1. 线性结构：数组，队列，链表，栈 非线性结构：二维数组，多维数组，广义表，树结构，图结构
2. 稀疏数组：当一个数组大部分元素都为零，或者为同一个值的时候，可以使用稀疏数组来保存改数组
3. 代码实现

|  |
| --- |
| **ublic** **static** **void** main(String[] args) {  // 创建一个二维数组11\*11  // 0.表示没有棋子，1.表示白棋子，2.表示蓝棋子  **int** chessArr1[][] = **new** **int**[11][11];  chessArr1[1][2] = 1;  chessArr1[2][3] = 2;  chessArr1[4][5] = 2;  //输出原始的二维数组  System.***out***.println("原始的二维数组~~");  **for** (**int**[] row : chessArr1) {  **for** (**int** data : row) {  System.***out***.printf("%d\t", data);  }  System.***out***.println();  }  //将二维数组转换称为稀疏数组  // 1. 先遍历二维数组得到非零的数值  **int** sum = 0;  **for** (**int** i = 0; i < 11; i++) {  **for** (**int** j = 0; j < 11; j++) {  **if** (chessArr1[i][j] != 0) {  sum++;  }  }  }  // 2. 创建对应的稀疏数组  **int** sparseArr[][] = **new** **int**[sum + 1][3];  // 给稀疏数组赋值ֵ  sparseArr[0][0] = 11;  sparseArr[0][1] = 11;  sparseArr[0][2] = sum;    // 遍历二维数组，经非零的值存放到sparseArr中  **int** count = 0; //count 用于记录是第几个非零  **for** (**int** i = 0; i < 11; i++) {  **for** (**int** j = 0; j < 11; j++) {  **if** (chessArr1[i][j] != 0) {  count++;  sparseArr[count][0] = i;  sparseArr[count][1] = j;  sparseArr[count][2] = chessArr1[i][j];  }  }  }    // 输出稀疏数组的形式  System.***out***.println();  System.***out***.println("输出的稀疏数组~~~~");  **for** (**int** i = 0; i < sparseArr.length; i++) {  System.***out***.printf("%d\t%d\t%d\t\n", sparseArr[i][0], sparseArr[i][1], sparseArr[i][2]);  }  System.***out***.println();    //将稀疏数组转换成原始的二维数组  /\*  \* 1.先读取稀疏数组的第一行，根据第一行的数据，创建原始的二维数组，比如上面的chessArr2=new int[][]  2. 在读取稀疏数组后的几行数据，并赋值给原始的二维数组即可  \*/    //1. 先读取稀疏数组的第一行，根据第一行的数据，创建原始的二维数组  **int** chessArr2[][]=**new** **int**[sparseArr[0][0]][sparseArr[0][1]];  // 2.在读取稀疏数组后几行的数据（从第二行开始），并赋值给原始的二维数组  **for** (**int** i=1;i<sparseArr.length;i++){  chessArr2[sparseArr[i][0]][sparseArr[i][1]]=sparseArr[i][2];  }  // 输出恢复后的二维数组  System.***out***.println();  System.***out***.println("恢复后的二维数组");  **for** (**int**[] data:chessArr2){  **for** (**int** row:data){  System.***out***.printf("%d\t",row);  }  System.***out***.println();  }    }   1. } |

1. 队列：队列是一个有序列表，可以用数组或者链表来实现 遵循先进先出的原则：先存入队列的数据，要先取出。后存入的要后取出

#### 队列的代码实现

public class ArrayQueueDemo {

public static void main(String[] args) {

//测试一把

//创建一个队列

ArrayQueue queue = new ArrayQueue(3);

char key = ' '; //接收用户输入

Scanner scanner = new Scanner(System.in);//

boolean loop = true;

//输出一个菜单

while(loop) {

System.out.println("s(show): 显示队列");

System.out.println("e(exit): 退出程序");

System.out.println("a(add): 添加数据到队列");

System.out.println("g(get): 从队列取出数据");

System.out.println("h(head): 查看队列头的数据");

key = scanner.next().charAt(0);//接收一个字符

switch (key) {

case 's':

queue.showQueue();

break;

case 'a':

