Advanced Data Structure

Prolegomenon

Grading Policies

Homework(10)
Discussions or quiz(10)
Research Project(30)

> done in groups of 3
> choose 2 out of 7
> Report(20+20)/2 points
> In-class presentation(10~15minutes,10 points)
> The speaker will be chosen randomly
> E-mail to sign up for presentation
> Bonus

MidTerm(10)
Final Exam(40)

> You can replace the MidTerm by a higher grade in Final Exam

- E-mail denghaoran@zju.edu.cn before 6th March to send your group.

AVL Trees,Splay Trees and Amortized Analysis

AVL Tree

Target

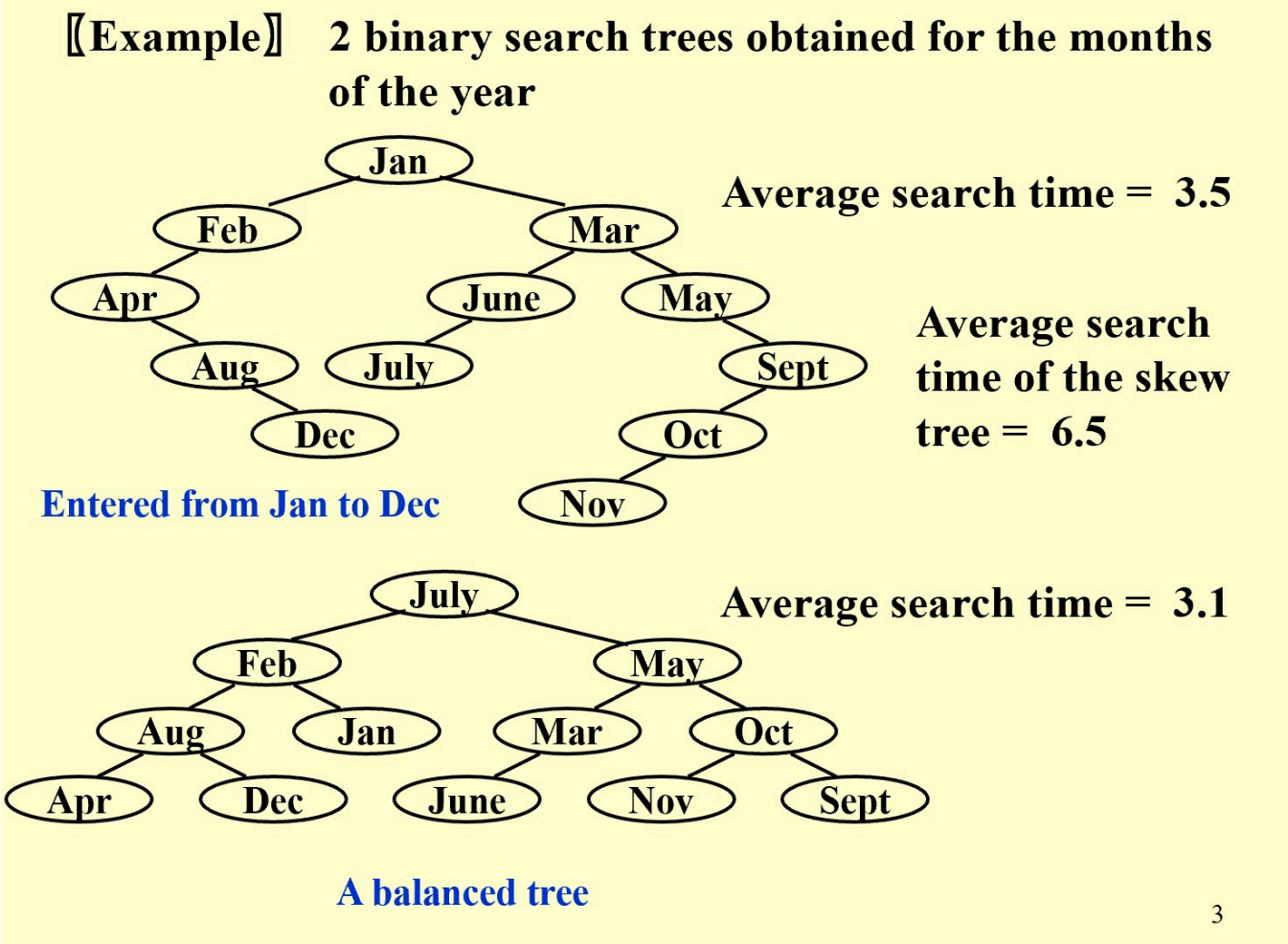
speed up searching(with insertion and deletion)

Recall:To solve this problem,the time complexity is:

- Array: $O(N)$ (Search) ; $O(1)$ (Insert)

- Sorted Array: $O(logN)$; $O(N)$ (Insert)
- Binary Sorted Tree: $O(height)$; $O(height)$(Insert)
  In the worst case,the $O(height)$ is $O(N)$

Optimize:Balanced Tree



Definition

Balanced Tree

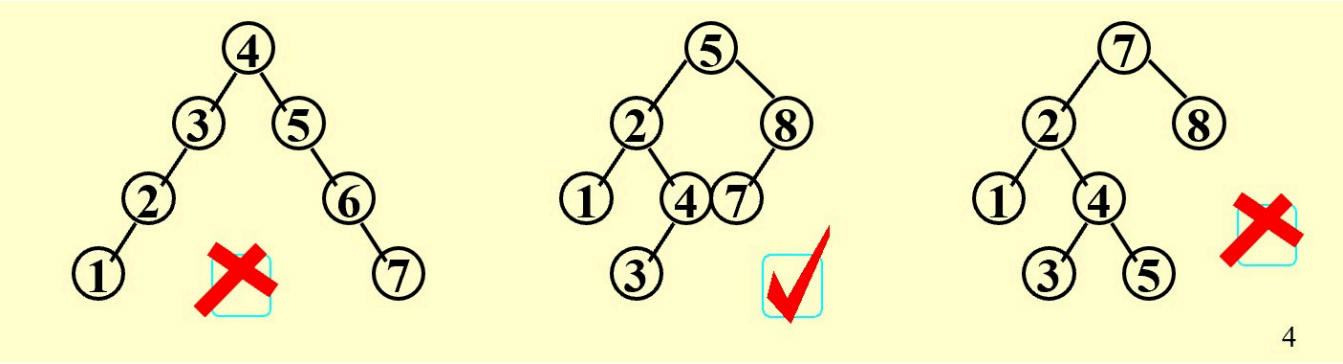
An empty binary tree is height balanced. If T is a nonempty binary tree with T_L and T_R as its left and right subtrees, then T is height balanced iff

1. T_L and T_R are height balanced
2. $|h_L - h_R| \leq 1$ where h_L and h_R are the heights of T_L and T_L , respectively.

Balance Factor(BF)

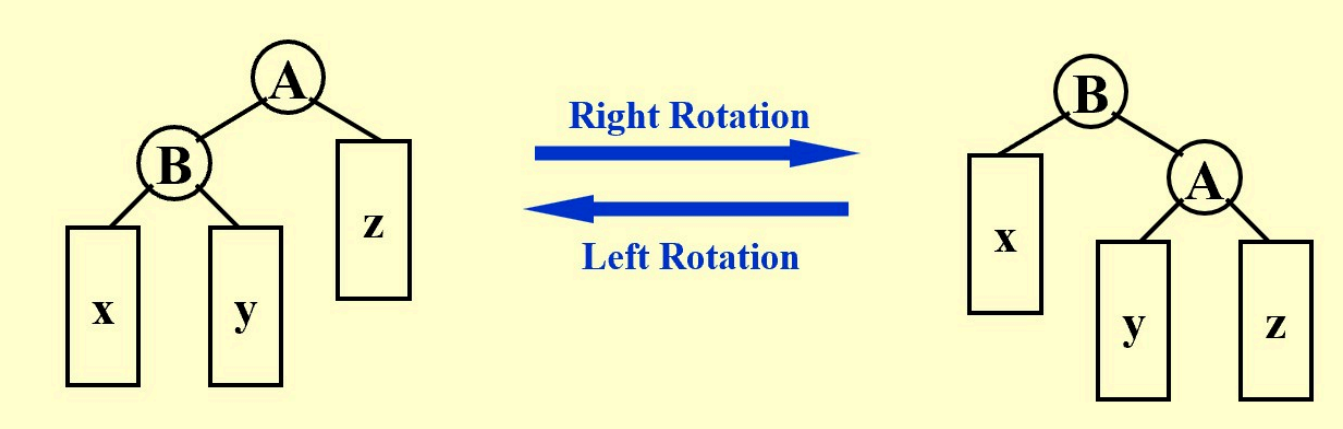
The Balance Factor(BF) of a node=$h_L - h_R$.

In an AVL Tree ,$BF(node) = -1, 0, or 1.$



Tree Roation

1. **Tree Rotation** is an operation on a binary tree that changes the suructure without interfering with the order of elements.



2. After a rotation,the side of rotation increase its height by 1 whilst the side opposite the rotation decreases its height similarly.

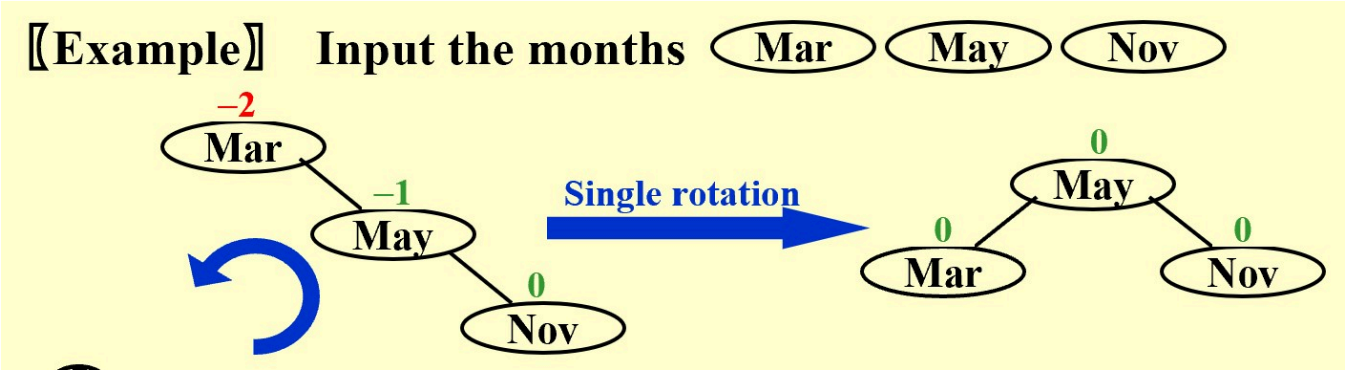
pseudo code of Left Rotation

```
Tree Temp = A->Left
A->Left = B
B->Right = Temp
```

3. The time complexity is $O(1)$.

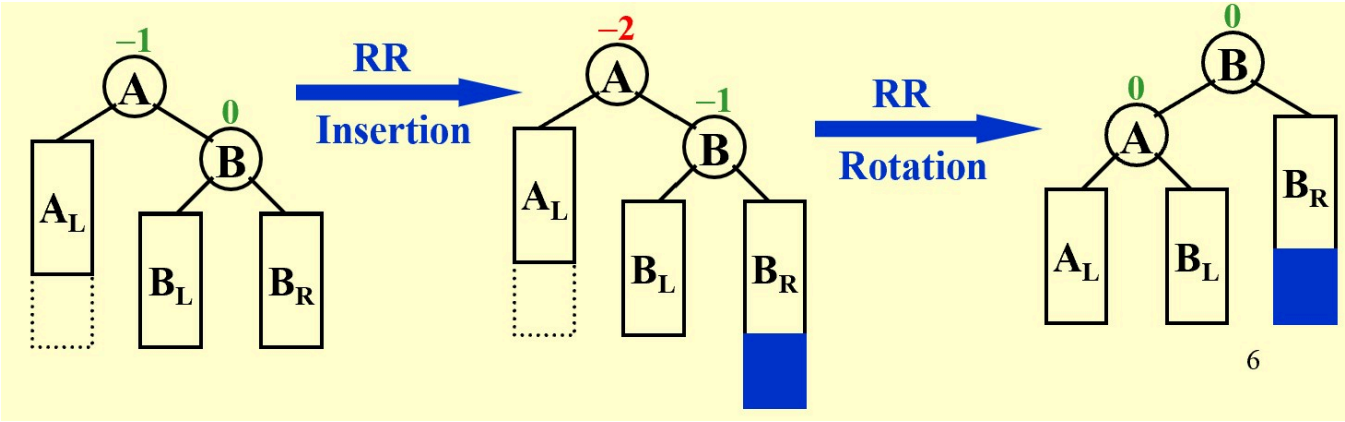
Insertion of an AVL Tree

RR Rotation



RR Rotation:

In General:

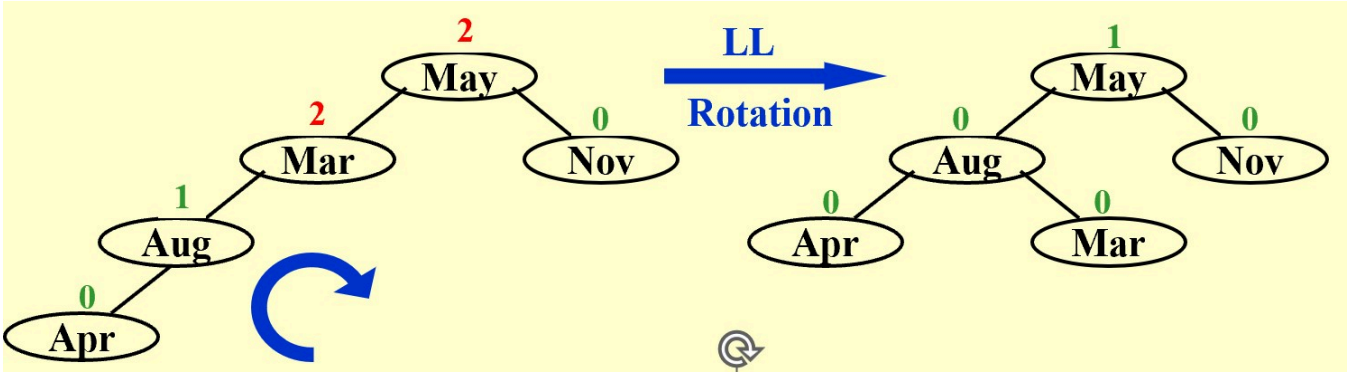


Note:

A is **NOT** necessarily thr root of the tree.**A is the first Node that has a wrong BF after the insertion**.

