Fundamental Algorithms

Lecture #6

Cluj-Napoca

November, 06, 2019

Agenda

- Trees
  - Basic operations
    - walk, search, insert, delete– review
    - min, max, pred, succ
  - Special types
    - Balanced trees
      - PBT (seminar #3)
      - AVL
      - Red-Black (next lecture)
    - Augmented Trees
      - Order-statistic trees
      - Tree/lists

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

BST - delete

- Remove the node
- Cases:
  - Leaf – remove it
  - 1-successor node – link parent with the only child
  - 2-successors 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 - 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 =>

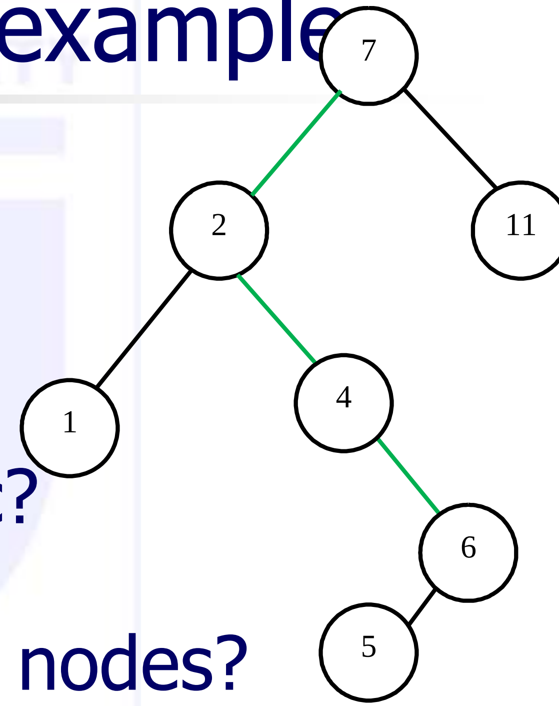
  `find_tree_max(left[x])`

- 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

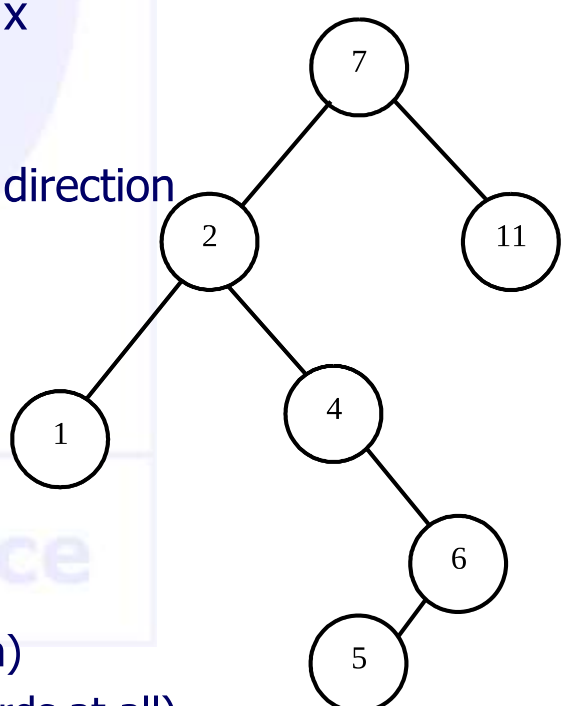
Determine (for succ) a triangle:

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 #3
- many other possibilities (for balance)

Balanced trees - PBT

- Perfect Balanced Trees
- Balance refers to nb of nodes, not to heights
- $b=n_R-n_L \in \{-1, 0, 1\}$
- $h=\lg n$
- Insert O(n): as in regular BST O(h)=O(lgn)
  requires n rotations O(n)
- Delete O(n): as from a regular BST O(h)=O(lgn)
  requires n rotations O(n)
- Best h property; difficult (costly) to maintain

Balanced trees - AVL

- On height (AVL=Adelson-Velskii, Landis)
- $b=h_R-h_L \in \{-1, 0, 1\}$
- PBTs are AVLs
- 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)
- Insert O(h): as in regular BST O(h)=O(lgn)

