# Priority Queues and Huffman Encoding

Introduction to Homework 8

Hunter Schafer

Paul G. Allen School of Computer Science - CSE 143

# Priority Queue

**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("Taylor");
tas.add("Andrew");
tas.remove();
```

# Priority Queue

**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 |

```java
Queue<String> tas = new PriorityQueue<String>();
tas.add("Taylor");
tas.add("Andrew");
tas.remove(); // "Andrew"
```

# Homework 8: Huffman Coding

# File Compression

**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

**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.

| Character | ASCII value |
|---|---|
| ‘ ’ | 32 |
| ‘a’ | 97 |
| ‘b’ | 98 |
| ‘c’ | 99 |
| ‘e’ | 101 |
| ‘z’ | 122 |

# ASCII

**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).

| Character | ASCII value | Binary Representation |
|---|---|---|
| ‘ ’ | 32 | 00100000 |
| ‘a’ | 97 | 01100001 |
| ‘b’ | 98 | 01100010 |
| ‘c’ | 99 | 01100011 |
| ‘e’ | 101 | 01100101 |
| ‘z’ | 122 | 01111010 |

# ASCII Example

| Character | ASCII value | Binary Representation |
|---|---|---|
| ‘ ’ | 32 | 00100000 |
| ‘a’ | 97 | 01100001 |
| ‘b’ | 98 | 01100010 |
| ‘c’ | 99 | 01100011 |
| ‘e’ | 101 | 01100101 |
| ‘z’ | 122 | 01111010 |

**What is the binary representation of the following String?**

cab z

# ASCII Example

| Character | ASCII value | Binary Representation |
|---|---|---|
| ‘ ’ | 32 | 00100000 |
| ‘a’ | 97 | 01100001 |
| ‘b’ | 98 | 01100010 |
| ‘c’ | 99 | 01100011 |
| ‘e’ | 101 | 01100101 |
| ‘z’ | 122 | 01111010 |

**What is the binary representation of the following String?**

cab z

**Answer**

01100011

# ASCII Example

| Character | ASCII value | Binary Representation |
|---|---|---|
| ‘ ’ | 32 | 00100000 |
| ‘a’ | 97 | 01100001 |
| ‘b’ | 98 | 01100010 |
| ‘c’ | 99 | 01100011 |
| ‘e’ | 101 | 01100101 |
| ‘z’ | 122 | 01111010 |

**What is the binary representation of the following String?**

ca̲b z

**Answer**

`01100011 01100001`

# ASCII Example

| Character | ASCII value | Binary Representation |
|---|---|---|
| ‘ ’ | 32 | 00100000 |
| ‘a’ | 97 | 01100001 |
| ‘b’ | 98 | 01100010 |
| ‘c’ | 99 | 01100011 |
| ‘e’ | 101 | 01100101 |
| ‘z’ | 122 | 01111010 |

**What is the binary representation of the following String?**

cab␣z

**Answer**

`01100011 01100001 01100010`

# ASCII Example

| Character | ASCII value | Binary Representation |
|---|---|---|
| ‘ ’ | 32 | 00100000 |
| ‘a’ | 97 | 01100001 |
| ‘b’ | 98 | 01100010 |
| ‘c’ | 99 | 01100011 |
| ‘e’ | 101 | 01100101 |
| ‘z’ | 122 | 01111010 |

**What is the binary representation of the following String?**

cab_z

**Answer**

`01100011 01100001 01100010 00100000`

# ASCII Example

| Character | ASCII value | Binary Representation |
|---|---|---|
| ‘ ’ | 32 | 00100000 |
| ‘a’ | 97 | 01100001 |
| ‘b’ | 98 | 01100010 |
| ‘c’ | 99 | 01100011 |
| ‘e’ | 101 | 01100101 |
| ‘z’ | 122 | 01111010 |

**What is the binary representation of the following String?**

cab ␣z

