Day1

Java是面对象的，所以Java的所有代码肯定在class中

```
public class HelloWorld {   //若使用public的话文件名应该为HelloWorld.jav
    public static void main(String[] args) {
        System.out.println("Hello world!"); //输出的时候会自动换行
    }
}
```

Java中的函数

```
public class LargeDemo {
    public static int Larger (int x,int y){
        if (x > y) {
            return x;
        }
        return y;
    }

    public static void main(String[] args){
        System.out.println(larger(8,10));
    }
}
```

Java中的对象

```
public class Dog{
    public String kinds;
    public String status;
    public int old;

    public Dog(String k,int o){
        this.kinds = k;
        this.status = "Fine";
        this.old = o;
    }

    public void grow(){
        old = old + 1;
    }

    public int how_old(){
        return old;
    }

    public  void DogBark(){
        System.out.println(kinds + " Bark!");
    }
}
```

在java中定义和使用classes（关注non-static和static的区别）

```
public class Dog {

    //public int weightInpounds;
    public Dog(int w) {
        weightInpounds = w;
    }//这是更常见的做法

    public  void makeNoise() { //简单来说，如果这个方法将会被实例采用，那么i
        if (weightInpounds < 10) {
            System.out.println("yip");

        }else if (weightInpounds <30){
            System.out.println("bark");
        }else {
            System.out.println("arooo");
        }
    }

    public static maxDog (Dog d1,Dog d2){
        if(d1.weightInpounds < d2.weightInpounds){
            return d2;
        }
        return d1;
    }

    public maxDog(Dog d2){
        if(weightInpounds < d2.weightInpounds){
            return d2;
        }
        return d1;
    }
}
```

也就是说，使用makeNoise时肯定是 `somedog.makeNoise();`

对于static的方法，肯定是 `Dog.makeNoise();`

并且Non-static menbers 不能被class name invoked(Dog.makeNoise)

```
public static void main(String[] args)
```

- `String[] args` :传递给主函数的参数

Exersise

Exersise1

现在我们想给 `Dog` 设计一个 `method` :

```
public static Dog[] largerThanFourNeighbors(Dog[] dogs){
    //这个函数希望能返回比左右4个Dog都重的狗的列表

}
```

Exersise2:

Try to write a program that sums up the command line arguments, assuming they are numbers.

Answer:

```
public class SumArgs {
    public static void main(String[] args) {
        int sum =0;
        if (args.lenth == 0) {
            System.out.println("NULL");
            return;
        }else{
            for(int i=0;i<args.lenth;i++){
                sum = sum + args[i];
            }
            System.out.printlf("%d",sum);
        }
    }
}
```

java基础

Type

class	类型
int	整型
double	浮点型
String	"This is a string"
boolean	true , false
char	'a'

运算

```
int x = Math.pow(2, 10); //java的^被XOR占用了
```

String Class

```
String s = "Hello";
s += "world";
s += 5;
// 在java中你可以向String中添加任何东西，他们都会被隐式转换
int sLength = s.length();
String substr = s.substring(1,5);
char c = s.charAt(2);
if (s.indexOf("hello") != -1) {  // 检查子串"hello"是否存在于s中
    System.out.println("\"hello\" in s");  // 如果存在，则输出
}
```

Java语言练习

Day2

Java中对象实例的赋值

```
Walus a = new Walus(1000,8.3);
Walus b;
b = a;
b.weight = 5;
System.out.println(a); // a会同步改变吗
```

答案显然，因为a，b只是对对象的引用
进一步说，new Walus()只是返回对象的地址(a,b也只是地址)
而java中，除了基本的几种数据(8 primitive type)，其他都是reference type(即对象)

A-List and recursive data structure metaphor

对于List，如果已知其中存储的数据类型，可以用一个class实现类似 IntList
但是，Int其实并不是List的属性，所以我们可以用两个class实现拆分：

- Node
- SLList

> 此举在C Primer中亦有记载> 此举在C Primer中亦有记载
> 并且，如果在引用class时采用 privatA-,就可以完美实现对 Node 的封装 private Node example`

Private Class

Hide implementation details from users of your class

- easy - understanding
- safe - use
- 简单理解就是不懂不能软动的

借此可以进一步思考哪些方法是 Public 的：

- 容易理解的

- 不用调整的

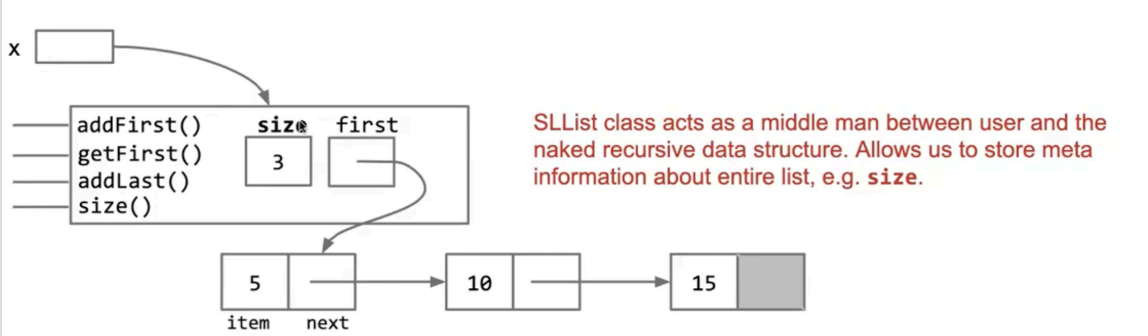
另外，对于recursive data structure

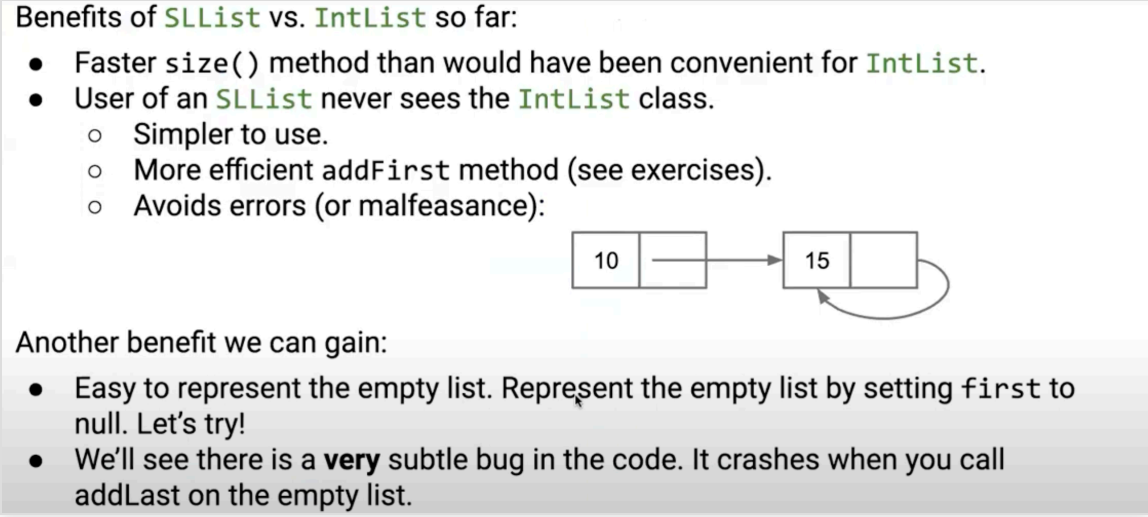
可以直接把底层class(`nested classes`)放到外层class里面(就像 `struct` 前套一样)

Static Class

注意到，当我们实现methods时应该"与class的层次匹配",比如

- create a private recursive method helper public method





另外，对于我们创建的 `SLList` ，如果List为空会怎么样呢?

其实会导致许多许多的bug

于是引入了 `sentinel node`

sentinel node

这是为了使得空list和非空list **the same** 的操作

此时对于单向链表，first node就位于sentinel node 的next

An invariant is a condition that is guaranteed to be true during code execution (assuming there are no bugs in your code).

An `SLList` with a sentinel node has at least the following invariants:

- The `sentinel` reference always points to the sentinel node.
- The first node (if it exists), is always at `sentinel.next`.
- The `size` variable is always the total number of items that have been added.

Invariants make it easier to reason about code:

- Can assume they are true to simplify code (e.g. `addLast` doesn't need to worry about nulls).
- Must ensure that methods preserve invariants.

D-List and Node类型

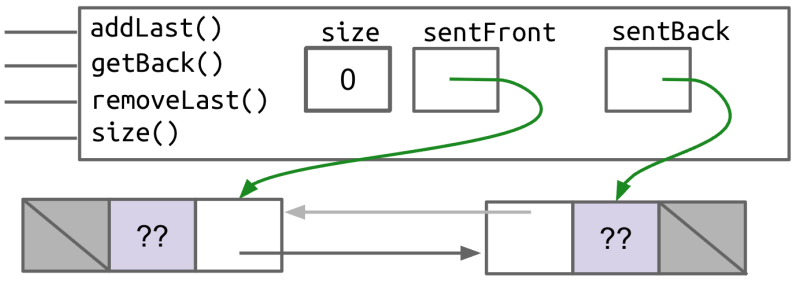
注意到，ALList中的remove操作，似乎还是必须要遍历(要得到前一个指针)，特别是在移除最后一个元素的，这可能会与我们使用体验相悖。

首先想到的是双向链表，但是怎么处理边界呢

- Try1:在最后引入第二 `sentinel node`

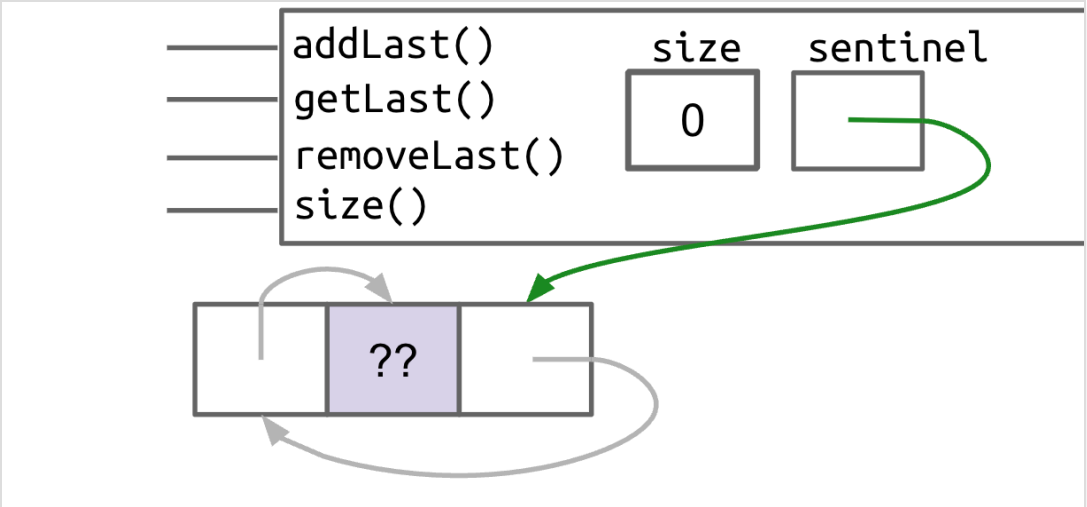
  但是这样会有一个很大的问题，对于最后一个node，其仍然是特殊的(指向null)

One solution: Have two sentinels.