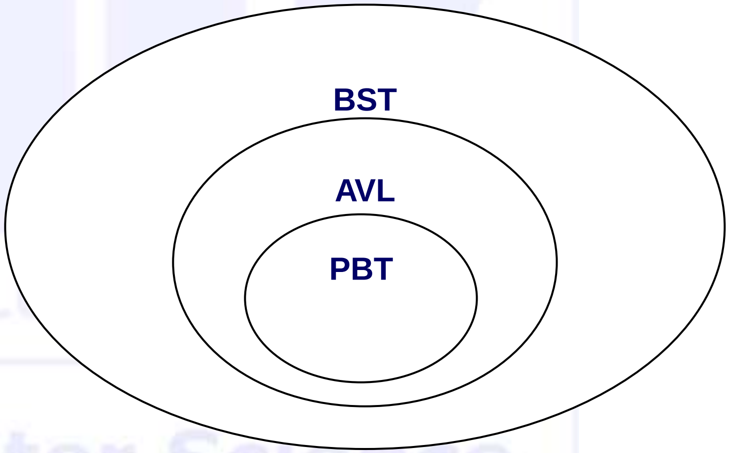
  Blackboard justification requires at most **1 rotation O(1)**
- Delete O(h+lgn): as from a regular BST O(h) =O(lgn)

  requires at most **lgn rotation O(lgn)**
- $h \leq 1.45 \lg n$=> Good height property;
- easy to maintain for insertion; deletion might make many changes in the structure

AVL – rotations

- Preserve the search property
- Ensure the balance property
- Self-balancing:
  - Single rotation (blackboard)
  - Double rotation (blackboard)
  - Both take O(1)!
- 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!

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?

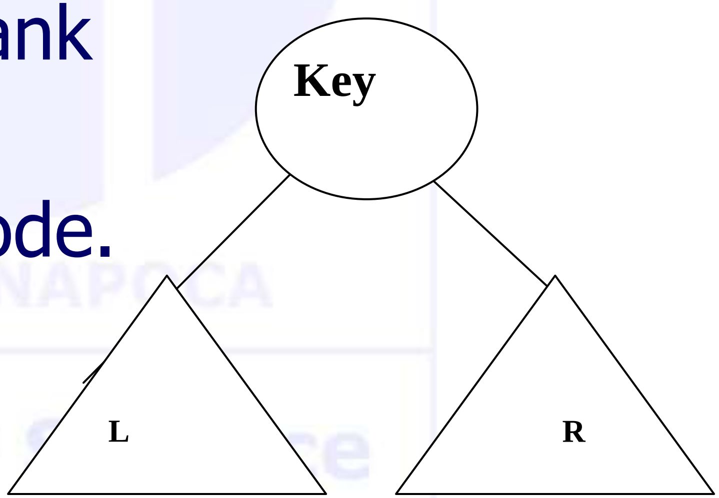
Selection

- 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 at 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

Resembles Hoare's selection on unordered arrays (partition is missing, as we have a BST; so, just the recursive call)

