CMSC 420: Lecture 12
Extended and Scapegoat Trees

Overview: Today's lecture will focus on two concepts, extended binary search trees and scapegoat trees. (The material on the SG-Tree, which was discussed in class is only covered in the lecture slides.)

Extended Binary Search Trees: Recall from an earlier lecture the concept of an *extended binary tree*, that is, a binary tree whose nodes have either two children or zero children. The former are called *internal nodes* and the latter are called *external nodes*. An example is shown in Fig. 1(a).

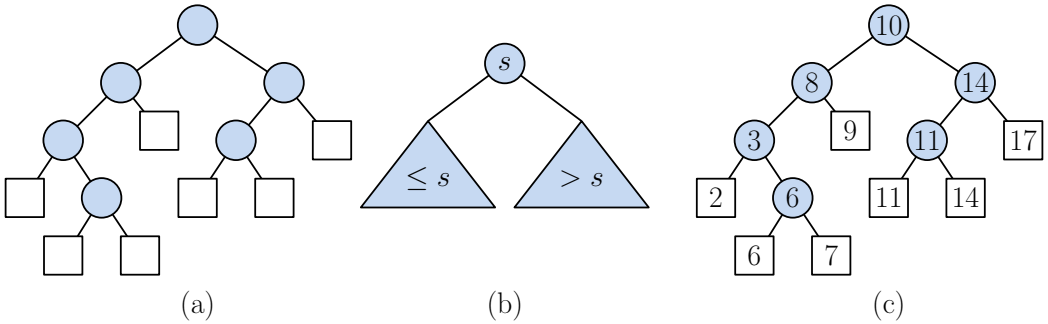

Fig. 1: (a) Extended binary tree, (b) extended binary search tree structure, and (c) extended binary search tree containing the keys $\{2, 6, 7, 9, 11, 14, 17\}$.

As we saw in our discussion of B+ trees in an earlier lecture, it is often useful to employ extended trees in the context of search trees. While B+ trees are multiway trees, we will explore this in the context of binary search trees. The idea is to store all the key-value pairs in just the external nodes. The internal nodes are merely an index structure whose purpose is to allow us to rapidly identify an external node of interest.

More formally, each internal node stores a key s, called a *splitter*, with the property that all external nodes whose key value x is at most s reside in s's left subtree and all external nodes whose key value is strictly greater than s reside within s's right subtree (see Fig. 1(b)). An example of an extended binary search tree is shown in Fig. 1(c).

It is important to note that the tree's *contents* consist stored in the external nodes, not the internal nodes. For example, in Fig. 1(c), the tree's contents are $\{2, 6, 7, 9, 11, 14, 17\}$. The splitter values 3, 8, and 10 appear in internal nodes, but they are not counted among the tree's contents.

Motivation - Multi-dimensional trees: In the context of binary search trees, the advantage of this extended-tree approach is not very obvious. The useful of distinguishing data from splitters becomes more evident when we consider search structures in a multi-dimensional context of *partition trees*.

For example, consider the hierarchical decomposition of space shown in Fig. 2 (left). In this case, the splitters correspond to lines in the plane. Each such line could be expressed as its equation (e.g., $y = ax + b$). The points lying on one side of the line are stored in the left subtree and the points lying on the other are stored in the right subtree. Continuing in this manner, we obtain a data structure called a *binary space partition tree*, or *BSP tree* for short.

The key-value pairs in this case are points and whatever additional data we wish to decorate each point with. In this case, it is easy to see that there is a fundamental difference between splitters (line equations) and data (points). Before exploring extended trees in the context of multi-dimensional space, it will be useful to consider them in the simpler 1-dimensional context, which we are familiar with.

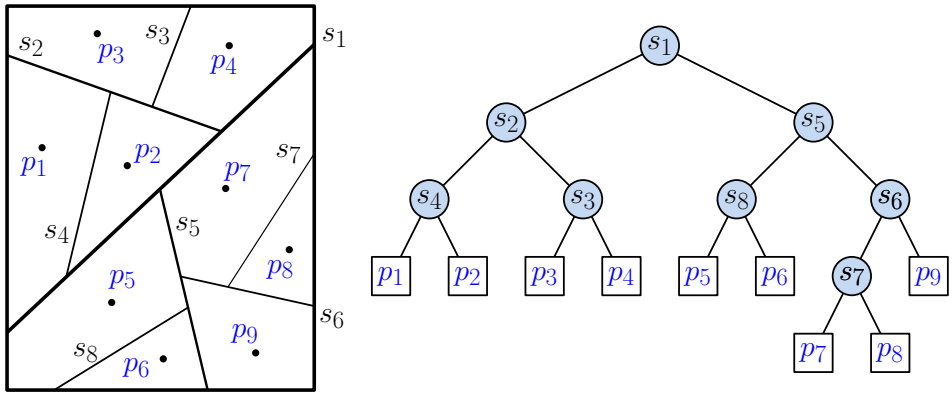

Fig. 2: Binary space partition tree, (a) decomposition of space and (b) the tree structure.

