Programming Assignment 3: Huffman

Background

ASCII and Binary Representation

In Java, every `char` has an equivalent `int` value between 0 and 255. For example:

```
public class Client {
    public static void main(String[] args) {
        char c1 = 'a';
        char c2 = 'z';
        System.out.println("'" + c1 + "' has an int value of " + ((int) c1));
        System.out.println("'" + c2 + "' has an int value of " + ((int) c2));
    }
}
```

In this assignment, we will go a step further: computers actually see integers as *binary* (1's and 0's). Binary is a lot like the normal numbers we use, except it's based around 2 instead of 10. For example, "1204" is written that way, because

$$1204 = 1 \cdot 10^3 + 2 \cdot 10^2 + 0 \cdot 10^1 + 4 \cdot 10^0$$

Similarly, because

$$97 = 0 \cdot 2^7 + 1 \cdot 2^6 + 1 \cdot 2^5 + 0 \cdot 2^4 + 0 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0$$

97 is "01100001" in binary! In fact, all `char`'s can be written using exactly 8 binary digits (also known as bits). This encoding, which is used pretty universally by computers, is called ASCII.

Huffman Encoding

Huffman encoding is an algorithm devised by David A. Huffman of MIT in 1952 for compressing data to make a file occupy a smaller amount of space. Unbelievably, this algorithm is still used today in a variety of very important areas. For example, mp3s and jpgs both use Huffman Coding. The general idea behind Huffman Coding is the following: **What if we used fewer than 8 bits for characters that are more common?**

Consider the following (simple) example. Imagine we have the following data in a text file:

```
bbbbaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
cccaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
ddaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
```

How many bits would it take to store with ASCII Coding?

First, we find the ASCII code for each letter:

- `a` → 01100001
- `b` → 01100010
- `c` → 01100011
- `d` → 01100100

Since each line has 80 letters, and each letter code is 8 bits, the number of bits required is:

$$80 \cdot 8 \cdot 4 = \mathbf{2560}$$

How many bits would it take to store with Huffman Coding?

Since `a` has the highest frequency, we use a short code for it, then we use the next (longest) short code for `b`, etc:

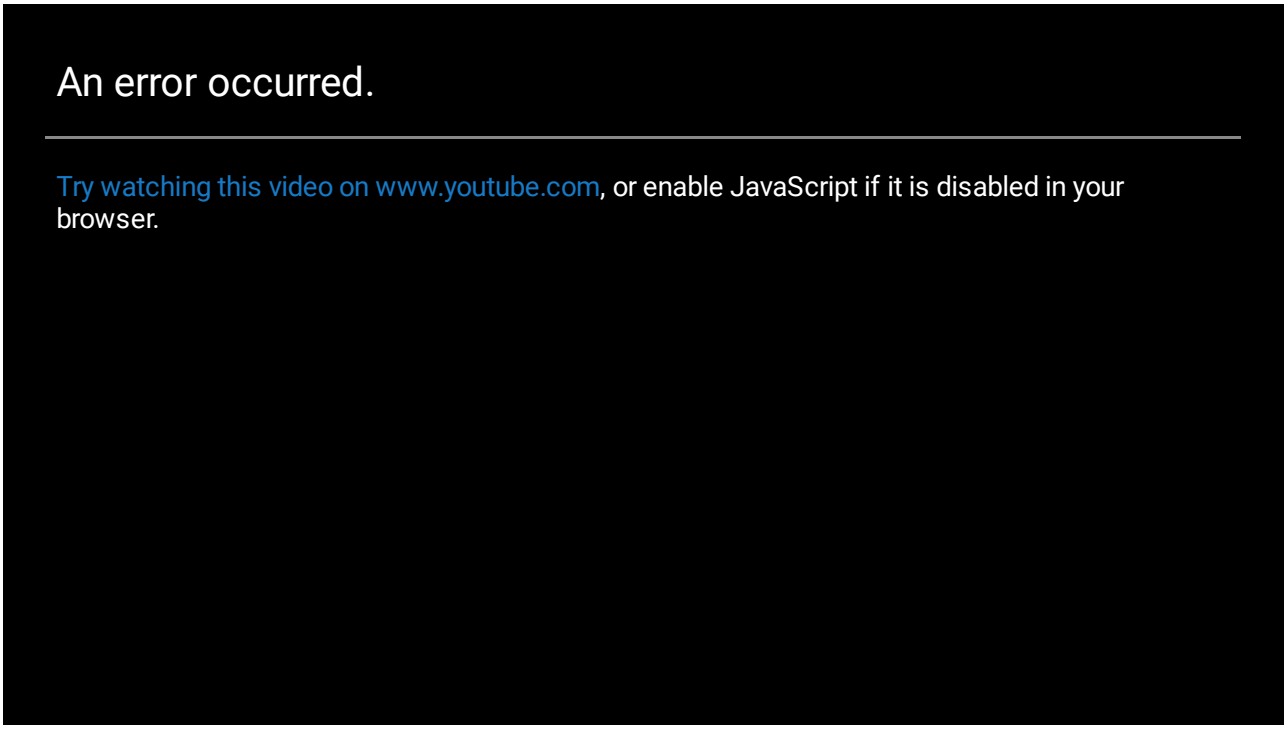
- `a` → 1
- `b` → 00
- `c` → 011
- `d` → 010

This data has 229 `a`'s, 4 `b`'s, 3 `c`'s, and 2 `d`'s. Since we need one bit for `a`, two for `b`, and three for `c` and `d`, the total count of bits is:

$$229 \cdot 1 + 4 \cdot 2 + 3 \cdot 3 + 2 \cdot 3 = \mathbf{255}$$

Whoa! By changing the coding, we compressed the data by a factor of 10!

Spec Walkthrough



Building a Huffman Code

In the previous slide, we magically arrived at the special binary sequences to be used as codes. In this section, we explain the algorithm to create these special binary sequences. Throughout this section, we will use the following text (which can be found in `simple-spec-example.txt`):

```
aba ab cabbb
```

Step 1: Count the Frequencies of the Characters

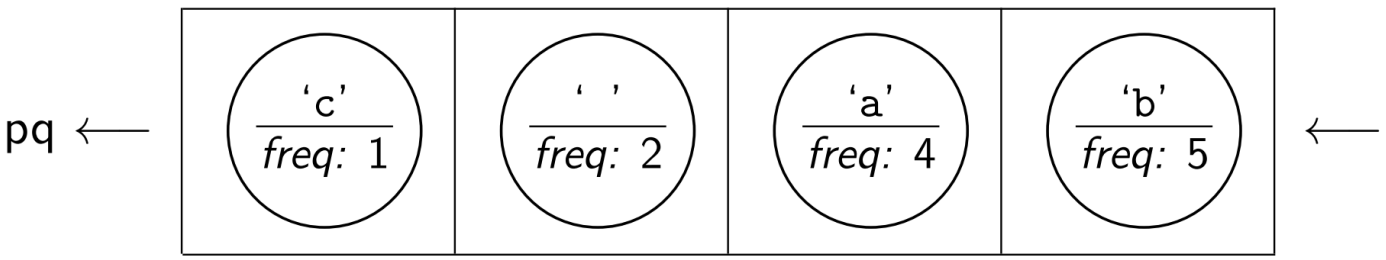
The file `simple-spec-example.txt` has the characters `'a'`, `'b'`, `'c'`, and `' '`. Counting up the frequencies, we get the following:

- `' '` → 2
- `'a'` → 4
- `'b'` → 5
- `'c'` → 1

Step 2: Create a Priority Queue Ordered By Frequency

Since our ultimate goal is to create a code based on frequencies, we need to use a data structure that helps us keep track of the order of the letters based on frequencies. We will use a *priority queue* for this. A *priority queue* is a queue that is ordered by *priorities* (in this case frequencies) instead of FIFO order. In other words, we can ask the priority queue to insert a new element (`add(element)`) and we can ask it to remove the highest priority element (`remove()`). We will use the `PriorityQueue<E>` class in Java for the implementation of priority queues, but we will **still use the `Queue<E>` interface for variables**. For example, to create a *priority queue* of integers: `Queue<Integer> pq = new PriorityQueue<>();`

For our Huffman Code, we want to remove the node with *lowest* frequency first. (Intuitively, the reason is that the things we remove first will end up having the longest codes). We begin by creating a node for each letter in our text:



