Introduction to Algorithms

George Voutsadakis[1]

[1]Mathematics and Computer Science
Lake Superior State University

LSSU Math 400

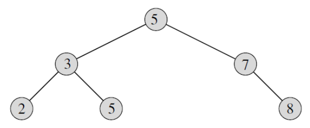
1. Binary Search Trees
    - Defining Binary Search Trees
    - Querying a Binary Search Tree
    - Insertion and Deletion
    - Randomly Built Binary Search Trees

Subsection 1

Defining Binary Search Trees

Binary Search Trees

- A **binary search tree** is organized in a binary tree:



We can represent such a tree by a linked data structure in which each node is an object.

In addition to a key and satellite data, each node contains attributes:
  - left, pointing to its left child;
  - right, pointing to its right child;
  - p, pointing to its parent.
- If a child or the parent is missing, the appropriate field contains the value NIL.
- The keys are always stored in such a way as to satisfy the **binary search tree property**: Let x be a node in a binary search tree.
  - If y is a node in the left subtree of x, then $y.\text{key} \leq x.\text{key}$.
  - If y is a node in the right subtree of x, then $y.\text{key} \geq x.\text{key}$.

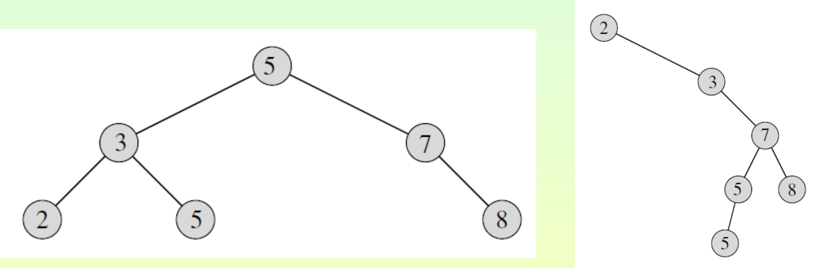
Binary Search Trees

- We can print out all the keys in a binary search tree in sorted order by a simple recursive algorithm, called an **inorder tree walk**.
- It is so named because the key of the root of a subtree is printed between the values in its left subtree and those in its right subtree.
- To use the following procedure to print all the elements in a binary search tree T, we call InorderTreeWalk(T.root):

```
InorderTreeWalk(x)
if x ≠ NIL
    InorderTreeWalk(x.left)
    print x.key
    InorderTreeWalk(x.right)
```

Illustration of Inorder Tree Walk

- The inorder tree walk prints the keys in each of the two binary search trees in the order



2, 3, 5, 5, 7, 8.

- The correctness of the algorithm follows by induction directly from the binary-search-tree property.

Time for Inorder Tree Walk

Theorem

If x is the root of an n-node subtree, then $\textsc{InorderTreeWalk}(x)$ takes $\Theta(n)$ time.

- Let $T(n)$ denote the time taken by $\textsc{InorderTreeWalk}$ when it is called on the root of an n-node subtree.
  - Since it visits all n nodes of the subtree, we have $T(n) = \Omega(n)$.
  - It remains to show that $T(n) = O(n)$.
    Since $\textsc{InorderTreeWalk}$ takes a small, constant amount of time on an empty subtree (for the test $x \neq \text{NIL}$), we have $T(0) = c$, for some constant $c > 0$.
    For $n > 0$, suppose that $\textsc{InorderTreeWalk}$ is called on a node x whose left subtree has k nodes and whose right subtree has $n - k - 1$ nodes.

Time of Inorder Tree Walk (Cont'd)

- The time to perform $\textsc{InorderTreeWalk}(x)$ is bounded by

$$T(n) \leq T(k) + T(n-k-1) + d,$$

for some constant $d > 0$ that reflects an upper bound on the time to execute the body of $\textsc{InorderTreeWalk}(x)$, exclusive of the time spent in recursive calls. We use the substitution method to show that $T(n) = \mathrm{O}(n)$ by proving that $T(n) \leq (c+d)n + c$.
  - For $n = 0$, we have $(c+d) \cdot 0 + c = c = T(0)$.
  - For $n > 0$, we have

$$\begin{array}{rcl} T(n) & \leq & T(k) + T(n-k-1) + d \\ & = & ((c+d)k + c) + ((c+d)(n-k-1) + c) + d \\ & = & (c+d)n + c - (c+d) + c + d \\ & = & (c+d)n + c. \end{array}$$

