Group 7

CSCI-313

HW1- Question 7: Vector & LinkList Class

In our project, we created a vector class using a template so that it would work for all datatypes. We declared an integer variable called size to track the number of elements in the current vector and a second integer variable, maxSize, that tells us the capacity of the vector. We also created a generic type array pointer that points to the first element in the array. Moving on to the public part of our vector class, we have the default constructor, the regular one parameter constructor, and the regular two parameter constructor that will be called when creating objects of the Vector class. For instance, if we created a vector using the following notation Vector<int> v(10,5), it will call the regular 2 parameter constructor and create a vector of integers of size ten with all elements initialized to 5.

In addition, we also have a copy constructor, a move constructor, a destructor, and an assignment operator. The copy constructor will pass in a reference to a vector object and copy all the information to our field vector. The move constructor will “steal” the information from the r-value vectorObj and move it into the l-value Array without copying. This is done by using the keyword “move” that turns a value into an r-value. The destructor deletes the array when we no longer need it. The assignment operator copies each element in the vectorObj that was passed into the function over to the field vector and returns it when done. Within the assignment operator we also have an if statement that checks for equality to prevent self-assignment. Moving on with our Vector class, we also have two getter functions that return Size (the length of our current array) and maxSize (the capacity of our array).

Next, we have a push_back function that takes in a generic type variable value. Within our push_back function, we have an if statement that checks if Size (the number of elements in array) is less than maxSize (array capacity). If the statement is true, meaning that there are open spots, then we are allowed to add values into the array. We assign that value to the current position of the array and Size increases by one, since an element is added. If the if statement was tested to be false, meaning that there's no spots left to input values into the array, then we continue with the "else" block of the code. First, we create a new array and update the size to be double the original one. Next, we use a for loop to traverse each index and copy every value from the old array to newArray. Then, we delete all the information in the old array, copy information from newArray to Array, insert the value at the desired position in Array, and increase Size by one because an element is added. Furthermore, our Vector class contains three more functions, pop_back, back, and printInfo. Our pop_back function deletes the last element in the vector, but can only do so when there are elements in the vector. This condition is checked with an if statement. The back function returns the last element in the vector and the printInfo function prints the elements in the vector by using a for loop.

Moreover, we created a function called randomFillArrayInt. This function adds random numbers between 1 and 100 to the vector, which we will be using in main for part a. We also created another function, randomString, in our Vector class that generates random strings of random sizes and returns the string result to functionFillArrayString (called here). Next, we have our randomString function that adds random strings (return back from randomString function) to a vector of strings, which we will be using in main for part b. Similarly, the randomFillWithMoveString function also fills vectors with strings, but by using the move semantic technique instead. This function is later used in our main function as part c. Next, we

have the fillVectorfunction, which is the vector from the C++ library. This function will fill the vector with random strings of length between 1 to 10 that is generated and returned from the randomString function. Finally, the last piece of code we have in our Vector class is a template class used for timing.

In our main function, we were suppose to use the Vector class that we created and do the following things: a) fill vector with random numbers, b) fill vector with random strings, c) fill vector with random strings using move semantics, and d) fill vector with random string using the vector in C++ library (part d wasn't ask for in the assignment, but we were just curious to know the timing of it compared to the Vector class we created). In our main function, we implemented all four different cases and timed it using the timer. After running the code successfully, we came to the result that filling a vector with random numbers was the fastest and filling a vector with random strings using move semantics was the slowest. After running the code many times, we found out that it is mainly faster to use the vector in our C++ library to fill in random strings compared to using the Vector class we created. However, there were a couple of runs where our Vector class is faster than the vector library.

For the LinkList part of the project, we created a LinkList class using a template so it will work for all datatypes. Within our LinkList class, we created a Node struct; similar to a class with the exception that member variables and methods are public. In our struct Node, we have a generic variable, Info, that represents the node with the value. We also have a Node pointer, Next, that points to the next value in the list. Moving on, we have the default constructor (create a default empty node), 1 parameter constructor (create a node storing input value), and a 2 parameter constructor (create a node storing input value and pointing to another node). We then

created setter and getter functions for each member variable in the struct Node. The setters will allow us to change and update information, while the getters will allow us to return things.

Moving on to the public part of the code, we created an Iterator class that has a friend class LinkedList, this allows the iterator to access the private variable in the LinkList class. We also have a private nodePointer variable, which is an iterator pointing to the current node. In our Iterator class, we have a default constructor that creates an iterator with a nodePointer pointing to null. We have another regular 1 parameter constructor that takes in newPointer as the parameter and assigns it to nodePointer. The last three things we have in our Iterator class were three operator overloads. First, we have an operator !=, which tells us that the field pointer doesn't equal the itr nodePointer. Second, we have operator*, which is the dereference operator and it's used to get the value. Finally, we have our operator++ and that's the post increment operator that functions just like i++.

Next, we declared an integer variable Counter to track the number of nodes in the list and another Node pointer variable head that points to the first node of the list. Within our LinkList, we have a default constructor, a getter function for Counter variable, and an isEmpty function that lets us know that the list is empty. Furthermore, we have an add function and it adds any generic data type value to the LinkList. Within our add function, we created a new node with a value. Then, we check if the head is empty. If the head is empty it means that the node that we are currently inserting will become the head. However, if the head is not empty, then we add the new node to the head. Since we added an element to the list, we increase the Counter by 1. We created a remove function that removes the last element from the LinkList. In our remove function, we have to first check if the LinkList is empty. If it is empty, then there's nothing to

remove from the list so we just simply throw a string stating that the list is empty for the user to know.

Let's suppose that the LinkList is not empty, then we assign head to node. Next, we assign the info of the last element to generic variable ret and delete the node. Then, assign head to the "new" last node. Since we removed an element from the list, we decrease the Counter by 1 and generic variable ret is returned. For example, let's say we have a link list 2 <- 7 <- 10 <- 5 <- head and we want to remove 5. First, we need to store the value 5 in the generic variable ret. Once value 5 is stored at a safe place, then we can change the head pointer to point to 10, meaning no pointer will be pointing to 5. In the next step, we delete the node and return the value 5. The begin function will return an iterator from the beginning of the list and likewise, the end function will return an iterator from the end of the list. The printList function will print the initial node until it finds NULL for the Next pointer that indicates that it's the end of the node chain. Lastly, the remaining functions in our LinkList class starting from fillList_int to the timing template do the exact same thing described in the Vector Class section, but instead using LinkList.

In our main function, we had to do the same thing explained earlier in the Vector Class section, but using LinkList this time through. After running the code many times, we came to the conclusion that filling a list with random numbers was always the fastest. Filling a list with random strings using move semantics appeared to be the slowest one for many runs, but there were a couple of times when filling random strings was the slowest. In general, to answer the question whether a vector or a list is faster by timing them via filling them with random numbers, filling them with random strings, and filling them with random strings using move semantics; we concluded that it was faster to use Vector based on our tests.