ADS notes

collected by wxb

1. AVL Tree

【background】 Binary Tree 平均搜索复杂度 $T(P) = O(height)$，但height的最差情况可能为$O(N)$。为提升search速度，引入AVL Tree，使height(search)始终保持$O(logN)$

【Definition】 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, T_R are height balanced
2. $|h_L - h_R| \leq 1, h_{Empty} = -1$

【Definition】 The balanced factor $BF(node) = h_L - h_R$. In an AVL Tree, BF(node) =1, 0, -1

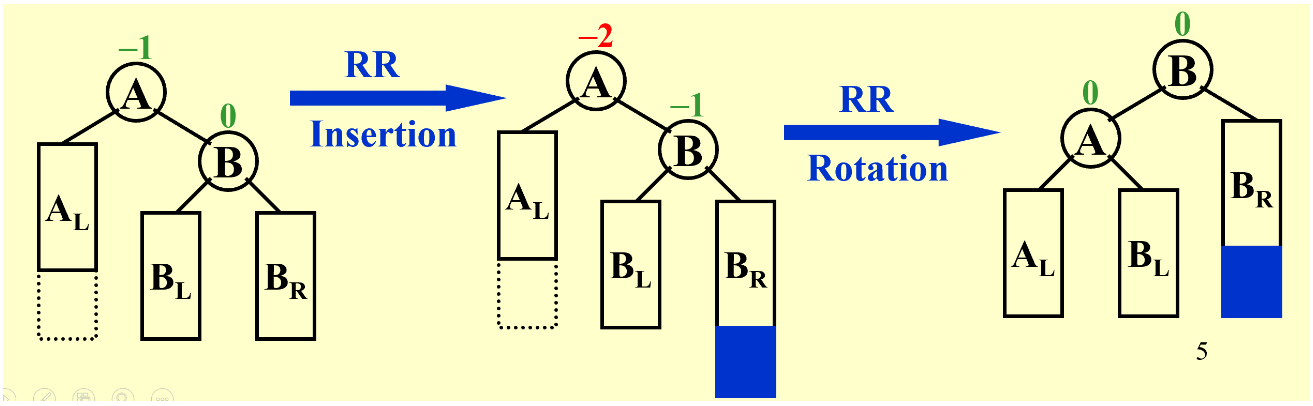
Search/Insert: $O(logN)$;

Rotation

LL Rotation 向右单旋

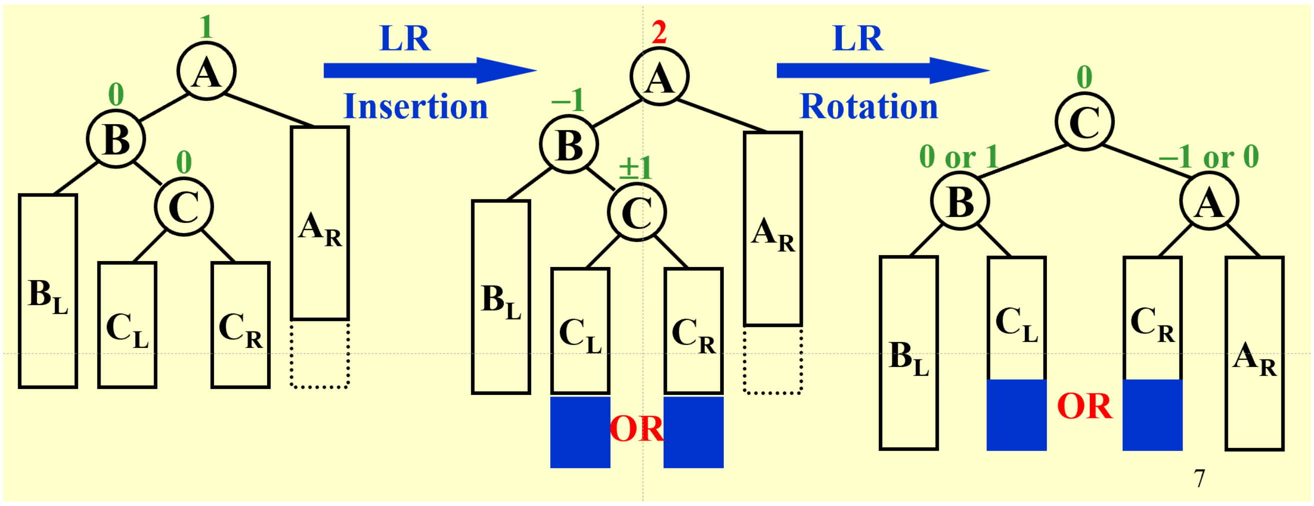
RR Rotation 向左单旋

```
// store
NotePtr rightN = root->right;
NodePtr rightlN = rightN->left;
// rotate
root->right = rightN->right;
rightN->left = root;
root->right = rightlN;
// update root and bf(node)
```



LR Rotation 左右双旋

先左旋，后右旋。记忆：C 成root, B, A左右护法，C_L, C_R左右拆分



RL Rotation 右左双旋

先右旋，后左旋。记忆：C 成root, A, B左右护法，C_L, C_R左右拆分

Note: *插入节点时即使不需要重平衡，也会影响部分节点的BF(node)，需要及时更新；或者增加height field*

$h_n = h_{n-1} + h_{n-2} + 1$，可由斐波那契数列推导得到 $h_n = O(logN)$

2. Splay Tree

【Target】Any M consecutive tree operations starting from an empty tree take at most $O(MlogN)$ time. And AVL rotation cannot satisfy the requirement.(单旋可能成链) -> 引入splay操作

