Applied Programming I

Programming computers in a nutshell.

Introduction to algorithms and data structures

Some minimal basics

Today's objectives

Break:
Midterm evaluation

- Basic understanding of
  - Algorithms and
  - Data structures.
- What are they, why are they important, what to consider?

Algorithm

- An algorithm is well-defined step-by-step procedure to solve a specific (computational) problem. The algorithm typically gets input and produces output.
- A cooking recipe is a good illustration of an algorithm. It takes the ingredients as input, and by a step-by-step procedure, the dish as output is produces.

Sorting

The **sorting problem** is quite popular to illustrate algorithms. The sorting problem is defined by:

- Input: a sequence of n numbers: $(a_1, a_2, a_3, \ldots, a_n)$
- Output: a permutation of the input: $(a'_1, a'_2, a'_3, \ldots, a'_n)$, so that $a'_1 \leq a'_2 \leq a'_3 \leq \ldots \leq a'_n$

Insertion sort

```
For  j = 2 to A.length:
    key = A[j]
    i = j – 1
    while i>0 and A[i] > key:
        A[i+1] = A[i]
        i = i – 1
    A[i+1] = key
```

Index / step	1	2	3	4	5	6
a	14	6	11	17	3	8
b	6	14	11	17	3	8
c	6	11	14	17	3	8
d	6	11	14	17	3	8
e	3	6	11	14	17	8
F	3	6	8	11	14	17

Complexity

- Complexity of algorithms refers to the amount of required resources regarding time and memory/space.
- **Time complexity** is a measure of time (or certain kind of operations) an algorithms requires to solve a problem as a function of the input size.
- **Space complexity** is a measure of memory as a function of the algorithm's input size.
- The complexity considers worst-case scenarios and is given as an upper bound, usually in **Big O notation**.

Big O notation: example for time complexity

The Big O notion describes how the running time of the algorithm increases as follows with the size n of its input:

- $O(n)$: linearly
- $O(n^2)$: quadratically
- $O(\log n)$: logarithmically
- $O(e^n)$: exponentially

Insertion sort: analysis

```
For j = 2 to A.length:        (1)
  key = A[j]                  (2)
  i = j–1                     (3)
  while i>0 and A[i]>key:     (4)
    A[i+1] = A[i]             (5)
    i = i–1                   (6)
  A[i+1] = key                (7)
```

How often? …-times:

(1) n

(2) n-1

(3) n-1

(4) depends on A[i] > key

(5) One less than (4)

(6) One less than (4)

(7) n-1

Insertion sort: analysis

```
For j = 2 to A.length:        (1)
  key = A[j]                  (2)
  i = j–1                     (3)
  while i>0 and A[i]>key:     (4)
    A[i+1] = A[i]             (5)
    i = i–1                   (6)
  A[i+1] = key                (7)
```

Best case: sorted array as input:

(4) only n-1 times

Worst case: reverse sorted array:

(4) $n(n+1)/2 – 1$

(5) $n(n-1)/2$

(6) $n(n-1)/2$

-> Insertion sort is $O(n^2)$

The code is in the github repository… as usually.

Your task:
Implement insertion sort

In (your project) groups, write a program in Python that implements insertion sort. Use the program compare-times-sort.py as starting point. Optional: Translate the program in another language (Javascript, C, C#) using tools such as Copilot and try to understand

Time: 20 minutes.

Divide and conquer

- The **divide-and-conque**r approach is powerful algorithmic technic to tackle problems. The problem is broke down into smaller, but easier, problems.
- The problem is **divided** into smaller subproblems thar are similar to the original problem but easier to solve.
- The subproblems are solved (recursively) - they are **conquered**.
- Finally, the solved solutions of the subproblem are **combined** and form the solution to the original problem.

Binary search

- Task: search an element x in a sorted array .
- Divide: Split the array into two halves with its middle element m.
- Conquer: Compare x with the middle element m,
  - if x == m: element found, done
  - if x< m: repeat process with left/lower half of current array
  - If x> m: repeat process with right/higher half of current array
  - if half of current array is empty, element not found

Binary search: illustration – search 9

1	2	4	7	8	9	10	13	15	17	19	23	24	27	30
1	2	4	5	6	9	10	9 <							
			9 >	6	9	10								
					==9									

Binary search in Python, AI generated code

```
def binary_search_iterative(arr, x):
  low = 0
  high = len(arr) - 1
  while low <= high:
    mid = (low + high) // 2        # / 2 and round down
    if arr[mid] < x:
      low = mid + 1
    elif arr[mid] > x:
      high = mid - 1
    else:
      return mid
    return -1              # not found
```

Merge sort

- **Divide** the array in two halves if it has more than one element.
- Recursively sort each half with merge sort (**conquer**).
- Merge (**combine**) the two sorted halves by comparing and arranging their elements.

Merge sort illustration

18	8	25	17	5	19	22	12
Merge		Merge		Merge		Merge	
8	18	17	25	5	19	12	22
Merge				Merge			
8	17	18	25	5	12	19	22
Merge							
5	8	12	17	18	19	22	25

Merge sort: pseudo code 1/2

```
function mergeSort(array):
  if length of array <= 1:
    return array
  mid = length of array / 2
  leftHalf = mergeSort(array[0:mid])
  rightHalf = mergeSort(array[mid:length of array])
  return merge(leftHalf, rightHalf)
```

Merge sort: pseudo code 2/2

```
function merge(left, right):
  result = []
  i = 0
  j = 0
  while (i < length of left
        and j < length of right):
    if left[i] <= right[j]:
      append left[i] to result
      i = i + 1
    else:
      append right[j] to result
      j = j + 1

  while (i < length of left):
    append left[i] to result
    i = i + 1

  while (j < length of right):
    append right[j] to result
    j = j + 1

  return result
```

Merge sort and complexity

- It can be shown that merge sort has a time complexity of O(n log n) for n elements and, thus, faster as insertion sort.

Your task:
Implement merge sort and compare its performance with insertion sort

In (your project) groups, write a program that merge sort in Python. Optional: Translate this program into JavaScript.

Time: 30 minutes.

