LEC-1

--Abstraction 抽象

- As a process: 抽象是指提取一个项目或一组项目的基本细节，而忽略非必要的细节。
- As an entity: 抽象指的是一个模型、一个视图或一些实际项目的表示，它忽略了项目的一些细节。
- Abstraction 规定了某些信息比其他信息更重要，但并没有提供一个具体的机制来处理不重要的信息。

在软件开发中，有以下几种不同的抽象类型：

- Data abstraction: The aim of data abstraction is to identify which **details of how data is stored and can be manipulated** are important and which are not
- Procedural abstraction: The aim of procedural abstraction is to identify which **details of how a task is accomplished** are important and which are not

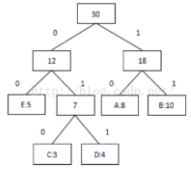
Key of abstraction

Extracting the **commonality** of components and hiding their details. (提取组建的共性并且隐藏细节)

Abstraction typically focuses on the **outside view** of an object or concept. (抽象通常侧重于对象或概念的外部观点)

--Huffman coding

1.首先计算出每个字符出现的频率。 2.然后根据频率排序后将最小的两个相加，作为左右子树。3.二叉树的左边定位 0，右边定为 1。 4.根据路径获得每个符号的代码



字符	次数	编码
B	10	11
A	8	10
C	3	010
D	4	011
E	5	00

```
HUFFMAN  ENCODING (building tree from leaves)
n ← |S|; Q ← S;
for i ← 1 to n-1
{
z ← ALLOCATE-NODE()
right[z] ← EXTRACT-MIN(Q)
x ← right[z]
left[z] ← EXTRACT-MIN(Q)
y ← left[z]
f [z] ← f [x]+ f [y]
INSERT( Q, z)
}
return EXTRACT-MIN(Q)
```

What is input if the following is received? (1) 011011100 (2) 0110111001 (3) 1001110110

(1) ADEAA (2) ADEAA + error (3) BEAD　　注意要一一对应，若有多余就报错

A priority queue supports the following operations: –INSERT(Q, x) inserts the element x into Q. –MIN(Q) returns the element of Q with minimal key. –EXTRACT-MIN(Q) removes and returns the element of Q with minimal key.

-- Information hiding：部分信息对于用户来说是不重要的

-- Encapsulation 封装

- As a process, encapsulation means the act of enclosing one or more items (data/functions) within a (physical or logical) container. 作为一个过程，封装是指将一个或多个项目（数据/功能）封闭在一个（物理或逻辑）容器中的行为。
- As an entity, encapsulation, refers to a package or an enclosure that holds (contains, encloses) one or more items (data/functions). 作为一个实体，封装是指容纳（包含，包围）一个或多个项目（数据/功能）的包或外壳。

在 O-O world 里的封装

In object-oriented programming, encapsulation is the inclusion within an object of all the resources needed for the object to function – i.e., the methods and the data. 在面向对象的编程中，封装是将对象运行所需的所有资源(即方法和数据) 包含在一个对象中。

The object is said to publish its interfaces. Other objects adhere to these interfaces to use the object without having to worry how the object accomplishes it. 这个对象发布一个接口，其他对象遵守这些接口来使用该对象，而不必担心该对象是如何完成的。

An object can be thought of as a self-contained atom. The object interface consists of public methods and instantiated data. 一个对象可以被认为是一个独立的原子。对象的接口由公共方法和实例化的数据组成。

在通信中的封装

In communication, encapsulation is the inclusion of one data structure within another structure so that the first data structure is hidden. For example, a TCP/IP-formatted packet can be encapsulated within an ATM frame. Within the context of sending and receiving the ATM frame, the encapsulated packet is simply a bit stream that describes the transfer. 在通信中，封装是将一个数据结构包含在另一个结构中，从而使第一个数据结构被隐藏。 例如，一个TCP/IP格式的数据包可以被封装在一个ATM帧中。 在发送和接收ATM帧的情况下，封装的数据包只是一个用来描述传输的比特流。

Comparison

1. Abstraction is **a technique** that helps us identify which specific information is important for the user of a module, and which information is unimportant. 确定哪些重要哪些不重要
2. Information hiding is **the principle** that all unimportant information should be hidden from a user. 把不重要的隐藏起来
3. 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. 打包，将应该隐藏的东西隐藏起来，并将打算显示的东西显示出来。

-- Efficiency in space & time

A well-chosen data structure will include operations that are efficient in terms of speed of execution (based on some wellchosen algorithm). For our purposes the most important measure for the speed of execution will be the number of accesses to data items stored in the data structure. 对于我们的目的来说，衡量执行速度的最重要标准是对存储在数据结构中的数据项的访问次数。

对于空间来说，好的数据结构应当尽可能减少内存使用的空间

-- Static data structures：fixed at the time of creation

好处

- **Ease of specification.** Programming languages usually provide an easy way to create static data structures of almost arbitrary size.
- **No memory allocation overhead**. Since static data structures are fixed in size, 1) There are no operations that can be used to extend static structures; 2) Such operations would need to allocate additional memory for the structure (which takes time). 没有内存分配的开销。 由于静态数据结构的大小是固定的，1）没有可以用来扩展静态结构的操作；2）这种操作需要为结构分配额外的内存（这需要时间）。

坏处

- **Must make sure there is enough capacity.** Since the number of data items we can store in a static data structure is fixed, once it is created, we have to make sure that this number is large enough for all our needs. 必须确保有足够的容量。 由于我们可以在静态数据结构中存储的数据项的数量是固定的，一旦它被创建，我们必须确保这个数字足以满足我们所有的需求。
- **More elements? (errors), fewer elements? (waste)** 1) However, when our program tries to store more data items in a static data structure than it allows, this will result in an error (e.g. ArrayIndexOutOfBoundsException) 2) On the other hand, if fewer data items are stored, then parts of the static data structure remain empty, but the memory has been allocated and cannot be used to store other data. 1）当我们的程序试图在静态数据结构中存储超过它所允许的数据项时，这将导致一个错误（例如ArrayIndexOutOfBoundsException）2）另一方面，如果存储较少的数据项，那么静态数据结构的部分仍然是空的，但内存已被分配，不能用来存储其他数据。

-- Dynamic data structures：grow or shrink during run-time

好处

- There is no requirement to know the exact number of data items since dynamic data structures can shrink and grow to fit exactly the right number of data items, there is no need to know how many data items we will need to store. 没有要求知道数据项的确切数量，因为动态数据结构可以收缩和增长，以适应确切的数据项数量，所以没有必要知道我们需要存储多少数据项。
- **Efficient use of memory space.** Extend a dynamic data structure in size whenever we need to add data items which could otherwise not be stored in the structure and shrink a dynamic data structure whenever there is unused space in it, then the structure will always have exactly the right size and no memory space is wasted. 有效地利用内存空间。每当我们需要增加数据项时，就扩大动态数据结构的大小，否则就不能存储在结构中，每当动态数据结构中出现未使用的空间时，就缩小动态数据结构的大小，那么该结构将永远有准确的大小，没有内存空间被浪费。

-- Dynamic data structures：grow or shrink during run-time

好处

- There is no requirement to know the exact number of data items since dynamic data structures can shrink and grow to fit exactly the right number of data items, there is no need to know how many data items we will need to store. 没有要求知道数据项的确切数量，因为动态数据结构可以收缩和增长，以适应确切的数据项数量，所以没有必要知道我们需要存储多少数据项。
- **Efficient use of memory space.** Extend a dynamic data structure in size whenever we need to add data items which could otherwise not be stored in the structure and shrink a dynamic data structure whenever there is unused space in it, then the structure will always have exactly the right size and no memory space is wasted. 有效地利用内存空间。每当我们需要增加数据项时，就扩大动态数据结构的大小，否则就不能存储在结构中，每当动态数据结构中出现未使用的空间时，就缩小动态数据结构的大小，那么该结构将永远有准确的大小，没有内存空间被浪费。

坏处

- Memory allocation/de-allocation overhead. 内存分配/释放的开销
- 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). 每当动态数据结构增长或收缩时，就必须增加或删除分配给该数据结构的内存空间(这需要时间)。

LEC-2

-- Library

Programming with **Linear** collections 线性集.

- Kinds of collections:
  - Lists, Sets, Bags, Maps, Stacks, Queues, Priority Queues

Java Library 使用别人用过的代码.
也叫 API，Application Programming Interface. 应用程序编程接口.

