# Data Structures and Algorithms 120

## Lecture 4: Arrays, Stacks and Queues

Department of Computing

Curtin University

Last updated: 26 August 2014

# Copyright Warning

# This Week

- Data structure: arrays
- ADTs: stacks, queues
  - Implementing them with arrays
- Searching
- Algorithm Complexity
- ReadInt
  - using stack
    - compare to recursion
- Postfix:
  - Postfix evaluation
  - Infix to postfix conversion

# Arrays

- The variables that we have seen so far represent an single item.
  - eg: int numTiles is a single integer number
- But we also often work with *sets* of similar data
  - eg: the list of student marks in DSA120. How to handle?
    - » double student1Mark, student2Mark, student3Mark, …?
  - Clumsy!
    - » Variable names defined at compile time – 'hard-coding': program can never change the number of students
    - » Calculating the average involves a massive amount of typing
    - » Can't conveniently pass the set of students around

# Arrays

- Arrays are a solution to this problem
- Simplest kind of data structure for storing sets of data
  - Arrays are built-in to *all* programming languages
  - Instead of just one element, an array is a variable that contains *many* elements
  - The array variable itself is a reference to the first element of the array
    - » Java: the array variable also knows how large the array is
    - » C: doesn't store the array length – you have to do it yourself!

# Array Properties

- Elements are located sequentially in memory
  - ie: the array is a *contiguous* block of memory
- All elements must have same data type
  - eg: double
- Arrays can be initialised to any size
  - within memory limits
- However, once initialised they cannot be resized
  - Must create a new array and copy over the contents of the old array in order to 'resize' an array

# Arrays - Properties

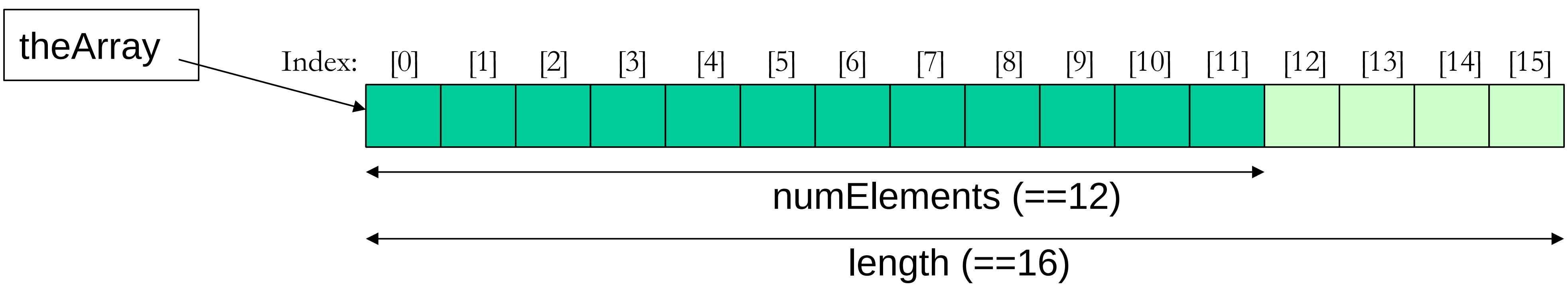
– Array capacity (length) vs actually used elements
  - Initialising an array to (say) length=20 doesn't *set* 20 elements, it merely *reserves space* for 20 elements
    - » Hence initialisation is also referred to as *allocation*
  - You therefore typically need to keep track of how many elements you have *actually used* in the array
    - » ie: the count of elements, as distinct from the array capacity
    - » It is typical that you allocate more space than you initially need, since arrays have a fixed capacity and cannot be resized
    - » Java does not track the array's count of used elements

# Arrays – Accessing Elements

– Once you have allocated an array, you need to be able to work with the elements inside the array

– Elements are accessed via an *index* (or *subscript*)

- The index is the element number in the array
  - » 0,1, 2, 3, 4, … N-1, where N is the allocated size (length)
  - » the index is an *offset* from the first element



# Arrays In Code (Java)

- Declaring: put '[ ]' on the end of the data type
  - » eg: double[ ] theArray;
  - » Any data type can be used with arrays, including classes
- Allocating: use new keyword with special [ ] syntax
- Indexing: theArray[index], index must be an int
  - » Negative indexes or indexes that are past the end of the array (ie: >= length) will cause an error during runtime
- Assignment: sameArray = theArray;
  - » Assignment doesn't copy the array *contents*, it only makes the l.h.s. variable point at the *same array* as the r.h.s.
  - » Same with passing an array as a parameter to a method
  - » Use a loop or System.arraycopy() to copy contents

# Searching

- It is often necessary to search through a list (array) for a particular value.
  - What if it is not in the list?
- Unfortunately, if the list is not sorted we might have to look at every element.
- Start at the first element – is it the one we want? No, look at the next one. Yes, finish, we found it.
  - what loop should we use?

