CS 261: Graduate Data Structures

Week 5: Priority queues

David Eppstein
University of California, Irvine

Spring Quarter, 2021

Priority queues

Example: Dijkstra's algorithm

To find the shortest path distance from a starting vertex s to each other vertex t in a directed graph with positive edge lengths:

1. Initialize dictionary D of distances; $D[s] = 0$ and, for all other vertices t, $D[t] = +\infty$
2. Initialize collection Q of not-yet-processed vertices (initially all vertices)
3. While Q is non-empty:
   - Find the vertex $v \in Q$ with the smallest value of $D[v]$
   - Remove v from Q
   - For each edge $v \to w$, set $D[w] = \min\big(D[w], D[v] + \text{length}(v \to w)\big)$

Same algorithm can find paths themselves (not just distances) by storing, for each w, its predecessor on its shortest path: the vertex v that gave the minimum value in the calculation of $D[w]$

Priority queue operations used by Dijkstra, I

For Dijkstra to be efficient, we need to organize Q into a *priority queue* data structure allowing vertex with minimum D to be found quickly.

First operation used by Dijkstra: Create a new priority queue

1. Initialize dictionary D of distances; $D[s] = 0$ and, for all other vertices t, $D[t] = +\infty$
2. Initialize collection Q of not-yet-processed vertices (initially all vertices)
3. While Q is non-empty:
   - Find the vertex $v \in Q$ with the smallest value of $D[v]$
   - Remove v from Q
   - For each edge $v \to w$, set $D[w] = \min\big(D[w], D[v] + \text{length}(v \to w)\big)$

Priority queue operations used by Dijkstra, II

Second operation used by Dijkstra:
Find and remove item with minimum value

(Some applications use maximum value; some descriptions of this data structure separate find and remove into two operations.)

1. Initialize dictionary D of distances; $D[s] = 0$ and, for all other vertices t, $D[t] = +\infty$
2. Initialize collection Q of not-yet-processed vertices (initially all vertices)
3. While Q is non-empty:
   - Find the vertex $v \in Q$ with the smallest value of $D[v]$
   - Remove v from Q
   - For each edge $v \to w$, set $D[w] = \min\big(D[w], D[v] + \text{length}(v \to w)\big)$

Priority queue operations used by Dijkstra, III

Third operation used by Dijkstra: Change item's priority

In Dijkstra the priority always gets smaller (earlier in the priority ordering); this will be important for efficiency.

1. Initialize dictionary D of distances; $D[s] = 0$ and, for all other vertices t, $D[t] = +\infty$
2. Initialize collection Q of not-yet-processed vertices (initially all vertices)
3. While Q is non-empty:
   - Find the vertex $v \in Q$ with the smallest value of $D[v]$
   - Remove v from Q
   - For each edge $v \to w$, set $D[w] = \min\big(D[w], D[v] + \text{length}(v \to w)\big)$

Summary of priority queue operations

Used by Dijkstra:

- Create new priority queue for elements with priorities
- Find and remove minimum-priority element
- Decrease the priority of an element

Also sometimes useful:

- Add an element
- Remove arbitrary element
- Increase priority
- Merge two queues

Priority queues in Python

Main built-in type: heapq

Implements binary heap (later this week), represented as dynamic array

Operations:

- `heapify`: Create new priority queue for collection of elements (no separate priorities: ordered by standard comparisons)
- `heappop`: Find and remove minimum priority element
- `heappush`: Add an element

No ability to associate numeric priorities to arbitrary vertex objects.
No ability to change priorities of elements already in queue.

Dijkstra in Python

Cannot use decrease-priority; instead, store bigger priority queue of triples (d, u, v) where $u \to v$ is an edge and d is the length of the shortest path to u plus the length of the edge

First triple (d, u, v) found for v is the one giving the shortest path

```
def dijkstra(G,s,length):
    D = {}
    Q = [(0,None,s)]
    while Q:
        dist,u,v = heapq.heappop(Q)
        if v not in D:
            D[v] = dist
            for w in G[v]:
                heapq.heappush(Q,(D[v]+length(v,w),v,w))
    return D
```

Binary heaps

Binary heaps

Binary heaps are a type of priority queue

(Min-heap: smaller priorities are earlier; max-heap: opposite)

Based on *heap-ordered trees* (min is at root) and an efficient representation of binary trees by arrays

(No need for separate tree nodes and pointers

Allows priorities to change

(But to do this you need a separate data structure to keep track of the location of each object in the array – this extra complication is why Python doesn't implement this operation)

Find-and-remove, add, change-priority all take $O(\log n)$ time
Creating a binary heap from n objects takes $O(n)$ time

