Today:
- Huffman Codes
- Data Structures for Disjoint Sets

COSC 581, Algorithms

February 20, 2014

Many of these slides are adapted from several online sources

Reading Assignments

- Today’s class:
  - Chapter 16.3, Ch. 21
- Reading assignment for next class:
  - Chapter 23

Huffman Coding

- Huffman codes can be used to compress information (savings of 20% to 90%)
  - Like WinZip – although WinZip doesn't use the Huffman algorithm
  - .mp3 and .jpg file formats use Huffman coding at one stage of the compression
- The basic idea is that instead of storing each character in a file as an 8-bit ASCII value, we will instead store the more frequently occurring characters using fewer bits and less frequently occurring characters using more bits
  - On average this should decrease the file size (usually ½)

Huffman Coding Problem

- The more frequent a symbol occurs, the shorter should be the Huffman binary word representing it.

- The Huffman code is a prefix code.
  - No prefix of a code word is equal to another codeword.

Prefix Code

- Prefix(-free) code: no codeword is also a prefix of some other codewords (Un-ambiguous)
  - An optimal data compression achievable by a character code can always be achieved with a prefix code
  - Simplify the encoding (compression) and decoding
    - Encoding: abc ➔ 0 . 101. 100 = 0101100
    - Decoding: 001011101 = 0. 0. 101. 1101 ➔ aabe
      - Use binary tree to represent prefix codes for easy decoding
- An optimal code is always represented by a full binary tree, in which every non-leaf node has two children
  - |C| leaves and |C|-1 internal nodes Cost:

$$B(T) = \sum_{c \in C} \underbrace{f(c)}_{\text{Frequency of c}} \underbrace{\boxed{d_T(c)}}_{\text{Depth of c (length of the codeword)}}$$

