CSE 332 Winter 2024
Lecture 6: Priority Queues and Recurrences

Nathan Brunelle

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

ADT: Priority Queue

- What is it?
  - A collection of items and their "priorities"
  - Allows quick access/removal to the "top priority" thing
- What Operations do we need?
  - insert(item, priority)
    - Add a new item to the PQ with indicated priority
    - Usually, smaller priority value means more important
  - deleteMin
    - Remove and return the "top priority" item from the queue
  - Is_empty
    - Indicate whether or not there are items still on the queue
- Note: the "priority" value can be any type/class so long as it's comparable (i.e. you can use "<" or "compareTo" with it)

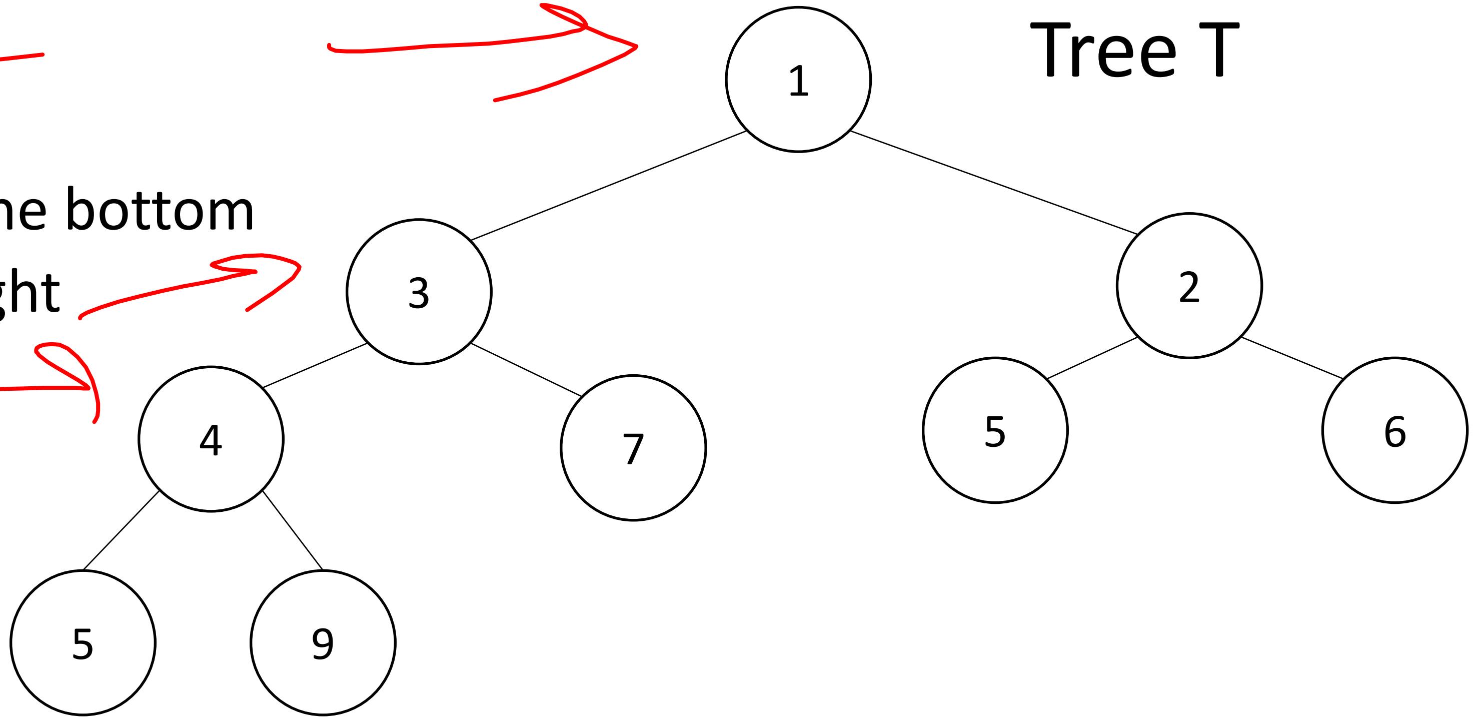
Thinking through implementations

Data Structure	Worst case time to insert	Worst case time to deleteMin
Unsorted Array	$\Theta(1)$	$\Theta(n)$
Unsorted Linked List	$\Theta(1)$	$\Theta(n)$
Sorted Array	$\Theta(n)$	$\Theta(n)$
Sorted Linked List	$\Theta(n)$	$\Theta(1)$
Binary Search Tree	$\Theta(n)$	$\Theta(n)$
Binary Heap	$\Theta(\log n)$	$\Theta(\log n)$

Note: Assume we know the maximum size of the PQ in advance

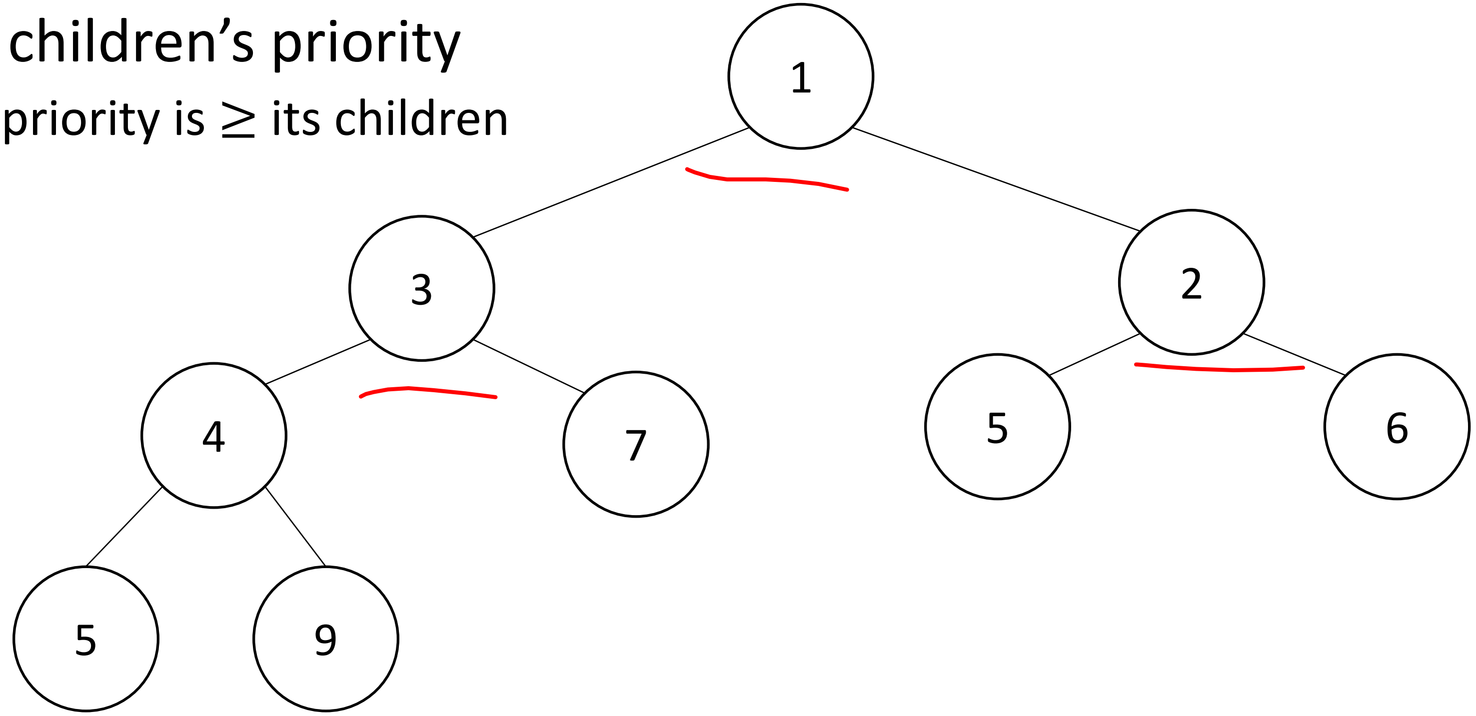
Trees for Heaps

- Binary Trees:
  - The branching factor is 2
  - Every node has ≤ 2 children
- Complete Tree:
  - All “layers” are full, except the bottom
  - Bottom layer filled left-to-right



