Advanced Data Structures

Dr Ian Cornelius

Hello

Hello (1)

Learning Outcomes

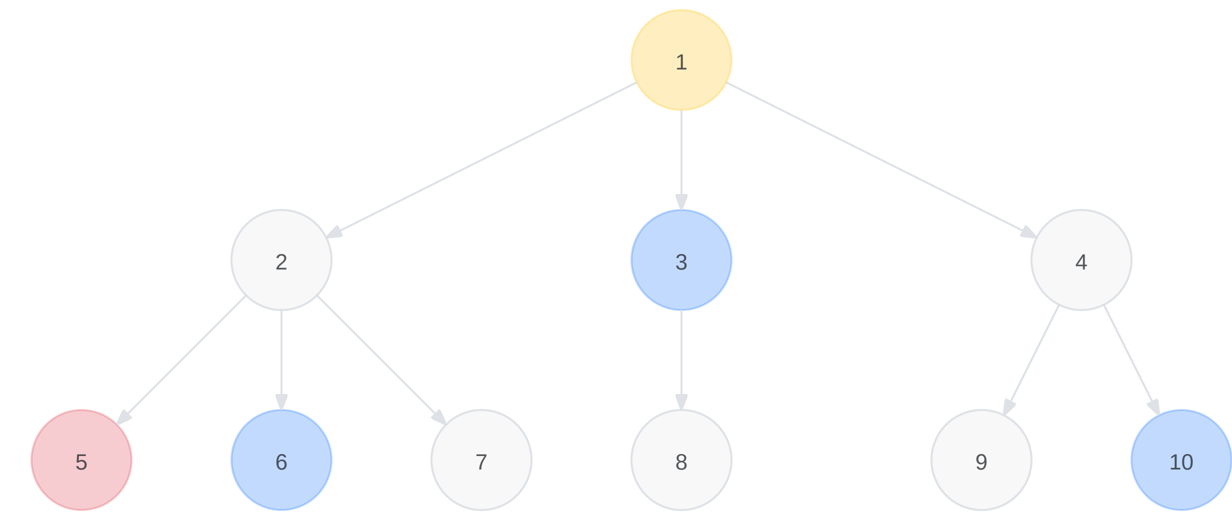
1. Understand the concept of trees and linked-lists and their function as a data structure
2. Practice and implement the use of trees and linked-lists in work undertaken for lab activities

Trees

Trees (1)

What are Trees?

- Trees are a connected, undirected graph with no cycles (more on this later)
- **Root** of a tree is the topmost node, or the *start* node for traversal
  - if a tree has a **root** node, it is called a *rooted tree*
- **Branches** of a tree is the path from the root to an end-point; the end-point is known as a **leaf**
- **Height** of a tree is equal to the number of edges
  - that is those that connect the root node to the leaf node that is the furthest away
- The number of *edges* (E) of a tree is equal to the number of *nodes* (N) minus one
  - $E = N - 1$



Tress (2)