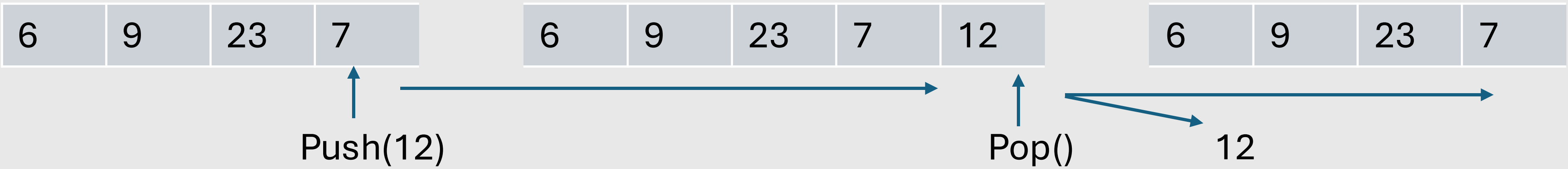
Break and midterm evaluation

40 minutes

Data structures

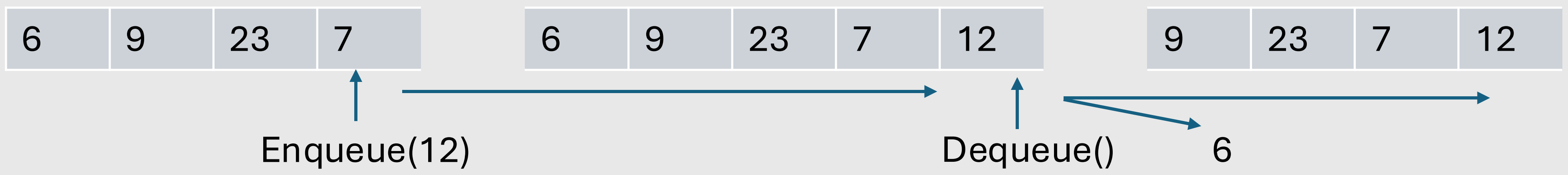
- Data structures (as the already covered arrays) are means of organizing and storing data. Using algorithms, it can be beneficial to use or develop specific data structures to solve a problem.
- Most programming languages come along with elementary data structures.

Elementary data structure: Stack



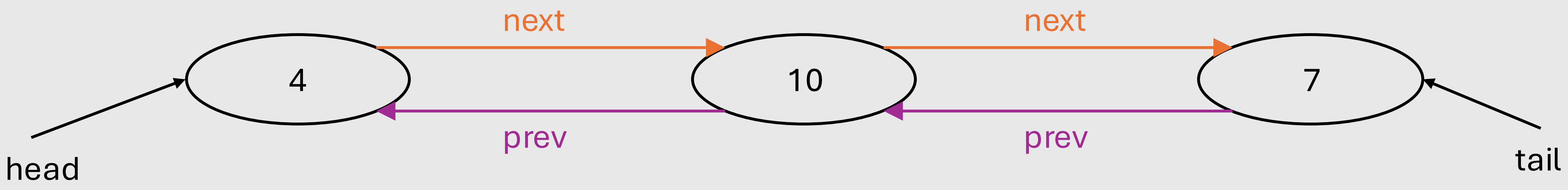
- LIFO: Last-in-first-out data structure
- A stack provides three operations:
  - Empty: returns true if stack is empty else false
  - Push element: insert element onto the top of the stack
  - Pop element: return and delete element off the top of the stack

Elementary data structure: Queue



- FIFO: First-in-first-out data structure
- A queue supports the operations
  - Length: return the size of the queue
  - Enqueue: put an element to the end of the queue
  - Dequeue: get and remove the last element of the queue

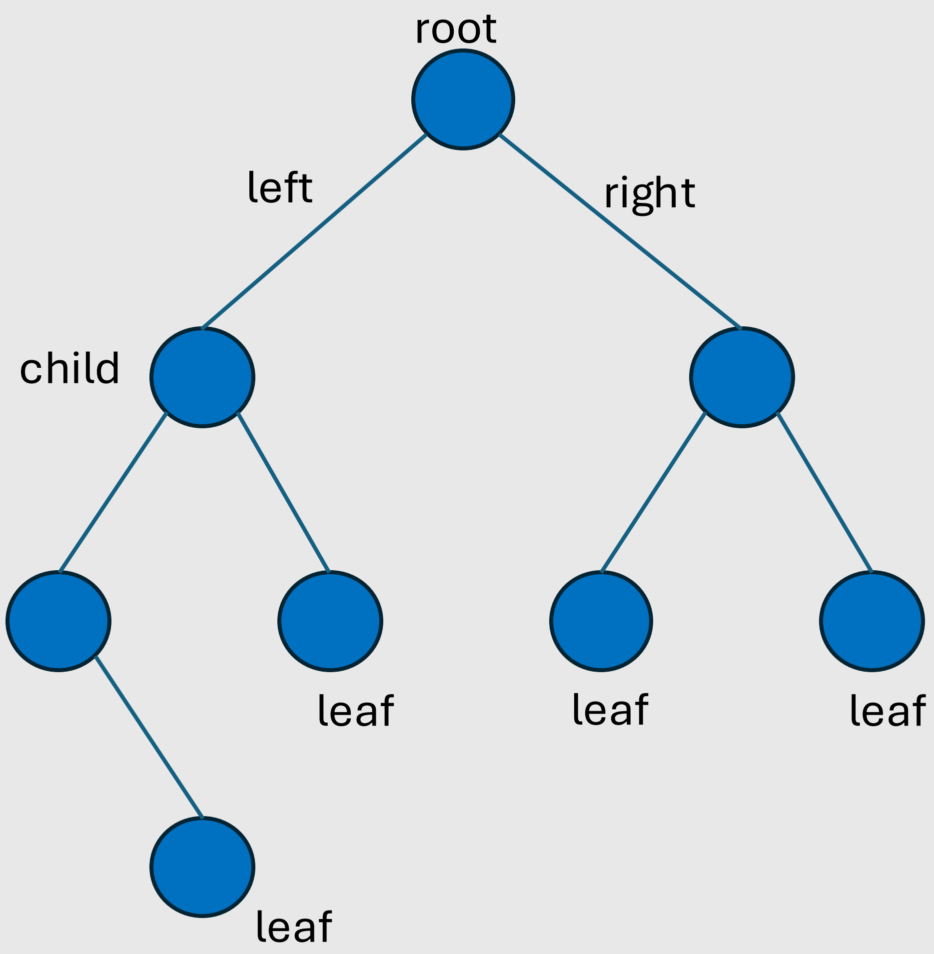
Elementary data structure: List



- A double-linked list is a data structure with elements that contain a key (content) and pointers to the previous and next elements.
- Compared with an array, an insertion/deletion at any position does not require a lot copying of data.