System.out.println("输出一个数");

int value = scanner.nextInt();

queue.addQueue(value);

break;

case 'g': //取出数据

try {

int res = queue.getQueue();

System.out.printf("取出的数据是%d\n", res);

} catch (Exception e) {

// TODO: handle exception

System.out.println(e.getMessage());

}

break;

case 'h': //查看队列头的数据

try {

int res = queue.headQueue();

System.out.printf("队列头的数据是%d\n", res);

} catch (Exception e) {

// TODO: handle exception

System.out.println(e.getMessage());

}

break;

case 'e': //退出

scanner.close();

loop = false;

break;

default:

break;

}

}

System.out.println("程序退出~~");

}

}

// 使用数组模拟队列-编写一个ArrayQueue类

class ArrayQueue {

private int maxSize; // 表示数组的最大容量

private int front; // 队列头

private int rear; // 队列尾

private int[] arr; // 该数据用于存放数据, 模拟队列

// 创建队列的构造器

public ArrayQueue(int arrMaxSize) {

maxSize = arrMaxSize;

arr = new int[maxSize];

front = -1; // 指向队列头部，分析出front是指向队列头的前一个位置.

rear = -1; // 指向队列尾，指向队列尾的数据(即就是队列最后一个数据)

}

// 判断队列是否满

public boolean isFull() {

return rear == maxSize - 1;

}

// 判断队列是否为空

public boolean isEmpty() {

return rear == front;

}

// 添加数据到队列

public void addQueue(int n) {

// 判断队列是否满

if (isFull()) {

System.out.println("队列满，不能加入数据~");

return;

}

rear++; // 让rear 后移

arr[rear] = n;

}

// 获取队列的数据, 出队列

public int getQueue() {

// 判断队列是否空

if (isEmpty()) {

// 通过抛出异常

throw new RuntimeException("队列空，不能取数据");

}

front++; // front后移

return arr[front];

}

// 显示队列的所有数据

public void showQueue() {

// 遍历

if (isEmpty()) {

System.out.println("队列空的，没有数据~~");

return;

}

for (int i = 0; i < arr.length; i++) {

System.out.printf("arr[%d]=%d\n", i, arr[i]);

}

}

// 显示队列的头数据， 注意不是取出数据

public int headQueue() {

// 判断

if (isEmpty()) {

throw new RuntimeException("队列空的，没有数据~~");

}

return arr[front + 1];

}

}

1. 实现环形队列的思想：1.front变量的含义：front就指向队列的第一个元素，也就是arr【front】就是队列的第一个元素 front=0 2.rear变量的含义：rear指向队列的最后一个元素的后一个位置，因为希望空出一个空间作为约定，rear=0；3.当队列满时：条件呢是（rear+1）%maxSize=front 4.当队列为空时：rear=front 5.队列中有效的个数 （rear+maxSize-front）%amxSize

#### 环形列表的代码实现

package com.ly.queueArrayDemo;

import java.util.Scanner;

public class CircleArrayQueue {

public static void main(String[] args) {

System.out.println("测试数组模拟唤醒列表");

CicleArray CicleArray=new CicleArray(4);//此时环形队列的有效数据为3

char key=' ';

Scanner scanner=new Scanner(System.in);

boolean loop=true;

while (loop) {

System.out.println("s(show),显示队列");

System.out.println("e(exit),退出队列");

System.out.println("g(get),从队列中去除数据");

System.out.println("a(add),向队列中添加数据");

System.out.println("h(head),显示队列的头部数据");

key=scanner.next().charAt(0);

switch (key){

case's':

CicleArray.showQueue();

break;

case 'a':

System.out.println("输入一个数");

int value=scanner.nextInt();

CicleArray.addQueue(value);

break;

case 'g':

try {

int res=CicleArray.getQueue();

System.out.printf("取出的数据是%d\n",res);

}catch (Exception e){

System.out.println(e.getMessage());

}

break;

case 'h':

try {

System.out.printf("队列的头数据是%d\n",CicleArray.headQueue());

}catch (Exception e){

System.out.println(e.getMessage());

}

break;

case 'e':

scanner.close();

loop=false;

break;

default:

break;

}

}

System.out.println("程序退出");

