数据结构实验报告

【算法说明】

普通二叉搜索树：

插入：

1、 判断插入数据与根节点的大小关系，寻找插入数据的位置，返回值为位置+该数据是否已经存在。
2、 插入数据，新建节点，该节点肯定为叶子节点，左右孩子均为空指针。

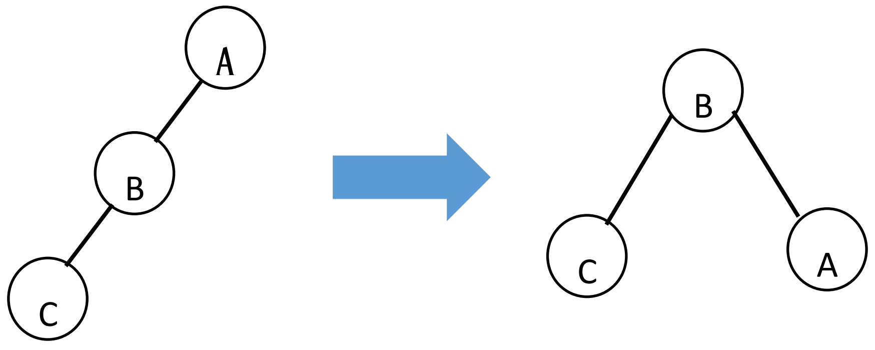
删除：

1、 判断插入数据与根节点的大小关系，寻找插入数据的位置，返回值为位置+该数据是否存在。
2、 判断该节点是否有左右孩子节点
3、 删除节点（记得要将所有能指向该位置的指针指向新位置或者置零）
    （1） 没有左右孩子节点：直接删除该节点
    （2） 只有左孩子或者右孩子节点：使该节点父节点的左/右孩子指针指向该节点的左/右孩子，删除该节点（注意要记得将该节点的左/右孩子节点的父指针指向该节点的父节点）如果是根节点则交换该指针和其左/右孩子节点的值，然后删除其左/右孩子节点。
    （3） 左右孩子节点都非空：将该节点的值与其左子树的最右叶子节点（也就是该节点的前驱）的值交换，交换后删除该节点的前驱，注意如果该前驱有子节点，则要将其子节点指向新的父节点。

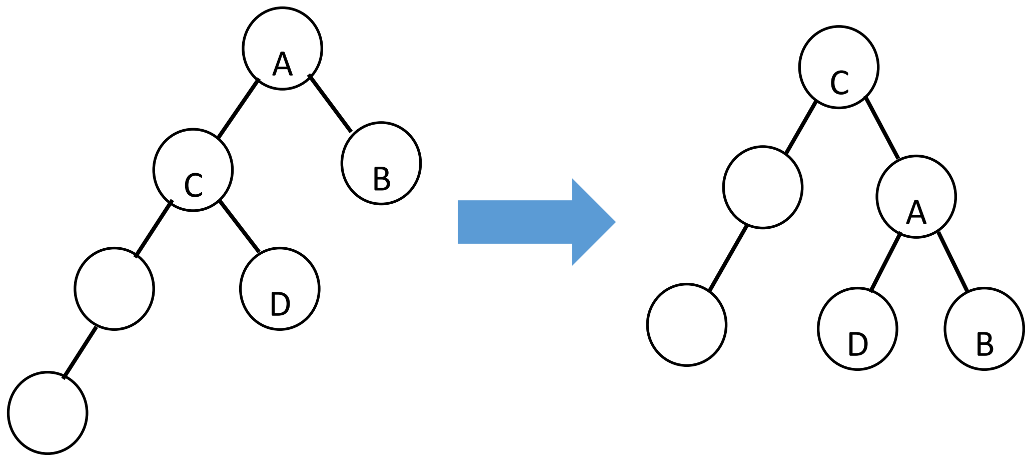
AVL 平衡树：

插入/删除

1、 与普通二叉搜索树的插入/删除基本一致，但是在插入和删除后要对二叉树进行调整，旋转二叉树使其平衡。在插入节点后，改变这一段节点的 imbalance 直到有节点的 imbalance 为 2 或–2。
2、 旋转：
    旋转主要是要找到离插入的节点最近的不平衡点的位置（也就是最近的 imbalance 为 2 或–2 的位置）然后根据插入/删除的情况不同来进行不同旋转。
    下面以 imbalance=–1 为例，等于 1 时与下图镜面对称。
    （1）**左旋**

