Chapter 1

Introduction

- **证明算法的正确性**
  - **单调性：**
  - **不变性：**
- **stirling逼近公式：书p59**
- **TRICKS：**
  - **典型递归递推式：**
    - **T(n) = 2T(n/2) + O(n) => T(n) = O(nlogn)**

Chapter 2

Vector

- **扩容**
  - **递增策略**
    - **累计增容时间：O(n^2)**
    - **分摊增容时间：O(n)**
    - **装填因子 ≈ 100%**
  - **倍增策略**
    - **累计增容时间：O(n)**
    - **分摊增容时间：O(1)**
    - **装填因子 > 50%**
- **置乱（Permute）**
  - **反复调用permute()算法，可以生成向量V[0, n)的所有n！种排列**
  - **各元素处于任一位置的概率均为1/n**
  - **生成各排列的概率均为1/n!**
- **成功查找长度 & 失败查找长度**

- 成功查找长度：查找成功时，需要进行的关键码比较次数的均值
- 失败查找长度：查找失败时，需要进行的关键码比较次数的均值

- **二分查找（版本A）**

  > **CLICK - Binary Search（Ver.A）**

  > **精通算法以及成功/失败查找长度计算**

  > **TRICK：“左闭右开” => [lo, hi) => [lo, mi) + (mi, hi)**

  - **注意外层循环判断条件为：lo < hi**
  - **注意查找区间为[lo, hi)，hi为哨兵**
  - **“先左”：e < A[mi]**
    - **成立则深入前半段[lo, mi)查找**
  - **“后右”：A[mi] < e**
    - **成立则深入后半段(mi, hi)查找（相当于[mi + 1, hi)）**
  - **“最后”：无需判断**
  - **O(1.5logn)**
  - **最好：O(1)**
  - **最差：O(logn)**
  - **二分查找（版本A）在失败情况下的平均比较次数不超过1.5 · log2(n + 1) = O(1.5logn)**

```
//二分查找（版本A）
while(lo < hi){
    Rank mi = (lo + hi) >> 1; //以中点为轴点
    if(e < A[mi]) hi = mi; //深入前半段[lo, mi)
    else if(A[mi] < e) lo = mi + 1; //深入后半段(mi, hi)查找，即[mi + 1, hi)
    else return mi; //注意此处没有判断
}
return -1;
```

- **Fibonacci查找**

  > **CLICK - Fibonacci Search**

  > **Fibonacci查找在算法上和二分查找（版本A）基本一样，不同的是切分点的选择而已**

  - **除了切分点的选择，其余细节与二分查找（版本A）一致**
  - **向量长度为n = fib(k) - 1，则**

        - **轴点位置：mi = fib(k - 1) - 1**
        - **轴点前向量长度为：fib(k - 1) - 1**
        - **轴点后向量长度为：fib(k - 2) - 1**
    - **O(1.44logn)，在算法结构不改变的前提下，已经达到最优了**
    - **最好：O(1)**
    - **最差：O(logn)**
    - **常用fib数列值：**

      > **记前两个**

        - **fib(0) = 0**
        - **fib(1) = 1**
        - **fib(2) = 1**
        - **fib(3) = 2**
        - **fib(4) = 3**
        - **fib(5) = 5**
        - **fib(6) = 8**
        - **fib(7) = 13**
        - **fib(8) = 21**
    - **在对Fib(n) - 1长度的向量做Fibonacci查找，比较操作的次数最多为n-1**
    - **Fibonacci查找在失败情况下的平均比较次数不超过λ · log2(n + 1) = O(λlogn)**

```
//Fibonacci查找
Fib fib(hi - lo); //构建Fib数列
while(lo < hi){
    while(hi - lo < fib.get()) fib.prev(); //从后向前查找（分摊O(1)）
    Rank mi = lo + fib.get() - 1; //确定形如Fib(k) - 1的轴点
    if(e < A[mi]) hi = mi; //深入前半段[lo, mi)
    else if(A[mi] < e) lo = mi + 1; //深入后半段(mi, hi)查找，即[mi + 1, hi)
    else return mi;
}
return -1;
```

- **二分查找（版本B）**

  > **CLICK - Binary Search（Ver.B）**

> **书p55下方图**

- **最好：O(logn)**
- **最差：O(logn)**

```
while(1 < hi - lo){ //注意外层循环的判断条件的变化
    Rank mi = (lo + hi) >> 1; //中点为轴点
    (e < A[mi]) ? hi = mi : lo = mi; //深入[lo, mi)或[mi, hi)，注意后半段区间
}
return (e == A[lo]) ? lo : -1; //查找成功返回对应的秩，否则统一返回-1
```

- **二分查找（版本C）**

> **CLICK - Binary Search（Ver.C）**

> **前三个算法均不能在查找失败时返回符合语义的信息（不大于元素e的最大元素的秩）**

- **版本C和版本B的主体结构基本一致，主要是为了符合语义而作细微改进**

```
while(lo < hi){
    Rank mi = (lo + hi) >> 1;
    (e < A[mi]) ? hi = mi : lo = mi + 1; //深入[lo, mi)或(mi, hi)，注意后半段区间
}
return --lo;
```

- **在数值独立均匀分布的条件下，二分各版本的平均成功和失败查找长度的关系为：(S + 1) · n = F · (n + 1)**
- **插值查找（Interpolation Search）**

> **元素均匀且独立随机分布，习题解析[2-24]**

- **（mi - lo）/（hi - lo） = （e - A[lo]）/（A[hi] - A[lo]）**
- **每经过一次比较，查找范围n缩小至√n**
- **期望（平均）时间复杂度O(loglogn)**

- **气泡排序（Bubble Sort）**

> **CLICK - Bubble Sort**

- **改进一：设置sort标志**
  - **当某一趟循环没有任何元素交换时，sort=true，提前结束算法**
- **改进二：设置last标志**

> **在某趟循环中，可能很长一段后缀都没有交换发生，这说明这段后缀其实已经排序就位，所以在下一趟循环时，将hi设置为last，从lo~last这段区间进行排序即可**

  - **每一次交换令last=lo，每趟循环结束后将hi=last**

- **归并排序（Merge Sort）**

  > **CLICK - Merge Sort**

  -

```
//向量归并排序
template <typename T>
void Vector<T>::mergeSort(Rank lo, Rank hi){
    if(hi - lo < 2) return; //单区间元素自然有序，否则...
    int mi = (lo + hi) / 2; //以中点为界
    mergeSort(lo, mi); mergeSort(mi, hi); //分别排序
    merge(lo, mi, hi); //归并
}

//有序向量归并
template <typename T>
void Vector<T>::merge(Rank lo, Rank mi, Rank hi){
    T* A = _elem + lo;
}
```

Chapter 3

> List

```
//注意List的插入算法实现的语义，不太常规
```

- **选择排序**

  > **CLICK - Selection Sort**

- **插入排序**

  > **CLICK - Insert Sort**

  > **逆序对（Inversion）定义：i(p)，其中p为某一元素的秩，i(p)表示“以p为右值”的逆序对（即p的逆序对数为p之前[0, p)中的所有元素与p形成的逆序对之和）**

  > **简单来说，这里的逆序对定义是针对某一个元素p而言的，在p的前驱序列中寻找能与之形成逆序对的元素，其总个数即为p的逆序对个数**

一个列表中逆序对总数为：I = Σi(p)

- **O(I + n)**

Chapter 4

Stack & Queue

- **逆序输出**
  - **进制转换**
- **递归嵌套**
  - **括号匹配**
    - **遇左括号入栈，遇右括号弹出栈顶左括号。若扫描完毕栈空，则说明匹配**
  - **栈混洗**
    - **长为n的序列的栈混洗总数为：SP(n) = catalan(n) = (2n)! / (n+1)!n!**
    - **任意三个元素能否按某相对次序出现于混洗中，与其它元素无关**
    - **对于任何1 <= i < j < k <= n，[..., k, ..., i, ..., j, ...>必非栈混洗**
    - **对于任何1 <= i < j < n，[..., j + 1, ..., i, ..., j, ...>必非栈混洗**
    - **n对括号构成的所有表达式就是n个元素的栈混洗总数SP(n)**
- **延迟缓冲**
  - **evaluation**
    - **当Prio(栈顶符号) > Prio(当前扫描符号)：取出栈顶符号与操作数计算，扫描指针不后移**
    - **当Prio(栈顶符号) < Prio(当前扫描符号)：将当前扫描符号压入栈，扫描指针后移**
    - **当Prio(栈顶符号) = Prio(当前扫描符号)：弹出栈顶括号，丢弃当前扫描符号，扫描指针后移**
- **栈式计算**
  - **RPN**
    - **中缀表达式 => RPN：操作符的先后次序可能改变，但操作数的先后次序保持不变**