Dictionary Operations on Extended Trees: The dictionary operations that we defined for standard (unbalanced) binary search trees are readily generalized to extended binary search trees.

`find(Key x, Node p)`: The initial call is `find(x, root)`. The procedure operates recursively. If `x` $\leq$ `p.key` we recurse on the left subtree, and otherwise we recurse on the right subtree. On arriving to an external node `p`, we test whether `x` $=$ `p.key`. If so, we report success and otherwise we report failure.

Note that if we encounter an internal node whose key value is equal to `x`, we cannot report success, since the key values in the internal nodes are not reflective of the tree's contents. They are merely an aid to finding the key in the external nodes. For example, on the tree shown in Fig. 1(c), `find(10)` returns false, even though there is an internal node containing `10`. (In this case, the search ends at the external node containing `9`.)

`insert(Key x, Value v, Node p)`: This function returns a reference to the root of the updated subtree where `x` is inserted. The initial call is `root = insert(x, v, root)`.

The procedure operates recursively. We use the same process as in `find` to locate an external node `p`. If there is no such node because the tree is empty, we create a single external node containing `x`, which we return. Otherwise, we check whether `x` $=$ `p.key`, and if so we signal a duplicate-key error. If neither of these happens, we create a new external node containing `x`, and an internal node to split between `x` and `p.key`.

More formally, let $y \leftarrow$ `p.key`. Following our convention that the left subtree contains key values that less than or equal to the splitter and the right subtree is strictly greater, the splitter can be any value s such that $\min(x, y) \leq s < \max(x, y)$. We will assume the simple convention of setting $s = \min(x, y)$. We first create a new internal node containing s and a new external node containing key x and value v. Between the two external nodes x and y, we make the smaller the left child of s and the larger is its right child (see Fig. 3). Finally, we return a reference to the internal node containing s, which is stored in the child link of the parent of `p`. (An example is shown in Fig. 4.)

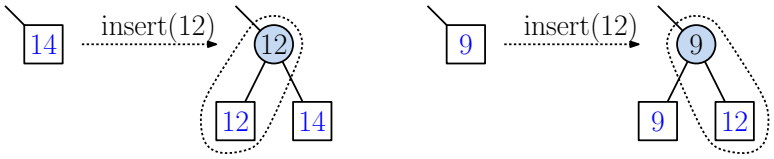

Fig. 3: Inserting a new external node into an extended binary search tree. Note that the internal node is assigned the smaller of the two key values.

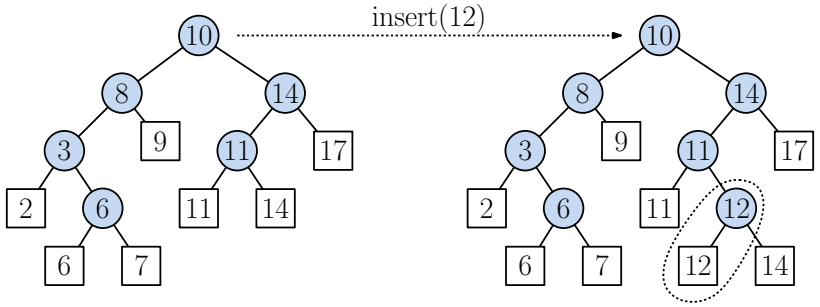

Fig. 4: Inserting a key `12` into an extended binary search tree. A search for `12` leads to the external node `14`. Two nodes are created, one external node containing `12` and one internal node containing the minimum of `12` and `14`.

`delete(Key x, Node p)`: This function returns a reference to the root of the updated subtree from which `x` is deleted. The initial call is `root = delete(x, root)`.

The procedure operates recursively. We use the same process as in `find` to locate the external node `p` that contains `x`. If `x` is not found, we signal an nonexistent-key error. Otherwise, if this external node is the root of the tree, we remove it and return the value `null`. If neither of these occurs, we delete the external node and its internal node parent. We return a reference to the other child of the parent (see Fig. 5).

