### • Algorithmic asymptotic analysis

### – Big-Oh, big-Theta, and big-Omega

### • ADTs: Lists, stacks, and queues

### • Sorting

### – Insertion, merge sort, quick sort, heap sort, radix, etc.

### – Lowest bound on sorting

### • Search trees

### – Binary search tree

### – AVL tree

### – B- tree

### • Hashing

### • Graphs

### – Breadth-first search and Depth-first search

### – Shortest paths

### – Minimal spanning trees

# 引入 基本概念和术语

## 数据 (Data)

• 数据是客观事物的符号表示，在计算机科学中指的是所有能输

入到计算机中并被计算机程序处理的符号的总称。

## 数据元素 (Data Element)

• 数据元素是数据的基本单位，在程序中通常作为一个整体来进

行考虑和处理。

• 一个数据元素可由若干个数据项 (Data Item)组成。数据项是数

据的不可分割的最小单位。数据项是对客观事物某一方面特性

的数据描述。

## 数据对象 (Data Object)

• 数据对象是性质相同的数据元素的集合，是数据的一个子集。

如字符集合 C = {‘A’, ’B’, ’C’, … } 。

## 4. 数据的逻辑结构

• 数据结构 (Data Structure) 是指相互之间具有 (存在) 一定联系

(关系) 的数据元素的集合。数据元素之间的相互联系 (关系) 称

为数据的逻辑结构，它们可以是自然的或者是人为约定的。数

据的逻辑结构有四种基本类型：

– 集合：结构中的数据元素除了“同属于一个集合”外，没有

其它关系。

– 线性结构：结构中的数据元素之间存在一对一的关系。

– 树型结构：结构中的数据元素之间存在一对多的关系。

– 图状结构或网状结构：结构中的数据元素之间存在多对多的

关系。

## 数据结构的存储方式 (数据的物理结构)

• 数据结构在计算机内部的存储方式包括数据元素的存储和数据

元素之间的逻辑关系的表示，也称为数据的物理结构。

– 数据元素之间的逻辑关系在计算机内部有两种不同的表示方

法：顺序表示和非顺序表示。由此得出两种不同的存储结构： 顺序存储结构和链式存储结构。

– 顺序存储结构：

»用数据元素在存储器中的相对位置来表示数据元素之间 的逻辑关系，通常表现为连续的存储地址。

– 链式存储结构：

»在每一个数据元素中增加一个存放另一个元素地址的指 针 (pointer )，用该指针来表示数据元素之间的逻辑关系， 因此对地址的连续性没有要求。

## 数据类型 (Data Type)

• 数据类型指的是一个值的集合和定义在该值集上的一组操作的

总称。

• 数据类型是和数据结构密切相关的一个概念。

– 在 C 语言中数据类型有：基本类型和构造类型。

• 数据结构不同于数据类型，也不同于数据对象，它不仅要描述

数据类型的数据对象，而且要描述数据对象各元素之间的相互

关系。

## 7. 抽象数据类型 (Abstract Data Type, ADT)

• ADT 指一个数学模型以及定义在该模型上的一组操作。

• ADT 仅仅是一组逻辑特性描述， 与其在计算机内的表示和实现

无关。因此，不论 ADT 的内部结构如何变化，只要其数学特性

不变，都不影响其外部的使用特征。

• ADT 的形式化定义

ADT = (D, S, P)

其中：D 是数据对象，S 是 D 上的关系集，P 是对 D 的基本

操作集。

# Part 1 算法分析——时间复杂度、空间复杂度

##### O(big-Oh)：

首先，这是我们在学习工作中描述算法时间复杂度用的最普遍的符号。它是渐进上界，其作用是将我们得到的算法在最坏情况下（worst case）时间复杂度表达式简化成对应的多项式（比如n^2等）。所以在我们证明的过程中，目的是证明我们的式子要“小于等于”目标多项式。

