# 关键字

**流程**

只是作者本人使用的，一种将过程用图片展示，再配上一些伪代码和说明的表现方式，

不同算法还有不同的流程图呢，简直生动，

**文档模板**

名称

描述

流程

伪代码

实现

分析

**概念**

任何比较排序都至少是nlgn的复杂度，不同的比较排序最多只是常数因子的不同，

# 思想

## 分治思想

1，分解：将问题分解为若干子问题 和 特殊子问题 ，子问题是原问题较小的实例，特殊子问题需要特殊处理，这些解互不相干，

2，解决：递归解决这些子问题，当问题足够小时，直接解决，

3，合并：找到方法合并子问题的解，解决合并有时是一起的，

## 动态规划

dynamic programing 表格法，类似分治，应用于子问题重叠的情况，它将重复的子问题解用表格记录，

满足：

1，一个问题的最优解包含其子问题的最优解，则称为此问题具有“最优子结构性质”，

2，重叠子问题

通常有两种方法：

1，带备忘的自顶向下法

2，自底向上法

从小到大地求解子问题，使子问题所依赖的“更小的子问题”都事先求解完毕。

## 贪心算法

动态规划的“贪心”版本，它无需求解所有子问题的解（备忘或者自底向上），而是做一次“贪心”选择，

贪心算法通常都是“自顶向下”设计的：做出一个选择，然后求解剩下的子问题，

# 算法

## 概述

一个陌生的实际问题，无从下手，书上找不到原型，有时候问题本身就被描述得非常复杂，实际处理时要知道只是一开始一脸懵逼，算法就是找规律，时间久了自然有解；

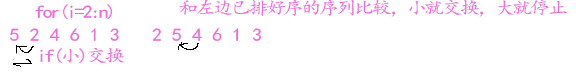
## 插入排序

描述

输入：n个数的序列（，。。。，）

输出：另一个排序序列，满足：。。。，

流程



伪代码

实现

void INSERT\_SORT(int a[], int len)

{

if (len <= 0) return;

int val = 0;

for (int i = 1; i < len; ++i) {

val = a[i];

int j = i;

while (j > 0) {

if (a[j] >= a[j - 1])

break;//小就交换，大就停止

std::swap(a[j], a[j - 1]);

--j;

}

}

}

分析

复杂度：，

可以使用“二分思想”先找到左边第一个比处理元素大的，然后后移中间的元素，再把处理元素放在那个第一个比它大的元素位置上，复杂度：

## 归并排序

描述

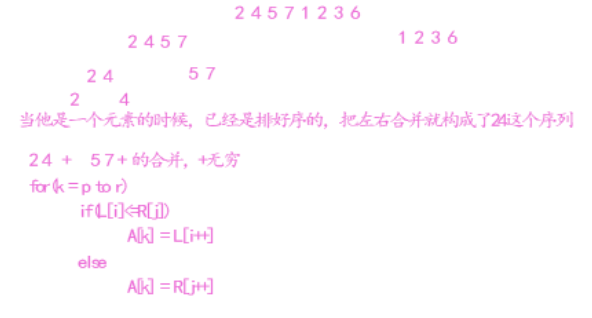
输入：n个数的序列（，。。。，）

输出：另一个排序序列，满足：。。。，

流程

分解：略。

合并的过程



伪代码

MERGE\_SORT(A,p,r)

if(p<r)

q = (p+r)/2 ] 向下取整

MERGE\_SORT(A,p,q) //分解为两个子问题

MERGE\_SORT(A,q+1,r)

MERGE(A,p,q,r) //解决、合并

MERGE(A,p,q,r)

n1 = q-p+1 元素个数

n2 = r-q

L[] = A的前n1个元素，再插入一个哨兵值

R[] = A的后n2个元素，再插入一个哨兵值

i = j = 1

for( k = p to r ) //to意味着<=

if(L[i] <= R[j])

A[k] = L[i]

++i

else

A[k] = R[j]

++j

实现

void MERGE(int A[], int p, int q, int r)

{

int nleft = q - p + 1;

int nright = r - q;

vector<int> vleft, vright;

vleft.resize(nleft + 1);

vright.resize(nright + 1);

vleft.assign(A + p, A + p + nleft);

vleft.push\_back(MAX\_VALUE);//哨兵值

vright.assign(A + q + 1, A + q + 1 + nright);

vright.push\_back(MAX\_VALUE);

int i(0),j(0);

for (int k = p; k <= r; ++k) {

if (vleft[i] <= vright[j])

A[k] = vleft[i++];

else

A[k] = vright[j++];

}

}

void MERGE\_SORT(int A[], int p, int r)

{

if (p < r) {

int q = (p + r) / 2;

MERGE\_SORT(A, p, q);

MERGE\_SORT(A, q + 1, r);

MERGE(A, p, q, r);

}

}

分析

复杂度：

**“二分”，通常的复杂度就是lgn**，二分查找的分治，归并就是一种“二分”，

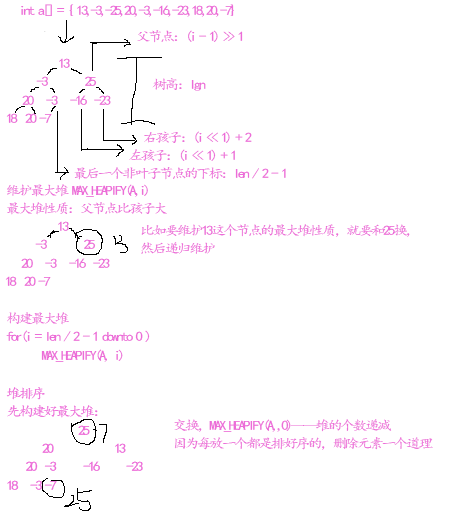
在一个序列中查找两个数之和为x，

## 堆排序

描述

原址，nlgn，把一个数组看成一颗完全二叉树（左到右填充，精妙在于满足一定公式的下标和树的节点对应），

流程



伪代码

实现

void MAXHEAPIFY(int a[], int len, int i)

{

int l = LEFT(i);

int r = RIGHT(i);

int largest = i;

//if(l<len||r<len) return;

//max3三者取大并返回下标

if (l<len && a[l]>a[i])

largest = l;

if (r<len && a[r]>a[largest])

largest = r;

if (largest != i) {

std::swap(a[largest], a[i]);

MAXHEAPIFY(a, len, largest);

}

}

void BUILD\_MAXHEAP(int a[], int len)

{

for (int i = len / 2 - 1; i >= 0; --i) {

MAXHEAPIFY(a, len, i);

}

}

void MAXHEAP\_DELETE(int a[], int len, int i)

{

//在堆中删除了，但数组中还有

int value = a[i];

std::swap(a[i], a[len - 1]);

MAXHEAPIFY(a, len - 1, i);

}

void MAXHEAP\_SORT(int a[], int len)

{

BUILD\_MAXHEAP(a, len);

int heap\_size = len;

for (int i = len - 1; i > 0; --i) {

std::swap(a[0], a[i]);

MAXHEAPIFY(a, --heap\_size, 0);

