1. Data structure overview

1) Different access disciplines

No structure-just a collection of values, linear structure of values-the order matters, set of key-value pairs, hierarchical structures, grid/table

2) Information hiding: related to privacy & security

a. Definition: Information hiding is the principle that users of a module need to know only the essential details of this module (as identified by abstraction). The important details of a module that a user needs to know form the **specification** of a module. information hiding means that modules are used via their specifications, not their implementations. 关注是能 specification 不能 implementations

b. Examples: 一个车很多的组件但是人们只需要学会如何开车，并不需要很清楚每一个组件做的什么事情

c. Use cases: 隐藏数据的物理存储布局，这样如果数据被更改，则更改仅限于整个程序的一小部分

d. Information hiding in O-O (object-oriented) world: **hides** the internal details both of how it performs these services and of the data (attributes & structures) that it maintains in order to support these services.

3) Java method signature: method's name + number and types of the parameters and their order

E.g. public void setMap (int x, int y) The method signature is **setMapReference(int, int)**

4) Encapsulation: as a process / entity

a. Definition: As a **process**, encapsulation means the act of enclosing one or more items (data/functions) within a (physical or logical) container. As an **entity**, encapsulation, refers to a package or an enclosure that holds (contains, encloses) one or more items (data/functions).

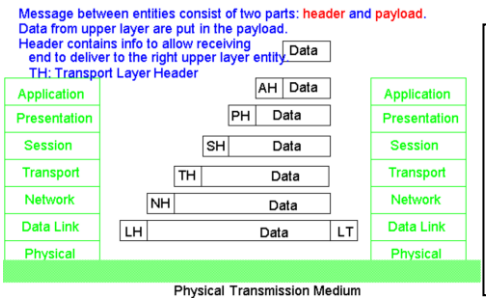
b. Wall / barrier: The separator between the inside and the outside of this enclosure

c. Encapsulation in an O-O world: inclusion within an object of all the resources needed for the object to function – i.e., the methods and the data.

d. Encapsulation in communication: inclusion of one data structure within another structure so that the first data structure is hidden.

E.g. TCP/IP-formatted packet can be encapsulated within an ATM frame.

e. Packet encapsulation:



f. advantages of encapsulation:
Simpler, modular programs; **Side-effects** from direct manipulation of data are **eliminated or minimized**; **Localization of errors** (only methods defined on a class can operate on the class data), which allows localized testing; Program modules are **easier to read, change, and maintain.**

5) Summary of abstraction & information hiding & encapsulation & Data strucutre

a. **Abstraction** is a technique that helps us identify which specific information is important for the user of a module, and which information is unimportant.

b. **Information hiding** is the principle that all unimportant information should be hidden from a user.

c. **Encapsulation** is then the technique for packaging the information in such a way as to hide what should be hidden, and make visible what is intended to be visible.

d. **Data structures** represent one big common factor across programs. Specification of data structures requires the use of abstraction, info hiding, encapsulation

6) Efficiency: time efficiency & space efficiency

a. Space: try to minimize memory usage (avoid the allocation of unnecessary space) E,g, Storing a drawing/map via vectors VS. bitmaps VS. compressed bitmaps; Tradeoffs of space efficiency vs convenience.

b. Time: will include operations that are efficient in terms of speed of execution (based on some well chosen algorithm) E.g. binary search VS. linear search INT102 具体有讲 Big-O notation

7) Dynamic & static data structures: 静态还是动态是一种结构而不是数据

a. Dynamic: grow or shrink during run-time to fit current requirements 在运行期间增长或收缩以适应当前需求 e.g. a structure used in modelling traffic flow. **Ad:** no requirement to know the exact number of data items,

efficient use of memory space. **Dis:** memory allocation/de-allocation overhead, whenever a dynamic data structure grows or shrinks, then memory space allocated to the data structure has to be added or removed (which requires time).

b. Static: are fixed at the time of creation 在创建时运行时间是固定的 E.g. a structure used to store a postcode or credit card number (which has a fixed format)- int [ ] a = new int [50]; **Ad:** ease of specification, no memory allocation overhead. **Dis:** need to make sure is enough capacity, more errors and wastes.

