Tutorial 04 - Midterm Test, PQ ADT

CS2040C Semester 2 2018/2019

By Jin Zhe, adapted from slides by Ranald+Si Jie, AY1819 S2 Tutor

Midterm Solutions

Question A: Solution using `find` and `substr`

```
string call_as(string Singaporean_name) {
    auto space_1 = Singaporean_name.find(" ");              // position of 1st space
    auto space_2 = Singaporean_name.find(" ", space_1 + 1); // position of 2nd space
    /* If name has 2 words */
    if (space_2 == string::npos) {
        return Singaporean_name;
    }
    auto space_3 = Singaporean_name.find(" ", space_2 + 1); // position of 3rd space
    /* If name has 3 words */
    if (space_3 == string::npos) {
        return Singaporean_name.substr(space_1 + 1);
    }
    /* If name has 4 words */
    return Singaporean_name.substr(space_3 + 1) + " " + Singaporean_name.substr(0, space_1);
}
```

Complexity: $O(1)$

Question B: `inClassRoster`

```
int inClassRoster(string studentName) {
    int index = lower_bound(roster.begin(), roster.end(), studentName) - index.begin();
    if (index >= roster.size() || roster[index] != studentName)
        return -1;
    return index;
}
```

Complexity: $O(\log N)$

There is no need to manually implement binary search if you are familiar with `lower_bound`

Question B: `addStudent`

```
void addStudent(string studentName) {
    /* Do nothing if student name is already in roster */
    if (inClassRoster(studentName) != -1) return;

    /* Use lower_bound to get index for insertion */
    int index = lower_bound(roster.begin(), roster.end(), studentName) - roster.begin();
    roster.insert(index, studentName); // O(N)
}
```

Complexity: $O(N)$

Alternatively, just iterate through roster and compare strings until you find the right position and insert there. There's no need to do binary search, since insertion will be $O(N)$ anyway.

Question B: `dropStudent`

Analogous to `addStudent`.

Question C.1

For those who are not familiar with comparators, it’s time to learn it now! Here is a quick explanation of what they do:

A comparator function is a function that takes in 2 parameters, say `a` and `b`, which simply returns a `bool` that answers the question: must `a` come before `b` after sorting? Conceptually, the comparator function is simply a binary function that attempts to model the default `<` relationship between `a` and `b`, where it returns `true` iff `a < b` and `false` otherwise (`>=`). In other words, if the comparator returns `false` for a given pair of `a` and `b`, it's saying that we don't care about their relative order. i.e. if a case in your comparator wants to treat `a` and `b` as non-distinct (i.e. equal), you should return `false` for that case.

Question C.1: Comparator

```
bool cmp(int a, int b) {
    /* Case 1: Both even */
    if (a%2 == 0 && b%2 == 0)
        return a > b;   // order descending
    /* Case 2: a even, b odd */
    if (a%2 == 0 && b%2 != 0)
        return true;    // order a before b
    /* Case 3: a odd, b odd */
    if (a%2 != 0 && b%2 != 0)
        return a < b;   // order ascending
    /* Case 4: a odd, b even */
    if (a%2 != 0 && b%2 == 0)
        return false;   // do not order a before b
}
```

Question C.1: Comparator

A slightly more elegant comparator!

```
bool cmp(int a, int b) {
    /* If different parity */
    if (a%2 != b%2)
        return a%2 < b%2;    // order even before odd
    /* Else if both even */
    else if (a%2 == 0)
        return a > b;        // order descending
    /* Else if both odd */
    else
        return a < b;        // order ascending
}
```

Question C.1: Comparator

A much more concise comparator, albeit cryptic!

If you have truly understood comparator functions, you should understand this solution.

```
bool cmp(int a, int b) {
    return
        (a%2 != b%2 && a%2 < b%2)         ||
        (a%2 == 0 && b%2 == 0 && a > b)   ||
        (a%2 != 0 && b%2 != 0 && a < b);
}
```

Question C.2

```
int main() {
    int n; cin >> n; cin.get();
    vector<string> TA(n);
    for (int i = 0; i < n; i++)
        getline(cin, TA[i]);
    stable_sort(TA.begin(), TA.end(), [](string a, string b) {
        return call_as(a) < call_as(b);
    });
    for (auto &v : TA)
        cout << call_as(v) << " (" << v << ")" << endl;
    return 0;
}
```

Question E

In the next few slides, a general strategy is proposed for tackling such questions

Question E

Let's approach the question by first writing out the pseudocode of enqueue and dequeue methods

`RLEnqueue(int ID)`:

```
1. Find the team the person belongs to given his ID
2. Check if team is in RLQ
3. If team not in RLQ, enqueue ID into team queue and enqueue team queue into RLQ
```

`RLDequeue(int ID)`:

```
1. Find first team queue in RLQ and dequeue frontmost person ID
2. If team queue is now empty, dequeue team queue from RLQ
3. Return person ID
```

Question E

Given the pseudocode, we can identify 3 DSes that are required:

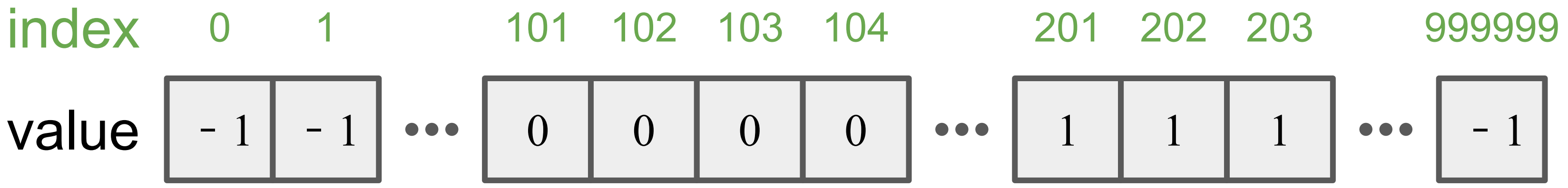
- One for storing the association between team id and person id
    - Needed by `RLEnqueue` step 1
- One for representing a team in RLQ
    - Needed by `RLEnqueue` steps 1, 2
    - Needed by `RLDequeue` steps 1, 2
- One for representing RLQ
    - Needed by `RLEnqueue` steps 2, 3
    - Needed by `RLDequeue` steps 1, 2

Ideally, these DSes should fulfill the steps in the most optimal manner

Question E

For storing the association between team id and person id

- We shall simply use a vector where for a member element, the index corresponds to person id and the element value correspond to team id
- This scheme known as a *map* where we store key-value pairs. More on this in the future lectures when we learn about Hashmaps
- Let’s call it `person_id_to_team_id` since that is what it maps
- Note that we actually have to cover the entire range of possible IDs!
- We can actually do one more layer for mapping explicit person ids to implicit ones to avoid having to deal with such a huge range, but we’ll leave that out for sake of brevity



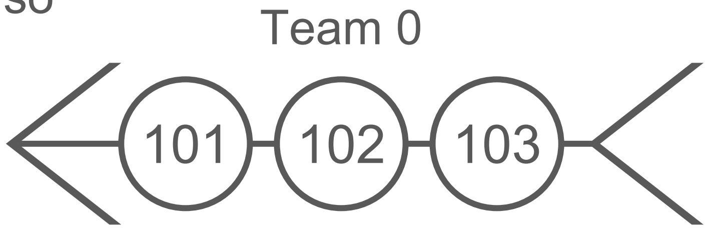
Populating the `person_id_to_team_id` map using given example in question

Question E

For representing a team in RLQ

- It is natural to represent it using a queue DS since we have to maintain FIFO ordering of members within each team
- Let’s call it a `team_queue`

For the next couple of slides, I shall represent a queue in drawing using a double-ended arrow like so



