AP Computer Science | Project 23 - Boundlist | Name:
Dr. Paul L. Bailey | Monday, November 6, 2017

Project 23 is due at 11:59 PM on Wednesday, November 20, 2019. 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`. Name the file `LastnameFirstname_P23_Boundlist.zip`, where of course you should use your own name.

This sequence of projects will be due in three parts. Project 23 is `BoundList` (lists which grow logically but not physically), Project 24 is `GrowthList` (lists which grow physically using arrays), and a later project is `ChainList` (lists which grow automatically using linked list data structures).

DO NOT `import java.util.Arrays`.

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

The official AP Computer Science Course Description includes the following sections.

IV. Standard Data Structures

Data structures are used to represent information within a program. Abstraction is an important theme in the development and application of data structures.

(A) Simple data types (int, boolean, double)

(B) Classes

(C) Lists

(D) Arrays

V. Standard Algorithms

Standard algorithms serve as examples of good solutions to standard problems. Many are intertwined with standard data structures. These algorithms provide examples for analysis of program efficiency.

(A) Operations on data structures previously listed

- **(1)** Traversals
- **(2)** Insertions
- **(3)** Deletions

(B) Searching

- **(1)** Sequential
- **(2)** Binary

(C) Sorting

- **(1)** Selection
- **(2)** Insertion
- **(3)** Mergesort

Hopefully, we already have a good understanding of most of section **(IV)**; in this sequence of projects, we code all of the algorithms in section **(V)**. To focus on the algorithms, we do this in the case of an array of positive integers with a fixed length, but with variable size. The length of the list is the length of the underlying array, but the size of the list is the number of occupied slots. We implement all of the algorithms as instance methods in a new class called `BoundList`.

Be aware of this: *none of the methods in these projects methods print anything*, other than in the testing methods of the `Program` class. Low lying methods do not communicate with the (human) user; they only communicate with the programs which invoke them, by returning values.

Preparation

Program 0 (Bound List)**.** Create a new folder `BoundList`. In it, create new source files `Program.java` and `BoundList.java`. In the `BoundList.java` file, type this code.

```
import java.util.Random;

public class BoundList
{
    private int[] a;        // Fixed Length Array
    private int n;          // Number of Occupied Slots

    public BoundList()
    {
        a = new int[1000];
        n = 0;
    }

    public BoundList(int length)
    {
        a = new int[length];
        n = 0;
    }

    public static BoundList generate(int length, int size, int max)
    {
        BoundList bl = new BoundList(length);
        bl.n = size;
        Random random = new Random();
        for (int i = 0; i < size; i++)
        {
            bl.a[i] = random.nextInt(max) + 1;
        }
    }
}
```

In the `Program.java` file, create a `Program` class to contain test cases.

```
class Program
{
    public static void main(String[] args)
    {
        test0();
    }

    public static void test0()
    {
        BoundList bl = BoundList.generate(1000, 100, 100);
    }
}
```

Compile and run `Program`.

Program 1 (Properties)**.** Create the following instance methods in the `BoundList` class. After each method is coded, add code to a method `Program.test1` to test the new method, and then compile and run `Program`.

(a) `public int length()` - returns the length of the private array.

(b) `public int size()` - returns the number of occupied slots.

(c) `public int get(int i)` - returns `a[i]`, only if this slot is occupied; otherwise returns 0.

(d) `public boolean set(int i, int v)` - sets the $\texttt{i}^{\text{th}}$ slot to `v`, only if this slot is occupied. The value `v` must be positive. Returns `true` if successful, otherwise returns `false`.

(e) `public void clear()` - sets the number of occupied slots to zero.

Program 2 (Copy Constructors)**.** A *copy constructor* instantiates a new object by copying an existing one.

(a) Implement a constructor `public BoundList(int[] x)` which creates a new instance of `BoundList` which contains the values of the array; the constructor creates new array, sets the size to the length, and copies the contents of `x` into the new array. Only the positive values are copied.

(b) Implement a constructor `public BoundList(BoundList bl)` which creates a new instance of `BoundList` which is identical to `bl`; the constructor creates new array and copies the size and contents of `bl` into the new array.

(c) Implement a constructor `public BoundList(BoundList bl, int length)` which behaves like **(b)** except that the length of the new array is `length`. This will truncate `bl`'s data if `length < bl.size`.

(d) Implement a constructor `public BoundList(BoundList bl, int length, int size)` which behaves like **(c)** except that a maximum of `size` elements are copied.

After each method is coded, add code to a method `Program.test2` to test the new method, and then compile and run `Program`.

Operations

Program 3 (Traversals)**.** A *traversal* travels through a sequence of objects, looking at each one. A traversal for the `BoundList` class would traverse up to size, not up to length.

Create the following instance methods in `BoundList` which are traversals. Use the `generate` method to help create test cases for each new method in a `Program.test3` method.

(a) `public void print()`: print the index and value of each member of the list (up to `size`).

(b) `public int minimum()`: return the index of the minimum value in the list.

(c) `public int maximum()`: return the index of the maximum value in the list.

Program 4 (Additions)**.** Create an instance method `public boolean add(int v)` which, if size is less than length, adds the value `v` into the $\texttt{n}^{\text{th}}$ slot, where n is the number of occupied slots, thus increasing the number of occupied slots by 1. Only positive values may be added. Return `true` if this is successful; if there is no room left for an addition, or if v is not positive, return `false`. Write, compile, and run appropriate testing code in a `Program.test4` method.

Program 5 (Insertions)**.** Create an instance method `public boolean insert(int i, int v)` which, if size is less than length, inserts the value `v` into the $\texttt{i}^{\text{th}}$ slot, pushing all other values forward, thus increasing the number of occupied slots by 1. Only positive values may be inserted. Return `true` if this is successful; if the index is out of bounds, or if there is no room left for an insertion, or if v is not positive, return `false`. Write, compile, and run appropriate testing code in the `Program.test4` method.

Program 6 (Deletions)**.** Create an instance method `public boolean delete(int i)` which removes the value in the $\texttt{i}^{\text{th}}$ slot, moving all other values backward, thus decreasing the number of occupied slots by 1. Return `true` if this is successful; if the index is out of bounds, return `false`. Write, compile, and run appropriate testing code in the `Program.test4` method.

If you have gotten this far, go on to the next project; the next two sections are part of it.

Sorting

Program 7 (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 8 (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 members are sorted. Get the $(k+1)^{\text{st}}$ value v. Find its correct position j in the front of the list (it is possible $j = k+1$). Move everything from the j^{th} slot to the $(k+1)^{\text{st}}$ slot forward one slot, and set the j^{th} slot equal to v.

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

Growing

Program 9 (Growth)**.** Copy your `BoundList` class into a new class, and call it `GrowthList`. Modify all necessary parts of the software to allow the physical size of the list to grow upon insertion of a new element. More details on this next week.