

目录

1. How do we keep/make a BST balanced?
 1. Right rotation
 2. Left rotation
 3. Examples
 4. Counting the number of nodes in a subtree
 5. Selecting the i-th element from a BST
2. Balanced Trees
 1. Approach 1: Global rebalancing
 1. Rotating with a count
 2. Partitioning
 3. Balancing the BST
 4. Problems
 2. Approach 2: Local Rebalancing
 1. Splay tree demo link
 2. zigzag operation
 3. zigzig operation
 4. Example
 5. Inserting an item in a splay tree
 6. Operation splay-search
 7. Operation splay-insertion
 8. Operation splay-deletion
 9. Minimum or maximum of a splay tree
 10. in summary
 11. Splay Tree Analysis
3. 2-3-4 trees
 1. 1. Implementation

How do we keep/make a BST balanced?

We can rotate a child node to the position of its parent

- interchange the root with one of its children:
 - *right rotation*: interchange root and left child
 - *left rotation*: interchange root and right child

... but we must make sure the BST order is still satisfied

- the value of the parent is greater than the left child, and less than ...

Right rotation

root (E) 的左子树 (C) 作为根节点, E变为C的右节点
C的右节点变为E的左节点
其实就是把C作为root, 自然DE都大于C需要在右边

Make the **left** child the new root means:

```
its right child moves to the root's left child
the root moves to its right child
```

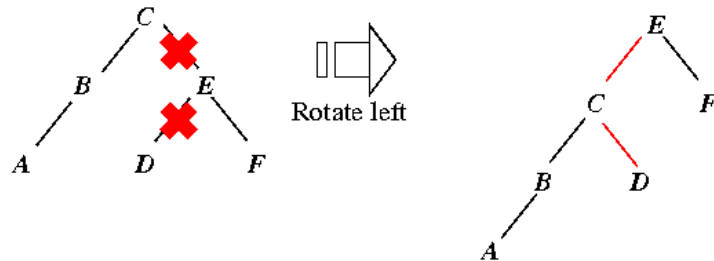
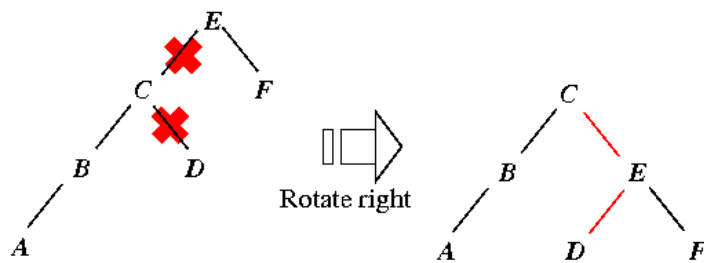
Left rotation

往左旋转的话, 就是让E作为root, E是最大的
因此CD都在左侧

Make the **right** child of the root the new root means:

```
its left child moves to the root's right child
the root moves to its left child
```

Examples



Notice that in both cases that:

- C and E remain 'in order'
- the subtree **D** between C and E
 - remains 'in order'
 - moves to the other side of the root

节点重新组合，新建一个树的指针，
然后访问的时候就是这棵树了

In code:

切换行号显示

```

1 // NOT REAL CODE: CHECKS REQUIRED: ACTUAL CODE BELOW
2 Tree rotateR(Tree root) {    // old root
3     Tree newr = root->left;  // newr is the new root
4     root->left = newr->right; // old root has new root's right child
5     newr->right = root;      // new root has old root as right child
6     return newr;            // return the new root
7 }
8 // NOT REAL CODE: CHECKS REQUIRED: ACTUAL CODE BELOW
9 Tree rotateL(Tree root) {    // old root
10    Tree newr = root->right;   // newr will become the new root
11    root->right = newr->left;   // old root has new root's left child
12    newr->left = root;         // new root has old root as left child
13    return newr;             // return the new root
14 }
```

Note:

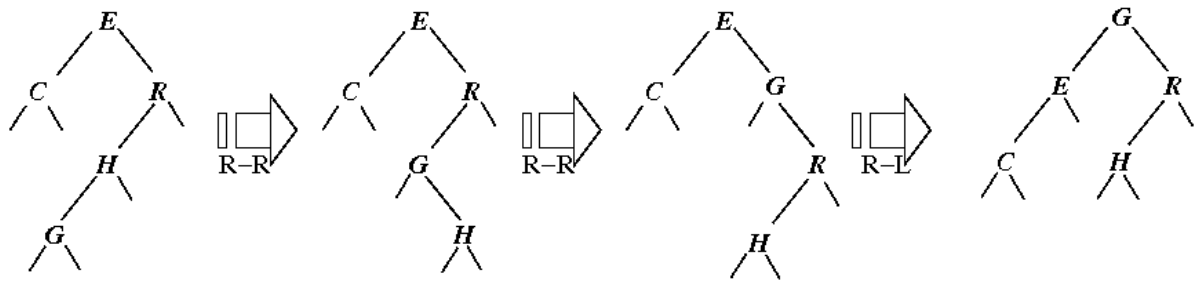
- the change is local involving just:
 - 2 nodes (e.g. in a right rotate: *root* and *root-left*) and
 - 2 links (e.g. in a right rotate: *root->left* and *newr->right*)

