COM6002 Big Data Management

Data structure

Objectives

- Understand the meaning of Big-O
- Apply various data structures in Python
- Apply parallel processing in Python

- What is not included in this chapter?
  - Most of the theories and concepts behind the data structures
  - We focus on usage (coding!) here

LeetCode

- A free online platform
- Has a great database of questions for coding
  - Some questions are real job interview questions from big companies like Microsoft
  - Most with answers and some with tutorials
- Cover many basic concepts in programming and data structure
- Not just to achieve the functional goal, but ensure your program is fast enough to run within the time limit

Motivating example

- Check experiment 1

- Key observation
  - The same code segment but different data structure
  - Lead to a significant difference in running time!

How do we analyze the performance?

- In computer science, we usually use complexity
- Big O notation
  - Measure the order of the running cost (time / space etc.)

- Example 1:
  - For each student in the class
    - Mark his/her assignment

- The above task has a complexity of $O(n)$ where n is the number of students in the class
- The teacher's workload grows linearly with n

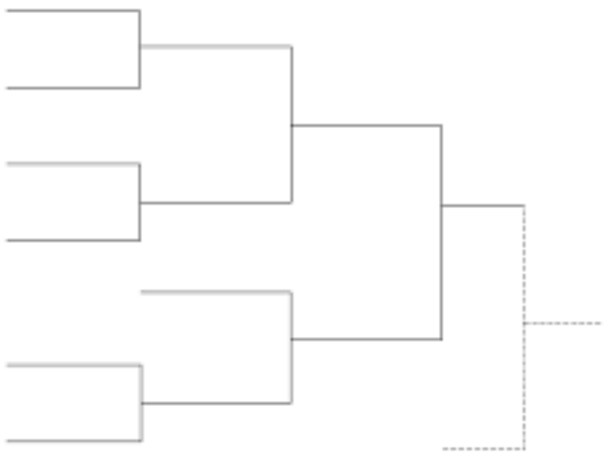
Big O notation examples

- Example 2:
  - For each student in the class
    - The student writes a peer evaluation report to every other student in the class
  - For each peer evaluation report
    - The teacher marks the report

- The above marking task has a complexity of $O(n^2)$ where n is the number of students in the class
- The teacher's workload grows linearly with n^2

Big O notation examples

- Example 3:
  - While there are more than one player left
    - Every two players play a game: the winner stays and the loser leaves
    - ** Many two-player games can happen at the same time; we call it a round



- Number of rounds is O(lg n)
- Number of games is O(n)
- Where n is the number of players

Big O notation examples

- Example 4:
  - Find any ONE student in the class
- Cost is O(1)
- O(1) means it is constant time regardless of how big n is
  - Where n is the number of students in the class

Big O rules

- If the cost is $3n^2 + 4n + 100 \rightarrow O(n^2)$
  - We only care about the highest order term
  - We don't care about the coefficient

- We usually consider the average case or the worst case
  - Example:
    - For each student in the class
      - If the student is Tom, bingo and quit; otherwise, keep looking

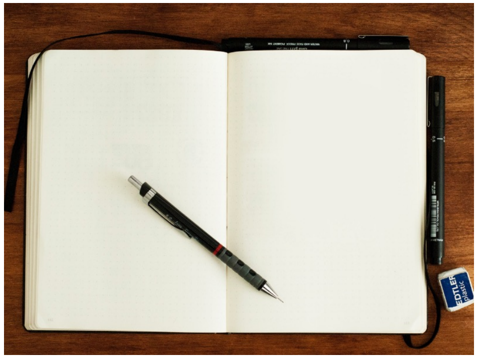
  - Best case: the first student is a hit, O(1)
  - Worst case: O(n)
  - Average: O(n) --- We need to check half of the list on average

Python data structures

- Native
  - List
  - Set
  - Dict
- Commonly used packages
  - Deque
  - Heap
- Check the provided sample codes to see how we can use them in programming

Understanding the data structure

- We use a (physical) notebook to explain the concepts



- Python List
  - The first data element goes to page 1
  - The second data element goes to page 2
  - … and so on

Cost analysis of Python List

- Adding a new element in the middle of the list?
  - O(n)
  - Why?
    - You need to move elements backwards
      - Insert to page 2
        - The old page 2 moves to page 3
        - The old page 3 moves to page 4
        - … and so on

- For all discussions on the costs, n refers to number of elements in the data structure

Cost analysis of Python List

- Getting the i-th element [also called random access]
  - O(1)
  - Easy. Go to page i directly
- Delete the i-th element
  - O(n)
- Adding element at the back (push / append)
  - O(1)
- Deleing the last element (pop)
  - O(1)
- Query: is x in the data structure?
  - O(n)

Understanding set

- Like mathematical set
  - No duplicated element
  - Example
    - If we put [1, 3, 3, 5, 5, 5] into a set, it keeps {1, 3, 5}
  - There is no order between the data
    - When you loop the elements in the set, there is no guaranteed order of data
- Behind the scene, set is implemented as a hash table

Mechanism of set

- Assume the notebook has 100 pages
- We use a hash function to convert an element to [1, 100] or [0, 99]
  - Note: say 0 means page 100
  - In reality, the hash function also needs to consider string / float / boolean data etc. as input
- Example:
  - Hash function h(x) = x % 100
    - % is the modulo operator
      - Do the division and take the remainder
        - Examples: 16%5 = 1, 13%7 = 6, 23%4 = 3
  - Element 33967 % 100 becomes 67
    - Then, we write the data 33967 on page 67

