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　　：202200400053	姓名：　王宇涵	班级：　22 级 2 班
实验题目：搜索树		
实验学时：2	实验日期：　　2023-11-29	

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：

Vscode

1. 实验内容

题目描述： 创建带索引的二叉搜索树类（采用链表存储），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：第一行输入一个不超过 1000000 的整数 m 表示操作的次数，接下来 m 行每行输入两个数 a、b： 1.当 a＝0 时，向搜索树中插入 b； 2.当 a＝1 时，在搜索树中查找 b；3.当 a＝2 时，从搜索树中删除 b；4.当 a＝3 时，查找搜索树中名次为 b 的元素；5.当 a＝4 时，删除搜索树中名次为 b 的元素。

输出：对于每次操作输出一行，每行含有一个数，表示执行此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）

main()函数

插入操作 (a = 0):

使用 idxTree.insert(pair<int, int>(b, b)) 将键值对 (b, b) 插入到索引二叉搜索树中，并输出插入的结果。

查找操作 (a = 1):

使用 idxTree.find(b) 查找键为 b 的值，并输出异或的结果。

删除操作 (a = 2):

使用 idxTree.erase(b) 删除键为 b 的值，并输出异或的结果。

获取特定索引元素操作 (a = 3):

使用 idxTree.get(b - 1) 获取索引为 b - 1 的元素，并输出异或的结果。

删除特定索引位置元素操作 (a = 4):

使用 idxTree.eraseWithIndex(b - 1) 删除索引位置为 b - 1 的元素，并输出异或的结果。

实现了根据输入的不同操作类型，使用 IndexedBinarySearchTree 类的实例 idxTree 执行相应的操作，并输出相应的比较值异或结果。

IndexedBinarySearchTree 类

继承关系：

继承自 indexedBSTree<K, E> 和 LinkedBinaryTree<IdxNode<K, E>>。

数据成员：

使用了父类 LinkedBinaryTree<IdxNode<K, E>> 中定义的 root 和 treeSize。

方法

- get(int theIndex):

获取特定索引值对应的元素。

- eraseWithIndex(int theIndex):

删除特定索引位置的元素。

- empty():

判断索引树是否为空。

- size():

获取索引树中元素的个数。

- find(const K& theKey):

查找具有给定键值的元素。

- erase(const K& theKey):

删除具有给定键值的元素。

- insert(const pair<K, E>& thePair):

插入键值对到索引树中。

get 函数解析

get() 函数根据给定的索引值 theIndex 在索引二叉搜索树中查找对应的元素。

遍历索引树节点，根据当前节点的索引值与目标索引值的大小关系不断向左子树或右子树移动，直到找到目标索引值所在的节点

注意向右子树结点移动时需要更新索引 index=index-theIndex-1;

insert 函数解析

insert() 函数将键值对 thePair 插入到索引二叉搜索树中。

遍历索引树节点，找到合适的插入位置，并在找到的位置创建新的节点，并将 thePair 插入到树中。

如果已经存在相同的键值对，则不执行插入操作，直接返回 0。

如果成功插入，返回插入的键值对的值。

注意:向左子树进行查找时需要对路径上的元素 Index++;

erase 函数解析

erase() 函数用于删除具有给定键值 theKey 的元素。

注意: 查找要删除的元素时向左子树进行查找时需要对路径上的元素 Index--,如果要删除的元素有两个孩子,寻找左孩子最大值或右孩子最小值的时候向左查找的时候需要对 index--;

3. 测试结果（测试输入，测试输出）

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:

0

6

6

2

2

7

0

7

2

3

1

6

3

输入

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

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）

本次实验存在一些难点和问题,最后通过不断调试和思考成功解决

难点一:如何实现建立每个结点的索引?

答 ： 通过插入和删除操作时,如果遍历了左子树,则对结点进行 index++(--)

难点二:如何查找搜索树名次为 b 的元素?

