Priority queue

- Insert \( Q, x \)
- Retrieve \( x \) from \( Q \) s.t. \( x.\text{value} \) is min (or max)
- Sorted linked list:
  - \( O(n) \) to insert \( x \) into right place
  - \( O(1) \) access-min, \( O(1) \) delete-min

Binary heap

Complete – missing nodes only at the lowest level

Heap property – on any path the parent has higher priority than child

Typically: min-heaps

Priority queue
- insert \( (Q, x) \)
- pop \( Q \)

Preorder

Complete Binary Trees

Array Storage

- Fill the array following a breadth-first traversal:

\[
\text{left}(j) = 2j, \quad \text{right}(j) = 2j+1, \quad \text{parent}(j) = \lfloor j/2 \rfloor
\]

Heap/Priority queue

- Find min/Delete; Insert;
- Decrease key (change value of the key)
- Merge two heaps ...

Binomial heaps:

- Performance: All of the following operations work in \( O(\log n) \)
  time on a binomial heap with \( n \) elements:
    - Insert a new element to the heap
    - Find the element with minimum key
    - Delete the element with minimum key from the heap
    - Decrease key of a given element
    - Delete a given element from the heap
    - Merge two given heaps to one heap
    - Finding the element with minimum key can also be done in \( O(1) \) by using an additional pointer to the minimum.
Binomial heaps, Fibonacci heaps, and applications

Dan Feldman

Binomial trees

19.1 Binomial trees

The binomial tree $B_i$ is a rooted tree (see Section 9.1.2) defined recursively. As shown in Figure 19.2a, the binomial tree $B_0$ consists of a single node. The binomial tree $B_i$, consists of two binomial trees $B_{i-1}$ and $B_{i-2}$ that are deleted together: the root of one is the leftmost child of the root of the other. Figure 19.2b shows the binomial trees $B_i$ through $B_4$.

Figure 19.2a: The recursive definition of the binomial tree $B_i$. Triangles represent rooted subtrees.

Some links

- http://www.jucs.org/jucs_7_5/animation_for_teaching_purposes/Lauer_T.html
Lemma 20.1

- For the binomial tree $B_k$,
  1. there are $2^k$ nodes,
  2. the height of the tree is $k$,
  3. there are exactly $\binom{k}{i}$ nodes at depth $i$ for $i = 0, 1, \ldots, k$, and
  4. the root has degree $k$, which is greater than that of any other node; moreover if the children of the root are numbered from left to right by $k-1, k-2, \ldots, 0$, child $i$ is the root of a subtree $B_i$.

Properties of binomial trees

1) $|B_k| = 2^k$
2) $\text{degree}(\text{root}(B_k)) = k$
3) $\text{depth}(B_k) = k$

The degree and depth of a binomial tree with at most $n$ nodes is at most $\log(n)$.

Define the rank of $B_k$ to be $k$.

Binomial heaps (def)

A collection of binomial trees with at most one of every rank.
Items at the nodes, heap ordered.

Possible rep: Doubly link roots and children of every node. Parent pointers needed for delete.

Figure 20.4 The binomial tree $B_4$ with nodes labeled in binary by a postorder walk.

Binomial heaps

Binomial heaps (operations)

Operations are defined via a basic operation, called linking, of binomial trees:
Produce a $B_k$ from two $B_{k-1}$, keep heap order.

Binomial heaps (ops cont.)

Basic operation is $\text{meld}(h1,h2)$:
Like addition of binary numbers.

<table>
<thead>
<tr>
<th>$B_5$</th>
<th>$B_4$</th>
<th>$B_3$</th>
<th>$B_1$</th>
<th>$B_0$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$h1:$</td>
<td>$B_4$</td>
<td>$B_3$</td>
<td>$B_1$</td>
<td>$B_0$</td>
</tr>
<tr>
<td>$h2:$</td>
<td>$B_4$</td>
<td>$B_3$</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$+$</td>
<td>$B_5$</td>
<td>$B_4$</td>
<td>$B_3$</td>
<td>$B_2$</td>
</tr>
</tbody>
</table>

The execution of $\text{BINOMIAL-HEAP-UNION.}(a)$
Binomial heaps $H_1$ and $H_2$.