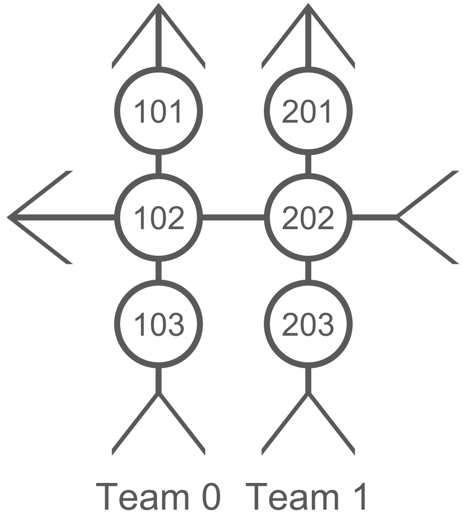
Using given example in question, the diagram above illustrates team 0’s `team_queue` just before any RLQ dequeuing

Question E

For representing RLQ

- Once again it is natural to represent it using a queue DS since we have to maintain FIFO ordering of teams that are in the RLQ
- Hence the natural thinking is to use a queue of queues
- Let's just name it RLQ as we called it

Using given example in question, the diagram on the right illustrates how the RLQ looks just before any dequeuing takes place



Question E

Based on the previous proposals for the 3 DSes we need, we are able to achieve $O(1)$ for every step in both `RLEnqueue` and `RLDequeue` except step 2 in `RLEnqueue`.

Step 2 in `RLEnqueue` will be $O(T)$ because we'll have to cycle through all teams in the RLQ to check if the given person's team is already in the RLQ.

This was the original solution, until TA Lin Si Jie proposed a better idea! :O

Question E

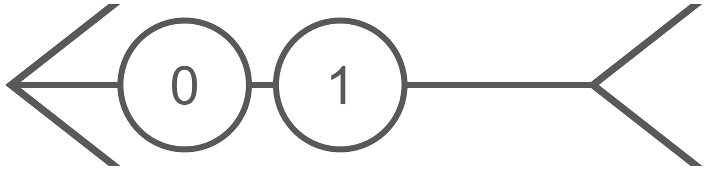
Although it is natural to model RLQ as a queue of queues, we can actually “decompose” it down into:

1. A queue of team ids
2. A map of team id to team queues

Question E

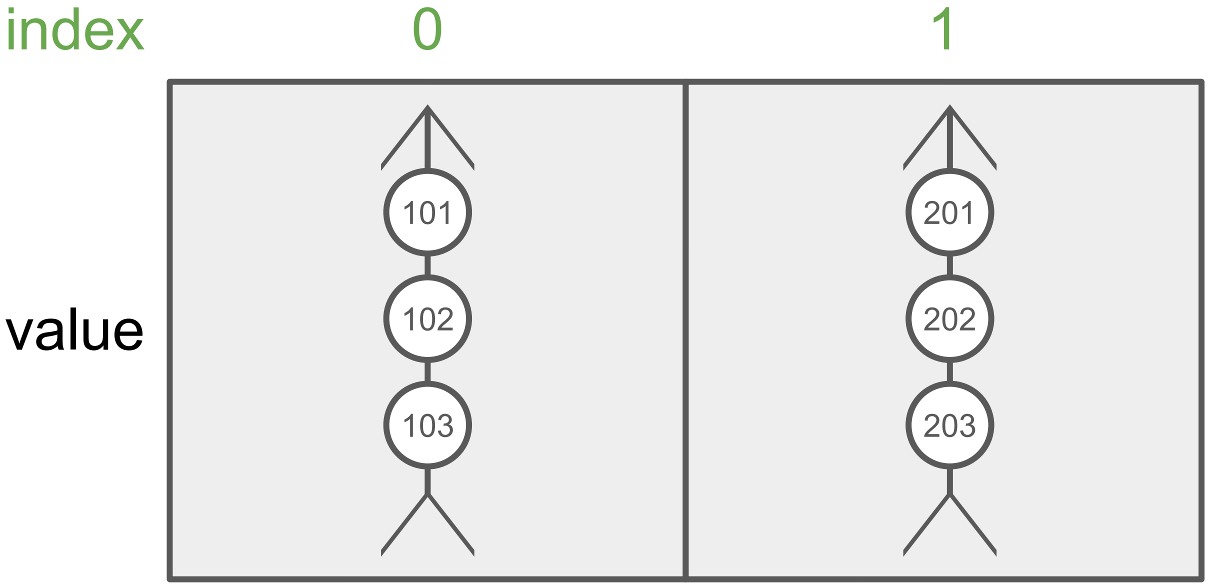
Our RLQ now simply becomes a queue of team ids

We only enqueue a person’s team id if his team is not already inside the RLQ!



Question E

We now need to maintain an additonal team id to team queue map, which we shall call `team_id_to_team_queue`



Question E

Realize that now in order to check if a person’s team is already in the RLQ, we simply need to retrieve his team queue from the `team_id_to_team_queue` map in $O(1)$ and check

- If it’s empty then the team isn’t in RLQ yet
- Otherwise it must already be in the RLQ

Basic Binary Heap Stuffs

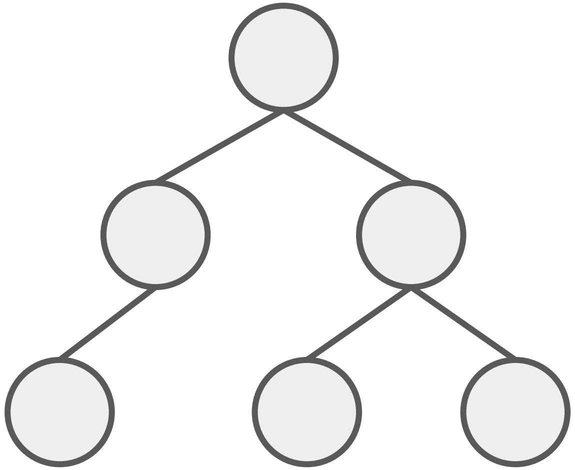
Priority Queue (PQ) ADT

Common PQ ADT operations

`insert(v)`	Insert item with value `v` into the PQ.
`ExtractMax()`	Return the the item with highest priority from PQ.
`create(A)`	Create a PQ from the given array `A`.
`HeapSort(A)`	Return the sorted order of items in given array `A`.

Binary tree

Example of a binary tree



A binary tree **is a tree** where every vertex in the tree

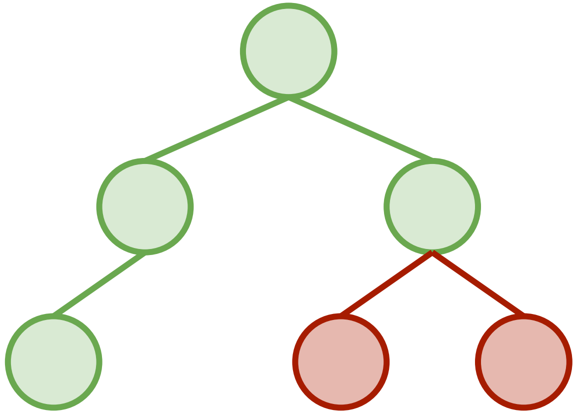
- has at most 2 children (therefore *binary*)
- is itself a binary *subtree* rooted at that vertex

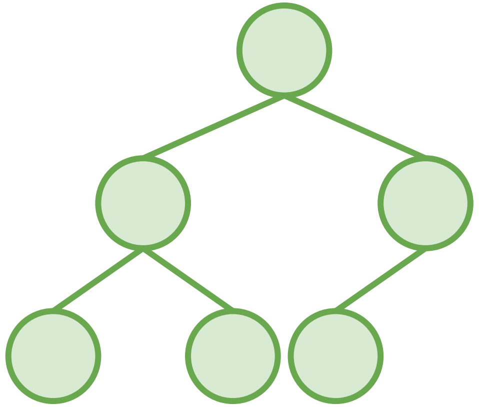
The **minimum height** of a binary tree with a total of N vertices is $O(\log N)$. Why?