Chapter 5

> Binary Tree

- **基本概念**
  - **树是连通无环图**
  - **degree(r)为r的（出）度**
  - **树的边数 e = Σdegree(r) = n - 1 = Θ(n)，即树的边数等于所有节点的度之和，也等于节点总数减1**
  - **depth(v) + height(v) <= height(T)**
  - **仅有一个节点的树高度为0，空树高度为-1**
- **多叉树 => 二叉树**

  > **“长子 + 兄弟”：从根节点开始依据原多叉树，找该节点的长子与兄弟，长子为左，兄弟为右，依次进行直到转换完毕**
- **二叉树**

  > **每个节点的度数degree(r) <= 2**
  - **深度为k的节点，至多2^k个**
  - **含n个节点、高度为h的二叉树中，h < n < 2^(h + 1)**
  - **Q：考查任何一棵高度为h的二叉树T，设其中深度为k的叶节点有nk个，0 <= k <= h，则：**
    - **Σ(nk / 2^k) <= 1**
    - **真二叉树时取等号**
  - **先序遍历**

    > **V-L-R**
    - **递归式**
      - **O(n)**
    - **迭代式**
      - **左侧链**
  - **中序遍历**

    > **L-V-R**
    - **递归式**

        - 迭代式
    - 后序遍历（W-I）
        > L-R-V
        - 递归式
        - 迭代式
    - 层次遍历（广度优先遍历）
        > 自上而下，先左后右
        - Q1：考查层次遍历算法，设二叉树共含n个节点
            - 只要辅助队列Q的容量不低于⌈n/2⌉，就不致于出现中途溢出的问题
            - 完全二叉树层次遍历所需辅助队列Q的空间最大
        - Q2：考查对规模为n的完全二叉树（含满二叉树）的层次遍历
            - 在整个遍历过程中，辅助队列的规模变化是单峰对称的，即：
                - {0，1，2，…，(n + 1)/2，…，2，1，0}（n为奇数时）
                - {0，1，2，…，n/2，n/2，…，2，1，0}（n为偶数时）
            - 非完全二叉树的层次遍历，辅助队列规模不可能达到⌈n/2⌉，不具有上述性质
    - 重构
        - ---
        - 后序 + 层次可以确定先序
        - 先序 + 层次不能确定后序
        - [先序 | 后序] + 中序：唯一确定一颗二叉树的拓扑结构
        - [先序 + 后序] x 真二叉树：唯一确定一颗二叉树的拓扑结构
        - 当二叉树中某个节点仅仅只有一个孩子节点的时候，就无法根据其先序和后序唯一的确定一个二叉树（这个孩子是左孩子还是右孩子无法得知），换句话说，也就是无法确定单孩子的左右，其余均可确定；反之，若已知二叉树中某个节点有两个孩子或没有孩子（即度为0或2，即真二叉树），则可以唯一确定
- 真二叉树
    > 不含一度节点的二叉树称作真二叉树
- 完全二叉树
    - ---

    - **节点规模介于 2^h <= n <= 2^(h + 1) - 1 之间**
    - **h = O(logn)**
- **满二叉树**

  > **每个节点的度数degree(r) = 2**

    - **节点数：2^(h + 1) - 1**
    - **叶节点数：2^h**
    - **内部节点数：2^h - 1**
- **PFC（Prefix-free Code）**

  > **前缀无歧义编码**
- **OET（Optimal Encoding Tree）**

  > **最优编码树**

    - **必为真二叉树（双子性）**
- **Huffman**
    - **Q1：任何CBA式Huffman树构造算法，在最坏情况下都需要运行Ω(nlogn)**
    - **Q2：字符权重已排序时，可在线性时间内构造出Huffman树**
    - **每次取最小权值两个节点合成子树**
    - **权值越小的节点（字符出现频率越低）深度越大，权值越大的节点（字符出现频率越高）深度越小**
    - **n个叶节点，则节点总数为2n - 1**
- **TRICKS：**
    - **PFC的权值是指叶节点的深度，Huffman的权值是指叶节点所代表字符的出现频率**

Chapter 6

> Graph

- **基本概念**
    - **邻接：描述两个顶点间的关系**
    - **关联：描述顶点和边之间的关系**
    - **路径（Path）：**

- 简单路径（Simple Path）：
- 有向无环图（DAG）
- 欧拉环路：
- 哈密尔顿环路：
- 平面图：嵌入平面（无交叉边）
  - 平面图必满足e = O(n)
  - 欧拉公式：n - e + f - c = 1
- 无向图的边数等于各顶点度数之和的一半

- 邻接矩阵

> n行n列，描述顶点与顶点之间的关系

  - Q1：即便计入向量扩容所需的时间，就分摊意义而言，图插入节点GraphMatrix::insert(v)算法的时间复杂度依然不超过O(n)
    - 当然，在最坏情况下，可能需要O(n^2)来完成节点插入操作

- 关联矩阵

> n行e列，描述顶点与边之间的关系

  -

- 邻接表
  - 空间复杂度O(n + e)
- Vertex
  - Status
    - UNDISCOVERED
    - DISCOVERED
    - VISITED
  - dTime：discover time
  - fTime：finish time
  - parent
  - priority

- **Edge**
  - **Status**
    - **UNDETERMINED**
    - **TREE**
    - **CROSS**
    - **FORWARD**
    - **BACKWARD**
  - **weight**
- **广度优先搜索（BFS）**

  > **breadth-first search**

  - **初始顶点处于UNDISCOVERED，边处于UNDETERMINED，队列中为起始节点（DISCOVERED）**
  - **从队列里取出一个节点，遍历访问它所有的邻居，并对每个邻居：**
    - **若邻居节点的状态为UNDISCOVERED**
      - **将邻居节点设为DISCOVERED状态，然后将邻居节点推入队列**
      - **将当前访问节点与邻居节点之间的边设为TREE**
    - **若邻居节点的状态为DISCOVERED或者VISITED**
      - **将当前访问节点与邻居节点之间的边设为CROSS**
  - **将当前节点的状态设为VISITED**
  - **O(n + e)**
  - **Q1：辅助队列中的节点始终按照其深度单调排列，且任何时刻同处于辅助队列中的顶点，深度彼此相差不超过一个单位**
  - **Q2：所有顶点按其在BFS树中的深度，以非降次序接受访问**
  - **Q3：所有顶点按其到s的距离，以非降次序接受访问**

  > **Q1、Q2、Q3对无向图与有向图均成立**

  - **Q4：**

```
//BFS
```

- **深度优先搜索（DFS）**

  > **depth-first search**

  - **嵌套引理**
    - **活跃期**
    - **顶点v是u的祖先，当且仅当v的活跃期完全包含u的活跃期（子集关系）**
    - **顶点v与u无承袭关系，当且仅当v的活跃期与u的活跃期无交集（交集为空）**
  - **O(n + e)**
  - **TRICKS：**
    - **TREE边的数量总是等于顶点数减去连通分量的数量**
    - **DFS中出现BACKWARD边说明有回路**

```
//DFS
```

- **n个互异顶点组成的图，共有n^(n-2)棵生成树**
- **Prim**

  > **最小生成树，无根的概念**

  - **每次迭代寻找uk到Tk间的最短割边**
  - **Q1：考查某些边的权重不是正数的带权网络**
    - **Prim算法依然可行**
  - **Q2：最小生成树唯一性**
    - **权重互异，最小生成树唯一**
    - **权重未必互异，最小生成树未必唯一**
  - **Q3：在允许多边等权的图中，完全由极短跨越边构成的生成树，未必是最小的**
- **Dijkstra**

