CE204 Data Structures and Algorithms

David Richerby

University of Essex

Spring term 2021

Introduction

Learning outcomes

At the end of this module, you will be able to:

1. demonstrate an understanding of core data types such as stacks, queues, trees and graphs;
2. implement core data types in Java and write programs that use them efficiently;
3. reason about the time and space complexity of programs;
4. demonstrate knowledge of commonly used algorithms;
5. explain the main concepts of computability and how some problems have no algorithmic solution.

Learning and teaching methods

- One 2hr lecture each week (Monday 4–6pm).
- One 2hr lab each week (Tuesday 2–4pm or Thursday 4-6pm).
- Self study and practice.
- Resources on Moodle.
- Labs can be done on your own computer: you just need a Java development environment (e.g., IntelliJ).
- My office hours: TBA.
- Second module supervisors: Mike Sanderson and Yunfei Long.

Assessment

- 70% exam (online, 2hr exam + 1hr submission)
- 10% assessed exercise 1 (due week 21)
- 10% progress test (week 21)
- 10% assessed exercise 2 (due week 25)

Textbooks

- R. Sedgewick and K. Wayne, *Algorithms* (4th edition, Pearson, 2011)
- M. A. Weiss, *Data Structures and Algorithm Analysis in Java* (2nd Edition, Pearson/Addison-Wesley, 2007)

Both available electronically in the library; no need to buy.

If you want to buy, earlier editions are fine (much cheaper second-hand).

Lots of other good books, e.g., Cormen, Rivest, Leiserson and Stein, *Introduction to Algorithms*; Kleinberg and Tardos, *Algorithm Design*; Aho, Hopcroft and Ullman, *Data Structures and Algorithms*; Skeina, *The Algorithm Design Manual*.

I don't recommend Knuth's books.

What is a data structure?

- An organization of data in a computer memory that enables certain operations to be performed on that data.
- E.g., arrays, lists, strings and even numerical datatypes.

What is an algorithm?

- A precise, finite sequence of finite instructions for performing a particular task ("recipe").
- Every computer program is an algorithm.
- Conversely, the Church–Turing thesis proposes that every algorithm is a computer program.

Why study DS&A

"Who cares? Java's libraries can do (nearly) all this stuff!"

- Not all languages have big libraries.
- Understanding the tools available helps you choose the right one.
- Often, you need custom tools – "I need something like an X that can also do Y."
- Library implementations often do more than you need, so are inefficient: e.g.,
  - thread-safety is slow;
  - storing `int`s as `Integer`s doubles space needed.
- Often useful for job interviews!

Algorithm analysis

- Analysis techniques will help you understand how your code will behave on large inputs.
- Suppose your program needs to sort a list of million numbers.
    - Writing the code really carefully might make it twice as fast.
    - So would waiting 18 months and buying a faster computer.
    - Replacing a bad sorting algorithm with a good one could make it 10,000 times faster.

Notes on code

- All code from lectures available on Moodle.
- Code is *not* production-quality:
  - minimal error checking;
  - little to no use of exceptions;
  - often makes convenient assumptions about inputs.
- However, code should be correct – yell at me if it isn't!
- Primary goal is to teach you DS&A, not programming.

Abstract Data Types

What is an array?

A collection of values of the same type, each identified by an integer index and stored in consecutive memory locations, so that the location of each element can be computed from its index.

index	0	1	2	$\cdots$	$n-1$
value	$\star$	$\star$	$\star$	$\cdots$	$\star$

What is an array, behaviourally?

As users of modern programming languages, we don't normally care about these precise implementation details.

We can:

- create a new array,
- set the value of the `ith` element,
- get the value of the `ith` element.

Java interfaces

Suggestive of Java interfaces:

```
interface ObjectArray {
    void create (int length);
    void set (int index, Object value);
    Object get (int index);
}
```

A valid implementation (?)

```
class UselessArray implements ObjectArray {
    void create (int length) { return; }

    void set (int index, Object value) {
        System.out.println ("Sure, I did that.");
    }

    Object get (int index) {
        System.out.println ("Oops, I forgot what you said.");
        return null;
    }
}
```

Abstract data types (ADTs)

To specify a data structure, we need more than just an interface.

Abstract datatypes combine:

- an interface – the available operations;
- a behavioural description of these operations;
- requirements on running time.

ADTs say nothing about implementation or language.

ADTs were proposed by Barbara Liskov (one of the originators of OOP) and Stephen Zilles.

Arrays as an ADT

Operations. `create`, `set`, `get` as before.

Behaviour. For any integer value $i \in \{0, \ldots, \texttt{length} - 1\}$, `get(`$i$`)` returns the `Object` argument of the most recent call `set(`i`, something)`.

```
    a.create (10);
    a.set (4, "Hello");
    // arbitrary code with NO calls to a.set (4, ...)
    a.get (4); // returns "Hello"
```

Running time of `set` and `get` does not depend on the length of the array.

Abstract vs concrete

Concrete datatypes are actual datatypes in actual languages, e.g., `int`, `double[][]`, `class MyClass { ... }` in Java.

Concrete datatypes defined entirely by their implementation.

ADTs are implemented as concrete datatypes.

Implementing ADTs

In Java, ADTs are implemented as classes.

Application programmers should be able to switch to a different implementation of an ADT with

- minimal code changes;
- no change in behaviour.

ADT implementers should only allow users to access data through the defined operations (e.g., fields and helper methods should be private).

Abstract data types – summary

- Abstract data types (ADTs) specify a datatype by defining:
  - the operations on the data,
  - the behaviour of the operations and
  - their running times.
- We defined arrays as an ADT as an example.

