山东大学 计算机科学与技术 学院

数据结构与算法 课程实验报告

学号：202000130198	姓名： 隋春雨	班级： 20.4
实验题目：搜索树		
实验学时：2	实验日期： 2021-12-9	

实验目的：

掌握二叉搜索树结构的定义、描述方法、操作实现。

软件开发环境：

CLION2020

1. 实验内容

1、题目描述：

创建带索引的二叉搜索树类。存储结构使用链表，提供操作:插入、删除、按名次删除、查找、按名次查找、升序输出所有元素。

输入输出格式：

输入：

输入第一行一个数字 m (m<=1000000)，表示有 m 个操作。

接下来 m 行，每一行有两个数字 a，b；

当输入的第一个数字 a 为 0 时，输入的第二个数字 b 表示向搜索树中插入 b；

当输入的第一个数字 a 为 1 时，输入的第二个数字 b 表示向搜索树中查找 b；

当输入的第一个数字 a 为 2 时，输入的第二个数字 b 表示向搜索树中删除 b；

当输入的第一个数字 a 为 3 时，输入的第二个数字 b 表示查找搜索树中名次为 b 的元素；

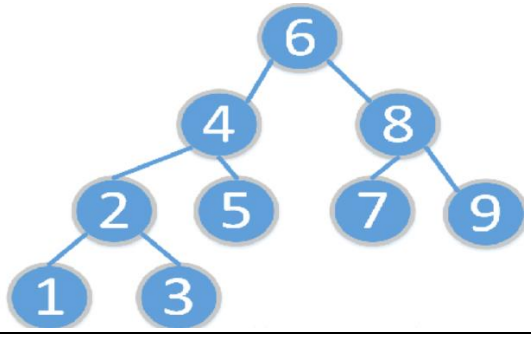
当输入的第一个数字 a 为 4 时，输入的第二个数字 b 表示删除搜索树中名次为 b 的元素；

输出：

对于输入中的每一种操作，输出执行操作的过程中依次比较的元素值的异或值。

2. 数据结构与算法描述 （整体思路描述，所需要的数据结构与算法）

(1) 数据结构：选择二叉搜索树。二叉搜索树的特征：任取一个结点，如果存在左孩子，则左孩子的值一定小于等于该结点。如果存在右孩子，那么右孩子的值一定大于等于该结点。如下图



(2) 算法：

(一)查找操作：类似于二分查找。其实二叉搜索树本质上也就是二分查找。我们将要查找的值和当前结点的值进行比较，如果相等，那么成功找到。如果不等，向左子树或者右子树继续查找。(这取决于大小关系)，如果最后发现当前指向的结点是空，那么就查找失败。代码如下：

```
template <class T>
int binarySearchTree<T>::get(int theIndex)
{//根据索引去查找节点
    binaryTreeNode<T>* p = root;
    int answer = 0;
    while (p != nullptr && p->leftSize != theIndex)
    { //检察p->leftSize
        answer ^= p->element;
        if (p->leftSize > theIndex) //index小，指向左子树
            p = p->leftChild;
        else if (p->leftSize < theIndex) //index大，指向右子树
        {
            theIndex = theIndex - (p->leftSize + 1);  //减去leftSize+1，书P348
            p = p->rightChild;
        }
    }
    if (!p)
        return 0;
    else
    {//找到匹配的元素
        answer ^= p->element;
        return answer;
    }
}
```

