CMSC 420: Lecture X01
Supplemental: Extended Search Trees

Extended Search Trees: In this lecture, we will discuss how to implement a standard binary search tree as an extended tree. Recall that an extended binary tree has two different node types, *internal* and *external*. Every internal node has exactly two children and external nodes have no children. Extended trees (of various sorts) are often used in the design of search trees. By separating node types, we can better tailor each node to its particular function. Data (that is, the key-value pairs) are stored only in the external nodes. Internal nodes only store keys, called *splitters*, and their purpose is to serve as an index, directing the search to the appropriate external node where the data is stored. In this lecture, we will consider how to do this in the particular context of AA trees.

Extended AA Tree: Recall (from an earlier lecture) that AA trees and red-black trees are binary variants of 2-3 trees. There are two notable features of AA trees compared to red-black trees. First, rather using colors, each node stores a *level*, where leaves are at level 1, and the root is at the highest level. A node is declared to be red if it is at the same level as its parent. Second, rather than using `null` pointers, there is a special sentinel node called `nil`, which resides at level zero. This node is constructed so that `nil.left = nil.right = nil`.

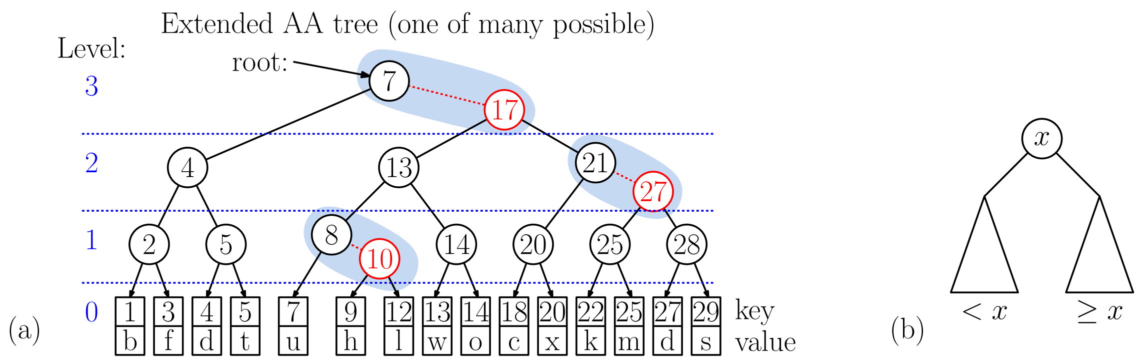

Fig. 1: (a) An extended AA Tree storing the key-value pairs $\{(1, \mathrm{b}), (3, \mathrm{f}), (4, \mathrm{d}), \ldots\}$, (b) ordering convention.

To convert this to an extended tree, we make the following modifications (see Fig. 1(a)):

Internal nodes: Each stores a key (but no value), left and right child pointers, and a level number, which is ≥ 1. Our convention is that if an internal node stores a key x, then the keys in its left subtree are all strictly smaller than x and the keys in the right subtree are greater than or equal to x (see Fig. 1(b)).

External nodes: Each stores just a single key-value pair. There are no child links, and there is no need to explicitly store a level number because the level number must be zero.

What about `nil`? We will not make use of an explicit node `nil`, but we will describe below how we can "fake" its existence.

Java Node Structure: Let's call our extended AA tree the `AAXTree`. It will naturally be templated by two types `Key` and `Value`. We assume that the `Key` type implements the Java `Comparable` interface, meaning that it defines a function `compareTo()` for comparing keys. The most natural way to implement the two node types is to define an inner node class (within `AAXTree`) and use inheritance. There will be an *abstract* parent class, `Node`, and two subclasses, `InternalNode` and `ExternalNode`. Since `Node` is only a placeholder, its methods are all declared to be "`abstract`." Here is a possible outline for the file `AAXTree.java`:

```
public class AAXTree<Key extends Comparable<Key>, Value> {

    private abstract class Node { // generic node (purely abstract)
        abstract Value find(Key x);
        // ... other helper functions omitted
    }

    private class InternalNode extends Node { // internal node
        Key key;
        Node left, right;
        int level;

        Value find(Key x) { /* ... */ }
        // ... other helper functions omitted
    }

    private class ExternalNode extends Node { // external node
        Key key;
        Value value;

        Value find(Key x) { /* ... */ }
        // ... other helper functions omitted
    }

    // ... the rest of the class
}
```

The helper functions will naturally be members of the node classes, as opposed to static functions as we described in lecture. Functions are called using a different syntax. Rather than "`p.left = insert(x, v, p.left)`", the node `p` would invoke its own function using "`left = left.insert(x, v)`". This will invoke the appropriate helper (internal or external) depending on the type of `p.left`.

