目录

[实验要求 2](#_Toc438812699)

[二叉查找树 2](#_Toc438812700)

[二分查找 4](#_Toc438812701)

[排序算法 5](#_Toc438812702)

# 实验要求

1.随机产生一组整数序列（最少10个），建立二叉查找树，并完成查找、插入、删除任一结点的算法。

2.对二叉查找树中序遍历得到一个升序序列，对该序列完成折半查找。

3.随机生成一组10000个左右的数据，分别用冒泡排序、插入排序、选择排序、快速排序完成排序工作，并给出时间效率。

# 二叉查找树

###### 增加节点

从根节点开始，如果要插入的元素大于当前结点，那么遍历二叉树的右子树，否则遍历二叉树的左子树，直到当前的节点为空，就将此节点的值赋值为需要插入的元素。

void add(Node \*pre, Node \*&now, int d)

{

if(now == NULL) now = new Node(d, NULL, NULL, pre);

else if(d < now->data) add(now, now->lson, d);

else add(now, now->rson, d);

}

###### 删除节点

首先查找节点， 如果查找到就删除当前结点，删除的时候先将左子树中最大的节点（或者右子树中最大的节点）和需要删除的节点交换位置。让后以左子树为根再调用删除的函数，这样可以保证需要删除的节点一定在子树的最左或者最右，然后直接将其删除即可。

Node\* find(Node \*now, int d)

{

if(now == NULL) return NULL;

if(now->data == d) return now;

if(now->data < d) return find(now->rson, d);

else return find(now->lson, d);

}

Node\* findmax(Node \*now)

{

while(now->rson != NULL) now = now->rson;

return now;

}

Node\* findmin(Node \*now)

{

while(now->lson != NULL) now = now->lson;

return now;

}

void deldata(Node \*now, int d)

{

Node \*tmp = find(now, d);

if(tmp == NULL)

{

return;

}

if(tmp -> lson == NULL && tmp -> rson == NULL)

{

Node \*fa = tmp->fa;

if(fa != NULL)

{

if(fa->data < d)

fa->rson = NULL;

else

fa->lson = NULL;

}

delete tmp;

}

else if(tmp->lson == NULL)

{

Node \*fa = tmp->fa;

if(fa != NULL)

{

cout << "in del" << endl;

cout << tmp->data << endl;

if(fa->data < d)

fa->rson = tmp->rson;

else

fa->lson = tmp->rson;

}

tmp->rson->fa = fa;

Node \*tt = tmp->rson;

delete tmp;

tmp = tt;

}

else if(tmp ->rson == NULL)

{

Node \*fa = tmp->fa;

if(fa != NULL)

{

if(fa->data < d)

fa->rson = tmp->lson;

else

fa->lson = tmp->lson;

}

tmp->lson->fa = fa;

Node \*tt = tmp->lson;

delete tmp;

tmp = tt;

}

else

{

Node \*nd = findmax(tmp->lson);

// cout << " !! " << nd->data << endl;

swap(tmp->data, nd->data);

deldata(tmp->lson, d);

}

}

###### 遍历节点

二叉查找树的中序遍历就是已给有序的序列。

void dfssort(Node \*now, vector<int> &vi)

{

if(now == NULL) return ;

if(now -> lson != NULL) dfssort(now -> lson, vi);

if(now-> data != 0x3f3f3f3f)vi.push\_back(now->data);

if(now -> rson != NULL) dfssort(now -> rson, vi);

}

# 二分查找

因为数组是有序的，所以每次只需要比较区间的中间值与需要查找的数据的大小关系，就可以缩小需查找元素可能的区间范围为原来的一半。知道查找到该元素或者区间长度为0 即为不存在该元素。

int binary\_search(vector<int> v, int l, int r, int x)

{

int mid = (l + r) / 2;

while(v[mid] != x && l < r)

{

if(v[mid] < x) l = mid+1;

else r = mid - 1;

mid = (l + r) / 2;

}

if(v[mid] == x)

cout << "mid is " << mid << endl;

else

cout << "no such element" <<endl;

return mid;

}

# 排序算法

typedef vector<int> VI;

void buble\_sort(VI &v)

{

int len = v.size();

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

{

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

{

if(v[j] > v[j+1])

swap(v[j], v[j+1]);

}

}

}

冒泡排序:O(n\*n)

void select\_sort(VI &v)

{

int len = v.size();

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

{

int pos = i;

for(int j = 0; j < i; j++)if(v[j] > v[pos])

pos = j;

swap(v[pos], v[i]);

}

}

选择排序:O(n\*n)

void insert\_sort(VI &v)

{

int len = v.size();

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

{

int tmp = v[i];

int pos = i;

while(tmp < v[pos-1] && pos > 0)

{

v[pos] = v[pos-1];

pos--;

}

v[pos] = tmp;

}

}

插入排序:O(n\*n)

void quick\_sort(VI &v, int l, int r)

{

if(l >= r) return ;

int i = l-1;

int mid = v[r];

for(int j = l; j < r; j++)

if(v[j] < mid)

{

i++;

swap(v[i], v[j]);

}

swap(v[i+1], v[r]);

quick\_sort(v, l, i);

quick\_sort(v, i+2, r);

}

快速排序:O(n\*logn)