哈尔滨工业大学计算机科学与技术学院

实验报告

课程名称：数据结构与算法课程

类型：必修

实验项目： 查找结构的实验比较

实验题目： BST查找结构与折半查找方法的实现与实验比较

实验日期： 2018.12.20

班级：1703002

学号：1170500913

姓名：熊健羽

|  |  |  |
| --- | --- | --- |
| 设计成绩 | 报告成绩 | 指导老师 |
|  |  | 张岩 |

1. **实验目的**
2. **实验要求及实验环境**
   1. 实验要求
   2. 实验环境
      1. 硬件环境

CPU：Intel(R) Core(TM) i5-7200U @ 2.50GHz (64位)

GPU：Intel(R) HD Graphics 620

Nvidia GeForce 940MX

物理内存：8.00GB

磁盘：1TB HDD

128GB SSD

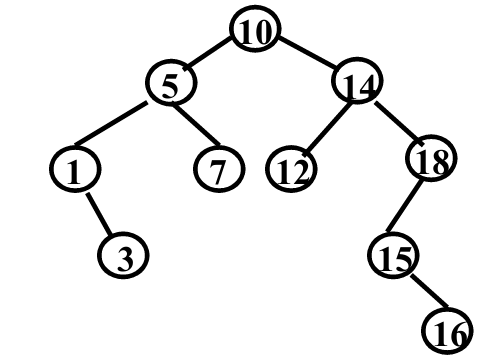
* + 1. 软件环境：

OS: Microsoft Windows 10 家庭中文版

编译环境：MinGW-W64

IDE：VS code、code::blocks

1. **设计思想**
   1. 逻辑设计
      1. 数据结构:
         1. BST树：



* + - 1. 线性数组：

A[1024]

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| index | 0 | 1 | 2 | 3 | … | 1020 | 1021 | 1022 | 1023 |
| value | 1 | 3 | 5 | 7 | … | 2041 | 2043 | 2045 | 2047 |

* + 1. 各个函数的功能及其调用关系：

void init\_array(int A[], int len);

功能：初始化0 – 2048之间的奇数数组

调用本地函数：无

void random\_array(int A[], int len);

功能：随机打乱数组

调用本地函数：无

int binSearch(int tar, int A[], int len);

功能：在顺序数组A[]中，二分查找关键字为tar的节点，返回其下标

调用本地函数：无

long binSearch\_count(int tar, int A[], int len);

功能：在顺序数组A[]中，二分查找关键字为tar的节点，对查找次数计数，返回查找次数

调用本地函数：无

double ASL\_binS\_success(int A[], int len);

功能：计算并返回数组A[]的平均成功查找长度

调用本地函数：binSearch\_count

double ASL\_binS\_fail(int A[], int len);

功能：计算并返回数组A[]的平均失败查找长度

调用本地函数：binSearch\_count

BST insert\_BST(BST tree, int x);

功能：向BST树中，插入值为x的节点

调用本地函数：无

BST createBST(int A[], int len);

功能：以A[]为数据输入次序，建立BST树

调用本地函数：insert\_BST

pos searchBST(BST tree, int tar);

功能：在BST中查找值为tar的数据，返回节点指针

调用本地函数：无

BST deleteNode(BST tree, pos p, pos q);

功能：删除p指向的节点，（q为p的双亲节点）

调用本地函数：无

BST delete\_BST(BST tree, int tar);

功能：删除树中第一个搜索到的值为tar的节点

调用本地函数：deleteNode

int searchBST\_count\_suc(BST tree, int tar);

功能：查找树中存在的，值为tar的节点，并返回查找的次数

调用本地函数：无

double ASL\_BST\_success(BST tree, int A[], int len);

功能：计算并返回BST树的平均成功查找长度

调用本地函数：searchBST\_count\_suc

long searchBST\_count\_fail(BST tree, int n);

功能：遍历树的所有外节点，返回比较次数之和

调用本地函数：searchBST\_count\_fail（递归）

double ASL\_BST\_fail(BST tree, int len);

功能：计算并返回BST树的平均失败查找长度

条用本地函数：searchBST\_count\_fail

void traverseBST(BST tree, int A[]);

