Algorithm Design IX

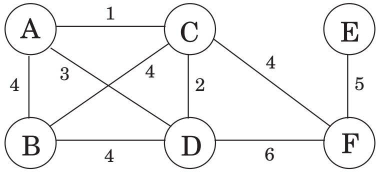
Advanced Data Structures

Guoqiang Li
School of Software

SHANGHAI JIAO TONG UNIVERSITY

Review of Previous Lectures

Build a Network

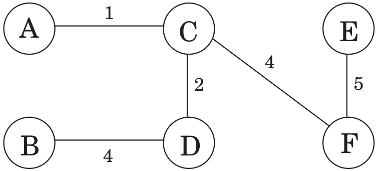


Suppose you are asked to network a collection of computers by linking selected pairs of them.

This translates into a graph problem in which

- nodes are computers,
- undirected edges are potential links, each with a maintenance cost.

Build a Network



The goal is to

- pick enough of these edges that the nodes are connected,
- the total maintenance cost is minimum.

One immediate observation is that the optimal set of edges cannot contain a cycle.

The Prim Algorithm

```
PRIM($G, w$)
input  : A connected undirected graph $G = (V, E)$, with edge weights $w_e$
output: A minimum spanning tree defined by the array $prev$

for all $u \in V$ do
    $cost(u) = \infty$; $prev(u) = nil$;
end
pick any initial node $u_0$; $cost(u_0) = 0$;
$H$ =makequeue($V$) \\ using cost-values as keys;
while $H$ is not empty do
    $v$=deletemin($H$);
    for each $(v, z) \in E$ do
        if $cost(z) > w(v, z)$ then
            $cost(v) = w(v, z)$; $prev(z) = v$; decreasekey ($H$,$z$);
        end
    end
end
```

The Kruskal's Algorithm

```
KRUSKAL($G, w$)
input  : A connected undirected graph $G = (V, E)$, with edge weight $w_e$
output: A minimum spanning tree defined by the edges $X$

for all $u \in V$ do
    makeset (u);
end
$X = \{\ \}$;
Sort the edges $E$ by weight;
for all $(u, v) \in E$ in increasing order of weight do
    if find $(u) \neq$ find $(v)$ then
        add $(u, v)$ to $X$;
        union (u,v)
    end
end
```

Priority Queue

Priority Queue

Priority queue is a data structure usually implemented by heap.

- Insert: Add a new element to the set.
- Decrease-key: Accommodate the decrease in key value of a particular element.
- Delete-min: Return the element with the smallest key, and remove it from the set.
- Make-queue: Build a priority queue out of the given elements, with the given key values. (In many implementations, this is significantly faster than inserting the elements one by one.)

The Prim Algorithm

```
PRIM($G, w$)
input  : A connected undirected graph $G = (V, E)$, with edge weights $w_e$
output: A minimum spanning tree defined by the array $prev$

for all $u \in V$ do
    $cost(u) = \infty$; $prev(u) = nil$;
end
pick any initial node $u_0$; $cost(u_0) = 0$;
$H$ =makequeue($V$) \\ using cost-values as keys;
while $H$ is not empty do
    $v$=deletemin($H$);
    for each $(v, z) \in E$ do
        if $cost(z) > w(v, z)$ then
            $cost(v) = w(v, z)$; $prev(z) = v$; decreasekey ($H$,$z$);
        end
    end
end
```

Which Heap is Best

SHANGHAI JIAO TONG UNIVERSITY

Implementation	deletemin	insert/decreasekey	$\|V\|\times$`deletemin`$+(\|V\|+\|E\|)\times$ `insert`
Array	$O(\|V\|)$	$O(1)$	$O(\|V\|^2)$
Binary heap	$O(\log \|V\|)$	$O(\log \|V\|)$	$O((\|V\|+\|E\|)\log \|V\|)$
d-ary heap	$O(\frac{d\log \|V\|}{\log d})$	$O(\frac{\log \|V\|}{\log d})$	$O(\frac{(d\|V\|+\|E\|)\log \|V\|}{\log d})$
Fibonacci heap	$O(\log \|V\|)$	$O(1)$ (amortized)	$O(\|V\|\log \|V\|+\|E\|)$

Which heap is Best

SHANGHAI JIAO TONG UNIVERSITY

A naive array implementation gives a respectable time complexity of $O(|V|^2)$, whereas with a binary heap we get $O((|V| + |E|) \log |V|)$. Which is preferable?