Splay

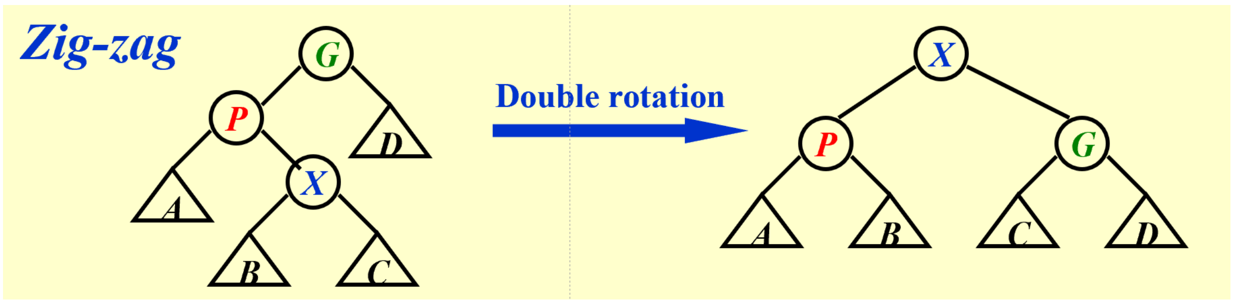
P(parent), X(insert node), G(grandparent)

P is the root

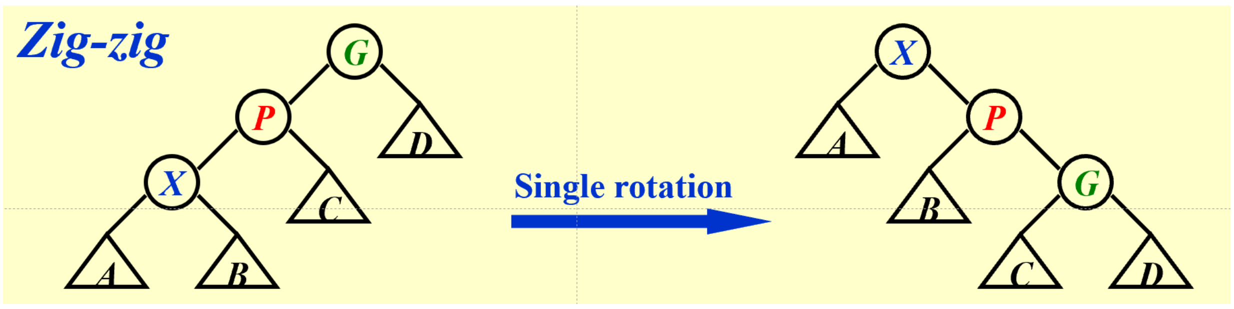
Rotate P and X

P isn't the root

1. Zig-zag Double rotation



2. Zig-zig Single rotation



不仅将插入的节点移到了根节点，还让大部分的节点深度减半

Operation

操作名称	时间复杂度	说明
Find(X)	$O(1)$	X is root
Remove(X)	$O(1)$	left and right tree left
FindMax(T_L)	$O(height)$	找到左子树最大节点
Delete(X)	$O(logN)$	上述3步后，make T_R the right child of the root of T_L

3. Amortized Analysis

$worst-case\ bound \geq amortized\ bound \geq average-case\ bound$

Amortized bound: probability is not involved

Aggregate analysis

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.

$T_{amortized} = T(n)/n$

Accounting method

如果一个操作的amortized cost超过了实际花销，可以将这个差作为信用分配给特定对象 savings credits

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 $\sum_{i=1}^{n} \hat{c}_i \le \sum_{i=1}^{n} c_i$

$T_{amortized} = T(n)/n = \frac{\sum_{i=1}^{n} \hat{c}_i}{n}$

Potential method

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

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

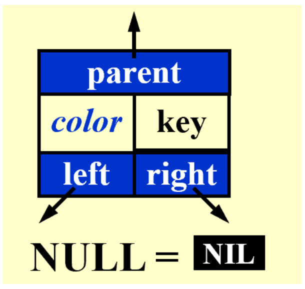
【Theorem】 The amortized time to splay a tree with root T at node X is at most $3(R(T) – R(X)) + 1 = O(logN)$.

4. Red-Black Tree

【Target】Balanced binary search tree 其中*平衡*是由红黑树本身的特性实现的，并不是绝对的高度差不超过1

【Definition】满足下列5条要求

1. Every node is either red or black.
2. the root is black.
3. Every leaf (NIL) is black.
4. If a node is red, then both its children are black.
5. For each node, all simple paths from the node to descendant leaves contain the same number of black nodes.



【Definition】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).

