COMP110: Principles of Computing

7: Data Structures I

Assignments

- Peer review **next Thursday**
- Please come to the session with a **draft** of your research journal

- Worksheet 5 due **tomorrow**
- **No new worksheet** this week — work on your research journal instead

FALMOUTH
UNIVERSITY

Basic containers in Python

Memory allocation

Memory allocation

- Memory is allocated in **blocks**

Memory allocation

- Memory is allocated in **blocks**
- The program specifies the size, in bytes, of the block it wants

Memory allocation

- Memory is allocated in **blocks**
- The program specifies the size, in bytes, of the block it wants
- The OS allocates a **contiguous** block of that size

Memory allocation

- Memory is allocated in **blocks**
- The program specifies the size, in bytes, of the block it wants
- The OS allocates a **contiguous** block of that size
- The program owns that block until it frees it

Memory allocation

- Memory is allocated in **blocks**
- The program specifies the size, in bytes, of the block it wants
- The OS allocates a **contiguous** block of that size
- The program owns that block until it frees it
- Forgetting to free a block is called a **memory leak** (not really possible in Python, but a common bug in C++)

Memory allocation

- Memory is allocated in **blocks**
- The program specifies the size, in bytes, of the block it wants
- The OS allocates a **contiguous** block of that size
- The program owns that block until it frees it
- Forgetting to free a block is called a **memory leak** (not really possible in Python, but a common bug in C++)
- Blocks can be allocated and deallocated at will, but can **never grow or shrink**

Containers

Containers

- Memory management is hard and programmers are lazy

Containers

- Memory management is hard and programmers are lazy
- Containers are an **abstraction**

Containers

- Memory management is hard and programmers are lazy
- Containers are an **abstraction**
  - Hide the details of memory allocation, and allow the programmer to write simpler code

Containers

- Memory management is hard and programmers are lazy
- Containers are an **abstraction**
  - Hide the details of memory allocation, and allow the programmer to write simpler code
- Containers are an **encapsulation**

Containers

- Memory management is hard and programmers are lazy
- Containers are an **abstraction**
  - Hide the details of memory allocation, and allow the programmer to write simpler code
- Containers are an **encapsulation**
  - Bundle together the data's representation in memory along with the algorithms for accessing it

Arrays

Arrays

- An **array** is a contiguous block of memory in which objects are stored, equally spaced, one after the other

Arrays

- An **array** is a contiguous block of memory in which objects are stored, equally spaced, one after the other
- Each array element has an **index**, starting from zero

Arrays

- An **array** is a contiguous block of memory in which objects are stored, equally spaced, one after the other
- Each array element has an **index**, starting from zero
- Given the address of the 0th element, it is easy to find the ith element:

$$\text{address}_i = \text{address}_0 + (i \times \text{elementSize})$$

Arrays

- An **array** is a contiguous block of memory in which objects are stored, equally spaced, one after the other
- Each array element has an **index**, starting from zero
- Given the address of the 0th element, it is easy to find the *i*th element:

$$\text{address}_i = \text{address}_0 + (i \times \text{elementSize})$$

- E.g. if the array starts at address 1000 and each element is 4 bytes, the 3rd element is at address $1000 + 4 \times 3 = 1012$

Arrays

- An **array** is a contiguous block of memory in which objects are stored, equally spaced, one after the other
- Each array element has an **index**, starting from zero
- Given the address of the 0th element, it is easy to find the ith element:

$$\text{address}_i = \text{address}_0 + (i \times \text{elementSize})$$

- E.g. if the array starts at address 1000 and each element is 4 bytes, the 3rd element is at address $1000 + 4 \times 3 = 1012$
- Accessing an array element is **constant time** $O(1)$

Lists

Lists

- An array is a block of memory, so its size is **fixed** once created

Lists

- An array is a block of memory, so its size is **fixed** once created
- A **list** is a variable size array

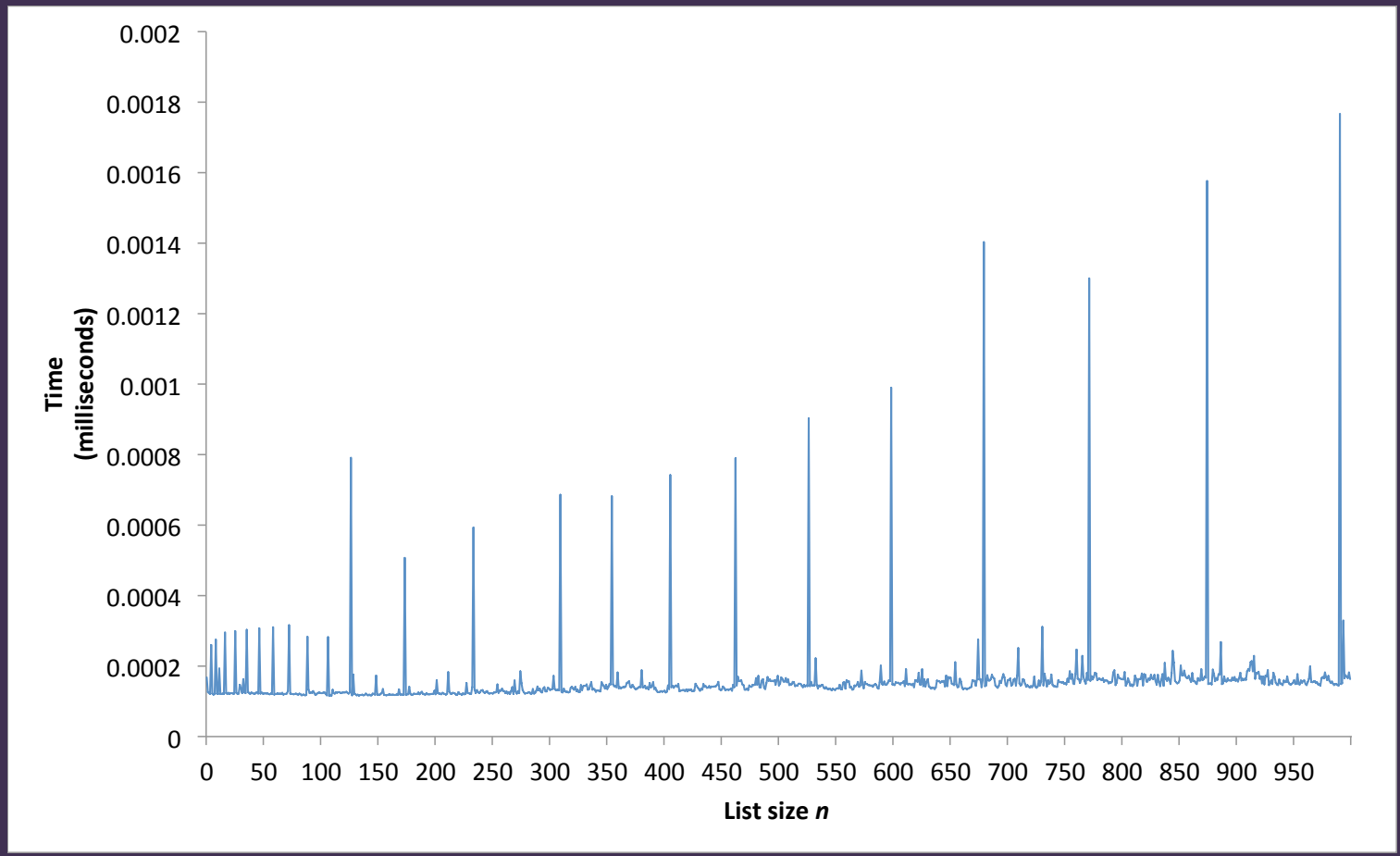
Lists

- An array is a block of memory, so its size is **fixed** once created
- A **list** is a variable size array
- When the list needs to change size, it **creates** a new array, **copies** the contents of the old array, and **deletes** the old array