Application:

- rotating can bring more balance into the tree
- rotations always maintain BST order
- can use rotations to insert nodes into a BST at the root where it is easy to access (instant hit)
 - how do we do this?
 1. as before: recursively descend BST and insert the new element as a leaf
 2. then rotate to make this new leaf the root of the whole tree

Example:

- assume that we have just inserted node **G** in the BST below
 - rotate right the subtree with root the parent of G (i.e. **H**)
 - rotate right the subtree with root the parent of G (i.e. **R**)
 - rotate left the subtree with root the parent of G (i.e. **E**)



This is called **root insertion** in BSTs

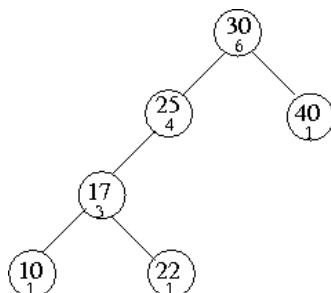
- actually means **leaf insertion** plus enough left/right rotations to get the leaf to the root
 - the structure can change a lot by doing this
- in a heap we inserted at the leaf and then did a *fix-up* towards the root
 - the structure did not change (CTP)

Useful?

- more recently inserted elements (i.e. active) will be close to the top
 - nodes around a newly inserted leaf nodes also move up the tree
- many applications have inherent *active* elements and *inactive* elements
 - performance with root insertion will be better

Counting the number of nodes in a subtree

How can we change our original BST structure to keep track of the number of nodes in each sub-tree?



BST with a node count in each node:

切换行号显示

```
1 typedef struct node *Tree;
2 struct node {
3     int data;
4     Tree left;
5     Tree right;
6     int count; 节点数
7 };
```

New nodes have a count of 1, so write:

切换行号显示

```
1 Tree createTree(int v) {
2     Tree t = malloc (sizeof(struct node));
3     if (t == NULL) {
4         fprintf(stderr, "Out of memory\n");
5         exit(1);
6     }
7     t->data = v;
8     t->left = NULL;
9     t->right = NULL;
10    t->count = 1;
11    return t;
12 }
13
14 int sizeTree(Tree t) {
15     int retval = 0;
16     if (t != NULL){
17         retval = t->count;
18     }
19     return retval;
20 }
```

20 }

Updating the counters

- Insertion
 - The new node is added as a leaf so it will always be initialised with a count of 1
 - Each node on search path will now have an extra node in their sub-tree

切换行号显示

```

1  Tree insertTree(Tree t, int v) {
2      if (t == NULL) {
3          t = createTree(v);
4      }
5      else {
6          if (v < t->data) {
7              t->left = insertTree (t->left, v);
8          }
9          else {
10             t->right = insertTree (t->right, v);
11         }
12         t->count++;    // update the counter at each ancestor
13     }
14     return t;
15 }

```

- Deletion
 - If the deleted node is a leaf, or has only 1 sub-tree, each node on the search path will have one less node in their sub-tree
 - If it has 2 sub-trees, it is a bit harder

Why have we included a count field in the BST?

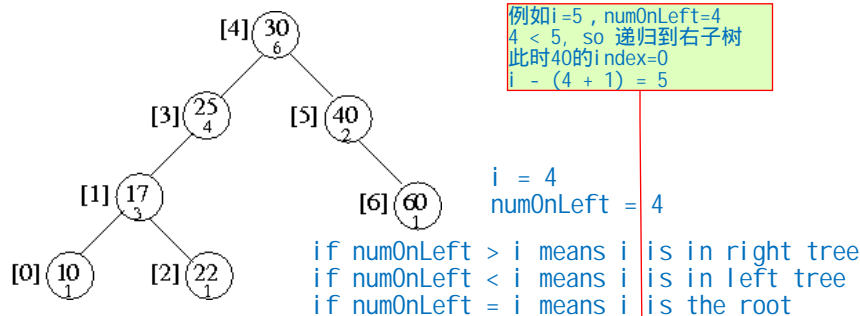
- so we can implement a select operation

Selecting the i -th element from a BST选择BST里面的第 i 个元素

Basic idea:

- check the number of nodes in the left subtree
 - if the left subtree contains $numOnLeft$ elements, and
 - $numOnLeft > i$, we recursively look in this left subtree for element i , else if
 - $numOnLeft < i$, we recursively look in the right subtree for element $i - (numOnLeft + 1)$, else if
 - $numOnLeft = i$, return with the current element's data

$numOnLeft$ 表示左子树的节点数目, 因为左子树的所有值都小于root



切换行号显示

```

1  // For tree with n nodes - indexes are 0..n-1
2  int selectTree(Tree t, int i) { // for tree with n nodes - indexed from 0..n-1
3      int retval = 0;
4      if (t != NULL) {
5          int numOnLeft = 0;    // this is if there is no left branch
6          if (t->left != NULL) {
7              numOnLeft = t->left->count; // this is if there is a left branch
8          }    计算左子树的数目
9          if (numOnLeft > i) {    // left subtree or ...
10             retval = selectTree(t->left, i);
11         }    如果左子树数目 > i, 则说明i在左子树里面
12         else if (numOnLeft < i) { // ... right subtree ?
13             retval = selectTree(t->right, i - (numOnLeft + 1));
14         }    如果左子树数目 < i, 则说明i在右子树里面, 新的子树重新计算
15         else {
16             retval = t->data;    // value index i == numOnLeft
17         }
18     }
19     else {