Extended Search Tree Operations: Let us explain how to perform the standard dictionary operations on an extended tree.

`Value find(Key x)`: Determines whether there is a key-value pair (x, v), and if so returns a reference to v. Otherwise, it returns `null`.

Top level: If the root is `null`, then return `null`. Otherwise, invoke the find helper function on the root node.

Internal: Apply the find recursively on the appropriate child and return the result. For example, if `x < key`, return `left.find(x)`. Otherwise, apply the process symmetrically to the right child. (Note that if `x == key` we *cannot* infer that the key is in the dictionary.)

External: If `x` is equal to this node's `key`, we return a reference to the associate value, and otherwise we return `null`.

Example: In Fig. 2(a) we show the result of a successful search, and in Fig. 2(b), we show an unsuccessful search. Note that all searches descend to the leaf level before returning.

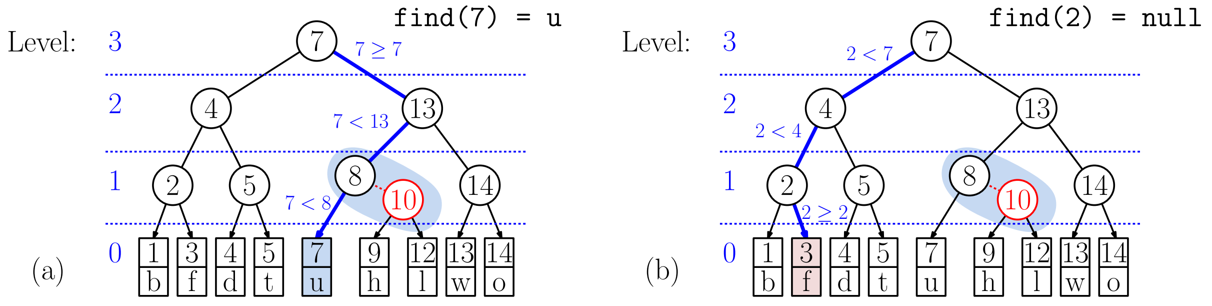

Fig. 2: (a) `find(7)` descends to external node $(7, u)$ and returns "`u`" and (b) `find(2)` descends to external node $(3, f)$ returns `null`.

`void insert(Key x, Value v)`: Inserts key value (x, v), throwing an exception if there is a key-value pair in the dictionary with key x.

Top level: If the root is `null` (meaning that the tree is empty), create a single external node containing the pair (x, v) and set the root to point to this node. Otherwise, invoke the insert helper function on the root.

Internal: Apply the insert helper function recursively on the appropriate child, and update the child link. For example, if `x < key`, then do `left = left.insert(x,v)`. Otherwise, apply the insertion symmetrically on the right child. (Note that there is no error if `x == key`, since we don't know whether x is in the tree until we arrive at an external node.)
As in the standard AA-tree code, just before returning, invoke skew and split and return their result. (As mentioned above, the function call syntax differs from the standard AA tree. For example, rather than using "`return split(skew(p))`", node `p` invokes "`return skew().split()`".)

External: Let (y, w) denote this node's contents. If $x = y$, we throw an `Exception` with the error message `"Insertion of duplicate key"`. Otherwise, we create a new external node containing the pair (x, v) and link it into the tree.

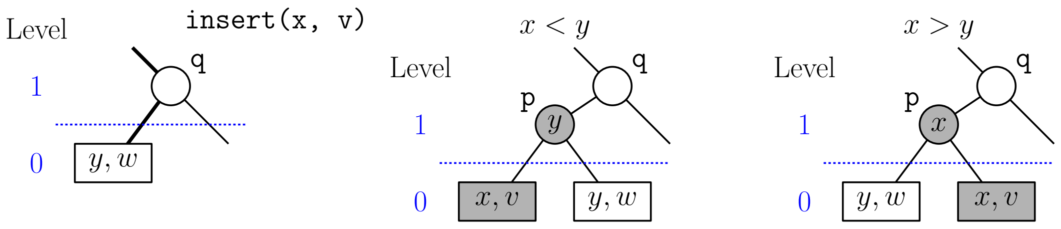

Fig. 3: Insertion of the new external node.

There are two cases for how to do this. If $x < y$, create a new internal node `p` at level 1 and set its splitter key to y. We make (x, v) its left child and make (y, w) its right child (see Fig. 1 middle). On the other hand, if $x > y$, create a new internal node at level 1 and set its splitter key to x. Make (y, w) its left child and make (x, v) its right child (see Fig. 1 right). In either case, we return a pointer to the newly created internal node. (There is no need to invoke skew or split on the the new external node nor on `p`.)