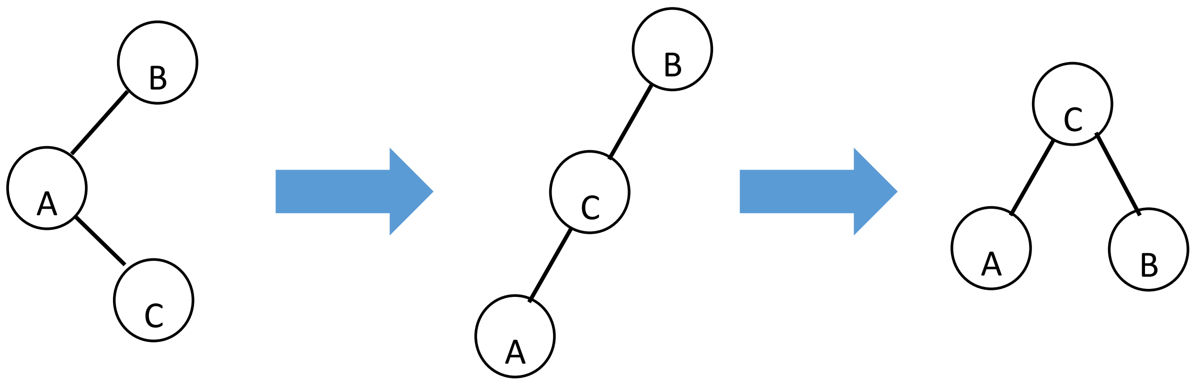


情况 1

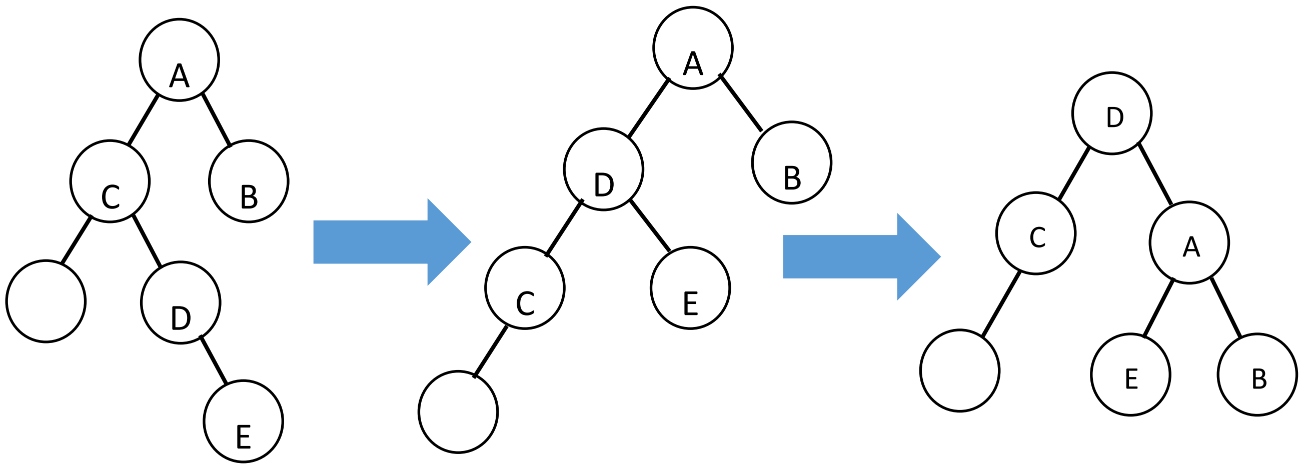


情况 2

（2）右左旋：



情况 1

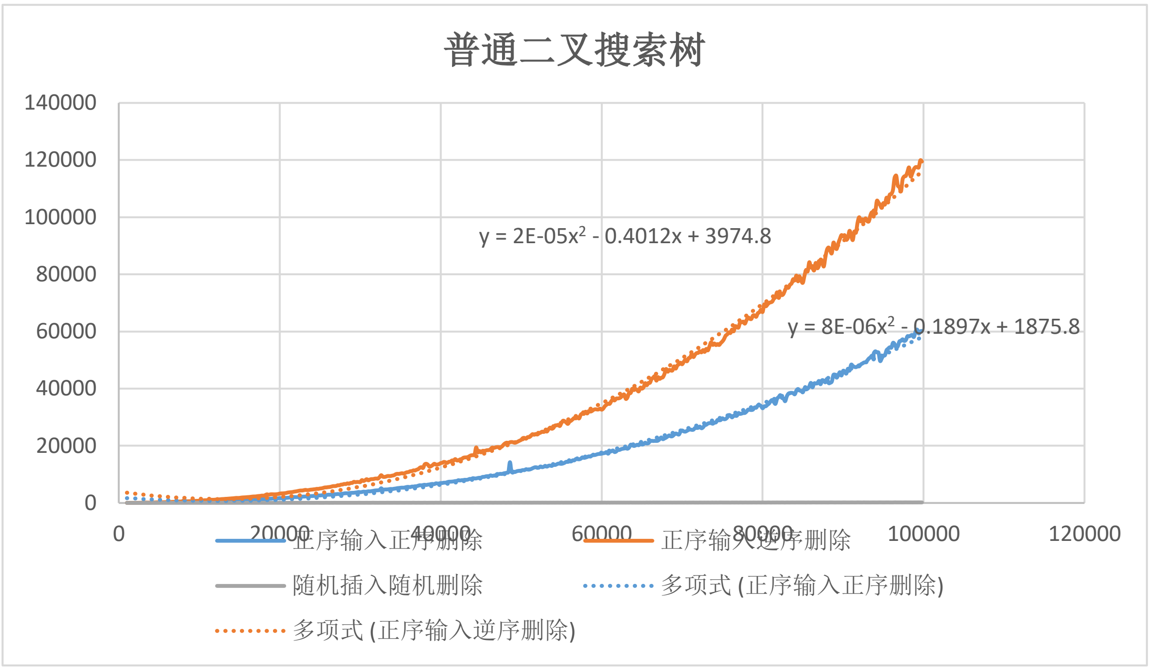


情况 2

【测试结果】

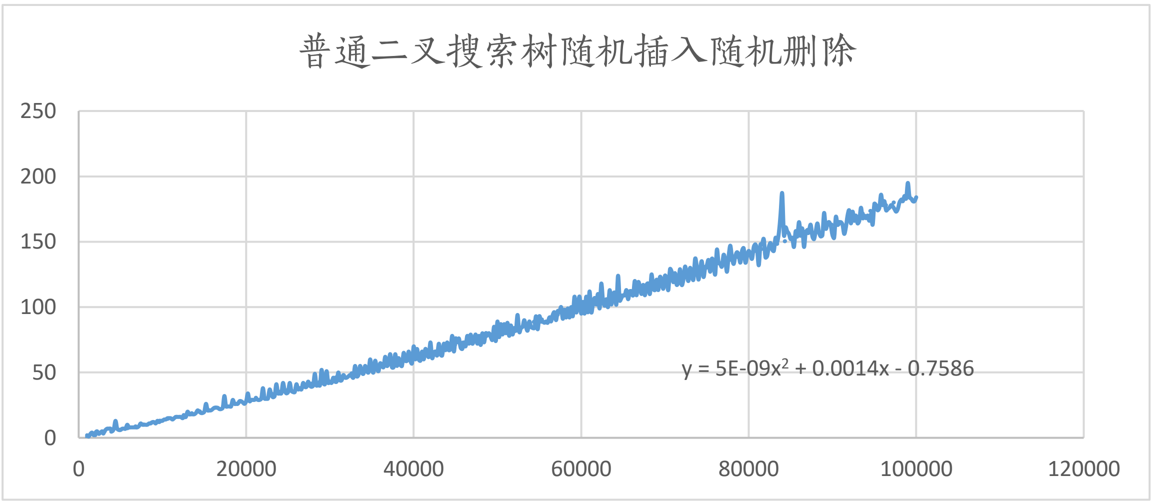
测 试 用 例 表						
	二叉搜索树			AVL 平衡树		
输入顺序	正序输入 正序删除	正序插入 逆序删除	随机插入 随机删除	正序输入 正序删除	正序插入 逆序删除	随机插入 随机删除
理论时间 复杂度	$O(n)+O(1)$	$O(n)+O(n)$	$O(\log_2 n)$	$O(\log_2 n)$	$O(\log_2 n)$	$O(\log_2 n)$
实际时间 复杂度	$O(n^2)$	$O(n^2)$	$O(\log_2 n!)$	$O(\log_2 n!)$	$O(\log_2 n!)$	$O(\log_2 n!)$
错误原因	无	无	无	无	无	无
当前状态	良好	良好	良好	良好	良好	良好