Insert

默认结点以红色插入，之后再调整解构和颜色

空树

直接插入，设置为黑

插入结点的父结点为黑色

直接插入，设置为红色

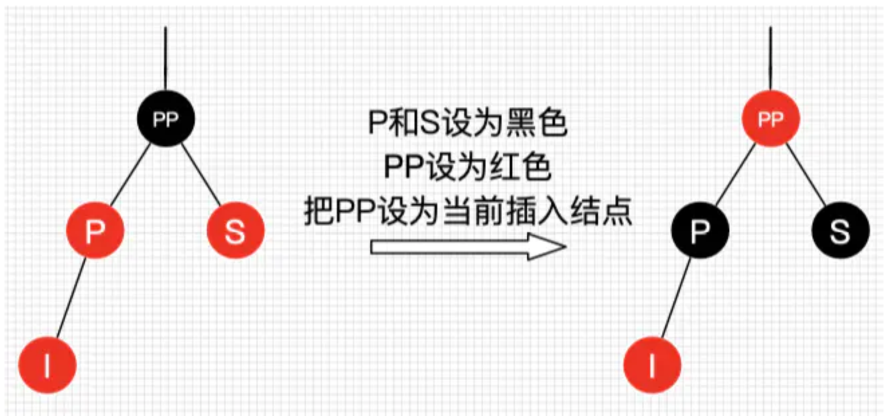
插入结点的父结点为红色

根节点是黑色，因此当父结点为红色时，父结点必不是根结点，因此祖父结点必然存在，且必然为黑色

祖父结点 PP, 父结点 P, 叔叔结点 S, 插入结点 I

1. 叔叔结点S存在且为红色

红结点的孩子必为黑。因此祖父结点 PP 必然是黑色。



Note：如果祖父结点 **PP** 的父结点为红色，则需要继续自平衡

Note：如果祖父结点 **PP** 就是根结点，设置为红色后仍需要设置成黑色 -> 唯一增加黑色结点层数的插入情景

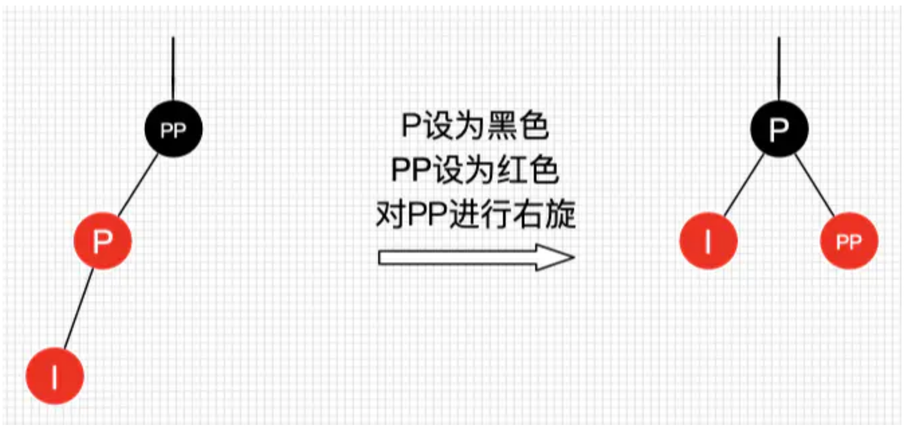
Note：红黑树的生长是自底向上的，普通的二叉搜索树是自顶向下生长的。

2. 叔叔结点S为黑色或不存在

2.1 父节点P是祖父结点PP的左子树

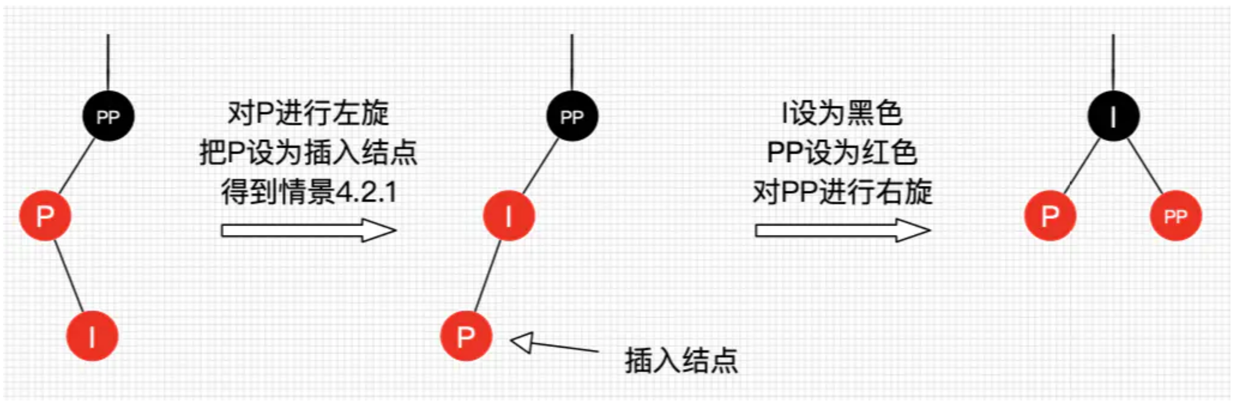
1. 插入结点I是父结点P的左子树