How do we modify the child link of the parent `q`? This happens automatically, because the internal-node insertion function always updates the child link with the return result.

Example: As an example, consider the insertion of $(11, e)$ in Fig. 4. We first locate the external node $(12, l)$ at which to begin. Because $11 < 12$, we create the external node $(11, e)$ at level 0 and the internal node storing the key $y = 12$ at level 1. This new internal node becomes the new right child of 10 (upper right in Fig. 4).

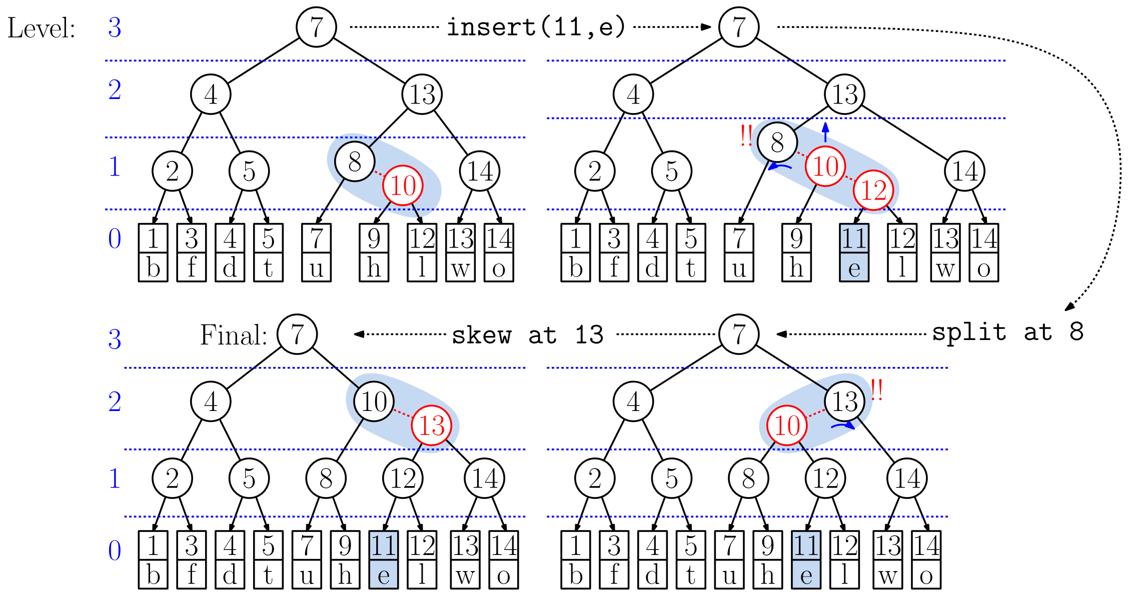

Fig. 4: Insertion of $(11, e)$.

We then return back up the search path starting from 10 performing skews and splits at each node. The first place where action is needed is node 8, since its right-right grandchild is at the same level. We split here by performing a left rotation at 8 and promote its right child 10 to the next higher level. This results in 10 moving to level 2 where it becomes the left child of 13. When we return to 13, we apply skew, which performs a right rotation at 13. This makes 10 the current node, becoming 7's new right child. No further changes are made, and we eventually return to the root and are done (lower left in Fig. 4).

`void delete(Key x)`: Deletes the entry with key x, throwing an exception if there is no key-value pair in the dictionary with key x.

Top level: If the root is `null` (meaning that the tree is empty), we throw an `Exception` with the error message `"Deletion of nonexistent key"`. Otherwise, invoke the delete helper function on the root.

External: We'll discuss the external node case, since it affects how internal nodes are processed. If the key stored in this node does not match x, we throw a non-existent key exception, as above. Otherwise, we will remove both this node and its parent from the tree, replacing it with the parent's other child `q` (see Fig. 5). Since we do not have parent links, we will do a little dance with our parent node `p`. To signal the dance, we return `null`. (For example, in Fig. 5 the call `r.delete(x)` returns `null`.)

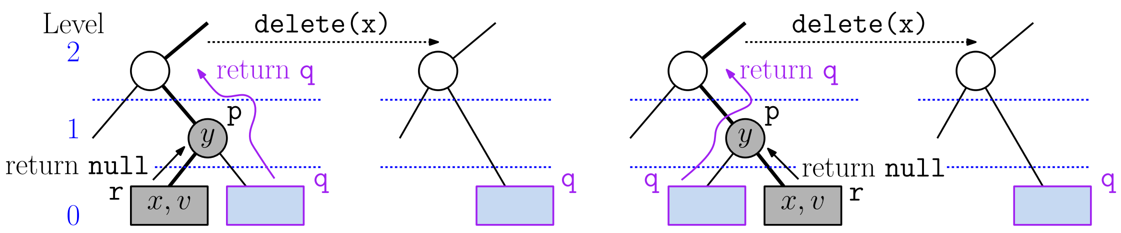

Fig. 5: Deletion at the external node level.

