Lecture 17: Binary search tree (12.1-12.3)

CS303: Algorithms

Last update: February 17, 2014

1 Review

- Hasing functions: division/multiplication for open hashing
- Closed hashing: three different hashing methods (deletion might be tricky)

2 Binary search tree

The need of BST stems from the need to fast insert, search and delete elements in real time. BST also offers min, max, sorting, and search with a range (not an exact value).

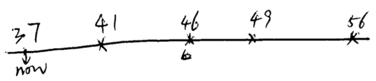
Note: hashing can only offer insert, search and delete in constant time

e.g. airport runway reservation system

- airport with single (very busy) runway (CLE has 3 runways)
- "reservation" for future landings
- when plane lands, it is removed from set of pending events
- reservation request specify "requested landing time" t. we can add t to the set if no other landings are scheduled with less than 3 minutes either way. Or else, don't schedule

e.g.

$R = \{41, 46, 49, 56\}$ – set of reserved time



request for time :

44 – not allowed (too close to 46)

53 – ok

20 – not allowed (already past)

Goal: run this system efficiently with lgn efficiency.

2.1 Proposal 1

keep R sorted as a sorted array. Then

request (t) has to go through the list first. It will take $\Theta(lgn)$ if R is sorted using binary search. However, an actual insertion has to move things around. That is roughly $\Theta(n)$

2.2 Proposal 2: BST

a binary search tree is a binary tree whose nodes contain elements of a set of orderable items, one element per node, so that for every node all elements in the left subtree are $\leq$ parent and all the elements in the right subtree are $\geq$ the element in the subtree's root.

e.g.
Build a BST

BST

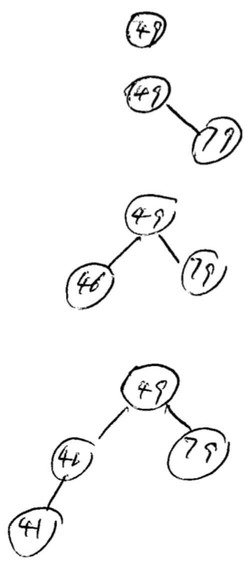
insert (49)

insert (79)

insert (46)

insert (41)

BST = nil



Note: Don't confuse BST with heap. There is an ordering in BST from left to right.

2.3 Operations on a BST with height h

1. Find min: Just go left (until you can't anymore)

```
BST-Min (x)
while x.left≠nil
      x=x.left;
return x
```

note: find max is just to go right

efficiency: $\Theta(h)$

2. search: very similar to binary search except you can't guarantee that you are dividing the tree in half at all time.

```
BST-search (x,k)
while x≠nil and k≠x.key
    if(k<x.key)
x=x.left;
    else
x=x.right
return x
```

Efficiency: $\Theta(h)$

3. print out in order: inorder traversal of the BST (inorder means left, root, right. There is also preorder and postorder)

```
Inorder-BST (x)
if x≠nil
  Inorder-BST(x.left);
  print x.key
  Inorder-bST(x.right);
```

efficiency: $\Theta(n)$ because we have to visit every node exactly once.

4. successor/predecessor: who is after or in front of me in the sorted result.

Assuming that 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 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.

There are two cases:

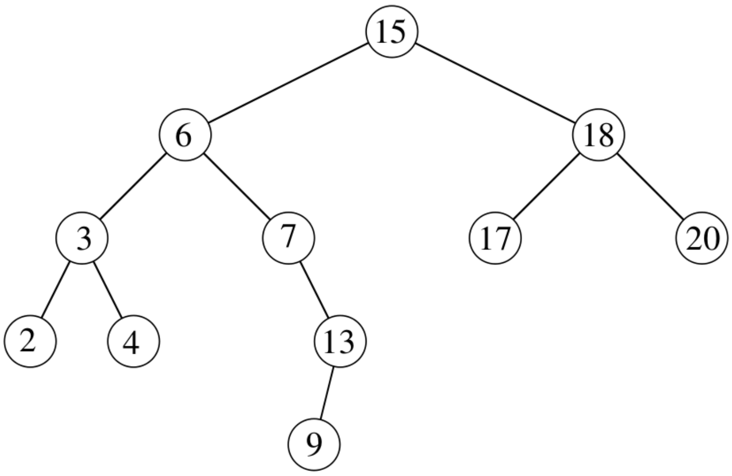
(a) If node x has a non-empty right subtree, then x's successor is the minimum in x's right subtree.

(b) If node x has an empty right subtree, notice that:
   - As long as we move to the left up the tree (move up through right children), we're visiting smaller keys.
   - x's successor y is the node that x is the predecessor of (x is the maximum in y's left subtree)

```
 BST-successor(x)
 if(x.right≠NIL)
  return BST-Min(x.right)
 y=x.p
 while y≠NIL and x==y.right
  x=y
  y=y.p
return y
```

Predecessor is almost the same.

e.g Find the successor of 15, 6, and 4. How about the predecessor of 6?



Efficiency: $\Theta(h)$

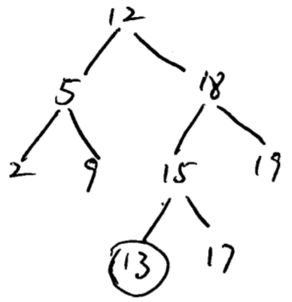
5. BST insertion: find the path to a leaf and insert at the right position

```
BST-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
else if z.key <y.key
  y.left=z
else
  y.right=z
```

Pay attention to the role of y as the trailing pointer.

e.g.

insert 13 into the following BST



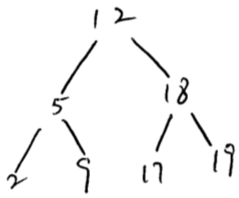
6. BST deletion three cases if we want to delete a node z

   - If z is a leaf, just delete it and mark its parent to replace z with nil
   - If z has one child, the we elevate that child to take z's position in the tree by modifying z's parent to replace z by z's parent
   - If z has two children, the we will find the successor of z and use it to replace z.

   e.g.

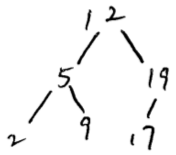
Case 1 : delete 13 in the above tree. → just remove it.

Case 2 : then delete 15 in the tree → use its child to sub 15



Case 3 : (tricky)

if we wants to delete 18, then use 19 (its successor) to sub it.



if we want to delete 12, then its successor is 17. We have to use the right child of 17 to sub 17, then sub 12 w/ 17