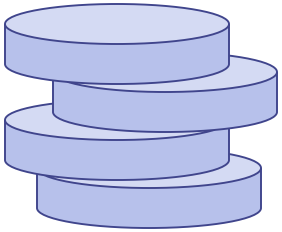
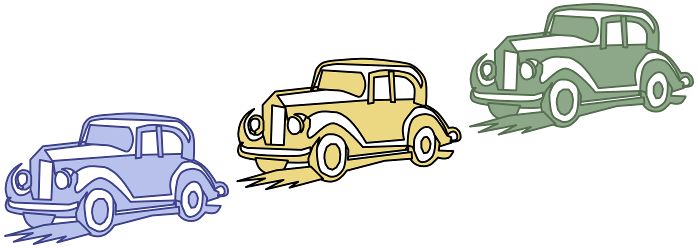
G52ACE
Abstract Data Types: Stacks & Queues





Lecturer: Andrew Parkes

http://www.cs.nott.ac.uk/~pszajp/

2017-18

Abstract Data Types (ADTs)

- An abstract data type (ADT) is an abstraction of a data structure
- **An ADT specifies:**
  - **Data stored**
  - **Operations on the data**
  - **Error conditions associated with operations**
- **An ADT does not specify the implementation itself - hence "abstract"**

Abstract Data Types (ADTs)

- Example: ADT modeling a simple stock trading system
  - The data stored are buy/sell orders
  - The operations supported are
    - order buy(stock, shares, price)
    - order sell(stock, shares, price)
    - void cancel(order)
  - Error conditions:
    - Buy/sell a nonexistent stock
    - Cancel a nonexistent order

Concrete Data Types (CDTs)

- The actual date structure that we use
  - Possibly consists of Arrays or similar
- An ADT might be implemented using different choices for the CDT
  - The choice of CDT will not be apparent from the interface: "data hiding" "encapsulation" – e.g. see 'Object Oriented Methods'
  - The choice of CDT will affect the runtime and space usage – and so is a major topic of this module

ADT & Efficiency

- Often the ADT comes with efficiency requirements expressed in big-Oh notation, e.g.
  - “cancel(order) must be O(1)”
  - “sell(order) must be O(log(|orders|))”
- However, such requirements do not automatically force a particular CDT.
  - The underlying implementation is still not specified
- This is typical of many “library functions”
- Note that such efficiency specifications rely on using the big-Oh family.

ADT and CDT in Java

- Can implement the ADT/CDT split in many ways but might use "interface". Rough example:

- public interface ADT {
      public int f1(); // no implementation!!
      …
  }
- public class1 implements ADT {
      public int f1() { return 99; } // (dummy) implementation
  }
- "interface" and "implements" keywords together give a promise that class1 implements f1()

Vital Skills To Be Developed

- Designing a suitable set of ADTs for a task
  - Design decisions affect whether the ADT is
    "good to program with"
- Designing/selecting suitable CDT(s) for the ADT
  - A simple CDT might allow the ADT to be implemented; e.g. can do almost everything with an array but it might not be efficient
  - Design decisions affect
    - time and space usage
    - maintainability/safety of the code
- Such skills are vital to being a good programmer!

Exercise (from old exam)

- How would you extend an “Array” ADT with an operation to reverse the storage?
- E.g. want to be able to do things like:

```
revArray  a(3)   // create such an array of size 3
 a.set(0)=0
 a.set(1)=1
 a.set(2)=2
 a.get(0)  gives 0
 a.reverse();
 a.get(0)  gives   2
 a.reverse();
 a.get(0)  gives   0
```

Exercise (from old exam)

- How would you extend an “Array” ADT with an operation reverse() that will reverse the storage?
- (Poor) ANSWER

  explicitly reverse the array,
  - E.g. swap the pairs at each end
  - E.g. do a backwards copy to a new array
  - What is the big Oh of this?
    - (ans in class)

Exercise (from old exam)

- How would you extend an “Array” ADT with an operation to reverse the storage?
- Better ANSWER:
  - Do not reverse the storage at all,
  - Just “reverse the way it is accessed” schematically:

```
array x[N]
boolean flipped = false
reverse() {flipped = ! flipped; }
get(i) {if (flipped) return x[N-i];
                  else return x[i];}
```

Exercise (from old exam)