2. ADT：outcome of abstraction / encapsulation

1) Interface: each interface corresponds to an ADT
接口是 Java 编程中一个抽象类型，是抽象方法的集合，接口通常以 interface 来声明，一个 class 通过继承接口的方式，从而继承接口的抽象方法
定义接口: [public] interface 接口名 [extends 父接口名] { }
实现接口: [修饰符] class implement 接口名 { }

2) 接口的特点：
No constructors, no fields, no method bodies
接口中的方法会被隐式指定为 public abstract 使用其他修饰符会报错；接口可以含有变量。但是只能 public static final 修饰（不可以用 private 修饰），接口的方法都是公有的且都需要在类中实现；接口不是为了被 extend 而是要被 implement；接口支持多继承

3) Collection: bag, set, map, list, queue, stack, graph, tree 都继承 collection 类
isEmpty()返回 Boolean 类型, size(), contains(item)返回 Boolean 类型, add(item), remove(item), iterator()返回迭代器

4) List: 继承自 collection 类
   a. 特点：没有访问限制（随机访问），允许重复
   b. Methods
      Add(index, item), remove(index), get(index), set(index, item), indexOf(item)返回元素下标, subList(from, to)返回子 list
   c. List classes include ArrayList, LinkedList

5) ArrayList: 继承自 list 类
   a. 特点：相比于 Array，ArrayList 可以自动扩容
   b. Methods
      Size(), add(item), add(index, item), set(index, item)取代 index 位置的 item, contains(item), get(index), remove(item), remove(index)
   c. Remove 方法移除元素后，删除原 ArrayList 的最后一位
   d. Time complexity:

6) AbstractList：继承自 list 类
   a. 特点：没有访问限制（随机访问），允许重复
   b. Methods
      声明了 size()，get(index)等方法；定义了 isEmpty(), add(), contains(),clear()等方法

7) LinkedList：继承自 AbstractList
   a. 特点：其内元素有顺序，每个节点会记录下一个节点的地址，形成链表，访问时必须顺序访问，元素可重复
   b. Methods
      Get(index)返回节点值, set(index, item)设立节点值, add(index, item)添加新元素, remove(item)删除某一元素, remove(index)删除下标处元素
   c. Traverse the list by following the links, insert by changing of links, remove by changing links

8) Bag: 继承自 list 类
   a. 特点：没有访问限制（随机访问），bag 内元素没有顺序，允许重复
   b. Methods
      Add(value)返回操作状态, remove(value)返回操作状态, contains(value)返回是 Boolean 类型, findElement(value)返回匹配的元素，size(), isEmpty(), iterator(), clear(), addAll(collection), removeAll(collection), containsAll(collection)

9) Set: 继承自 list 类
   a. 特点：没有访问限制（随机访问），**set 内不允许重复项**，set 内元素没有顺序

b. Methods

Add(value)返回操作状态 元素重复返回 false, remove(value)返回操作状态, contains(value)返回 Boolean 类型, findElement(value)返回匹配的元素, clear()清空

c. Set classes include HashSet, TreeSet

10) ArraySet
    a. 特征：其内元素无顺序，可以实现自动扩容。随机访问，**排除重复**
    b. Methods:

    Contains(item), add(item), remove(item)与 ArrayListremove 方法不同的是无需将之后的元素前移而只是把最后一位元素挪入空位

11) SortedArraySet
    a. 特点：与 ArraySet 类似，只是元素在存储时按照**排序存储，**因此**定位元素**时可以二分存储，加速查找速度
    b. Methods：类似 ArraySet
12) Stack: 继承自 list
    a. 特点：stack 内元素**有顺序**，先进先出，后进后出，**顺序访问**，允许重复项出现
    b. Methods

    Push(value)将元素 入栈，放置在栈顶, pop()从栈顶移除一个元素, peek()/top()不弹出栈顶元素并返回

    c. Stack 的应用

    Processing files of structures (nested) data (HTML, XML), program execution, undo in editors, expression 前中后缀的应用，利用 stack 来判断括号是不是平衡的，在递归中应用 stack