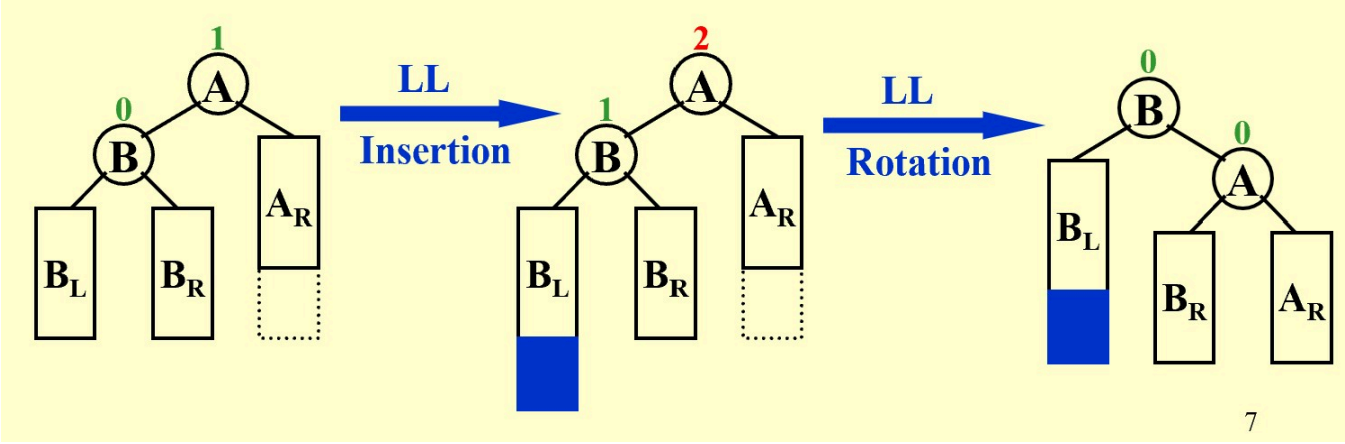
The insertion could be just happened in the right subtree of B **but NOT necessarliy be the right child of B**.

LL Rotation

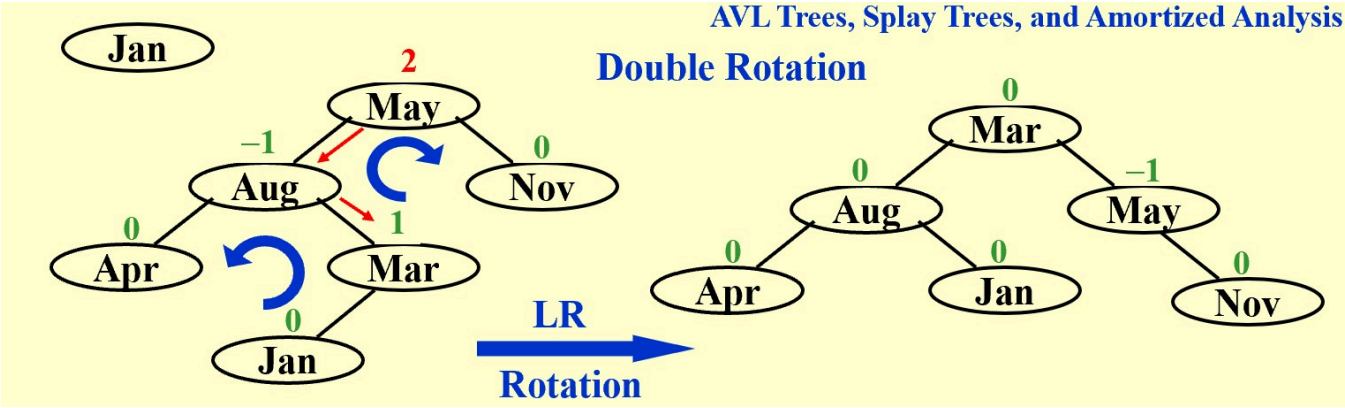


Mind the NOTE!

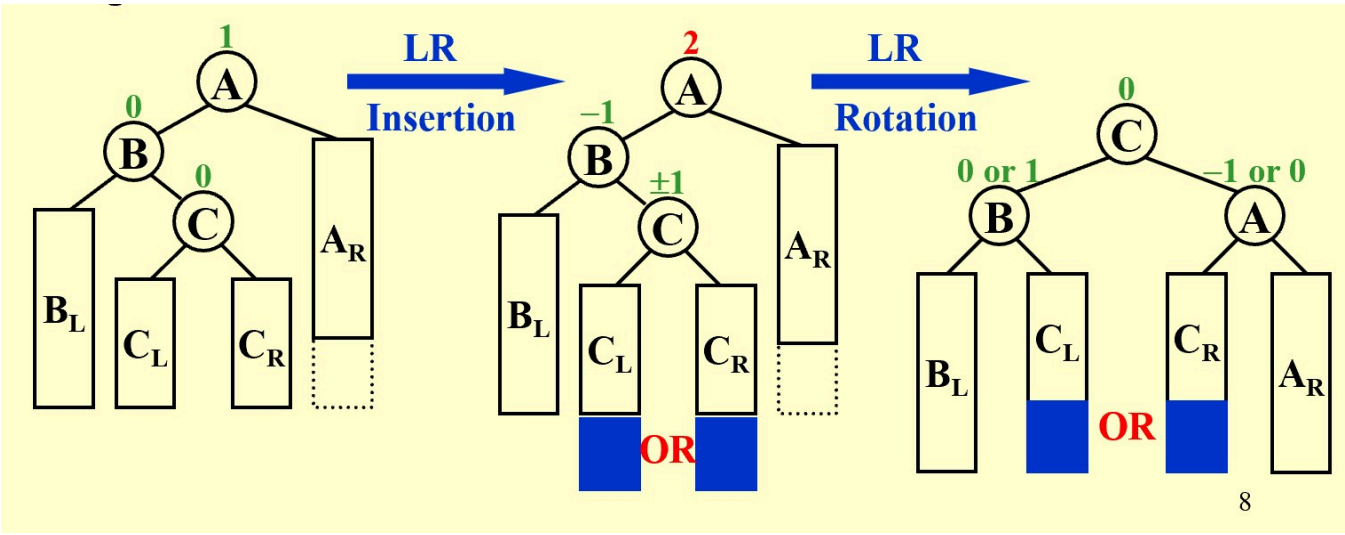
In General:



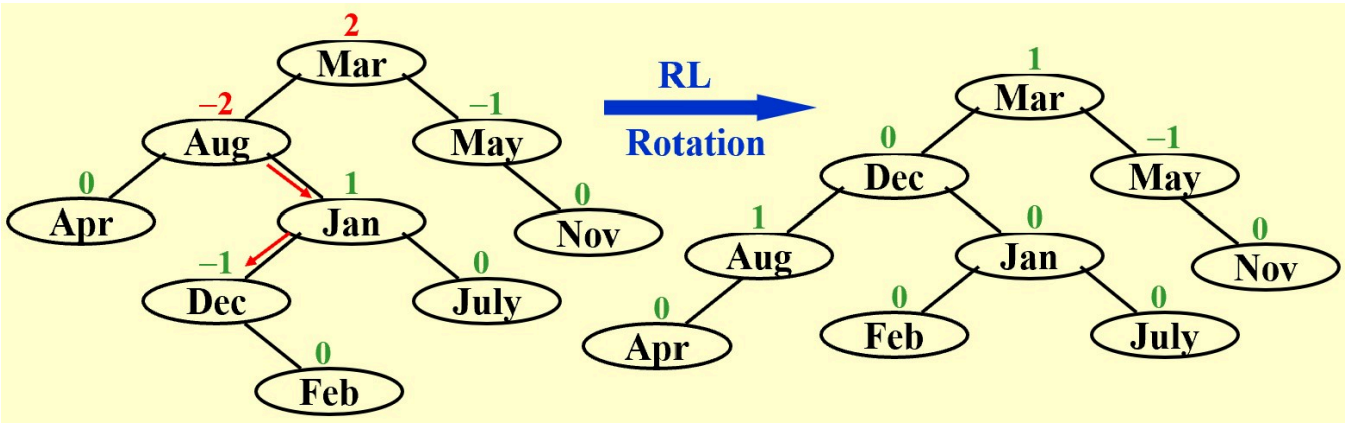
LR Rotation



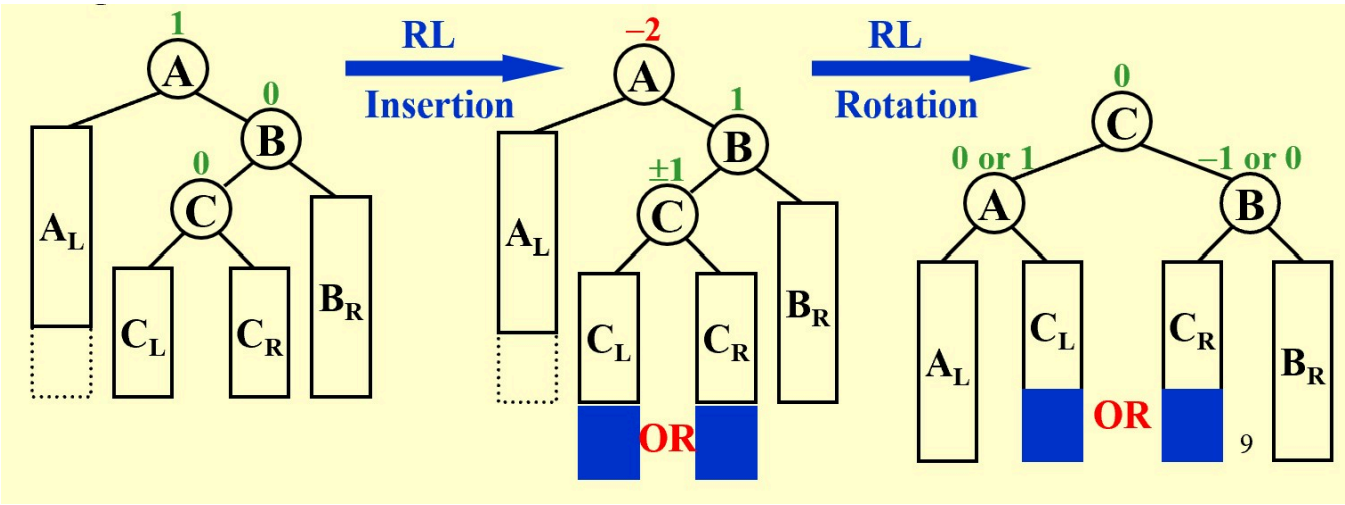
In General:



RL Rotation



In General:

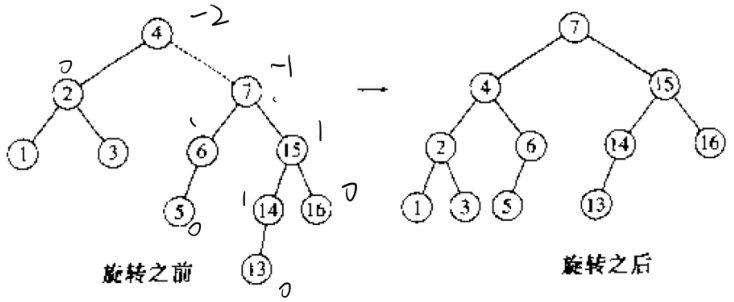


注意：

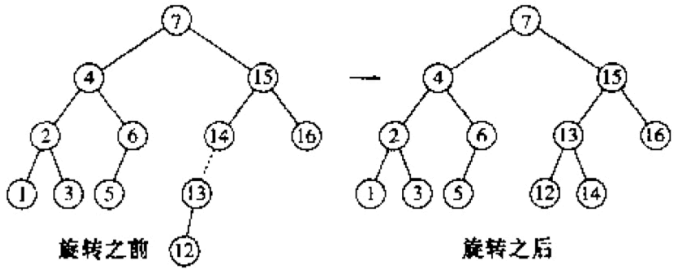
单旋转(Single Rotation)发生在要插入的值**不在**关键边(连接BF值产生异常的点和其相邻点的边)

之间。

如果现在插入 13，那么在根处就会产生一个不平衡。由于 13 不在 4 和 7 之间，因此我们知道一次单旋转就能完成修正的工作。



12 的插入也需要一个单旋转：

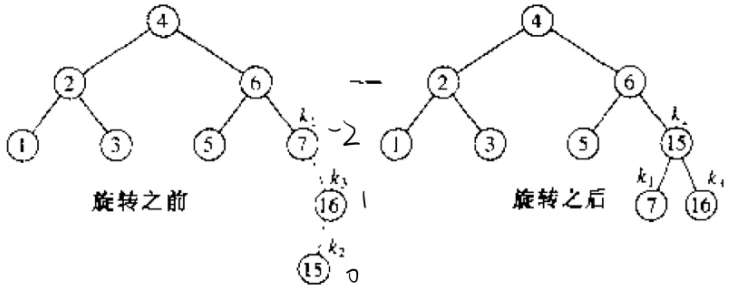


图中虚线的边即为关键边，图中待插入的13不在关键边的4 ~ 7之间；12不在关键边13 ~ 14之间。

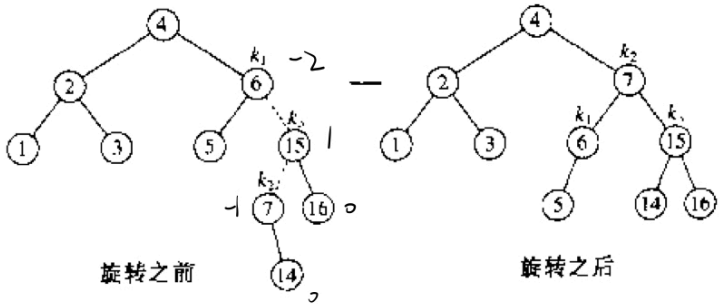
双旋转(Double Rotation)发生在要插入的值**在**关键边的值**之间**，并分为RL旋转与LR旋转。

旋转时k_2成为最后的根节点，k_2左子树给k_1作为右子树，k_2右子树给k_3作为左子树。k_1、k_3分别作为k_2的左右子树。

这属于情形 3，需要通过一次右-左双旋转来解决。在我们的例子中，这个右-左双旋转将涉及 7，16 和 15，此时，k_1 是具有关键字 7 的节点，k_3 是具有关键字 16 的节点，而 k_2 是具有关键字 15 的节点。子树 A、B、C 和 D 都是空树。



下面我们插入 14，它也需要一个双旋转。此时修复该树的双旋转还是右-左双旋转，它将涉及 6、15 和 7。在这种情况下，k_1 是具有关键字 6 的节点，k_2 是具有关键字 7 的节点，而 k_3 是具有关键字 15 的节点。子树 A 的根在关键字为 5 的节点上，子树 B 是空子树，它是关键字 7 的节点原先的左儿子，子树 C 置根于关键字 14 的节点上，最后，子树 D 的根在关键字为 16 的节点上。



图中BF值出现异常的点为k_1，与其相连的点为k_3，判断是LR旋转还是RL旋转的点为k_2。

Reference

> Read the declaration and functions in [1] Figures 4.42 – 4.48

Algorithm Analysis

n_h=F_n-1
n_h->$((1+\sqrt{5})/2)^h$
h->$O(ln(N))$

Splay Tree

Target

Any M consecutive tree opertaions starting from an empty tree take at most $O(MlogN)$ time.