Decode the following

E	0
T	11
N	100
I	1010
S	1011

1101001001010101011

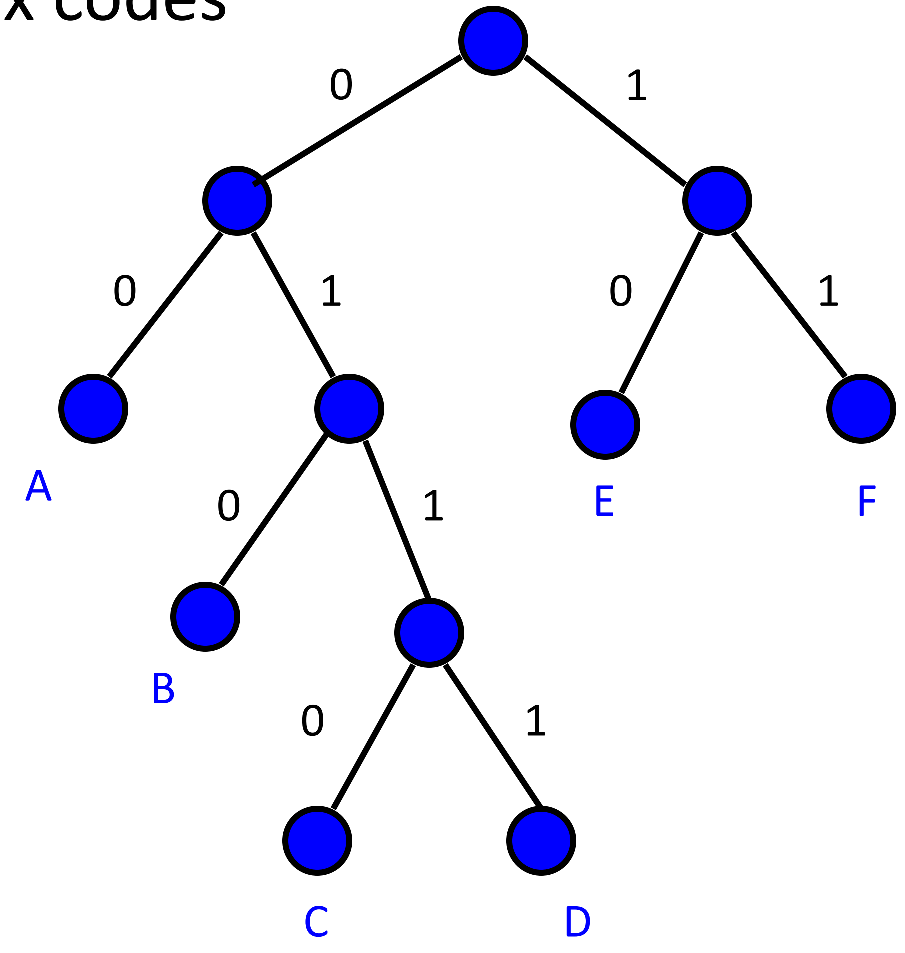
E	0
T	10
N	100
I	0111
S	1010

100100101010

Prefix codes and binary trees

Tree representation of prefix codes

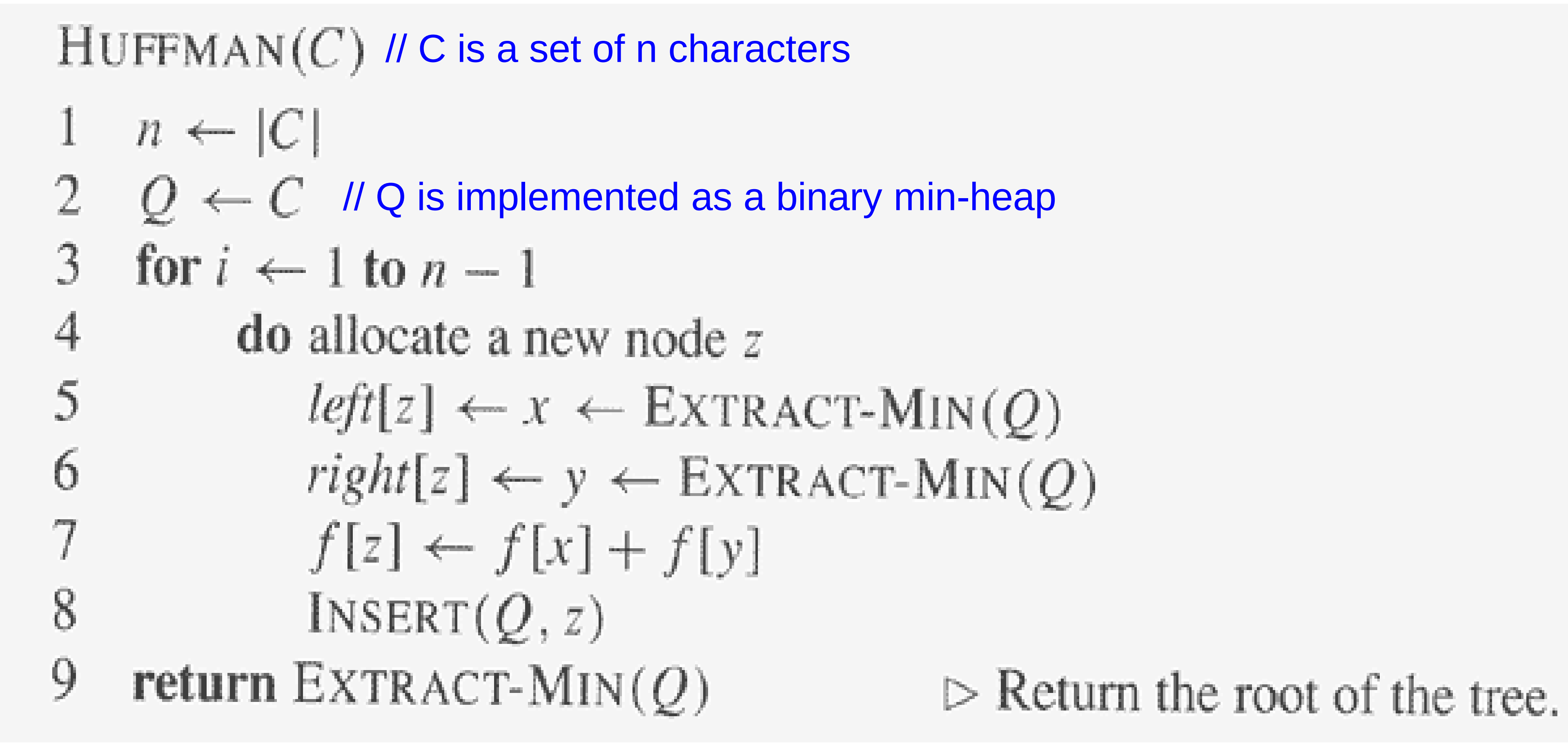
A	00
B	010
C	0110
D	0111
E	10
F	11



Idea for Building a Huffman Tree

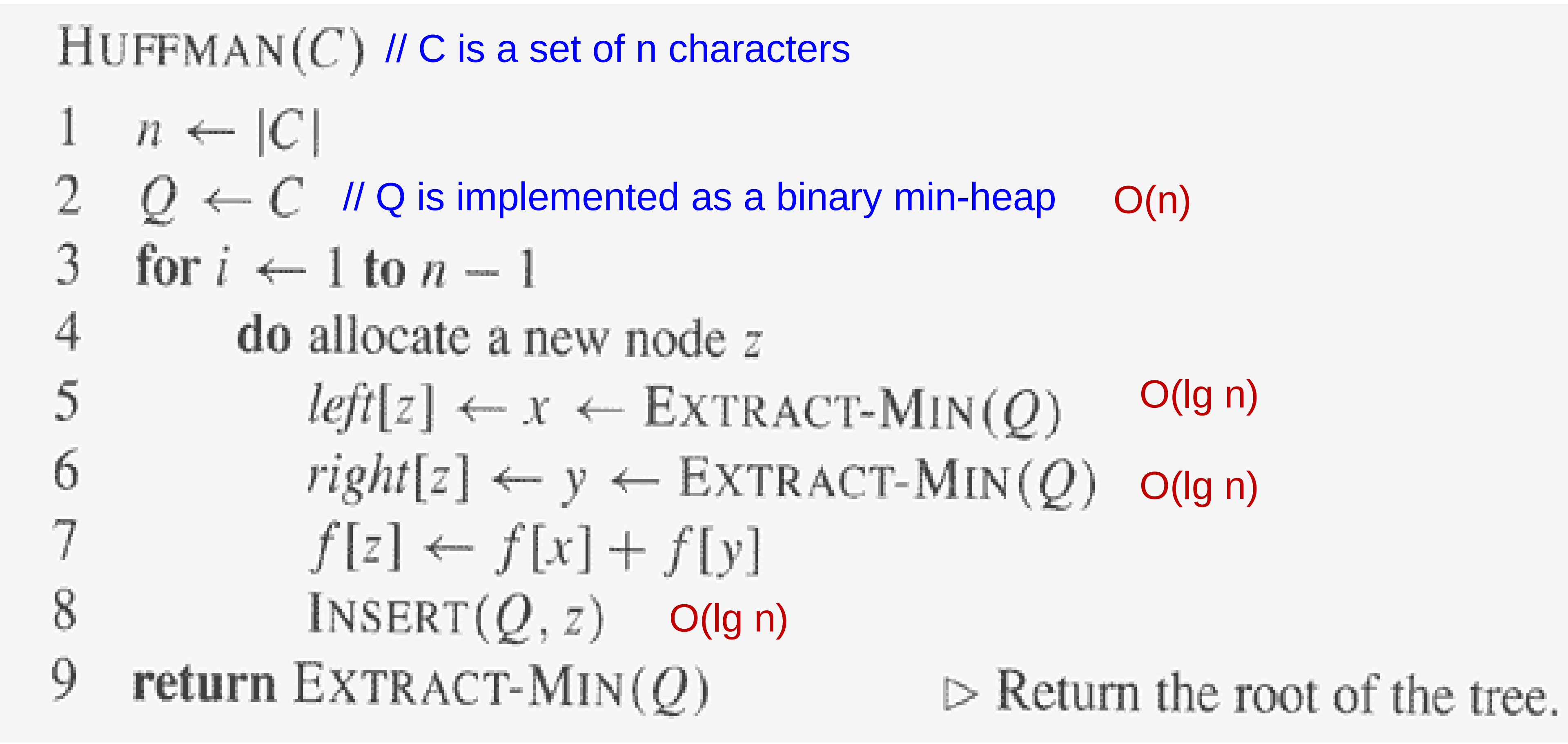
1. **Compute the frequencies** of each character in the alphabet
2. **Build a tree forest with one-node trees**, where each node corresponds to a character and contains the frequency of the character in the text to be encoded
3. **Select two** parentless nodes with the **lowest frequency**
4. **Create a new node** which is the parent of the two lowest frequency nodes.
5. **Label** the **left link with 0** and the **right link with 1**
6. **Assign** the new node a **frequency equal to the sum** of its children's frequencies.
7. **Repeat Steps 3 through 6** until there is only one parentless node left.