Linear Data Structures

Stacks

A stack of books on a table. We can

- add another book to the top;
- remove the top book from the stack.

We can't access books below the top one.

Access is last in, first out.

The stack ADT

Operations and behaviour.

- `void create()` makes a new empty stack.
- `boolean isEmpty()` returns whether the stack is empty.
- `int length()` returns the number of items on the stack.
- `void push(String s)` puts `s` on top of the stack.
- `String pop()` removes and returns the top item of the stack.

Running time of all operations is independent of the stack's size.

Java interface

```
interface StackADT {
    boolean isEmpty ();
    int length ();
    void push (String s);
    String pop ();
}
```

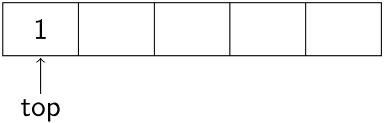
For all our Java implementations, `create()` will be done by the constructor.

Implementation

Store the stack in an array, with the bottom at index 0 (left).

Implementation

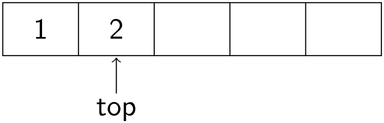
Store the stack in an array, with the bottom at index 0 (left).



```
push(1);
```

Implementation

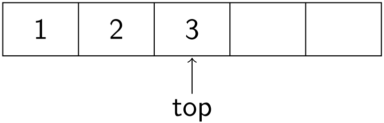
Store the stack in an array, with the bottom at index 0 (left).



```
push(1); push(2);
```

Implementation

Store the stack in an array, with the bottom at index 0 (left).



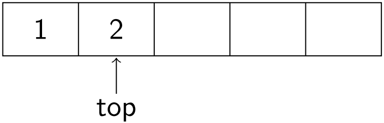
1 2 3

top

```
push(1); push(2); push(3);
```

Implementation

Store the stack in an array, with the bottom at index 0 (left).



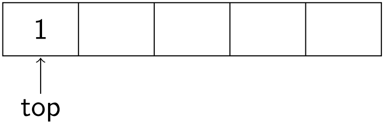
1	2			

top

```
push(1); push(2); push(3); pop();
```

Implementation

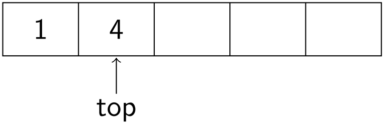
Store the stack in an array, with the bottom at index 0 (left).



```
push(1); push(2); push(3); pop(); pop();
```

Implementation

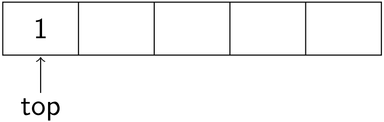
Store the stack in an array, with the bottom at index 0 (left).



```
push(1); push(2); push(3); pop(); pop(); push(4);
```

Implementation

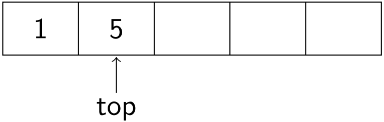
Store the stack in an array, with the bottom at index 0 (left).



```
push(1); push(2); push(3); pop(); pop(); push(4); pop();
```

Implementation

Store the stack in an array, with the bottom at index 0 (left).



```
push(1); push(2); push(3); pop(); pop(); push(4); pop();
push(5);
```

Java implementation (1)

```
class Stack implements StackADT {
    private int length = 0;
    private String[] items = new String[10];

    boolean isEmpty() { return length == 0; }
    int length()      { return length;      }

    ...
}
```

Java implementation (2)

```java
class Stack implements StackADT {
    private int length = 0;
    private String[] items = new String[10];
    ...

    void push (String s) {
        items[length] = s;
        length++;
    }

    String pop () {
        length--;
        return items[length];
    }
}
```

Code review!

- No error checking – `push()` on full stack and `pop()` on empty stack cause exceptions and invalidate `length`.
- Limit of ten items is arbitrary and not part of the ADT.

Java implementation (3)

pop() is easy to fix.

```
    String pop() {
        if (isEmpty())
            return null;
        else {
            length--;
            return items[length];
        }
    }
```

Java implementation (4)

We fix `push()` by extending the array if it's full.

```
void Push (String s) {
    if (length == items.length) {
        String[] newItems = new String[length*2];
        System.arrayCopy (items, 0, newItems, 0, length);
        items = newItems;
    }
    items[length] = s;
    length++;
}
```

Java implementation (4)

We fix push() by extending the array if it's full.

```
    void Push (String s) {
        if (length == items.length) {
            String[] newItems = new String[length*2];
            System.arrayCopy (items, 0, newItems, 0, length);
            items = newItems;
        }
        items[length] = s;
        length++;
    }
```

System.arrayCopy (A, x, B, y, z) copies z consecutive entries from array A to array B, starting reading from A[x] and writing to B[y]. Above, it copies the whole of items to the first half of newItems.

Code review! (2)

- The array grows but never shrinks. A better implementation would halve the array when it's a quarter full.
- Running time requirements not met: `push()` takes time proportional to `length` because of `arrayCopy()`.

Actually, the run-time is not as bad as it looks.

Analysis of `push()` (1)

Suppose the initial array length is 1 (!) and we do 2^n `push`es.

First push fits in the array and fills it.
Second push copies one element and new array has length 2.
Third push copies two elements and new array has length 4.
Fourth push fits in the array.
Fifth push copies four elements and new array has length 8.
Sixth, seventh, eighth pushes fit.
Ninth push copies eight elements and new array has length 16.

