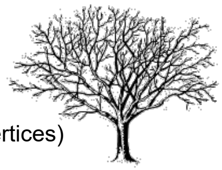
Lecture 8
Heaps, Priority Queues, and Heapsort

EECS 281: Data Structures & Algorithms

Trees



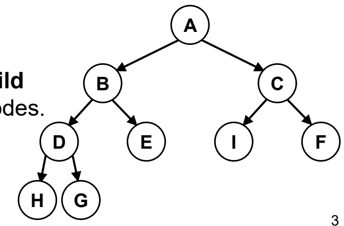
A **graph** consists of **nodes** (sometimes called vertices) connected together by **edges**.

Each node can contain some data.

A **tree** is:
(1) a connected graph (nodes + edges) w/o cycles.
(2) a graph where any 2 nodes are connected by a unique shortest path.
(the two definitions are equivalent)

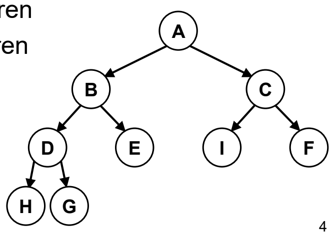
In a directed tree, we can identify **child** and **parent** relationships between nodes.

In a **binary tree**, a node has at most two children.



Tree Terminology

Root: The "topmost" node in the tree
Parent: Immediate predecessor of a node
Child: Node where current node is parent
Ancestor: Parent of a parent (closer to root)
Descendent: Child of a child (further from root)
Internal Node: A node with children
Leaf Node: A node without children



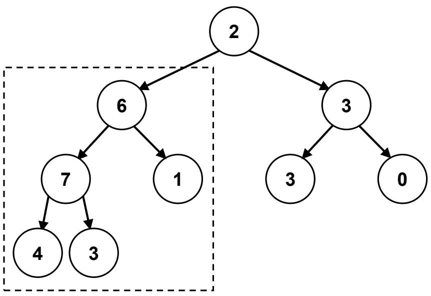
Data Representation: Nodes and Pointers

```
template <class Item>
struct Node {    // binary tree node
  Node *left;    // pointer to left child
  Node *right;   // pointer to right child
  Item item;     // data or KEY
}; // Node{}
```

- A node contains some information, and points to its left child node and right child node
  - Can also include a pointer for parent node
  - Can include pointers to 3 children, or a vector of pointers
- Efficient for moving *down* a tree from parent to child

Trees are Recursive Structures

Any subtree is just as much a "tree" as the original!

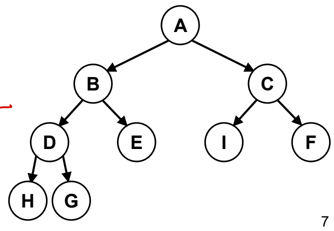


Tree Properties

Height:
`height(empty) = 0`
`height(node) = max(height(left_child), height(right_child)) + 1`

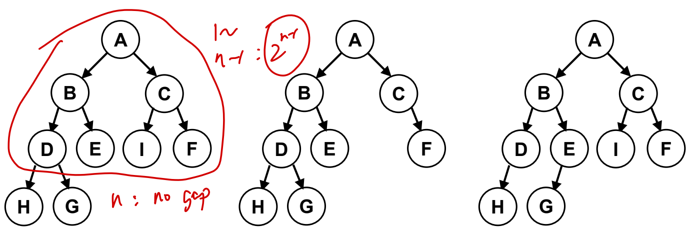
Size:
`size(empty) = 0`
`size(node) = size(left_child) + size(right_child) + 1`

