Unit #2: Priority Queues

CPSC 221: Algorithms and Data Structures

Will Evans and Jan Manuch

2016W1

Unit Outline

- Rooted Trees, Briefly
- Priority Queue ADT
- Heaps
  - Implementing Priority Queue ADT
  - Focus on Create: Heapify
  - Brief introduction to *d*-Heaps

Learning Goals

- Provide examples of appropriate applications for priority queues and heaps
- Manipulate data in heaps
- Describe and apply the Heapify algorithm, and analyze its complexity

Rooted Trees

- Family Trees
- Organization Charts
- Classification trees (a.k.a. keys)
  - What kind of flower is this?
  - Is this mushroom poisonous?
- File directory structure
  - folders, subfolders in Windows
  - directories, subdirectories in UNIX
- Non-recursive call graphs

Tree Terminology

vertex
node

root: A

leaf: D E F ...

child: of A: B and C

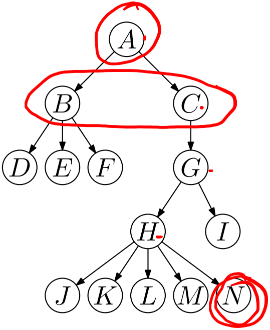
parent: of H: G

sibling: B and C

ancestor: of N: N, H, G, C, A

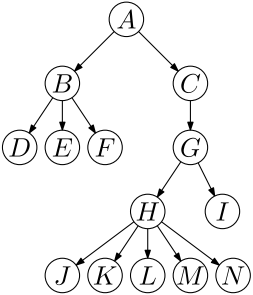
descendent: of C: C, G, I, H, J, K, L, M, N

subtree: of G: G and all descendents



Tree Terminology Reference

root: the single node with no parent



leaf: a node with no children

child: a node pointed to by me

parent: the node that points to me

sibling: another child of my parent

ancestor: my parent or my parent's ancestor

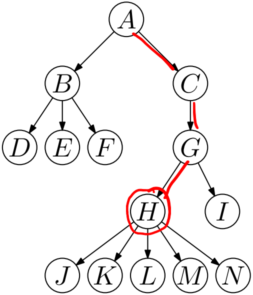
descendent: my child or my child's descendent

subtree: a node and its descendents

More Tree Terminology

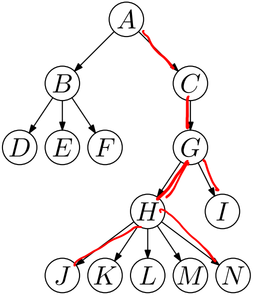
depth: Number of edges on path from root to node

depth of H? 3



More Tree Terminology

height: Number of edges on longest path from node to descendent or, for whole tree, from root to leaf



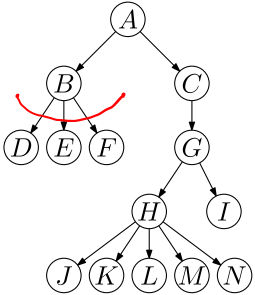
height of tree? ≡ height of root
= 4

height of G?
= 2

More Tree Terminology

(downward) degree: Number of children of a node

degree of B? 3

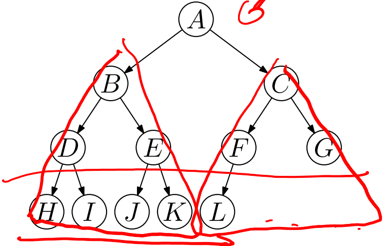


One More Tree Terminology Slide

ordered (structure)

ht = 3

num nodes = 4



binary: each node has degree at most 2

d-**ary:** degree at most *d*

nodes n in a binary tree of height h

$$h+1 \le \boxed{n \le 2^{h+1} - 1}$$

complete: as many nodes as possible for its height (each row filled in)

nearly complete: each row except the last one is filled in, all nodes in the last row are as far left as possible