Lists

- An array is a block of memory, so its size is **fixed** once created
- A **list** is a variable size array
- When the list needs to change size, it **creates** a new array, **copies** the contents of the old array, and **deletes** the old array
- Implementation details: `http://www.laurentluce.com/posts/python-list-implementation/`

Time taken to append an element to a list of size *n*



Operations on lists

Operations on lists

- **Appending** to a list is **amortised constant time**

Operations on lists

- **Appending** to a list is **amortised constant time**
  - Usually $O(1)$, but can go up to $O(n)$ if the list needs to change size

Operations on lists

- **Appending** to a list is **amortised constant time**
  - Usually $O(1)$, but can go up to $O(n)$ if the list needs to change size
- **Inserting** anywhere other than the end is **linear time**

Operations on lists

- **Appending** to a list is **amortised constant time**
  - Usually $O(1)$, but can go up to $O(n)$ if the list needs to change size
- **Inserting** anywhere other than the end is **linear time**
  - Can't just insert new bytes into a memory block — need to move all subsequent list elements to make room

Operations on lists

- **Appending** to a list is **amortised constant time**
  - Usually $O(1)$, but can go up to $O(n)$ if the list needs to change size
- **Inserting** anywhere other than the end is **linear time**
  - Can't just insert new bytes into a memory block — need to move all subsequent list elements to make room
- Similarly, **deleting** anything other than the last element is **linear time**

Tuples

Tuples

- Tuples are like lists, but are **immutable**

Tuples

- Tuples are like lists, but are **immutable**
  - Read-only

Tuples

- Tuples are like lists, but are **immutable**
  - Read-only
  - Once created, can't be changed

Tuples

- Tuples are like lists, but are **immutable**
  - Read-only
  - Once created, can't be changed
- Useful for storing sequences of values where adding, inserting, deleting or changing individual values does not make sense

Tuples

- Tuples are like lists, but are **immutable**
  - Read-only
  - Once created, can't be changed
- Useful for storing sequences of values where adding, inserting, deleting or changing individual values does not make sense
  - E.g. *xy* coordinates, RGB colours, ...

Tuples

- Tuples are like lists, but are **immutable**
  - Read-only
  - Once created, can't be changed
- Useful for storing sequences of values where adding, inserting, deleting or changing individual values does not make sense
  - E.g. *xy* coordinates, RGB colours, ...
- Create tuples with `()`, just as you create lists with `[]`

Tuples

- Tuples are like lists, but are **immutable**
  - Read-only
  - Once created, can't be changed
- Useful for storing sequences of values where adding, inserting, deleting or changing individual values does not make sense
  - E.g. *xy* coordinates, RGB colours, ...
- Create tuples with `()`, just as you create lists with `[]`
  - Exception: a single element tuple is created as `(foo,)` because `(foo)` would be interpreted as a bracketed expression

Tuples

- Tuples are like lists, but are **immutable**
  - Read-only
  - Once created, can't be changed
- Useful for storing sequences of values where adding, inserting, deleting or changing individual values does not make sense
  - E.g. *xy* coordinates, RGB colours, ...
- Create tuples with `()`, just as you create lists with `[]`
  - Exception: a single element tuple is created as `(foo,)` because `(foo)` would be interpreted as a bracketed expression
- Can often omit the parentheses entirely, e.g.

```
my_tuple = 1,2,3
```

Unpacking

If `foo` is a list or tuple of length 4, the following are equivalent:

Unpacking

If `foo` is a list or tuple of length 4, the following are equivalent:

```
a, b, c, d = foo
```

Unpacking

If `foo` is a list or tuple of length 4, the following are equivalent:

```
a, b, c, d = foo
```

```
a = foo[0]
b = foo[1]
c = foo[2]
d = foo[3]
```

Unpacking

If `foo` is a list or tuple of length 4, the following are equivalent:

```
a, b, c, d = foo
```

```
a = foo[0]
b = foo[1]
c = foo[2]
d = foo[3]
```

- Unpacking requires the number of elements to match exactly — if `foo` has more than 4 elements, the code on the left will give an error

One weird trick (Java programmers hate it!)

The following are equivalent:

One weird trick (Java programmers hate it!)

The following are equivalent:

```
a, b = b, a
```

One weird trick (Java programmers hate it!)

The following are equivalent:

```
a, b = b, a
```

```
temp = a
a = b
b = temp
```

Strings are immutable

Strings are immutable

- **Strings** are immutable in Python

Strings are immutable

- **Strings** are immutable in Python
  - This is not true of all programming languages

Strings are immutable

- **Strings** are immutable in Python
  - This is not true of all programming languages
- But wait... we change strings all the time, don't we?

```
my_string = "Hello "
my_string += "world"
```

Strings are immutable

- **Strings** are immutable in Python
  - This is not true of all programming languages
- But wait... we change strings all the time, don't we?

```
my_string = "Hello "
my_string += "world"
```

- This isn't changing the string, it's creating a new one and throwing the old one away!

Strings are immutable

- **Strings** are immutable in Python
  - This is not true of all programming languages
- But wait... we change strings all the time, don't we?

```
my_string = "Hello "
my_string += "world"
```

- This isn't changing the string, it's creating a new one and throwing the old one away!
- Hence building a long string by appending can be slow (appending strings is $O(n)$)

Dictionaries

Dictionaries

- Dictionaries are **associative maps**

Dictionaries

- Dictionaries are **associative maps**
- A dictionary maps **keys** to **values**

Dictionaries

- Dictionaries are **associative maps**
- A dictionary maps **keys** to **values**
  - Keys must be immutable (numbers, strings, tuples etc)

Dictionaries

- Dictionaries are **associative maps**
- A dictionary maps **keys** to **values**
  - Keys must be immutable (numbers, strings, tuples etc)
  - Values can be anything (including dictionaries or other containers)

Dictionaries

- Dictionaries are **associative maps**
- A dictionary maps **keys** to **values**
  - Keys must be immutable (numbers, strings, tuples etc)
  - Values can be anything (including dictionaries or other containers)
- A dictionary is implemented as a **hash table**

Using dictionaries

Using dictionaries

Create them using `{}`:

```
age = {"Alice": 23, "Bob": 36, "Charlie": 27}
```

Using dictionaries

Create them using `{}`:

```
age = {"Alice": 23, "Bob": 36, "Charlie": 27}
```

Access values using `[]`:

```
print(age["Alice"]) # prints 23
age["Bob"] = 40     # overwriting an existing item
age["Denise"] = 21  # adding a new item
```

Iterating over dictionaries

Iterating over dictionaries

Iterating over a dictionary gives the **keys**:

```
for x in age:
    print(x)    # prints Alice, Bob, Charlie
```

Iterating over dictionaries

Iterating over a dictionary gives the **keys**:

```
for x in age:
    print(x)    # prints Alice, Bob, Charlie
```

Use `items` to get **key,value** pairs:

```
for key, value in age.items():
    print(key, "is", age, "years old")
```

