Data Compression & Huffman Encoding

Algorithm Design & Analysis

Spring 2019
Outline

More on Union-Find
  Union-Find Recap
  An Improvement?
  An Improvement!

Data Compression: Huffman Encoding
Outline

More on Union-Find

Union-Find Recap
An Improvement?
An Improvement!

Data Compression: Huffman Encoding
A Union-Find Structure

Union-Find Data Structure
A Union-Find Structure

Union-Find Data Structure

- Manages a dynamic partition of a set $S$
A Union-Find Structure

Union-Find Data Structure

- Manages a dynamic partition of a set $S$
- Provides the following methods
More on Union-Find

A Union-Find Structure

Union-Find Data Structure

- Manages a dynamic partition of a set $S$
- Provides the following methods

  \textit{MakeUnionFind()}: Initialize the structure

Kruskal's Algorithm can then use \textit{Find} for cycle checking and \textit{Union} to update the structure after adding an edge to $T$. 
A Union-Find Structure

Union-Find Data Structure

- Manages a dynamic partition of a set $S$
- Provides the following methods

  - $\text{MakeUnionFind}()$: Initialize the structure
  - $\text{Find}(x)$: Return name of set containing $x$
A Union-Find Structure

Union-Find Data Structure

- Manages a dynamic partition of a set $S$
- Provides the following methods

  - $\text{MakeUnionFind}()$: Initialize the structure
  - $\text{Find}(x)$: Return name of set containing $x$
  - $\text{Union}(X, Y)$: Replace sets $X$ and $Y$ of partition with $Z = X \cup Y$. 
A Union-Find Structure

Union-Find Data Structure

- Manages a dynamic partition of a set $S$
- Provides the following methods

  $\text{MakeUnionFind}()$: Initialize the structure
  $\text{Find}(x)$: Return name of set containing $x$
  $\text{Union}(X, Y)$: Replace sets $X$ and $Y$ of partition with $Z = X \cup Y$.

Kruskal’s Algorithm can then use Find for cycle checking and Union to update the structure after adding an edge to $T$. 
First Union-Find Implementation

Let $S = \{1, \ldots, n\}$ be our set of items
First Union-Find Implementation

Let $S = \{1, \ldots, n\}$ be our set of items

- $\textit{MakeUnionFind}()$ creates one set for each vertex $v \in S$; the name of set is the name of the vertex.
First Union-Find Implementation

Let $S = \{1, \ldots, n\}$ be our set of items

- *MakeUnionFind()* creates one set for each vertex $v \in S$; the name of set is the name of the vertex.
  - We can use an array $UFSets[1 \ldots n]$ to hold the names: $UFSets[v] = v$: $O(n)$ time
First Union-Find Implementation

Let $S = \{1, \ldots, n\}$ be our set of items

- $\text{MakeUnionFind()}$ creates one set for each vertex $v \in S$; the name of set is the name of the vertex.
  - We can use an array $\text{UFSets}[1 \ldots n]$ to hold the names: $\text{UFSets}[v] = v$: $O(n)$ time

- $\text{Find}(v)$ works by looking up the name of the set containing $v$ in the array $\text{UFSets}[1 \ldots n]$: $O(1)$ time
First Union-Find Implementation

Let $S = \{1, \ldots, n\}$ be our set of items

- $\text{MakeUnionFind}()$ creates one set for each vertex $v \in S$; the name of set is the name of the vertex.
  - We can use an array $UFSets[1 \ldots n]$ to hold the names: $UFSets[v] = v$: $O(n)$ time

- $\text{Find}(v)$ works by looking up the name of the set containing $v$ in the array $UFSets[1 \ldots n]$: $O(1)$ time

- $\text{Union}(X, Y)$: $X \cup Y$ gets the name of whichever set $X$ or $Y$ is larger (ties are broken arbitrarily)
More on Union-Find

First Union-Find Implementation

Let $S = \{1, \ldots, n\}$ be our set of items

- $\text{MakeUnionFind}()$ creates one set for each vertex $v \in S$; the name of set is the name of the vertex.
  - We can use an array $\text{UFSets}[1 \ldots n]$ to hold the names: $\text{UFSets}[v] = v$: $O(n)$ time

- $\text{Find}(v)$ works by looking up the name of the set containing $v$ in the array $\text{UFSets}[1 \ldots n]$: $O(1)$ time

- $\text{Union}(X, Y)$: $X \cup Y$ gets the name of whichever set $X$ or $Y$ is larger (ties are broken arbitrarily)

- $\text{Union}(X, Y)$ changes the names of each of the elements in the smaller set to the name of the larger set: $O(n)$ time
First Union-Find Implementation

Let \( S = \{1, \ldots, n\} \) be our set of items

- \( \text{MakeUnionFind}() \) creates one set for each vertex \( v \in S \); the name of set is the name of the vertex.
  - We can use an array \( UFSets[1 \ldots n] \) to hold the names: \( UFSets[v] = v \): \( O(n) \) time

- \( \text{Find}(v) \) works by looking up the name of the set containing \( v \) in the array \( UFSets[1 \ldots n] \): \( O(1) \) time

- \( \text{Union}(X, Y) \): \( X \cup Y \) gets the name of whichever set \( X \) or \( Y \) is larger (ties are broken arbitrarily)

- \( \text{Union}(X, Y) \) changes the names of each of the elements in the smaller set to the name of the larger set: \( O(n) \) time
  - Doing this changes fewer names
**First Union-Find Implementation**

Let $S = \{1, \ldots, n\}$ be our set of items

- *MakeUnionFind()* creates one set for each vertex $v \in S$; the name of set is the name of the vertex.
  - We can use an array $UFSets[1 \ldots n]$ to hold the names: $UFSets[v] = v$: $O(n)$ time

- *Find*(v) works by looking up the name of the set containing $v$ in the array $UFSets[1 \ldots n]$: $O(1)$ time

- *Union*(X, Y): $X \cup Y$ gets the name of whichever set $X$ or $Y$ is larger (ties are broken arbitrarily)

- *Union*(X, Y) changes the names of each of the elements in the smaller set to the name of the larger set: $O(n)$ time
  - Doing this changes fewer names
  - Keeping linked lists of the elements of each set makes it easy to find the elements whose names need changing
Our First Union-Find Theorem

Theorem

Union-Find can be implemented so that MakeUnionFind takes $O(n)$ time, Find takes $O(1)$ time and any initial sequence of $k$ Unions takes $O(k \log k)$ time.
Our First Union-Find Theorem

Theorem
Union-Find can be implemented so that MakeUnionFind takes $O(n)$ time, Find takes $O(1)$ time and any initial sequence of $k$ Unions takes $O(k \log k)$ time.

