Splay Trees

❖ Splay Trees

Splay tree = one style of "self-balancing" tree ...

Splay tree insertion modifies insertion-at-root method:

- by considering parent-child-granchild (three level analysis)
- by performing double-rotations based on p-c-g orientation

The idea: appropriate double-rotations improve tree balance.

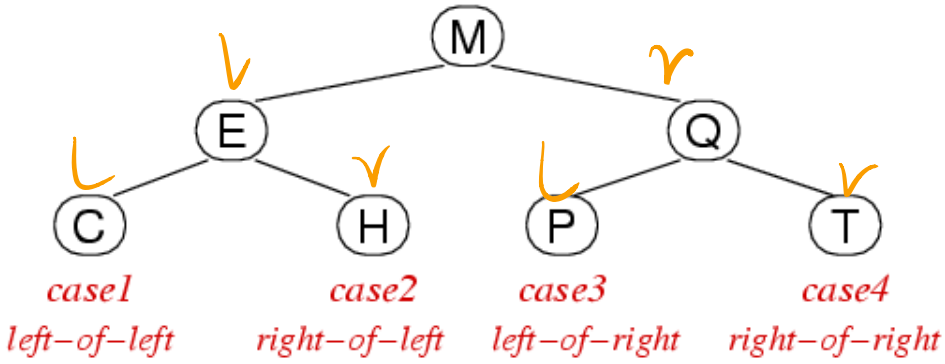
Splay tree implementations also do rotation-in-search:

- can provide similar effect to periodic rebalance
- improves balance, but makes search more expensive

❖ ... Splay Trees

Cases for splay tree double-rotations:

- case 1: grandchild is left-child of left-child
- case 2: grandchild is right-child of left-child
- case 3: grandchild is left-child of right-child
- case 4: grandchild is right-child of right-child



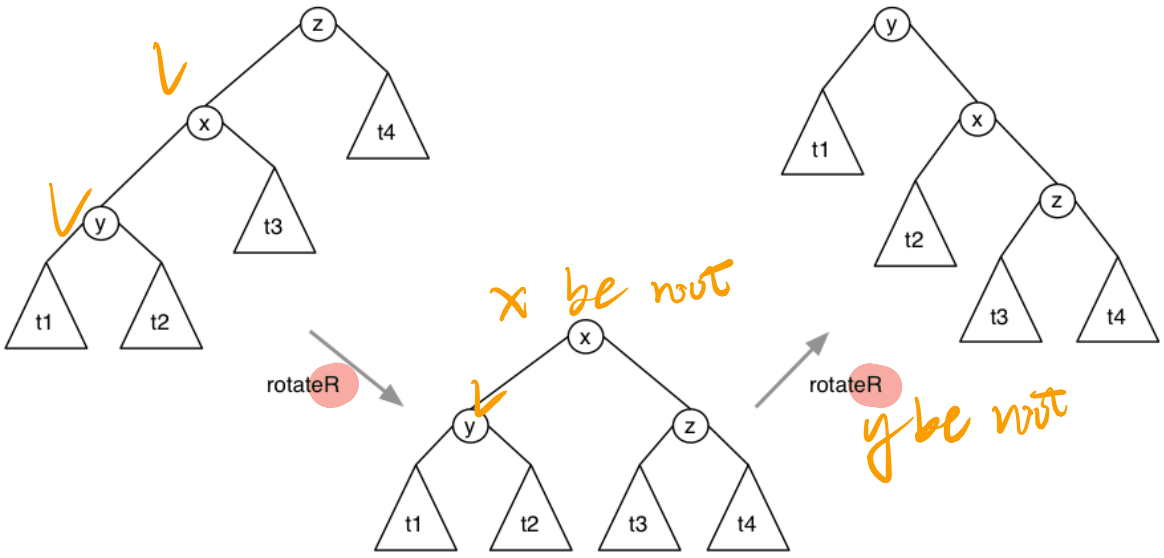
❖ ... Splay Trees

Actions for splay tree double-rotations:

- case 1: grandchild is left-child of left-child
  - insert into left subtree, rotate right, rotate right
- case 2: grandchild is right-child of left-child
  - insert into left subtree, rotate left, rotate right
- case 3: grandchild is left-child of right-child
  - insert into right subtree, rotate right, rotate left
- case 4: grandchild is right-child of right-child
  - insert into right subtree, rotate left, rotate left