Ω(big-Omega)：

这个符号我们一般用的比较少，一个是因为我们一般不会去考虑算法运行时间的下界，另一个是因为下界时间也不好证明。没错，他就是渐进下界，其作用是将我们得到的算法在最好情况下（best case）时间复杂度表达式简化成对应的多项式（也比如n^2等）。所以在我们证明的过程中，目的是证明我们的式子要“大于等于”目标多项式。

Θ(big-theta)：

如果O和Ω可以用同一个多项式表示，那么这个多项式就是我们所要求的渐进紧的界了。其作用是将我们可以较准确地得到算法的时间复杂度表达式对应的多项式（也比如n^2等）。所以在我们证明的过程中，目的是证明我们的式子要“等于”目标多项式。

##### 1、加法规则

T(n,m) = T1(n) + T2(m) = O(max{f(n), g(m)})

##### 2、乘法规则

T(n,m) = T1(n) \* T2(m) = O(max{f(n)\*g(m)})

##### 3、<bb题>

(1) Assuming that T1(n) = O(f1(n)), T2(n) = O(f2(n))，prove that

T(n) = T1(n) + T2(n) = O(f1(n)) + O(f2(n)) = O(max(f1(n), f2(n))).

设T1 是 O(f1)， T2是 O(f2)

因为 T1(n) = O(f1(n)), 所以存在C1 > 0 和 N1，当 n > N1 时有 |T1(n)| < C1|f1(n)|；

对T2(n) = O(f2(n)) 同理可得 存在C2> 0 和 N2，当 n > N2 时有 |T2(n)| < C2|f2(n)|，

从而当 n > max(N1, N2)时有：

|(T1 + T2)(n)| = |T1(n) + T2(n)| <= |T1(n)| + |T2(n)| <= C1|f1(n)| + C2|f2(n)| <= C| max( |f1(n)|, |f2(n)| ) | = C| max(f1, f2)(n) |

C = C1 + C2, 所以 T(n) = T1(n) + T2(n) = O(f1(n)) + O(f2(n)) = O(max(f1(n), f2(n))). 得证

1. Suppose that T(n)=2T(n/2) + n, T(1) = 1. Prove that

T(n) = O(n log n) by induction.

T(2) = 2T(1) + 2 = 4 = 2 \* 2^1;

T(4) = 2T(2) + 4 = 12 = 3 \* 2^2;

T(8) = 2T(4) + 8 = 32 = 4 \* 2^3;

……

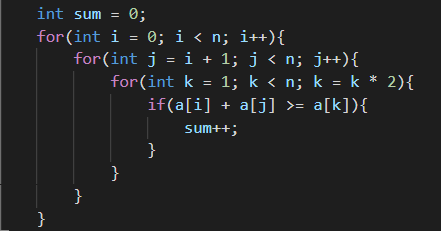
T(2^m) = (m + 1) \* 2^m;

令 n = 2^m; m = logn;

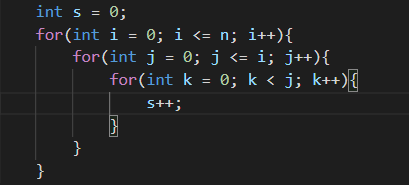
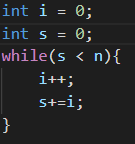
则T(n) = (logn + 1) \* n = O(nlogn);



### 时间复杂度

1.  **O(n^2 log n)**

**遇到这样多个循环嵌套的不光要看初始值，还要看判断条件之后的步骤**

1.   **O(n^3)**
2.   **O(sqrt(n))**
3. If T(n) = 6 \* 2^n + n^3, T(n) = Ω(n^100) (√)
4. If n = 1, T(n) = 1; if n > 1, T(n) = 3T(n/4) + n ; T(n)=O(n) (×)
5. If 3n + 12 <= T(n) <= 3n^2 + 5,then T(n) = Θ(n log n) (×)

|  |
| --- |
|  |
| |  |  | | --- | --- | | 所选答案： | IMG_257 错 |   11个 |

斐波那契数列

递归算法的空间复杂度 O(n)

迭代算法的空间复杂度 O(1)

递归/迭代算法的时间复杂度 O(n)

