Lecture 12: Chapter 4 Part 4

The Greedy Approach
CS3310

In-Class Exercise

1. Use Dijkstra's Algorithm to find the shortest path from v_5 to all the other vertices in the following graph. Show the values in F, `length`, and `touch` after each step

	1	2	3	4	5	6
1	0	∞	1	5	9	2
2	∞	0	3	2	5	7
3	1	3	0	∞	15	9
4	5	2	∞	0	2	3
5	9	8	15	2	0	8
6	2	7	9	3	8	0

Huffman Code

- The capacity of secondary storage devices keeps increasing and their cost keeps getting smaller, but they continue to fill up due to increased storage demands.
- The problem of **data compression** is to find the most efficient method for encoding a data file.
- The **Huffman Code** is a greedy algorithm for finding a Huffman encoding for a given file.

Huffman Code

- A common way to represent a file is with a **binary code**.
  - Each character is represented by a unique binary string, called its **codeword**.
  - A **fixed-length binary code** represents each character in a file with the same number of bits as every other character.

How many bits would we need to use a fixed-length binary code to encode the character set { a, b, c } ?

Huffman Code

- A common way to represent a file is with a **binary code**.
  - Each character is represented by a unique binary string, called its **codeword**.
  - A **fixed-length binary code** represents each character in a file with the same number of bits as every other character.

How many bits would we need to use a fixed-length binary code to encode the character set { a, b, c } ?

- 2 bits, which gives us enough for four possible codewords:
  - 01, 00, 10, 11
- We can simply choose three of them for our character set:
  - a: 00 b: 01 c: 11

Huffman Code

- Given this code:
  - a: 00 b: 01 c: 11
- And this file:
  - ababcbbbc
- Our encoding is:
  - 000100011101010111

Huffman Code

- A more efficient coding uses a **variable-length binary code** which can represent different characters with a different numbers of bits.

  ababcbbbc

- Given the above string, what character should we code as 0?

Huffman Code

- A more efficient coding uses a **variable-length binary code** which can represent different characters with a different numbers of bits.

  ababcbbbc

- Given the above string, what character should we code as 0?
- ‘b’. It occurs the most frequently, so we represent it with the least # of bits possible.
- Once we have assigned 0 to ‘b’, can assign ‘00’ to ‘a’?

Huffman Code

- A more efficient coding uses a **variable-length binary code** which can represent different characters with a different numbers of bits.

  ababcbbbc

- Given the above string, what character should we code as 0?
- ‘b’. It occurs the most frequently, so we represent it with the least # of bits possible.
- Once we have assigned 0 to ‘b’, can assign ‘00’ to ‘a’?
- **No**! We would not be able to distinguish one ‘a’ from two ‘b’s.
  - We also can’t encode ‘a’ as ‘01’, because when we encounter a 0, we won’t be able to determine if it represents a ‘b’ or the beginning of an ‘a’.

Huffman Code

- A valid variable-length binary code for the following string: ababcbbbc

  a: 10   b: 0   c: 11

- With this code, the string ababcbbbc is encoded as 1001001100011

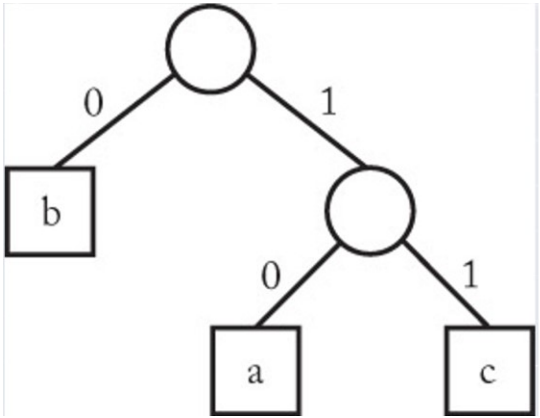
- This takes 5 bits less than the fixed-length binary code.

Huffman Code

- Given a file, the Optimal Binary Code problem is to find a binary character code for the characters in the file such that they are represented in the least number of bits.

