Data Structures

AVL Trees

Teacher : Wang Wei

1. Ellis Horowitz,etc., Fundamentals of Data Structures in C++
2. 殷人昆,　　　　数据结构
3. 金远平,　　　　数据结构
4. http://inside.mines.edu/~dmehta/

1

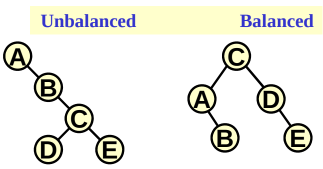
Dynamic Dictionaries

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

2

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

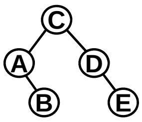


3

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 T_L and T_R
    - T_L : left subtrees
    - T_R : right subtrees
  – Then **T** is height-balanced iff
    (1) T_L and 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



4

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

5

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 $\log_2 (n+1)$

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

- The height or the depth of an AVL tree is at most $O(\log_2 n)$
- Search for any node cost $O(\log_2 n)$
- Inserts or deletes cost $O(\log_2 n)$, even in the worst case

6

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

7

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

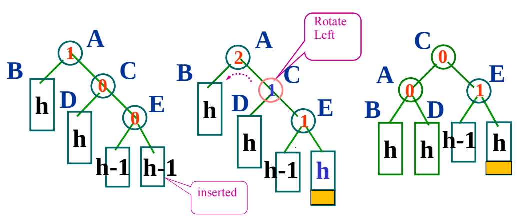
Double Rotations

- 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

8

RR

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

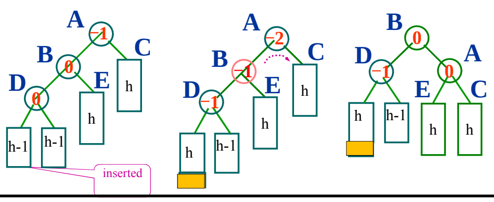


9

```
template <class E>
void AVLTree<E>::RotateL (AVLNode<E> *& ptr)
{  //右子树比左子树高: 做左单旋转后新根在ptr
   AVLNode<E> *subL = ptr;
                    ptr = subL->right;
                    subL->right = ptr->left;
                    ptr->left = subL;
                    ptr->bf = subL->bf = 0;
}
```


LL(右单旋转)

- Be inserted in the left subtree of the left subtree of A
- LL : adjustments to be rebalanced

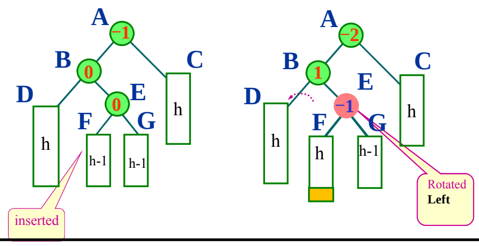



```
template <class E>
void AVLTree<E>::RotateR (AVLNode<E> *& ptr)
{ //左子树比右子树高,旋转后新根在ptr
   AVLNode<E> *subR = ptr;          //要右旋转的结点
                    ptr = subR->left;
                    subR->left = ptr->right;
                                                //转移ptr右边负载
                    ptr->right = subR;          //ptr成为新根
                    ptr->bf = subR->bf = 0;
}
```

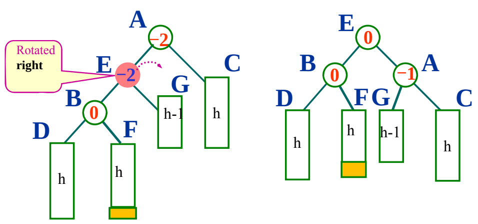
LR

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

- RL : adjustments to be rebalanced



13



14

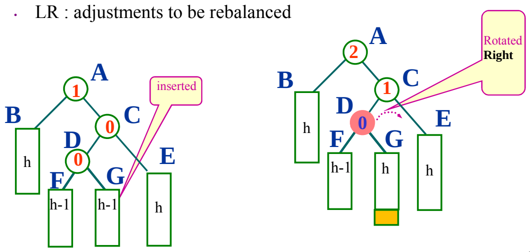
```
template <class E>
void AVLTree<E>::RotateLR (AVLNode<E> *& ptr)
{  AVLNode<E> *subR = ptr;
   AVLNode<E> *subL = subR->left;
                   ptr = subL->right;
                   subL->right = ptr->left;
                   ptr->left = subL;
  if (ptr->bf <= 0)
           subL->bf = 0;
  else  subL->bf = - 1;
                  subR->left = ptr->right;
                  ptr->right = subR;
   if (ptr->bf = =  - 1)
           subR->bf = 1;
   else  subR - >bf = 0;
           ptr->bf = 0;
}
```