Analysis of `push()` (1)

Suppose the initial array length is 1 (!) and we do 2^n `pushes`.

First push fits in the array and fills it.
Second push copies one element and new array has length 2.
Third push copies two elements and new array has length 4.
Fourth push fits in the array.
Fifth push copies four elements and new array has length 8.
Sixth, seventh, eighth pushes fit.
Ninth push copies eight elements and new array has length 16.

General case: $(1+2^i)$th push copies 2^i elements.

Total number of copied elements is $1+2+4+\cdots+2^{n-1} = 2^n - 1$.

On average, each push copies $\frac{2^n-1}{2^n} < 1$ elements.

On average, pushes take constant time.

Analysis of push() (2)

Do we have to double the array each time?

Suppose initial length is 10 and we add 10 more cells when full.
Suppose we do $10n$ pushes.

11th, 21st, $31st$, $\ldots$ pushes copy 10, 20, 30, $\ldots$ elements.

General case: $(1 + 10i)$th push copies $10i$ elements.

Total number of copied elements is

$$10 + 20 + 30 + \cdots + 10(n-1) = \tfrac{1}{2}(n-1)(10 + 10(n-1)) = 5n(n-1)\,.$$

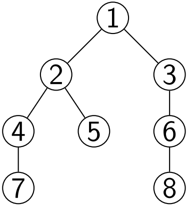
On average, each push copies $\frac{5n(n-1)}{10n} = (n-1)/2$ elements.

The average cost is not constant: it depends on how many pushes we do.

Uses of stacks

- Remembering something to come back to later.
- Implementing function/method calls – stack used to remember, e.g., local variables of caller.
- Depth-first search is “exploring a tree with a stack”.

Depth-first search



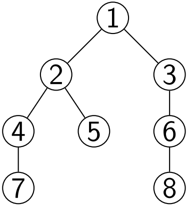
Visited:

Current:

Stack:

```
push (start node)
while (!isEmpty) {
    pop node
    if node is target
        return found
    push all children
}
return not-found
```

Depth-first search



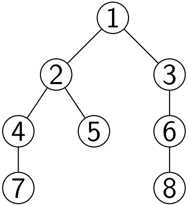
Visited:

Current:

Stack: 1

```
▷ push (start node)
  while (!isEmpty) {
      pop node
      if node is target
          return found
      push all children
  }
  return not-found
```

Depth-first search



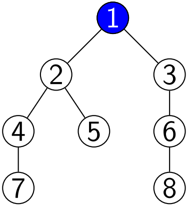
Visited:

Current:

Stack: 1

```
   push (start node)
▷  while (!isEmpty) {
       pop node
       if node is target
           return found
       push all children
   }
   return not-found
```

Depth-first search



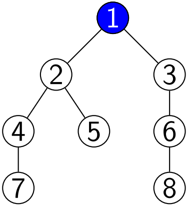
Visited: 1

Current: 1

Stack:

```
   push (start node)
   while (!isEmpty) {
▷      pop node
       if node is target
           return found
       push all children
   }
   return not-found
```

Depth-first search



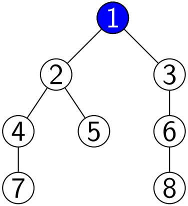
Visited: 1

Current: 1

Stack: 2, 3

```
  push (start node)
  while (!isEmpty) {
       pop node
       if node is target
           return found
▷      push all children
  }
  return not-found
```

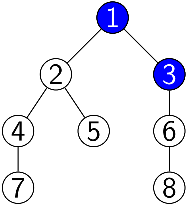
Depth-first search



Visited:	1
Current:	1
Stack:	2, 3

```
  push (start node)
▷ while (!isEmpty) {
      pop node
      if node is target
          return found
      push all children
  }
  return not-found
```

Depth-first search



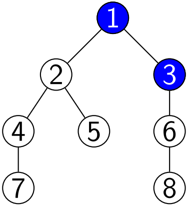
Visited: 1, 3

Current: 3

Stack: 2

```
  push (start node)
  while (!isEmpty) {
▷     pop node
      if node is target
          return found
      push all children
  }
  return not-found
```

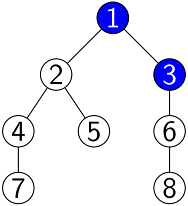
Depth-first search



Visited:	1, 3
Current:	3
Stack:	2, 6

```
   push (start node)
   while (!isEmpty) {
        pop node
        if node is target
            return found
▷       push all children
   }
   return not-found
```

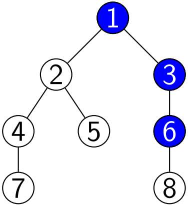
Depth-first search



Visited:	1, 3
Current:	3
Stack:	2, 6

```
  push (start node)
▷ while (!isEmpty) {
      pop node
      if node is target
          return found
      push all children
  }
  return not-found
```

Depth-first search



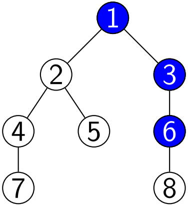
Visited: 1, 3, 6

Current: 6

Stack: 2

```
  push (start node)
  while (!isEmpty) {
▷     pop node
      if node is target
          return found
      push all children
  }
  return not-found
```

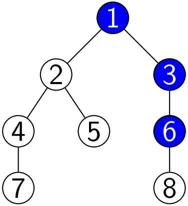
Depth-first search



Visited:	1, 3, 6
Current:	6
Stack:	2, 8

```
   push (start node)
   while (!isEmpty) {
        pop node
        if node is target
            return found
▷       push all children
   }
   return not-found
```

