Huffman codes

- used for data compression, typically saving 20%–90%

Huffman codes

- used for data compression, typically saving 20%–90%
- Basic idea:
  represent often encountered characters by shorter (binary) codes

Huffman codes

Example

- Suppose we have the following data file with total 100 characters:

Char.	a	b	c	d	e	f
Freq.	45	13	12	16	9	5
3-bit fixed length code	000	001	010	011	100	101
variable length code	0	101	100	111	1101	1100

Huffman codes

Example

- Suppose we have the following data file with total 100 characters:

Char.	a	b	c	d	e	f
Freq.	45	13	12	16	9	5
3-bit fixed length code	000	001	010	011	100	101
variable length code	0	101	100	111	1101	1100

- Total number of bits required to encode the file:
  - Fixed-length code:

$$100 \times 3 = 300$$

Huffman codes

Example

- Suppose we have the following data file with total 100 characters:

Char.	a	b	c	d	e	f
Freq.	45	13	12	16	9	5
3-bit fixed length code	000	001	010	011	100	101
variable length code	0	101	100	111	1101	1100

- Total number of bits required to encode the file:
  - Fixed-length code:

$$100 \times 3 = 300$$

  - Variable-length code:

$$1{\cdot}45 + 3{\cdot}13 + 3{\cdot}12 + 3{\cdot}16 + 4{\cdot}9 + 4{\cdot}5 = 225$$

Huffman codes

Example

- Suppose we have the following data file with total 100 characters:

Char.	a	b	c	d	e	f
Freq.	45	13	12	16	9	5
3-bit fixed length code	000	001	010	011	100	101
variable length code	0	101	100	111	1101	1100

- Total number of bits required to encode the file:
  - Fixed-length code:

$$100 \times 3 = 300$$

  - Variable-length code:

$$1{\cdot}45 + 3{\cdot}13 + 3{\cdot}12 + 3{\cdot}16 + 4{\cdot}9 + 4{\cdot}5 = 225$$

- Variable-length code saves 25%.

Huffman codes

Prefix(-free) codes:

[1] *a word, letter or number placed before another*

Huffman codes

Prefix(-free) codes:

1. No codeword is also a prefix[1] of some other code.

[1] *a word, letter or number placed before another*

Huffman codes

Prefix(-free) codes:

1. No codeword is also a prefix[1] of some other code.
2. A prefix code for Example:

Char.	a	b	c	d	e	f
Code	0	101	100	111	1101	1100

3. Encoding and decoding with a prefix code.

[1] *a word, letter or number placed before another*

Huffman codes

Prefix(-free) codes:

1. No codeword is also a prefix[1] of some other code.
2. A prefix code for Example:

Char.	a	b	c	d	e	f
Code	0	101	100	111	1101	1100

3. Encoding and decoding with a prefix code.
4. Example, *cont'd*.
   - Encode:
     - beef $\longrightarrow$ 101110111011100
     - face $\longrightarrow$ 110001001101

[1] *a word, letter or number placed before another*

Huffman codes

Prefix(-free) codes:

1. No codeword is also a prefix[1] of some other code.
2. A prefix code for Example:

Char.	a	b	c	d	e	f
Code	0	101	100	111	1101	1100

3. Encoding and decoding with a prefix code.
4. Example, *cont'd*.
   - Encode:
     - beef $\longrightarrow$ 101110111011100
     - face $\longrightarrow$ 110001001101
   - Decode:
     - 101110111011100 $\longrightarrow$ beef
     - 110001001101 $\longrightarrow$ face

[1] *a word, letter or number placed before another*

Huffman codes

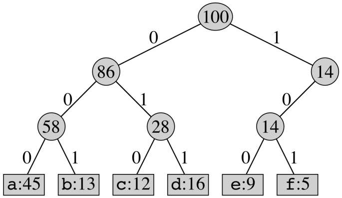
5. Representation of prefix code:
   - full binary tree: every nonleaf node has two children.

Huffman codes

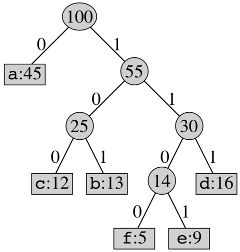
5. Representation of prefix code:
   - full binary tree: every nonleaf node has two children.
   - All legal codes are at the leaves, since no prefix is shared

Huffman codes

5. Representation of prefix code:
   - full binary tree: every nonleaf node has two children.
   - All legal codes are at the leaves, since no prefix is shared
6. Example, *cont'd*
   (a) the (not-full-binary) tree corresponding to the fixed-legnth code,
   (b) the (full-binary) tree corresponding to the prefix code:



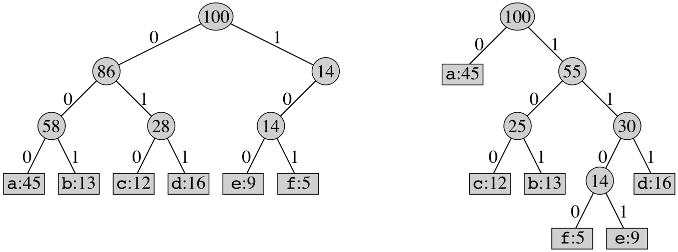
(a)



