Analysis of Algorithms

V. Adamchik CSCI 570 Fall 2016

Lecture 4 University of Southern California

Amortized Analysis
Binomial Heaps
Shortest Path

Amortized Analysis

In a sequence of operations the worst case does not occur often in each operation - some operations may be cheap, some may be expensive.

Therefore, a traditional worst-case per operation analysis can give overly pessimistic bound.

For example, in a dynamic (unbounded) array only some inserts take a linear time, though others - a constant time.

When different inserts take different times, how can we accurately calculate the total time?

Amortized Analysis

Amortized analysis gives the average performance (over time) of each operation in the worst case.

The amortized cost per operation for a sequence of n operations is the total cost of the operations divided by n.

It requires that the total real cost of the sequence should be bounded by the total of the amortized costs of all the operations.

Amortized Analysis

Three methods are used in amortized analysis

1. Aggregate Method (or brute force)
2. Accounting Method (or the banker's method)
3. Potential Method (or the physicist's method)

We won't use a potential method in this class.

Unbounded Array

Thinking about it abstractly, an unbounded array should be like an array in the sense that we can get and set the value of an arbitrary element via its index.

We should also be able to add a new element to the end of the array, and delete an element from the end of the array.

Unbounded Array

The general implementation strategy:

we maintain an array of a fixed length limit and an internal index size which tracks how many elements are actually used in the array. When we add a new element we increment size, when we remove an element we decrement size.

The tricky issue is how to proceed when we are already at the limit and want to add another element.

Unbounded Array

At that point, we allocate a new array twice as large and copy the elements we already have to the new array.

So, if the current array is full, the cost of insertion is linear; if it is not full, insertion takes a constant time.

In order to make the analysis as concrete as possible, we will count the number of inserts and the number of copyings. We won't analyze deletions.

insert	old size	new size	copy
1	1	-	-
2	1	2	1
3	2	4	2
4	4	-	-
5	4	8	4
6	8	-	-
7	8	-	-
8	8	-	-
9	8	16	8

Aggregate Method

The table shows that 9 inserts require

$1 + 2 + 4 + 8$ copy operations.

Let us generalize the pattern.

Assume we start with the array of size 1 and make $2^n + 1$ inserts.

These inserts will require

$1 + 2 + 4 + \ldots + 2^n = 2^{n+1} - 1$

copy operations.

Thus, the total work is $(2^n + 1) + (2^{n+1} - 1)$.

Aggregate Method

The total work is $(2^n + 1) + (2^{n+1} - 1) = 3 \cdot 2^n$.

Next we computer the average cost per insert.

$$\lim_{n\to\infty} \frac{3\times 2^n}{2^n+1} = 3$$

We say that the amortized cost of insertion is constant.

Such method of analysis is called an aggregate method.

The Accounting Method

The aggregate method seeks an upper bound on the overall running time of a sequence of operations.

The accounting method seeks a payment for each individual operation.

Intuitively, we maintain a bank account and each operation is charged to it.

Some operations are charged very little but also generate a surplus. Others, drain the savings.

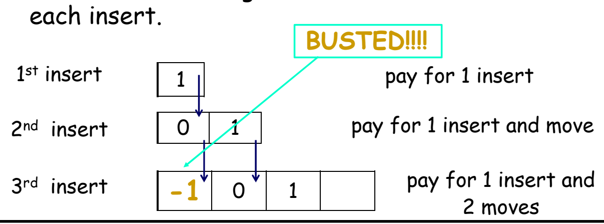
The balance in the bank account must always remain positive.

Unbounded Array

We will assign a dollar token to each operation.

Say it costs 1 token to insert an element and 1 token to move it when the table is doubled.

So, we have to assigned at least 2 tokens to each insert.



Unbounded Array

Let us assign 3 tokens to each insert.

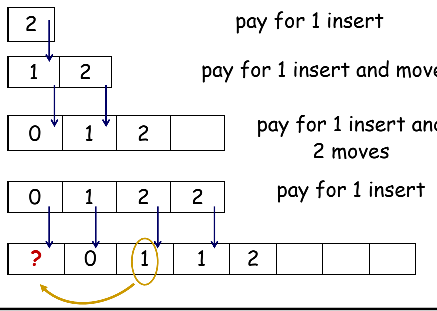
1st insert — pay for 1 insert

2nd insert — pay for 1 insert and move

3rd insert — pay for 1 insert and 2 moves

4th insert — pay for 1 insert

5th insert



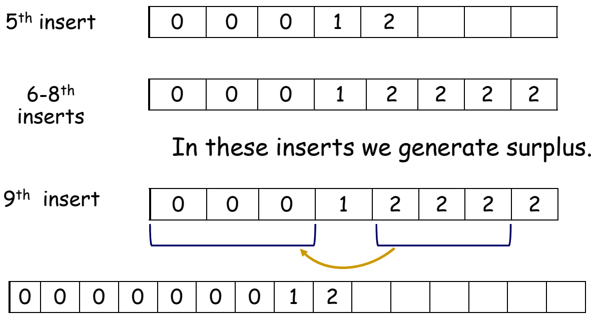
There are enough money in the bank account to bail out the busted element.

5th insert

6-8th inserts

In these inserts we generate surplus.

9th insert



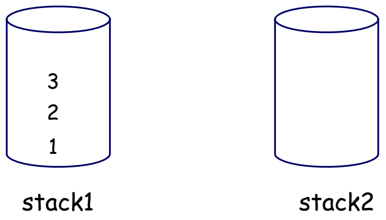
In short, we run a 'ponzi' scheme…

FIFO Queue with two stacks

We can implement a FIFO queue using two stacks.

How de we implement enqueue?

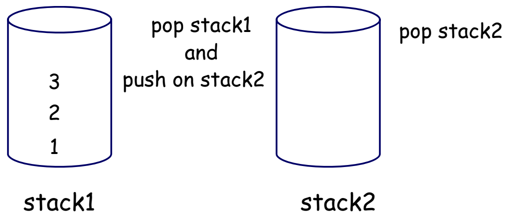
Example: enqueue(1), enqueue(2), enqueue(3).