功能：中序遍历BST树，并把遍历序列按顺序放入A[]数组中

调用本地函数：traverseBST（递归）

* 1. 物理设计

void init\_array(int A[], int len);

实现思路：

循环填入0 – 2048之间的奇数即可

void random\_array(int A[], int len);

实现思路：

在第1 – 1024个数据中随机挑选一个数据，与第1024个数据交换。再在1 – 1023个数据中随机挑选一个，与第1023个数据交换……以此类推。

int binSearch(int tar, int A[], int len);

实现思路：

二分查找。在有序表中，取中间记录作为比较对象，若给定值与中间记录的关键码相等，则查找成功；若给定值小于中间记录的关键码，则在中间记录的左半区继续查找；若给定值大于中间记录的关键码，则在中间记录的右半区继续查找。不断重复上述过程，直到查找成功，或所查找的区域无记录，查找失败。

long binSearch\_count(int tar, int A[], int len);

实现思路：在二分查找算法中添加一个计数数器。

double ASL\_binS\_success(int A[], int len);

实现思路：遍历每个元素，累加每次成功的查找长度，最后除以元素个数len即可。

double ASL\_binS\_fail(int A[], int len);

实现思路：遍历每个偶数元素，累加每次失败的查找长度，最后除以元素个数len + 1即可。

BST insert\_BST(BST tree, int x);

实现思路：

从根节点开始，将x与节点关键字比较，向左/右子树移动，直到外节点，新建一个节点，插入作为叶节点。

BST createBST(int A[], int len);

实现思路：多次调用insert\_BST函数即可。

pos searchBST(BST tree, int tar);

实现思路：将x与节点关键字比较，向左/右子树移动。直到找到，返回指针即可。

BST deleteNode(BST tree, pos p, pos q);

实现思路：

删除某结点，并保持二叉排序树特性，分三种情况处理：

1）如果删除的是叶结点，则直接删除；

2）如果删除的结点只有一株左子树或右子树，则直接继承：将该子树移到被删结点位置；

3）如果删除的结点有两株子树，则用继承结点代替被删结点，这相当于删除继承结点——按1) 或2) 处理继承结点。

BST delete\_BST(BST tree, int tar);

实现思路：搜索值为tar的节点，并保留该节点的父节点。调用deleteNode函数。

int searchBST\_count\_suc(BST tree, int tar);

实现思路：搜索过程中，每次比较，加入一个计数器即可。

double ASL\_BST\_success(BST tree, int A[], int len);

实现思路：调用searchBST\_count\_suc搜索tree中的每一个元素，累计搜索次数，除以总结点数len即可。

long searchBST\_count\_fail(BST tree, int n);

实现思路：递归遍历树的所有外节点，递归终止条件为tree = NULL，n为到达该节点时，已经比较的次数。tree不为外节点时，返回searchBST\_count\_fail(tree->lchild, n + 1) + searchBST\_count\_fail(tree->rchild, n + 1)

如此，可累加搜索所有外节点的比较次数。

double ASL\_BST\_fail(BST tree, int len);

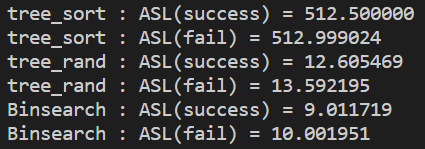
实现思路：以searchBST\_count\_fail(tree, 0)为初始条件调用上面的递归函数。意为：访问根节点时，比较了0次。如此得到n+1次搜索的比较次数之和，除以n+1即可。

void traverseBST(BST tree, int A[]);