}

}

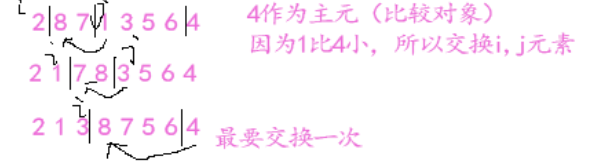
分析

## 快速排序

描述

nlgn，平均性能非常好，原址，

流程



然后递归，

伪代码

QUICKSORT(A,p,r)

if(p<r)

q = PARTITION(A,p,r) //返回下一个不用比较的元素下标（主元交换后的下标）

QUICKSORT(A,p,q - 1) //传q就会死循环

QUICKSORT(A,q+1,r)

PARTITION(A,p,r)

x = A[r] //主元

i = p – 1

for(j = p to r-1)

if(A[j] <= x)

++i

swap(a[i],a[j])

swap(a[i+1],a[r])

return i+1

实现

void QUICKSORT(int a[], int len, int p, int r)

{

if (p < r) {

int q = PARTITION(a, len, p, r);//第q+1小的元素已经排好序

QUICKSORT(a, len, p, q - 1);

QUICKSORT(a, len, q + 1, r);

}

}

int PARTITION(int a[], int len, int p, int r)

{

int x = a[r];//主元

int i = p - 1;

for (int j = p; j < r; ++j) {

if (a[j] <= x) {

++i;

std::swap(a[i], a[j]);

}

}

std::swap(a[i + 1], a[r]);

return i + 1;

}

分析

随机化版本：随机选择主元得到更好期望性能

RANDOMIZED-PARTITION(A, p, r)

i = RANDOM(p,r)

swap(A[i],A[r])

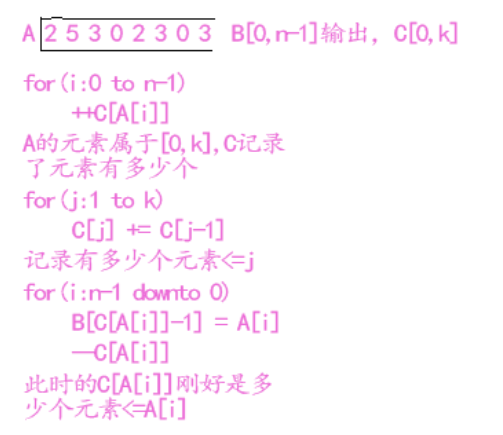
return PARTITION(A,p,r)

## 计数排序

描述

假设元素属于[0,k]的整数，对于负数、小数等要转换，因为它用下标，

流程



伪代码

实现

void COUNTSORT(int a[], int len, int b[])

{

const int k = 20;

int c[k] = { 0 };//假设元素属于[0,k]

for (int i = 0; i < len; ++i) {

++c[a[i]];

}

for (int j = 1; j < k; ++j) {

c[j] += c[j - 1];

}

for (int i = len - 1; i >= 0; --i) {

b[c[a[i]]-1] = a[i];

--c[a[i]];

}

}

分析

[0,k]，如果k很大，额外数组开销大，且对于负数小数等需要转换，Θ(k+n)，实际当k=Ο(n)时采用计数排序，

## 基数排序

描述

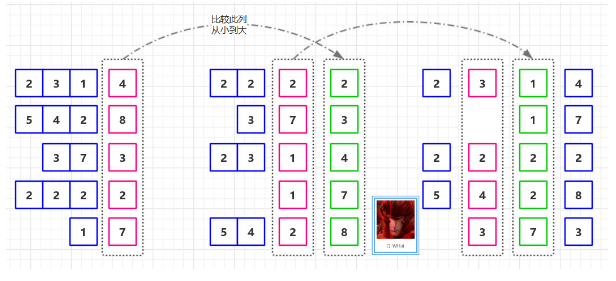
Least significant digit（LSD）

短的关键字被认为是小的，排在前面，然后相同长度的关键字再按照词典顺序或者数字大小等进行排序。比如1，2，3，4，5，6，7，8，9，10，11或者”b, c, d, e, f, g, h, i, j, ba” 。

Most significance digit（MSD）

直接按照字典的顺序进行排序，对于字符串、单词或者是长度固定的整数排序比较合适。比如：1, 10, 2, 3, 4, 5, 6, 7, 8, 9和 “b, ba, c, d, e, f, g, h, i, j”。

流程



伪代码

实现

void RadixSortLSD(int \*a, int arraySize)

{

const int MAX = 10;//

int i, bucket[MAX], maxVal = 0, digitPosition = 1;

for (i = 0; i < arraySize; i++) {

if (a[i] > maxVal) maxVal = a[i]; //找最大值，必须n-1次

}

int pass = 1;

while (maxVal / digitPosition > 0) {

int digitCount[10] = { 0 };//计数排序[0,k]，这里[0,9]

//计数排序三连招，[out]bucket

for (i = 0; i < arraySize; i++)

digitCount[a[i] / digitPosition % 10]++;

for (i = 1; i < 10; i++)

digitCount[i] += digitCount[i - 1];

for (i = arraySize - 1; i >= 0; i--)

bucket[digitCount[a[i] / digitPosition % 10] - 1] = a[i];

//rearrange

for (i = 0; i < arraySize; i++)

a[i] = bucket[i];

cout << "pass #" << pass++ << ": ";

print(a, arraySize);

//个十百千...

digitPosition \*= 10;

}

}

分析

O(d \* (n + radix))，d位数，n元素个数，radix基数如8,10,16，

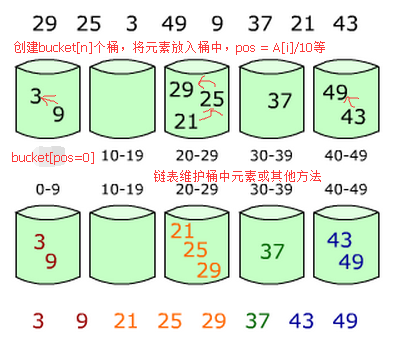
基于计数排序，[0,k]，基数使用了[0,10]，但是空间度挺大的，

## 桶排序

描述

空间换时间的排序，O(n)，

流程



伪代码

BUCKET\_SORT(A)

n = A.length

[out] : B[0,n-1] 桶，B[i]维护一张链表

for(i:0 to n-1)

A[i] insert into B[pos]的链表

for(i:0 to n-1)

sort B[i]，INSERT\_SORT或其他

合并B[0],B[1],...,B[n-1]

实现

分析

元素服从均匀分布，故落在每个桶的数量接近，桶的容量、个数、桶元素的排序等都影响性能，

## 冒泡排序

描述

两两交换，

流程

伪代码

实现

void xsort2(int\* a, int n)

{

//5,12,4,-1,3,4,28

int c = 0;

bool flag = true;

for (int i = 0; i < n; ++i) {

flag = true;

for (int j = 1; j < n - i; ++j) {

++c;

if (a[j - 1] > a[j]) {//两两比较取大的后移

flag = false;//如果前面的都排好了

swap(a[j], a[j - 1]);