  > **最短路径树，有根的概念**

  - **每次迭代寻找uk到s间的最短路径**
  - **Q1：考查某些边的权重不是正数的带权网络**
    - **即便不含负权重环路，Dijkstra未必可行**

- **Q2：最短路径树唯一性**
  - **即便权重互异，最短路径树也未必唯一**
  - **权重未必互异，最短路径树未必唯一**

- **最短路径 & 最短路径树 & 最小生成树**
  - **最短路径：从一个指定的顶点出发，计算从该顶点出发到其他所有顶点的最短路径**
  - **最短路径树SPT（Short Path Tree）：是网络的源点到所有结点的最短路径构成的树**
    - **分别找出源点到每一个节点的最短路径，然后把路径合并起来就成为最短路径树**
  - **最小生成树MST（Minimum Spanning Tree）：没有根的概念，从某一节点出发以最小的边权重总和覆盖所有点**

- **习题解析总结**
  - **Q1：考查无向图**

    > **对于图中任一顶点v，其余顶点到它的距离的最大值，称作其偏心率**
    >
    > **图中偏心率最小的顶点称作中心点，并将这个最小偏心率称作图的半径**
    >
    > **图中偏心率最大的顶点称作边缘点，并将这个最大偏心率称作图的直径**

    - **考查图的特例：树T**
      - **T的中心点或者唯一，或者是一对相邻顶点**
      - **若T的中心点唯一，则diameter(T) = 2 · radius(T)**
      - **若T的中心点是一对相邻顶点，则diameter(T) = 2 · radius(T) - 1**
      - **树的直径必然就是最长的通路，该通路必然经过中心点**
      - **从任意顶点v出发的BFS过程中，最后一个出队并接受访问的顶点u必是边缘点**

Chapter 7

> Search Tree

☆Binary Search Tree (BST)

- **习题解析**
  - **Q1：由n个互异节点组成的二叉搜索树，共有Catalan(n) = (2n)!/n!/(n+1)!棵**
  - **Q2：含n个节点的二叉树的最小高度为⌊log(2)n⌋**

- Q3：考查包含n个互异节点的二叉搜索树
  - 无论树的具体形态如何，BST::search()必然恰有n种成功情况和n + 1种失败情况
- Q4-1：规模为n的任何二叉搜索树，经过不超过n - 1次旋转调整，都可等价变换为仅含左分支的二叉搜索树，即最左侧通路（leftmost path）
  - 任一节点需要通过一次旋转归入最左侧通路，当且仅当它最初不在最左侧通路上
  - 若最左侧通路的长度为s，则上述算法所作的旋转调整，恰好共计n - s - 1次
  - 当根节点的左子树为空时，s = 0，最坏情况，n - 1次旋转
- Q4-2：规模为n的任何两棵等价二叉搜索树，至多经过2n - 2次旋转调整，即可彼此转换

- 顺序性
- 中序遍历序列单调非降（投影）
  - 一颗二叉树是二叉搜索树，当且仅当其中序遍历单调非降
- 查找
  - _hot节点指向“命中节点的父亲”（不存在的节点处设置“哨兵”）
  - 最好情况下O(1)，即关键码正好在树根及其附近
  - 最坏情况下，BST的高度为Ω(n)，即退化为一条单链表，查找时间复杂度O(n)
  - 当BST为BBST时，有h=O(logn)，此时平均查找时间复杂度为O(h)=O(logn)
- 插入
  - insert(e)函数先调用search(e)函数对插入值e进行查找，查找完毕后_hot节点即指向待插入节点的父亲节点
  - 插入时间复杂度主要由search()函数决定，最好情况下为O(1)，最坏情况下不超过全树高度为Ω(h)
  - 当BST为BBST时，即h=O(logn)，插入操作的平均时间复杂度为O(h)=O(logn)
- 删除
  - remove(e)也需要先调用search(e)找到待删除节点的位置，此时_hot更新为待删除节点的父亲节点
  - 三种情况：
    - 待删除节点无分支（即为叶节点）：直接删除就行
    - 待删除节点仅有单分支：删除节点后以孩子节点代替原位置即可

        - **待删除节点有双分支：调用succ()找到待删除节点的直接后继节点，与之交换，然后删除该节点并用孩子节点代替(直接后继节点必没有左孩子，故交换后可退化为单分支节点的情况，可假设反证)**
    - **删除操作总体的渐进时间复杂度不超过全树高度，即Ω(h)**
- **注意BST的高度h和节点数n并不一定存在h=O(logn)，此式当BST适度平衡时才有**

☆Balance Binary Search Tree (BBST)

- **理想平衡与适度平衡，主要是适度平衡，即h=O(logn)**
- **等价变换**
    - **局部性**
    - **zig：顺时针旋转**
    - **zag：逆时针旋转**
    - **等价变换不改变中序遍历序列（若改变了肯定不为BST，那也不是BBST）**

☆AVL Tree

> AVL树可将高度控制在O(logn)以内，从而保证每次查找、插入或删除操作均可在O(logn)内完成

- **balFac(v) = height(lc(v)) - height(rc(v))**
- **平衡因子绝对值小于等于1（<=1）**
- **完全二叉树各节点平衡因子非0即1，必为AVL树**
- **高度为h的AVL树至少包含fib(h+3) - 1个节点，平凡条件：fib(3)=2, fib(4)=3**
- **Q1：在高度为h的AVL树中，任一叶节点的深度均不小于⌈h/2⌉**
- **Q2：AVL插入节点后，失衡节点可能多达Ω(logn)个**

    > **Fib-AVL树：每个内部节点的左子树，都比右子树在高度上少一（Image）**

    - **Fib-AVL树的高度h与其规模n之间的关系：n = fib(h + 3) - 1**
    - **Fib-AVL树也是在高度固定的前提下，节点总数最少的AVL树**
- **Q3：AVL删除节点后，失衡节点至多1个**
- **Q4：对于任意大的正整数n，都存在一棵规模为n的AVL树，从中删除某一特定节点之后，的确需要做Ω(logn)次旋转，方能使全树恢复平衡**
- **Q5：按递增次序将2^(h + 1) - 1个关键码插入初始为空的AVL树中，必然得到高度为h的满树**
    - **在上述过程中，AVL树的右侧分支上的节点的左子树必为满树**

- **Q6：**
- **插入**

  > **调用BST标准search()接口查找_hot，然后直接插入即可，以下为失衡分析**

  > **由插入节点x向上寻找，首次遇到的失衡节点为g，由g出发，g的两个子树高度较高者为p，同理，p的两个子树高度较高者为v**

  - **单旋（g、p、v 同方向排列）：依据排列方向分别为zig-zig与zag-zag**
  - **双旋（g、p、v “之”字型排列）：依据排列方向分别为zig-zag与zag-zig**
  - **时间复杂度O(logn)，主要耗费在search()操作上**
  - **拓扑结构改变（即旋转操作）的时间复杂度为O(1)**
  - **插入操作前后局部子树高度复原**
  - **不会出现 失衡传播 局部满足AVL后全局即满足**
- **删除**

  > **BST::remove()**

  - **单旋（g、p、v 同方向排列）：依据排列方向分别为zig-zig与zag-zag**
  - **双旋（g、p、v “之”字型排列）：依据排列方向分别为zig-zag与zag-zig**
  - **时间复杂度O(logn)，主要耗费在search()操作上**
  - **拓扑结构改变（即旋转操作）的时间复杂度最差可达为O(logn)**
  - **删除操作前后局部子树高度可能复原也可能降低**
  - **若高度降低会出现 失衡传播 每一个祖先节点都有可能出现不平衡现象**
- **统一重平衡（3 + 4重构）**
  - **将三个g、p、v节点按中序遍历序列（即投影）重命名为a、b、c**
  - **将四棵子树也按中序重命名为T0，T1，T2，T3**
  - **此处看书p200图**
  - **无论怎么单旋、双旋，最后都会变成如图所示的结构，故在注重效率的基础上，直接进行3 + 4重构比单旋、双旋的技巧更加效率**

