HUMBOLDT-UNIVERSITÄT ZU BERLIN

Algorithms and Data Structures

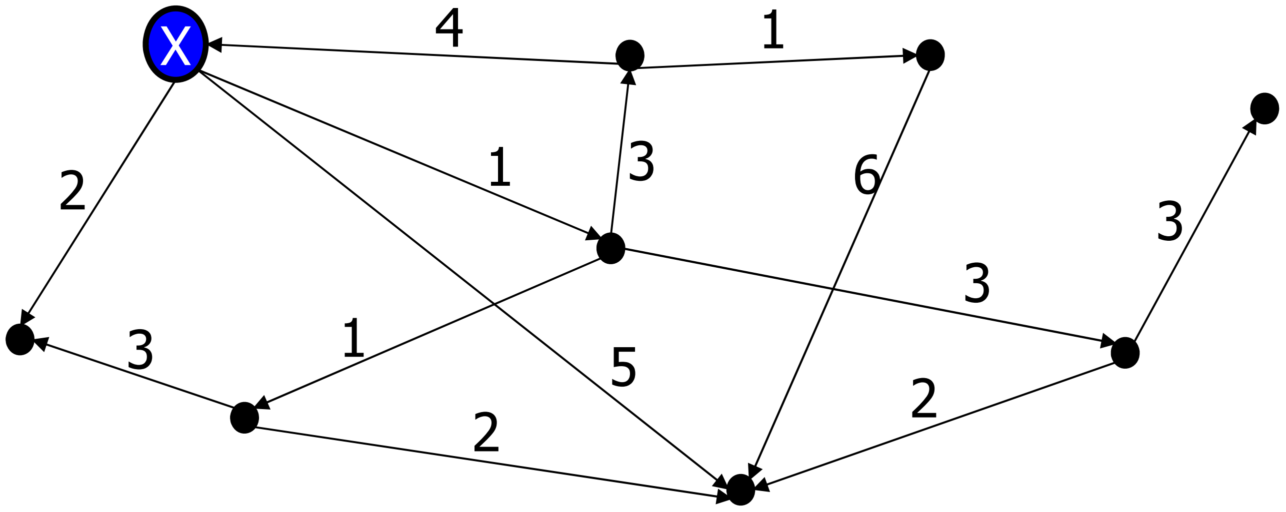
Priority Queues

Marius Kloft

Special Scenarios for Searching

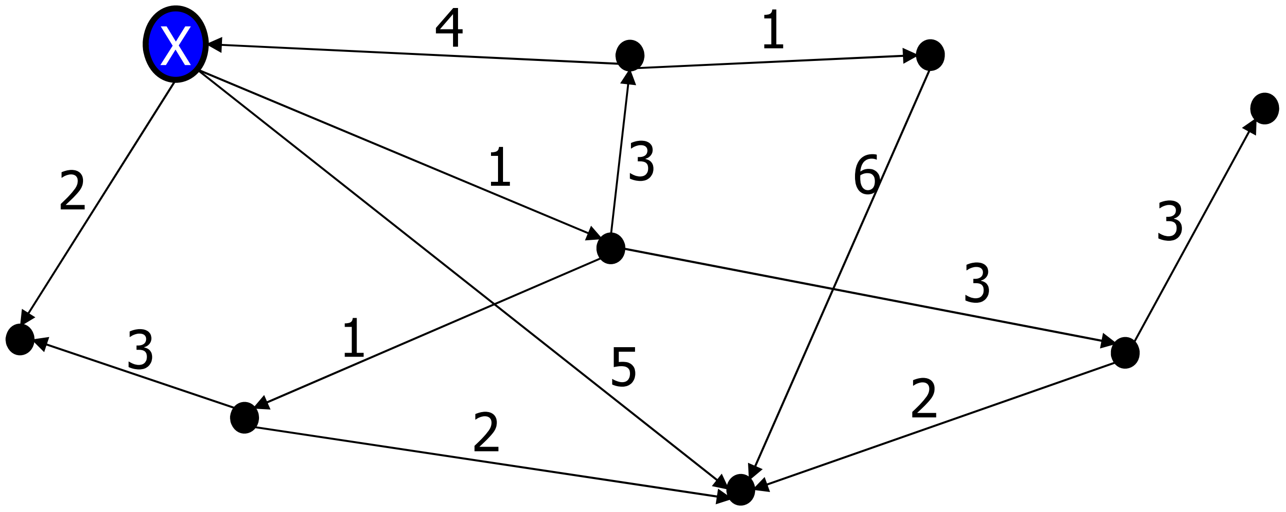
- Up to now, we assumed that all elements of a list are equally important and that any of them could be searched next (with varying probability)
- What if some elements are more important than others?
  - There is a (maybe partial) order on list elements
  - Most important elements are always (not mostly) retrieved next
  - Priority Queues
- Difference to Self-Organizing Lists
  - Most important element is always retrieved next – should be O(1)
  - List should be kept ordered by importance
  - We look at a scenario where new elements are inserted all the time and the most important element is removed regularly

Shortest Paths in a Graph



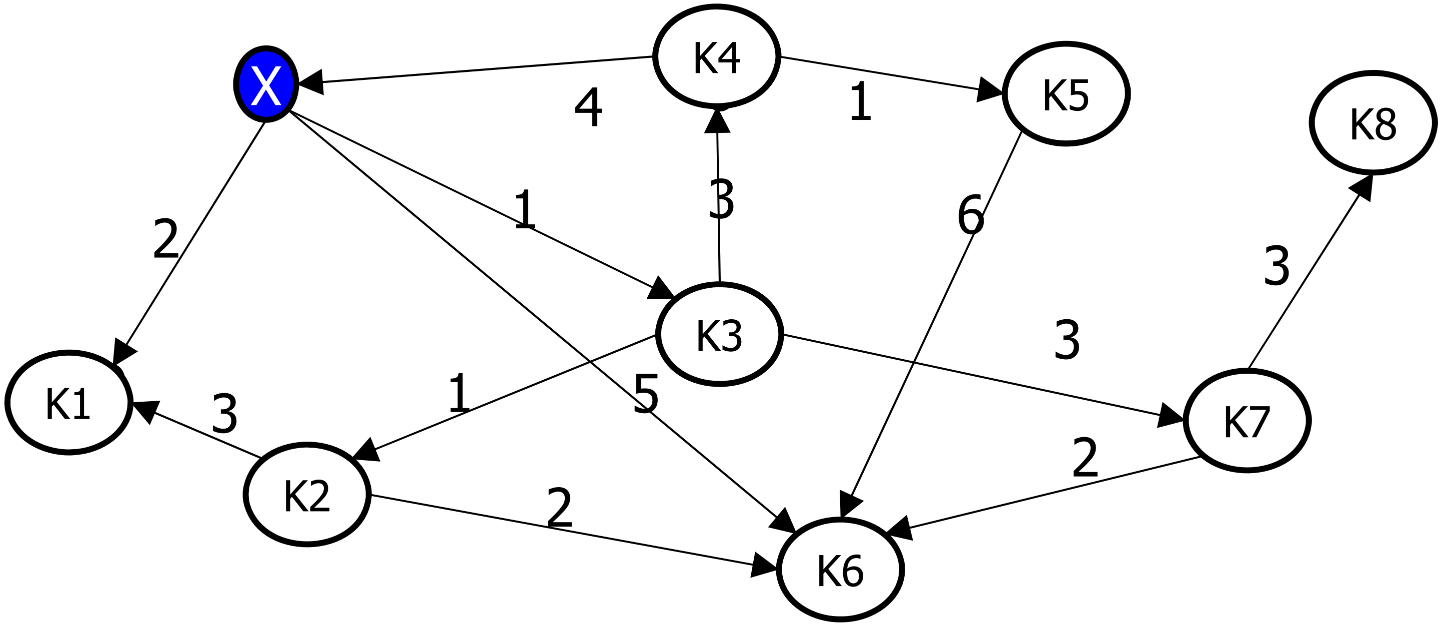
- Task: Find the distance between X and all other nodes
  - Classical problem: Single-Source-Shortest-Paths
  - Famous solution: Dijkstra’s algorithm

