Fundamental Algorithms

Lecture #6

@cs.utcluj.ro

Cluj-Napoca

Agenda

- Trees
  - Basic operations
    - walk, search, insert, delete – review
    - walk iterative
    - min, max, pred, succ
  - Special types
    - Balanced trees
      - PBT (seminar #4)
      - AVL (SDA class + review here)
      - Red-Black (next lecture)
    - Augmented Trees
      - Order-statistic trees

BST – walk, search, insert

- **Walk**
  - pre/in/post-orders **O(n)** if O(1) outside recursive calls
  - else apply master theorem
- **Search**
  - **O(n)** for BT
  - **O(h)** for B**S**T, h $\in$ **[lgn, n]**
  - **O(lgn)** for **balanced** BST
- **Insert**
  - **Search** for it and reach a leaf/1-child node (parent for the new node)
  - Insert as **leaf** always, as child of the given leaf/1-child node

Tree traversal – iterative version

- **Any** recursive implementation can be rewritten iteratively
  - Using a stack is one possible approach
  - Without a stack?
    - if "keep track of the calls" + need parent link in the structure!
- IDEA: should remember where you are coming from (OR one pointer behind to "model" that), so:
  - Either: keep a "counter" to tell how many times you reach the node and act accordingly …
  - Or: keep a pointer behind the current node (i.e. the previously visited node)

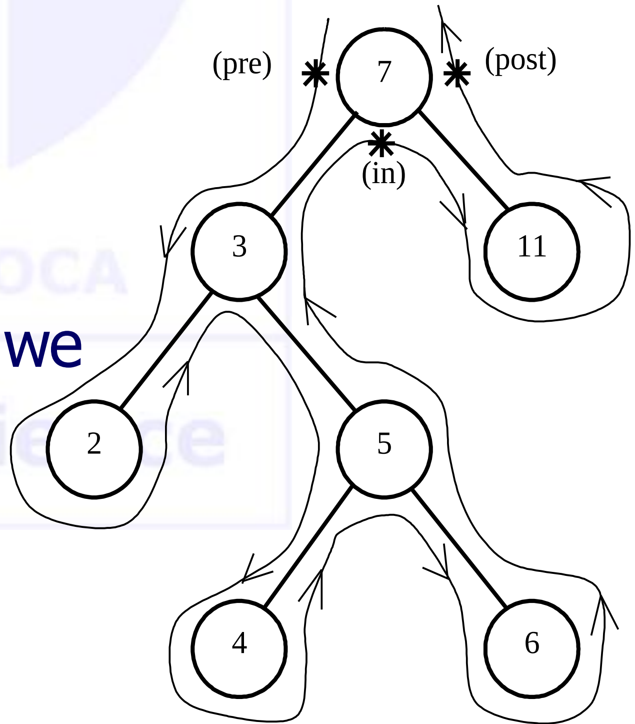
BST – walk iterative

- Non-recursive traversal
  - No additional memory
  - No explicit stack
- Needs parent pointer in the structure
- Should keep track of the advancement (WHERE we are on the track)
  - Top -> down (pre)
  - Left ->root (in)
  - Right -> root (post)
- We can keep track of the DIRECTION we

