数据结构与算法8-集合和映射

笔记本:	我的笔记		
创建时间:	2020/10/4 8:19	**更新时间:**	2020/10/4 9:15
作者:	liuhouer		
标签:	算法		

1.集合set定义以及用二分搜索树BST实现set

```
public interface Set<E> {
    void add(E e);
    boolean contains(E e);
    void remove(E e);
    int getSize();
    boolean isEmpty();
}
```

```
import java.util.LinkedList;
import java.util.Queue;
import java.util.Stack;


public class BST<E extends Comparable<E>> {


    private class Node{
        public E e;
        public Node left, right;


        public Node(E e){
            this.e = e;
            left = null;
            right = null;
        }
    }


    private Node root;
    private int size;


    public BST(){
        root = null;
        size = 0;
    }


    public int size(){
        return size;
    }


    public boolean isEmpty(){
        return size == 0;
    }


    // 向二分搜索树中添加新的元素e
    public void add(E e){
        root = add(root, e);
    }
```

```
// 向以node为根的二分搜索树中插入元素e，递归算法
// 返回插入新节点后二分搜索树的根
private Node add(Node node, E e){


    if(node == null){
        size ++;
        return new Node(e);
    }


    if(e.compareTo(node.e) < 0)
        node.left = add(node.left, e);
    else if(e.compareTo(node.e) > 0)
        node.right = add(node.right, e);


    return node;
}


// 看二分搜索树中是否包含元素e
public boolean contains(E e){
    return contains(root, e);
}


// 看以node为根的二分搜索树中是否包含元素e, 递归算法
private boolean contains(Node node, E e){


    if(node == null)
        return false;


    if(e.compareTo(node.e) == 0)
        return true;
    else if(e.compareTo(node.e) < 0)
        return contains(node.left, e);
    else // e.compareTo(node.e) > 0
        return contains(node.right, e);
}


// 二分搜索树的前序遍历
public void preOrder(){
    preOrder(root);
}


// 前序遍历以node为根的二分搜索树, 递归算法
private void preOrder(Node node){


    if(node == null)
        return;


    System.out.println(node.e);
    preOrder(node.left);
    preOrder(node.right);
}


// 二分搜索树的非递归前序遍历
public void preOrderNR(){


    Stack<Node> stack = new Stack<>();
    stack.push(root);
    while(!stack.isEmpty()){
        Node cur = stack.pop();
        System.out.println(cur.e);
```

```
            if(cur.right != null)
                stack.push(cur.right);
            if(cur.left != null)
                stack.push(cur.left);
        }
    }


    // 二分搜索树的中序遍历
    public void inOrder(){
        inOrder(root);
    }


    // 中序遍历以node为根的二分搜索树, 递归算法
    private void inOrder(Node node){


        if(node == null)
            return;


        inOrder(node.left);
        System.out.println(node.e);
        inOrder(node.right);
    }


    // 二分搜索树的后序遍历
    public void postOrder(){
        postOrder(root);
    }


    // 后序遍历以node为根的二分搜索树, 递归算法
    private void postOrder(Node node){


        if(node == null)
            return;


        postOrder(node.left);
        postOrder(node.right);
        System.out.println(node.e);
    }


    // 二分搜索树的层序遍历
    public void levelOrder(){


        Queue<Node> q = new LinkedList<>();
        q.add(root);
        while(!q.isEmpty()){
            Node cur = q.remove();
            System.out.println(cur.e);


            if(cur.left != null)
                q.add(cur.left);
            if(cur.right != null)
                q.add(cur.right);
        }
    }


    // 寻找二分搜索树的最小元素
    public E minimum(){
        if(size == 0)
            throw new IllegalArgumentException("BST is empty!");


        return minimum(root).e;
    }


    // 返回以node为根的二分搜索树的最小值所在的节点
    private Node minimum(Node node){
        if(node.left == null)
            return node;
        return minimum(node.left);
    }


    // 寻找二分搜索树的最大元素
    public E maximum(){
        if(size == 0)
            throw new IllegalArgumentException("BST is empty");


        return maximum(root).e;
    }


    // 返回以node为根的二分搜索树的最大值所在的节点
    private Node maximum(Node node){
        if(node.right == null)
            return node;