(二)插入操作：首先，我们需要判断一下，要插入的值是否在这棵树之中。如果在，那么我们就不插入了。如果不在，那么我们需要找到插入的位置。我们需要记录一下其父亲结点的位置，然后用一个指针去绑定他们俩。最后通过一下比较大小关系，决定是放在左子树还是右子树上。同时最后，需要更新一下 index 的值。代码如下：

```cpp
template <class T>
int binarySearchTree<T>::insert(const T& theElement)
{//插入元素值为theElement的元素
    binaryTreeNode<T>* p = root;
    binaryTreeNode<T>* pp = nullptr; //用于记录父节点
    int answer = 0;
    while (p != nullptr)
    {//检察元素p->element
        answer ^= p->element;
        pp = p;//p移到它的一个孩子节点
        if (p->element < theElement)
        {
            p = p->rightChild;
        }
        else if (p->element > theElement)
        {
            p = p->leftChild;
        }
        else if (p->element == theElement)
        {
            return 0;
        }
    }
```

```cpp
    //为theElement建立一个节点，然后与pp连接
    binaryTreeNode<T>* newNode = new binaryTreeNode<T>(theElement);
    if (pp != nullptr)   //树不空
    {
        if (theElement > pp->element)
            pp->rightChild = newNode;
        else if (theElement < pp->element)
            pp->leftChild = newNode;
    }
    else
    {
        root = newNode; //插入空树
    }
    treeSize++;
    //对节点名次进行计算
    p = root;
    while (p->element != theElement)
    {
        if (p->element < theElement)
        {
            p = p->rightChild;
        }
        else if (p->element > theElement)
```

(三)删除操作：对于删除操作，我们首先需要判断一下这棵树中是否有这个元素。如果没有，则返回。否则，我们需要删除这个元素，并且重新组织这棵树的结构。我们需要记录一下其父亲结点，然后分别考虑要删除的结点有几个孩子，如果有 0 个，也就是一个叶子结点，那么我们直接删除就可以了，将父亲结点的孩子指向了空。如果有 1 个那么我们直接让其代替被删除的结点在这棵树中的位置，并且更新一下 index 即可。如果有两个，我们选择右子树中最小的元素/左子树中最大的元素来替换其位置即可。代码如下：

```cpp
template <class T>
int binarySearchTree<T>::erase(const T& theElement)
{//删除元素值为theElement的元素
//删除操作中，如果当前元素有两个孩子，替换的为 右子树中最小的，
//如果只有一个孩子，直接用该孩子替换当前元素，如果没有孩子，直接删除
    binaryTreeNode<T>* p = root;
    binaryTreeNode<T>* pp = nullptr;
    int answer = 0;
    while (p != nullptr && p->element != theElement)
    {//p移到它的一个孩子节点
        answer ^= p->element;
        pp = p;
        if (p->element < theElement)
            p = p->rightChild;
        else if (p->element > theElement)
            p = p->leftChild;
    }
    if (p == nullptr)
        return 0;//不存在与关键值theElement匹配的元素

    //重新组织树结构
    answer ^= p->element;
    p = root;
    while (p!= nullptr && p->element != theElement)
    {
        if (p->element < theElement)
            p = p->rightChild;
        else if (p->element > theElement)
        {
            p->leftSize--;
            p = p->leftChild;
        }
    }
    //当p有两个孩子时的处理
    //转化为空或只有一个孩子
    //在p的右子树中寻找最小元素
    if (p->leftChild != nullptr && p->rightChild != nullptr)
    {
        binaryTreeNode<T>* s = p->rightChild;
        binaryTreeNode<T>* ps = p;
        while (s->leftChild != nullptr)
        {//移到最小的元素
            s->leftSize--;
```

```
        binaryTreeNode<T>* q = new binaryTreeNode<T>(s->element, p->leftChild, p->rightChi
        if (pp == nullptr)
            root = q;
        else if (p == pp->leftChild)
            pp->leftChild = q;
        else
            pp->rightChild = q;

        if (ps == p)  pp = q;
        else  pp = ps;

        delete p;
        p = s;
    }

    //p最多有一个孩子，把孩子指针存放在c
    binaryTreeNode<T>* c;
    if (p->leftChild != nullptr)
        c = p->leftChild;
    else
        c = p->rightChild;
    //删除p
    if (p == root)
        root = c;
```

