We have been considering search trees in which we can search, insert, delete, and perform other elementary operations in $O(\log n)$ time.

What if we don't want to search for an element by its value, but instead want to find an element in a search tree based off of some other characteristic?

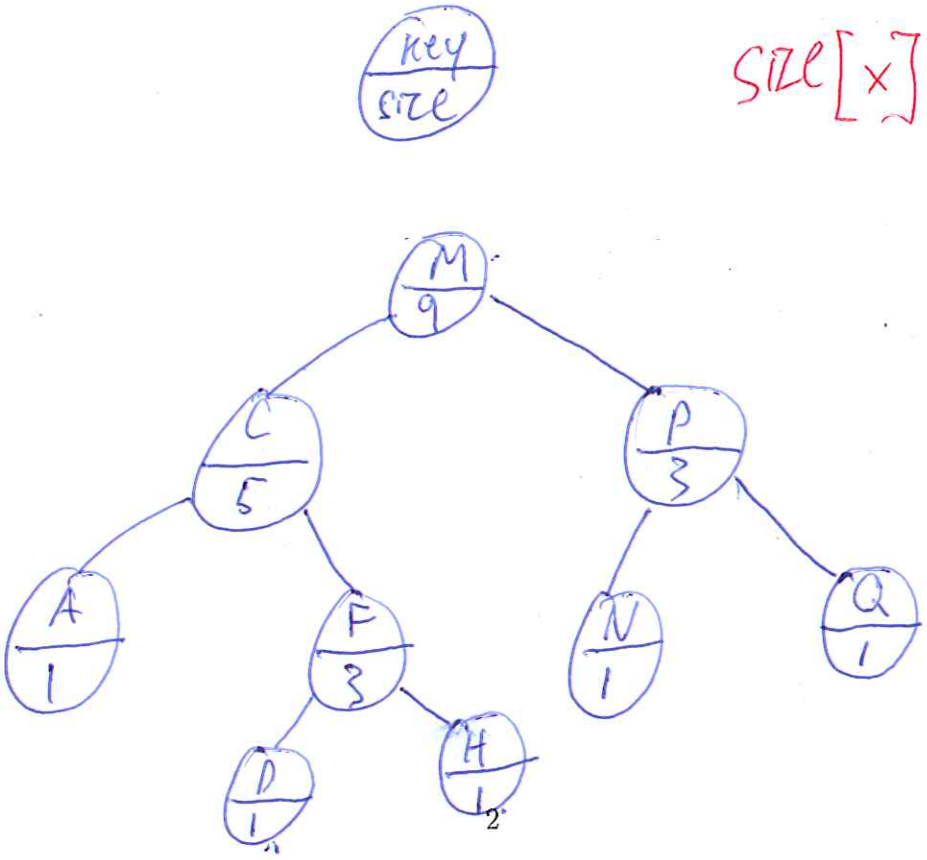
For example, we might want to find the ith smallest element in a search tree. In the search trees we have considered, we have no way of identifying the ith smallest element in $O(\log n)$ time (even in a red-black tree).

Idea: store a constant amount of additional information in each node in a way so that more complicated operations can be performed in $O(\log n)$ time.

Suppose we want to keep a dynamic search tree that can perform the following operations in $O(\log n)$ time:

- OS-Select(i, S): return the ith smallest element in the dynamic set S.
- OS-Rank(x, S): return the rank of $x \in S$ in the sorted order of the elements in S.

Use a red-black tree for the set S, and each node also stores the number of nodes in its subtree.



size[x] = size[left[x]] + size[right[x]] + 1

Implementation of OS-Select(x, i). Assume leaf nodes have NIL children such that $size[NIL] = 0$.