# find()

- assume arrayToSearch and numElements are classfields

- Submodule: find (AKA linear search)
- Import: key (item to find)
- Export: location (index)
- Assertion: returns the location of key if it exists in the array, otherwise throws an exception

```
location=0, found = false
DO
   IF arrayToSearch[location].equals<--key
      found = true
   ELSE
      increment location
WHILE NOT found AND location<numElements
IF NOT found
   throw appropriate exception
```

# insert

- three scenarios
  - end
    - » element [numElements]
    - » easy!

```
IF arrayForInsert is not full                        ← throw exception if it is!
   arrayForInsert[numElements] = insertValue
   increment numElements
ENDIF
```

  - beginning
    - » element [0]

```
IF arrayForInsert is not full                       ← throw exception if it is!
   FOR ii= numElements, ii>0, decrement ii          ← Shuffle elements away to make room
      arrayForInsert[ii] = arrayForInsert[ii-1]
   ENDFOR
   arrayForInsert[0] = insertValue
   increment numElements
ENDIF
```

# insert - somewhere else

- Must be in sorted order

```
position=0
IF arrayForInsert is not full                         ← throw exception if it is!
   WHILE  insertValue > arrayForInsert[position] AND position < numElements
      increment position
   ENDWHILE

   increment position                                 ← Putting value in next element

   FOR ii=numElements, ii>position, decrement ii      ← Shuffle elements away
      arrayForInsert[ii] = arrayForInsert[ii-1]
   ENDFOR
   arrayForInsert[position]=insertValue
   increment numElements
ENDIF
```

# delete (remove)

- three scenarios
  - » need to ensure the array is not empty!
  - » throw exception if it is
- end
  - » element [n-1]
  - » Decrement count.
- begining
  - » element [0]
  - » Starting from element [1], shuffle the rest of the elements down by one, overwriting element [0]. Decrement count.
- somewhere else
  - » element[x?]
  - » Find the element to delete. Starting from the next element, shuffle the rest of the elements down by one, overwriting the element to delete. Decrement count.

# Introduction to Time Complexity Analysis

- In computers, time in seconds is not a useful measure of an algorithm since faster hardware can reduce the time
- Instead, we need to talk about how many steps are needed
  - Which is independent of hardware speed, so is a better 'absolute' measure of speed
  - And where 'steps' really is CPU instructions
- Unfortunately, we can never know *exactly* how many CPU instructions something takes
  - Different CPUs have different instruction sets and are faster/slower with different instructions vs other CPUs

# Big-O Notation

- Instead we give an estimate of the number of steps, focusing on how the algorithm scales with more data
  - We want to know whether the algorithm will handle lots of data well or if it will become quickly unusable
- Big-O notation was developed for this purpose
  - Indicates the 'order-of' the algorithm (ie: what ballpark it is in)
  - Notation: O(<numSteps>), eg: $O(N)$, $O(N \log N)$, $O(N^2)$
  - Ignores multiplying constants
    - » Only concerned with scalability as the amount of data increases
    - » eg: $O(N)$ means "double the data N, and you double the time"
    - » eg: $O(N^2)$ means "triple the data N, and you 9x the time"
  - We will use this to compare algorithms

# Sample Values

- O(1)
  - » Number of steps is constant, no matter N
- O(log N)
  - » N=100 steps=7, N=1000 steps=10, N=$10^6$ steps=20
- O(N)
  - » N=100 steps=100, N=1000 steps=1000, N = $10^6$ steps = $10^6$
- O(N log N)
  - » N=100 steps=700, N=1000 steps=10000, N=$10^6$ steps = 2 *$10^7$
- O($N^2$)
  - » N=10 steps=100, N=1000 steps=1000000, N=$10^6$ steps = $10^{12}$

# Example

– To see how Big-O works, lets analyse find()