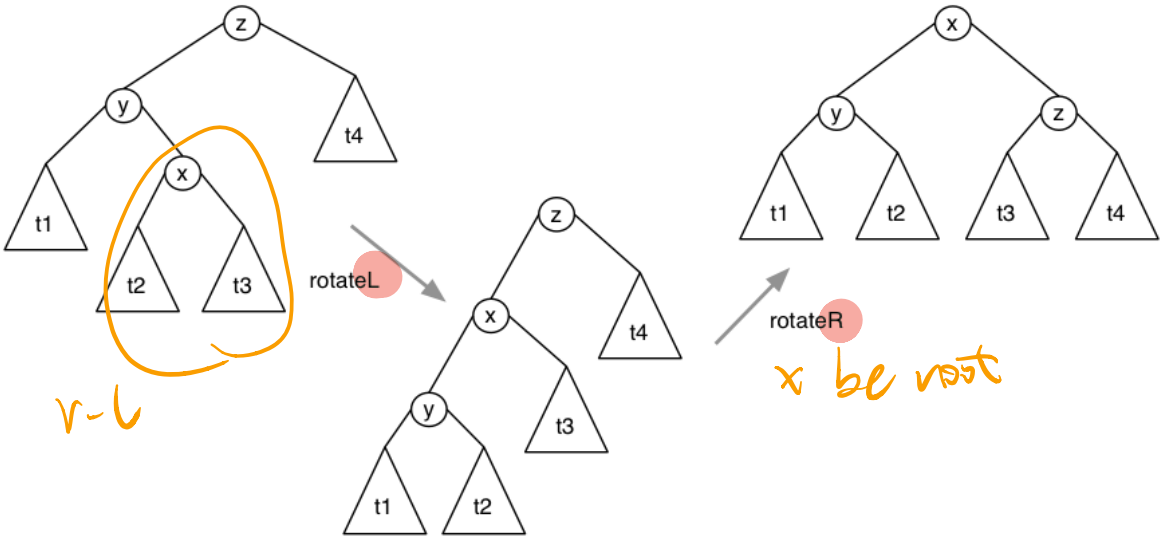
❖ ... Splay Trees

Example: double-rotation case for left-child of left-child:



❖ ... Splay Trees

Example: double-rotation case for right-child of left-child:



❖ Splay Tree Insertion Algorithm

In describing splay trees, it is convenient to use abbreviations

```
tl  = tr->left = left(tree)
tr  = tree->right = right(tree)
tll = tree->left->left
tlr = tree->left->right
trr = tree->right->right
trl = tree->right->left
```

These could be implemented using **`#define`** in C, e.g.

```
#define  tll  t->left->left
```