Internal: We apply the delete helper function recursively on the appropriate child (left side if `x < key` and right side otherwise) and save the return result. If the return result is `null`, this is the signal that we just deleted an external node. In this case, we return a reference to our other child. (For example, in Fig. 5 the call `p.delete(x)` returns `q`.)
If the return result is not `null`, we handle this the same as the standard AA tree. We set our child pointer to the return result. (For example, in Fig. 5, `p`'s parent sets its right child to `q`.) Also, just before returning, we invoke `fixAfterDelete` to rebalance the tree, and we return its result.

Example: Let's consider the deletion of 4 in Fig. 6. We make recursive calls until arriving at the external node $(4, d)$ containing the key to be deleted. This returns `null` to internal node 5. On seeing this `null` value, internal node 5 returns a reference to its other child $(5, t)$ to internal node 4.

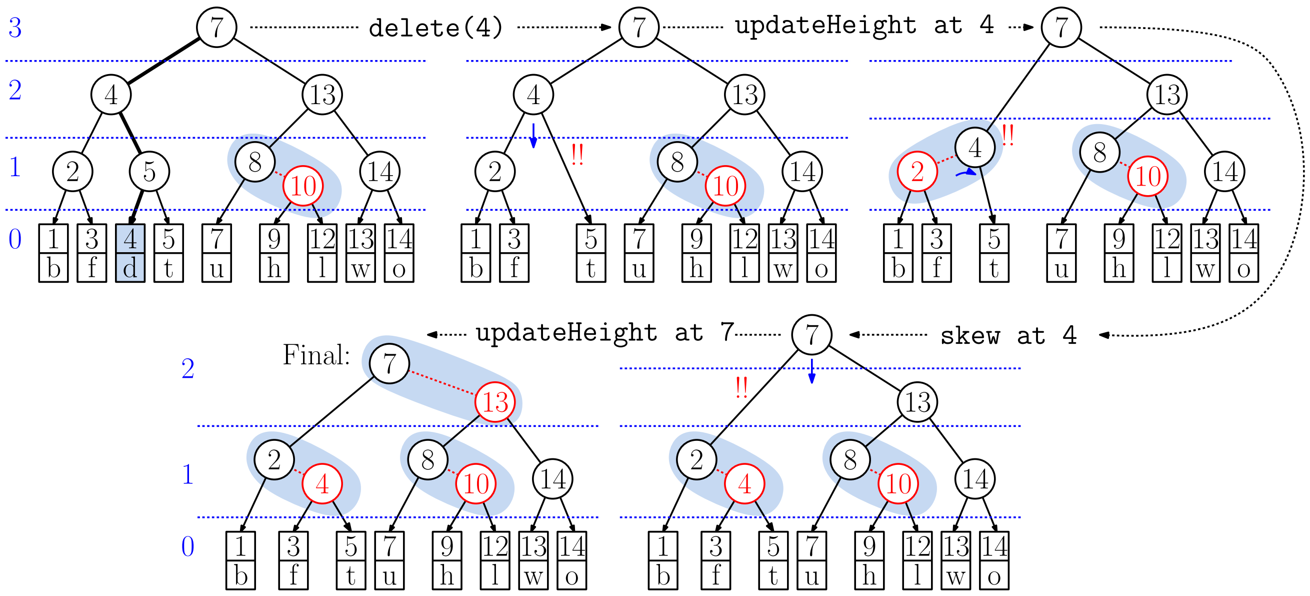

Fig. 6: Deletion of 4.

On returning to the internal node 4, we see that the return value is non-null, so we update our right child link to point to this return value, $(5, t)$. We also invoke `fixAfterDelete`. We see that our right child is two levels lower, so `updateHeight` decreases node 4's level to 1 (upper-right in Fig. 6). The subsequent rebalancing operations resulting in a right rotation (skew) at 4. This returns node 2, which becomes 7's new left child. On returning to node 7, it sees that its child node 2 is two levels lower, and so `updateHeight` moves node 7 down to level 2. At this point, the tree's structure is correct, and we are done.

Find smaller/larger: To see whether you understand the general pattern, think about how you would implement two other queries, `findSmaller(x)` and `findLarger(x)`. Given a key `x`, these return the value associated with the key that is strictly smaller and strictly larger than `x`, respectively (or `null` if there is not such key).

