Data Structures and Advanced Programming

Overview

Ling-Chieh Kung

Department of Information Management
National Taiwan University

My plan for today

- Introducing this course and the way we run it
- Giving lectures
- Helping you understand whether you should/may take this course

Road map

- **What are "data structures" and "advanced programming?"**
- Course policy

Data structures (DS)

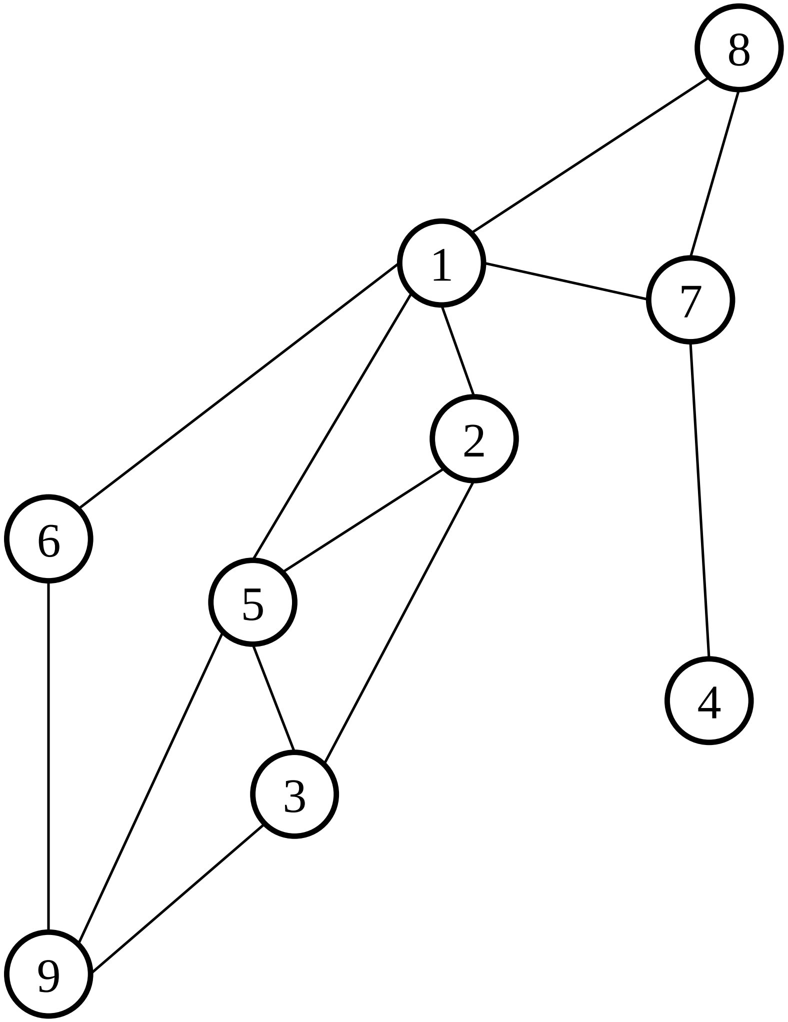
- A **data structure** is a specific way to **store** data.
- Usually it also provides interfaces for people to **access** data.
- Real-life examples: A dictionary.
  - It stores words.
  - It sorts words alphabetically.
- In large-scale software systems, there are a lot of data. We want to create data structures to store and manage them.
- We want our data structures to be **safe**, **effective**, and **efficient**.
  - Encapsulation: People can access data only through managed interfaces.
  - We can store and access data correctly.
  - The number of steps required for a task is small; consider a dictionary with words not sorted!

Data structures

- "**Computer Programming = Data Structures + Algorithms**."
  - To write correct programs, any data structure works.
  - To write "**good**" programs, data structures (and algorithms) matter.
  - Here goodness basically means **efficiency**, either **time** or **space**.
- Recall some examples we mentioned in Programming Design.

