1. AVL Tree

(1) Definition: $|h_L - h_R| \le 1$

(2) The balance factor $BF(node) = h_L - h_R$ (某节点的两孩子的BF可以同时为1或-1)

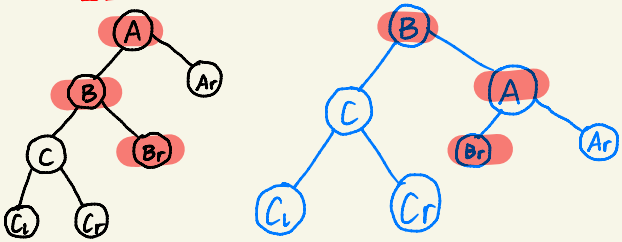
(4) n_h：高度为 h 的 AVL Tree 中最少结点数。

$\Rightarrow n_h = n_{h-1} + n_{h-2} + 1$

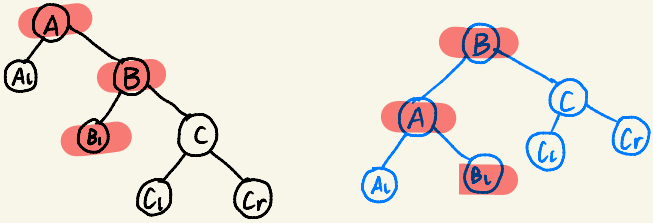
其中 $n_0 = 1$, $n_1 = 2$, $n_2 = 4$

只有黄光笔标注部分形状发生变化

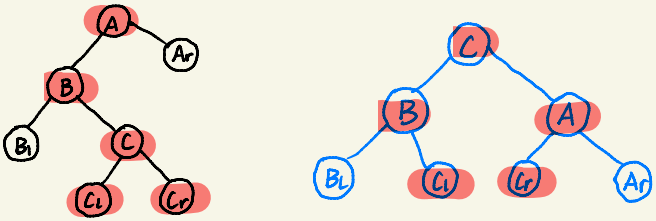
(3) LL



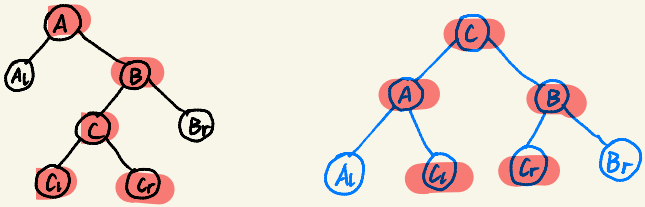
RR



LR



RL



更新 C 为 root，并将 C 的左右子树插到 C 的 parent 和 grandparent 上。

2. Splay Tree (half the depth of nodes visited)

(1) Definition: Rotate every node to root when it's visited.

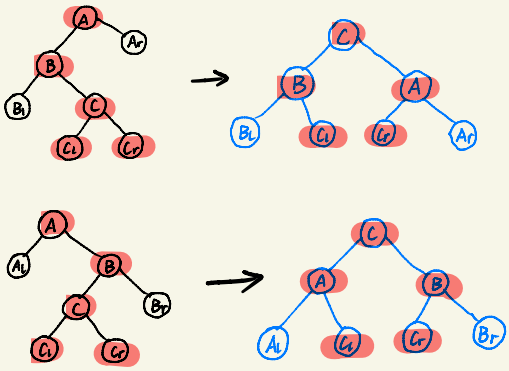
(2) Visit X, P is parent, G is grandparent.

① P is root: Rotate P and X (Just like LL、RR)

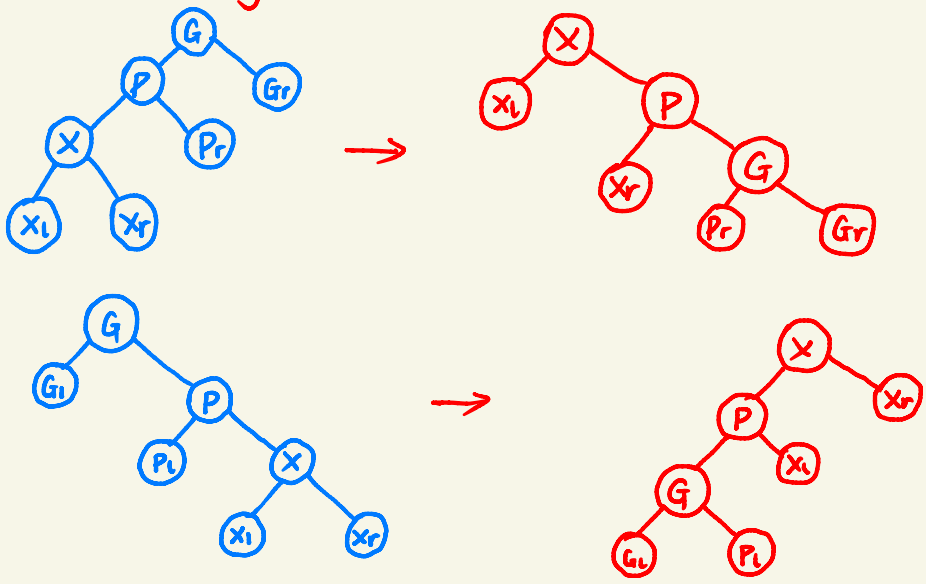
② P isn't root: (C is X, B is P, A is G) visit C.

(I) Zig-Zag:

Exactly same as LR or RL.



(II) Zig-Zig:



把 X、P、G 看成 翘翘板!

(3) Delete X:

① Find X (X will be at the root) ② Remove X

③ FindMax(T_L) (The max in T_L will be the root) ④ Make T_R right child of root of T_L

3. Amortized Analysis 摊还分析

(1) worst-case bound ≥ amortised bound ≥ average-case bound

- Aggregate analysis 聚合分析：对一个 n 操作序列最坏情况下花费总时间为 $T(n)$，则摊还代价为 $T(n)/n$.
- Accounting Method 记账法：若可以满足 $\sum_{i=1}^{n} \hat{c}_i \geq \sum_{i=1}^{n} c_i$ (amortized cost exceeds actual cost)
  则摊还代价为 $\frac{\sum_{i=1}^{n} \hat{c}_i}{n}$.
  (If $\hat{c}_i > c_i$, then credit $= \hat{c}_i - c_i$)
- Potential Method: 势能法
  $\hat{c}_i - c_i = \text{credit} = \Phi(D_i) - \Phi(D_{i-1})$
  Φ: potential function. $\Rightarrow \sum_{i=1}^{n} \hat{c}_i = \sum_{i=1}^{n} c_i + \underbrace{\Phi(D_i) - \Phi(D_0)}_{\geq 0}$

$\Rightarrow$ A good potential function should always assume its minimum at the start of the sequence.

Splay tree 的具体均摊分析看补充文档

Red-Black Tree

正常非空黑结点: internal nodes.

external nodes.

1. Definition:

① Root is black. ② Every leaf (NULL) is black.

③ 红结点的子结点都是黑结点。

④ ∀node, 至树底的所有路径上黑结点数相同。

每一条路径上红结点数不多于黑结点数.

2. Black-Height

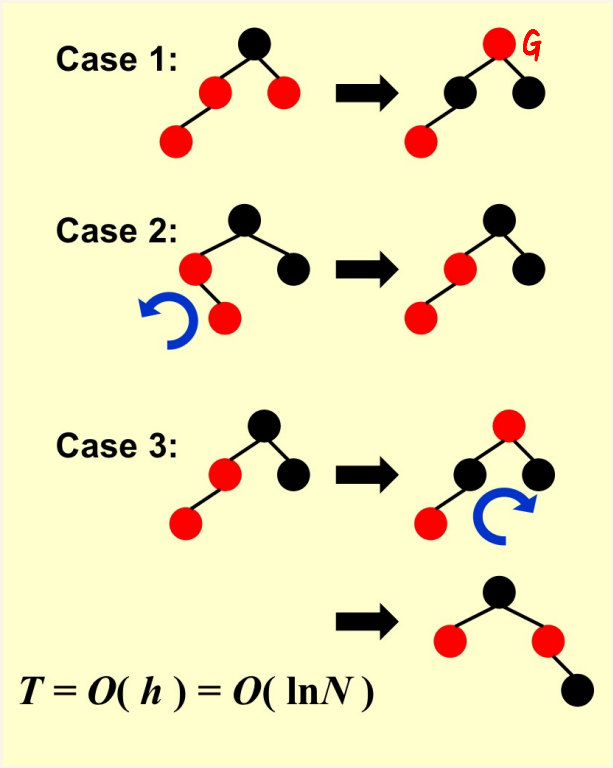
Bh(x) is the number of black nodes on any simple path from x (x not included) down to a leaf.