Sets

Sets

- Sets are like dictionaries without the values

Sets

- Sets are like dictionaries without the values
- Sets are **unordered** collections of **unique** elements

Sets

- Sets are like dictionaries without the values
- Sets are **unordered** collections of **unique** elements
  - Sets **cannot** contain **duplicate** elements

Sets

- Sets are like dictionaries without the values
- Sets are **unordered** collections of **unique** elements
  - Sets **cannot** contain **duplicate** elements
  - Attempting to `add` an element already present in the set does nothing

Sets

- Sets are like dictionaries without the values
- Sets are **unordered** collections of **unique** elements
  - Sets **cannot** contain **duplicate** elements
  - Attempting to `add` an element already present in the set does nothing
- Certain operations on sets scale better on average than the equivalent operations on lists:

Sets

- Sets are like dictionaries without the values
- Sets are **unordered** collections of **unique** elements
  - Sets **cannot** contain **duplicate** elements
  - Attempting to `add` an element already present in the set does nothing
- Certain operations on sets scale better on average than the equivalent operations on lists:

Operation	**List**	**Set**
Add element	Append: $O(1)$ Insert: $O(n)$	$O(1)$
Delete element	$O(n)$	$O(1)$
Contains element?	$O(n)$	$O(1)$

Using sets

Using sets

Create them using `{}`:

```
numbers = {1, 4, 9, 16, 25}
```

Using sets

Create them using `{}`:

```
numbers = {1, 4, 9, 16, 25}
```

Add and remove members with `add` and `remove` methods

```
numbers.add(36)
numbers.remove(4)
```

Using sets

Create them using `{}`:

```
numbers = {1, 4, 9, 16, 25}
```

Add and remove members with `add` and `remove` methods

```
numbers.add(36)
numbers.remove(4)
```

Test membership with `in` operator

```
if 9 in numbers:
    print("Set contains 9")
```

FALMOUTH
UNIVERSITY

Stacks and queues

Stacks and queues

Stacks and queues

- A **stack** is a **last-in first-out (LIFO)** data structure

Stacks and queues

- A **stack** is a **last-in first-out (LIFO)** data structure
- Items can be **pushed** to the **top** of the stack

Stacks and queues

- A **stack** is a **last-in first-out (LIFO)** data structure
- Items can be **pushed** to the **top** of the stack
- Items can be **popped** from the **top** of the stack

Stacks and queues

- A **stack** is a **last-in first-out (LIFO)** data structure
- Items can be **pushed** to the **top** of the stack
- Items can be **popped** from the **top** of the stack
- A **queue** is a **first-in first-out (FIFO)** data structure

Stacks and queues

- A **stack** is a **last-in first-out (LIFO)** data structure
- Items can be **pushed** to the **top** of the stack
- Items can be **popped** from the **top** of the stack

- A **queue** is a **first-in first-out (FIFO)** data structure
- Items can be **enqueued** to the **back** of the queue

Stacks and queues

- A **stack** is a **last-in first-out (LIFO)** data structure
- Items can be **pushed** to the **top** of the stack
- Items can be **popped** from the **top** of the stack

- A **queue** is a **first-in first-out (FIFO)** data structure
- Items can be **enqueued** to the **back** of the queue
- Items can be **dequeued** from the **front** of the queue

Lists in Python

Lists in Python

- Implemented as a (variable-sized) **array**

Lists in Python

- Implemented as a (variable-sized) **array**
- Appending is $O(1)$

Lists in Python

- Implemented as a (variable-sized) **array**
- Appending is $O(1)$
- Inserting is $O(n)$

Lists in Python

- Implemented as a (variable-sized) **array**
- Appending is $O(1)$
- Inserting is $O(n)$
- Deleting is $O(n)$

Stacks in Python

Stacks in Python

- Stacks can be implemented efficiently as lists

Stacks in Python

- Stacks can be implemented efficiently as lists
- `append` method adds an element to the end of the list

Stacks in Python

- Stacks can be implemented efficiently as lists
- `append` method adds an element to the end of the list
  - What is the time complexity?

Stacks in Python

- Stacks can be implemented efficiently as lists
- `append` method adds an element to the end of the list
  - What is the time complexity?
- `pop` method removes and returns the last element of the list

Stacks in Python

- Stacks can be implemented efficiently as lists
- `append` method adds an element to the end of the list
  - What is the time complexity?
- `pop` method removes and returns the last element of the list
  - What is the time complexity?

Queues in Python

Queues in Python

- Queues can be implemented as lists, but not efficiently

Queues in Python

- Queues can be implemented as lists, but not efficiently
- Could use `append(item)` to enqueue and `pop(0)` to dequeue

Queues in Python

- Queues can be implemented as lists, but not efficiently
- Could use `append(item)` to enqueue and `pop(0)` to dequeue
  - What is the time complexity of `pop(0)`?

Queues in Python

- Queues can be implemented as lists, but not efficiently
- Could use `append(item)` to enqueue and `pop(0)` to dequeue
  - What is the time complexity of `pop(0)`?
- Could use `insert(0, item)` to enqueue and `pop()` to dequeue

Queues in Python

- Queues can be implemented as lists, but not efficiently
- Could use `append(item)` to enqueue and `pop(0)` to dequeue
  - What is the time complexity of `pop(0)`?
- Could use `insert(0, item)` to enqueue and `pop()` to dequeue
  - What is the time complexity of `insert(0, item)`?

Queues in Python

- Queues can be implemented as lists, but not efficiently
- Could use `append(item)` to enqueue and `pop(0)` to dequeue
  - What is the time complexity of `pop(0)`?
- Could use `insert(0, item)` to enqueue and `pop()` to dequeue
  - What is the time complexity of `insert(0, item)`?
- `deque` (from the `collections` module) implements an efficient **double-ended queue**

Queues in Python

- Queues can be implemented as lists, but not efficiently
- Could use `append(item)` to enqueue and `pop(0)` to dequeue
  - What is the time complexity of `pop(0)`?
- Could use `insert(0, item)` to enqueue and `pop()` to dequeue
  - What is the time complexity of `insert(0, item)`?
- `deque` (from the `collections` module) implements an efficient **double-ended queue**
- Provides methods `append`, `appendleft`, `pop`, `popleft`

Queues in Python

- Queues can be implemented as lists, but not efficiently
- Could use `append(item)` to enqueue and `pop(0)` to dequeue
  - What is the time complexity of `pop(0)`?
- Could use `insert(0, item)` to enqueue and `pop()` to dequeue
  - What is the time complexity of `insert(0, item)`?
- `deque` (from the `collections` module) implements an efficient **double-ended queue**
- Provides methods `append`, `appendleft`, `pop`, `popleft`
  - All of which are $O(1)$

Stacks and function calls

Stacks and function calls

- Stacks are used to implement **nested function calls**

Stacks and function calls

- Stacks are used to implement **nested function calls**
- Each invocation of a function has a **stack frame**

Stacks and function calls

- Stacks are used to implement **nested function calls**
- Each invocation of a function has a **stack frame**
- This specifies information like **local variable values** and **return address**

Stacks and function calls

- Stacks are used to implement **nested function calls**
- Each invocation of a function has a **stack frame**
- This specifies information like **local variable values** and **return address**
- Calling a function **pushes** a new frame onto the stack

