CS1083 Week 4 : Polymorphism

Interfaces, Comparable, Searching

David Bremner

2018-01-22

Interfaces

Linear Search

Binary Search

Interfaces

An interface is like a class, with all the implementation left out.

```java
public abstract interface Belch{
    public abstract void burp();
}
```

Interfaces

An interface is like a class, with all the implementation left out.

```
public abstract interface Belch{
    public abstract void burp();
}
```

By saying “class A implements B”, you claim to provide certain methods.

```
public class Polite implements Belch{
  public void burp(){
    System.out.println("burp.␣Oh,␣excuse␣me!");
  }
}
```

Interfaces

By saying “class A implements B”, you claim to provide certain methods.

```
public class Polite implements Belch{
  public void burp(){
    System.out.println("burp.␣Oh,␣excuse␣me!");
  }
}
```

But Java knows nothing of what these methods do!

```
public class Rude implements Belch{
    public void burp(){
        System.out.println("Burrrrpppppp!");
    }
}
```

Comparable Interface

The generic interface *Comparable*<*T*> has only one method

```java
public int compareTo(T o)
```

From the doc

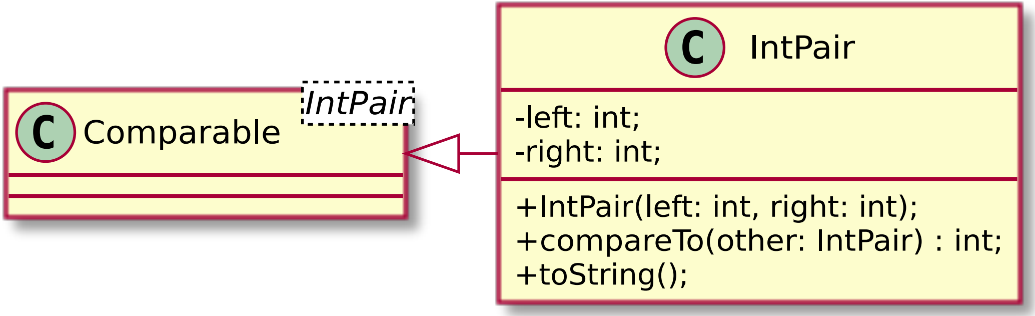
Compares this object with the specified object for order.

Parameters o - the Object to be compared.

Returns a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object.

Throws ClassCastException - if the specified object's type prevents it from being compared to this Object.

Designing a ”Comparable class”


C
Comparable
IntPair
C
IntPair
-left: int;
-right: int;
+IntPair(left: int, right: int);
+compareTo(other: IntPair) : int;
+toString();

Implementing Comparable

```
public class IntPair implements Comparable<IntPair> {
    private int left, right;

    public IntPair(int left, int right) {
        this.left=left;
        this.right=right;
    }

    public int compareTo(IntPair other) {
        if (this.left == other.left)
            return (this.right - other.right);
        else
            return (this.left - other.left);
    }
    ⋮
}
```

Using our Comparable class

```java
for (int i=0; i<pairs.size(); i++) {
    for (int j=0; j<pairs.size(); j++) {
        IntPair a = pairs.get(i);
        IntPair b = pairs.get(j);

        int order = a.compareTo(b);
        String op = "==";
        if (order < 0)
            op = "<";
        if (order > 0)
            op = ">";
        System.out.println(a + op + b);
    }
}
```

IntPair

Interfaces

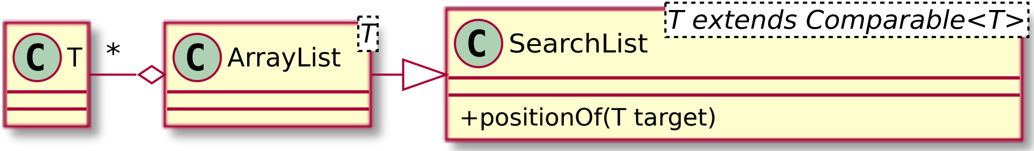
Linear Search

Binary Search

Searching with Comparable

```
public int positionOf(T target) {
    for (int i=0; i<this.size(); i++) {
        if (this.get(i).compareTo(target)==0) {
            return i;
        }
    }
    return -1;
}
```

