Question 1 (basic): complexity

Below are some operations parametrized by a variable n.

A. Merge sorting an array of size n.

B. Adding n^2 elements to an empty dynamic array.

C. Binary search in a sorted array of size 2^n.

Answer

For each operation, state its asymptotic time complexity in terms of n.
(For example, you can use O-notation with a simple, but exact bound.)

Question 2 (basic): heaps

Here is a minimum-heap with integer values:

0	1	2	3	4	5	6	7	8
2	12	5	16	20	8	10	33	17

We remove the minimum.

Answer

Draw the resulting heap in tree representation.

Question 3 (basic): searching and sorting

Here are my pets:

- **A**ndy, 5-year old ant
- **B**aba, 2-year old bee
- **C**indy, 18-year-old cat
- **D**avid, 9-year-old duck
- **E**lon, 3-year-old elephant
- **K**eke, 8-year old koala
- **Z**ack, 14-year-old zebra

I put them in an array alphabetically:

0	1	2	3	4	5	6
A	B	C	D	E	K	Z

But now I want to sort them **by age** instead (young before old). I heard quicksort can do that.

Can you show me how the partitioning works? Andy has agreed to be the pivot.
You only have to do the initial (first) partitioning, not the rest of quicksort.

Answer

State the swaps performed by the partitioning:

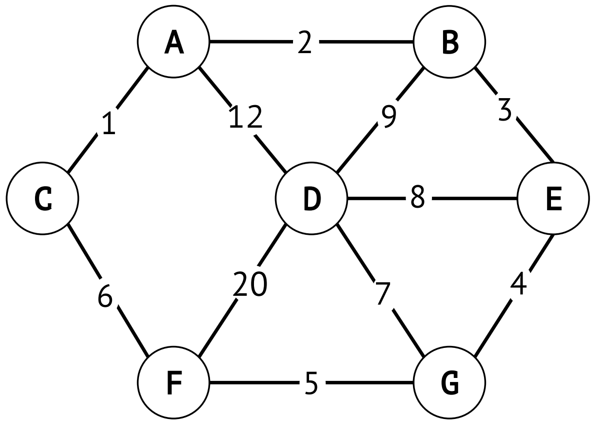
Show the array after partitioning:

0	1	2	3	4	5	6

If you used a partitioning algorithm different from that of your course, explain it:

Question 4 (basic): graphs

Here is an undirected graph with integer weights:



We run **Prim's algorithm** with starting node **D**.

Answer

Draw the resulting **minimum spanning tree** and **state the order** in which its edges are added.

Question 5 (basic): abstract data types

Here are some real-life situations:

A. You want to store the points each student gets in this exam. If they complain about their grade, you need to check their points to see if you calculated the grade incorrectly.

B. You don't have time to read all your emails. So you store the unread ones away. Later when you have time, you go through them by order of recency (most recent one first).

C. Someone gives you a map of Europe. You want to represent the information of which countries border each other.

For each situation, pick the abstract data type that models the situation most appropriately.
You can choose from the following:

- graph
- map
- priority queue
- queue
- set
- stack

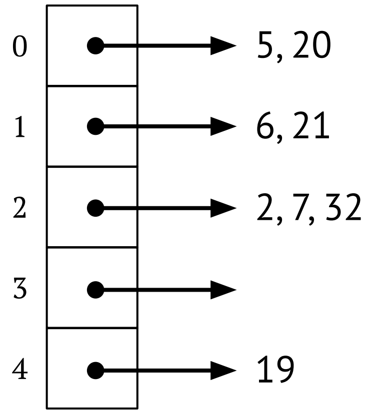
No justification is necessary.

Answer

Question 6 (basic): hash tables

We use a separate-chaining hash table to store integer values, using a **sorted list** in each slot. The hash function is modular compression (the remainder of dividing by the table length).