Insertion of a Splay Tree

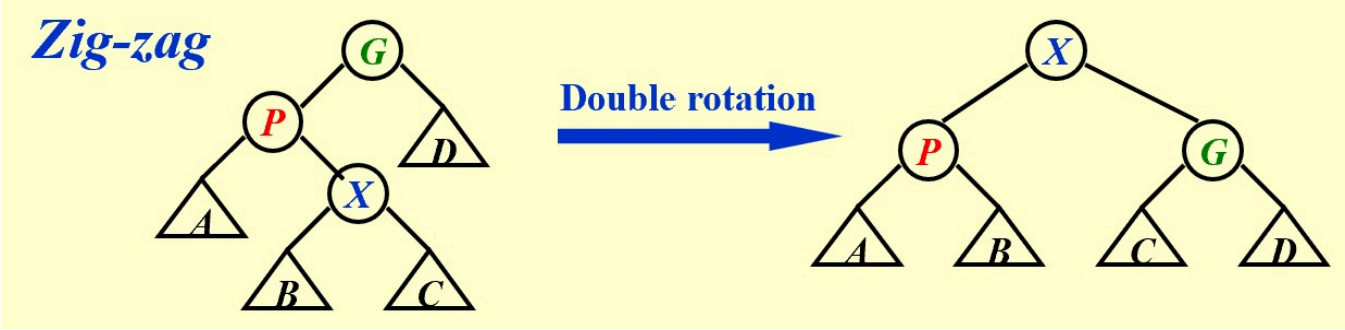
X:a Node that is NOT root
P:parent of X
G:grandParent of X

Case1:P is the root

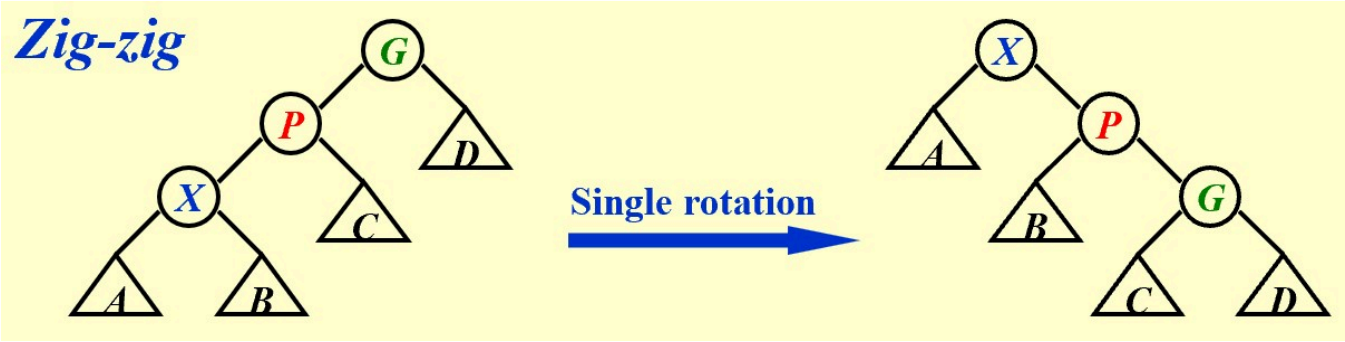
Rotate X and P

Case2:P is not the root

Zig-zag:Double Rotation



Zig-zig:Single Rotation



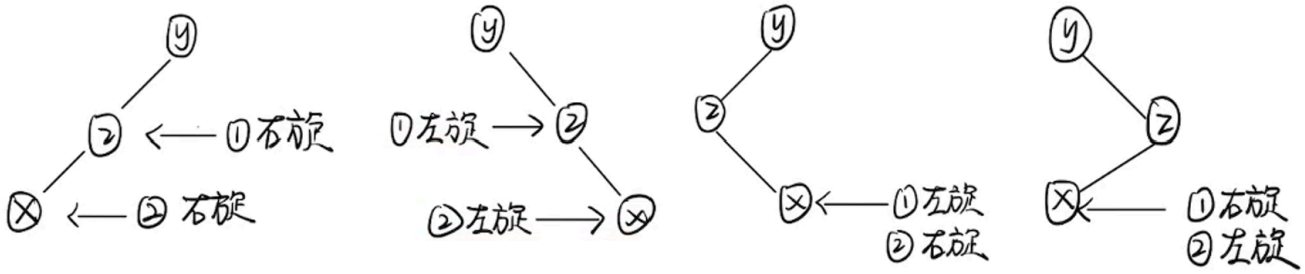
Reference

> Read the 32-node example given in Figures 4.52 – 4.60

Note

四种旋转情况的总结（适用于选择题画图）

其中①②表示操作的顺序



Deletion of a Splay Tree

1. Find(X) *The Node X will be at root.*
2. Remove(X) *There will be two subtrees* T_L *and* T_R*.*
3. FindMax(T_L) *The Largest element will be the root of* T_L*, and has NO right child.*

Appendix：Implementation in C(AVL Tree,Splay Tree)

详见`Project1.c`

其中，AVL Tree的删除与普通二叉搜索树相同，为`Delete()`函数。

Splay Tree相关部分需要先初始化`NullNode`变量。

Amortized Analysis(均摊分析)

Recall

The target of a Splay Tree: Any M consecutive tree opertaions starting from an empty tree take at most $O(MlogN)$ time.

--**Amortized time bound**

Aggregate analysis(聚合分析)

Idea

Show that for all n, a sequence of n operations takes worst-case time T(n) in total. In the worst case, the average cost, or amortized cost, per operation is therefore $T(n)/n$.

【**Example**】Stack with `MultiPop(int k,Stack S)`

```
Algorithm  {
    while ( !IsEmpty(S) && k>0 ) {
        Pop(S);
        k - -;
    } /* end while-loop */
}
```

Consider a sequence of n `Push`, `Pop`,and `MultiPop` operations on an initially empty stack. Because we can pop each object from the stack at most once for each time we have pushed it onto the stack.

$sizeof(S) \leq n$

$T_{amortized} = O(n)/n = O(1)$

Accounting method(信用方法)

Idea

When an operation's amortized cost exceeds its actual cost, we assign the difference to specific objects in the data structure as credit. Credit can help pay for later operations whose amortized cost is less than their actual cost.

Note

For all sequences of n operations, we must have:

$$T_{amortized} = \frac{\sum_{i=1}^{n} \hat{c}_i \geq \sum_{i=1}^{n} c_i}{n}$$

【**Example**】Stack with `MultiPop(int k,Stack S)`

```
    while ( !IsEmpty(S) && k>0 ) {
        Pop(S);
        k - -;
    } /* end while-loop */
}
```

Consider a sequence of n `Push`, `Pop`,and `MultiPop` operations on an initially empty stack.

c_i for Push: 1 ; Pop: 1 ; and MultiPop: min (sizeof(S), k)

$\hat{c}_i$ for Push: 2 ; Pop: 0 ; and MultiPop: 0

Starting from an empty stack —— *Credits* for

Push: +1 ; Pop: –1 ; and MultiPop: –1 for each +1

sizeof(S) ≥ 0 ➡ *Credits* ≥ 0

$$\Rightarrow O(n) = \sum_{i=1}^{n} \hat{c}_i \geq \sum_{i=1}^{n} c_i$$

$$\Rightarrow T_{amortized} = O(n)/n = O(1)$$

Potential method(势能方法)

Idea

Take a closer look at the credit.

为了避免给每一种操作都设计一个均摊代价，我们设计一个势能函数来统一化设计均摊代价。

$$\hat{c}_i - c_i = Credit_i = \Phi(D_i) - \Phi(D_{i-1})$$

Potential function

$$\sum_{i=1}^{n} \hat{c}_i = \sum_{i=1}^{n} \left(c_i + \Phi(D_i) - \Phi(D_{i-1})\right)$$

$$= \left(\sum_{i=1}^{n} c_i\right) + \underbrace{\Phi(D_n) - \Phi(D_0)}_{\geq 0}$$

In general, a good potential function should always **assume its minimum** at the start of the sequence.

【**Example0**】Stack with `MultiPop(int k,Stack S)`

```
    while ( !IsEmpty(S) && k>0 ) {
        Pop(S);
        k - -;
    } /* end while-loop */
}
```

D_i = the stack that results after the i-th operation

$\Phi(D_i)$ = the number of objects in the stack D_i

$$\Phi(D_i) \geq 0 = \Phi(D_0)$$

Push: $\Phi(D_i) - \Phi(D_{i-1}) = (sizeof(S)+1) - sizeof(S) = 1$

$\Rightarrow \hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = 1+1 = 2$

Pop: $\Phi(D_i) - \Phi(D_{i-1}) = (sizeof(S)-1) - sizeof(S) = -1$

$\Rightarrow \hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = 1-1 = 0$

MultiPop: $\Phi(D_i) - \Phi(D_{i-1}) = (sizeof(S)-k') - sizeof(S) = -k'$

$\Rightarrow \hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = k'-k' = 0$

$$\sum_{i=1}^{n} \hat{c}_i = \sum_{i=1}^{n} O(1) = O(n) \geq \sum_{i=1}^{n} c_i \Rightarrow T_{amortized} = O(n)/n = O(1)$$

【**Example1**】 Splay Trees: $T_{amortized}$ = $O(logN)$

$\Phi(T) = \Sigma_{i \in T} ln(S(i))$

$S(i)$:以`x`为根的子树中的节点个数。也叫做子树的质。

D_i = the root of the resulting tree

$\Phi(D_i)$ = must increase by at most $O(\log N)$ over n steps, AND will also cancel out the number of rotations (*zig*:1; *zig-zag*:2; *zig-zig*:2).

$$\Phi(T) = \sum_{i \in T} \log S(i)$$ **where $S(i)$ is the number of descendants of i (i included).**

Appendix:Math

If $a + b \leq c$, and a and b are both positive integers, then:

$$loga + logb \leq 2logc - 2$$

Proof:

By the arithmetic-geometric mean inequality,

$$(ab)^{1/2} \leq (a+b)/2$$

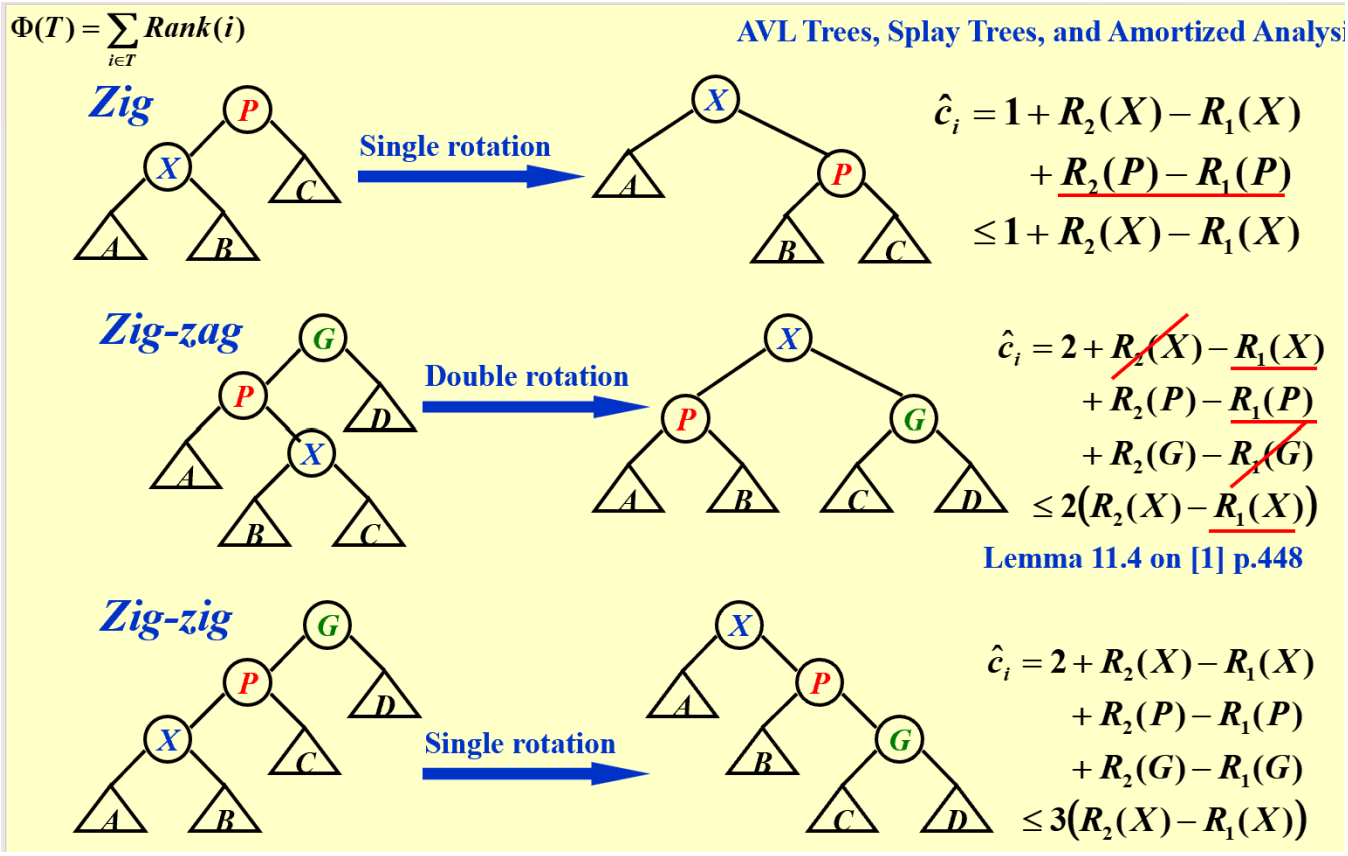
Thus

$$(ab)^{1/2} \leq c/2$$

Squaring both sides gives

$$ab \leq c^2/4$$

Taking logarithms of both sides proves the lemma.