Subsection 2

Querying a Binary Search Tree

Searching a Binary Search Tree

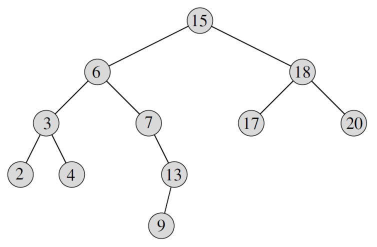
- Besides SEARCH, which searches for a key stored in a binary search tree, binary search trees can support the queries MINIMUM, MAXIMUM, SUCCESSOR and PREDECESSOR.
- We examine these operations and show how to support each one in time $\mathrm{O}(h)$ on any binary search tree of height h.
- **Searching**: Given a pointer to the root of the tree and a key k, TREESEARCH returns a pointer to a node with key k, if one exists, and NIL, otherwise.

TREESEARCH(x, k)

```
if $x ==$ NIL or $k == x$.key
    return $x$
if $k < x$.key
    return TREESEARCH($x$.left, $k$)
else return TREESEARCH($x$.right, $k$)
```

Illustration of Tree Search

- The procedure begins its search at the root and traces a path downward:



- For each node x it encounters, it compares the key k, with x.key.
  - If the two keys are equal, the search terminates.
  - If k is smaller than x.key, the search continues in the left subtree of x.
  - If k is larger than x.key, the search continues in the right subtree.
- The nodes encountered during search form a path downward from the root. Thus, the running time of TREESEARCH is $O(h)$, where h is the height of the tree.

Replacing Recursion by Iteration

- The same procedure can be written iteratively by "unrolling" the recursion into a while loop.

  On most computers, this version is more efficient:

ITERATIVETREESEARCH(x, k)

```
while $x \neq$ NIL and $k \neq x$.key
    if $k < x$.key
        $x = x$.left
    else $x = x$.right
return $x$
```

Minimum in a Binary Search Tree

- We find an element whose key is a minimum by following left child pointers from the root until we encounter a NIL.
- The following procedure returns a pointer to the minimum element in the subtree rooted at a given node x, assumed to be non-NIL.

TreeMinimum(x)

```
while $x$.left $\neq$ NIL
    $x = x$.left
return $x$
```

- The binary-search-tree property guarantees correctness:
    - If a node x has no left subtree, then since every key in the right subtree of x is at least as large as x.key, the minimum key in the subtree rooted at x is x.key.
    - If node x has a left subtree, then since no key in the right subtree is smaller than x.key and every key in the left subtree is not larger than x.key, the minimum key resides in the subtree rooted at x.left.

Maximum in a Binary Search Tree

- The pseudocode for TreeMaximum is symmetric:

TreeMaximum(x)

```
while $x$.right $\neq$ NIL
    $x = x$.right
return $x$
```

- Correctness is similar to that of TreeMinimum.
- Both of these procedures run in $O(h)$ time on a tree of height h:

  As in TreeSearch, the sequence of nodes encountered forms a simple path downward from the root.

Successor and Predecessor

- If all keys are distinct, the successor of a node x is the node with the smallest key greater than x.key.
- The structure of a binary search tree allows us to determine the successor of a node without ever comparing keys.
- The following procedure returns the successor of a node x in a binary search tree if it exists, and NIL if x has the largest key in the tree.

TreeSuccessor(x)

```
if $x$.right $\neq$ NIL
    return TreeMinimum($x$.right)
$y = x$.p
while $y \neq$ NIL and $x == y$.right
    $x = y$
    $y = y$.p
return $y$
```

Correctness and Running Time of Successor

- We break the code for TREESUCCESSOR into two cases:
  - If the right subtree of node x is nonempty, then the successor of x is just the leftmost node in x's right subtree. This we find by calling TREEMINIMUM(x.right).
  - If the right subtree of node x is empty and x has a successor y, then y is the lowest ancestor of x whose left child is also an ancestor of x. To find y, we simply go up the tree from x until we encounter a node that is the left child of its parent.
- The running time of TREESUCCESSOR (and TREEPREDECESSOR, which is symmetric) on a tree of height h is $O(h)$, since we either follow a simple path up or a simple path down the tree.

Theorem

We can implement the dynamic-set operations SEARCH, MINIMUM, MAXIMUM, SUCCESSOR and PREDECESSOR so that each one runs in $O(h)$ time on a binary search tree of height h.

