Trees

Rossano Venturini[1]

1 **Computer Science Department, University of Pisa, Italy**
rossano.venturini@unipi.it

Notes for the course *"Competitive Programming and Contests"* at Department of Computer Science, University of Pisa.
Web page: `https://github.com/rossanoventurini/CompetitiveProgramming`

These notes sketch the content of the 4th lecture. These are dirty and non-exaustive notes that I used while lecturing. Please use them just to have a list of topics of each lecture and use the reported references to study these arguments.

The topics of this lecture are treated in any introductory book on Algorithms. You may refer to Chapters 10.4 and 12 of *Introduction to Algorithms, 3rd Edition*, Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein, The MIT Press, 2009 for a more detailed explanation.

1 Binary search tree

▶ **Problem 1** (Predecessor problem)**.** *Given a dynamic set S of keys, we would like to support the following operations.*

- Insert(S, x)*: insert the new key x in S*
- Delete(S, x)*: remove the key x from S*
- Lookup(S, x)*: establish if the key x belongs to S*
- Max(S)*: return the largest element in S*
- Min(S)*: return the smallest element in S*
- Successor(S, x)*: return the successor of x in S, i.e.,* $\max\{y \mid y \in S \ AND \ y > x\}$*;*
- Predecessor(S, x)*: return the predecessor of x in S, i.e.,* $\min\{y \mid y \in S \ AND \ y < x\}$

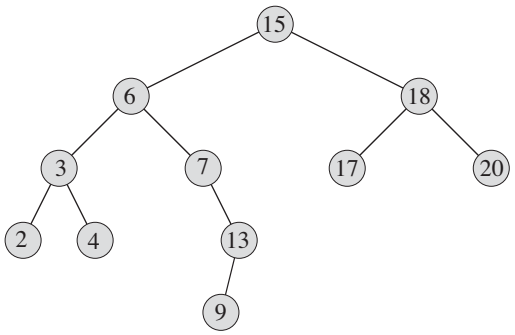
Binary search trees support these operations in time proportional to the height of the tree.

A BST is a binary tree which satisfies the following property.

BST property

Let u be a node in a BST. If v is a node in the *left* subtree of u, then $v.key \leq u.key$. Instead, if v is a node in the *right* subtree of u, then $v.key > u.key$.

The following is a BST for the set $S = \{2, 3, 4, 6, 7, 9, 13, 15, 17, 18, 20\}$.



1.1 Operations in a BST

Given a key k, it's easy to search it in the BST. We simply start from the root and move down to the leaves accordingly to the comparisons performed on the internal nodes.

Lookup(u, k)
1. if ($u == NULL$) or ($u.key == k$)
2. return u
3. if ($k < u.key$)
4. Lookup(u.left, k)
5. else
6. Lookup(u.right, k)

Where are the smallest and largest keys of S in its BST? The min is the leftmost node and the max is the rightmost one.

Min(u)
1. p = NULL
2. while ($u.left \neq NULL$)
3. p = u
4. u = u.left
5. return p

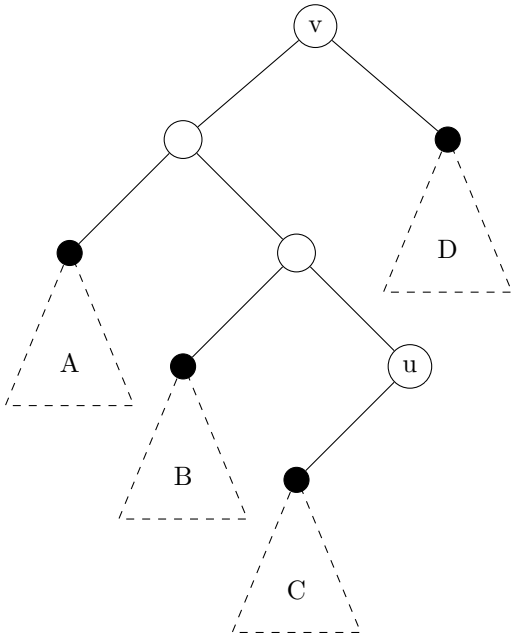
Max(u)
1. p = NULL
2. while ($u.right \neq NULL$)
3. p = u
4. u = u.right
5. return p

Finding the predecessor and the successor of a key k is more involved. Let us assume we are given a node u of the BST and our goal is to find the node that contains the successor of $u.key$.

