JAVA 2420 Note

By Samuel_233, UoU

笔记可能有错误

基础复习

Types in Java

	Primitive	Reference
描述	直接储存在内存	内存中储存指针，指向另一篇内存
例子	Boolean, char, byte, short, int, long, float, double	String, array, object
传参后	直接复制，在函数中做的修改	传递的是地址，在函数中做的

	不会影响实际变量(除非 return)	修改会直接影响到原本的参数
备注		String 是一个特例，虽然它是引用，但函数修改不会使原变量修改

Operator in Java

Operator	Example
Arithmetic operator	+, -, *, /, %, ++, --
Relational operator	==, >=, <=, !=
Bitwise operator	&, >>, <<, >>>, ~
Logical operator	&&, ‖, !
Assignment operator	=, +=, -=, *=, /=, &=, ^=, |=, >>=, <<=, ~=
Conditional operator	? : int x = 10; int y = (x>5)? 1:-1

.equals()和==

在做比较的时候，有 equals()和==，这两个比较分别不一样，“==”是比较两个 reference 的地址，如果相同，则返回 true，而 equals 则是比较每个元素，在比较 String, array 的时

候，除非地址相同，不然==返回的都是 false

Array & Array List

Array	Array List
Fixed Size 修改数值用[]	大小可变 修改数值用.get(), .set()

OOP

Class

一个 Class 是一个对象的模板，它拥有自己的 attribute 和自己的函数

当一个 attribute 或者 method 是 static 的时候，这个方法不需要依靠 object 就可以调用

(直接 class.method() 调用即可)

Encapsulation:

OOP 的封装特性可以管理外界获取这个 class 的属性的方法，还可以重要隐藏信息

Code Re-use:

当两个函数看起来比较相似的时候，可以把相同的方法提取出来

Inheritance:

Super class (base class) →继承→ Sub Class(derived)

我们可以把 class 继承，当继承后，子类可以获得 Method 和 Field(Attribute)，但是继承不了 private 的东西，也不能继承构造函数(可以使用 super()来在构造函数中使用父类的构造)

继承的好处:	继承不能干什么:
1: 覆写之前的方法 2: 增加更多的方法 3: 增加更多的参数	1: 不能从多个父类继承 2: 不能改变父类的方法

Method Overloading

同样的方法名称，但是由于参数不一样，函数的作用也不一样

Method Overriding

同样的方法名称，但是由于子类有一个新方法，所以覆写了父类继承的方法

Abstract base classes

Cannot be instantiated

Method may be either abstract or concrete

Can be extended by either abstract or concrete class

抽象类，就是不能实例化的类，但它可以被正常的类/抽象类继承，类似只是给了一个大框架，方法什么的可以由子类实现，所以抽象类不能写 final，因为他需要被子类继承

If inherit from abstract class(for a concrete class), You must implement every abstract method，因为父类(抽象类)的方法大部分都是空的

在子类实现父类的空方法的时候，可以在方法上一行加上@Override，这样可以更加清楚，程序也可以识别出一些潜在的风险

Interface

Every method must be abstract and public

Every field must be public, static, and final

Interface 跟 abstract 有点类似，但抽象类需要继承，而 interface 不是一个类，是一种接口，他需要用 implement 方法实现.

Implements

Implement 和 extend 有点相似，但 extend 只能有一个，但一个类可以有无数个实现．比如我们想让我们自己写的类也有.compareTo() 方法，我们就可以写

public class TYPE implements Comparator<TYPE>

然后再这个类里面自己实现一个 compare 方法.

.compare() 和.compareTo()

```
public class Student implements Comparable {
private String name;
private int age;
public Student(String name, int age) {
this.name = name;
this.age = age;
}
public int getAge() {
return this.age;
}
public String getName() {
return this.name;
}
@Override
public String toString() {
return "";
}
@Override
public int compareTo(Student per) {
if(this.age == per.age)
return 0;
else
return this.age > per.age ? 1 : -1;
}

public static void main(String[] args) {
Person e1 = new Person("Adam", 45);
Person e2 = new Person("Steve", 60);
int retval = e1.compareTo(e2);

switch(retval) {
case -1: {
System.out.println("The " + e2.getName() + " is older!");
break;
}

case 1: {
System.out.println("The " + e1.getName() + " is older!");
break;
}

default:
System.out.println("The two persons are of the same age!");

}
}
}
```

Integer.compare(4,9) -> -1

Integer.compare(4,4) -> 0

Integer.compare(4,3) -> 1

“hi” .compareTo(“hello”)->4

“hi” .compareTo(“hi”) ->0

“hi” .compareTo(“hi how are you”) ->-12

可以看到, compare 和 compareTo 都可以用来比较，但具体有什么区别呢?

其实这是两个不同的接口，但他们都可以让我们实现比较我们自己定义的类

compareTo()方法是 Comparable 接口的函数，他需要在最外层的 class 中写上“implement Comparable”才可以在里面再写 compareTo 方法：如右侧上方图片．这样就导致我们很难对 String 等类使用 Comparable，除非我们写一个 string 的子类，让子类 implement Comparable

```java
import java.util.Comparator;

public class School {
private int num_of_students;
private String name;
public Company(String name, int num_of_students) {
this.name = name;
this.num_of_students = num_of_students;
}
public int getNumOfStudents() {
return this.num_of_students;
}
public String getName() {
return this.name;
}
}
public class SortSchools implements Comparator {
@Override
public int compare(School sch1, School sch2) {
if(sch1.getNumOfStudents()== sch2.getNumOfStudents())
return 0;
else
return sch1.getNumOfStudents() > sch2.getNumOfStudents() ? 1 : -1;
}
public static void main(String[] args) {
School sch1 = new School("sch1", 20);
School sch2 = new School("sch2", 15);
SortSchools sortSch = new SortSchools();
int retval = sortSch.compare(sch1, sch2);
switch(retval) {
case -1: {
System.out.println("The " + sch2.getName() + " is bigger!");
break;
}
case 1: {
System.out.println("The " + sch1.getName() + " is bigger!");
break;
}
default:
System.out.println("The two schools are of the same size!");
}
}
}

Output:
The sch1 is bigger!
```

而 compare()方法是 Comparator 接口的实现，他并不需要修改原有的类，他只需要在我们想比较的类里面单独写一个 compare 方法 implement Comparator 就可以覆写之前的 object compare 的方法了.

