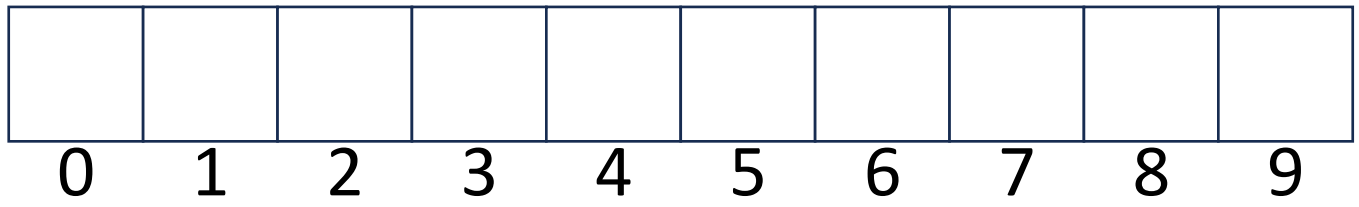
CSE 332 Winter 2026
Lecture 12: Sorting

Nathan Brunelle

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

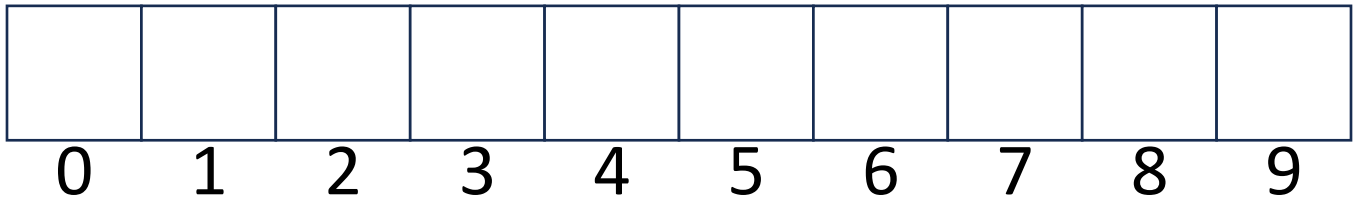
Collision Resolution

- 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



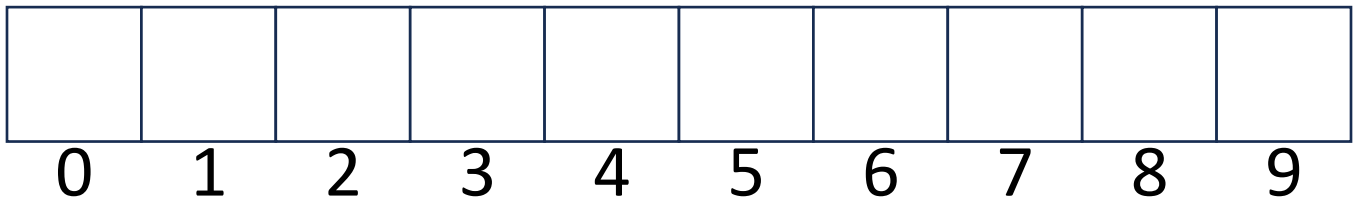
Collision Resolution: Linear Probing

- When there's a collision, use the next open space in the table



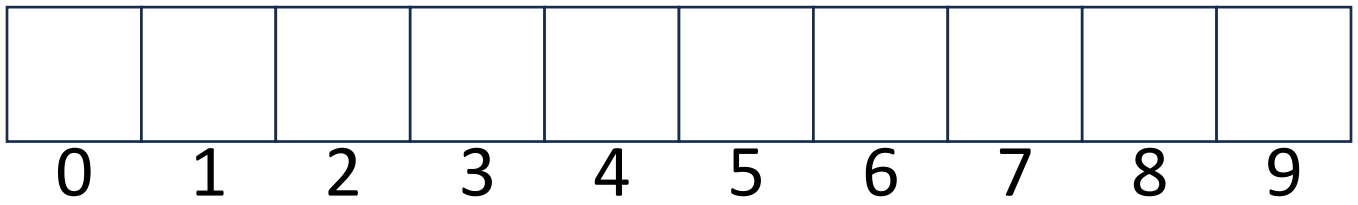
Linear Probing: Insert Procedure

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



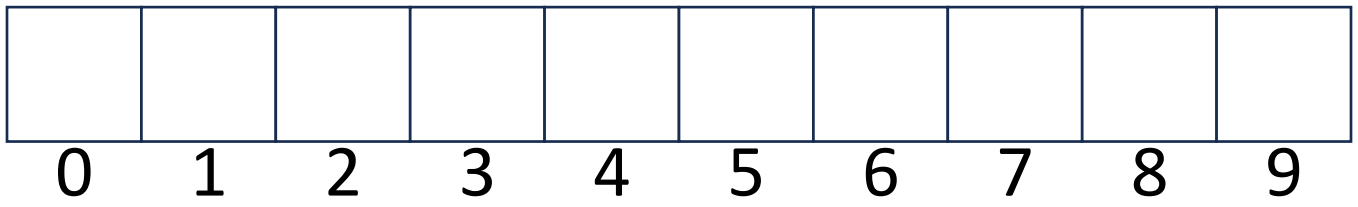
Linear Probing: How to find?

- What do you think?



Linear Probing: Find

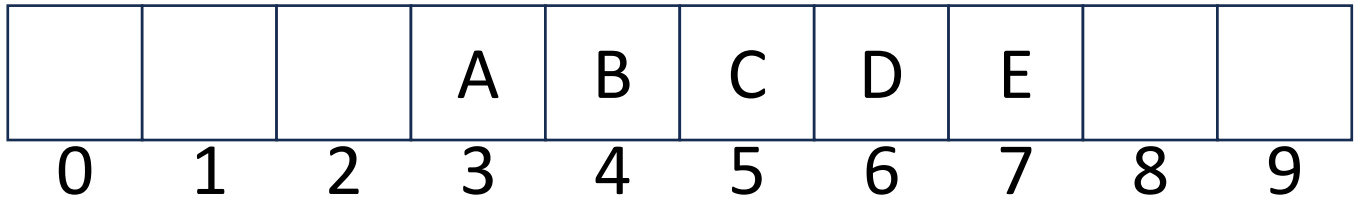
- To find key k
  - Calculate `i = h(k) % table.length`
  - If `table[i]` is occupied but doesn't have k, check `(i+1) % table.length`
  - If that is occupied and doesn't contain k, check `(i+2) % table.length`
  - If that is occupied and doesn't contain k, check `(i+3) % table.length`
  - Repeat until you either find k or else you reach an empty cell in the table



Linear Probing: Delete is Hard (Example 1)

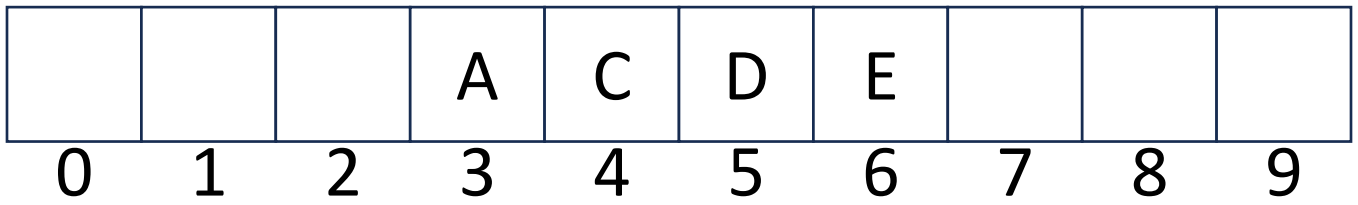
- Suppose we insert A, B, C, D, and E in that order, where all map to 3
- How should we delete B?

Before:



We need future probing to work correctly, so we must fill in the hole.

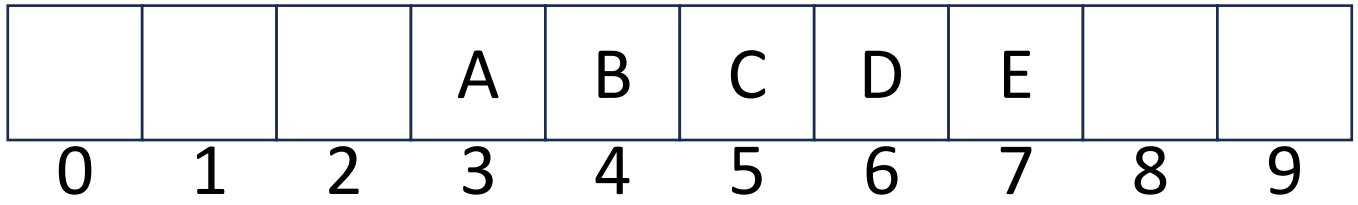
After:



Linear Probing: Delete is Hard (Example 2)

- Suppose we insert A, B, C, D, and E in that order, where A,B,E all map to 3 and C,D map to 5.
- How should we delete B?

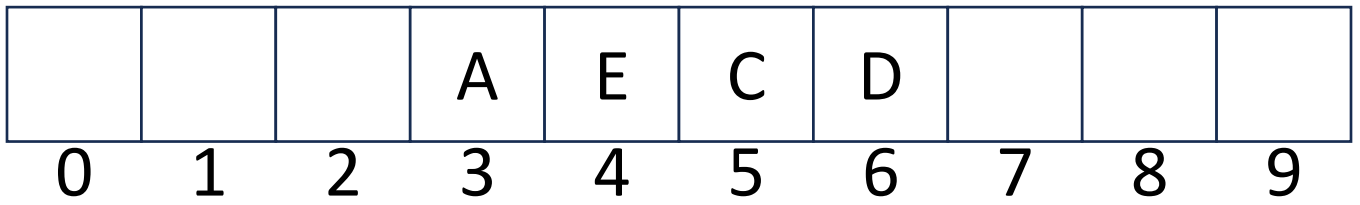
Before:



We need future probing to work correctly, so we must fill in the hole.

We cannot move C or D over because they map to an index later in the probe path

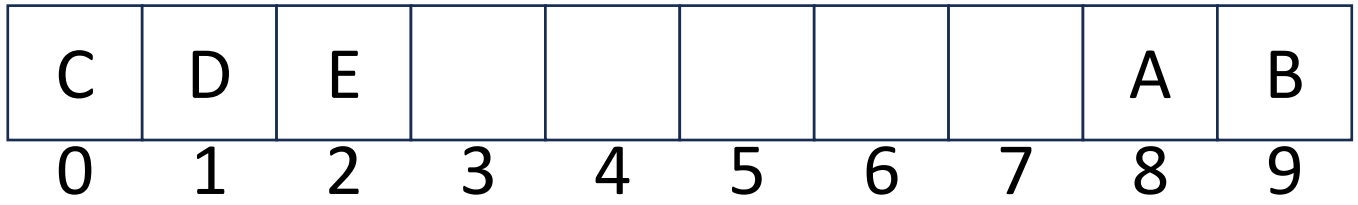
After:



Linear Probing: Delete is Hard (Example 3)

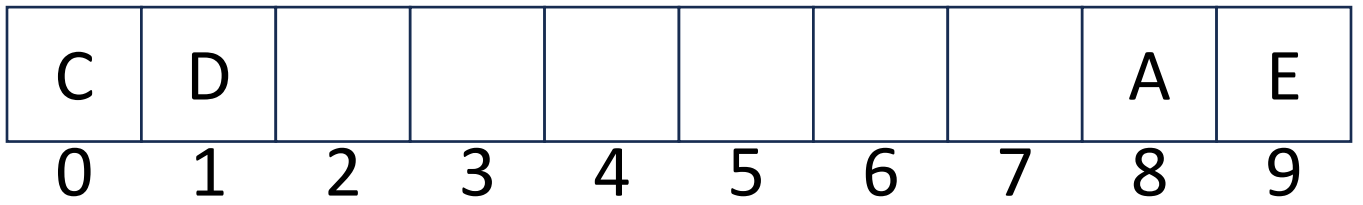
- Suppose we insert A, B, C, D, and E in that order, where A,B,E map to 8 and C,D map to 0.
- How should we delete B?

