线性结构

目录

[线性结构 1](#_Toc450580242)

[跳表 2](#_Toc450580243)

[KMP算法 5](#_Toc450580244)

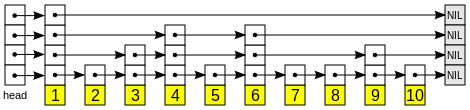
[优先队列 6](#_Toc450580245)

[矩阵优化 7](#_Toc450580246)

## 跳表

跳跃列表（也称跳表）是一种随机化数据结构，基于并联的链表，其效率可比拟于二叉查找树（对于大多数操作需要O(log n)平均时间）。

基本上，跳跃列表是对有序的链表增加上附加的前进链接，增加是以随机化的方式进行的，所以在列表中的查找可以快速的跳过部分列表，因此得名。所有操作都以对数随机化的时间进行。



上图是一个跳表的存储模型，可以发现，对每一个节点，都多了一个高度的定义，以及他下方的节点的定义，所以我们的存储结构可以如下定义

struct Node

{

int data, level;

Node \*next\_node,\*next\_level;

};

其中 next\_node 保存该右边链接的节点的地址，next\_level 保存该节点下方节点的地址。

Level 即为当前结点所在的高度。

高度我们可以用类似于投硬币的随机生成方式生成，即以下代码：

int roll\_level()

{

int ret = 1;

while (rand() % M == 1) ret++;

return ret;

}

这里的 M 是一个整数，理论上来说在M = 4 的时候跳表的性能比较优秀。

我们发现挡在level高的层次查询k时，节点信息的跨度很大，当我们发现过了我们需要查询的信息时，我们可以确定需要查询的信息就在上一次的小于k的值下方节点的右边，在上一次大于k的值的下方节点的左边。这样就缩小查询的区间。

插入操作：

void sp\_insert(int d)

{

int tmp\_level = roll\_level();

while (heads.size() <= tmp\_level)

{

int headsize = heads.size();

heads.push\_back(new Node(INF, headsize, NULL, heads[headsize - 1]) );

}

Node \*pre\_node = heads[heads.size()-1];

Node \*upper\_node = NULL;

while (true)

{

while (pre\_node->next\_node != NULL && pre\_node->next\_node->data > d)

{

pre\_node = pre\_node->next\_node;

}

if (pre\_node->level > tmp\_level)

{

pre\_node = pre\_node->next\_level;

continue;

}

Node \*tmp = new Node(d, pre\_node->level, pre\_node->next\_node, NULL);

pre\_node->next\_node = tmp;

if (upper\_node != NULL)

{

upper\_node -> next\_level = tmp;

}

upper\_node = tmp;

pre\_node = pre\_node->next\_level;

if (upper\_node->level == 1) break;

}

}

查询操作：

pair<Node\*, Node\*> sp\_search(int d)

{

Node \*now\_node = heads[heads.size()-1];

Node \*pre\_node = NULL;

if (now\_node == NULL) return make\_pair((Node \*)NULL, (Node \*)NULL);

while (true)

{

while (now\_node->next\_node != NULL && now\_node->next\_node->data >= d)

{

pre\_node = now\_node;

now\_node = now\_node->next\_node;

}

// cout << "hello" << endl;

// cout << now\_node->data << endl;

if (now\_node == NULL) return make\_pair((Node \*)NULL, (Node \*)NULL);

if (now\_node->data == d)

{

return make\_pair(now\_node, pre\_node);

}

else

{

now\_node = now\_node->next\_level;

}

if (now\_node == NULL) return make\_pair((Node \*)NULL, (Node \*)NULL);

}

return make\_pair((Node \*)NULL, (Node \*)NULL);

}

删除操作：

bool find(int d)

{

if(sp\_search(d).first == NULL) return false;

else return true;

}

int sp\_delete(int d)

{

pair<Node\*, Node\*> pnn = sp\_search(d);

if (pnn.first == NULL)

{

return -1;

}

Node \*pre = pnn.second;

Node \*now = pnn.first;

while (true)

{

while(pre->next\_node->data > d) pre = pre->next\_node;

Node \*tmp = now;

pre->next\_node = now->next\_node;

now = now->next\_level;

pre = pre->next\_level;

delete tmp;

if(pre == NULL) break;

