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：202300130183	姓名： 宋浩宇	班级：23 级人工智能班
实验题目：2024 数据结构--数据/智能 实验 11 搜索树		
实验学时：2	实验日期：2024/11/27	

实验目的：

1. 掌握搜索树结构的定义与实现；
2. 掌握搜索树结构的使用。

软件开发工具：

1. visual studio code 2022（使用 C/C++、C/C++ Extension Pack、C/C++ Themes 插件）
2. mingw64 工具包

1. 实验内容

完成 2024 数据结构--数据/智能 实验 11 搜索树 A 二叉搜索树

1. 创建二叉搜索树类
2. 实现操作:插入、删除、按名次删除、查找、按名次查找、升序输出所有元素

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）

本题使用的数据结构为：二叉搜索树。具体来说，我们使用的是在左子树存储较小值，在右子树存储较大值的二叉搜索树。我们将在下文具体描述这个二叉搜索树的插入、删除、按名次删除、查找、按名次查找、升序输出所有元素操作的实现方式。另外，题目要求我们输出每次进行操作时进行比较的节点的异或值，因此我们在描述的时候也会着重描述一下比较的顺序。另外，因为这个题还要求我们实现按名次查找、删除相关的操作，需要用到左子树节点个数这个属性，因此我们也会在描述插入和删除的操作的时候描述左子树节点个数这个值的更新方式，且关于这一点，我们将左子树节点个数储存到每个节点的属性中，且在创建节点的时候这个值默认为 0。

首先是插入操作，当树是空的时候，直接将插入值设为根节点，插入的时候需要将插入值与节点值进行比较，我们将与插入值进行比较的节点称为操作节点。在处理每个操作节点时，都会将这个节点的值与插入的值进行比较，如果插入值比操作节点小，且操作节点没有左子节点，则为操作节点创建左子节点，将插入值存入操作节点的左子节点。如果插入值比操作节点小，且操做节点有左子节点，则将操作节点转换为当前操做节点的左子节点；如果插入值比操作节点大，且操作节点没有右子节点，则为操作节点创建右子节点，将插入值存入操作节点的右子节点。如果插入值比操作节点大，且操做节点有右子节点，则将操作节点转换为当前操做节点的右子节点。如果插入值和操作节点一样大，根据题目要求，此时插入失败。我们只需依次将操作节点的值加入异或值即可。如果插入成功的话，则从插入的位置开始向父节点遍历，如果遍历到的节点是父节点的左子节点，则将父节点的左子树节点个数加 1，直到遍历到根节点停止，这样就可以完成左子树节点个数的刷新。

然后是查询操作，当树为空的时候，直接按查询失败处理。查询值和操作节点名称规则与插入操作一致。我们将查询值与操作节点进行比较，如果查询值比操作节点小，且操作节点的左子节点不为空，则将查询值转为它的左子节点。如果查询之比操作节点小，且操作节点的左子节点为空，则按查询失败处理；如果查询值比操作节点大，且操作节点的右子节点不为空，则将查询值转为它的右子节点。如果查询之比操作节点大，且操作节点的右子节点为空，则按查询失败处理；如果查询之与操作节点一样大，则按照查询成功处理。其中，只需要依次将操作节点的值（包括查询值所在节点）异或进结果即可。

