Programming Assignment 2 (Applications of Binary Search, Linked List, and Dynamic Arrays)

Department of Computer Science, University of Wisconsin – Whitewater
Data Structure (CS 223)

1 Applications of Binary Search

Complete the functions in the BinarySearchApplications.cpp/BinarySearchApplications.java file.

Details of the functions to be implemented are provided in the following subsections.

1.1 Counting the number of occurrences of key in a sorted array

In certain applications, we are interested in counting the number of times *key* appears in a sorted array. The technique to solve such problems is to determine:

- minIndex: the minimum index where *key* appears
- maxIndex: the maximum index where *key* appears

The number of occurrences is given by $(maxIndex - minIndex + 1)$.

Hence, our task is to find both the minimum and maximum positions where a key occurs. We seek to solve this using **Binary Search**. To this end, complete the following functions:

- **int** minIndexBinarySearch(**int** array[ ], **int** arrayLength, int key): returns the minimum index where *key* appears. If *key* does not appear, then returns -1.
- **int** maxIndexBinarySearch(**int** array[ ], **int** arrayLength, int key): returns the maximum index where *key* appears. If *key* does not appear, then returns -1.
- **int** countNumberOfKeys(**int** array[ ], **int** arrayLength, int key): Returns 0 if *key* is not the in the array, else it returns the number of occurrences of *key*.

Caution: Your code should have complexity $O(\log n)$, where $n = arrayLength$. If your code ends up scanning the entire array (has a complexity $O(n)$), you will be awarded partial credit, even if you get the correct output.

Algorithm for finding the minimum index

- The main idea is to use binary search with a slight modification. Declare a variable called $minIndex$ along with $left$ and $right$. Initially, $minIndex = -1$.
- Now, when you find *key* at index *mid*, do not return mid, but set $minIndex = mid$, $right = mid - 1$, and continue.
- Finally, after the while loop expires return $minIndex$ (instead of -1).

Algorithm for finding the maximum index

Use a variable $maxIndex$ (instead of $minIndex$). Algorithm remains the same as above, just that when you find key at mid, we set $maxIndex = mid$ and $left = mid + 1$. Finally, return $maxIndex$.

Algorithm to count number of occurrences of key

Use the above two algorithms to get $minIndex$ and $maxIndex$. Then, use the formula to count and return the number of occurrences.

1.2 The Predecessor Problem

Given a set of numbers, the predecessor of a number x is the highest number in the set that is less than or equal to x. Thus, if I have the set $\{6, 9, 10, 13, 22, 31, 34, 88\}$, then the predecessor of 31 is 31 itself, whereas the predecessor of 30 is 22, and the predecessor of 5 is not defined.

The predecessor problem has remarkable applications in *network routing*, where we send information from one computer to another, making email and other uses of the internet possible. Another application is *nearest-neighbor search*, akin to locating restaurants on Google Maps, where it returns the closest match to a cuisine of your choice.

Our task is to find predecessor of a number in an array using a **Binary Search approach**. To this end, complete the following function:

- **int** findPredecessor(**int** A[ ], **int** arrayLen, **int** key): returns a position in the array A where the predecessor of key lies. Needless to say that the array A is sorted in ascending order. If the predecessor of key is not defined, return -1.

Caution: You MUST use a binary search approach. Thus, the complexity should be $O(\log n)$. If your code ends up scanning the entire array (has a complexity $O(n)$), you will be awarded partial credit, even if your code is correct.

Algorithm for finding the predecessor index

- The main idea is to use binary search with a slight modification. Declare a variable called $predIndex$ along with $left$ and $right$. Initially, $predIndex = -1$.
- Now, when $A[mid] < key$, then mid is a better estimate of your predecessor index; so, set $pred = mid$, $left = mid + 1$, and continue. Rest remains unchanged withing the while loop.
- Finally, after the while loop expires return $predIndex$ (instead of -1).

2 Linked List

Your task is to complete the following functions in the `LinkedList.h/LinkedList.java` file:

- **void** insertAfter(**ListNode** *argNode, **int** value) for C++, or
  void insertAfter(**ListNode** argNode, **int** value) for Java:

Function inserts a node with value *value* after the node *argNode*. You may assume that *argNode* is not null.