Assumptions



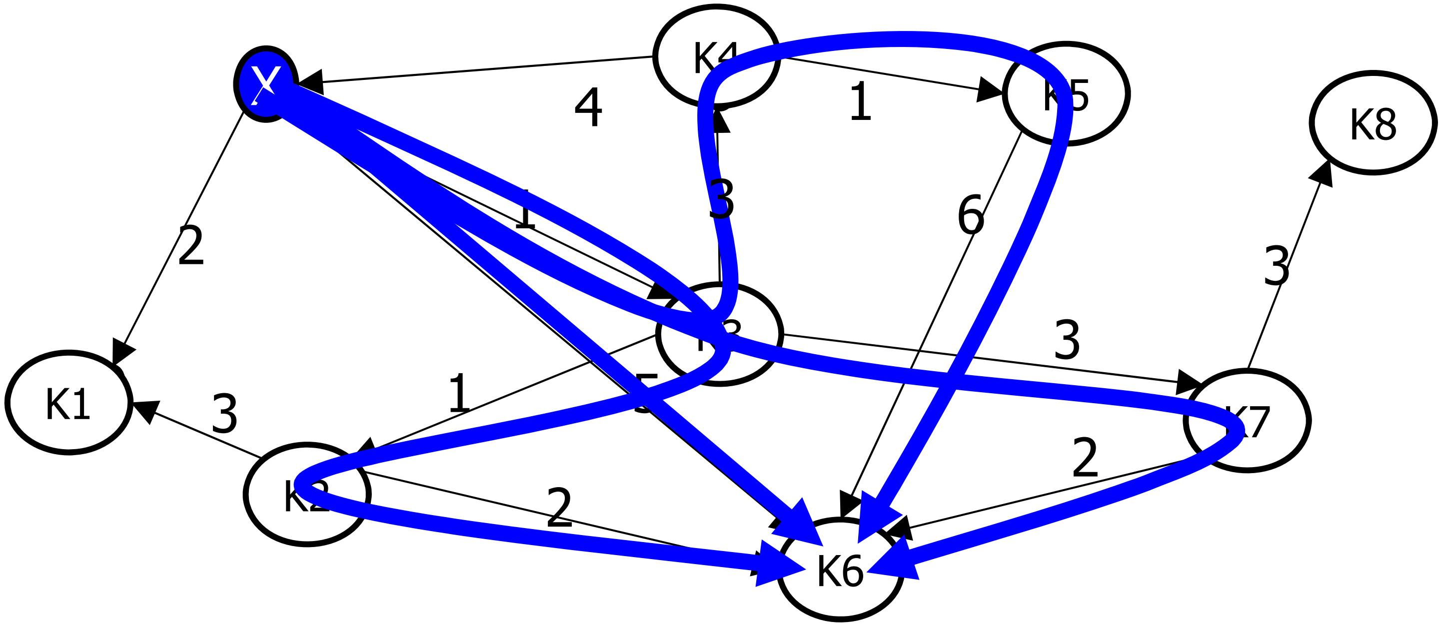
- We assume that there is at least one path between X and any other node (every node is reachable from X)
- We assume strictly positive edge weights
- Distance is the length (=sum of weights) of the shortest path
- There might be many shortest paths, but distance is unique
- We only want the distances and need no "witness paths"

Exhaustive Solution



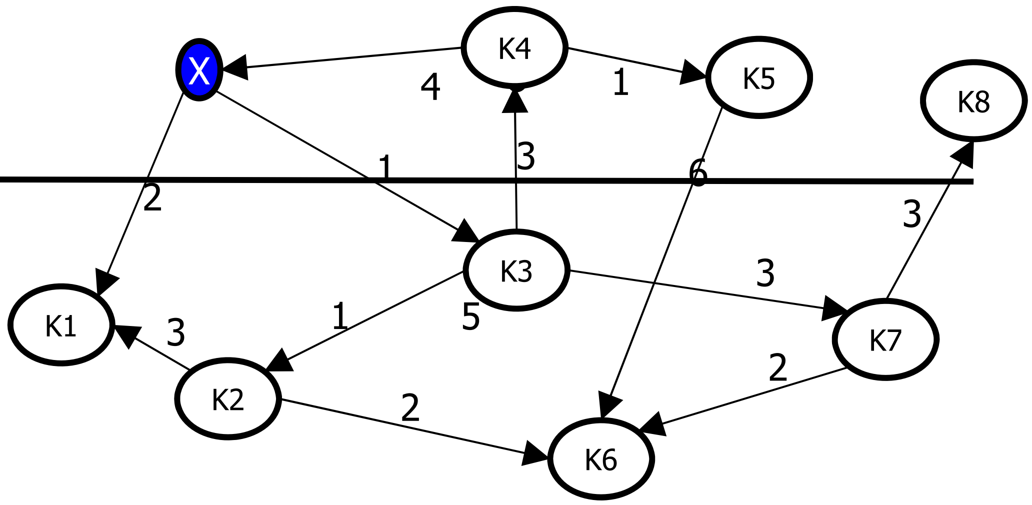
- First approach: Enumerate all paths
  - Need to break cycles (e.g. X – K3 – K4 – X – K3 - …)
  - Using DFS: X – K3 – K4 – X [BT-K4] – K5 – K6 [BT-K5] [BT-K4] [BT-K3] – K7 – K8 [BT-K7] – K6 [BT-K7] [BT-K3] – K2 – K6 [BT-K2] – K1 [BT-K2] [BT-K3] [BT-X] K6 - …

Redundant work



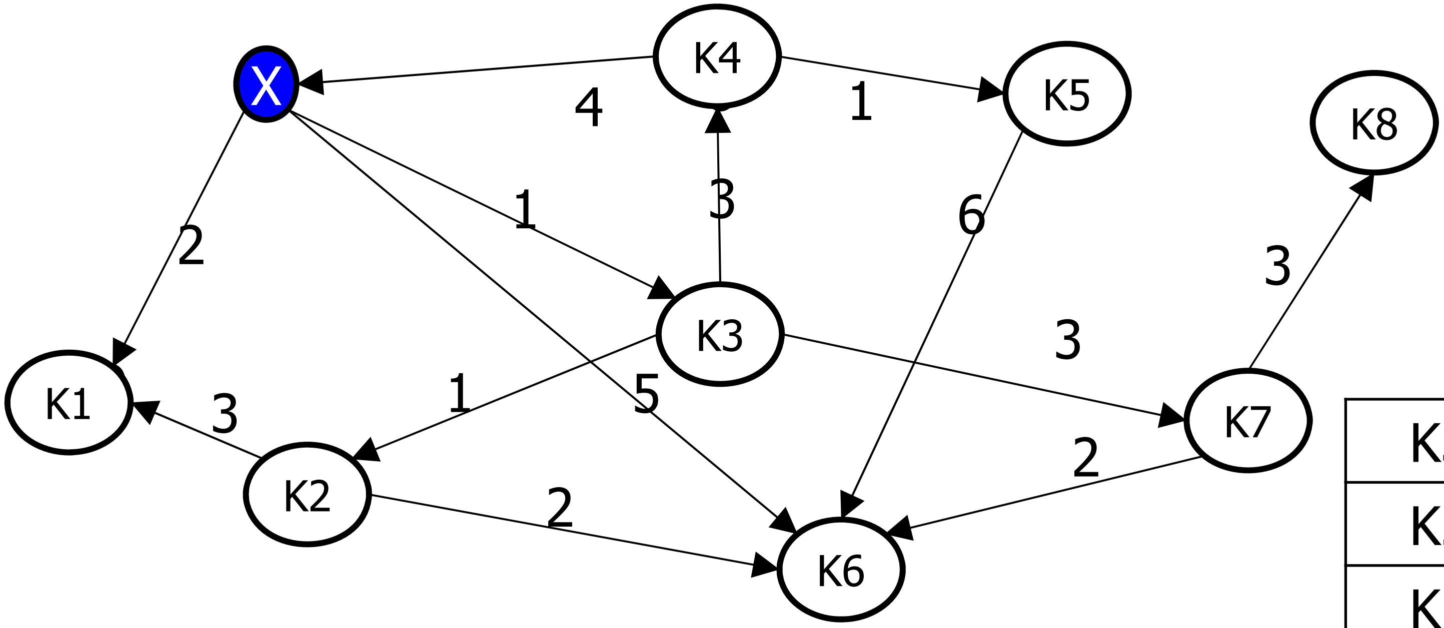
- First approach: Enumerate all paths
  - Need to break cycles (e.g. X – K3 – K4 – X – K3 - …)
  - Using DFS: X – K3 – K4 – X [BT-K4] – K5 – K6 [BT-K5] [BT-K4] [BT-K3] – K7 – K8 [BT-K7] – K6 [BT-K7] [BT-K3] – K2 – K6 [BT-K2] – K1 [BT-K2] [BT-K3] [BT-X] K6 - …

Dijkstra’s Idea



- Enumerate paths from X by their length
  - Neither DFS nor BFS
- Assume we reach a node Y by a path p of length l and we have already explored all paths from X with length l’ ≤ l and that Y was not reached yet
- Then p must be a shortest path between X and Y
  - Because any p’ between X and Y would have a prefix of length at least l and (a) a continuation with length>0 or (b) would not need a continuation (then p is as short as p’)

Example for Idea



- 1: X – K3
- 2: X – K3 – K2
- 2: X – K1
- 4: X – K3 – K2 – K6
- 4: X – K3 – K4
- 4: X – K3 – K7

- 5: X – K3 – K4 – K5
- 7: X – K3 – K7 – K8
- Stop (all nodes found)

