Lecture 10
Heaps, Priority Queues, and HeapSort

EECS 281: Data Structures & Algorithms

Wanted: a Container that Supports…

- Creation [empty *or* from existing data]

- Insertion of element
- Deletion of element
- Retrieval/search of element

- Auxiliary/“nice” functionality, e.g., **`isEmpty()`**
- Specialized functionality, e.g., **`getMax()`**

Heaps - Executive Summary

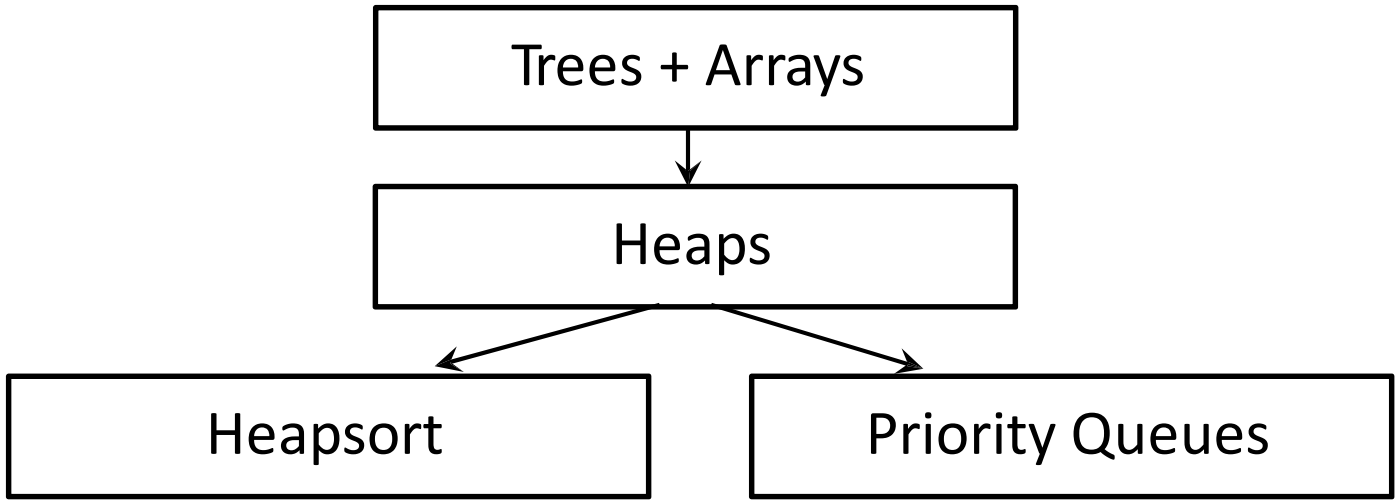
Loose definition: data structure that gives easy access to the most extreme element, e.g., maximum or minimum

Maxheap: heap with maximum element "on top"

Minheap: heap with minimum element "on top"

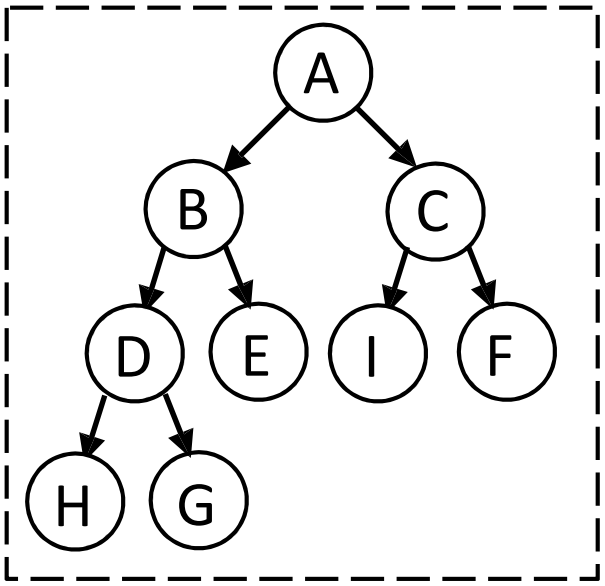
Heaps use complete (binary) trees* as the underlying structure, but are often implemented using arrays

**Do not confuse with binary SEARCH trees*



Tree Science

Undirected tree: (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)



Root: a "selected" node in the tree

A root defines edge directions

Ancestor [of node]: parent of a parent… (closer to root)