**Answer**

`01100011 01100001 01100010 00100000 01111010`

# ASCII Example

| Character | ASCII value | Binary Representation |
|---|---|---|
| ‘ ’ | 32 | 00100000 |
| ‘a’ | 97 | 01100001 |
| ‘b’ | 98 | 01100010 |
| ‘c’ | 99 | 01100011 |
| ‘e’ | 101 | 01100101 |
| ‘z’ | 122 | 01111010 |

**What is the binary representation of the following String?**

cab z

**Answer**

0110001101100001011000100010000001111010

# Another ASCII Example

| Character | ASCII value | Binary Representation |
|---|---|---|
| ‘ ’ | 32 | 00100000 |
| ‘a’ | 97 | 01100001 |
| ‘b’ | 98 | 01100010 |
| ‘c’ | 99 | 01100011 |
| ‘e’ | 101 | 01100101 |
| ‘z’ | 122 | 01111010 |

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

011000010110001101100101

# Another ASCII Example

| Character | ASCII value | Binary Representation |
|---|---|---|
| ‘ ’ | 32 | 00100000 |
| ‘a’ | 97 | 01100001 |
| ‘b’ | 98 | 01100010 |
| ‘c’ | 99 | 01100011 |
| ‘e’ | 101 | 01100101 |
| ‘z’ | 122 | 01111010 |

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

01100001 01100011 01100101

**Answer**

# Another ASCII Example

| Character | ASCII value | Binary Representation |
|---|---|---|
| ‘ ’ | 32 | 00100000 |
| ‘a’ | 97 | 01100001 |
| ‘b’ | 98 | 01100010 |
| ‘c’ | 99 | 01100011 |
| ‘e’ | 101 | 01100101 |
| ‘z’ | 122 | 01111010 |

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

01100001 01100011 01100101

**Answer**

a

# Another ASCII Example

| Character | ASCII value | Binary Representation |
|---|---|---|
| ‘ ’ | 32 | 00100000 |
| ‘a’ | 97 | 01100001 |
| ‘b’ | 98 | 01100010 |
| ‘c’ | 99 | 01100011 |
| ‘e’ | 101 | 01100101 |
| ‘z’ | 122 | 01111010 |

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

01100001 <u>01100011</u> 01100101

**Answer**

ac

# Another ASCII Example

| Character | ASCII value | Binary Representation |
|---|---|---|
| ‘ ’ | 32 | 00100000 |
| ‘a’ | 97 | 01100001 |
| ‘b’ | 98 | 01100010 |
| ‘c’ | 99 | 01100011 |
| ‘e’ | 101 | 01100101 |
| ‘z’ | 122 | 01111010 |

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

01100001 01100011 <u>01100101</u>

**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.

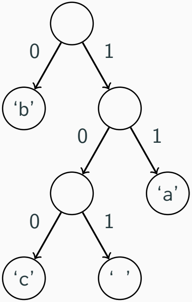
# Huffman Encoding

- 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**

| Character | Binary Representation |
|---|---|
| ‘ ’ | 00100000 |
| ‘a’ | 01100001 |
| ‘b’ | 01100010 |
| ‘c’ | 01100011 |
| ‘e’ | 01100101 |
| ‘z’ | 01111010 |

**Huffman Tree**



# Homework 8: Huffman Coding

Homework 8 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 |

# 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}`

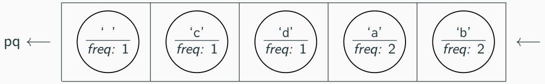
# Part A: Making a HuffmanCode Overview

| 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



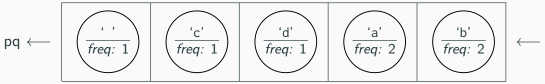
# Part A: Making a HuffmanCode Overview

| **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




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

# Part A: Making a HuffmanCode Overview

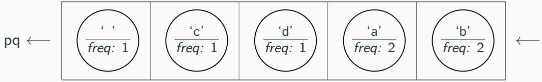
**Input File Contents**