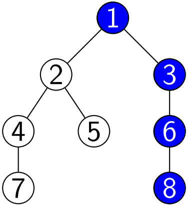
Depth-first search



Visited:	1, 3, 6
Current:	6
Stack:	2, 8

```
   push (start node)
▷  while (!isEmpty) {
        pop node
        if node is target
            return found
        push all children
   }
   return not-found
```

Depth-first search



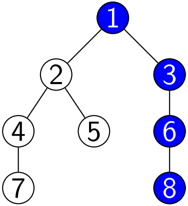
Visited: 1, 3, 6, 8

Current: 8

Stack: 2

```
  push (start node)
  while (!isEmpty) {
▷     pop node
      if node is target
          return found
      push all children
  }
  return not-found
```

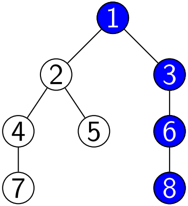
Depth-first search



Visited:	1, 3, 6, 8
Current:	8
Stack:	2

```
    push (start node)
    while (!isEmpty) {
        pop node
        if node is target
            return found
▷       push all children
    }
    return not-found
```

Depth-first search



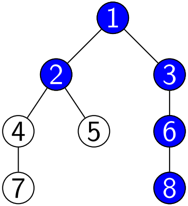
Visited: 1, 3, 6, 8

Current: 8

Stack: 2

```
  push (start node)
▷ while (!isEmpty) {
      pop node
      if node is target
          return found
      push all children
  }
  return not-found
```

Depth-first search



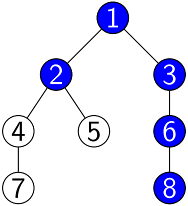
Visited: 1, 3, 6, 8, 2

Current: 2

Stack:

```
    push (start node)
    while (!isEmpty) {
▷        pop node
         if node is target
             return found
         push all children
    }
    return not-found
```

Depth-first search



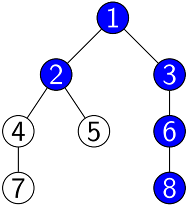
Visited: 1, 3, 6, 8, 2

Current: 2

Stack: 4, 5

```
    push (start node)
    while (!isEmpty) {
         pop node
         if node is target
             return found
▷        push all children
    }
    return not-found
```

Depth-first search



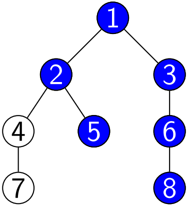
Visited: 1, 3, 6, 8, 2

Current: 2

Stack: 4, 5

```
  push (start node)
▷ while (!isEmpty) {
      pop node
      if node is target
          return found
      push all children
  }
  return not-found
```

Depth-first search



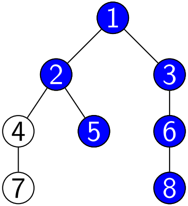
Visited: 1, 3, 6, 8, 2, 5

Current: 5

Stack: 4

```
    push (start node)
    while (!isEmpty) {
▷       pop node
        if node is target
            return found
        push all children
    }
    return not-found
```

Depth-first search



Visited: 1, 3, 6, 8, 2, 5

Current: 5

Stack: 4

```
   push (start node)
   while (!isEmpty) {
        pop node
        if node is target
            return found
▷       push all children
   }
   return not-found
```

Stacks – summary

- A stack is a last-in-first-out memory.
- Operations are `push` (store) and `pop` (retrieve).
- The stack can grow to any length but push (on average) and pop (always) take the same amount of time.
- Uses: function call implementation, depth-first search.

The queue ADT

Operations and behaviour.

- `void create()` makes a new empty queue.
- `boolean isEmpty()` returns whether the queue is empty.
- `int length()` returns the number of items in the queue.
- `void add(String s)` puts `s` at the back of the queue.
- `String remove()` removes and returns the front item of the queue.

Running time of all operations is independent of the queue's size.

Java interface

```
interface QueueADT {
    boolean isEmpty ();
    int length ();
    void add (String s);
    String remove ();
}
```

`create()` will be done by the constructor.

Java interface

```
interface QueueADT {
    boolean isEmpty ();
    int length ();
    void add (String s);
    String remove ();
}
```

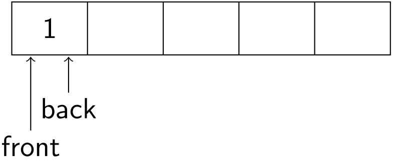
```
interface StackADT {
    boolean isEmpty ();
    int length ();
    void push (String s);
    String pop ();
}
```

`create()` will be done by the constructor.

Interface is identical to Stack ADT except for renaming.

Implementation with arrays

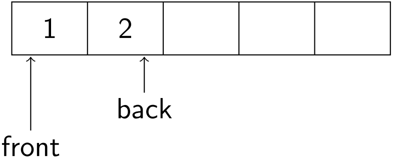
Queues can be implemented using arrays, similar to stacks.



```
add(1);
```

Implementation with arrays

Queues can be implemented using arrays, similar to stacks.



1

2

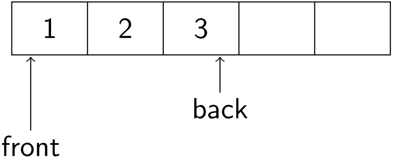
back

front

```
add(1); add(2);
```

Implementation with arrays

Queues can be implemented using arrays, similar to stacks.



