Analysis of Algorithms

V. Adamchik CSCI 570 Fall 2016

Lecture 3 University of Southern California

# Binary Heaps
# Huffman Codes
# MST
# Union-Find

## The Money Changing Problem

You are to compute the minimum number of coins needed to make change for a given amount.

## Example

Suppose we have an unlimited supply of nickels, dimes and quarters. What is the number of coins needed to make change for $0.40?

40 = 0*25 + 2*10 + 4*5

40 = 0*25 + 4*10 + 0*5

What is the minimum number of coins?

40 = 1*25 + 1*10 + 1*5

## The Algorithm

40 = 1*25 + 1*10 + 1*5

We always start with the largest coin and use it as many times as we can;

then we use the second largest coin, and so on.

This is a so-called greedy algorithm.

## Greedy Algorithm

- It is used to solve optimization problems
- It makes a local optimal choice at each step
- Earlier decisions are never undone
- Do not always yield optimal solutions

Now, you have to think about this question:

where does efficiency come from?

## Greedy Algorithm

Greedy Algorithm does not always yield optimal solutions.

Example:

Let coins be 5, 10, and 25.
Make a change of 40 cents. 40 = 25 + 10 + 5

Let coins be 5, 10, 20 and 25.
Make a change of 40 cents. 40 = 20 + 20

## Scheduling Problem

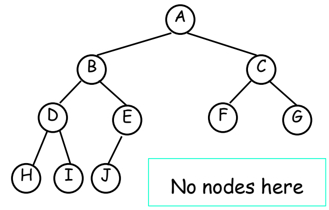
In the scheduling problem (from the previous lecture) we use sorting to solve the optimization problem.

Sorting in general could be expensive, especially if your data changes during the algorithm execution.

In such problems, we use <u>heaps</u> to represent almost ordered data.

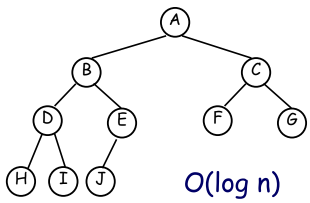
## Terminology: <u>complete</u> binary tree

completely filled, except the bottom level that is filled from left to right




## Complete Binary Tree

What is the height of a complete binary tree with n nodes?




O(log n)

## Binary Heaps

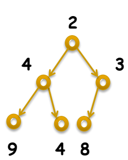
A heap is a complete binary tree which satisfies the heap ordering property.

<u>the min-heap property</u>: the value of each node is greater than or equal to the value of its parent

<u>the max-heap property</u>: the value of each node is less than or equal to the value of its parent,

## Binary Heap

The word "heap" will always refer to a min-heap, unless otherwise noted.




## Not a Heap

## Binary Heap Invariants

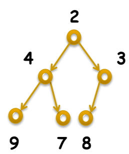
1. Structure Property
2. Ordering Property

## Heap Operations

insert
deleteMin
decreaseKey
build
meld

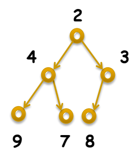
## Implementation

A heap tree is uniquely represented by storing it in an <u>array</u>.




| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| | 2 | 4 | 3 | 9 | 7 | 8 | | |

## Implementation



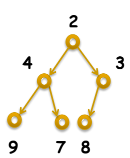

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| | 2 | 4 | 3 | 9 | 7 | 8 | | |

Consider k-th element of the array,

- its left child is located at 2*k index
- its right child is located at 2*k+1 index
- its parent is located at k/2 index

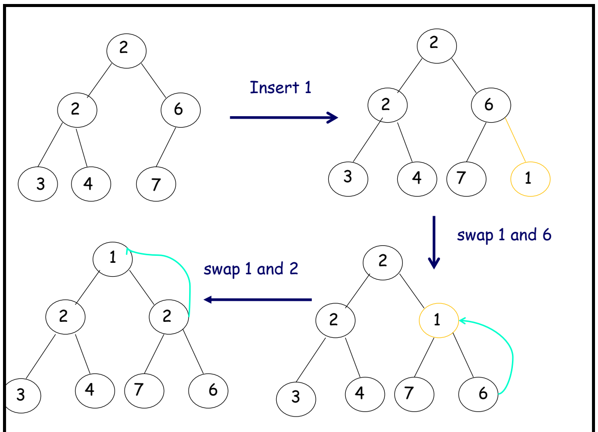
## Insert

The new element is initially appended to the end of the heap array.



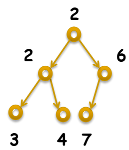

This will preserve the structure property.

Then we percolate it up by swapping positions with the parent, if it's necessary.



## Implementation

Insert 1.




| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| | 2 | 2 | 6 | 3 | 4 | 7 | 1 | |
| | 2 | 2 | 1 | 3 | 4 | 7 | 6 | |
| | 1 | 2 | 2 | 3 | 4 | 7 | 6 | |

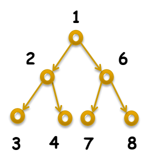
## Insert

What is the worst-case complexity of insert()?

O(log n)

## deleteMin

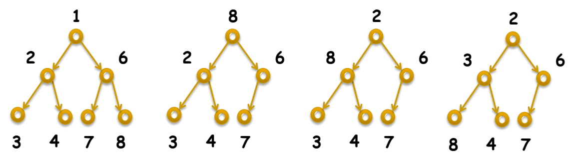
The minimum element can be found at the root of the heap, which is the first element of the array.




Clearly, we cannot delete it.

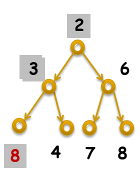
We move the last element of the heap to the root and then restore the heap property by percolating down.

