数据结构与算法6-链表和递归

笔记本: 我的笔记
创建时间: 2020/10/3 9:25 **更新时间:** 2020/10/3 10:05
作者: liuhouer
标签: 算法
URL: https://github.com/liuhouer/np-algorithm/blob/master/08x-Linked-List/03-L...

1.自己实现一个链表

```
public class LinkedList<E> {

    //链表结构的定义
    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();
    }
}
```

2.用链表实现栈 【相应的栈结构在上一章】

```
public class LinkedListStack<E> implements Stack<E> {


    private LinkedList<E> list;


    public LinkedListStack(){
        list = new LinkedList<>();
    }


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


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


    @Override
    public void push(E e){
        list.addFirst(e);
    }


    @Override
    public E pop(){
        return list.removeFirst();
    }


    @Override
    public E peek(){
        return list.getFirst();
    }


    @Override
    public String toString(){
        StringBuilder res = new StringBuilder();
        res.append("Stack: top ");
        res.append(list);
        return res.toString();
    }


    public static void main(String[] args) {


        LinkedListStack<Integer> stack = new LinkedListStack<>();


        for(int i = 0 ; i < 5 ; i ++){
            stack.push(i);
            System.out.println(stack);
        }


        stack.pop();
        System.out.println(stack);
```

```
    }
}
```

3.用链表实现队列

```
public class LinkedListQueue<E> implements Queue<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 head, tail;
    private int size;


    public LinkedListQueue(){
        head = null;
        tail = null;
        size = 0;
    }


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


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


    @Override
    public void enqueue(E e){
        if(tail == null){
            tail = new Node(e);
            head = tail;
        }
        else{
            tail.next = new Node(e);
            tail = tail.next;
        }
        size ++;
    }
```

```
    @Override
    public E dequeue(){
        if(isEmpty())
            throw new IllegalArgumentException("Cannot dequeue from an empty
queue.");


        Node retNode = head;
        head = head.next;
        retNode.next = null;
        if(head == null)
            tail = null;
        size --;
        return retNode.e;
    }


    @Override
    public E getFront(){
        if(isEmpty())
            throw new IllegalArgumentException("Queue is empty.");
        return head.e;
    }


    @Override
    public String toString(){
        StringBuilder res = new StringBuilder();
        res.append("Queue: front ");


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


    public static void main(String[] args){


        LinkedListQueue<Integer> queue = new LinkedListQueue<>();
        for(int i = 0 ; i < 10 ; i ++){
            queue.enqueue(i);
            System.out.println(queue);


            if(i % 3 == 2){
                queue.dequeue();
                System.out.println(queue);
            }
        }
    }
}
```

链表的性能问题

虽然猛地看上去，如果我们只在链表头添加元素，时间复杂度是 O(1) 的。同时，因为使用链表不需要 ``ressize``，所以，凭直觉，链表的性能应该更好。

但实际上，当数据量达到一定程度，链表的性能是更差的。

这是因为，对于链表来说，每添加一个元素，都需要重新创建一个 Node 类的对象，也就是都需要进行一次 new 的内存操作。而**对内存的操作，是非常慢的。**

铜须门可以尝试一下，对于我们这个课程中所实现 ``Array`` 类和 ``LinkedList`` 类，进行如下的测试：

```
// 创建一个动态数组，再创建一个链表
Array<Integer> array = new Array<>();
LinkedList<Integer> list = new LinkedList<>();

// 对于 1000 万规模的数据
int n = 10000000;
System.out.println("n = " + n);

// 计时，看将 1000 万个元素放入数组中，时间是多少
long startTime = System.nanoTime();
// 对于数组，我们使用 addLast，每一次操作时间复杂度都是 O(1) 的
for(int i = 0; i < n; i ++)
    array.addLast(i);
long endTime = System.nanoTime();
double time = (endTime - startTime) / 1000000000.0;
System.out.println("Array : " + time + " s");

// 计时，看将 1000 万个元素放入链表中，时间是多少
startTime = System.nanoTime();
// 对于链表，我们使用 addFirst，每一次操作时间复杂度都是 O(1) 的
for(int i = 0; i < n; i ++)
    list.addFirst(i);
endTime = System.nanoTime();
time = (endTime - startTime) / 1000000000.0;
System.out.println("LinkedList : " + time + " s");
```

在我的计算机上，结果是这样的：

```
n = 10000000
Array : 0.203133984 s
LinkedList : 3.418495718 s
```