13) Map: 继承自 list
    a. 特点：其内元素无顺序，无访问限制（随机访问），key 不能重复，每个元素存储为（key，value）
    b. Methods

    get(key): 返回该 key 对应的 val, put(key, value): 重新设定 key 对应的 val, remove(key): 移除 key 为指定 key 的元素, containsKey(key)返回是否包含 key 为指定 key 的元素, keySet(): 返回 keys 的 Set 对象, values(): 返回 vals 的 Collection 对象, entrySet(): 返回所有元素的 Set 对象

14) Queue: 继承自 list
    a. 特点：其内元素无顺序 ，删除操作只能在一段进行，允许重复项出现
    b. Methods

    Offer(value)添加元素进队列入端，返回操作状态，也叫做“enqueue”, poll()移除并返回出端的一个元素，返回具体的值，也叫“dequeue”, peek()不移除地返回出端元素, remove()移除并返回出端元素，对 queue 为空的时候抛出异常, element()非移除地返回出端元素，对队列为空时抛出异常

15) Array：数组内元素有顺序，随机访问，数组长度不可变，允许重复
16) Linked Stack
    a. 特点：由链表存储的栈结构，头节点方向代表栈顶
    b. Methods：

    Push(item)元素入栈，头节点指向新元素, pop()元素出栈。头节点指向原来第二位的元素, peek()非弹出返回栈顶元素, size()返回链表长度

17) Linked Queue
    a. 特点：由链表存储的队列结构
    b. 链表头节点可以作为队列入端，也可以作为出端，相应地，链表尾节点可以作为出端或者入端。更高效的做法是改变下链表头节点，使头节点有两指针分别指向队列入端元素和队列出端元素（学校 PPT 做法是头指针指向队列出端，尾指针指向队列入端）
    c. Methods：

    Offer(value)添加元素进队列入端返回操作状态, poll()移除并返回出端的一个元素，返回具体的值，也叫“dequeue”, peek()不移除地返回出端的元素

18) Linked Node

节点类，串起来可表示链表；Value 记录值，next 记录指针；setNext()：设立指针指向下一个节点

19) Hash Tables
    a. 特点：use an array named T of capacity N; define a hash function that returns an integer int H(String key); must return an integer between 0 and N-1; store the key and info at T[H(key)]; H() must always return the same integer

for a given key

b. Table size: is usually prime to avoid bias; overly large table size means wasted space; overly small table size means more collisions

c. Hash function: is any well-defined procedure or mathematics function for turning data into an index into array; the values returned by a hash function are called hash values or simply hashes
Hash function can avoid collisions, spreads keys evenly in the array, inexpensive to compute-must be O(1)
e.g. a hash function H is transformation that:
take a variable-size input k and returns a fixed-size (or int), which is called hash value h (that is h = H(k))

d. Modular hash function

Example of a Modular Hash Function

- H(k) = **k mod m** (or **k % m**)
- message1 = '723416'
- hash function = modulo 11
- Hash value$_1$ = (7+2+3+4+1+6) mod 11 = 1
- message2 = 'test' = ASCII '74', '65', '73', '74'
- Hash value$_2$ = (74+65+73+74) mod 11 = 0
- another hash function example: **a*k mod m**

Hash Function for Signed Integer Keys

- remainder after division by table length

```
int hash(int key, int N) {
return abs(key) % N;
}
```

- if keys are positive, you can eliminate the abs

e. Hash function for strings: 范围是 0 到-1，会使用到 ASCII 码表，important insight (letters and digits fall in range 0101 and 0172 octal so all useful information is in lowest 6 bits), hash is O(1)

f. Mid-square method: 用于将输入的关键字映射到散列表的索引位置。该方法的基本思想是将关键字的平方值的中间一部分作为散列值。
e.g. 有一个大小为 10 的 hash table，并且要使用 Mid-square Method 来计算关键字的散列值
第一步，假设我们有一个关键字为 25。第二步，将关键字转换为整数形式，这里关键字本身就是整数。第三步，计算关键字的平方，得到 25 * 25 = 625。第四步，取平方值的中间一部分作为散列值。在这个例子中，我们可以取中间的两位数 62 作为散列值。第五步，将散列值 62 作为关键字在散列表中的索引位置。

g. Hash functions for string keys: this functions takes a string as input and it processes the string 4 bytes at a time, and interprets of the four-byte chunks as a single long integer value.注意一次就是处理四个字节

h. How to deal with collisions
Open addressing, linear probing, quadratic probing, double hashing, separate chaining, each array slot is a searchList, never gets "full", deletions are not a problem

i. How to deal with a full table
Allocate a larger hash table, rehash each from the smaller into larger, delete the smaller