Depth:
`depth(empty) = 0`
`depth(node) = depth(parent) + 1`



Complete (Binary) Trees

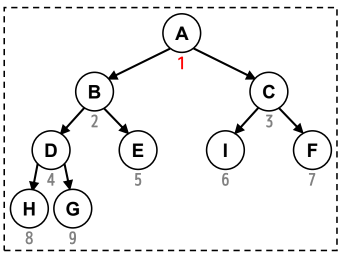
Binary Tree: every node has 2 or fewer children
Complete Binary Tree: every level, *except possibly the last*, is completely filled, and all nodes are as far left as possible



complete NOT complete NOT complete

Data Representation: Complete Binary Trees

- A complete binary tree can be stored efficiently in a growable array (i.e. vector) by indexing nodes according to level-ordering.
  - The completeness ensures no gaps in the array.
  - We index starting at 1 because it makes some math work out better…
  - To gracefully achieve 1-based indexing with a 0-indexed vector, you can just add a dummy element at position 0 and ignore it.



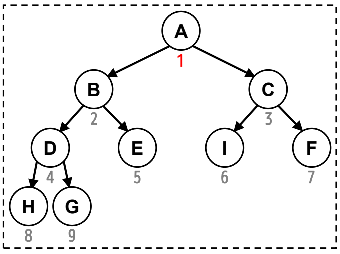
1 2 4

A	B	C	D	E	I	F	H	G
[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]

Notice: root is stored at position [1], not [0]

Inductive Learning: Parent/Child Math

- Given the index of a node in the tree, find formulas for:
  1. The index of its parent
  2. The index of its left child
  3. The index of its right child
  4. Whether that index represents a leaf node?



A	B	C	D	E	I	F	H	G
[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]

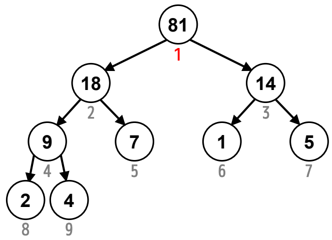
Notice: root is stored at position [1], not [0]

Heap-Ordered Trees, Heaps

Definition: A tree is (max) *heap-ordered* if each node's priority is not greater than the priority of the node's parent

Definition: A *heap* is a set of nodes with keys arranged in a complete heap-ordered tree, represented as an array

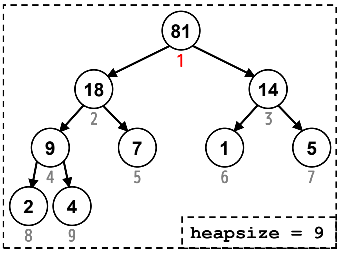
Property: No node in a heap has a key larger than the root's key



Heaps

A *heap* has two crucial properties (representation invariants):
1. Completeness
2. Heap-ordering

We'll leverage these two properties to create an efficient priority queue and an efficient sorting algorithm using a heap!



81	18	14	9	7	1	5	2	4
[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]

Notice: root is stored at position [1], not [0]

Heaps - Executive Summary

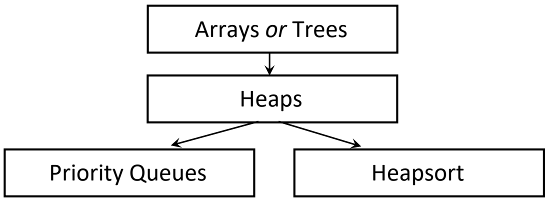
Loose definition: data structure that gives easy access to the most extreme element, e.g., maximum or minimum

"Max Heap": heap with largest element being the "most extreme"

"Min Heap": heap with smallest element being the "most extreme"

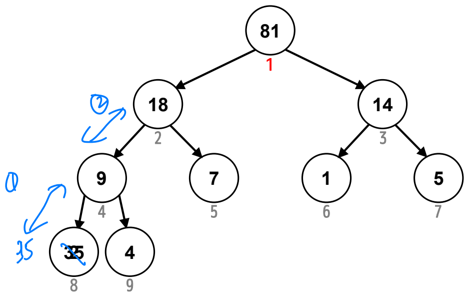
Heaps use complete (binary) trees* as the underlying structure, but are often implemented using arrays

**Do not confuse with binary SEARCH trees*



Example: An Increasing Priority

Increase key at `heap[8]` from 2 to 35



81	18	14	9	7	1	5	~~2~~ 35	4
[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]

