Project 4

Due Date: Friday 19 November 2021 by 11:59 PM

General Guidelines.

The APIs given below describe the required methods in each class. You may need additional methods or classes that will not be directly tested but may be necessary to complete the assignment.

Unless otherwise stated in this handout or in the files themselves, you are welcome (and encouraged) to add to/alter the provided java files as well as create new java files as needed. Your solution must be coded in Java.

In general, you are not allowed to import any additional classes in your code without explicit permission from your instructor!
For this assignment, you may import any classes necessary for reading the input files.

Note on academic dishonesty: Please note that it is considered academic dishonesty to read anyone else's solution code, whether it is another student's code, code from a textbook, or something you found online. You MUST do your own work! It is also considered academic dishonesty to share your code with another student. Anyone who is found to have violated this policy will be subject to consequences according to the syllabus and university policy. **For this assignment, you are allowed to work with a partner. Only groups of 2 are allowed, and you should both submit the project separately for grading. Please make a comment in the *Huffman.java* file indicating who you worked with so that the TAs know to expect the same code.**

Note on grading and provided tests: The provided tests are to help you as you implement the project and (in the case of auto-graded sections) to give you an idea of the number of points you are likely to receive. Please note that the points indicated when you run these tests locally are not your final grade. Your solution will be graded by the TAs after you submit. Please also note that these test cases are not likely to be exhaustive and find every possible error. Part of programming is learning to test and debug your own code, so if something goes wrong, we can help guide you in the debugging process but we will not debug your code for you.

Project Overview.

In this project, you will implement a few data structures and then use them in an implementation of Huffman Encoding. There are three programming parts, each tested separately, and no written report. Note that Parts 1 and 2 can be implemented independently of the other parts, but Part 3 depends on Parts 1 and 2.

Part 1. Hashtable

In this part, you must implement a hashtable that uses linear probing to handle collisions. The hashtable should be array-based and should be generic. In order to help with some of the details (especially the generics), you re provided with a *Pair.java* class, as well as a *Hashtable.java* class with starter code.

Requirements for the Hashtable:

- generic, using the provided Pair class
- array-based
- uses linear probing
- maintains the load factor to be between ⅛ and ½
- dynamically resized by halving or doubling
- uses prime number table sizes (there are methods provided for this)
- default starting capacity of the array is 11
- implemented in file called Hashtable.java

`Hashtable.java API`

`Method`	**`Description`**
`Hashtable()`	`default constructor--set initial table size to 11`
`Hashtable(int cap)`	`constructor--set initial table size to` *`cap`*
`V get(K key)`	`return the value associated with` *`key`* `or null if` *`key`* `is not in the table`
`void put(Key key, V value)`	`put the key-value pair into the table or update the value if the key already exists in the table; if the put causes the load factor`

	`to exceed ½, resize the table to be the first prime number >= 2*M (where M is the table size)`
`V remove(K key)`	`remove the key-value pair associated with` *`key`* `and return the value (or null if` *`key`* `does not exist in the table); if the` *`remove`* `causes the load factor to drop below ⅛ and M/2 would still be >= the initial capacity, resize the table to be the first prime number >= M/2 (where M is the table size)`
`int size()`	`return the number of elements in the table`
`Pair<K, V>[] getArray()`	`This method is already implemented and is only used for testing. Do not change it and do not use it in any of your submitted code.`

Part 2. Min Priority Queue

In this part, you must implement a heap-based min priority queue using an array. The PQ should be generic, accepting types that implement the interface Comparable.. In order to help with some of the details (especially the generics), you are provided with a *MinPQ.java* class that has starter code.

Requirements for the MinPQ:

- generic for types that implement Comparable
- array-based
- dynamically resizable by halving or doubling
- default starting capacity of the array is 10
- when *sinking* an element, if the children have the same value and they are less than the element you are sinking, swap with the left child
- implement *sink* and *swim* in separate private methods--this is just good coding practice and the TAs will check for it when they check your code
- implemented in file called *MinPQ.java*

MinPQ.java API

Method	**Description**
`MinPQ()`	default constructor--initial capacity is 10. (This is already implemented.)
`MinPQ(int cap)`	constructor--set initial array size to *cap*. (This is already implemented.)
`void insert(T item)`	insert the new item into the PQ; if the array is full before the insert, resize the array by doubling
`T delMin() throws EmptyQueueException`	remove and return the min item; throw EmptyQueueException (class provided) if the PQ is empty; resize array to ½ its size if the remove makes the number of items drop below ¼ of the array size AND if halving the array would not make its size go below the initial capacity
`T getMin() throws EmptyQueueException`	return the min item; throw EmptyQueueException (class provided) if the PQ is empty
`int size()`	return the number of elements currently in the PQ
`boolean isEmpty()`	return true if the PQ is empty and false otherwise
`T[] getArray()`	This method is already implemented and is only used for testing. Do not change it and do not use it in any of your submitted code.