测 试 用 例 表 2						
	四阶 B 树			红黑树		
输入顺序	正序输入 正序删除	正序插入 逆序删除	随机插入 随机删除	正序输入 正序删除	正序插入 逆序删除	随机插入 随机删除
理论时间 复杂度	$O(\log_2 n)$	$O(\log_2 n)$	$O(\log_2 n)$	$O(\log_2 n)$	$O(\log_2 n)$	$O(\log_2 n)$
实际时间 复杂度	O（n）	O（n）	O（n）	$O(n)$	$O(n)$	$O(\log_2 n!)$
错误原因	无	无	无	无	无	无
当前状态	良好	良好	良好	良好	良好	良好



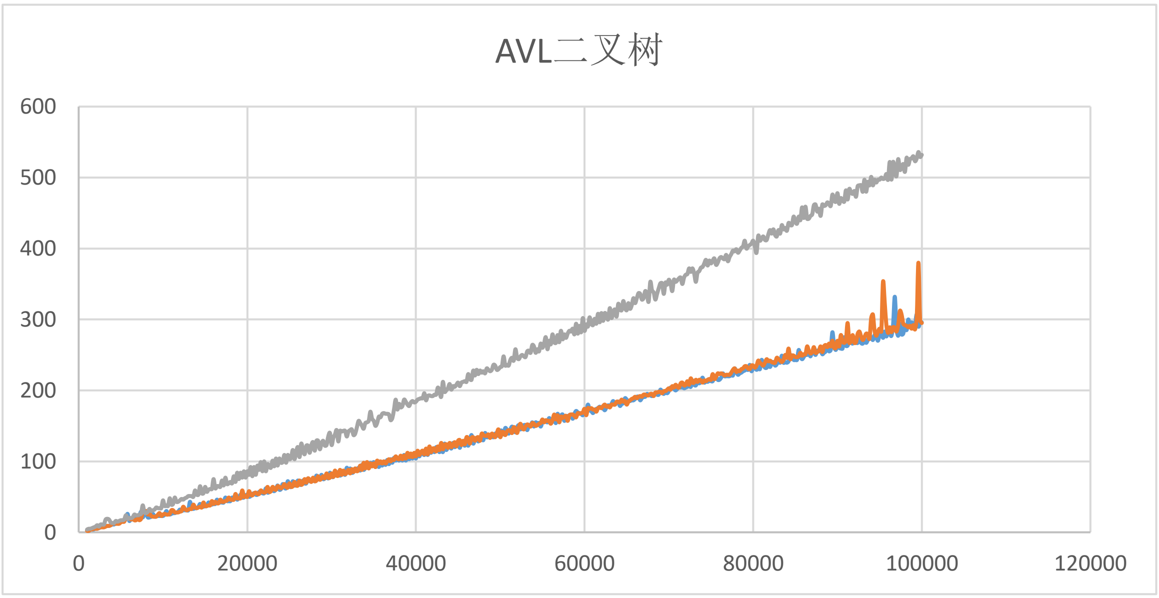
普通二叉搜索树
140000
120000
100000
80000
60000
40000
20000
0
0
20000
40000
60000
80000
100000
120000
y = 2E-05x² - 0.4012x + 3974.8
y = 8E-06x² - 0.1897x + 1875.8
正序输入正序删除
正序输入逆序删除
随机插入随机删除
多项式 (正序输入正序删除)
多项式 (正序输入逆序删除)