Huffman Coding Algorithm



```
HUFFMAN($C$)   // C is a set of n characters
1  $n \leftarrow |C|$
2  $Q \leftarrow C$   // Q is implemented as a binary min-heap
3  for $i \leftarrow 1$ to $n - 1$
4       do allocate a new node $z$
5          $left[z] \leftarrow x \leftarrow$ EXTRACT-MIN($Q$)
6          $right[z] \leftarrow y \leftarrow$ EXTRACT-MIN($Q$)
7          $f[z] \leftarrow f[x] + f[y]$
8          INSERT($Q, z$)
9  return EXTRACT-MIN($Q$)          ▷ Return the root of the tree.
```

Huffman Coding Algorithm



```
HUFFMAN($C$)  // C is a set of n characters
$n \leftarrow |C|$
$Q \leftarrow C$   // Q is implemented as a binary min-heap   O(n)
for $i \leftarrow 1$ to $n - 1$
    do allocate a new node $z$
       $left[z] \leftarrow x \leftarrow$ EXTRACT-MIN($Q$)   O(lg n)
       $right[z] \leftarrow y \leftarrow$ EXTRACT-MIN($Q$)   O(lg n)
       $f[z] \leftarrow f[x] + f[y]$
       INSERT($Q, z$)   O(lg n)
return EXTRACT-MIN($Q$)        $\triangleright$ Return the root of the tree.
```

Total computation time = O(*n* lg *n*)

Huffman Algorithm correctness:

Need to prove two things:

(1) Greedy Choice Property:

There exists a minimum cost prefix tree where the two smallest frequency characters are indeed siblings with the longest path from root.

This means that the greedy choice does not hurt finding the optimum.

Algorithm correctness:

(2) Optimal Substructure Property:

The optimal solution to the subproblem, combined with the greedy choice (i.e., the choice of the two least frequent elements), leads to an optimal solution to the original problem.

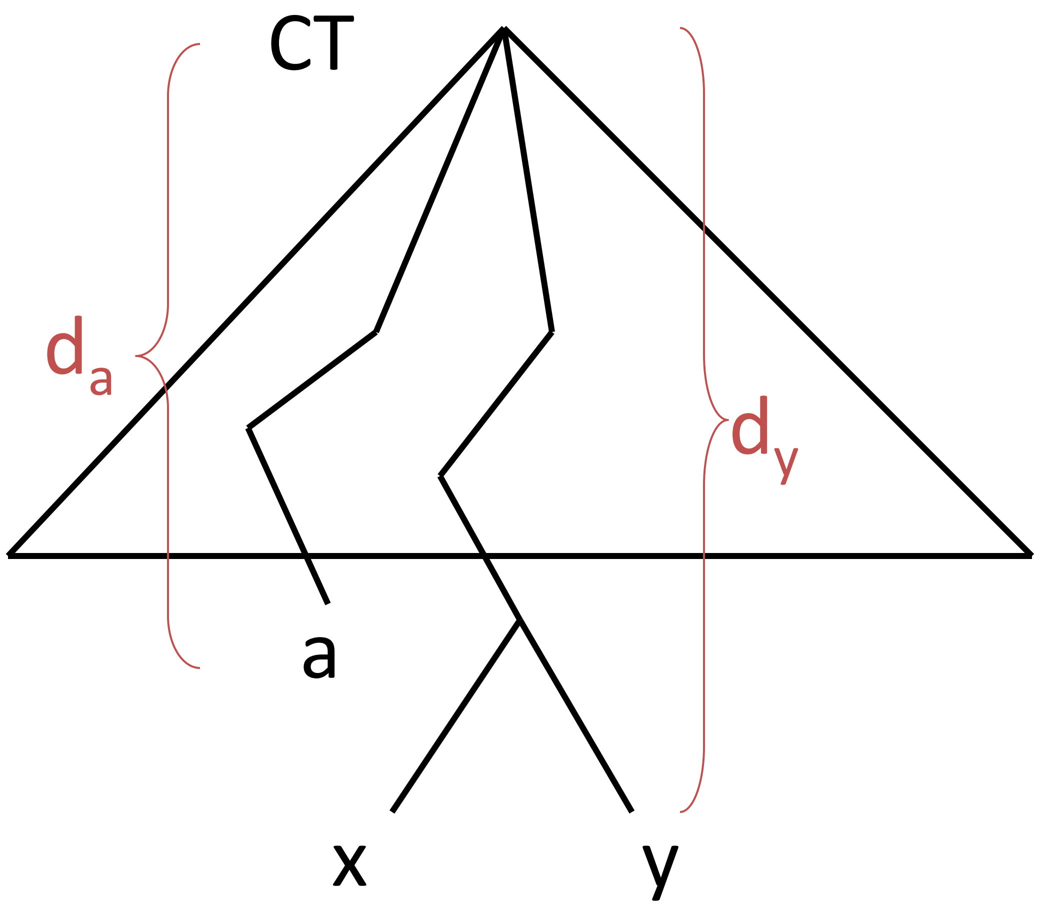
Algorithm correctness:

(1) Greedy Choice Property:
There exists a minimum cost tree where the minimum frequency elements are longest path siblings:

Assume that is not the situation.
Then there are two elements in the longest path.

Say a, b are the elements with smallest frequency and x, y the elements in the longest path.

Algorithm correctness:

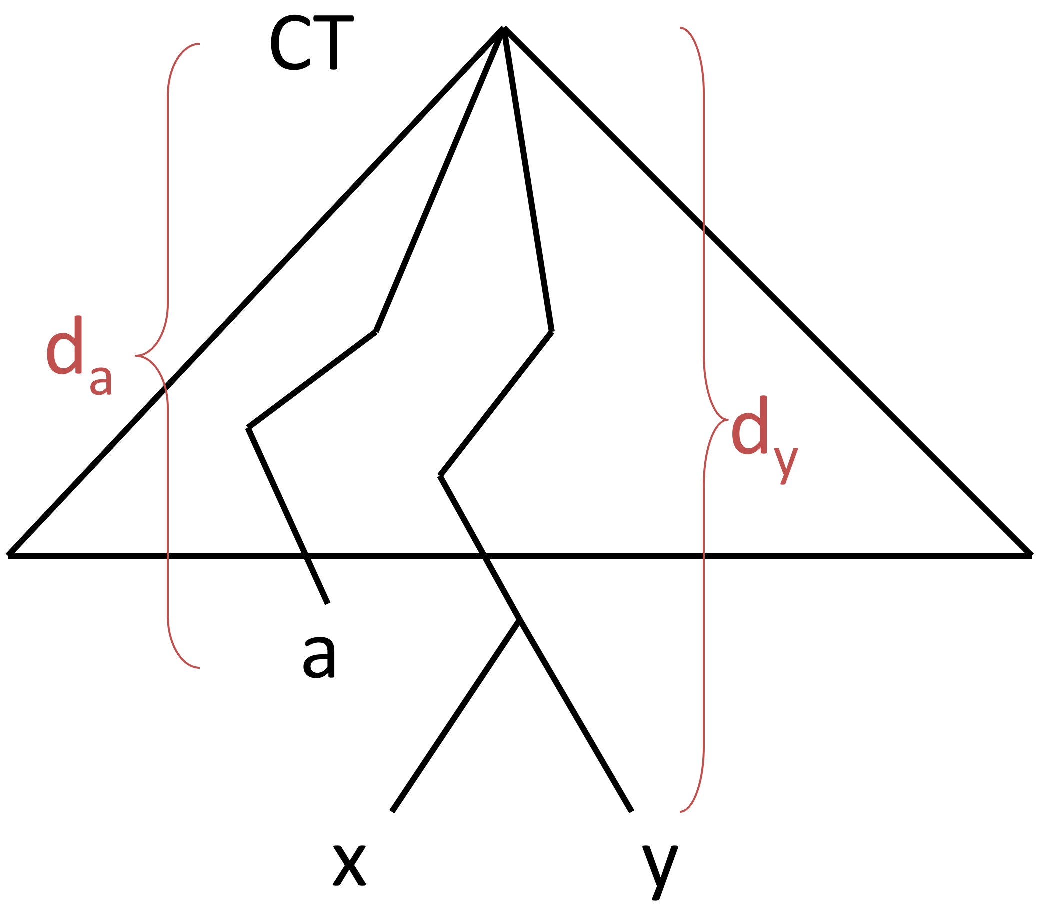


We know about depth and frequency:

$d_a \leq d_y$

$f_a \leq f_y$

Algorithm correctness:



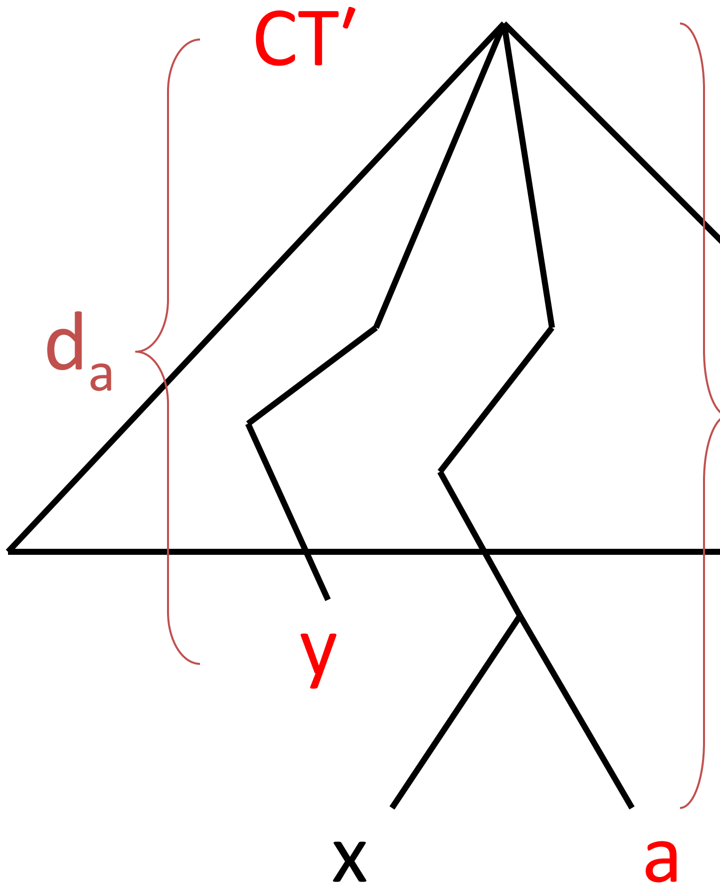
We also know about code tree CT:

$$\sum_{c \in C} f_c d_c$$

is smallest possible.

Now exchange a and y.

Algorithm correctness:



$$\text{Cost}(CT) = \sum_{c \in C} f_c d_c =$$

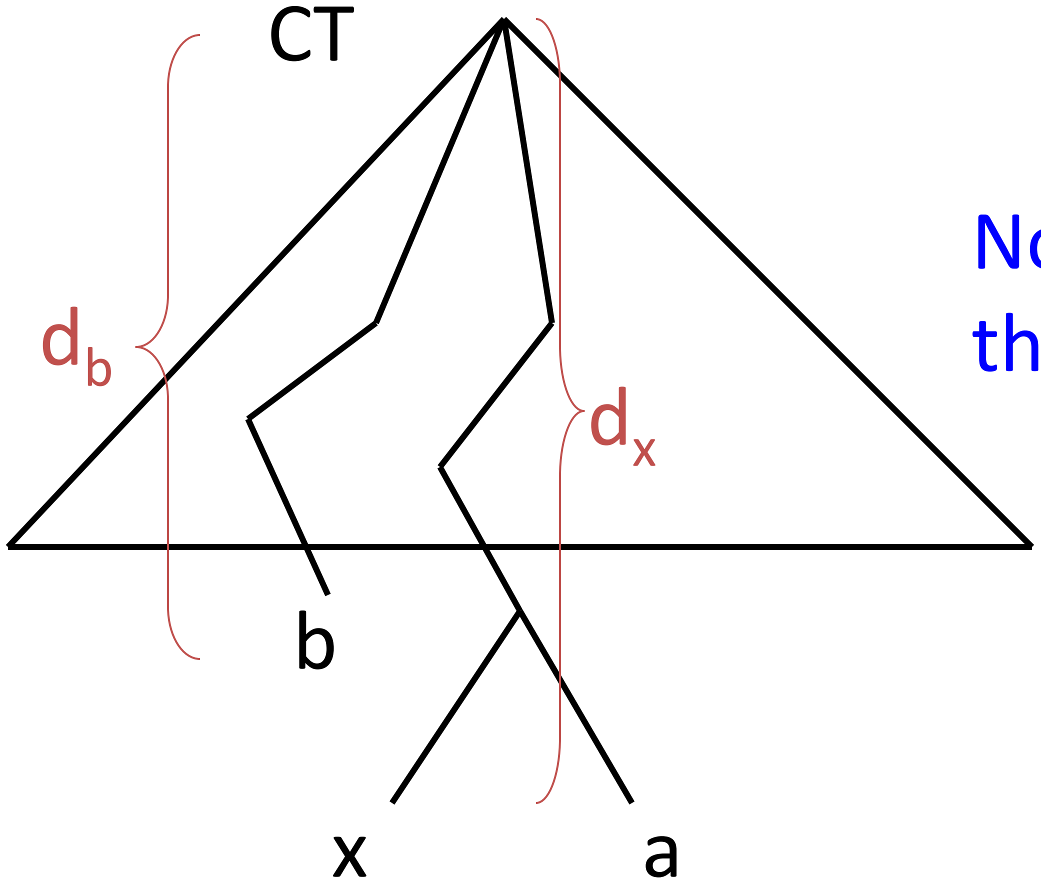
$$\sum_{c \in C \neq a,y} f_c d_c + f_a d_a + f_y d_y \geq$$

