Notes on Binary Search Trees

Geoffrey Matthews

May 16, 2018

Search Trees

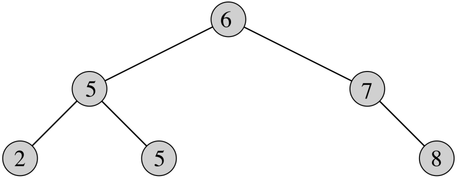
- Data structures that support many dynamic-set operations.
- Dictionaries and priority queues.
- Basic operations take time proportional to height of the tree.
  - Best case: $\Theta(\lg n)$
  - Worst case: $\Theta(n)$
- Different types of search trees:
  - binary search trees
  - red-black trees
  - B-trees
- Only assume a comparison operator on keys.
  - Hash tables assume a key we can hash well.

Binary search trees

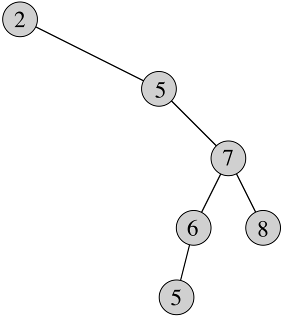
- Many dynamic-set operations in $O(h)$ time, where $h =$ height of tree.
- We represent a binary tree by a linked data structure where each node is an object.
- $T.root$ points to the root of the tree T.
- Each node contains the attributes:
  - *key* (and possibly other satellite data).
  - *left*: points to left child.
  - *right*: points to right child.
  - *p:* points to parent. $T.root.p = \text{NIL}$

Binary search tree property

- If y is in the left subtree of x, then $y.key \leq x.key$
- If y is in the right subtree of x, then $y.key \geq x.key$



(a)



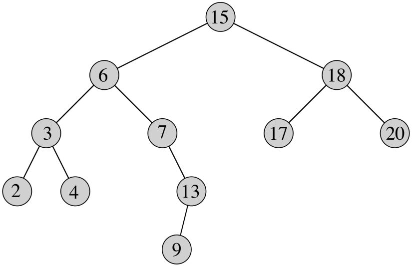
(b)

- Frequently we assume keys are unique.

Inorder-Tree-Walk

Inorder-Tree-Walk(x)

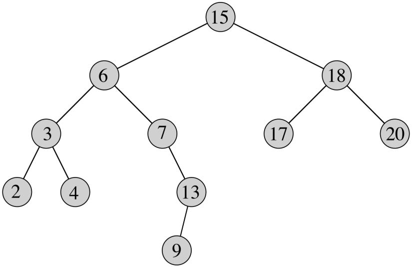
if $x \neq= \text{NIL}$
    Inorder-Tree-Walk($x.\mathit{left}$)
    print $x.\mathit{key}$
    Inorder-Tree-Walk($x.\mathit{right}$)



- Correctness follows from binary search tree property.
- Time: $\Theta(n)$, because we visit and print each node once.
  - Formal proof in book.

Tree-Search

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



- The algorithm has a single recursion on a downward path from the root.
- Time: $O(h)$ where h is the height of the tree.

Iterative version

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

```
ITERATIVE-TREE-SEARCH(x, k)
while x ≠= NIL and k ≠= x.key
    if x < x.key
        x = x.left
    else x = x.right
return x
```

- Tail recursion is easy to eliminate.

Minimum and maximum

```
TREE-MINIMUM(x)
    while x.left ≠= NIL
        x = x.left
    return x
```

```
TREE-MINIMUM-REC(x)
    if x.left == NIL
        return x
    return TREE-MINIMUM-REC(x.left)
```

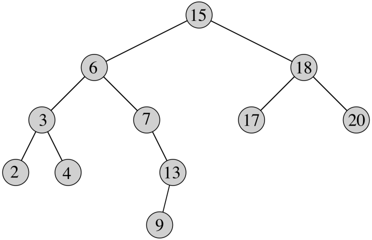
```
TREE-MAXIMUM(x)
    while x.right ≠= NIL
        x = x.right
    return x
```

```
TREE-MAXIMUM-REC(x)
    if x.right == NIL
        return x
    return TREE-MINIMUM-REC(x.right)
```

- Both procedures trace a path from root to leaf.
- $O(h)$

Successor and predecessor

- Assume all keys are distinct.
- The successor of a node x is the node y such that
  - $y.key$ is the smallest key $> x.key$.
- We can find successor without looking at keys.
- If x has the largest key, its successor is NIL.
- Two cases:
  1. If node x has a non-empty right subtree, return its minimum.
  2. Otherwise, move up the tree until the first right turn.



TREE-SUCCESSOR(x)

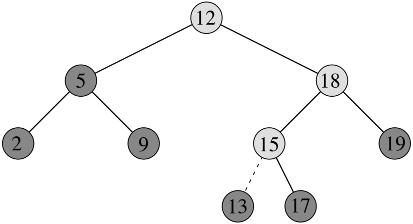
```
if x.right ≠= NIL
    return TREE-MINIMUM(x.right)
y = x.p
while y ≠= NIL and x == y.right
    x = y
    y = y.p
return y
```

- Can also move up until parent key $\geq$ child key, but that uses keys.
- TREE-PREDECESSOR similar. Both are $O(h)$.