Delete min

- Find min (=1)
- Extract tree
- Split tree, reverse
- Merge/meld
Decrease key (y=26 => y=7)

Binomial heaps (ops cont.)

Findmin(h): obvious
Insert(x,h): meld a new heap with a single B, containing x, with h
delete(min(h)): Chop off the minimal root. Meld the subtrees with h. Update minimum pointer if needed.
delete(x,h): Bubble up and continue like delete-min
decrease-key(x,h,d): Bubble up, update min ptr if needed

All operations take \(O(\log n)\) time on the worst case, except
find-min(h) that takes \(O(1)\) time.

What is the time complexity?

# A.len = k

for i=1..n do Increment(A); # O(?)

Increment(A)
1. i=0
2. while i<A.len and A.i==1
3. A[i] = 0
4. i++
5. if i < A.len
6. A[i] = 1

Amortized analysis

We are interested in the worst case running time of a sequence of operations.

Example: binary counter

single operation – increment

Increment(A)
0.00000
1. 00001
2. 00010
3. 00011
4. 01000
5. 01001

Total cost

In the amortized running time analysis we pretend that very fast operations take a little bit longer than they actually do.

This additional time is then later subtracted from the actual running time of slow operations.

The amount of time saved for later use is measured at any given moment by a potential function.

Incrementing binary counter

<table>
<thead>
<tr>
<th>value</th>
<th>bits</th>
<th>Total cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>7</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>8</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>10</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>11</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>15</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>16</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>17</td>
</tr>
<tr>
<td>9</td>
<td>0</td>
<td>19</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>22</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>23</td>
</tr>
<tr>
<td>12</td>
<td>0</td>
<td>25</td>
</tr>
<tr>
<td>13</td>
<td>0</td>
<td>26</td>
</tr>
<tr>
<td>14</td>
<td>0</td>
<td>28</td>
</tr>
<tr>
<td>15</td>
<td>0</td>
<td>31</td>
</tr>
<tr>
<td>16</td>
<td>0</td>
<td>32</td>
</tr>
</tbody>
</table>
Amortized analysis (Cont.)

On the worst case increment takes \(O(k)\).

\(k = \#\text{digits}\)

What is the complexity of a sequence of increments (on the worst case)?

Define a potential of the counter:

\[ \Phi(c) = ? \]

Amortized(increment) = actual(increment) + \(\Delta\Phi\)

Amortized analysis (Cont.)

Amortized(increment) = actual(increment) + \(\Phi_{\text{new}} - \Phi_{\text{old}}\)

... + ...

Amortized(increment) = actual(increment) + \(\Phi_{\text{new}} - \Phi_{\text{old}}\)

\[\sum\text{Amortized(increment)} \leq \sum\text{actual(increment)}\]

if \(\Phi_{\text{new}} - \Phi_{\text{old}} \geq 0\)

Define a potential of the counter:

\[ \Phi(c) = \#(\text{ones}) \]

Amortized(increment) = actual(increment) + \(\Delta\Phi\)

Amortized(increment) = \(1 + \#(1 \Rightarrow 0) + 1 - \#(1 \Rightarrow 0) = O(1)\)

\[\Rightarrow \text{Sequence of } n \text{ increments takes } O(n) \text{ time}\]

Binomial heaps - amortized ana.

\[ \Phi(\text{collection of heaps}) = \#(\text{trees}) \]

Amortized cost of insert \(O(1)\)

Amortized cost of other operations still \(O(\log n)\)

Binomial heaps + lazy meld

Allow more than one tree of each rank.

Meld \((h1,h2)\):

- Concatenate the lists of binomial trees.
- Update the minimum pointer to be the smaller of the minimums

\(O(1)\) worst case and amortized.

Binomial heaps + lazy meld

As long as we do not do a delete-min our heaps are just doubly linked lists:

\[ \begin{array}{cccccc}
\text{5} & \text{6} & \text{9} & \text{11} & \text{4} & \text{6}
\end{array} \]

Delete-min: Chop off the minimum root, add its children to the list of trees.

Successive linking: Traverse the forest keep linking trees of the same rank, maintain a pointer to the minimum root.
Binomial heaps + lazy meld

Possible implementation of delete-min is using an array indexed by rank to keep at most one binomial tree of each rank that we already traversed.

