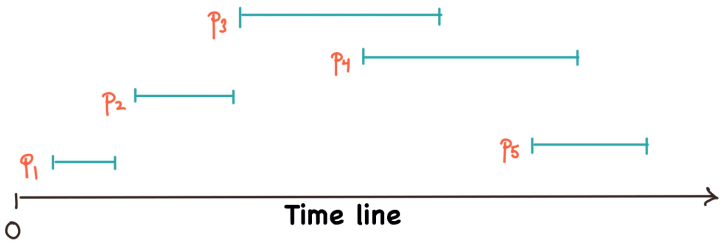
Interval Trees

1 Interval Trees

In this section we look at another example of an augmented binary search tree. Consider the problem of storing a set of *intervals*. As a particular example, suppose each interval represents the start and end times of a project. Often we draw a pictorial representation of the intervals over a time line, as as in the figure below. If each line segment represents the start to end time of a project, then this figure enables us to identify the overlapping intervals. It is clear that $p3$ and $p4$ overlap, as do $p4$ and $p5$. However $p1$ and $p3$ do not overlap.



A manager might want to store all these timelines in a system and be able easily insert and delete the timelines of new projects. It might also be extremely relevant to be able to search for any interval in the system that currently *overlaps* with a given interval. This would be the case for instance, if a new project were proposed and the manager needed to determine if the new timeline overlapped with any others currently in the system.

In this section we look at how to create an augmented binary search tree that stores these intervals and allows for the following operations:

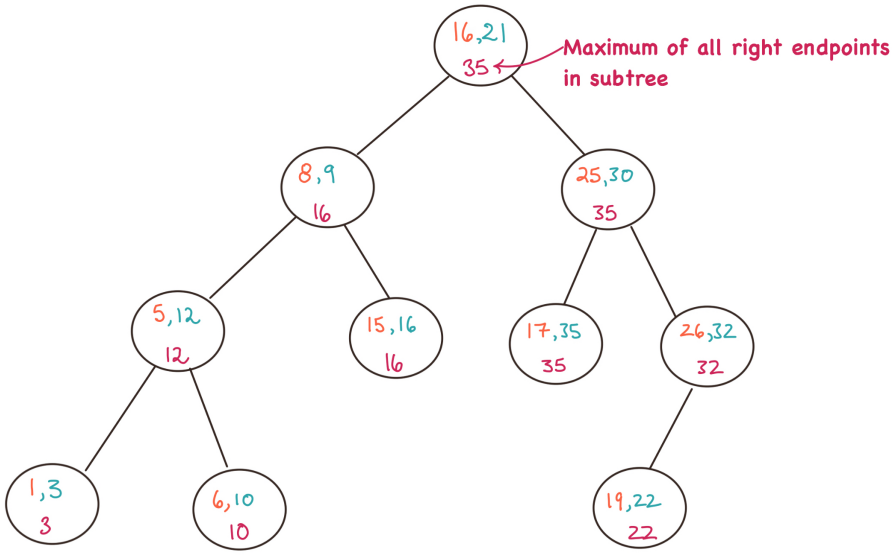
- Insert a new interval into the tree
- Delete an interval from the tree
- Search for an interval in the tree that overlaps with a given interval i

1.1 The tree structure

The intervals are stored in a binary search tree. Each node of the tree is an object which contains the usual BST attributes ($x.parent$, $x.left$, $x.right$), as well as the following additional attributes:

- The left endpoint of the interval: $x.int.low$
- The right endpoint of the interval: $x.int.high$
- The maximum value of all endpoints in its subtree: $x.max$.

Note that there is not attribute $x.key$. Instead, the BST is built using the value in $x.int.low$ as the key. An example of a BST built using the value $x.int.low$ as the key is shown below. Notice that the left endpoints are those that are satisfy the BST property at each node. The right endpoints are also stored at each node, but they are not used as part of the BST property. Each node also contains $x.max$, which is the **maximum** of all right endpoints in the subtree rooted at x:



It remains to show how this new information can be used to search for interval overlaps.