## Delete Min




## Implementation

delMin




| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| | 1 | 2 | 6 | 3 | 4 | 7 | 8 | |
| | 8 | 2 | 6 | 3 | 4 | 7 | | |
| | 2 | 8 | 6 | 3 | 4 | 7 | | |
| | 2 | 3 | 6 | 8 | 4 | 7 | | |

## deleteMin

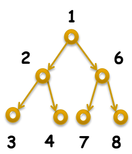
What is the worst-case complexity of deleteMin()?

O(log n)

## HEAPSORT

Run delMin n-times

in-place
nonstable

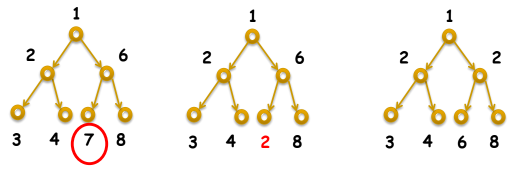



| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| | 1 | 2 | 6 | 3 | 4 | 7 | 8 | |
| | 2 | 3 | 6 | 8 | 4 | 7 | 1 | |
| | 3 | 4 | 6 | 8 | 7 | 2 | 1 | |
| | 4 | 8 | 6 | 7 | 3 | 2 | 1 | |

## decreaseKey

We change the key (value) of one of the heap elements.

To restore a heap property we need to percolate up this item.




## Building a heap

Given an array - turn it into a heap.

There are two algorithms:

1) by insertion O(n log n)

2) heapify O(n)

## Heapify

We insert all the elements into an array in any order.

Next, starting at position *n*/2 and working toward position 1, we push each element down the heap by swapping it with its smallest child.

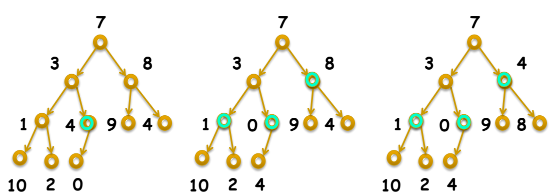
## Heapify

Apply buildHeap operation to the following set of data. Show all intermediate steps.

7, 3, 8, 1, 4, 9, 4, 10, 2, 0

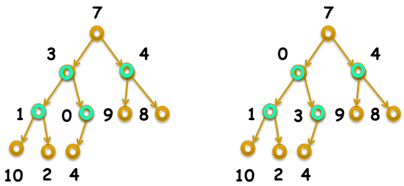
## Heapify

swap 4 and 0 swap 8 and 4



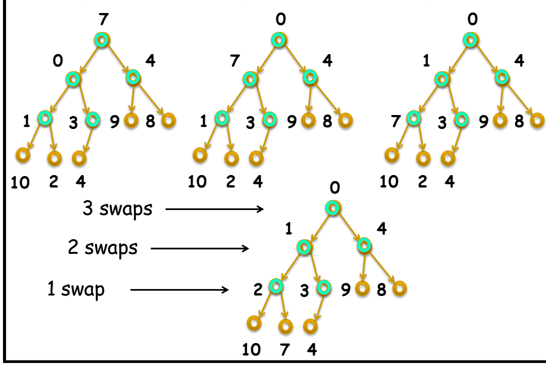
# Heapify

swap 3 and 0




# Heapify

swap 7 and 0 swap 7 and 1 swap 7 and 2




What is the worst-case complexity of heapify?

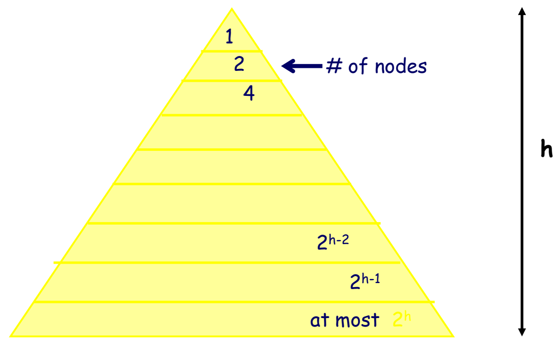
Quick analysis:

during the algorithm execution at most n/2 heap elements percolate down the heap.

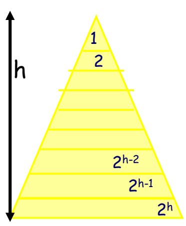
Thus, O(n log n) in the worst-case.

A good upper bound, but it's not asymptotically tight.

# Complexity of heapify




# Complexity of heapify




We will count the number of swaps at each level

| height | # of nodes | # of swaps |
|---|---|---|
| 0 | 1 | h |
| 1 | 2 | h-1 |
| 2 | 4 | h-2 |
| … | … | … |
| h-1 | $2^{h-1}$ | 1 |

What is total work ?

# Complexity of heapify

To compute the total work we multiply the number of swaps by the number of nodes on each level:

$$T(n) = \sum_{k=1}^{h} k\, 2^{h-k}$$

| height | # of nodes | # of swaps |
|---|---|---|
| 0 | 1 | h |
| 1 | 2 | h-1 |
| … | … | … |
| h-2 | $2^{h-2}$ | 2 |
| h-1 | $2^{h-1}$ | 1 |

## Complexity of heapify

$$\sum_{k=1}^{h} k\,2^{h-k} = 2^h \sum_{k=1}^{h}\frac{k}{2^k} \le 2^h \sum_{k=1}^{\infty}\frac{k}{2^k} = 2^{h+1} = O(n)$$

Let

$$x = \sum_{k=1}^{\infty}\frac{k}{2^k} = \frac{1}{2}+\frac{2}{4}+\frac{3}{8}+\dots$$

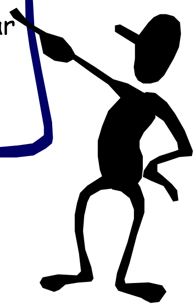
Compute

$$\frac{x}{2} = \frac{1}{4}+\frac{2}{8}+\frac{3}{16}+\dots$$

Subtract

$$x-\frac{x}{2} = \frac{1}{2}+\frac{1}{4}+\frac{1}{8}+\dots \quad = 1$$

Building a heap has a linear runtime complexity.