# Part 2 ADT: List, stack, and queue

### Vector & List

1. For STL vector, a vector’s iterators are invalidated when its memory is reallocated. (√)

<对于STL vector，在重新分配内存时，向量的迭代器将失效。>

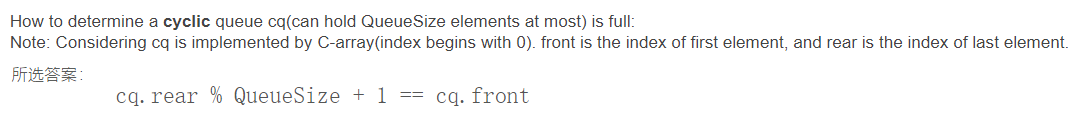
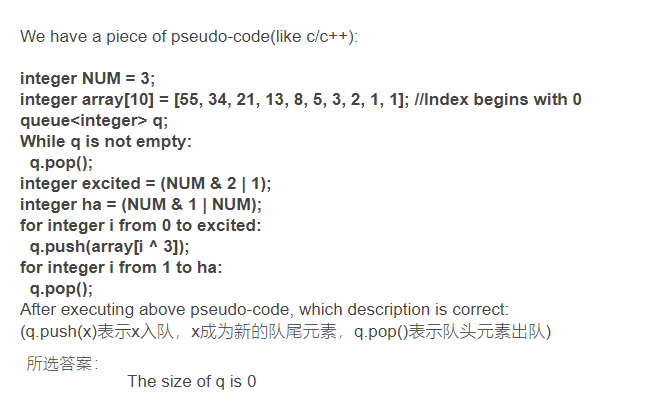
1. For STL vector, memory will be reallocated automatically if more than capacity-size elements are inserted into the vector. (√)

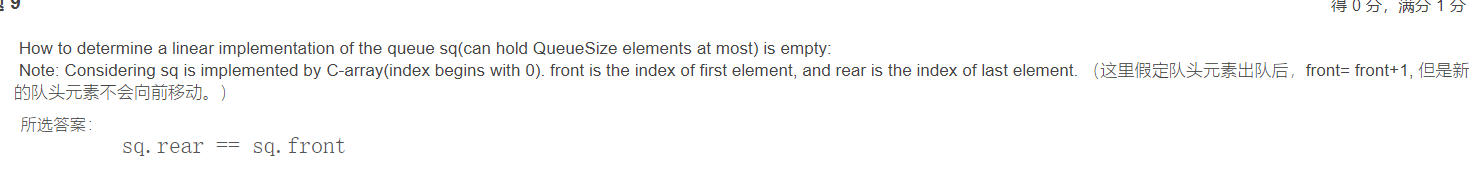
<对于STL向量，如果在向量中插入超过容量大小的元素，内存将自动重新 分配。>

1. STL vector reallocation will change both size() and capacity()(×)
2. 对于连续结构存储的线性表(数组)，时间复杂度:访问结点O(1)、增加结点O(n)、删除结点O(n)
3. 线性表L={a1, a2,…, an}采用连续结构存储，假定在不同的n+1个位置上插入的概率相同，则插入一个新元素平均需要移动的元素个数为 n/2
4. 如果最常用的操作是取第i个节点及其前驱，则采用顺序表的存储方式最节省时间。
5. Both push\_back() and pop\_back() in STL list take constant time.(√)
6. Both push\_front() and pop\_front() in STL list take constant time.(√)
7. 某线性表最常用的操作是在最后一个结点之后插入一个结点或删除第一个结点，故采用仅有尾指针的单循环链表

注意：带头结点和不带头结点的区别

### Stack & Queue

1. 
2. 



