CISC 610 90:
Data Structures and Algorithms

Fall 2025

Daqing Yun

Introduction to Algorithms

Outline

- What are algorithms?
- Importance of algorithms
- What kinds of problems are solved by algorithms?
- Design of algorithms
- Data structures
- Analysis of algorithms
- Useful tips
- Course syllabus review

What is an Algorithm?

- An *algorithm* is any well-defined computational procedure that takes some value, or set of values, as *input* and produces some value, or set of values, as *output* in a finite amount of time.
- An algorithm is a tool for solving a well-specified *computational problem*.
- Here is how we formally define the *sorting problem*:
  - **Input:** A sequence of *n* numbers $\langle a_1, a_2, \ldots, a_n \rangle$.
  - **Output:** A permutation (reordering) $\langle a'_1, a'_2, \ldots, a'_n \rangle$ of the input sequence such that $a'_1 \leq a'_2 \leq \cdots \leq a'_n$.
- Thus, given the input sequence $\langle 31, 41, 59, 26, 41, 58 \rangle$, a correct sorting algorithm returns as output the sequence $\langle 26, 31, 41, 41, 58, 59 \rangle$. Such input sequence is called an *instance* of the sorting problem.
- In general, an *instance of a problem* consists of the input needed to compute a solution of the problem.

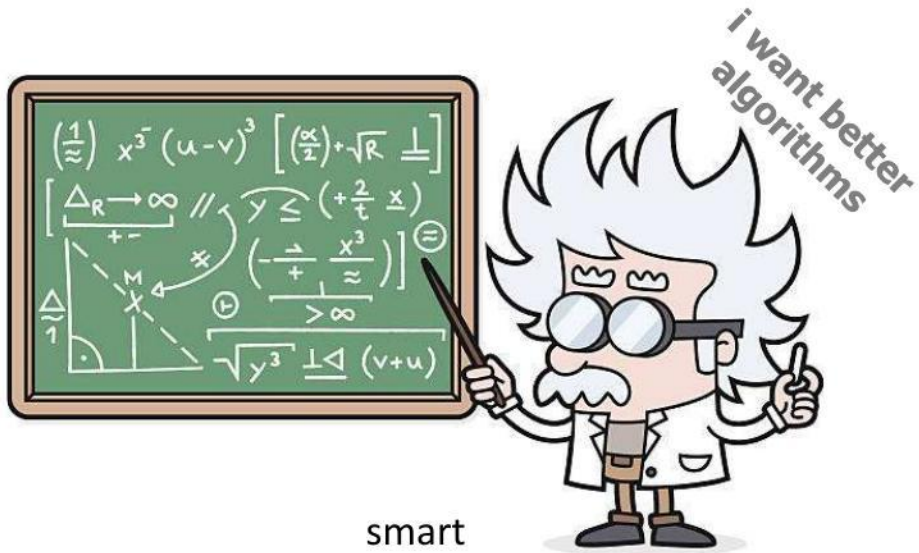
What is an Algorithm?

Algorithm ?= Program

- An algorithm for a computation problem is *correct* if, for every problem instance provided as input, it *halts* (finishes its computing in finite time) and outputs the correct solution to the problem instance.
- A correct algorithm *solves* the given computation problem.
- An incorrect algorithm might not halt at all on some input instances, or it might halt with an incorrect answer.
- An algorithm can be specified in English, as a computer program, or even as a hardware design.
- The only requirement is that the specification must provide a precise description of the computational procedure to be followed.

Importance of algorithms

rich people



smart mathematician

What kinds of problems are solved by algorithms?

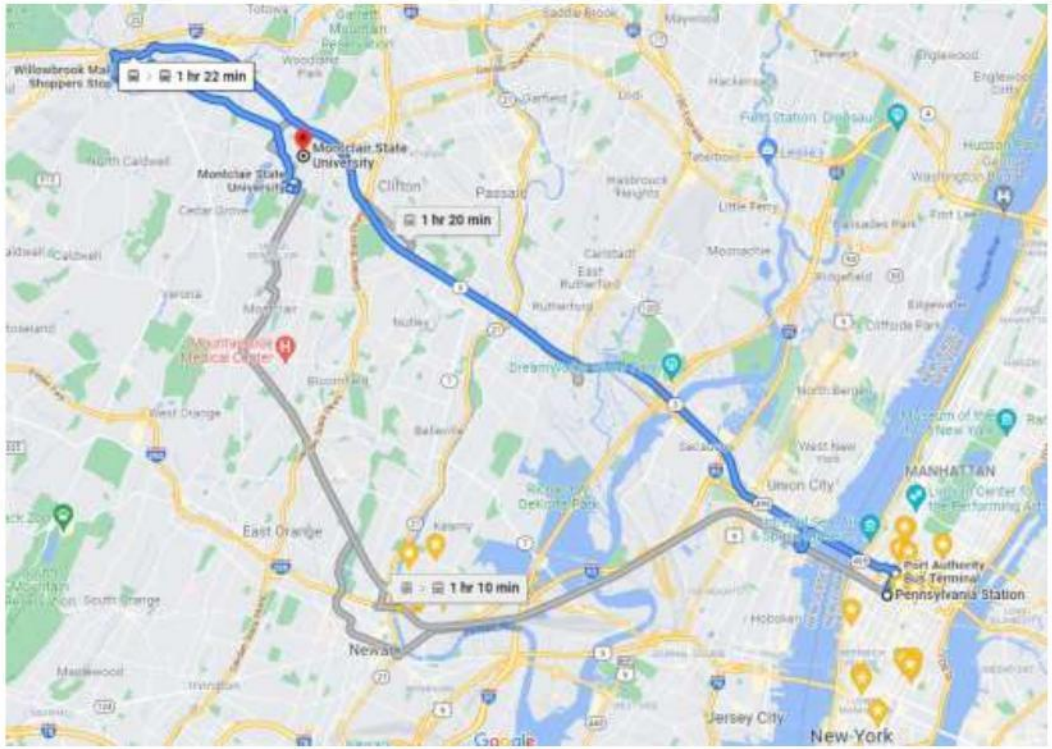
- Human Genome Project
  - Determining sequences of 3 billion chemical base pairs
- Internet
  - Finding good routes on which data will travel
  - Search engine
- Electronic Commerce
  - Public-key cryptography and digital signatures
- Manufacturing
  - Allocating limited resources in the most beneficial way

What kinds of problems are solved by algorithms?

