Data Structures

AVL Trees

Teacher : Wang Wei

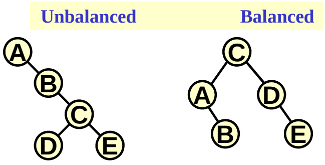
1. Ellis Horowitz,etc., Fundamentals of Data Structures in C++
2. ,
3. ,
4. http://inside.mines.edu/~dmehta/

Dynamic Dictionaries

- Primary Operations
  - Get(key) => search
  - Insert(key, element) => insert
  - Delete(key) => delete
- Additional operations
  - Ascend()
  - Get(index)
  - Delete(index)

Balanced Trees

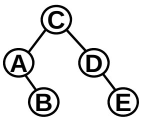
- BST
  - has a high risk of becoming unbalanced
- AVL Tree
  - Should be viewed as a BST with the following additional property
  - For every node, the heights of its left and right subtrees differ by at most 1



AVL Tree

- named for its inventors Adelson-Velskii and Landis

- Definition
  - An empty binary tree is height-balanced
  - If **T** is a nonempty binary tree with $\mathbf{T_L}$ and $\mathbf{T_R}$
    - T_L : left subtrees
    - T_R : right subtrees
  - Then **T** is height-balanced iff
    - **(1)** $\mathbf{T_L}$ and $\mathbf{T_R}$ are height-balanced
    - (2) $|h_L\text{-}h_R|$ <=1
      - h_L : the height of T_L
      - h_R : the height of T_R



balanced

bf (balance factor)

- for every node x, define its balance factor
  - bf(x) = $h_L\text{-}h_R$

balance factor of x = height of left subtree of x
– height of right subtree of x

- balance factor of every node x, bf(x) , is – 1, 0, or 1

- The new tree is not an AVL tree only if you reach a node whose balance factor is either 2 or –2

- this case is said the tree has become unbalanced

Height Of An AVL Tree

- The height of an AVL tree that has ***n*** nodes
  - is at most 1.44 $\log_2$ (n+2)

- The height of every binary tree that has ***n*** nodes
  - is at least $\log_2$ (n+1)

$\log_2$ (n+1) <= height <= 1.44 $\log_2$ (n+2)

- The height or the depth of an AVL tree is at most *O(n)*

- Search for any node cost *O(n)*
- Inserts or deletes cost *O(n)*, even in the worst case

Unbalanced AVL tree

- The new tree is not an AVL tree only if you reach a node whose balance factor is either 2 or –2
- this case is said the tree has become unbalanced

Rotations Types

For a new node Y, let A be the nearest ancestor of Y

Single Rotations

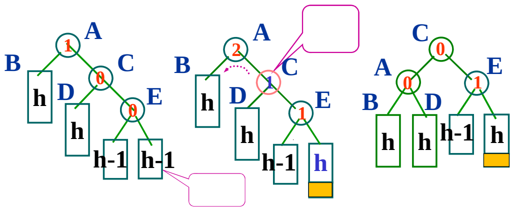
- LL :
  - Y is inserted in the left subtree of the left subtree of A
- RR :
  - Y is inserted in the right subtree of the right subtree of A

Double Rotations

- LR : is RR followed by LL
  - Y is inserted in the right subtree of the left subtree of A
- RL : is LL followed by RR
  - Y is inserted in the left subtree of the right subtree of A

RR

- Be inserted in the right subtree of the right subtree of A
- RR : adjustments to be rebalanced



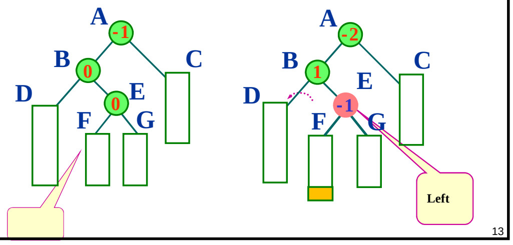
```
              E
AVLTree  E    RotateL (AVLNode  E       ptr)
```