20)

21) Tree: non-linear data strcuture

a. Binary tree 只能一分二 & general tree 可以一分多

b. Some definitions: level / depth / leaf node / non-leaf node / balanced

c. AVL: using rotation to balance

d. tree's 'Methods:
insert(element): 将元素插入到树中; delete(element): 从树中删除指定元素; search(element): 在树中搜索指定元素; getRoot(): 获取树的根节点; getSize(): 获取树的节点数量; getHeight(): 获取树的高度（最长路径的节点数）; isEmpty(): 检查树是否为空

e. binary tree's methods:
getvalue(), getLeft(), getRight(), setValue(), setLeft(tree), setRight(tree)设置左子树, setRight(tree)设置右子树, isleaf(), find(value)返回 value 值的节点

f. general tree's methods:
getChildren()返回所有子节点, getChild(i)返回第 i 个子节点, addChild(child)往 list 里添加一个节点, addChild(i, child)在 list 第 i 个位置添加一个节点

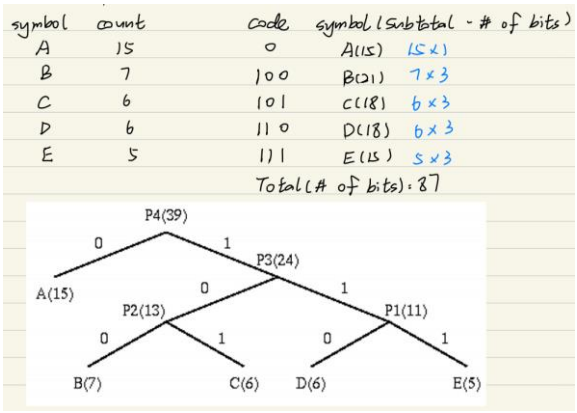
g. 遍历：preorder (

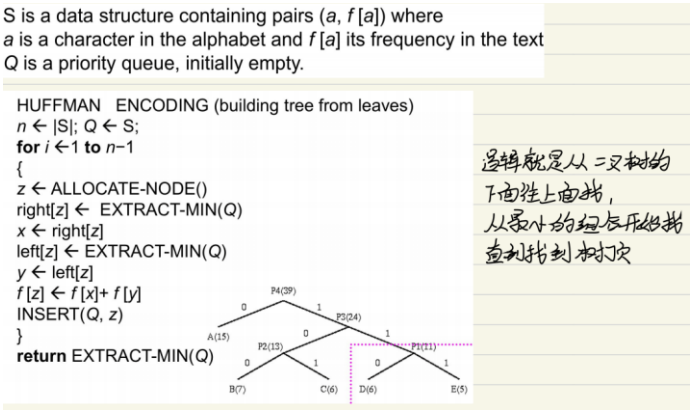
3. Analysis the cost：讨论 ADT 具体的方法的时间复杂度

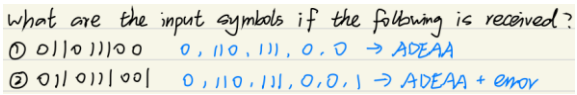
1) ArrayList:
   Get: O(1); set: O(1); remove: O(n); add at an end: O(n) or O(1); add: O(n); contains: O(n); isEmpty: O(1); size: O(1)
2) ArraySet:
   Contains: O(n); add: O(n) or O(1); remove: O(n); isEmpty: O(1); size: O(1); clear: O(1)
3) SortedArraySet:
   binary search: O(logn); add/insertion, remove: O(n); add n item: O(n); O(nlogn) to initialize with n items, with fasting sort; contain: O(logn); isEmpty: O(1); size: O(1); clear(): O(1); iterator: O(1)
4) Stack: O(1)
5) Linked Stack and Queue: O(1) for all main operations
6) LinkedList: O(n) in average for insert, lookup and remove
7) Hash tables: O(1) in average for insert/add, lookup and remove
8) Tree:
   Insert(element): O(logn) or O(n), delete(element): O(logn) or O(n); search(element); getRoot(), getSize(), getHeight(), isEmpty(): O(1)
9) Binary seqrch tree (BST): O(logn) in arerage for insert, lookup and remove, O(n) in worst
10) AVL:

Algorithm	Average	Worst case
Search	$O(\log n)$	$O(\log n)$
Insert	$O(\log n)$	$O(\log n)$
Delete	$O(\log n)$	$O(\log n)$
Space	$O(n)$	$O(n)$

4. Iterator & comparator
1) Iterator & iterable
   a. 可迭代对象（iterable）是指那些可以提供一个迭代器的对象。换句话说，可迭代对象是一类对象，可以被迭代（遍历）。迭代器（iterator）是一个实现了迭代协议的对象，它可以被用于遍历可迭代对象中的元素。**简单来说，可迭代对象是一个容器，它能够提供一个迭代器来遍历它的元素。而迭代器则是负责实际遍历可迭代对象的对象。**
   b. Iterator 接口的方法:
      Next()返回下一个元素，没有抛出异常 StopIteration, hasNext()检查迭代器有没有更多的元素，返回 Boolean 类型, remove()从迭代器的当前位置删除元素
2) Comparator & comparable
   a. "comparable"（可比较对象）是指那些可以被比较大小的对象。"comparator"（比较器）是一个独立的对象，它用于比较两个对象的大小。
   b. Comparator 接口的方法:
      Compare(T object1, T object1)用于比较两个对象的大小，返回一个整数值（如果 object1 小于 object2 返回负整数，object1 等于 object2 返回 0，object1 大于 object2 返回正整数）
   c. Comparable 接口的方法
      Int compreTo(T other)用于比较当前对象与另一个对象 other 的大小关系，返回一个整数值（如果当前对象小于 other 返回负整数，当前对象等于 other 返回零，当前对象大于 other 返回正整数）。用于控制 natural order
5. Huffman coding: is an abstraction example
   通过 Huffman coding 来构架一个 binary tree 也就是二叉树的过程





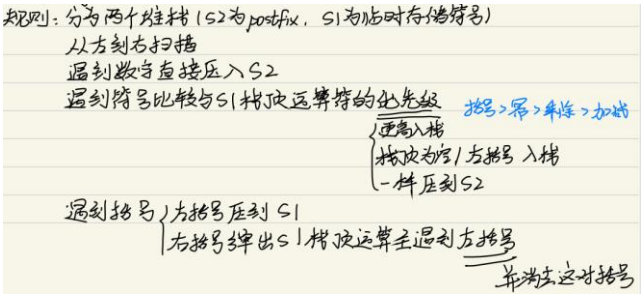


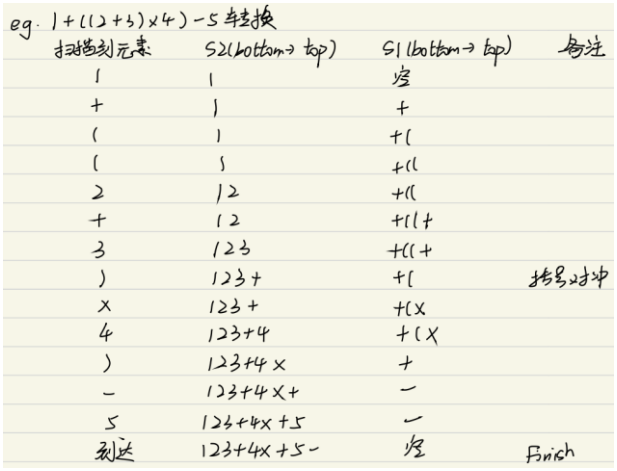
6. Process algebraic expressions:

1) Three types of expressions:
Infix: binary operator between (a + b); Prefix: binary operator before (+ab); Postfix: after (ab+)

2) Checking for balanced ( ), [ ], { }
算法描述：做一个空栈，从这串代码的开始读到末尾。如果读到的字符是一个开放符号——左括号，那么入栈。如果是一个封闭符号——右括号，这时将栈中的元素弹出。如果弹出的元素是封闭符号对应的开放符号，那么正确（正确的时候不做任何提示），否则就报错。如果这时栈为空，那么说明缺失了开放字符，报错。当这串代码读完时，如果栈不为空，那么报错。

