CSE 332 Summer 2024
Lecture 11: Hashing 3

Nathan Brunelle

http://www.cs.uw.edu/332

Next topic: Hash Tables

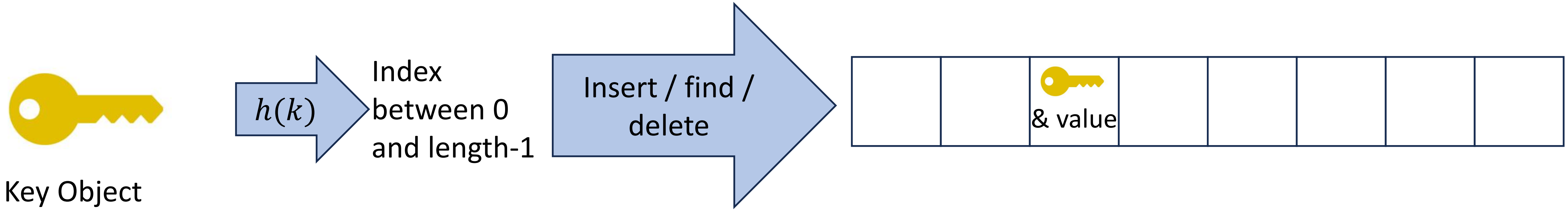
Data Structure	Time to insert	Time to find	Time to delete
Unsorted Array	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
Unsorted Linked List	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
Sorted Array	$\Theta(n)$	$\Theta(\log n)$	$\Theta(n)$
Sorted Linked List	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
Binary Search Tree	$\Theta(\text{height})$	$\Theta(\text{height})$	$\Theta(\text{height})$
AVL Tree	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$
Hash Table (Worst case)	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
Hash Table (Expected and amortized)	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$

Dictionary (Map) ADT

- Contents:
  - Sets of key+value pairs
  - ~~Keys must be comparable~~
- Operations:
  - insert(key, value)
    - Adds the (key,value) pair into the dictionary
    - If the key already has a value, overwrite the old value
      - Consequence: Keys cannot be repeated
  - find(key)
    - Returns the value associated with the given key
  - delete(key)
    - Remove the key (and its associated value)

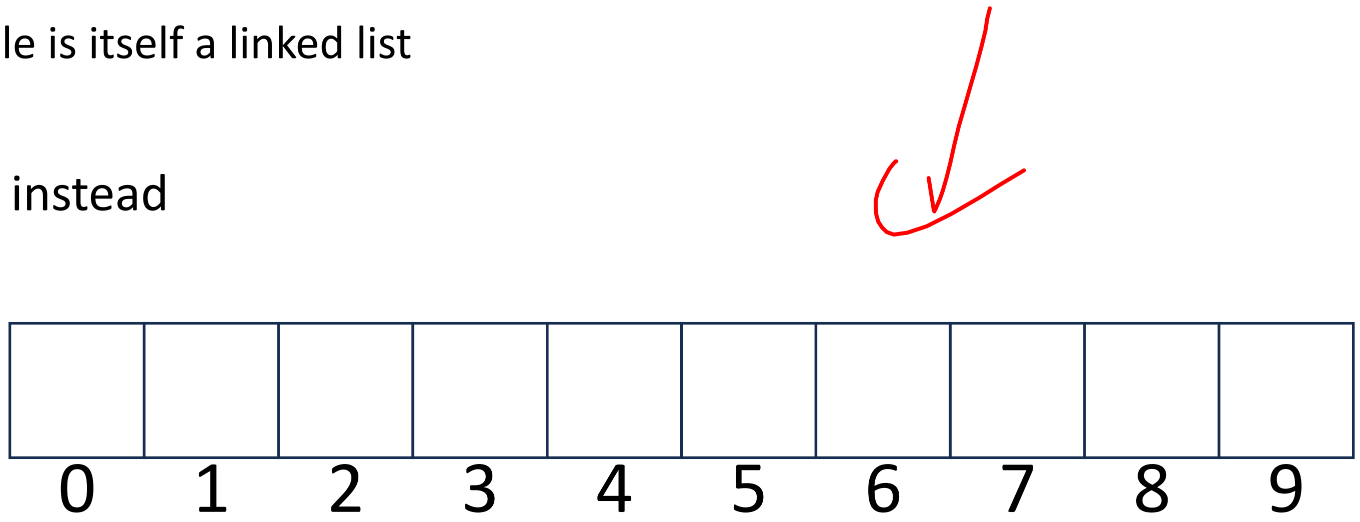
Hash Tables

- Idea:
  - Have a small array to store information
  - Use a **hash function** to convert the key into an index
    - Hash function should "scatter" the keys, behave as if it randomly assigned keys to indices
  - Store key at the index given by the hash function
  - Do something if two keys map to the same place (should be very rare)
    - Collision resolution



Collision Resolution

- A Collision occurs when we want to insert something into an already-occupied position in the hash table
- 2 main strategies:
  - Separate Chaining
    - Use a secondary data structure to contain the items
      - E.g. each index in the hash table is itself a linked list
  - Open Addressing
    - Use a different spot in the table instead
      - Linear Probing
      - Quadratic Probing
      - Double Hashing



Rehashing

- If your load factor λ gets too large, copy everything over to a larger hash table
  - To do this: make a new, larger array
  - Re-insert all items into the new hash table by reapplying the hash function
    - We need to reapply the hash function because items should map to a different index
  - New array should be "roughly" double the length (but probably still want it to be prime)
- What does "too large" mean?
  - For separate chaining, typically we want $\lambda < 2$
  - For open addressing, typically we want $\lambda < \frac{1}{2}$

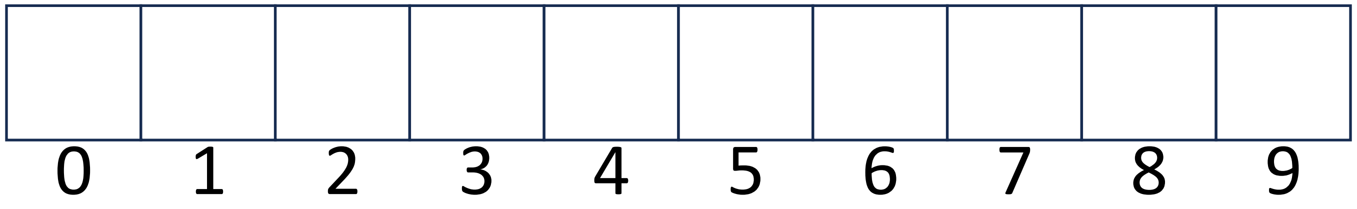
Linear Probing: Insert Procedure

- To insert k, v
  - Calculate $i = h(k)\ \%\ length$
  - If $table[i]$ is occupied then try $(i+1)\%\ length$
  - If that is occupied try $(i+2)\%\ length$
  - If that is occupied try $(i+3)\%\ length$
  - …
  - $h(k) = k\%10$

		2	32	24	62	54			
0	1	2	3	4	5	6	7	8	9

Linear Probing: Find

- To find key k
  - Calculate $i = h(k)\ \%\ length$
  - If $table[i]$ is occupied and does not contain k then look at $(i + 1)\ \%\ length$
  - If that is occupied and does not contain k then look at $(i + 2)\ \%\ length$
  - If that is occupied and does not contain k then look at $(i + 3)\ \%\ length$
  - Repeat until you either find k or else you reach an empty cell in the table



Linear Probing: Delete

- To delete key k, where $h(k) = i$
  - Assume it is present
- Beginning at index i, probe until we find k (call this location index j)
- Mark j as empty (e.g. null), then continue probing while doing the following until you find another empty index
  - If you come across a key which hashes to a value $\leq j$ then move that item to index j and update j.