($d_a \leq d_y$, $f_a \leq f_y$
Therefore
$f_a d_a \geq f_y d_a$ and
$f_y d_y \geq f_a d_y$)

$$\sum_{c \in C \neq a,y} f_c d_c + f_y d_a + f_a d_y =$$

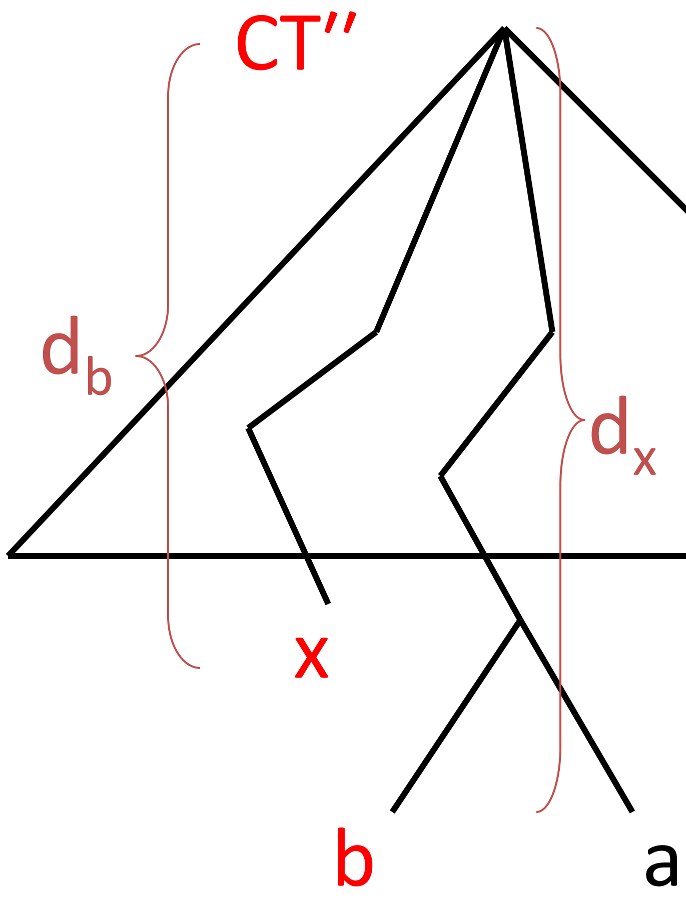
$$\text{cost}(CT')$$

Algorithm correctness:



Now do the same thing for b and x

Algorithm correctness:



And get an optimal code tree where a and b are siblings with the longest paths ■

Algorithm correctness:

Optimal substructure property:

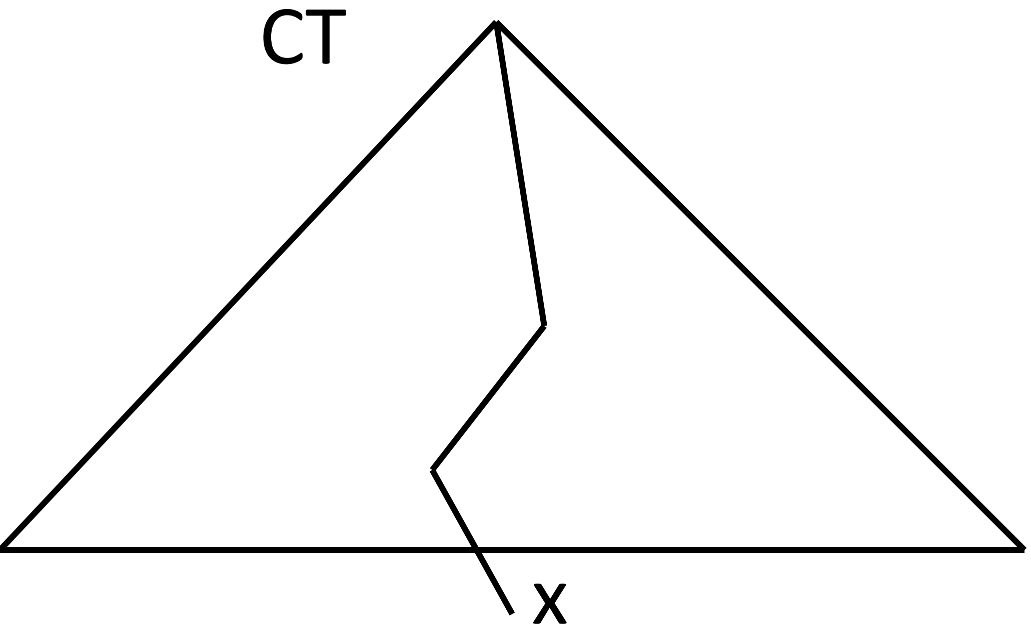
Let a,b be the symbols with the smallest frequency.

Let x be a new symbol whose frequency is $f_x = f_a + f_b$.

