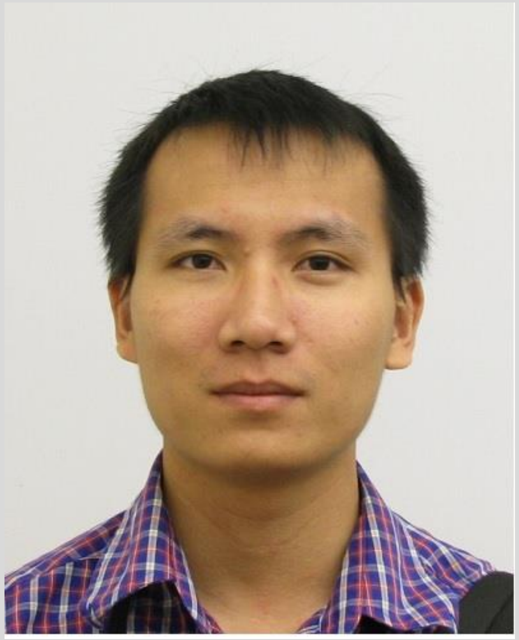
Data Structures and Algorithms

RMIT UNIVERSITY

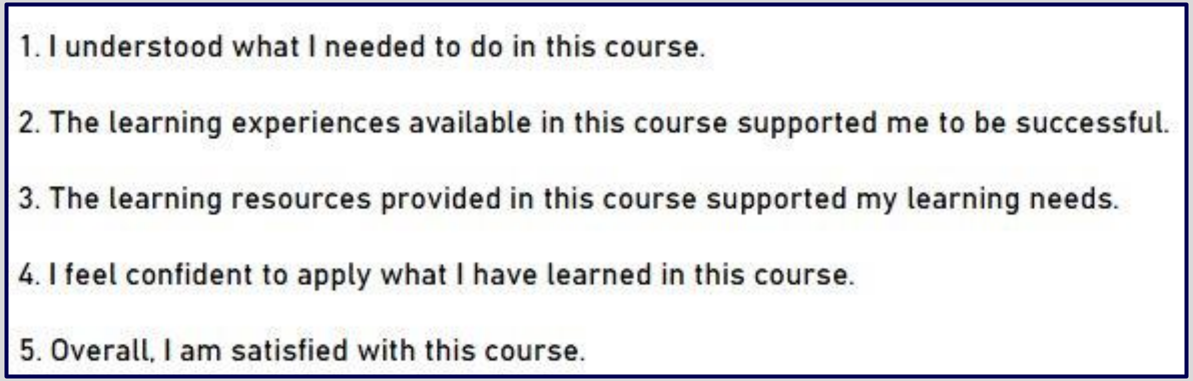
Teaching Team

Tri Dang Tran (SGS)
tri.dangtran@rmit.edu.vn



Tuan Anh Hoang (HN)
anh.hoang62@rmit.edu.vn

RMIT Learning Experience



1. I understood what I needed to do in this course.

2. The learning experiences available in this course supported me to be successful.

3. The learning resources provided in this course supported my learning needs.

4. I feel confident to apply what I have learned in this course.

5. Overall, I am satisfied with this course.

- Item #1: course learning outcomes, learning materials, weekly exercises, sample tests, dedicated group project discussion
- Item #2: live and recorded lectures, face-to-face tutorials, one-on-one meeting and consultation
- Item #3: lecture slides, tutorial solutions, sample tests and solutions, external references
- Item #4: weekly summaries, assessment feedback, course consultation

Assessments

- Weekly Quizzes (10%)
  - Open in 48 hours after lectures
  - Four multiple choice questions, open book, individual, 10 minutes duration
- Midterm Test (20%)
  - Week 5, in-class, limited resource access, individual
  - Sample problems/solutions will be provided
- Group Project (30%)
  - Requirement released: Week 5; Submission due: Week 11
  - Presentation: Week 12/13
  - Group of 4 or 5
- Final Test (40%)
  - Week 12/13, in-class, limited resource access, individual
  - Sample problems/solutions will be provided

Course Goals

- Theory:
    - Learn a variety of algorithms and data structures
    - Understand how to estimate running times of algorithms
- Practice:
    - Measure algorithm complexity and performance empirically
    - Design & implement various algorithms and data structures to solve a variety of problems

Why?

