

Linear List

1	Array	3
1.1	Overview of Array	3
1.2	Practice: To create an Array Class with Java	3
2	LinkedList	7
2.1	Overview of LinkedList	7
2.2	LinkedList In Java	7
2.3	MikeLinkedList 的搭建 1: 完成 Node Class 的实现与 MikeLinkedList 的基础	8
2.4	MikeLinkedList 的搭建 2: 实现具体的方法	8
2.5	练习题 1: 反转链表	12
2.6	练习题 2: 寻找倒数第 K 个链表节点	13
3	Stack	15
3.1	Overview of Stack	15
3.2	Stack in Java	15
3.3	MikeStack 的实现	16
3.4	练习题 1: 反转字符串	17
3.5	练习题 2: 判断表达式的平衡	18
4	Tree	21
4.1	Binary Search Tree(BST)	21
4.2	Build a tree	21
4.3	Implement the method: Insert	22
4.4	Implement the method: find	23
4.5	Traverse the tree	24
4.6	Depth and height of node	26
4.7	Min Value in a tree	27
4.8	Exercise: Equality Checking	27
4.9	Exercise: isBinarySearchTree	28
4.10	Exercise: Node at K Distance	29
4.11	Exercise: Level Traverse	29



1. Array

1.1 Overview of Array

在本节中，我们首先来思考一下数组是什么？

3	1	4	5	6	7
---	---	---	---	---	---

这是一个数组:[3,1,4,5,6,7], 它本质上就是在一段连续的内存片段上存储的一串数据, 由于 int 占了四个字节, 所以每一个数组元素的地址相差四位。

接下来分析数组中常见的操作的算法复杂度:

- Lookup $O(1)$ 根据索引去查找元素, 由于数组中可以通过中括号访问符进行访问, 所以不论访问哪一个元素, 算法复杂度均为 $O(1)$
- Insert $O(n)$ 进行元素插入, 如果在末尾进行元素插入, 算法复杂度为 $O(1)$, 但是在第一位插入, 我们就需要将每一位都往后挪动一位, 所以复杂度为 $O(n)$, 通常我们都考虑最坏的情况, 所以算法复杂度为 $O(n)$
- Delete $O(n)$ 无论删除哪一个元素, 其他元素都需要往前移动一位, 所以最终算法复杂度为 $O(n)$

1.2 Practice: To create an Array Class with Java

■ **Example 1.1** 在 Java 中, 创建一个 MikeArray 类型, 要求要满足以下几点, 1. 具有 insert 方法, 可以在数组末尾插入元素, 2. 具有 removeAt(int index) 方法, 在 index 处删除对应的元素, 3. 实现 print 方法, 打印出数组中所有的元素 4. indexOf 方法, 返回目标元素第一次出现在数组中的索引, 如果没有出现元素, 则返回索引-1, 4. 动态扩充, 在使用 insert 方法的时候如果超出了数组本身的长度, 扩容到原数组容量的 2 倍, 5. 构造方法是给出数组的长度 length 进行数组构造的。 ■

```
public class MArray {  
    private int[] items;  
    private int count=0;  
    public MArray(int length){  
        items=new int[length];  
    }  
  
    public void print(){  
        for(int i=0;i<count;i++){  
            System.out.println(items[i]);  
        }  
    }  
  
    public void insert(int element){  
        //if the array is full  
        if(items.length==count){  
            //Create a new array (twice the size)  
            int array[]=new int[count*2];  
            //Copy all the items into new Array  
            for(int i=0;i<count;i++){  
                array[i]=items[i];  
            }  
            //set the items to the new array  
            items=array;  
        }  
        items[count++]=element;  
    }  
  
}
```

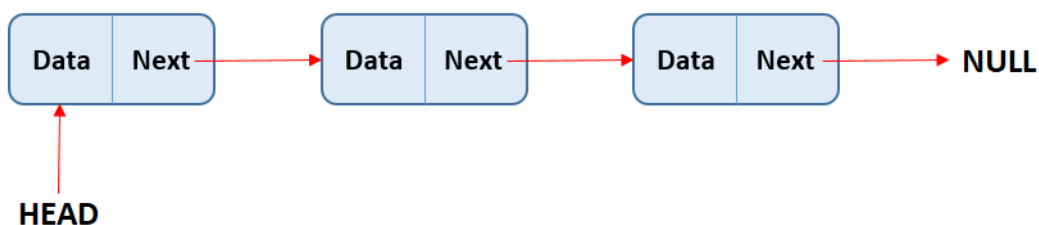
```
        public void removeAt(int index){
            if(index<0||index>=count){
                throw new IllegalArgumentException();
            }
            else{
                //[10,20,30,40]
                // index:1
                for(int i=index;i<count;i++){
                    items[i]=items[i+1];
                }
                count--;
            }
        }

        public int indexOf(int item){
            //获取元素的位置
            //肯定是有可能遍历完整个都没有存在的
            for(int i=0;i<count;i++){
                if(items[i]==item){
                    return i;
                }
            }
            return -1;
        }
    }
```


2. LinkedList

2.1 Overview of LinkedList

链表是由一串节点 (Node) 连接而成, 这个节点由两部分组成, 一部分是数据 Data, 另外一部分是指针 Pointer, 每一个 Node 的指针都指向了下一个节点。一般的链表具有头节点, 当然也有的具有尾节点, 除了一般的单向链表, 还有一种双向链表, 在双向链表中每一个节点都具有两个指针, 一个指向之前的节点, 一个指向了之后的节点。但是总体上来说, 链表的结构都大同小异。



2.2 LinkedList In Java

在 Java 中就有 LinkedList, 常用的方法有:

- addFirst 在链表首添加元素
- addLast 在链表尾部添加元素
- contains 是否包含元素
- isEmpty 判断链表是否为空
- indexOf 返回链表中某一元素的位置

这些方法都是由 Java 内部进行实现的, 我们直接引入 LinkedList 这个类即可直接使用, 当然在下一节中, 我们将开始写属于我们自己的链表 (MikeLinkedList)。


```
LinkedList<Integer>linkedList=new LinkedList<>();
linkedList.addFirst(10);
linkedList.addFirst(10);
linkedList.addFirst(10);
System.out.println(linkedList.indexOf(10));
System.out.println(linkedList.contains(10));
System.out.println(linkedList.isEmpty());
linkedList.addLast(29);
System.out.println(linkedList);
```

2.3 MikeLinkedList 的搭建 1: 完成 Node Class 的实现与 MikeLinkedList 的基础

在第一节中我们说明了链表是由一个一个节点组成的，所以首先实现一个 Node Class(Integer 整型)，使用整数进行示范，不设置为泛型。

```
public class Node {
    public Node(int value){
        this.value=value;
    }

    public int value;//当前节点的值
    public Node next;//指向下一个节点的指针
}
```

然后完成 MikeLinkedList 的搭建，这个链表首先要两个节点，一个是首节点，一个是尾节点，还需要设置一个 size 属性，记录链表的长度。

```
public class MikeLinkedList{
    //头节点 p
    private Node First;
    private Node Last;
    private int Size;
}
```

2.4 MikeLinkedList 的搭建 2: 实现具体的方法

在实现过程中，要考虑极限情况，没有元素的时候，只有一个元素的时候。

2.4.1 addFirst

实现方法很简单，新建一个节点，将它标记为头节点，然后把它的 next 指针指向原本的头节点即可，算法复杂度 $O_n = 1$

```
public void addFirst(int value) {  
    var node = new Node(value);  
    //当前链表是否为空  
    if (isEmpty()) {  
        First = Last = node;  
    } else {  
        var temp = First;  
        First = node;  
        First.next = temp;  
        size++;  
    }  
}
```

2.4.2 addLast

addLast 方法也非常简单，只需要设置将最后一个节点的 next 指向新节点，然后将 Last 标记为最后一个节点即可，算法复杂度为 $O_n = 1$

```
public void addLast(int value) {  
    var node = new Node(value);  
    if (isEmpty()) {  
        First = Last = node;  
    } else {  
        Last.next = node;  
        Last = node;  
    }  
    size++;  
}
```

2.4.3 isEmpty

isEmpty 只需要判断头节点是否为空，如果为空的话就说明这个链表为空，非常简单

```
private boolean isEmpty() {  
    return First == null;  
}
```

2.4.4 indexOf

实现 `indexOf` 方法只需要从链表头开始遍历，直到发现某个节点的 `value` 和目标节点的 `value` 是一样的，如果一样就返回对应的 `index`，如果遍历完整个链表都没有发现目标，则返回-1。由于在最坏的情况需要完整遍历一遍链表，所以算法复杂度为 $O_n = n$ 。

```
public int indexOf(int value) {  
    if (isEmpty()) {  
        return -1;  
    } else {  
        var cur = First;  
        var index = 0;  
        while (cur != null) {  
            if (cur.value == value) {  
                return index;  
            }  
            index++;  
            cur = cur.next;  
        }  
    }  
    return -1;  
}
```

2.4.5 contains

`contains` 方法的实现在 `indexOf` 写过之后就变得非常简单，只需要执行一次 `indexOf` 方法，如果返回的是-1，则表示不包含，返回 `false`，否则返回 `true`

```
public boolean contains(int value) {  
    return indexOf(value) != -1;  
}
```

2.4.6 removeFirst

`removeFirst` 方法的核心是将第二个节点设置为 `First`，然后将第一个节点解除关联。

```
public void removeFirst() {  
    if (isEmpty()) {  
        throw new NoSuchElementException();  
    }  
    if (First.next == null) {  
        First = Last = null;  
        return;  
    } else {  
        var second = First.next;  
        First.next = null;  
        First = second;  
    }  
    size--;  
}
```

2.4.7 removeLast

要实现 removeLast 稍微比 removeFirst 难一些，因为我们的思路应该还是找到 Last 之前的那个节点，然后把最后一个节点引用解除，设置倒数第二个节点为 Last 即可。但是如何获取倒数第二个节点？我们可以先写一个 getPrevious 的函数，然后再写这个函数

```
public Node getPrevious(Node node) {  
    var current = First;  
    while (current != null) {  
        if (current.next == node) {  
            return current;  
        }  
        current = current.next;  
    }  
    return null;  
}
```

```
public void removeLast() {  
    if (isEmpty()) {  
        throw new NoSuchElementException();  
    } else if (First == Last) {  
        First = Last = null;  
    } else {  
        var previous = getPrevious(Last);  
        Last = previous;  
        Last.next = null;  
    }  
    size--;  
}
```

2.4.8 toArray

将链表转为数组，方法很简单，遍历一遍，把链表的元素装数组里即可

```
public int[] toArray() {  
    int[] array = new int[size];  
    var cur = First;  
    var index = 0;  
    while (cur != null) {  
        array[index] = cur.value;  
        cur = cur.next;  
        index++;  
    }  
    return array;  
}
```

2.5 练习题 1: 反转链表

■ **Example 2.1** 将链表反转，比如 10→20→30→40→50 反转为 50→40→30→20→10。仔细思考，该题并不难，我们发现其实就是将链表的箭头顺序反转了，本质上就是 10←20←30←40←50，每一个指针都指向了之前的指针，所以我们只需要构造一个 previous 指针指向前一个节点即可

```
        public void reverse() {  
            //first second  
            //previous current  
            var previous = First;  
            var current = First.next;  
            Last = First;  
            Last.next = null;  
            while (current != null) {  
                var next = current.next;  
                current.next = previous;  
                previous = current;  
                current = next;  
            }  
            First = previous;  
        }  
    }
```