```

减掉左子树+root

```

20     printf("Index does not exist\n");
21     retval = 0;
22 }
23 return retval;
24 }
25 }

```

The select operation can be used as basis for a **partition** operation

- put the i^{th} element at the root of a BST
- see later this lecture for code
 - a key operation used in BST rebalancing

The shape of the BST affects the performance of search, insertion and deletions

- want the BST to be balanced to ensure $O(\log(n))$ performance
- *how do we do this?*
- (remember heaps are always balanced because of **CTP**)

Balanced Trees

Goal is to build BSTs of size N that have **guaranteed performance**:

- average case search performance $O(\log(N))$
- worst case search performance $O(\log(N))$

Previously, BSTs had:

- average case search performance $O(\log(N))$
- worst case search performance $O(N)$, which is terrible
- best case is always $O(1)$ (get lucky; search key is at the root node)

Perfectly balanced BSTs have:

- depth of $\log(N)$ 左子树与右子树的数目差为 -1, 0, 1
- for every node $|size(LeftSubtree) - size(RightSubtree)| < 2$

Over time, BSTs can become unbalanced because data input is never truly random.

To achieve **guaranteed performance**

- we do not need perfect BSTs of height $\log(N)$
- (near perfect) BSTs of height $< 2\log(N)$ would also ensure good performance

Approach 1: Global rebalancing

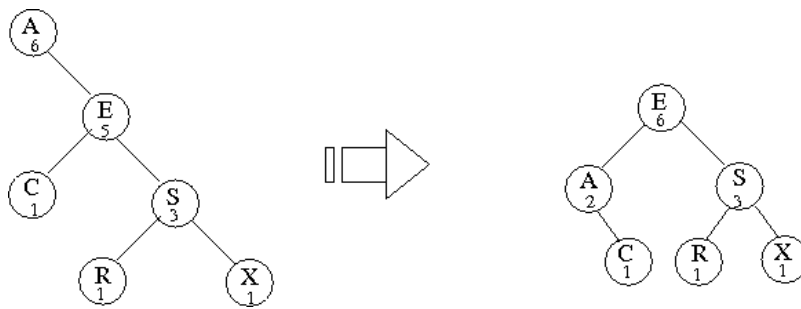
Insert nodes normally at leaves.

- Have a function to rebalance the whole tree. How?
 - Fact: *the median of a sequence of keys will partition the keys equally into left and right sub-trees*
- Basic Idea
 - Move the median to the root of the tree 将中间值调整为root
 - Recursively: 把左子树的中间值作为root的left节点
 - Get the median of the left sub-tree and move it to the root of the left sub-tree
 - Get the median of the right sub-tree and move it to the root of the right sub-tree 同理将右子树的中间值作为root的right节点
- This is **partitioning** the BST:
 - median will be the $N/2^{\text{th}}$ node
 - select the median
 - rotate it to the root

Rotating with a count

We first need to reconsider right and left rotation, but now with a *count* field in each node.

- Example of left rotation



切换行号显示

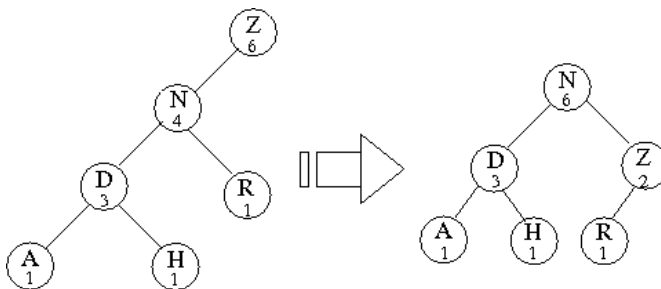
根节点A

```

1 Tree rotateLeft(Tree t) { // Rotate left code: includes count field
2   Tree retval = NULL;
3   if (t != NULL) {   nuroot为新的顶点, E
4     Tree nuroot = t->right; // left rotate: hence right root-child will
become new root E就是root的右节点
5     if (nuroot == NULL) {
6       retval = t; 如果右节点为NULL, 就没有必要旋转了, 因此直接返回t, root
7     }
8     else {
9       t->right = nuroot->left; // the left child of nuroot becomes old
root's right child 原root的右节点跟E的左节点相连, E的左节点与root相连
10      nuroot->left = t; // nuroot's left child is the old root
11      nuroot->count = t->count; // nuroot and old root have the same count
12      t->count = 1 + sizeTree(t->left) + sizeTree(t->right); // recompute
count in old root 将t->count赋值给新的nuroot, 然后计算t->count的值
13      retval = nuroot; // return with the new root
14    }
15  }
16  return retval;
17 }