The following hash table of length 5 has gotten quite full:



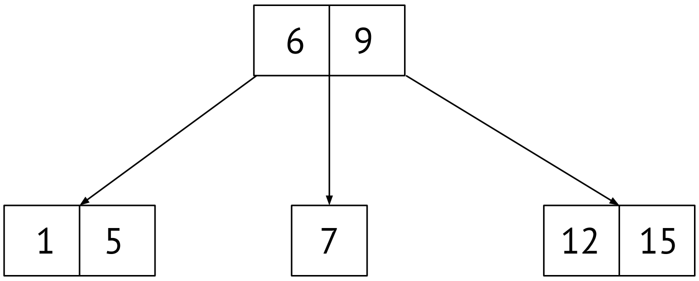
We resize it to table length 8.

Answer

Draw the resulting hash table. What is the new **load factor**?

Question 7 (basic): search trees

Here is a 2-3 tree (null nodes not drawn) with integer values:



We insert value 4.

Answer

Draw the resulting 2-3 tree.

Question 8 (basic): order of growth

Here are some mathematical functions in a variable n:

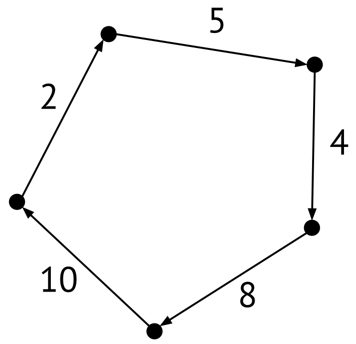
- $f(n) = 495n^3 + 182 \cdot 3^n$
- $g(n) = 2024\, log(n + 12)$
- $h(n) = 1000 + 40\, n^2 + 64\, n^4$
- $k(n) = 9001$

Answer

Order the functions f, g, h, k according to their **growth rate**.

The smallest growth rate (i.e., slowest growing) should come first.

Question 9 (advanced): shortest cycle



We consider directed graphs with natural number weights.

The *length* of a cycle is the sum of the weights of its edges.

Describe an algorithm that takes a graph and finds the smallest length of a cycle in that graph (returning ∞ if no cycle exists).

The graph is represented using adjacency lists, for example:

- graph.nodes(): Set<V>
- graph.outgoingEdges(*v*: V): List<E>

Your algorithm should run in time O(|*V*| |*E*| log(|*E*|)) where |*V*| is the number of nodes and |*E*| is the number of edges of the given graph. You can assume that:

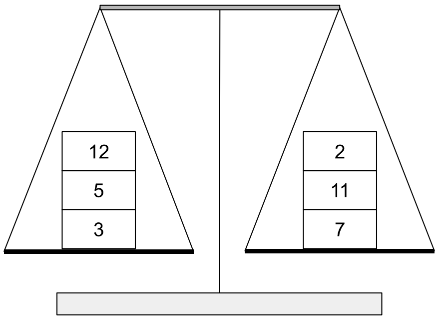
- the above graph API methods are O(1),
- iterating over a set or list does not have hidden costs.

You can freely use data structures and algorithms from the course – you do not have to explain how they work.

Answer

function *length_of_shortest_cycle*(*graph*: **AdjacencyListGraph**) → natural number or ∞:

Question 10 (advanced): complexity



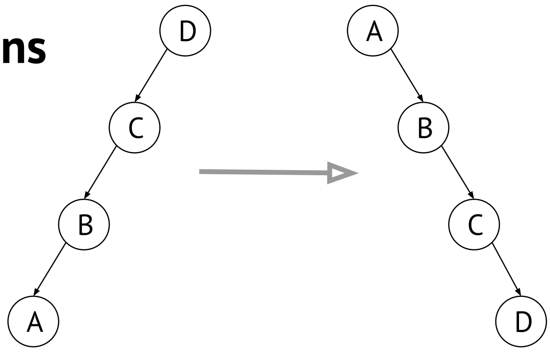
The following function balances an array of weights:

```
function balance(weights):
    left = new stack (using linked list)
    right = new stack (using linked list)

    function helper(k: int, left_total: int, right_total: int) → bool:
        if k equals length(weights):
            return left_total equals right_total

        left.push(weights[k])
        if helper(k + 1, left_total + weights[k], right_total):
            return true
        left.pop()

        right.push(weights[k])
        if helper(k + 1, left_total, right_total + weights[k]):
            return true
        right.pop()

        return false

    if helper(0, 0, 0):
        print "Solution found. Left side:"
        until left is empty: print left.pop()
        print "Right side:"
        until right is empty: print left.pop()
    else:
        print "Impossible to balance."
```

Determine the asymptotic **time complexity** and **space complexity** in the number *N* of weights.