Devise the algorithm of merging two heaps into one. What is its running time?

O(n)

Merge two arrays in one, and then run heapify.

## Priority Queues

A priority queue is a data structure which supports two basic operations: insert a new item according its priority, and remove the item with the highest priority.

It's implemented as a heap.

## Binary Heaps

.

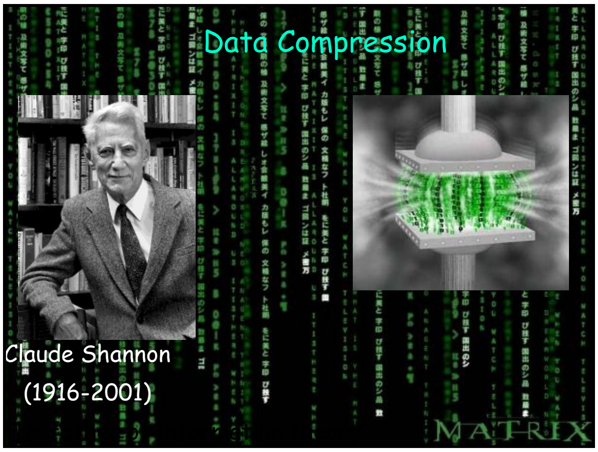
| | Complexity |
|---|---|
| findMin | $\Theta(1)$ |
| deleteMin | $\Theta(\log n)$ |
| insert | $\Theta(\log n)$ |
| decreaseKey | $\Theta(\log n)$ |
| build | $\Theta(n)$ |
| merge (meld) | $\Theta(n)$ |

Efficient searching is not supported

## MORE HEAPS

In the next lecture I will describe a different kind of heap that has a slight improvement over the binary heap.

That data structure was introduced by Vuillemin in 1978, and then further extended by Fredman and Tarjan in 1987.




## Basic Data Compression Concepts

original → compressed → decompressed

X → [Encoder] → y → [Decoder] → X*

Compression - bit reduction

byte (char) -> codeword (bit string)

Lossless compression X = X*

Lossy compression X != X*

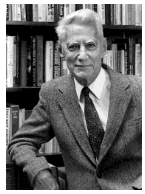
## Why is Data Compression Possible?

Most data has redundancy

There is more data than the actual information contained in the raw data.

The more data random – the less it's compressible.

## Entropy

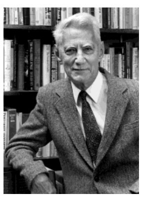


Shannon (1948) established that there is a fundamental limit to lossless data compression.

This limit is called the entropy rate H.

Entropy is a measure of the amount of information contained in the source.

## Entropy



It is possible to compress the source, in a lossless manner, with compression rate close to the entropy H.

It is mathematically impossible to do better than H.

## First-Order Model

In the English language some letters occur more frequently than others.

Let each letter in the alphabet has a certain probability $p_k$. The entropy is given by

$$\boxed{H = \sum_{k=0}^{n} p_k \log \frac{1}{p_k}}$$

## Huffman coding

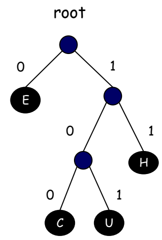
(1952)

David Huffman developed this algorithm as a PhD student in a class of information theory at MIT.

## Huffman Code

Algorithm is used to assign a prefix-free codeword to each char (byte) in the text according to their frequencies.

A prefix-free code is one where NO codeword is a prefix of another codeword.




A codeword is a path from the root to the character.

A codeword for C is 100.

A codeword for H is 11.

## Huffman Code

In general, we want to minimize the overall length of encoding.

$$\text{cost of tree} = \text{MIN} \sum_{k=0}^{n} f(x_k)\, d(x_k)$$

f(x) – frequency of x char/node.

d(x) – depth of x char/node

This suggests *a greedy approach* to constructing a tree.

## Building a Huffman Tree

Given the table of frequencies, let us draw a Huffman tree

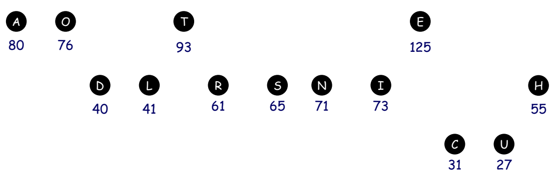
| Char | Freq |
|---|---|
| E | 125 |
| T | 93 |
| A | 80 |
| O | 76 |
| I | 72 |
| N | 71 |
| S | 65 |
| R | 61 |
| H | 55 |
| L | 41 |
| D | 40 |
| C | 31 |
| U | 27 |

We need to get chars with the lowest frequencies at the bottom of the tree. This will guarantee longer codewords assigned to them.

This suggests using a min-heap.

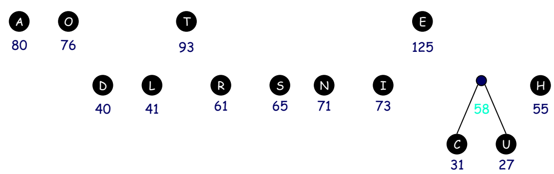
## Building a Huffman Tree

Initially, there are only single-node trees: one for each character.