Chapter 8

> Senior Search Tree，**all of these are BBST**

☆Splay Tree

- **习题解析**
  - **Q1：伸展树可在任意情况下均保持所有基本操作接口分摊时间复杂度O(logn)**
    - **单次操作时间起伏量极大，不能控制在O(logn)内**
  - **Q2：**
- **局部性**：当前访问的节点或被访问节点周围的节点在下一次访问时有很大机率会被访问到
- **逐层伸展**
  - **每访问一个节点后，以其父节点为轴，进行zig或zag操作，将其提升一层，直至成为树根（v为左孩子，则zig；v为右孩子，则zag）**
  - **单次访问的最坏分摊时间复杂度高达Ω(n)**
- **双层伸展**
  - **zig-zig/zag-zag：每次都以v节点的祖父节点g进行旋转，然后再对v的父节点p进行旋转（即v、p、g 同方向排列时进行双层伸展）**
  - **zig-zag/zag-zig：与AVL树双旋过程一致，即先对父节点p旋转，再对祖父节点g旋转（即v、p、g “之”字型排列时进行逐层伸展）**
  - **每次双层伸展都可令树高几乎减半**
  - **采用双层伸展策略后，单次操作均可在分摊的O(logn)时间内完成**
- **查找**

> **BST::search() => 改变拓扑结构（伸展）**

  - **search()操作每次先调用标准的BST内部接口searchIn()定位目标节点，同时更新_hot值，无论成功与否，最后被访问的节点都将伸展至树根（若找到目标节点，则双层伸展目标节点，若目标节点不存在，则双层伸展_hot节点）**
  - **伸展树的查找算法与常规的BST::search()不同，很可能会改变树的拓扑结构，不再属于静态操作**
- **插入**

> **BST::search(e) => 将e的直接前驱伸展至树根 => 直接将e作为树根插入，由于e > pred(e)，故直接前驱部分的子树在e的左边**

  - **伸展树的插入操作会将插入后的新节点伸展至树根**
  - **直接调用Splay::search()算法查找待插入节点e，不失一般性，查找失败，此时会将_hot节点伸展至树根，然后根据新节点e与此时的树根节点比较大小，将树根节点与左右子树**

分离，插入节点e，并根据大小将树根节点作为e的左或右孩子，原先的两棵子树保持不变接上去，图见书p210

- **删除**

> **BST::search(e) => 将待删除节点e伸展至树根并删除该节点 => 在右子树中再次BST:search(e)，将右子树中的最小值（即原树根的直接后继）伸展至右子树根，然后和左子树合并即可**

  - **伸展树的删除操作会将删除后的节点的周围节点伸展至树根**
  - **直接调用Splay::search()算法查找待删除节点e，不失一般性，查找成功，此时会将该节点伸展至树根，直接删除树根，然后在右子树中再次Splay::search()节点e，必然失败，但会将右子树中最小节点m伸展至树根，图见书p210**

- **伸展树的插入删除操作都基于查找操作，核心思想是既然插入删除后的节点（或其附近的节点）需要伸展至树根，那不如先通过Splay::search()先找到该节点或_hot并将其伸展至树根，然后在树根处进行插入或删除操作即可**
- **综合评价**
  - **无需记录节点高度或平衡因子，优于AVL树**
  - **分摊复杂度为O(logn)，与AVL树相当**
  - **局部性强，缓存命中率极高时，效率可以更高**
  - **不能保证单次最坏情况的出现**
  - **不适用于对效率敏感的场合**

☆B-Tree

> 为了减少查询次数，做到在一次查询中取出最多的数据

> **不成文规定：在B-树中，n表示内部节点数；N表示关键码数**

- **习题解析**
  - **Q1：考查任意阶B-树T**

    > **下列结论与各关键码的数值大小以及具体的插入/删除过程均无关，仅取决于B-树最初和最终的状态——高度h和内部节点数n**

    - **若T的初始高度为1，而在经过连续的若干次插入操作之后，高度增加至h且共有n个内部节点，则在此过程中T总共分裂过多少次?**
      - **n - h次**
      - **B-树每次分裂都将导致n与h的差值加1，故可据此推断分裂次数**

        - 若T的初始高度为h且含有n个内部节点，而在经过连续的若干次删除操作之后高度下降至1，则在此过程中T总共合并过多少次?
            - n - h次
            - B-树每次合并都将导致n与h的差值减1，故可据此推断合并次数
        - 设T的初始高度为1，而且在随后经过若干次插入和删除操作——次序任意，且可能彼此相间。若在此期间总共做过S次分裂和M次合并，且最终共有n个内部节点，高度为h，则必有 S - M = n - h
    - Q2：按单调顺序依次插入关键码构成B-树，反复对其交替执行插入、删除操作，则每次插入或删除操作都可能会引发h次分裂或合并
    - Q3：考虑一下B-树的插入操作，为何不用类似于删除操作中的“旋转”来修复上溢，而是直接分裂节点?
- m阶B-树，即m路平衡搜索树，也称作(⌈m/2⌉, m)-树
    - 约束条件

      > 内部节点分支数 [⌈m/2⌉, m]，内部节点关键码个数 [⌈m/2⌉ - 1, m - 1]

        - 每个内部节点不超过m-1个关键码，分支不超过m个引用
        - 根节点分支数大于等于2
        - 除根以外的每个内部节点分支数大于等于⌈m/2⌉
    - 外部节点深度均相等（内部节点的叶子节点深度均相等）
    - B-树高度取决于最底层的外部节点（尽管它们不存在）
    - 孩子向量的实际长度总是比关键码向量多1（一个节点的孩子数，即分支数，总是比该节点的关键码个数多1）
- 查找
    - 在每个内部节点中进行顺序查找。查找成功返回关键码所在节点的位置；查找失败返回对应外部节点，_hot节点同步被更新为父节点
    - 查找失败总是在内部节点的叶子节点中发生
- 树高
    - 最大树高O(logmN)
    - 最小树高Ω(logmN)
    - logm(N + 1) <= h <= log⌈m/2⌉(⌊(N + 1) / 2⌋) + 1
    - 存有N个关键码的m阶B-树的高度h=Θ(logm(N))

- **与同规模的BBST相比，树高缩小为1/log2(m)**

- **插入**

  > 以下只讨论插入后发生上溢进行分裂修复的情况

  - **书图p220**
  - **取发生上溢的节点的中位数，即取秩为⌊m/2⌋的关键码，将该关键码沿父节点的引用提升一层，原节点于关键码处一分为二**
  - **经过如上操作，父节点也可能再次发生上溢（上溢的向上传递），那么重复上述操作即可，这种情况最坏不过到树根处**
  - **若上溢传递到了根节点，则如上述操作一样，将中位数节点提升一层，成为新的树根，此时树根一定恰好有两个分支，全树高度增加1，这也是B-树树高增加的唯一一种情况**
  - **每次插入操作时间复杂度为O(h)=O(logm(N))**

- **删除**

  - **先调用search()找到关键码所在的节点，进而通过向量的查找接口确定其秩，若该节点为叶子节点，直接删除该关键码以及其孩子引用即可**
  - **若上述节点不为叶子节点，则需找到该关键码的直接后继（从关键码的右子树一路向左找），并与之交换位置。因为关键码的直接后继一定位于叶节点处，故经过交换后，关键码一定处于叶节点，退化为上述情况处理即可**

  > 以下讨论删除后发生下溢进行合并修复的情况

  - **书图p223**
  - **发生下溢后首选的操作是"旋转"，即若该节点有左或右兄弟节点，这些节点在借出一个节点后不至于发生下溢，则间接的先向父亲节点借出一个关键码补给下溢节点，兄弟节点借出一个关键码给父亲节点（为了保持中序意义上的顺序性）**
  - **当左或右兄弟不存在（左和右兄弟不可能同时不存在，也就是必存在其一）或不足以借出一个关键码时，取下溢节点与其兄弟节点对应的父节点关键码，将其插入到两个下溢节点与兄弟节点之间，合并为一个新节点，父节点空缺位置左右合并并删除一个孩子引用即可。该新节点经数学论证不可能发生上溢。但此时父节点由于少了一个关键码可能会发生下溢，与上溢的传递一样，下溢最坏情况也不过传递到树根，重复上述步骤处理即可**
  - **下溢传递到树根时，会使全树高度下降1，这也是B-树高度下降的唯一可能**
  - **每次删除操作时间复杂度为O(h)=O(logm(N))**