1. 深度优先搜索依据的数据结构时stack，广度优先搜索依据的是queue

### Map

# Part 3 Sorting

1、冒泡排序

最简单的一种排序算法。假设长度为n的数组arr，要按照从小到大排序。则冒泡排序的具体过程可以描述为：首先从数组的第一个元素开始到数组最后一个元素为止，对数组中相邻的两个元素进行比较，如果位于数组左端的元素大于数组右端的元素，则交换这两个元素在数组中的位置，此时数组最右端的元素即为该数组中所有元素的最大值。接着对该数组剩下的n-1个元素进行冒泡排序，直到整个数组有序排列。算法的时间复杂度为O(n^2)。

算法伪代码

```

for i <- 0 to n - 1

for j <- 0 to n - 1 - i

if array[j] > array[j + 1]

temp <- array[j];

array[j] <- array[j + 1]

array[j + 1] <- temp

j <- j + 1

i <- i + 1

```

2、选择排序

严蔚敏版《数据结构》中对选择排序的基本思想描述为：每一趟在n-i+1(i=1,2,...,n-1)个记录中选取关键字最小的记录作为有序序列中第i个记录。具体来说，假设长度为n的数组arr，要按照从小到大排序，那么先从n个数字中找到最小值min1，如果最小值min1的位置不在数组的最左端(也就是min1不等于arr[0])，则将最小值min1和arr[0]交换，接着在剩下的n-1个数字中找到最小值min2，如果最小值min2不等于arr[1]，则交换这两个数字，依次类推，直到数组arr有序排列。算法的时间复杂度为O(n^2)。

选择排序伪代码

```

Select\_Sort (input arr[],input length)

for i <- 1 to length step 1

min <- i

for j <- i+1 to length step 1

if arr[j] < arr[min]

min <- j

end if

swap(arr[j],arr[min])

End

```

3、插入排序

 插入排序的基本思想就是将无序序列插入到有序序列中。例如要将数组arr=[4,2,8,0,5,1]排序，可以将4看做是一个有序序列(图中用蓝色标出)，将[2,8,0,5,1]看做一个无序序列。无序序列中2比4小，于是将2插入到4的左边，此时有序序列变成了[2,4]，无序序列变成了[8,0,5,1]。无序序列中8比4大，于是将8插入到4的右边，有序序列变成了[2,4,8],无序序列变成了[0,5,1]。以此类推，最终数组按照从小到大排序。该算法的时间复杂度为O(n^2)。

插入排序算法伪代码

```

Algorithm sort(A)

Input: a array, A, size n

for j = 2 to A.length

key <- A[j]

i <- j - 1

while i > 0 and A[i] > key

A[i+1] <- A[i]

i <- i - 1

A[i+1] <- key

```

4、快速排序

快速排序的基本思想是：通过一趟排序将待排记录分割成独立的两部分，其中一部分记录的关键字均比另一部分记录的关键字小，则可分别对这两部分记录继续进行排序，已达到整个序列有序。一趟快速排序的具体过程可描述为：从待排序列中任意选取一个记录(通常选取第一个记录)作为基准值，然后将记录中关键字比它小的记录都安置在它的位置之前，将记录中关键字比它大的记录都安置在它的位置之后。这样，以该基准值为分界线，将待排序列分成的两个子序列。

一趟快速排序的具体做法为：设置两个指针low和high分别指向待排序列的开始和结尾，记录下基准值base\_val(待排序列的第一个记录)，然后先从high所指的位置向前搜索直到找到一个小于base\_val的记录并互相交换，接着从low所指向的位置向后搜索直到找到一个大于base\_val的记录并互相交换，重复这两个步骤直到low=high为止。平均n log n; 最坏n^2