3. 测试结果（测试输入，测试输出）

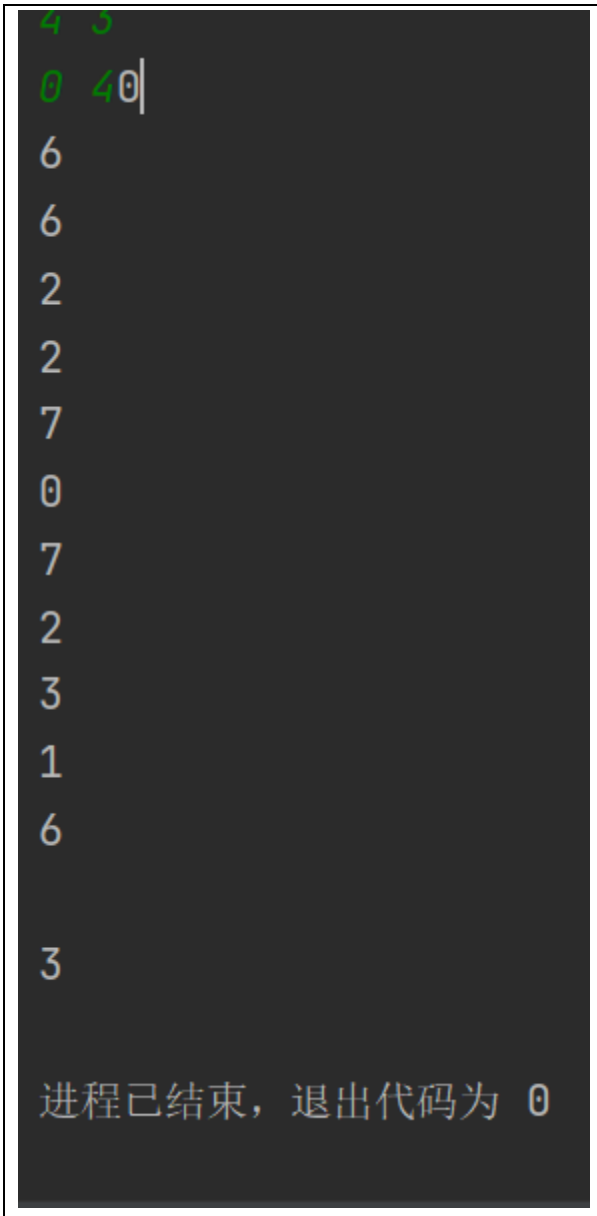
(1) a 题

输入：

输入

```
13
0 6
0 7
0 4
0 5
0 1
1 5
0 7
3 3
2 4
1 5
3 4
4 3
0 4
```

输出：



提交 OJ 的结果：

✓Accepted

#	Result	Score	Time	Memory
1	✓Accepted	10	0 ms	3496 KiB
2	✓Accepted	10	0 ms	3444 KiB
3	✓Accepted	10	0 ms	3404 KiB
4	✓Accepted	10	1 ms	3368 KiB
5	✓Accepted	10	1 ms	3428 KiB
6	✓Accepted	10	2 ms	3488 KiB
7	✓Accepted	10	2 ms	3564 KiB
8	✓Accepted	10	3 ms	3508 KiB
9	✓Accepted	10	4 ms	3564 KiB
10	✓Accepted	10	6 ms	3788 KiB

4. 分析与探讨（结果分析，若存在问题，探讨解决问题的途径）

(1) 在执行完函数体的时候，要更新 Index，因为我们删除了一个元素，需要更新它所在子树的 index 值，把大于它的结点的 index--。代码如下：

```
while (s->leftChild != nullptr)
{//移到最小的元素
    s->leftSize--;
    ps = s;
    s = s->leftChild;
}
```

(2) 我们在使用指针的时候，一定要防止短路。比如这段，一定要先判断其是否为空，再判断是否等于 index，否则会导致访问错误而 RE

```
while (p != nullptr && p->leftSize != theIndex)
{//p移到它的一个孩子节点
    answer ^= p->element;
    pp = p;
    if (p->leftSize > theIndex)
        p = p->leftChild;
    else if (p->leftSize < theIndex)
    {
        theIndex = theIndex - p->leftSize - 1;
        p = p->rightChild;
    }
}
```

(3) 我们在写完类内函数的时候，要考虑全面，比如本实验中的孩子结点的判断，需要分成 3 种情况：0 个、1 个、2 个，对于 0 个的我们直接删除，父节点孩子指针指向空，对于 1 个，我们直接代替。对于 2 个，我们寻找左子树中的最大元素/右子树中的最小元素来代替。同时需要注意的是，寻找最大/最小元素的时候，对于孩子指针的判断，需要特判一下左子树还是右子树，因为执行次数是不是 1 会导致结果的不同。