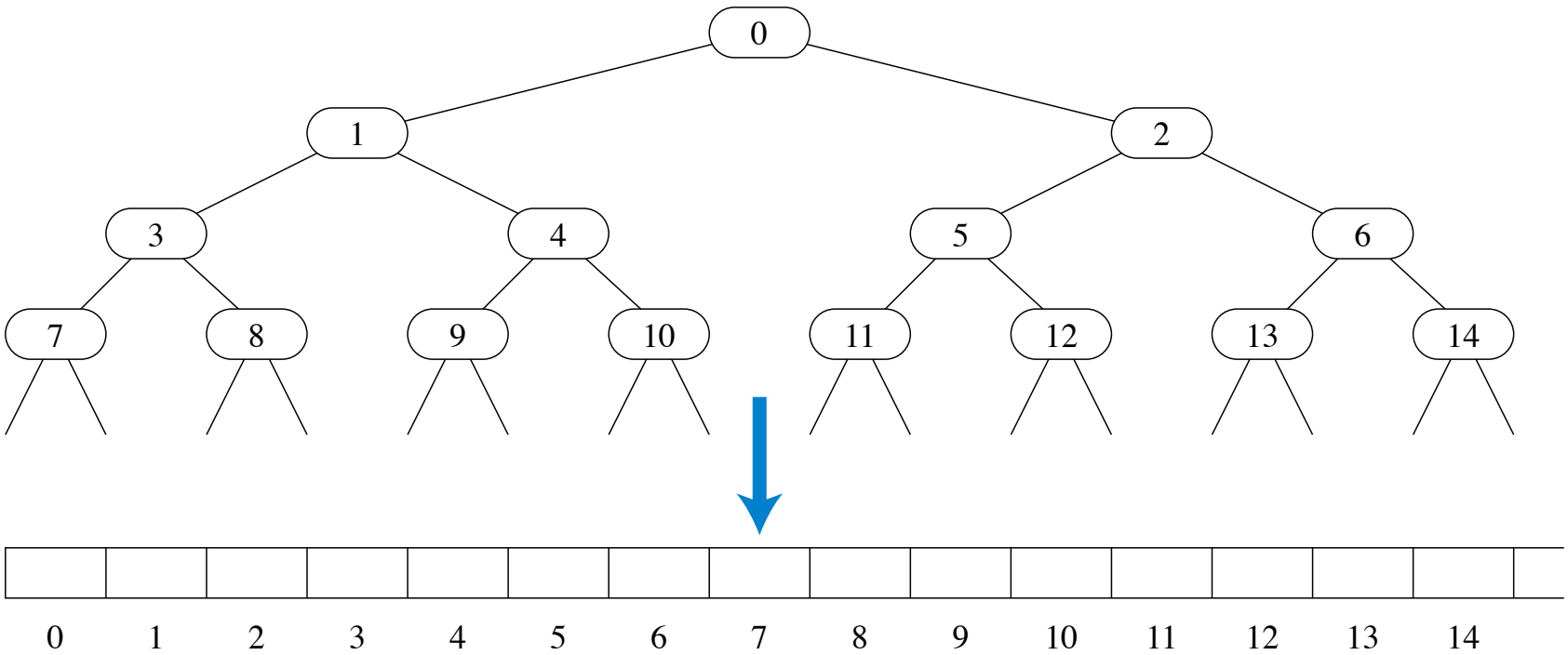
This is what Python `heapq` uses

Representing trees using arrays

Key idea: number the nodes of infinite complete binary tree row-by-row, and left-to-right within each row

Use node number as index for array cell representing that node

For finite trees with n nodes, use the first n indexes



$\text{parent}(x) = \lfloor (x-1)/2 \rfloor$ $\quad\quad$ $\text{children}(x) = (2x+1, 2x+2)$

