FTP_Alg_Lectures 5 and 6
Binary Search Tree (BST)

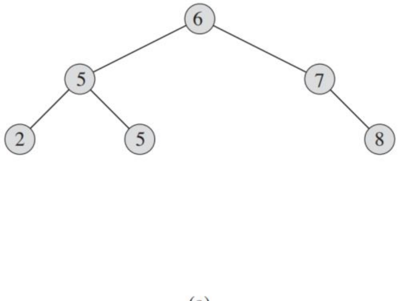
jungkyu.canci@hslu.ch

HS2024

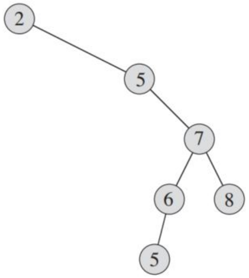
Binary Search Trees

- In this lecture, we focus on binary trees. A binary tree is a tree where each node is restricted to having at most two children.
- Basic operation on trees have $O(h)$ running time where h denotes the height of a tree (i.e. max length path from the node and a leave).
- Thus for a complete tree with n nodes, basic operations run in $\Theta(\lg n)$ in the worst case.
- The height of a linear chain tree is n, so basic operations run in $\Theta(n)$.
- A randomly built binary search tree has expected height $O(\lg n)$.

What is a binary search tree?



(a)



(b)

A binary tree is a linked data structure as in the above figure, which supports the following operations: SEARCH, MINIMUM, MAXIMUM, PREDECESSOR, SUCCESSOR, INSERT, and DELETE.

The Binary–Search–tree property

If x is a node we denote by $x.key$ its key. In a binary–search–tree the keys are stored such that they respect the following:

Binary–search–tree–property. *Let x be a node in a binary search tree. If y is a node in the left subtree of x, then $y.key \leq x.key$. If y is a node in the right subtree of x, then $y.key \geq x.key$.*

The binary-search-tree property allows us to print out all the keys in a binary search tree in sorted order using a simple recursive algorithm, called an **inorder tree walk**. This algorithm is so named because it prints the key of the root after printing the values in its left subtree and before printing those in its right subtree.
(A preorder tree walk prints the root before the values in either subtree, and a postorder tree walk prints the root after the values in its subtrees.)

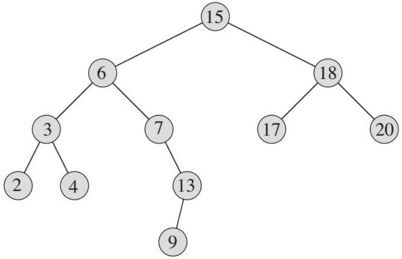
Inorder tree walk

```
INORDER-TREE-WALK($x$)
if $x \neq$ NIL
    INORDER-TREE-WALK($x.left$)
    print $x.key$
    INORDER-TREE-WALK($x.right$)
```

It takes $\Theta(n)$ time to walk an n-node binary search tree, since after the initial call, the procedure calls itself recursively exactly twice for each node in the tree, once for its left child and once for its right child.

BST: Searching

```
TREE-SEARCH($x, k$)
1  if $x$ == NIL or $k$ == $x.key$
2      return $x$
3  if $k < x.key$
4      return TREE-SEARCH($x.left, k$)
5  else return TREE-SEARCH($x.right, k$)
```



For a given pointer to a node x the procedure consider the tree with root x and provides the path from x to the node having key k. Thus the running time is $O(h)$, where h is the height of the tree (having x as root). In the example of the right, with x the root and $k = 13$ the output is

$$15 \to 6 \to 7 \to 13$$

BST: Iterative Searching

We can rewrite the previous procedure by using a while loop

```
ITERATIVE-TREE-SEARCH$(x, k)$
while $x \neq$ NIL and $k \neq x.key$
    if $k < x.key$
        $x = x.left$
    else $x = x.right$
return $x$
```

Minimum and maximum

```
TREE-MINIMUM($x$)
while $x.left \neq$ NIL
    $x = x.left$
return $x$
```

For determining a minimum it is enough to walk always left downward.

```
TREE-MAXIMUM($x$)
while $x.right \neq$ NIL
    $x = x.right$
return $x$
```

For determining a maximum it is enough to walk always right downward.

Both of these procedures run in $O(h)$ time on a tree of height h

Successor and predecessor

```
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 $y$
```

For a node in a binary search tree, we want to find its successor in the sorted order determined by an inorder tree walk. We break the code in two.

If the right subtree of x is nonempty, then the successor is leftmost node in the right subtree of x, so we call TREE–MINIMUM(x.right).
If the right subtree of x is empty and x has a successor y, then y is the lowest ancestor of x whose left child is an ancestor of x (e.g. see nodes 13 and 15 in previous tree). Prove this last statement as an exercise.

Successor and predecessor

- The running time of TREE-SUCCESSOR on a tree of height h is $O(h)$, since we either follow a simple path up the tree or follow a simple path down the tree.
- The procedure TREE-PREDECESSOR, which is symmetric to TREE-SUCCESSOR, also runs in time $O(h)$.
- Even if keys are not distinct, we define the successor and predecessor of any node x as the node returned by calls made to TREE–SUCCESSOR(x) and TREE–PREDECESSOR(x), respectively.