但是要把 null 空结点算在内。

3. 对于一个 N 结点的红黑树, 高度 $\leq 2\ln(N+1)$

证明: lemma 1: $Bh(Tree) \geq \frac{1}{2}h(Tree)$

lemma 2: $sizeof(x) \geq 2^{bh(x)}-1$



红黑树插入：

第1步，插入节点x，记为红。

若x的P是黑，插入结束。

若x的P也是红，则

判断：①若x的叔叔为红，则令x的父亲与叔叔为黑、x的爷爷为红，插入结束。

②若x的叔叔为黑：

1°若x是叔叔的近侄，先转到远侄，再做2°。

2°若x是远侄：令x的P为黑，GrandP为红，然后以P为顶点右旋。

注意：始终要保持RB树性质。

比如：向空树中插入第一个node，必须是黑，对case1，若G节点是root，则需要染黑。

	Number of Rotations	
	AVL	Red-Black Tree
Insertion	≤ 2	≤ 2
Deletion	$O(\log N)$	≤ 3

红黑树的删除

1. 执行标准 BST 删除操作（记被删节点为 P）

① 直接删 leaf：reset its parent link to NULL.

② 删仅有 1 孩子的结点：replace the node by its single children.

③ 删除有 2 孩子的结点：replace the node by 左子树最大结点 or 右子树最小结点（记为 X）

然后将 X 从树中删除（又回到 3①、②）

（X 至多仅有 1 孩子）

如果有孩子，X 一定是黑色。

对红黑树而言：

① 被删除节点 P 保留其原来的颜色，但数据被替换为 X 的数据。实际操作是删除节点 X（X 的孩子至多仅有 1 个）。直接执行 BST 删除操作（①②），将 X 用孩子代替（孩子可能是 NULL），然后令被替代后的新 X 颜色为黑。

② 根据原来 X 及其孩子的颜色分类：

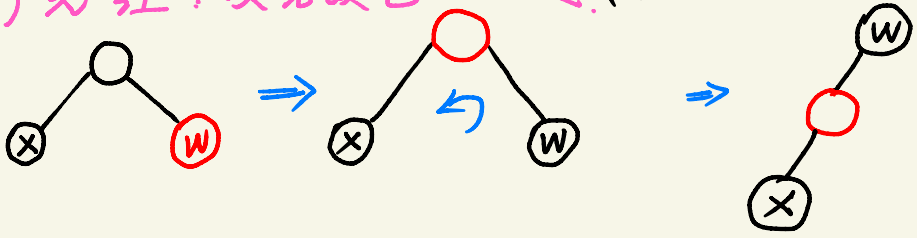
(1) 若原 X 和其孩子一红一黑：以上操作已满足。

(2) 若均为黑色：将该新黑色结点重新记为 X，且这个节点的黑高为 2！

现在任务是"去 X"，消除这种需要算 2 个黑高的点

根据新 X 的情况分类：

(I) 兄红：父兄换色父左旋（然后接着推）



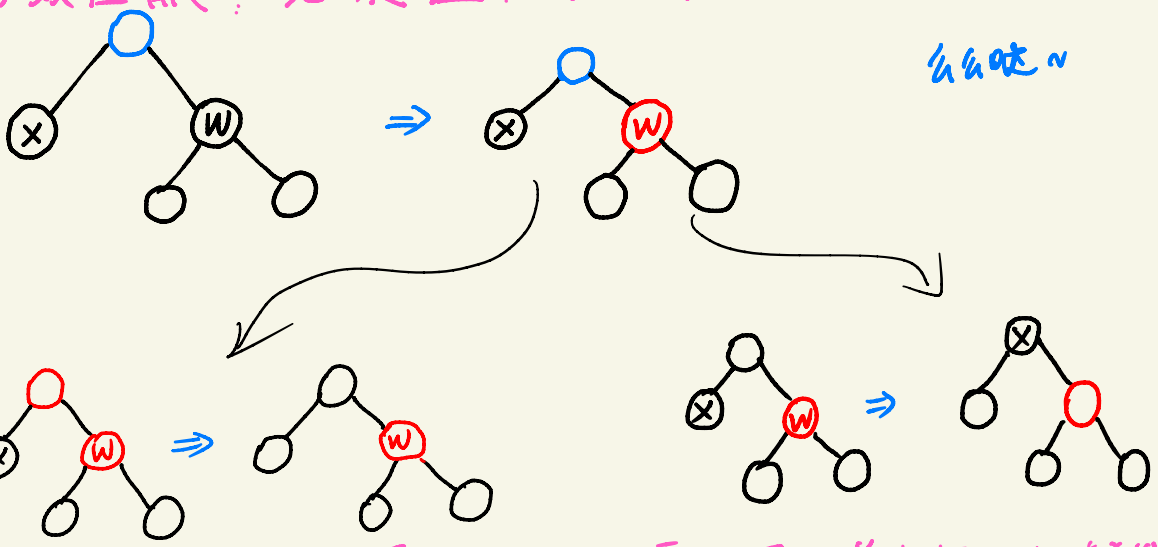
现在已转化成了兄黑的情况

(蓝色代表红/黑)

宝宝你写字真可爱~

么么哒~

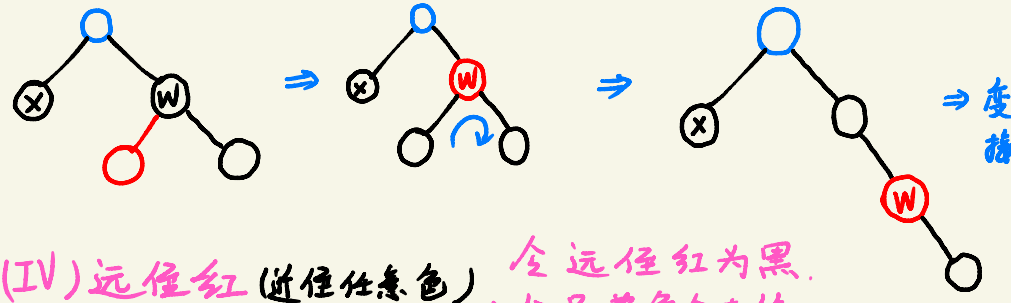
(II) 双任黑: 兄染红, 再分类



原来父为红: 直接染黑即可去X.

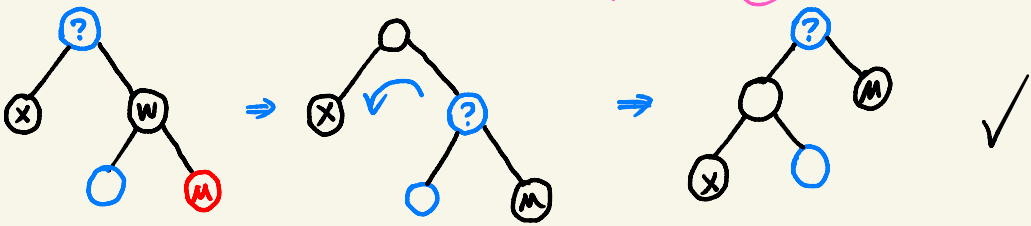
原父为黑: 将父标为X, 接着推

(III) 近任红, 远任黑: 兄任换色兄右旋, 变成远任红.