Elementary data structure: Tree

- Trees are hierarchical data structures. They extend the concept of linked elements (as in lists).
- Nodes are the elements of trees with keys (data) and information (usually pointers) to parent and children.
- Binary trees only have two nodes that can have left and right children.
- Applications are, e.g., decision trees.

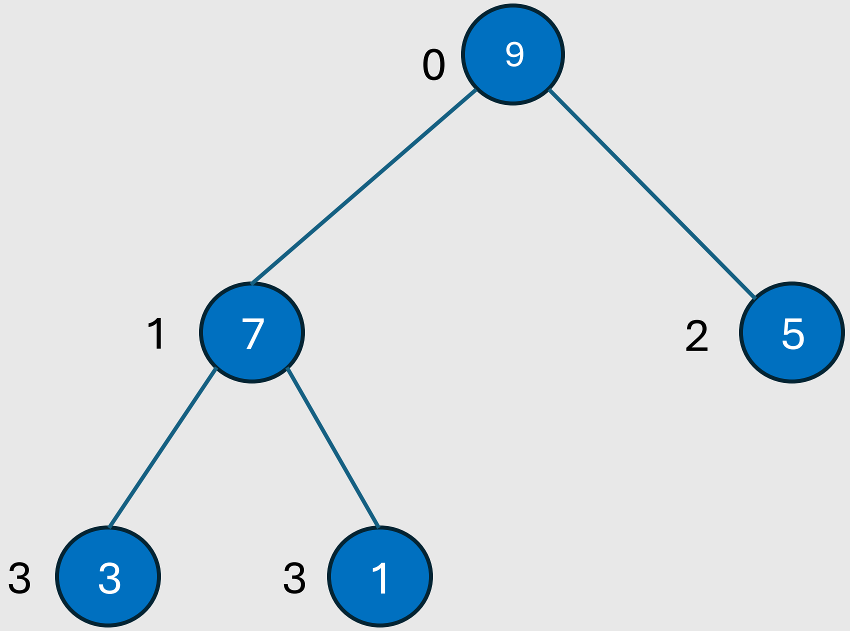


Elementary data structure: Heap

- A binary heap is a data structure that represents a binary tree for numbers. A heap hast to satisfy a heap property.
- The heap property can be max-heap or min-heap.
  - Max-heap: the value of a given node is less or equal to its parent node. The maximum value is in the root.
  - Min-heap: analogous to max-heap but with greater or equal
- A heap can be implemented using an array.

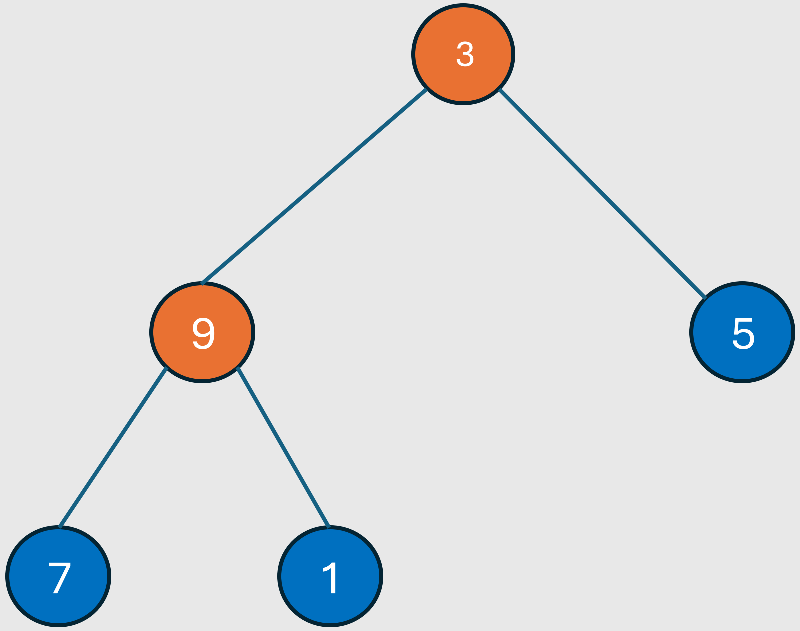
Elementary data structure: heap as array

Indices:				
[0]	[1]	[2]	[3]	[4]
Data:				
9	5	3	7	1



Elementary data structure: heapify an array

Array to heapify:				
3	9	5	7	1
Transform to hold max-heap property: swap				
9	3	5	7	1

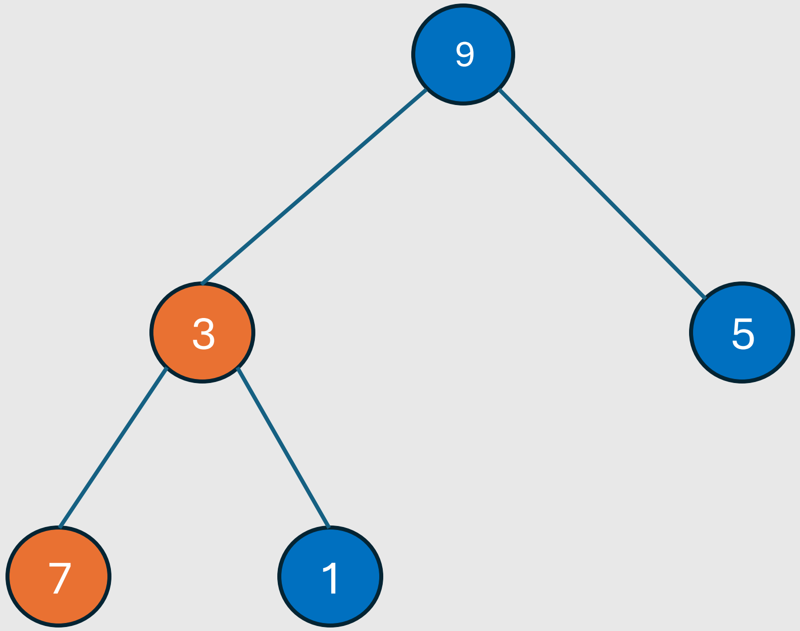


Build a max-heap in Python

```
def heapify(arr, n, i):
  max = i # largest as root
  lf = 2 * i + 1 # left = 2*i + 1
  rt = 2 * i + 2 # right = 2*i + 2

  # left child? greater than root?
  if lf < n and arr[i] < arr[lf]:
    max = lf

  # right child? greater than root?
  if rt < n and arr[max] < arr[rt]:
    max = rt
  # Change root, if needed
  if max != i:
    arr[i], arr[max] = arr[max],arr[i]

  # Heapify the root.
  heapify(arr, n, max)


def max_heap(arr):
  n = len(arr)

  # Build a maxheap.
  for i in range(n // 2 - 1, -1, -1):
    heapify(arr, n, i)
```

