Priority Queues and Huffman Encoding

Introduction to Homework 7

Hunter Schafer

Paul G. Allen School of Computer Science - CSE 143
ER Scheduling. How do we *efficiently* chose the most urgent case to treat next? Patients with more serious ailments should go first.

OS Context Switching. How does your operating system decide which process to give resources to? Some applications are more important than others.
I Think You Have Some Priority Issues

**ER Scheduling.** How do we *efficiently* chose the most urgent case to treat next? Patients with more serious ailments should go first.

**OS Context Switching.** How does your operating system decide which process to give resources to? Some applications are more important than others.

How can we solve these problems with the data structures we know?
Possible Solution

• Store elements in an unsorted list
  • add: Add at end
  • remove: Search for highest priority element

• Store elements in a sorted LinkedList
  • add: Search for position to insert, place there
  • remove: Remove from front

• Store elements in a TreeSet (hope they are unique!)
  • add: Traverse tree for position to insert, place there
  • remove: Traverse tree for smallest element, remove
Possible Solution

- Store elements in an unsorted list
  - add: Add at end
  - remove: Search for highest priority element

- Store elements in a sorted LinkedList
  - add: Search for position to insert, place there
  - remove: remove from front

- Store elements in a TreeSet (hope they are unique!)
  - add: Traverse tree for position to insert, place there
  - remove: Traverse tree for smallest element, remove
Priority Queue

A collection of ordered elements that provides fast access to the minimum (or maximum) element.

```java
public class PriorityQueue<E> implements Queue<E>

PriorityQueue<E>() constructs an empty queue
add(E value) adds value in sorted order to the queue
peek() returns minimum element in queue
remove() removes/returns minimum element in queue
size() returns the number of elements in queue
```

```java
Queue<String> tas = new PriorityQueue<String>();
tas.add("Jin");
tas.add("Aaron");
tas.remove();
```
Priority Queue

A collection of ordered elements that provides fast access to the minimum (or maximum) element.

public class PriorityQueue<E> implements Queue<E>

PriorityQueue<E>() constructs an empty queue
add(E value) adds value in sorted order to the queue
peek() returns minimum element in queue
remove() removes/returns minimum element in queue
size() returns the number of elements in queue

Queue<String> tas = new PriorityQueue<String>();
tas.add("Jin");
tas.add("Aaron");
tas.remove(); // "Aaron"
What does this code print?

```java
Queue<TA> tas = new PriorityQueue<TA>();
tas.add(new TA("Kyle", 7));
tas.add(new TA("Ayaz", 3));
tas.add(new TA("Zach", 6));
System.out.println(tas);
```

Prints:

```
```
What does this code print?

```java
Queue<TA> tas = new PriorityQueue<TA>();
tas.add(new TA("Kyle", 7));
tas.add(new TA("Ayaz", 3));
tas.add(new TA("Zach", 6));
System.out.println(tas);
```


**Common Gotchas**

- Elements must be Comparable.
- `toString` doesn’t do what you expect! Use `remove` instead.
Inside the Priority Queue

- Usually implemented with a heap
- Guarantees children have a lower priority than the parent so the highest priority is at the root (fast access).
- Take CSE 332 or CSE 373 to learn about how to implement more complicated data structures like heaps!
Homework 7: Huffman Coding
Compression

Process of encoding information so that it takes up less space.

Compression applies to many things!

- Store photos without taking up the whole hard-drive
- Reduce size of email attachment
- Make web pages smaller so they load faster
- Make voice calls over a low-bandwidth connection (cell, Skype)

Common compression programs:

- WinZip, WinRar for Windows
- zip
ASCII (American Standard Code for Information Interchange)

Standardized code for mapping characters to integers

We need to represent characters in binary so computers can read them.

- Most text files on your computer are in ASCII.

Every character is represented by a byte (8 bits).