Before:



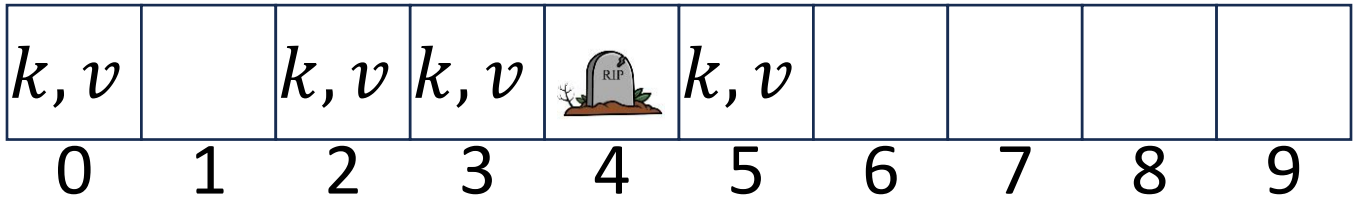
We cannot move C or D over because they map to an index later in the probe path, even though the index number happens to be smaller.
E moves to a larger index because it's earlier in its probe path

After:



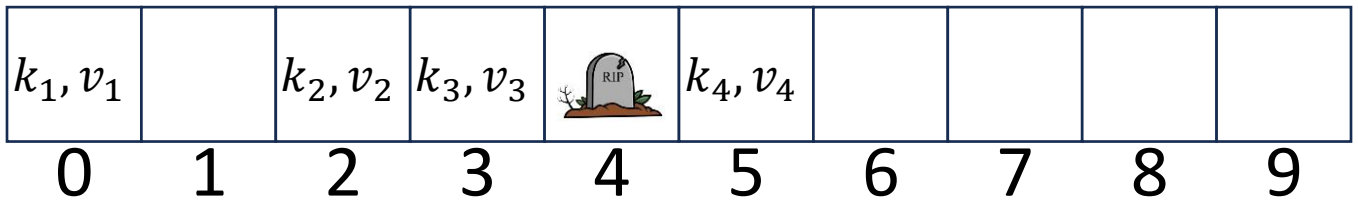
Linear Probing: Delete

- **Option 1 (harder)**: Plug the hole with other items in a way that makes probes behave correctly
  - Something like: to delete at index i, starting from i and going until we find an empty index, if probe path of the key at index j has index i before j then move the key-value pair at j to index i, then repeat on index j.
- **Option 2 (easier)**: "Tombstone" deletion. Leave a special object that indicates an something 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



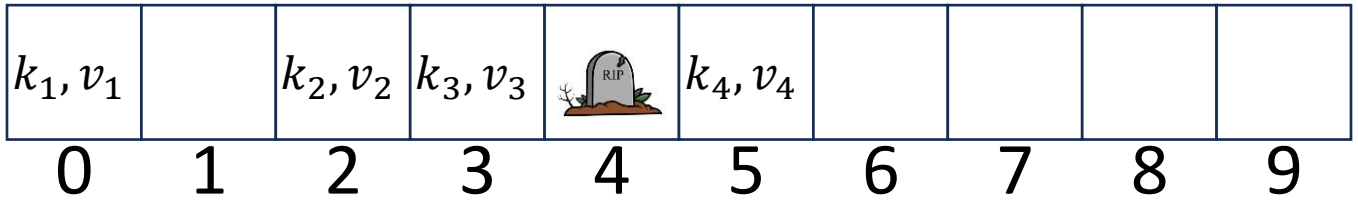
Linear Probing + Tombstone: Find

- To find key k
  - Calculate `i = h(k) % table.length`
  - While `table[i]` has a key other than k, set `i = (i+1) % table.length`
  - If you come across k return `table[i]`
  - If you come across an empty index, the find was unsuccessful
    - Tombstones do not count as empty!



Linear Probing + Tombstone: Insert

- To insert `k,v`
  - Calculate `i = h(k) % table.length`
  - While `table[i]` has a key other than k, set `i = (i+1) % table.length`
    - If `table[i]` has a tombstone, set `dest = i`
      - That is where we will insert if the find is unsuccessful
    - If you come across k, set `table[i] = k,v`
    - If you come across an empty index, the find was unsuccessful
      - Set `table[x] = k,v` if we saw a tombstone
    - Set `table[x] = k,v` otherwise

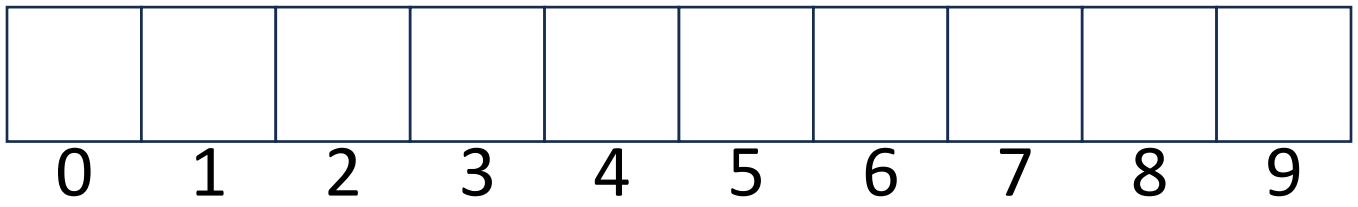


Downsides of Linear Probing

- 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

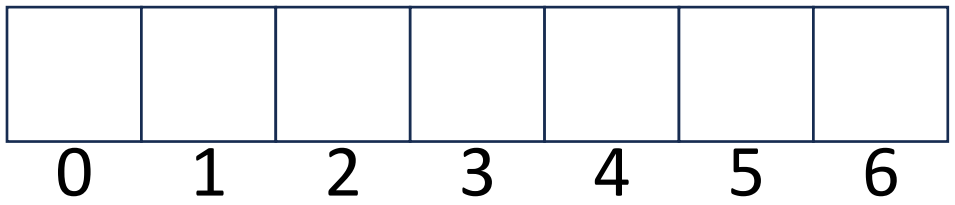
Quadratic Probing: Insert Procedure

- To insert `k,v`
  - Calculate `i = h(k) % table.length`
  - If `table[i]` is occupied then try `(i+1²) % table.length`
  - If that is occupied try `(i+2²) % table.length`
  - If that is occupied try `(i+3²) % table.length`
  - If that is occupied try `(i+4²) % table.length`
  - …



Quadratic Probing: Example

- Insert:
  - 76
  - 40
  - 48
  - 5
  - 55
  - 47



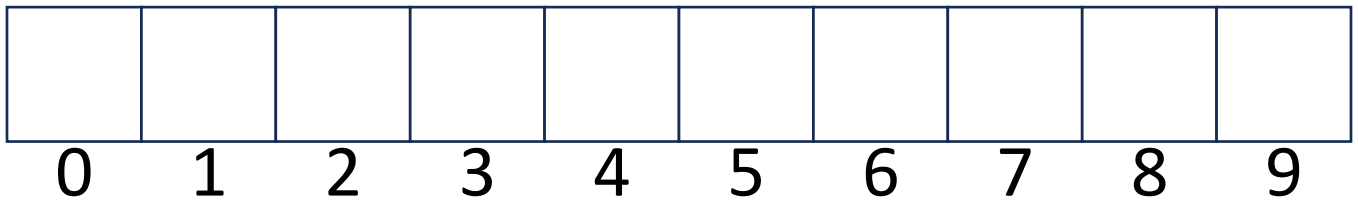
Using Quadratic Probing

- If you probe `table.length` times, you start repeating indices
- If `table.length` is prime and $\lambda < \frac{1}{2}$ then you're guaranteed to find an open spot in at most `table.length`$/2$ 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) % table.length`
  - If `table[i]` is occupied then try `(i+g(k)) % table.length`
  - If that is occupied try `(i+2*g(k)) % table.length`
  - If that is occupied try `(i+3*g(k)) % table.length`
  - If that is occupied try `(i+4*g(k)) % table.length`
  - …



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 $\lambda < \frac{1}{2}$

Sorting

- Rearrangement of items into some defined sequence
  - Usually: reordering a list from smallest to largest according to some metric
- Why sort things?
  - Enable things like binary search
    - It makes some algorithms faster
  - Nicer for human algorithms too
  - Data organization

More Formal Definition

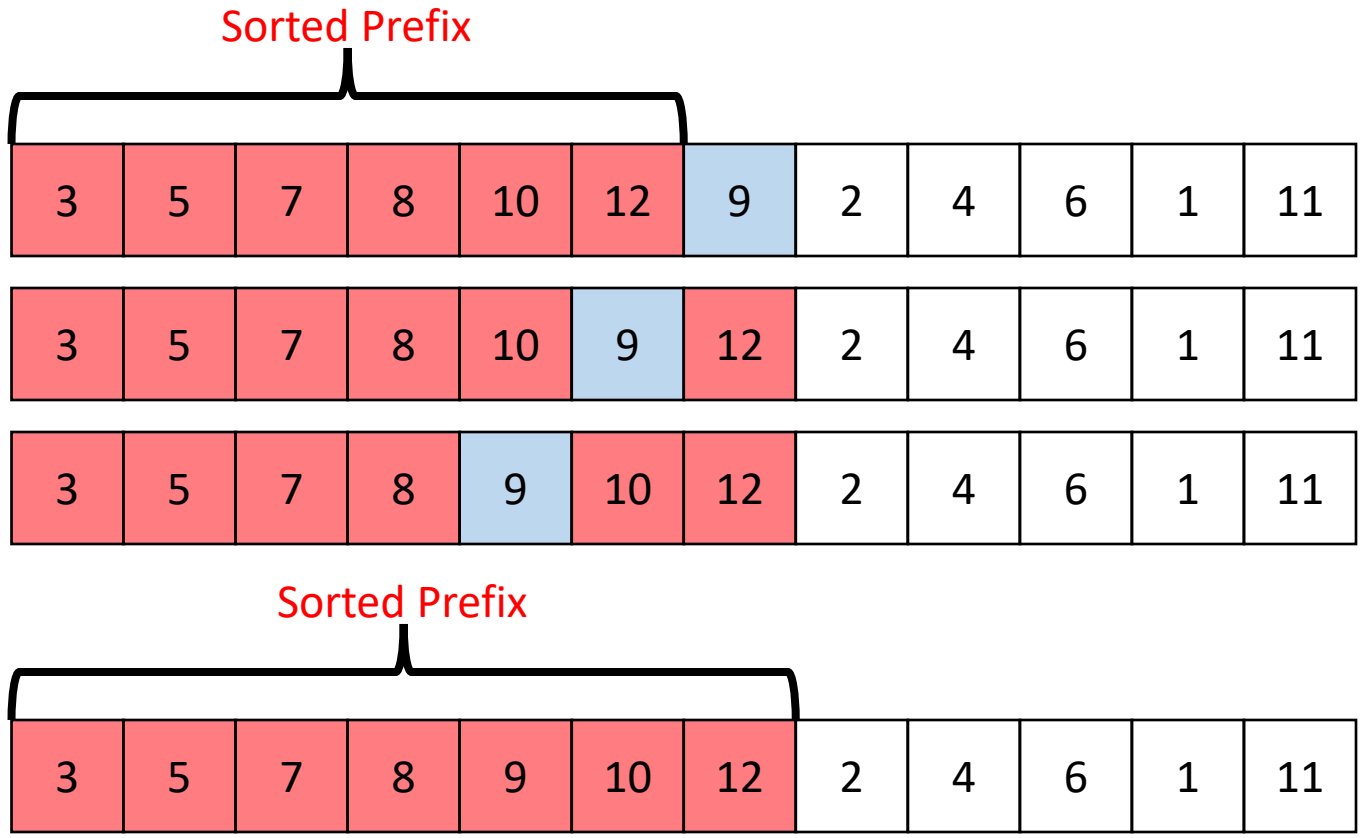
- Input:
  - An array A of items
  - A comparison function for these items
    - Given two items x and y, we can determine whether $x < y$, $x > y$, or $x = y$
- Output:
  - A permutation of A such that if $i \leq j$ then $A[i] \leq A[j]$
  - Permutation: a sequence of the same items but perhaps in a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