FIFO Queue with two stacks

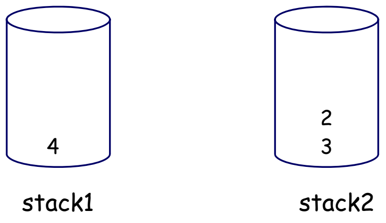
How de we implement dequeue?

Continue with the example: dequeue()



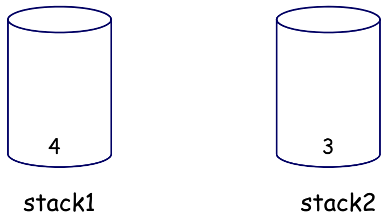
FIFO Queue with two stacks

Continue with example: enqueue(4)



FIFO Queue with two stacks

Continue with example: dequeue()



FIFO Queue with two stacks

enqueue(x): push x onto stack1.

dequeue(): if stack2 is empty, pop the entire stack1 and push it into stack2.
pop stack2.

What is the worst-case complexity of enqueue?
O(1)

What is the worst-case complexity of dequeue?
O(n)

FIFO Queue with two stacks: amortized cost

The accounting method.

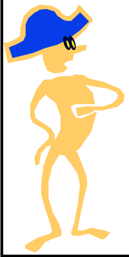
How many tokens we need to assign to enqueue?

Think about a case when some items enqueued but may never be dequeued.

…but they might still be popped and pushed to the other stack.

We assign three tokens to enqueue: one to push the item onto stack1, the other to pop it from stack1 and one more to push it onto stack2.

BINOMIAL HEAPS

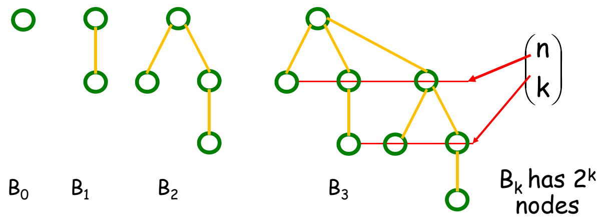


We describe a different kind of heap that has a slight improvement over the binary heap. This data structure was introduced by Vuillemin in 1978.

Binomial Trees

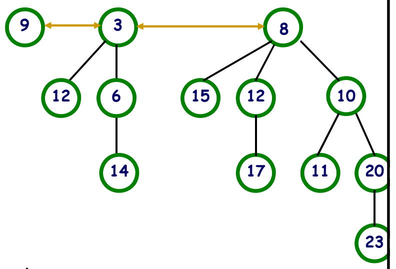
The binomial tree of rank k, B_k, is defined recursively as follows

1. B_0 is a single node
2. B_k is formed by joining two B_{k-1} trees



Binomial Heaps

A binomial heap is a collection of at most Celling(log n) binomial trees in increasing order of size where each tree has a heap ordering property.



In a binomial heap there is <u>at most</u> one binomial tree of any given rank.

Binomial Heaps and Binary Expansion

To store 11 items we need B_3, B_1 and B_0 binomial trees. It follows from the binary expansion:

$$11_{10} = 1011_2$$

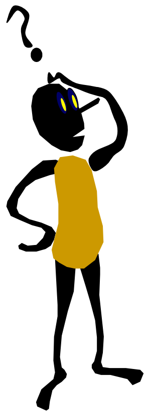
To store 25 items

$$25_{10} = 11001_2$$

we need B_4, B_3 and B_0.

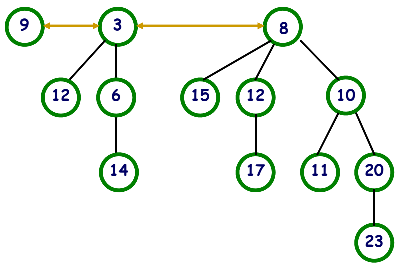
DEMO

https://www.cs.usfca.edu/~galles/visualization/BinomialQueue.html



How can we find the minimum element in a binomial heap?

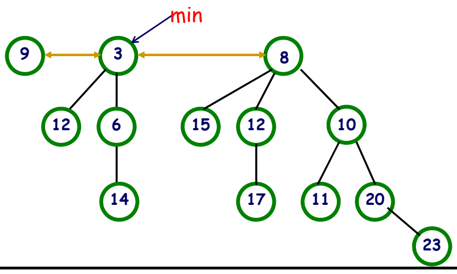
How long does it take?



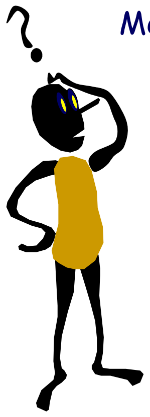
FindMin

Look at the root of each binomial tree for the minimum – O(log n)

Store the pointer to the minimum root – O(1)



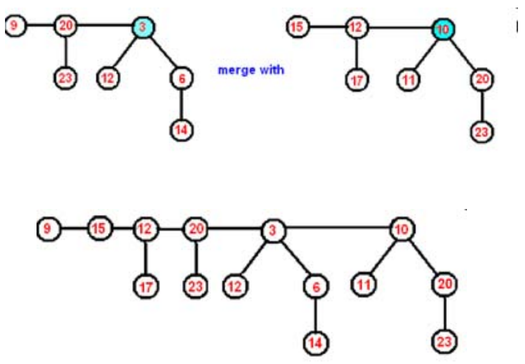
Merging Binomial Heaps



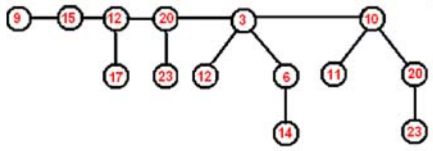
Devise an algorithm for merging two binomial heaps and discuss its complexity.