答: 如果需要寻找某索引为 a 的结点的左子树,则 b 不变,若右子树,则 b 需要更新为 b-a-1

易错点:注意输入需要查的第 b 大元素实际上搜索索引为 b-1 的元素

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）

```
#pragma once
#include<iostream>
using namespace std;
#include<utility>


template<class K ,class E>
class Dictionary
{
    public:
        virtual ~Dictionary(){}
        virtual bool empty()const=0;
        virtual int size()const=0;
        virtual int find(const K&)const=0;
        virtual int erase(const K&)=0;
        virtual int insert(const pair<K,E>&)=0;
};
template <class K,class E>
```

```
class indexedBSTree:public Dictionary<K,E>
{
    public:
        virtual int get(int )const=0;
        virtual int eraseWithIndex (int) =0;
};
#pragma once
#pragma once
#include<iostream>
using namespace std;
template<class T>
struct BinaryTreeNode
{
    T element;
    BinaryTreeNode<T>* leftChild;
    BinaryTreeNode<T>* rightChild;
    BinaryTreeNode()
    {
        leftChild=rightChild=NULL;
    }
    BinaryTreeNode(const T& theElement)
    {
        element=theElement;
        leftChild=rightChild=NULL;
    }
    BinaryTreeNode(const T& theElement,BinaryTreeNode * theLeftChild,BinaryTreeNode* theRightChild)
    {
        element=theElement;
        leftChild=theLeftChild;
        rightChild=theRightChild;
    }
};
```

```
#pragma once
#include<iostream>
using namespace std;
template<class T>
class BinaryTree
{
    public:
        virtual ~BinaryTree(){}
        virtual bool empty()const =0;
        virtual int size()const =0;
        virtual void preOrder(void (*) (T*))=0;
        virtual void inOrder(void(*)(T*))=0;
        virtual void postOrder(void(*)(T*))=0;
        virtual void leverOrder(void(*)(T*))=0;
};
#include<queue>

template<class E>
class LinkedBinaryTree:public BinaryTree<BinaryTreeNode<E>>
{
    public:
        LinkedBinaryTree()
        {
            root=NULL;
            treeSize=0;
            visit=output;
        }
        ~LinkedBinaryTree()
        {
            erase();
        }
        void createTreeByLevel();
```

```
bool empty() const
{
    return treeSize==0;
}
int size()const
{
    return treeSize;
}
//遍历函数,传入 visit 函数指针
void preOrder(void (*theVisit) (BinaryTreeNode<E>*))
{
    visit=theVisit;preOrder(root);
}
void inOrder(void (*theVisit) (BinaryTreeNode<E>*))
{
    visit=theVisit;inOrder(root);
}
void postOrder(void (*theVisit) (BinaryTreeNode<E>*))
{
    visit=theVisit;postOrder(root);
}
void leverOrder(void (*theVisit) (BinaryTreeNode<E>*))
{
    visit=theVisit;leverOrder(root);
}
void preOrderOutput()
{
    preOrder(output);
    cout<<endl;
}
void inOrderOutput()
{
```

```
        inOrder(output);
        cout<<endl;
    }
    void postOrderOutput()
    {
        postOrder(output);
        cout<<endl;
    }
    void leverOrderOutput()
    {
        leverOrder(output);
        cout<<endl;
    }
    void erase()
    {
        postOrder(dispose);
        root=NULL;
        treeSize=0;
    }
    int height() const
    {
        return height(root);
    }

    void makeTree(const E& theElement,LinkedBinaryTree<E>&a,LinkedBinaryTree<E>&b);
protected:
    BinaryTreeNode<E>* root;
    int treeSize;
    //定义函数指针,规定如何访问元素
    static void (*visit)(BinaryTreeNode<E>*);
    static void preOrder(BinaryTreeNode<E>*t);
    static void inOrder(BinaryTreeNode<E>*t);
```

```
        static void postOrder(BinaryTreeNode<E>*t);
        static void leverOrder(BinaryTreeNode<E>*t);
        static void output(BinaryTreeNode<E>*t)
        {
            cout<<t->element.index<<" "<<t->element.value.first<<" "<<t->element.value.second<<endl;
        }
        static void dispose(BinaryTreeNode<E>*t)
        {
            delete t;
        }