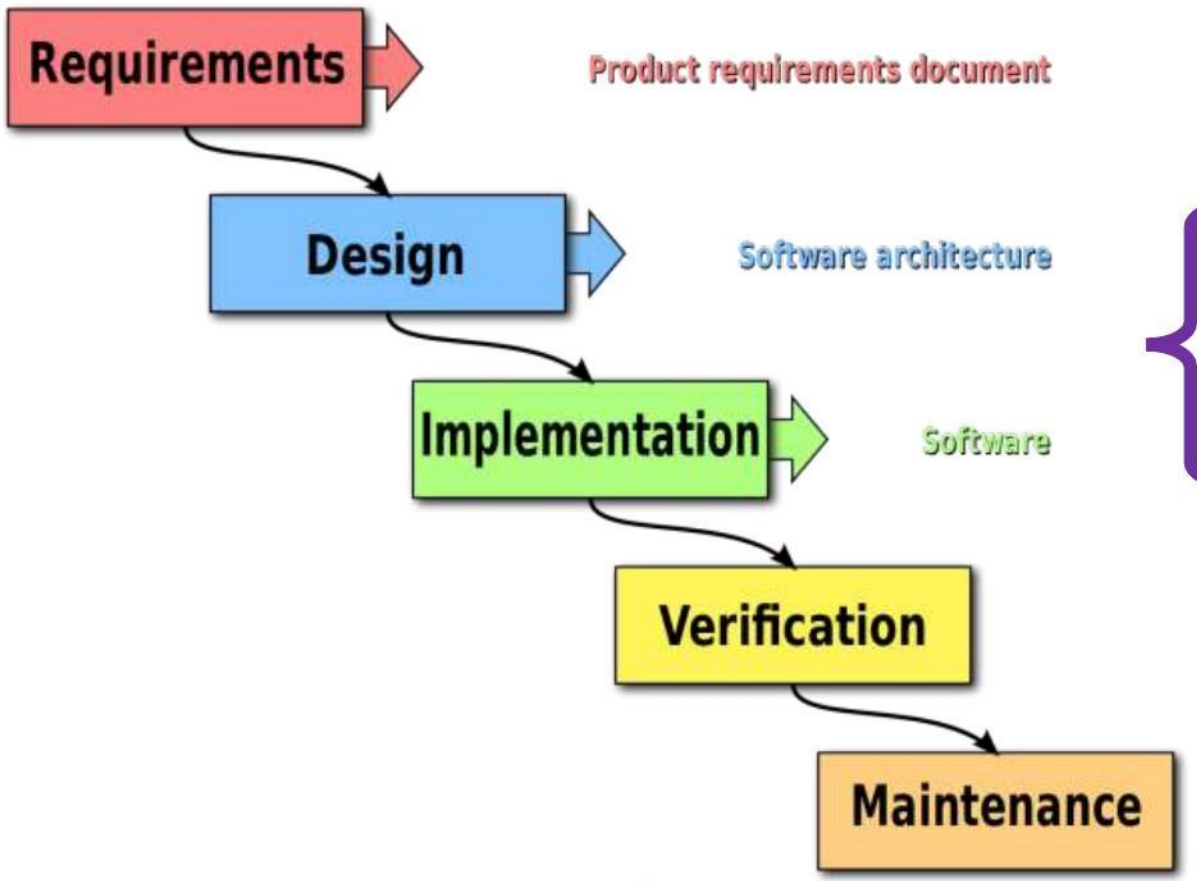
- Search problems
  - Sequence search, binary search, hash search, max (min, second) search
- Sort problems
  - Insertion sort, merge sort, quick sort
- Graph problems
  - Euler, Hamilton, TSP, graph coloring
- Combinatorial problems
  - 0-1 knapsack, task assignment
- Geometrical problems
  - Closest points

What kinds of problems are solved by algorithms?

- These lists are far from exhaustive
- Many exhibit two characteristics
  - There are many candidate solutions, most of which are not what we want, finding one that we do want can be quite a challenge
  - There are practical applications



Data structures and algorithms in software development



...
Algorithm Design
Algorithm Implementation
...

https://en.wikipedia.org/wiki/Waterfall_model

Design of algorithms

- How can I propose an algorithm for a specific problem?
- There are many ways to design algorithms
  - Divide-and-conquer
  - Recursive
  - Brute-force
  - Dynamic programming
  - Greedy
  - Back-tracking
  - Approximation algorithms
  - Random algorithms
  - …

Data structures

- A data structure is a way to store and organize data in order to facilitate access and modification.
- No single data structure works for all purposes.
- They all have strengths and limitations.