Libraries to use

• java.util	**Collection** classes Other **utility** classes 实用类.
• java.io	Classes for **input and output**
• javax.swing java.awt	Large library of classes for **GUI** programs 图形用户界面.

import the package or class into your program

```
import java.util.*;
import java.io.*;
```

Read the documentation to identify how to use

- **Constructors** for making instances
- **Methods** to call
- **Interfaces** to implement

Use the classes as if they were part of your program

Java Collections library

Interfaces:

- *Collection*
  = Bag (most general)
- *List*
  = ordered collection
- *Set*
  = unordered, no duplicates
- *Queue*
  ordered collection, limited access (add at one end, remove from other)
- *Map*
  = key-value pairs (or mapping)

Classes

- List classes:
  ArrayList, LinkedList, vector
- Set classes:
  HashSet, TreeSet,…
- Map classes:
  HashMap, TreeMap, …
- …

-- ADT

Set, Bag, Queue, List, Stack, Map, etc are

Abstract Data Types (outcome of abstraction / encapsulation)

- an ADT is a type of data, described at an abstract level:
  - Specifies the **operations** that can be done to an object of this type
  - Specifies how it will **behave**.

-- Interface

- A Java ***Interface*** corresponds to an Abstract Data Type 一对应.
  - Specifies what methods can be called on objects of this type (specifies name, parameters and types, and type of return value)
  - Behaviour of methods is only given in comments (but cannot be enforced)

× No constructors - can't make an instance: **new** Set()
× No fields - doesn't say how to store the data
× No method bodies. - doesn't say how to perform the operations

Parameterised Types 参数化种类.

- Interfaces may have type parameters (*eg*, type of the element):

```
public interface Set <T> {
    public void add(T item);          /*…description…*/
    public void remove(T item);       /*…description…*/
    public boolean contains(T item); /*…description…*/
    …  // (lots more methods in the Java Set interface)
```

It's a Set of something, as yet unspecified

- When declaring variable, specify the actual type of element

```
private Set <Person>  friends;
private List <Shape> drawing;
```

Collection Type 集合种类.

Type of value in Collection 值的种类.

```
private List <Shape> drawing = new ArrayList <Shape> ( );
Set <Person> friends = new HashSet <Person> ( );
```

-- ArrayList：保存一列 item，并有一个特定的顺序

need to **import java.util.*;** at head of file

- Don't have to specify its size
- Should specify the type of items.
  - new syntax: "type parameters"
- Like an infinitely stretchable array
- But, you can't use the […] notation
- you have to call methods to access and assign

Array vs. ArrayList

- Array:

```
private static final int maxStudents = 1000;
private Student[ ] students = new Student[maxStudents];
private int count = 0;
```

- Alternatively, we can do the following…
- ArrayList:

```
private ArrayList <Student> students = new ArrayList <Student>();
```

值的种类.

- The type of values in the list is between "<" and ">" after ArrayList.
- No maximum; no initial size; no explicit count

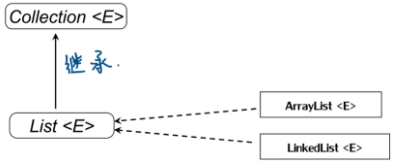
ArrayList has many methods! , including:

- size(): returns the number of items in the list
- add(*item*): adds an item to the *end* of the list
- add(*index*, *item*): inserts an item at *index* (relocates later items) 后面的项目重新定位.
- set(*index*, *item*): replaces the item at *index* with *item*
- contains(*item*): true if the list contains an item that equals *item*
- get(*index*): returns the item at position *index*
- remove(*item*): removes an occurrence of item
  (what if there are duplicates in the ArrayList?) → 多次出现则删掉第一次出现的.
- remove(*index*): removes the item at position *index* (both relocate later items)

- You can use the "for each" loop on an array list, as well as a **for** loop

LEC-3

-- Collection and List



Interfaces can **extend** other interfaces:
The **sub** interface has all the methods of the **super** interface plus its own methods (**sub** means? **super** means?)
下一级 上一级

Methods on Collection and List

- Collection <E>
  - isEmpty() → boolean
  - size() → int
  - contains(E elem) → boolean
  - add(E elem) → boolean (whether it succeeded)
  - remove(E elem) → boolean (whether it removed an item)
  - iterator() → iterator <E>
  - …

Methods on all types of collections

- List <E>
  - add(int index, E elem)
  - remove(int index) → E (returns the item removed)
  - get(int index) → E
  - set(int index, E elem) → E (returns the item replaced)
  - indexOf(E elem) → int
  - subList(int from, int to) → List<E> 注意 index 要减1.

Additional methods on all Lists

Variable or field declared to be of the interface type
- Specify the type of the collection
- Specify the type of the value

```
private List <Task>  tasks;
```

- The type between "<" and ">" is the type of the elements

Create an object of a class that implements the type:
- Specify the class
- Specify the type of the value

```
tasks = new ArrayList <Task> ();
```

List vs. Array

jobList.set(ind, value)	jobArray[ind] = value
jobList.get(ind)	jobArray[ind]
jobList.size()	? *(Not the length!!!)*
jobList.add(value)	? *(Where is the last value? What happens if it's full?)*
jobList.add(ind, value)	? *(Have to shift everything up!!!)*
jobList.remove(ind)	? *(Have to shift everything down!!!)*
jobList.remove(value)	? *(Have to find value, then shift things down!!!)*

- **for** (Task t : tasks) *vs* **for**(int i = 0; i< ???; i++){ Task t = taskArray[i];

-- Iterator

```
for (Task task : tasks){
    textArea.append(task + "\n");
```

⇕ 等价.

```
Iterator <Task> iter = tasks.iterator();
while (iter.hasNext()){
    Task  task = iter.next();
    textArea.append(task +  "\n");
```

Iterator is an interface:

```
public interface Iterator <E> {
    public boolean hasNext();
    public E next();
}
```

A Scanner is a fancy iterator

- Operations on Iterators:
  - **hasNext**() : *returns* true *iff there is another value to get*
  - **next**() : *returns the next value*

- Standard pattern of use:

```
Iterator<type > itr  = construct iterator
while (itr.hasNext() ){
    type var = itr.next();
    … var …
}
```

-- Iterator and Iterable

Iterable : 可迭代对象.(序列大小确定)
Iterator : 迭代器的对象.(不知道多少个元素)

```
for (type var : Iterable <type> ){
   … var …
}
```

eg, all Collections

Iterable <T>	Iterator <T>
public Iterator<T> iterator();	public boolean hasNext(); public T next();

```
Iterator<type > itr  = construct iterator
while (itr.hasNext() ){
   type var = itr.next();
   … var …
}
```

例子：

Iterator

```
public class NumCreator implements Iterator<Integer>{
    private int num  = 1,
    public boolean hasNext(){
       return true;
    }
    public Integer next(){
       num = (num * 92863) % 104729 + 1;
       return num;
    }
 :
Iterator<Integer> lottery = new NumCreator();
for (int i = 1; i<1000; i++)
    textArea.append(lottery.next()+ "\n");
```

Iterable

```
public class NumberSequence implements Iterable<Integer>{
    private int start;
    private int step;
    public NumberSequence(int start, int step){
       this.start = start;
       this.step = step;
    }
    public Iterator<Integer> iterator(){
       return new NumberSequenceIterator(this);
    }
```

LEC-4

-- Bags

- A Bag is a collection with
  - no structure or order maintained 无顺序
  - no access constraints (access any item any time)
  - duplicates allowed 可重复.
- Minimal Operations:
  - add(value) → returns true *iff* a collection was changed
  - remove(value) → returns true *iff* a collection was changed
  - contains(value) → returns true *iff* value is in bag uses equal to test.
  - findElement(value) → returns a matching item, *iff* in bag
- Plus
  - size(), isEmpty(), iterator(),clear(), addAll(collection), removeAll(collection), containsAll(collection), …

-- Set

- Set is a collection with:
  - no structure or order maintained 无序
  - no access constraints (access any item any time)
  - Only property is that duplicates are excluded 不可重复.
- Operations:

  (Same as Bag, but different behaviour)
  - add(value) → true *iff* value *was added (ie, no duplicate)*
  - remove(value) → *true iff value removed (was in set)*
  - contains(value) → *true iff value is in the set*
  - findElement(value) → *matching item, iff value is in the set*
  - …
- Sets are as common as Lists