- Best case: myArray[0] is element to find
  - » This is one step, so O(1)
- Worst case: myArray[n-1] is the match, which is n steps
  - » O(N) in Big-O notation
  - » Each step involves multiple CPU instructions, but we aren't concerned with these details, so we don't talk about O(5N)
- Average case: On average, we must go halfway: N/2 steps
  - » O(N) – again, the constant multiplying factor of ½ is irrelevant
- We are mostly interested in the average and worst cases

# Arrays In Code (Java) – Sum of Squares

```
public static void main() {
   double[] squaredVals; // Our array – starts off as null (ie: unallocated)
   double[] sameArray;   // Another array reference, also null
   double val, sumOfSquares;
   int ii, numVals, maxNumVals = 100;  // 100 could've been entered by user

   squaredVals = new double[maxNumVals];    // Allocate array with length=100
   val = ConsoleInput.readDouble("Enter a number (0 to stop): ");

   numVals = 0;            // Counter to track how many elements actually used
   while ((val != 0.0) && (numVals < squaredVals.length)) { // .length==capacity
      squaredVals[numVals] = val * val;       // Store the squared value
      numVals++;
      val = ConsoleInput.readDouble("Enter a number (0 to stop): ");
   }

   sumOfSquares = 0.0;                     // Now sum up all the squared values
   for (ii = 0; ii < numVals; ii++) {      // Only sum up entered values
      sumOfSquares += squaredVals[ii];     // Sum up element values
   }
   System.out.println("Sum of squares is : " + sumOfSquares);

   sameArray = squaredVals;    // sameArray now points at squaredVals array
   sameArray[0] = -1;  // Also affects squaredVals[0] since they are the same!
}
```

# Arrays Pros and Cons

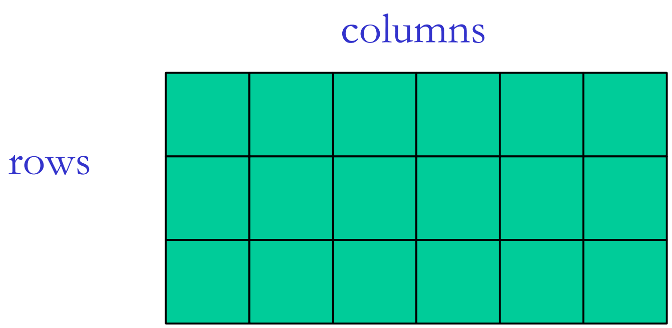
- ☑ Available in *all* programming languages
- ☑ Fast (direct) access to any element in array
  - Indexing is just a small arithmetic operation (arr + idx)
- ☑ No storage overhead – each element exactly fits data
- ☒ Fixed size once allocated – does not grow/shrink
- ☒ Can't insert a new element at the front without shuffling all other elements up by one
  - Same goes for inserting in the middle (sorted) and removing
- ☒ Doesn't track how many elements actually used
  - And when `used < capacity`, you are wasting space

# Multi-dimensional Arrays

- So far we've only talked about 1D arrays
  - ie: a single list of elements
- What about 2D arrays?
  - eg: a matrix in maths is a 2D structure.
  - eg: a 3Mpix digital image is a 2048x1536 array (2D)
    - » Actually, it's 2048x1536x3 since colour images have three channels: Red, Green, Blue
- Or even higher?
  - No reason why we can't have a 3D, 4D, 5D, N-D array