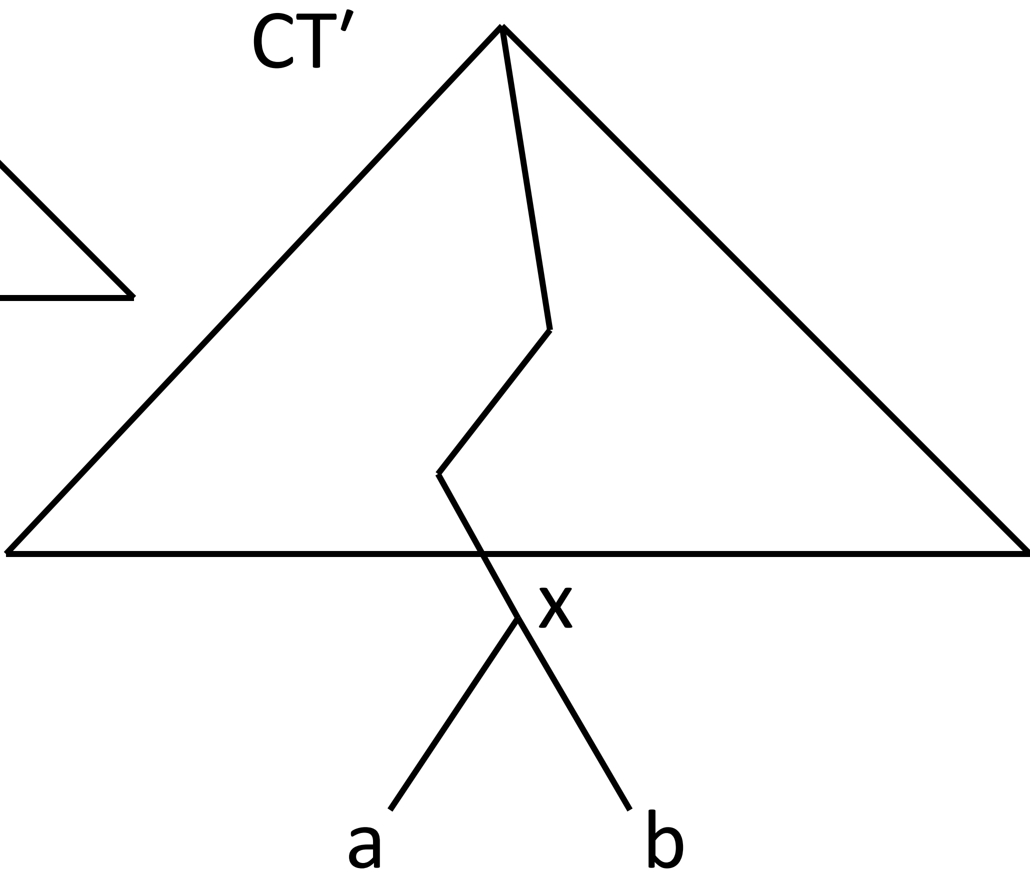
Delete characters a and b, and find the optimal code tree CT for the reduced alphabet.

Then CT' = CT ∪ {a,b} is an optimal tree for the original alphabet.

Algorithm correctness:



$f_x = f_a + f_b$



Algorithm correctness:

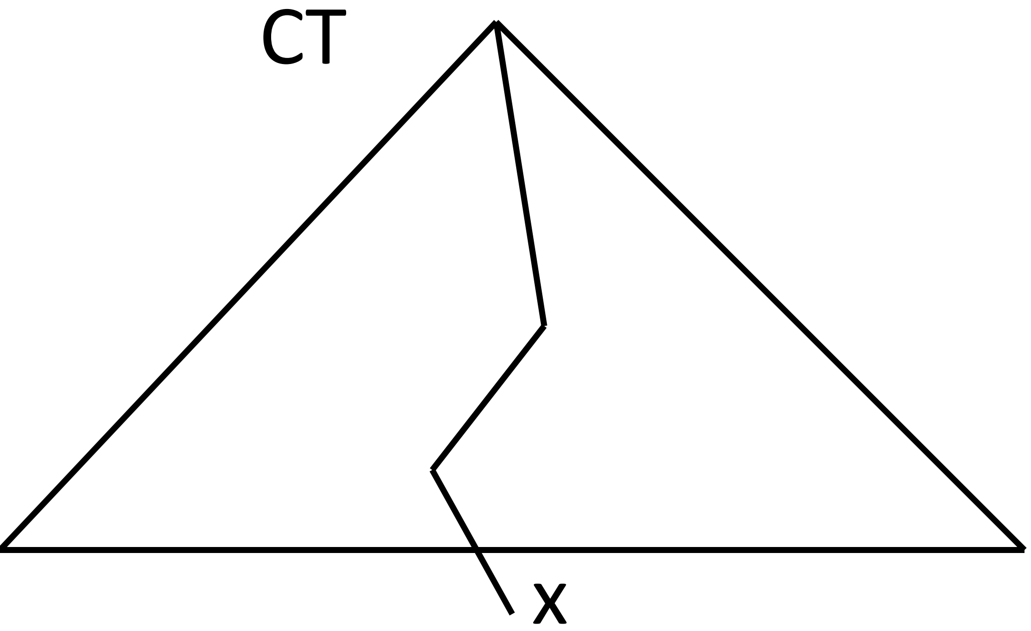
$$\text{cost}(CT')=\sum_{c\in C} f_c d'_c = \sum_{c\in C \neq a,b} f_c d'_c + f_a d'_a + f_b d'_b$$

$$= \sum_{c\in C \neq a,b} f_c d'_c + f_a(d_x+1) + f_b\,(d_x+1)$$

$$= \sum_{c\in C \neq a,b} f_c d'_c+(f_a + f_b)(d_x+1)$$

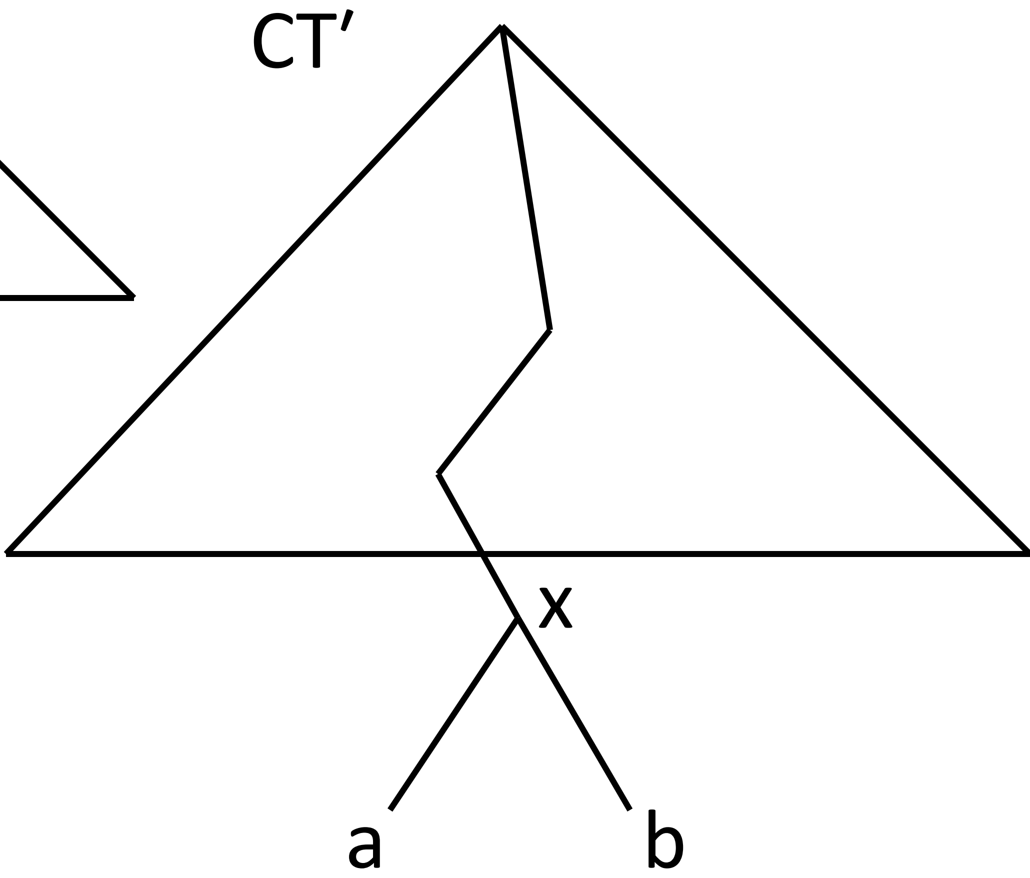
$$= \sum_{c\in C \neq a,b} f_c d_c+f_x(d_x+1)+f_x = \text{cost}(CT) + f_x$$

Algorithm correctness:



$f_x = f_a + f_b$

$\mathrm{cost}(CT)+f_x = \mathrm{cost}(CT')$



Algorithm correctness:

Assume CT’ is not optimal.