⇒ 变成(IV) 接着推

(IV) 远任红 (近任任意色): 令远任红为黑, 父兄换色父左旋



✓

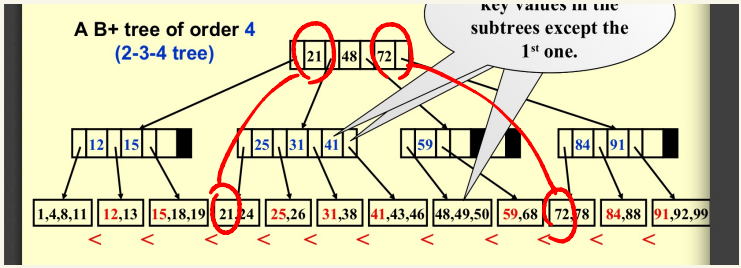
B+ Tree

1. Definition: B+ Tree of M order

① 根节点要么是 leaf，要么有 [2, M] 个子结点。

② All noneleaf nodes (except root) have [⌈M/2⌉, M] children

③ 所有叶节点深度相同。



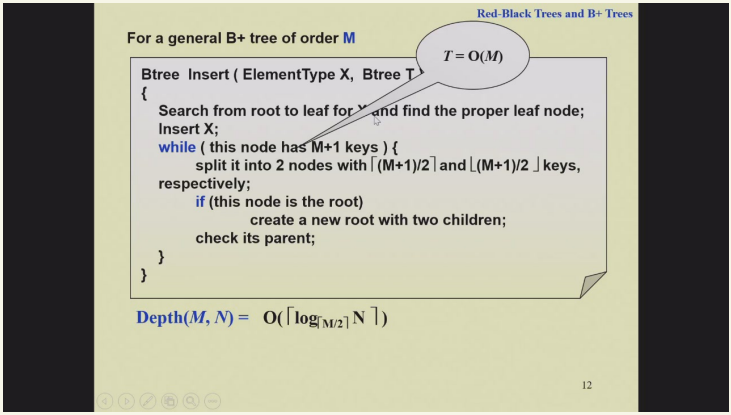
root

level 1

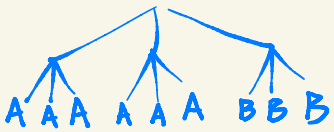
level 2

注意：root = level 2 中的 key，而非 level 1。
(从索引角度理解)

具体操作看 ppt



21个结点的 B^+ 树（3阶），最多 4 个度为 3 的 node



A 代表有 2 个 key

B 代表有 3 个 key

Inverted File Index

No 编号 Term 词 Times；(Doc ID，place) 出现次数（出现的文章编号，出现在文章中的具体位置）

Stop words 停用词（这些词过于普遍而导致没必要检索）

Word stemming 词干分析（把同一单词的不同形式归一为词干）

搜索单词的方式：hash / search tree

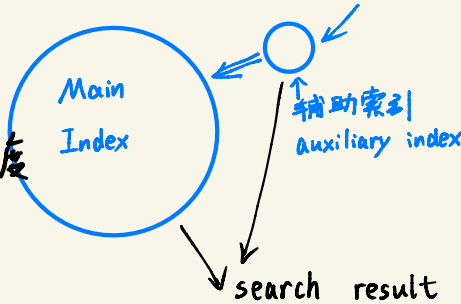
recall ↑
precision ↓

Distributed Indexing：分布式

① term partitioned：按单词分，把包含某单词的文章存在一起

② document partitioned：正常文章编号

③ Dynamic：可插入、删除 New Docs



How fast it index：往库中加新文章的速度

How fast it search：搜索速度

Compression：

若需存储的位置信息为 2, 15, 47, …, 58879, 58890, …

实际可储存成：2, 13, 32, …, …, 11, ……（储存差值！）

Thresholding：阈值

① Document：选取相关度较高的文章 → Boolean Queries不灵活

② Query：按频率升序对查询词排序 → 可能遗漏

	Relevant	Irrelevant
Retrieved	R_R	I_R
Not Retrieved	R_N	I_N

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

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

Leftest Tree

1. Definition

① NPL(X): X节点至叶节点的最短距离.

null path length

归定 NPL(NULL) = -1

NPL(Leaf) = 0

$NPL(X) = \min\{NPL(C)+1 \mid C \text{ is a child of } X\}$

② Leftist Tree: 左孩子的NPL永不小于右孩子

左边茂密, 右边稀疏. (右路径最短!)

2. Theorem:

一棵左倾树, 右路径上有r个节点, 则该树至少有2^r-1节点

3. Merge: 将顶大的堆归并到顶小堆的右子树.

Declaration:

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

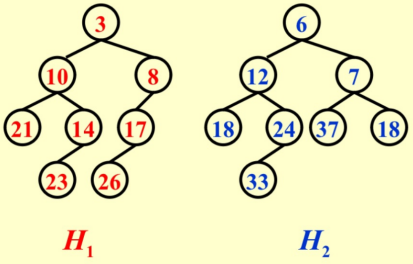
```
PriorityQueue  Merge ( PriorityQueue H1, PriorityQueue H2 )
{
    if ( H1 == NULL )   return H2;
    if ( H2 == NULL )   return H1;
    if ( H1->Element < H2->Element )  return Merge1( H1, H2 );
    else return Merge1( H2, H1 );
}
```

```
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;
}
```

What if *Npl* is NOT updated?

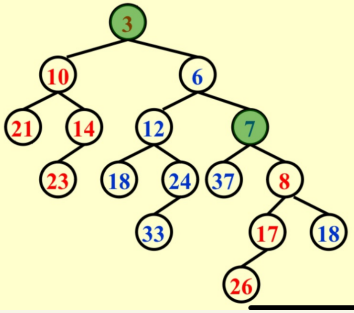
$T_p = O(\log N)$

☞ Merge (iterative version):

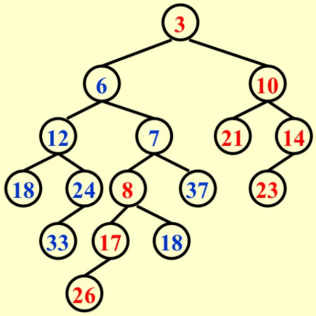


H_1 H_2

Step 1: Sort the right paths without changing their left children



Step 2: Swap children if necessary



☞ DeleteMin:

Step 1: Delete the root

Step 2: Merge the two subtrees

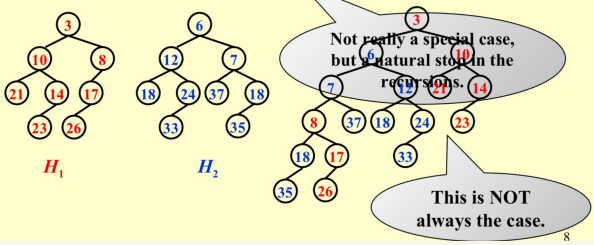
$T_p = O(\log N)$

Skew Heap 斜堆

Target：任意M个连续操作时间复杂度 $O(M \log N)$

Merge：Always swap the left and right children

☞ Merge: 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.

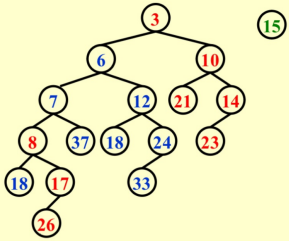


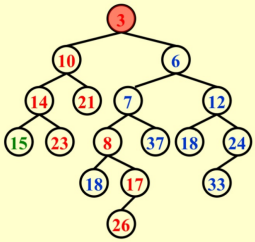
8

好处是，不需要保地 NPL

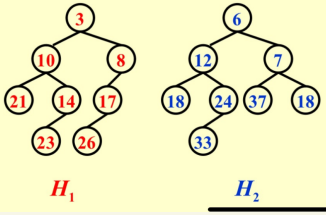
坏处是，右路径长度无法得到保证

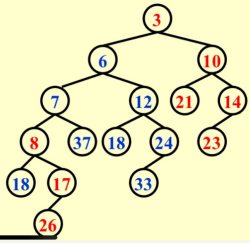
【Example】 Insert 15





☞ Merge (iterative version):





9

向空的斜堆中插入 1到2^k-1 （k>4）个节点，斜堆一定会变成满二叉树！

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

【 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.