- Try2:循环链表

  只需要一个sentinel在中间就好了



注意到，很多时候Node的种类并非一成不变

```
public class SLList<Pineapple> {
    private class Node {
        public Pineapple item;
        public Node next;

        public Node(Pineapple f, Node r){
            item = f;
            next = r;
        }
    }

}

//此时使用的时候就可以自定义了

SLList<Integer> L1 = new SLList<Integer>();
SLList<String> L2 = new SLList<String>();
//注意需要的是Object形式
```

Array

对于Array

- 用A[i],访问元素
- 固定的长度，一致的类型，连续存储

`System.arraycopy(b,0,x,3,2);`

- source array
- start
- Target array
- Start
- Nunber to copy

`array.length`

`Arrays.toString(array)`

Force Loop:

```
int[] array = {1,2,3};
for(int i : array){
    System.out.println(i);
}
```

List(ArrayList)

```
List<String> lst = new ArrayList<>();
lst.add("zero");
lst.add("one");
lst.set(0, "zed");
System.out.println(lst.get(0));
System.out.println(lst.size());
if (lst.contains("one")) {
    System.out.println("one in lst");
}
for (String elem : lst) {
    System.out.println(elem);
}
```

Set(HashSet)

```
Set<Integer> set = new HashSet<>();
set.add(1);
set.add(1);
set.add(2);
set.remove(2);
System.out.println(set.size());
if (set.contains(1)) {
    System.out.println("1 in set");
}
for (int elem : set) {
    System.out.println(elem);
}
```

Map(HashMap)

```
Map<String, String> map = new HashMap<>();
map.put("hello", "hi");
map.put("hello", "goodbye");
System.out.println(map.get("hello"));
System.out.println(map.size());
if (map.containsKey("hello")) {
    System.out.println("\"hello\" in map");
}
for (String key : map.keySet()) {
    System.out.println(key);
}
```

Day3

Testing

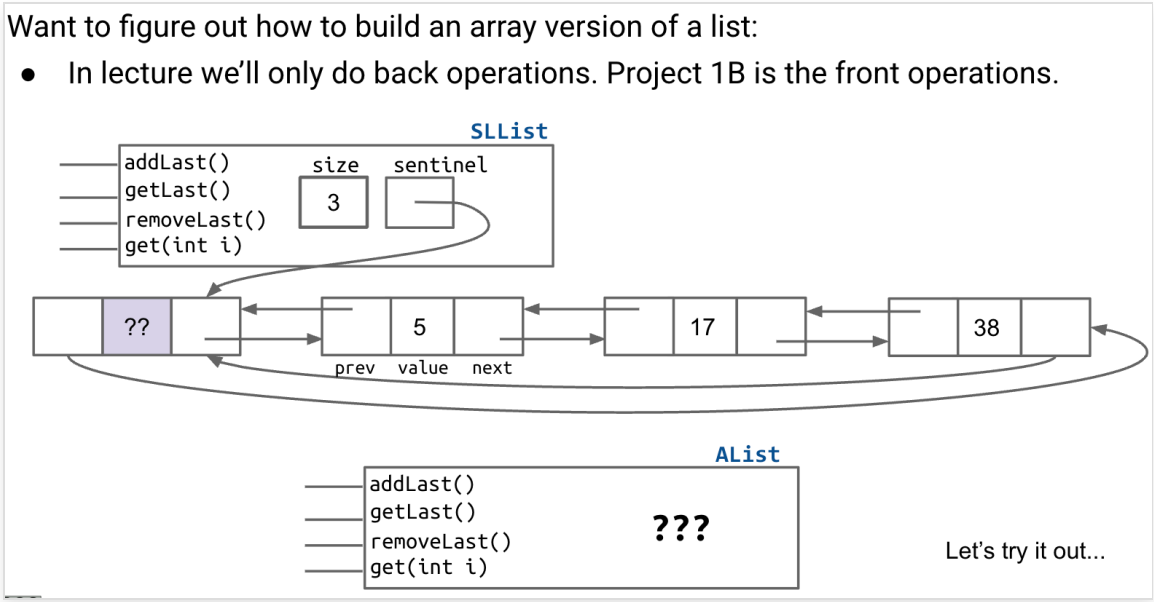
```
import static com.google.common.truth.Truth.assertThat; //

public class Test{

    public void testamethod{

        @Test
        //input
        //expected
        //actuall
        assertThat(actual).isEqualTo(expected);
    }
}
```

Array and List

当我们尝试使用D-List的get方法时，D-List需要遍历与访问

而又想到，Array的访问是迅速的

但是Array的内存是有限的，我们应该怎么拓展呢?

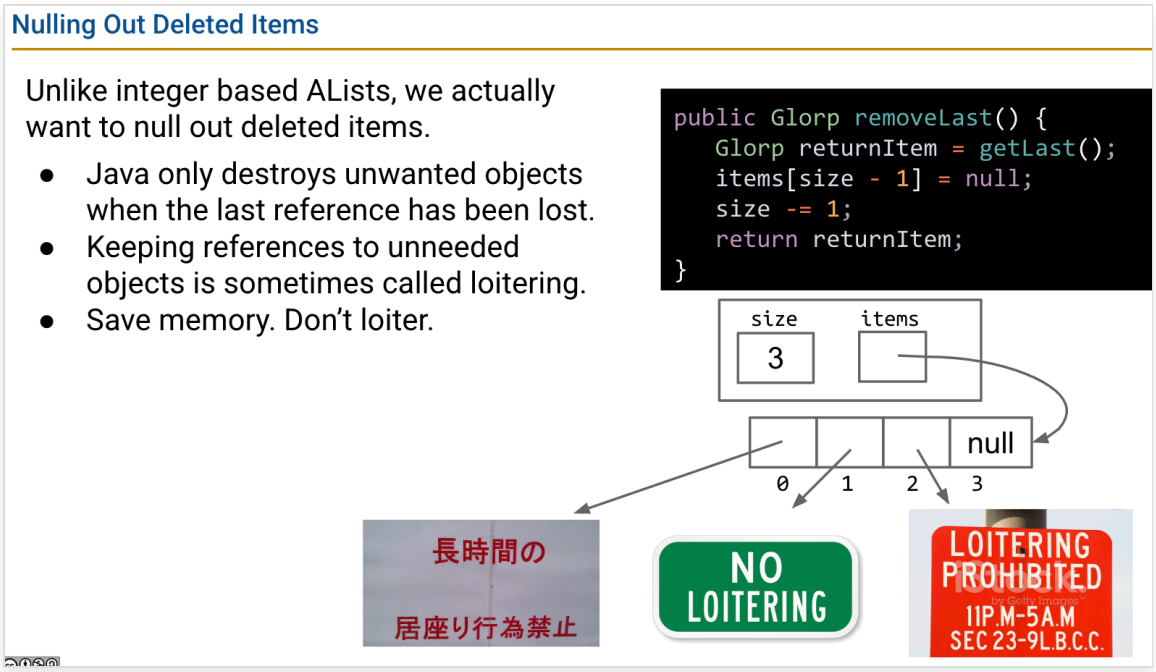


答案是 Array Resizing

此外，还需要注意到：

```
public class AList<Glorp>{
    private Glorp[] items; //没问题
    items = (Glorp[]) new Object[8]; //这个不能使用 items = Glorp[8];
}
```

此外还有必要 null out deleted items



Inheritance_1

注意到，对于已有的AList和SLList，我们已经实现了相同的方法

但是我们如果在这些方法的基础上构建新方法：

- 我们同样需要给所有class写一遍，it is gross.
- 我们将要不得不debugg多遍

就像这张幻灯片一样：

Hypernyms

In natural languages (English, Spanish, Chinese, Tagalog, etc.), we have a concept known as a "hypernym" to deal with this problem.

- Dog is a "hypernym" of poodle, malamute, yorkie, etc.

Washing your poodle: 1. Brush your poodle before a bath. ... 2. Use lukewarm water. ... 3. Talk to your poodle in a calm voice. ... 4. Use poodle shampoo. ... 5. Rinse well. ... 6. Air-dry. ... 7. Reward your poodle.	Washing your malamute: 1. Brush your malamute before a bath. ... 2. Use lukewarm water. ... 3. Talk to your malamute in a calm voice. ... 4. Use malamute shampoo. ... 5. Rinse well. ... 6. Air-dry. ... 7. Reward your malamute.

里面的poodle、malamute可以是任意一种dog

- dog : Hypernym of poodle,malamute...
- malamute : Hyponym of dog

所以：List is a Hypernym of SLList and AList

在java中用两步来实现

- Define a reference type for hypernym

```
public interface List61B<Blorp>{
}
```

interface: 声明 List61B有哪些可以做的

- specify SLList and AList are hyponyms of that type.

```
public class SLList<Blorp> implements List61B<Blorp>{
    ...
}

public class AList<Blorp> implements List61B<Blorp>{
    ...
}
```

现在可以在List61B的类实现：

```
public static String longest(List61B<String> List){

}
```

需要注意的是：这里的 List可以是任何一个指向属于List61B的subclass的指针

Question

Will the code below compile? If so, what happens when it runs?

a. Will not compile.

b. Will compile, but will cause an error at runtime on the **new** line.

c. When it runs, an **`SLList`** is created and its address is stored in the **`someList`** variable, but it crashes on **`someList.addFirst()`** since the **`List`** interface doesn't implement **`addFirst`**.

d. **When it runs, an `SLList` is created and its address is stored in the `someList` variable. Then the string "elk" is inserted into the `SLList` referred to by `addFirst`.**

```
public static void main(String[] args) {
   List61B<String> someList = new SLList<String>();
   someList.addFirst("elk");
}
```

方法重写 vs. 方法重载

- overrides:subclass和superclass有相同的method signature和name
- overloaded:subclass和superclass仅有相同的signature

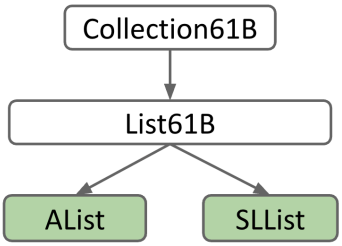
我们会给overrides添加 `@override` tags

interface Inheritance

Interface Inheritance

Specifying the capabilities of a subclass using the **implements** keyword is known as **interface inheritance**.