A container class for Comparables



```
public class SearchList<T extends Comparable<T>>
    extends ArrayList<T> {
⋮
}
```

SearchList

Using Search List for Strings

```
SearchList<String> staff = new SearchList<String>();
staff.add("Tom");
⋮
staff.add("Harry");

String[] queries = {"Harry", "Tom", "Moe", "Curly"};
for (String name : queries) {
     int pos = staff.positionOf(name);
     if (pos <0 )
          System.out.println(name + "␣not␣found");
     else
          System.out.println(name+"␣found␣@␣"+pos);
}
```

SearchList

Using SearchList for integers

```
SearchList<Integer> numbers = new SearchList<Integer>();

for (int i=0; i<size; i++){
    numbers.add(random.nextInt(size));
}

for (int j=0; j<searches; j++){
    int target = random.nextInt(numbers.size());
    int pos = numbers.positionOf(target);
    ⋮
}
```

SearchBench

Using SearchList for IntPairs

SearchList

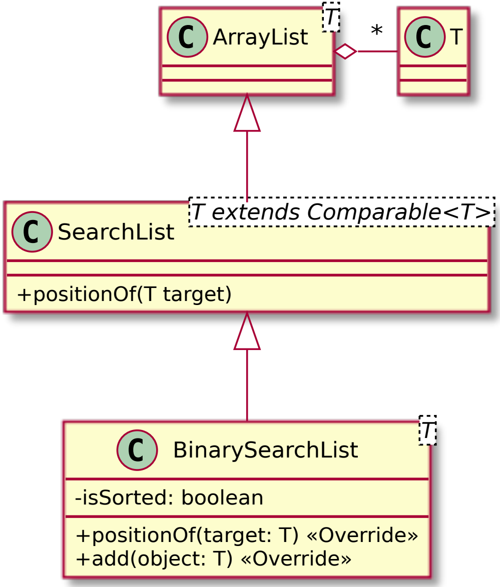
```
for (int i=0; i<pairs.size(); i++) {
    IntPair intQuery = new IntPair(random.nextInt(3),
                                   random.nextInt(3));
    int pos = pairs.positionOf(intQuery);
    if (pos < 0)
        System.out.println(intQuery + "␣not␣found");
    else
        System.out.println(intQuery+"␣found␣@␣"+pos);
}
```

Interfaces

Linear Search

Binary Search

BinarySearch



ArrayList T

*

T

SearchList T extends Comparable<T>

+positionOf(T target)

BinarySearchList T

-isSorted: boolean

+positionOf(target: T) «Override»
+add(object: T) «Override»

BinarySearchList

```java
int left=0;
int right=this.size()-1;
while (left<=right){
    int mid=(left+right)/2;
    int diff =
        get(mid).compareTo(target);
    if (diff==0) return mid;
    if (diff<0)
        left=mid+1;
    else
        right=mid-1;
}
return -1;
```

Running binary search

looking for 7

0	1	3	3	4	6	6	$\boxed{8}$	9	...	17	18	19
0	1	3	$\boxed{3}$	4	6	6						
				4	$\boxed{6}$	6						
						6						

Running binary search

looking for 7

0	1	3	3	4	6	6	$\boxed{8}$	9	...	17	18	19
0	1	3	$\boxed{3}$	4	6	6						
				4	$\boxed{6}$	6						
						6						

looking for 4

0	2	2	2	$\boxed{3}$	3	5	5	6	7
					3	5	$\boxed{5}$	6	7
					$\boxed{3}$	5			
						5			

Running binary search

looking for 4

0	2	2	2	$\boxed{3}$	3	5	5	6	7
					3	5	$\boxed{5}$	6	7
					$\boxed{3}$	5			
						5			

finding 4

0	2	2	2	$\boxed{3}$	4	5	5	5	7
					4	5	$\boxed{5}$	5	7
					$\boxed{4}$	5			

Ensuring list is sorted

- binary search needs sorted input
- requiring user to insert in order is sometimes impossible
- requiring user to sort is error prone
- when is this efficient?

```
public boolean add(T obj) {
  isSorted = false;
  return super.add(obj);
}

public int positionOf(T target) {
  if (!isSorted) {
    // uses List,
    // Comparable interfaces
    Collections.sort(this);
    isSorted = true;
  }
  ⋮
}
```