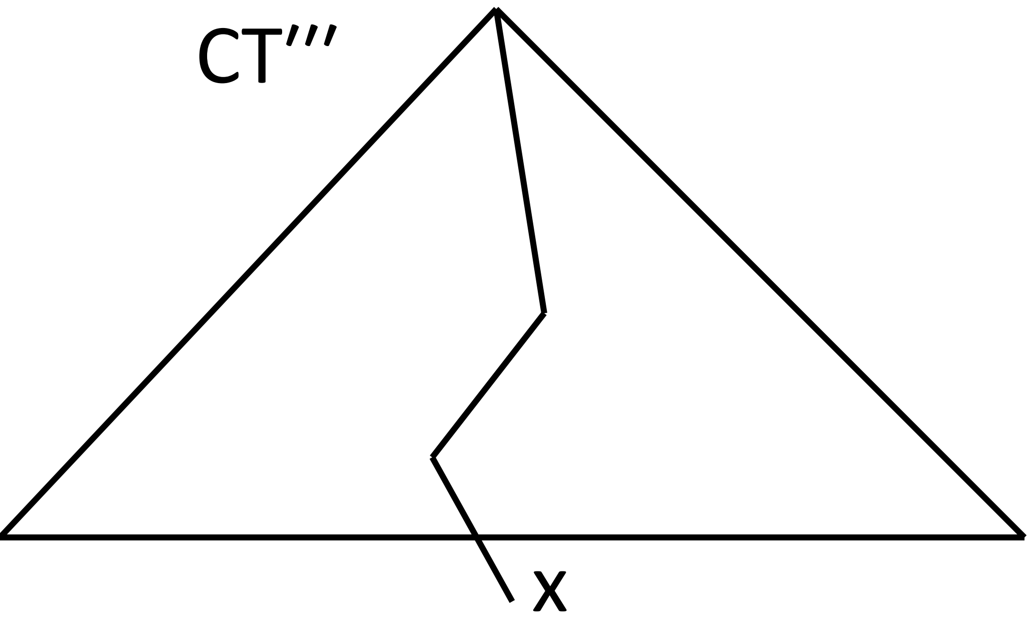
By the previous lemma there is a tree CT’’ that is optimal, and where a and b are siblings.

So:

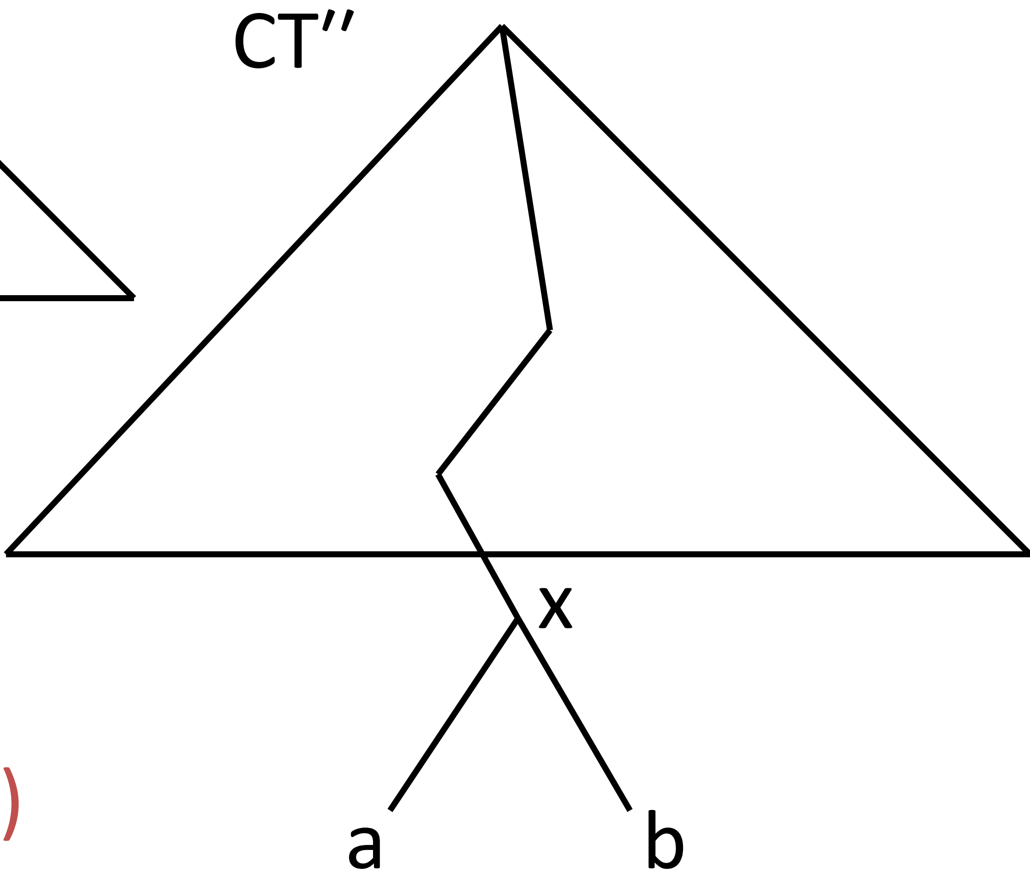
cost(CT’’) < cost(CT’)

Algorithm correctness:

Consider:



$f_x = f_a + f_b$



By a similar argument:

$\text{cost}(CT''') + f_x = \text{cost}(CT'')$

Algorithm correctness:

We get:

$$\begin{aligned} \text{cost}(CT''') &= \text{cost}(CT'') - f_x < \text{cost}(CT') - f_x \\ &= \text{cost}(CT) \end{aligned}$$

and this contradicts the minimality of cost(CT). ■

Disjoint Sets Data Structure (Chap. 21)

- A disjoint-set is a collection $S = \{S_1, S_2, \ldots, S_k\}$ of distinct dynamic sets.