Comparable v/s Comparator in Java

Comparable in Java	Comparator in Java
Comparable interface is used to sort the objects with natural ordering.	Comparator in Java is used to sort attributes of different objects.
Comparable interface compares “this” reference with the object specified.	Comparator in Java compares two different class objects provided.
Comparable is present in java.lang package.	A Comparator is present in the java.util package.
Comparable affects the original class, i.e., the actual class is modified.	Comparator doesn’t affect the original class
Comparable provides compareTo() method to sort elements.	Comparator provides compare() method, equals() method to sort elements.

[一个 Comparable 和 Comparator 的例子]

Generic Programming and Collection Classes

Generics

Generics 是另一种方法避免代码复用(第一种是继承)

在我们写方法的时候，需要定义输入的类型，不同的类型会重载.

虽然这样有好处，但有时候会很麻烦：比如我们想创建一个方法，能把任意的 array 变换成

```
public static <T> ArrayList<T> fromArray(T[]array){
    ArrayList<T> arrayList = new ArrayList<T>();
    for(T element: array){
            arrayList.add(element);
    }
    return arrayList;
}
```

array list，如果用以前的方法，就需要写无数个方法，但当使用 generics 后，就可以让方法接受任意 type:

这里面的 T 就代表任意 Type (当然这也可以是其他字母如 E)

在上面第一行可以看到有两个尖括号，只有第一个尖括号是都要写的，第二个尖括号是因为 ArrayList 也要指定类型，所以加上个<T>.

```
public class CS2420StudentGeneric<Type> extends UofUStudent{
    2 usages
    private Type contactInfo;
    3 usages
```

上面的是对于方法的 generic，而 class 的 generic 有点不一样:

这个只需要写一个 T 在类名后面就行了，并不需要像方法那样既在方法前面写 T，又在方法里面用 T, method 之所以这么做是为了防止有类就叫 “T”，所以要声明两遍

并且，如果 class 是 generic 的，我们不能声明一个传入泛型当参数的 static 函数，因为这个 class 的种类是 generic 确定的，而 class 需要实例化才能确定种类，但 static 函数不需要实例化，所以这会出现错误

*详细信息请看【【Java 泛型】Java Generics 泛型教程 价值 19 美元 付费完整版】

https://www.bilibili.com/video/BV13M411H7Dk/

Collection Frame

Java Collection Frame 简称 JCF，它包含了两个大类: Collection<T> 和 Map<T,S>

两组都是字面意思，前者是一组数据，后者是数据的映射(类似字典)

任何实现 Collection / Map 中的接口(如 list)都也会自动实现 Collection

Collection

Collection 接口又有两个子接口: Lists & Sets

Lists 就是类似 ArrayList (ArrayList 其实就是 List 的一种实现),

Sets 也是一组数据，但是 Sets 中的数据都不会重复出现

Collection 接口自带了一些操作方法，他们输入的参数都是非常通用的参数(object) 所以这就是为什么当我们想输入 integer 进去的时候不能写 int 而是得写 Integer 作为他的 type，前者是 primitive type，后者是 Object:

http://opendatastructures.org/ods-java/1_Introduction.html

自带方法：

- $coll.size()$ — returns an int that gives the number of objects in the collection.
- $coll.isEmpty()$ — returns a boolean value which is true if the size of the collection is 0.
- $coll.clear()$ — removes all objects from the collection.
- $coll.add(tobject)$ — adds tobject to the collection. The parameter must be of type *T*; if not, a syntax error occurs at compile time. (Remember that if *T* is a class, this includes objects belonging to a subclass of *T*, and if *T* is an interface, it includes any object that implements *T*.) The add() method returns a boolean value which tells you whether the operation actually modified the collection. For example, adding an object to a Set has no effect if that object was already in the set.
- $coll.contains(object)$ — returns a boolean value that is true if object is in the collection. Note that object is **not** required to be of type *T*, since it makes sense to check whether object is in the collection, no matter what type object has. (For testing equality, null is considered to be equal to itself. The criterion for testing non-null objects for equality can differ from one kind of collection to another; see Subsection 10.1.6, below.)
- $coll.remove(object)$ — removes object from the collection, if it occurs in the collection, and returns a boolean value that tells you whether the object was found. Again, object is not required to be of type *T*. The test for equality is the same test that is used by contains().
- $coll.containsAll(coll2)$ — returns a boolean value that is true if every object in coll2 is also in coll. The parameter can be any collection.
- $coll.addAll(coll2)$ — adds all the objects in coll2 to coll. The parameter, coll2, can be any collection of type *Collection<T>*. However, it can also be more general. For example, if *T* is a class and *S* is a sub-class of *T*, then coll2 can be of type *Collection<S>*. This makes sense because any object of type *S* is automatically of type *T* and so can legally be added to coll.
- $coll.removeAll(coll2)$ — removes every object from coll that also occurs in the collection coll2. coll2 can be any collection.
- $coll.retainAll(coll2)$ — removes every object from coll that **does not occur** in the collection coll2. It "retains" only the objects that do occur in coll2. coll2 can be any collection.

- $coll.toArray()$ — returns an array of type Object[] that contains all the items in the collection. Note that the return type is Object[], not T[]! However, there is another version of this method that takes an array of type T[] as a parameter: the method coll.toArray(tarray) returns an array of type T[] containing all the items in the collection. If the array parameter tarray is large enough to hold the entire collection, then the items are stored in tarray and tarray is also the return value of the collection. If tarray is not large enough, then a new array is created to hold the items; in that case tarray serves only to specify the type of the array. For example, coll.toArray(new String[0]) can be used if coll is a collection of *Strings* and will return a new array of type String[].

List 接口

Queue, Stack, Deque

除了最常见的 ArrayList 以外，这以下三种数据结构也是包含在 List 接口中的

Queue 是一种数据结构，符合先进先出的特性(FIFO)：就跟名字一样，可以把数据想象成排队，先放进去的数据也只能先取出来.

add(x) /enqueue()：放入队列一个元素

remove()/dequeue()：取出队列中最先放进去的一个元素

remove(x)/deleteMin()：按特定规则取出元素，让我们不必一直遵循先进先出的规则(priority queue)

Stack 是一种先进后出的数据结构(LIFO)，就像我们洗盘子时把盘子垒成一摞，先放进去的盘子在最下面，我们只能取出最上面的盘子.