- How would you extend an "Array" ADT with an operation to reverse the storage?

```
array x[N]
boolean flipped = false
reverse() {flipped = ! flipped; }
get(i) {if (flipped) return x[N-i];
                    else return x[i];}
```

The array x is concrete and is used to implement the revArray as an ADT, but the implementation is not direct. Rather it uses the split, so that the array x can be used in 'reversed' form.

revArray: complexity

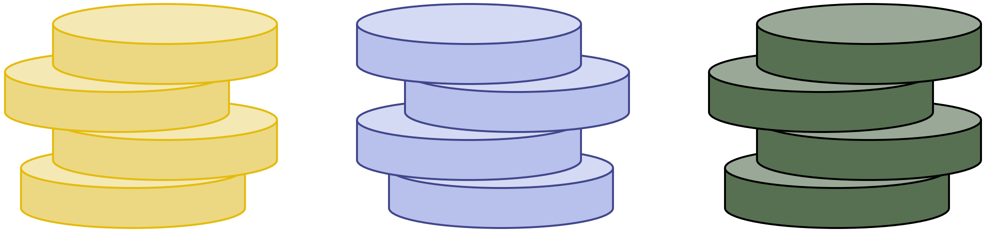
- Compare
  - Direct: use x[] directly
  - Indirect: use flipped and x[]
- reverse():
  - Direct: O(n)   ← EXPENSIVE!
  - Indirect: O(1)
- get/set(i)
  - Direct: O(1)
  - Indirect: O(1)
- Note: get/set on indirect will have a higher constant factor
  - In practice, will need to decide which to use depending on how often reverse() is needed

Expectation:

- Understand and appreciate the point that the split of ADT and CDT allowed reverse() to be implemented
    - very easily
    - more efficiently

  than the direct obvious way

Stacks



The Stack ADT

- The Stack ADT stores arbitrary (references to) objects
- Insertions and deletions follow
  last-in first-out (LIFO)
- Think of a spring-loaded plate dispenser
- Main stack operations:
  - push(object): inserts an element
  - object pop(): removes and returns the last inserted element - other elements are **NOT** accessible
- Auxiliary (“const” methods in C++) stack operations:
  - object top(): returns the last inserted element without removing it
  - integer size(): returns the number of elements stored
  - boolean isEmpty(): returns true iff no elements are stored

Stack Interface in Java

- Java interface corresponding to our Stack ADT
- Requires the definition of EmptyStackException
- Different from the built-in Java class java.util.Stack

```
public interface Stack {

    public void push(Object o);

    public Object pop()
            throws EmptyStackException;

    public Object top()
            throws EmptyStackException;

    public int size();

    public boolean isEmpty();
}
```

Exceptions

- Attempting the execution of an operation of ADT may sometimes cause an error condition, called an exception
- Exceptions are said to be “thrown” by an operation that cannot be executed
- In the Stack ADT, operations pop and top cannot be performed if the stack is empty
- Attempting the execution of pop or top on an empty stack throws an EmptyStackException

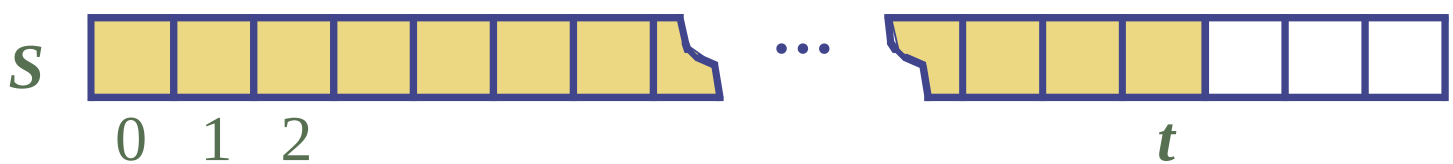
Applications of Stacks

- Direct applications
  - Page-visited history in a Web browser
  - Undo sequence in a text editor
  - Chain of method calls in the Java Virtual Machine
- Indirect applications
  - Auxiliary data structure for algorithms
  - Component of other data structures

Array-based Implementation of Stack ADT

- A simple way of implementing the Stack ADT uses an array as the CDT
- We add elements from left to right
- A variable keeps track of the index of the top element

```
Algorithm size()
   return t + 1

Algorithm pop()
   if isEmpty() then
      throw EmptyStackException
    else
      t ← t − 1
      return S[t + 1]
```



Array-based Stack (cont.)