```
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




**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}

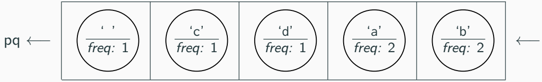
# Part A: Making a HuffmanCode Overview

**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




**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**

| **Input File** |
| --- |
| bad cab |

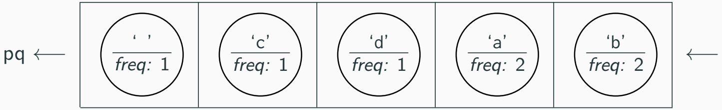
Generate Counts Array:

| index | 0 | 1 | ... | 32 | ... | 97 | 98 | 99 | 100 | 101 |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| value | 0 | 0 | | 1 | | 2 | 2 | 1 | 1 | 0 |

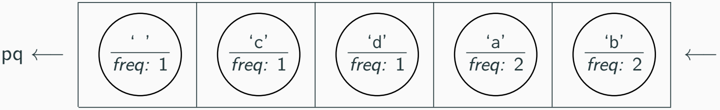
This is super similar to LetterInventory but works for all characters!

# Step 2: Create PriorityQueue

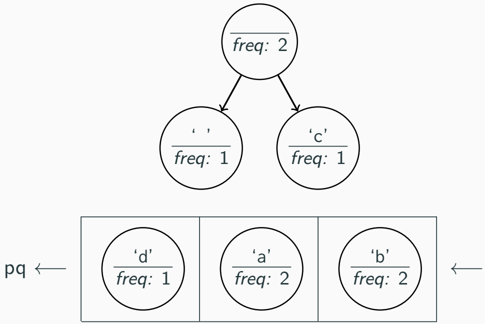
- Store each character and its frequency in a `HuffmanNode` object.
- Place all the `HuffmanNodes` in a `PriorityQueue` so that they are in ascending order with respect to **frequency**