Initial call: OS-Select(root, i)

```
k ← size[left[x]] + 1   // k is the rank of x
if k = i then return x
if i < k
    return OS-Select(left[x], i)
else
    return OS-Select(right[x], i-k)
```

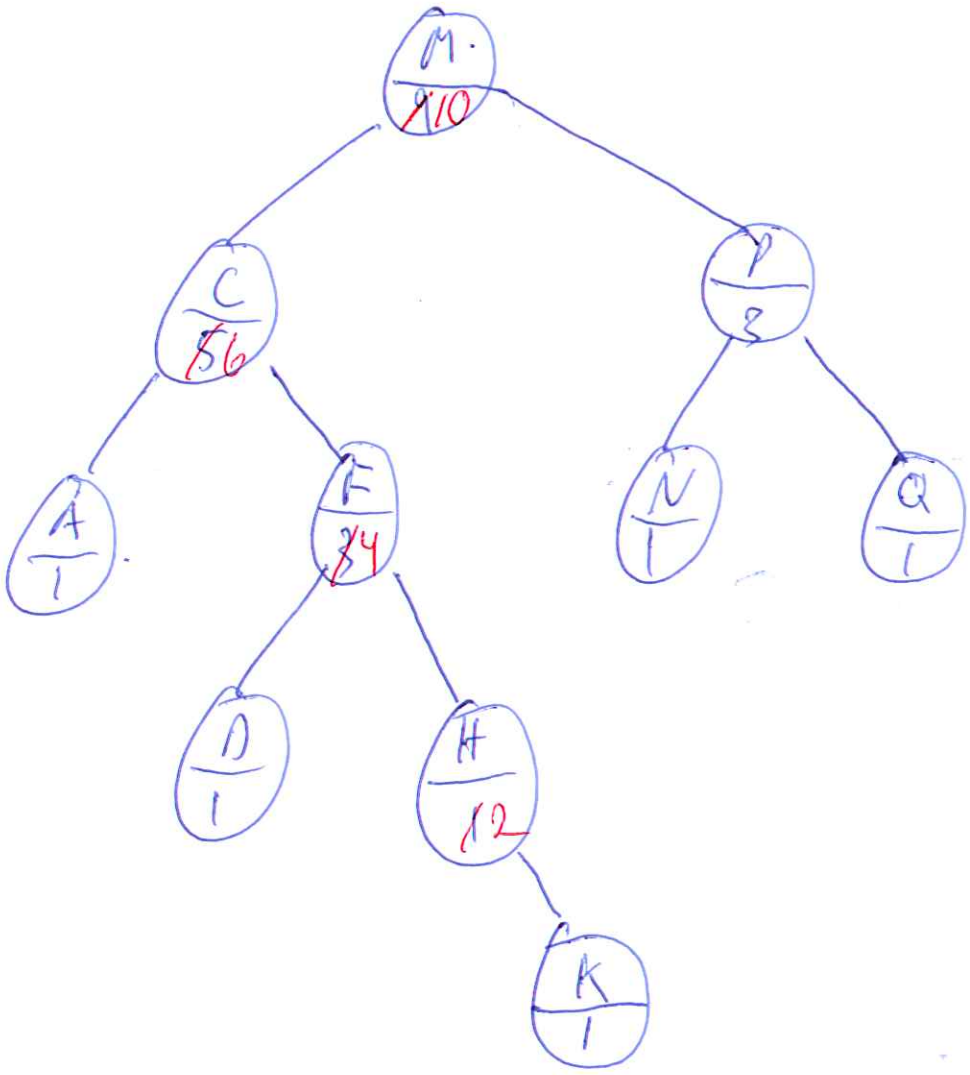
Why do we keep the the size of the subtree instead of the rank of a node itself?

Whenever we insert or delete a node, we need to be able to quickly update each of these values. It is more costly to maintain than the node's rank than it is the number of nodes in its subtree.

- For example, if we were to insert a node that is less than the root of the tree, then we would need to increment the rank of all of the nodes in the right subtree of the root.

