University of Nottingham
UK | CHINA | MALAYSIA

COMP2054 Tutorial Session 6: Hash Maps, Heaps, and BSTs

Rebecca Tickle
Warren Jackson
AbdulHakim Ibrahim

Session outcomes

- Understand difference between heaps and BSTs
- Know how to apply the various operations on heaps and BSTs
- Understand how hash maps work and deal with collisions

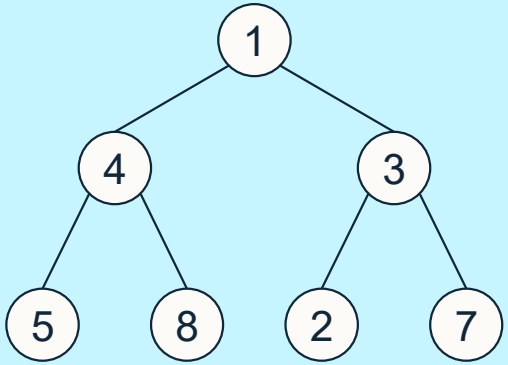
University of Nottingham
UK | CHINA | MALAYSIA

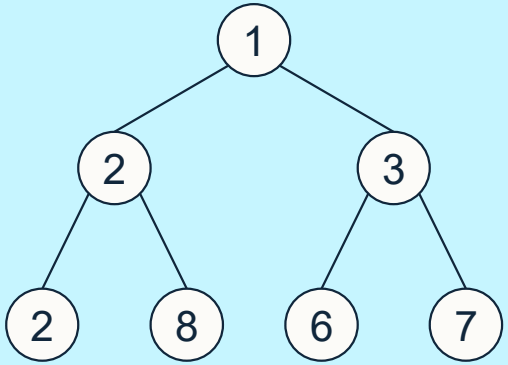
Trees

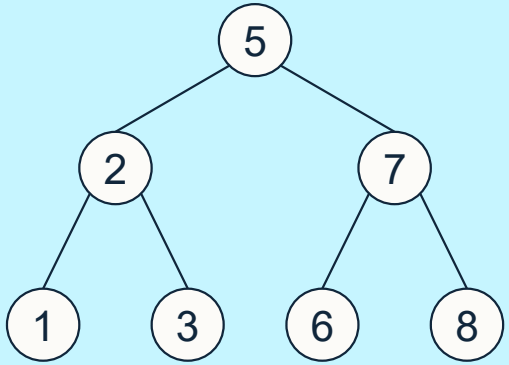
Heaps and BSTs

Quiz – Q1

Classify each of the following trees as a Heap, a BST, or neither:



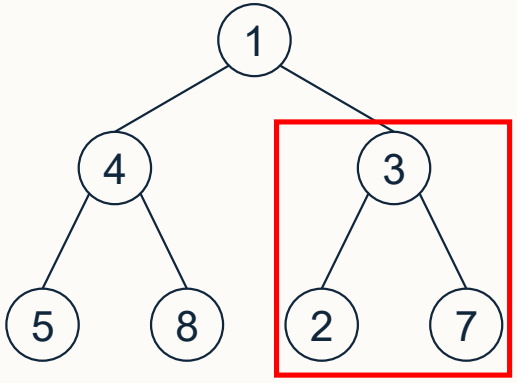




Quiz – Q1

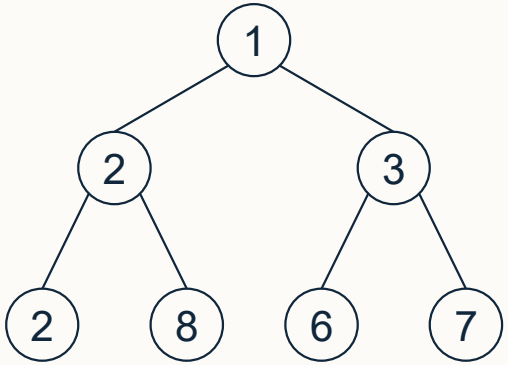
Classify each of the following trees as a Heap, a BST, or neither:

Neither



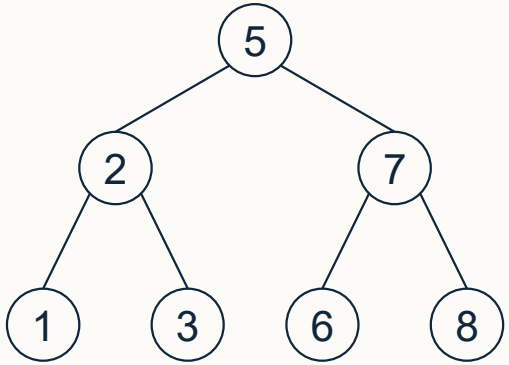
While at first this appears to be a heap with the smallest element (1) at the root, the highlighted subtree does not have the minimum element (2) at its root which is (3).

Heap



This is a valid heap. The tree has the minimum element (1) at its root, and all sub-trees contain the smallest element at their roots.

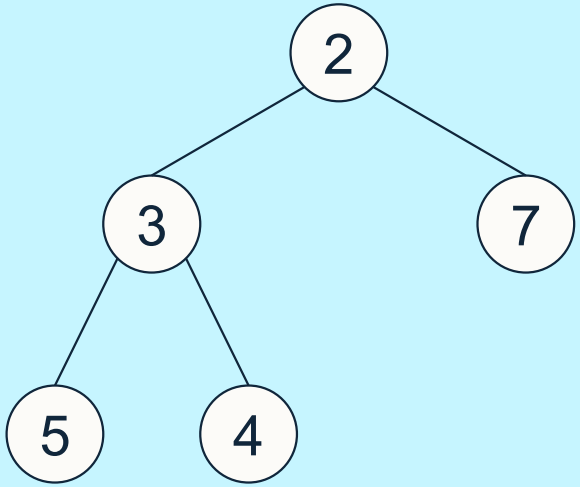
BST



This is a valid BST since for all nodes, their left subtree contains only values less than the given node, and the right subtree only values greater than the given node. E.g. max[-,2,1,3] < 5 < min[-,7,6,8]

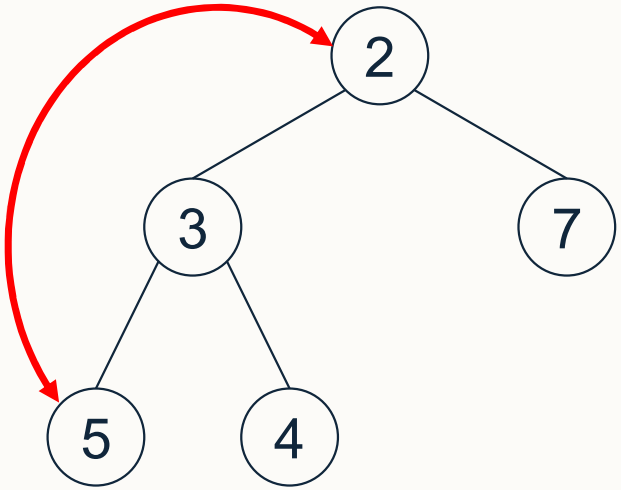
Quiz – Q2

Which two nodes in the following heap should be swapped to create a BST?



Quiz – Q2

Which two nodes in the following heap should be swapped to create a BST?

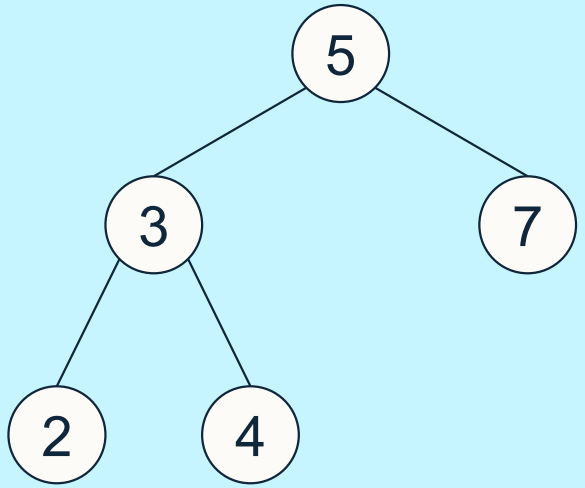


Swapping the 2 and the 5 will result in the binary tree [-,5,3,7,2,4] which is a BST.

Array Representation of Binary Trees

How can we represent the below BST in an array?

Index	[0]	[1]	[2]	[3]	[4]	[5]
Value						



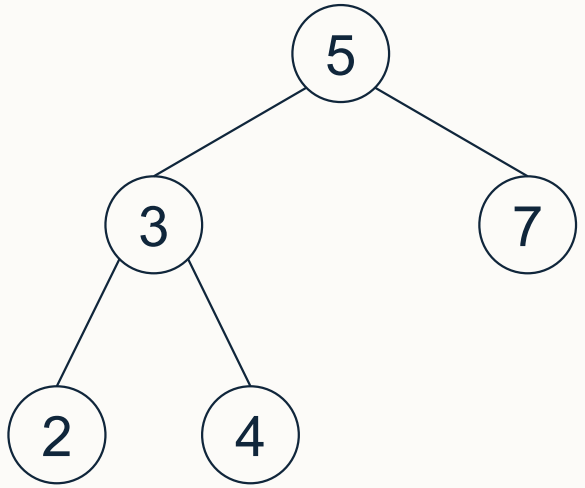
Remember:

```
index(root) = 1
index(left_child, i) = 2i
index(right_child, i) = 2i + 1
```

Array Representation of Binary Trees

How can we represent the below BST in an array?