Heap order

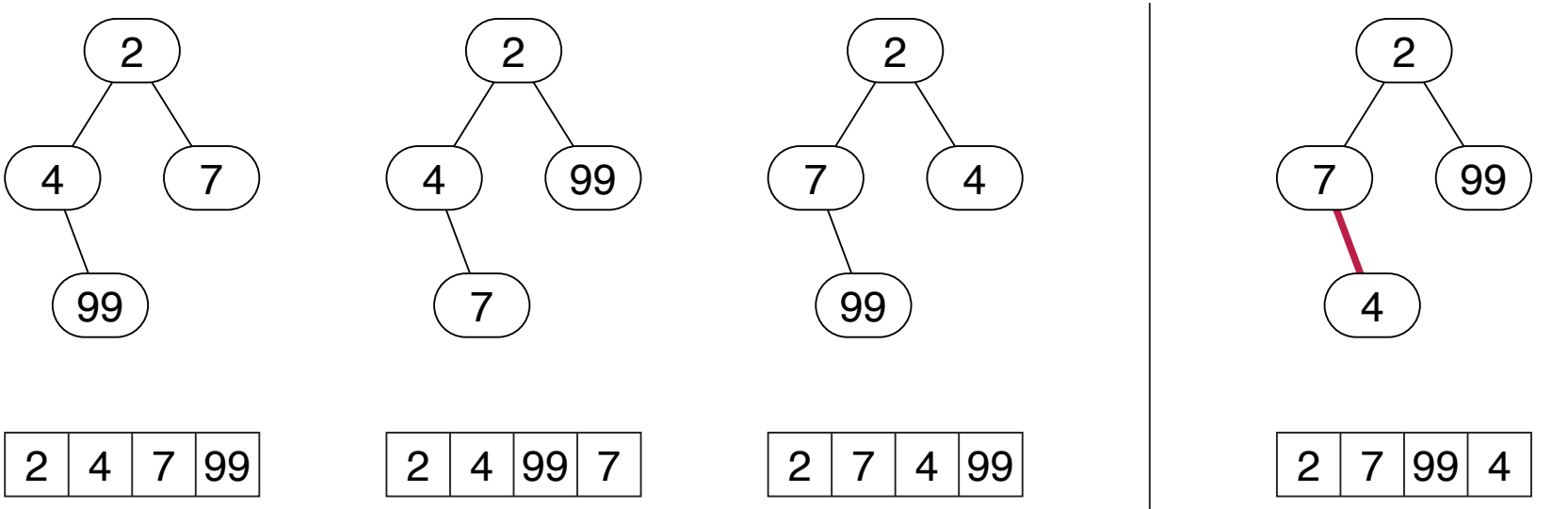
A tree is *heap-ordered* when, for every non-root node x,

$$\text{priority}\big(\text{parent}(x)\big) \leq \text{priority}(x)$$

A *binary heap* on n items is an array H of the items, so that the corresponding binary tree is heap-ordered: For all $0 < i < n$,

$$\text{priority}\left(H\left[\left\lfloor \tfrac{i-1}{2} \right\rfloor\right]\right) \leq \text{priority}\big(H[i]\big)$$

Examples of heap order



Sorted arrays are heap-ordered (left example)
but heap-ordered arrays might not be sorted (middle two examples)

Find and remove minimum-priority element

Where to find it: always in array cell 0

How to remove it:

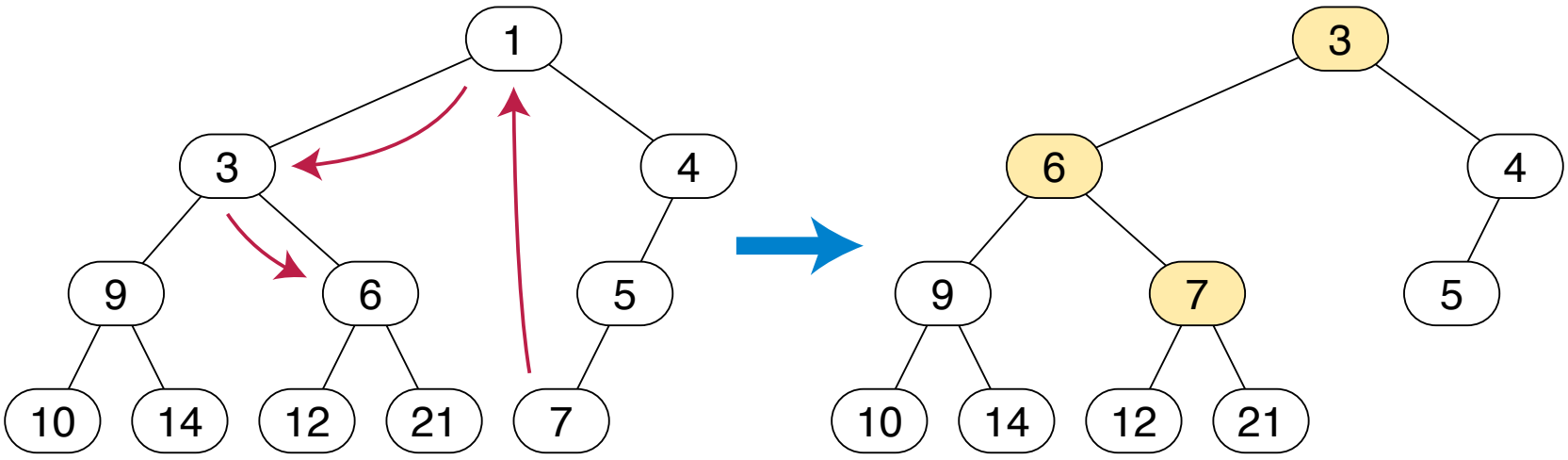
1. Swap element zero with element $n - 1$
2. Reduce array length by one
3. Fix heap at position 0 (might now be larger than children)

To fix heap at position p, repeat:

1. If p has no children ($2p + 1 \geq n$), return
2. Find child q with minimum priority
3. If priority(p) $\leq$ priority(q), return
4. Swap elements at positions p and q
5. Set $p = q$ and continue repeating

