COMP110: Principles of Computing

8: Data Structures

Learning outcomes

- **Define** the key concepts of graph theory
- **Distinguish** advanced data structures such as trees, DAGs and graphs
- **Determine** the complexity of accessing and manipulating data in these data structures
- **Choose** the correct data structure for a given task

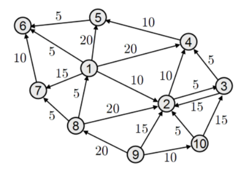
Exercise Sheet iii

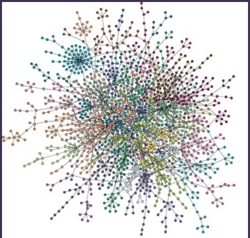
Due **tomorrow**

Graphs

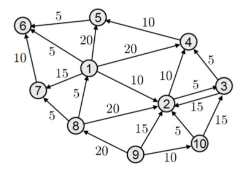
Graphs

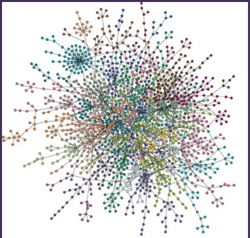
Graphs





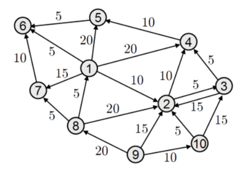
Graphs

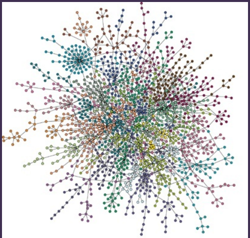




- A **graph** is defined by:

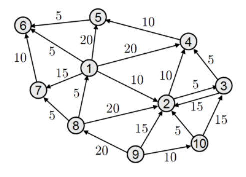
Graphs

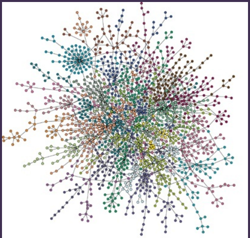




- A **graph** is defined by:
  - A collection of **nodes** or **vertices** (points)

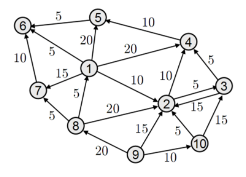
Graphs

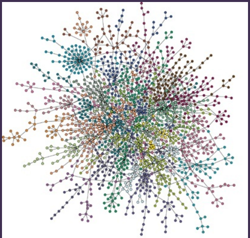




- A **graph** is defined by:
  - A collection of **nodes** or **vertices** (points)
  - A collection of **edges** or **arcs** (lines or arrows between points)

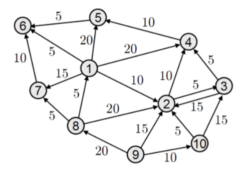
Graphs

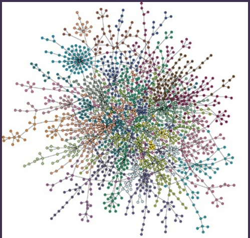




- A **graph** is defined by:
  - A collection of **nodes** or **vertices** (points)
  - A collection of **edges** or **arcs** (lines or arrows between points)
- Often used to model **networks** (e.g. social networks, transport networks, game levels, automata, ...)

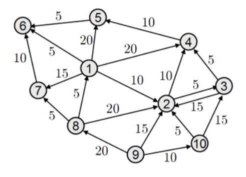
Graphs

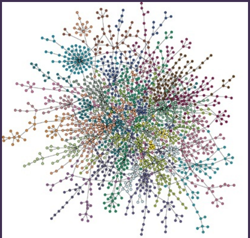




- A **graph** is defined by:
  - A collection of **nodes** or **vertices** (points)
  - A collection of **edges** or **arcs** (lines or arrows between points)
- Often used to model **networks** (e.g. social networks, transport networks, game levels, automata, ...)
- **Directed** graph: edges are arrows

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")

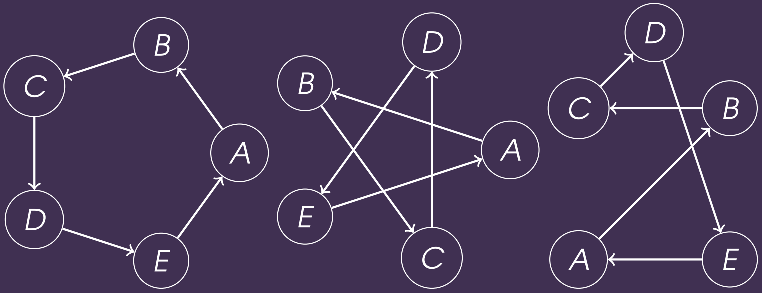
Drawing graphs

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**

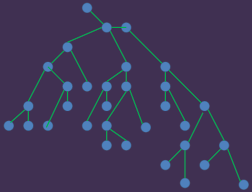
Planar graphs

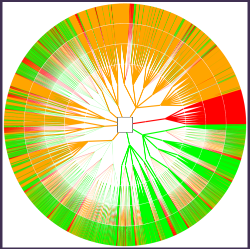
- A graph is **planar** if it can be drawn with no overlapping edges
- A region enclosed by edges is called a **faces**
- A connected planar graph obeys **Euler's formula**:

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

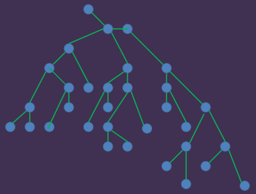
Trees

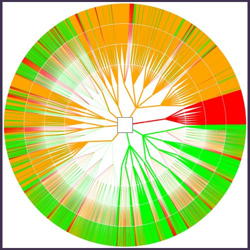
Trees





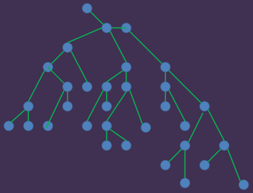
Trees

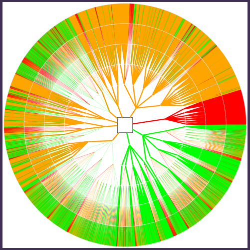




- A **tree** is a special type of directed graph where:

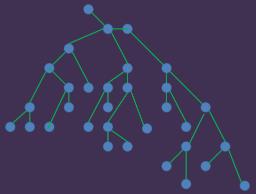
Trees

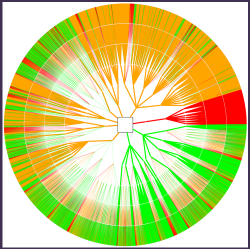




- A **tree** is a special type of directed graph where:
  - One node (the **root**) has no incoming edges

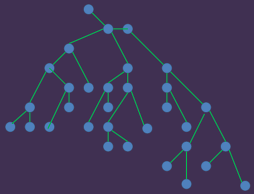
Trees

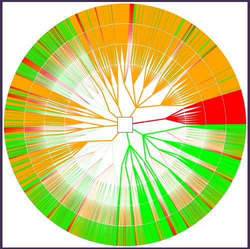




- A **tree** is a special type of directed graph where:
  - One node (the **root**) has no incoming edges
  - All other nodes have exactly 1 incoming edge

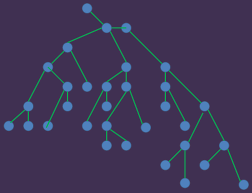
Trees

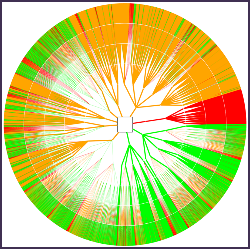




- A **tree** is a special type of directed graph where:
  - One node (the **root**) has no incoming edges
  - All other nodes have exactly 1 incoming edge
- Edges go from **parent** to **child**

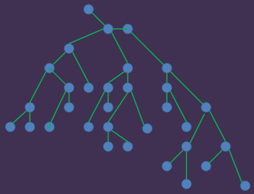
Trees

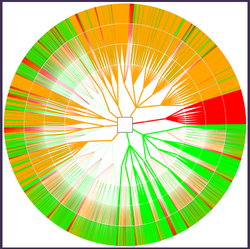




- A **tree** is a special type of directed graph where:
  - One node (the **root**) has no incoming edges
  - All other nodes have exactly 1 incoming edge
- Edges go from **parent** to **child**
  - All nodes except the root have exactly one parent

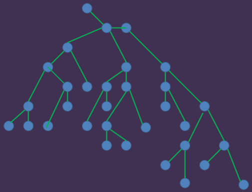
Trees

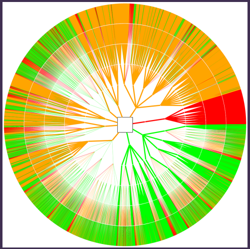




- A **tree** is a special type of directed graph where:
  - One node (the **root**) has no incoming edges
  - All other nodes have exactly 1 incoming edge
- Edges go from **parent** to **child**
  - All nodes except the root have exactly one parent
  - Nodes can have 0, 1 or many children

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**

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 (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
- Items can be **popped** from the **top** of the stack

- A **queue** is a **first-in first-out (LIFO)** 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 (LIFO)** 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

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

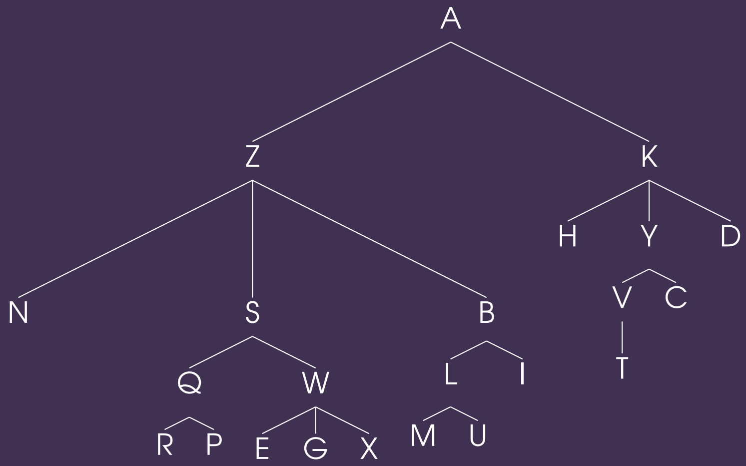
```
procedure DepthFirstSearch
    let S be a stack
    push root node onto S
    while S is not empty do
        pop n from S
        print n
        push children of n onto S
    end while
end procedure

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?

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