Theorem
Restated Union-Find can be implemented so that MakeUnionFind takes $O(n)$ time, Find takes $O(1)$ time and any initial sequence of $k$ Unions and Finds takes $O(k \log k)$ time.
Our First Union-Find Theorem

Theorem
Union-Find can be implemented so that MakeUnionFind takes $O(n)$ time, Find takes $O(1)$ time and any initial sequence of $k$ Unions takes $O(k \log k)$ time.

Theorem
Restated Union-Find can be implemented so that MakeUnionFind takes $O(n)$ time, Find takes $O(1)$ time and any initial sequence of $k$ Unions and Finds takes $O(k \log k)$ time.

Corollary
Kruskal’s Algorithm can be implemented to run in $O(m \log m)$ time. [Repeatedly deleting from the heap is the bottleneck.]
Outline

More on Union-Find
  Union-Find Recap
  An Improvement?
  An Improvement!

Data Compression: Huffman Encoding
An Improvement?

Can we rename fewer vertices during a Union?
An Improvement?

Can we rename fewer vertices during a Union?

Idea: Relax the naming strategy: If \( v \in X_w \), then \( UFSets[v] \) needn’t be \( w \).
An Improvement?

Can we rename fewer vertices during a Union?

Idea: Relax the naming strategy: If $v \in X_w$, then $UFSets[v]$ needn’t be $w$.

- $UFSets[]$ encodes a tree for each set $X$ in our partition
An Improvement?

Can we rename fewer vertices during a Union?

Idea: Relax the naming strategy: If $v \in X_w$, then $UFSets[v]$ needn’t be $w$.

- $UFSets[]$ encodes a tree for each set $X$ in our partition
- The root of the tree for $X$ is name of $X$
An Improvement?

Can we rename fewer vertices during a Union?

Idea: Relax the naming strategy: If $v \in X_w$, then $UFSets[v]$ needn’t be $w$.

- $UFSets[]$ encodes a tree for each set $X$ in our partition
- The root of the tree for $X$ is name of $X$
- So $X$ has the name $x$ for some particular $x \in X$. 
An Improvement?

Can we rename fewer vertices during a Union?

Idea: Relax the naming strategy: If \( v \in X_w \), then \( UFSets[v] \) needn’t be \( w \).

- \( UFSets[] \) encodes a tree for each set \( X \) in our partition
- The root of the tree for \( X \) is name of \( X \)
- So \( X \) has the name \( x \) for some particular \( x \in X \).
- **We’ll only call** \( Union(x, y) \) **when** \( x \) **and** \( y \) **are vertices that name sets**
An Improvement?

Can we rename fewer vertices during a Union?

Idea: Relax the naming strategy: If $v \in X_w$, then $UFSets[v]$ needn’t be $w$.

- $UFSets[]$ encodes a tree for each set $X$ in our partition
- The root of the tree for $X$ is name of $X$
- So $X$ has the name $x$ for some particular $x \in X$.
- **We’ll only call** $Union(x,y)$ **when** $x$ **and** $y$ **are vertices that name sets**
- $Union(x,y)$ creates a set named by larger of the two sets
An Improvement?

Can we rename fewer vertices during a Union?

Idea: Relax the naming strategy: If \( v \in X_w \), then \( UFSets[v] \) needn’t be \( w \).

- \( UFSets[] \) encodes a tree for each set \( X \) in our partition
- The root of the tree for \( X \) is name of \( X \)
- So \( X \) has the name \( x \) for some particular \( x \in X \).
- **We’ll only call** \( Union(x, y) \) **when** \( x \) **and** \( y \) **are vertices that name sets**
- \( Union(x, y) \) **creates a set named by larger of the two sets**
- Thus, if the set named \( x \) is larger, \( UFSets[y] \leftarrow x \). (Set \( y \) now points to set \( x \))
An Improvement?

Can we rename fewer vertices during a Union?

Idea: Relax the naming strategy: If $v \in X_w$, then $UFSets[v]$ needn’t be $w$.

- $UFSets[]$ encodes a tree for each set $X$ in our partition
- The root of the tree for $X$ is name of $X$
- So $X$ has the name $x$ for some particular $x \in X$.
- **We’ll only call** $Union(x, y)$ **when** $x$ **and** $y$ **are vertices that name sets**
- $Union(x, y)$ creates a set named by larger of the two sets
- Thus, if the set named $x$ is larger, $UFSets[y] \leftarrow x$. (Set $y$ now points to set $x$)
- So, Union now takes $O(1)$ time!
An Improvement?

Can we rename fewer vertices during a Union?
Idea: Relax the naming strategy: If \( v \in X_w \), then \( \text{UFSets}[v] \) needn’t be \( w \).

- \( \text{UFSets}[\] encodes a tree for each set \( X \) in our partition
- The root of the tree for \( X \) is name of \( X \)
- So \( X \) has the name \( x \) for some particular \( x \in X \).
- We’ll only call \( \text{Union}(x, y) \) when \( x \) and \( y \) are vertices that name sets
- \( \text{Union}(x, y) \) creates a set named by larger of the two sets
- Thus, if the set named \( x \) is larger, \( \text{UFSets}[y] \leftarrow x \). (Set \( y \) now points to set \( x \))
- So, Union now takes \( O(1) \) time!
- Lists of vertices in each set no longer needed
An Improvement?

But how long does Find take now?
An Improvement?

But how long does Find take now?

Observation We can refer to the sets of the partition as trees.
An Improvement?

But how long does Find take now?

Observation We can refer to the sets of the partition as trees.

- An element of \( x \in S \) is the root of a tree iff \( \text{UFSets}[x] = x \).
An Improvement?

But how long does Find take now?

Observation We can refer to the sets of the partition as trees.

- An element of $x \in S$ is the root of a tree iff $UFSets[x] = x$.
- The tree resulting from a $Union(X, Y)$ has height at most 1 greater than the heights of $X$ and $Y$. 
An Improvement?

But how long does Find take now?

Observation We can refer to the sets of the partition as trees.

- An element of $x \in S$ is the root of a tree iff $UFSets[x] = x$.
- The tree resulting from a $Union(X, Y)$ has height at most 1 greater than the heights of $X$ and $Y$.
- Thus the height of any such tree of size $K$ is at most $\log K$
More on Union-Find

An Improvement?

But how long does Find take now?

**Observation** We can refer to the sets of the partition as trees.

- An element of \( x \in S \) is the root of a tree iff \( \text{UFSets}[x] = x \).
- The tree resulting from a \( \text{Union}(X, Y) \) has height at most 1 greater than the heights of \( X \) and \( Y \).
- Thus the height of any such tree of size \( K \) is at most \( \log K \).
  - Why? Let \( T_1 \) and \( T_2 \) be trees with heights \( h_1, h_2 \) and sizes \( k_1 \leq k_2 \), respectively, and such that \( h_i \) is \( O(\log k_i) \).
An Improvement?