}

}

class CicleArray{

private int maxSize;

private int front;

private int rear;

private int[] arr;

public CicleArray(int arrMaxSize){

maxSize=arrMaxSize;

arr=new int[arrMaxSize];

}

public boolean isFull(){

return (rear+1)%maxSize==front;

}

public boolean isEmpty(){

return rear==front;

}

public void addQueue(int n){

if (isFull()){

System.out.println("队列已满，没法添加数据");

return;

}

// rear指向最后一个元素的后一个位置，所以直接添加即可

arr[rear]=n;

// 将rear后移，这里必须考虑取模

rear=(rear+1)%maxSize;

}

public int getQueue(){

if (isEmpty()){

throw new RuntimeException("队列为空，没有数据");

}

// 这里需要知道front是直接指向队列第一个元素的

// 首先把front对应的值保存在一个临时变量中

// 将front后移

// 然后返回临时保存的变量值

int value=arr[front];

front=(front+1)%maxSize;

return value;

}

public int headQueue(){

if (isEmpty()){

throw new RuntimeException("队列为空，没有数据");

}

return arr[front];

}

public void showQueue(){

if (isEmpty()){

System.out.println("队列为空，没有数据");

}

// 思路：从front开始遍历，遍历多少个是数据就好

for (int i=front;i<front+size();i++){

System.out.printf("arr[%d]=%d",i%maxSize,arr[i%maxSize]);

}

}

// 求出当前队列的有效的数据个数

public int size(){

return (rear+maxSize-front)%maxSize;

}

}

### 链表

##### 代码实现

package com.ly.linklist;

import com.sun.org.apache.xerces.internal.dom.PSVIAttrNSImpl;

import javax.swing.text.AbstractDocument;

import javax.swing.text.html.HTMLEditorKit;

public class SingleLinkListDemo {

public static void main(String[] args) {

// 先创建几个节点

HeroNode hero1=new HeroNode(1,"宋江","及时雨");

HeroNode hero2=new HeroNode(2,"卢俊义","玉麒麟");

HeroNode hero3=new HeroNode(3,"吴用","智多星");

HeroNode hero4=new HeroNode(4,"林冲","豹子头");

SingleLinkedList singleLinkedList=new SingleLinkedList();

// singleLinkedList.add(hero1);

// singleLinkedList.add(hero2);

// singleLinkedList.add(hero3);

// singleLinkedList.add(hero4);

// 按照编号添加

singleLinkedList.addByOrder(hero4);

singleLinkedList.addByOrder(hero1);

singleLinkedList.addByOrder(hero2);

singleLinkedList.addByOrder(hero3);

singleLinkedList.list();

// 修改节点的代码

HeroNode newHeroNode=new HeroNode(2,"lyyy","李大帅");

singleLinkedList.update(newHeroNode);

System.out.println("修改后的链表情况");

singleLinkedList.list();

System.out.println("删除后的节点链表");

singleLinkedList.del(1);

singleLinkedList.del(2);

singleLinkedList.del(3);

singleLinkedList.del(4);

singleLinkedList.list();

}

}

class SingleLinkedList {

private HeroNode head = new HeroNode(0, "", "");

public void add(HeroNode heroNode) {

HeroNode temp=head;

while (true) {

if (temp.next == null) {

break;

}

temp=temp.next;

}

temp.next=heroNode;

}

// 第二种添加英雄的方式，按照排名添加

public void addByOrder(HeroNode heroNode){

// 因为是单链表，我们找的temp是位于添加位置的前一个节点

HeroNode temp=head;

boolean flag=false;

while (true){

if (temp.next==null){//说明temp已经在链表的最后

break;

}

if (temp.next.no>heroNode.no){ //位置找到，就在temp的后面插入

break;

}else if (temp.next.no==heroNode.no){//说明希望添加的heroNode的编号已然存在

flag=true;

break;

}

temp=temp.next; //后移，遍历当前链表

}

// 判断flag的值

if(flag){//说明变好已经存在

System.out.printf("待插入的值%d已经存在\n",heroNode.no);

}else {

// 插入到链表中，temp的后面

heroNode.next=temp.next;

temp.next=heroNode;

}

}

// 修改节点的name和nickname属性，根据编号来修改，即no不能修改

// 根据newHeroNode的no来修改

public void update(HeroNode newHeroNode){

HeroNode temp=head.next;

boolean flag=false;

if (head.next==null){

System.out.println("链表为空，没有数据");

return;