然后是删除操作，当树为空的时候，直接按删除失败处理。删除值和操作节点名称规则与查

询操作是一致的，实际上在找到该删除的节点之前进行的就是一个查询操作，逻辑是一致的因此在此不做赘述，仅描述删除节点时的具体操作。
首先先对这个被删除节点进行判断，如果这个节点是根节点，有以下处理：
①如果根节点没有左右子节点，则直接删除并将搜索树的根节点指针指向空地址。
②如果根节点有左子节点且没有右子节点，则直接删除根节点并让搜索树的根节点指针指向其左子节点。
③如果根节点有右子节点且没有左子节点，则直接删除根节点并让搜索树的根节点指针指向其右子节点。
④如果根节点同时具有左右子节点，首先这种情况有两种处理方式，一种是把左子树的最大值转到这个位置，并删除左子树最大值的那个位置的节点。另一种是将右子树的最小值转到这个位置，并删除右子树最小值的那个位置的节点。显然我们可以确定的是，左子树的最大值它一定是叶节点或者只有左子树，右子树的最小值它一定是叶节点或者只有右子树，对于它们的处理可以参照下文删除非根节点时处理的①②③条。
如果这个节点不是根节点，则有以下处理：
①如果这个节点没有左右子节点，则直接删除这个节点，并进行左子树节点个数的更新。
②如果这个节点有左子节点且没有右子节点，则直接删除这个节点并让其左子节点代替这个节点原本的位置，并进行左子树节点个数的更新。
③如果这个节点有右子节点且没有左子节点，则直接删除这个节点并让其右子节点代替这个节点原本的位置，并进行左子树节点个数的更新。
④如果这个节点同时具有左右子节点，首先这种情况有两种处理方式，一种是把左子树的最大值转到这个位置，并删除左子树最大值的那个位置的节点。另一种是将右子树的最小值转到这个位置，并删除右子树最小值的那个位置的节点。显然我们可以确定的是，左子树的最大值它一定是叶节点或者只有左子树，右子树的最小值它一定是叶节点或者只有右子树，对于它们的处理可以参照上文删除非根节点时处理的①②③条。
上文中提到的更新左子树节点个数，则指从被删除节点的位置开始向上遍历，如果被遍历到的节点是其父节点的左子节点，则将其父节点的左子节点个数减 1，直到遍历到根节点为止。且根据题意，我们只需要像查询操作时做的那样把被访问到的节点的值异或到结果里即可。
然后是按名次查找。这个操作就需要使用到我们在插入和删除操作中所记录下的左子树节点的个数了。首先如果给的名次小于 1 或者大于树的节点个数，则直接按照查询失败处理。从根节点开始作为操作节点，我们将操作节点的左子树节点个数加 1 之后与查询名次进行比较，如果操作节点的左子树个数加 1 之后比查询名次大，则说明要查询的名次在左子树，则查询名次不变，操作节点变为其左子节点。如果操作节点的左子树个数加 1 之后比查询名次小，则说明要查询的名次在右子树，则查询名次减去操作节点的左子树节点个数再减 1，操作节点转为其右子节点。如果操作节点的左子树节点个数加 1 之后与查询名次相等，则说明要查询的节点就是当前的操作节点。根据题意，我们只需要输出我们在这个过程中所有操作节点的异或值即可。
然后是按名次删除，这个操作就是上文的复合型，即把按名次查询和删除节点的操作综合起来即可，且按照题意，我们输出的异或和也是按名次查询那个过程的异或和。
最后是按照升序输出，由二叉搜索树的定义，我们只需要将这个二叉搜索树进行中序遍历即可。
3. 测试结果（测试输入，测试输出）
测试输入 1：
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
输出为：
0
6
6
2
2
7
0
7
2
3
1
6
3
测试输入 2：
14
0 43
0 17
0 55
0 62
0 57
0 66
0 67
4 5
0 67
0 70
3 6
4 7
0 20
2 43
输出为：
0
43
43
28
34

```
34
96
34
0
29
29
91
58
43
```

4. 分析与探讨（结果分析，若存在问题，探讨解决问题的途径）

从测试结果来看，我们的算法成功解决了这个问题。存在的问题主要在于我们的内存管理上。在代码中我们有许多地方使用的是值传递，尤其是在递归过程中，这有可能会导致内存溢出。并且我们在进行删除操作的时候，存在内存空间未删除的情况，这有可能会造成碎片内存的存在。还有就是因为删除过程为了匹配题目要求的异或输出，且删除本身又是分为多种情况，因此我们的删除代码写的比较冗长，维护起来会很困难，且阅读大段代码本身也会带来麻烦。

