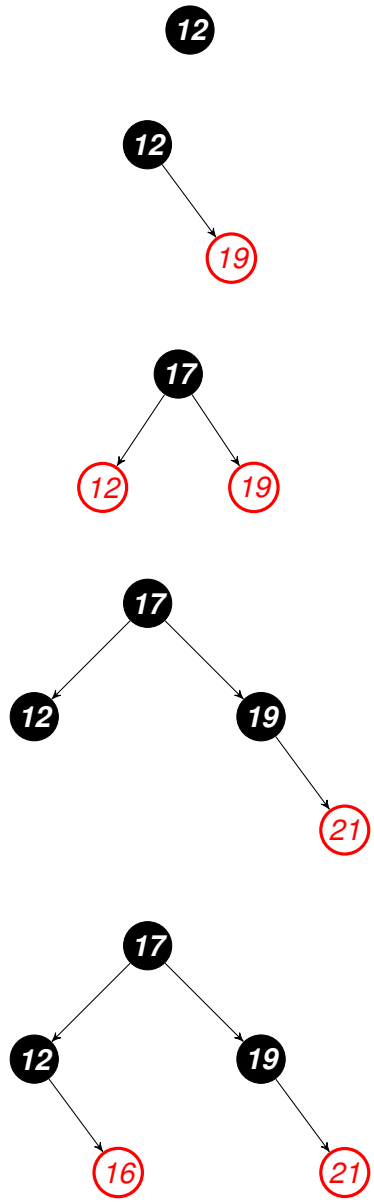
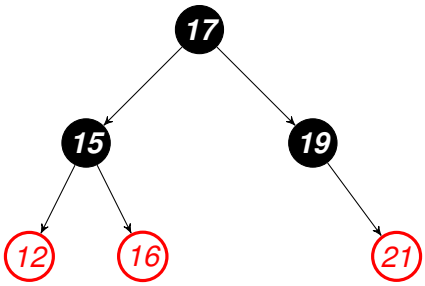
CSCD320 Homework4

Ethan Tuning

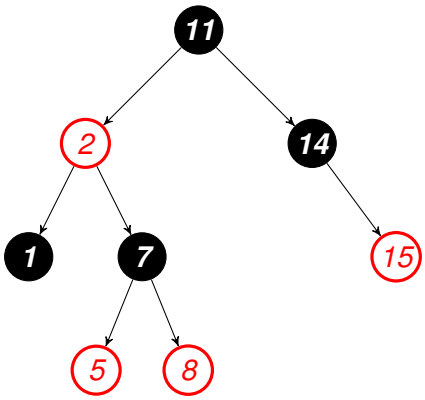
Problem 1. *Show the trace of the construction the Red-Black tree for the sequence 12, 19, 17, 21, 16, 15. That is, you need to draw the state of the tree after inserting each number.*

Answer 1. *The following pictures show the R-B tree after the keys have been added from left to right from the problem.*

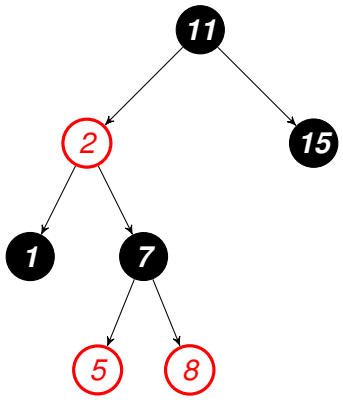


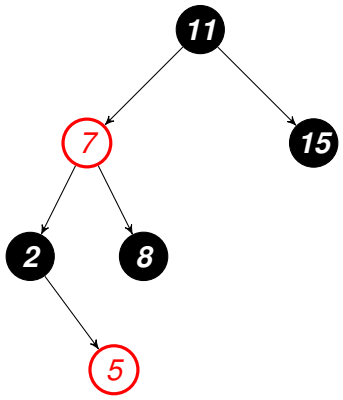


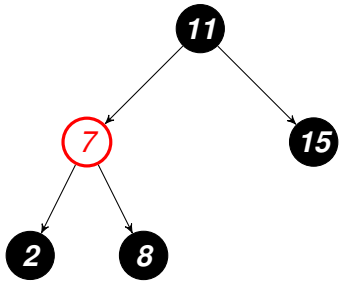
Problem 2. *Show the trace of deleting the nodes from the Red-Black tree below in the order of 14, 1, 5, 2, 11, 7, 15, 8. That is, show the state of the tree after deleting each node.*