<table>
<thead>
<tr>
<th>Character</th>
<th>ASCII value</th>
<th>Binary Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>‘ ‘</td>
<td>32</td>
<td>00100000</td>
</tr>
<tr>
<td>‘a’</td>
<td>97</td>
<td>01100001</td>
</tr>
<tr>
<td>‘b’</td>
<td>98</td>
<td>01100010</td>
</tr>
<tr>
<td>‘c’</td>
<td>99</td>
<td>01100011</td>
</tr>
<tr>
<td>‘e’</td>
<td>101</td>
<td>01100101</td>
</tr>
<tr>
<td>‘z’</td>
<td>122</td>
<td>01111010</td>
</tr>
</tbody>
</table>
What is the binary representation of the following String?
cab z

<table>
<thead>
<tr>
<th>Character</th>
<th>ASCII value</th>
<th>Binary Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>‘ ‘</td>
<td>32</td>
<td>00100000</td>
</tr>
<tr>
<td>‘a’</td>
<td>97</td>
<td>01100001</td>
</tr>
<tr>
<td>‘b’</td>
<td>98</td>
<td>01100010</td>
</tr>
<tr>
<td>‘c’</td>
<td>99</td>
<td>01100011</td>
</tr>
<tr>
<td>‘e’</td>
<td>101</td>
<td>01100101</td>
</tr>
<tr>
<td>‘z’</td>
<td>122</td>
<td>01111010</td>
</tr>
</tbody>
</table>
## ASCII Example

<table>
<thead>
<tr>
<th>Character</th>
<th>ASCII value</th>
<th>Binary Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>‘ ‘</td>
<td>32</td>
<td>00100000</td>
</tr>
<tr>
<td>‘a’</td>
<td>97</td>
<td>01100001</td>
</tr>
<tr>
<td>‘b’</td>
<td>98</td>
<td>01100010</td>
</tr>
<tr>
<td>‘c’</td>
<td>99</td>
<td>01100011</td>
</tr>
<tr>
<td>‘e’</td>
<td>101</td>
<td>01100101</td>
</tr>
<tr>
<td>‘z’</td>
<td>122</td>
<td>01111010</td>
</tr>
</tbody>
</table>

What is the binary representation of the following String?

`cab z`

**Answer**

01100011
### ASCII Example

<table>
<thead>
<tr>
<th>Character</th>
<th>ASCII value</th>
<th>Binary Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>' '</td>
<td>32</td>
<td>00100000</td>
</tr>
<tr>
<td>'a'</td>
<td>97</td>
<td>01100001</td>
</tr>
<tr>
<td>'b'</td>
<td>98</td>
<td>01100010</td>
</tr>
<tr>
<td>'c'</td>
<td>99</td>
<td>01100011</td>
</tr>
<tr>
<td>'e'</td>
<td>101</td>
<td>01100101</td>
</tr>
<tr>
<td>'z'</td>
<td>122</td>
<td>01111010</td>
</tr>
</tbody>
</table>

What is the binary representation of the following String?

```
cab z
```

**Answer**

01100011 01100001
ASCII Example

<table>
<thead>
<tr>
<th>Character</th>
<th>ASCII value</th>
<th>Binary Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>‘ ’</td>
<td>32</td>
<td>00100000</td>
</tr>
<tr>
<td>‘a’</td>
<td>97</td>
<td>01100001</td>
</tr>
<tr>
<td>‘b’</td>
<td>98</td>
<td>01100010</td>
</tr>
<tr>
<td>‘c’</td>
<td>99</td>
<td>01100011</td>
</tr>
<tr>
<td>‘e’</td>
<td>101</td>
<td>01100101</td>
</tr>
<tr>
<td>‘z’</td>
<td>122</td>
<td>01111010</td>
</tr>
</tbody>
</table>

What is the binary representation of the following String?

cab z

Answer

01100011 01100001 01100010
### ASCII Example

<table>
<thead>
<tr>
<th>Character</th>
<th>ASCII value</th>
<th>Binary Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>' '</td>
<td>32</td>
<td>00100000</td>
</tr>
<tr>
<td>'a'</td>
<td>97</td>
<td>01100011</td>
</tr>
<tr>
<td>'b'</td>
<td>98</td>
<td>01100010</td>
</tr>
<tr>
<td>'c'</td>
<td>99</td>
<td>01100011</td>
</tr>
<tr>
<td>'e'</td>
<td>101</td>
<td>01100101</td>
</tr>
<tr>
<td>'z'</td>
<td>122</td>
<td>01111010</td>
</tr>
</tbody>
</table>