■

2.6 练习题 2: 寻找倒数第 K 个链表节点

■ **Example 2.2** 题目要求不允许直接遍历一整遍链表，然后获取对应的节点，我们使用快慢双指针来解决问题：快指针在一开始的时候就进行向前推进，当推进了 K 步之后慢指针开始运行，当快指针遍历完链表的时候，慢指针指向的节点就是对应的倒数第 K 个节点。

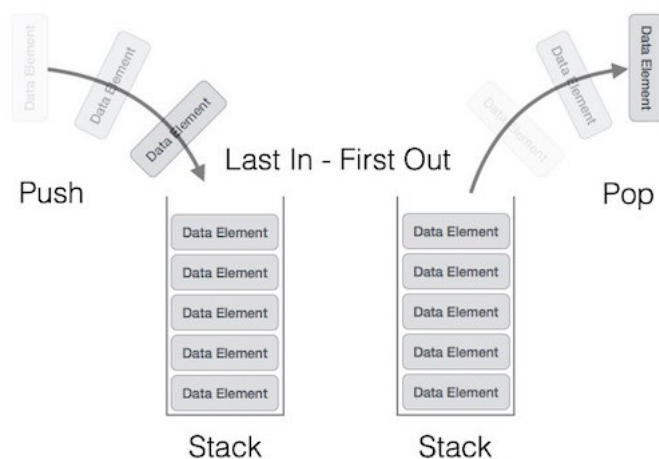
■

```
public int getKthFromTheEnd(int k) {  
    if (isEmpty()) {  
        throw new IllegalStateException();  
    }  
    if (k > size || k <= 0) {  
        throw new IllegalArgumentException();  
    }  
    //声明快指针  
    var a = First;  
    //声明慢指针  
    var b = First;  
    for (int i = 0; i < k - 1; i++) {  
        //先让 a 走 k 步  
        a = a.next;  
    }  
    while (a != Last) {  
        a = a.next;  
        b = b.next;  
    }  
    return b.value;  
}
```

3. Stack

3.1 Overview of Stack

栈是一种很常见的数据结构, 它遵循先入后出 (FILO) 的原则, 先入栈的元素反而最后出栈, 所以对于一些需要进行反转数据顺序的操作使用栈是非常适合的。



举一个非常通俗易懂的例子: 吃薯片。我们按照顺序向薯片口袋装入薯片, 先吃掉的反而是最后装入的薯片。还有 web 开发中的路由跳转, 每次跳转路由, 我们都将对应的路由压入栈中, 当我们要进行路由回退操作的时候, 回退的都是最后压入的路由

3.2 Stack in Java

在 Java 中有对应的 Stack 类型, 它主要的方法是 push 入栈, pop 出栈, peek 方法用于查看位于栈顶的元素, 但是不会弹出, empty 方法用于判断这个 stack 是否为空


```
Stack<Integer> stack = new Stack<>();
stack.push(1);
stack.push(2);
stack.push(3);
System.out.println(stack); //[1,2,3]
System.out.println(stack.pop()); //3
System.out.println(stack.peek()); //2
System.out.println(stack.empty()); //false
```

3.3 MikeStack 的实现

MikeStack 的内部，我们使用数组来实现，但是使用数组来实现难免会出现一些问题。问题一：Java 实现的 Stack 不会出现栈满的问题，但是我们使用数组来实现，必然会会遇到数组元素占满，为了简便不对数组进行复制然后扩充，而是抛出一个异常状态。问题二：使用数组实现，在数组没有填满的时候，比如容量为 5，填充三个元素的时候，输出必然为 [1,2,3,0,0]，这样是不像 java 中栈的实现，所以我们使用一个 counter 计数器，只输出真正有值的部分。gg 对于 push,pop,peek 方法都需要进行临界情况的判定，如果 count==0 的时候，我们需要抛出一个 `IllegalStateException` 异常。

```
public class MikeStack {
    private final int[] array=new int[5];
    private int pointer=0;
}
```

```
public void push(int item){
    if(pointer>=array.length){
        throw new IllegalStateException();
    }
    array[pointer++]=item;
}
public int pop(){
    if(pointer==0){
        throw new IllegalStateException();
    }
    return array[--pointer];
}
public int peek(){
    if(pointer==0){
        throw new IllegalStateException();
    }
    return array[pointer-1];
}
public String toArray(){
    return Arrays.toString(Arrays.copyOfRange(array,0,pointer));
}
```

3.4 练习题 1: 反转字符串

■ **Example 3.1** 现在有一个字符串 Hello,World, 我们需要写一个方法来反转这个字符串。由于在前文中提过, 由于 Stack 的特性, 很容易进行一些反转操作, 所以我们使用 Stack 来完成这一题

```
public static String reverseString(String str){
    if(str==null){
        throw new IllegalArgumentException();
    }
    var stack=new Stack<Character>();
    for(char c :str.toCharArray()){
        stack.push(c);
    }
    StringBuilder stringBuilder=new StringBuilder("");
    while(!stack.empty()){
        stringBuilder.append(stack.pop());
    }
    return stringBuilder.toString();
}
```

