COS 423 2/17/14

Heaps

©Robert E. Tarjan 2014
**Heap** (priority queue): a set of items, each item $x$ having a key $k(x)$ and possibly other information. Keys are **totally ordered**. We assume no ties.

**Basic Operations:**

- *make-heap*: Return a new, empty heap.
- *insert*(x, H): Insert x and its info into heap H.
- *delete-min*(H): Delete the item of min key from H.

This is a **min-heap**. A **max-heap** supports *delete-max* instead of *delete-min*. 
Additional Operations:

*find-min*(H): Return the item of minimum key in H.

*meld*(H₁, H₂): Combine item-disjoint heaps H₁ and H₂ into one heap, and return it.

*decrease-key*(x, k, H): Replace the key of item x in heap H by k, which is smaller than the current key of x.

*delete*(x, H): Delete item x from heap H.

**Assumption:** Heaps are item-disjoint: each item in one heap at a time.
A heap is like a dictionary but without access by key: can only retrieve the item of min key.

Operations $\text{decrease-key}(x, k, H)$ and $\text{delete}(x, H)$ are given a pointer to the location of $x$ in heap $H$.

$m = \#\text{operations}, \ n = \#\text{items in heap}$

**Applications:**
- Priority-based scheduling and allocation
- Discrete event simulation
- Network optimization: **Shortest paths**, Minimum spanning trees
Lower bound from sorting

Can sort $n$ numbers by doing $n$ inserts followed by $n$ delete-min’s.

Since sorting by binary comparisons takes $\Omega(n \log n)$ comparisons, the amortized time for either insert or delete-min must be $\Omega(\log n)$.

One can modify any heap implementation to reduce the amortized time for insert to $O(1)$ → delete-min takes $\Omega(\log n)$ amortized time.
Our goal

$O(\lg n)$ amortized time for delete-min and delete

$O(1)$ amortized time for all other operations

(Does delete really need $\Omega(\lg n)$ time?)
Heap = binary search tree

Represent a heap by a binary tree whose nodes are the items, with key order = symmetric order. The tree is a search tree.

Do a decrease-key as a delete followed by an insert. All operations except meld take $O(\lg n)$ time, worst-case if tree is balanced, amortized if self-adjusting.

**Good:** Don’t need pointer to location to do decrease-key or delete (assuming no key ties)

**Bad:** insert, decrease-key take $\Omega(\lg n)$ time, meld takes $\Omega$ time.

**Solution:** Maintain less order in the data structure.
Heap = Heap-ordered tree

**Heap order:** \( k(p(x)) \leq k(x) \) for all nodes \( x \), where \( p(x) = \text{parent of } x \). Defined for **rooted** trees (arbitrary degree), not just binary trees

Heap order \( \rightarrow \) item in root has min key

\( \rightarrow \text{find-min takes } O(1) \text{ time} \)

What tree structure? How to implement heap operations?
Two heap implementations

**Constant degree:** Implicit heap. Very simple, fast, small space. $O(lgn)$ worst-case time per operation except for *meld*.

**Non-constant degree:** Fibonacci heap. Achieves our goal.
Heap-ordered tree: constant degree

Store items in nodes of a rooted tree, in heap order by key.

*Find-min*: return item in root.

*Insert*: replace any null child by a new leaf containing the new item \( x \). To restore heap order, *swim* (sift up): while \( x \) is not in the root and \( x \) has key less than that in parent, swap \( x \) with item in parent.
**Delete-min or delete:** Delete item. To restore heap order, *sink* (sift down): while empty node is not a leaf, fill with item of smallest key in children. Either delete empty leaf, or fill with item from another leaf, swim moved item, and delete empty leaf. (Allows deletion of an arbitrary leaf, so tree shape can be controlled)

**Decrease-key:** swim.

Choice of leaf to add or delete is arbitrary: add level-by-level, delete last-in, first-out.
A binary heap
Numbers in nodes are keys.
Numbers next to nodes are order of addition.
Insert 15
delete-min: remove item in root, sink empty node
End of sinking:
swap item in last leaf
into empty leaf;
swim.
Implicit binary heap

Binary tree, nodes numbered in addition order

root = 1

children of \( v \) = 2\( v \), 2\( v \) + 1

\( p(v) = \lfloor v/2 \rfloor \)

→ no pointers needed! Can store in array

*insert:* add node \( n + 1 \)  
*delete:* delete node \( n \)

depth = \([\log n]\)
Each operation except *meld* takes \( O(\log n) \) time: 

*insert* takes \( \leq \log n \) comparisons (likely \( O(1) \): see [https://webdocs.cs.ualberta.ca/~hayward/papers/heap.pdf](https://webdocs.cs.ualberta.ca/~hayward/papers/heap.pdf))

*delete* takes \( \leq 2 \log n \) comparisons (likely \( \log n + O(1) \))

Can reduce comparisons (but not data movement) to \( \log \log n \) worst-case for *insert*, 

\( \log n + \log \log n \) for *delete*

Instead of binary, can make tree \( d \)-ary. Current consensus is that 4-ary is best in practice.
Heap-ordered tree: non-constant degree