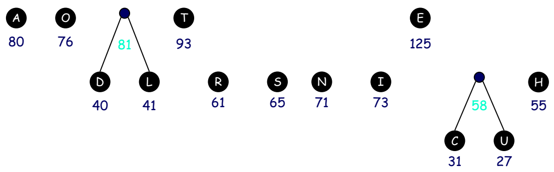

## Building a Huffman Tree

Select two trees of the smallest weights (C and U in this example), breaking ties arbitrarily, and form a new tree with the weight 31+27=58 , and put it back into a heap.

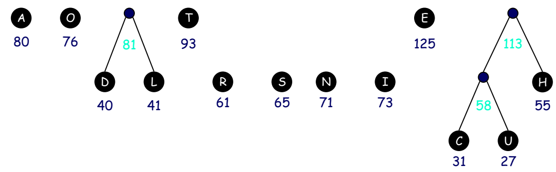


## Building a Huffman Tree

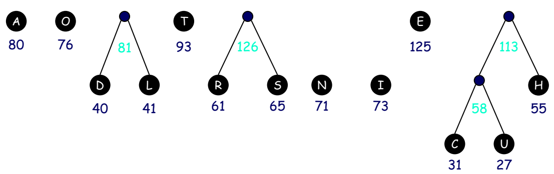
Select two trees of the smallest weights (D and L in this example), and form a new tree with the weight 40+41=81, and put it back into a heap.



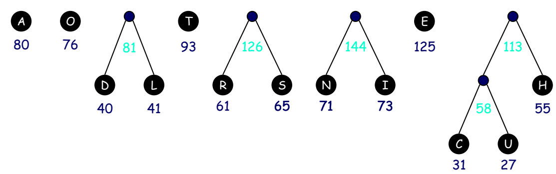

## Building a Huffman Tree




## Building a Huffman Tree

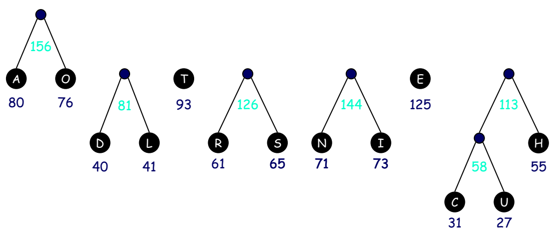



## Building a Huffman Tree

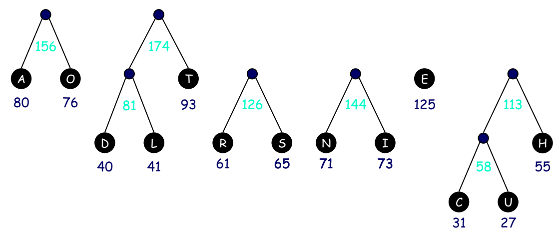



## Building a Huffman Tree

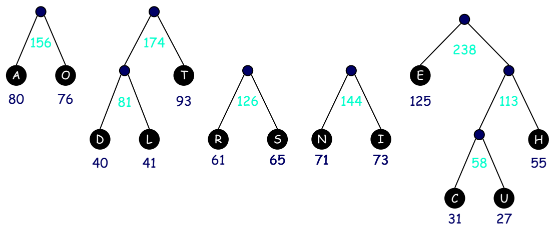
Continue until there is only 1 tree. That tree is the optimal Huffman coding tree.