可以看出来，使用链表明显慢于使用动态数组。

为什么即使有 ``resize``，对于大规模数据，动态数组还是会快于链表？

因为对于动态数组来说，一方面，每次 ``resize`` 容量增倍，这将使得，对于大规模数据，实际上触发 ``resize`` 的次数是非常少的。

更重要的是，``resize`` 的过程，试一次申请一大片内存空间。但是对于链表来说，每次只是申请一个空间。

申请一次 10 万的空间，是远远快于申请 10 万次 1 的空间的。而相较于堆内存空间的操作，动态数组的 ``resize`` 过程虽然还需要赋值，把旧数组的元素拷贝给新数组。但是这个拷贝过程，是远远快于对内存的操作的。

链表的时间复杂度分析

- 增： O(n)
- 删： O(n)

→ 如果只对链表头进行操作：O(1)

- 改： O(n)
- 查： O(n) → 只查链表头的元素：O(1)

递归

- 本质上，将原来的问题，转化为更小的同一问题
- 举例：数组求和

Sum(arr[0...n-1]) = arr[0] + Sum(arr[1...n-1]) ← 更小的同一问题

Sum(arr[1...n-1]) = arr[1] + Sum(arr[2...n-1]) ← 更小的同一问题

.....

Sum(arr[n-1...n-1]) = arr[n-1] + Sum([]) ← 最基本的问题

4.用递归解决累加的问题

```
public class Sum {


    public static int sum(int[] arr){
        return sum(arr, 0);
    }


    // 计算arr[l...n)这个区间内所有数字的和
    private static int sum(int[] arr, int l){
        if(l == arr.length)
            return 0;
        return arr[l] + sum(arr, l + 1);
    }


    public static void main(String[] args) {


        int[] nums = {1, 2, 3, 4, 5, 6, 7, 8};
        System.out.println(sum(nums));
    }
}
```

5.用递归实现链表【更简洁】

```
import javafx.util.Pair;
```

```
/// 递归实现的LinkedList
/// 类名称中LinkedListR里的R，是Recursion的意思，表示递归实现：）
public class LinkedListR<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();
        }
    }


    // 在链表的递归实现中，我们不使用虚拟头结点，也能无差异的处理位置0的问题：）
    private Node head;
    private int size;


    public LinkedListR(){
        head = null;
        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
    public void add(int index, E e){


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


        head = add(head, index, e);
        size ++;
    }


    // 在以node为头结点的链表的index位置插入元素e，递归算法
    private Node add(Node node, int index, E e){
```

```
        if(index == 0)
            return new Node(e, node);


        node.next = add(node.next, index - 1, e);
        return node;
    }


    // 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
    public E get(int index){


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


        return get(head, index);
    }


    // 在以node为头结点的链表中，找到第index个元素，递归算法
    private E get(Node node, int index){
        if(index == 0)
            return node.e;
        return get(node.next, index - 1);
    }


    // 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
    public void set(int index, E e){
        if(index < 0 || index >= size)
            throw new IllegalArgumentException("Update failed. Illegal
index.");


        set(head, index, e);
    }


    // 修改以node为头结点的链表中，第index(0-based)个位置的元素为e，递归算法
    private void set(Node node, int index, E e){
        if(index == 0){
            node.e = e;
            return;
        }
        set(node.next, index - 1, e);
    }


    // 查找链表中是否有元素e
```

```
    public boolean contains(E e){
        return contains(head, e);
    }


    // 在以node为头结点的链表中，查找是否存在元素e，递归算法
    private boolean contains(Node node, E e){
        if(node == null)
            return false;
        if(node.e.equals(e))
            return true;
        return contains(node.next, e);
    }


    // 从链表中删除index(0-based)位置的元素，返回删除的元素
    public E remove(int index){
        if(index < 0 || index >= size)
            throw new IllegalArgumentException("Remove failed. Index is
illegal.");


        Pair<Node, E> res = remove(head, index);
        size --;
        head = res.getKey();
        return res.getValue();
    }


    // 从以node为头结点的链表中，删除第index位置的元素，递归算法
    // 返回值包含两个元素，删除后的链表头结点和删除的值：）
    private Pair<Node, E> remove(Node node, int index){
        if(index == 0)
            return new Pair<>(node.next, node.e);
        Pair<Node, E> res = remove(node.next, index - 1);
        node.next = res.getKey();
        return new Pair<>(node, res.getValue());
    }


    // 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){


        head = removeElement(head, e);
    }


    // 从以node为头结点的链表中，删除元素e，递归算法
    private Node removeElement(Node node, E e){
        if(node == null)
            return null;
        if(node.e.equals(e)){
            size --;
            return node.next;
        }
        node.next = removeElement(node.next, e);
        return node;
    }


    @Override
    public String toString(){
```

```
        StringBuilder res = new StringBuilder();


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


        return res.toString();
    }


    public static void main(String[] args) {


        LinkedListR<Integer> list = new LinkedListR<>();
        for(int i = 0 ; i < 10 ; i ++)
            list.addFirst(i);


        while(!list.isEmpty())
            System.out.println("removed " + list.removeLast());
    }
}
```