(Min) Heap Data Structure

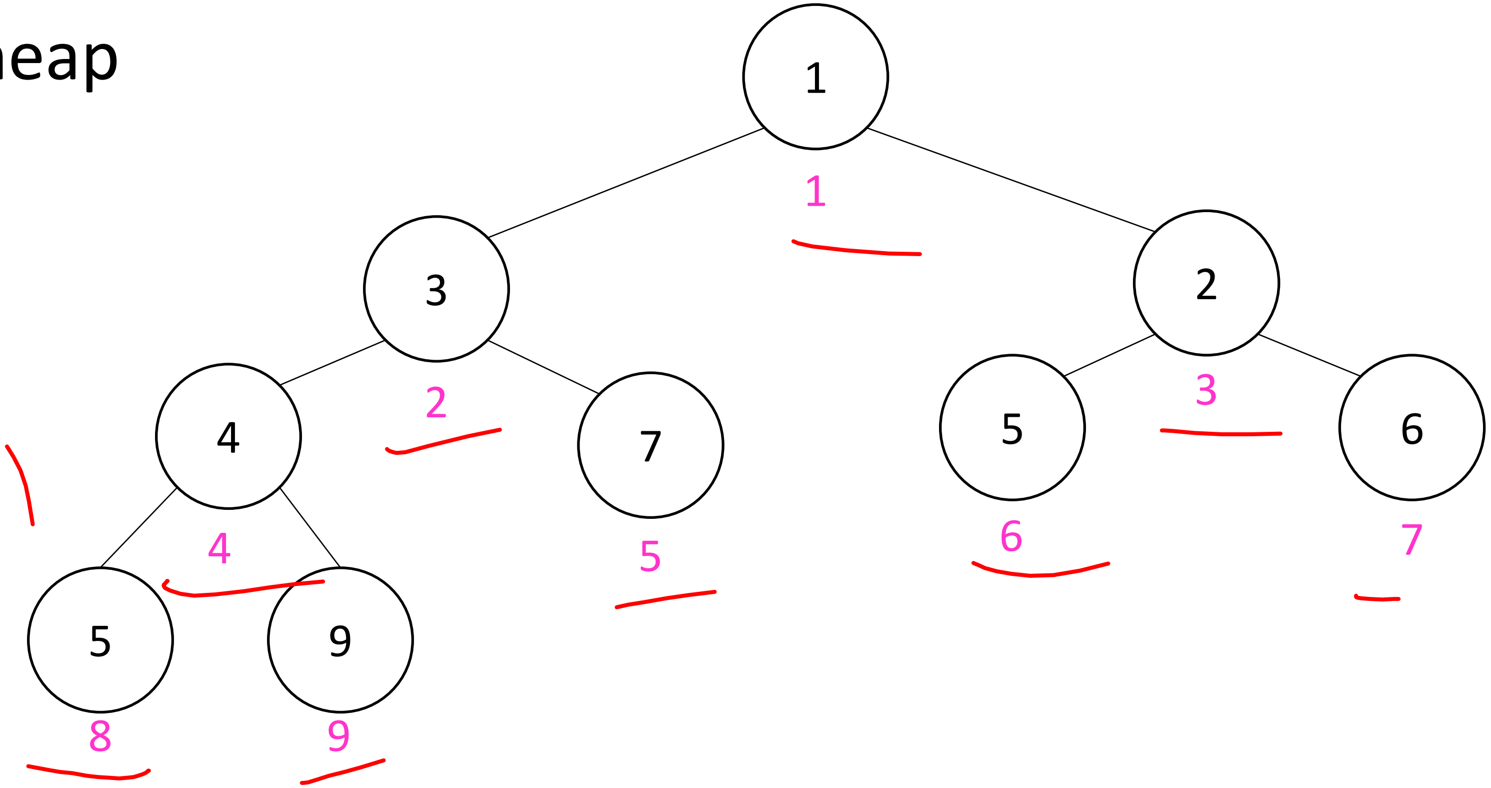
- Keep items in a complete binary tree
- Maintain the "(Min) Heap Property" of the tree
  - Every node's priority is ≤ its children's priority
  - Max Heap Property: every node's priority is ≥ its children



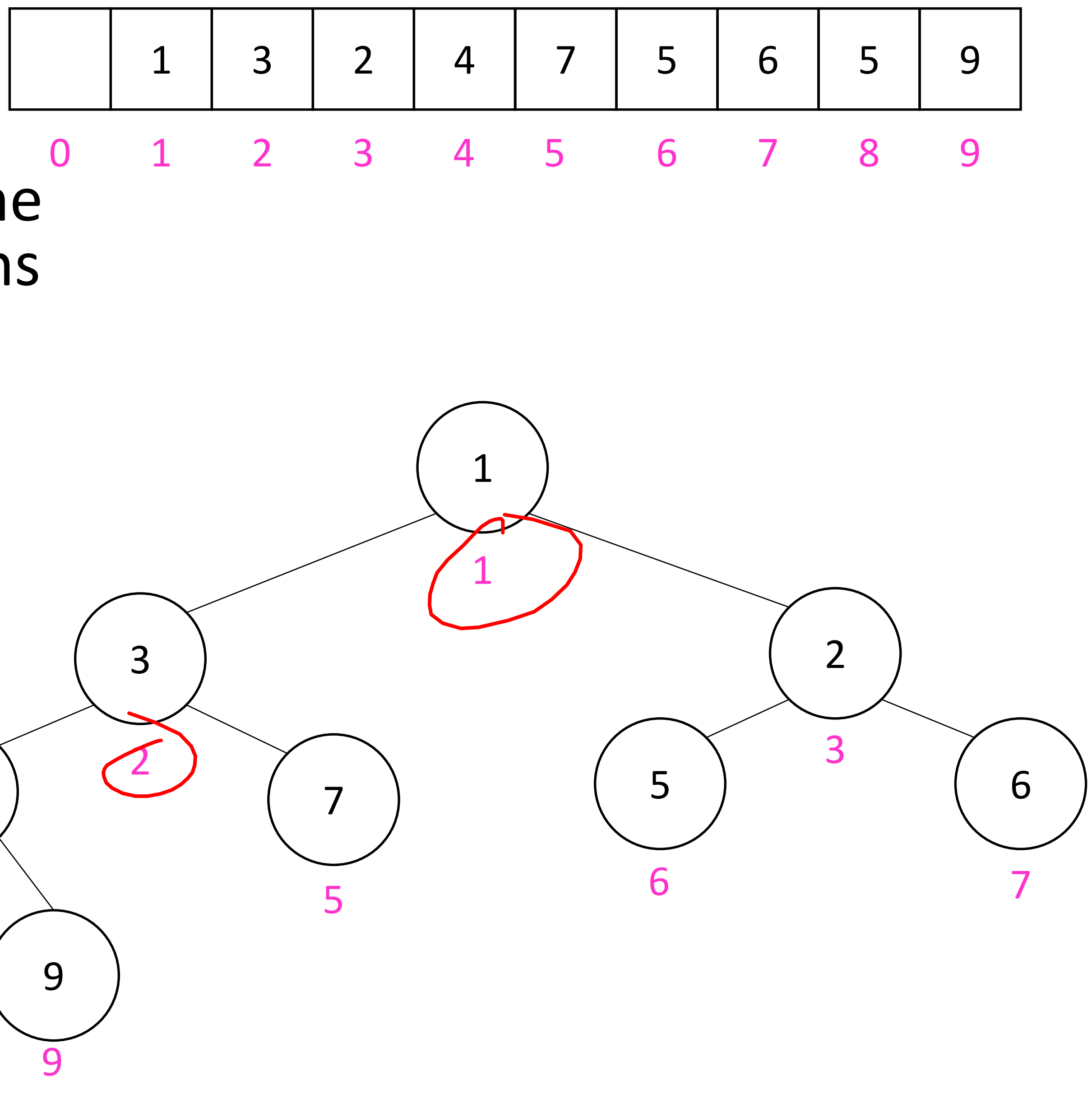
Representing a Heap

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

- Every complete binary tree with the same number of nodes uses the same positions and edges
- Use an array to represent the heap
- Index of root: 1
- Parent of node i: $\left\lfloor \frac{i}{2} \right\rfloor$
- Left child of node i: $2i$
- Right child of node i: $2i+1$
- Location of the leaves:



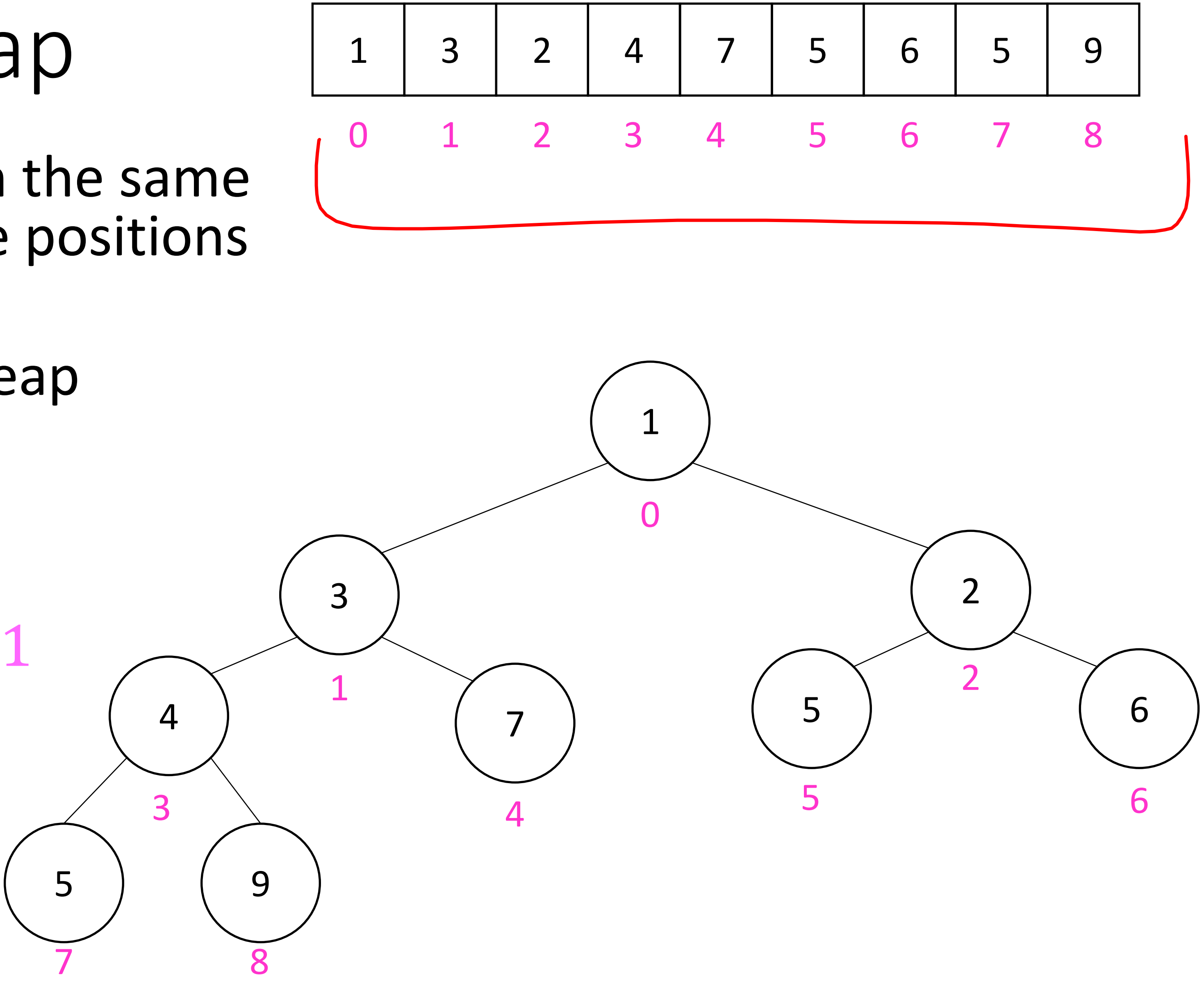
Representing a Heap



- Every complete binary tree with the same number of nodes uses the same positions and edges
- Use an array to represent the heap
- Index of root: 1
- Parent of node i: $\left\lfloor \frac{i}{2} \right\rfloor$
- Left child of node i: $2i$
- Right child of node i: $2i + 1$
- Location of the leaves: last $\left\lceil \frac{n}{2} \right\rceil$

Representing a Heap

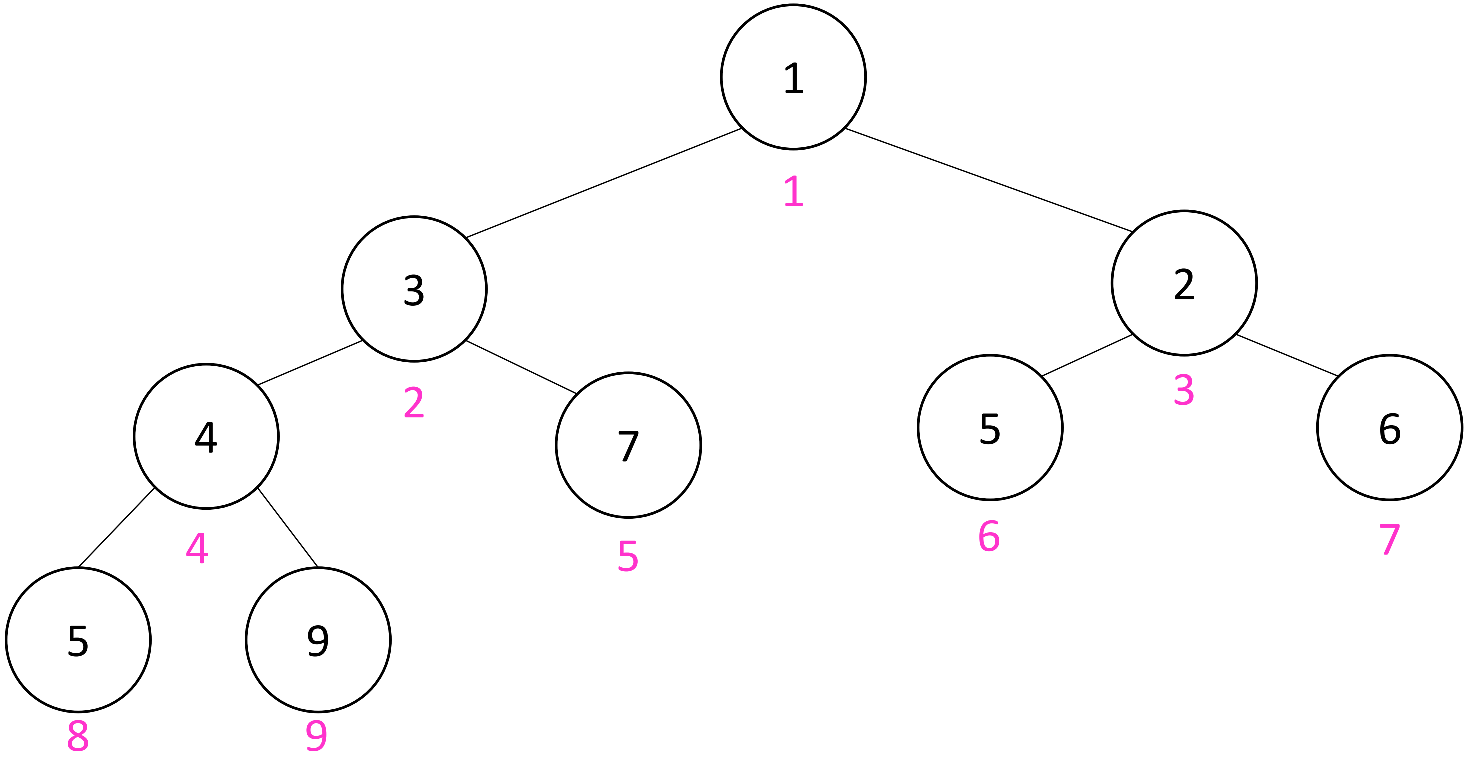
- Every complete binary tree with the same number of nodes uses the same positions and edges
- Use an array to represent the heap
- Index of root: 0
- Parent of node i: $\left\lfloor \frac{i+1}{2} \right\rfloor - 1$
- Left child of node i: $2(i+1)-1$
- Right child of node i: $2(i+1)$
- Location of the leaves: last $\left\lceil \frac{n}{2} \right\rceil$



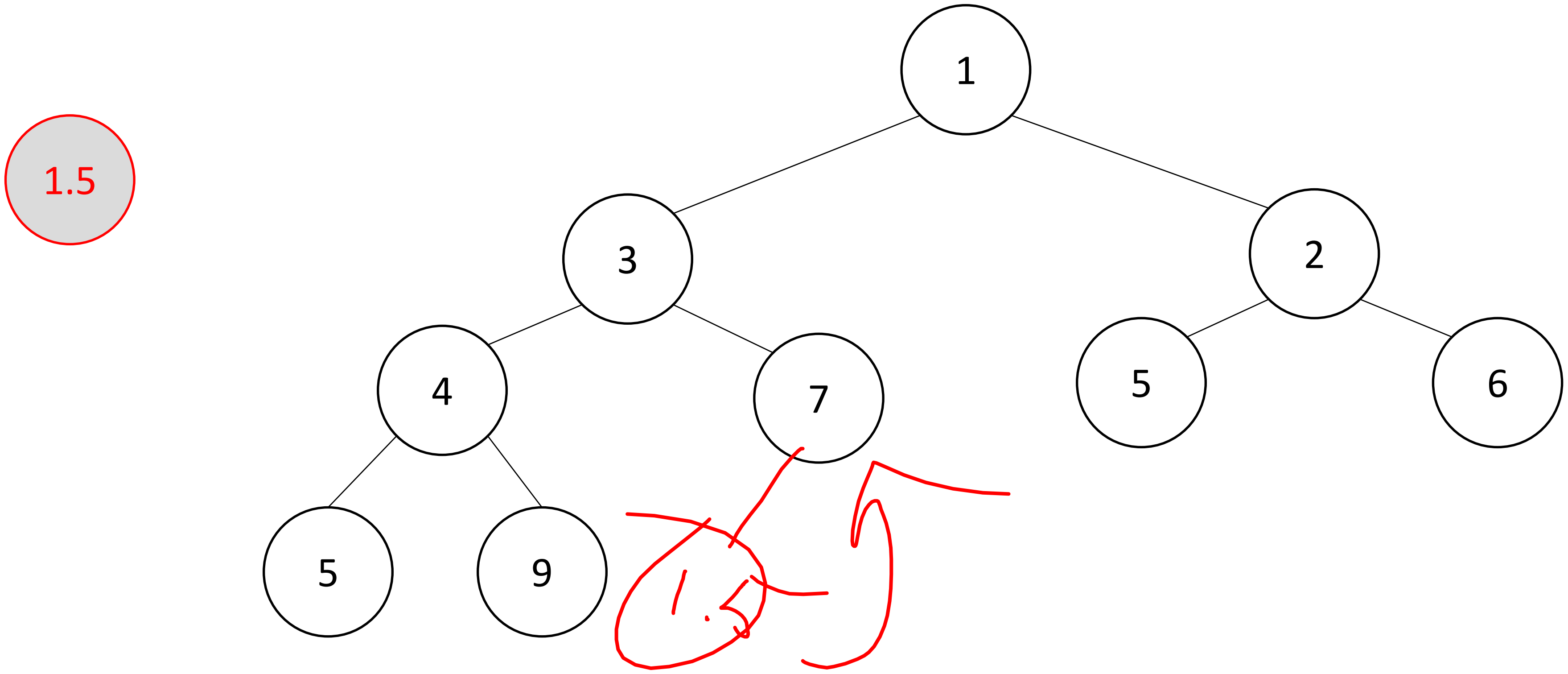
Insert Psuedocode

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

```
insert(item){
    if(size == arr.length – 1){resize();}
    size++;
    arr[i] = item;
    percolateUp(i)
}
```



Heap Insert



insert(item){

put item in the “next open” spot (keep tree complete)