}

}

if (flag)

break;

}

cout << "c2: " << c << endl;

}

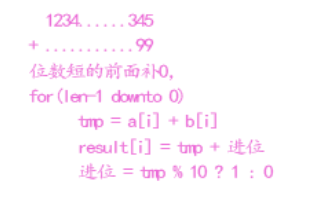
分析

## 大数相加

描述

2个超过int或long等最大范围的整数相加，

流程



伪代码

alen = astr, blen = bstr;

if(alen > blen)

bstr前补0

else

astr前补0

len = max(alen,blen)

int bflag = 0;//进位

for(k = len-1 downto 0)

tmp = astr[k] + bstr[k]

result[k] = tmp + bflag;

bflag = tmp % 10 ? 1 : 0

实现

string BigIntAdd(string a, string b)

{

int alen = a.size();

int blen = b.size();

if (alen > blen) {

b.insert(0, alen - blen, '0');

}

else {

a.insert(0, blen - alen, '0');

}

int len = std::max(alen, blen);

int bflag = 0;//进位

int tmp = 0;

int change = '0';

string result;

result.resize(len + 1);

for (int k = len - 1; k >= 0; --k) {

tmp = a[k] + b[k] - change \* 2;

result[k] = (tmp + bflag) % 10 + change;

bflag = tmp / 10 ? 1 : 0;

}

return result;

}

分析

复杂度：

对于12345678948985541345155131215545 + 123 这样的，有多余的遍历，但代码思路非常清晰，如果你的“大数”大部分是如此，那效率相差不大

之前想过一种方法是：

保存相差的部分，比如12345678948985541345155131215545 + 123，只计算123 + 545，其余拼接，但是如果有进位，比如 99...9999 + 1， 你还是要循环，所以这种吃力不讨好的事就放弃了，

## 最大子数组

描述

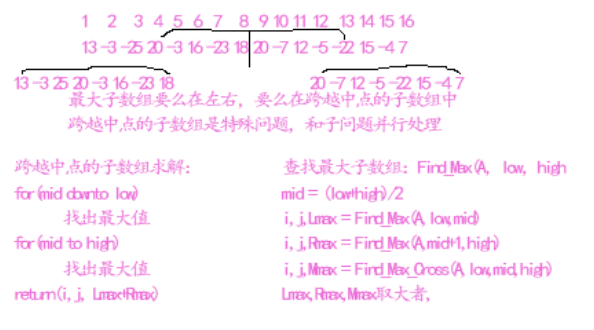
一个实际的问题是：你知道未来几日股票价格的走势，那么你如何知道再哪天买入，哪天卖出，可以达到最大收益呢？

问题转化为“最大子数组问题”：

记录16天的股票差值（相对于前一天的价格差值），假设今天股票100￥，明天113，后天110。。。



流程



伪代码

FindMaxSubArray(A,low,high)

if(high == low)

return(low,high,A[low])

else

mid = (low+high) /2

(i,j,Lmax) = FindMaxSubArray(A,low,mid)

(i,j,Rmax) = FindMaxSubArray(A,mid+1,high)

(i,j,Mmax) = FindMaxCrossSubArray(A,low,mid,high)

max(Lmax,Rmax,Mmax)

实现

main：

int a[] = { 13,-3,-25,20,-3,-16,-23,18,20,-7,12,-5,-22,15,-4,7 };

int len = sizeof(a) / sizeof(a[0]);

RetMaxSubArray ret = FindMaxSubArray(a, len, 0, len - 1);

const int MIN\_VALUE = -0xffff;

typedef tuple<int, int, int> RetMaxSubArray;

RetMaxSubArray FindMaxCrossSubArray(int A[], int len, int low, int mid, int high)

{

int lmax = MIN\_VALUE;

int sum = 0;

int lbound = 0;//左边界

for (int i = mid; i >= low; --i) {//因为是跨越中点，所以必然要mid开始

sum += A[i];

if (sum > lmax) {

lmax = sum;

lbound = i;

}

}

int rmax = MIN\_VALUE;

sum = 0;

int rbound = 0;//右边界

for (int j = mid + 1; j <= high; ++j) {

sum += A[j];

if (sum > rmax) {

rmax = sum;

rbound = j;

}

}

return make\_tuple(lbound, rbound, lmax + rmax);

}

RetMaxSubArray FindMaxSubArray(int A[], int len, int low, int high)

{

if (low == high)

return make\_tuple(low, high, A[low]);

else {

int mid = (low + high) / 2;

RetMaxSubArray lRet,rRet,mRet;

lRet = FindMaxSubArray(A, len, low, mid);

rRet = FindMaxSubArray(A, len, mid + 1, high);//high就是和low对称的下标

mRet = FindMaxCrossSubArray(A, len, low, mid, high);

//三者取最大，有卫星数据，不然max(mmax,max(lmax,rmax))

int lmax(get<2>(lRet)), rmax(get<2>(rRet)), mmax(get<2>(mRet));

if (lmax > rmax && lmax > mmax)

return lRet;

else if (rmax > lmax && rmax > mmax)

return rRet;

else

return mRet;

}

}

分析

## 最小/大值

描述

找到最小/最大值必须经过O(n-1)次比较，

同时找到最小最大值，如果分别找则需要2(n-1)次，可以花3n/2次：取2个输入元素比较，把小的与当前最小值比较，大的与当前最大值比较，这样就是比较3次，

流程

伪代码

if(n&1)

minV = maxV = a[0]

else

if(a[0]>a[1]) maxV = a[0]

else minV = a[1]

for(i: 2 to len,i+=2)

if(a[i] > a[i+1])

maxV = max(a[i], maxV)

minV = min(a[i+1],minV)

else

maxV = max(a[i+1], maxV)

minV = min(a[i],minV)

实现

typedef tuple<int, int> MinMaxRet;

MinMaxRet MinMax(int a[], int len)

{

int minV = 0, maxV = 0;

if (len & 1) {//奇数

minV = maxV = a[0];

}

else {//偶数

if (a[0] > a[1]) {

maxV = a[0];

minV = a[1];

}

else {

maxV = a[1];

minV = a[0];

}

}

int i = (len & 1) ? 1 : 2;

for (; i < len; i += 2) {

if (a[i] > a[i + 1]) {

/\*输入元素先比较，把小的与当前最小值比较，

大的与当前最大值比较，共三次比较\*/

if (a[i] > maxV)

maxV = a[i];

if (a[i + 1] < minV)

minV = a[i + 1];

}

else { //a[i] < a[i + 1]

if (a[i] < minV)

minV = a[i];

if (a[i + 1] > maxV)

maxV = a[i + 1];

}

}

return make\_tuple(minV, maxV);

}

分析

## 选择第i小的元素

描述

int q = RANDOMIZED\_PARTITION(a, len, p, r);

int k = q - p + 1;//第几小，利用这一特性

期望为线性时间

流程

伪代码

实现

int PARTITION(int a[], int len, int p, int r)

{

int x = a[r];//主元

int i = p - 1;

for (int j = p; j < r; ++j) {

if (a[j] <= x) {

++i;

std::swap(a[i], a[j]);