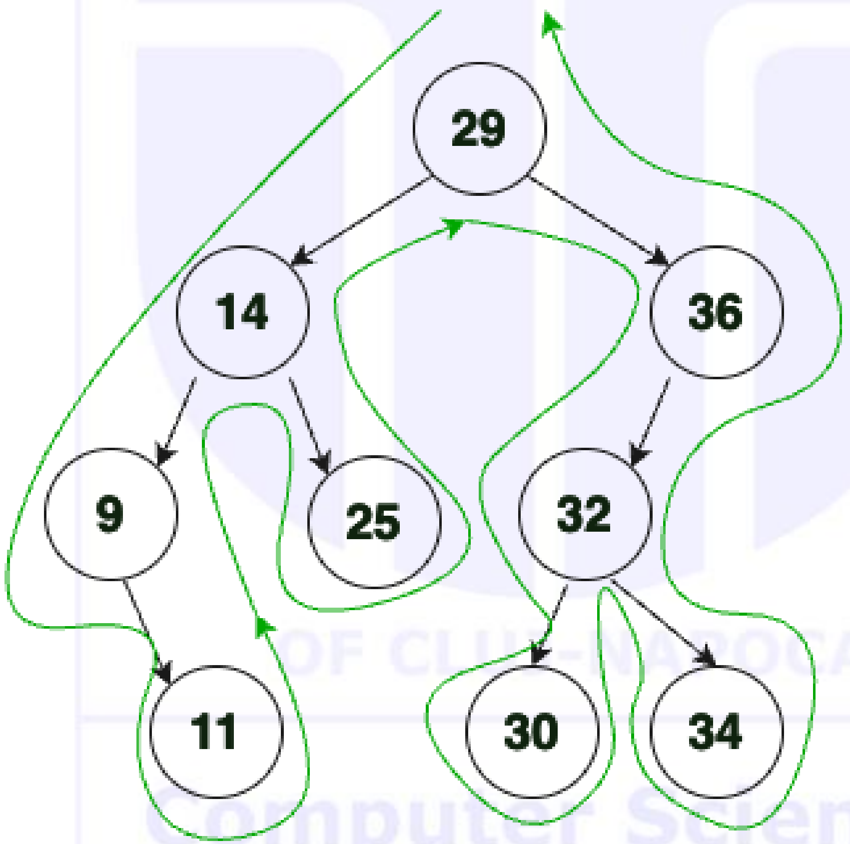
are: top->down =#1

left -> root =#2

right -> root =#3



BST – walk iterative - contd

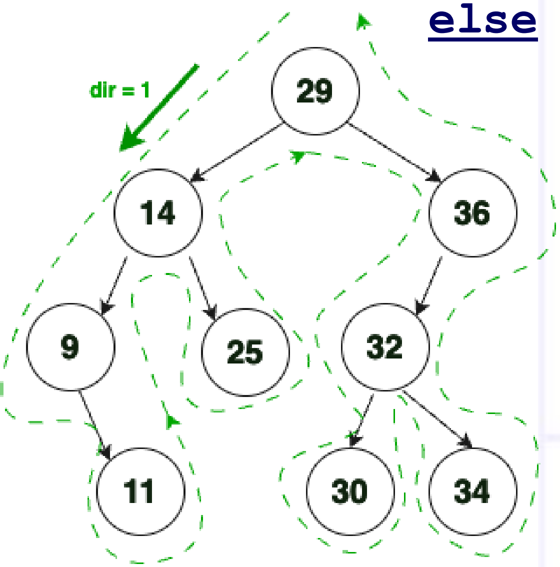


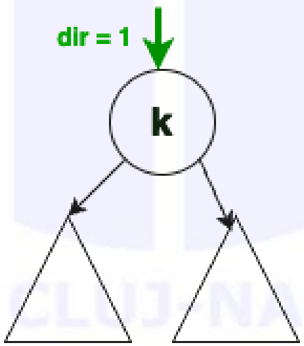
Tree traversal – iterative version – contd.

```
d<-1                    //initialize d to 1 before you call it on your Tree (on main)
printTree(T)
node<-root[T]
repeat
if d = 1   then         //without else branch &        //here print in preord
           if left[node]!=NIL
                        then node <- left[node]        //advance to the left with direction still 1
                        else d<-2 //set dir to 2 as you meet node second time => advance to the right
if d = 2   then         //without else branch &        //here print in inord
           if right[node]!=NIL
                        then node <- right[node]; d<-1//advance to the right=> FIRST time
                        else d<-3          //set dir to 3 as you meet node 3rd time => advance to parent
 if d = 3  then         //without else branch &        //here print in postord
           if parent[node]!=NIL                        // we are not done;
                        then      if node = left[parent[node]]   //check the dir we are coming from
                                  then d<-2                             //else remains on 3
                                  node<-parent[node]                    //advance to the parent
until (node=root[T] and d=3)                           // node=root[T] means parent[root]=nil
```

BST – walk iterative - contd

```
printTree(T)
node<-root[T]
repeat
if d = 1 then //without else branch && //here print in preord
        if left[node]!=NIL
              then node <- left[node] //advance to the left
              else d<-2
```