- **void** deleteAfter(**ListNode** *argNode) for C++, or
  void deleteAfter(**ListNode** argNode):

  Function deletes the node after *argNode*. You may assume that *argNode* is not null.

- **void** selectionSort():

  Function that sorts the linked list using selection sort method.

- **bool** removeDuplicatesSorted() for C++, or
  boolean removeDuplicatesSorted():

  Function that checks whether or not the linked list is sorted in ascending order. If it is not sorted, then the function returns *false*. Otherwise, the function removes the duplicate occurrences of each number from the list (i.e., only one occurrence of each number remains). Then the function returns *true*.

- **void** pushOddIndexesToTheBack():

  Function that pushes all the odd indexes to the back of the linked list, starting with index 1, then 3, then 5, and so on.

Caution: For the last three methods above, you CANNOT use any other data structure (linked list or arrays) for storage. You CAN ONLY use variables. Also on a linked list of length n:

- the first two functions must have a complexity of $O(1)$,
- selection sort must achieve a complexity of $O(n^2)$
- removing the duplicates method must achieve a complexity of $O(n)$
- pushing the odd indexes to the back must achieve a complexity of $O(n)$

2.1 insertAfter

Pseudo-code

- create newNode
- newNodes's next points to argNode's next
- argNode's next points to newNode
- if argNode is tail, then newNode becomes tail
- increment size

2.2 deleteAfter

Pseudo-code

- if argNode is tail, then there is nothing to delete
- else if argNode's next is tail, then
  - get rid of all references to tail (Java) or invoke delete (C++)
  - argNode becomes tail
  - decrement size
- else
  - use a placeholder variable to point argNode's next
  - argNode's next points to placeholder's next
  - get rid of all references from placeholder (Java) or invoke delete (C++)
  - decrement size

2.3 Selection Sort

Pseudo-code

- You will use virtually the same idea as in a selection sort algorithm on arrays
- Since random accesses are not possible, use three placeholder variables – $iNode$ to point to the ListNode on which the outer loop i is iterating, $minNode$ to point to the minimum valued node in the portion of the linked list starting from i all the way to the end, and $jNode$ to point to the ListNode on which the outer loop j is iterating
- Initially $iNode$ points to *head*
- Within the outer-loop, $minNode$ is initially $iNode$ and $jNode$ is the node after $iNode$
- Within the inner-loop, you compare the values of $jNode$ and $minNode$ and you set $minNode$ to $jNode$, if the latter has a smaller value. Set $jNode$ to its next node.
- Once the inner loop terminates, you swap the values of $iNode$ and $minNode$ (if needed). Then, set $iNode$ to its next node.

2.4 Removing Duplicates in Ascending Sorted List

Pseudo-code

- To check if the linked list is ascending sorted, use a loop with a place holder variable (similar to getNodeAt function). At any point, if the value of the placeholder node is larger than the value of the next node, then you return false, else move placeholder to its next node.
- Once you are done with the above for-loop (and did not return false), it is guaranteed that the linked list is sorted.
- Once again scan through the linked list using a loop and a placeholder. If the value of the placeholder equals the value of the next node[a], then you can delete the duplicate value in the next node by calling deleteAfter on the placeholder, else move placeholder to its next node.

[a] If the next node is null, then stop the process!

2.5 Pushing Odd Indexes to the End

Example 1

If the input list is $[5 \to 8 \to 16 \to 21 \to 32 \to 50 \to 66]$, then after executing this function the list will become $[5 \to 16 \to 32 \to 66 \to 8 \to 21 \to 50]$

Example 2

If the input list is $[-12 \to 5 \to 16 \to 32 \to 66 \to 8 \to 21 \to 50]$, then after executing this function the list will become $[-12 \to 16 \to 66 \to 21 \to 5 \to 32 \to 8 \to 50]$

Pseudo-code

- Scan through the linked list using a loop and a placeholder, but this time only loop over the even indexes 0, 2, 4, 6, and so on.
- Within the loop, insert the value in the node immediately after the placeholder at the end of the linked list and delete the node after the placeholder. Move placeholder to its next node.

3 Dynamic Array

Use the notes posted on Dynamic Array to implement the `insertAtEnd` and `deleteLast` methods in the `DynamicArray.h`/ `DynamicArray.java` files.