Complete binary tree

A *complete binary tree* **is a binary tree**, where for every level, except possibly the last, is completely filled, and all nodes in the last level are **as far left as possible**.

Note that the height of a complete binary tree with N vertices is $O(\log N)$

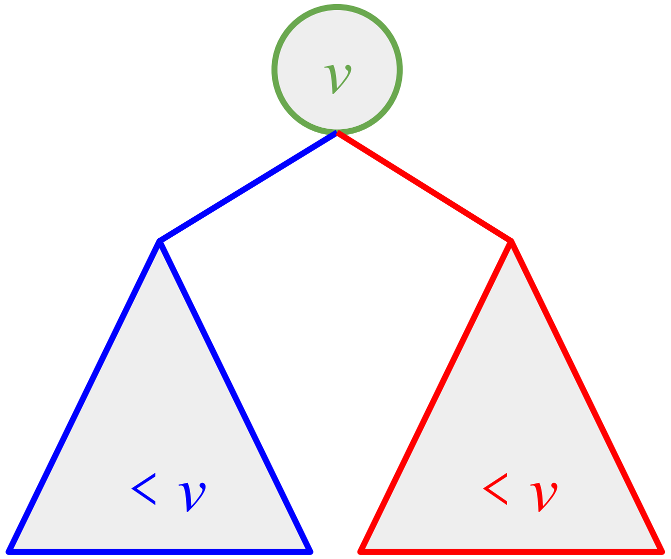

Not a complete binary tree


A complete binary tree

Binary (Max) heap

- A *binary heap* is one common implementation for PQ ADT
- **It is a complete binary tree**
- Therefore every vertex in a binary heap is itself also a binary heap (with the root being the vertex)
- *Max-heap property*: The value of every vertex in a binary max-heap is greater than every value of its children. i.e. the root's value is the maximum of all values in the heap
- *Min-heap property* is just the reverse of above
- Note: we ignore duplicate values for now (i.e ignore equality)

A vertex in a binary max heap



Binary heap data-structure

Implement using an **array** in breadth-first order as implicit representation!

In the next slide, we will use

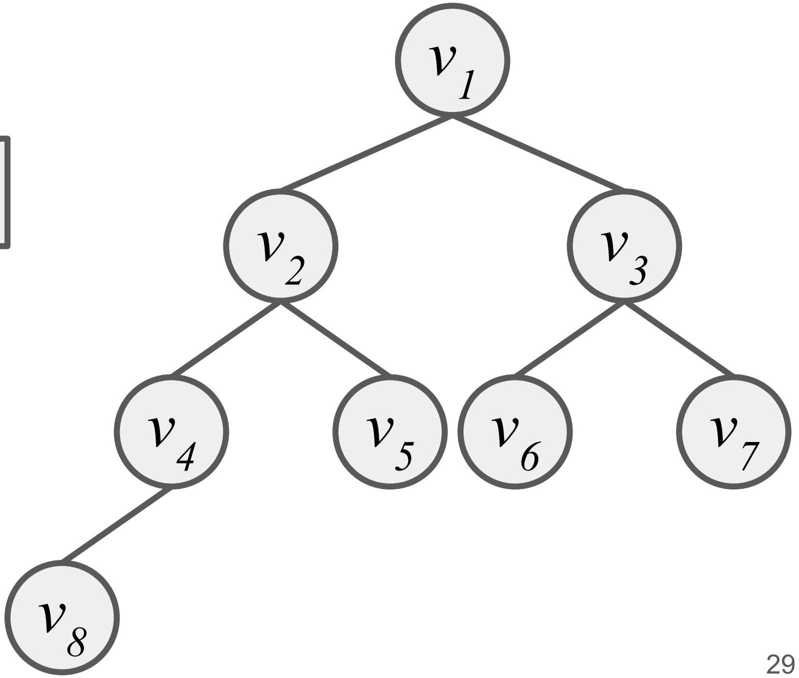
- v_i to represent value at index i
- L_i to represent left child index of index i
- R_i to represent right child index of index i
- $P_{i,j}$ to represent parent index of index i and j

Note that for mathematical simplicity in illustration, we will use 1-based indexing.

Binary heap data-structure

We use the left array to implicitly represent the right heap!

index	1	2	3	4	5	6	7	8
value	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8
		L_1	R_1	L_2	R_2	L_3	R_3	L_4
	$P_{2,3}$	$P_{4,5}$	$P_{6,7}$	P_8				



Array member relationship formulas

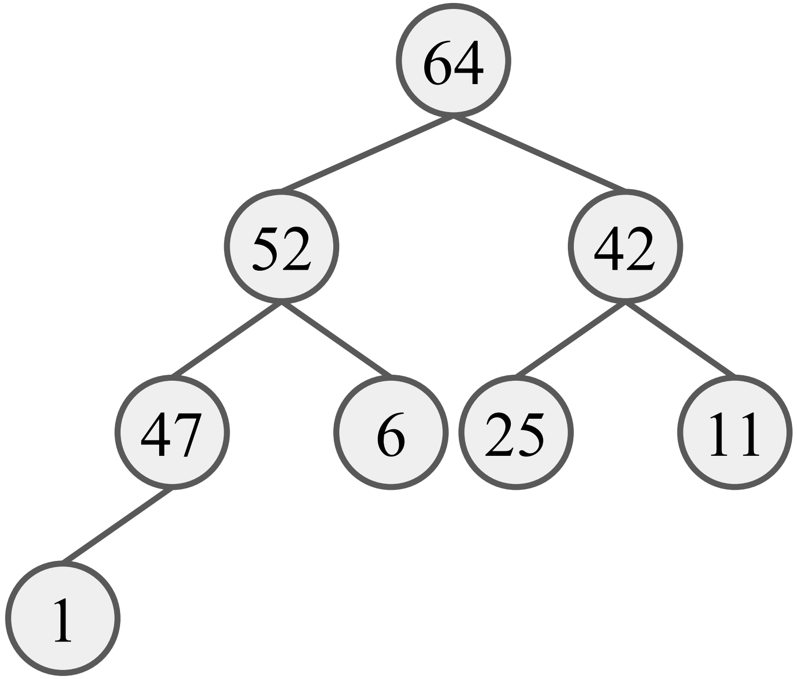
For array element at index i

1-based indexing	**0-based indexing**
L_i : index $i \times 2$	L_i : index $(i + 1) \times 2 - 1$
R_i : index $i \times 2 + 1$	R_i : index $(i + 1) \times 2$
P_i : index $\lfloor i \div 2 \rfloor$	P_i : index $\lfloor (i + 1) \div 2 \rfloor - 1$

We demonstrated with 1-based indexing so that the math becomes clearer. You should be able to derive these formulas by yourself easily!