从入栈到出栈就会导致数据以相反的顺序输出出来，所以我们只需要将数据压入栈中，最后再使用 pop 依次出栈即可。 ■

3.5 练习题 2: 判断表达式的平衡

■ **Example 3.2** 该题相较于上题来说，比较复杂。表达式中的符号有 {,(,<,[一共四种符号。

- {1+2+3} 平衡
- >1231< 不平衡
- <{1+2+3}> 平衡
- <{1+2}} 不平衡

一次性考虑所有的情况，比较复杂，所以我们首先考虑只有 () 的情况，遇到开始符号入栈，遇到结尾符号出栈，最后栈里是空的，那么就平衡。在考虑清楚单一情况之后，我们考虑所有的情况，可以将这些符号分别录入两个数组，但是一定要注意顺序对应，两个数组 AB,A[1] 和 B[1] 才是完全对应的。

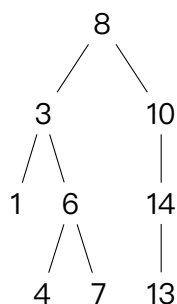
```
public class Expression {
    private List<Character> leftBracket =
        Arrays.asList('(', '<', '[', '{');
    private List<Character> rightBracket =
        Arrays.asList(')', '>', ']', '}');
    String str;

    public Expression(String str) {
        this.str = str;
    }
    private boolean isLeftBracket(char c){
        return leftBracket.contains(c);
    }
    private boolean matchBracket(char left, char top){
        return leftBracket.indexOf(left)==rightBracket.indexOf(top);
    }
    private boolean isRightBracket(char c){
        return rightBracket.contains(c);
    }
    public boolean bBalancedEquation() {
        var stack = new Stack<Character>();
        for (char c : str.toCharArray()) {
            if (isLeftBracket(c)) {
                stack.push(c);
            }
            if (isRightBracket(c)) {
                if (stack.empty()) return false;
                char top = stack.pop();
                if (matchBracket(c,top)) {
                    return false;
                }
            }
        }
        return true;
    }
}
```


4. Tree

树是数据结构中很常见的一种，和大自然里的一样，有不同的类别。比如二叉树，二叉搜索树，红黑树等等。在树这一章节中，最需要学习到的思想就是递归。使用递归解决问题最重要有两个点，一个是如何让递归终止，第二个是思考递归如何进行。

4.1 Binary Search Tree(BST)



二叉搜索树是一种非常经典的数据结构，它既有链表的快速插入与删除操作的特点，又有数组快速查找的优势，应用十分广泛，比如文件系统和数据库系统一般就会采用这种数据结构进行高效率的排序和检索操作。

二叉搜索树的特点是，如果它的左子树不为空，则左子树上所有的节点均小于它根节点的值；如果它的右子树不为空，右子树上所有的节点均大于它根节点的值。它的左右子树也分别是二叉搜索树

4.2 Build a tree

由于 delete 方法比较有难度，我们仅实现 find 方法和 insert 方法。树和链表非常类似，也是由一个一个节点组成的，所以我们自然也需要先实现 Node 这个 InnerClass。没有任何方法实现的框架实现如下：

```
public Class MikeTree{  
    private Class Node{  
        private int value;  
        private Node LeftChild;  
        private Node RightChild;  
        public Node(int value){  
            this.value=value;  
        }  
    }  
}
```

4.3 Implement the method:Insert

在上文中我们提过在树中递归是非常重要的一个方法，但是在这个方法中，我们不使用递归来完成，而使用遍历。插入节点的逻辑很简单，根据数据的大小进行选择下一个节点遍历的内容是左节点还是右节点，当下一次遍历的时候如果目标为 null，那么就是我们z需要插入节点的位置了。当然这样插入形成的是二叉搜索树的结构。


```
public void insert(int value) {  
    //generate node  
    var node = new Node(value);  
    if (root == null) {  
        root = node;  
        return;  
    }  
    var current = root;  
    while (true) {  
        if (value > current.value) {  
            if (current.rightChild == null) {  
                current.rightChild = node;  
                break;  
            }  
            current = current.rightChild;  
        } else {  
            if (current.leftChild == null) {  
                current.leftChild = node;  
                break;  
            }  
            current = current.leftChild;  
        }  
    }  
    current = node;  
}
```

4.4 Implement the method: find

find 方法和 insert 方法原理是类似的，由于我们在 insert 的时候进行了大小排序，形成的是一棵二叉搜索树，所以 find 也可以根据这个原理进行寻找。

```

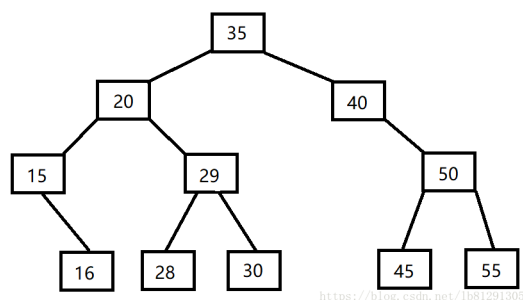
public boolean find(int value) {
    var current = root;
    while (current != null) {
        if (value < current.value) {
            current = current.leftChild;
        } else if (value > current.value) {
            current = current.rightChild;
        } else {
            return true;
        }
    }
    return false;
}

```

4.5 Traverse the tree

从大类上来区分，一共有两种，一种是深度优先，另外一种广度优先。深度优先的遍历顺序多种多样，有前序遍历，中序遍历以及后序遍历，命名方式是根据 root 节点的位置而决定的

- PreOrder Traverse 前序遍历: root,left,right
- InOrder Traverse 中序遍历: left,root,right
- PostOrder Traverse 后序遍历:left,right,root



- 广度优先:35,20,40,15,29,50,16,28,30,45,55
- 深度优先
 - 前序遍历:35,20,15,16,29,28,30,40,50,45,55
 - 中序遍历:15 16 20 28 29 30 35 40 45 50 55
 - 后序遍历:16 15 28 30 29 20 45 55 50 40 35

