CPSC 5031 : Data Structures & Algorithms

Lecture 7: Heap, Heapsort, Priority Queue

(Levitin, Chapter 6.4)

Overview

- What is a heap? Semiheap?
- How does heapsort work?
- Time efficiency of heapsort
- Priority queues

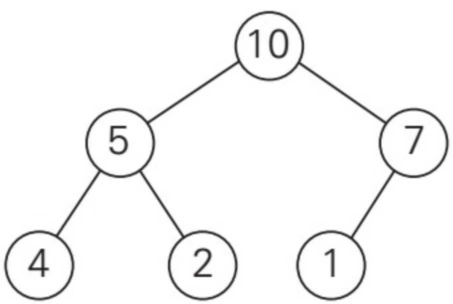
Heaps (max heap)

A **heap** is a binary tree with properties:

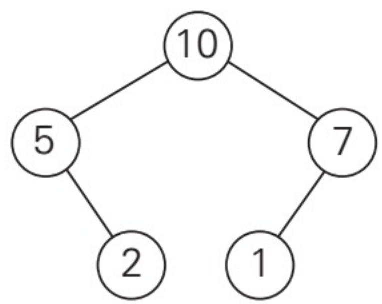
1. It is complete
   - Each level of tree completely filled
   - Except possibly bottom level (nodes in left most positions)
2. It satisfies *heap-order property*
   - Data in each node >= data in children

Heaps

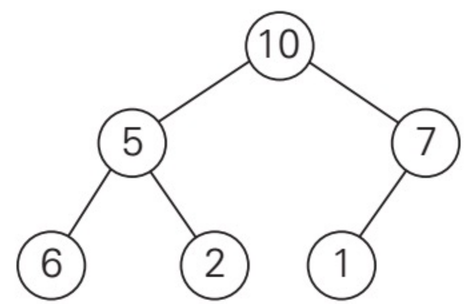
- Which of the following are heaps?



A



B



C

Heaps

- Maxheap?– by default
- Minheap?

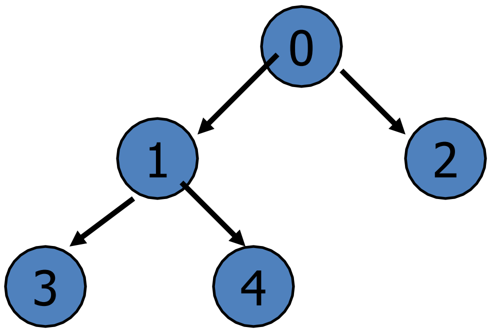
Implementing a Heap

- What data structure is good for its implementation?

Implementing a Heap

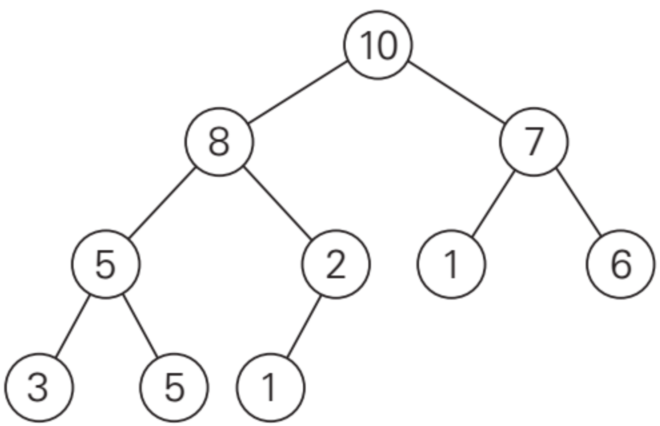
- Use an array or vector, why?
- Number the nodes from top to bottom
  - Number nodes on each row from left to right
- Store data in i^{th} node in i^{th} location of array (vector)

A heap is an array pretending to be a tree, NOT the other way around!



Implementing a Heap

- Note the placement of the nodes in the array



the array representation

index	0	1	2	3	4	5	6	7	8	9	10
value		10	8	7	5	2	1	6	3	5	1

parents | leaves

Implementing a Heap

- Using an array **myArray[]**
  - Children of i^{th} node are at **myArray[2*i+1]** and **myArray[2*i+2]**
  - Parent of the i^{th} node is at **mayArray[(i-1)/2]**