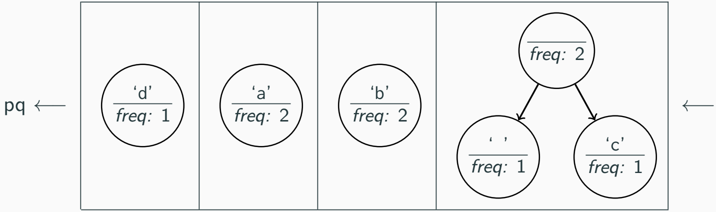
# Step 3: Remove and Merge



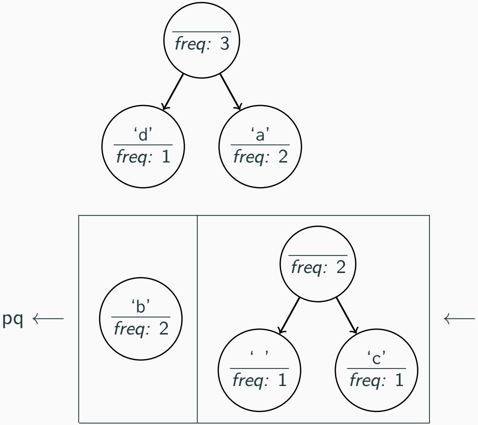
# Step 3: Remove and Merge



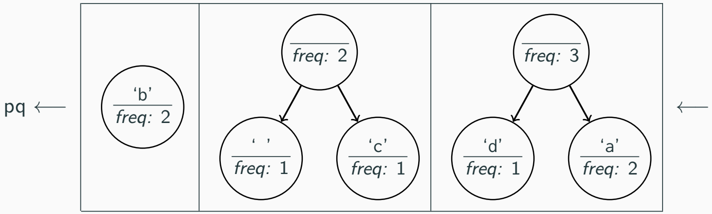
# Step 3: Remove and Merge



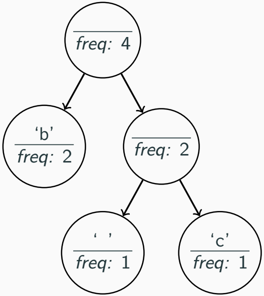
# Step 3: Remove and Merge

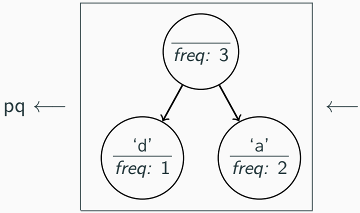


# Step 3: Remove and Merge

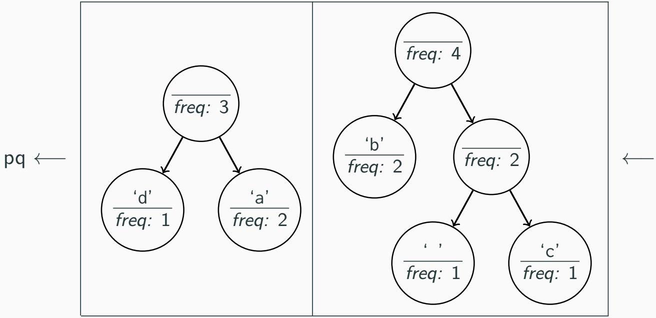


# Step 3: Remove and Merge

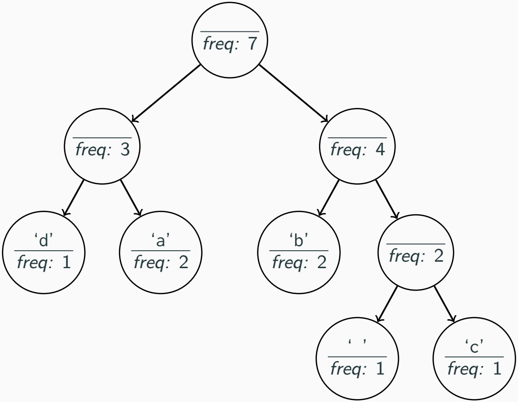


# Step 3: Remove and Merge



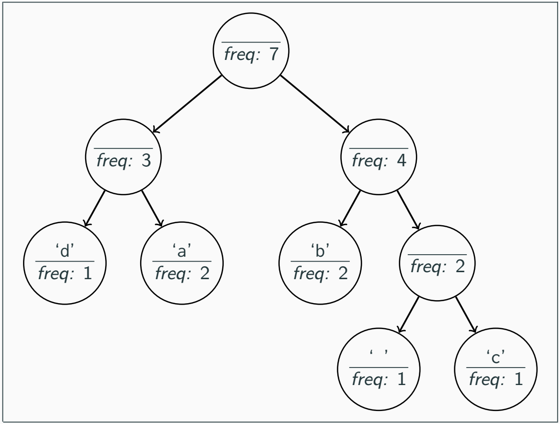
# Step 3: Remove and Merge




pq ⟵ □ ⟵

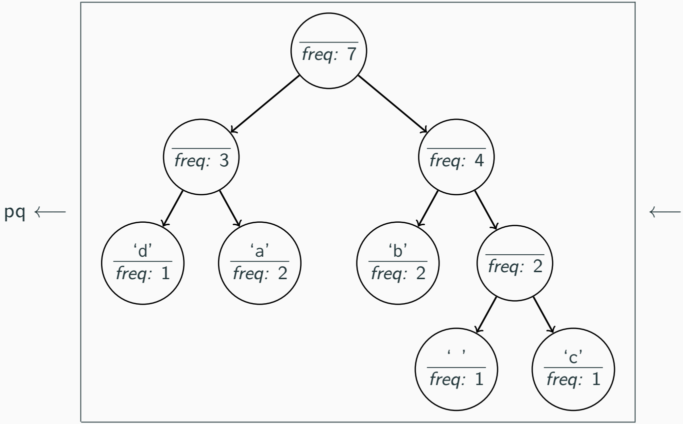
# Step 3: Remove and Merge

pq ⟵




⟵

# Step 3: Remove and Merge




- What is the relationship between frequency in file and binary representation length?

