Algorithms and Data Structures
Lecture notes: Binary Search Trees, Cormen chapters 12

Lecturer: Michel Toulouse

Vietnamese-German University
michel.toulouse@vgu.edu.vn

2 avril 2016

Binary Search Trees

A set of elementary data structures (elements) link together by pointers such to form a binary tree data structure as a whole.

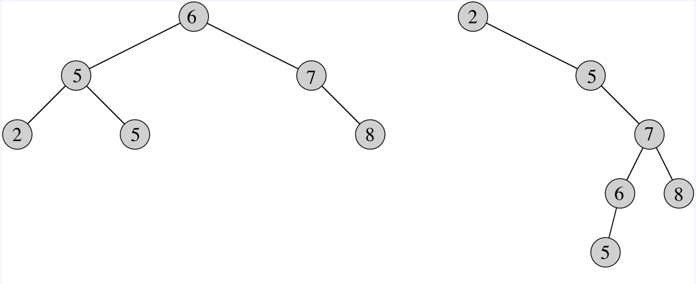
Each element has :

- *key* : an identifying field inducing a total ordering
- *left* : pointer to a left child (may be NULL)
- *right* : pointer to a right child (may be NULL)
- *p* : pointer to a parent node (NULL for root)

Binary Search Trees

Binary search tree should satisfy the following property :
$key[leftSubtree(x)] \leq x.key \leq key[rightSubtree(x)]$

Examples :

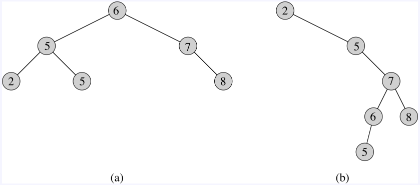


(a) (b)

Inorder tree traversal

Inorder tree traversal :

- Visits elements in sorted (increasing) order

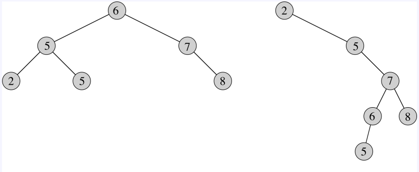


InorderTreeWalk(x)
  if $x \neq NIL$
    InorderTreeWalk(x.left) ;
    print(x.key) ;
    InorderTreeWalk(x.right) ;

Preorder tree traversal

Preorder tree traversal :

- Visits root before left and right subtrees are visited



(a) (b)

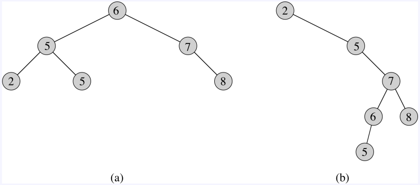
PreorderTreeWalk(x)
if $x \neq NIL$
  print(x.key) ;
  PreorderTreeWalk(x.left) ;
  PreorderTreeWalk(x.right) ;

Postorder tree traversal

Postorder tree traversal :

- Visits root after visiting left and right subtrees

PostorderTreeWalk(x)
if $x \neq NIL$
  PostorderTreeWalk(x.left) ;
  PostorderTreeWalk(x.right) ;
  print(x.key) ;



Cost of tree traversals

Tree traversal, Inorder, Postorder and Preorder cost $\Theta(n)$ where n is the number of nodes in the tree. Proof :

- $T(n)$ is the time for tree traversal called on the root of an n-node subtree
- Each traversal visits the n nodes, $T(n) \in \Omega(n)$
- Traversal of an empty subtree cost c
- For $n > 0$, $T(n) = T(k) + T(n-k-1) + d$ where d represents the cost of printing key x

Cost of tree traversals

For $n > 0$, $T(n) \leq T(k) + T(n-k-1) + d$

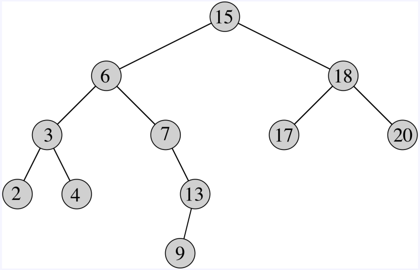
$T(n) \in O(n)$. Proof by substitution, we guess $T(n) = (c+d)n + c$ where $c+d$ is the constant amount time for each call + some constant c for the base case.