其中，`Zig-zag`和`Zig-zig`都是双旋转，开销为2。

`Zig-zag`中：

$R_2(X)$和$R_1(G)$都是整棵树的质，故相互抵消。$R_1(P) < R_1(X)$,故放缩。

`Zig-zig`中：

$R_1(X) + R_2(G) \le 2R_2(X) - 2$

$2 + R_2(X) - R_1(X) + R_2(P) - R_1(P) + R_2(G) - R_2(G) \le 2R_2(X) -$

$2R_1(X) + R_2(P) - R_1(P) \le 3(R_2(X) - R_1(X))$

Red-Black Trees and B+ Trees

Red-Black Trees

Target

Balanced binary seach tree using a key called *color*.

Definition

Red-Black Tree

红黑树是一种含有红黑结点并能自平衡的二叉查找树。它必须满足下面性质：

- 性质1：每个节点要么是黑色，要么是红色。
- 性质2：**根**节点是**黑色**。
- 性质3：每个**叶子**节点（NIL）是**黑色**。(所有叶子节点**都是哨兵**，哨兵都是**黑色**的)
- 性质4：每个**红色**结点的两个子结点一定都是**黑色**。
- 性质5：任意一结点到每个叶子结点的**简单路径**都包含**数量相同**的**黑**结点。

性质3存在的意义：
为了保证不存在奇怪的红黑树，所有**孩子为空的节点**都必须是**哨兵**，是**黑色**的，这样极度不平衡的树会**不满足**性质5。

black-height

The black-height of any node x,denoted by `bh(x)`,is the number of **black** nodes on any simple path from x(**x not included**) down to a leaf.`bh(Tree)=bh(root)`.

【Lemma】
A red-black tree with N internal nodes has height at most $2ln(N+1)$.

Proof：

Lemma1: For any node x, $sizeof(x) \geq 2^{bh(x)}-1$.
If $h(x)=0$, x is `NULL` -> $sizeof(x)=2^0-1=0$
Suppose it is true for all x with $h(x) \leq k$.
For x with $h(x)=k+1$, $bh(child)=bh(x)\ or\ bh(x)-1$
Since $h(child) \leq k$, $sizeof(child) \geq 2^{bh(child)}-1 \geq 2^{bh(x)-1}-1$
Hence $sizeof(x)=1+2sizeof(child) \geq 2^{bh(x)}-1$

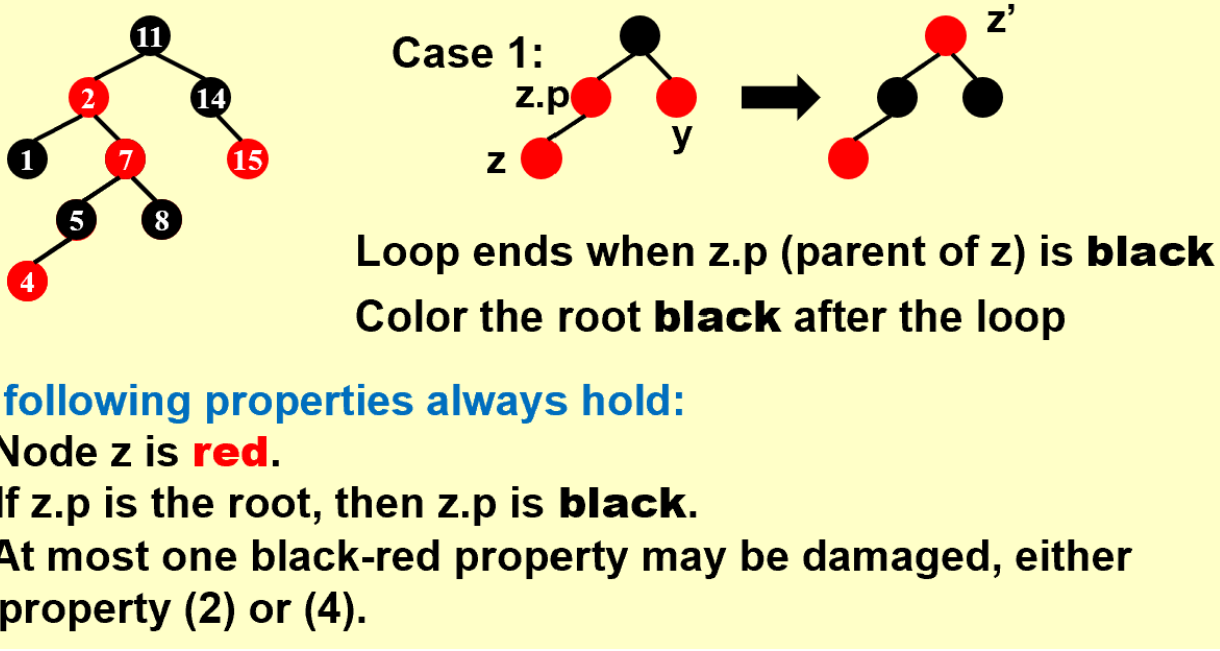
Lemma2: $bh(Tree) \geq h(Tree)/2$
Since for every red node, both of its children must be black, hence on any simple path from *root* to a leaf, at least half the nodes (*root* not included) must be black.
$Sizeof(root)=N \geq 2^{bh(Tree)}-1 \geq 2^{h/2}-1$

由**Lemma1**和**Lemma2**可以得到**Lemma**的结论。

Insert

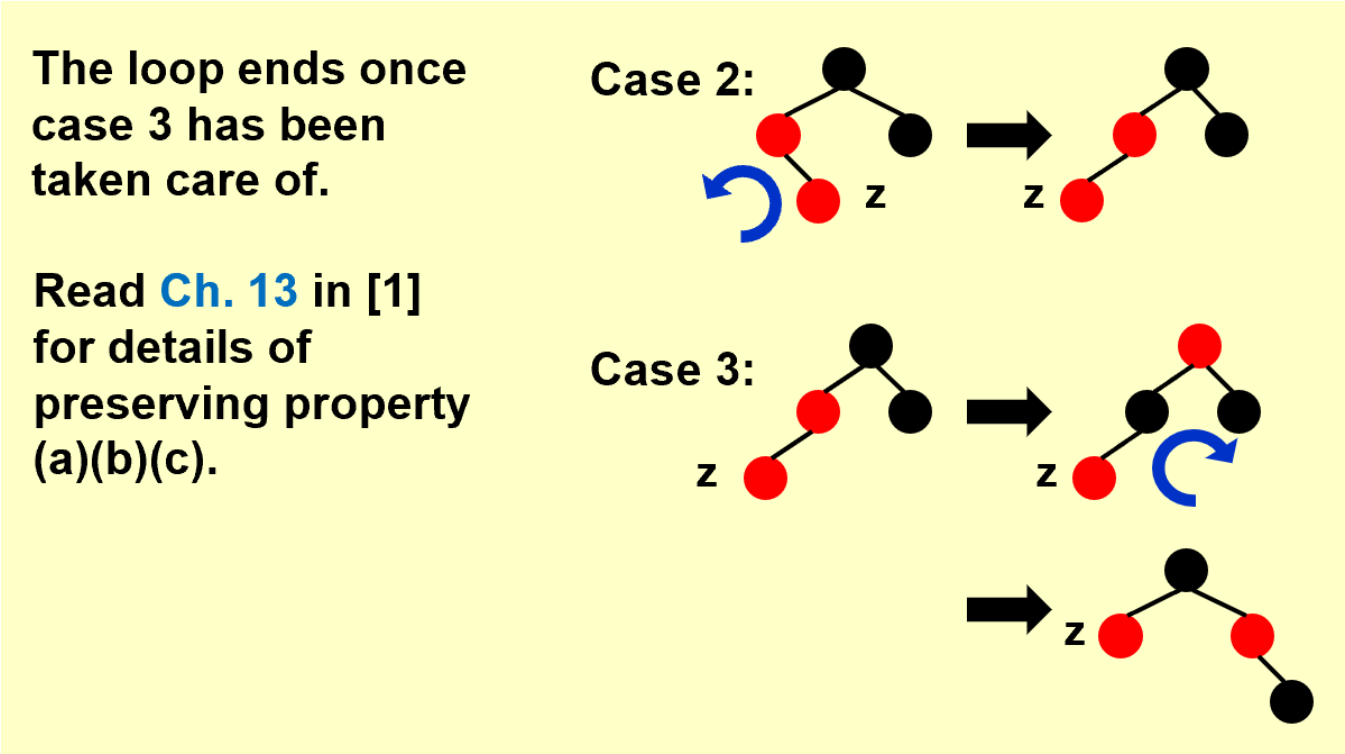
由于贸然插入黑色节点很有可能会导致性质5出现问题，故我们尽量插入**红色**节点。
综合考虑五条性质，插入**红色**节点时，**只有性质2**和**性质4**有可能会被破坏。

Case 1



此时插入的红色节点`z`依然可能破坏性质2或者4，故作迭**代处理**。
定义`z`，`z.p`，`y`，`z'`
循环终止条件：**为了保证满足性质4**。
自底向上修正树，z的子树都满足红黑性质
结束后把根节点染成黑色
循环过程中，如果`z`变为根节点，则(2)被破坏，(4)被保留；否则`z.p`及其祖先的颜色不会改变，(2)被保留，如果`z.p`为红色则(4)被破坏。性质4**只可能**在`z`和`z.p`之间被破坏
循环开始前，如果`z.p`是根，则红黑树性质决定`z.p`为黑色，循环调整之前不会改变`z.p`的颜色
循环结束后，`z.p`是黑色的（如果`z`为根节点那么其父节点哨兵结点也是黑色的），性质(4)不会被破坏，只有可能性质(2)被破坏，所以对根节点进行染色
还有三项对称情况。

Case 2 & Case 3



case 2、3**不需要**迭代。
对于case2、3来说，红黑树插入操作**循环总数不超过2**，因为case2、3一旦被执行就结束了。

Algorithm Analysis

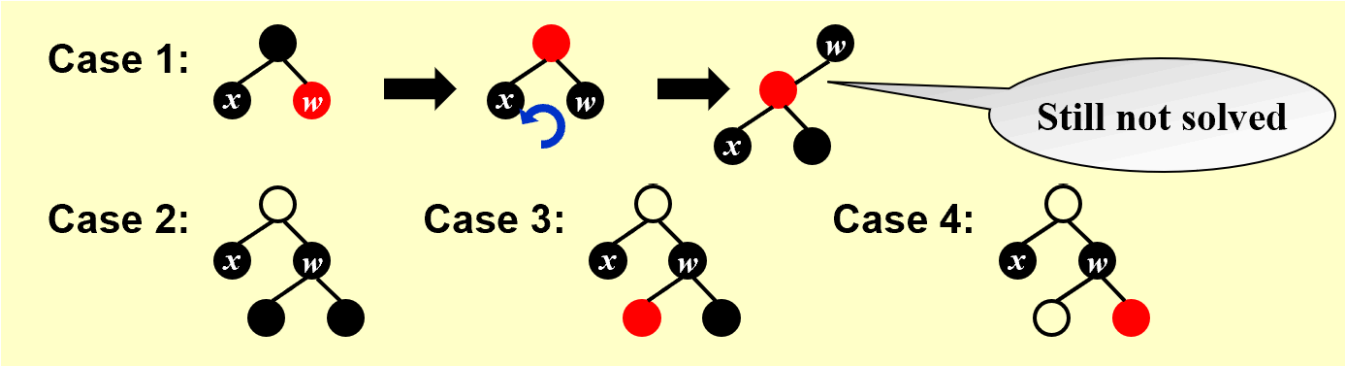
总的来说，红黑树的插入操作中，我们有：
$T = O(h) = O(lnN)$

Delete

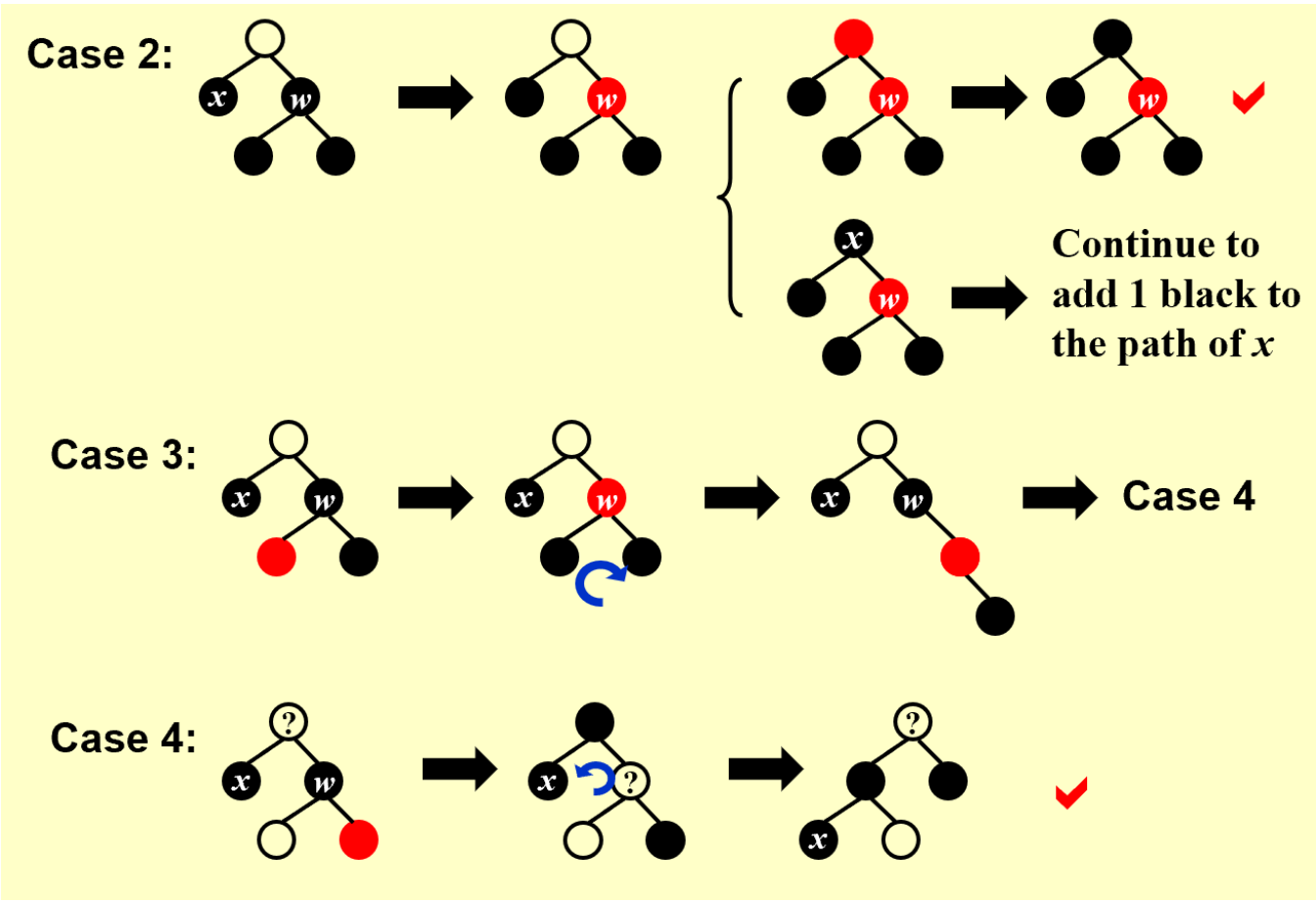
When we do deletions,we must add 1 blcak to the path of the replacing node.
If the node is red,we can delete it directly because it has no influence on th red-black tree.