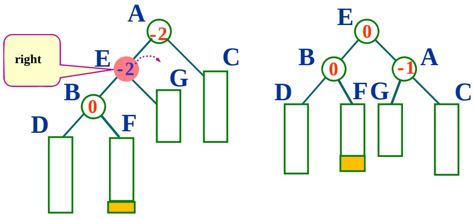
10

LR

- Be inserted in the right subtree of the left subtree of A

- RL : adjustments to be rebalanced





```
                       ->
                    ->
               ->          ->
             ->
->
      ->
    ->      -
               ->          ->
            ->
 ->    == -
      ->
     ->
    ->
```

RL

16

Insertion

- When a new node p is inserted
  - AVL tree has become unbalanced
    - **| bf | > 1** , for any node of the tree

- Method :

  (1) following insert
  (2) retrace path towards root
  (3) adjust balance factors as needed
  (4) stop when reach a node whose balance factor becomes 0, 2, or –2, or the root

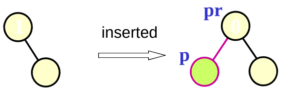
19

Let : bf(p)=0, pr is parent of p

- bf(pr) have three case :

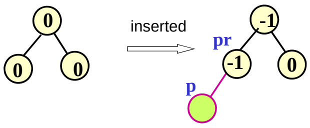
1. bf(pr)=0 ,after inserted

   - Subtree height is unchanged
   - No further adjustments to be done



20

- No further adjustments to be done
- Subtree height is changed, +1/-1
- Must continue on path to root

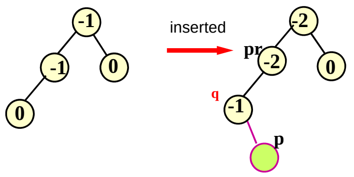


21

- Subtree height is reduced by 1
- Must continue on path to root
- Similar to LL and RL rotations

