CMSC 420: Lecture 4
Binary Search Trees

Searching: Searching is among the most fundamental problems in data structure design. We are storing a set of *entries* $\{e_1, \ldots, e_n\}$, where each e_i is a pair (x_i, v_i), where x_i is a *key value* drawn from some totally ordered domain (e.g., integers or strings) and v_i is an associated *data value*. The data value is not used in the search itself, but is needed by whatever application is using our data structure.

We assume that each key value occurs at most once in the data structure, and given an arbitrary search key x, the basic search problem is determining whether there exists an entry matching this key value. To implement this, we will assume that we are given two types, `Key` and `Value`. (In a Java implementation, this can be handled by defining a class with two generic types, one for the key and one for the value.) We will also assume that key values can be compared using the usual comparison operators, such as `<`, `==`, `>=`. In actual implementation, it is assumed that the `Key` class supports a function for comparing keys. For example, in Java's various map classes, it is assumed that the `Key` class implements the `Comparator` interface. This means that there is a function `compare`(x_1, x_2), which returns a negative integer, zero, or a positive integer depending on whether the x_1 is less than, equal to, or greater than x_2, respectively.

The Dictionary ADT: Perhaps the most basic example of a search data structure is the dictionary. A *dictionary* is an ADT that supports the operations of insertion, deletion, and finding. There are a number of additional operations that one may like to have supported, but these are the core operations.

`void insert(Key x, Value v)`: Stores an entry with the key-value pair (x, v). We assume that keys are unique, and so if this key already exists, an error condition will be signaled (e.g., an exception will be thrown).

`void delete(Key x)`: Delete the entry with x's key from the dictionary. If this key does not appear in the dictionary, then an error conditioned is signaled.

`Value find(Key x)`: Determine whether there is an entry matching x's key in the dictionary? If so, it returns a reference to associated value. Otherwise, it returns a `null` reference.

Other operations that might like to see in a dictionary include iterating the entries, answering range queries (that is, reporting or counting all objects between some minimum and maximum key values), returning the kth smallest key value, and computing set operations such as union and intersection.

There are three common methods for storing dictionaries: sorted arrays, hash tables, and binary search trees. We discuss two of these below. Hash tables will be presented later this semester.

Sequential Allocation: The most naive approach for implementing a dictionary data structure is to simply store the entries in a linear array without any sorting. To find a key value, we simply run sequentially through the list until we find the desired key. Although this is simple, it is not efficient. Searching and deletion each take $O(n)$ time in the worst case, which is very bad if n (the number of items in the dictionary) is large. Although insertion only involves $O(1)$ to insert a new item at the end of the array (assuming we don't overflow), it would require $O(n)$ to check that we haven't inserted a duplicate element.

An alternative is to sort the entries by key value. Now, *binary search* can be used to locate a key in $O(\log n)$ time, which is much more efficient. (For example, if $n = 1,000,000$, $\log_2 n$ is only around 20.) While searches are fast, updates are slow. Insertion and deletion require $O(n)$ time, since the elements of the array must be moved around to make space.

Binary Search Trees: In order to provide the type of rapid access that binary search offers, but at the same time allows efficient insertion and deletion of keys, the simplest generalization is called a *binary search tree*. The idea is to store the records in the nodes of a binary tree, such that an inorder traversal visits the nodes in increasing key order. In particular, if x is the key stored in the root node, then the left subtree contains all keys that are less than x, and the right subtree stores all keys that are greater than x (see Fig. 1(a)). (Recall that we assume that keys are distinct, so no other key can be equal to x.)

