Algorithms 2005

Ramesh Hariharan

Amortization in Dynamic Algorithms

- A single insertion/deletion might take say O(log n) time
- Does a sequence of n insertions or deletions take O(nlog n) time? Could it take less?
- What is the amortized time per insertion, i.e., total time divided by number of insertions?

Amortization Example 1

- Consider a data structure supporting deletes, find-mins, insert where a delete(x) operation should delete x and all items bigger than x and insert always inserts an item larger than what is already there.

- What data structure will you keep for this?

- What is the worst case time taken per deletion/find-min? What is the amortized time taken per deletion/find-min over a sequence of n deletion operations starting with a data structure having n inserts?

  - O(n) worst case, O(1) amortized for deletion
  - O(1) worst case for find-min

Amortization Example 1

- Keep a sorted list, find min in O(1) time
- Each deletion knocks out as many items as the time spent.
  Charge the time spent on this deletion to the items knocked out, one unit per item.
- Total Time taken over all deletions is the sum of the charges on all items.
- Each item can be knocked out by at most one deletion, so each item is charged only 1 unit over all deletions.
- So total time over all n deletions is at most O(n), i.e, O(1) amortized time per deletion.

Amortization Example 2

- Given a string A[0..n-1] of n bits, initially set to 0
- Treat this string as a binary number and add a 1 to this number m<2^n times; each addition operation will start at some specified A[j] and scan through the higher order digits until the carrying-over process stops.
- Worst case time per addition is O(n).
- What is the amortized time per addition?

Amortization Example 2

- Consider the total number of 1s in the string. This is a potential function.
- Each addition add at most one 1 and reduces t-1 1s to 0s, where t is the number of bits scanned by this addition.
- So Δ Pf<=2-t, for a particular addition.
- $\Sigma \Delta$ Pf <= Σ (2-t), sum over all additions.
- Total time taken
  2*number of additions - $\Sigma \Delta$ Pf < 2*number of additions
- Amortized time taken per addition is < 2

Amortization Example 2

A less notational argument

- Each addition operation pumps in at most one 1.
- The total number of ones ever pumped into the system is at most #additions.
- The total number of 1s that can be removed from the system is at most the number of ones pumped in.
- The time taken by an addition is at most 1 + the number of ones removed
- The total time over all additions is thus at most 2#additions

Amortization Example 3

- How much time does a sequence of n sorted insertions take in the hybrid list-array structure of size m (we discussed this structure last week)?

- It could be as high as $O(\log(n+m))$ time per insertion. But is it actually smaller than this?

Amortization Example 3

- Searching for n sorted items in the structure of size m takes only O(min(n+m,n\log(n+m)) time!! Do finger searches, i.e., search x_i only from the previously inserted item x_{i-1} onwards.

- Inserting n sorted items in the structure of size m takes only O(n+m) time!! Why? Relate the insertion time to a phyical property of the structure and show a bound on this physical property.

- Amortized time per insertion is O(min(1+m/n,log(m+n)).

Potential Function Exercise

- Given n numbers 0,0,0,..,0.
- Several iterations: each iteration removes the largest item and increases the cumulative weight of all remaining items by an amount 1, distributed in an arbitrary way over the remaining items.
- How big is the last number standing? Hint: Find a potential function.

