ECS510

Algorithms and Data Structures in an Object Oriented Framework "ADSOOF"

Using Arrays

Array Types

- In Java, for any type adding [] to the type gives a new type which we call "array of …" where … is the original type (called the "base type")
- So:
  - `int[]` array of integers
  - `String[]` array of strings
  - `DrinksMachine[]` array of `DrinksMachines`
  - `int[][]` array of array of integers

Use of Array Types/Variables

- You can use array types in the same way as you can use any other type:
  - Declare variables of the type
  - Have method parameters of the type
  - Have it as a method return type
- You can use variables of an array type in the same way as any other variables
  - Assign to them
  - Pass them as arguments to methods
  - Use them in return statements in methods

Arrays as objects

An array type is an object type, so

- Declaring a variable of an array type does not create an object of that type

  ```
  int[] a,b;
  ```

- Assignment of array type variables leads to aliasing

  ```
  b=a;
  ```

- So when an array is an argument to a method call, the local variable when the method is evaluated is initialised to an alias of the argument variable

Array Syntax

- Arrays are an indexed collection of items of the base type
- If `a` refers to an array, `a[`*`expr`*`]` refers to an item in the array, where *`expr`* evaluates to an integer value when the code is run
- You can treat `a[`*`expr`*`]` as a variable, use it in expressions, assign to it:

```
n=a[i]+1;
m=test(a[i*2]);
a[i]=n*2;
```

Array construction

- An array object is constructed by

  `new t[expr]`

  where *t* is the base type, and *expr* evaluates to an integer value when the code is run
- If *v* is the value of *expr* when the code runs, then the array object is fixed at having *v* items, indexed from 0 to *v*-1.
- Although an array object is of fixed size, an array variable can be reassigned from referring to an array of one length to referring to an array of another length.

Other array issues

- If `a` is a variable of an array type, `a.length` is the length of the array it currently refers to
- With `a[`*`expr`*`]` if *`expr`* evaluates to a negative integer or one equal to or greater than `a.length`, an `ArrayIndexOutOfBoundsException` is thrown
- Arrays are the only objects which have special symbols for accessing parts of them. All other collection objects only use ordinary method calls

Array Aliasing

- If `a` and `b` are array variables and currently aliases, and *`expr1`* and *`expr2`* evaluate to the same value, then `a[`*`expr1`*`]` and `b[`*`expr2`*`]` are the same variable
- So changing one changes the other, `a[i]=val` changes `b[i]`because `a[i]` and `b[i]` are the same location
- Usual aliasing rules apply, so `b=c` causes `b` to stop being an alias of `a` and become an alias of `c`

Searching for an item in an array

```
public static boolean isIn(String[] a, String str)
{
 for(int i=0; i<a.length; i++)
    if(a[i].equals(str))
       return true;
 return false;
}
```

- When `isIn(dict,word)` is called, `a` is an alias for `dict` (and `str` is an alias for `word`)
- This is linear search, if we know the array is ordered, we can use binary search
- Same algorithm used for any base type (`String` here)

Position of an item in an array

```
public static int isIn(String[] a, String str)
{
 for(int i=0; i<a.length; i++)
    if(a[i].equals(str))
       return i;
 return -1;
}
```

- Returning `-1` for "not found" is a common convention
- If the item occurs more than once, this would return the lowest index where it occurs
- Best to specify exactly what is wanted when there is more than one possibility

Biggest item in an array

```
static int biggest(int[] a)
{
 int biggestSoFar=a[0];
 for(int i=1; i<a.length; i++)
     if(a[i]>biggestSoFar)
        biggestSoFar=a[i];
 return biggestSoFar;
}
```

- Same algorithm can be used for "most …" `t` in an array of `t` by varying base type and test
- Java has ways of generalising code to enable one method to be used in a variety of circumstances (see later)
- Assumes array length is not 0