右子树比左子树更大

性质：① 只有右路径上的节点性质会发生变化.

② 重节点一定会变成轻节点，
轻节点不一定会变成重节点.

③ 右路径上的轻节点数 $O(\log N)$，而非 $\Theta(\log N)$

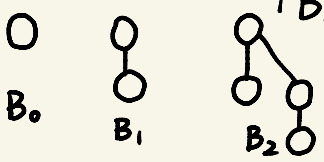
对于一个 Heap 来说：

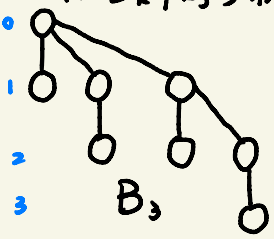
	Insertion	Deletion
Worst Case	$O(\log N)$	$O(\log N)$
Amortized	$O(\log N)$	$O(1)$

Binomial Queue

1. Definition:

① Binomial Tree: { The height of 1 node tree is 0.
B_K 是由一棵 B_{K-1} 的子树接到另一棵 B_{K-1} 的根上.





第d层的节点数: C_K^d
K阶二项树节点数: 2^K

② Binomial Queue

由一系列不同阶的 Binomial Tree 的组成的森林,
且每棵对都符合堆结构. (every parent is bigger/smaller than kids)

2. Operations

① Find Min
1° 直接在一系列根节点中找: $O(\log N)$
2° 专门配一个指向Min的指针: $O(1)$

② Merge: 看作二项式加法

③ Insert: 看作一棵0阶二项树向另一棵树合并.
1°: 插入一个元素, 若最小空二项树为B_i, 则这次插入的 $T_p = const \cdot (i+1)$
2°: N次插入到原始为空的二项队列, 最差情况为O(N). (average time is constant)

④ Delete Min(H)
1° FindMin in B_K $O(\log N)$ 2° Remove B_K from H $O(1)$
3° Remove root from B_K $O(\log N)$ 4° Merge(H', H'') $O(\log N)$

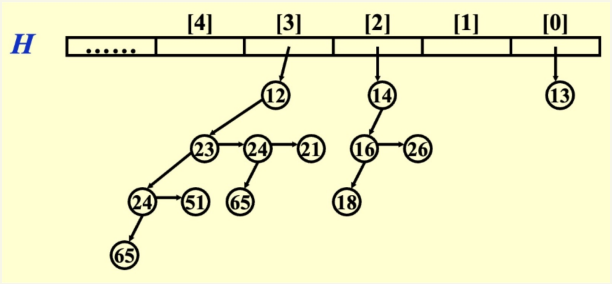
具体实现：

1. 用 sibling 链表 实现每一棵二项树（左孩右兄）

2. 每个节点的孩子按 从大到小(从左向右)排列（方便合并）

指的是 树 从大到小

这是子树降序排列的根本原因




```
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 ];
};
```

```
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;
}
```

把顶点大的接到顶点小的树上。

每个树都是优先队列，也就是说满足 parent 小于 child.

```
BinQueue  Merge( BinQueue H1, BinQueue H2 )
{   BinTree T1, T2, Carry = NULL;
    int i, j;
    if ( H1->CurrentSize + H2-> CurrentSize > Capacity )  ErrorMessage();
    H1->CurrentSize += H2-> CurrentSize;
    for ( i=0, j=1; j<= H1->CurrentSize; i++, j*=2 ) {
       T1 = H1->TheTrees[i]; T2 = H2->TheTrees[i]; /*current trees */
       switch( 4*!!Carry + 2*!!T2 + !!T1 ) {   /* assign each digit to a tree */
            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;
       } /* end switch */
    } /* end for-loop */
    return H1;
}
```

Carry	T2	T1

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

     if ( IsEmpty( H ) )  {  PrintErrorMessage();  return –Infinity; }

     for ( i = 0; i < MaxTrees; i++) {  /* Step 1: find the minimum item */
        if( H->TheTrees[i] && H->TheTrees[i]->Element < MinItem ) {
              MinItem = H->TheTrees[i]->Element;  MinTree = i;    } /* end if */
     } /* end for-i-loop */
     DeletedTree = H->TheTrees[ MinTree ];
     H->TheTrees[ MinTree ] = NULL;   /* Step 2: remove the MinTree from H => H' */
     OldRoot = DeletedTree;   /* Step 3.1: remove the root */
     DeletedTree = DeletedTree->LeftChild;   free(OldRoot);
     DeletedQueue = Initialize();   /* Step 3.2: create H" */
     DeletedQueue->CurrentSize = ( 1<<MinTree ) – 1;  /* 2^MinTree – 1 */
     for ( j = MinTree – 1; j >= 0; j – – ) {
        DeletedQueue->TheTrees[j] = DeletedTree;
        DeletedTree = DeletedTree->NextSibling;
        DeletedQueue->TheTrees[j]->NextSibling = NULL;
     } /* end for-j-loop */
     H->CurrentSize  – = DeletedQueue->CurrentSize + 1;
     H = Merge( H, DeletedQueue ); /* Step 4: merge H' and H" */
     return MinItem;
}
```

每次插入后：

树的数量增加 2−C，其中 C 是插入成本，C=树合并次数+1

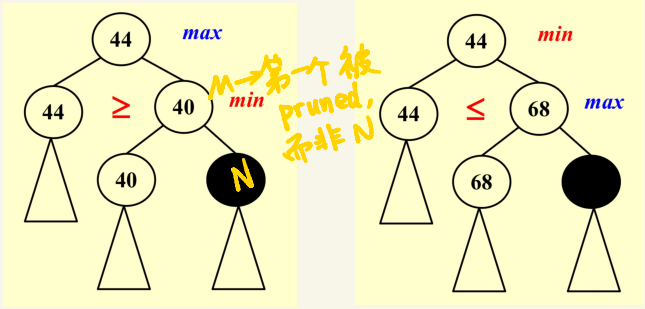
$T_{worst} = O(\log N)$，$T_{amortized} = 2$

Backtracing 回溯法 (不断探索每一种可能，一旦不满足，回退到上一步满足的情况。)

1. 八皇后解决方案 √

2. α、β 剪枝 (prun)

α：剪 min　　β：剪 max




```
bool Backtracking ( int i )
{  Found = false;
   if ( i > N )
      return true; /* solved with (x1, …, xN) */
   for ( each xi ∈ 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;
}
```

$f(P) = W_{computer} - W_{human}$

3. The Turnpike Reconstruction Problem：收费站重建问题

也就是：给出 $\frac{N(N-1)}{2}$ 个"距离"，要求安排N个收费站，要求满足任意两个站点间距离的集合恰好是上述 $\frac{1}{2}N(N-1)$ 个距离。

```
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;
}
```

如果不同的 solution 大小不同，则从小空间的 solution 开始测试能更快地节约时间开销。

4. n个人，m个狼人，1人撒谎。输入：

```cpp
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

// 全局变量定义
int N, M, L; // N: 玩家数量, M: 狼人数量, L: 说谎者数量
vector<bool> isWerewolf; // 标记每个玩家是否是狼人
vector<int> statements,solution, tempSolution; // 玩家的陈述、存储最终解和临时解

// 检查说谎者数量和狼人说谎数量是否满足条件
bool checkLies() {
    int lieCount = 0;
    for (int i = 0; i < N; ++i) {
        int statement = statements[i];
        // 判断玩家陈述的真假
        bool statementTruth = ((statement > 0) == !isWerewolf[abs(statement) - 1]);
        if (!statementTruth) lieCount++;
    }
    int werewolvesLying = 0;
    for (int i = 0; i < N; ++i) {
        // 统计狼人说谎的数量
        if (isWerewolf[i] && (statements[i] < 0 == isWerewolf[abs(statements[i]) - 1])) {
            werewolvesLying++;
        }
    }
    // 返回说谎者数量和狼人说谎数量是否符合条件
    return lieCount == L && werewolvesLying > 0 && werewolvesLying < M;
}
  // 回溯函数
  void backtrack(int index, int werewolves) {
      // 如果已经选出了所需数量的狼人，则进行检查
      if (werewolves == M) {
          if (checkLies()) {
              tempSolution.clear();
              for (int i = 0; i < N; ++i) {
                  // 将选出的狼人加入临时解
                  if (isWerewolf[i]) tempSolution.push_back(i + 1);
              }
              // 更新最终解
              if (tempSolution > solution) {
                  solution = tempSolution;
              }
          }
          return;
      }
      // 如果所有玩家都已经考虑完毕，则返回
      if (index == N) return;