Shortest Paths, Heaps and F-Heaps

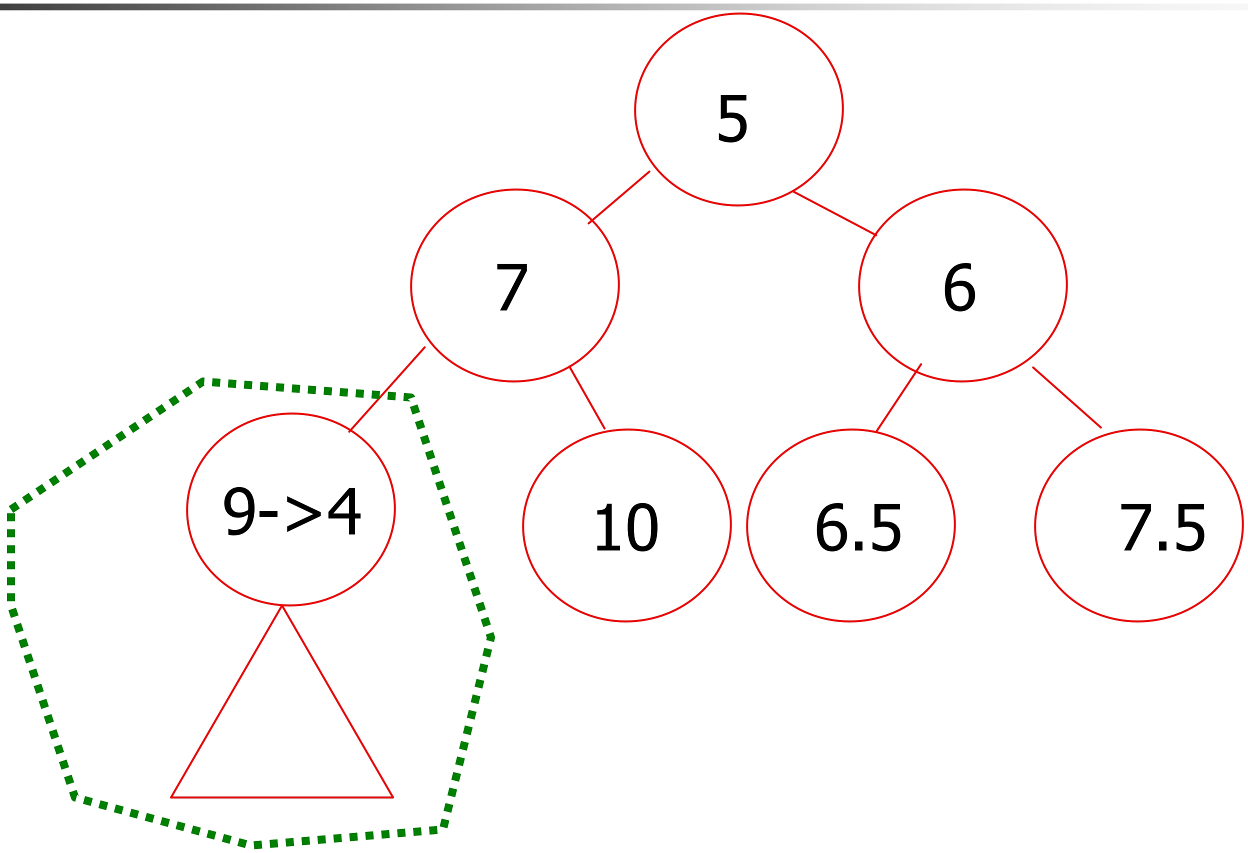
- Dijkstra’s algorithm: Generalization of BFS to weighted graphs, needs a priority queue or a heap instead of a plain queue.

```
Initialize heap to all vertices in G, with key value 0 for s and 1 for others
while heap not empty {
    x=find-and-remove-min-in-heap
    Decrease key of each neighbour y of x in the heap to
            min(key(y),key(x)+ w(y,x))
  }
```

Shortest Paths, Heaps and F-Heaps

- n Delete-Mins, m Decrease-Keys.
- Time O((n+m)log n), assuming worst case O(log n) time for decrease keys and delete mins.
- How about amortized time? Can you construct a graph in which the amortized time is also Ω (log n) per delete-min and decrease key?
- n Delete-Mins necessarily take Ω (nlog n). Why?
- Can m Decrease-Keys be made O(m)?