If nearly complete tree has n nodes, what is its height? $\lfloor \lg n \rfloor$

tricky ~~to~~ to prove

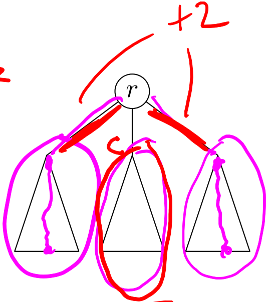
Longest Path

Find the longest *undirected* path in a tree

Longest path ~~is~~ either

(1) is within a child subtree

or (2) contains the root



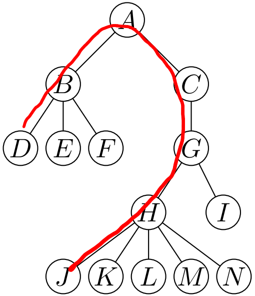
$$\text{longpath}(T) = \max \begin{cases} (1) \ \max\limits_{\text{child } c} \text{longpath}(T_c), \\ (2) \ 2 + \max\limits_{\text{child } c \neq d} \text{height}(T_c) + \text{height}(T_d) \end{cases}$$

T_c

$$\text{longpath}(T) = 0 \quad \text{if } |T| = 1$$

$$\text{height}(T) = 1 + \max_{\text{child } c} \text{height}(T_c)$$

Longest Path Example



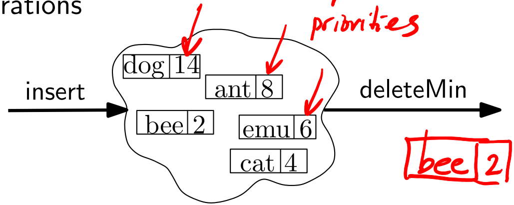
Back to Queues

- Applications
  - ordering CPU jobs
  - simulating events
  - picking the next search site
- But we don't want FIFO ...
  - *short* jobs should go first
  - *earliest* (simulated time) events should go first
  - *most promising* sites should be searched first

Priority Queue ADT

priorities associated with elements in queue

- Priority Queue operations
  - create
  - destroy
  - insert
  - deleteMin
  - is_empty



- Priority Queue property: For two elements in the queue, x and y, if x has a lower priority value than y, x will be deleted before y.

Applications of the Priority Q

- Hold jobs for a printer in order of length
- Store packets on network routers in order of urgency
- Simulate events
- Select symbols for compression
- Sort numbers
- Anything *greedy*: an algorithm that makes the "locally best choice" at each step

Priority Q Data Structures

- Unsorted list
  - insert time: $\Theta(1)$
  - deleteMin time: $\Theta(n)$
- Sorted list
  - insert time: $\Theta(n)$
  - deleteMin time: $\Theta(1)$

Binary Heap Priority Q Data Structure

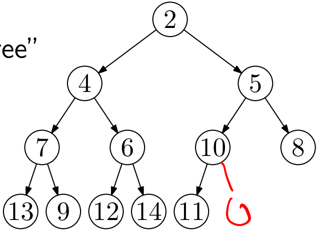
only showing priorities

Heap-order property: parent's key $\leq$ children's keys.

- minimum is always at the top

Structure property: "nearly complete tree"

- depth is always O(log n)
- next open location always known



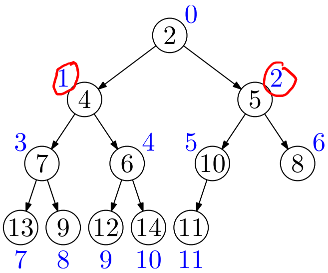
WARNING: This has NO SIMILARITY to the "heap" you hear about when people say "things you create with `new` go on the heap".

Nifty Storage Trick

n elements

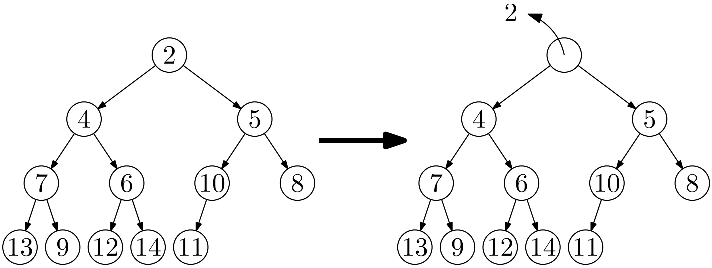
Navigation using indices:

- left_child(i) = $2i+1$
- right_child(i) = $2i+2$
- parent(i) = $\lceil \frac{i}{2} \rceil - 1 = \lfloor \frac{i-1}{2} \rfloor$
- root = 0
- next free position = n