Descendent [of node]: child of a child … (further from root)

Internal node: a node with children

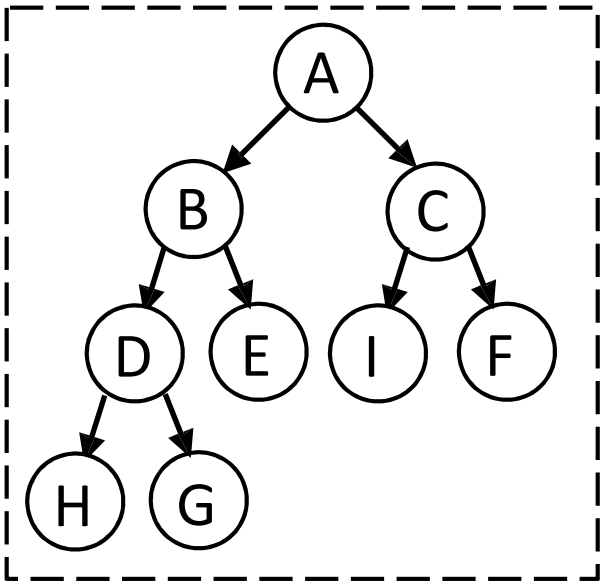
Leaf node: a node without children

(compare to the leaves in undirected trees)

A priori, children's KEY has no relation to parent

Tree Science

Undirected tree: (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)



Depth: depth(root) = 1;

depth(node) = depth(parent) + 1;

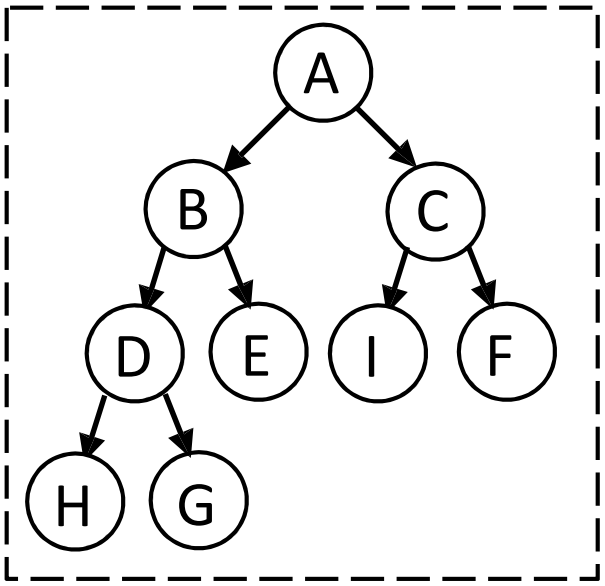
Height: height(leaf) = 1;

height(node) = height(child) + 1;

Max Height/Depth: maximum height or depth of tree's nodes

Tree Science

Undirected tree: (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)



Binary tree: a tree in which every node has 2 or fewer children (node degree ≤ 3)

Complete binary tree: binary tree in which every level, *except possibly the last*, is completely filled, and all nodes are as far left as possible

Concrete Implementation

```
1 template <class Item>
2 struct Node // binary tree node
3    Item item; // data or KEY
4    Node *left; // pointer to left child to
5    Node *right; // pointer right child
6 };
```

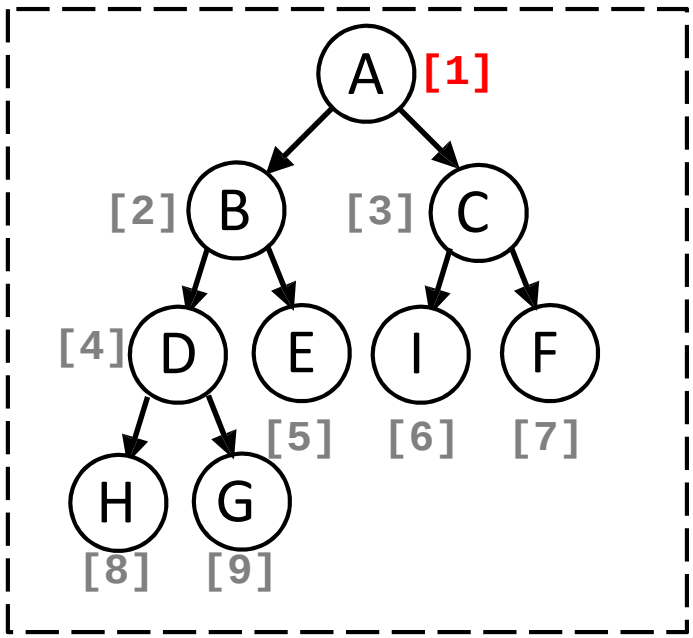
- A node contains some information, and points to its left child node and right child node
  - Can also include a pointer for parent node
  - Can include pointers to 3 children, or a vector of pointers
- Efficient for moving *down* a tree from parent to child
- To avoid pointers, all nodes can be put into an array and accessed by using integer indices
- Another idea: pack a tree into an array in a regular way
  - Similar to how 2D arrays pack into a 1D array