Stacks and function calls

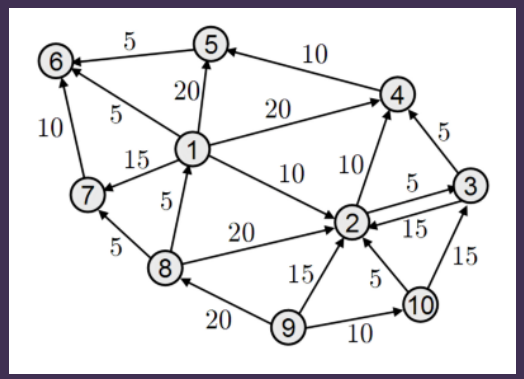
- Stacks are used to implement **nested function calls**
- Each invocation of a function has a **stack frame**
- This specifies information like **local variable values** and **return address**
- Calling a function **pushes** a new frame onto the stack
- Returning from a function **pops** the top frame off the stack

FALMOUTH
UNIVERSITY

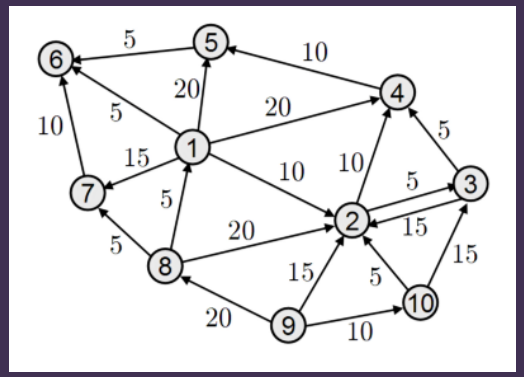
Graphs

Graphs

Graphs

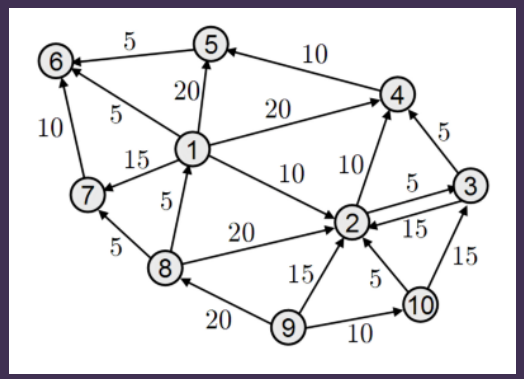


Graphs



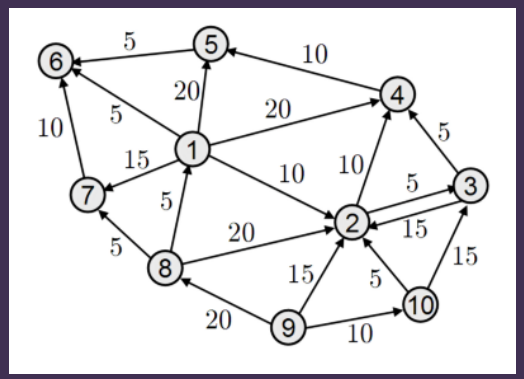
- A **graph** is defined by:

Graphs



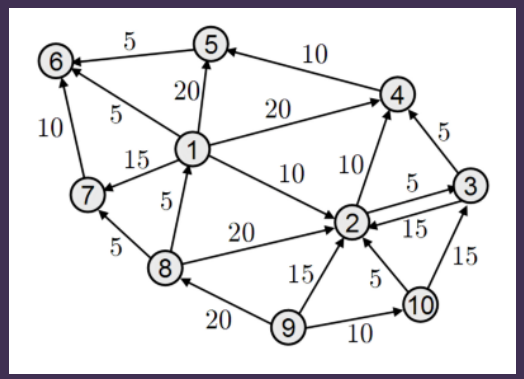
- A **graph** is defined by:
  - A collection of **nodes** or **vertices** (points)

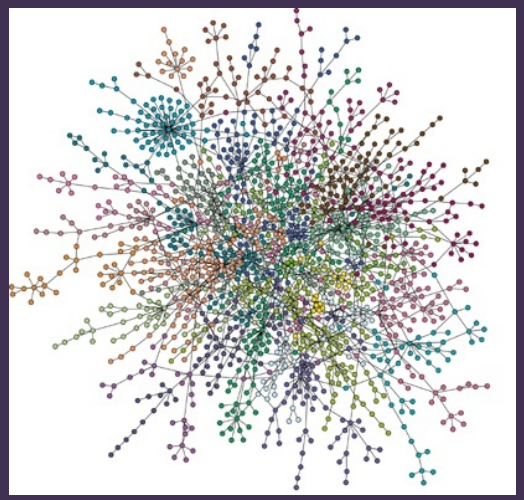
Graphs



- A **graph** is defined by:
  - A collection of **nodes** or **vertices** (points)
  - A collection of **edges** or **arcs** (lines or arrows between points)

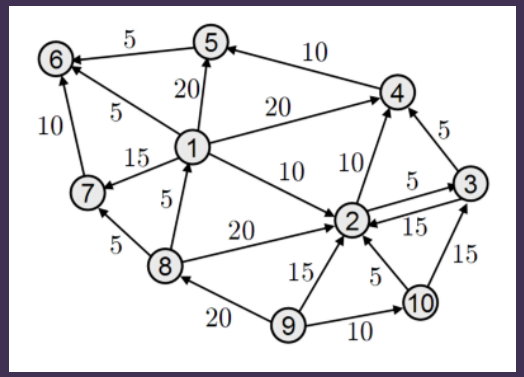
Graphs





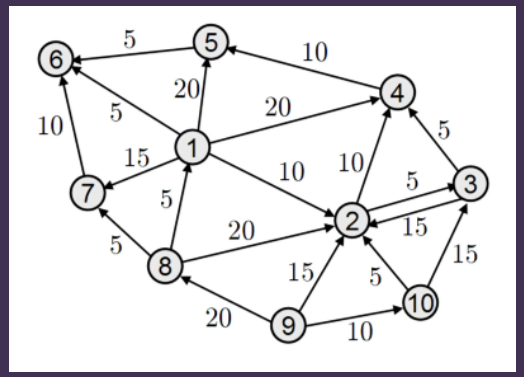
- A **graph** is defined by:
  - A collection of **nodes** or **vertices** (points)
  - A collection of **edges** or **arcs** (lines or arrows between points)
- Often used to model **networks** (e.g. social networks, transport networks, game levels, automata, ...)

Graphs



- A **graph** is defined by:
  - A collection of **nodes** or **vertices** (points)
  - A collection of **edges** or **arcs** (lines or arrows between points)
- Often used to model **networks** (e.g. social networks, transport networks, game levels, automata, ...)
- **Directed** graph: edges are arrows

Graphs



- A **graph** is defined by:
  - A collection of **nodes** or **vertices** (points)
  - A collection of **edges** or **arcs** (lines or arrows between points)
- Often used to model **networks** (e.g. social networks, transport networks, game levels, automata, ...)
- **Directed** graph: edges are arrows
- **Undirected** graph: edges are lines

Implementing graphs

Implementing graphs

- A graph has a **set of nodes** and a **set of edges**

Implementing graphs

- A graph has a **set of nodes** and a **set of edges**
- Each edge has exactly **two nodes** associated with it (e.g. “from” and “to”)