        static int height(BinaryTreeNode<E>*t)
        {
            if(t==NULL)
                return 0;
            int hl=height(t->leftChild);
            int hr=height(t->rightChild);
            if(hl>hr)
                return ++hl;
            else
                return ++hr;
        }

};


template<class E>
void (*LinkedBinaryTree<E>::visit)(BinaryTreeNode<E>*);

template <class E>
inline void LinkedBinaryTree<E>::createTreeByLevel()
{
    int n;cin>>n;
```

```
        int a[n];
        for(int i=0;i<n;i++)
            cin>>a[i];

        queue<BinaryTreeNode<E>*>q;

        int index=0;
        root=new BinaryTreeNode<E> (a[index++]);
        q.push(root);
        BinaryTreeNode<E>* p=NULL;
        while(!q.empty()&&index<n)
        {
            p=q.front();
            q.pop();
            //如果不空就创建一个节点
            BinaryTreeNode<E>* leftNode=new BinaryTreeNode<E>(a[index++]);
            p->leftChild=leftNode;
            q.push(leftNode);
            treeSize++;

            if(index<n)
            {
                BinaryTreeNode<E>* rightNode=new BinaryTreeNode<E>(a[index++]);
                p->rightChild=rightNode;
                q.push(rightNode);
                treeSize++;
            }
        }
}

template <class E>
inline void LinkedBinaryTree<E>::preOrder(BinaryTreeNode<E> *t)
```

```
{
    if(t!=NULL)
    {
        LinkedBinaryTree::visit(t);
        preOrder(t->leftChild);
        preOrder(t->rightChild);
    }
}

template <class E>
inline void LinkedBinaryTree<E>::inOrder(BinaryTreeNode<E>*t)
{
    if(t!=NULL)
    {
        inOrder(t->leftChild);
        LinkedBinaryTree<E>::visit(t);
        inOrder(t->rightChild);
    }
}

template <class E>
inline void LinkedBinaryTree<E>::postOrder(BinaryTreeNode<E>*t)
{
    if(t!=NULL)
    {
        postOrder(t->leftChild);
        postOrder(t->rightChild);
        LinkedBinaryTree<E>::visit(t);
    }
}

template <class E>
```

```
inline void LinkedBinaryTree<E>::leverOrder(BinaryTreeNode<E>*t)
{
    queue<BinaryTreeNode<E>*>q;
    while(t!=NULL)
    {
        visit(t);

        if(t->leftChild!=NULL)
            q.push(t->leftChild);
        if(t->rightChild!=NULL)
            q.push(t->rightChild);

        if(!q.empty())
        {
            t=q.front();
            q.pop();
        }
        else
            return;
    }
}

template <class E>
inline void LinkedBinaryTree<E>::makeTree(const E &theElement, LinkedBinaryTree<E> &a, LinkedBinaryTree<E> &b)
{
    BinaryTreeNode<E>* tmp=b.root;
    root=new BinaryTreeNode<E> (theElement,a.root,tmp);
    treeSize=a.treeSize+b.treeSize+1;

    a.root=b.root=NULL;
    a.treeSize=b.treeSize=0;
}
```

```
template<class K,class E>
struct IdxNode
{
    int index;
    pair <K,E> value;
};



template<class K,class E>
class IndexedBinarySearchTree:public indexedBSTree<K,E>,public LinkedBinaryTree<IdxNode<K,E>>
{
    public:
        using LinkedBinaryTree<IdxNode<K,E>>::inOrderOutput;
        using LinkedBinaryTree<IdxNode<K,E>>::treeSize;
        using LinkedBinaryTree<IdxNode<K,E>>::root;
        int get(int theIndex)const;
        int eraseWithIndex (int theIndex);
        bool empty()const
        {
            return treeSize==0;
        }
        int size()const
        {
            return treeSize;
        }
        int find(const K& theKey)const;
        int erase(const K& theKey);
        int insert(const pair<K,E>& thePair);
};
```

```
template <class K, class E>
inline int IndexedBinarySearchTree<K, E>::get(int theIndex)const
{
	BinaryTreeNode<IdxNode<K,E>>*p=root;
	int sum=0;
	while(p!=NULL&&p->element.index!=theIndex)
	{
		if(theIndex<p->element.index)
		{
			sum^=p->element.value.first;
			p=p->leftChild;