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）

```cpp
/*2024 级数据结构--数据智能 实验 11 搜索树 A 二叉搜索树.cpp*/
#include <iostream>

using namespace std;


template<class T>
struct Node
{
    T data;
    Node<T>* left, * right, * parent;
    size_t depth;
    size_t sons;
    bool touched;
    size_t left_size;
    size_t right_size;
    Node(Node<T>* parent, const T& data) : data(data), left(nullptr), right(nullptr), parent(parent), depth(0), sons(0),left_size(0), right_size(0), touched(false) {}
    Node(): left(nullptr), right(nullptr), parent(nullptr), depth(0), sons(0), touched(false), left_size(0), right_size(0) {}
};


template<class T>
class BinarySearchTree
{
private:
    Node<T>* root;
```

```
28.       size_t count;
29.       size_t depth;
30.   public:
31.       int xor1;
32.       BinarySearchTree() : root(nullptr), count(0), depth(0), xor1(0)
      {}
33.       BinarySearchTree(size_t size);
34.       ~BinarySearchTree() {}
35.       Node<T>* getRoot() { return root; }
36.       bool insert(const T& data);
37.       bool find(const T& data);
38.       bool erase(const T& data);
39.       void print() const;
40.       void preOrderPrint(const Node<T>* p) const;
41.       void inOrderPrint();
42.       void iniNode(Node<T>* p);
43.       void iniSize(Node<T>* p);
44.       bool eraseRank(size_t rank);
45.       Node<T>* findRank(size_t rank);
46.   };
47.
48.   template<class T>
49.   BinarySearchTree<T>::BinarySearchTree(size_t size)
50.   {
51.       for (size_t i = 0; i < size; i++)
52.       {
53.           T data;
54.           cin >> data;
55.           insert(data);
56.       }
57.   }
58.
59.   template<class T>
60.   bool BinarySearchTree<T>::insert(const T& data)
61.   {
62.       if (root == nullptr)
63.       {
64.           root = new Node<T>(nullptr, data);
65.           count++;
66.           depth = 1;
67.           return true;
68.       }
69.       Node<T>* p = root;
70.       while (p != nullptr)
71.       {
72.           xor1 ^= p->data;
```

```
73.             if (data < p->data && p->left != nullptr)
74.             {
75.                 p = p->left;
76.             }
77.             else if (data > p->data && p->right != nullptr)
78.             {
79.                 p = p->right;
80.             }
81.             else if (data < p->data && p->left == nullptr)
82.             {
83.                 Node<T>* node = new Node<T>(p, data);
84.                 p->left = node;
85.                 count++;
86.                 p = p->left;
87.                 while (p != nullptr)
88.                 {
89.                     if (p->parent == nullptr)
90.                     {
91.                         break;
92.                     }
93.                     if (p->parent->left == p)
94.                     {
95.                         p->parent->left_size++;
96.                         p = p->parent;
97.                     }
98.                     else if (p->parent->right == p)
99.                     {
100.                        p->parent->right_size++;
101.                        p = p->parent;
102.                    }
103.                }
104.                return true;
105.            }
106.            else if (data > p->data && p->right == nullptr)
107.            {
108.                Node<T>* node = new Node<T>(p, data);
109.                p->right = node;
110.                count++;
111.                p = p->right;
112.                while (p != nullptr)
113.                {
114.                    if (p->parent == nullptr)
115.                    {
116.                        break;
117.                    }
118.                    if (p->parent->left == p)
```

```
                {
                    p->parent->left_size++;
                    p = p->parent;
                }
                else if (p->parent->right == p)
                {
                    p->parent->right_size++;
                    p = p->parent;
                }
            }
            return true;
        }
        else if (data == p->data)
        {
            cout << 0 << endl;
            return false;
        }
    }
    return false;
}

template<class T>
bool BinarySearchTree<T>::find(const T& data)
{
    Node<T>* p = root;
    while (p != nullptr)
    {
        if (data == p->data)
        {
            // cout << 1 << endl;
            xor1 ^= p->data;
            return true;
        }
        else if (data < p->data && p->left != nullptr)
        {
            xor1 ^= p->data;
            p = p->left;
        }
        else if (data > p->data && p->right != nullptr)
        {
            xor1 ^= p->data;
            p = p->right;
        }
        else
        {
            xor1 ^= p->data;
```

```
165.                 break;
166.             }
167.         }
168.         cout << 0 << endl;
169.         return false;
170.     }
171.
172.     template<class T>
173.     bool BinarySearchTree<T>::erase(const T& data)
174.     {
175.         Node<T>* p = root;
176.         while (p != nullptr)
177.         {
178.             if (data == p->data)
179.             {
180.                 this->xor1 ^= p->data;
181.
182.                 /*  删除节点
183.                     1. 如果节点为叶节点则直接结束
184.                     2. 如果节点只有左/右子树，则将左/右子树的根节点转移过来
185.                     3. 如果节点同时用左右子树，则将左子树中的最大值转移过来，然后删除最大值节点
186.                         这个被删除的最大值节点只有两种情况：为叶节点或者只有左子树(题目让使用右子树中最小的节点，等价)
187.                 */
188.                 if (p == root)
189.                 {
190.                     if (p->left == nullptr && p->right == nullptr)
191.                     {
192.                         count--;
193.                         delete p;
194.                         root = nullptr;
195.                         return true;
196.                     }