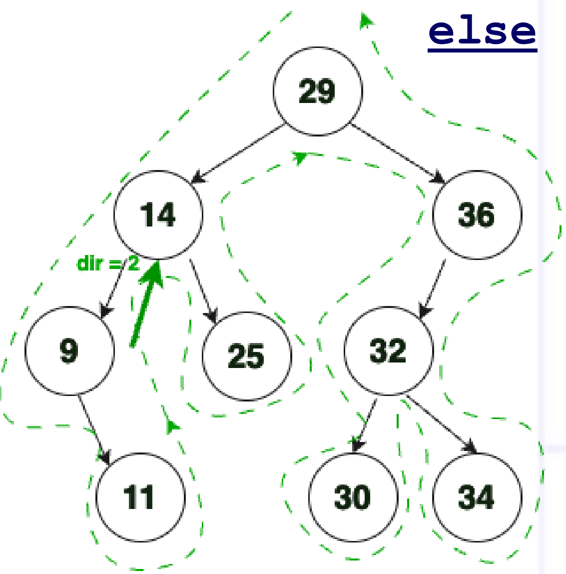


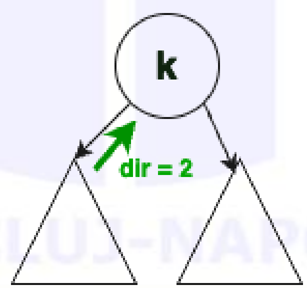



```
until (node=root[T] and d=3)//node=root[T] means parent[root]=nil
```

BST – walk iterative - contd

```
printTree(T)
node<-root[T]
repeat
if d = 2 then //without else branch && //here print in inord
        if right[node]!=NIL
              then node <- right[node]; d<-1//advance to the right
              else d<-3
```

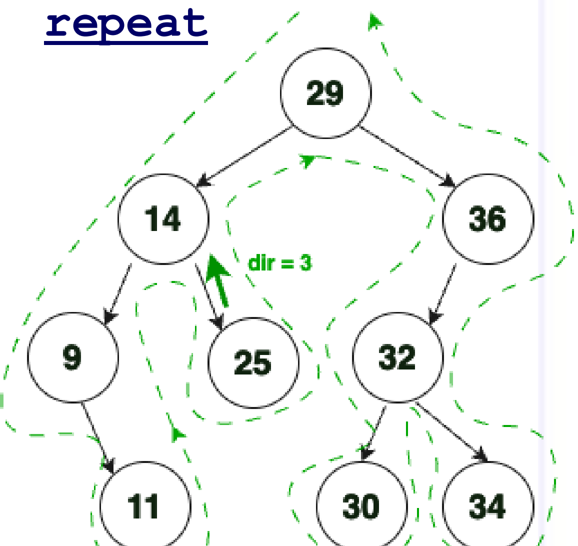


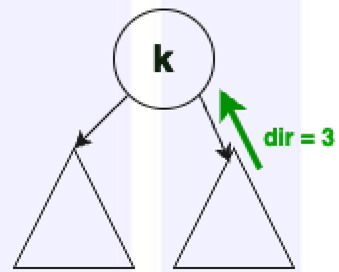



```
until (node=root[T] and d=3)//node=root[T] means parent[root]=nil
```