Part 3. Huffman Encoding

Huffman Encoding is a method for data compression. The idea of data compression is to take data (often a text) and compress it so that it takes up less space.

For example, let **s** = “mississippi”.
Using **regular encoding with Extended ASCII**, each of those characters requires 8 bits to encode it, so the total number of bits for storing that string would be 8×11 = 88 bits. But why do we need 8 bits per character when there are only 4 distinct characters (*m*, *i*, *s*, and *p*) in this text?
The answer is, we really don’t, and that’s where data compression comes in.

The following go through a few options for encoding so that the text takes up less space. Note that the third option is the one you will implement, as it tends to be the best.

1. **run-length encoding**: The idea here is to indicate repeated characters with numbers, so **s** would be encoded as “m1i1s2i1p2i1”. This may be useful in situations where you have long strings of repeated characters, but not that helpful in regular English text. Even in this example, the encoding would be even worse than the original because it requires $8 \times 12 = 96$ bits. (You could probably reduce this by not encoding a number when a character is by itself, but it still won’t help that much.)

2. **fixed code**: In this method, we determine how many distinct characters we have and encode each one with a distinct bit encoding, where every character is encoded with the same number of bits. This is like using ASCII except that with fewer elements in the “alphabet,” we won’t need as many bits per character. In this case, we have 4 distinct characters, which means we need 2 bits per character. For example, we could map “m” to 00, “i” to 01, “s” to 10, and “p” to 11. That would mean that our encoding would take $2 \times 11 = 22$ bits. This is much better than the first two approaches. However, #3 is still better.

3. **Huffman Encoding** is a greedy algorithm that constructs a **variable-length code** for a specific text. What this means is each distinct character in the text is mapped to a bit encoding, but they are not all necessarily the same length. In this case, some of them may be longer than 2 bits, and some of them may be shorter. The key is that **the longer encodings map to less frequent characters and the shorter encodings map to more frequent characters.**

For example, we could use the following mapping:
$m = 000$
$i = 01$
$s = 1$
$p = 001$

With this mapping, the encoding of “mississippi” would be 00001110100100101, which is only 17 bits. This provides an optimal encoding that reduces the amount of space required to store this particular string.

The Huffman Encoding Algorithm is given below. Note that you are very likely to use both your Hashtable and your MinPQ in this implementation. You will probably also want some kind of Tree class that implements Comparable.

```
Algorithm HuffmanEncoding(X)
Input: X, a string of size n
Output: An optimal encoding tree for X
    Go through X and determine the frequency of each distinct
    character in X. (Hint: Use a hashtable for this.)
    Q := an empty min priority queue
    for each distinct character c in X:
        T := a single-node tree that stores (c, freq(c))
        Q.insert(T) //rank is the frequency
    end for
    while Q.size() > 1:
        T = Q.delMin()
        S = Q.delMin()
        U = T·S //join the two trees together with a new root
        that stores the sum of the frequencies in S and T
        Q.insert(U)
    end while
    return Q.delMin()
```

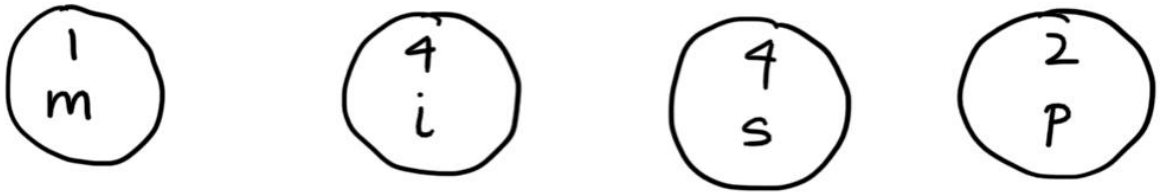
Example.