- The array storing the stack elements may become full
- A push operation will then throw a FullStackException
  - Limitation of the array-based implementation
  - Not intrinsic to the Stack ADT

Algorithm *push*(o)
  if $t = S.length - 1$ **then**
    throw *FullStackException*
  else
    $t \leftarrow t + 1$
    $S[t] \leftarrow o$

Performance and Limitations of Array-based Stack

- Performance
  - Let n be the number of elements in the stack
  - The space used is $O(n)$
  - Each operation runs in time $O(1)$
- Limitations
  - The maximum size of the stack must be defined in advance and cannot be changed dynamically
  - Trying to push a new element into a full stack causes an implementation-specific exception

Array-based Stack in Java

```
public class ArrayStack
      implements Stack {

   // holds the stack elements
   private Object S[ ];

   // index to top element
   private int t = -1;

   // constructor
   public ArrayStack(int capacity) {
      S = new Object[capacity];
   }
```

```
   public Object pop()
         throws EmptyStackException {
      if isEmpty()
         throw new EmptyStackException
            ("Empty stack: cannot pop");
      Object temp = S[t];
      t--;
      return temp;
   }
```

Array-based Stack in Java

```
public class ArrayStack
      implements Stack {

   // holds the stack elements
   private Object S[ ];

   // index to top element
   private int t = -1;

   // constructor
   public ArrayStack(int capacity) {
      S = new Object[capacity];
   }
```

```
   public Object pop()
         throws EmptyStackException {
      if isEmpty()
         throw new EmptyStackException
            ("Empty stack: cannot pop");
      Object temp = S[t];
      // facilitates garbage collection
      S[t] = null;
      t--;
      return temp;
   }
```

Exercise (offline): Finish off this, and implement and test it all.

Remark

- For ADTs is is vital to become familiar with the interface
  - for most ADTs this is relatively straightforward
- Often, the hard but vital skill, is to recognise when an ADT is applicable; or to pick the best ADT for a task

The Queue ADT

- The Queue ADT stores arbitrary objects
- Insertions and deletions follow the **first**-in first-out FIFO scheme
- Insertions are at the rear (tail, end) of the queue and removals are at the front (head) of the queue
- Main queue operations:
  - enqueue(object): inserts an element at the **end** of the queue
  - object dequeue(): removes and returns the element at the **front** of the queue
- Auxiliary ("const") queue operations:
  - object front(): returns the element at the front without removing it
  - integer size(): returns the number of elements stored
  - boolean isEmpty(): indicates whether no elements are stored
- Exceptions
  - Attempting the execution of dequeue() or front() on an empty queue throws an EmptyQueueException

Queue Example

Operation	***Returns***	***State of** Q*
"new"		()
enqueue(5)	–	(5)
enqueue(3)	–	(5, 3)
dequeue()	*5*	(3)
enqueue(7)	–	(3, 7)
dequeue()	*3*	(7)
front()	*7*	(7)
dequeue()	*7*	()
dequeue()	*"error"*	()

Queue Example

Operation	***Returns***	***State of*** *Q*
isEmpty()	*true*	()
enqueue(9)	–	(9)
enqueue(7)	–	(9, 7)
size()	*2*	(9, 7)
enqueue(3)	–	(9, 7, 3)
enqueue(5)	–	(9, 7, 3, 5)
dequeue()	*9*	(7, 3, 5)

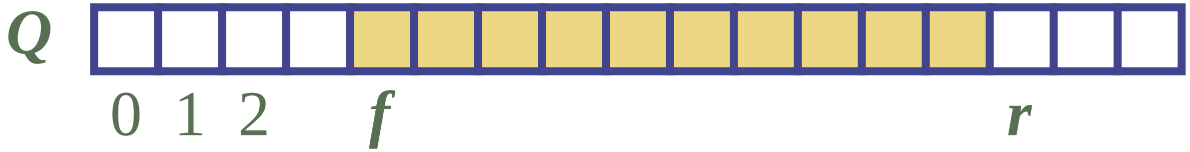
Applications of Queues

- Direct applications
  - Waiting lists, bureaucracy
  - Access to shared resources (e.g., printer)
  - Event queues in GUIs and simulations
- Indirect applications
  - Auxiliary data structure for algorithms
  - Component of other data structures