   以P为支点右旋，P设置为黑，PP设置为红



2. 插入结点I是父结点P的右子树

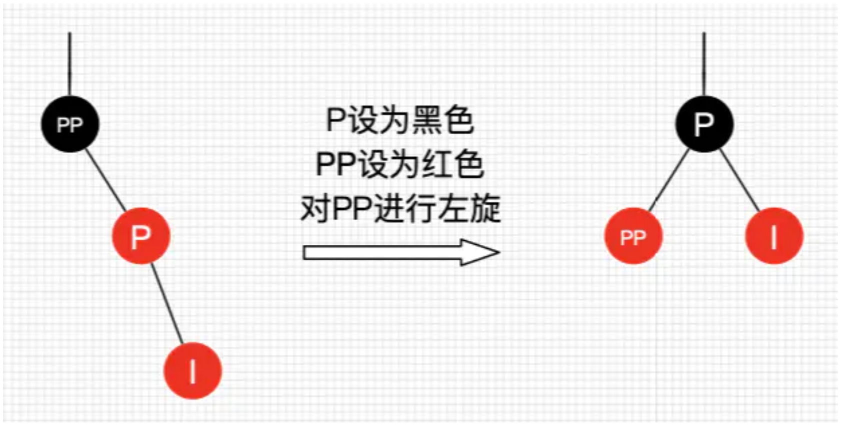
   以P为支点左旋，以I为支点右旋，I设置为黑，PP设置为红



2.2 父节点P是祖父结点PP的右子树

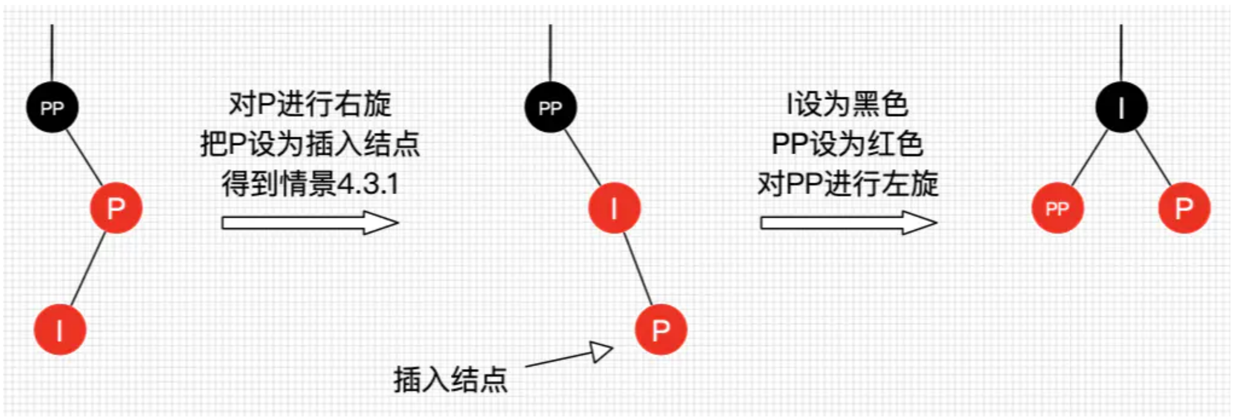
1. 插入结点I是父结点P的右子树

   以P为支点左旋，P设置为黑，PP设置为红



2. 插入结点I是父结点P的左子树

   以P为支点右旋，以I为支点左旋，I设置为黑，PP设置为红



Delete

查找目标结点 -> 自平衡 -> 替代

删除结点被替代后，在不考虑结点键值的情况下，对于子树来说，可认为删除的是替代节点

1. 目标结点无子树

直接删除，无需替代，自平衡

2. 目标结点只有一个子树

用子树代替即可；

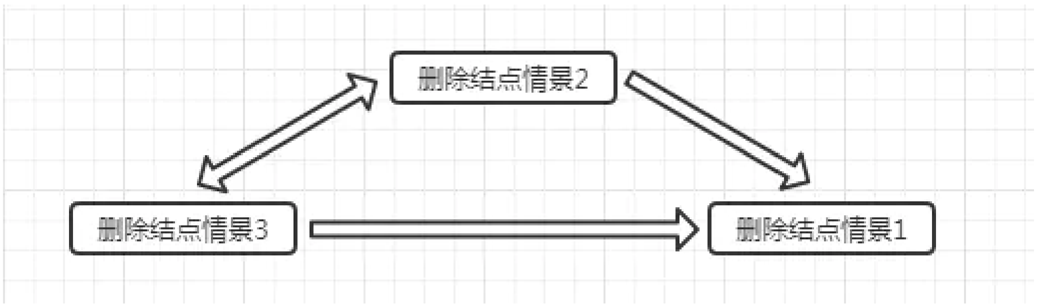
1. 若子树有两个子节点，则转到3；
2. 若只有一个子节点，则继续2；
3. 若子树没有孩子，则转到1 -> 结束