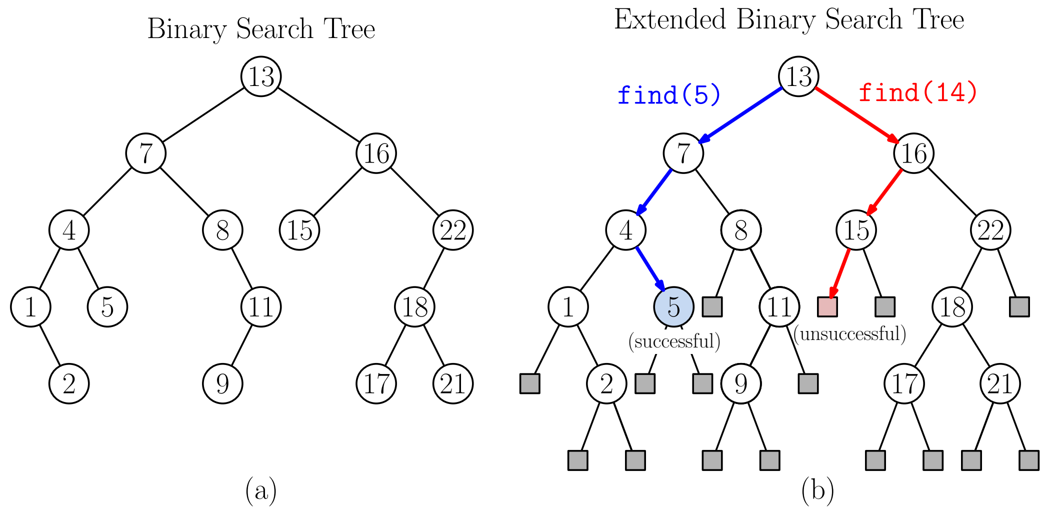

Fig. 1: Binary Search Tree.

Defining such an object in an object-oriented language like Java typically involves two class definitions. The main class is for the dictionary itself, and the other is for the individual nodes of the tree. We will call these `BinarySearchTree` and `BinaryNode`, respectively. The `BinarySearchTree` class has all the public functions and stores a reference to the root node of the tree. But most of the hard work is done by the methods associated with the `BinaryNode` class.

A node in the binary search tree would typically store the key, the value, and left and right pointers. It may also store additional information, such as parent pointers. When presenting our code examples, we will not be concerned with the value component, and will just focus on the key and the other pointers.

Search in Binary Search Trees: The search for a key x proceeds as follows. We start by assigning a pointer p to the root of the tree. We compare x to p's key, that is, `p.key`. If they are equal, we are done. Otherwise, if x is smaller, we recursively search p's left subtree, and if x is larger, we recursively visit p's right subtree. The search proceeds until we either find the key (see Fig. 1(b)) or we fall out of the tree (see Fig. 1(b)).

Note that if we think of the tree as an *extended tree*, then an unsuccessful search terminates at an external node. Each external node represents the unsuccessful searches for keys that lie between its inorder predecessor and inorder successor. (For example, in Fig. 1(b), the external node where the search for 14 ends represents all the searches for keys that are larger than 13 and smaller than 15.) One argument in favor of using extended trees is that the external node provides this additional information (as opposed to a simple `null` pointer).

A natural way to handle this would be to make the search procedure a recursive member function of the `BinaryNode` class. The initial call is made from the `find()` method associated with the `BinarySearchTree` class, which invokes `find(x, root)`, where `root` is the root of the tree.

Recursive Binary Tree Search

```
Value find(Key x, BinaryNode p) {
    if (p == null) return null;             // unsuccessful search
    else if (x < p.key)                     // x is smaller?
        return find(x, p.left);             // ... search left
    else if (x > p.key)                     // x is larger?
        return find(x, p.right);            // ... search right
    else return p.value;                    // successful search
}
```

It is easy to see based on the definition of a binary tree why this is correct. While most tree-based algorithms are best expressed recursively, this one is easy enough to do iteratively, and shown in the following code block. If you really wanted the best in performance, you would likely prefer this iterative form.