- Printing a weight takes constant time and space.
- Printing a string takes time and space that is linear in the length of the string.
- Remember that your answer needs to be justified.

Question 11 (advanced): tree rotations



This question is about binary search trees (BSTs).
The following class represents non-empty nodes:

class *Node*:
    value
    left: Node
    right: Node

Recall that binary search trees can be unbalanced. Here, we look at two special cases:

- A binary tree is *left extreme* if the right child of any node is empty.
- A binary tree is *right extreme* if the left child of any node is empty.

We wish to turn a left extreme BST into a right extreme BST **using only rotations**.
Fortunately, I have already implemented the rotation functions for you to use.
They take a node and return the node that replaces it after the rotation.

- *rotate_left*(*node*: Node) → Node
- *rotate_right*(*node*: Node) → Node

Design an algorithm *left_to_right* that takes a left extreme BST and returns a right extreme BST storing the same values. Constraints:

- You may not create any new nodes or change node values.
- You may only change a child pointer by replacing it with the result of applying a rotation.

Answer

function *left_to_right*(*node*: Node) → Node:

Question 12 (advanced): heaps

In a binary minimum-heap, we rely on two operations to restore the heap invariant:

- *swim up*: while node is smaller than parent, swap it with parent,
- *sink down*: while node is larger than child, swap it with smaller child.

We have an array of *N* elements (representing a complete binary tree) and wish to turn it into a minimum heap. Instead of inserting the elements one-by-one, we want to run swim-up and sink-down operations in the array in some order to make the heap invariant satisfied.

Here are two possible strategies:

A. Go over the array from start to end and run sink-down at each position.
B. Go over the array from end to start (reverse order) and run sink-down at each position.

Answer

Determine which of the above strategies work correctly.

- For the ones that do, explain why.
- For the ones that do not, show a small counterexample.

Question 1 (basic): complexity

Below are some operations parametrized by a variable n.

A. Merge sorting an array of size n.

B. Adding n^2 elements to an empty dynamic array.

C. Binary search in a sorted array of size 2^n.

Answer

For each operation, state its asymptotic time complexity in terms of n.
(For example, you can use O-notation with a simple, but exact bound.)

A. O(n log(n))
B. O(n^2)
C. O(n)

Question 2 (basic): heaps

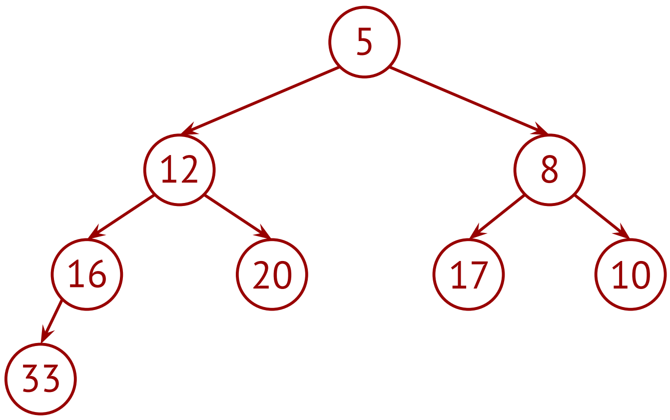
Here is a minimum-heap with integer values:

0	1	2	3	4	5	6	7	8
2	12	5	16	20	8	10	33	17

We remove the minimum.

Answer

Draw the resulting heap in tree representation.