Huffman Code

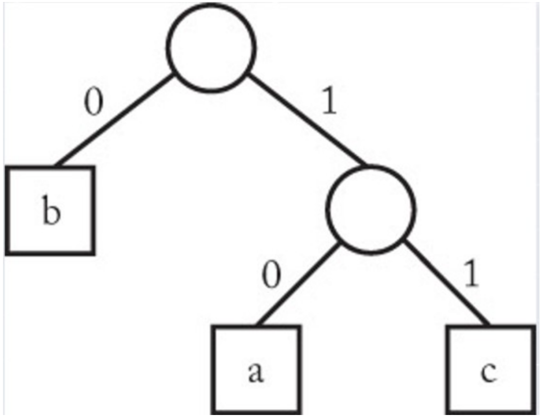
- In a **prefix code**, a codeword for one character cannot be the beginning of the codeword for another character.
  - i.e. if 01 is the codeword for ‘a’, 011 could not be the codeword for ‘b’.
- A prefix code can be represented by a binary tree whose leaves are encoded characters.



Huffman Code

- To decode a file, we read its first bit and traverse the tree starting at the root:

1001001100011

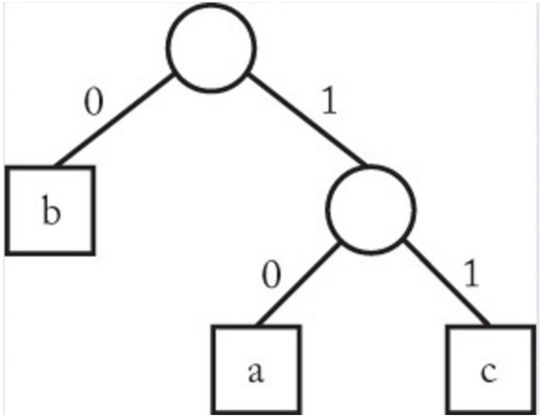


Huffman Code

- To decode a file, we read its first bit and traverse the tree starting at the root:

`1001001100011`

- The first bit is 1, so we go right in the tree. The second bit is 0, so we go left. We reach a leaf, so the first letter is ‘a’.
- We return to the tree’s root and repeat the process starting with the next bit in the file.
- The third bit is a 0, so we go left.
- We reach a leaf, so the second letter is ‘b’.
- This continues until we reach the end of the file.



Huffman Code

- Suppose we have the character set { a, b, c, d, e, f } with the following frequencies.
- The table shows how we would encode each letter using a different technique.

Character	Frequency	Code1 (Fixed)	Code2	Code3 (Huffman)
a	16	000	10	00
b	5	001	11110	1110
c	12	010	1110	110
d	17	011	110	01
e	10	100	11111	1111
f	25	101	0	10

Huffman Code

The # of bits used for each encoding is:

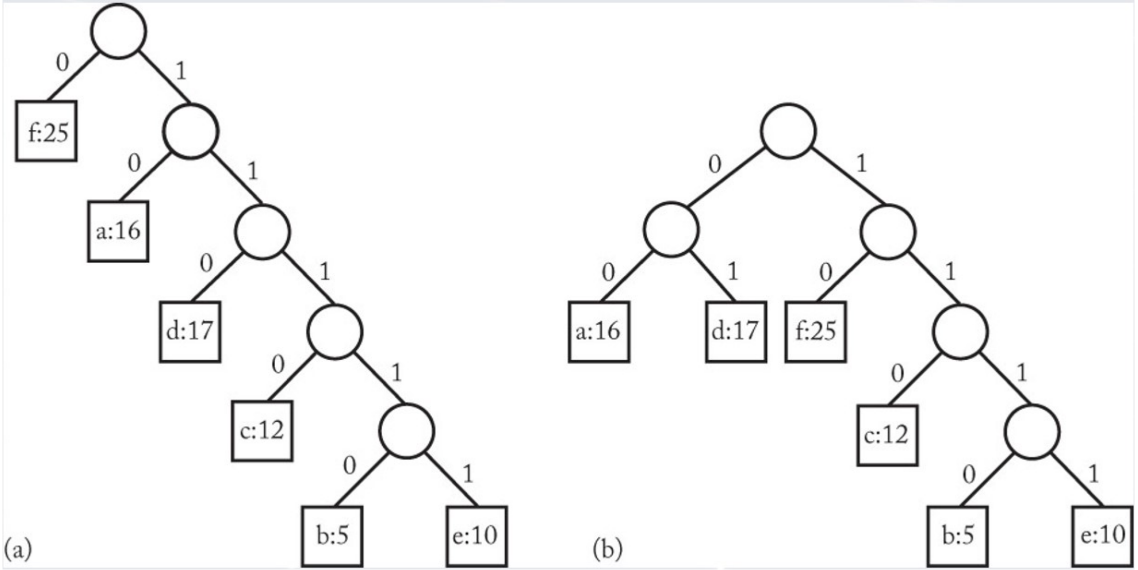
- Bits(Code1) = 16(3) + 5(3) + 12(3) + 17(3) + 10(3) + 25(3) = 255
- Bits(Code2) = 16(2) + 5(5) + 12(4) + 17(3) + 10(5) + 25(1) = 231
- Bits(Code3) = 16(2) + 5(4) + 12(3) + 17(2) + 10(4) + 25(2) = 212