```

- Example of right rotation



切换行号显示

```

1 Tree rotateRight(Tree t) { // Rotate right code: includes count field
2   Tree retval = NULL;
3   if (t != NULL) {
4     Tree nuroot = t->left; // 跟上面类似
5     if (nuroot == NULL) {
6       retval = t;
7     }
8     else {
9       t->left = nuroot->right;
10      nuroot->right = t;
11      nuroot->count = t->count;
12      t->count = 1 + sizeTree(t->left) + sizeTree(t->right);
13      retval = nuroot;
14    }
15  }
16  return retval;
17 }

```

Partitioning

分隔

We can rewrite `select()` to `partition` around the i^{th} element in the BST:

- we descend recursively
 - to the i^{th} element of each subtree
- rotate-with-count that element in the *opposite* direction to its position
 - if it is a left child: do a right rotation

- if it is a right child: do a left rotation
- this will make the i^{th} element the root of the BST

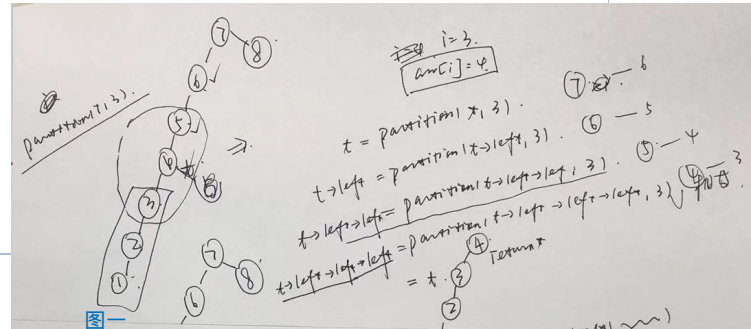
切换行号显示

将第i个元素作为root

```

1 Tree partition(Tree t, int i) { // make node at index i the root
2     Tree retval = NULL;
3     if (t != NULL) {
4         int numOnLeft = 0;
5         if (t->left != NULL) {
6             numOnLeft = t->left->count;
7         } 获取左子树的节点数
8         if (numOnLeft > i) { 左子树大于i的话, 应该要右转 一直运行到numOnLeft=i位置, 如下图, 就是4那个点, 左子树3个
9             t->left = partition(t->left, i); t->left是4那个点, 因此t就是5那个点, 然后旋转到图二, t为4返回
10            t = rotateRight(t); 返回上一个t->left, 因此t为6, 向右转到图三, t为4返回
11        } 就结束了
12        if (numOnLeft < i) {
13            t->right = partition(t->right, i-(numOnLeft+1));
14            t = rotateLeft(t);
15        }
16        retval = t;
17    }
18    else {
19        printf("Index does not exist\n");
20        retval = NULL;
21    }
22    return retval;
23 }

```



Balancing the BST

Move the median node to the root by partitioning on $i = \text{count}/2$

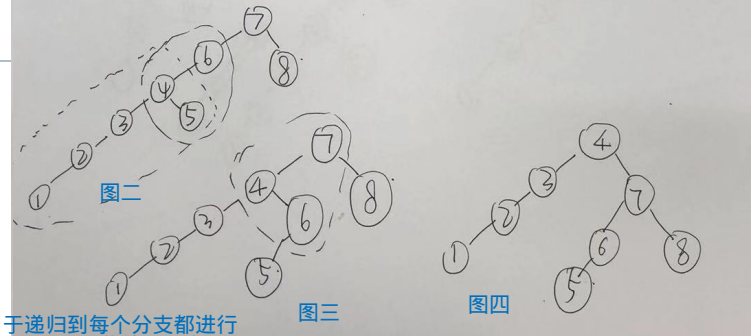
- balance the left sub-tree
- balance the right sub-tree

切换行号显示

```

1 Tree balance(Tree t) {
2     Tree retval = NULL;
3     if (t != NULL) {
4         if (t->count <= 1) {
5             retval = t;
6         }
7         else {
8             t = partition(t, t->count/2);
9             t->left = balance(t->left); 相当于递归到每个分支都进行
10            t->right = balance(t->right); partition, 因此就是所有balance
11        }
12        retval = t;
13    }
14    return retval;
15 }

```



这是 balance 效果

这是 partition(t, 8/2) 效果

Problems

- Cost of rebalancing is $O(n)$. (Bad news.)
- When do we rebalance? After every insert maybe??!!
 - Rebalance often – too expensive
 - Rebalance periodically, say:
 - after every 'k' insertions
 - or when the 'unbalance' exceeds some threshold
 - either way, we must tolerate worse search (i.e. $O(N)$) performance for periods of time
 - Does it solve the problem for dynamically changing trees? ... Not really.

由此可见, 虽然都是5作为顶点, partition只是移动balance做到了平衡。。。

Approach 2: Local Rebalancing

Local rebalancing is an *incremental* approach:

- the BST keeps itself 'balanced' as a 'side effect' of certain operations, such as insert and delete node
- the BST is said to be **self-balancing**
- in contrast with global rebalancing, which rebalances the whole BST, local rebalancing is incremental
- the aim is the same: avoid worst-case behaviour by reducing BST height to $\log(n)$

A **Splay tree** is a *locally re-balancing* BST

- the principle behind re-balancing is called **amortisation**

What is amortisation?