Note, binary heaps are complete binary trees, and two complete binary trees cannot easily be linked to one another.

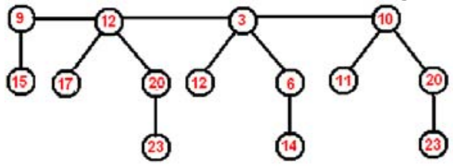
Merging Binomial Heaps



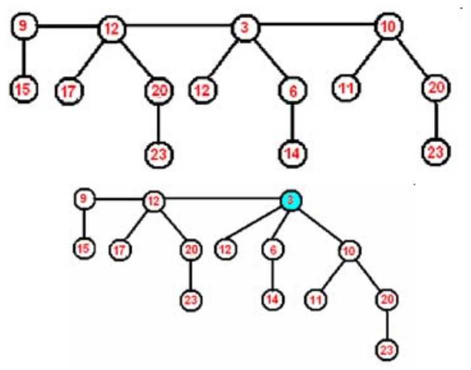
Merging Binomial Trees



Merging trees of the same size requires O(1) time.
Make the smaller root the child of the larger root.



Merging Binomial Trees



Merging Binomial Heaps: binary addition

Merge $B_0B_1B_2B_4$ with B_1B_4 to get $B_0B_3B_5$

We do a binary addition:

```
  10111
  10010
 101001
```

We need to merge at most log n binomial trees, and each merge takes O(1) time, so the whole operation takes O(log n) time

Worst-case complexity of merge.

Think of a binomial heap $B_0B_0B_1B_2B_3 \ldots B_{\log n}$

First merge - $B_1B_1B_2B_3 \ldots B_{\log n}$

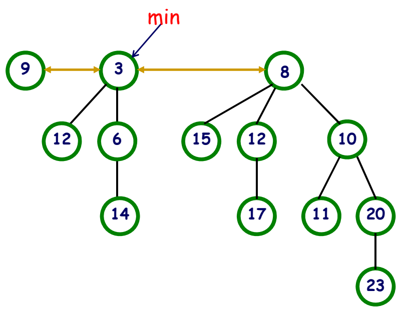
Second merge - $B_2B_2B_3 \ldots B_{\log n}$

Third merge – $B_3B_3B_4 \ldots B_{\log n}$ and so on…

each merge takes O(1) time

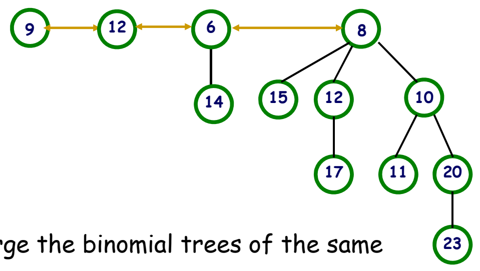
so the whole operation takes O(log n) time

Devise an algorithm for deleteMin() and discuss its complexity.



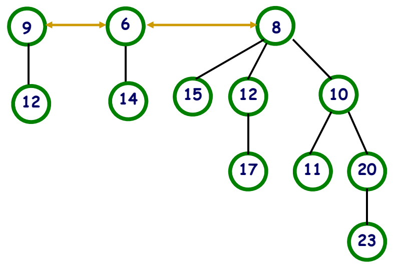
1. Find the binomial tree that contains the min.
2. Delete the root and move subtrees to top list.

Deleting the root of B_k yields $B_0, B_1, \ldots, B_{k-1}$.

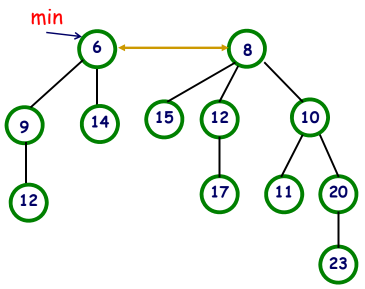


3. Merge the binomial trees of the same rank.

3. Merge the binomial trees of the same rank.



3. Merge the binomial trees of the same rank.
4. Set a pointer to the new min.



Worst-case complexity of delMin?

Same as merging two heaps!

It takes O(log n) time

Insertion

How do we insert into a binomial heap?

Essentially this is a union of two binomial heaps: one of size 1 and other of the size n.

Therefore it takes O(log n) in the worst case.

…but not all inserts are so slow…

Insertion

What is the amortized cost of insertion?

Use the accounting method.

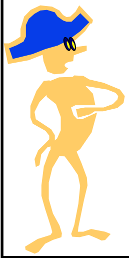
Each insertion requires only two tokens. One to create a single binomial tree. The other to pay for the future merge.

Insertion: Binomial vs Binary Heaps

The cost of inserting n elements into a binary heap, one after the other, is Θ(n log n) in the worst-case.

If n is known in advance, we run heapify, so a binary heap can be constructed in time Θ(n).

The cost of inserting n elements into a binomial heap, one after the other, is Θ(n) (amortized cost), even if n is not known in advance.



FIBONACCI HEAPS

Fredman andTarjan, 1987

Idea: relaxed binomial heaps
Goal: decreaseKey in O(1)

FIBONACCI HEAPS

We want to incorporate the decreaseKey operation into the binomial heap. How can we do this?

One idea to try is simply this. To decrease the key of a node, we simply change its key value. If its new key is less than that of its parent, we swap the node with its parent. We repeat this till we get to the top of the binomial tree, and stop.

Its running time is O(log n).
But we're striving for O(1) running time.

FIBONACCI HEAPS

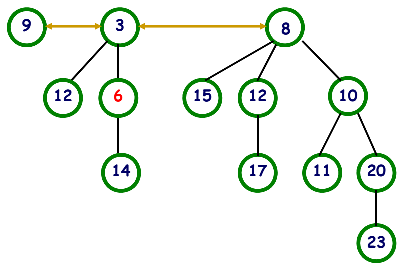
Another idea (similar to deleteMin):

take the node you want to decrease, and change its key, and disconnect it and its entire subtree from where it is, and attach it to the tree root list.