We have two cases.

- If u has a right subtree, the successor of $u.key$ is the smallest key in that subtree. Why?
- Otherwise, the successor of u is its closest ancestor which has a left child which is also an ancestor of u. Why?

The figure below illustrates the case 2 for successor search.



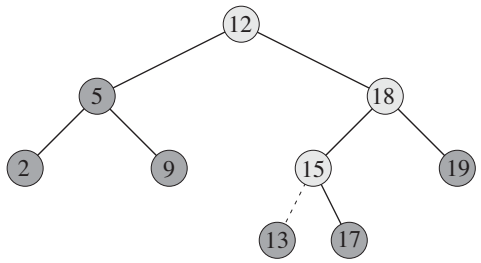
Note that 1) $u.key < v.key$, 2) keys in subtrees A, B, C are smaller than $u.key$ and 3)keys in subtree D are larger than both $u.key$ and $v.key$. This means that $v.key$ is the successor of $u.key$.

Successor(u)

1. if (u.right $\neq NULL$)
2. return Min(u.right)
3. v = u.p
4. while (($v \neq NULL$) and ($u == v.right$))
5. u = v
6. v = u.p
7. return v

Inserting and deleting keys

The following example shows the insertion of the key 13.



The obtained tree depends on the insertion order. What is the sequence that gives the worst tree? and the best one?

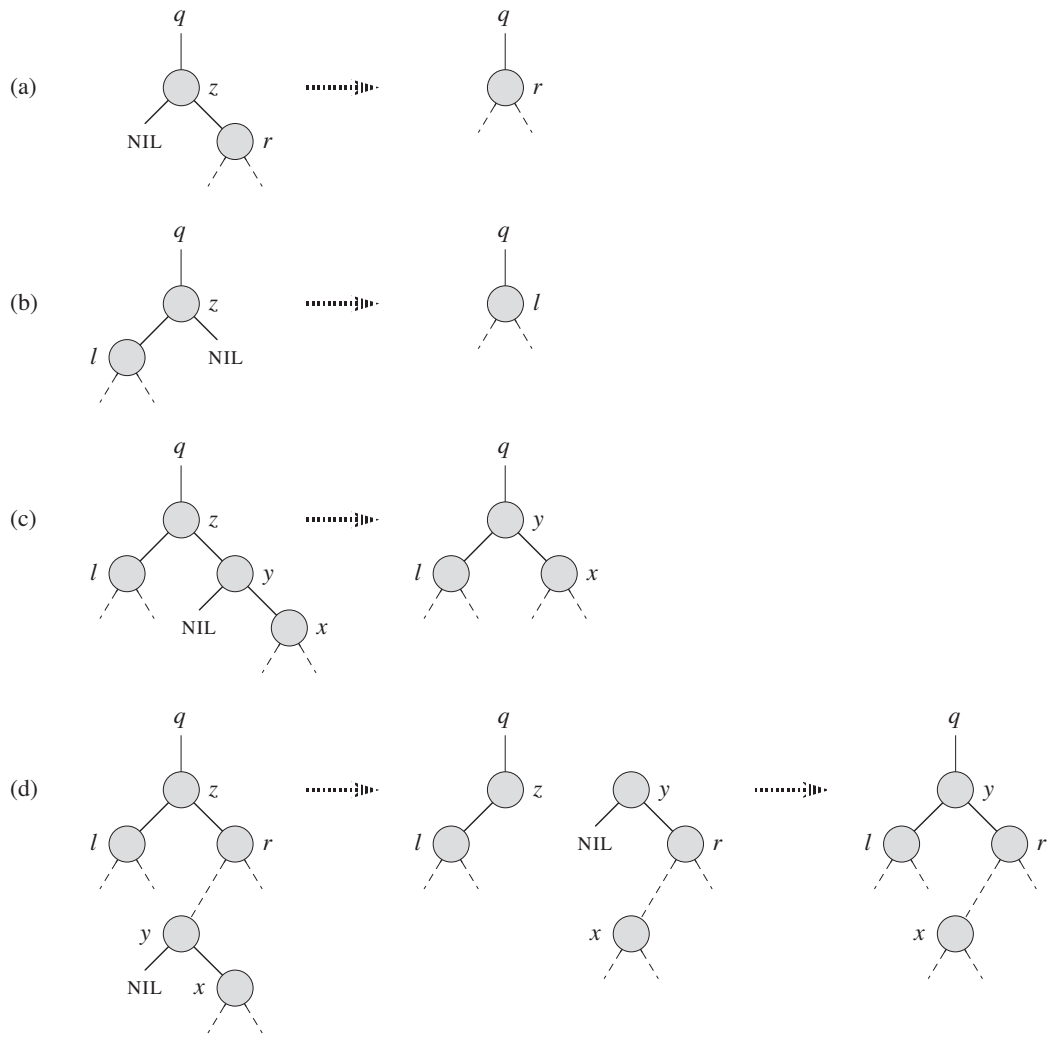
```
TREE-INSERT(T, z)
 1  y = NIL
 2  x = T.root
 3  while x ≠ NIL
 4       y = x
 5       if z.key < x.key
 6            x = x.left
 7       else x = x.right
 8  z.p = y
 9  if y == NIL
10       T.root = z          // tree T was empty
11  elseif z.key < y.key
12       y.left = z
13  else y.right = z
```