Data structures

- Choice of data structures
  - How are they arranged in relation to each other
  - What data are kept in memory
  - What are calculated when needed
  - What are kept in files, and how the files are arranged
  - …

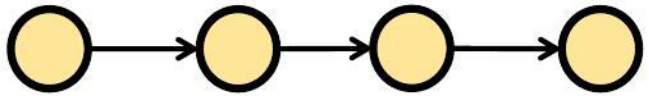
Algorithm + Data Structure = Program

Data structures

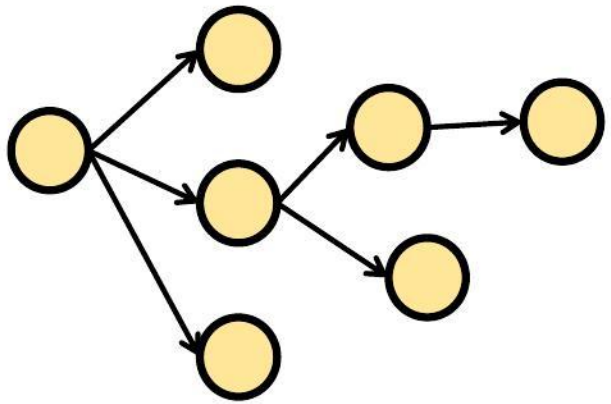
- Linear data structures and their operations
  - Arrays, stacks, queues, linked lists, strings, etc.
  - Searching, sorting, insertion, retrieval, etc.
- Non-linear data structures and their operations
  - Trees (binary trees, multiway trees, red-black trees, etc.),
  - Graphs (directed graphs, undirected graphs, directed acyclic graphs, etc.)
  - Traversal, node addition/deletion, reversal, etc.

