CS61B Lecture 21

Wednesday, March 11, 2020

Announcements

- CS61B is now fully online due to the suspension of classes.
- Several deadlines have been pushed back.
- The midterm will be online.

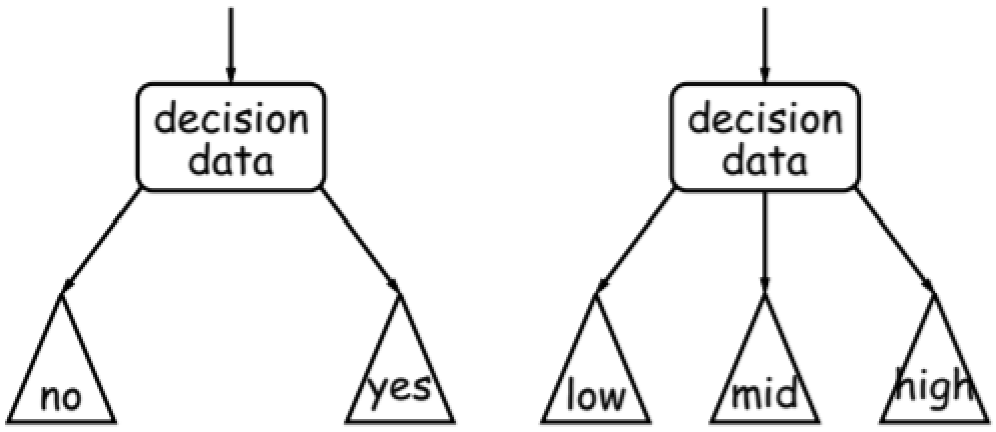
Divide and Conquer

Trees represent hierarchies of data and they can mean one of many different things. For the purposes of searching, a very useful thing is to have the hierarchy be something that models what we call "divide and conquer".

It is a way of recursively breaking the problem down into small pieces and attacking the pieces separately, which if done correctly, can vastly speed up our algorithms.

We'll say that a node either contains data or some information that tells us which child might contain the output.

We are going to store some enough information to decide which child might have the data we are looking for. We saw the figure for logarithmic algorithms: at one microsecond per comparison, we could process 10 to the power of 300000 items per second. The tree is a natural framework for this representation.



Binary Search Trees

We have seen in some small capacity before, the Binary Search Tree (BST). Tree nodes contain keys and possibly other data. The binary search property states that all nodes in the left subtree of the node have smaller keys, and all nodes in the right subtree have larger keys.

"Smaller" means any complete transitive, anti-symmetric ordering on keys. We are going to play fast and loose with the definition of smaller, which can satisfy one of these conditions:

- Exactly one of `x < y` or `y < x` is true
- `x < y` and `y < z` imply `x < z`

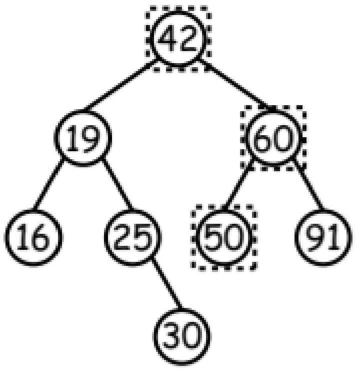
- Duplicate keys can mean one of several different things, but for the sake of simplicity, we will not deal with duplicate keys this semester.

For example, in a dictionary database, the node label wouild be `(word, definition)`, where `word` is the key.

For concreteness, we will use the standard Java convention of calling `compareTo`, or the `<`, `>` operators.

Finding

Here's an example of what searching for the labels 50 and 49 would look like in a BST:

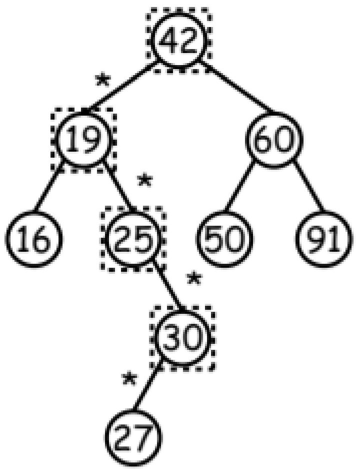


```
static BST find(BST T, Key L) {
    if (T == null) { return T; }
    if (L.compareTo(T.label()) == 0 ) {
        return T;
    } else if (L.compareTo(T.label()) < 0) {
        return find(T.left(), L);
    } else {
        return find(T.right(), L);
    }
}
```

Dashed boxes show which node labels we look at, and the nodes looked at is directly proportional to the height of the tree.