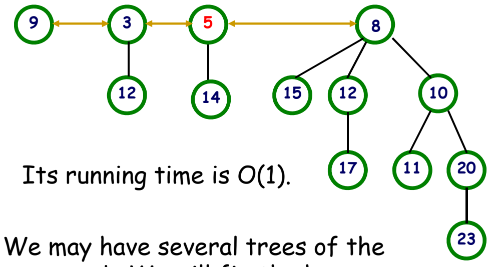
This is clearly O(1) time.

decreaseKey: example

Suppose we want to change 6 to 5.



decreaseKey: example



Its running time is O(1).

We may have several trees of the same rank. We will fix the heap when deleteMin is called.

decreaseKey

The problem is that we can no longer prove the bound on the time of deleteMin. That heap may contain more than log(n) binomial trees.

There's a clever way to fix this devised by Fredman and Tarjan, the complexity of decreaseKey will be O(1) amortized. This is the Fibonacci heap algorithm.

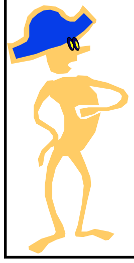
The algorithm is outside of the scope of this course.

HEAPS

	Binary	Binomial	Fibonacci
findMin	Θ(1)	Θ(1)	Θ(1)
deleteMin	Θ(log *n*)	Θ(log *n*)	O(log *n*) (ac)
insert	Θ(log *n*)	Θ(1) (ac)	Θ(1) (ac)
decreaseKey	Θ(log *n*)	Θ(log *n*)	Θ(1) (ac)
merge(meld)	Θ(*n*)	Θ(log n)	Θ(1) (ac)

ac – amortized cost.

The Shortest Path Problem



Edsger Dijkstra
(1930-2002)

Dijkstra's quotes

A programming language is a tool that has a profound influence on our thinking habits.

The question of whether computers can think is like the question of whether submarines can swim.

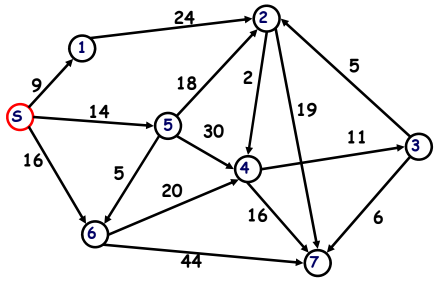
Computer science is no more about computers than astronomy is about telescopes.

Dijkstra's quotes

Right from the beginning, . . . we stress that the programmer's task is not just to write down a program, but that his main task is to give a formal proof that the program he proposes meets the equally formal functional specification.

The Shortest Path Problem

Given a positively weighted graph G with a source vertex s, find the shortest path from s to all other vertices in the graph.



Greedy approach

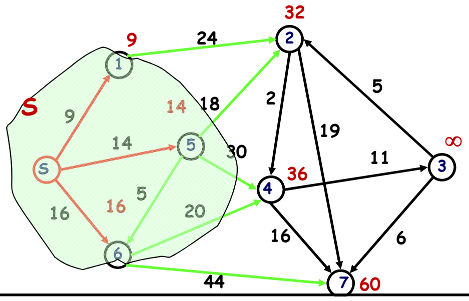
When algorithm proceeds all vertices are divided into two groups

- vertices whose shortest path from the source is known
- vertices whose shortest path from the source is NOT known

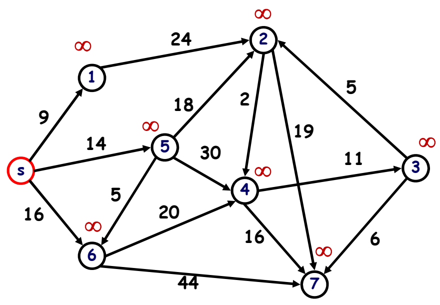
Move vertices one at a time from the unknown set to the known set, based on the shortest distance to the source.

The Shortest Path Problem

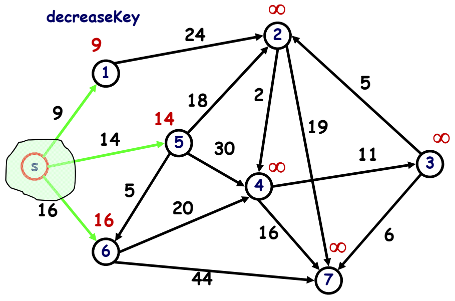
The algorithm is almost identical to Prim's algorithm. Prim's algorithm picks a minimum cost edge, but Dijkstra's a shortest path.



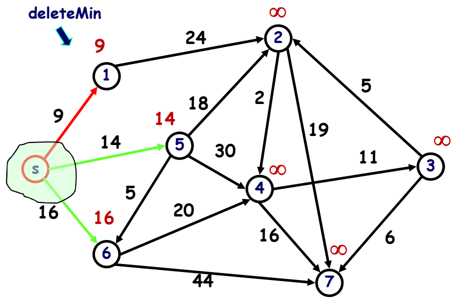
S = { }
PQ = { s, 1, 2, 3, 4, 5, 6, 7}



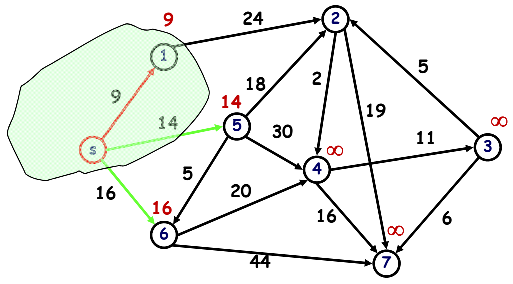
S = { s }
PQ = {1, 2, 3, 4, 5, 6, 7}



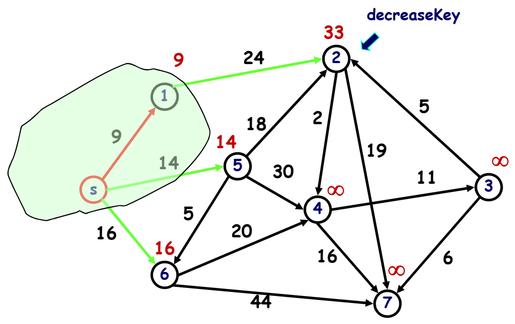
Solution = { s }
PQ = {1, 2, 3, 4, 5, 6, 7}