(b)

Huffman codes

5. Representation of prefix code:
   - full binary tree: every nonleaf node has two children.
   - All legal codes are at the leaves, since no prefix is shared
6. Example, *cont'd*
   (a) the (not-full-binary) tree corresponding to the fixed-legnth code,
   (b) the (full-binary) tree corresponding to the prefix code:



(a) (b)

7. *Fact: an optimal code for a file is always represented by a full binary tree.*

Huffman codes

Cost and optimality

Let C = alphabet (set of characters), then

Huffman codes

Cost and optimality

Let C = alphabet (set of characters), then

- A code = a binary tree T

Huffman codes

Cost and optimality

Let C = alphabet (set of characters), then

- A code = a binary tree T
- For each character $c \in C$, define

$$f(c) \quad = \quad \text{frequency of } c \text{ in the file}$$

Huffman codes

Cost and optimality

Let C = alphabet (set of characters), then

- A code = a binary tree T
- For each character $c \in C$, define

$$
\begin{aligned}
f(c) &= \text{frequency of } c \text{ in the file} \\
d_T(c) &= \text{length of the code for } c \\
&= \text{number of bits} \\
&= \text{depth of } c\text{' leave in the tree } T
\end{aligned}
$$

Huffman codes

Cost and optimality

Let C = alphabet (set of characters), then

- A code = a binary tree T
- For each character $c \in C$, define

$$
\begin{aligned}
f(c) &= \text{frequency of } c \text{ in the file} \\
d_T(c) &= \text{length of the code for } c \\
&= \text{number of bits} \\
&= \text{depth of } c\text{' leave in the tree } T
\end{aligned}
$$

Then the number of bits ("**cost** of the tree/code T") required to encode the file

$$B(T) = \sum_{c \in C} f(c) \cdot d_T(c),$$

Huffman codes

Cost and optimality

Let C = alphabet (set of characters), then

- A code = a binary tree T
- For each character $c \in C$, define

$$\begin{aligned} f(c) &= \text{frequency of } c \text{ in the file} \\ d_T(c) &= \text{length of the code for } c \\ &= \text{number of bits} \\ &= \text{depth of } c\text{' leave in the tree } T \end{aligned}$$

Then the number of bits ("**cost** of the tree/code T") required to encode the file

$$B(T) = \sum_{c \in C} f(c) \cdot d_T(c),$$

- A code T is **optimal** if $B(T)$ is minimal.

Huffman codes

Let C = alphabet (set of characters), basic idea of Huffman codes to produce a prefix code for C:

represent often encountered characters by shorter (binary) codes

Huffman codes

Let C = alphabet (set of characters), basic idea of Huffman codes to produce a prefix code for C:

represent often encountered characters by shorter (binary) codes

via

1. Building a full binary tree T in a *bottom-up* manner

Huffman codes

Let C = alphabet (set of characters), basic idea of Huffman codes to produce a prefix code for C:

> *represent often encountered characters by shorter (binary) codes*

via

1. Building a full binary tree T in a *bottom-up* manner
2. Beginning with $|C|$ leaves, performs a sequence of $|C| - 1$ "merging" operations to create T

Huffman codes

Let C = alphabet (set of characters), basic idea of Huffman codes to produce a prefix code for C:

> *represent often encountered characters by shorter (binary) codes*

via

1. Building a full binary tree T in a *bottom-up* manner
2. Beginning with $|C|$ leaves, performs a sequence of $|C| - 1$ "merging" operations to create T
3. "Merging" operation is *greedy:* the two with lowest frequencies are merged.

Review: priority queue

- A priority queue is a data structure for maintaining a set S of elements, each with an associated key.

Review: priority queue

- A priority queue is a data structure for maintaining a set S of elements, each with an associated key.
- A min-priority queue supports the following operations:
  - Insert(S,x): inserts the element x into the set S, i.e., $S = S \cup \{x\}$.
  - Minimum(S): returns the element of S with the smallest "key".
  - ExtractMin(S): removes and returns the element of S with the smallest "key".
  - DecreaseKey(S,x,k): decreases the value of element x's key to the new value k, which is assumed to be at least as small as x's current key value.

Review: priority queue

- A priority queue is a data structure for maintaining a set S of elements, each with an associated key.
- A min-priority queue supports the following operations:
  - Insert(S,x): inserts the element x into the set S, i.e., $S = S \cup \{x\}$.
  - Minimum(S): returns the element of S with the smallest "key".
  - ExtractMin(S): removes and returns the element of S with the smallest "key".
  - DecreaseKey(S,x,k): decreases the value of element x's key to the new value k, which is assumed to be at least as small as x's current key value.
- A max-priority queue supports the operations: Insert(S, x), Maximum(S), ExtractMax(S), IncreaseKey(S, x, k).