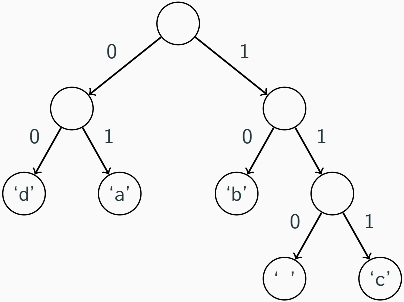
# Step 3: Remove and Merge Algorithm

**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.
```

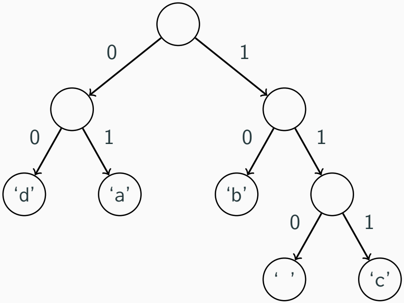
# Step 4: Print Encodings

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



# Step 4: Print Encodings

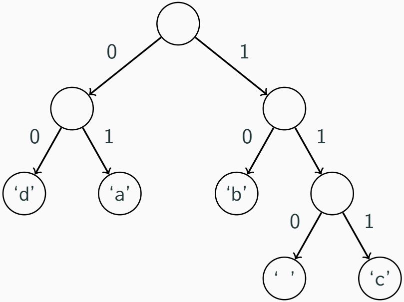
Save the tree to a file to save the encodings for the characters we made.




**Output of save**

# Step 4: Print Encodings

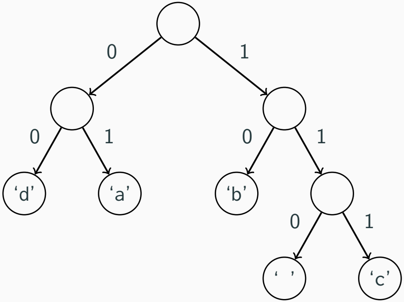
Save the tree to a file to save the encodings for the characters we made.




| Output of save |
| --- |
| 100 |
| 00 |

# Step 4: Print Encodings

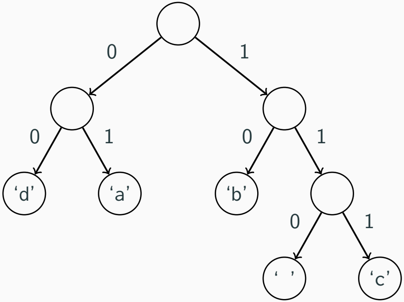
Save the tree to a file to save the encodings for the characters we made.




| Output of save |
|---|
| 100 |
| 00 |
| 97 |
| 01 |

# Step 4: Print Encodings

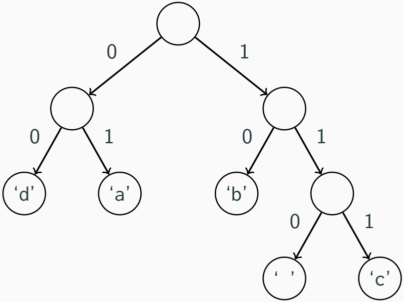
Save the tree to a file to save the encodings for the characters we made.




| Output of save |
|---|
| 100 |
| 00 |
| 97 |
| 01 |
| 98 |
| 10 |

# Step 4: Print Encodings

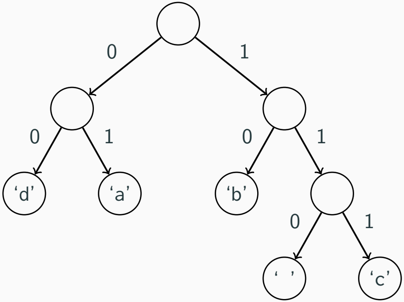
Save the tree to a file to save the encodings for the characters we made.




| Output of save |
|---|
| 100 |
| 00 |
| 97 |
| 01 |
| 98 |
| 10 |
| 32 |
| 110 |

# 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 |
| 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.

| Input File |
| --- |
| bad cab |

| Compressed Output |
| --- |
| |

**Huffman Encoding**

