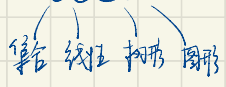


复习要点:

解 疑

① 数据结构三元素: 逻辑结构, 存储结构, 数据运算



② put (r, e): 向量 arr[] = e. (插入)  
 disordered(): return 降序对 4 3 7 + 9 6 : 3

- < deduplicate(): 无序
- < uniquify(): 有序
- < find(): 无序 等于 e 且 rank max 后 => 前, 顺序查找 失败: lo-1
- < search(): 有序 小于 等于 e 且 rank max

**Vector 模板类**

```

template< class T>
class Vector {
public:
    Vector(): capacity = 8, data = T(0) {}
    Vector(int n): capacity = n, data = T(0) {}
    Vector(int n, const T& v): capacity = n, data = T(v) {}
    Vector(const Vector& v): capacity = v.capacity(), data = T(v.data) {}
    ~Vector() {}
    Vector& operator=(const Vector& v) {
        capacity = v.capacity();
        data = T(v.data);
        return *this;
    }
    int capacity() const { return data.capacity(); }
    int size() const { return data.size(); }
    const T& operator[](int i) const { return data[i]; }
    T& operator[](int i) { return data[i]; }
    void push_back(const T& v) { data.push_back(v); }
    void pop_back() { data.pop_back(); }
    void clear() { data.clear(); }
};
    
```

**构造与析构**

```

// 默认构造函数
Vector() { capacity = 8; data = T(0); }
// 指定容量构造函数
Vector(int n) { capacity = n; data = T(0); }
// 指定容量和初始值构造函数
Vector(int n, const T& v) { capacity = n; data = T(v); }
// 拷贝构造函数
Vector(const Vector& v) { capacity = v.capacity(); data = T(v.data); }
// 析构函数
~Vector() {}
// 赋值运算符重载
Vector& operator=(const Vector& v) {
    capacity = v.capacity();
    data = T(v.data);
    return *this;
}
    
```

**构造与析构**

```

// 重载赋值运算符
Vector& operator=(const Vector& v) {
    capacity = v.capacity();
    data = T(v.data);
    return *this;
}
// 重载移动运算符
Vector& operator=(Vector&& v) {
    capacity = v.capacity();
    data = T(v.data);
    return *this;
}
// 重载移动赋值运算符
Vector& operator=(Vector&& v) {
    capacity = v.capacity();
    data = T(v.data);
    return *this;
}
    
```

**可扩充向量**

```

// 扩充操作示意图
// 1. 扩容: 申请新内存, 拷贝旧数据, 释放旧内存.
// 2. 插入: 扩容, 插入元素.
// 3. 删除: 扩容, 删除元素.
// 4. 查找: 扩容, 查找元素.
// 5. 遍历: 扩容, 遍历元素.
    
```

**向量插入**

```

// 插入操作示意图
// 1. 插入到末尾: push_back.
// 2. 插入到指定位置: insert.
// 3. 插入范围: insert_range.
// 4. 插入迭代器: insert_iterator.
    
```

$\Delta -elem[l+1] = -elem[h+1]$   
 $\Delta -size = lo$  此时  $[lo, hi]$  为  $[-size)$   
 lo 已更新  
 不要的 part 删除

**常规向量 (无序) 查找**

```

// 查找操作示意图
// 1. 查找元素: find.
// 2. 查找范围: find_range.
// 3. 查找迭代器: find_iterator.
// 4. 查找失败: return end().
    
```

$f lo-1$   
 $return hi+1$   
 so be like:  $hi--$ , 判断  $lo$  与  $hi+1$   
 之后判断  $e$  和  $-elem[hi]$   
 $lo = hi$  时,  $hi--$ , 返回  $hi+1 (lo)$ , 得 false now  $hi = lo-1$   
 到  $hi$