But how long does Find take now?

**Observation** We can refer to the sets of the partition as trees.

- An element of \( x \in S \) is the root of a tree iff \( UFSets[x] = x \).
- The tree resulting from a \( Union(X, Y) \) has height at most 1 greater than the heights of \( X \) and \( Y \).
- Thus the height of any such tree of size \( K \) is at most \( \log K \)
  - Why? Let \( T_1 \) and \( T_2 \) be trees with heights \( h_1, h_2 \) and sizes \( k_1 \leq k_2 \), respectively, and such that \( h_i \) is \( O(\log k_i) \). Create \( T \) by merging \( T_1 \) and \( T_2 \) so that root of \( T_1 \) has root of \( T_2 \) as its parent and let \( T \) have height \( h \leq 1 + \max(h_1, h_2) \) and size \( k = k_1 + k_2 \).
An Improvement?

But how long does Find take now?

**Observation** We can refer to the sets of the partition as trees.

- An element of $x \in S$ is the root of a tree iff $\text{UFSets}[x] = x$.
- The tree resulting from a $\text{Union}(X, Y)$ has height at most 1 greater than the heights of $X$ and $Y$.
- Thus the height of any such tree of size $K$ is at most $\log K$
  - Why? Let $T_1$ and $T_2$ be trees with heights $h_1, h_2$ and sizes $k_1 \leq k_2$, respectively, and such that $h_i$ is $O(\log k_i)$. Create $T$ by merging $T_1$ and $T_2$ so that root of $T_1$ has root of $T_2$ as its parent and let $T$ have height $h \leq 1 + \max(h_1, h_2)$ and size $k = k_1 + k_2$. Then $h$ is $O(\log k)$
An Improvement?

But how long does Find take now?

**Observation** We can refer to the sets of the partition as trees.

- An element of $x \in S$ is the root of a tree iff $\text{UFSets}[x] = x$.
- The tree resulting from a $\text{Union}(X, Y)$ has height at most $1$ greater than the heights of $X$ and $Y$.
- Thus the height of any such tree of size $K$ is at most $\log K$
  - Why? Let $T_1$ and $T_2$ be trees with heights $h_1$, $h_2$ and sizes $k_1 \leq k_2$, respectively, and such that $h_i$ is $O(\log k_i)$. Create $T$ by merging $T_1$ and $T_2$ so that root of $T_1$ has root of $T_2$ as its parent and let $T$ have height $h \leq 1 + \max(h_1, h_2)$ and size $k = k_1 + k_2$. Then $h$ is $O(\log k)$
  - This is easy to prove by induction
Our New Result
**Our New Result**

- Thus, for $x \in X$, $Find(x)$ now uses UFSets array to find root of tree containing $x$ (that is, find $s \in X$ with $UFSets[s] = s$)
Our New Result

- Thus, for $x \in X$, $\text{Find}(x)$ now uses UFSets array to find root of tree containing $x$ (that is, find $s \in X$ with $\text{UFSets}[s] = s$)
- Since tree has height at most $\log |X|$, Find takes at most $O(\log |X|)$ time
Our New Result

- Thus, for \( x \in X \), \( \text{Find}(x) \) now uses UFSets array to find root of tree containing \( x \) (that is, find \( s \in X \) with \( \text{UFSets}[s] = s \))
- Since tree has height at most \( \log |X| \), Find takes at most \( O(\log |X|) \) time

**Theorem**

**Version 2** Union-Find can be implemented so that \( \text{MakeUnionFind} \) takes \( O(n) \) time, Union takes \( O(1) \) time, Find takes \( O(\log n) \) time and any initial sequence of \( k \) Unions and Finds takes \( O(k \log k) \) time.
Outline

More on Union-Find
Union-Find Recap
An Improvement?
An Improvement!

Data Compression: Huffman Encoding
But wait! There’s more...

- Maybe using sizes of trees to determine how Union merges isn’t optimal
But wait! There’s more...

- Maybe using sizes of trees to determine how Union merges isn’t optimal
- We could use tree height instead: merge shallower tree into deeper tree
But wait! There’s more...

- Maybe using sizes of trees to determine how Union merges isn’t optimal
- We could use tree height instead: merge shallower tree into deeper tree
- Maintains fact that height is at most $\log(size)$
But wait! There’s more...

- Maybe using sizes of trees to determine how Union merges isn’t optimal
- We could use tree height instead: merge shallower tree into deeper tree
- Maintains fact that height is at most log(size)
- Even better: Every time we do a Find(x), redirect all nodes traversed in tree to root

Theorem
Using path compression, any initial sequence of m Union and Find operations on n items after a MakeUnionFind can be carried out in $O(n + m \log^* n)$ time.
But wait! There’s more...

- Maybe using sizes of trees to determine how *Union* merges isn’t optimal
- We could use tree height instead: merge shallower tree into deeper tree
- Maintains fact that height is at most $\log(size)$
- Even better: Every time we do a *Find*($x$), redirect all nodes traversed in tree to root
- This is called *path compression*
But wait! There’s more...

- Maybe using sizes of trees to determine how \textit{Union} merges isn’t optimal
- We could use tree height instead: merge shallower tree into deeper tree
- Maintains fact that height is at most \( \log(size) \)
- Even better: Every time we do a \textit{Find}(x), redirect all nodes traversed in tree to root
- This is called \textit{path compression}

Let’s see an example at visualgo.net
But wait! There’s more...

- Maybe using sizes of trees to determine how Union merges isn’t optimal
- We could use tree height instead: merge shallower tree into deeper tree
- Maintains fact that height is at most $\log(size)$
- Even better: Every time we do a $Find(x)$, redirect all nodes traversed in tree to root
- This is called path compression

Let’s see an example at visualgo.net

**Theorem**

*Using path compression, any initial sequence of $m$ Union and Find operations on $n$ items after a MakeUnionFind can be carried out in $O(n + m \log^* n)$ time.*
But What is This $\log^* n$ Function?
But What is This $\log^* n$ Function?

Definition
For any base $b > 1$, $\log^*_b(n)$ is the number of times $\log_b$ must be repeatedly applied to $n$ before the result is at most 1. Precisely:

$$\log^*(n) = \begin{cases} 0 & \text{if } n \leq 1 \\ 1 + \log^*(\log n) & \text{if } n > 1 \end{cases}$$
But What is This $\log^* n$ Function?