- Delete a leaf node : Reset its parent link to NIL.(only if the node is black originally)
- Delete a degree 1 node : Replace the node by its single child.
- Delete a degree 2 node :
    1. Replace the node by the largest one in its left subtree or the smallest one in its right subtree.(keep the color)
    2. Delete the replacing node from the subtree.

登场人物：4个Case



分别操作(Case 1 见上图)



兄弟红易兄弟黑，换色单旋接着推；
侄黑父红换兄色，父侄双黑染红兄，上推；
近侄红来远侄黑，旋转能把模样改；
远侄红来单旋转，父兄换色侄变黑。

B+ Trees

Definition

A B+ tree of order M is a tree with the following structural properties:

- The root is either a leaf or has between 2 and M children.
- All nonleaf nodes (except the root) have between $\lceil M/2 \rceil$ and M children.
- All leaves are at the same depth.

Assume each nonroot leaf also has between $\lceil M/2 \rceil$ and M children.

Find

Insert

```
Btree  Insert ( ElementType X,  Btree T )
{
        Search from root to leaf for X and find the proper leaf node;
        Insert X;
        while ( this node has M+1 keys ) {
                    split it into 2 nodes with [(M+1)/2 ]  and [(M+1)/2 ] keys, respectively;
                    if (this node is the root)
                        create a new root with two children;
                    check its parent;
        }
}
```

Inverted File Index

Inverted File Index

Definition

Index is a mechanism for locating a given term in a text.
Inverted file contains a list of pointers(e.g. the number of a page) to all occurrences of that term in the text.

Doc	Text
1	Gold silver truck
2	Shipment of gold damaged in a fire
3	Delivery of silver arrived in a silver truck
4	Shipment of gold arrived in a truck

Inverted File Index

No.	Term	Times; Documents
1	a	<3; 2,3,4>
2	arrived	<2; 3,4>
3	damaged	<1; 2>
4	delivery	<1; 3>
5	fire	<1; 2>
6	gold	<3; 1,2,4>
7	of	<3; 2,3,4>
8	in	<3; 2,3,4>
9	shipment	<2; 2,4>
10	silver	<2; 1,3>
11	truck	<3; 1,3,4>

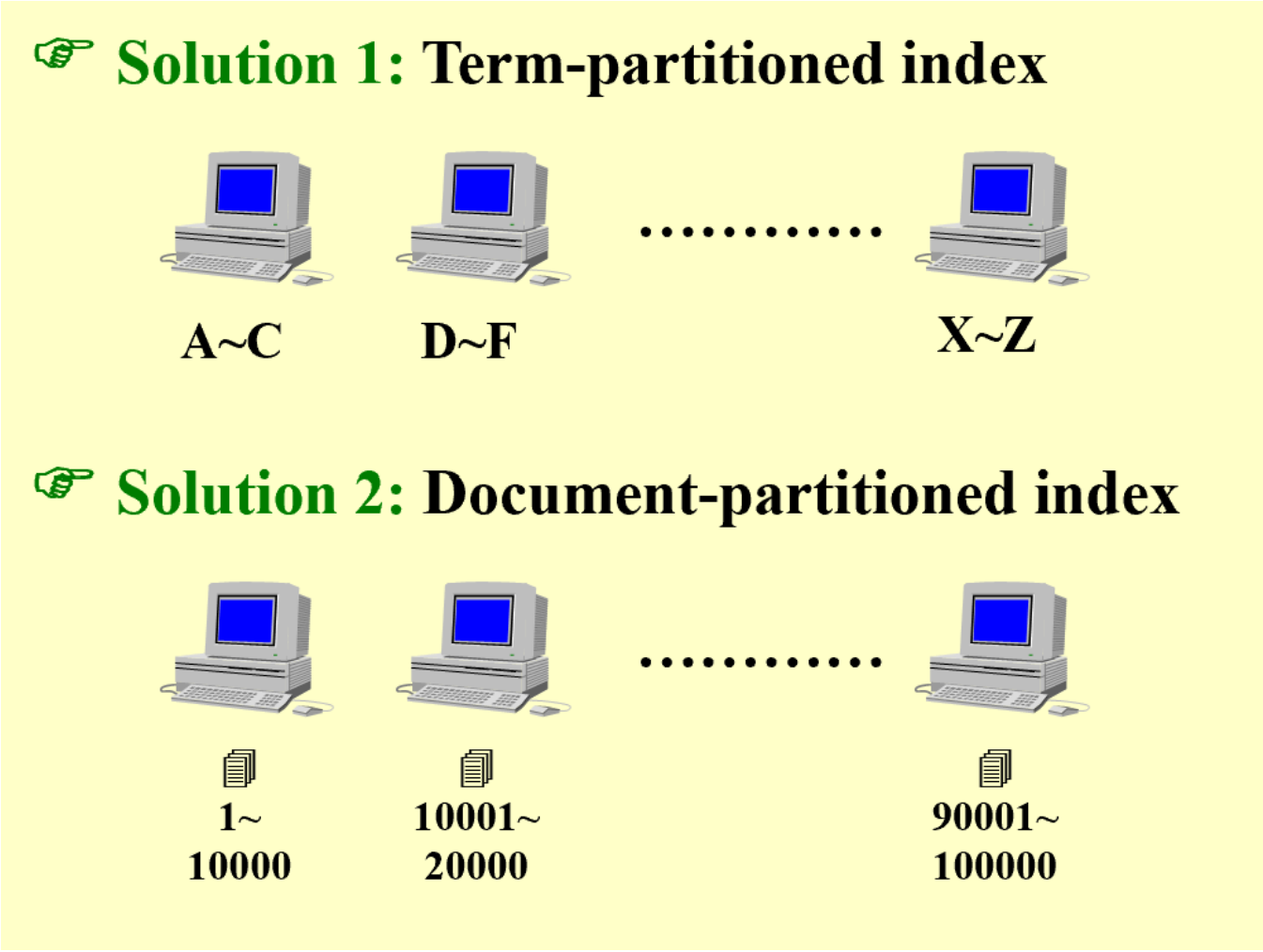
简单来说，inverted是因为它是根据term来索引的，而不是根据文档来索引的。也就是说用词找文档，而不是用文档找词。
为了更快的找到匹配项，我们先考虑频率低的词，因为它们的文档数目更少，所以我们可以先在这些词中找到匹配项，然后再在这些文档中找到匹配项。

Index Generateor

```
while ( read a document D ) {
    while ( read a term T in D ) {
        if ( Find( Dictionary, T ) == false )
            Insert( Dictionary, T );
        Get T’s posting list;
        Insert a node to T’s posting list;
    }
}
Write the inverted index to disk;
```

Distributed indexing

Each node contains index of a subset of collection



第二种相对更好，因为第一种划分法，如果1号服务器挂了，那么所有涉及A~C的搜索都将无法继续。

Dynamic indexing

Docs come in over time

- postings updates for terms already in dictionary
- new terms added to dictionary

Docs get deleted
创建一个临时性辅助index，每次更新在辅助index中，搜索时再合并到主index中。

Index Compression

- 删除不必要的词汇如a,the等

- 每次都记录与上一次出现的距离，而不是绝对位置，这样可以节省空间（否则很长的表导致绝对位置很大）。
-

Thresholding 阈值

- Document: only retrieve the top x documents where the documents are ranked by weight
- Query: Sort the query terms by their frequency in ascending order; search according to only some percentage of the original query terms.The world has a **low frequency** in the query, so it is **more important** to the query.
  相当于把搜索范围缩小了，但是搜索结果的质量也会下降，所以这个阈值要合理设置。

Measuring the Performance of an Index

Data Retrieval Performance Evaluation (after establishing correctness)

- Response time
- Index space

Information Retrieval Performance Evaluation

- How relevant is the answer set?

Relevance(only for imformation retrieval)

Relevance measurement requires 3 elements:

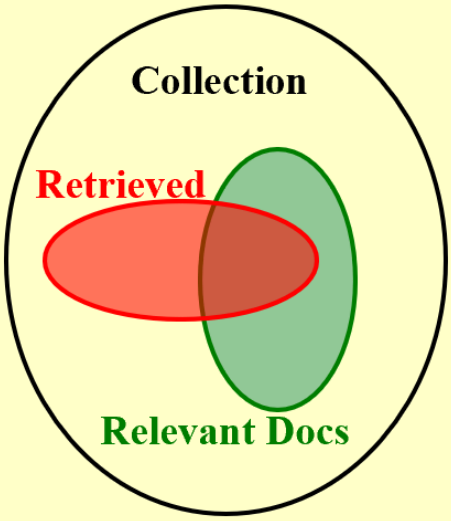
1. A benchmark document collection
2. A benchmark suite of queries
3. A binary assessment of either Relevant or Irrelevant for each query-doc pair

Precision and Recall

	Relevant	Irrelevant
Retrieved	R_R	I_R
Not Retrieved	R_N	I_N

Precision $P = R_R / (R_R + I_R)$

Recall $R = R_R / (R_R + R_N)$

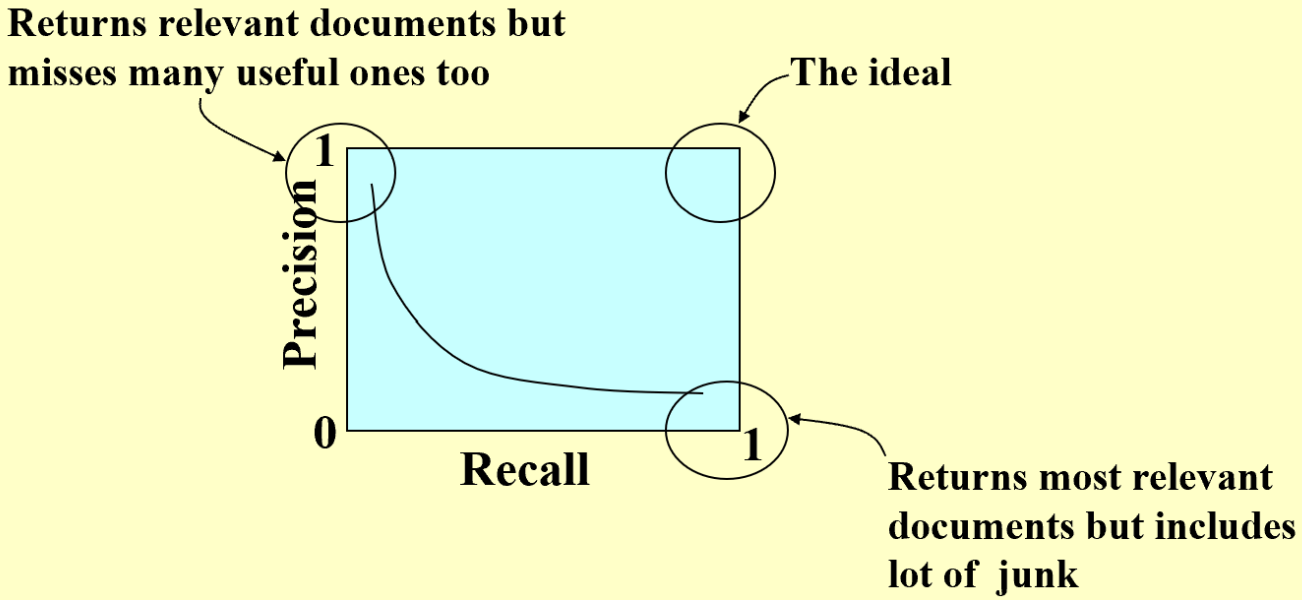


Precision: the percentage of the retrieved documents that are relevant

Recall: the percentage of the relevant documents that are retrieved

$$Precision = \frac{Retrieved\ and\ Relevant}{Retrieved}$$

$$Recall = \frac{Retrieved\ and\ Relevant}{Relevant}$$



重要：AVL树、splay树(伸展树)和红黑树比较

一、AVL树：

优点：查找、插入和删除，最坏复杂度均为$O(logN)$。实现操作简单

> 如过是随机插入或者删除，其理论上可以得到$O(logN)$的复杂度，但是实际情况大多不是随机的。如果是随机的，则AVL树能够达到比红黑树更优的结果，因为AVL树的高度更低。如果只进行插入和查找，则AVL树是优于红黑树的，因为红黑树更多的优势还是在删除动作上。

缺点：

1. 借助高度或平衡因子，为此需要改造元素结构，或额外封装-->伸展树可以避免。
2. 实测复杂度与理论复杂度上有差距。插入、删除后的旋转成本不菲。删除操作后，最多旋转$O(logN)$次，(Knuth证明，平均最坏情况下概率为0.21次)，若频繁进行插入/删除操作，得不偿失。
3. 单次动态调整后，全树拓扑结构的变化量可达$O(logN)$次。-->红黑树为$O(1)$

二、伸展树(splay tree)

优点：

1. 无需记录节点高度和平衡因子，编程实现简单易行
2. 分摊复杂度为$O(logN)$
3. 局部性强，缓存命中率极高时，效率甚至可以更高。

注：伸展树是根据数据访问的局部性而来的主要是：

1. 刚刚被访问的节点，极有可能在不就之后再次被访问到
2. 将被访问 的下一个节点，极有可能就处于不就之前被访问过的某个节点的附近。

缺点：

1. 仍不能保证单次最坏情况的出现，不适用效率敏感的场合
2. 复杂度分析比较复杂

三、红黑树

优点:

1. 所有的插入、删除、查找操作的复杂度都是$O(logN)$
2. 插入操作能够在最多2次旋转后达到平衡状态，而删除操作更是能够在一次旋转后达到平衡状态。删除操作有可能导致递归的双黑修正，但是在旋转之前，只是染色而树的结构没有任何实质性的改变，因此速度优于AVL树。

3. 红黑树可以保证在每次插入或删除操作之后的重平衡过程中，全书拓扑结构的更新仅涉及常数个节点。尽管最坏情况下需对$O(logN)$个节点重染色，但就分摊意义而言，仅为$O(1)$个。

缺点：左右子树高度相差比AVL树大。

四、总结

树	插入	删除	查找	空间
AVL树	$O(logN)$	$O(logN)$	$O(logN)$	$O(N)$
伸展树	$O(logN)$	$O(logN)$	$O(logN)$	$O(N)$
红黑树	$O(logN)$	$O(logN)$	$O(logN)$	$O(N)$

（均摊下红黑树插入、删除为$O(1)$）

Leftist Heaps and Skew Heaps

Leftist Heap

Recall

Heap: Structure Property + Order Property
结构性质：

- 堆是完全二叉树
- 堆的每一个子树都是堆

顺序性质：

- 堆的每一个孩子都比自己大（小）

堆的操作：

- percolate up & percolate down

堆的合并：

- 开销为$O(N)$，线性的把两个堆中的元素逐一插入。我们现在就要加速这一过程。

Leftist Heap

Structure Property:

- a binary tree,but **unbalanced**

Order Property:

- the same as normal heap

Definiition

NPL,The null path length

The null path length,$Npl(x)$,of any node X is the length of the **shortest path** from X to a node **without two childeren**. Define Npl(NULL)=-1.

