Shortest Path

Shortest Paths

- $G = (V, E, \ell)$, $n = |V|$, $m = |E|$, $\ell: E \to \mathbb{R}$.
- **APSP** (*All Pairs Shortest Paths*)
  - $O(n^3 \log n)$ — Path Doubling Algorithm
  - $O(n^3)$ — Floyd-Warshall algorithm
  - $O(mn + n^2 \log n)$ — $n \times$ Dijkstra
  - $O(mn + n^2 \log\log n)$ — [Pettie 2002]
- **SSSP** (*Single Source Shortest Paths*)
  - $O(mn)$ — Bellman-Ford algorithm
  - $O(m\sqrt{n} \log C)$ — [Goldberg 1995] $\ell: E \to \{-C, \ldots, C\}$.
  - $O(m \log^8 n \log C)$ — [Bernstein, Nanongkai, Wulff-Nilsen 2022] Best Paper Award, FOCS 2022
- **SSSP$^+$** ($\ell: E \to \mathbb{R}^+ \cup \{0\}$)
  - $O(m + n \log n)$ — Dijkstra's algorithm + Fibonacci Heaps

SSSP+

There is a monotonicity property here. So,

SSSP$^+$

- Given directed graph $G = (V, E)$ with $\ell: E \to \mathbb{R}^+ \cup \{0\}$ and a source $s \in V$, compute $\text{dist}(s, v)$ for every $v \in V$.
  - Key observation: list vertices in increasing distance from the source: $\{s = v_0, v_1, v_2, \ldots, v_{n-1}\}$. The shortest v_0-v_i path only uses intermediate vertices from $\{v_1, \ldots, v_{i-1}\}$.

→ define for $S \subseteq V$ $\quad d: V \to \mathbb{R}^+ \cup \{0\}$ tentative dist func

→ $v \in S$, $d(v) = \text{dist}(s, v)$

→ $v \notin S$, $d(v) = \min_{u \in S} \{\text{dist}(s,u) + \ell(u,v)\}$ → 1 hop from S

→ if $v^* = \underset{v \notin S}{\text{argmin}} \{d(v)\}$ then $d(v^*) = \text{dist}(s, v^*)$

Pf By contra, let us say that $u^* \in S$ so u^* is the companion to v^*.

Suppose the above isn't the shortest path to v^* from s.

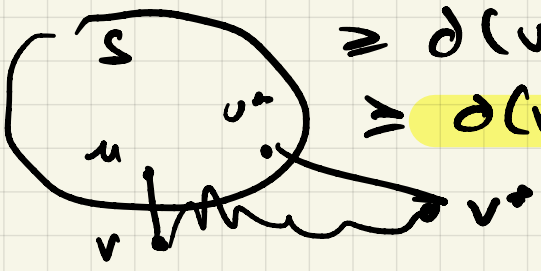
for the optimal path, say it exist s at u to v.

$$\text{dist}(s, v^*) = \text{dist}(s, u) + \ell(u, v) + \text{dist}(v, v^*)$$

$$= d(u) + \ell(u, v) + \text{dist}(v, v^*)$$

$$\geq d(v^*) + \text{dist}(v, v^*) \quad \text{→ positive}$$

$$\geq d(v^*)$$



oops

Djikstra

Dijkstra's shortest path algorithm

- Dijkstra's algorithm:
  - LOOP INVARIANT:
    - If $v \in S$ then $d(v) = \text{dist}(s, v)$
    - If $v \notin S$ then $d(v) = \text{dist}_S(s, v)$
  - Initialization: $S := \emptyset$; $d(s) := 0$; $d(v) := \infty$ for all $v \neq s$.
  - Repeat until all vertices are in S: ← **Insert** all vertices in a priority queue, keyed by d( )
    - Find the $v \notin S$ with minimum $d(v)$ ← **Deletemin** from priority queue
    - $S := S \cup \{v\}$
    - "Relax" each outgoing edge $(v, u) \in E$
      - $d(u) := \min\{d(u), d(v) + \ell(v, u)\}$ ← Perform ***decreasekey*** on u.
  - Return d

