CS 624
Lecture 7: Binary Search Trees

1 Graphs, paths, and trees

Definition *A **path** in a graph is a sequence*

$$v_0, v_1, v_2, \cdots, v_n$$

where each v_j is a vertex in the graph and where for each i, v_i and v_{i+1} are joined by an edge. (If the graph is directed, then the edge must go from v_i to v_{i+1}.)

To make things simple, we insist that any path contain at least one edge.

Usually we write

$$v_0 \to v_1 \to v_2 \to \cdots \to v_n$$

to denote a path. It is understood that the arrow represent edges.

*A path in a graph is **simple** iff it contains no vertex more than once.*

Definition *A **loop** in a graph is a path which begins and ends at the same vertex.*

*A loop $v_0 \to v_1 \to v_2 \to \cdots \to v_k$ is **simple** iff*

1. *$k \geq 3$ (that is, there are at least 4 vertices on the path), and*
2. *it contains no vertex more than once, except of course for the first and last vertices, which are the same, and*
3. *That (first and last) vertex occurs exactly twice.*

1.1 Exercise *Prove that a simple loop contains no edge more than once.*

Definition *A **tree** is a connected*[1] *undirected graph in which there are no simple loops.*

1.2 Exercise *Prove that if x and y are any two distinct vertices in a tree, there is a unique simple path from x to y.*

Hint*: If there were two simple paths from x to y, then of course you could follow one forwards, followed by the other one backwards to see that there is a loop in the graph. However, this loop*

[1]Do you know what it means to say that a graph is *connected*? If you don't, you need to look it up right now.

might not be a simple *loop, which is what you would need to get a contradiction. However, if the paths are not identical, then you should be able to take part of the first path followed by part of the second (going backwards) to produce a simple loop in the graph. Be as precise as you can in doing this. Give names to everything you are talking about. Remember that I can't read your mind.*

Definition *A* ***rooted tree*** *is a tree with a distinguished vertex, which we call the* ***root****. We will denote the root by r.*

Definition *If T is a rooted tree with root r, if x and y are vertices in T (and either or both of them might be r) and if there is a simple path from r through x to y, then we say that x is an* ***ancestor*** *of y and y is a* ***descendant*** *of x. If the part of the path from x to y consists of exactly one edge, we say that x is the* ***parent*** *of y and y is a* ***child*** *of x.*

Note that a vertex is both an ancestor and a descendant of itself. But a vertex cannot be its own parent.

1.3 Exercise

1. *Prove that a vertex in a rooted tree can have at most one parent.*
2. *Prove that every vertex other than the root has exactly one parent.*

1.4 Exercise *Prove that in a rooted tree T, if x is an ancestor of y and y is an ancestor of x, then $x = y$.*

1.5 Exercise *Suppose T is a rooted tree with root r and $x \neq r$ is an vertex in T. (This is just a standard way of saying that x is a vertex in T and $x \neq r$.)*

Further, suppose a and b are both ancestors of x. And to make things simple, suppose that $a \neq b$.

Prove that either a is an ancestor of b or b is an ancestor of a.

1.6 Exercise *Prove that any two nodes x and y in a rooted tree have a unique* ***least common ancestor*** *z. This simply means that there is a path P_1 from z to x, and a path P_2 from z to y, and that the only vertex those two paths have in common is z. (And further, that there is exactly one node z with this property.)*

We think of the root as being at the top of the tree, so the least common ancestor is the ancestor that is as far away "down" the tree as possible. That's where the term "least" comes from.

2 Traversing binary trees

When we write pseudocode involving binary trees, we will denote the left and right children of a node x by $x.\mathit{left}$ and $x.\mathit{right}$, respectively, and for brevity, we will denote the parent of x by $x.p$. If x has no parent (which of course can only happen if x is the root) then by convention we say that $x.p =$ NIL.

There are three basic ways to traverse (or "walk") a binary tree. The idea is that we want to perform some function at each node. This is called "visiting" the node. And we want to recursively traverse each child of the node. The only difference between these three methods of traversal is the order in which these things happen.

preorder tree walk. Visit the node. Then traverse its children, left-to-right.

inorder tree walk. Traverse the left child. Then visit the node. Then traverse the right child.

postorder tree walk. Traverse the children, left-to-right. Then visit the node.

The reason for the use of the term "order" is that (as we will see below, a major use of binary trees is to keep ordered data in such a way that all the elements on the left subtree of a node are less than the element at the node, and all the elements on the right subtree of the node are greater than the element at the node. So an inorder tree walk visits the nodes "in order".