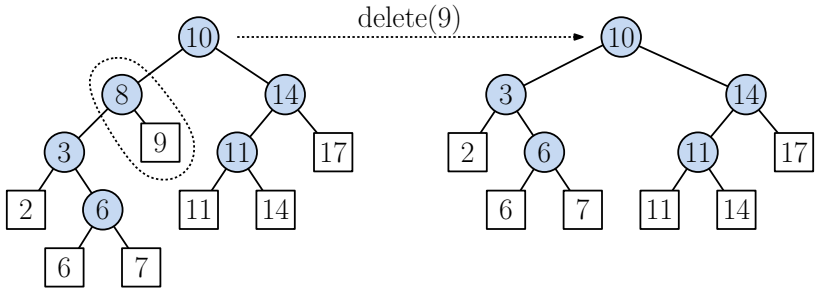

Fig. 5: Deleting a key `9` from an extended binary search tree. After finding the external node containing `9`, we remove it and its parent, and link the other child of the parent into the grandparent.

As with standard (unbalanced) binary search trees, all operations take time proportional to the height of the tree. The height of the tree is (up to a constant additive term) the same in expectation for the extended tree as for the standard tree, namely $O(\log n)$ if n keys are inserted in random order.

Scapegoat Trees: We have previously studied the *splay tree*, a data structure that supports dictionary operations in $O(\log n)$ amortized time. Recall that this means that, over a series of

operations, the average cost per operation is $O(\log n)$, even though the cost of any individual operation can be as high as $O(n)$. We will now study another example of a binary search tree that has good amortized efficiency, called a *scapegoat tree*. The idea underlying the scapegoat tree was due initially to Arne Anderson (of AA trees) in 1989. This idea was rediscovered by Galperin and Rivest in 1993, who made some refinements and gave it the name "scapegoat tree" (which we will explain below).

While amortized data structures often interesting in their own right, there is a particular motivation for studying the scapegoat tree. So far, all of the binary search trees that we have studied achieve balance through the use of rotation operation. Scapegoat trees are unique in that they do not rely on rotation. This is significant because there exist binary trees that cannot be balanced through the use of rotations. (One such example is the binary space partition tree shown in Fig. 2.) As we shall see, the scapegoat tree achieves good balance by "rebuilding" subtrees that exhibit poor balance. The trick behind scapegoat trees is figuring out which subtrees to rebuild and when to do this.

Below, we will discuss the details of how the scapegoat tree works. Here is a high-level overview. A scapegoat tree is a binary search tree, which does not need to store any additional information in the nodes, other than the key, value, and left and right child pointers. (Additional information, such as parent pointers may be added to simplify coding, however, but these are not needed.) Nonetheless, it height will always be $O(\log n)$. (Note that this is not the case for splay trees, whose height can grow to as high as $O(n)$.) Insertion and deletions work roughly as follows.

Insertion:

- The key is first inserted just as in a standard (unbalanced) binary tree
- We monitor the depth of the inserted node after each insertion, and if it is too high, there must be at least one node on the search path that has poor weight balance (that is, its left and right children have very different sizes).
- In such a case, we find such a node, called the *scapegoat*,[1] and we completely rebuild the subtree rooted at this node so that it is perfectly balanced.

Deletion:

- The key is first deleted just as in a standard (unbalanced) binary tree
- Once the number of deletions is sufficiently large relative to the entire tree size, rebuild the entire tree so it is perfectly balanced.

You might wonder why there is a notable asymmetry between the rebuilding rules for insertion and deletion. The existence of a single very deep node is proof that a tree is out of balance. Thus, for insertion, we can use the fact that the inserted node is too deep to trigger rebuilding. However, observe that the converse does not work for deletion. The natural counterpart would be "if the depth of the external node containing the deleted key is too small, then trigger a rebuilding operation." However, the fact that a single node has a low depth, does not imply that the rest of the tree is out of balance. (It may just be that a single search path has low depth, but the rest of the tree is perfectly balanced.) Could we instead apply the deletion rebuilding trigger to work for insertion? Again, this will not work. The natural counterpart would be, "given a newly rebuild tree with n keys, we will rebuild it after inserting roughly $n/2$

[1] A "scapegoat" is an individual who is assigned the blame when something goes wrong. In this case, the unbalanced node takes the blame for the tree's height being too great.

new keys." However, if we are very unlucky, all these keys may fall along a single search path, and the tree's height would be as bad as $O((\log n) + n/2) = O(n)$, and this is unacceptably high.

How to Rebuild a Subtree: Before getting to the details of how the scapegoat tree works, let's consider the basic operation that is needed to maintain balance, namely rebuilding subtrees into balanced form. We shall see that if the subtree contains k keys, this operation can be performed in $O(k)$ time. Suppose that p is a pointer to the node of the scapegoat tree whose subtree is to be rebuilt. We begin be performing an inorder traversal of p's subtree, copying the keys to an array $A[0, ..., k-1]$. Because we use an inorder traversal, the elements of A are in ascending sorted order.