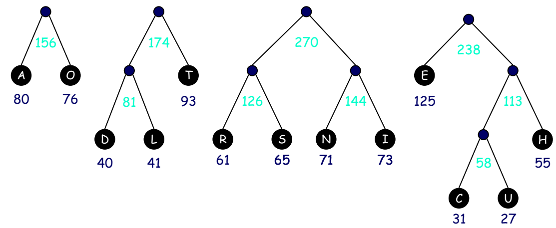

## Building a Huffman Tree



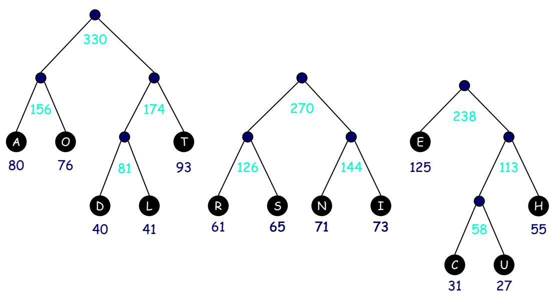
## Building a Huffman Tree



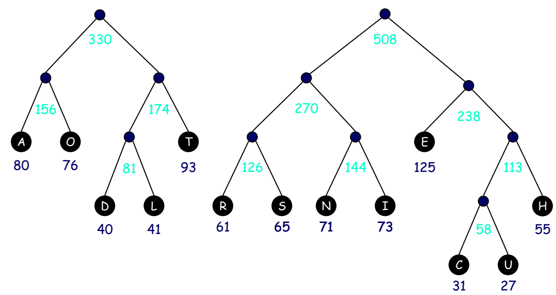

## Building a Huffman Tree



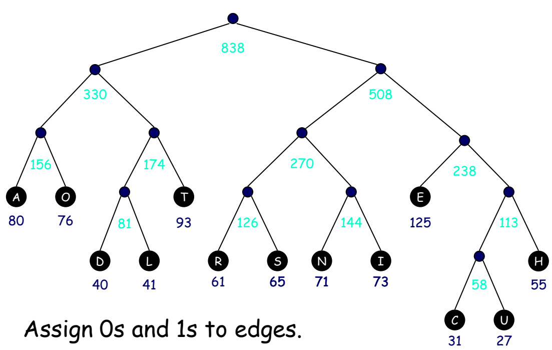

## Building a Huffman Tree




## Building a Huffman Tree

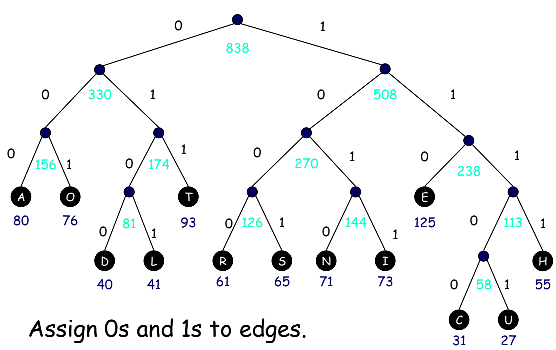



## Building a Huffman Tree



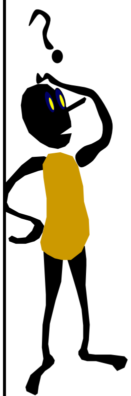

Assign 0s and 1s to edges.

## Building a Huffman Tree




Assign 0s and 1s to edges.

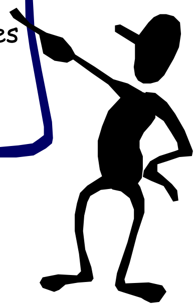
## Huffman Algorithm



What about decompress?

How do we decompress 000011011001011?

To be able to decompress we have to have the Huffman tree.



### Theorem

The Huffman tree provides the optimal prefix-free encoding.

## Proof

A - alphabet
f(x) – frequency of x char/node.
d(x) – depth of x char/node

$$\text{cost of tree} = \sum_{k=0}^{n} f(x_k)\, d(x_k)$$

## Proof (by induction)

For $|A|=2$, the tree (2 leaves) is optimal.
Suppose, it holds for $|A|-1$.
Choose two least frequent chars, $c_1$ and $c_2$.
Consider a new alphabet

$$A^* = A - \{c_1, c_2\} \cup \{c^*\}$$

Observe, that $|A^*|=|A|-1$ and

$$f(c_1) + f(c_2) = f(c^*)$$

## $A^* = A - \{c_1, c_2\} \cup \{c^*\}$

We have two trees now

T – over the alphabet $A$

T* - over the alphabet $A^*$

Note, T* is optimal by ind. hypothesis
We want to prove that T is also optimal.

## $A^* = A - \{c_1, c_2\} \cup \{c^*\}$

By contradiction.

Let us assume that there is $T_1$ such that

$$\text{cost}(T_1) < \text{cost}(T)$$

Compute cost(T)

## $A^* = A - \{c_1, c_2\} \cup \{c^*\}$

$cost(T) = cost(T^*) + f(c_1)d(c_1) + f(c_2)d(c_2) - f(c^*)d(c^*)$

But

$$f(c^*) = f(c_1) + f(c_2)$$

$$d(c^*) = d(c_1) - 1 = d(c_2) - 1$$

Then

$$cost(T) = cost(T^*) + f(c_1) + f(c_2)$$

## Need to prove $cost(T_1) < cost(T)$

We showed that

$$cost(T) = cost(T^*) + f(c_1) + f(c_2)$$

Similarly, we can remove $c_1$ and $c_2$ (if they are siblings) from $T_1$.

Thus

$$cost(T_1) = cost(T_1^*) + f(c_1) + f(c_2)$$

It follows,

$$cost(T_1) < cost(T) \Rightarrow cost(T_1^*) < cost(T^*)$$

But $T^*$ is optimal. Contradiction.

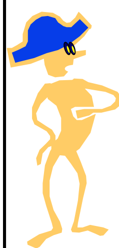
## What if $c_1$ and $c_2$ are not siblings?

Lemma.

Pick x and y such that f(x) and f(y) are minimal. Then there is an optimal prefix code such that x and y are siblings.

Without proof.

## The Minimum Spanning Tree

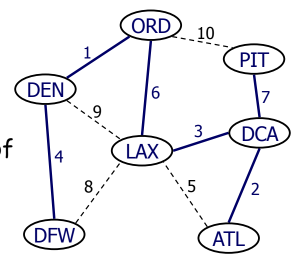


Find a spanning tree of the minimum total weight.

MST is fundamental problem with diverse applications.

Spanning tree is a subgraph (connected and acyclic) of a graph containing all the vertices

Minimum spanning tree (MST) is a spanning tree of a weighted undirected graph with the minimum total edge weight




The weight of a spanning tree is the sum of the weights on all the edges which comprise the spanning tree.

## The Minimum Spanning Tree

Joseph Kruskal (1929-2010)

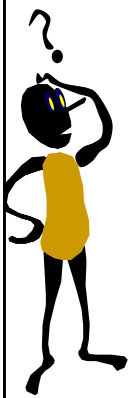
Prim's Algorithm (1957)

Kruskal's Algorithm (1956)

Boruvka's Algorithm (1926)

Robert Prim (1921-)

## The MST



Brute Force Algorithm:

Using BFS, find ALL spanning trees and then pick one with the minimum cost.

What's wrong with this idea?

## Cayley's Formula

The number of spanning trees in $K_n$ is $n^{n-2}$

Arthur Cayley
(1821-1895)

$K_n$ is a *complete* simple graph in which every pair of distinct vertices is connected by a unique edge.