归并排序  $O(n \log n)$

插入排序  $O(n^2)$

链记排序  $O(n^2)$

选择排序  $O(n^2)$  不稳定

**归并排序 (递归/非递归)**

```

// 归并排序操作示意图
// 1. 递归: 递归分解, 递归合并.
// 2. 非递归: 使用栈模拟递归.
    
```

**向量排序 (归并/插入排序)**

```

// 归并排序操作示意图
// 1. 归并排序: merge_sort.
// 2. 插入排序: insert_sort.
    
```

**向量排序 (起泡排序)**

```

// 起泡排序操作示意图
// 1. 起泡排序: bubble_sort.
    
```

**向量排序 (选择排序)**

```

// 选择排序操作示意图
// 1. 选择排序: selection_sort.
    
```

排序

插入&归并版

插入排序 代码实现

希尔排序 代码实现

归并排序 代码实现

冒泡排序 代码实现

j > 0

$O(n^2 \cdot \log n)$

快速排序 性能分析 (递归)

快速排序 性能分析 (递归)

选择排序 代码实现

堆排序 代码实现

堆排序 算法下逐层遍历 (选择排序进阶)

排序算法比较

快速排序 性能分析 (递归)

快速排序 性能分析 (递归)

散列

散列冲突处理(4)

散列应用: 桶排序

散列应用: 桶排序

散列应用: 基数排序

散列应用: 基数排序

图

广度优先搜索

深度优先搜索

深度优先搜索 (基本版, 递归版, 迭代版)

最小支撑树

最短路径(4)

优先队列在最短路径中的应用

基于优先队列的最小生成树与最短路径

单色值的最短路径问题

单色值的最短路径问题

单色值的最短路径问题

优先队列支持

实现

不少算法 没k-relate 变量 迭代更新, 不要保持

多源最短路径问题

拓扑排序

e log n

重复算法

可拆

可拆

### 有序向量化

唯一化(高效版)

```

int cmp(const void* a, const void* b) {
    return *(int*)a - *(int*)b;
}

int* unique(int nums[], int n) {
    int i, j;
    for (i = 0; i < n; i++) {
        for (j = i + 1; j < n; j++) {
            if (nums[i] == nums[j]) {
                nums[j] = nums[i + 1];
                j--;
            }
        }
    }
    return i + 1;
}
    
```

### 二分查找改进C

二分查找改进2

- 两个指针人为地比较次数，三分变两分
- 多个条件同时满足，避免死循环

```

int search(int arr[], int n, int target) {
    int l = 0, r = n - 1;
    while (l <= r) {
        int m = (l + r) / 2;
        if (arr[m] == target) return m;
        if (arr[m] < target) l = m + 1;
        else r = m - 1;
    }
    return -1;
}
    
```

## ② 列表 哨兵

### 列表的定义2

单向列表、带表头节点方式

```

struct ListNode {
    int data;
    ListNode* next;
};

class List {
public:
    List() { first = new ListNode(); }
    List(int val) { first = new ListNode(val); }
    List(int val, List* list) { first = new ListNode(val, list); }
};
    
```

哨兵: 表头结点位于最前，本身不存储数据。遍历表头结点，使得表头输入与链表输入具有相同的操作，简化实现。

### 查找

无序列表查找

有序列表查找

```

int search(int arr[], int n, int target) {
    int l = 0, r = n - 1;
    while (l <= r) {
        int m = (l + r) / 2;
        if (arr[m] == target) return m;
        if (arr[m] < target) l = m + 1;
        else r = m - 1;
    }
    return -1;
}
    
```

### 无序列表唯一化

```

int* unique(int nums[], int n) {
    int i, j;
    for (i = 0; i < n; i++) {
        for (j = i + 1; j < n; j++) {
            if (nums[i] == nums[j]) {
                nums[j] = nums[i + 1];
                j--;
            }
        }
    }
    return i + 1;
}
    
```

列表删除max: 新插max  
 向末尾max和最后一位swap  
 列表的选择: 稳定

双向列表: header, tailer 哨兵

## ④ 完全二叉树: 除最后一层, rest全是满的, 且最后一层尽可能往左靠

满二叉树

真二叉树: 每个节点孩子为0或2

平衡二叉树: 任意节点左右子树高度差 ≤ 1

$O(h) = O(\log n)$

空树树高 = -1

△ 性质:

i: 二叉树, 深度为 i 节点有  $n_i$  个  $n_0 = n_2 + 1$

总节点 =  $0 + 1 + 1 + 2 + n_2 = n_2 + n_2 + 1 = 2n_2 + 1$

$n_2 = n_0 - 1$

ii: 树节点数 = 总节点 + 1

约瑟夫法: 空塔

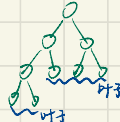
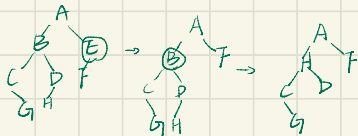
森林到二叉树的转换