A graph

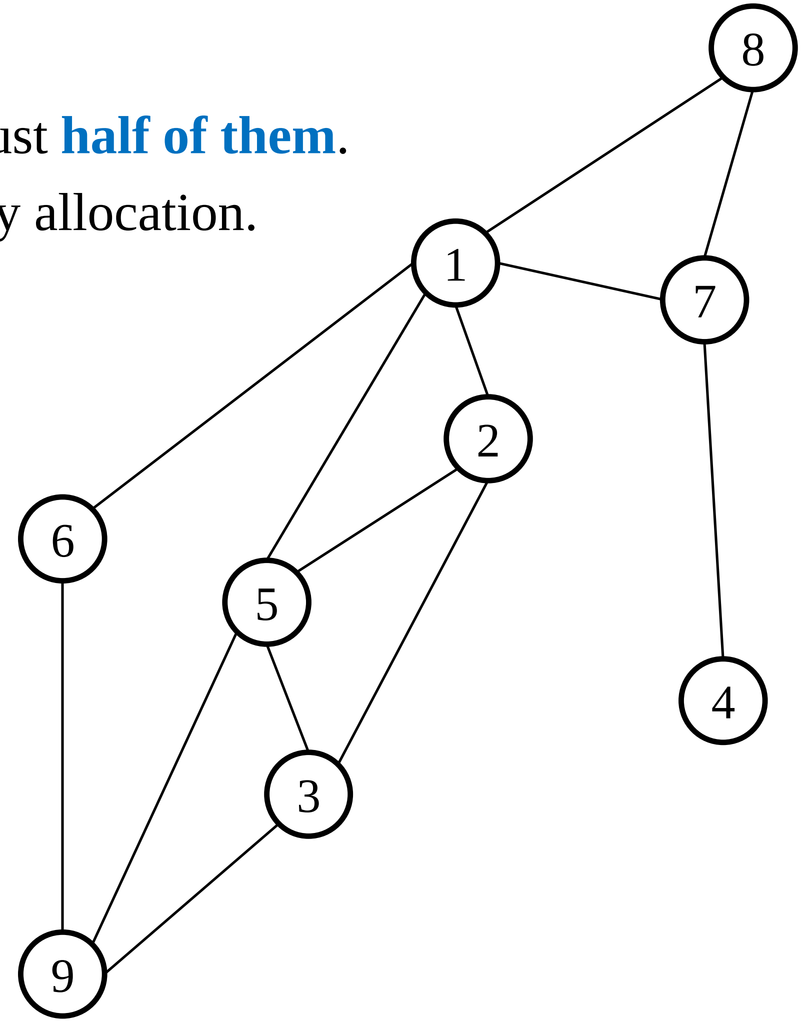
- Consider an **undirected graph** of n nodes and m edges.
  - In the example, $n = 9$ and $m = 13$.
  - In general, $m \leq \frac{n(n-1)}{2}$.
- Suppose that there is no weight on edges.
- How may we store the information of this graph?



Adjacency matrices

- As the matrix is symmetric, we may store just **half of them**.
  - We need to implement dynamic memory allocation.