Collision

- What happens if we want to put two elements with the same hash value to the same page
  - Example elements:
    - 33967 and 28767
- There are different collision resolve mechanisms
- In our notebook case, just imagine we write both data elements on the same page
  - So, in the worst case, we write all the data on the same page → searching becomes very difficult / slow: O(n)

Cost analysis

- Adding a new element
  - Average: O(1); Worst: O(n)
- Deleting an element
  - Average: O(1); Worst: O(n)
- Query: is x in the data structure?
  - Average: O(1); Worst: O(n)

- Q: Is set better than list?
  - Think about how you add a new element
  - Don't just look at complexity

Understanding dict

- Similar to set
- Keep key-value pairs
  - In the form of {key: value}
- Example:
  - {1: 30, “a”: 25, 30: “c”}
  - Similar to a set of {1, “a”, 30}; Then we keep additional information about the data element

Cost analysis

- Adding a new key
  - Average: O(1); Worst: O(n)
- Deleting a key
  - Average: O(1); Worst: O(n)
- Query: is x in a key of dict?
  - Average: O(1); Worst: O(n)

Understanding deque

- Data element is written on a random page
- We also write the the page numbers of the previous data element and the next data element
- If there is no sibling page (this is the last element; there is no next page), we write “None” (or null)
- We remember the first page number (and the last page for a doubly linked list)

- Behind the scene, deque is a doubly linked list

Deque examples

- At the beginning, there is no data element

First Page: None
Last Page: None

- Say, an element 13 is added
  - We find an empty page (say, 52) and write the data element on the page

First Page: 52
Last Page: 52

Data: 13
Prev: None
Next: None

Page 52

Deque examples

- Say, an element 26 is added
  - We find an empty page (say, 78) and write the data element on the page

First Page: 52 Last Page: 78	Data: 13 Prev: None Next: 78	Data: 26 Prev: 52 Next: None
	Page 52	Page 78

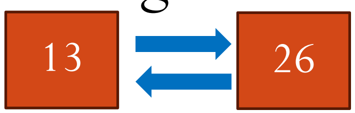
- Adding 30 on page 11

First Page: 52 Last Page: 11	Data: 30 Prev: 78 Next: None	Data: 13 Prev: None Next: 78	Data: 26 Prev: 52 Next: 11
	Page 11	Page 52	Page 78

Simplified visual for linked list

- Adding 13

- Adding 26



- Adding 30

- Pros:
  - Easy handling of adding / deleting data in the middle
    - Does not require moving data forward / backward
- Cons:
  - Not easy for random access

Cost analysis

- Getting the i-th element [also called random access]
  - O(n)
- Adding a new element (assume we know where to add)
  - O(1)
- Deleting an element (assume we know where to delete)
  - O(1)
- Query: is x in the data structure?
  - O(n)

Understanding Heap

- Also called priority queue
  - Very efficient in finding the smallest (or largest) element in a queue
  - Allow adding new data to the queue

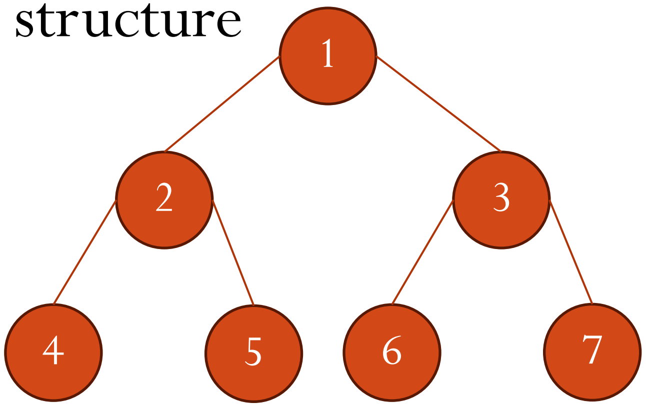
- Consider this problem:
  - Find the smallest number --- O(n)
  - Find again the 2nd smallest number in the remaining data --- O(n)
  - Find again the 3rd smallest number --- O(n)
  - …
  - Outputting the data from smallest to largest takes $O(n^2)$

Can we reuse some previous information to reduce the cost?

Heap structure

- View the data list as a tree-like structure

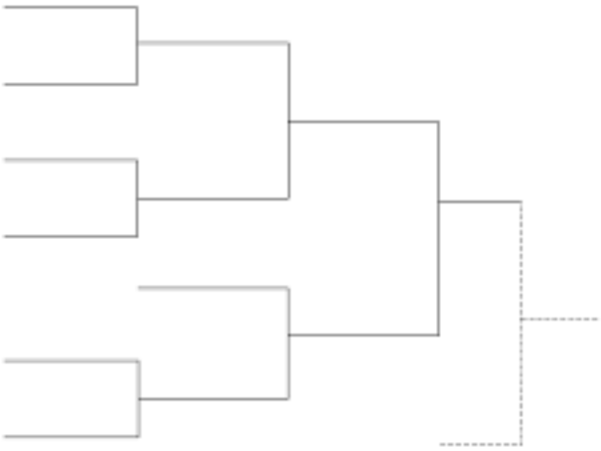
- Converting from a list to a heap structure is called “heapify”