}

}

std::swap(a[i + 1], a[r]);

return i + 1;

}

int RANDOMIZED\_PARTITION(int a[], int len, int p, int r)

{

srand(unsigned(time(0)));

int i = (rand() % (r - p + 1)) + p;

std::swap(a[i], a[r]);

return PARTITION(a, len, p, r);

}

int RANDOMIZED\_SELECT(int a[], int len, int p, int r, int i)

{

if (p == r)

return a[p];

int q = RANDOMIZED\_PARTITION(a, len, p, r);

int k = q - p + 1;//第几小，利用这一特性

if (k == i)

return a[q];

else if (i < k)

return RANDOMIZED\_SELECT(a, len, p, q - 1, i);

else

return RANDOMIZED\_SELECT(a, len, q + 1, r, i - k);

}

分析

还有一种“最坏情况为线性时间的选择算法”

int SELECT(int a[], int len, int p, int r, int i)

{

if (p == r)

return a[p];

int x = MEDIAN(a, len, p, r);//查找中位数 O(+...)空间

int q = MEDIAN\_PARTITION(a, len, p, r, x); //O(n)

int k = q - p + 1;//第几小，利用这一特性

if (k == i)

return a[q];

else if (i < k)

return SELECT(a, len, p, q - 1, i);

else

return SELECT(a, len, q + 1, r, i - k);

}

//获取中位数

int MEDIAN(int a[], int len, int p, int r)

{

int n = r - p + 1;//多少个元素

if (1 == n)

return a[p];

int nRemains = n % 5; //不完全组的元素个数

int nGroups = nRemains ? n / 5 + 1 : n / 5;//多少组

int j = 0;

int\* b = new int[nGroups];//存放组的中位数

while (j < nGroups - 1) { //最后一组单独处理

INSERT\_SORT(a + p + j \* 5, 5);//插入排序

b[j] = \*(a + p + j \* 5 + 2);//中位数

++j;

}

//最后一组

if (nRemains) {

INSERT\_SORT(a + p + j \* 5, nRemains);

b[j] = (nRemains & 1) ?

\*(a + p + j \* 5 + nRemains / 2) : //奇数取中间一个

\*(a + p + j \* 5 + nRemains / 2 - 1);//偶数个取较小的一个

}

int nRet = MEDIAN(b, nGroups, 0, nGroups - 1); //递归

delete[] b; b = nullptr;

return nRet;

}

int PARTITION(int a[], int len, int p, int r)

{

int x = a[r];//主元

int i = p - 1;

for (int j = p; j < r; ++j) {

if (a[j] <= x) {

++i;

std::swap(a[i], a[j]);

}

}

std::swap(a[i + 1], a[r]);

return i + 1;

}

int MEDIAN\_PARTITION(int a[], int len, int p, int r,int k)

{

int i = 0;

while (a[i] != k) ++i;//还得找到中位数的位置

std::swap(a[i], a[r]);

return PARTITION(a, len, p, r);

}

## 钢铁切割问题

描述

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 长度i | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| 价格p | 1 | 5 | 8 | 9 | 10 | 17 | 17 | 20 | 24 | 30 |

问题：给定一定长度的钢条和价格表，求切割方案，使得收益最大。

新问题，慢慢磨，画画流程图，找找规律，

流程

伪代码

**纯递归版**

CUT-ROD(p,n)

if(n == 0)

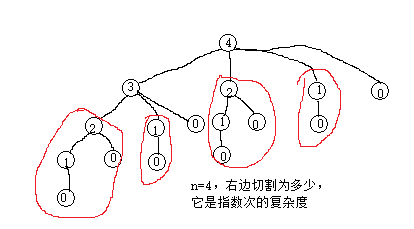
reutrn 0

q = -∞

for(i=1 to n)

q = max(q, p[i] + CUT-ROD(p,n-i)) //n=20 和 n=23差距

reutrn q



**带备忘的动态规划法**

MEMOIZED-CUT-ROD(p,n)

r[n] = {-∞}

return MEMOIZED-CUT-ROD-AUX(p,n,r)

MEMOIZED-CUT-ROD-AUX(p,n,r)

if(r[n] >= 0

return r[n]

if(n == 0)

q = 0

else q = -∞

for(i =1 to n)

q = max(q, p[i] + MEMOIZED-CUT-ROD-AUX(p,n-i,r)

r[n] = q

return q

**自底向上的动态规划法**

BOTTOM-UP-CUT-ROD(p,n)

r[n]

r[0] = 0

for(j=1 to n)

q = -∞

for(i=1 to j)

q = max(q, p[i] + r[j-i])

//if(q < p[i]+r[j-i] q = p[i]+r[j-i] s[j] = i //记录中间结果

r[j] = q

return r[n]

实现

//钢条长度对应的价格，单位：米

const int p[] = { 0,1,5,8,9,10,17,17,20,24,30,35,37,40,45,50,66,78,79,80 };

int CUT\_ROD(const int p[], int len, int n)

{

if (n == 0) {

return 0;

}

int q = MIN\_VALUE;

for (int i = 1; i <= n; ++i) { //左边切成i段

q = std::max(q, p[i] + CUT\_ROD(p, len, n - i));//左+右（递归）

}

return q;

}

//带备忘

int MEMOIZED\_CUT\_ROD\_AUX(const int p[], int len, int n, int\* r, int\*& s)

{

if (r[n] >= 0) {

return r[n];

}

int q = MIN\_VALUE;

if (n == 0)

q = 0;

else {

for (int i = 1; i <= n; ++i) { //左边切成i段

//q = std::max(q, p[i] + MEMOIZED\_CUT\_ROD\_AUX(p, len, n - i, r));//左+右（递归）

int tmp = MEMOIZED\_CUT\_ROD\_AUX(p, len, n - i, r, s);

if (q < p[i] + tmp) {

q = p[i] + tmp;

s[n] = i; //对于n米长的钢条，在i处切一刀能使最大

}

}

}

r[n] = q;

return q;

}

int MEMOIZED\_CUT\_ROD(const int p[], int len, int n, int\*& s)

{

int\* r = new int[n + 1];

memset(r, MIN\_VALUE, sizeof(int) \* (n + 1));

s = new int[n + 1];

memset(s, MIN\_VALUE, sizeof(int) \* (n + 1));

int q = MEMOIZED\_CUT\_ROD\_AUX(p, len, n, r, s);

delete[] r; r = nullptr;

return q;

}

//自底向上

int BOTTOM\_UP\_CUT\_ROD(const int p[], int len, int n, int\*& s)