Elementary data structure: heapify an array

Array to heapify:				
3	9	5	7	1
Transform to hold max-heap property: swap				
9	3	5	7	1
Transform to hold max-heap property: swap				
9	5	3	7	1

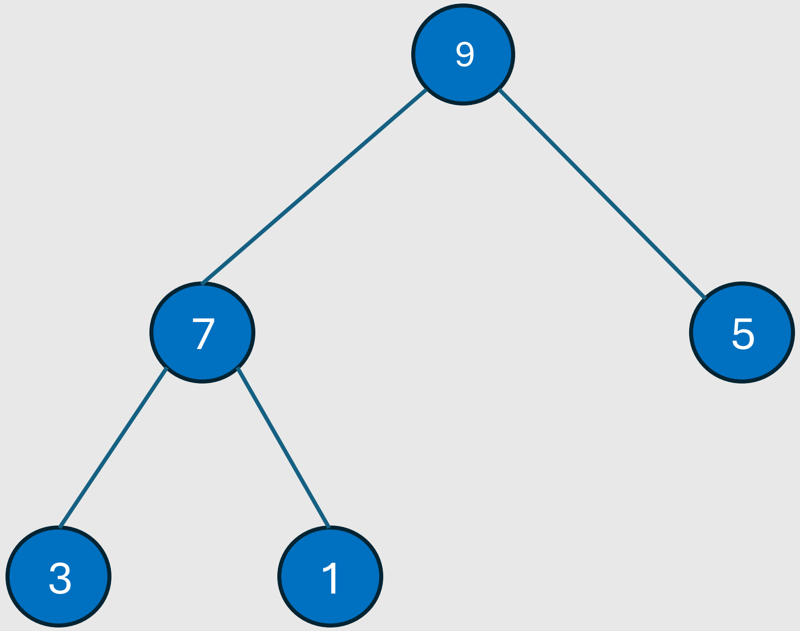


Build a max-heap in Python

```
def heapify(arr, n, i):
  max = i # largest as root
  lf = 2 * i + 1 # left = 2*i + 1
  rt = 2 * i + 2 # right = 2*i + 2

  # left child? greater than root?
  if lf < n and arr[i] < arr[lf]:
    max = lf

  # right child? greater than root?
  if rt < n and arr[max] < arr[rt]:
    max = rt
  # Change root, if needed
    if max != i:
      arr[i], arr[max] = arr[max],arr[i]

  # Heapify the root.
  heapify(arr, n, max)


def max_heap(arr):
  n = len(arr)

  # Build a maxheap.
  for i in range(n // 2 - 1, -1, -1):
    heapify(arr, n, i)
```

Elementary data structure: heapify an array

Array to heapify:				
3	9	5	7	1
Transform to hold max-heap property: swap				
9	3	5	7	1
Transform to hold max-heap property: swap				
9	5	3	7	1



Heap sort

1. Build a max-heap
2. Until heap is empty,
   a) Take largest element from heap and put into result array
   b) Heapify remaining heap

The result array can be build using the array of the heap by filling it from the back.

Advice: Research what this line does first:

```
for i in range(n // 2 - 1, -1, -1):
```

Your task:
Implement the remaining step of heap sort.

In (your project) groups, write a Python program that implements the remaining step of heap sort.

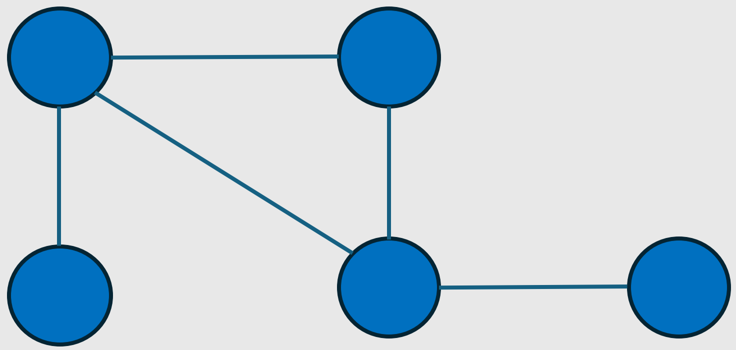
Time: 20 minutes.

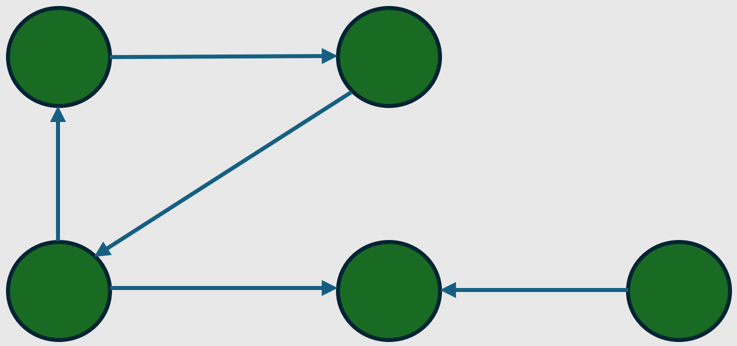
Heap sort and complexity

- Heap sort has a time complexity of ???
- What do you think?