-- Stacks

- Stacks are a special kind of List:
  - Sequence of values, ('sequence' means?) 只能从一边删除加入.
  - Constrained access: add, get, and remove only from one end.
  - There exists a Stack interface and different implementations of it (ArrayStack, LinkedStack, etc)
  - In Java Collections library:
    - Stack is a class that implements List
    - Has extra operations: push(value), pop(), peek()
- push(value): Put value on top of stack
- pop(): Removes and returns top of stack
- peek(): Returns top of stack, *without* removing
- plus the other List operations

Applications of Stacks

1. Processing files of structured (nested) data. E.g., reading files with structured markup (HTML, XML) 2. Program execution, e.g., working on subtasks, then returning to previous task. 3. Undo in editors. 4. Expression evaluation : (6 + 4) * ((12.1 *sin (15)) – (cos (20) / 38))

- HTML example → 超文本标记语言. 人与浏览器之间的交流.

```
<html>
<body>
The content of the body element is displayed in your browser.
</body>
</html>
```

- XML examples →

```
<Person>
        <name>Henry Ford</name>
</Person>

<Book>
        <title>My Life and Work</title>
        <author>Henry Ford</author>
</Book>
```

-- Infix to Postfix

• Infix 中缀：a + b • Prefix 前缀：+ a b • Postfix 后缀：a b +

具体转换方式:
1.从左到右进行遍历
2.运算数,直接输出.
3.左括号,直接压入堆栈,(括号是最高优先级,无需比较)(**入栈后优先级降到最低**,确保其他符号正常入栈)
4.右括号,(意味着括号已结束)不断弹出栈顶运算符并输出直到遇到左括号(弹出但不输出)
5.运算符,将该运算符与栈顶运算符**进行比较**,
如果**优先级高于栈顶运算符**则压入堆栈(该部分运算还不能进行),
如果**优先级低于等于栈顶运算符**则将栈顶运算符弹出并输出,然后比较新的栈顶运算符.
(低于弹出意味着前面部分可以运算,先输出的一定是高优先级运算符,等于弹出是因为同等优先级,从左到右运算)
直到优先级大于栈顶运算符或者栈空,再将该运算符入栈.
6.如果对象处理完毕,则按顺序弹出并输出栈中所有运算符.

步骤	待处理表达式	栈顶状态(低-->高)	输出状态
1	a*(b+c)/d		
2	*(b+c)/d		a
3	(b+c)/d	*	a
4	b+c)/d	* (	a
5	+c)/d	*(	a b
6	c)/d	*(+	a b
7	)/d	*(+	a b c
8	/d	*	a b c +
9	d	/	a b c + *
10		/	a b c + * d
11			a b c + * d /

- Methods in class `Stack` in `java.util`

```
public void push(Object item);
public Object pop();
public Object peek();
public boolean isEmpty();
public void clear();
```

```
public void reverseNums(Scanner sc){
  Stack<Integer> myNums = new ArrayStack<Integer>();
  while (sc.hasNext())
     myNums.push(sc.Next())
  while (! myNums.isEmpty())
     textArea.append(myNums.pop() + "\n");
}
```

例子

-- Maps

- Collection of data, but not of single values:
  - Map = **Set** of pairs of keys to values 键值对.
  - Constrained access: get values via keys.
  - **No duplicate keys** 无重复键.
  - Lots of implementations, most common is **HashMap**.
- When declaring and constructing, must specify two types:
  - Type of the key, and type of the value

```
private Map<String, Integer> phoneBook;
  :
phoneBook = new HashMap<String, Integer>();
```

- Central operations:
  - get(key), → returns value associated with key (or null)
  - put(key, value), → sets the value associated with key (and returns the old value, if any)
  - remove(key), → removes the key *and* associated value (and returns the old value, if any)
  - containsKey(key), → boolean
  - size()

```
/** Construct histogram of counts of all words in a file */
public Map<String, Integer> countWords(Scanner scan){
   Map<String, Integer>  counts = new HashMap<String, Integer> ();
   for (String word : scan){
     if ( counts.containsKey(word) )
        counts.put(word, counts.get(word)+1);
     else
        counts.put(word, 1);
   return counts;
}
```

一个找最高频率的例子

```
/** Find word in histogram with highest count */
public String findMaxCount(Map<String, Integer> counts){
   // for each word in map
   //   if has higher count than current max, record it
   // return current max word
}
```

- keySet() → Set of all keys 键组合.
  - for (String name : phonebook.keySet()){….
- values() → Collection of all values 值组合.
  - for (Integer num : phonebook.values()){….
- entrySet() → Set of all Map.Entry's 键值组合.
  - for (Map.Entry<String, Integer> entry : phonebook.entrySet()){….
    … entry.getKey() …
    … entry.getValue()…

LEC-5

-- Queues

- Collection of values with an order 有顺序
- Constrained access:
  - Only remove from the front 从前移除
- Two varieties:
  - **Ordinary queues**: only add at the back
- **Priority queues**: add or remove with a given priority

- Used for
  - Operating Systems, Network Applications, Multi-user Systems
    - Handling requests/events/jobs that must be done in order 处理有序任务
    - (memory pool holding such requests are often called a "buffer" in this context)
  - Simulation programs 模拟程序.
    - Representing queues in the real world (traffic, customers, deliveries, ….)
    - Managing events that must happen in the future
  - Search Algorithms
    - Computer Games
    - Artificial Intelligence

- Java provides
  - a Queue interface
  - several classes: **LinkedList, PriorityQueue**

指令.

- **offer**(value) ⇒ boolean
  - add a value to the queue
  - (sometimes called "enqueue")
- **poll**() ⇒ *value*
  - remove and return value at front/head of queue or null if the queue is empty
  - (sometimes called "dequeue", like "pop")
- **peek**() ⇒ *value*
  - return value at head of queue, or null if queue is empty (doesn't remove from queue)
- **remove**() and **element**()
  - like poll() and peek(), but throw exception if queue is empty.

LEC-6

-- Comparator & comparable

- If a class implements the Comparable<*T* > interface
  - Objects from that class have a "natural ordering"
  - Objects can be compared **using** the compareTo() method
  - Collections.sort() can sort Lists of those objects automatically

- Comparable <*T* > is an Interface:
  - Requires
    - compareTo(*T* ob) → int
- ob1.compareTo(ob2)
  - returns –ve if ob1 ordered before ob2
  - returns 0 if ob1 ordered with ob2
  - returns +ve if ob1 ordered after ob2

Comparable vs. Comparator

↓

只有 compareTo() 一个方法.

进行的是自然顺序排序

↘ 有多个.

在外部制定排序规则

- Classes should implement the **Comparable interface** to control their ***natural ordering.***

- Objects that implement Comparable can be sorted by **Collections.sort()** and **Arrays.sort()** and can be used as keys in a sorted map or elements in a sorted set without the need to specify a *Comparator*.

« Interface »
Comparable
+ compareTo(Object) : int

- **compareTo()** compares this object with another object and returns a *negative* integer, *zero*, or a *positive* integer as this object is *less* than, *equal* to, or *greater* than the other object.

- Use **Comparator** to sort objects in an order other than their natural ordering.

« Interface »
Comparator
+ compare(Object, Object) : int

- **compare()** compares its two arguments for order, and returns a *negative* integer, *zero*, or a *positive* integer as the first argument is *less* than, *equal* to, or *greater* than the second.
- Collections.sort(…) has two forms:
  - Sort by the natural order

    Collections.sort(todoList)
    - the values in todoList must be Comparable
  - Sort according to a specified order:

    Collections.sort(crowd, faceByArea)
    - faceByArea is a Comparator object for comparing the values in crowd.

Comparable	⟺	Comparator
interface for objects that can be compared		An object that can compare other objects

- Comparator <T> is an Interface
- Requires
  - **public** int compare(T o1, T o2);
    - → –ve if o1 ordered before o2
    - → 0 if o1 equals o2 [must be compatible with equals()!]
    - → +ve if o1 ordered after o2

```
/** Compares faces by the position of their top edge */
private class TopToBotComparator implements Comparator<Face>{
   public int compare(Face f1, Face f2){
      return (f1.getTop() - f2.getTop());
   }
}
```

多种比较器

```
String button = event.getActionCommand();
if (button.equals("SmallToBig")){
    Collections.sort(crowd);   // use the "natural ordering" on Faces.
    render();
}
else if (button.equals("BigToSmall")){
    Collections.sort(crowd, new BigToSmallComparator());
    render();
}
else if (button.equals("LeftToRight")){
    Collections.sort(crowd, new LftToRtComparator());
    render();
}
else if (button.equals("TopToBottom")){
    Collections.sort(crowd, new TopToBotComparator());
    render();
}
```

LEC-7

-- Exceptions

当方法出错时抛出异常. 并抓住/解决.