```
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.

| Input File |
| --- |
| bad cab |

| Compressed Output |
| --- |

**Huffman Encoding**

```
100 'd'
00
97  'a'
01
98  'b'
10
32  ' '
110
99  'c'
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.

| Input File |
|---|
| bad cab |

| Compressed Output |
|---|
| 10 01 100 110 111 01 10 |

**Huffman Encoding**

```
100 'd'
00
97  'a'
01
98  'b'
10
32  ' '
110
99  'c'
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.

**Input File**

bad cab

**Compressed Output**

10 01 100 110 111 01 10

**Uncompressed Output**

01100010 01100001 01100100
00100000 01100011 01100001
01100010

**Huffman Encoding**

```
100  'd'
00
97   'a'
01
98   'b'
10
32   ' '
110
99   'c'
111
```

# 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.

| Input code File |
| --- |
| 97 |
| 0 |
| 101 |
| 100 |
| 32 |
| 101 |
| 112 |
| 11 |

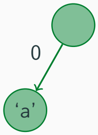
Initially the tree is empty

# Step 1: Reconstruct the Huffman Tree

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

| **Input code File** |
|---|
| 97 |
| 0 |
| 101 |
| 100 |
| 32 |
| 101 |
| 112 |
| 11 |

Tree after processing first pair

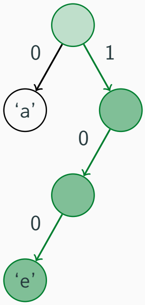


# Step 1: Reconstruct the Huffman Tree

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

| Input code File |
|---|
| 97 |
| 0 |
| 101 |
| 100 |
| 32 |
| 101 |
| 112 |
| 11 |

Tree after processing second pair

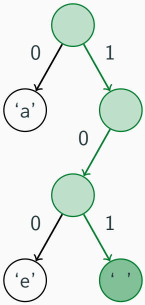


# Step 1: Reconstruct the Huffman Tree

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

| Input code File |
| --- |
| 97 |
| 0 |
| 101 |
| 100 |
| 32 |
| 101 |
| 112 |
| 11 |

Tree after processing third pair

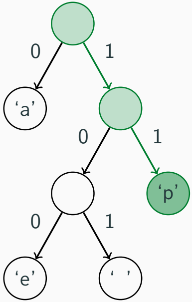


# Step 1: Reconstruct the Huffman Tree

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

| Input code File |
| --- |
| 97 |
| 0 |
| 101 |
| 100 |
| 32 |
| 101 |
| 112 |
| 11 |

Tree after processing last pair



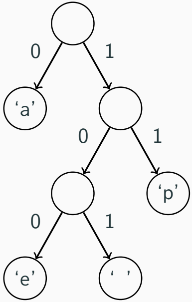
# Step 2 Example

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

**Input**

`0101110110101011100`

**Output**

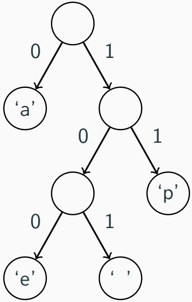


# Step 2 Example

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

| Input |
| --- |
| 0101110110101011100 |

| Output |
| --- |
| a papa ape |



# Working with Bits? That Sounds a Little Bit Hard

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

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 |

# Review - Homework 8

**Part A: Compression**

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

- Slides 15-17

```
public void save(PrintStream out)
```

- Slide 18

**Part B: Decompression**

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

- Slide 21

```
public void translate(BitInputStream in,
                      PrintStream out)
```

- Slide 22