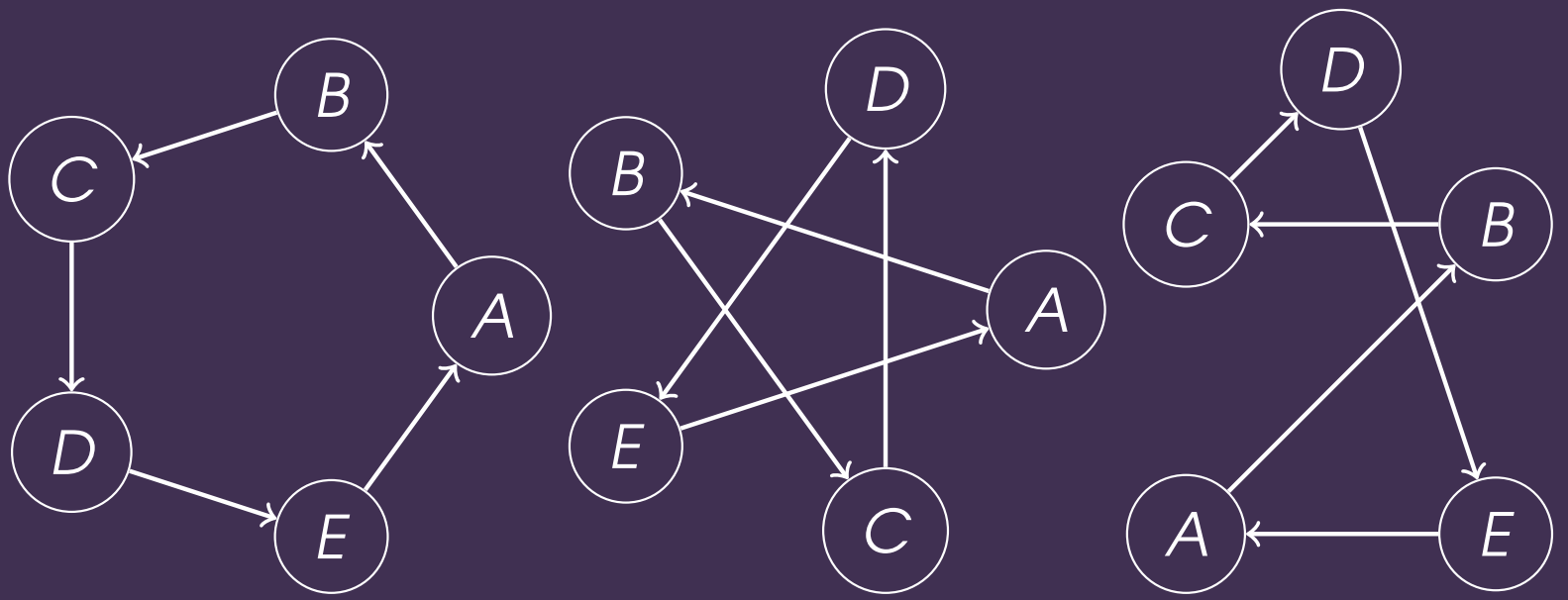
Drawing graphs

Drawing graphs

- A graph does not necessarily specify the physical **positions** of its nodes

Drawing graphs

- A graph does not necessarily specify the physical **positions** of its nodes
- E.g. these are technically the same graph:



Planar graphs

Planar graphs

- A graph is **planar** if it can be drawn with no overlapping edges

Planar graphs

- A graph is **planar** if it can be drawn with no overlapping edges
- A region enclosed by edges is called a **faces**

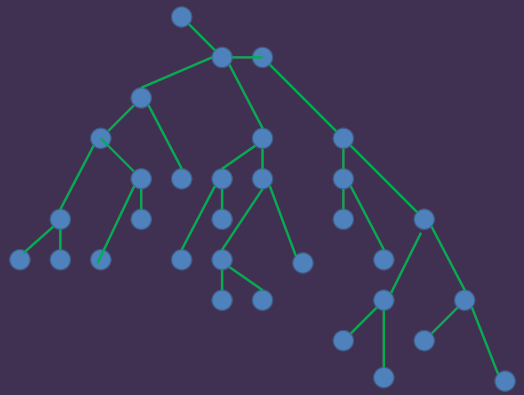
Planar graphs

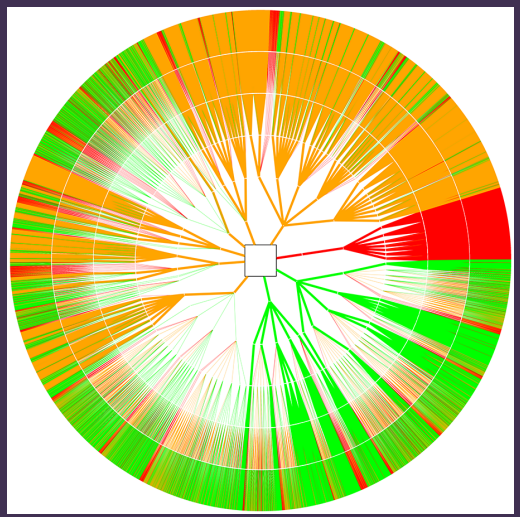
- A graph is **planar** if it can be drawn with no overlapping edges
- A region enclosed by edges is called a **faces**
- A connected planar graph obeys **Euler's formula**:

$$n_{\text{nodes}} - n_{\text{edges}} + n_{\text{faces}} = 2$$

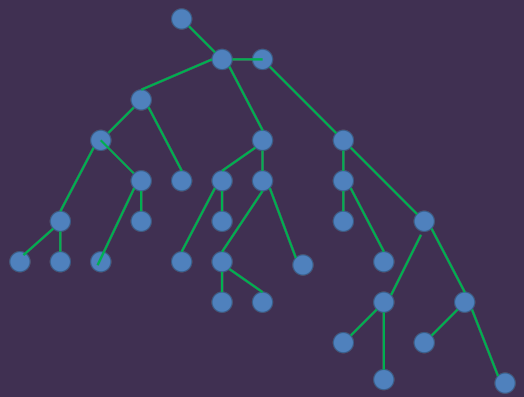
Trees

Trees

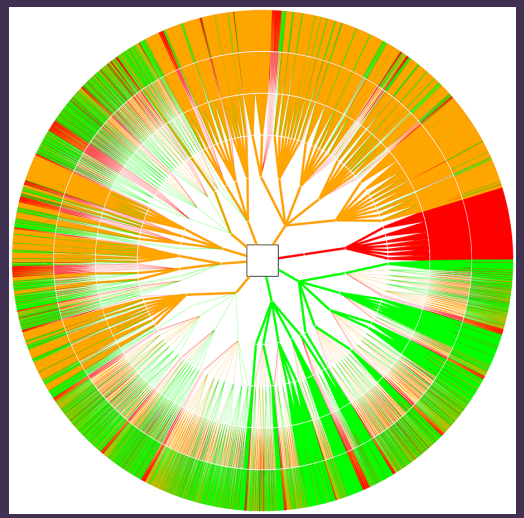




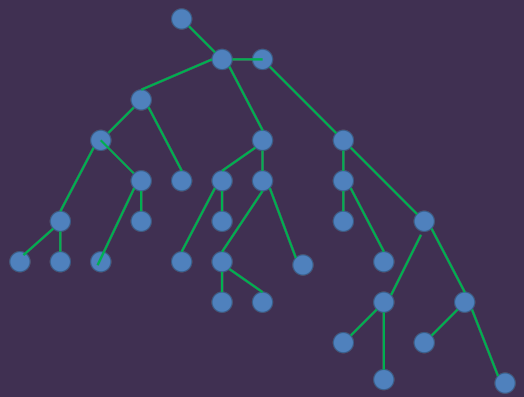
Trees



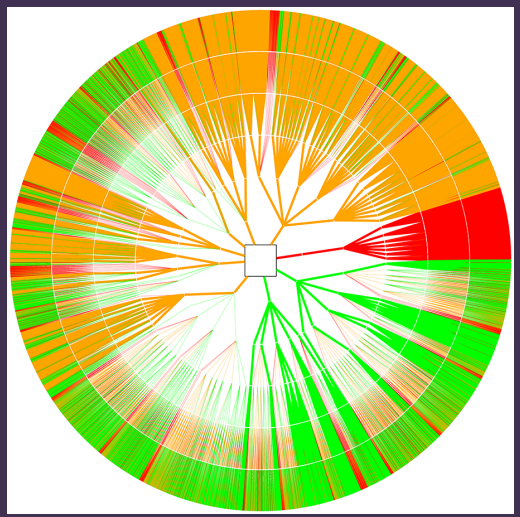
- A **tree** is a special type of directed graph where:



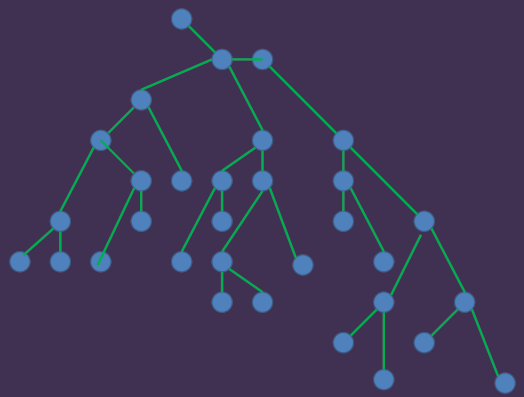
Trees



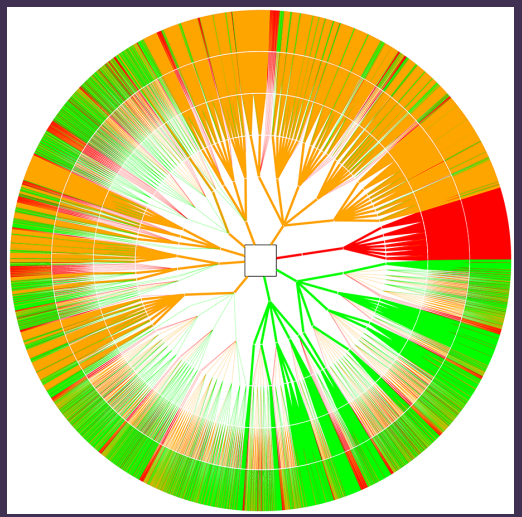
- A **tree** is a special type of directed graph where:
  - One node (the **root**) has no incoming edges



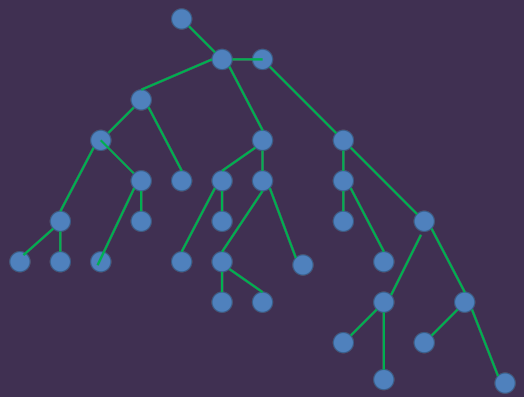
Trees

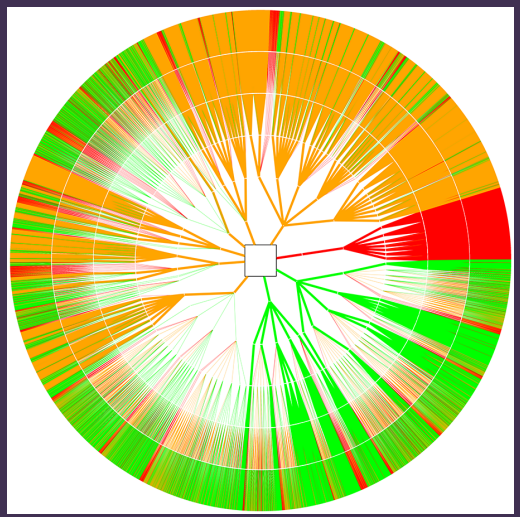


- A **tree** is a special type of directed graph where:
  - One node (the **root**) has no incoming edges
  - All other nodes have exactly 1 incoming edge



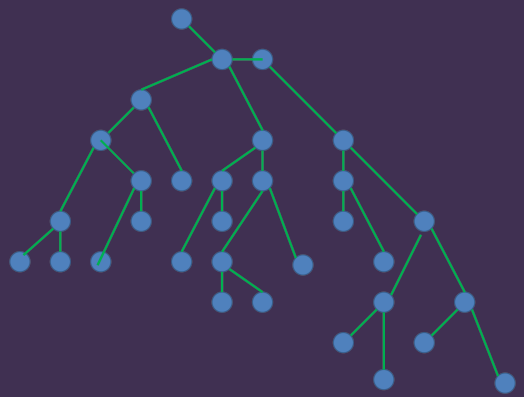
Trees

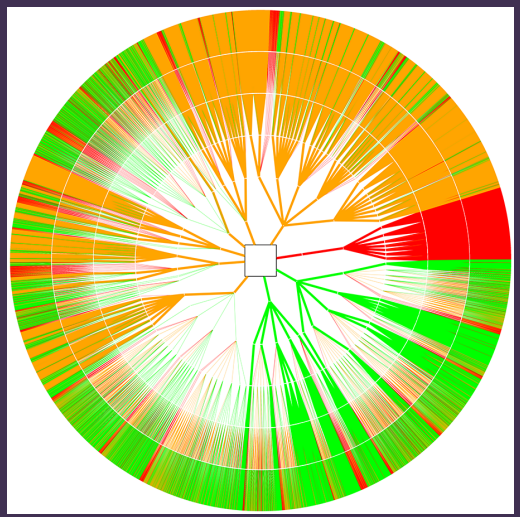




- A **tree** is a special type of directed graph where:
  - One node (the **root**) has no incoming edges
  - All other nodes have exactly 1 incoming edge
- Edges go from **parent** to **child**

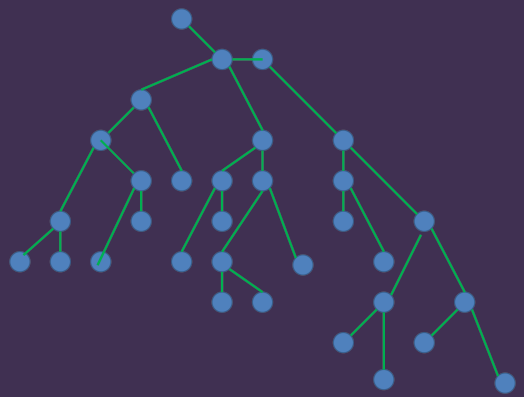
Trees

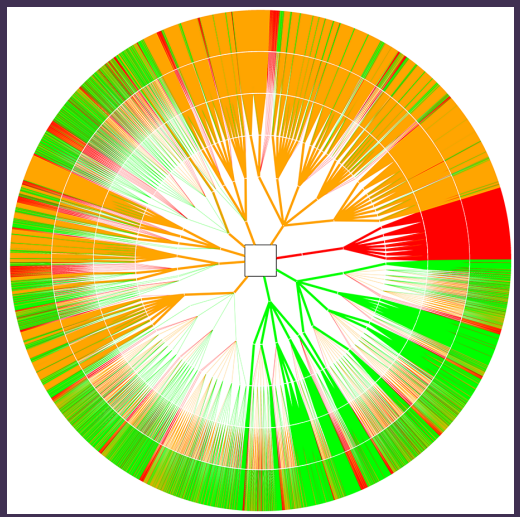




- A **tree** is a special type of directed graph where:
  - One node (the **root**) has no incoming edges
  - All other nodes have exactly 1 incoming edge
- Edges go from **parent** to **child**
  - All nodes except the root have exactly one parent

