TUMGAD - Solutions

Generated: Wed May 13 18:54:52 CEST 2020
Seed: 586059998

——Disclaimers and Infos:——

1. You can find explanations to all the components in this repository and instructions on how to use the generator here.

2. This paper was generated by automated software written by students. It might contain flaws. If you find one (or even think you've found one), please report it here.

3. TUMGADs creators are not affiliated with the lecture organization whatsoever, the exercises/explanations are not guaranteed to be accurate or complete.

4. The algorithms in this tool are sometimes substantially oversimplified, this is due to the students having to execute them by hand in exams and thus it is impractical to increase the difficulty (e.g. RadixSort only deals with 3-digit numbers).

5. If you like this tool, a good thing you can do is spread the word or star the repo on GitHub to help out more of your fellow students as well as the creators.

Hashing w/ Chaining		Double Hashing	
MergeSort		QuickSort	
RadixSort		Binary Heaps	
BinomialHeaps		AVL Trees	
(a,b) Trees		BFS & DFS	
Floyd Warshall		Dijkstra	

TUMGAD was created by Sebastian Oßner and Daniel Krüger View it on GitHub

Hashing with Chaining

Visualize **Hashing with Chaining**. The size of a hashtable is set to $m = 11$.
The following operations are to be executed:

Insert: 13, 15, 12, 2, 0, 20, 3, 11, 7, 5, 14

Use the following hash function:

$$h(x) = (7x + 7) \mod 11$$

a) Compute the values to which each one of the above numbers map to and enter them into this table:

x	0	2	3	5	7	11	12	13	14	15	20
$h(x)$	7	10	6	9	1	7	3	10	6	2	4

b) Execute the operations above in their given order and enter them into the hashtable preprints.

Insert: 13

0	1	2	3	4	5	6	7	8	9	10
										13

Insert: 15

0	1	2	3	4	5	6	7	8	9	10
		15								13

Insert: 12

0	1	2	3	4	5	6	7	8	9	10
		15	12							13

Insert: 2

0	1	2	3	4	5	6	7	8	9	10
		15	12							13
										2

Insert: 0

0	1	2	3	4	5	6	7	8	9	10
		15	12				0			13
										2

Insert: 20

0	1	2	3	4	5	6	7	8	9	10
		15	12	20			0			13 2

Insert: 3

0	1	2	3	4	5	6	7	8	9	10
		15	12	20		3	0			13 2

Insert: 11

0	1	2	3	4	5	6	7	8	9	10
		15	12	20		3	0 11			13 2

Insert: 7

0	1	2	3	4	5	6	7	8	9	10
	7	15	12	20		3	0 11			13 2

Insert: 5

0	1	2	3	4	5	6	7	8	9	10
	7	15	12	20		3	0 11		5	13 2

Insert: 14

0	1	2	3	4	5	6	7	8	9	10
	7	15	12	20		3 14	0 11		5	13 2

Double Hashing

Use **Double Hashing** and the following Hash functions to resolve any collisions.

$$h(x,i) = (h(x) + i * h'(x)) \mod 11$$
$$h(x) = (6x + 0) \mod 11$$
$$h'(x) = 5 - (x \mod 5)$$

Consider these operations:

Insert: 12, 7, 2, 11
Delete: 7, 2, 11
Insert: 5, 9, 7

Multiple hashtables with the size $m = 11$ are provided on the next page.

a) Generate the **collision table**, in the printed template on this page for the numbers above.
b) Execute the above operations in the provided order, also explicitly state **all checked positions** in the field "Position(s)" above the table (even in deletions).

This table shows the collisions the numbers could go through when inserting them into the hashtable:

x	$h(x)$	$h'(x)$	$h(x,0)$	$h(x,1)$	$h(x,2)$	$h(x,3)$	$h(x,4)$	$h(x,5)$
2	1	3	1	4	7	10	2	5
5	8	5	8	2	7	1	6	0
7	9	3	9	1	4	7	10	2
9	10	1	10	0	1	2	3	4
11	0	4	0	4	8	1	5	9
12	6	3	6	9	1	4	7	10

Operation: Insert(12) Position(s): 6

0	1	2	3	4	5	6	7	8	9	10
						12				

Operation: Insert(7) Position(s): 9

0	1	2	3	4	5	6	7	8	9	10
						12			7	

Operation: Insert(2) Position(s): 1

0	1	2	3	4	5	6	7	8	9	10
	2					12			7	

Operation: Insert(11) Position(s): 0

0	1	2	3	4	5	6	7	8	9	10
11	2					12			7	

Operation: Delete(7) Position(s): 9

0	1	2	3	4	5	6	7	8	9	10
11	2					12				

Operation: Delete(2) Position(s): 1

0	1	2	3	4	5	6	7	8	9	10
11						12				

Operation: Delete(11) Position(s): 0

0	1	2	3	4	5	6	7	8	9	10
						12				

Operation: Insert(5) Position(s): 8

0	1	2	3	4	5	6	7	8	9	10
						12		5		

Operation: Insert(9) Position(s): 10

0	1	2	3	4	5	6	7	8	9	10
						12		5		9

Operation: Insert(7) Position(s): 9

0	1	2	3	4	5	6	7	8	9	10
						12		5	7	9

MergeSort

Sort the following array according to the MergeSort algorithm:

4, 33, 6, 36, 35, 49, 34, 39, 21, 44, 32, 9, 48, 37, 16, 31

Indicate divisions of the array by clear vertical lines like so:

b, h, j, l, g, a | c, e, d, k, f, i

Split:

4, 33, 6, 36, 35, 49, 34, 39 | 21, 44, 32, 9, 48, 37, 16, 31

Split:

4, 33, 6, 36 | 35, 49, 34, 39 | 21, 44, 32, 9 | 48, 37, 16, 31

Split:

4, 33 | 6, 36 | 35, 49 | 34, 39 | 21, 44 | 32, 9 | 48, 37 | 16, 31

Split:

4 | 33 | 6 | 36 | 35 | 49 | 34 | 39 | 21 | 44 | 32 | 9 | 48 | 37 | 16 | 31

Merge:

4, 33 | 6, 36 | 35, 49 | 34, 39 | 21, 44 | 9, 32 | 37, 48 | 16, 31

Merge:

4, 6, 33, 36 | 34, 35, 39, 49 | 9, 21, 32, 44 | 16, 31, 37, 48

Merge:

4, 6, 33, 34, 35, 36, 39, 49 | 9, 16, 21, 31, 32, 37, 44, 48

Merge:

4, 6, 9, 16, 21, 31, 32, 33, 34, 35, 36, 37, 39, 44, 48, 49

QuickSort

Sort the following array according to the **QuickSort** algorithm:

$$6, 15, 11, 14, 10, 4, 7, 3, 2$$

Choose the last element of the array as the pivot element.
It helps to clearly mark sub-arrays in each step like so:

Step one - pivot: i
b, h, f, l, g, a, c, e, d | i | j, k

pivot: 2

New Array: 2, 15, 11, 14, 10, 4, 7, 3, 6

pivot: 6

New Array: 2, 3, 4, 6, 10, 11, 7, 15, 14

pivot: 4

New Array: 2, 3, 4, 6, 10, 11, 7, 15, 14

pivot: 14

New Array: 2, 3, 4, 6, 10, 11, 7, 14, 15

pivot: 7

New Array: 2, 3, 4, 6, 7, 11, 10, 14, 15

pivot: 10

New Array: 2, 3, 4, 6, 7, 10, 11, 14, 15

RadixSort

a) Sort the following array of integers according to the **LSD RadixSort** algorithm:

608, 926, 798, 313, 779, 679, 536, 722, 794

First round:

0	1	2	3	4	5	6	7	8	9
		722	313	794		926 536		608 798	779 679

New List: 722, 313, 794, 926, 536, 608, 798, 779, 679

Second round:

0	1	2	3	4	5	6	7	8	9
608	313	722 926	536				779 679		794 798

New List: 608, 313, 722, 926, 536, 779, 679, 794, 798

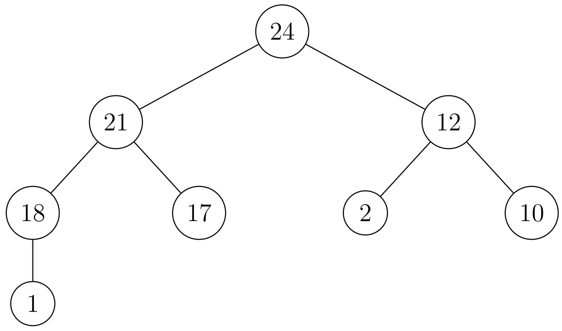
Third round:

0	1	2	3	4	5	6	7	8	9
			313		536	608 679	722 779 794 798		926

Sorted Array: 313, 536, 608, 679, 722, 779, 794, 798, 926

Binary Heaps

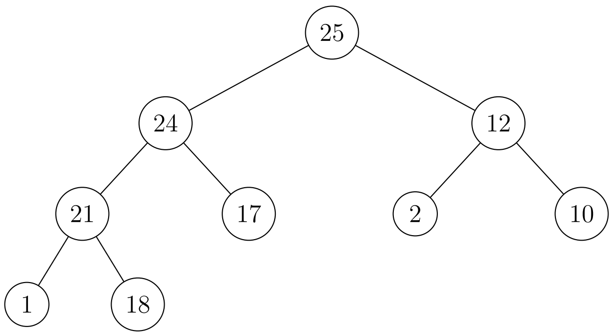
Given the following Binary max-Heap:



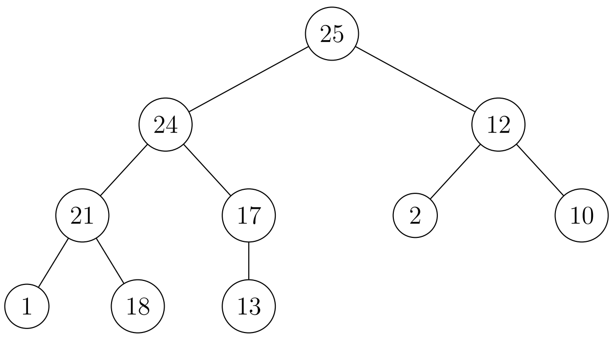
a) Perform the following insert-operations and their corresponding sift-operations on the above heap:

25, 13, 11, 7, 4

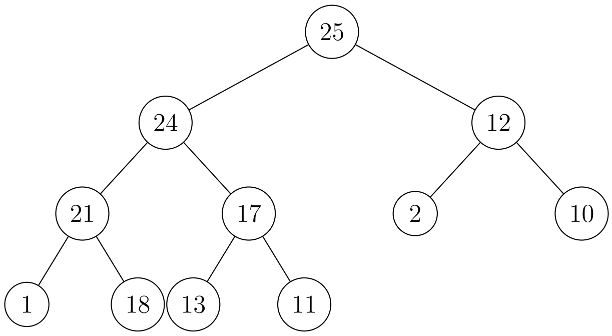
Insert: 25



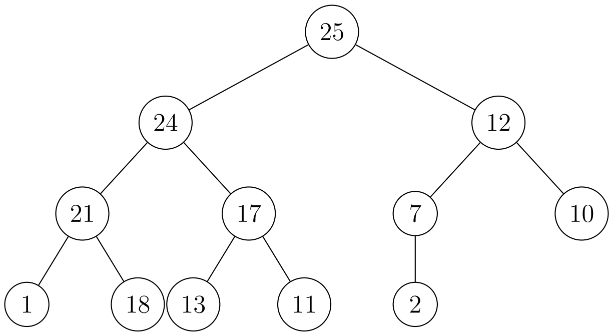
Insert: 13



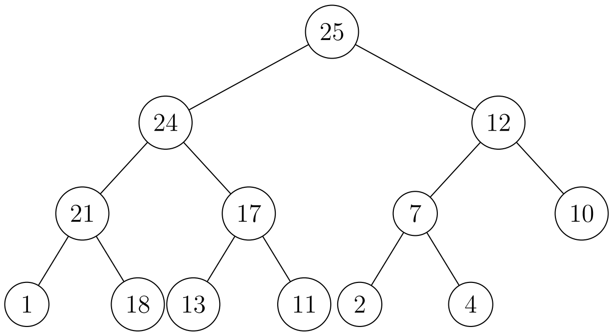
Insert: 11



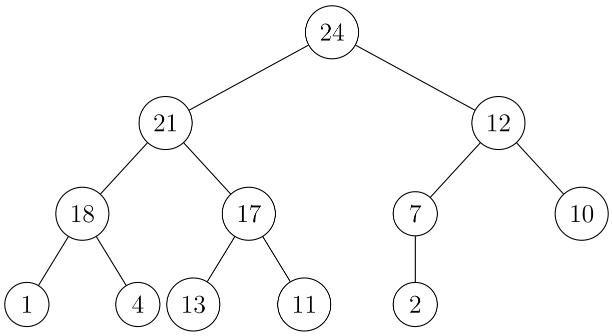
Insert: 7

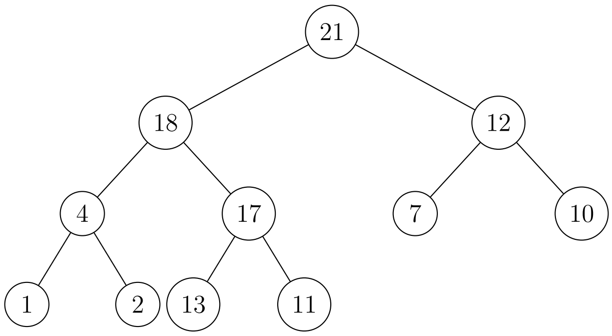


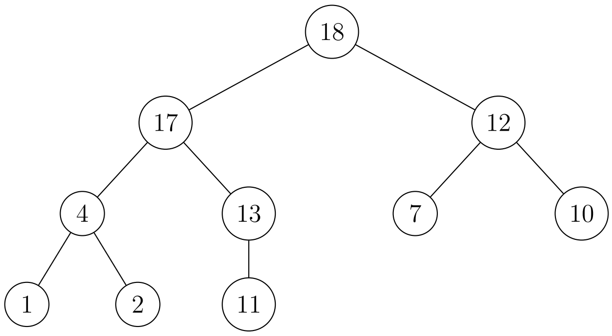
Insert: 4

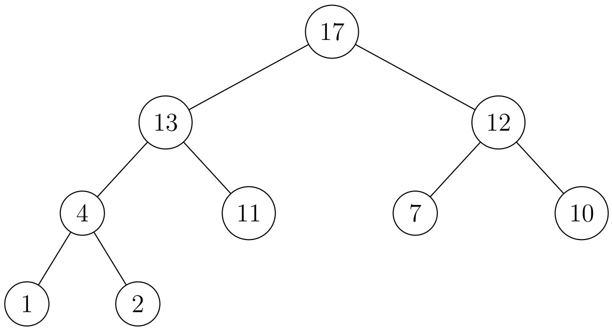


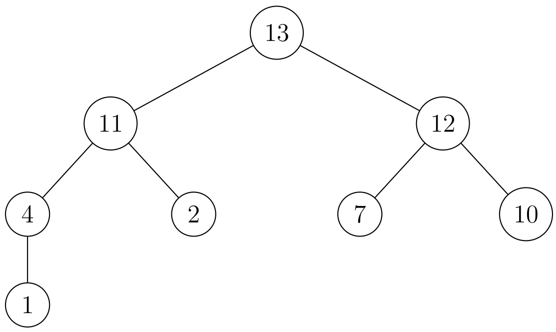
b) Now perform the known deleteMax() operation and its sift-operations 5 times




21
18
12
4
17
7
10
1
2
13
11

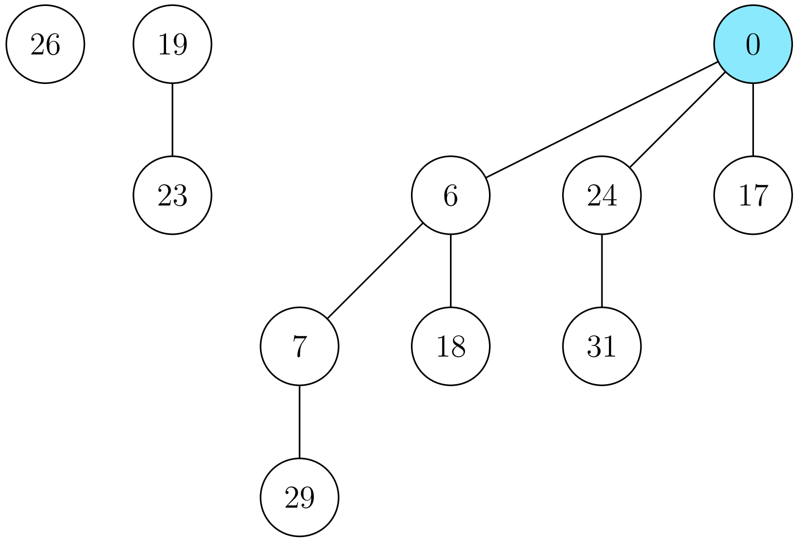

18
17
12
4
13
7
10
1
2
11


17
13
12
4
11
7
10
1
2


13
11
12
4
2
7
10
1

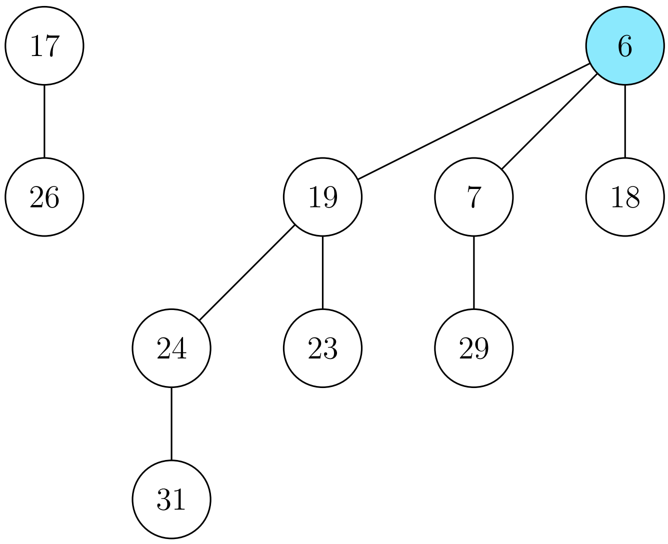
Binomial Heaps

Given the following Binomial Heap:

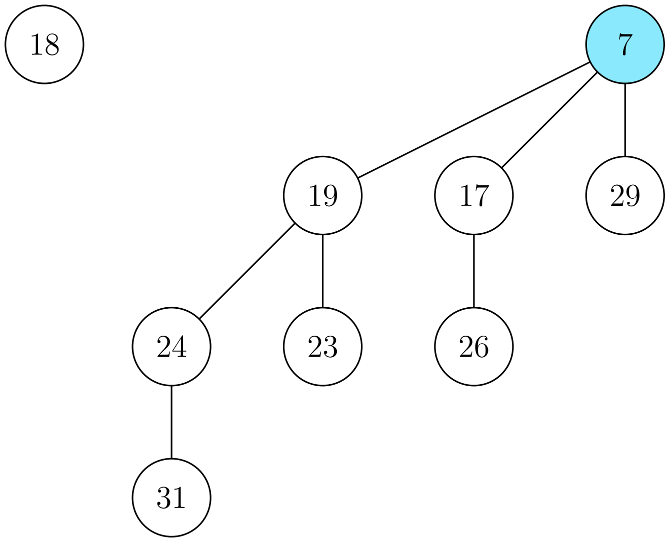


a) Execute the known deleteMin() operation **four times** on the above heap and be careful to meet the heap-criteria after each step:

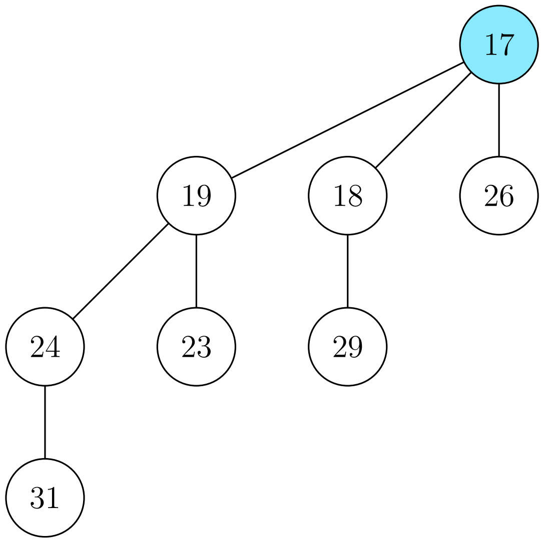
deleteMin(): min = 0



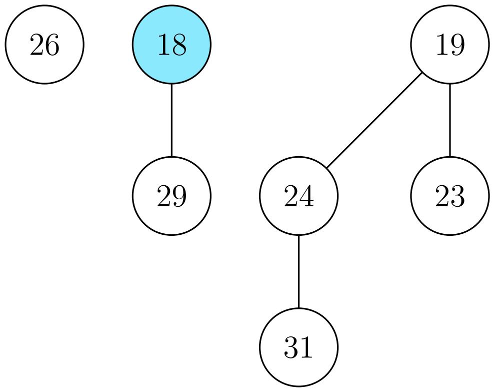
deleteMin(): min = 6



deleteMin(): min = 7



deleteMin(): min = 17

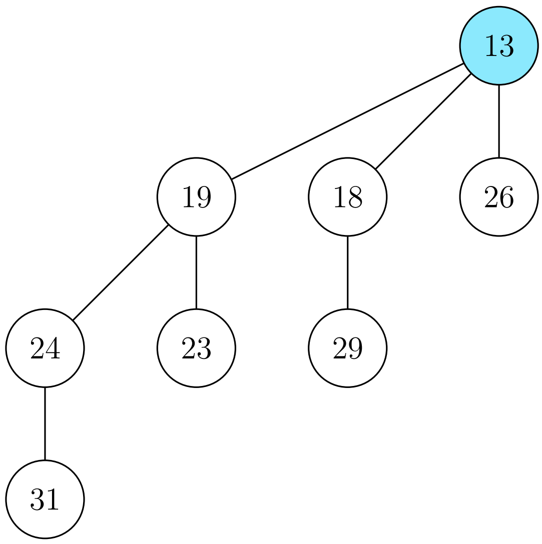


b) Now insert the following 5 values:

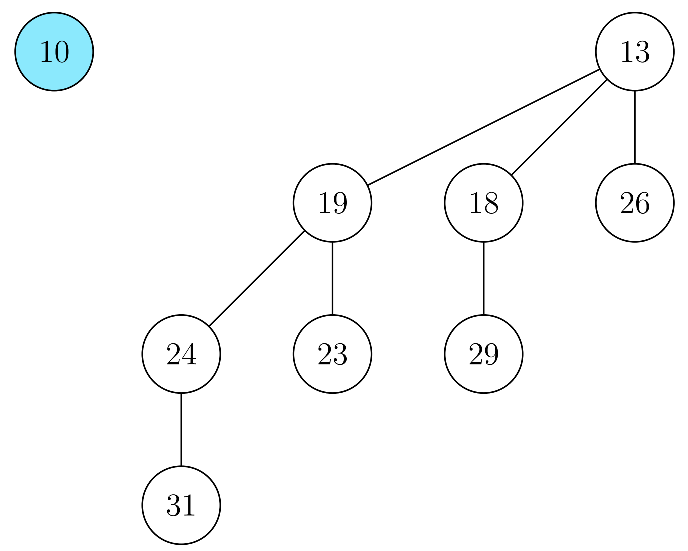
13, 10, 1, 20, 12

again, make sure to meet the heap-criteria after each step.