Insertion Sort

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



Insertion Sort - Summary

- 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: $\Theta(n^2)$

Best Case: $\Theta(n)$

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

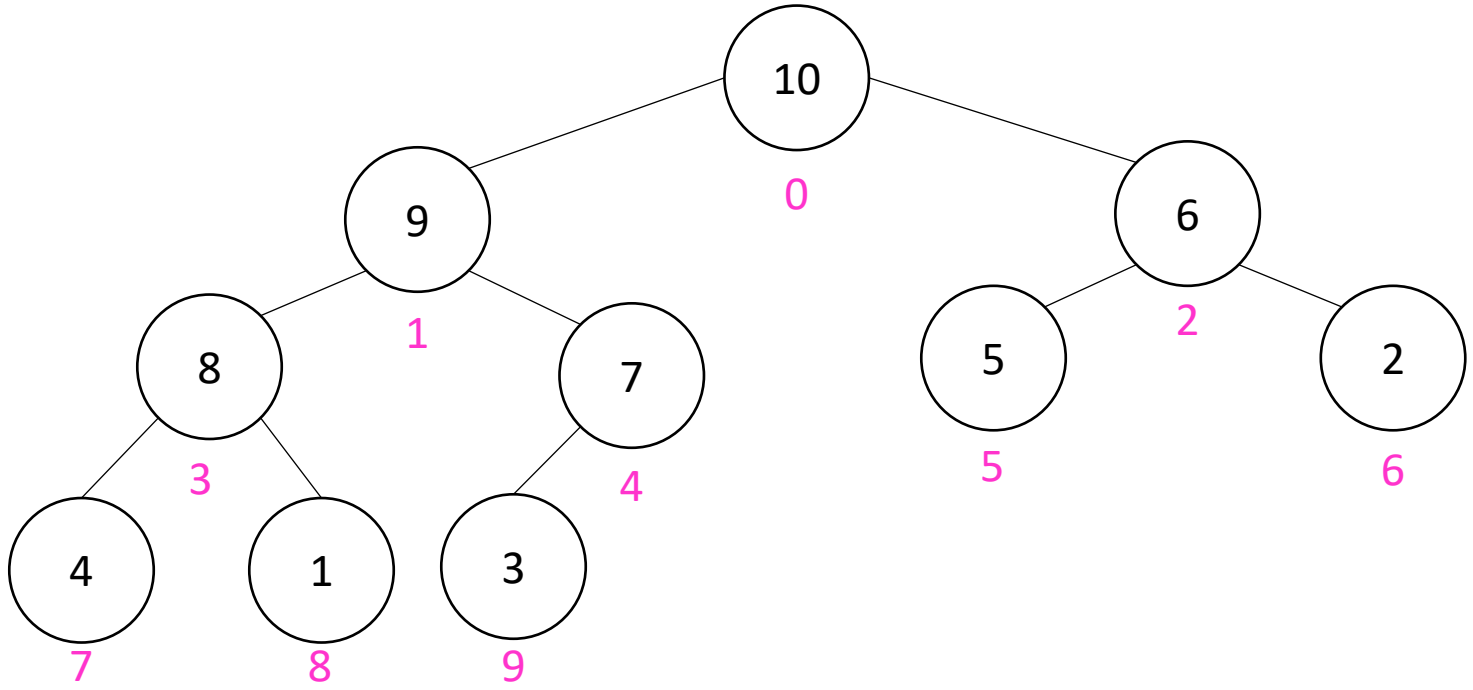
Aside: Bubble Sort – we won’t cover it

"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 extract the max element from the heap to build sorted list Right-to-Left

Max Heap Property: Each node is larger than its children

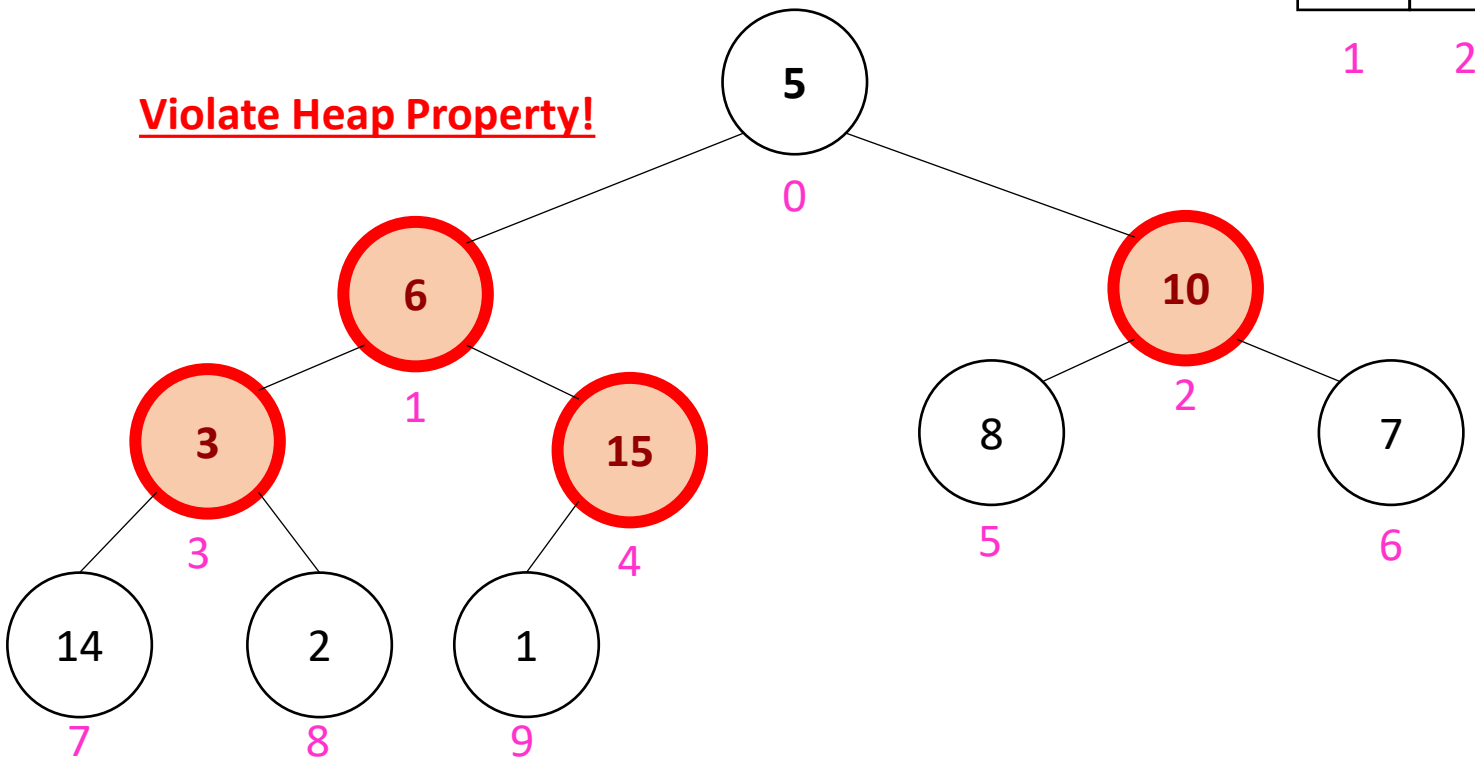


10	9	6	8	7	5	2	4	1	3
0	1	2	3	4	5	6	7	8	9

Building a Heap From "Scratch"

- Suppose we had n items and wanted to "heapify" them

5	6	10	3	15	8	7	14	2	1
1	2	3	4	5	6	7	8	9	10



Two ways for "fix" the heap:

1) Percolate Up
2) Percolate Down

Floyd's buildHeap method

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

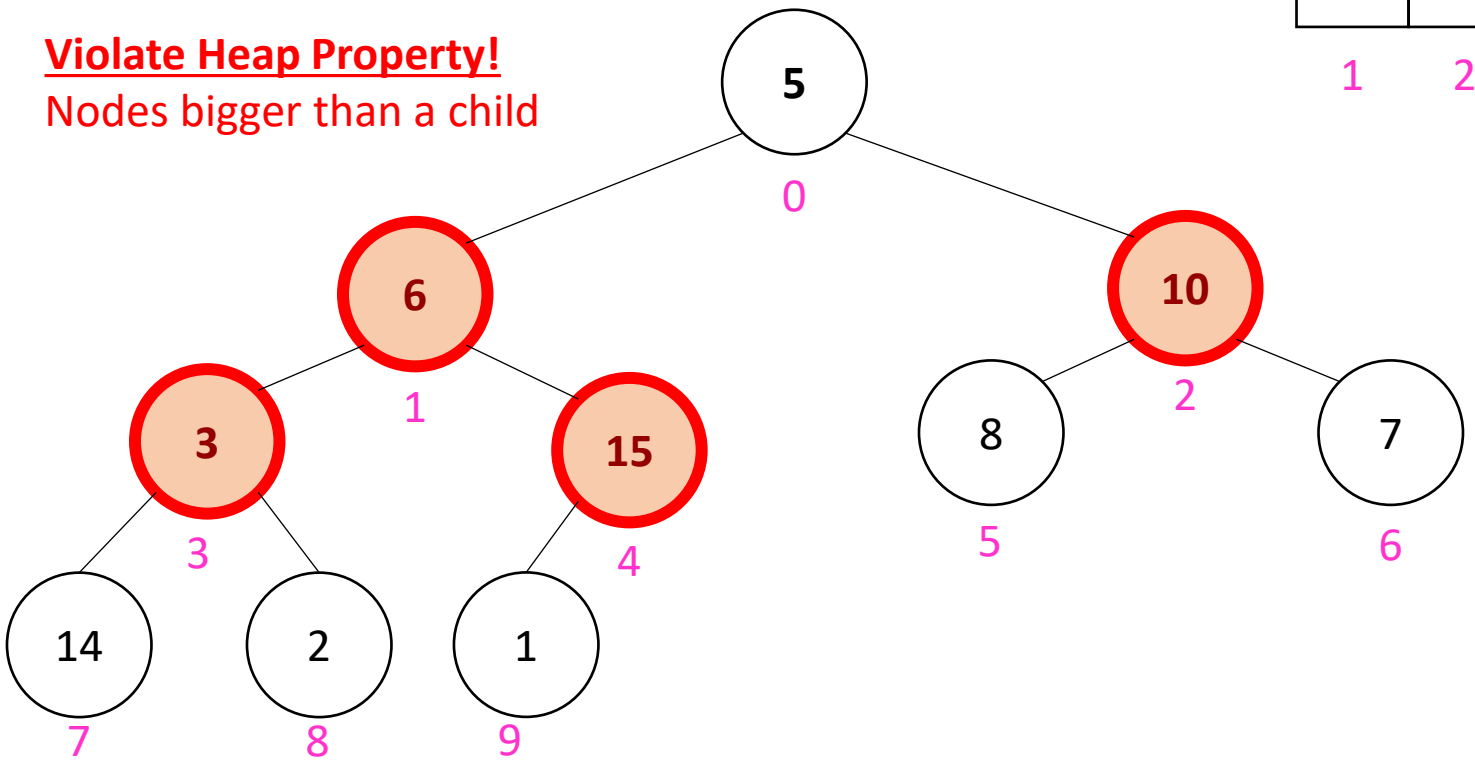
```
buildHeap(){
    for(int i = size; i>0; i--){
        percolateDown(i);
    }
}
```

Floyd's buildHeap method (Percolate 15)

- Suppose we had n items and wanted to "heapify" them

5	6	10	3	15	8	7	14	2	1
1	2	3	4	5	6	7	8	9	10

Violate Heap Property!
Nodes bigger than a child

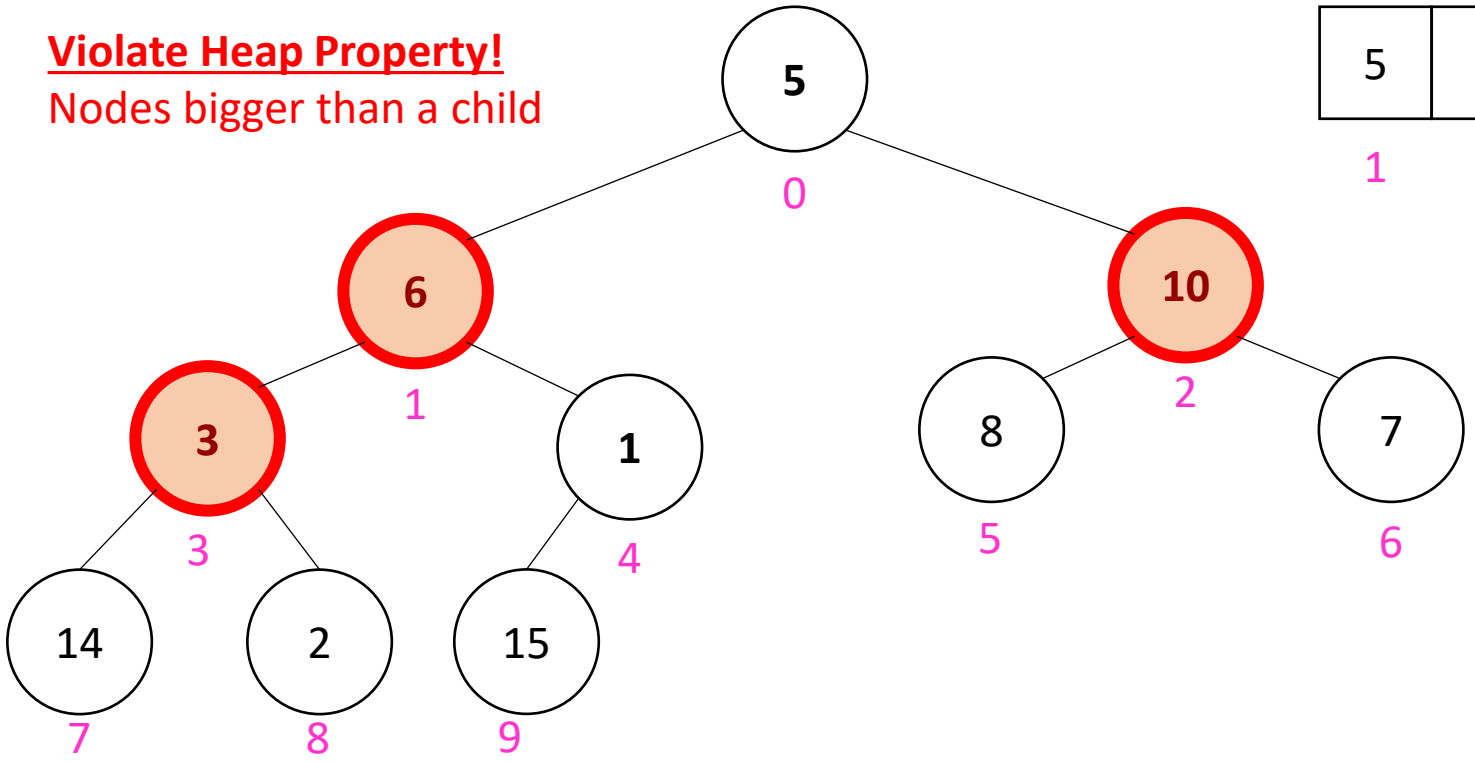



```
buildHeap(){
    for(int i = size; i>0; i--){
        percolateDown(i);
    }
}
```