This depends on whether the graph is sparse or dense.

- $|E|$ is less than $|V|^2$. If it is $\Omega(|V|^2)$, then clearly the array implementation is the faster.
- On the other hand, the binary heap becomes preferable as soon as $|E|$ dips below $|V|^2 / \log |V|$.
- The d-ary heap is a generalization of the binary heap and leads to a running time that is a function of d. The optimal choice is $d \approx |E|/|V|$;

Array

The simplest implementation is as an unordered array of key values for all potential elements.

An insert or `decreasekey` is fast, because it just involves adjusting a key value, an $O(1)$ operation.

To `deletemin`, on the other hand, requires a linear-time scan of the list.

Binary Heap

Elements are stored in a complete binary tree.

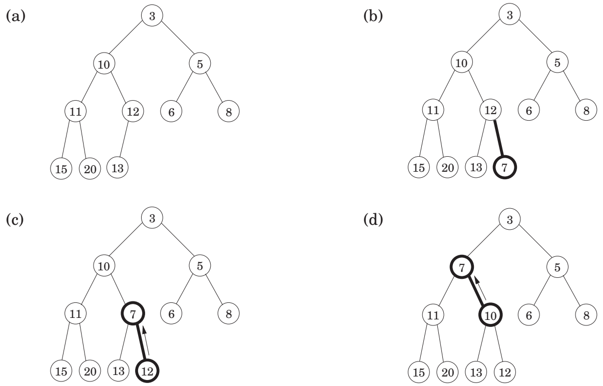
In addition, The key value of any node of the tree is less than or equal to that of its children.

In particular, therefore, the root always contains the smallest element.

Binary Heap

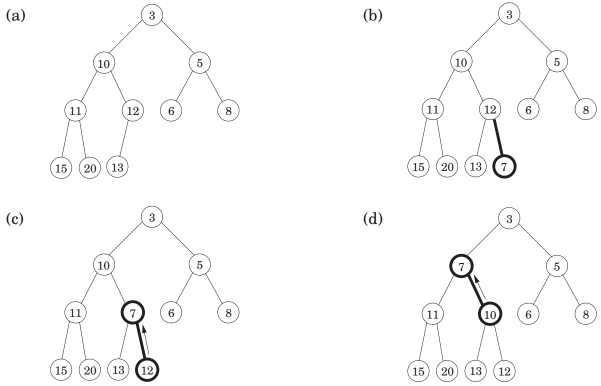
To insert, place the new element at the bottom of the tree (in the first available position), and let it "bubble up".

The number of swaps is at most the height of the tree $\lfloor \log_2 n \rfloor$, when there are n elements.



Binary Heap

A `decreasekey` is similar, except the element is already in the tree, so we let it bubble up from its current position.

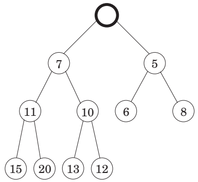


Binary Heap

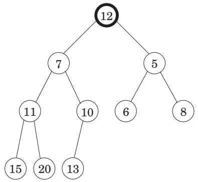
To `deletemin`, return the root value, and remove it from the heap, take the last node in the tree (in the rightmost position in the bottom row) and place it at the root.

Then let it "shift down". Again this takes $O(\log n)$ time.

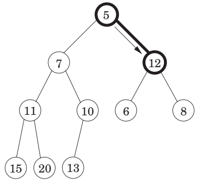
(e)



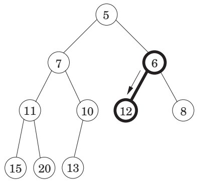
(f)



(g)



(h)



The Implementation of Binary Heap

The regularity of a complete binary tree makes it easy to represent using an array.

The tree nodes have a natural ordering: row by row, starting at the root and moving left to right within each row.

If there are n nodes, this ordering specifies their positions $1, 2, \ldots, n$ within the array.