K3	1
K2	2
K1	2
K6	4
K4	4
K7	4
K5	5
K8	7

A Further Trick

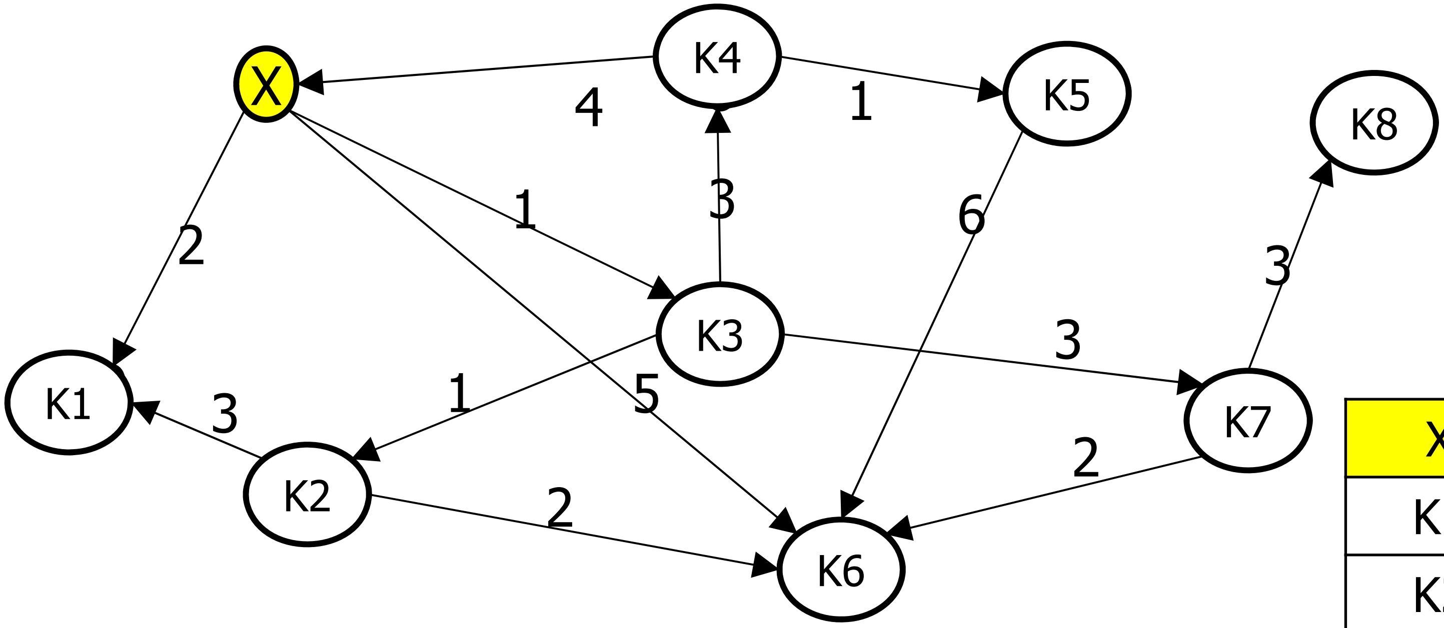
- Enumerate paths by iteratively extending short paths by all possible extensions
  - All edges outgoing from the end node of a short path
- These extensions
  - … either lead to a node which we didn't reach before – then we found a path, but cannot yet be sure it is the shortest
  - … or lead to a node which we already reached but we are not yet sure of we found the shortest path to it – update current best distance
  - … or lead to a node which we already reached and for which we also surely found a shortest path already – these can be ignored
- Eventually, we enumerate nodes by their distance

Algorithm

```
G = (V, E);
x : start_node;     # x∈V
A : array_of_distances;
∀i: A[i]:= ∞;
L := V;
A[x] := 0;
while L≠∅
  k := L.get_closest_node();
  L := L \ k;
  forall (k,f,w)∈E do
    if f∈L then
      new_dist := A[k]+w;
      if new_dist < A[f] then
        A[f] := new_dist;
      end if;
    end if;
  end for;
end while;
```

- Assumptions
  - Nodes have IDs between 1 … |V|
  - Edges are `(from, to, weight)`
- We enumerate nodes by length of their shortest paths
  - In the first loop, we pick x and update distances (A) to all adjacent nodes
  - When we pick a node k, we already have computed its distance to x in A
  - We adapt the current best distances to all neighbors of k we haven't picked yet
- Once we picked all nodes, we are done