Data structures

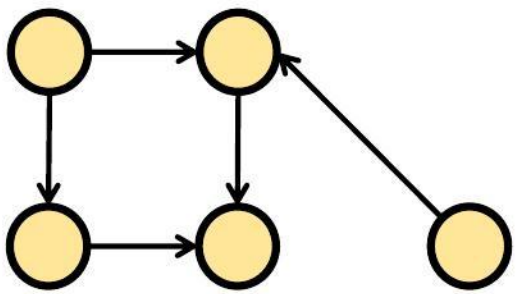
- Linear structures



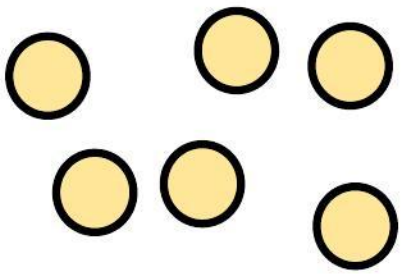
- Tree structures



- Graph structures

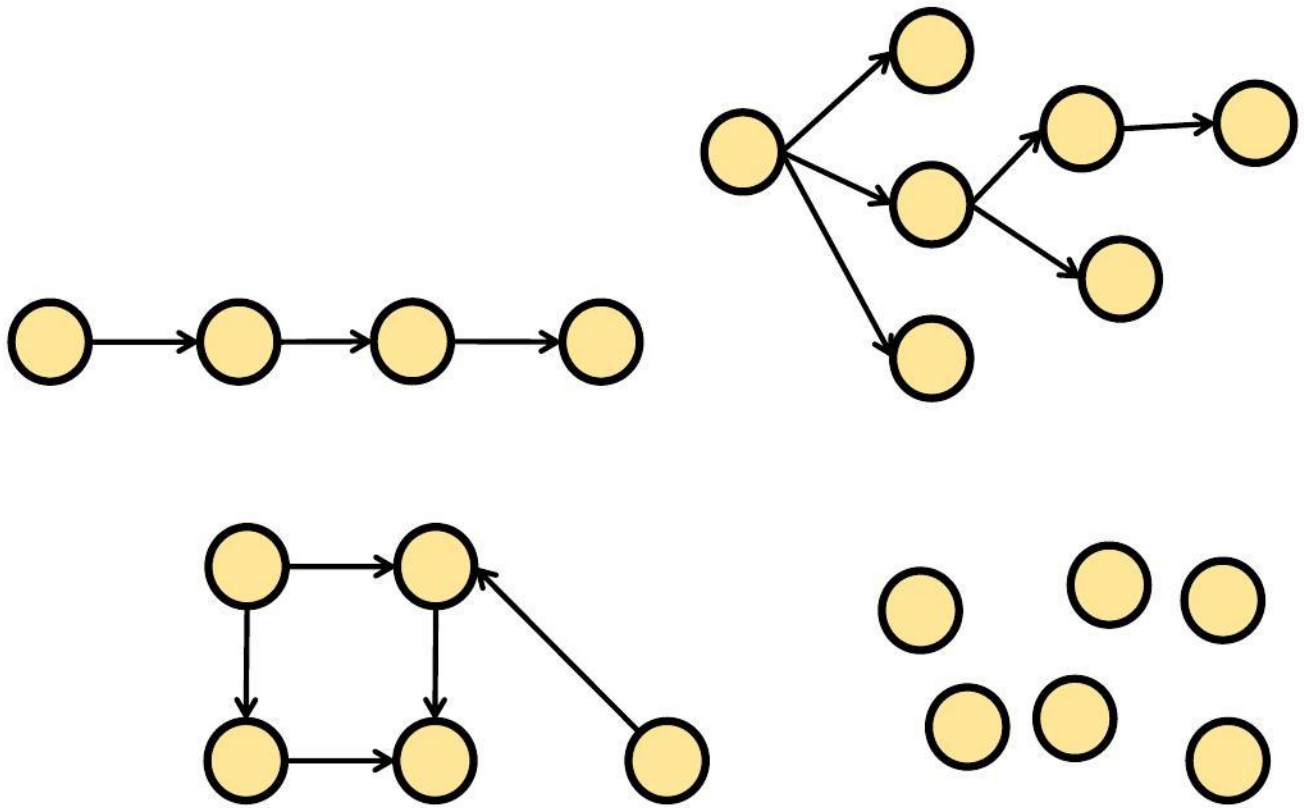


- Set structures



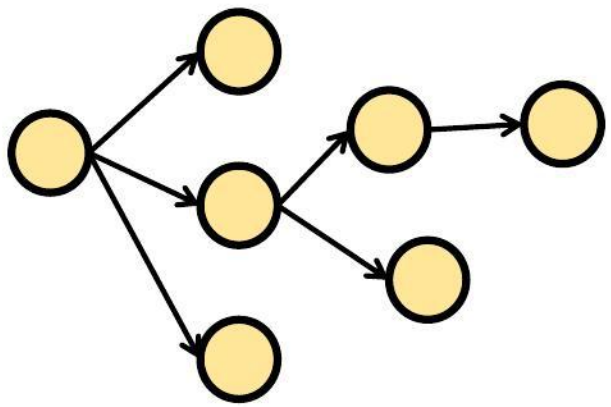
Data structures

- Linear structures
  - one → one
- Tree structures
  - one → many
- Graph structures
  - many → many
- Set structures
  - none → none



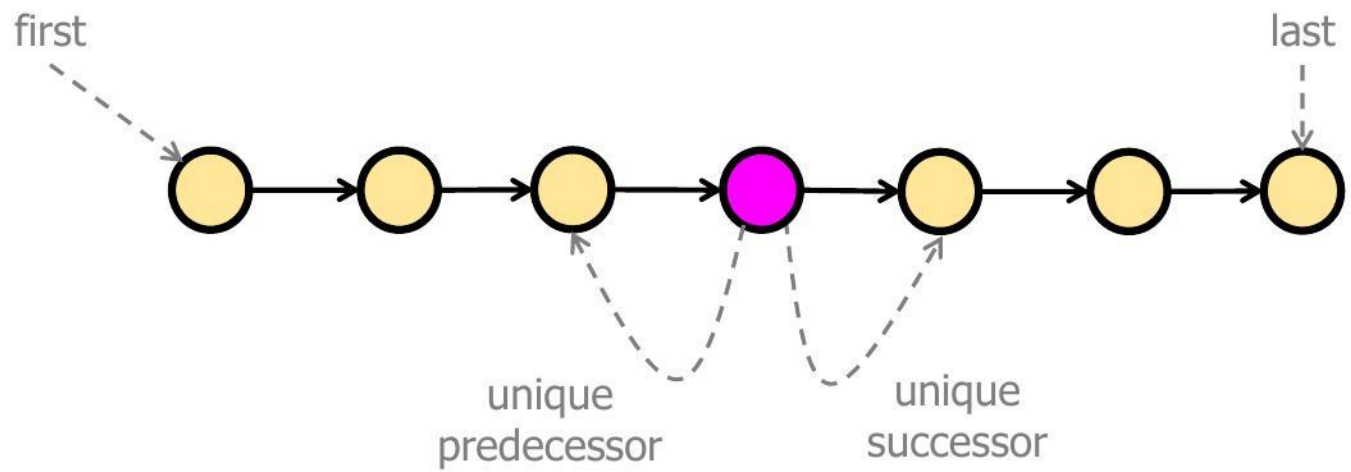
Data structures

- Core operations
  - **Add** element(s)
  - **Remove** element(s)
  - **Iterate** over (traverse) all elements
  - **Compare** elements



Linear data structures

- One-to-one relationship between elements
  - Each element has a **unique** predecessor
  - Each element has a **unique** successor

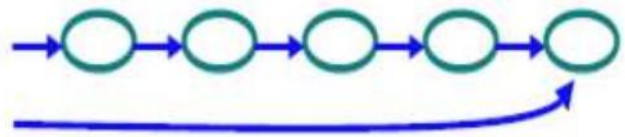


Linear data structures

- Core operations
  - Find first element (**head**)
  - Find next element (**successor**)
  - Find last element (**tail**)
- Terminology
  - **Head** – no predecessor
  - **Tail** – no successor