Subsection 3

Insertion and Deletion

Insertion in a Binary Search Tree

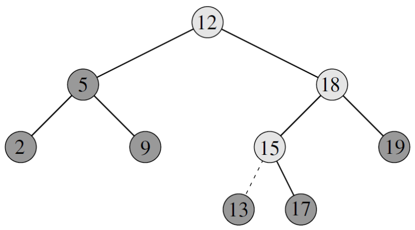
- To insert a new value v into a binary search tree T, we use the procedure TREEINSERT.
- It takes a node z for which $z.\text{key} = z.\text{left} = \text{NIL}$, and $z.\text{right} = \text{NIL}$.
- It modifies T and some of the attributes of z in such a way that it inserts z into an appropriate position in the tree.

TREEINSERT(T)

TREEINSERT(T)

```
y = NIL
x = T.root
while x ≠ NIL
    y = x
    if z.key < x.key
        x = x.left
    else x = x.right
z.p = y
if y == NIL
    T.root = z
elseif z.key < y.key
    y.left = z
else y.right = z
```

How Tree Insertion Works



- TREEINSERT begins at the root of the tree and traces a path downward.

  The pointer x traces the path, and the pointer y is maintained as the parent of x.

  After initialization, the while loop causes these two pointers to move down the tree, going left or right depending on the comparison of z.key with x.key, until x is set to NIL.

  This NIL occupies the position where we wish to place the item z.

  Lines 8-13 set the pointers that cause z to be inserted.
- Like the other primitive operations on search trees, the procedure TREEINSERT runs in $O(h)$ time on a tree of height h.

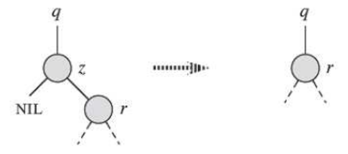
Deleting a Node

- Deleting a node z from a binary search tree consists of three cases:
  - If z has no children, we simply remove it by modifying its parent to replace z by NIL.
  - If z has a single child, we elevate the child to assume z's position in the tree by modifying z's parent to replace z by z's child.
  - If z has two children, then we must find z's successor y, which lies in z's right subtree, and have y assume z's position in the tree.
    - The rest of z's original right subtree becomes y's right subtree.
    - z's left subtree becomes y's left subtree.

  The process is tricky because it matters whether y is z's right child.

Deletion: The First Cases

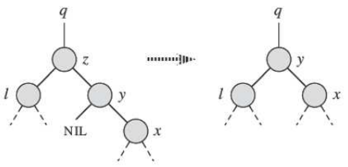
- The procedure for deleting a given node z from a binary search tree T takes as arguments pointers to T and z.
- It organizes its cases a bit differently as follows:
  - If z has no left child, then we replace z by its right child, which may or may not be NIL.

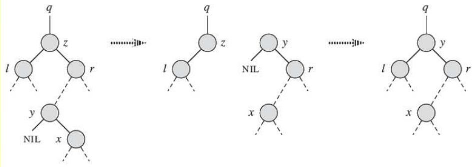


    - When z's right child is NIL, this case deals with the situation in which z has no children.
    - When z's right child is non-NIL, this case handles the situation in which z has just one child, which is its right child.
  - If z has just one child, which is its left child, then we replace z by its left child.

Deletion: The Remaining Case

- Otherwise, z has both a left and a right child. We find z's successor y, which lies in z's right subtree and has no left child. We want to splice y out of its current location and have it replace z in the tree.
  - If y is z's right child, then we replace z by y, leaving y's right child alone.
  - Otherwise, y lies within z's right subtree but is not z's right child.
    We replace y by its own right child x.
    We set y to be r's (z's right child) parent.
    Finally, we replace z by y.





Transplanting a Subtree

- To move subtrees around within the binary search tree, we define a subroutine TRANSPLANT, which replaces one subtree as a child of its parent with another subtree.
- When TRANSPLANT replaces the subtree rooted at node u with the subtree rooted at node v, node u's parent becomes node v's parent, and u's parent ends up having v as its appropriate child.

TRANSPLANT(T, u, v)

```
if $u$.p == NIL
    $T$.root = $v$
elseif $u == u$.p.left
    $u$.p.left = $v$
else $u$.p.right = $v$
if $v \neq$ NIL
    $v$.p = $u$.p
```

How Transplant Works

- Lines 1-2 handle the case in which u is the root of T.
- Otherwise, u is either a left child or a right child of its parent.
  - Lines 3-4 take care of updating u.p.left if u is a left child.
  - Line 5 updates u.p.right if u is a right child.
- We allow v to be NIL, and Lines 6-7 update v.p if v is non-NIL.
- Note that TRANSPLANT does not attempt to update v.left and v.right; doing so, or not doing so, is the responsibility of TRANSPLANT's caller.