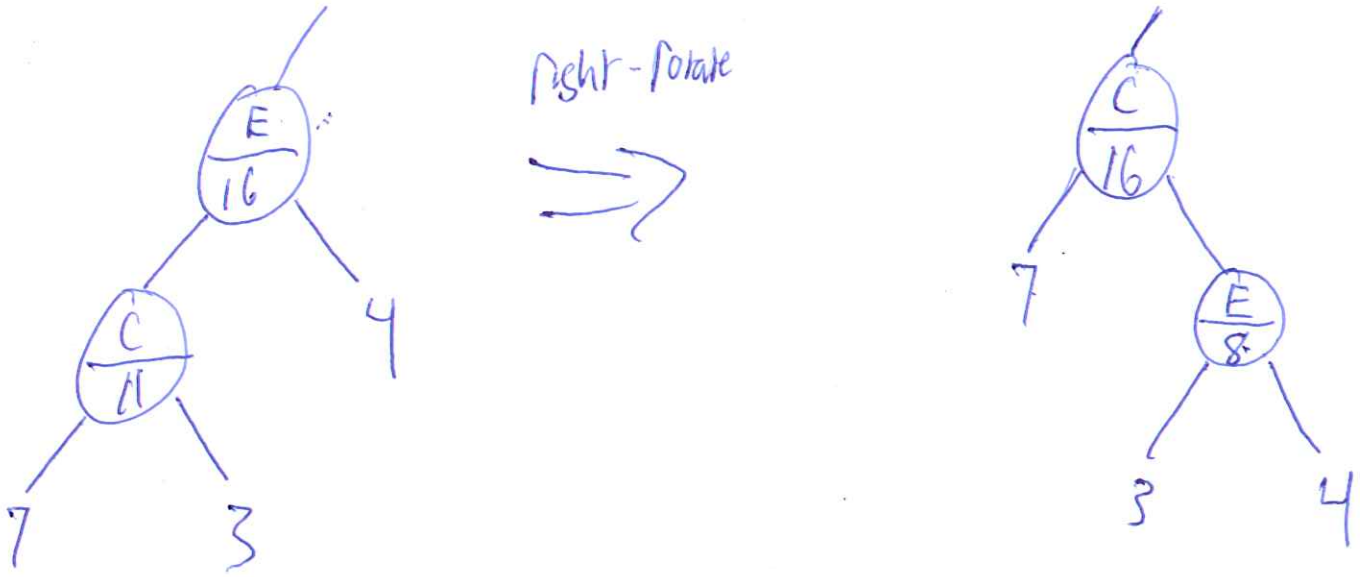
Recall that our underlying data structure is a red-black tree. How can we update the subtree sizes when we insert or delete a node into this tree?

Insert("K")



Inserting and deleting in red-black trees can cause recolorings and rotations.

- Recoloring has no effect on subtree size.
- Rotations may effect the subtree sizes.



A set of guidelines which may be useful when augmenting a data structure to allow for additional operations:

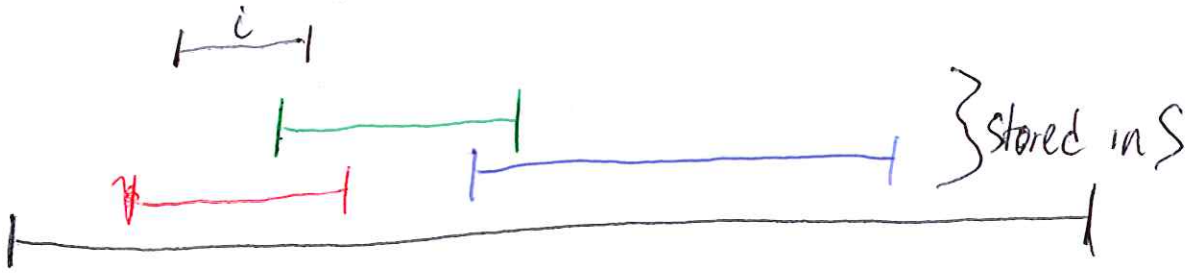
1. Choose an underlying data structure (e.g. red-black tree).