Decrease Key



Note: half the items are at the bottom, and decrease keys at the bottom have to percolate all the way upwards:

IDEA: Why not just cut off the subtree and create a new tree

F-Heaps: A Forest of heaps

- Each heap has the min at top property but not the balanced property?
- Find Min has to go through the min item in all the trees in the forest; so the number of trees has to be capped at O(log n) somehow.
- Each decrease-key creates a new tree in the forest, so trees will have to be merged to maintain the above cap. Can we merge 2 trees in O(1) time. Yes, but then we have to work with larger node degrees.
- Find-Mins can be implemented to create as many new trees as the number of children of the node containing the min.

F-Heaps: A Forest of heaps

- There are several trees in the forest, at most one for each degree (at the root).

- Uniqueness of degrees defines the tree addition algorithm: if the tree to be added has degree i, merge it with the tree of degree i in the heap, if any. This creates a tree of degree i+1, continue this sweep until uniqueness of degrees is ensured.

F-Heaps: Tree Structure

- What do the trees look like on account of the merges (assume first that there is no subtree cutting hapenning)?
- If the root has degree i, then the last child to be added will have degree i-1, the penultimate will have degree i-2 and so on. Each child subtree will have a recursive structure. Therefore the size of a subtree with root degree i is at least c^i (determine c exactly). It follows that the degree i must be O(log n).
- If subtrees start getting cut off, this bound no longer holds..

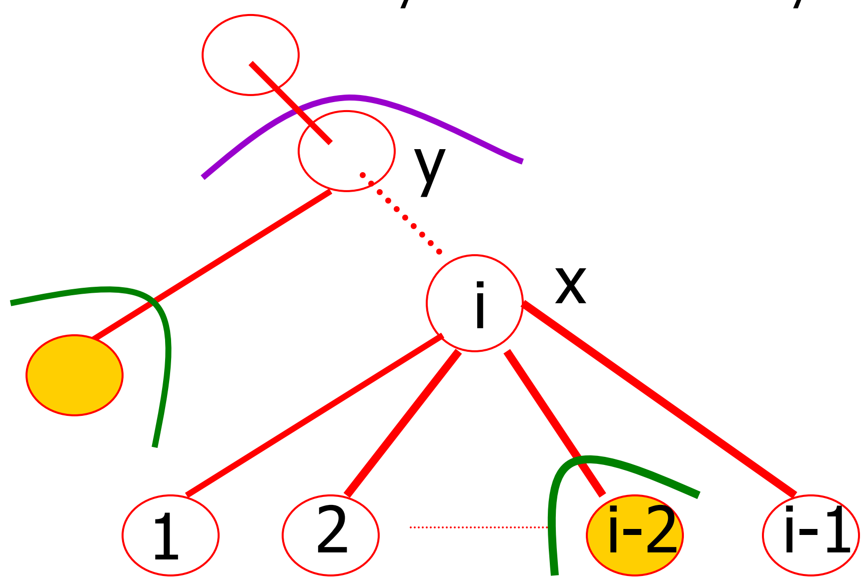
F-Heaps: Analysis

- The total time taken for all the new tree additions is just proportional to the number of such tree additions (think of the presence of a particular degree as 1 and absence as 0, the sweeps are now reminiscent of the bit addition problem described earlier). The number of tree additions is O(nlog n+m), one per decrease key and *up to log n per delete-min.*

- If subtrees start getting cut off, then the log n per delete min could go up, as degrees could be much larger than log n (why? subtree cutting seems to reduce degrees rather than increasing them)

F-Heaps: Restricting Degree

- Subtree cutting at the root is fine (why?). Never allow subtree cutting at non root nodes to destroy the structure very much.



- Allow one child of a non-root node x to be cut; for the second cut, instead cut above the nearest ancestor y of x such that parent of y is either a root or has not had a child removed so far. The sequence of nodes x..y traversed in this process is called a cascade.