      // 选择当前玩家作为狼人
      isWerewolf[index] = true;
      backtrack(index + 1, werewolves + 1);
      // 不选择当前玩家作为狼人
      isWerewolf[index] = false;
      backtrack(index + 1, werewolves);
  }
  int main() {
      // 输入玩家数量、狼人数量和说谎者数量
      cin >> N >> M >> L;
      statements.resize(N);
      isWerewolf.assign(N, false);

      // 输入每个玩家的陈述
      for (int i = 0; i < N; ++i) {
          cin >> statements[i];
      }

      // 调用回溯函数
      backtrack(0, 0);

      // 输出结果
      if (solution.empty()) {
          cout << "No Solution" << endl;
      } else {
          for (int i = 0; i < M; ++i) {
              cout << solution[M-1-i] << (i < M - 1 ? ' ' : '\n');
          }
      }
      return 0;
  }
```

Sample Input 1:

```
5 2 2
-2   指控2是狼
+3   指控3是人
-4
+5
+4
```

Sample Output 1:

```
4 1   4和1在撒谎
```

Sample Input 2:

```
6 2 3
-2
+3
-4
+5
+4
-3
```

Sample Output 2:

```
6 4
```

Divide & Conquer 分治法

将N规模的问题分成 a 个 $\frac{N}{b}$ 的问题

则 $T(N) = aT(\frac{N}{b}) + f(N)$，其中 $f(N)$ 是合并的时间复杂度

① 代换法：肉眼观察预设结果，代入检验 Substitution Method

② 递归树法：画出递归树，节点上写 $f(N)$，

Rercursion_Tree Method 每个节点有 a 个孩子，每个孩子的函数中 N 值变成父结的 $\frac{1}{b}$

③ 主方法：Master Method

1° (I) $f(N) = O(N^{\log_b a - \varepsilon}) \Rightarrow T(N) = \Theta(N^{\log_b a})$

(II) $f(N) = \Theta(N^{\log_b a}) \Rightarrow T(N) = \Theta(N^{\log_b a} \cdot \log N)$

(III) $f(N) = \Omega(N^{\log_b a + \varepsilon})$，* $N^{\log_b a} \cdot \log N$ 并不满足(III)！只要 $\varepsilon > 0$，$N^{\log_b a + \varepsilon} = \Omega(N^{\log_b a} \cdot \log N)$

且 $\exists C < 1$、$\exists$ 足够大的 N，使得 $af(\frac{N}{b}) < Cf(N)$. $\Rightarrow T(N) = \Theta(f(N))$

↳ regular condition

2° $T(N) = aT(\frac{N}{b}) + \Theta(N^K \log^P N)$ $a \geq 1$, $b \geq 1$, $P \geq 0$

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

* 相比而言：2 更具有普适性，1 更具有精确性.

↳ 排除了1条件下3种情况都不满足的可能

↳ Θ 比 O 更精确

3° (1) $af(\frac{N}{b}) \leq kf(N)$, $k < 1 \Rightarrow T(N) = \Theta(f(N))$

(2) $af(\frac{N}{b}) \leq Kf(N)$, $K > 1 \Rightarrow T(N) = \Theta(N^{\log_b a})$

(3) $af(\frac{N}{b}) \leq f(N) \Rightarrow T(N) = \Theta(f(N) \cdot \log N)$

Eg 1：MergeSort： $a=2$，$b=2$，$f(N)=N$

由主方法1的(II)可知：$f(N)=\Theta(N^{\log_b a})\Rightarrow T(N)=N\log N$

Eg 2：Tree_traversals： a、b 不确定，但均为正，而 $f(N)=O(1)$

$\Rightarrow O(N)$

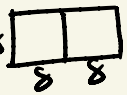
Eg 3：The max subsequence sum： $O(N\log N)$

Eg 4：在一系列点中，找到两点间的最小距离

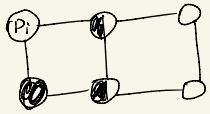
1°画一条线 l，将点等分。（$a=b=2$）$\Rightarrow$ 重点确定 $f(N)$

2°分别计算两侧的 $T(\frac{N}{2})$，取最小值为 δ

3°画出 l 的两条平行线，距离 l 都是 δ。也就是说，对于所有两点在 l 两侧且距离小于 δ 的情况，都被新做的两条平行线包围，记该区域为 S

4°在 S 区域进行从上至下扫描，对于每个点，如果有另一个点与其距离小于 δ，则必在  大小的矩形中。

5°在这种大小的矩形中，设初始点为 P_i，则最多可以有7个点满足 $d<\delta$。



$\Rightarrow f(N)=O(N)$（最坏情况：所有点 $\in S$）

由主方法的方法2的(I)，$k\le 1=\log_b a$

$\Rightarrow T(N)=O(N^{\log_b a}\cdot\log^{k+1}N)=O(N\log N)$

Dynamic Programming 动态规划

Example 1: Fibonacci Number

Solution: Record the two most recently computed values to avoid recursive calls.

```
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;
}
```

$T(N) = O(N)$

3

求最少的计算次数?

Example 2: Ordering Matrix Multiplications.

$M_{1n} = M_1 \cdot M_2 \cdot M_3 \cdot M_4 \cdots M_n \quad (M_{ij} = M_i \cdot M_{i+1} \cdot \cdots \cdot M_j)$

若令 b_n = 计算 $\prod_{i=1}^{n} M_i$ 的方法，则 $b_n = \sum_{i=1}^{n-1} b_i b_{n-i}$ ⇒ 卡特兰数! $b_2 = 1$, $b_3 = 2$

$O\left(\frac{4^n}{n\sqrt{n}}\right)$

记 m_{ij} 为计算 M_{ij} 的最少计算次数.

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

```
/* 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 */
}
```

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

$$\begin{matrix} m_{1,N} & & & & \\ m_{1,N-1} & m_{2,N} & & & \\ \vdots & \vdots & \ddots & & \\ m_{1,2} & m_{2,3} & \cdots & m_{N-1,N} & \\ m_{1,1} & m_{2,2} & \cdots & m_{N-1,N-1} & m_{N,N} \end{matrix}$$

Example 3: Optimal Binary Search Tree: 最优二叉搜索树

每个节点被访问的概率P不同。目标：最小化 $T(N)=\sum_{i=1}^{N} P_i(1+d_i)$

(d_i是深度)

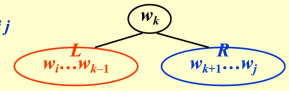
Dynamic Programming

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

c_{ij} ::= cost of T_{ij} ($c_{ii}=0$ if $j<i$)

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

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



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

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

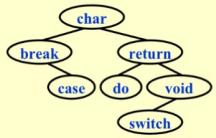
$C_{ij} = \min_{i \le k \le j} \{ w_{ij} + C_{i,k-1} + C_{k+1,j} \}$

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

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.

Example 4: 找到图中两点间的最短距离。

$O(N^3)$ ↑

思1°：利用 Dijkstra 算法，算出一点出发到其它所有点的最短路径。

思2°：$D^k[i][j]$ = 从i点到j点，除i,j外经过至多k个点，需要的最短路径。

($D^{-1}[i][j]$ = i,j连线边的权)

Algorithm Start from D^{-1} and successively generate $D^0, D^1, \ldots, D^{N-1}$. If D^{k-1} is done, then either

① $k \notin$ the shortest path $i \to \{l \le k\} \to j \Rightarrow D^k = D^{k-1}$; or

② $k \in$ the shortest path $i \to \{l \le k\} \to j$

$= \{$ the S.P. from i to $k\} \cup \{$the S.P. from k to $j\}$

$\Rightarrow D^k[i][j] = D^{k-1}[i][k] + D^{k-1}[k][j]$

$\therefore D^k[i][j] = \min\{D^{k-1}[i][j], D^{k-1}[i][k] + D^{k-1}[k][j]\}, k \ge 0$