1	2	3		

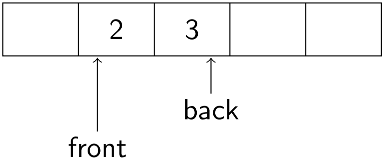
back

front

```
add(1); add(2); add(3);
```

Implementation with arrays

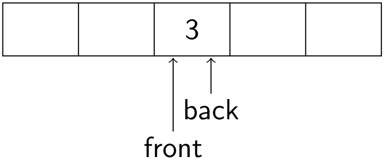
Queues can be implemented using arrays, similar to stacks.



`add(1); add(2); add(3); remove();`

Implementation with arrays

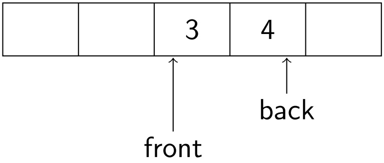
Queues can be implemented using arrays, similar to stacks.



```
add(1); add(2); add(3); remove(); remove();
```

Implementation with arrays

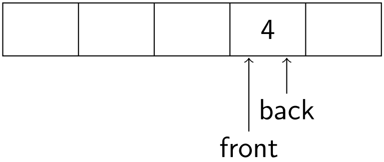
Queues can be implemented using arrays, similar to stacks.



```
add(1); add(2); add(3); remove(); remove(); add(4);
```

Implementation with arrays

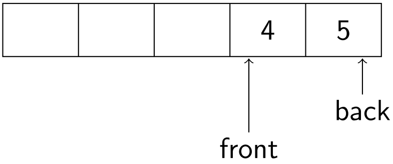
Queues can be implemented using arrays, similar to stacks.



```
add(1); add(2); add(3); remove(); remove(); add(4);
remove();
```

Implementation with arrays

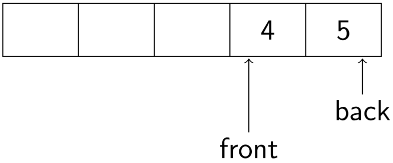
Queues can be implemented using arrays, similar to stacks.



```
add(1); add(2); add(3); remove(); remove(); add(4);
remove(); add(5);
```

Implementation with arrays

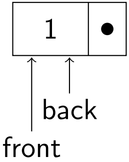
Queues can be implemented using arrays, similar to stacks.



`add(1); add(2); add(3); remove(); remove(); add(4); remove(); add(5);`

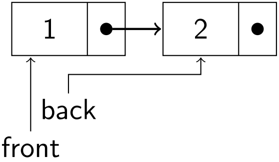
Now things get a bit fiddly...

Implementation with references



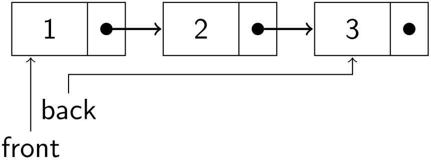
```
add(1);
```

Implementation with references



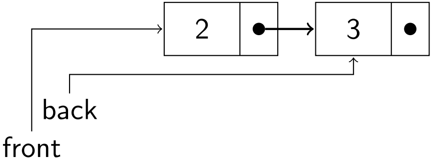
```
add(1); add(2);
```

Implementation with references



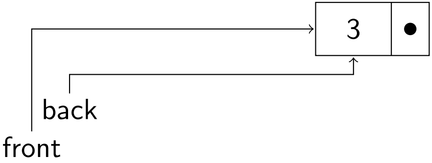
```
add(1); add(2); add(3);
```

Implementation with references



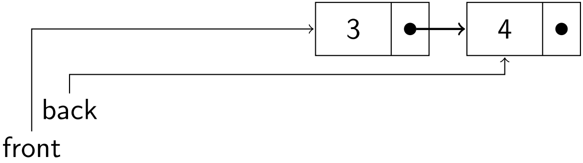
```
add(1); add(2); add(3); remove();
```

Implementation with references



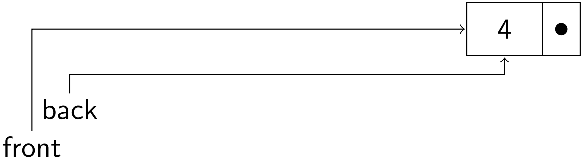
```
add(1); add(2); add(3); remove(); remove();
```

Implementation with references



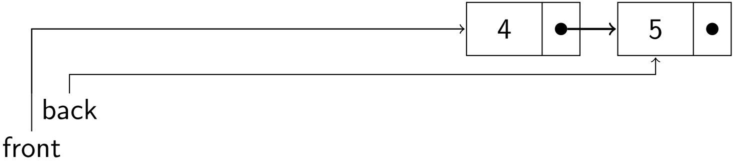
```
add(1); add(2); add(3); remove(); remove(); add(4);
```

Implementation with references



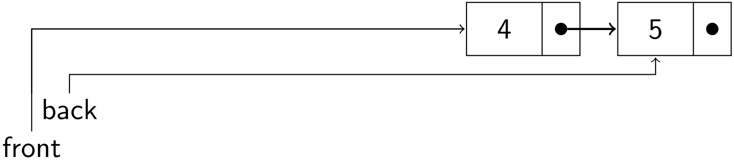
```
add(1); add(2); add(3); remove(); remove(); add(4);
remove();
```

Implementation with references



```
add(1); add(2); add(3); remove(); remove(); add(4);
remove(); add(5);
```

Implementation with references



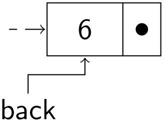
`add(1); add(2); add(3); remove(); remove(); add(4); remove(); add(5);`

We can also implement stacks in a similar way.