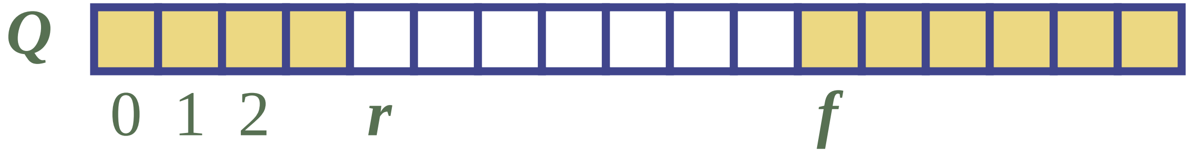
Queue using Array as the CDT

- Use an array of size N in a circular fashion
- Two variables keep track of the front and rear
  - f index of the front element
  - r index immediately past the rear element
- Array location r is kept empty

normal configuration



wrapped-around configuration

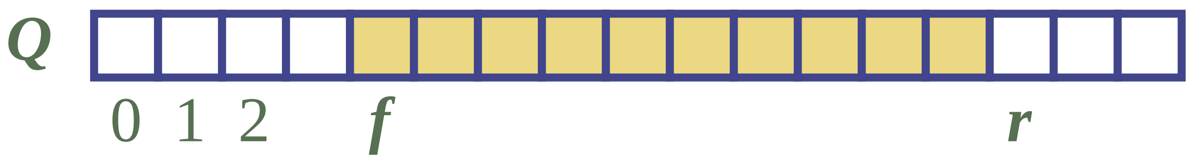


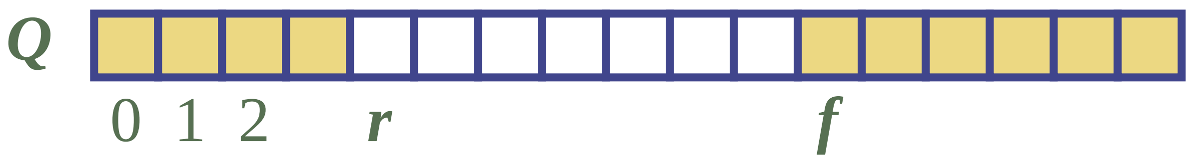
Queue Operations

- If not wrapped then size() is "r-f"
- Due to wrapping, we use the modulo operator ("%")

Algorithm ***size***()
  return $(N - f + r) \bmod N$
  // +N is to keep size positive

Algorithm ***isEmpty***()
  return $(f = r)$



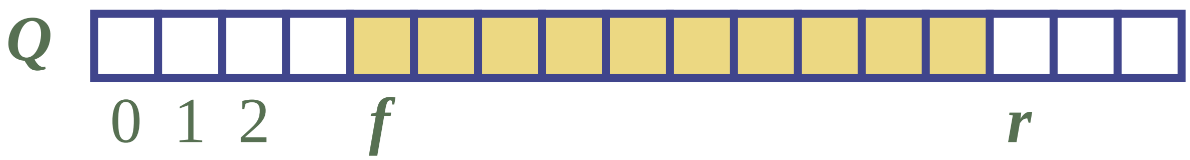


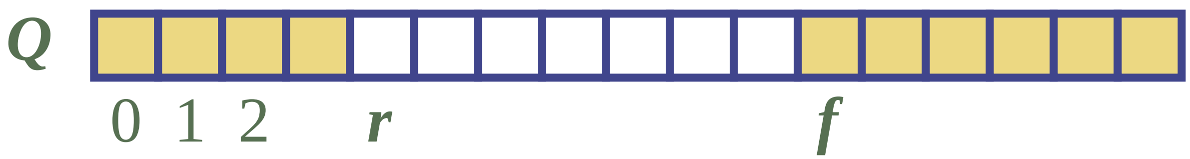
Exercise (offline): compare with r being last element

Queue Operations (cont.)