Deletions Using Transplant

- With the TRANSPLANT procedure in hand, the procedure that deletes node z from a binary search tree T is:

TREEDELETE(T, z)

```
if $z$.left == NIL
    TRANSPLANT($T, z, z$.right)
elseif $z$.right == NIL
    TRANSPLANT($T, z, z$.left)
else $y$ = TREEMINIMUM($z$.right)
    if $y$.p $\neq z$
        TRANSPLANT($T, y, y$.right)
        $y$.right = $z$.right
        $y$.right.p = $y$
    TRANSPLANT($T, z, y$)
    $y$.left = $z$.left
    $y$.left.p = $y$
```

How Deletion Works

- The TreeDelete procedure executes the four cases as follows.
  - Lines 1-2 handle the case in which node z has no left child;
  - Lines 3-4 handle the case in which z has a left child but no right child.
  - Lines 5-12 deal with the remaining two cases, in which z has two children.

    Line 5 finds node y, which is the successor of z. Because z has a nonempty right subtree, its successor must be the node in that subtree with the smallest key, whence the call to TreeMinimum(z.right). As we noted before, y has no left child. We want to splice y out of its current location, and it should replace z in the tree.
    - If y is z's right child, then Lines 10-12 replace z as a child of its parent by y and replace y's left child by z's left child.
    - If y is not z's left child, Lines 7-9 replace y as a child of its parent by y's right child and turn z's right child into y's right child. Then Lines 10-12 replace z as a child of its parent by y and replace y's left child by z's left child.

Time Requirements

- Each line of TREEDELETE, including the calls to TRANSPLANT, takes constant time, except for the call to TREEMINIMUM in Line 5.
- Thus, TREEDELETE runs in $O(h)$ time on a tree of height h.

Theorem

We can implement the dynamic-set operations INSERT and DELETE so that each one runs in $O(h)$ time on a binary search tree of height h.

Subsection 4

Randomly Built Binary Search Trees

Randomly Built Binary Search Trees

- The height of a binary search tree varies as items are inserted and deleted.
  - If the n items are inserted in strictly increasing order, the tree will be a chain with height $n-1$.
  - Also, $h \geq \lfloor \log n \rfloor$.
- We can show that the behavior of the average case is much closer to the best case than to the worst case.
  - Unfortunately, little is known about the average height of a binary search tree when both insertion and deletion are used to create it.
  - When the tree is created by insertion alone, the analysis becomes more tractable.
- We define a **randomly built binary search tree** on n keys as one that arises from inserting the keys in random order into an initially empty tree, where each of the $n!$ permutations of the input keys is equally likely.

Expected Height of a Randomly Built Tree

Theorem

The expected height of a randomly built binary search tree on n distinct keys is $\mathrm{O}(\log n)$.

- We define three random variables that help measure the height of a randomly built binary search tree:
  - The height of a randomly built binary search on n keys is X_n;
  - The **exponential height** is $Y_n = 2^{X_n}$.
  - When we build a binary search tree on n keys, we choose one key as that of the root, and we let R_n denote the random variable that holds this key's rank within the set of n keys, i.e., R_n holds the position that this key would occupy if the set of keys were sorted. The value of R_n is equally likely to be any element of the set $\{1, 2, \ldots, n\}$.
- If $R_n = i$, then the left subtree of the root is a randomly built binary search tree on $i - 1$ keys, and the right subtree is a randomly built binary search tree on $n - i$ keys.

Expected Height (Cont'd)

- Because the height of a binary tree is 1 more than the larger of the heights of the two subtrees of the root, the exponential height of a binary tree is twice the larger of the exponential heights of the two subtrees of the root: If we know that $R_n = i$, it follows that $Y_n = 2 \cdot \max\{Y_{i-1}, Y_{n-i}\}$.

  As base cases, we have that $Y_1 = 1$, because the exponential height of a tree with 1 node is $2^0 = 1$ and, for convenience, we define $Y_0 = 0$.

  Next, define indicator random variables $Z_{n,1}, Z_{n,2}, \ldots, Z_{n,n}$, where $Z_{n,i} = \mathrm{I}\{R_n = i\}$. Because R_n is equally likely to be any element of $\{1, 2, \ldots, n\}$, it follows that $\Pr\{R_n = i\} = \frac{1}{n}$, for $i = 1, 2, \ldots, n$. Hence, we have $E[Z_{n,i}] = \frac{1}{n}$, for $i = 1, 2, \ldots, n$.

  Because exactly one value of $Z_{n,i}$ is 1 and all others are 0, $Y_n = \sum_{i=1}^{n} Z_{n,i}(2 \cdot \max(Y_{i-1}, Y_{n-i}))$.