通过 try-catch 来抓住.

- exceptions thrown in a **try ... catch** can be "caught":

```
try {
    … code that might throw an exception
}
catch (⟨ExceptionType1⟩ e1) { …actions to do in this case….}
catch (⟨ExceptionType2⟩ e2) { …actions to do in this case….}
catch (⟨ExceptionType3⟩ e3) {   System.out.println(e.getMessage());
                                e.printStackTrace();
```

Exception object, containing information about the state

E.g., IOException, NoSuchMethodException, RuntimeException, IndexOutOfBoundsException…

- **RuntimeExceptions don't have to be handled**:
  - An uncaught RuntimeException will result in an error message
  - You can catch them if you want.

- **Other Exceptions must be handled**:
  - eg IOException

    (which is why we always used a ***try…catch*** when opening files).

LEC-8

-- Generics

泛型是一种参数化类型的机制，可以用来定义类、接口和方法。泛型的好处是可以在编译时检查类型安全，避免了类型转换的麻烦。泛型的实现是通过类型擦除来实现的，即在编译时擦除类型信息，只保留原始类型。泛型的实现是通过类型擦除来实现的，即在编译时擦除类型信息，只保留原始类型。

public interface List <E> **extends** Collection <E> {

E will be bound to a real type when a List is declared or constructed.

-- ArrayList

- Design the data structures to store the values
  - array of items
  - count

- Define the fields and constructors

```
public class ArrayList <E> implements List<E> {
    private E [ ] data;
    private int count;
```

- Define all the methods specified in the List interface

size()	add(E o)	add(int index, E element)	contains(Object o)	get(int index)
isEmpty()	clear()	set(int index, E element)	indexOf(Object o)	remove(int index)
remove(Object o)		lastIndexOf(Object o)	iterator()	
equals(Object o)		hashCode()	listIterator()	.....

由于需要用到List中许多方法. 所以对基础的方法进行抽象.

• Interface
  • specifies type
  • defines method headers
	• List <E>
	  • Specifies sequence of E *type*

• Abstract Class
  • implements Interface
  • defines some methods
  • leaves other methods "abstract"
	• AbstractList <E>
	  • implements List <E>
	  • defines array of <E>
	  • defines addAll, subList, …
	  • add, set, get, … are leftabstract

• Class
  • extends Abstract Class
  • defines data structures
  • defines basic methods
  • defines constructors
	• ArrayList <E>
	  • extends AbstractList
	  • implements fields & constructor
	  • implements add, get, …

```
public abstract class AbstractList <E> implements List<E>{
```
No constructor or fields

```
  public abstract int size();
```
declared abstract - must be defined in a real class

```
  public boolean isEmpty(){
    return (size() == 0);
  }
```
defined in terms of size()

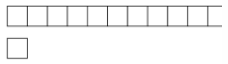
```
  public abstract E get(int index);
```
declared abstract - must be defined in a real class

```
  public void add(int index, E element){
    throws new UnsupportedOperationException();
  }
```
defined to throw exception should be defined in a real class

```
  public boolean add(E element){
    add(size(), element);
```
defined in terms of other add

例子

- Data structure:



- size:
  - returns the value of count
- get and set:
  - check if within bounds, and
  - access the appropriate value in the array
- add(index, elem):
  - check if within bounds, (0..size)
  - move other items up, and insert
  - as long as there is room in the array !

```
/**Ensure data array has sufficient number of elements
  * to add a new element */
private void ensureCapacity () {
  if (count < data.length) return;          ← room already
  E [ ] newArray = (E[]) (new Object[data.length+INITIALCAPACITY]);
    for (int i = 0; i < count; i++)         ← copy to new array
      newArray[i] = data[i];
    data = newArray;                        ← replace (replace what?)
}
```

OR

```
private void ensureCapacity () {
    if (count < data.length) return;        ← room already
    E [ ] newArray = (E[ ]) (new Object[data.length * 2]);
    for (int i = 0; i < count; i++)         ← copy to new array
      newArray[i] = data[i];
    data = newArray;                        ← replace
}
```

```
/** Returns number of elements in collection as integer */
  public int size() {
    return count;
  }
```

```
/** Returns true if this set contains no elements. */
   public boolean isEmpty(){
      return count==0;
   }

/**Returns the value at the specified index.
 * Throws an IndexOutOfBoundsException if index is out of
 bounds */
  public E get(int index){
    if (index < 0 || index >= count)
    throw new IndexOutOfBoundsException();
    return data[index];
  }
/**Replaces the value at the specified index by the specified value
 * Returns the old value.
 * Throws an IndexOutOfBoundsException if index is out of
 bounds */
public E set(int index, E value){
   if (index < 0 || index >= count)
      throw new IndexOutOfBoundsException();
   E ans = data[index];
   data[index] = value;
   return ans;
 }

  /** Removes the element at the specified index, and returns it.
   * Throws an IndexOutOfBoundsException if index is out of
   bounds */ public E remove (int index){
     if (index < 0 || index >= count)
       throw new IndexOutOfBoundsException();
     E ans = data[index];                   ←remember
     for (int i=index+1; i< count; i++)     ←move items down
       data[i-1]=data[i];
     count--;                               ←decrement
     data[count] = null;                    ←delete previous last element
     return ans;                            ←return
   }

  /** Adds the specified element at the specified index.*/
  public void add(int index, E item){
     if (index < 0 || index > count)        ←can add at end?
       throw new IndexOutOfBoundsException();
     ensureCapacity();                      ←make room
     for (int i=count; i > index; i--)      ←move items up
       data[i]=data[i-1];
     data[index]=item;                      ←insert
     count++;                               ←increment
  }
```

确保有足够的空间 增加新元素.

-- ArrayList: iterator

```
/** Definition of the iterator for an ArrayList
  * Defined inside the ArrayList class, and can therefore access
    * the private fields of an ArrayList object. */
  private class ArrayListIterator <E> implements Iterator <E>{
    // fields to store state
    // constructor
    // hasNext(),
    // next(),
    // remove()    (an optional operation for Iterators)
  }
```

Eg.

```
  private ArrayList<E> list;// reference to the list it is iterating down
  private int nextIndex = 0; // the index of the next value to return
  private boolean canRemove = false;
            // to disallow the remove operation initially

 /** Constructor */
 private ArrayListIterator (ArrayList <E> list) {
   this.list = list;
 }
```

```
 /** Return true if iterator has at least one more element */
 public boolean hasNext() {
   return (nextIndex < list.count);
 }

/** Return next element in the List */
public E next() {
   if (nextIndex >= list.count)  throw new NoSuchElementException();
        return list.get(nextIndex++); ← increment and return
      }


/** Remove from the list the last element returned by the iterator.
  * Can only be called once per call to next. */
public void remove(){
    throw new UnsupportedOperationException();
  }

 /** Return next element in the List */
  public E next () {
     if (nextIndex >= list.count)  throw new
  NoSuchElementException();
     canRemove = true;              ← for the remove method
     return list.get(nextIndex++);  ← increment and return
    }

 /** Remove from the list the last element returned by the iterator.
  Can only be called once per call to next. */
  public void remove(){
     if ( ! canRemove )  throw new IllegalStateException();
     canRemove = false;             ← can only remove once
     nextIndex--;           ← put counter back to last item
     list.remove(nextIndex);        ← remove last item
   }
```

1. Each iterator keeps track of its own position in the List 2. Removing the last item returned is possible, but the implementation is not smart, and may be corrupted if any changes are made to the ArrayList that it is iterating down. 3. Note that because it is an **inner** class, it has access to the **ArrayList's private** fields.

LEC-10

-- Analyzing Costs

时间: 程序运行的时间. 空间: 程序占用的内存.
一个算法所花费的步骤. 一个算法储存的基本数据项.