BST – walk iterative - contd

```
printTree(T)
node<-root[T]
repeat
```



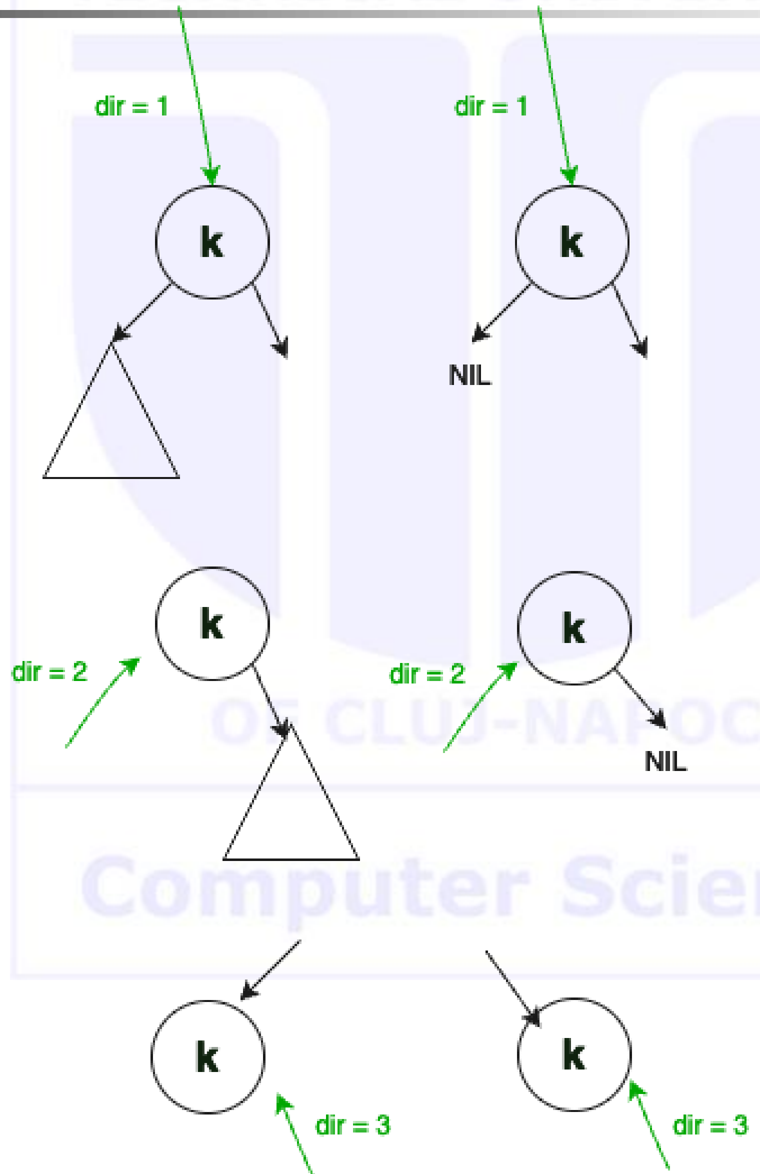



```
if d = 3 then //without else branch && //here print in postord
        if parent[node]!=NIL
              then
              if node = left[parent[node]]//check the dir we are coming from
                      then d<-2
                     node<-parent[node]    //advance to the parent
until (node=root[T] and d=3)//node=root[T] means parent[root]=nil
```

BST – walk iterative - contd

```
printTree(T)
node<-root[T]
repeat
if d = 1 then //without else branch && //here print in preord
        if left[node]!=NIL
              then node <- left[node] //advance to the left
              else d<-2
if d = 2 then //without else branch && //here print in inord
        if right[node]!=NIL
              then node <- right[node]; d<-1//advance to the right
              else d<-3
if d = 3 then //without else branch && //here print in postord
        if parent[node]!=NIL
              then
              if node = left[parent[node]]//check the dir we are coming from
                      then d<-2
              node<-parent[node]   //advance to the parent
until (node=root[T] and d=3)//node=root[T] means parent[root]=nil
```

BST – walk iterative - contd



BST - delete

- Remove the node
- Cases:
  - Leaf – remove it
  - 1-child node – link parent with the only child
  - 2-children nodes
    - Chain the tree (fast, unbalances the tree)
    - Replace the node with an appropriate one (content of predecessor/successor), and remove (the location of) that one (same time, better balance)

BST – delete - code

```
tree_delete(T,z)                //z=node to delete; y physically deleted
if left[z]=nil or right[z]=nil
      then y<-z                 //Case 1 OR 2; z has at most 1 child => del z
      else y<-tree_successor(z) //find replacement=min(right)
if left[y]<>nil                 //we are in Case 2; y is a single child node
      then x<-left[y]           //y has no child to the right; x=y's child
      else  x<-right[y]               //case 2 or 3. Why?
if x<>nil                       //y is not a leaf;
      then p[x]<-p[y]           // y's child redirected to y's parent = x's parent
   //becomes the former single (why?) grandparent
if p[y]=nil                     //means y were the root
      then root[T]<-x           //y's child becomes the new root
      else  if y=left[p[y]] //link y's parent to x which becomes its child
                  then left[p[y]]<-x
                  else right[p[y]]<-x
return[y]       //outside the procedure: copy y's info into z; dealloc y
```