☆Red-Black Tree

> 红黑树的插入和删除操作的拓扑结构改变的时间复杂度均可控制在O(1)

- 习题解析
  - Q1：红黑树在其平衡过程中可能需要对多达Ω(logn)个节点做重染色
  - Q2：就分摊意义而言，红黑树重平衡过程中需重染色的节点不超过O(1)个
- 约束条件
  - 由红、黑两类节点组成的BST
  - 与B-Tree一样，统一增设外部节点NULL，使之成为真二叉树（满二叉树）
  - 树根：必为黑色
  - 外部节点：均为黑色
  - 其余节点：若为红，则只能有黑孩子（红之子、之父必黑）
  - 外部节点到根：途中黑节点数目相等（黑深度，不包括外部节点）
- 提升变换
  - (2, 4)-树 == 红黑树
- 平衡性
  - 红黑树高度h=O(logn)
  - 红黑树的黑高度即是其对应B-Tree的高度
  - 红黑树的黑高度一定不小于总高度的一半
- 树高
  - 树高度：h_min = ⌊log2(N)⌋
    - 当树高h为偶数时
      - h_max = 2 · (⌊log2(N + 2)⌋ - 1)
    - 当树高h为奇数时
      - h_max = 2 · ⌊log2[(N + 2)/3]⌋ + 1
  - 树高度：log2(n + 1) <= h <= 2·log2(n + 1)——考虑外部节点
  - 黑高度：log4(n + 1) <= h <= log2(⌊(n + 1) / 2⌋) + 1

    > 由于红黑树经提升变换后相当于(2, 4)-树，故其黑高度与(2, 4)-树相同
- 插入
  - 在调用search(e)查找后，插入新节点e，随即将其染为红色（除非它是根）

- 若该节点的父节点为黑色，则插入宣告完成
- 若该节点的父亲节点为红色，则不满足红黑树的约束条件

> 因新节点的插入，而导致父子节点同为红色的情况，称作“双红缺陷”（书图p231）

> 将x的父亲与祖父记作p和g，既然此前红黑树合法，故红节点p的父亲g必存在且为黑色。g作为内部节点，其另一孩子（即p的兄弟、x的叔父）也必然存在，将其记作u，并视节点u的颜色不同，分两类情况处理

- 双红修正（RR-1），若u为黑色
  - x、p、g三者的排列有zig-zig/zag-zag/zig-zag/zag-zig四种情况，从B-Tree的角度等效的看，此时的节点颜色顺序为RRB或BRR。故将节点中中间的关键码与左或右端的关键码颜色互换，即变成RBR，即完成B-Tree意义上的修复
  - 从B-Tree的角度看，产生双红缺陷的原因是，在某个三叉节点中插入了红关键码，使得原黑关键码不再居中
  - 从B-Tree的角度看，调整之后B-Tree的拓扑结构不变，只是在节点中关键码的颜色变成了RBR
  - 注意：从红黑树本身来看，x、p、g先进行zig/zag一系列的旋转操作将x提至局部的子树根（与AVL一样），然后对三个节点进行了重染色，即中间为黑，两端为红。上述过程也可用AVL的3 + 4重构代替，即找出中序遍历意义下的中间节点，将其染为黑色，其余两个为红，重新拼接即可。
  - 上述的调整就红黑树的拓扑结构而言，仅需局部的调整即可达到全局的平衡，所以这种拓扑调整时间复杂度为O(1)
  - 双红修正（RR-1）：2次颜色翻转、2次黑高度更新，1~2次旋转，无递归
- 双红修正（RR-2），若u为红色
  - 与双红修正（RR-1）类似，经过提升变换后，实际上是B-Tree角度的节点上溢缺陷。对此，从B-Tree角度看，只需将中间关键码g分裂，并将g转红，p和u转黑即可
  - 从B-Tree的角度看，调整之后B-Tree的拓扑结构改变。且g节点被移出归入上层节点后，可能会发生上溢的向上传播，在上层节点再次引发红黑树的双红缺陷
  - 注意：从红黑树本身来看，x、p、g、u的拓扑结构没有改变，只是将g转红，p和u转黑而已（无论对于x、p、g、u的何种排列都是如此）
  - 上述的调整就红黑树的拓扑结构而言，并没有发生变化，仅需要3次的重染色操作即可完成局部的缺陷修复。虽然在双红修正（RR-2）中可能发生缺陷向上传播，但最坏也不过到树根，且缺陷的位置每次上升两层，故累计至多迭代O(logn)次
  - 双红修正（RR-2）：3次颜色翻转、3次黑高度更新、0次旋转，需要递归

- 红黑树插入节点之后的双红修正，累计耗时不过O(logn)。即便计入此前的关键码查找及节点接入等操作，红黑树的每次节点插入操作，都可在O(logn)时间内完成
- 注意：红黑树的插入操作中，只有在双红修正（RR-1）时才需要对红黑树做1~2次旋转，改变拓扑结构，而且这种旋转，在局部修复后，全局也随即修复完成。故就全树的拓扑结构而言，红黑树每次插入操作仅需O(1)次调整

- 删除
  - 在调用search(e)查找后，若查找成功，调用removeAt(x)实施删除。按照BST删除算法，此时x为实际被删除节点，其父p=_hot，x的接替者为r
  - 若此时，x为红色，r为黑色，则直接将x删除后以r代替原位置，局部子树的黑高度即可复原
  - 若此时，x为黑色，r为红色，则直接将x删除后以r代替原位置，并将r重染色为黑色，亦可使局部子树的黑高度复原
  - 若此时，x与r均为黑色，则在删除操作后，局部子树的黑高度必将降低一个单位，从而不满足红黑树约束条件

> 在节点删除过程中，被删除节点x与替代者r同为黑色的情况，称作“双黑缺陷”（书图p234）

> 为了解决双黑缺陷，需要考察原黑节点x的兄弟s（必然存在，虽可能是外部节点），并视s和p颜色的不同组合分四种情况来处理

  - 双黑修正（BB-1），若s为黑色，且s至少有一个红孩子t（假设为左孩子）
    - 从B-Tree的角度看，该情况下，节点x的删除等效于B-Tree的节点下溢缺陷，故可仿照B-Tree修复下溢缺陷的“旋转”操作，从其父节点处借出一个关键码（即p）补给x位置的空缺，再从x的兄弟节点s处借出一个关键码（即s）补给父节点的空缺。除此之外，t和p节点重染色为黑色，s节点继承p原来的颜色，整个过程r节点的颜色保持不变
    - 注意此处能用下溢的“旋转”操作，是因为s至少有一个红孩子，即经过提升变换后，包含s的超级节点中至少有两个关键码，足以借出一个
    - 从红黑树本身来看，相当于对节点t、s、p实施一次3 + 4重构（即zig与zag操作），即对t、s、p进行中序排序，将两端染为黑色，中间的节点继承p原来的颜色
    - 上述的调整使红黑树的约束条件在局部以及全局都得到恢复，删除操作遂告完成
    - 上述的调整就红黑树的拓扑结构而言，仅需局部的调整即可达到全局的平衡，所以这种拓扑调整时间复杂度为O(1)
    - 双黑修正（BB-1）：旋转1~2次，染色3次，单轮修正之后调整随即完成
  - 双黑修正（BB-2-R），若s为黑色，且s的两个孩子均为黑色，p为红色

- 与双黑修正（BB-1）一样，同样等效于B-Tree发生下溢，但此时由于s的两个孩子均为黑色，包含s的超级节点只有s这一个关键码，不足以再借出一个关键码，故此时需要用到B-Tree下溢的第二种处理方式，即将关键码p取出下移一层，与原左、右孩子合并为一个节点，并且s转红，p转黑，即s与p颜色互换
- 从红黑树本身来看，这种情况下，红黑树的拓扑结构没有发生变化，即将节点x删除后，s与p节点的颜色互换即可
- 在这种情况下，节点p为红色，故其父节点必存在且为黑色（有且仅有一个父节点），所以在p原来的超级节点中移除p之后，还余下一个关键码，不至于再次发生节点下溢，故红黑树在全局的约束条件也得到修复
- 上述的调整就红黑树的拓扑结构而言，没有发生变化，仅需局部重染色即可达到局部以及全局的平衡
- 双黑修正（BB-2-R）：0次旋转，2次染色，单轮修正之后调整随即完成