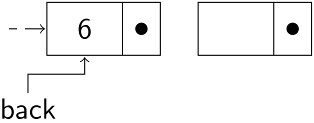
Java implementation (1)

```
public class Queue {
    private class Item {
        String value;
        Item next;

        Item (String value) {
            this.value = value;
            this.next = null;
        }
    }

    private Item front = null;
    private Item back = null;
    private int size = 0;
    ...
}
```

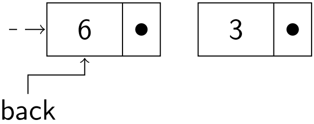
Java implementation (2)



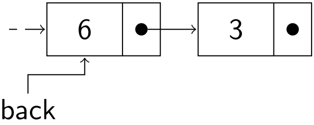
Java implementation (2)



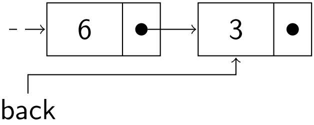
Java implementation (2)



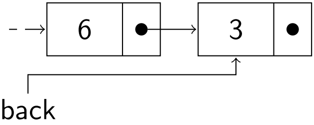
Java implementation (2)



Java implementation (2)

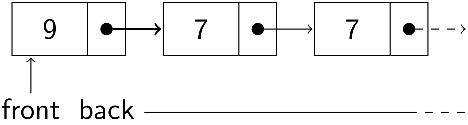


Java implementation (2)

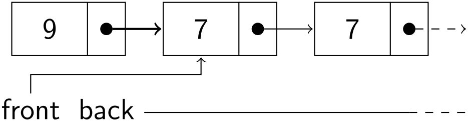


```
public void add (String s) {
    if (isEmpty()) {
        front = back = new Item(s);
    } else {
        back.next = new Item(s);
        back = back.next;
    }
    size++;
}
```

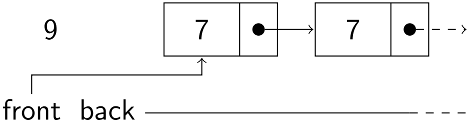
Java implementation (3)



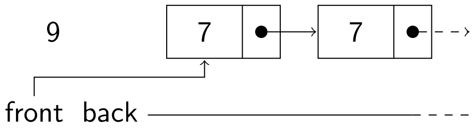
Java implementation (3)



Java implementation (3)



Java implementation (3)



```
public String remove () {
    if (isEmpty())
        return null;

    String s = front.value;
    front = front.next;
    if (front == null)
        back = null;
    size--;
    return s;
}
```

Code review!

Is back redundant? We can always find the back of the queue by following .next references from front.

```
    private Item getBack () {
        if (isEmpty())
            return null;

        Item cur = front;
        while (cur.next != null)
            cur = cur.next;
        return cur;
    }
```

Implementation trade-offs (1)

Advantages of using `getBack()`:

- saves a small (tiny!) amount of memory;
- code for `add()` and `remove()` simplified slightly.

Disdvantages of relying on `getBack()`:

- every call to `add()` must find the back of the queue;
- this takes time proportional to the queue's length.

Conclusion: relying on `getBack()` breaks the queue ADT's requirement that `add()` runs in constant time.

Implementation trade-offs (2)

`back` reference vs `getBack()`:

- the reference uses a tiny bit more memory;
- `getBack()` is much slower.
- Which is better? Almost certainly the reference.

Array vs references for stack/queue:

- array wastes a lot of space unless mostly full;
- references use extra space for every item (significant if items are small, e.g., `ints`);
- references fast for every `add()`; arrays even faster most of the time but very slow if need to grow the array.
- Which is better? Neither. It depends.

Implementation trade-offs (3)

Doubling vs additive increase for array stack/queues:

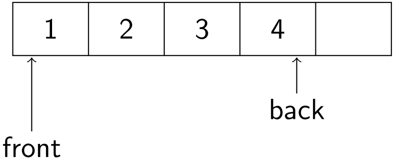
- Doubling uses more memory;
- Additive increase spends too much time copying.
- Which is better? If the stack is small, it doesn't matter; if it's big, additive increase will be very slow.
- Conclusion: double it.
- If you can't afford the memory to double, multiply by a smaller amount instead, e.g., 1.5 or 1.25.

Uses of queues

- Whenever things need to be processed in the order they're created/discovered.
- Network switches keep a queue of packets waiting to be despatched on each interface.
- Queueing theory: a whole branch of mathematics modelling the behaviour of queues.
- Event queues in GUIs.
- Breadth-first search is "exploring a tree with a queue".

Circular buffers: a specialized queue

- Network switches need very fast queues for packets waiting to be sent.
- Compromise: queues are fixed size; packets discarded if no room.



1	2	3	4	

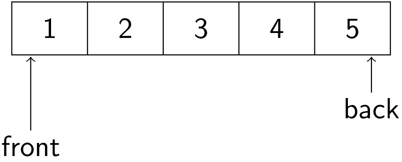
back

front

```
add(1); ...  add(4);
```

Circular buffers: a specialized queue

- Network switches need very fast queues for packets waiting to be sent.
- Compromise: queues are fixed size; packets discarded if no room.