Question 3 (basic): searching and sorting

Here are my pets:

- **A**ndy, 5-year old ant
- **B**aba, 2-year old bee
- **C**indy, 18-year-old cat
- **D**avid, 9-year-old duck
- **E**lon, 3-year-old elephant
- **K**eke, 8-year old koala
- **Z**ack, 14-year-old zebra

I put them in an array alphabetically:

0	1	2	3	4	5	6
A	B	C	D	E	K	Z

But now I want to sort them **by age** instead (young before old). I heard quicksort can do that.

Can you show me how the partitioning works? Andy has agreed to be the pivot.
You only have to do the initial (first) partitioning, not the rest of quicksort.

Answer

State the swaps performed by the partitioning:

1. Swap positions 2 and 4 (C and E).
2. Swap positions 0 and 2 (A and E, swapping the pivot into place).

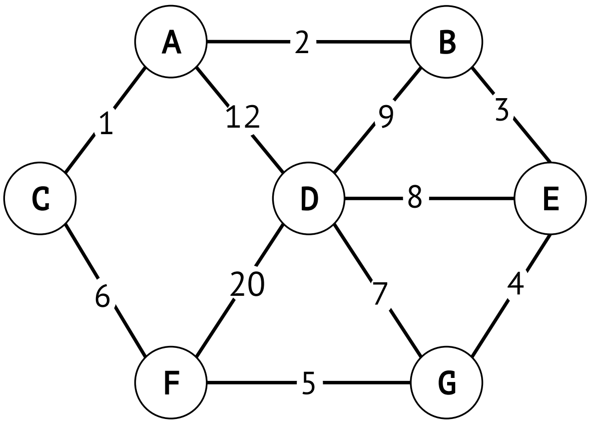
Show the array after partitioning:

0	1	2	3	4	5	6
E	B	A	D	C	K	Z

If you used a partitioning algorithm different from that of your course, explain it:

Question 4 (basic): graphs

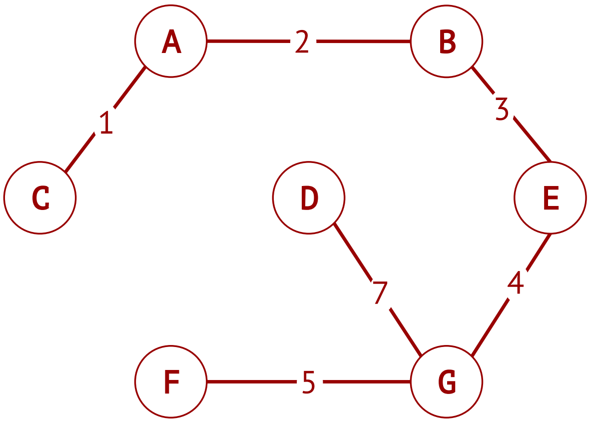
Here is an undirected graph with integer weights:



We run **Prim's algorithm** with starting node **D**.

Answer

Draw the resulting **minimum spanning tree** and **state the order** in which its edges are added.



The edges are added in this order: DG, GE, EB, BA, AC, GF.

Question 5 (basic): abstract data types

Here are some real-life situations:

A. You want to store the points each student gets in this exam. If they complain about their grade, you need to check their points to see if you calculated the grade incorrectly.

B. You don't have time to read all your emails. So you store the unread ones away. Later when you have time, you go through them by order of recency (most recent one first).

C. Someone gives you a map of Europe. You want to represent the information of which countries border each other.

For each situation, pick the abstract data type that models the situation most appropriately.
You can choose from the following:

- graph
- map
- priority queue
- queue
- set
- stack

No justification is necessary.

Answer

A. A **map** from students (keys) to points (values).

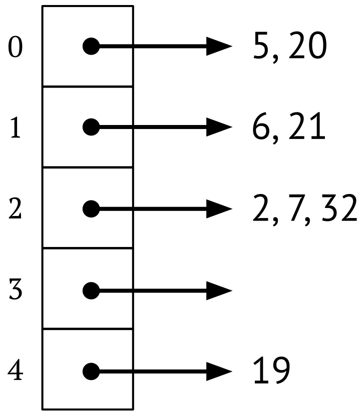
B. A **stack** of unread emails. New ones go on top (push) and the next to go through is also taken from the top (pop). Last in, first out.