- 双黑修正（BB-2-B），若s为黑色，且s的两个孩子均为黑色，p也为黑色
  - 与双黑修正（BB-2-R）类似，也只能通过B-Tree下溢的第二种方式处理，即将p下移一层与s节点合并，并将s节点转红。但此时由于p为黑色，故含p的超级节点仅有p这一个关键码，在借出p的同时，该节点也会发生下溢，即下溢的向上传播
  - 从红黑树本身来看，红黑树的拓扑结构没有发生任何变化，即将节点x删除后，节点s重染色为红色即可
  - 经过上述调整后，红黑树的约束条件在局部得到满足，但由于一定会发生下溢的向上传播，故最多需要O(logn)次迭代
  - 双黑修正（BB-2-B）：0次旋转，1次染色，单轮修正之后必将再次双黑
- 双黑修正（BB-3），若s为红色（其他孩子均为黑）
  - 这种情况下，由于s为红色，故其父亲节点（也是x的父亲）必为黑，同时s的两个孩子节点也必为黑。此时从B-Tree角度来看，并不需要直接作下溢处理，而是将s与p互换颜色，即s重染色为黑，p重染色为红
  - 从红黑树本身来看，局部子树围绕p节点做了一次zig/zag旋转，拓扑结构发生了变化，并将s与p的颜色互换，即s转黑，p转红
  - 此时，虽然双黑缺陷还尚未得到解决，但在转换后的红黑树中，被删除节点x有了一个新兄弟s'，由于此时p已经重染色为红色，故s'一定为黑色。转换后的情况可以简述为，待删除节点x的兄弟节点s'为黑且其父节点p为红，所以这种缺陷实质上已经转化为了双黑缺陷（BB-1）或（BB-2-R）。这就意味着接下来至多再进行一次迭代调整，即可完成双黑修正
  - 经过上述至多两轮调整后，红黑树的约束条件必将在全局得到满足，拓扑结构调整的时间复杂度不过O(1)

        - **双黑修正（BB-3）：1次旋转，2次染色，单轮修正之后转为（BB-1）或（BB-2-R)**
    - **红黑树删除节点后的双黑修正，累计耗时不过O(logn)，即便计入此前的关键码查找与节点删除操作，红黑树的节点删除操作总是可在O(logn)时间内完成**
    - **注意：红黑树的删除操作中，一旦在某步迭代做过节点的旋转调整，整个修复过程便会随即完成。因此与双红修正一样，双黑修正的整个过程，也仅涉及常数次的拓扑结构调整**
- **红黑树的每次插入或删除操作，时间复杂度均为O(logn)，与AVL一样。但就拓扑结构调整的时间复杂度而言，红黑树无论是插入还是删除操作，都可控制在O(1)；而AVL树只有插入操作能将拓扑结构调整的时间复杂度控制在O(1)，删除操作最坏可达到O(logn)，这也是红黑树与AVL树最本质的一项差异**

☆kd-Tree

- **习题解析**
    - **Q1：若令Q(n) = 规模为n的子树中与查询区域边界相交的子区域（节点）总数，则有：Q(n) = O(√n)**
    - **Q2：kdSearch()的运行时间为：O(r + √n)，其中r为实际命中并被报告的点数**

Chapter 9

> Dictionary

跳转表

- **跳转表高度取决于最高塔高**
- **期望塔高为2**
- **期望跳转表高度为O(logn)**

散列表

- **☆散列函数**
    - **除余法**
        - **散列表长M取素数**
        - **hash(key) = key mod M**
    - **MAD法**
        - **M取作素数**

        - **hash(key) = (a x key + b) mod M**
    - **数字分析法**
        - **关键码key特定进制展开中抽取出若干位**
    - **平方取中法**
        - **关键码key的平方的二进制或十进制展开中取居中的若干位**
    - **折叠法**
    - **伪随机数法**
- **☆排解冲突**
    - **开散列策略**
        - **多槽位法**
            - **每个桶预留固定个数槽位供冲突元素**
        - **独立链法**
            - **冲突元素用链表接在桶单元后**
        - **公共溢出区法**
    - **闭散列策略**
        - **线性试探法**
            > **ht[hash(key)] => ht[hash(key) + 1] => ht[hash(key) + 2]...**
            - **查找链**
            - **懒惰删除**
        - **单向平方试探法**
            > **ht[(hash(key) + j^2) mod M]，j = 0, 1, 2...**
            - **当表长M为素数时，且当前装填因子小于等于（<=）50%时，平方试探总能成功；当装填因子大于（>）50%时，平方探测必然以找不到空桶而宣告插入失败（注意表述，只要M为素数，装填因子小于等于50%，一定能成功，即当M为素数数，恰好有[M/2]种取值）**
            - **当表长M为合数时，n^2 % M可能的取值可能小于[M/2]（也可能大于等于[M/2]）（注意表述）**
        - **双向平方试探**
            > **ht[(hash(key) ± j^2) mod M], j = 0, 1, 2...**

        - 对同一个j，先加后减，两个位置尝试完后再j++
        - 当表长M=4k+3时，能取遍整个散列表，即装填因子达到100%之前，插入操作必然成功
    - 再散列

      > ht[(hash(key) + j x hash'(key)) mod M]，j = 0, 1, 2...

        - 注意hash'只是作为hash的偏移量8

散列应用

- 桶排序

  > CLICK - Bucket Sort

  > 给定[0, M)内的n个互异整数（n <= M）

    - 建立一个长为M的散列表，采用hash(key) = key，将n个互异整数全部插入散列表中，最后顺序扫描散列表，打印出非空桶中的关键码

- 计数排序

  > CLICK - Accumulate Sort

  > 给定[0, M)内的n个整数

    - 建立一个长为M的数组A，初始元素均为0，将n个整数的数值作为数组下标对数组A进行更新，即A[val]++，最后顺序遍历数组，下标作为数值，对应元素作为打印次数

- 基数排序

  > CLICK - Radix Sort

- TRICKS：

    - 散列表长度M与词条关键码间隔T之间的最大公约数越大，发生冲突的可能性也将越大
    - 若关键码间隔T本身足够大而且恰好可被M整除，则所有访问词条都将互相冲突

Chapter 10

> Priority Queue

☆堆

> Heap

完全二叉堆

- **习题解析：**
  - **Q1：从堆顶通往任一叶节点的沿途上，各节点对应的关键码必然单调变化**
  - **Q2：无论怎么改进上滤操作，只能使关键码比较次数减少至O(loglogn)，但关键码交换次数依然为O(logn)，总体时间复杂度依然为O(logn)**
  - **Q3：在关键码独立均匀分布时，插入操作平均仅需O(1)**
- **逻辑上等同于完全二叉树，物理上等同于向量**
- **堆序性：父节点的优先级不小于其孩子**
- **基于向量的紧凑表示**
  - **i(lchild(v)) = 2 * i(v) + 1**
  - **i(rchild(v)) = 2 * i(v) + 2**
  - **i(parent(v)) = ⌈i(v) / 2⌉ - 1**
- **插入与上滤**
  - **元素插入到向量的末元素，满足完全二叉树的结构特性**
  - **若此时堆序性也得到满足，则插入宣告完毕**
  - **否则，若新插入节点e的优先级大于其父，则将e与父节点互换位置。这一调整策略很显然有可能引发向上传播，但最坏不超过根节点**
  - **注意：新插入节点e与父节点交换的过程在物理上是在向量中进行的，得益于完全二叉树结构上的紧凑性，每次交换可以直接在向量中直接找到需交换的两个元素的秩直接进行交换**
  - **上述过程在最坏情况下时间复杂度也不过O(logn)**
- **删除与下滤**
  - **由于完全二叉堆维护的一个优先级队列，故删除元素只涉及优先级最大元素的删除，即根节点的删除**
  - **在删除根节点之后，代之以末元素，此时完全二叉堆的结构性得以保持。若此时堆序性也得以保持，则删除操作宣告完毕**
  - **否则，将根节点与其孩子中优先级最大者互换位置。相对地，这一调整策略很显然可能引发向下传播，但最坏不过到树叶子节点**
  - **注意：下滤过程元素交换也是在向量中进行的，直接获取两元素秩交换**
  - **上述过程在最坏情况下时间复杂度也不过O(logn)**