- In practice, an single operation may take $O(n)$ time, but ...
 - if you start with an empty tree, and it grows to size n nodes using k operations then
 - this will take $O(k \log(n))$
- In other words, over time:
 - some operations are 'slow' and may take up to linear time
 - other operations are 'fast' and take constant time
 - they balance out resulting in overall $\log(n)$ performance
- Note that at any given time during the k operations, the tree may not be perfectly balanced

A splay tree keeps recently-accessed elements near the top of the tree

- insertion, search and delete are done in $O(\log(n))$ **amortized** time


'Invented' by Sleator and Tarjan

- the fastest self-balancing BST data structure known
- the most popular data structure over the last 25 years
- used a lot in industry

In a splay tree, all operations will attempt to rebalance the BST:

- splaySearch(item)*: the item will be splayed to the top
 - even if the item is not found, the deepest item is splayed to the top
- splayInsert(item)*: the new item will be splayed to the top
- splayDelete(item)*: the parent of the item that replaces the deleted node is splayed to the top

Splay tree demo link

 Splay tree demo

Splaying an item to the top means to move an item to the root of the BST, using one of the operations:

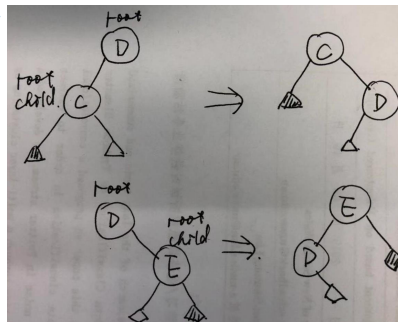
- zigzag** 之字形, 左右, 右左
将新插入的要素移动到root节点
- zigzig** 一条线, 左左, 右右

Both are based on a **zig operation** 左转, 右转

- move a root-child to the root position
- this is either a **rotate-right or rotate-left operation**

Terminology not consistent in the literature

- zig can mean left rotation **zig左转**
- zag can mean right rotation **zag右转**



but I will use *zigzag* to mean 'opposite' rotations, and *zigzig* to mean 'same' rotations

在本教程中, zigzag表示方向不同, 之字形
zigzig表示方向相同, 一条线那种

zigzag operation

This is used when:

先是左转parent / 然后再右转grandparent \

- the item is the right child of the parent, which is a left child of the grandparent
 - do a rotate-left of the parent followed by a rotate-right of the grandparent **左右型: 左左右转**
- the item is the left child of the parent, which is a right child of the grandparent
 - do a rotate-right of the parent followed by a rotate-left of the grandparent **右左型: 右转左转 先下后上**

zigzig operation

先是右转parent \ 然后再左转grandparent \

This is used when:

先是右转grandparent \ 然后再右转parent \

- the item is the left child of the parent, which is a left child of the grandparent
 - do a rotate-right of the **grandparent** followed by a rotate-right of the parent **左左型: 右转右转**
 - (not a rotate-right of the parent followed by a rotate-right of the grandparent **先上后下**)
- the item is the right child of the parent, which is a right child of the grandparent
 - do a rotate-left of the **grandparent** followed by a rotate-left of the parent **右右型: 左转左转**
 - (not a rotate-left of the parent followed by a rotate-left of the grandparent)

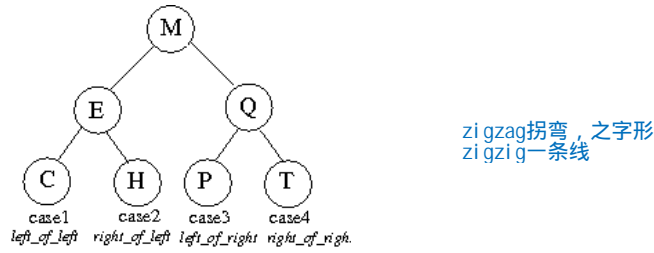
Splay(k): move a node k to the root through double rotation operations

两次操作, 先操作下面, 再上面

先是左转grandparent \ 然后再左转parent /

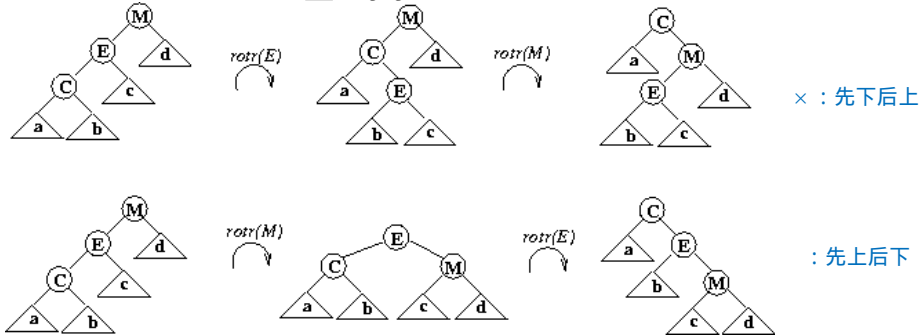
- a single rotation is not used in a splay tree
- most nodes halve their depth when a node is accessed (amortised)
译文：当访问一个节点时，大多数节点的深度减半(摊销)