Simple answer:

- We want to ***solve problems***, so we need to have the ***tools*** (data structures & algorithms)
- We want to ***select the right tools***, hence the need to understand how the tools work and how efficient they are

More importantly:

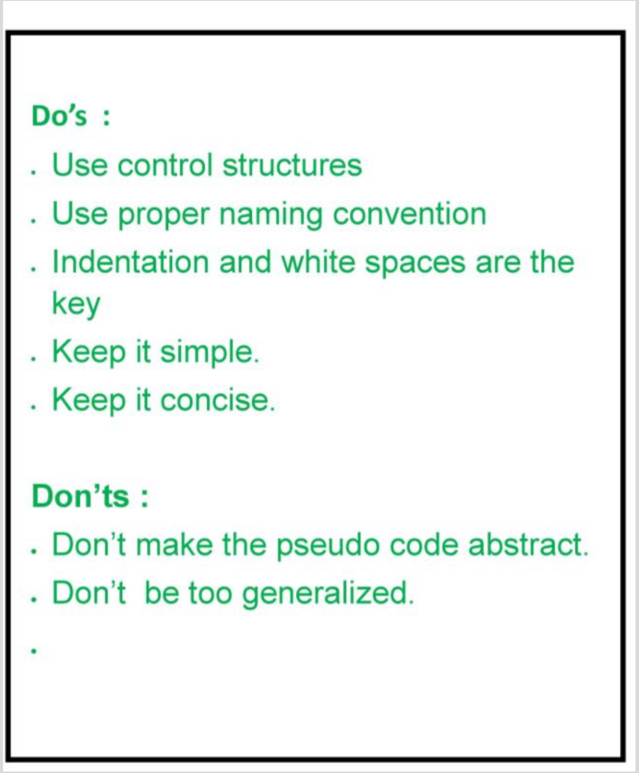
- **It makes you a better problem solver and programmer!**

Requirements

- This course is **<u>not</u>** about programming, but we do need a language for the practical parts
    - We use Java because all of you are familiar with it
- At a minimum, you should be comfortable with:
    - Basic data types and structures
    - Defining new classes and data types
    - Inheritance and polymorphism
- **Pseudocode** is also used to describe algorithms

What is Pseudocode

- Pseudocode/Pseudo-code/Pseudo Code
  - Is written in English
  - Help you learn to succinctly and clearly describe a solution
  - Used in job interviews and by management
- Some of the assessments may ask you to provide the answer using Pseudocode



Do's :

. Use control structures

. Use proper naming convention

. Indentation and white spaces are the key

. Keep it simple.

. Keep it concise.

Don'ts :

. Don't make the pseudo code abstract.

. Don't be too generalized.

.

Examples of Pseudocode

```
N = 10  // usually, uppercase for constant
total = 0  // variable and assignment
counter = 0
grades[N]  // array of N elements
while counter < N  // conditional
        total = total + grades[counter]
        counter = counter + 1
average = total / N
print average
```

- Use dot to access properties of objects (student.GPA)
- Declare function
- function add(a, b)
    return a + b
- There is no standard - but it has to be clear!!!

Housekeeping

- Course will be a combination of lectures and labs
    - Ratio will vary
- Come to class/Teams on time
- Sit up front (N/A in online mode)
- Be professional
    - Please don’t disturb your classmates
    - You are welcome to step up for phone calls etc. but please do not disturb your peers

We are here to help

- We are responsible for this course
  - You may contact us via email to ask questions or setup an appointment
  - Using the discussion forum to ask technical questions are highly recommended
  - Please do not wait until the day before the due date for clarifications or to seek help
- Be sure to read Course Announcements (ensure that you turn ON notification of new announcements)
- Early feedback will be provided on the first quiz; all assessments have written feedback

Learning objectives

- Understand the concept of ***algorithms*** and ***data structures***, and the motivation behind their analysis
- Learn about different ***abstract data types*** and using built-in data structures to implement them

What is an Algorithm?

What is an Algorithm?