图 1.1

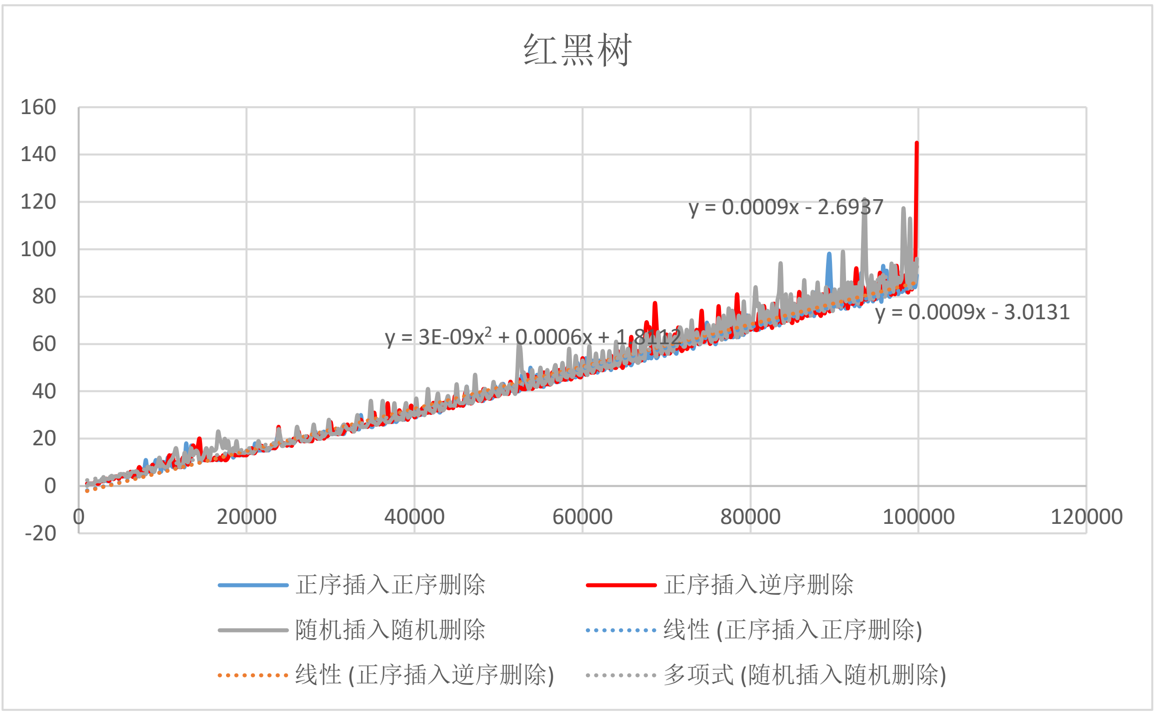


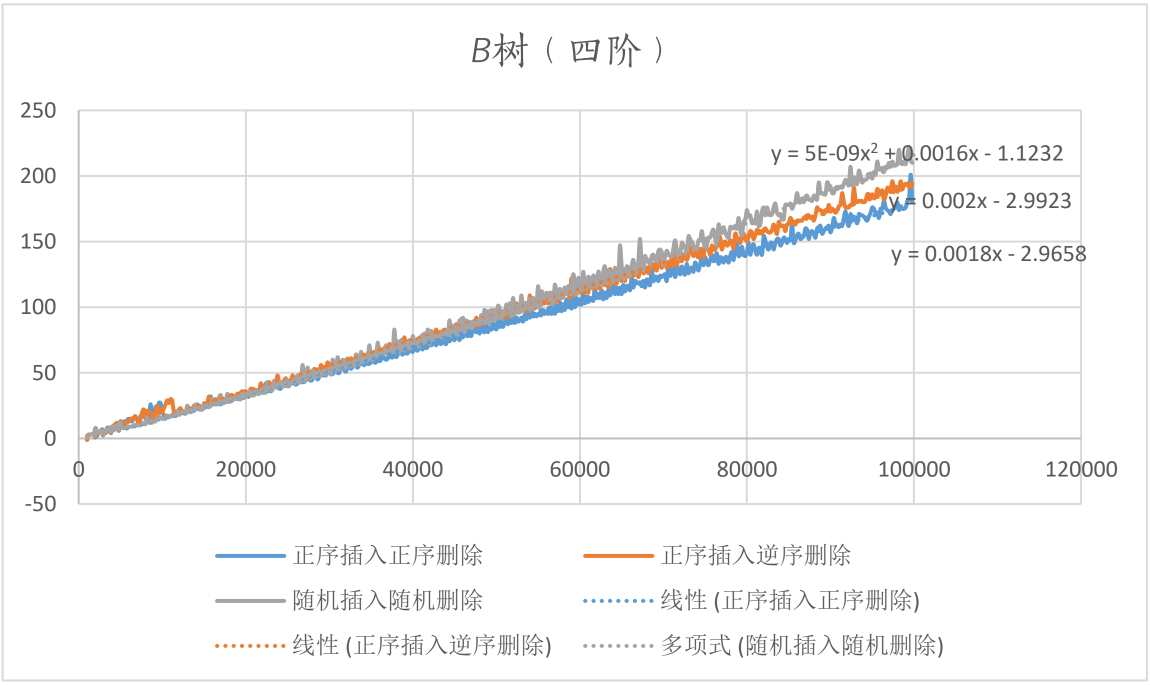
普通二叉搜索树随机插入随机删除
250
200
150
100
50
0
0
20000
40000
60000
80000
100000
120000
y = 5E-09x² + 0.0014x - 0.7586

图 1.2



AVL二叉树
600
500
400
300
200
100
0
0
20000
40000
60000
80000
100000
120000





【分析与报告】

一、测试结果分析

普通二叉树：

非平衡树，在正序插入时形成链表，插入一个节点的时间复杂度为 O（n）。在随机插

入时，插入的时间复杂度取决于树的高度，如果二叉搜索树是平衡树，那么树的高度为$\log_2 n+1$,插入的时间复杂度为$O(\log_2 n)$，随机删除的时间复杂度与随机插入相同。在正序删除时，删除的每一个节点都是根节点，删除的时间复杂度为 O（1）, 逆序删除时，每次都删除最后一个节点，时间复杂度为 O（n）。在本实验中，输出的时间是插入删除 n 个节点的时间，因此，正序插入的时间复杂度为：

$O(1)+O(2)+\ldots\ldots+O(n)=O((n-1)n/2)=O(n^2)$

随机插入的时间复杂度为：

$O(\log_2 1)+O(\log_2 2)+\ldots\ldots+O(\log_2 n)=O(\log_2 1*2\ldots\ldots*n)=O(\log_2 n!)$

正序删除的时间复杂度为：

$O(1)+O(1)+\ldots\ldots+O(1)=n*O(1)=O(n)$

逆序删除的时间复杂度为：

$O(1)+O(2)+\ldots\ldots+O(n)=O((n-1)n/2)=O(n^2)$

可以得到正序插入正序删除的时间复杂度为 O（n^2）+O（n）, 正序插入逆序删除的时间复杂度为 O（n^2）+O（n^2）, 在图 1.1 中可以看出正序插入逆序删除的时间复杂度要高于正序插入正序删除的时间复杂度，并且在数据量较大时，正序插入正序删除和正序插入逆序删除的时间复杂度远高于随机插入随机删除的时间复杂度。图 1.2 单独显示了随机插入随机删除 n 个数的运行时间，函数的拟合曲线为 $y=k\log_2 n!$ ，由于在 word 中无该拟合方式，且 k 太小，图像与 $y=kx^2$ 相似，所以图中用 $y=kx^2$ 来拟合数据。

AVL 平衡树：

在每一次插入和删除后都对树进行调整，通过旋转使树的左右子树的高度一直保持一致，树的高度大约能够保持在$\log_2 n$左右，搜索的时间复杂度也就是 $O(\log_2 n)$。此时，三种插入和删除方式生成的树的高度都差不多。因此这三种插入删除方式时间复杂度

也相差不大。插入 n 个元素的时间复杂度都为：

$O(\log_2 1)+O(\log_2 2)+\ldots\ldots+O(\log_2 n)=O(\log_2 1*2\ldots\ldots*n)=O(\log_2 n!)$