- Cost for the cascade is charged to the previous cuts (each node in a cascade has had child-loss previously)

F-Heaps: Invariants

- A node x has two states:
  - Untouched (no child removed since x became untouched)
  - Touched (1 child removed since x became untouched last)

- An untouched node with degree i has
  - Smallest child degree>=0
  - Second smallest child degree>=1
  - Third smallest child degree >=2 and so on until i-1

- An touched node with degree i has
  - Smallest child degree>=0
  - Second smallest child degree>=1
  - Third smallest child degree >=2 and so on until i-2

- The degree of a touched node with actual-degree i-1 is i.
- The root of a tree is always untouched.
- An untouched non-root node with degree i becomes a touched node with actual-degree i-1 when one of its children is removed.
- A touched node x becomes untouched when it is part of a cascade.

Maintaining Invariants

- UT(i): Min number of nodes in a subtree rooted at an untouched node of degree i
- TT(i): Min number of nodes in a subtree rooted at a touched node of actual-degree i-1

Inductively, by the invariant,

- UT(i)>=TT(i-1)+TT(i-2)+…TT(0) for untouched degree i
- TT(i)>= TT(i-2)+…TT(0) for touched actual-degree i-1

- Untouched of degree i to Touched of actual-degree i-1: Invariant maintained
  TT(i)>=TT(i-2)+…TT(0)
- Touched of actual-degree i-1 to Untouched of degree i-1: Invariant Maintained
  UT(i-1)>=TT(i-2)+…TT(0)
- Increase in degree from i to i+1 due to merging: Invariant maintained
  UT(i+1)>=UT(i)+UT(i)>=TT(i) + TT(i-1)+TT(i-2)+…TT(0)

F-Heaps: A Forest of heaps

- $UT(I) >= 1.6^{i}$, therefore degrees are O(log n)
- So total number of tree additions: m+nlog n
  - Amortized time per tree addition is O(1). Recall the bit addition problem we did earlier?
  - Find Min takes time O(log n).
  - m decrease keys take O(1) time each.
  - Cascades are charged to the above tree additions, O(1) per tree addition.
- Total time for Shortest Paths is now O(m+nlog n).

Minimum Spanning Tree

- Given a spanning tree, least cost network connecting all nodes
- Least weight edge must be in the network
- Algo: Contract least weight edge and recurse on resulting graph.
- Leads to self-loops: New algorithm handling self loops

  Contract least weight non-self loop edge and recurse on resulting graph.

Time taken: mlog n for sorting edges

m calls to self-loop check

n contractions

Disjoint Set Union-Find

Contractions create disjoint sets of vertices

- Self Loop checking involves checking if the two endpoints belong to the same set
- Contraction invoves unioning two sets.

m finds, n Unions

How do we implement a data structure for this?

Disjoint Set Union-Find

Simple approach

- Keep an array of vertices
- Each vertex stores its set number in the array
- Find is O(1) time
- Union requires changing the set numbers in one of the two sets. Which one?

Time: Find O(1), Union O(log n) amortized over n unions.