What is the binary representation of the following string?

cab_z

**Answer**

01100011 01100001 01100010 00100000
## ASCII Example

<table>
<thead>
<tr>
<th>Character</th>
<th>ASCII value</th>
<th>Binary Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>‘ ’</td>
<td>32</td>
<td>00100000</td>
</tr>
<tr>
<td>‘a’</td>
<td>97</td>
<td>01100001</td>
</tr>
<tr>
<td>‘b’</td>
<td>98</td>
<td>01100010</td>
</tr>
<tr>
<td>‘c’</td>
<td>99</td>
<td>01100011</td>
</tr>
<tr>
<td>‘e’</td>
<td>101</td>
<td>01100101</td>
</tr>
<tr>
<td>‘z’</td>
<td>122</td>
<td>01111010</td>
</tr>
</tbody>
</table>

What is the binary representation of the following String?

cab z

**Answer**

01100011 01100011 01100010 00100000 01111010
What is the binary representation of the following String?
cab z

Answer
0110001101100001011000100010000001111010
### Another ASCII Example

<table>
<thead>
<tr>
<th>Character</th>
<th>ASCII value</th>
<th>Binary Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>‘ ‘</td>
<td>32</td>
<td>00100000</td>
</tr>
<tr>
<td>‘a’</td>
<td>97</td>
<td>01100001</td>
</tr>
<tr>
<td>‘b’</td>
<td>98</td>
<td>01100010</td>
</tr>
<tr>
<td>‘c’</td>
<td>99</td>
<td>01100011</td>
</tr>
<tr>
<td>‘e’</td>
<td>101</td>
<td>01100101</td>
</tr>
<tr>
<td>‘z’</td>
<td>122</td>
<td>01111010</td>
</tr>
</tbody>
</table>

**How do we read the following binary as ASCII?**

011000010110001101100101
## Another ASCII Example

<table>
<thead>
<tr>
<th>Character</th>
<th>ASCII value</th>
<th>Binary Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>‘ ‘</td>
<td>32</td>
<td>00100000</td>
</tr>
<tr>
<td>‘a’</td>
<td>97</td>
<td>01100001</td>
</tr>
<tr>
<td>‘b’</td>
<td>98</td>
<td>01100010</td>
</tr>
<tr>
<td>‘c’</td>
<td>99</td>
<td>01100011</td>
</tr>
<tr>
<td>‘e’</td>
<td>101</td>
<td>01100101</td>
</tr>
<tr>
<td>‘z’</td>
<td>122</td>
<td>01111010</td>
</tr>
</tbody>
</table>

### How do we read the following binary as ASCII?

01100001 01100011 01100101

### Answer
Another ASCII Example

<table>
<thead>
<tr>
<th>Character</th>
<th>ASCII value</th>
<th>Binary Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>‘ ‘</td>
<td>32</td>
<td>00100000</td>
</tr>
<tr>
<td>‘a’</td>
<td>97</td>
<td>01100001</td>
</tr>
<tr>
<td>‘b’</td>
<td>98</td>
<td>01100010</td>
</tr>
<tr>
<td>‘c’</td>
<td>99</td>
<td>01100011</td>
</tr>
<tr>
<td>‘e’</td>
<td>101</td>
<td>01100101</td>
</tr>
<tr>
<td>‘z’</td>
<td>122</td>
<td>01111010</td>
</tr>
</tbody>
</table>

How do we read the following binary as ASCII?