Loop invariant

In:

```
biggestsofar=a[0];
for(int i=1; i<a.length; i++)
        if(a[i]>biggestSoFar)
                biggestSoFar=a[i];
```

- Each time the loop body starts, `biggestsofar` holds the largest integer in the portion of the array up to but not including position `i`.
- When the loop finishes, we know `i` is equal to `a.length`
- So when the loop finishes, we know `biggestsofar` holds the largest integer in the whole array
- A condition which holds each time a loop body starts is known as a “loop invariant”, and is a way we can prove algorithms to be correct

Naming conventions

- Java recommends initial upper case letter for class names, initial lower case letter for method names and variable names
- Class and method names should always be meaningful
- Meaningful variable names are good if it helps explain what they are being used for
- Variable names can be short in short generalised code, where the use of the variable is obvious
- Short names also have conventions

Short name conventions

- Use `i`, `j` for integers that loop through ranges, but not anywhere else
- Array parameters are commonly named `a` or `arr`, `b` for further arrays
- Use `m`, `n` for integer parameters
- Use `x`, `y` for floating point values only
- Use `str`, or `str1`, `str2` for Strings (or `name`, `word`, `key` etc if appropriate)
- Short abbreviations for other types
- Be cautious of `temp` and `flag` (can indicate poorly structured code in their conventional uses)

Changing arrays destructively

```
public static void dchange(String[] a, String w1, String w2)
{
 for(int i=0; i<a.length; i++)
   if(a[i].equals(w1))
      a[i]=w2;
}
```

means

```
dchange(b,word1,word2);
```

changes each `word1` in `b` to `word2`

- Works because `a` in the method call aliases `b` outside

Changing arrays constructively

```
public static String[] cchange(String[] a, String w1, String w2)
{
 String[] a1 = new String[a.length];
 for(int i=0; i<a.length; i++)
   if(a[i].equals(w1))
      a1[i]=w2;
   else
      a1[i]=a[i];
 return a1;
}
```

means

```
c=cchange(b,word1,word2);
```

makes a new array which is like `b` but each `word1` is changed to `word2`, sets `c` to that array

Using constructive array methods

- `c=cchange(b,word1,word2);`

  The new array is created in the method call, but remains in existence because `c` aliases it
- `b=cchange(b,word1,word2);`

  Does not change `b` destructively, seems to because `b` is assigned to refer to the new array
- `cchange(b,word1,word2);`

  On its own achieves nothing, because nothing is assigned to alias the array created in the method call

Effect on aliases

- `c=b;`

  `dchange(b,word1,word2);`

  changes `c` as well, it is an alias to `b`, the old value of the array is destroyed.

- `c=b;`

  `b=cchange(b,word1,word2);`

  does not change `c`, it continues to refer to what was the old value of `b`

Changing the size of an array

- The contents of an array can be changed destructively, but its size cannot
- So any operation which changes the size of an array must be done constructively
- Technique is to create a new array of the required size, copy items into required position, return array

Copying an array

```
public static String[] copy(String[] a)
{
 String[] b = new String[a.length];
 for(int i=0; i<a.length; i++)
    b[i]=a[i];
 return b;
}
```

- Be careful to distinguish between a copy of an array and an alias of an array
- Note `b[i]=a[i]` means the contents of the copy are aliases if the base type is an object type …

Adding an item to an array

```
public static String[] add(String[] a, String str)
{
 String[] b = new String[a.length+1];
 for(int i=0; i<a.length; i++)
    b[i]=a[i];
 b[a.length]=str;
 return b;
}
```

- means

  `c=add(c,word);`

  "adds" `word` to the end of `c`

This won't work!

```
public static void add(String[] a, String str)
// THIS IS SILLY CODE!
{
 String[] b = new String[a.length+1];
 for(int i=0; i<a.length; i++)
    b[i]=a[i];
 b[a.length]=str;
 a=b;
}
```

will not cause `add(c,word)` to add `word` to the end of `c` destructively because `a` starts aliasing `c`, but the `a=b` causes `a` to alias `b` while leaving `c` as it was