0	1	2	3	4	5	6	7	8	9	10	11	12
2	4	5	7	6	10	8	13	9	12	14	11	

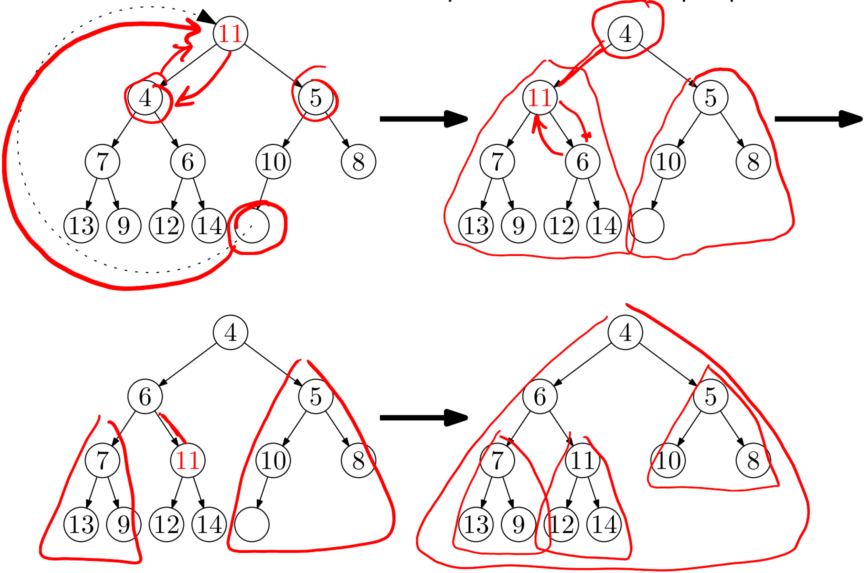
DeleteMin



Invariants violated! No longer “nearly complete”

Swap (Heapify) Down

Move last element to root then swap it down to its proper position.



DeleteMin Code

will be the index of smaller of i here

s

```
int deleteMin() {
  assert(!isEmpty());
  int returnVal = Heap[0];
  Heap[0] = Heap[n-1];
  n--;
  swapDown(0);
  return returnVal;
}
```

Runtime:

$O(\log n)$

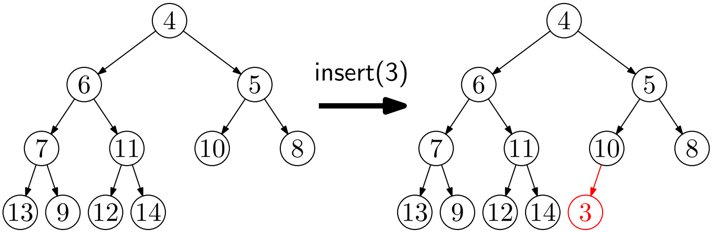
if recursive call is made

$s > 2i$

$\Rightarrow$ # recursive calls $< \log_2 n$

```
void swapDown(int i) {
  int s = i;
  int left = i * 2 + 1;
  int right = left + 1;
  if( left < n &&
      Heap[left] < Heap[s] )
    s = left;
  if( right < n &&
      Heap[right] < Heap[s] )
    s = right;
  if( s != i ) {
    int tmp = Heap[i];
    Heap[i] = Heap[s];
    Heap[s] = tmp;
    swapDown(s);
  }
}
```

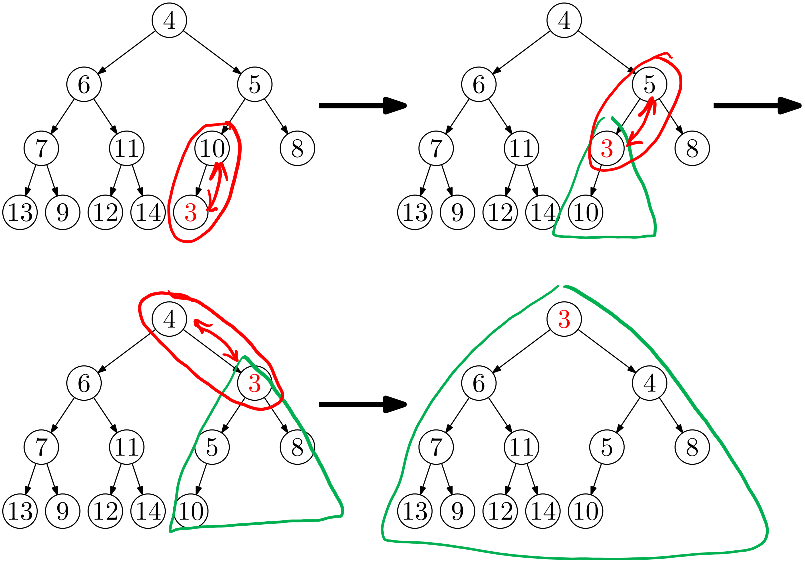
Insert



Invariant violated! Child has smaller key than parent.