{

int\* r = new int[n + 1];

s = new int[n + 1];

memset(r, 0, sizeof(int) \* (n + 1));

memset(s, 0, sizeof(int) \* (n + 1));

for (int j = 1; j <= n; ++j) { //右边切成i段

int q = MIN\_VALUE;

for (int i = 1; i <= j; ++i) { //右边i段再切

if (p[i] + r[j - i] > q) {

q = p[i] + r[j - i];

s[j] = i; //对于j米长的钢条，在i处切一刀能使最大

}

//q = std::max(q, p[i] + r[j - i]); //左+右（之前求过）

}

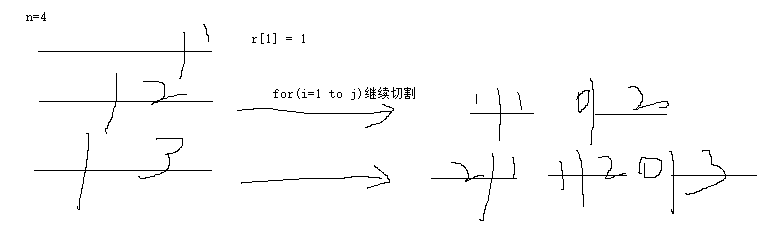
r[j] = q;

}

int nRet = r[n];

delete[] r; r = nullptr;

return nRet;

}

main：

int q = CUT\_ROD(p, len, len);

q = MEMOIZED\_CUT\_ROD(p, len, len);

int\* s = nullptr;

q = BOTTOM\_UP\_CUT\_ROD(p, len, len - 1, s);

int n = len - 1;

while (n > 0) {

cout << s[n] << ' ';

n -= s[n];

}

分析

## 最长公共子序列

### 描述

X={ A,B,C,B,D,A,B }，Y={ B,C,D,B }， 公共子序列：Z={ B,C,D,B }，可以不连续，

通过不断地“找规律”，发现如下性质：

对于X = { }，Y= {}，Z = { } 是X和Y的任意LCS（longest-common-subsequence），则有：

1，如果，则和 的一个LCS，

2，如果，那么，意味着Z是和Y的一个LCS，

3,，如果，那么，意味着Z是和的一个LCS，

有了这些性质，就可以推导出一个递归解：

令c[i,j] = 和 的LCS的长度，

c[i,j] =

其中：

1，求解和Y、和的LCS时，它的子问题的解包含在和中，构成了“重叠子问题”，

2，由“性质”可知，“自顶向下”是从序列最后一个元素开始的（按照递推来），所以“自底向上”就是从第一个元素开始，

3，钢条切割，拿q（收益）作判断，这里拿公共序列长度；因为是公共序列，所以涉及到2个对象，重构解也使用了二维数组，

流程



伪代码

### 实现

//O(mn)

void LCS\_LENGTH(const vector<char>& x, const vector<char>& y,

vector<vector<int>>& vecCnt, vector<vector<char>>& vecArrow)

{

int m = x.size();

int n = y.size();

vecCnt.resize(m + 1);

for (auto& v : vecCnt)

v.resize(n + 1);//初始值就是0

vecArrow.resize(m);

for (auto& v : vecArrow)

v.resize(n); //初始值'\0'

//vecCnt，LCS的长度，vecArrow，帮助重构最优解

for (int i = 0; i < m; ++i) {

for (int j = 0; j < n; ++j) {

if (x[i] == y[j]) {

vecCnt[i+1][j+1] = vecCnt[i][j] + 1;

vecArrow[i][j] = '\\';

}

else if (vecCnt[i][j+1] >= vecCnt[i+1][j]) {

vecCnt[i+1][j+1] = vecCnt[i][j+1];

vecArrow[i][j] = '|';

}

else {

vecCnt[i+1][j+1] = vecCnt[i+1][j];

vecArrow[i][j] = '-';

}

}

}

}

//O(m+n)

void PRINT\_LCS(vector<vector<char>>& vecArrow, const vector<char>& x, int i, int j)

{

if (i < 0 || j < 0) {

return;

}

char ch = vecArrow[i][j];

if (ch == '\\') {

PRINT\_LCS(vecArrow, x, i - 1, j - 1);

cout << x[i] << ' ';

}

else if (ch == '|') {

PRINT\_LCS(vecArrow, x, i - 1, j);

}

else {

PRINT\_LCS(vecArrow, x, i, j - 1);

}

}

分析

### 最大连续公共子序列

//连续的公共子串

void SLCS(const char\* s1, const char\* s2)

{

int m = strlen(s1);

int n = strlen(s2);

int nMaxLen = 0, nIdx = 0;

vector<vector<int>> vecCnt;

vecCnt.resize(m + 1);

for (auto& v : vecCnt)

v.resize(n + 1);//初始值就是0

for (int i = 0; i < m; ++i) {

for (int j = 0; j < n; ++j) {

if (s1[i] == s2[j]) {

//如果前一个字符相等了，它会递增，所以达到了选取“对角线”的效果

vecCnt[i+1][j+1] = vecCnt[i][j] + 1;

if (nMaxLen < vecCnt[i + 1][j + 1]) {

nMaxLen = vecCnt[i + 1][j + 1];

nIdx = i;

}

}

}

}

for (int i = 0; i<nMaxLen; i++)

{

printf("%c", \*(s1 + nIdx - nMaxLen + 1 + i));

}

}

## 活动选择

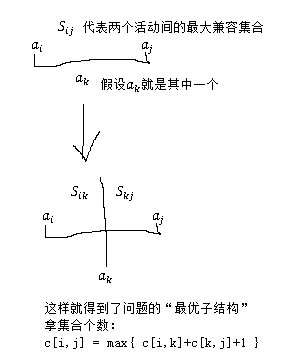
描述

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
|  | 1 | 3 | 0 | 5 | 3 | 5 | 6 | 8 | 8 | 2 | 12 |
|  | 4 | 5 | 6 | 7 | 9 | 9 | 10 | 11 | 12 | 14 | 16 |

、活动开始和结束时间，共用一个教室，如何安排活动，使选出最大兼容活动集？

流程

动态规划求解



贪心选择

实际上我们无需算出所有子问题的解，只要做一次“贪心选择”，

直观上，我应该选择这样一个活动，选出它之后，剩下的资源会被更多的其他任务所用，

此处就是活动结束时间最早的活动，

伪代码

假定活动已经按结束时间单调递增排序，添加一个，它的结束时间为0，

递归贪心算法

RECURSIVE-ACTIVITY-SELECTOR(s,f,k,n)

m = k + 1

while(m<=n and s[m]<f[k])

++m //因为f单调递增，所以第一个满足条件的就是贪心选择

if(m<=n)

return {} ∪ RECURSIVE-ACTIVITY-SELECTOR(s,f,m,n)

return ∅

迭代贪心算法

GREEDY-ACTIVITY-SELECTOR(s,f)

n = s.length

A = {}

k = 1

for(m = 2 to n)

if(s[m] >= f[k])

A = A ∪ {}

k = m

return A

实现

struct ACTIVITY {

int s = 0;

int f = 0;

ACTIVITY(int \_s, int \_f) : s(\_s),f(\_f) {}

bool operator<(const ACTIVITY& rhs) const {

return s < rhs.f;

}

};

void RECURSIVE\_ACTIVITY\_SELECTOR(const vector<ACTIVITY>& vec, int k, int n,

vector<ACTIVITY>& vecRet)