- An ***algorithm*** is a sequence of unambiguous instructions or steps for solving a problem
- An algorithm should be:
  - Independent of programming language
  - A finite, deterministic and effective problem-solving method

Example - Sum Range Query

- **Problem:** answer range queries
- **Input:** an array X of numbers and a number of ranges [L1, R1], [L2, R2], [L3, R3], etc.
- **Output:** the sum of the ranges S1, S2, S3, etc.
- Example
  - Array: [5, 7, 8, 1, 3, 6, 9, 2, 4] (0-based index)
  - Range/Output: [0, 8] => 45, [1, 4] => 19, [0, 2] => 20

Range Query – Algorithm 1

-
  ```
  sum = X[L]
  start = L + 1
  while (start <= R)
      sum += X[start]
      start++
  print sum
  ```
- How many steps are needed if
  - Array size ~ 1M
  - Number of queries per day ~ 1M

Range Query – Algorithm 2

- Assume the array is static, we can precompute the sums of all the ranges [0, 0], [0, 1], [0, 2], [0, 3], [0, 4], …, [0, N-1]. These sums are called prefix sums.
- `We have prefixSum[Z] = X[0] + X[1] + … X[Z]`
- `So, X[L] + X[L+1] + … + X[R] = prefixSum[R] - prefixSum[L-1]`
- How to construct the prefixSum array?
- How many steps to answer a range query?
- What is the disadvantage?

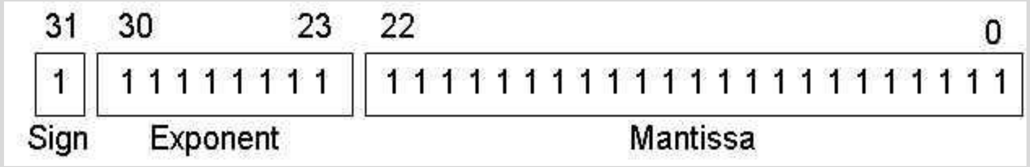
What is Data Abstraction?

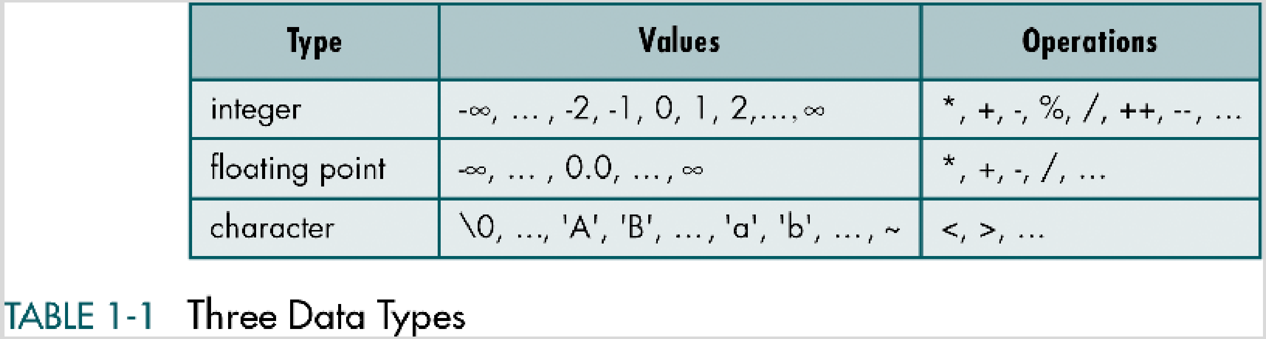
What is Data Abstraction

- A representation of data and operations allowed on that data
  - Asks you to think what you can do to data independently of how it is done
  - Implementation details are hidden
- All programming languages provide built-in Abstract Data Types and operations that you can perform on them, for example:
  - Data type int models integers between a range (usually –2^31 to 2^31 – 1) and supports the operations create, assign, add, subtract, multiply, etc.
  - We do not need to know how the data is represented
- For example, how are integers or floating-point numbers represented?

Abstract Data Types

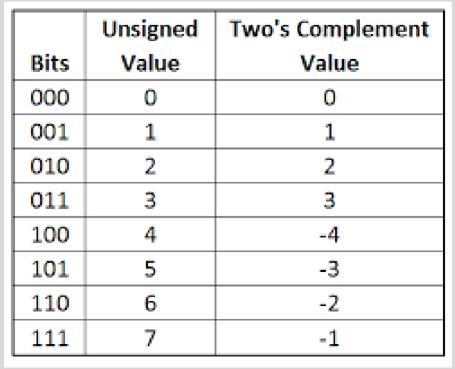
- Characters – ASCII
- Integers – two's complement
- Floating point





Type	Values	Operations
integer	-∞, ... , -2, -1, 0, 1, 2,..., ∞	*, +, -, %, /, ++, --, ...
floating point	-∞, ... , 0.0, ... , ∞	*, +, -, /, ...
character	\0, ..., 'A', 'B', ... , 'a', 'b', ... , ~	<, >, ...