Example linear data structures

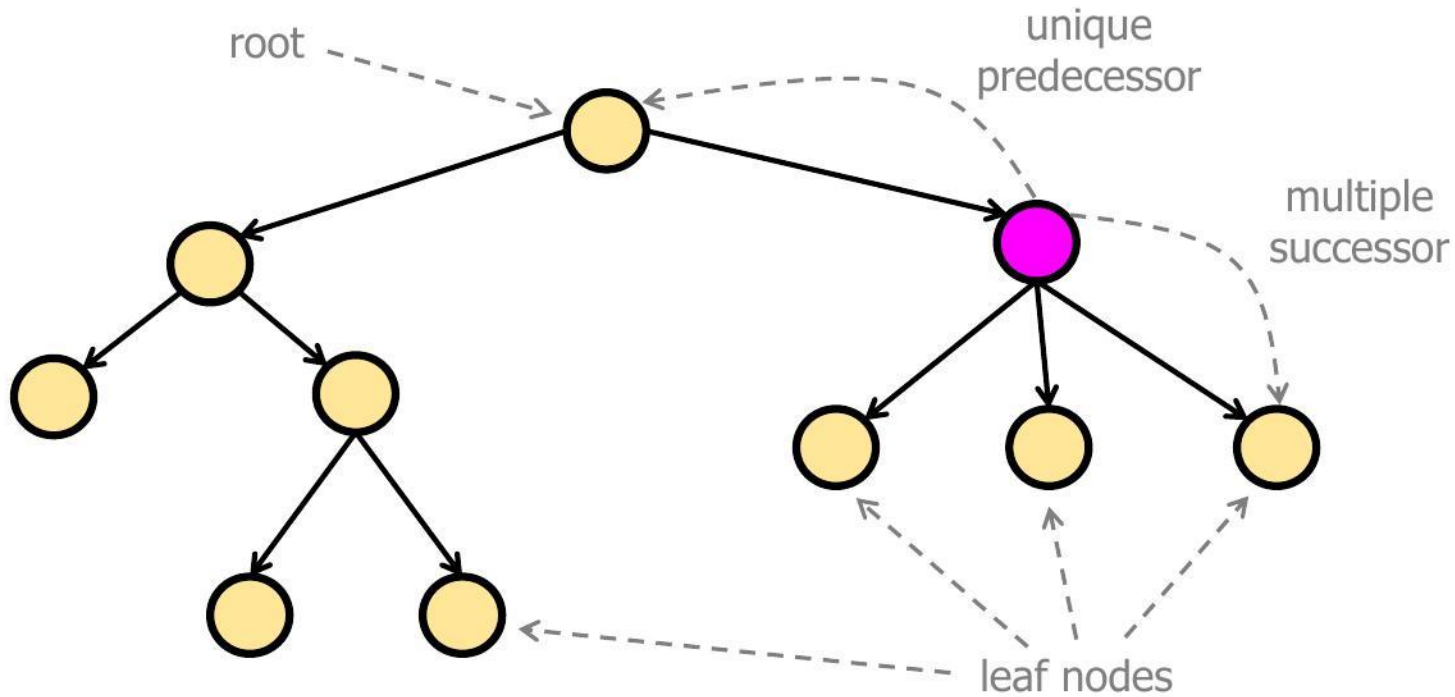
- Array
  - collection of elements, each identified by at least an index or key
- List
  - collection of elements in order
- Queue
  - Elements removed in order of insertion
  - First-In, First-Out (FIFO)



- Stack
  - Elements removed in opposite order of insertion
  - First-In, Last-Out (FILO)

Hierarchical data structures

- One-to-many relationship between elements
  - Each element has a **unique** predecessor
  - Each element has **multiple** successors

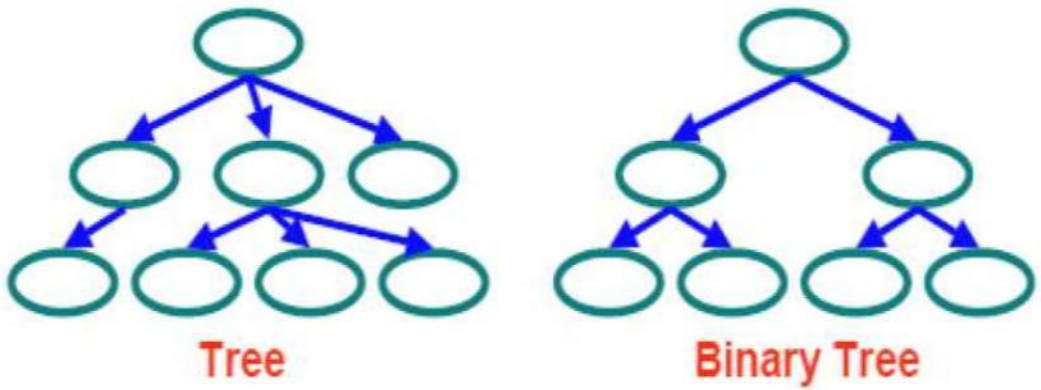


Hierarchical data structures

- Core operations
  - Find first element (**root**)
  - Find successor elements (**children**)
  - Find predecessor element (**parent**)
- Terminology
  - **Root** – no predecessor
  - **Leaf** – no successor
  - **Interior** (internal nodes) – non-leaf
  - **Children** – successors
  - **Parent** – predecessor