```
/* A[ ] contains the adjacency matrix with A[ i ][ i ] = 0 */
/* D[ ] contains the values of the shortest path */
/* N is the number of vertices */
/* A negative cycle exists iff D[ i ][ i ] < 0 */
void AllPairs( TwoDimArray A, TwoDimArray D, int N )
{  int  i, j, k;
   for ( i = 0; i < N; i++ )  /* Initialize D */
       for( j = 0; j < N; j++ )
           D[ i ][ j ] = A[ i ][ j ];
   for( k = 0; k < N; k++ )  /* add one vertex k into the path */
       for( i = 0; i < N; i++ )
           for( j = 0; j < N; j++ )
             if( D[ i ][ k ] + D[ k ][ j ] < D[ i ][ j ] )
                   /* Update shortest path */
                   D[ i ][ j ] = D[ i ][ k ] + D[ k ][ j ];
}
```

$T(N) = O(N^3)$, but faster in a *dense* graph.

负边无影响，但负环会有影响。

Example 5: 2条流水线，组装同一样东西。(工艺不同)

流水线间转移需时间。

问怎么组装最快？

```
f[0][0]=0;
f[1][0]=0;
for(stage=1; stage<=n; stage++){
  for(line=0; line<=1; line++){
    f_stay = f[  line][stage-1] + t_process[  line][stage-1];
    f_move = f[1-line][stage-1] + t_transit[1-line][stage-1];
    if (f_stay<f_move){
      f[line][stage] = f_stay;
    }
    else {
      f[line][stage] = f_move;
    }
  }
```

```
line = f[0][n]<f[1][n]?0:1;
for(stage=n; stage>0; stage--){
  plan[stage] = line;
  line = L[line][stage];
}
```

Greedy Algorithm 贪心

Activity Selection Problem

Given a set of activities $S = \{ a_1, a_2, \ldots, a_n \}$ that wish to use a resource (e.g. a classroom). Each a_i takes place during a time interval $[s_i, f_i)$.

Activities a_i and a_j are *compatible* if $s_i \geq f_j$ or $s_j \geq f_i$ (i.e. their time intervals do not overlap).

Select a maximum-size subset of mutually compatible activities.

Assume: $f_1 \leq f_2 \leq \ldots \leq f_{n-1} \leq f_n$

〖Example〗

i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	9	9	10	11	12	14	16

以哪种策略贪心？

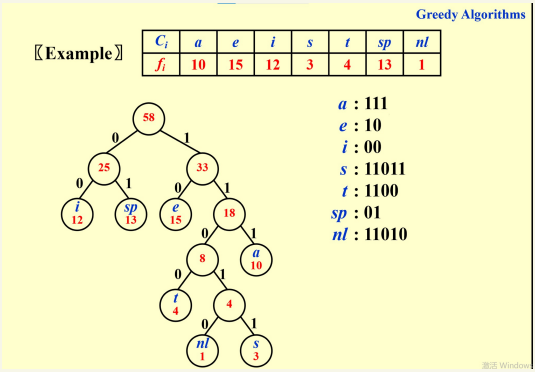
最早结束最佳.

Greedy 使用条件：

① 存在子问题 ② 存在最优解

③ 在做出贪婪选择后，剩下子问题的最优解加上贪婪选择，可以构成原问题的最优解.

Ex2：Huffman Codes 哈夫曼编码



不断找到现有树中 weight 最小的两棵并合并，直至只剩最后一棵树.

（一开始所有 node 各自是一棵树）

➢ **Huffman's Algorithm (1952)**

```
void Huffman ( PriorityQueue  heap[ ],  int  C )
{  consider the C characters as C single node binary trees,
     and initialize them into a min heap;
     for ( i = 1; i < C; i++ ) {
        create a new node;
        /* be greedy here */
        delete root from min heap and attach it to left_child of node;
        delete root from min heap and attach it to right_child of node;
        weight of node = sum of weights of its children;
        /* weight of a tree = sum of the frequencies of its leaves */
        insert node into min heap;
     }
}
```

$T = O(\ C \log C\ )$

NP Completeness

两种图灵机：
- Deterministic（确定型）：准确判断 next unique instruction.（算法确定）
- Nondeterministic（非确定型）：free to choose next step from a finite set, and will always choose the correct one which leads to a solution.

P问题：可以用确定型图灵机在多项式时间内解决。

NP问题：可以用确定型图灵机在多项式时间内验证，
可以用非确定型图灵机在多项式时间内解决。

NPC问题：
- 是一个NP问题。
- 所有NP问题都可以多项式归约为该问题。（所有NPC问题可在多项式时间内互相转换。）

所有NPC难度相同。一旦一个NPC问题被解决（在多项式时间内）：
则全部NPC问题被解决，
且 NP = P = NPC。

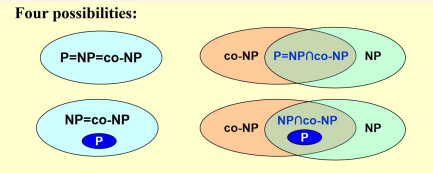
目前无法证明 P是否等于NP。
若 P≠NP：一些问题无法在多项式内被解决。

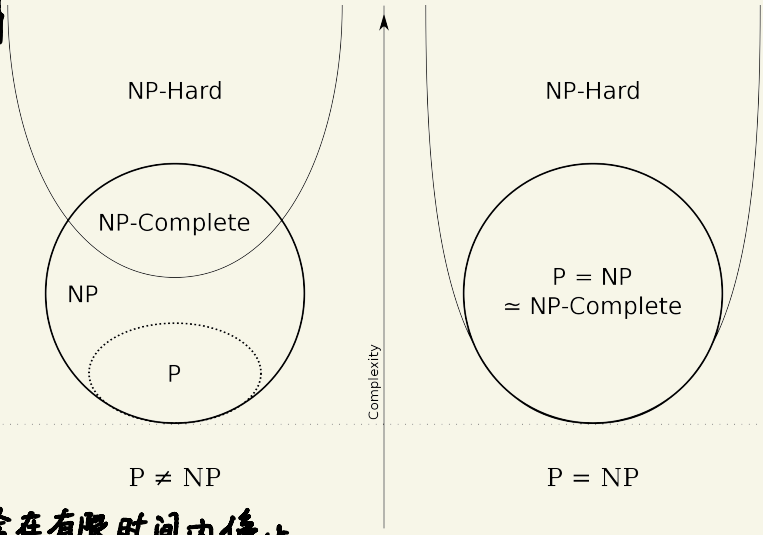
证明是NPC问题：
1. 证明是NP
2. 可由一个已知的NPC问题多项式转化而来。

NPH：所有NPC问题均可转化为NPH问题即可。

NPC = NP ∩ NPH

co-NP: $\{L \mid \bar{L} \in NP\}$





Ex1: Halting Problem

Undecidable Problem (不可判定问题)

↳ 无法设计算法实现. 对任意一个程序, 我们无法设计算法判断它是否会在有限时间内停止.

Ex2: Hamilton Cycle Problem ⇒ NPC

判断图中是否存在一条环路, 经过每个顶点一次.

Ex3: Traveling Salesman Problem

给定边有权重的完全图: NPC 问题

给定整数 K, 判断是否存在经过所有顶点的环路, 使得 cost < K?

(另一版本: 求经过所有顶点的环路的最小 cost?) NPH

Ex4: Circuit-SAT. 最早被证明的 NPC 问题.

具有 n 个布尔变量的布尔表达式, 判断是否具有 True 解.

A 可被多项式归约为问题 B: A 不会比 B 难.

$$A \leq_p B \Leftrightarrow \exists f() \text{ which runs in polynomial time,}$$
$$s.t.\ \forall x \in A,\ f(x) \in B$$
$$\text{and } \forall f(x) \in B,\ y \in A$$

Approximation Algorithm 近似算法

定义近似比：$\forall n,\ \rho = \max\left\{\frac{f(n,x)}{f(n,x^*)}, \frac{f(n,x^*)}{f(n,x)}\right\}$，其中 x^* 为准确结果，x 是算法得

对于 $\varepsilon > 0$，近似算法的近似比为 $1+\varepsilon$. ⇒ 算法复杂度为 $f(n,\varepsilon)$

PTAS：多项式时间近似范式，$f(n,\varepsilon)$ 是关于 n 的多项式

FPTAS：完全多项式时间近似范式，$f(n,\varepsilon)$ 是关于 n 和 $\frac{1}{\varepsilon}$ 的多项式.

Ex1：Bin Packing

给出一系列 $(0,1]$ 之间的小数，问需要至少多少个大小为 1 的盒子？

❖ Next Fit NF