TABLE 1-1 Three Data Types



Bits	Unsigned Value	Two's Complement Value
000	0	0
001	1	1
010	2	2
011	3	3
100	4	-4
101	5	-3
110	6	-2
111	7	-1

The String ADT

- Models an indexed sequence of characters with the following operations
  - Java: length(), charAt(int i), indexOf(String p), substring(int i, int j), split(String delim),…..
- We do not need to know how strings, or even the characters, are represented
- Why not use arrays of characters instead of Strings?

Defining your own ADTs

- ADTs is a key programming technique for procedural, functional and OO languages
  - Picking the right one for the job is key step in design

    Get your data structures correct first, and the rest of the program will write itself – David Jones

    Bad programmers worry about the code. Good programmers worry about data structures and their relationships – Linus Torvalds
- Example, the Fraction ADT
  - Why would this be important/useful?

The Fraction ADT

- Why have an ADT for Fractions
  - Speed of integer arithmetic versus real numbers
  - Ease in comparing two fractions
  - Can approximate irrational numbers using rational numbers
- How to implement an ADT for Fractions
  - Many options - two integers or strings for the numerator and denominator or a triplet for the whole part and remaining fraction
  - Operations - add, subtract, multiply, divide, print, round up, etc.

Using the Fraction ADT

- Operations
  - Create: F1 = Fraction(Numerator, Denominator), Assign: F2 = assignFrac(7,3), addFrac(F1, F2), multipyFrac(F1, F2), etc.
  - The implementation will depend upon your internal representation
- Question
  - How to simplify a fraction – e.g. 16/12 becomes 3/2?
  - How do you Print a fraction?

Abstract Data Type (ADT)

Key ADTs include

- Set, sequences, dictionary/map, stack, queue, priority queue, graph, tree

Java has thousands of ADTs, e.g.

- Standard : Integer, Double, String
- Counter, Accumulator, Stopwatch
- Stack, Queue, Bag, Graph, Digraph, Edge

Sets

- A collection of distinguishable objects, called members or elements, e.g.
  - Binary - {0,1}
  - Character - {c, a, y, s, t}
  - Word – {apple, bird, cat, dog}
  - Roman numerals – {I, V, X, D, C, M)
- Sets do not impose any ordering (but some specific types of sets do)
- Typical operations: add, remove, search
- Each element may only appear in the set once - if they may appear multiple times it is referred to as a bag or multiset

Sequences and Lists

A sequence is a collection of elements in which the **order** of the elements must be maintained, and elements can occur **any number of times**.

- A sequence is simple a multiset in which the **order** is important
- Typical operations: add, remove, search
- In computer science, this is also referred to as a list (ordered collection of elements)

Sequences

Examples of sequences

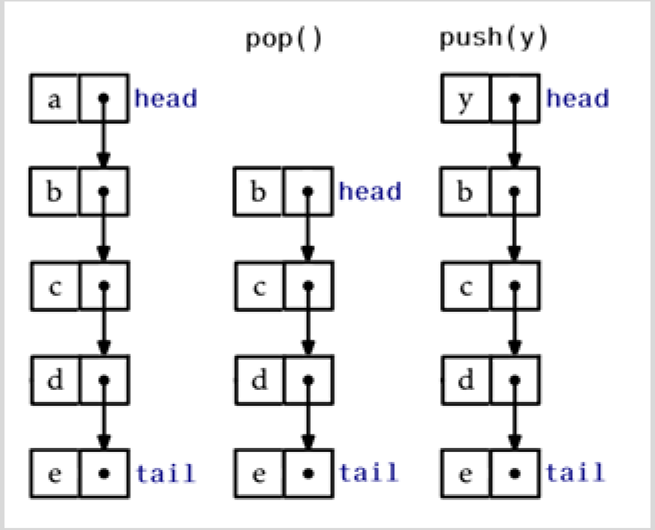
- Binary: 10101010
- English: "Hello, world"
- Genomic: cgagttcgatgtgactgatgatgttgaac

Stack

A stack is a collection with two operations:

- **Push** adds new element at the top of the stack
- **Pop** removes element at the top from the stack