2 Searching for an interval that overlaps with i

The interval search algorithm in this section is a very basic search procedure. We refer to the algorithm as Interval-Search(x,i), with the following specification:

- Input parameter x is the root of an interval tree
- Input parameter i in an interval from $i.low$ to $i.high$.
- The algorithm returns *false* if no interval in tree overlaps with i
- If there is **at least one** interval in the tree that overlaps with i, the algorithm returns a pointer to **one of those intervals**.

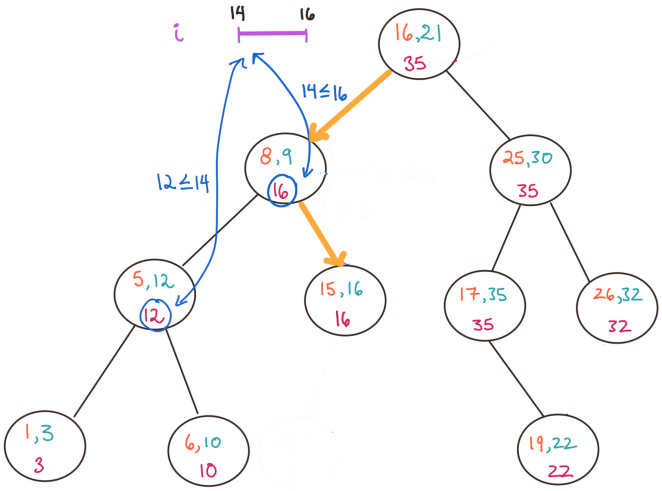
The algorithm works much like the traditional tree-search algorithm. We begin by checking if the interval i overlaps with the root, x, and if it does, it returns the interval at the root. Otherwise it continues the search either down the left path or the right path. The decision to continue either left or right is summarized below:

Which way to search?

Search left If $i.low \leq x.left.maximum$, then if an overlap exists in this tree, one such overlap must exist in the left subtree. So the search continues to the left.

Search right If there is no left subtree, or if $i.low > x.left.maximum$ then the only possible overlap is in the right subtree. So the search continues to the right.

The example below demonstrates the first step of the search for interval $i = (14, 16)$. The maximum right endpoint of the left subtree is 16 and $i.low$ is 14. Therefore one overlapping interval must exist in the left subtree, so the search proceeds to the left. At the second step, the maximum endpoint of the left subtree is only 12 and $i.low$ is 14. Therefore the search continues to the right. At this point, the interval i overlaps directly with the interval $15 \leq x \leq 16$, and therefore we return interval $(15, 16)$ as our result.



The algorithm is shown below, where x is initialized as the root of the tree.

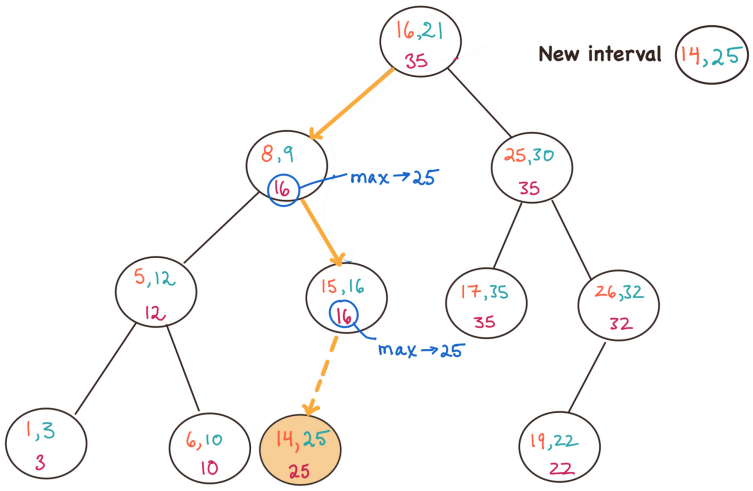
```
INTERVAL-SEARCH(x,i)
    While x ≠ NIL and i does not overlap with x.interval
      if x.left ≠ NIL and x.left.max ≥ i.low
        x = x.left
      else x = x.right
    return x
```

Runtime The algorithm above searches from the root to a leaf in the worst case. Therefore the runtime is $O(h)$, and if the tree is a red-black or AVL tree, then the runtime is $O(\log n)$.