# 2D Arrays

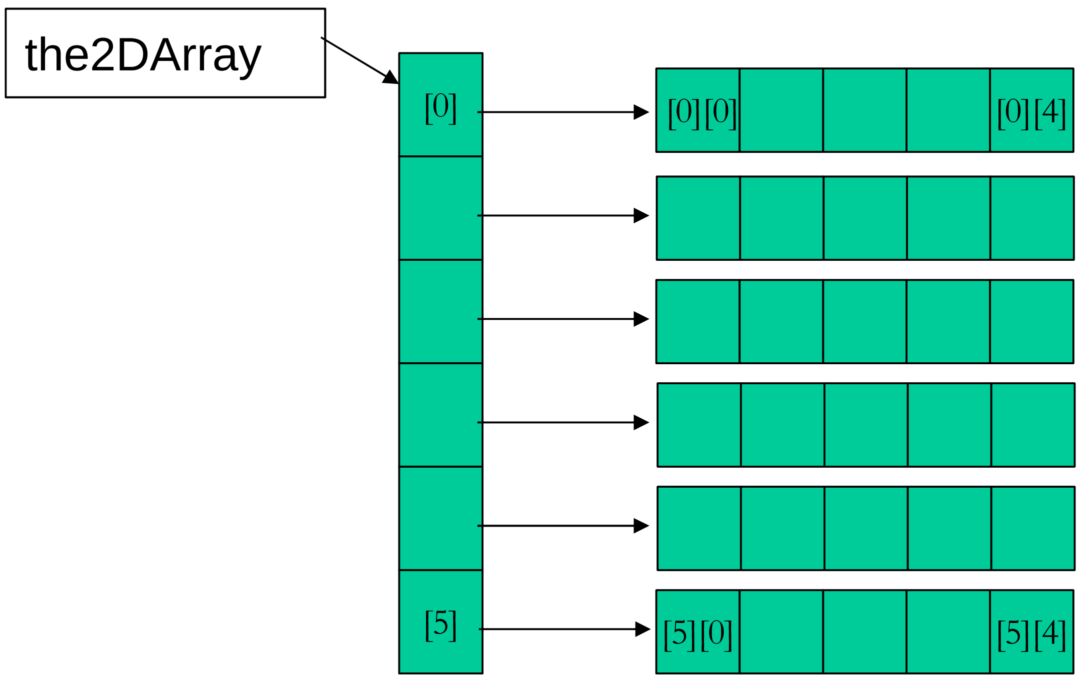
- Declaring 2D arrays is similar to 1D arrays
  - Just add an extra [ ] to the data type:
  - `int [][] a2DArray;`
- Accessing a 2D array is also similar:
  - `a2DArray[row][column]`




  - The above is a 3x6 *matrix* (rows x cols)
  - OR, it is a 6x3 *image* (width x height)
    - » depends on order of processing

# 2D Arrays – What Are They?

- 2D arrays are really an 'array of arrays'
  - Java hides this detail, but C doesn't



# N-D Arrays

- What works with 2D also works for 3D, 4D, .., N-D
  - Just keep adding extra [ ]'s to the data type
  - It can start to get hard to keep track of it all, but just remember that each array dimension works the same
    - » ie: a 4D array is just a list of 3D arrays, each of which are just a list of 2D arrays, which in turn are just a list of 1D arrays.
    - » eg:

```
int[][][] array3D;
int[][] array2D;
int[] array1D;

array2D = new int[3][6]; // Allocate a 3x6 array
array1D = array2D[0];    // Elements of array2D are just 1D arrays
```

# Passing and Returning Arrays

- Arrays can be passed as a parameter to a method, just like any other variable
  - One difference from primitives: passing the array doesn't *copy* the array, it only passes a *reference* to the array
    - » In this way, arrays are more like objects than primitives
  - So if the method changes the passed-in array, it will affect the 'original' array in the calling method
    - » Since the two are in fact the exact same array
- You can also return an array from a method

# Passing and Returning Arrays: Code

```
public static void main() {
   int[] anArray = new int[3];
   int[] copyOfAnArray;

   anArray[0] = 1;                           ← Init anArray to some values
   anArray[1] = -16;
   anArray[2] = 5;

   copyOfAnArray = copyIntArray(anArray);    ← Passing arrays just uses the name, not []
}


public static int[] copyIntArray(int[] arrayToCopy) {      ← Note: data type is int[]
   int[] dupArray;                           ← Array that will hold the copy
   int ii, n;

   n = arrayToCopy.length;
   dupArray = new int[n];
   for (ii = 0; ii < n; ii++) {              ← do it n times, where n is the size of the array
      dupArray[ii] = arrayToCopy[ii];        ← 'deep copy' of array elements
   }

   return dupArray;                          ← Return the array reference
}
```

# Arrays of Objects

– The datatype of each element of an array is the datatype the array was declared as.

```
int[] anIntArray = new int[3];
   anIntArray[1] IS an int
double[] aDoubleArray = new double[3];
   aDoubleArray[1] IS a double
String[] aStringArray = new String[3];
   aStringArray[1] IS a String
```

– This is used just like any variable of the type:

```
aStringArray[1].equals(“Wilma”)
```

# Abstract Data Types: Stacks and Queues

- Arrays are a type of *data structure*
  - They define how to organise data in memory
  - In particular, arrays store a set of elements in a single contiguous block of memory, accessed via an index
- Data structures such as arrays can be useful as they are, but they aren't always a perfect fit
  - Many applications need to access data differently to the array's 'index-update' approach
    - » eg: an order processing queue: take from front, add to rear
  - Problem: an array is really *how a computer operates*
    - » RAM is just one long 1D array (same with disk storage)