The pseudocode for these three methods is then quite immediate.

```
PREORDER-TREE-WALK($x$)
    if $x \neq$ NIL then
        visit $x$
        PREORDER-TREE-WALK($x.left$)
        PREORDER-TREE-WALK($x.right$)

INORDER-TREE-WALK($x$)
    if $x \neq$ NIL then
        INORDER-TREE-WALK($x.left$)
        visit $x$
        INORDER-TREE-WALK($x.right$)

POSTORDER-TREE-WALK($x$)
    if $x \neq$ NIL then
        POSTORDER-TREE-WALK($x.left$)
        POSTORDER-TREE-WALK($x.right$)
        visit $x$
```

It's intuitively obvious that the cost of any of these tree walks on a tree with n nodes is $\Theta(n)$. Here's a proof:

2.1 Theorem *If x is the root of a binary tree with n nodes, then each of the above traversals takes $\Theta(n)$ time.*

PROOF. Let us define

$$\begin{aligned} c &= \text{time for the test } x \neq \text{NIL} \\ v &= \text{time for the call to visit } x \\ T(k) &= \text{time for the call to traverse a tree with } k \text{ nodes} \end{aligned}$$

Then certainly we have

$$T(0) = c \tag{1}$$

and if the tree with n nodes has a right child with k nodes (so its left child must have $n-k-1$ nodes), then

$$T(n) = c + T(k) + T(n-k-1) + v \tag{2}$$

Exercise 2.2 then shows that $T(n) = (2c+v)n + c$ □

2.2 Exercise *Let us guess that perhaps $T(n)$ can be expressed as an affine function of n:*

$$T(n) = \alpha n + \beta \tag{3}$$

(We have already seen that this is a reasonable guess.) Substitute equation (3) in the equations (1) and (2) and solve the two resulting equations for α and β.

3 Binary search trees

Definition *A **binary search tree** is a binary tree, each node of which contains some data, one of whose fields is a key, such that for each node x,*

- *If y is a node in the left subtree of x, then $y.\mathit{key} \le x.\mathit{key}$.*
- *If y is a node in the right subtree of x, then $y.\mathit{key} \ge x.\mathit{key}$.*

As a consequence, we can print out all the nodes in a binary search tree in the order of their keys by using an inorder tree walk, where the "visit" procedure just consists of printing out the node.

4 Operations on binary search trees

Searching Here is a procedure which starts at a node x and searches all the descendants of x until it finds one with key k. It returns that node (or NIL if it doesn't find one).

```
TREE-SEARCH($x, k$)
    if $x =$ NIL or $k = x.key$ then
        return $x$
    if $k < x.key$ then
        return TREE-SEARCH($x.left, k$)
    else
        return TREE-SEARCH($x.right, k$)
```

The running time is $O(h)$, where h is the height of the tree.

Here is an iterative version:

```
ITERATIVE-TREE-SEARCH($x, k$)
    while $x \ne$ NIL and $k \ne x.key$ do
        if $k < x.key$ then
            $x \leftarrow x.left$
        else
            $x \leftarrow x.right$
    return x
```

Minimum and maximum This procedure finds the descendant of x whose key is minimum. It just follows the left child as far as possible. And to find the maximum, we just follow the right child as far as possible.

```
TREE-MINIMUM($x$)
    while $x.left \neq$ NIL do
        $x \leftarrow x.left$
    return $x$
```

```
TREE-MAXIMUM($x$)
    while $x.right \neq$ NIL do
        $x \leftarrow x.right$
    return $x$
```

The cost of each of these procedures is also $O(h)$.

Successor and predecessor By the **successor** of a node in a binary search tree we mean the node which is the successor in the order determined by the tree. (Be careful: "successor" is *not* generally the same as "child".)

4.1 Exercise *In doing this exercise, you may assume that all the keys in a binary search tree are distinct, i.e., that no two are equal.*

Let x be a node in a binary search tree.

