Lecture 13 - Dijkstra's algorithm

We have a set of elements - each with an associated value called key.

- in Dijkstra's algorithm, the set is the set V of vertices and the key value associated with vertex v is $d[v]$, which is the distance estimate from s.

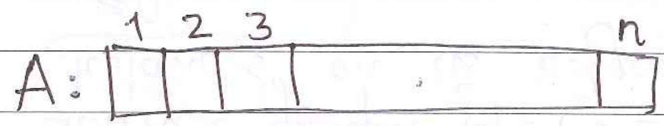
We want to design a data structure Q for storing such a set of elements such that the following operations can be efficiently performed:

- extract-min(Q)
- decrease-key(v, k) where $v \in Q$ and $k < d[v]$

Goal: Perform m decrease-key operations and n extract-min operations in $O(m + n\log n)$ time.

We considered 2 options so far:

(1) maintain Q as an array $A[1..n]$.
We assume the vertices are numbered $1, 2, \ldots, n$.

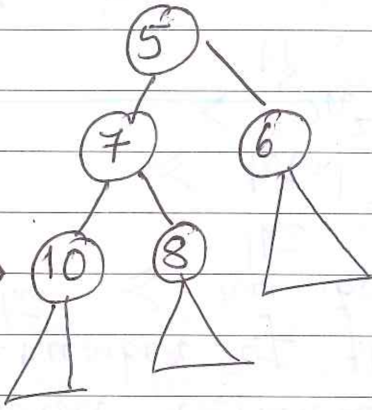


Store $d[v]$ in $A[v]$. Here decrease-key takes $O(1)$ time. However extract-min takes $\Theta(n)$ time.

(2) maintain Q as a min-heap.
Here we will have 2 arrays: a look-up array $L[1..n]$ and a min-heap array A.

- for any vertex v, $L[v]$ tells us the location in array A where v is currently placed. Here decrease-key takes $O(\log n)$ time.

In order to make decrease-key operation take O(1) time, we had the following idea:



Decrease this key value to 4.

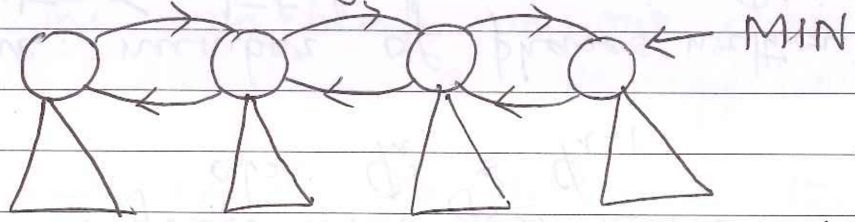
← This is a min-heap. We are just writing the key values here.

Our idea was to cut off the subtree rooted at the node whose key value got decreased to 4 and start a new tree.

- So we will have a collection of min-heap ordered trees now.

There will be a doubly linked list of root nodes

Root List:



Observe that each tree is min-heap ordered. So the min key value is associated with a root and the MIN pointer will point to the root node which the min key value.

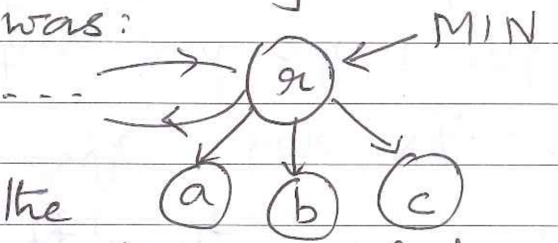
We will again have a look-up array $\mathcal{L}[1..n]$ where for any vertex v, $\mathcal{L}[v]$ will have a pointer pointing to the node in our data structure which stores v's key value.

Let us look at the extract-min operation now.

- Thanks to the MIN pointer, we know which node has the min key value.

- We need to remove this node (pointed to by MIN) from the root list.

- What about the children of this node? Say, the picture was:



- Once we remove the node with r from the root list, we have to accommodate the descendants of r somewhere.
  * We will make a root of each of r's children. So the nodes with a, b, c will be added to the root list.

Remark: Though each node had ≤ 2 children in the min-heap, in our current data structure, a node may have more children.

The main task now is to determine the new node that MIN pointer should point it.
- for this we need to scan all nodes in the root list (and also all of r's children: these are already in the root list).