- Each set is identified by a member of the set, called *representative*.

- Disjoint set operations:
  - MAKE-SET(x): create a new set with only x. assume x is not already in some other set.
  - UNION(x,y): combine the two sets containing x and y into one new set. A new representative is selected.
  - FIND-SET(x): return the representative of the set containing x.

Multiple Operations

- Suppose multiple operations:
  - n: #MAKE-SET operations (executed at beginning).
  - m: #MAKE-SET, UNION, FIND-SET operations.
  - $m \geq n$, #UNION operation is at most n-1.

An Application of Disjoint-Set

- Determine the connected components of an undirected graph.

```
CONNECTED-COMPONENTS(G)
1.  for each vertex v ∈ V[G]
2.      do MAKE-SET(v)
3.  for each edge (u,v) ∈ E[G]
4.      do if FIND-SET(u) ≠ FIND-SET(v)
5.          then UNION(u,v)
```

```
SAME-COMPONENT(u,v)
1.  if FIND-SET(u)=FIND-SET(v)
2.     then return TRUE
3.     else return FALSE
```

Disjoint-set Implementation: Forests

- Rooted trees, each tree is a set, root is the representative. Each node points to its parent. Root points to itself.

$$\begin{tikzcd}
 & c \arrow[loop above] & \\
h \arrow[ur] & & e \arrow[ul]
\end{tikzcd}$$

Set {*c*,*h*,*e*}

$$\begin{tikzcd}
f \arrow[loop above] \\
d \arrow[u]
\end{tikzcd}$$

Set {*f*,*d*}

$$\begin{tikzcd}
 & & f \arrow[loop above] & \\
 & c \arrow[ur] & & d \arrow[ul] \\
h \arrow[ur] & & e \arrow[ul] &
\end{tikzcd}$$

UNION

Operations

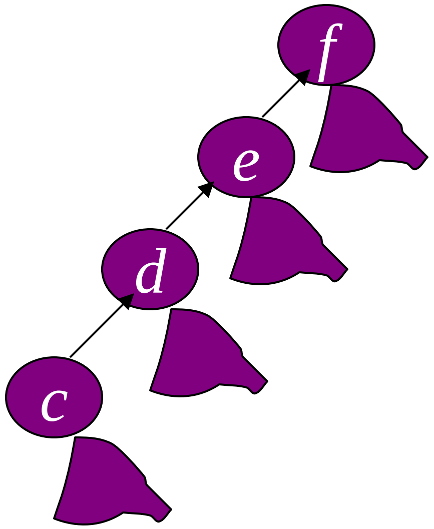
- Three operations:
  - MAKE-SET(x): create a tree containing x. $O(1)$
  - FIND-SET(x): follow the chain of parent pointers until to the root. O(height of x's tree)
  - UNION(x,y): let the root of one tree point to the root of the other. $O(1)$
- It is possible that n-1 UNIONs results in a tree of height n-1 (just a linear chain of n nodes).
- So n FIND-SET operations will cost $O(n^2)$.

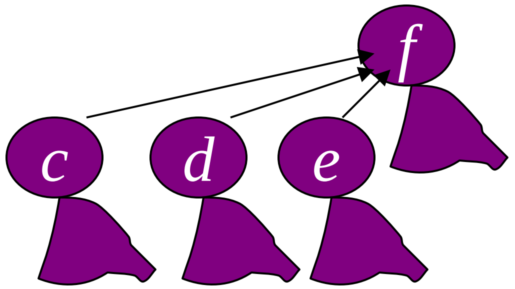
Union by Rank & Path Compression

- Union by Rank: Each node is associated with a rank, which is the upper bound on the height of the node (i.e., the height of subtree rooted at the node), then when UNION, let the root with smaller rank point to the root with larger rank.

- Path Compression: used in FIND-SET(*x*) operation, make each node in the path from *x* to the root directly point to the root. Thus reduce the tree height.

Path Compression





Algorithm for Disjoint-Set Forest

```
MAKE-SET($x$)
1.   $p[x] \leftarrow x$
2.   $rank[x] \leftarrow 0$
```

```
UNION($x,y$)
1. LINK(FIND-SET($x$),FIND-SET($y$))
```

```
LINK($x,y$)
1.   if $rank[x]>rank[y]$
2.   then $p[y] \leftarrow x$
3.   else  $p[x] \leftarrow y$
4.         if $rank[x]=rank[y]$
5.         then $rank[y]$++
```

```
FIND-SET($x$)
1.   if $x \neq p[x]$
2.      then $p[x] \leftarrow$FIND-SET($p[x]$)
3.   return $p[x]$
```

Worst case running time for *m* MAKE-SET, UNION, FIND-SET operations is:
$O(m\alpha(n))$ where $\alpha(n)\leq 4$. So nearly linear in *m (but not actually linear!)*.

Understanding $\alpha(n)$… Inverse is $A_k(n)$

- $A_k(j) = \begin{cases} j+1 & \text{if } k=0 \\ A_{k-1}^{(j+1)}(j) & \text{if } k \geq 1 \end{cases}$
- Examine $A_k(1)$: for k=0,1,2,3,4:

  $A_0(1)=1+1=2$

  $A_1(1)=2\cdot 1+1=3$

  $A_2(1)=2^{1+1}(1+1)-1=7$

  $A_3(1)=A_2^{(1+1)}(1)=A_2^{(2)}(1)=A_2(A_2(1))=A_2(7)=2^{7+1}(7+1)-1=2^8\cdot 8-1=2047$

  $A_4(1)=A_3^2(1)=A_3(A_3(1)) =A_3(2047)=A_2^{(2048)}(2047)$

  $\gg A_2(2047) =2^{2048}.2048-1 >2^{2048} =(2^4)^{512} =(16)^{512}$

  $\gg$ **10^{80}** (estimated number of atoms in universe!)

Inverse of $A_k(n)$: $\alpha(n)$

$\alpha(n) = \min\{k: A_k(1) \geq n\}$ (so, $A_{\alpha(n)}(1) \geq n$)

$$\alpha(n)=\begin{cases} 0 & \text{for } 0\leq n \leq 2 \\ 1 & \text{for } n = 3 \\ 2 & \text{for } 4\leq n \leq 7 \\ 3 & \text{for } 8\leq n \leq 2047 \\ 4 & \text{for } 2048\leq n \leq A_4(1). \end{cases}$$

Extremely slow increasing function.

$\alpha(n) \leq 4$ for all practical purposes.

Reading Assignments

- Reading assignment for next class:
  - Chapter 23