- 每棵二叉树
- 二叉树树根 → 看成左孩子 → 长子 → 左弟

前序固定, 可能的中序序列: 卡特兰数

n 节点 = 二叉树形态数目: 卡特兰数 (只看森林)

n 个节点组成 = 二叉搜索树: 卡特兰数



内部节点: 除叶子外全是

0 h=0  
空 h=-1

叶子可能不止最后一层

### 二叉搜索树

- Normal: search, insert, remove
- 平衡: search, insert, remove
- B树: search, insert, remove
- 堆: 构造, insert, delMax

堆: 构造: 需要 insert, 堆合并 等-1, 上滚  
 insert: 插入后 上滚  
 delMax: 置换 + 下滚

卡特兰数:  $C_n = \frac{1}{n+1} \binom{2n}{n}$   
 $lc(n) = in-1$   
 $rc(n) = 0+2$

### 二叉搜索树的删除

**删除**

- 情况1: 被删除节点在左子树且不为空, 可进行以下处理
- 情况2: 在它的右子树中递归寻找比它大的第一个节点(最小), 用它的值替换被删除节点
- 再递归处理该节点的左子树

```

bool removeNode(TreeNode* root, int val) {
    BSTreeNode* temp = nullptr;
    while (temp != nullptr) {
        if (temp->val == val) {
            // 找到要删除的节点
            return true;
        }
        if (temp->val < val) {
            temp = temp->right;
        } else {
            temp = temp->left;
        }
    }
    return false;
}
    
```

### 二叉搜索树的删除

**删除代码实现**

```

TreeNode* deleteNode(TreeNode* root, int key) {
    if (!root) return nullptr;
    if (key < root->val) {
        root->left = deleteNode(root->left, key);
    } else if (key > root->val) {
        root->right = deleteNode(root->right, key);
    } else {
        // 找到要删除的节点
        if (!root->left) return root->right;
        if (!root->right) return root->left;
        // 找到右子树的最小节点
        TreeNode* minNode = root->right;
        while (minNode->left) minNode = minNode->left;
        // 用最小节点替换当前节点
        root->val = minNode->val;
        // 删除右子树中的最小节点
        root->right = deleteNode(root->right, minNode->val);
    }
    return root;
}
    
```

### 二叉搜索树的删除

**迭代实现**

```

TreeNode* deleteNode(TreeNode* root, int key) {
    if (!root) return nullptr;
    BSTreeNode* temp = root;
    while (temp != nullptr) {
        if (temp->val == key) {
            // 找到要删除的节点
            if (!temp->left) return temp->right;
            if (!temp->right) return temp->left;
            // 找到右子树的最小节点
            BSTreeNode* minNode = temp->right;
            while (minNode->left) minNode = minNode->left;
            // 用最小节点替换当前节点
            temp->val = minNode->val;
            // 删除右子树中的最小节点
            temp->right = deleteNode(temp->right, minNode->val);
        } else if (temp->val < key) {
            temp = temp->right;
        } else {
            temp = temp->left;
        }
    }
    return root;
}
    
```

时间复杂度为O(n)

### 二叉搜索树 (教材实现)

**插入的实现**

```

TreeNode* insert(TreeNode* root, int val) {
    if (!root) return new BSTreeNode(val);
    if (val < root->val) {
        root->left = insert(root->left, val);
    } else {
        root->right = insert(root->right, val);
    }
    return root;
}
    
```

### 二叉搜索树的插入实现

**代码实现**

```

TreeNode* insertIntoBST(TreeNode* root, int val) {
    if (!root) return new BSTreeNode(val);
    if (val < root->val) {
        root->left = insertIntoBST(root->left, val);
    } else {
        root->right = insertIntoBST(root->right, val);
    }
    return root;
}
    
```