# Stacks and Queues

- So there can be a gap between the data structure (how it works) and the *usage* of that structure
- Abstract data types are there to bridge the gap
  - ADT: a set of methods that provide access to data in a way that is natural for the application
  - How the methods manipulate the underlying data structure to achieve this is not the app’s problem
    - » Even what data structure is used is hidden!
  - ADTs make developing applications much easier
    - » Write the ugly details <u>once</u> and wrap it all in nice methods
    - » Lets you later concentrate on the application logic rather than the details of manipulating the data structure

# Stacks and Queues

- Two very common ADTs are stacks and queues
  - Queue: elements taken out in the order they were added
    - FIFO: first-in, first-out (although not all queues are FIFO queues)
  - Stack: data elements are taken out in *reverse* order
    - LIFO: last-in, first-out
  - Elements *must* be taken out in the appropriate order: you can't jump in and grab the 5th element
- Such processing occurs a lot in the real world
  - And we often need to model such processes in software
- But: arrays aren't necessarily the best for implementing these ADTs

# Queue vs Array

- Consider the behaviour of a queue vs an array:
  - Nothing stops you from accessing array element [5]
    - » But a queue should only take the first element each time
  - If you take the first array element [0], element [1] doesn't automatically move to position [0]
    - » So then you have to remember that the 'new-first' element is [1],
    - » or shuffle all the elements up by one yourself
- Solution: *force* the array to behave like a queue
  - Just because it's messy doesn't mean it's impossible
    - » …but it means we only have to CODE AND TEST IT ONCE!
  - If we code it right, using it in the application will simplify (and clarify) the rest of the code enormously

# Stacks

- Let's start with stacks, because they are easier!
- A stack is an ADT that implements a LIFO list
  - Think of a stack of plates – add to top, take from top
  - Some example applications for stacks:
    - » Converting a character string into an int (eg: "10" → 10)
    - » Storing information for method calls
    - » Evaluating a mathematical expression ( We'll see later on)
- Since it's an ADT, we'll first talk about *what* a stack's behaviour is
  - Then we will discuss *how* to implement a stack
    - » In particular: with an array data structure

# Stack Methods

– Being LIFO, a stack has a few obvious methods, with standard names that everyone recognises:
  - push() – add a new item to the top of the stack
  - pop() – take the top-most item from the stack
  - top() – look at the top-most item, but leave it on the stack
    - » Synonym: peek()
  - isEmpty() – check if the stack is empty

– There are also extra methods that often appear
  - isFull() – checks if the stack is full
    - » Arrays can get full, but some data structures don't have this issue
  - count() – number of elements in the stack
    - » Synonyms: size(), numElements() (not too standardised!)

# Stack Implemented with an Array

– Java has a Stack class (in `java.util.*`), but we'll look at our own DSAStack to illustrate the concept
  - Let's create a stack of double values to hold numbers
– The only data structure we know (so far) for storing sets of data is the array … so we'll use arrays
– How are we going to do it?
  - Look for similarities that we can exploit
  - Consider: A stack grows and shrinks on *one side*
  - Similarly, array elements start at [0], and can be added to / removed from the end until the array capacity is reached

# Stacks with Arrays

- So, if we make the *top* of the stack be the *back* of the array, we can grow/shrink without much hassle
  - Counter-intuitive, but simplifies the code a lot!
- The idea is to keep track of the count of elements in the array
  - The element at [count – 1] is then the top of the stack
    - » – 1 because arrays are zero-based in Java, remember!
  - New items then get stored in slot [count]
    - » [count-1] is the top, so [count] is the next unused slot
  - When count == array.length, the stack isFull

# Stack - Pseudocode

```
Class DSAStack
Class fields : stack (double array), count (integer)
Class constant : DEFAULT_CAPACITY ← 100

Default constructor
  alloc stack array with DEFAULT_CAPACITY elements
  count ← 0

Alternate constructor IMPORT maxCapacity (integer)
  alloc stack array with maxCapacity elements
  count ← 0


ACCESSOR getCount IMPORT none EXPORT count

ACCESSOR isEmpty IMPORT none EXPORT empty (boolean)
  empty ← (count = 0)

ACCESSOR isFull IMPORT none EXPORT full (boolean)
  full ← (count = stack length)
```