1	2	3	4	5

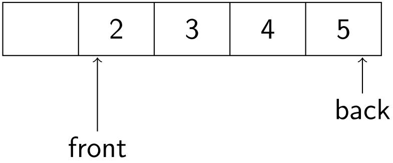
back

front

```
add(1);  ...  add(4); add(5);
```

Circular buffers: a specialized queue

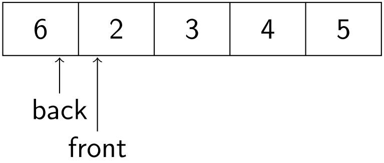
- Network switches need very fast queues for packets waiting to be sent.
- Compromise: queues are fixed size; packets discarded if no room.



```
add(1); ...  add(4); add(5); remove();
```

Circular buffers: a specialized queue

- Network switches need very fast queues for packets waiting to be sent.
- Compromise: queues are fixed size; packets discarded if no room.



6	2	3	4	5

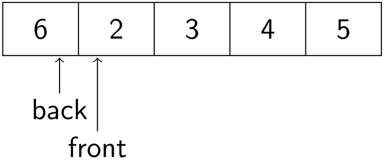
back

front

```
add(1); ...  add(4); add(5); remove(); add(6);
```

Circular buffers: a specialized queue

- Network switches need very fast queues for packets waiting to be sent.
- Compromise: queues are fixed size; packets discarded if no room.



6	2	3	4	5

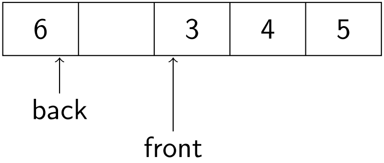
back

front

```
add(1); ...  add(4); add(5); remove(); add(6);
add(7); /* fails */
```

Circular buffers: a specialized queue

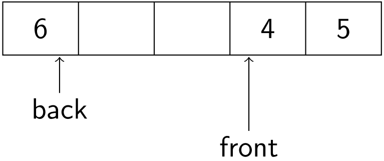
- Network switches need very fast queues for packets waiting to be sent.
- Compromise: queues are fixed size; packets discarded if no room.



```
add(1); ...  add(4); add(5); remove(); add(6);
add(7); /* fails */
remove();
```

Circular buffers: a specialized queue

- Network switches need very fast queues for packets waiting to be sent.
- Compromise: queues are fixed size; packets discarded if no room.



6			4	5

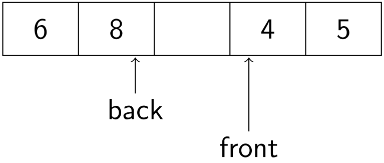
back

front

```
add(1); ...  add(4); add(5); remove(); add(6);
add(7); /* fails */
remove(); remove();
```

Circular buffers: a specialized queue

- Network switches need very fast queues for packets waiting to be sent.
- Compromise: queues are fixed size; packets discarded if no room.



```
add(1); ...  add(4); add(5); remove(); add(6);
add(7); /* fails */
remove(); remove(); add(8);
```

Java implementation (1)

```
public class CircularBuffer {
    private String[] items;
    private int front = 0;
    private int back = 0;
    private int length = 0;

    public CircularBuffer (int size) {
        items = new String[size];
    }

    public boolean isFull() {
        return length == items.length;
    }

    public boolean isEmpty() { return length == 0; }
    ...
}
```

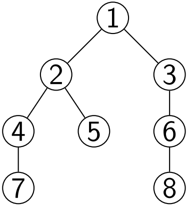
Java implementation (2)

```
public boolean add (String s) {
    if (isFull())
        return false;

    items[back] = s;
    back = (back + 1) % items.length;
    length++;
    return true;
}
```

Java implementation (3)

```
public String remove () {
    if (isEmpty())
        return null;

    String s = items[front];
    front = (front + 1) % items.length;
    length--;
    return s;
}
```

Breadth-first search



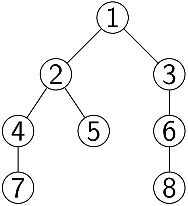
Visited:

Current:

Queue:

```
add (start node)
while (!isEmpty) {
    remove next node
    if node is target
        return found
    add all children
}
return not-found
```

Breadth-first search



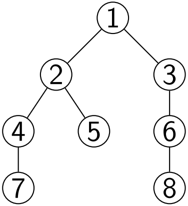
Visited:

Current:

Queue: 1

```
▷  add (start node)
   while (!isEmpty) {
        remove next node
        if node is target
            return found
        add all children
   }
   return not-found
```

Breadth-first search



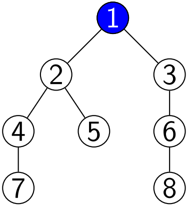
Visited:

Current:

Queue: 1

```
    add (start node)
▷   while (!isEmpty) {
        remove next node
        if node is target
            return found
        add all children
    }
    return not-found
```

Breadth-first search



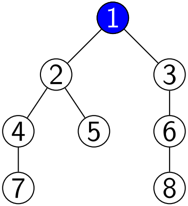
Visited: 1

Current: 1

Queue:

```
    add (start node)
    while (!isEmpty) {
▷       remove next node
        if node is target
            return found
        add all children
    }
    return not-found
```

Breadth-first search



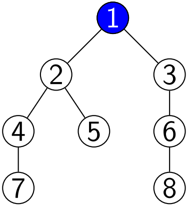
Visited: 1

Current: 1

Queue: 2, 3

```
  add (start node)
  while (!isEmpty) {
      remove next node
      if node is target
          return found
▷     add all children
  }
  return not-found
```

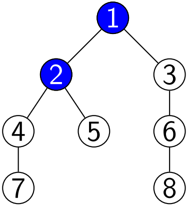
Breadth-first search



Visited:	1
Current:	1
Queue:	2, 3

```
  add (start node)
▷ while (!isEmpty) {
       remove next node
       if node is target
           return found
       add all children
  }
  return not-found
```