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 tree_select(left[x],i)
               else                  //ith smallest is on the right
          return tree_select(right[x],i-r)
```

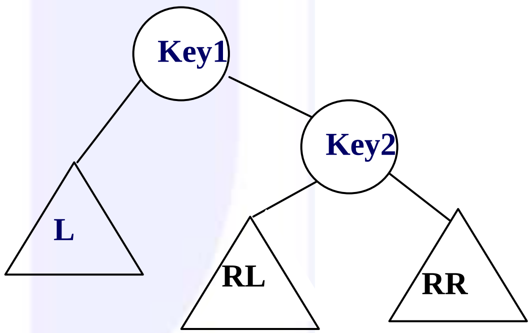
Ranking

- 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



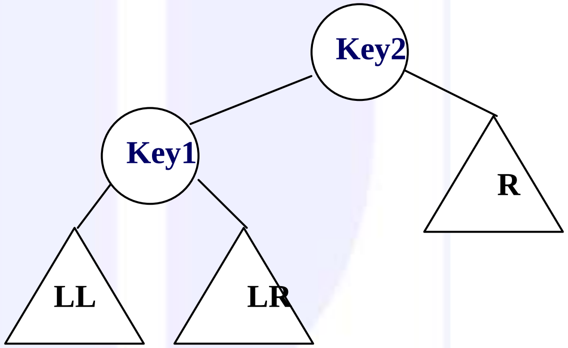
While going upwards in the tree, evaluate what type of child the current node is:

-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
  y<-p[y]                       //case #2
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 OStree, besides the regular info/tasks specific to RBT) when updates occur
  - Insert? Discussion/Analysis
  - Delete? Discussion/Analysis

Augmented trees (type 2)

- Requirements:
  - Regular operations are performed as (**same performance also**) in BST (walk (**O(n)**), search, ins, del (**O(h)**))
  - Several other operations are enhanced (i.e. performed faster)
    - Succ
    - Pred
    - Min
    - Max
    - **All required to be performed in O(1)!!!**
  - BUT NONE of the before-defined operations should degrade their performance

Augmented trees – contd.

- Info in a node:
  - Usual info:
    - key
    - left pointer
    - right pointer
    - parent pointer
  - Supplementary info (see picture on the blackboard):
    - succ pointer
    - pred pointer (together ensure walking through the list)
    - pp ensures min/max oper. (in regular BST, succ/pred calculated **either** based on min/max **or** pp which is determined at the execution time)

Augmented trees – contd.

- The structure acts BOTH as a BST and DLL!!

(check the blackboard for an example)

- Regular operations are:
  - done like in any other BST
  - in addition, need to make some updates
- They (the additional updates) refer to:
  - making the appropriate links within the DLL (set/update the pred and succ pointers)
  - link the double pointer (set/update the pp pointer)

Augmented trees - Insert

- Regular **insert** operation in a BST (x inserted) **+**

```
if x=right[p[x]]   //node inserted = right child
  then    //case #1
      pp[x]<-succ[p[x]]
      dl_list_ins_after(p[x],x)
  else    //case #2; node inserted = left child
      pp[x]<-pred[p[x]]
      dl_list_ins_after(pp[x],x)
```

Augmented trees – Delete

(z = node requested to be removed; it's content is replaced by y's content
y=node actually removed = at most 1 child node;
x = its only child/if any, might be nil;
z=y if z has at most one child)

- Apply regular **delete** operation in a BST **+** code below

```
if right[y]=nil
  then                                //no child to the right
     x<-left[y]                       //x = y's only child
     while x<>nil                     //along x's right branch
          do    pp[x]<-pp[y]                //update pp
                x<-right[x]
     dl_list_del(y)
  else                          //symmetric on the left
```

Augmented trees – Min

- **min** (based on *succ* and pp as opposed to regular BST where *succ* is calculated based on *min* **or** determined *pp*)

```
if x=left[p[x]]
  then
     return succ[pp[x]]
```

//on the leftmost branch, HAS TO BE pp[x]=nil!!!

```
  else
     return succ[p[x]]
```

Augmented trees – Max

- **max** (based on *pred* and pp as opposed to regular BST where *pred* is calculated based on *max* **or** determined *pp*)

```
if x=left[p[x]]
  then
     return pred[p[x]]
  else
     return pred[pp[x]]
```

//on the rightmost branch, **HAS TO BE** pp[x]=nil and pred[nil[T]] = last node in inorder = last node in the list

Augmented trees – contd.

- Particular (initial) cases discussion on the blackboard!
- First **insert** (in the empty tree)

```
Tree_ins(T,x)
if x = root[T]
   then        //the node just inserted is the root = tree is empty
      p[x]<-nil[T]
      pp[x]<-nil[T]
      dl_list_ins_after(pp[x],x)
   else        //the regular case described earlier (needs another if-then-else;
               //here only left case)
      pp[x]<-pred[p[x]]
      dl_list_ins_after(pp[x],x)
```

Homework: updates for delete!