B 树（四阶）:

三种插入删除的效率近似相同，随机插入随机删除的时间复杂度微微高一些。因为 m 叉 B 树的每个节点上至少要有 m/2 个元素，所以 m 叉 B 树的最大高度为$\log_{m/2} n$，插入以及删除的时间复杂度也就相当于搜索的时间复杂度也就是树的高度，实验中用 4 阶 B 树，所以插入一个元素的时间复杂度为 $O(\log_2 n)$(将合并与分裂的时间复杂度看作 $O(1)$),插入 n 个元素则是：

$O(\log_2 1)+O(\log_2 2)+\ldots\ldots+O(\log_2 n)=O(\log_2 1*2\ldots\ldots*n)=O(\log_2 n!)$

红黑树：

三种插入删除方式的时间复杂度基本一致，在插入和删除的过程中，时间复杂度应该为树的高度，所以插入一个节点的时间复杂度依旧为 $O(\log_2 n)$，插入 n 个节点的时间复杂度也为 $O(\log_2 n!)$

【附录】

普通二叉树代码：

Binarytree.h

```
#pragma once
#include<utility>
#include<iostream>
using namespace std;
struct node
{
    node* leftchild;
    node* rightchild;
    node* fathernode;
    int element;
    node() { element = 0; leftchild = rightchild = NULL; }
    node(int theElement, node*theleft, node*theright,node*thefathernode)
    {
        element = theElement;
        leftchild = theleft;
        rightchild = theright;
        fathernode = thefathernode;
    }
};
class binarytree :public node
{
public:
    binarytree();
    ~binarytree();
    void erase();
    node*root;
    pair<node*, int> search(binarytree*tree, int key);
    void insertnode(pair<node*, int> result, int operating);
    void deletenode(pair<node*, int> result, int operating);
};
```

Binarytree.cpp

```
#include "binarytree.h"
#define NULL 0
using namespace std;


binarytree::binarytree()
{
    element = 0;
    leftchild = NULL;
    rightchild = NULL;
}


binarytree::~binarytree()
{
    erase();
}


void binarytree::erase(){ root = NULL; element = 0; }


pair<node*, int> binarytree::search(binarytree*tree,int key)
{
    bool found = 0;
    node*lastnode=NULL;
    node*p = tree->root;
    while (p != NULL&&found!=1)
    {
        lastnode = p;
        if (key < p->element)
            p = p->leftchild;
        else if (key>p->element)
            p = p->rightchild;
        else
            found = 1;
    }
    if (found == 0)
        p = lastnode;
    pair<node*, int> a = make_pair(p, found);
    return a;
}


void binarytree::insertnode(pair<node*, int> result, int operating)
{
    if (result.second == 0)
    {
        if (result.first->element==0)
        {
            result.first->element = operating;
        }
        if (result.first->element > operating)
        {
            result.first->leftchild = new node(operating, NULL, NULL,result.first);
        }
        else if (result.first->element < operating)
        {
            result.first->rightchild = new node(operating, NULL, NULL,result.first);
        }
    //  else
        //  throw"search error";
    }
    else if (result.second == 1)
        return;
    else
        throw"前面有错误";
}
```

```
void binarytree::deletenode(pair<node*, int> result, int operating)
{
    bool isroot=0;
    bool isnull = 0;
    node* temp=NULL;//用于交换两个指针的临时变量
    if (result.second == 0)
    {
        //cout << "inexistent element";
        return;
    }
    else
    {
        if (result.first->leftchild == NULL || result.first->rightchild == NULL)
        {
            if (result.first->leftchild == NULL&&result.first->rightchild ==
NULL&&result.first!=root)
            {
                if (result.first->fathernode->leftchild == result.first)
                    result.first->fathernode->leftchild = NULL;
                else
                    result.first->fathernode->rightchild = NULL;
                delete result.first;
            }
            else if (result.first->leftchild == NULL)//其左子节点为零
            {
                if (result.first->fathernode == NULL||result.first==root)
                {
                    if (result.first->rightchild != NULL)
                    {
                        result.first->element = result.first->rightchild->element;
                        temp = result.first->rightchild;
                        if(temp->rightchild!=NULL)
                            temp->rightchild->fathernode = temp->fathernode;
                        if(temp->leftchild!=NULL)
                            temp->leftchild->fathernode = temp->fathernode;
                        result.first->rightchild =
result.first->rightchild->rightchild;
                        result.first->leftchild = temp->leftchild;
                        delete temp;
                        isroot = 1;
                    }

                }
                else if (result.first->fathernode->leftchild == result.first)
                {
                    result.first->fathernode->leftchild = result.first->rightchild;
                    result.first->rightchild->fathernode = result.first->fathernode;
                    delete result.first;
                }
                else
                {
                    result.first->fathernode->rightchild = result.first->rightchild;
                    result.first->rightchild->fathernode = result.first->fathernode;
                    delete result.first;
                }
            }
            else
            {
                if (result.first->fathernode == NULL || result.first == root)
                {
                    if (result.first->leftchild != NULL)
                    {
                        result.first->element = result.first->leftchild->element;
                        temp = result.first->leftchild;
                        if(temp->leftchild!=NULL)
                            temp->leftchild->fathernode = temp->fathernode;
                        if (temp->rightchild != NULL)
                            temp->rightchild->fathernode = temp->fathernode;
                        result.first->rightchild =
result.first->leftchild->rightchild;
                        result.first->leftchild = temp->leftchild;
                        delete temp;
                        isroot = 1;
                    }

                }
                else if (result.first->fathernode->leftchild == result.first)
                {
                    result.first->fathernode->leftchild = result.first->leftchild;
                    result.first->leftchild->fathernode = result.first->fathernode;
                    delete result.first;
                }
                else
                {
                    result.first->fathernode->rightchild = result.first->leftchild;
                    result.first->leftchild->fathernode = result.first->fathernode;
                    delete result.first;
                }
            }
            if (isroot == 0)
                result.first = NULL;
        }
        else
        {
            temp = result.first;
            result.first = result.first->leftchild;
            if (result.first->rightchild == NULL)
                isnull = 1;
            while(result.first->rightchild!=NULL)
            {
                result.first = result.first->rightchild;
            }
            if (result.first->leftchild == NULL)
            {
                temp->element = result.first->element;
                temp = result.first->fathernode;
                delete result.first;
                if(isnull!=0)
                    temp->leftchild = NULL;
                else
                    temp->rightchild = NULL;
            }
            else
            {
                temp->element = result.first->element;
                if(isnull==0)
                    result.first->fathernode->rightchild = result.first->leftchild;
                else
                    result.first->fathernode->leftchild = result.first->leftchild;
                result.first->leftchild->fathernode = result.first->fathernode;
                temp = result.first->fathernode;
                delete result.first;
                result.first = NULL;
            }
        }
    }
}
```