To create the new subtree, we will define a procedure that extracts the median element of the array as the root, and then recursively rebuilds the subarrays to the left and right of the median, and then makes the resulting subtrees the left and right children of the median node. More formally, let us define a function `buildSubtree(A, i, k)`, which returns a reference to a balanced subtree containing the k-element subarray of A whose first element is $A[i]$, that is, $A[i, \ldots, i+k-1]$. Pseudocode is given in the code block below.

Building a Balanced Tree from an Array

```
BinaryNode buildSubtree(Key[] A, int i, int k) {
    if (k == 0) return null;                          // empty array
    else {
        int m = ceiling(k/2);                         // root is the median
        BinaryNode p = new BinaryNode(A[i+m]);        // A[i+m] is root
        p.left = buildSubtree(A, i, m);               // A[i..m-1] in left subtree
        p.right = buildSubtree(A, i+m+1, k-m-1);      // A[i+m+1..i+k-1] in right
        return p;                                     // return root of the subtree
    }
}
```

Ignoring the recursive calls, we spend $O(1)$ time within each recursive call, so the overall time is proportional to the size of the tree, which is k, so the total time is $O(k)$.

Scapegoat Tree Operations: In addition to the nodes themselves, the scapegoat tree maintains two integer values. The first, denoted by n, is just the number of keys in the tree. The second, denoted by m, is an upper bound on the size of the tree. This latter value plays a role in deciding when the rebalance the tree when deletions are performed. In particular, whenever we insert a key, we increment m, but whenever we delete a key we do not decrement m. Thus, $m \geq n$ and the difference $m - n$ intuitively represents the number of deletions. When we reach a point where $m > 2n$ (or equivalently $m - n > n$) we can infer that the number of deletions exceeds the number of keys remaining in the tree. In this case, we will rebuild the entire tree in balanced form.

We are now in a position to describe how to perform the dictionary operations for a scapegoat tree.

`find(Key x)`: The find operation is performed exactly as in a standard (unbalanced) binary search tree. They height of the tree never exceeds $\log_{3/2} n$, so this is guaranteed to run ins $O(\log n)$ time.

`delete(Key x)`: This operates exactly the same as deletion in a standard binary search tree. After the deletion, we decrement n, but we do not change m. As mentioned above, if $m > 2n$, we rebuild the entire tree, and set $m \leftarrow n$.

`insert(Key x, Value v)`: The begins exactly as insertion does for a standard binary search tree. But, as we are tracing the search path to the insertion point, keep track of our depth in the tree. (Recall that depth is the number of edges to root.) Increment both n and m. If the depth of the inserted node exceeds $\log_{3/2} m$ then we trigger a *rebuilding event*. This involves the following:

- Walk back up along the insertion search path. Let `u` be the current node that is visited, and let `u.child` be the child of `u` that lies on the search path.
- Let `size(u)` denote the *size* of the subtree rooted at `u`, that is, the number of nodes in this subtree. If
$$\frac{\texttt{size(u.child)}}{\texttt{size(u)}} > \frac{2}{3},$$
then rebuild the subtree rooted at `u` (e.g., using the method described above).

The fact that a child has over 2/3 of the nodes of the entire subtree intuitively means that this subtree has roughly speaking more than twice as many nodes as its sibling. We call such a node on the search path a *scapegoat candidate*. A short way of summarize the above process is "rebuild the scapegoat candidate that is closest to the insertion point." An example is shown in Fig. 6.

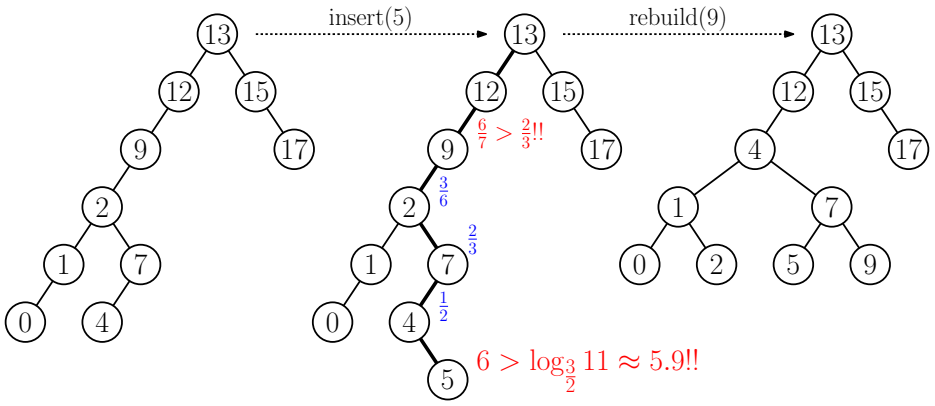

Fig. 6: Inserting a key into a scapegoat tree, which triggers a rebuilding event. The node containing 9 is the first scapegoat candidate encountered while backtracking up the search path and is rebuilt.