- Interface: The list of all method signatures.
- Inheritance: The subclass "inherits" the interface.
- Specifies what the subclass can do, but not how.
- Subclasses must override all of these methods!
- Such relationships can be multi-generational.
  - Figure: Interfaces in white, classes in green.
  - We'll talk about this in a later lecture.



Interface inheritance is a powerful tool for generalizing code.

- `WordUtils.longest` works on SLLists, ALists, and even lists that have not yet been invented!

implementation inheritance

```
public interface List61B<Item> {

    default public void print(){
        for (int i=0; i< size(); i +=1){
            System.out.print(get(i) + " " );

        }
    }
    //可以在interface中创建函数
    //这个函数只能利用List61B中声明的函数实现
}
```

这个方法对于Alist是十分高效的，但是对于SLList，这是很低效的

所以我们可以在SLList中创建自己的快速的print方法

```
@Overrride
public voif print(){
    ...
}
```

此时如果对于SLList类型，使用print(),将会使用这个print()函数

Static Type vs. Dynamic Type

Static Type vs. Dynamic Type

Every variable in Java has a “compile-time type”, a.k.a. “static type”.

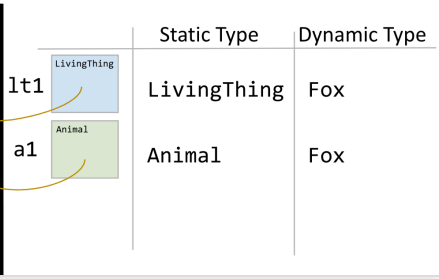
- This is the type specified at **declaration**. Never changes!

Variables also have a “run-time type”, a.k.a. “dynamic type”.

- This is the type specified at **instantiation** (e.g. when using `new`).
- Equal to the type of the object being pointed at.

```
public static void main(String[] args) {
   LivingThing lt1;
   lt1 = new Fox();
   Animal a1 = lt1;
   Fox h1 = new Fox();
   lt1 = new Squid();
}
```

Technically requires a “cast”. See next lecture.



Dynamic Method Selection For Overridden Methods

Suppose we call a method of an object using a variable with:

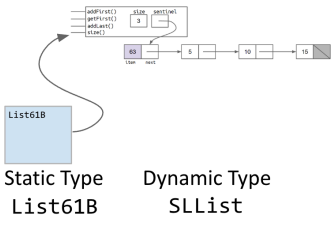
- compile-time type X
- run-time type Y

Then if Y **overrides** the method, Y’s method is used instead.

- This is known as “dynamic method selection”.

This term is a bit obscure.

```
public static void main(String[] args) {
   LivingThing lt1;
   lt1 = new Fox();
   Animal a1 = lt1;
   Fox h1 = new Fox();
   lt1 = new Squid();
}
```



Day4

做完了project0

试着直接莽但是失败了

里面处理方向的方法很有启发性，以及分割问题的刀法十分精湛（bushi

Inheritance_2

Extends KeyWord

```
public class RotatingSLList<Item> extends SLList<Item> {

}
```

`extends` 和 `implements` 在使用上似乎一样，只是当SLList是 `class` ，我们采用 `extends` ，当SLList是 `interface` ,我们采用 `implements`

Because of **extends**, `RotatingSLList` inherits all members of `SLList`:

- All instance and static variables.
- All methods.
- All nested classes.

… but members may be private and thus inaccessible! More later.

Constructors are not inherited.

注意到，**java中的 `private` 关键字使得subclass也无法访问**

那我们怎么override superclass中的method呢?

答案是 `super`

Because of **extends**, `RotatingSLList` inherits all members of `SLList`:

- All instance and static variables.
- All methods.
- All nested classes.

… but members may be private and thus inaccessible! More later.

Constructors are not inherited.

此外由于java的extends不会继承construcor，所以当我们在构建RotatingSLList的constructor时最好还是加入

```
super(input ……);
```

Implementaton Inheritance plus and Object methods

Implementation Inheritance Breaks Encapsulation

What would vd.barkMany(3) output?

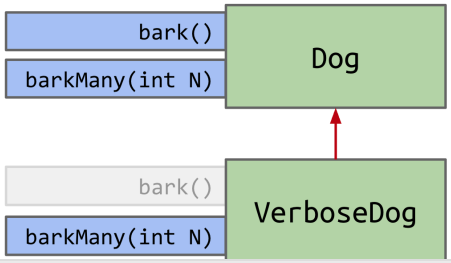
c. Something else.

- Gets caught in an infinite loop!

(assuming vd is a Verbose Dog)

Dog.java

```
public void bark() {
   barkMany(1);
}
public void barkMany(int N) {
   for (int i = 0; i < N; i += 1) {
       System.out.println("bark");
   }
}
```



VerboseDog.java

```
@Override
public void barkMany(int N) {
   System.out.println("As a dog, I say: ");
   for (int i = 0; i < N; i += 1) {
       bark();   ← calls inherited bark method
   }
}
```

Compiletime Type Checking and casting

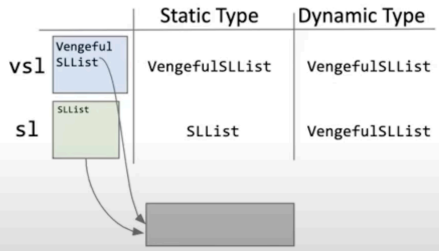
[61B SP24] Lecture 9 - Inheritance II: Extends, Casting, Higher Order Functions

Compile-Time Type Checking

Also called static type.

Compiler allows method calls based on **compile-time** type of variable.

- sl's runtime type: VengefulSLList.
- But cannot call printLostItems.

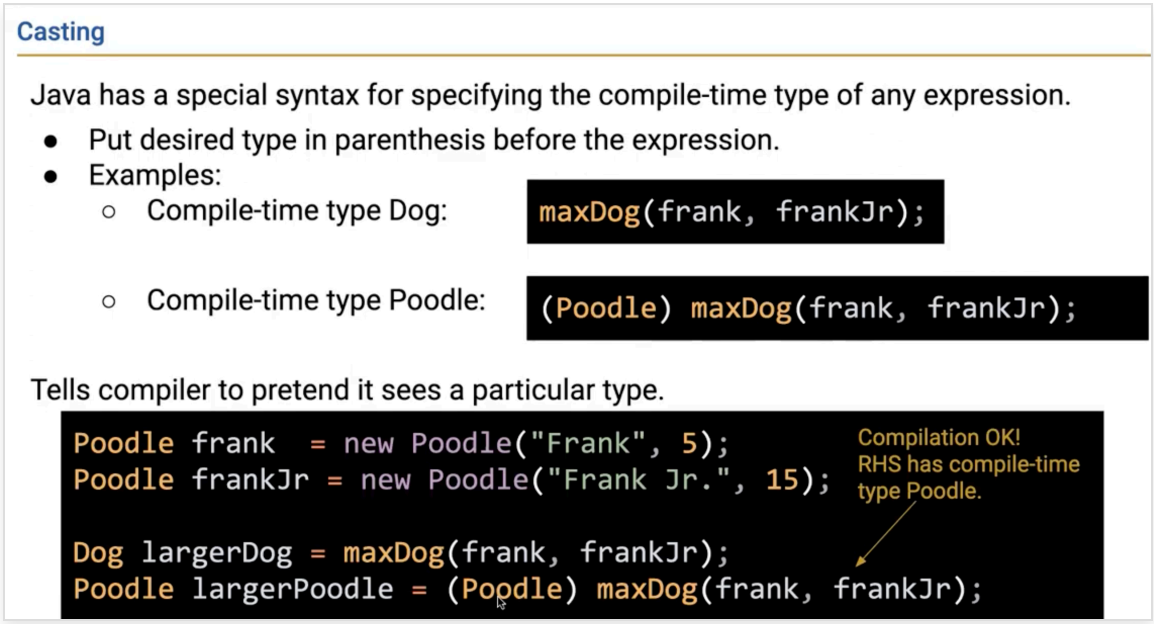



```
public static void main(String[] args) {
   VengefulSLList<Integer> vsl =
           new VengefulSLList<Integer>(9);
   SLList<Integer> sl = vsl;

   sl.addLast(50);
   sl.removeLast();

   sl.printLostItems();
   VengefulSLList<Integer> vsl2 = sl;
}
```

Compilation errors!

Compiler also allows assignments based on compile-time types.

- Even though sl's runtime-type is VengefulSLList, cannot assign to vsl2.
- Compiler plays it as safe as possible with type checking.



此外：

对于被重写的方法(override)，实际采用的方法是变量的 **动态类型**

Higher-Order Function

对于java7及以下的版本不支持传入函数指针

但是可以传入 `interface`

Day5

Subtype polymorphism(子类型多态)

用object的子类和java的Dynamic type机制实现的对于多态输入的处理方法

built-in interface

在利用子类型多态时，如果superclass采用的是java内建interface将会简单非常多

Comparator and Comparable

```
import java.util.Comparator;
public class Dog implements Comparable<Dog> {
   private String name;
   private int size;

   /** Returns <0 if this dog is less than the dog pointed at by o, and so forth. */
   public int compareTo(Dog uddaDog) {
       return this.size - uddaDog.size;
   }

   private static class NameComparator implements Comparator<Dog> {
       public int compare(Dog a, Dog b) {
           return a.name.compareTo(b.name);
       }
   }

   public static Comparator<Dog> getNameComparator() {
       return new NameComparator();
   }
}
```

注意到这个comparator是一个private static class，那我们该如何使用呢

```java
public class DogLauncher {
   public static void main(String[] args) {
       Dog d1 = new Dog("Elyse", 3);
       Dog d2 = new Dog("Sture", 9);
       Dog d3 = new Dog("Benjamin", 15);
       Dog[] dogs = new Dog[]{d1, d2, d3};



       Dog.NameComparator nc = new Dog.NameComparator();
       if (nc.compare(d1, d3) > 0) { // if d1 comes later than d3 in the alphabet
           d1.bark();
       } else {
           d3.bark();
       }

   }
}
```

怎样支持 `Enhanced loop`

先看一个等价语法糖

An alternate, uglier way to iterate through a Set is to use the `iterator()` method.