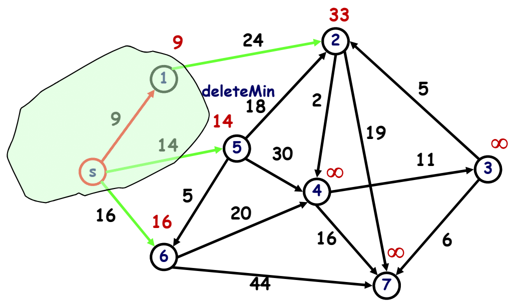
Solution = { s, 1 }
PQ = {2, 3, 4, 5, 6, 7}



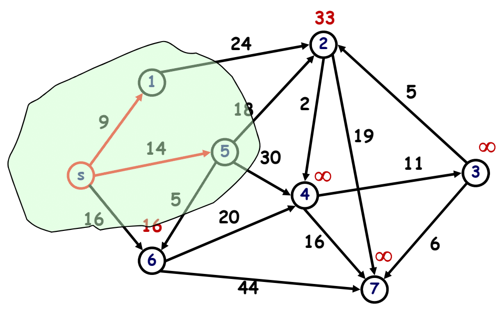
Solution = { s, 1 }
PQ = {2, 3, 4, 5, 6, 7}



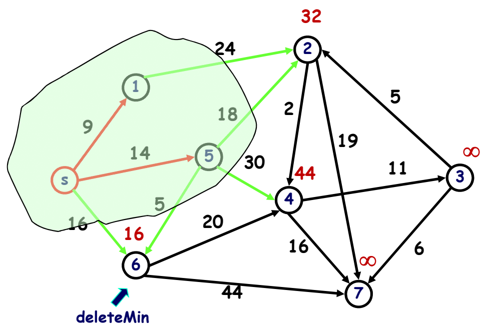
Solution = { s, 1 }
PQ = {2, 3, 4, 5, 6, 7}



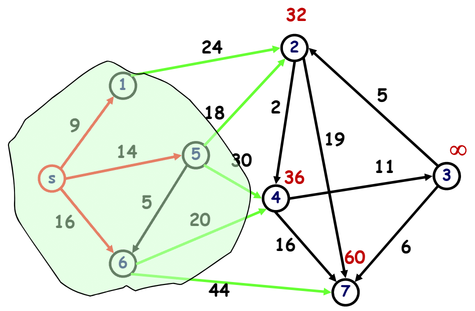
Solution = { s, 1, 5 }
PQ = {2, 3, 4, 6, 7}



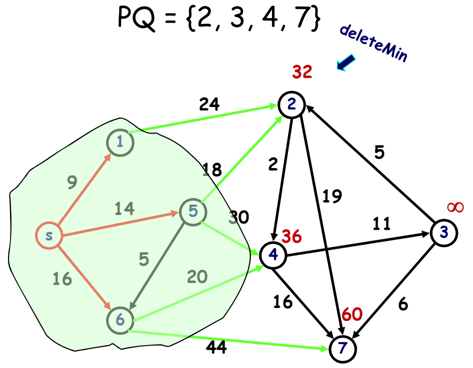
Solution = { s, 1, 5 }
PQ = {2, 3, 4, 6, 7}



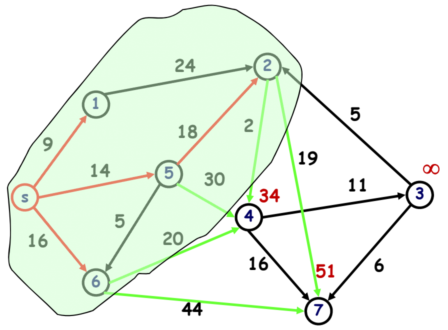
Solution = { s, 1, 5, 6 }
PQ = {2, 3, 4, 7}



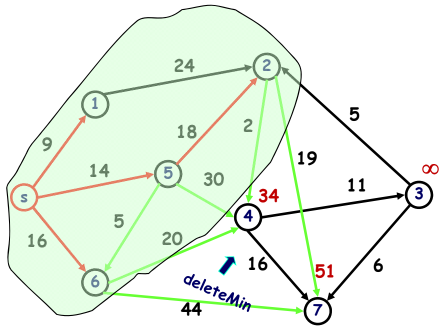
Solution = { s, 1, 5, 6 }
PQ = {2, 3, 4, 7}



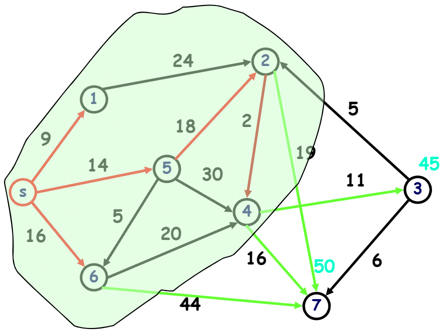
Solution = { s, 1, 5, 6, 2 }
PQ = {3, 4, 7}



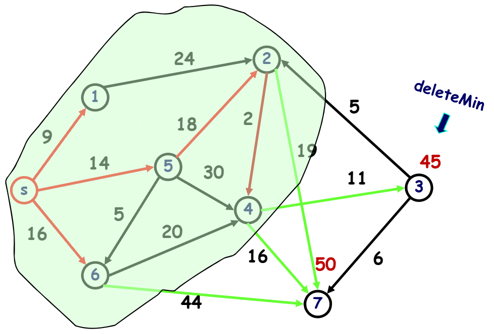
Solution = { s, 1, 5, 6, 2 }
PQ = {3, 4, 7}