Swap (Heapify) Up

Put new element last then swap it up to its proper position.



Insert Code

```
void insert(int x) {
  assert(!isFull());
  Heap[n] = x;
  n++;
  swapUp(n-1);
}
```

```
void swapUp(int i) {
  if( i == 0 ) return;
  int p = (i - 1)/2;
  if( Heap[i] < Heap[p] ) {
    int tmp = Heap[i];
    Heap[i] = Heap[p];
    Heap[p] = tmp;
    swapUp(p);
  }
}
```

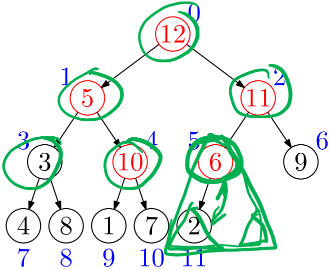
Runtime: $O(\log n)$

$p < i/2$

Heapify: Build a Heap from a non-Heap Array

1. Start with the input array.

12	5	11	3	10	6	9	4	8	1	7	2



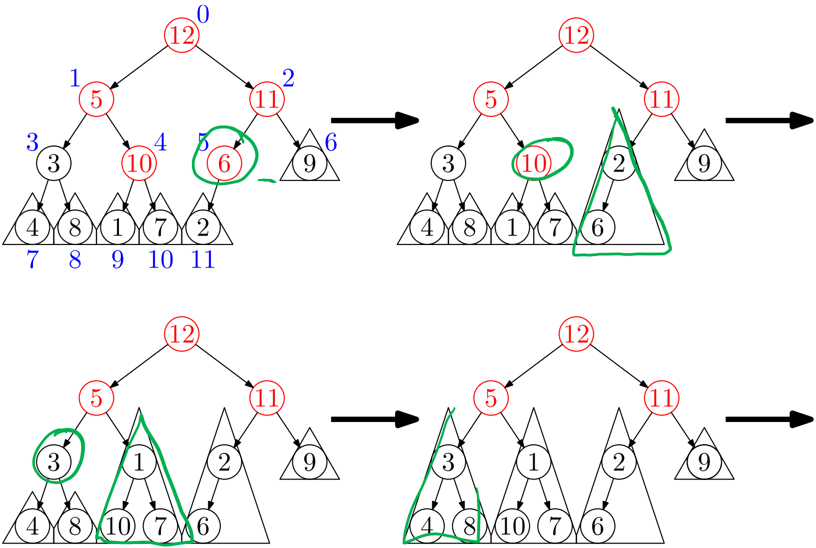
Invariant violated!

2. Fix the heap-order property bottom up. Use `swapDown`.

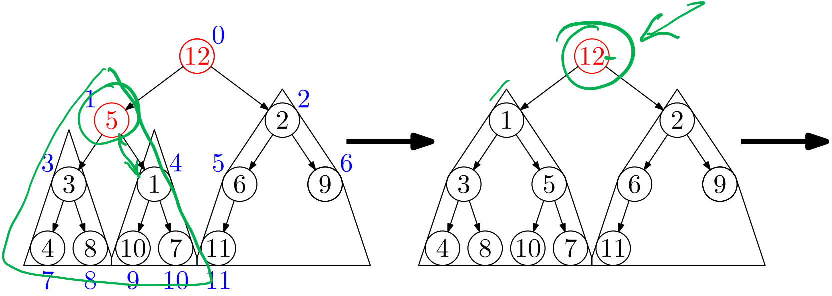
```
for( i=n/2-1; i >=0; i-- ) swapDown(i);
```

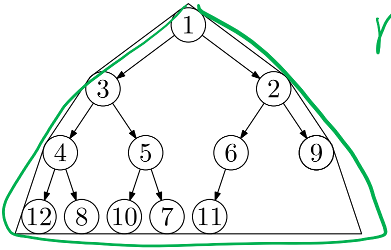
Heapify Example...