Set.java:

```java
public Iterator<E> iterator();
```

Suppose we have a **`Set<Integer>`** called **`javaset`**.

- In that case, we can iterate with either of the two equivalent pieces of code.
- Left code is shorthand for right code.

```java
for (int x : javaset) {
   System.out.println(x);
}
```

"Nice" iteration.

```java
Iterator<Integer> seer
   = javaset.iterator();

while (seer.hasNext()) {
   int x = seer.next();
   System.out.println(x);
}
```

所以我们应该着手设计

- iterator()
  - hasNext
  - next

```java
public class ArraySet<T> {
   /** returns an iterator (a.k.a. seer) into ME */
   public Iterator<T> iterator() {
       return new ArraySetIterator();
   }

   private class ArraySetIterator implements Iterator<T> {
       private int wizPos;

       public ArraySetIterator() {
           wizPos = 0;
       }

       public boolean hasNext() {
           return wizPos < size;
       }

       public T next() {
           T returnItem = items[wizPos];
           wizPos += 1;
           return returnItem;
       }
   }
```

最后，还要告诉java： `ArraySet<T>` 一定有iterator()

此时需要说明：

```java
public class ArraySet<T> implements Iteratable<T> {

}
```

To support the enhanced for loop:

- Add an `iterator()` method to your class that returns an `Iterator<T>`.
- The `Iterator<T>` returned should have a useful `hasNext()` and `next()` method.
- Add `implements Iterable<T>` to the line defining your class.

`toString`

弄清楚java的 `.toString` 到底在干什么：

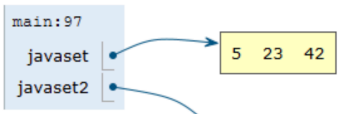
在某些时候可以override

== .vs .equals

As mentioned in an offhand manner previously, == and .equals() behave differently.

- == compares the bits. For references, == means "referencing the same object."

```
Set<Integer> javaset = Set.of(5, 23, 42);
Set<Integer> javaset2 = Set.of(5, 23, 42);
System.out.println(javaset.equals(javaset2));
```




```
$ java EqualsDemo
True
```

To test equality in the sense we usually mean it, use:

- `.equals` for classes. Requires writing a `.equals` method for your classes.
  - Default implementation of `.equals` uses == (probably not what you want).
- BTW: Use `Arrays.equal` or `Arrays.deepEquals` for arrays.

所以很多时候我们将会override `.equals`

但是有时原函数的类型会是superclass，此时需要新的关键字

instanceOf Demo

The `instanceof` keyword is very powerful in Java.

- Checks to see if o's dynamic type is Dog (or one of its subtypes).
  If no, returns false.
- If yes, returns true *and* casts o into a variable of static type Dog called uddaDog.
- Works correctly, even if o is `null`.

```
@Override
public boolean equals(Object o) {
   if (o instanceof Dog uddaDog) {
       return this.size == uddaDog.size;
   }
   return false;
}
```

.of

有的时候还可以override `.of` method

Day6

Asymptotics_1

有些简化规则：

1. 只考虑最坏情况
2. 忽视小量级项
3. 忽视系数
4. 所有操作花费相同的constant时间

此外是一些数学符号

$R(N) \in \Theta(f(N))$

means there exist positive constants k_1 and k_2 such that:

$k_1 \cdot f(N) \leq R(N) \leq k_2 \cdot R(N)$

Disjoint Set

```
public interface Disjoint<T>{
    // 连接两个
    void connect(T x,T y);

    // 判断两个是否相连
    // 不需要直接相连
    boolean isConnected(T x, T y);
}(
```

Ideas:list of integers where ith entry gives set number (a.k.a. “id”) of item i.

- connect(p, q): Change entries that equal id[p] to id[q]

  注意到这种方式构造出来的树为

Implementation	constructor	connect	isConnected
ListOfSetsDS	Θ(N)	O(N)	O(N)
QuickFindDS	Θ(N)	Θ(N)	Θ(1)

让我们换个思路：

Idea: Assign each item a parent (instead of an id). Results in a tree-like shape.

- An innocuous sounding, seemingly arbitrary solution.
- Unlocks a pretty amazing universe of math that we won't discuss.

很容易知道这种方式构造出来的树的所有method的时间复杂度都在O(N);

But why?

显然是极端情况导致的

所以我们可以采用 `Weighted Union Tree` 的方式避免极端情况出现

Minimal changes needed:

- Use parent[] array as before.
- isConnected(int p, int q) requires no changes.
- connect(int p, int q) needs to somehow keep track of **sizes**.
- Replace -1 with **-weight** for roots (top approach).

可以证明：

QuickUnion's runtimes are O(H), and WeightedQuickUnionDS height is given by H = O(log N). Therefore connect and isConnected are both O(log N).

> 至于为什么用Weights insteads of Heights：
>
> - Worst case performance for HeightedQuickUnionDS is asymptotically the same! Both are Θ(log(N)).
> - Resulting code is more complicated with no performance gain.

Day7

Amortized Runtime

- Any single operation may take longer, but if we use it over many operations, we're guaranteed to have a better average performance
- So amortized runtime gives a better estimate of how much time it takes to use something in practice

- Disjoint sets also used amortized runtime; WQU with path compression still has Θ(log(n)) runtime in the worst case, but Θ(α(n)) amortized runtime.

WQU(WeightedQuickUnionWithPathCompressionDS):point all node to the bigger tree root when `isconnect()` or `connect()`
这个想法代码很简单，但是证明不是很显然

ADTs,BSTs

An Abstract Data Type (ADT) is defined only by its operations, not by its implementation.
BST:Binary Search Tree
ramdomly insert into ADT will make tree bushy,**BUT:**
有时候加入的数据是单调的，比如时间:tree will be spindly
最初的想法是用一个node来存储最大的一系列数，
但是这会导致这个node过大，进一步导致 `search` ,等操作十分困难
Solution?
限制node的数量，如果过多，把其中一个递交给father node
进一步的，可以重新分割这个father node的子节点:Bigger , Middle , Right
当father节点node满了，会形成新的father节点，并且可以触发链式反应
这个完整的Tree 被命名为B-Tree:

- 所有的叶子节点距离root相同(这个树是从下往上构建的)
- 一个非叶子节点有k个items，一定有k+1个son节点

B-Trees are a modification of the binary search tree that avoids Θ(N) worst case.
- Nodes may contain between 1 and L items.
- contains works almost exactly like a normal BST.
- add works by adding items to existing leaf nodes.
  - If nodes are too full, they split.
- Resulting tree has perfect balance. Runtime for operations is O(log N).
- Have not discussed deletion. See extra slides if you're curious.
- Have not discussed how splitting works if L > 3 (see some other class).
- B-trees are more complex, but they can efficiently handle ANY insertion order.

但是B-Tree太难implement了

Red-Black Tree

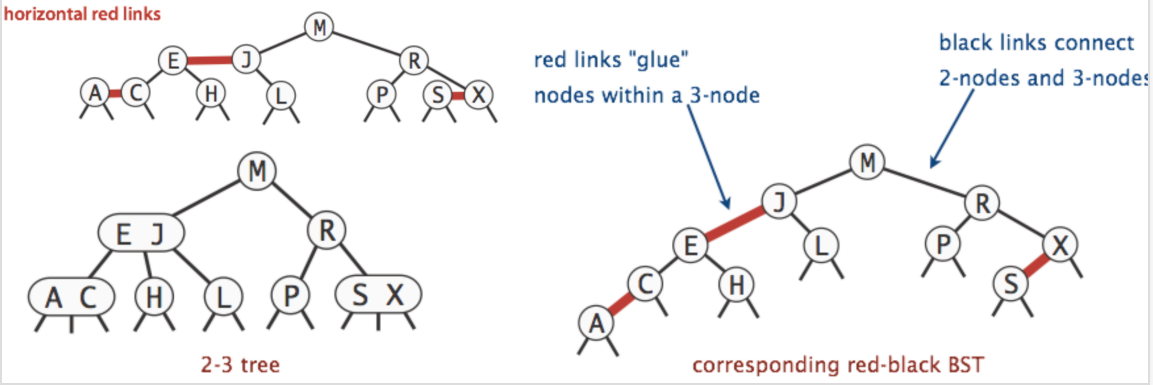
Rotate

- rotateLeft(G): Let x be the right child of G. Make G the new left child of x.
  - Can think of as temporarily merging G and P, then sending G down and left.
  - Preserves search tree property. No change to semantics of tree.
- rotateRight(P): Let x be the left child of P. Make P the new right child of x.
  - Can think of as temporarily merging G and P, then sending P down and right.
  - Note: k was G's right child. Now it is P's left child.

LLRBT(Left Learning Red-Black Tree)

A BST with left glue links that represents a 2-3 tree is often called a "Left Leaning Red Black Binary Search Tree" or LLRB.

- LLRBs are normal BSTs!
- There is a 1-1 correspondence between an LLRB and an equivalent 2-3 tree.
- The red is just a convenient fiction. Red links don't "do" anything special.