Breaking and Fixing a Heap

What if the priority of an item in the heap is **increased**?
→ need to *bottom-up fix*: `fixUp()`

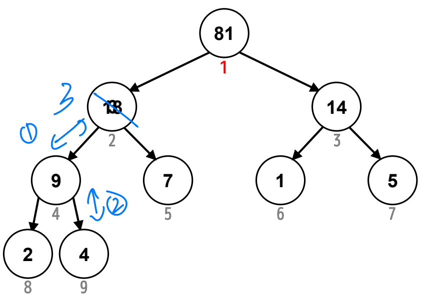
```
void fixUp(Item heap[], int k) {
  while (k > 1 && heap[k / 2] < heap[k]) {
    swap(heap[k], heap[k / 2]);
    k /= 2;  // move up to parent
  } // while
} // fixUp()
```

Note: root is well-known (position 1)

1) Pass index *k* of array element with increased priority
2) Swap the node's key with the parent's key until:
   a. the node has no parent (it is the root), or
   b. the node's parent has a higher (or equal) priority key

Example: A Decreasing Priority

Reduce key at `heap[2]` from 18 to 3



81	~~18~~ 3	14	9	7	1	5	2	4
[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]

Breaking and Fixing a Heap

What if priority of item in heap is **decreased**?
→ need to *top-down fix*: `fixDown()`

```
void fixDown(Item heap[], int heapsize, int k) {
  while (2 * k <= heapsize) {
    int j = 2 * k;  // start with left child
    if (j < heapsize && heap[j] < heap[j + 1]) ++j;
    if (heap[k] >= heap[j]) break;  // heap restored
    swap(heap[k], heap[j]);
    k = j;  // move down
  } // while
} // fixDown()
```

1) Pass index *k* of array element with decreased priority
2) Exchange the key in the given node with the highest priority key among the node's children, moving down until:
   a. the node has no children (leaf node), or
   b. the node has no children with a higher key

Priority Queue (PQ)

Definition: a ***priority queue*** is a data structure that supports three basic operations:

- **Insertion** of a new item `push()`
- **Inspection** of the highest priority item `top()`
- **Removal** of the item with the highest priority `pop()`

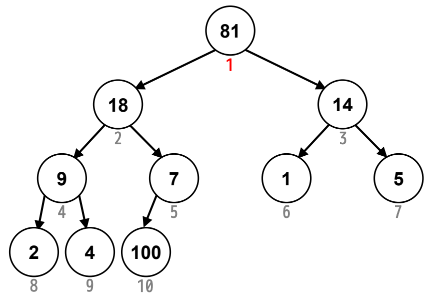
PQ **essential** for upcoming algorithms, e.g., shortest-path, Dijkstra's algorithm

PQ **useful** for past and current (this lecture) algorithms, e.g., heapsort, sorting in reverse order

*Priority queues are often implemented using **heaps** because insertion/removal operations have the same time complexity*

PQ – Insertion

`pq.push(100)`



81	18	14	9	7	1	5	2	4	100
[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]

PQ – Insertion

Insertion operation must maintain **heap invariants**,

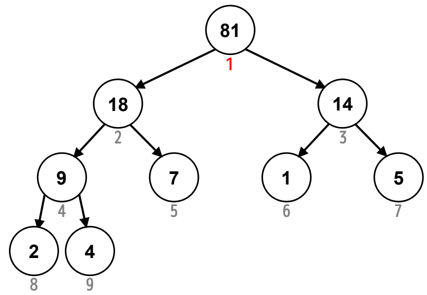
– *"most extreme element must be root"*

```
void push(Item newItem) {
  heap[++heapsize] = newItem;
  fixUp(heap, heapsize);
} // push()
```

1) Insert `newItem` into bottom of tree/heap, i.e., last element
2) `newItem` "bubbles up" tree swapping with parent while parent's key is less (use greater for min-heap)

PQ – Deletion

`pq.pop()`



81	18	14	9	7	1	5	2	4
[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]