Index	[0]	[1]	[2]	[3]	[4]	[5]
Value		5	3	7	2	4



Remember:

```
index(root) = 1
index(left_child, i) = 2i
index(right_child, i) = 2i + 1
```

Quiz – Q3

Given the two binary trees in an array-based format, identify which is a heap and which is a BST, and explain why.

Binary tree 1:

Index	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
Value		2	3	5	4	6		

This cannot be a BST since the root contains the smallest element and has a left child (3) which is not less than or equal to itself. Drawing out the tree, we can see that this **is** a valid heap.

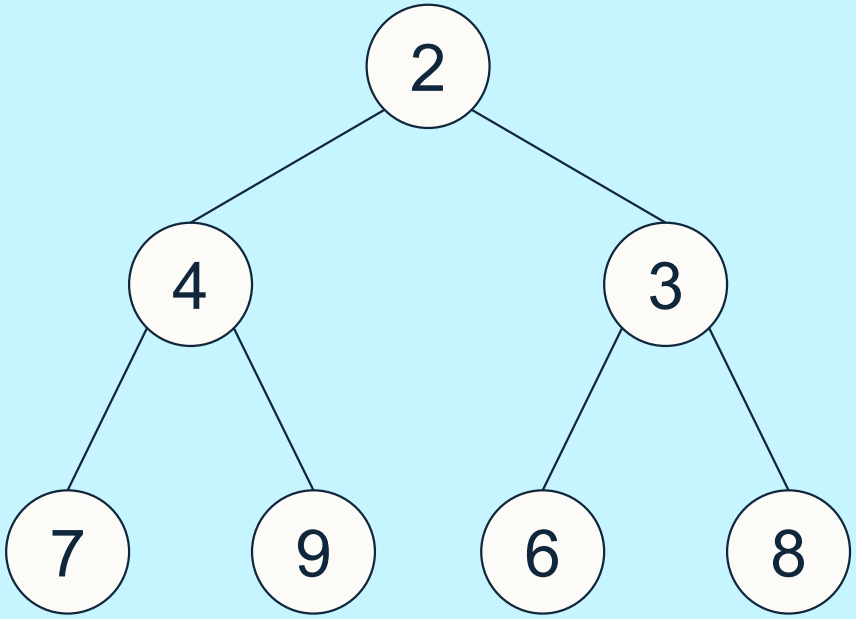
Binary tree 2:

Index	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
Value		4	3	5	2			6

This cannot be a heap since the root does not contain the smallest element. Furthermore, there are “gaps” meaning the shape property of the heap is violated. Drawing out the tree, we can see this **is** a valid BST.

Operations on Heaps – insertItem(x)

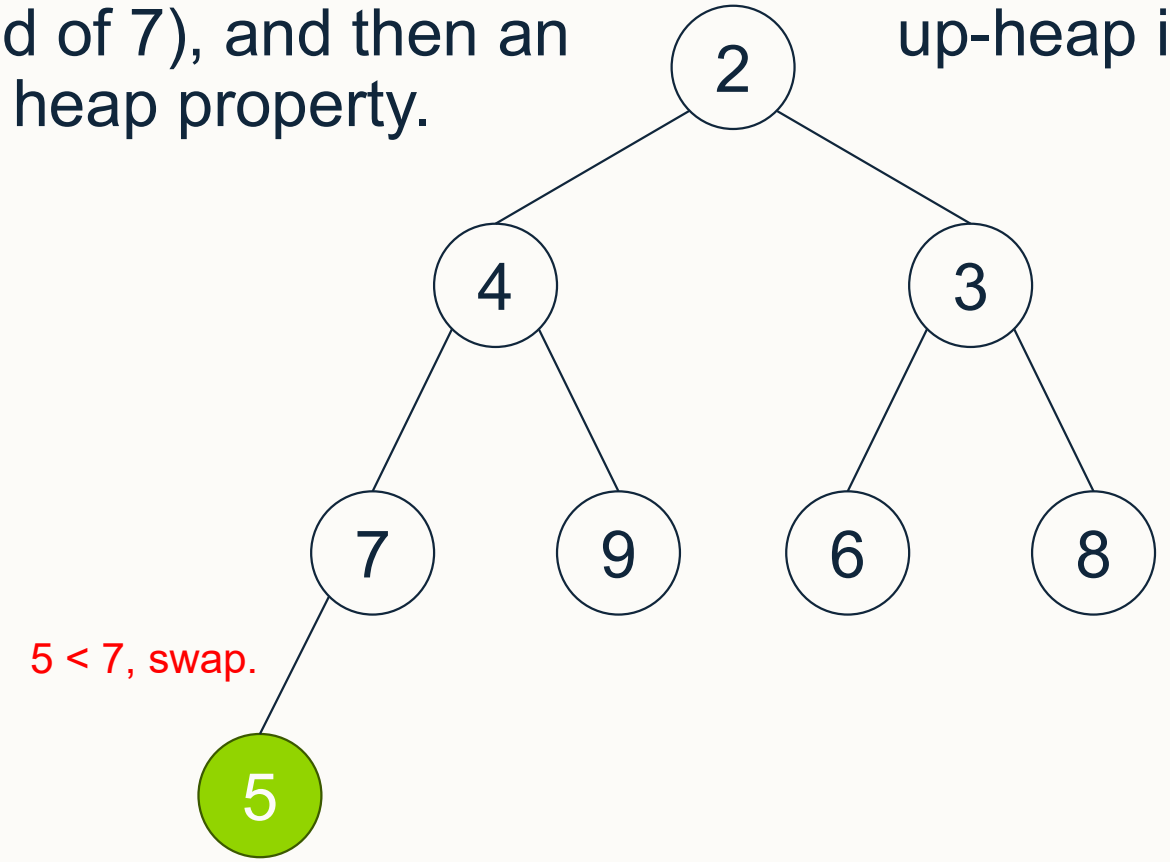
- Starting with the heap:



Perform insertItem(5), insertItem(1), then insertItem(10).

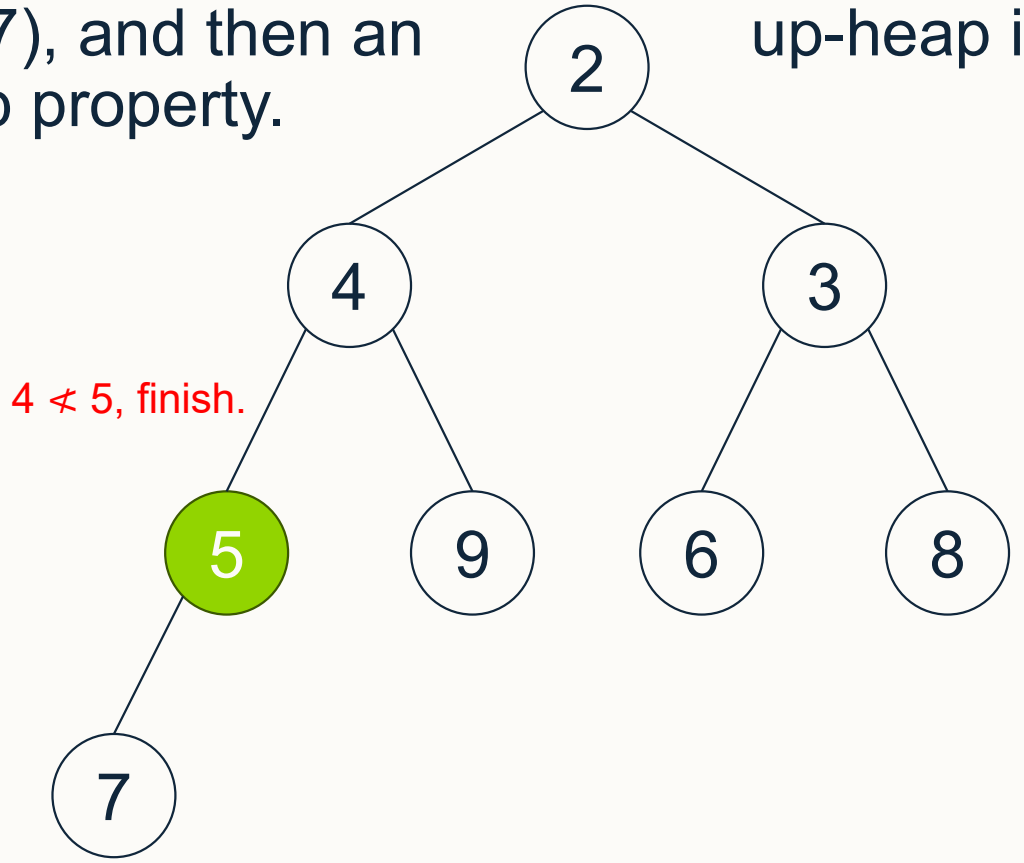
Operations on Heaps – insertItem(x)

- InsertItem(5): The 5 is placed at the left-most available space (left child of 7), and then an up-heap is performed to preserve the heap property.