Disjoint Set Union-Find

Analysis

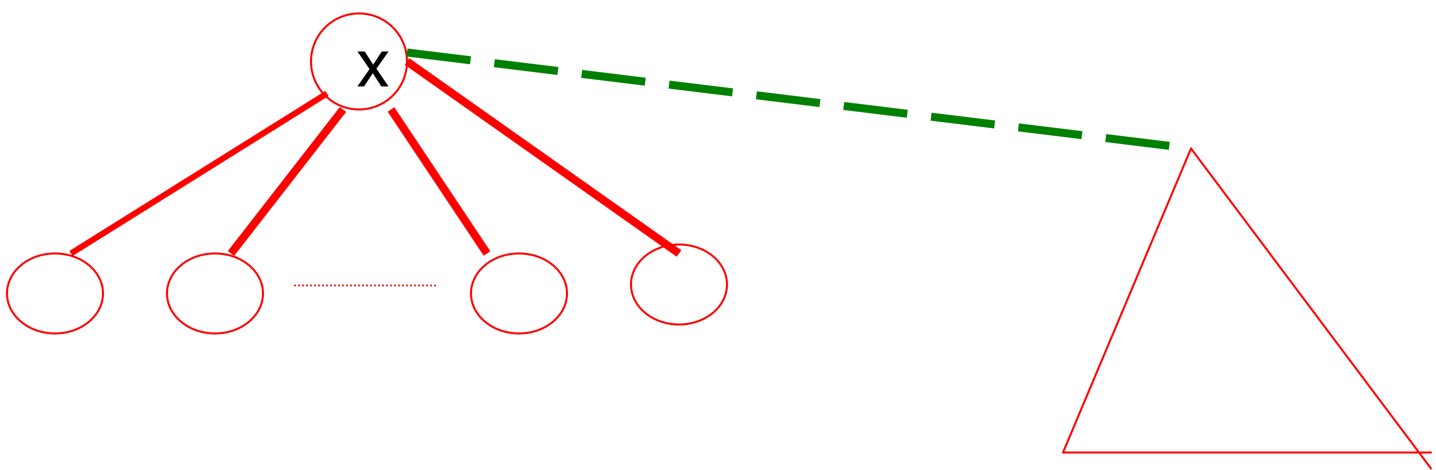
- How many times does an item change set labels?
- If an item changes set labels i times, it must be in a set of size >=2^i
- So unions take O(nlog n) on the whole

Disjoint Set Union-Find: Another Algorithm

Can we merge faster, in O(1) time?

Nodes in a set point to a common location containing the representative set element.

O(1) merge is as shown; but this leads to trees with increased depth, so the time for a find goes up.

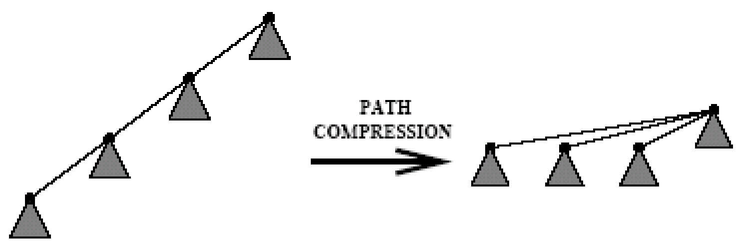


Which tree becomes the parent? The one with larger height.

Disjoint Set Union-Find: Path Compression

Each find can also reduce the tree depth, at no extra cost. This can change degrees of nodes as we go along.

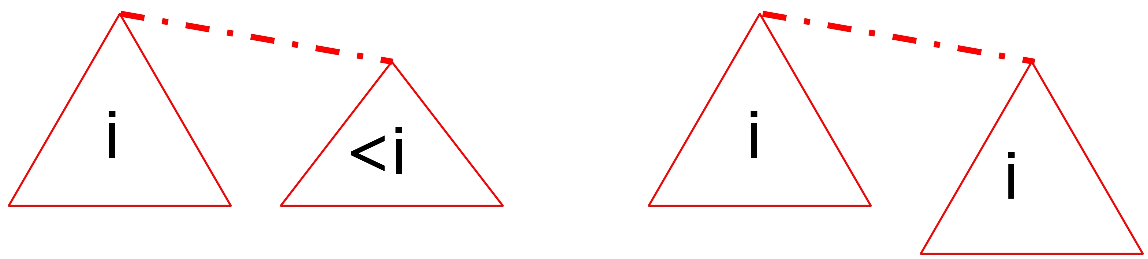
Rank: Height assuming no compression happens. Unions are done by rank and not by height (why?)



What is the total time taken? How many times does an element change parents?

Disjoint Set Union-Find: Path Compression

- A node with rank i has subtree-size at least 2^i.
- The parent of a node x has rank strictly larger than the rank of x.
- The final rank of a node is frozen when it ceases to be a root.