0	1	2	3	4	5	6	7	8	9

Linear Probing: Delete

- Option 1: Fill in with items that hashed to before the empty slot
- Option 2: "Tombstone" deletion. Leave a special object that indicates an object was deleted from there
  - The tombstone does not act as an open space when finding (so keep looking after its reached)
  - When inserting you can replace a tombstone with a new item

k, v		k, v	k, v		k, v				
0	1	2	3	4	5	6	7	8	9

Downsides of Linear Probing

- What happens when λ approaches 1?
  - Get longer and longer contiguous blocks
  - A collision is guaranteed to grow a block
    - Larger blocks experience more collisions
    - Feedback loop!
- What happens when λ exceeds 1?
  - Impossible!
  - You can't insert more stuff

Quadratic Probing: Insert Procedure

- To insert k, v
  - Calculate $i = h(k)\ \%\ size$
  - If $table[i]$ is occupied then try $(i + 1^2)\%\ size$
  - If that is occupied try $(i + 2^2)\%\ size$
  - If that is occupied try $(i + 3^2)\%\ size$
  - If that is occupied try $(i + 4^2)\%\ size$
  - …

4%10

14

44

24

				14	44			24	
0	1	2	3	4	5	6	7	8	9

Quadratic Probing: Example

- Use $i = k\%7$
- Insert:
  - 76 -> 6; insert at index 6
  - 40 -> 5; insert at index 5
  - 48 -> 6; 6+1^2; insert at index 0
  - 5 -> 5; 5+2^2; insert at indx 2
  - 55 -> 6; 6+2^2; insert at index 3
  - 47 -> 5; 5+5^2

48		5	55		40	76
0	1	2	3	4	5	6

Using Quadratic Probing

- If you probe $tablesize$ times, you start repeating the same indices
- If $tablesize$ is prime and $\lambda < \frac{1}{2}$ then you're guaranteed to find an open spot in at most $tablesize/2$ probes
- Helps with the clustering problem of linear probing, but does not help if many things hash to the same value

Double Hashing: Insert Procedure

- Given h and g are both good hash functions
- To insert k, v
  - Calculate $i = h(k)\ \%\ size$
  - If $table[i]$ is occupied then try $\left(i + g(k)\right)\ \%\ size$
  - If that is occupied try $\left(i + 2 \cdot g(k)\right)\%\ size$
  - If that is occupied try $\left(i + 3 \cdot g(k)\right)\%\ size$
  - If that is occupied try $\left(i + 4 \cdot g(k)\right)\%\ size$
  - …

0	1	2	3	4	5	6	7	8	9

Double Hashing: Insert Procedure

- Given h and g are both good hash functions
  - $h(k) = k\%10$
  - $g(k) = k\%7$
- $g(k)\%len = 0$
- If $g(k)\%len == 0$ then add 1 to $g(k)$

			13		23			33	
0	1	2	3	4	5	6	7	8	9

Hash tables

- Can use hash functions with “good” properties to make it so a small array has similar performance to a huge one
  - The function should randomly scatter keys
- Because the array is small, it’s possible for collisions to occur
  - Sep. chaining
    - Insert k,v pairs into another DS
  - Open addressing
    - Insert k,v pairs directly into indices
    - Use probing strategy when collisions occur
      - Linear
      - Quad
      - Double hashing
- Need to watch our load factor, and rehash when it gets to big
  - SC: <2
  - OA: <1/2

Sorting

- Rearrangement of items into some defined sequence
  - Usually: reordering a list from smallest to largest according to some metric
- Why sort things?

More Formal Definition

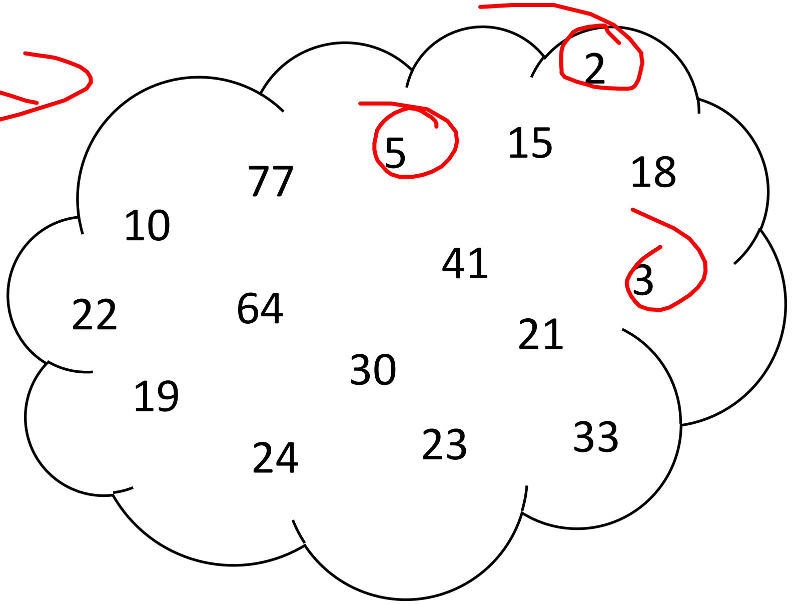
- Input:
  - An array A of items
  - A comparison function for these items
    - Given two items x and y, we can determine whether $x < y$, $x > y$, or $x = y$
- Output:
  - A permutation of A such that if $i \leq j$ then $A[i] \leq A[j]$
  - Permutation: a sequence of the same items but perhaps in a different order

Sorting “Landscape”

- There is no singular best algorithm for sorting
- Some are faster, some are slower
- Some use more memory, some use less
- Some are super extra fast if your data matches particular assumptions
- Some have other special properties that make them valuable
- No sorting algorithm can have only all the “best” attributes

$$n + n-1 + n-2 + n-3 + \dots + 1 = \frac{n(n+1)}{2}$$

"Moving Day" Sorting Algorithm



2	3	5													
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Selection Sort

In-place

- Idea: Find the next smallest element, swap it into the next index in the array

Already In Position

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

Already In Position

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

Selection Sort

- Swap the thing at index 0 with the smallest thing in the array
- Swap the thing at index 1 with the smallest thing after index 0
- …
- Swap the thing at index i with the smallest thing after index $i-1$

```
for (i=0; i<a.length; i++){
	smallest = i;
	for (j=i; j<a.length; j++){
		if (a[j]<a[smallest]){ smallest=j;}
	}
	temp = a[i];
	a[i] = a[smallest];
	a[smallest] = a[i];
}
```

Running Time:

Worst Case: $\Theta(\cdot)$ n^2

Best Case: $\Theta(\cdot)$ n^2

10	77	5	15	2	22	64	41	18	19	30	21	3	24	23	33
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Insertion Sort

- Idea: Maintain a sorted list prefix, extend that prefix by "inserting" the next element

Sorted Prefix

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

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

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

Sorted Prefix

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

Insertion Sort

- If the items at index 0 and 1 are out of order, swap them
- Keep swapping the item at index 2 with the thing to its left as long as the left thing is larger
- …
- Keep swapping the item at index i with the thing to its left as long as the left thing is larger

```
for (i=1; i<a.length; i++){
	prev = i-1;
	while(a[i] < a[prev] && prev > -1){
		temp = a[i];
		a[i] = a[prev];
		a[prev] = a[i];
		i--;
		prev--;
	}
}
```