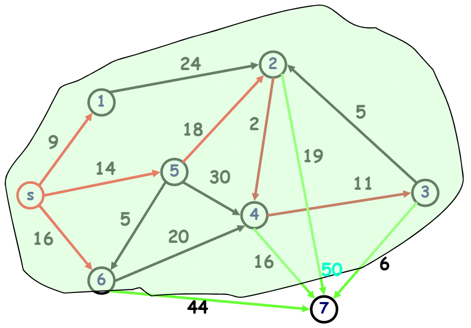
Solution = { s, 1, 5, 6, 2 }
PQ = {3, 4, 7}



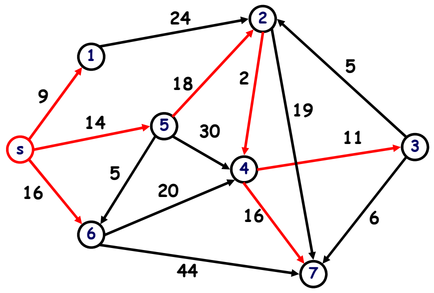
Solution = { s, 1, 5, 6, 2 }
PQ = {3, 4, 7}



Solution = { s, 1, 5, 6, 2 }
PQ = {3, 4, 7}



Solution = { s, 1, 5, 6, 2, 4, 3, 7 }
PQ = {}



Complexity

Let D(v) denote a length from the source s to vertex v. We store distances D(v) in a PQ.

INIT: D(s) = 0; D(v)= ∞
LOOP:
Delete a node v from PQ using deleteMin()
Update D(w) for all w in adj(v) using decreaseKey()

D(w) = min[D(w), D(v) + c(v, w)]

Complexity

Let D(v) denote a length from the source s to vertex v. We store distances D(v) in a PQ.
O(V)
INIT: D(s) = 0; D(v)= ∞
LOOP: O(log V)
Delete a node v from PQ using deleteMin()
Update D(w) for all w in adj(v) using decreaseKey()
O(log V)
D(w) = min[D(w), D(v) + c(v, w)]

Complexity

Let D(v) denote a length from the source s to vertex v. We store distances D(v) in a PQ.

O(V)

INIT: D(s) = 0; D(v)= ∞ PQ has V vertices

LOOP: O(log V)

Delete a node v from PQ using deleteMin()

Update D(w) for all w in adj(v) using decreaseKey()

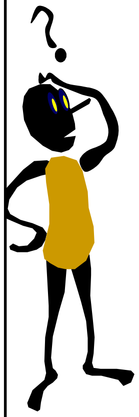
O(log V)

We do O(E) updates

D(w) = min[D(w), D(v) + c(v, w)]

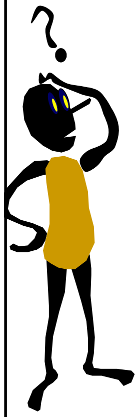
Complexity

O (V Log V + E log V)



What would be the runtime complexity of Dijkstra's algorithm if we use the Fibonacci heap?

O (V Log V + E)



Assume that an unsorted array is used instead of a priority queue.

What would the algorithm's running time in this case?

PQ is a linear array

findMin takes O(V) - for one vertex
findMin takes O(V^2) - for all vertices

decreaseKey takes O(1) - for one edge
decreaseKey takes O(E) – for all edges

the total runtime O(V^2 + E)

Compare it with O (V Log V + E)

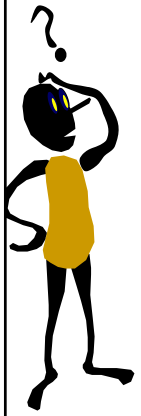
Complexity

If a graph is dense (E=V^2) we use an array

O (V^2)

If a graph is sparse we use a PQ

O (V Log V)



Design an algorithm to find shortest distances in a DAG.

Process the vertices in a topological order, and calculate the path length for each vertex to be the minimum length obtained via any of its incoming edges.

This is known as the Viterbi algorithm for shortest distances.

Proof of Correctness

Lemma. For each node u ∈ S, d(u) is the shortest s-u path.

Proof. (by induction on |S|

Base case: |S| = 1 is trivial.

IH: Assume true for |S| = k.

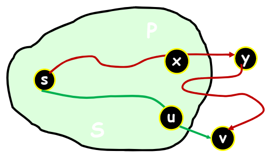
Notations:

d(x) is the shortest s-x path,

π(x) is a path defined by

$\pi(x) = \min (d(v) + c(v, x))$

Proof of Correctness



Let v be next node added to S, and let u-v be the chosen edge.

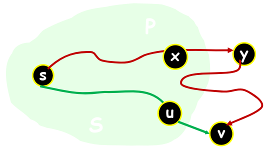
The shortest s-u path d(u) plus (u, v) edge is an s-v path of length π(v).

Consider any other s-v path P.

Let x-y be the first edge in P that leaves S.

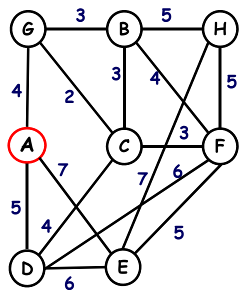
Proof of Correctness

Compute the length of s-x-y-v path P.



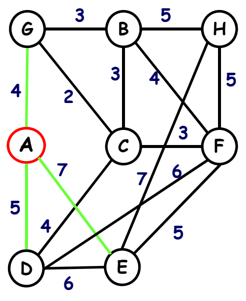
len(P) ≥ len(s,x) + len(x,y)	since len(y,v) ≥ 0
= d(x) + len(x,y)	IH
≥ π(y)	by def. of π
≥ π(v)	Dijkstra chose v instead of y

Find a shortest path from A to each vertex using Dijkstra's algorithm.



