- Fibonacci Heap

union(H_1, H_2) : create and return a new heap contain all element of heaps H_1 and H_2.

decrease(Node X, Key k) : x的key变小为k, restore the heap.

Roots of trees are also connected as a circular doubly linked list.

insert: put node into root list, update H.min if necessary.

* extractMin : ① remove min and concatenate its children into root list.

去除min. size $\lfloor \log_\phi n \rfloor$ $\phi = 1.618$

② consolidate the root list. { 让每个root有不同degree. 若x, y degree同, x.key ≤ y.key, 把y变成x的child. use array D(n)+1 size to store.

O(log n). 从min node右开始

- Maximum number of node in a tree. $S(n) = S(n-1) + S(n-2)$

decreasekey : (1) min heap property not violated 只change pointer

(2) violate min heap :

① cut between node & parent.

② move subtree to root list

若一个node n 失去child两次, subtree rooted at n should be cut from n's parent and move to root list. (可能recurse, 需mark flag 失去为true).

amortized time O(1)

- ## Binary Search Tree

左subtree 都小，右subtree 都大

search, insertion, removal average case $O(\log n)$ *

insert: 要 node *&root 才接进树. Ps. 若key存在直接return

remove: node *&root

when root->item.key == k

(1) 移去node 是 leaf 直接删

(2) degree-one node

(3) degree-two node

case (2)
```
{node *tmp = root;
 root = root->left/right (非空) 接上
 delete tmp }
```

case (3)
```
{node *& replace = findMax(root->left)
 root->item = replace->item
 node *temp = replace
 replace = replace->left   接左树,可能null
 delete temp }
```

- ## Average case Time complexity

depth (height) of BST is h. worst case $O(h)$, average $O(h)$

n nodes BST. worst case $O(n)$

- K-d tree

左边 subtree 的 node 在 DIM 小于 root，右 subtree 的 node DIM 上大于 root
DIM 不断 cycle.

insert:

k-d Tree Insert

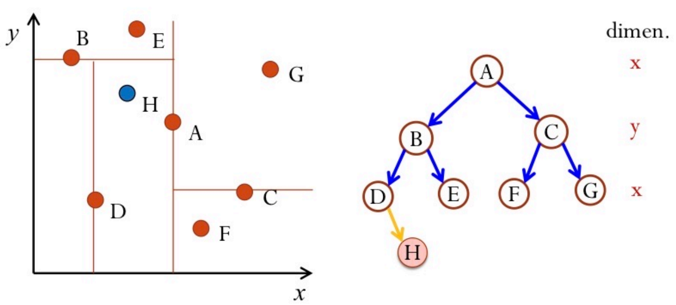
- If new item's key is equal to the root's key, return;
- If new item has a key smaller than that of root's along the dimension of the current level, recursive call on left subtree
- Else, recursive call on the right subtree
- In recursive call, cyclically increment the dimension

```
void insert(node *&root, Item item, int dim) {
  if(root == NULL) {
    root = new node(item);
    return;
  }
  if(item.key == root->item.key) // equal in all
    return;                      // dimensions
  if(item.key[dim] < root->item.key[dim])
    insert(root->left, item, (dim+1)%numDim);
  else
    insert(root->right, item, (dim+1)%numDim);
}
```

dim refers to the dimension of the root

Insert Example

- Insert H
- Initial function call: insert(A, H, 0) // 0 indicates dimension x



search.

```
node *search(node *root, Key k, int dim) {
  if(root == NULL) return NULL;
  if(k == root->item.key)
    return root;
  if(k[dim] < root->item.key[dim])
    return search(root->left, k, (dim+1)%numDim);
  else
    return search(root->right, k, (dim+1)%numDim);
}
```

Time complexities of insert and search are all $O(\log n)$

find min,

Find Minimum Value in a Dimension

```
node *findMin(node *root, int dimCmp, int dim) {
// dimCmp: dimension for comparison
  if(!root) return NULL;
  node *min =
    findMin(root->left, dimCmp, (dim+1)%numDim);
  if(dimCmp != dim) {
    rightMin =
      findMin(root->right, dimCmp, (dim+1)%numDim);
    min = minNode(min, rightMin, dimCmp);
  }
  return minNode(min, root, dimCmp);
}
```