1. *If a is an ancestor of x and $a.key > x.key$, then for any descendant d of x, we have $a.key > d.key$.*
2. *If a is the successor of x, then a is either an ancestor or descendant of x. (Hint: use the result of Exercise 1.6.)*
3. *If the right subtree of x is nonempty, then the successor of x is just the leftmost node in the right subtree. (Hint: use the previous result.)*
4. *If the right subtree of x is empty, and if x has a successor y (i.e., x is not the largest element in the tree), then y is the lowest ancestor of x whose left child is also an ancestor of x. (Hint: again use part 2 of this exercise.)*
5. *If x has a right child, then the successor of x does not have a left child.*
6. *Similarly, if x has a left child, then the predecessor of x does not have a right child.*

For instance, in Figure 1,

1. *The successor of the node with key 15 is the node with key 17.*
2. *The successor of the node with key 13 is the node with key 15.*
3. *The successor of the node with key 17 is the node with key 18.*

```
TREE-SUCCESSOR($x$)
    if $x.right \neq$ NIL then
        return TREE-MINIMUM($x.right$)
    $y \leftarrow x.p$
    while $y \neq$ NIL and $x = y.right$ do
        $x \leftarrow y$
        $y \leftarrow x.p$
    return $y$
```

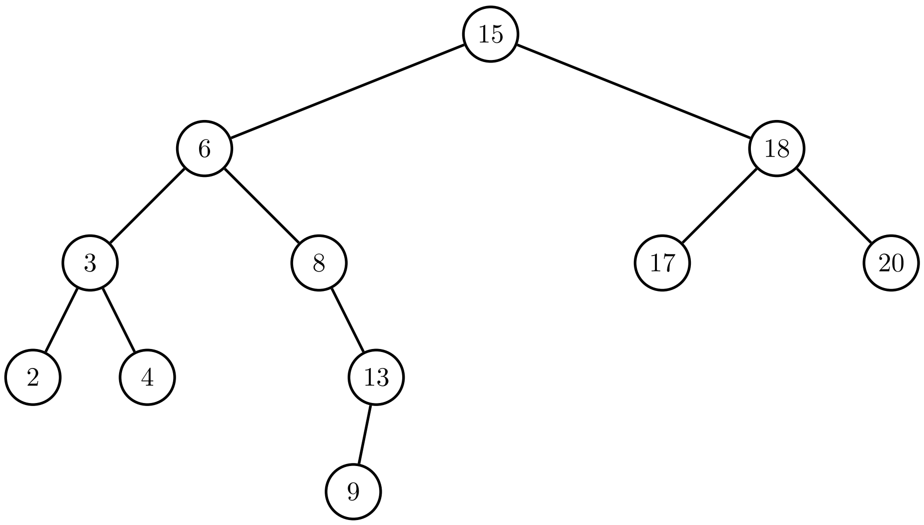

Figure 1: A binary search tree.

(What happens when we apply this procedure to the node in Figure 1 whose key is 20?)

The running time of TREE-SUCCESSOR on a tree of height h is again $O(h)$, since the algorithm consists on following a path from a node to its successor, and the maximum path length up or down the tree is by definition h.

TREE-PREDECESSOR is defined similarly and also has running time $O(h)$ on a tree of height h. Thus we have proved the following theorem:

4.2 Theorem *The dynamic-set operations* SEARCH, MINIMUM, MAXIMUM, SUCCESSOR, *and* PREDECESSOR *can be made to run in $O(h)$ time on a binary search tree of height h.*

Insertion To insert a new node z in the binary search tree T, we just walk down from the root, comparing each node to z until we get to a place where we can insert z as a new leaf. None of the other nodes in the tree is moved by this procedure. Figure 2 illustrates how a new node with key 7 is inserted into the tree in Figure 1.

As we walk down the tree, x is the node we currently are at, and y is its parent.

```
TREE-INSERT($T, z$)
    $y \leftarrow$ NIL
    $x \leftarrow T.root$
    while $x \neq$ NIL do
        $y \leftarrow x$
        if $z.key < x.key$ then
            $x \leftarrow x.left$
        else
        $x \leftarrow x.\mathit{right}$
$z.p \leftarrow y$
**if** $y = \textsc{nil}$ **then**
    // This can only happen if the tree $T$ was empty to begin with.
    $T.\mathit{root} \leftarrow z$
**else if** $z.\mathit{key} < y.\mathit{key}$ **then**
    $y.\mathit{left} \leftarrow z$
**else**
    $y.\mathit{right} \leftarrow z$
```