Primitive operation *link*: combine two trees by comparing their roots, making the root with smaller key the parent of the other. This increases the degree of the new root, hence non-constant degree.
A link takes one comparison and $O(1)$ time. We will build all operations out of links and cuts (breaking a link)
Heap-ordered representation

Heap-ordered tree, children are nodes that lost links (most recent first)
Binary tree representation

*first child, next sibling* representation of heap-ordered tree
Augmented binary tree representation

To support *decrease-key*, *delete*, also need *parent, previous child*
Linking trees

One comparison, $O(1)$ time
Heap operations:

*find-min*: return item in root

*make-heap*: return a new, empty tree

*insert*: create a new, one-node tree, link with existing tree

*meld*: link two trees

*decrease-key*: if not root, cut from parent, link with root

*delete*: *decrease-key* to $-\infty$, *delete-min*
delete-min: Delete root, link trees rooted at its children.

Time is proportional to number of children: need to link in a way that keeps degrees small.

How?
delete-min

![Binary Min Heap]

- 5
- 7
- 16
- 27
- 10
- 21
- 18
- 12
- 28
- 24
- 30
- 28
Bad case

Repeated linking of trees whose roots have much different degrees, if the larger-degree root wins.

Extreme case: repeated insert, if the new root always loses to the old root. (Old root remains the root, degree grows by 1 with each insert.)
Linking by Rank

Give each node $x$ an integer $rank$, $r(x)$, 0 at first. During delete-min, link two nodes of equal rank if possible, adding 1 to the rank of the new root (*fair link*). When not possible, link any two roots, changing no ranks (*unfair link*).

Links during *insert*, *meld*, and *decrease-key* are unfair and change no ranks, even if they link nodes of equal rank.

$r(x) = \#children\ acquired\ by\ fair\ links$
Bound on rank

Theorem 1: If there are no decrease-key or delete operations, a node of rank $k$ has at least $2^k$ nodes in its subtree.

Proof: By induction on $k$. True for $k = 0$. The only way for a node $x$ of rank $k$ to increase in rank is to acquire a new child of rank $k$, giving $x$ rank $k + 1$ and increasing its number of nodes in its subtree to at least $2 \times 2^k = 2^{k+1}$ by the induction hypothesis.

Corollary 1: $r(x) \leq \lg n$. 
Implementation of *delete-min*

Use an array indexed from 0 to $\log_2 n$. After deleting the root, put each new tree into the array cell whose index is the rank of its root. If the cell is already full, link the tree in the cell with the current tree and make the tree formed by the link the current tree. Once all trees are in cells, empty all cells and link trees in any order. (Need not scan through all cells, can store indices of non-empty cells in a set.)
This data structure (without *decrease-key* and *delete*) is a one-tree version of a *binomial queue*. 
Running Time

All operations are $O(1)$-time worst-case except delete-min, which takes $O(lgn)$ time for filling and emptying array cells plus $O(1)$ time per link. A single delete-min can do many links, even $\Theta(n)$: $n$ inserts, followed by a delete-min. But can there be lots of slow delete-mins?
No!

The total time of a sequence of $m$ operations of which $k$ are $delete-mins$ is $O(m + k\lg n)$.

How do we prove this?

Amortized Analysis
The total running time is $O(m + k\lg n)$ plus $O(1)$ per link. There is at most 1 unfair link for each insert and each meld and at most $\lg n$ for each delete-min, for a total of at most $m + k\lg n$.

We need a way to count the fair links. This is where we use amortization. We do a careful accounting of events that can lead to fair links.
What is destroyed must be created
(What comes down must go up.)

We define the potential of a set of trees to be the number of trees plus the number of children that were attached to their parents by unfair links. (The term “potential” comes from physics, as we discuss later.)

Our potential bounds the number of fair links that can be done. We study how the various heap operations change it.
Changes in potential

The potential is initially zero and can never be negative.
Each fair link reduces the potential by one (one root becomes a child attached by a fair link).
Hence the number of fair links is at most the sum of any increases in the potential.
An unfair link does not change the potential (one root becomes a child attached by an unfair link).
Each insert increases the potential by 1: it creates a new root and does an unfair link. Each *meld* does not change the potential. Deleting a root at the beginning of a delete-min increases the potential by at most $\lg n + 1$, by converting at most $\lg n + 1$ children previously attached by fair links into roots.

Hence the only increases in potential are $O(1)$ per insert plus $O(\lg n)$ per delete-min.

We conclude that the number of fair links is $O(m + k\lg n)$. 
We have proved:

**Theorem 2:** If there are no *decrease-key* or *delete* operations, the total time of $m$ operations on a set of binomial queues is $O(m + k\lg n)$. 
What about *decrease-key*?