Elementary data structure: Graph

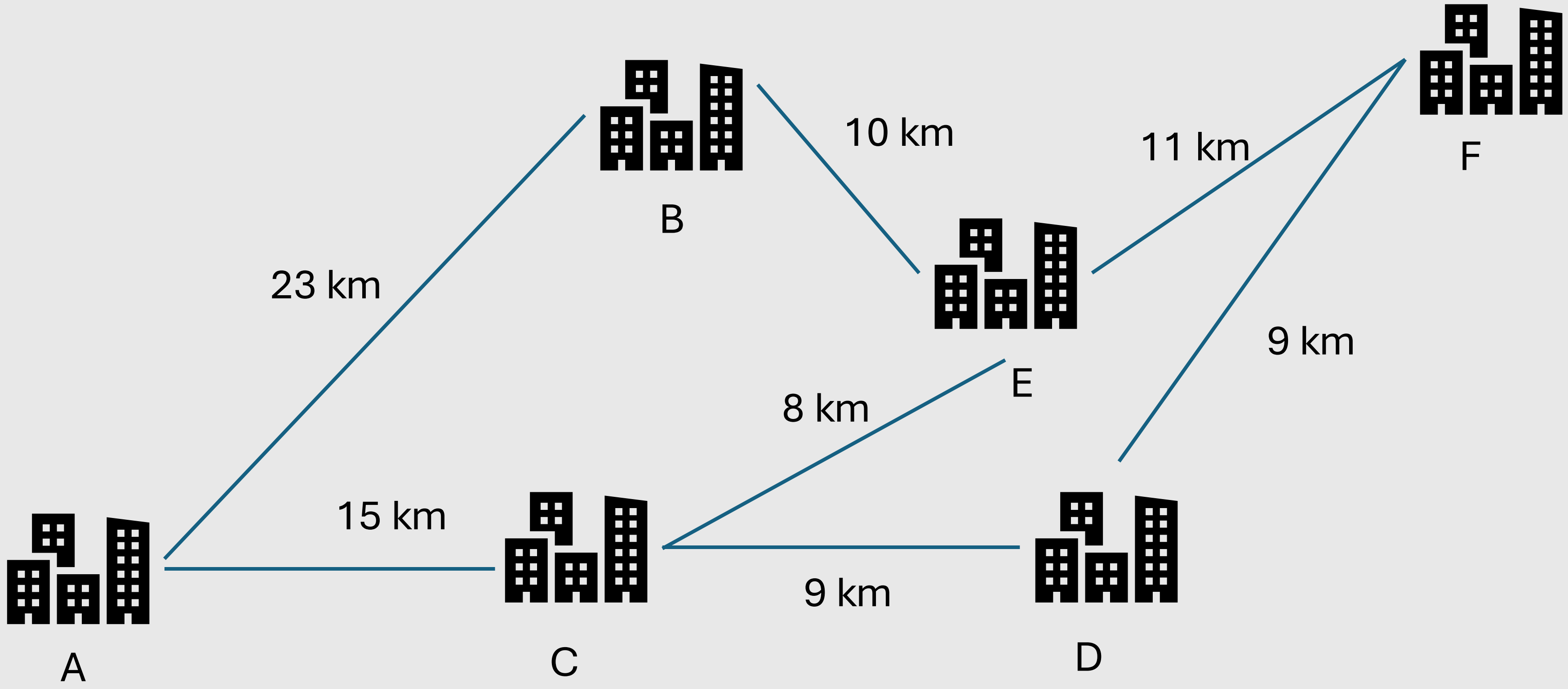
- Graphs have vertices and edges: G = (V, E).
- Graphs can be undirected and directed. Directed graphs have edges with directions.
- Graphs can be represented by adjacency-lists or adjacency-matrices. Edges can also be weighted.
- Graphs are used in many applications, e.g., route planning.





Graph algorithm: shortest path

- Objective: compute the shortest path between vertices in a graph. The graph can represent, e.g., a road map between cities.



Graph algorithm: formal setting

$G = (V, E), \text{with } w : E \to \mathcal{R}$

The path $p = (v_0, v_1, \dots v_k)$ has the weight $w(p)$:

$$w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i)$$

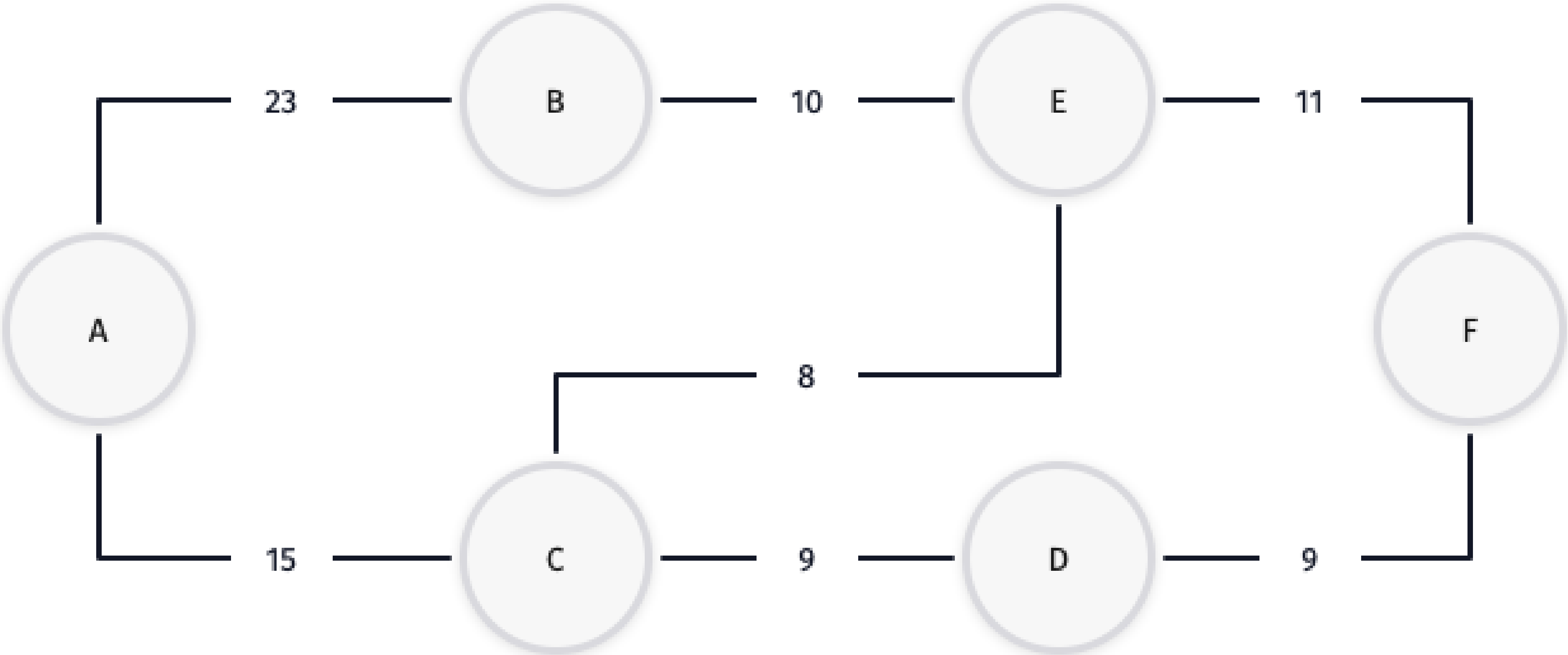
The shortest-path weight between u and v is