$Lemma$:

$H_{LLRBT} \leq 2 * H_{B-Tree}$

Lab

Lab似乎只要求编写BST，其中还有一个Asymptotics Problems

TWO key points:

- build the tree()
- traverse the tree

Day8

Hash

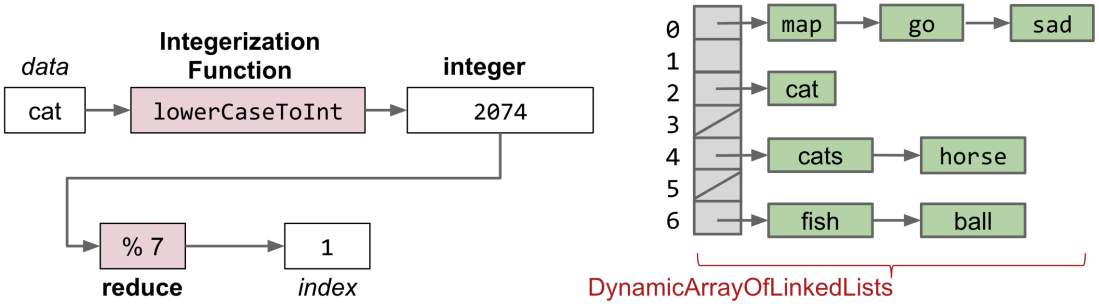
LLRBT 的算法仍然可以改进：

- 怎么取消comparable的限制
- O(n)还能更少吗

String gets to decide how it wants to be categorized,

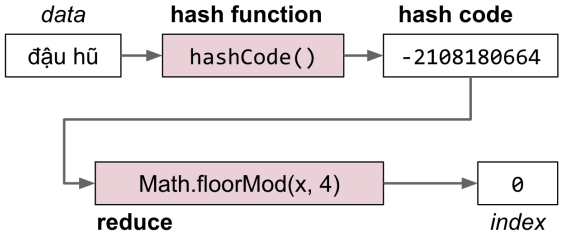
Set gets to decide when it wants to resize.

- *Data* is converted by a **integerization function** into an integer representation.
- The **integer** is then **reduced** to a valid *index*, usually using the modulus operator, e.g. 2348762878 % 10 = 8.



Hash tables:

- *Data* is converted into a hash code.
- The **hash code** is then **reduced** to a valid *index*.
- *Data* is then stored in a bucket corresponding to that *index*.
- Resize when load factor N/M exceeds some constant.
- If items are spread out nicely, you get Θ(1) average runtime.



	contains(x)	add(x)
Bushy BSTs	Θ(log N)	Θ(log N)
Separate Chaining Hash Table With No Resizing	Θ(N)	Θ(N)
… With Resizing	Θ(1)†	Θ(1)*†

*: Amortized.
†: Assuming items are evenly spread.

Integer Overflow and Hashcode

Define a method f (in String) to convert Strings into an int,

and store String s in the bin corresponding to f(s).

注意到：the default hashCode achieves good spread

但是我们通常还是会选择自定义一个 `hash function`

因为我们希望在HashSet/map/...中 `equal` ≡ `they have the same Hashcode` ，否则函数可能根据 `old hash` 来分类，但是根据新的 `equal` 来判断是否要添加新的node

(但是Hash相同时，equal可以不同)

Immutable Data Types

An immutable data type is one for which an instance cannot change in any observable way after instantiation.

The `final` keyword will help the compiler ensure immutability.

- final variable means you may assign a value once (either in constructor of class or in initializer), but after it can never change.
- Final is neither sufficient nor necessary for a class to be immutable.

Immutable: an instance cannot change in any observable way after instantiation.

```
public class Pebble {
  public int weight;
  public Pebble() {
    weight = 1;
  }
}
```

```
public class Rock {
  public final int weight;
  public Rock (int w) {
    weight = w;
  }
}
```

```
public class RocksBox {
  public final Rock[] rocks;
  public RocksBox (Rock[] rox) {
    rocks = rox;
  }
}
```

```
public class SecretRocksBox {
  private Rock[] rocks;
  public SecretRocksBox(Rock[] rox) {
    rocks = rox;
  }
}
```

Bottom line: **Never mutate an Object being used as a key.**

Heap and Priority Queue

```
//Priority Queue
public interface MinPQ<Item>{

    public void add(Item x);

    public Item getSmallest();

    public Item removeSmallest();

    public int size();
}
```

这个ADS通常用于记录最大/最小的n个数

	Ordered Array	Bushy BST	Hash Table	Heap
add	Θ(N)	Θ(log N)	Θ(1)	
getSmallest	Θ(1)	Θ(log N)	Θ(N)	
removeSmallest	Θ(N)	Θ(log N)	Θ(N)	
Caveats		Dups tough		

Binary min-heap:完全二叉树 obeys: min-heap property

min-heap:Every node is less than or equal to both of its child

对于Binary min-heap:找到最小值是最简单的

how to ADD?

1. obey: complete binary tree
2. find its location(with swaping)

how to Delete the min?

1. swap the last with the root
2. swap with the smaller son

But how we represent a Tree in this situation?

1. With Node
2. Use a Array(more suitable)

> Array always the best way to represent the **Complete Binary Tree With a fixed length**
>
> 1. Offset everything by 1 spot.Same as 3, but leave spot 0 empty.
> 2. Makes computation of children/parents “nicer”.
>
> leftChild(k) = k*2*
>
> *rightChild(k)* = *k*2 + 1
>
> parent(k) = k/2

a summury

Name	Storage Operation(s)	Primary Retrieval Operation	Retrieve By:
List	`add(key)` `insert(key, index)`	`get(index)`	index
Map	`put(key, value)`	`get(key)`	key identity
Set	`add(key)`	`containsKey(key)`	key identity
PQ	`add(key)`	`getSmallest()`	key order (a.k.a. key size)
Disjoint Sets	`connect(int1, int2)`	`isConnected(int1, int2)`	two int values

Tree traversal

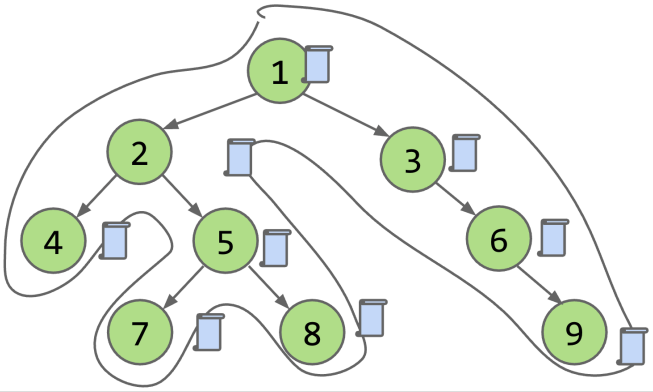
Depth First Traversals

- 3 types: Preorder, Inorder, Postorder
  - That means visit which node first.
  - Preorder traversal: “Visit” a node, then traverse its children: DBACFEG
  - Inorder traversal: Traverse left child, visit, traverse right child: ABCDEFG
  - Postorder traversal: Traverse left, traverse right, then visit: ACBEGFD

- Preorder traversal: We trace a path around the graph, from the top going counter-clockwise. “Visit” every time we pass the LEFT of a node.
- Inorder traversal: “Visit” when you cross the bottom of a node.
- Postorder traversal: “Visit” when you cross the right a node.

Example: Post-Order Traversal

- 4 7 8 5 2 9 6 3 1

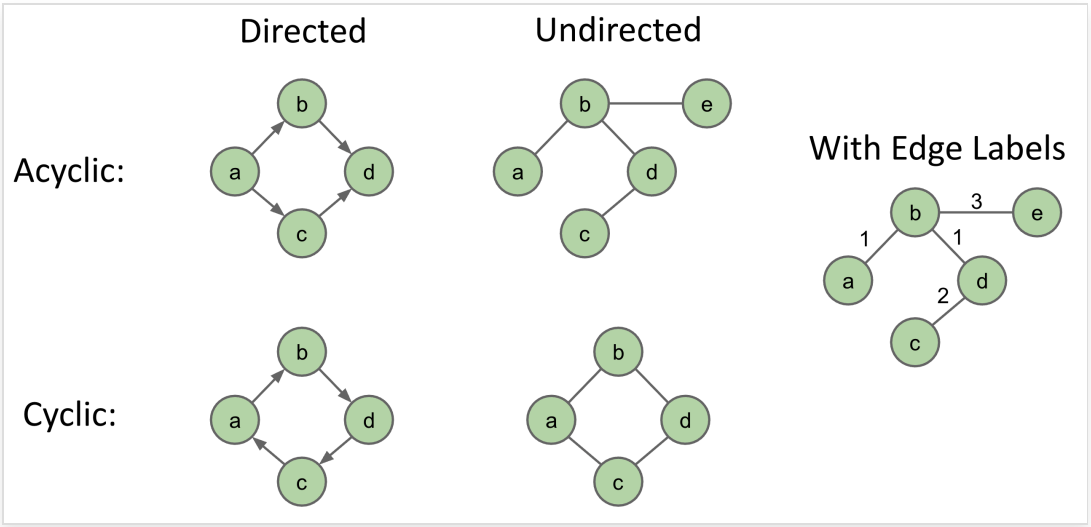


- Basic (rough) idea: Traverse “deep nodes” (e.g. A) before shallow ones (e.g. F).
- Note: Traversing a node is different than “visiting” a node. See next slide.

Graph

A simple graph is a graph with:

- No edges that connect a vertex to itself, i.e. no “loops”.
- No two edges that connect the same vertices, i.e. no “parallel edges”.



所以我们可以简单建模

Graph:

- Set of vertices, a.k.a. nodes.
- Set of edges: Pairs of vertices.
- Vertices with an edge between are adjacent.
  - Optional: Vertices or edges may have labels (or weights).
- A path is a sequence of vertices connected by edges.
  - A simple path is a path without repeated vertices.
  - A cycle is a path whose first and last vertices are the same.
  - A graph with a cycle is ‘cyclic’.
  - Two vertices are connected if there is a path between them. If all vertices are connected, we say the graph is connected.

s-t Connectivity

connected(s, t):

- Mark s.
- Does s == t? If so, return true.
- Otherwise, if connected(v, t) for any unmarked neighbor v of s, return true.
- Return false.

> 这是一种深度优先搜索，详见下

Day9

Graph API

The Graph API from Princeton’s algorithms textbook.

```
public class Graph {
  public Graph(int V):                Create empty graph with v vertic
  public void addEdge(int v, int w): add an edge v-w
  Iterable<Integer> adj(int v):      vertices adjacent to v
  int V():                           number of vertices
  int E():                           number of edges
...
```

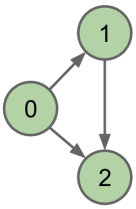
Common design pattern in graph algorithms:

- Decouple type from processing algorithm.
- Create a graph object.
- Pass the graph to a graph-processing method (or constructor) in a client class.
- Query the client class for information.

Adjacency Matrix

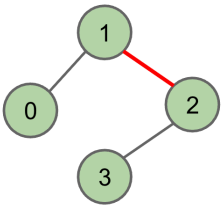
Graph Representation 1: Adjacency Matrix.

s \ t	0	1	2
0	0	1	1
1	0	0	1
2	0	0	0



For undirected graph: Each edge is represented twice in the matrix. Simplicity at the expense of space.