C. A **graph** with countries as nodes and borders as edges. Alternatively, a **map** from countries to sets of neighbouring countries (this is one way to represent a graph).

Question 6 (basic): hash tables

We use a separate-chaining hash table to store integer values, using a **sorted list** in each slot. The hash function is modular compression (the remainder of dividing by the table length).

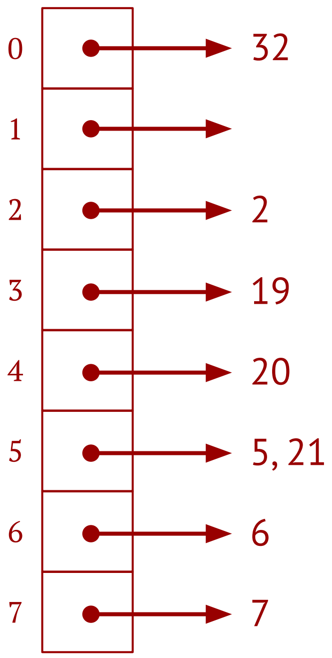
The following hash table of length 5 has gotten quite full:



We resize it to table length 8.

Answer

Draw the resulting hash table. What is the new **load factor**?

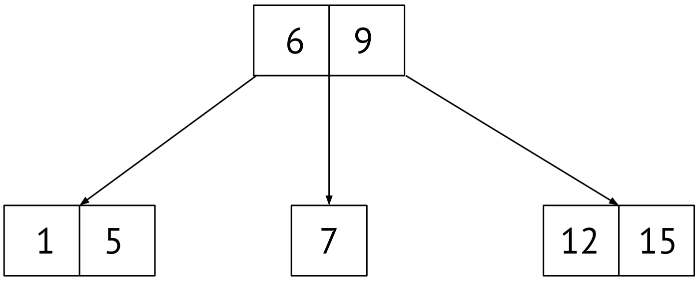


The new load factor is "number of elements" / "table length" = 8 / 8 = 1.

Note: a minor mistake with the modulus calculation is acceptable.

Question 7 (basic): search trees

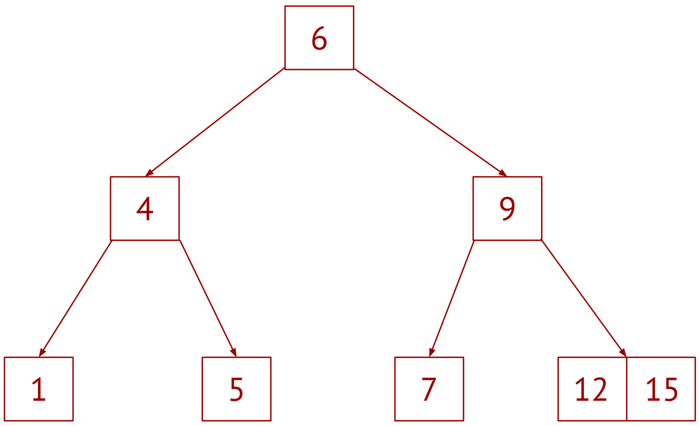
Here is a 2-3 tree (null nodes not drawn) with integer values:



We insert value 4.

Answer

Draw the resulting 2-3 tree.



Question 8 (basic): order of growth

Here are some mathematical functions in a variable n:

- $f(n) = 495n^3 + 182 \cdot 3^n$
- $g(n) = 2024\, log(n + 12)$
- $h(n) = 1000 + 40\, n^2 + 64\, n^4$
- $k(n) = 9001$

Answer

Order the functions f, g, h, k according to their **growth rate**.

The smallest growth rate (i.e., slowest growing) should come first.

