Binary Search Trees

Contents:

- What is a binary search tree?
- Querying a binary search tree
- Insertion and deletion
- Randomly built binary search trees (*)

Search Trees

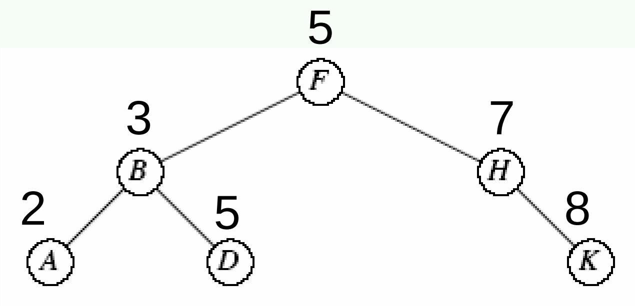
- Data structures that support many dynamic-set operations
  - SEARCH, MINIMUM, MAXIMUM, PREDECESSOR, SUCCESSOR, INSERT, DELETE
- Can be used as both a dictionary and as a priority queue
- Basic operations take time proportional to the height of the tree
- For complete binary tree with n nodes: worst case $\Theta(\lg n)$
- For linear chain of n nodes: worst case $\Theta(n)$
- Different types of search trees include binary search trees, red-black trees (Ch13), and B-trees (Ch18)
- We will cover binary search trees, tree walks, and operations on binary search trees

Binary Search Trees

Important data structure for dynamic sets

Height of a tree

- Accomplish many dynamic-set operation in O(h) time
- Use a linked data structure to represent a binary tree
  - Node : object contains key, left, right, p
  - root[T] : root of tree T, p[root[T]]=NIL
- Binary-search-tree property
  - If *y* is in left subtree of *x*, then $key[y] \leq key[x]$
  - If *y* is in right subtree of *x*, then $key[y] \geq key[x]$



Inorder-Tree-Walk

Print keys in a binary search tree in order, recursively

- Check to make sure that *x* is not NIL
- Recursively, print the keys of the nodes in *x*'s left subtree
- Print *x*'s key
- Recursively, print the keys of the nodes in *x*'s right subtree

INORDER-TREE-WALK*(x)*
if *x* $\neq$ NIL
then INORDER-TREE-WALK*(left*[*x*]*)*
print *key*[*x*]
INORDER-TREE-WALK*(right*[*x*]*)*

e.g. ABDFHK

Correctness: by induction directly from the binary-search-tree property

Time: O*(n)* time for a tree with *n* nodes,
because we visit and print each node once

Searching a Binary Search Tree

- Search for the element with key=k, TREE-SEARCH*(root* [*T*], *k)*

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

e.g. search for D

- ***Time:*** The algorithm recurses, visiting nodes on a downward path from the root. Thus, running time is $O(h)$

Minimum and Maximum

The binary-search-tree property guarantees that

- The minimum (maximum) key of a binary search tree is located at the leftmost (rightmost) node

Traverse the appropriate pointers (*left* or *right*) until NIL is reached.

TREE-MINIMUM(*x*)	TREE-MAXIMUM(*x*)
while *left*[*x*] $\neq$ NIL	**while** *right*[*x*] $\neq$ NIL
do *x* $\leftarrow$ *left*[*x*]	**do** *x* $\leftarrow$ *right*[*x*]
return *x*	**return** *x*

Time: Both procedures visit nodes that form a downward path from the root to a leaf. Both procedures run in *O(h)* time

Successor and Predecessor

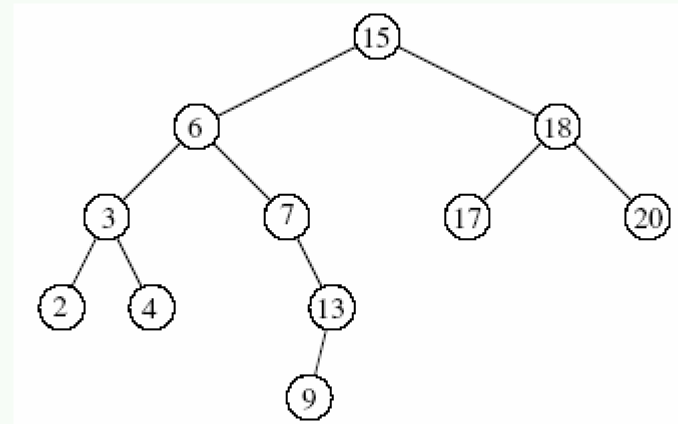
- Assuming that all keys are distinct,
  - y=successor[x] ➔ *key*[*y*] is the smallest key > *key*[*x*]
  - Y=predecessor[x] ➔ *key*[*y*] is the largest key < *key*[*x*]
- We can find *x*'s successor based entirely on the tree structure. No key comparisons are necessary
- If *x* has the largest key in the binary search tree, then we say that *x*'s successor is NIL

To identify y, the successor of x, there are two cases:

1. If node *x* has a non-empty right subtree, then y is the minimum in *x*'s right subtree (i.e. the leftmost node in the right subtree)
2. If node *x* has an empty right subtree,
   - x must NOT be in y's right subtree, and be the maximum in *y*'s left subtree
   - y is the lowest ancestor of x whose left child is also an ancestor of x
   - i.e. if we move up from x, y is the first ancestor we encounter when we go right