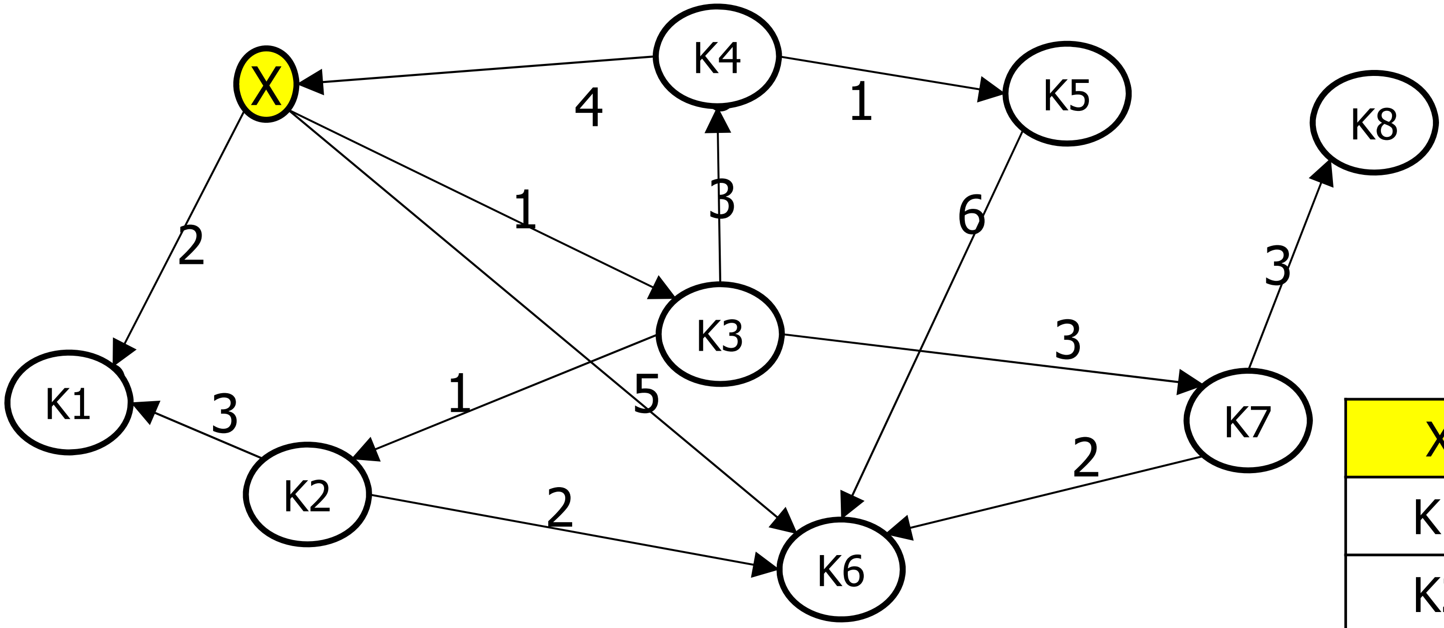
Example for Algorithm



X	0
K1	∞
K2	∞
K3	∞
K4	∞
K5	∞
K6	∞
K7	∞
K8	∞

- Pick x

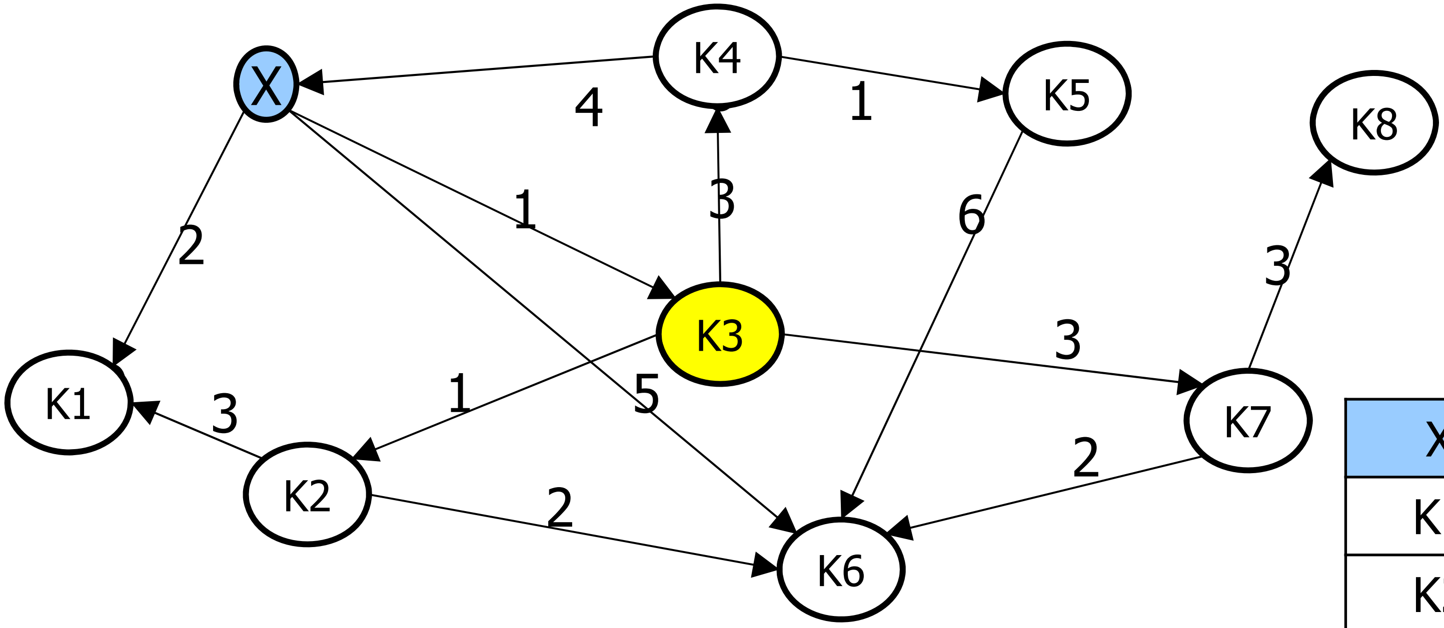
Example for Algorithm



- Pick x
- Adapt distances to all neighbors

X	0
K1	2
K2	∞
K3	1
K4	∞
K5	∞
K6	5
K7	∞
K8	∞

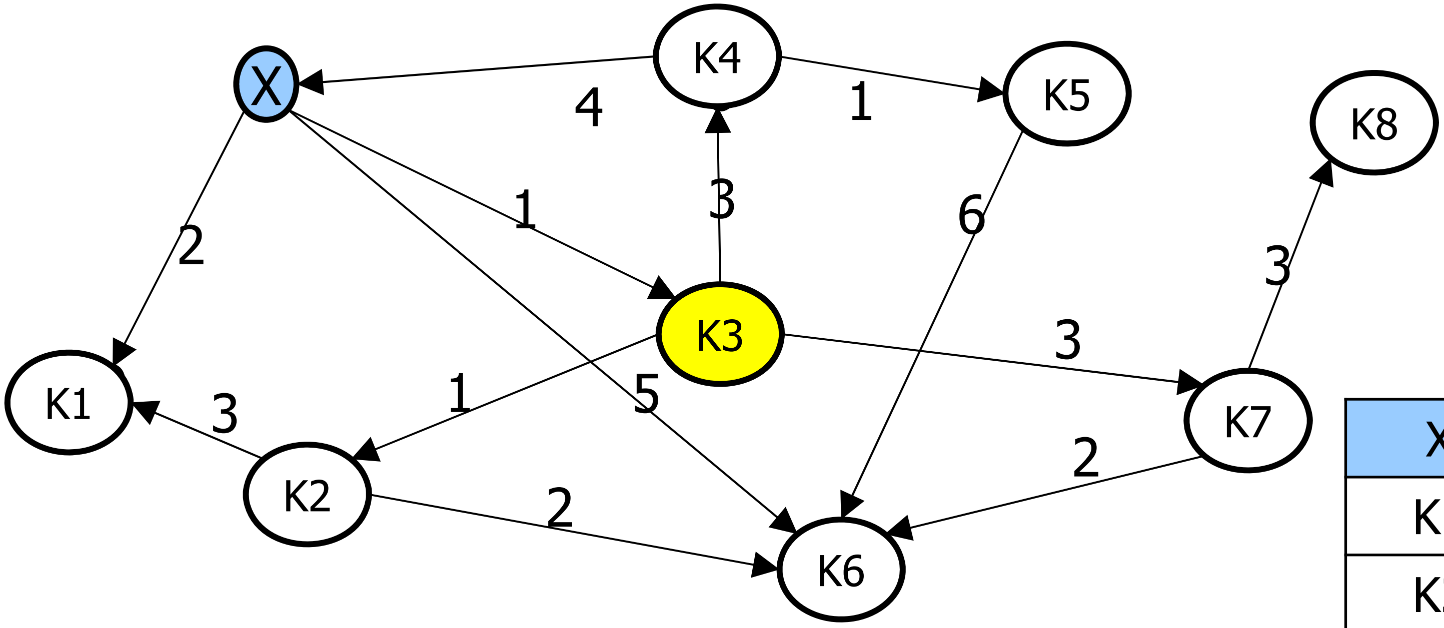
Example for Algorithm



X	0
K1	2
K2	∞
K3	1
K4	∞
K5	∞
K6	5
K7	∞
K8	∞

- Pick K3 (closest to x)

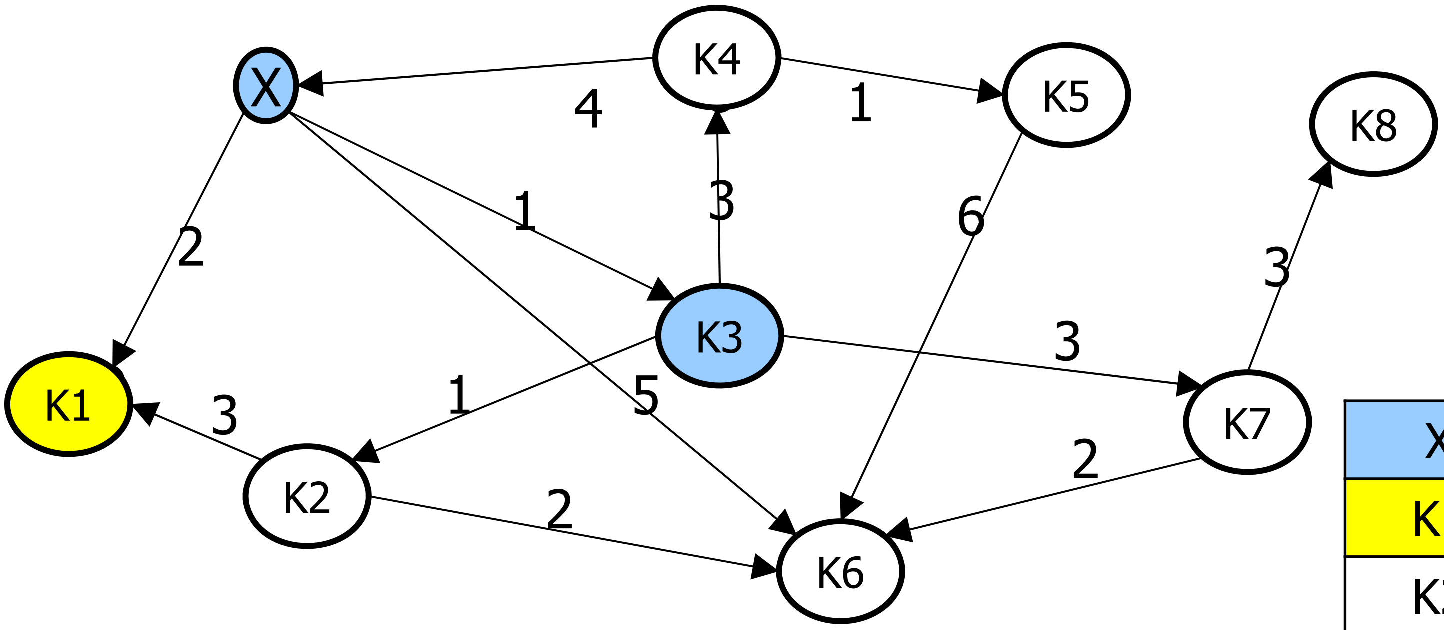
Example for Algorithm



- Pick K3
- Adapt distances (from x) to all neighbors (of K3)