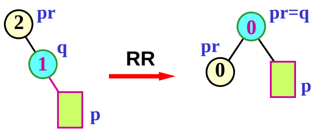
  or

  RR and LR rotations

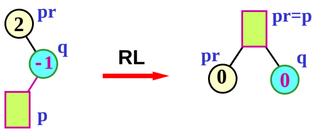


❶

–



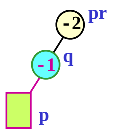
– -



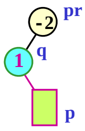
❷ -

– -

–

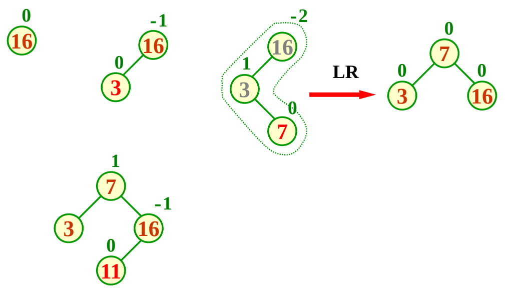


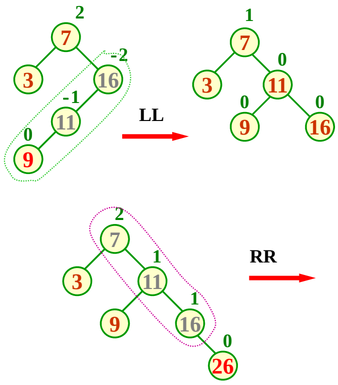
LL

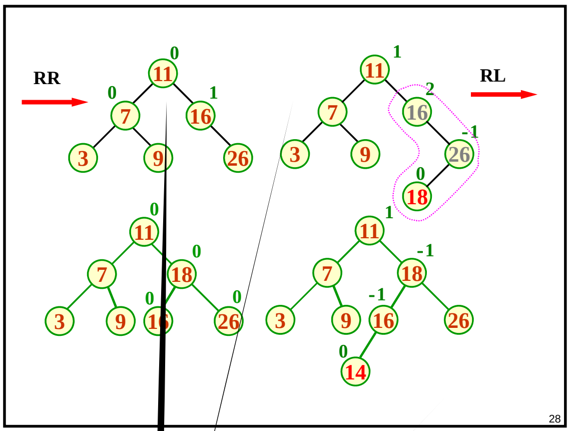


LR

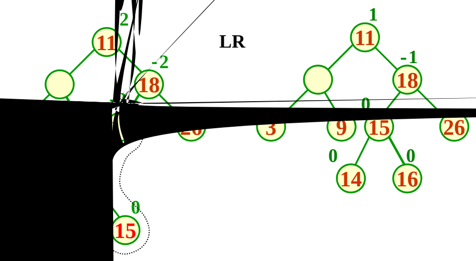
Constructing an AVL tree





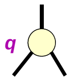


28



29

New Balance Factor Of q



- Deletion from left subtree of q => bf--
- Deletion from right subtree of q => bf++
- New balance factor = 1 or –1
  => no change in height of subtree rooted at q
- New balance factor = 0
  => height of subtree rooted at q has decreased by 1
- New balance factor = 2 or –2
  => tree is unbalanced at q

Imbalance Classification

- Let A be the nearest ancestor of the deleted node
  - whose balance factor has become 2 or –2 following a deletion

- Deletion from left subtree of A => type L
- Deletion from right subtree of A => type R

- Type R => new bf(A) = 2

- So, old bf(A) = 1
- So, A has a left child B
  - bf(B) = 0 => Rotation
  - bf(B) = 1 => Rotation
  - bf(B) = –1 => Rotation

Deletion

1. x is leaf node
   - Remove X
2. x has a child
   - Replace X by the child
   - Remove the child
3. x has two children
   - Replace X by Y
     - Y is the *inorder* predecessor or the the i*norder* successor of X
   - Remove *Y*

A Boolean Variable

- 1 bool *shorter* = true
  - Notes : subtree height is unchanged or reduced

shorter

bf

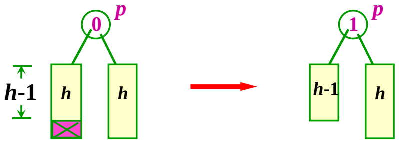
bf

Must continue on path every *p* from *parent*(X) to root

1) Old bf(*p*)=0 and left/right subtree height of *p* is reduced then

New bf (*p*)=1/-1

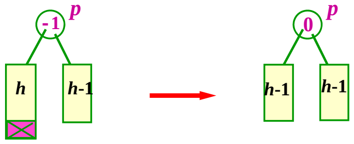
shorter=false



2) Old bf (*p*)= <>0 and the heighter subtree of *p* is reduced then

New bf (*p*)= 0

shorter=true



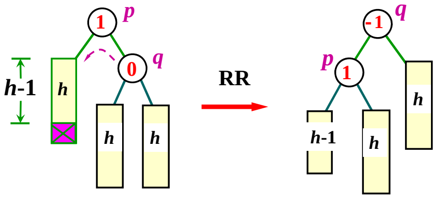
3) Old bf (*p*)= <>0 and the shorter subtree of *p* is reduced
then
New bf (*p*)= 2/-2 => imbalance
shorter=true

How to rebalance

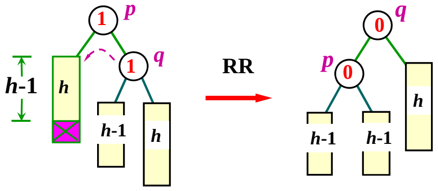
– Rotation : the subtree is reduced
– Let *q* = the heighter subtree root
– Then

q

No further adjustments to be done



b) Old bf(*q*)= bf(*p*)
Rotated LL or RR to rebalance
New bf(*q*)= bf(*p*)=0
shorter=true
Must continue on path to root

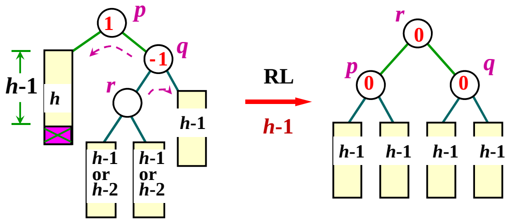


c) Old bf (*q*) = -bf (*p*)