为了防止混淆上面的 add/remove 函数，我们使用 push(x)和 pop()函数来代表增删堆栈中

的元素

Deque 是 FIFO Queue 和 LIFO Queue (Stack) 的概括。 Deque 表示元素序列，具有正面和背面。元素可以添加到序列的前面或序列的后面. Deque 可以用的函数是:$addFirst(\quad), addLast(\quad), removeFirst(\quad), removeLast(\quad)$

值得注意的是: Stack 可以仅使用 $addFirst()$ 和 $removeFirst()$ 来实现，而 FIFO Queue 可以使用$addLast()$和$removeFirst()$

Set 接口

Set 接口分为 USet 和 SSet. 其中 USet 表示的是无序集(unsorted Set)，一组无序的唯一的元素.

SSet 表示的是有序的唯一的元素(Sorted Set)，他与 USet 的方法大部分都相同:

$$size(\quad), add(x), remove(x)$$

唯一不同的就是$find(x)$，相较于 USet 的寻找方法，他会返回有用的数据就算没有找到那个元素，但是他的运行时间是$O(\log(N))$，相较于 USet 的哈希表寻找方法(永远固定时间)来说，还是需要更大的消耗

Algorithm

Computation measurement

我们使用“N”表示数据的数量

问题：当 N 数量级很大的时候，这个算法需要多久？

Big-O Notation：

O(1)：不管数量多大执行时间固定

O(N)：所需时间与数据量呈线性相关

O(N^2)：所需时间与 N 呈平方相关

O($\log(N)$)：所需时间与 logN 相关

P vs NP

P: set of decision problems that can be solved in polynomial time(by the deterministic sequential machine)

NP: set of decision problems that can be verified in polynomial time. Equivalently, can be solved by non-deterministic machine)

Sort

Selection Sort

对于每个元素，选取这个元素后面存在的最小值并放到当前位置.

所有情况都时间复杂度都是$O(N^2)$

Insertion Sort

遍历每个元素，每个元素循环与前面的元素交换，直到遇到前面的元素比自己小的情况停止.

最好的情况是$O(N)$，最差的情况是$O(N^2)$，平均情况是$O(N^2)$

Shell Sort

是一种 insertion sort 的变种，我们会选择不同的 gap 来交换元素，这样能避免一些 insertion 交换元素很多次的情况.

时间复杂度由 Gap 决定，平均时间复杂度比$O(N^2)$好，比$O(N\ log(N))$差

Merge Sort

Merge sort 就是把一个数组递归式的切成两半，直到每个小数组长度是 1. 因为当长度是 1 的时候，数组就一定是排好序的. 最后再把每个数组合并到一起.

合并的方法就是一起同时遍历 lower->mid 和 mid+1->upper

比较 lower 和 mid+1，哪个数值大就把它存进去再把对应的 index+1

比如第一次比较是$array[mid+1] > array[lower]$，我们就把 mid+1 存进去，再接着遍历$array[lower]$和$array[mid+2]$

时间复杂度为$O\big(Nlog(N)\big)$，空间复杂度是$O(N)$

Quick Sort

Quick sort 的时间复杂度也跟 merge sort 的复杂度一样

1. 选一个数值作为 pivot

2. 把数组小于 pivot 的数放左边，否则放右边

-具体做法是先把 pivot 移到最后,用i, j遍历数组，首先i指向-1(null), j 指向第一个数字. 我们的目标是让i刚好 pivot index 的前面，所以当$arr[j] < pivot$时，j与i前面的一个数对调，且把$i++$，当$arr[j] > pivot$时，只$i++$. 到 j 遍历完最后我们就会发现i会停在一个比 pivot 小的值上，而后面都是比 pivot 大的值. 所以我们只需要把 pivot 和 i 后面的值对调就行

3. 递归式的对两边重复上面的步骤

在理想状态下，我们想把 pivot 值取为中值，这样就能完美分成两份. 但是我们很难求出中值. 不过我们可以取三个值，再算出三个数值的中值，这样至少比随机好一点：取$array[lower], array[mid], array[upper]$.

时间复杂度最佳是$O(N\log(N))$，最差情况是$O(N^2)$，一般情况是$O(N\log(N))$

Topological Sort

对图的顶点创建一个顺序，这个顺序的规则是：如果 W 是 V 的邻居$V \to W$，则排序的时候 V 必须在 W 的前面.

比如 $a \to b, a \to c, c \to b$，排序完则是$a, c, b$

这种排序可以应用在任务管理方面，比如一个任务必须在另一个任务之前完成.

如何实现:

核心是计算“in degree”，也就是指向这个顶点的边的数量

$$Q = Queue(empty)$$

$$L = List(empty) - 最后储存排序结果$$

$$enqueue\ each\ vertex\ with\ inDegree = 0$$

$$While(Q\ !=\ empty)\{$$
$$\quad v = Q.dequeue(\ \ );$$
$$\quad L.append(v);$$
$$\quad foreach(neighbor\ n\ in\ v)\{$$
$$\quad\quad n.inDegree - -;$$
$$\quad\quad if(n.inDegree == 0)\ Q.enqueue(n)$$
$$\quad \}$$
$$\}$$

注意：本方法没有检测是否循环，如果图包含循环，是 sort 不了的，因为不能存在两个互相依赖的节点.

