Jakob Delossantos

Professor Papachristos

CS 202

4/15/2020

Project 8 Documentation

For this week’s project, we were required to implement an array-based list and node-based list, each with their respective operations/functions. This project was very difficult, but now I can say I do understand quite a bit about what a list does and how it works. Below is my test driver output showcasing each function on each type of list, which I will explain with comments denoted by // or /\* in this paper:

-------------------------------------------------------Now testing the ArrayList container methods-------------------------------------------------------

Testing default-ctor... /\*The following three “Testing…” statements are the default, param, and copy c-tors. I used the operator<< overload to output their respective values.\*/

Default-ctor call (Array)

The array list holds the following DataType valid values: //Default holds nothing valid

There are 1 invalid values in the container

Testing param-ctor... //Made a list object of size five with values 1 for int and 0.5 for value

Parametrized-ctor call (Array)

The array list holds the following DataType valid values:

{1,0.5}, {1,0.5}, {1,0.5}, {1,0.5}, {1,0.5}, There are 0 invalid values in the container //All valid data

Testing copy-ctor...

Copy-ctor call (Array) /\*This call made a copy of the values of the param-ctor, with following output the same showing that it indeed work.\*/

The array list holds the following DataType valid values:

{1,0.5}, {1,0.5}, {1,0.5}, {1,0.5}, {1,0.5}, There are 0 invalid values in the container

Testing operator=... /\*Sets the param-ctor container equal to another object that I created of size 5 and values 2 for int, and 0 for double. Also outputs the container, now equal to the new values, 2 and 0.0\*/

Parametrized-ctor call (Array)

Operator= successful.

The array list holds the following DataType valid values:

{2,0}, {2,0}, {2,0}, {2,0}, {2,0}, There are 0 invalid values in the container

Testing front()... /\*Front and back act on the above container, and return a pointer if it has a front and back. It has 5 valid values of value 2.0 so there is indeed a front and a back. Mostly used for internal class operations.\*/

Front succeeded.

Testing back()...

Back succeeded.

Testing find()... /\*Finds element 2.0 in the same above container, and only prints target found when it is actually found, based on the class code. Mostly used for internal class operation. \*/

Target found.

Testing insertAfter... /\*This test first prints the list before inserting, resizes it, then is prints the list after inserting, showing that it worked. Inserts value 1.5 after the first element in the list. The destructors in the middle are from tempArrayLists used to resize the container.\*/

Parametrized-ctor call (Array)

The array list holds the following DataType valid values:

{2,0}, {2,0}, {2,0}, {2,0}, {2,0}, There are 0 invalid values in the container

Parametrized-ctor call (Array)

List is now resized to fit specified count: 6

Destructor call (Array)

Inserted after successfully.

The array list holds the following DataType valid values:

{2,0}, {1,0.5}, {2,0}, {2,0}, {2,0}, {2,0}, There are 0 invalid values in the container

Testing insertBefore... /\*This test uses insertBefore to insert before an element. It inserts into a container with 5 values of 2.0, a value of 1.5, as shown in the output after resizing.\*/

Parametrized-ctor call (Array)

The array list holds the following DataType valid values:

{2,0}, {2,0}, {2,0}, {2,0}, {2,0}, There are 0 invalid values in the container

Parametrized-ctor call (Array)

List is now resized to fit specified count: 6

Destructor call (Array)

Inserted before successfully.

The array list holds the following DataType valid values:

{1,0.5}, {2,0}, {2,0}, {2,0}, {2,0}, {2,0}, There are 0 invalid values in the container

Testing erase... //Erases one value out of a 5 size list with values 1.5, as shown by <<’s

The array list holds the following DataType valid values:

{1,0.5}, {1,0.5}, {1,0.5}, {1,0.5}, {1,0.5}, There are 0 invalid values in the container

Value successfully erased.

The array list holds the following DataType valid values: //Erases one value, so now there are only 4 valid values.

{1,0.5}, {1,0.5}, {1,0.5}, {1,0.5}, There are 1 invalid values in the container

Testing size... //Returns the m\_size of the above container, or only valid values.