Iterative Binary Tree Search

```
Value find(Key x) {
    BinaryNode p = root;                    // start at the root
    while (p != null) {                     // proceed until we fall out of the tree
        if (x < p.key) p = p.left;          // x is smaller? ...search left
        else if (x > p.key) p = p.right;    // x is larger? ...search right
        else return p.value;                // successful search
    }
    return null;                            // unsuccessful search
}
```

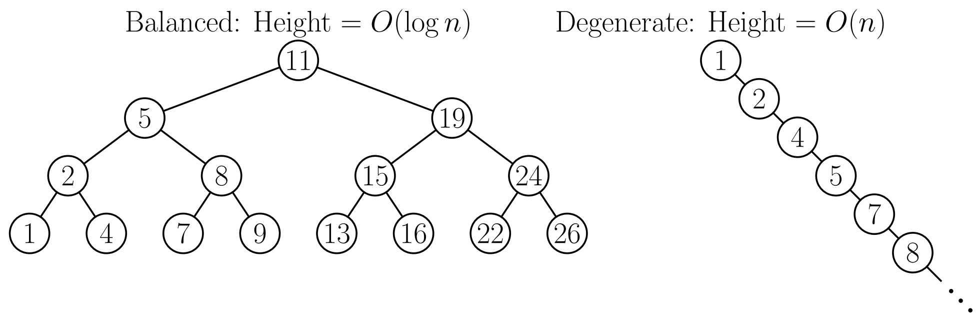

Fig. 2: Balanced and degenerate binary trees.

What is the running time of the search algorithm? Well, it depends on the key you are searching for. In the worst case, the search time is proportional to the height of the tree. The height of a binary search tree with n entries can be as low as $O(\log n)$ for the case of *balanced tree* (see Fig. 2 right) or as large as $O(n)$ for the case of a *degenerate tree* (see Fig. 2 left). However, we shall see that if the keys are inserted in random order, the expected height of the tree is just $O(\log n)$.

Insertion: To insert a new key-value entry (x, v) in a binary search tree, we first try to locate the key in the tree. If we find it, then the attempt to insert a duplicate key is an error. If not, we effectively "fall out" of the tree at some node p. We insert a new leaf node containing the desired entry as a child of p. It turns out that this is always the right place to put the new node. (For example, in Fig. 3, we fall out of the tree at the left child of node 15, and we insert the new node with key 14 here.)

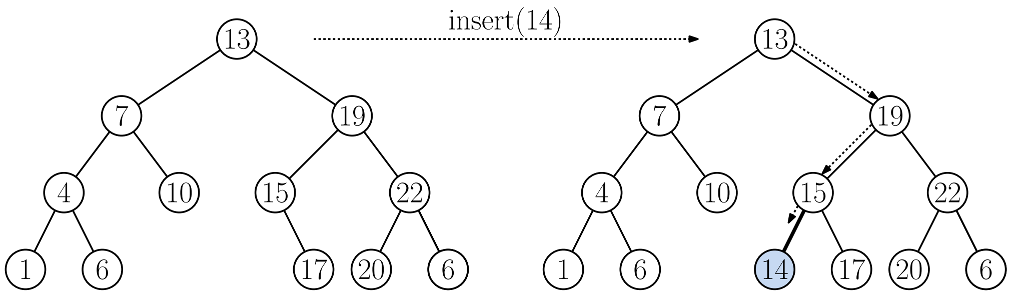

Fig. 3: Binary tree insertion.

The insertion procedure is shown in the code fragment below. There is one technical difficulty with implementing this in Java (or generally, any language that uses pass-by-value in function calls). When we create the new node, we want to "reach up" and modify one of the pointer fields in the parent's node. Unfortunately, this is not easy to do in our recursive formulation, since the parent node is not a local variable. There are a number of ways to fix this issue (including coding the procedure iteratively or explicitly passing in a reference to the parent node). Instead, we will employ a coding trick to get around this. In particular, the insertion function will return a reference to the modified subtree after insertion, and we store this value in the appropriate child pointer for the parent.

The initial call from the `BinarySearchTree` class is `root = insert(x, v, root)`. We assume that there is a constructor for the `BinaryNode`, which is given the key, value, and the initial values of the left and right child pointers.