左总要找

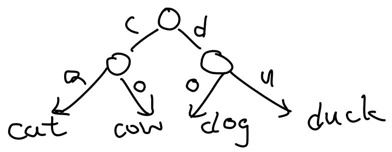
DIM不同 找右 min 后 compare

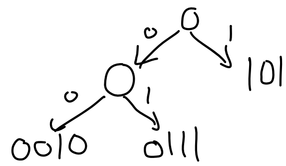
- **minNode** takes two nodes and a dimension as input, and returns the node with the smaller value in that dimension

☆ remove: 有 right subtree 找 right subtree 中 current DIM 的 min value. M
replace value of R with M.
recurse on M until leaf reached, remove the leaf.
若无 right subtree，找 left subtree 中的 max. replace and recurse.

- Tries

Data records only stored in leaf nodes. Internal nodes not store nodes, they're branch points direct the search process.





prefix. 用 "$" to indicate the end

search { no branch : return false
reach leaf : compare with the key at leaf

remove : 若删完只有一个child, remove parent node, move key c one level up.

Worst case inserting or finding a key consists of k symbol is $O(k)$

- AVL Tree

n nodes, average case for search, insertion, removal on BST all $O(\log n)$.
worst case still $O(n)$.

"balanced" : 1. Height of tree of n nodes = $O(\log n)$
2. Balanced condition can be maintained efficiently : $O(\log n)$ to rebalance a tree.

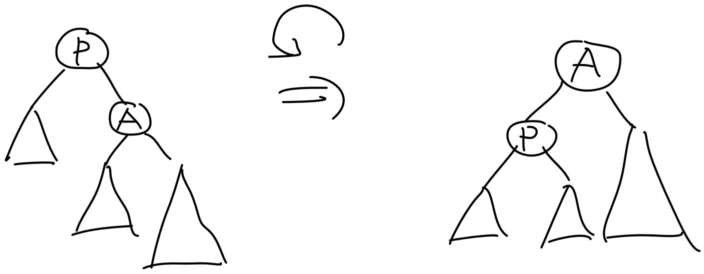
AVL tee's balanced condition:

1. empty tree is AVL balanced
2. non-empty balanced if { left and right subtrees are AVL balanced
height of left and right subtrees differ by at most 1

height h of AVL balanced tree n nodes, $\log_2(n+1)-1 \le h \le 1.44\log_2(n+2)$

Right rotation,

left rotation.

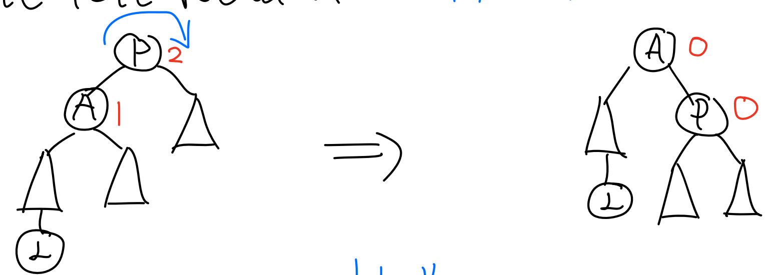


Balance Factor: T_L, T_r be left, right subtree of a tree rooted at T. h_L be height of T_L, h_r be height of T_r.

$B(T) = h_L - h_r$. every node T, $|B_T| \leq 1$

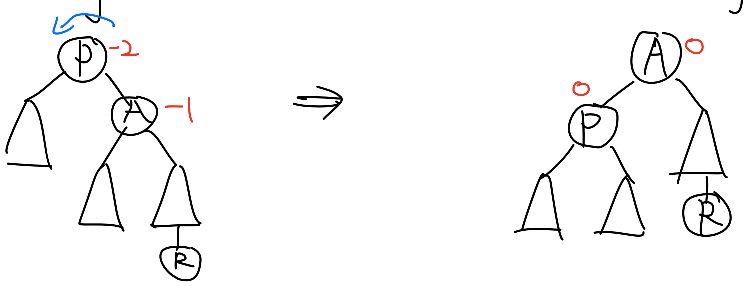
- Insertion: 加完以后可能 unbalance，要 check

▷ left-left rotation: 中左为+.