3. 目标结点有两个子树

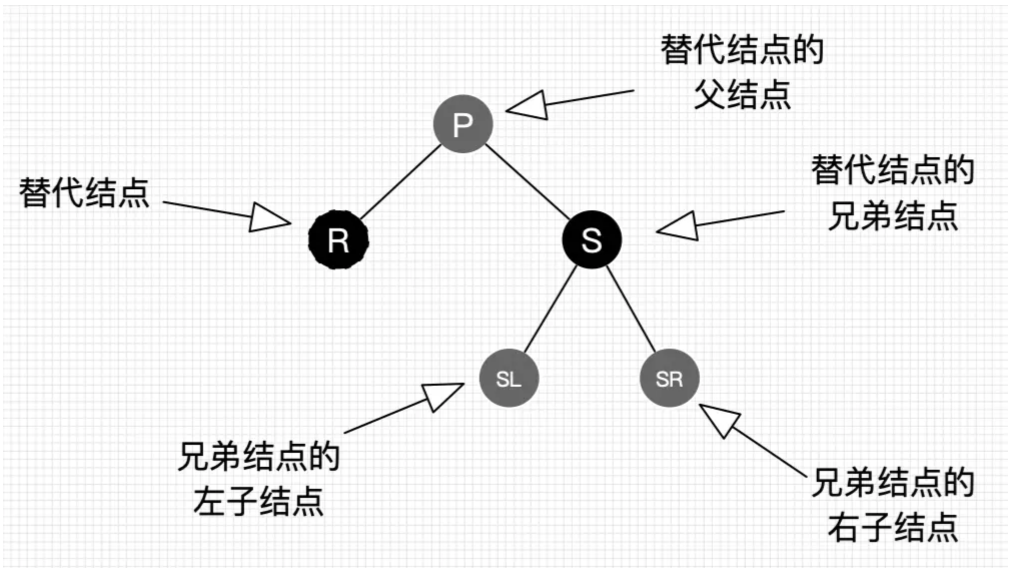
查找后继节点(大于删除结点的最小结点)替换删除结点

1. 若后继结点有右子树，则转到2
2. 若后继结点没有孩子，则转到1 -> 结束

因此，删除也可以迭代实现



替代结点R，替代结点的父结点P，兄弟节点S



1. 替换结点R是红色

把替换结点设为目标结点的颜色即可

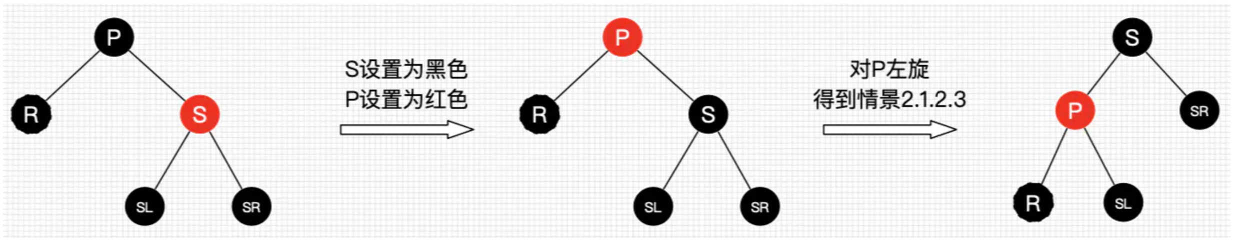
2. 替换结点R是黑色

2.1 替换结点R是父结点P的左结点

1. 兄弟结点S为红色

   由红黑树性质可知，父结点P和S的子节点都必为黑色

   将兄弟结点S设置为黑，父结点P设置为红 -> 以P为支点左旋

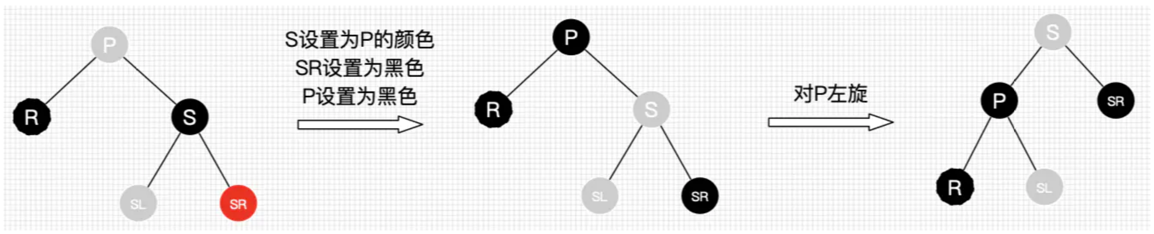
   

2. 兄弟结点S为黑色

   1. 替换结点的兄弟结点的右子结点是红色，左子结点任意颜色

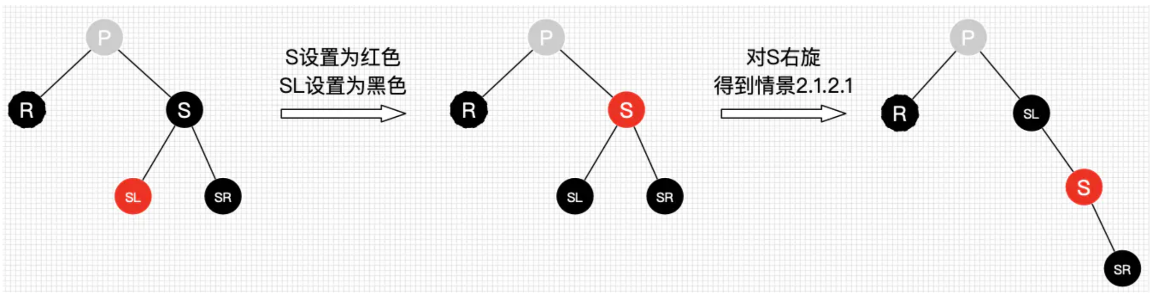
      兄弟结点S设置为父结点P的颜色，父结点P和兄弟结点的右节点SR都设置为黑色 -> 以P为支点左旋

      下图是第一次替换和自底向上处理的情况，如果只考虑第一次替换，根据红黑树性质，SL肯定是红色或Nil，所以最终树是平衡的

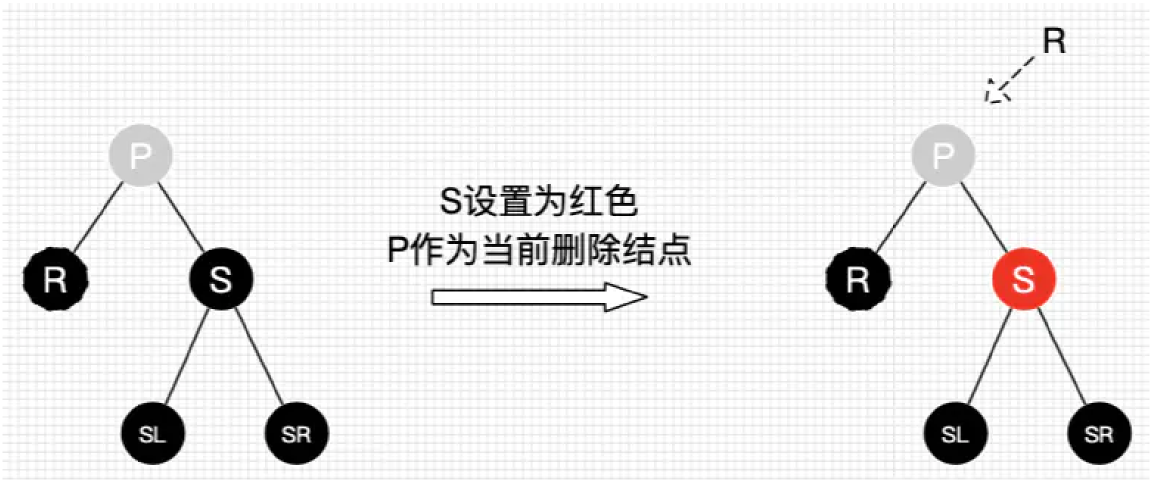
   2. 替换结点的兄弟结点的右子结点是黑色，左子结点为红色

      将兄弟结点S设置为红色，兄弟结点的左子结点SL设置为黑色 -> 以S为支点右旋 -> 转到2.1



3. 替换结点的兄弟结点的子结点都为黑色

   将S设置为红色，将P当作替代结点 -> 自底向上处理，寻找适配情景



Rotation次数

Tree	Insertion	Deletion
AVL Tree	≤ 2	$O(logN)$
RB Tree	≤ 2	≤ 3

5. B+ Tree

【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 non-leaf nodes (except the root) have between $\lceil M/2 \rceil$ and M children.**
- All leaves are at the same depth.

  Assume each non-root leaf also has between $\lceil M/2 \rceil$ and M children

Note: 所有实际数据都存储在叶结点中；每个内部结点都有M个指向孩子的指针，并且按照大小顺序存储了第二个到第M个孩子键值中的最小值

Insertion

```
Btree  Insert ( ElementType X,  Btree T ) {
    // 查找合适的位置
    Search from root to leaf for X and find the proper leaf node;
    // 插入结点
    Insert X;
    // 当前结点若有 M+1 个键值(超过规定)
    while ( this node has M+1 keys ) {
        // 切分成两个节点
        split it into 2 nodes with ⌈(M+1)/2⌉ and ⌊(M+1)/2⌋ keys,
respectively;
        // 如果已经到根节点
        if (this node is the root)
            // 创建一个新的有两个孩子的根节点
            create a new root with two children;
        // 自底向上检查
        check its parent;
    }
}
```

删除操作也类似，但如果根节点的孩子少于2个，则要移除根节点

$$T_{find}(M,N)=O(logN),\ Depth(M,N)=O(\lceil log_{\lceil M/2\rceil}N\rceil)$$

6. Inverted File Index

【Definition】Inverted file contains a list of pointers (e.g. the number of a page) to all occurrences of that term in the text.

