

## Unit Outline

### Unit #2: Priority Queues

CPSC 221: Basic Algorithms and Data Structures

Anthony Estey, Ed Knorr, and Mehrdad Oveis

2016W2: January-April 2017

- ▶ Rooted Trees (Briefly)
- ▶ Priority Queue ADT
- ▶ Heaps
  - ▶ Implementing a Priority Queue ADT
  - ▶ Operations on a Heap
  - ▶ Building a Heap via Heapify
  - ▶ Analysis of Operations
  - ▶ Brief Introduction to  $d$ -Heaps

2 / 32

## Learning Goals

- ▶ Define terminology about trees.
- ▶ Provide examples of appropriate applications for priority queues and heaps.
- ▶ Manipulate data in heaps.
- ▶ Describe and apply the Heapify algorithm, and analyze its complexity.

3 / 32

## Rooted Trees and Some Applications

- ▶ Family Trees
- ▶ Organization Charts
- ▶ Classification Trees
  - ▶ What kind of flower is this?
  - ▶ Is this mushroom poisonous?
- ▶ File Directory Structure
  - ▶ Folders and Subfolders in Windows
  - ▶ Directories and Subdirectories in UNIX
- ▶ Non-Recursive Call Graphs
- ▶ Indexes in Database Systems



4 / 32

## Tree Terminology: Examples

root:

leaf:

child of \_\_\_\_\_:

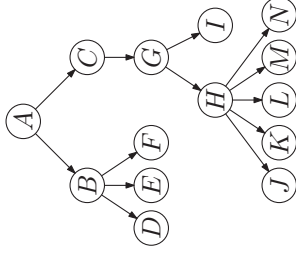
parent of \_\_\_\_\_:

sibling:

ancestor of \_\_\_\_\_:

descendent of \_\_\_\_\_:

subtree of \_\_\_\_\_:

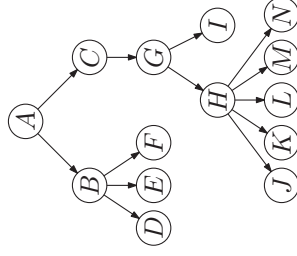


5 / 32

## More Tree Terminology

depth: number of edges on path from root to node

depth of *H*?



7 / 32

## Tree Terminology Reference

root: the single node with no parent

leaf: a node with no children

child: a node pointed to by me

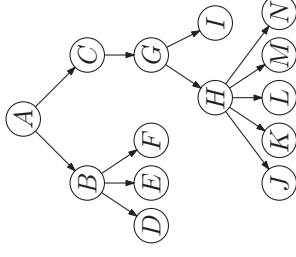
parent: the node that points to me

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 descendants



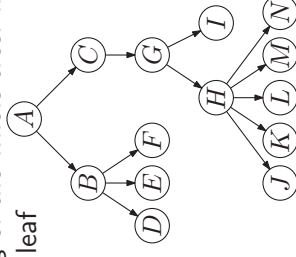
6 / 32

## More Tree Terminology

height: number of edges on longest path from a given node to its furthest descendent; or, when speaking of the whole tree: number of edges on longest path from root to leaf

height of tree?

height of *G*?

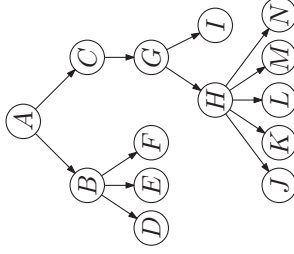


8 / 32

## More Tree Terminology

(downward) degree: number of children of a given node

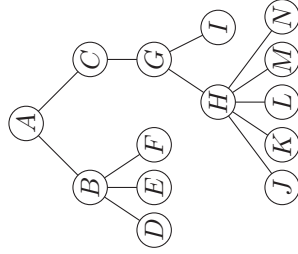
degree of  $B$ ?



9 / 32

## Example: Finding the Longest Undirected Path in a Tree

Does such a path always include the root?



11 / 32

## One More Tree-Terminology Slide

**binary:** Each node has degree at most 2.

**$d$ -ary:** The degree is at most  $d$ .

**full:** Each internal (non-leaf) node has the maximum number of children (2 in the case of a binary tree).

**complete:** It has as many nodes as possible for its height (i.e., each row is filled in).

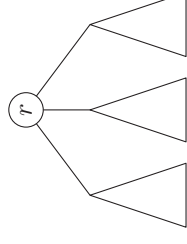
**nearly complete:** Each row, except possibly the last one, is filled in, and all nodes in the last row are as far left as possible.

(Warning: Some authors like Koffman/Wolfgang call this a *complete* tree. We'll stick with *nearly complete*.)