In order to delete a node z we have to distinguish 3 cases.

1. Node z is a leaf. Very easy! Remove z and change its parent.

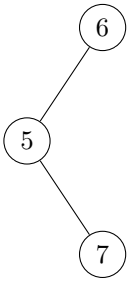
2. Node z has only one child. Easy! We remove z and we replace it with its child. See case (a) and (b) in the figure below.

3. Node z has two children. We search the successor y of z, which must be in z's right subtree. Node y replaces z in the tree. Node y is removed, but it's one of the previous (easy) cases. See cases (c) and (d) in the figure below.



▶ **Problem 2** (Is BST?). *Given a binary tree T, check if T is a BST.*

The following example shows why the algorithm that only checks children of every node is not correct.



2 Tree traversals

There are three tree traversals: In-order, Pre-order and Post-order. They differentiate from each other depending on when the current node is processed.

```
In_Order_Visit(u)
1.  if (u ≠ NULL)
2.      In_Order_Visit(u.left)
3.      print u.key
4.      In_Order_Visit(u.right)
```

Given a tree T, print its keys level by level. To solve this we have to visit the tree in BFS order.

```
BFS(root)
1.  Q = {root}
2.  while not Q.empty()
3.      n = Q.pop()
4.      print n.key
5.      if n.left != NULL
6.          Q.append(n.left)
7.      if n.right != NULL
8.          Q.append(n.right)
```

2.1 Euler tour

Sometimes is very useful to linearize the tree to obtain a sequence. The Euler tour obtains one of these linearizations.

```
Euler_Tour(u)
1.  if (u ≠ NULL)
2.      print u.key
3.      Euler_Tour(u.left)
4.      Euler_Tour(u.right)
5.      print u.key
```

The following is the Euler Tour of the previous BST.

```
15 6 3 2 2 4 4 3 7 13 9 9 13 7 6 18 17 17 20 20 18 15
```

What is an important property of this linearization?

The subtree of any node u is linearized between the first and the second occurrence of $u.key$.

▶ **Problem 3** (Path sum). *Given a tree T with values of its node, we would like support path sum queries. Given two nodes u and v with u ancestor of v, return the sum of the keys in the path from u to v.*

Solution. Use Euler tour to linearize the tree and obtain a sequence L. The only difference is that we emit $u.key$ when first visit u and $-u.key$ when we finish its visit.

```
L = 15 6 3 2 -2 4 -4 -3 7 13 9 -9 -13 -7 -6 18 17 -17 20 -20 -18 -15
```

Keep an array P that, for every node, "points" to its first and second occurrence in L. Compute prefix-sum array S of L.

```
S = 15 21 24 26 24 28 24 21 28 41 50 41 28 21 15 33 50 33 53 33 15 0
```

What's the answer of a path query?

It can actually be solved easily by storing in every node u the sum of the root-to-u path. However, the approach with Euler Tour is easier to dynamize (i.e., we can change node keys) with a data structure for Dynamic prefix sums that we will see in the next lecture.

In a problem of today's lecture you'll solve a problem similar to the one above.

▶ **Problem 4** (Maximum path sum)**.** *Given a binary tree T in which each node element contains a number. Find the maximum possible sum from one leaf node to another.*

Can you see the subtle difference in approaching this problem wrt the previous one? Is there any need to linearize the tree and use prefix-sums? No the problem is offline and you can easily solve it by visiting the tree.