Inserting

And here's how we would insert into a BST:



```
/** Insert L in T, replacing existing
* value if present, and returning
* new tree. */
static BST insert(BST T, Key L) {
    if (T == null) { return new BST(L); }
    if (L.compareTo(T.label()) == 0) {
        T.setLabel(L);
    } else if (L.compareTo(T.label()) < 0) {
        T.setLeft(insert(T.left(), L));
    } else {
        T.setRight(insert(T.right(), L));
    }
    return T;
}
```

Starred edges are set (to themselves, unless initially null). And the time complexity remains proportional to the height of the tree.

Deletion

Deletion can be rather a pain in the neck, as we will see with many of these data structures throughout the course. There are three possible cases:

Removal of childless node

In this case, it would be fairly easy, as we would simply reassign the parent node's pointers to no longer point at the deleted node. Our deletion algorithm would try to see what, in the example above, the children of 30 would look like without the node 27, and return that. Because 27 is a leaf, then it would just return null, and 30's children will now be null.

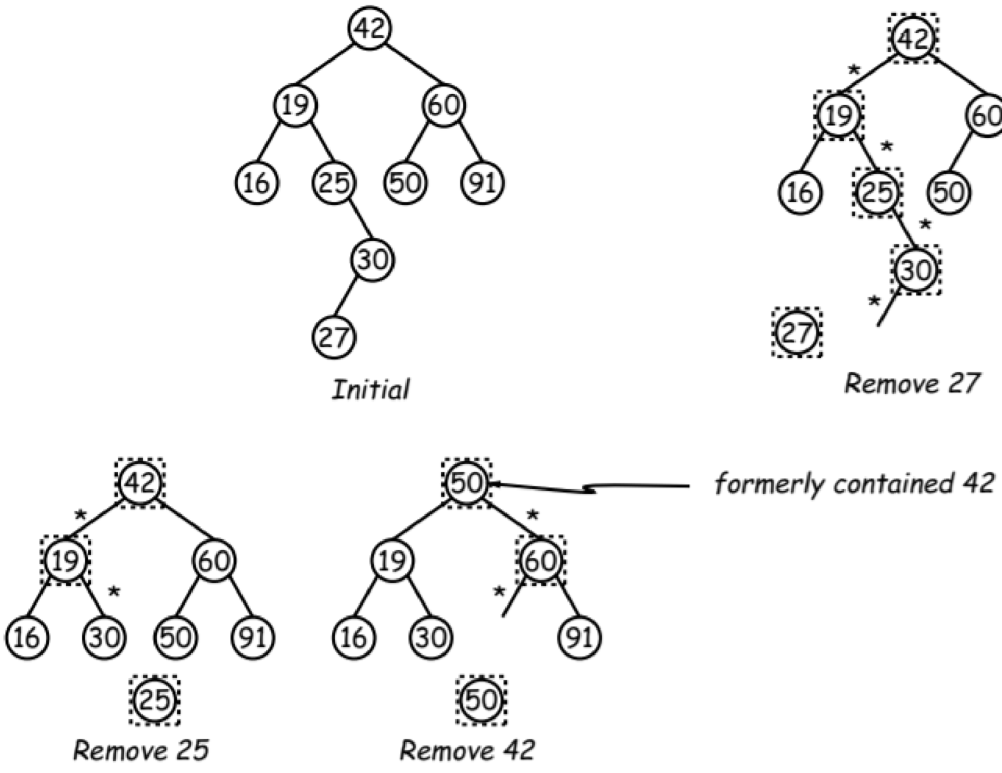
Removal of node with one child

In this case, it would also be a fairly simple procedure of moving the children "up one level", as the one child inherits from the parent and replaces the parent with itself. Our deletion algorithm would find that the children of 19 without the node 25 is just the tree with the single node 30, and its right child is now that tree.

Removal of node with two children

We've now taken care of the above two cases. This last case is a little complicated. We must look down into the children and find a node that replaces 42 without disturbing the tree's binary search property. Obviously, the danger is that we will disturb this property and end up with, for example, nodes on the right that are less than the node we choose to replace 42 with.