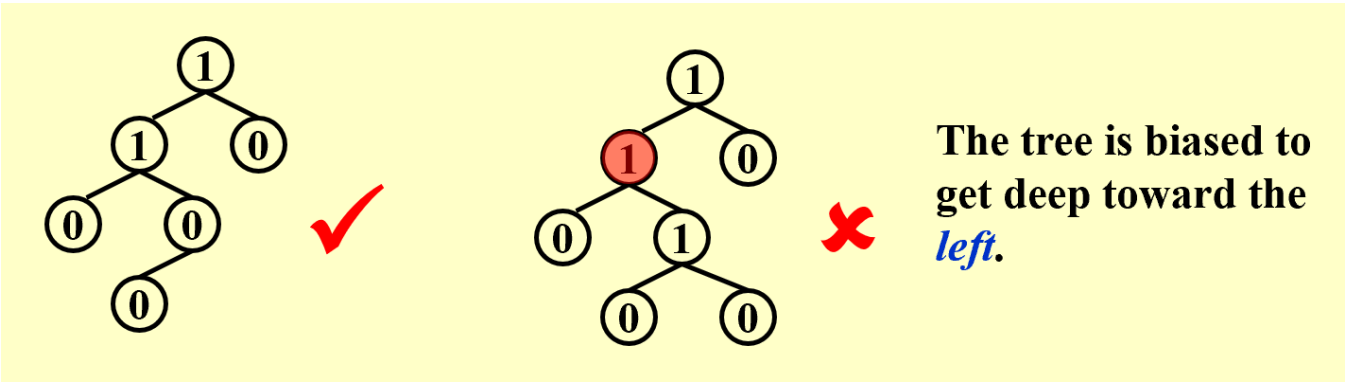
Note:

$Npl(x) = min\{Npl(c) + 1\}$, for all c as children of x

Leftist Heap

The leftist heap property is that for every node x in the heap,the null path length of the left child is **at least as large as** ($\geq$) that of the right child.

Example:



【Theorem】 :

A leftist tree with r nodes on the right path must have at least $2^r - 1$ nodes.

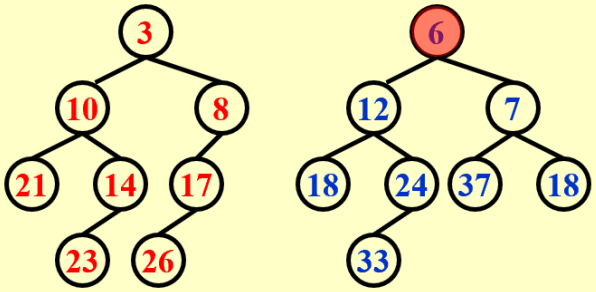
So we can find that the length of the right path of a leftist tree with N nodes is $\lfloor log(N + 1) \rfloor$

Merge Operation Recursive

☞ **Declaration:**

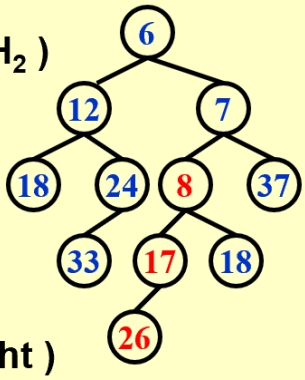
```
struct TreeNode
{
    ElementType     Element;
    PriorityQueue   Left;
    PriorityQueue   Right;
    int             Npl;
} ;
```

☞ **Merge (recursive version):**

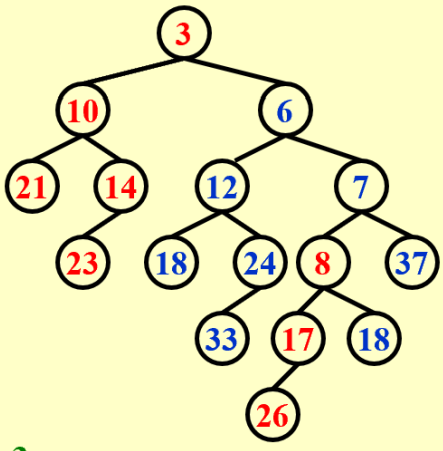


H_1 H_2

Step 1:
Merge(H_1->Right, H_2)



Step 2:
Attach(H_2, H_1->Right)



Step 3:
Swap(H_1->Right, H_1->Left)
if necessary

5

1. 比较需要合并的两棵树树根节点谁更小，大的与小的右子树合并
2. 递归方法合并根节点小的右子树和根节点大的整棵树。
3. 比较新树的左右子树哪一个的Npl更大，若右边更大则交换左右子树。

Declaration

```
struct TreeNode
{
        ElementType Element;
        PriorityQueue Left;
        PriorityQueue Right;
        int Npl;
} ;
```

Implementation

```
PriorityQueue  Merge ( PriorityQueue H1, PriorityQueue H2 )
{
        if ( H1 == NULL )   return H2;
        if ( H2 == NULL )   return H1;
        if ( H1->Element < H2->Element )  return Merge1( H1, H2 );
        else return Merge1( H2, H1 );
}

static PriorityQueue
Merge1( PriorityQueue H1, PriorityQueue H2 )
{
        if ( H1->Left == NULL )         /* single node */
                H1->Left = H2;        /* H1->Right is already NULL
                                      and H1->Npl is already 0 */
        else {
                H1->Right = Merge( H1->Right, H2 );     /* Step 1 & 2 */
                if ( H1->Left->Npl < H1->Right->Npl )
                        SwapChildren( H1 );        /* Step 3 */
                H1->Npl = H1->Right->Npl + 1;
        } /* end else */
        return H1;
}
```

Analysis

$T_p = O(logN)$

Merge Operation Iterative

1. Sort the right paths without changing their left children
2. Swap children if necessary

Implementation

```
// 定义左偏树结点结构体
struct Node {
    int val; // 结点值
    int dist; // 结点距离
    Node* left; // 左子结点
    Node* right; // 右子结点
    Node(int v): val(v), dist(0), left(nullptr), right(nullptr) {} // 构造函数
};
// 合并两个左偏树，返回合并后的根结点
Node* merge(Node* x, Node* y) {
    if (!x) return y; // 如果x为空，返回y
    if (!y) return x; // 如果y为空，返回x
    if (x->val > y->val) swap(x, y); // 保证x的值小于等于y的值
    stack<Node*> st; // 创建一个栈，用来存储沿途修改的路径
    while (true) {
        if (!x->right) { // 如果x没有右子结点，直接将y作为其右子结点，并结束循环
            x->right = y;
            break;
        }
        st.push(x); // 将x压入栈中，记录修改路径
        x = x->right; // 将x移动到其右子结点上
        if (x->val > y->val) swap(x, y); // 保证x的值小于等于y的值
    }
    while (!st.empty()) { // 从栈中弹出修改过的结点，并调整其距离和左右子结点位置，保持左偏性质
        Node* p = st.top();// 弹出一个父结点p
        p->right = x; // 将p的右子结点设为当前处理的结点x（初始为合并后的右子树）
        if (!p->left || p->left->dist < p->right->dist) {
            swap(p->left, p->right); // 如果p没有左子结点或者其左子距离小于右子距离，则交换其左右子结点位
        }
        p->dist = p->right ? p->right->dist + 1 : 0; // 更新p的距离为其右子距离加一（如果有右子）或者零
        x = p; // 将当前处理的结点设为p（向上回溯）
    }
    return x; // 返回最终合并后的根节点（初始为最底层修改过的父节点）
}
```

Analysis

$$T_p = O(logN)$$

DeleteMin Operation

1. Delete the root
2. Merge the two subtrees

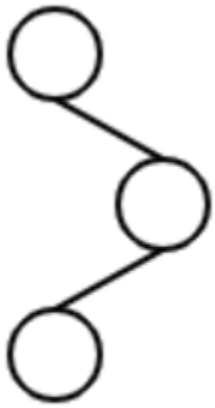
Analysis

$T_p = O(logN)$

Skew Heap

Definition

Idea:
Always swap the left and right children except that the largest of all the nodes on the right paths does not have its children swapped. **No Npl.**
总是交换左右子节点，除了右边路径上最大的节点没有交换它的子节点。**不需要Npl。**
由于没有npl，它可以节省更多空间。斜堆也不要求堆是左倾堆。
由于斜堆并不是严格的左倾堆，最坏的情况下右路长度可能为`N`，因此采用递归调用`merge`的风险是出现`stack overflow`。
只要一直插入如下图所示的结构（第一个节点不是，是为了方便画），`Skew Heap`就会退化成右倾的一条链。



Merge

1. 如果一个空斜堆与一个非空斜堆合并，返回非空斜堆。
2. 如果两个斜堆都非空，那么比较两个根节点，取较小堆的根节点为新的根节点。将"较小堆的根节点的右孩子"和"较大堆"进行合并。
3. 合并后，交换新堆根节点的左孩子和右孩子。

Implementation

```
SkewNode* merge_skewheap(SkewHeap x, SkewHeap y)
{
    if(x == NULL)
        return y;
    if(y == NULL)
        return x;

    // 合并x和y时，将x作为合并后的树的根；
    // 这里的操作是保证：x的key < y的key
    if(x->key > y->key)
        swap_skewheap_node(y, x);

    // 将x的右孩子和y合并，
    // 合并后直接交换x的左右孩子，而不需要像左倾堆一样考虑它们的npl。
    SkewNode *tmp = merge_skewheap(x->right, y);
    x->right = x->left;
    x->left  = tmp;

    return x;
}
```

对于`Skew Heap`插入新的节点实质上也可以看作`Merge`操作。

Amortized Analysis

【**Definition**】 A node p is **heavy** if the number of descendants of p's right subtree is at least half of the number of descendants of p, and light otherwise. Note that the number of descendants of a node includes the node itself.
如果节点p的右子树的子代数至少是p子代数的一半，则节点p为**重**，否则为轻。请注意，节点的后代数量包括节点本身。

The **only** nodes whose heavy/light status can change are nodes that are initially on the right path.
合并操作中，**只有**右路径上的节点轻重状态回改变。右路径上**重节点一定变轻节点，而轻节点不一定变重节点**。

这里我们使用均摊分析中的势能方法分析斜堆的算法复杂度。
如果我们如下定义D_i:
D_i =the root of the resulting tree

那么，我们如下定义势能函数：

$\Phi(D_i)$ =number of **heavy** nodes

设合并前两个斜堆的右路径上的**重节点**数量分别为h_1,h_2，所有左路径上的**重节点**数量为h（由于这些重节点的轻重**不再**变化），所有右路径上的轻节点数量分别为l_1,l_2。
我们可以观察到，在合并之前，
$\Phi_0 = h_1 + h_2 + h$

由于在合并过程中，重节点**一定**变轻节点（左右子树要经过交换），而轻节点**不一定**变重节点。
经过合并之后：
$\Phi_N \leq l_1 + l_2$（最差情况，**所有**的轻节点都叛变了）
而最差情况：
$T_{worst} = l_1 + h_1 + l_2 + h_2$
所以，根据势能方法的定义，
$T_{amortized} = T_{worst} + \Phi_N - \Phi_0 \leq 2(l_1 + l_2)$
而$l = O(logN)$
故$T_{amortized} = O(logN)$

Binomial Queue

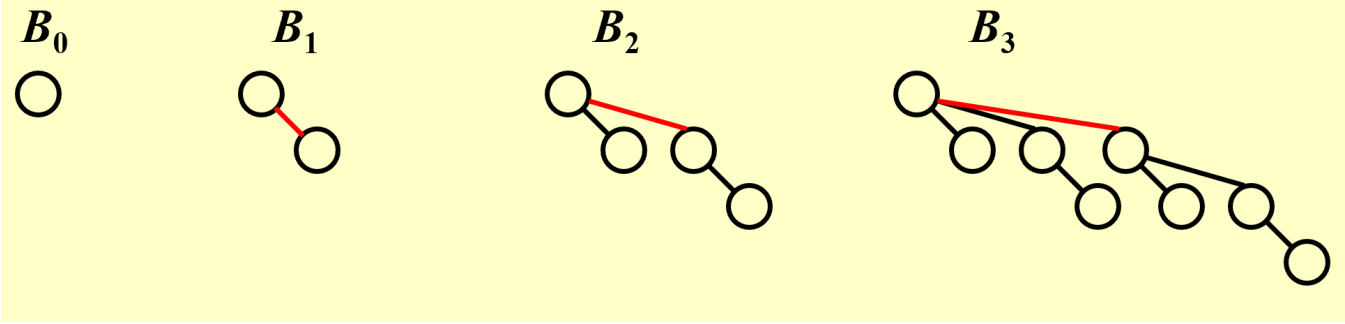
Structure

A binomial queue is not a heap-ordered tree, but rather a collection of heap-ordered trees, known as a forest. Each heap-ordered tree is a binomial tree.

