Problem 1 Assume the priority queue also must support the update DELETE(A,i), where i gives you the location in the data structure where element i is located. In a binary heap, that would be A[i]. Write pseudocode to achieve DELETE(A,i) in O(log n) time for binary heaps, where n is the size of the heap. Justify the running time.

Solution

To delete an element at i^{th} index in max heap we increase its value to positive infinite using HEAP-INCREASE-KEY function which leads i^{th} index value at the top of max heap. After calling the HEAP-EXTRACT-MAX function by which it removes that infinite value and i^{th} index value gets deleted.

```
DELETE(A, i)
1       HEAP-INCREASE-KEY(A, i, +∞)
2       HEAP-EXTRACT-MAX(A)
```

The time complexity of HEAP-INCREASE-KEY and HEAP-EXTRACT-MAX both is log n, so the total time complexity of DELETE will be log n.

```
HEAP-INCREASE-KEY(A, i, key)
1       if key < A[i]
2               error “new key is smaller than current key”
3       A[i] = key
4       while i > 1 and A[PARENT(i)] < A[i]
5               exchange A[PARENT(i)] with A[i]
6               i = PARENT(i)
```

The time complexity of HEAP-INCREASE-KEY on the n-element heap is O(log n) since traced from the node update in line 3 to root has length O(log n).

```
HEAP-EXTRACT-MAX(A)
1       if A.heap-size < 1
2               error “heap underflow”
3       max = A[1]
4       A[1] = A[A.heap-size]
5       A.heap-size = A.heap-size - 1
6       MAX-HEAPIFY(A, 1)                   // time complexity of O(logn)
7       return max
```

The time complexity of HEAP-EXTRACT-MAX is O(log n) since it performs only a constant amount of work on top of the O(log n) time for MAX-HEAPIFY.

```
MAX-HEAPIFY(A, i)
1       l = LEFT(i)
2       r = RIGHT(i)
3       if l <= A.heap-size and A[l] > A[i]
4               largest = l
5       else    largest = i
6       if r <= A.heap-size and A[r] > A[largest]
7               largest = r
8       if largest != i
9               exchange A[i] with A[largest]
10      MAX-HEAPIFY(A, largest)
```

```
PARENT(i)
1       return floor(i / 2)
```

LEFT(i)

```
1       return 2i
```

RIGHT(i)

```
1       return 2i + 1
```

Problem 2 Give an algorithm with worst-case running time of O(k log k) to print the k biggest elements in a n-element max-heap (binary heap stored in an array) without modifying the max-heap (or if you do modify it, restore it to the input heap within the given time bound). For example, if the max-heap is A = [100, 88, 44, 66, 77, 11, 22, 5, 55] and k = 5, the output could be 100, 88, 66, 77, 55 (the order in which the algorithm prints does not matter). Do not assume n = O(k). Here n can be much larger than k. A correctness proof is not needed (but your pseudocode must be correct). You can use additional data structures. Argue the running time.