```
void NextFit ( )
{  read item1;
   while ( read item2 ) {
      if ( item2 can be packed in the same bin as item1 )
           place item2 in the bin;
      else
           create a new bin for item2;
      item1 = item2;
   } /* end-while */
}
```

【Theorem】 Let M be the optimal number of bins required to pack a list I of items. Then *next fit* never uses more than $2M-1$ bins. There exist sequences such that *next fit* uses $2M-1$ bins.

↳ 注意这个结论！

❖ First Fit FF

```
void FirstFit ( )
{  while ( read item ) {
      scan for the first bin that is large enough for item;
      if ( found )
           place item in that bin;
      else
           create a new bin for item;
   } /* end-while */
}
```

Can be implemented in $O(N \log N)$

【Theorem】 Let M be the optimal number of bins required to pack a list I of items. Then *first fit* never uses more than $17M/10$ bins. There exist sequences such that *first fit* uses $17(M-1)/10$ bins.

❖ Best Fit BF

Place a new item in the tightest spot among all bins.
$T = O(N \log N)$ and bin no. $\le 1.7M$

以上都是 online 算法.

如果限定使用 online 算法，其最坏情况的结果至少使用准确结果的 $\frac{5}{3}$.

Off-line Algorithm：先按 decreasing 次序排序，然后再用 FF / BF

↳ First Fit Decreasing

【Theorem】 Let M be the optimal number of bins required to pack a list I of items. Then *first fit decreasing* never uses more than $11M/9 + 6/9$ bins. There exist sequences such that *first fit decreasing* uses $11M/9 + 6/9$ bins.

Ex 2 : The Knapsack Problem 背包问题

1 如果可以选 fraction 的物品，那就用 greedy，贪心 $\frac{P_i}{W_i}$（单位利润）最高的

0-1 version：某个物品要么全选，要么全不选。⇒ NPC 问题

思1°：放 P 最大的

思2°：放 $\frac{P_i}{W_i}$ 最大的

The approximation ratio is 2.

Proof: $p_{max} \le P_{opt} \le P_{frac}$

$p_{max} \le P_{greedy}$ ➡ $P_{opt} / P_{greedy} \le 1 + p_{max} / P_{greedy} \le 2$

$P_{opt} \le P_{frac} \le P_{greedy} + p_{max}$

$W_{i,p}$ = the minimum weight of a collection from $\{1, \ldots, i\}$ with total profit being exactly p

① take i: $W_{i,p} = w_i + W_{i-1,p-p_i}$

② skip i: $W_{i,p} = W_{i-1,p}$

③ impossible to get p: $W_{i,p} = \infty$

DP

$$W_{i,p} = \begin{cases} \infty & i = 0 \\ W_{i-1,p} & p_i > p \\ \min\{W_{i-1,p}, w_i + W_{i-1,p-p_i}\} & otherwise \end{cases}$$

$i = 1, \ldots, n;\ p = 1, \ldots, n\,p_{max}$ ➡ $O(n^2 p_{max})$

若 P_{max} 很大，可向上取整数。

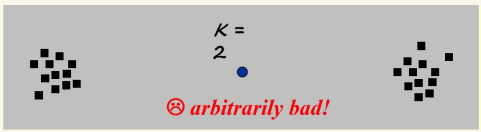
Ex3： K center.

在平面上找 K 个点，以它们为圆心作圆，要求覆盖给出的所有点。要求这 K 个圆的最大半径最小。

思1°：每次选择能让 r 下降最多的点作为 center，选 K 次，直至选齐 K 个点。

Naive Greedy ✗

比如对两团点云的情况：



思2°：2r-greedy：记optimal solution为r_o. 先猜r_x，后验证。

```
Centers  Greedy-2r ( Sites S[ ], int n, int K, double r )
{  Sites  S'[ ] = S[ ]; /* S' is the set of the remaining sites */
   Centers  C[ ] = ∅;
   while ( S'[ ] != ∅ ) {
      Select any s from S' and add it to C;
      Delete all s' from S' that are at dist(s', s) ≤ 2r;
   } /* end-while */
   if ( |C| ≤ K ) return C;
   else ERROR(No set of K centers with covering radius at most r);
}
```

①若搜索成功，则$r_o < 2r_x$

②若搜索失败，（center数量大于K）

则1°：不能说明$r_o > 2r_x$（因为搜索不一定最优）

2°：一定能说明$r_o > r_x$

⇒最终有$r_x < r_o < 2r_x$ → $\rho = 2$

最终给出的近似解是$2r_x$（r_x不满足，不是解）

思3°：Smarter Greedy：

在思2°选新center时，选择离已有center最远的。

ρ还是2.

定理：除非P=NP，否则不存在$\rho < 2$的K-Center算法.

Local Search

梯度下降算法

☞ S ~ S' : S' is a *neighboring solution* of S – S' can be obtained by a small modification of S.

☞ N(S): *neighborhood* of S – the set { S': S ~ S' }.

```
SolutionType Gradient_descent()
{  Start from a feasible solution S ∈ FS ;
   MinCost = cost(S);
   while (1) {
      S' = Search( N(S) ); /* find the best S' in N(S) */
      CurrentCost = cost(S');
      if ( CurrentCost < MinCost ) {
         MinCost = CurrentCost;    S = S';
      }
      else  break;
   }
   return S;
}
```

Ex 1:

❖ **Vertex cover problem:** Given an undirected graph G = (V, E). Find a *minimum* subset S of V such that for each edge (*u*, *v*) in E, either *u* or *v* is in S.

☞ **Search: Start from S = V; delete a node and check if S' is a vertex cover with a smaller cost.**

会有很多情况不合适.

Ex 2:

```
SolutionType Metropolis()
{  Define constants k and T;
   Start from a feasible solution S ∈ FS ;
   MinCost = cost(S);
   while (1) {
      S' = Randomly chosen from N(S);
      CurrentCost = cost(S');
      if ( CurrentCost < MinCost ) {
         MinCost = CurrentCost;    S = S';
      }
      else {
         With a probability e^(−Δcost/(kT)) , let S = S';
         else  break;
      }
   }
   return S;
}
```

Metropolis 算法

Ex 3: Hopfield Neural Network 反馈神经网络.

在一张 G=(V, E) 图中，不同边有不同权重（可正可负）。

〖Definition〗 In a configuration S, edge $e = (u, v)$ is *good* if $w_e s_u s_v < 0$ ($w_e < 0$ iff $s_u = s_v$); otherwise, it is *bad*.

好边的定义

〖Definition〗 In a configuration S, a node u is *satisfied* if the weight of incident good edges ≥ weight of incident bad edges.

$$\sum_{v: e=(u,v)\in E} w_e s_u s_v \le 0$$

坏点的定义

〖Definition〗 A configuration is *stable* if all nodes are satisfied.

稳定图的定义

☞ State-flipping Algorithm