- Operation enqueue throws an exception if the array is full
- This exception is implementation-dependent

Algorithm *enqueue*(*o*)
  if *size*() = *N* – 1 **then**
    throw *FullQueueException*
  else
    $Q[r] \leftarrow o$
    $r \leftarrow (r + 1) \bmod N$

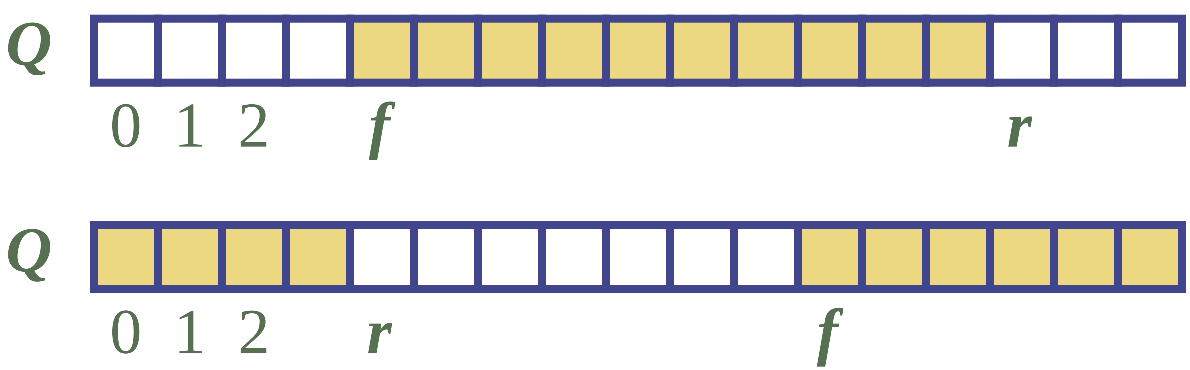




Queue Operations (cont.)

- Operation dequeue throws an exception if the queue is empty
- This exception is specified in the queue ADT

Algorithm *dequeue*()
  if *isEmpty*() **then**
    throw *EmptyQueueException*
  else
    $o \leftarrow Q[f]$
    $f \leftarrow (f + 1) \bmod N$
    return o



Queue Interface in Java

- Java interface corresponding to our Queue ADT
- Requires the definition of class EmptyQueueException
- No corresponding built-in Java class (although LinkedList has all the Queue methods)

```
public interface Queue {

   public int size();

   public boolean isEmpty();

   public Object front()
         throws EmptyQueueException;

   public void enqueue(Object o);

   public Object dequeue()
          throws EmptyQueueException;
}
```

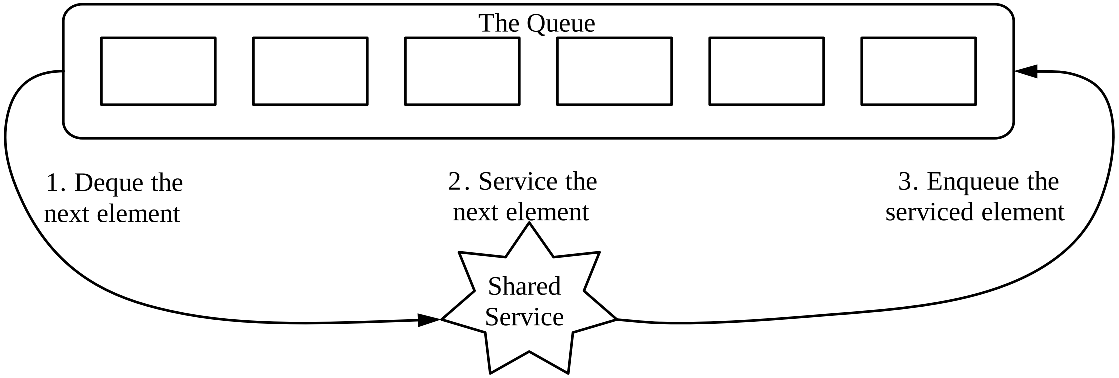
Array-based Queue in Java

```
public class ArrayQueue
      implements Queue {

  // holds the queue elements
 private Object Q[ ];

  // front, end+1, size

 private int f, r, s;

  // constructor
 public ArrayQueue(int capacity) {
      Q = new Object[capacity+1]);
   }
```

```
   public void enqueue(Object o)
        throws FullQueueException {
     if (s == Q.length - 1)
        throw new FullQueueException
           ("The queue is full");
     Q[r] = o;
     r = (r + 1)%(Q.length);
     s++;
   }

// add other methods yourself
```

Application: Round Robin Schedulers

- We can implement a round robin scheduler using a queue, *Q*, by repeatedly performing the following steps:
  1. *e* = *Q*.dequeue()
  2. Service element *e*
  3. *Q*.enqueue(*e*)



So why not just use an Array instead of a Queue ADT !?

