CS624 - Analysis of Algorithms

Binary Search Trees

October 3, 2019
A path in a graph is a sequence $v_0, v_1, v_2, ..., v_n$ 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. To make things simple, we insist that any path contain at least one edge.

Usually we write $v_0 \rightarrow v_1 \rightarrow v_2 \rightarrow ... \rightarrow v_n$ to denote a path.

A path in a graph is simple iff it contains no vertex more than once.
A loop in a graph is a path which begins and ends at the same vertex.

A loop $v_0 \rightarrow v_1 \rightarrow v_2 \rightarrow \ldots \rightarrow v_k$ 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.
A rooted tree is a tree with a distinguished vertex, which we call the root.

We will denote the root by \( r \).

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 there is a simple path from \( r \) through \( x \) to \( y \), 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.
A binary search tree is a binary tree with each node containing some data, some of which is a key. For each node $x$: If $y$ is a node on the left of $x$, $key[y] \leq key[x]$. If $y$ is a node on the right of $x$, $key[y] \geq key[x]$.

Reminder, tree traversal (any binary tree):

- Preorder – Visit the node. Then traverse its children, left-to-right.
- Inorder – Traverse the left child. Then visit the node. Then traverse the right child.
- Postorder – Traverse the children, left-to-right. Then visit the node.
Pre/In/Post-order Walks

Algorithm 1 Preorder-Tree-Walk(x)
1: if \( x \neq \text{nil} \) then
2: visit(x)
3: Preorder-Tree-Walk(x. left)
4: Preorder-Tree-Walk(x. right)
5: end if

Algorithm 2 Inorder-Tree-Walk(x)
1: if \( x \neq \text{nil} \) then
2: Inorder-Tree-Walk(x. left)
3: visit(x)
4: Inorder-Tree-Walk(x. right)
5: end if

Algorithm 3 Postorder-Tree-Walk(x)
1: if \( x \neq \text{nil} \) then
2: Postorder-Tree-Walk(x. left)
3: Postorder-Tree-Walk(x. right)
4: visit(x)
5: end if
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:

- $c =$ time for the test $x \neq nil$
- $v =$ time for the call to visit $x$
- $T(k) =$ time for the call to traverse a tree with $k$ nodes

Then certainly we have

1. $T(0) = c$ 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

2. $T(n) = c + T(k) + T(n - k - 1) + v$

We can show that $T(n) = (2c + v)n + c$
BST – Example

Nurit Haspel  
CS624 - Analysis of Algorithms
Operations on BST – Search

Recursive version

Algorithm 4 TreeSearch(x,k)
1: if x = nil or k = key[x] then
2: return x
3: end if
4: if k < key[x] then
5: return TreeSearch(left[x], k)
6: else
7: return TreeSearch(right[x], k)
8: end if

Iterative version

Algorithm 5 TreeSearch(x,k)
1: while x ≠ nil and k ≠ key[x] do
2: if k < key[x] then
3: x ← left[x]
4: else
5: x ← right[x]
6: end if
7: end while
8: return x

The running time is $O(h)$, where $h$ is the height of the tree.
<table>
<thead>
<tr>
<th>Algorithm 6 TreeMinimum($x$)</th>
<th>Algorithm 7 TreeMaximum($x$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: while $\text{left}[x] \neq \text{nil}$ do</td>
<td>1: while $\text{right}[x] \neq \text{nil}$ do</td>
</tr>
<tr>
<td>2: $x \leftarrow \text{left}[x]$</td>
<td>2: $x \leftarrow \text{right}[x]$</td>
</tr>
<tr>
<td>3: end while</td>
<td>3: end while</td>
</tr>
<tr>
<td>4: return $x$</td>
<td>4: return $x$</td>
</tr>
</tbody>
</table>

The running time is $O(h)$, where $h$ is the height of the tree.
Algorithm 8 TreeSuccessor(x)

1: if right[x] ≠ nil then
2:    return TreeMinimum(right[x])
3: end if
4: y ← p[x]
5: while y ≠ nil and x == right[y] do
6:    x ← y
7:    y ← p[x]
8: end while
9: return y

- The running time of TreeSuccessor 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 is $h$.
- What happens when we apply this procedure to the node in the figure above whose key is 20?
TreePredecessor runs in a similar fashion with a similar running time. Therefore we can state the following:

**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 $h$.***
Algorithm 9 TreeInsert(T, z)

1:  \( y \leftarrow nil \)
2:  \( x \leftarrow Root[T] \)
3:  \textbf{while} \( x \neq nil \) \textbf{do}
4:      \( y \leftarrow x \)
5:      \textbf{if} \( key[z] < key[x] \) \textbf{then}
6:          \( x \leftarrow left[x] \)
7:      \textbf{else}
8:          \( x \leftarrow right[x] \)
9:      \textbf{end if}
10: \textbf{end while}
11: \( p[z] \leftarrow y \)
12: \textbf{if} \( y == nil \) (This can only happen if the tree \( T \) was empty.) \textbf{then}
13:      \( Root[T] \leftarrow z \)
14: \textbf{else}
15:      \textbf{if} \( key[z] < key[y] \) \textbf{then}
16:          \( left[y] \leftarrow z \)
17:      \textbf{end if}
18: \textbf{else}
19:      \( right[y] \leftarrow z \)
20: \textbf{end if}
Obviously, Insert runs in $O(h)$ time on a tree of height $h$.
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 when deleting a node $d$:

1. $d$ is a leaf.
2. $d$ has one child.
3. $d$ has two children.
Operations on BST – Delete

Case I: 0 children

Case II: 1 child

Case III: 2 children
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.

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 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 $d$. What we need to do is find $d$’s successor and replace $d$ with it. Then delete $d$’s successor. Since the successor has at most one child (why?) then we revert to case I or II.
An algorithm for building a binary search tree from an array A[1..n]:

Algorithm 10 BuildBST(A)

1: Create Empty Tree
2: for i = 1⋯n do
3: \( \text{TreeInsert}(A[i]) \)
4: end for

- What is the runtime?
- Worst case – array already sorted – quadratic.
- Best case – looks like \( O(n \log n) \).
- What does it remind us of?
Modified Version of Partition

- $pivot \leftarrow A[p]$
- 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:
  $$L \mbox{ pivot } U$$
- This may require more time than the original partition but not asymptotically more.
Exercise

Show that the comparisons needed to build a BST from an array $A[1..n]$ are exactly the same comparisons needed to do quicksort on the array, using ModifiedPartition.

**Hint:** The comparisons in quicksort are against the pivot elements and the successive pivot elements are the successive elements added to the BST.
We know that the average runtime for quicksort is $\Theta(n \log n)$.

What is the “average run time” for building a BST?

It is the average over running on all possible permutations of the input array.

This is exactly what we get with randomized quicksort.

**Theorem**

The average runtime for constructing a BST is $\Theta(n \log n)$. 
The average search time in a BST is $h$, the height of a tree.

What is the average height of a BST?

We know the search time is the depth of a node.

Which is the number of comparisons we make when inserting the node into the tree.
We see that the total expected number of comparisons is $O(n \log n)$.

So the average number of comparisons is $O(\log n)$ per node.

The average cost for search in a randomly build BST is therefore $O(\log n)$.

There may be longer paths – in a linear tree the average search time is $O(n)$.

However, the average height of a randomly build BST is $O(\log n)$. 
Let $X_n$ be a random variable whose value is the height of a binary search tree on $n$ keys.

Let $P_n$ be the set of all permutations of those $n$ keys. (So the number of elements of $P_n$ is $n!$)

Let $\pi$ to denote a permutation in $P_n$. $X_n$ is actually a function on $P_n$.

Its value $X_n(\pi)$ when applied to a permutation $\pi \in P_n$ is the height of the binary search tree built from that permutation $\pi$.

We want to find $E(X_n)$, the expectation of $X_n$.

This is by definition $\sum_{\pi \in P_n} p(\pi) X_n(\pi)$ where $p(\pi)$ denotes the probability of the permutation $\pi$.

Assuming that all permutations have equal probability, $p(\pi) = \frac{1}{n!}$ for all $\pi$, and so $E(X_n) = \frac{1}{n!} \sum_{\pi \in P_n} X_n(\pi)$.
Note on Distribution

If $A$ and $B$ are two random variables on the same space $P_n$, then

$$E(A + B) = \sum_{\pi \in P_n} p(\pi) (A(\pi) + B(\pi))$$

$$= \sum_{\pi \in P_n} p(\pi) A(\pi) + \sum_{\pi \in P_n} p(\pi) B(\pi)$$

$$= E(A) + E(B)$$

Note that $\max\{A, B\}$ is also a random variable on $P_n$ – its value at $\pi$ is just $\max\{A(\pi), B(\pi)\}$. And we have the useful inequality

$$E(\max\{A, B\}) = \sum_{\pi \in P_n} p(\pi) \max\{A(\pi), B(\pi)\}$$

$$\leq \sum_{\pi \in P_n} p(\pi) (A(\pi) + B(\pi))$$

$$= E(A + B) = E(A) + E(B)$$
Consider a permutation $\pi$. The root of the tree will be the first element of $\pi$.

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.