Removing item at a position

```
public static String[] removePos(String[] a, int pos)
{
 String[] b = new String [a.length-1];
 for(int i=0; i<pos; i++)
    b[i]=a[i];
 for(int i=pos+1; i<a.length; i++)
    b[i-1]=a[i];
 return b;
}
```

- New array one less in size
- Items before `pos` put into same position
- Items after `pos` put one place before previous position
- Throw an `ArrayIndexOutOfBoundsException` if `pos` is not a position in the array. Specify this? Add check to code?

Destructive removal of an item at a position?

```
public static void removePos(String[] a, int pos)
{
 a[pos]="";
}
```

- No, this is replacing the element at position `pos` by `""`, not removing it.
- The array:

  `["apple","pear","plum","","peach","banana"]`

  is not the same as:

  `["apple","pear","plum","peach","banana"]`
- Neither is:

  `["apple","pear","plum",null,"peach","banana"]`
- Setting a position to `null` does not cause following elements to be moved down by one position.

Removing a particular item

```
public static String[] remove(String[] a, String w)
{
 int i=0;
 for(; i<a.length; i++)
    if(a[i].equals(w))
       break;
 if(i<a.length)
    {
     String[] b = new String[a.length-1];
     for(int j=0; j<i; j++)
        b[j]=a[j];
     for(int j=i+1; j<a.length; j++)
        b[j-1]=a[j];
     return b;
    }
 else
    {
     String[] b = new String[a.length];
     for(i=0; i<a.length; i++)
        b[i]=a[i];
     return b;
    }
}
```

Removing a particular item

- We did not know whether the item occurred, so we could not create the new array until we found out
- If the item occurred, the new array is one less in size of the argument array
- If the item did not occur, we still create a new array (copy of argument) because otherwise after

  `b=remove(a,str);`

  we would not know if `b` is an alias of `a` or not, better to know it can't be

Recursion

- Recursion is when a problem is solved by solving a smaller version of the problem and using that to get the required result
- In coding terms it means a method that makes a call to the same method
- Binary recursion is when the problem is solved by solving two smaller versions and putting the results together, so two calls to the same method
- For processing arrays, it is generally better to use iteration (that is, loops) rather than recursion
- So the following examples are just given to illustrate some of the concepts, and to help you understand recursion

Finding the biggest recursively (1)

```
public static int biggest(int[] a) {
 return biggest(a,a[0],1);
}

private static int biggest(int[] a,
                           int biggestSoFar, int i) {
 if(i<a.length)
    if(a[i]>biggestSoFar)
       return biggest(a,a[i],i+1);
    else
       return biggest(a,biggestSoFar,i+1);
 else
    return biggestSoFar;
}
```

- This is tail recursion, as the result of a recursive call is returned directly.

Tail Recursion and Iteration

- Tail recursion is identical in terms of algorithm to the use of a loop (iteration)
- However, where a loop involves changing the value of a variable or variables each time round the loop, tail recursion involves creating new variables of the same name but with the different values as the arguments to recursive calls
- So in this example, instead of one variable called `i` giving the current position, each call has its own variable called `i`
- Also, each call has its own variable called `a`, but they all alias the same array object
- Each call has its own variable called `biggestSoFar`, setting it to a different value is equivalent to assignment in iteration
- Tail recursion takes up space due to the old environments remaining in place even though they are not used again

Finding the biggest recursively (2)

```
public static int biggest(int[] a) {
 return biggest(a,0);
}

private static int biggest(int[] a, int from) {
 if(from==a.length-1)
    return a[from];
 else {
     int biggestOfRest = biggest(a,from+1);
     if(a[from]>biggestOfRest)
         return a[from];
     else
         return biggestOfRest;
    }
}
```