Test yourself

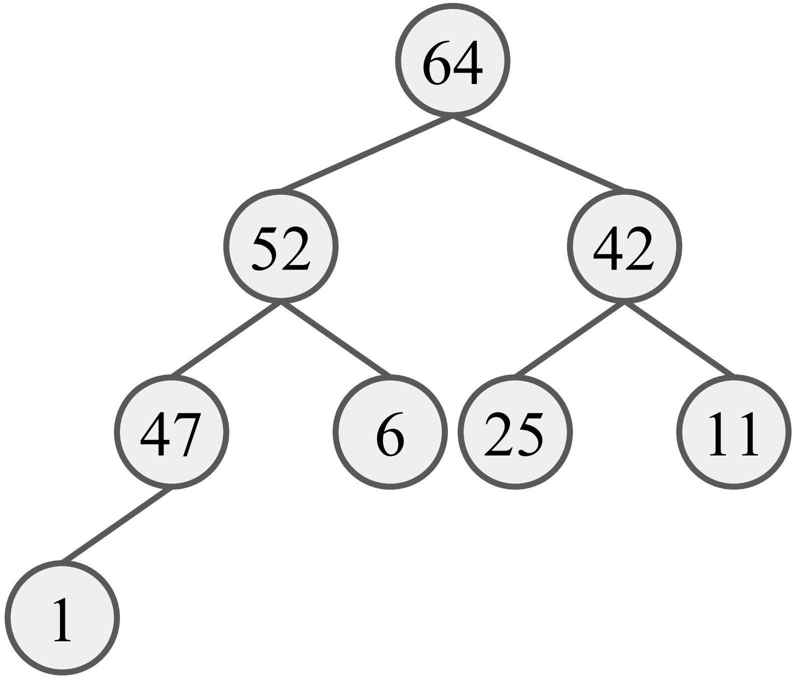
Is the value at the top of the max heap is greater than all other values in the heap?



Test yourself

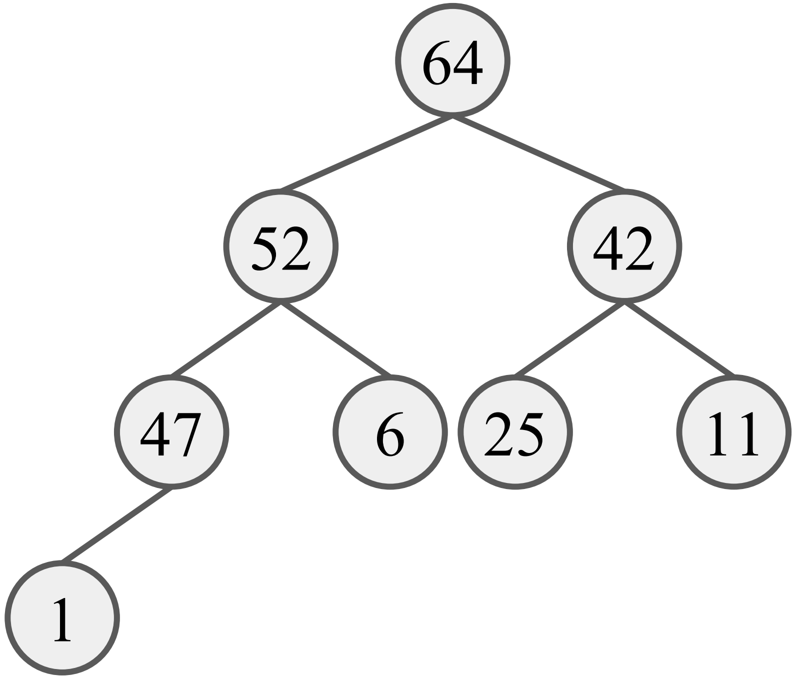
Is the value at the top of the max heap is greater than all other values in the heap?

Yes! By definition!



Test yourself

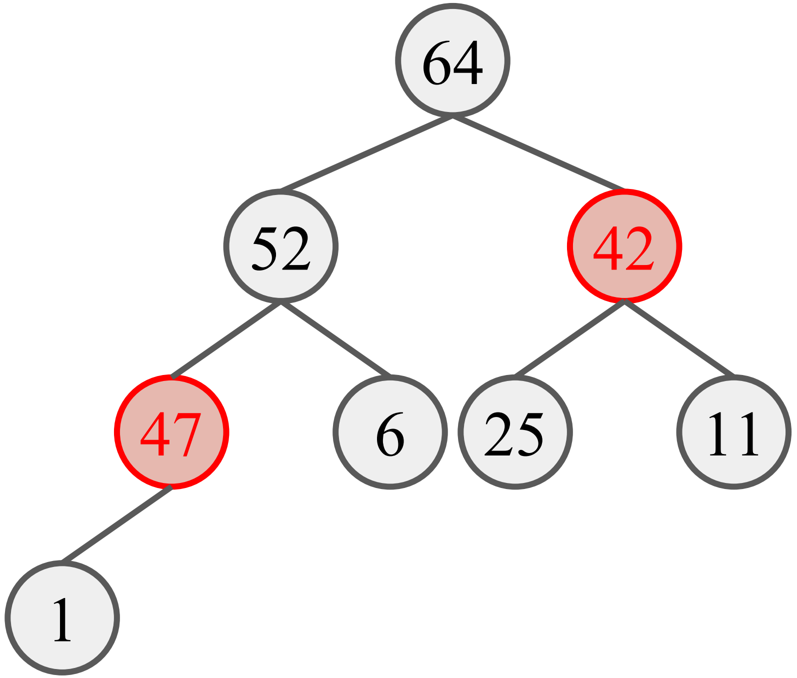
Is the value at a higher value greater than all values from lower levels?



Test yourself

Must the value at a higher level be greater than all values from lower levels?

Not necessarily!



`Insert(v)`

Steps:

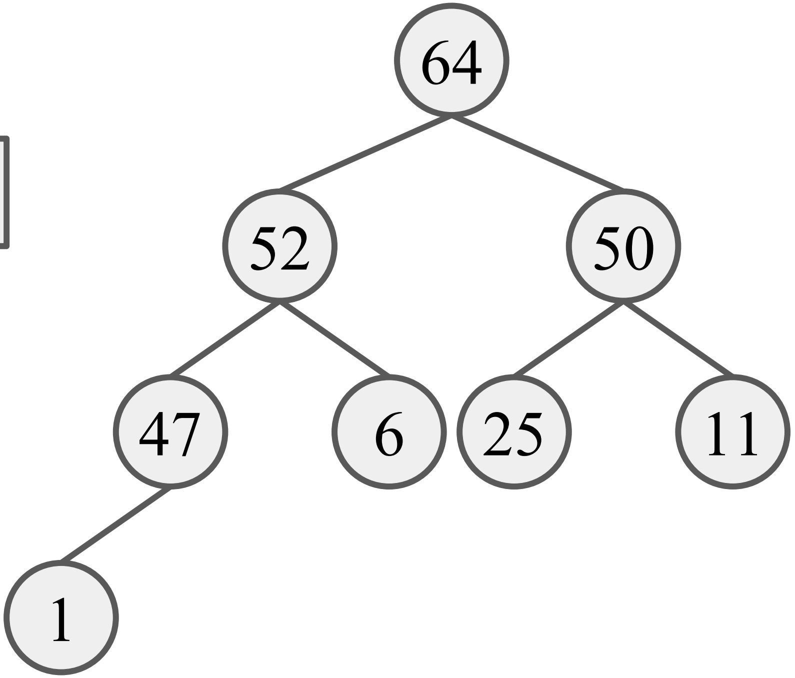
1. Attach a new vertex with value `v` and insert it into the leftmost position at the bottommost level. In the 1-based indexed array, this value will be inserted at index $N + 1$
2. *Bubble up* (A.K.A *shift up*, *increase key*) that value up until a valid spot in the heap is reached