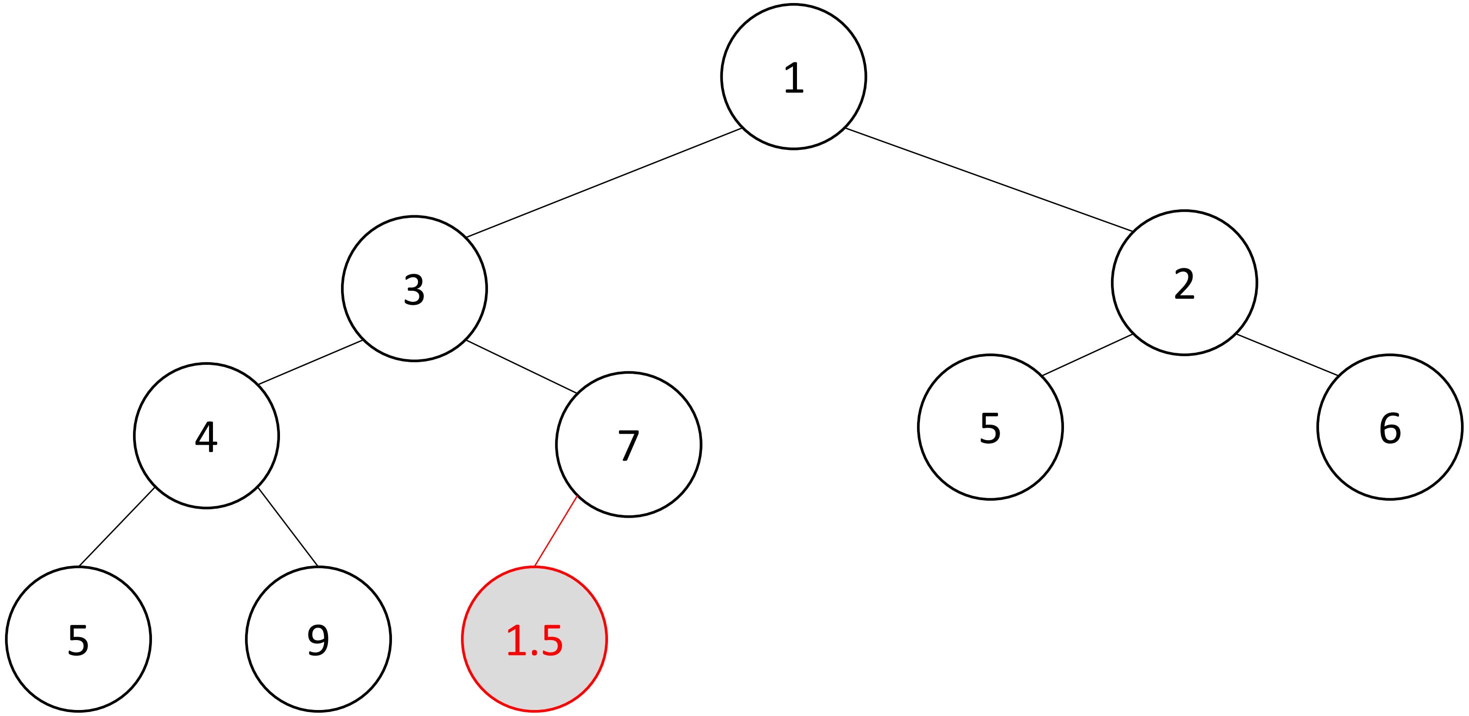
while (item.priority < parent(item).priority){

swap item with parent

}

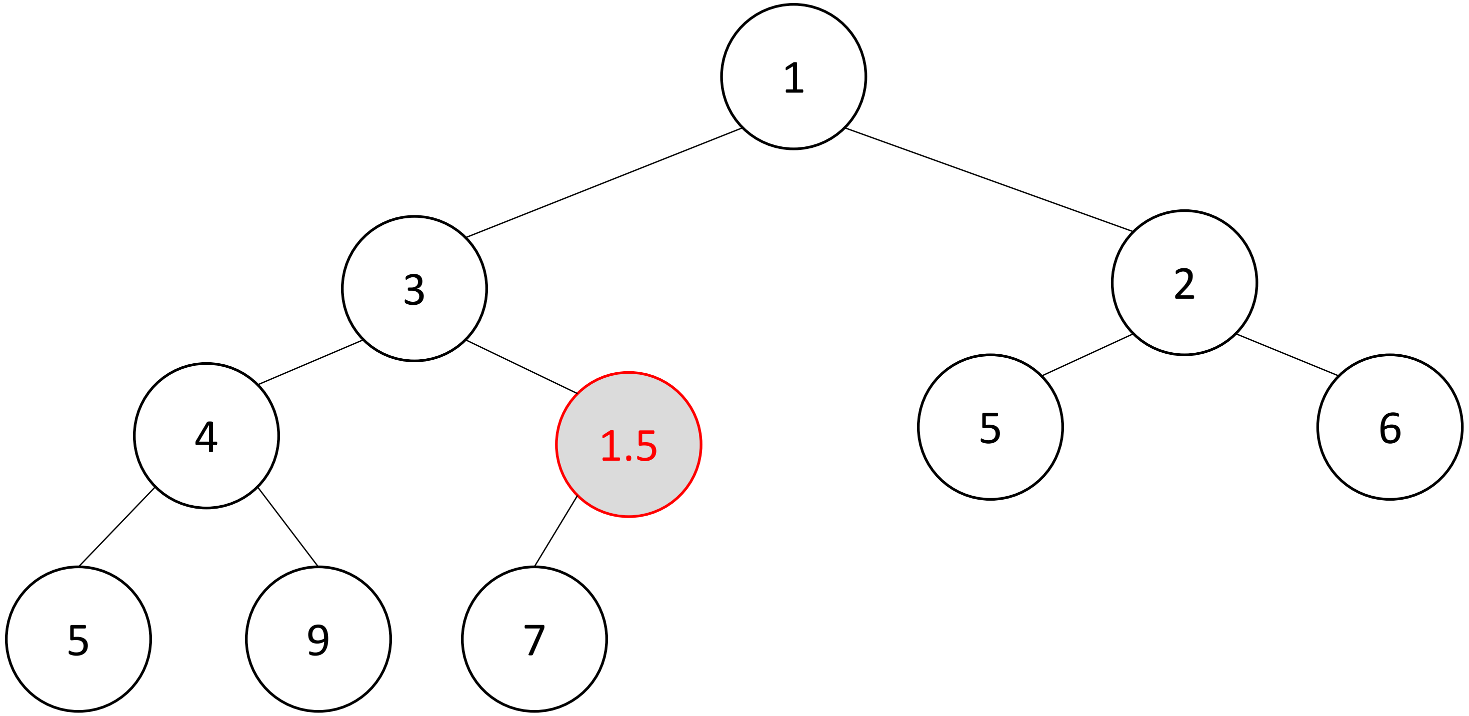
}

Heap Insert




```
insert(item){
    put item in the "next open" spot (keep tree complete)
    while (item.priority < parent(item).priority){
        swap item with parent
    }
}
```

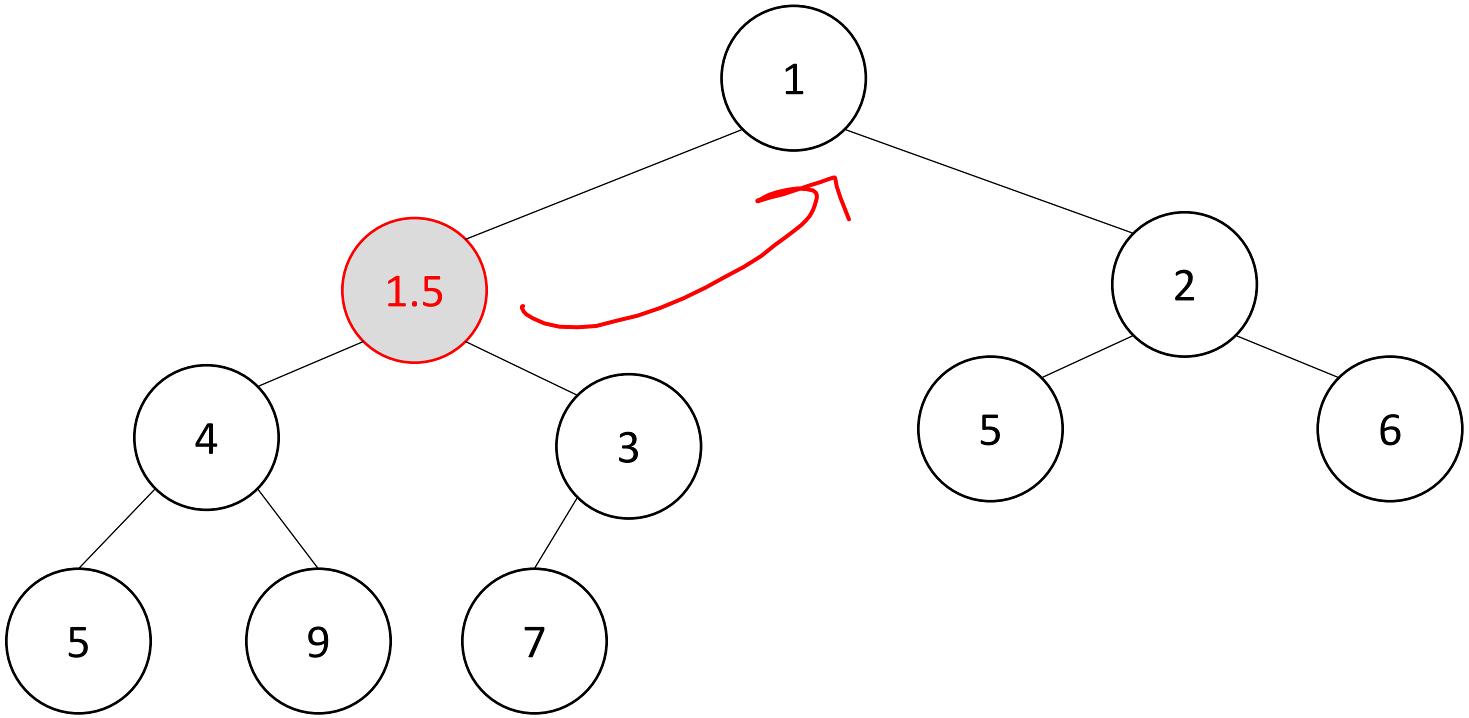
Heap Insert




```
insert(item){
    put item in the “next open” spot (keep tree complete)
    while (item.priority < parent(item).priority){
        swap item with parent
    }
}
```