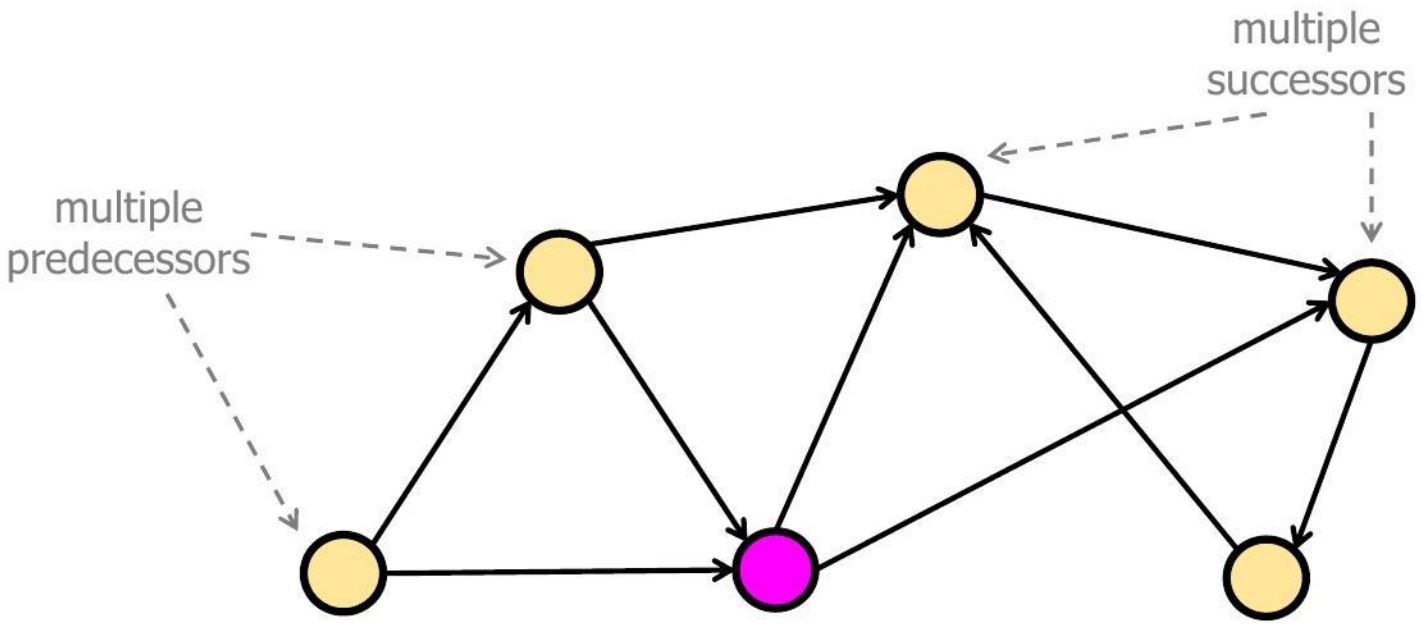
Example hierarchical data structures

- Tree
  - Single root
- Forest
  - disjoint union of trees
  - multiple roots
- Binary tree
  - tree with 0~2 children per node



Graph data structures

- Many-to-many relationship between elements
  - Each element has **multiple** predecessors
  - Each element has **multiple** successors

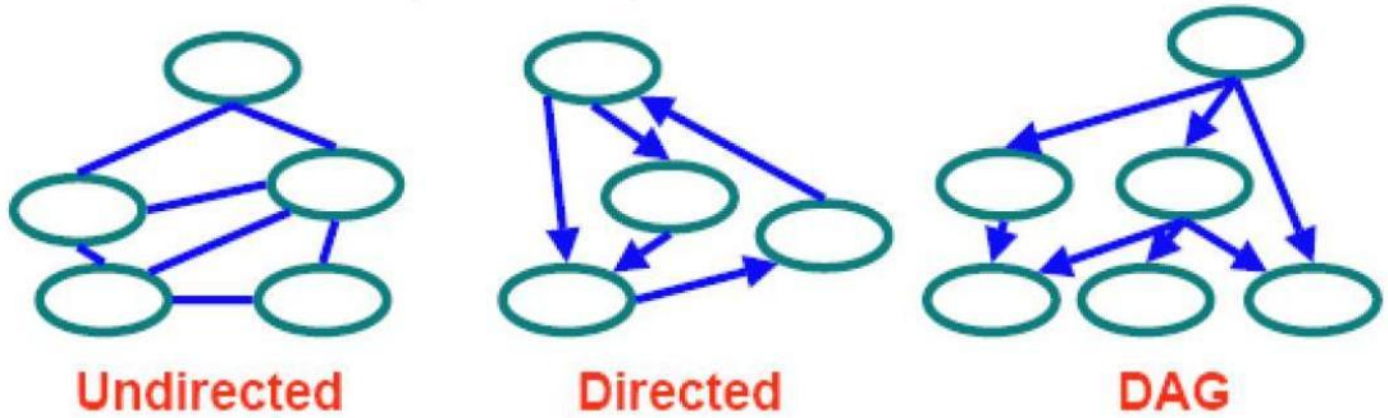


Graph data structures

- Core operations
  - Find successor nodes
  - Find predecessor nodes
  - Find adjacent nodes (neighbors)
- Terminology
  - **Directed** – traverse edges in one direction
  - **Undirected** – traverse edges in both directions
  - **Neighbor** – adjacent nodes
  - **Path** – sequence of (connected) edges
  - **Cycle** – path returning to the same starting node
  - **Acyclic** – no cycles

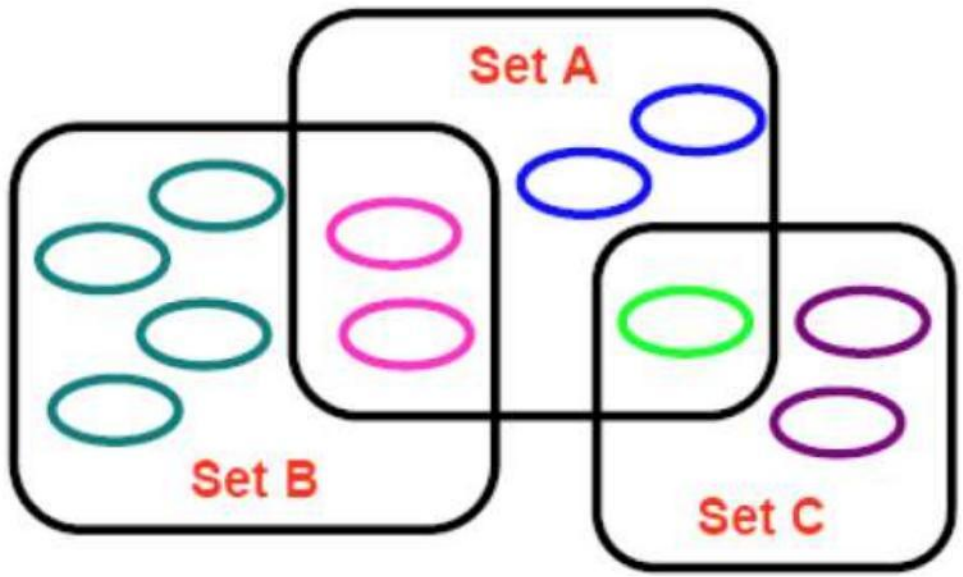
Example graph data structures

- Undirected graph
  - Undirected edges
- Directed graph
  - Directed edges
- Directed acyclic graph (DAG)
  - Directed edges, no cycles