**Definition**

For any base $b > 1$, $\log^*_b(n)$ is the number of times $\log_b$ must be repeatedly applied to $n$ before the result is at most 1. Precisely:

$$\log^*(n) = \begin{cases} 
0 & \text{if } n \leq 1 \\
1 + \log^*(\log n) & \text{if } n > 1 
\end{cases}$$

$log^* n$ grows very slowly....
**But What is This \( \log^* n \) Function?**

**Definition**
For any base \( b > 1 \), \( \log_b^*(n) \) is the number of times \( \log_b \) must be repeatedly applied to \( n \) before the result is at most 1. Precisely:

\[
\log^*(n) = \begin{cases} 
0 & \text{if } n \leq 1 \\
1 + \log^*(\log n) & \text{if } n > 1
\end{cases}
\]

\( \log^* n \) grows very slowly....

<table>
<thead>
<tr>
<th>( n )</th>
<th>1</th>
<th>2</th>
<th>4 = 2^2</th>
<th>16 = 2^4</th>
<th>65,536 = 2^{16}</th>
<th>2^{65,536}</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \log^*(n) )</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
</tbody>
</table>
Why Data Compression?
Why Data Compression?

Some typical file sizes:

Tom Sawyer: Over 200,000 bytes
Why Data Compression?

Some typical file sizes:

*Tom Sawyer:* Over 200,000 bytes

*Remembrance of Things Past:* Over 9,600,000 bytes
Why Data Compression?

Some typical file sizes:

*Tom Sawyer:* Over 200,000 bytes

*Remembrance of Things Past:* Over 9,600,000 bytes

*$100 digital camera:* over 30MB

(10MP resolution x 3 bytes/pixel)
Why Data Compression?

Some typical file sizes:

*Tom Sawyer*: Over 200,000 bytes

*Remembrance of Things Past*: Over 9,600,000 bytes

*$100 digital camera*: over 30MB
  
  (10MP resolution x 3 bytes/pixel)

*Single 1080p image frame*: 1920*1080 pixels for 6.2M bytes
Why Data Compression?

Some typical file sizes:

*Tom Sawyer:* Over 200,000 bytes

*Remembrance of Things Past:* Over 9,600,000 bytes

*$100 digital camera:* over 30MB
  (10MP resolution x 3 bytes/pixel)

*Single 1080p image frame:* 1920*1080 pixels for 6.2M bytes

*2-hour streaming film:* 2.6TB at 60 frames/sec
Data Compression Characteristics

- Compression Ratio: uncompressed/compressed
Data Compression Characteristics

- Compression Ratio: uncompressed/compressed
- Data Loss: Lossless vs Lossy
Data Compression Characteristics

- Compression Ratio: uncompressed/compressed
- Data Loss: Lossless vs Lossy
- Fast Encoding: Frequently, but not always, needed.
Data Compression Characteristics

- Compression Ratio: uncompressed/compressed
- Data Loss: Lossless vs Lossy
- Fast Encoding: Frequently, but not always, needed.
- Fast Decoding: Frequently, but not always, needed.

We’ll focus on lossless encoding using mapping from source text alphabet $A$ to strings over a target alphabet $B$. 
Data Compression Characteristics

- Compression Ratio: uncompressed/compressed
- Data Loss: Lossless vs Lossy
- Fast Encoding: Frequently, but not always, needed.
- Fast Decoding: Frequently, but not always, needed.

We’ll focus on lossless encoding using mapping from source text alphabet $A$ to strings over a target alphabet $B$
An Old Example

International Morse Code

1. The length of a dot is one unit.
2. A dash is three units.
3. The space between parts of the same letter is one unit.
4. The space between letters is three units.
5. The space between words is seven units.

Figure: Courtesy wikipedia.org
An Old Example

International Morse Code

1. The length of a dot is one unit.
2. A dash is three units.
3. The space between parts of the same letter is one unit.
4. The space between letters is three units.
5. The space between words is seven units.