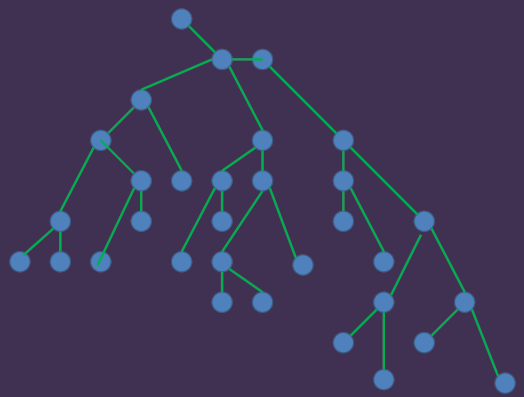
Trees

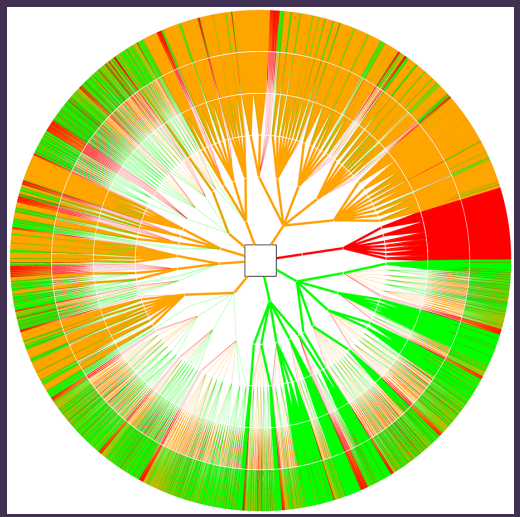




- A **tree** is a special type of directed graph where:
  - One node (the **root**) has no incoming edges
  - All other nodes have exactly 1 incoming edge
- Edges go from **parent** to **child**
  - All nodes except the root have exactly one parent
  - Nodes can have 0, 1 or many children

Trees





- A **tree** is a special type of directed graph where:
  - One node (the **root**) has no incoming edges
  - All other nodes have exactly 1 incoming edge
- Edges go from **parent** to **child**
  - All nodes except the root have exactly one parent
  - Nodes can have 0, 1 or many children
- Used to model **hierarchies** (e.g. file systems, object inheritance, scene graphs, state-action trees, ...)

Implementing trees

Implementing trees

- A graph has a **root node**

Implementing trees

- A graph has a **root node**
- Each node has a **collection of children**

Implementing trees

- A graph has a **root node**
- Each node has a **collection of children**
- Each node other than the root has a **single parent**

FALMOUTH
UNIVERSITY

Graph traversal

Tree traversal

Tree traversal

- **Traversal**: visiting all the nodes of the tree

Tree traversal

- **Traversal**: visiting all the nodes of the tree
- Two main types

Tree traversal

- **Traversal**: visiting all the nodes of the tree
- Two main types
  - Depth first

Tree traversal

- **Traversal**: visiting all the nodes of the tree
- Two main types
  - Depth first
  - Breadth first

Tree traversal

Tree traversal

procedure DEPTHFIRSTSEARCH

Tree traversal

procedure DEPTHFIRSTSEARCH
    let S be a stack

Tree traversal

procedure DEPTHFIRSTSEARCH
    let *S* be a stack
    push root node onto *S*

Tree traversal

procedure DEPTHFIRSTSEARCH
    let *S* be a stack
    push root node onto *S*
    while *S* is not empty **do**

Tree traversal

procedure DEPTHFIRSTSEARCH
    let *S* be a stack
    push root node onto *S*
    while *S* is not empty **do**
        pop *n* from *S*

Tree traversal

procedure DEPTHFIRSTSEARCH
    let *S* be a stack
    push root node onto *S*
    while *S* is not empty **do**
        pop *n* from *S*
        print *n*

Tree traversal

procedure DEPTHFIRSTSEARCH
    let *S* be a stack
    push root node onto *S*
    while *S* is not empty **do**
        pop *n* from *S*
        print *n*
        push children of *n* onto *S*

Tree traversal

```
procedure DepthFirstSearch
    let S be a stack
    push root node onto S
    while S is not empty do
        pop n from S
        print n
        push children of n onto S
    end while
end procedure
```

Tree traversal

```
procedure DepthFirstSearch
    let S be a stack
    push root node onto S
    while S is not empty do
        pop n from S
        print n
        push children of n onto S
    end while
end procedure

procedure BreadthFirstSearch
```

Tree traversal

```
procedure DepthFirstSearch
    let S be a stack
    push root node onto S
    while S is not empty do
        pop n from S
        print n
        push children of n onto S
    end while
end procedure

procedure BreadthFirstSearch
    let Q be a queue
```

Tree traversal

```
procedure DepthFirstSearch
    let S be a stack
    push root node onto S
    while S is not empty do
        pop n from S
        print n
        push children of n onto S
    end while
end procedure

procedure BreadthFirstSearch
    let Q be a queue
    enqueue root node into Q
```

Tree traversal

```
procedure DepthFirstSearch
    let S be a stack
    push root node onto S
    while S is not empty do
        pop n from S
        print n
        push children of n onto S
    end while
end procedure

procedure BreadthFirstSearch
    let Q be a queue
    enqueue root node into Q
    while Q is not empty do
```

Tree traversal

```
procedure DepthFirstSearch
    let S be a stack
    push root node onto S
    while S is not empty do
        pop n from S
        print n
        push children of n onto S
    end while
end procedure

procedure BreadthFirstSearch
    let Q be a queue
    enqueue root node into Q
    while Q is not empty do
        dequeue n from Q
```

Tree traversal

```
procedure DepthFirstSearch
    let S be a stack
    push root node onto S
    while S is not empty do
        pop n from S
        print n
        push children of n onto S
    end while
end procedure

procedure BreadthFirstSearch
    let Q be a queue
    enqueue root node into Q
    while Q is not empty do
        dequeue n from Q
        print n
```