算法代码

```

sort(int a[],int min,int max) {

int key=a[min];//准基数

int start=min; //开始位置

int end =max;//结束位置

while(end>start) { //循环条件是否数值交叉

while(end>start&&a[end]>=key) {//从后开始往前查找

end--;

}//如果找到的值小于基数值，那么进行值交换

if(a[end]<key) {

int temp=a[end];

a[end]=a[start];

a[start]=temp;

}

while(end>start&&a[start]<=key) {//从前往后找

start++;

}//如果找到的值大于基数值，那么进行值交换

if(a[start]>key) {

int temp=a[end];

a[end]=a[start];

a[start]=temp;

}

}

//这部分的数据都是小于准基数，通过递归在进行一趟快排

if(start>min) {

sort(a, min, start-1); //开始位置为第一位，结束位置为关键索引-1

}

if(end<max) {

sort(a, end+1, max); //开始位置为关键索引+1，结束位置最后一位

}

}

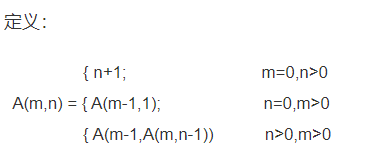
···

5、归并排序

6、基数排序



### **阿克曼函数**



1.

|  |
| --- |
| The value of ackermann(0,0) is \_\_\_ .  答案 |
| |  |  | | --- | --- | | 所选答案： | IMG_256  1 | |

2.

|  |
| --- |
| The value of ackermann(0,5) is \_\_\_ .  答案 |
| |  |  | | --- | --- | | 所选答案： | IMG_256  6 | |

3.

|  |
| --- |
| The value of ackermann(4,0) is \_\_\_ .  答案 |
| |  |  | | --- | --- | | 所选答案： | IMG_256  13 | |

4.

|  |
| --- |
| The value of ackermann(3,10) is \_\_\_ .  答案 |
| |  |  | | --- | --- | | 所选答案： | IMG_256  8189 | |

5.

|  |
| --- |
| The value of ackermann(3,3) is \_\_\_ .  答案 |
| |  |  | | --- | --- | | 所选答案： | IMG_256  61 | |

6.

|  |  |
| --- | --- |
|  | The value of ackermann(4,3) is  答案 |



### **hanoi(n, start, end, temp)**

1.

|  |
| --- |