时间复杂度为O(n), 空间复杂度为O(1)

### 二叉搜索树等价: 中序遍历序列相同

**平衡因子: h<sub>l</sub> - h<sub>r</sub>**

**AVL树的平衡度**

- 规模为n的AVL, height(AVL) = O(logn)
- 规模为n的AVL, AVL的平衡因子不超过±1

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

$h(N) = \log_2(N+1)$

$a < b < c$   
 $T(a, b, c) = T(a, c, b)$

删除, 用其中序直接前驱A代替, 删除原的A  
 → 之后修改平衡因子

### 插入的实现

```

TreeNode* insertIntoBST(TreeNode* root, int val) {
    if (!root) return new BSTreeNode(val);
    if (val < root->val) {
        root->left = insertIntoBST(root->left, val);
    } else {
        root->right = insertIntoBST(root->right, val);
    }
    return root;
}
    
```

### 3+4 重组

**统一最小平衡算法实现**

```

TreeNode* rebalance(TreeNode* root) {
    if (!root) return nullptr;
    if (root->left) root->left = rebalance(root->left);
    if (root->right) root->right = rebalance(root->right);
    // 这里应该写平衡因子判断和旋转逻辑
    return root;
}
    
```

### 3+4 重组

**统一最小平衡算法实现**

```

TreeNode* rebalance(TreeNode* root) {
    if (!root) return nullptr;
    if (root->left) root->left = rebalance(root->left);
    if (root->right) root->right = rebalance(root->right);
    // 这里应该写平衡因子判断和旋转逻辑
    return root;
}
    
```

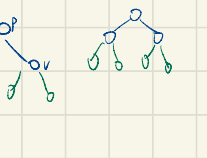
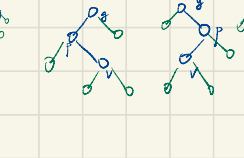
$a < b < c$   
 $T(a, b, c) = T(a, c, b)$

删除, 用其中序直接前驱A代替, 删除原的A  
 → 之后修改平衡因子

### 插入的实现

```

TreeNode* insertIntoBST(TreeNode* root, int val) {
    if (!root) return new BSTreeNode(val);
    if (val < root->val) {
        root->left = insertIntoBST(root->left, val);
    } else {
        root->right = insertIntoBST(root->right, val);
    }
    return root;
}
    
```



### 插入

**插入实现**

```

TreeNode* insertIntoBST(TreeNode* root, int val) {
    if (!root) return new BSTreeNode(val);
    if (val < root->val) {
        root->left = insertIntoBST(root->left, val);
    } else {
        root->right = insertIntoBST(root->right, val);
    }
    return root;
}
    
```

**复杂度分析: O(logn)搜索, 最多O(logn)回溯, 调整次数最多2次, 故总体O(logn)**

### 插入

**插入实现**

```

TreeNode* insertIntoBST(TreeNode* root, int val) {
    if (!root) return new BSTreeNode(val);
    if (val < root->val) {
        root->left = insertIntoBST(root->left, val);
    } else {
        root->right = insertIntoBST(root->right, val);
    }
    return root;
}
    
```

**复杂度分析: O(logn)搜索, 最多O(logn)回溯, 调整次数最多2次, 故总体O(logn)**

### 删除实现

```

TreeNode* deleteNode(TreeNode* root, int key) {
    if (!root) return nullptr;
    if (key < root->val) {
        root->left = deleteNode(root->left, key);
    } else if (key > root->val) {
        root->right = deleteNode(root->right, key);
    } else {
        // 找到要删除的节点
        if (!root->left) return root->right;
        if (!root->right) return root->left;
        // 找到右子树的最小节点
        TreeNode* minNode = root->right;
        while (minNode->left) minNode = minNode->left;
        // 用最小节点替换当前节点
        root->val = minNode->val;
        // 删除右子树中的最小节点
        root->right = deleteNode(root->right, minNode->val);
    }
    return root;
}
    
```