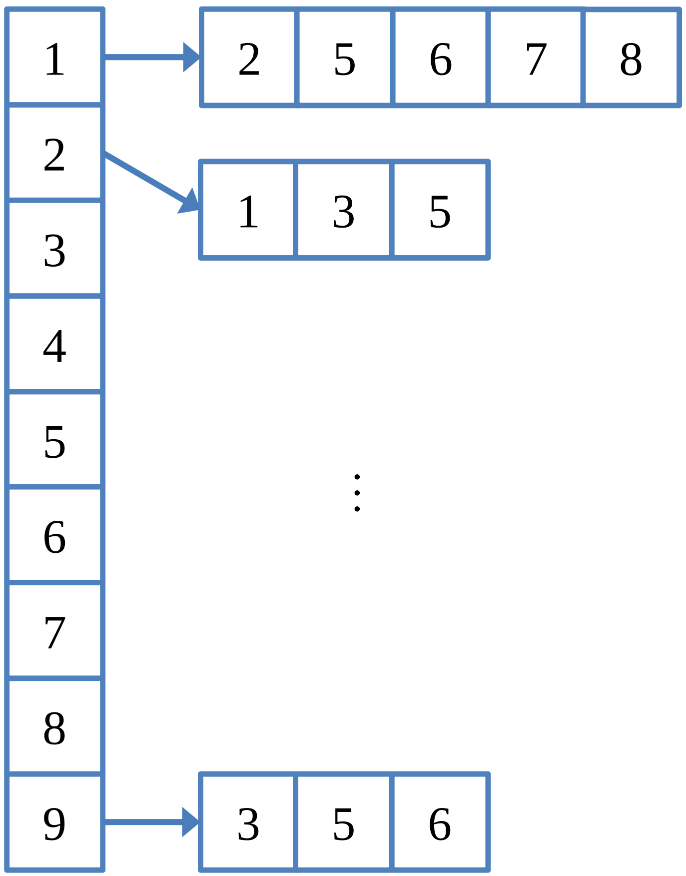
	1	2	3	4	5	6	7	8	9
1									
2	1								
3		1							
4									
5	1	1	1						
6	1								
7	1			1					
8	1						1		
9			1		1	1			

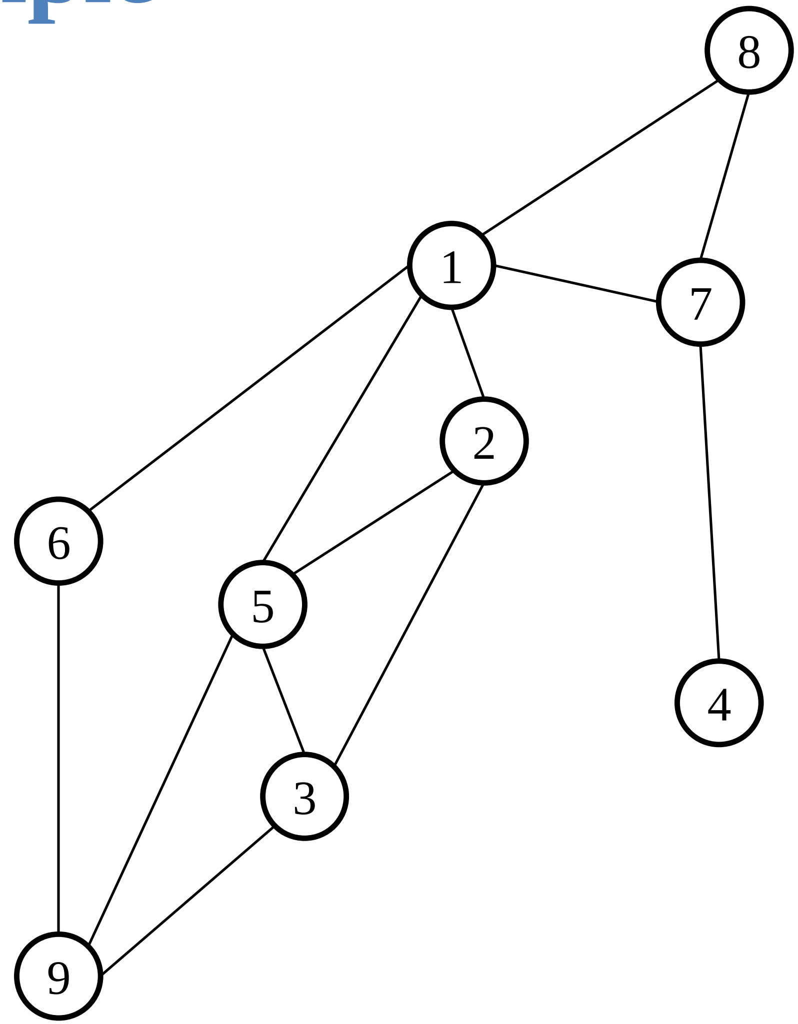


Adjacency list

- An **adjacency list** of a graph may be constructed as follows.
  - Given the number of nodes n, create a **static array** nodes of length n.
  - Each array element is an integer pointer pointing to a **dynamic array** whose length is the node degree.
  - In a node's dynamic array, each element is the index of one of its neighbor.