}

return 1;

}

## KMP算法

相对于一般的字符串匹配，KMP算法优化的地方就在于   当发现当前匹配的位置 k + 1 匹配失败时，不是再回到 p 的开始位置进行匹配，而是回到 next[k] 开始匹配！

next[k] 记录了 p[0...k] 中，最长的相同前缀后缀长度 - 1 , -1 是为了转移的时候方便(数组从 0 开始)。

比如：

对于

/\*

p:

| a | b | a | a | b | c | a | c |

next:

| -1| -1| 0 | 0 | 1 | -1| 0 | -1|

\*/

相同前缀后缀就是前缀和后缀相同；

然后就是 next 数组的求法：

void get\_next(char \*p, int \*next){

next[0] = -1;

int k = -1;

for(int i = 1; i < p.length; i++){

while(k > -1 && p[k + 1] != p[i])

k = next[k];

if(p[k + 1] == p[i]){

k++;

}

next[i] = k;

}

}

有了 next 数组之后，在匹配字符串程序中发现不匹配的位置时，不需要将 p 的位置变量重置为 0 了，将其赋值为 next[k] 就行。

## 优先队列

优先队列是计算机科学中一类特殊的数据结构的统称。优先队列通常是一个可以被看做一棵树的数组对象。在队列中，调度程序反复提取队列中第一个作业并运行，因为实际情况中某些时间较短的任务将等待很长时间才能结束，或者某些不短小，但具有重要性的作业，同样应当具有优先权。优先队列即为解决此类问题设计的一种数据结构。

class Priority\_queue

{

private:

int len;

int data[100000];

// vector<int> data;

public:

void heap\_adjust(int pos)

{

for (int nxt = pos \* 2; nxt <= len; pos = nxt, nxt = pos \* 2)

{

if (nxt < len && data[nxt + 1] > data[nxt])

nxt ++;

if(data[pos] > data[nxt]) break;

swap(data[pos], data[nxt]);

}

}

void init(vector<int> vi)

{

len = vi.size();

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

{

data[i+1] = vi[i];

}

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

{

heap\_adjust(i);

}

}

void push(int d)

{

data[++len] = d;

int now = len, p = now / 2;

while(p > 0 && data[p] < data[now])

{

swap(data[p], data[now]);

now = p;

p = now / 2;

}

}

void pop()

{

data[1] = data[len--];

heap\_adjust(1);

}

int top()

{

return data[1];

}

bool empty()

{

return len == 0;

}

void clear()

{

len = 0;

memset(data, 0, sizeof(data));

}

void show()

{

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

{

cout << data[i] << " ";

}

cout << endl;

}

Priority\_queue()

{

clear();

}

}pq;

对于优先队列，插入、删除一个元素的复杂度都是 log(n) , 其中n为优先队列中元素的个数。

## 矩阵优化

因为稀疏矩阵中有大量的0元素，如果用二维数组存储的话会浪费大量的空间，所以我们采用三元组的方式储存矩阵

struct Triple

{

int i, j, data;

Triple(){}

Triple(int i, int j, int data):i(i),j(j),data(data){}

void show() {

printf("Matrix[%d][%d] = %d\n", i,j,data);

}

bool operator < (const Triple& A) const {

if(i == A.i) return j < A.j;

return i < A.i;

}

};

在这中矩阵的储存方式下，转制运算有一个可以优化的地方，就是当矩阵中元素是以行坐标为第一关键字，列坐标为第二关键字时排序时，转至即为将其以列为第一关键字，行为第二关键字排序，所以我们观察到，对于某一列的元素，以第一种排序的方式和第二种排序的方式相对顺序不变，所以我们可以用类似于基数排序的方式来排序，这样就将排序的复杂度从nlogn降低至n。代码如下：

Matrix qtranspose() {

memset(sum, 0, sizeof(sum));

memset(pos, 0, sizeof(pos));

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

sum[matrix[i].j] ++;

for (int i = 1; i <= m; i++)

pos[i] = pos[i-1] + sum[i-1];

Matrix ret = Matrix(m, n, cnt);

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

{

ret.matrix[pos[matrix[i].j]++] = matrix[i].swapij();

}

return ret;

}