1	2	3	4	5	6	7

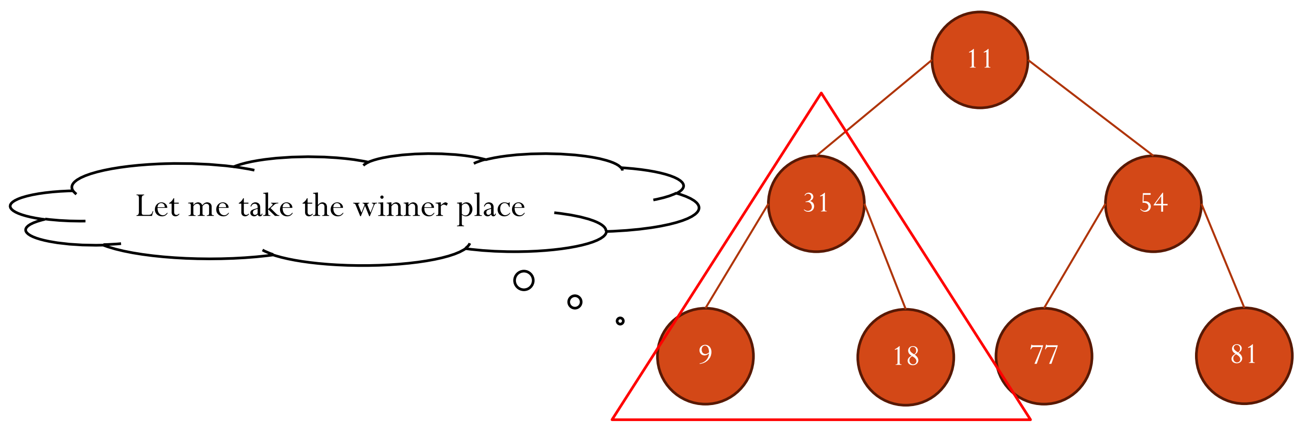
How do we find the smallest value?

- Like a tournament
  - Smallest of the smallest



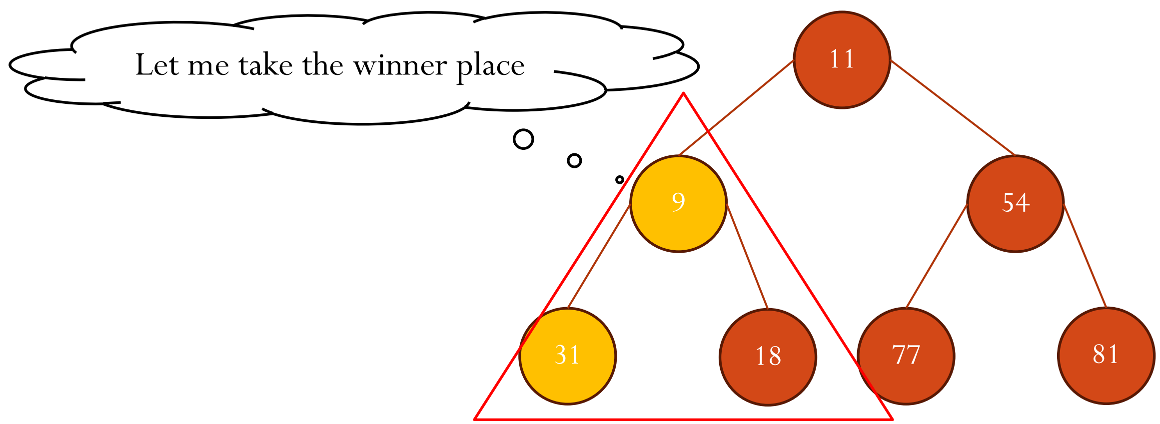
Finding the smallest in a group

- 9 is the smallest



11	31	54	9	18	77	81

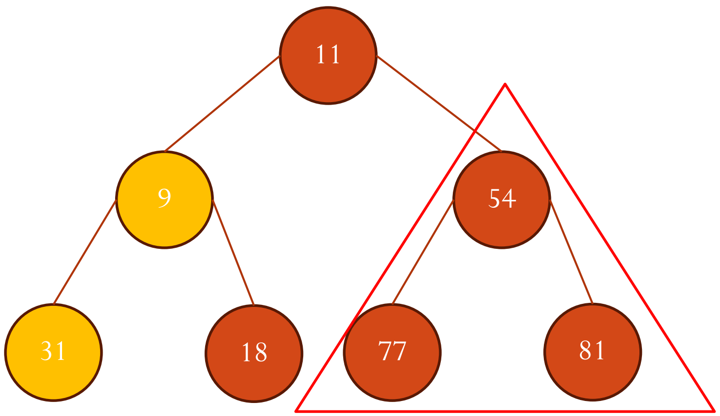
Swapping the positions



11	9	54	31	18	77	81

Check the other group

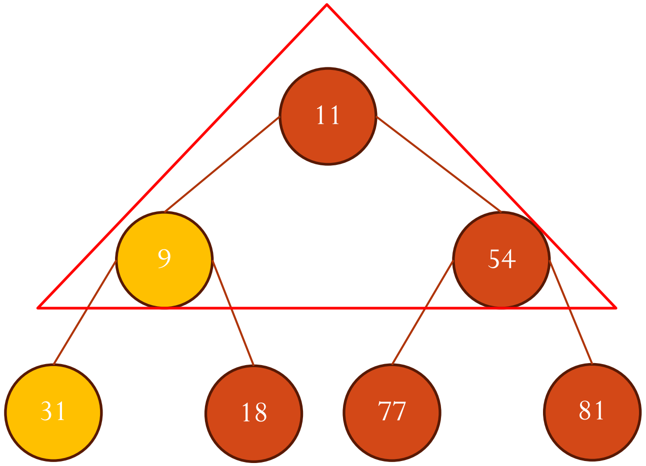
- 54 is the smallest. Keep the positions of all