Percolate Up

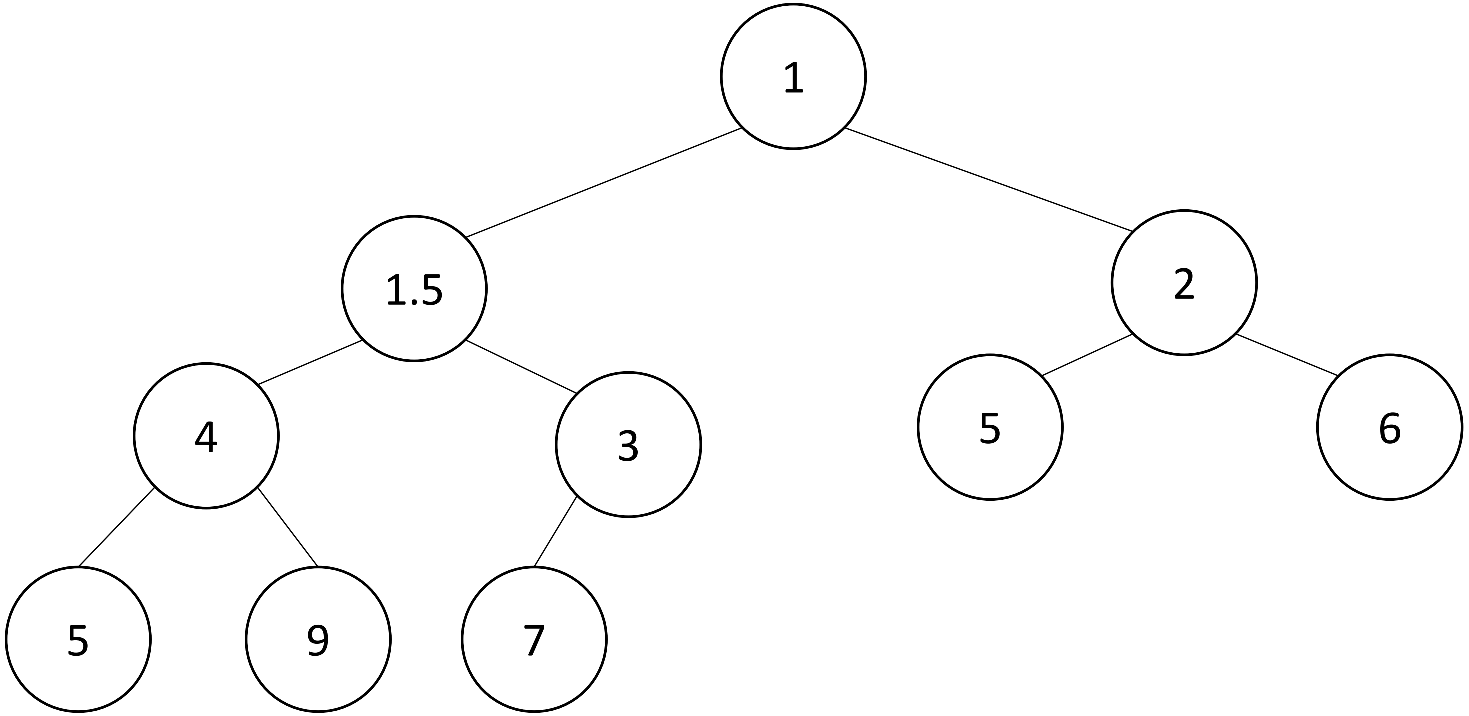
Heap Insert




```
insert(item){
    put item in the "next open" spot (keep tree complete)
    while (item.priority < parent(item).priority){
        swap item with parent
    }
}
```