Tree traversal

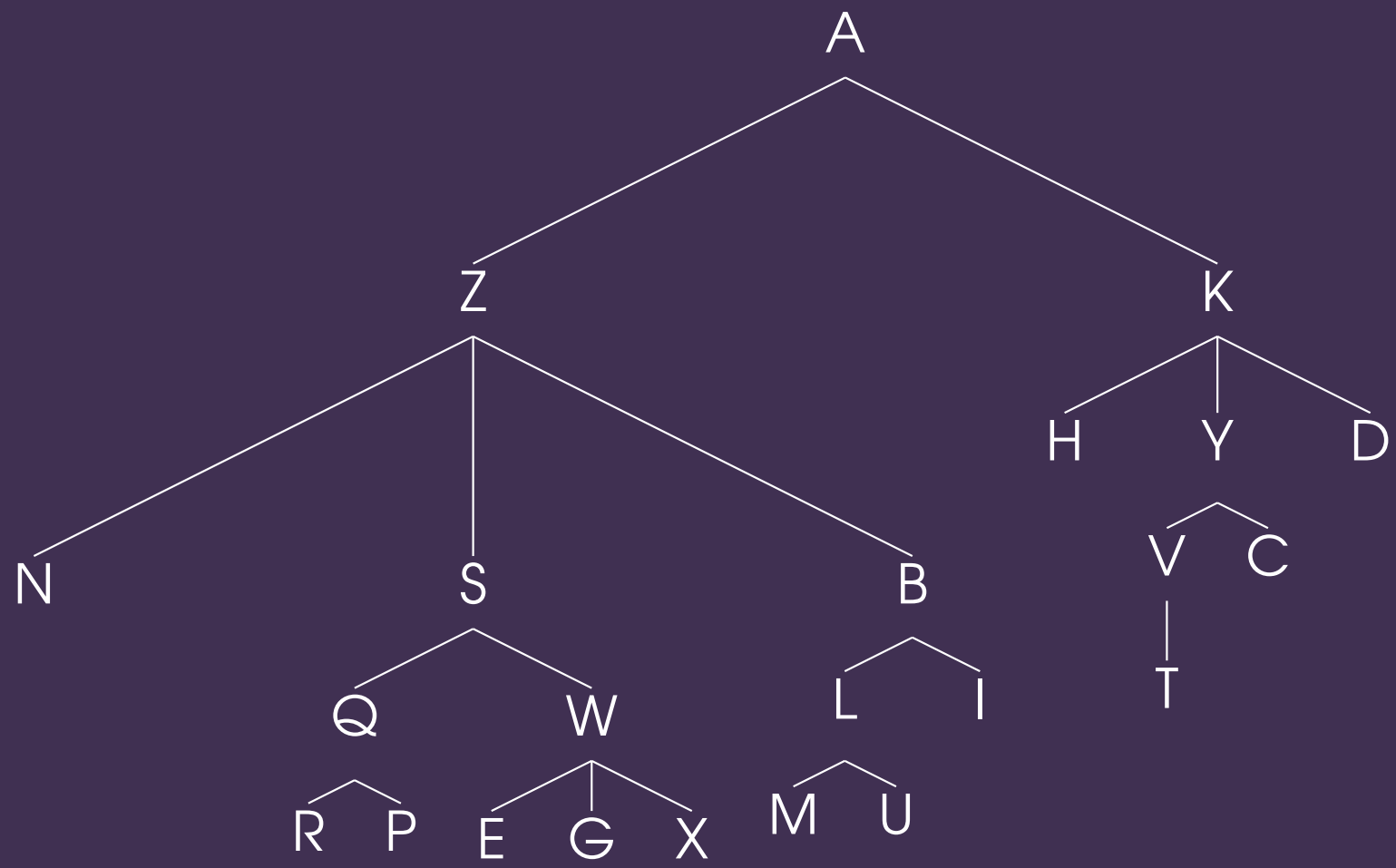
```
procedure DepthFirstSearch
    let S be a stack
    push root node onto S
    while S is not empty do
        pop n from S
        print n
        push children of n onto S
    end while
end procedure

procedure BreadthFirstSearch
    let Q be a queue
    enqueue root node into Q
    while Q is not empty do
        dequeue n from Q
        print n
        enqueue children of n into Q
```

Tree traversal

```
procedure DepthFirstSearch
    let S be a stack
    push root node onto S
    while S is not empty do
        pop n from S
        print n
        push children of n onto S
    end while
end procedure
```

```
procedure BreadthFirstSearch
    let Q be a queue
    enqueue root node into Q
    while Q is not empty do
        dequeue n from Q
        print n
        enqueue children of n into Q
    end while
end procedure
```

Tree traversal example



Recursive depth first search

Recursive depth first search

procedure DEPTHFIRSTSEARCH(n)

Recursive depth first search

procedure DEPTHFIRSTSEARCH(*n*)
    print *n*

Recursive depth first search

procedure DEPTHFIRSTSEARCH(*n*)
    print *n*
    for each child *c* of *n* **do**

Recursive depth first search

```
procedure DepthFirstSearch(n)
    print n
    for each child c of n do
        DepthFirstSearch(c)
```

Recursive depth first search

```
procedure DepthFirstSearch(n)
    print n
    for each child c of n do
        DepthFirstSearch(c)
    end for
end procedure
```

Recursive depth first search

```
procedure DepthFirstSearch(n)
    print n
    for each child c of n do
        DepthFirstSearch(c)
    end for
end procedure
```

- Compare to the pseudocode on the previous slide. Where is the stack?

FALMOUTH
UNIVERSITY

Linked lists

Linked list

Linked list

- Composed of a number of **nodes**

Linked list

- Composed of a number of **nodes**
- Each node contains:

Linked list

- Composed of a number of **nodes**
- Each node contains:
  - An **item** — the actual data to be stored

Linked list

- Composed of a number of **nodes**
- Each node contains:
  - An **item** — the actual data to be stored
  - A pointer or reference to the **next node** in the list (null for the last item)

Linked list

- Composed of a number of **nodes**
- Each node contains:
  - An **item** — the actual data to be stored
  - A pointer or reference to the **next node** in the list (null for the last item)

Linked lists vs arrays

Operation	Array	Linked list

[1] If we already have a reference to the last node
[2] If we already have a reference to the relevant node

Linked lists vs arrays

Operation	Array	Linked list
Append	$O(1)$	$O(1)$ [1]

[1] If we already have a reference to the last node
[2] If we already have a reference to the relevant node

Linked lists vs arrays

Operation	Array	Linked list
Append	$O(1)$	$O(1)$ [1]
Pop	$O(1)$	$O(1)$ [1]

[1] If we already have a reference to the last node
[2] If we already have a reference to the relevant node

Linked lists vs arrays

Operation	Array	Linked list
Append	$O(1)$	$O(1)$ [1]
Pop	$O(1)$	$O(1)$ [1]
Index lookup	$O(1)$	$O(n)$

[1] If we already have a reference to the last node
[2] If we already have a reference to the relevant node

Linked lists vs arrays

Operation	Array	Linked list
Append	$O(1)$	$O(1)$ [1]
Pop	$O(1)$	$O(1)$ [1]
Index lookup	$O(1)$	$O(n)$
Count elements	$O(1)$	$O(n)$

[1] If we already have a reference to the last node

[2] If we already have a reference to the relevant node

Linked lists vs arrays

Operation	Array	Linked list
Append	$O(1)$	$O(1)$ [1]
Pop	$O(1)$	$O(1)$ [1]
Index lookup	$O(1)$	$O(n)$
Count elements	$O(1)$	$O(n)$
Insert	$O(n)$	$O(1)$ [2]

[1] If we already have a reference to the last node

[2] If we already have a reference to the relevant node

Linked lists vs arrays

Operation	Array	Linked list
Append	$O(1)$	$O(1)$ [1]
Pop	$O(1)$	$O(1)$ [1]
Index lookup	$O(1)$	$O(n)$
Count elements	$O(1)$	$O(n)$
Insert	$O(n)$	$O(1)$ [2]
Delete	$O(n)$	$O(1)$ [2]

[1] If we already have a reference to the last node

[2] If we already have a reference to the relevant node

Implementing a linked list