Solution - 5

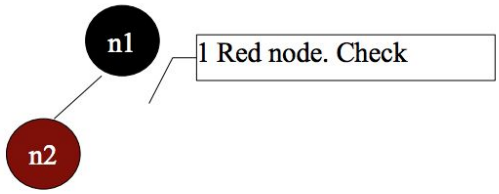
1.

There are many ways of proof. All reasonable can get points.

Proof by Induction:

Base Case:

N = 2



Inductive Hypothesis: Assume that for a red-black tree of n nodes, where $1 < n \leq N$, there exists at least 1 red node.

Inductive Step: Prove that for a red-black tree of $N+1$ nodes there exists at least 1 red node.

There are 2 cases we have to of interest on the $N+1$ th insertion.

Case 1: Red Node $N+1$ is inserted as a child of a black node. This is the base case which we proved to be true already. This is the trivial case.

Case 2: Red Node $N+1$ is inserted as a child of a red node. We must look at the insertion cases that have X as a child of a red node P.

Insertion Case 1:	X remains red after the recoloring of P, G, U. It also remains the same color after we move reference X to Grandparent of X. Thus we have atleast 1 red node.
Insertion Case 2:	The parent P of X remains red after the Zig-Zag rotation of X about P and then G. P remains red after recoloring X and G. Thus we have atlease 1 red node.
Insertion Case 3:	X remains red after the rotation of P about G. X remains red after the recoloring of P and G. Thus we have atleast 1 red node.
Insertion Case 0:	Not included since $n! = 1$, Insertion Cases 2 & 3 are terminating conditions, and Insertion Case 1 still produces 1 red node.

2.

Max 2000^7-1=1.28 * 10^23 - 1

Min 2 * 10^18 - 1

3.

```
1: procedure TREE-FIND-MIN(x)
2:     if x == NIL then                                   ▷ Tree is empty
3:         return NIL
4:     else if x.leaf then                                ▷ x is leaf
5:         return x.key1                                  ▷ return the minimum key of x
6:     else
7:         DISK-READ(x.c1)
8:         return B-TREE-FIND-MIN(x.c1)
```

Predecessor rules:

(a) If x is not a leaf, return the maximum key in the i^{th} child of x, which is also the maximum key of the subtree rooted at $x.c_i$.

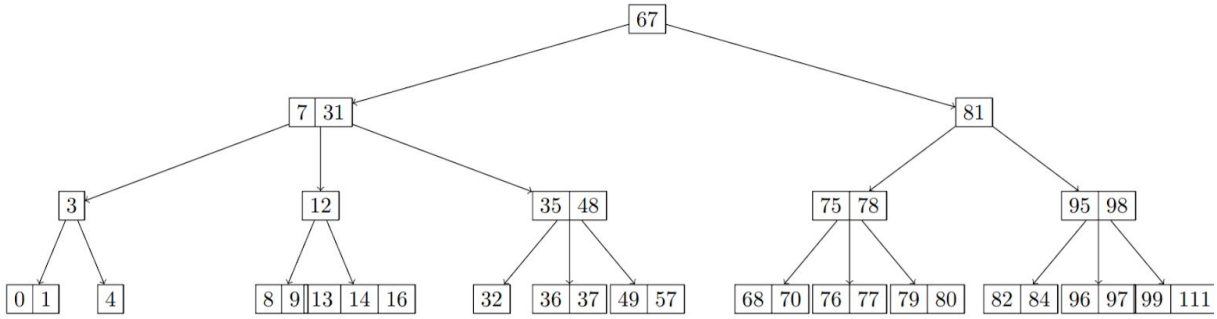
(b) If x is a leaf and $i > 1$, return the $(i-1)^{th}$ key of x, *i.e.*, $x.key_i - 1$

(c) Otherwise, look for the last node y (from the bottom up) and $j > 0$, such that $x.key_i$ is the leftmost key in $y.c_j$; if $j = 1$, return NIL since $x.key_i$ is the minimum key in the tree; otherwise we return $y.key_j - 1$.