总结

Sort	best	avg	worst	Stable? (两相同元素排序后相对位置不变)	In-place? (不占用额外空间)
Selection	$O(N^2)$	$O(N^2)$	$O(N^2)$	N	Y
Insertion	$O(N)$	$O(N^2)$	$O(N^2)$	Y	Y
Shell	?	?	?	N	Y
Merge	$O(N\log(N))$	$O(N\log(N))$	$O(N\log(N))$	Y	N
Quick	$O(N\log(N))$	$O(N\log(N))$	$O(N^2)$	N	Y
Heap	$O(N\log(N))$	$O(N\log(N))$	$O(N\log(N))$	N	Y

Non Comparation Sort

Bucket Sort

这个排序算法是把一堆东西切分成一个个的 bucket，然后再把每个 bucket 分别 sort，接下来，按照 bucket 的顺序遍历，由于里面都排序好了，遍历出来的自然也是排好序的

这样的优势是我们可以多线程来同时排序多个 bucket

比如，我们有一群动物，可以创建从 a-z 的 bucket，把动物按照名字的首字母放进 bucket 中．接下来用常规 sort 办法来 sort 每个 bucket 里的动物．最后遍历出来放回 array 中

这个算法的 big o 是$O(N^2)$在最坏情况，$O(N+\frac{N^2}{k}+k)$在平均情况, k 是 bucket 的数量，在

很多情况下，我们希望 k 越多越好，甚至跟 n 的数量差不多

Pigeon Hole Sort

这个是 bucket sort 的变种，是用来排序键值对的，把键值对的 key 当做 bucket，剩下的就是跟 bucket 差不多了

Radix Sort

这种也是 bucket sort 的变种，只不过是用来 sort 整数的：取整数最后一位，然后创建 0-9 一种 10 个 bucket 来存放数字．从 bucket 取出后，会发现顺序还是乱的．我们接下来取每个数字的十位数来放进 0-9bucket 中，就可以发现，所有三位数以上的数字是乱的，但两位数和一位数的数字都排序好了接下来就这么一直搞就行

在平均情况下, big o 是$O(k * N)$, k 是数字的位数个数

Counting Sort

Counting Sort 跟之前的那些 bucket sort 不太一样了：

假设有一个 array Integer，或者有着整数 key 的键值对

1：首先我们先要找到最大和最小值 $range = max - min + 1$ 这个范围表示了我们有多少个可能的数据

2：然后再创建一个 range 那么大的 array

3：在 array 中记录下每个 key 的数量(比如 key 是啥就先减去 min 后放到 array 对应的

index 中)

4：最后把这个 array 中的元素遍历一遍就是排好序的了

Traversal Algorithms

最有名的两种遍历是 BFS 和 DFS，分别是广度优先搜索和深度优先搜索

这两种算法的 bigO 都是$O(|V|+|E|)$，但是广度优先搜索总是能找到最短路径(不是最短加权路径)

BFS – breadth-first search:

这种搜索是从顶点的 root 开始，每次搜索都搜索同样深度的顶点，直到当前深度的所有节点搜索完毕，再搜索更深一层的

如果我们想实现：

$$
\begin{aligned}
&root\ vertex\ SET\ root.visited = true\\
&init\ Q: queue\ of\ vertices\\
&Q.enque(root)\\
&while(Q\ !=\ empty)\{\\
&vertex\ v = Q.dequeue(\ \ );\\
&foreach(neighbor\ n\ in\ v)\{\\
&\quad if(not\ visited)\{\\
&\qquad n.visited = true;\\
&\qquad n.previousVertex = v\\
&\qquad Q.enqueue(n)\\
&\qquad \}\\
&\quad \}\\
&\}
\end{aligned}
$$

DFS – Depth First Search

深度优先搜索是从图的根部开始探索，直到访问到最深的地方，再反向遍历刚才访问的节点，直到遇到了有相邻节点的节点.

实现方法:

$$
\begin{aligned}
&DFS(x)\\
&set\ node\ x\ as\ visited\\
&for\ each\ neighbor\ n\ in\ x:\\
&\quad if\ n\ is\ not\ visited\\
&\quad\quad set\ x\ as\ perivious\ vertex\ to\ n: x \rightarrow n\\
&\quad\quad DFS(n)
\end{aligned}
$$

Dijkstra's Algorithm

上面那两种算法虽然简单，但对于有权重的路径不好求一个节点到另一个节点的最短 cost.

这种算法就解决了这种问题.

相较于 BFS 和 DFS 从队列或者栈中取出没有访问过的节点，我们会优先访问 cost 较小的临近节点，并更新路径的消耗如果找到更短的路径

如何实现:

首先把起始节点 distance 设为 0，其余节点 distance 设成无限，并设置成未访问状态

在这里面，比如我们寻找时，从节点 a 开始，则所有节点的 distance 属性都是表示这个节点和 a 的距离

$$
\begin{aligned}
&while(there\ are\ univisited\ vert)\{\\
&\ v = unvisited\ vertex\ with\ smallest\ cost\\
&\ v.visited = true\\
&\ for(neighbor\ n\ of\ v)\{\\
&\quad newDist = v.dist + weight(v,n)\\
&\quad replace\ n.distance\ with\ newDist\ if\ newDist\ is\ smaller
\end{aligned}
$$

```
        }
    }
```

遍历二叉树

有三种递归方法和一种非递归方法：

递归方法：

pre, post, in order 描述的是访问父节点的时机：pre 是先访问父节点，post 是后访问父节点

Pre Order	先访问节点，再访问左子节点，最后访问右边(像 DFS)
Post Order	先访问左子节点，再访问右子节点，最后访问节点(像倒过来的 DFS)
In Order	先访问左子节点，再访问节点，最后访问右子节点 使用这种方法遍历 Binary Search tree 可以获得从小到大的排序好的所有节点

非递归方法：

Level order： 就像 BFS 一样，用 Queue 来规定访问顺序

搜索 Binary Search Tree

因为我们的 BST 是左边小，右边大的，所以可以从 root 节点出发，不停遍历，如果目标比当前节点小就往左边，比当前大就往右边的子节点走．直到找到目标节点，有一点像 binary