BST – delete - eval

- Find node to delete O(h)
- Find successor/predecessor O(h)
- BUT:
  - if finding node to delete takes O(h) => the node is a leaf => case 1 => no succ needed
  - if node to delete not a leaf, succ searched from that place down => find node+find succ=O(h)
- Delete takes only O(h)

Find-min/max **O(h)**

- Root's leftmost/rightmost leaf in the tree rooted at x;

```
find_tree_min(x)                    //x=root;
while left[x]<>nil
do   x<-left[x]
return x
```

Q: what if left[x]=nil?

```
find_tree_max(x)                    //x=root;
while right[x]<>nil
do   x<-right[x]
return x
```

Find-pred/succ

- pred = max in the left subtree =>

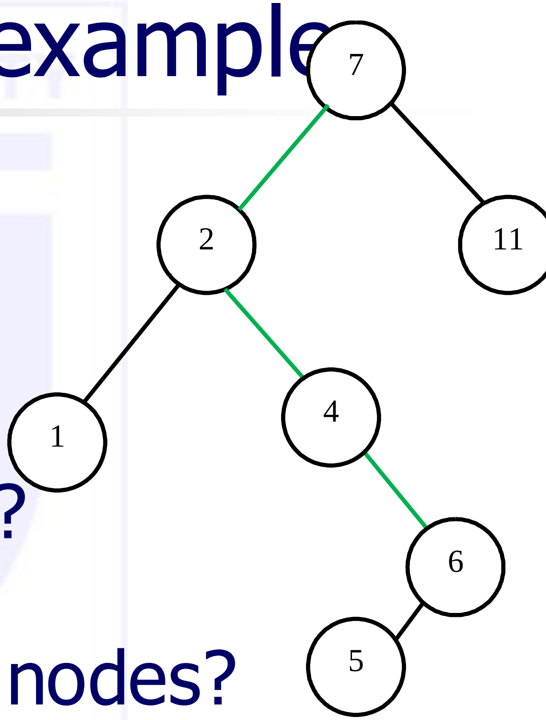
  find_tree_max(left[x])

- succ=min in the right subtree

  find_tree_min(right[x])

- Any other situation possible?
  - What if the node has no left/right subtree? Possible?
  - It has no pred/succ?
  - Not necessarily: counterexample!

Find-pred/succ- counterexample



- 6 has no right child.
- It means it has no successor?
  - False! 7 is its successor!
- 5 has no left/right child.
- It means it has no predecessor/succ?
  - False! 4 is its predecessor/6 its pred!
- How can we find pred/succ for such nodes?

(identify the property such nodes posses)

succ=lowest level ancestor whose left child is an ancestor as well

pred=lowest level ancestor whose right child is an ancestor as well

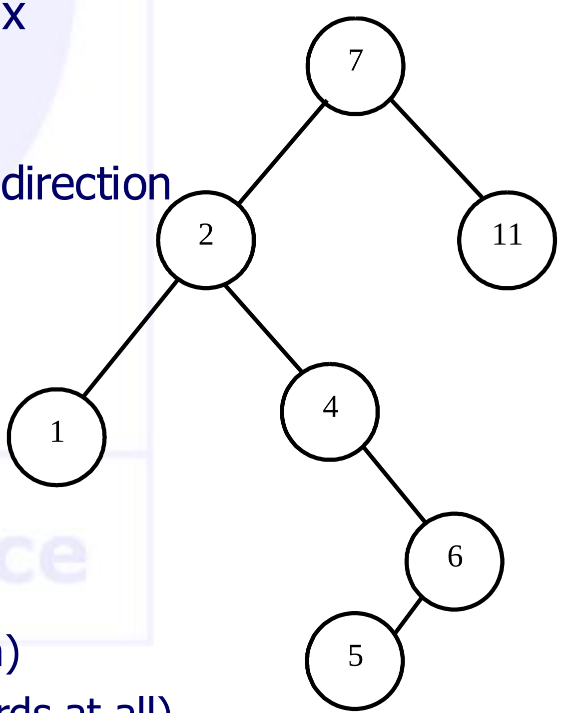
Determine (for succ) a triangle:

node-upwards while on a right child link

the first time the node is a left child= it is the succ node

Find-succ-code

```
find_tree_successor(x)          //returns x's successor
  if right[x]<>nil    //regular case; the succ belongs to the same subtree
     then return find_tree_min(right[x])
  y<-p[x]         //y keeps a pointer 1 level above x
while y<>nil and x=right[y]
// as long as we haven't reached the root  and not changed the direction
// along the upwards path, go upwards 1 level
do    x<-y
      y<-p[y]
return y
```



Note: 2's successor is 4 (in find_tree_min)

6's successor is 7 (take **while twice** and change direction)

5's successor is 6 (0 while, exit while without going upwards at all)

Find-succ **O(h)**

- Cases:
  - ***find_tree_min(right[x])***, worst case: x=root, succ lowest leaf => **O(h)**
  - x has no right child; worst case: x=leaf on the lowest level, direction changes at the root level=> succ root of the tree => **O(h)**
- ***find_tree_successor*** **O(h)**
- Find the predecessor is symmetric (change right with left and min with max) - **Homework**

BST-eval

- Theorem: All operations in a BST (except traversal) take O(h)
- Adv: faster than on lists!
- Limitation: h? Worst case h=n (why?) Therefore, no improvement at all!
- Enhancement?
  - **Balanced trees**!

Balanced trees

- Augmented BST to keep the height under control
- No matter the balance type, the height is proportional to lgn (**c·lgn**, with c≥1, but c a SMALL CONSTANT)
- The best possible balanced trees – PBT (perfect balanced trees) – seminar #4
- many other possibilities (for balance)

Balanced trees - PBT

- Perfect Balanced Trees = BST + balance (nodes rel)
- Any subtree of a PBT is a PBT as well!
- Balance refers to nb of nodes, not to heights
- $b=n_R-n_L \in \{-1, 0, 1\}$
- $h=\lg n$
- Insert O(n): **ins** as in regular BST O(h)=O(lgn)
  but requires n rotations to rebalance => O(n)
- Delete O(n): **del** as for regular BST O(h)=O(lgn)
  but requires n rotations => O(n)
- Best h property; difficult (costly) to maintain
- Discussion: when should be use PBTs?

Balanced trees - AVL

- AVL = BST + balance (height related)
- Any subtree of an AVL tree is an AVL tree as well!
- (AVL=Adelson-Velskii, Landis)
- Balance on height $b=h_R-h_L \in \{-1, 0, 1\}$
- PBTs are AVLs. Why? Discussion!
- Most unbalanced out of AVL=Fibonacci trees (i.e. nb of left/right nodes specified by fib. numb.)

$\mathbf{F_n=F_{n-1}+F_{n-2}+1}$ (b=-1 in every node)