Stack ADT implements **LIFO principle**, i.e., Last In, First Out

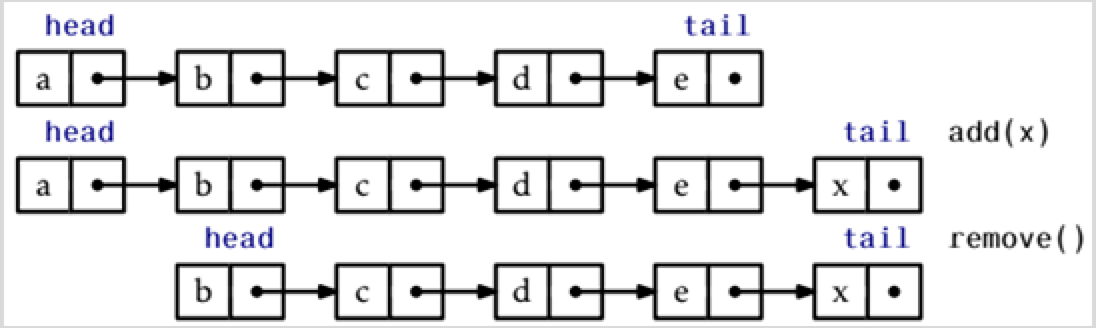


Queue

A queue is a collection with two operations

- **Enqueue** adds new elements at the back of the queue
- **Dequeue** removes elements at the front from the queue.

Queue ADT implements **FIFO principle**, i.e., First In, First Out



Dictionaries/Maps

A dictionary/map is a collection of **(key, value)** pairs, such that each key can only appear once in the dictionary/map.

Example: a sample dictionary

Key	Value
A+	HD
A	HD
B	DI
C	CR

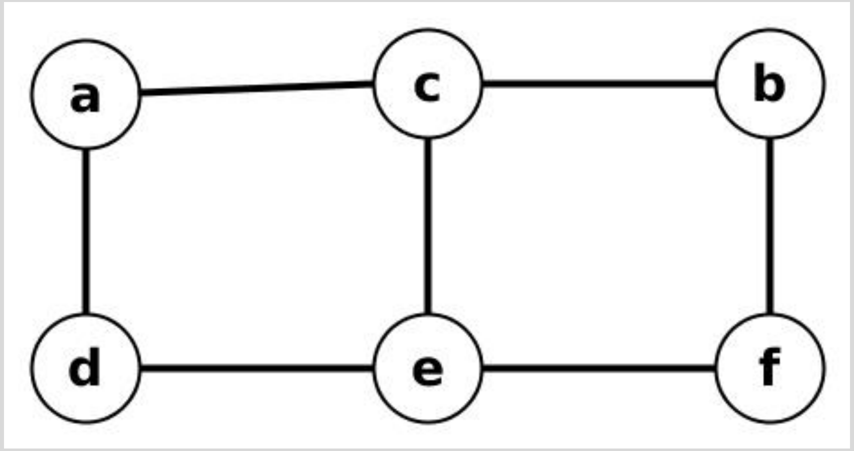
Graphs

A graph $G = \{V, E\}$ is defined by a pair of two sets: a finite set V of items called **vertices** and a set E called **edges**, representing links/relations/connections between pairs of vertices.

Example:

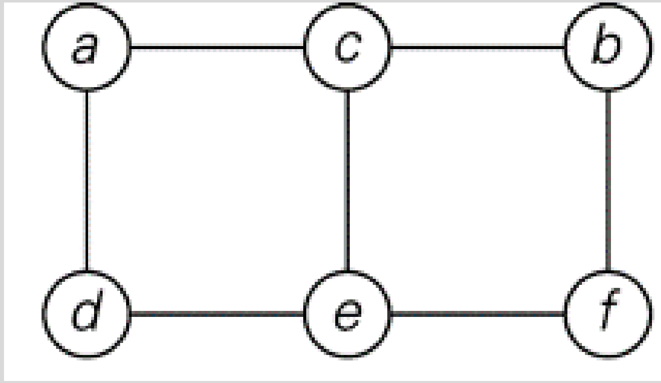
$V = \{a, b, c, d, e, f\}$

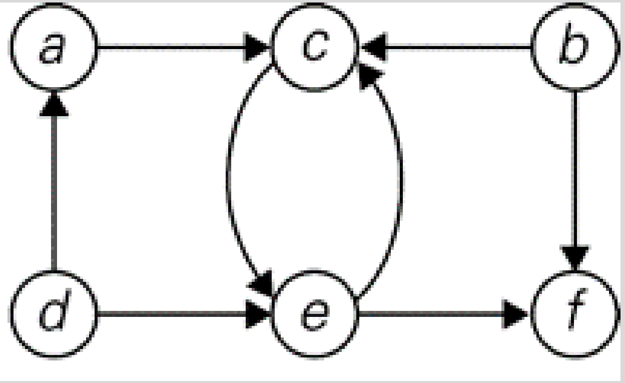
$E = \{(a, d), (a, c), (d, e),$
$\quad (c, e), (c, b), (e, f), (b, f)\}$