Recall that our goal is to implement extract-min in $O(\log n)$ amortized time.
So while scanning the root list, we will also clean up this data structure.

Extract-min
1. Remove the node pointed to by MIN from the root list after adding its ~~2. Add~~ children to the root list.

2. Link root nodes of equal degree until at most 1 root remains of each degree.
   - find 2 roots x and y in the root list with the same degree.
     - let $d[y] \geq d[x]$
     - link y to x, i.e, y becomes a child of x : so y is deleted from the root list.

So when we are scanning the root list to find the new "minimum", we want to store this hard work that we are doing
- this is the clean-up step.
  * when we discover 2 root nodes with the same degree, i.e, the same number of children, we make the one with higher key value the child of the other.

So the clean-up step ensures that there is at most 1 root node of any degree. Let us write down the pseudo-code for Clean-Up.

Clean-Up(H) (∵ H is our data structure which is a collection of min-heap ordered trees)

We will use an array $A[0..D]$ here where D = maximum degree possible

1. For $i = 0$ to D do : $A[i] = nil$
2. For each node w in the root list of H do :
   - $x = w$
   - $d = deg(x)$
   - while $A[d] \neq nil$ do
     { $*\, y = A[d]$

* if $d[x] > d[y]$ then exchange x & y.
* link y to x.
* $A[d] = $ nil.
* $d = d+1$

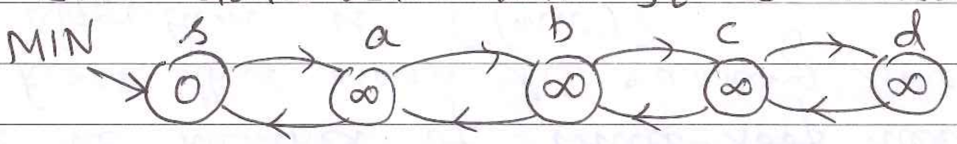
}

• $A[d] = x$

3. Determine min(H) and make the root list of H using A.

Let us run the above pseudo-code on an example. When we start Dijkstra's algorithm, we set $d[s] = 0$ and $d[u] = \infty \ \forall\ u \in V - \{s\}$.

The data structure H is initialized as follows:



There are 4 vertices a, b, c, d here.

The first Extract-min operation deletes s from this data structure and runs the clean-up step.

A:

0	1	2	3	...
nil	nil	..	- -	- - -

- The first root node we visit is a and its degree is 0. So it is entered in A[0].

A:

0	1	2	
a	nil	nil	

- Then we visit b and its deg is also 0 and a is already located in A[0]. Since $d[b] \geq d[a]$, we make b the child of a: so a's degree is 1.

A:

0	1	2	
nil	a	nil	

- Then we visit c. Its deg is 0. So it is entered in A[0] (which is now empty)

A:

0	1	2	
c	a	nil	

A:

0	1	2	
nil	nil	a	

← Then we visit d, its deg is 0, however A[0] is occupied by c, so d becomes c's child.

So c wants to enter in A[1], however a is already there. So c becomes a's child; a's deg is 2 now.

What is the work done during this Extract-min?
$= \underbrace{O(D)}$ + number of roots deleted

- This counts the work done in making a root out of each of the previous MIN's children.
  * D is the maximum degree possible for any node in our data structure.

- this also counts the work in initializing the array A in Clean-Up and traversing the array at the end to find min(H) and making $\underbrace{\text{the new root list}}$.

  This has $\leq D+1$ nodes since we will have at most 1 root node of any degree between 0 and D.

Total work done during all extract-min operations = total number of roots deleted $+ O(D) \cdot n$

Question: How many roots are deleted in the entire algorithm?

Total number of roots deleted
$\leq$ Total number of roots created $+ \underbrace{n}$

Question: How many roots are created?

during initialization

Total number of roots created in the entire algorithm
$\leq$ Total number of decrease-key operations
$+ \underbrace{O(D)}$

Each decrease-key operation creates at most 1 new root

Each of MIN's children becomes a root & every node has $\leq D$ children.

$= m + O(D) \cdot n$

So each decr-key costs O(1) & amortized cost of Extr-Min is O(D).