Tree insert

TREE-INSERT(T, z)

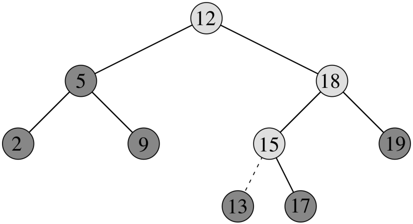
```
y = NIL
x = T.root
while x ≠ NIL
    y = x
    if z.key < x.key
        x = x.left
    else x = x.right
z.p = y
if y == NIL
    T.root = z        // tree T was empty
elseif z.key < y.key
    y.left = z
else y.right = z
```



- Trace downward path, maintaining parent pointer.
- Don't need parent pointer if we use a "null structure" for empty leaves.

Recursive tree insert

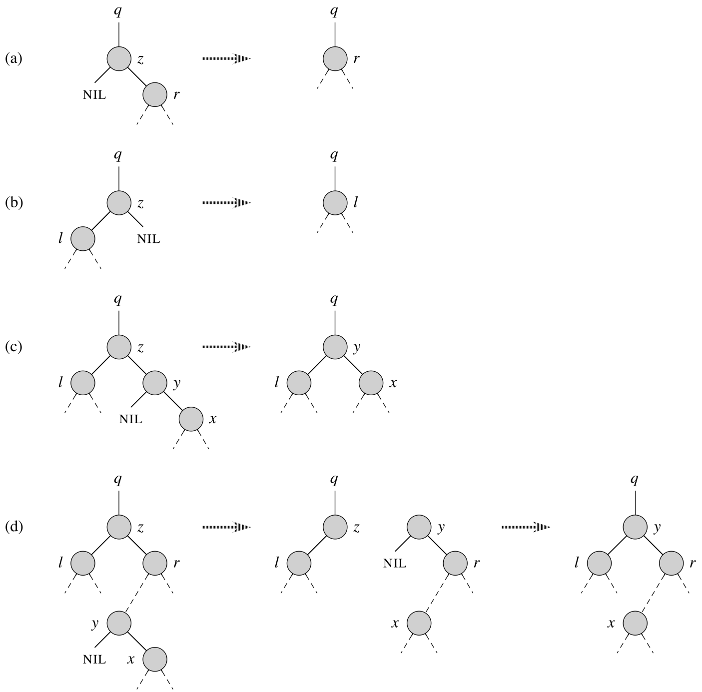
```
TREE-INSERT-REC(T, z)
  T.root = NODE-INSERT(T.root, z)

NODE-INSERT(x, z)
  if x == NIL
        return z
  z.p = x
  if z.key < x.key
        x.left = NODE-INSERT(x.left, z)
  else
        x.right = NODE-INSERT(x.right, z)
  return x
```



Deletion

To delete node z from tree T:
(a) If z has no children, just remove it.
(b) If z has just one child, then make that child take z's position in the tree.
(c) If z has two children, then

- Find z's successor y.
- y must be in z's right subtree and have no left child.
- $y.key$ must be the smallest key in z's right subtree.
- y can therefore replace z at z's position in the tree.
- Deleting y's node from the tree is easy because it has only one child.
- z's right subtree (now without y) becomes y's right subtree.
- z's left child becomes y's left child.


(a)
q
z
NIL
r
(b)
l
(c)
y
x
(d)

TRANSPLANT and TREE-DELETE

TRANSPLANT(T, u, v) replaces the subtree at u with the subtree at v.

```
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 ≠= NIL
    v.p = u.p
```

```
TREE-DELETE(T, z)
if z.left == NIL
    TRANSPLANT(T, z, z.right)
elseif z.right == NIL
    TRANSPLANT(T, z, z.left)
else
    y = TREE-MINIMUM(z.right)
    if y.p ≠= 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
```

- $O(h)$

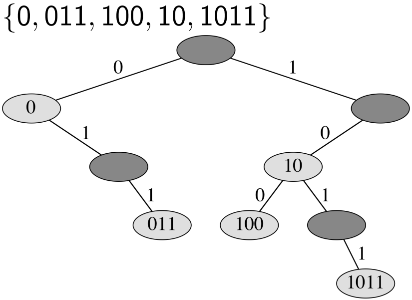
Theorem 12.4

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

Proof in text.

- Red-black trees and B-trees actively maintain a $O(\lg n)$ height in worst case.

Problem 12-2, Radix trees



- $a = a_0 a_1 \ldots a_p$ is **lexicographically less than** $b = b_0 b_1 \ldots b_q$:

  1. there exists and integer j, where $0 \leq j \leq \min(p, q)$, such that $a_i = b_i$ for all $i = 0, 1, \ldots, j-1$ and $a_j < b_j$, or
  2. $p < q$ and $a_i = b_i$ for all $i = 0, 1, \ldots, p$.

- Show that a set S of bit strings can be sorted lexicographically in $\Theta(n)$ time,
  where n is the sum of the lengths of the strings in S.