10 / 32

## Longest Path

An algorithm to find the longest *undirected* path in a tree:



12 / 32

## Back to Queues

- ▶ Applications
  - ▶ Ordering jobs/processes on a CPU
  - ▶ Simulating events
  - ▶ Picking the next search site
- ▶ But we don't necessarily want FIFO. You can choose your order, according to some carefully thought-out priority. Maybe:
  - ▶ *Shorter* jobs should go first.
  - ▶ *Earliest* (simulated time) events should go first.
  - ▶ *Most promising* sites should be searched first.

13 / 32

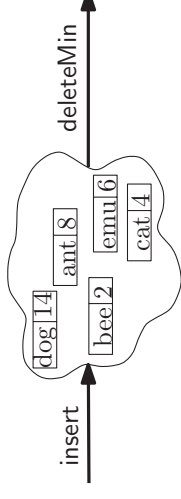
## Applications of a Priority Queue

- ▶ 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*: In this case, an algorithm makes the “locally best choice” (not necessarily the overall best choice) at each step.

15 / 32

## Priority Queue ADT

- ▶ Priority Queue Operations
  - ▶ create
  - ▶ destroy
  - ▶ insert
  - ▶ deleteMin
  - ▶ is\_empty
- ▶ Priority Queue Property (in a minimum priority queue): For two elements in the queue,  $x$  and  $y$ , if  $x$  has a lower priority value than  $y$ ,  $x$  will be deleted before  $y$  when performing a deleteMin operation.



14 / 32

## Priority Queue Data Structures

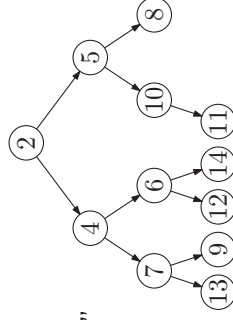
- ▶ Unsorted List
  - ▶ insert time:
  - ▶ deleteMin time:
- ▶ Sorted List
  - ▶ insert time:
  - ▶ deleteMin time:

16 / 32

## Binary Heap Priority Queue Data Structure

Heap-Order Property: parent's key  $\leq$  children's key (we often call this a *minimum* heap)

- ▶ minimum is always at the top



Structure Property: "nearly complete tree"

- ▶ depth is always  $O(\lg n)$
- ▶ next open location is always known

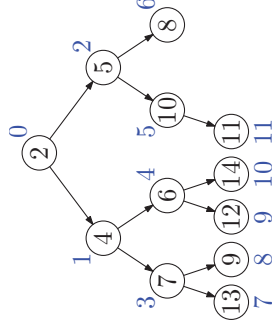
**WARNING:** This has no similarity to the memory "heap" we talk about when using C++'s `new` operator.

17 / 32

## Nifty Storage Trick

Navigation using indices:

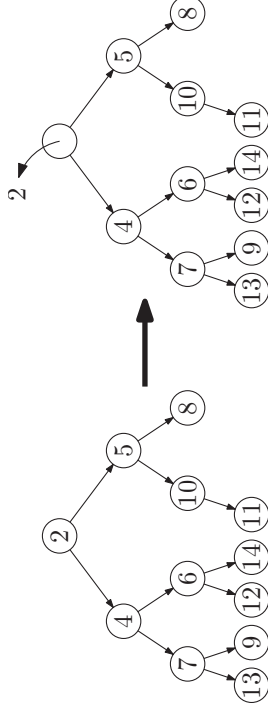
- ▶  $\text{left\_child}(i) =$
- ▶  $\text{right\_child}(i) =$
- ▶  $\text{parent}(i) =$
- ▶  $\text{root} =$
- ▶ next free position =



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	

18 / 32

## deleteMin

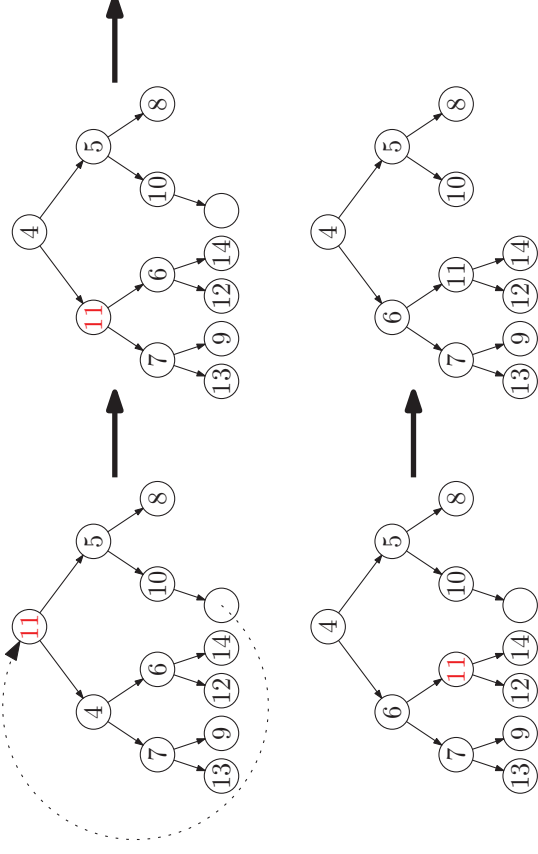


Invariants violated! It's no longer a "nearly complete" binary tree.