  We shall show that $E[Y_n]$ is polynomial in n, which will ultimately imply that $E[X_n] = \mathrm{O}(\log n)$.

$Z_{n,i}$ is independent of the values of Y_{i-1} and Y_{n-i}

Claim: The indicator random variable $Z_{n,i} = \mathrm{I}\{R_n = i\}$ is independent of the values of Y_{i-1} and Y_{n-i}.

Suppose $R_n = i$ has been chosen.

- The left subtree (whose exponential height is Y_{i-1}) is randomly built on the $i-1$ keys whose ranks are less than i. This subtree is just like any other randomly built binary search tree on $i-1$ keys. Other than the number of keys it contains, this subtree's structure is not affected at all by the choice of $R_n = i$. Hence, the random variables Y_{i-1} and $Z_{n,i}$ are independent.
- Likewise, the right subtree, whose exponential height is Y_{n-i}, is randomly built on the $n-i$ keys whose ranks are greater than i. Its structure is independent of the value of R_n, and so the random variables Y_{n-i} and $Z_{n,i}$ are independent.

Obtaining a Recurrence for $E[Y_n]$

- Now we have:

$$\begin{array}{rcl} E[Y_n] & = & E\left[\sum_{i=1}^n Z_{n,i}(2\cdot\max(Y_{i-1},Y_{n-i}))\right] \\ & = & \sum_{i=1}^n E[Z_{n,i}(2\cdot\max(Y_{i-1},Y_{n-i}))] \\ & = & \sum_{i=1}^n E[Z_{n,i}]E[2\cdot\max(Y_{i-1},Y_{n-i})] \\ & = & \sum_{i=1}^n \frac{1}{n}E[2\cdot\max(Y_{i-1},Y_{n-i})] \\ & = & \dfrac{2}{n}\sum_{i=1}^n E[\max(Y_{i-1},Y_{n-i})] \\ & \leq & \dfrac{2}{n}\sum_{i=1}^n (E[Y_{i-1}]+E[Y_{n-i}]). \end{array}$$

  Since each term $E[Y_0], E[Y_1], \ldots, E[Y_{n-1}]$ appears twice in the last summation, once as $E[Y_{i-1}]$ and once as $E[Y_{n-i}]$, we have the recurrence

$$E[Y_n] \leq \frac{4}{n}\sum_{i=0}^{n-1} E[Y_i].$$

Solving the Recurrence for $E[Y_n]$

Claim: For all integers $n > 0$, the recurrence $E[Y_n] \leq \frac{4}{n} \sum_{i=0}^{n-1} E[Y_i]$ has the solution

$$E[Y_n] \leq \frac{1}{4} \binom{n+3}{3}.$$

We use the identity $\sum_{i=0}^{n-1} \binom{i+3}{3} = \binom{n+3}{4}$.

- For the base cases, we have
  - $E[Y_0] = Y_0 = 0 \leq \frac{1}{4} = \frac{1}{4}\binom{3}{3} = \frac{1}{4}\binom{0+3}{3}$;
  - $E[Y_1] = Y_1 = 1 \leq \frac{1}{4} \cdot 4 = \frac{1}{4}\binom{4}{3} = \frac{1}{4}\binom{1+3}{3}$.
- For the inductive case, we have that

$$\begin{aligned} E[Y_n] &\leq \frac{4}{n} \sum_{i=0}^{n-1} E[Y_i] \leq \frac{4}{n} \sum_{i=0}^{n-1} \frac{1}{4} \binom{i+3}{3} \\ &= \frac{1}{n} \sum_{i=0}^{n-1} \binom{i+3}{3} = \frac{1}{n} \binom{n+3}{4} \\ &= \frac{1}{n} \frac{(n+3)!}{4!(n-1)!} = \frac{1}{4} \frac{(n+3)!}{3!n!} = \frac{1}{4} \binom{n+3}{3}. \end{aligned}$$

Bounding $E[X_n]$

- We have bounded $E[Y_n]$, but our ultimate goal is to bound $E[X_n]$. Since the function $f(x) = 2^x$ is convex, we can employ Jensen's inequality: $2^{E[X_n]} \leq E[2^{X_n}] = E[Y_n]$.

$$\begin{array}{rcl} 2^{E[X_n]} & \leq & \frac{1}{4}\binom{n+3}{3} \\ & = & \dfrac{1}{4}\dfrac{(n+3)(n+2)(n+1)}{6} \\ & = & \dfrac{n^3+6n^2+11n+6}{24}. \end{array}$$

Taking logarithms of both sides gives $E[X_n] = O(\log n)$.