Theorem 1, which bounds the maximum rank, fails if we allow decrease-key operations, which can arbitrarily prune a tree. We need an inexpensive way to keep track of such pruning. We give each child $x$ a *type $\text{type}(x)$* of $0$, $1$, or $2$. We store the node type in a *trit* (ternary bit).
Fibonacci Heap
(lazy, one-tree)

$r(x)$ = number of type 0 or 1 children

**type 0:** last link lost was fair, and since then no type 0 or 1 children have been lost (via a cut or conversion to type 2)

**type 1:** last link lost was fair, and since then one type 0 or 1 child has been lost

**type 2:** last link lost was unfair, or last link lost was fair and since then two or more type 0 or 1 children have been lost
Rules for updating node types

When a root loses a fair link, make it type 0.
When a root loses an unfair link, make it type 2.
When doing \textit{decrease-key}(x), if x is a child of type 0 or 1, start at \( p(x) \) and visit each ancestor, decreasing its rank and changing it from type 1 to type 2, until reaching a root or a child of type 0 or 2. Decrease the rank of this last node by 1, and if it is a child of type 0, change it to type 1.
Pseudocode to update types and ranks during *decrease-key*

\[ y \leftarrow p(x); \]

**while** \( p(y) \neq \text{null} \) **and** \( \text{type}(y) = 1 \) **do**

\[ \{ \text{type}(y) \leftarrow 2; \ r(y) \leftarrow r(y) - 1; \ y \leftarrow p(y) \}; \]

\[ r(y) \leftarrow r(y) - 1; \]

**if** \( p(y) \neq \text{null} \) **and** \( \text{type}(y) = 0 \) **then** \( \text{type}(y) \leftarrow 1 \)

In addition, the cut during the *decrease-key* creates a new root; the subsequent unfair link converts a root to a type 2 node.
Time bound

Let $r(n) =$ maximum rank of the root of an $n$-node tree. (Later we shall prove $r(n) = O(\lg n)$.)

Let the potential ($\Phi$) of a set of trees be

$\#	ext{roots} + \#\text{type 2 nodes} + 2 \times \#\text{type 1 nodes}$

Each fair link reduces $\Phi$ by 1: a root becomes type 0.

Each iteration of the while loop in decrease-key reduces $\Phi$ by 1: a type 1 node becomes type 2.
Increases in $\Phi$:

**insert**: $+1$ by creation of a new root.

**decrease-key**: $\leq 3$ by conversion of a type 0 node to type 1 and a type 0 node to a root (by a cut).

**delete-min**: $\leq (r(n) + 1)$ by conversion of at most $r(n) + 1$ type 0 nodes to roots (before links).

Thus total number of fair links and iterations of the **while** loop in **decrease-key** is $O(m + kr(n))$

$\rightarrow$ total time for $m$ operations including $k$ **delete-mins and deletes** is $O(m + kr(n))$. 
Fibonacci Numbers

\[ F_0 = 0, \ F_1 = 1, \ F_{i+2} = F_{i+1} + F_i \]

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ...

\[ F_{k+2} \geq \phi^k \] where \( \phi = (1 + \sqrt{5})/2 \) is the golden ratio
The Last Piece of the Puzzle: Rank Bound

**Theorem 3:** A subtree of \( n \) nodes has a root of rank at most \( \lg n / \lg \phi \), where \( \phi = (1 + \sqrt{5})/2 \) is the golden ratio.

**Proof:** Let \( x \) be a node of rank \( k \). Order the \( k \) type 0 and 1 children of \( x \) in the order they were linked to \( x \), earliest to latest. If \( y \) is the \( i^{th} \) such child, \( y \) has rank at least \( i - 2 \): when \( x \) and \( y \) were linked, \( x \) had rank at least \( i - 1 \), and so did \( y \); after the link, \( y \) can lose only one type 0 or 1 child, or it would become type 2.
Let $n_k$ be the minimum number of nodes in the subtree of a node of rank $k$. Then $n_k$ satisfies the following recurrence:

$$n_0 = 1, \quad n_1 = 2, \quad n_k \geq 2 + n_0 + n_1 + \ldots + n_{k-2}$$

This implies $n_k \geq n_{k-1} + n_{k-2}$ for $k \geq 3$, which also holds for $k = 2$. It follows that $n_k \geq F_{k+2}$, where $F_k$ is the $k^{th}$ Fibonacci number. Since $F_{k+2} \geq \phi^k$, $n_k \geq \phi^k$. 
**Theorem 4:** The amortized time per operation on a Fibonacci heap is $O(1)$ per `make-heap`, `find-min`, `insert`, `meld`, and `decrease-key`, and $O(\lg n)$ per `delete-min` and `delete`, where $n$ is the current number of items in the heap.
What we have learned

Implicit \textit{d-ary heaps} for \textit{d} constant are simple and support all the heap operations except \textit{meld} in $O(lgn)$ worst-case time. Such a heap is a good choice to implement Dijkstra’s shortest path algorithm. The current consensus is that $d = 4$ is best in practice.

\textbf{Fibonacci heaps} support all heap operations in $O(1)$ amortized time except \textit{delete-min} and \textit{delete}, which take $O(lgn)$ amortized time. Dijkstra’s algorithm implemented with such a heap runs in $O(m + nlgn)$ time, faster in theory than using a \textit{d-ary heap}. 