**复杂度分析: 重平衡节点都是x的先驱, 因此最多O(logn)次重平衡, 总体复杂度O(logn)**

### 删除实现

```

TreeNode* deleteNode(TreeNode* root, int key) {
    if (!root) return nullptr;
    if (key < root->val) {
        root->left = deleteNode(root->left, key);
    } else if (key > root->val) {
        root->right = deleteNode(root->right, key);
    } else {
        // 找到要删除的节点
        if (!root->left) return root->right;
        if (!root->right) return root->left;
        // 找到右子树的最小节点
        TreeNode* minNode = root->right;
        while (minNode->left) minNode = minNode->left;
        // 用最小节点替换当前节点
        root->val = minNode->val;
        // 删除右子树中的最小节点
        root->right = deleteNode(root->right, minNode->val);
    }
    return root;
}
    
```

**复杂度分析: 重平衡节点都是x的先驱, 因此最多O(logn)次重平衡, 总体复杂度O(logn)**

### B树 (L1, m) 树

**B树: 多路平衡查找**

**多路平衡查找树**

- m为B树的实际阶数, B-Tree, m路平衡查找树(m<=3)
- 除叶子节点外, 每个非叶子节点的关键字个数不超过m-1, 也不能超过L-1, 且L-2 <= L-1
- 证明: 每个关键字最多占L-1, 其一部分至少占L-1, 本部分至少占L-1, 所以L-1 <= L-1, 可证明

$L = N_1 + N_2 + N_3 + \dots + N_L = L_1$

### B树 (L1, m) 树

**B树: 多路平衡查找**

**多路平衡查找树**

- m为B树的实际阶数, B-Tree, m路平衡查找树(m<=3)
- 除叶子节点外, 每个非叶子节点的关键字个数不超过m-1, 也不能超过L-1, 且L-2 <= L-1
- 证明: 每个关键字最多占L-1, 其一部分至少占L-1, 本部分至少占L-1, 所以L-1 <= L-1, 可证明

$L = N_1 + N_2 + N_3 + \dots + N_L = L_1$

### B-Tree: 关键字查找

**关键字查找**

```

TreeNode* search(TreeNode* root, int key) {
    if (!root) return nullptr;
    for (int i = 0; i < root->keys.size(); i++) {
        if (key < root->keys[i]) {
            return search(root->children[i], key);
        }
    }
    return root;
}
    
```

**复杂度: 每次访问的键值(N)为外键, 所以总体的log(m, N)复杂度为256至1024改进后的效率在一个数量级左右**

### B-Tree: 关键字插入

**关键字插入**

- 首先调用search在树中查找关键字, 若查找成功返回false
- 否则, 查找过程终止于叶子节点的某个槽位, 其父节点为叶子节点, leaf
- 调用leaf的search接口, 在leaf节点查找合适的插入位置
- 将关键字插入该位置+1
- 判断父节点上是否已满(每个槽位大小m-1), 由leafOverFlow函数进行处理

**复杂度: 每次访问的键值(N)为外键, 所以总体的log(m, N)复杂度为256至1024改进后的效率在一个数量级左右**

### B树 (L1, m) 树

**B树: 多路平衡查找**

**多路平衡查找树**

- m为B树的实际阶数, B-Tree, m路平衡查找树(m<=3)
- 除叶子节点外, 每个非叶子节点的关键字个数不超过m-1, 也不能超过L-1, 且L-2 <= L-1
- 证明: 每个关键字最多占L-1, 其一部分至少占L-1, 本部分至少占L-1, 所以L-1 <= L-1, 可证明

$L = N_1 + N_2 + N_3 + \dots + N_L = L_1$

### B树 (L1, m) 树

**B树: 多路平衡查找**

**多路平衡查找树**

- m为B树的实际阶数, B-Tree, m路平衡查找树(m<=3)
- 除叶子节点外, 每个非叶子节点的关键字个数不超过m-1, 也不能超过L-1, 且L-2 <= L-1
- 证明: 每个关键字最多占L-1, 其一部分至少占L-1, 本部分至少占L-1, 所以L-1 <= L-1, 可证明