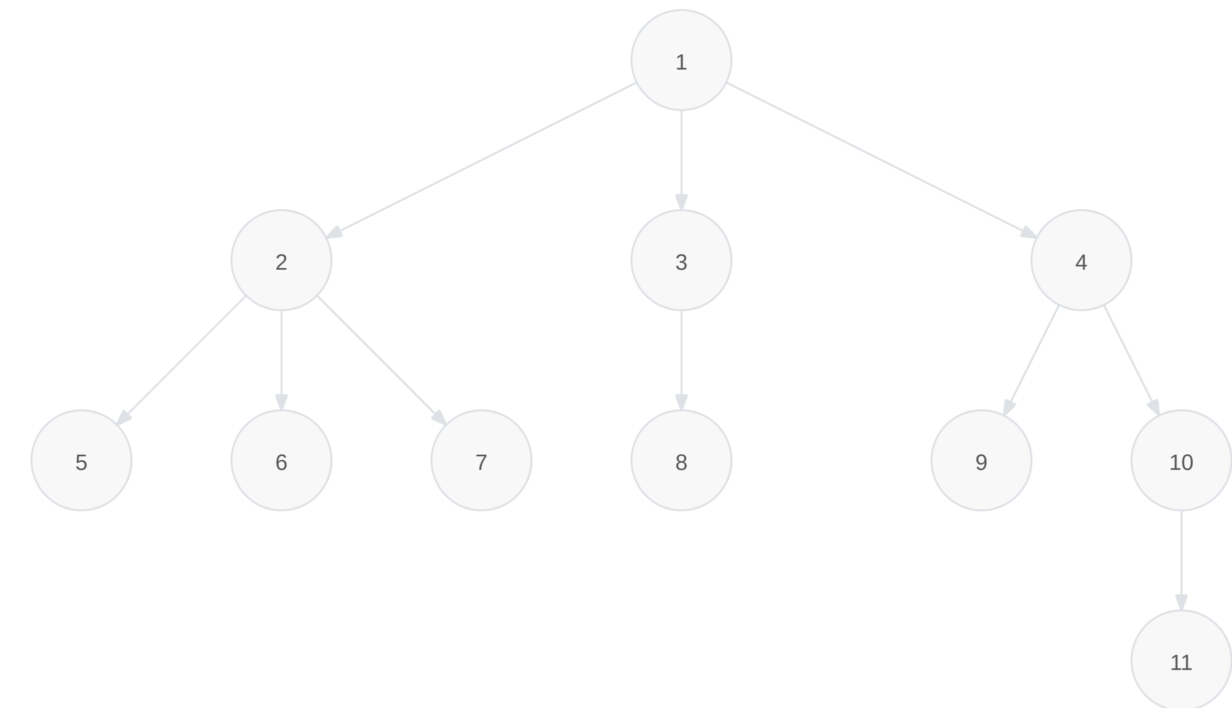
Types of Trees

- There are various different types of trees:
  1. Balanced and Unbalanced Trees
  2. Rooted Trees
  3. Binary Search Trees

Trees (3)

Balanced and Unbalanced Trees

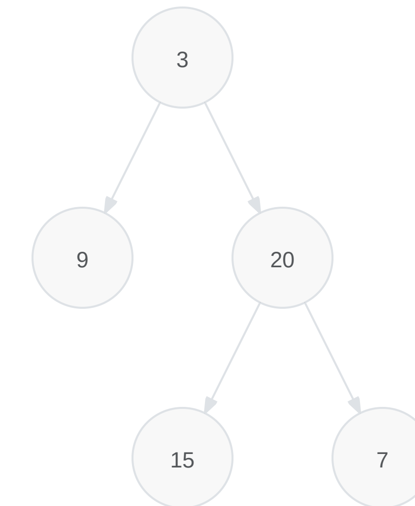
- Trees can be either balanced or unbalanced:
  - for a **balanced** tree, every leaf is around the same distance away from the root as any other leaf
  - for an **unbalanced** tree, one or more leaves are much further away from the root than any other leaf
- A *balanced* tree does not need to have the same number of nodes in the left and right subtrees
  - each branch (from root to leaf) must have the same *height*



Trees (4)