源.cpp

```
#include<iostream>
#include<fstream>
#include<time.h>
```

```
#include<cstdlib>
#include"binarytree.h"
using namespace std;
int main()
{
    ofstream output;
    output.open("output.txt", ofstream::out);


    int n = 1000;
    int a[100000];
    srand((unsigned)time(0));
    for (int i = 0; i <= 100000 - 1; ++i) a[i] = i;

    while(n<100001)
    {
        node thenode;
        binarytree thetree;
        binarytree* tree = &thetree;
        tree->root = &thenode;
        pair<node*, int> result;
        for (int i = 100000 - 1; i >= 1; --i) swap(a[i], a[rand() % i]);
        clock_t start_time = clock();
        {
            for (int i = 0; i < n; i++)
            {
                result = thetree.search(tree, a[i]);
                thetree.insertnode(result, a[i]);
            }
            for (int i = 0; i < n ;i++)
            {
                result = thetree.search(tree, a[i]);
                thetree.deletenode(result, a[i]);
            }
        }
        clock_t end_time = clock();
        output << static_cast<double>(end_time - start_time) / CLOCKS_PER_SEC * 1000
<<endl;
        n = n + 200;
    }
}
```

AVL 树代码

AVLtree.h

```
#pragma once
```

```
#include<utility>
#include<iostream>
using namespace std;
struct node
{
    struct node*leftchild;
    struct node*rightchild;
    struct node*fathernode;
    int element;
    int balance;
    node(int theElement,int theBalance, node* theLeft, node* theRight, node*
theFathernode)
    {
        element = theElement;
        balance = theBalance;
        leftchild = theLeft;
        rightchild = theRight;
        fathernode = theFathernode;
    }
    node() { element = 0; leftchild = rightchild =fathernode = NULL; }
};
class AVLtree:public node
{
public:
    AVLtree();
    ~AVLtree();
    node* root;
    pair<node*, int> search(node*tree,int key);
    node* insertnode(pair<node*, int> result, int operating);
    void deletenode(pair<node*, int> result, int operating);
    void adjust(node*thenode,node*imbalance,AVLtree*tree);
    node* findimbalance(node* newnode);
};
```