Solution

```
'''
    Let's say our original max heap is A where a number of elements = n
    Here we create another max heap B which contains the value and index of
    original max heap A
    1. first in max heap B we insert a max element of A which is at 0th index
       of the array of A
    2. Then for k times we run a loop where we do the following:
        a. pop max element of max heap B and insert in k_max_elements list and
           print the value of the max element
        b. Now from max heap A we insert left and right child of the popped
           out max element using the index of it. This process takes log k
           times as number of elements present in max heap B will always be
           <= k
        c. we decrement value of k
    3. return the k_max_elements

    Here initially size of max heap B is 1 then with every iteration of loop
    size is decreased by 1 and increased by 2 so total increment will be 1
    with every k. So the max number of elements max heap B will contain is k.

    Here 2nd step runs for k times and 2.b step time complexity is O(log k)
    which are bottleneck for the algo so complexity of k_max_elements
    algorithm will be O(k log k).

    Here we are using tuples to compare values of max heap B while storing
    index of max heap A. So we are saving indexes in - so that in case two
    similar value elements are inserted consecutively in max heap B then the
    the weightage of more indexed value should be less because in max heap A
    more indexed value can be a child to less indexed value.
    eg.
            first element inserted is 100, index (0) tuple1 = (100, 0)
            second element inserted is 100, index (1) tuple2 = (100, 1)
            during max_heap_insert tuple1 < tuple2 and order of max heap A
            will not be maintained in max heap B as tuple2(100, 1) will go the
            max element of it. But if tuple1 = (100, 0) and tuple2 = (100, -1)
            then tuple1 > tuple2 so order of max heap A will remain in max
            heap B.


'''

1.      def k_max_elements(self, k: int):
```

```
2.          if k > len(self.elements):
3.                return Exception("number of k max elements is greater than
number of elements in the heap")

4.          k_max_elements = []
5.          k_element_and_index_heap = MaxHeap(None)
6.          k_element_and_index_heap.max_heap_insert((self.elements[0], 0))
7.          while k > 0:
8.              k -= 1
9.              max_element = k_element_and_index_heap.extract_max()
10.             print("Value of Max", max_element[0])
11.             max_element_value = max_element[0]
12.             k_max_elements.append(max_element_value)
13.             max_element_index = - max_element[1]

14.             left = self.left_child(max_element_index)
15.             if left < self.heap_size():
16.                 k_element_and_index_heap.max_heap_insert
17.                  ((self.elements[left], - left))

18.             right = self.right_child(max_element_index)
19.             if right < self.heap_size():
20.                 k_element_and_index_heap.max_heap_insert
21.                  ((self.elements[right], - right))

22.         return k_max_elements


"""
   To use k_max_elements algorithms we will require some MaxHeap class
   functions which is included in MaxHeap class
"""

1.   class MaxHeap:
2.       def __init__(self, elements: []):
3.           if elements is None:
4.               self.elements = []
5.           else:
6.               self.elements = elements
7.               self.build_max_heap()

8.       def left_child(self, i: int):
9.           return 2 * i + 1

10.      def right_child(self, i: int):
11.          return 2 * i + 2

12.      def parent(self, i: int):
13.          return (i - 1) // 2

14.      def heap_size(self):
15.          return len(self.elements)
```

```
    def build_max_heap(self):
        # if the len of a tree is 10 the index will be 0 to 9 and the first element we need to heapify will be on
        # first_heapify_node.
            first_heapify_node = len(self.elements) // 2 - 1
            for i in range(first_heapify_node, -1, -1):
                self.max_heapify(i)

    def max_heapify(self, i: int):
            left = self.left_child(i)
            right = self.right_child(i)
            largest = i
            if left < len(self.elements) and self.elements[left] > self.elements[largest]:
                largest = left
            if right < len(self.elements) and self.elements[right] > self.elements[largest]:
                largest = right
            if largest != i:
                self.elements[largest], self.elements[i] = self.elements[i], self.elements[largest]
                self.max_heapify(largest)

    def extract_max(self):
            if self.heap_size() == 1:
                return self.elements.pop()

            max_element = self.elements[0]
            self.elements[0] = self.elements.pop()
            self.max_heapify(0)
            return max_element

    def max_heap_insert(self, element):
            self.elements.append(element)
            i = self.heap_size() - 1
            while i > 0 and self.elements[self.parent(i)] < self.elements[i]:
                self.elements[self.parent(i)], self.elements[i] = self.elements[i], self.elements[self.parent(i)]
                i = self.parent(i)
```

Problem 3 Using only the definition of a binary search tree, show that if a node in a binary search tree has two children, then its successor has no left child and that its predecessor has no right child. Here all the keys in the binary search tree are distinct. The successor of a node j is defined as follows: if the node j has the biggest key, the successor is NIL; otherwise the successor is the node i that has a key that is bigger than the key of j and such that no other node has key between the key of i and the key of j.

Solution

Definitions to Understand

Binary Search Tree

Binary Search Tree is a Binary Tree which has the following properties:

- The left subtree of a node contains only nodes with keys lesser than or equal to node's key.
- The right subtree of a node contains only nodes with keys greater than or equal to node's key.
- The left and right subtree each must also be a binary search tree

Successor

The successor of a node j is defined as follows: if the node j has the biggest key, the successor is NIL; otherwise the successor is the node i that has a key that is bigger than the key of j and such that no other node has a key between the key of i and the key of j.

