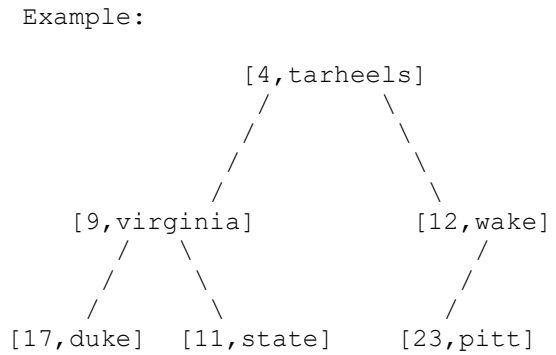
UNC-CH COMP 410

Priority Queue / Binary Heap Implementation

Overview

You are going to build in Java a Priority Queue (PrQUE) ADT in Java. It will be done by building a **minimum binary heap** (BHEAP) with the numerically smallest priority at the root of the tree.

Instead of a single integer as heap element (as we have done in class) your heap will allow manipulation of more information. We represent this as a integer (for priority queue position) and a string (associated information). As elements move around in the heap, keep the associated information associated. We will do this with a element object that contains both (see Interfaces below).



Now that we have established that we are using a heap to produce a priority queue, don't worry about the various namings we use in the code (which may mix up a bit the technical difference between PrQUE and BHEAP).

What to Hand-in

Hand in your code as per the instructions under Assignment 3 in Sakai.

Interfaces

For uniformity, to aid grading, you must write your program with these specific structures. Note that this time there are interfaces for both **Heap** and for **EntryPair**

Interface Heap_Interface

```
package MinBinHeap_A3;

public interface Heap_Interface {
  /*
    Interface: The BHEAP will provide this collection of operations:

    insert
      in: an EntryPair object, containing the priority and string,
          assume no duplicate priorities will be inserted
      return: void
    delMin
      in: nothing
      return: void
    getMin
      in: nothing
      return: an element (an EntryPair object)
    size
      in: nothing
      return: integer 0 or greater
    build
      in: array of elements that need to be in the heap
      return: void
      (assume for a build that the bheap will start empty)
  */

  // ADT operations
  void insert(EntryPair entry);
  void delMin();
  EntryPair getMin();
  int size();
  void build(EntryPair [] entries);
  EntryPair[] getHeap();
}
```

Interface EntryPair_Interface

```
package MinBinHeap_A3;

public interface EntryPair_Interface {
  String getValue();
  int getPriority();
}
```

Classes

Class MinBinHeap

```
package MinBinHeap_A3;

public class MinBinHeap implements Heap_Interface {
```

```
  private EntryPair[] array; //load this array
  private int size;
  private static final int arraySize = 10000; //Everything in the array will initially
                                              //be null. This is ok! Just build out
                                              //from array[1]

  public MinBinHeap() {
    this.array = new EntryPair[arraySize];
    array[0] = new EntryPair(null, -100000); //0th will be unused for simplicity
                                             //of child/parent computations...
                                             //the book/animation page both do this.
  }

  //Please do not remove or modify this method! Used to test your entire Heap.
  @Override
  public EntryPair[] getHeap() {
    return this.array;
  }
}
```

Class EntryPair

```
package MinBinHeap_A3;

public class EntryPair implements EntryPair_Interface {
  public String value;
  public int priority;

  public EntryPair(String aValue, int aPriority) {
    value = aValue;
    priority = aPriority;
  }

  public String getValue() { return value; }
  public int getPriority() { return priority; }
}
```

Class MinBinHeap_Playground

```
package MinBinHeap_A3;

public class MinBinHeap_Playground {
  public static void main(String[] args){
    //Add more tests as methods and call them here!!
    TestBuild();
  }

  public static void TestBuild(){
    // constructs a new minbinheap, constructs an array of EntryPair,
    // passes it into build function. Then print collection and heap.
    MinBinHeap mbh= new MinBinHeap();
    EntryPair[] collection= new EntryPair[8];
    collection[0]=new EntryPair("i",3);
    collection[1]=new EntryPair("b",5);
    collection[2]=new EntryPair("c",1);
    collection[3]=new EntryPair("d",0);
    collection[4]=new EntryPair("e",46);
    collection[5]=new EntryPair("f",5);
    collection[6]=new EntryPair("g",6);
    collection[7]=new EntryPair("h",17);
    mbh.build(collection);
    printHeapCollection(collection);
    printHeap(mbh.getHeap(), mbh.size());
```

```
  }

  public static void printHeapCollection(EntryPair[] e) {
    //this will print the entirety of an array of entry pairs you will pass
    //to your build function.
    System.out.println("Printing Collection to pass in to build function:");
    for(int i=0;i < e.length;i++){
      System.out.print("("+e[i].value+","+e[i].priority+")\t");
    }
    System.out.print("\n");
  }

  public static void printHeap(EntryPair[] e,int len) {
    //pass in mbh.getHeap(),mbh.size()... this method skips over unused 0th index....
    System.out.println("Printing Heap");
    for(int i=1;i < len+1;i++){
      System.out.print("("+e[i].value+","+e[i].priority+")\t");
    }
    System.out.print("\n");
  }
}
```

Class RunTests

```
package MinBinHeap_A3;
import gradingTools.comp410s19.assignment3.testcases.Assignment3Suite;

public class RunTests {
  public static void main(String[] args){ //runs Assignment 3 oracle tests
    Assignment3Suite.main(args);
  }
}
```

Notes on Operations

insert

Ordering is done based on the integer priorities in the elements that are inserted. Ignore the string for ordering. In the test data we use, the integer priorities in the elements will be **unique** -- there will be no duplicate priorities.

getMin

This operation returns an element (the entire object, priority and data value). It does NOT alter the heap. The heap has the same elements in the same arrangement after as it did before. If getMin is done on an empty heap, return null.

delMin

This operation removes the root element (the entire object) from the heap. The size of the heap goes down by 1 (if the heap was not already empty). If delMin is done on an empty heap, treat it as a no-op... i.e., do nothing other than return void.

build

The build operation should start with an empty heap. It receives an array of element objects as input. The effect is to produce a valid heap that contains exactly those input elements. This means when done the heap will have both a proper structure and will exhibit heap order.

Build is not the same as doing an insert for every element in the array. It is the special O(N) operation from the text (and shown in class) that starts with placing all elements into the heap array with no regard to heap order. You then go to the parent of the last node, and bubble down as needed. Go to the next node back towards the root, bubble down as needed. Repeat until you have done the root.

size

The size operation simply returns the count of how many elements are in the heap. It should be 0 if the heap is empty, and always return a 0 or greater.

Testing and Output

For testing, use the animation page given in class to get test cases and examples of correct behavior. Make sure your code is doing the same.

Their is no specific output or format required for this assignment. We assume you will use the "print" capability provided (and potentially expand on it) to test your code beyond the scope of the oracle.

Furthermore, please note that just because the Oracle gives you 100, you are not guaranteed any specific grade on the assignment. We do try to put as many things as we can in there edge-case wise, but testing your programs is your responsibility. The main goal of the Oracle is to provide *some direction*. However, it is not the end-all be-all determiner of correctness. Please do not use this tool as a crutch.

Zip/Submit Instructions

Follow the instructions given in the assignment on Sakai.