- **注意：虽然完全二叉堆的插入和删除操作的时间复杂度都为O(logn)，但由于插入操作每次上滤只需将新节点与其父节点作一次比较，而删除操作每次下滤都需要将父节点与孩子节点至少做1次比较（除了到叶子节点附近其余都是2次比较），故删除操作的时间开销其实更大，较插入操作而言有实质性差别**

☆建堆

- **习题解析**
  - **Q1：Floyd建堆算法中，同层节点下滤的次序对建堆结果、所需时间均无影响**
- **自上而下的上滤**
  - **遍历待建堆的向量，从第一个元素开始进行上滤操作，直至遍历完毕**
  - **相当于依次将向量中的各元素调用Heap::insert()接口插入堆中**
  - **每个内部节点所需的调整时间正比于其深度**
  - **最坏情况下，时间复杂度为O(nlogn)**
- **自下而上的下滤（Floyd）**
  - **选出最末尾内部节点所对应的元素，其秩为Floor(N / 2) - 1**
  - **从该元素开始自右向左，自下而上进行局部子树的下滤操作**
  - **每个内部节点所需的调整时间正比于其高度**
  - **最坏情况下，时间复杂度为O(n)**

☆堆排序

> **CLICK - Heap Sort**

- **基于选择排序改进：选择排序每次选出前缀中最大元素插入后缀中，每次选出最大元素需遍历一遍未排序的前缀，总共需要执行n趟，故选择排序时间复杂度为O(n2)**
- **利用堆的特点，将未排序元素组成一个向量并建堆，此时最前端的元素一定是最大的元素，将该元素与末元素互换，并执行一次下滤过程，即完成一趟选择排序操作，一趟排序的时间复杂度即为堆排序下滤的时间复杂度O(logn)，总共需执行n趟，故堆排序时间复杂度为O(nlogn)**
- **堆排序 = 建堆 + 排序，实际上建堆最快需要O(n)，排序需要O(nlogn)，综合起来堆排序时间复杂度即为O(nlogn)**
- **堆排序在满足空间复杂度O(1)的要求后称为就地堆排序**

☆左式堆

> LeftHeap

- **习题解析**
  - **Q1：左孩子的npl不小于右孩子，并不意味着左孩子的高度不小于右孩子**
- **左式堆为高效的堆合并而存在**
- **NPL定义为npl(x) = 1 + min(npl(lc(x)), npl(rc(x)))**
- **约束条件**
  - **npl(lc(x)) >= npl(rc(x))，从而可推得npl(x) = 1 + npl(rc(x))**
- **包含n个节点的左式堆中，最右侧通路（右侧链）的长度不会超过 |log2(n + 1)| - 1 = O(logn)**
- **左式堆合并**
  - **递归地将a的右子堆与堆b合并，合并完成的子堆作为a的右孩子**
  - **为保证左倾性，还需要比较合并完成后a的左右孩子的npl时，必要时互换左右孩子**
  - **书p300**
  - **O(logn + logm) = O(log(max(n, m)))**
- **左式堆插入**
  - **将新插入节点视作左式堆，调用LeftHeap::merge()即可**
- **左式堆删除**
  - **左式堆删除操作即删除根节点，并调用LeftHeap::merge()合并左右子堆即可**

Chapter 11

> String，书p327

- **习题解析**
  - **Q1：BM和KMP分别擅长于处理何种类型的字符串？为什么？**
    - **习题解析[11-10]**
  - **Q2：书327四种算法时间对比图**
- **T：文本串，长为n**
- **P：模式串，长为m**

☆BF

- **最好：O(m)**

- **最坏：O(mn)**
- **在字母表长度足够大时，期望达到O(n)**

☆KMP

> CSDN-写得非常好

- **next表构造O(m)**
- **KMP总体O(n + m)**
- **Conclusion**
  - **next[]**

    > **前缀与后缀是同序比较，不是逆序比较，别和回文串搞混了**

    - **快速构造：计算某个字符对应的next值，就是看这个字符之前的字符串中有多大长度的相同前缀后缀**
    - **例子：ABCDABC**
      - **next[0] = -1（固定）**
      - **next[1] = 0（字符B之前的字符串为A，最大长度相同的前缀后缀为0）**
      - **next[2] = 0（字符C之前的字符串为AB，最大长度相同的前缀后缀为0）**
      - **next[3] = 0（字符D之前的字符串为ABC，最大长度相同的前缀后缀为0）**
      - **next[4] = 0（字符A之前的字符串为ABCD，最大长度相同的前缀后缀为0）**
      - **next[5] = 1（字符B之前的字符串为ABCDA，最大长度相同的前缀后缀为1）**
      - **next[6] = 2（字符C之前的字符串为ABCDAB，最大长度相同的前缀后缀为2）**
  - **改进next[]**

    > **需先构造出next[]**

    - **快速构造：针对模式串p以及已构造出的next[]表，必须满足p[j] != p[next[j]]，即j处的字符不能等于以next[j]为索引的字符。若p[j] != p[next[j]]，则无需改动next[j]；若p[j] == p[next[j]]，则next[j] = next[next[j]]**
    - **例子：ABCDABC**
      - **next[0] = -1（固定）**

- **next[1] = 0（p[1] = B != p[next[j]] = p[0] = A，无需改动）**
- **next[2] = 0（p[2] = C != p[next[j]] = p[0] = A，无需改动）**
- **next[3] = 0（p[3] = D != p[next[j]] = p[0] = A，无需改动）**
- **next[4] = -1（p[4] = A == p[next[j]] = p[0] = A，需要改进，next[4] = next[next[4]] = next[0] = -1）**
- **next[5] = 0（p[5] = B != p[next[j]] = p[1] = B，需要改进，next[5] = next[next[5]] = next[1] = 0）**
- **next[6] = 0（p[6] = C != p[next[j]] = p[2] = C，需要改进，next[6] = next[next[6]] = next[2] = 0）**

☆BM_BC（W-I）

> **BM算法模式串移动依然自左向右，但模式串每一位置的比对是自右向左的**

> **坏字符策略（Bad Charactor）**

- **bc表构造：O(s + m)，附加空间O(s)**
- **BM_BC总体最好情况：O(n / m)**
- **BM_BC总体最差情况：O(nm)**
- **Conclusion**
  - **bc[]**

    > **构造见如下伪码，下面几个注意点**

    - **bc[]表的长度与字母表等长，而不是与模式串等长**
    - **bc[]表的含义是模式串的每一个字符在模式串中的靠后位置（秩）**
    - **BM_BC算法每次比对失败，模式串位移 Shift = j - bc['X']**

```
//构造gs[]表（Bad Charactor）
int* buildBC(char* P){
    int* bc = new int[256]; //BC表，与字符表等长
    for(size_t j = 0; j < 256; j++) bc[j] = -1; //初始化：假设所有字符均未在P中出现
    for(size_t m = strlen(P), j = 0; j < m; j++) //自左向右扫描模式串
        bc[P[j]] = j; //将字符P[j]的BC项更新为j（单调递增）——画家算法
}
```