Figure: Courtesy wikipedia.org

This encoding has some problems.....
### ASCII TABLE

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Hex</th>
<th>Char</th>
<th>Decimal</th>
<th>Hex</th>
<th>Char</th>
<th>Decimal</th>
<th>Hex</th>
<th>Char</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>[NULL]</td>
<td>32</td>
<td>20</td>
<td>[SPACE]</td>
<td>64</td>
<td>40</td>
<td>@</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>[START OF HEADING]</td>
<td>33</td>
<td>21</td>
<td>!</td>
<td>65</td>
<td>41</td>
<td>A</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>[START OF TEXT]</td>
<td>34</td>
<td>22</td>
<td>*</td>
<td>66</td>
<td>42</td>
<td>B</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>[END OF TEXT]</td>
<td>35</td>
<td>23</td>
<td>#</td>
<td>67</td>
<td>43</td>
<td>C</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>[END OF TRANSMISSION]</td>
<td>36</td>
<td>24</td>
<td>$</td>
<td>68</td>
<td>44</td>
<td>D</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>[ENQUIRY]</td>
<td>37</td>
<td>25</td>
<td>%</td>
<td>69</td>
<td>45</td>
<td>E</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>[ACKNOWLEDGE]</td>
<td>38</td>
<td>26</td>
<td>&amp;</td>
<td>70</td>
<td>46</td>
<td>F</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>[BEL]</td>
<td>39</td>
<td>27</td>
<td>'</td>
<td>71</td>
<td>47</td>
<td>G</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>[BACKSPACE]</td>
<td>40</td>
<td>28</td>
<td>(</td>
<td>72</td>
<td>48</td>
<td>H</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
<td>[horizontal tab]</td>
<td>41</td>
<td>29</td>
<td>)</td>
<td>73</td>
<td>49</td>
<td>I</td>
</tr>
<tr>
<td>10</td>
<td>A</td>
<td>[line feed]</td>
<td>42</td>
<td>2A</td>
<td>*</td>
<td>74</td>
<td>4A</td>
<td>J</td>
</tr>
<tr>
<td>11</td>
<td>B</td>
<td>[vertical tab]</td>
<td>43</td>
<td>2B</td>
<td>+</td>
<td>75</td>
<td>4B</td>
<td>K</td>
</tr>
<tr>
<td>12</td>
<td>C</td>
<td>[form feed]</td>
<td>44</td>
<td>2C</td>
<td>,</td>
<td>76</td>
<td>4C</td>
<td>L</td>
</tr>
<tr>
<td>13</td>
<td>D</td>
<td>[carriage return]</td>
<td>45</td>
<td>2D</td>
<td>-</td>
<td>77</td>
<td>4D</td>
<td>M</td>
</tr>
<tr>
<td>14</td>
<td>E</td>
<td>[shift-out]</td>
<td>46</td>
<td>2E</td>
<td>.</td>
<td>78</td>
<td>4E</td>
<td>N</td>
</tr>
<tr>
<td>15</td>
<td>F</td>
<td>[shift-in]</td>
<td>47</td>
<td>2F</td>
<td>/</td>
<td>79</td>
<td>4F</td>
<td>O</td>
</tr>
<tr>
<td>16</td>
<td>10</td>
<td>[data link escape]</td>
<td>48</td>
<td>30</td>
<td>0</td>
<td>80</td>
<td>50</td>
<td>P</td>
</tr>
<tr>
<td>17</td>
<td>11</td>
<td>[device control 1]</td>
<td>49</td>
<td>31</td>
<td>1</td>
<td>81</td>
<td>51</td>
<td>Q</td>
</tr>
<tr>
<td>18</td>
<td>12</td>
<td>[device control 2]</td>
<td>50</td>
<td>32</td>
<td>2</td>
<td>82</td>
<td>52</td>
<td>R</td>
</tr>
<tr>
<td>19</td>
<td>13</td>
<td>[device control 3]</td>
<td>51</td>
<td>33</td>
<td>3</td>
<td>83</td>
<td>53</td>
<td>S</td>
</tr>
<tr>
<td>20</td>
<td>14</td>
<td>[device control 4]</td>
<td>52</td>
<td>34</td>
<td>4</td>
<td>84</td>
<td>54</td>
<td>T</td>
</tr>
<tr>
<td>21</td>
<td>15</td>
<td>[negative acknowledge]</td>
<td>53</td>
<td>35</td>
<td>5</td>
<td>85</td>
<td>55</td>
<td>U</td>
</tr>
<tr>
<td>22</td>
<td>16</td>
<td>[synchronous idle]</td>
<td>54</td>
<td>36</td>
<td>6</td>
<td>86</td>
<td>56</td>
<td>V</td>
</tr>
<tr>
<td>23</td>
<td>17</td>
<td>[end of trans. block]</td>
<td>55</td>
<td>37</td>
<td>7</td>
<td>87</td>
<td>57</td>
<td>W</td>
</tr>
<tr>
<td>24</td>
<td>18</td>
<td>[cancel]</td>
<td>56</td>
<td>38</td>
<td>8</td>
<td>88</td>
<td>58</td>
<td>X</td>
</tr>
<tr>
<td>25</td>
<td>19</td>
<td>[end of medium]</td>
<td>57</td>
<td>39</td>
<td>9</td>
<td>89</td>
<td>59</td>
<td>Y</td>
</tr>
<tr>
<td>26</td>
<td>1A</td>
<td>[substitute]</td>
<td>58</td>
<td>3A</td>
<td>:</td>
<td>90</td>
<td>5A</td>
<td>Z</td>
</tr>
<tr>
<td>27</td>
<td>1B</td>
<td>[escape]</td>
<td>59</td>
<td>3B</td>
<td>;</td>
<td>91</td>
<td>5B</td>
<td>l</td>
</tr>
<tr>
<td>28</td>
<td>1C</td>
<td>[file separator]</td>
<td>60</td>
<td>3C</td>
<td>&lt;</td>
<td>92</td>
<td>5C</td>
<td>\</td>
</tr>
<tr>
<td>29</td>
<td>1D</td>
<td>[group separator]</td>
<td>61</td>
<td>3D</td>
<td>=</td>
<td>93</td>
<td>5D</td>
<td>]</td>
</tr>
<tr>
<td>30</td>
<td>1E</td>
<td>[record separator]</td>
<td>62</td>
<td>3E</td>
<td>&gt;</td>
<td>94</td>
<td>5E</td>
<td>^</td>
</tr>
<tr>
<td>31</td>
<td>1F</td>
<td>[unit separator]</td>
<td>63</td>
<td>3F</td>
<td>?</td>
<td>95</td>
<td>5F</td>
<td>[DEL]</td>
</tr>
</tbody>
</table>

**Figure:** Each symbol is encoded as 2 hexadecimal digits (or 7 bits)
A Class of Encoding Schemes

- Input: string $C = c_1 \ldots c_m$ of symbols from $A$
A Class of Encoding Schemes

- Input: string $C = c_1 \ldots c_m$ of symbols from $A$
- Encoding function: $\gamma : A \rightarrow B^+$
A Class of Encoding Schemes

- Input: string $C = c_1 \ldots c_m$ of symbols from $A$
- Encoding function: $\gamma : A \rightarrow B^+$
- $B^+$: set of all non-empty, finite strings over $B$
A Class of Encoding Schemes

- Input: string $C = c_1 \ldots c_m$ of symbols from $A$
- Encoding function: $\gamma : A \rightarrow B^+$
- $B^+$: set of all non-empty, finite strings over $B$
- Result: $C$ is encoded as $\gamma(C) = \gamma(c_1) \ldots \gamma(c_m)$
A Class of Encoding Schemes

- Input: string $C = c_1 \ldots c_m$ of symbols from $A$
- Encoding function: $\gamma : A \rightarrow B^+$
- $B^+$: set of all non-empty, finite strings over $B$
- Result: $C$ is encoded as $\gamma(C) = \gamma(c_1) \ldots \gamma(c_m)$
- Encoding Length (Symbol): $l_a = |\gamma(a)|$, for $a \in A$. 
A Class of Encoding Schemes

- Input: string $C = c_1 \ldots c_m$ of symbols from $A$
- Encoding function: $\gamma : A \rightarrow B^+$
- $B^+$: set of all non-empty, finite strings over $B$
- Result: $C$ is encoded as $\gamma(C) = \gamma(c_1) \ldots \gamma(c_m)$
- Encoding Length (Symbol): $l_a = |\gamma(a)|$, for $a \in A$.
- Length of encoding:

$$EL(C) = |\gamma(C)| = \sum_{i=1}^{m} |\gamma(c_i)| = \sum_{i=1}^{m} l_{c_i}$$
**Data Compression Notation**

If each \( a \in A \) occurs \( m_a \) times in \( C \) and \( m = \sum_{a \in A} m_a \) then
Data Compression Notation

If each $a \in A$ occurs $m_a$ times in $C$ and $m = \sum_{a \in A} m_a$ then

$$EL(C) = \sum_{i=1}^{m} l_{c_i} = \sum_{a \in A} m_a l_a = m \sum_{a \in A} \frac{m_a}{m} l_a = m \sum_{a \in A} f_a l_a$$

Frequency Distribution:

$f_a = m_a / m$. Note: $f_a : A \rightarrow [0, 1]$, where $\sum_{a \in A} f_a = 1$ and each $f_a \geq 0$. That is: $f_a$ is a probability distribution on $A$. Encoding Length (text): $EL(C) = \sum_{i=1}^{m} l_{c_i}$

Average Encoding Length (of a symbol in $A$): $AEL(f, \gamma) = \sum_{a \in A} f_a l_a$