Time = $O(1)$ per level of the tree = $O(\log n)$ total

Example of delete-min operation



To delete the minimum element (element 1):

Swap the final element (7) into root position and shrink array

Swap element 7 with its smallest child (3) to fix heap order

Swap element 7 with its smallest child (6) to fix heap order

Both children of 7 are larger, stop swapping

Changing element priorities

First, find the position p of the element you want to change

(Simplest: keep a separate dictionary mapping elements to positions, update it whenever we change positions of elements)

To increase priority: fix heap at position p, same as for find-and-remove

To decrease priority, repeat:

1. If $p = 0$ or parent of p has smaller priority, return
2. Swap elements at positions p and parent
3. Set $p =$ parent and continue repeating

. . . swapping on a path upward from p instead of downward

For both increase and decrease, time is $O(\log n)$

Insert new element

To insert a new element x with priority p into a heap that already has n elements:

- Add x to array cell $H[n]$ with priority $+\infty$
- The heap property is satisfied!
  (because of the big but fake priority)
- Change the priority of x to p
  (swapping along path towards root)

Time: $O(\log n)$

Heapify

Given an unsorted array, put it into heap order

Main idea: Fix subtrees that have only one element out of order

(by swapping with children on downward path in tree)

Recursive version:

To recursively heapify the subtree rooted at i:

1. Recursively heapify left child of i
2. Recursively heapify right child of i
3. Fix heap at i

Non-recursive version:

For $i = n-1, n-2, \ldots 0$:
    Fix heap at i

Heapify analysis (either version)

In a binary heap with n tree nodes:

- $\lceil n/2 \rceil$ are leaves, fix heap takes one step
- $\lceil n/4 \rceil$ are parents of leaves, fix heap takes two steps

  ...
- $\lceil n/2^i \rceil$ are i levels up, fix heap takes i steps

Total steps:

$$\sum_{i=1}^{\lceil \log_2(n+1) \rceil} i \left\lceil \frac{n}{2^i} \right\rceil \leq O(\log^2 n) + n \sum_{i=1}^{\infty} \frac{i}{2^i} = 2n + O(\log^2 n)$$

The $O(\log^2 n)$ term is how much the rounding-up part of the formula can contribute, compared to not rounding.)

So total time is $O(n)$, as stated earlier

k-ary heaps

Can we speed this up?

In a comparison model of computation, some priority queue operations must take $\Omega(\log n)$ time

- We can use a priority queue to sort n items by repeatedly finding and removing the smallest one (heapsort)
- But comparison-based sorting requires $\Omega(n \log n)$ time

But in Dijkstra, different operations have different frequencies

- In a graph with n vertices and m edges...
- Dijkstra does n find-and-remove-min operations,
- but up to m decrease-priority operations
- In some graphs, m can be much larger than n

Goal: speed up the more frequent operation (decrease-priority) without hurting the other operations too much

k-ary heap

Same as binary heap but with $k > 2$ children per node

Parent$(x) = \lfloor (x-1)/k \rfloor$

Children$(x) = (kx+1, kx+2, \ldots kx+k)$

Height of tree: $\log_k n = \dfrac{\log n}{\log k}$

Time to decrease priority $= O(\text{height}) = O\left(\dfrac{\log n}{\log k}\right)$

(because path to root is shorter)

Time to find-and-remove-min $= O\left(\dfrac{k \log n}{\log k}\right)$

(because each step finds smallest of k children)

Dijkstra using k-ary heap

Time for m decrease-priority operations: $O\left(m\frac{\log n}{\log k}\right)$

Time for n find-and-remove-min operations: $O\left(nk\frac{\log n}{\log k}\right)$

To minimize total time, choose k to balance these two bounds

$$k = \max(2, \lceil m/n \rceil)$$

$$\text{Total time} = O\left(m\frac{\log n}{\log m/n}\right)$$

This becomes $O(m)$ whenever $m = \Omega(n^{1+\varepsilon})$ for any constant $\varepsilon > 0$

k-ary heaps in practice

4-ary heaps may be better than 2-ary heaps for all operations

For heaps that are too large to fit into cache memory, even larger choices of k may be better

(Lower tree height beats added complexity of more children, especially when each level costs a cache miss)

See `https://en.wikipedia.org/wiki/D-ary_heap` for references

Fibonacci heaps

Overview

A different priority queue structure optimized for Dijkstra:

- Create n-item heap takes time $O(n)$ (same as binary heap)
- Find-and-remove-min is $O(\log n)$ (same as binary tree)
- Insert takes time $O(1)$
- Decrease-priority takes time $O(1)$

$\Rightarrow$ Dijkstra + Fibonacci heaps takes time $O(m + n \log n)$

Can merge heaps (destructively) in time $O(1)$; occasionally useful

All time bounds are amortized, not worst case

In practice, high constant factors make it worse than k-ary heaps

Structure of a Fibonacci heap

It's just a heap-ordered forest!