Complexity: $O(\log N)$

`Insert(v)`

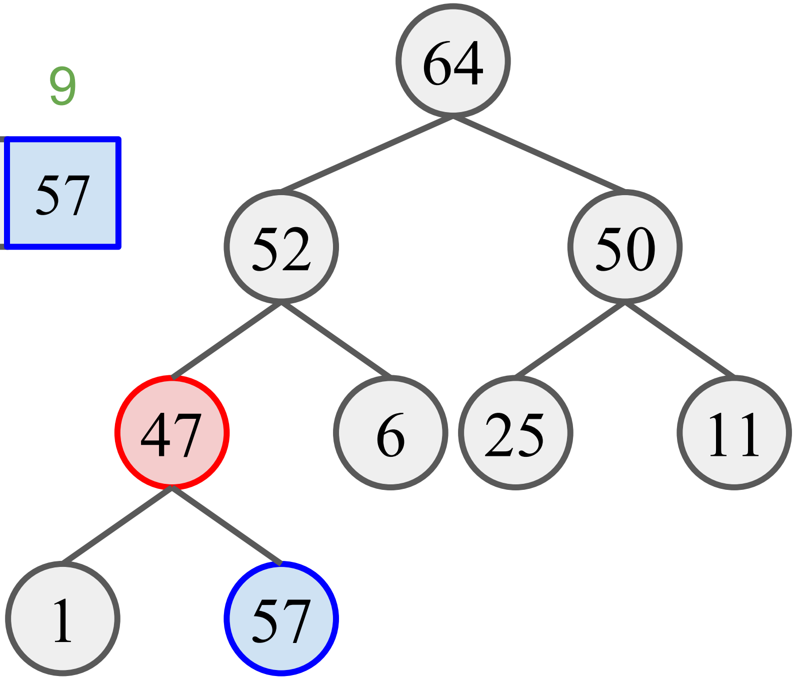
index	1	2	3	4	5	6	7	8
value	64	52	50	47	6	25	11	1

Say for instance, we insert the value 57 into this max-heap



`Insert(v)`

index	1	2	3	4	5	6	7	8	9
value	64	52	50	47	6	25	11	1	57

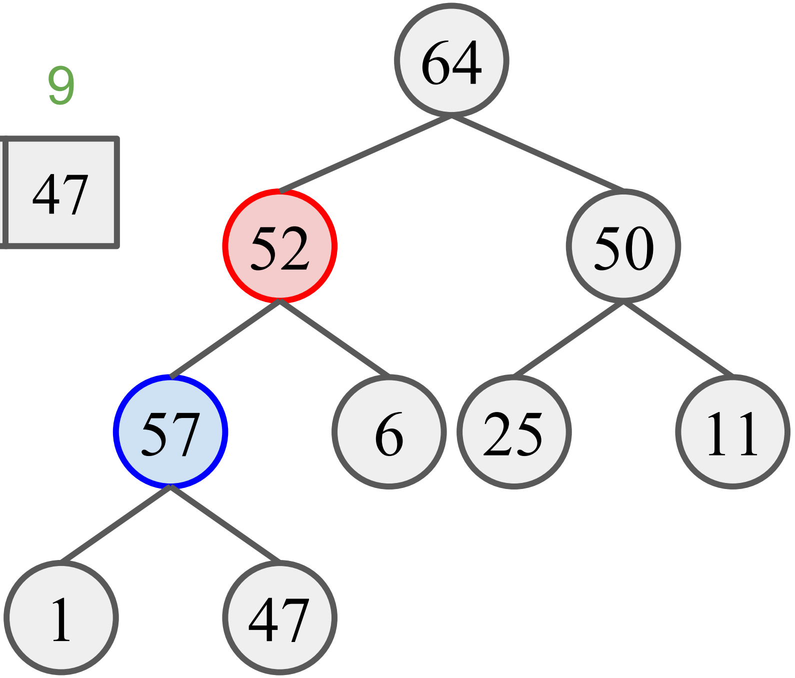


Insert 57 at last position in max-heap

57 is greater than its parent 47, so we will bubble up!

`Insert(v)`

index	1	2	3	4	5	6	7	8	9
value	64	52	50	57	6	25	11	1	47

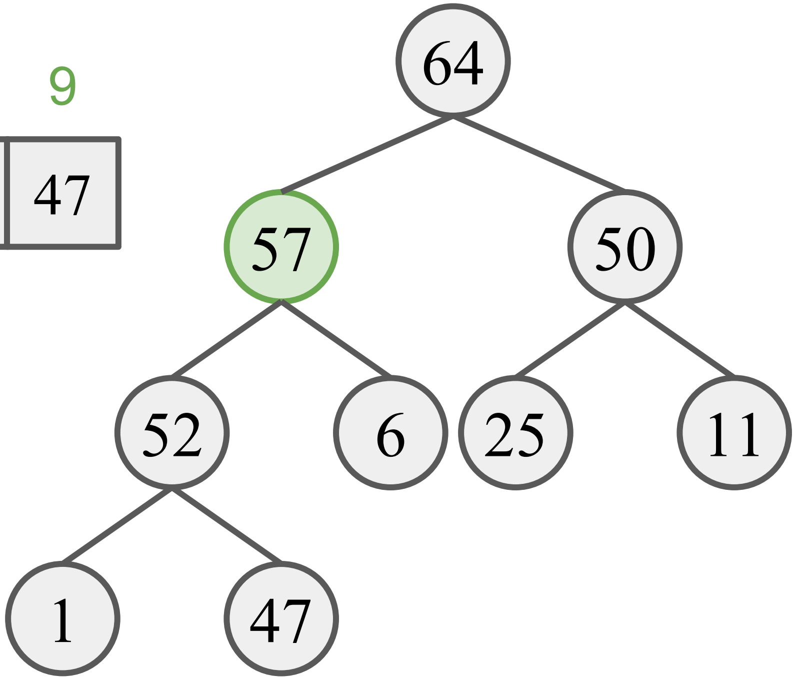


47 is now in its rightful spot.

57 is still greater than its new parent 52, so we will bubble up!

`Insert(v)`

index	1	2	3	4	5	6	7	8	9
value	64	57	50	52	6	25	11	1	47



52 is now in its rightful spot.

57 is no longer greater than its new parent 64, so we are done!

`ExtractMax()`

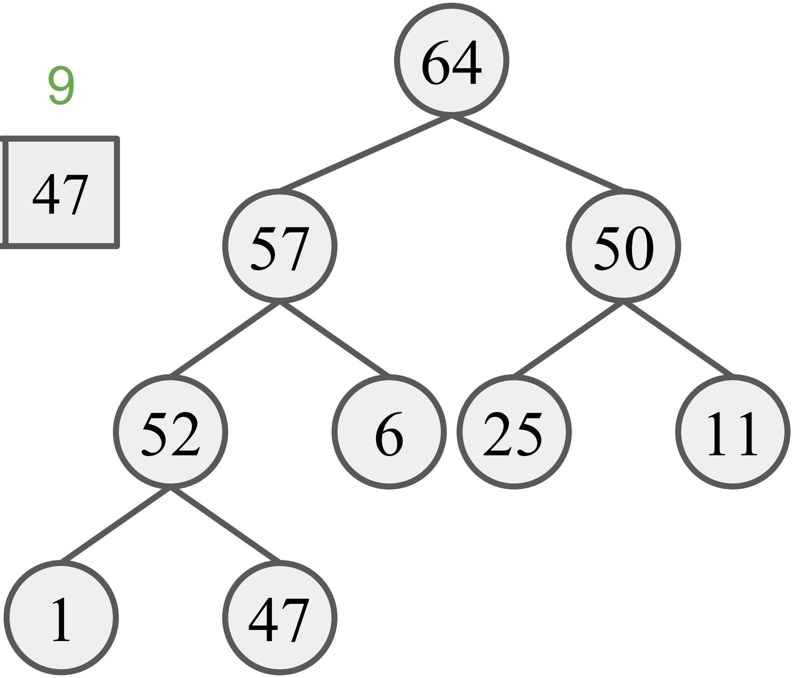
Steps

1. Remove topmost vertex in max-heap
2. Replace topmost vertex with bottommost leftmost vertex. In the 1-based indexed array, this value is at index N
3. *Bubble down* (A.K.A *shift down*, *heapify*) that value until a valid spot is reached