AVLtree.cpp

```
#include "AVLtree.h"

void turnleft(node*imbalance,AVLtree*tree)//左旋
{
    node*temp;
    if (imbalance->rightchild == NULL)
    {
        if (imbalance->fathernode != NULL)
        {
            if (imbalance == imbalance->fathernode->leftchild)
                imbalance->fathernode->leftchild = imbalance->leftchild;
            else
                imbalance->fathernode->rightchild = imbalance->leftchild;
        }
        else
        {
            tree->root = imbalance->leftchild;
        }
        imbalance->leftchild->fathernode = imbalance->fathernode;
        imbalance->leftchild->balance = 0;
        imbalance->leftchild->rightchild = imbalance;
        imbalance->fathernode = imbalance->leftchild;
        imbalance->leftchild = NULL;
    }
    else
    {
        if (imbalance->fathernode != NULL)
        {
            if (imbalance == imbalance->fathernode->leftchild)
                imbalance->fathernode->leftchild = imbalance->leftchild;
            else
                imbalance->fathernode->rightchild = imbalance->leftchild;
        }
        else
            tree->root = imbalance->leftchild;
        temp = imbalance->leftchild->rightchild;
        imbalance->leftchild->fathernode = imbalance->fathernode;
        imbalance->leftchild->balance = 0;
        imbalance->leftchild->rightchild = imbalance;
        imbalance->fathernode = imbalance->leftchild;
        imbalance->leftchild = temp;
        if(imbalance->leftchild!=NULL)
            imbalance->leftchild->fathernode = imbalance;
    }
    imbalance->balance = 0;
}
void turnright( node*imbalance, AVLtree*tree)//右旋
{
    node*temp;
    if (imbalance->leftchild == NULL)
    {
        if (imbalance->fathernode != NULL)
        {
            if (imbalance == imbalance->fathernode->leftchild)
                imbalance->fathernode->leftchild = imbalance->rightchild;
            else
                imbalance->fathernode->rightchild = imbalance->rightchild;
        }
        else
        {
            tree->root = imbalance->rightchild;
        }
        imbalance->rightchild->fathernode = imbalance->fathernode;
        imbalance->rightchild->balance = 0;
        imbalance->rightchild->leftchild = imbalance;
        imbalance->fathernode = imbalance->rightchild;
        imbalance->rightchild = NULL;
    }
    else
    {
        if (imbalance->fathernode != NULL)
        {
            if (imbalance == imbalance->fathernode->leftchild)
                imbalance->fathernode->leftchild = imbalance->rightchild;
            else
                imbalance->fathernode->rightchild = imbalance->rightchild;
        }
        else
            tree->root = imbalance->rightchild;
        temp = imbalance->rightchild->leftchild;
        imbalance->rightchild->fathernode = imbalance->fathernode;
        imbalance->rightchild->balance = 0;
        imbalance->rightchild->leftchild = imbalance;
        imbalance->fathernode = imbalance->rightchild;
        imbalance->rightchild = temp;
        if(imbalance->rightchild!=NULL)
            imbalance->rightchild->fathernode = imbalance;
    }
    imbalance->balance = 0;
}
void turnrl(node*imbalance, AVLtree*tree)//右左旋,在这个函数后要调用左旋或者右旋函数,并
且调用函数时不能直接用thenode
{
    imbalance->leftchild->rightchild->balance = -1;
    if (imbalance->leftchild->leftchild != NULL)
        imbalance->leftchild->balance = -1;
    else
```

```
		imbalance->leftchild->balance = 0;
	node*temp;
	temp = imbalance->leftchild->rightchild->leftchild;
	imbalance->leftchild->fathernode = imbalance->leftchild->rightchild;
	imbalance->leftchild->rightchild->fathernode = imbalance;
	imbalance->leftchild->rightchild->leftchild = imbalance->leftchild;
	imbalance->leftchild = imbalance->leftchild->rightchild;
	imbalance->leftchild->leftchild->rightchild = temp;
}
void turnlr(node*imbalance, AVLtree*tree)//左右旋，在这个函数后要调用左旋或者右旋函数
{
	imbalance->rightchild->leftchild->balance = 1;
	if (imbalance->rightchild->rightchild != NULL)
		imbalance->rightchild->balance = 1;
	else
		imbalance->rightchild->balance = 0;
	node* temp;
	temp = imbalance->rightchild->leftchild->rightchild;
	imbalance->rightchild->fathernode = imbalance->rightchild->leftchild;
	imbalance->rightchild->leftchild->fathernode = imbalance;
	imbalance->rightchild->leftchild->rightchild = imbalance->rightchild;
	imbalance->rightchild = imbalance->rightchild->leftchild;
	imbalance->rightchild->rightchild->leftchild = temp;
}
void changeimbalance(node*thenode,node*thenode2)//改值放在插入后，在这里thenode的值会改
变，可能要加一个参量
{
	int temp = thenode2->element;
	thenode = thenode->fathernode;
	if (thenode->element>temp)
	{
		thenode->balance --;
		if (thenode->balance!=-2&&thenode->fathernode!=NULL)
			changeimbalance(thenode,thenode2);
	}
	else
	{
		thenode->balance ++;
		if (thenode->balance!=2&&thenode->fathernode!=NULL)
			changeimbalance(thenode,thenode2);
	}
}
```

```
AVLtree::AVLtree()
{
    element = 0;
    leftchild = rightchild = NULL;
}


AVLtree::~AVLtree()
{
}

pair<node*, int> AVLtree::search(node * treeroot, int key)
{
    node* operating;
    node* lastnode=NULL;
    bool found=0;
    operating = treeroot;
    while (operating != NULL&&found==0)//这里的在3的位置operating->rightchild=operating
    {
        lastnode = operating;
        if (key > operating->element)
            operating = operating->rightchild;
        else if (key < operating->element)
            operating = operating->leftchild;
        else
            found = 1;
    }
    if (found == 0)
        operating = lastnode;
    pair<node*, int> a = make_pair(operating, found);
    return a;
}

node* AVLtree::insertnode(pair<node*, int> result, int operating)
{
    node*temp = NULL;
    if (result.second == 0)
    {
        if (result.first->element == 0)
        {
            result.first->element = operating;
            result.first->balance = 0;
            return result.first;
        }
        if (result.first->element > operating)
        {
            result.first->leftchild = new node(operating,0, NULL, NULL, result.first);
            return result.first->leftchild;
        }
        else if (result.first->element < operating)
        {
            result.first->rightchild = new node(operating,0, NULL, NULL,
result.first);
            return result.first->rightchild;
        }
        //  else
        //  throw"search error";
    }
    else if (result.second == 1)
        return NULL;
    else
        throw"前面有错误";
}

void AVLtree::deletenode(pair<node*, int> result, int operating)
{
    bool isroot = 0;
    bool isnull = 0;
    node* temp = NULL;//用于交换两个指针的临时变量
    if (result.second == 0)
    {
        //cout << "inexistent element";
        return;
    }
    else
    {
        if (result.first->leftchild == NULL || result.first->rightchild == NULL)
        {
            if (result.first->leftchild == NULL&&result.first->rightchild ==
NULL&&result.first != root)
            {
                if (result.first->fathernode->leftchild == result.first)
                    result.first->fathernode->leftchild = NULL;
                else
                    result.first->fathernode->rightchild = NULL;
                delete result.first;
            }
            else if (result.first->leftchild == NULL)//其左子节点为零
            {
                if (result.first->fathernode == NULL || result.first == root)
                {
                    if (result.first->rightchild != NULL)
                    {
                        result.first->element = result.first->rightchild->element;
                        temp = result.first->rightchild;
                        if (temp->rightchild != NULL)
                            temp->rightchild->fathernode = temp->fathernode;
                        if (temp->leftchild != NULL)
                            temp->leftchild->fathernode = temp->fathernode;
                        result.first->rightchild =
result.first->rightchild->rightchild;
                        result.first->leftchild = temp->leftchild;
                        delete temp;
                        isroot = 1;
                    }

                }
                else if (result.first->fathernode->leftchild == result.first)
                {
                    result.first->fathernode->leftchild = result.first->rightchild;
                    result.first->rightchild->fathernode = result.first->fathernode;
                    delete result.first;
                }
                else
                {
                    result.first->fathernode->rightchild = result.first->rightchild;
                    result.first->rightchild->fathernode = result.first->fathernode;
                    delete result.first;
                }
            }
            else
            {
                if (result.first->fathernode == NULL || result.first == root)
                {
                    if (result.first->leftchild != NULL)
                    {
                        result.first->element = result.first->leftchild->element;
                        temp = result.first->leftchild;
                        if (temp->leftchild != NULL)
                            temp->leftchild->fathernode = temp->fathernode;
                        if (temp->rightchild != NULL)
                            temp->rightchild->fathernode = temp->fathernode;
                        result.first->rightchild =
result.first->leftchild->rightchild;
                        result.first->leftchild = temp->leftchild;
                        delete temp;
                        isroot = 1;
                    }

                }
                else if (result.first->fathernode->leftchild == result.first)
                {
                    result.first->fathernode->leftchild = result.first->leftchild;
                    result.first->leftchild->fathernode = result.first->fathernode;
                    delete result.first;
                }
                else
                {
                    result.first->fathernode->rightchild = result.first->leftchild;
                    result.first->leftchild->fathernode = result.first->fathernode;
                    delete result.first;
                }
            }
            if (isroot == 0)
                result.first = NULL;
        }
        else
        {
            temp = result.first;
            result.first = result.first->leftchild;
            if (result.first->rightchild == NULL)
                isnull = 1;
            while (result.first->rightchild != NULL)
            {
                result.first = result.first->rightchild;
            }
            if (result.first->leftchild == NULL)
            {
                temp->element = result.first->element;
                temp = result.first->fathernode;
                delete result.first;
                if (isnull != 0)
                    temp->leftchild = NULL;
                else
                    temp->rightchild = NULL;
            }
            else
            {
                temp->element = result.first->element;
                if (isnull == 0)
                    result.first->fathernode->rightchild = result.first->leftchild;
                else
                    result.first->fathernode->leftchild = result.first->leftchild;
                result.first->leftchild->fathernode = result.first->fathernode;
                temp = result.first->fathernode;
                delete result.first;
                result.first = NULL;
                /*if (isnull != 0)
                temp->leftchild = NULL;
                else
                temp->rightchild = NULL;*/
            }
        }
    }
}

void AVLtree::adjust(node*thenode,node*imbalance,AVLtree*tree)
{
    if (imbalance == NULL)
        return;

    //如果左子树不平衡
    if (imbalance->balance == -2)
    {
        if (imbalance->rightchild == NULL)
        {
            if (thenode == thenode->fathernode->leftchild)
                turnleft(imbalance, tree);
            else
            {
                turnrl(imbalance, tree);
                turnleft(imbalance, tree);
            }
        }
        else
        {
            if (thenode->fathernode == thenode->fathernode->fathernode->leftchild)
            {
                if(thenode == thenode->fathernode->rightchild)
                {
                    thenode->fathernode->fathernode->leftchild = thenode;
                    thenode->leftchild = thenode->fathernode;
                    thenode->fathernode = thenode->fathernode->fathernode;
                    thenode->leftchild->fathernode = thenode;
                    thenode->leftchild->rightchild = NULL;
                    turnleft(imbalance, tree);
                    thenode->balance = -1;
                    imbalance->leftchild->balance = 0;//
                }
                else
                    turnleft(imbalance, tree);

            }
            else
            {
                if (thenode == thenode->fathernode->leftchild)
                {
                    thenode->balance = 1;
                    thenode->fathernode->balance = 0;
                    thenode->fathernode->fathernode->rightchild = thenode;
                    thenode->rightchild = thenode->fathernode;
                    thenode->fathernode = thenode->fathernode->fathernode;
                    thenode->rightchild->fathernode = thenode;
                    thenode->rightchild->leftchild = NULL;
                    turnrl(imbalance, tree);
                    turnleft(imbalance, tree);
                }
                else
                {
                    turnrl(imbalance, tree);
                    turnleft(imbalance, tree);
                }
            }
        }
    }
    else
    {
        if (imbalance->leftchild == NULL)
        {
            if (thenode == thenode->fathernode->rightchild)
                turnright(imbalance, tree);
            else
            {
                turnlr(imbalance, tree);
                turnright(imbalance, tree);
            }
        }
        else
        {
            if (thenode->fathernode == thenode->fathernode->fathernode->leftchild)
            {
                if (thenode == thenode->fathernode->rightchild)
                {
                    thenode->balance = -1;
                    thenode->fathernode->balance = 0;
                    thenode->fathernode->fathernode->leftchild = thenode;
                    thenode->leftchild = thenode->fathernode;
                    thenode->fathernode = thenode->fathernode->fathernode;
                    thenode->leftchild->fathernode = thenode;
                    thenode->leftchild->rightchild = NULL;
                    turnlr(imbalance, tree);
                    turnright(imbalance, tree);
                }
                else
                {
                    turnlr(imbalance, tree);
                    turnright(imbalance, tree);//thenode没有父指针？？？
                }

            }
            else
            {
                if (thenode == thenode->fathernode->leftchild)
                {
                    thenode->balance = 1;
                    thenode->fathernode->balance = 0;
                    thenode->fathernode->fathernode->rightchild = thenode;
                    thenode->rightchild = thenode->fathernode;
                    thenode->fathernode = thenode->fathernode->fathernode;
                    thenode->rightchild->fathernode = thenode;
                    thenode->rightchild->leftchild = NULL;
                    turnright(imbalance, tree);
                }
                else
                    turnright(imbalance, tree);
            }
        }
    }
```

```
}

node * AVLtree::findimbalance(node * newnode)
{
    node* temp;
    node* imbalance;
    if (newnode->fathernode == NULL)
    {
        imbalance = NULL;
        return imbalance;
    }
    else
    {
        changeimbalance(newnode,newnode);
        while ((newnode->balance !=2&&newnode->balance!=-
2)&&(newnode->fathernode!=NULL))
        {
            newnode = newnode->fathernode;
        }
        if (newnode->balance == 2 || newnode->balance == -2)
            imbalance = newnode;
        else
            imbalance = NULL;
        return imbalance;
    }//其实findimbalance和changeimbalance可以写成一个函数
}
```