- Conceptual Clarity
- Self-Documentation
- Safety of Coding – prevents kludges
- Potential Compiler Optimisations
- Easier to change to a dynamically sizeable data structure
  - Useful when overall memory usage is critical

"Narrow" vs. "Wide" ADTs

- "Narrow": small set of methods
  - E.g. Stack ADT
  - less flexible to use (good or bad?)
  - more flexible to implement, hence maybe more efficient
- "Wide": large set of methods
  - E.g. Java Stack
  - more flexible to use
  - possibly more difficult to implement efficiently

Finding a good balance is a difficult design decision

Summary

- Stacks and Queues are Abstract Data Types
- Stack is LIFO, Queue is FIFO
- Vital data structures, often auxiliary
- Can be implemented efficiently using an array, but this implementation has disadvantages (stack or queue may get full)
- We'll see other implementations (using lists) next

Example Problem: Parentheses Matching

Algorithm ParenMatch(X,n):

Input: An array X of n tokens, each of which is a grouping symbol "(", ")", "[", "]", "{" or "}"

Output: **true** if and only if all the grouping symbols in X match

Exercise

- What is the shortest string that has correctly matched parentheses?

- <answered in lecture>

- Relevance:
  Designing test cases for your algorithms

Example Problem: Parentheses Matching

- Each “(”, “{”, or “[” must be paired with a matching “)”, “}”, or “[”
  - correct: ( )(( )){([( )])}
  - incorrect: ((( )(( )){([( )])}
  - incorrect:)(( )){([( )])}
  - incorrect: ({[ ])}
  - incorrect: (
- Exercise (online): How would you implement “parentheses matching”? Write down a sketch of one idea (not necessarily a good one!)

Parentheses Matching: Scan & Reduce

Example: ( )(( )){([( )])}

- Observation: there are "matched pairs" e.g. "()"
- If these are removed the correctness (or not) will not be changed
- Corresponding (sketch of) algorithm:

    Repeat:

        scan along the string

        remove first matched pair found

Example of “Scan & reduce”

Input: s = “([()])”

1. scan left to right; find & remove the “()”
   - s= “([ ])”
2. scan left to right; find & remove the “[]”
   - s= “()”
3. scan left to right; find & remove the “()”
   - s= “”

Return true as the empty string is correct.

Return false if string is non-empty but no pair is found, e.g. “({) }”

Exercise: Spot the inefficiency?

Example of "Scan & reduce"

Input: s = "(((((( ))))))"

1. scan left to right; find & remove the "()"
   - s= "((((( )))))"
2. "scan left to right" // needs a full scan?

- Observe: we do not need to rescan from the beginning, only from where we removed the "()"
- Idea: Keep track of the location of the "working region", and do not need to access outside that region
- Suggests: keep the "characters scanned so far" on a Stack

Better Algorithm:

- Basic Idea: Read left to right, but try to check matching as we proceed; e.g.
  - When see a matched “open-close” pair we want to be able to “drop the pair”
  - When we see a “close” without an appropriate open then report a mismatch
  - When see an “open” then “just remember it”
- As long as we are “dropping matching pairs recursively” then on seeing a “close” we don’t need to look at anything other than the last remembered symbol
  - Implies consider using a “Stack”

Example

Input: ({ [ ]) }

1. (ok
2. ({ ok
3. ({ [ok
4. ({ [with] is ok, but can reduce to
5. ({
6. ({) mismatch; close bracket “)” with no adjacent opener “(“ – this cannot be the start of any legal string

Parentheses Matching Algorithm

```
Let S be an empty stack
for i=0 to n-1 do
    if X[i] is an opening grouping symbol then
        S.push(X[i])
    else if X[i] is a closing grouping symbol then
        if S.isEmpty() then
            return false // nothing to match with
        if S.pop() does not match the type of X[i] then
            return false // wrong type
if S.isEmpty() then
    return true // every symbol matched
else
    return false // some symbols were never matched
```

Remark

Algorithm development often follows the pattern:

1. Write down some algorithm with little or no concern for efficiency
2. Study the algorithm to spot inefficiencies when it runs
3. Try to fix the inefficiencies – e.g. by choosing appropriate data structures

Moral: do not be afraid to start from a simple algorithm, then revise it

Exercises (offline)

- For Scan-Reduce and Stack-Based Parentheses matching:
  - Implement both
    - Experimentally compare them
  - Find and compare their big-Oh behaviours for both time and space usage
- (Or at least make sure that you would know how to do these if you were forced ☺)