$L = N_1 + N_2 + N_3 + \dots + N_L = L_1$

### B-Tree: 关键字插入

**关键字插入**

- 首先调用search在树中查找关键字, 若查找成功返回false
- 否则, 查找过程终止于叶子节点的某个槽位, 其父节点为叶子节点, leaf
- 调用leaf的search接口, 在leaf节点查找合适的插入位置
- 将关键字插入该位置+1
- 判断父节点上是否已满(每个槽位大小m-1), 由leafOverFlow函数进行处理

**复杂度: 每次访问的键值(N)为外键, 所以总体的log(m, N)复杂度为256至1024改进后的效率在一个数量级左右**

### B-Tree: 关键字插入

**关键字插入**

- 首先调用search在树中查找关键字, 若查找成功返回false
- 否则, 查找过程终止于叶子节点的某个槽位, 其父节点为叶子节点, leaf
- 调用leaf的search接口, 在leaf节点查找合适的插入位置
- 将关键字插入该位置+1
- 判断父节点上是否已满(每个槽位大小m-1), 由leafOverFlow函数进行处理

**复杂度: 每次访问的键值(N)为外键, 所以总体的log(m, N)复杂度为256至1024改进后的效率在一个数量级左右**

# 二叉树进阶

**B-树：关键字删除**

★

关键字删除：在B-树中删除关键字，需要找到该关键字所在的叶子节点，并将其删除。如果该叶子节点是根节点的直接子节点，则可以直接删除。否则，需要将该叶子节点与其兄弟节点合并，然后将合并后的新叶子节点插入到父节点中。

**B-树：关键字删除**

下溢与“合并-合并”：在删除关键字时，可能会出现下溢（即叶子节点中的关键字数量少于B-树的阶数）。此时，需要将下溢的叶子节点与其兄弟节点合并，然后将合并后的新叶子节点插入到父节点中。

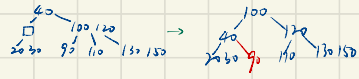
**B-树：关键字删除**

下溢与“合并-合并”：在删除关键字时，可能会出现下溢。此时，需要将下溢的叶子节点与其兄弟节点合并，然后将合并后的新叶子节点插入到父节点中。

**堆排序**

堆排序的实现：通过反复将堆顶元素与堆尾元素交换，然后将新的堆顶元素重新堆化，直到整个数组有序。

2-d 树



**2d-树构造实现**

1. 选择中位数作为根节点，递归地在左右子树中构造2d-树。

**2d-树范围查询实现**

1. 递归遍历2d-树，根据查询范围剪枝，返回符合条件的点。

**2d-树最近邻查询实现**

1. 递归遍历2d-树，维护当前最近邻点，返回最近邻点。

随机更新

暴力(逐行输入):  $O(n^2)$   
堆排序法:  $O(n \log n)$

堆

**二叉堆**

插入：上述的代码实现

分析和优化：比较(交换)  $\rightarrow$  optimize it  $\rightarrow$  不是 data[index]

重新再赋 value 值

childTree 是可能  
被 value 初始  
place.  $\rightarrow$  这些  
value 的  
赋值

**二叉堆**

删除最大元素：置换+下溢

删除最大元素：下溢的代码实现

**堆构建**

堆构建的代码实现

**应用：堆排序**

堆排序实现

```

heapSort(int[] arr) {
    makeHeap(arr);
    while (arr.length > 0) {
        swap(arr[0], arr[arr.length-1]);
        heapSort(arr);
    }
}
    
```

堆操作：总结

插入即上滤，  
删除置换下，  
构建自底下，  
排序始构建，  
迭代做删除。

## Master Theorem

比如  $T(n) = aT(n/b) + O(f(n))$

- ① 严格大:  $T(n) = O(n^{\log_b a})$
- ②  $f(n) = \Theta(n^{\log_b a} \cdot \log n)$ ,  $T(n) = O(n^{\log_b a} \cdot \log^2 n)$  如:  $T(n) = 7T(n/2) + O(1) \Rightarrow O(\log^2 n)$
- ③ 严格小:  $T(n) = O(f(n))$