197.                     else if (p->left == nullptr && p->right != nullptr)
198.                     {
199.                         root = p->right;
200.                         root->parent = nullptr;
201.                         count--;
202.                         delete p;
203.                         return true;
204.                     }
205.                     else if (p->left != nullptr && p->right == nullptr)
206.                     {
```

```
207.                         root = p->left;
208.                         root->parent = nullptr;
209.                         count--;
210.                         delete p;
211.                         return true;
212.                     }
213.                     else if (p->left != nullptr && p->right != nullptr)

214.                     {
215.                         Node<T>* max_node = p->right;
216.                         while (max_node->left != nullptr)
217.                         {
218.                             max_node = max_node->left;
219.                         }
220.                         p->data = max_node->data;
221.                         auto temp = max_node;
222.                         while (temp != nullptr)
223.                         {
224.                             if (temp->parent == nullptr)
225.                             {
226.                                 break;
227.                             }
228.                             if (temp->parent->left == temp)
229.                             {
230.                                 temp->parent->left_size--;
231.                                 temp = temp->parent;
232.                             }
233.                             else if (temp->parent->right == temp)
234.                             {
235.                                 temp->parent->right_size--;
236.                                 temp = temp->parent;
237.                             }
238.                         }
239.                         if (max_node->right != nullptr)
240.                         {
241.                             max_node->right->parent = max_node->parent;

242.                             if (max_node->parent->left == max_node)
243.                             {
244.                                 max_node->parent->left = max_node->righ
    t;
245.                             }
246.                             if (max_node->parent->right == max_node)
247.                             {
248.                                 max_node->parent->right = max_node->rig
    ht;
```

```
249.                             }
250.                         }
251.                         if (max_node->right == nullptr)
252.                         {
253.                             if (max_node->parent->left == max_node)
254.                             {
255.                                 max_node->parent->left = nullptr;
256.                             }
257.                             else if (max_node->parent->right == max_nod
    e)
258.                             {
259.                                 max_node->parent->right = nullptr;
260.                             }
261.                         }
262.
263.
264.                         delete max_node;
265.                         count--;
266.                         return true;
267.                     }
268.                 }
269.
270.                 else if (p != root)
271.                 {
272.                      if (p->left == nullptr && p->right == nullptr)//叶
    节点
273.                     {
274.                         auto temp = p;
275.                         while (temp != nullptr)
276.                         {
277.                             if (temp->parent == nullptr)
278.                             {
279.                                 break;
280.                             }
281.                             if (temp->parent->left == temp)
282.                             {
283.                                 temp->parent->left_size--;
284.                                 temp = temp->parent;
285.                             }
286.                             else if (temp->parent->right == temp)
287.                             {
288.                                 temp->parent->right_size--;
289.                                 temp = temp->parent;
290.                             }
291.                         }
292.                         if (p->parent->left == p)
```

```
293.                         {
294.                             p->parent->left = nullptr;
295.                         }
296.                         else if (p->parent->right == p)
297.                         {
298.                             p->parent->right = nullptr;
299.                         }
300.                         count--;
301.
302.                         delete p;
303.                         return true;
304.                     }
305.                     else if (p->left == nullptr && p->right != nullptr)
    //有右子树
306.                     {
307.                         auto temp = p;
308.                         while (temp != nullptr)
309.                         {
310.                             if (temp->parent == nullptr)
311.                             {
312.                                 break;
313.                             }
314.                             if (temp->parent->left == temp)
315.                             {
316.                                 temp->parent->left_size--;
317.                                 temp = temp->parent;
318.                             }
319.                             else if (temp->parent->right == temp)
320.                             {
321.                                 temp->parent->right_size--;
322.                                 temp = temp->parent;