Each node = an object that stores:

- An element in the priority queue, and its priority
- Pointer to parent node
- Doubly-linked list of children
- Its degree (number of children)
- "Mark bit" (see next slide)

Heap property: all non-roots have priority(parent) $\leq$ priority(self)

Whole heap = object with doubly-linked list of tree roots
and a pointer to the root with the smallest priority

No requirements limiting the shape or number of trees

(But we will see that operations cause their shapes to be limited)

Mark bits and potential function

Each node has a binary "mark bit"

- True: This node is a non-root, and after becoming a non-root it had one of its children removed
- False: It's a root, or has not had any children removed

(We do not allow the removal of more than one child from non-root nodes)

Potential function Φ:
$2 \times$ (number of marked nodes) $+ 1 \times$ (number of trees)

Some easy operations

Create new heap: Make each element its own tree root

Actual time: $O(n)$ $\Delta\Phi = +n$ Total: $O(n)$

Insert new element: Make it the root of its own tree

Actual time: $O(1)$ $\Delta\Phi = +1$ Total: $O(1)$

Merge two heaps: Concatenate their lists of roots

Actual time: $O(1)$ $\Delta\Phi = 0$ Total: $O(1)$

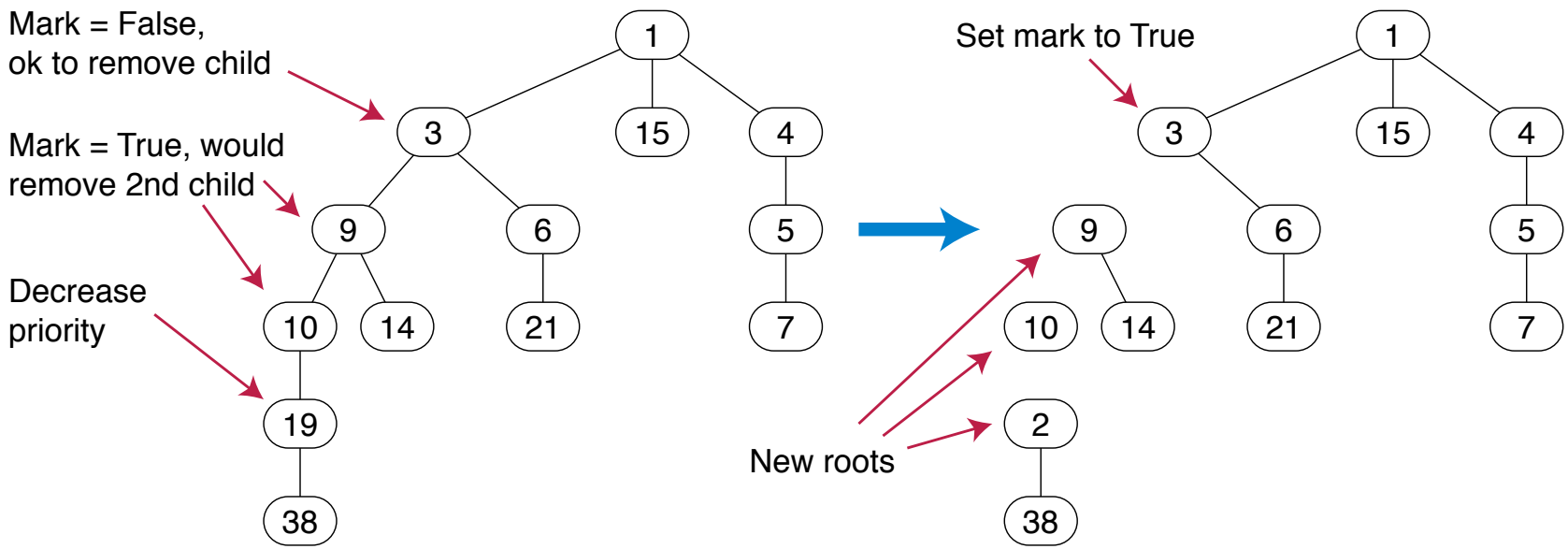
Find min-priority element: Follow pointer to minimum root

Actual time: $O(1)$ $\Delta\Phi = 0$ Total: $O(1)$

Decrease-priority operation, main idea

Cut edge from decreased node to its parent, making it a new root

If this removes 2nd child from any node, make it a new root too



Decrease-priority details

To decrease the priority of element at node x:

1. $y = \text{parent}(x)$
2. Cut link from x to y (making x a root)
3. Set $\text{mark}(x) = \text{False}$
4. While y is marked:
   - $z = \text{parent}(y)$
   - Cut link from y to z (making y a root)
   - Set $\text{mark}(y) = \text{False}$
   - Set $y = z$
5. If y is not a root, set $\text{mark}(y) = \text{True}$

If we cut k links, then the total actual time is $O(k)$. We unmark $k - 1$ nodes, create k new trees, and mark at most one node, so $\Delta\Phi \leq -2(k-1) + k + 2 = 4 - k$. The $-k$ term in $\Delta\Phi$ cancels the $O(k)$ actual time leaving $O(1)$ amortized time.