X	0
K1	2
K2	2
K3	1
K4	4
K5	∞
K6	5
K7	4
K8	∞

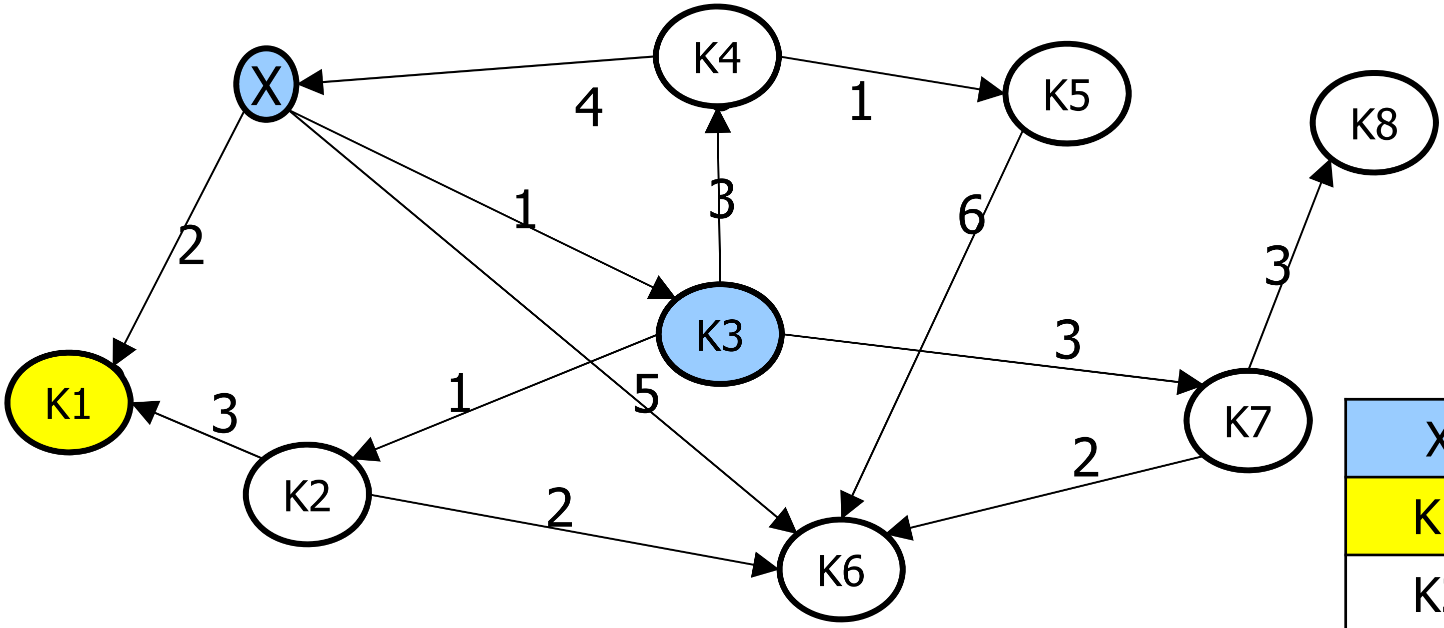
Example for Algorithm



- Pick K1 (or K2)