323.                             }
324.                         }
325.                         if (p->parent->left == p)
326.                         {
327.                             p->parent->left = p->right;
328.                         }
329.                         else if (p->parent->right == p)
330.                         {
331.                             p->parent->right = p->right;
332.                         }
333.                         p->right->parent = p->parent;
334.                         count--;
335.
336.                         delete p;
337.                         return true;
```

```
338.                 }
339.                 else if (p->left != nullptr && p->right == nullptr)
   //有左子树
340.                 {
341.                     auto temp = p;
342.                     while (temp != nullptr)
343.                     {
344.                         if (temp->parent == nullptr)
345.                         {
346.                             break;
347.                         }
348.                         if (temp->parent->left == temp)
349.                         {
350.                             temp->parent->left_size--;
351.                             temp = temp->parent;
352.                         }
353.                         else if (temp->parent->right == temp)
354.                         {
355.                             temp->parent->right_size--;
356.                             temp = temp->parent;
357.                         }
358.                     }
359.                     if (p->parent->left == p)
360.                     {
361.                         p->parent->left = p->left;
362.                     }
363.                     else if (p->parent->right == p)
364.                     {
365.                         p->parent->right = p->left;
366.                     }
367.                     p->left->parent = p->parent;
368.                     count--;
369.
370.                     delete p;
371.                     return true;
372.                 }
373.                 else if (p->left != nullptr && p->right != nullptr)
   //有左右子树
374.                 {
375.                     {
376.                         Node<T>* max_node = p->right;
377.                         while (max_node->left != nullptr)
378.                         {
379.                             max_node = max_node->left;
380.                         }
381.                         p->data = max_node->data;
```

```
382.                         count--;
383.                         auto temp = max_node;
384.                         while (temp != nullptr)
385.                         {
386.                             if (temp->parent == nullptr)
387.                             {
388.                                 break;
389.                             }
390.                             if (temp->parent->left == temp)
391.                             {
392.                                 temp->parent->left_size--;
393.                                 temp = temp->parent;
394.                             }
395.                             else if (temp->parent->right == temp)
396.                             {
397.                                 temp->parent->right_size--;
398.                                 temp = temp->parent;
399.                             }
400.                         }
401.                         if (max_node->right != nullptr)//有右子树
402.                         {
403.                              max_node->right->parent = max_node->par
    ent;
404.                              if (max_node->parent->left == max_node)

405.                             {
406.                                  max_node->parent->left = max_node->
    right;
407.                             }
408.                              else if (max_node->parent->right == max
    _node)
409.                             {
410.                                  max_node->parent->right = max_node-
    >right;
411.                             }
412.                         }
413.                         if (max_node->right == nullptr)//叶节点
414.                         {
415.                              if (max_node->parent->left == max_node)

416.                             {
417.                                 max_node->parent->left = nullptr;
418.                             }
419.                              else if (max_node->parent->right == max
    _node)
420.                             {
```

```
421.                                     max_node->parent->right = nullptr;
422.                                 }
423.                             }
424.
425.                             delete max_node;
426.                             return true;
427.                         }
428.                     }
429.                 }
430.
431.             }
432.             else if (data < p->data && p->left != nullptr)
433.             {
434.                 this->xor1 ^= p->data;
435.                 p = p->left;
436.             }
437.             else if (data > p->data && p->right != nullptr)
438.             {
439.                 this->xor1 ^= p->data;
440.                 p = p->right;
441.             }
442.             else
443.             {
444.                 this->xor1 ^= p->data;
445.                 break;
446.             }
447.         }
448.         cout << 0 << endl;
449.         return false;
450. }
451.
452.
453. //前序遍历输出
454. template<class T>
455. void BinarySearchTree<T>::print() const
456. {
457.     preOrderPrint(this->root);
458.     cout << endl;
459. }
460.
461.
462.
463.
464. template<class T>
465. void BinarySearchTree<T>::preOrderPrint(const Node<T>* p) const
```

```
466. {
467.     if (p == nullptr)
468.     {
469.         return;
470.     }