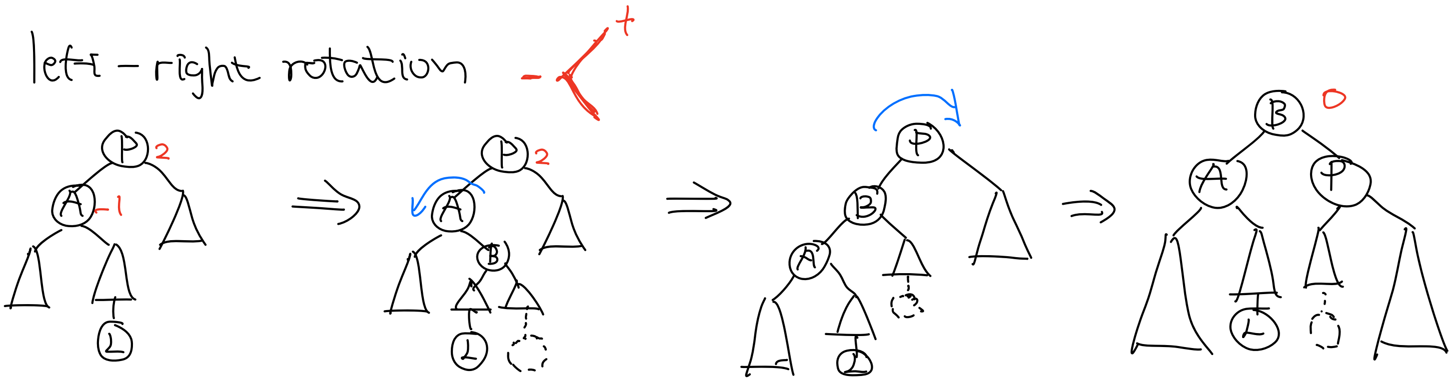


① A, P have BF = 0.
② height of tree after rotate = ~ —— — before insert

▷ right-right rotation. 中右为− node, node->right: BF negative.



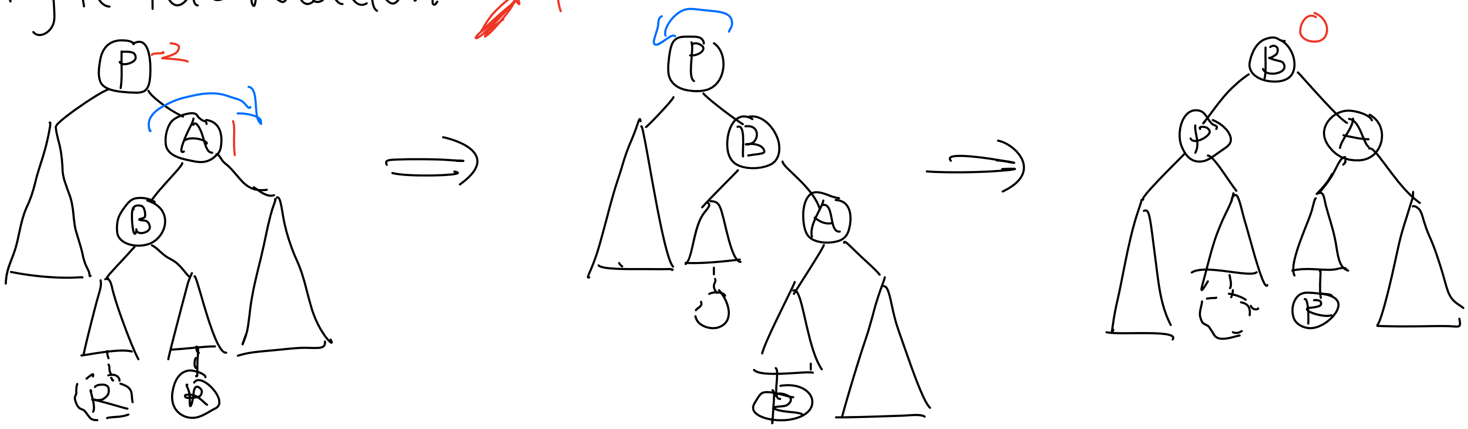
▷ left-right rotation



BF: node positive, left subtree negative.

BF: B is 0; height after rotation = height before insertion

△ right-left rotation



BF: node negative, right subtree positive.