		}
		else if(theIndex>p->element.index)
		{
			sum^=p->element.value.first;
			theIndex=theIndex-p->element.index-1;
			p=p->rightChild;
		}
	}
	if(p==NULL)
		return 0;
	else
	{
		sum^=p->element.value.first;
		return sum;
	}
}

template <class K, class E>
inline int    IndexedBinarySearchTree<K, E>::eraseWithIndex(int theIndex)
{
```

```
int sum=0;
BinaryTreeNode<IdxNode<K,E>>*p=root,*pp=NULL;
while(p!=NULL&&p->element.index!=theIndex)
{
    pp=p;
    if(theIndex<p->element.index)
    {
        sum^=p->element.value.first;
        p->element.index--;
        p=p->leftChild;
    }
    else if(theIndex>p->element.index)
    {
        sum^=p->element.value.first;
        theIndex=theIndex-p->element.index-1;
        p=p->rightChild;
    }
}
if(p==NULL)
    return 0;
else
    sum^=p->element.value.first;
if(p->leftChild!=NULL&&p->rightChild!=NULL)
{
    //找到右孩子最小值
    BinaryTreeNode<IdxNode<K,E>>* s=p->rightChild,*ps=p;
    //沿着左孩子搜
    while (s->leftChild!=NULL)
    {
        ps=s;
        s->element.index--;
        s=s->leftChild;
```

```
        }

        //先建立 q 结点
        IdxNode<K,E>tmp={p->element.index,s->element.value};
        BinaryTreeNode<IdxNode<K,E>>* q=new BinaryTreeNode<IdxNode<K,E>>
        (tmp,p->leftChild,p->rightChild);
        //给 q 找到位置
        if(pp==NULL)
            root=q;
        else if(p==pp->leftChild)
            pp->leftChild=q;
        else
            pp->rightChild=q;

        //因为马上要删除 p,所以不能将 ps 直接赋给 pp,而是将 q 赋给 pp.因为 q->leftchild==s
        if(p==ps)
            pp=q;
        else
            pp=ps;

        delete p;
        p=s;
    }
    //此时 p 最多只有一个孩子
    BinaryTreeNode<IdxNode<K,E>>*c=NULL;
    if(p->leftChild!=NULL)
        c=p->leftChild;
    else
        c=p->rightChild;

    if(p==root)
        root=c;
```

```
        else if(p==pp->leftChild)
                pp->leftChild=c;
        else
                pp->rightChild=c;
        treeSize--;
        delete p;

        return sum;
}

template <class K, class E>
inline int IndexedBinarySearchTree<K, E>::find(const K &theKey) const
{
        int sum=0;
        BinaryTreeNode<IdxNode<K,E>>*p=root;
        while(p!=NULL)
        {
                if(theKey<p->element.value.first)
                {
                        sum^=p->element.value.first;
                        p=p->leftChild;
                }
                else if(theKey>p->element.value.first)
                {
                        sum^=p->element.value.first;
                        p=p->rightChild;
                }
                else
                {
                        sum^=theKey;
                        return sum;
                }
```

```
        }
        return 0;
}

template <class K, class E>
inline int IndexedBinarySearchTree<K, E>::erase(const K &theKey)
{
        int sum=0;
        BinaryTreeNode<IdxNode<K,E>> * p=root,*pp=NULL;
        while(p!=NULL&&p->element.value.first!=theKey)
        {
                pp=p;
                if(theKey<p->element.value.first)
                {
                        sum^=p->element.value.first;
                        p=p->leftChild;
                }
                else
                {
                        sum^=p->element.value.first;
                        p=p->rightChild;
                }
        }

        if(p==NULL)
                return 0;
        else
                sum^=theKey;

        p=root;
        while(p!=NULL&&p->element.value.first!=theKey)
        {
```

```
        if(theKey<p->element.value.first)
        {
            p->element.index--;
            p=p->leftChild;
        }
        else if(theKey>p->element.value.first)
        {
            p=p->rightChild;