△'s mean proper heaps



Heapify Example





runtime

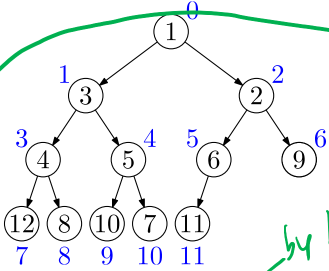
$\le$ #swapDown's

$\times \lg n$

$\in O(n \log n)$

Heapify Runtime

`swapDown` on a heap of height h takes at most $\underline{\quad h \quad}$ steps.



Let H be the height of the heap.

$(H = \lfloor \lg n \rfloor)$

by heapify

`swapDown` is called	once	on heap of height	H
	≤ 2 times	on heap of height	$H-1$
	≤ 4 times	on heap of height	$H-2$
	$\vdots$ $\leq 2^{H-h}$	" " " "	h
	$\leq 2^{H-1}$ times	on heap of height	1

Total # steps $\leq \sum_{h=1}^{H} h 2^{H-h} = 2^H \sum_{h=1}^{H} h/2^h \leq 2^{H+1} = O(n)$

$$2^H \sum_{h=1}^{H} h/2^h \le 2^{H+1}$$

$$\le 2^H \left(\boxed{\tfrac{1}{2} + \tfrac{2}{4} + \tfrac{3}{8} + \cdots} \right) = S$$

$$= 2^H \cdot S$$

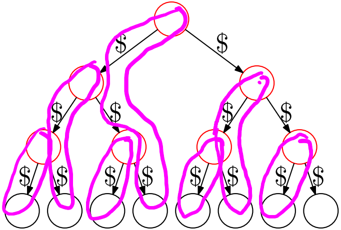
$$2S = 1 + \tfrac{2}{2} + \tfrac{3}{4} + \cdots$$

$$-S \quad \tfrac{1}{2} + \tfrac{2}{4} + \tfrac{3}{8} \cdots$$

$$= 1 + \tfrac{1}{2} + \tfrac{1}{4} + \tfrac{1}{8} + \cdots$$

$$S \qquad \le 2$$

Heapify Runtime: Charging Scheme



Possible violations. How much time to fix them?
Place a dollar on each edge of the heap. One dollar pays for one step of `swapDown`. By induction, we can show that when `swapDown` is called on a node v, both children of v have a path (the rightmost path) to a leaf that is uncharged. The edges on the left child's rightmost path plus the edge to the left child pay for the steps of `swapDown` at v. The edges on the right child's rightmost path plus the edge to the right child form the uncharged path available to the parent of v.

Thinking about Binary Heaps

Observations

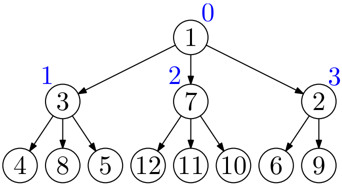
- finding a child/parent index is a multiply/divide by two
- deleteMin and insert access far-apart array locations
- deleteMin accesses all children of visited nodes
- insert accesses only parent of visited nodes
- insert is at least as common as deleteMin

Realities

- division and multiplication by powers of two are fast
- far-apart array accesses ruin cache performance
- with huge data sets, disk I/O dominates

Solution: *d*-Heaps

Nearly complete *d*-ary trees (representable by array) with Heap-order property.



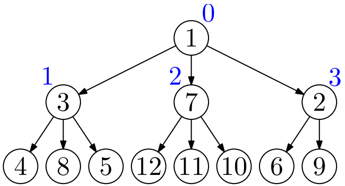
1	3	7	2	4	8	5	12	11	10	6	9
0	1	2	3	4	5	6	7	8	9	10	11

Good choices for *d*:

- fit one set of children on a memory page/disk block
- fit one set of children in a cache line
- optimize performance based on ratio of inserts/deleteMins
- make *d* a power of two for efficiency

d-Heap Navigation

- jth-child(i) = $di + j$
- parent(i) = $\left\lfloor \frac{i-1}{d} \right\rfloor$
- root = 0
- next free position = n



1	3	7	2	4	8	5	12	11	10	6	9
0	1	2	3	4	5	6	7	8	9	10	11