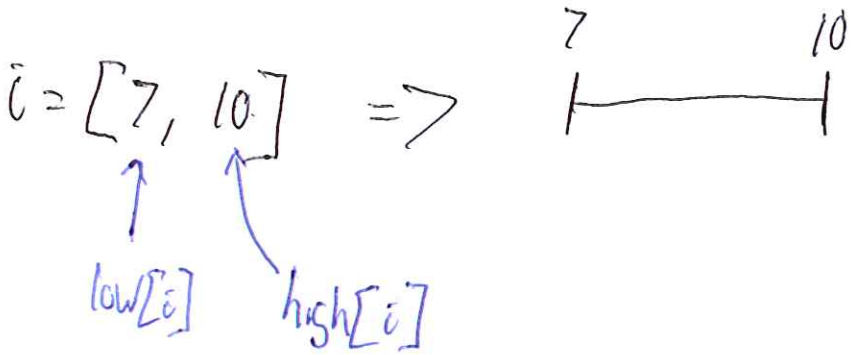
2. Determine additional information to maintain in the underlying data structure (e.g. subtree sizes).

3. Verify that we can maintain the additional information for the basic modifying operations on the underlying data structure (e.g. RB-Insert, RB-Delete, and rotations).

4. Develop new operations which can utilize this new information (e.g. OS-Select and OS-Rank).

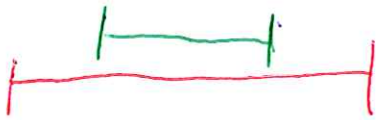
Suppose we want to maintain a dynamic set S of **intervals** which given a query interval i can find an interval in S which overlaps with i.





Following our guidelines for augmenting a data structure, we first need to choose an underlying data structure.

Note that in order to use a search tree, we need to be able to define a linear ordering of the elements. How can we define an ordering of a set of intervals? In particular, if one interval is contained within another interval, how do we decide which interval comes first in our ordering?



We can sort the intervals by increasing left endpoint. This allows us to use a search tree such as a red-black tree.