Delete-min operation

The only operation that creates bigger trees from smaller ones...

1. Make all children of deleted node into new roots
2. Merge trees with equal degree until no two are equal

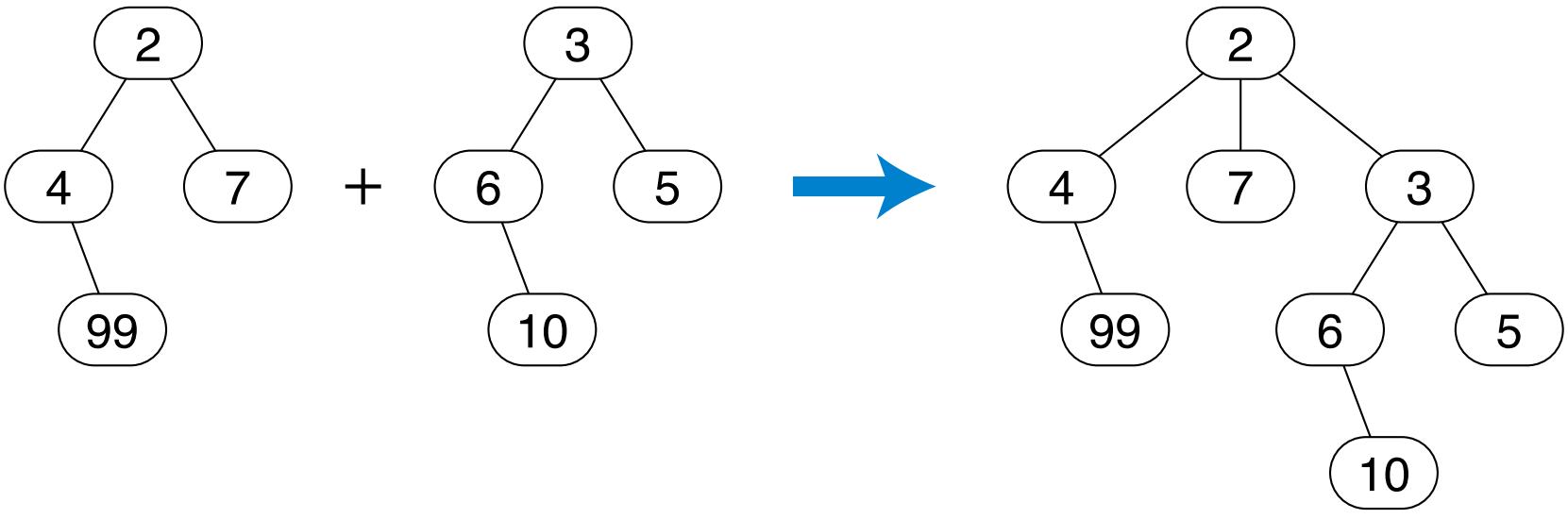
Actual time: $O(\#\text{ children} + \#\text{ merges} + \#\text{ remaining trees})$

$\Delta\Phi$: number of children - number of merges

Total amortized time:
$O(\#\text{ children} + \#\text{ remaining trees}) = O(\max \#\text{ children})$

Merging two trees

Make the larger root become the child of the smaller root!



We will only do this when both trees have equal degree (number of children)

$\Rightarrow$ Degree goes up by one

Merging until all trees have different degrees

1. Make a dictionary D, where $D[x]$ will be a tree of degree x
   x will be a small integer, so D can just be an array
2. Make a list L of trees needing to be processed
   Initially $L =$ all trees
3. While L is non-empty:
   - Let T be any tree in L; remove T from L
   - Let x be the degree of tree T
   - If key x is not already in D, set $D[x] = T$
   - Otherwise merge $D[x]$ and T, remove x from D, and add the merged tree to L
4. Return the collection of trees stored in D

Total times through loop $= 2 \times \#$ merges $+ \#$ remaining trees

Degrees of children

Lemma: Order the children of any node by the time they became children (from a merge operation)