Therefore: the number of rank i nodes is at most $n/2^i$ and $i<=\log n$

Disjoint Set Union-Find: Path Compression

Each unit time spent in a find changes the parent of a node, the new parent has a larger rank than the previous parent. So the rank of the parent of a node keeps going up.

How many times does a node change parents? Clearly at most log n times. So all finds together take O(nlog n+m).

Can one do better?

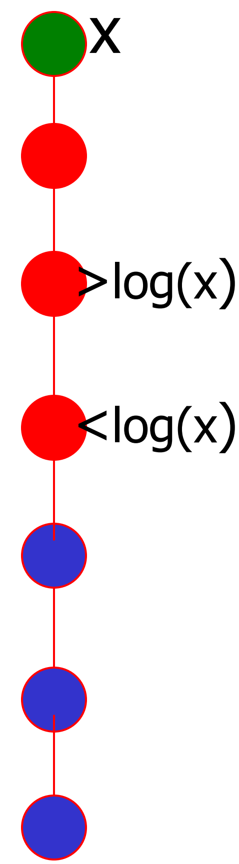
Disjoint Set Union-Find: Path Compression

- For the time to be as large as nlog n, most nodes must climb up one rank level at a time
- If most nodes climb up one step at a time, then the corresponding finds will themselves take only constant time each.
- If finds take more than constant time each, then nodes will jump upwards at a faster rate, so there will be fewer change of parents.

Trade-off between time taken for a find and the distance by which nodes jump in levels.

Disjoint Set Union-Find: Path Compression

- Suppose a node currently has a parent of rank k
- Two kinds work done by a find
  - **those which now give it a parent of rank >= 2^k: blue**
  - **those which now give it a new parent of rank k..2^k-1: red**
- The work done by a find on the various nodes it encounters can be partitioned into blue work and red work



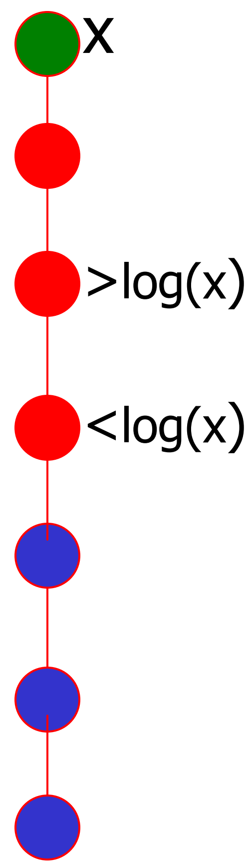
- Total blue work doable is at most nlog* n
- It remains to count the total red work
  - Red work moves a node x of rank i to a parent of rank at most 2^i
  - This can be done at most 2^i times for x
  - Adding this over all nodes of rank i gives n/2^i * 2^i
  - Adding over all ranks gives nlogn, so no improvement
- **Do some chunking of ranks, chunk ranks k..2^k in one chunk**

Disjoint Set Union-Find: Path Compression

- Two kinds of finds for a node
  - **those which cause a change of chunk at the parent: blue**
  - **those which do not cause a change of chunk at the parent: red**

- The work done by a find on the various nodes it encounters can be partitioned into blue work and red work

- Total blue work doable is at most $n\log^* n$
- It remains to count the total red work
  - Red work moves a node x of rank i to another parent within the same chunk
  - This can be done at most 2^i times for x
  - Adding this over all nodes in the chunk gives at most $2*n/2^i * 2^i$
  - Adding over all chunks gives $n\log^* n$
- **Done!!**



MST Analysis

- m Finds and n Unions takes $m\log^* n + n$ time.
- Sorting mlog n dominates the time
- Total time: O(mlog n+n)

- *Can you identify why there is scope for even further improvement?*
- *Also read Seidel and Sharir for some new top down analysis..*

Thank You