01100001 01100011 01100101

Answer

a
### Another ASCII Example

<table>
<thead>
<tr>
<th>Character</th>
<th>ASCII value</th>
<th>Binary Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>‘ ’</td>
<td>32</td>
<td>00100000</td>
</tr>
<tr>
<td>‘a’</td>
<td>97</td>
<td>01100001</td>
</tr>
<tr>
<td>‘b’</td>
<td>98</td>
<td>01100010</td>
</tr>
<tr>
<td>‘c’</td>
<td>99</td>
<td>01100011</td>
</tr>
<tr>
<td>‘e’</td>
<td>101</td>
<td>01100101</td>
</tr>
<tr>
<td>‘z’</td>
<td>122</td>
<td>01111010</td>
</tr>
</tbody>
</table>

How do we read the following binary as ASCII?
01100001 01100011 01100101

**Answer**
ac
### Another ASCII Example

<table>
<thead>
<tr>
<th>Character</th>
<th>ASCII value</th>
<th>Binary Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>‘ ’</td>
<td>32</td>
<td>00100000</td>
</tr>
<tr>
<td>‘a’</td>
<td>97</td>
<td>01100001</td>
</tr>
<tr>
<td>‘b’</td>
<td>98</td>
<td>01100010</td>
</tr>
<tr>
<td>‘c’</td>
<td>99</td>
<td>01100011</td>
</tr>
<tr>
<td>‘e’</td>
<td>101</td>
<td>01100101</td>
</tr>
<tr>
<td>‘z’</td>
<td>122</td>
<td>01111010</td>
</tr>
</tbody>
</table>

**How do we read the following binary as ASCII?**

01100001 01100011 01100101

**Answer**

ace
Huffman Idea

**Huffman’s Insight**

Use variable length encodings for different characters to take advantage of frequencies in which characters appear.

- Make more frequent characters take up less space.
- Don’t have codes for unused characters.
- Some characters may end up with longer encodings, but this should happen infrequently.
Huffman Encoding

- Create a “Huffman Tree” that gives a good binary representation for each character.
- The path from the root to the character leaf is the encoding for that character; left means 0, right means 1.

**ASCII Table**

<table>
<thead>
<tr>
<th>Character</th>
<th>Binary Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>' '</td>
<td>00100000</td>
</tr>
<tr>
<td>'a'</td>
<td>01100001</td>
</tr>
<tr>
<td>'b'</td>
<td>01100010</td>
</tr>
<tr>
<td>'c'</td>
<td>01100011</td>
</tr>
<tr>
<td>'e'</td>
<td>01100101</td>
</tr>
<tr>
<td>'z'</td>
<td>01111010</td>
</tr>
</tbody>
</table>

**Huffman Tree**
Homework 7 asks you to write a class that manages creating and using this Huffman code.

(A) Create a Huffman Code from a file and compress it.
(B) Decompress the file to get original contents.
Part A: Making a HuffmanCode Overview

**Input File Contents**

bad  cab

---

**Step 1:** Count the occurrences of each character in file

- ' ' = 1
- 'a' = 2
- 'b' = 2
- 'c' = 1
- 'd' = 1

**Step 2:** Make leaf nodes for all the characters put them in a PriorityQueue

- ' ' : freq = 1
- 'c' : freq = 1
- 'd' : freq = 1
- 'a' : freq = 2
- 'b' : freq = 2

**Step 3:** Use Huffman Tree building algorithm (described in a couple slides)

**Step 4:** Save encoding to `.code` file to encode/decode later.

- 'd' = 00
- 'a' = 01
- 'b' = 10
- ' ' = 110
- 'c' = 111

**Step 5:** Compress the input file using the encodings

Compressed Output: 1001001101110110
Part A: Making a Huffman Code Overview

**Input File Contents**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>bad</code></td>
<td><code>cab</code></td>
</tr>
</tbody>
</table>

**Step 1:** Count the occurrences of each character in file

\[
\{ ' ' = 1, 'a' = 2, 'b' = 2, 'c' = 1, 'd' = 1 \}
\]
### Part A: Making a Huffman Code Overview

**Input File Contents**

bad  cab

**Step 1:** Count the occurrences of each character in file

```
{ ' ' = 1, 'a' = 2, 'b' = 2, 'c' = 1, 'd' = 1 }
```

**Step 2:** Make leaf nodes for all the characters put them in a PriorityQueue

```
pq ←−
```

<table>
<thead>
<tr>
<th>Character</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>' '</td>
<td>1</td>
</tr>
<tr>
<td>'c'</td>
<td>1</td>
</tr>
<tr>
<td>'d'</td>
<td>1</td>
</tr>
<tr>
<td>'a'</td>
<td>2</td>
</tr>
<tr>
<td>'b'</td>
<td>2</td>
</tr>
</tbody>
</table>