- This is not tail recursion because the result of the recursive call is processed to give the return value instead of just being returned

Thinking recursively

- The way of thinking about this second example of finding the biggest expressed recursively is:
  - To find the biggest element in the portion of an array starting at position `from`, find the biggest in the portion starting at position `from+1`, and return whichever is the biggest of that and the element at position from
  - If `from` is the last position, just return the element at that position (the base case)
- When `from` is 0, that means the biggest of the whole array is returned
- It can help develop algorithms to be able to think recursively
- However, tail recursion should be converted to iteration

Finding the biggest recursively (3)

```
public static int biggest(int[] a) {
 return biggest(a,0,a.length);
}

private static int biggest(int[] a, int from, int to) {
 if(to==from+1)
    return a[from];
 else if(to==from+2)
    return a[from]>a[from+1] ? a[from] : a[from+1];
 else {
     int mid=(from+to+1)/2;
     int biggest1=biggest(a,from,mid);
     int biggest2=biggest(a,mid,to);
     return biggest1>biggest2 ? biggest1 : biggest2;
    }
}
```

- This is an example of binary recursion

Recursion and arrays

- In the binary recursion example, finding the biggest uses the algorithm:
  - Cut the array in half, find the biggest in each half and return whichever of those is the biggest
  - Or, if the array portion being considered has just one element, just return that element (the base case)
  - Or if it has two elements, return the biggest of the two
- What this actually involves is having an array and indexes to the start and finish positions of the portion being considered
- This technique is also used in the binary search and quicksort algorithms when used with arrays
- However, recursion makes more sense when used with data structures that can more naturally be divided into pieces

Removing an item recursively (1)

```
public static int[] remove(int[] a, int n)
// Works, but very inefficient
{
 if(a.length==0)
    return a;
 int[] b = new int[a.length-1];
 for(int i=0; i<b.length; i++)
    b[i]=a[i+1];
 if(a[0]==n)
    return b;
 int[] c = remove(b,n);
 int[] d = new int[c.length+1];
 d[0]=a[0];
 for(int i=0; i<c.length; i++)
    d[i+1]=c[i];
 return d;
}
```

Removing an item from an array recursively (1): Algorithm

- If the array is of length 0, return the array
- If the first item in the array (index 0) is the item to be removed, return a new array consisting of all elements except the first element
- Otherwise, get the result of removing the item from an array consisting of all elements except the first element, and create and return a new array consisting of that result plus the original first element put at the front

- This is very inefficient because each recursive call involves creating an entire new array and copying references into it
- It is a more efficient approach when used with a data structure where something similar can be done but without copying every reference

Removing an item from an array recursively (2)

```
public static int[] remove(int[] a, int n) {
 return remove(a,0,n);
}

private static int[] remove(int[] a, int pos, int n) {
  if(pos==a.length)
     return new int[a.length];
 if(a[pos]==n) {
     int[] b = new int[a.length-1];
     for(int i=pos; i<b.length; i++)
        b[i]=a[i+1];
     return b;
    }
 int[] c = remove(a,pos+1,n);
 c[pos]=a[pos];
 return c;
}
```

Removing an item from an array recursively (2)

- This is efficient because it only creates one new array
- The new array is created when the position of the item being removed is found, or the end of the array is reached and it has not been found
- These are the “base cases” (no further recursion)
- What is passed to a recursive call is a reference to the original array and a position
- What is returned from the recursive calls is a reference to the one new array created
- References to elements are copied into the new array after the recursive call returns a reference to it
- It is still better to use iteration for processing arrays in most cases

Using Arrays

- Arrays are the oldest way of structuring data, they have been in programming languages since the earliest days, and reflect the underlying hardware
- But they are inflexible, fixed length, cannot insert/delete items
- Arrays are a building block for constructing other more flexible “abstract data types” (see later)
- In this section we also considered issues such as destructive/constructive which apply more generally