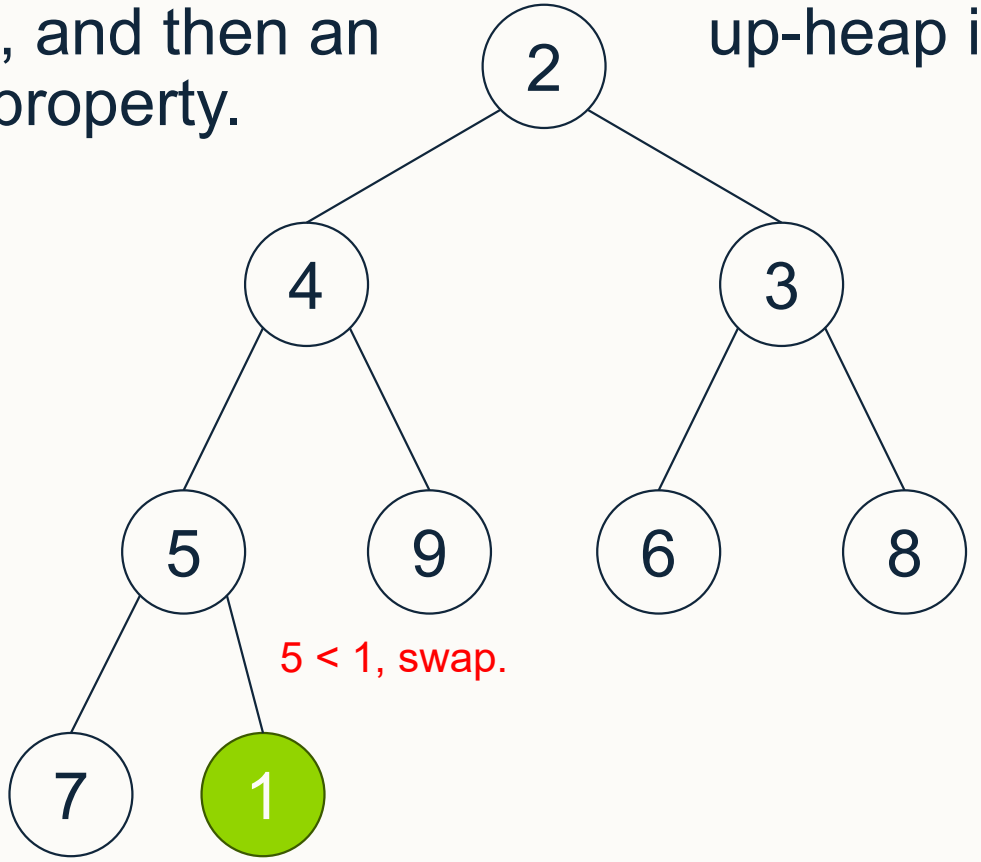
Operations on Heaps – insertItem(x)

- InsertItem(5): The 5 is placed at the left-most available space (left child of 7), and then an up-heap is performed to preserve the heap property.



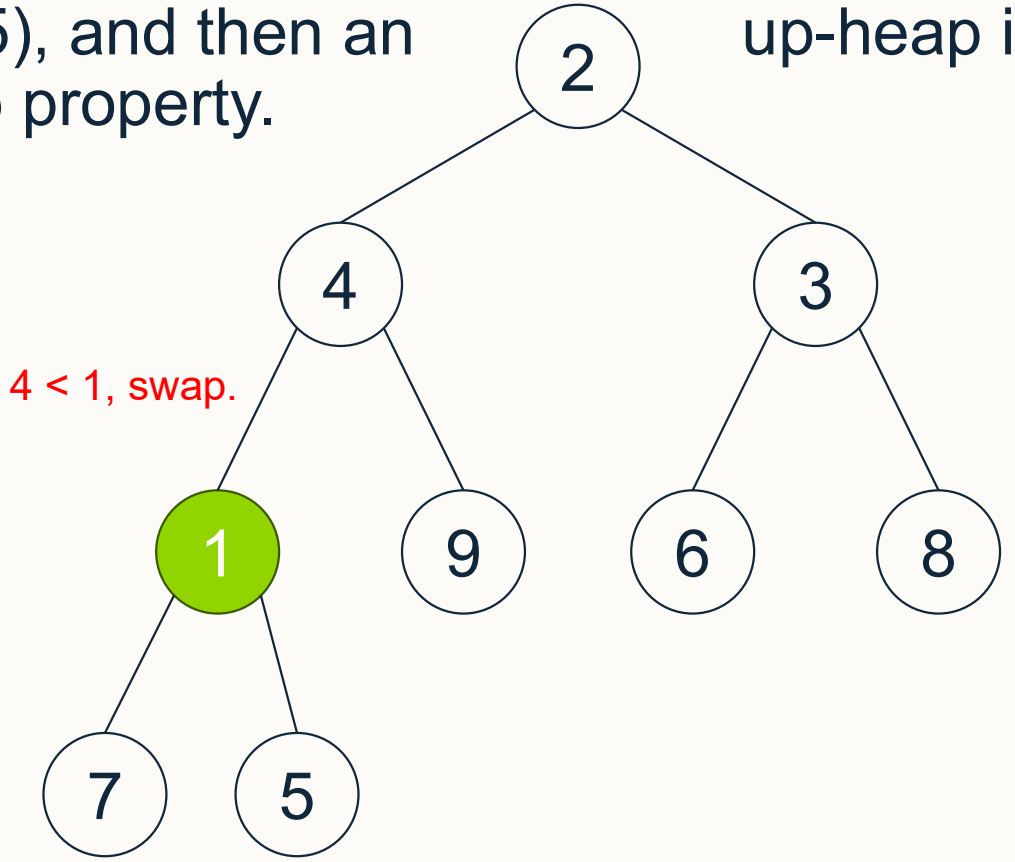
Operations on Heaps – insertItem(x)

- InsertItem(1): The 1 is placed at the left-most available space (right child of 5), and then an up-heap is performed to preserve the heap property.



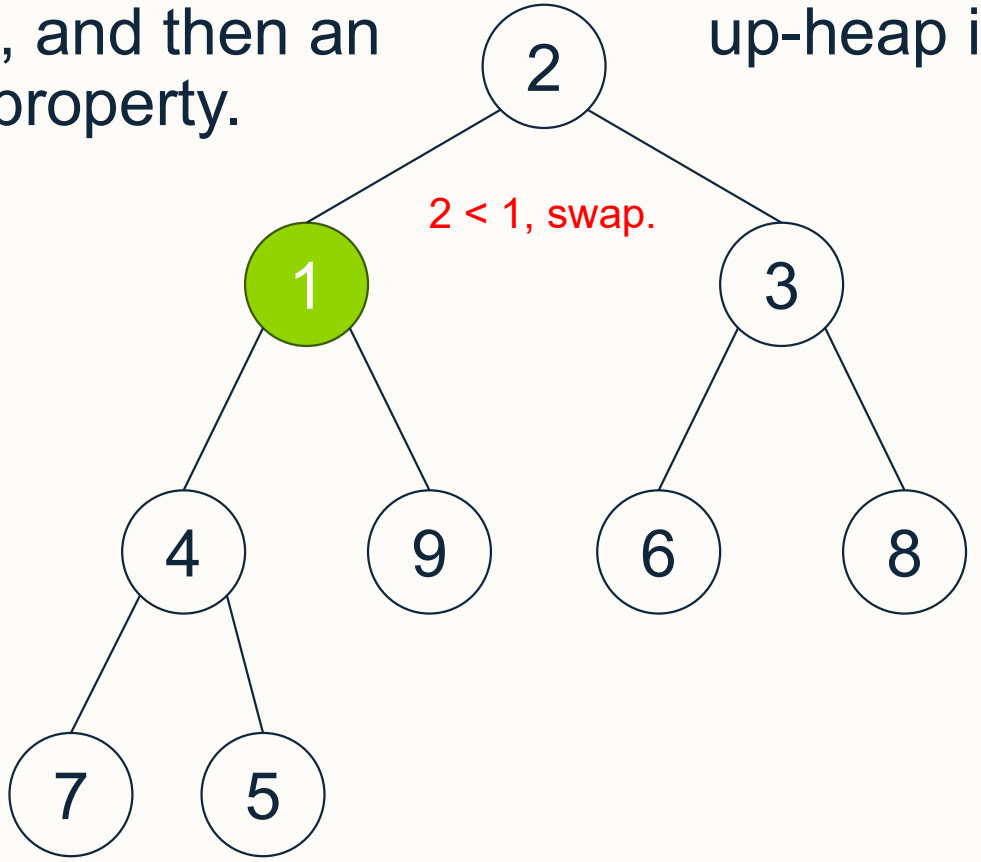
Operations on Heaps – insertItem(x)

- InsertItem(1): The 1 is placed at the left-most available space (right child of 5), and then an up-heap is performed to preserve the heap property.