the array representation

index	0	1	2	3	4	5	6	7	8	9	10
value		10	8	7	5	2	1	6	3	5	1

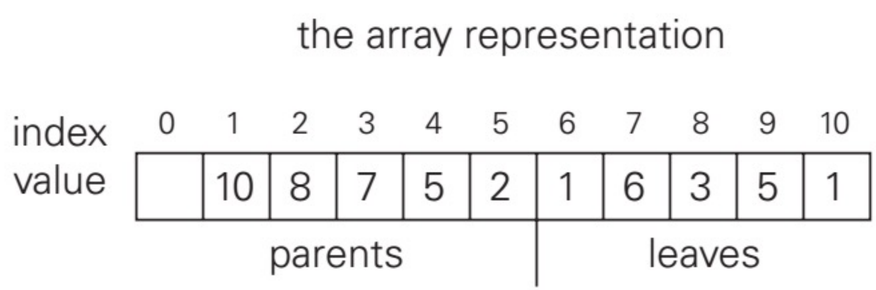
parents | leaves

For size *n*, where is this line drawn?

Compare to BST?

- Isn’t this the same thing as an array BST?

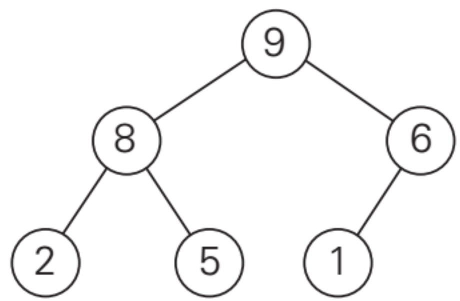
- NO!
  - Look at the ordering of the array below
  - Heaps are good when you need to know the max/first element, *but nothing past that*
  - Hence very good for prioritization (as opposed to sorting)



Basic Heap Operations

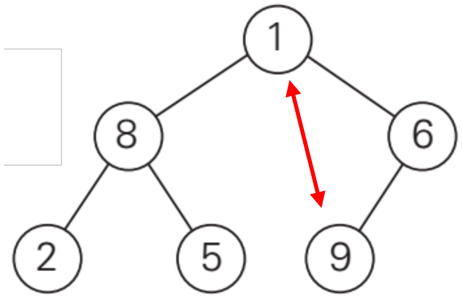
- Constructor
  - Set `mySize` to 0, allocate array
- Empty
  - Check value of `mySize`
- Retrieve the max item
  - Return root of the binary tree, `myArray[0]`
- How about delete max item?
  - Think about it? 🤔

Delete Max Item?



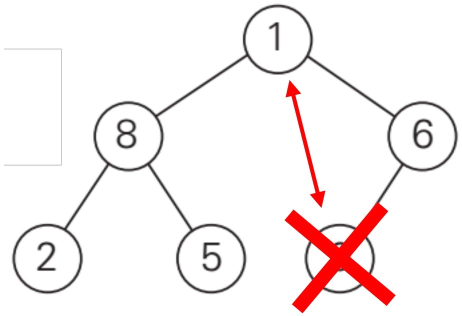
Basic Heap Operations

- Delete max item
  - Max item is the root, replace with last node in tree



Basic Heap Operations

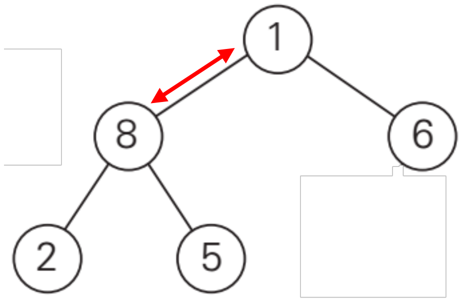
- Delete max item
  - Max item is the root, replace with last node in tree



  - Take the last element

Basic Heap Operations

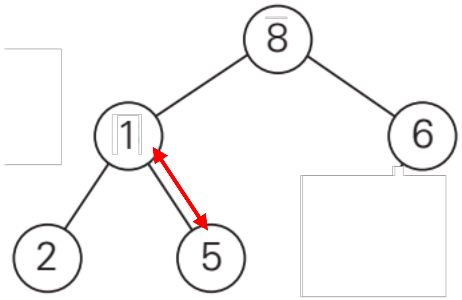
- Delete max item
  - Max item is the root, replace with last node in tree



  - Take the last element
  - Then interchange root with larger of two children
  - Continue this with the resulting sub-tree(s) → percolate down

Basic Heap Operations

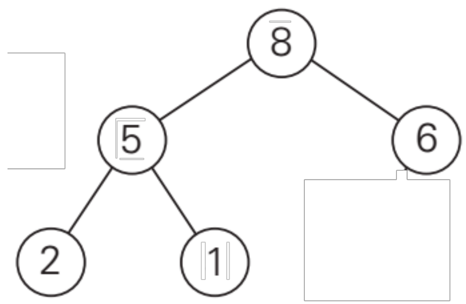
- Delete max item
  - Max item is the root, replace with last node in tree