X = “mississippi”

HuffmanEncoding(X):

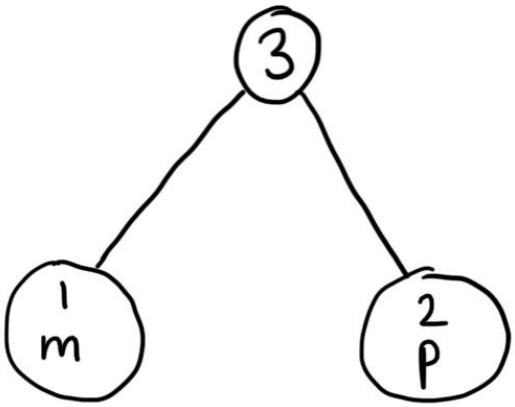
- run through X and finds the following frequencies:
  - (m, 1)
  - (i, 4)
  - (s, 4)
  - (p, 2)
- for each of these, construct a single-node tree and insert it into the min PQ (ranked by frequencies)

Currently, our trees look like this:

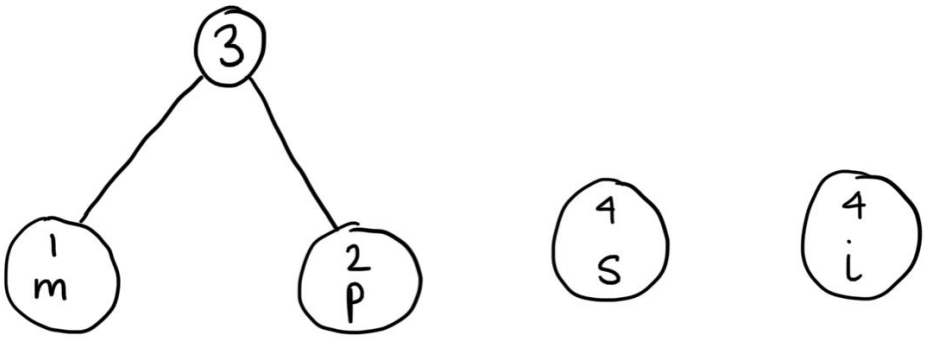


- until the PQ is down to just one tree:
    - remove the minimum twice:

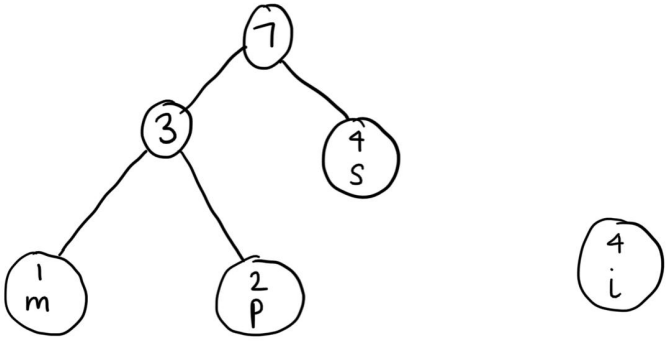
- join the two trees together by adding a new root with the combined frequencies



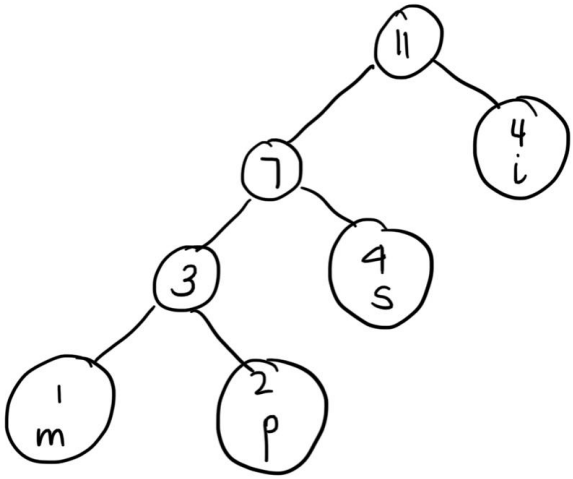
- insert that new tree into the PQ; now the PQ contains the following trees:



- keep looping until the PQ only has one tree in it
    - combining the first and second trees:



    - combining the last two trees:

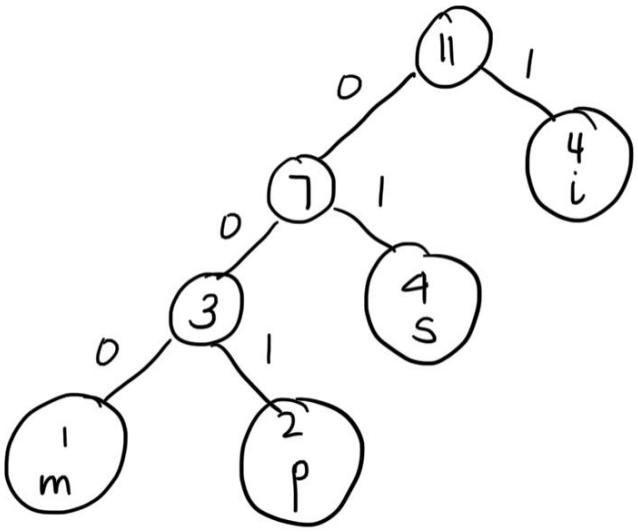


This is the final encoding tree. Below, you will find out how to use it to compress and decompress the text.