Floyd's buildHeap method (Percolate 3)

- Suppose we had n items and wanted to "heapify" them

Violate Heap Property!
Nodes bigger than a child

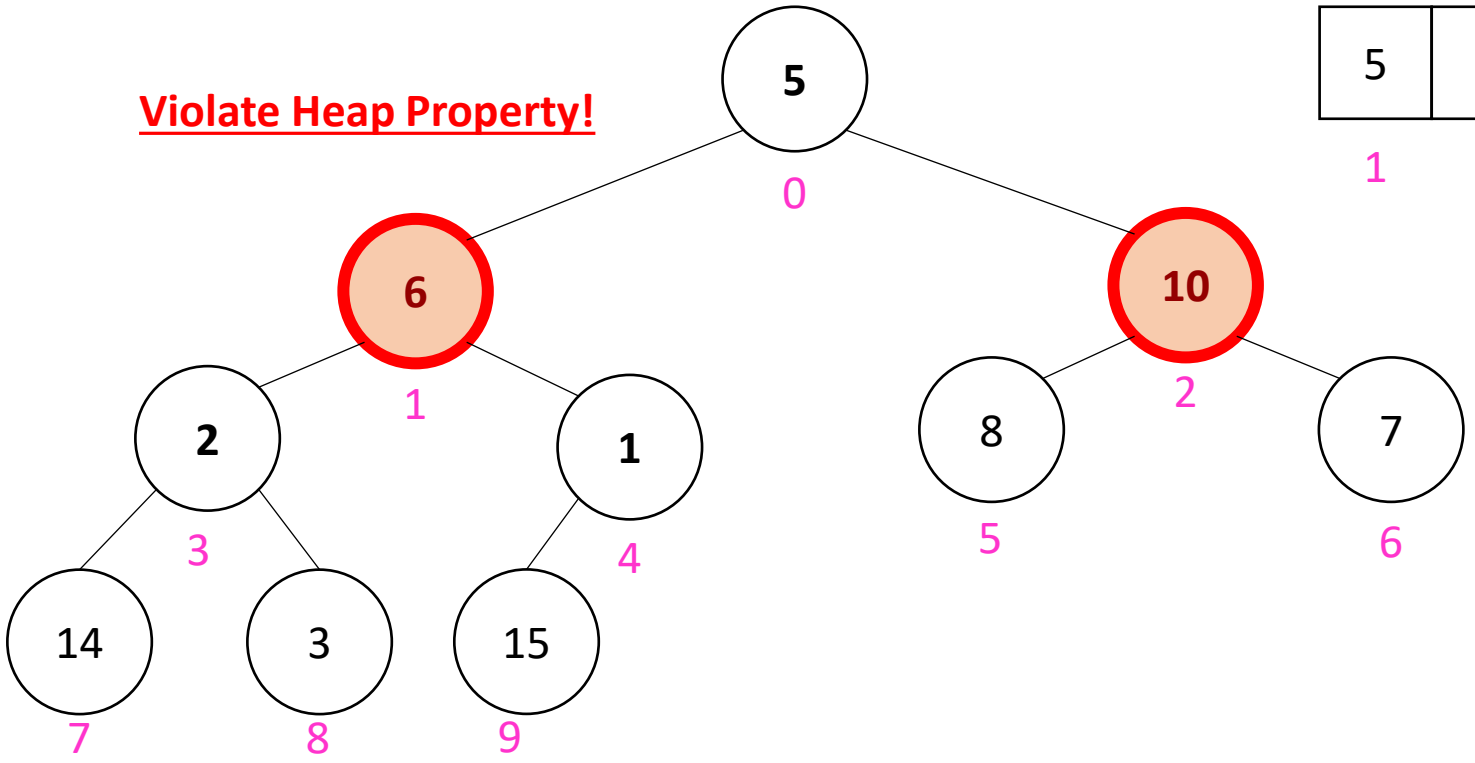


5	6	10	3	1	8	7	14	2	15
1	2	3	4	5	6	7	8	9	10

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

Floyd's buildHeap method (Percolate 10)

- Suppose we had n items and wanted to "heapify" them

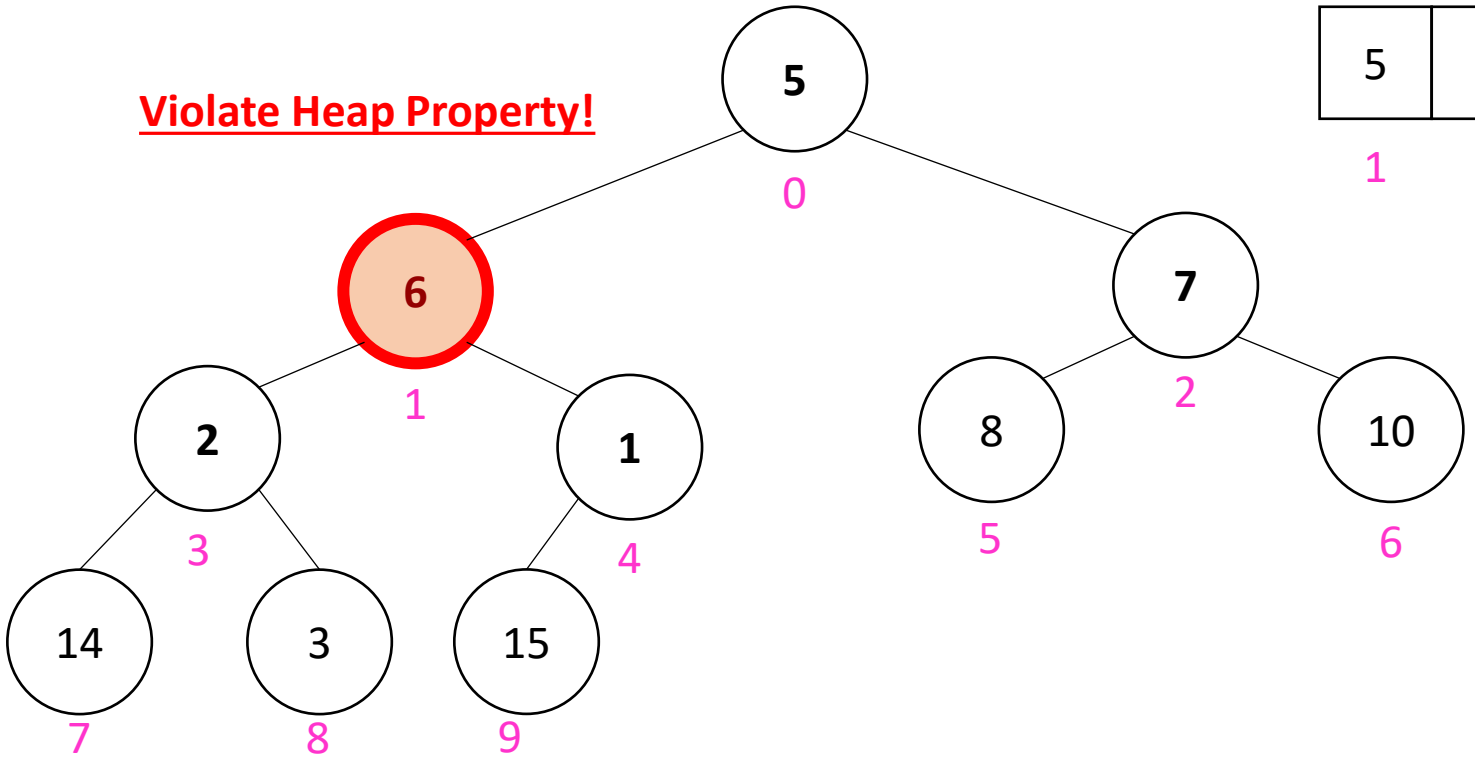


5	6	10	2	1	8	7	14	3	15
1	2	3	4	5	6	7	8	9	10

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

Floyd's buildHeap method (Percolate 6)

- Suppose we had n items and wanted to "heapify" them

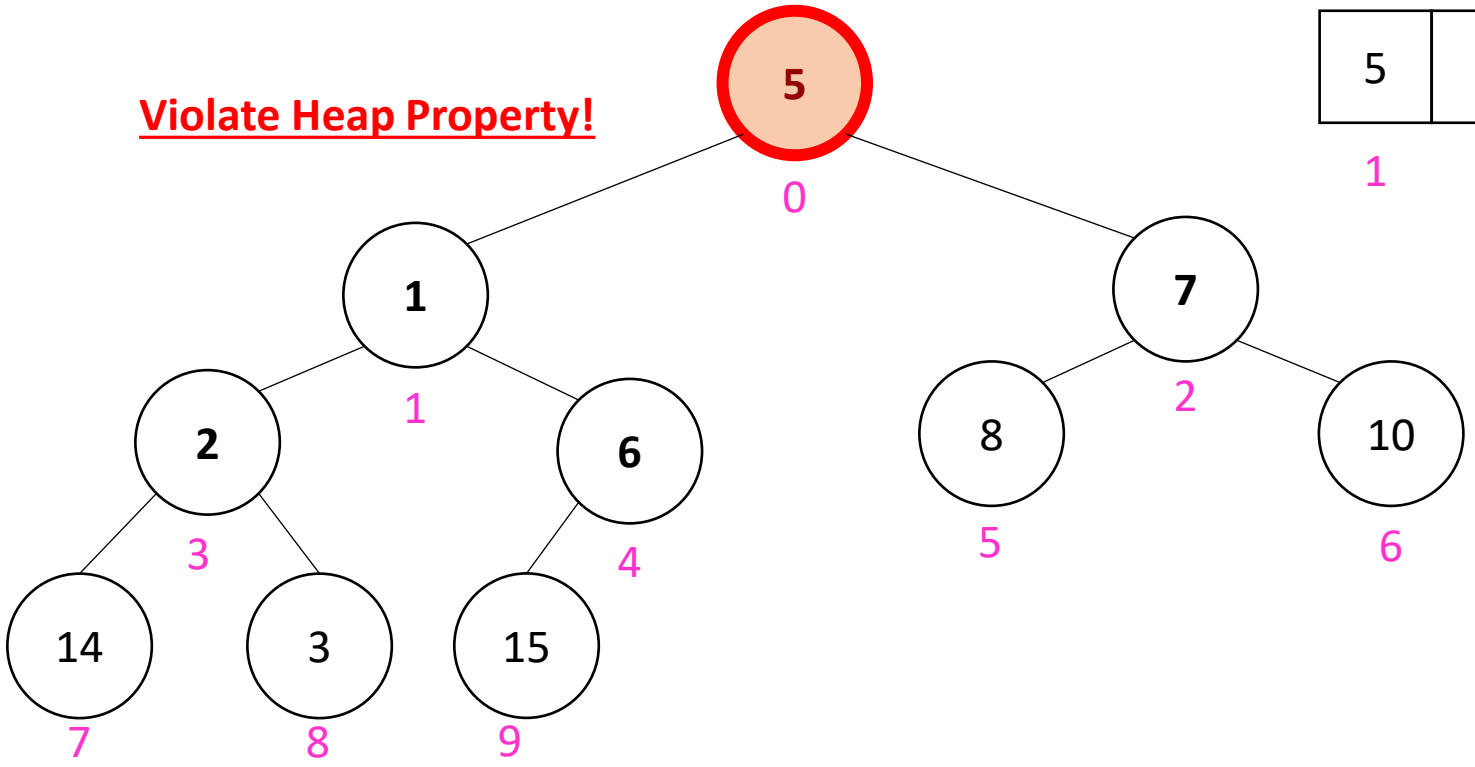


5	6	7	2	1	8	10	14	3	15
1	2	3	4	5	6	7	8	9	10

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

Floyd's buildHeap method (Percolate 5)