Benchmarking: program cost 基准化分析

- Measure:
  - actual programs
  - on real machines
  - on specific input
  - measure elapsed time
    - **System.currentTimeMillis**()
      → time from system clock in milliseconds (long)
  - measure real memory usage
- Problems:
  - what input ⇒ choose test sets carefully, use large data sets, don't include user input
  - other users/processes ⇒ minimise, average over many runs
  - which computer? ⇒ specify details
- Measure number of "steps" as a function of the data size.
  - worst case (easier)
  - average case (harder)
  - best case (easy, but useless)

三种情况.

- Construct an expression for the number of steps:
  - cost = 3.47 n^2 - 67n + 53 steps
  - cost = 3n log(n) - 5n + 6 steps

  simplified into terms of different powers/functions of n

只考虑具有决定价值的 n，忽略常数项，使用 Big O

- assuming a 100-MHz clock, N = 1024k = 2^{20}
- O(1) - constant time, 10 ns
- O(log N) - logarithmic time, 200 ns
- O(N) - linear time, 10.5ms
- O(N log N) - n log n time, 210 ms
- O(N ^ 2) - quadratic time, 3.05 hours
- O(N ^ 3) - cubic time, 365 years
- O(2 ^ N) - exponential, 10 ^ (10 ^ 5) years

$O(\log n) < O(\sqrt{n}) < O(n) < O(n \log n) < O(n^2) < O(2^n) < O(n!)$

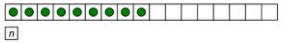
```
public E remove (int index){
    if (index < 0 || index >= count) throw new ....Exception();
    E ans = data[index];
    for (int i=index+1; i< count; i++)      ← in the innermost loop
      (data[i-1]=data[i]; )                 ←Key Step
     count--;
    data[count] = null;
    return ans;
  }
```

Each for loop:
1 comparison: i<count → 表.
1 addition: i++ → 便宜
1 data retrieval: data[i]
1 subtraction: i-1
1 memory store: data[i-1]=data[i]

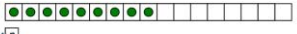
-- Cost in ArrayList

- Assume List contains *n* items.
- Cost of get and set:
  - best, worst, average: O(1) 一次就获得.
  - ⇒ constant number of steps, regardless of *n*

- Cost of Remove:



  - worst case:
    - what is the worst case? O(n) 数在最后例. 剩余的都要左移
    - how many steps?
  - average case:
    - what is the average case? O(n-i) i是数的index
    - how many steps?
- Cost of add(index, value):
  - what's the key step?
  - worst case: O(n) 如在最末端.



  - average case:

- Cost of add(value): add at the end.
  - what's the key step?
  - worst case:

如果不需要resize，直接 O(1)，最末端
如果需要，则为 O(n)，将原数组依次放入新数组.
  - average case:

Arrayset. —与 ArrayList 同，没必要有序

contains(item) : O(n).
remove (item) : O(1). 移除最后一个. O(n) → avg
add : O(1) / O(n).

LEC-11

-- Recursion

- base case:
  if (number <= 1) return 1;

- recursive step:
  return (number * fac (number - 1));

Fibonacci: base case: fibonacci (0) = 0 fibonacci (1) = 1

recursive step: fibonacci (n) = fibonacci (n-1) + fibonacci (n-2)

LEC-12

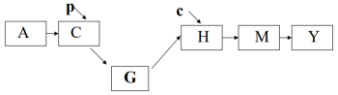
-- 测试接口的实现

1. 写一个测试方法（考虑优劣）2. black box testing& white box testing 3. 报错

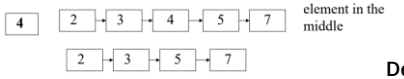
-- LinkedList：在每一个节点里存到下一个节点的地址，继承于 AbstractList

一个单向链表包含两个值：当前节点的值和一个指向下一个节点的链接。

此时，通过链接，可以实现删除和增加，并查询链表里的任意值；元素可重复



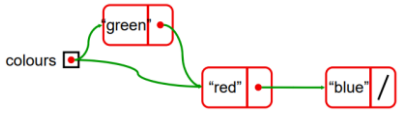
Insert



Delete

```
public class LinkedNode <E>{
    private E value;
    private LinkedNode<E> next;
    public LinkedNode(E item, LinkedNode<E> nextNode){
        value = item;
        next = nextNode;
    }
```

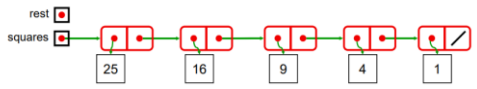
```
LinkedNode<String> colours = new LinkedNode<String> ("red",null);
colours.setNext(new LinkedNode<String>("blue", null));
colours = new LinkedNode<String>("green", colours);
```



使用 setnext 可以将指定 node 进行移除

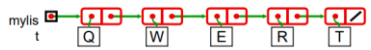
打印一个 linked List:

```
public void printList(LinkedNode<E> list){
    for (LinkedNode<Integer> rest=list; rest!=null;  rest=rest.next )
        System.out.format("%d, ", rest.value);
}
```



Insert

```
/** Insert the value at position n in the list (counting from 0)
    Assumes list is not empty,  n>0, and n <= length of list */
public void insert (E item, int n, LinkedNode<E>list){
    int pos =0;
    LinkedNode<E> rest=list;   // rest is the pos'th node
    while (pos <n-1){
        pos++;
        rest=rest.next;
    }
    rest.next = new LinkedNode<E>(item, rest.next);
}
```



Remove

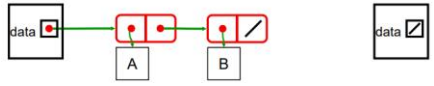
```
/** Remove the value from the list
    Assumes list is not empty, and value not in first node */
public void remove (E item, LinkedNode<E>list){
    LinkedNode<E> rest=list;
    while (rest.next != null  && !rest.next.value.equals(item))
        rest=rest.next;
    if (rest.next != null)
        rest.next = rest.next.next;
}
```

Why have a 'rest' to hold 'list'?

LEC- 13-14

- `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

```
public E get(int index){
    if (index < 0) throw new IndexOutOfBoundsException();
    Node<E> node=data;
    int i = 0;  // position of node
    while (node!=null && i++ < index)  node=node.next;
    if (node==null)   throw new IndexOutOfBoundsException();
    return node.value;
}

public void add(int index, E item){
    if (item == null) throw new IllegalArgumentException();
    if (index==0){              // add at the front.
        data = new Node(item, data);
        count++;
        return;
    }
    Node<E> node=data;
    int i = 1;                  // position of next node
    while (node!=null && i++ < index) node=node.next;
    if (node == null) throw new IndexOutOfBoundsException();
    node.next = new Node(item, node.next);
    count++;
    return;
}
```



Cost of linked list

`get / set: O(n); insert: O(1); remove: O(1)`

Cost of linked Set (items in sorted order)

`contains: O(logn); insert: O(n); remove: O(logn)`

LEC-15

-- Linked Stack (FIFO)

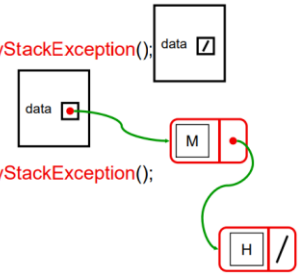
元素从队列的尾部（rear）入队，从队列的头部（front）出队。

```
public class LinkedStack <E> extends AbstractCollection <E> {
    private Node<E> data = null;
    public LinkedStack(){ }
    public int size(){…
    public boolean isEmpty(){…
    public E peek(){…
    public E pop(){…
    public void push(E item){…
    public Iterator <E> iterator(){
```

delete

insert

```
public E peek(){
  if (data==null) throw new EmptyStackException();
  return data.value;
}
public E pop(){
  if (data==null) throw new EmptyStackException();
  E ans = data.value;
  data = data.next;
  return ans;
}
public void push(E item){
  if (item == null) throw new IllegalArgumentException();
  data = new Node(item, data);
}
public Iterator <E> iterator(){
    return new NodeIterator(data);
}
```



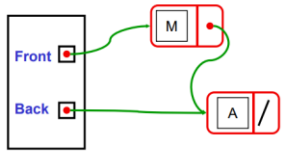
Iterator

```
private class NodeIterator <E> implements Iterator <E>{
    private Node<E> node;    // node containing next item
    public NodeIterator (Node <E> node) {
        this.node = node;
    }
    public boolean hasNext () {
        return (node != null);
    }
    public E next () {
        if (node==null) throw new NoSuchElementException();
        E ans = node.get();
        node = node.next();
        return ans;
    }
    public void remove(){
        throw new UnsupportedOperationException();
    }
}
```