  - Take the last element
  - Then interchange root with larger of two children
  - Continue this with the resulting sub-tree(s) → percolate down

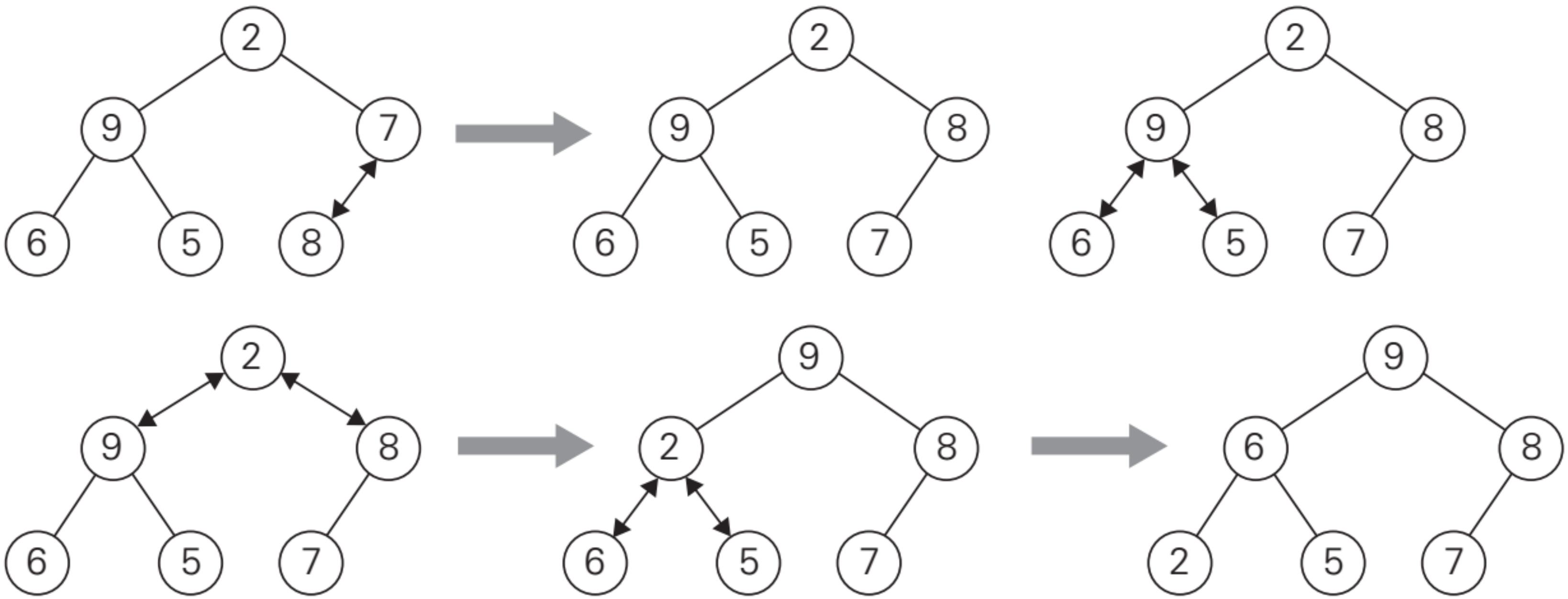
Basic Heap Operations

- Delete max item
  - Max item is the root, replace with last node in tree



  - Take the last element
  - Then interchange root with larger of two children
  - Continue this with the resulting sub-tree(s) → percolate down

Percolate Down Algorithm



Percolate Down Algorithm

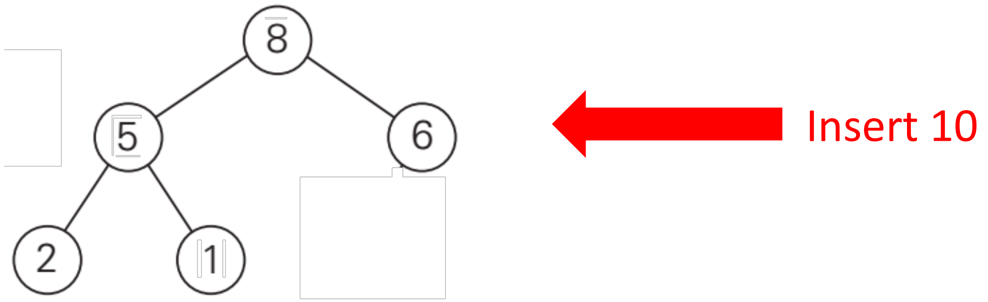
```
//Constructs a heap from elements of a given array
// by the bottom-up algorithm
//Input: An array H[1..n] of orderable items
//Output: A heap H[1..n]
for i ← ⌊n/2⌋ downto 1 do
    k ← i;   v ← H[k]
    heap ← false
    while not heap and 2 ∗ k ≤ n do
        j ← 2 ∗ k
        if j < n   //there are two children
            if H[j] < H[j + 1]  j ← j + 1
        if v ≥ H[j]
            heap ← true
        else H[k] ← H[j];   k ← j
    H[k] ← v
```