X	0
K1	2
K2	2
K3	1
K4	4
K5	∞
K6	5
K7	4
K8	∞

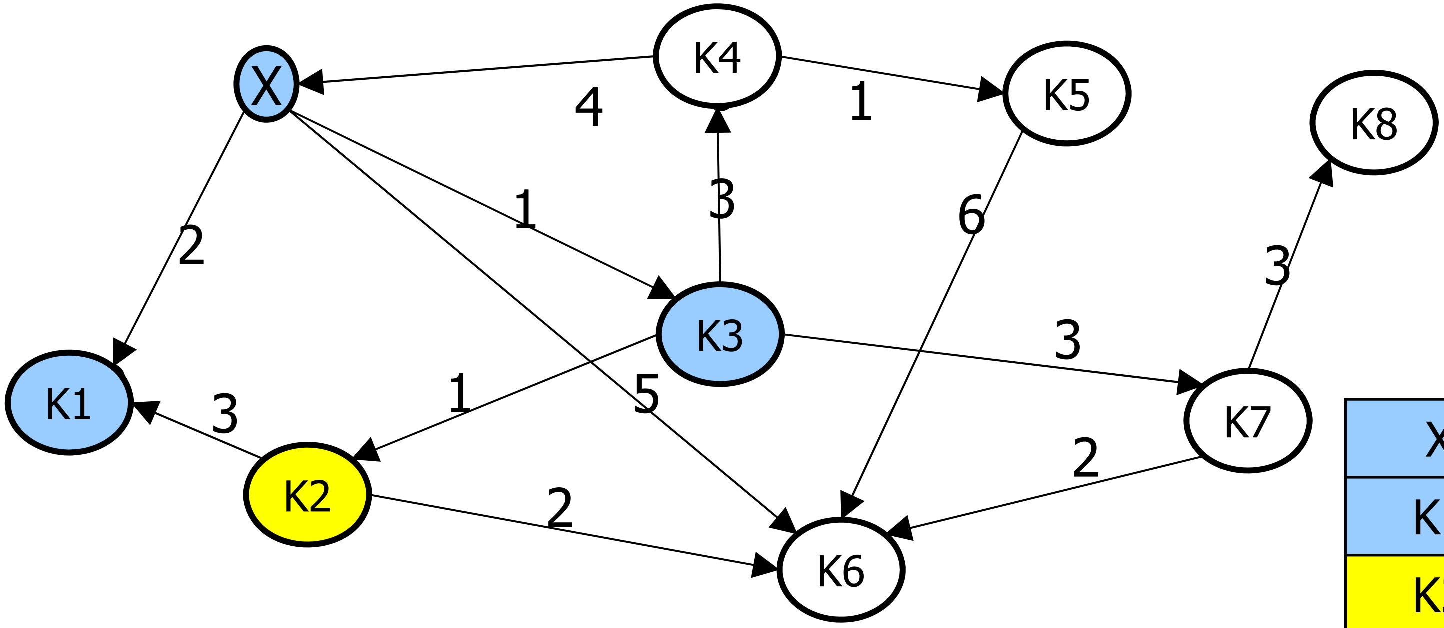
Example for Algorithm



X	0
K1	2
K2	2
K3	1
K4	4
K5	∞
K6	5
K7	4
K8	∞

- Pick K1
- Adapt distances to all neighbors
  - There are none

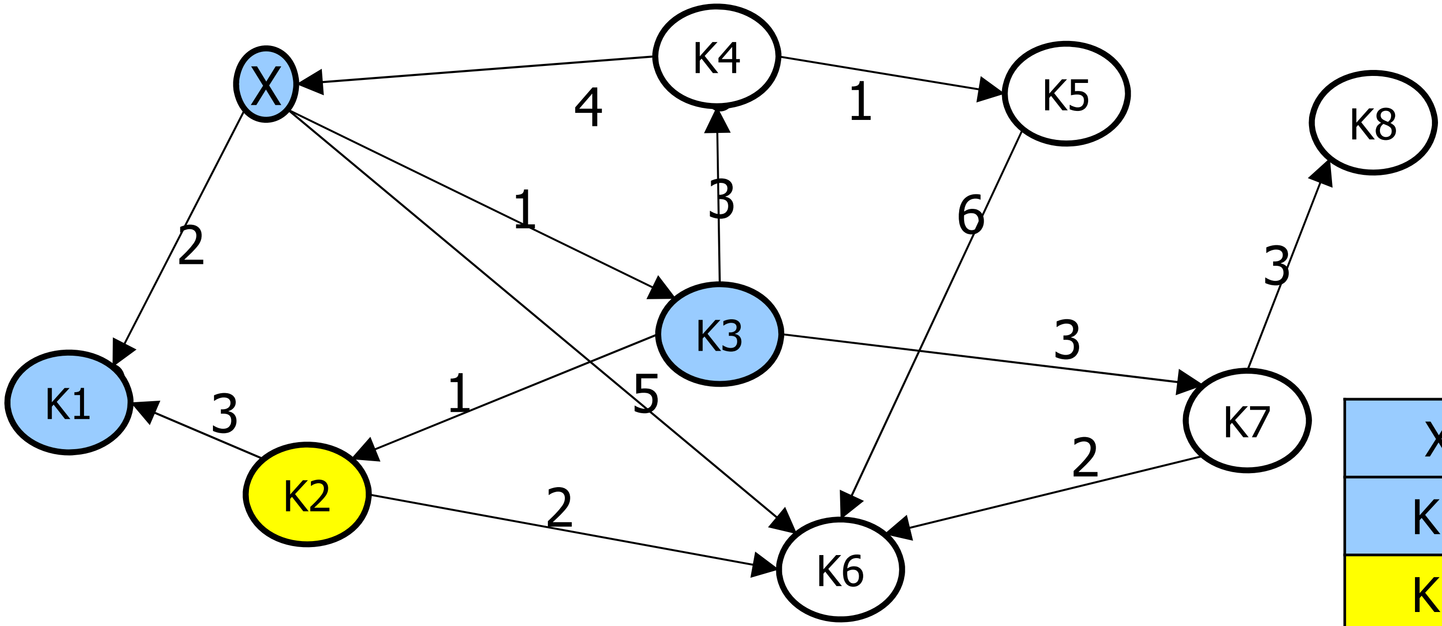
Example for Algorithm



- Pick K2

X	0
K1	2
K2	2
K3	1
K4	4
K5	∞
K6	5
K7	4
K8	∞

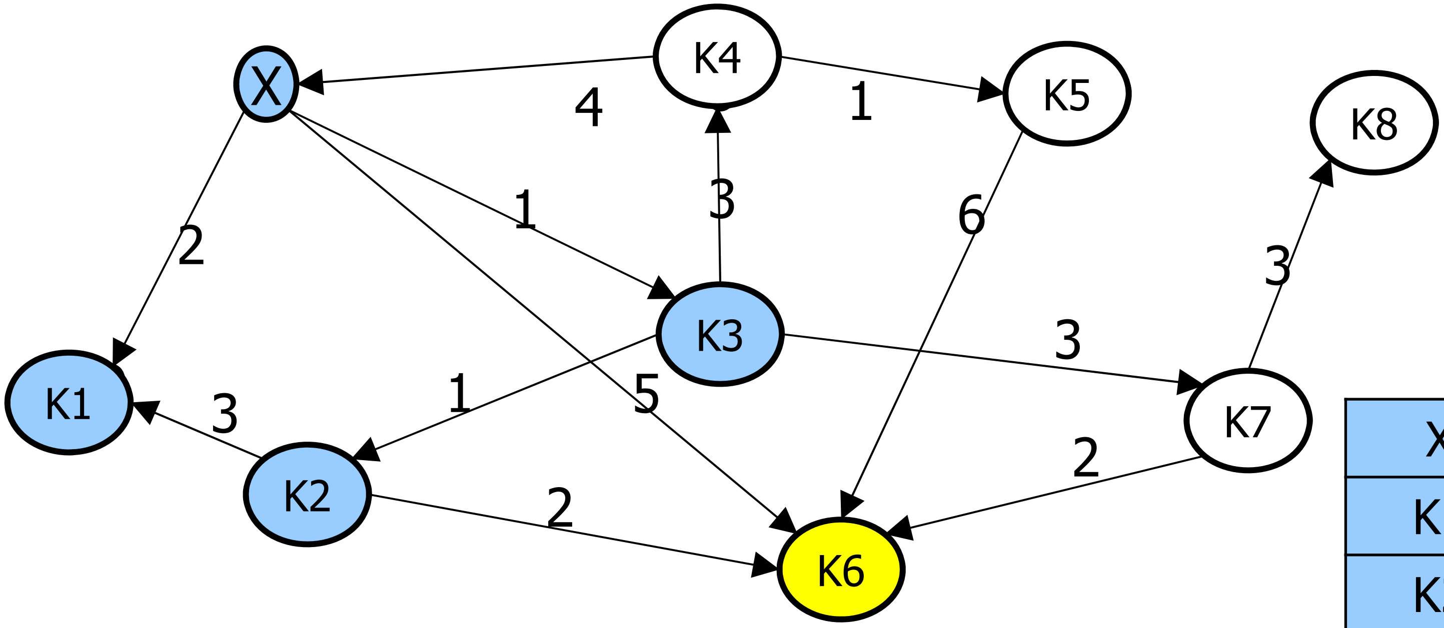
Example for Algorithm



- Pick K2
- Adapt distances to all neighbors
  - K1 was picked already – ignore
  - We found a shorter path to K6

X	0
K1	2
K2	2
K3	1
K4	4
K5	∞
K6	4
K7	4
K8	∞

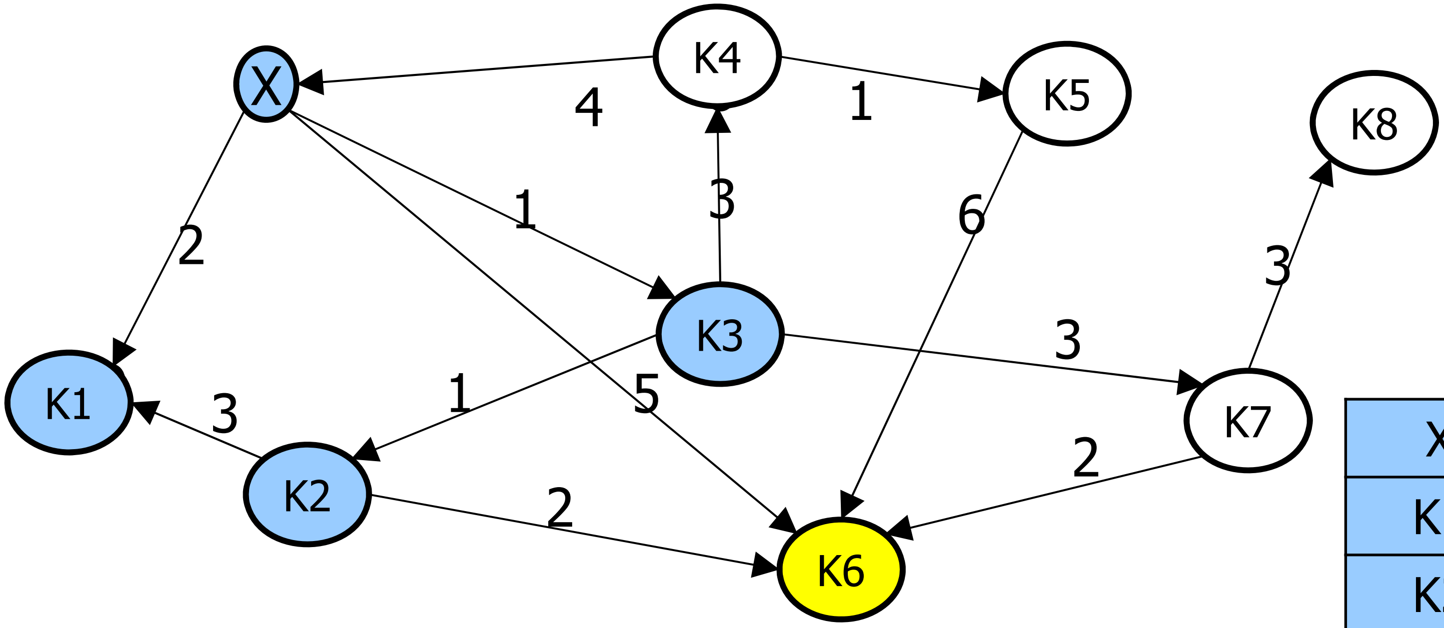
Example for Algorithm



- Pick K6 (or K4 or K7)