Moving up and down the tree is easily simulated on the array, using the fact that node number j has parent $\lfloor j/2 \rfloor$ and children $2j$ and $2j+1$.

d-ary heap

A d-ary heap is identical to a binary heap, except that nodes have d children.

This reduces the height of a tree with n elements to

$$\Theta(\log_d n) = \Theta((\log n)/(\log d))$$

`Inserts` are therefore speeded up by a factor of $\Theta(\log d)$.

`Deletemin` operations, however, take a little longer, namely $O(d \log_d n)$.

Fibonacci Heap

A Fibonacci heap is a collection of min-heap-ordered trees. Trees within Fibonacci heaps are rooted but unordered linked by a circular, doubly linked list.

Each node x contains a pointer $p[x]$ to its parent and a pointer $child[x]$ to any one of its children.

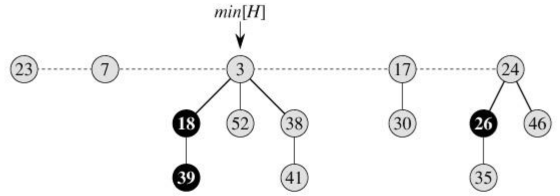
The children of x are linked together in a circular, doubly linked list, which we call the child list of x.

Each child y in a child list has pointers $left[y]$ and $right[y]$ that point to y's left and right siblings, respectively.

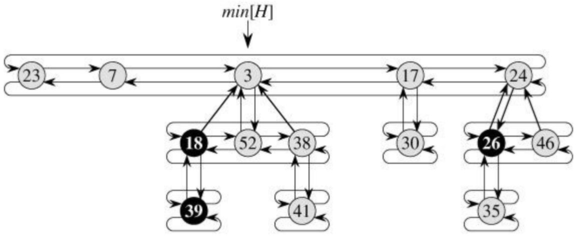
If node y is an only child, then $left[y] = right[y] = y$. The order in which siblings appear in a child list is arbitrary.

Fibonacci Heaps

(a)



(b)



Priority Queue Review

Implementation	deletemin	insert/decreasekey	$\|V\|\times$`deletemin`$+(\|V\|+\|E\|)\times$ `insert`
Array	$O(\|V\|)$	$O(1)$	$O(\|V\|^2)$
Binary heap	$O(\log\|V\|)$	$O(\log\|V\|)$	$O((\|V\|+\|E\|)\log\|V\|)$
d-ary heap	$O(\frac{d\log\|V\|}{\log d})$	$O(\frac{\log\|V\|}{\log d})$	$O(\frac{(d\|V\|+\|E\|)\log\|V\|}{\log d})$
Fibonacci heap	$O(\log\|V\|)$	$O(1)$ (amortized)	$O(\|V\|\log\|V\|+\|E\|)$

Disjoint Set

The Kruskal's Algorithm

```
KRUSKAL($G, w$)
input  : A connected undirected graph $G = (V, E)$, with edge weight $w_e$
output: A minimum spanning tree defined by the edges $X$

for all $u \in V$ do
    makeset (u);
end
$X = \{ \}$;
Sort the edges $E$ by weight;
for all $(u, v) \in E$ in increasing order of weight do
    if find $(u) \neq$ find $(v)$ then
        add $(u, v)$ to $X$;
        union (u,v)
    end
end
```

Disjoint Sets

`makeset`(x)	create a singleton set containing x	$\lvert V\rvert$
`find`(x)	find the set that x belong to	$2 \cdot \lvert E\rvert$
`union`(x, y)	merge the sets containing x and y	$\lvert V\rvert - 1$

Disjoint Sets

Union by rank

We store a set is by a directed tree. Nodes of the tree are elements of the set, arranged in no particular order, and each has parent pointers that eventually lead up to the root of the tree.

This root element is a convenient representative, or name, for the set. It is distinguished from the other elements by the fact that its parent pointer is a self-loop.

In addition to a parent pointer π, each node also has a rank that, for the time being, should be interpreted as the height of the subtree hanging from that node.

Union by Rank

```
MAKESET($x$)
$\pi(x) = x$;
rank(x)=0;
```

```
FIND($x$)
while $x \neq \pi(x)$ do
    $x = \pi(x)$;
end
return($x$);
```

MAKESET is a constant-time operation.