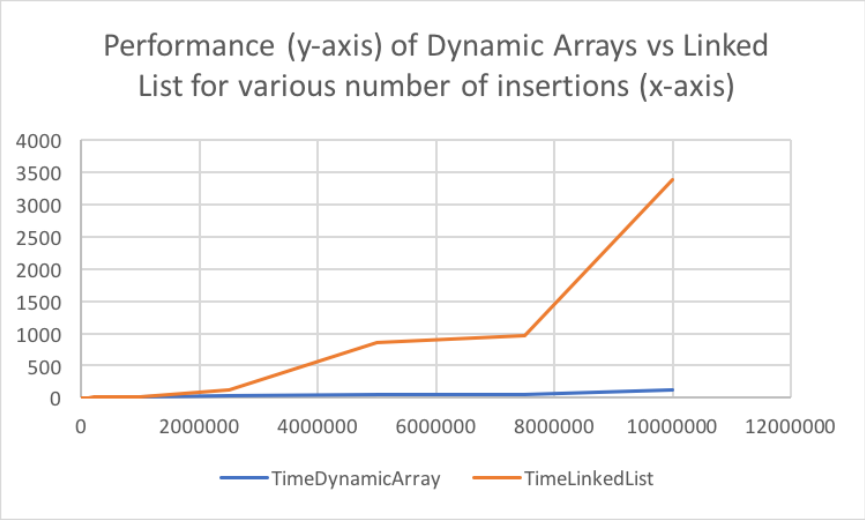
4 Correctness

Use the `TestCorrectness.cpp`/ `TestCorrectness.java` files to test your code. The expected output is provided in `ExpectedOutput` file. You can use `www.diffchecker.com` to tally the output.

5 Comparative Analysis

Recall that for insertion, dynamic arrays have amortized $O(1)$ complexity, whereas linked lists have worst case $O(1)$ complexity. However, as shown in Fig. 1 below, dynamic arrays outperform linked lists. This is because of the way arrays and linked lists are stored in memory. Remember that an array occupies a contiguous block in memory, whereas a linked list is scattered all through out. This makes arrays more cache friendly, as we bring in a larger chunk of the array into the cache from the RAM in one shot as opposed to linked lists. Hence, using arrays lead to fewer page faults, causing fewer accesses to the RAM, making them practically faster (or at least comparable).

Figure 1: Performance of dynamic arrays vs linked lists for various number of insertions.



If you run the `TestTime.cpp/TestTime.java` files, you will get an output similar to the ones in the next page; the numbers kind of show you that they are comparable in all aspects. Moreover, you can run all standard array based algorithms (such as Binary Search) on Dynamic Arrays, making them incredibly useful.

C++ Time Output

```
****************** Time Test Dynamic Array vs LinkedList ******************

Round 0 Completed
Round 1 Completed
Round 2 Completed
Round 3 Completed
Round 4 Completed
Round 5 Completed
Round 6 Completed
Round 7 Completed
Round 8 Completed
Round 9 Completed
Round 10 Completed
Round 11 Completed
Round 12 Completed
Round 13 Completed
Round 14 Completed

Total number of insertions, scans, and deletions (each) = 167286268
Total number of random accesses = 15000

Insertion time of dynamic array = 2256.48ms
Insertion time of linked list = 9897.41ms

Scan time of dynamic array = 688.17ms
Scan time of linked list = 568.23ms

Deletion time of dynamic array = 1453.05ms
Deletion time of linked list = 12540.89ms

Random access time of dynamic array = 0.66ms
Random access time of linked list = 237216.89ms
```

Java Time Output

```
****************** Time Test Dynamic Array vs LinkedList ******************

Round 0 Completed
Round 1 Completed
Round 2 Completed
Round 3 Completed
Round 4 Completed
Round 5 Completed
Round 6 Completed
Round 7 Completed
Round 8 Completed
Round 9 Completed
Round 10 Completed
Round 11 Completed
Round 12 Completed
Round 13 Completed
Round 14 Completed

Total number of insertions, scans, and deletions (each) = 167286268
Total number of random accesses = 15000

Insertion time of dynamic array = 785.00ms
Insertion time of linked list = 13212.00ms

Scan time of dynamic array = 14.00ms
Scan time of linked list = 404.00ms

Deletion time of dynamic array = 8201.00ms
Deletion time of linked list = 532.00ms

Random access time of dynamic array = 1.00ms
Random access time of linked list = 201374.00ms
```