3) Transforming infix to postfix



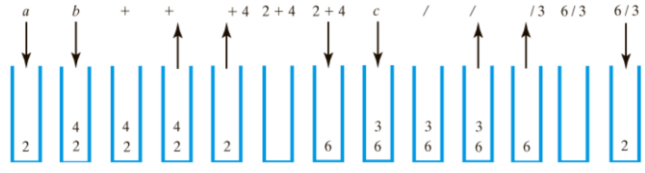


数字和字符 push，操作符 pop

4) Evaluating postfix expression
规则：从左到右进行扫描。扫描到数字，将数字压入堆栈；扫描到运算符，弹出栈顶的两个数（次顶和栈顶元素），先出栈的作为 操作符后数字，后出栈的作为操作符前的数字，并将计算结果入栈
例子：give a postfix expression 34+5*6-
3 4 → 3 + 4 = 7 入栈 → 7 5 → 7 * 5 = 35 入栈 → 35 6 → 35-6 = 29

5) Evaluating prefix expression
规则：从左到右进行扫描。扫描到数字，将数字压入堆栈；扫描到运算符，弹出栈顶的两个数（次顶和栈顶元素），先出栈的作为 操作符后数字，后出栈的作为操作符前的数字，并将计算结果入栈

6) Evaluating infix expression using two stacks
规则：从左到右进行扫描。扫描到数字，将数字压入堆栈；扫描到运算符，按照 transforming infix to postfix 的规则将运算符压入堆栈；如果 pop 出了运算符，那么对于被 pop 出来的操作符，依次 pop 操作数，数字从后向前交叉运算符，将计算结果入栈；如果扫描结束后，运算符非空，继续按照以上规则 pop 运算符计算
NB：就是分成两个堆栈来处理这个问题



数字和字符 push，操作符 pop

7. Sorting
Fast sorting (

1) Selection sort (in-place sorting, $O(n^2)$): 从左到右一词来选择最小的并交换
e.g. 10 8 11 7 4 → 4 8 11 7 10 → 4 7 11 8 10 → 4 7 8 11 10 → 4 7 8 10 11

2) Insertion sort (in-place sorting, $O(n^2)$): 从左到右依次选择最小的并插入排序后的序列
e.g. 10 8 11 7 4 → 4 10 8 11 7 → 4 7 10 8 11 → 4 7 8 10 11

3) Bubble sort (in-place sorting, $O(n^2)$): 从左到右两次一组进行比较并交换

e.g. 10 8 11 7 4 → 8 10 11 7 4 → 8 10 7 11 4 → 8 10 7 4 11 → 8 7 10 4 11 → 8 7 4 10 11 → 8 4 7 10 11 → 4 8 7 10 11 → 4 7 8 10 11 一共进行了三轮

4) Merge sort (not in-place sorting, O(nlogn)): 一直进行一分为 2 的操作然后在过程中排序

5) Quick sort (in-place sorting, average O(nlogn), worst $O(n^2)$): 设定一个基准元素 pivot（通常是第一个或者最后一个元素），将数组分成两部分，比基准元素小的放在左边，比基准元素大的放在右边，相等的可以放在任意一边，然后再一次选择基准元素重复上述过程

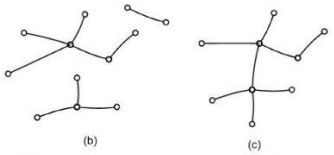
e.g. 8 3 1 5 10 6 2 7 → 选择 8 左边的元素 3 1 5 6 2 7 右边的元素 10 → 选择 3 左边的元素 1 2 右边的元素 5 6 7 → … → 1 2 3 5 6 7 8 10

8. Graph theory

1) Basic definitions: vertices 顶点, edges 边, order (the number of vertices in G is called the order of G), degree (edges 的二倍，loop 算 2), adjacent (two vertices joined by an edge) → neighbors (the adjacent vertices), incident (the edge which joins vertices is said to be incident to them)

2) Types of graph: simple graph (a graph containing no multiple edges or loops) & non-simple graph, weighted graph, subgraph, digraph (also called a directed graph, a graph where instead of edges we have directed edges with an arrow indicating the direction of flow) & undirected graph