**Step 3:** Use Huffman Tree building algorithm (described in a couple slides)

**Step 4:** Save encoding to `.code` file to encode/decode later.

```
{'d'=00, 'a'=01, 'b'=10, ' '=110, 'c'=111}
```

**Step 5:** Compress the input file using the encodings

Compressed Output: 1001001101110110
**Part A: Making a Huffman Code Overview**

**Input File Contents**

bad  cab

**Step 1:** Count the occurrences of each character in file

{ ‘ ’=1, ‘a’=2, ‘b’=2, ‘c’=1, ‘d’=1 }

**Step 2:** Make leaf nodes for all the characters put them in a PriorityQueue

\[
\begin{array}{c}
\text{pq} \leftarrow \\
\quad \text{‘ ’} \quad \text{freq: 1} \\
\quad \text{‘c’} \quad \text{freq: 1} \\
\quad \text{‘d’} \quad \text{freq: 1} \\
\quad \text{‘a’} \quad \text{freq: 2} \\
\quad \text{‘b’} \quad \text{freq: 2} \\
\end{array}
\]

**Step 3:** Use Huffman Tree building algorithm (described in a couple slides)

**Step 4:** Save encoding to `.code` file to encode/decode later.

{ ‘d’=00, ‘a’=01, ‘b’=10, ‘ ’=110, ‘c’=111 }

**Step 5:** Compress the input file using the encodings

Compressed Output: 1001001101110110
Part A: Making a HuffmanCode Overview

**Input File Contents**

bad  cab

**Step 1:** Count the occurrences of each character in file

{‘ ’=1, ‘a’=2, ‘b’=2, ‘c’=1, ‘d’=1}

**Step 2:** Make leaf nodes for all the characters put them in a PriorityQueue

\[
\text{pq} \leftarrow \begin{array}{c}
\text{‘ ’} \quad \text{freq: 1} \\
\text{‘c’} \quad \text{freq: 1} \\
\text{‘d’} \quad \text{freq: 1} \\
\text{‘a’} \quad \text{freq: 2} \\
\text{‘b’} \quad \text{freq: 2}
\end{array}
\]

**Step 3:** Use Huffman Tree building algorithm (described in a couple slides)

**Step 4:** Save encoding to .code file to encode/decode later.

{‘d’=00, ‘a’=01, ‘b’=10, ‘ ’=110, ‘c’=111}
Step 1: Count the occurrences of each character in file
\{‘ ’=1, ‘a’=2, ‘b’=2, ‘c’=1, ‘d’=1\}

Step 2: Make leaf nodes for all the characters put them in a PriorityQueue

\[
\begin{array}{c}
\text{pq} \leftarrow \begin{array}{c}
\text{‘ ’} & \text{freq: 1} \\
\text{‘c’} & \text{freq: 1} \\
\text{‘d’} & \text{freq: 1} \\
\text{‘a’} & \text{freq: 2} \\
\text{‘b’} & \text{freq: 2} \\
\end{array}
\end{array}
\]

Step 3: Use Huffman Tree building algorithm (described in a couple slides)

Step 4: Save encoding to .code file to encode/decode later.
\{‘d’=00, ‘a’=01, ‘b’=10, ‘ ’=110, ‘c’=111\}

Step 5: Compress the input file using the encodings
Compressed Output: 1001001101110110
### Step 1: Count Character Occurrences

We do this step for you

<table>
<thead>
<tr>
<th>Input File</th>
</tr>
</thead>
<tbody>
<tr>
<td>bad cab</td>
</tr>
</tbody>
</table>