}

// 找到需要修改的节点，根据no编号来修改

// 定义一个辅助变量

while(true){

if (temp==null){

break;//表示已经到链表的最后

}

if (temp.no==newHeroNode.no){

// 找到

flag=true;

break;

}

temp=temp.next;

}

// 根据flag判断是否找到要修改的节点

if(flag==true){

temp.name=newHeroNode.name;

temp.nickname= newHeroNode.nickname;

}else {

// 没有找到

System.out.printf("没有找到%d节点",newHeroNode.no);

}

}

// 删除节点

// head节点不能动

// 需要设置一个辅助节点，找到要删除节点的前一个节点

public void del(int no){

HeroNode temp=head;

boolean flag=true;//表示是否找到待删除的节点

while (true){

if (temp.next==null){

break;//表示已经到链表的最后

}

if(temp.next.no==no){

// 表示已经找到要删除的节点，是temp的下一个节点

flag=true;

break;

}

temp=temp.next;//temp后移，遍历数组

}

if (flag){

temp.next=temp.next.next;

}else {

// 表示没有找到要删除的节点

System.out.printf("%d节点不存在\n",no);

}

}

//遍历链表

public void list () {

if (head.next == null) {

System.out.println("链表为空");

return;

}

HeroNode temp=head.next;

while (true){

if (temp==null){

break;

}

System.out.println(temp);

temp=temp.next;

}

}

}

class HeroNode{

public int no;

public String name;

public String nickname;

public HeroNode next;

public HeroNode(int no,String name,String nickname){

this.name=name;

this.nickname=nickname;

this.no=no;

}

@Override

public String toString() {

return "HeroNode{" +

"no=" + no +

", name='" + name + '\'' +

", nickname='" + nickname + '\'' +

'}';

}

}

### 栈

##### 前缀表达式

前缀表达式又称波兰表达式，即所有的运算符位于操作符之前

###### 中缀表达式转后缀表达式

1. 初始化两个栈：运算符栈s1和储存中间结果的栈s2；
2. 从左到右扫描中缀表达式；
3. 遇到操作数，将其压入s2；
4. 遇到运算符时，比较其与栈顶运算符的优先级
   * 1. 如果s1为空，或栈顶运算服为“（”，则直接将此运算符入栈
     2. 否则，若优先级比栈顶运算符的高，也将运算符压入s1；
     3. 否则，将s1栈顶的运算符弹出并压入到s2中，再次转到（4.1）与s1中新的栈顶运算符相比较
5. 遇到括号时，
6. 如果是左括号“（”，则直接压入s1
7. 如果是右括号，“）”，则依次弹出s1栈顶的运算符，并压入到s2，知道遇到左括号为止，此时将这一对括号丢弃
8. 重复步骤2到5，直到表达式的最右边
9. 将是中剩余的运算符依次弹出并压入s2
10. 依次弹出s2中的元素并输出，**结果的逆序即为中缀表达式对应的后缀表达式的值**

### 递归调用规则：

1. 当程序执行到一个方法时，就会开辟一个独立的空间（栈）
2. 每个空间的数据（局部变量） 是独立的，比如变量n
3. 如果方法中使用的是引用型变量（比如数组），就会共享该引用类型的变量
4. 递归必须向退出递归的条件逼近，否则就是无限递归，出现Stack Overflow错误
5. 当一个方法执行完毕，或者遇到return，就会返回，遵守谁调用，就会将结果返回给谁，同时当方法执行完毕或者返回时，该方法也就执行完毕。

冒泡排序：

1. 一共进行数组大小-1次排序
2. 每一次排序的次数在逐渐的减小
3. 如果我们发现在某次排序中，没有发生一次交换，可以提前结束冒泡排序（优化）

选择排序；

1. 选择排序也属于内部排序法，是从欲排序的数据中，按指定的规则选出来某元素再依次交换位置后达到排序的目的

插入排序：

插入式排序属于内部排序法，是对于欲排序的元素以插入的方式寻找该元素的适当位置，以达到排序的目的

希尔排序：

希尔排序的基本思想： 希尔排序是把记录按下标的一定增量分组，对每组使用直接插入排序算法排序；随着增量逐渐减少，每组包含的关键词越来越多，当缩减量减至1时，整个文件恰被分成一组，算法便终止