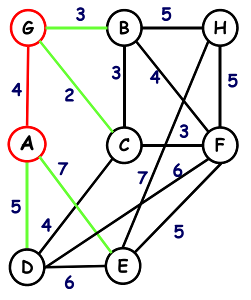
	A	B	C	D	E	F	G	H
	0	∞	∞	∞	∞	∞	∞	∞
A								

Find a shortest path from A to each vertex using Dijkstra's algorithm.



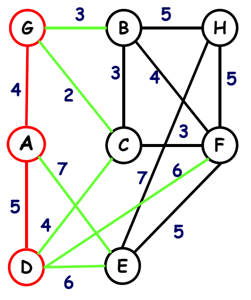
	A	B	C	D	E	F	G	H
	0	∞	∞	∞	∞	∞	∞	∞
A				5	7		4	

Find a shortest path from A to each vertex using Dijkstra's algorithm.



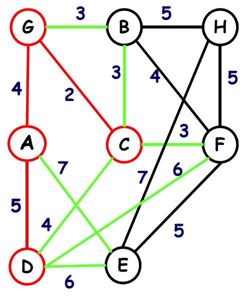
	A	B	C	D	E	F	G	H
	0	∞	∞	∞	∞	∞	∞	∞
A				5	7		4	
G		7	6					

Find a shortest path from A to each vertex using Dijkstra's algorithm.



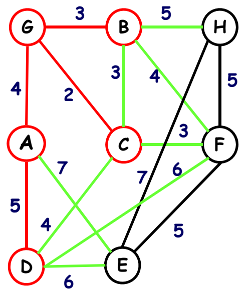
	A	B	C	D	E	F	G	H
	0	∞	∞	∞	∞	∞	∞	∞
A				5	7		4	
G		7	6					
D						11		

Find a shortest path from A to each vertex using Dijkstra's algorithm.



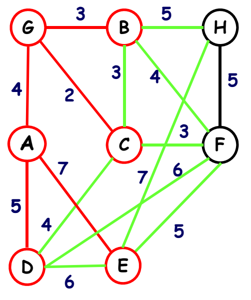
	A	B	C	D	E	F	G	H
	0	∞	∞	∞	∞	∞	∞	∞
A				5	7		4	
G		7	6					
D						11		
C						9		

Find a shortest path from A to each vertex using Dijkstra's algorithm.



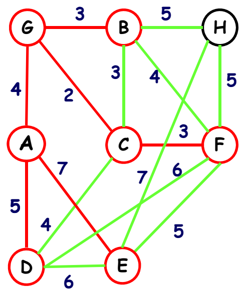
	A	B	C	D	E	F	G	H
	0	∞	∞	∞	∞	∞	∞	∞
A				5	7		4	
G		7	6					
D						11		
C						9		
B								12

Find a shortest path from A to each vertex using Dijkstra's algorithm.

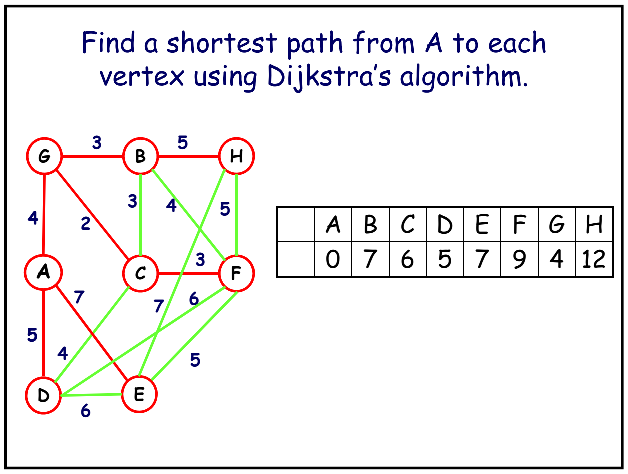


	A	B	C	D	E	F	G	H
	0	∞	∞	∞	∞	∞	∞	∞
A				5	7		4	
G		7	6					
D						11		
C						9		
B								12
E								

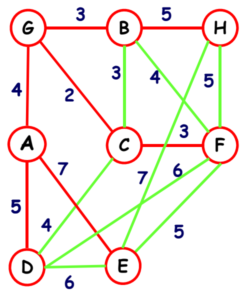
Find a shortest path from A to each vertex using Dijkstra's algorithm.



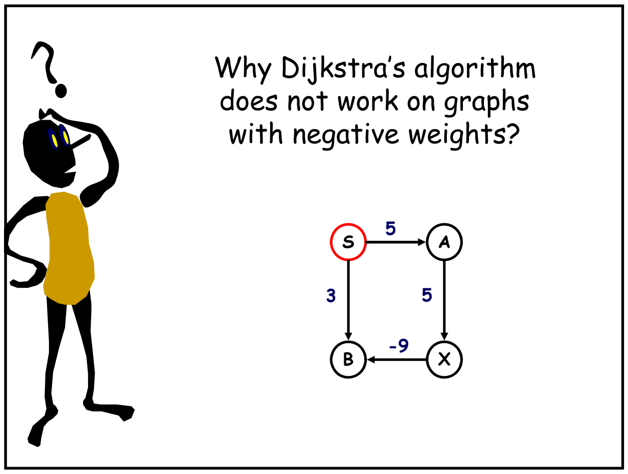
	A	B	C	D	E	F	G	H
	0	∞	∞	∞	∞	∞	∞	∞
A				5	7		4	
G		7	6					
D						11		
C						9		
B								12
E								
F								



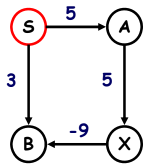
Find a shortest path from A to each vertex using Dijkstra's algorithm.