Recursive Binary Tree Insertion

```
BinaryNode insert(Key x, Value v, BinaryNode p) {
    if (p == null)                              // fell out of the tree?
        p = new BinaryNode(x, v, null, null);   // ... create a new leaf node here
    else if (x < p.key)                         // x is smaller?
        p.left  = insert(x, v, p.left);         // ...insert left
    else if (x > p.key)                         // x is larger?
        p.right = insert(x, v, p.right);        // ...insert right
    else throw DuplicateKeyException;           // x is equal ...duplicate key!
    return p                                    // return ref to current node (sneaky!)
}
```

A Closer Look at the Trick: To better understand how our coding trick works, see Fig. 4 to insert 14. We first search for 14 in the tree, falling out of the tree at the left child of node 15, that is, when the local variable p refers to node 15. Let's call this local variable p_2. We generate a call `p2.left = insert(14, v, p2.left)`.

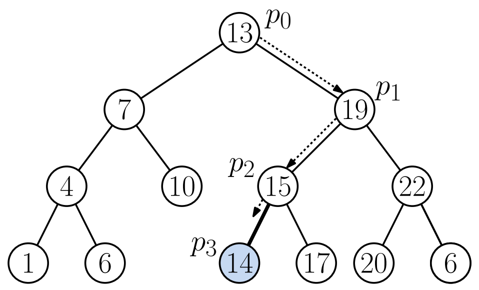

Fig. 4: Child link update in insertion.

Since node 15 has no left child, the next recursive call discovers right away that its local p (which has the value `p2.left`) value is `null`, and so we create the new node with key value 14, and assign it to the current local variable p. Let's call this p_3. The last line of the recursive procedure returns p_3 to the calling procedure at node 15, at the statement `p2.right = insert(14, v, p2.right)`. Since the `insert` function returns the pointer p_3 to the new node, we effectively perform the action `p2.right = p3`, which links the new node into the tree as desired. Voila!

Deletion: Next, let us consider how to delete an entry from the tree. Deletion is a more involed than insertion. While insertion adds nodes at the leaves of the tree, but deletions can occur at any place within the tree. Deleting a leaf node is relatively easy, since it effectively involves "undoing" the insertion process (see Fig. 5(a)). Deleting an internal node requires that we "fill the hole" left when this node goes away. The easiest case is when the node has just a single child, since we can effectively slide this child up to replace the deleted node (see Fig. 5(b)).

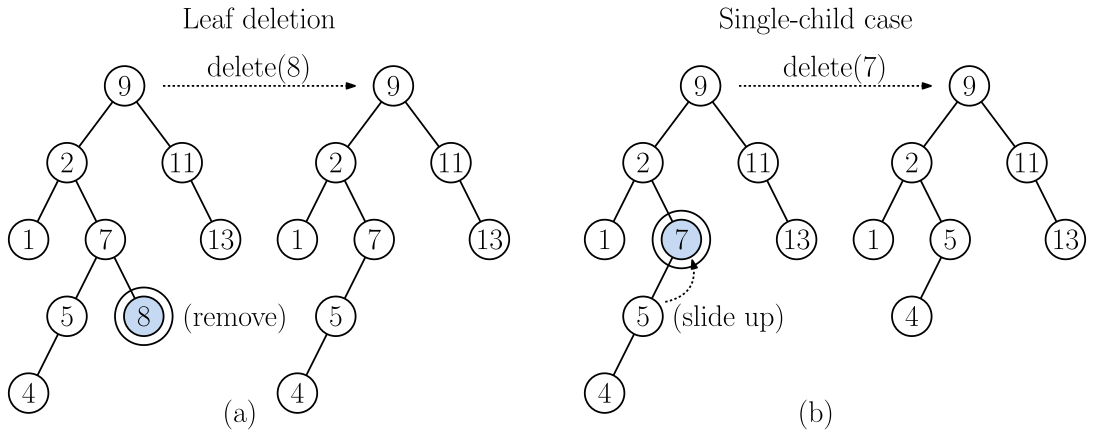

Fig. 5: Deletion: (a) Leaf and (b) single-child case.