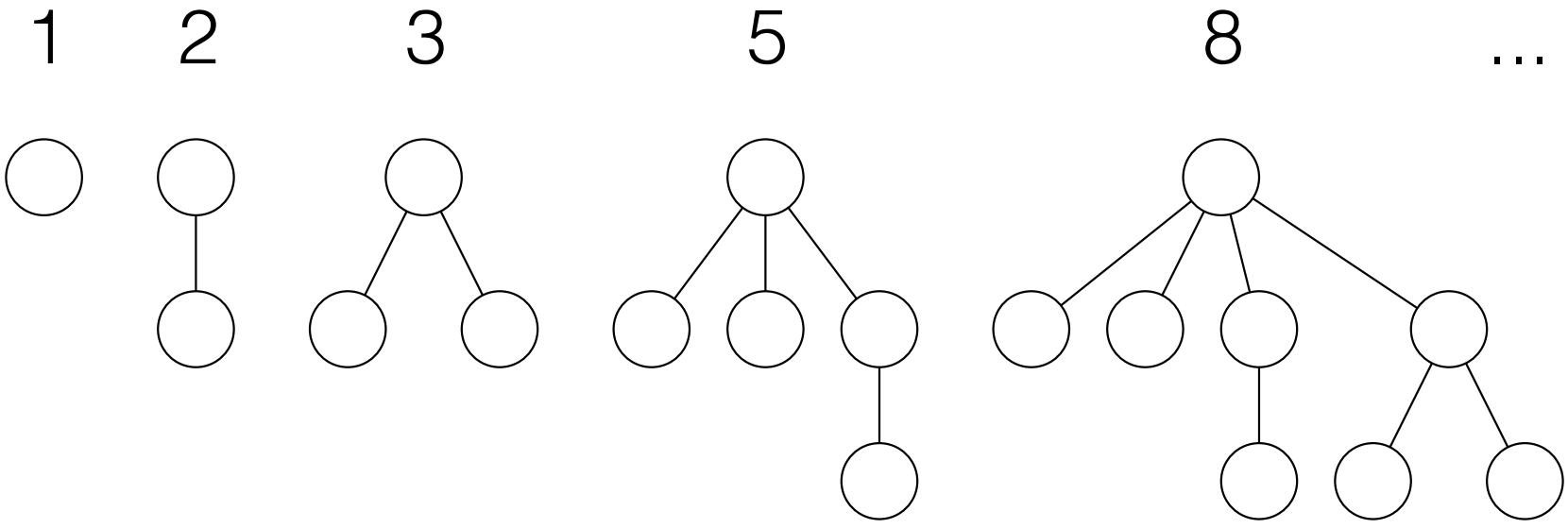
Then their degrees are $\geq 0, 0, 1, 2, 3, 4, 5, \ldots$

Proof: When ith child was merged in, earlier children were already there $\Rightarrow$ parent degree was $\geq i - 1$

At the time of the merge, it had same degree as parent, $\geq i - 1$

After taking away at most one child, its degree is $\geq i - 2$

The smallest trees obeying the degree lemma



Tree size is exponential in degree

Lemma: Subtree of degree x has # nodes $\geq$ Fibonacci(x)

Proof: By induction

Base cases $x = 0$ (one node) and $x = 1$ (two nodes)

For $x > 1$:

$$\begin{aligned}
\text{number of nodes} &\geq \text{root} + \text{numbers in each subtree} \\
&\geq 1 + \sum_{i=1}^{x} \max(1, F_{i-2}) \\
&\geq (\text{same sum for one fewer child}) + F_{x-2} \\
&= F_{x-1} + F_{x-2} = F_x
\end{aligned}$$

Consequences of the degree lemma

Corollary 1: Maximum degree in an n-node Fibonacci heap is $\leq \log_\varphi n$ where $\varphi = (1 + \sqrt{5})/2$ is the golden ratio

Corollary 2: Amortized time for delete-min operation is $O(\log n)$

Ongoing research question

Can we achieve the same time bounds as Fibonacci (constant amortized decrease-priority, logarithmic delete-min) with a structure as practical as binary heaps?

Integer priority queues

Main idea

k-ary heaps, Fibonacci heaps only use comparison of priorities

If priorities are small integers, we can use bitwise binary operations, array indexing, etc

Sometimes this can be faster

Integer priority queues versus sorting

Many non-comparison-based sorting algorithms are known

E.g. radix sort: Sort n integers in range $[0, X]$ in time $O\left(n\frac{\log X}{\log n}\right)$

If we can do priority queue operations in time T, we can use them to sort in time $O(nT)$ ("heap sort")

Reverse is true!

If we can sort in time $nT(n)$, we can do priority queue operations in time $T(n) + T(\log n) + T(\log\log n) + \cdots$

Thorup, "Equivalence between priority queues and sorting", *JACM* 2007

Theoretically fast but completely impractical

Current best known (expected) time bound for integer sorting, when priorities are small enough to be represented by machine integers: $O(n\sqrt{\log\log n})$

Han & Thorup, "Integer sorting in $O(n\sqrt{\log\log n})$ expected time and linear space", FOCS 2002

Corollary: Can do integer priority queue operations in $O(\sqrt{\log\log n})$ expected time per operation

We will look at simpler data structures whose time depends on the range of values, not just on n

Open: Can we do better or is there a non-constant lower bound?