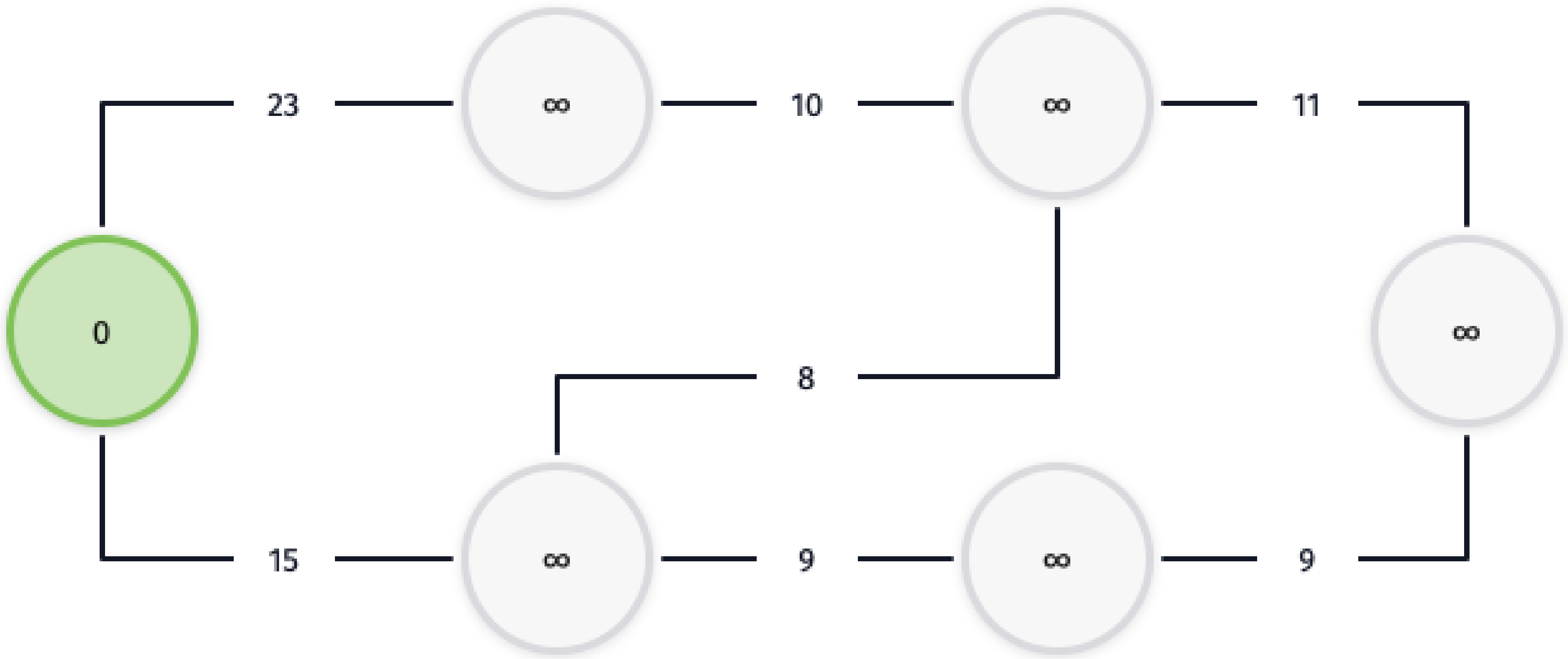
$$d(u, v) = \begin{cases} min\{w(p) : u \to_p v\} & \text{if a path from } u \text{ to } v \\ \infty & \text{otherwise} \end{cases}$$

Graph algorithm: Dijkstra shortest path

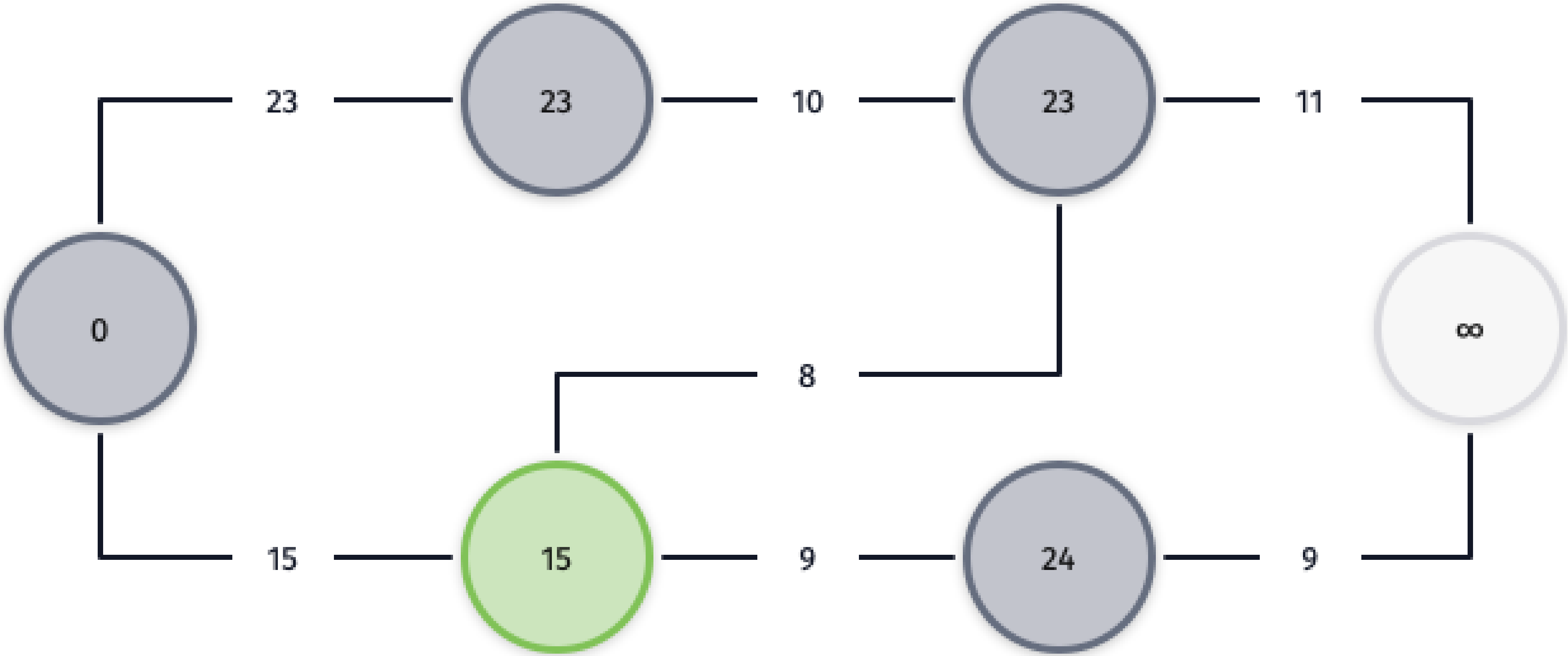
1. Initialize: Start with source node and set distance to infinity to all other nodes. Put the source node on a min-heap.
2. While the heap is not empty
   a) Take the node from the heap (it is the node with the smallest distance)
   b) Update its distance if a shorter path is found through the current node.