Insertion

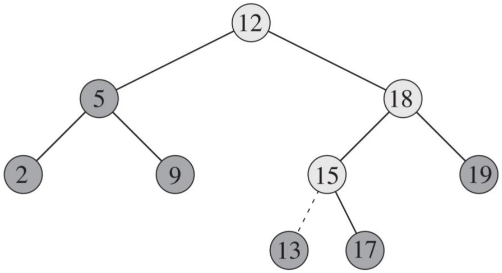
TREE–INSERT inserts the value *v* in a new node that is always placed as a new leaf. For any given value *v* this can be done maintaining the binary–search–tree–property.

```
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
```

The procedure maintains the **trailing pointer** *y* as the parent of *x*. The while loop in lines 3–7 causes these two pointers to move down, going left or right depending on the comparison of *z.key* with *x.key*, until *x* becomes NIL in the "corrected" position.

We need the trailing pointer y, because by the time we find the NIL where *z* belongs, the search has proceeded one step beyond the node that needs to be changed. Lines 8–13 set the pointers that cause *z* to be inserted.

Insertion: example



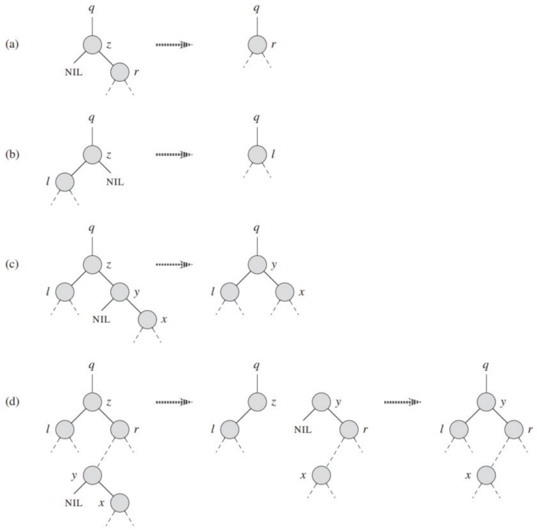
The above figure show the insertion of the node with key 13. Lightly shaded nodes indicate the simple path from the root down to the position where the item is inserted. The dashed line indicates the link in the tree that is added to insert the item.

Deletion

For deleting a node z we have to consider three basic cases:

- z is a leaf (so no children). We simply replace in z's parent the correct child position as NIL instead of z.
- z has only one child. We elevate z's child to take z's position and the z's parent has new child z's child.
- z has two children. We find the successor of z, let's denote it by y. Recall that y must be in the right z's subtree. We put y in the z's position. Note that it matter whether y is the z's right child.

Deletion


(a)
q
z
NIL
r
q
r
(b)
q
z
l
NIL
q
l
(c)
q
z
l
y
NIL
x
q
y
l
x
(d)
q
z
l
r
y
NIL
x
q
z
l
y
NIL
r
x
q
y
l
r
x

Deletion: Transplant

TRANSPLANT replaces one subtree as a child of its parent with another subtree. When TRANSPLANT replaces the subtree rooted at node *u* with the subtree rooted at node *v*, node u's parent becomes node *v*'s parent, and *u*'s parent ends up having *v* as its appropriate child.

```
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
```

1-2 considers *u* as the root. If not *u* is a right child or left child of its parent, so see lines 3-5. *v* is allowed to be NIL. Lines 6-7 update $v.p$ if *v* is not NIL.

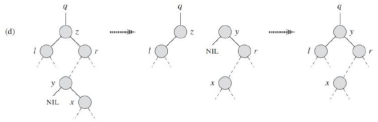
TRANSPLANT does not upadate $v.\mathit{left}$ and $v.\mathit{right}$. This will be done by the code, which calls TRANSPLANT.

TREE–DELETE

```
TREE-DELETE($T, z$)
 1  if $z.left$ == NIL
 2      TRANSPLANT($T, z, z.right$)
 3  elseif $z.right$ == NIL
 4      TRANSPLANT($T, z, z.left$)
 5  else $y$ = TREE-MINIMUM($z.right$)
 6      if $y.p \neq z$
 7          TRANSPLANT($T, y, y.right$)
 8          $y.right = z.right$
 9          $y.right.p = y$
10      TRANSPLANT($T, z, y$)
11      $y.left = z.left$
12      $y.left.p = y$
```

1-2 consider the case where z has no left child. 3-4 the case where there is left child but not right child. 5-12 consider the two remaining cases where z has two children. In 5, y is the successor of z, which has no left child. We want to replace z with y.

If y is z's right child, 10-12 update $y.left$ and new $y.left.p$, the right subtree does not change. If y is not z's (right) child, 7–9 transplant y and with its $y.right$ (denote above with x). Line 8 put y as root of the right tree (see figure below in the middle) and then transplant the new tree rooted in y to z.



References

The content of these slides are taken from
Introduction to Algorithm Third Edition
by T.H. Cormen, C.E. Leiserson, R.L. Rivest and C. Stein
Sections: 12.1, 12.2 and 12.3