我们需要根据情景不同而选择合适的遍历方式进行遍历，遍历的逻辑就是递归，只需要找到对应的终止点，和递归逻辑即可执行。

4.5.1 PreOrder Traverse

```
public void traversePreOrder() {
    traversePreOrder(root);
}

private void traversePreOrder(Node root) {
    if (root == null) return;
    System.out.println(root.value);
    traversePreOrder(root.leftChild);
    traversePreOrder(root.rightChild);
}
```

4.5.2 InOrder Traverse

```
public void traverseInOrder() {
    traverseInOrder(root);
}

private void traverseInOrder(Node root) {
    if (root == null) return;

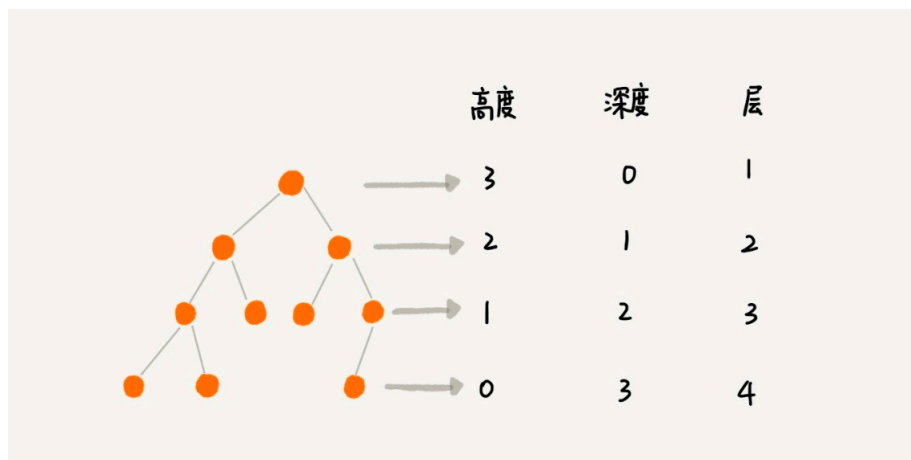
    traverseInOrder(root.leftChild);
    System.out.println(root.value);
    traverseInOrder(root.rightChild);
}
```

4.5.3 PostOrder Traverse

```
public void traversePostOrder() {
    traversePostOrder(root);
}

private void traversePostOrder(Node root) {
    if (root == null) return;
    traversePostOrder(root.leftChild);
    traversePostOrder(root.rightChild);
    System.out.println(root.value);
}
```

4.6 Depth and height of node



高度和深度可以说是同一个类型的概念，但是他们的遍历顺序是相反的。深度是从上往下数的，而高度恰恰相反是从下往上数的。

- 深度的定义：某节点的深度是指从根节点到该节点的最长简单路径边的条数。
- 高度的定义：高度是指从该节点到叶子节点的最长简单路径边的条数。

对于一整棵树来说，深度和高度都是相等的，但是对于具体的节点，深度和高度却不一定是相等的。我们以高度为例，使用递归来编写代码。递归停止的条件就是遍历的节点如果不存在子节点，那么就返回高度 0。但也要注意对于一棵不存在的树，我们需要返回-1。接下来就需要思考递归的条件，对于每一个节点的高度来说 $1 + \text{Math.max}(\text{左节点的高度}, \text{右节点的高度})$ ，取这两边的最大值来使用。

```
public int height() {
    return height(root);
}
private int height(Node root) {
    if (root == null) return -1;
    if (root.leftChild == null &&
        root.rightChild == null)
        return 0;
    return 1 + Math.max(
        height(root.leftChild),
        height(root.rightChild));
}
```

4.7 Min Value in a tree

这是一个求一棵树中最小值的问题，但是我们需要声明一个前提，这不是一棵二叉搜索树，而仅仅是一棵二叉树（虽然它确实是一棵二叉搜索树）。计算这棵树的最小值，我们还是使用递归的方法来思考。如果左右子树都不存在，返回 root 的值即可，如果左子树存在，右子树不存在，就返回左子树的值。如果右子树存在，左子树不存在，就返回右子树的值，这就是递归停止的条件。而递归执行的逻辑自然是返回左右子树,root 中的最小值。

```
public int min() {  
    return min(root);  
}  
private int min(Node root) {  
  
    if (root.leftChild == null && root.rightChild == null)  
        return root.value;  
    if (root.leftChild != null && root.rightChild == null)  
        return Math.min(root.leftChild.value, root.value);  
    if (root.rightChild != null && root.leftChild == null)  
        return Math.min(root.rightChild.value, root.value);  
  
    var left = min(root.leftChild);  
    var right = min(root.rightChild);  
    return Math.min(Math.min(left, right), root.value);  
}
```

4.8 Exercise:Equality Checking

这是一个判断两树相等的问题。解决这个问题还是使用递归即可。递归的循环逻辑非常简单，只需要判断每一个节点都是相等的，那么这棵树就是相等的。而递归的终止条件自然是空节点。

```
public boolean equals(MikeTree other) {
    if (other == null) return false;
    return equals(root, other.root);
}

private boolean equals(Node first, Node second) {
    if (first == null && second == null) {
        return true;
    }
    if (first != null && second != null) {
        return first.value == second.value
            && equals(first.leftChild, second.leftChild)
            && equals(first.rightChild, second.rightChild);
    }
    return false;
}
```