20 Red - black Tree

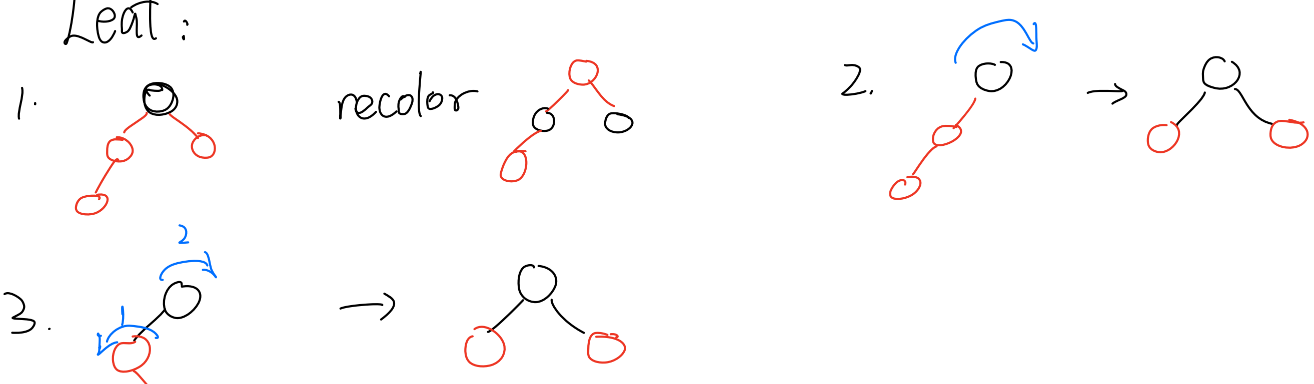
- property:
  - root black;
  - red node only black children
  - every path from node x to NULL must have same number of black nodes.
- chain of length 3 cannot be red-black tree.
- black height of node x : # of black nodes from x to NULL, include x.
- red node 一定有 2 children (或无 child), be black.
  - black node 若只有 1 child 必为 red.
- red-black tree with n nodes has height $\leq 2\log_2(n+1)$
- search, min, max, succ, pred $O(\log n)$ in worst case.
- Insertion: new leaf must be red
  - 若 parent red, grandparent black ⇒ violation at leaf.
  - ↳ parent → black, grandparent → red
  - 3种情况: P20开始
  - violation at leaf / Internal Nodes.

rotation : $O(1)$.

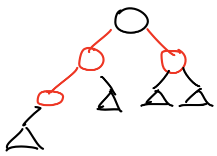
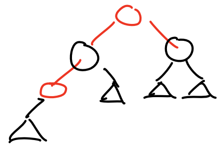
recoloring : worst case $O(\log n)$

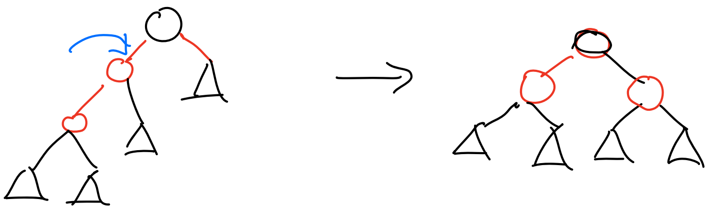
Run time complexity $O(\log n)$.

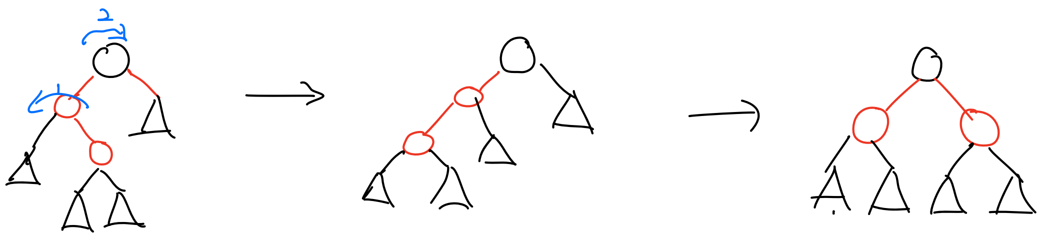
Leaf:



Internal Nodes.

1.  recolor 

2. 

3. 

rotation O(1)

worst case O(log n).

Runtime complexity O(log n)