# Pre-Class Work 16: Huffman

## Data Compression [Background Reading]

## Motivation

We are living in an exponentially growing digital age! It begs the question of how we can efficiently store and process the endless amounts of data being generated.

This is where data compression can help us. The goal of data compression is to represent the same information but reduce the size the data takes up, thereby making it more efficient to store, transmit, and process the data. There are two main types of data compression: lossy and lossless.

## 1️⃣ Lossy Compression

Lossy compression is a type of data compression where some of the original data is lost during the compression process, but the remaining data is still useful. Lossy compression is achieved by using algorithms that remove less important data.

For example, in image compression algorithms, similar pixels might be treated as the same, causing the resulting image to look close to the original but reduce the amount of space it takes up. Lossy compression is typically used for data that does not need to be preserved exactly, such as images, audio, and video. A common example is JPEG image compression, which often results in blocky or fuzzy looking images.

## 2️⃣ Lossless Compression

Lossless compression is a type of data compression where the original data can be exactly reconstructed from the compressed data. This is achieved by using algorithms that eliminate redundancy in the data, such as **Huffman coding**.

Lossless compression is typically used for data that needs to be preserved exactly, such as text, financial data, and program files.

## Main Points

- We are living in a digital age so we need to efficiently store and process endless amounts of data.
- *Data compression* allows us to represent the same information but reduce the size the data takes up.
- *Lossy compression* involves using algorithms to remove less important data.
    - Some of the original data is lost, but the remaining data is useful.
- *Lossless compression* allows us to compress data and then exactly reconstruct the original data.
    - We use algorithms to eliminate redundancy in the code, such as via **Huffman coding**.

# Huffman Coding [Background Reading]

## Background

ASCII is a character encoding standard that represents characters using 7 bits, allowing for a total of 128 different characters to be represented. ASCII was developed in the 1960s and became widely used in computer systems, communication protocols, and other areas. Eventually, 128 characters wasn't enough so Extended ASCII was created. Extended ASCII extends the original 7-bit ASCII character set to use 8 bits by adding 128 additional characters (128-255) to the original ASCII character set (0-127). T

Thus, in simple text files, each character is represented with 8 bits. If we wanted to compress a text file, how would we achieve that? What if different characters are represented by different numbers of bits instead of the standard 8 bits?

**Huffman coding** is a lossless data compression algorithm that was developed by David A. Huffman in 1952 while he was a graduate student at MIT. The basic idea behind **Huffman coding** is to represent each symbol in a message using a variable-length code, where more frequently occurring symbols are assigned shorter codes and less frequently occurring symbols are assigned longer codes. This results in a more compact representation of the message, as shorter codes are used more frequently.

## Huffman Tree

In your next assignment, you will be creating a **Huffman Tree**. Essentially, **Huffman Trees** are just binary trees which allows us to implement a relatively simple form of file compression. To create your **Huffman Trees**, you will use the **Huffman coding algorithm** which works by constructing a binary tree based on the frequency of each symbol in the message.

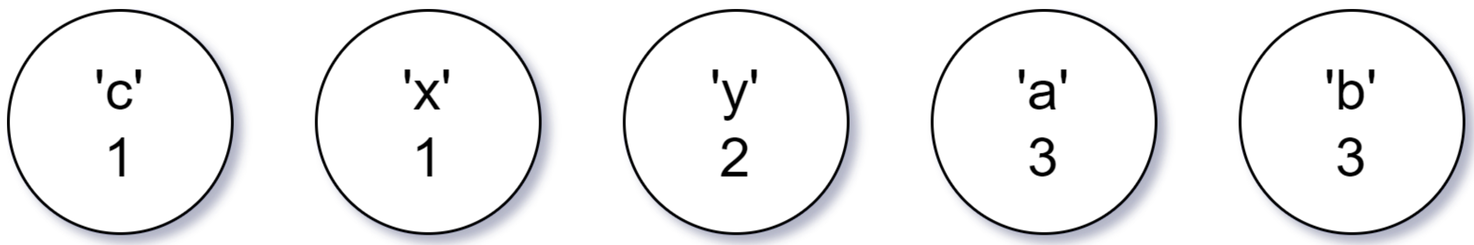
The symbols with the lowest frequency are assigned to the leaves of the tree, and the symbols with the highest frequency are assigned to the root of the tree. The process of constructing the binary tree involves repeatedly merging the two symbols with the lowest frequency into a new node, until all nodes are combined into a single tree.