Running Time:

Worst Case: $\Theta(\cdot)$

Best Case: $\Theta(\cdot)$

10	77	5	15	2	22	64	41	18	19	30	21	3	24	23	33
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Aside: Bubble Sort – we won’t cover it

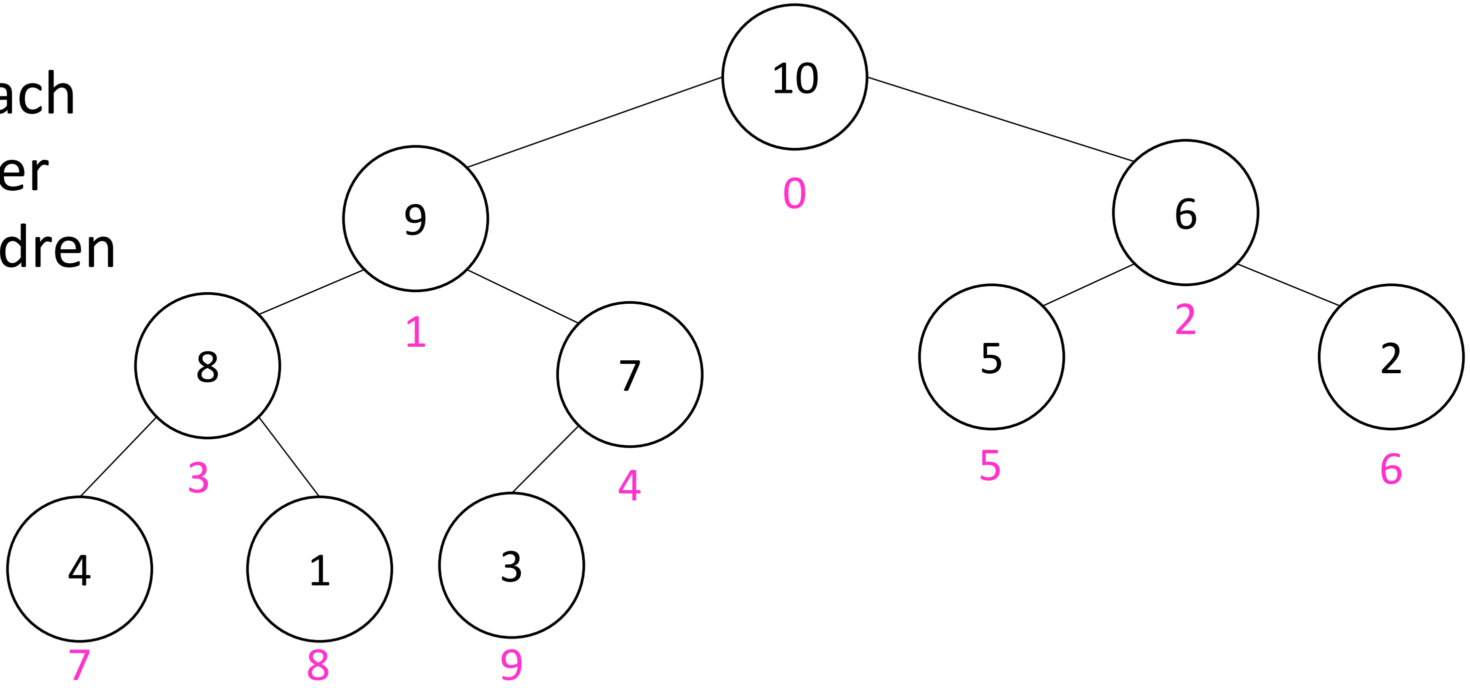
"the bubble sort seems to have nothing to recommend it, except a catchy name and the fact that it leads to some interesting theoretical problems” –Donald Knuth, The Art of Computer Programming

Heap Sort

- Idea: Build a maxHeap, repeatedly delete the max element from the heap to build sorted list Right-to-Left

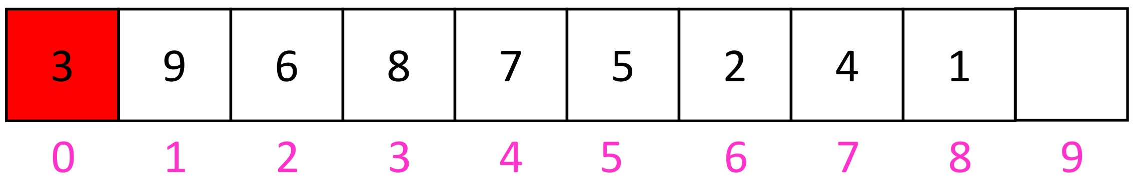
10	9	6	8	7	5	2	4	1	3
0	1	2	3	4	5	6	7	8	9

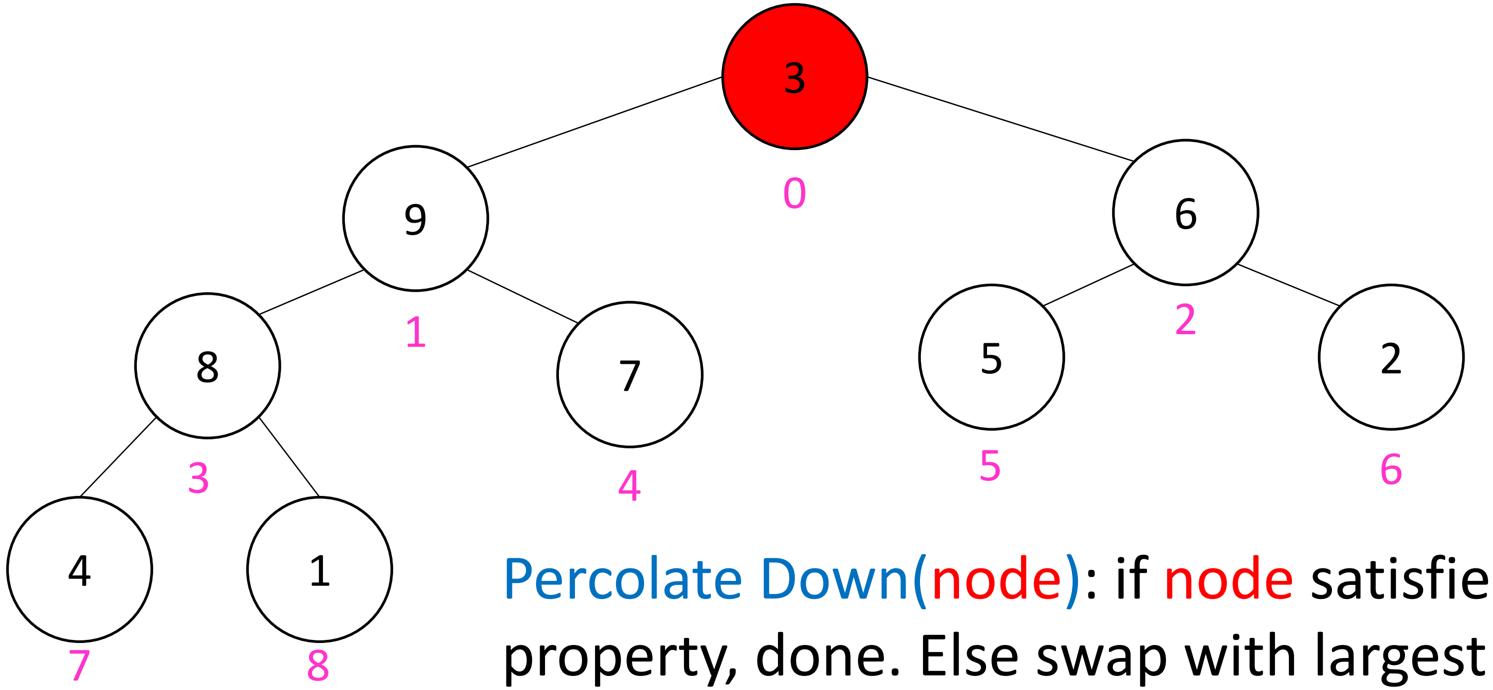
Max Heap Property: Each node is larger than its children