The hardest case is when the deleted node that has two children. Let p denote the node to be deleted (see Fig. 6(a)):

- Find the node r that is p's inorder successor in the tree (see Fig. 6(b)). Note that because

p has two children, its inorder successor is the "leftmost" node of p's right subtree. Call r the *replacement node*.

- Copy the contents of r to p (see Fig. 6(c)).
- Delete node r (see Fig. 6(d)). (Because r's key immediately follows p's key, this replacement maintains the sorted order of keys.)

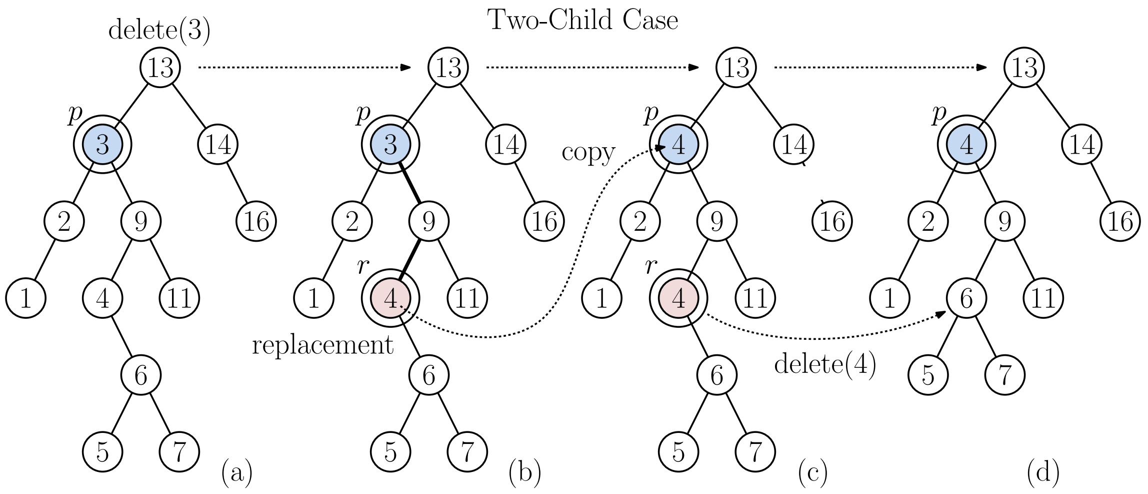

Fig. 6: Deletion: Two-child case.

It may seem that we have made no progress, because we have just replaced one deletion problem (for p) with another (for r). However, the task of deleting r is much simpler. The reason is, since r is p's inorder successor, r is the leftmost node of p's right subtree. It follows that r has no left child. Therefore, r is either a leaf or it has a single child, implying that it is one of the two "easy" deletion cases that we discussed earlier.

Deletion Implementation: Before giving the code for deletion, we first present a utility function, `findReplacement()`, which returns a pointer to the node that will replace p in the two-child case. As mentioned above, this is the inorder successor of p, that is, the leftmost node in p's right subtree. As with the insertion method, the initial call is made to the root of the tree, `delete(x, root)`. Again, we will employ the sneaky trick of returning a pointer to the revised subtree after deletion, and store this value in the child link. See the code fragment below.

Replacement Node for the Two-child Case

```
BinaryNode findReplacement(BinaryNode p) {        // find p's replacement node
   BinaryNode r = p.right;                         // start in p's right subtree
   while (r.left != null) r = r.left;              // go to the leftmost node
   return r;
}
```

The full deletion code is given in the following code fragment. As with insertion, the code is quite tricky. For example, can you see where the leaf and single-child cases are handled in the code? We do not have a conditional that distinguishes between these cases. How can that be correct. (But it is!)