4.9 Exercise:isBinarySearchTree

判断一棵树是否是二叉搜索树，虽然在我们构建的树已经是二叉搜索树了（当成一般的二叉树）。判断的要点自然是根据二叉搜索树的判断要点，根节点的值大于左节点，并且，根节点的值小于右节点，而且每一棵子树也要符合条件。所以肯定使用递归方法来进行判断。我们对每一个节点标注一个值范围，树根节点的范围是 $(-\infty, +\infty)$ 。递归结束的终止条件也是遍历得到的节点为空。

```
public boolean isBinarySearchTree() {
    return isBinarySearchTree(root, Integer.MIN_VALUE,
        Integer.MAX_VALUE);
}

private boolean isBinarySearchTree(Node root, int min, int max) {
    if (root == null) return true;
    return root.value > min && root.value < max
        && isBinarySearchTree(root.leftChild, min, root.value)
        && isBinarySearchTree(root.rightChild, root.value, max);
}
```

4.10 Exercise: Node at K Distance

这个练习时寻找距离 root 节点 K 个单位的节点。递归持续的条件是：每一次转到下一个节点，distance 都-1。递归结束的条件是：当 distance 为 0 的时候就得到了目标节点。

```
public ArrayList<Integer> printNodesAtDistance(int distance) {
    var list=new ArrayList<Integer>();
    if(root==null){
        return null;
    }
    printNodesAtDistance(root,distance,list);

    return list;
}

private void printNodesAtDistance(Node root,int distance,
ArrayList<Integer>arrayList){
    if(root==null) return;
    if(distance==0) {

        arrayList.add(root.value);
        return;
    }
    printNodesAtDistance(root.leftChild,distance-1,arrayList);
    printNodesAtDistance(root.rightChild,distance-1,arrayList);

}
```

4.11 Exercise:Level Traverse

在有之前的结论之后，我们进行广度优先遍历是比较简单的，首先利用高度函数获取高度，然后遍历打印出每一层高度的节点即可


```
public void traverseLevelOrder(){  
  
    for(var i=0;i<=height();i++){  
  
        for(var value: printNodesAtDistance(i)){  
            System.out.println(" 第"+i+" 层 "+value);  
        }  
    }  
}
```



Part One Title

5	Sectioning Examples	33
5.1	Section Title	33
6	In-text Element Examples	37
6.1	Referencing Publications	37
6.2	Link Examples	37
6.3	Lists	37
6.4	International Support	38
6.5	Ligatures	38



5. Sectioning Examples

5.1 Section Title

Lorem ipsum dolor sit amet, consectetur adipiscing elit¹. Praesent porttitor arcu luctus, imperdiet urna iaculis, mattis eros. Pellentesque iaculis odio vel nisl ullamcorper, nec faucibus ipsum molestie. Sed dictum nisl non aliquet porttitor. Etiam vulputate arcu dignissim, finibus sem et, viverra nisl. Aenean luctus congue massa, ut laoreet metus ornare in. Nunc fermentum nisi imperdiet lectus tincidunt vestibulum at ac elit. Nulla mattis nisl eu malesuada suscipit.

Aliquam arcu turpis, ultrices sed luctus ac, vehicula id metus. Morbi eu feugiat velit, et tempus augue. Proin ac mattis tortor. Donec tincidunt, ante rhoncus luctus semper, arcu lorem lobortis justo, nec convallis ante quam quis lectus. Aenean tincidunt sodales massa, et hendrerit tellus mattis ac. Sed non pretium nibh. Donec cursus maximus luctus. Vivamus lobortis eros et massa porta porttitor.

5.1.1 Subsection Title

Fusce varius orci ac magna dapibus porttitor. In tempor leo a neque bibendum sollicitudin. Nulla pretium fermentum nisi, eget sodales magna facilisis eu. Praesent aliquet nulla ut bibendum lacinia. Donec vel mauris vulputate, commodo ligula ut, egestas orci. Suspendisse commodo odio sed hendrerit lobortis. Donec finibus eros erat, vel ornare enim mattis et. Donec finibus dolor quis dolor tempus consequat. Mauris fringilla dui id libero egestas, ut mattis neque ornare. Ut condimentum urna pharetra ipsum consequat, eu interdum elit cursus. Vivamus scelerisque tortor et nunc ultricies, id tincidunt libero pharetra. Aliquam eu imperdiet leo. Morbi a massa

¹Footnote example text...Lorem ipsum dolor sit amet, consectetur adipiscing elit. Praesent porttitor arcu luctus, imperdiet urna iaculis, mattis eros. Pellentesque iaculis odio vel nisl ullamcorper, nec faucibus ipsum molestie.

volutpat velit condimentum convallis et facilisis dolor.

In hac habitasse platea dictumst. Curabitur mattis elit sit amet justo luctus vestibulum. In hac habitasse platea dictumst. Pellentesque lobortis justo enim, a condimentum massa tempor eu. Ut quis nulla a quam pretium eleifend nec eu nisl. Nam cursus porttitor eros, sed luctus ligula convallis quis. Nam convallis, ligula in auctor euismod, ligula mauris fringilla tellus, et egestas mauris odio eget diam. Praesent sodales in ipsum eu dictum. Mauris interdum porttitor fringilla. Proin tincidunt sodales leo at ornare. Donec tempus magna non mauris gravida luctus. Cras vitae arcu vitae mauris eleifend scelerisque. Nam sem sapien, vulputate nec felis eu, blandit convallis risus. Pellentesque sollicitudin venenatis tincidunt. In et ipsum libero.

5.1.1.1 Subsubsection Title

Maecenas consectetur metus at tellus finibus condimentum. Proin arcu lectus, ultrices non tincidunt et, tincidunt ut quam. Integer luctus posuere est, non maximus ante dignissim quis. Nunc a cursus erat. Curabitur suscipit nibh in tincidunt sagittis. Nam malesuada vestibulum quam id gravida. Proin ut dapibus velit. Vestibulum eget quam quis ipsum semper convallis. Duis consectetur nibh ac diam dignissim, id condimentum enim dictum. Nam aliquet ligula eu magna pellentesque, nec sagittis leo lobortis. Aenean tincidunt dignissim egestas. Morbi efficitur risus ante, id tincidunt odio pulvinar vitae.