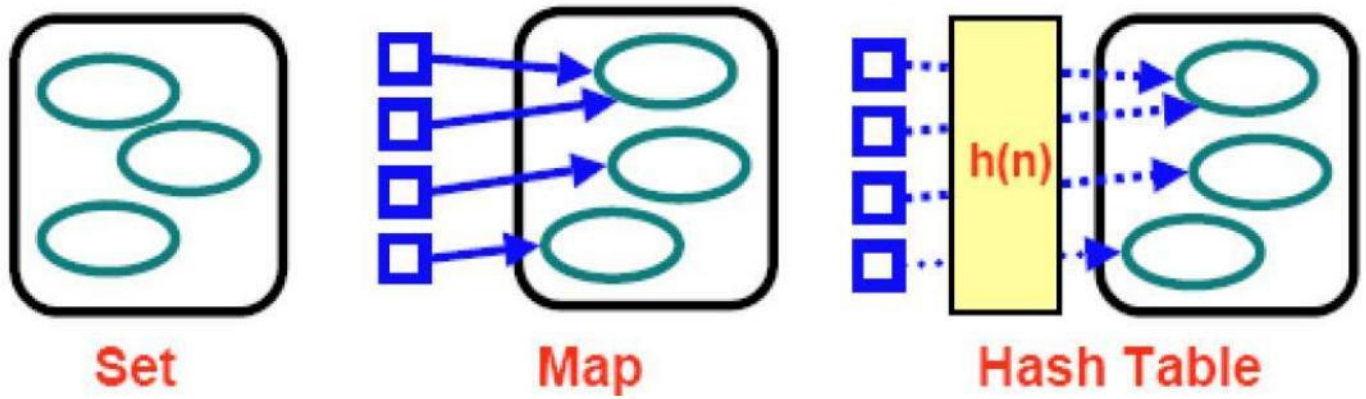
Set data structures

- No relationship between “peer” elements
  - Elements have no predecessor / successor
  - Only one copy of an element is allowed in set



Example set data structures

- Set
  - Basic set



- Map
  - Maps values to elements in a set

- Hash Table
  - Maps values to elements in a set using hash functions

Analysis of algorithms

Is the algorithm good enough?

- The theoretical study of computer program performance and resource usage
- What are more (or equally) important than performance?
  - modularity
  - correctness
  - maintainability
  - functionality
  - robustness
  - user-friendliness
  - programmer time
  - simplicity
  - extensibility
  - reliability

Why study algorithm and performance?

- Hard problems
  - There are some problems, for which no **efficient** solution is known (NP-complete problems)
  - But nobody has ever been able to prove that one cannot exist
  - If an efficient algorithm exists for any of these problems, then efficient algorithms exist for all of them
  - Some of these hard problems look very similar to the problems for which we do know of efficient algorithms
  - Many of these hard problems arise surprisingly often in practice!

Hard problems

- What are NP-complete problems? NP stands for nondeterministic polynomial (chapter 34).
- No one knows whether efficient algorithms exist for NP-complete problems.
- These problems have the remarkable property that if an efficient algorithm exists for any one of them, then efficient algorithms exist for all of them.
- Several NP-complete problems are similar, but not identical, to problems for which we do know of efficient algorithms.

Hard problems

- NP-complete problems arise surprisingly often in real applications.
- If you can show that a problem is NP-complete, you can spend your time developing an efficient approximation algorithm, that is, an algorithm that gives a good, but not necessarily the best possible, solution.
- The “traveling salesman problem” is a known NP-complete problem.

Alternative Computing Models – Parallel Algorithms

For many years, we could count on processor clock speeds increasing at a steady rate.

Because power density increases superlinearly with clock speed, chips run the risk of melting once their clock speeds become high enough.

In order to perform more computations per second, chips are being designed to contain several processing cores.

For these multicore computers, we need to design parallel algorithms (chapter 26).

Alternative Computing Models – Online Algorithms

Much of the work on algorithm design assumes that the input data is available when the algorithm begins running.

For many real-world problems, the input actually arrives over time, and the algorithm must decide how to proceed without knowing what data will arrive in the future.

In a data center, jobs are constantly arriving and departing, and a scheduling algorithm must decide when and where to run a job, without knowing what jobs will be arriving in the future.

Algorithms that receive their input over time, rather than having all the input present at the start, are *online algorithms* (chapter 27).