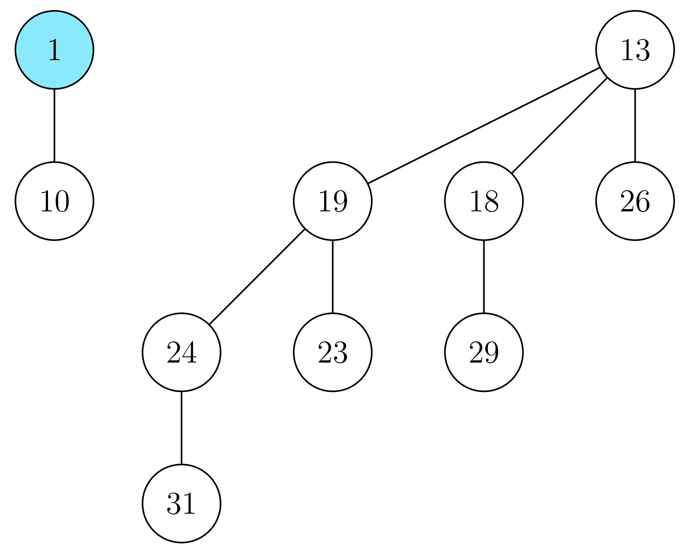
Insert: 13



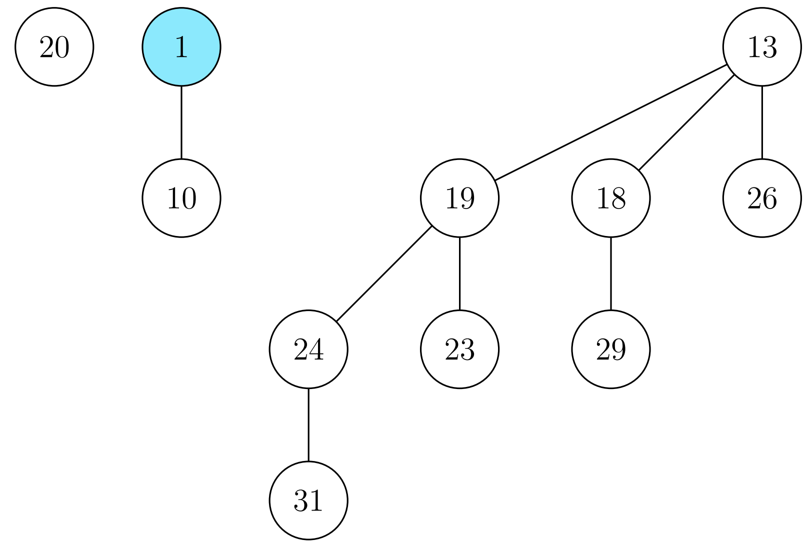
Insert: 10



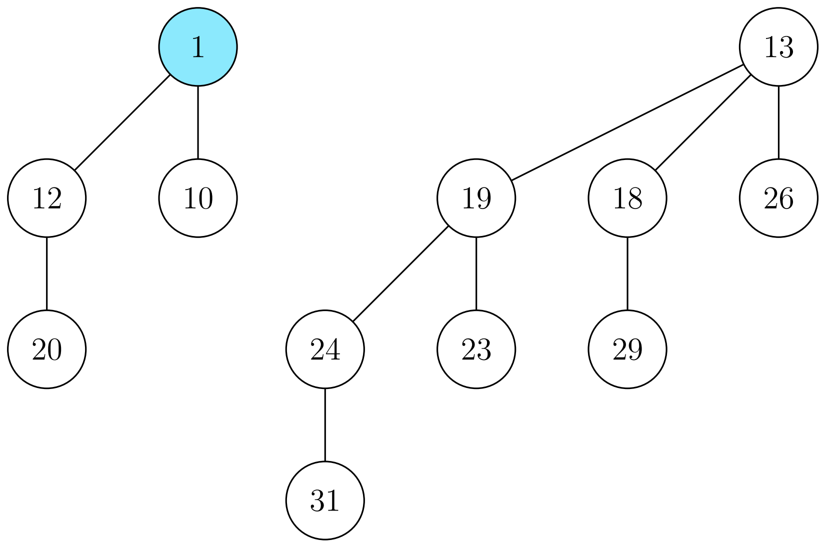
Insert: 1



Insert: 20

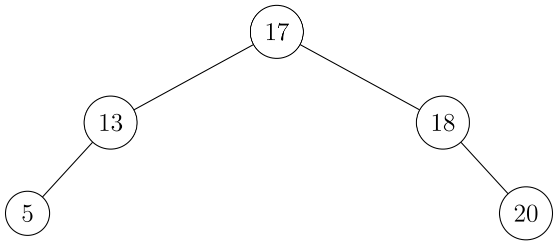


Insert: 12



AVL Trees

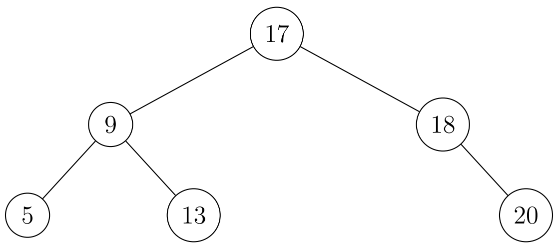
Given the following AVL Tree:



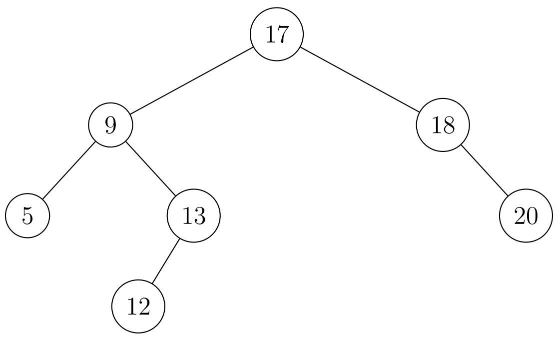
Perform the following operations in their given order on the tree above and check the boxes above the preprints if any rotation was performed.

Insert: 9, 12, 23, 3, 7
Delete: 13, 3
Insert: 1, 2

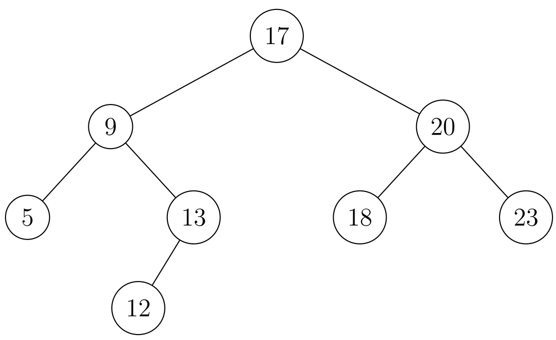
Insert: 9 ☐ l rotation ☐ r rotation ☒ l-r rotation ☐ r-l rotation ☐ no rotation



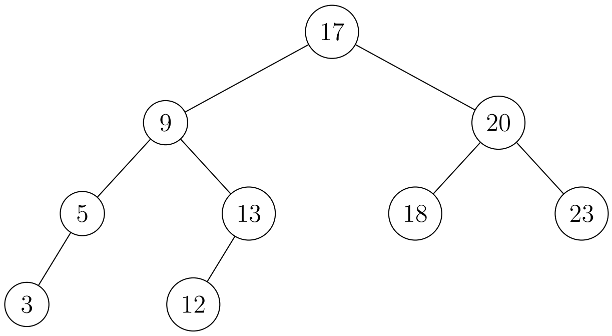
Insert: 12 ☐ l rotation ☐ r rotation ☐ l-r rotation ☐ r-l rotation ☒ no rotation



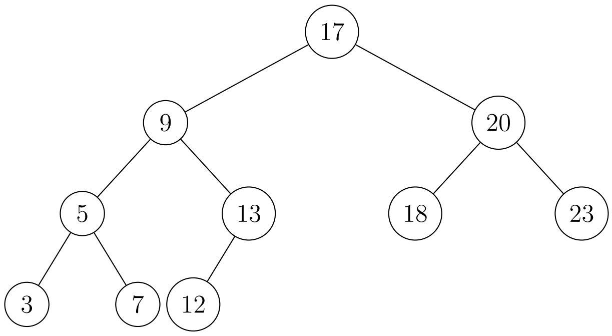
Insert: 23 ☒ l rotation ☐ r rotation ☐ l-r rotation ☐ r-l rotation ☐ no rotation



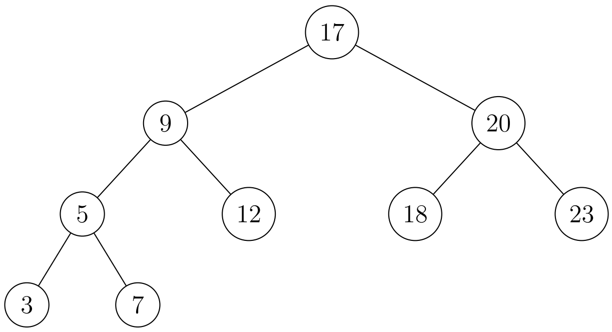
Insert: 3 ☐ l rotation ☐ r rotation ☐ l-r rotation ☐ r-l rotation ☒ no rotation



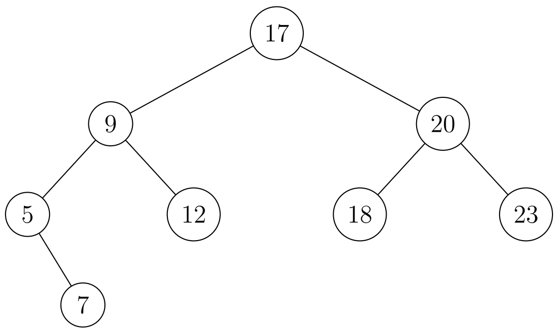
Insert: 7 ☐ l rotation ☐ r rotation ☐ l-r rotation ☐ r-l rotation ☒ no rotation