<continued next slide>

# Stack - Pseudocode (cont.)

```
MUTATOR push IMPORT value EXPORT none
  IF isFull() THEN
    ABORT                          ← ie: throw an exception
  ELSE
    stack[count] ← value
    count ← count + 1
  ENDIF

MUTATOR pop IMPORT none EXPORT topVal
  topVal ← top()
  count ← count - 1

ACCESSOR top IMPORT none EXPORT topVal
  IF isEmpty() THEN
    ABORT
  ELSE
    topVal ← stack[count - 1]
  ENDIF
```

# Application: ReadInt

- From the lecture on recursion we have seen that we need to convert characters read from the keyboard to an integer.
  - We can also achieve this with a stack.

```
create a new intStack
ch = readChar
WHILE ‘0’ <= ch <= ‘9’
   digit = ch – ’0’
   intStack.push<-- digit
   ch = readChar
ENDWHILE

value = 0
powerOfTen = 1

WHILE NOT intStack.isEmpty
   digit = intStack.pop
   value = value + digit * powerOfTen
   powerOfTen *= 10
ENDWHILE
```

# ReadInt

- Compare this with the recursive method
  - more lines of code
  - just as many (more) method calls
  - and now we have to maintain a stack

# Application: Evaluation Maths Equations

- Stacks *really* become useful for non-obvious tasks
  - Evaluation of maths expressions is one of those tasks
- The problem:
  - We normally see equations in the form:

    ```
    (10.3 * (14 + 3.2)) / (5 - 2 * 3)
    ```
  - There are many precedence rules that need to be followed
    - » BIMDAS or BOMDAS
    - » Makes it hard to write code to solve it in the right order

# Infix to Postfix

- Solution: Re-order the equation so that higher precedence operations come before lower ones
  - Plus we get rid of brackets, even nested brackets
  - Then we just need to read it from left-to-right
- How?
  - Normal equations are in what is called 'infix' notation
    - » Unfortunately it's not possible to rewrite equations in infix to get rid of precedence ordering and brackets. Consider:

```
Normal:        (10.3 * (14 + 3.2)) / (5 + 2 - 4 * 3)
Left-to-Right: 14 + 3.2 * 10.3 / -4 * 3 + 5 + 2  (ie: no BIMDAS)
```

    - » Close, but the 10.3 / -4 is wrong – we needed to 'postpone' evaluating it until after the + 2. But with infix we can't postpone

# Postfix

- Solution: use a different notation, postfix
  - Put the operator *after* the operands it applies to (the 'post')
  - Each operator then applies to the two operands that precede the operator
- How does this help?
  - You only evaluate operands once you see an operator
    - Before that, you just keep adding operands to a pile
    - Since the operator must be applied to the *last* two operands (LIFO), your 'pile' is in fact a stack

# Infix vs Postfix Examples

– The original equation in Postfix:

```
Infix: (10.3 * (14 + 3.2)) / (5 + 2 - 4 * 3)
Postfix: 10.3 14 3.2 + * 5 2 + 4 3 * - /
```

– Some simpler examples:

| Infix | Postfix |
|---|---|
| `3 * 4` | `3 4 *` |
| `2 - 4 + 3` | `2 4 - 3 +` |
| `4 + 2 * 3` | `4 2 3 * +` |
| `(4 + 2) * 3` | `4 2 + 3 *` |
| `((2 - 3) / 4 * (1 + 9)) * 2` | `2 3 – 4 / 1 9 + * 2 *` |

# Postfix Properties

– Points to note:
  - The order of the operands is left **unchanged**
  - Operators are listed in **precedence order**
    - » … even the effect of brackets has been taken into account
  - Equal-precedence operators are kept in the infix order
    - » left to right associativity
      - – eg: 2 – 4 + 3 → 2 4 – 3 + NOT 2 4 3 + –
      - – Reason: 2 – 4 is in fact 2 + (-4), so we *must* keep the –ve sign related to the 4: 2 – 4 ≠ 4 – 2

        2 4 3 + – is actually postfix for 2 – (4 + 3)
      - – Same reasoning applies to \ : A \ B ≠ B \ A
      - – + and * aren't so problematic, since A + B = B + A