search

File Compression

Character Encoding：用某种规律来表示每个字母，如摩斯电码，盲文, unicode 等

压缩文件可以

- 减少储存占用，让上传下载更快
- 把输入的文件 encode，然后占用更少的内存
- 把文件 decode 后可以还原掉原始文件

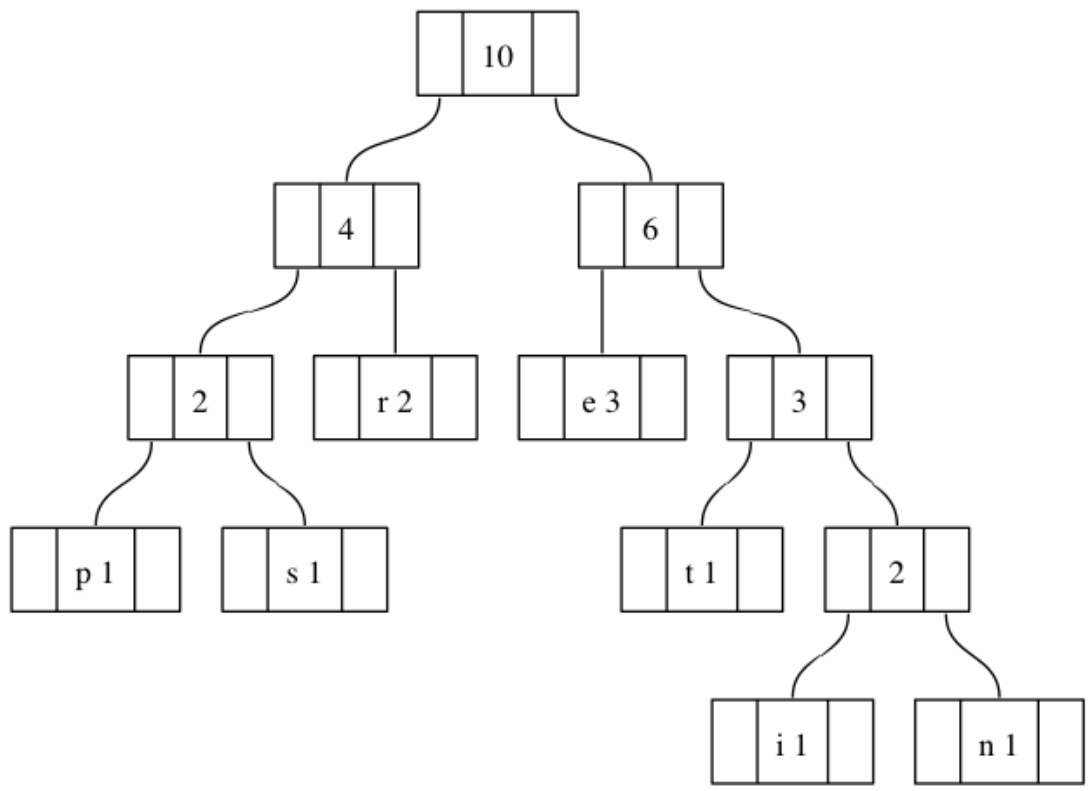
压缩文件的策略主要就是把那些常用的字母用占用更小的符号代替，这样就可以省下占用空间

霍夫曼树

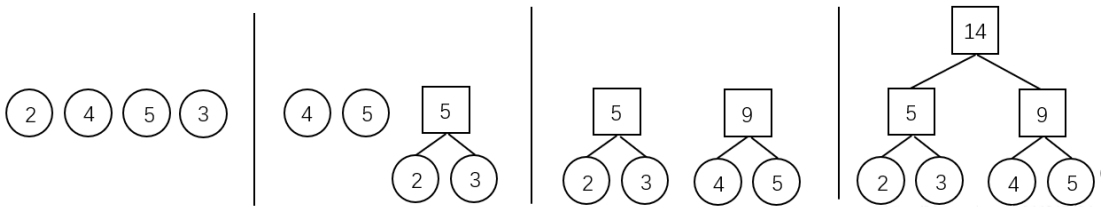
霍夫曼树就是一种这样的压缩方式：比如我们想压缩一个单词: enterprise，这里面 e 出现了 3 次, r 出现了两次，剩下的出现了一次.

我们就可以创建一个 queue，以出现的次数从小到大排．接下来我们就可以构建树了．可看到在 queue 中, i 和 n 出现在最前，所以我们用他们俩先组成一个小树，我们就把它看做一个新字母，出现了 2 回，构建的树是两个 leaf 各一个字母, root 是表示出现的总次数 2.

接下来把这个新 “字母”放回 queue 中，然后继续进行构建.



就像在这里先构建n, i 得到权重 2，接下来合并p, s，就只剩下一个t的权重是 1，所以和i, n树合并，得到权重 3.... 直到只剩一个树



接下来就会在文件头存下我们存入的这些字母，以方便构建回霍夫曼树，还拿之前单词举例表示t的 5 个 byte 是74 00 00 00 01．最开始的 74 是 ASCII 中 t 的表示方式,后面的 01 表示的是出现了 1 次.

压缩率

压缩率指的是一个文件压缩后和压缩前的比，比如e本来的 ASCII 表示方式是 01100101，但

是在上面压缩完后变成了 10，所以小了很多倍，在计算上出现次数后，才是正确的压缩比.

但由于压缩后会有表头文件，所以在压缩很小的文件的时候，文件大小反而容易变大.

Data Struct

Data Struct & Abstract Data Type

Abstract Data type 是对某种数据类型的理论，我们会在里面定义：他们能干什么，能储存什么，或者复杂度是什么．比如 Interface

Data Structure 是已经实现的数据类型，很多时候是 implement Abstract Data Type．比如我们最常写的 class

Linked Data Structed

Linked List

Link List 是一个数据结构，不仅有存储的数据，还会指向下一个数据(仅有一个引用)

比如我们有一个 Node Class:

$$
\begin{aligned}
&class\ Node\{\\
&char\ value;\\
&Node\ nextNode;\\
&\}
\end{aligned}
$$

这个就是每个节点可以存储一个 char，然后这个节点还存着下一个节点的 reference.

这种数据的好处就是增删所需要的时间都是$O(1)$,但是访问的时间是$O(N)$

如果我们想遍历的话:

$$
\begin{aligned}
&Node\ curr = head;\\
&while(curr! = null)\{\\
&(do\ sth);\\
&curr = curr.nextNode;\\
&\}
\end{aligned}
$$

这里的 head 就是表示的这个 Linked list 的第一个节点.

Double Linked List

这个就是 linked list 的升级版，可以双向遍历

$$
\begin{aligned}
&class\ Node\{\\
&char\ value;\\
&Node\ nextNode;\\
&Node\ prevNode;\\
&\}
\end{aligned}
$$

Stack

是一种 abstract data type，它定义的一些函数是：

$push(e)$ 往 stack 最上面添加元素

$peek()$ 找到 stack 最上面的元素

$pop()$ 取出 stack 最上面的元素

上面这三种方法的复杂度都是$O(1)$

这是一种后进先出的数据结构(最后进去的元素会最先出来)

具体的应用就可以看 IDE 如何检测代码的括号完整性：一个一个遍历字母，把每个括号都

push 进 stack 里，如果找到相反的括号就说明这个括号跟 stack 最上面的这个括号是成对

的，就这么一直遍历知道没有多余的括号.

