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

Lecturer: Michel Toulouse

Hanoi University of Science & Technology
michel.toulouse@soict.hust.edu.vn

1er juin 2020

Outline

Linear (sequential) search

Input : An array A of n elements and the target item x. The array A does not need to be sorted.

Algorithm : Search starts at the first element of A and continues until either x is found - or entire array is searched. $O(n)$

```
void linearSearch(int A[], int size, int x)
  int i;
  for (i = 1; i <= size; i++)
    if (A[i] == x) break;
  if (i ≤ size)
    printf("The target is in the list  index = %d", i);
  else
    printf("The target is NOT in the list");
```

Binary search

Input : An array A of integer already sorted in increasing order, a value x.

Algorithm : Search x in the array A. If x not in A, then returns -1. Otherwise, returns an index such that $A[index] == x$

int *binsearch*(**int** *low*, **int** *high*, **int** *A*[], **int** *x*)
  if ($low \leq high$)
    $mid = \lfloor (low + high)/2 \rfloor$;
    if ($A[mid] == x$) **return** *mid* ;
    else if ($x <$ A[mid])
      return $binsearch(low, mid - 1, A, x)$;
    else
      return $binsearch(mid + 1, high, A, x)$;
  else return -1 ;

Binary Search Trees

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

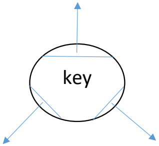
Each element has :

- *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)

Nodes

A node in C++ is declared as follow :

```
struct node{
   int key;
   node *parent;
   node *left;
   node *right;
};
```

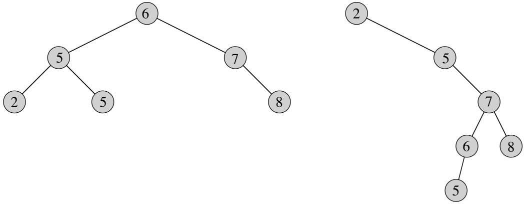


Then a pointer *r of type node is declared, the address of an object of type node is assigned to the pointer *r : node *r ; r = new node() ;

Binary Search Trees

Binary search tree should satisfy the following property :
$x.left.key \leq x.key \leq x.right.key$

Examples :



(a) (b)

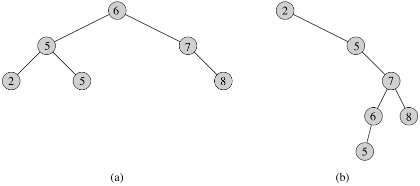
In other words, if a key y is in the left subtree of key x then $y.key \leq x.key$ and if y is in the right subtree of key x then $y.key \geq x.key$.

Inorder tree traversal

Inorder tree traversal :

- Visits elements in sorted (increasing) order

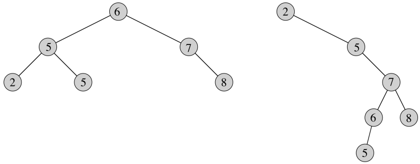
InorderTreeWalk(x)
  if $x \neq NIL$
    InorderTreeWalk(x.left) ;
    print(x.key) ;
    InorderTreeWalk(x.right) ;



Preorder tree traversal

Preorder tree traversal :

- Visits root before left and right subtrees are visited



(a) (b)

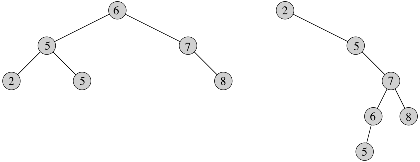
PreorderTreeWalk(x)
if $x \neq NIL$
print(x.key) ;
PreorderTreeWalk(x.left) ;
PreorderTreeWalk(x.right) ;

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) ;



(a) (b)

Cost of tree traversals

Theorem : 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. Since each traversal visits the n nodes, we have $T(n) \in \Omega(n)$. We need to show that $T(n) \in O(n)$

Traversal of an empty subtree, for the test $x \neq NIL$, i.e. $T(0)$, cost c

For $n > 0$, suppose a tree traversal is called on a node x where the left subtree has k nodes and the right subtree has $n - k - 1$ nodes

The time to perform a tree traversal from x is bounded by $T(n) \leq T(k) + T(n-k-1) + d$ where d represents the cost of printing key x

Cost of tree traversals

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

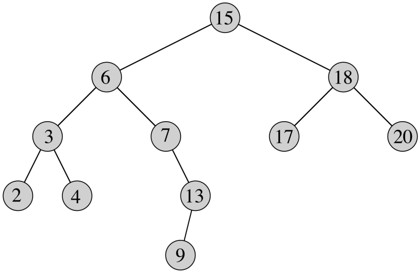
$T(n) \in O(n)$. Proof by substitution, we guess $T(n) \leq (c+d)n + c$ where $c + d$ is the constant amount time for each call + some constant c for empty subtrees.