These operations can be implemented to run in $O(\log n)$ time by performing two searches in the tree. In Fig. 7 we illustrate these two search paths for both `findLarger` and `findSmaller`. Let's consider `findLarger`. The first (primary) search employs just a standard find operation for the key `x` itself. Suppose that `x` is found in some external node u (`(14,o)` in the figure). Then we need to backtrack up the search path the an appropriate node (17) from which to launch a secondary search to find u's inorder successor (`(18,c)` in the figure).

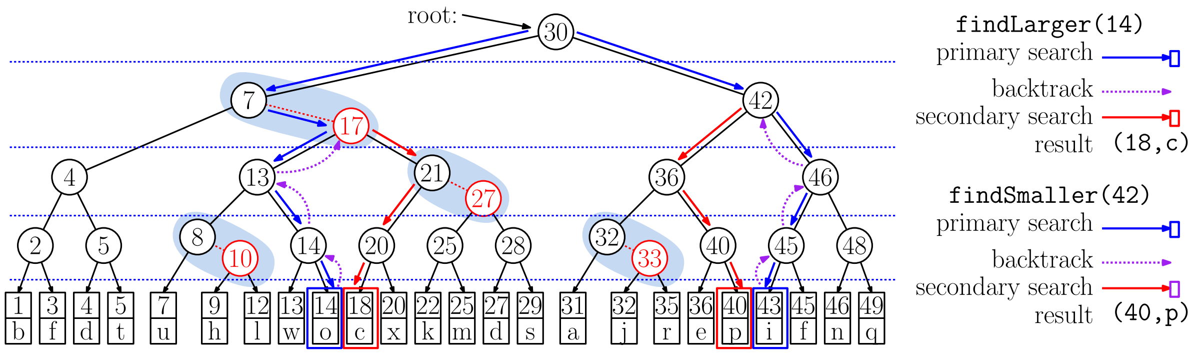

Fig. 7: The operations `findLarger` and `findSmaller`.

We will leave as an exercise the question of how to determine where the backtracking stops and the secondary search begins. (As a hint, it is noteworthy that the backtracking and secondary searches are determined purely by the tree's structure. It is not necessary to compare keys.)

The backtracking process would be easy if we had additional information, such as threads or parent links, but we have neither of these. However, all the ancestors of u are implicitly stored in the system's recursion stack. While you could solve this problem by explicitly maintaining your own stack to do this, there is a more direct solution which we will let you think about. Either approach will lead to $O(\log n)$ running time, since you are simulating two searches, each taking $O(\log n)$ time.

Note that if `x` is not found in the node from the primary search, backtracking may not be needed. For example, in the case of `findSmaller(44)`, there is no need to backtrack since the desired result is in the external node `(43,i)`.

Faking Nil: Recall that traditional AA trees use the sentinel node `nil` to avoid the need many if-statements checking for `null` children. This approach will not work for extended trees, but there is a simple and elegant way to "fake" having `nil`.

We can define accessor methods for our node classes, which will cause external nodes to behave the same manner as `nil`.

Getters: We define node methods `getLeft()`, `getRight()`, and `getLevel()`. For internal nodes, these will return the actual left, right, and level number for the node, respectively. For external nodes, `getLeft()` and `getRight()` both return `this`, and `getLevel()` returns zero.

Setters: We define setter functions `setLeft(Node v)` and `setRight(Node v)`. When applied to internal nodes, these set the left and right child links to `v`, respectively. In the case of external nodes, these do nothing.[1] We also define a function `setLevel(int l)`. For internal nodes this sets the level, and for external nodes it does nothing.

Rebalancing utilities: Finally, we will need to define both internal and external versions of the rebalancing functions, `skew()`, `split()`, `updateHeight()`, and `fixAfterDelete()`. For internal nodes, these will have the expected behavior, and on external nodes they will do nothing.[2]

With the aid of these utility functions, the original AA rebalancing code can be adapted with only minimal changes. There is no need to perform any type casting nor to employ Java's `instanceof` function. Again, function calling method changes. instead of "`p.right = split(p.right)`", we would use "`p.setRight(p.getRight().split())`".

[1]While it is hard to believe, there are actually situations where the AA tree code attempts to change the value of `nil`'s children, but whenever this happens it sets the child to `nil`, so the operation can be safely ignored.

[2]Why should we define functions that do nothing? This is mostly to make the compiler happy. Even though we know logically that these functions will never be invoked on external nodes, the compiler does not. If we do not define these (trivial) functions, the compiler will complain.