How do we avoid that? Well, we find the smallest node, and move that up to the top after removing it from the subtree! The binary search property assures us that this smallest node from the right subtree will still be bigger than any node in the left subtree.



```java
/** Remove L from T, returning new tree. */
static BST remove(BST T, Key L) {
    if (T == null) { return null; }
    if (L.compareTo(T.label()) == 0) {
        if (T.left() == null) { return T.right(); }
        else if (T.right() == null) { return T.left(); }
        else {
            Key smallest = minVal(T.right()); // ??
            T.setRight(remove(T.right(), smallest));
            T.setLabel(smallest);
        }
    } else if (L.compareTo(T.label()) < 0) {
        T.setLeft(remove(T.left(), L));
    } else {
        T.setRight(remove(T.right(), L));
    }
    return T;
}
```

Finding minimum and maximum nodes

> To find the minimum node in any tree, keep going down the left branch until it is null. Conversely, finding the maximum node in any tree involves going down the right side continuously until we get to null. This is how the `minVal` method in the code above works, which still has a slight `TypeError`. Try to find the error and fix it in the code above!

These are the basic operations, which are the basic operations defined in the Java library: `contains`, `add` and `remove`, which we have defined here.

> **The performance problem with BSTs**
>
> If we were to delete enough (and the right) nodes from our BST, we will find that we will have converted our BST into a linked list, which means that in the worst cases, the time of insertion and deletion will be proportional to the size of the linked list. It is a performance problem we will come back to solve later.

Quadtrees

Let's say we want to index information about 2D locations so that items can be retrieved by position. What we find is that binary search trees *don't have to be binary*. Quadtrees use the standard data-structuring trick of divide and conquer.

The idea is to divide 2D space into four quadrants, and store items in the appropriate quadrant. Repeat this recursively with each quadrant that contains more than one item.

Original definition -- a quadtree is either:

- empty,
- an item at some point, `(x, y)`, called the root, plus
- four quadtrees, each containing only items that are northwestn, northeast, southwest and southeast of `(x, y)`

The big idea is that if you are looking for some point `(x', y')`, and the root is not the point you are looking for, you can narrow down which of the four subtrees of the root to look in by comparing the coordinates of the root with the coordinates of the point you are looking for.

There is more information than we need to know regarding quadtrees in this version of the course, because this data structure was previously a required part of a project. This project is not being done this year, but perhaps you will still find it useful.

Example

We start at some root node, which is point A in this case. If we want to find the point D. The point D is clearly northeast of A, so we go to the northeast child, which is B. From B, the point D is to its southwest, so we go to the southwest child C. And from here, the point D is to the southeast, so we go to its southeast child and find D.

In each case, we know the coordinates of the point we are on, and we know the coordinates of the point we are looking for.

A slight variation of the quadtree has us represent it as a binary search tree. The way we do that is that we have every odd level be divided by east-west, and every even level be north-south, which gives us the same effect.