快速排序：

快速排序是对冒泡排序的一种改进，基本思想：通过一趟排序将要排序的数据分割成独立的两部分，其中一部分的数据都比另外一部分的数据都要小，然后再按照此方法对这两部数据分别进行快速排序，整个排序过程可以递归进行，已达到整个数据变成有序序列

归并排序

归并排序：是利用归并的思想实现的排序方法，该算法采用经典的分治算法策略成一些小的问题然后递归求解，而治的阶段则将分的阶段得到的各答案“修补”在一起，即分而治之

基数排序

1. 基数排序属于“分配式排序”，又称“桶子法”，或者binsort，顾名思义，它是通过键值的各个位的值，将要排序的元素分配至某些“桶”中，达到排序的作用
2. 基数排序法是属于稳定性质的排序，基数排序法是效率高的稳定性排序法
3. 基数排序是桶排序的扩展
4. 基数排序的实现：将整数位数切割成不同的数字，然后按每个位数分别比较
5. 基数排序基本思想：将所有待比较数值统一为同样的数位长度，数位短的数前面补零，然后，从最低位开始，依次进行一次排序。这样从最低位排序一直到最高位排序完成以后，数列就变成一个有序序列

# 查找算法

线性查找

二分法查找

数组必须是有序的，然后采用递归的思想，进查找

插值查找

1. 插值查找算法类似于二分查找，不同的是插值查找每次从自适用mid处开始查找
2. 将折半查找中的求mid索引的公式，low表示左边索引，high表示右边的索引
3. Int midIndex=low+（high-low）\*(key-arr[low])/(arr[high]-arr[low]); 对应的公式：  
   int mid=low+(high-low)\*(findVal-arr[low])/(arr[high]-arr[low]);
4. 插值查找的注意事项：
   * + - 1. 对于数据量比价大，关键字分布比较均匀的查找表来说，采用插值查找，速度较快
         2. 关键字分布不均匀，该方法不一定比折半查找要好

斐波那契（黄金分割）查找算法

斐波那契原理：斐波那契查找原理与前面两种相似，仅仅改变了中间节点的位置，mid不再是中间或者插值得到，而是位于黄金分割点附近，即mid=low+F(k-1)-1 F代表斐波那契数列

|  |  |
| --- | --- |
| 哈希项目 | 所需数目 |
| 书籍 | 1 |
| 杂志 | 3 |
| 笔记本 | 1 |
| 便笺簿 | 1 |
| 钢笔 | 3 |
| 铅笔 | 2 |
| 荧光笔 | 2 色 |
| 剪刀 | 1 把 |

1. 散列表；是根据关键码值而直接进行访问的数据结构。也就是说，它通过把关键值映射到表中一个位置来访问记录，以加快查找的速度。这个映射函数叫做散列函数，存放记录的数组叫做散列表

# 树

二叉树

1. 树有很多，每个节点最多只有两个子节点的一种形式成为二叉树
2. 如果该二叉树的所有叶子节点都在最后一层，并且结点总数=2^n-1,n为层数，则我们成为满二叉树
3. 如果该二叉树的所有叶子节点都在最后一层或者倒数第二层，而且最后一层的叶子节点在左边连续，倒数第二层的叶子节点在右边连续，我们成为完全二叉树
4. 前序遍历：先输出父节点，再遍历左子树和右子树
5. 中序遍历：先遍历左子树，再输出父节点，最后输出父节点
6. 后序遍历：先遍历左子树，再遍历右子树，最后输出节点

顺序存储二叉树

说明：从数据存储来看数组存储方式和树的存储方式可以相互转换，即数组可以转换成树，树也可以转换成数组

* 1. 顺序儿二叉树通常只考虑完全二叉树
  2. 第n个元素的左子节点为2\*n+1；
  3. 第n个元素的右子节点为2\*n+1
  4. 第n个元素的父节点为（n-1）/2
  5. N：表示二叉树中的第几个元素（按0开始编号）