(4) 在提交 oj 的时候要把自己的测试删除，比如多输出了一个换行可能就会导致全部的判错。

(5) 对于 BOOL 数组的初始化，绝对不能想当然，每一次的定义都要对其进行初始化。在本次的实验中，因为 bool 数组没有初始化导致 debug 了很久。

```
bool* jud_push = new bool[_size + 1]();
```

(6) 对于下标从 1 开始的数组，要多动态分配一块内存。因为索引为 0 的地方我们是没有访问的。

```
int* height = new int[_size + 1];
```

(7) 对于一个只有两个私有成员的 struct，我们可以直接使用 pair 来存储。简化了我们的代码量。同时对于返回多个值的函数，我们只能使用引用来返回。这也是为什么兴起了 Go 语言的一个原因。

(8) 对于 height 等操作的计算，一定要特判是否合法。因为对于 root 结点来说，它的父节点是自身，不能直接增加。

```
if(node.first.element!=node.second.element)//非根结点
{
    height[node.first.element] = max(height[node.first.element], height[node.secor
}
s.pop();
```

(9) 我们在写循环条件判断的时候，对于短路情况的判断一定要慎重，我们对于指针的使用一定要先判断是否为空，在进行取值操作，如下：

```
40      while (currentNode != NULL &&//值不等于输入，且不越界，对于NULL值的判断要放到前边
41             currentNode->element.first != theKey)
42          currentNode = currentNode->next;
```

(10) 对于维护私有变量的时候，要特殊情况特殊判：比如说删除结点的时候，要考虑这个是不是头结点，如果是，那么更新私有变量。如下：

```
112     if (p != NULL && p->element.first == theKey)
113     {
114         if (tp == NULL) firstNode = p->next;//头结点特殊处理
115         else tp->next = p->next;
116         delete p;//删除
117         dSize--;
118     }
119 }
```

(10)要注意私有成员的更新，public 函数知道自己所处的状态都是靠着私有成员才知道的，如果没有及时更新，那数据就成了垃圾数据，没有任何意义。我在写实验的时候，也经历过没更新导致的 Bug，最终 debug 查出来，就是下面这个：

```
    // switch to newQueue and set theFront and theBack
    theFront = 2 * arrayLength - 1;//更新私有成员
    theBack = arrayLength - 2;    // queue size arrayLength - 1
    arrayLength *= 2;
    queue = newQueue;//指针赋值
}

// put theElement at the theBack of the queue
theBack = (theBack + 1) % arrayLength;
queue[theBack] = theElement;
```