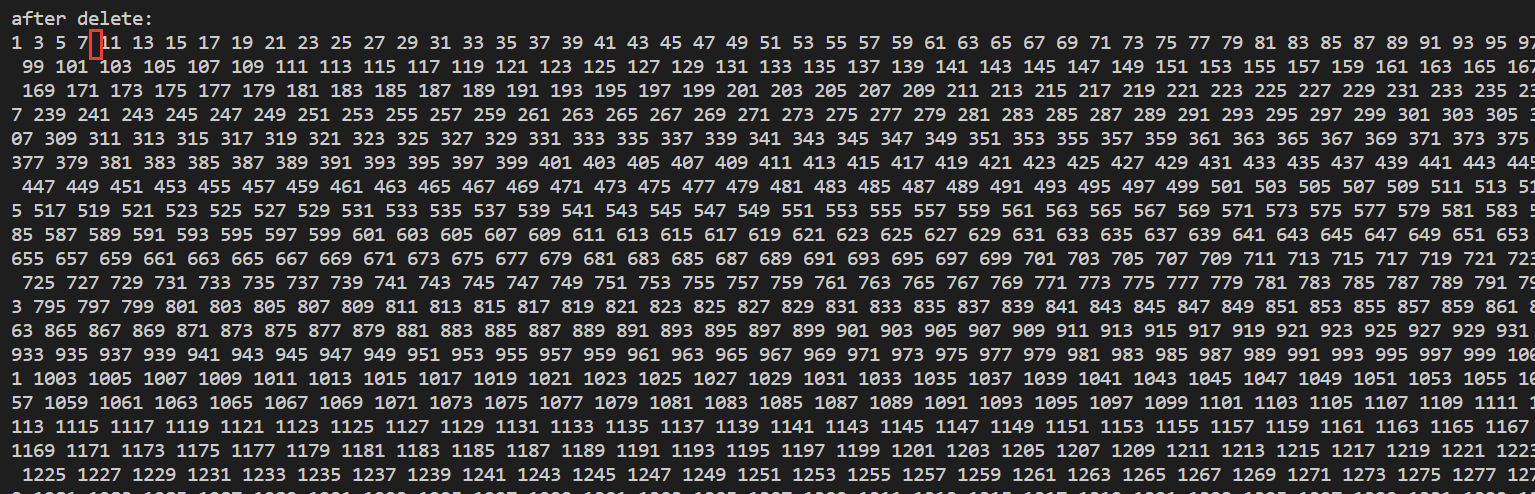
函数思路：递归中序遍历即可。

1. **测试结果**

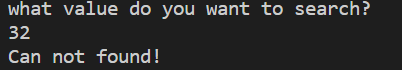
平均查找长度：

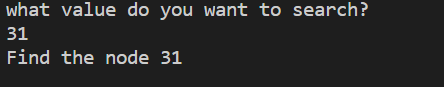


删除节点：（删除 9）



搜索节点：





1. **经验体会与不足**

经验：

平均性能而言，BST 的查找与折半查找差不多。就维护表的有序性而言，二叉查找树更有效。

BST树的形态与数据输入顺序有很大关系，有序序列输入时，BST退化成一个线性表，查找复杂度变低。防止的方法：

1.随机输入各结点

2.在建立、插入和删除各结点过程中平衡相关结点的左、右子树高度，如AVL树

不足：

BST删除算法写的比较繁杂，没有使用递归来简化算法。

1. **附录：源代码**

#include <stdio.h>

#include <stdlib.h>

#include <math.h>

#include <string.h>

#include <stdbool.h>

#include <time.h>

#define LEN 1024

**typedef** **struct** st\_BST

{

**int** key;

**struct** st\_BST \*lchild;

**struct** st\_BST \*rchild;

} BST\_NODE;

**typedef** BST\_NODE \*BST;

**typedef** BST\_NODE \*pos;

**void** init\_array(**int** A[], **int** len) *// init*

{

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

A[i] = 2 \* i + 1;

}

**void** random\_array(**int** A[], **int** len)

{

srand(time(NULL));

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

{

**int** index = rand() % i;

**int** temp = A[i];

A[i] = A[index];

A[index] = temp;

}

}

**int** binSearch(**int** tar, **int** A[], **int** len)

{

**int** down = 0;

**int** up = len - 1;

**int** mid;

**while** (up >= down)

{

mid = (up + down) / 2; *// choose the middle one*

**if** (A[mid] == tar)

**return** mid;

**else** **if** (A[mid] > tar)

up = mid - 1;

**else**

down = mid + 1;

