AP Computer Science | Project 24 - GrowthList | Name:
Dr. Paul L. Bailey | Tuesday, November 19, 2019

This project is ON OR BEFORE due Tuesday, November 26, at 11:59 PM. Make sure to put your name as a comment on the top line of each source file. Zip all `.java` files and email to `paul.bailey@basised.com`. Your name should be part of your `.zip` file.

DO NOT `import java.util.Arrays`.

DO NOT import anything from the standard library into the `GrowthList` class.

Program 1. Copy your `BoundList` class from Project 23 into a new class called `GrowthList`. Rename all of the constructors. Compile and test.

Program 2 (Sequential Search)**.** A *sequential search* simply starts at the beginning of a list and iterates through it until the desired object is found, or until the end of the list is reached.

Implement a method `public int sequentialSearch(int v)` which performs a sequential search for value `v`. If `v` exists in the list, return the index of the first position where it resides; otherwise return -1.

Selection sort and *insertion sort* are "sort-in-place" algorithms. This means that a new array is not created; the algorithms manipulate the given array.

Program 3 (Selection Sort)**.** A *selection sort* finds the lowest member in a list using a transversal, switches it with the first (index 0) member, then finds the next lowest member in the rest of the list and switches it with the second (index 1) member, and so forth until the list is sorted.

Implement a method `public void selectionSort()` which implements this algorithm.

Program 4 (Insertion Sort)**.** An *insertion sort* keeps the front of the list sorted by inserting the next object into the front. More precisely, suppose the first $k-1$ members are sorted. Get the k^{st} value v. Starting at the $(k-1)^{\text{th}}$ slot and moving backwards, compare v to the value in the current slot; if the value is greater than v, move it into the current slot and continue; otherwise, put v in the current slot and quit.

Implement a method `public void insertionSort()` which implements this algorithm.

Program 5 (Binary Search)**.** A *binary search* on a sorted list computes the middle element, and tests it against the desired value. If the desired value equals the middle value, we are done. If the desired value is less than the middle value, we repeat with the first half of the list. If the desired value is greater than the middle value, we repeat with the second half of the list.

Implement a method `public int binarySearch(int v)` which performs a binary search for value `v`. If `v` exists in the list, return the index of the first position where it resides; otherwise return -1.

Program 6 (Growth)**.** Modify the class so that the physical size of the array can change. If an insert is attempted which is too large for the physical array, instead of returning an error condition, allocate a larger array, copy the old array into the new one, and set the classes array variable to point at the new array.

- Create a method `private static int[] grow(int[] x, int k)` which creates a new array whose length is `x.length + k`, copies `x` into this array, and returns the new array.
- Maintain a private instance variable which is the "growth rate":

  ```
  private int r = 10;
  ```

  The new array should be created with this many more positions, all else being equal.
- Add accessor `public int getRate()` and mutator `public void setRate(int r)` methods for the rate. Make sure, when setting the rate, it is between 1 and 1000.

You do not have to submit the next method with the project, but you should be aware that this topic is listed on the official AP syllabus. Implement merge sort using recursion. I suggest that you do not try to implement merge sort as a sort in place algorithm. Rather, the divide step should be "in place", but the recombination step should return the merged arrays as new arrays.

The following merge sort description copied from

`http://www.personal.kent.edu/ rmuhamma/Algorithms/MyAlgorithms/Sorting/mergeSort.htm`.

Program 7 (Merge Sort)**.** The *merge sort* is based on the divide-and-conquer paradigm. Since we are dealing with subproblems, we state each subproblem as sorting a subarray A[p .. r].
Initially, `p = 1` and `r = n`, but these values change as we recurse through subproblems.

To sort A[p .. r]:

(1) Divide Step
If a given array A has zero or one element, simply return; it is already sorted. Otherwise, split `A[p .. r]` into two subarrays `A[p .. q]` and `A[q + 1 .. r]`, each containing about half of the elements of `A[p .. r]`. That is, `q` is the halfway point of `A[p .. r]`.

(2) Conquer Step
Conquer by recursively sorting the two subarrays `A[p .. q]` and `A[q + 1 .. r]`.

(3) Combine Step
Combine the elements back in `A[p .. r]` by merging the two sorted subarrays `A[p .. q]` and `A[q + 1 .. r]` into a sorted sequence.

Note that the recursion bottoms out when the subarray has just one element, so that it is trivially sorted.

Implement a method `public void mergeSort()` which implements this algorithm.

If you get this far, or if you finish the rest but don't feel up to programming the merge sort, then try the following. Make sure that whatever you submit compiles.

Program 8 (List Interface)**.** Implement as many methods from the `List` interface as you can. If you can implement them all, then modify the class to state that it implements the interface, and test this using standard Java library methods.