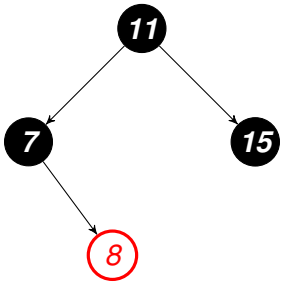


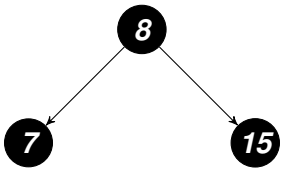
Answer 2. *The following pictures show the R-B tree after the keys have been deleted from left to right from the problem.*

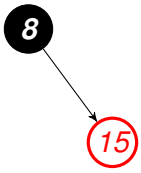



11
7
15
2
8
5


11
7
15
2
8


11
7
15
8

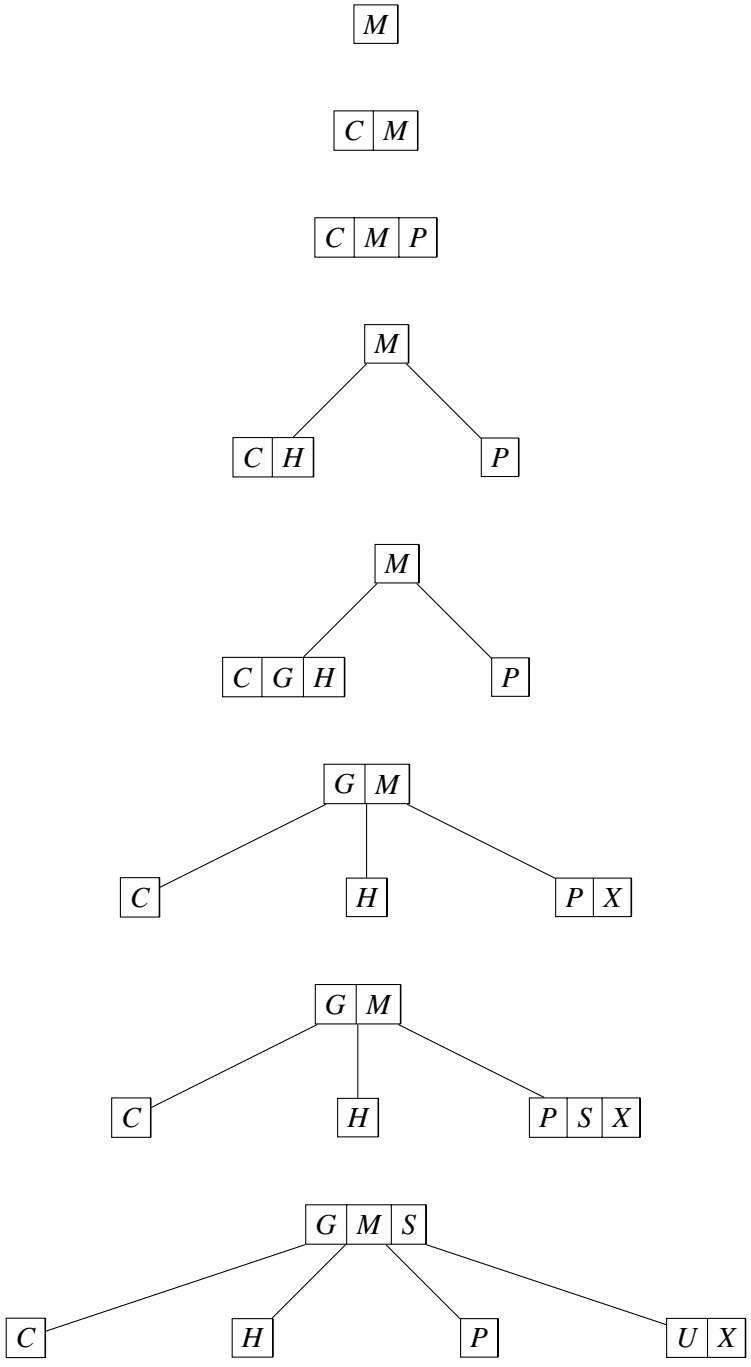

8
7
15

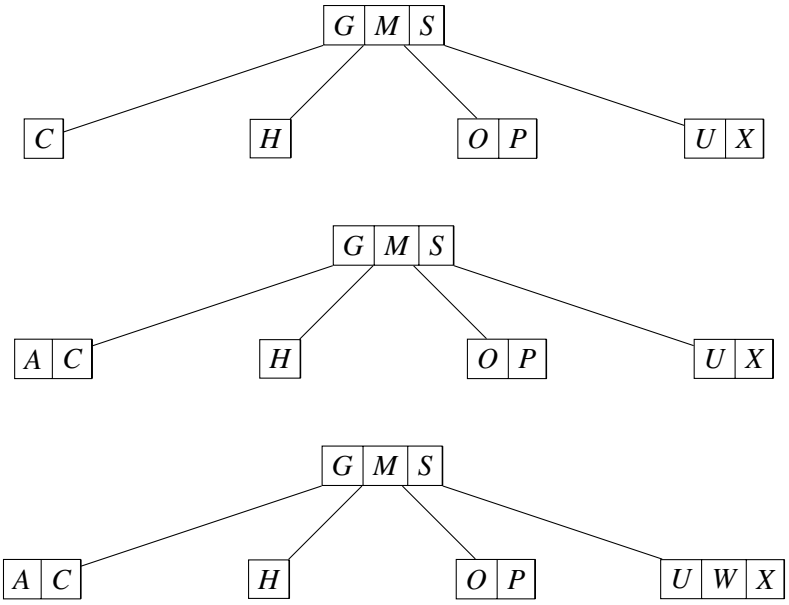

8
15


8

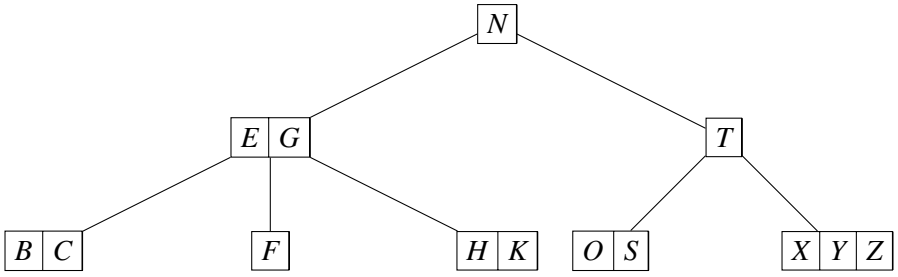
Problem 3. *Trace the ONE-PASS construction of the B-tree for the sequence* $M, C, P, H, G, X, S, U, O, A, W$. *Draw the configuration of the B-tree after inserting each letter. We use t = 2 as the branching degree threshold of the B-tree, so that:* (1) *all the non-leaf nodes must have at least* $t-1=1$ *keys and at most* $2t-1=3$ *keys; and* (2) *The root node of an non-empty B-tree must have at least one key and at most* $2t-1=3$ *keys.*

Answer 3. *The following pictures show the B-tree after the keys have been added from left to right from the problem.*