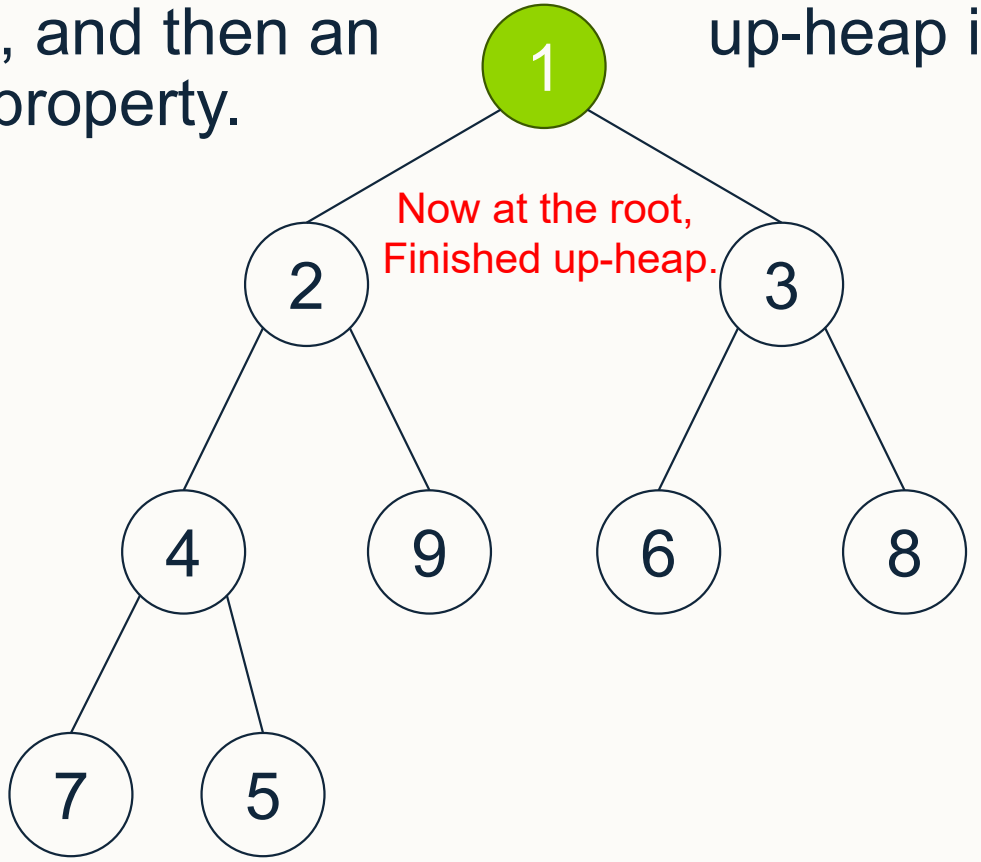
Operations on Heaps – insertItem(x)

- InsertItem(1): The 1 is placed at the left-most available space (right child of 5), and then an up-heap is performed to preserve the heap property.



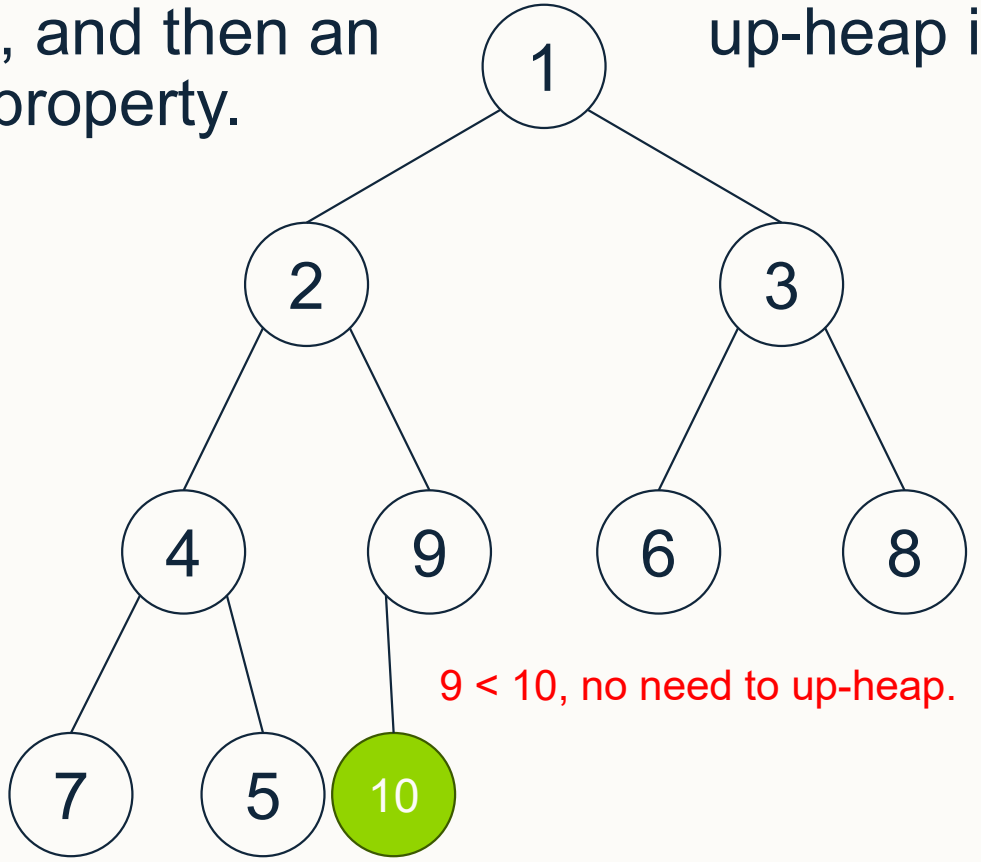
Operations on Heaps – insertItem(x)

- InsertItem(1): The 1 is placed at the left-most available space (right child of 5), and then an up-heap is performed to preserve the heap property.



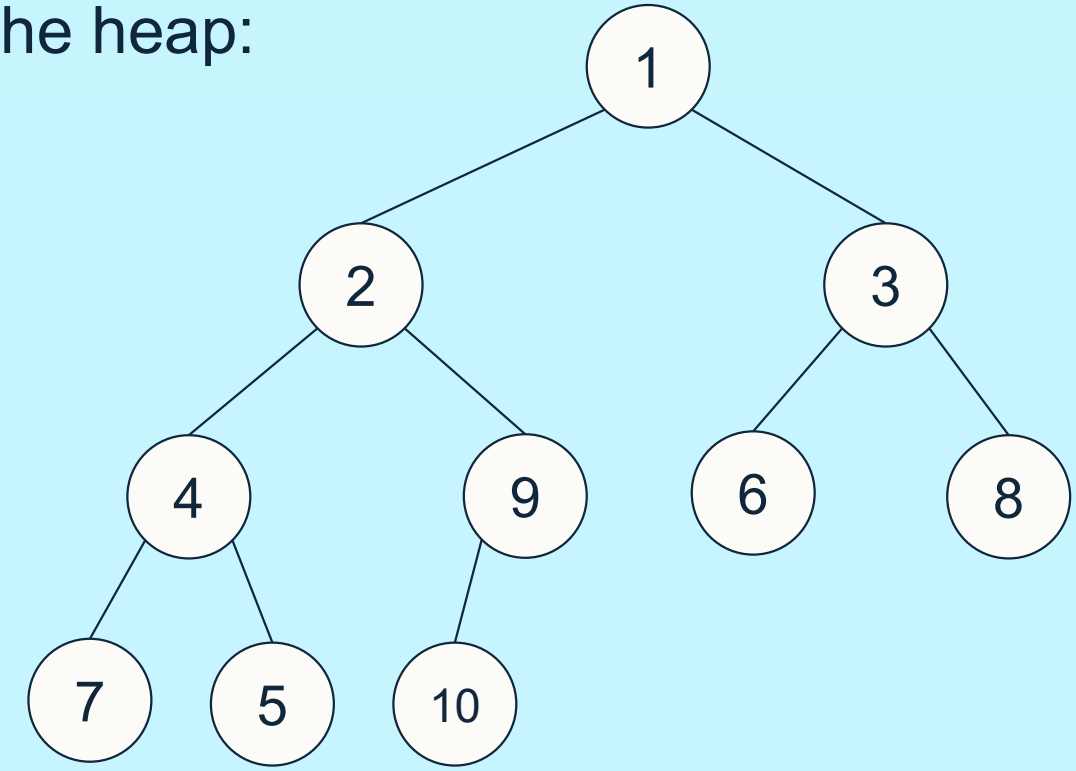
Operations on Heaps – insertItem(x)

- InsertItem(10): The 10 is placed at the left-most available space (left child of 9), and then an up-heap is performed to preserve the heap property.



Operations on Heaps – removeMin()

- Starting with the heap:



Perform removeMin() swapping with smallest child (if less than current).

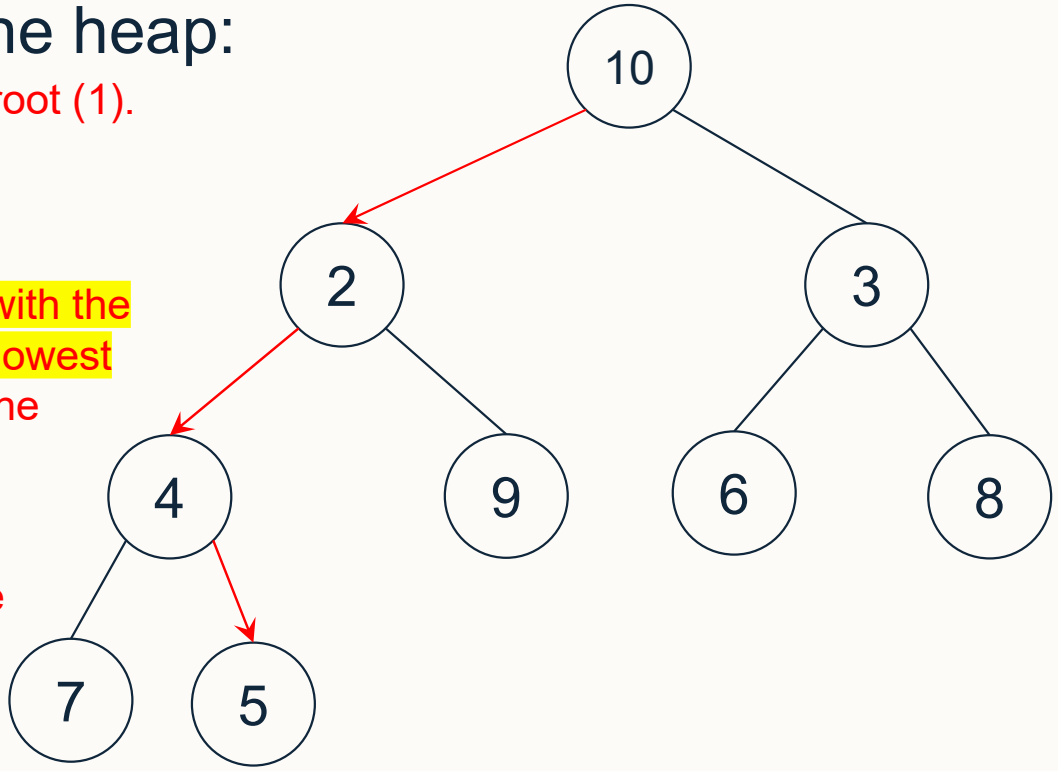
Operations on Heaps – removeMin()

- Starting with the heap:

Start by removing the root (1).