A binomial tree of height 0 is a one-node tree.

A binomial tree, B_k, of height k is formed by attaching a binomial tree, B_{k-1}, to the root of another binomial tree, B_{k-1}.

仅包含一个结点的有序树是一棵二项树称为B_0树。二项树B_k由两棵B_{k-1}树组成，其中一棵B_{k-1}树的根作为另一棵B_{k-1}树根的最左孩子$(k \geq 0)$。



Note:B_k consists of a root with k children, which are $B_0, B_1, \ldots, B_{k-1}$. B_k has exactly 2^k nodes. The number of nodes at depth d is $\binom{k}{d}$.

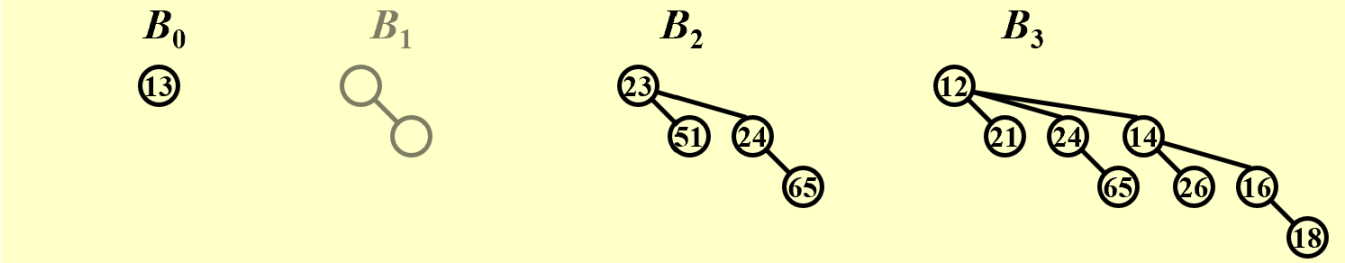
B_k structure + heap order + one binomial tree for each height.

$\Rightarrow$ A priority queue of any size can be uniquely represented by a collection of binomial trees.

一些由二项树**构成**的树林被称作二项堆，是一种优先队列。

【Example】Represent a priority queue of size 13 by a collection of binomial trees.

Solution: $13 = 2^0 + 0\times 2^1 + 2^2 + 2^3 = 1101_2$



Operations

FindMin

The minimum key is in one of the roots.

There are at most $\lceil logN \rceil$ roots, hence $T_p = O(logN)$.

We can remember the minimum and update whenever it is changed.Then this operation will take $O(1)$.

Merge

Note: If the smallest nonexistent binomial tree is Bi , then $T_p = Const\cdot (i+1)$.

Performing N Inserts on an initially empty binomial queue will take $O(N)$ worst-case time. Hence the average time is constant.

DeleteMin

Step 1: FindMin in B_k
Step 2: Remove B_k from H
Step 3: Remove root from B_k
Step 4: `Merge (H', H")`

Implementation

```
//definition
typedef struct BinNode *Position;
typedef struct Collection *BinQueue;
typedef struct BinNode *BinTree;  /* missing from p.176 */

struct BinNode
{
        ElementType            Element;
        Position            LeftChild;
        Position             NextSibling;
} ;

struct Collection
{
        int                    CurrentSize;  /* total number of nodes */
        BinTree        TheTrees[ MaxTrees ];
};

//merge two binomial trees
BinTree Merge( BinTree T1, BinTree T2 )
{
    if( T1->Element > T2->Element )
        return Merge( T2, T1 );
    T2->NextSibling = T1->LeftChild;
    T1->LeftChild = T2;
    return T1;
}

//merge two binomial queues
BinQueue Merge_BinQueue( BinQueue H1, BinQueue H2 )
{
    BinTree T1, T2, Carry = NULL;
    int i, j;

    if( H1->CurrentSize + H2->CurrentSize > Capacity )
        Error( "Merge would exceed capacity" );

    H1->CurrentSize += H2->CurrentSize;
    for( i = 0, j = 1; j <= H1->CurrentSize; i++, j *= 2 )
    {
        T1 = H1->TheTrees[ i ]; T2 = H2->TheTrees[ i ];

        switch( !!T1 + 2 * !!T2 + 4 * !!Carry )
        {
          case 0: /* No trees */
          case 1: /* Only H1 */
```

```
            break;
          case 2: /* Only H2 */
            H1->TheTrees[ i ] = T2;
            H2->TheTrees[ i ] = NULL;
            break;
          case 4: /* Only Carry */
            H1->TheTrees[ i ] = Carry;
            Carry = NULL;
            break;
          case 3: /* H1 and H2 */
            Carry = Merge( T1, T2 );
            H1->TheTrees[ i ] = H2->TheTrees[ i ] = NULL;
            break;
          case 5: /* H1 and Carry */
            Carry = Merge( T1, Carry );
            H1->TheTrees[ i ] = NULL;
            break;
          case 6: /* H2 and Carry */
            Carry = Merge( T2, Carry );
            H2->TheTrees[ i ] = NULL;
            break;
          case 7: /* All three */
            H1->TheTrees[ i ] = Carry;
            Carry = Merge( T1, T2 );
            H2->TheTrees[ i ] = NULL;
            break;
        }
    }
    return H1;
}

//delete the minimum element in binomial queue
ElementType DeleteMin( BinQueue H )
{
    int i, j;
    int MinTree; /* The tree with the minimum item */
    BinQueue DeletedQueue;
    Position DeletedTree, OldRoot;
    ElementType MinItem;

    if( IsEmpty( H ) )
    {
        Error( "Empty binomial queue" );
        return -Infinity;
    }
```

```
    MinItem = Infinity;
    for( i = 0; i < MaxTrees; i++ )
    {
        if( H->TheTrees[ i ] &&
            H->TheTrees[ i ]->Element < MinItem )
        {
            /* Update minimum */
            MinItem = H->TheTrees[ i ]->Element;
            MinTree = i;
        }
    }

    DeletedTree = H->TheTrees[ MinTree ];
    OldRoot = DeletedTree;
    DeletedTree = DeletedTree->LeftChild;
    free( OldRoot );

    DeletedQueue = Initialize( );
    DeletedQueue->CurrentSize = ( 1 << MinTree ) - 1;
    for( j = MinTree - 1; j >= 0; j-- )
    {
        DeletedQueue->TheTrees[ j ] = DeletedTree;
        DeletedTree = DeletedTree->NextSibling;
        DeletedQueue->TheTrees[ j ]->NextSibling = NULL;
    }

    H->TheTrees[ MinTree ] = NULL;
    H->CurrentSize -= DeletedQueue->CurrentSize + 1;

    Merge_BinQueue( H, DeletedQueue );
    return MinItem;
}
```

Backtracking

A sure-fire way to find the answer to a problem is to make a list of all candidate answers, examine each, and following the examination of all or some of the candidates, declare the identified answer.

Backtracking enables us to eliminate the explicit examination of a large subset of the candidates while still guaranteeing that the answer will be found if the algorithm is run to termination.

The basic idea is that suppose we have a partial solution ($x_1, ..., x_i$) where each $x_k \in S_k$ for $1 \leq k \leq i < n$. First we add $x_{i+1} \in S_{i+1}$ and check if ($x_1, ..., x_i, x_{i+1}$) satisfies the constrains. If the answer is “yes” we continue to add the next x, else we delete x_{i+1} and backtrack to the previous partial solution ($x_1, ..., x_i$).

我们不断枚举，当发现**不满足条件**时，回溯到上一步，继续枚举。

The Turnpike Reconstruction Problem

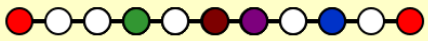
Given N points on the x-axis with coordinates $x_1 < x_2 < \ldots < x_N$. Assume that $x_1 = 0$. There are $N(N-1)/2$ distances between every pair of points.

〖Example〗 Given D = { 1, 2, 2, 2, 3, 3, 3, 4, 5, 5, 5, 6, 7, 8, 10 }

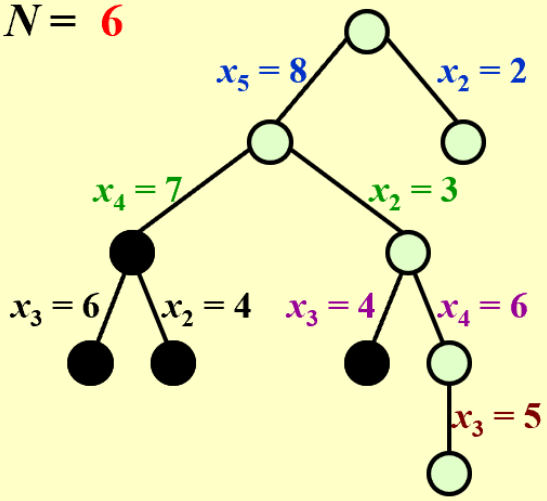
Step 1: $N(N-1)/2 = 15$ **implies** $N =$ 6

Step 2: $x_1 = 0$ **and** $x_6 = 10$

Step 3: find the next largest distance and check



(0, 3, 5, 6, 8, 10)



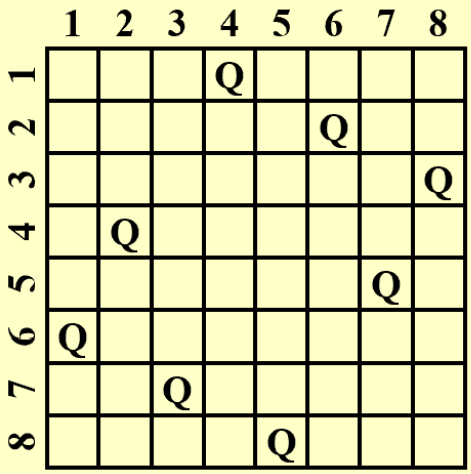
```
bool Reconstruct ( DistType X[ ], DistSet D, int N, int left, int right )
{ /* X[1]...X[left-1] and X[right+1]...X[N] are solved */
    bool Found = false;
    if ( Is_Empty( D ) )
        return true; /* solved */
    D_max = Find_Max( D );
    /* option 1: X[right] = D_max */
    /* check if |D_max-X[i]|∈D is true for all X[i]'s that have been solved */
    OK = Check( D_max, N, left, right ); /* pruning */
    if ( OK ) { /* add X[right] and update D */
        X[right] = D_max;
        for ( i=1; i<left; i++ )  Delete( |X[right]-X[i]|, D);
        for ( i=right+1; i<=N; i++ )  Delete( |X[right]-X[i]|, D);
        Found = Reconstruct ( X, D, N, left, right-1 );
        if ( !Found ) { /* if does not work, undo */
            for ( i=1; i<left; i++ )  Insert( |X[right]-X[i]|, D);
            for ( i=right+1; i<=N; i++ )  Insert( |X[right]-X[i]|, D);
        }
    }
    /* finish checking option 1 */

    if ( !Found ) { /* if option 1 does not work */
        /* option 2: X[left] = X[N]-D_max */
        OK = Check( X[N]-D_max, N, left, right );
        if ( OK ) {
            X[left] = X[N] - D_max;
            for ( i=1; i<left; i++ )  Delete( |X[left]-X[i]|, D);
            for ( i=right+1; i<=N; i++ )  Delete( |X[left]-X[i]|, D);
            Found = Reconstruct (X, D, N, left+1, right );
            if ( !Found ) {
                for ( i=1; i<left; i++ ) Insert( |X[left]-X[i]|, D);
                for ( i=right+1; i<=N; i++ ) Insert( |X[left]-X[i]|, D);
            }
        }
        /* finish checking option 2 */
    } /* finish checking all the options */

    return Found;
}
```

Eight Queens

Find a placement of 8 queens on an 8 X 8 chessboard such that no two queens attack.

Two queens are said to attack iff they are in the same row, column, diagonal, or antidiagonal of

the chessboard.



Q_i ::= queen in the i-th row

x_i ::= the column index in which Q_i is

Solution = $(x_1, x_2, \ldots, x_8)$
= (4, 6, 8, 2, 7, 1, 3, 5)

Constrains: ① $S_i = \{ 1,2,3,4,5,6,7,8 \}$ for $1 \le i \le 8$

② $x_i \neq x_j$ if $i \neq j$ ③ $(x_i - x_j) / (i - j) \neq \pm 1$

```
void EightQueens ( int k )
{   int i;
    if ( k > N ) { /* a solution is found */
        PrintSolution( );
        return;
    }
    for ( i=1; i<=N; i++ ) {
        X[k] = i;
        if ( Place( k ) ) /* pruning */
            EightQueens( k+1 );
    }
}
bool Place ( int k )
{   int i;
    for ( i=1; i<k; i++ ) {
        if ( ( X[i] == X[k] ) || ( abs(X[i]-X[k]) == abs(i-k) ) )
            return false;
    }
    return true;
}
```

General Backtracking

```
bool Backtracking ( int i )
{   Found = false;
    if ( i > N )
        return true; /* solved with (x1, …, xN) */
    for ( each xi in Si ) {
        /* check if satisfies the restriction R */
        OK = Check((x1, …, xi) , R ); /* pruning */
        if ( OK ) {
            Count xi in;
            Found = Backtracking( i+1 );
            if ( !Found )
                Undo( i ); /* recover to (x1, …, xi-1) */
        }
        if ( Found ) break;
    }
    return Found;
}
```

回溯的效率跟S的规模、约束函数的复杂性、满足约束条件的结点数相关。
约束函数决定了剪枝的效率，但是如果函数本身太复杂也未必合算。
满足约束条件的结点数最难估计，使得复杂度分析很难完成。

Divide and Conquer

Algorithm Design

Divide: the problem into a number of sub-problems.
Conquer: the sub-problems by solving them recursively.
Combine: the solutions to the sub-problems into the solution for the original problem.
General recurrence: $T(N) = aT(N/b) + f(N)$