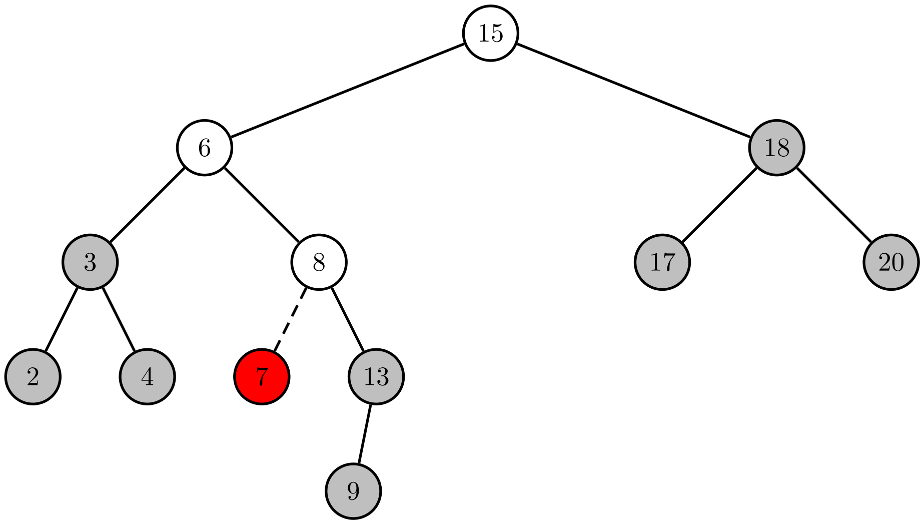

Figure 2: Inserting a new node in a binary search tree.

For exactly the same reason as before, the procedure TREE-INSERT runs in $O(h)$ time on a tree of height h.

Deletion Deleting a node is somewhat more complicated, since if the node is buried within the tree, we will have to move some of the other nodes around. Nevertheless, the idea is quite simple. There are three possible cases we need to consider; they are shown in Figure 3.

Let us say that we are deleting a node d.

Case I: d is a leaf. This case is trivial. Just delete the node. This amounts to figuring out which child it is of its parent, and making the corresponding child pointer NIL. The node d itself can be returned from the procedure (to be used or recycled by the caller), or it can simply be deleted inside the procedure itself.

Case II: d has one child. In this case, delete d and "splice" its child to its parent—that is, make the parent's child pointer that formerly pointed to d now point to d's child, and make that