Then replace the root with the right-most node in the lowest level (10) to preserve the shape property.

Now need to perform down-heap to preserve the numeric property, swapping with the smallest child.



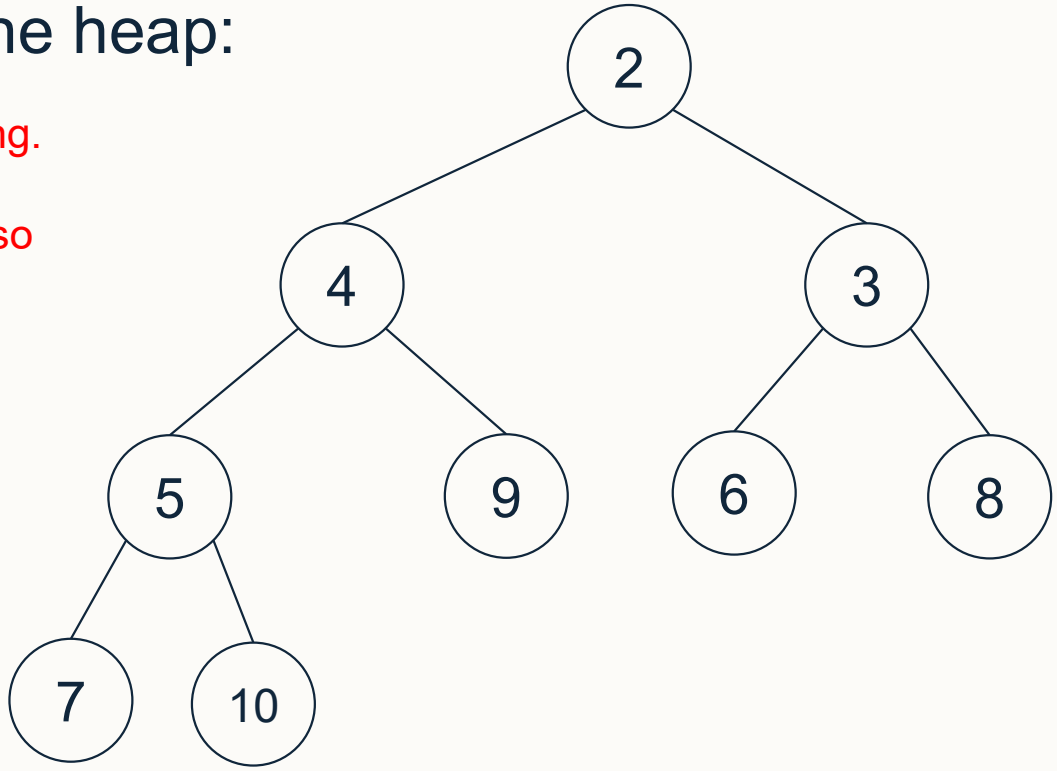
Perform removeMin() swapping with smallest child (if less than current).

Operations on Heaps – removeMin()

- Starting with the heap:

Which gives the following.

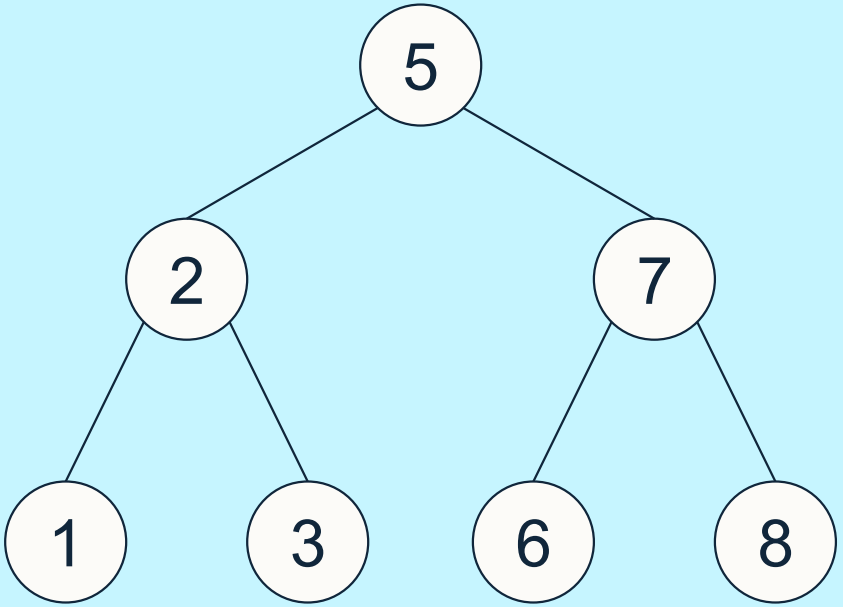
Note how this is now also a valid heap.



Perform removeMin() swapping with smallest child (if less than current).

Operations on BSTs – insert(x)

- Starting with the BST:



Perform insert(4).

Operations on BSTs – insert(x)

- Starting with the BST:

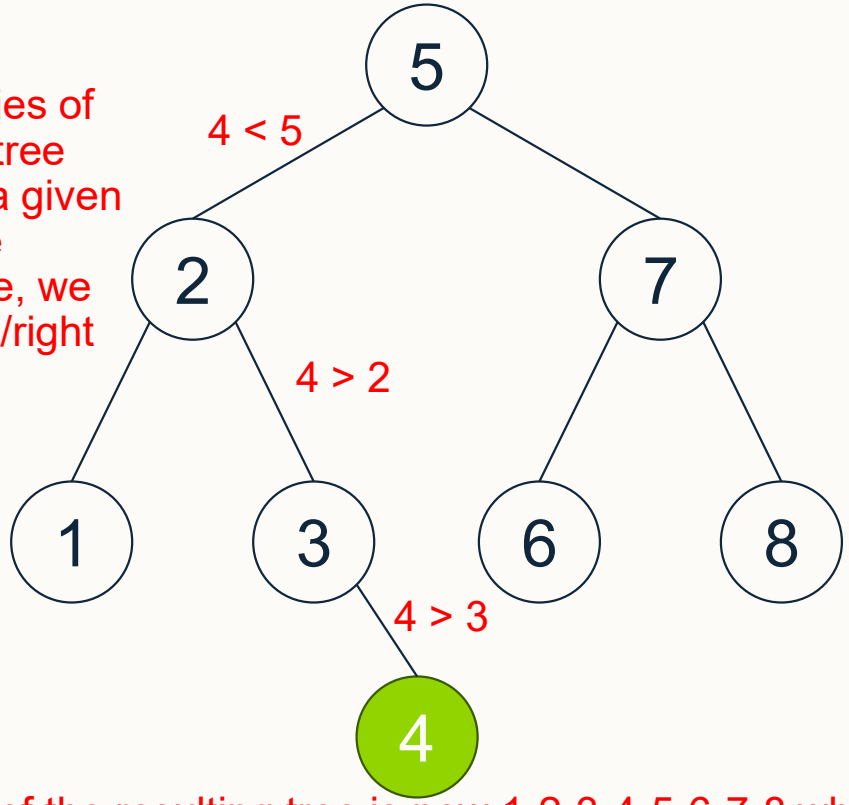
Inserting into a BST is not the same as inserting into a heap.

Since we know that the properties of a BST ensures that the left subtree contains only values less than a given root node, and the right subtree greater than the given root node, we traverse from the root going left/right depending on the value being inserted.

4 < 5 so need to go left.

4 > 2 so need to go right.

