BM40A1500 DATA STRUCTURES AND ALGORITHMS

# INTRODUCTION

2024

# WHAT YOU WILL LEARN?

1. Various commonly used data structures such as lists, binary search trees, hash tables.
   - How to select the best one for your application?
2. A set of algorithms to do various tasks efficiently, such as sorting and searching algorithms.
3. Methods to analyze algorithms, especially their efficiency.
4. A set of principles to design your own algorithms
5. Limits of computing.
   - Hard problems for which efficient solution is not known.
   - How to recognize such problems?
   - How to scope with them?

# DATA STRUCTURES

- A data structure is any data representation and its associated operations.
- An abstract data type (ADT) is the specification of a data type (collection of values) within some language, independent of an implementation.
  - Defined in terms of a type and a set of operations on that type.
  - Each operation is determined by its inputs and outputs.
  - An ADT does not specify how the data type is implemented.
  - A data structure is the implementation for an ADT.

- **Example:** an ADT for a list of integers could include the following operations:
  - insert (input: value and position, output: successful/unsuccessful)
  - delete (input: value or position, output: successful/unsuccessful)
  - find (input: value, output: position)
- A data structure can be the implementation of this ADT using linked list.

# ALGORITHMS

- **Problem:** a task to be performed
  - a function or a mapping of inputs to outputs.
- **Algorithm:** a method or a process followed to solve a problem
- An algorithm has all the following properties:
  1. It must be correct.
  2. It is composed of a series of concrete steps.
  3. There can be no ambiguity as to which step will be performed next.
  4. It must be composed of a finite number of steps.
  5. It must terminate.

- **Program:** an instance, or concrete representation, of an algorithm in some programming language.

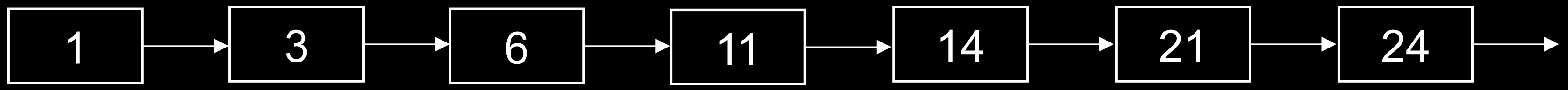
# EXAMPLE

- **Task**: we need to store an **ordered** set of numbers with possibility to **add** and **delete** numbers, and quickly **check whether certain numbers are found**.
- **Solution 1**: an array
  - Elements are stored in consecutive memory locations.
  - Quick to access numbers → quick to find numbers using binary search.
  - Time consuming to add and delete numbers.

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 3 | 6 | 11 | 14 | 21 | 24 | 30 | 40 | 42 |

# EXAMPLE

- Solution 2: a linked list
  - Each element contains a link to the next element.
  - Quick to add and delete numbers.
  - Not as quick to find a number.



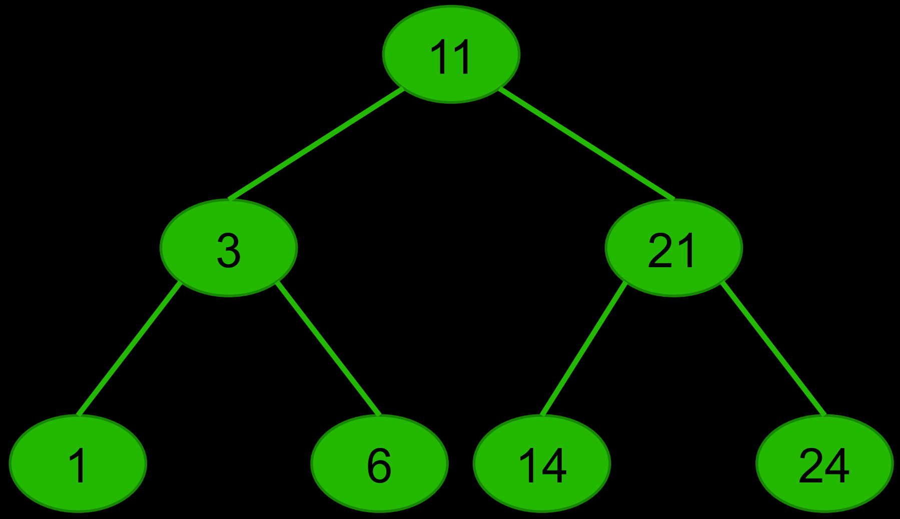
# EXAMPLE

- Solution 3: hash table
  - A search key value converted into a position within a hash table.
  - Quick to add and delete numbers.
  - Very quick to find numbers.

|  |  |  |  |  |  |  |  |  |  |
|---|---|---|---|---|---|---|---|---|---|
| 24 |  | 21 |  | 3 | 1 | 14 | 11 |  | 6 |
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |

  - Elements are not in order → no efficient way to do range queries.

- Solution 4: binary search tree
  - Quick to add and delete numbers.
  - Quick to find numbers if the tree is balanced.
    - How to keep the tree balanced?



# COURSE SCHEDULE

- Week 1: Introduction
- Week 2: Algorithm analysis
- Week 3: Lists, queues, and stacks
- Week 4: Hashing
- Week 5-6: Binary trees and heaps

- Week 7-8: Algorithm design principles
- Week 9-10: Graphs
- Week 11: Limits of computing and NP-completeness
- Week 12: Course recap

LUT
University