5. 附录：实现源代码（本实验的全部源程序代码，程序风格清晰易理解，有充分的注释）

```
template <class T>
class binarySearchTree //二叉搜索树
{
private:
 binaryTreeNode<T>* root; //根
 int treeSize; //树中元素个数

public:
 binarySearchTree()
 {
  root = NULL;
  treeSize = 0;
 }
 int find(const T& theElement);
 int get(int theIndex);
 int insert(const T& theElement);
 int erase(const T& theElement);
 int deleteByIndex(int theIndex);

};
template <class T>
int binarySearchTree<T>::find(const T& theElement)
{//根据元素值去查找节点
 binaryTreeNode<T>* p = root;
 int answer = 0;
 while (p != NULL && p->element != theElement)
 {//检察元素 p->element
  answer ^= p->element;
```

```
29.   if (p->element > theElement)
30.    p = p->leftChild;
31.   else if (p->element < theElement)
32.    p = p->rightChild;
33.  }
34.  if (p == NULL)
35.   return 0;
36.  else
37.  {//找到匹配的元素
38.   answer ^= p->element;
39.   return answer;
40.  }
41. }
42.
43. template <class T>
44. int binarySearchTree<T>::get(int theIndex)
45. {//根据索引去查找节点
46.  binaryTreeNode<T>* p = root;
47.  int answer = 0;
48.  while (p != NULL && p->leftSize != theIndex)
49.  { //检察 p->leftSize
50.   answer ^= p->element;
51.   if (p->leftSize > theIndex) //index 小，指向左子树
52.    p = p->leftChild;
53.   else if (p->leftSize < theIndex) //index 大，指向右子树
54.   {
55.    theIndex = theIndex - (p->leftSize + 1); //减去 leftSize+1， 书P348
56.    p = p->rightChild;
57.   }
58.  }
59.  if (p == NULL)
60.   return 0;
61.  else
62.  {//找到匹配的元素
63.   answer ^= p->element;
64.   return answer;
65.  }
66. }
67. template <class T>
68. int binarySearchTree<T>::insert(const T& theElement)
69. {//插入元素值为 theElement 的元素
70.  binaryTreeNode<T>* p = root;
71.  binaryTreeNode<T>* pp = NULL; //用于记录父节点
72.  int answer = 0;
73.  while (p != NULL)
74.  {//检察元素 p->element
```

```
75.       answer ^= p->element;
76.       pp = p;//p 移到它的一个孩子节点
77.     if (p->element < theElement)
78.     {
79.      p = p->rightChild;
80.     }
81.     else if (p->element > theElement)
82.     {
83.      p = p->leftChild;
84.     }
85.     else if (p->element == theElement)
86.     {
87.      //如果是 pair 类型，这里就需要覆盖旧值
88.      return 0;
89.     }
90.   }
91.   //为 theElement 建立一个节点，然后与 pp 连接
92.   binaryTreeNode<T>* newNode = new binaryTreeNode<T>(theElement);
93.   if (pp != NULL)   //树不空
94.   {
95.    if (theElement > pp->element)
96.     pp->rightChild = newNode;
97.    else if (theElement < pp->element)
98.     pp->leftChild = newNode;
99.   }
100.      else
101.      {
102.       root = newNode; //插入空树
103.      }
104.      treeSize++;
105.      //对节点名次进行计算
106.      p = root;
107.      while (p->element != theElement)
108.      {
109.       if (p->element < theElement)
110.       {
111.        p = p->rightChild;
112.       }
113.       else if (p->element > theElement)
114.       {
115.        p->leftSize++;
116.        p = p->leftChild;
117.       }
118.      }
119.      return answer;
120.     }
```

```
121.
122.  template <class T>
123.  int binarySearchTree<T>::erase(const T& theElement)
124.  {//删除元素值为 theElement 的元素
125.  //删除操作中，如果当前元素有两个孩子，替换的为 右子树中最小的，
126.  //如果只有一个孩子，直接用该孩子替换当前元素，如果没有孩子，直接删除
127.   binaryTreeNode<T>* p = root;
128.   binaryTreeNode<T>* pp = NULL;
129.   int answer = 0;
130.   while (p != NULL && p->element != theElement)
131.   {//p 移到它的一个孩子节点
132.    answer ^= p->element;
133.    pp = p;
134.    if (p->element < theElement)
135.     p = p->rightChild;
136.    else if (p->element > theElement)
137.     p = p->leftChild;
138.   }
139.   if (p == NULL)
140.    return 0;//不存在与关键值 theElement 匹配的元素
141.
142.  //重新组织树结构
143.  answer ^= p->element;
144.  p = root;
145.   while (p!= NULL && p->element != theElement)
146.   {
147.   if (p->element < theElement)
148.    p = p->rightChild;
149.   else if (p->element > theElement)
150.   {
151.    p->leftSize--;
152.    p = p->leftChild;
153.   }
154.   }
155.     //当 p 有两个孩子时的处理
156.     //转化为空或只有一个孩子
157.  //在 p 的右子树中寻找最小元素
158.   if (p->leftChild != NULL && p->rightChild != NULL)
159.   {
160.    binaryTreeNode<T>* s = p->rightChild;
161.    binaryTreeNode<T>* ps = p;