{

//假定vec按结束时间排好序了

int m = k + 1;

while (m <= n && vec[m] < vec[k]) { //m<=n,因为多了一个虚拟活动

++m;

}

if (m <= n) {

vecRet.push\_back(vec[m]);

return RECURSIVE\_ACTIVITY\_SELECTOR(vec, m, n, vecRet);

}

}

void GREEDY\_ACTIVITY\_SELECTOR(const vector<ACTIVITY>& vec, vector<ACTIVITY>& vecRet)

{

int n = vec.size() - 1;

vecRet.push\_back(vec[1]);

int k = 1;

for (int m = 2; m <= n; ++m) {

if (!(vec[m] < vec[k])) {

vecRet.push\_back(vec[m]);

k = m;

}

}

}

main:

vector<ACTIVITY> v = {

ACTIVITY(0,0), //虚拟活动

ACTIVITY(1,4),

ACTIVITY(3,5),

ACTIVITY(0,6),

ACTIVITY(5,7),

ACTIVITY(3,9),

ACTIVITY(5,9),

ACTIVITY(6,10),

ACTIVITY(8,11),

ACTIVITY(8,12),

ACTIVITY(2,14),

ACTIVITY(12,16)

};

vector<ACTIVITY> vecRet;

RECURSIVE\_ACTIVITY\_SELECTOR(v, 0, v.size() - 1, vecRet);

vecRet.clear();

GREEDY\_ACTIVITY\_SELECTOR(v, vecRet);

分析

# 数据结构

## 优先队列

描述

基于堆的数据结构，元素有一个关键字key，还有一些卫星数据，O(lgn)，

支持的操作有：

INSERT(S, x) 插入元素

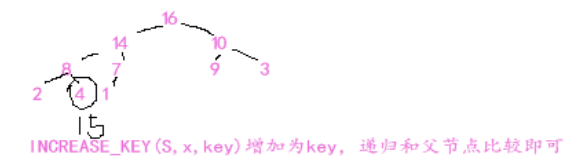
MAXIMUN(S) 返回最优先（最大/最小）

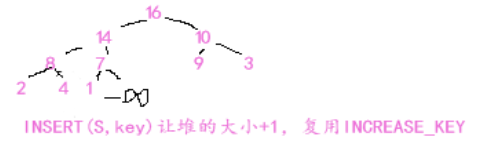
EXTRACT\_MAX(S) 弹出并返回最优

INCREASE\_KEY(S, x, k) 将关键字x增加到k

**STL：priority\_queue**

流程





伪代码

实现

void INCREASE\_KEY(vector<int>& v, int x, int k)

{

if (k < v[x])

return;

v[x] = k;

while (x > 0 && v[PARENT(x)] < v[x]) {

std::swap(v[x], v[PARENT(x)]);

x = PARENT(x);

}

}

void INSERT(vector<int>& v, int k)

{

v.resize(v.size() + 1);

v[v.size() - 1] = MIN\_VALUE;

INCREASE\_KEY(v, v.size() - 1, k);

}

另外两个非常简单，略。

分析

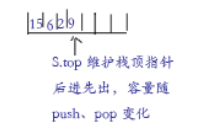
根据关键字的不同，有不同应用，作业调度-》相对优先值，事件驱动-》事件时间，

对应最大优先队列，有最小优先队列，

## 栈

描述

流程



伪代码

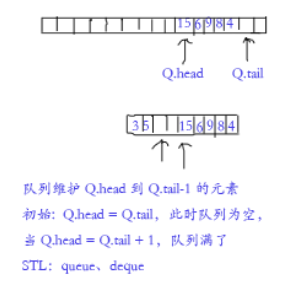
实现

分析

## 队列

描述

流程



伪代码

实现

分析

## 链表

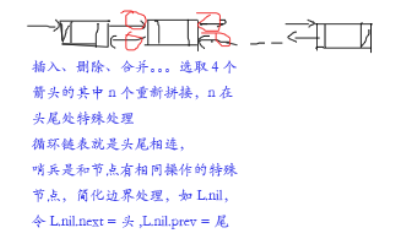
描述

单双链表、循环链表

为链表添加哨兵（多个短链表空间消耗大），

**STL: list、forward\_list**

流程



伪代码

实现

分析

## 散列表

### 描述

hash table，INSERT/SEARCH/DELETE性能极好，

用**[]**运算符，我们可以在O(1)时间内完成访问，但实际我们的数据所属的空间非常大，如[0,65535]，散列就是用与实际存储数据数目成比例的数组存储数据，并根据关键字计算下标，

直接寻址元素k存在槽k中，散列方式存在槽h(k)中，h(k)称为k的散列值，

多个关键字链接到同一个下标——冲突，解决：链接法、开放寻址法，

装载因子：α = n / m（平均每个槽的元素个数）,

散列函数的性能依赖于h(k)将元素均匀分布的程度，实际常常用启发式（试验）寻找性能好的函数；

多数散列函数的散列值都是自然数N=0,1,2...，因此字符串、汉字、负数等，要转化为一个可能很大的自然数；

除法散列

h(k) = k % m; m不应是，常是一个不太接近的素数，假设n=2000，希望α=3，则m可取701,；

乘法散列

h(k) = [m(kA – [kA])]; 取kA小数部分，m\*小数部分向下取整，A = (-1)/2；

全域散列

能将给定关键字映射到槽下标范围中的一组函数，

1)选择素数p，使关键字ε[0,p-1],aε[1,p-1],bε[0,p-1]

2)(k) = ((ak+b) % p) % m

开放寻址

减少指针的使用，每个元素都在槽里，α<1，那为什么不一个元素对应一个槽？因为元素不都是有唯一索引的，可能只是不同名，但得到相同的散列值，

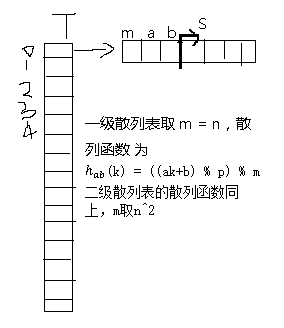
h(k,i) i为探查号，有三种方法计算探查号，双重散列能达到个探查序列，最接近均匀散列要求的m!个，

h(k,i) = ((k) + i(k)) % m; (k)必须与m互素，可以取m=,(k)总产生奇数，或者取m为素数，(k)总返回比m小的正整数，如(k)=k%m,(k)=1+k%(m-1)；

最多探查1/(1-α)次，在必须删除关键字的应用中，多用链接法；

完全散列

链表变成二级散列表了，对于静态关键字集合能在最坏O(1)时间内完成，



STL的hash\_set 和 hash\_map已经封装，具体如何使用，参照《developmemo C++基础 STL》，

流程

随机生成100个数，[0,1000]

实验几个散列函数

初始化散列表T[m]

如何找到一个素数？《算法Chapter31.8》

伪代码

### 实现

#### 链接法

class CHashTable

{

public:

CHashTable(int a[], const int len, int nSlots);

//操作

int Search(int k);

int Insert(int k);

int Delete(int k);

private:

int m\_nSlots = 0;//槽数

std::vector<std::shared\_ptr<std::list<int>>> m\_vecTable;