Breadth-first search



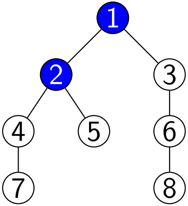
Visited: 1, 2

Current: 2

Queue: 3

```
   add (start node)
   while (!isEmpty) {
▷      remove next node
       if node is target
           return found
       add all children
   }
   return not-found
```

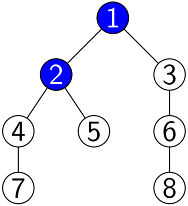
Breadth-first search



Visited:	1, 2
Current:	2
Queue:	3, 4, 5

```
   add (start node)
   while (!isEmpty) {
        remove next node
        if node is target
            return found
▷       add all children
   }
   return not-found
```

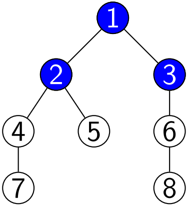
Breadth-first search



Visited:	1, 2
Current:	2
Queue:	3, 4, 5

```
    add (start node)
▷   while (!isEmpty) {
          remove next node
          if node is target
               return found
          add all children
    }
    return not-found
```

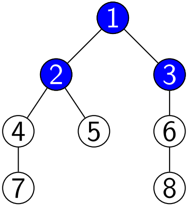
Breadth-first search



Visited:	1, 2, 3
Current:	3
Queue:	4, 5

```
  add (start node)
  while (!isEmpty) {
▷     remove next node
      if node is target
          return found
      add all children
  }
  return not-found
```

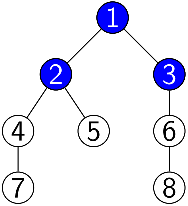
Breadth-first search



Visited:	1, 2, 3
Current:	3
Queue:	4, 5, 6

```
    add (start node)
    while (!isEmpty) {
         remove next node
         if node is target
             return found
▷        add all children
    }
    return not-found
```

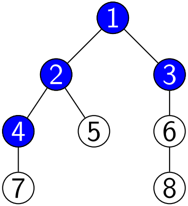
Breadth-first search



Visited:	1, 2, 3
Current:	3
Queue:	4, 5, 6

```
▷ add (start node)
  while (!isEmpty) {
       remove next node
       if node is target
           return found
       add all children
  }
  return not-found
```

Breadth-first search



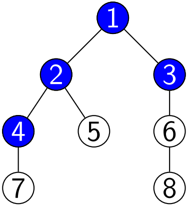
Visited: 1, 2, 3, 4

Current: 4

Queue: 5, 6

```
    add (start node)
    while (!isEmpty) {
▷       remove next node
        if node is target
            return found
        add all children
    }
    return not-found
```

Breadth-first search



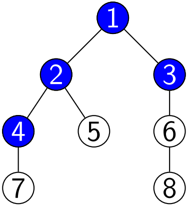
Visited: 1, 2, 3, 4

Current: 4

Queue: 5, 6, 7

```
  add (start node)
  while (!isEmpty) {
       remove next node
       if node is target
           return found
▷      add all children
  }
  return not-found
```

Breadth-first search



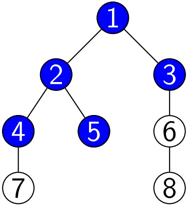
Visited: 1, 2, 3, 4

Current: 4

Queue: 5, 6, 7

```
   add (start node)
▷  while (!isEmpty) {
        remove next node
        if node is target
            return found
        add all children
   }
   return not-found
```

Breadth-first search



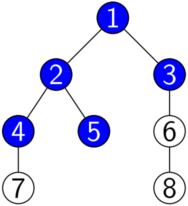
Visited: 1, 2, 3, 4, 5

Current: 5

Queue: 6, 7

```
   add (start node)
   while (!isEmpty) {
▷      remove next node
       if node is target
           return found
       add all children
   }
   return not-found
```

Breadth-first search



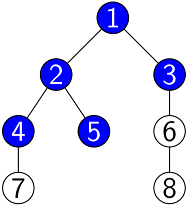
Visited: 1, 2, 3, 4, 5

Current: 5

Queue: 6, 7

```
  add (start node)
  while (!isEmpty) {
      remove next node
      if node is target
          return found
▷     add all children
  }
  return not-found
```

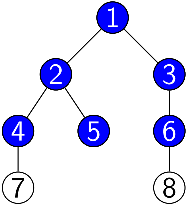
Breadth-first search



Visited:	1, 2, 3, 4, 5
Current:	5
Queue:	6, 7

```
   add (start node)
▷  while (!isEmpty) {
        remove next node
        if node is target
            return found
        add all children
   }
   return not-found
```

Breadth-first search



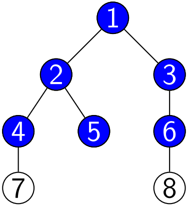
Visited: 1, 2, 3, 4, 5, 6

Current: 6

Queue: 7

```
    add (start node)
    while (!isEmpty) {
▷       remove next node
        if node is target
            return found
        add all children
    }
    return not-found
```

Breadth-first search



Visited:	1, 2, 3, 4, 5, 6
Current:	6
Queue:	7,8

```
   add (start node)
   while (!isEmpty) {
        remove next node
        if node is target
            return found
▷       add all children
   }
   return not-found
```

Queues: summary

- A queue is a first-in-first-out memory.
- Operations are `add` and `remove`.
- The queue can grow to any length but `add` and `remove` always take the same amount of time.
- Uses: network switches, event handling, breadth-first search.
- Circular buffers are a specialized fixed-size queue for network switches.