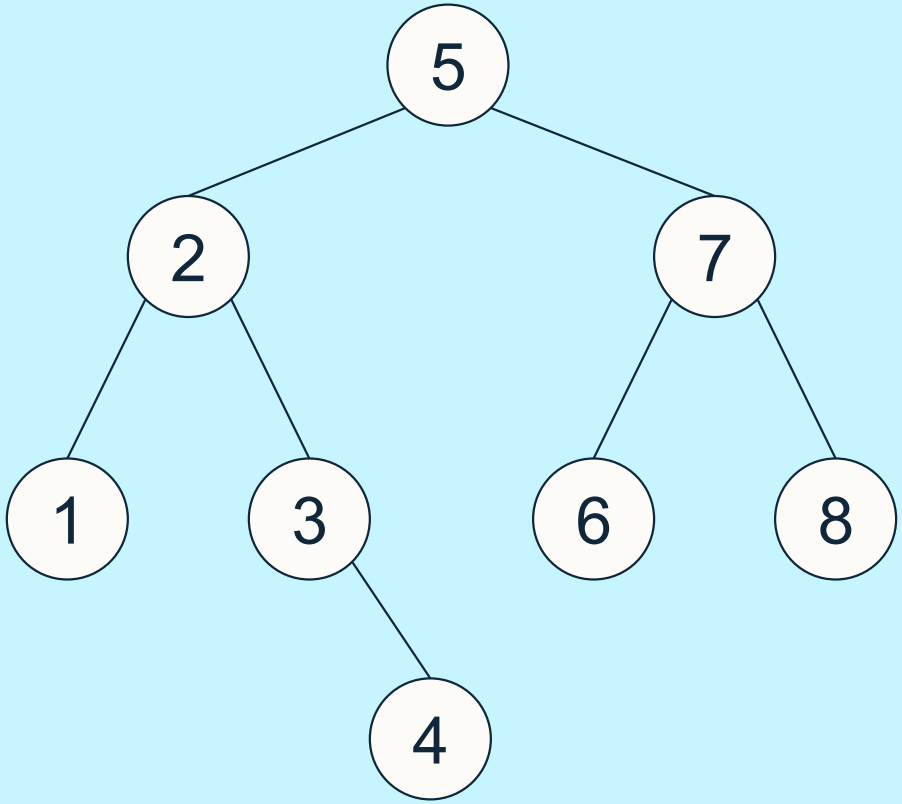
4 > 3 and 3 is a leaf node so make 4 the right child of 3.



Note that the in-order traversal of the resulting tree is now 1-2-3-4-5-6-7-8 which preserves numerical ordering.

Operations on BSTs – remove(x)

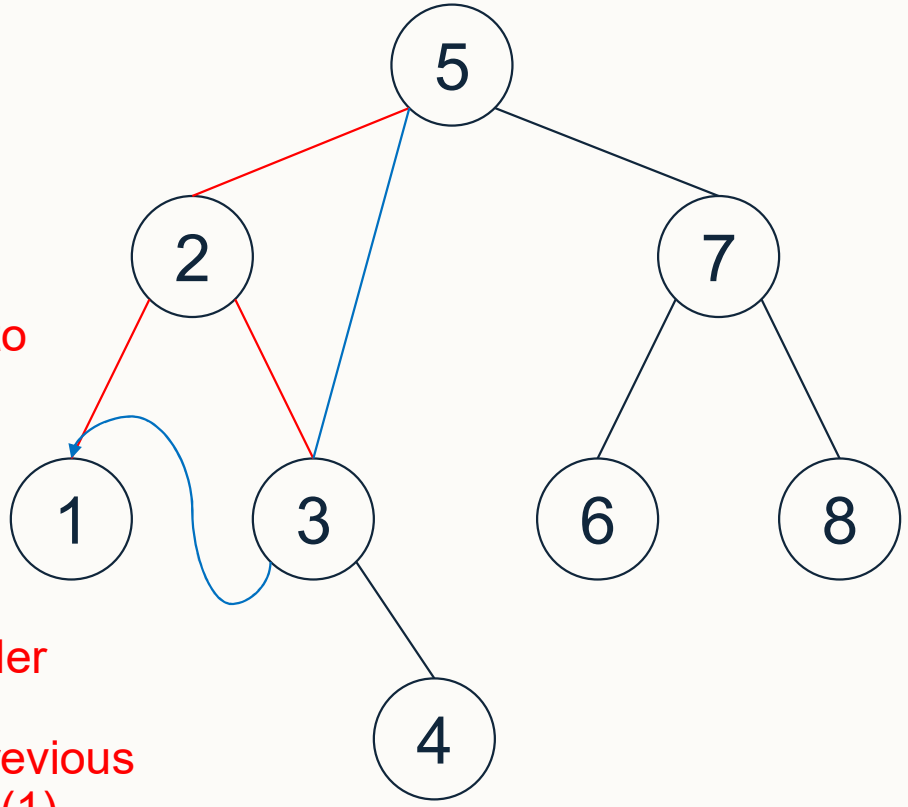
- Starting with the BST:



Perform remove(2), remove(7), then remove(5).

Operations on BSTs – remove(x)

- Starting with the BST:

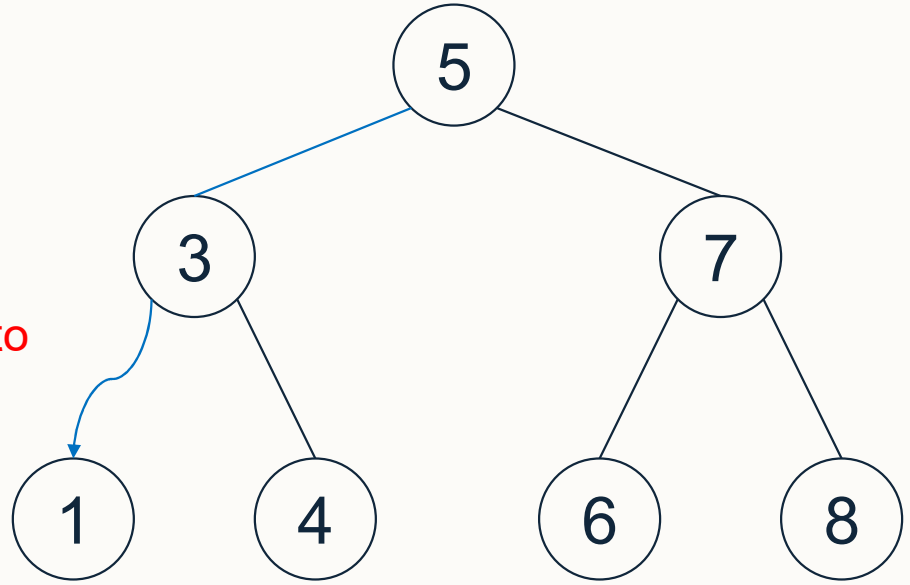


In this example, all of the nodes being removed have 2 children. The 1 child case is trivial – refer to the lecture notes.

To remove (2), we replace it with the node that follows in the in-order traversal which is (3).
The parent of (3) becomes the previous parent of (2) and the left child as (1).

Operations on BSTs – remove(x)

- Starting with the BST:

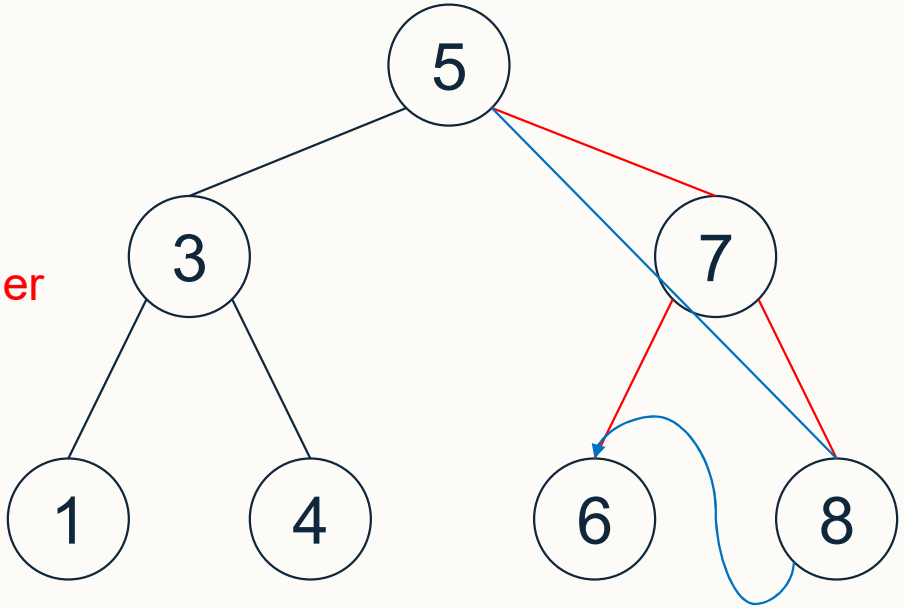


In this example, all of the nodes being removed have 2 children. The 1 child case is trivial – refer to the lecture notes.

To remove (2), we replace it with the node that follows in the in-order traversal which is (3).
The parent of (3) becomes the previous parent of (2) and the left child as (1).

Operations on BSTs – remove(x)

- Starting with the BST:

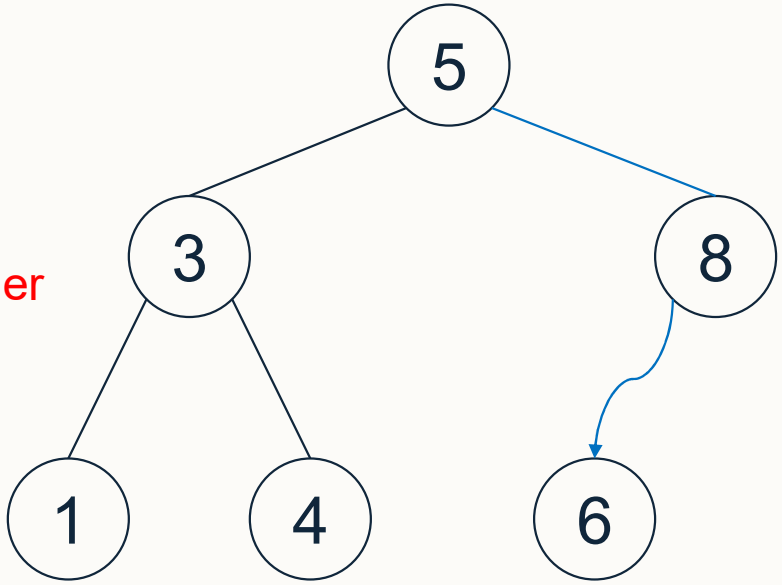


To remove (7), we replace it with the node that follows in the in-order traversal which is (8).