11	9	54	31	18	77	81

Continue

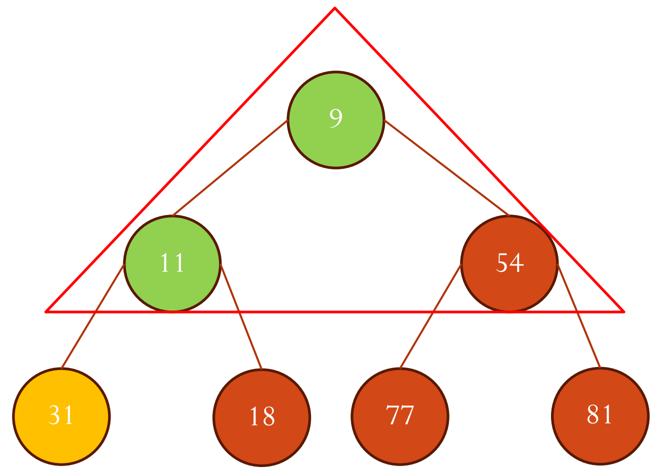
- 9 is the smallest. Swap the positions



11	9	54	31	18	77	81

Result

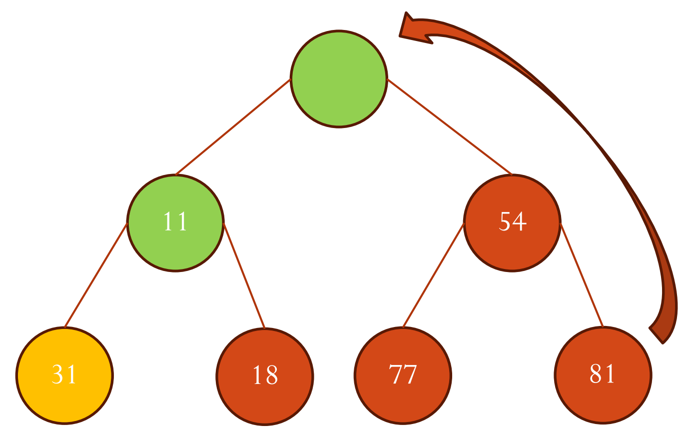
- The root node (the node at the top level) is guaranteed to be the smallest
  - Q: Can you prove it?



9	11	54	31	18	77	81

Heappop

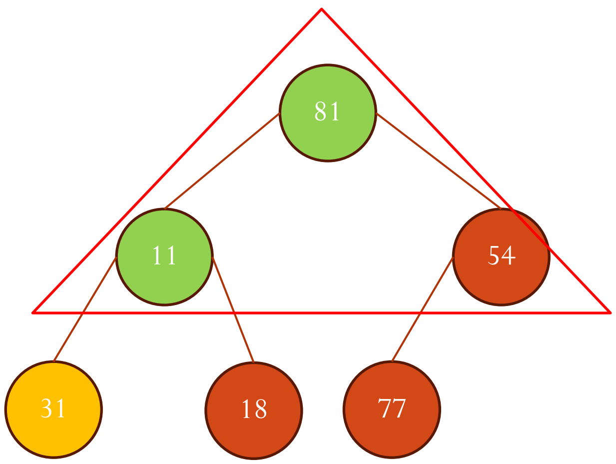
- Heappop:
  - Remove the smallest element from the queue
- As a list, it is efficient to remove the last element
- We move the last the element to the first



	11	54	31	18	77	81

Heappop maintenance

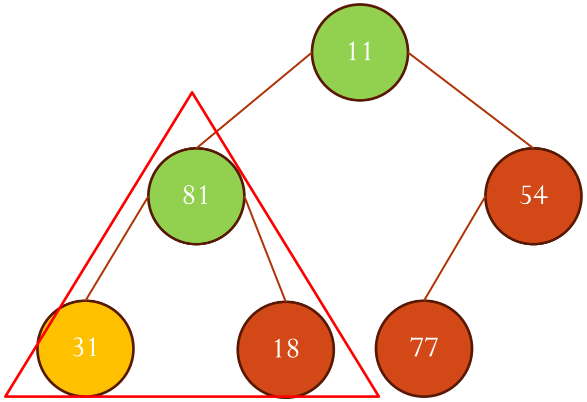
- The heap structure has a property that the smallest node should be placed as the parent
- 11 should be swapped with 81