Rooted Trees

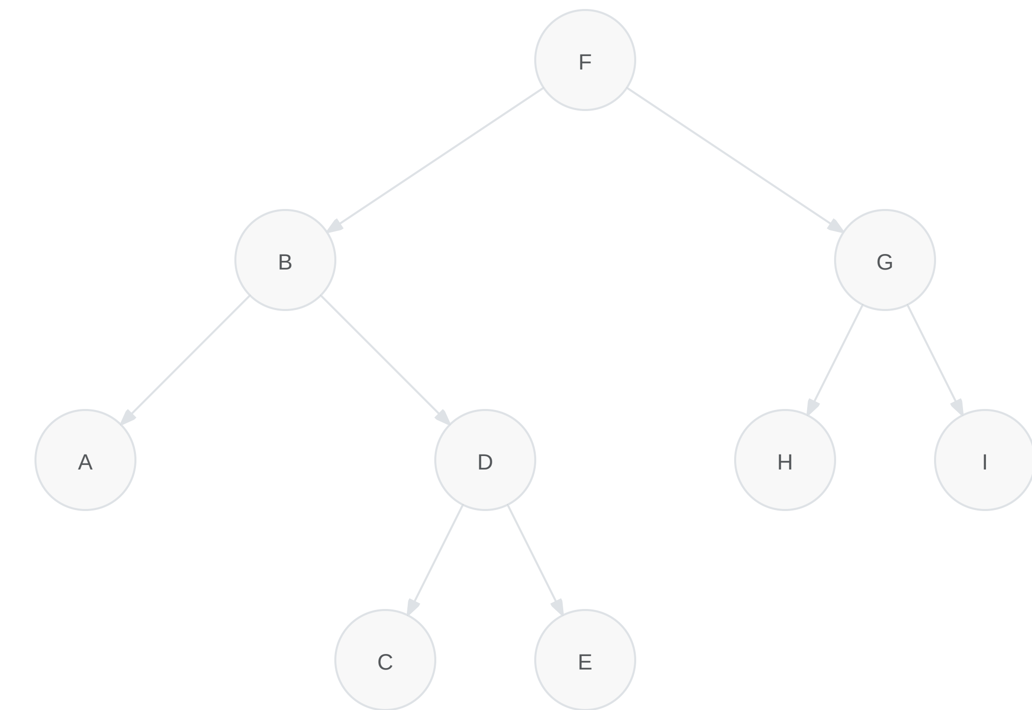
- Rooted trees are trees with one node that has been designated as the root
- The root node is commonly situated at the start (above all the other nodes)
  - branches descend to the leaf nodes
- Nodes are connected in a *parent-child* relationship
- A **parent** node is a node that comes directly before another adjacent node
  - the adjacent node is considered to be its **child**
- Nodes can have any number of children
- A **leaf** is a node that does not consist of any children



Trees (5)

Binary Search Trees (BST)

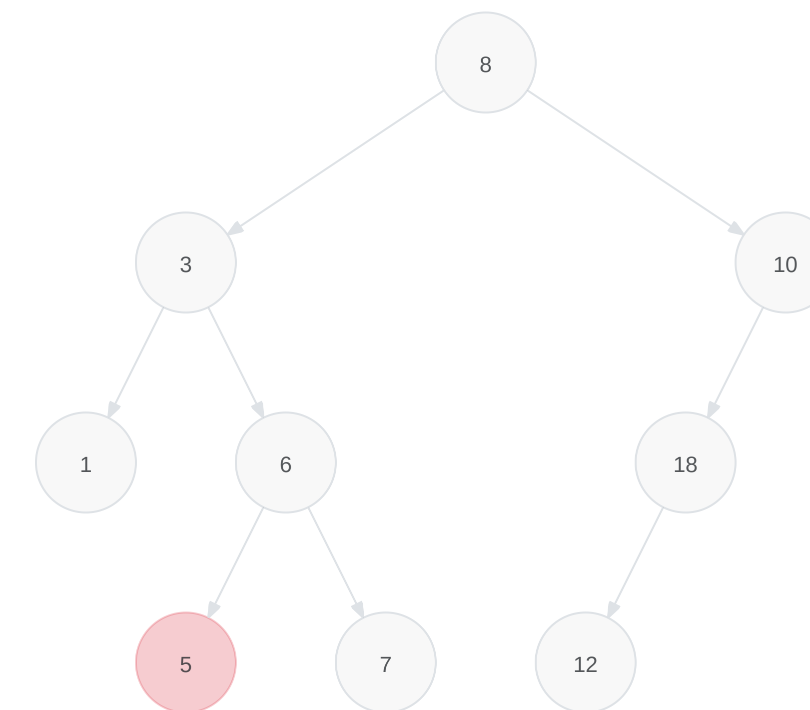
- A binary tree is a tree that is *rooted* and every node has at most two children
- A binary **search** tree is a special implementation of a *rooted binary tree*
  - ordered on a way to optimise searching
- Nodes of this tree type are ordered **ascending** (low to high):
  - the nodes on the left subtree have values that are *lower* than the root
  - the nodes on the right subtree have values that are *higher* than the root



Trees (6)

Methodology of a BST

- Click **Start** to proceed!