Once the binary tree is constructed, the code for each symbol is determined by traversing the tree from the root to the leaf node that corresponds to the symbol. The resulting code is a sequence of 0s and 1s, with the code for each symbol being a unique prefix of its corresponding code in the binary tree.

This property allows for efficient decoding of the compressed message, as the decoder can simply follow the code through the binary tree to determine the original symbol.

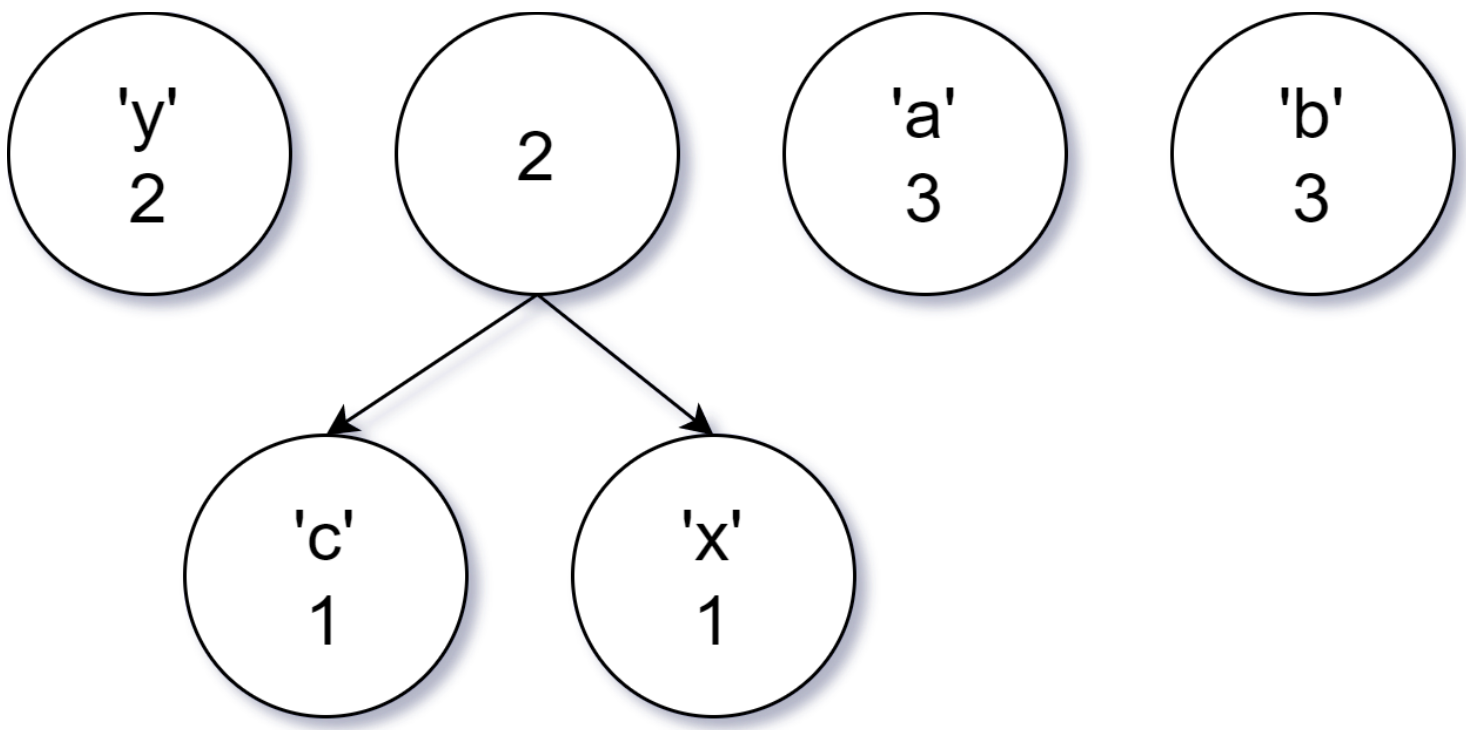
We will briefly discuss how a **Huffman Tree** is created. Note that the assignment specification will offer a much more detailed explanation.