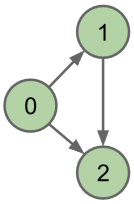
v \ w	0	1	2	3
0	0	1	0	0
1	1	0	**1**	0
2	0	**1**	0	1
3	0	0	1	0

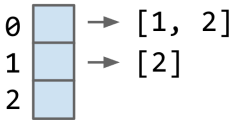


Adjacency list

Representation 3: Adjacency lists.

- Common approach: Maintain array of lists indexed by vertex number.
- Most popular approach for representing graphs.
  - Efficient when graphs are “sparse” (not too many edges).





这个数据结构很省空间，其时间复杂度怎么样?

What is the order of growth of the running time of the print client if the graph uses an ***adjacency-list*** representation, where V is the number of vertices, and E is the total number of edges?

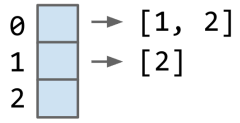
A. Θ(V)
B. Θ(V + E)
C. $\Theta(V^2)$
D. Θ(V*E)

```
for (int v = 0; v < G.V(); v += 1) {
    for (int w : G.adj(v)) {
        System.out.println(v + "-" + w);
    }
}
```

Best case: Θ(V) Worst case: $\Theta(V^2)$

Cost model in this analysis is the sum of:
- v +=1 operations
- println calls

All cases: Θ(V + E)
- v+=1 happens V times.
- Print happens 2E times.



这种思考方式在graph中十分常见：

Give a O bound for the runtime for the DepthFirstPaths constructor.

```
public class DepthFirstPaths {
  private boolean[] marked;
  private int[] edgeTo;
  private int s;
  public DepthFirstPaths(Graph G, int s) {
      ...
      dfs(G, s);
  }
  private void dfs(Graph G, int v) {
    marked[v] = true;
    for (int w : G.adj(v)) {
      if (!marked[w]) {
        edgeTo[w] = v;
        dfs(G, w);
      }
    }
  } ...
}
```

Assume graph uses adjacency list!

O(V + E)
- Each vertex is visited at most once (O(V)).
- Each edge is considered at most twice (O(E)).

vertex visits (no more than V calls)

edge considerations, each constant time (no more than 2E calls)

Cost model in analysis above is the sum of:
- Number of dfs calls.
- marked[w] checks.

Graph Traverse

由于每个节点一般只遍历一遍，所以当Traverse Graph时其实做了两件事：

1. 构建了一个Tree
2. 遍历

DFS in Graph

dfs(v):

- Mark v.
- For each unmarked adjacent vertex w:
  - if null return;
  - set edgeTo[w] = v.
  - place1
  - dfs(w)
  - place2

如果你选择在place1对w进行操作(比如说 `print()`),可以视为一种前序遍历
如果你选择在place2对w进行操作，可以视为一种后序遍历
此外，由于我们只能记录通往当前节点的上一个节点，所以edgeTo[w]在place1

BFS in Graph

bfs(v):

- initialize a array,addLast(v),mark v;
- while(array!=null)
  - i = removeFirst()
  - For each unmarked adjacent vertex w
    - mark w;addLast(w)
    - set edgeTo[w] = v ...operations

注意到这种遍历方式可以记录当前点到root的距离/深度
若以Adjacency List为Graph API的基础，有

Problem	Problem Description	Solution	Efficiency (adj. list)
s-t paths	Find a path from s to every reachable vertex.	DepthFirstPaths.java Demo	O(V+E) time Θ(V) space
s-t shortest paths	Find a shortest path from s to every reachable vertex.	BreadthFirstPaths.java Demo	O(V+E) time Θ(V) space

Space Efficiency. Is one more efficient than the other?

- DFS is worse for spindly graphs.
  - Call stack gets very deep.

- Computer needs Θ(V) memory to remember recursive calls (see CS61C).

- BFS is worse for absurdly “bushy” graphs.
  - Queue gets very large. In worst case, queue will require Θ(V) memory.
  - Example: 1,000,000 vertices that are all connected. 999,999 will be enqueued at once.

- Note: In our implementations, we have to spend Θ(V) memory anyway to track distTo and edgeTo arrays.
  - Can optimize by storing distTo and edgeTo in a map instead of an array.

The Shortest Paths Tree

对于现实中的路径，depth不能很好的反应距离，此时BFS记录的distance没有比较价值，我们需要新的ADS and algorithm。
注意到，DFS和BFS都是通过构造树来解决问题：
所以我们也能构造出 **The Shortest Paths Tree**：Dijkstra’s Algorithm

- Insert all vertices into fringe PQ, storing vertices in order of distance from source.
- Repeat: Remove (closest) vertex v from PQ, and relax all edges pointing from v.

怎么证明呢?

最优美的方法靠两步:

1. edge == v -1
2. 当我们relax a edge,总会有一个path更小，并且这个edge不会再被使用

Dijkstra’s Algorithm Runtime

结合PQ ADS以及图可以得到:

Priority Queue operation count, assuming binary heap based PQ:

- add: V, each costing O(log V) time.
- removeSmallest: V, each costing O(log V) time.
- changePriority: E, each costing O(log V) time.

Overall runtime: O(V*log(V) + V*log(V) + E*logV).

- Assuming E > V, this is just O(E log V) for a connected graph.

	# Operations	Cost per operation	Total cost
PQ add	V	O(log V)	O(V log V)
PQ removeSmallest	V	O(log V)	O(V log V)
PQ changePriority	E	O(log V)	O(E log V)

Introducing A*

Simple idea:

- Visit vertices in order of d(Denver, v) + h(v, goal), where h(v, goal) is an estimate of the distance from v to our goal NYC.
- In other words, look at some location v if:
  - We know already know the fastest way to reach v.
  - AND we suspect that v is also the fastest way to NYC taking into account the time to get to v.

对于最短路径问题，A*算法中的 `h(v,goal)` 可以取v的最小权重出边
仍然是相同的思路：

- Insert all vertices into fringe PQ, storing vertices in order of d(source, v) + h(v, goal).
- Repeat: Remove best vertex v from PQ, and relax all edges pointing from v.

How do we get our estimate?

- Estimate is an arbitrary heuristic h(v, goal).
- heuristic: “using experience to learn and improve”
- Doesn’t have to be perfect!

But:

For our version of A* to give the correct answer, our A* heuristic must be:

- Admissible: h(v, NYC) ≤ true distance from v to NYC.
- **Consistent**: For each neighbor of w:

  h(v, NYC) ≤ dist(v, w) + h(w, NYC).

  Where dist(v, w) is the weight of the edge from v to w.
- the choice of heuristic matters, and that if you make a bad choice, A* can give the wrong answer.

Admissibility and consistency are sufficient conditions for certain variants of A*.

- If heuristic is admissible, A* tree search yields the shortest path.
- If heuristic is consistent, A* graph search yields the shortest path.
- These conditions are sufficient, but not necessary.

Graph Summary

Problem	Problem Description	Solution	Efficiency
paths	Find a path from s to every reachable vertex.	DepthFirstPaths.java Demo	O(V+E) time Θ(V) space
shortest paths	Find the shortest path from s to every reachable vertex.	BreadthFirstPaths.java Demo	O(V+E) time Θ(V) space
shortest weighted paths	Find the shortest path, considering weights, from s to every reachable vertex.	DijkstrasSP.java Demo	O(E log V) time Θ(V) space
shortest weighted path	Find the shortest path, consider weights, from s to some target vertex	A*: Same as Dijkstra's but with h(v, goal) added to priority of each vertex. Demo	Time depends on heuristic. Θ(V) space

Spannig Tree

Given an undirected graph, a spanning tree T is a subgraph of G, where T:

- Is connected.
- Is acyclic.
- Includes all of the vertices.

A Minimum Spanning Tree

A minimum spanning tree is a spanning tree of minimum total weight.

Lemma:

- A cut is an assignment of a graph's nodes to two non-empty sets.
- A crossing edge is an edge which connects a node from one set to a node from the other set.
- Cut property: Given any cut, minimum weight crossing edge is in the MST.

proof :

Suppose that the minimum crossing edge e were not in the MST.

- Adding e to the MST creates a cycle.
- Some other edge f must also be a crossing edge.
- Removing f and adding e is a lower weight spanning tree.
- Contradiction!

(其实我也不知道这个lemma有什么用，但是很厉害的样子)
不难看出，可以用Dijkstra'Algorithm的算法变体实现，并且时间复杂度和SPT相同

Kruskal Spanning Tree

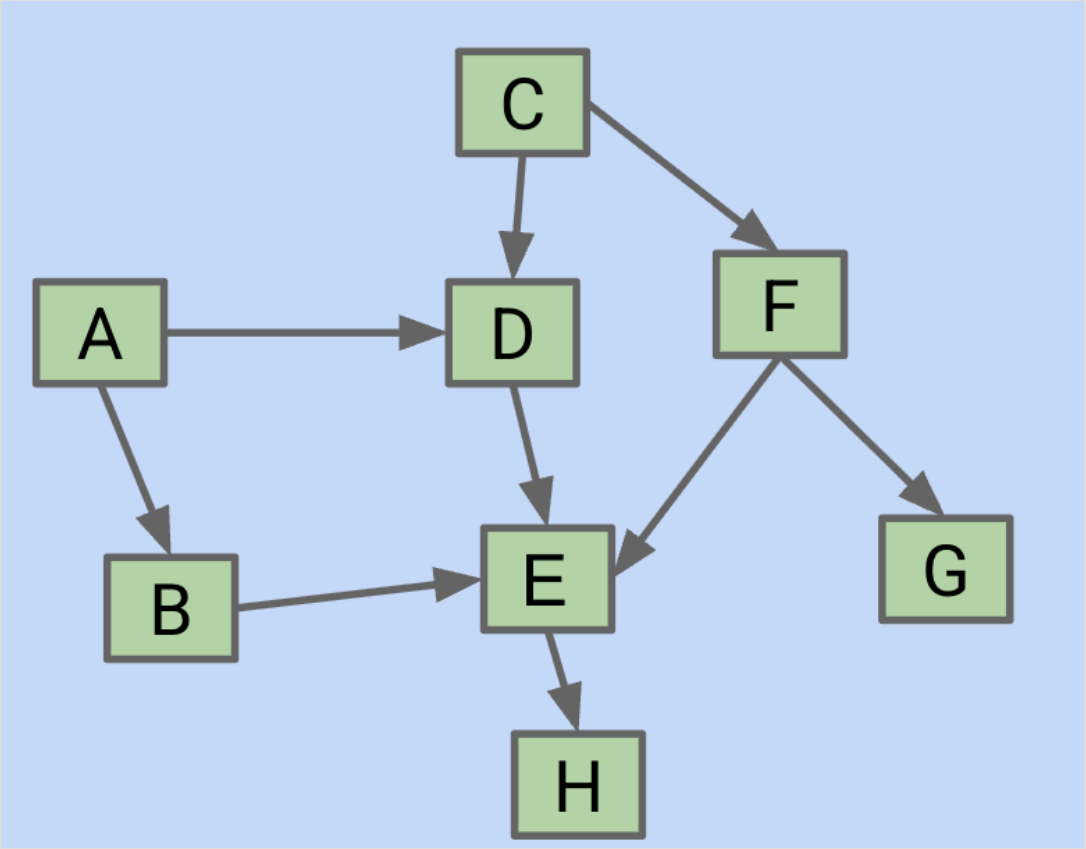
Insert all edges into PQ.
Repeat: Remove smallest weight edge. Add to MST if no cycle created.
(我们可以用WWQU来判断是否在同一个集合中)

Day10

Topological Sort

在一个Directed Acyclic Graphs中:
Suppose we have tasks A through H, where an arrow from v to w indicates that v must happen before w.