The parent of (8) becomes the previous parent of (7) and the left child as (6).

Operations on BSTs – remove(x)

- Starting with the BST:

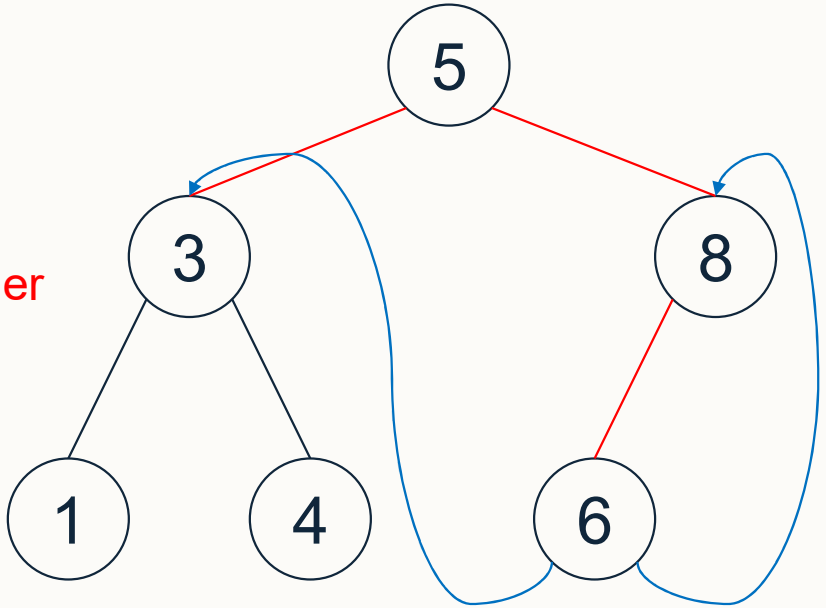


To remove (7), we replace it with the node that follows in the in-order traversal which is (8).

The parent of (8) becomes the previous parent of (7) and the left child as (6).

Operations on BSTs – remove(x)

- Starting with the BST:

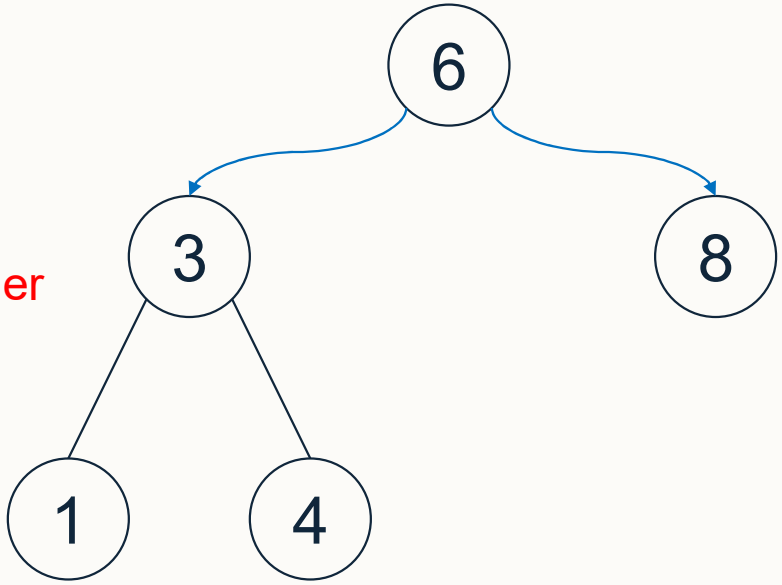


To remove (5), we replace it with the node that follows in the in-order traversal which is (6).

The parent of (6) becomes null, it is the root node, the left child becomes (3) and the right child (8).

Operations on BSTs – remove(x)

- Starting with the BST:



To remove (5), we replace it with the node that follows in the in-order traversal which is (6).

The parent of (6) becomes null, it is the root node, the left child becomes (3) and the right child (8).

Questions

Draw the following as binary trees and label each with “Heap”, “BST”, or “Neither”: $T_1 = [-, 1,2,4,7,8,5,6]$ and $T_2 = [-, 1,2,5,7,8,4,6]$.

If you identify any of these as a **heap**, then perform `x = removeMin()` followed by `insertItem(x)`.

- Is the resulting heap identical?

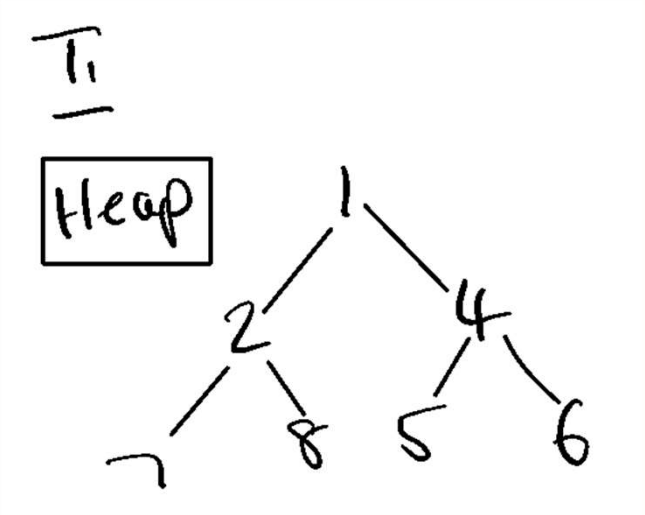
Use heapsort to sort the values [2,3,1,4,3’,2’,5]. Using your resulting sorted list, explain if heapsort is stable or not. Heapsort was covered in lec15 slides 39-40.

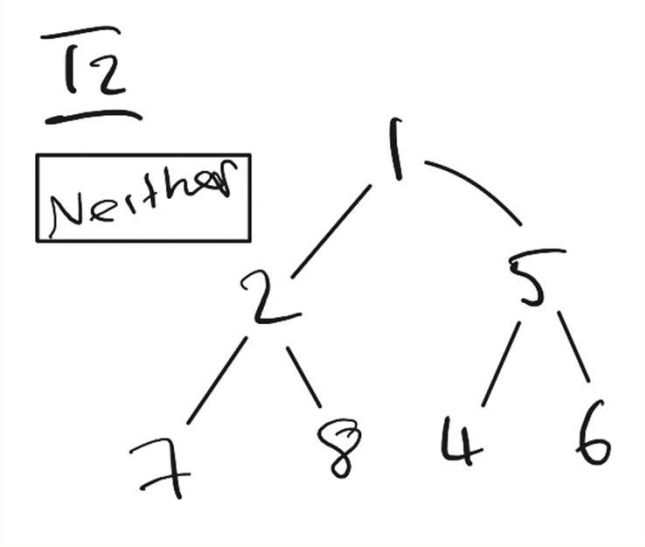
With the following BST $[-, 6,2,10,1,5,7,11, -, -, 3, -, -, 9, -, -]$ perform at least the following operations and state the array-based representation after each step:

- Remove(7); remove(2); insert(7); remove(11); remove(10).

Answers (1)

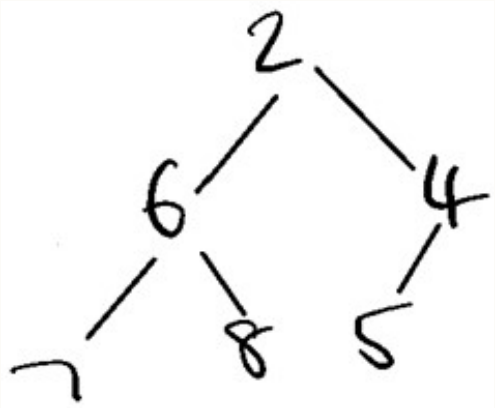
- Draw the following as binary trees and label each with “Heap”, “BST”, or “Neither”: $T_1 = [-, 1,2,4,7,8,5,6]$ and $T_2 = [-, 1,2,5,7,8,4,6]$.



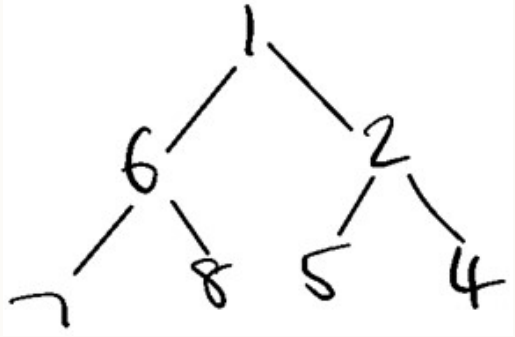


Answers (2)

- If you identify any of these (T_1 and T_2) as a **heap**, then perform `x = removeMin()` followed by `insertItem(x)`.
- Is the resulting heap identical?
- Only T_1 is a heap hence need to perform removeMin() and insertItem(1) which gives the following heap which is not identical.