The size is:4

Testing empty... //Checks if the container that just had a value erased is completely empty.

The list is not empty.

Testing clear... //Clears the container above of 4 valid values. First prints out the container before erasing, then again after erasing. Also checks if it is empty, which it is, and what is printed out.

The array list holds the following DataType valid values:

{1,0.5}, {1,0.5}, {1,0.5}, {1,0.5}, There are 1 invalid values in the container

List is now clear.

The array list holds the following DataType valid values:

There are 5 invalid values in the container

The list is empty.

-------------------------------------------------------Now testing the NodeList container methods-------------------------------------------------------

Testing default-ctor... /\* Tests the constructors for the NodeList, and prints them out\*/

Default-ctor call

The node list holds the following values:

Testing param-ctor...

Param-ctor call

The node list holds the following values:

{1,0.5} ,{1,0.5} ,{1,0.5} ,{1,0.5} ,{1,0.5} ,

Testing copy-ctor... //Creates new object based on data of param-ctor

Copy-ctor call

The node list holds the following values:

{1,0.5} ,{1,0.5} ,{1,0.5} ,{1,0.5} ,{1,0.5} ,

Testing operator=... /\* Sets the param-ctor object equal to a newly created object of size 5 and value 2.0, then prints it out to show it works with operator<<\*/

Param-ctor call

Operator= successful

The node list holds the following values:

{2,0} ,{2,0} ,{2,0} ,{2,0} ,{2,0} ,

Testing front()... //Front, back, and find perform similar to the arraylist implementations, but use different logic by use of pointers and how Node’s utilize the m\_next variables.

Front succeeded.

Testing back()...

Back succeeded.

Testing find()...

Target found as first element.

Testing insertAfter... //Prints before and after inserted new value after target, just like ArrayList

The node list holds the following values:

{2,0} ,{2,0} ,{2,0} ,{2,0} ,{2,0} ,

Inserted successfully.

The node list holds the following values:

{2,0} ,{1,0.5} ,{2,0} ,{2,0} ,{2,0} ,{2,0} ,

Testing insertBefore... //Prints before and after inserting new value before target, just like ArrayList

The node list holds the following values:

{1,0.5} ,{1,0.5} ,{1,0.5} ,{1,0.5} ,{1,0.5} ,

Inserted successfully.

The node list holds the following values:

{2,0} ,{1,0.5} ,{1,0.5} ,{1,0.5} ,{1,0.5} ,{1,0.5} ,

Testing erase... /\*This test erases the node that was just inserted with insertBefore into that container. It prints before, then after deleting.\*/

The node list holds the following values:

{2,0} ,{1,0.5} ,{1,0.5} ,{1,0.5} ,{1,0.5} ,{1,0.5} ,

Erased the first element.

The node list holds the following values:

{1,0.5} ,{1,0.5} ,{1,0.5} ,{1,0.5} ,{1,0.5} ,

Testing size... //This test prints out the size using size() of the node list above, after erasing the first value.

The size is:5

Testing empty... //Tests the list if it is empty, which it is not. Then prints what it holds to show it isn’t empty.

The list is not empty.

The node list holds the following values:

{1,0.5} ,{1,0.5} ,{1,0.5} ,{1,0.5} ,{1,0.5} ,

Testing clear... /\*Tests clear by showing the above container’s values after calling clear on it, then enforces this to check if it is empty, which it is.\*/

The list is now clear.

The node list holds the following values:

The list is empty

-------------------------------------------------------Now calling all object destructors-------------------------------------------------------

Destructor call (Node) //Calls all of the destructors for individual nodes and arrays with no memory leaks, so all of my dynamic memory management works successfully.

Destructor call (Node)

Destructor call (Node)

Destructor call (Node)

Destructor call (Array)

Destructor call (Array)

Destructor call (Array)

Destructor call (Array)

Destructor call (Array)

Destructor call (Array)

In conclusion, the dynamic memory management of this project was extremely difficult, and visualizing how elements are inserted, erased, etc. Valgrind/GDB came in handy and taught me more about dynamic memory and how it works.