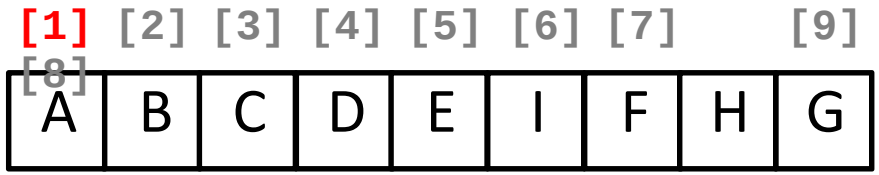
Heap-Ordered Trees, Heaps

Definition: A tree is (max)-*heap-ordered* if the key at each node is not less than the keys of all the node's children

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





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

Ordering Array Elements

Given: an array implementation of a heap and the i^{th} position in the array:

(1) what is the location of *i*'s parent?

(2) what are the locations of *i*'s two children?

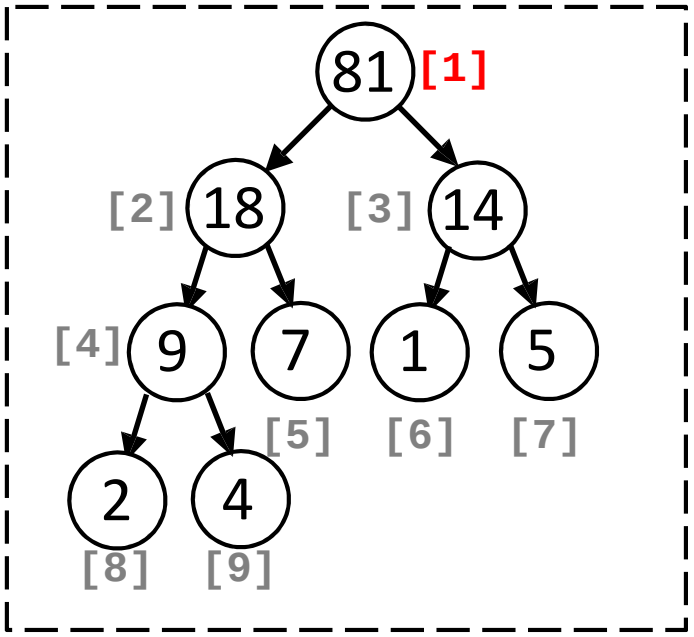
Assume the heap's root is in position 1, not 0.

Array-based Heaps

- Constant **`MAX_HEAP`**
- Data members
  - **`items`**: an array of heap items
  - **`size`**: current number of items in the heap

- Need specified KEY and COMP function
- Store in breadth first order of complete binary tree
- Compact representation for complete tree

Bottom-up Fix Example



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

```
heap[k = 8]: 2 → 35
```

Breaking and Fixing a Heap

What if the priority of an item in the heap is increased?

→ need to *bottom-up fix*: **`fixUp()`**

```
1  void  fixUp(Item  heap[],      int k)  {
2     while  (k  >  1  &&  heap[k  / 2]  <  heap[k])  {
3        swap(heap[k],  heap[k  / 2]);
4        k  /=  2;  //  move up
5     }
6  }
```

Note: root is well-known (position 1)