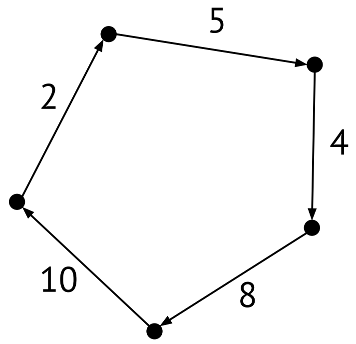
We simplify the growth rates, for example using Θ-notation:

- $\mathrm{f}(n) \in \Theta\left(3^n\right)$ (f is exponential with base 3)
- $\mathrm{g}(n) \in \Theta(\log(n))$ (g is logarithmic)
- $\mathrm{h(n)} \in \Theta\left(n^4\right)$ (h is polynomial with exponent 4, or *quadric*)
- $\mathrm{k(n)} \in \Theta(1)$ (k is constant)

So the desired order of growth rates is:

$$\mathrm{k} \prec \mathrm{g} \prec \mathrm{h} \prec \mathrm{f}$$

Question 9 (advanced): shortest cycle



We consider directed graphs with natural number weights.

The *length* of a cycle is the sum of the weights of its edges.

Describe an algorithm that takes a graph and finds the smallest length of a cycle in that graph (returning ∞ if no cycle exists).

The graph is represented using adjacency lists, for example:

- graph.nodes(): Set<V>
- graph.outgoingEdges(*v*: V): List<E>

Your algorithm should run in time O(|*V*| |*E*| log(|*E*|)) where |*V*| is the number of nodes and |*E*| is the number of edges of the given graph. You can assume that:

- the above graph API methods are O(1),
- iterating over a set or list does not have hidden costs.

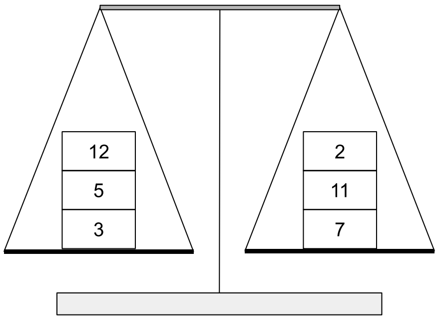
You can freely use data structures and algorithms from the course – you do not have to explain how they work.

Answer

There are many ways. This one goes over all |V| nodes and runs a version of uniform-cost search from the node to find the shortest non-empty "path" to itself (O(|*E*| log(|*E*|)).

```
function length_of_shortest_cycle(graph: AdjacencyListGraph) → natural number or ∞:
    r = ∞
    for v in graph.nodes():
        r = min(r, cycle_search(graph, v))
    return r

function cycle_search(graph: AdjacencyListGraph, start: V) → natural number or ∞:
    visited = new set of nodes
    agenda = new min-priority queue (using binary heap) of nodes with cost
    for e in graph.outgoingEdges(start):
        agenda.add(target of e, cost: weight of e)
    while agenda not empty:
        (v, cost) = agenda.removeMin()
        if not v in visited:
            add v to visited
            if v equals start:
                return cost
            for e in graph.outgoingEdges(start):
                agenda.add(target of e, cost: cost plus weight of e)
    return ∞
```

Question 10 (advanced): complexity



The following function balances an array of weights:

```
function balance(weights):
    left = new stack (using linked list)
    right = new stack (using linked list)

    function helper(k: int, left_total: int, right_total: int) → bool:
        if k equals length(weights):
            return left_total equals right_total

        left.push(weights[k])
        if helper(k + 1, left_total + weights[k], right_total):
            return true
        left.pop()

        right.push(weights[k])
        if helper(k + 1, left_total, right_total + weights[k]):
            return true
        right.pop()

        return false

    if helper(0, 0, 0):
        print "Solution found. Left side:"
        until left is empty: print left.pop()
        print "Right side:"
        until right is empty: print left.pop()        Typo: should be right.pop()
    else:
        print "Impossible to balance."
```

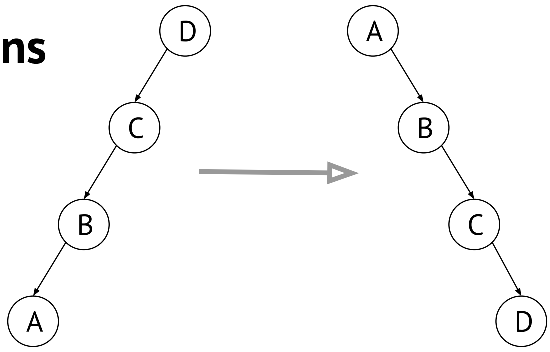
Determine the asymptotic **time complexity** and **space complexity** in the number N of weights.