Example



- *H* is the right child of the left child so requires a **zigzag** splay
 - this is simply a left-rotate followed by a right-rotate 左右型
- *P* is the left child of the right child so requires a **zigzag** splay 右左型
 - this is simply a right-rotate followed by a left-rotate
- *C* is the left child of the left child so requires a **zigzig** splay 左左型，先上后下
 - a **zigzig** splay does a right-rotate of the grandparent *M*, followed by another right-rotate of the parent *E*
- *T* is the right child of the right child so requires a **zigzig** splay 右右型，先上后下
 - a **zigzig** splay does a left-rotate of the grandparent *M*, followed by a left-rotate of the parent *Q*

The first double rotation below is not a **zigzig** rotation, the second is.



Inserting an item in a splay tree

Here is the code to insert an item *it*:

- it splays the item after it's been created as a leaf
 - to splay an item means to **zigzig/zigzag/zig** the item as necessary to get it to the root

切换行号显示

??? 没看懂这个Item

```

1 Tree splayInsertion(Tree t, Item it){
2 // multiple return here ONLY because I need the room for comments on the right
3   if (it == NULL){
4     return t;
5   }
6   if (t == NULL){
7     return createNode(it);
8   }
9   if (it->data < t->data) {
10    if (t->left == NULL) {
11      // ZIG
12      t->left = createNode(it);      // make a new link for this element
13      t->nNodes++;                  // increment the count at parent
14      return rotateRight(t);        // rotate parent
15    }
16    if (it->data < t->left->data) { // it < gparent & it < lparent
17      // ZIGZIG
18      t->left->left = splayInsertion(t->left->left, it);
19      t->left->nNodes++;            // incr lparent
20      t->nNodes++;                // incr gparent
21      t = rotateRight(t);         // ZIG: rotate gparent
22    } else {                      // it < gparent & it >= lparent
23      // ZIGZAG
24      t->left->right = splayInsertion(t->left->right, it);
25      t->left->nNodes++;            // incr lparent
26      t->nNodes++;                // incr gparent
27      t->left = rotateLeft(t->left); // ZAG: rotate lparent
28    }
  }
}

```

```

29     return rotateRight(t);           // ZIG: rotate gparent
30 } else {
31     if (t->right == NULL) {
32         // ZIG
33         t->right = createNode(it);    // analogous to the left case above
34         t->nNodes++;
35         return rotateLeft(t);
36     }
37     if (it->data < t->right->data) { // it > gparent & it < rparent
38         // ZIGZAG
39         t->right->left = splayInsertion(t->right->left, it);
40         t->right->nNodes++;           // incr rparent
41         t->nNodes++;                 // incr gparent
42         t->right = rotateRight(t->right); // ZAG: rotate rparent
43     } else {                       // it > gparent & it >= rparent
44         // ZIGZIG
45         t->right->right = splayInsertion(t->right->right, it);
46         t->right->nNodes++;           // incr rparent
47         t->nNodes++;                 // incr gparent
48         t = rotateLeft(t);         // ZIG: rotate gparent
49     }
50     return rotateLeft(t);           // ZIG: rotate gparent
51 }
52 }

```

- the 'zigs' and 'zags' correspond to the rotations on [wikipedia's](#) description of splay trees.
- the use of *zigzig* rules is crucial to the balancing of the BST.
 - in general it can be proved that:

a node on the search path at a depth d will move to a final depth of $\leq 3 + d/2$

after an insertion.

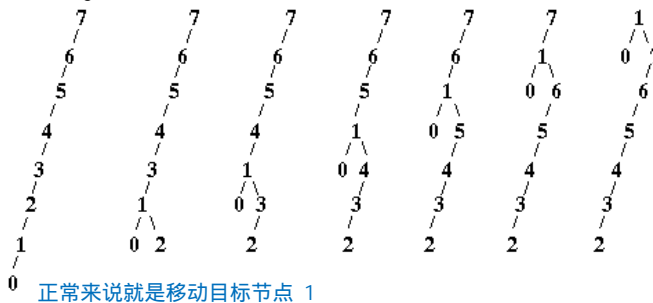
We now list a number of operations on splay trees and show examples.

- note that in every operation, some item is splayed

Operation splay-search

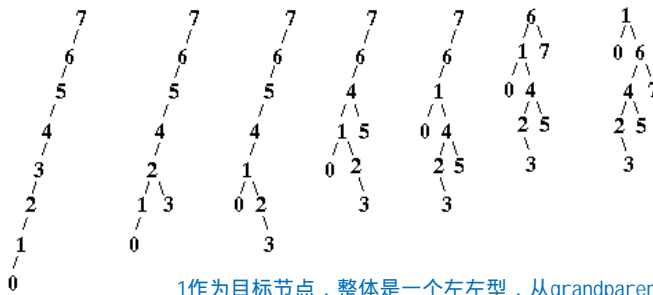
If we *search* for an item, and move the item to the root by using normal left and right rotations then there is no rebalancing

- for example: a normal search for item 1 would result in



If we *splay-search* for an item, once found we

- splay the item to the root

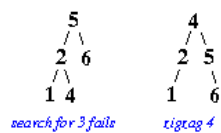


If the item is not found

- the last item (leaf) on the path is splayed to the root

For example, in the following splay tree we search for item 3, and because it fails, the 'last' item accessed, which is 4, is splayed:

找3，结果找到了4，那就移动4，就是判断点是否为叶子节点，是的话就动



Operation splay-insertion

We have seen 2 types of BST insertion, **leaf insertion and root insertion**. Assume that we wish to insert the sequence of numbers 3, 2, 4, 5, 1.