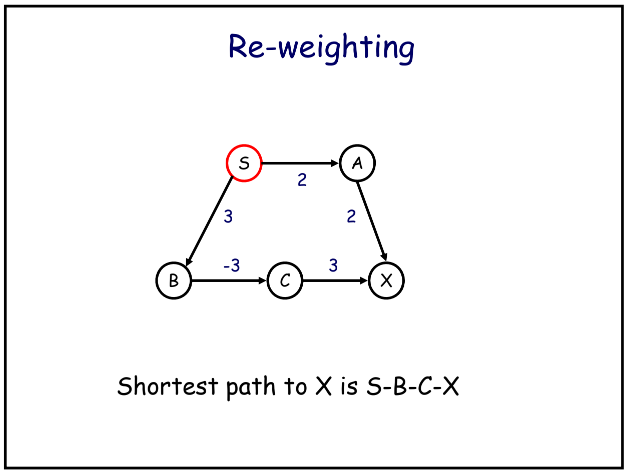


	A	B	C	D	E	F	G	H
	0	7	6	5	7	9	4	12

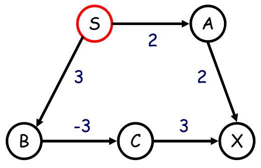


Why Dijkstra's algorithm does not work on graphs with negative weights?

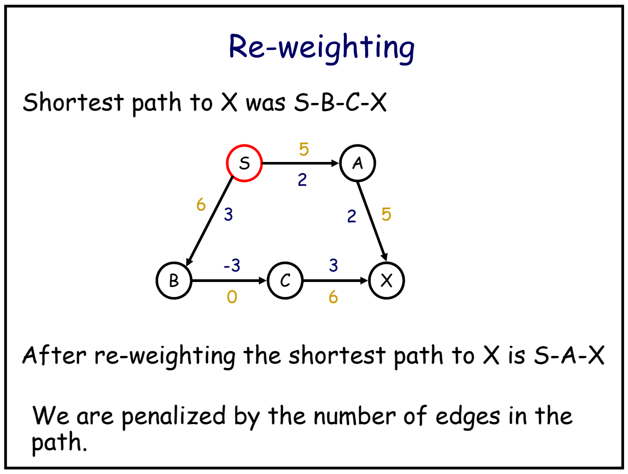




Re-weighting

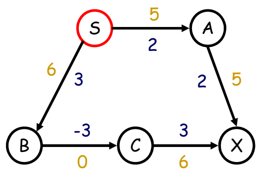


Shortest path to X is S-B-C-X



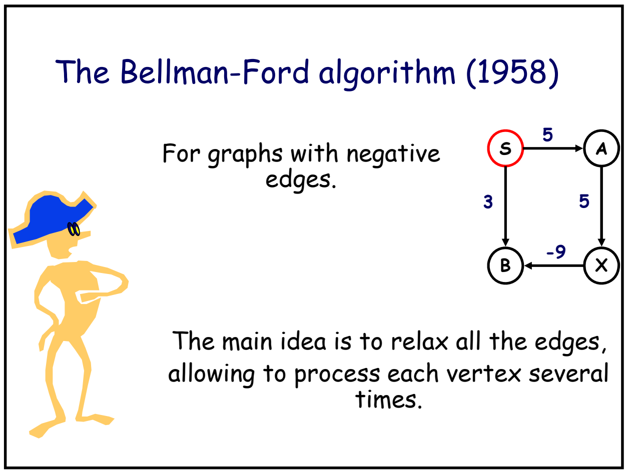
Re-weighting

Shortest path to X was S-B-C-X



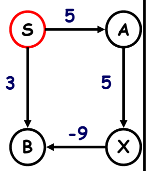
After re-weighting the shortest path to X is S-A-X

We are penalized by the number of edges in the path.

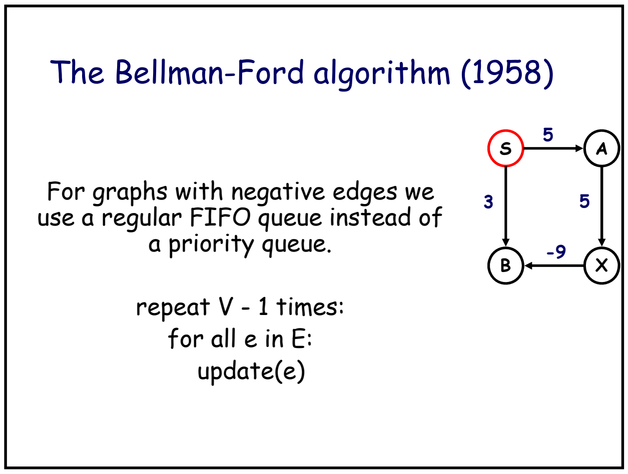


The Bellman-Ford algorithm (1958)

For graphs with negative edges.

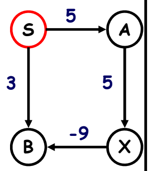


The main idea is to relax all the edges, allowing to process each vertex several times.



The Bellman-Ford algorithm (1958)

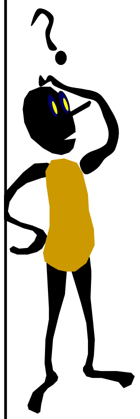
For graphs with negative edges we use a regular FIFO queue instead of a priority queue.



```
repeat V - 1 times:
    for all e in E:
        update(e)
```

The Bellman-Ford Algorithm

```
for (k = 0; k < V; k++)  dist[k] = INFINITY;
Queue q = new Queue();
dist[s] = 0; q.enqueue(s);
while (!q.isEmpty())                        ←——  V
 {
   v = q.dequeue();
   for each w in adj(v) do                  ←——  E
      if (dist[w] > dist[v] + weight[v,w]) {
           dist[w] = dist[v] + weight[v,w];
           if (!q.isInQueue(w)) q.enqueue(w);
       } }
   check for neg cycle
```



How would you terminate the Bellman-Ford algorithm if a graph has a negative cycle?

Do not stop after V-1 iteration, perform one more round. If there is such a cycle, then some distance will be reduced…