Goal: Minimize $EL$ or, equivalently, $AEL$. 
Data Compression Notation

If each $a \in A$ occurs $m_a$ times in $C$ and $m = \sum_{a \in A} m_a$ then

$$EL(C) = \sum_{i=1}^{m} l_{c_i} = \sum_{a \in A} m_a l_a = m \sum_{a \in A} \frac{m_a}{m} l_a = m \sum_{a \in A} f_a l_a$$

Frequency Distribution: $f_a = \frac{m_a}{m}$. 
Data Compression Notation

If each $a \in A$ occurs $m_a$ times in $C$ and $m = \sum_{a \in A} m_a$ then

$$EL(C) = \sum_{i=1}^{m} l_{c_i} = \sum_{a \in A} m_a l_a = m \sum_{a \in A} \frac{m_a}{m} l_a = m \sum_{a \in A} f_a l_a$$

**Frequency Distribution:** $f_a = \frac{m_a}{m}$.

Note: $f : A \rightarrow [0, 1]$, where $\sum_{a \in A} f_a = 1$ and each $f_a \geq 0$
Data Compression Notation

If each $a \in A$ occurs $m_a$ times in $C$ and $m = \sum_{a \in A} m_a$ then

$$EL(C) = \sum_{i=1}^{m} l_{c_i} = \sum_{a \in A} m_a l_a = m \sum_{a \in A} \frac{m_a}{m} l_a = m \sum_{a \in A} f_a l_a$$

**Frequency Distribution:** $f_a = \frac{m_a}{m}$.

Note: $f : A \rightarrow [0, 1]$, where $\sum_{a \in A} f_a = 1$ and each $f_a \geq 0$

That is: $f$ is a *probability distribution* on $A$
**Data Compression Notation**

If each \( a \in A \) occurs \( m_a \) times in \( C \) and \( m = \sum_{a \in A} m_a \) then

\[
EL(C) = \sum_{i=1}^{m} l_{c_i} = \sum_{a \in A} m_a l_a = m \sum_{a \in A} \frac{m_a}{m} l_a = m \sum_{a \in A} f_a l_a
\]

**Frequency Distribution:** \( f_a = \frac{m_a}{m} \).

Note: \( f : A \rightarrow [0, 1] \), where \( \sum_{a \in A} f_a = 1 \) and each \( f_a \geq 0 \)

That is: \( f \) is a *probability distribution* on \( A \)

**Encoding Length (text):** \( EL(C) = m \sum_{a \in A} f_a l_a \)
Data Compression Notation

If each $a \in A$ occurs $m_a$ times in $C$ and $m = \sum_{a \in A} m_a$ then

$$EL(C) = \sum_{i=1}^{m} l_{c_i} = \sum_{a \in A} m_a l_a = m \sum_{a \in A} \frac{m_a}{m} l_a = m \sum_{a \in A} f_a l_a$$

Frequency Distribution: $f_a = \frac{m_a}{m}$.

Note: $f : A \rightarrow [0, 1]$, where $\sum_{a \in A} f_a = 1$ and each $f_a \geq 0$

That is: $f$ is a probability distribution on $A$

Encoding Length (text): $EL(C) = m \sum_{a \in A} f_a l_a$

Average Encoding Length (of a symbol in $A$): $AEL(f, \gamma) = \sum_{a \in A} f_a l_a$
Data Compression Notation

If each \( a \in A \) occurs \( m_a \) times in \( C \) and \( m = \sum_{a \in A} m_a \) then

\[
EL(C) = \sum_{i=1}^{m} l_{c_i} = \sum_{a \in A} m_a l_a = m \sum_{a \in A} \frac{m_a}{m} l_a = m \sum_{a \in A} f_a l_a
\]

Frequency Distribution: \( f_a = \frac{m_a}{m} \).

Note: \( f : A \rightarrow [0, 1] \), where \( \sum_{a \in A} f_a = 1 \) and each \( f_a \geq 0 \)

That is: \( f \) is a probability distribution on \( A \)

Encoding Length (text): \( EL(C) = m \sum_{a \in A} f_a l_a \)

Average Encoding Length (of a symbol in \( A \)): \( AEL(f, \gamma) = \sum_{a \in A} f_a l_a \)

Goal: Minimize \( EL \) or, equivalently, \( AEL \)
Uniquely Decodable Encodings

Definition
An encoding $\gamma : A \rightarrow B^+$ satisfies the prefix property if for all $a, a' \in A$, $\gamma(a)$ is not a prefix of $\gamma(a')$. 

Note: Morse Code doesn't have this property

Goal: For a frequency distribution $f$, find optimal encoding having prefix property

• Optimal encoding: Minimizes $AEL(f, \gamma)$.
• Encoding with prefix property corresponds to encoding tree with all $a \in A$ labeling leaves.
• $l_a = \text{depth}(a)$ in $T$.
• Optimal trees don't have internal node with only one child (called full trees).
Uniquely Decodable Encodings

Definition

An encoding $\gamma : A \rightarrow B^+$ satisfies the prefix property if for all $a, a' \in A$, $\gamma(a)$ is not a prefix of $\gamma(a')$.

Note: Morse Code doesn’t have this property
Uniquely Decodable Encodings

Definition

An encoding $\gamma : A \rightarrow B^+$ satisfies the prefix property if for all $a, a' \in A$, $\gamma(a)$ is not a prefix of $\gamma(a')$.

Note: Morse Code doesn’t have this property

Goal: For a frequency distribution $f$, find optimal encoding having prefix property
Uniquely Decodable Encodings

**Definition**
An encoding $\gamma : A \rightarrow B^+$ satisfies the *prefix property* if for all $a, a' \in A$, $\gamma(a)$ is not a prefix of $\gamma(a')$.

**Note:** Morse Code doesn’t have this property

**Goal:** For a frequency distribution $f$, find *optimal* encoding having prefix property

- Optimal encoding: Minimizes $AEL(f, \gamma)$. 

Uniquely Decodable Encodings

**Definition**
An encoding $\gamma : A \rightarrow B^+$ satisfies the *prefix property* if for all $a, a' \in A$, $\gamma(a)$ is not a prefix of $\gamma(a')$.

**Note:** Morse Code doesn’t have this property

**Goal:** For a frequency distribution $f$, find *optimal* encoding having prefix property

- Optimal encoding: Minimizes $AEL(f, \gamma)$.
- Encoding with *prefix property* corresponds to *encoding tree* with all $a \in A$ labeling leaves
Uniquely Decodable Encodings

**Definition**
An encoding $\gamma : A \rightarrow B^+$ satisfies the *prefix property* if for all $a, a' \in A$, $\gamma(a)$ is not a prefix of $\gamma(a')$.