CHashTable() {}

protected:

int hash(int k);

};

CHashTable::CHashTable(int a[], const int len, int nSlots)

{

m\_nSlots = nSlots;

m\_vecTable.resize(nSlots);

for (int i = 0; i < len; ++i) {

const int& val = a[i];

int pos = hash(val);//元素的散列值

auto& spList = m\_vecTable[pos];

if (!spList) {

spList = std::make\_shared<std::list<int>>();

}

spList->push\_back(val);

}

}

int CHashTable::Search(int k)

{

int pos = hash(k);

auto& spList = m\_vecTable[pos];

if (spList) {

auto it = std::find(spList->begin(), spList->end(), k);

//插入：spList->push\_back(k); 删除：spList->remove(k);

if (it != spList->end()) {

return 1;

}

}

return 0;

}

int CHashTable::hash(int k)

{

//return k % m\_nSlots;

static double A = (sqrt(5) - 1) / 2;

return static\_cast<int>(floor(m\_nSlots \* (k \* A - floor(k \* A))));

}

#### 开放寻址法

int CHashTable::hash(int k, int i)

{

return (k % m\_nSlots + i \* (1 + k % (m\_nSlots - 1))) % m\_nSlots; //双重散列

}

int CHashTable::Insert(int k)

{

int i = 0, j = 0;

do {

j = hash(k, i); //vector元素都为0，执行时间O(n);元素越密集，探查次数越多

if (-1 == m\_vecTable2[j]) { //初始值是-1，假设vector元素都不是-1

m\_vecTable2[j] = k;

return j;

}

} while (i++ < m\_nSlots);

return -1; //表已经满了

}

CHashTable::CHashTable(int a[], const int len, int nSlots)

{

m\_nSlots = nSlots;

m\_vecTable2.resize(nSlots);

m\_vecTable2.assign(nSlots, -1);

for (int k = 0; k < len; ++k) {

Insert(a[k]);

}

}

int CHashTable::Search(int k)

{

int i = 0, j = 0;

do {

j = hash(k, i);

if (k == m\_vecTable2[j]) {

return j;

}

} while (i++ < m\_nSlots);

return -1;

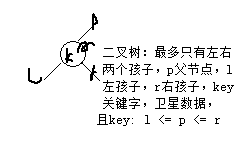
}

分析

## 二叉树

### 描述

和堆（优先队列）很像，



中序遍历：l->p->r， 先序 p->l->r，后序，l->r->p，

所有操作：O(h = lgn)，

后继：大于key的最小关键字节点，前驱：小于key的最大关键字节点，.

维护关系时，就是把l,p,r“搭好线”，线是相互的，节点在增删时，涉及三个以上节点，

对于相同关键字，要修改代码，

流程

### 伪代码

中序遍历

INORDER\_TREE\_WALK(x)

if(x != nullptr)

INORDER\_TREE\_WALK(x.left)

print x.key

INORDER\_TREE\_WALK(x.right)

**查找元素**：

TREE\_SEARCH(x,k关键字)，从节点x开始找，如果它不为空且不等于k，则判断比k小，令x=x.left，大则x.righ，

TREE\_SEARCH(x,k)

if(x == nullptr || x.key = k)

return x

else if(k < x.key)

return TREE\_SEARCH(x.left,k)

else return TREE\_SEARCH(x.right, k)

非递归会更快，

while(x != nullptr and k != x.key)

if(k < x.key)

x = x.left

else x = x.right

return x

**最大和最小关键字**：

TREE\_MINIMUM(x)，TREE\_MAXIMUM最小在最左边一个，x.left，最大在最右边x.right，

TREE\_MINIMUM(x)

while(x.left != nullptr) //x.right对称

x = x.left

return x

**前驱后继**：

TREE\_SUCCESSOR(x) ，TREE\_PREDECESSOR(x)，分有无右（后继），左（前驱）子树，

TREE\_SUCCESSOR(x) //TREE\_PREDECESSOR(x)把x.right换成x.left

if(x.right != nullptr)

return TREE\_MINIMUM(x.right)

//如果没有右孩子，则x的后继y一定是x的最底层祖先，且y的左孩子也是x的祖先，

y = x.p

while(y != nullptr && x = y.right)

x = y

y = y.p

return y

**TREE\_INSERT**(T树根,x)，插入一定是在树的最下面一层插入，

TREE\_INSERT(T,z)

x = T.root

while(x != nullptr)

y = x

if(z.key < x.key)

x = x.left

else x = x.right

z.p = y

if(y == nullptr)

T.root = z

else if(z.key < y.key)

y.left = z

else y.right = z

**TREE\_DELETE**(T,z)，分三种情况，

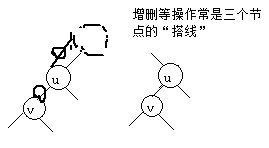
无左/右孩子，（用右/左孩子“搭线”），

有左右，找z的后继，后继是z的右孩子y（性质：y是没有左孩子的），（用右孩子“搭线”），

后继不是z的右孩子，（先y的右孩子替换y，再y替换z），

TREE\_DELETE(T,z)

定义一个子过程：TRANSPLANT(T, u, v) 用v替换u节点，没有搭v孩子的线



TRANSPLANT(T,u,v)

if(u.p == nullptr)

T.root = v

else if(u = u.p.left)

u.p.left = v

else u.p.right = v

if(v != nullptr)

v.p = u.p

TREE\_DELETE(T,z)

if(z.left == nullptr)

TRANSPLANT(T,z,z.right)

else if(z.right == nullptr

TRANSPLANT(T,z,z.left) //无左右孩子，用右/左孩子替换

else y = TREE\_SUCCESSOR(z.right) //有左右孩子，则找z的后继

if(y.p != z) //后继不是z的右孩子

TRANSPLANT(T,y,y.right)

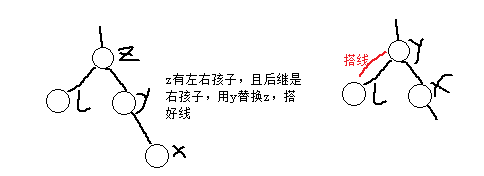
y.right = z.right

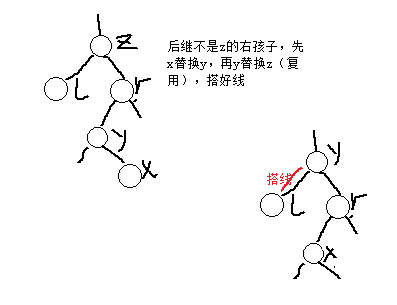
y.right.p = y

TRANSPLANT(T,z,y) //后继是z的右孩子

y.left = z.left

y.left.p = y





### 实现

struct Node {

int key = 0;

Node\* l = nullptr, \*r = nullptr, \*p = nullptr;

};

void TREE\_INSERT(Node\*& T, Node\* z)

{

Node\* x = T;

Node\* y = nullptr;

while (x != nullptr) {

y = x;

if (z->key < x->key)

x = x->l;

else

x = x->r;