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

Operations on BSTs : Recursive Search

Search for a key k at node x (x is a pointer)

```
TreeSearch(x, k)
  if (x = NULL or k = x.key)
    return x ;
  if (k < x.key)
    return TreeSearch(x.left, k) ;
  else
    return TreeSearch(x.right, k) ;
```



Cost $\Theta(h)$ (where h is the height of the tree) since search is performed along one path in the tree

Operations on BSTs : Iterative Search

Given a key k and a pointer x to a node, the following iterative procedure returns an element with that key or NULL :

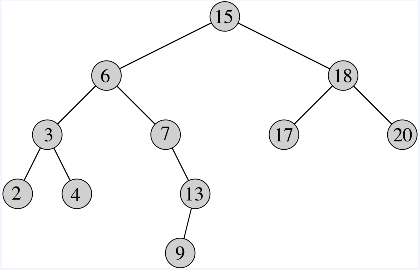
```
TreeSearch(x, k)
  while (x != NULL and k != x.key)
    if (k < x.key)
      x = x.left;
    else
      x = x.right;
  return x;
```

Same asymptotic time complexity $\Theta(h)$, but faster in practice.

Other BST Operations

Get the key with the minimum or the maximum value.

```
Tree-Minimum(x)
  while x.left ≠ NIL
    x = x.left
  return x
Tree-Maximum(x)
  while x.right ≠ NIL
    x = x.right
  return x
```



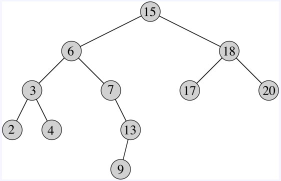
$\Theta(h)$

Successor operation

Given a node x, get its successor node in the sorted order of the keys.

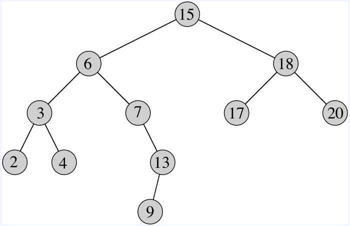
Successor :

- x has a right subtree : successor is minimum node in right subtree
- x has no right subtree : successor is first ancestor of x whose left child is also ancestor of x
  - Intuition : As long as you move to the left up the tree, you're visiting smaller nodes.



Successor Operation

Tree-Successor(x)
  if x.right $\neq$ NIL
    return Tree-Minimum(x.right)
  y = x.p
  while y $\neq$ NIL and x == y.right
    x = y ; y = y.p
  return y



$\Theta(h)$. Tree-predecessor symmetric to Tree-successor

Operations of BSTs : Insert

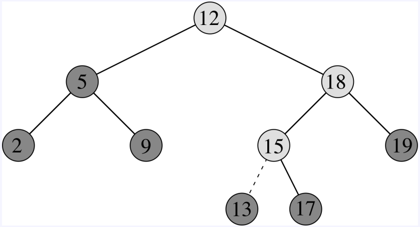
Adds an element x to the tree so that the binary search tree property continues to hold

The basic algorithm is like the search procedure above

- Insert x in place of a NIL pointer
- Use a "trailing pointer" to keep track of where you came from