-- LinkedQueue：由链表存储的队列结构(LIFO)

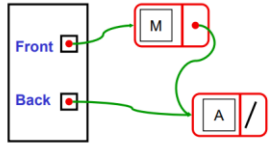
元素从栈顶（top）入栈，也从栈顶出栈。

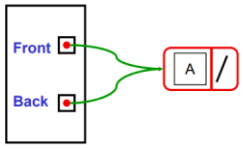
```
public class LinkedQueue <E> implements AbstractQueue <E> {
  private Node<E> front = null;
  private Node<E> back = null;
  public LinkedQueue(){ }
  public int size(){…
  public boolean isEmpty(){…
  public E peek(){…
  public E poll(){…
  public void offer(E item){…
  public Iterator <E> iterator(){
```



`Offer`:将元素添加到队尾

```
public boolean offer(E item){
  if (item == null) return false;
  if (front == null){
      back = new Node(item, null);
      front = back;
  }
  else {
      back.next = (new Node(item, null));
      back= back.next;
  }
  return true;
}
```





Queue & Stack Sum:

在链表头进行添加或删除操作非常容易，但链表尾进行添加或删除操作较为困难;若有指向尾节点的指针，很容易在链表的尾部进行添加操作。

所有主要操作的时间复杂度为 O(1)；

LEC-16

-- Binary Search：分而治之（Divide and Conquer）

Substitution method

$$T(n) = \begin{cases} 1 & \text{if } n=1 \\ T(n/2)+1 & \text{otherwise} \end{cases}$$

Make a guess, T(n) ≤ 2 log n

We prove statement by MI.

Assume true for all n' < n [assume T(n/2) ≤ 2 log (n/2)]

T(n) = T(n/2) + 1

≤ 2 log (n/2) + 1 ← *by hypothesis*

= 2(log n - 1) + 1 ← ***log(n/2) = log n - log 2***

< 2log n

Cost: **i.e., T(n) ≤ 2 log n**

```
private boolean contains(Object item){
    Comparable<E> value = (Comparable<E>) item;
    int low = 0;                        // min possible index of item
    int high = count-1;                 // max possible index of item
                                        // item in [low .. high]  (if present)
    while (low <= high){
        int mid  =  (low + high) / 2;
        int comp =  value.compareTo(data[mid]);
        if (comp == 0)                  // item is present
            return true;
        if (comp < 0)                   // item in [low .. mid-1]
            high = mid - 1;             // item in [low .. high]
        else                            // item in [mid+1 .. high]
            low = mid + 1;              // item in [low .. high]
    }
    return false;    // item in [low .. high] and low > high,
                     // therefore item not present
}
```

low mid high

Cost of SortedArraySet: with Binary Search: • contains: O(log(n)) • add: O(n) • remove: O(n)

LEC-17

-- Selection sort: 依次找到最小的数并在新序列中排序

➢ sort (34, 10, 64, 51, 32, 21) in ascending order

Sorted part	Unsorted part	Swapped
	34 **10** 64 51 32 21	10, 34
10	34 64 51 32 **21**	21, 34
10 21	64 51 **32** 34	32, 64
10 21 32	51 64 **34**	51, 34
10 21 32 34	64 **51**	51, 64
10 21 32 34 51	**64**	--
10 21 32 34 51 64		

```
public void selectionSort(E[ ] data, int size, Comparator<E> comp){
    // for each position, from 0 up, find the next smallest item
    // and swap it into place
    for (int place=0; place<size-1; place++){
        int minIndex = place;
        for (int sweep=place+1; sweep<size; sweep++){
            if (comp.compare(data[sweep], data[minIndex]) < 0)
                minIndex=sweep;
        }
        swap(data, place, minIndex);
    }
```

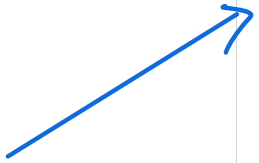
Worst = average = $O(n^2)$

-- Insertion sort: 依次找数，若是都比前面的都大，就向前移（理牌）

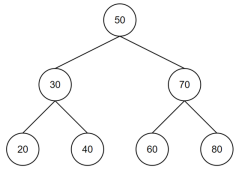
➢ sort (34, 8, 64, 51, 32, 21) in ascending order

Sorted part	Unsorted part	int moved
	34 8 64 51 32 21	
34	**8** 64 51 32 21	-
8 34	**64** 51 32 21	34
8 34 64	**51** 32 21	-
8 34 51 64	**32** 21	64
8 32 34 51 64	**21**	34, 51, 64
8 21 32 34 51 64		32, 34, 51, 64

```java
    /* 哨兵划分 */
    int partition(int[] nums, int left, int right) {
        // 以 nums[left] 为基准数
        int i = left, j = right;
        while (i < j) {
            while (i < j && nums[j] >= nums[left]) j--;
            // 从右向左找首个小于基准数的元素
            while (i < j && nums[i] <= nums[left]) i++;
            // 从左向右找首个大于基准数的元素
            swap(nums, i, j); // 交换这两个元素
        }
        swap(nums, i, left); // 将基准数交换至两子数组的分界线
        return i;
    }
// 返回基准数的索引
```

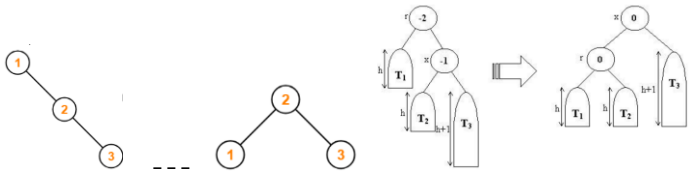


Tree Traversals

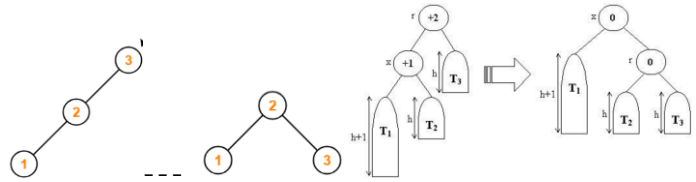


Depth First Traversals:
(a) Post-order (Left, Right, Root) : 20 40 30 60 80 70 50
(b) Pre-order (Root, Left, Right) : 50 30 20 40 70 60 80
(c) In-order (Left, Root, Right) : 20 30 40 50 60 70 80
Breadth First or Level Order Traversal : 50 30 70 20 40 60 80

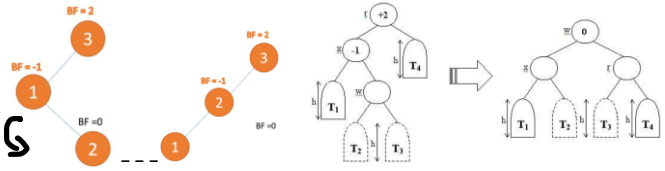
Rotation



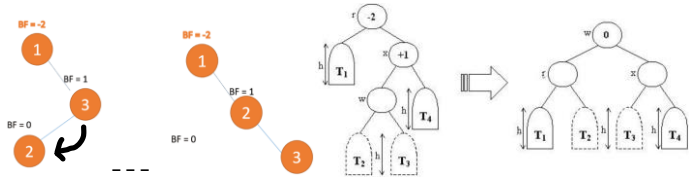
LL： 插入左孩子的左子树，右旋



RR： 插入右孩子的右子树，左旋



LR： 插入左孩子的右子树，先左旋，再右旋



RL： 插入右孩子的左子树，先右旋，再左旋

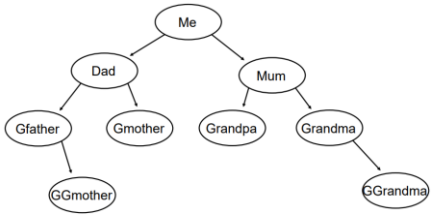
Algorithm	Average	Worst case
Search	$O(\log n)$	$O(\log n)$
Insert	$O(\log n)$	$O(\log n)$
Delete	$O(\log n)$	$O(\log n)$
Space	$O(n)$	$O(n)$