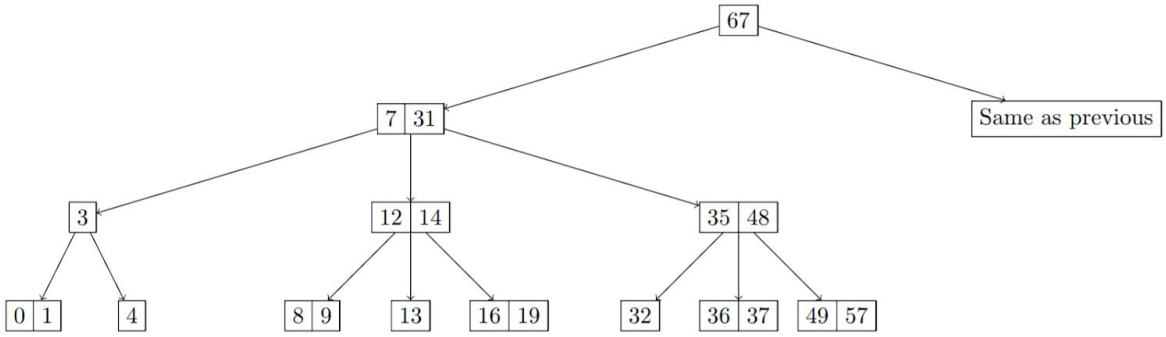
```
1: procedure B-TREE-FIND-PREDECESSOR(x, i)
2:     if x is not leaf then
3:         DISK-READ(x.c_i)
4:         return B-TREE-FIND-MAX(x.c_i)
5:     else if i > 1 then                                 ▷ x is a leaf and i > 1
6:         return x.key_i − 1
7:     else                                               ▷ x is a leaf and i = 1
8:         z = x
9:     while true do
10:        if z.p == NIL then                             ▷ z is root
11:            return NIL                                 ▷ z.key_i is the minimum key in T; no predecessor
12:        y = z.p
13:        j = 1
14:        DISK-READ(y.c_1)
15:        while y.c_j ≠ x do
16:            j = j + 1
17:            DISK-READ(y.c_j)
18:        if j == 1 then
19:            z = y
20:        else
21:            return y.key_j − 1
```

4.

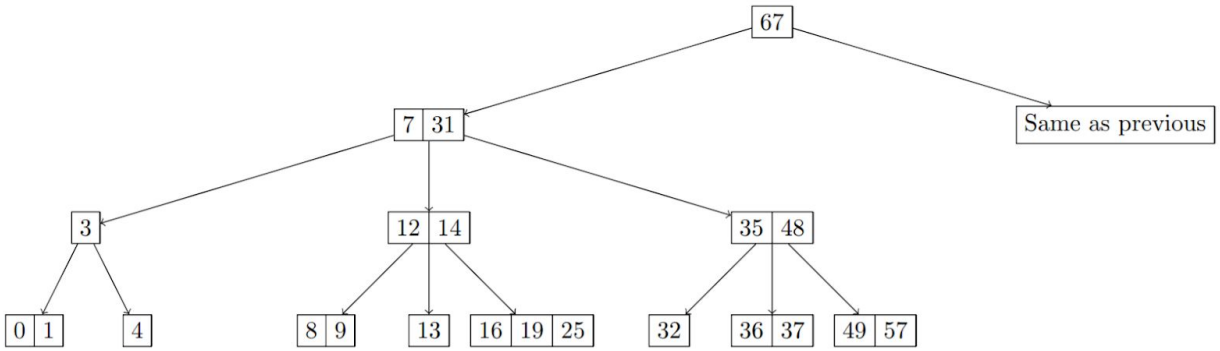
(a) Insert 16:



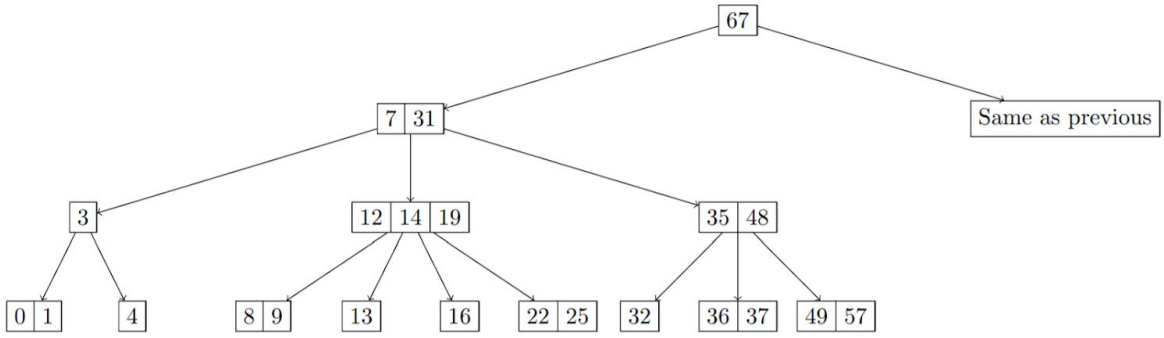
(b) Insert 19:



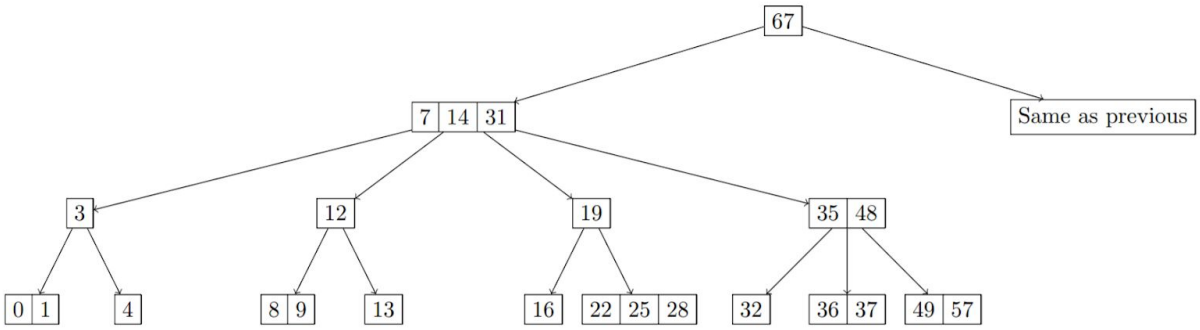
(c) Insert 25



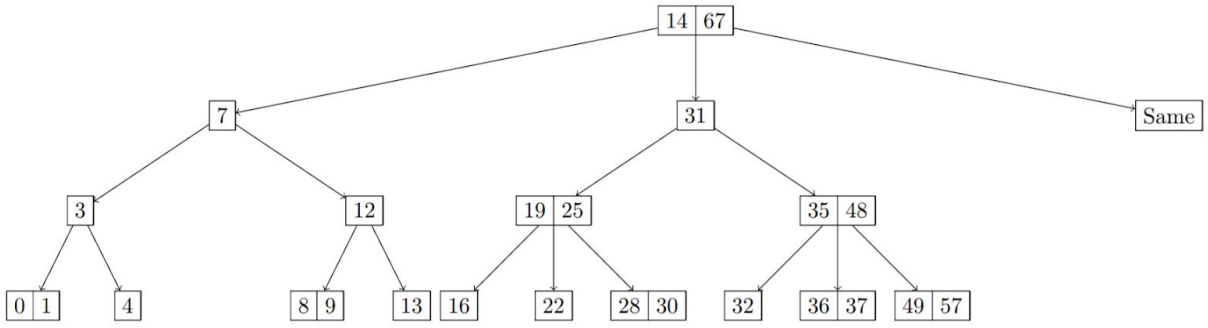
(d) Insert 22:



(e) Insert 28:



(f) Insert 30:

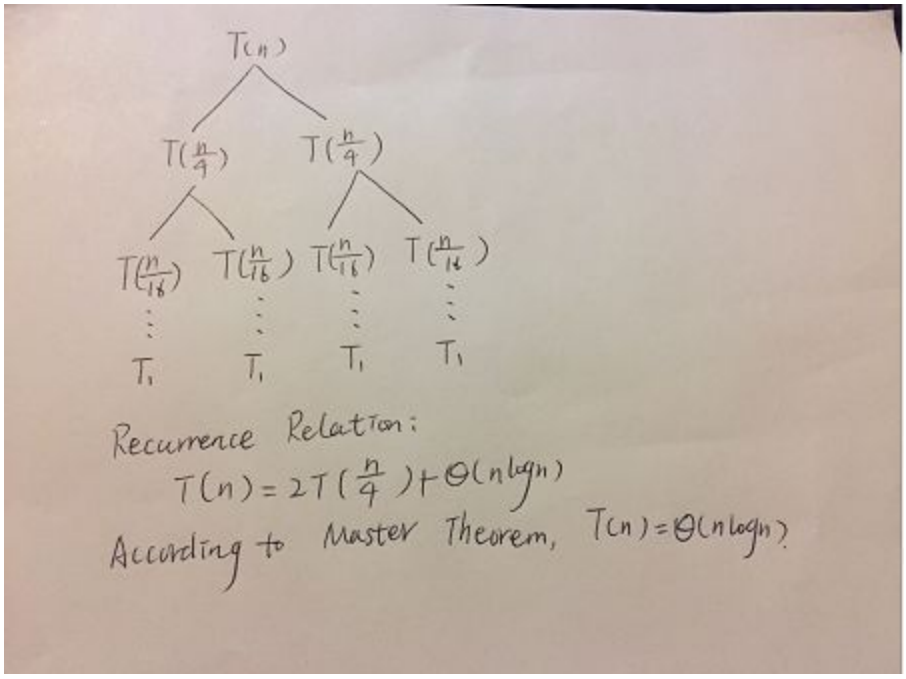


5.