```
ConfigType State_flipping()
{
   Start from an arbitrary configuration S;
   while ( ! IsStable(S) ) {
      u = GetUnsatisfied(S);
      s_u = - s_u;
   }
   return S;
}
```

获得稳定图的算法。一定会终止。

$$\Phi(S) = \Sigma_{e \text{ is good}} |w_e|$$

Claim: The state-flipping algorithm terminates at a stable configuration after *at most* $W = \Sigma_e |w_e|$ iterations.

Claim: Any local maximum in the state-flipping algorithm to maximize Φ is a stable configuration.

使 Φ 最大化的任何局部最大值都是稳定配置

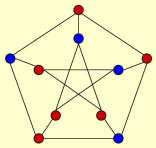
Is it a polynomial time algorithm?

Still an open question: to find an algorithm that constructs stable states in time polynomial in n and $\log W$ (rather than n and W), or in a number of primitive arithmetic operations that is polynomial in n alone, independent of the value of W.

Ex 4. The Maximum Cut Problem

❖ **Maximum Cut problem:** Given an undirected graph G = (V, E) with positive integer edge weights w_e, find a node partition (A, B) such that the total weight of edges crossing the cut is maximized.

$$w(A, B) := \sum_{u\in A, v\in B} w_{uv}$$



- **Toy application**
  - n activities, m people.
  - Each person wants to participate in two of the activities.
  - Schedule each activity in the morning or afternoon to maximize number of people that can enjoy both activities.
- **Real applications** Circuit layout, statistical physics

☞ S ~ S': S' can be obtained from S by moving one node from A to B, or one from B to A.

☝ A special case of Hopfield Neural Network – with w_e all being positive!

```
ConfigType State_flipping()
{
   Start from an arbitrary configuration S;
   while ( ! IsStable(S) ) {
      u = GetUnsatisfied(S);
      s_u = - s_u;
   }
   return S;
}
```

• How good is this local optimum?

Claim: Let (A, B) be a local optimal partition and let (A^*, B^*) be a global optimal partition. Then $w(A, B) \geq \frac{1}{2} w(A^*, B^*)$.

• May NOT in polynomial time

☝ stop the algorithm when there are no "*big enough*" improvements.

Big-improvement-flip: Only choose a node which, when flipped, increases the cut value by at least

$$\frac{2\varepsilon}{|V|} w(A, B)$$

Claim: Upon termination, the big-improvement-flip algorithm returns a cut (A, B) so that

$$(2+\varepsilon)\, w(A, B) \geq w(A^*, B^*)$$

Claim: The big-improvement-flip algorithm terminates after at most $O(n/\varepsilon \log W)$ flips.

• Try a better *local* ?

☞ The neighborhood of a solution should be rich enough that we do not tend to get stuck in bad local optima; but ☞ the neighborhood of a solution should not be too large, since we want to be able to efficiently search the set of neighbors for possible local moves.

Single-flip ➡ k-flip ➡ $\Theta(n^k)$ for searching in neighbors

[Kernighan-Lin 1970] *K-L heuristic*

Step 1: make 1-flip as good as we can – $O(n)$ ➡ (A_1, B_1) and v_1

Step k: make 1-flip of an *unmarked* node as good as we can – $O(n-k+1)$ ➡ (A_k, B_k) and $v_1 \ldots v_k$

Step n: $(A_n, B_n) = (B, A)$

Neighborhood of $(A, B) = \{ (A_1, B_1), \ldots, (A_{n-1}, B_{n-1}) \}$ $O(n$

Randomized Algorithm

Ex1 : Hiring Problem

☞ Hire an office assistant from headhunter

☞ Interview a different applicant per day for N days

☞ Interviewing Cost = C_i << Hiring Cost = C_h

☞ Analyze interview & hiring cost instead of running time

Naïve Solution

```
int Hiring ( EventType C[ ], int N )
{  /* candidate 0 is a least-qualified dummy candidate */
   int Best = 0;
   int BestQ = the quality of candidate 0;
   for ( i=1; i<=N; i++ ) {
      Qi = interview( i ); /* Ci */
      if ( Qi > BestQ ) {
         BestQ = Qi;
         Best = i;
         hire( i );  /* Ch */
      }
   }
   return Best;
}
```

Worst case: The candidates come in increasing quality order

$O(NC_h)$

Radomized Algorithm

```
int RandomizedHiring ( EventType C[ ], int N )
{  /* candidate 0 is a least-qualified dummy candidate */
   int Best = 0;
   int BestQ = the quality of candidate 0;

   randomly permute the list of candidates;

   for ( i=1; i<=N; i++ ) {
      Qi = interview( i ); /* Ci */
      if ( Qi > BestQ ) {
         BestQ = Qi;
         Best = i;
         hire( i );  /* Ch */
      }
   }
```

takes time

no longer need to assume that candidates *are presented* in random order

```
void PermuteBySorting ( ElemType A[ ], int N )
{
   for ( i=1; i<=N; i++ )
      A[i].P = 1 + rand()%(N^3);
      /* makes it more likely that all priorities are unique */
   Sort A, using P as the sort keys;
}
```

Online Hiring Algorithm – hire only once

```
int OnlineHiring ( EventType C[ ], int N, int k)
{
   int Best = N;
   int BestQ = - ∞ ;
   for ( i=1; i<=k; i++ ) {
      Qi = interview( i );
      if ( Qi > BestQ )   BestQ = Qi;
   }
   for ( i=k+1; i<=N; i++ ) {
      Qi = interview( i );
      if ( Qi > BestQ ) {
         Best = i;
         break;
      }
   }
   return Best;
}
```

- What is the probability we hire the best qualified candidate for a given k?
- What is the best value of k to maximize above probability?

先在前k个中找出最好的B，然后从k+1个开始，录用第一个比B好的。

Online Hiring Algorithm

S_i := the ith applicant is the best

What needs to happen for S_i to be TRUE?

{ A:= the best one is at position i } (independent)
∩ { B:= no one at positions $k+1 \sim i-1$ are hired }

$$\Pr[S_i] = \Pr[A \cap B] = \underset{=1/N}{\Pr[A]} \cdot \underset{=k/(i-1)}{\Pr[B]} = \frac{k}{N(i-1)}$$

$$\Pr[S] = \sum_{i=k+1}^{N} \Pr[S_i] = \sum_{i=k+1}^{N} \frac{k}{N(i-1)} = \frac{k}{N} \sum_{i=k}^{N-1} \frac{1}{i}$$

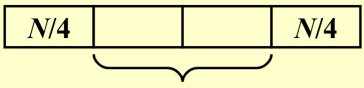
$\Pr(B) = \frac{k}{i-1}$ 是因为：第k+1到i-1个人没被录用 ⇔ 前i-1个人中最好的那个排在前k个中。

Ex 2 : Quicksort $\begin{cases} \Theta(N^2) & \text{worst} \\ \Theta(N \log N) & \text{average} \end{cases}$

Central Splitter : each side contains 至少 $\frac{n}{4}$.

Modified Quicksort : always choose a central splitter as pivot.

Claim: The expected number of iterations needed until we find a central splitter is at most 2.



Pr[*find a central splitter*] = 1/2 ✓

Type j : the subproblem S is of *type j* if $N\left(\frac{3}{4}\right)^{j+1} \le |S| \le N\left(\frac{3}{4}\right)^{j}$

Claim: There are at most $\left(\frac{4}{3}\right)^{j+1}$ subproblems of *type j*.

$$E[T_{type\ j}] = O(N\left(\frac{3}{4}\right)^{j}) \times \left(\frac{4}{3}\right)^{j+1} = O(N)$$

Number of different types = $\log_{4/3} N = O(\log N)$

$\Big\}\ O(N \log N)$

前：势能（难）

AVL（代码）

DP（代码!!!）

后续算法：定义！题目！

（适当放弃）

并行
外部排序 } 简单拿下！

编分：① 输出样例
② 输出0，-1？
③ 用简单贪心代替DP

红黑树：

实在记不清先记定义，

很多答案非红黑树。

势能分析、DP、Greedy（创新、难）

✱DP 背包、子序列（基本模型）

树状：{ 权限：树→链
区间：log

贪心：会很难！X

4类题：
① 没把握
② 没思路
③ 看不懂题干
④ 非常麻烦（几个返回对）