### Generate Counts Array:

<table>
<thead>
<tr>
<th>index</th>
<th>0</th>
<th>1</th>
<th>...</th>
<th>32</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>0</td>
<td>0</td>
<td>...</td>
<td>1</td>
<td>...</td>
</tr>
</tbody>
</table>

| value | 2 | 2 | 1 | 1 | 0 |

This is super similar to LetterInventory but works for all characters!
Step 2: Create PriorityQueue

- Store each character and its frequency in a HuffmanNode object.
- Place all the HuffmanNodes in a PriorityQueue so that they are in ascending order with respect to frequency

\[
\begin{array}{|c|}
\hline
\text{Character} & \text{Frequency} \\
\hline
\text{\textquote{\textquote{}} } & 1 \\
\text{\textquote{c} } & 1 \\
\text{\textquote{d} } & 1 \\
\text{\textquote{a} } & 2 \\
\text{\textquote{b} } & 2 \\
\hline
\end{array}
\]
Step 3: Remove and Merge

pq ← ' ' freq: 1

'c' freq: 1

'd' freq: 1

'a' freq: 2

'b' freq: 2
Step 3: Remove and Merge

pq ←

\[
\begin{array}{c}
\text{freq: 2} \\
\text{\cdot \cdot} \\
\text{freq: 1} \\
\text{'c'} \\
\text{freq: 1} \\
\text{'+a'} \\
\text{freq: 2} \\
\text{+'b'} \\
\text{freq: 2}
\end{array}
\]
Step 3: Remove and Merge

\[
\begin{align*}
\text{pq} &\leftarrow \begin{array}{c}
\text{freq: 1} \\
\text{freq: 2} \\
\text{freq: 2} \\
\end{array}
\end{align*}
\]
Step 3: Remove and Merge

pq ←

freq: 3

'c'
freq: 1

'.'
freq: 1

'c'
freq: 1

'.'
freq: 1

'b'
freq: 2

'pq ←

'd'
freq: 1

'a'
freq: 2

freq: 2
Step 3: Remove and Merge

pq ←

\[
\begin{array}{c}
\text{‘b’} \\
\text{freq: 2}
\end{array}
\]

\[
\begin{array}{c}
\text{‘’} \\
\text{freq: 1}
\end{array}
\]

\[
\begin{array}{c}
\text{‘c’} \\
\text{freq: 1}
\end{array}
\]

\[
\begin{array}{c}
\text{‘d’} \\
\text{freq: 1}
\end{array}
\]

\[
\begin{array}{c}
\text{‘a’} \\
\text{freq: 2}
\end{array}
\]

\[
\begin{array}{c}
\text{freq: 3}
\end{array}
\]
Step 3: Remove and Merge

freq: 4

'b'
freq: 2

'c'
freq: 1

'a'
freq: 2

pq ←

freq: 3

'd'
freq: 1

'a'
freq: 2
Step 3: Remove and Merge

pq ← freq: 3

\[
\begin{align*}
'd' & \quad \text{freq: 1} \\
'a' & \quad \text{freq: 2}
\end{align*}
\]

freq: 4

\[
\begin{align*}
'b' & \quad \text{freq: 2} \\
'c' & \quad \text{freq: 1}
\end{align*}
\]
Step 3: Remove and Merge

freq: 7

freq: 3

freq: 4

‘d’
freq: 1

‘a’
freq: 2

‘b’
freq: 2

‘c’
freq: 1

pq ← □ ←
Step 3: Remove and Merge

freq: 7

freq: 3

freq: 4

'd' freq: 1

'a' freq: 2

'b' freq: 2

'c' freq: 1

pq ←− 16
Step 3: Remove and Merge

- What is the relationship between frequency in file and binary representation length?
Step 3: Remove and Merge Algorithm

Algorithm Pseudocode

```
while P.Q. size > 1:
    remove two nodes with lowest frequency
    combine into a single node
    put that node back in the P.Q.
```
Step 4: Print Encodings

Save the tree to a file to save the encodings for the characters we made.
Step 4: Print Encodings

Save the tree to a file to save the encodings for the characters we made.
Step 4: Print Encodings

Save the tree to a file to save the encodings for the characters we made.