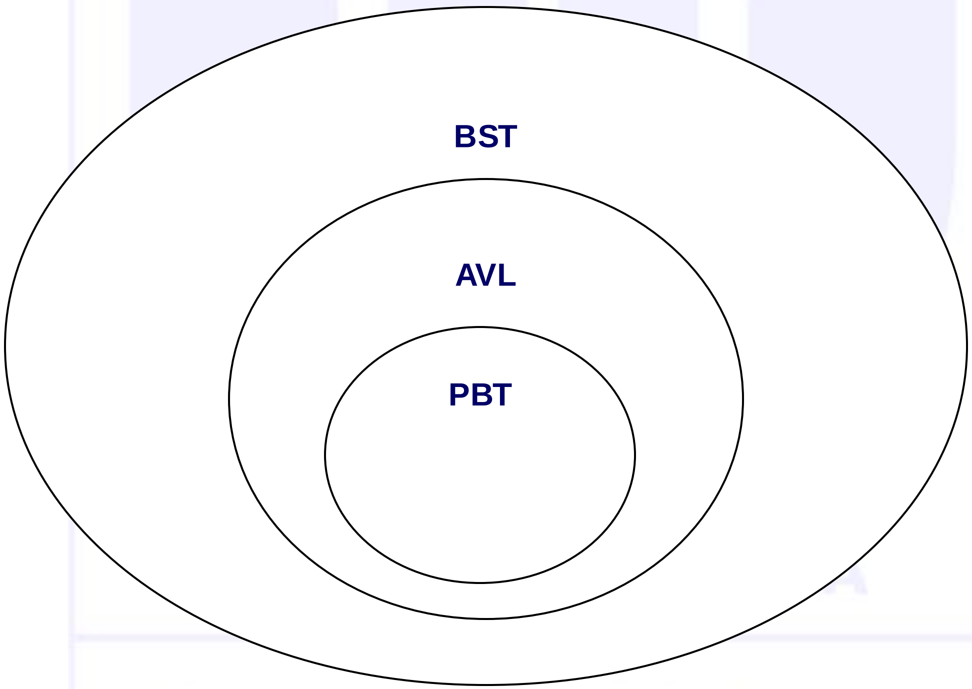
Balanced trees - AVL

- Insert O(h):**ins** as in regular BST
  O(h)=O(lgn)
  requires at most **1/2 rotations O(1)**
- Delete O(h+lgn): **del** as from a regular BST
  O(h) =O(lgn)
  requires at most **lgn rotation O(lgn)**
- h ≤ 1.45lgn=> Good height property;
- easy to maintain for insertion;
- deletion might make many changes in the structure
- Discussion: when should be use AVL trees?

AVL – rotations

- Preserve the search property
- Ensure the balance property
- Self-balancing:
  - Single rotation (see pictures)
  - Double rotation (see pictures)
  - Both take JUST O(1) => do NOT impact the regular insert
- After an insertion, at MOST 1 rotation may occur. Discussion.
- No other situation may occur. Why? Justification.
- After a rotation, the **NEXT** insertion along the same branch would **NOT** require a self-balancing (rotation)
- The same rotations are used for Red-Black trees (see next lecture)!

BST-balanced trees relationship



Augmented DS

- Augmented = additional property and/or behavior to help (i.e. speed up) various tasks preserving ALL existing properties and behavior with (at least) the SAME performance
- Balanced BST are augmented trees (objective, keep the height under control)
- Current objective =better (=faster) select operations on BST
- **Order Statistic (OS) Tree**
- Augmentation= store at the node level as additional information the dimension of the tree (i.e. the number of nodes in the tree rooted by the given node)
- dim[x]=dim[left[x]]+dim[right[x]]+1
- How is calculated? (if the information is not already stored?) – postorder.

Augmented DS – contd.

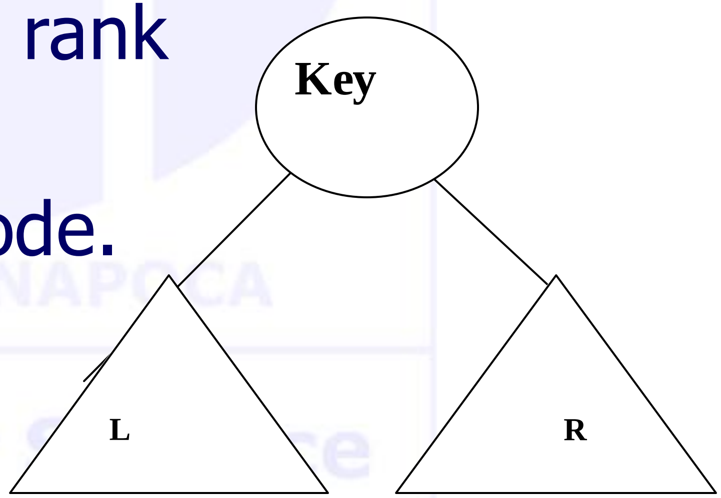
- How to maintain this information for the basic tasks (search, insert, delete, traversal, update)?
- What operations are improved?
- Other tasks: Selection and Ranking
  - Selection (i^{th} selection) = find the node which is the i^{th} one in inorder traversal
  - Selection
    - in arrays – ordered? Not ordered?
    - in lists – ordered.
    - in trees
  - Can we do better for BST?