Graphs

- A graph *G* is **undirected** if the edges do not have a direction, i.e., all of the pairs of vertices in *E* are unordered.
- A graph *G* is **directed** if the edges form a direction, i.e., all of the pairs of vertices in *E* have an ordering imposed.





Graphs

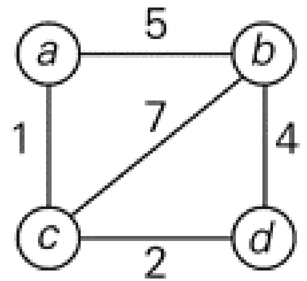
- How to represent a graph without drawing it?
  - Adjacency matrix and adjacency lists
  - If undirected graph, we only need one half of the matrix
  - The diagonals are marked with 0 as there is no edge from a to a, etc.

$$
\begin{array}{c|cccccc|}
 & a & b & c & d & e & f \\
\hline
a & 0 & 0 & 1 & 1 & 0 & 0 \\
b & 0 & 0 & 1 & 0 & 0 & 1 \\
c & 1 & 1 & 0 & 0 & 1 & 0 \\
d & 1 & 0 & 0 & 0 & 1 & 0 \\
e & 0 & 0 & 1 & 1 & 0 & 1 \\
f & 0 & 1 & 0 & 0 & 1 & 0
\end{array}
$$

$$
\begin{array}{l}
\boxed{a} \rightarrow c \rightarrow d \\
\boxed{b} \rightarrow c \rightarrow f \\
\boxed{c} \rightarrow a \rightarrow b \rightarrow e \\
\boxed{d} \rightarrow a \rightarrow e \\
\boxed{e} \rightarrow c \rightarrow d \rightarrow f \\
\boxed{f} \rightarrow b \rightarrow e
\end{array}
$$

Graphs

- What if edges have weights associated with them?
- Note, here too, that the distance from a to a is infinite….



$$\begin{array}{c|cccc} & a & b & c & d \\ \hline a & \infty & 5 & 1 & \infty \\ b & 5 & \infty & 7 & 4 \\ c & 1 & 7 & \infty & 2 \\ d & \infty & 4 & 2 & \infty \end{array}$$

$\boxed{a}$ $\rightarrow$ b, 5 $\rightarrow$ c, 1

$\boxed{b}$ $\rightarrow$ a, 5 $\rightarrow$ c, 7 $\rightarrow$ d, 4

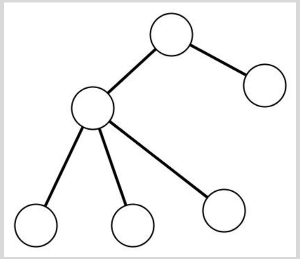
$\boxed{c}$ $\rightarrow$ a, 1 $\rightarrow$ b, 7 $\rightarrow$ d, 2

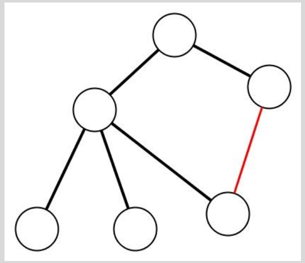
$\boxed{d}$ $\rightarrow$ b, 4 $\rightarrow$ c, 2

Trees

- A tree is a connected acyclic graph – i.e. no cycles

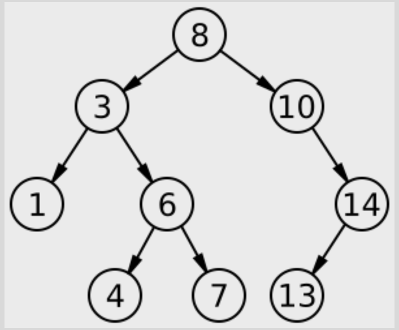
Acyclic graph





Not an acyclic graph

- Each node in the trees can have multiple edges
- Trees that have directed edges from the root to the leaves are called directed acyclic graphs (DAGs)
- Binary trees is an example of a DAG where each node has at most 2 children