线索化二叉树

1. n个节点的二叉链表中含有n+1公式2n-（n-1）=n+1个空指针域。利用二叉链表中的空指针域，存放指向节点在某种遍历次序下的前序和后继结点的指针（这种附加党的指针成为“线索”）
2. 这种加上了线索的二叉链表成为线索链表，相应的二叉树成为线索二叉树。根据线索性质的不同，线索性质的不同，线索二叉树可分为前序线索二叉树，中序线索二叉树和后序线索二叉树
3. 一个节点的前一个节点成为前驱节点
4. 一个节点的后一个节点成为后继节点
5. 说明：当线索化二叉树后，Node节点的属性left和right，有如下情况：
6. ①left指向的是左子树，也可能是指向的前驱节点② right 指向的是右子树，也可能指向后继节点

堆排序

1. 堆排序是利用堆这种数据结构而设计的一种排序算法，堆排序是一种选择排序，它的最坏，最好，平均时间复杂度均为O(nlogn)，它也是不稳定排序
2. 堆排序具有以下性质的完全二叉树，每个节点的值都大于或等于其左右孩子节点的值，称为大顶堆，注意：没有要求节点的左孩子和右孩子的值的大小关系
3. 每个节点的值都小于或者等于其左右孩子节点的值，称为小顶堆
4. 大顶堆的特点：arr[i]>=arr[2\*i+1]&&arr[i]>=arr[2\*i+2]
5. 一般升序采用大顶堆，降序采用小丁堆

赫夫曼编码

基本介绍：

1. 赫夫曼编码也翻译为哈夫曼编码，又称霍夫曼编码，是一种编码方式，属于一种算法
2. 赫夫曼编码是赫夫曼树在电讯通信中的经典的应用之一
3. 赫夫曼编码广泛地应用在数据文件压缩，其压缩率通常在20%~300%
4. 赫夫曼码是可变字长编码的一种，huffman于1952年提出一种编码方法，称之为最佳编码

二叉排序树：

二叉排序树：BST，对于二叉排序树的任何一个非叶子节点，要求左子节点的值比当前节点的值小，右子节点的值比当前节点的值大

特别说明：如果有相同的值，可以将该节点放在左子节点或者右子节点

平衡二叉树

基本介绍：

1)：平衡二叉树也叫做平衡二叉搜索树又被称为AVL树，可以保证查询效率高

2）具有以下的特点：它是一颗空树或他的左右两个子树的高度差的绝对值不超过1，并且左右两个子树都是一颗平衡二叉树。平衡二叉树的常用实现方法有红黑树，AVL，替罪羊树，Treap，伸展树等

左旋转的思路：

1. 创建一个新的节点newNode，值等于当前根节点的值
2. 把新节点的左子树设置成为当前节点的左子树
3. 把新节点的右子树设置成为当前节点的右子树的左子树
4. 把当前节点的值置换为右子节点的值
5. 把当前节点的右子树设置成为右子树的右子树
6. 把当前节点的左子树设置成为新节点

右旋转的思路：

1. 创建一个新的节点newNode（以10这个值创建），创建一个新的节点，值等于当前节点的值
2. 把新节点的右子树设置了当前节点的右子树
3. 把新节点的左子树设置为当前节点的左子树的右子树
4. 把当前节点的值换为左子节点的值
5. 把当前节点的左子树设置成左子树的左子树
6. 把当前节点的右子树设置为新节点

2-3树

1)2-3树的所有叶子节点都在同一层（只要是b树都满足这个条件）

2)有两个子节点的节点叫二节点，二节点要么没有子节点，要么有两个子节点

3）有三个子节点的节点叫三节点，三节点要么没有子节点，要么有是三个子节点

4）2-3树是由二节点和三节点构成的树

B树

1. b树的阶：节点的做多子节点个数。比如2-3树的阶是3,2-3-4树的阶是4
2. B-树的搜索，从根节点开始，对结点内的关键字（有序）序列进行二分查找，如果命中则结束，否则进入查询关键字所属范围内的儿子节点；重复，直到所对应的儿子指针为空，或已经是叶子节点
3. 关键字集合分布在整棵树中，即叶子节点和非叶子节点都存放数据
4. 搜索有可能在非叶子节点结束
5. 其搜索性等价于在关键字全集内做一次二分查找