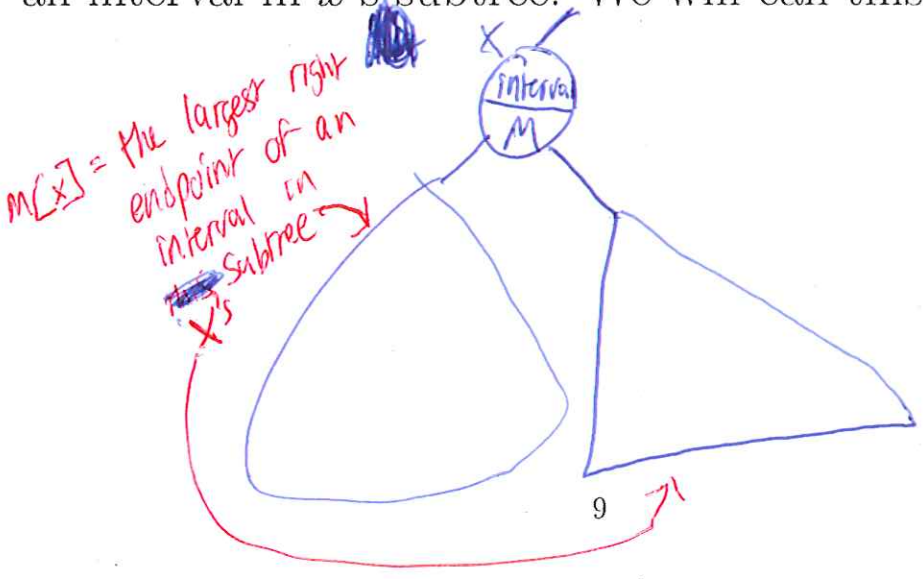
In example: red < green

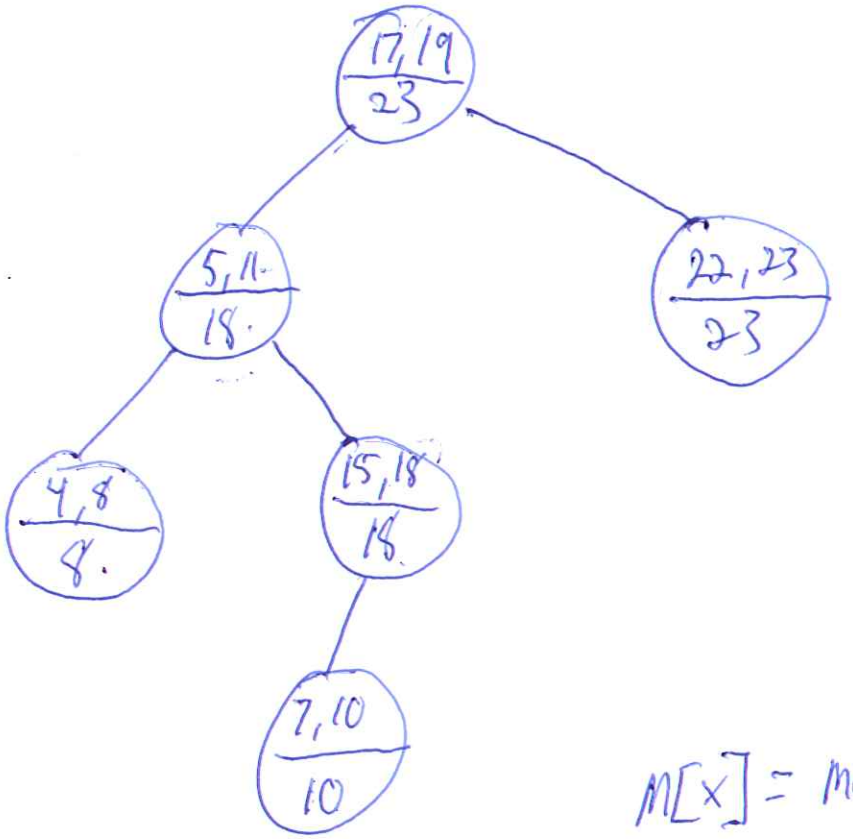
The next step is to determine what additional information needs to be stored in the data structure.

Suppose our query interval is $[17, 20]$ and the root of the tree is $[12, 15]$.

1. There might be an interval $[10, 18]$ in the left subtree of the root.
2. There might be an interval $[13, 19]$ in the right subtree of the root.
3. Which subtree should we check? Without more information we would need to check both.

Idea: store in each node x the largest right endpoint of an interval in x's subtree. We will call this value $m[x]$.