## Prim's Algorithm

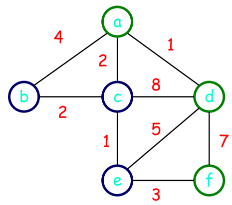
algorithm builds a tree one VERTEX at a time.

- Start with an arbitrary vertex as component *C*
- Expand *C* by adding a vertex having the minimum weight edge of the graph having exactly one end point in *C*.
- Continue to grow the tree until *C* gets all vertices.

## Cut Property

A cut of a graph is a partition of its vertices into two disjoint sets. Yellow and green below.

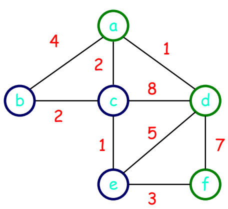
A crossing edge is an edge that connects a vertex in one set with a vertex in the other. For example, (d, e) in the picture




## Proof of the correctness.

Lemma: Given any cut in a weighted graph, the crossing edge of minimum weight is in the MST of the graph.

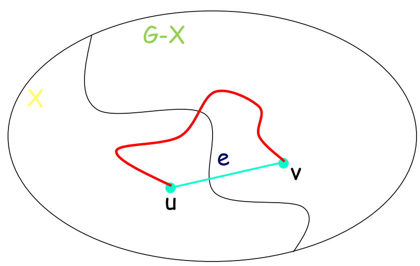
Among five crossing edges, (a, c) is the smallest, so it must be in the MST.




## Proof of the Lemma

Let T be the MST but e (crossing edge) is not in T

Adding e to T creates a cycle.



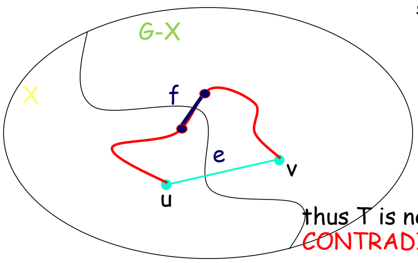

e is the smallest edge

e is not in T

## Proof of the Lemma

There is some other crossing edge f > e in T.

Create another T1 = T - f + e < T




e is the smallest edge

e is not in T

f is in T

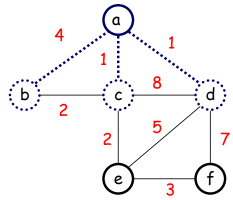
f > e

thus T is not the MST

CONTRADICTION

## Prim's Algorithm

C={a}

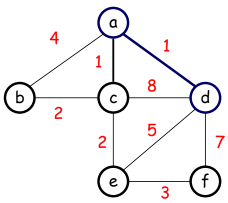



heap

d-1 c-1 b-4 e-oo f-oo

## Prim's Algorithm

C={a,d}

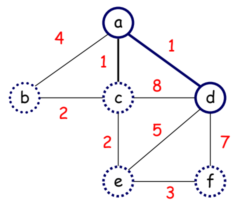



heap

c-1 b-4 e-oo f-oo

## Prim's Algorithm

C={a,d}



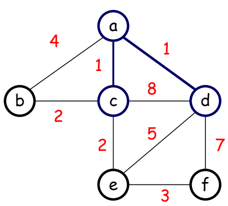

heap

c-1 b-4 e-5 f-7

decreaseKey

## Prim's Algorithm

C={a,d,c}

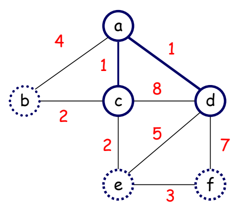



heap

b-4 e-5 f-7

## Prim's Algorithm

C={a,d,c}

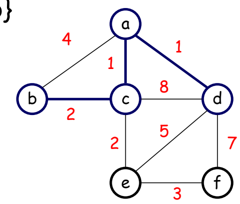



heap

b-2 e-2 f-7

## Prim's Algorithm

C={a,d,c,b}

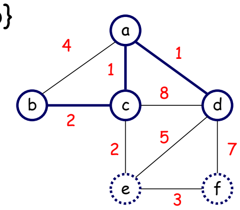



heap

e-2 f-7

## Prim's Algorithm

C={a,d,c,b}

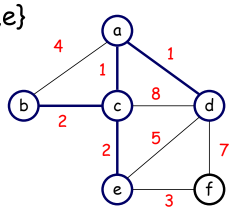



heap

e-2 f-7

## Prim's Algorithm

C={a,d,c,b,e}

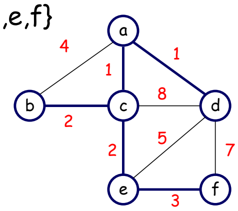



heap

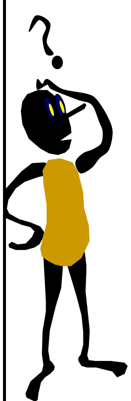
f-3

## Prim's Algorithm

C={a,d,c,b,e,f}




Weight = 1+1+2+2+3 = 9



What is the worst-case runtime complexity of Prim's Algorithm?

## Complexity of Prim's Algorithm

To find a shortest distance to C, we maintain a priority queue of vertices.

deleteMin – O(log V)
decreaseKey – O(log V)

We run deleteMin V times.

We update the queue E times.

The total cost:

O(V*log V + E*log V)

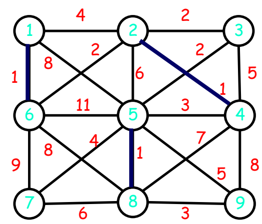
## Kruskal's Algorithm

algorithm builds a tree one EDGE at a time.