Once we encounter a second tree of some rank we link them and keep linking until we do not have two trees of the same rank. We record the resulting tree in the array.

\[
\text{Amortized}(\text{delete-min}) =
\begin{align*}
&= (\#\text{links} + \text{max-rank}) - \#\text{links} \\
&= O(\log(n))
\end{align*}
\]

Fibonacci heaps (Fredman & Tarjan 84)

Want to do \textit{decrease-key}(x,h,d) faster than delete+insert.

Ideally in $O(1)$ time.

Why?

Dijkstra’s shortest path algorithm

Let $G = (V,E)$ be a weighted (weights are non-negative) undirected graph, let $s \in V$. Want to find the distance (length of the shortest path), $d(s,v)$ from $s$ to every other vertex.

Application #2 : Prim’s algorithm for MST

Start with $T$ a singleton vertex.

Grow a tree by repeating the following step:

Add the minimum cost edge connecting a vertex in $T$ to a vertex out of $T$.

Application #2 : Prim’s algorithm for MST

Maintain the vertices out of $T$ but adjacent to $T$ in a heap.

The key of a vertex $v$ is the weight of the lightest edge $(v,w)$ where $w$ is in the tree.

Iteration: Do a delete-min. Let $v$ be the minimum vertex and $(v,w)$ the lightest edge as above. Add $(v,w)$ to $T$. For each edge $(w,u)$ where $u \in T$,

- if $\text{key}(u) = \infty$ insert $u$ into the heap with key($u$) = $w(v,u)$
- if $w(w,u) < \text{key}(u)$ decrease the key of $u$ to be $w(w,u)$.

With regular heaps $O(m \log(n))$.

With F-heaps $O(n \log(n) + m)$.

Fibonacci heap

![Fibonacci heap diagram](image-url) 

Figure 20.1: (a) A Fibonacci heap consisting of five min-heap-ordered trees and 14 nodes. The dashed line indicates the root list. The minimum node of the heap is the node containing the key 0. The three marked nodes are blackened. The potential of this particular Fibonacci heap is $|E| - 3 + 11$. (b) A more complete representation showing pointers $p_i$ (up arrows), $n_i$ (to-children arrows), and left and right side-to-leaf arrows. These details are omitted in the remaining figures in this chapter, since all the information shown here can be determined from what appears in part (a).
Insert (left from root)

Finding the minimum node

- The minimum node of a Fibonacci heap $H$ is given by the pointer $\text{min}[H]$, so we can find the minimum node in $O(1)$ actual time. Because the potential of $H$ does not change, the amortized cost of this operation is equal to its $O(1)$ actual cost.

Fibonacci heaps (cont.)

Decrease-key $(x, h, d)$: indeed cuts the subtree rooted by $x$ if necessary as we showed.

In addition we maintain a mark bit for every node. When we cut the subtree rooted by $x$ we check the mark bit of $p(x)$. If it is set then we cut $p(x)$ too. We continue this way until either we reach an unmarked node in which case we mark it, or we reach the root.

This mechanism is called cascading cuts.
Suggested implementation for decrease-key(x,h,d):
If x with its new key is smaller than its parent, cut the subtree rooted at x and add it to the forest. Update the minimum pointer if necessary.

Two calls of FIB-HEAP-DECREASE-KEY.

Decrease-key (cont.)

Does it work?

Obs1: Trees need not be binomial trees any more...
Do we need the trees to be binomial?
Where have we used it?
In the analysis of delete-min we used the fact that at most log(n) new trees are added to the forest. This was obvious since trees were binomial and contained at most n nodes.

Decrease-key (cont.)

Such trees are now legitimate.
So our analysis breaks down.

Fibonacci heaps (cont.)

We shall allow non-binomial trees, but will keep the degrees logarithmic in the number of nodes.

Rank of a tree = degree of the root.