X	0
K1	2
K2	2
K3	1
K4	4
K5	∞
K6	4
K7	4
K8	∞

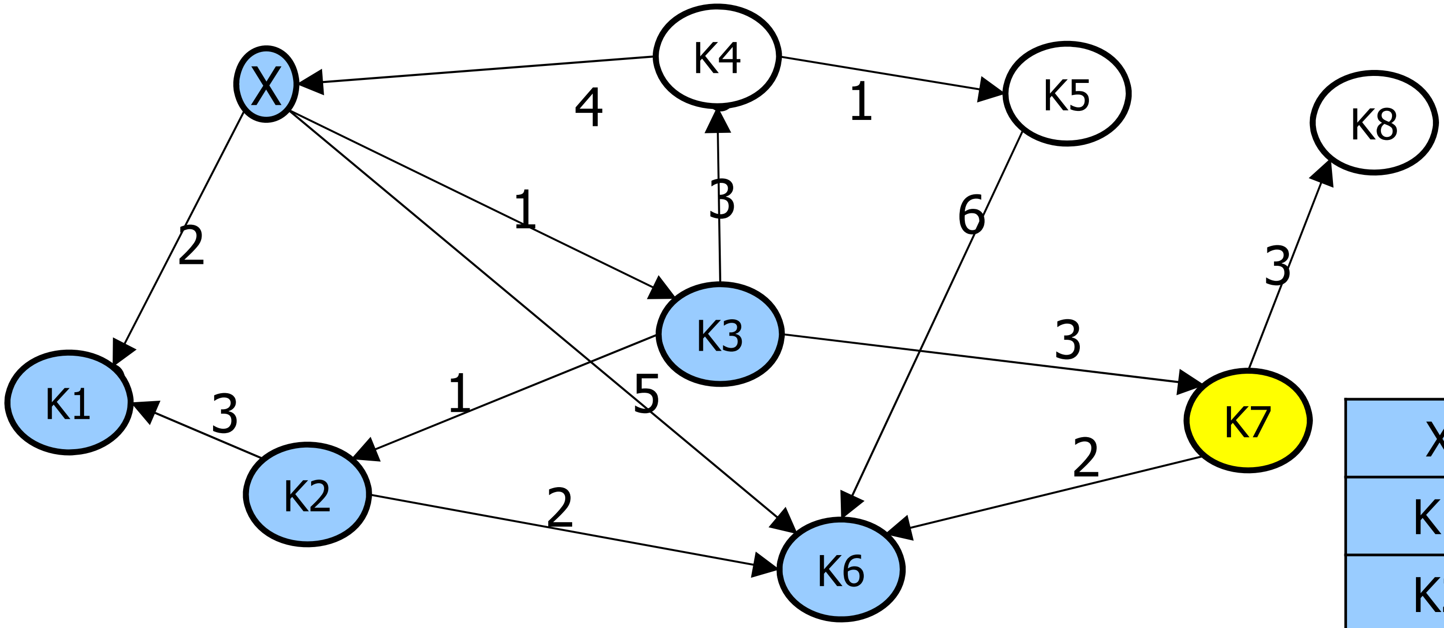
Example for Algorithm



X	0
K1	2
K2	2
K3	1
K4	4
K5	∞
K6	4
K7	4
K8	∞

- Pick K6
- Adapt distances to all neighbors
  - There are none

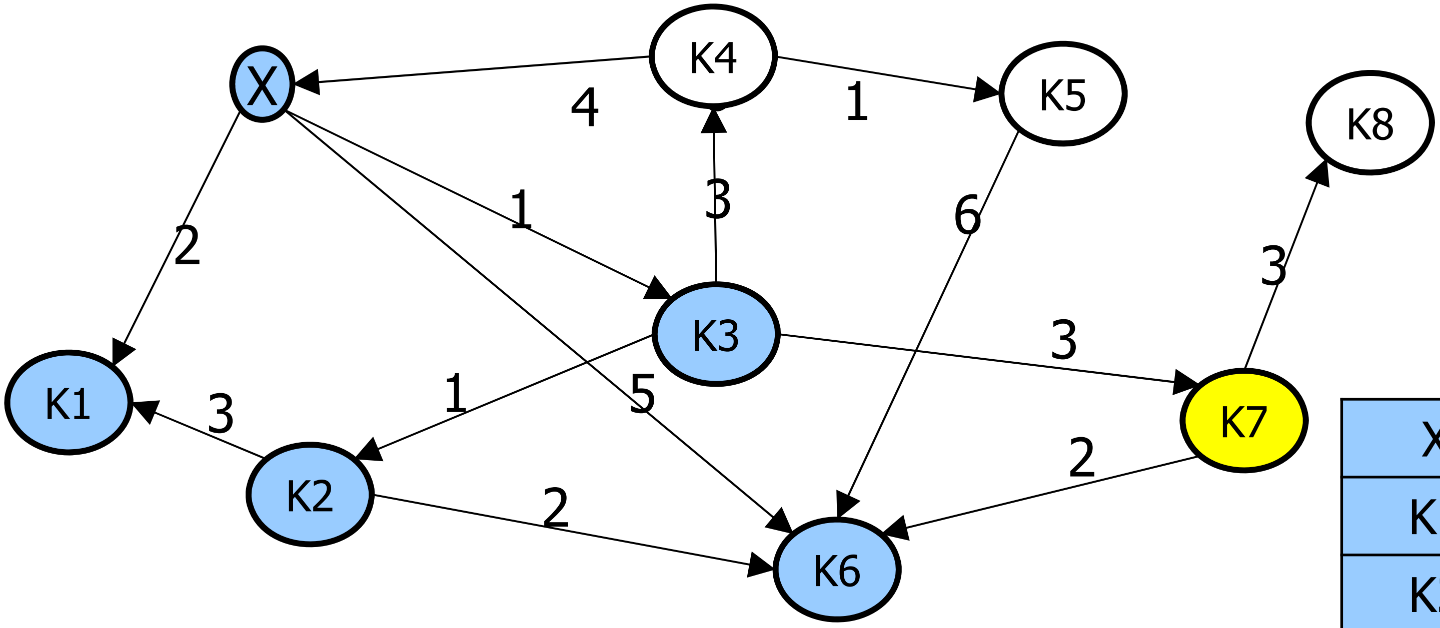
Example for Algorithm



- Pick K7

X	0
K1	2
K2	2
K3	1
K4	4
K5	∞
K6	4
K7	4
K8	∞

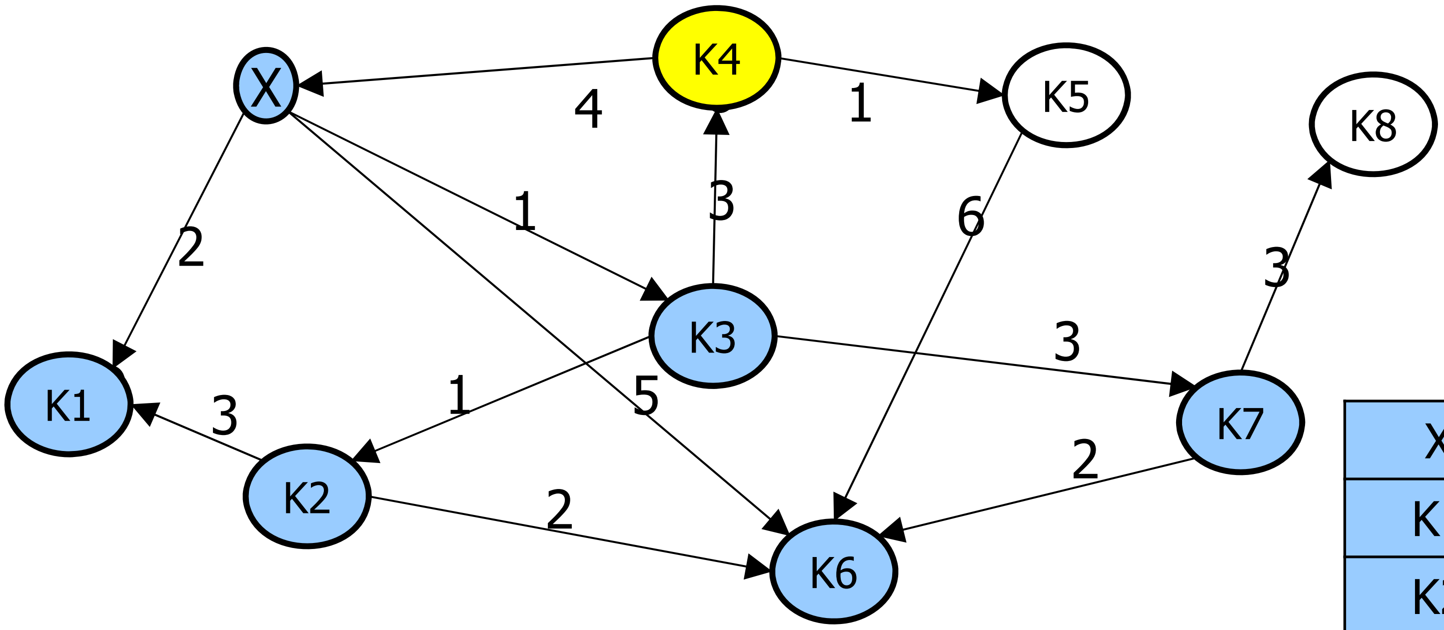
Example for Algorithm



X	0
K1	2
K2	2
K3	1
K4	4
K5	∞
K6	4
K7	4
K8	7

- Pick K7
- Adapt distances to all neighbors
  - K6 was visited already

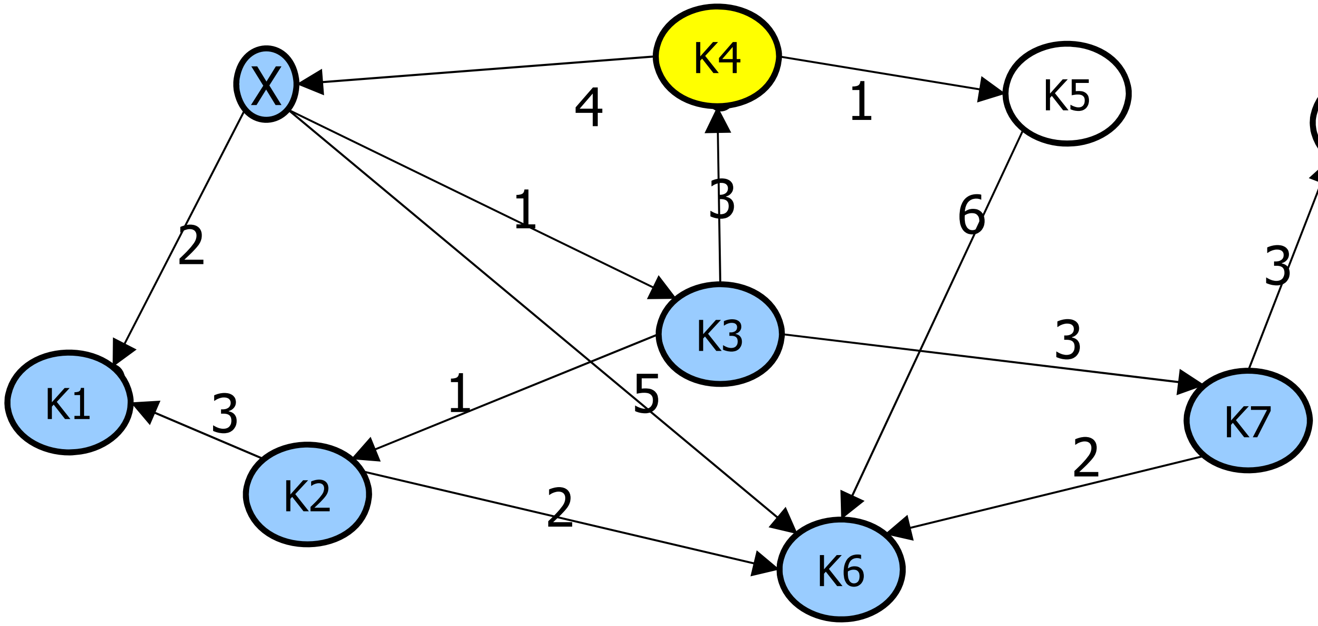
Example for Algorithm



- Pick K4

X	0
K1	2
K2	2
K3	1
K4	4
K5	∞
K6	4
K7	4
K8	7

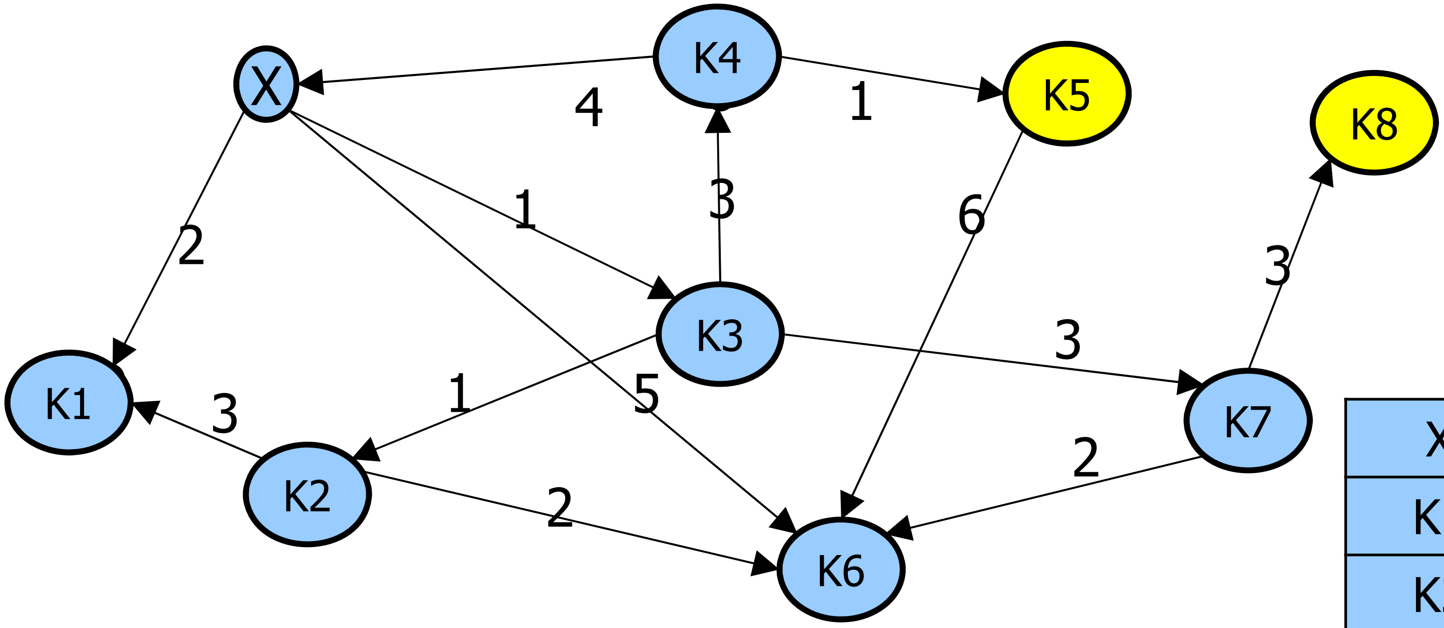
Example for Algorithm



- Pick K4
- Adapt distances to all neighbors
  - X was visited already

X	0
K1	2
K2	2
K3	1
K4	4
K5	5
K6	4
K7	4
K8	7

Example for Algorithm



- Pick K5 … Pick K8

X	0
K1	2
K2	2
K3	1
K4	4
K5	5
K6	4
K7	4
K8	7

A Closer Look

```
G = (V, E);
x : start_node;     # x∈V
A : array_of_distances;
∀i: A[i]:= ∞;
L := V;
A[x] := 0;
while L≠∅
  k := L.get_closest_node();
  L := L \ k;
  forall (k,f,w)∈E do
    if f∈L then
      new_dist := A[k]+w;
      if new_dist < A[f] then
        A[f] := new_dist;
      end if;
    end if;
  end for;
end while;
```

- Algorithm seems to work
  - Proof and analysis will follow later
- Central: `get_closest_node()`
  - Needs to find the node k in L for which A[k] is the smallest
  - A[k] is changed a lot during a run
- Searching A? Resorting A?
- Better: Priority queue
  - List of tuples (o, v) (object,value)
  - Central operation: Return tuple where v is smallest

Content of this Lecture

- Priority Queues
- Using Heaps

Priority Queues

- A priority queue (PQ) is an ADT with 3 essential operations
  - `add(o,v)`: Add element o with value (priority) v
  - `getMin()`: Retrieve element with highest priority
  - `removeMin()`: Remove element with highest priority
- Typical additional operations
  - `merge(p1, p2)`: Merge two PQs into one (properly sorted)
  - `create (L)`: Convert a list in a priority queue