        return maximum(node.right);
    }


    // 从二分搜索树中删除最小值所在节点, 返回最小值
    public E removeMin(){
        E ret = minimum();
        root = removeMin(root);
        return ret;
    }


    // 删除掉以node为根的二分搜索树中的最小节点
    // 返回删除节点后新的二分搜索树的根
    private Node removeMin(Node node){


        if(node.left == null){
            Node rightNode = node.right;
            node.right = null;
            size --;
            return rightNode;
        }


        node.left = removeMin(node.left);
        return node;
    }


    // 从二分搜索树中删除最大值所在节点
    public E removeMax(){
        E ret = maximum();
        root = removeMax(root);
        return ret;
    }


    // 删除掉以node为根的二分搜索树中的最大节点
    // 返回删除节点后新的二分搜索树的根
    private Node removeMax(Node node){


        if(node.right == null){
            Node leftNode = node.left;
            node.left = null;
            size --;
            return leftNode;
        }


        node.right = removeMax(node.right);
        return node;
    }


    // 从二分搜索树中删除元素为e的节点
    public void remove(E e){
        root = remove(root, e);
    }


    // 删除掉以node为根的二分搜索树中值为e的节点, 递归算法
    // 返回删除节点后新的二分搜索树的根
    private Node remove(Node node, E e){


        if( node == null )
            return null;


        if( e.compareTo(node.e) < 0 ){
            node.left = remove(node.left , e);
            return node;
        }
        else if(e.compareTo(node.e) > 0 ){
            node.right = remove(node.right, e);
            return node;
        }
        else{   // e.compareTo(node.e) == 0


            // 待删除节点左子树为空的情况
            if(node.left == null){
                Node rightNode = node.right;
                node.right = null;
                size --;
                return rightNode;
            }


            // 待删除节点右子树为空的情况
            if(node.right == null){
                Node leftNode = node.left;
                node.left = null;
                size --;
                return leftNode;
            }


            // 待删除节点左右子树均不为空的情况


            // 找到比待删除节点大的最小节点, 即待删除节点右子树的最小节点
            // 用这个节点顶替待删除节点的位置
            Node successor = minimum(node.right);
            successor.right = removeMin(node.right);
            successor.left = node.left;


            node.left = node.right = null;


            return successor;
        }
    }


    @Override
    public String toString(){
        StringBuilder res = new StringBuilder();
        generateBSTString(root, 0, res);
        return res.toString();
    }


    // 生成以node为根节点，深度为depth的描述二叉树的字符串
    private void generateBSTString(Node node, int depth, StringBuilder res){


        if(node == null){
            res.append(generateDepthString(depth) + "null\n");
            return;
        }


        res.append(generateDepthString(depth) + node.e +"\n");
        generateBSTString(node.left, depth + 1, res);
        generateBSTString(node.right, depth + 1, res);
    }


    private String generateDepthString(int depth){
        StringBuilder res = new StringBuilder();
        for(int i = 0 ; i < depth ; i ++)
            res.append("--");
        return res.toString();
    }
}
```

```
public class BSTSet<E extends Comparable<E>> implements Set<E> {


    private BST<E> bst;


    public BSTSet(){
        bst = new BST<>();
    }


    @Override
    public int getSize(){
        return bst.size();
    }


    @Override
    public boolean isEmpty(){
        return bst.isEmpty();
    }


    @Override
    public void add(E e){
        bst.add(e);
    }


    @Override
    public boolean contains(E e){
        return bst.contains(e);
    }


    @Override
    public void remove(E e){
        bst.remove(e);
    }
}
```

```
import java.io.FileInputStream;
import java.util.ArrayList;
import java.util.Scanner;
import java.util.Locale;
import java.io.File;
import java.io.BufferedInputStream;
import java.io.IOException;


// 文件相关操作