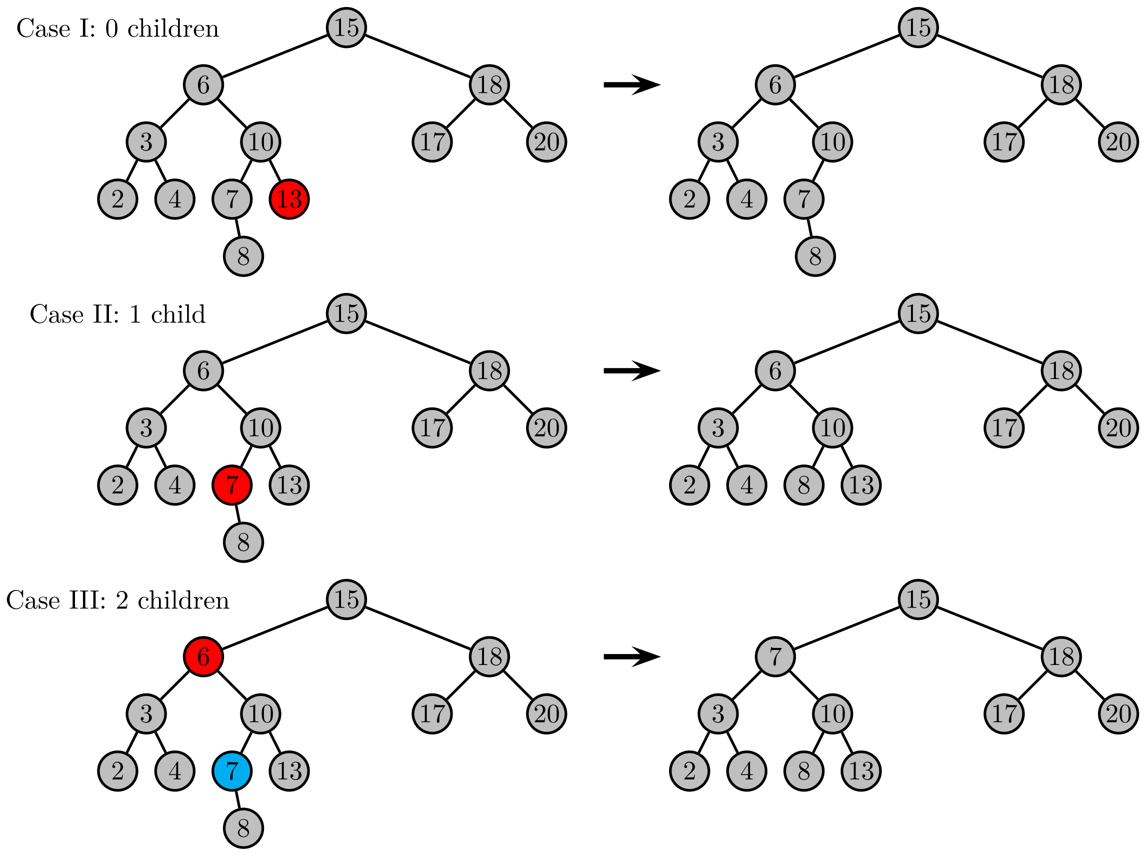

Figure 3: Deleting a node in a binary search tree. In each case, the red node is the node that is being deleted.

The only somewhat tricky case is Case III. In that case, the red node is replaced by the blue node, which is its successor in the tree. Then the blue node is deleted from its original position. We know that the blue node has at most 1 child, so it falls into Case I or Case II.

child's parent pointer now point to d's parent.

Case III: d has two children. In this case we can't simply move one of the children of d into the position of d. We we need to do is find d's successor and replace d with it. Then delete d's successor.

This sounds like a recursive call, but it really isn't. We know that d's successor has at most 1 child, so we can immediately apply either Case I or Case II to it, and then we are done.

The text has some pseudocode which performs essentially this processing, but they have organized it in a clever fashion which is probably slightly more efficient. Nevertheless, this is the idea, and this is really all you have to understand.

5 Expected cost of building a binary search tree

An algorithm to build a binary search tree from an array $A[1\,..\,n]$ of unsorted inputs would be something like this:

```
BUILD-BINARY-SEARCH-TREE($A[1\,..\,n]$)
    $T \leftarrow \varnothing$ // Create an empty tree
    for $i \leftarrow 1$ to $n$ do
        TREE-INSERT($T, A[i]$)
```

How expensive is this?

- If A is already in sorted order, this is $\Theta(n^2)$.
- The best case would appear to be $\Theta(n \log n)$.

This looks suspiciously like quicksort, and indeed the analysis is entirely the same:

5.1 Exercise *Here is a modified version of the* PARTITION *algorithm for* QUICKSORT*:*

```
MODIFIED-PARTITION($A, p, r$)
    $pivot \leftarrow A[p]$
    Let L be the sequence of elements of $A[p+1\,..\,q]$ that are less than pivot,
        in the order they appear in A.
    Let U be the sequence of elements of $A[p+1\,..\,q]$ that are greater than pivot,
        in the order they appear in A.
    Rearrange the elements in $A[p\,..\,q]$ so that they appear like this:

                                $L\ pivot\ U$
```

To implement this would be some constant multiple more expensive than the cost of the original PARTITION*—which doesn't matter for our analysis—and would require more memory. For instance, one obvious way to implement this would be to allocate a new array of the same size as the original array and perform the* MODIFIED-PARTITION *by putting the original elements in the new array one by one. This would still be an $O(n)$ partition algorithm. And since all we're concerned about here is running time, the fact that this uses more memory is irrelevant. (Also,* MODIFIED-PARTITION *chooses the pivot element to be the left-most, rather than the right-most element of the array. This is a small point of no real significance.)*

So in any case, the analysis of the running time of quicksort is the same with MODIFIED-PARTITION *as with* PARTITION.

Prove that the comparisons needed to build a binary search tree from an array $A[1\,..\,n]$ are exactly the same comparisons needed to run QUICKSORT *on that array, providing we use* MODIFIED-PARTITION.

Hint: the comparisons used in QUICKSORT *are the comparisons with appropriate pivot elements. And the successive pivot elements are exactly the successive elements added to the binary search tree. Your job is to write this up clearly and convincingly.*

A good way to start would be to take a simple example—say, a very small array—and work through it by hand. Perform QUICKSORT *using the* MODIFIED-PARTITION *algorithm. Then start again* *with the same initial array and perform* BUILD-BINARY-SEARCH-TREE*. Convince yourself that exactly the same comparisons are performed, and that the successive pivot elements in the modified* QUICKSORT *algorithm are the same as the successive elements added to the binary search tree in* BUILD-BINARY-SEARCH-TREE*.*

Then when you have convinced yourself of this for a relatively small array (don't make it too small!), you should be able to write an explanation—a convincing *explanation—that works for any array.*

We know that the average-case running time of QUICKSORT is $\Theta(n \log n)$. What would it mean to talk about the "average-case" running time to build a binary search tree?