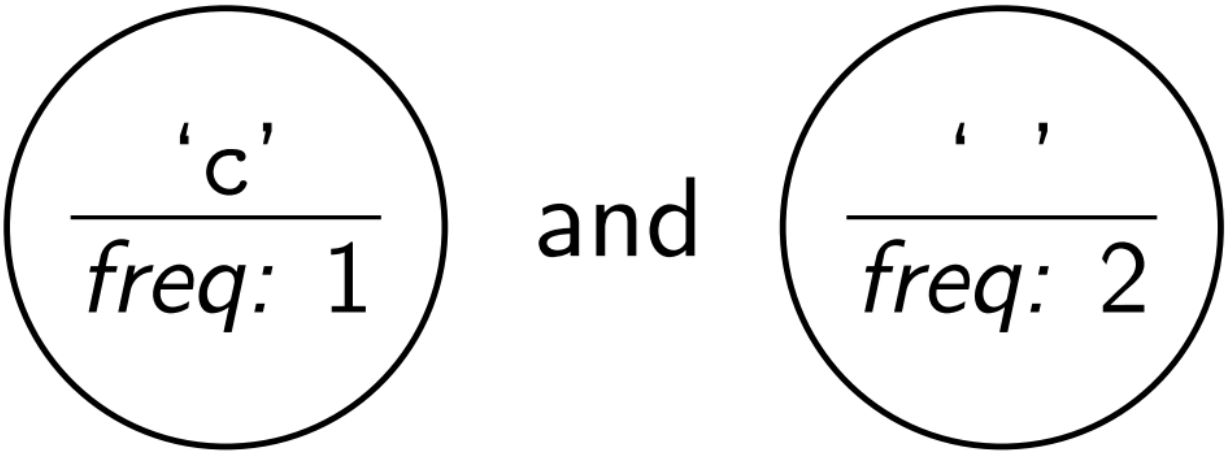
In Java, if you print out a priority queue, the elements will ***not*** appear in priority order. This is expected behavior, and can't be easily changed.

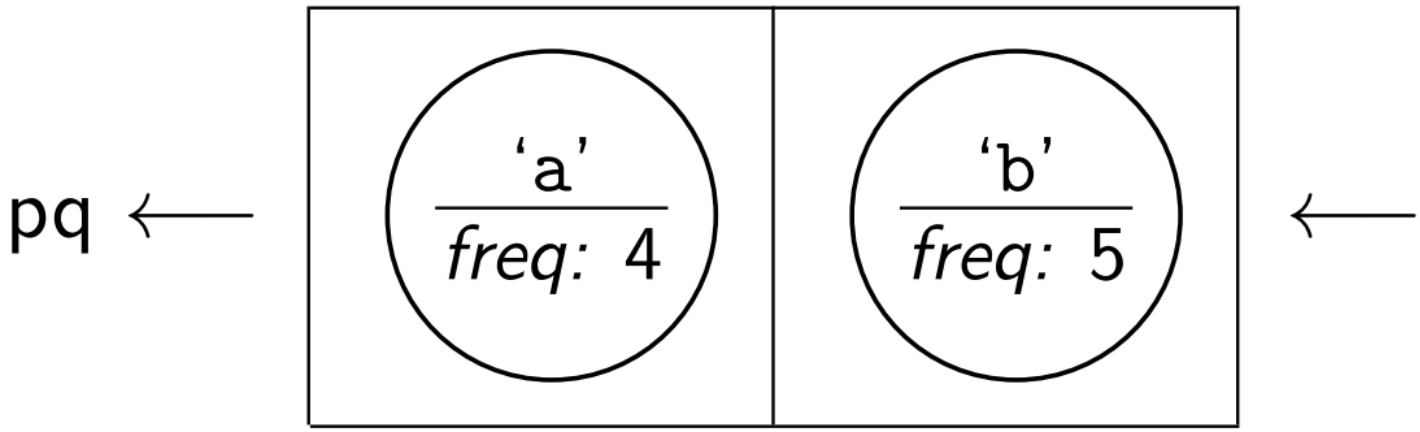
Step 3: Combine the Nodes Until Only One is Left

Now that we have a priority queue of the nodes, we want to put them together into a tree. To do this, we repeatedly remove the smallest two (the ones at the front of the priority queue) and create a new node with them as children. (The new node does not have a character associated with it.) Note that the priority queue is *arbitrary* in how it breaks ties; you should just take nodes in the order the priority queue provides them.

Step 3a: Remove Two Smallest

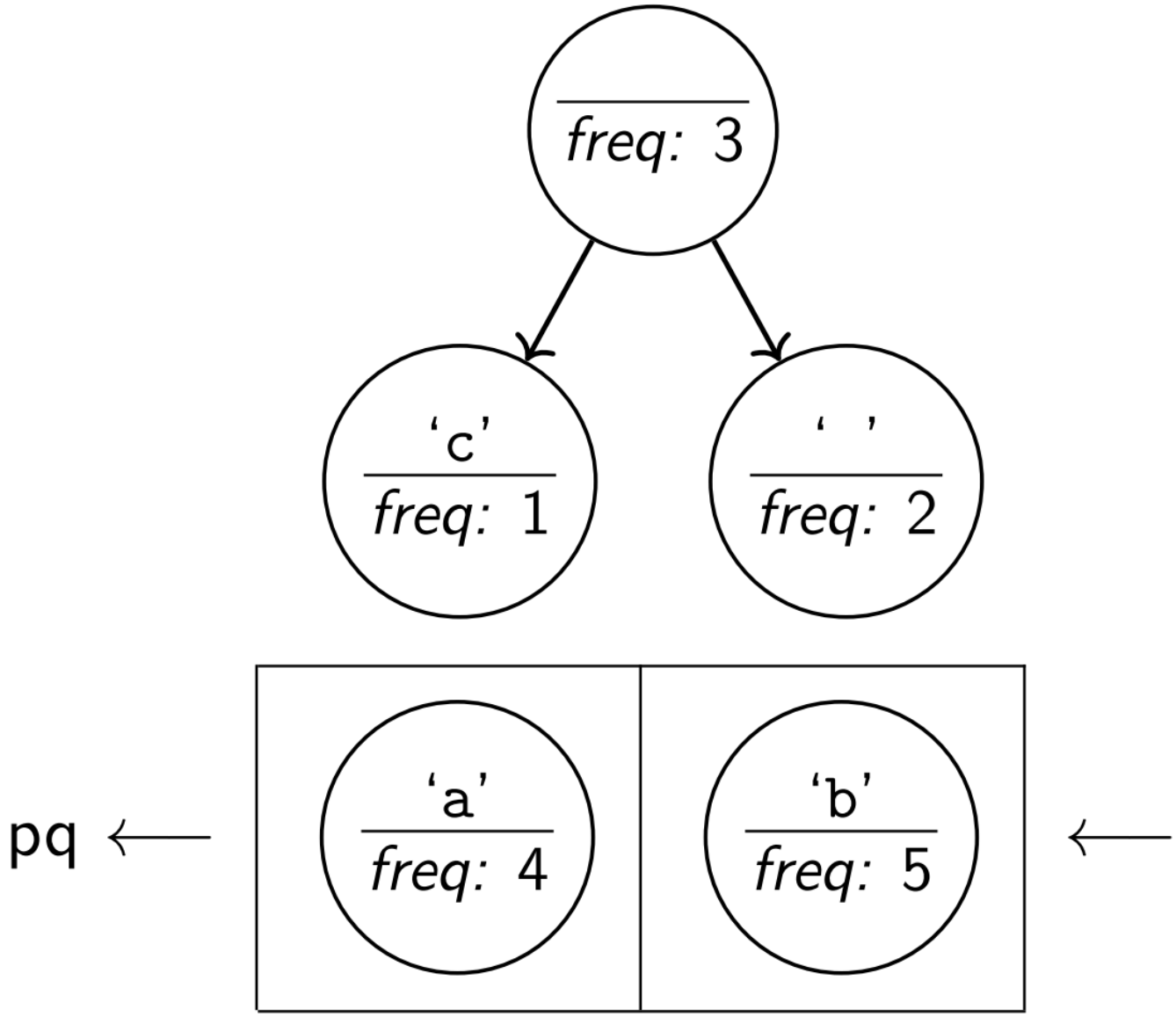
First, remove the smallest two nodes from the priority queue:





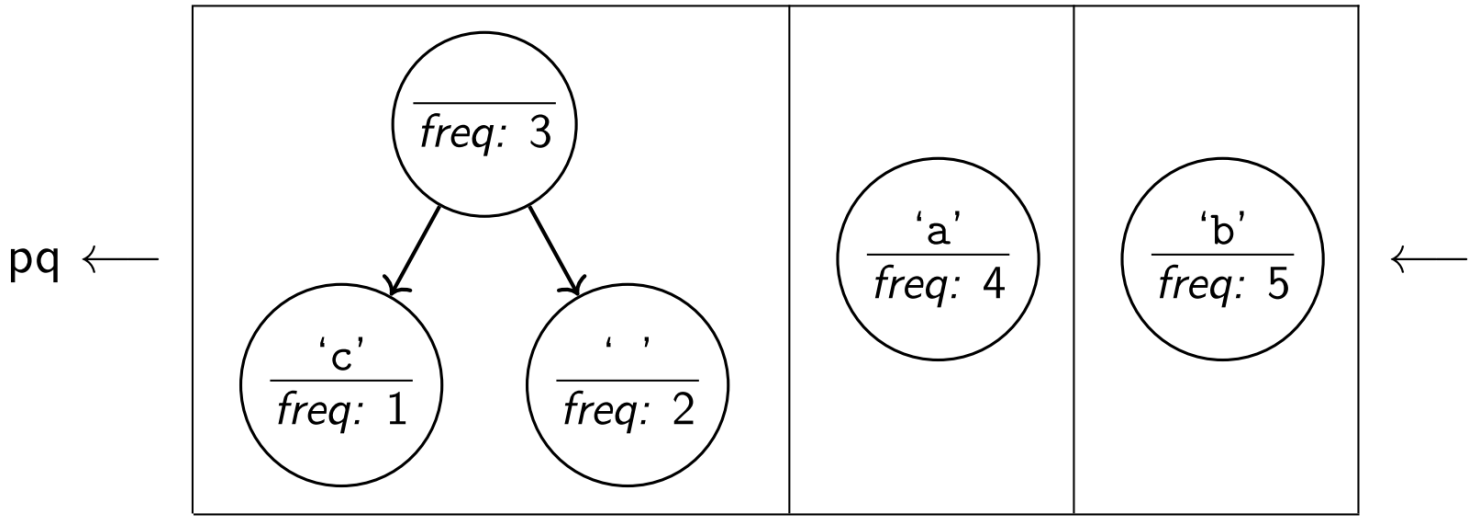
Step 3b: Combine Them Together

Then, create a new node. The frequency is the sum of two nodes:

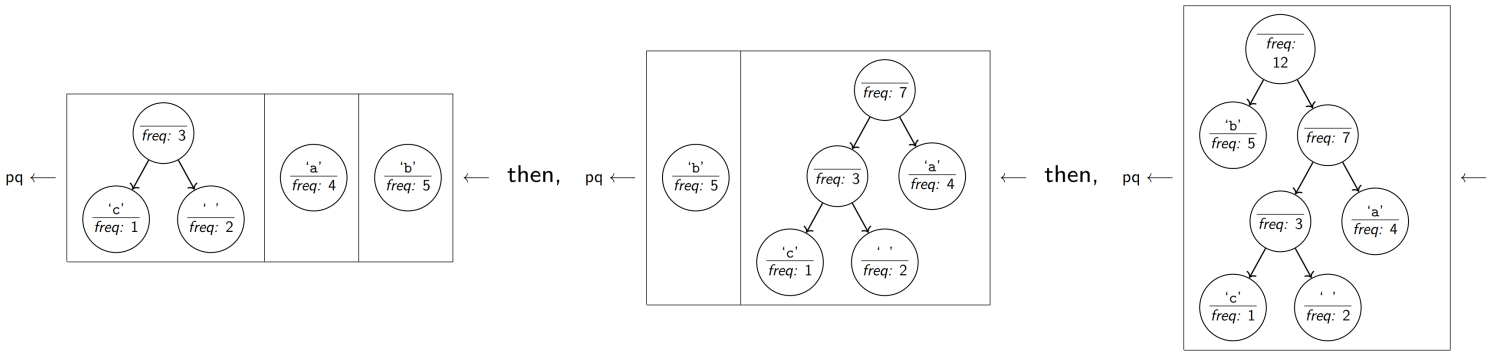


Step 3c: Add Back To Priority Queue

Now that we have a "new node", add it back to the priority queue:



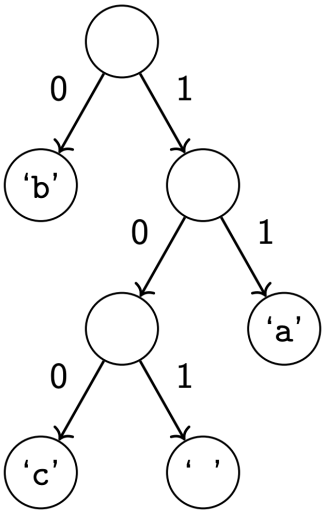
We repeat this process until there is only one node left in the priority queue. Here are the remaining steps (each time, we remove the two smallest frequencies, combine them, and put the result back):



Now that we only have one node left, we can use the tree we constructed to create the Huffman codes!

Step 4: Read Off the Huffman Codes

At this point, the frequencies of the letters have already been taken into account; so, we no longer even look at them. Thus, the tree we've constructed looks like the following:



simple-spec-example.code

```
98
0
99
100
32
101
97
11
```

To figure out the Huffman code for a letter, we traverse the tree from the root to the node with the letter in it. When we go left, we print a 0 and if we go right, we print a 1. For example, `'c'` would be 100, because we go right, then left, then left to reach it.

Just like in BrunelleFeed, we will output the tree in "standard format". Notice that the only actual information is in the leaves of the tree. So, the code file—which you can get by asking the main to "make a code"—will consist of line pairs: the first line will be the ASCII value of the character in the leaf and the second line will be the Huffman code for that character. For example, the output of the tree we just constructed would look like the above. The leaves should appear in the order of a **pre-order** traversal.

Specification

Learning Objectives

By completing this assignment, students will demonstrate their ability to:

- Implement a well-designed Java class to meet a given specification.
- Define data structures to represent compound and complex data
- Write a functionally correct Java class to represent a binary tree.
- Implement the `Comparable` interface.
- Write classes that are readable and maintainable, and that conform to provided guidelines for style, implementation, and performance.
- Produce clear and effective documentation to improve comprehension and maintainability of programs, methods, and classes.

Overview

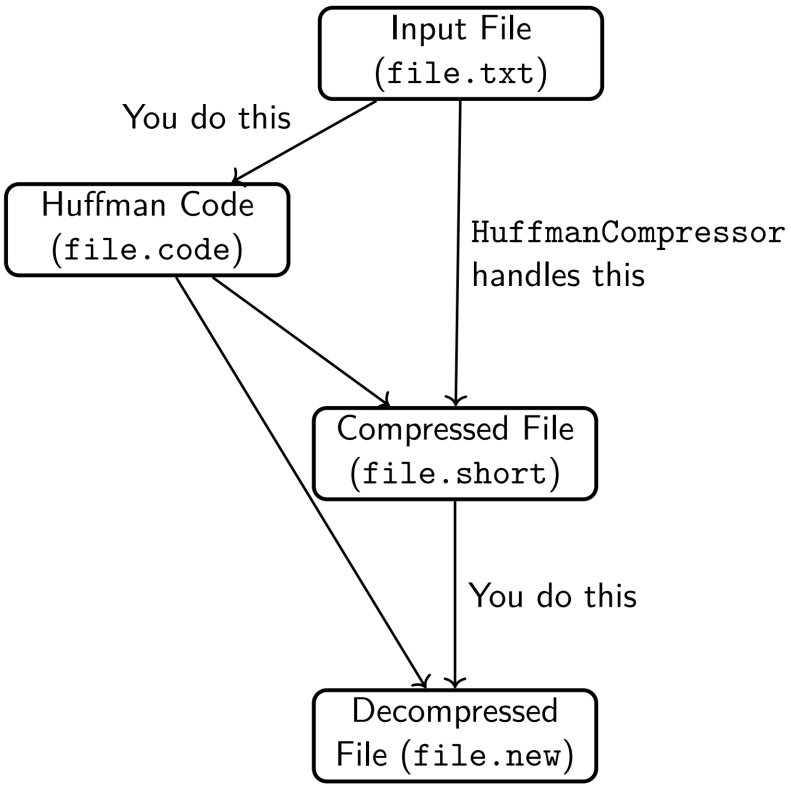
Now that we know how to construct a Huffman code, we are ready to understand the Huffman compression and decompression algorithms. Here is an overview of how they work:

Compression Algorithm

1. `HuffmanCompressor` generates the frequencies in a given file.
2. `HuffmanCode` creates the Huffman code from the frequencies.
3. `HuffmanCode` writes out the Huffman codes to a `.code` file.
4. `HuffmanCompressor` writes a `.short` file.