BinarySearchList

More polymorphism

```
public long timeSearches(SearchList<Integer> numbers){
    long startTime = System.nanoTime();
    for (int j=0; j<searches; j++){
        int target = random.nextInt(numbers.size());
        int pos = numbers.positionOf(target);
    }
    long endTime = System.nanoTime();
    return endTime-startTime;
}
```

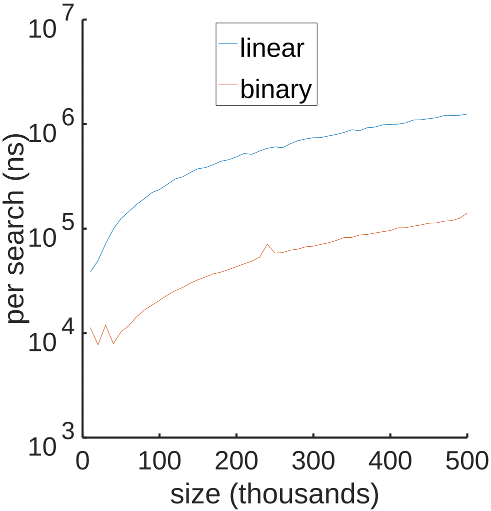
SearchBench

More polymorphism

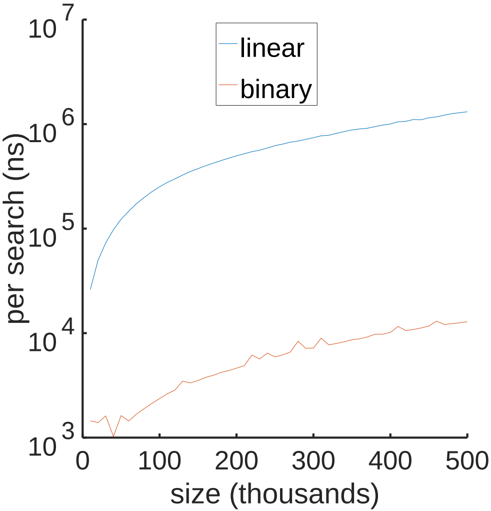
```
SearchList<Integer> numbers = new SearchList<Integer>();

fill(numbers, size);
System.out.print(timeSearches(numbers));

BinarySearchList<Integer> bnumbers
     = new BinarySearchList<Integer>();

fill(bnumbers, size);
System.out.println("\t"+timeSearches(bnumbers));
```

SearchBench

Comparing running times

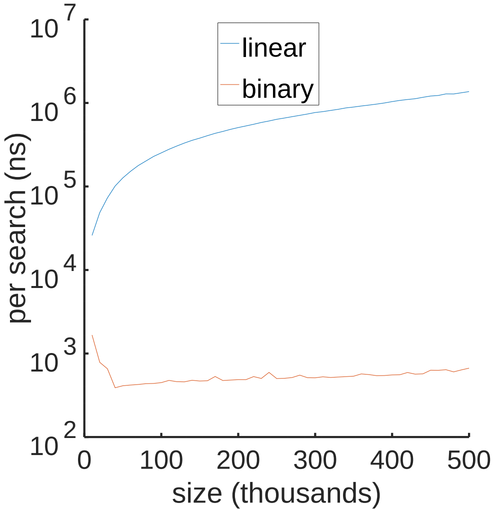


1000 queries



10000 queries

Comparing running times



1 query per array element

Analysis of binary search

BinarySearchList

```
while (left<=right){
    int mid=(left+right)/2;
    ⋮
    if (diff<0)
        left=mid+1;
    else
        right=mid-1;
}
return -1;
```

$$
\begin{aligned}
n_0 &= n \\
n_i &< \frac{n_{i-1}}{2} \qquad \text{(why?)} \\
&\vdots \\
&< \frac{n}{2^i}
\end{aligned}
$$

Let n_i be `right` − `left` after i iterations of `while`

How many times can we divide n by 2 before we get 1?