Output of `save`

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>00</td>
<td></td>
</tr>
</tbody>
</table>
Step 4: Print Encodings

Save the tree to a file to save the encodings for the characters we made.

Output of save

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td></td>
</tr>
<tr>
<td>00</td>
<td></td>
</tr>
<tr>
<td>97</td>
<td></td>
</tr>
<tr>
<td>01</td>
<td></td>
</tr>
</tbody>
</table>
Step 4: Print Encodings

Save the tree to a file to save the encodings for the characters we made.

Output of `save`

100
00
97
01
98
10
Step 4: Print Encodings

Save the tree to a file to save the encodings for the characters we made.

Output of save

<table>
<thead>
<tr>
<th>100</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
</tr>
<tr>
<td>97</td>
</tr>
<tr>
<td>01</td>
</tr>
<tr>
<td>98</td>
</tr>
<tr>
<td>10</td>
</tr>
<tr>
<td>32</td>
</tr>
<tr>
<td>110</td>
</tr>
</tbody>
</table>
### Step 4: Print Encodings

Save the tree to a file to save the encodings for the characters we made.

![Diagram of a tree with encodings]

**Output of save**

- 100
- 00
- 97
- 01
- 98
- 10
- 32
- 110
- 99
- 111
**Step 5: Compress the File**

*We do this step for you*

Take the original file and the `.code` file produced in last step to translate into the new binary encoding.

<table>
<thead>
<tr>
<th>Input File</th>
<th>Huffman Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>bad cab</td>
<td>100 00 97 01 98 10 32 110 99 111</td>
</tr>
</tbody>
</table>

**Compressed Output**
Step 5: Compress the File

We do this step for you

Take the original file and the .code file produced in last step to translate into the new binary encoding.

Input File

<table>
<thead>
<tr>
<th>Input File</th>
</tr>
</thead>
<tbody>
<tr>
<td>bad cab</td>
</tr>
</tbody>
</table>

Compressed Output

<table>
<thead>
<tr>
<th>Huffman Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>100 'd'</td>
</tr>
<tr>
<td>00</td>
</tr>
<tr>
<td>97 'a'</td>
</tr>
<tr>
<td>01</td>
</tr>
<tr>
<td>98 'b'</td>
</tr>
<tr>
<td>10</td>
</tr>
<tr>
<td>32 ' '</td>
</tr>
<tr>
<td>110</td>
</tr>
<tr>
<td>99 'c'</td>
</tr>
<tr>
<td>111</td>
</tr>
</tbody>
</table>
Step 5: Compress the File

We do this step for you

Take the original file and the .code file produced in last step to translate into the new binary encoding.

<table>
<thead>
<tr>
<th>Input File</th>
<th>Huffman Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>bad cab</td>
<td>100  'd'</td>
</tr>
<tr>
<td></td>
<td>00</td>
</tr>
<tr>
<td></td>
<td>97  'a'</td>
</tr>
<tr>
<td></td>
<td>01</td>
</tr>
<tr>
<td></td>
<td>98  'b'</td>
</tr>
<tr>
<td></td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>32  ' '</td>
</tr>
<tr>
<td></td>
<td>110</td>
</tr>
<tr>
<td></td>
<td>99  'c'</td>
</tr>
<tr>
<td></td>
<td>111</td>
</tr>
</tbody>
</table>

Compressed Output

10 01 100 110 111 01 10
Step 5: Compress the File

We do this step for you

Take the original file and the .code file produced in last step to translate into the new binary encoding.

<table>
<thead>
<tr>
<th>Input File</th>
<th>Huffman Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>bad cab</td>
<td>100 'd'</td>
</tr>
<tr>
<td></td>
<td>00</td>
</tr>
<tr>
<td></td>
<td>97 'a'</td>
</tr>
<tr>
<td></td>
<td>01</td>
</tr>
<tr>
<td></td>
<td>98 'b'</td>
</tr>
<tr>
<td></td>
<td>10</td>
</tr>
</tbody>
</table>
|                  | 32 '
'           |
|                  | 110              |
|                  | 99 'c'           |
|                  | 111              |

<table>
<thead>
<tr>
<th>Compressed Output</th>
<th>Uncompressed Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>10 01 100 110 111 01 10</td>
<td>01100010 01100001 01100100 00100000 01100011 01100001 01100010</td>
</tr>
</tbody>
</table>
Part B: Decompressing the File

**Step 1:** Reconstruct the Huffman tree from the code file

**Step 2:** Translate the compressed bits back to their character values.
Step 1: Reconstruct the Huffman Tree

Now are just given the code file produced by our program and we need to reconstruct the tree.