Start

Trees (7)

Efficiency of a BST

- Time complexity is dependent upon the balance of the tree
- A **balanced** tree in its:
  - *best case* will have a time complexity of $O(1)$
  - *worst case* will be $O(log n)$, where n, is the number of nodes in the tree
- An **unbalanced** tree in its:
  - *best case* will have a time complexity of $O(1)$
  - *worst case* will be $O(n)$, where n, is the number of nodes in the tree

Trees (8)

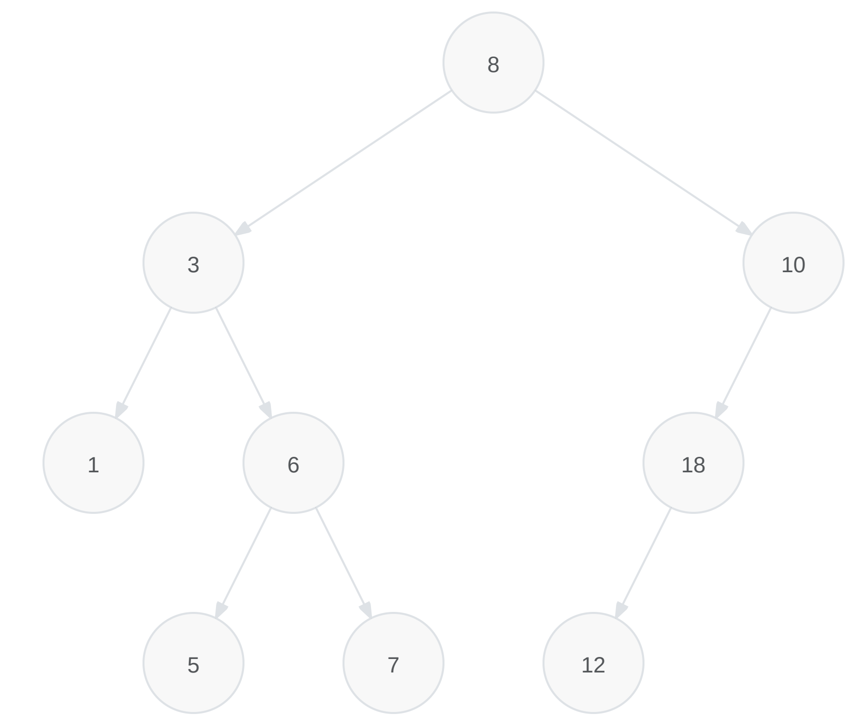
Traversing a BST

- Processing the data of a tree often requires **traversing**
  - traversing is concerned with systematically visiting every node
- A tree can be traversed left to right, or right to left; depending on the application
- Traversing in a particular order will specify a point in the traversal at which the node contents are *processed*
  - *processed* in this instance is where we print the value of the nodes
- When a node content has been processed, it is classified as **visited**
  - each node can be visited several times during traversal
  - it is only on one of the visits the nodes contents are processed
- There are three common algorithms for tree traversal:
  1. Pre-order
  2. Post-order
  3. In-order

Trees (9)

Traversing a BST: Pre-Order

- Click **Start** to proceed!