- Start with all vertices as a forest
- Choose the cheapest edge and joint correspondent vertices (subject to cycles)
- Continue to grow the forest

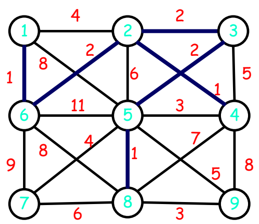
## Kruskal's Algorithm

Start with minimal weight edges.
There are three edges of weight 1



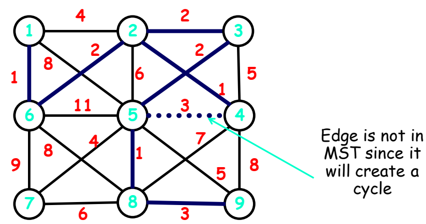
## Kruskal's Algorithm

There are three edges of weight 2



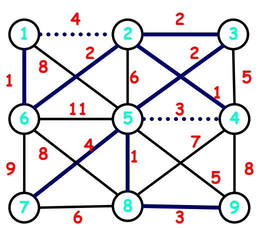
## Kruskal's Algorithm

There are two edges of weight 3



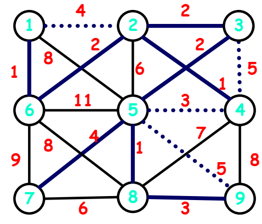

## Kruskal's Algorithm

There are two edges of weight 4



## Kruskal's Algorithm

There are two edges of weight 5



We don't have to consider all edges, we stop as soon as we get a spanning tree.

## Complexity of Kruskal's Algorithm

Sorting edges – O(E log E)

Cycle detection – O(V)

Total: O(V*E + E*log E)

Compare it with Prim's: O(V*log V + E*log V)

## Implementation of Kruskal's Algorithm

We need a new data structure:

a disjoint set

When examining an edge, we need to check if both vertices are in the same disjoint set:

if no, accept the edge and take the union of the two sets, otherwise

if yes, then this would cause a cycle

## Disjoint Set

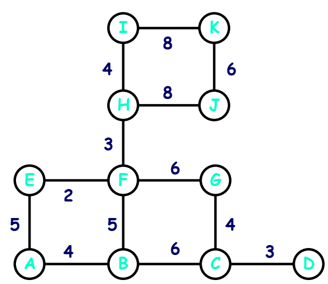
This data structure maintains a collection of disjoint sets, with the operations:

-find(u): return the set storing u

-union(u, v): joins two sets containing u and v together

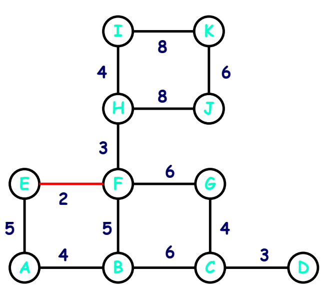
## Kruskal's Algorithm: Disjoint Set

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| A | B | C | D | E | F | G | H | J | I | K |



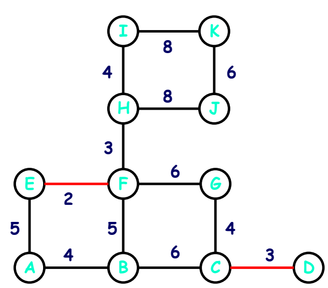

## Disjoint Set

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| A | B | C | D | E F | | G | H | J | I | K |



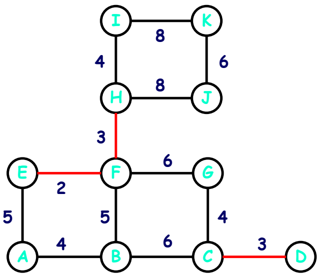

## Disjoint Set

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| A | B | C D | | E | F | G | H | J | I | K |



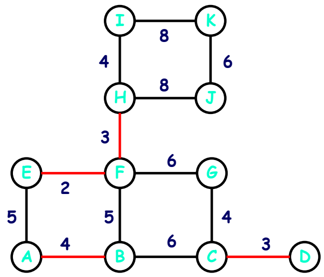
## Disjoint Set

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| A | B | C D | E F H | G | J | I | K |