Complexity: $O(\log N)$

`ExtractMax()`

index	1	2	3	4	5	6	7	8	9
value	64	57	50	52	6	25	11	1	47



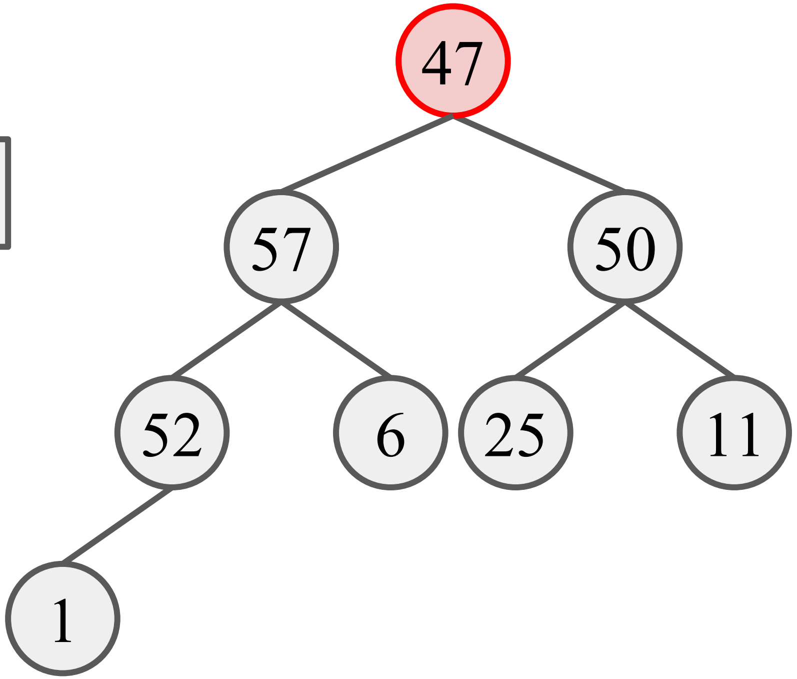
Say for instance, we call `ExtractMax()` on this max-heap.

We'll remove 64 and replace it with 47

`ExtractMax()`

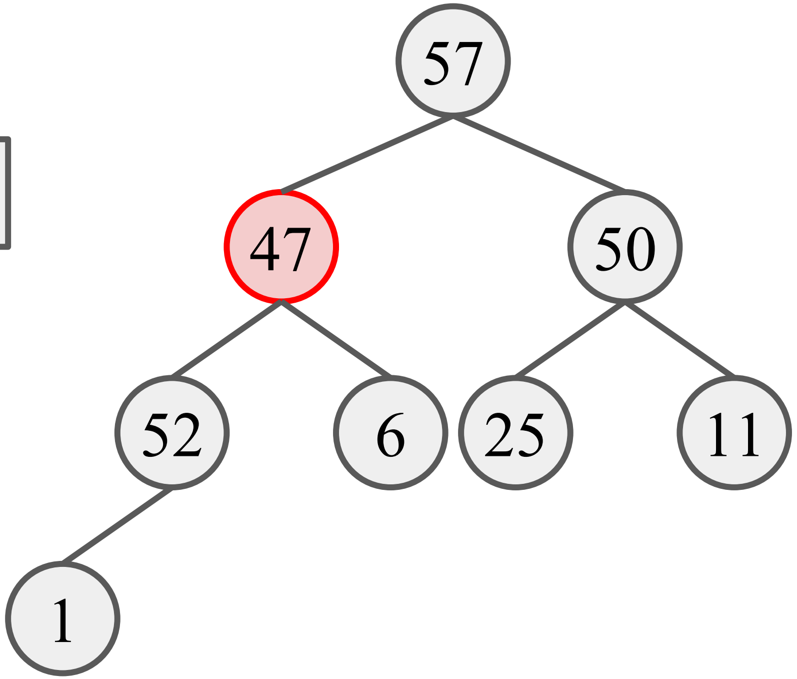
index	1	2	3	4	5	6	7	8
value	47	57	50	52	6	25	11	1

47 is lower than both its children, so we will bubble it down with 57, the greater of the two.



`ExtractMax()`

index	1	2	3	4	5	6	7	8
value	57	47	50	52	6	25	11	1

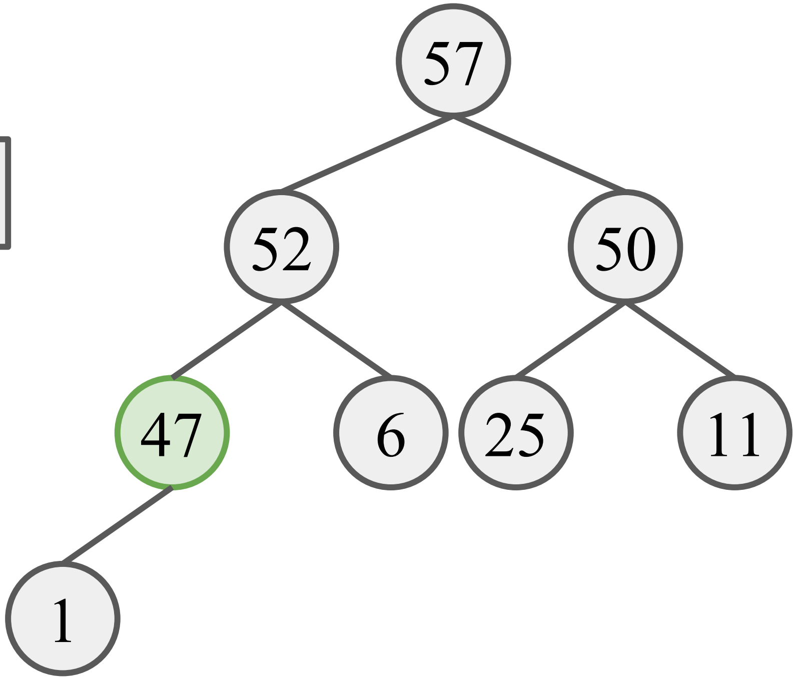


57 is now in its rightful spot.

47 is lower than both its new children, so we will bubble it down with 52, the greater of the two.

`ExtractMax()`

index	1	2	3	4	5	6	7	8
value	57	52	50	47	6	25	11	1



52 is now in its rightful spot.

47 is now greater than all its children, so we are done!

`Create(A)`

Two versions:

1. $O(N \log N)$ version
2. $O(N)$ version

`Create(A)` $O(N \log N)$ version

Approach: `Insert` each and every value from array `A` into the heap that is being built.

N values total and each value potentially bubbled-up $\log N$ levels so total time complexity is $O(N \log N)$.

A more mathematical analysis:

$\log 1 + \log 2 + \log 3 + \ldots + \log N \leqslant \log N + \log N + \log N + \ldots + \log N$

$\log 1 + \log 2 + \log 3 + \ldots + \log N \leqslant N \log N$

$O(\log 1 + \log 2 + \log 3 + \ldots + \log N) = O(N \log N)$

`Create(A)` $O(N)$ version

Invented by Robert W. Floyd in 1964.

Approach: Take in entire raw array as heap and build it level-by-level from the "bottom-up".
https://visualgo.net/en/heap?slide=7-2

Pseudocode:

```
For vertex $v_i$ from $v_N$ down to $v_1$:
    heapify($v_i$)
```

`Create(A)` $O(N)$ version

But why is it $O(N)$? A loose analysis using same reasoning as insertion method seem to also suggest complexity $O(N \log N)$!

It turns out that the tight bound is not $O(N \log N)$.

