

Priority Queue; Heapsort; Huffman Code

Goals

In the first part of this lab we will design an efficient implementation of the *Priority queue*, and use it to implement the *heapsort* algorithm.

The second part focuses on the *Huffman code* for data compression.

11.1 Priority Queue: Heap and Heapsort

Our first goal in this section is to design a *priority queue* using the heap algorithm. We then use this to implement the *heapsort* algorithm and add it to a collection of algorithms to be evaluated.

Recall the properties of the *Heap* from yesterday's lecture:

- A *heap* is a complete binary tree. It means that every level is filled, except for the last level. The last level is filled from the left.

  So a heap with five nodes will have the following shape:

```
          node-1
          /    \
      node-2  node-3
      /    \
  node-4  node-5
```

  The nodes are labeled by levels from left to right, starting at 1.

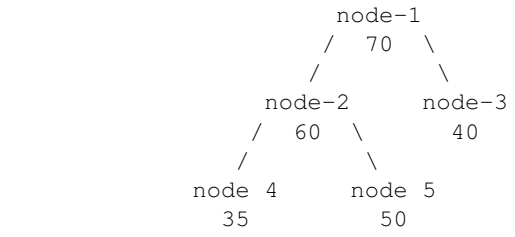
- The *value* in each node is greater-than or equal-to (for some comparison definition) the values in both of its children. So the item with the highest value (highest priority) is at the root of the tree.
- The label of the parent of node `i` is `i/2`
- The label of the left-child of node `i` is `2*i`
- The label of the right-child of node `i` is `2*i+1`

If it helps you, draw the picture of the tree and manipulate stickies or pieces of paper to see how the algorithms for inserting and deleting nodes work.

Typically, we represent this *heap* as an `ArrayList` with the first item (at index 0) unused (maybe `null`).

- Add a `HeapExamples` class to your project.
- Make three examples of the following *heap*s by defining an `ArrayList<Integer>` for each case in your examples class and an `initHeaps` method to add the values in the appropriate order.

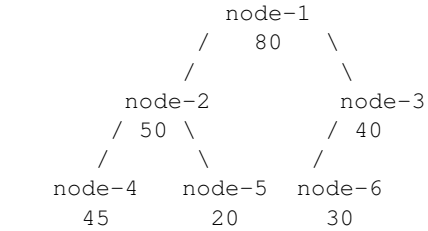
  To build the following *heap*

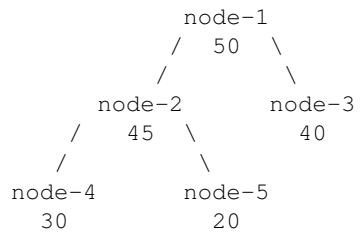


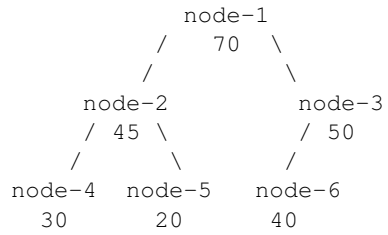
  we would proceed as follows:

```
ArrayList<Integer> myheap = new ArrayList<Integer>();

void initHeap(){
  this.myheap.add(null); // the unused first item
  this.myheap.add(70);
  this.myheap.add(60);
  this.myheap.add(40);
  this.myheap.add(35);
  this.myheap.add(50);
}
```

  Here are the three heaps to define:







- Define the class `PriorityQueue<T>` that will represent a *heap-based* priority queue. It has two fields: an `ArrayList<T>` and a `Comparator<T>` that determines the ordering of priorities.
- Design the method `isLeaf` that consumes a node label (an `int`) and returns `true` if the node has no children (remember the numberings...).
- Design the method `higherPriorityChild` that consumes the index of a node that is not a *leaf* and produces the index of its child with the highest priority.

  Note: If the node has only one child, then this will be the one with the higher priority, of course.
- Design the method `insert` that inserts a new node into the *heap*.

  Here is what we went over in lecture yesterday:

  - Insert the new item at the next position in the bottom level (the end of the list representation). Say, this is position k.
  - Upheap from k:
    While ($k > 1$ and $heap(k) > heap(k/2)$) {
    swap $heap(k)$ and $heap(k/2)$
    set k to $k/2$
    }

- Design the method `remove` that removes the node with the highest priority from the *heap*.

  Here is what we went over in lecture yesterday:

  - save the *root* item as a temporary
  - move the *last* item into the root position
  - Downheap from $k = 1$:
    While (k is not a leaf) {
      find ck the node with the larger child of the node k
      if $heap(k) < heap(ck)$ {
        swap $heap(k)$ and $heap(ck)$
        set k to ck }
      else stop
    }

11.2 Huffman Coding

Imagine that we want to define the most efficient way for encoding the letters of alphabet using only sequences of bits (values 0 and 1). David Huffman gave us some suggestions.

We start by looking at the text we want to encode. Assume it is represented as a single `String`. For example, the text may be the following 45 characters:

```
In the midst of the word he was trying to say
....x....x....x....x....x....x....x....x....x
```

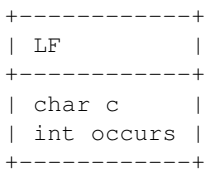
Or, you can use a smaller (20 character) String from a popular song:

```
oh, say, can you see
....x....x....x....x
```

1. Our first task is to produce a histogram that records the frequencies of each character (including space) occurs in the given String.

   We decided to use the following the data representation for the histogram data, so that you can easily tell how often each letter appears in the text. Our histogram is then an `ArrayList<LF>`.