源.cpp

```
#include<iostream>
#include<fstream>
#include<time.h>
#include<cstdlib>
#include"AVLtree.h"
using namespace std;
int main()
{
    ofstream output;
    output.open("output.txt", ofstream::out);


    int n = 1000;
    int a[100000];
    srand((unsigned)time(0));
//  for (int i = 0; i <= 100000 - 1; ++i) a[i] = i;
```

```
    while (n<100001)
    {
        node* newnode;
        node* imbalance;
        node thenode;
        AVLtree thetree;
        AVLtree* tree = &thetree;
        tree->root = &thenode;
        pair<node*, int> result;
        //for (int i = 100000 - 1; i >= 1; --i) swap(a[i], a[rand() % i]);
        clock_t start_time = clock();
        {
            for (int i = 1; i < n; i++)
            {
                result = thetree.search(tree->root, i);
                newnode = thetree.insertnode(result, i);
                imbalance = thetree.findimbalance(newnode);
                thetree.adjust(newnode, imbalance, tree);
            }
            for (int i = 0; i < n; i++)
            {
                result = thetree.search(tree->root, i);
                thetree.deletenode(result,i);
            }
        }
        clock_t end_time = clock();
        output << static_cast<double>(end_time - start_time) / CLOCKS_PER_SEC * 1000
<< endl;
        n = n + 200;
    }
```