Index Generator

Token Analyzer, Stop Filter -> Vocabulary Scanner -> Vocabulary Inserter -> Memory management

```
while ( read a document D ) {    // 文件未结束
    while ( read a term T in D ) {  // 读入term
        if ( Find( Dictionary, T ) == false )    // 若字典中没有，则插入
            Insert( Dictionary, T );
        Get T's posting list;    // 获取该term的<出现>列表
        Insert a node to T's posting list;  // 向列表中插入新节点<当前位置>/<
页码>/...
    }
}
Write the inverted index to disk;   // 写入磁盘
```

若内存不足，则可按如下方式操作

```
BlockCnt = 0; // 计数用
while ( read a document D ) {
      while ( read a term T in D ) {
        // 如果超出内存，则将这部分lists写入磁盘，并释放内存
        if ( out of memory ) {
          Write BlockIndex[BlockCnt] to disk;
          BlockCnt++;
          FreeMemory;
        }
        if ( Find( Dictionary, T ) == false )
          Insert( Dictionary, T );
        Get T's posting list;
        Insert a node to T's posting list;
      }
}
// 合并，sorted
for ( i=0; i<BlockCnt; i++ )
      Merge( InvertedIndex, BlockIndex[i] );
```

Word Stemming 分词

英文中不同时态、人称的单词应当归属于同一类，需要转换

Stop Words

过于常见的词对于检索没有意义，如"a", "the"等，可以从原文档中忽略

Search

访问term术语时，可以用搜索树(B+, B-, Tries...)或者哈希，各有优缺点

distributed indexing 分布式索引

Term-partitioned index e.g. A-C, D-F...

Document-partitioned index e.g. 1-10000, 10001-20000...

Dynamic Index

总是有新文档加入，旧文档删除，因此需要不断更新索引

引入辅助索引auxiliary index，由 main index 和 auxiliary index 一起进行索引；之后再re-index

Compression

去除 stop words，直接将term连接起来；哈希索引

Thresholding

阈值编码(权重分配)

1. document: 只检索权重最高的前X个文档
    - 会错过部分相关文档
    - 对于布尔查询来说不可行
2. query: 按照频率升序对query term进行排序，根据原始query term的百分比进行查找

搜索引擎的要求

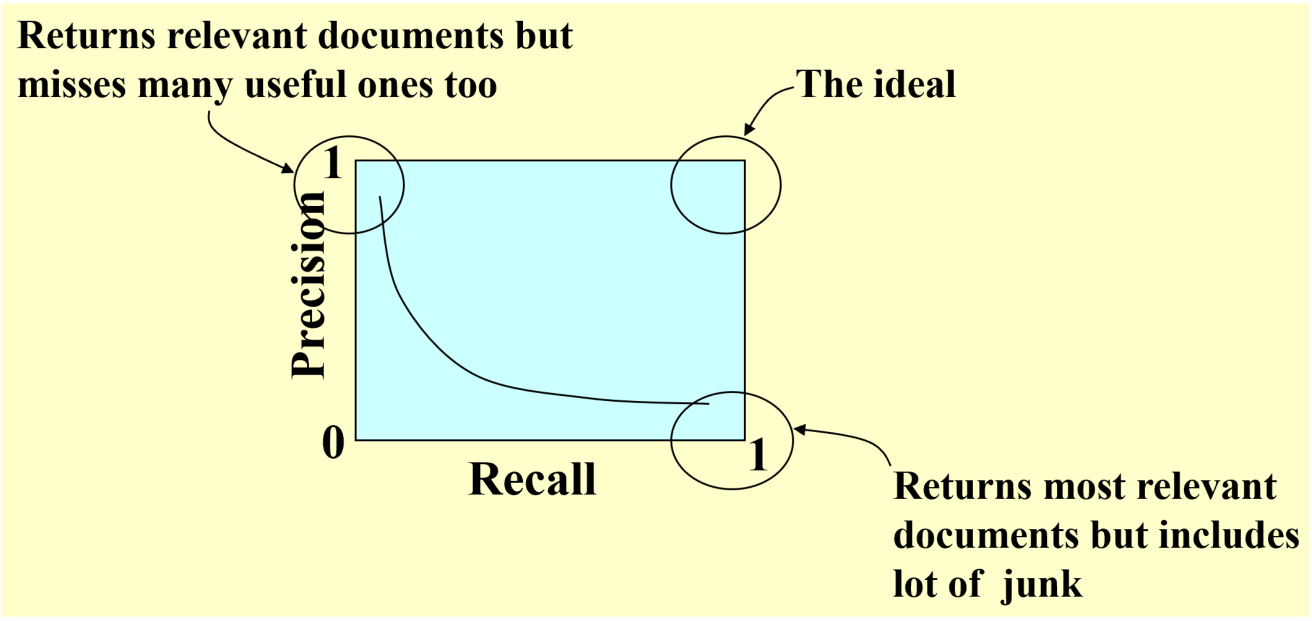
- speed of index: Number of documents/hour
- speed of search: Latency, function of index size
- Expressiveness of query language: 表达复杂信息的能力，复杂检索的速度
- User happiness: Data(响应时间，索引空间) & Information(relevant)

Relevance

	Relevant	Irrelevant
Retrieved	R_R	I_R
Not Retrieved	R_N	I_N

Precision 精确性 $P = R_R/(R_R + I_R)$

Recall 召回率 $R = R_R/(R_R + R_N)$


Returns relevant documents but misses many useful ones too
The ideal
1
Precision
0
Recall
1
Returns most relevant documents but includes lot of junk

7. Leftist Tree

【Definition】The null path length, Npl(X), of any node X is the length of the shortest path from X to a node without two children. Define Npl(NULL) = −1.

Note: Npl(X) = min { Npl(C) + 1 for all C as children of X }

【Definition】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 that of the right child.

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

Merge

Insert 只是merge的特殊形式

```
PriorityQueue  Merge ( PriorityQueue H1, PriorityQueue H2 ) {
    if ( H1 == NULL )   return H2;  // 若为空，则返回另一个堆即可
    if ( H2 == NULL )   return H1;
    if ( H1->Element < H2->Element )  return Merge1( H1, H2 );  // 若均不空，则进行merge
    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 {  // 若不是单结点，则3步走
        H1->Right = Merge( H1->Right, H2 ); // 1. sort右路径，左边不动
        if ( H1->Left->Npl < H1->Right->Npl )   // 2. 若不符合leftist，则swap children
            SwapChildren( H1 );
        H1->Npl = H1->Right->Npl + 1;   // 3. update npl，不然npl会乱掉
    }
    return H1;
}
```

Delete Min

1. 删除根结点，因为二叉堆的性质就是根结点是最小元素
2. Merge左右子树

Leftist Tree是可并堆的一种，merge时间复杂度：$T_p = O(logN)$，普通二叉堆则为$O(N)$

8. Skew Heap

a simple ver of the leftist heap

【Target】Any M consecutive operations take at most $O(MlogN)$ time

Merge

递归，每次都交换左右结点，除非没有孩子可交换

优点：不需要额外的空间存储Npl，也不需要判断是否需要交换(废话因为每次都交换)

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.