| The time complexity for recursive hanoi(n, start, end, temp), which solves the hanoi tower problem of size n,  is *O*(2*n*).  答案 |
| |  |  | | --- | --- | | 所选答案： | IMG_256 对 | |

2.

|  |
| --- |
| The space complexity for recursive hanoi(n, start, end, temp), which solves the hanoi tower problem of size n,  is *O*(*n*).  答案 |
| |  |  | | --- | --- | | 所选答案： | IMG_256 对 | |

# Part 4 Search trees

二分查找时间复杂度O(log n)

### Binary search tree

1.定义：所有

### – AVL tree

### – B- tree

外部路径长度：从扩充的二叉树的根到每个外部结点的路径长度之和称为外部路径长度（E）

内部路径长度：扩充的二叉树里从根到每个内部结点的路径长度之和称为内部路径长度（I），

它们之间的关系满足E=I+2N（N为内部结点数）。

外部节点：即叶节点

内部节点：除内部节点外的节点

### B树、B+树

### 引入

二叉查找树、平衡二叉查找树、红黑树，是典型的二叉查找树结构，其查找时间复杂度为O（log2N）与树的深度有关，想要提高查找效率，则就要降低树的高度。但是树节点存储的元素数量是有限的（如果过多，就退化成节点内部的线性查找了），这样导致二叉查找树结构由于树的深度过大而造成磁盘I/O读写过于频繁，从而导致查询效率低下，而解决该问题一个基本的想法是：采用多叉树结构。

### **扩展知识**

1. 计算机存储设备：内存储器和外存储器。内存存取速度快但容量小，价格贵，且不能长期保存数据；外存储器——磁盘是一种直接存取的存储设备。存取时间变化不大，可以直接存取任何字符组，且容量大、速度较其他外存设备更快。
2. 所以，在大规模数据存储方面，大量数据存储在外存磁盘中，而在外存磁盘中读取/写入块(block)中某数据时，首先需要定位到磁盘中的某块，如何有效地查找磁盘中的数据，需要一种合理高效的外存数据结构，就是下面所要重点阐述的B-tree结构，以及相关的变种结构：****B+-tree结构和B\*-tree结构。****

#### B树

1. m阶的B树具有如下几个特征:
2. 根节点至少有两个孩子
3. 每个中间节点都包含k - 1个元素和k 个孩子,其中 m / 2 <= k <= m
4. 每个叶子节点都包含k - 1 个元素,其中 m / 2 <= k <= m
5. 所有的叶子节点都位于同一层
6. 每个节点中的元素从小到大排列,节点当中k - 1个元素正好是k个孩子包含的元素的值域的划分

#### B+树

1. 一个m阶的B+树有如下几个特征
2. 有k个子树的中间节点包含有k个元素,每个元素不保存数据,只用来索引,所有数据都保存在叶子节点
3. 所有的叶子节点中包含了全部元素的信息,即指向含这些元素记录的指针,且叶子节点本身依关键字的大小自小而大顺序链接
4. 所有的中间元素都同时存在于子节点,在子节点元素中是最大(或最小)元素