- What algorithm do we use to find a valid ordering for these tasks?
- Valid orderings include: [A, C, B, D, F, E, H, G], [C, A, D, F, B, E, G, H], …

- 

首先尝试用dfs解决问题：

当我们遍历时，可以将路径存入stack中

- while(存在只出不进节点:A,C...)
  - if 没有子节点，出栈并标记
  - else 回退到上一个节点
- 反转即得到了

Another better topological sort algorithm:

- Run DFS from an arbitrary vertex.
- If not all marked, pick an unmarked vertex and do it again.
- Repeat until done.

SPT in DAG(Directed Acyclic Graph)

If we allow negative edges, Dijkstra's algorithm can fail.

For example, below we see Dijkstra's just before vertex C is visited.

Relaxation on E succeeds, but distance to F will never be updated.

此时可以借助Topological Sort得到的拓扑序来帮助解决问题：
不难想到，当含有负点时，我们需要所有的inedge都使用过了之后才能确定min{distTo}
而我们得到Topological Sort满足：序列[A,C,B...]当C，B确定min{disTo}时，A一定可以先确定
得到启发，有：

- 得到Topological Array,node = root;
- 根据得到的Array,遍历:
  - 由in-edge计算Todis并确定To-node

Problem	Problem Description	Solution	Efficiency
topological sort	Find an ordering of vertices that respects edges of our DAG.	Demo Code: Topological.java	O(V+E) time Θ(V) space
DAG shortest paths	Find a shortest paths tree on a DAG.	Demo Code: AcyclicSP.java	O(V+E) time Θ(V) space

LPT in DAG and Reduction

由上启发而来
DAG LPT solution for graph G:

- Form a new copy of the graph G' with signs of all edge weights flipped.
- Run DAGSPT on G' yielding result X.
- Flip signs of all values in X.distTo. X.edgeTo is already correct.

这个过程被称为“归约”(Reduction)。既然有向无环图的最短路径算法（DAG-SPT）可以用于解决有向无环图的最长路径问题（DAG-LPT），我们就说“DAG-LPT可归约为DAG-SPT”。

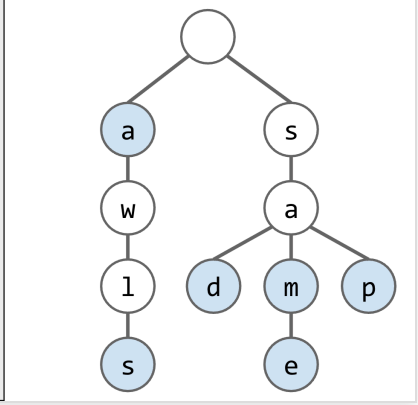
Trie(Retrival Tree)

For String keys, we can use a “Trie”. Key ideas:

- Every node stores only one letter.
- Nodes can be shared by multiple keys.
  但是这样无法知道在何处结束，所以我们还要给结尾一个tag

```
public class TrieSet {
  private static final int R = 128; // ASCII
  private Node root; // root of trie

  private static class Node {
    private char ch;
    private boolean isKey;
    private DataIndexedCharMap<Node> next;
    private Node(char c, boolean b, int R) {
       ch = c; isKey = b;
       next = new DataIndexedCharMap<>(R);
    }
 }
}
```

Since we know our keys are characters, can use a DataIndexedCharMap.



- The first approach might look something like the code below.
- Each node stores a letter, a map from c to all child nodes, and a color.

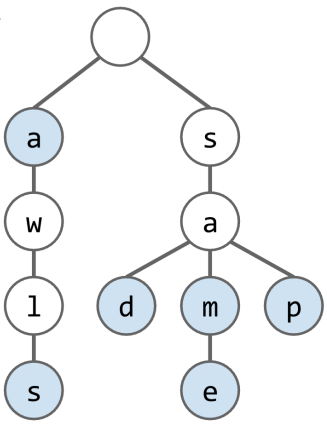
注意到：我们已经给了edge关于下一个节点的信息了，所以其实不用再显示的存储一次

> ch is useless!

此外，不难发现Map本质是一个key-value Contains

Using a BST or a Hash Table to store links to children will usually use less memory.

- DataIndexedCharMap: 128 links per node.
- BST: C links per node, where C is the number of children.
- Hash Table: C links per node.
- Note: Cost per link is higher in BST and Hash Table.



Using a BST or a Hash Table will take slightly more time.

- DataIndexedCharMap is Θ(1).
- BST is O(log R), where R is size of alphabet.
- Hash Table is O(R), where R is size of alphabet.

Since R is fixed (e.g. 128), can think of all 3 as Θ(1).

实际上，Tries的魅力还是主要在于其能很方便的实现很多String特有的method(比如说字符串的交集)

Day11

Sorting

inversion and sorting

the goal of sorting:

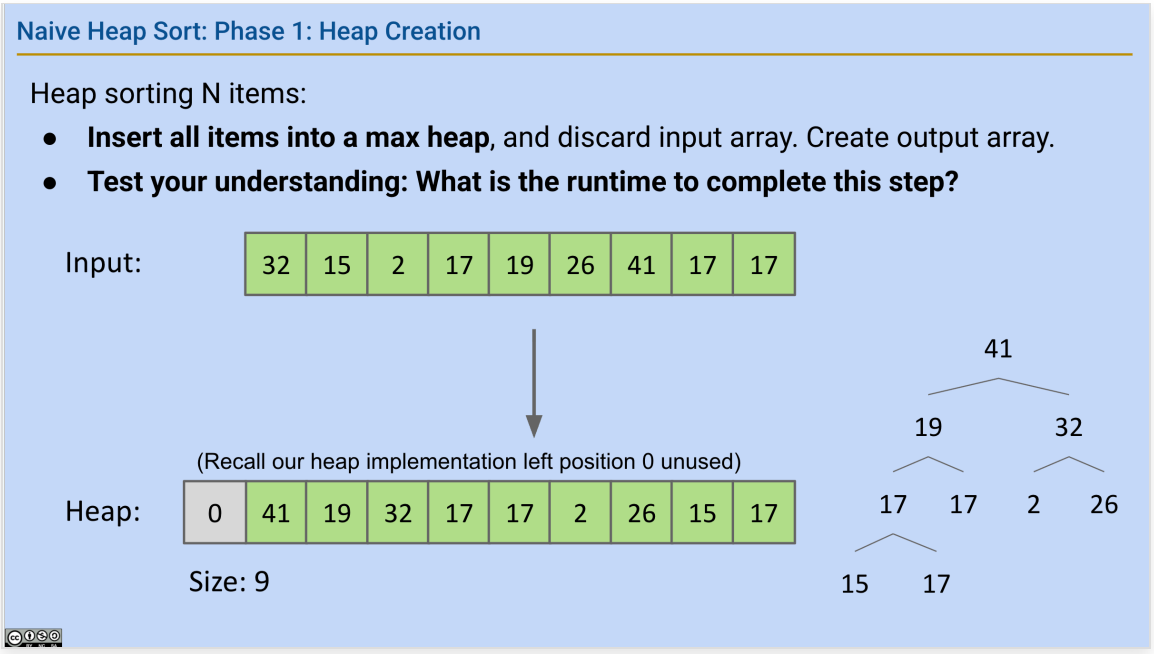
- Given a sequence of elements with Z inversions
- Perform a sequence of operations that reduces inversions to 0

Selection Sort

选择排序(特指Array中的选择排序)

Naive Heapsort

朴素堆排序: insert all items into a max heap,and discard input array. Create out put array.



in-place HeapSort

注意到可以通过避免复制为Heap中去来实现优化

- Bottom-up heapify input array:
  - Sink nodes in reverse level order:sink(k)

- After sinking,guaranteed that tree rooted at position k is a heap

此时能得到一个heap
注意到，每次取出最大的item，heap本身就会前移，所以刚好可以把取出的item放在array最后。这样就在一个array中实现了Sort

Mergesort

融合排序
特别的，对于Top-Down merge sorting N items:

- Split items into 2 roughly even pieces
- Mergesort each half
- Merge the two sorted halves to form the final result
  - Compare input[i] < input[j] (if necessary).
  - Copy smaller item and increment p and i or j.

in-place Insertion Sort

将未排序的队首通过swap转换至合适的位置
这是一个简单的算法，但是:
如果需要排序的队列只有少量的inversion，他会很快

- One exchange per inversion (and number of comparisons is similar). Runtime is $\Theta(N + K)$ where K is number of inversions.
- Define an almost sorted array as one in which number of inversions $\leq cN$ for some c. Insertion sort is excellent on these arrays.

因此，可以合理推出，当队列元素较少时insertion效率很高
进一步的，在merge排序时就可以利用这个特性处理小队列的情况

	Best Case Runtime	Worst Case Runtime	Space	Demo	Notes
Selection Sort	$\Theta(N^2)$	$\Theta(N^2)$	$\Theta(1)$	Link	
Heapsort (in place)	$\Theta(N)$*	$\Theta(N \log N)$	$\Theta(1)$	Link	Bad cache (61C) performance.
Mergesort	$\Theta(N \log N)$	$\Theta(N \log N)$	$\Theta(N)$	Link	Fastest of these.
Insertion Sort (in place)	$\Theta(N)$	$\Theta(N^2)$	$\Theta(1)$	Link	Best for small N or almost sorted.