Heap Sort

- Remove the Max element (i.e. the root) from the Heap: replace with last element, call percolateDown(root)



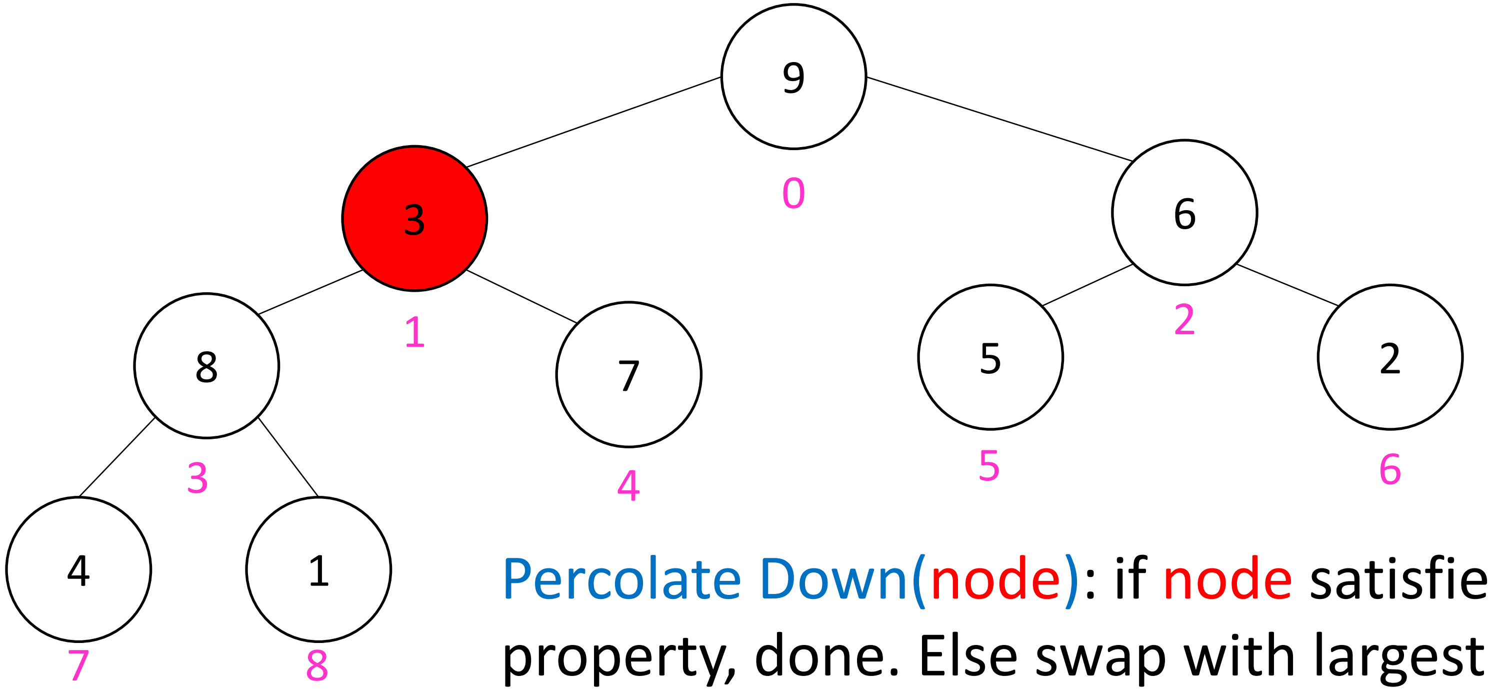


Percolate Down(node): if node satisfies heap property, done. Else swap with largest child and repeat on that subtree

Heap Sort

- Remove the Max element (i.e. the root) from the Heap: replace with last element, call percolateDown(root)