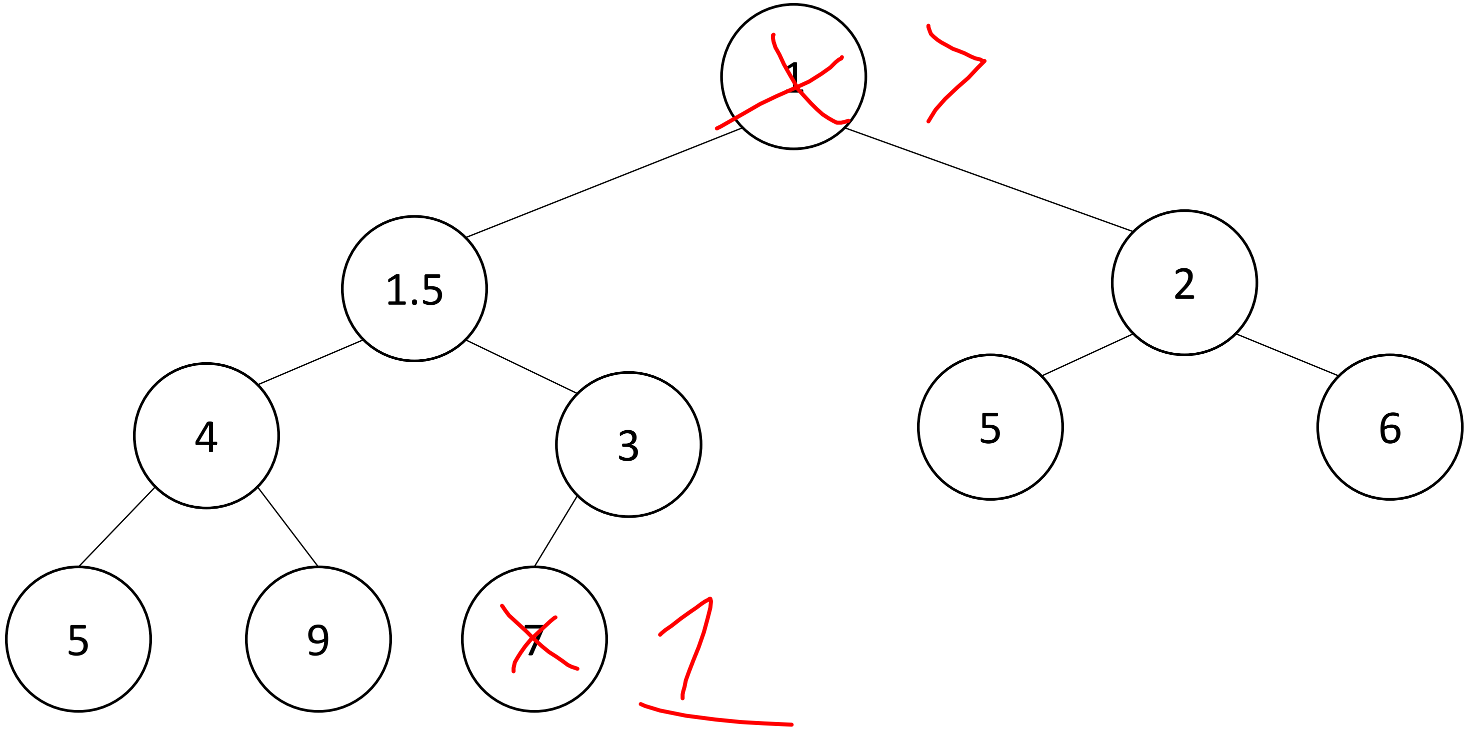
Percolate Up

Heap Insert



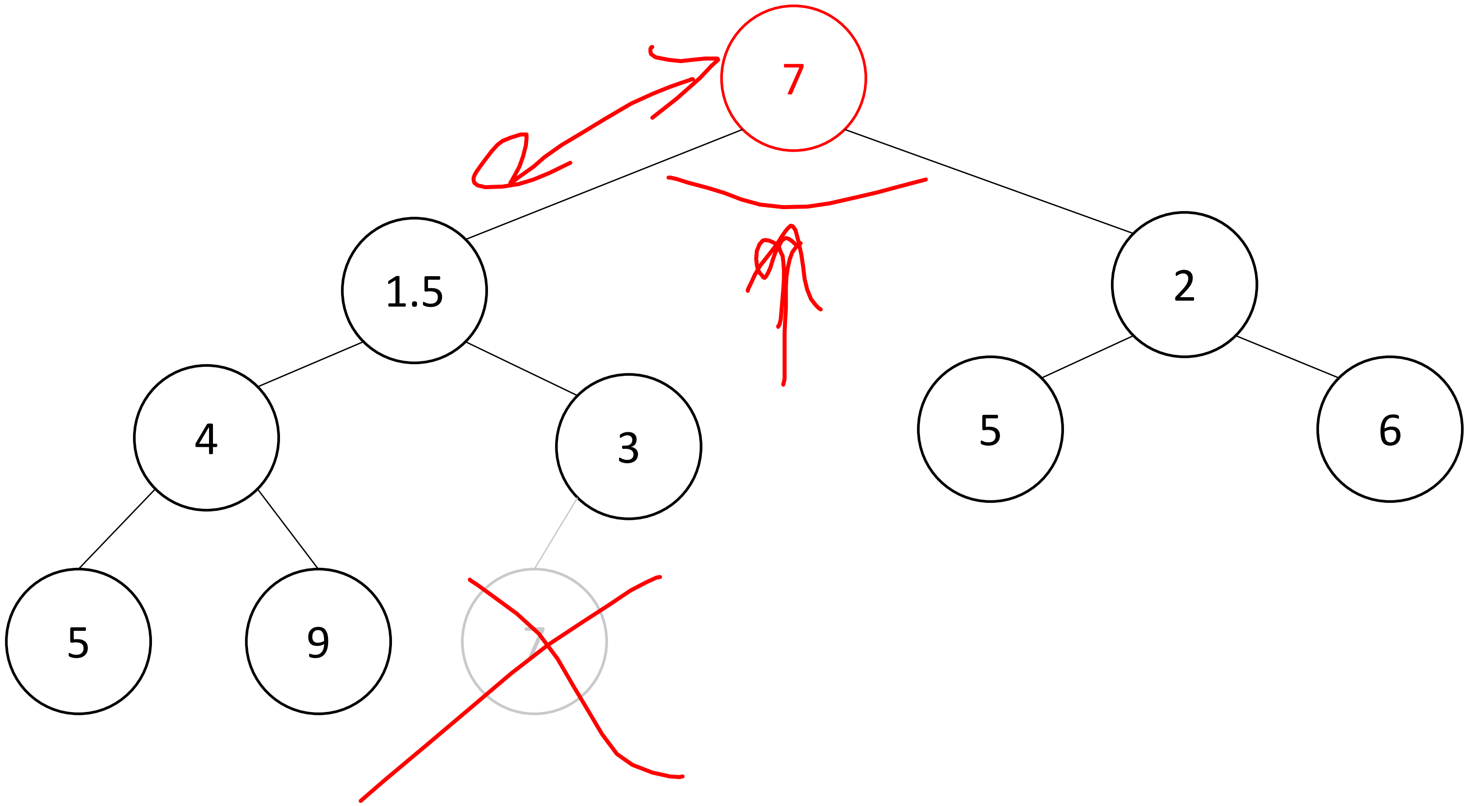

```
insert(item){
    put item in the "next open" spot (keep tree complete)
    while (item.priority < parent(item).priority){
        swap item with parent
    }
}
```

Heap deleteMin



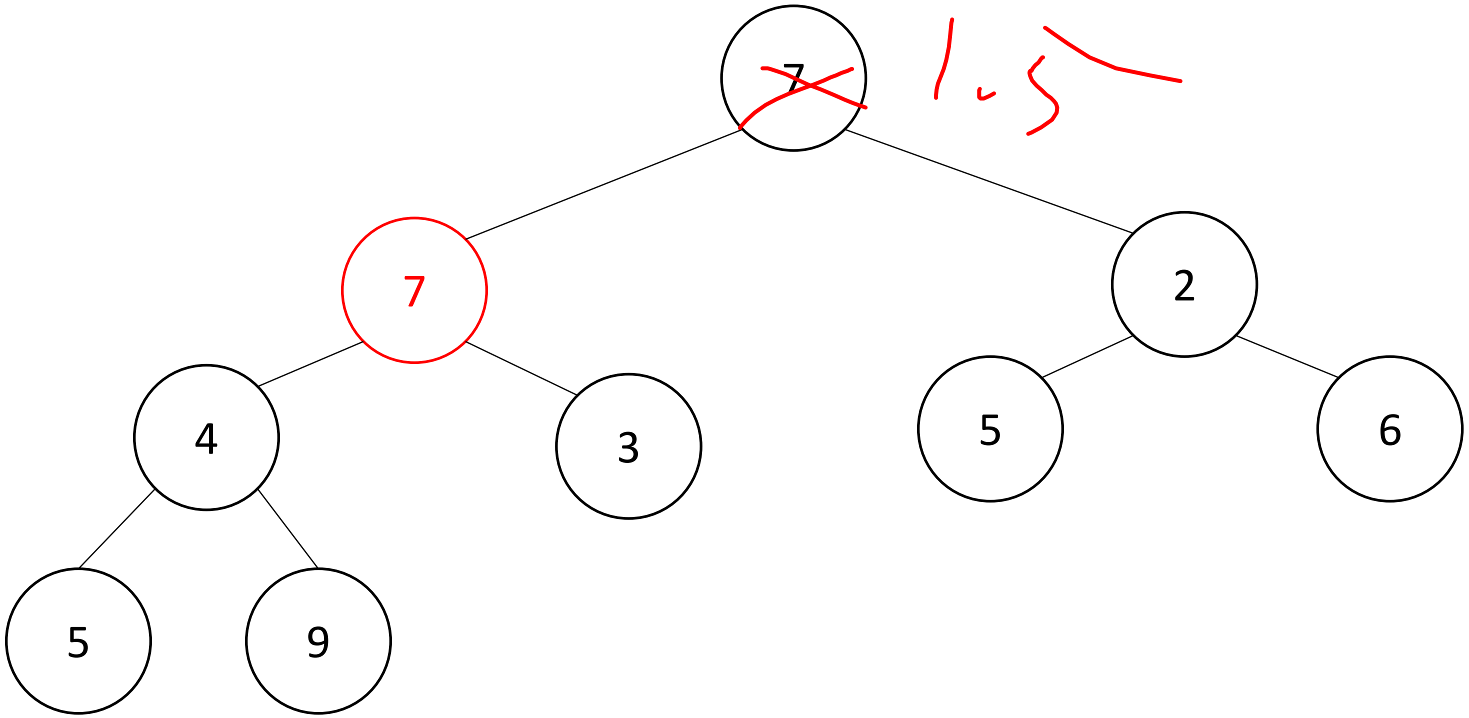

```
deleteMin(){
    min = root
    br = bottom-right item
    move br to the root
    while(br > either of its children){
        swap br with its smallest child
    }
    return min
}
```

Heap deleteMin




```
deleteMin(){
    min = root
    br = bottom-right item
    move br to the root
    while(br > either of its children){
        swap br with its smallest child
    }
    return min
}
```

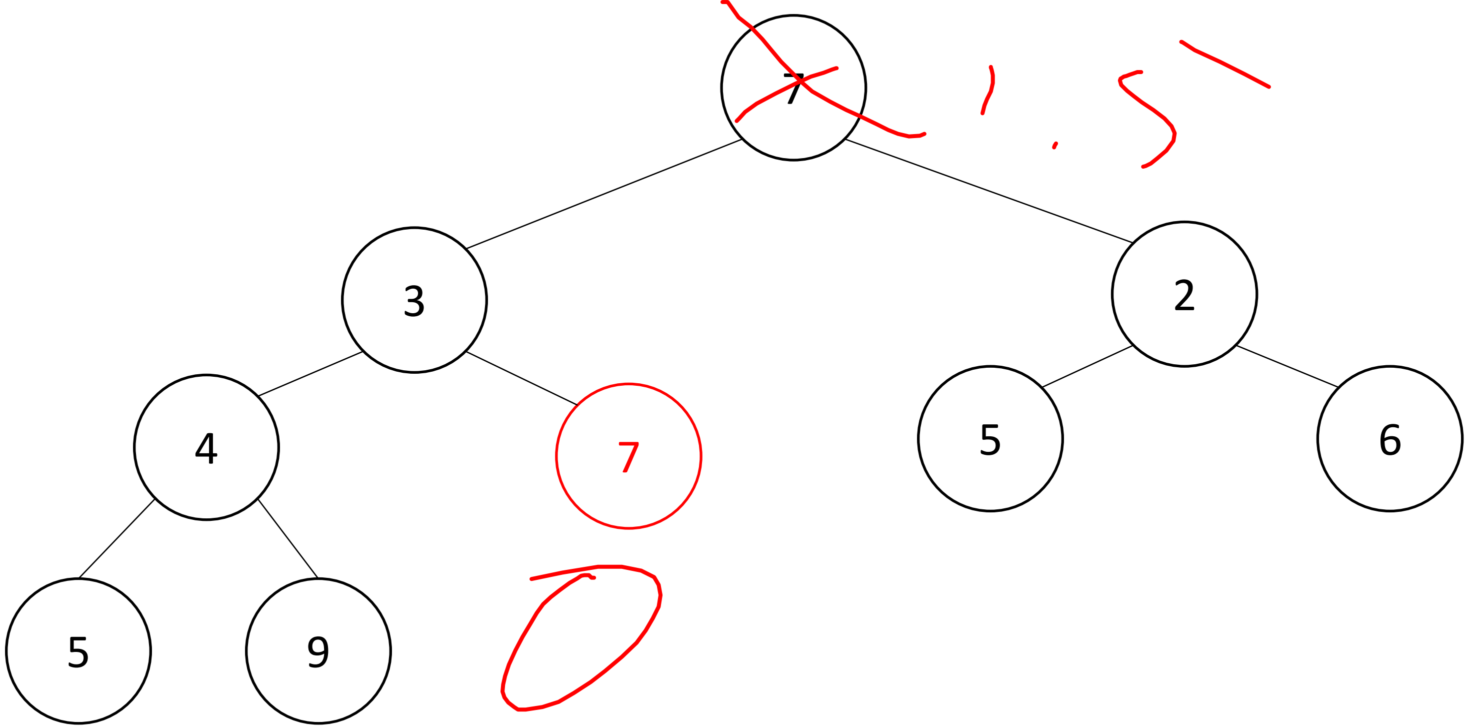
Heap deleteMin




```
deleteMin(){
    min = root
    br = bottom-right item
    move br to the root
    while(br > either of its children){
        swap br with its smallest child
    }
    return min
}
```