Rotated LR or RL to rebalance, form *q to p*

New bf(root) =0, the bf of other must be adjusted

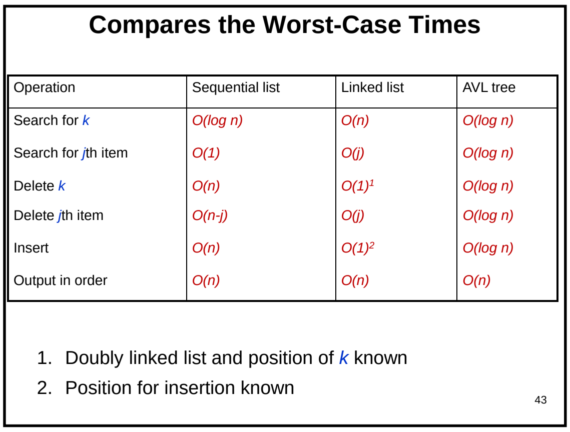
shorter=true



Rotation Frequency

- Insert random numbers
  - No rotation … 53.4% (approx)
  - LL/RR … 23.3% (approx)
  - LR/RL … 23.2% (approx)

Class Definition



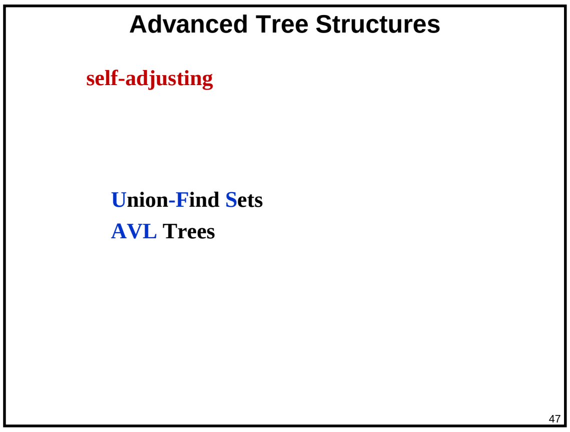
Compares the Worst-Case Times

Operation	Sequential list	Linked list	AVL tree
Search for *k*	*O(log n)*	*O(n)*	*O(log n)*
Search for *j*th item	*O(1)*	*O(j)*	*O(log n)*
Delete *k*	*O(n)*	*O(1)*[1]	*O(log n)*
Delete *j*th item	*O(n-j)*	*O(j)*	*O(log n)*
Insert	*O(n)*	*O(1)*[2]	*O(log n)*
Output in order	*O(n)*	*O(n)*	*O(n)*

1. Doubly linked list and position of *k* known
2. Position for insertion known

43

44

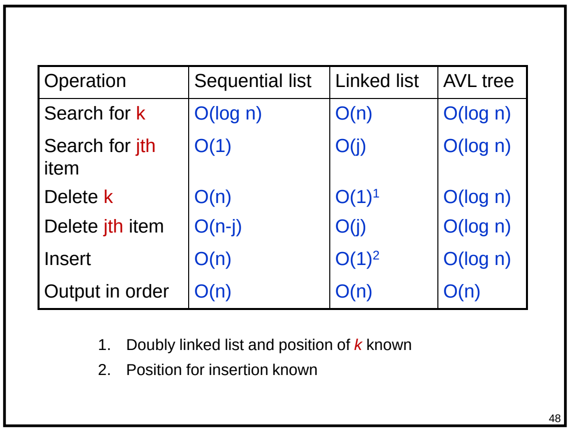


Advanced Tree Structures

self-adjusting

Union-Find Sets

AVL Trees



Operation	Sequential list	Linked list	AVL tree
Search for k	O(log n)	O(n)	O(log n)
Search for jth item	O(1)	O(j)	O(log n)
Delete k	O(n)	O(1)[1]	O(log n)
Delete jth item	O(n-j)	O(j)	O(log n)
Insert	O(n)	O(1)[2]	O(log n)
Output in order	O(n)	O(n)	O(n)

1. Doubly linked list and position of *k* known
2. Position for insertion known