Operations of BSTs : 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 /* Tree was empty */
        T.root = z
    elseif z.key < y.key
        y.left = z
    else y.right = z
```



BST Search/Insert : Running Time

The running time of TreeSearch() or Tree-Insert() is $O(h)$, where $h =$ height of tree

The height of a binary search tree in the worst case is $h = O(n)$ when tree is just a linear string of left or right children

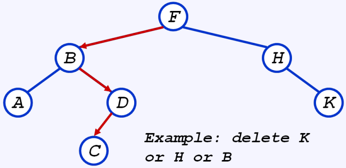
This worst case happens if keys are selected to be inserted in the tree in increasing or decreasing order

- We kept all analysis in terms of h so far for now
- We can maintain $h = O(\lg n)$ in a similar way as for quick sort by randomly picking among the keys the next one that is inserted in the tree

Operations of BSTs : Delete

The operation to delete a key z has 3 cases :

1. z has no children : Remove z
2. z has one child : Replace z by his child
3. z has two children :
   - Swap z with successor
   - Perform case 1 or 2 to delete it



Transplant routine

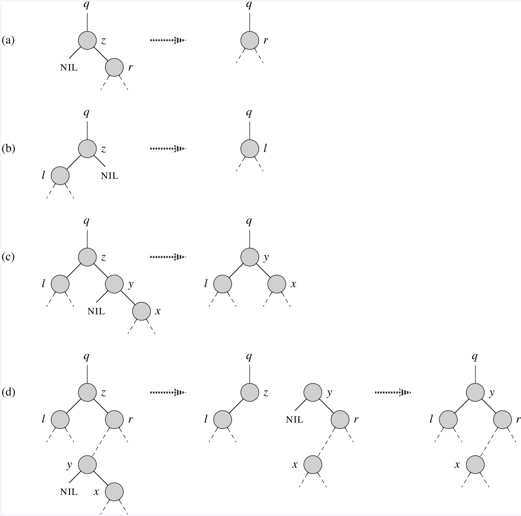
TRANSPLANT(T, u, v) replaces the subtree rooted at u by the subtree rooted at v

```
Transplant(T, u, v)
1   if u.p == NIL
2       T.root = v
3   elseif u == u.p.left
4       u.p.left = v
5   else u.p.right = v
6   if v ≠ NIL
7       v.p = u.p
```

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* for *u* left child, line 5 updates *u.p.right* for *u* a right child. TRANSPLANT does not attempt to update *v* : *left* and *v* : *right*

BST delete routine

```
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
```



Cases of procedure delete

This procedure for deleting a given node z has 3 cases :

1. If z has no left child (part (a) of fig.), replace z by its right child (which may or may not be NIL).
2. If z has just a left child (part (b) of fig.), replace z by its left child.
3. z has both a left and a right child. Find z's successor y. Want splice y out of its current location and have it replace z in the tree.
   - If y is z's right child (part (c)), 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 (part (d)). In this case, we first replace y by its own right child, and then we replace z by y.

Sorting with BST

Informal code for sorting array A of length n :

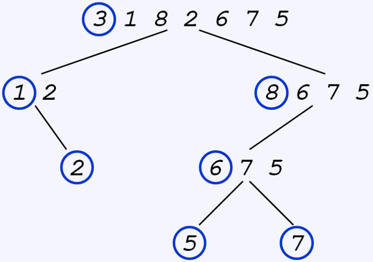
```
BSTSort(A)
  for i=1 to n
    Tree-Insert(A[i]) ;
  InorderTreeWalk(root) ;
```

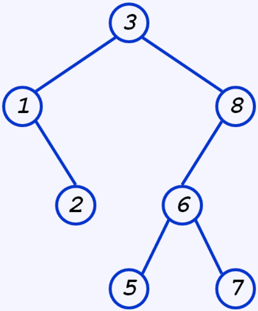
- Best case is $\Omega(n \lg n)$
- Worst case is $O(n^2)$ (when tree is just a linear string of left or right children)
- Average case is $O(n \lg n)$

BST sort : Average case analysis

Average case analysis

- It's a form of quicksort !





BST sort : Average case analysis

Same partitions are done as with quicksort, but in a different order

In previous example

- Everything was compared to 3 once
- Then those items < 3 were compared to 1 once
- Etc.

Same comparisons as quicksort, different order !

- Example : consider inserting 5

BST sort : Average case analysis

Since run time is proportional to the number of comparisons, same average time as quicksort : $O(n \lg n)$

But quicksort better than BSTSort because quicksort has

- Better constants
- Sorts in place
- Doesn't need to build data structure