- **leaf insertion** 与前面学的一致

- each node is simply added as a leaf

input node splay insertion

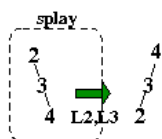
3

3

2



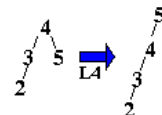
4



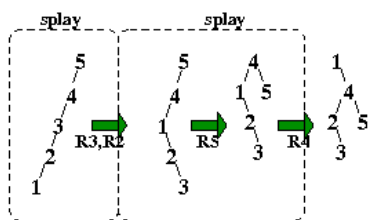
从上到下

按照前面学习的规律

5



1



- **root insertion**

- each node is added as a leaf, and then
- that leaf is promoted to the root position rotating the parent node to the left or right at each step

input node root insertion

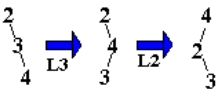
3

3

2

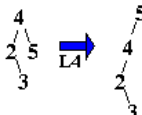


4

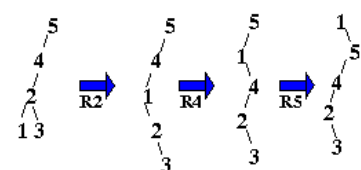


从下到上

5



1

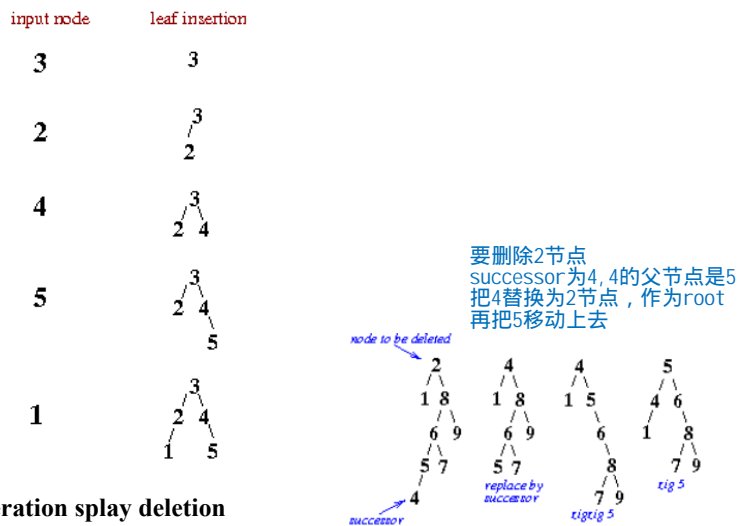


一个一个变化从下往上

We saw the code for splay insertion above. The rotations are different than with root insertion:

- each node is still first added as a leaf, and then
- that leaf is splayed to the node of the BST: rotating the grandparent and parent left-left or right-right whenever it can

An example of splay insertion for the same sequence as above is:



Operation splay deletion

Normally, when an item is deleted in a BST, the node is simply replaced by its **DLMD** predecessor or **DRMD** successor.

In *splay-deletion*, if you wish to delete a node y :

- delete the node y as normal using either:
 - DRMD: if the predecessor of y is x , and the parent of x is p , then
 - then follow the deletion by splaying p
 - DLMD: if the successor node of y is z , and the parent of z is p , then
 - then follow the deletion by splaying p

For example, let us splay-delete item 2 from the following splay tree

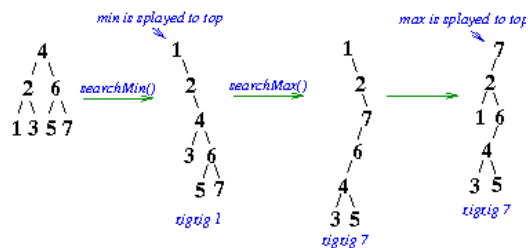
- we will use DLMD (DRMD would be similar)
- splaydeletion.png

Minimum or maximum of a splay tree

- search for the minimum or maximum item in a BST
- splay the item to the root

For example, we show below a splay tree after

- a search for the minimum and then
- a search for the maximum item



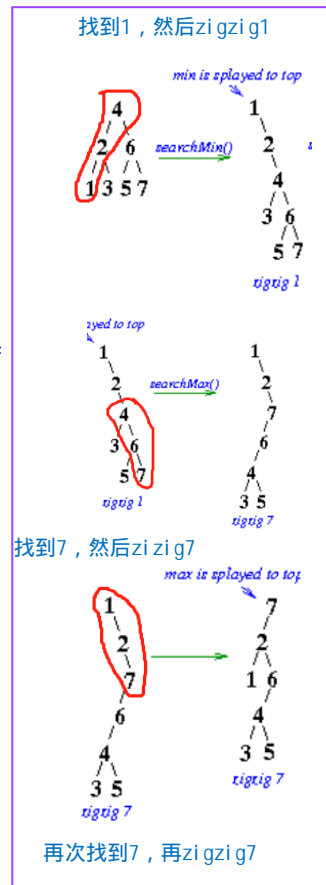
in summary

We note that in each operation:

- search for an item in a splay tree
 - the found item is splayed
 - if not found, the last accessed item is splayed
- insertion of an item in a splay tree
 - the new item is splayed
- deletion of an item in a splay tree
 - the parent of the item that replaces the deleted item is splayed
- search for the minimum/maximum item in a splay tree
 - the found minimum/maximum is splayed

A 'splay' of an item involves:

- zigzig whenever the item is left-left or right-right 从上到下
- zigzag whenever the item is left-right or right-left 从下到上
- zig whenever the item is the child of the root (so a zigzag or zigzig is not possible)
不满足上面两个的时候



Splay Tree Analysis

- number of comparisons per operation is $O(\log(n))$
- gives good (amortized) cost overall
- no guarantee for any individual operation: worst-case behaviour may still be $O(N)$

2-3-4 trees

(Chapter 13.3 Sedgwick)

Local balancing approaches:

- splay trees: self-balancing, generally improved performance ...
 - ... but worst-case behaviour $O(n)$

Is there a search tree that is guaranteed to have $O(\log(N))$ behaviour for insertion and search?

- yes, **2-3-4 trees**

2-3-4 trees

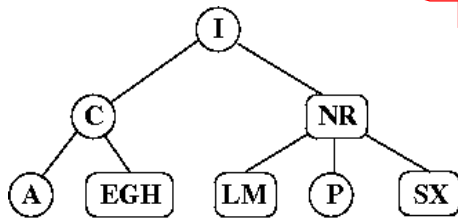
- is self-balancing
- is commonly used to implement dictionaries
- has the property that all external nodes are at the same depth 🧠
- generalises the concept of a node
 - 3 types of nodes:
 - **2nodes**: 1 key, a left link to smaller nodes, a right link to larger nodes (this is the normal node)
 - **3nodes**: 2 keys, a left link to smaller, a middle link to in-between, and right link to larger
 - **4nodes**: 3 keys: links to the 4 ranges

里面只有一个key就是一个字母，类似二叉树，因此只存在大小之分

里面两个字母，小于，中间，大于三部分

Example: 所有的叶子节点都有相同的深度

三个字母，M O S 分成四个部分



In the above example:

- 'A' is an example of a 2node
- 'LM' of a 3node
- 'EGH' of a 4node

Example of a search, for 'P':

https://www.cnblogs.com/gaochundong/p/balanced_search_tree.html#two_three_four_tree

- start at the root 'I'
- larger, go right to 'NR'
- middle, go middle to 'P'

Examples of insertions:

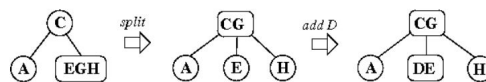
- 'B': easy, change node 'A' into 'AB'
- 'O': easy, change node 'NR' into 'NOR'
- 'D': the node is full ??

4node insertions

- split the 4node: 分裂4度节点
 - left key becomes a new 2node 左节点变成一个新的2度节点
 - middle key goes up to parent 中间节点上提到parent里面
 - right node becomes a new 2node 右节点变成一个新的2度节点
- add new key to one of the new 2nodes, creating a 3node

Example: insert 'D' in a 2-3-4 tree

insertion234.png



This insertion assumes that you can move the 'middle' key up to the parent.

What happens if the parent is a 4node, and its parent is a 4node, and its parent ... ?

Solution, during insertion, *on the way down*:

- [A] transform 2node \rightarrow 4node into 3node \rightarrow (2node+2node)
- [B] transform 3node \rightarrow 4node into 4node \rightarrow (2node+2node)
- [C] if the root is a 4node, transform into 2node \rightarrow (2node+2node)

Example:


transform234.png

Basic insertion strategy:

- because of the transformations (splits) on inserts *on the way down*, we often end on a 2node or 3node, so easy to insert
- insert at the bottom:
 - if 2node or 3node, simply include new key
 - if 4node, split the node 4node需要将中间节点上调，上调后parent又需要上调，以此类推
 - if the parent is a 4node, this needs to be split
 - if the grandparent is a 4node, this needs to be split
 - ...

Results:

- trees are
 - *split* from the top down
 - (*split* and) *grow* from the bottom up
- **after insertions or deletions, 2-3-4 trees remain in perfect balance**
- *What do you notice about these search trees?*

 2-3-4 tree animation

Implementation

2-3-4 trees are actually implemented using BSTs!

- 3nodes and 4nodes are written as 2nodes with 1 or 2 children
- there are 2 kinds of links: *red* and *black*
 - *red* special internal links to signify a 3node or 4node
 - *black* normal links between nodes
- red-black BSTs implement 2-3-4 trees