15

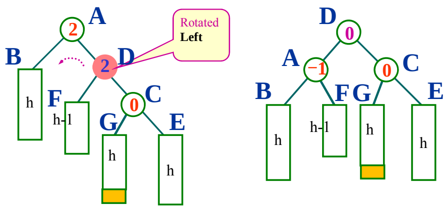
RL

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

- LR : adjustments to be rebalanced



16



17

```
template <class E>
void AVLTree<E>::
        RotateRL (AVLNode<E> *& ptr)
{
   AVLNode<E> *subL = ptr;
   AVLNode<E> *subR = subL−>right;
           ptr = subR−>left;
           subR−>left = ptr−>right;
           ptr−>right = subR;
   if (ptr−>bf >= 0) subR−>bf = 0;
   else subR−>bf = 1;
          subL−>right = ptr−>left;
          ptr−>left = subL;
   if (ptr−>bf == 1) subL−>bf = - 1;
   else subL−>bf = 0;
        ptr−>bf = 0;
};
```

18

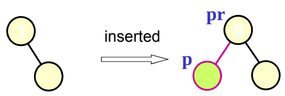
Insertion

- 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

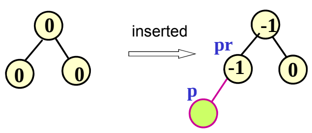
Let : bf(p)=0, pr is parent of p

- bf(pr) have three case :

1. bf(pr)=0 ,after inserted
   - Subtree height is unchanged
   - No further adjustments to be done



2. |bf(pr)| = 1
   - bf(pr)=0 , before inserted
   - No further adjustments to be done
   - Subtree height is changed, +1/-1
   - Must continue on path to root
     - pr = Parent(pr)

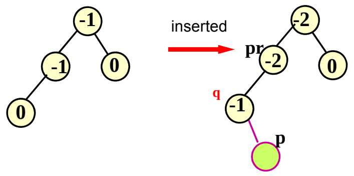


3. |bf(pr)| = 2

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

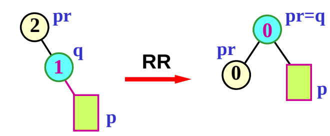
  or

  RR and LR rotations

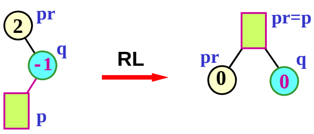


❶ bf(pr) = 2

– bf(q)=1

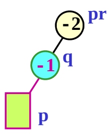


– bf(q) = -1

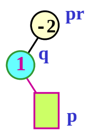


❷ bf(pr) = -2

– bf(q) = -1
– bf(q) =1



LL

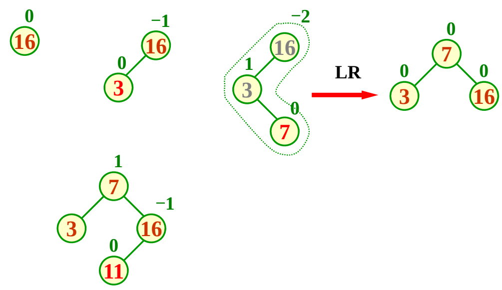


LR

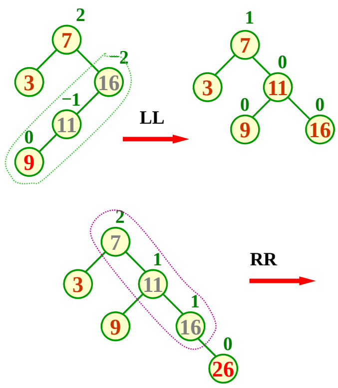
Constructing an AVL tree

25

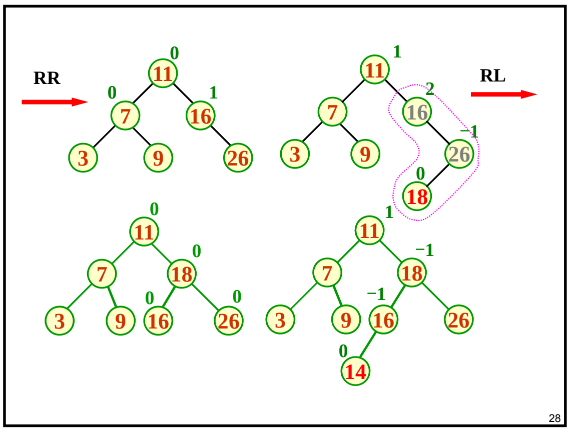
- keys : { 16, 3, 7, 11, 9, 26, 18, 14, 15 }
- Inserting and rebalancing