Predecessor

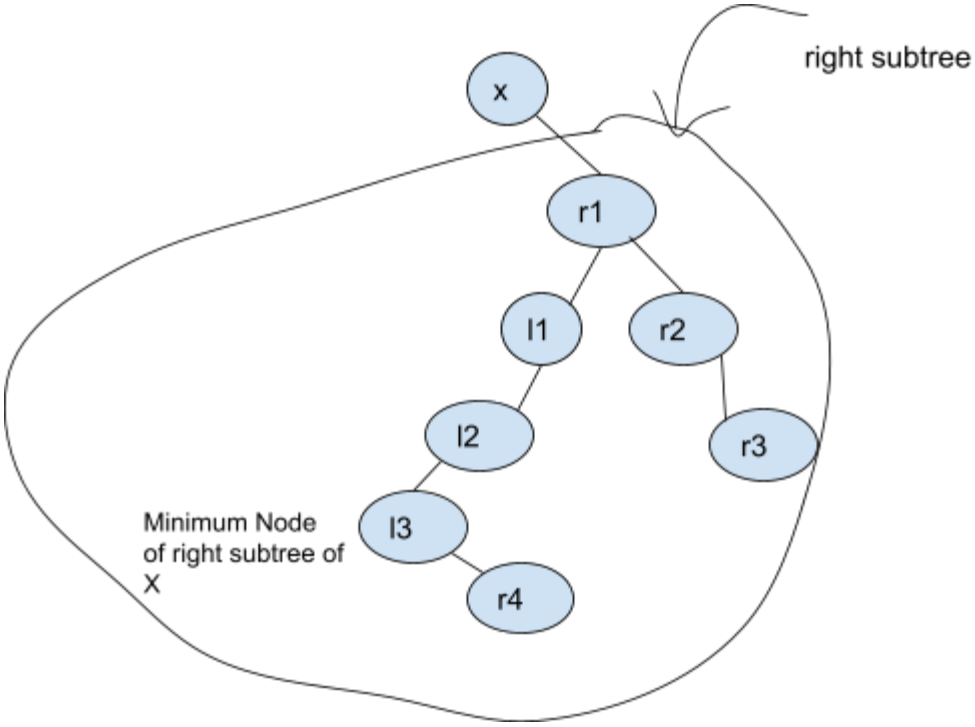
The predecessor of a node j is defined as follows: if the node j has the smallest key, the predecessor is NIL; otherwise the predecessor is the node i that has a key that is smaller than the key of j and such that no other node has a key between the key of i and the key of j.

Argument

For Successor

When node x has the right child then the successor will be the minimum of the right subtree. Minimum of any node in a binary search tree will always be in the left subtree until we reach the condition where no left subtree is possible. So Minimum of any tree does not contain any left child.
Example:

By the definition of binary search tree every left child will be <= of node and we have a condition here where all keys are distinct so left child < node. so l3 < l2 < l1 < r1. Now every other node in the right subtree is greater than l3 and also l3 does not have any left child.



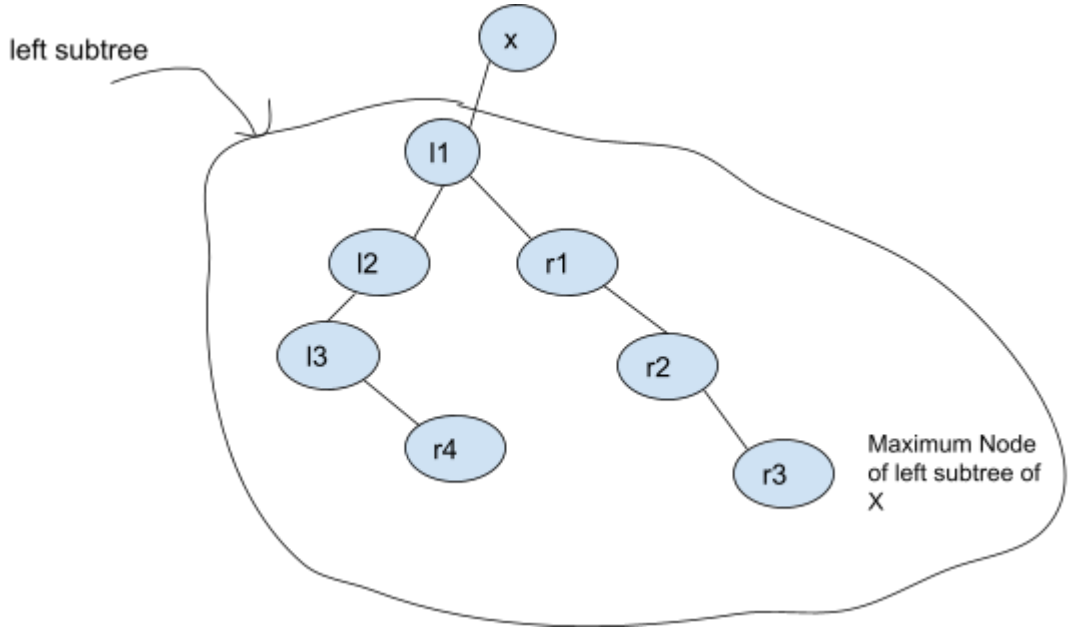
Therefore we can say that the minimum value of node x right subtree, which is the successor of node x, will also not contain any left child. Because if the minimum node contains the left child then then the left child of it will be going to have minimum value.