471.     cout << p->data << " ";
472.     preOrderPrint(p->left);
473.     preOrderPrint(p->right);
474. }
475.
476.
477. template<class T>
478. void BinarySearchTree<T>::inOrderPrint()
479. {
480.     Node<T>* node = root;
481.     iniNode(node);
482.     while (1)
483.     {
484.         if (node->left != nullptr && node->left->touched == false)
485.         {
486.             node = node->left;
487.         }
488.         else
489.         {
490.             if (node->touched == false)
491.             {
492.                 cout << node->data << " ";
493.                 node->touched = true;
494.                 if (node->right != nullptr)
495.                 {
496.                     node = node->right;
497.                 }
498.                 else
499.                 {
500.                     node = node->parent;
501.                 }
502.                 if (node == nullptr)
503.                 {
504.                     break;
505.                 }
506.             }
507.             else
508.             {
509.                 if (node->parent != nullptr)
510.                 {
```

```
511.                     node = node->parent;
512.                 }
513.                 else if (node->right != nullptr && node->right->tou
    ched == true)
514.                 {
515.                     break;
516.                 }
517.             }
518.
519.         }
520.         if (node->parent == nullptr && node->right != nullptr && no
    de->right->touched == true)
521.         {
522.             break;
523.         }
524.     }
525.
526.     cout << endl;
527.     return;
528. }
529.
530. template<class T>
531. void BinarySearchTree<T>::iniNode(Node<T>* p)
532. {
533.     if (p == nullptr)
534.     {
535.         return;
536.     }
537.     iniNode(p->left);
538.     iniNode(p->right);
539.     p->touched = false;
540. }
541.
542. template<class T>
543. void BinarySearchTree<T>::iniSize(Node<T>* p)
544. {
545.     if (p == nullptr)
546.     {
547.         return;
548.     }
549.     iniSize(p->left);
550.     iniSize(p->right);
551.     if (p->left == nullptr && p->right == nullptr)
552.     {
553.         p->left_size = 0;
554.         p->right_size = 0;
```

```
555.      }
556.      else if (p->left == nullptr && p->right != nullptr)
557.      {
558.          p->left_size = 0;
559.          p->right_size = p->right->left_size + p->right->right_size
    + 1;
560.      }
561.      else if (p->left != nullptr && p->right == nullptr)
562.      {
563.          p->left_size = p->left->left_size + p->left->right_size + 1
    ;
564.          p->right_size = 0;
565.      }
566.      else
567.      {
568.          p->left_size = p->left->left_size + p->left->right_size + 1
    ;
569.          p->right_size = p->right->left_size + p->right->right_size
    + 1;
570.      }
571.  }
572.
573.  template<class T>
574.  Node<T>* BinarySearchTree<T>::findRank(size_t rank)
575.  {
576.      if (rank > count || rank < 1)
577.      {
578.          return nullptr;
579.      }
580.      size_t temp = rank;
581.      Node<T>* node = root;
582.      while (node != nullptr)
583.      {
584.          this->xor1 ^= node->data;
585.          if (node->left_size + 1 == temp)
586.          {
587.              return node;
588.          }
589.          else if (node->left_size + 1 < temp)
590.          {
591.              temp -= (node->left_size + 1);
592.              node = node->right;
593.          }
594.          else
595.          {
596.              node = node->left;
```

```
597.             }
598.         }
599.         return nullptr;
600.     }
601.
602.     template<class T>
603.     bool BinarySearchTree<T>::eraseRank(size_t rank)
604.     {
605.         if (rank > count || rank < 1)
606.         {
607.             return false;
608.         }
609.         size_t temp = rank;
610.         Node<T>* p = root;
611.         while (p != nullptr)
612.         {
613.             this->xor1 ^= p->data;
614.             if (p->left_size + 1 == temp)
615.             {
616.                 if (p == root)
617.                 {
618.                     if (p->left == nullptr && p->right == nullptr)
619.                     {
620.                         count--;
621.                         delete p;
622.                         root = nullptr;
623.                         return true;
624.                     }
625.                     else if (p->left == nullptr && p->right != nullptr)
626.                     {
627.                         root = p->right;
628.                         root->parent = nullptr;
629.                         count--;
630.                         delete p;
631.                         return true;
632.                     }
633.                     else if (p->left != nullptr && p->right == nullptr)
634.                     {
635.                         root = p->left;
636.                         root->parent = nullptr;
637.                         count--;
638.                         delete p;
639.                         return true;
640.                     }
```

```
641.                 else if (p->left != nullptr && p->right != nullptr)
642.                 {
643.                     Node<T>* max_node = p->right;
644.                     while (max_node->left != nullptr)
645.                     {
646.                         max_node = max_node->left;
647.                     }
648.                     p->data = max_node->data;
649.                     auto temp = max_node;
650.                     while (temp != nullptr)
651.                     {
652.                         if (temp->parent == nullptr)
653.                         {
654.                             break;
655.                         }
656.                         if (temp->parent->left == temp)
657.                         {