81	**11**	**54**	**31**	**18**	**77**

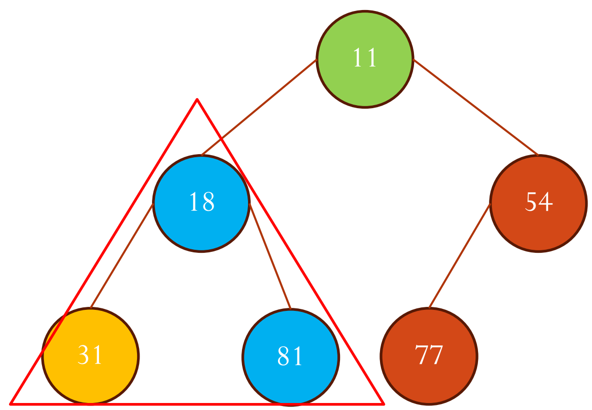
Heappop maintenance

- The same applies to the left side child tree
  - 18 swaps with 81
- We do not need to examine the right child
  - The swap at the higher level only affects the left child



11	81	54	31	18	77

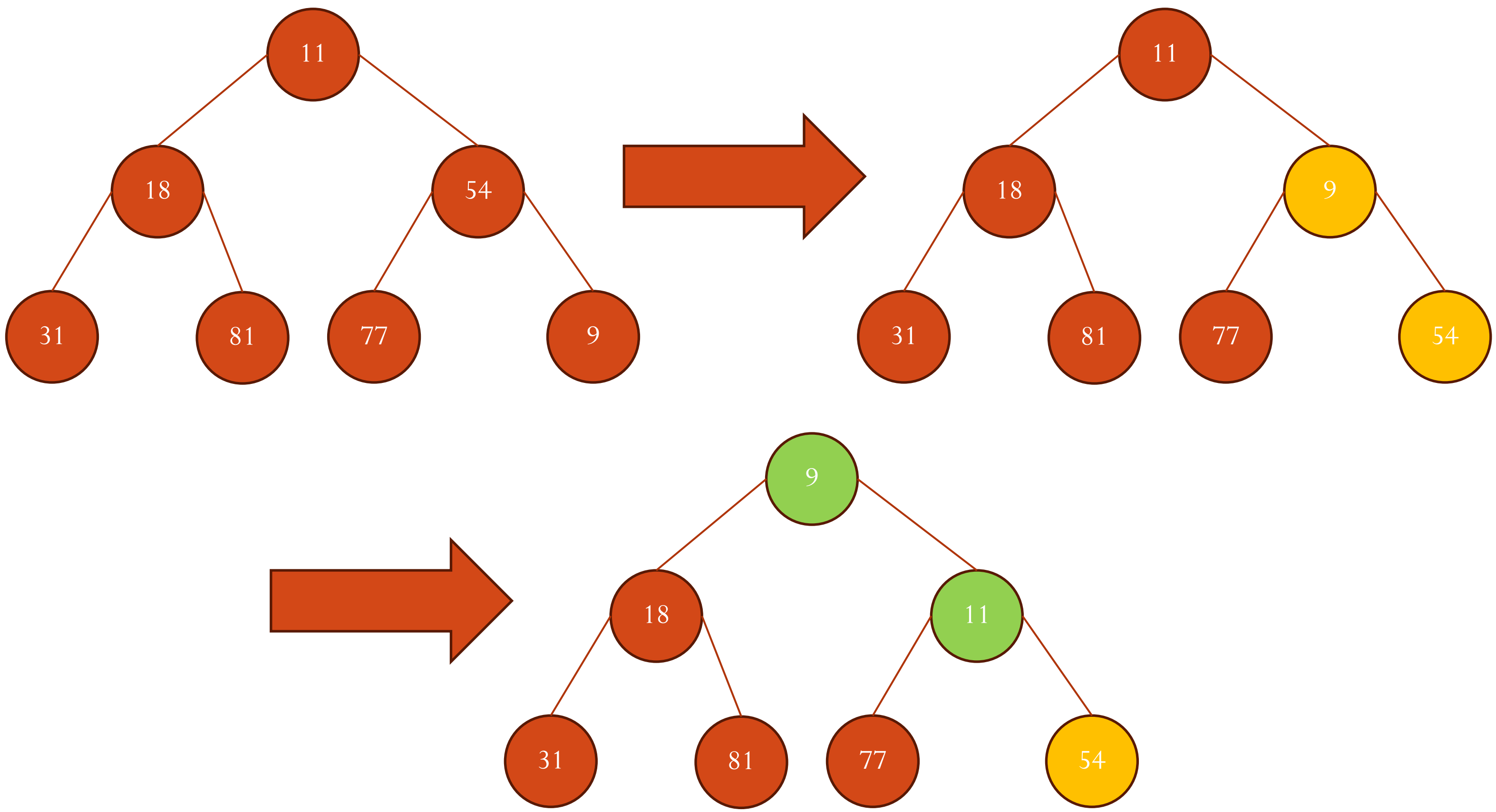
Heappop result



11	18	54	31	81	77

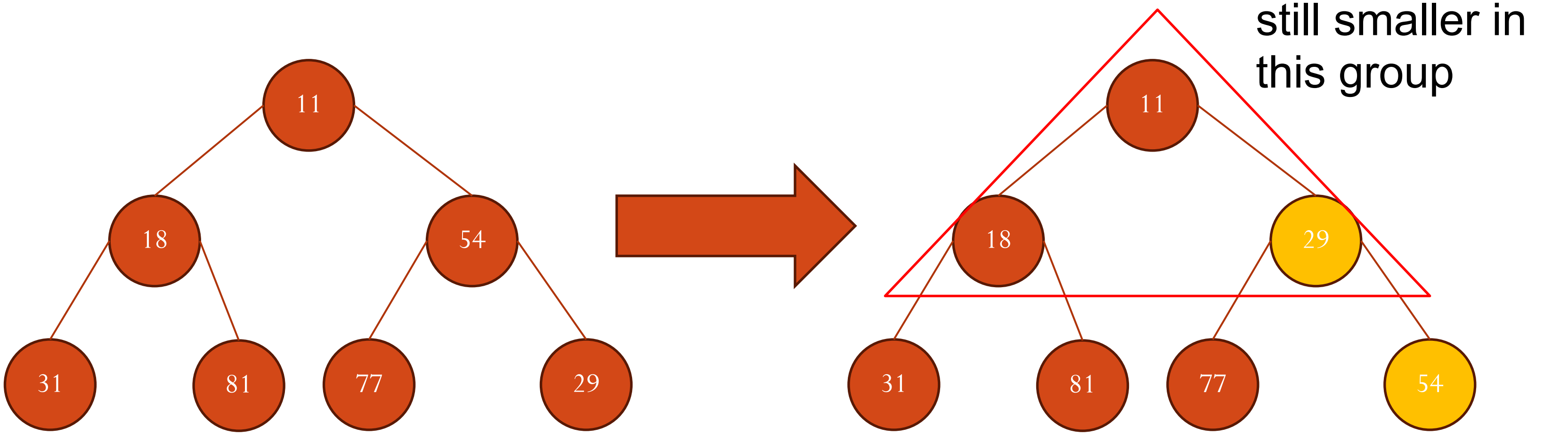
Heappush:
Adding new element to the queue

- Say adding 9
  - Update from the bottom to the top



Heappush: Another example

- Say adding 29
  - Update from the bottom to the top



Cost analysis

- Heapify
  - O(n)
- Heappush
  - O(lg(n))
- Heappop
  - O(lg(n))

More information [self-study]

- More about the complexity information of Python data structures
  - https://wiki.python.org/moin/TimeComplexity

- More information about Heap
  - https://en.wikipedia.org/wiki/Binary_heap

Example: Keys and Rooms

- https://leetcode.com/problems/keys-and-rooms/

- Your [non-coding] task:
  - Try to describe a solution to your classmate
    - Make sure it can be easily understood
  - Listen to your classmate's solution
    - Try to understand what it is doing
    - Reason about the correctness of the solution
      - Imagine someone is stupid and will just follow your instructions to work. Will he/she get the correct answer?
      - Computer is stupid and will just follow your instructions

Step #1 – the basic visiting logic

1. Visit room 0
2. Add every room it can open to a queue / list (or whatever structure that is appropriate) "q"
3. Pick any room from "q"
4. Go back to step 2 until "q" is empty

Step #2 – count how many rooms are visited

1. Visit room 0