↳ a new edge is internal to S

Priority Queue Operations

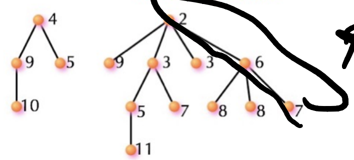
- Maintain a set Q (initially empty)
- Each element $x \in Q$ has a key $d(x)$
  - Insert(x,k) : $Q := Q \cup \{x\}$; $d(x) := k$ — $O(1)$ amortized
  - Deletemin() : Find $x \in Q$ minimizing $d(x)$; $Q := Q \setminus \{x\}$; return x; — $O(\log n)$
  - Decreasekey(x,k) : $d(x) := \min\{d(x), k\}$ — $O(1)$ amortized
- Binary heaps do all operations in $O(\log n)$ time
- ***Fibonacci heaps*** are optimal in an *amortized* sense.

Heap ordered tree

Heap-ordered trees

- Represent Q by a rooted, heap ordered tree (or forest of trees)
- One element per node
- *Heap ordered* means $d(x) \geq d(\text{parent}(x))$

⇒ element with min key at a root



↗ in value

→ you can link 2 trees in constant time. make larger key child of smaller

Binomial Heap

↳ Made of binomial trees. B_i binomial of rank i

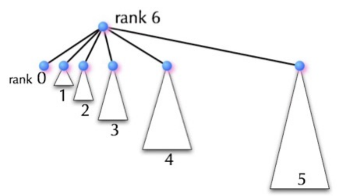
→ B_0 → one node

B_{i+1}

link two B_i

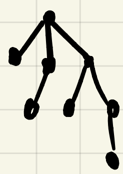
Binomial Trees

- Equivalent definition:
  - B_{i+1} = one root node with children $B_0, B_1, \ldots, B_i$



- Size of binomial trees: $|B_0| = 1$, $|B_i| = 2|B_{i-1}|$ ⇒ $|B_i| = 2^i$

→ Consider tree of rank 3



1
3
3
1

→ A binomial Heap structure

Binomial *Heaps*

- Structure of a binomial heap:
  - (1) Collection of heap-ordered binomial trees
  - (2) At most one tree of each rank
- Insert(X,k) :
  - Create a new node X (a B_0 tree)
  - Add X to the collection (1) and restore property (2)
- Deletemin() : → can have mult roots
  - Identify and delete root node Y with minimum key
  - Add children of Y to the collection (1) and restore prop. (2)

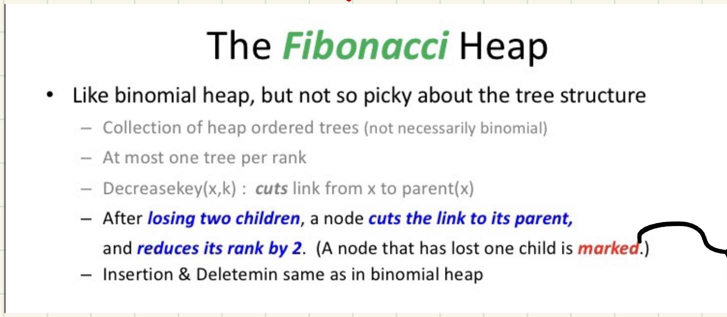
$\log_2 n$

Adding an elt ≅ adding 1 in binary

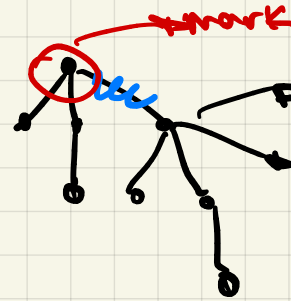
Analysis of Binomial Heaps

- n = # elements in the heap
- Claim: Insert, Deletemin take $O(\log n)$ time *in the worst case*. Why?
- Running time of insert:
  $1 + (\#\text{linkings}) \leq 1 +$ maximum rank
- Running time of deletemin:
  $1 + (\text{rank of element}) + (\#\text{linkings}) \leq 2(\text{max. rank}) + 1$