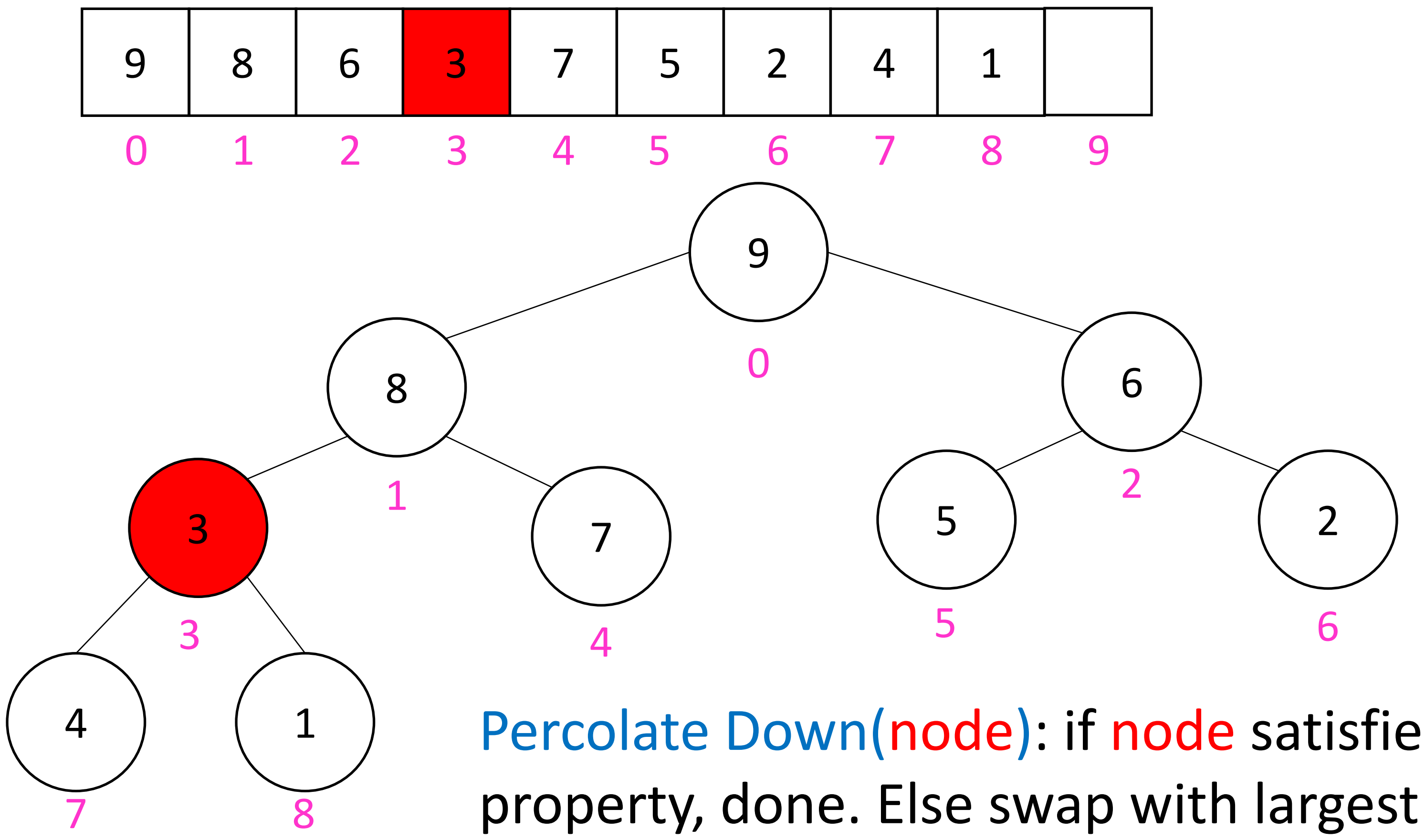
9	3	6	8	7	5	2	4	1	
0	1	2	3	4	5	6	7	8	9



Percolate Down(node): if node satisfies heap property, done. Else swap with largest child and repeat on that subtree

Heap Sort

- Remove the Max element (i.e. the root) from the Heap: replace with last element, call percolateDown(root)

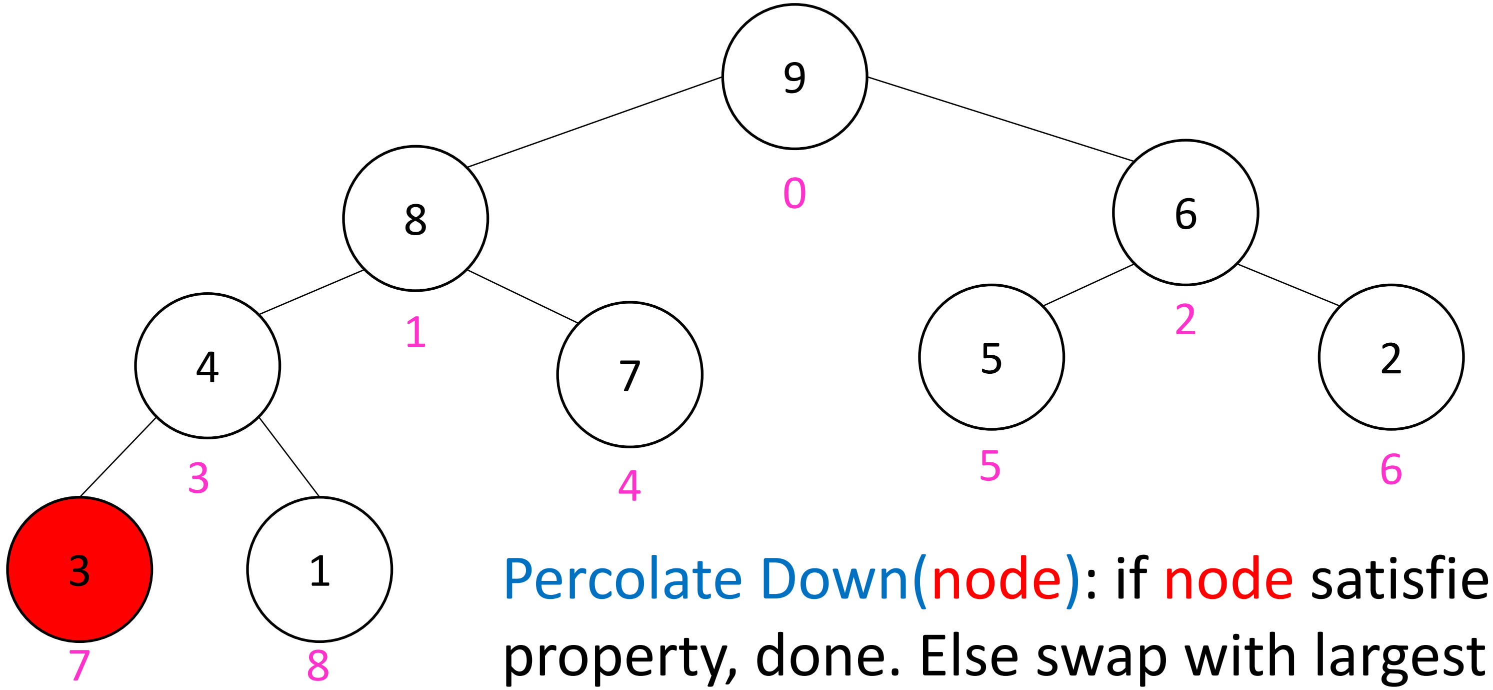


Percolate Down(node): if node satisfies heap property, done. Else swap with largest child and repeat on that subtree

Heap Sort

- Remove the Max element (i.e. the root) from the Heap: replace with last element, call percolateDown(root)

9	8	6	4	7	5	2	3	1	
0	1	2	3	4	5	6	7	8	9



Percolate Down(node): if node satisfies heap property, done. Else swap with largest child and repeat on that subtree

Heap Sort

- Build a heap
- Call deleteMax
- Put that at the end of the array

```
myHeap = buildHeap(a);
for (int i = a.length-1; i>=0; i--){
	item = myHeap.deleteMax();
	a[i] = item;
}
```

Running Time:

Worst Case: $\Theta(\cdot)$

Best Case: $\Theta(\cdot)$

“In Place” Sorting Algorithm

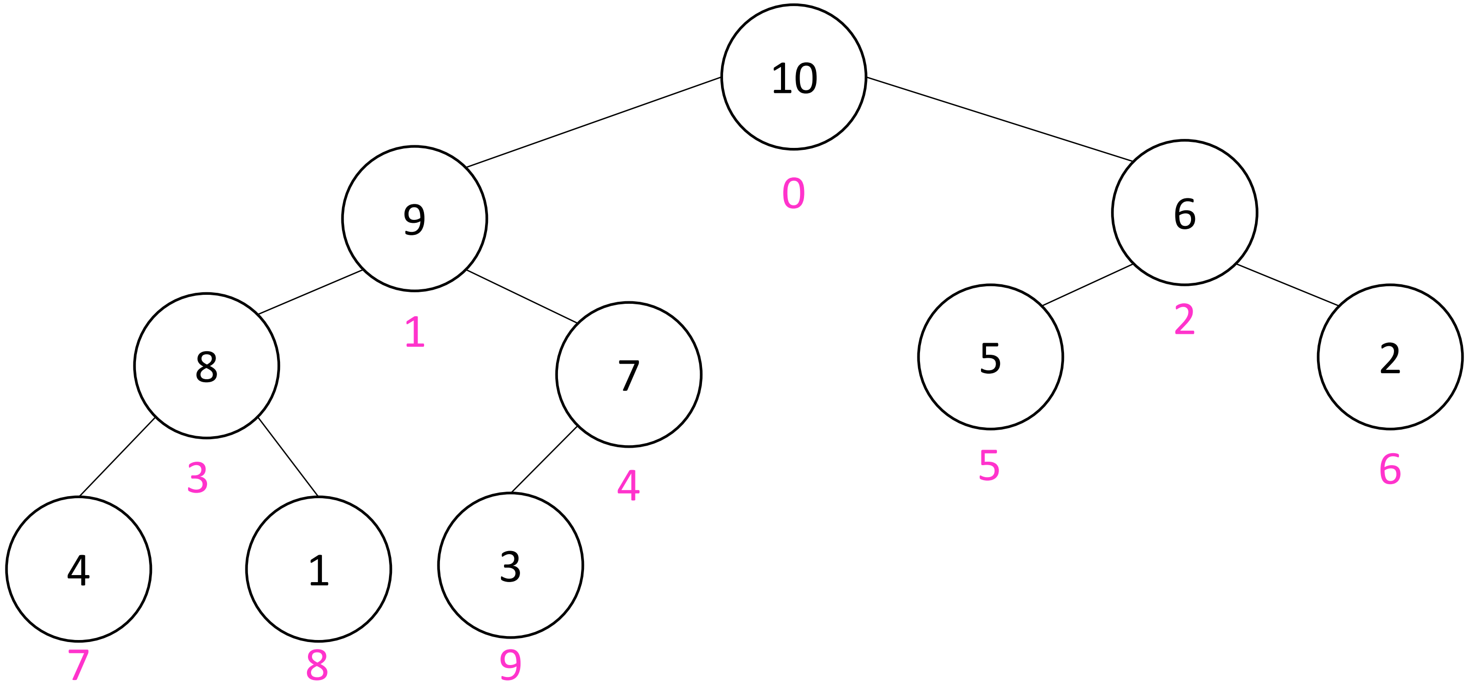
- A sorting algorithm which requires no extra data structures
- Idea: It sorts items just by swapping things in the same array given
- Definition: it only uses $\Theta(1)$ extra space