B+树

1. B+树的搜索与B树也基本相同，区别是B+树只有达到叶子节点才命中（B树可以在非叶子节点命中），其性能也等价于在关键字全集做一次二分查找
2. 所有关键字都出现在叶子节点的链表中（即数据只能在叶子节点（也叫稠密索引）），且链表中的关键字（数据）恰好是有序的
3. 不可能在非叶子节点命中
4. 非叶子节点相当于是叶子节点的索引（稀疏索引），叶子节点相当于是存储（关键字）数据的数据层
5. 更适合文件索引系统
6. B树和B+树各有自己的应用场景，不能说B+树完全比B树好，反之亦然

B\*树

1. B\*树定义了非叶子节点关键字个数至少为（2/3）\*M，即块的最低使用率为2/3，而B+树的块的最低使用率为B+树的1/2
2. 从第一个特点我们可以看出，B\*树分配节点的概率比B+树要低，空间使用率更高

# 图

### 图的深度优先遍历介绍（DFS）

所谓图的遍历，即是对结点的访问。一个图有那么多个节点，如何遍历这些结点，需要特定策略，一般有两种访问策略（1）深度优先遍历（2）广度优先遍历

深度优先遍历基本思想

图的深度优先搜索

1. 深度优先遍历，从初始访问结点出发，初始访问结点可能有多个邻接节点，深度优先遍历的策略就是首先访问第一个邻接结点，然后再以这个被访问的邻接结点作为初始节点，访问它的第一个邻接结点，可以这样理解：每次都在访问完当前节点后首先访问当前节点的第一个邻接结点
2. 我们可以看到，这样的访问策略是优先级往纵向挖掘深入，而不是对一个结点的所有邻接结点进行横向访问
3. 显然，深度优先遍历是一个递归的过程

### 深度优先遍历介绍

1. 访问初始结点v,并标记结点v为已访问
2. 查找结点v的第一个邻接结点w
3. 若w存在，则继续执行4，如果w不存在，则回到第一步，将从v的下一个结点继续
4. 若w未被访问，对w进行深度优先遍历递归（即把w当做另一个v，然后进行步骤123）
5. 查找结点v的w邻接结点的下一个邻接结点，转到步骤3

### 图的广度优先遍历算法步骤

1. 访问初始结点v并标记结点v为已访问
2. 结点v入队列
3. 当队列非空时，继续执行，否则算法结束
4. 出队列，取得队头结点u
5. 查找结点u的第一个邻接结点w
6. 若结点u的邻接结点w不存在，则转到步骤3；否则循环执行一下三个步骤

6.1若结点w尚未被访问，则访问结点w并标记为已访问

6.2结点w入列

6.3查找结点u的继邻接结点后的下一个邻接结点w，并转到步骤6

# 算法

## 分治算法

分治算法的基本步骤

1. 分解：将原问题分解为若干个规模较小，相互独立，与原问题形式相同的子问题
2. 解决：若干个问题规模较小而容易被解决则直接接，否则递归地解决各个子问题
3. 合并：将各个问题的解合并为原问题的解

## 动态规划问题—背包问题

1. 动态规划算法的核心思想是：将大问题划分为小问题进行解决，从而一步步获取最优解的处理算法
2. 动态规划算法与分治算法类似：其基本思想也是将带求解问题分解成若干个子问题，先求解子问题，然后从这些子问题的解得到原问题的解
3. 与分治法不同的是，适合于动态会话求解的问题，经分解得到子问题往往不是互相独立的
4. 冬天规划可以通过填表的方式来逐步推进，得到最优解
5. 所以，我们得到下面的步骤
6. v[i][0]=v[0][j]=0;//第一行和第一列全部为零
7. w[i]>j,v[i][j]=v[i-1][j] 当放入的物品的重量大于背包的容量时，把上一次放入方案移下来
8. w[i]<=j,v[i][j]=max{v[i-1][j],v[i]+v[i-1][j-w[i]]}//当物品的容量小于背包的容量，比较两次的最大值，取其最大值