Binary Tree Deletion

```
BinaryNode delete(Key x, BinaryNode p) {
    if (p == null)                                  // fell out of tree?
        throw KeyNotFoundException;                 // ...error - no such key
    else {
        if (x < p.data)                             // look in left subtree
            p.left = delete(x, p.left);
        else if (x > p.data)                        // look in right subtree
            p.right = delete(x, p.right);
                                                    // found it!
        else if (p.left == null || p.right == null) { // either child empty?
            if (p.left == null) return p.right;     // return replacement node
            else                return p.left;
        }
        else {                                      // both children present
            r = findReplacement(p);                 // find replacement node
            copy r's contents to p;                 // copy its contents to p
            p.right = delete(r.key, p.right);       // delete the replacement
        }
    }
    return p;
}
```

Analysis of Binary Search Trees: It is not hard to see that all of the procedures `find()`, `insert()`, and `delete()` run in time that is proportional to the height of the tree being considered. (The `delete()` procedure is the only one for which this is not obvious. Because the replacement node is the inorder successor of the deleted node, it is the leftmost node of the right subtree. This implies that the replacement node has no left child, and so it will fall into one of the easy cases, which do not require a recursive call.)

The question is, given a binary search tree T containing n keys, what can be said about the height of the tree? It is not hard to see that in the worst case, if we insert keys in either strictly increasing or strictly decreasing order, then the resulting tree will be completely degenerate, and have height $n - 1$. We will show that, if keys are inserted in random order, then the expected depth of any node is $O(\log n)$. (We emphasize that this assumes insertions only. See below for a discussion of the situation when insertions and deletions are combined.)

Proving that the expected depth is $O(\log n)$ is not an trivial exercise. The proof involves setting up a fairly complicated recurrence and solving it. (If you have ever seen the complete analysis of QuickSort, the two recurrences are very similar.) Instead, we will produce a "quick and dirty" proof, which hopefully will convince you that the assertion is reasonable, if not fully convincing. In particular, rather than proving that every node of the tree is at expected depth $O(\log n)$, we will prove that the *leftmost node* of the tree (that is, the node associated with the smallest key value) will be at expected depth $O(\log n)$.

Theorem: Given a set of n keys $x_1 < x_2 < \ldots < x_n$, let $D(n)$ denote the expected depth of node x_1 after inserting all these keys in a binary search tree, under the assumption that all $n!$ insertion orders are equally likely. Then $D(n) \leq 1 + \ln n$, where ln denotes the natural logarithm.

Proof: We will track the depth of the leftmost node of the tree through the sequence of insertions. Suppose that we have already inserted $i - 1$ keys from the sequence, and we are in the process of inserting the ith key. The only way that the leftmost node changes is when the ith key is the lowest key value that has been seen so far in the sequence. That is, the i element to be inserted in the new minimum value among all the keys in the tree.

For example, consider the insertion sequence $S = \langle 9, 5, 10, 6, 3, 4, 2\rangle$. Observe that the minimum value changes three times, when 5, 3, and 2 are inserted. If we look at the binary tree that results from this insertion sequence we see that the depth of the leftmost node also increases by one with each of these insertions.

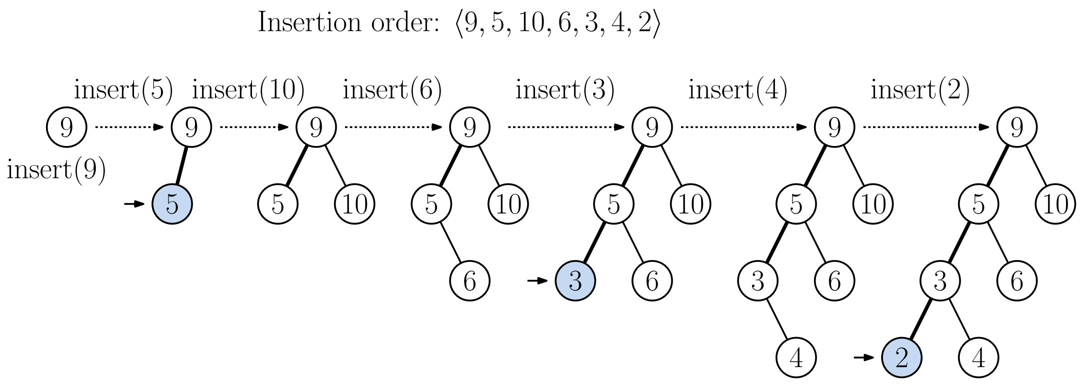

Fig. 7: Length of the leftmost chain.