What we would need to do is to average the running time over all permutations of the input array. This is exactly what we get when we computing the running time of randomized QUICKSORT. Therefore, we can say that

5.2 Theorem *The average-case running time of* BUILD-BINARY-SEARCH-TREE *is* $\Theta(n \log n)$.

6 Expected cost of a search in a binary search tree

We have already seen that the cost of a search in a binary search tree is $O(h)$, where h is the height of the tree. So this leads naturally to the question: What's the cost on average?

Questions like this are tricky, and this is in part because small changes in the wording of the question can give vastly different answers.

Here is one way we might approach this: We know that the cost of searching for a node is the depth of that node. Now the depth of a node is also the number of comparisons that are made in inserting the node into the tree. And we have seen that the total number of comparisons that are made in building an "average" tree with n nodes is $O(n \log n)$. Therefore the average node depth is that divided by the number of nodes (i.e., divided by n), which is $O(\log n)$. That is, the average node depth (averaged over all binary search trees) is $O(\log n)$, and so the average cost of searching for a node in a randomly built binary search tree is $O(\log n)$. That is, we have proved the following:

6.1 Theorem *The expected cost of a search in a binary search tree with n vertices is* $O(\log n)$.

7 Expected worst-case cost of search in a binary search tree

What we just proved is an interesting result. However, it still might be that even though the average cost of searching for a node is $(\log n)$, there might still be some nodes with much higher costs. Of course this *can* be true—if we have a "linear" tree, then the average cost of searching for a node is $O(n)$. And even if some tree containing n nodes is reasonably balanced so that the average of searching for a node in that tree is $O(\log n)$, it still might be the case that the tree in question contains a long "linear spike" leading to a leaf node of much higher cost. Thus, there might be a significant sub-family of trees having a nice $O(\log n)$ cost on average, but perhaps an $\Omega(n)$ worst-case cost.

In fact however, this can't happen to any great extent. It turns out that the *average worst-case* cost (again, averaged over all binary search trees) is still $O(\log n)$. This is remarkable.

We will show this by proving the following theorem:

7.1 Theorem *The expected height of a binary search tree with n vertices is $O(\log n)$.*

Let X_n be a random variable whose value is the height of a binary search tree on n keys. What does this actually mean? It actually means this:

Let $\mathcal{P}_n$ be the set of all permutations of those n keys. (So the number of elements of $\mathcal{P}_n$ is $n!$.) Suppose we use the symbol π to denote a permutation in $\mathcal{P}_n$. X_n is actually a function on $\mathcal{P}_n$. Its value $X_n(\pi)$ when applied to a permutation $\pi \in \mathcal{P}_n$ is the height of the binary search tree built from that permutation π.

We want to find $E(X_n)$, the *expectation* of X_n. This is by definition simply

$$\sum_{\pi\in\mathcal{P}_n} p(\pi)X_n(\pi)$$

where $p(\pi)$ denotes the probability of the permutation π. We are assuming that all permutations have equal probability, so $p(\pi) = \frac{1}{n!}$ for all π, and so

$$E(X_n) = \frac{1}{n!}\sum_{\pi\in\mathcal{P}_n} X_n(\pi)$$

The derivation I will give here is a bit simpler than that in the text, I think.

Note, by the way, that if A and B are two random variables on the same space $\mathcal{P}_n$, then

$$\begin{aligned} E(A+B) &= \sum_{\pi\in\mathcal{P}_n} p(\pi)\bigl(A(\pi)+B(\pi)\bigr) \\ &= \sum_{\pi\in\mathcal{P}_n} p(\pi)A(\pi) + \sum_{\pi\in\mathcal{P}_n} p(\pi)B(\pi) \\ &= E(A)+E(B) \end{aligned}$$

Note that $\max\{A, B\}$ is also a random variable on $\mathcal{P}_n$—its value at π is just $\max\bigl\{A(\pi), B(\pi)\bigr\}$. And we have the useful inequality (provided only that $A(\pi)$ and $B(\pi)$ are both ≥ 0 for all π)

$$\begin{aligned} E\bigl(\max\{A,B\}\bigr) &= \sum_{\pi\in\mathcal{P}_n} p(\pi)\max\bigl\{A(\pi),B(\pi)\bigr\} \\ &\leq \sum_{\pi\in\mathcal{P}_n} p(\pi)\bigl(A(\pi)+B(\pi)\bigr) \\ &= E(A+B) \\ &= E(A)+E(B) \end{aligned}$$