Induction hypothesis: $T(k) \leq ck\log^2 k \ \ \forall k \ < \ n$

$$
\begin{aligned}
&T(n) = 2T(n/2) + bnlogn \\
&\leq 2(c(n/2)\log^2(n/2)) + bn\log n \\
&\leq cn(\log^2(n/2)) + bn\log n \\
&\leq cn(\log n - \log 2)^2 + bn\log n \\
&\leq cn(\log^2 n - 2logn) + bn\log n \\
&\leq cn\log^2 n - 2cn\log n + bn\log n \\
&\leq cn\log^2 n
\end{aligned}
$$

6.



7.

For 2 matrices A and B. If A is $n \times 3n$ and B is $3n \times n$, then we can express the 2 matrices as follows.

$$A = \begin{bmatrix} A_1 & A_2 & A_3 \end{bmatrix}, B = \begin{bmatrix} B_1 \\ B_2 \\ B_3 \end{bmatrix}$$

Then the outcome of $A \times B$ is as follows.

$$A \times B = \begin{bmatrix} A_1 B_1 + A_2 B_2 + A_3 B_3 \end{bmatrix}$$

And the result contains 3 multiplications and each of them is a $n \times n$ matrix multiplied by a $n \times n$ matrix, so it can be calculated using Strassen?s algorithm, and the running time is $\theta(3n^{log7})$, which equals to $\theta(n^{log7})$.

8.

Solution 1: (do not support deletion)

Since the question does not require the system to support deletion, we can use the following approach.

First, we maintain a Red-Black tree, where each node contains a timestamp t. The nodes are sorted in the increasing order of timestamps.

- **near_miss()**: We maintain a global minimum, which is initialized to Integer.MAX_VALUE. This method simply returns the global minimum, which take O(1) time.
- **already_reserved(t)**: This method is equivalent to searching a node in the Red-Black tree, which can be accomplished in O(logn) time.
- **reserve(t)**: This is equivalent to inserting a new node into the Red-Black tree. After we insert the new node, we find its **predecessor** and **successor** [O(logn)]. Then we compute the difference between the node and its predecessor and successor, and we update the global minimum accordingly.

In order to find predecessor and successor, our tree node will need to store an additional pointer that points to its parent. Then we can find predecessor and successor in O(logn) time.

More detail about how to find predecessor and successor here:
https://www.quora.com/How-can-you-find-successors-and-predecessors-in-a-binary-search-tree-in-order

Solution 2: (support deletion)

If we want to support deletion, we will need more information in each node. We still maintain a Red-Black tree.

Every node now will contains:

1. ts: the tree will be sorted based on the timestamp
2. min: minimum value in the subtree rooted at the current node
3. max: maximum value in the subtree rooted at the current node
4. minDiff: minimum difference in the subtree rooted at the current node
5. left: a pointer to the left child
6. right: a pointer to the right child
7. parent: a pointer to the parent

- **near_miss()**: We only need to look at the `minDiff` value at the root node. Time: O(1)
- **already_reserved(t)**: This method is equivalent to searching a node in the Red-Black tree, which can be accomplished in O(logn) time.
- **reserve(t)**: This is equivalent to inserting a new node into the Red-Black tree. After the insert process, we update nodes by traversing nodes upwards, and we update the values that they contain.

reserve(t):

```
node = Red_Black_Tree_Insert(t);

while (node != null) {

    node.min = Math.min(node.left.min, node.ts);

    node.max = Math.max(node.ts, node.right.max);

    node.minDiff = Math.min(node.left.minDiff, node.right.minDiff, node.ts - node.left.max,
node.right.min - node.ts);

    node = node.parent;

}
```

Since the height of the tree is O(logn), so the time complexity of this process will be O(logn).

Similarly, when we delete a node, we can update the tree using similar process.