Part I. Data Structures

- because every algorithm must store and access/search data

Assume you know:

- priority queue (heap)
- dictionary (hashing, balanced binary search trees)

In this course

- more advanced structures/analysis
  in particular amortized analysis

Priority Queue

Store n elements, each with integer key

Operations:

- Insert, DeleteMin, DecreaseKey, Build
  (and Delete)
Heap
- binary tree of elements
- structure

```
      a
     /|
    / |
   /  |
  b   c
```

Thus min key is at root

Implementing priority queue operations on a heap
- Insert - insert at bottom, bubble up $\Theta(\log n)$
- DeleteMin - remove root element, put last item there, bubble down $\Theta(\log n)$
  (Deleting other element - same idea)
- DecreaseKey - bubble up $\Theta(\log n)$
- BuildHeap - can be done faster than repeated insertion ($\Theta(n \log n)$)
- Merge - merge two heaps into one $\Theta(n)$ (useful in some applications)
- Destroy 2 heaps and build new one

Almost perfect
So can store in an array in level order and use indexing instead of pointers
Height is $\Theta(\log n)$
Application

Prim's algorithm for **Minimum Spanning Tree**: (recall that we used this last day for TSP)

Given an (undirected) graph with weights on edges, find min. weight spanning tree

- **Prim's Algorithm** (from CS 341)
  - Grow MST greedily starting from some node $s$
  - General step

$$\begin{align*}
U & \quad 3 \\
V - U & \quad 1 \\
S & \quad 2 \\
\end{align*}$$

Add to tree $T$ the cheapest edge from $U$ to $V - U$

$n =$ # vertices $m =$ # edges

Using heap of edges from $U$ to $V - U$

- $n$ DeleteMin + $m$ (Insert & Delete)
- Heap has $\leq m$ elements
- Time: $n \log m + m \log m = O(n \log m)$

Using heap of vertices in $V - U$

- Key of vertex $v =$ min weight of edge from $U$ to $v$

- Heap has $\leq n$ elements
- $n$ DeleteMin + $m$ DecreaseKey
- $n \log n + m \log n = O((n+m) \log n)$

This is only better by constant since $m = O(n^2)$ so $\log m$ is $O(\log n)$

But we will see a way to improve DecreaseKey.
Binomial Heaps

- Improve merge (not yet improvement for Prim)
- Use pointers to implement trees, not an array

This allows

- Keep heap order but go beyond binary trees
  Also relax shape and allow multiple trees.

Binomial Tree Definition

\[ B_0 \quad B_1 \quad B_2 \]

\[ B_k = B_{k-1} \cup B_{k-1} \]

\[ k = \text{rank of } B_k = \text{degree of root} \]

Claims

- \( B_k \) has \( 2^k \) nodes
  \( (2^{k-1} + 2^{k-1} = 2^k) \)
- Height (\( B_k \)) = \( k \)
  \( (k-1 + 1) \)
- There are \( \binom{k}{i} \) nodes at depth \( i \) in \( B_k \)
  \( \binom{k-1}{i} + \binom{k-1}{i-1} = \binom{k}{i} \)

Hence the name

\[ \text{all proved by induction} \]
Binomial Heap Definition - collection of binary trees, at most one of each rank, each one heap ordered.

e.g. for \( n = 13 \) use \( B_3 \quad B_2 \quad B_0 \)

\[
2^3 = 8 \quad 2^2 = 4 \quad 2^0 = 1
\]

\[
13 = 1 \quad 1 \quad 0 \quad 1 \quad \text{base 2}
\]

Expressing \( n \) in binary tells us which trees to use.

\# trees \( \leq \lfloor \log n \rfloor + 1 \) since this is \# binary bits in \( n \)

Height of each tree is \( \leq \lfloor \log n \rfloor \)

Merging two binomial heaps (all other operations will be implemented in terms of this).

Like binary addition.

For \( i = 0, 1, \ldots \) if there are two \( B_i \)'s, link to form a \( B_{i+1} \).
Example of Merging Binomial Heaps

\[ n_1 = 6 \]
\[ n_2 = 3 \]

\[
\begin{array}{c}
i = 2 \\
8 \\
10 \\
11 \\
\end{array}
\quad
\begin{array}{c}
i = 1 \\
4 \\
12 \\
6 \\
\end{array}
\quad
\begin{array}{c}
i = 0 \\
2 \\
5 \\
10 \\
\end{array}
\]

\[ 110 \]
\[ + 11 \]
\[ 1001 \]

\[ \text{final binomial heap} \]

Note: in case \[ 1011 \] two \( B_0 \)'s merge to a \( B_1 \), then we have 3 \( B_1 \)'s, we merge two of them and leave one alone.
**Analysis of Binomial Heap Operations** (worst case)

- **Merge** $\Theta(n \log n)$ time (joining $B_i$ and $B_j$ takes $\Theta(1)$, so the cost is exactly the same as the bit cost of binary addition)

- **Insert** $\Theta(n \log n)$ - merge binomial heap with singleton (so it's like adding 1 to binary counter)

- **DeleteMin** $\Theta(n \log n)$
  - Look through roots (at most $\Theta(n \log n)$) to find the min.
  - Yank it out of its tree, leaving $\Theta(n \log n)$ binomial trees representing $2^i - 1$, i.e. $B_{i-1}, B_{i-2}, \ldots, B_0$.
  - Merge this with the rest of the binomial heap.

$\Theta(n \log n)$

- **Decrease Key** $\Theta(n \log n)$ - bubble up as in standard heap.

---

**Creating a binomial heap of $n$ items**

- Repeated insertion $\Theta(n \log n)$ worst case
  - But we'll see that in fact it's $\Theta(n)$.

Idea: the worst case cost of insertion, $\Theta(n \log n)$, does not happen too often.

Cost of repeated insertion = cost of incrementing binary counter