}

z->p = y;

if (y == nullptr)

T = z;

else if (z->key < y->key)

y->l = z;

else

y->r = z;

}

void INORDER\_TREE\_WALK(Node\* x)

{

if (x != nullptr) {

INORDER\_TREE\_WALK(x->l);

cout << x->key << ' '; //前序和后序不是排序输出的

INORDER\_TREE\_WALK(x->r);

}

}

Node\* TREE\_SEARCH(Node\* x, int k)

{

while (x != nullptr && k != x->key) {

if (k < x->key)

x = x->l;

else x = x->r;

}

return x;

}

Node\* TREE\_MINIMUM(Node\* x)

{

while (x->l != nullptr)

x = x->l;

return x;

}

Node\* TREE\_MAXIMUM(Node\* x)

{

while (x->r != nullptr)

x = x->r;

return x;

}

Node\* TREE\_SUCCESSOR(Node\* x)

{

//空节点就尴尬

if (x->r != nullptr) {

return TREE\_MINIMUM(x->r);

}

//没有右孩子，则x的后继y一定是x的最底层祖先，且y的左孩子也是x的祖先

Node\* y = x->p;

while (y != nullptr && x == y->r) {

x = y;

y = y->p;

}

return y;

}

Node\* TREE\_PREDECESSOR(Node\* x)

{

if (x->l != nullptr) {

return TREE\_MINIMUM(x->l);

}

//没有左孩子，则x的前驱y一定是x的最底层祖先，且y的右孩子也是x的祖先

Node\* y = x->p;

while (y != nullptr && x == y->l) {

x = y;

y = y->p;

}

return y;

}

void TRANSPLANT(Node\* T, Node\* u, Node\* v)

{

//迁移对于new出来的元素是有内存泄露的

if (u->p == nullptr)

T = v;

else if (u == u->p->l)

u->p->l = v;

else u->p->r = v;

if (v != nullptr)

v->p = u->p;

}

void TREE\_DELETE(Node\* T, Node\* z)

{

if (z->l == nullptr)

TRANSPLANT(T, z, z->r);

else if (z->r == nullptr)

TRANSPLANT(T, z, z->l);

else {

Node\* y = TREE\_SUCCESSOR(z);

if (y->p != z) { //后继不是z的右孩子，先x替换y，再y替换z

TRANSPLANT(T, y, y->r);

y->r = z->r;

y->r->p = y;

}

TRANSPLANT(T, z, y); //后继是z的右孩子，y替换z

y->l = z->l;

y->l->p = y;

}

}

main ：

//对于相同元素，INSERT,MAX/MIN,SEARCH等，看代码

Node\* T = nullptr;

for (int i = 0; i < len; ++i) {

Node\* n = new Node;

n->key = a[i];

TREE\_INSERT(T, n);

}

INORDER\_TREE\_WALK(T);

cout << endl;

Node\* n = TREE\_SEARCH(T, -23);

n = TREE\_MINIMUM(T);

n = TREE\_MAXIMUM(T);

n = TREE\_SEARCH(T, 20);

//n = TREE\_MAXIMUM(n);

n = TREE\_SUCCESSOR(n);

//n = TREE\_PREDECESSOR(n);

n = TREE\_SEARCH(T, 20);

TREE\_DELETE(T, n);

INORDER\_TREE\_WALK(T);

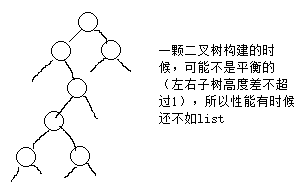
cout << endl;

### 分析

红黑树是一个更好的二叉树，STL map，multimap就是用红黑树实现的，

## 红黑树

描述



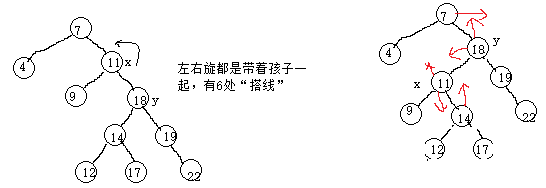
红黑树添加了性质，使平衡，从而得到最坏O(lgn)的操作时间，

1. 节点有红黑属性，根和NIL都是黑色的，
2. 一个红色节点的两个孩子都是黑色的，
3. 一个节点到其所有后代的叶节点的简单路径上，均包含相同数目的黑色节点，称为“黑高”（不包含自己），

红黑树的插入删除操作比较复杂，因为要维护红黑性质，STL map,multimap封装了红黑树，基础操作是在二叉树基础上改的，STL要重载，看二叉树即可，如果想看实现原理的话，《算法导论 Chapter红黑树》，

流程

旋转



伪代码

LEFT\_ROTATE(T,x) //RIGHT\_ROTATE

y = x.right //x = y.left

x.right = y.left //y.left = x.right

if(y.left != T.nil) //x.right

y.left.p = x //x.right.p

y.p = x.p //x.p = y.p

if(x.p = T.nil) //y.p

T.root = y //x

else if(x == x.p.left)//y == y.p.left

x.p.left = y //y.p.left = x

else x.p.right = y //y.p.right = y

y.left = x //x.right = y

x.p = y //y.p = x

实现

分析

## 数据结构的扩张

标准的数据结构已经够用了，很少情况下需要创造全新的结构，通常是存储额外信息和提供操作来扩张，

扩张指导

1. 选择基础数据结构
2. 设计附加信息
3. 对信息的维护
4. 设计新操作

### 区间树

红黑树，key值为区间，value随意，

用map，区间在map中的位置排序要调用operator<，而find也是调用它，但我们的find是要找到重叠，所以要封装新的方法

struct CRange {

int low;

int high;

CRange(int l,int h) : low(l),high(h){}

bool operator<(const CRange& r) const {

if (low < r.low)

return true;

else if (low == r.low && high < r.high)

return true;

else return false;

}

}

bool CmpOverlap(const pair<CRange,int> l, const pair<CRange, int> r) {

if (l.first.high < r.first.low || l.first.low > r.first.high)

return false;

return true;

}

main：

map<CRange, int> mapRange = {

{CRange(0,3),1},

{CRange(5,8),2},

{ CRange(6,10),3 },

{ CRange(8,9),4 },

{ CRange(5,9),4 },

{ CRange(5,9),4 },

{ CRange(5,6),4 }

};

/\*auto it = mapRange.find(CRange(0, 8));\*/ 这样是找不到的

using std::placeholders::\_1;

auto it = find\_if(mapRange.begin(), mapRange.end(),

bind(CmpOverlap, pair<CRange,int>(CRange(5, 8), 0), \_1)); //\_1是一个pair，要对应上，封装成函数就好看了

要重新指定 operator<，STL提供许多如lower\_bound可以接收二元谓词，就是让你重新指定查找排序方法，上面的操作可以封装成一个“查找区间”的函数，

# 方法

## 位运算

奇偶： a & 1 = 0/1

取第k位：a >> k & 1

清零：a = a &~ (1 << k)，置位：a = a | (1 << k)

循环左移： a = a << k | a >> 16 – k，右移： a = a >> k | a << 16 – k