Now consider a permutation π. The root of the tree will be the first element of π. Suppose the root has position k in the sorted list of keys. That means that there will be $k-1$ keys less than it and $n-k$ keys greater than it. So the left subtree will have $k-1$ elements and the right subtree will have $n-k$ elements. And those elements are also chosen randomly from sets of size $k-1$ and $n-k$ respectively, so we have

$$X_n(\pi) = 1 + \max\{X_{k-1}(\pi), X_{n-k}(\pi)\} \tag{4}$$

This is our fundamental recursion[2].

Now in fact, since each value of k is chosen with the same probability (that probability being $\frac{1}{n}$), we have

$$E(X_n) = \sum_{k=1}^{n} \frac{1}{n} E\bigl(1 + \max\{X_{k-1}, X_{n-k}\}\bigr) \tag{5}$$

Here is where something tricky happens: it turns out that if we continued in this manner, we would not be able to get a good estimate for $E(X_n)$. What turns out to be more effective is this:

- Set $Y_n = 2^{X_n}$. So Y_n is itself a random variable defined on the set $\mathcal{P}$ whose value on the permutation π is

  $$Y_n(\pi) = 2^{X_n(\pi)}$$

  There is no intuitive significance to the reason for doing this—at least, not at first. It's just that we can do better with the mathematics that way.

- Compute $E(Y_n)$.

- Then use this to get a bound on $E(X_n)$. This step is also somewhat tricky if you haven't seen it before, but it is a general technique.

So let us proceed.

Starting with equation (4) using each side as a power of 2, we get

$$\begin{aligned} Y_n(\pi) &= 2^{X_n(\pi)} \\ &= 2^{1+\max\{X_{k-1}(\pi), X_{n-k}(\pi)\}} \\ &= 2 \cdot 2^{\max\{X_{k-1}(\pi), X_{n-k}(\pi)\}} \\ &= 2 \cdot \max\{2^{X_{k-1}(\pi)}, 2^{X_{n-k}(\pi)}\} \\ &= 2 \cdot \max\{Y_{k-1}(\pi), Y_{n-k}(\pi)\} \end{aligned} \tag{6}$$

and so (again, since each value of k is chosen with probability $\frac{1}{n}$)

$$\begin{aligned} E(Y_n) &= \sum_{k=1}^{n} \frac{1}{n} \cdot 2E\bigl(\max\{Y_{k-1}, Y_{n-k}\}\bigr) \\ &= \frac{2}{n} \sum_{k=1}^{n} E\bigl(\max\{Y_{k-1}, Y_{n-k}\}\bigr) \\ &\le \frac{2}{n} \sum_{k=1}^{n} \bigl(E(Y_{k-1}) + E(Y_{n-k})\bigr) \end{aligned} \tag{7}$$

[2] This notation is actually a bit sloppy. X_{k-1} really should be a function on permutations of $k-1$ elements. So what I really mean by $X_{k-1}(\pi)$ is the depth of the binary search tree built from the first $k-1$ elements of the permutation π (which are less than the pivot). And $X_{n-k}(\pi)$ really is the depth of the binary search tree built from the $n-k$ nodes of the permutation π which are greater than the pivot. It should be at least intuitively clear that the average value of $X_{k-1}(\pi)$ as π runs over all permutations of size n is the same as the average value of $X_{k-1}(\pi')$, where π' runs over all permutations of size $k-1$—this is what we would normally mean by the average value (or "expectation") of X_{k-1}

Now each term in the sum occurs twice (once as the "left term" and once as the "right term"). So we can simplify to get this:

$$\begin{aligned} E(Y_n) &\le \frac{4}{n}\sum_{k=1}^{n} E(Y_{k-1}) \\ &= \frac{4}{n}\sum_{k=0}^{n-1} E(Y_k) \end{aligned}$$

It would actually be a lot more convenient for us if this were a strict equality, rather than an inequality. It turns out that we can actually assume this to be the case, for the following reason: we're really only concerned here with an upper bound. Now here's a lemma that gives us what we need:

7.2 Lemma *If f and g are two functions such that*

$$f(0) = g(0) \tag{8}$$

$$f(n) \le \frac{4}{n}\sum_{k=0}^{n-1} f(k) \tag{9}$$

$$g(n) = \frac{4}{n}\sum_{k=0}^{n-1} g(k) \tag{10}$$

then $f(k) \le g(k)$ for all $k \ge 1$.