或者是计算机中的计算，我们输入一串数字：$123*+4-$,

计算机会从 1 开始 push 数字，直到遇见操作符比如 “*”，就会 pop 两次出来两个数字作为运算的右边和左边，运算完再把结果 push 回去

$$(1+(2*3))-4$$

Queue

$enqueue(e)$ 把元素添加进队列的最后头

$peek()$ 获得队列最前的一个元素

$dequeue()$ 把队列中最前的一个元素移除

这个数据结构是一种先进先出的结构，就跟名字一样，所有的元素都在排队

Graph

图是一种数据类型由一系列顶点-vertices 和边-edges 互相连接组成, linked lists 就是一种特殊的图.

$$G(V,E)$$
$$V = Vertices$$
$$E = Edges$$

$$|V| = numbers\ of\ vertices$$
$$|E| = numbers\ \ of\ edges$$

机场，早期的互联网都可以用图来表示：每个节点/网站互相连接组成一个系统.

我们还可以把不同的 edge 附加上不同的 cost，这样就可以充当现实中走这条路所花的成本

(这个和途径多少节点是有区别的，途径多少 edge 叫length，而 cost 叫 做weighted length)

图的节点数量很好数，边的数量取决于边是否是双向的-如果一条边是双向的则他是算两条

图的术语

Edge：表示连接一对顶点，如$w是v的neighbor，则(v,w) is\ an\ edge: v \rightarrow w$

Path：表示一系列顶点:v_1, v_2, v_3, v_i，$(v_i v_{i+1})$表示一条边

Digraph: Directed graph 的简写，表示图的边是有方向的

Simple path：一条 path，它不包含重复的顶点

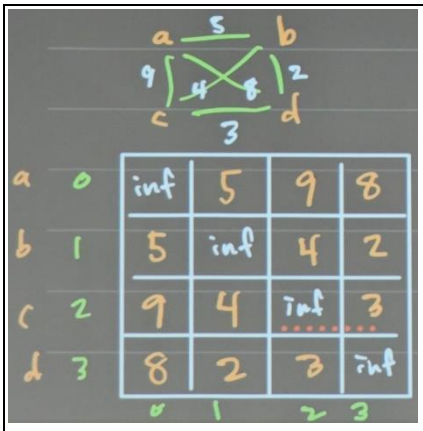
Cycle：一条 path，他的头跟尾是一样的(长度需≥2)

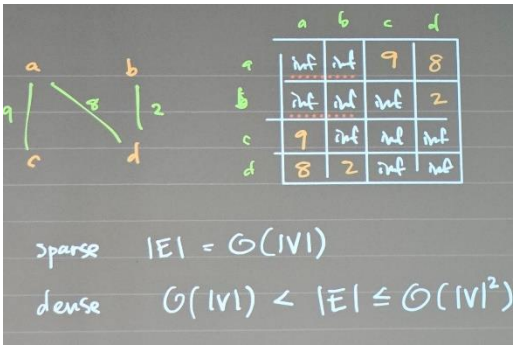
画出一张图

一般图分为 dense graph 和 sparse graph，如果我们想用矩阵表示 sparse graph 的话，

就会比较浪费

Dense Graph	Sparse Graph





对于 sparse graph，我们可以在每个顶点里创建一个 list，存放他连接了哪些其他顶点：

$$a: c, 9 \rightarrow d, 8$$
$$b: d, 2$$
$$c: a, 9$$
$$d: b, 2$$

在这里用箭头表示是因为我们可以用 linked list，因为 linked list 比较擅长储存少量数据，

他不会像 array 那样浪费内存

Tree

树是一种特殊的图：它不包含循环，并且是有向的.

A rooted tree 是一种树：

他的根节点没有任何指向他的边

每个不是根的节点只有一个指向它的边

对于每个不同的节点，都只有一个独特的路径从根部到那个节点

当在树上的一个节点没有子节点(这个节点不指向任何其他节点)时，我们称它为一片树叶

树的术语：

Ancestors：一个节点的所有父节点

Descendants：一个节点的所有子节点

二叉树 - binary tree

二叉树是一种特殊的 Rooted tree，每个节点最多有两个子节点，可以把这两个子节点称作 left child 和 right child.

有些时候会遇到用数组来表示的二叉树，这是一种二叉树特殊的表达形式：

其实这个数组表达的形式就是最常见的从 root 开始 BFS 遍历二叉树，但是我们会把这个二叉树给补完(不能存在只有一个子节点的节点，空元素用 null 表示)，如果节点没有子节点并且它并不是最深的 leaf，则会把这个节点补全到它的子节点到达图的最深处

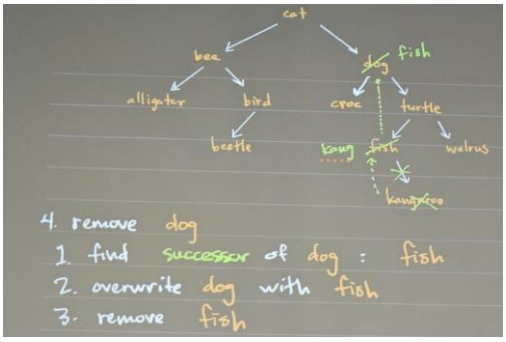
Binary Search Tree

这是一种特殊的二叉树，专门为搜索准备的：这个二叉树的特点就是一个节点的俩子节点左边节点的永远比右边的子节点和父节点小，右子节点比左子节点和父节点要大，这里所指的父节点不仅是这个节点的上一层，还包括所有的父节点

而且要注意的是，这种树是不能添加重复节点的

这种存储方法比 Linked List 和 Array 都好，因为 Array 虽然可以用 binary 快速搜索，但是它插入元素就要把所有元素往后移．而 linked list 虽然插入速度很快，但是遍历速度又很慢，但是寻找最大值和最小值没有 Sorted Array 的$O(1)$快