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

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 :

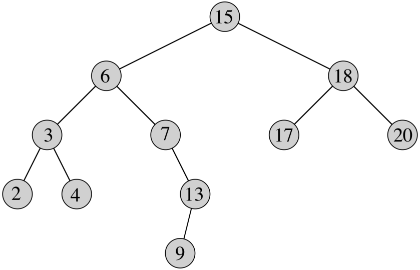
```
TreeSearch(x, k)
  while (x != NULL and k != x.key)
    if (k < x.key)
      x = x.left;
    else
      x = x.right;
  return x;
```

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

Min/max values

Get the key with the minimum or the maximum value. These algorithms return a pointer to the node with the min or max 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
```



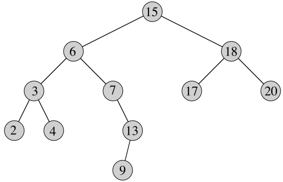
$\Theta(h)$

Successor operation

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

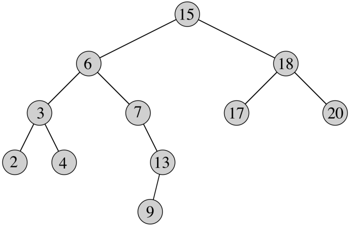
Successor :

- 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 ≠ NIL
    return Tree-Minimum(x.right)
  y = x.p
  while y ≠ NIL and x == y.right
    x = y ; y = y.p
  return y
```



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

Operations of BSTs : Insert

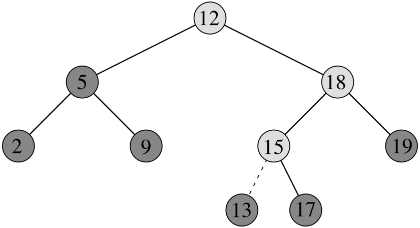
Adds an element x to the tree so that the binary search tree property continues to hold

The basic algorithm is like the TreeSearch 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

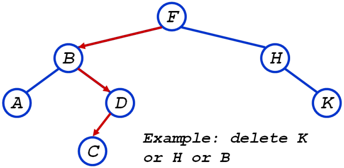
This worst case happens if keys are selected to be inserted in the tree in increasing or decreasing order

- 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

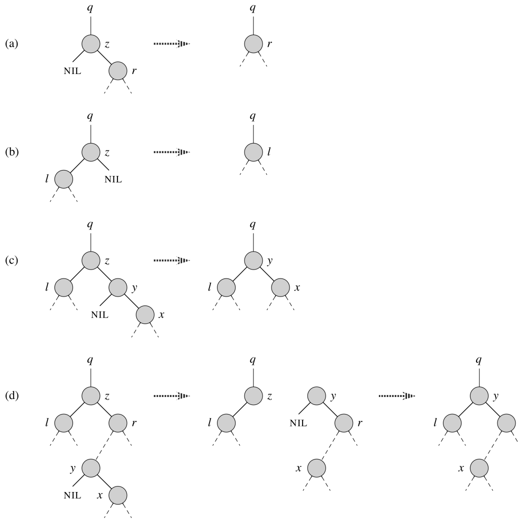
The operation to delete a key z has 3 cases :

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



BST delete : case 1 and 2

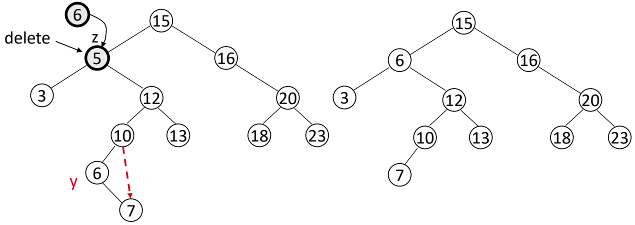
```
Tree-Delete(T, z)
  if z.left == NIL
    r.p = q ; q.right or q.left = x
  elseif z.right == NIL
    l.p = q ; q.right or q.left = l
  else
    y = Tree-minimum(z.right)
```



BST delete case 3

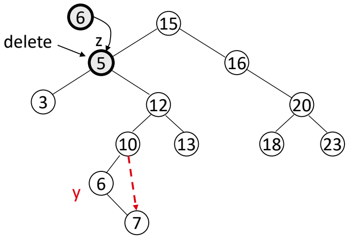
Case 3 : z has two children

- z's successor (y) is the minimum node in z's right subtree
- y has either no children or one right child (but no left child)
- Delete y from the tree (via Case 1 or 2)
- Replace z's key and satellite data with y's.