Decompression Algorithm

1. `HuffmanCode` reads in a `.code` file in standard format.
2. `HuffmanCode` writes out a `.new` file.



Client Program

`HuffmanCompressor` is a client program that we will provide to you. In addition to handling user interactions, it implements some of the steps of the compression and decompression algorithms that we are not asking you to deal with. Specifically, `HuffmanCompressor` handles:

- computing character frequencies for a given input file (these frequencies are passed to the first `HuffmanCode` constructor described below)
- compressing a text file using a given Huffman tree
- producing a `BitInputStream` from a given compressed input file (this stream is passed to the `translate` method)

You do not need to implement any of the above behavior. You **only** need to implement the behavior and methods described below. You may click "Expand" below to see a sample execution of each step (user input is **bold and underlined**).

Making a Huffman Code (produces .code file)

▶ Expand

Compressing a File (produces .short file)

▶ Expand

Decompressing a File (produces .new file)

▶ Expand

Compressing and then Decompressing a File (produces .short and .new file)

▶ Expand

Required Class

In this assignment, you will create a class called `HuffmanCode` to be used in compression of data. You are provided with a client program, `HuffmanCompressor.java`, that handles user interaction and calls your `HuffmanCode` methods to compress and decompress a given file.

Your `HuffmanCode` class must include the following methods:

```
public HuffmanCode(int[] frequencies)
```

- This constructor should initialize a new `HuffmanCode` object using the algorithm described above for building a Huffman code from an array of frequencies. `frequencies` is an array of frequencies where `frequences[i]` is the count of the character with ASCII value `i`.
    - Make sure to use a `PriorityQueue` to build the Huffman tree.
- If there exists a character with a frequency ≤ 0, the character should not be included in the `HuffmanCode` object.

```
public HuffmanCode(Scanner input)
```

- This constructor should initialize a new `HuffmanCode` object by reading in a previously constructed code from a `.code` file. You may assume the `Scanner` is not `null` and always contains data encoded in legal, valid standard format (see below).

```
public void save(PrintStream output)
```

- This method should store the current Huffman Code to the given output stream in the standard format (see below).

```
public void translate(BitInputStream input, PrintStream output)
```

- This method should read individual bits from the input stream and write the corresponding characters to the output. It should stop reading when the `BitInputStream` is empty. You may assume that the input stream contains a legal encoding of characters for this tree's Huffman Code. **See below for the methods in the `BitInputStream` class.**

`HuffmanNode`

Your `HuffmanCode` class must also include a class called `HuffmanNode` as a **private static inner**

class of `HuffmanCode` (recall `Commit` in `Repository` from Mini-Git and `QuizTreeNode` in `QuizTree` from BrunelleFeed). The contents of this class are up to you, but in particular, you must meet the following requirements:

- Your `HuffmanNode` class must implement the `Comparable<E>` interface, using frequencies to order nodes.
- The fields of the `HuffmanNode` class must be `public`.
    - All data fields should be declared `final` as well. This does not include fields representing the children of a node.

`BitInputStream`

The provided `BitInputStream` class reads data bit by bit. This will be useful for the `translate` method in `HuffmanCode`. The interface for `BitInputStream` works very much like a `Scanner` and should be used similarly. `BitInputStream` has the following methods:

```
public int nextBit()
```

- This method returns the next bit in the input stream. If there is no such bit, then it throws a `NoSuchElementException`.

```
public boolean hasNextBit()
```

- This method returns `true` if the input stream has at least one more bit, and `false` otherwise.

`.code` File Format

The `.code` files that are both created by the `save` method and read by the `Scanner` constructor will follow the same format. These files will contain a pair of lines to represent each character in the Huffman code. The first line in each pair will contain the ASCII code of the character, and the second line will contain the Huffman encoding for that character.

For example, consider the following sample file (`simple-spec-example.code`):

```
98
0
99
100
32
101
97
11
```

This file represents a Huffman code in which the character `'b'` (ASCII code 98) has the encoding `0`, `'c'` (ASCII code 99) has the encoding `100`, `' '` (ASCII code 32) has the encoding 101, and `'a'`

(ASCII code 97) has the encoding 11.

HINT: You can get the ASCII code of a `char` by casting the value to an `int`.

Creative Aspect (`secretmessage.short` and `secretmessage.code`)

Along with your program, you should turn in files named `secretmessage.short` and `secretmessage.code` that represents a “secret” compressed message from you to your section TA and its code file. The message can be anything you want, as long as it is not offensive. Your section TA will decompress your message with your tree and read it while grading.

You can create a file called `secretmessage.txt` in the Ed editor and use your Huffman compression algorithm to create `secretmessage.short` and `secretmessage.code`. You should delete `secretmessage.txt` afterward, as that file contains your "secret" message.

Implementation Guidelines

Your program should exactly reproduce the format and general behavior demonstrated in the Ed tests. Note that this assignment has two mostly separate parts: creating a Huffman code and decompressing a message using your Huffman code. Of the four methods to implement, two are relevant to each part. Our recommended approach is as follows: first, implement the required methods for creating a Huffman Code, and then proceed with implementing the methods for decompressing a message. See below for more details about the process.

Creating a Huffman Code

You will write methods to (1) create a Huffman code from an array of frequencies and (2) save the code you've created in standard format.

Frequency Array Constructor

▶ Expand

`save`

▶ Expand

Decompressing A Message

Next, you will write methods to (1) read in a `.code` file created with `save()` and (2) translate a compressed file back into a decompressed file.

`Scanner` Constructor

▶ Expand

`translate`

▶ Expand

Code Quality

As always, your code should follow all guidelines in the Code Quality Guide and Commenting Guide.

In particular, pay attention to these requirements:

- **`x = change(x)`:**
  - An important concept introduced in lecture was called `x = change(x)`. This idea is related to the proper design of recursive methods that manipulate the structure of a binary tree. **You should follow this pattern where necessary when modifying your trees**.
- **Avoid redundancy:**
  - Create "helper" method(s) to capture repeated code. As long as all extra methods you create are private (so outside code cannot call them), you can have additional methods in your class beyond those specified here.
  - If you find that multiple methods in your class do similar things, you should create helper method(s) to capture the common code.
- **Data Fields:**
  - Properly encapsulate your objects by making data fields in your `HuffmanCode` class private. (Fields in your `HuffmanNode` class should be public following the pattern from class.)
  - Avoid unnecessary fields; use fields to store important data of your objects but not to store temporary values only used in one place.
  - Fields should always be initialized inside a constructor or method, never at declaration.
- **Commenting**
  - Each method should have a comment including all necessary information as described in the Commenting Guide. Comments should be written in your own words (i.e. not copied and pasted from this spec) and should not include implementation details.

Huffman -- contains scaffold DO NOT PUBLISH

Be sure to complete all files:

- `HuffmanCode.java`: the main Huffman program
- Upload `secretmessage.code` and `secretmessage.short`

Instructions on how to run this assignment are different than most assignments. You can run your program as follows:

- Clicking "Terminal" and "Click here to activate the terminal" will **run** `HuffmanCompressor.java` and save output files to your workspace
- Click "Reset" in the top right of the terminal if you have already activated the terminal, this will **re-run** `HuffmanCompressor.java`
- Clicking "Mark" will **run all tests on HuffmanCode AND submit your code.**

You can safely ignore the following message from the terminal:
Note: Some input files use or override a deprecated API that is marked for removal.
Note: Recompile with -Xlint:removal for details.

Huffman

Download starter code:

Huffman.zip

Be sure to complete all files:

- `HuffmanCode.java`: the main Huffman program
- Upload `secretmessage.code` and `secretmessage.short`

Instructions on how to run this assignment are different than most assignments. You can run your program as follows:

- Clicking "Terminal" and "Click here to activate the terminal" will **run** `HuffmanCompressor.java` and save output files to your workspace
- Click "Reset" in the top right of the terminal if you have already activated the terminal, this will **re-run** `HuffmanCompressor.java`
- Clicking "Mark" will **run all tests on HuffmanCode AND submit your code.**

You can safely ignore the following message from the terminal:
Note: Some input files use or override a deprecated API that is marked for removal.
Note: Recompile with -Xlint:removal for details.