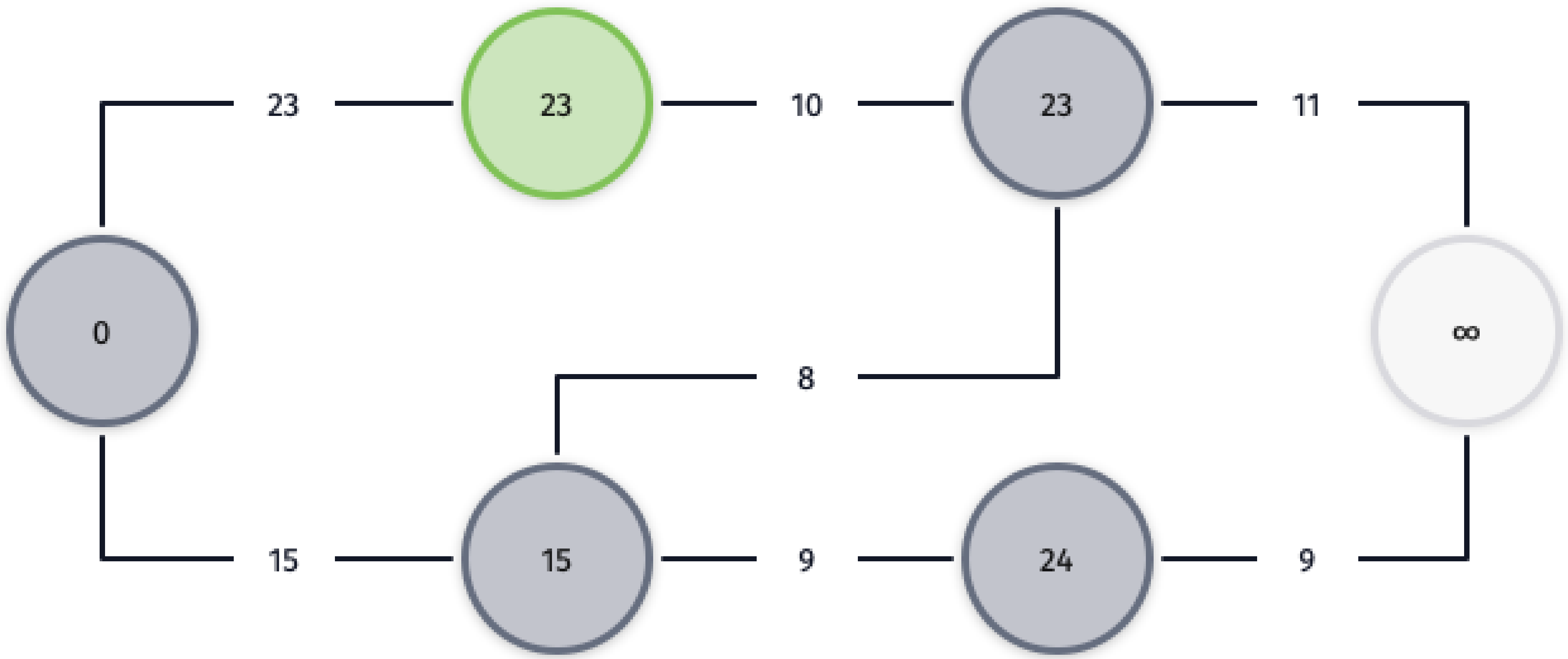
Note: To get the path, store and update the previous node to each node in 2. b). Then in a separate block, go back from the target node until source is reached.