如果结点P右子树的后代(包括右孩子)占P的后代的一半以上，则为heavy，反之light

The only nodes whose heavy/light status can change are nodes that are initially on the right path.

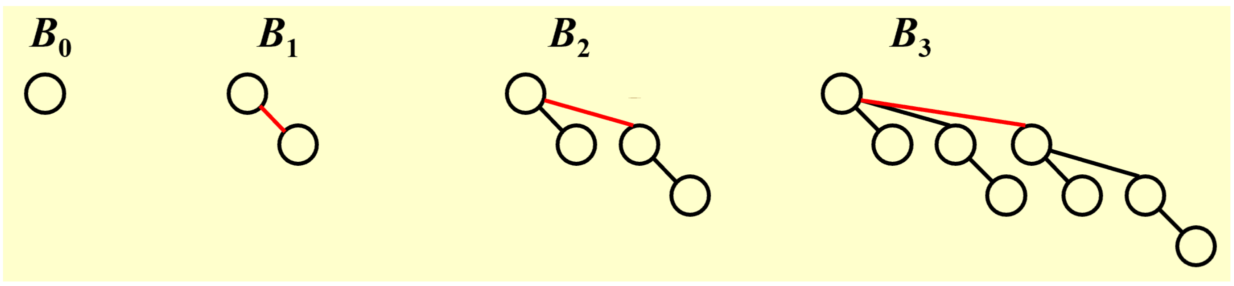
$$\begin{gathered}
D_i = the\ root\ of\ the\ resulting\ tree \\
\Phi(D_i) = number\ of\ heavy\ nodes \\
Before\ Merge:\ \Phi_i = h_1 + h_2 + h\ \rightarrow\ T_{worst} = l_1 + h_1 + l_2 + h_2 \\
After\ Merge:\ \Phi_{i+1} \le l_1 + l_2 + h\ \rightarrow\ T_{amortized} = T_{worst} + \Phi_{i+1} - \Phi_i \le 2(l_1 + l_2) \\
l = O(logN)\ \rightarrow\ T_{amortized} = O(logN)
\end{gathered}$$

9. Binomial Queue

【Background】对于普通的二叉堆而言，从空开始插入N个元素总耗时为$O(N)$，平均下来每个插入耗时$O(1)$，因此leftist Tree, Skew Heap的$O(logN)$不是最好的，需要优化。

【Definition】A binomial queue is 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}.



B_k consists of a root with k children. B_k has exactly 2^k nodes. The number of nodes at depth d is $\binom{k}{d}$(二项式系数).

任何大小的优先级队列都可以由二叉树的集合唯一地表示

Operation

FindMin

最小的键值总是存储在roots中。最多有$\lceil logN \rceil$个roots -> $T_p = O(logN)$

Note: 如果标记最小值，并且每次变换时更新，则可以降至$O(1)$

Merge

类似于二进制加法，进位保留，并按照树高对 binomial queue 排序

```
BinQueue  Merge( BinQueue H1, BinQueue H2 ) {
    BinTree T1, T2, Carry = NULL;
    if ( H1->CurrentSize + H2->CurrentSize > Capacity ) // 如果溢出，报错
        ErrorMessage();
    H1->CurrentSize += H2-> CurrentSize;     // 更新大小
    for ( int i=0, int j=1; j <= H1->CurrentSize; i++, j*=2 ) {
        T1 = H1->TheTrees[i]; T2 = H2->TheTrees[i]; /*current trees, T1是H1的左子树, T2是H2 */
        switch( 4*!!Carry + 2*!!T2 + !!T1 ) {
        case 0: /* 000 */
        case 1: /* 001 */  break;
        case 2: /* 010 */  H1->TheTrees[i] = T2; H2->TheTrees[i] = NULL; break;
        case 4: /* 100 */  H1->TheTrees[i] = Carry; Carry = NULL; break;
        case 3: /* 011 */  Carry = CombineTrees( T1, T2 );
                        H1->TheTrees[i] = H2->TheTrees[i] = NULL; break;
        case 5: /* 101 */  Carry = CombineTrees( T1, Carry );
                        H1->TheTrees[i] = NULL; break;
        case 6: /* 110 */  Carry = CombineTrees( T2, Carry );
                        H2->TheTrees[i] = NULL; break;
        case 7: /* 111 */  H1->TheTrees[i] = Carry;
                        Carry = CombineTrees( T1, T2 );
                        H2->TheTrees[i] = NULL; break;
        }
    }
    return H1;
}
```

```
BinTree  CombineTrees( BinTree T1, BinTree T2 ){
    /* merge equal-sized T1 and T2 */
    if ( T1->Element > T2->Element )
        /* attach the larger one to the smaller one */
        return CombineTrees( T2, T1 );
    /* insert T2 to the front of the children list of T1 */
    T2->NextSibling = T1->LeftChild;
    T1->LeftChild = T2;
    return T1;
}
```

Insert

merge的特殊情形

Note: If the smallest nonexistent binomial tree is B_i , 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**.

The worst case time for each insertion is $O(logN)$, $T_{amortized} = 2$.