Selection

- Returns the i^{th} smallest key in the tree
  - rank given (i)
  - key returned (pointer to the i^{th} smallest key in the tree)
- Input: rank (i.e. index in inorder),
- Output: node with the given rank
- Augmentation: dimension =

=nb of nodes rooted by the node.

- dim[x]= dim[left[x]]+
  dim[right[x]]+1
- dim[nil]=0



OS Select **O(h)**

Initial call with root(T) and returns pointer to the i^{th} key

What procedure does it resemble? What differs?

OS_Select(x, i)

```
r<-dim[left[x]]+1//number of nodes on the left + root
if   i=r                              //found it
  then    return x
  else    if i<r                      //ith smallest is on the left
               then
          return OS_Select(left[x],i)
               else                   //ith smallest is on the right
          return OS_Select(right[x],i-r)
```

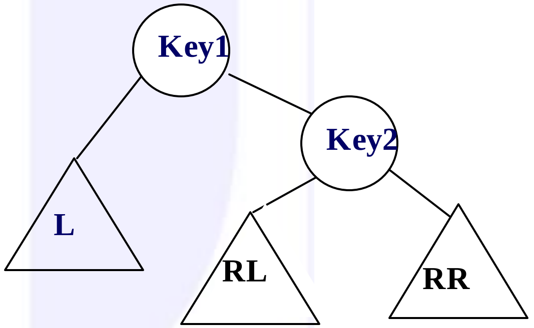
Ranking

- Reverse problem:
  - key given
  - rank returned
- Input:
  - given an existing key from the tree (that is, a pointer to the node containing that key)
- Output:
  - Return its rank in the tree (i.e. its position in the inorder walk)
  - Rank = nb of keys smaller than the checked key in the tree. Approach: count them all (all before = all to left)

Ranking – contd.

Case #1 node is a right child of its parent (Ex: rank Key2)

rank(Key2)=dim(RL)+1+dim(L)+1



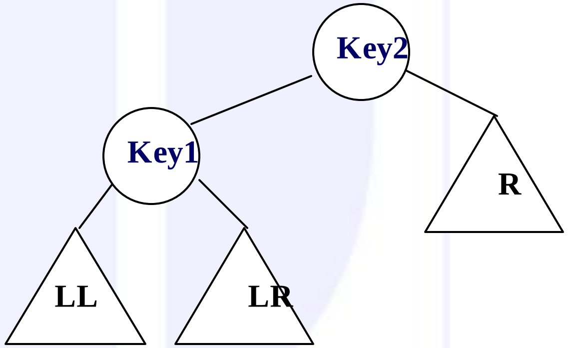
While going upwards in the tree, evaluate what type of child the current node is:

-if a right child (case #1)

Count the nb of nodes in any subtree to the left of the branch starting from the current node (x) up to the root (T)

Ranking – contd.

Case #2 node is a left child of its parent (Ex: rank Key1)
rank(Key1)=dim(LL)+1



While going upwards in the tree, evaluate what type of the child the current node is:

-if a left child (case #2)

Count the nb of nodes in any subtree to the left of the branch starting from the current node (x) up to the root (T)

OS Rank **O(h)**

```
OS_Rank(T,x)
r<-dim[left[x]]+1
y<-x
while y<>root[T]
do
  if y=right[p[y]]
  then                        //case #1
     r<-r+ dim[left[p[y]]]+1
                   //case #2 (do nothing)
  y<-p[y]
return r
```

Augmented trees (by dimension)

- Evaluation (performance for select and rank)
- Worst case O(h)
- For balanced trees h= lgn =>O(lgn)
- OS trees are Red-Black Trees (RBT – check lecture #7)
- What happens (what changes in the tree, besides the regular info/tasks specific to RBT) when updates occur
  - Insert? Discussion/Analysis
  - Delete? Discussion/Analysis

Required Bibliography

- From the Bible – Chapter 12 (Binary Search Trees), Section 14.1 (Dynamic Order Statistics)