Bucket queue

Priority queue version of bucket sort

When all priorities are small non-negative integers:

- Make an array of buckets, indexed by priority
- Each bucket: collection of elements with that priority
- Add element; put it into the bucket for its priority
- Change priority: move it from one bucket to another
- Find-minimum: scan array for first non-empty bucket

Time: O(current minimum priority) for find-min, $O(1)$ otherwise

Monotonic priorities and Dial's algorithm

In Dijkstra's algorithm, priorities never decrease

$\Rightarrow$ Can start scan for non-empty bucket at the last one we found, instead of starting at the beginning of the array

Total time for priority queue operations:
$O(\#\text{ operations} + \text{max priority})$

Total time for shortest paths in graphs with integer edge lengths:
linear in size of graph + $O(\text{max path length})$

Dial, "Algorithm 360: Shortest-path forest with topological ordering", *CACM* 1969

Flat trees
(also known as Van Emde Boas trees)

Main idea

Think of priorities as b-bit binary numbers, for some b

Partition each priority x into two $b/2$-bit numbers
(the two digits in its base-$2^{b/2}$ representation)

- x_{low} : `x & ((1 << b/2) - 1)`
- x_{high} : `x >> (b/2)`

Build recursive priority queues for the low parts and the high parts

Each operation recurses into low or high but not both
$\Rightarrow$ time $O(\log b) = O\big(\log\log(\text{max priority})\big)$

Simplifying assumption: elements = priorities

If we have elements that are not the same thing as the priorities of the elements, then:

- Make a dictionary mapping each priority to a collection of elements with that priority (like bucket queue)
- Keep a priority queue of the set of priorities whose collections are non-empty
- To find an element of minimum priority: use the priority queue to find the priority, and the dictionary to find the element

The dictionary part is $O(1)$ expected time per operation (using a hash table) so we only need to worry about priority queues whose elements are integers and whose priorities are their values

Flat tree structure

The flat tree for a set S of integers stores:

- The minimum element in S
- Recursive flat tree H of x_{high} for $x \in S \setminus \{\min\}$
- Dictionary D mapping each x_{high} to a recursive flat tree of values x_{low} for the subset of elements $x \in S \setminus \{\min\}$ that have that value of x_{high}

To find the minimum priority:
Look at the value stored as minimum at root-level structure

To make one-element flat tree:
Set minimum to element, H to None, D to empty dictionary

Example

Suppose we are storing a flat tree for the nine 8-bit keys 00000001, 00000011, 00001010, 00001111, 10100101, 10101010, 11011110, 11011111, and 11101010. Then:

The minimum element is 00000001

The values x_{high} for the remaining keys are 0000, 1010, 1101, and 1110. The recursive flat tree H stores these four 4-bit keys.

The dictionary D maps each of these four values x_{high} to a recursive flat tree $D[x_{\text{high}}]$:

- $D[0000]$ stores the low 4-bit keys whose high 4 bits are 0000, not including the minimum element 00000001. These 4-bit keys are: 0011, 1010, and 1111.
- $D[1010]$ stores the two low 4-bit keys 0101 and 1010
- $D[1101]$ stores the two low 4-bit keys 1110 and 1111
- $D[1110]$ stores as its single element the low 4-bit key 1010.

To insert an element x:

1. If $x <$ minimum, swap x with minimum
2. If x_{high} is in D:

   Recursively insert x_{low} into $D[x_{\text{high}}]$
3. Else:

   Make one-element flat tree containing x_{low}

   Add the new tree to D as $D[x_{\text{high}}]$

   Recursively insert x_{high} into H
   (or make one-element flat tree for x_{high} if H was empty)

To find the second-smallest element

1. If D is empty, raise an exception
2. Let x_{high} be the minimum stored in H
3. Let x_{low} be the minimum stored in $D[x_{\text{high}}]$
4. Combine x_{high} and x_{low} and return the result

Expected time $O(1)$

Key subroutine in element removal

To remove an element x:

1. If x is the stored minimum:

   Set x and stored minimum to second-smallest value

   (still need to remove from recursive structure,
   just as if we were deleting it)
2. Compute x_{high} and x_{low}
3. If $D[x_{\text{high}}]$ is a one-element flat tree (its recursive D is empty):

   Recursively delete x_{high} from H

   Remove x_{high} from D
4. Else:

   Recursively delete x_{low} from $D[x_{\text{high}}]$

Summary

Summary

- Priority queue operations, application in Dijkstra's algorithm
- Limited set of operations in Python and how to work around it
- Binary heaps, $O(\log n)$ time per operation, $O(n)$ heapify
- k-ary heaps, slower delete-min and faster decrease-priority
- Optimal choice of k for k-ary heaps in Dijkstra's algorithm
- Fibonacci heaps, $O(\log n)$ delete-min, $O(1)$ decrease-priority
- Integer priority queues and their relation to integer sorting
- Bucket queues and Dial's algorithm
- Flat trees, $O(\log \log (\text{max priority}))$ per operation