LEC-22 Binary tree 码

```
public interface BinaryTree<T> {
    BinaryTree<T> BinaryTree(T value);

    BinaryTree<T> getLeft();
    BinaryTree<T> getRight();
    void setLeft(BinaryTree<T> subtree);
    void setRight(BinaryTree<T> subtree);
    BinaryTree<T> find(T value);
    boolean contains(T value);

    boolean isEmpty();
    boolean isLeaf();
    int size();
}
```

```
public class BinaryTree<V> {

    private V value;
    private BinaryTree<V> left;
    private BinaryTree<V> right;

    /**Creates a new tree with one node
        (a root node) containing the specified value  */
    public BinaryTree(V value) {
      this.value = value;
    }

  public V getValue() {
      return value;
    }
   public BinaryTree<V> getLeft() {
      return left;
    }
   public BinaryTree<V> getRight() {
      return right;
    }
   public void setValue(V val) {
       value = val;
    }
   public void setLeft(BinaryTree<V> tree) {
       left = tree;
    }
   public void setRight(BinaryTree<V> tree) {
       right = tree;
    }

  public boolean isLeaf() {
      return (right == null) && (left == null);
    }

   public BinaryTree<V> find(V val) {
      if (value.equals(val))      return this;
      if (left != null) {
          BinaryTree<V> ans = left.find(val);
          if (ans != null)          return ans;
      }
      if (right != null) {
          BinaryTree<V> ans = right.find(val);
          if (ans != null)          return ans;
      }
      return null;
    }
```

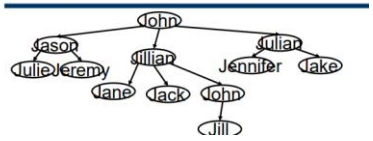
```
public static void main(String[] args){
    BinaryTree<String> myTree = new BinaryTree<String>("Me");
    myTree.setLeft(new BinaryTree<String>("Dad"));
    myTree.setRight(new BinaryTree<String>("Mum"));
    myTree.getLeft().setLeft(new BinaryTree<String>("Gfather"));
    myTree.getLeft().setRight(new BinaryTree<String>("Gmother"));
    myTree.getRight().setLeft(new BinaryTree<String>("Grandpa"));
    myTree.getRight().setRight(new BinaryTree<String>("Grandma"));
    myTree.getRight().getRight().setRight(
              new BinaryTree<String>("GGrandma"));

    BinaryTree<String> gf = myTree.find("Gfather");
    if (gf!=null)
        gf.setRight(new BinaryTree<String>("GGmother"));
}
```

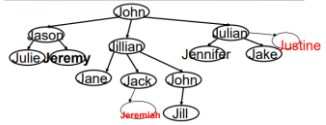


General tree

```
  public V getValue() {
     return value;
  }

  public List< GeneralTree<V> > getChildren() {
     return children;
  }

  public GeneralTree<V> getChild(int i) {
     if (i>=0 && i < children.size())
        return children.get(i);
     else
        return null;
  }

  public void addChild(GeneralTree<V> child) {
      children.add(child);
   }
   public void addChild(int i, GeneralTree<V> child) {
      children.add(i, child);
   }

  public GeneralTree<V> find(V val) {
      if (value.equals(val))       return this;
      for(GeneralTree<V> child : children){
          GeneralTree<V> ans = child.find(val);
           if (ans != null)          return ans;
       }
      return null;
   }
```

```
public static void main(String[] args){
 GeneralTree<String> mytree = new GeneralTree<String>("John");
    mytree.addChild(new GeneralTree<String>("Jason"));
    mytree.addChild(new GeneralTree<String>("Jillian"));
    mytree.addChild(new GeneralTree<String>("Julian"));
    mytree.getChildren().get(0).addChild(new GeneralTree<String>("Julie"));
    mytree.getChildren().get(0).addChild(new GeneralTree<String>("Jeremy"));
    mytree.getChildren().get(1).addChild(new GeneralTree<String>("Jane"));
    mytree.getChildren().get(1).addChild(new GeneralTree<String>("Jack"));
    mytree.getChildren().get(1).addChild(new GeneralTree<String>("John"));
    mytree.getChildren().get(2).addChild(new GeneralTree<String>("Jennifer"));
    mytree.getChildren().get(2).addChild(new GeneralTree<String>("Jake"));
    mytree.getChildren().get(1).getChildren().get(2).addChild(new
            GeneralTree<String>("Jill"));
}
```






```
GeneralTree<String> thrd = mytree.getChildren().get(2);
thrd.addChild(new GeneralTree<String>("Justine"));

GeneralTree<String> gc = mytree.find("Jack");
if (gc!=null)
    gc.addChild(new GeneralTree<String>("Jeremiah"));
```

```
public static void printAll(GeneralTree<String> tree, String indent){
    System.out.println(indent+ tree.getValue());
    for(GeneralTree<String> child : tree.getChildren())
      printAll(child, indent+"  " );
  }
```

Preorder

```
public static void printAllInorder(GeneralTree<String> tree, String indent) {
    if (tree.getChildren().isEmpty()) {
        System.out.println(indent + tree.getValue());
    } else {
        GeneralTree<String> firstChild = tree.getChildren().get(0);
        printAllInorder(firstChild, indent + " ");
        System.out.println(indent + tree.getValue());

        for (int i = 1; i < tree.getChildren().size(); i++) {
            GeneralTree<String> child = tree.getChildren().get(i);
            printAllInorder(child, indent + " ");
        }
    }
}
public static void printAllPostorder(GeneralTree<String> tree, String indent) {
    for (GeneralTree<String> child : tree.getChildren()) {
        printAllPostorder(child, indent + " ");
    }
    System.out.println(indent + tree.getValue());
}
```

Inorder

Postorder

LEC-23

-- HashTable

呈现映射关系：集合 A 的某个元素只能对应集合 B 中的一个元素。但集合 B 中的一个元素可能对应多个集合 A 中的元素。映射叫做 hashing

large table size means wasted space // small table size means more collisions

Hash function: 将键映射到正确的目标所在位置的函数

- takes a variable-size input k and
- returns a fixed-size string (or int), which is called the ***hash value*** h (that is, h = H(k))

may map several different keys to the same hash value.

- avoids collisions
- spreads keys evenly in the array
- inexpensive to compute - must be O(1)

A good hash function

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

For integers

Hash Functions for Strings

- must be careful to cover range from 0 through capacity-1
- some poor choices
  - summing all the ASCII codes
  - multiplying the ASCII codes
- important insight
  - letters and digits fall in range 0101 and 0172 octal
  - so all useful information is in lowest 6 bits
- hash(s) is O(1)

Mid-square Method

1. 初始种子为一个四位数（可以自己设定，比如 1234）。
2. 将种子平方，得到一个八位数。
3. 取得这个八位数的中间四位数作为下一个随机数，即为本次迭代的结果。
4. 重复 2、3 步骤，得到一系列随机数。

Dealing with Collisions

- ***open addressing*** - collision resolution
- key/value pairs are stored in array slots
- ***linear probing***
  - hash(k, i) = (hash1(k) + i) mod N
  - increment hash value by a constant, 1, until free slot is found
  - simplest to implement
  - leads to *primary clustering*
- ***quadratic probing***
  - hash(k, i) = (hash1(k) + c_1*i + c_2*i*i) mod N
  - leads to *secondary clustering*
- ***double hashing***
  - hash(k, i) = (hash1(k) + i*hash2(k)) mod N
    - avoids clustering

LEC - 24-27

--Graph

V 点数 E 边数

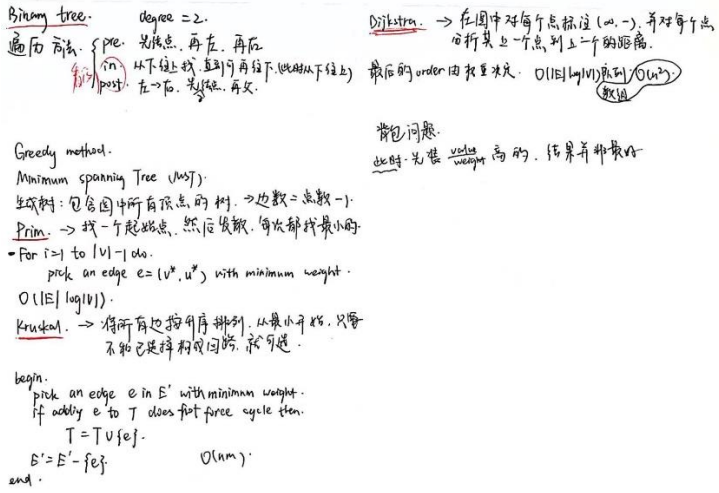
The sum of the values of the degrees, $d(V)$, over all the vertices of a simple graph is twice the number of edges:

$$\sum d(V_i) = 2|E|$$

在做 adjcency matrix 时若有两条边则值为 2

对于至少有两个点的 tree: |E| = |V| - 1

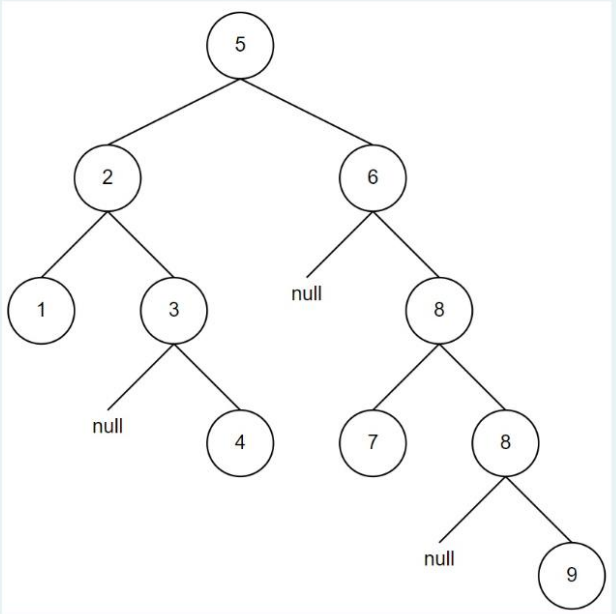
Dijkstra: 对每一个点标注



Drag-and-drop the correct answers to their missing places in order to complete the user-defined Generic classes' program segment below.

```
public class Car< T > {
    String model;
    int vinNumber;
    T serialCode;
    T weight;
    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<>("F150", 38038282, "A39384224", "3004.5943");
        Car< Integer > sedan = new Car<>("Camry", 293393934, 102929383, 8938822);

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

A Binary Search Tree (BST) was created as shown below.



	Correct Integer Sequence
Index 0	5
Index 1	2
Index 2	1
Index 3	3
Index 4	4
Index 5	6
Index 6	8
Index 7	7
Index 8	8
Index 9	9

PRE

Drag-and-drop the correct and most efficient steps in implementing the process of accessing the content while virtually removing it from the stack. Note that your sequence must absolutely match the step numbers to the left-most column of the table otherwise marks will be deducted for each incorrect match.

Step 1	Checks if the stack is empty.
Step 2	If the stack is empty, produces an error and exit.
Step 3	Otherwise, accesses the data element at which top is pointing.
Step 4	Decreases the value of top by one.
Step 5	Returns success.

- Otherwise, increments top to point next empty space.
- Returns success.
- Checks if the stack is full.
- Repeats Step 1-3.
- Decreases the value of top by one.
- Checks if the stack is empty.
- Otherwise, accesses the data element at which top is pointing.
- If the stack is empty, produces an error and exit.

Generics make errors appear at compile time rather than at run time.	Type Safety
Data can simply be retrieved from an ArrayList without explicitly specifying their types.	Individual Type Casting Not Needed

We can implement procedures that work on different types of objects, and at the same time, they are type-safe too.	Promotes Generic Algorithms
We can write a method/class /interface once and use it for any type we want.	Code Reusability

```
   /**
   * 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.
   **/
   public void add(int element){
      if (manyItems == data.length){
         ensureCapacity((manyItems + 1)*2);
        // Ensure twice as much space as we need.
      }

      data[manyItems] = element;
      manyItems++;
   }
/**
   * 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.
   **/
  public void addMany(int... elements){
      if (manyItems + elements.length > data.length){
         ensureCapacity((manyItems + elements.length)*2);
        // Ensure twice as much space as we need.
      }

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

Two different ways to create an array:

- Array: simple fixed sized arrays.
  - int[] arr = new int[3];
- ArrayList: Dynamic sized arrays, implements List interface
  - ArrayList<Integer> arrL = new ArrayList<>(2);

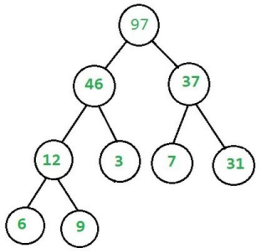
- Generic type: a generic class or interface that is parameterized over type.
  - Such as : `public class ArrayList<E>`
- Type parameters can represent only reference types, not primitive types (like int, double and char).
- The most commonly used type parameter names are:
  E - Element (used extensively by the Java Collections Framework)
  K - Key
  N - Number
  T - Type
  V - Value

- *Generic methods* are methods that introduce their own type parameters.
- Can be called with arguments of different type. Compiler handles the rest.
  - public void myMethod(Integer i);
  - public void myMethod(Double d);
  - public <T> void myMethod(T t);
- Code example.

- A binary tree is a recursive data structure where each node can have 2 children at most.
- Binary Search Tree, is a node-based binary tree data structure which has the following properties:
- The left subtree of a node contains only nodes with keys lesser than the node's key.
- The right subtree of a node contains only nodes with keys greater than the node's key.
- The left and right subtree each must also be a binary search tree. There must be no duplicate nodes.

Binary Heap

- It should be a **complete tree** (i.e. all levels except last should be full).
- Every node's value should be greater than or equal to its child node (considering max-heap).



- Types of binary trees
  - Full binary tree : each node has either 0 or 2 children
  - Perfect binary tree : all interior nodes has two children and all leaves at the same level.
  - **Complete binary tree** : every level, *except possibly the last*, is completely filled, and all nodes in the last level are as far left as possible.

If root is at index `[0]`, one node is at index `[i]`,

- the parent of this node is at $\lfloor (i-1)/2 \rfloor$ if $i \neq 0$, i.e. integer division with truncation.
- the left child is at `[2i+1]`, and the right child is at `[2i+2]`.

How to check if a tree is a complete tree?

- Algorithm :
  - Calculate the number of nodes (count) in the binary tree.
  - Start recursion of the binary tree from the root node of the binary tree with index (i) being set as 0 and the number of nodes in the binary (count).
  - If the current node under examination is NULL, then the tree is a complete binary tree. Return true.
  - If index (i) of the current node is greater than or equal to the number of nodes in the binary tree (count) i.e. (i>= count), then the tree is not a complete binary. Return false.
  - Recursively check the left and right sub-trees of the binary tree for same condition. For the left sub-tree use the index as (2*i + 1) while for the right sub-tree use the index as (2*i + 2).

- How do we check if a tree is a binary heap?
  - It should be a **complete binary tree**
  - Every node's value should be greater than or equal to its child node (considering max-heap).

How to heck if a complete tree is a Heap?

Algorithm:

Every Node can have 2 children, 0 child (last level nodes) or 1 child (there can be at most one such node).

- If Node has no child then it's a leaf node and return true (Base case)
- If Node has one child (it must be left child because it is a complete tree), compare this node with its left child.
- If Node has both child then check heap property at Node at recur for both subtrees.

Stack

- clone():
  - Generate a copy of this array stack. Please refer to *ArrayBag* class, we will discuss about this method in detail in the coming tutorial.
  - **Returns**: a copy of the stack.
- peek():
  - Looks at the object at the top of this stack without removing it from the stack.
  - **Returns**: the object at the top of this stack (the last item of the array).
  - **Throws**: *EmptyStackException* - if this stack is empty.

pop():

- Removes the object at the top of this stack and returns that object as the value of this function.
- **Returns**: The object at the top of this stack (the last item of the array).
- **Throws**: *EmptyStackException* - if this stack is empty.

push():

- Pushes an item onto the top of this stack. This has exactly the same effect as add an element to the stack.
- **Parameters**: item - the item to be pushed onto this stack.
- **Returns**: the item argument.

remainedCapacity():

- Find out how many items can be pushed to the stack still.
- **Returns**: the remained capacity of the stack.

BST

search():

- function to search for a given key.
- return boolean.

delete():

- function to delete a given key in the tree.
- No return.

printRange():

- function to print all the keys which in the given range [k1..k2]. Given two values k1 and k2 (where k1 < k2). Print all the keys of tree in range k1 to k2. i.e. print all x such that k1<=x<=k2 and x is a key of given BST. Print all the keys in increasing order.
- parameters: int k1, int k2.
- No return.
- Algorithm:
  1) If value of node's key is greater than k1, then recursively call in left subtree.
  2) If value of node's key is in range, then print the node's key.
  3) If value of node's key is smaller than k2, then recursively call in right subtree.