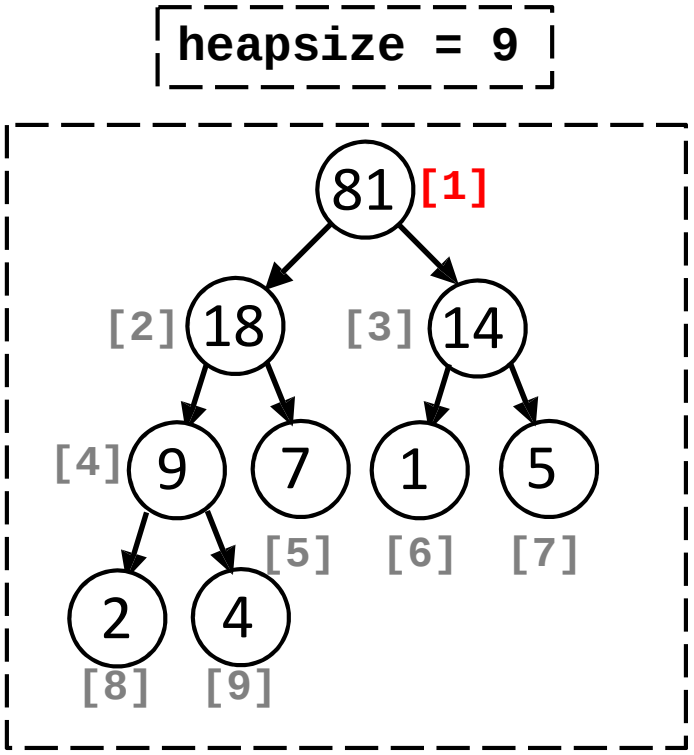
0) Pass index [**k**] of array element with increased priority

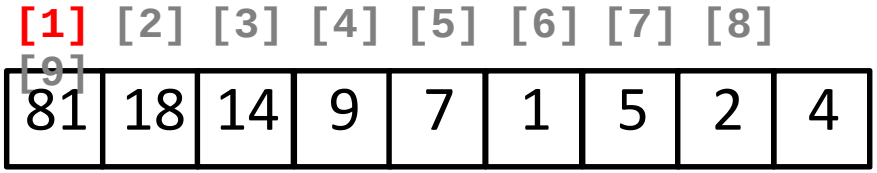
1) Swap the node's key with the parent's key until:

   (*i*) we reach the root, or

   (*ii*) we reach a parent with a larger (or equal) key

Top-down Fix Example





heap[k = 2]: 18 → 3

Breaking and Fixing a Heap

What if priority of item in heap is decreased?

→ need to *top-down fix*: **`fixDown()`**

```
1  void  fixDown(Item  heap[], int  heapsize,  int  k)  {
2     while  (2  *  k  <=  heapsize)  {
3        int j  =  2  *  k;        //  startwith  left  child
4        if (j  <  heapsize  &&  heap[j]  <  heap[j  +  1])  j++;
5        if (heap[k]  >= heap[j])  break;              // heap  restored
6        swap(heap[k],  heap[j]);
7        k  =  j;     // move down
8     }
9  }
```

0) Pass index [**k**] of array element with decreased priority

1) Exchange the key in the given node with the largest key among the node's children, moving down to that child, until:

a) we reach bottom of heap, or *b*) no children have a larger key

Priority Queue (pQ)

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

(*i*) **insertion** of a new item **`enqueue()`**

(*ii*) **removal** of the item with the largest key **`dequeue()`**

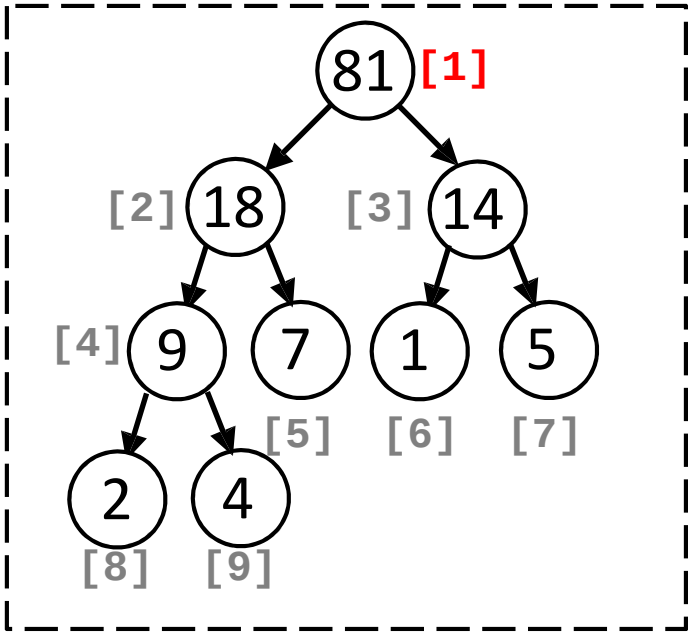
Priority queues **essential** for upcoming algorithms,
e.g., shortest-path, Dijkstra's algorithm