2. Keep a list / queue (or whatever structure that is appropriate) "visited" about visited rooms
3. Add room 0 to "visited"
4. Add every room it can open to a queue / list (or whatever structure that is appropriate) "q"
5. Add all these rooms to "visited"
6. Pick any room from "q"
7. Go back to step 4 until "q" is empty

Step #3 – avoid infinite loop

1. Visit room 0
2. Keep a list / queue (or whatever structure that is appropriate) "visited" about visited rooms
3. Add room 0 to "visited"
4. Add every room it can open and is not in "visited" to a queue / list (or whatever structure that is appropriate) "q"
5. Add all these rooms to "visited"
6. Pick any room from "q"
7. Go back to step 4 until "q" is empty

Step #4 – determine the most efficient structure

1. Visit room 0
2. Keep a set "visited" about visited rooms
3. Add room 0 to "visited"
4. Add every room it can open and is not in "visited" to a list "q"
5. Add all these rooms to "visited"
6. Pick the last room from "q" (if any room can do, we simply pick the last room for efficiency purpose)
7. Go back to step 4 until "q" is empty

Example:

- https://leetcode.com/problems/path-with-maximum-probability/
  - This example is more advanced
- Q: What is your approach to solve the problem?

Sub-problem: edge access efficiency

- The edges are given as a list

- How can we determine if there is a path from x to y?
  - Complexity: O(e)
    - Where e is number of edges. Number of edges is of $O(n^2)$ where n is number of nodes

- Sample method
  - Convert it to dict of dict

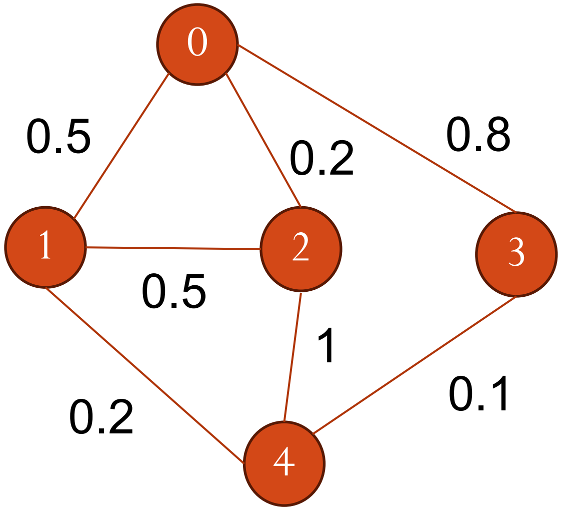
```
path: dict[int, dict[int, float]] = {}
```

  - path[u][v] = probability of the edge from u to v
- Q: Complexity to determine if there is a path from x to y?