Review: priority queue

- A priority queue is a data structure for maintaining a set S of elements, each with an associated key.
- A min-priority queue supports the following operations:
  - Insert(S,x): inserts the element x into the set S, i.e., $S = S \cup \{x\}$.
  - Minimum(S): returns the element of S with the smallest "key".
  - ExtractMin(S): removes and returns the element of S with the smallest "key".
  - DecreaseKey(S,x,k): decreases the value of element x's key to the new value k, which is assumed to be at least as small as x's current key value.
- A max-priority queue supports the operations: Insert(S, x), Maximum(S), ExtractMax(S), IncreaseKey(S, x, k).
- Section 6.5 describes a binary heap implementation.
  - Cost: let $n = |S|$, then
    - initialization building heap $= O(n)$
    - each heap operation $= O(\lg n)$

Huffman codes

- Pseudocode:

```
Huffmancode(C)
n = |C|
Q = C    // min-priority queue, keyed by freq attribute
for i = 1 to n-1
    allocate a new node z
    z_left = x = ExtractMin(Q)
    z_right = y = ExtractMin(Q)
    freq[z] = freq[x] + freq[y]
    Insert(Q,z)
endfor
return ExtractMin(Q)  // the root of the tree
```

Huffman codes

- Pseudocode:

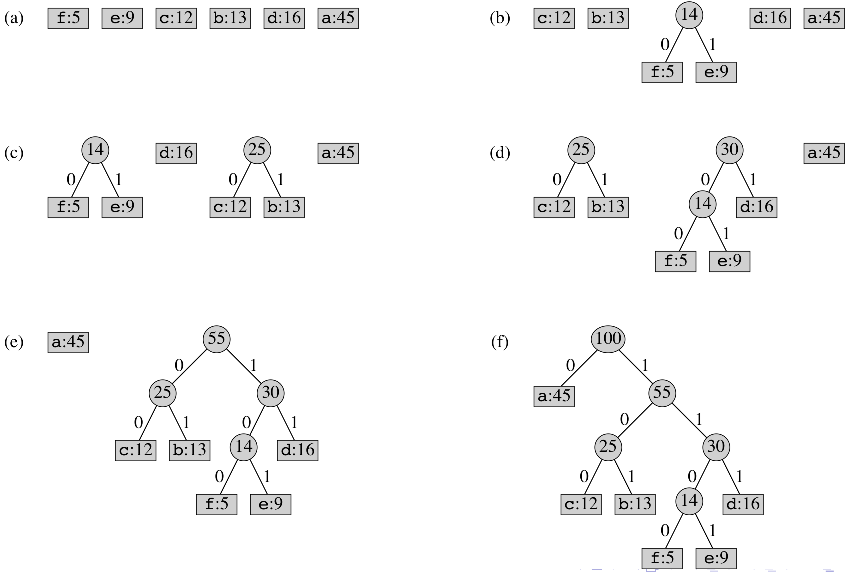
```
Huffmancode(C)
n = |C|
Q = C    // min-priority queue, keyed by freq attribute
for i = 1 to n-1
    allocate a new node z
    z_left = x = ExtractMin(Q)
    z_right = y = ExtractMin(Q)
    freq[z] = freq[x] + freq[y]
    Insert(Q,z)
endfor
return ExtractMin(Q)  // the root of the tree
```

- Running time:

$$T(n) = \underline{\text{init. Heap}} + \underline{(n-1)\text{ loop} \times \text{each Heap op.}}$$
$$= O(n) + O(n \lg n) = O(n \lg n)$$

Huffman codes

Example



Huffman codes

Optimality: To prove the greedy algorithm Huffmancode producing an optimal prefix code, we show that it exhibits the following two ingradients:

Huffman codes

Optimality: To prove the greedy algorithm Huffmancode producing an optimal prefix code, we show that it exhibits the following two ingradients:

1. **The greedy-choice property**

   If $x, y \in C$ having the lowest frequencies, then there exists an optimal code T such that

   - $d_T(x) = d_T(y)$
   - the codes for x and y differ only in the last bit

Huffman codes

Optimality: To prove the greedy algorithm Huffmancode producing an optimal prefix code, we show that it exhibits the following two ingradients:

1. **The greedy-choice property**

   If $x, y \in C$ having the lowest frequencies, then there exists an optimal code T such that

   - $d_T(x) = d_T(y)$
   - the codes for x and y differ only in the last bit

2. **The optimal substructure property**

   If $x, y \in C$ have the lowest frequencies, and let z be their parent. Then the tree

   $$T' = T - \{x, y\}$$

   represents an optimal prefix code for the alphabet

   $$C' = (C - \{x, y\}) \cup \{z\}.$$

Huffman codes

By the above two properties, after each greedy choice is made, we are left with an optimization problem of the same form as the original. By induction, we have

Theorem. Huffman code is an optimal prefix code.