QuickSort

- Partition
  - i=队首,j=队尾
  - while(i<j)
    - 如果[i+1]>[i]:swap(i,i+1),i=i+1
    - 如果[i]<[i+1]:swap(i+1,j),j=j-1

还能继续减少交换的次数吗?

其实只要稍做改进就可以:

- L:Doesn't like Bigger
- R:Doesn't like Smaller

  当两者都处于Doesn't like 时，Swap，否则向下一个遍历

  不断循环直到R在L左边

  交换pivot和R的位置(不难得到pivot左边一定 小于等于 pivot，右边则大于)
- QuickSort
  - Partition(array[0])
  - QuickSort(left)
  - QuickSort(right)

Listed by mechanism:

- Selection sort: Find the smallest item and put it at the front.
- Insertion sort: Figure out where to insert the current item.
- Merge sort: Merge two sorted halves into one sorted whole.
- Partition (quick) sort: Partition items around a pivot.

Listed by memory and runtime:

	Memory	Time	Notes
Heapsort	$\Theta(1)$	$\Theta(N \log N)$	Bad caching (61C)
Insertion	$\Theta(1)$	$\Theta(N^2)$	$\Theta(N)$ if almost sorted
Mergesort	$\Theta(N)$	$\Theta(N \log N)$	
Quicksort	$\Theta(\log N)$ (call stack)	$\Theta(N \log N)$ expected	Fastest sort

做简单分析会发现，如果原有序列的顺序越"好"，其实实际上时间越长

MayBe改进：

- shuffle before Sort
- 选取中间的位置作为pivot
- 提前检查

Theoretical Bounds on Sorting

Radix Sorts

Digit-by-Digit sorting

按位分解：将整数或字符串分解为多个独立的位（digit）

逐位排序：从最低有效位（LSD, Least Significant Digit）或最高有效位（MSD, Most Significant Digit）开始，依次对每一位进行排序：

- LSD方法：从最低位（如个位）向最高位排序，确保低位有序后高位排序不破坏低位顺序。
- MSD方法：从最高位开始，适用于类似字典序的排序（如字符串），但实现更复杂(每一步都是在之前的桶内再次拆分)稳定性要求：每一轮排序必须是稳定排序（如计数排序），即相同位值的元素保持原有相对顺序。

算法效率

- 时间复杂度：O(d·(n+r))

  d：最大位数，n：元素数量，r：基数（如十进制r=10）。

  当 d较小且 n较大时，效率接近线性 O(n)。

- 空间复杂度：O(n+r)，用于存储桶和计数数组。

  稳定性：依赖子排序算法的稳定性，是正确性的关键。

Count sorting

1. 确定范围
2. 统计频次
3. 累加计数
4. 反向填充

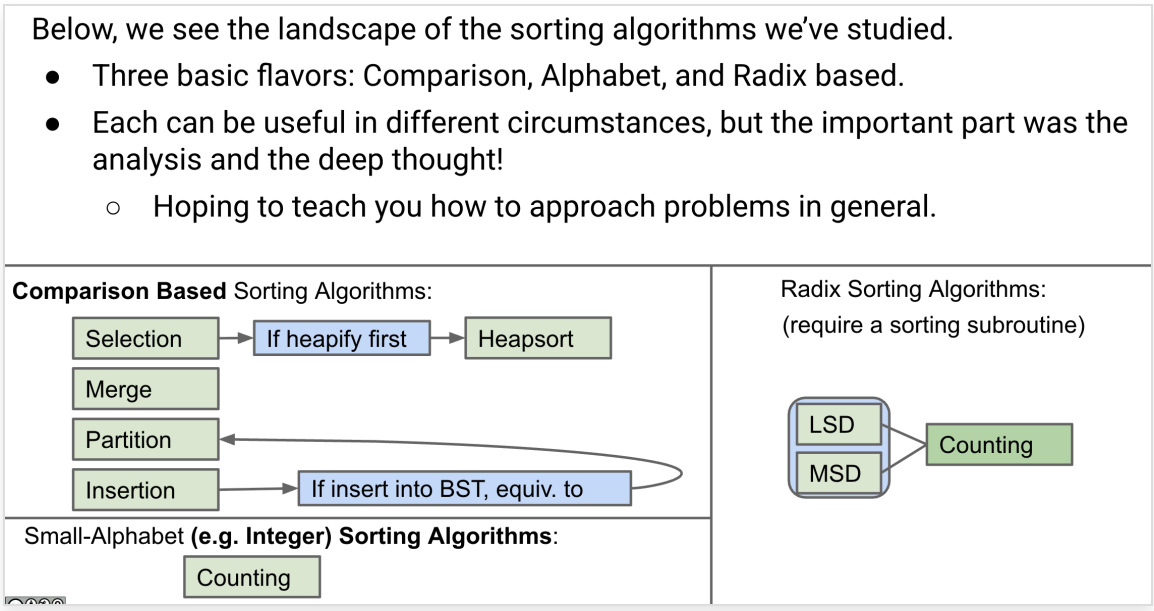
Radix sorting

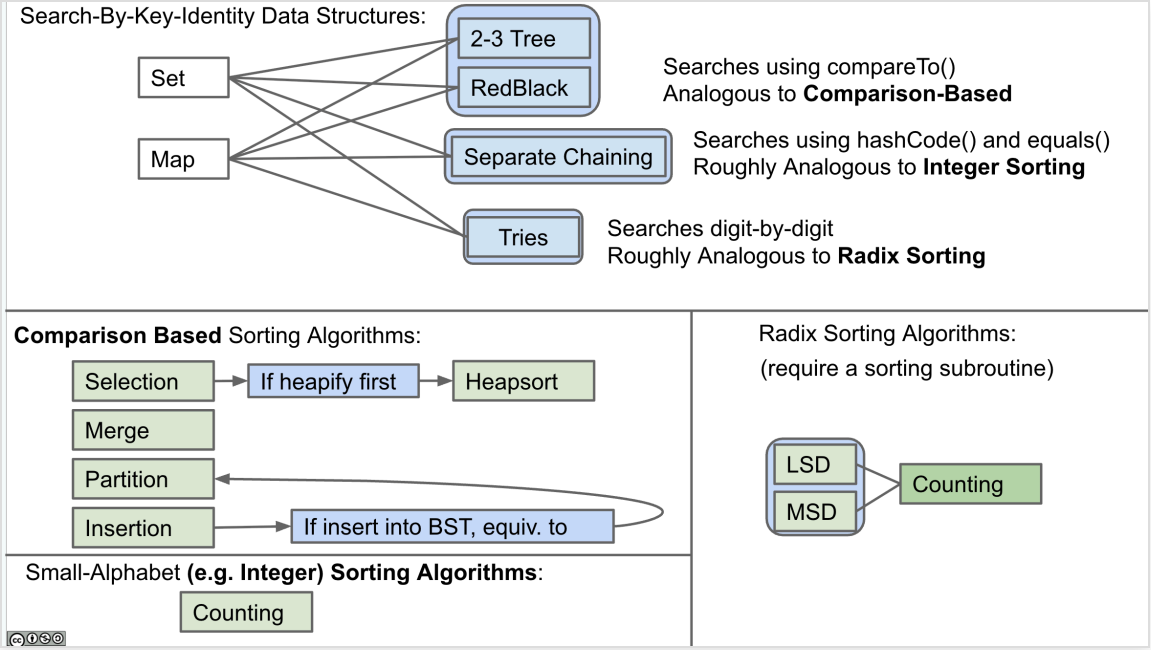
其实就是按照Digit-by-digit拆分的思路，配上count sort子排序的算法

Radix sorting Vs Compare sorting

对于String Compare:当N充分大，且字符串间高度相似时LSD会更快

对于字符串间高度不相似时，Compare sorting(特别是merge sort)会更快





Day12

Software Engineering Ⅰ

老哥的博客

61B concepts and knowledge is applicable in real world scenarios.

- Good code is more than just working code.
- Code complexity scales with functionality and poses challenges to maintain.
- Practice good design principles in your classes.
- Testing is necessary to have confidence in your code.
- The real world will give you ambiguous problems with loosely defined constraints; it is your duty to make sense of these and then leverage your skills to select the best solutions given constraints while considering tradeoffs.

Software Engineering Ⅱ :How To Build A Project Up?

Design

需要注意

- Ensures consideration of all requirements and constraints

- Consider multiple alternatives and their advantages/disadvantages
- identifies potential costs or pitfalls while change is cheap
- All multiple minds to weigh in and leverages diverse perspectives

Design document should:

- Quick bring the reader up to speed
- include what alternatives were considered and why the trade-offs were made
- Be just as much about the 'why' as it is about the 'how'
- update as the project progresses

你还应该思考:

- each class you want to create and how they will fit together
  ideally each class should have one role
- consider what methods are required to meet each class's functionality
- The major algorithms you'll need to think about are room and hallway generation

Writing Clean Code

technical debt

- u work need to be available to other people
- deadlines exit

How to deal with this?

SFEI may give the answer

We also use **refactoring** to clean our code

- NO Immediate clear gains - it's all investment in the future

Documentation Strategies

对于函数注释:

1. 给出一个关于函数功能的summary
2. 给出special cases时的应对方式
3. @param:解释输入的意义(提供函数接口的说明)

4. @return:解释函数的返回情况
5. @throws:解释不规范使用措施
6. @usage:给出函数使用示例

对于Commit Message:

这个博客

Working with Team

Turn-taking:轮流发言，建议

Reflexivity:

- “A group’s ability to collectively reflect upon team objectives, strategies, and processes, and to adapt to them accordingly.”
- Recommended that you “cultivate a collaborative environment in which giving and receiving feedback on an ongoing basis is seen as a mechanism for reflection and learning.”
    - It’s OK and even expected for you and your partner to be a bit unevenly matched in terms of programming ability.

Pair Programming

Pair programming is a technique where two programmers work on a single station

- Driver: focus on implementation
- Navigator: consider design, accuracy

Benefits:

- Reduce errors
- Improve code quality
- Increase productivity

Tips:

- Clearly define roles
- Swap often (15-30 mins)

Software Engineering III

two fundamental steps to write a program:

- Try to come up with a algorithm that solves the problem
- convert this algorithm into a working code

Whiteboarding

- separate the algortithm writing from coding
- **Break down the problem even further to little pieces or simplest versions**
- 流程图