Solution logic

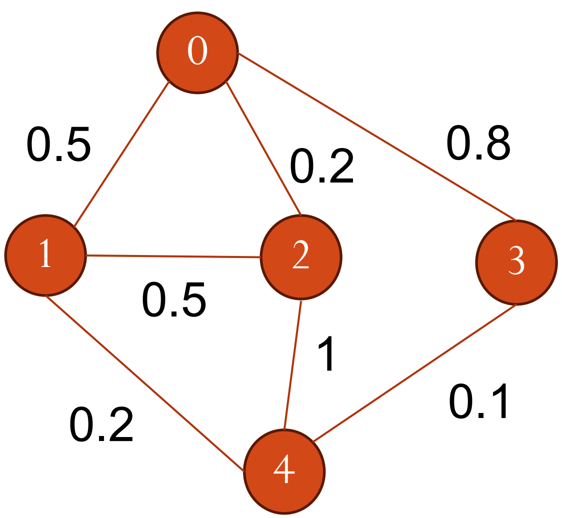
- Idea:
  - Keep exploring the highest probability path

- Example
  - From node 0 to node 4. What is the best path?



Example run

- At the beginning, we start at node 0 with probability of 1



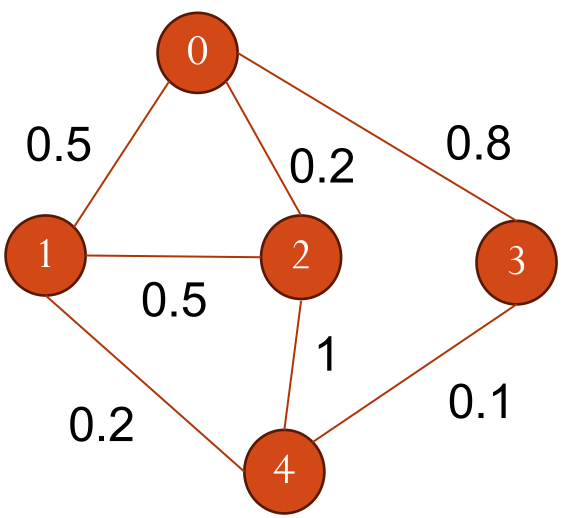
Queue

Node	Probability
0	1

Visited

Example run

- With no other choice in the queue, we check where we can go from node 0



Queue

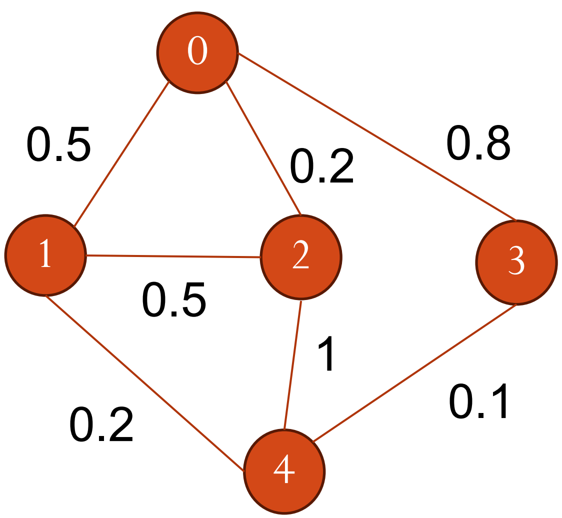
Node	Probability
1	0.5
2	0.2
3	0.8

Visited

0

Example run

- Next, we pick node 3 as it has the highest probability
  - Node 0 should NOT be added to the queue
    - Avoid infinite loop. Handled by using the visited set



Queue

Node	Probability
1	0.5
2	0.2
4	0.08

Visited

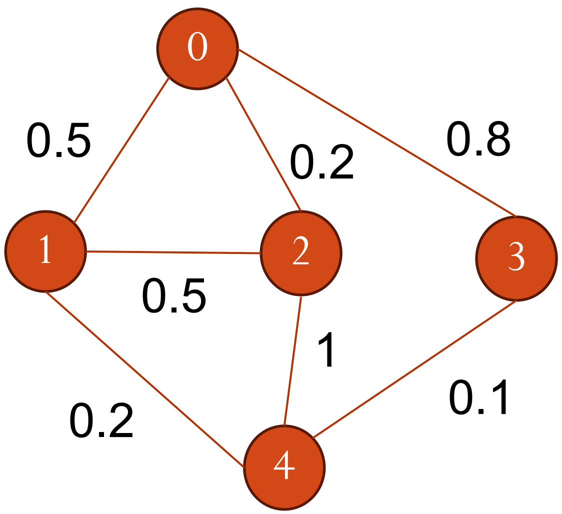
0, 3

Target is in the queue. Do we want to stop here?