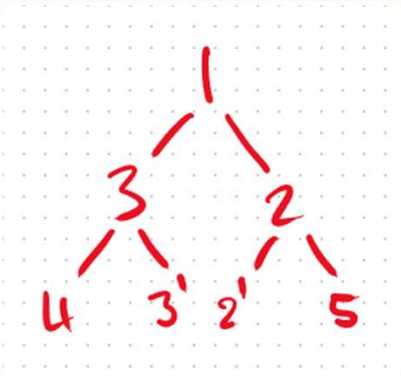
T_1 after removeMin()



T_1 after removeMin() followed by insertItem(1)

Answers (3)

- Use heapsort to sort the values [2,3,1,4,3’,2’,5]. Using your resulting sorted list, explain if heapsort is stable or not.
- Adding each element one-by-one into a heap will result in the following heap:

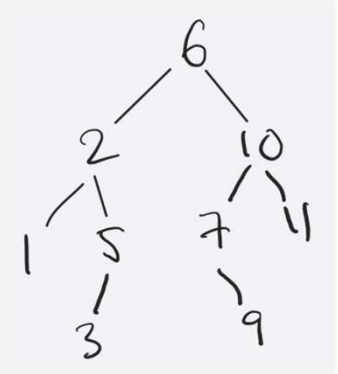


- …and then performing removeMin() 7 times will give the sorted ordering [1, 2, 2’, 3’, 3, 4, 5] which gives a counterexample for heapsort being stable since 3’ now appears before 3.

Answers (4)

With the following BST $[-, 6,2,10,1,5,7,11, -, -, 3, -, -, 9, -, -]$ perform at least the following operations and state the array-based representation after each step:

- Remove(7); remove(2); insert(7); remove(11); remove(10).
- Drawn as a tree, the BST is:



- Remove(7): the 7 is replaced with its single child 9 to give $[-, 6,2,10,1,5,9,11, -, -, 3, -, -, -, -, -]$.
- Remove(2): the 2 is replaced with the node 3 that follows in an in-order traversal to give $[-, 6,3,10,1,5,9,11, -, -, -, -, -, -, -, -]$.
- Insert(7): 7 > 6 so go to the right sub-tree; 7 < 10 so go to the left sub-tree, 7 < 9 and 9 has no left child so add 7 as the left child of 9 to give $[-, 6,3,10,1,5,9,11, -, -, -, -, 7, -, -, -]$.
- Remove(11): 11 has no children so can remove it to give $[-, 6,3,10,1,5,9, -, -, -, -, -, 7, -, -, -]$.
- Remove(10): 10 has one left child which itself has a left child. Set the parent of 9 to be 6 and the right child of 6 to be 9 to give $[-, 6,3,9,1,5,7, -, -, -, -, -, -, -, -, -]$.

University of Nottingham
UK | CHINA | MALAYSIA

Hash Maps

Array-based Hash Maps with Linear Probing

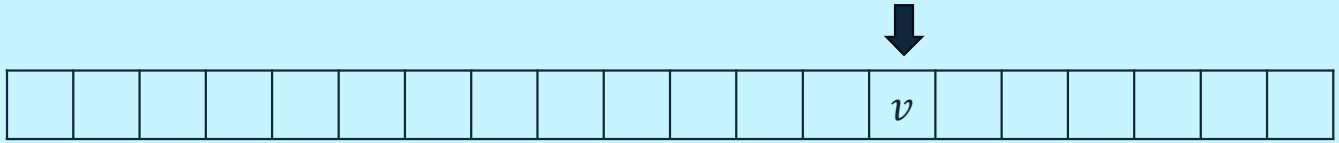
Hash maps

Hashing function to improve average case time complexity of operations from $O(\log n)$ for BST to $O(1)$.

Both BST and hash maps have a worst case of $O(n)$.

Worst case of BST can be improved with self-balancing (e.g. see https://www.geeksforgeeks.org/self-balancing-binary-search-trees/)

$$(k, v) \rightarrow h(k) \rightarrow index$$



A good hash function reduces collisions on the input keys (dispersal)…

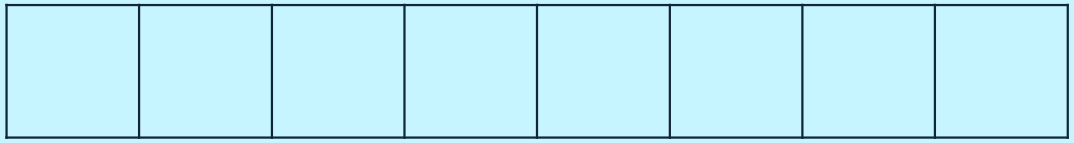
…but is unavoidable so need a mechanism to handle this.

Hash maps – Collision Handling

Can use separate chaining or a method of **open addressing** such as linear probing (today's focus).

Assuming a hash function $h(x) = x \,\%\, n$ where $n = 8$.

Insert the following values (in order): $\{2,4,8,16,32,64\}$



Hash maps – Collision Handling

Can use separate chaining or a method of **open addressing** such as linear probing (today's focus).

Assuming a hash function $h(x) = x \,\%\, n$ where $n = 8$.

Insert the following values (in order): $\{2,4,8,16,32,64\}$

Mapping h over the keys gives { 2, 4, 0, 0, 0, 0 }

8	16	2	32	4	64		

We can insert the keys { 2, 4, 8 } into the array at indices { 2, 4, 0 } respectively with no issues.

The problem now comes from inserting the remaining keys which are all mapped to [0] which is occupied.

- With linear probing, we start at [0] and scan right (looping around the array) until we find an empty position where we insert the value; hence, 16 goes into [1], 32 into [3], and 64 into [5].

Hash maps – Collision Handling

Can use separate chaining or a method of **open addressing** such as linear probing (today's focus).

Assuming a hash function $h(x) = x \,\%\, n$ where $n = 8$.

Insert the following values (in order): $\{2,4,8,16,32,64\}$

To do the get(k) with linear probing, need to continue scanning until the key is found, an empty cell is found, or we have inspected all cells.

E.g. to `get(2)` we hash 2 to get 2 and look into arr[2]. We find our key and return the value.

To `get(32)` we hash 32 to get 0 and look into arr[0]. We do not find our key so scan right until we find an empty position or the key we are looking for.

Hash maps – Collision Handling

Can use separate chaining or a method of **open addressing** such as linear probing (today's focus).

Assuming a hash function $h(x) = x \,\%\, n$ where $n = 8$.

Insert the following values (in order): $\{2,4,8,16,32,64\}$

8	16	2	32	4	64		

High number of collisions in both put(k,v) and get(k). How might we redesign the hash function if we know the pattern of the input keys?

In this specific example, the input keys are all powers of 2 so could set the hash function to be $h(x) = \log_2 x$ which disperses the keys well.
Note however if $x \geq 2^8$ when we need an extra component: $h(x) = \log_2 x \;\%\, 8$

Hash maps – Collision Handling

Can use separate chaining or a method of **open addressing** such as linear probing (today's focus).

Assuming a hash function $h(x) = x \,\%\, n$ where $n = 8$.

Insert the following values (in order): $\{2,4,8,16,32,64\}$

8	16	2	32	4	64		

Need to handle remove(k) using either lazy deletion or reinsertion.

Why is "just removing" a key incorrect?

Here if we were to just remove(32) by setting [4] back to "blank" than we could lose the 64 when performing linear probing. We would start at [0], scan right, and terminate at [4] despite 64 being in [6]. It is therefore necessary to use lazy deletion, marking deleted keys as `-1` or `D` to signal to the probing scheme "continue searching" or by reinserting everything to the right of [4] up until the next blank cell (reinsertion).

Questions

Insert the following keys into a hash map using linear probing with the hash function $h(k) = (k + 1)\ \%\ 7$ and $c = 1$ into an array of length 7: $\{4, 5, 11\}$.

Next remove 4 using an appropriate removal scheme.

Then explain how you find the key 11.

Using the below hash map and hash function, remove each of $\{2,8,16,32\}$ using lazy deletion, then again with reinsertion. Which method required the **least comparisons to find each key** and which method required the least comparisons overall (including comparison for the reinsertion)?

$h(x) = x\ \%\ n$ where $n = 8$

8	16	2	32				

Questions

- *"Insert the following keys into a hash map using linear probing with the hash function* $h(k) = (k + 1)\ \%\ 7$ *and* $c = 1$ *into an array of length 7:* $\{4, 5, 11\}$*."*

$h(\{4, 5, 11\}) \mapsto \{5, 6, 5\}$

Can insert 4 and 5 into [5] and [6] with no collisions:

[-, -, -, -, -, 4, 5]

Inserting 11 at [5] results in a collision. Scanning right, [6] already contains '4', and so wrap around and find [0] is empty hence can place 11 there.

[11, -, -, -, -, 4, 5]

Questions

- *"Next remove 4 using an appropriate removal scheme. Then explain how you find the key 11."*

Current array (hash map): [11, -, -, -, -, 4, 5]

To remove '4' we can mark [5] as deleted or reinsert '5' and '11'. Both schemes result in [11, -, -, -, -, "D", 5] or [-, -, -, -, -, 11, 5] respectively.

To find the key 11 with lazy deletion, we inspect [5] and find it is marked as deleted, so try [6] but $5 \neq 11$, next try [(6+1)%7] ([0]) which is 11 so return the associated value.

To find the key 11 with reinsertion, we inspect [5] and find 11 so return the associated value.

Questions

- *"Using the below hash map and hash function, remove each of* $\{2,8,16,32\}$ *using lazy deletion, then again with reinsertion. Which method required the* ***least comparisons to find each key*** *and which method required the least comparisons overall (including comparison for the reinsertion)?"*

$h(x) = x \,\%\, n$ where $n = 8$

8	16	2	32				

- To complete this exercise, you should do both removeAll with lazy deletion and again with reinsertion with the keys {2, 8, 16, 32} in that order.

	Lazy Deletion	Reinsertion
Find comparisons	8	4
Overall comparisons	8	19

University of Nottingham
UK | CHINA | MALAYSIA

Thank you