Huffman Code is better than Code2. In fact, it is optimal

Character	Frequency	Code1 (Fixed)	Code2	Code3 (Huffman)
a	16	000	10	00
b	5	001	11110	1110
c	12	010	1110	110
d	17	011	110	01
e	10	100	11111	1111
f	25	101	0	10

Huffman Code

- The following binary trees represent Code2 (left) and Code3 (Huffman, right)
- Note that the Huffman tree is less deep. The less deep the tree, the less bits used in the encoding.



Huffman Code

- Huffman developed a greedy algorithm that produces an optimal binary character code by constructing a binary tree corresponding to that optimal code.
- For the pseudocode algorithm, we construct this tree out of instances of `nodetype`:

```
struct nodetype
{
          char symbol;                          // the letter this node represents
          int frequency;                        // How often the letter appears in the file

          nodetype *left;
          nodetype *right;
};
```

Huffman Code

- We store each `nodetype` object in a **priority queue**. Recall that in a priority queue, the element with the highest priority is always the first one removed.
- For this algorithm, the `nodetype` with the highest priority is the one with the *lowest* `frequency`.

```
struct nodetype
{
        char symbol;                    // the letter this node represents
        int frequency;                  // How often the letter appears in the file

        nodetype *left;
        nodetype *right;
};
```

Huffman's Algorithm

We start by initializing a priority queue of `nodetype` objects, one corresponding to each type of character in our file.

n = # of distinct character types in the file.

Add *n* pointers to *nodetype* objects to a priority queue PQ:

- For each pointer *p* in PQ
  - *p*->symbol = a distinct character type in the file
  - *p*->frequency = how many times that character appears in the file
  - *p*->left = *p*->right = NULL

Huffman's Algorithm Pseudocode

```
for (i = 1; i <= n - 1; i++)
{
          remove(PQ, p);
          remove(PQ, q);
          r = new nodetype;
          r->left = p;
          r->right = q;
          r->frequency = p->frequency + q->frequency;
          insert(PQ, r);
}
remove(PQ, r); // remove the tree's root from the priority queue
return r;
```

- In each iteration, we remove the front two `nodetypes` from the priority queue.
- We then create a new `nodetype` (**note**: we don't give it a symbol).
- The new `nodetype`'s children are p and q. Its frequency is the frequencies of p and q. We then add the new `nodetype` to the priority queue.

Huffman's Algorithm Pseudocode

```
for (i = 1; i <= n - 1; i++)
{
          remove(PQ, p);
          remove(PQ, q);
          r = new nodetype;
          r->left = p;
          r->right = q;
          r->frequency = p->frequency + q->frequency;
          insert(PQ, r);
}
remove(PQ, r); // remove the tree's root from the priority queue
return r;
```

- Why do we want to remove the lowest frequency nodes first?