## KMP算法

### 暴力匹配算法

如果用暴力匹配的思路，并假设现在str1匹配到i位置，字串str2匹配到j位置，则有：

1. 如果当前字符匹配成功（即str1[i]==str2[j]）,则i++，j++，继续匹配下一个字符
2. 如果失败（即str1[i]==str2[j]）,令i=i-(j-1),j=0.相当于每次匹配失败时，i回溯，j被置为0
3. 用暴力方法解决的话就会有大量的回溯，每次只移动一位，若是不匹配，移动到下一位接着判断，浪费了大量的时间（不可行）

### KMP算法

### 贪心算法

1. 贪心算法是指在对问题进行求解时，在每一步选择中都采取最好或者最优（即最有利的）的选择，从而希望能够导致结果是最好或者是最优的算法
2. 贪婪算法所得到的结果不一定是最优的结果（有时候回事最优解），但是都是相对近似（接近）最优解的结果

### 普利姆算法

#### MST（最小生成树）

1. 给定一个带权的无向连通图，如何选取一刻生成树，使树上所有边上权的总和为最小，这叫最小生成树
2. N个顶点，一定有N-1条边
3. 包含全部顶点
4. N-1条边都在图中
5. 求最小生成树得算法主要有普利姆算法和克鲁斯卡尔算法

#### 普利姆算法

1. 普利姆算法求最小生成树，也就是包含n个顶点的连通图中，找出只有（n-1）条边包含所有n个顶点的连通子图，也就是所谓的极小连通子图
2. 步骤：
3. 设G=（V,E）是连通网，T=（U,D）是最小生成树，V,U是顶点集合，E,D是边的集合
4. 若从顶点u开始构造最小生成树，则从集合V中取出顶点u放入集合U中，标记顶点v的visited[u]=1
5. 若集合U中顶点ui与集合V-U中的顶点vj之间存在边，则寻找这些边中权值最小的边，但不能构成回路，将顶点vj加入到集合U中，将边（ui，vj）加入到集合D中，标记visited[vj]=1
6. 重复步骤②，直到U与V相等，即所有顶点都被标记为访问过，此时D中有n-1条边

#### 克鲁斯卡尔算法-

1. 克鲁斯卡尔算法，是用来求加权连通图的最小生成树的算法。
2. 基本思想：按照权值从大到小的顺序选择n-1条边，并保证这n-1条边不构成回路
3. 基本做法：首先构造一个只含有n个顶点的森林，然后权值从小到大从连通网中选择加入到森林中，并使森林不产生回路，直至森林变成一棵树为止
4. 如何判断是否产生回路：我们加入的边不能都指向同一个终点，否则构成回路
5. 关于终点的说明：就是将所有顶点按照从小到大的顺序排列好之后：某个顶点的终点就是与它连通的最大顶点

### 迪杰斯特拉算法

-----迪杰斯特拉算法是典型最短路径算法，用于计算一个节点到其他节点的最短路径，它的主要特点是以起始点为中心向外层层扩展（广度优先搜索思想），直到扩展到终点为止

算法过程：设置出发顶点为V，顶点集合V{V1，V2，Vi……},v到V中各顶点的距离构成距离集合Dis，Dis{d1，d2，di……}，Dis集合及记录着v到图中各顶点的距离（到自身可以看做0，v到vi距离对应为di）

1. 从Dis中选择值最小的di并移出Dis集合，同时移出V集合中对应的顶点vi，此时的v到vi即为最短路径
2. 更新Dis集合，更新规则为：比较v到V集合中顶点的距离值，与v通过vi到V集合中顶点的距离值，保留值较小的一个（同时也应该更新顶点的前驱节点为vi，表明是通过vi到达的）
3. 重复执行两步骤，直到最短路径顶点为项目顶点即可结束

### 马踏棋盘算法

1. 创建期盼chessBoard，是一个二维数组
2. 将当前位置设置成已经访问过了，然后根据当前位置，就算马儿还能走那些位置，并放入到一个集合中（ArrayList），最多有8个位置，每走一步，就使用step+1
3. 遍历ArrayList中存放的所有位置，看看那个可以走通，如果走通，就继续，走不通，就回溯
4. 判断马儿是否完成了任务，使用step和应该走的步数比较，如果没有达到数量，则表示没有完成任务，将整个棋盘置0
5. 注意：马儿不同的走法，会得到不同的结果，效率也会有影响（优化）