DeleteMin

FindMin -> Remove包含最小结点的树，剩下的形成新的Binomial Queue -> Remove最小结点(根结点)，形成新Binomial Queue -> Merge两个Queue

```
ElementType  DeleteMin( BinQueue H ) {
    BinQueue DeletedQueue;
    Position DeletedTree, OldRoot;
    ElementType MinItem = Infinity;  /* the minimum item to be returned 
*/
    int MinTree; /* MinTree is the index of the tree with the minimum 
item */

    if ( IsEmpty( H ) ) {   // 队列为空，无法删除，报错
        PrintErrorMessage(); return -Infinity;
    }
    /* Step 1: find the minimum item */
    for ( int i = 0; i < MaxTrees; i++) {
        if( H->TheTrees[i] && H->TheTrees[i]->Element < MinItem ) { // 结
点存在且小于当前最小值，更新MinItem
            MinItem = H->TheTrees[i]->Element;
            MinTree = i;
        }
    }
    /* Step 2: remove the MinTree from H => H’ */
    DeletedTree = H->TheTrees[MinTree];
    H->TheTrees[MinTree] = NULL;
    /* Step 3.1: remove the root */
```

```
        OldRoot = DeletedTree;
        DeletedTree = DeletedTree->LeftChild;
        free(OldRoot);
        /* Step 3.2: create H” */
        DeletedQueue = Initialize();
        DeletedQueue->CurrentSize = ( 1<<MinTree ) - 1;   // 2MinTree - 1
        for ( int j = MinTree - 1; j >= 0; j - - ) {
            DeletedQueue->TheTrees[j] = DeletedTree;
            DeletedTree = DeletedTree->NextSibling;
            DeletedQueue->TheTrees[j]->NextSibling = NULL;
        }
        H->CurrentSize -= DeletedQueue->CurrentSize + 1;
        /* Step 4: merge H’ and H” */
        H = Merge( H, DeletedQueue );
        return MinItem;
    }
```

$T = O(logN) + O(1) + O(logN) + O(logN) = O(logN)$

为了操作简单，二项树的子树采用递减的顺序(二项队列存储二项树仍按照高度递增排序)。因为新的子树将会是最大的子树，将它作为第一个儿子比将它作为最后一个儿子(要遍历孩子链表将新的子树放在链表末尾)更方便

---

10. Backtracking

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

- **Depth First Search**
- *Minimax Strategy**
- **α-β pruning**

  when both techniques are combined. In practice, it limits the searching to only $O(\sqrt{N})$ nodes, N is the size of the full game tree.

11. Divide and Conquer

1. **Divide** the problem into a number of sub-problems
2. **Conquer** the sub-problems by solving them recursively
3. **Combine** the solutions to the sub-problems into the solution for the original problem

General recurrence: $T(N) = aT(N/b) + f(N)$

Methods to solve the recurrence

Ignore if (N / b) is an integer or not, always assume $T(n) = \Theta(1)$ for small n

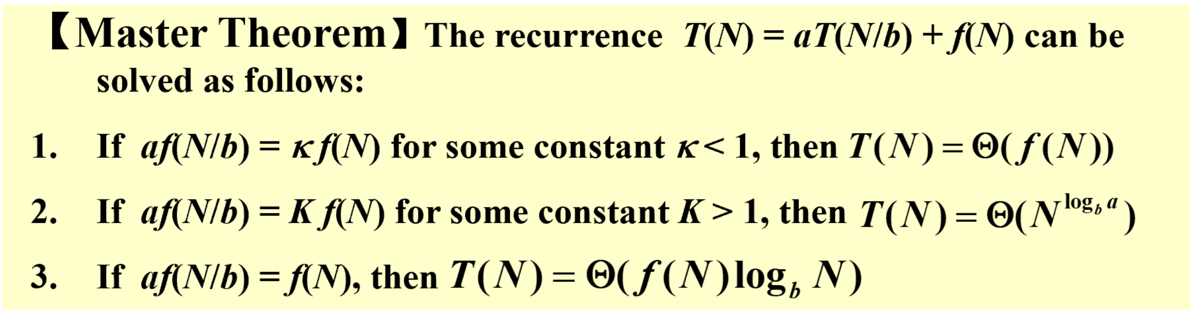
1. Substitution method

   猜测一个T(N), 然后通过归纳法验证。结果必须严格遵循假设才算成立。
2. Recursion trees method

   递归树求解。有些递归式分解不对称，可以先递归前几步，帮助猜测T(N)，再使用替代法
3. Master method

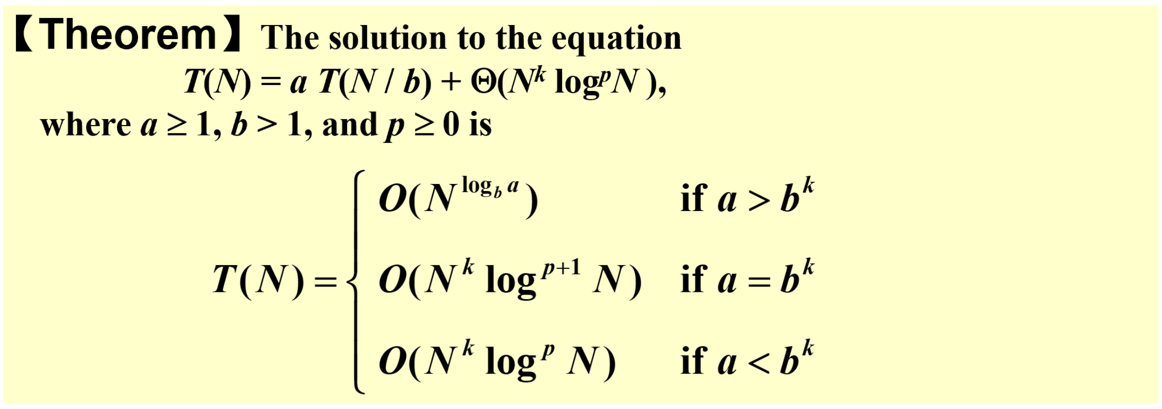
   【Master Theorem】Let $a \geq 1$ and $b > 1$ be constants, let f(N) be a function, and let T(N) be defined on the nonnegative integers by the recurrence $T(N) = aT(N/b) + f(N)$. Then:

   - If $f(N) = O(N^{log_a b - \varepsilon})$ for some constant $\varepsilon > 0$, then $T(N) = \Theta(N^{log_a b})$
   - If $f(N) = \Theta(N^{log_a b})$, then $T(N) = \Theta(N^{log_a b} logN)$
   - If $f(N) = O(N^{log_a b + \varepsilon})$ for some constant $\varepsilon > 0$, and if $af(N/b) < cf(N)$ for some constant c < 1 and all sufficiently large N, then $T(N) = \Theta(f(N))$



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



【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

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

12. Dynamic Programming

需要满足的条件

- 无后效性
- optimal sub-solution 最优子结构
- 重叠子问题：若子问题不重叠，则使用动态规划无法带来时间上的节省

分解出子问题 -> 确定最优解决方案 -> 递归决定最优值 -> 按一定顺序计算、更新 -> 重构最优解