- Suppose we had n items and wanted to "heapify" them

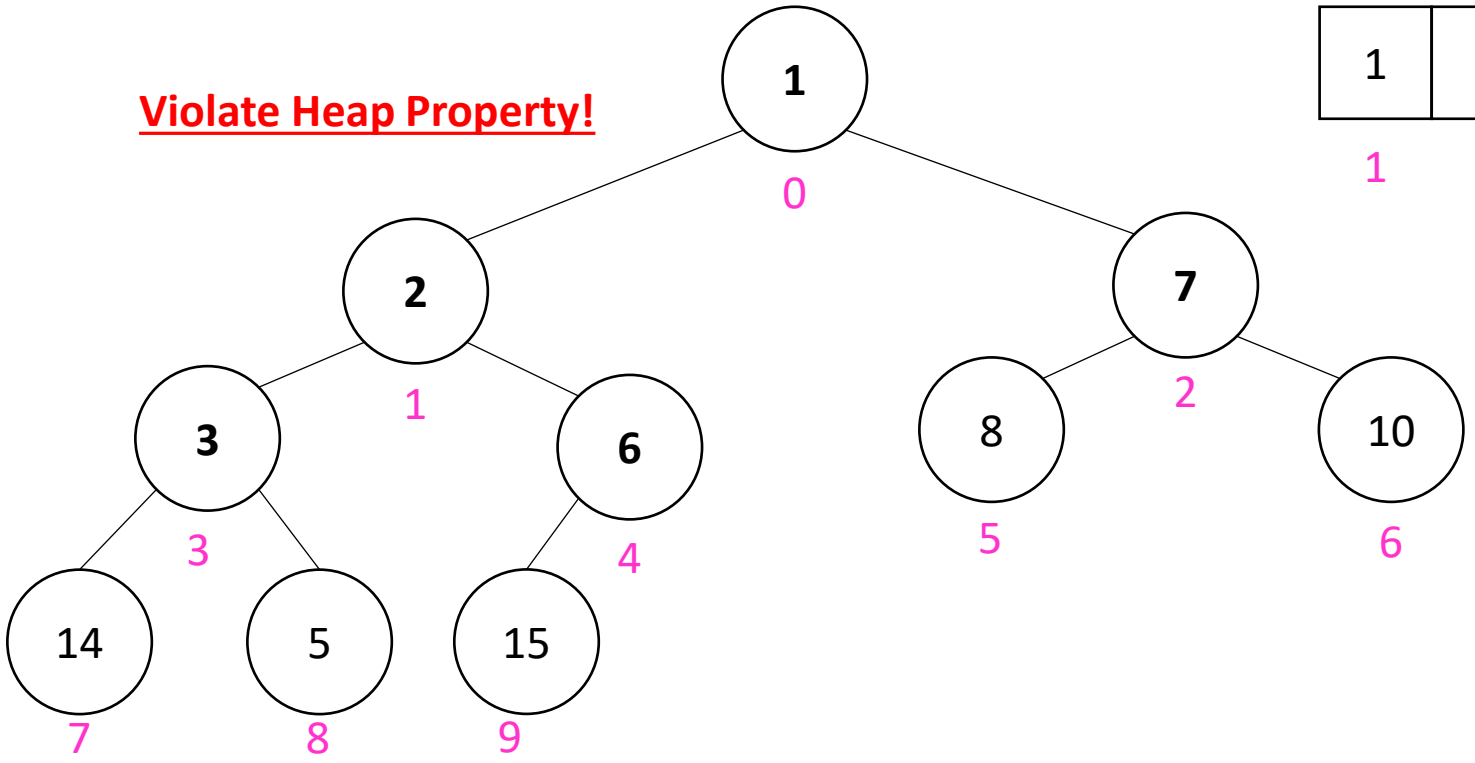


5	1	7	2	6	8	10	14	3	15
1	2	3	4	5	6	7	8	9	10

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

Floyd's buildHeap method (Done)

- Suppose we had n items and wanted to "heapify" them



1	2	7	3	6	8	10	14	5	15
1	2	3	4	5	6	7	8	9	10

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

How long did this take?

- Worst case running time of buildHeap:
- No node can percolate down more than the height of its subtree
  - When i is a leaf:
  - When i is second-from-last level:
  - When i is third-from-last level:
- Overall Running time:

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

How long did this take? (Answers)

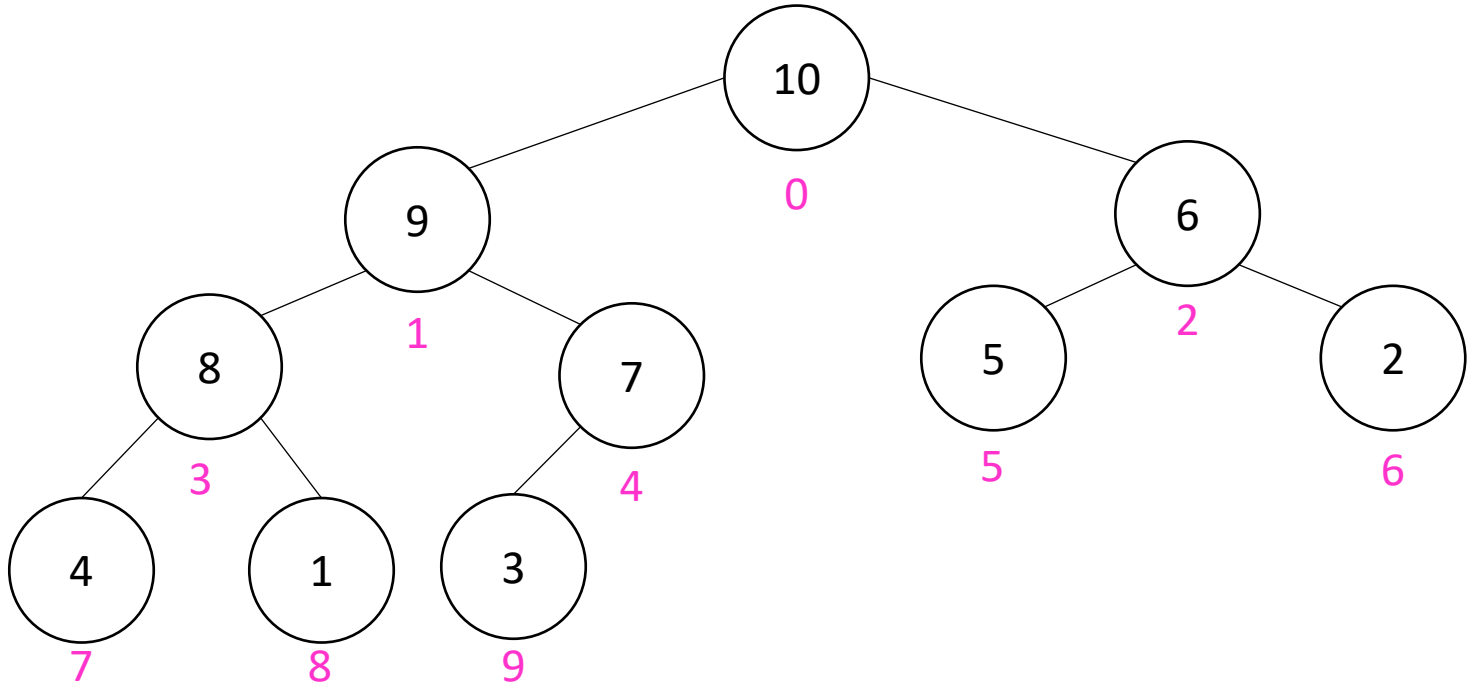
- Worst case running time of buildHeap:
- No node can percolate down more than the height of its subtree
  - When i is a leaf: 0
  - When i is second-from-last level: 1
  - When i is third-from-last level: 2
- Overall Running time:
  - $0 \cdot \frac{n}{2} + 1 \cdot \frac{n}{4} + 2 \cdot \frac{n}{8} + \cdots \log_2 n \cdot 1$
  - $\sum_{i=0}^{\log_2 n} i \cdot \frac{n}{2^{i+1}} = n \sum_{i=0}^{\log_2 n} \frac{i}{2^{i+1}} \leq n \sum_{i=0}^{\infty} \frac{i}{2^{i+1}} = 2n$
  - $\Theta(n)$

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

Heap Sort (Example)

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

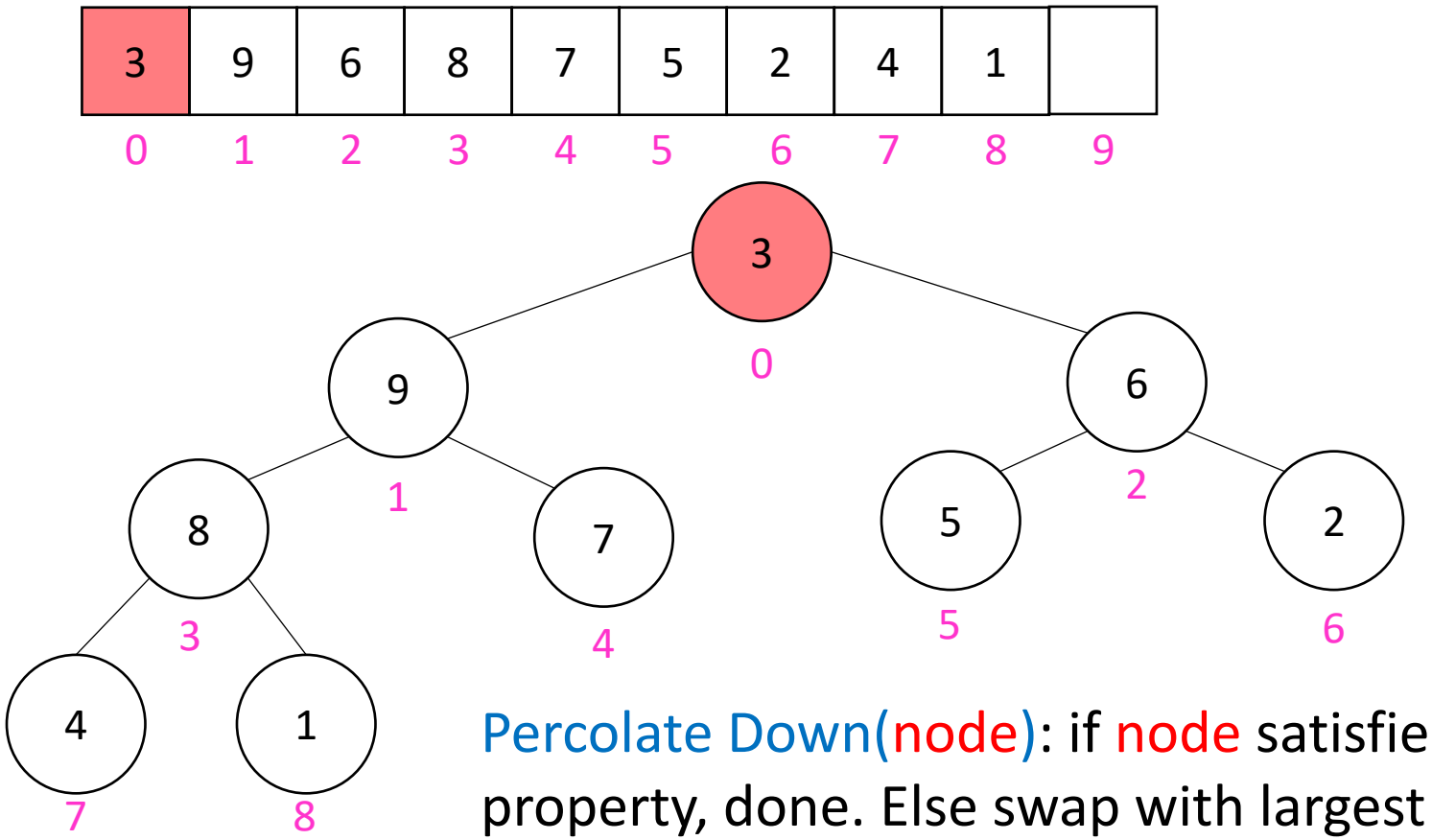
Max Heap Property: Each node is larger than its children