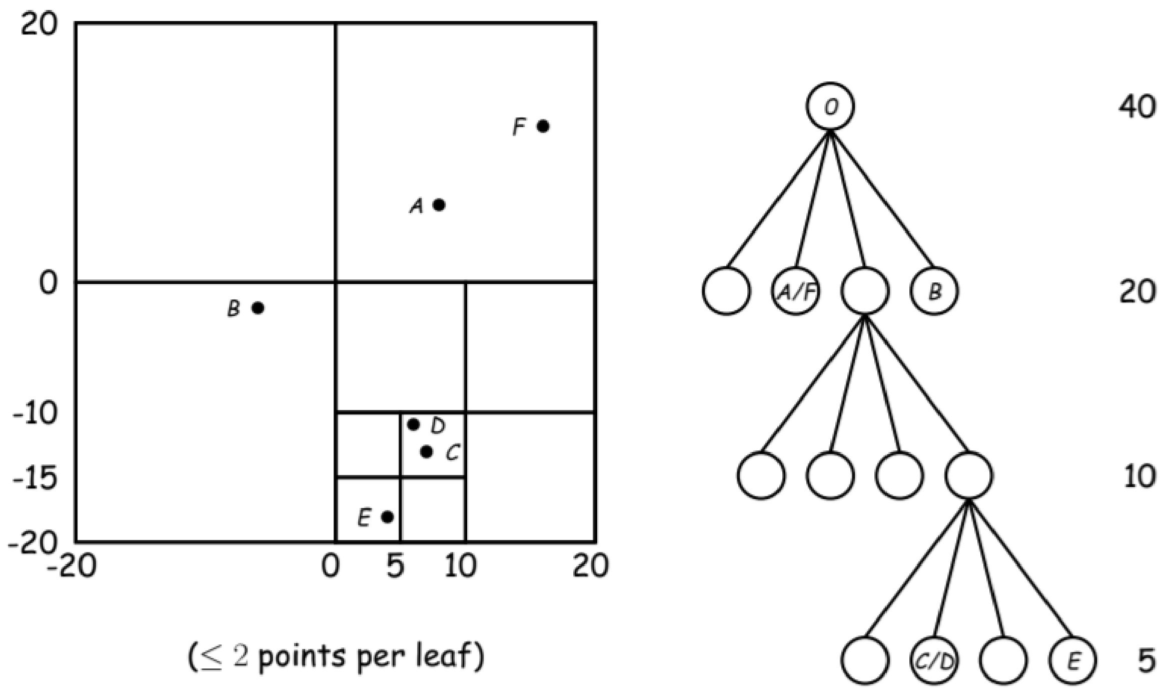
Point-region (PR) Quadtrees

If we use a Quadtree to track moving objects, it may be useful to delete items from a tree: when an object moves, the subtree that it goes in may change. It is diffcult to do this with our regular quadtree above, so we'll define a **bounding rectangle** B as part of the definition of a quadtree, and either:

- Zero or up to a small number of items that lie in that rectangle, or
- Four quadtrees whose bounding rectangles are the four quadrants of B (all of equal size).

A completely empty quadtree can have an arbitrary bounding rectangle, or you can wait for the first point to be inserted.

Example



(≤ 2 points per leaf)

Navigating PR Quadtrees

To find an item at `(x, y)` in quadtree T:

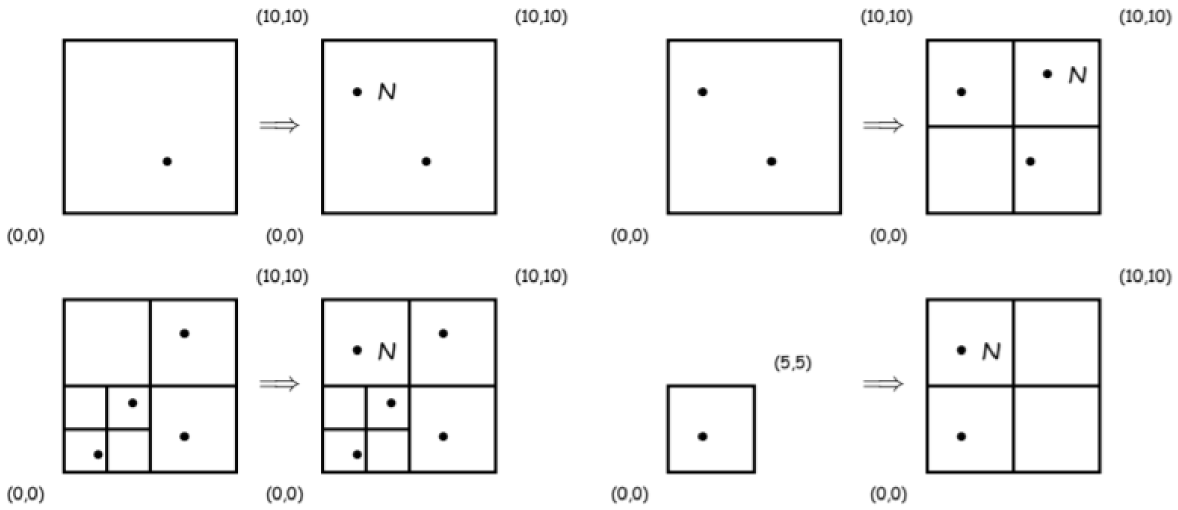
1. If `(x, y)` is outside the bounding rectangle of T, or T is empty, then `(x, y)` is not in T.
2. Otherwise, if T contains a small set of items, then `(x, y)` is in T if and only if it is among these items.
3. Otherwise, T contains of four quadtrees. Recursively look for `(x, y)` in each (jowever, step 1 above will cause all but one of these bounding boxes to reject the point immediately).

Similar procedure when looking for all items within some rectangle R:

1. If R does not intersect the bounding rectangle of T, or T is empty, then there are no items in R.
2. Otherwise, if T contains a set of items, return those that are in R, if any.
3. Otherwise, T consists of four quadtrees. Recursively look for points in R in each one of them.

Inserting into PR Quadtrees

There are various cases for inserting a new point N, assuming maximum occupancy of a region is 2, showing initial state =⇒ final state


(10,10)
(0,0)
N
(5,5)