**Note:** Morse Code doesn’t have this property

**Goal:** For a frequency distribution $f$, find *optimal* encoding having prefix property

- Optimal encoding: Minimizes $AEL(f, \gamma)$.
- Encoding with *prefix property* corresponds to *encoding tree* with all $a \in A$ labeling leaves
- $l_a = depth(a)$ in $T$. 
Uniquely Decodable Encodings

Definition
An encoding $\gamma : A \rightarrow B^+$ satisfies the prefix property if for all $a, a' \in A$, $\gamma(a)$ is not a prefix of $\gamma(a')$.

Note: Morse Code doesn’t have this property

Goal: For a frequency distribution $f$, find optimal encoding having prefix property

- Optimal encoding: Minimizes $AEL(f, \gamma)$.
- Encoding with prefix property corresponds to encoding tree with all $a \in A$ labeling leaves
- $l_a = \text{depth}(a)$ in $T$.
- Optimal trees don’t have internal node with only one child (called full trees).
Stating the Problem

For now, assume that $B = \{0, 1\}$
Stating the Problem

For now, assume that \( B = \{0, 1\} \)

**Goal:** Given a frequency distribution \( f \) on \( A \), find encoding function \( \gamma : A \rightarrow \{0, 1\}^+ \) that minimizes \( AEL(f, \gamma) \) over all encodings with prefix property.
Stating the Problem

For now, assume that $B = \{0, 1\}$

**Goal:** Given a frequency distribution $f$ on $A$, find encoding function $\gamma : A \rightarrow \{0, 1\}^+$ that minimizes $AEL(f, \gamma)$ over all encodings with prefix property.

Or, stated in purely graph-theoretic form
Stating the Problem

For now, assume that \( B = \{0, 1\} \)

**Goal:** Given a frequency distribution \( f \) on \( A \), find encoding function \( \gamma : A \to \{0, 1\}^+ \) that minimizes \( AEL(f, \gamma) \) over all encodings with prefix property.

Or, stated in purely graph-theoretic form

**Definition:** Given a frequency distribution \( f = \{f_1, \ldots, f_n\} \), \( T \) is an **encoding tree for** \( f \) if
Stating the Problem

For now, assume that \( B = \{0, 1\} \)

**Goal:** Given a frequency distribution \( f \) on \( A \), find encoding function \( \gamma : A \to \{0, 1\}^+ \) that minimizes \( AEL(f, \gamma) \) over all encodings with prefix property.

Or, stated in purely graph-theoretic form

**Definition:** Given a frequency distribution \( f = \{f_1, \ldots, f_n\} \), \( T \) is an *encoding tree for \( f \)* if

- \( T \) is a full binary tree with \( n \) leaves
Stating the Problem

For now, assume that $B = \{0, 1\}$

**Goal:** Given a frequency distribution $f$ on $A$, find encoding function $\gamma : A \rightarrow \{0, 1\}^+$ that minimizes $AEL(f, \gamma)$ over all encodings with prefix property.

Or, stated in purely graph-theoretic form

**Definition:** Given a frequency distribution $f = \{f_1, \ldots, f_n\}$, $T$ is an *encoding tree for $f$* if

- $T$ is a full binary tree with $n$ leaves
- The leaves of $T$ have been given an ordering $v_1, \ldots, v_n$
Stating the Problem

For now, assume that $B = \{0, 1\}$

**Goal:** Given a frequency distribution $f$ on $A$, find encoding function $\gamma : A \rightarrow \{0, 1\}^+$ that minimizes $AEL(f, \gamma)$ over all encodings with prefix property.

Or, stated in purely graph-theoretic form

**Definition:** Given a frequency distribution $f = \{f_1, \ldots, f_n\}$, $T$ is an encoding tree for $f$ if

- $T$ is a full binary tree with $n$ leaves
- The leaves of $T$ have been given an ordering $v_1, \ldots, v_n$

Given an encoding tree $T$ for $f$, let $AEL(f, T) = \sum_{i=1}^{n} f_i \text{depth}(v_i)$. 
Stating the Problem

For now, assume that $B = \{0, 1\}$

**Goal:** Given a frequency distribution $f$ on $A$, find encoding function $\gamma : A \rightarrow \{0, 1\}^+$ that minimizes $AEL(f, \gamma)$ over all encodings with prefix property.

Or, stated in purely graph-theoretic form

**Definition:** Given a frequency distribution $f = \{f_1, \ldots, f_n\}$, $T$ is an encoding tree for $f$ if

- $T$ is a full binary tree with $n$ leaves
- The leaves of $T$ have been given an ordering $v_1, \ldots, v_n$

Given an encoding tree $T$ for $f$, let $AEL(f, T) = \sum_{i=1}^{n} f_i \text{depth}(v_i)$.

**Goal:** Given a frequency distribution $f = \{f_1, \ldots, f_n\}$, find encoding tree $T$ minimizing $AEL(f, T)$
Lemma

If \( f_x < f_y \) and depth \((x)\) < depth \((y)\) in some encoding tree \( T \), then there is a \( T' \) with lower AEL.

Consequences

• If \( f_x < f_y \) for some optimal tree \( T \) for \( f \), then \( \text{depth}(x) \geq \text{depth}(y) \).

• Swapping a leaf \( x \) with a deeper leaf \( y \) when \( f_x \leq f_y \) never increases AEL.

Lemma

For any positive frequency distribution \( f \), and any two \( f_i, f_j \) of lowest frequencies, there is an optimal encoding tree \( T \) in which \( f_i, f_j \) are labels of siblings \( x \) and \( y \) in \( T \).
Tools

Lemma
If \( f_x < f_y \) and \( \text{depth}(x) < \text{depth}(y) \) in some encoding tree \( T \), then there is a \( T' \) with lower AEL.
Tools

Lemma
If \( f_x < f_y \) and \( \text{depth}(x) < \text{depth}(y) \) in some encoding tree \( T \), then there is a \( T' \) with lower AEL.

Consequences
Tools

Lemma
If $f_x < f_y$ and $\text{depth}(x) < \text{depth}(y)$ in some encoding tree $T$, then there is a $T'$ with lower AEL.

Consequences
- If $f_x < f_y$ for some optimal tree $T$ for $f$, then $\text{depth}(x) \geq \text{depth}(y)$. 

Tools

Lemma
If \( f_x < f_y \) and \( \text{depth}(x) < \text{depth}(y) \) in some encoding tree \( T \), then there is a \( T' \) with lower AEL.