Paragraph Title Nullam mollis tellus lorem, sed congue ipsum euismod a. Donec pulvinar neque sed ligula ornare sodales. Nulla sagittis vel lectus nec laoreet. Nulla volutpat malesuada turpis at ultricies. Ut luctus velit odio, sagittis volutpat erat aliquet vel. Donec ac neque eget neque volutpat mollis. Vestibulum viverra ligula et sapien bibendum, vel vulputate ex euismod. Curabitur nec velit velit. Aliquam vulputate lorem elit, id tempus nisl finibus sit amet. Curabitur ex turpis, consequat at lectus id, imperdiet molestie augue. Curabitur eu eros molestie purus commodo hendrerit. Quisque auctor ipsum nec mauris malesuada, non fringilla nibh viverra. Quisque gravida, metus quis semper pulvinar, dolor nisl suscipit leo, vestibulum volutpat ante justo ultrices diam. Sed id facilisis turpis, et aliquet eros.

In malesuada ullamcorper urna, sed dapibus diam sollicitudin non. Donec elit odio, accumsan ac nisl a, tempor imperdiet eros. Donec porta tortor eu risus consequat, a pharetra tortor tristique. Morbi sit amet laoreet erat. Morbi et luctus diam, quis porta ipsum. Quisque libero dolor, suscipit id facilisis eget, sodales volutpat dolor. Nullam vulputate interdum aliquam. Mauris id convallis erat, ut vehicula neque. Sed auctor nibh et elit fringilla, nec ultricies dui sollicitudin. Vestibulum vestibulum luctus metus venenatis facilisis. Suspendisse iaculis augue at vehicula ornare. Sed vel eros ut velit fermentum porttitor sed sed massa. Fusce venenatis, metus a rutrum sagittis, enim ex maximus velit, id semper nisi velit eu purus.

Unnumbered Section

Unnumbered Subsection

Unnumbered Subsubsection



6. In-text Element Examples

6.1 Referencing Publications

This statement requires citation [Smith:2022jd]; this one is more specific [Smith:2021qr].

6.2 Link Examples

This is a URL link: [LaTeX Templates](#). This is an email link: example@example.com.
This is a monospaced URL link: `https://www.LaTeXTemplates.com`.

6.3 Lists

Lists are useful to present information in a concise and/or ordered way.

6.3.1 Numbered List

1. First numbered item
 - a. First indented numbered item
 - b. Second indented numbered item
 - i. First second-level indented numbered item
2. Second numbered item
3. Third numbered item

6.3.2 Bullet Point List

- First bullet point item
 - First indented bullet point item
 - Second indented bullet point item
 - First second-level indented bullet point item
- Second bullet point item

- Third bullet point item

6.3.3 Descriptions and Definitions

Name Description

Word Definition

Comment Elaboration

6.4 International Support

àáâãäåèéêëìíîïòóôõöøùúûüýÿñçšž
 ÀÁÂÃÄÅÈÉÊËÌÍÎÏÒÓÔÕÖØÙÚÛÜÝŸÑ
 ßÇÆĖČŠŽ

6.5 Ligatures

fi fj fl ffl ffi Ty



Part Two Title

7	Mathematics	41
7.1	Theorems	41
7.2	Definitions	41
7.3	Notations	42
7.4	Remarks	42
7.5	Corollaries	42
7.6	Propositions	42
7.7	Examples	42
7.8	Exercises	43
7.9	Problems	43
7.10	Vocabulary	43
8	Presenting Information and Results with a Long Chapter Title	45
8.1	Table	45
8.2	Figure	45



7. Mathematics

7.1 Theorems

7.1.1 Several equations

This is a theorem consisting of several equations.

Theorem 7.1 — **Name of the theorem.** In $E = \mathbb{R}^n$ all norms are equivalent. It has the properties:

$$||\mathbf{x}|| - ||\mathbf{y}|| \leq ||\mathbf{x} - \mathbf{y}|| \quad (7.1)$$

$$||\sum_{i=1}^n \mathbf{x}_i|| \leq \sum_{i=1}^n ||\mathbf{x}_i|| \quad \text{where } n \text{ is a finite integer} \quad (7.2)$$

7.1.2 Single Line

This is a theorem consisting of just one line.

Theorem 7.2 A set $\mathcal{D}(G)$ is dense in $L^2(G)$, $|\cdot|_0$.

7.2 Definitions

A definition can be mathematical or it could define a concept.

Definition 7.1 — **Definition name.** Given a vector space E , a norm on E is an application, denoted $||\cdot||$, E in $\mathbb{R}^+ = [0, +\infty[$ such that:

$$||\mathbf{x}|| = 0 \Rightarrow \mathbf{x} = \mathbf{0} \quad (7.3)$$

$$||\lambda \mathbf{x}|| = |\lambda| \cdot ||\mathbf{x}|| \quad (7.4)$$

$$||\mathbf{x} + \mathbf{y}|| \leq ||\mathbf{x}|| + ||\mathbf{y}|| \quad (7.5)$$

7.3 Notations

■ **Notation 7.1** Given an open subset G of \mathbb{R}^n , the set of functions φ are:

1. Bounded support G ;
2. Infinitely differentiable;

a vector space is denoted by $\mathcal{D}(G)$.

7.4 Remarks

This is an example of a remark.



The concepts presented here are now in conventional employment in mathematics. Vector spaces are taken over the field $\mathbb{K} = \mathbb{R}$, however, established properties are easily extended to $\mathbb{K} = \mathbb{C}$.

