Joshua Scheck

October 13th, 2020

CSS 501 B

Design Document

Specification

The purpose of this program is to develop a vanilla linked list from scratch and a move-to-front (MTF) list as a subclass that employs the MTF strategy and will moved accessed elements to the front of the list. Furthermore, we will demonstrate and evaluate the sequential searching performance of the vanilla linked list and MTF list on the basis of searching for uniformly or normally distributed values.

The entry point for the program is LinkedListStats::main() where no user arguments are used. However, there are presets for number of values to be inputted into the either list (numValues) and number of accesses made to each list (numAccesses). Both LinkedList and MTFList objects will be instantiated as empty lists.

In order to populate both LinkedList and MTFList objects, the program will count down from numValues and sequentially add each of the values to both lists until the counted down value becomes negative. The addition of each of these integer values will require proper allocation of memory for each newly created Node in free store.

The program will subsequently generate random numbers in accordance to uniform or normal distribution and will call LinkedList::contains(int soughtVal) or MTFList::contains(int soughtVal) methods while also accounting for the number of sequential search traversals performed to ascertain the sought integer value. This will be looped through a number of times as dictated by the value of numAccesses.

Finally, an output will compare the average number of nodes traversed in LinkedList versus MTFList for both uniform distribution and normal distribution.

Input

No input will be provided by the user. Instead, preset values are assigned as follows:

- const int numValues. Represents the number of values allowed in LinkedList and MTFList. Values added to either list will count down from the integer value of numValues and will be sequentially added to the top of both lists until the count down value is negative.
- const int numAccesses. Number of times a LinkedList::contains(int soughtValue) and MTFList::contains(int soughtValue) methods are called where the sought value is an integer normally or uniformly distributed.

Output

The program result is outputted to cout. The program will first calculate the average number of nodes traversed for both LinkedList and MTFList using both uniformly and normally distributed values. Following these calculations, an output will be generated in the following format:

LinkedList - Average number of nodes traversed per access (uniform): linkedListAvgTravUni

MTFList - Average number of nodes traversed per access (uniform): MTFListAvgTravUni

LinkedList - Average number of nodes traversed per access (normal): linkedListAvgTravNorm

MTFList - Average number of nodes traversed per access (normal): MTFListAvgTravNorm

Outputted variables are calculated as follows:

- linkedListAvgTravUni is a double using the total number of traversals following the looping through of method calls to **LinkedList**::contains(int soughtValue) where the sought integer value is **uniformly** distributed.
- MTFListAvgTravUni is a double using the total number of traversals following the looping through of method calls to **MTFList**::contains(int soughtValue) where the sought integer value is **uniformly** distributed.
- linkedListAvgTravNorm is a double using the total number of traversals following the looping through of method calls to **LinkedList**::contains(int soughtValue) where the sought integer value is **normally** distributed.
- MTFListAvgTravNorm is a double using the total number of traversals following the looping through of method calls to **MTFList**::contains(int soughtValue) where the sought integer value is **normally** distributed.

Note: The number of loops of contains() calls are tied to the preset number of accesses allowed in the variable **numAccesses**.

Program Design

This program is composed an IList interface that is implemented by the LinkedList class. Additionally, MTFList is a subclass of LinkedList. Both LinkedList and MTFList are composed of Node struct and both are instantiated in LinkedListStats::main().

IList

The IList interface provides methods and attributes relevant to ADT lists that will be implement in subclasses (UML notations used for accessibility, variable, and return types):

- Constructor. Initializes a new empty list and sets traverseCount to zero.
- Deconstructor. Calls clear() method which will iteratively go from Node to Node and deallocate memory from each Node. This destroys the object and frees up all the memory previously allocated by this object in free store.
- + getCurrentSize(): int. Retrieves the current number of Nodes that exists in this list.
- + isEmpty(): bool. Assesses whether the number of Nodes in this list is zero. Returns true if the list is empty, false otherwise.
- + add(int newEntry): bool. Dynamically allocates memory for a new Node which will store the integer newEntry as its item. If successfully added, the current size of the list increases by one and return true, otherwise return false.

- + remove(int anEntry): bool. Removes the first occurrence of the integer anEntry in the list by sequentially looking for the integer in each Node starting from the head Node. If successfully removed, the current size of the list is decreased by one and returns true, otherwise return false.
- + clear(): void. Removes all entries from this list by sequentially deallocating memory from each Node existing in this list. The list will have no Nodes and a current count of zero.
- + contains(int anEntry): bool. Assesses whether the specified integer value is in at least one node in this list through sequential search. traverseCount will be increased by one for each node traversed until the specified integer value (anEntry) is found in a Node. If the integer value is found return true, otherwise return false.
- + getTraverseCount(): int. Retrieves the number of nodes traversed since the last time the count was reset.
- + resetTraverseCount(): void. Sets traverseCount to zero.
- - traverseCount: int. A counter for the number of nodes traversed.

LinkedList::Node

This struct represents a single node that composes a list ADT. This struct two data members (UML notations used for accessibility and variable types):

- + item: int. The integer value stored in this Node.
- + next: LinkedList::Node*. A reference to the memory address of the next Node in this LinkedList. If no next Node exists then this data member will be set to nullptr.