Percolate Down

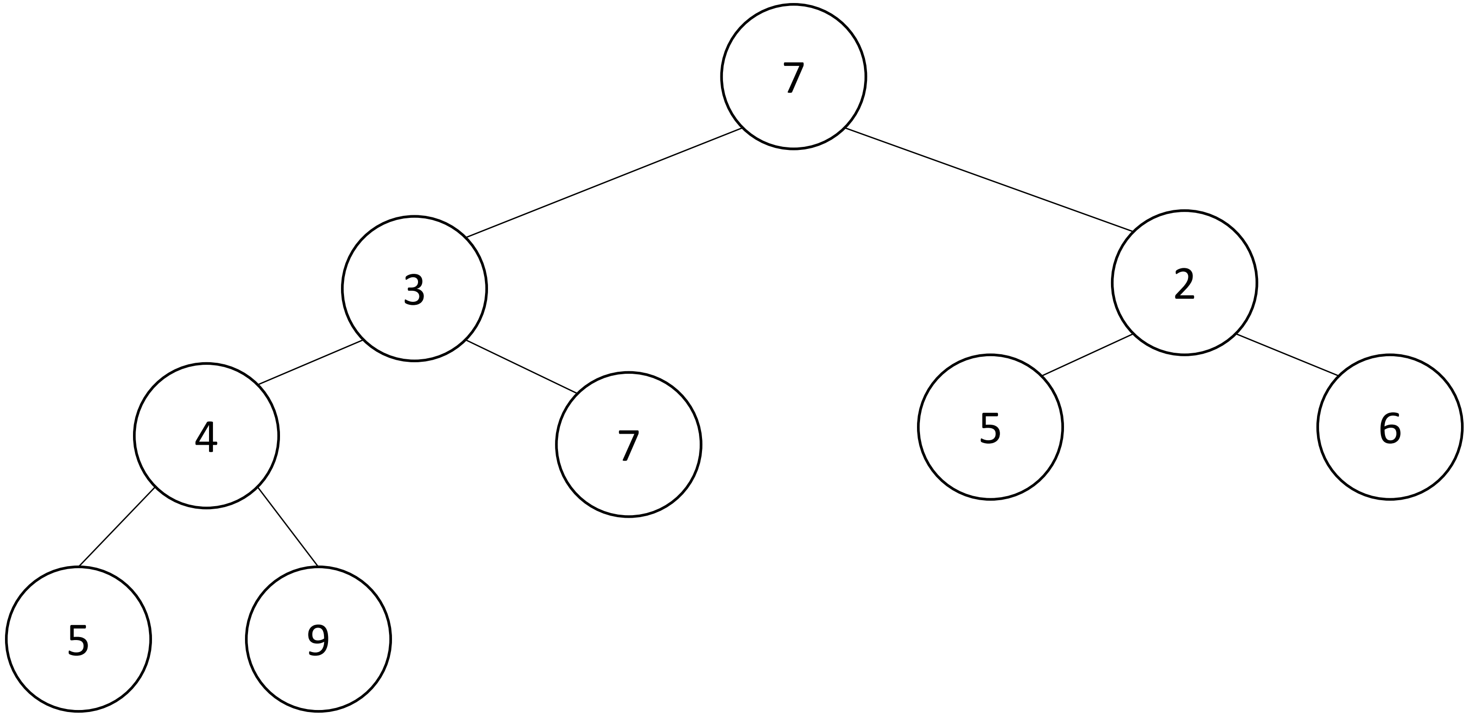
Heap deleteMin




```
deleteMin(){
    min = root
    br = bottom-right item
    move br to the root
    while(br > either of its children){
        swap br with its smallest child
    }
    return min
}
```

Percolate Down

Heap deleteMin




```
deleteMin(){
    min = root
    br = bottom-right item
    move br to the root
    while(br > either of its children){
        swap br with its smallest child
    }
    return min
}
```

Percolate Up and Down (for a Min Heap)

- Goal: restore the "Heap Property"
- Percolate Up:
  - Take a node that may be smaller than a parent, repeatedly swap with a parent until it is larger than its parent
- Percolate Down:
  - Take a node that may be larger than one of its children, repeatedly swap with smallest child until both children are larger
- Worst case running time of each:
  - $\Theta(\log n)$

Percolate Up

1: index

```
percolateUp(int i){
    int parent = i/2;  \\ index of parent
    Item val = arr[i];  \\ value at current location
    while(i > 1 && arr[i].priority < arr[parent].priority){  \\ until location is root or heap property holds
        arr[i] = arr[parent];  \\ move parent value to this location
        arr[parent] = val; \\ put current value into parent’s location
        i = parent;  \\ make current location the parent
        parent = i/2;  \\ update new parent
    }
}
```

DeleteMin Psuedocode

```
deleteMin(){
    theMin = arr[1];
    arr[1] = arr[size];
    size--;
    percolateDown(1);
    return theMin;
}
```

Percolate Down

```
percolateDown(int i){
  int left = i*2;  \\ index of left child
  int right = i*2+1;  \\ index of right child
  Item val = arr[i];  \\ value at location
  while(left <= size){  \\ until location is leaf
    int toSwap = right;
    if(right > size || arr[left].priority < arr[right] .priority){  \\ if there is no right child or if left child is smaller
      toSwap = left;  \\ swap with left
    } \\ now toSwap has the smaller of left/right, or left if right does not exist
    if (arr[toSwap] .priority < val.priority){  \\ if the smaller child is less than the current value
      arr[i] = arr[toSwap];
      arr[toSwap] = val; \\ swap parent with smaller child
      i = toSwap; \\ update current node to be smaller child
      left = i*2;
      right = i*2+1;
    }
    else{ return;} \\ if we don't swap, then heap property holds
  }
}
```

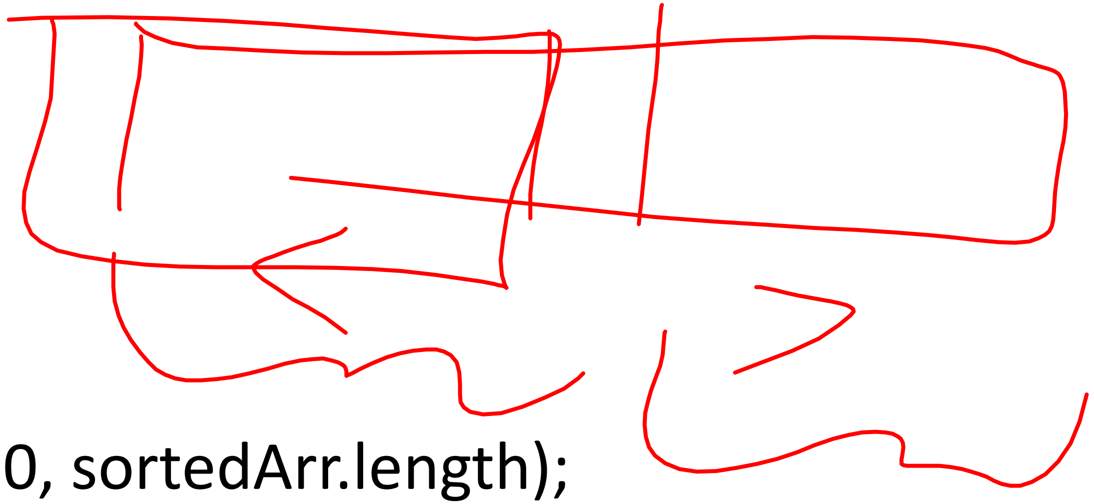
Other Operations

3 → 12

- Increase Key
  - Given the index of an item in the PQ, make its priority value larger
    - Min Heap: Then percolate down
    - Max Heap: Then percolate up
- Decrease Key
  - Given the index of an item in the PQ, make its priority value smaller
    - Min Heap: Then percolate up
    - Max Heap: Then percolate down
- Remove
  - Given the item at the given index from the PQ

3 → 1

$T(n) = c + T\left(\frac{n}{2}\right)$



Binary Search

```
search(value, sortedArr){
        return helper(value, sortedArr, 0, sortedArr.length);
}
helper(value, arr, low, high){
        if (low == high){ return false; }
        mid = (high + low) / 2;
        if (arr[mid] == value){ return true; }
        if (arr[mid] < value){ return helper(value, arr, mid+1, high); }
        else { return helper(value, arr, low, mid); }
}
```

$T(n)$

1

1

1

$T\left(\frac{n}{2}\right)$

Analysis of Recursive Algorithms

- Overall structure of recursion:
  - Do some non-recursive "work"
  - Do one or more recursive calls on some portion of your input
  - Do some more non-recursive "work"
  - Repeat until you reach a base case
- Running time: $T(n) = T(p_1) + T(p_2) + \cdots + T(p_x) + f(n)$
  - The time it takes to run the algorithm on an input of size n is:
  - The sum of how long it takes to run the same algorithm on each smaller input
  - Plus the total amount of non-recursive work done at that step
- Usually:
  - $T(n) = a \cdot T\left(\frac{n}{b}\right) + f(n)$
    - Called "divide and conquer"
  - $T(n) = T(n - c) + f(n)$
    - Called "chip and conquer"

How Efficient Is It?