7.5 Corollaries

Corollary 7.1 — Corollary name. The concepts presented here are now in conventional employment in mathematics. Vector spaces are taken over the field $\mathbb{K} = \mathbb{R}$, however, established properties are easily extended to $\mathbb{K} = \mathbb{C}$.

7.6 Propositions

7.6.1 Several equations

Proposition 7.1 — Proposition name. It has the properties:

$$||\mathbf{x}| - |\mathbf{y}|| \leq \|\mathbf{x} - \mathbf{y}\| \quad (7.6)$$

$$||\sum_{i=1}^n \mathbf{x}_i|| \leq \sum_{i=1}^n ||\mathbf{x}_i|| \quad \text{where } n \text{ is a finite integer} \quad (7.7)$$

7.6.2 Single Line

Proposition 7.2 Let $f, g \in L^2(G)$; if $\forall \varphi \in \mathcal{D}(G)$, $(f, \varphi)_0 = (g, \varphi)_0$ then $f = g$.

7.7 Examples

7.7.1 Equation Example

■ **Example 7.1** Let $G = \{x \in \mathbb{R}^2 : |x| < 3\}$ and denoted by: $x^0 = (1, 1)$; consider the function:

$$f(x) = \begin{cases} e^{|x|} & \text{si } |x - x^0| \leq 1/2 \\ 0 & \text{si } |x - x^0| > 1/2 \end{cases} \quad (7.8)$$

The function f has bounded support, we can take $A = \{x \in \mathbb{R}^2 : |x - x^0| \leq 1/2 + \varepsilon\}$ for all $\varepsilon \in]0; 5/2 - \sqrt{2}[$. ■

7.7.2 Text Example

■ **Example 7.2 — Example name.** Aliquam arcu turpis, ultrices sed luctus ac, vehicula id metus. Morbi eu feugiat velit, et tempus augue. Proin ac mattis tortor. Donec tincidunt, ante rhoncus luctus semper, arcu lorem lobortis justo, nec convallis ante quam quis lectus. Aenean tincidunt sodales massa, et hendrerit tellus mattis ac. Sed non pretium nibh. Donec cursus maximus luctus. Vivamus lobortis eros et massa porta porttitor. ■

7.8 Exercises

Exercise 7.1 This is a good place to ask a question to test learning progress or further cement ideas into students' minds. ■

7.9 Problems

Problem 7.1 What is the average airspeed velocity of an unladen swallow?

7.10 Vocabulary

Define a word to improve a students' vocabulary.

■ **Vocabulary 7.1 — Word.** Definition of word.



8. Presenting Information and Results with a Long Chapter Title

8.1 Table

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Praesent porttitor arcu luctus, imperdiet urna iaculis, mattis eros. Pellentesque iaculis odio vel nisl ullamcorper, nec faucibus ipsum molestie. Sed dictum nisl non aliquet porttitor. Etiam vulputate arcu dignissim, finibus sem et, viverra nisl. Aenean luctus congue massa, ut laoreet metus ornare in. Nunc fermentum nisi imperdiet lectus tincidunt vestibulum at ac elit. Nulla mattis nisl eu malesuada suscipit.

Treatments	Response 1	Response 2
Treatment 1	0.0003262	0.562
Treatment 2	0.0015681	0.910
Treatment 3	0.0009271	0.296

表 8.1: Table caption.

Referencing Table 8.1 in-text using its label.

8.2 Figure

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Praesent porttitor arcu luctus, imperdiet urna iaculis, mattis eros. Pellentesque iaculis odio vel nisl ullamcorper, nec faucibus ipsum molestie. Sed dictum nisl non aliquet porttitor. Etiam vulputate arcu dignissim, finibus sem et, viverra nisl. Aenean luctus congue massa, ut laoreet metus ornare in. Nunc fermentum nisi imperdiet lectus tincidunt vestibulum at ac elit. Nulla mattis nisl eu malesuada suscipit.

Treatments	Response 1	Response 2
Treatment 1	0.0003262	0.562
Treatment 2	0.0015681	0.910
Treatment 3	0.0009271	0.296

表 8.2: Floating table.



图 8.1: Figure caption.

Referencing Figure 8.1 in-text using its label.



图 8.2: Floating figure.

Bibliography

Articles

Books



A. Appendix Chapter Title

A.1 Appendix Section Title

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Aliquam auctor mi risus, quis tempor libero hendrerit at. Duis hendrerit placerat quam et semper. Nam ultricies metus vehicula arcu viverra, vel ullamcorper justo elementum. Pellentesque vel mi ac lectus cursus posuere et nec ex. Fusce quis mauris egestas lacus commodo venenatis. Ut at arcu lectus. Donec et urna nunc. Morbi eu nisl cursus sapien eleifend tincidunt quis quis est. Donec ut orci ex. Praesent ligula enim, ullamcorper non lorem a, ultrices volutpat dolor. Nullam at imperdiet urna. Pellentesque nec velit eget est euismod pretium.



B. Appendix Chapter Title

B.1 Appendix Section Title

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Aliquam auctor mi risus, quis tempor libero hendrerit at. Duis hendrerit placerat quam et semper. Nam ultricies metus vehicula arcu viverra, vel ullamcorper justo elementum. Pellentesque vel mi ac lectus cursus posuere et nec ex. Fusce quis mauris egestas lacus commodo venenatis. Ut at arcu lectus. Donec et urna nunc. Morbi eu nisl cursus sapien eleifend tincidunt quis quis est. Donec ut orci ex. Praesent ligula enim, ullamcorper non lorem a, ultrices volutpat dolor. Nullam at imperdiet urna. Pellentesque nec velit eget est euismod pretium.