# Evaluating Postfix

- Evaluating postfix expressions will give some more insight into why it all works
  - We'll discuss infix → postfix conversion a little later
    - » … because it's harder!
- Unsurprisingly, we use a stack in the evaluation
  - Push operands onto stack until an operator is encountered
  - Pop off last two operands and apply the operator to them
    - » Apply the operator *in-order*, not LIFO order (important for –, /)
  - Push the result back on the stack ready for the next op
  - When no more operands/operators are left in the postfix, the answer is the (single) value remaining on the stack

# Postfix Evaluation Example

`Infix: (10.3 * (14 + 3.2)) / (5 + 2 - 4 * 3)`

`Postfix: 10.3 14 3.2 + * 5 2 + 4 3 * - /`

| PFix | Eval Stack Contents | What's Happening? |
|---|---|---|
| 10.3 | 10.3 | <push 10.3> |
| 14 | 10.3 14 | <push 14> |
| 3.2 | 10.3 14 3.2 | <push 3.2> |
| + | 10.3 17.2 | <2 pops> → 14 + 3.2, <push ans> |
| * | 177.16 | <2 pops> → 10.3 * 17.2, <push ans> |
| 5 | 177.16 5 | <push 5> |
| 2 | 177.16 5 2 | <push 2> |
| + | 177.16 7 | <2 pops> → 5 + 2, <push ans> |
| 4 | 177.16 7 4 | <push 4> |
| 3 | 177.16 7 4 3 | <push 3> |
| * | 177.16 7 12 | <2 pops> → 4 * 3, <push ans> |
| - | 177.16 -5 | <2 pops> → 7 - 12, <push ans> |
| / | -35.432 | <2 pops> → 177.16 / -5, <push ans> |
| <end> | -35.432 | <pop> → Final answer |

# Infix to Postfix Conversion

– Converting infix to postfix *also* uses a stack
  - Postfix needs to re-arrange operators into the right place
  - So we need to 'hold on' to operators until we reach the right point in the equation to insert them back in
    - » Remember that operands don't change their order
  - The method behind this is to hold back an operator until we see an equal-or-lower-precedence operator
    - » If the new operator is higher precedence, we have to put it 'on top' of the other operator (in a stack), since it takes precedence
  - Brackets are an extra wrinkle
    - » Approach: treat sub-equations in brackets as if they were isolated from the rest of the equation (because they are!)

# Infix to Postfix Conversion: Algorithm

NOTE: Methods in red must also be implemented, but are fairly straightforward tasks

```
postfix ← empty
WHILE infix has more terms DO
   term ← ParseNextTerm()                          ← Extract next term (operator, operand) from infix eqn

   IF (term = '(') THEN
      opStack.push('(')                            ← '(' gets put straight onto the stack

   ELSE IF (term = ')') THEN
      WHILE (opStack.top ≠ '(') DO                 ← Find corresponding '('
         postfix ← postfix + opStack.pop           ← Pop remaining operators for the bracketed sub-equation
      ENDWHILE
      opStack.pop                                  ← Pop the '(' and discard it

   ELSE IF (term = '+') OR (term = '-') OR (term = '*') OR (term = '\') THEN
      WHILE (NOT opStack.isEmpty) AND (opStack.top ≠ '(') AND
            (PrecedenceOf(opStack.top) >= PrecedenceOf(term)) DO
         postfix ← postfix + opStack.pop           ← Move higher/equal precedence ops to postfix eqn
      ENDWHILE
      opStack.push(term)                           ← Always put the new operator onto the stack

   ELSE                                            ← Term must be an operand if it isn't an operator
      postfix ← postfix + term                     ← Add operand to postfix equation
   ENDIF
ENDWHILE

WHILE (NOT opStack.isEmpty) DO                     ← Pop any remaining operators from the stack
   postfix ← postfix + opStack.pop
ENDWHILE
```