658.                             temp->parent->left_size--;
659.                             temp = temp->parent;
660.                         }
661.                         else if (temp->parent->right == temp)
662.                         {
663.                             temp->parent->right_size--;
664.                             temp = temp->parent;
665.                         }
666.                     }
667.                     if (max_node->right != nullptr)
668.                     {
669.                         max_node->right->parent = max_node->parent;
670.                         if (max_node->parent->left == max_node)
671.                         {
672.                             max_node->parent->left = max_node->righ
    t;
673.                         }
674.                         if (max_node->parent->right == max_node)
675.                         {
676.                             max_node->parent->right = max_node->rig
    ht;
677.                         }
678.                     }
679.                     if (max_node->right == nullptr)
680.                     {
681.                         if (max_node->parent->left == max_node)
682.                         {
```

```
683.                                 max_node->parent->left = nullptr;
684.                             }
685.                             else if (max_node->parent->right == max_nod
    e)
686.                             {
687.                                 max_node->parent->right = nullptr;
688.                             }
689.                         }
690.
691.
692.                         delete max_node;
693.                         count--;
694.                         return true;
695.                     }
696.                 }
697.                 else if (p != root)
698.                 {
699.                     if (p->left == nullptr && p->right == nullptr)
700.                     {
701.
702.                         auto temp = p;
703.                         while (temp != nullptr)
704.                         {
705.                             if (temp->parent == nullptr)
706.                             {
707.                                 break;
708.                             }
709.                             if (temp->parent->left == temp)
710.                             {
711.                                 temp->parent->left_size--;
712.                                 temp = temp->parent;
713.                             }
714.                             else if (temp->parent->right == temp)
715.                             {
716.                                 temp->parent->right_size--;
717.                                 temp = temp->parent;
718.                             }
719.                         }
720.                         if (p->parent->left == p)
721.                         {
722.                             p->parent->left = nullptr;
723.                         }
724.                         else if (p->parent->right == p)
725.                         {
726.                             p->parent->right = nullptr;
727.                         }
```

```
728.                     count--;
729.
730.                     delete p;
731.                     return true;
732.                 }
733.                 else if (p->left == nullptr && p->right != nullptr)

734.                 {
735.                     auto temp = p;
736.                     while (temp != nullptr)
737.                     {
738.                         if (temp->parent == nullptr)
739.                         {
740.                             break;
741.                         }
742.                         if (temp->parent->left == temp)
743.                         {
744.                             temp->parent->left_size--;
745.                             temp = temp->parent;
746.                         }
747.                         else if (temp->parent->right == temp)
748.                         {
749.                             temp->parent->right_size--;
750.                             temp = temp->parent;
751.                         }
752.                     }
753.                     if (p->parent->left == p)
754.                     {
755.                         p->parent->left = p->right;
756.                     }
757.                     else if (p->parent->right == p)
758.                     {
759.                         p->parent->right = p->right;
760.                     }
761.                     p->right->parent = p->parent;
762.                     count--;
763.
764.                     delete p;
765.                     return true;
766.                 }
767.                 else if (p->left != nullptr && p->right == nullptr)

768.                 {
769.                     auto temp = p;
770.                     while (temp != nullptr)
771.                     {
```

```
772.                         if (temp->parent == nullptr)
773.                         {
774.                             break;
775.                         }
776.                         if (temp->parent->left == temp)
777.                         {
778.                             temp->parent->left_size--;
779.                             temp = temp->parent;
780.                         }
781.                         else if (temp->parent->right == temp)
782.                         {
783.                             temp->parent->right_size--;
784.                             temp = temp->parent;
785.                         }
786.                     }
787.                     if (p->parent->left == p)
788.                     {
789.                         p->parent->left = p->left;
790.                     }
791.                     else if (p->parent->right == p)
