PROG20799 Data Structures and Algorithms – C

Assignment 2

This work is to be done individually

Part 1: Coding

You are going to write a program that tests the speed of various operations on some sample collections and data structures. Most of the code has already been provided for you.

There are 4 collections provided.

GrowableArray and SortedArray are growable arrays – that is they contain data that is stored in a contiguous block of memory and they can grow to accommodate new elements being inserted into the array. In addition, SortedArray stores data that is sorted according to a comparison function. The comparison function can be provided at runtime as function pointer, although there is a "default" version that uses a built-in function to sort the data in ascending order.

List and SortedList are doubly linked lists – that is each List has a pointer to both the head (front) and tail (back) of the list, data is stored in the list as nodes (which can be anywhere in memory), and each node contains a pointer to the previous and next node in the list. SortedList stores data that is sorted in ascending order – you cannot specify a custom sort function.

You will be required to implement 3 additional features for the code.

Feature 1: Searching algorithms for data in a sorted collection

Implement the functions

int sortedListContains(SortedList *this, ListData data, unsigned *index) (found in main.c)

and

int sortedArrayContains(SortedArray *this, ArrayData data, unsigned *index) (found in main.c)

(Normally the implementation for these functions would be in the appropriate .c files, but putting them here makes submission of your assignment easier.)

For each function, you need to search the collection (Either a SortedArray or a SortedList to see if it contains the value specified by the parameter data. If the value is found in the collection and the parameter index is not equal to NULL, set the value referenced by index to the index of the element in the collection.

Your code should take advantage of the fact that the data stored in the collections is sorted by starting at the either the front or the back element of the collection when doing the sort. You should have some sort of test that you use to determine if you should start searching from the front or the back.

You can assume that data sorted in a SortedList and SortedArray will always be sorted in ascending order. You cannot do the same for List and GrowableArray. (You don't need to implement a fancier search algorithm like a binary search).

Feature 2: Testing harness – functions for repeatable experiments

Implement a function that allows you to perform an experiment. The experiment should allow you to time how long it takes to do some sort of operation. If you take a look at main.c you will see two example functions experimentInsertNElementsIntoArray() and experimentInsertNElementsIntoList().

In the event that your experiments use randomly generated data, your experiments should be repeatable. You can accomplish this in a number of ways – for example you can use a pregenerated list of random data that you stored in a text file and read from, or you could seed a pseudo-random number generator.

You never run an experiment just once – you always repeat the experiment multiple times. Sometimes each iteration of the experiment will be an exact duplication, and other times you will make subtle variations. For example, an experiment in which you insert 1000 random values into a SortedArray can be repeated with the exact same 1000 values being inserted in the exact same order, with the exact same 1000 values being inserted in a different order, or with a different set of 1000 values each time. For purposes of this exercise, an experiment should be repeated with the exact same values in the exact same order for each iteration.

If you are generating random values it can be help to cap the values to a range. For example if you are testing the insertion speed of a collection that ignores duplicates (like a Set), it can be helpful to constrain the random values so that duplicates are more likely to be generated. At the same time you want to ensure that you don't over constrain the range and make duplicates too likely.

For example, if you are testing insertion speed into a Set you might time how long it takes to insert 1000 values. If your experiments can 1000000 possible random values, there is a good chance you will never see a duplicate. If your experiments can only generate 2000 possible random values, there is a chance you will never see a duplicate. If your experiments can only generate 1000 possible random values, there is a guaranteed chance that you will see at least 1 duplicate. If your experiments can only generate 100 possible random values, you will see at least 900 duplicates. The range of possible values matters.

Don't forget to clean up any memory allocated by an experiment when the experiment is complete. You should avoid reusing collections between iterations of experiments – create a new Array/List for each iteration of the experiment.

Don't put all of your code in main(). The main functions should ideally be nothing more then a list of function calls that you can easily comment (and uncomment) in order to run a specific set of experiments.

Code the following experiments:

1. Experiments to test insertions to the back of a collection.
   a. Fill a GrowableArray with X random elements, where each element is inserted to the back of the Array
   b. Fill a List with X random elements, where each element is inserted to the back of the List
2. Experiments to test insertions to the front of a collection.
   a. Fill a GrowableArray with X random elements, where each element is inserted to the front of the Array
   b. Fill a List with X random elements, where each element is inserted to the front of the List
3. Experiments to test insertions to a random position in a collection
   a. Fill a GrowableArray with X random elements, where each element is inserted at a random index N=[0, array_size)
   b. Fill an List with X random elements, where each element is inserted at a random index N=[0, list_size)
4. Experiments to test insertions of elements into a sorted collection where it is unlikely that the inserted value is a duplicate.
   a. Fill a SortedArray with X random elements
   b. Fill a SortedList with X random elements
5. Experiments to test searching a collection for a value
   a. Search an Array that was previously filled with X random elements for Y specific values that you know are contained somewhere in the Array
   b. Search a List that was previously filled with X random elements for Y specific values that you know are contained somewhere in the List
   c. Search a SortedArray that was previously filled with X random elements for Y specific values that you know are contained somewhere in the SortedArray

    d. Search a SortedList that was previously filled with X random elements for Y specific values that you know are contained somewhere in the SortedList

For all of these experiments the value of X should be a variable – that is you can specify how many elements should be inserted at runtime. This is because in Feature 3. we are going to tune the value of X for each computer so that the experiments take about the same amount of time to run. On a really fast computer X might be in the millions, while a slower computer X might be in the ten thousands.