Insert an item into the heap

- Anybody have an idea?

Basic Heap Operations

- Insert an item
  - Amounts to a percolate up routine
  - Place new item at end of array



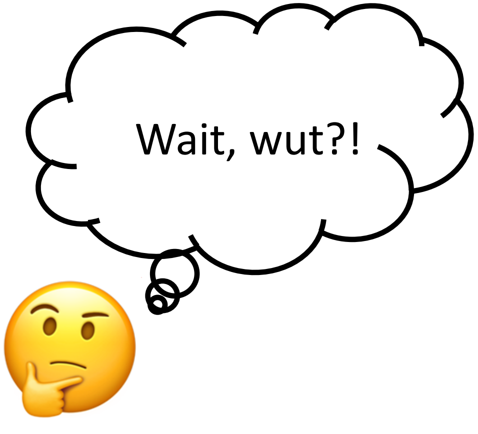
  - Interchange with parent so long as it is greater than its parent

Percolate Up Algorithm

- Why percolate up?
  - When to terminate the up process?
- void Heap::percolate_up()
- void Heap::insert(const T& item)

Heapsort

- Two-stage algorithm.
- Given a list of integers…
  - “Heapify” the list
  - Then delete them all.



Heapsort - Heapify

Stage 1 (heap construction)

2	9	**7**	6	5	8
2	**9**	8	6	5	7
2	9	8	6	5	7
9	**2**	8	6	5	7
9	6	8	2	5	7

(Hint: it helps to draw the line showing where leaves are…)

Heapsort – "Delete" all nodes

Stage 2 (maximum deletions)

9	6	8	2	5	7		
7	6	8	2	5	|	**9**	
8	6	7	2	5			
5	6	7	2	|	**8**		
7	6	5	2				
2	6	5	|	**7**			
6	2	5					
5	2	|	**6**				
5	2						
2	|	**5**					
2							

(Remember: deleting just moves the root to the very end!)

Summary of HeapSort

- Given an array…
- Stage 1: Heapify this array into a heap (percolate up)
- Stage 2: Exchange the root node with the last element and shrink the list by pruning the last element (percolate down). Repeat on |n-1| array.

Heapsort complexity

- $T(n) = O(n \log n)$
  - Stage 1: $O(n)$
  - Stage 2: $C(n)=2n - 2 \log n$, so $O(n \log n)$

- $O(1)$ additional space requirement!

Priority Queue

- A collection of data elements…
  - Items stored in order by priority
  - Higher priority items removed ahead of lower
  - Do we care about sort order other than priority?
- Operations
  - Constructor
  - Insert
  - Find, remove smallest/largest (priority) element
  - Replace
  - Change priority
  - Delete an item
  - Join two priority queues into a larger one

"Multi-set" vs.
strictly ordered array

Priority Queue

- Implementation possibilities
  - As a list (array, vector, linked list)
    - T(n) for search, removeMax, insert operations?
  - As an ordered list
    - T(n) for search, removeMax, insert operations?
  - Best is to use a heap
    - T(n) for basic operations?
    - Why?

Should we use a PQ?

Given a list of m words, I'd like to query for the nth most frequent word(s). (e.g. Given 1000 words, tell me the 5th most frequent word(s), or the 10th most frequent word(s) etc). Would like to query multiple times for different n's.

An approach…

1. Count word freq
2. Insert into a max-heap
3. Delete n-1 times
4. Take the top!

…but…

Priority Queues are heaps
…and PQ delete is destructive!

In C#

```
var freq = rawWords.Split(' ')
    .GroupBy(w=>w)
    .ToDictionary(w=>w.Key,w=>w.Count())
    .ToLookup(kvp=>kvp.Value, kvp=>kvp.Key)
    .OrderByDescending(kvp=>kvp.Key)
    .ToList();

var fifth = freq[4].Value;
```

…terrible perf, but short code…? 🤔