public class FileOperation {


    // 读取文件名称为filename中的内容，并将其中包含的所有词语放进words中
    public static boolean readFile(String filename, ArrayList<String> words){


        if (filename == null || words == null){
            System.out.println("filename is null or words is null");
            return false;
        }


        // 文件读取
        Scanner scanner;


        try {
            File file = new File(filename);
            if(file.exists()){
                FileInputStream fis = new FileInputStream(file);
                scanner = new Scanner(new BufferedInputStream(fis), "UTF-8");
                scanner.useLocale(Locale.ENGLISH);
            }
            else
                return false;
        }
        catch(IOException ioe){
            System.out.println("Cannot open " + filename);
            return false;
        }


        // 简单分词
        // 这个分词方式相对简陋，没有考虑很多文本处理中的特殊问题
        // 在这里只做demo展示用
        if (scanner.hasNextLine()) {


            String contents = scanner.useDelimiter("\\A").next();


            int start = firstCharacterIndex(contents, 0);
            for (int i = start + 1; i <= contents.length(); )
                if (i == contents.length() ||
!Character.isLetter(contents.charAt(i))) {
                    String word = contents.substring(start, i).toLowerCase();
                    words.add(word);
                    start = firstCharacterIndex(contents, i);
                    i = start + 1;
                } else
                    i++;
        }


        return true;
    }


    // 寻找字符串s中，从start的位置开始的第一个字母字符的位置
    private static int firstCharacterIndex(String s, int start){


        for( int i = start ; i < s.length() ; i ++ )
            if( Character.isLetter(s.charAt(i)) )
                return i;
        return s.length();
    }
}
```

2.用链表实现集合set

```
public class LinkedList<E> {


    private class Node{
        public E e;
        public Node next;


        public Node(E e, Node next){
            this.e = e;
            this.next = next;
        }


        public Node(E e){
            this(e, null);
        }


        public Node(){
            this(null, null);
        }


        @Override
        public String toString(){
            return e.toString();
        }
    }


    private Node dummyHead;
    private int size;


    public LinkedList(){
        dummyHead = new Node();
        size = 0;
    }


    // 获取链表中的元素个数
    public int getSize(){
        return size;
    }


    // 返回链表是否为空
    public boolean isEmpty(){
        return size == 0;
    }


    // 在链表的index(0-based)位置添加新的元素e
    // 在链表中不是一个常用的操作，练习用：）
    public void add(int index, E e){


        if(index < 0 || index > size)
            throw new IllegalArgumentException("Add failed. Illegal index.");


        Node prev = dummyHead;
        for(int i = 0 ; i < index ; i ++)
            prev = prev.next;


        prev.next = new Node(e, prev.next);
        size ++;
    }
    // 在链表头添加新的元素e
    public void addFirst(E e){
        add(0, e);
    }


    // 在链表末尾添加新的元素e
    public void addLast(E e){
        add(size, e);
    }


    // 获得链表的第index(0-based)个位置的元素
    // 在链表中不是一个常用的操作，练习用：）
    public E get(int index){


        if(index < 0 || index >= size)
            throw new IllegalArgumentException("Get failed. Illegal index.");


        Node cur = dummyHead.next;
        for(int i = 0 ; i < index ; i ++)
            cur = cur.next;
        return cur.e;
    }


    // 获得链表的第一个元素
    public E getFirst(){
        return get(0);
    }


    // 获得链表的最后一个元素
    public E getLast(){
        return get(size - 1);
    }


    // 修改链表的第index(0-based)个位置的元素为e
    // 在链表中不是一个常用的操作，练习用：）
    public void set(int index, E e){
        if(index < 0 || index >= size)
            throw new IllegalArgumentException("Set failed. Illegal index.");


        Node cur = dummyHead.next;
        for(int i = 0 ; i < index ; i ++)