Note: This is very similar the the Shakespeare problem in your current homework assignment, though we use the primitive Java type **`char`** to represent individual letters. Character literals are enclosed in single quotes: e.g., `'a'` or `'Z'`.



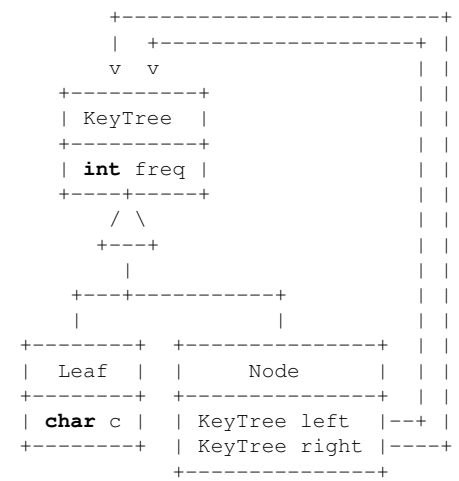
Make an example of the histogram you would get from the shorter text given previously.

2. Design the method `computeHisto` that computes the histogram for a given String of text. String contains a helpful method `charAt(int)` that returns the character from the string at the given index.

   Feel free to implement this method using your favorite form of Java loop... we suggest *for-each* (ask your friendly TA/Tutor if you need help).

3. Given the letter frequencies, we can build a binary-tree that represents the optimal encoding for each letter in our `String`. We first show examples of the trees that represent such encodings.

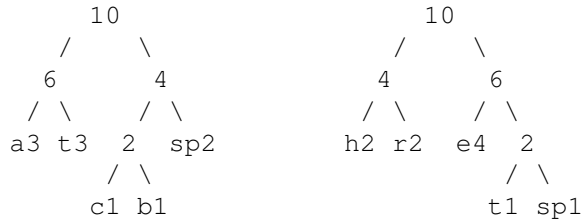
   The `KeyTree` class hierarchy is defined as:



The code that defines these classes is given below:

```
// Represents a Huffman Tree
abstract class KeyTree{
   // the frequency of this character or collection
   int freq;

   KeyTree(int freq){
      this.freq = freq;
   }
}

// Represents a Single Character
class Leaf extends KeyTree{
   char c;

   Leaf(char c, int freq){
      super(freq);
      this.c = c;
   }
}
// Represents a split in the KeyTree
class Node extends KeyTree{
   KeyTree left;
   KeyTree right;

   Node(KeyTree left, KeyTree right){
      super(left.freq + right.freq);
      this.left = left;
      this.right = right;
   }
}
```

Make examples of the following two trees. The first one represents the optimal encoding for the `String` *cat at bat*, the second one represents the optimal encoding for the `String` *here there* (the number next to the each `Leaf` is it's frequency):



4. To construct our encoding we can save the histogram data in a priority queue of `KeyTree`s, where the highest priority item is the one with the *lowest frequency*. This means we first need a `Comparator<KeyTree>` that compares using `freq`.

Design the comparator `ByFreq`.

5. The encoding for the character `'t'` in our first example is the `String` `"01"`, but in the second example it is `"110"`. Notice that the numbers `0` and `1` tell us whether we should choose the `left` or `right` tree.

   Design the method `findPath` for the `KeyTree` classes that consumes a character and produces a `String` representing this encoding, ending with the given character.

   So, for the input `'t'` the first tree would produce `"01t"`, and the second tree would produce `"110t"`.

6. Design the method `encode` that consumes a `KeyTree` and a `String` that consists only of the characters encoded in the given `KeyTree` and produces the `String` of `0`s and `1`s that represents the encoding of the given `String`.

   Note: In a real application each character `0` or `1` in the encoding would be represented as a single *bit*. Considering that the encoding of every character in a `String` usually requires 8 *bits*, this is typically a serious improvement.

7. Design the method `nextChar` for the `KeyTree` classes that consumes a `String` of `0`s and `1`s and produces the **`char`** that the encoding represents. The method should throw an exception if the given `String` is not a valid encoding.

   So, in the first tree the input `"01"`, would produce `'t'`, in the second tree the input `"110"` would produce `'t'`.

8. Design the method `decode` for the `KeyTree` classes that consumes a `String` that represents an encoding of a sequence of characters and produces a `String` that the encoding represents. The method should throw an exception if the given `String` is not a valid encoding.

   So, in the first tree the input `"1000001"`, would produce the `"cat"`, and in the second tree `"1100010"` would produce `"the"`. The second tree would fail for the `"001"`.

9. Design the method `buildTree` that consumes a priority queue you built and produces the `KeyTree` that represents the encoding. It works as follows:

   - if the priority queue is empty, no tree can be built and we should throw an exception.

- if the size of the priority queue is 1, it contains a single `KeyTree`, so the tree is removed and returned.
- otherwise, the method removes the two `KeyTree`s with the highest priority (lowest frequencies) and adds a new `KeyTree` to the queue that has the removed items as its `left` and `right` subtrees.

You can use your heap implementation from earlier, or you can look up the documentation for the methods for the Java `PriorityQueue` in the *Java Collections Library*.

The method `buildPQ` shown below can be used to build the initial priority queue (of `Leafs`) from the `LF` data, though you can write a similar function if you use your Heap implementation.

```
// insert the frequency data into a priority queue
// produce an priority queue of LF data
public PriorityQueue<KeyTree> buildPQ(ArrayList<LF> lfList){
   PriorityQueue<KeyTree> pq =
        new PriorityQueue<KeyTree>(lfList.size(), new ByFreq());
   for(LF lf : lfList){
       pq.offer(new Leaf(lf.c, ch.occurs));
   }
   return pq;
}
```