5. 插入

|  |  |  |
| --- | --- | --- |
| 叶节点 | 中间节点（父亲） | 操作 |
| 不满 | / | 直接插入 |
| 满 | 不满 | 1. **拆分叶子节点** 2. 将叶子节点的中间元素放入父亲节点 3. 小于中间元素的记录放到左子树， 4. 大于或等于中间元素的放到右子树 |
| 满 | 满 | 1. **拆分叶节点** 2. 将叶子节点的中间元素放入父亲节点 3. 小于中间元素的记录放到左子树， 4. 大于或等于中间元素的放到右子树 5. **拆分中间节点** 6. **中间元素再向上顶** |

1. 删除

如果叶子结点中没有相应的key，则删除失败。否则执行下面的步骤

1）删除叶子结点中对应的key。删除后若结点的key的个数大于等于Math.ceil(m-1)/2 – 1，删除操作结束,否则执行第2步。

2）若兄弟结点key有富余（大于Math.ceil(m-1)/2 – 1），向兄弟结点借一个记录，同时用借到的key替换父结（指当前结点和兄弟结点共同的父结点）点中的key，删除结束。否则执行第3步。

3）若兄弟结点中没有富余的key,则**当前结点和兄弟结点合并成一个新的叶子结点**，并删除父结点中的key（父结点中的这个key两边的孩子指针就变成了一个指针，正好指向这个新的叶子结点），将当前结点指向父结点（必为索引结点），执行第4步（第4步以后的操作和B树就完全一样了，主要是为了更新索引结点）。

4）若索引结点的key的个数大于等于Math.ceil(m-1)/2 – 1，则删除操作结束。否则执行第5步

5）若兄弟结点有富余，父结点key下移，兄弟结点key上移，删除结束。否则执行第6步

6）当前结点和兄弟结点及父结点下移key合并成一个新的结点。将当前结点指向父结点，重复第4步。

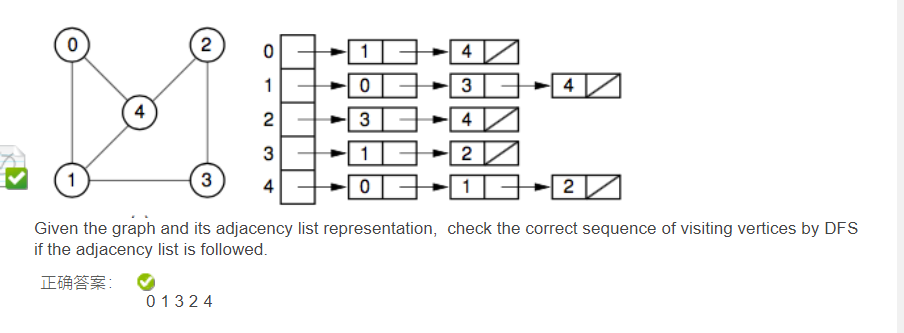
**B+ tree**

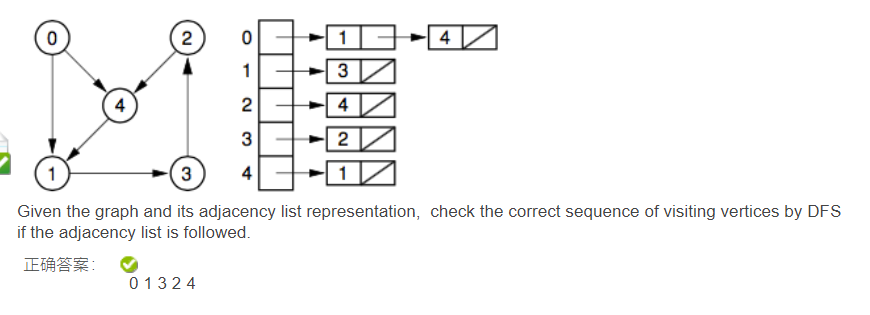
1. Suppose we will build a 512-way B+ tree for 10^9 entries, and each leaf can store 39 entries. Assuming the worst case that all leaves and internal nodes are half full except the root which may has only two children, then the tree has height \_5\_.
2. Suppose we will build a 512-way B+ tree for 10^9 entries, each leaf can store 39 entries. The optimal tree has height \_3\_.
3. Suppose our key is 4 bytes and we are storing 200 byte per record, and assume a link is 8 bytes.With a block of 4 KB, the number of links an internal node can store, the order M of the B+ tree is 341.