FIND follows parent pointers to the root of the tree and therefore takes time proportional to the height of the tree.

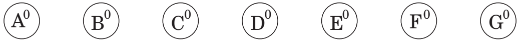
The tree actually gets built via the third operation, UNION, and so we must make sure that this procedure keeps trees shallow.

Union

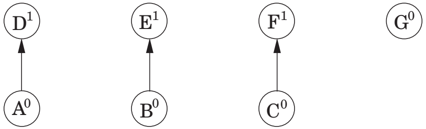
```
UNION($x, y$)
$r_x$=find($x$);
$r_y$=find($y$);
if $r_x = r_y$ then return;
if rank($r_x$)>rank($r_y$) then
    $\pi(r_y) = r_x$;
end
else
    $\pi(r_x) = r_y$;
    if rank($r_x$)=rank($r_y$) then rank($r_y$)=rank($r_y$)+1;
end
```

An Example

After $\texttt{makeset}(A), \texttt{makeset}(B), \ldots, \texttt{makeset}(G)$:

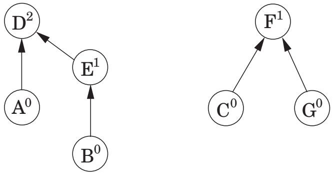


After $\texttt{union}(A, D), \texttt{union}(B, E), \texttt{union}(C, F)$:

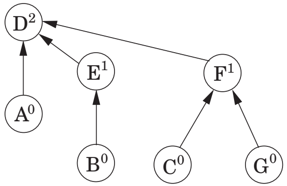


An Example

After `union`(C, G), `union`(E, A):



After `union`(B, G):



Properties

SHANGHAI JIAO TONG UNIVERSITY

Lemma (1)

For any non-root x, $\texttt{rank}(x) < \texttt{rank}(\pi(x))$.

Proof Sketch:

By design, the rank of a node is exactly the height of the subtree rooted at that node. This means, for instance, that as you move up a path toward a root node, the rank values along the way are strictly increasing.

Properties

SHANGHAI JIAO TONG UNIVERSITY

Lemma (2)

Any root node of `rank` k has least 2^k nodes in its tree.

Proof Sketch:

A root node with rank k is created by the merger of two trees with roots of rank $k-1$. By induction to get the results.

Properties

Lemma (3)

If there are n elements overall, there can be at most $n/2^k$ nodes of `rank` k.

Proof Sketch:

A node of rank k has at least 2^k descendants.

Any internal node was once a root, and neither its rank nor its set of descendants has changed since then.

Different rank-k nodes cannot have common descendants. Any element has at most one ancestor of rank k.

The Analysis

With the data structure as presented so far, the total time for Kruskal's algorithm becomes

- $O(|E|\log|V|)$ for sorting the edges ($\log|V| \approx \log|E|$),
- $O(|E|\log|V|)$ for the `union` and `find` operations that dominate the rest of the algorithm.

But what if the edges are given to us sorted? Or if the weights are small (say, $O(|E|)$) so that sorting can be done in linear time?

Then the data structure part becomes the bottleneck!

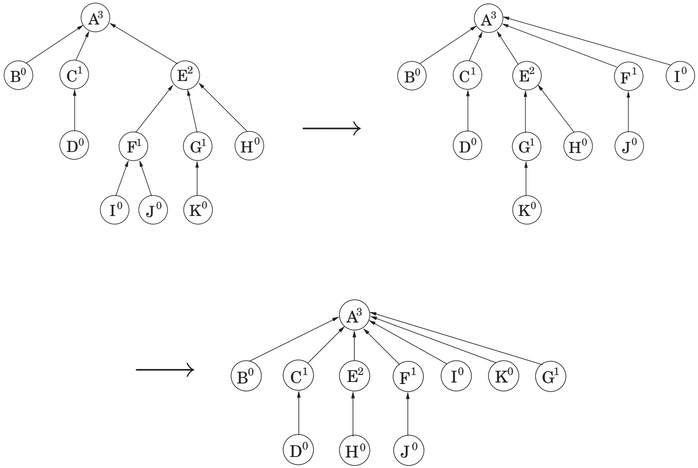
The main question:

How can we perform `union`'s and `find`'s faster than $\log n$?