You might wonder whether we will necessarily encounter an scapegoat candidate when we trace back along the search path. The following lemma shows that this is always the case.

Lemma: Given a binary search tree of n nodes, if there exists a node p such that $\text{depth}(p) > \log_{3/2} n$, then p has an ancestor (possibly p itself) that is a scapegoat candidate.

Proof: The proof is by contradiction. Suppose to the contrary that no node from p to the root is a scapegoat candidate. This means that for every ancestor node u from p to the root, we have $\text{size}(u.\text{child}) \leq \frac{2}{3} \cdot \text{size}(u)$. We know that the root has a size of n. It follows that if p is at depth k in the tree, then

$$\text{size}(p) \geq \left(\frac{2}{3}\right)^k n.$$

We know that size$(p) \geq 1$ (since the subtree contains p itself, if nothing else), so it follows that $1 \geq (2/3)^k n$. With some simple manipulations, we have

$$\left(\frac{3}{2}\right)^k \leq n,$$

which implies that $k \leq \log_{3/2} n$. However, this violates our hypothesis that p's depth exceeds $\log_{3/2} n$, yielding the desired contradiction.

Recall that $m \geq n$, and so if a rebuilding event is triggered, the insertion depth is at least $\log_{3/2} m$, which means that it is at depth at least $\log_{3/2} n$. Therefore, by the above lemma, there must be a scapegoat candidate along the search path.

How to Compute Subtree Sizes? We mentioned earlier that the scapegoat tree does not store any information in the nodes other than the key, value, and left and right child pointers. So how can we compute `size(u)` for a node `u` during the insertion process?

Unfortunately, there is no clever way to do this efficiently (say in $O(\log n)$ time). Since we are doing this as we back up the search path, we may assume that we already know the value of $s' =$ `size(u.child)`, where this is the child that lies along the insertion search path. So, to compute `size(u)`, it suffices to compute the size of `u`'s other child. To do this, we perform a traversal of this child's subtree to determine its size s''. Given this, we have `size`$(u) = 1 + s' + s''$, where the $+1$ counts the node `u` itself.

You might wonder, how can we possibly expect to achieve $O(\log n)$ amortized time for insertion if we are using brute force (which may take as much as $O(n)$ time) to compute the sizes of the subtrees? The reason is to first recall that we do not need to compute subtree sizes unless a rebuild event has been triggered. Every node that we are visiting in the counting process will need to be visited again in the rebuilding process. Thus, the cost of this counting process can be accounted for in the cost of the rebuilding process, and hence it essentially comes for free!

By the way, there is an alternative method for computing sizes. This is to store the size value of each node explicity within each node. The size of a node is easy to update whenever there are changes in the tree's structure, since we have:

```
size(u) = (u == null ? 0 : size(u.left) + size(u.right))
```

While we are at it, it is worth noting that the height is just as easy to store and update:

```
height(u) = (u == null ? 0 : 1 + max(height(u.left), height(u.right)))
```

Amortized Analysis: We will not present a formal analysis of the amortized analysis of the scapegoat tree. The following theorem (and the rather sketchy proof that follows) provides the main results, however.

Theorem: Starting with an empty tree, any sequence of k dictionary operations (find, insert, and delete) to a scapegoat tree can be performed in time $O(k \log k)$.

Proof: (Sketch)

- Find: Because the tree's height is at most $\log_{3/2} m \leq \log_{3/2} 2n = O(\log n)$ the costs of a find operation is $O(\log n)$ (unconditionally).

- Delete: In order to rebuild a tree due to deletions, at least half the entries since the last full rebuild must have been deleted. By token-based analyses (recall stacks and rehashing from earlier lectures), it follows that the $O(n)$ cost of rebuilding the entire tree can be amortized against the time spent processing the deletions.
- Insert: This is analyzed by a potential argument. Intuitively, after any subtree of size k is rebuilt it takes $O(k)$ insertions to force this subtree to be rebuilt again. Charge the rebuilding time against these "cheap" insertions.

Corollary: The amortized complexity of the scapegoat tree with at most n nodes is $O(\log n)$.