Consequences

- If \( f_x < f_y \) for some optimal tree \( T \) for \( f \), then \( \text{depth}(x) \geq \text{depth}(y) \).
- Swapping a leaf \( x \) with a deeper leaf \( y \) when \( f_x \leq f_y \) never increases AEL.
Tools

**Lemma**

If \( f_x < f_y \) and depth(x) < depth(y) in some encoding tree \( T \), then there is a \( T' \) with lower AEL.

**Consequences**

- If \( f_x < f_y \) for some optimal tree \( T \) for \( f \), then depth(x) \( \geq \) depth(y).
- Swapping a leaf \( x \) with a deeper leaf \( y \) when \( f_x \leq f_y \) never increases AEL.

**Lemma**

For any positive frequency distribution \( f \), and any two \( f_i, f_j \) of lowest frequencies, there is an optimal encoding tree \( T \) in which \( f_i, f_j \) are labels of siblings \( x \) and \( y \) in \( T \).
Huffman’s Algorithm

Algorithm 1 Huffman Encoding

procedure Huffman((A, f))
    if |A| = 2 then
        return a tree with one letter encoded by 0 and the other by 1
    else
        Select 2 lowest-frequency symbols x, y ∈ A
        Delete x and y from A
        Add xy to A with frequency f_{xy} = f_x + f_y
        T = Huffman(A, f)
        Replace leaf xy of T with a node having leaves x and y and edges labeled 0/1 respectively
        return T
end procedure
Correctness of Huffman

Proof: induction on $n = |A|$. We assume that Huffman produces an optimal encoding for all alphabets of length less than $n$. Now assume $|A| = n$. Use proof by contradiction. Make the following points:
Correctness of Huffman

Proof: induction on $n = |A|$. We assume that Huffman produces an optimal encoding for all alphabets of length less than $n$. Now assume $|A| = n$. Use proof by contradiction. Make the following points:

- Let $T$ be the tree produced by Huffman and let $x$ and $y$ be the first two symbols it identifies.
Correctness of Huffman

Proof: induction on $n = |A|$. We assume that Huffman produces an optimal encoding for all alphabets of length less than $n$. Now assume $|A| = n$. Use proof by contradiction.

Make the following points:

- Let $T$ be the tree produced by Huffman and let $x$ and $y$ be the first two symbols it identifies.
- Let $\bar{T}$ be the tree Huffman produces after replacing $x$ and $y$ with new symbol $xy$ having frequency $f_{xy} = f_x + f_y$. 
Correctness of Huffman

Proof: induction on $n = |A|$. We assume that Huffman produces an optimal encoding for all alphabets of length less than $n$. Now assume $|A| = n$. Use proof by contradiction. Make the following points:

- Let $T$ be the tree produced by Huffman and let $x$ and $y$ be the first two symbols it identifies.
- Let $\bar{T}$ be the tree Huffman produces after replacing $x$ and $y$ with new symbol $xy$ having frequency $f_{xy} = f_x + f_y$.
- Let $T^*$ be an optimal tree in which $x$ and $y$ are siblings.
Correctness of Huffman

Proof: induction on $n = |A|$. We assume that Huffman produces an optimal encoding for all alphabets of length less than $n$. Now assume $|A| = n$. Use proof by contradiction. Make the following points:

- Let $T$ be the tree produced by Huffman and let $x$ and $y$ be the first two symbols it identifies.
- Let $\tilde{T}$ be the tree Huffman produces after replacing $x$ and $y$ with new symbol $xy$ having frequency $f_{xy} = f_x + f_y$.
- Let $T^*$ be an optimal tree in which $x$ and $y$ are siblings.
- Let $\tilde{T}^*$ be tree produced by replacing $x$ and $y$ in $T^*$ with $xy$ and $f_{xy} = f_x + f_y$. 
Correctness Proof Continued
Correctness Proof Continued

- Note: $AEL(T^*) = AEL(\tilde{T}^*) - (f_{xy} \ast l_{xy}^*) + (f_x + f_y)(l_{xy}^* + 1) = AEL(\tilde{T}^*) + (f_x + f_y)$
Correctness Proof Continued

- Note: $AEL(T^*) = AEL(\bar{T}^*) - (f_{xy} * l_{xy}^*) + (f_x + f_y)(l_{xy}^* + 1) = AEL(\bar{T}^*) + (f_x + f_y)$

- Similarly, $AEL(T) = AEL(\bar{T}) - (f_{xy} * l_{xy}) + (f_x + f_y)(l_{xy} + 1) = AEL(\bar{T}) + (f_x + f_y)$
Correctness Proof Continued

• Note: \( AEL(T^*) = AEL(\bar{T}^*) - (f_{xy} \ast l_{xy}^*) + (f_x + f_y)(l_{xy}^* + 1) = AEL(\bar{T}^*) + (f_x + f_y) \)

• Similarly, \( AEL(T) = AEL(\bar{T}) - (f_{xy} \ast l_{xy}) + (f_x + f_y)(l_{xy} + 1) = AEL(\bar{T}) + (f_x + f_y) \)

• Thus \( AEL(T^*) = AEL(\bar{T}^*) + (f_x + f_y) \geq AEL(\bar{T}) + (f_x + f_y) = AEL(T) \)
Complexity Analysis
Complexity Analysis

- Consider each element $a \in A$ to be represented by a single tree node
Complexity Analysis

- Consider each element $a \in A$ to be represented by a single tree node.
- When we merge two 'symbols', we merely create a new tree node with the two symbol trees as subtrees.
Complexity Analysis

- Consider each element $a \in A$ to be represented by a single tree node.
- When we merge two 'symbols', we merely create a new tree node with the two symbol trees as subtrees.
- This can be done in constant time, given the two 'symbol' trees.
**Complexity Analysis**

- Consider each element $a \in A$ to be represented by a single tree node.
- When we merge two ’symbols’, we merely create a new tree node with the two symbol trees as subtrees.
- This can be done in constant time, given the two ’symbol’ trees.
- How do we find two lowest-frequency symbol trees? Use a priority queue!
Consider each element \( a \in A \) to be represented by a single tree node

When we merge two ‘symbols’, we merely create a new tree node with the two symbol trees as subtrees.

This can be done in constant time, given the two ‘symbol’ trees.

How do we find two lowest-frequency symbol trees? Use a priority queue!

Thus the algorithm consists of repeatedly deleting two highest priority nodes, merging them and reinserting the result.
Complexity Analysis

- Consider each element $a \in A$ to be represented by a single tree node.
- When we merge two 'symbols', we merely create a new tree node with the two symbol trees as subtrees.
- This can be done in constant time, given the two 'symbol' trees.
- How do we find two lowest-frequency symbol trees? Use a priority queue!
- Thus the algorithm consists of repeatedly deleting two highest priority nodes, merging them and reinserting the result.

Thus: Huffman can be implemented in $O(n)$ space and $O(n \log n)$ time.