19 / 32

## Swap (Heapify) Down

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



20 / 32

## deleteMin Code

```

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

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);
    }
}

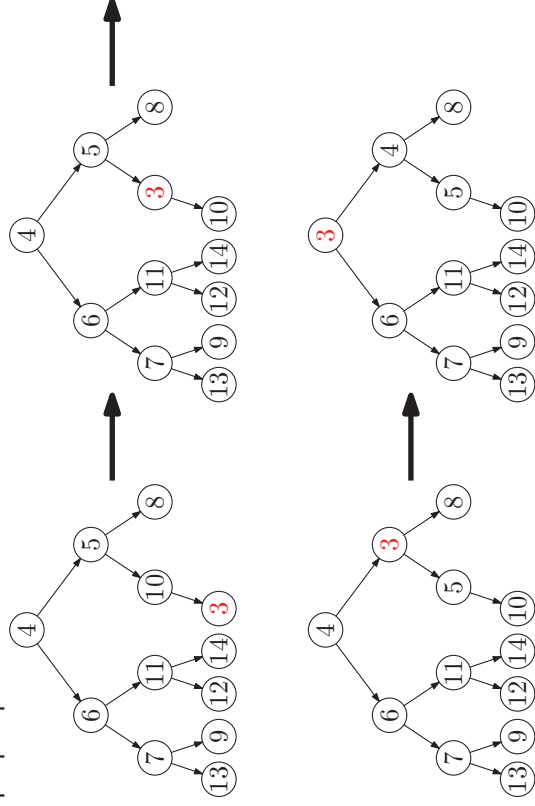
```

Runtime:

21 / 32

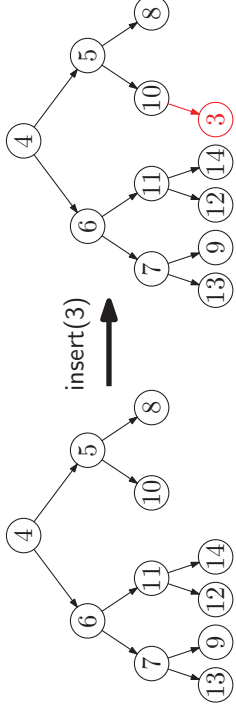
## Swap (Heapify) Up

Begin by putting the new element last, then swap it up to its proper position.



23 / 32

## Inserting a New Node



Invariant violated! Child has smaller key than parent.