}

**return** -1; *// not found*

}

**long** binSearch\_count(**int** tar, **int** A[], **int** len)

{

**int** mid;

**int** counter = 0;

**int** down = 0, up = len - 1;

**while** (down <= up)

{

mid = (down + up) / 2;

counter++;

**if** (A[mid] == tar)

**return** counter;

**else** **if** (A[mid] > tar)

up = mid - 1;

**else**

down = mid + 1;

}

**return** counter;

}

**double** ASL\_binS\_success(**int** A[], **int** len)

{

**long** res = 0;

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

{

res += binSearch\_count(A[i], A, len);

}

**return** (**double**)res / len;

}

**double** ASL\_binS\_fail(**int** A[], **int** len)

{

**long** res = 0;

res = binSearch\_count(0, A, len);

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

{

res += binSearch\_count(A[i] + 1, A, len);

}

**return** (**double**)res / (len + 1);

}

BST insert\_BST(BST tree, **int** x)

{

**if** (tree == NULL) *// tree is empty*

{

tree = (BST)malloc(**sizeof**(BST\_NODE));

tree->key = x;

tree->lchild = NULL;

tree->rchild = NULL;

}

**else**

{

pos p, q, r;

p = tree;

**while** (p != NULL)

{

q = p;

**if** (x > p->key) *// bigger*

p = p->rchild;

**else** **if** (x < p->key) *// smaller*

p = p->lchild;

**else** *// already exist*

{

printf("%d is already exist in the tree!**\n**");

**return** tree;

}

}

r = (BST)malloc(**sizeof**(BST\_NODE));

r->key = x;

r->lchild = NULL;

r->rchild = NULL; *// create new node*

**if** (x > q->key)

q->rchild = r;

**if** (x < q->key)

q->lchild = r; *// add it to the tree*

}

**return** tree;

}

BST createBST(**int** A[], **int** len)

{

BST tree = NULL;

**for** (**int** i = 0; i < len; i++) *// insert circularly*

{

tree = insert\_BST(tree, A[i]);

}

**return** tree;

}

pos searchBST(BST tree, **int** tar)

{

pos p = tree;

**while** (p != NULL) *// search until leave node*

{

**if** (p->key == tar) *// find it*

**return** p;

**else** **if** (tar > p->key) *// bigger*

p = p->rchild;

**else** *// smaller*

p = p->lchild;

}

**return** NULL;

}

BST deleteNode(BST tree, pos p, pos q)

{

bool isRoot = false;

**if** (p == tree) *//if delete root*

isRoot = true;

**if** (p->rchild == NULL && p->lchild == NULL) *//if p is a leaf*

{

**if** (isRoot)

tree = NULL;

**else** **if** (p == q->lchild)

q->lchild = NULL;

**else** **if** (p == q->rchild)

q->rchild = NULL;

free(p);

}

**else** **if** (p->rchild == NULL) *//rchild is NULL*

{

**if** (isRoot)

tree = p->lchild;

**else** **if** (p == q->lchild)

q->lchild = p->lchild;

**else** **if** (p == q->rchild)

q->rchild = p->lchild;

free(p);

}

**else** **if** (p->lchild == NULL) *// lchild is NULL*

{

**if** (isRoot)

tree = p->rchild;

**else** **if** (p == q->lchild)

q->lchild = p->rchild;

**else** **if** (p == q->rchild)

q->rchild = p->rchild;

free(p);

}

**else** *//if p has two children*

{

**if** (p->rchild->lchild != NULL) *//p's right - left is not NULL*

{

pos tp, tq;

tq = p->rchild;

tp = p->rchild->lchild;

**while** (tp->lchild != NULL)

{

tq = tp;

tp = tp->lchild;

} *// find the min*

p->key = tp->key;

tq->lchild = tp->rchild;

free(tp);

}

**else** *//p's right - left is NULL*

{

**if** (isRoot)

tree = p->rchild;

**else** **if** (p == q->lchild)

q->lchild = p->rchild;

**else** **if** (p == q->rchild)

q->rchild = p->rchild;

p->rchild->lchild = p->lchild;

free(p);