Consider a file has 3 'a's, 3 'b's, 1 'c', 1 'x', and 2 'y's. Then, we would have the following nodes:

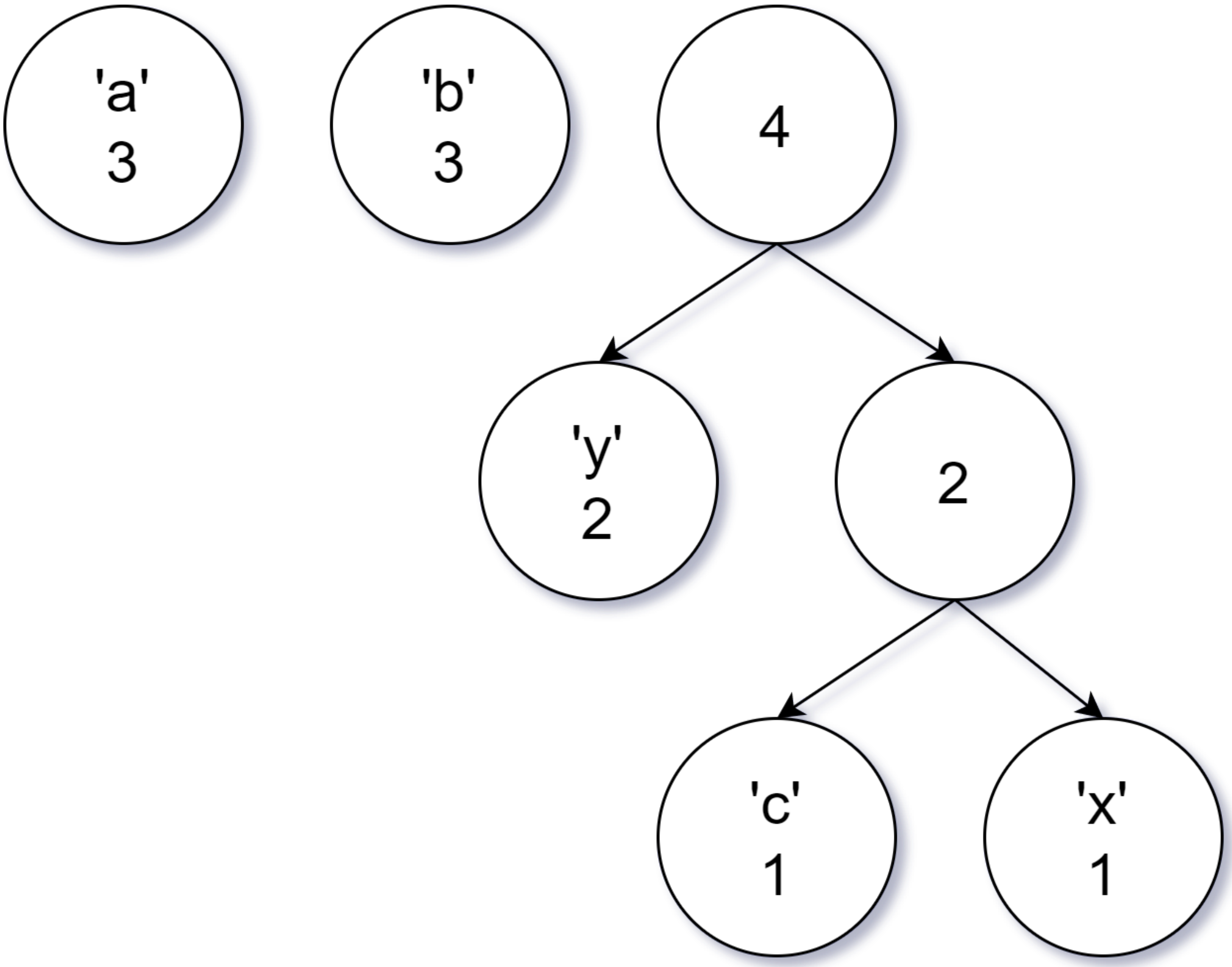


The next steps will visualize the **Huffman coding algorithm** where the nodes with lowest frequency are merged together until all nodes are combined together:

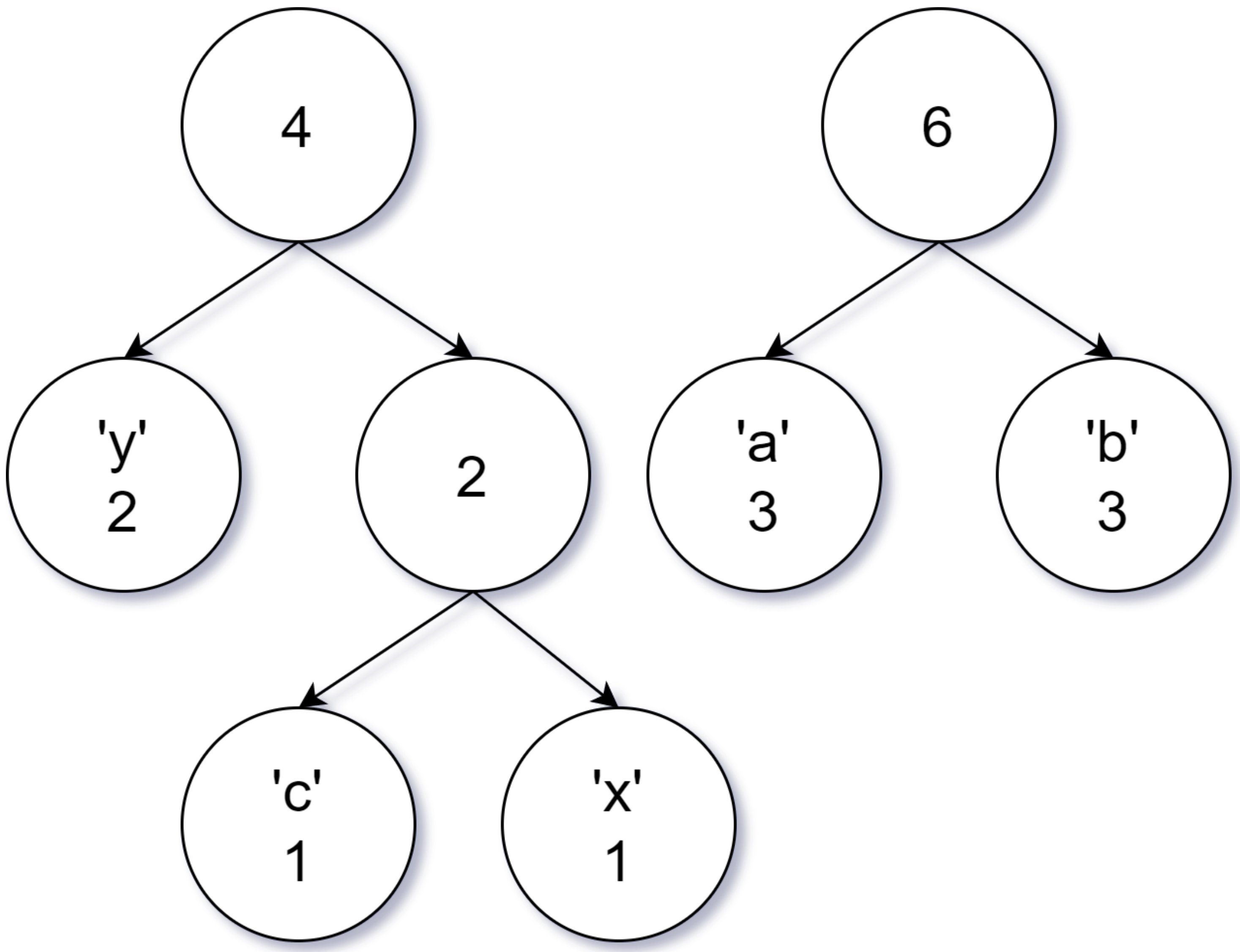
## 1▯ First merge



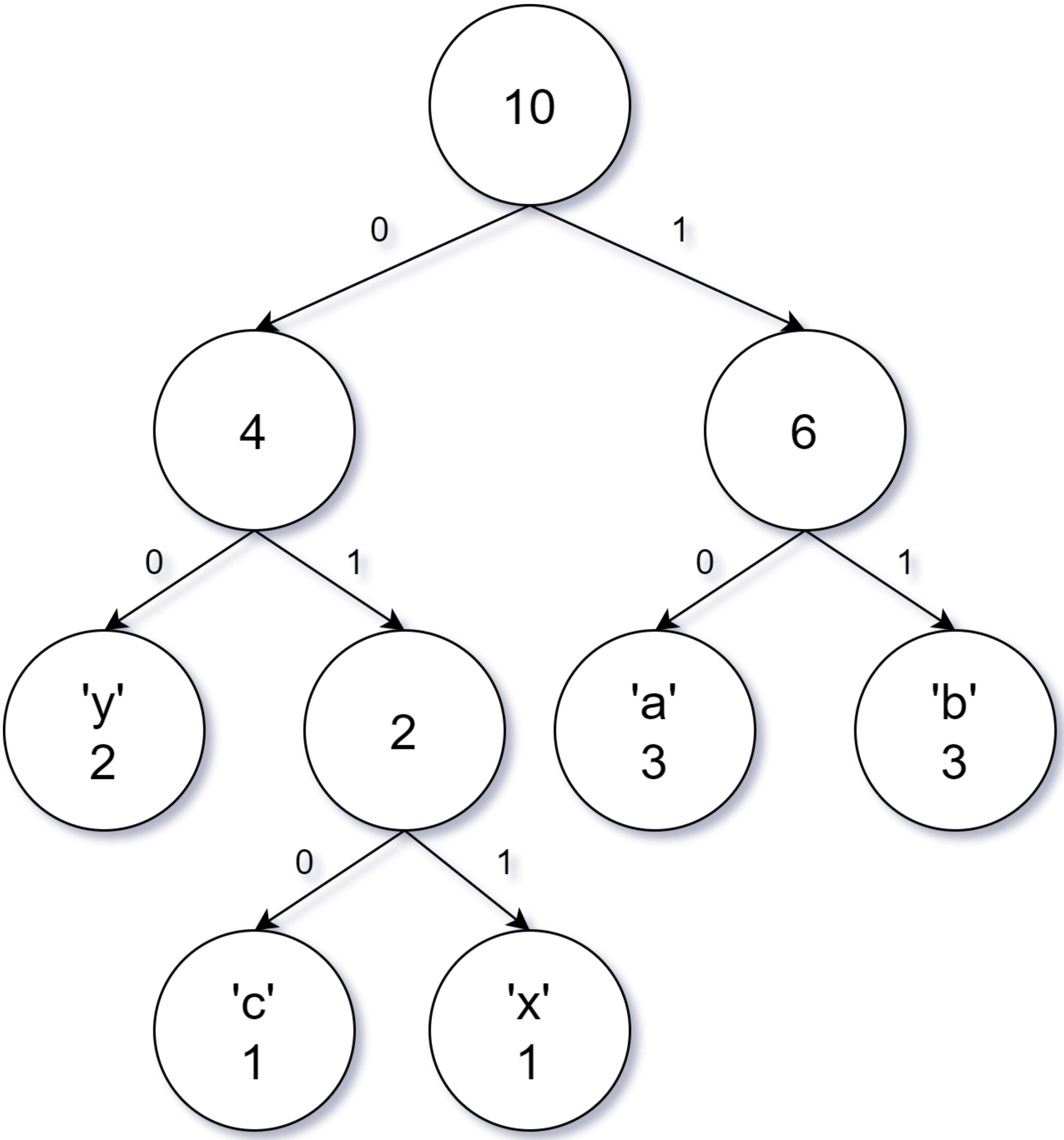
## 2▯ Second merge



3️⃣ Third merge



4 Fourth merge (The final Huffman Tree)



We can see that the top node that says `10` has the node that says `4` to the left of it with a line that has a label `0`. We can also see that the top node that says `10` has the node that says 6 to the left of it with a line that has a label `1`.

Every node that is less than that of the previous node is to the bottom left of that node connected by a line that says `0`, and every node that is more than that of the previous node is to the bottom right of that node connected by a line that says `1`.

Based on the **Huffman Tree** the following character encodings are as follows:

- 00 is the character encoding for ‘y’
- 010 is the character encoding for ‘c’
- 011 is the character encoding for ‘x’
- 10 is the character encoding for ‘a’
- 11 is the character encoding for ‘b’

Notice how each character doesn't utilize the full 8 bits to be encoded. 'y' needs 2 bits, 'c' needs 3 bits, 'x' needs 3 bits, 'a' needs 2 bits, and 'b' needs 2 bits. This is the power of the **Huffman coding algorithm**!