LinkedList

The LinkedList class implements the IList interface, thus the LinkedList class will have implementations of all of the methods and attributes present in the IList interface. In addition, the linked list class has additional methods and data members (UML notations used for accessibility, variable, and return types):

- Constructor. The constructor initializes an empty linked list by head to nullptr and setting traverseCount and currentSize to 0.
- Deconstructor. LinkedList::clear() will be called to sequentially deallocate memory from each Node existing in this list. This destroys the object and frees up all the memory previously allocated by this object in free store.
- -head: LinkedList::Node*. A reference to the memory address of the first Node in this LinkedList. If this linked list is empty then this data member will be set to nullptr.
- -currentSize: int. The current amount of Nodes existing in this LinkedList.

MTFList

The MTFList class is a subclass of LinkedList. Because of this, MTFList will inherit all the methods and attributes of LinkedList. In addition, it has a single overridden method (UML notations used for accessibility, variable, and return types):

- + contains(int anEntry): bool. Overrides LinkedList::contains(int anEntry) method in parent class (LinkedList). Similarly to LinkedList::contains(int anEntry), this method sequentially searches for the first occurrence of integer anEntry in a Node existing in this MTFList. If a Node is found with

the integer, it will be moved to the front of this MTFList and subsequently will return true. If integer anEntry is not found in any Node in this MTFList then this method will simply return false.

LinkedListStats

The LinkedListStats class includes the main() method and is the entry point for this program. Furthermore, main() evaluates the difference in sequential search performance of LinkedList and MTFList using uniformly distributed values and normally distributed values (UML notations used for accessibility and return types):

- + main(): int. Entry point for this program and instantiates LinkedList and MTFList. Populates both lists and evaluates sequential search performance.

Implementation Plan

To validate the implementation of LinkedList::Node, LinkedList methods, and MTFList methods will undergo a series of unit and integration tests. The overall implementation plan is:

1. LinkedList::Node. ADT lists are composed of Nodes. This struct will be tested along with the methods of LinkedList.
2. LinkedList. Many methods are available for LinkedList as highlighted in the program design. Testing will be performed in the following order:
    a. testLinkedListConstructor. This test method will make sure LinkedList::LinkedList() properly constructs a LinkedList object. The instantiated LinkedList object should be an empty list so we assert that LinkedList::empty() will return true. Furthermore, traverseCount and currentSize data members should have been set to zero in LinkedList constructor, thus we will also assert that LinkedList::getTraverseCount() and LinkedList::getCurrentSize() will return the integer zero.
    b. testAddNode. This test method will make verify the validity of LinkedList::add(int newEntry), LinkedList::getCurrentSize(), and LinkedList::isEmpty(). First, we will construct a LinkedList object, which we know to be empty. We will subsequently call LinkedList::add(int newEntry), where newEntry is an with integer value. Here, we assert that LinkedList::getCurrentSize() returns the integer one and LinkedList::empty() returns false. We can repeat adding integers a number of times to validate that the current size of the LinkedList will continue to proportionally increase with the amount of add operations performed.
    c. testRemoveNode. This test method will make verify the validity of LinkedList::remove(int anEntry). We will start with a constructed LinkedList. LinkedList::add(int newEntry) will be called with three times with different integer values. We will now assert that LinkedList::remove(int anEntry) will return false when the removed integer anEntry is not in the instantiated LinkedList. We also assert that LinkedList::getCurrentSize() returns the integer three to make sure that current size is accounted correctly. Next, we will assert that LinkedList::remove(int anEntry) will return true when the removed integer is in the instantiated LinkedList. Finally, it will be

asserted that LinkedList::getCurrentSize() returns the integer two. The remaining nodes can now be removed from the instantiated LinkedList.

d. testClearNodes. This test method will make verify the validity of LinkedList::clear(). We will start with a constructed LinkedList. LinkedList::add(int newEntry) will be called with three times with different integer values. We now will assert that LinkedList::getCurrentSize() returns the integer three and LinkedList::isEmpty() returns false. We will then call LinkedList::clear() which should delete all Nodes present in the instantiated LinkedList object. Lastly, we will assert that LinkedList::getCurrentSize() returns the integer zero and LinkedList::isEmpty() returns true.

e. testContainsAndTraverseCount. This test method will make verify the validity of LinkedList::contains(int anEntry), LinkedList::getTraverseCount(), and LinkedList::resetTraverseCount(). We will start with a constructed LinkedList and will call LinkedList::contains(int anEntry), where anEntry is any integer and assert the returned value will be false. We will also assert that LinkedList::getTraverseCount() returns 0. We expect both of these outcomes because no Nodes currently reside in the instantiated LinkedList. Next we add 3 nodes with different integer values and assert that LinkedList::contains(int anEntry) returns true when using the last added integer as the value for anEntry. Additionally, we will also assert that LinkedList::getTraverseCount() returns 1. We now will call LinkedList::resetTraverseCount() and also assert that LinkedList::getTraverseCount() returns 0. We can repeat this type of test for different integer values that we know exists in specific positions in the instantiated list.

3. MTFList. Only one method is overridden and will be tested:

   a. testModifiedContainsAndTraverseCount. This test method will make verify the validity of MTFList::contains(int anEntry). We will first construct a MTFList object, which we know to be empty as shown in the LinkedList tests. We will add 3 nodes with different integer values and assert that MTFList::contains(int anEntry) returns true when using the first added integer as the value for anEntry. Additionally, we will also assert that MTFList::getTraverseCount() returns 3. To test the MTF strategy, we will first reset the traversal count through MTFList::resetTraverseCount() and now will repeat our tests with the same integer value we previously searched for. We can now expect MTFList::contains(int anEntry) to return true and MTFList::getTraverseCount() returns 1.