然而代价是复杂的删除元素方法：



删除树的 leaf 简单，但是想删除有两个子节点的节点就比较复杂：

我们需要找到一个合适的继承节点，移到删除的节点的位置，再把继承节点的子节点往上移

寻找继承节点需要有两个要求：

1. 我们希望继承节点最好是删除节点右侧的最小值

2. 继承节点只能有最多一个子节点

Heap

Heap 是一种由 rooted tree 组成的数据结构，他有一个特性 - heap property

Max-heap property：每个父节点都是大于它的子节点

Min-heap property：每个父节点都是小于它的子节点

这种数据结构的$peek(\quad), insert(\quad)$的耗时都是$O(1)$，$pop()$ 的耗时是$O(\log(N))$

Binary heap 是一种由二叉树组成的 heap，并且拥有 structure property：树是完全填充

的，除了最下面的数据是从左到右填充到最后可能有一点空余

也就是说, binary heap 是一个 balanced tree，它的高度是$h = \log(N)$

比如我们想存储 a-j 到这个结构里：那就是第一行是 a，第二行是 b,c,第三行是 d,e,f,g ...

然后我们如果把 a-j 放到 array 里

0	1	2	3	4	5	6	7	8	9	10
Null	a	b	c	d	e	f	g	h	I	j

P = parent Index	L = left Index	R = right Index
1	2	3
2	4	5
3	6	7
4	8	9
5	10	X

我们可以发现在树中，如果我们知道了一个 parent node 的 index，我们可以很轻松的求出它的左右子节点在 array 中的 index，$l = 2 * p$，$r = 2 * p + 1$

既然这是一棵树，怎么能做到$insert()$的耗时是固定的呢?

$Insert(\ )$实现方法:

- 如果插入的结果符合结构的要求(从小到大/从大到小)，那就直接放在 binary heap 的最后就行
- 如果不符合要求的时候，先把元素插入到 array 的最后，然后把当前刚插入的元素往上交换就行(不是在 array 往前移动，是在 tree 中往上移动)

这种插入方法最坏的情况就是$O(\log(N))$，但在平均情况下耗时是$O(1)$

$Remove()$实现方法：

- 我们想把想删掉的元素和最后一个元素交换，然后删掉 array 最后的元素，最后再 down heap，也就是把交换到上面的元素往下面移动.
- 在 down heap 的过程中，我们要选择和左边还是和右边的交换，我们遵循这样的规则：选择更小的子节点交换如果这是 min heap，选择更大的子节点如果是 max heap，因为如果在 min heap 中选择更大的子节点交换，更大的子节点会成为父节点，也就是说新父节点大于子节点，也就违反了 min heap 的规则

从一个 Array 创建一个 Heap：

- 第一种方法是创建一个空 heap，然后一个一个插入
- 第二种方法更加的快一些：因为 Array 已经符合 structure property，所以直接当做我们 heap 就行，但是为了确保他是按照大小排序，所以可以从 Array 的后面依次向前把每个元素做 down heap(因为 leaf 就是最后的节点了，所以可以跳过 leaf) 这里的 down heap 是$O(N)$因为大部分 down heap 都不是跟 remove 方法中的一样，从顶端开始

Binary Min-Max Heap

这是一种特殊的 Heap：他的$level\ 0 \leq level\ 1$，$level\ 1 \geq level\ 2$，$level\ 2 \leq level\ 3$

这种 heap 的$insert()$，$remove()$ 是$O(\log(N))$，$getMin(\quad),getMax(\quad)$是$O(1)$

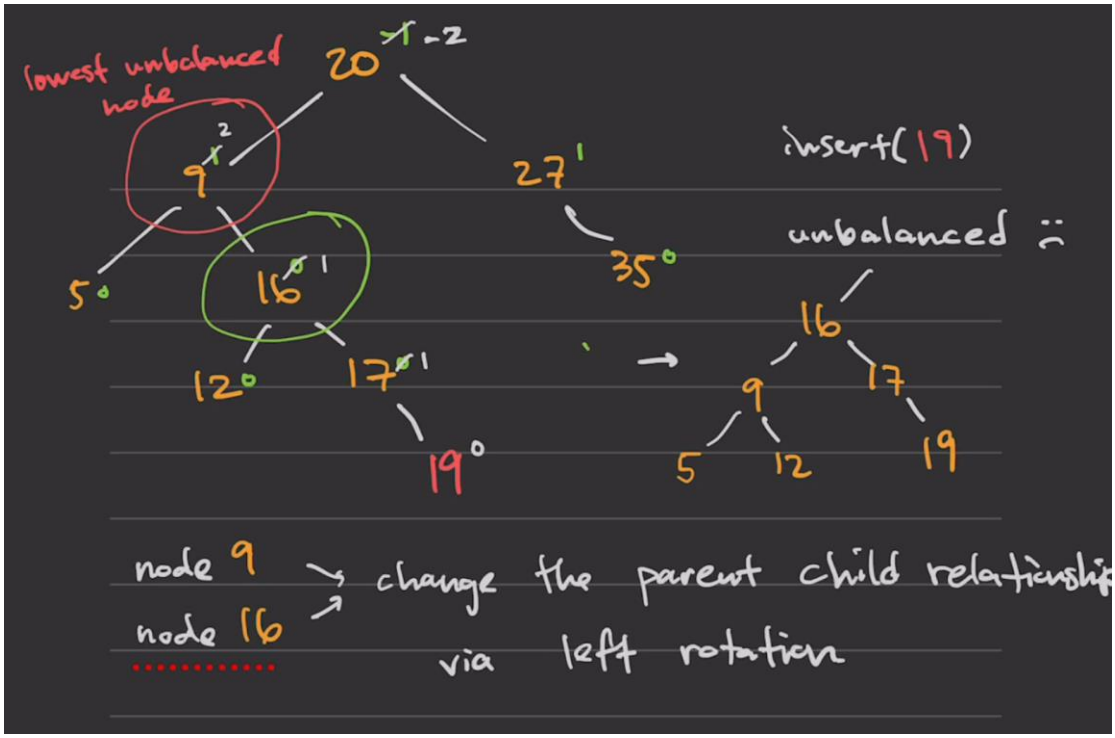
AVL Tree

这个树也叫 Adelson-Velsky and Landis Tree

这个树的特性是对于每个节点，他的左子树和右子树的高度不能相差大于 1，并且还保留 BST 特性（空的树的高度为-1，也就是所有的 leaf 的高度都是-1）

我们可以给每个节点创建 balance factor：右节点高度 - 左节点高度，并且永远保持在 −1~1之间

如果某个节点开始不平衡了，就找到 lowest unbalanced node，然后从那开始重新整理一下：



比如在我们添加 19 以后，这个树就是 unbalanced 了，我们先要找到最下面这个不平衡的节点(9)，如果修好它了，自然上面的也就平衡了．而是因为节点 9 的右子节点导致的不平衡，所以我们可以对 16 做一个 “左旋转”：以节点 16 作为轴心，做逆时针旋转, 9 变成了 16 的左子节点，因为 16 变成了父节点, 17 保持不变，夹在中间的 12 就变成了 9 的右子节点