The key difference here is that bubble-up and bubble-down (heapify) operations have different time complexities:

- Bubble-up: $O(\log N)$ comparisons at the worst when all levels traversed
- Bubble-down: $O(h)$ comparisons at the worst where h is the height of the subtree rooted at the vertex we wish to bubble-down

`Create(A)` $O(N)$ version

Steps for a heap with N vertices:

- It has at most $N/2$ leaves. Since all leaves are by themselves valid subheaps, we can ignore them and start heaifying on the next level onwards
- Next level has $N/4$ vertices with each subheap rooted at them having a height of 2
- Next level has $N/8$ vertices with each subheap rooted at them having a height of 4
- And so on… until we reach the root node (N/N vertices) with a height of $log\ N$

`Create(A)` *O(N)* version

Total comparisons

$$\underbrace{\sum_{h=0}^{\lfloor \lg(n) \rfloor}}_{\text{Sum over all levels}} \underbrace{\overbrace{\left\lceil \frac{n}{2^{h+1}} \right\rceil}^{\text{\# of nodes at height h}} \overbrace{O(h)}^{\text{Cost to Heapify a node at height h}}}_{\text{Cost for a level}} = \sum_{h=0}^{\lfloor \lg(n) \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil c*h = O\left(n \sum_{h=0}^{\lfloor \lg(n) \rfloor} \frac{h}{2^h}\right) = O\left(n \underbrace{\sum_{h=0}^{\infty} \frac{h}{2^h}}_{\sum_{k=0}^{\infty} kx^k = \frac{x}{(1-x)^2} = 2}\right) = O(2\mathbf{N}) = O(\mathbf{N})$$

$$\boxed{x = 1/2}$$

Question 3

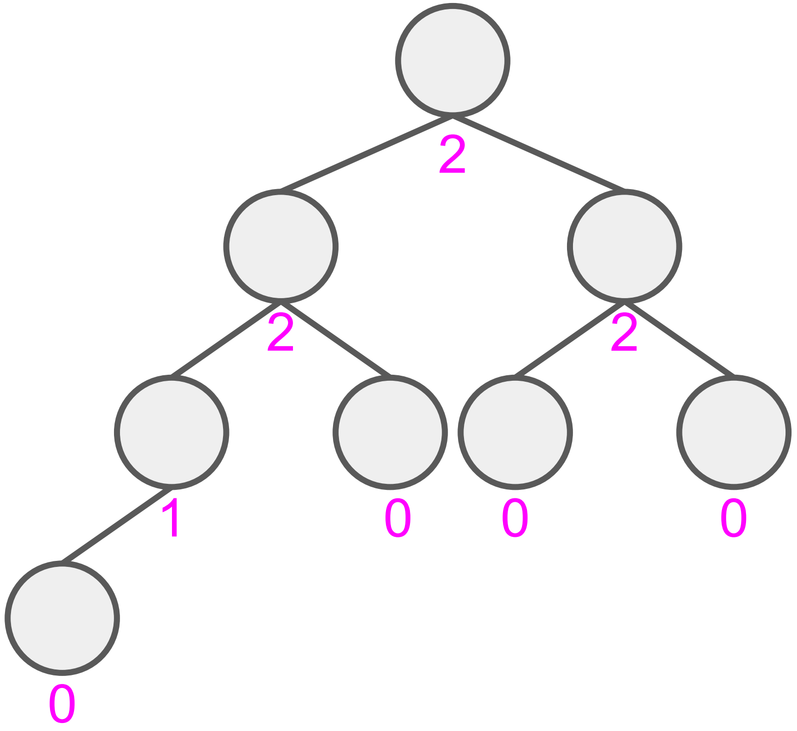
What is the **minimum** and **maximum** number of comparisons between Binary Heap elements required to construct a Binary (Max) Heap of arbitrary n elements using the $O(N)$ `Create(A)`?

Realize that for `heapify`, each vertex must have make the number of comparisons equal to its number of children. For a vertex with 2 children, it must first check which is the greater of the 2 children, then it must check if its value is lower than the greater of its 2 children.

Question 3

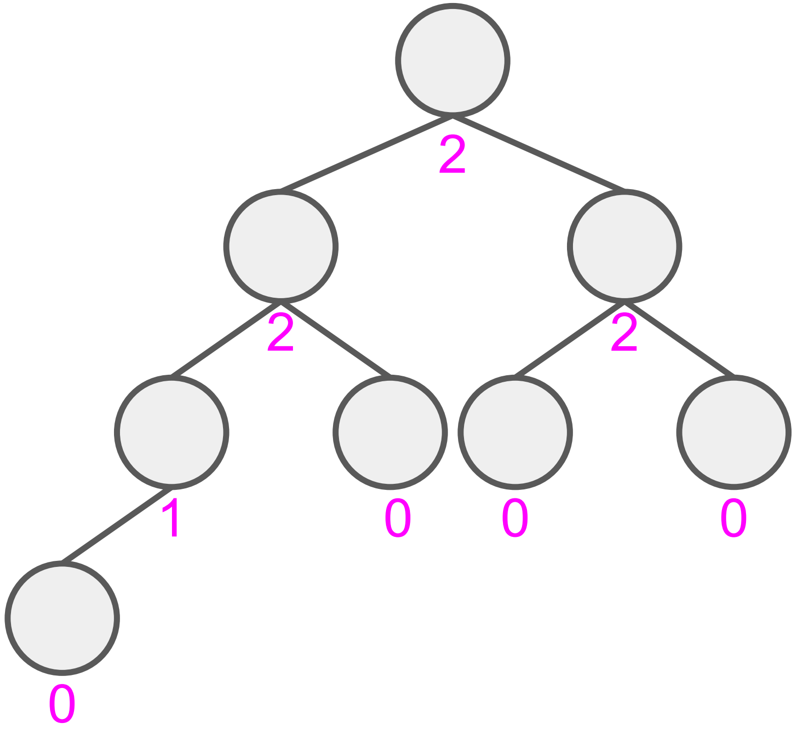
Take for instance a heap with 8 vertices.

Number of comparisons needed at each vertex displayed below vertex in pink



Question 3

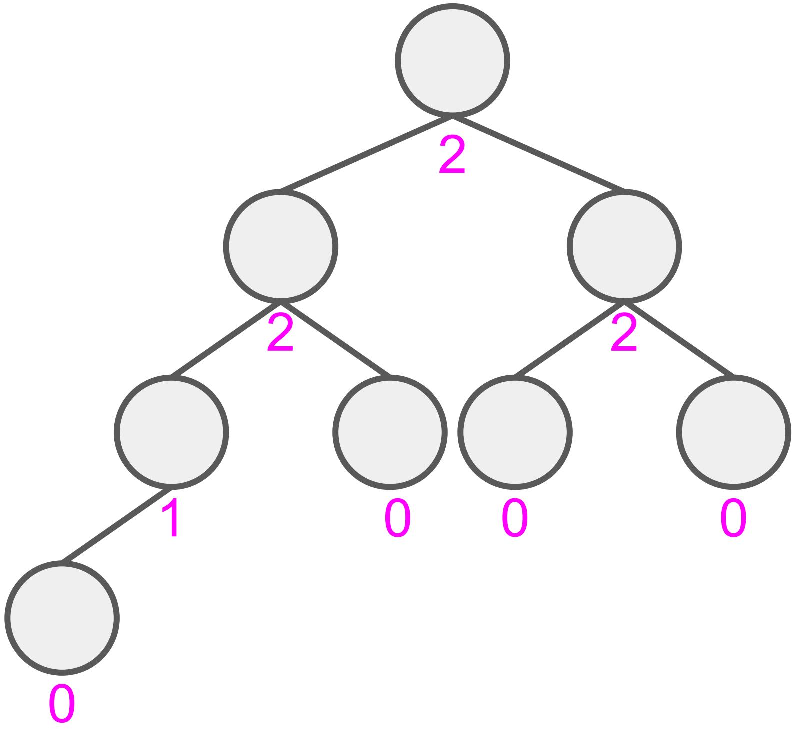
We incur minimum number of total comparisons if the array is already in heap order. So each vertex just need to conduct its own comparisons once and determine that no further heapify is required.