3) Trees: is connected graph with no cycles 所有的点都接在一起. Forest: a graph with no cycles and it may not be connected 由多个 tree 组成



(b) A forest (c) A tree

4) Application: minimum/**maximum** spanning tree (MST)可能是最大生成树 & spanning tree

Lab & TTL

```java
public class Car<T> {
    2 usages
    String model;
    1 usage
    int vinNumber;
    2 usages
    T serialCode;
    2 usages
    T weight;
    2 usages
    public Car(String model, int vinNumber, T serialCode, T weight) {
        this.model = model;
        this.vinNumber = vinNumber;
        this.serialCode = serialCode;
        this.weight = weight;
    }
    public String toString() {
        return "Model: " + model + ", Vin Number: " + ", Serial Code: " + serialCode + ", Weight: " + weight;
    }

    //Main method to test the object
    public static void main(String[] args) {
        Car<String> truck = new Car<>( model: "F150", vinNumber: 38038282, serialCode: "A39384224", weight: "3004.5943");
        Car<Integer> sedan = new Car<>( model: "Camry", vinNumber: 293393934, serialCode: 102929383, weight: 8938822);

        System.out.println("Truck: " + truck.toString());
        System.out.println("Sedan: " + sedan);
    }
}
```

```java
public boolean remove(int target){
    int index; // The location of target in the data array.

    // First, set index to the location of target in the data array,
    // which could be as small as 0 or as large as manyItems-1; If target
    // is not in the array, then index will be set equal to manyItems;
    for (index = 0; (index < manyItems) && (target != data[index]); index++);

    if (index == manyItems) // The target was not found, so nothing is removed.
        return false;
    else {  // The target was found at data[index].

        // So reduce manyItems by 1 and copy the last element onto data[index].
        data[index] = data[--manyItems];
        return true;
    }
    /*
    data[0] = 11
    data[1] = 12 <--manyItems
    data[2] = 13
    在remove后会把manyItems的指向data[1]，但是也会保留data[2]的数据
     */
}
```

```java
/**
 * Add new elements to this bag. If the new elements would take this
 * bag beyond its current capacity, then the capacity is increased
 * before adding the new elements.
 **/
4 usages
public void addMany(int... elements){ // int... 是为了添加一个list也就是多个elements
    if (manyItems + elements.length > data.length){
        ensureCapacity( minimumCapacity: (manyItems + elements.length)*2);// Ensure twice as much space as we need.
        // the ensureCapacity方法是用于增加这个bag的容量
    }

    System.arraycopy(elements, srcPos: 0, data, manyItems, elements.length);
    manyItems += elements.length;
}
```

```java
public IntArrayBag clone( ){  // Clone an Tutorial_1.IntArrayBag object.
    IntArrayBag answer;

    try{
        answer = (IntArrayBag) super.clone( );
    }
    catch (CloneNotSupportedException e){
        throw new RuntimeException("This class does not implement Cloneable");
    }

    answer.data = data.clone( );
    return answer;
}
```

```java
/**
 * Add a new element to this bag. If the new element would take this
 * bag beyond its current capacity, then the capacity is increased
 * before adding the new element.
 **/
2 usages
public void add(int element){
    if (manyItems == data.length){
        ensureCapacity( minimumCapacity: (manyItems + 1)*2); // Ensure twice as much space as we need.
    }

    data[manyItems] = element;
    manyItems++;
}
```

```java
public E pop() {
    if (manyItems == 0)
        // EmptyStackException is from java.util and its constructor has no
        // argument.
        throw new EmptyStackException();
    return data[--manyItems];
}
```

```java
public E peek() {
    if (manyItems == 0)
        // EmptyStackException is from java.util and its constructor has no
        // argument.
        throw new EmptyStackException();
    return data[manyItems - 1];
}
```

```java
public void push(E item) {
    if (manyItems == data.length) {
        // Double the capacity and add 1; this works even if manyItems is 0.
        // However, in
        // case that manyItems*2 + 1 is beyond Integer.MAX_VALUE, there will
        // be an
        // arithmetic overflow and the bag will fail.
        ensureCapacity( minimumCapacity: manyItems * 2 + 1);
    }
    data[manyItems] = item;
    manyItems++;
}
```