To complete the analysis, it suffices to determine (in expectation) the number of times that the minimum in a sequence of n random values changes. To make this formal, for $2 \leq i \leq n$, let X_i denote the random variable that is 1 if the ith element of the random sequence is the minimum among the first i elements, and 0 otherwise. (In our sequence S above, $X_2 = X_5 = X_7 = 1$, because the minimum changed when the second, fifth, and seventh elements were added. The remaining X_i's are zero.)

To analyze X_i, let's just focus on the first i elements and ignore the rest. Since every permutation of the numbers is equally likely, the minimum among the first i is equally likely to come at any of the positions first, second, ..., up to ith. The minimum changes only if comes last out of the first i. Thus, $\Pr(X_i = 1) = \frac{1}{i}$ and $\Pr(X_i = 0) = 1 - \frac{1}{i}$. Whenever this random event occurs ($X_i = 1$), the minimum has changed one more time. Therefore, to obtain the expected number of times that the minimum changes, we just need to sum the probabilities that $X_i = 1$, for $i = 2, \ldots, n$. Thus we have

$$D(n) \;=\; \sum_{i=2}^{n} \frac{1}{i} \;=\; \left(\sum_{i=1}^{n} \frac{1}{i}\right) - 1.$$

This summation is among the most famous in mathematics. It is called the *Harmonic Series*. Unlike the geometric series $(1/2^i)$, the Harmonic Series does not converge. But it is known that when n is large, its value is very close to $\ln n$, the natural log of n. (In fact, it is not more than $1 + \ln n$.)

Therefore, we conclude that the expected depth of the leftmost node in a binary search tree under n random insertions is at most $1 + \ln n = O(\log n)$, as desired.

Random Insertions and Deletions: Interestingly, this analysis breaks down if we are doing both insertions and deletions. Suppose that we consider a very long sequence of insertions and deletions, which occur at roughly the same rate so that, in steady state, the tree has roughly n nodes. Let us also assume that insertions are random (drawn say from some large domain of candidate elements) and deletions are random in the sense that a random element from the tree is deleted each time.

It is natural to suppose that the $O(\log n)$ bound should apply, but remarkably it does not! It can be shown that over a long sequence, the height of the tree will converge to a significantly larger value of $O(\sqrt{n})$.[1]

The reason has to do with the fact that the replacement element was chosen in a biased manner, always taking the inorder successor. Over the course of many deletions, this repeated bias causes the tree's structure to skew away from the ideal. This bias can be eliminated by selecting the replacement node (randomly) as the inorder successor or inorder predecessor. It has been shown experimentally that this resolves the issue, but (to the best of my knowledge) it is not known whether the expected height of this balanced version of deletion matches the expected height for the insertion-only case (see Culberson and Munro, Algorithmica, 1990).

[1]There is an interesting history regarding this question. It was believed for a number of years that random deletions did not alter the structure of the tree. A theorem by T. N. Hibbard in 1962 proved that the tree structure was probabilistically unaffected by deletions. The first edition of D. E. Knuth's famous book on data structures, quotes this result. In the mid 1970's, Gary Knott, a Ph.D. student of Knuth and later a professor at UMD, discovered a subtle flaw in Hibbard's result. While the structure of the tree is probabilistically the same, the distribution of keys is not. However, Knott could not resolve the asymptotic running time. The analysis showing that $O(\sqrt{n})$ bound was due to Culberson and Munro in the mid 1980's.