        }
    }

    if(p->leftChild!=NULL&&p->rightChild!=NULL)
    {
        //找到右孩子最小值
        BinaryTreeNode<IdxNode<K,E>>* s=p->rightChild,*ps=p;
        //沿着左孩子搜
        while (s->leftChild!=NULL)
        {
            ps=s;
            s->element.index--;
            s=s->leftChild;
        }

        //先建立 q 结点
        IdxNode<K,E>tmp={p->element.index,s->element.value};
        BinaryTreeNode<IdxNode<K,E>>* q=new BinaryTreeNode<IdxNode<K,E>>
        (tmp,p->leftChild,p->rightChild);
        //给 q 找到位置
        if(pp==NULL)
            root=q;
        else if(p==pp->leftChild)
            pp->leftChild=q;
```

```
            else
                pp->rightChild=q;

            //因为马上要删除 p,所以不能将 ps 直接赋给 pp,而是将 q 赋给 pp.因为 q->leftchild==s
            if(p==ps)
                pp=q;
            else
                pp=ps;

            delete p;
            p=s;
        }
        //此时 p 最多只有一个孩子
        BinaryTreeNode<IdxNode<K,E>>*c=NULL;
        if(p->leftChild!=NULL)
            c=p->leftChild;
        else
            c=p->rightChild;

        if(p==root)
            root=c;
        else if(p==pp->leftChild)
            pp->leftChild=c;
        else
            pp->rightChild=c;
        treeSize--;
        delete p;

        return sum;
    }

    template <class K, class E>
```

```
inline int    IndexedBinarySearchTree<K, E>::insert(const pair<K, E> &thePair)
{
      int sum=0;
      BinaryTreeNode<IdxNode<K,E>>* p=root,*pp=NULL;
      int theKey=thePair.first;

      while(p!=NULL)
      {
            pp=p;
            if(theKey<p->element.value.first)
            {
                  sum^=p->element.value.first;
                  p=p->leftChild;
            }
            else if(theKey>p->element.value.first)
            {
                  sum^=p->element.value.first;
                  p=p->rightChild;
            }
            else
            {
                  return 0;
            }
      }

      IdxNode<K,E> tmp={0,pair<K,E>(thePair)};
      BinaryTreeNode<IdxNode<K,E>>* newNode=new BinaryTreeNode<IdxNode<K,E>> (tmp);

     if(root!=NULL)
      {
            if(theKey<pp->element.value.first)
            {
```

```
                pp->leftChild=newNode;
            }
            else
            {
                pp->rightChild=newNode;
            }
        }
        else
        {
            root=newNode;
        }
        treeSize++;

        p=root;
        while(p!=NULL&&p->element.value.first!=theKey)
        {
            if(theKey<p->element.value.first)
            {
                p->element.index++;
                p=p->leftChild;
            }
            else if(theKey>p->element.value.first)
            {
                p=p->rightChild;
            }
        }

        return sum;
}

IndexedBinarySearchTree<int,int> idxTree;
int main()
```

```
{
    int m;
    scanf("%d",&m);
    int a,b;
    while(m--)
    {
        scanf("%d%d",&a,&b);
        switch (a)
        {
            case 0:
            {
                cout<<idxTree.insert(pair<int,int>(b,b))<<endl;
                break;
            }
            case 1:
            {
                cout<<idxTree.find(b)<<endl;
                break;
            }
            case 2:
            {
                cout<<idxTree.erase(b)<<endl;
                break;
            }
            case 3:
            {
                cout<<idxTree.get(b-1)<<endl;
                break;
            }
            case 4:
            {
                cout<<idxTree.eraseWithIndex(b-1)<<endl;
```

```
                    break;
                }
                default:
                    break;
            }
        }
        return 0;
    }
```