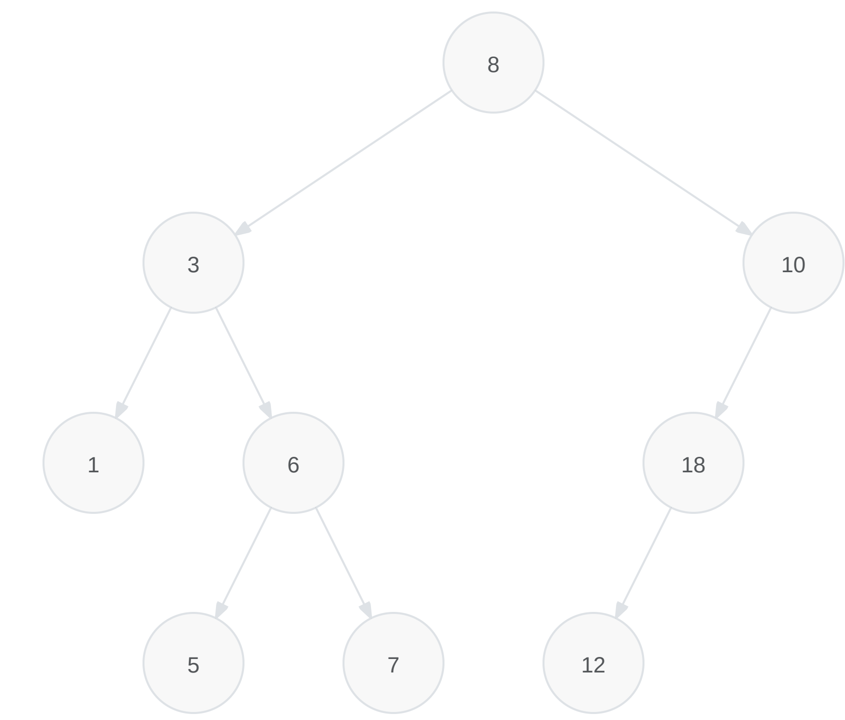


Start

Trees (10)

Traversing a BST: Post-Order

- Click **Start** to proceed!

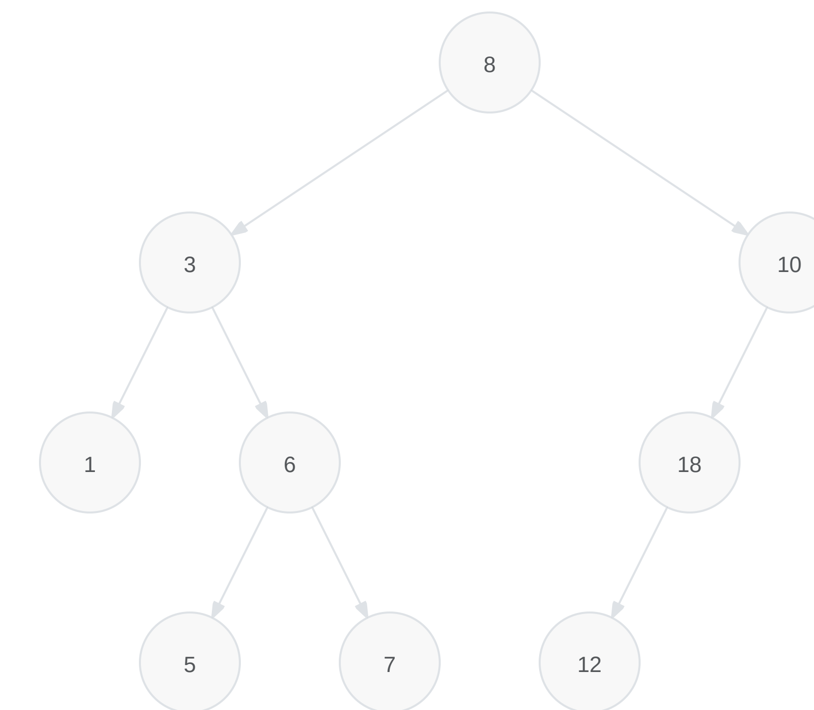


Start

Trees (11)

Traversing a BST: In-Order

- Click **Start** to proceed!