Fibonacci Heap



→ rule! can only be marked once. Else cut earlier and work up

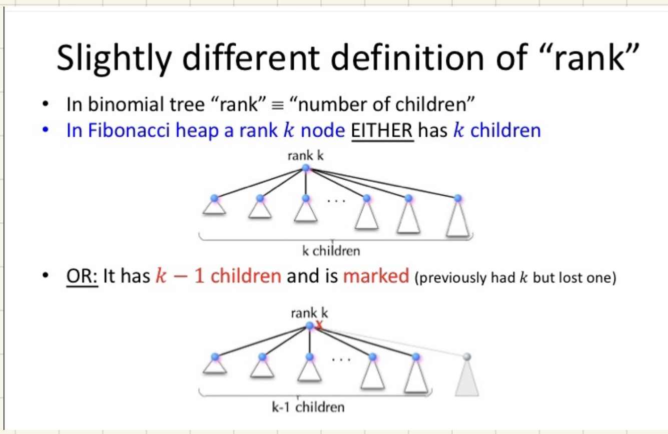


→ now a new root

decrease this!

cut to parent and unmark

↳ if a root gets double marked then remove mark and shift to rank n-2



Maximal Rank of Fib

Let T_i = minimum size of a rank i tree in Fib heap

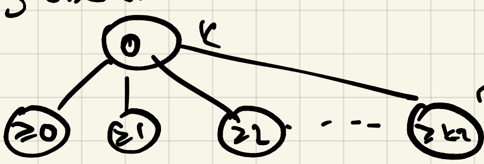
→ Claim: the ith child of any node has rank $\geq i-1$

↳ cutting just makes this weaker! ✓

→ adding involves merging and it works! ✓

↳ on right

So, observe



→ might not hold ($\geq k-1$) if marked

↳ removing ($\geq k-1$) biggest dent.

T_k = min size of rank k tree

$= 1 + T_0 + \cdots + T_{k-3} + T_{k-2}$

$= T_{k-1} + T_{k-2}$ → Fibonacci

$T_0 = 1$, $T_1 = 1$ $\Rightarrow T_k \geq \gamma^{k-1}$

γ → golden ratio

↳ 1.618

Cor max rank is $(\log_\gamma n) + 1$

$= 1.44 \log_2 n + 1$

↳ no of trees

Nice

Amortized Analysis

Amortized Analysis w/ Potential Functions

- t_i = actual time for the i^{th} operation.
- Φ_i = "potential" after i^{th} operation.
- $a_i = t_i + (\Phi_i - \Phi_{i-1})$ = amortized time for i^{th} oper.

actual time; net increase in potential

- Why it's useful:

$$\sum_{1 \le i \le n} a_i = \sum_{1 \le i \le n} (t_i + \Phi_i - \Phi_{i-1})$$
$$= \sum_{1 \le i \le n} t_i + (\Phi_n - \Phi_0)$$

Total amortized time = total actual time + net increase in potential

... and if $\Phi_0 = 0$, $\Phi_i \ge 0$ for all i ...

$$\ge \sum_{1 \le i \le n} t_i$$

Total amortized time ≥ total actual time.

work doing extra work to decrease potential

Need a good p

- Decreasekeys: variable number of cu
  number of trees but reduces the nu
- Potential function should depend or

• insert : va
linkings
ob

The Upshot:

- Fibonacci heaps perform n inserts, n' deletemins, m dec.keys in $O(m + n + n' \log n)$ time
- In Dijkstra's shortest path algorithm
  - $n = n'$ = number of vertices
  - m = number of edges
  - Running time is $O(m + n \log n)$. (*Optimal* linear time if $m > n \log n$.)

l Analysis

2)
at least 1 cut

Amortized Analysis

- Φ_i = (# trees) + 2(# marked nodes) after i operations
- insert
  - Actual time = t_i = 1 + linking
  - $\Phi_i - \Phi_{i-1}$ = 1 − linking
  - $a_i = t_i + \Phi_i - \Phi_{i-1} = 2$
- decrease
  - Actual time = t_i = #cuts + # linking ≥ 1
  - $\Phi_i - \Phi_{i-1}$ = (#cuts − # linking) − 2(#cuts − 2)
    Δ tree Δ marked
  - $a_i = t_i + \Phi_i - \Phi_{i-1} = 4$

- delete min
  - actual time t_i = 1 + # chil + # linking
  - $\Phi_i - \Phi_{i-1}$ = (# chil − 1) − # link
    Δ tree
    (Δ marked might be −2)
  - a_i = 2 (# children)
    ≤ $2 \log_{\phi} n + 1$
    ≈ $2.88 \log_2 n$

2 next time