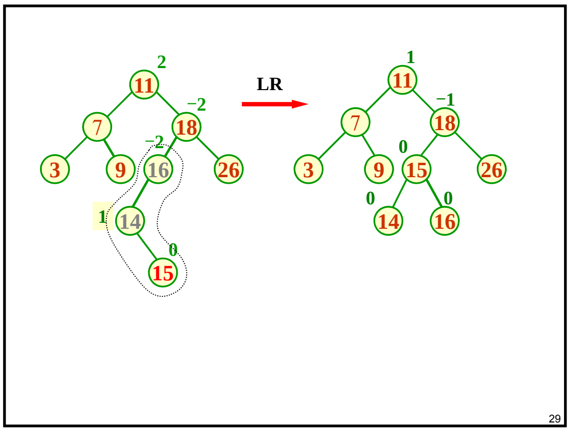


26



27



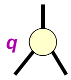


Deletion

1. x is leaf node
2. x has a child

30

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

31

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

32

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*

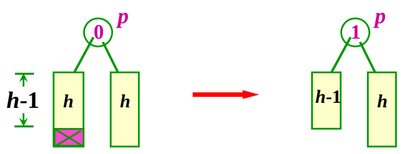
33

A Boolean Variable

- 1 bool *shorter* = true
  - Notes : subtree height is unchanged or reduced
- 2 For every node, new balance factor depends on
  - ***shorter***
  - **bf**(X)
  - **bf**(child(X))
- 3 Must continue on path every *p* from *parent*(X) to root
  - if shorter=false stop
  - else

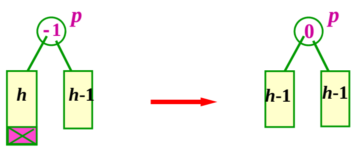
34

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



35

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



36

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

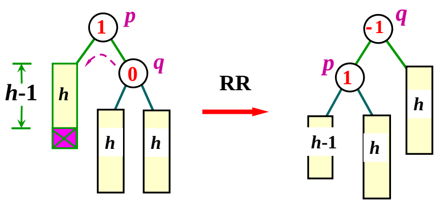
37

a) Old bf (q)= 0

Rotated LL or RR to rebalance

shorter=false

No further adjustments to be done



38

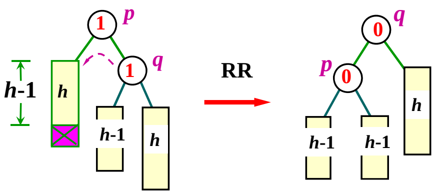
b) Old bf(q)= bf(p)

Rotated LL or RR to rebalance

New bf(q)= bf(p)=0

shorter=true

Must continue on path to root



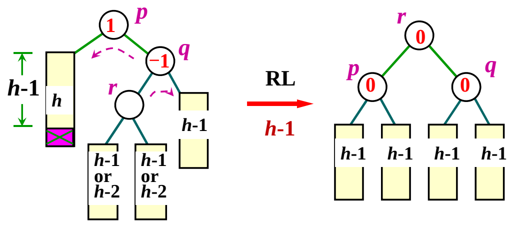
39

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

```
//AVL树结点的类定义
#include <iostream.h>
#include "stack.h"
template <class E>
struct AVLNode : public BSTNode<E>
{
    int bf;
   AVLNode() { left = NULL; right = NULL; bf = 0; }
   AVLNode (E d, AVLNode<E> *l = NULL,
                   AVLNode<E> *r = NULL)
       { data = d;  left = l;  right = r;  bf = 0; }
};
```

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


```
//平衡的二叉搜索树（AVL）类定义
template <class E>
class AVLTree : public BST<E>
{
public:
   AVLTree() { root = NULL; }        //构造函数
   AVLTree (E Ref) { RefValue = Ref; root = NULL; }
     //构造函数：构造非空AVL树
```



```
int Height() const;   //高度
AVLNode<E>* Search (E x,
              AVLNode<E> *& par) const;  //搜索

bool Insert (E& e1) { return Insert (root, e1); } //插入
bool Remove (E x, E& e1)
    { return Remove (root, x, e1); }   //删除

friend istream& operator >> (istream& in,
    AVLTree<E>& Tree);     //重载：输入

friend ostream& operator << (ostream& out,
    const AVLTree<E>& Tree);  //重载：输出
```

```
protected:
    int Height (AVLNode<E> *ptr) const;

    bool Insert (AVLNode<E>*& ptr, E& e1);
    bool Remove (AVLNode<E>*& ptr, E x, E& e1);
    void RotateL (AVLNode<E>*& ptr);          //左单旋
    void RotateR (AVLNode<E>*& ptr);          //右单旋
    void RotateLR (AVLNode<E>*& ptr); //先左后右双旋
    void RotateRL (AVLNode<E>*& ptr); //先右后左双旋
};
```

46

Advanced Tree Structures

- **self-adjusting** data structure
  - Dynamic collections of elements
- Such as
  - **Union-Find Sets**
  - **AVL Trees**
  - Red-Black Trees
  - Splay Trees
  - Tries

47

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

48