Those elements are also chosen randomly from sets of size $k - 1$ and $n - k$ respectively, so we have

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

This is our fundamental recursion.
Since each value of \( k \) is chosen with the same probability (that probability being \( \frac{1}{n} \)), we have

\[
E(X_n) = \sum_{k=1}^{n} \frac{1}{n} E\left(1 + \max\{X_{k-1}, X_{n-k}\}\right)
\]

An effective way to estimate it would be to set \( Y_n = 2^{X_n} \).

So \( Y_n \) is itself a random variable defined on the set \( P \) whose value on the permutation \( \pi \) is \( Y_n(\pi) = 2^{X_n(\pi)} \).

At first there is no intuitive significance to the reason for doing this. It’s just that we can do better with the mathematics that way.

Compute \( E(Y_n) \) and 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.
Expected Height of a BST

\[ Y_n(\pi) = 2^{X_n(\pi)} = 2^{1+\max\{X_{k-1}(\pi), X_{n-k}(\pi)\}} \]

\[ = 2 \cdot 2^{\max\{X_{k-1}(\pi), X_{n-k}(\pi)\}} = 2 \cdot \max\{2^{X_{k-1}(\pi)}, 2^{X_{n-k}(\pi)}\} \]

\[ = 2 \cdot \max\{Y_{k-1}(\pi), Y_{n-k}(\pi)\} \]

Since each value of \( k \) is chosen with probability \( \frac{1}{n} \):

\[ E(Y_n) = \sum_{k=1}^{n} \frac{1}{n} \cdot 2E(\max\{Y_{k-1}, Y_{n-k}\}) = \frac{2}{n} \sum_{k=1}^{n} E(\max\{Y_{k-1}, Y_{n-k}\}) \]

\[ \leq \frac{2}{n} \sum_{k=1}^{n} (E(Y_{k-1}) + E(Y_{n-k})) \]

Each term is counted twice so we can simplify to get this:

\[ E(Y_n) \leq \frac{4}{n} \sum_{k=1}^{n} E(Y_{k-1}) = \frac{4}{n} \sum_{k=0}^{n-1} E(Y_{k}) \]
It is more convenient to use a strict equality, rather than an inequality. It turns out that we can assume this to be the case since we’re really only concerned with an upper bound.

**Lemma**

If $f$ and $g$ are two functions such that

\[
f(0) = g(0) \tag{1}
\]

\[
f(n) \leq \frac{4}{n} \sum_{k=0}^{n-1} f(k) \tag{2}
\]

\[
g(n) = \frac{4}{n} \sum_{k=0}^{n-1} g(k) \tag{3}
\]

then $f(k) \leq g(k)$ for all $k \geq 1$. 
Proof.

We’ll prove this by induction. The inductive hypothesis is that $f(k) \leq g(k)$ for all $k < n$. We know that this statement is true for $n = 1$ by the equation above. The inductive step is then to show that this statement remains true for $k = n$. To show this, we just compute as follows:

\[
f(n) \leq \frac{4}{n} \sum_{k=0}^{n-1} f(k) \leq \frac{4}{n} \sum_{k=0}^{n-1} g(k) = g(n)
\]
Based on this, we can assume that $E(Y_n) = \frac{4}{n} \sum_{k=0}^{n-1} E(Y_k)$.

because any upper bound we obtain for $E(Y_n)$ from this identity will also be an upper bound for the “real” $E(Y_n)$.

This is a similar trick to the one we used in deriving the average case running time of Quicksort.

We can do something very similar here, although it is a little more complicated:

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

We get rid of the denominators:

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

$nE(Y_n) = 4 \sum_{k=0}^{n-1} E(Y_k)$
Now let us subtract and get:
\[(n + 1)E(Y_{n+1}) - nE(Y_n) = 4E(Y_n)\]
\[(n + 1)E(Y_{n+1}) = (n + 4)E(Y_n)\]
Divide both sides by \((n + 1)(n + 4)\). We get \(\frac{E(Y_{n+1})}{n+4} = \frac{E(Y_n)}{n+1}\).
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 nice:
\[\frac{E(Y_{n+1})}{(n+4)(n+3)(n+2)} = \frac{E(Y_n)}{(n+3)(n+2)(n+1)}\].
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 \( E(Y_n) = c(n + 3)(n + 2)(n + 1) = O(n^3) \)

We are not done yet! We have to find \( E(X_n) \)
Expected Height of a BST

- 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 $n \geq n_0$, $2^{E(X_n)} \leq E(2^{X_n}) = E(Y_n) \leq Cn^3$.

- Taking the logarithm of both sides we get $E(X_n) \leq \log_2 C + 3 \log_2 n = O(\log n)$.

- In other words – the expected height of a randomly build binary search tree is $O(\log n)$. 