BST delete case 3

Tree-Delete(T, z)
  if z.left == NIL
    r.p = q ; q.right or q.left = x
  elseif z.right == NIL
    l.p = q ; q.right or q.left = l
  else
    y = Tree-minimum(z.right)
    if y.p $\neq$ z
      y.right.p = y.p ; y.p.left = y.right
      y.right = z.right
      y.right.p = y
    y.p = z.p
    z.p.left = y
    y.left = z.left
    y.left.p = 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) ;
```

The "for" loop executes n iterations, each iteration i calls Tree-Insert(A[i]) which is in $O(\log n)$ in best and average cases. Total cost is $n \log n$.

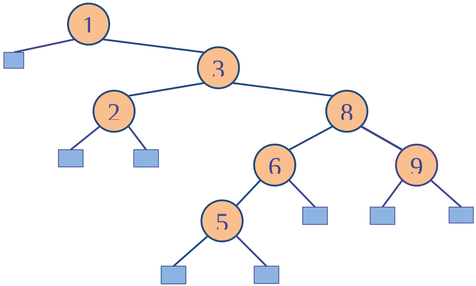
The last part, InorderTreeWalk(root), runs in $O(n)$ as already proved. Therefore $O(n) + O(n \log n) = O(n \log n)$

In the worst case we have Tree-Insert(A[i]) running in $O(n)$ (when tree is just a linear string of left or right children). Therefore $n \times O(n) = O(n^2)$

Exercises

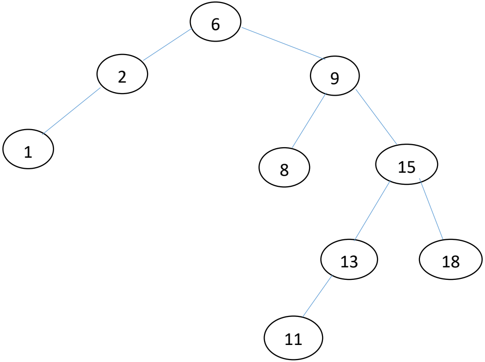
1. Draw a binary search tree for the nodes 11, 8, 14, 5, 9, 13, 16, 3, 6, 10, 17, 1
2. Draw the binary search tree from inserting the following sequence of keys : 49 75 92 24 107 26 112 101 19 2 11 25 59 16 28 95
3. For the BST of the previous question, run the *PostorderTreeWalk*(x) algorithm and list the keys in the same order as they are printed by this algorithm
4. Beginning with an empty binary search tree, what binary search tree is formed when you insert the following values in the order given ?
   4.1 W, T, N, J, E, B, A
   4.2 W, T, N, A, B, E, J
   4.3 A, B, W, J, N, T, E

Exercise 5



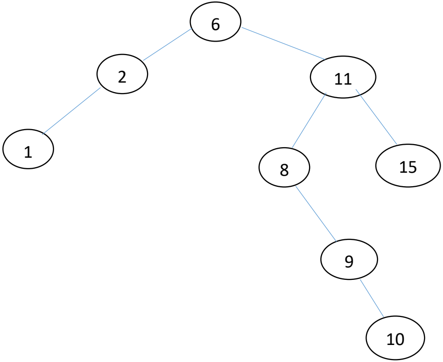
5. Considering the BST above
   5.1 Show the steps to insert two nodes with both keys as 7.
   5.2 Show the steps to delete the node with key=8 (after inserting two nodes with key=7).

Exercise 6



Considering the BST above, show the steps to delete the node with key=9

Exercise 7



Considering the BST above, show the steps to delete the node with key=6

Exercises

8. Describe an implementation of quicksort in which the comparisons to sort a set of elements are exactly the same as the comparisons to insert the elements into a binary search tree
9. Suppose that we have numbers between 1 and 1000 in a binary search tree, and we want to search for the number 363. Which of the following sequences could not be the sequence of nodes examined ?
   - 9.1 2, 252, 401, 398, 330, 344, 397, 363.
   - 9.2 924, 220, 911, 244, 898, 258, 362, 363.
   - 9.3 925, 202, 911, 240, 912, 245, 363.
   - 9.4 2, 399, 387, 219, 266, 382, 381, 278, 363.
   - 9.5 935, 278, 347, 621, 299, 392, 358, 363.
10. Write recursive versions of TREE-MINIMUM and TREE-MAXIMUM.
11. Write an iterative algorithm that executes an inorder tree traversal. (Hint : Use a stack as an auxiliary data structure)