            cur = cur.next;
        cur.e = e;
    }


    // 查找链表中是否有元素e
    public boolean contains(E e){
        Node cur = dummyHead.next;
        while(cur != null){
            if(cur.e.equals(e))
                return true;
            cur = cur.next;
        }
        return false;
    }


    // 从链表中删除index(0-based)位置的元素，返回删除的元素
    // 在链表中不是一个常用的操作，练习用：）
    public E remove(int index){
        if(index < 0 || index >= size)
            throw new IllegalArgumentException("Remove failed. Index is
illegal.");
```

```
        Node prev = dummyHead;
        for(int i = 0 ; i < index ; i ++)
            prev = prev.next;


        Node retNode = prev.next;
        prev.next = retNode.next;
        retNode.next = null;
        size --;


        return retNode.e;
    }


    // 从链表中删除第一个元素, 返回删除的元素
    public E removeFirst(){
        return remove(0);
    }


    // 从链表中删除最后一个元素, 返回删除的元素
    public E removeLast(){
        return remove(size - 1);
    }


    // 从链表中删除元素e
    public void removeElement(E e){


        Node prev = dummyHead;
        while(prev.next != null){
            if(prev.next.e.equals(e))
                break;
            prev = prev.next;
        }


        if(prev.next != null){
            Node delNode = prev.next;
            prev.next = delNode.next;
            delNode.next = null;
            size --;
        }
    }


    @Override
    public String toString(){
        StringBuilder res = new StringBuilder();


        Node cur = dummyHead.next;
        while(cur != null){
            res.append(cur + "->");
            cur = cur.next;
        }
        res.append("NULL");


        return res.toString();
    }
}
```

```
import java.util.ArrayList;


public class LinkedListSet<E> implements Set<E> {


    private LinkedList<E> list;
```

```java
    public LinkedListSet(){
        list = new LinkedList<>();
    }


    @Override
    public int getSize(){
        return list.getSize();
    }


    @Override
    public boolean isEmpty(){
        return list.isEmpty();
    }


    @Override
    public void add(E e){
        if(!list.contains(e))
            list.addFirst(e);
    }


    @Override
    public boolean contains(E e){
        return list.contains(e);
    }


    @Override
    public void remove(E e){
        list.removeElement(e);
    }


    public static void main(String[] args) {


        System.out.println("Pride and Prejudice");


        ArrayList<String> words1 = new ArrayList<>();
        if(FileOperation.readFile("pride-and-prejudice.txt", words1)) {
            System.out.println("Total words: " + words1.size());


            LinkedListSet<String> set1 = new LinkedListSet<>();
            for (String word : words1)
                set1.add(word);
            System.out.println("Total different words: " + set1.getSize());
        }


        System.out.println();




        System.out.println("A Tale of Two Cities");


        ArrayList<String> words2 = new ArrayList<>();
        if(FileOperation.readFile("a-tale-of-two-cities.txt", words2)){
            System.out.println("Total words: " + words2.size());


            LinkedListSet<String> set2 = new LinkedListSet<>();
            for(String word: words2)
                set2.add(word);
            System.out.println("Total different words: " + set2.getSize());
        }
    }
```

```
}
```

3.映射的定义

```
public interface Map<K, V> {
    void add(K key, V value);
    V remove(K key);
    boolean contains(K key);
    V get(K key);
    void set(K key, V newValue);
    int getSize();
    boolean isEmpty();
}
```

4.用链表实现映射

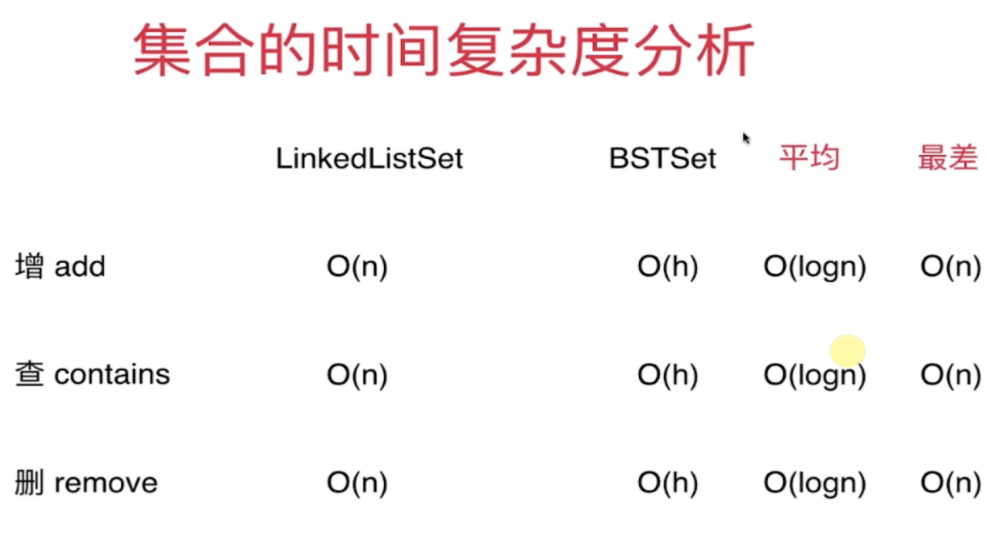
```
import java.util.ArrayList;


public class LinkedListMap<K, V> implements Map<K, V> {


    private class Node{
        public K key;
        public V value;
        public Node next;


        public Node(K key, V value, Node next){
            this.key = key;