Delete-min: do successive linking of trees of the same rank and update the minimum pointer as before.
Insert and meld also work as before.
Fibonacci heaps (delete)

Delete(x,h) : Cut the subtree rooted at x and then proceed with cascading cuts as for decrease key.
Chop off x from being the root of its subtree and add the subtrees rooted by its children to the forest
If x is the minimum node do successive linking

Fibonacci heaps (analysis)

Want everything to be O(1) time except for delete and delete-min.
*** cascading cuts should pay for themselves
Φ (collection of heaps) = #(trees) + 2#(marked nodes)
Actual(decrease-key) = O(1) + #(cascading cuts)
ΔΦ(decrease-key) = O(1) - #(cascading cuts)
*** amortized(decrease-key) = O(1)!

Fibonacci heaps (analysis)

Lemma 1 : Let x be any node in an F-heap. Arrange the children of x in the order they were linked to x, from earliest to latest. Then the i-th child of x has rank at least i-2.

Proof:
When the i-th node was linked it must have had at least i-1 children.
Since then it could have lost at most one.

Fibonacci heaps (analysis)

Corollary 1 : A node x of rank k in a F-heap has at least φ^k descendants, where φ = (1 + √5)/2 is the golden ratio.

Proof:
Let s_k be the minimum number of descendants of a node of rank k in a F-heap.
By Lemma 1 s_k ≥ φ^{i-2} + 2

The potential of a Fibonacci heap is given by
Potential = t + 2m where t is the number of trees in the Fibonacci heap, and m is the number of marked nodes. A node is marked if at least one of its children was cut since this node was made a child of another node (all roots are unmarked).
Fibonacci heaps (analysis)

Proof (cont): Fibonacci numbers satisfy
\[ F_{k+2} = \sum_{i=2}^{k} F_i + 2, \] for \( k \geq 2 \), and \( F_2 = 1 \)
so by induction \( s_k \geq F_{k+2} \)
It is well known that \( F_{k+2} \geq 2^k \)
It follows that the maximum degree \( k \) in a F-heap with \( n \) nodes is such that \( 2^k \leq n \)
so \( k \leq \log(n) / \log(2) = 1.4404 \log(n) \)

Make-Fibonacci-Heap()
\[ n[H] := 0 \]
\[ min[H] := NIL \]
return \( H \)

Fibonacci-Heap-Minimum(H)
return \( min[H] \)

CONSOLIDATE(H)
for \( i = 0 \) to \( D(n[H]) \)
do \( A[i] := NIL \)
for each node \( w \) in the root list of \( H \)
do \( x := w \)
\( d := \text{degree}[x] \)
while \( d[i] := NIL \)
do \( y := A[i] \)
if \( \text{key}[y] > \text{key}[x] \)
then exchange \( x \leftarrow y \)
Fibonacci-Heap-Link(H, y, x)
\( d[i] := d[i]+1 \)
\( A[i] := x \)
\( n[H] := n[H] + n[H] \)
do if \( A[i] := NIL \)
then add \( A[i] \) to the root list of \( H \)
if \( min[H] := NIL \) or \( \text{key}[A[i]] > \text{key}[min[H]] \)
then \( min[H] := A[i] \)

Fibonacci-Heap-Union(H1,H2)
\( H := \text{Make-Fibonacci-Heap()} \)
\( min[H] := min[H1] \)
Concatenate the root list of \( H2 \) with the root list of \( H \)
if \( \text{(min[H1] = NIL) or (min[H2] := NIL and min[H2] < min[H1])} \)
then \( min[H] := min[H2] \)
\( n[H] := n[H1] + n[H2] \)
free the objects \( H1 \) and \( H2 \)
return \( H \)
Fibonacci-Heap-Insert($H, x$)

- $\text{degree}[x] := 0$
- $p[x] := \text{NIL}$
- $\text{child}[x] := \text{NIL}$
- $\text{left}[x] := x$
- $\text{right}[x] := x$
- $\text{mark}[x] := \text{FALSE}$

- Concatenate the root list containing $x$ with root list $H$
- If $\text{min}[H] = \text{NIL}$ or $\text{key}[x] < \text{key}[\text{min}[H]]$
  - Then $\text{min}[H] := x$
- $n[H] := n[H] + 1$

Fibonacci-Heap-Extract-Min($H$)