A
B
C
D
E
F
23
10
11
8
15
9
9

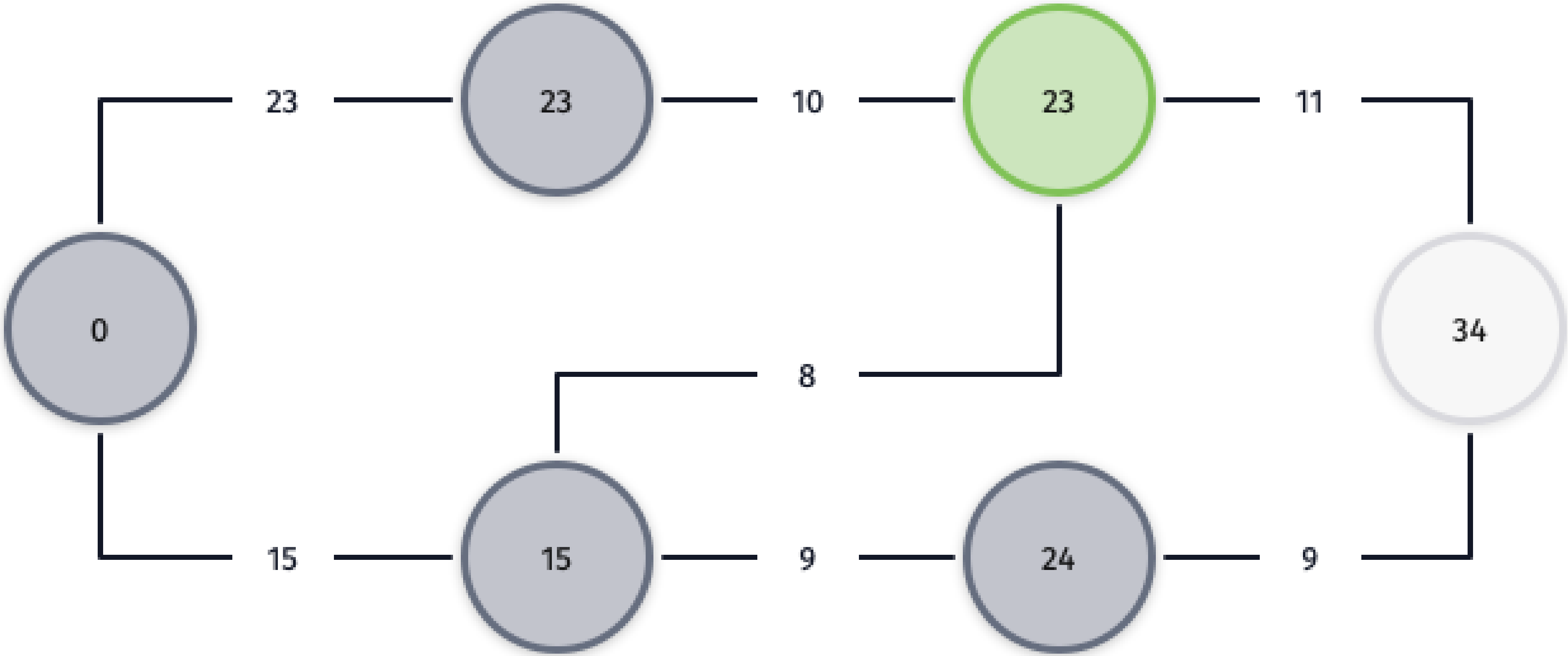


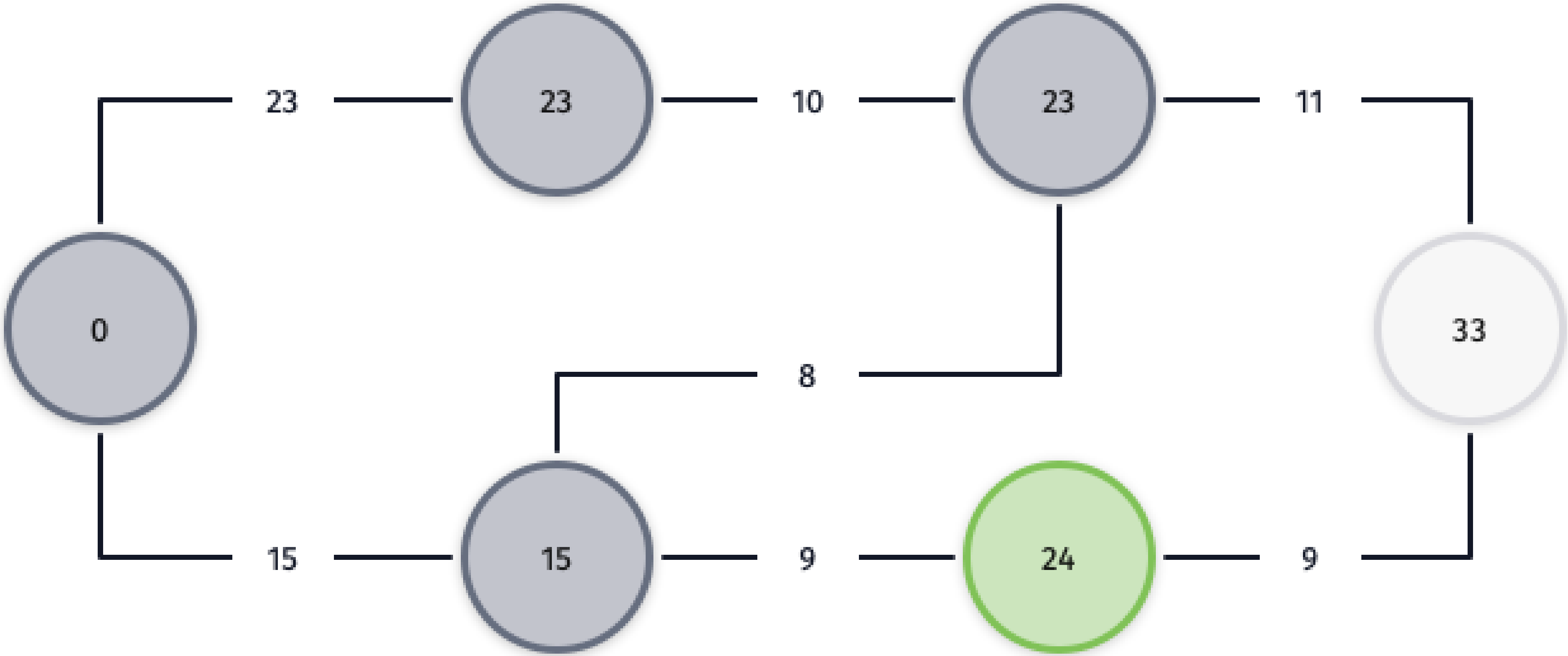
23
∞
10
∞
11
0
8
∞
15
∞
9
∞
9
∞


23
23
10
23
11
0
∞
8
15
15
9
24
9



23
23
10
23
11
0
∞
8
15
15
9
24
9


23
23
10
23
11
0
34
8
15
15
9
24
9


23
23
10
23
11
0
33
8
15
15
9
24
9

Dijkstra in pseudo code

Dijkstra(Graph, source):

1. Initialize distances: all to infinity, zero for source
2. Enqueue source with distance zero onto min-heap
3. While min-heap is not empty
   a) Node u = dequeue from min-heap
   b) For each neighbor v of u:

      If distance from source through u to v < current shortest path:

         Update distance of current shortest path

```python
def dijkstra(graph, start):
  # Initialize distances and priority queue
  distances = {node: float('infinity') for node in graph}
  distances[start] = 0
  priority_queue = [(0, start)]
  while priority_queue:
    current_distance, current_node = heapq.heappop(priority_queue)
    if current_distance > distances[current_node]:
      continue
    for neighbor, weight in graph[current_node].items():
      distance = current_distance + weight
      if distance < distances[neighbor]:
        distances[neighbor] = distance
        heapq.heappush(priority_queue, (distance, neighbor))
  return distances
```

Today's summary

- Algorithms explained using different sorting algorithms:
  - Insertion sort
  - Merge sort
  - Heap sort
- Concept of divide-and-conquer
- Elementary data structures
  - Stack, queue, list, tree, graph, heap
- Dijkstra as example of graph algorithms
- Basic idea of complexity of algorithms

Questions?