A few things to notice…

- Characters only occur at the bottom of the tree. This ensures that no encoding is a prefix of another encoding. If this wasn't the case, the code system would be ambiguous and decoding would not necessarily return correct results.
- Less frequent characters are further down the tree than more frequent characters. This corresponds to having longer/shorter encodings.

Once you have the tree set up, you use the tree to determine encodings for each of the characters. This requires that you traverse the tree (using DFS). It also requires that you store the mappings somewhere. I suggest using another hashtable. The actual method for encoding is pretty simple. Go down a path in the tree. If you go to the left, add a 0, if you go to the right, add a 1. When you reach a leaf node, map that character to that bitstring.



Character Encodings:

$m = 000$
$p = 001$
$s = 01$
$i = 1$

Once you have these mappings, you can encode the original text by replacing each character with the appropriate bit string: $000\ 1\ 01\ 01\ 1\ 01\ 01\ 1\ 001\ 001\ 1$. (Note that you wouldn't really put spaces in between these. I just added them to help clarify how the encoding maps back to the characters.)

Note: You are not expected to implement this with actual bits. We are just simulating the process by using “0” and “1” strings to represent bits. Obviously, this is going to use more space, not less, but you’ll still get to see how the algorithm is supposed to work.

Note: Depending on how the PQ is implemented, how trees are joined together, and how ties are handled, you can get variations in the resulting tree, which results in variations in the encodings. **This is okay.** What should not vary is the total number of “bits” used to encode the original text, and this is what we will be testing for the encoding procedure.

Decoding:
Obviously, you need to also be able to decompress (or “decode”) the compressed text. For this, you would use the original tree to get the characters from the bit string. The encoded bitstring is: 00010101101010110010011.
We can decode this by using the tree, going left or right according to whether the next bit is a 0 or a 1. When we reach a leaf, we know the character.

000 → m; 1→ i; 01→ s; 01→ s; 1→ i; 01→ s; 01→ s; 1→ i; 001→ p; 001→ p; 1→ i

This is where it is essential that no encoding is a prefix of another encoding. That way there is only one possible path given the input bit string.

Below is the required API for the *Huffman.java* class.

`Huffman.java API`

`Method`	**`Description`**
`Huffman(String fn)`	`constructor--build Huffman Encoding from the text in the given file (`*`fn`*` is the file name)`
`public String encode()`	`encode the original text using the coding system→ use “0” and “1” to represent bits; return the encoded text`
`public String decode(String str)`	`use the Huffman Code to decode the String that is passed in; return the decoded String`

Submission Procedure.

To submit, please upload the following files to **lectura** and use the **turnin** command to submit. Once you log in to lectura and transfer your files, you can submit using the following command:

```
turnin cs345p4 Hashtable.java MinPQ.java Huffman.java
```

Upon successful submission, you will see this message:

```
Turning in:
     Hashtable.java -- ok
     MinPQ.java -- ok
     Huffman.java -- ok
All done.
```

Note: If your implementation uses any additional classes (that you created), include those in your submission as well.

Note: Your submission must be able to run on **lectura**, so make sure you give yourself enough time before the deadline to check that it runs and do any necessary debugging. I recommend finishing the project locally 24 hours before the deadline to make sure you have enough time to deal with any submission issues.

Grading.

The following rubric gives the basic breakdown of your grade for this Project.

Item	**Points**
Part 1 Code	25
Part 2 Code	25
Part 3 Code	45
Coding Style	5
Total	**100**

Other notes about grading:

- If you do not follow the directions, you may not receive credit for that part of the assignment.
- If you implement something correctly but in an inefficient way, you may not receive full credit.

- In cases where efficiency is determined by counting something like array accesses, it is considered academic dishonesty to *intentionally* try to avoid getting more access counts, so if you are in doubt, it is always best to ask. If you are asking, then we tend to assume that you are trying to do the project correctly.
- If you have questions about your graded project, you may contact the TAs and set up a meeting to discuss your grade with them in person. Regrades on programs that do not work on lectura may be allowed but will typically result in a deduction in points, which may vary depending on the situation. All regrade requests should be submitted according to the guidelines in the syllabus and within the allotted time frame.