}

}

**return** tree;

}

BST delete\_BST(BST tree, **int** tar)

{

pos p = tree;

pos q = NULL;

**while** (p != NULL) *// search until leave node*

{

**if** (p->key == tar) *// find it*

**break**;

**else** **if** (tar > p->key) *// bigger*

{

q = p;

p = p->rchild;

}

**else** *// smaller*

{

q = p;

p = p->lchild;

}

}

**if** (p != NULL)

tree = deleteNode(tree, p, q);

**else**

printf("not find!**\n**");

**return** tree;

}

**int** searchBST\_count\_suc(BST tree, **int** tar) *// search for "tar", count the comparative times*

{

**int** counter = 0;

pos p = tree;

**while** (p)

{

counter++;

**if** (tar == p->key)

**return** counter;

**else** **if** (tar < p->key)

p = p->lchild;

**else**

p = p->rchild;

}

**return** -1; *// error*

}

**double** ASL\_BST\_success(BST tree, **int** A[], **int** len) *// figure out the ASL when success*

{

**long** res = 0;

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

{

res += searchBST\_count\_suc(tree, A[i]);

}

**return** ((**double**)res / len);

}

**long** searchBST\_count\_fail(BST tree, **int** n) *// when arrive the "tree", already compare "n" times*

{

**if** (tree == NULL) *// "tree" is the failed node (NULL pointer), has compared n times*

**return** n;

**else** *// "tree" isn't the failed node, accumulate its children's comparative times*

**return** searchBST\_count\_fail(tree->lchild, n + 1) + searchBST\_count\_fail(tree->rchild, n + 1);

}

**double** ASL\_BST\_fail(BST tree, **int** len)

{

**long** res = 0;

res = searchBST\_count\_fail(tree, 0); *// when the "tree" is root, has already compared 0 times*

**return** ((**double**)res / (len + 1)); *// NULL nodes = ALL nodes + 1*

}

**int** count = 0;

**void** traverseBST(BST tree, **int** A[])

{

**if** (tree != NULL)

{

traverseBST(tree->lchild, A);

A[count++] = tree->key;

traverseBST(tree->rchild, A);

}

}

**int** main()

{

*/\*initialize the data\*/*

**int** rand\_A[LEN], sort\_A[LEN], out\_A[LEN], delete\_A[LEN];

init\_array(sort\_A, LEN);

memcpy(rand\_A, sort\_A, **sizeof**(**int**) \* LEN);

random\_array(rand\_A, LEN);

*/\*create the tree\*/*

BST tree\_rand = createBST(rand\_A, LEN);

BST tree\_sort = createBST(sort\_A, LEN);

*// pos res;*

*// int tar;*

*// scanf("%d", &tar);*

*// if (res = searchBST(tree\_sort, tar))*

*// printf("Find the node %d\n", res->key);*

*// else*

*// printf("Can not found!\n");*

*// printf("\n");*

printf("tree\_sort : ASL(success) = %lf**\n**", ASL\_BST\_success(tree\_sort, sort\_A, LEN));

printf("tree\_sort : ASL(fail) = %lf**\n**", ASL\_BST\_fail(tree\_sort, LEN));

printf("tree\_rand : ASL(success) = %lf**\n**", ASL\_BST\_success(tree\_rand, rand\_A, LEN));

printf("tree\_rand : ASL(fail) = %lf**\n**", ASL\_BST\_fail(tree\_rand, LEN));

count = 0;

traverseBST(tree\_rand, out\_A);

printf("Binsearch : ASL(success) = %lf**\n**", ASL\_binS\_success(out\_A, LEN));

printf("Binsearch : ASL(fail) = %lf**\n**", ASL\_binS\_fail(out\_A, LEN));

*/\*delete\*/*

tree\_rand = delete\_BST(tree\_rand, 5);

count = 0;

traverseBST(tree\_rand, delete\_A);

printf("**\n**after delete:**\n**");

**for** (**int** i = 0; i < LEN - 1; i++)

printf("%d ", delete\_A[i]);

printf("**\n**");

**return** 0;

}