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

Heap Sort (First Extract)

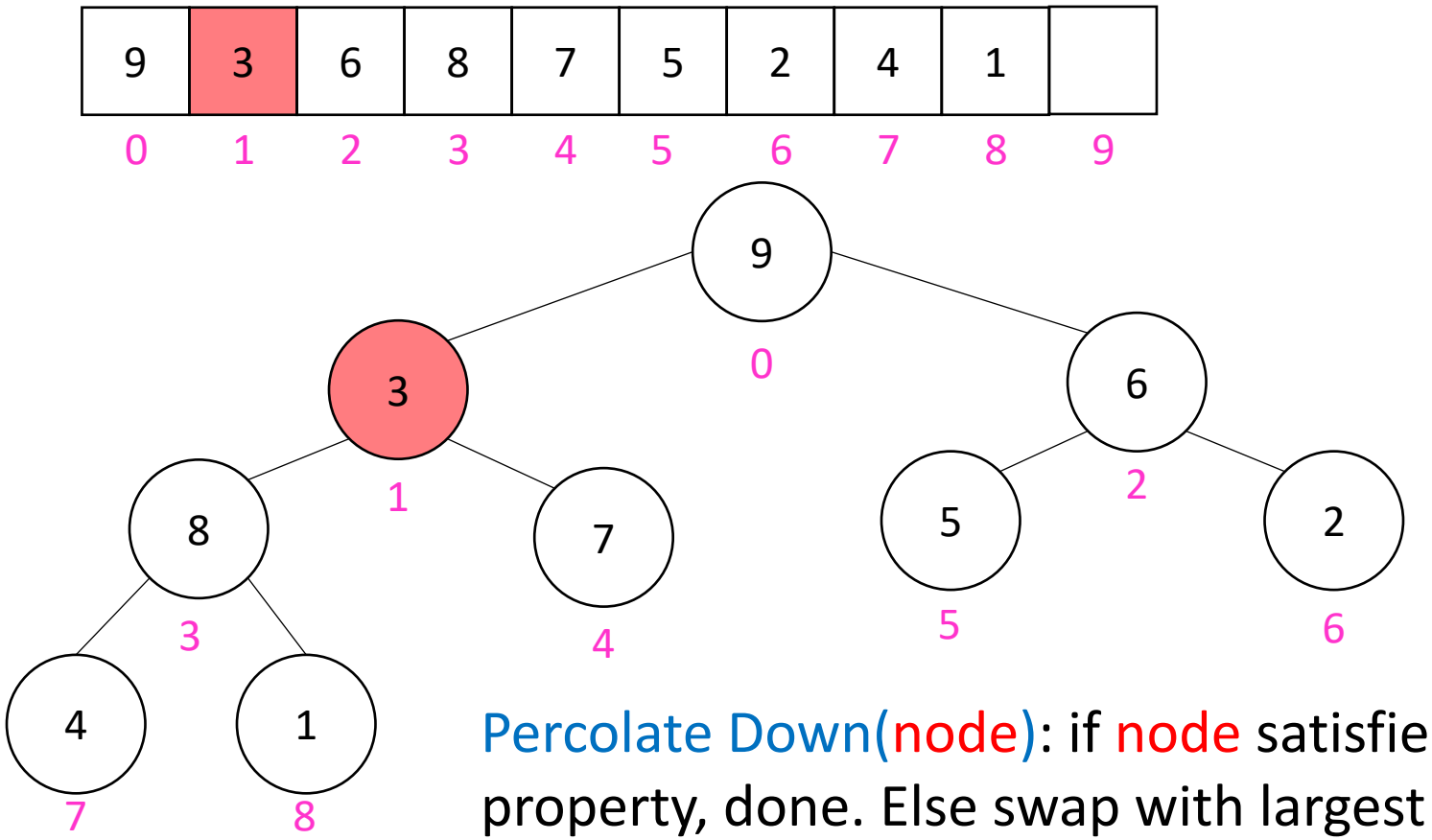
- Remove the Max element (i.e. the root) from the Heap:
  replace with last element, call percolateDown(root)



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

Heap Sort (Percolate Down, Swap 1)

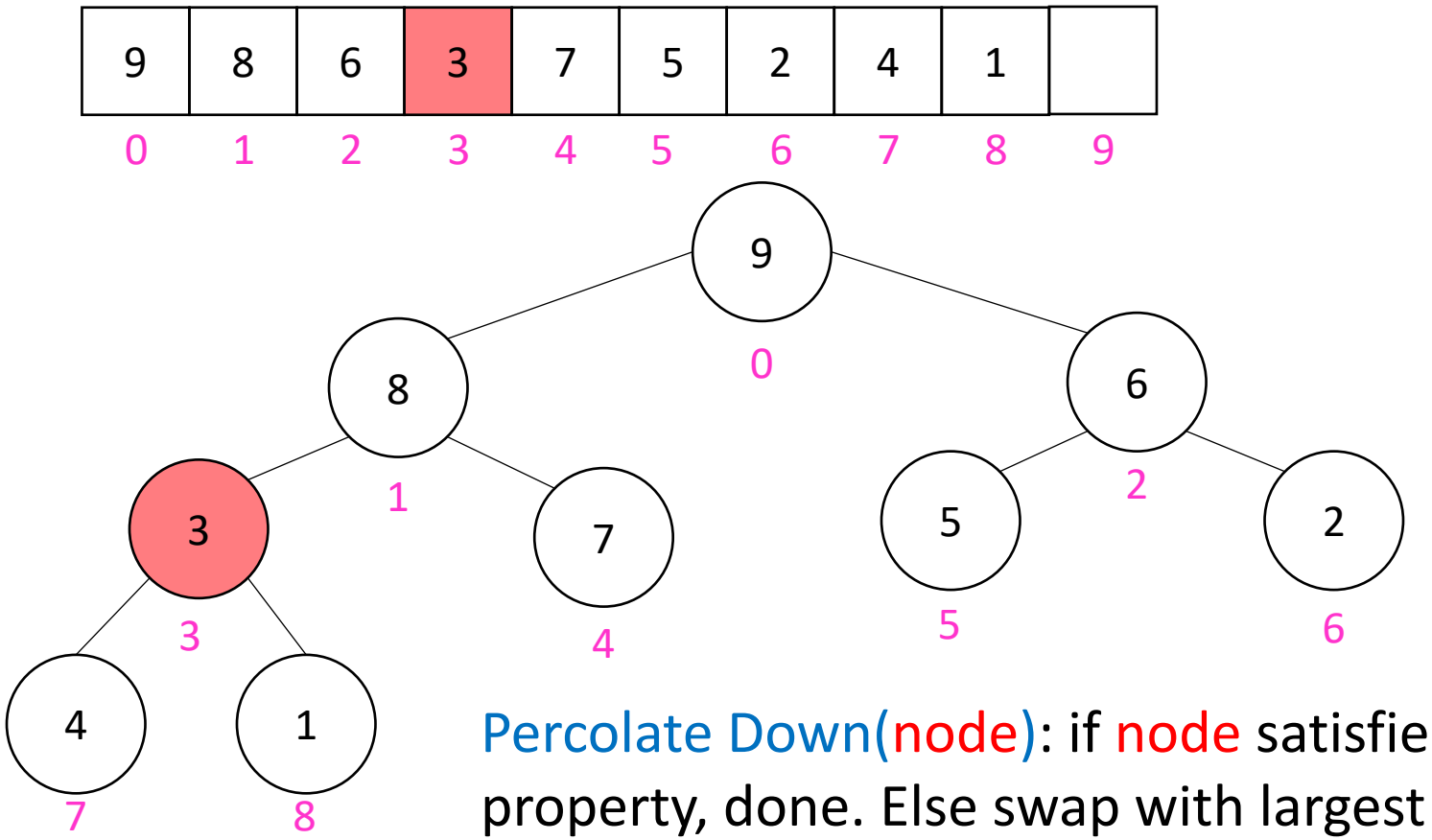
- Remove the Max element (i.e. the root) from the Heap:
  replace with last element, call percolateDown(root)



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

Heap Sort (Percolate Down, Swap 2)

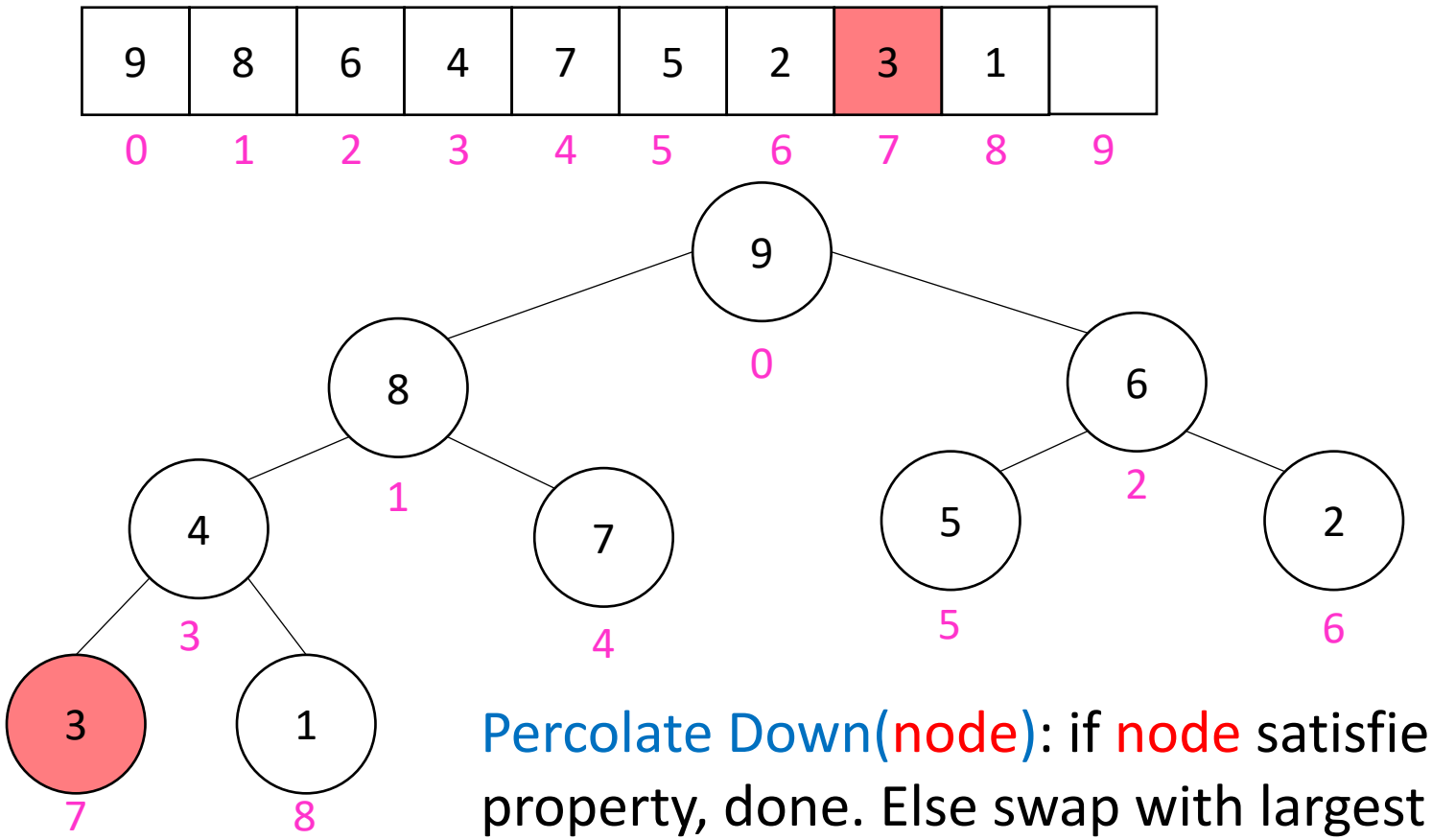
- Remove the Max element (i.e. the root) from the Heap:
  replace with last element, call percolateDown(root)



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

Heap Sort (Percolate Down, Swap 3)

- 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 - Summary

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

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

Running Time:

Worst Case: $\Theta(n \log n)$

Best Case: $\Theta(n \log n)$

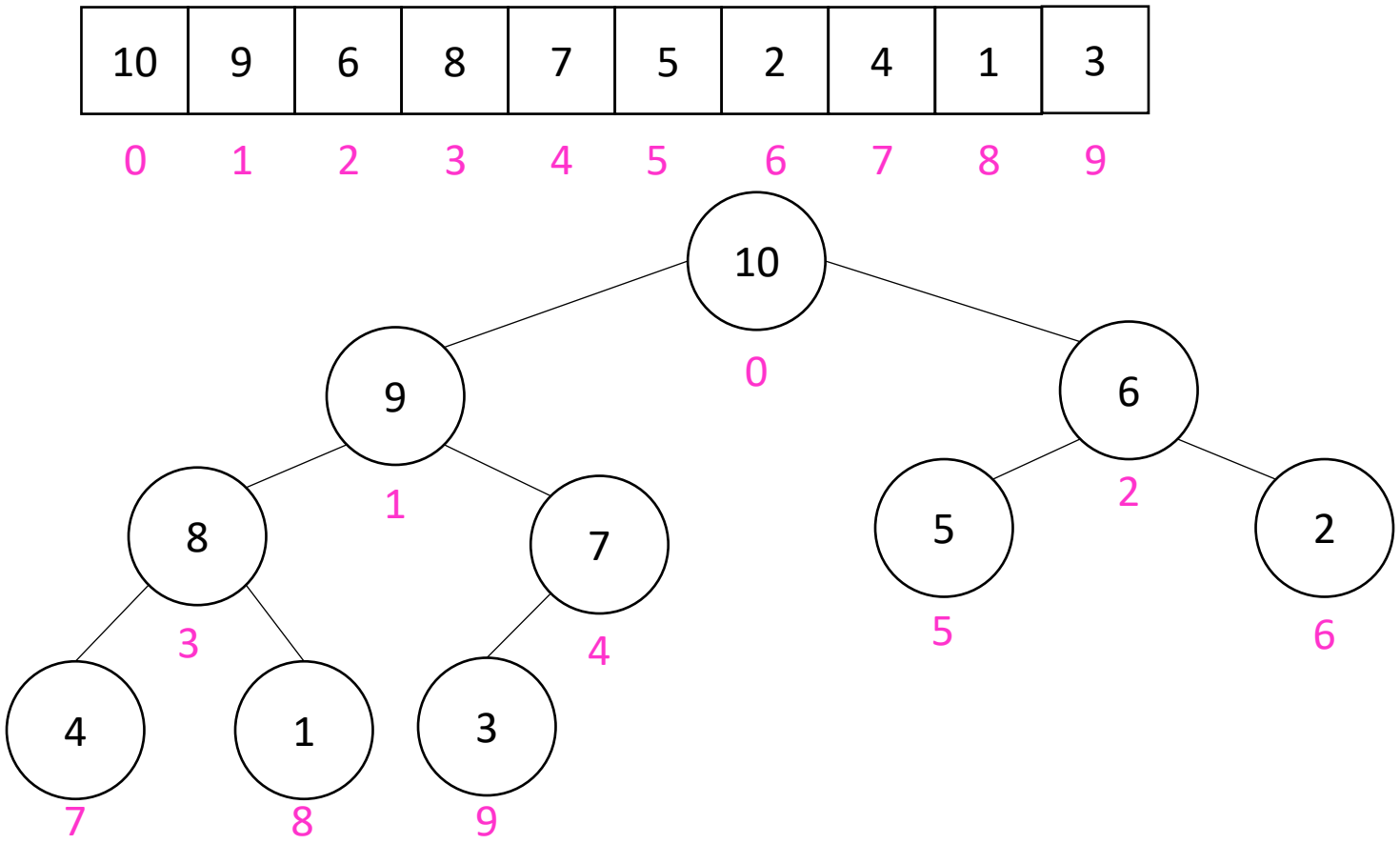
“In Place” Sorting Algorithm

- 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

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

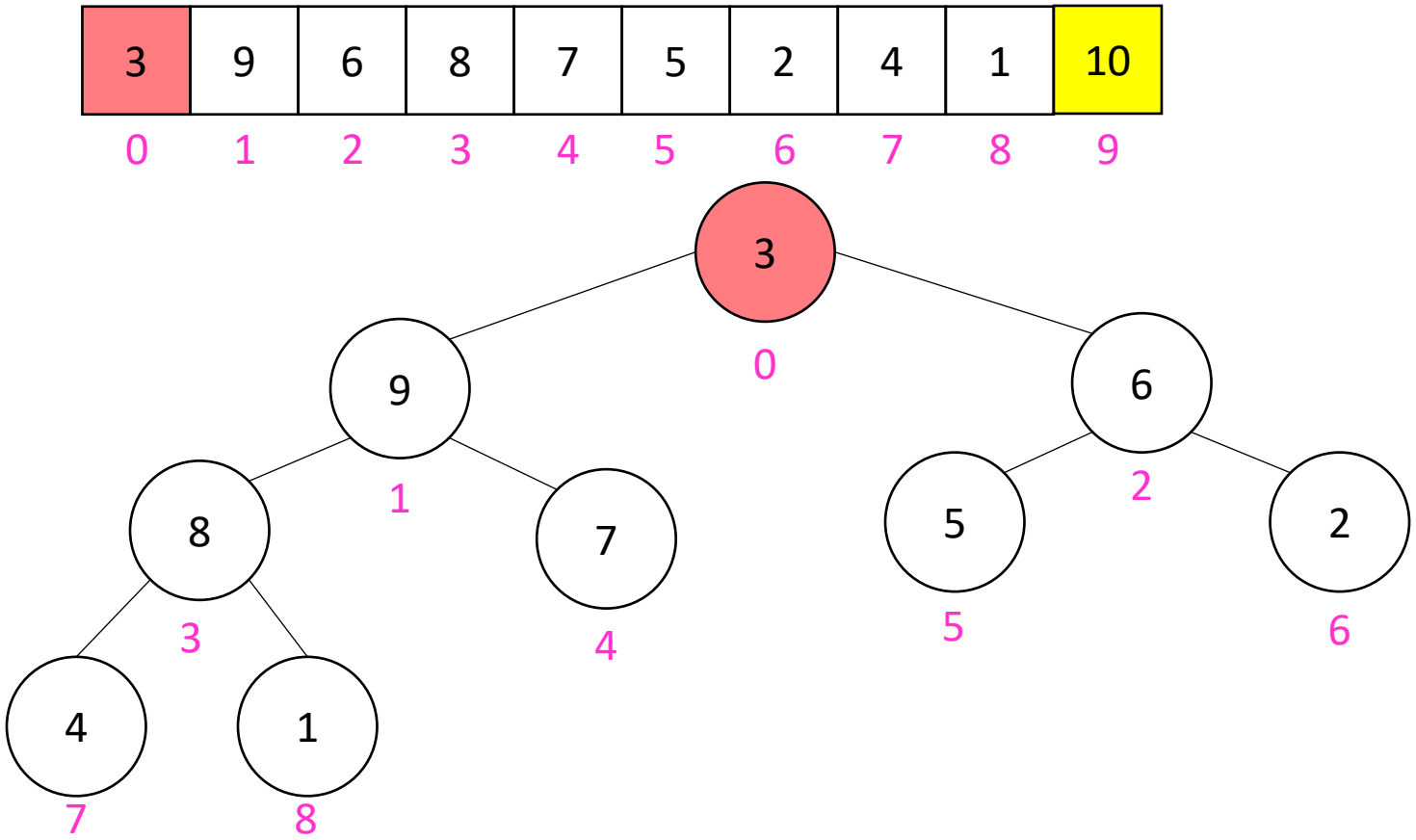
In Place Heap Sort

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



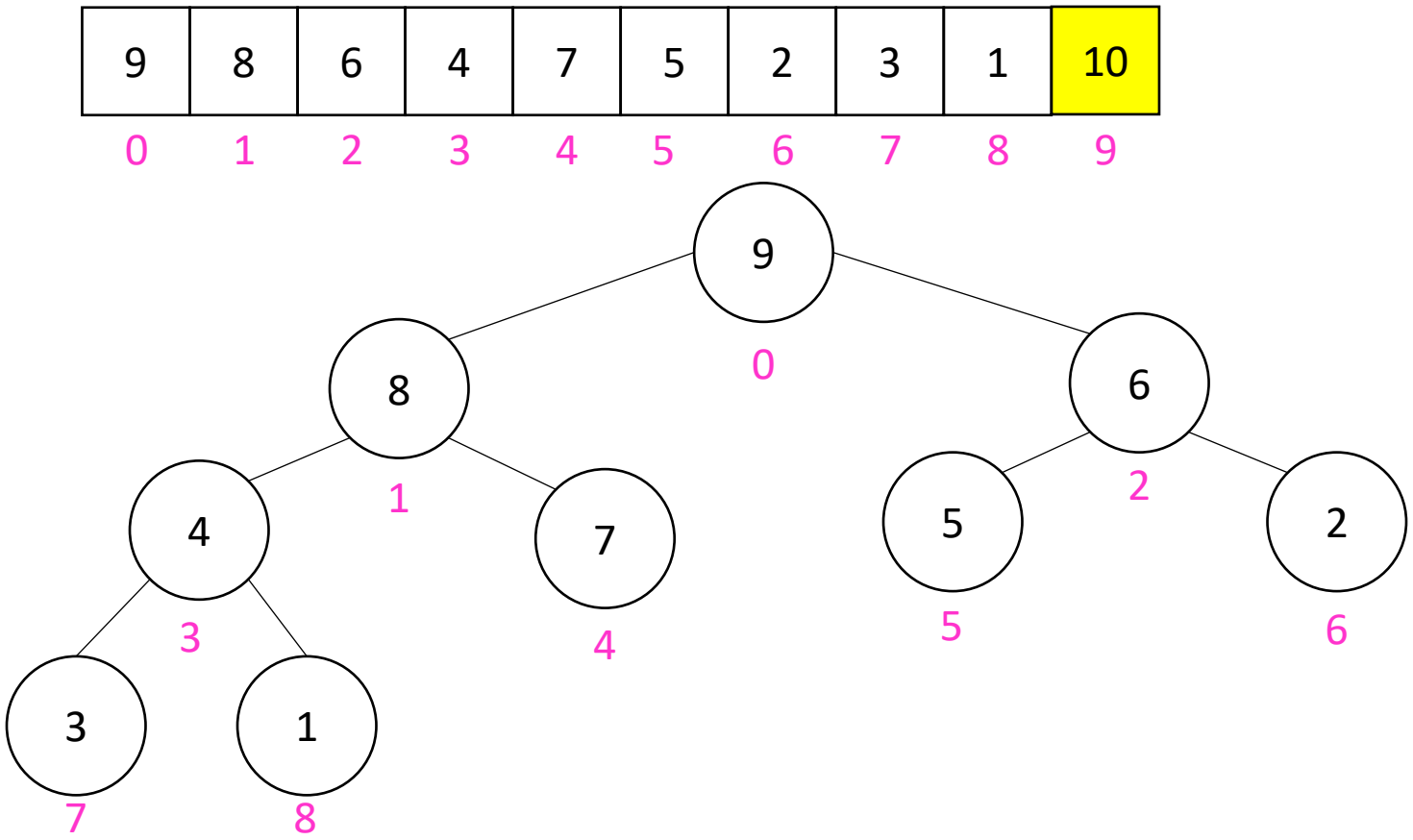
In Place Heap Sort (First Extract)