Searching for Successor

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



Q: the successor of the node with key value 15,6,4,17=? A: 17,7,6,18

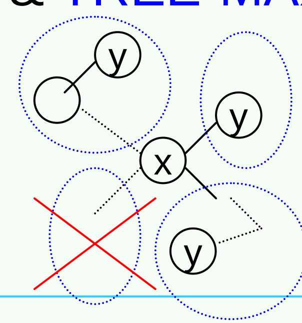
Q: the predecessor of the node with key value 15,6,4,17=? A: 13,4,3,15

- TREE-PREDECESSOR is symmetric to TREE-SUCCESSOR

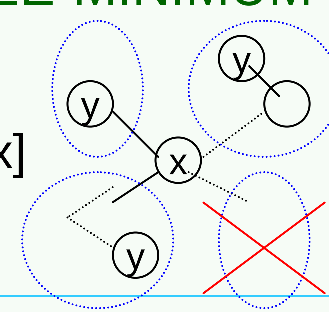
i.e. change right[x] by left[x] & TREE-MAXIMUM by TREE-MINIMUM

Time: *O(h)*

y=successor[x]



y=predessor[x]



Insertion

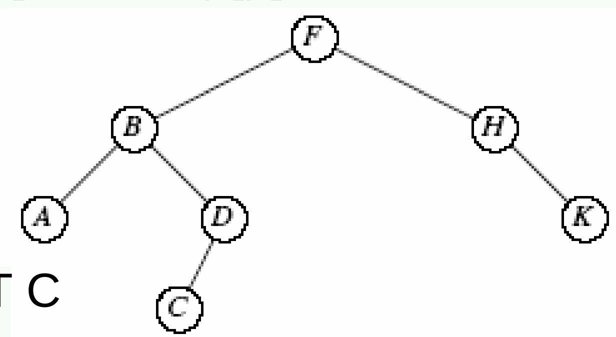
- To insert z with *key*[z] = *v*, *left*[z] = NIL, and *right*[z] = NIL

Idea: for a node x, compare key[z] and key[x]

- and y=p[x],
- If key[z]<key[x], z should be in x's left subtree
  else z is in x's right subtree
- Diving: Record y=x,
  dive on the left (or right) subtree of x
  by x=left[x] (or x=right[x]) , thus y=p[x]
- As long as x≠NIL, we keep diving as above
- Then compare key[z] and key[y]
- If key[z]<key[y], ther
- else right[y]=z

Time: *O(h)*

e.g. INSERT C



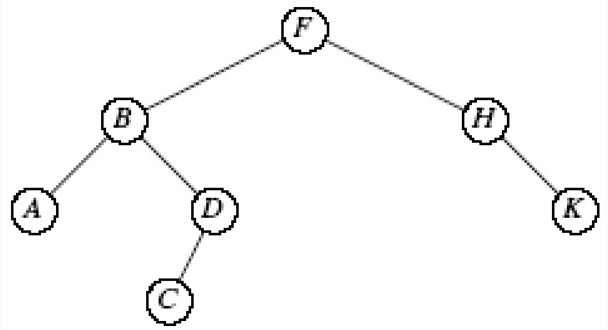
TREE-INSERT*(T, z)*
y ← NIL
x ← *root*[*T*]
while *x* ≠ NIL
  do *y* ← *x*
    if *key*[*z*] < *key*[*x*]
    then *x* ← *left*[*x*]
    else *x* ← *right*[*x*]
p[*z*] ← *y*
if *y* = NIL // T was empty
then *root*[*T*] ← *z*
else if *key*[*z*] < *key*[*y*]
  then *left*[*y*] ← *z*
  else *right*[*y*]← *z*

Deletion

TREE-DELETE(T,z) deletes node z from T

- **Case 1**: *z* has no children
  - Delete *z* by making the parent of *z* point to NIL, instead of to *z*
- **Case 2:** *z* has one child
  - Delete *z* by making the parent of *z* point to *z*'s child, instead of to *z*
- **Case 3:** *z* has two children
  - *y=successor[z] must be* in z's right subree, and *have* either no children or one child.
    (*y* is the minimum node—with no left child—in *z*'s right subtree.)
  - Delete *y* from the tree (via Case 1 or 2).
  - Replace *z*'s key and satellite data with *y*'s.

e.g. Case 1: delete K
Case 2: delete H
Case 3: delete B, swap it with C



```
TREE-DELETE(T, z)
//Determine which node y to splice out: either z or z's successor.
if left[z] = NIL or right[z] = NIL //z has one or zero child
then y ← z
else y ← TREE-SUCCESSOR(z) //z has two children
// x is set to a non-NIL child of y, or to NIL if y has no children.
if left[y] ≠ NIL
then x ← left[y]
else x ← right[y]
// y is removed from the tree by manipulating pointers of p[y] and x.
if x ≠ NIL    //y has 1 child
then p[x] ← p[y]
if p[y] = NIL //y is root
then root[T ]← x
else if y = left[p[y]] //y is not root, splice out y
        then left[p[y]] ← x
        else right[p[y]] ← x
// If it was z's successor that was spliced out, copy its data into z.
if y ≠ z
then key[z] ← key[y]
         copy y's satellite data into z
return y
```

Time: *O(h)*

Randomly built Binary Search Trees

Given a set of n distinct keys. Insert them in random order into an initially empty binary search tree

- Each of the $n!$ permutations is equally likely
- Different from assuming that every binary search tree on n keys is equally likely
- the expected height of a randomly built binary search tree is $O(\lg n)$