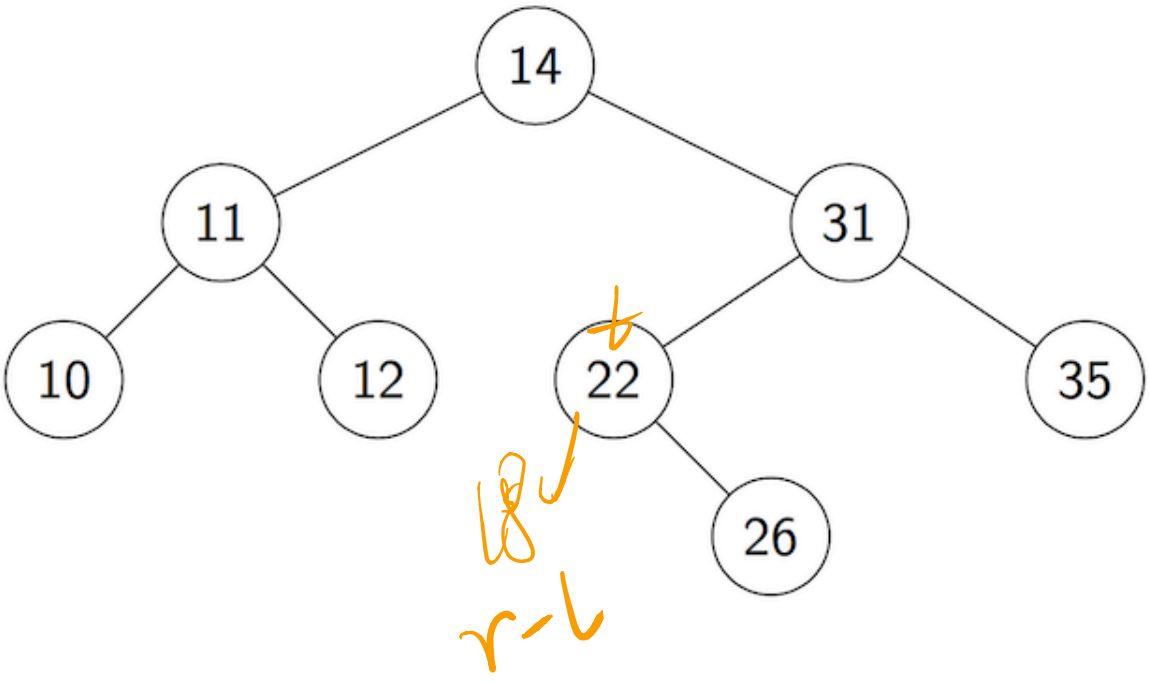
❖ ... Splay Tree Insertion Algorithm

Algorithm for splay tree insertion:

```
insertSplay(tree,item):
|   Input  tree, item
|   Output tree with item splay-inserted
|
|   if tree is empty then return new node containing item
|   else if item=data(tree) then return tree
|   else if item < data(tree) then
|   |   if left(tree) is empty then
|   |      left(tree) = new node containing item
|   |   else if item < data(left(tree)) then
|   |         // Case 1: left-child of left-child
|   |      tll = insertSplay(tll),item)
|   |      tree = rotateRight(tree)
|   |   else  // Case 2: right-child of left-child
|   |      tlr = insertSplay(tlr,item)
|   |      left(tree) = rotateLeft(left(tree))
|   |   end if
|   |   return rotateRight(tree)
|   else if item > data(tree) then
|   |   if right(tree) is empty then
|   |      right(tree) = new node containing item
|   |   else if item < data(right(tree)) then
|   |         // Case 3: left-child of right-child
|   |      trl = insertSplay(trl,item)
|   |      right(tree) = rotateRight(right(tree))
|   |   else  // Case 4: right-child of right-child
|   |      trr = insertSplay(trr,item)
|   |      tree = rotateLeft(tree)
|   |   end if
|   |   return rotateLeft(tree)
|   end if
```

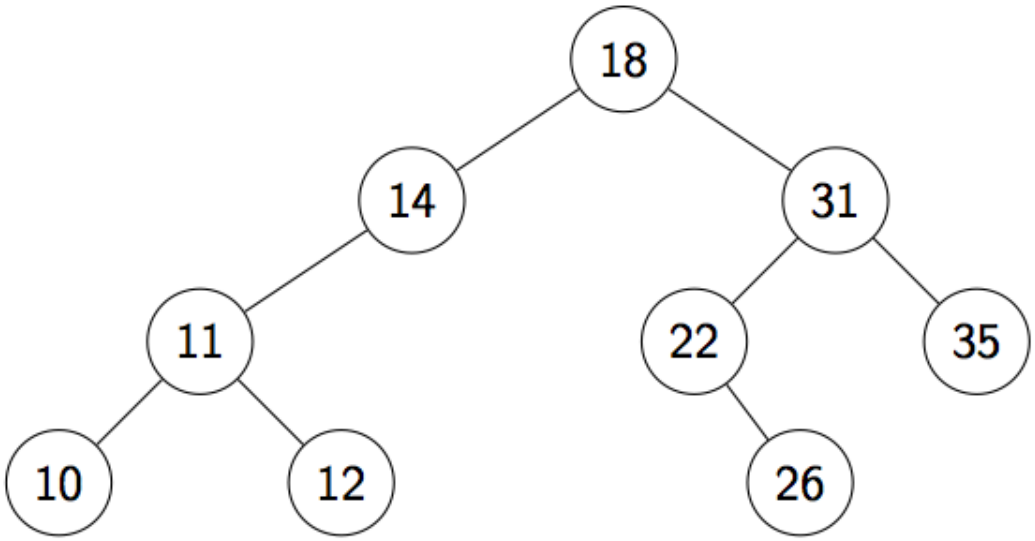
❖ Insertion into Splay Trees

Example: insert **18** into this splay tree:



❖ ... Insertion into Splay Trees

New node is moved to root via right then left rotation



❖ Searching in Splay Trees

Searching in splay trees:

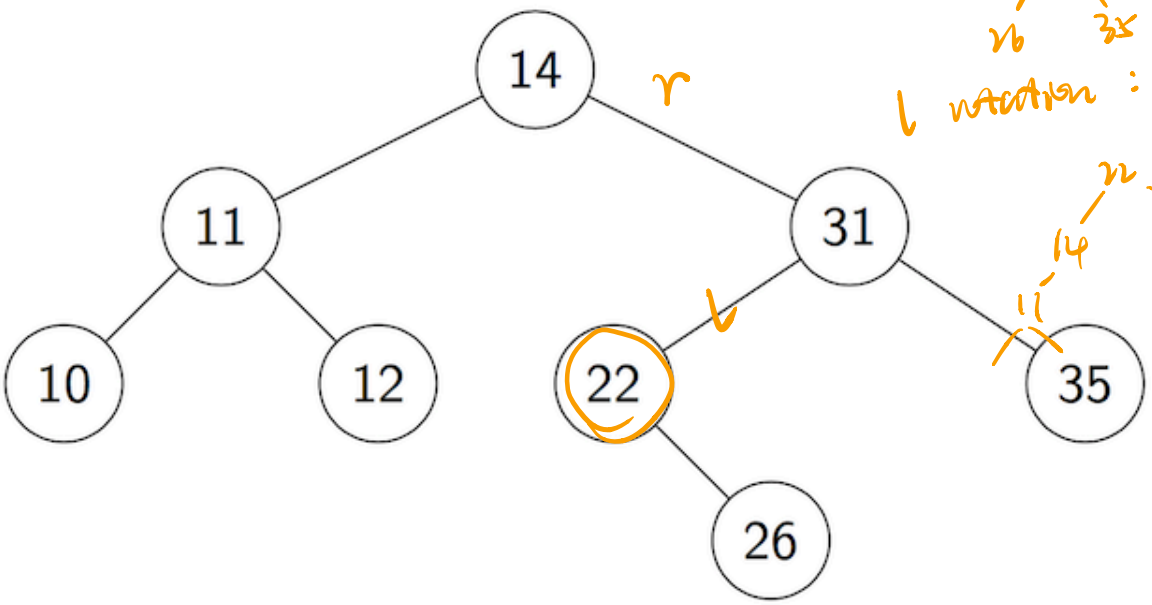
```
searchSplay(tree,item):
|   Input  tree, item
|   Output address of item if found in tree
|          NULL otherwise
|
|   if tree=NULL then
|      return NULL
|   else
|   |   tree = splay(tree,item)
|   |   if data(tree)=item then
|   |      return tree
|   |   else
|   |      return NULL
|   |   end if
|   end if
```

splay() is similar to **insertSplay()**, but doesn't add a node

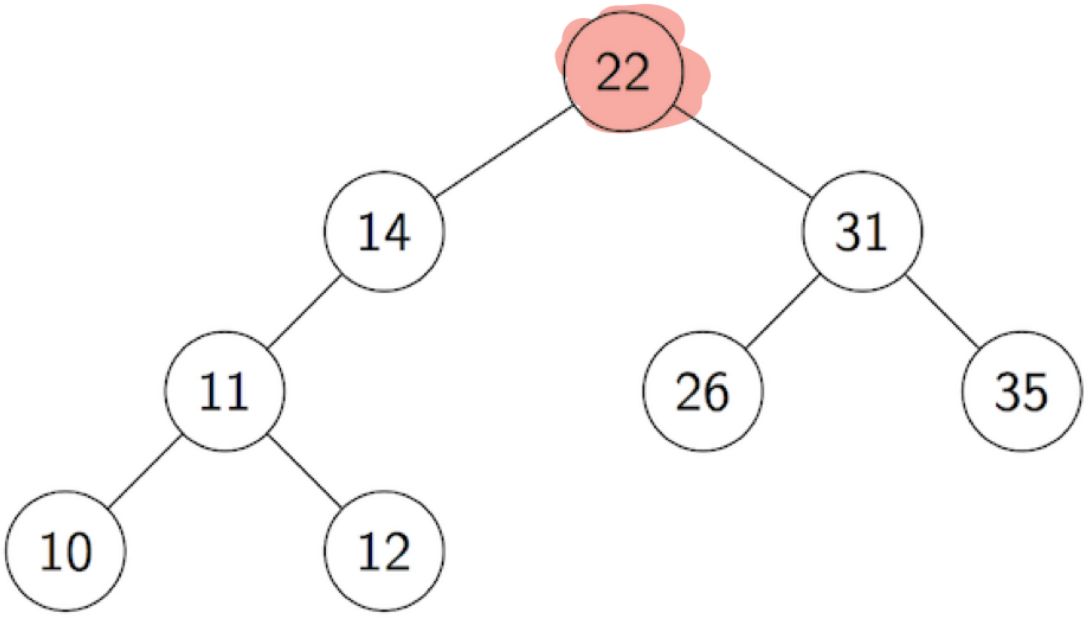
moves **item** to root if found, moves nearest node to root if not found

❖ ... Searching in Splay Trees

Example: search for **22** in the splay tree



Found node is moved to root via right then left rotations



❖ Splay Tree Performance

Analysis of splay tree performance:

- assume that we "splay" for both insert and search
- consider: m insert+search operations, n nodes
- total number of comparisons: average
  $O((n+m) \cdot log_2(n+m))$ $n \log n$

Derivation of the above beyond the scope of this course.

❖ ... Splay Tree Performance

Implications of performance analysis

- no guarantee that cost of each operation is efficient
- but overall cost of operations is efficient

i.e. gives good overall (amortized) cost.

- insert cost not significantly different to insert-at-root
- search cost increases, but ...
  - tends to improve balance on each search
  - moves frequently accessed nodes closer to root

But still has worst-case search cost *O(n)*