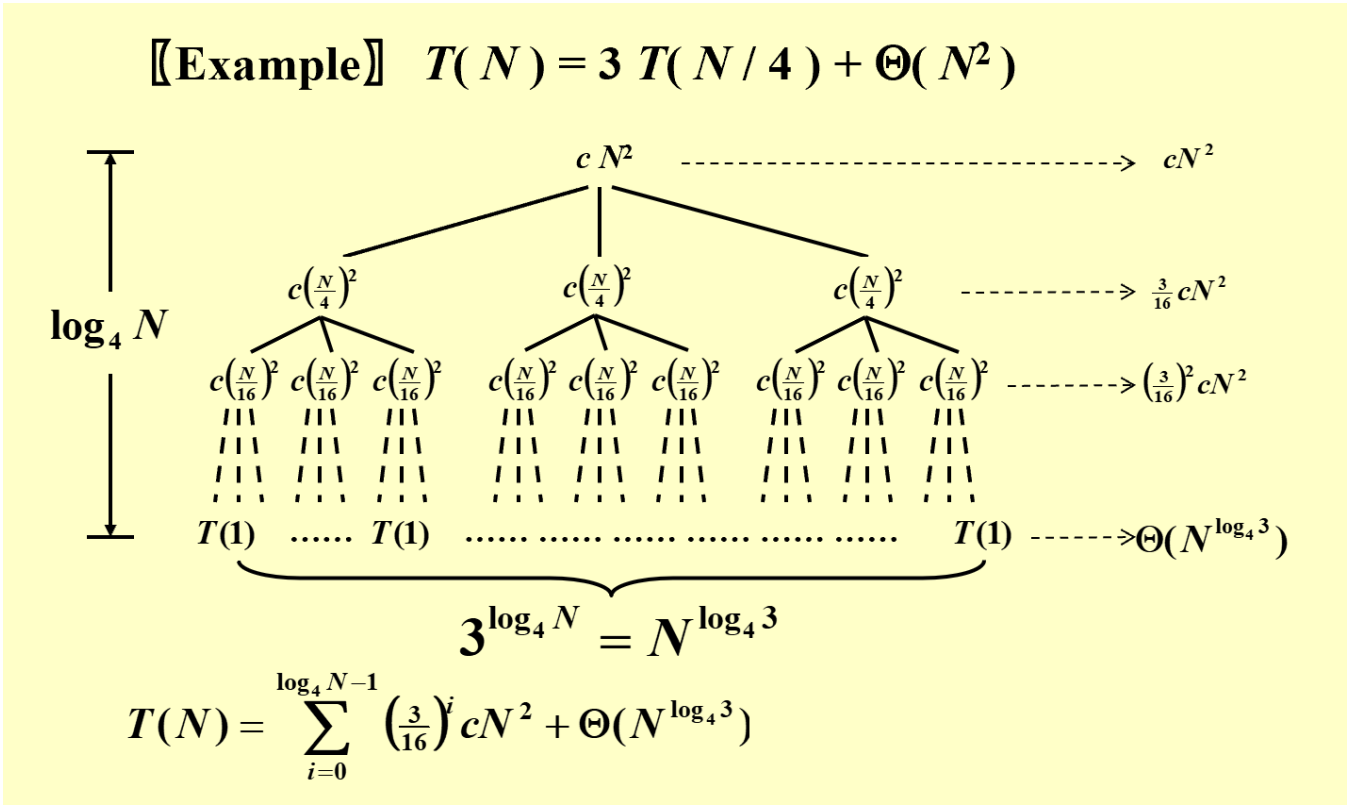
Closest Points Problem

Given N points in a plane. Find the closest pair of points. (If two points have the same position, then that pair is the closest with distance 0.)

Algorithm Analysis

Substitution method

Recursion-tree method



一个示例，用于分析算法复杂度的方法，叶子节点表示划分完成后最小子问题的时间开销，中间节点表示合并子问题的时间开销。

Master method

【Master Theorem】

Let $a \geq 1$ and $b > 1$ be constants, let $f(N)$ be a function, and let $T(N)$ be defined on the nonnegative integers by the recurrence $T(N) = aT(N/b) + f(N)$. Then:

1. If $f(N) = O(N^{log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(N) = \Theta(N^{log_b a})$
2. If $f(N) = \Theta(N^{log_b a})$, then $T(N) = \Theta(N^{log_b a} logN)$
3. If $f(N) = \Omega(N^{log_b a + \epsilon})$ for some constant $\epsilon > 0$, **and if** $af(N/b) < cf(N)$ for some constant $c < 1$ and all sufficiently large N, then $T(N) = \Theta f(N)$

【Example】

- Mergesort has $a = b = 2$, and case 2
  $T = O(NlogN)$
- $a = b = 2$, $f(N) = NlogN$?
  $T = O(NlogN)$ (X)
  这时候主方法就挂了，因为没有满足三种中的任意一种，这种情况下要用递归树。

其他两种主方法的形式：

【Theorem】The solution to the equation

$$T(N) = a\,T(N/b) + \Theta(N^k \log^p N),$$

where $a \geq 1$, $b > 1$, and $p \geq 0$ is

$$T(N) = \begin{cases} O(N^{\log_b a}) & \text{if } a > b^k \\ O(N^k \log^{p+1} N) & \text{if } a = b^k \\ O(N^k \log^p N) & \text{if } a < b^k \end{cases}$$

〖Example〗 Mergesort has $a = b = 2$, $p = 0$ and $k = 1$.
→ $T = O(N \log N)$

〖Example〗 Divide with $a = 3$, and $b = 2$ for each recursion;
Conquer with $O(N)$ – that is, $k = 1$ and $p = 0$.
→ $T = O(N^{1.59})$

If conquer takes $O(N^2)$ then $T = O(N^2)$.

〖Example〗 $a = b = 2$, $f(N) = N\log N$ → $T = O(N \log^2 N)$

【Master Theorem】The recurrence $T(N) = aT(N/b) + f(N)$ can be solved as follows:

1. If $af(N/b) = \kappa f(N)$ for some constant $\kappa < 1$, then $T(N) = \Theta(f(N))$
2. If $af(N/b) = Kf(N)$ for some constant $K > 1$, then $T(N) = \Theta(N^{\log_b a})$
3. If $af(N/b) = f(N)$, then $T(N) = \Theta(f(N)\log_b N)$

〖Example〗 $a = 4, b = 2, f(N) = N\log N$

$$af(N/b) = 4(N/2)\log(N/2) = 2N\log N - 2N \ ?$$

$$f(N) = N\log N \qquad O(N^{\log_b a - \varepsilon}) = O(N^{2-\varepsilon})$$

$$\rightarrow T = O(N^2)$$

Dynamic Programming

Recall

Use a table instead of recursion.

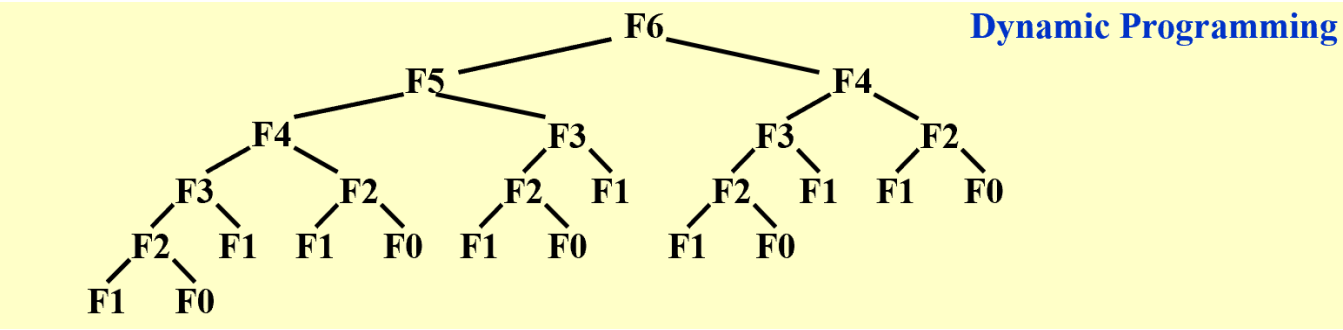
动态规划最核心的思想，就在于**拆分子问题**，**记住过往**，减少重复计算。

【Example】

Fibonacci Numbers

$F(N) = F(N-1) + F(N-2)$

```
int  Fib( int N )
{
    if ( N <= 1 )
        return  1;
    else
        return  Fib( N - 1 ) + Fib( N - 2 );
}
```



可以发现，在运算递归中，有很多重复的计算，这就是动态规划的基础。我们可以保留前两项的值，然后每次计算新的一项，这样就不用重复计算了。

```
int  Fibonacci ( int N )
{   int  i, Last, NextToLast, Answer;
    if ( N <= 1 )  return  1;
    Last = NextToLast = 1;    /* F(0) = F(1) = 1 */
    for ( i = 2; i <= N; i++ ) {
        Answer = Last + NextToLast;   /* F(i) = F(i-1) + F(i-2) */
        NextToLast = Last; Last = Answer;  /* update F(i-1) and F(i-2) */
    }  /* end-for */
    return  Answer;
}
```

$$T(N) = O(N)$$

Matrix Chain Multiplication

Given a sequence of N matrices $M_1, M_2, \ldots, M_N$, where for $i = 1, 2, \ldots, N$, matrix M_i has dimension $r_{i-1} \times r_i$.In which order can we compute the product of N matrices with minimal computing time?

设b_i为共有i个矩阵相乘得到的结果数，比如b_2=1,b_3=2,...
我们又递推式：
$b_n = \Sigma_{i=1}^{n-1} b_i b_{n-i}$

Suppose we are to multiply n matrices M_1 M_N where M_i is an $r_{i-1} \times r_i$ matrix. Let m_{ij} be the cost of the optimal way to compute $M_i * \ldots * M_j$. Then we have the recurrence equations:

$$m_{ij} = \begin{cases} 0 & \text{if } i = j \\ \min\limits_{i \le l < j} \{ m_{il} + m_{l+1\,j} + r_{i-1} r_l r_j \} & \text{if } j > i \end{cases}$$

```
/* r contains number of columns for each of the N matrices */
/* r[ 0 ] is the number of rows in matrix 1 */
/* Minimum number of multiplications is left in M[ 1 ][ N ] */
void OptMatrix( const long r[ ], int N, TwoDimArray M )
{   int  i, j, k, L;
    long  ThisM;
    for( i = 1; i <= N; i++ )   M[ i ][ i ] = 0;
    for( k = 1; k < N; k++ ){/* k = j - i */
        for( i = 1; i <= N - k; i++ ) { /* For each position */
            j = i + k;
            M[ i ][ j ] = Infinity;
            for( L = i; L < j; L++ ) {
                ThisM = M[ i ][ L ] + M[ L + 1 ][ j ] + r[ i - 1 ] * r[ L ] * r[ j ];
                if ( ThisM < M[ i ][ j ] )  /* Update min */
                M[ i ][ j ] = ThisM;
            }  /* end for-L */
        }  /* end for-Left */
    }
}
```

$T(N) = O(N^3)$

Optimal Binary Search Tree

Given N words $w_1 < w_2 < \ldots\ldots < w_N$, and the probability of searching for each w_i is p_i . Arrange these words in a binary search tree in a way that minimize the expected total access time.

$T(N) = \Sigma_{i=1}^{N} p_i(1 + d_i)$

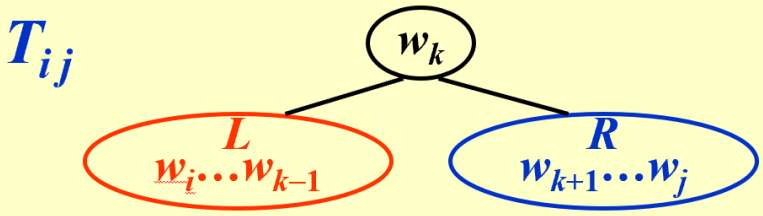
word	break	case	char	do	return	switch	void
probability	0.22	0.18	0.20	0.05	0.25	0.02	0.08

T_{ij} ::= OBST for $w_i, \ldots\ldots, w_j$ $(i < j)$

c_{ij} ::= cost of T_{ij} $(c_{ii} = 0)$

r_{ij} ::= root of T_{ij}

w_{ij} ::= weight of T_{ij} $= \sum_{k=i}^{j} p_k$ $(w_{ii} = p_i)$



$c_{ij} = p_k + \text{cost}(L) + \text{cost}(R) + \text{weight}(L) + \text{weight}(R)$

$= p_k + c_{i,k-1} + c_{k+1,j} + w_{i,k-1} + w_{k+1,j} = w_{ij} + c_{i,k-1} + c_{k+1,j}$

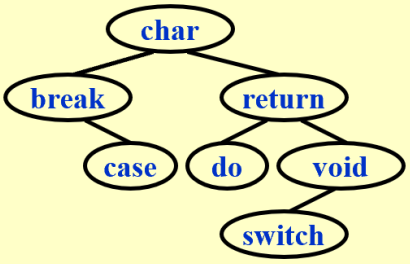
T_{ij} is optimal $\Rightarrow$ $r_{ij} = k$ is such that $c_{ij} = \min_{i<l\le j}\{w_{ij} + c_{i,l-1} + c_{l+1,j}\}$

$$c_{ij} = \min_{i<l\le j}\{w_{ij} + c_{i,l-1} + c_{l+1,j}\}$$

Dynamic Programming

word	break	case	char	do	return	switch	void
probability	0.22	0.18	0.20	0.05	0.25	0.02	0.08

break.. break		case..case		char.. char		do..do		return..return		switch..switch		void.. void	
0.22	break	0.18	case	0.20	char	0.05	do	0.25	return	0.02	switch	0.08	void
break.. case		case.. char		char..do		do.. return		return..switch		switch.. void			
0.58	break	0.56	char	0.30	char	0.35	return	0.29	return	0.12	void		
break.. char		case..do		char.. return		do.. switch		return.. void					
1.02	case	0.66	char	0.80	return	0.39	return	0.47	return				
break..do		case.. return		char.. switch		do.. void							
1.17	case	1.21	char	0.84	return	0.57	return						
break.. return		case.. switch		char.. void									
1.83	char	1.27	char	1.02	return								
break.. switch		case.. void											
1.89	char	1.53	char										
break.. void													
2.15	char												



$T(N) = O(N^3)$

Please read 10.33 on p.419 for an O(N^2) algorithm.

All-Pairs Shortest Path

Given a weighted directed graph $G = (V, E)$, where V is the set of vertices and E is the set of edges, and each edge $e = (u, v)$ has a weight $w(e)$. The **shortest path** from vertex u to vertex v is a path from u to v with the minimum total weight. The **all-pairs shortest path problem** is to find the shortest path from each vertex to every other vertex.