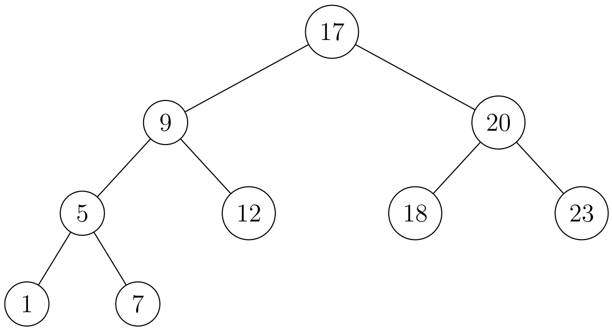
Delete: 13 ☐ l rotation ☐ r rotation ☐ l-r rotation ☐ r-l rotation ☒ no rotation



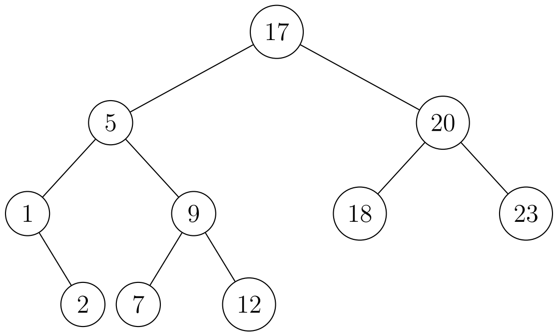
Delete: 3 ☐ l rotation ☐ r rotation ☐ l-r rotation ☐ r-l rotation ☒ no rotation



Insert: 1 ☐ l rotation ☐ r rotation ☐ l-r rotation ☐ r-l rotation ☒ no rotation



Insert: 2 ☐ l rotation ☒ r rotation ☐ l-r rotation ☐ r-l rotation ☐ no rotation

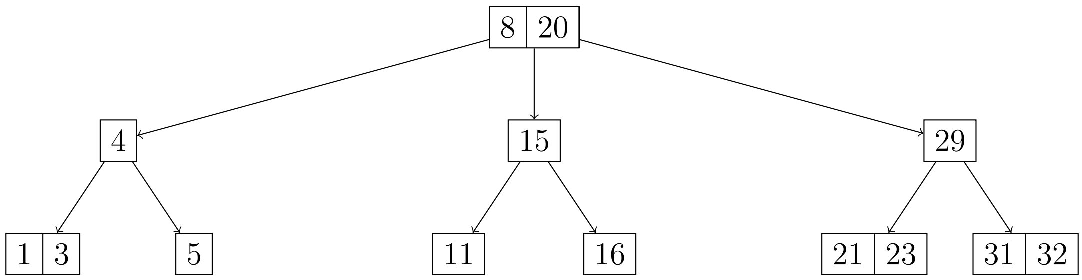


Now give the Pre-Order visitation sequence of the AVL-Tree:

17, 5, 1, 2, 9, 7, 12, 20, 18, 23

(a,b) Trees

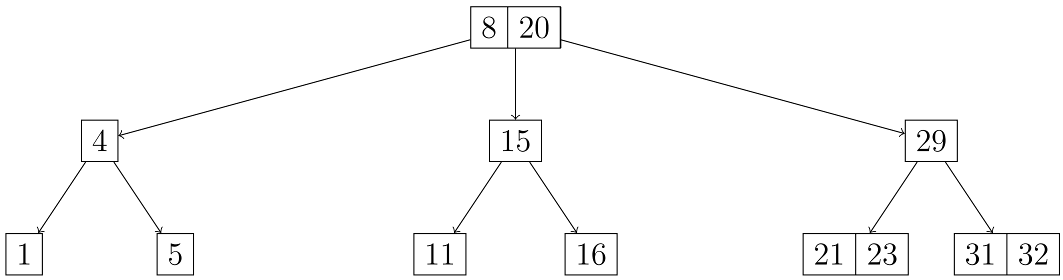
Given the following (a,b) Tree with a = 2 and b = 3:



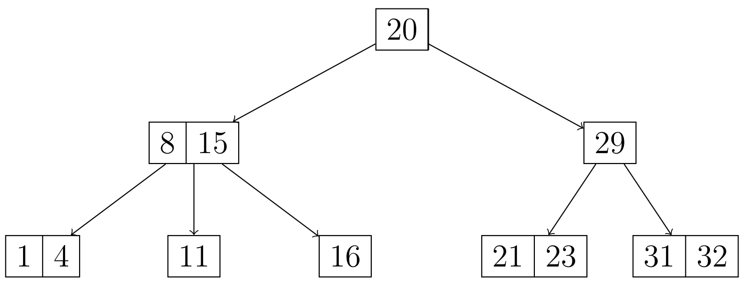
Perform the following operations in their given order on the tree above:

Delete: 3, 5, 31, 4, 20
Insert: 34, 18, 10, 30, 26

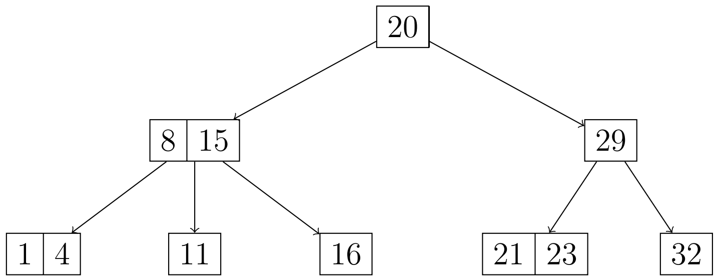
Delete: 3



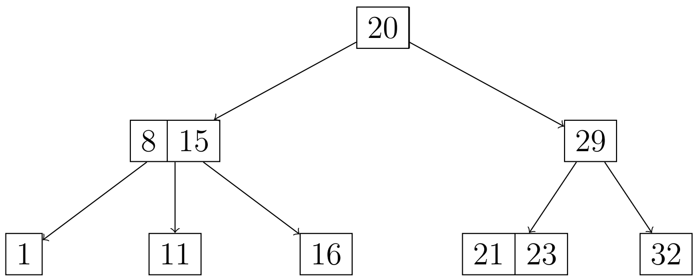
Delete: 5



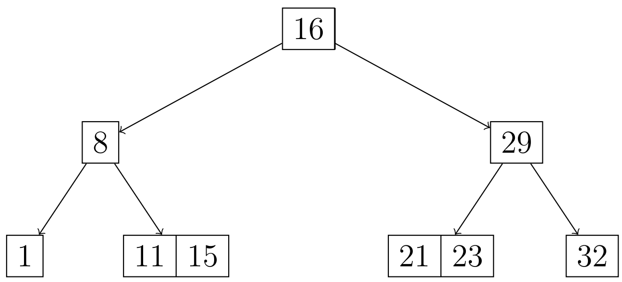
Delete: 31



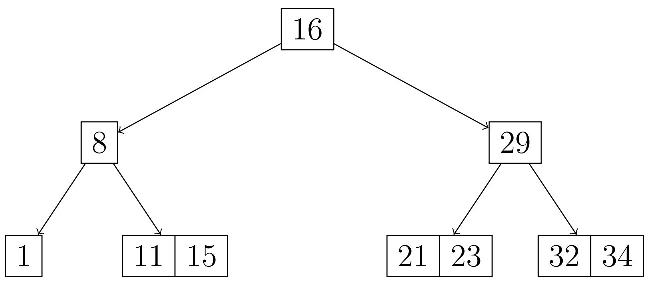
Delete: 4



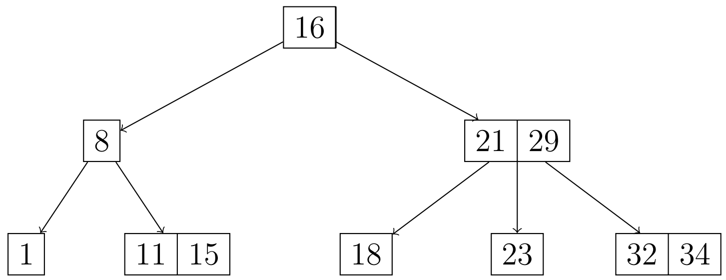
Delete: 20



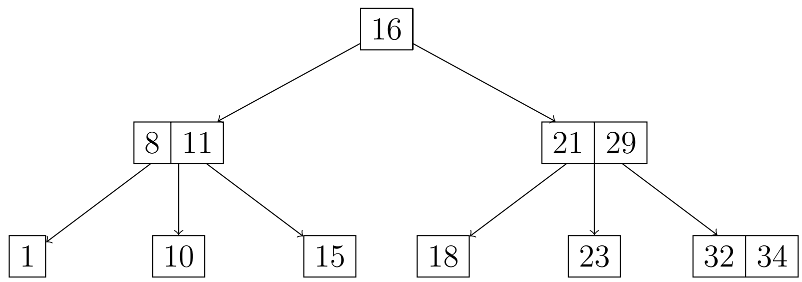
Insert: 34



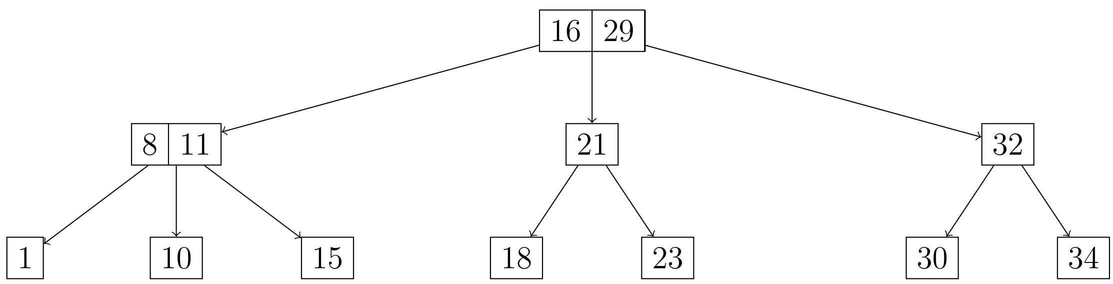
Insert: 18



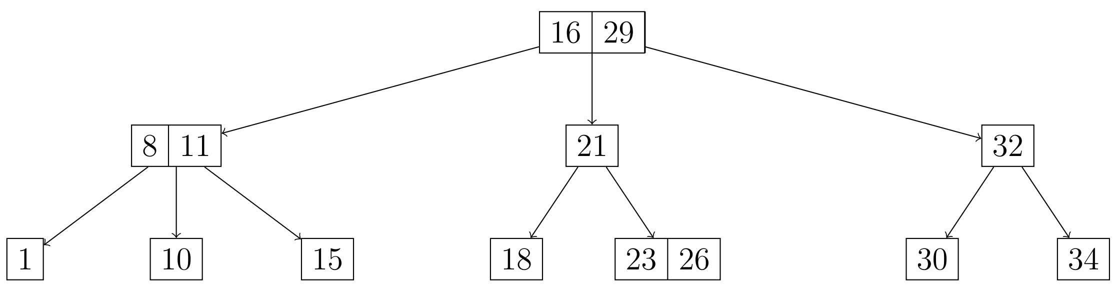
Insert: 10



Insert: 30

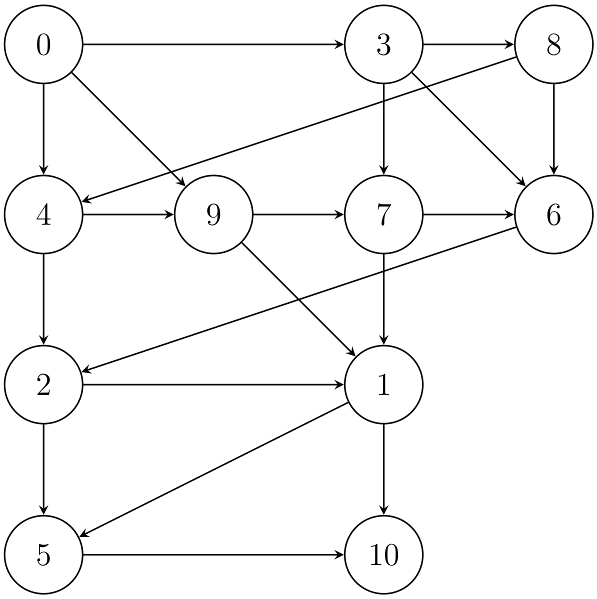


Insert: 26



Graph Traversal (DFS & BFS)

Given the following directed graph:



a) In which order will the nodes in the graph be visited when traversing the graph with Breadth-First Search?
Start at node 0.

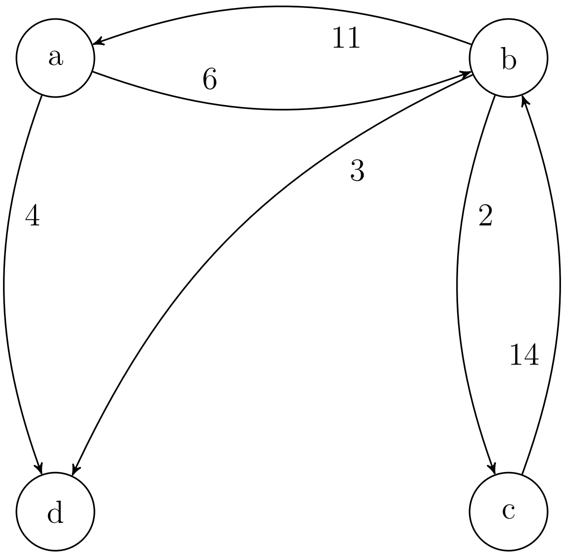
0, 3, 4, 9, 6, 7, 8, 2, 1, 5, 10

b) In which order will the nodes in the graph be visited when traversing the graph with Depth-First Search?
Start at node 0 again.