### 新的构造

$j=0$ 时  $P[0]=P[0]$   $\dots$   $P[j-1]=P[j-1]$   
 转移码表  $O(N^2)$

```

    模式串分析-Next表
    P: [a b a b a b a b]
    1 2 3 4 5 6 7 8

    if %s == %s:
        next[j] = next[j-1] + 1
    else if %s == %s:
        next[j] = next[j-1]
    else if %s == %s:
        next[j] = 0
    else if %s == %s:
        next[j] = 0
    
```

因为  $P[next[j]]$  和  $P[j]$  中长度为  $j$  的真前缀 应与长度为  $j$  的真后缀完全匹配, 故  $i$  来自集合:  
 $N(P, j) = \{0 \leq i < j | P(0..i) = P(j-i..j)\}$

### KMP算法

发现: KMP算法: D.E.Knuth, V.R.Pratt与J.H.Morris同时  
 构造next表

```

    int match (char* a, char* b, int m, int n) {
        int i = 0, j = 0;
        while (i < m && j < n && a[i] == b[j]) {
            ++i; ++j;
        }
        return i;
    }

    void buildNext (char* P, int* next) {
        int m = strlen(P);
        next[0] = 0;
        for (int i = 1; i < m; ++i) {
            int j = next[i-1];
            while (j > 0 && P[i] != P[j]) j = next[j-1];
            if (P[i] == P[j]) ++j;
            next[i] = j;
        }
    }
    
```

这里特殊定义可出现  $next[j]=j-1$ , 并进行特殊处理,  $j=-1$  是什么意思呢?

### 进一步改进

引入失败的经验教训: 递归的不能使得  $P(i)=P(j)$   
 $N(P, j) = \{0 \leq i < j | P(0..i) = P(j-i..j), \cup P(i) = P(j)\}$   
 $next[j] = \max(N(P, j))$

原	1	a	b	1	2	3	4	a	b	1	2	3	4	5	6	7	8
Next	0	a	b	1	0	0	0	a	b	1	0	0	a	b	1	0	0

改进的	1	a	b	1	2	3	4	a	b	1	2	3	4	5	6	7	8
Next	0	a	b	1	0	0	0	a	b	1	0	0	a	b	1	0	0
表	0	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1

$P[next[j]] = P[j]$   
 $\Rightarrow$  递归... 继续找  
 找不到, 试试  $j-1$   
 $j-1 \Rightarrow -1$

### KMP算法

Next表的计算  
 $O(N+M)$

```

    int buildNext (char* P) {
        int m = strlen(P);
        int* next = new int[m];
        next[0] = 0;
        for (int i = 1; i < m; ++i) {
            int j = next[i-1];
            while (j > 0 && P[i] != P[j]) j = next[j-1];
            if (P[i] == P[j]) ++j;
            next[i] = j;
        }
        return next;
    }
    
```

$\Rightarrow O(N+M)$

### 栈的应用: 括号匹配

**策略: 后开先闭!**

表达式	匹配?
(A + B)	Yes
((A + B) * (C + D))	Yes
((x + y) * z	No
[2 * 3] + 4	No
{a = z}	No
5 + 6) / 4	No
{[]}	Yes
{()}	No

### 栈的应用: 表达式求值

表达式求值  
 建立包括括号在内的各种符号间的优先关系比较, 简化程序

### 栈混洗

入栈出栈可行操作序列分析——卡特兰数问题  
 条件: 操作序列  $2n$  位的任意前  $m$  位, 入栈数不比出栈数少  
 相同的组合数学问题:  
 买票找零:  $2n$  个人排队一行进入剧场, 入场费5元, 其中只有  $n$  个人有一张5元钞票, 另  $n$  个人只有10元钞票, 剧院无其它钞票, 问有多少种排队方法使得找零总是足够的? (得将5元者到达找零将5元入栈, 持10元者到达找零使栈中无5元)  
 十二角路经规则: 律师在住所以北  $n$  个街区 and 以东  $n$  个街区工作, 每天走  $2n$  个街区去上班, 如果他从不穿越 (但可以碰到) 从家到办公室的对角线, 有多少条可能路径?  
 带证书的门票, 多边形分割三角形问题

### 栈混洗

判断可行混洗序列  
 给定置换序列  $\sigma = 4$  下的  $(2, 4, 1, 3)$ , 如何判断该序列是否否是混洗序列?