- Selection sort: In Place!
- Insertion sort: In Place!
- Heap sort: Not In Place!
  - But we can fix that!

In Place Heap Sort

- Idea: When "removing" an element from the heap, swap it with the last item of the heap then "pretend" the heap is one item shorter

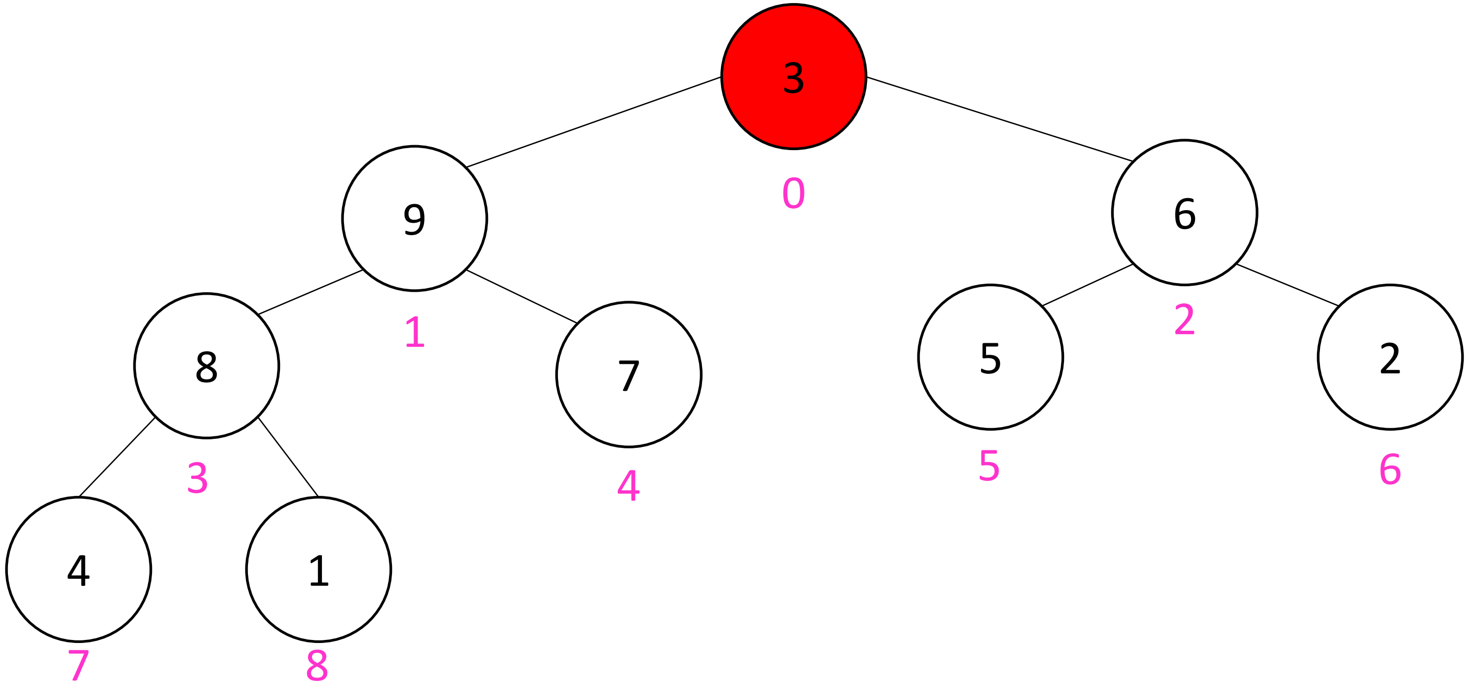
10	9	6	8	7	5	2	4	1	3
0	1	2	3	4	5	6	7	8	9



Heap Sort

- Idea: When “removing” an element from the heap, swap it with the last item of the heap then “pretend” the heap is one item shorter

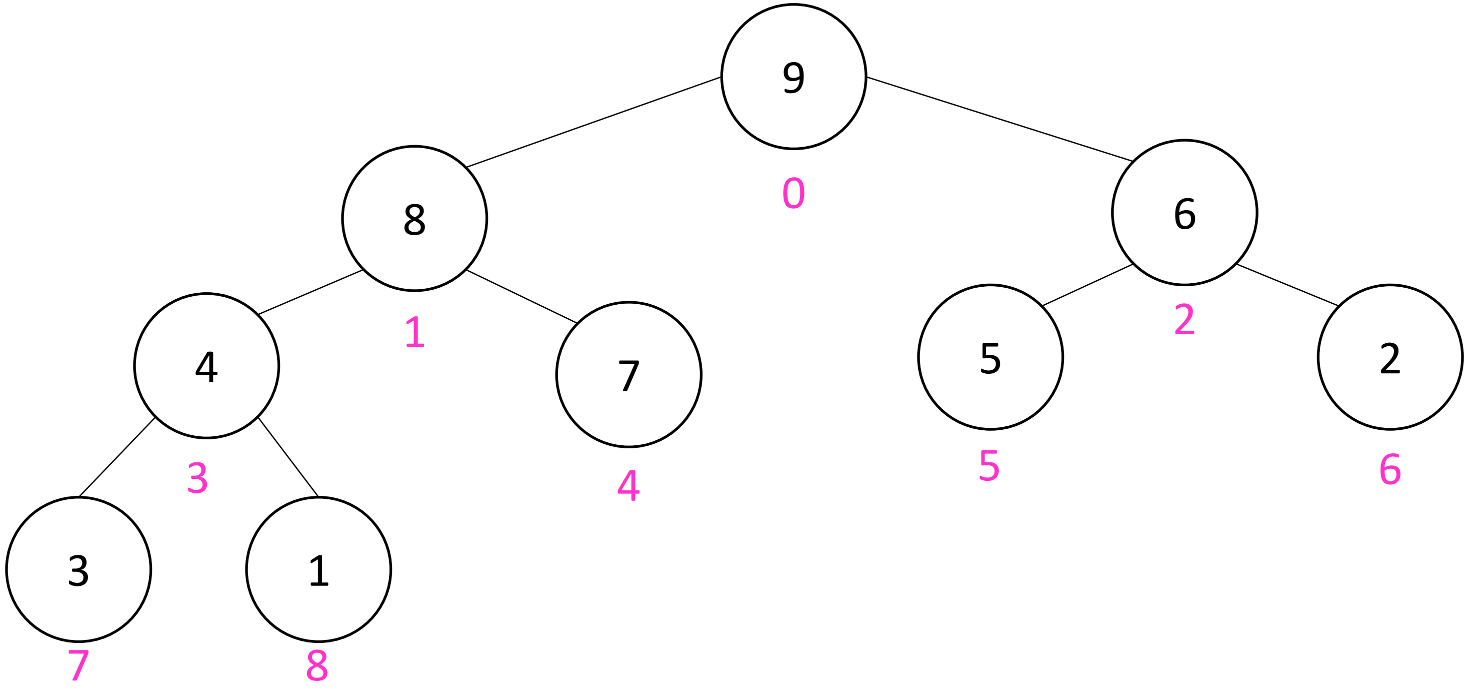
3	9	6	8	7	5	2	4	1	10
0	1	2	3	4	5	6	7	8	9