Proof. We'll prove this by induction. The inductive hypothesis is that $f(k) \le g(k)$ for all $k < n$. We know that this statement is true for $n = 2$ by (8). The inductive step is then to show that this statement remains true for $k = n$. To show this, we just compute as follows:

$$\begin{aligned} f(n) &\le \frac{4}{n}\sum_{k=0}^{n-1} f(k) && \text{(by (9))} \\ &\le \frac{4}{n}\sum_{k=0}^{n-1} g(k) && \text{(by the inductive hypothesis)} \\ &= g(n) && \text{(by (10))} \end{aligned}$$

and that completes the proof of the lemma. □

So based on this, we can assume that

$$E(Y_n) = \frac{4}{n}\sum_{k=0}^{n-1} E(Y_k) \tag{11}$$

because by the Lemma, any upper bound we obtain for $E(Y_n)$ from this identity will also be an upper bound for the "real" $E(Y_n)$.

Now (11) might remind you of the clever device we used in deriving the average case running time of Quicksort. We can in fact do something very similar here, although it is a little more complicated (but not much):

First, we want to get rid of the sum. So we write

$$E(Y_{n+1}) = \frac{4}{n+1}\sum_{k=0}^{n} E(Y_k)$$

$$E(Y_n) = \frac{4}{n}\sum_{k=0}^{n-1} E(Y_k)$$

This is almost ready to be subtracted. To make it come out cleanly, we multiply through by $n+1$ (in the first equation) and n (in the second):

$$(n+1)E(Y_{n+1}) = 4\sum_{k=0}^{n} E(Y_k)$$

$$nE(Y_n) = 4\sum_{k=0}^{n-1} E(Y_k)$$

Now we can subtract. We get

$$(n+1)E(Y_{n+1}) - nE(Y_n) = 4E(Y_n)$$

so

$$(n+1)E(Y_{n+1}) = (n+4)E(Y_n)$$

Now let us divide both sides by $(n+1)(n+4)$. We get

$$\frac{E(Y_{n+1})}{n+4} = \frac{E(Y_n)}{n+1}$$

This is starting to look better. And if you look at it closely for a little while, you will see that if we now divide each side by $(n+2)(n+3)$, we will get something that is very good indeed:

$$\frac{E(Y_{n+1})}{(n+4)(n+3)(n+2)} = \frac{E(Y_n)}{(n+3)(n+2)(n+1)}$$

and so if we define

$$g(n) = \frac{E(Y_n)}{(n+3)(n+2)(n+1)}$$

then we have just derived the fact that

$$g(n+1) = g(n)$$

In other words, $g(n)$ is some constant. Call it c. Then we have

$$E(Y_n) = c(n+3)(n+2)(n+1) = O(n^3)$$

Now we're almost done. Since the function $f(x) = 2^x$ is convex, we can apply Jensen's inequality to go from our bound on $E(Y_n)$ to an even better bound on $E(X_n)$: We know that there is a constant

$C > 0$ and a number $n_0 \geq 0$ such that for all $n \geq n_0$, $E(Y_n) \leq Cn^3$. Hence for all $n \geq n_0$,

$$2^{E(X_n)} \leq E(2^{X_n}) = E(Y_n) \leq Cn^3$$

and so (taking the logarithm of both sides)

$$E(X_n) \leq \log_2 C + 3 \log_2 n = O(\log n)$$

And that's it: we have proved that the expected height of a randomly built binary search tree is $O(\log n)$.

By the way, it's important to be very precise in stating things like this. If you look back at what I've written above, you'll see that I've been careful to talk about the "average binary search tree", not the "average binary tree". And in fact, if we talk instead about average binary trees, things work out differently. For instance, what do you think the average height of a binary tree with n nodes is? Well, it's not $O(\log n)$. It's actually $O\big(\sqrt{n}\big)$, which is rather larger. This is because a binary tree might correspond to several randomly built binary search trees. For example, the sequences (2 1 3) and (2 3 1) yield the same binary tree, but when we are randomizing, we regard them as two separate cases. In fact, we build randomized binary search trees from $n!$ different permutations of the input. But there are not $n!$ binary trees with n nodes.