# Infix to Postfix Example

`Infix: (10.3 * (14 + 3.2)) / (5 + 2 - 4 * 3)`

`Postfix: 10.3 14 3.2 + * 5 2 + 4 3 * - /`

| Infix | Postfix So Far | Operator Stack |
|---|---|---|
| ( | | ( |
| 10.3 | 10.3 | ( |
| * | 10.3 | ( * |
| ( | 10.3 | ( * ( |
| 14 | 10.3 14 | ( * ( |
| + | 10.3 14 | ( * ( + |
| 3.2 | 10.3 14 3.2 | ( * ( + |
| ) | 10.3 14 3.2 + | ( * |
| ) | 10.3 14 3.2 + * | <empty> |
| / | 10.3 14 3.2 + * | / |
| ( | 10.3 14 3.2 + * | / ( |
| 5 | 10.3 14 3.2 + * 5 | / ( |
| + | 10.3 14 3.2 + * 5 2 | / ( + |
| 2 | 10.3 14 3.2 + * 5 2 | / ( + |
| - | 10.3 14 3.2 + * 5 2 + | / ( – |
| 4 | 10.3 14 3.2 + * 5 2 + 4 | / ( – |
| * | 10.3 14 3.2 + * 5 2 + 4 | / ( – * |
| 3 | 10.3 14 3.2 + * 5 2 + 4 3 | / ( – * |
| ) | 10.3 14 3.2 + * 5 2 + 4 3 * – | / |
| <end> | 10.3 14 3.2 + * 5 2 + 4 3 * – / | <empty> |

# Postfix Conversion 'Checklist'

– Things to keep in mind:
  - Don't forget to write down the brackets in the infix!
  - New operators ALWAYS go onto the stack
    - » They *never* get put directly onto the postfix expression
    - » The only question is whether to first pop the operator that is *already on the stack* off to the postfix expression
  - Brackets NEVER appear in the postfix
    - » And closing brackets never appear in the operator stack – they are only markers to indicate the end of the sub-equation
  - Remember to pop off any remaining operators at the end of each sub-equation or at the end of the full equation

# FIFO Queues

- A FIFO queue is an ADT implementing a FIFO list
  - Other kinds of queues aren't FIFO, eg: priority queue
- Examples of where FIFO queues are needed
  - Bank transactions: processed in the order they are made
  - Customer orders: first come, first served

# Queue Methods

- Queues (FIFO or otherwise) have the following methods
  - » Note: naming isn't as standardised as it is with stacks
  - enqueue() – add item to the queue
    - » FIFO queues add to the end, priority queues insert in priority order
    - » Synonyms: add(), insert()
  - dequeue() – take item from the front of the queue
    - » Synonyms: remove(), delete()
  - peek() – check the front item, but don't take it off
    - » Synonyms: front()
  - isEmpty() – check if the queue is empty
  - isFull() – check if the queue is full. Optional
  - count() - number of elements in the queue. Optional

# FIFO Queue with an Array

- Unlike stacks, queues grow on one side (the end) and shrink on the other (the front)
  - No synergies with arrays to be taken advantage of here!
- Two options are available:
  - Shuffle queue elements forward when front is dequeued
    - » Exactly like a real-world queue, like at the bank
  - Leave elements as-is and change which index is 'front'
    - » ie: dequeued indexes are no longer used
    - » Circular queue: allow the queue to cycle around the array, so that previously-dequeued indexes can be re-used

# 'Shuffling' vs Circular Queues

- Time Efficiency:
  - Shuffling: every dequeue must move N elements up by 1
  - Circular: Only need to adjust front index – much faster
- Space Efficiency:
  - Both have same space usage: circular queues can just start at idx [5], go thru [length-1] and wrap around to end at [4].
  - But both still have a maximum size (due to fixed-size array)
- Code Complexity:
  - Shuffling: easy to understand, code, and maintain
  - Circular: Dealing with the wrap-around can be tricky – simplify it by storing the count as well as start/end indexes

# FIFO Queue – Pseudocode (Shuffling)

```
Class DSAQueue
Class field : queue (double array), count (integer)
Class constant : DEFAULT_CAPACITY ← 100

Default constructor
  // implement this yourself

Alternate constructor IMPORT maxCapacity (integer)
 // implement this yourself


ACCESSOR getCount IMPORT none EXPORT count

ACCESSOR isEmpty IMPORT none EXPORT empty (boolean)
 // implement this yourself

ACCESSOR isFull IMPORT none EXPORT full (boolean)
 // implement this yourself
```

<continued next slide>

# FIFO Queue – Pseudocode (cont.)

```
MUTATOR enqueue IMPORT value EXPORT none
 // implement this yourself

MUTATOR dequeue IMPORT none EXPORT frontVal
 // implement this yourself

ACCESSOR peek IMPORT none EXPORT frontVal
 // implement this yourself
```

# FIFO Queues - Applications

- Don't worry about the implementation of circular queues
  - Although you can investigate them on your own!
- We'll be exploring an application for queues in the practicals
  - You will need to complete the queue pseudocode and convert it into Java

# Next Week

– Hash Tables
  - another ADT using arrays