Adjacency list: an example





Comparisons

- Which one is better?
- Let's compare the amount of **memory space** we need.
- Consider the number of variables we need:
  - Adjacency matrix (full): 9^2 variables = 81variables.
  - Adjacency matrix (half): $(1 + 2 + \cdots + 8)$ variables = 36 variables.
  - Adjacency list: 2×13 variables = 26 variables.
- Is the adjacency list always the winner?
- In general, the number of variables they need are $O(n^2)$, $O(n^2)$, and $O(2m)$.
  - An adjacency list wins if the adjacency matrix is **sparse**.

Case study: makespan minimization

- n jobs should be allocated to m machines. It takes p_j hours to complete job j.
  - p_j is called the **processing time** of job j.
- When a machine is allocated several jobs, its **completion time** is the sum of all processing times of allocated jobs.
- We want to **minimize** the completion time of the machine whose completion time is the **latest**.
  - This is called "**makespan**" in the subject of scheduling.
  - The problem is called "makespan minimization among identical machines."

Heuristics for makespan minimization

- Makespan minimization among identical machines is NP-hard.
- Two well-known heuristic algorithms were proposed by Graham (1966, 1969).
  - Both algorithms are iterative and greedy.
- Algorithm 1:
  - Let the jobs be ordered in any way.
  - In each iteration, assign the next job to the machine that is **currently having the earliest completion time**.
- Algorithm 2 (longest processing time first, **LPT**):
  - Let the jobs be ordered in the **descending order of processing times**.
  - In each iteration, assign the next job to the machine that is currently having the earliest completion time.

Time complexity

- Let's analyze the **worst-case time complexity** of an algorithm:
- The longest processing time first algorithm (LPT):
  - Sort jobs in the descending order of processing times: $\boldsymbol{O(n \log n)}$.
  - In each iteration, assign the next job to the machine that is currently having the earliest completion time.
- Let's analyze the second step.

Time complexity: the second step

- The pseudocode:

Let p be a vector of processing times of the n jobs.
Initialize C_i to 0 for all $i = 1, \dots, m$. // accumulated completion times
for j from 1 to n
    Find i^* such that $C_{i^*} \leq C_i$ for all $i = 1, \dots, m$. // how to implement?
    Assign job j to machine i^*; update C_{i^*} to $C_{i^*} + p_j$.

- Method A: **sort** all completion times to find a smallest one.
  - Sorting: $O(m \log m)$. The whole step: $O(nm \log m)$.
- Method B: do a **linear search** to find a smallest one.
  - Sorting: $O(m)$. The whole step: $O(nm)$.
- May we do better?

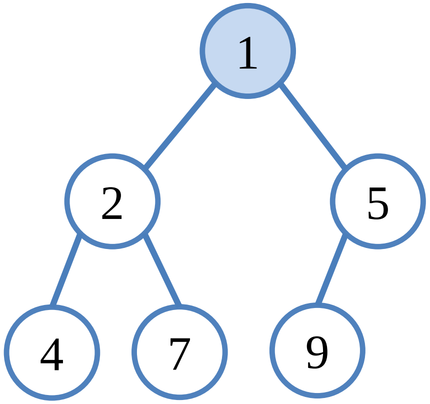
A min heap

- A **min heap** is a complete binary tree where a parent is **no greater than** any of its children.

- For each **subtree**, the root contains the **minimum value** in the subtree.
  - The root of the whole tree contains the minimum value in the tree.
  - There is no restriction on values in different subtrees.