规则：

n = 最下面的不平衡的节点

z = $n.right$ if $n.balancing\ factor == 2$, z = $n.left$ if $n.balancing\ factor == 2$ (谁的子树大就选谁)

所以就有 4 种情况：

$$z = n.right\ \&\&\ z.bf \geq 0 \rightarrow right\&right$$
$$z = n.right\ \&\&\ z.bf < 0 \rightarrow right\&left$$
$$z = n.left\ \&\&\ z.bf \geq 0 \rightarrow left\&right$$
$$z = n.left\ \&\&\ z.bf < 0 \rightarrow left\&left$$

这里的$right\&right$表示的是右边比左边的树要高，并且右子树的右子树还比右子树的左子树要高

知道了情况，就能给出对应的解决办法：

$right\&right$	左旋转
$right\&left$	右旋转再左旋转
$left\&right$	左旋转再右旋转
$left\&left$	右旋转

Associative Array

这是一种 Abstract Data Type:

这种数据结构的例子有: Map, Dictionary

这种数据结构的特点是键值对(key-value pairs)

- 每把钥匙是独一无二的

- 一个钥匙对应着一个数值
- 这个 pair 我们通常认为是一种映射：从 key 映射到 value

这个 abstract data type 预先定义的方法有：

- $insert$ / $put(key, value)$ 新增一个键值对，或者覆写键值对
- $remove$ / $delete(key)$ 移除一个键值对
- $lookup$ / $find$ / $get(key)$ 返回 key 对应的数值

Hash Table

这是一种 Associative Array / map / Dict 的实现

这种数据结构增加删除查找都会是$O(1)$

我们第一个想到的是 Array，但是如果我们输入了一个超级大的 key，这该怎么解决呢？而且要是我们想把某种 Object 当做 key 呢?

这可以用 Hash function 来解决：

Hash function 就是一个函数，我们输入进去一个 object，会得到一个整数，并且如果 Object 不变，整数就不会变.

比如一个简单的 String hash function：我们把 String 里每个字符的 ACSII 数值连接到一起

就可以得到一个 String 的 hash function.

然而 Java 里自带了一个 Hash function：我们想要获取就直接输入$object.hashCode()$就行

但是对于 Object 的$hashCode$方法跟自带的$equals()$方法一样，只是去检查内存地址关系，这就导致经常内容相同但 hash 不相同．这就需要我们覆写$toString$和$equals$方法一样，也覆写$hashCode()$方法

我们需要注意三件事:

- 一个好的$hashCode$输入一个相同的 object 需要在程序运行过程中返回的结果一致
- 如果$object1.equals(object2)$，那他们分别的哈希值应该也是相等的
- 如果两个物品不相等，他们的哈希值仍然可能相等

Hash Table with Hash Function:

- 使用$key.hashCode()$去 Array 里进行查找(并且想办法使 index 保持在合理范围并且不会有重叠的元素，如把$index\%array.length$)
- 哈希值函数算法消耗不能太高
- 我们希望每个物体都能有自己的哈希值，这样能避免冲突
- 我们希望能把哈希值 warp around，这样不会 out of bound error，并且他们的分布还是均匀的

Collisions Error

Collision Error 出现是因为两个 key 有相同的哈希值，或者是因为当我们把哈希值转换成 index 的时候，遇到了已经好的数值

解决办法:

Linear probing：当我们遇到已经存了的情况，那就往前遍历，直到找到空位置

128 d					45b			78a	18c
0	1	2	3	4	5	6	7	8	9

$insert(78,\text{a})$
$insert(45, b)$
$insert(18, c)$
$insert(128, d)$

当我们调用$get(128)$时，得到 index 是 8，跟当前位置保存的 key 比较(我们打算在 array 里不仅保存 value，还保存 key)，如果不相同就遍历向前，直到找到 index 是 0 的位置.

当要寻找不存在的 key 时，只要遍历到 null 的时候就能知道当前这个值不存在了

但问题又出现了：如果我们再插入一个 125，他会插入到 index 为 6 的位置．如果我们直接删除掉 45 的话，我们就永远找不到 125 了，因为 index 5 为 null，查找到那就会直接停止.

我们可以用 “soft” 或者 “lazy” 删除，本质就是并没有删除这个元素，而是标记为删除，遍历的时候就可以继续向前找，或者再创建一个 array 跟上面一样长，在那里标记哪个元素被删除了

Quadratic probing：我们还是往前遍历，但是增加的量不一样，如果超出范围则返回开头，

比如当前 index 有元素就加 1 去往前找，找不到就再加3，5, 7... 为了避免在加 index 的时候进入循环，我们最好选质数-prime number 作为起始容量，并且把λ严格控制在 0.5 以下

Separate Chaining：我们把 Array 中的每个元素替换成 linked list，当两个 hash code 一样，直接存在当前位置的 linked list 里就行

Load Factor

Load factor 是一种比例，表示列表里物品和总体容积的比例

$$\lambda = \frac{items\ in\ hash\ table}{capacity\ of\ hash\ table}$$

当$\lambda = 1$的时候，就有严重的问题，因为我们会陷入死循环中，并且我们也不希望它接近 1，或者接近 0．所以我们需要重新扩容当数组当λ要接近 1 的时候(Separate Chain 除外)

当我们需要扩容:

- 先扩容数组，也许扩容成以前的两倍
- 然后把所有的元素重新放进去，这样能避免元素聚在一起(如果直接把元素复制进来)