PQ - Deletion

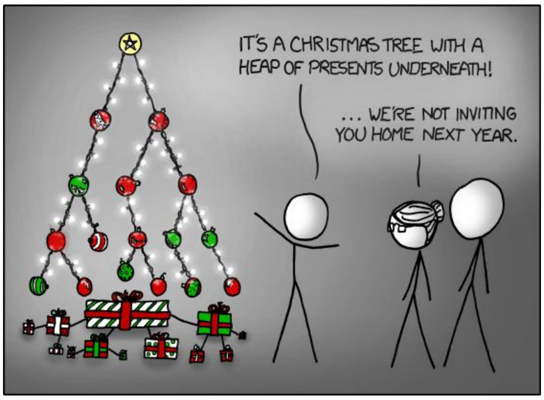
Deletion operation <u>can only remove root</u> and must maintain **heap invariants**

– *"most extreme element must be root"*

```
void pop() {
  heap[1] = heap[heapsize--];
  fixDown(heap, heapsize, 1);
} // pop()
```

1) Remove root element – results in disjoint heap
2) Move the last element into the root position
3) New root "sinks" down the tree swapping with highest priority child whose key is greater (less for min-heap)

http://xkcd.com/835/



"Not only is that terrible in general, but you just KNOW Billy's going to open the root present first, and then everyone will have to wait while the heap is rebuilt."

PQ – Summary

- Priority queue is an ADT
  - Supports insertion, inspection of top item, and removal
- Unordered array PQ
  - **O(1)** insertion of an item
  - ***O(n)*** inspection of top item max pos, front
  - ***O(n)*** removal of top item, <u>or</u> **O(1)** max pos, tail
- Sorted array PQ
  - ***O(n)*** insertion of an item (sort)
  - **O(1)** inspection of top item
  - **O(1)** removal of top item

- Heap
  - Efficient <u>**O(log *n*)** insertion</u> of an item using `fixUp()`
  - **O(1)** inspection of top item
  - Efficient <u>**O(log *n*)** removal</u> of top item using `fixDown()`
  - Must maintain heap property

Building a Heap: **Heapify**

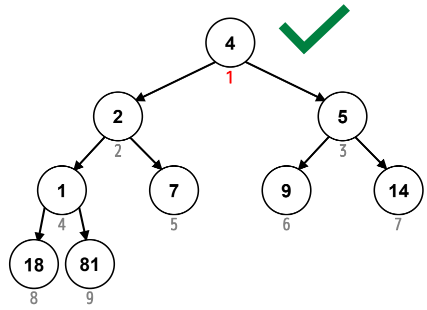
- You could start with an empty vector, and add elements one at a time, keeping the heap property valid after each `push()`
  - Insert *n* elements, ***O*(log *n*)** for each `push()` produces <u>***O*(*n* log *n*)** time</u>
  - Requires an extra vector, or <u>***O*(*n*)** extra memory</u>
- <u>Sort in reverse order: ***O*(*n* log *n*)**; ***O*(1)** memory</u>
- Instead, modify the given array: proceed from bottom to top *or* top to bottom, using `fixDown()` *or* `fixUp()`
  - 4 possibilities; two work and two don't
  - Those that work have different complexities

Building a Heap: **Heapify**

- Possibilities which require no extra array or vector
- Proceed bottom to top, doing repeated `fixDown()`
  - ***O*(*n*)**
  - This is what `std::make_heap()` does from STL
- Proceed bottom to top, doing repeated `fixUp()`
  - Produces invalid heap
  - Try [5, 3, 6, 1, 2, 4, 7]
- Proceed top to bottom, doing repeated `fixDown()`
  - Produces invalid heap
  - Try [1, 2, 3, 4, 5, 6, 7]
- Proceed top to bottom, doing repeated `fixUp()`
  - ***O*(*n* log *n*)**
  - Works but same complexity as a full sort