## Main Points

- **Huffman coding** is a *lossless* data compression algorithm that represents symbols in a message using variable-length codes.
    - For efficient compression, we assign shorter codes to more frequently occurring symbols and longer codes to less frequent symbols.
- In **Huffman coding**, we create a **Huffman tree**, which is a binary tree where we assign symbols with lower frequencies to leaves and symbols with higher frequencies to the root.
    - We merge nodes with the lowest frequencies!
- We start out with individual nodes for each symbol and include their frequencies. Then we keep merging nodes with the lowest frequencies until all nodes are connected and we form one single tree.
- We determine the code for each symbol by traversing the Huffman Tree from the root to the leaf. The code will be a bunch of 0s and 1s, with shorter codes for more frequent symbols.

## Priority Queues [Background Reading]

To implement a **Huffman coding algorithm**, you will need to familiarize yourself with a new data structure.

A **priority queue** is a data structure that stores a collection of elements and allows them to be retrieved in a specific order based on their priority. Thus, elements in a priority queue *must* implement the `Comparable` interface. The order of elements in a priority queue can be either ascending or descending, depending on the application.

To use a **priority queue**, you will need to import from `java.util.*.`

## Constructing a Priority Queue

To create a **priority queue** in Java, you can use the following syntax:

```java
Queue<Integer> pq = new PriorityQueue<>();
```

## Adding to a Priority Queue

To add to a **priority queue**, you can use the `add()` method:

```java
Queue<Integer> pq = new PriorityQueue<>();
pq.add(5);
pq.add(3);
pq.add(1);
```

## Removing from a Priority Queue

To remove from a **priority queue**, you can use the `remove()` method:

```java
Queue<Integer> pq = new PriorityQueue<>();
pq.add(5);
pq.add(3);
pq.add(1);
System.out.println(pq.remove()); // 1
System.out.println(pq.remove()); // 3
System.out.println(pq.remove()); // 5
```

## □ Main Points

- A **priority queue** is a data structure that stores a collection of elements so we can get them in a specific order based on their priority.
- Elements in a **priority queue** *must* implement the `Comparable` interface.
- We make a **priority queue** using this syntax: `Queue<Integer> pq = new PriorityQueue<>();`
- We add to a **priority queue** using this syntax: `pq.add(1);`
- We remove from a **priority queue** using this syntax: `pq.remove();`