            this.value = value;
            this.next = next;
        }


        public Node(K key, V value){
            this(key, value, null);
        }


        public Node(){
            this(null, null, null);
        }


        @Override
        public String toString(){
            return key.toString() + " : " + value.toString();
        }
    }


    private Node dummyHead;
    private int size;


    public LinkedListMap(){
        dummyHead = new Node();
        size = 0;
    }


    @Override
    public int getSize(){
        return size;
    }


    @Override
    public boolean isEmpty(){
        return size == 0;
    }
    private Node getNode(K key){
        Node cur = dummyHead.next;
        while(cur != null){
            if(cur.key.equals(key))
                return cur;
            cur = cur.next;
        }
        return null;
    }


    @Override
    public boolean contains(K key){
        return getNode(key) != null;
    }


    @Override
    public V get(K key){
        Node node = getNode(key);
        return node == null ? null : node.value;
    }


    @Override
    public void add(K key, V value){
        Node node = getNode(key);
        if(node == null){
            dummyHead.next = new Node(key, value, dummyHead.next);
            size ++;
        }
        else
            node.value = value;
    }


    @Override
    public void set(K key, V newValue){
        Node node = getNode(key);
        if(node == null)
            throw new IllegalArgumentException(key + " doesn't exist!");


        node.value = newValue;
    }


    @Override
    public V remove(K key){


        Node prev = dummyHead;
        while(prev.next != null){
            if(prev.next.key.equals(key))
                break;
            prev = prev.next;
        }


        if(prev.next != null){
            Node delNode = prev.next;
            prev.next = delNode.next;
            delNode.next = null;
            size --;
            return delNode.value;
        }


        return null;
    }


    public static void main(String[] args){
        System.out.println("Pride and Prejudice");


        ArrayList<String> words = new ArrayList<>();
        if(FileOperation.readFile("pride-and-prejudice.txt", words)) {
            System.out.println("Total words: " + words.size());


            LinkedListMap<String, Integer> map = new LinkedListMap<>();
            for (String word : words) {
                if (map.contains(word))
                    map.set(word, map.get(word) + 1);
                else
                    map.add(word, 1);
            }


            System.out.println("Total different words: " + map.getSize());
            System.out.println("Frequency of PRIDE: " + map.get("pride"));
            System.out.println("Frequency of PREJUDICE: " +
map.get("prejudice"));
        }


        System.out.println();
    }
}
```

5.用二分搜索树实现映射

```
import java.util.ArrayList;


public class BSTMap<K extends Comparable<K>, V> implements Map<K, V> {


    private class Node{
        public K key;
        public V value;
        public Node left, right;


        public Node(K key, V value){
            this.key = key;
            this.value = value;
            left = null;
            right = null;
        }
    }


    private Node root;
    private int size;


    public BSTMap(){
        root = null;
        size = 0;
    }


    @Override
    public int getSize(){
        return size;
    }


    @Override
    public boolean isEmpty(){
        return size == 0;
    }
```

```
// 向二分搜索树中添加新的元素(key, value)
@Override
public void add(K key, V value){
    root = add(root, key, value);
}


// 向以node为根的二分搜索树中插入元素(key, value)，递归算法
// 返回插入新节点后二分搜索树的根
private Node add(Node node, K key, V value){


    if(node == null){
        size ++;
        return new Node(key, value);
    }


    if(key.compareTo(node.key) < 0)