About this course

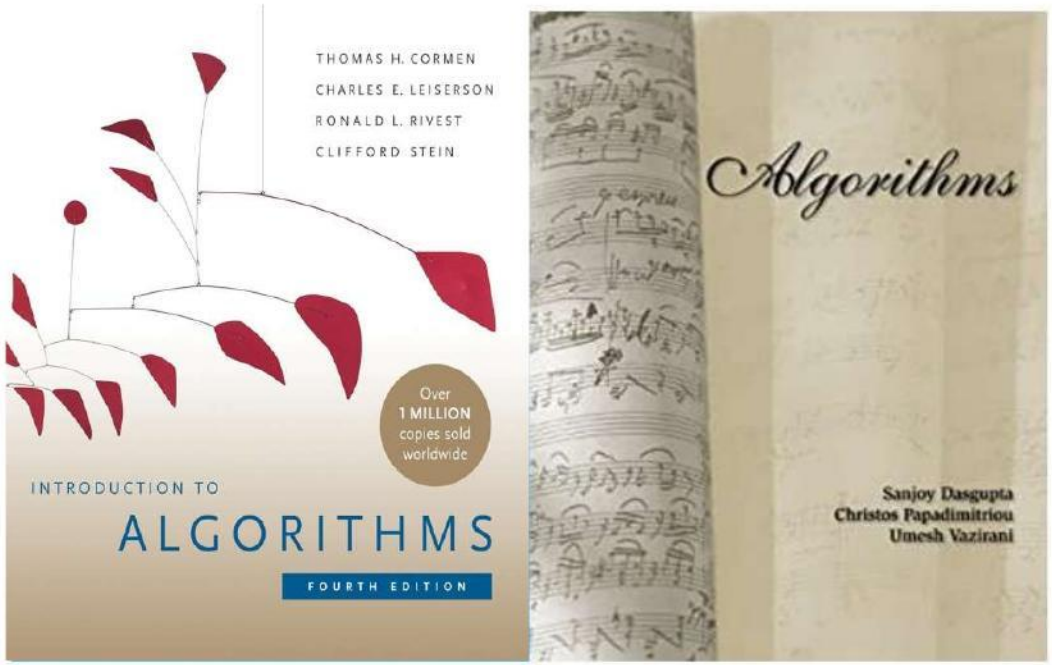
- A survey of algorithmic design techniques
- Abstract thinking
- How to develop new algorithms for (any/new) problems that may arise – problem solving!
- A survey of algorithm

  analysis techniques

Useful learning tips

Read ahead

- [CLRS] T. Cormen, C. Leiserson, R. Rivest, and C. Stein. ***Introduction to Algorithms***, The MIT Press, ISBN: 9780262533058/9780262046305.
- [DPV] S. Dasgupta, C. Papadimitriou, and U. Vazirani. ***Algorithms***, ISBN: 0073523402.



More about this course

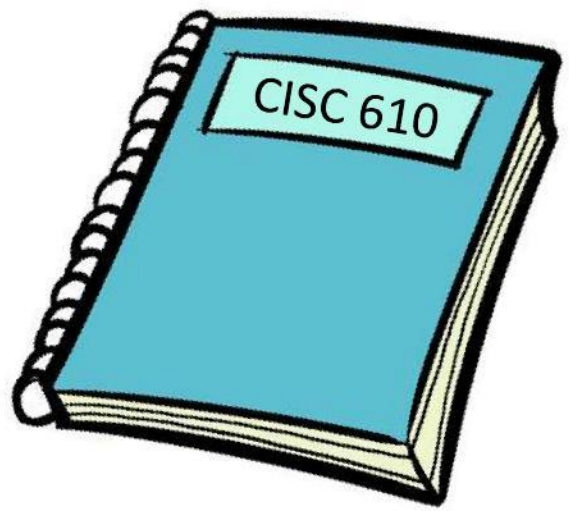
- Components:
  - Lectures (class participation, 10%)
  - Graded quizzes (exercises, 50%)
  - Project 1&2 (5% each)
  - Exam 1&2 (15% each)

More about this course

- Course project
  - Learn about an “advanced” topic about algorithms and prepare a presentation about it
- **See the textbook Part VII: Selected Topics**
- Example topics:
  - Artificial Neural Networks (ANNs)
  - Genetic Algorithms (GA)
  - Tabu Search (TS) algorithm
  - Simulated Annealing (SA)
  - Stochastic Approximation (SA) algorithm
  - … (**other algorithm(s) you find interesting**)

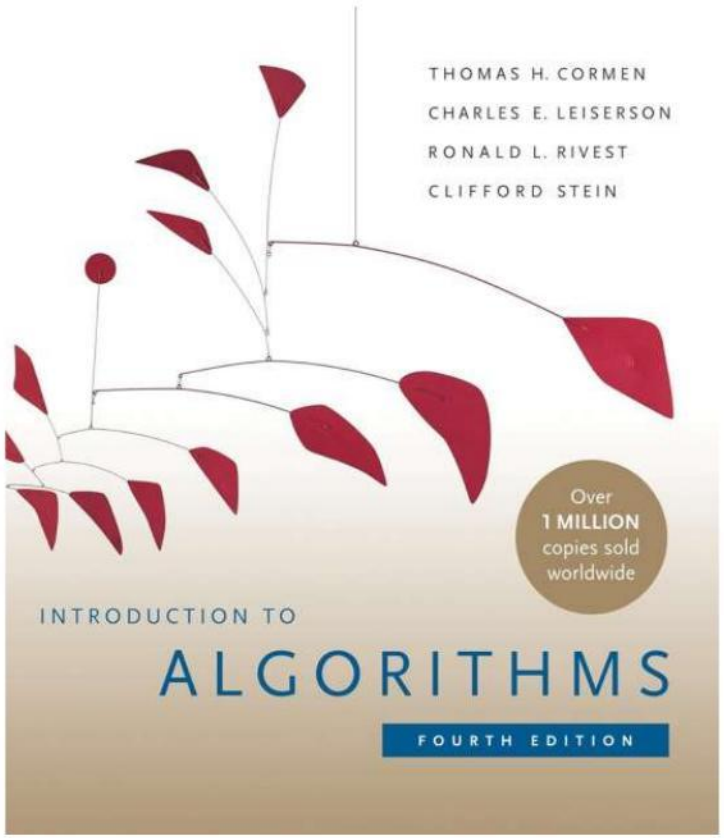
More about this course

- Let's check out the Canvas course page and syllabus …



Reading

- [CLRS] Chapter 1



Thanks ! ☺

Questions ?