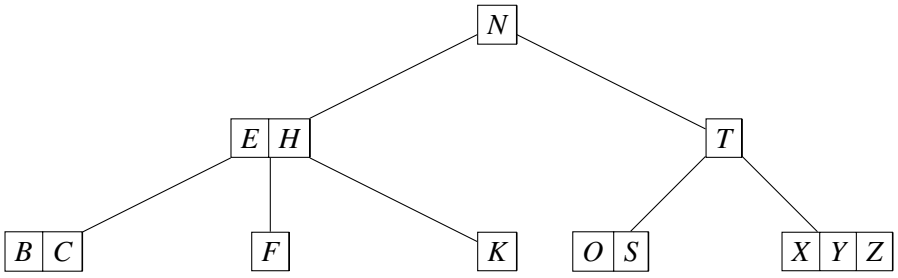


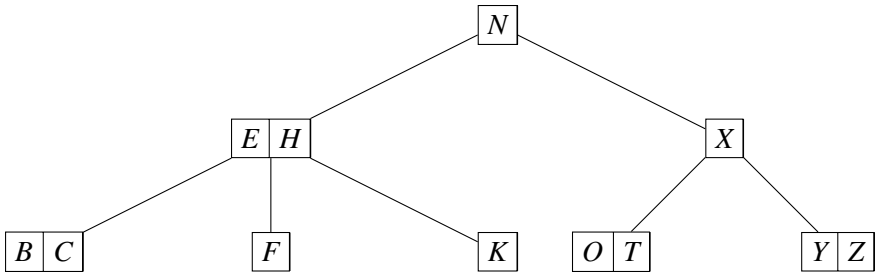


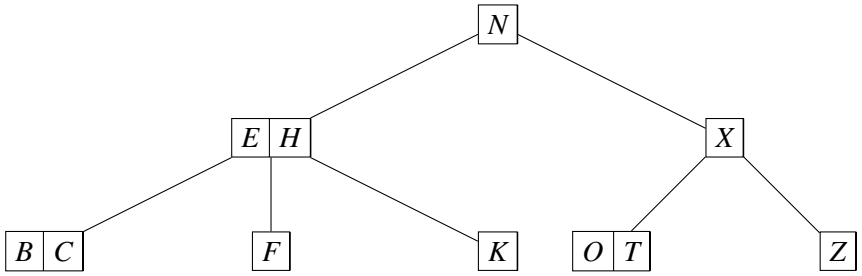
Problem 4. *Trace the deletion of the sequence of keys G, S, Y, T, F from the B-tree below. Draw the configuration of the B-tree after each deletion. We use $t = 2$ as the branching degree threshold of the B-tree, so that:* (1) *all the non-leaf nodes must have at least $t - 1 = 1$ keys and at most $2t - 1 = 3$ keys; and* (2) *the root node of a non-empty B-tree must have at least one key and at most $2t - 1 = 3$ keys.*

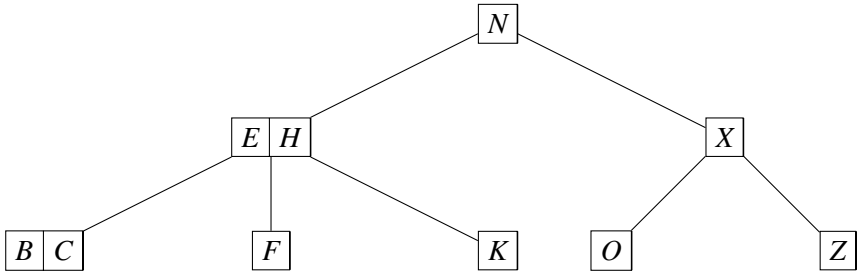


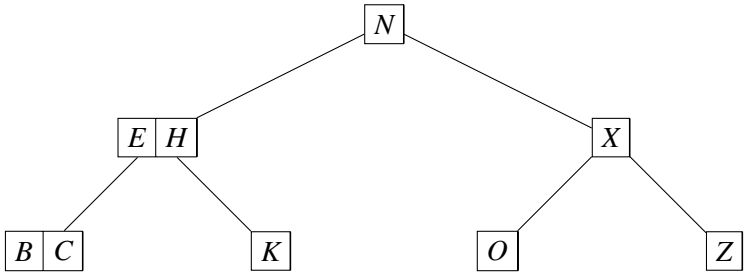
Answer 4. *The following pictures show the B-tree after the keys have been deleted from left to right from the problem.*











Problem 5. *We know binary search trees support the operations of finding* (1) *the minimum and maximum node of a given subtree; and* (2) *the successor and predecessor of a given node in the tree. Now you are asked to present the algorithmic idea and the pseudocode of the operations below for B-trees. Give the time cost of your algorithm in the big-oh notation.*

Answer 5. *For the maximum method we will just traverse over to the right most node. Then we will grab the greatest value of all the keys. Since the B-tree puts everything greater to the right and smaller goes to the left.*

```
max(node root){
  if(root == null)
    return null
  if(root.rightChild != null)
    max(root.rightChild)
  else
    return (greatest key in root)
}
```

The time cost here would of course be $O(\log n)$. This is because the height of the tree is bounded by $\log n$.

For the predecessor method we would just iterate the left sub-tree and find the maximum key.

```
predecessor(root, key){
  if(root != null)
    if(root.leftChild != null)
      node temp = root.leftChild;

      while(temp.right != null){
        temp = temp.rightChild;
      }
      return temp.data;
    }
  }
  return null;
}
```

Of course, again the time cost of this algorithm will be $O(\log n)$

Problem 6. *We have raised the new challenges in external memory based sorting. Suppose we need to sort a massive data set, which unfortunately cannot fit into the RAM of the machine. Propose any good idea for sorting this data set on external memory such as the hard disks, so that the performance will still be acceptable. Not that the guiding rule for external algorithm design is to make full use of and minimize the I/O operations, while the processing at the RAM is still maintained as efficient as possible. Use the Internet and cite your sources.*

Answer 6. *So for this issue what first comes to my mind is loading the data into main memory in sizes that can fit into the RAM. So with that we could just load in as much of that data at one time, read, and perform some sort, then write that data to some file. Then we could just keep doing that until all of the data is sorted within their respective chucks. Then we would just use a merge method(merge sort's merge) to merge the chucks that were stored in the temporary file. It would be some sort of flavor of merge sort but on a greater scale. Same concept as merge sort, we just have to do the entire algorithm in chucks to account for the massive data size.*