Start

Linked Lists

Linked Lists (1)

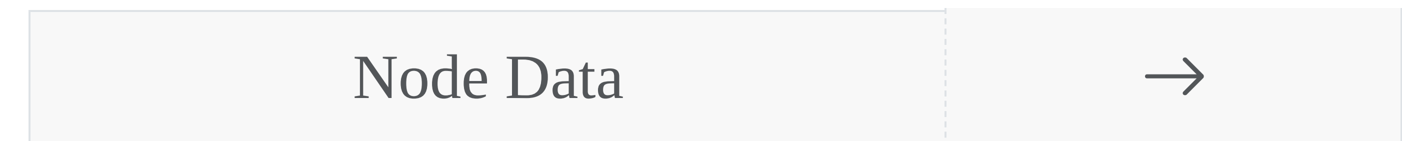
What are Linked Lists?

- Linked Lists are an ordered collection of objects
  - they are referred to as a **linear** data structure
- They differ from `lists` in the manner of how they store elements in the memory
- A normal list in Python will use a contiguous memory block
  - this will store the data element itself, and solely the data element
- Linked Lists store references to the next node
  - as well as the data element itself
  - **note**, reference refers to a memory address in this instance
- Variety type of linked lists, such as:
  - singly
  - doubly
  - circular

Linked Lists (2)

Structure of a Linked List

- Each element of a linked list is referred to as a **node**
- Every node has two different *fields*:
  1. **data**: contains the value that is stored in the node
  2. **next**: contains the reference to the next node on the list
- The *first* node is referred to as the **head**
  - is used as the starting point for any iteration through the list
- The *last* node of the list must have its *next* reference pointing to `None`
  - this will determine the end of the list



Linked Lists (3)

Queues and Stacks

- These differ in the process of retrieving the elements
- Queues will use a *first-in* and *first-out* approach, known as **FIFO**
- Stacks use a *last-in* and *fist-out* approach, known as **LIFO**

Linked Lists (4)

Queues (FIFO)

- Queues consist of two references to the nodes in a list
  - **head**: points to a node in the list which is the top/first node
  - **tail**: points to a node in the list which is the last node
- The first node inserted into the list will be the first one to be retrieved
- Appending new elements to the list goes on the rear (the end)
- Retrieving elements will be taken from the front of the queue

Linked Lists (5)

Queue Walkthrough

- Click **Start** to proceed!

None

Start

Linked Lists (6)

Stacks (LIFO)

- Stacks consist of a single reference to the nodes in a list
  - **top**: points to a node in the list which is the first node
- The last node inserted into the list will be the first one to be retrieved
- Appending new elements to the list goes on the top (the start)
- Retrieving elements will also be taken from the front of the queue

Linked Lists (7)

Stack Walkthrough

- Click **Start** to proceed!

None

Start

Linked Lists (8)

Retrieving Elements

- Accessing a particular element is different for a list compared to a linked list
  - lists can be performed in constant time, $O(1)$, when knowing which element you want to access
  - linked lists take longer, and is performed in $O(n)$
    - required to traverse the whole list to find the element
- Searching for a specific element in either is dependent upon the size of the list
  - therefore, the time complexity is $O(n)$
  - you are required to iterate through the entire list to find the element

Linked Lists (9)

Inserting Elements

- Inserting elements to a list can be done using:
  - `prepend()` - insert an element at the beginning of a list
  - `append()` - insert an element at the end of a list
- Prepending or appending elements at the beginning or end of a list is done in *constant time*

Linked Lists (10)

Removing Elements

- Removing elements from a list can be done using either:
    - `remove()` - removes an element from the beginning of a list
    - `pop()` - removes an element from the end of a list

Linked Lists (11)

Time Complexity

- Linked lists are **always constant** in their time complexity when it comes to inserting and deleting elements
- They have a performance advantage when implementing a **queue**
  - elements inserted and removed at the beginning of a list
- However, linked lists have the same performance when implementing a **stack**
  - elements are inserted and removed at the end of the list

Goodbye

Goodbye (1)

Questions and Support

- Questions? Post them on the **Community Page** on Aula
- Additional Support? Visit the Module Support Page
- Contact Details:
  - Dr Ian Cornelius, ab6459@coventry.ac.uk