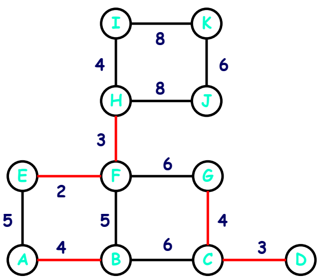

## Disjoint Set

| | | | | | | |
|---|---|---|---|---|---|---|
| A B | C D | E F H | G | J | I | K |



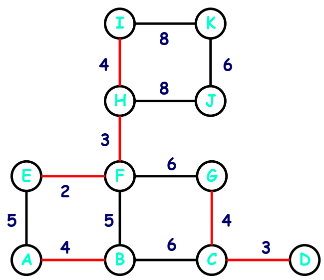

## Disjoint Set

| | | | | | |
|---|---|---|---|---|---|
| A B | C D G | E F H | J | I | K |



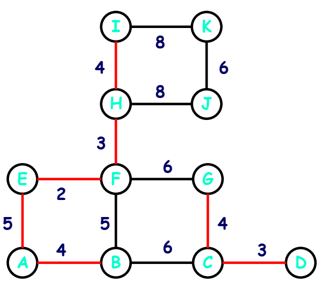

## Disjoint Set

| | | | | |
|---|---|---|---|---|
| A B | C D G | E F H I | J | K |



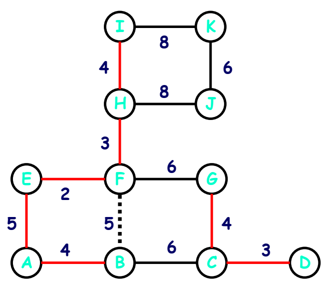

## Disjoint Set

| | | | |
|---|---|---|---|
| A B E F H I | C D G | J | K |



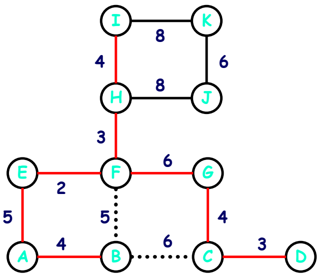

## Disjoint Set

| | | | |
|---|---|---|---|
| A B E F H I | C D G | J | K |



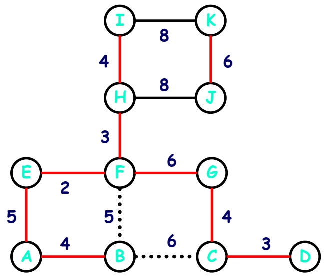
## Disjoint Set

| | | |
|---|---|---|
| A B E F H I C D G | J | K |



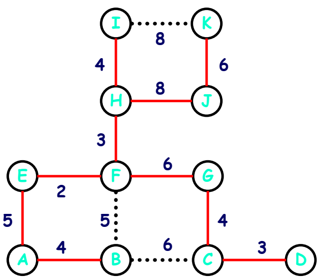

## Disjoint Set

| | |
|---|---|
| A B E F H I C D G | J K |




## Disjoint Set

| |
|---|
| A B E F H I C D G J K |




## Union – Find: implementation

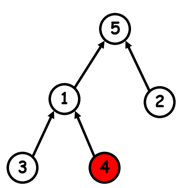
We implement each set as a tree, using parent pointers.

The root is a representative of all nodes in this tree.

Find() means a tree traversal to the root.
Union() means joining two trees.

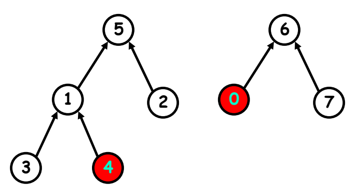
## FIND

FIND(4) returns the root, which is 5



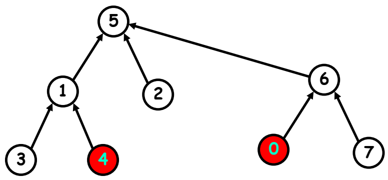

## UNION

UNION(4, 0) calls FIND(4) and FIND(0)



## UNION

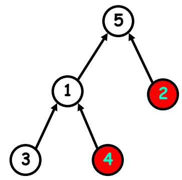
UNION(4, 0) calls FIND(4) and FIND(0)




In which order do we join two trees?

## UNION

UNION(4, 2) calls FIND(4) and FIND(2)



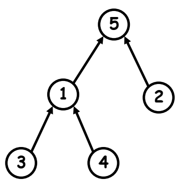

2 and 4 belong to the same set.

## FIND: implementation

Actually, a tree is implemented as an array.

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| -1 | 5 | 5 | 1 | 1 | 5 | -1 | -1 |

Vertex k has a parent that is stored at parent[k]




## FIND: implementation

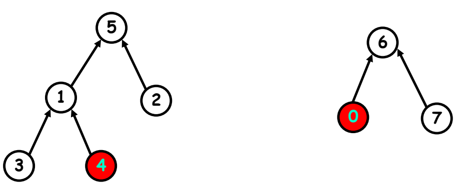
Vertex k has a parent that is stored at parent[k]

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| -1 | 5 | 5 | 1 | 1 | 5 | -1 | -1 |

```
find(i):
while( i != parent[i] && parent[i] >= 0 )
  i = parent[i];
return i;
```

## UNION: implementation

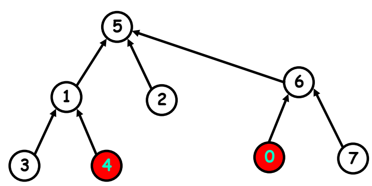
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| 6 | 5 | 5 | 1 | 1 | 5 | 6 | 6 |




UNION(4, 2) calls FIND(4) and FIND(2)

## UNION: implementation

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| 6 | 5 | 5 | 1 | 1 | 5 | 5 | 6 |



## Union by Rank

Maintain heights (called rank) of all trees.

During UNION, make a shorter tree a subset of a taller tree.

## UNION: implementation

```
union(i,j):
root1 = find(i);   root2 = find(j);
if(root1 != root2)
    if(height(root1) > height(root2))
        parent[root2] = root1;
    else
        parent[root1] = root2;
```

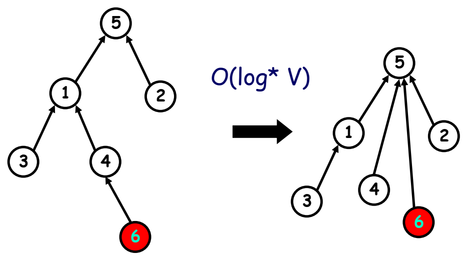
## Worst-case Complexity

FIND has cost $O(V)$

UNION has cost $O(V) + O(1)$

## Path Compression

The idea is to make the tree height smaller. During a FIND operation, we redirect all nodes on the path to the root.




## log* n – iterated log

log*n is the number of times we need to apply log to get 1.

$\log^*16 = 3$ since $\log \log \log 2^4 = 1$

$\log^* 2^{16} = 4$

$\log^* 2^{65536} = 5$

### Amortized Cost

n Union/Find operations take $O(n \log^* n)$ which is almost linear.

I will formally define an amortized cost in the next lecture.