For predecessor

When node x has the left child then the predecessor will be the maximum of the right subtree. Maximum of any node in a binary search tree will always be in the right subtree until we reach the condition where no right subtree is possible. So Maximum of any tree does not contain any right child.
Example:

Here, by the definition of BST r3 > r2 > r1 > l1. because r3 is bigger than every node in the left subtree it will be the maximum value node. If we say a node is maximum then it can not have the right child because then the right child will be the maximum node according to bst property.

So the maximum node does not contain any right child.



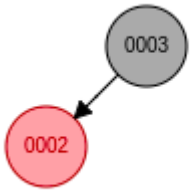
Therefore we can say that the maximum of node x left subtree, which is the predecessor of node x, will also not contain any right subtree. Because if the maximum node contains the right child then then the right child of it will be going to have maximum value.

Problem 4 Suppose that a node x is inserted into a red-black tree with RB-INSERT and then is immediately deleted with RB-DELETE. Is the resulting red-black tree the same as the initial red-black tree? Justify your answer.

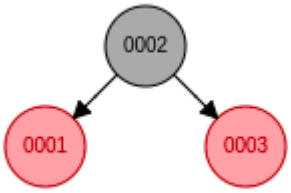
Solution

No, if x is inserted into a red-black tree with RB-INSERT and then is immediately deleted with RB-DELETE then the resulting red-black tree will not necessarily be the same as the initial red-black tree.

As we can provide counter example of this let's say we have the following red-black tree RB-T1:



Now we insert element x = 1 in the above red black tree, then after insertion red black tree will look like this:



Now after deleting the same element x = 1 in the above red black tree, the resulting red black tree RB-T2 will be as following:

As we can see RB-T1 is not identical to RB-T2, so we can say after inserting then deleting the same element in the red black tree does not produce the exact same tree, as there can be rotations to balance the tree which will lead to change in the structure of the red black tree.

Problem 5 Suppose you are given two n-element sorted sequences A and B, each representing a set (none has duplicate entries). Describe an O(n)-time method for computing a sequence representing the set A \ B (with no duplicates; note: the difference of sets A and B consists of those elements in A and not in B). You do not have to argue correctness (but, obviously, your method must be correct), but must justify the running time.
Solution

"""
Here every instruction from line 2 to 7 is of constant time so time complexity for line 2 to 7 will be O(c1) where c1 is constant.

*While loop will run here for len(first) + len(second) time for the worst case of line 8. For line 9 to line 16 every instruction is of constant time Let's say O(c2) so time complexity for whole while loop will be O(len(first) + len(second)) * O(c2) = O(n + n) * O(c2) = O(c2 * 2n). Here we can ignore the constant's so time complexity of the instruction from line 8 to line 16 will be O(n).*

Line 17 to line 19 is again of constant time so time complexity will be O(c3).

Total time complexity = O(c1) + O(n) + O(c3). Here the bottleneck for algorithm is O(n) so the time complexity of the whole algorithm will be O(n).
"""

```
1    def set_difference(first: [], second: []):
2        if first is None or second is None:
3            return Exception("Please provide proper Input :)")

4        first_ptr = second_ptr = 0
5        first_len = len(first)
6        second_len = len(second)
7        difference = []
8        while first_ptr < first_len and second_ptr < second_len:
9            if first[first_ptr] < second[second_ptr]:
10               difference.append(first[first_ptr])
11               first_ptr += 1
12           elif first[first_ptr] > second[second_ptr]:
13               second_ptr += 1
14           else:
15               first_ptr += 1
16               second_ptr += 1

17       if first_ptr < first_len:
18           difference += first[first_ptr:]

19       return difference
```