A min heap for completion times

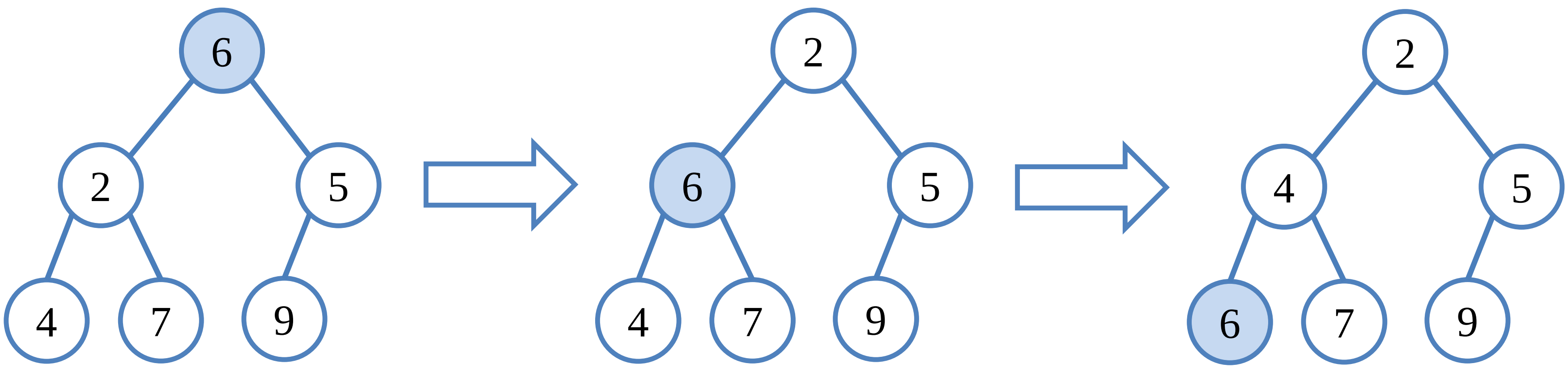
- Let's put the m completion times into a min heap.
- Find the **minimum completion time** is simple: Just look at the root.



- We then update that completion time by **adding a job's processing time** to it.
  - How to **update the tree** to make it still a min heap?

Keeping the tree as a min heap

- Suppose that we add 5 into the minimum completion time. 1 becomes 6.
  - We then exchange 6 with 2, the **smaller** one of its children.
  - We **keep doing so** if needed.



- To do an adjustment, the maximum number of exchange is roughly $\log m$.
- Doing this n **times** takes only $O(n \log m)$.

Time complexity: the second step

Let p be a vector of processing times of the n jobs.
Initialize C_i to 0 for all $i = 1, \dots, m$. // accumulated completion times
for j from 1 to n
    Find i^* such that $C_{i^*} \leq C_i$ for all $i = 1, \dots, m$. // how to implement?
    Assign job j to machine i^*; update C_{i^*} to $C_{i^*} + p_j$.

- One algorithm, three methods:
  - Method A: **sort** to find a smallest one: $O(nm \log m)$.
  - Method B: **linear search** to find a smallest one: $O(nm)$.
  - Method C: use a **min heap** to find a smallest one: $O(n \log m)$.
- A and B are different in **algorithms**; B and C are different in **data structures**.
  - Both B and C use a size-$O(m)$ array. Only the way of storing values differ.

Data structures matter

- To write a good program, data structures matter.
  - As long as you want time or space efficiency.
- In the second half of this course, we introduce fundamental data structures.
  - Lists, stacks, queues (heaps), trees, dictionaries (hash tables), maps/graphs.
  - To let you know when to use which.

DS and advanced programming (AP)

- What is "advanced programming?"
- In our department, currently it means **object-oriented programming** (**OOP**).
  - Key concepts that have been introduced: classes, data hiding, encapsulation, constructor, destructor, friend, copy constructor, etc.
- OOP is a programming paradigm (or philosophy).
  - It is very useful when one wants to build **large-scale** information systems (**with others**); recall your final project in the last semester.
  - It is not about the efficiency of program execution.
  - It is about the efficiency of **system development**.
- OOP also helps us learn data structures.
  - Though in the Department of CSIE they do not do this.

DSAP

- This course is divided into two parts:
  - Advanced programming (OOP): six weeks.
  - Data structures: twelve weeks.
- The first nine weeks are taught by me
- The last nine weeks are taught by professor Chien Chin Chen (陳建錦).