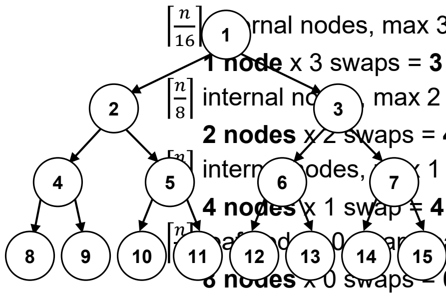
Heapify Exercise

Use bottom-up `fixDown()`: [4, 2, 5, 1, 7, 9, 14, 18, 81]



4	2	5	1	7	9	14	18	81
[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]

Bottom-Up `fixDown()` Complexity



$\lceil \frac{n}{16} \rceil$ internal nodes, max 3 swaps each

1 node x 3 swaps = **3 swaps max**

$\lceil \frac{n}{8} \rceil$ internal nodes, max 2 swaps each

2 nodes x 2 swaps = **4 swaps max**

$\lceil \frac{n}{4} \rceil$ internal nodes, max 1 swap each

4 nodes x 1 swap = **4 swaps max**

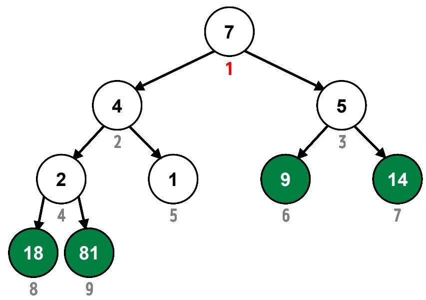
$\lceil \frac{n}{2} \rceil$ leaf nodes, 0 swaps each

8 nodes x 0 swaps = **0 swaps**

15 total nodes ⇒ 11 swaps max

$$O(n)$$

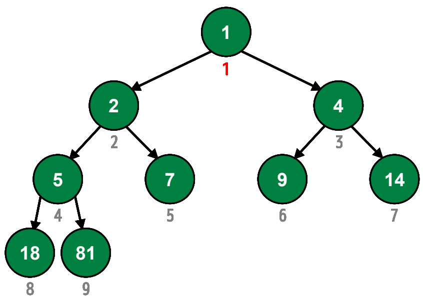
Sorting With a Heap



7	4	5	2	1	9	14	18	81
[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]

1) Swap: 81 ↔ 1
2) Fix-down [1]⋯[8]
3) Swap: 18 ↔ 2
4) Fix-down [1]⋯[7]
5) Swap: 14 ↔ 5
6) Fix-down [1]⋯[6]
7) Swap: 9 ↔ 2
8) Fix-down [1]⋯[5]

Sorting With a Heap



1	2	4	5	7	9	14	18	81
[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]

9) Swap: 7 ↔ 1
10) Fix-down [1]⋯[4]
11) Swap: 18 ↔ 2
12) Fix-down [1]⋯[3]
13) Swap: 14 ↔ 5
14) Fix-down [1]⋯[2]

Heapsort

81	18	14	4	7	9	5	2	1
[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]

Intuition: repeatedly **relocate** the highest-priority element from PQ to the back

Easily implemented as an array; entire sort done **in place**

```
void heapsort(Item heap[], int n) {
  heapify(heap, n);
  for (int i = n; i >= 2; --i) {
    swap(heap[i], heap[1]);
    fixDown(heap, i - 1, 1);
  } // heapsort()
} // heapsort()
```

Root is index 1

Part 1: Transform unsorted array into heap (*heapify*)

Part 2: Remove the highest priority item from heap, add it to sorted sequence, and fix the heap, repeat…

Heapsort Summary

- Take the given *n* elements, convert into a heap
  - Bottom-up `fixDown()`, heapify takes ***O*(*n*)** time
- Remove elements one at a time, filling original array from back to front
  - Swap element at top of heap with last unsorted: ***O*(1)**
  - `fixDown()` to bottom: each takes ***O*(log *n*)** time, *n* of them
- Total runtime: ***O*(*n* log *n*)**
- Total memory: ***O*(1)** or "in place"