        node.left = add(node.left, key, value);
    else if(key.compareTo(node.key) > 0)
        node.right = add(node.right, key, value);
    else // key.compareTo(node.key) == 0
        node.value = value;


    return node;
}


// 返回以node为根节点的二分搜索树中，key所在的节点
private Node getNode(Node node, K key){


    if(node == null)
        return null;


    if(key.equals(node.key))
        return node;
    else if(key.compareTo(node.key) < 0)
        return getNode(node.left, key);
    else // if(key.compareTo(node.key) > 0)
        return getNode(node.right, key);
}


@Override
public boolean contains(K key){
    return getNode(root, key) != null;
}


@Override
public V get(K key){


    Node node = getNode(root, key);
    return node == null ? null : node.value;
}


@Override
public void set(K key, V newValue){
    Node node = getNode(root, key);
    if(node == null)
        throw new IllegalArgumentException(key + " doesn't exist!");


    node.value = newValue;
}
```

```
    // 返回以node为根的二分搜索树的最小值所在的节点
    private Node minimum(Node node){
        if(node.left == null)
            return node;
        return minimum(node.left);
    }


    // 删除掉以node为根的二分搜索树中的最小节点
    // 返回删除节点后新的二分搜索树的根
    private Node removeMin(Node node){


        if(node.left == null){
            Node rightNode = node.right;
            node.right = null;
            size --;
            return rightNode;
        }


        node.left = removeMin(node.left);
        return node;
    }


    // 从二分搜索树中删除键为key的节点
    @Override
    public V remove(K key){


        Node node = getNode(root, key);
        if(node != null){
            root = remove(root, key);
            return node.value;
        }
        return null;
    }


    private Node remove(Node node, K key){


        if( node == null )
            return null;


        if( key.compareTo(node.key) < 0 ){
            node.left = remove(node.left , key);
            return node;
        }
        else if(key.compareTo(node.key) > 0 ){
            node.right = remove(node.right, key);
            return node;
        }
        else{   // key.compareTo(node.key) == 0


            // 待删除节点左子树为空的情况
            if(node.left == null){
                Node rightNode = node.right;
                node.right = null;
                size --;
                return rightNode;
            }


            // 待删除节点右子树为空的情况
            if(node.right == null){
                Node leftNode = node.left;
                node.left = null;
                size --;
                return leftNode;
            }
            // 待删除节点左右子树均不为空的情况


            // 找到比待删除节点大的最小节点, 即待删除节点右子树的最小节点
            // 用这个节点顶替待删除节点的位置
            Node successor = minimum(node.right);
            successor.right = removeMin(node.right);
            successor.left = node.left;


            node.left = node.right = null;


            return successor;
        }
    }


    public static void main(String[] args){


        System.out.println("Pride and Prejudice");


        ArrayList<String> words = new ArrayList<>();
        if(FileOperation.readFile("pride-and-prejudice.txt", words)) {
            System.out.println("Total words: " + words.size());


            BSTMap<String, Integer> map = new BSTMap<>();
            for (String word : words) {
                if (map.contains(word))
                    map.set(word, map.get(word) + 1);
                else
                    map.add(word, 1);
            }


            System.out.println("Total different words: " + map.getSize());
            System.out.println("Frequency of PRIDE: " + map.get("pride"));
            System.out.println("Frequency of PREJUDICE: " +
map.get("prejudice"));
        }


        System.out.println();
    }
}
```



集合的时间复杂度分析

	LinkedListSet	BSTSet	平均	最差
增 add	O(n)	O(h)	O(logn)	O(n)
查 contains	O(n)	O(h)	O(logn)	O(n)
删 remove	O(n)	O(h)	O(logn)	O(n)