Huffman's Algorithm Pseudocode

```
for (i = 1; i <= n - 1; i++)
{
          remove(PQ, p);
          remove(PQ, q);
          r = new nodetype;
          r->left = p;
          r->right = q;
          r->frequency = p->frequency + q->frequency;
          insert(PQ, r);
}
remove(PQ, r); // remove the tree's root from the priority queue
return r;
```

- Why do we want to remove the lowest frequency nodes first?
- We are building a tree from the bottom up.
    - The deeper a leaf, the more bits it needs to represent it.
- We want lower frequency letters to require more bits than higher frequency ones

Huffman's Algorithm Initialization

Given the following symbols and frequencies, what is the first step?

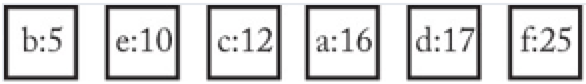
{ a: 16, b: 5, c: 12, d: 17, e: 10, f: 25 }

Huffman’s Algorithm Initialization

Given the following symbols and frequencies, what is the first step?

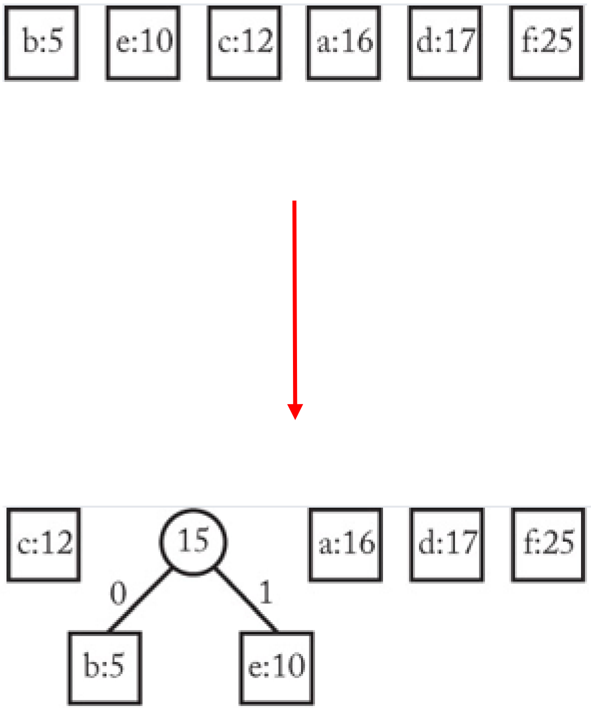
{ a: 16, b: 5, c: 12, d: 17, e: 10, f: 25 }

- Add each character to a `nodetype` object and place them in a priority queue.
- The front of the priority queue will contain the `nodetype` with the smallest frequency.