Road map

- What are “data structures” and “advanced programming?”
- **Course policy**

這課很重！

先修課程

- 資管系「程式設計」：
  - 或同等級同類型的課程
  - 總之你要會寫 C 或 C++ 到此刻一般資管系大一同學的程度
- 管院「商管程式設計」或其他用 C#、Java、Python 等語言的課程：
  - 請自行去 http://www.im.ntu.edu.tw/~lckung/courses/PD17fall/ 把課程影片看完，也去 PDOGS 把作業寫一寫
  - 學過一個程式語言，要學第二個就不會很難了
  - 但如果 C/C++ 是繁體中文，其他語言就類似是簡體中文

- 需要略懂 graph theory 和 complexity theory

加選

- 請填修課意向書：
  - https://goo.gl/zS4hGC（課程網站上有，不用抄）
  - 本週四結束前會寄信通知有無獲得修課資格
- 歡迎旁聽
  - 如果想被加入課程網站，請填上方表單

授課方式

- 傳統上：
  - 教師講原理
  - 回家研讀與練習
- 缺點：
  - 教師講課速度無法兼顧全班
  - 三小時很長
  - 教師講授內容沒有保存
  - 研讀時無人可問
  - 教師看不到學生練習狀況

授課方式

- 本學期的前九週（的某幾週）我們將用「**翻轉教室**」方式開課：
  - 我們**不在週一下午講課**，而是提供教師自製的**課程影片**
  - 在那些週一下午，2:20-5:20 教師**帶練習**
  - 請帶充好電的電腦來
  - 可能會提早下課，也可能不會
- 可能有部份週一下午我們會用傳統方式講課
- 本課程沒有實習課或助教 office hour

重要日期

Week	Date	Lecture	Textbook	Note
1	2/26	Overview	DD 1–7, 9, 19, & 22	Kung
2	3/5	Operator overloading	DD 10 & 11	Kung
3	3/12	File I/O, C++ strings, and header files	DD 8 & 18	Kung
4	3/19	Inheritance and polymorphism	DD 12 & 13	Kung
5	3/26	Template and exception handling	DD 14 & 16	Kung
6	4/2	(No class: spring recess)	N/A	Kung
7	4/9	Array- and link-based bags	CH 3 & 4	Kung
8	4/16	Recursion and algorithm efficiency	CH 2, 5, & 10	Kung
9	4/23	***Midterm exam***	N/A	Kung
10	4/30	TBD	TBD	Chen
11	5/7	TBD	TBD	Chen
12	5/14	TBD	TBD	Chen
13	5/21	TBD	TBD	Chen
14	5/28	TBD	TBD	Chen
15	6/4	TBD	TBD	Chen
16	6/11	TBD	TBD	Chen
17	6/18	(No class: Dragon Boat Festival)	N/A	Chen
18	6/25	***Final exam***	N/A	Chen

NTU COOL

- 臺大數位學習中心正在打造新一代的校內學習平臺
  - 舊的：CEIBA
  - 新的：**NTU COOL**
- 課程影片、講義、作業與作業解答會被上傳到 NTU COOL
- 請大家多多利用！
  - 幫忙測試
  - 幫忙回報錯誤
  - 或許有一天幫忙開發、維護

學習活動與成績計算

- Homework：20%
  - 手寫作業
- Programming assignment：20%
  - PDOGS 上繳交
- Final project：20%
  - 分組團隊合作
- Two exams：40%
  - 手寫考卷
- (Bonus) class participation：5%

課程資源

- 暫時：http://www.im.ntu.edu.tw/~lckung/courses/DSAP106-2/
- 看課程影片：NTU COOL
  - 請自行上去看
- 作業繳交與批改：課堂上繳交手寫作業，或 PDOGS 繳交程式作業
  - 請自行用你的 NTU/NTHU/NTUST 信箱註冊 PDOGS
- 線上論壇與**收公告信**：NTU COOL
- 查成績：CEIBA