162.    while (s->leftChild != NULL)
163.    {//移到最小的元素
164.     s->leftSize--;
165.     ps = s;
166.     s = s->leftChild;
```

```
167.       }
168.
169.       binaryTreeNode<T>* q = new binaryTreeNode<T>(s->element, p->leftChild, p->rightChi
       ld, p->leftSize);
170.       if (pp == NULL)
171.        root = q;
172.       else if (p == pp->leftChild)
173.            pp->leftChild = q;
174.            else
175.               pp->rightChild = q;
176.
177.       if (ps == p)  pp = q;
178.       else  pp = ps;
179.
180.       delete p;
181.       p = s;
182.      }
183.
184.        //p 最多有一个孩子，把孩子指针存放在 c
185.      binaryTreeNode<T>* c;
186.      if (p->leftChild != NULL)
187.       c = p->leftChild;
188.      else
189.       c = p->rightChild;
190.      //删除 p
191.      if (p == root)
192.       root = c;
193.      else
194.      {//p 是 pp 的左孩子还是右孩子
195.       if (p == pp->leftChild)
196.        pp->leftChild = c;
197.       else
198.        pp->rightChild = c;
199.      }
200.      treeSize--;
201.      delete p;
202.      return answer;
203.     }
204.
205.
206.     template <class T>
207.     int binarySearchTree<T>::deleteByIndex(int theIndex)
208.     {
209.      binaryTreeNode<T>* p = root;
210.      binaryTreeNode<T>* pp = NULL;
211.      int answer = 0;
```

```
212.
213.      while (p != NULL && p->leftSize != theIndex)
214.      {//p 移到它的一个孩子节点
215.       answer ^= p->element;
216.       pp = p;
217.       if (p->leftSize > theIndex)
218.        p = p->leftChild;
219.       else if (p->leftSize < theIndex)
220.       {
221.        theIndex = theIndex - p->leftSize - 1;
222.        p = p->rightChild;
223.       }
224.      }
225.      if (p == NULL)
226.       return 0;
227.
228.      //重新组织树结构
229.      answer ^= p->element;
230.      int theElement = p->element;//转换到用 element 去比较
231.      p = root;
232.      while (p!= NULL && p->element != theElement)
233.      {
234.       if (p->element < theElement)
235.        p = p->rightChild;
236.       else if (p->element > theElement)
237.       {
238.        p->leftSize--;
239.        p = p->leftChild;
240.       }
241.      }
242.        //当 p 有两个孩子时的处理
243.      if (p->leftChild != NULL && p->rightChild != NULL)
244.      {
245.       binaryTreeNode<T>* s = p->rightChild;
246.       binaryTreeNode<T>* ps = p;
247.       while (s->leftChild != NULL)
248.       {
249.        s->leftSize--;
250.        ps = s;
251.        s = s->leftChild;
252.       }
253.
254.       binaryTreeNode<T>* q = new binaryTreeNode<T>(s->element, p->leftChild, p->rightChi
      ld, p->leftSize);
255.       if (pp == NULL)
256.        root = q;
```

```
257.      else if (p == pp->leftChild)
258.          pp->leftChild = q;
259.          else
260.             pp->rightChild = q;
261.
262.      if (ps == p)  pp = q;
263.      else  pp = ps;
264.
265.      delete p;
266.      p = s;
267.     }
268.
269.       //p 最多有一个孩子，把孩子指针存放在 c
270.     binaryTreeNode<T>* c;
271.     if (p->leftChild != NULL)
272.      c = p->leftChild;
273.     else
274.      c = p->rightChild;
275.     //删除 p
276.     if (p == root)
277.      root = c;
278.     else
279.     {//p 是 pp 的左孩子还是有孩子
280.      if (p == pp->leftChild)
281.       pp->leftChild = c;
282.      else
283.       pp->rightChild = c;
284.     }
285.     treeSize--;
286.     delete p;
287.     return answer;
288.    }
289.
290.
291.
292.    int main()
293.    {
294.     binarySearchTree<int> BSTree;
295.
296.     int m, a, b;
297.     cin >> m;
298.     for (int i = 0; i < m; i++)
299.     {
300.      cin >> a >> b;
301.
302.      switch (a)
```

```
303.      {
304.      case 0:cout << BSTree.insert(b) << endl; break;
305.      case 1:cout << BSTree.find(b) << endl; break;
306.      case 2:cout << BSTree.erase(b) << endl; break;
307.      case 3:
308.       b = b - 1;//从 0 开始
309.       cout << BSTree.get(b) << endl;
310.       break;
311.      case 4:
312.          b = b - 1;//从 0 开始
313.       cout << BSTree.deleteByIndex(b) << endl;
314.       break;
315.      }
316.     }
317.
318.    }
```