0, 3, 6, 2, 1, 5, 10, 7, 8, 4, 9

APSP / Floyd-Warshall Algorithm

Given the following directed graph:



Follow the Floyd-Warshall algorithm for APSP and enter the node distance matrix after each step.
Assume the algorithm continues with the next element lexicographically.
As always, unknown distances should be abstracted as infinity.

Initial Matrix:

	a	b	c	d
a	0	6	∞	4
b	11	0	2	3
c	∞	14	0	∞
d	∞	∞	∞	0

Matrix for k = a:

	a	b	c	d
a	0	6	∞	4
b	11	0	2	3
c	∞	14	0	∞
d	∞	∞	∞	0

Matrix for k = b:

	a	b	c	d
a	0	6	8	4
b	11	0	2	3
c	25	14	0	17
d	∞	∞	∞	0

Matrix for k = c:

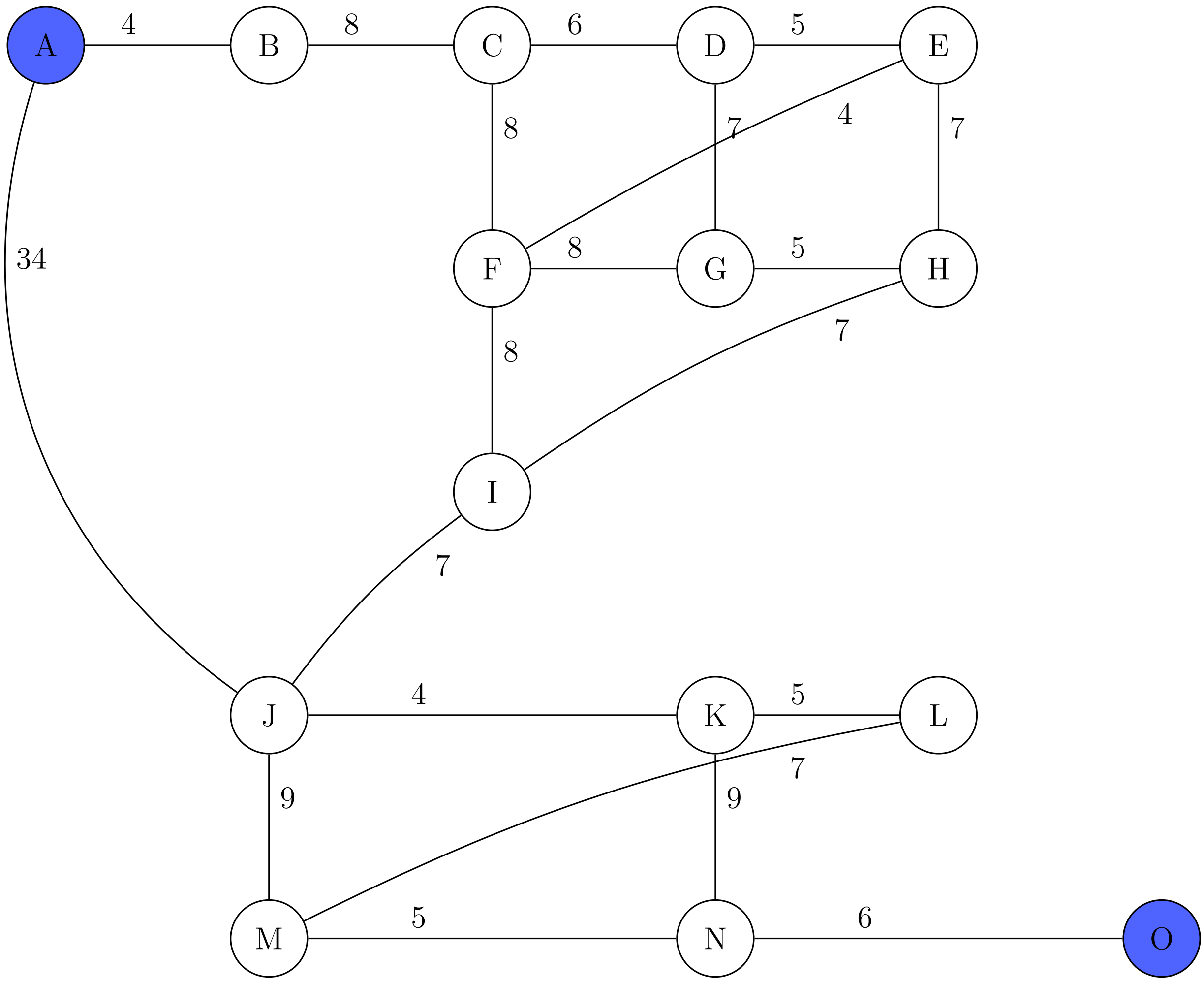
	a	b	c	d
a	0	6	8	4
b	11	0	2	3
c	25	14	0	17
d	∞	∞	∞	0

Matrix for k = d:

	a	b	c	d
a	0	6	8	4
b	11	0	2	3
c	25	14	0	17
d	∞	∞	∞	0

Dijkstra's Algorithm

Given the following graph:



a) Follow the Algorithm of Dijkstra as covered in the lecture and find the shortest path from node A to node O.
Note the algorithms priority queue after every step, as well as the changes made to the queue in the step.
If nodes have equal priorities, the algorithm follows the node that was inserted first.

Priority Queue	Updates in Queue
(B, 4), (J, 34)	(B, 4), (J, 34)
(C, 12), (J, 34)	(C, 12)
(D, 18), (F, 20), (J, 34)	(D, 18), (F, 20)
(F, 20), (E, 23), (G, 25), (J, 34)	(E, 23), (G, 25)
(E, 23), (G, 25), (I, 28), (J, 34)	(I, 28)
(G, 25), (I, 28), (H, 30), (J, 34)	(H, 30)
(I, 28), (H, 30), (J, 34)	—
(H, 30), (J, 34)	—
(J, 34)	—
(K, 38), (M, 43)	(K, 38), (M, 43)
(M, 43), (L, 43), (N, 47)	(L, 43), (N, 47)
(L, 43), (N, 47)	—
(N, 47)	—
(O, 53)	(O, 53)

b) What is the shortest path between the nodes and how long is it?

A →J→K→N→O

Length: 53