Heap Sort

- Idea: When “removing” an element from the heap, swap it with the last item of the heap then “pretend” the heap is one item shorter

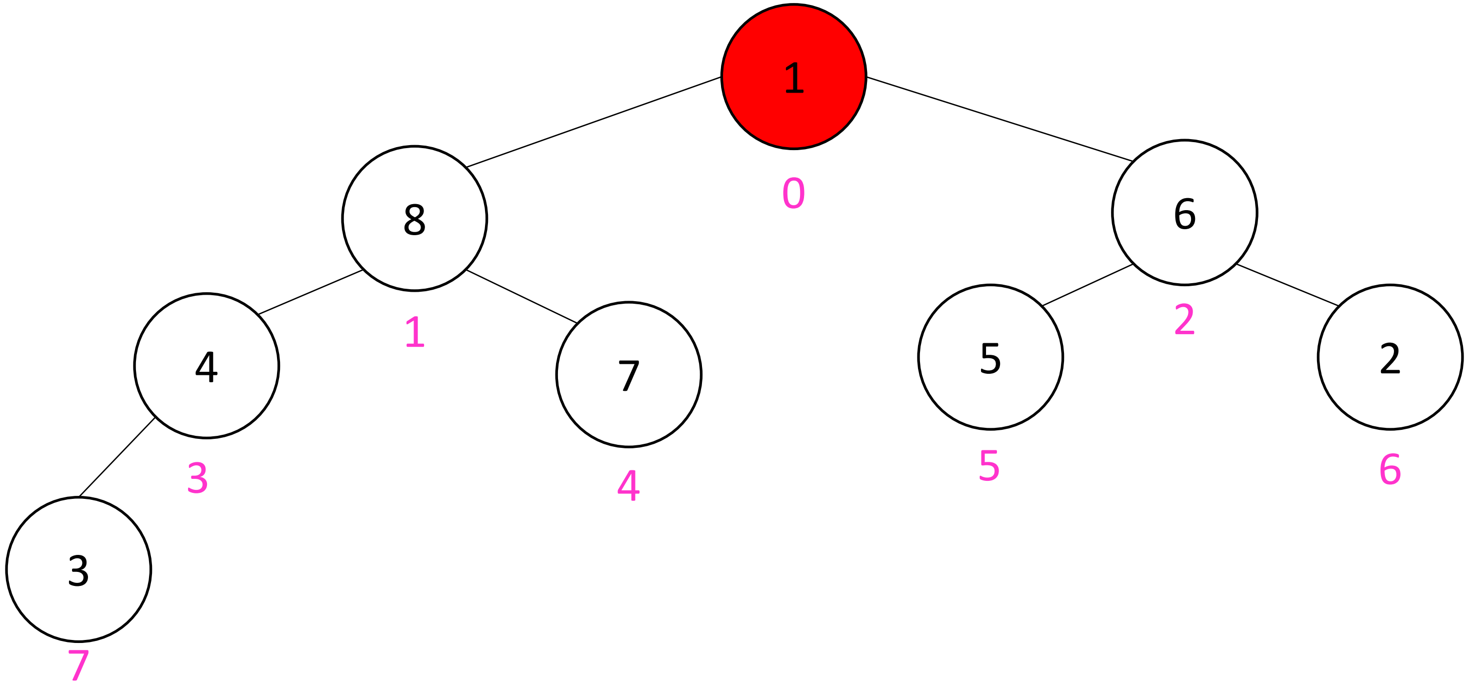
9	8	6	4	7	5	2	3	1	10
0	1	2	3	4	5	6	7	8	9



Heap Sort

- Idea: When "removing" an element from the heap, swap it with the last item of the heap then "pretend" the heap is one item shorter

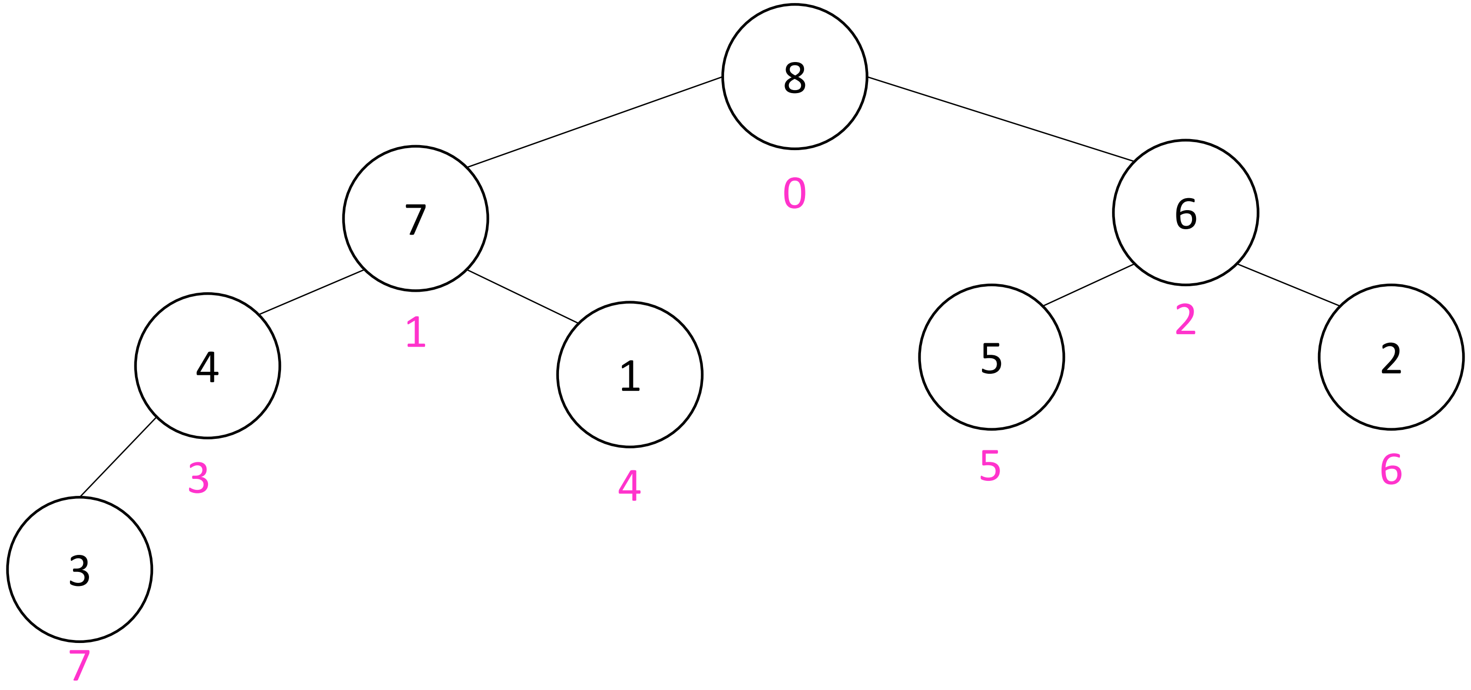
1	8	6	4	7	5	2	3	9	10
0	1	2	3	4	5	6	7	8	9