Path Compression

```
FIND($x$)
if $x \neq \pi(x)$ then $\pi(x)$=FIND($\pi(x)$);
return($\pi(x)$);
```

Path Compression



Path Compression

SHANGHAI JIAO TONG UNIVERSITY

The benefit of this simple alteration is long-term rather than instantaneous and thus necessitates a particular kind of analysis.

We need to look at sequences of `find` and `union` operations, starting from an empty data structure, and determine the average time per operation.

This amortized cost turns out to be just barely more than $O(1)$, down from the earlier $O(\log n)$.

Time Analysis

Think of the data structure as having a “top level” consisting of the root nodes, and below it, the insides of the trees.

There is a division of labor:

- `find` operations (with or without path compression) only touch the insides of trees,
- `union`’s only look at the top level.

Thus path compression has no effect on `union` operations and leaves the top level unchanged.

Time Analysis

We know that the ranks of root nodes are unaltered, but what about non-root nodes?

The key point is that once a node ceases to be a root, it never resurfaces, and its rank is forever fixed.

Therefore the ranks of all nodes are unchanged by path compression, even though these numbers can no longer be interpreted as tree heights.

In particular,

- For any x that is not a root, $rank(x) < rank(\pi(x))$
- Any root node of `rank` k has least 2^k nodes in its tree.
- If there are n elements overall, there can be at most $n/2^k$ nodes of `rank` k.

Time Analysis

If there are n elements, their rank values can range from 0 to $\log n$.

Divide the nonzero part of this range into the following intervals:

$$\{1\}, \{2\}, \{3,4\}, \{5,6,\ldots,16\}, \{17,18,\ldots,2^{16}=65536\},$$
$$\{65537, 65538, ..., 2^{65536}\}, \ldots$$

Each group is of the form $\{k+1, k+2, \ldots, 2^k\}$ where k is a power of 2.

The number of groups is $\log^* n$, which is defined to be the number of successive $\log$ operations that need to be applied to n to bring it down to 1 (or below 1).

- For instance, $\log^* 1000 = 4$ since $\log\log\log\log 1000 \leq 1$.
- In practice there will just be the first five of the intervals shown; more are needed only if $n > 2^{65536}$, in other words never.

Time Analysis

In a sequence of `find` operations, some may take longer than others.

We'll bound the overall running time using some creative accounting.

We will give each node a certain amount of pocket money, such that the total money doled out is at most $n \log^* n$ dollars.

We will then show that each `find` takes $O(\log^* n)$ steps, plus some additional amount of time that can be paid for using the pocket money of the nodes involved - one dollar per unit of time.

Thus the overall time for m `find`'s is $O(m \log^* n)$ plus at most $O(n \log^* n)$.

Time Analysis

A node receives its allowance as soon as it ceases to be a root, at which point its rank is fixed.

If this rank lies in the interval $\{k+1, \ldots, 2^k\}$, the node receives 2^k dollars.

By Lemma 3, the number of nodes with $\texttt{rank} > k$ is bounded by

$$\frac{n}{2^{k+1}} + \frac{n}{2^{k+2}} + \ldots \leq \frac{n}{2^k}$$

Therefore the total money given to nodes in this particular interval is at most n dollars, and since there are $\log^* n$ intervals, the total money disbursed to all nodes is $n \log^* n$.

Time Analysis

Now, the time taken by a specific `find` is simply the number of pointers followed.

Consider the ascending rank values along this chain of nodes up to the root.

Nodes x on the chain fall into two categories:

- either the rank of $\pi(x)$ is in a higher interval than the rank of x,
- or else it lies in the same interval.

There are at most $\log^* n$ nodes of the first type, so the work done on them takes $O(\log^* n)$ time.

The remaining nodes - whose parents' ranks are in the same interval as theirs - have to pay a dollar out of their pocket money for their processing time.

Time Analysis

This only works if the initial allowance of each node x is enough to cover all of its payments in the sequence of find operations.

Here's the crucial observation: each time x pays a dollar, its parent changes to one of higher rank.

Therefore, if x's rank lies in the interval $\{k+1, \ldots, 2^k\}$, it has to pay at most 2^k dollars before its parent's rank is in a higher interval; whereupon it never has to pay again.