Maybe there is another path, begin with lower probability but ends with a higher probability

Example run

- Next, we pick node 1



Queue

Node	Probability
2	0.2
4	0.08
2	0.25
4	0.1

Visited

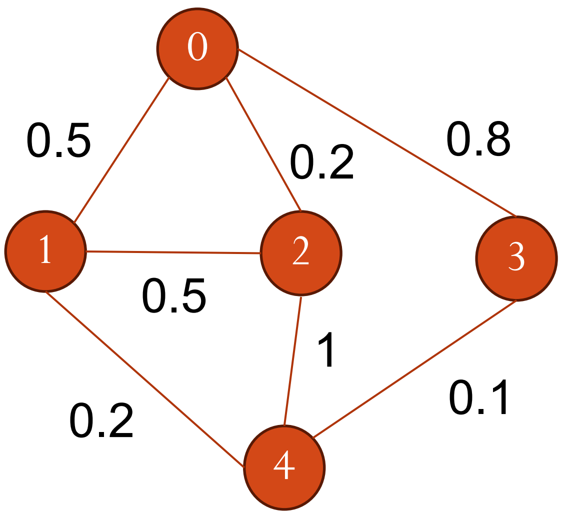
0, 3, 1

Two copies of node 4 in the queue. Reached by different paths. We only want the highest probability one.

Ignore the second copy and so on by using the visited

Example run

- Next, we pick node 2 with probability of 0.25



Queue

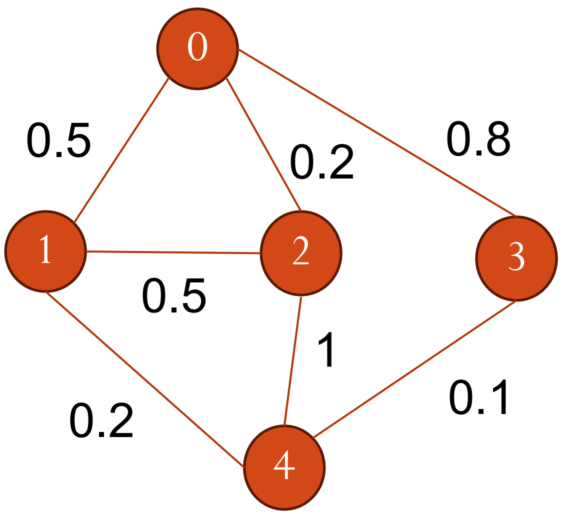
Node	Probability
2	0.2
4	0.08
4	0.1
4	0.25

Visited

0, 3, 1

Example run

- Next, we pick node 4 with probability of 0.25
  - This is the end. We reach the destination



Queue

Node	Probability
2	0.2
4	0.08
4	0.1

Visited

0, 3, 1

Q: There won't be another path reaching node 4 with a probability > 0.25. Why?

Q: What data structure should we use for the queue?

Converting into code-like steps

- Pseudocode
  - Put (start, 1) to the queue
  - While queue is not empty
    - Get the record (u, p) with the highest probability
      - Recall: u is the node, p is the probability
    - If u is the end
      - Bingo. The answer is p
    - If u is not visited
      - Add u to visited
      - For all reachable nodes from u that are not visited
        - Add them to the queue
  - End of while loop
  - Still not finding the end? It means it is not reachable. Return 0

Check the sample codes to see how this is converted into Python

Multiprocessing

- Run in one process vs run in multiple processes
  - If you write a normal Python program, it is only using one core of your CPU
- Theory and concepts behind multiprocessing will not be covered in this module

12th Gen Intel® Core™ Desktop Processors Comparison

	Intel® Core™ i9	Intel® Core™ i7	Intel® Core™ i5	Intel® Core™ i3
Max Turbo Frequency [GHz]	Up to 5.2	Up to 5.0	Up to 4.9	Up to 4.4
Intel® Turbo Boost Max Technology 3.0 Frequency [GHz][5]	Up to 5.2	Up to 5.0	n/a	n/a
Performance-core Max Turbo Frequency [GHz][6]	Up to 5.1	Up to 4.9	Up to 4.9	Up to 4.4
Efficient-core Max Turbo Frequency [GHz][6]	Up to 3.9	Up to 3.8	Up to 3.6	n/a
Performance-core Base Frequency [GHz]	Up to 3.2	Up to 3.6	Up to 3.7	Up to 3.5
Efficient-core Base Frequency [GHz]	Up to 2.4	Up to 2.7	Up to 2.8	n/a
Processor Cores (P-cores + E-cores)[7]	16 (8P + 8E)	12 (8P + 4E)	10 (6P + 4E) or 6 (6P + 0E)[7]	4 (4P + 0E)

GPU vs CPU

- GPU is also processing unit like CPU
- GPU has much more cores than CPU, but the computational ability of each core is weaker than a core of CPU
- GPU is good for tasks that can be run in parallel
  - Like matrix operations (which is heavily used in neural network)
    - Do you see why we usually use GPU in AI?

TPU? LPU?

- TPU – Tensor Processing Units
  - Customized to support computations around neural network (tensors)
    - Tensors: basic unit of data inside GPU (e.g., numbers / vectors / matrices)
  - Cons: not for other GPU tasks, e.g., video / image rendering
- LPU – Language Processing Units
  - Customized to support LLM operations

Sample codes for multiprocessing

A multiprocessing executor in Python

```
executor = ProcessPoolExecutor()
pool = [executor.submit(a_long_task, id, id) for id in range(10)]

for task in as_completed(pool):
      print(task.result())
```

Submit the tasks to the executor

Get back the results

Note: the results may not be in the same order as they are submitted

Data structure in pandas / numpy

- They are designed for supporting machine learning and data science operations, e.g., many operations are done on tabular data

- Check experiment 3

Summary

- Remember, no single data structure is the best in all scenarios
- Analyze your use case and pick the best one
  - The performance difference can be huge, especially when the data is huge
- More advanced topics for self-study
  - RAM
  - SSD
  - File system