4. Suppose our key is 4 bytes and we are storing 100 byte per record.With a block of 4 KB, how many links an internal node can store, what is the value for M (the order of the B+ tree)? 512
5. Suppose our key is 4 bytes and we are storing 200 bytes associated value, i. per record takes 204 bytes. Assume a link takes 8 bytes.With a block of 4 KB, the number of records a leaf can store (or the value for L) is \_20\_
6. Suppose our key is 4 bytes and we are storing 100 bytes associated value, i.per record takes 104 bytes. Suppose a link is 4 bytes.With a block of 4 KB, how many records a leaf can store, what is the value for L?  39

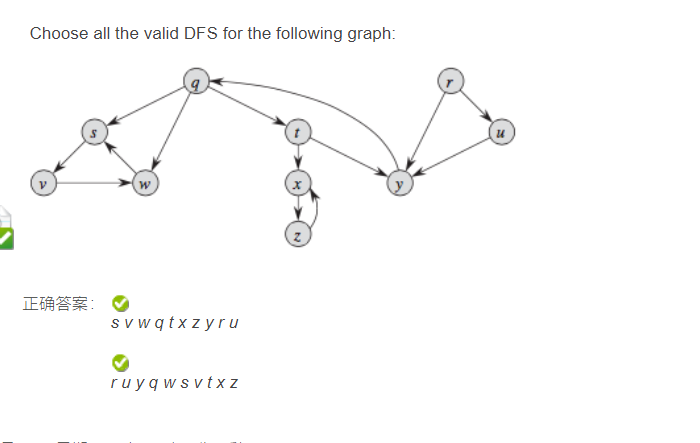
<block 及结点， 算M: link \* M + key \*(M - 1) = block; 算L: block / per = L; 算高度，最优->log M, 最坏 -> log(M/2) N N = #leaves)

图

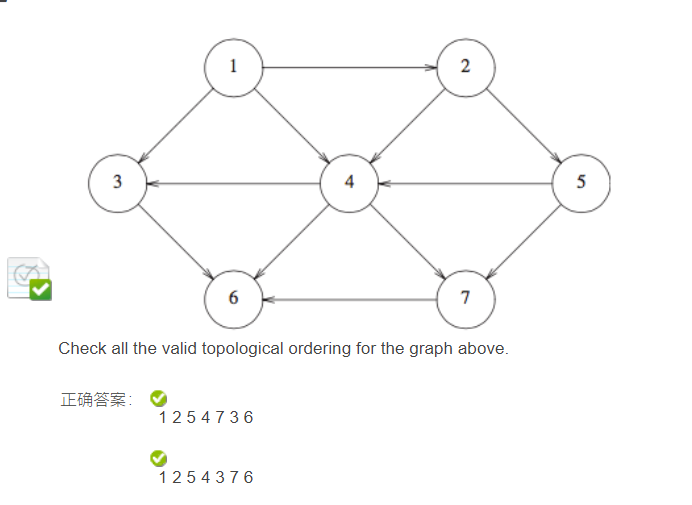
DFS



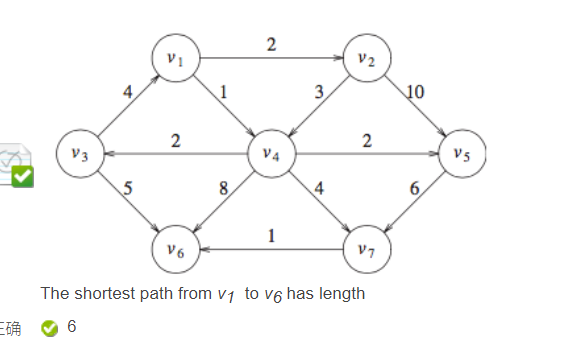




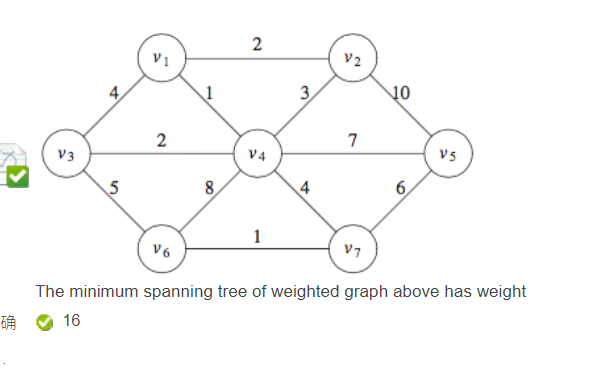
拓扑



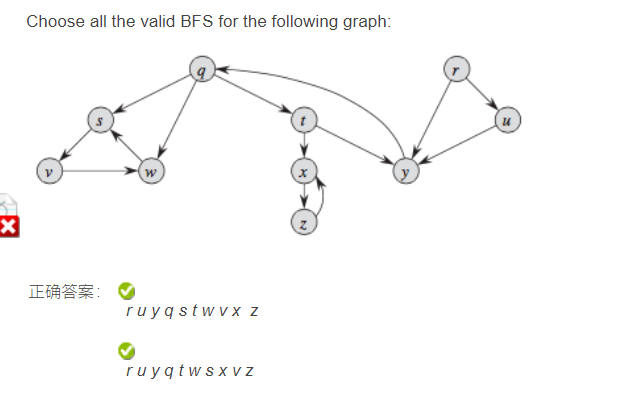
最短路径

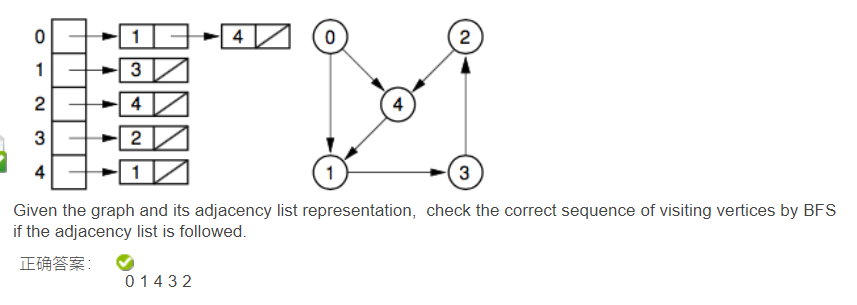


最小生成树



BFS





图

完全图：在图论的数学领域，完全图是一个简单的无向图，其中每对不同的顶点之间都恰连有一条边相连。完整的有向图又是一个有向图，其中每对不同的顶点通过一对唯一的边缘（每个方向一个）连接。n个端点的完全图有n个端点以及n(n − 1) / 2条边，以Kn表示。它是(k − 1)-[正则图](https://baike.baidu.com/item/%E6%AD%A3%E5%88%99%E5%9B%BE" \t "https://baike.baidu.com/item/_blank)。所有完全图都是它本身的团（[clique](https://baike.baidu.com/item/clique/6944698" \t "https://baike.baidu.com/item/_blank)）。