- $z := \text{min}[H]$
- If $x \neq \text{NIL}$
  - Then for each child $x$ of $z$
    - Do add $x$ to the root list of $H$
      - $p[x] := \text{NIL}$
      - Remove $z$ from the root list of $H$
      - If $z = \text{right}[z]$
        - Then $\text{min}[H] := \text{NIL}$
      - Else $\text{min}[H] := \text{right}[z]$
    - CONSOLIDATE($H$)
  - $n[H] := n[H] - 1$
- Return $z$

Fibonacci-Heap-Decrease-Key($H, x, k$)

- If $k > \text{key}[x]$
  - Then error "new key is greater than current key"
- $\text{key}[x] := k$
- $y := p[x]$
- If $y \neq \text{NIL}$ and $\text{key}[x] < \text{key}[y]$
  - Then CUT($H, x, y$)
- CASCADING-CUT($H, y$)
- If $\text{key}[x] < \text{key}[\text{min}[H]]$
  - Then $\text{min}[H] := x$

CUT($H, x, y$)

- Remove $x$ from the child list of $y$, decrementing degree[y]
- Add $x$ to the root list of $H$
- $p[x] := \text{NIL}$
- $\text{mark}[x] := \text{FALSE}$
- If $z \neq \text{NIL}$
  - Then if $\text{mark}[y] = \text{FALSE}$
  - Then $\text{mark}[y] := \text{TRUE}$
  - Else CUT($H, y, z$)
  - CASCADING-CUT($H, z$)

Fibonacci-Heap-Delete($H, x$)

- Fibonacci-Heap-Decrease-Key($H, x, -\infty$)
- Fibonacci-Heap-Extract-Min($H$)

### Summary of Running Times

<table>
<thead>
<tr>
<th>Operation</th>
<th>Linked List</th>
<th>Balanced Binary Tree</th>
<th>Min-Heap</th>
<th>Fibonacci Heap</th>
<th>B-Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>insert</td>
<td>$O(1)$</td>
<td>$O(\log n)$</td>
<td>$O(\log n)$</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>accessmin</td>
<td>$O(n)$</td>
<td>$O(\log n)$ or $O(1)$</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>decreasekey</td>
<td>$O(1)$</td>
<td>$O(\log n)$</td>
<td>$O(\log n)$</td>
<td>$O(\log n)$*</td>
<td>$O(\log n)$*</td>
</tr>
<tr>
<td>delete</td>
<td>$O(n)$</td>
<td>$O(\log n)$</td>
<td>$O(\log n)$</td>
<td>$O(\log n)$*</td>
<td>$O(\log n)$*</td>
</tr>
<tr>
<td>merge</td>
<td>$O(1)$</td>
<td>$O(m \log(n/m))$</td>
<td>$O(m \log(n/m))$</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
</tbody>
</table>

*Amortized time  
**With minor modification to the conventional Binary tree
van Emde Boas tree

A van Emde Boas tree (or van Emde Boas priority queue), also known as a VEB tree, is a tree data structure which implements an associative array with m-bit integer keys. It performs all operations in $O(\log m)$ time. Notice that $m$ is the size of the keys — therefore $O(\log m)$ is $O(\log \log n)$ in a full tree, exponentially better than a self-balancing binary search tree. They also have good space efficiency when they contain a large number of elements, as discussed below. They were invented by a team led by Peter van Emde Boas in 1977.

Wikipedia, Mar 20, 2010

PQ – Structure (Cont’d)

- Subset $S \subseteq \{1, \ldots, n\}$ Representation
  - Mark leaves in $S$ and all nodes on the paths from root to the marked leaves.

PQ – (Cont’d)
Sketches of Algorithms

- Insert($i$)
- Delete($i$)
- Member($i$)
- Min
- Predecessor
PQ – (Cont’d)
Sketches of Algorithms

• Insert(i)
• Delete(i)
• Member(i)
• Min
• Predecessor

PQ – (Cont’d)
Sketches of Algorithms

• Insert(i)
• Delete(i)
• Member(i)
• Min
• Predecessor

PQ – (Cont’d)
Sketches of Algorithms

• Insert(i)
• Delete(i)
• Member(i)
• Min
• Predecessor

PQ – (Cont’d)
Sketches of Algorithms

• Insert(i)
• Delete(i)
• Member(i)
• Min
• Predecessor