Reflection

The following questions will ask you practice **metacognition** to reflect on the topics covered on this assignment and your experience completing it. For each question, focus on your plan and/or process for working through the assignment along with the CS concepts. Think about things like how you organized your working time, what sorts of things tended to go wrong, and how you dealt with those errors or mistakes.

Please answer all questions.

Question 1

What are some situations in which compression, and lossless compression in particular, would be important?

No response

Question 2

Do you think Huffman coding is always a good choice for compression? Can you imagine scenarios in which we might *not* want to use Huffman coding? Why or why not?

No response

Question 3

Describe how you went about testing your implementation. What specific situations and/or test cases did you consider? Why were those cases important?

No response

Question 4

What skills did you learn and/or practice with working on this assignment?

No response

Question 5

What did you struggle with most on this assignment?

No response

Question 6

What questions do you still have about the concepts and skills you used in this assignment?

No response

Question 7

About how long (in hours) did you spend on this assignment? (Feel free to estimate, but try to be close.)

No response

Question 8

Was any part of the specification or requirements unclear? If so, which part(s), how was it unclear, and how could it have been made more clear?

No response

Question 9

[OPTIONAL] Do you have any other feedback, questions, or comments about this assignment?

(Note that we may not be able to respond to questions here, so please post on the message board if you would like a response!)

No response

□ Final Submission □

□ Final Submission□

Fill out the box below and click "Submit" in the upper-right corner of the window to submit your work.

Question

I attest that the work I am about to submit is my own and was completed according to the course Academic Honesty and Collaboration policy. If I collaborated with any other students or utilized any outside resources, they are allowed and have been properly cited. If I have any concerns about this policy, I will reach out to the course staff to discuss *before* submitting.

(Type "yes" as your response.)

No response

[OLD] Specification

Background

ASCII and Binary Representation

In Java, every `char` has an equivalent `int` value between 0 and 255. For example:

```
public class Client {
    public static void main(String[] args) {
        char c1 = 'a';
        char c2 = 'z';
        System.out.println("'" + c1 + "' has an int value of " + ((int) c1));
        System.out.println("'" + c2 + "' has an int value of " + ((int) c2));
    }
}
```

In this assignment, we will go a step further: computers actually see integers as *binary* (1's and 0's). Binary is a lot like the normal numbers we use, except it's based around 2 instead of 10. For example, “1204” is written that way, because

$$1204 = 1 \cdot 10^3 + 2 \cdot 10^2 + 0 \cdot 10^1 + 4 \cdot 10^0$$

Similarly, because

$$97 = 0 \cdot 2^7 + 1 \cdot 2^6 + 1 \cdot 2^5 + 0 \cdot 2^4 + 0 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0$$

97 is "01100001" in binary! In fact, all `char`'s can be written using exactly 8 binary digits (also known as bits). This encoding, which is used pretty universally by computers, is called ASCII.

Huffman Encoding

Huffman encoding is an algorithm devised by David A. Huffman of MIT in 1952 for compressing data to make a file occupy a smaller amount of space. Unbelievably, this algorithm is still used today in a variety of very important areas. For example, mp3s and jpgs both use Huffman Coding. The general idea behind Huffman Coding is the following: **What if we used fewer than 8 bits for characters that are more common?**

Consider the following (simple) example. Imagine we have the following data in a text file:

```
bbbbaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
cccaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
ddaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
```

How many bits would it take to store with ASCII Coding?

First, we find the ASCII code for each letter:

- `a` → 01100001
- `b` → 01100010
- `c` → 01100011
- `d` → 01100100

Since each line has 80 letters, and each letter code is 8 bits, the number of bits required is:

$$80 \cdot 8 \cdot 4 = \mathbf{2560}$$

How many bits would it take to store with Huffman Coding?

Since `a` has the highest frequency, we use a short code for it, then we use the next (longest) short code for `b`, etc:

- `a` → 1
- `b` → 00
- `c` → 011
- `d` → 010

This data has 229 `a`'s, 4 `b`'s, 3 `c`'s, and 2 `d`'s. Since we need one bit for `a`, two for `b`, and three for `c` and `d`, the total count of bits is:

$$229 \cdot 1 + 4 \cdot 2 + 3 \cdot 3 + 2 \cdot 3 = \mathbf{255}$$

Whoa! By changing the coding, we compressed the data by a factor of 10!

Learning Objectives

By completing this assignment, students will demonstrate their ability to:

- Implement a well-designed Java class to meet a given specification.
- Define data structures to represent compound and complex data
- Write a functionally correct Java class to represent a binary tree.
- Implement the `Comparable` interface.
- Write classes that are readable and maintainable, and that conform to provided guidelines for style, implementation, and performance.
- Produce clear and effective documentation to improve comprehension and maintainability of programs, methods, and classes.

Required Class

In this assignment, you will create a class called `HuffmanCode` to be used in compression of data. You are provided with a client program, `HuffmanCompressor.java`, that handles user interaction and calls your `HuffmanCode` methods to compress and decompress a given file.

Your `HuffmanCode` class must include the following methods:

```
public HuffmanCode(int[] frequencies)
```

- This constructor should initialize a new `HuffmanCode` object using the algorithm described above for building a Huffman code from an array of frequencies. `frequencies` is an array of frequencies where `frequences[i]` is the count of the character with ASCII value `i`.
    - Make sure to use a `PriorityQueue` to build the Huffman tree.
- If there exists a character with a frequency <= 0, the character should not be included in the `HuffmanCode` object.

```
public HuffmanCode(Scanner input)
```

- This constructor should initialize a new `HuffmanCode` object by reading in a previously constructed code from a `.code` file. You may assume the `Scanner` is not `null` and always contains data encoded in legal, valid standard format (see below).

```
public void save(PrintStream output)
```

- This method should store the current Huffman Code to the given output stream in the standard format (see below).

```
public void translate(BitInputStream input, PrintStream output)
```

- This method should read individual bits from the input stream and write the corresponding characters to the output. It should stop reading when the `BitInputStream` is empty. You may assume that the input stream contains a legal encoding of characters for this tree's Huffman Code. **See below for the methods in the `BitInputStream` class.**

`HuffmanNode`

Your `HuffmanCode` class must also include a class called `HuffmanNode` as a **private static inner class** of `HuffmanCode` (recall `Commit` in `Repository` from Mini-Git and `QuizTreeNode` in `QuizTree` from BrettFeed). The contents of this class are up to you, but in particular, you must meet the following requirements:

- Your `HuffmanNode` class must implement the `Comparable<E>` interface, using frequencies to order nodes.

- The fields of the `HuffmanNode` class must be `public`.
    - All data fields should be declared **`final`** as well. This does not include fields representing the children of a node.

`BitInputStream`

The provided `BitInputStream` class reads data bit by bit. This will be useful for the `translate` method in `HuffmanCode`. The interface for `BitInputStream` works very much like a `Scanner` and should be used similarly. `BitInputStream` has the following methods:

```
public int nextBit()
```

- This method returns the next bit in the input stream. If there is no such bit, then it throws a `NoSuchElementException`.

```
public boolean hasNextBit()
```

- This method returns `true` if the input stream has at least one more bit, and `false` otherwise.

`.code` File Format

The `.code` files that are both created by the `save` method and read by the `Scanner` constructor will follow the same format. These files will contain a pair of lines to represent each character in the Huffman code. The first line in each pair will contain the ASCII code of the character, and the second line will contain the Huffman encoding for that character.

For example, consider the following sample file (`simple-spec-example.code`):

```
98
0
99
100
32
101
97
11
```

This file represents a Huffman code in which the character `'b'` (ASCII code 98) has the encoding `0`, `'c'` (ASCII code 99) has the encoding `100`, `' '` (ASCII code 32) has the encoding 101, and `'a'` (ASCII code 97) has the encoding 11.

HINT: You can get the ASCII code of a `char` by casting the value to an `int`.

Creative Aspect (`secretmessage.short` and

`secretmessage.code`)

Along with your program you should turn in files named `secretmessage.short` and `secretmessage.code` that represent a “secret” compressed message from you to your TA and its code file. The message can be anything you want, as long as it is not offensive. Your TA will decompress your message with your tree and read it while grading.

You can create a file called `secretmessage.txt` in the Ed editor and use your Huffman compression algorithm to create `secretmessage.short` and `secretmessage.code`. You should delete `secretmessage.txt` afterwards as that file contains your "secret" message.