Heap Sort

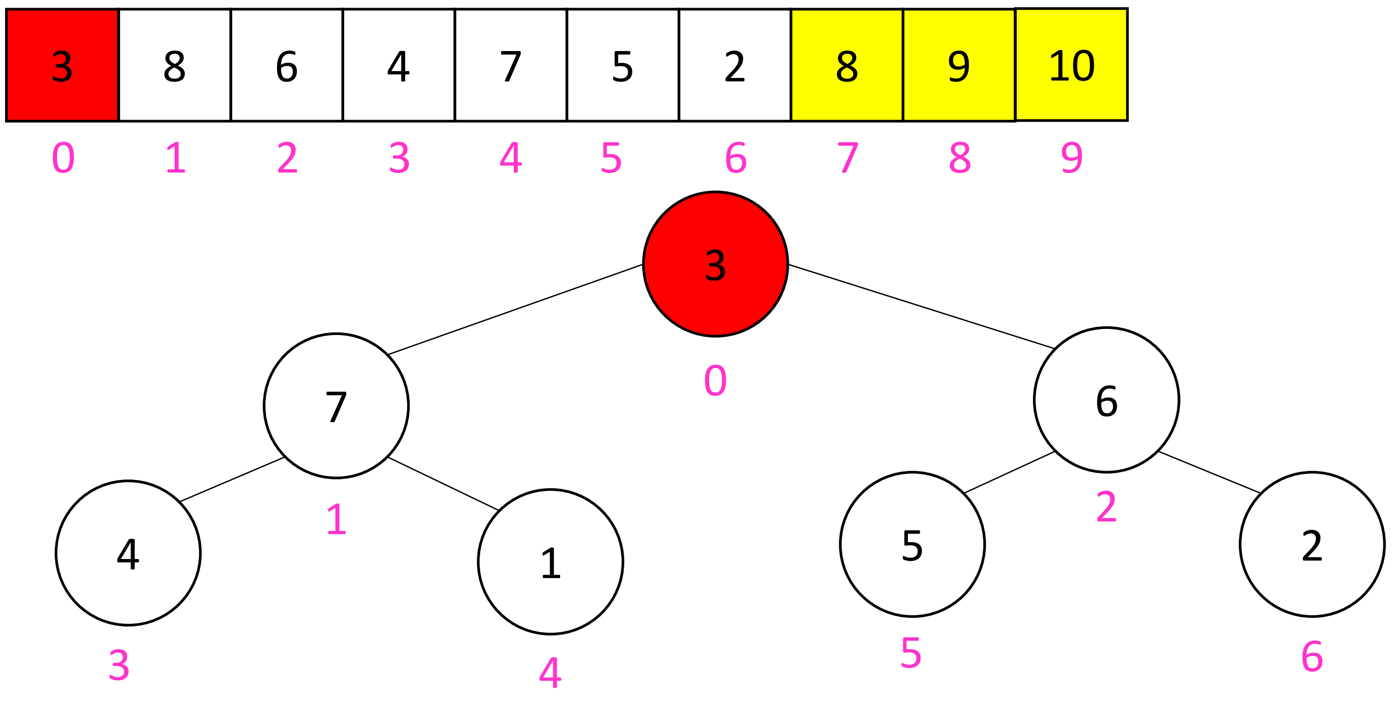
- Idea: When “removing” an element from the heap, swap it with the last item of the heap then “pretend” the heap is one item shorter

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



Heap Sort

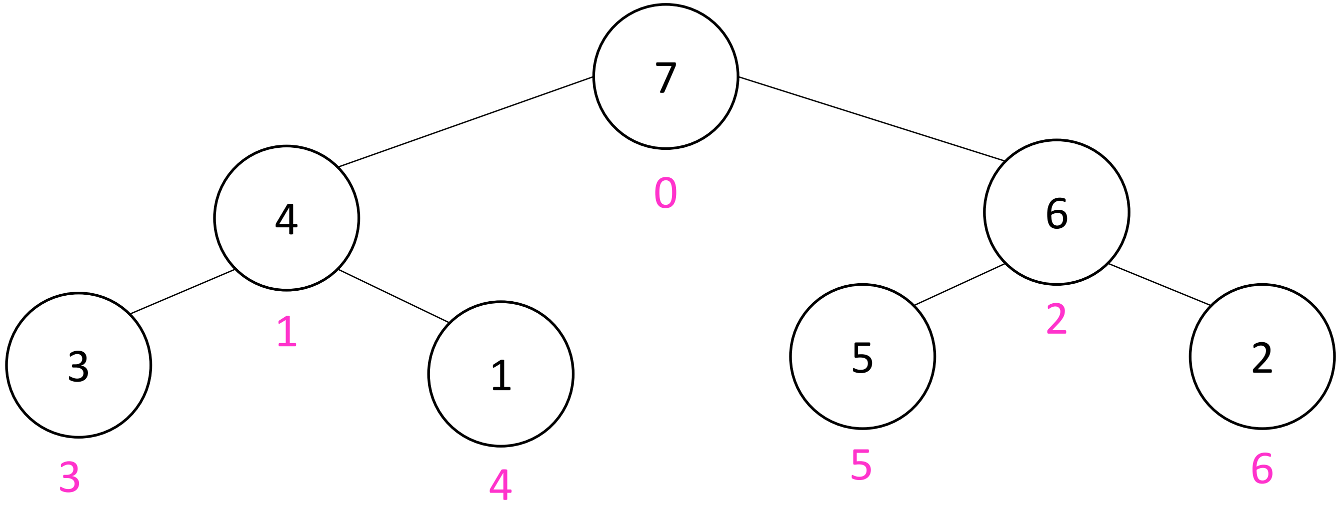
- Idea: When "removing" an element from the heap, swap it with the last item of the heap then "pretend" the heap is one item shorter



Heap Sort

- Idea: When "removing" an element from the heap, swap it with the last item of the heap then "pretend" the heap is one item shorter

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



In Place Heap Sort

- Build a heap using the same array (Floyd's build heap algorithm works)
- Call deleteMax
- Put that at the end of the array

```
buildHeap(a);
for (int i = a.length-1; i>=0; i--){
	temp=a[i]
	a[i] = a[0];
	a[0] = temp;
	percolateDown(0);
}
```

Running Time:

Worst Case: $\Theta(\cdot)$

Best Case: $\Theta(\cdot)$

Floyd’s buildHeap method

- Working towards the root, one row at a time, percolate down

```
buildHeap(){
    for(int i = size; i>0; i--){
        percolateDown(i);
    }
}
```