Priority queues **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 are the same

pQ – Insertion exercise

heapsize = 9



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

insert(100)

pQ - Insertion

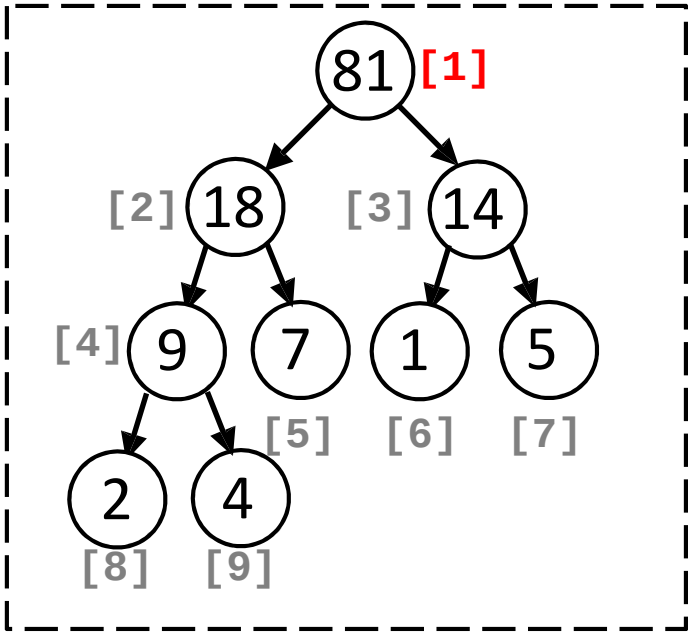
Insertion *operation must maintain* ***heap invariants****,*
e.g., max (or min) element must be root

```
1  void   insert(Item newItem)  {
2      heap[++N]  =  newItem;
3      fixUp(heap,  N);
4  }
```

1) Insert **`newItem`** into bottom of tree/heap, i.e., last element
2) **`newItem`** "trickles up" tree into appropriate spot,
   i.e., swap with parent if parent's key is < (or > for minheap)

pQ – Deletion exercise

heapsize = 9



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

getMax()

pQ - Deletion

Deletion *operation can only remove root, and must maintain* ***heap invariants****, e.g., max (or min) element must be root*

```
1  Item  getMax()  {
2      swap(heap[1],    heap[N]);
3      fixDown(heap,   N  - 1,  1);
4      return heap[N--];
5  }
```

1) Remove root element – results in disjoint heap
2) Move the last element into the root position
3) New root "trickles down" the tree into appropriate spot,
   i.e., swap with child that whose key is > and has larger difference

Exercise

- Describe two algorithms for buildHeap(heap, n)
- What is the complexity?

Heapsort

Intuition: repeatedly **dequeue** the highest-priority element from a pQ

Easily implemented as an array; entire sort can be done in place

```
1  void sortHeap(Item heap[], int n) {
2     buildHeap(heap, n); //calls heapify
3     for (int i = n; i >= 2; --i) {
4        swap(heap[i], heap[1]);
5        fixDown(heap, i - 1, 1);
6     }
7  }
```

Note: root is well-known (position 1)

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

Phase 2: Remove the largest item from heap,
add it to sorted sequence, and fix the heap

Heapsort Summary

- Take the given *n* elements, convert into a heap
  - Bottom-up 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: $O(n \log n)$
  - requires no additional space

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.”

Summary: Tree, Heap, pQ

- Priority queue is an ADT
  - supports insertion and removal
- Unordered array
  - $O(1)$ insertion of an item
  - $O(n)$ removal of largest item
- Heap
  - efficient $O(\log n)$ insertion of an item
  - efficient $O(\log n)$ removal of largest item
- Must be able to maintain heap property
  - bottom-up: fixUp()
  - top-down: fixDown()
- Heapsort
  - $O(n \log n)$ sort that takes advantage of heap properties