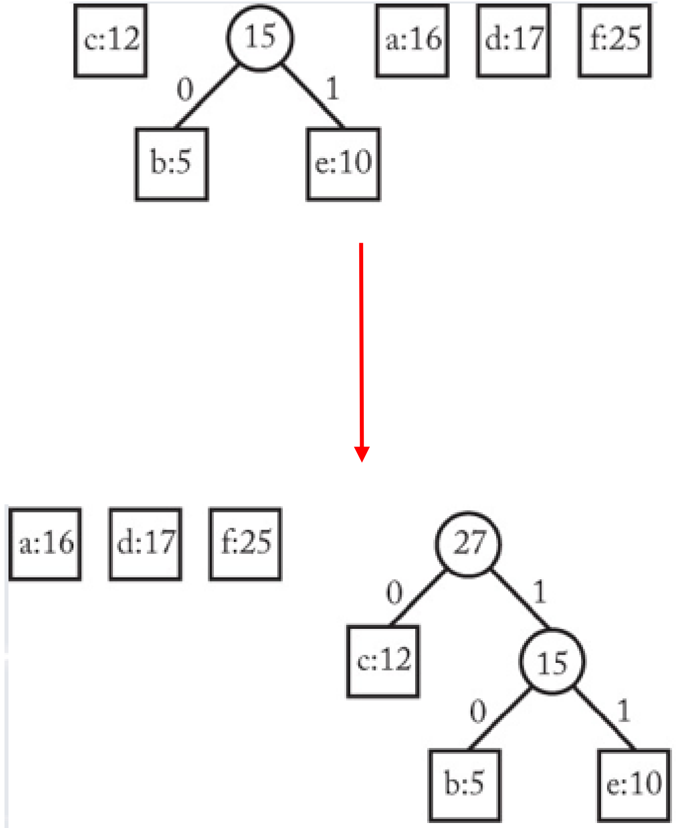
Huffman's Algorithm Step 1

- Remove first two entries from the priority queue.
  - i.e. the `nodetypes` for 'b' and 'e'
- Create a new `nodetype` object.
  - Its frequency is the frequencies of p and q:
    - 5 + 10 = 15.
    - This indicates that both *b* and *e* appear a total of 15 times in the file.
- Add the new `nodetype` to the priority queue.
- Since its frequency is 15, it becomes the second entry.



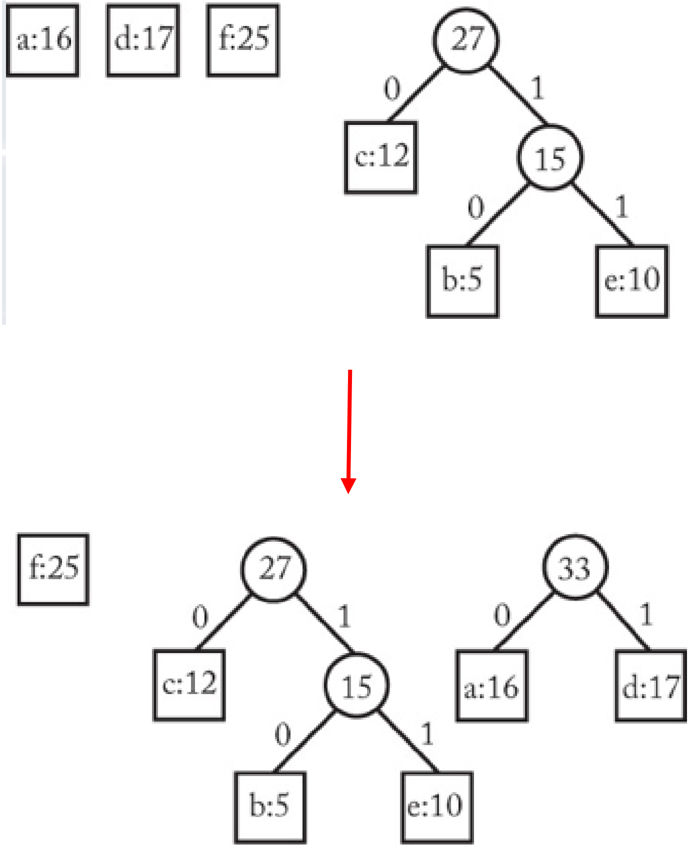
Huffman's Algorithm Step 2

- Remove first two entries from the priority queue.
  - i.e. the `nodetype` for 'c' and the `nodetype` with no symbol and a frequency of 15.
- Create a new `nodetype` object.
  - Its frequency is the frequencies of p and q:
    - 15 + 12 = 27.
- Add the new `nodetype` to the priority queue.
- Since its frequency is 27, it becomes the last entry



