The class is templated as to allow for use with a wide variety of data types.

|  |  |  |
| --- | --- | --- |
| Attributes/Data Members (private) | | |
| Name | Type | Description |
| currentNode | LL\_Node<T>\* | Stores a pointer to the current node the iterator is referencing. |

|  |  |  |  |
| --- | --- | --- | --- |
| Member Functions (public) | | | |
| Signature | |  | |
| Name | Type | Description & Parameters | |
| LL\_Iterator | (Constructor) | Default constructor.  LL\_Node<T>\* startNode - The node at which this iterator should point to upon creation. | |
| ~LL\_Iterator | (Destructor) | Default destructor. | |
| GetCurrentNode | LL\_Node<T>\* | Returns a pointer to the current node the iterator is referencing. | |
| GetNodeData | T | Returns the key value stored in the current nodethe iterator is referencing. | |
| IterateBack | LL\_Node<T>\* | Attempts to move to the previous node in the linked list. If the iterator is at the beginning of the list, returns nullptr. Otherwise, updates the iterator’s currentNode pointer to point to the previous node in the list, and returns a pointer to the new node. | |
| IterateFwd | LL\_Node<T>\* | Attempts to move to the next node in the linked list. If the iterator is at the end of the list, returns nullptr. Otherwise, updates the iterator’s currentNode pointer to point to the next node in the list, and returns a pointer to the new node. | |
| SetCurrentNode | void | Sets the node this iterator is pointing to.  LL\_Node<T>\* targetNode - The node at which this iterator should point to upon creation. | |

## 

## <LL\_Node.h>

template <typename T> class LL\_Node

A custom-built node class for use in a doubly-linked list. The class is templated as to allow for use with a wide variety of data types.

|  |  |  |
| --- | --- | --- |
| Attributes/Data Members (private) | | |
| Name | Type | Description |
| nextNode | LL\_Node<T>\* | Stores a pointer to the node immediately following this node in the linked list. Initializes to nullptr. |
| nodeData | T | The data being held by this node. |
| prevNode | LL\_Node<T>\* | Stores a pointer to the node immediately preceding this node in the linked list. Initializes to nullptr. |

|  |  |  |  |
| --- | --- | --- | --- |
| Member Functions (public) | | | |
| Signature | |  | |
| Name | Type | Description & Parameters | |
| LL\_Node | (Constructor) | Default constructor. |  |
| ~LL\_Node | (Destructor) | Default destructor. | |
| ClearNextNode | void | Sets the pointer to the node immediately following this node in the linked list to nullptr. | |
| ClearPrevNode | void | Sets the pointer to the node immediately preceding this node in the linked list to nullptr. | |
| GetNextNode | LL\_Node<T>\* | Returns a pointer to the node immediately following this node in the linked list. | |
| GetNodeData | T | Returns the key value stored in this node. | |
| GetPrevNode | LL\_Node<T>\* | Returns a pointer to the node immediately preceding this node in the linked list. | |
| SetNextNode | void\* | Changes the pointer to the node immediately following this node in the linked list.  LL\_Node<T>\* nodePtr - A pointer to the node that will be designated as immediately following the current node in the linked list. | |
| SetNodeData | void\* | Changes the key value to be stored in this node.  T dataToStore - The key value to store in this node. | |
| SetPrevNode | void\* | Changes the pointer to the node immediately preceding this node in the linked list.  LL\_Node<T>\* nodePtr - A pointer to the node that will be designated as immediately preceding the current node in the linked list. | |