- $T(n) = 1 + T\left(\left\lceil \frac{n}{2} \right\rceil\right)$
- Base case: $T(1) = 1$

$T(n)$ = "cost" of running the entire algorithm on an array of length n

Let's Solve the Recurrence!

$T(\frac{n}{2}) = 1 + T(\frac{n}{4})$

$T(1) = 1$

$T(n) = 1 + \cancel{T(n/2)}$

$1 + \cancel{T(n/4)}$

$1 + \cancel{T(n/8)}$

$\dots$

1

Substitute until $T(1)$

So $\log_2 n$ steps

after i subs, input is $\frac{n}{2^i}$

$$T(n) = \sum_{i=1}^{\log_2 n} 1 = \log_2 n$$

$2^{\log_2 n} = n$

$T(n) \in \Theta(\log n)$

Recursive Linear Search

```
search(value, list){
        if(list.isEmpty()){
                return false;
        {
        if (value == list[0]){
                return true;
        }
        list.remove(0);
        return search(value, list);
}
```

1

T

$T(n) = 1 + T(n-1)$

$T(1) = 1$

$T(n-1)$

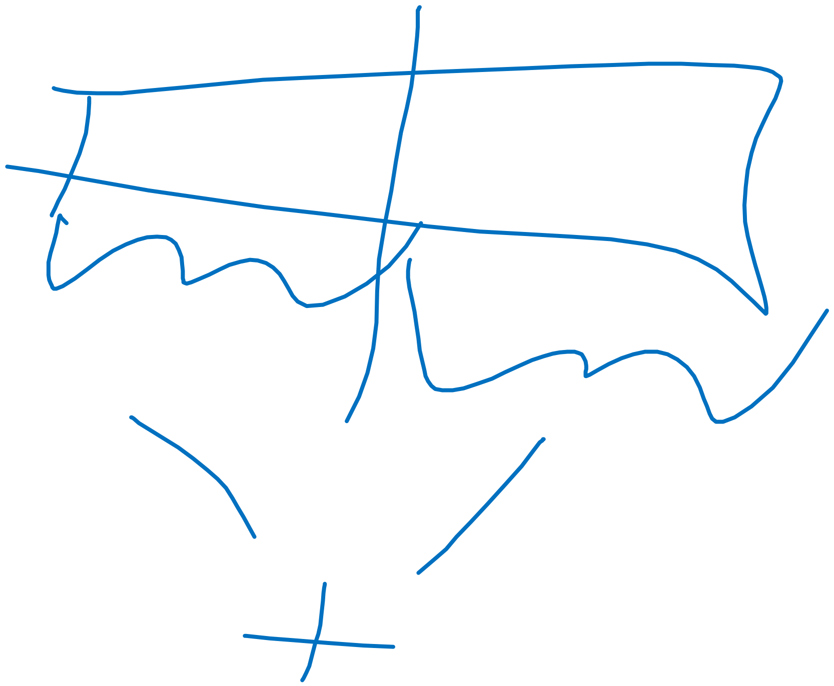
Unrolling Method

- Repeatedly substitute the recursive part of the recurrence
- $\boxed{T(n) = T(n-1) + c}$
- $T(n) = T(n-2) + c + c$
- $T(n) = T(n-3) + c + c + c$
- …
- $T(n) = c + c + c + \cdots + c$
  - How many c's? $\sim n$

$\Theta(n)$

$$T(n) = C + 2T\left(\frac{n}{2}\right)$$

Recursive List Summation



```
sum(list){
        return sum_helper(list, 0, list.size);
}
sum_helper(list, low, high){
        if (low == high){ return 0; }
        if (low == high-1){ return list[low]; }
        middle = (high+low)/2;
        return sum_helper(list, low, middle) + sum_helper(list, middle, high);
}
```

C

$T\left(\frac{n}{2}\right) \quad T\left(\frac{n}{2}\right)$

Loop Unrolling Method

- $T(n) = 2T\left(\frac{n}{2}\right) + c$

$$2\left(2T\left(\frac{n}{4}\right) + c\right) + c$$

$$= 4T\left(\frac{n}{4}\right) + 3c$$

Loop Unrolling Method

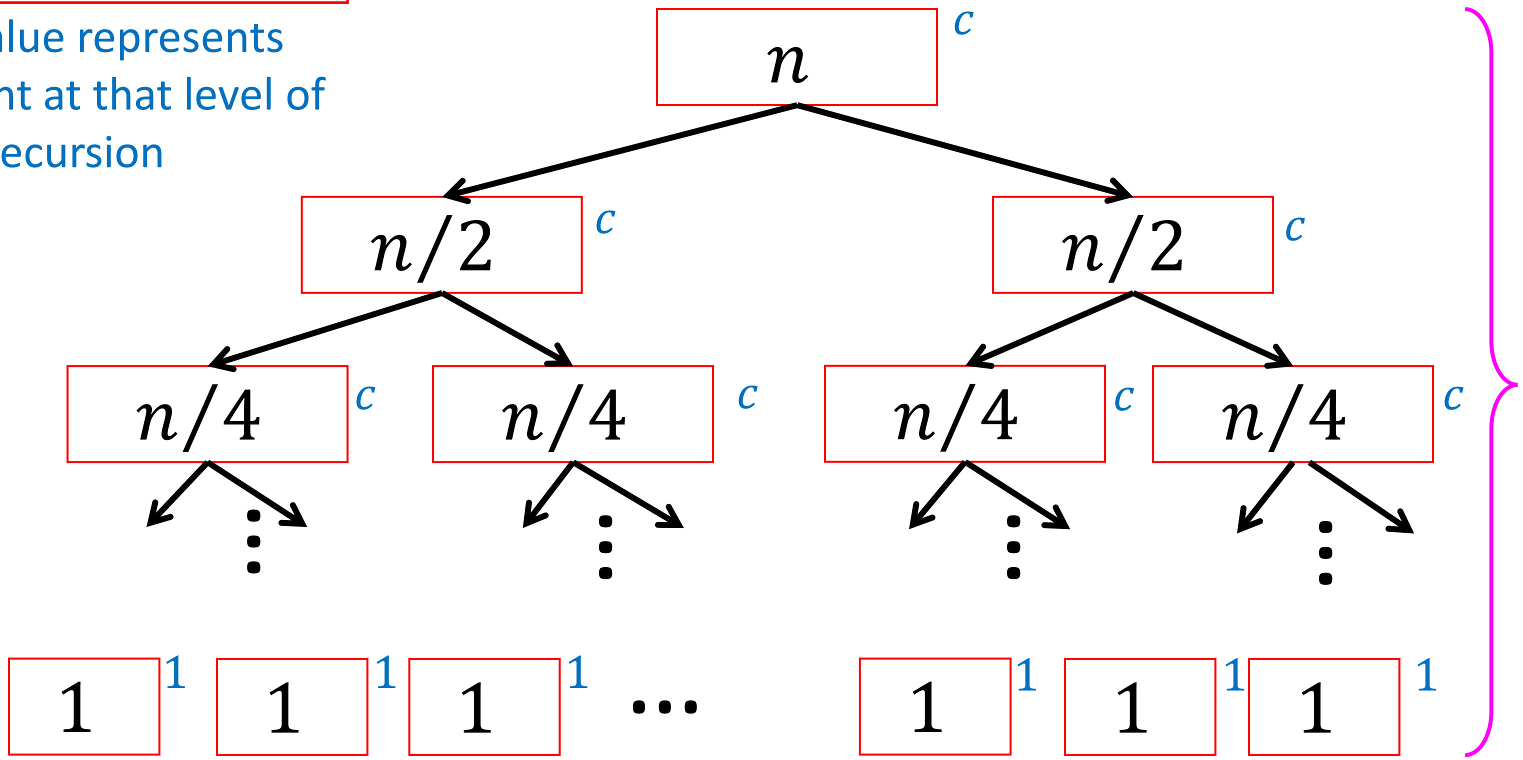
- $T(n) = 2T\left(\frac{n}{2}\right) + c$
- $T(n) = 2\left(2T\left(\frac{n}{4}\right) + c\right) + c = 4T\left(\frac{n}{4}\right) + 3c$
- $T(n) = 4\left(2T\left(\frac{n}{8}\right) + c\right) + 3c = 8T\left(\frac{n}{8}\right) + 7c$
- …after $i - 1$ substitutions
- $T(n) = 2^i T\left(\frac{n}{2^i}\right) + \left(2^i - 1\right)c$
  - $T\left(\frac{n}{2^i}\right) = T(1)$ when $i = \log_2 n$
- $T(n) = 2^{\log_2 n} T(1) + \left(2^{\log_2 n} - 1\right)c = n \cdot c_0 + cn - c = \Theta(n)$

Tree Method

Red box represents a problem instance

Blue value represents time spent at that level of recursion

$$T(n) = 2T\left(\frac{n}{2}\right) + c$$



$\Rightarrow 2^i \cdot c$ work per level

$\log_2 n$ levels of recursion

$$T(n) = \sum_{i=1}^{\log_2 n} 2^i \cdot c$$

Recursive List Summation

$$T(n) = \sum_{i=1}^{\log_2 n} 2^i \cdot c$$

$$= c \cdot \sum_{i=1}^{\log_2 n} 2^i$$

$$= c\left(\frac{1 - 2^{\log_2 n}}{1 - 2}\right)$$