So minimum comparisons is $1 + 2 + 2 + 2 = 7$

Question 3

We incur maximum number of total comparisons if each vertex need to heapify down its entire height to the leaves, incurring comparisons at each vertex along the way



So max comparisons is $1 + (2 + 1) + 2 + (2 + (2 + 1)) = 11$

`HeapSort(A)`

Steps to sort an array of values in descending order

1. Create max-heap using $O(N)$ `Create(A)`
2. Repeatedly call `ExtractMax()` until max-heap is empty (N times). $O(N \log N)$ time
3. The sequence of extracted values will be in descending order

Time and space complexity: $O(N \log N)$

Note: For sorting ascending order, repeat the same steps as above but use a min-heap

PS2 tips

PS2 A

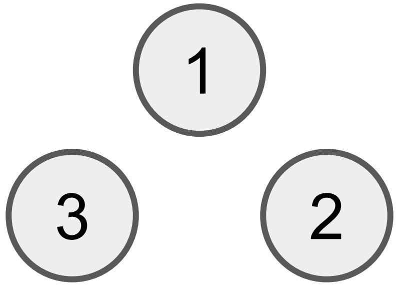
- Just simulate.
- “Innocent until proven guilty”
- Maintain a “can be X” boolean flag. If, at any time, expected value `!=` actual value, flag as not possible.
- Beware of Runtime Error, likely caused by popping an empty queue/stack/PQ.

PS2 B

- Just simulate
- The problem is probably harder to understand that the implementation itself
- Figure out who is last person standing, given how many syllables the phrase contains, and how many players
- In the next few slides we shall assume 10 syllabus and illustrate the game for a couple of rounds

PS2 B

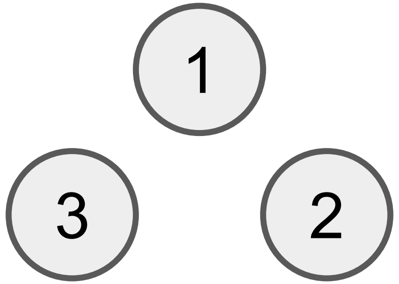
Co-co-nut, Co-co-nut, Co-co-nut, Splat!



PS2 B

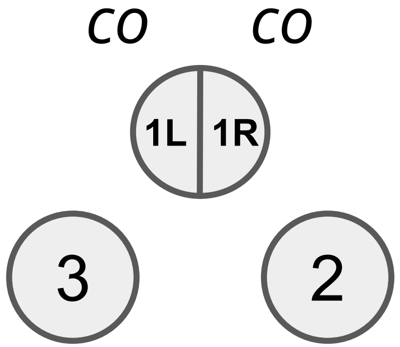
Co-co-nut, Co-co-nut, Co-co-nut, Splat!

splat



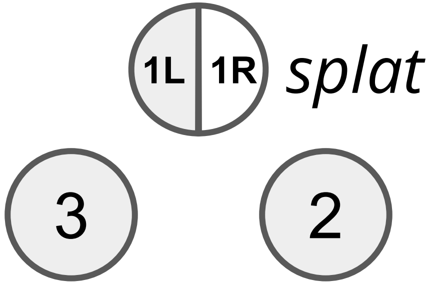
PS2 B

Co-co-nut, Co-co-nut, Co-co-nut, Splat!



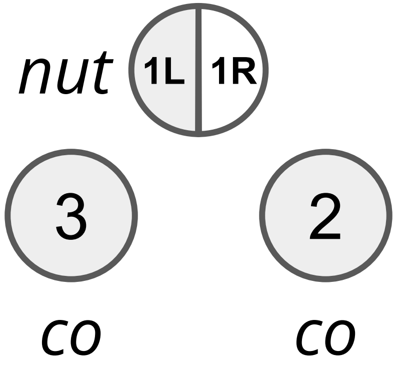
PS2 B

Co-co-nut, Co-co-nut, Co-co-nut, Splat!



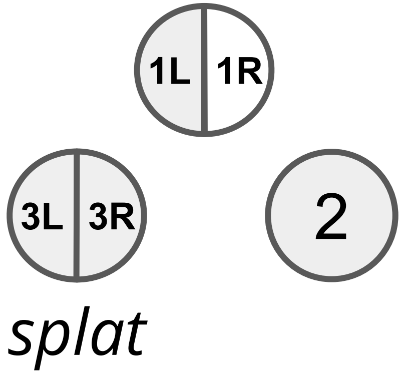
PS2 B

Co-co-nut, Co-co-nut, Co-co-nut, Splat!



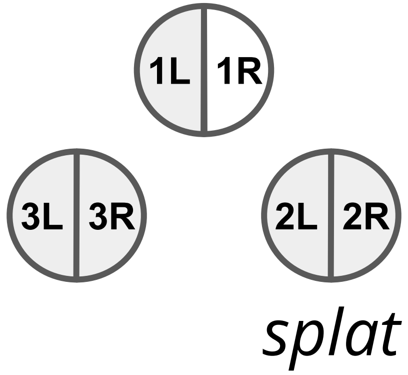
PS2 B

Co-co-nut, Co-co-nut, Co-co-nut, Splat!



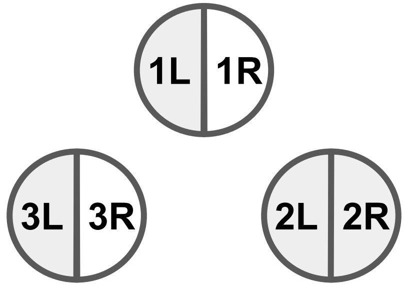
PS2 B

Co-co-nut, Co-co-nut, Co-co-nut, Splat!



PS2 B

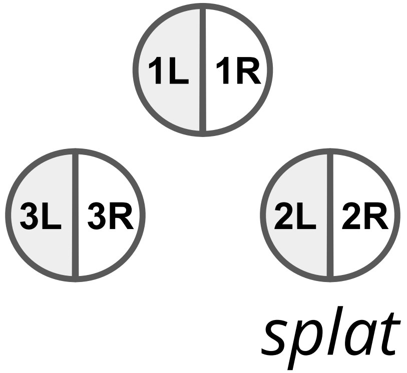
Co-co-nut, Co-co-nut, Co-co-nut, Splat!



splat

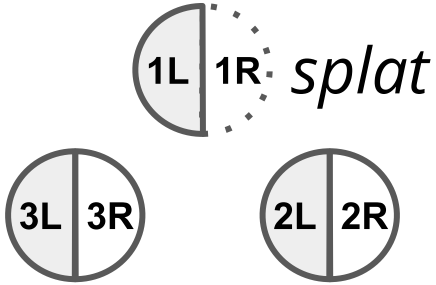
PS2 B

Co-co-nut, Co-co-nut, Co-co-nut, Splat!



PS2 B

Co-co-nut, Co-co-nut, Co-co-nut, Splat!



And so on...

PS2 C

- Use a suitable data structure that allows you to place the first car that arrives onto the ferry
- Remember that
  - Ferry has capacity limitations, some cars that arrive need to wait till next ferry
  - Ferry takes time to shuttle to opposite end if it was waiting at one side but car arrives at the other

PS2 C

- Cannot simulate minute-by-minute!
- Largest case: ship holds 1 car at a time, and takes 10000 minutes to ferry, total 10000 cars
  - 100 000 000 = likely TLE
- Can “fast-forward” till next event

PS2 C

Input

```
1
2 10 2
0 right
0 left
```

Output

```
20
10
```

Ferry will pick up all cars if cars arrive `<=` ferry departure time until it has maxed out its capacity