6.用递归翻转一个链表

```
public ListNode reverseList(ListNode head) {


if(head == null || head.next == null)
return head;

ListNode rev = reverseList(head.next);
head.next.next = head;
head.next = null;
return rev;
}
```

7.循环翻转一个链表【非递归】

```
public ListNode reverseList(ListNode head) {

ListNode pre = null;
ListNode cur = head;
while(cur != null){
ListNode next = cur.next;
cur.next = pre;
pre = cur;
cur = next;
}
return pre;
}
```

斯坦福大学的 18 个链表问题

在这一小节，我将把上一小节视频中，我推荐的斯坦福大学的链表相关文档的 18 个问题进行简单的翻译。

1\. ``Count`` : 计算链表的节点个数；

2\. ``GetNth`` : 获得链表第 n 个节点的值；

3\. ``CreateList`` : 根据数组（或者标准输入）创建链表；``DeleteList`` : 释放一个链表的所有节点空间；

注意：由于 Java 语言不需要释放空间，所以在 Java 语言中，不需要完成这个任务。如果有同学使用 C/C++，可以参考）；

4\. ``Push`` : 向链表头插入一个新节点；``Pop`` : 删除链表的第一个节点并返回；

注意：我们可以将链表看做是一个队列，此时，向链表的头或者尾插入或者删除元素，就可以衍生出两个 ``Push`` 实现和两个 ``Pop`` 实现。大家也可以显示地将这四个方法命名为 ``addFirst``, ``addLast``, ``removeFirst``, ``removeLast``（参考 Java 中的命名方式），并据此封装底层基于链表的栈，队列，双向队列等等线性数据结构：）

5\. ``InsertNth`` : 在链表的第 n 个位置插入一个新节点；

6\. ``SortedInsert`` : 给定一个有序链表，将一个新节点插入到有序链表的正确位置；

7\. ``InsertSort`` : 使用插入排序法为链表排序；

提示：之前实现的 ``SortedInsert`` 有用了：）

8\. ``Append`` : 挂接两个链表；

9\. ``FrontBackSplit`` : 将一个链表分割成大小相等的两个链表（对于原链表大小为奇数的情况，分割为大小只相差 1 的两个链表）；

10\. ``RemoveDuplicates`` : 给定一个有序链表，其中含有重复节点，删除链表中的重复节点，使得每个不同值的节点只有一个；

11\. ``MoveNode`` : 给定两个链表，``Pop`` 出第二个链表的元素，``Push`` 进第一个链表；

注意：这里的 ``Pop`` 和 ``Push`` 可以根据实际场景使用 4. 中的任意一组定义

12\. ``AlternatingSplit`` : 给定一个链表，将他分割成两个链表，其中奇数位置的节点在一个链表，偶数位置的节点在另一个链表；

提示：之前实现的 ``MoveNode`` 有用了：）

13\. ``ShuffleMerge`` : 给定两个链表，将这两个链表合并成一个链表，其中一个链表的元素在奇数位，另一个链表的元素在偶数位；

提示：之前实现的 ``MoveNode`` 有用了：）

14\. ``SortedMerge`` : 给定两个有序链表，将他们合并成为一个有序链表；

提示：之前实现的 ``MoveNode`` 有用了：）

注意，对于这个需求，看完下一章归并排序后再回头来做，可能更有感觉：）

15\. ``MergeSort`` : 对链表进行归并排序

注意：关于归并排序，下一章会进行详细介绍。看完下一章后，可以再回头来解决这个问题：）

提示：实现了 ``FrontBackSplit`` 和 ``SortedMerge`` 后，应该觉得这个问题并不难：）

16\. ``SortedIntersect`` : 给定两个有序链表，返回一个新链表，新链表中的元素是给定两个链表的公共元素。

提示：以这个方法为基础，可以封装基于链表的集合类。关于集合的介绍，大家在后续讲解二分搜索树的时候，会详细了解。可以届时再回头来看这个问题：）

17\. ``Reverse`` : 反转一个链表

18\. ``RecursiveReverse`` : 使用递归的方式反转一个链表

最后特意将递归方式翻转链表列出来，是因为这个问题实在是太经典了，充分体现了和链表相关算法的美丽。在下一章，是关于链表问题的一个习题章节，届时，我会仔细介绍一下反转链表这个经典问题：）