[OLD] Building a Huffman Code

In the previous slide, we magically arrived at the special binary sequences to be used as codes. In this section, we explain the algorithm to create these special binary sequences. Throughout this section, we will use the following text (which can be found in `simple-spec-example.txt`):

```
aba ab cabbb
```

Step 1: Count the Frequencies of the Characters

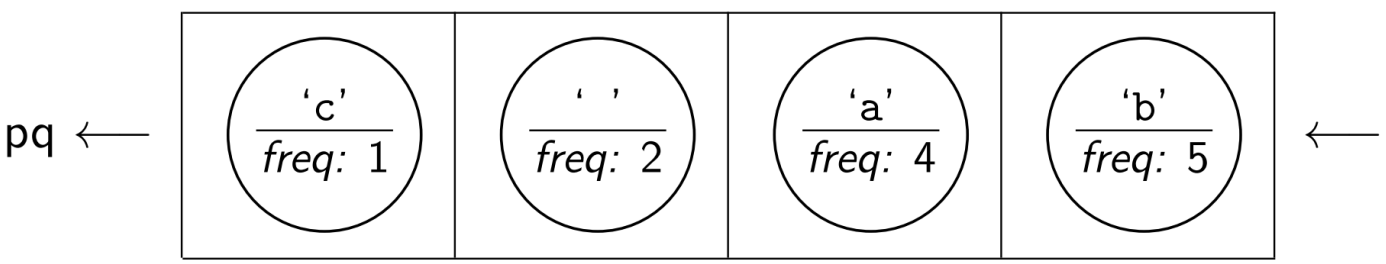
The file `simple-spec-example.txt` has the characters `'a'`, `'b'`, `'c'`, and `' '`. Counting up the frequencies, we get the following:

- `' '` → 2
- `'a'` → 4
- `'b'` → 5
- `'c'` → 1

Step 2: Create a Priority Queue Ordered By Frequency

Since our ultimate goal is to create a code based on frequencies, we need to use a data structure that helps us keep track of the order of the letters based on frequencies. We will use a *priority queue* for this. A *priority queue* is a queue that is ordered by *priorities* (in this case frequencies) instead of FIFO order. In other words, we can ask the priority queue to insert a new element (`add(element)`) and we can ask it to remove the highest priority element (`remove()`). We will use the `PriorityQueue<E>` class in Java for the implementation of priority queues, but we will **still use the `Queue<E>` interface for variables**. For example, to create a *priority queue* of integers: `Queue<Integer> pq = new PriorityQueue<>();`

For our Huffman Code, we want to remove the node with *lowest* frequency first. (Intuitively, the reason is that the things we remove first will end up having the longest codes). We begin by creating a node for each letter in our text:



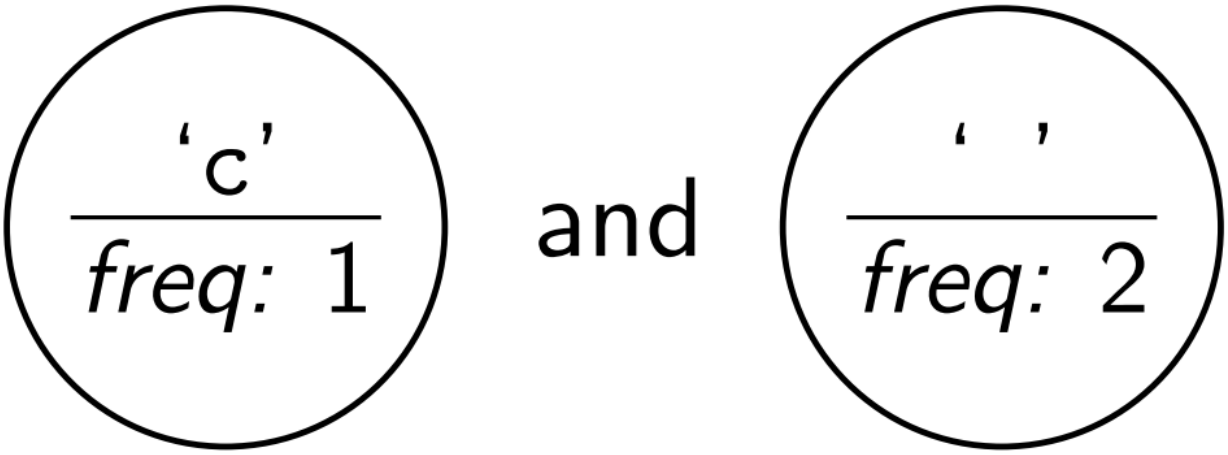
In Java, if you print out a priority queue, the elements will ***not*** appear in priority order. This is expected behavior, and can't be easily changed.

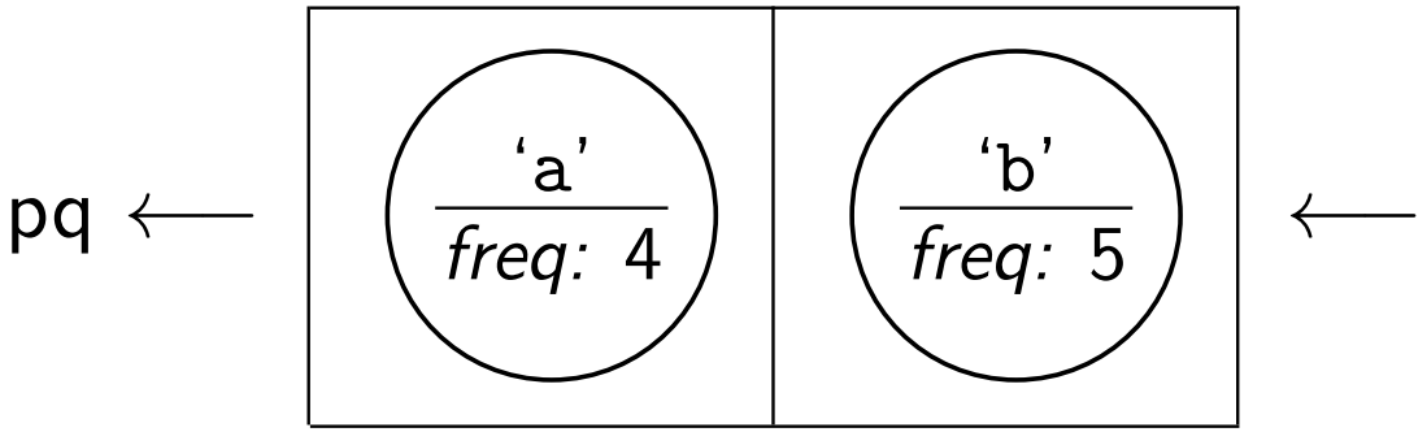
Step 3: Combine the Nodes Until Only One is Left

Now that we have a priority queue of the nodes, we want to put them together into a tree. To do this, we repeatedly remove the smallest two (the ones at the front of the priority queue) and create a new node with them as children. (The new node does not have a character associated with it.) Note that the priority queue is *arbitrary* in how it breaks ties; you should just take nodes in the order the priority queue provides them.