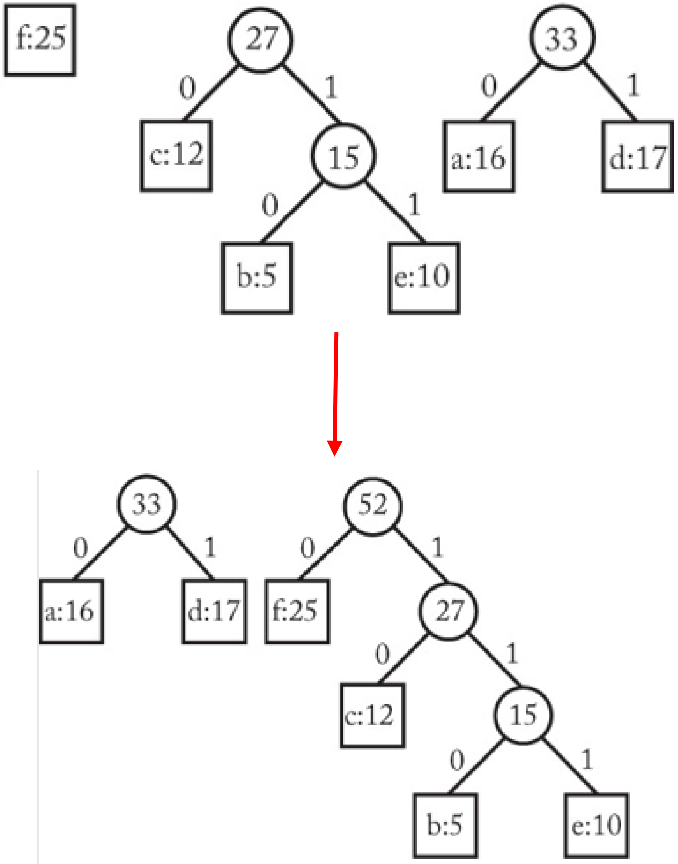
Huffman's Algorithm Step 3

- Remove first two entries from the priority queue.
  - i.e. the `nodetypes` for 'a' and 'd'
- Create a new `nodetype` object.
  - Its frequency is the frequencies of p and q:
    - $16 + 17 = 33$.
- Add the new `nodetype` to the priority queue.
- Since its frequency is 33, it becomes the last entry



Huffman's Algorithm Step 4

- Remove first two entries from the priority queue.
  - i.e. the `nodetype` for 'f' and the `nodetype` with no symbol and a frequency of 27.
- Create a new `nodetype` object.
  - Its frequency is the frequencies of p and q:
    - $25 + 27 = 52$.
- Add the new `nodetype` to the priority queue.
- Since its frequency is 52, it becomes the last entry

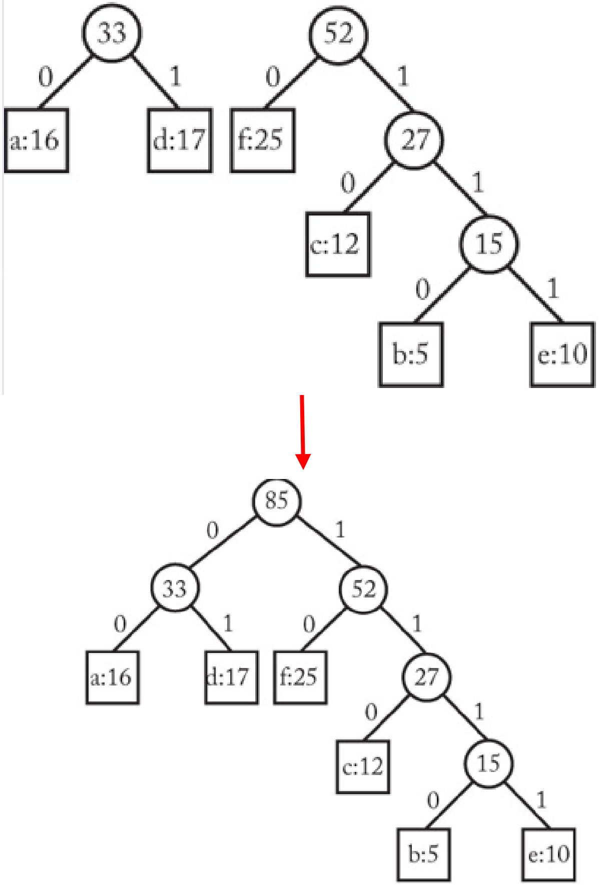


Huffman's Algorithm Step 5

- Remove first two entries from the priority queue.
  - i.e. the `nodetypes` with no symbols and frequencies of 33 and 52
- Create a new `nodetype` object.
  - Its frequency is the frequencies of p and q:
    - 33 + 52 = 85.
- The tree is completed!

Note that the lower the frequency, the further the depth (which is what we want)

This applies to `nodetypes` with and without symbols



In-Class Exercise

- Use Huffman's algorithm to construct an optimal binary prefix code for the letters in the following table

Letter	A	B	I	M	S	X	Z
Frequency	12	7	18	10	9	5	2