Once you specify a value of X for an experiment, you should use the same value of X for all of the sub-experiments. That is, if you are doing experiment 1., you might use X=10000. You should use that value of X for both 1a. and 1b. When you do experiment 2., you might be using a different value of X for 2a. and 2b. Same for values of Y used by experiment 5.

Each experiment should have the ability to be run for multiple iterations – that is you want to be able to repeat the experiment several times.

Feature 3: Timing Code – see how long experiments take to run

Use online resources to determine what code you need to add to time how long it takes for your code to run. Ensure that your timing is precise to sub-second accuracy (that is you must use timing code that lets you measure time differences < 1 second).

Now for each of the experiments above, determine a value of X (remember that each Experiment 1-5 Will have its own value of X) such that the longest sub-experiment takes about 3 seconds to complete 3 iterations. For example, experiment 1. Is testing the time it takes to insert X elements into an Array (1a.) and a List (1b.), where each element is inserted at the back. Pick a starting value for X and run the experiment. If both 1a. and 1b. complete in a time that is faster than 3 seconds, increase the value of X and run the experiment again. If either 1a. or 1b. completes in a time that is slower than 3 seconds, decrease X and run the experiment again. If at least one of 1a. or 1b. complete in a time that is about 3 seconds, and neither of 1a. or 1b. takes much longer than 3 seconds, you have found the correct value for X for that particular experiment.

For experiment 5 there are two values to tune – X and Y. X is the number of elements that should already be in the collection. Y is the number of elements you are going to search for. For example you might have a collection with 1000000 elements in it and a list of 300 elements (all of which you know are in the collection). You iterate through the list of 300 elements and get the index of each element in the collection. In that case, X is 1000000 and Y is 300.

Part 2: Run the experiments

Having determined a good value for X for each experiment, run the experiments and sub-experiments and record the results. For each sub-experiment, do 5 iterations and time how long it takes in total. Repeat each experiment twice more.

You should put your results in a table similar to the one found in appendix A.

Ensure that your final results table includes the results for experiments 1.-5., including all sub-experiments.

Part 3: Analyze the data

Examine the data.

1. Comparing the value of X for experiments 1, 2, and 3.
    a. For each of the three experiment, which collection (Array or List or both) was the main factor in your choice of X?
    b. Which of the three experiments has the largest value of X? Which of the three experiments has the smallest value of X?
2. For experiments 1, 2, and 3, answer the following questions:
    a. Compare the time it took for each sub-experiment. Was the time roughly the same, or was it different?
    b. Why do you think the time was the same (or different)? A good answer will include analysis of the source code provided in GrowableArray.c and LinkedList.c and/or diagrams to help me understand your answer.
3. Calculate rate values for experiments 1, 2, and 3:

   a. For each sub-experiment of the 3 experiments, calculate the rate of insertions. The rate of insertions should be calculated in elements inserted/second

   b. Compare the insertion rates for sub-experiments 1a and 2a. Does the insertion rate change based on where in the Array you are inserting the element? For each sub-experiment, do you think that the insertion rate will change if we change the value of X? Explain your answer.

   c. Compare the insertion rates for sub-experiments 1b and 2b. Does the insertion rate change based on where in the List you are inserting the element? For each sub-experiment, do you think that the insertion rate will change if we change the value of X? Explain your answer.

4. Comparing the value of X for experiments 3 and 4.

   a. For each of the two experiment, which collection (Array or List or both) was the main factor in your choice of X?

   b. Which of the two experiments has the largest value of X? Which of the two experiments has the smallest value of X?

   c. Do you think it is more efficient to insert a value into a random location of an unsorted array, or more efficient to insert a random value into a sorted array? Explain your answer, preferably with analysis of your experimental results and/or the provided source code.

   d. Do you think it is more efficient to insert a value into a random location of an unsorted list, or more efficient to insert a random value into a sorted list? Explain your answer, preferably with analysis of your experimental results and/or the provided source code.

5. Analyzing the results of experiment 5.

   a. Do you think it is more efficient to search for a value in an unsorted array, an unsorted list, a sorted array, or a sorted list? Explain your answer, preferably with analysis of your experimental results and/or the provided source code.

Part 4: Submission

Submit the following:

1. A .zip file named <slateusername>_<studentnumber>_assignment_2.zip that contains the following:

   a. A copy of your <slateusername>_<studentnumber>_main.c, which should include all of your experiments as well as your implementation of the two searching functions sortedListContains() and sortedArrayContains()

2. A document named <slateusername>_<studentnumber>_assignment_2.docx that contains your results and the answers to the questions. Please ensure you number your answers the same way they are numbered in this document.

   a. The results table must include the timing results (and average) for each sub-experiment, the values of X (and Y for experiment 5), number of iterations run, etc…

PROG20799 Assignment 2

Appendix A

Name:			Date:	
Experiment #:	1		Value of X:	
			Number of iterations:	
	Results			Average
1a	___ seconds	___ seconds	___ seconds	___ seconds
1b	___ seconds	___ seconds	___ seconds	___ seconds
Experiment #:	2		Value of X:	
			Number of iterations:	5
	Results			Average
2a	___ seconds	___ seconds	___ seconds	___ seconds
2b	___ seconds	___ seconds	___ seconds	___ seconds