22 / 32

## 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);
    }
}

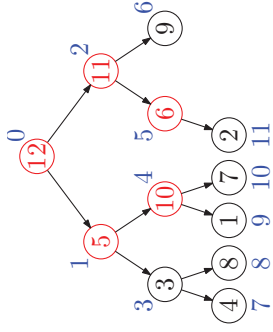
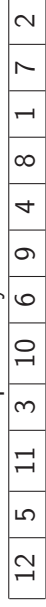
```

Runtime:

24 / 32

## Heapify: Build a Heap from an Array

1. Start with the input array.

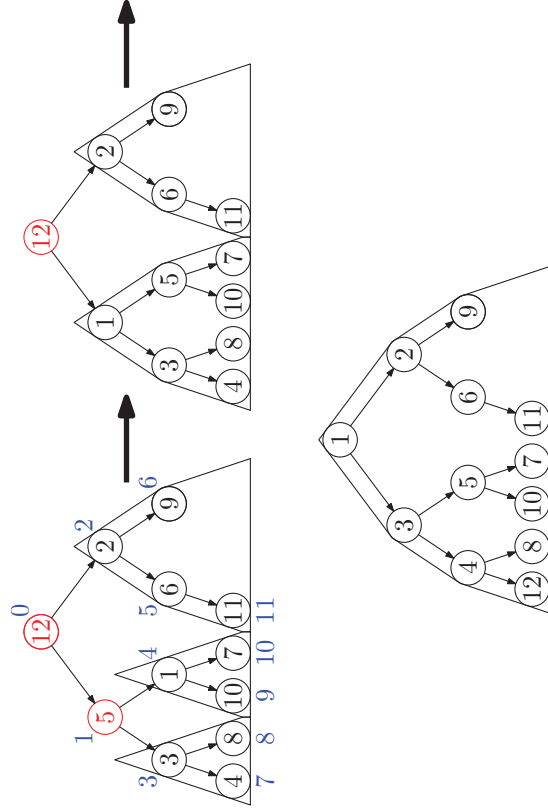


Invariant violated!

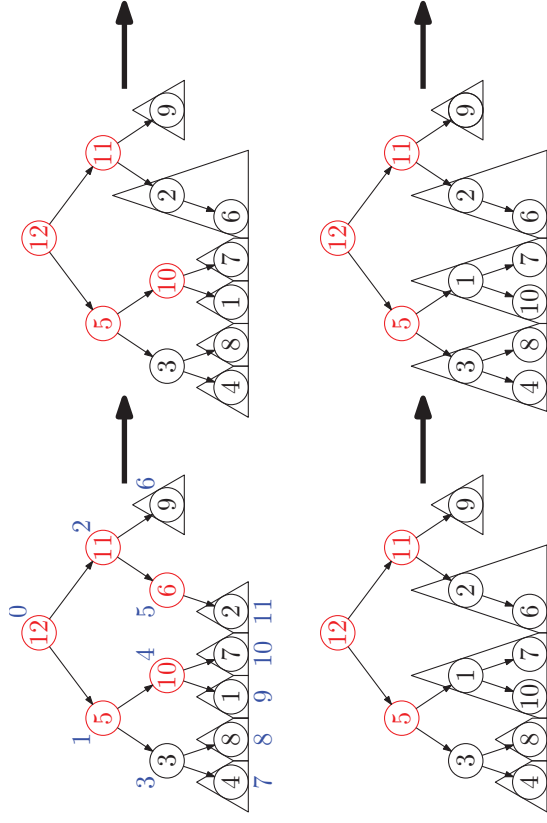
2. Fix the heap-order property, starting from the bottom, and going up. Use swapDown.

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

## Heapify Example

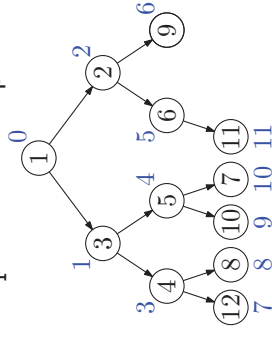


## Heapify Example...



## Heapify Runtime

swapDown on a heap of height  $h$  takes at most \_\_\_\_\_ steps.

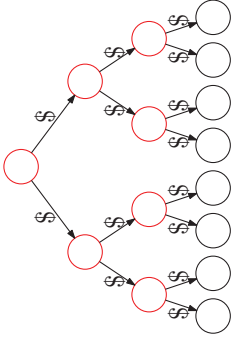


Let  $H$  be the height of the heap.

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

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

## Heapify Runtime: Charging Scheme



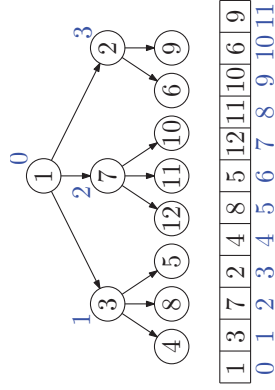
Possible **violations**. How much time to fix them?

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  $v$ .

29 / 32

## Solution: $d$ -Heaps

These are complete  $d$ -ary trees (representable by an array) with a heap-order property.



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

31 / 32

## Thinking about Binary Heaps

Observations

- ▶ Finding a child/parent index is a multiply/divide by two operation.
- ▶ Both `deleteMin` and the subsequent insert might access far-apart array locations.
- ▶ `deleteMin` accesses all children of visited nodes.
- ▶ insert accesses only the parent of visited nodes.
- ▶ insert is at least as common as `deleteMin`.

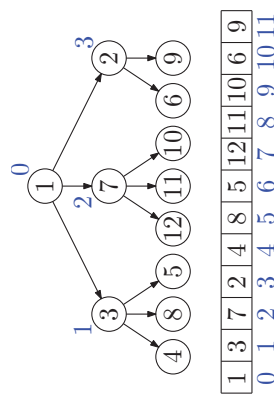
Realities

- ▶ Division and multiplication by powers of two are fast.
- ▶ Far-apart array accesses can ruin cache performance.
- ▶ With large datasets, disk I/O dominates CPU time.

30 / 32

## $d$ -Heap Navigation

- ▶  $j$ th-child( $i$ ) =
- ▶ parent( $i$ ) =
- ▶ root =
- ▶ next free position =



32 / 32