<table>
<thead>
<tr>
<th>Input code File</th>
</tr>
</thead>
<tbody>
<tr>
<td>97</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>101</td>
</tr>
<tr>
<td>100</td>
</tr>
<tr>
<td>32</td>
</tr>
<tr>
<td>101</td>
</tr>
<tr>
<td>112</td>
</tr>
<tr>
<td>11</td>
</tr>
</tbody>
</table>

Initially the tree is empty
Step 1: Reconstruct the Huffman Tree

Now are just given the code file produced by our program and we need to reconstruct the tree.

<table>
<thead>
<tr>
<th>Input code File</th>
</tr>
</thead>
<tbody>
<tr>
<td>97</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>101</td>
</tr>
<tr>
<td>100</td>
</tr>
<tr>
<td>32</td>
</tr>
<tr>
<td>101</td>
</tr>
<tr>
<td>112</td>
</tr>
<tr>
<td>11</td>
</tr>
</tbody>
</table>

Tree after processing first pair
Step 1: Reconstruct the Huffman Tree

Now are just given the code file produced by our program and we need to reconstruct the tree.

<table>
<thead>
<tr>
<th>Input code File</th>
</tr>
</thead>
<tbody>
<tr>
<td>97</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>101</td>
</tr>
<tr>
<td>100</td>
</tr>
<tr>
<td>32</td>
</tr>
<tr>
<td>101</td>
</tr>
<tr>
<td>112</td>
</tr>
<tr>
<td>11</td>
</tr>
</tbody>
</table>

Tree after processing second pair
Step 1: Reconstruct the Huffman Tree

Now are just given the code file produced by our program and we need to reconstruct the tree.

<table>
<thead>
<tr>
<th>Input code File</th>
</tr>
</thead>
<tbody>
<tr>
<td>97</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>101</td>
</tr>
<tr>
<td>100</td>
</tr>
<tr>
<td>32</td>
</tr>
<tr>
<td>101</td>
</tr>
<tr>
<td>112</td>
</tr>
<tr>
<td>11</td>
</tr>
</tbody>
</table>

Tree after processing third pair
Step 1: Reconstruct the Huffman Tree

Now are just given the code file produced by our program and we need to reconstruct the tree.

<table>
<thead>
<tr>
<th>Input code File</th>
</tr>
</thead>
<tbody>
<tr>
<td>97</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>101</td>
</tr>
<tr>
<td>100</td>
</tr>
<tr>
<td>32</td>
</tr>
<tr>
<td>101</td>
</tr>
<tr>
<td>112</td>
</tr>
<tr>
<td>11</td>
</tr>
</tbody>
</table>

Tree after processing last pair
Step 2 Example

After building up tree, we will read the compressed file bit by bit.

**Input**

0101110110101011100

**Output**

```
0
0
1
0
1
'a'
0
1
'p'
0
1
'e'
''
```
After building up tree, we will read the compressed file bit by bit.

**Input**
0101110110101011100

**Output**
a papa ape
Reading bits in Java is kind of tricky, we are providing a class to help!

```java
public class BitInputStream

BitInputStream(String file)  // Creates a stream of bits from file

hasNextBit()  // Returns true if bits remain in the stream

nextBit()  // Reads and returns the next bit in the stream
```
Part A: Compression

```java
public HuffmanCode(int[] counts)
```
- Slides 15-17

```java
public void save(PrintStream out)
```
- Slide 18

Part B: Decompression

```java
public HuffmanCode(Scanner input)
```
- Slide 21

```java
public void translate(BitInputStream in, PrintStream out)
```
- Slide 22