3 Inserts and Deletes

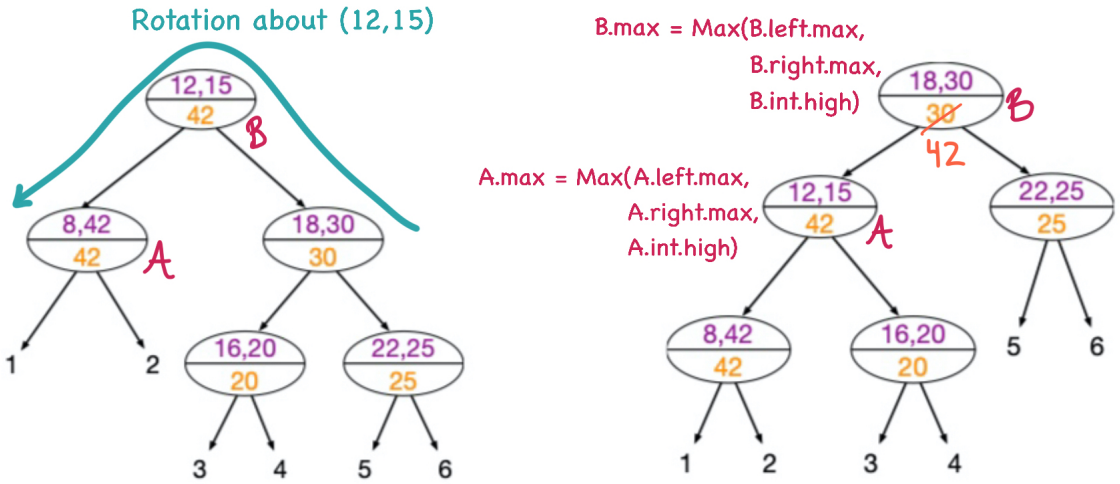
This new interval tree is simply a binary search tree using $x.int.low$ as the key. It is important to show that the additional information we have stored in the tree (the maximum right endpoint) can be quickly updated during an insert and delete operation.

When a new key is inserted, we perform the insert as usual in the binary search tree and insert the new key as a leaf. As we go down the path from the root to the leaf, we update the maximum right endpoint of all the nodes as necessary:



This update does only takes constant time per node, and thus the runtime of the insertion of this node is $O(h)$ as it is for usual BST trees.

However, if the BST is in particular a red-black tree or AVL tree, then the insert may make a call to **RB-repair** which requires rotations. So just like we saw for subtree sizes, we need to update the values of $x.max$ after a rotation is made. There are only **two nodes** whose values need to be updated after the rotation. In the figure below, these nodes are marked A, and B. We simply re-evaluate their max endpoint values after the rotation. This takes a *constant* amount of time per rotation, and so the overall runtime of **RB-repair** is still $O(h)$.



In conclusion then, it is possible to carry out the insert operation for an interval tree in time $O(h)$. Furthermore, if the tree is also a red-black tree, it is possible to update the values of $x.max$ after the rotation so that the overall runtime of *RB-repair* is still $O(h) = O(\log n)$.

3.1 Deletion

The delete operation for interval trees works by carrying out the usual deletion as we saw for BST. Recall that there were 3 cases. In each of these 3 cases, we must update the values of $x.max$ in the interval tree after the deletion. These 3 updates are part of the practice problems.