792.                     {
793.                         p->parent->right = p->left;
794.                     }
795.                     p->left->parent = p->parent;
796.                     count--;
797.
798.                     delete p;
799.                     return true;
800.                 }
801.                 else if (p->left != nullptr && p->right != nullptr)

802.                 {
803.                     {
804.                         Node<T>* max_node = p->right;
805.                         while (max_node->left != nullptr)
806.                         {
807.                             max_node = max_node->left;
808.                         }
809.                         p->data = max_node->data;
810.                         count--;
811.                         auto temp = max_node;
812.                         while (temp != nullptr)
813.                         {
814.                             if (temp->parent == nullptr)
815.                             {
816.                                 break;
```

```
817.                                }
818.                                if (temp->parent->left == temp)
819.                                {
820.                                    temp->parent->left_size--;
821.                                    temp = temp->parent;
822.                                }
823.                                else if (temp->parent->right == temp)
824.                                {
825.                                    temp->parent->right_size--;
826.                                    temp = temp->parent;
827.                                }
828.                            }
829.                            if (max_node->right != nullptr)
830.                            {
831.                                max_node->right->parent = max_node->par
    ent;
832.                                if (max_node->parent->left == max_node)

833.                                {
834.                                    max_node->parent->left = max_node->
    right;
835.                                }
836.                                else if (max_node->parent->right == max
    _node)
837.                                {
838.                                    max_node->parent->right = max_node-
    >right;
839.                                }
840.                            }
841.                            if (max_node->right == nullptr)
842.                            {
843.                                if (max_node->parent->left == max_node)

844.                                {
845.                                    max_node->parent->left = nullptr;
846.                                }
847.                                else if (max_node->parent->right == max
    _node)
848.                                {
849.                                    max_node->parent->right = nullptr;

850.                                }
851.                            }
852.
853.                            delete max_node;
854.                            return true;
```

```
                    }
                }
            }

            return true;
        }
        else if (p->left_size + 1 < temp)
        {
            temp -= p->left_size + 1;
            p = p->right;
        }
        else
        {
            p = p->left;
        }
    }
    return false;
}


class Solution
{
public:
    void solve();
    void test();
};


void Solution::solve()
{
    ios::sync_with_stdio(false);
    BinarySearchTree<long long> bst;
    size_t m;
    cin >> m;
    for (size_t i = 0; i < m; i++)
    {
        long long op, x;
        cin >> op >> x;
        if (op == 0)
        {
            bst.xor1 = 0;
            if (bst.insert(x))
            {
                cout << bst.xor1 << endl;
            }
```

```
901.         }
902.         else if (op == 1)
903.         {
904.             bst.xor1 = 0;
905.             if (bst.find(x))
906.             {
907.                 cout << bst.xor1 << endl;
908.             }
909.         }
910.         else if (op == 2)
911.         {
912.             bst.xor1 = 0;
913.             if (bst.erase(x))
914.             {
915.                 cout << bst.xor1 << endl;
916.             }
917.         }
918.         else if (op == 3)
919.         {
920.             // bst.iniSize(bst.getRoot());
921.             bst.xor1 = 0;
922.             Node<long long>* p = bst.findRank(x);
923.             if (p != nullptr)
924.             {
925.                 cout << bst.xor1 << endl;
926.             }
927.             else
928.             {
929.                 cout << 0 << endl;
930.             }
931.         }
932.         else if (op == 4)
933.         {
934.             // bst.iniSize(bst.getRoot());
935.             bst.xor1 = 0;
936.             if (bst.eraseRank(x))
937.             {
938.                 cout << bst.xor1 << endl;
939.             }
940.             else
941.             {
942.                 cout << 0 << endl;
943.             }
944.         }
945.         // cout << endl;
946.         // cout << "preorder: ";
```

```
947.             // bst.print();
948.             // cout << endl;
949.             // cout << "inorder: ";
950.             // bst.inOrderPrint();
951.             // cout << endl;
952.         }
953. 
954. }
955. 
956. void Solution::test()
957. {
958.     BinarySearchTree<int> bst;
959.     bst.insert(5);
960.     bst.insert(3);
961.     bst.insert(7);
962.     bst.insert(2);
963.     bst.inOrderPrint();
964.     bst.iniSize(bst.getRoot());
965.     bst.eraseRank(2);
966.     bst.inOrderPrint();
967.     cout << bst.findRank(2)->data << endl;
968. }
969. 
970. int main()
971. {
972.     Solution solution;
973.     solution.solve();
974.     // solution.test();
975.     return 0;
976. }
```