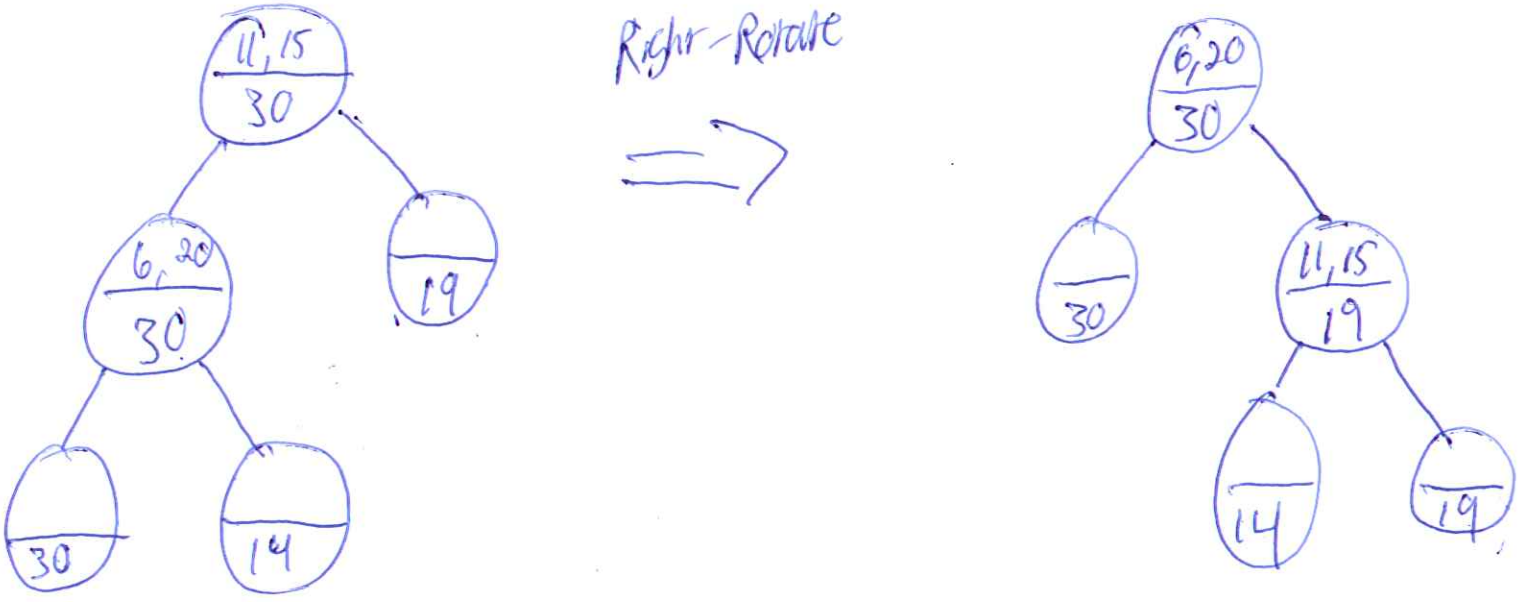




$$m[x] = \max \begin{cases} high[int[x]] \\ m[left[x]] \\ m[right[x]] \end{cases}$$

Now we need to verify that we can maintain these values when inserting and deleting.

1. On insert, we can update m values as we traverse down the tree similarly as we did in the order statistics example.
2. Swapping colors does not change m values. How about rotations?



Finally we need to develop the new operation we wished to implement. In this case, given a query interval i return an interval in the tree which overlaps i.

```
Interval-Search(i)
    x ← root
    while x ≠ NIL and (low[i] > high[int[x]] or
                       low[int[x]] > high[i])
    {
        if left[x] ≠ NIL and low[i] ≤ m[left[x]]
            then x ← left[x]
            else x ← right[x]
    }
    return x
```

Running time is clearly $O(\log n)$.

Proof of correctness is not trivial. We need to show that if we do not find an interval that overlaps i in the path we take then there is no interval in the entire tree which overlaps i. We will prove the following:

Theorem

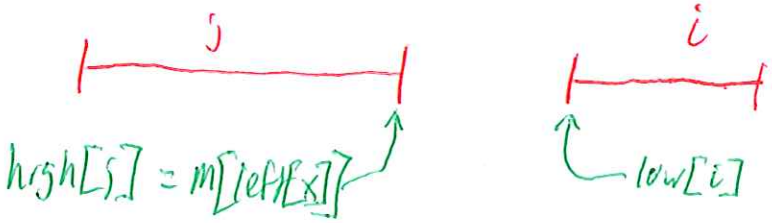
Let L be the set of intervals in the left subtree of node x, and let R be the intervals in x's right subtree.

- If the search algorithm moves to the right child of x, then $\{i' \in L : i' \text{ overlaps } i\} = \emptyset$.
- If the algorithm moves to the left child of x, then $\{i' \in L : i' \text{ overlaps } i\} = \emptyset \Rightarrow \{i' \in R : i' \text{ overlaps } i\} = \emptyset$.

This implies that if we reach a leaf and have not found an interval which overlaps i, then there is no interval in the tree which overlaps i.

Proof

First suppose the search goes right. If $L = \emptyset$ then the theorem clearly holds. If $L \neq \emptyset$ then we have $low[i] > M[left[x]]$. The value $M[left[x]]$ corresponds to the right endpoint of some $j \in L$, and no other interval in L can have its right endpoint to the right of $high[j]$.



This implies no interval in L can overlap i, and the theorem holds.

Now suppose the search goes left and we have $\{i' \in L : i' \text{ overlaps } i\} = \emptyset$. We need to show this implies $\{i' \in R : i' \text{ overlaps } i\} = \emptyset$.

Since we moved left, we have $low[i] \leq m[left[x]] = high[j]$ for some $j \in L$.

By our assumption, we know j does not overlap i, and therefore it must be that $high[i] < low[j]$.

Since our red-black tree sorts the intervals by left endpoint, it must be for every $i' \in R$, we have $low[j] \leq low[i']$ which implies i' does not overlap i. This implies $\{i' \in R : i' \text{ overlaps } i\} = \emptyset$, completing the proof of the theorem.