A Binary Tree

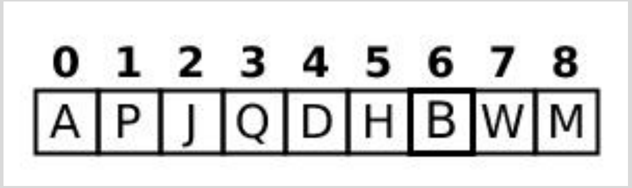
Data Structures

Data Structures

- A data structure is a particular way of organizing data in a computer so that it can be used effectively
- Data structures are used to implement an ADT
- Two important data structures are array and linked list which can be used to implement many ADTs
- We will learn more data structures as we progress through the course

Arrays and ADTs

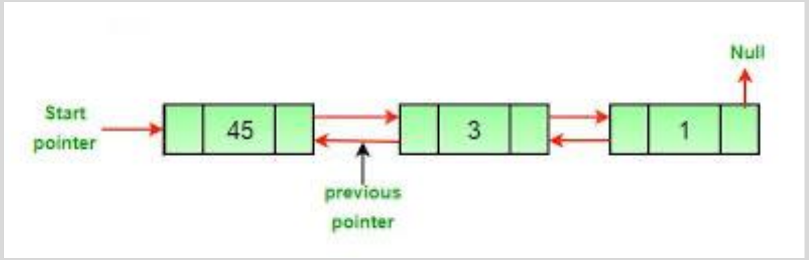
- Collection of elements, usually stored in a continuous manner
    - Support fast random access via indices
    - Need to estimate size beforehand

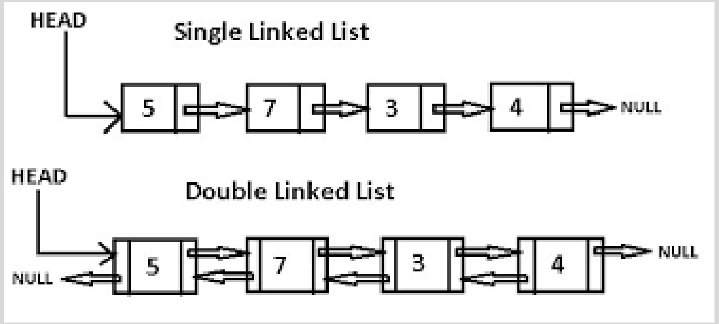


- Using Array to model ADTs
    - Stacks and queues can be implemented quite easily by keeping track of start and end entries
    - Can model sets and lists as elements in the array, but must keep array continuous when adding/deleting items
    - Can model trees but need to develop methods to relate items that are far apart in the array but are actually connected in the tree
    - We will cover this in more detail later in the lectures

Linked Lists

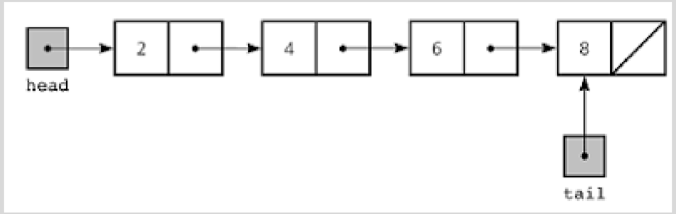
- Each node stores data and one pointer to the next node
  - There is a head pointer that points to the start of the list
  - Often there is also a tail pointer that points to the end of the list
- There are also doubly linked lists where each element has two pointers





Linked Lists and ADTs

- Lists and sets are implemented as elements of a linked list
- Stacks can be implemented where:
  - Push - adds an item at the front of the list
  - Pop – removes the item from the front of the list



- Queues can be implemented where:
  - Enqueue – adds an item to the front
  - Dequeue – removes an item from the end
- Graphs, as shown before, can be implemented using an array of linked lists
- We will cover this in more detail later in the lectures

To finish…

RMIT
UNIVERSITY

Learning Objectives

- Understand the concept and motivation for algorithms and data structures and how they are applied to problem solving
- Explained the use of key Abstract Data Types (ADTs) including sets, lists, stack, queue, graph, tree
- Demonstrated how the arrays and linked list data structures can be used to implement different ADTs

RMIT UNIVERSITY