☆BM_GS（W-I）

BM算法模式串移动依然自左向右，但模式串每一位置的比对是自右向左的

好后缀策略（Good Suffix）

- **gs表构造：O(m)**
- **BM_GS总体最好情况：O(n / m)**
- **BM_GS总体最差情况：O(n + m)**
- **Conclusion**
  - **MS[j]：在P[0, j]的所有后缀中，与P的某一后缀匹配的最长者**
  - **ss[]：ss[j] = |MS[j]|**
  - **gs[]：**
    - **例子："ICED RICE PRICE"**
      - **MS[] => ss[]**
        - **MS[0] = "" => ss[0] = 0**
        - **MS[1] = "" => ss[1] = 0**
        - **MS[2] = "ICE" => ss[2] = 3**
        - **MS[3] = "" => ss[3] = 0**
        - **MS[4] = "" => ss[4] = 0**
        - **MS[5] = "" => ss[5] = 0**
        - **MS[6] = "" => ss[6] = 0**
        - **MS[7] = "" => ss[7] = 0**
        - **MS[8] = "RICE" => ss[8] = 4**
        - **MS[9] = "" => ss[9] = 0**
        - **MS[10] = "" => ss[10] = 0**
        - **MS[11] = "" => ss[11] = 0**
        - **MS[12] = "" => ss[12] = 0**
        - **MS[13] = "" => ss[13] = 0**
        - **MS[14] = "ICED RICE PRICE" => ss[14] = 15**
      - **ss[] => gs[]**

```
//构造gs[]表（Good Suffix）
int* ss = buildSS(P); //构造ss[]表
size_t m = strlen(P); int* gs = new int[m]; //gs[]表
for(size_t j = 0; j < m; j++) gs[j] = m; //第一步：初始化gs[]表
for(size_t i = 0, j = m - 1; j < UINT_MAX; j--) //第二步：逆向逐一扫描各字符P[j]
    if(j + 1 == ss[j])
        while(i < m - j - 1)
            gs[i++] = m - j - 1;
for(size_t j = 0; j < m - 1; j++) //第三步：正向扫描P[]各字符（注意这里不包括最后一
    gs[m - ss[j] - 1] = m - j - 1;
```

☆Karp-Rabin

- **TRICKS：**
  - **KMP算法的效率只和字符集的大小有关，和字符出现概率无关，在字符集很大时，其效率接近于蛮力算法**

Chapter 12

> Sort

☆Quick Sort（W-I）

> **CLICK - Quick Sort**

☆K-th Selection

> **CLICK - K-th Selection**

- **习题解析**
  - **Q1：元素独立等概率分布条件下，quickSelect()平均运行时间为O(n)**

☆Shell Sort

> **CLICK - Shell Sort**

☆Tournament Sort

> **CLICK - Tournament Sort**

- **胜者树每次重构需和兄弟节点比较（默认先取出父节点再取其另一个孩子，即兄弟）**
- **败者树每次重构仅需和其父节点比较，减少了io次数**

Comparison-based Algorithm（CBA）

- **Conclusion：**
  - **（1）：考查采用CBA式算法对n个整数排序**

    > **则最坏情况下不可能少于⌈log2(n!)⌉次比较**

    - **n个互异整数每次经过比较后，在其比较树中生成两个分支，而n个互异整数共有n!种比较结果，这n!种结果位于叶结点处，故比较次数等于树高，为⌈log2(n!)⌉**

Insert Sort（Input Sensitive）

> **CLICK - Insert Sort**

- **Best：O(n)**
- **Worst：O(n^2)**
- **Average：O(n^2)**
- **Space Complexity：O(1)**
- **Stability：Stable**
- **Conclusion**
  - **列表的插入排序算法平均需做n^2/4 = O(n^2)次元素比较操作**
  - **向量的插入排序算法平均需做n^2/4 = O(n^2)次元素移动操作**
  - **平均有expected-O(logn)个元素无需移动**
  - **若所有逆序对的间距均不超过k，则运行时间为O(kn)**
  - **若共有I个逆序对，则关键码比较次数不超过O(I)，总运行时间为O(n + I)**

Selection Sort

> **CLICK - Selection Sort**

- **Best：O(n^2)**
- **Worst：O(n^2)**
- **Average：O(n^2)**
- **Space Complexity：O(1)**
- **Stability：取决于具体实现，倾向于unstable**
- **Conclusion**
  - **循环节**

Merge Sort

- **Best：O(nlogn)**
- **Worst：O(nlogn)**
- **Average：O(nlogn)**
- **Space Complexity：O(n)**
- **Stability：Stable**

Bubble Sort

- **Best：O(n)**
- **Worst：O(n^2)**
- **Average：O(n^2)**
- **Space Complexity：O(1)**
- **Stability：Stable**
- **Conclusion**
  - **注意sorted标志位，一趟外循无任何元素交换则结束算法，所以最好为O(n)**
  - **每个元素都发生n-1次交换，当且仅当原序列完全逆序**

Heap Sort

> **CLICK - Heap Sort**

- **Best：O(nlogn)**
- **Worst：O(nlogn)**
- **Average：O(nlogn)**
- **Space Complexity：O(1)**
- **Stability：unstable**
- **Conclusion**
  - **底层实现为Selection Sort，建堆O(n)，Heap用于在每一趟选择最大元素时以O(1)获取最大元素，O(logn)进行堆顶的删除下滤操作**

Quick Sort

- **Best：O(nlogn)**

- **Worst：O(n^2)**
- **Average：O(nlogn)**
- **Space Complexity：O(logn)**
- **Stability：unstable**

Shell Sort

- **Best：O(n)**
- **Worst：**
- **Average：**
- **Space Complexity：O(1)**
- **Stability：unstable**

Tournament Sort

- **Best：O(nlogn)**
- **Worst：O(nlogn)**
- **Average：O(nlogn)**
- **Space Complexity：O()**
- **Stability：stable**
- **Conclusion：**
  - **第一轮**
    - **n个输入元素，先对其两两比较，自下而上组建一棵完全二叉树，选出最小者，比较次数n-1**
  - **第二轮及以后**
    - **将上一轮获胜者标为+∞（或删去），从获胜者处出发，沿其祖先重新比较，其余节点无需重赛，故第二轮及以后比较次数正比于树高，即logn**

Other Sort Algorithm

Bucket Sort

> **CLICK - Bucket Sort**

> **给定[0, M)内的n个互异整数（n <= M）**

- **Best：O(n + M)**
- **Worst：O(n + M)**
- **Average：O(n + M)**
- **Space Complexity：O(M)**
- **Stability：stable**
- **Conclusion**
  - **散列表占空间O(M)，散列表的创建与初始化(M)，将所有关键码插入散列表耗时O(n)，依次读出非空桶中的关键码耗时O(M)**
  - **散列表采用独立链法解决冲突**

Accumulate Sort

> **CLICK - Accumulate Sort**

> **给定[0, M)内的n个整数**

- **Best：O(n + M)**
- **Worst：O(n + M)**
- **Average：O(n + M)**
- **Space Complexity：O(M)**
- **Stability：stable**
- **Conclusion**
  -

Radix Sort

> **CLICK - Radix Sort**

- **Best：O(t * (n + M))**
- **Worst：O(t * (n + M))**
- **Average：O(t * (n + M))**
- **Space Complexity：O(M)**
- **Stability：stable**
- **Conclusion**

- **基数排序的底层算法一定是稳定的**

Search Algorithm

Linear Search

- **Best：O(1)**
- **Worst：O(n)**
- **Average：O(n)**

Binary Search（Ver.A）

- **Best：O(1)**
- **Worst：O(logn)**
- **Average：O(logn)**

Fibonacci Search

- **Best：O(1)**
- **Worst：O(logn)**
- **Average：O(logn)**

Binary Search（Ver.B）

- **Best：O(1)**
- **Worst：O(logn)**
- **Average：O(logn)**

Binary Search（Ver.C）

- **Best：O(1)**
- **Worst：O(logn)**
- **Average：O(logn)**

Selection Algorithm

K-th Selection

基于优先队列的选取

> 书p346

- **Ver.A**
  - **O(n + klogn)**
- **Ver.B**
  - **O(k + 2(n - k)logk)**
- **Ver.C**
  - **O(...)**
- **当k≈n/2时，上述算法时间复杂度均退化为O(nlogn)，与全排列一致**

Quick Selection（基于快速划分的选取）

- **最坏O(n^2)**

Linear Selection

- **理论上O(n)，但常系数过大**

Algorithm Strategy

Divide and Conquer

> **分治，如归并排序**

Light House

Decrease and Conquer

> **减治**

Exponential Search

> **即倍增策略**

- **k = 2 * k**
  - **O(logk)**
- **k = k * k**
  - **O(loglogk)**
- **????**
  - **O(log*n)**

习题解析[2-21]

Saddleback Search

马鞍查找

习题解析[2-22]