Step 3a: Remove Two Smallest

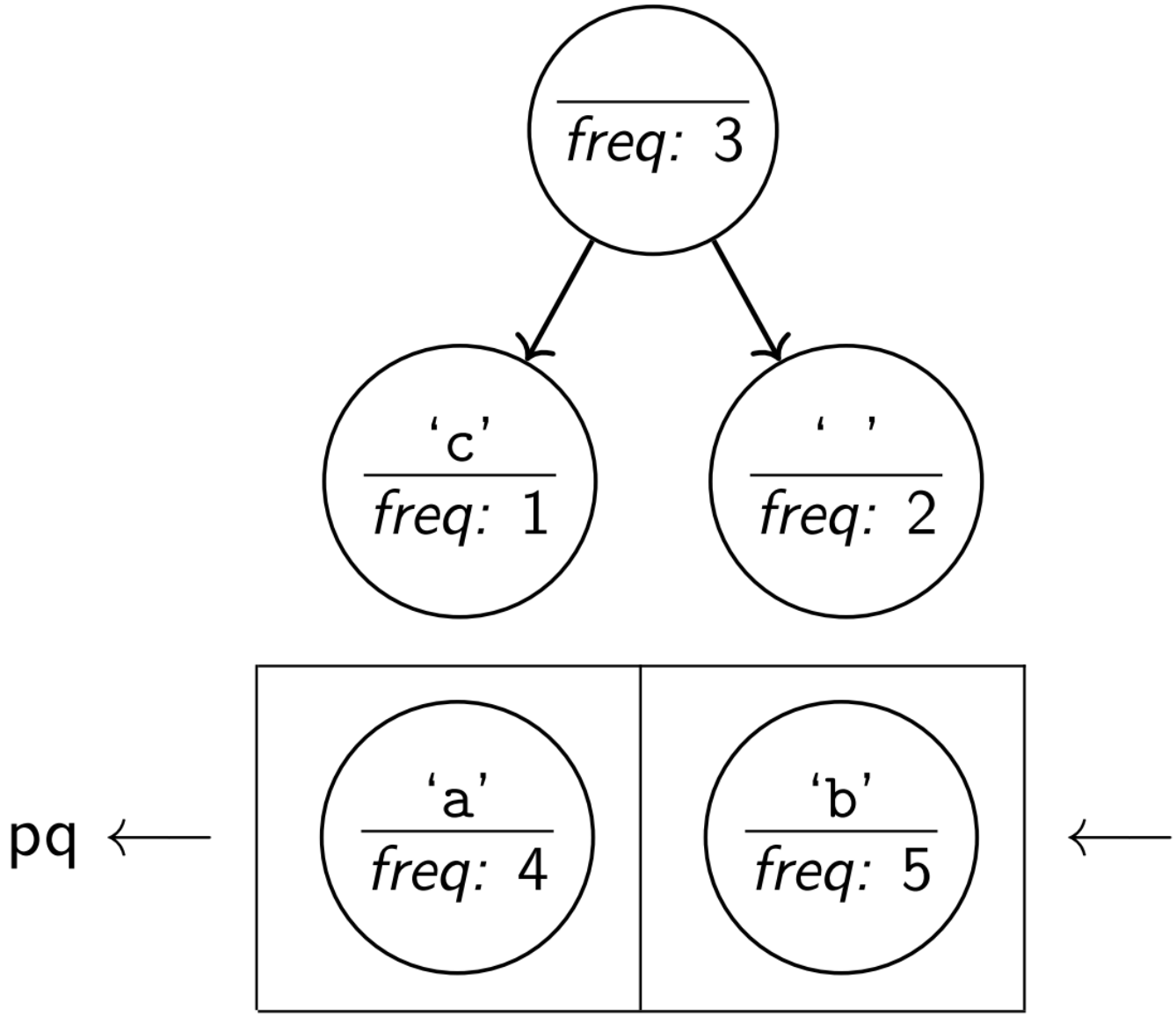
First, remove the smallest two nodes from the priority queue:





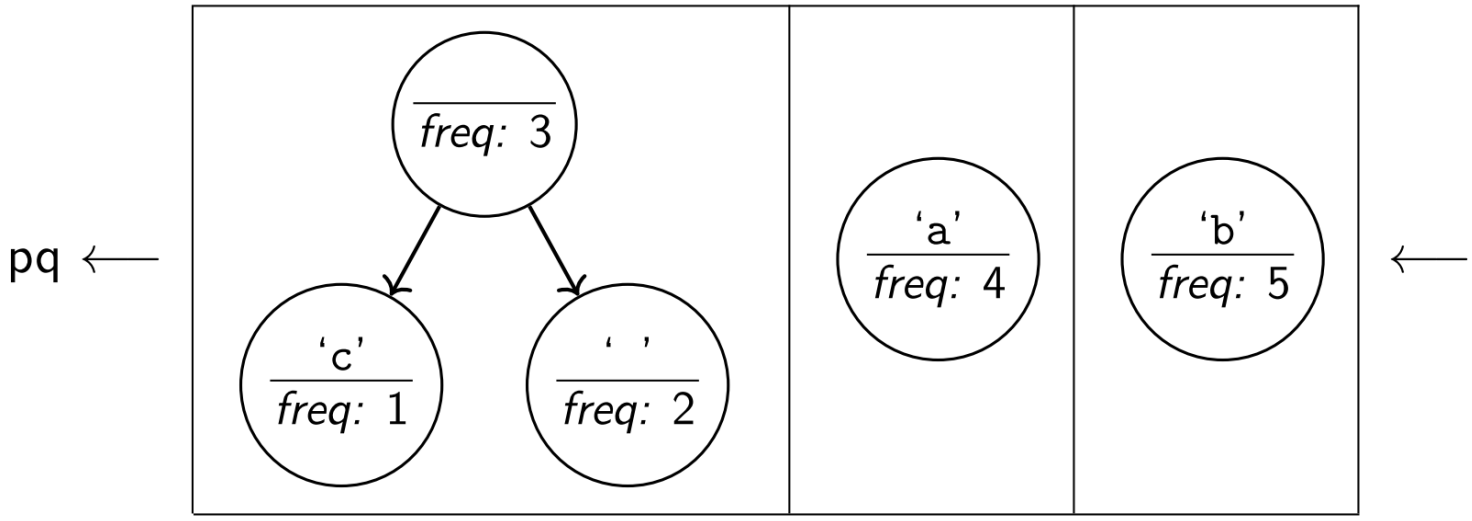
Step 3b: Combine Them Together

Then, create a new node. The frequency is the sum of two nodes:

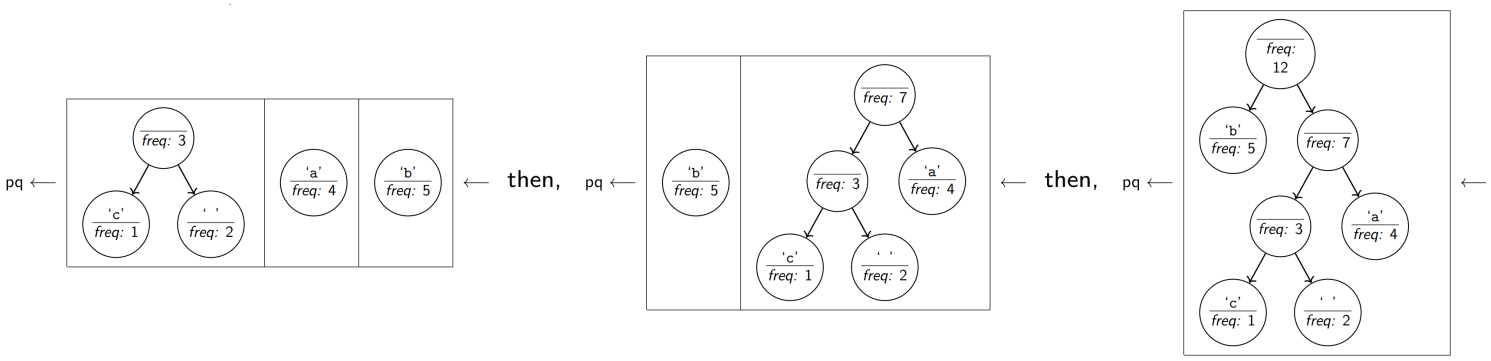


Step 3c: Add Back To Priority Queue

Now that we have a "new node", add it back to the priority queue:



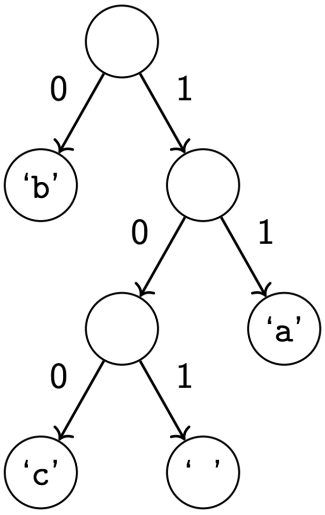
We repeat this process until there is only one node left in the priority queue. Here are the remaining steps (each time, we remove the two smallest frequencies, combine them, and put the result back):



Now that we only have one node left, we can use the tree we constructed to create the Huffman codes!

Step 4: Read Off the Huffman Codes

At this point, the frequencies of the letters have already been taken into account; so, we no longer even look at them. Thus, the tree we've constructed looks like the following:



simple-spec-example.code

```
98
0
99
100
32
101
97
11
```

To figure out the Huffman code for a letter, we traverse the tree from the root to the node with the letter in it. When we go left, we print a 0 and if we go right, we print a 1. For example, `'c'` would be 100, because we go right, then left, then left to reach it.