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



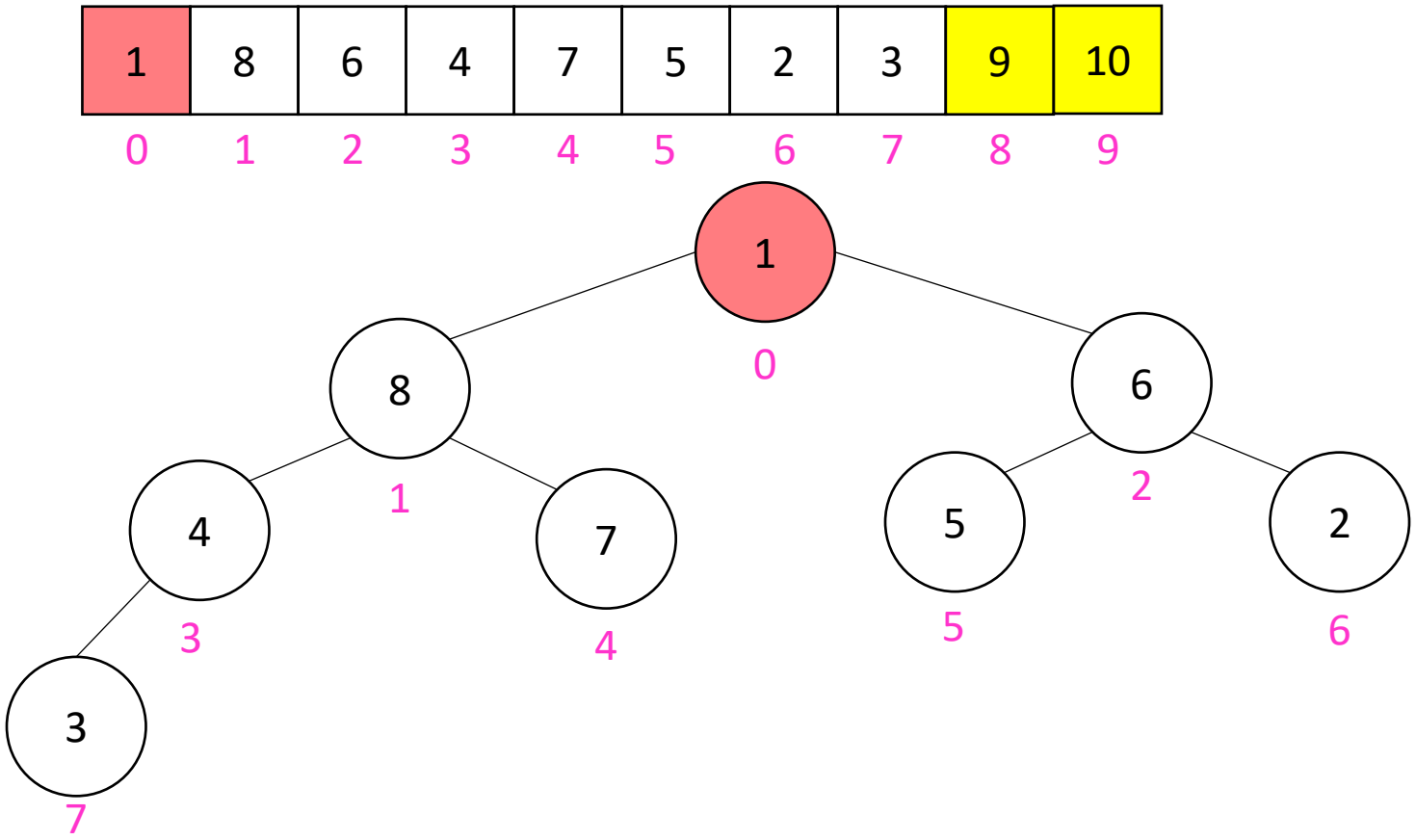
In Place Heap Sort (First Percolate Down)

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



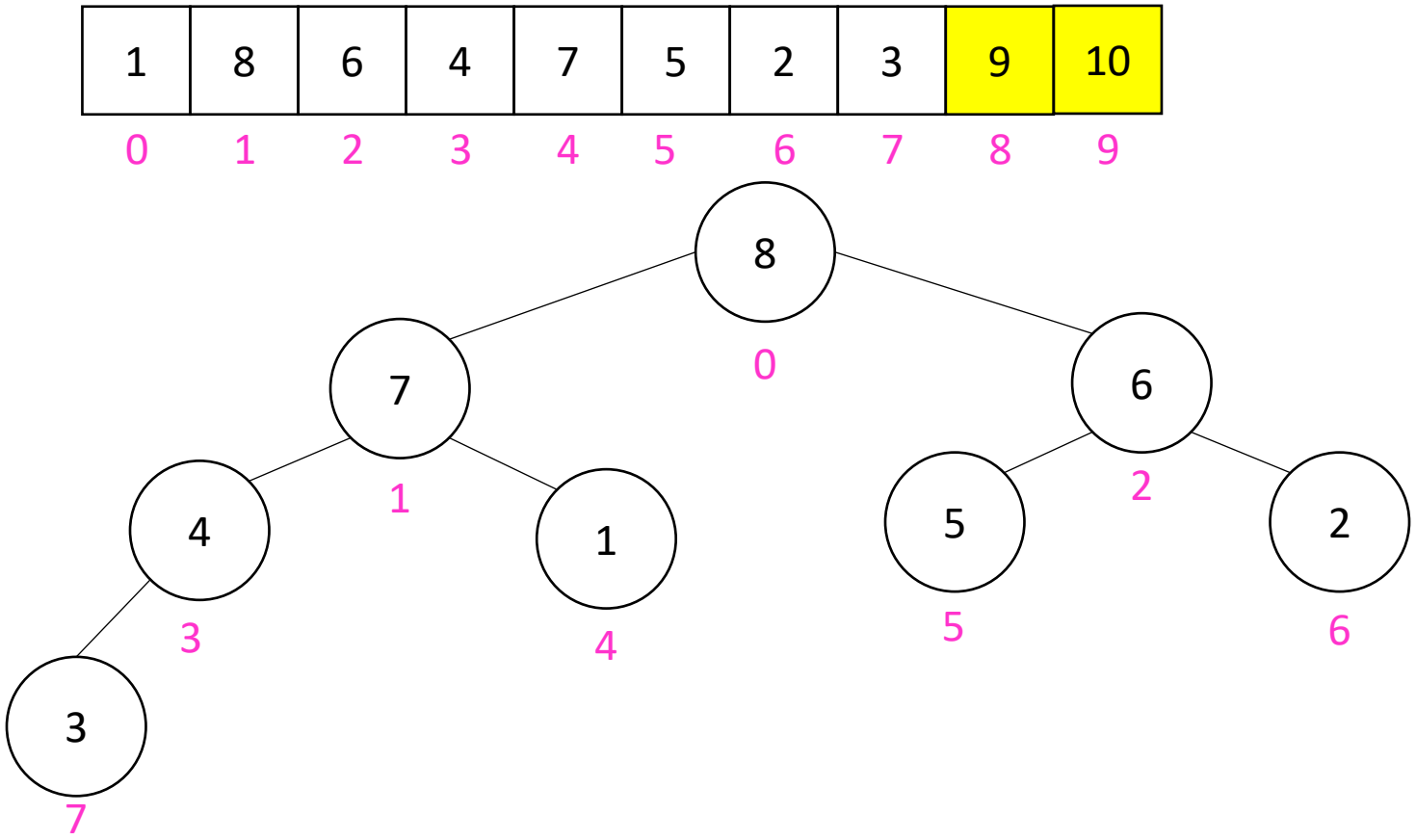
In Place Heap Sort (Second Extract)

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



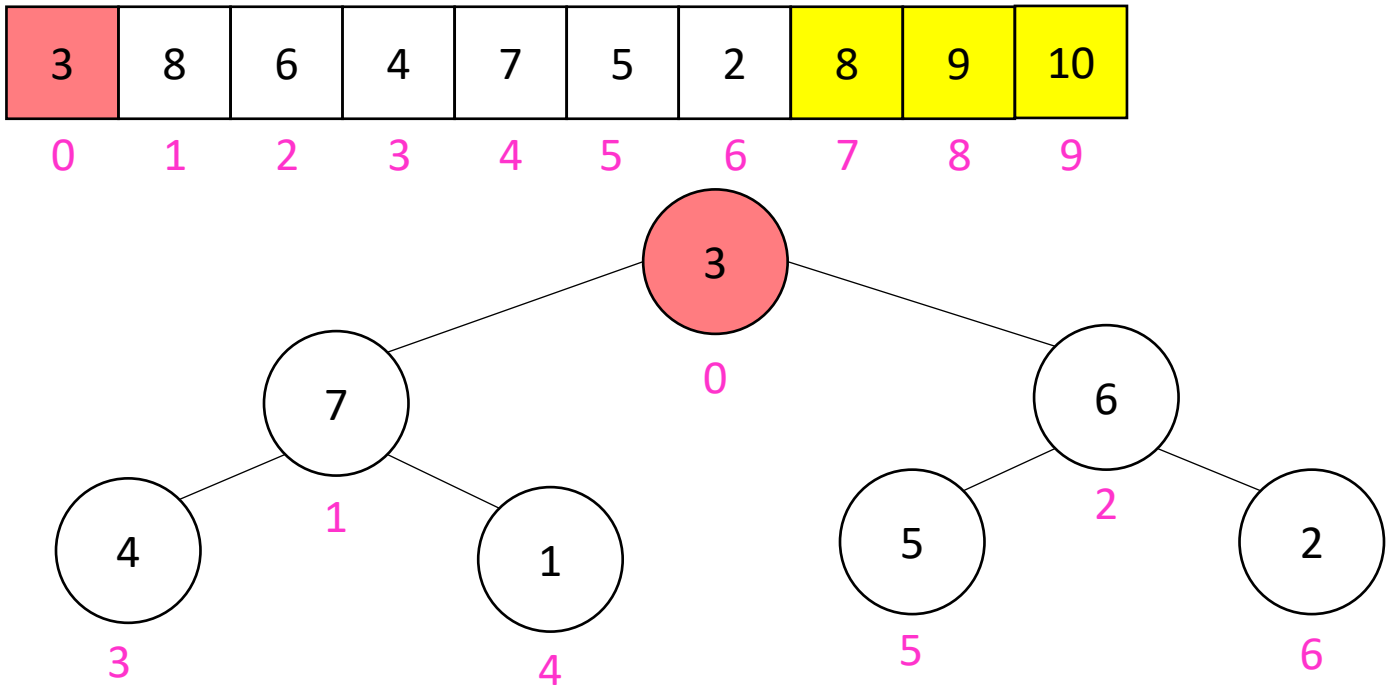
In Place Heap Sort (Second Percolate Down)

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



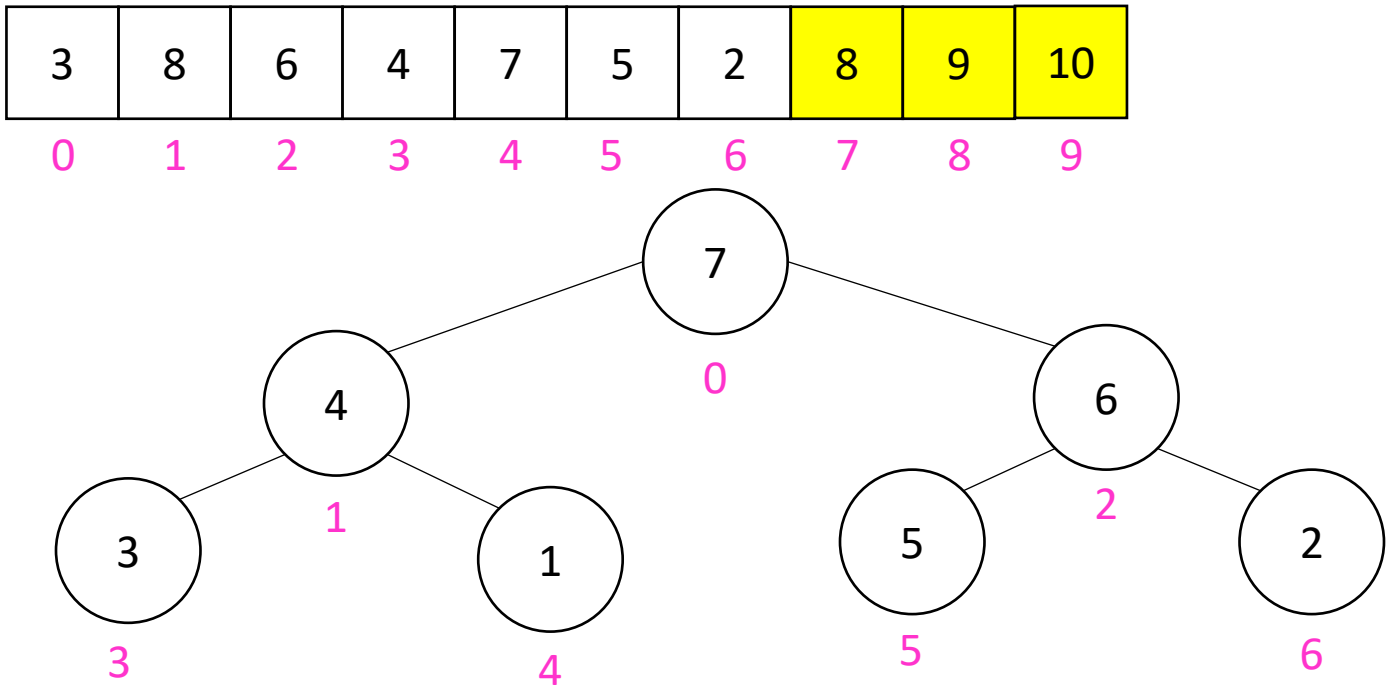
In Place Heap Sort (Third Extract)

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



In Place Heap Sort (Third Percolate Down)

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



In Place Heap Sort - Summary

- Build a heap using the same array (Floyd's build heap algorithm works)
- Call extract
- 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: $\Theta(n \log n)$

Best Case: $\Theta(n \log n)$