- Printing a weight takes constant time and space.
- Printing a string takes time and space that is linear in the length of the string.
- Remember that your answer needs to be justified.

The function helper is recursive. The index argument k starts at 0 (call helper(0, 0, 0)) and increases by one in the recursive calls. The base case is when k equals N. So the recursion depth is N. This also means that the stacks left and right always have at most N elements. The call stack also needs $O(N)$ space. Therefore, the **space complexity** of balance is **$O(N)$**.

Each call to helper is either the base case or makes two recursive calls (branching factor two). Therefore, there are $O(2^N)$ calls to helper in total. Pushing and popping is $O(1)$, so the time spent in each call to helper, ignoring the cost of the recursive calls, is $O(1)$. This makes helper(0, 0, 0) take $O(2^N)$ time. The printing is bounded by the stack sizes, so is $O(N)$. In total, the **time complexity** of balance is **$O(2^N)$**.

Question 11 (advanced): tree rotations



This question is about binary search trees (BSTs).
The following class represents non-empty nodes:

```
class Node:
    value
    left: Node
    right: Node
```

Recall that binary search trees can be unbalanced. Here, we look at two special cases:

- A binary tree is *left extreme* if the right child of any node is empty.
- A binary tree is *right extreme* if the left child of any node is empty.

We wish to turn a left extreme BST into a right extreme BST **using only rotations**.
Fortunately, I have already implemented the rotation functions for you to use.
They take a node and return the node that replaces it after the rotation.

- *rotate_left*(*node*: Node) → Node
- *rotate_right*(*node*: Node) → Node

Design an algorithm *left_to_right* that takes a left extreme BST and returns a right extreme BST storing the same values. Constraints:

- You may not create any new nodes or change node values.
- You may only change a child pointer by replacing it with the result of applying a rotation.

Answer

There are many ways to do this. They differ in the rotation count used for a BST of size *N*.

This one uses *N* - 1 rotations by right rotating only at the root:

```
function left_to_right(node: Node) → Node:
    if node is null:
        return null
    while node.left is not null:
        node = rotate_right(node)
    return node
```

It is possible to keep the tree linear throughout, at the cost of more rotations.

Question 12 (advanced): heaps

In a binary minimum-heap, we rely on two operations to restore the heap invariant:

- *swim up*: while node is smaller than parent, swap it with parent,
- *sink down*: while node is larger than child, swap it with smaller child.

We have an array of N elements (representing a complete binary tree) and wish to turn it into a minimum heap. Instead of inserting the elements one-by-one, we want to run swim-up and sink-down operations in the array in some order to make the heap invariant satisfied.

Here are two possible strategies:

A. Go over the array from start to end and run sink-down at each position.
B. Go over the array from end to start (reverse order) and run sink-down at each position.

Answer

Determine which of the above strategies work correctly.

- For the ones that do, explain why.
- For the ones that do not, show a small counterexample.

Strategy **A** fails. Consider the following array of characters with alphabetical ordering:

0	1	2	3
B	C	C	A

- Sinking down position 0 does nothing as C is not smaller than B.
- Sinking down position 1 swaps positions 1 (C) and 3 (A).
- Sinking down positions 2 and 3 does nothing as there are no children.

The resulting array is not a valid minimum-heap as the root B has smaller child A.

Strategy **B** works. We can show by induction that the heap invariant holds for all nodes for which we have called sink-down. More precisely, consider the following property:

P(K) After running sink-down for positions from K to N in reverse order,
every position in that range is smaller or equal than its children.

- P(N) is true because the range is empty.
- Given $K < N$ such that P($K + 1$), then sinking down at position K makes P(K) true.

By induction, P(K) holds for all $K < N$. In particular, P(0) holds.

Exercise: This procedure only takes O(N) time (why?). Can we sort in O(N) time this way?