Just like in BrettFeed, we will output the tree in "standard format". Notice that the only actual information is in the leaves of the tree. So, the code file, which you can get by asking the main to "make a code", will consist of line pairs: the first line will be the ASCII value of the character in the leaf and the second line will be the Huffman code for that character. For example, the output of the tree we just constructed would look like the above. The leaves should appear in the order of a **pre-order** traversal.

[OLD] Huffman Compression and Decompression

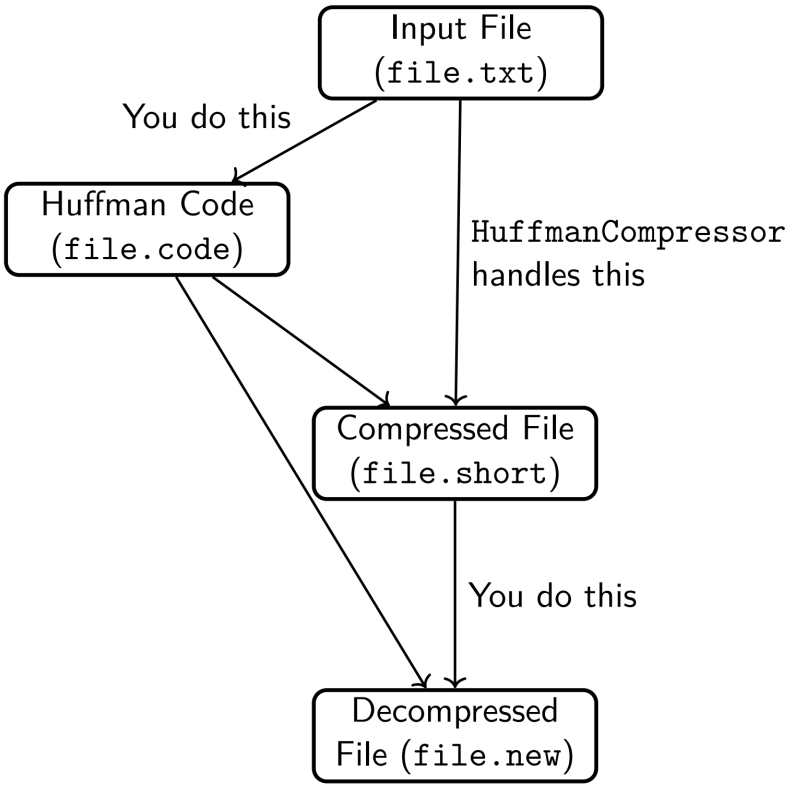
Now that we know how to construct a Huffman code, we are ready to understand the Huffman compression and decompression algorithms. Here is an overview of how they work:

Compression Algorithm

1. `HuffmanCompressor` generates the frequencies in a given file.
2. `HuffmanCode` creates the Huffman code from the frequencies.
3. `HuffmanCode` writes out the Huffman codes to a `.code` file.
4. `HuffmanCompressor` writes a `.short` file.

Decompression Algorithm

1. `HuffmanCode` reads in a `.code` file in standard format.
2. `HuffmanCode` writes out a `.new` file.



Client Program

`HuffmanCompressor` is a client program that we will provide to you. In addition to handling user interactions, it implements some of the steps of the compression and decompression algorithms that

we are not asking you to deal with. Specifically, `HuffmanCompressor` handles:

- computing character frequencies for a given input file (these frequencies are passed to the first `HuffmanCode` constructor above)
- compressing a text file using a given Huffman tree
- producing a `BitInputStream` from a given compressed input file (this stream is passed to the translate method)

You do not need to implement any of the above behavior. You **only** need to implement the behavior and methods described in the Specification slide. You may click "Expand" below to see a sample execution of each step (user input is **bold and underlined**).

Making a Huffman Code (produces .code file)

▶ Expand

Compressing a File (produces .short file)

▶ Expand

Decompressing a File (produces .new file)

▶ Expand

Compressing and then Decompressing a File (produces .short and .new file)

▶ Expand

[OLD] Implementation Guidelines

Your program should exactly reproduce the format and general behavior demonstrated in the Ed tests. Note that this assignment has two mostly separate parts: creating a Huffman code and decompressing a message using your Huffman code. Of the four methods to implement, two are relevant to each part.

1⃣ Creating a Huffman Code

You will write methods to (1) create a Huffman code from an array of frequencies and (2) write out the code you've created in standard format.

Frequency Array Constructor

You should use the algorithm described in the "Building a Huffman Code" slide to implement this constructor. You will need to use `PriorityQueue<E>` in the java.util package.

The only difference between a priority queue and a standard queue is that it uses the natural ordering of the objects to decide which object to dequeue first, with objects considered "less" returned first. You will be putting subtrees into your priority queue, which means you'll be adding values of type `HuffmanNode`.

This means that your `HuffmanNode` class will have to implement the `Comparable<E>` interface. It should use the frequency of the subtree to determine its ordering relative to other subtrees, with lower frequencies considered "less" than higher frequencies. If two frequencies are equal, the nodes are too.

Remember that, in order to make our code more flexible we should be declaring variables with their interface types when possible. This means you should define your `PriorityQueue` variables with the `Queue` interface.

The Huffman solution is not unique. You can obtain any one of several different equivalent trees depending upon how certain decisions are made. However, if you implement it as we have specified, then you should get exactly the same tree for any particular implementation of `PriorityQueue`. Make sure that you use the built-in `PriorityQueue` class and that when you are combining pairs of values taken from the priority queue, you make the first value removed from the queue the left subtree and you make the second value removed the right subtree.

2⃣ Decompressing A Message

You will write methods to (1) read in a `.code` file created with `save()` and (2) translate a compressed

file back into a decompressed file.

`Scanner` Constructor

This constructor will be given a `Scanner` that contains data produced by `save()`. In other words, the input for this constructor is the output you produced into a `.code` file. The goal of this constructor is to re-create the Huffman tree from your output. Note that the frequencies are irrelevant for this part, because the tree has already been constructed; so, you should set all the frequencies to some standard value (such as 0 or -1) when creating `HuffmanNodes` in this constructor.

Remember that the standard code file format is a series of pairs of lines where the first line has an integer representing the character's ASCII value and the second line has the code to use for that character. You might be tempted to call `nextInt()` to read the integer and `nextLine()` to read the code, but remember that mixing token-based reading and line-based reading is not simple. Here is an alternative that uses a method called `parseInt` in the `Integer` class that allows you to use two successive calls on `nextLine()`:

```
int asciiValue = Integer.parseInt(input.nextLine());
String code = input.nextLine();
```

Take a look at the Huffman Encoding in-class slides for a walk-through of this algorithm (particularly starting at page 55).

`translate`

This method takes in a `BitInputStream` representing a previously compressed message and outputs the original decompressed message. `BitInputStream` can be used in a very similar way to a `Scanner`; see the Specification slide.

This method reads sequences of bits that represent encoded characters to figure out what the original characters must have been. The algorithm works as follows:

- Begin at the top of the tree
- For each bit read in from the `BitInputStream`, if it's a 0, go left, and if it's a 1, go right.
- Eventually, we will hit a leaf node. Once we do, write out the integer code for that character to the output using the following `PrintStream` method:

  ```
  public void write(int b)
  ```

- Then, go back to the top of the tree, and do the process over again.

You will have to be careful if you use recursion in your decode method. Java has a limit on the stack depth you can use. For a large message, there are hundreds of thousands of characters to decode. It would not be appropriate to write code that requires a stack that is hundreds of thousands of levels deep. You might be forced to write some or all of this using loops to make sure that you don't exceed

the stack depth.

Translate Example

▶ Expand

Code Quality

As always, your code should follow all guidelines in the Code Quality Guide and Commenting Guide. In particular, pay attention to these requirements:

- **`x = change(x)`:**
  - An important concept introduced in lecture was called x = change(x). This idea is related to proper design of recursive methods that manipulate the structure of a binary tree. You should follow this pattern where necessary when modifying your trees.
- **Avoid redundancy:**
  - Create "helper" method(s) to capture repeated code. As long as all extra methods you create are private (so outside code cannot call them), you can have additional methods in your class beyond those specified here.
  - If you find that multiple methods in your class do similar things, you should create helper method(s) to capture the common code.
- **Data Fields:**
  - Properly encapsulate your objects by making data fields in your `HuffmanCode` class private. (Fields in your `HuffmanNode` class should be public following the pattern from class.)
  - Avoid unnecessary fields; use fields to store important data of your objects but not to store temporary values only used in one place.
  - Fields should always be initialized inside a constructor or method, never at declaration.