  - `delete(o)`: Delete o from PQ
  - `changeValue(o,v)`: Change value of o to v

Other Applications

- Games (e.g. chess)
  - The machine explores next movements but cannot look at all of them; give each move an assumed benefit and explore moves with probably highest benefit first (see also A* algorithm)
- Multi-modal route planning
  - Find fastest route through a map (network) with multiple ways of transportation (feet, bus, train, …) between edges where edge weights change dynamically (delay, congestion, …)
    - And departure times may depend on arrival: Timetable-based routing
- Quality of Service in a network
  - When bandwidth is limited, sort all transmission requests in a PQ and transmit by highest priority
- …

Naive Implementations (with |Q|=n)

- Using a linked list
  - **`add`** requires O(1) (at the end or start or anywhere)
  - **`getMin`** requires O(n) [bad]
  - **`deleteMin`** requires O(1) (if we keep the pointer after a getMin)
  - **`merge`** requires O(1)
- Using a linked list sorted by priority
  - **`add`** requires O(n) [bad]
  - **`getMin`** requires O(1)
  - **`deleteMin`** requires O(1)
  - **`merge`** requires O(n+m)

Maybe Arrays?

- Using a sorted array
  - `add` requires O(n) [bad - we find the position in log(n), but then have to free a cell by moving all elements after this cell]
  - `getMin` requires O(1)
  - `deleteMin` requires O(n) [bad]
- PQs are typically used in applications where elements are inserted and removed all the time
- We need a DS that can change its size dynamically at very low cost while keeping a certain order (min element)
- We want constant or at most log-time for all operations

Content of this Lecture

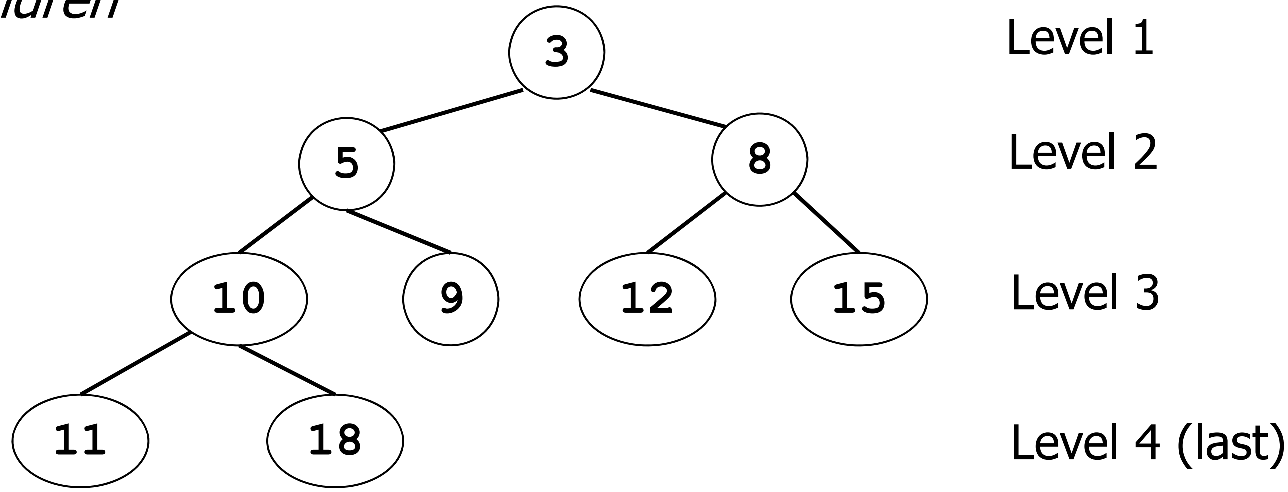
- Priority Queues
- Using Heaps
    - Heaps
    - Operations on Heaps
    - Heap Sort

Heap-based PQ

- Unsorted lists require O(n) for **`getMin()`**
  - We don't know where the smallest element is
- Sorted lists require O(n) for **`add()`**
  - We don't know where to put the new element
- Can we find a way to keep the list "a little sorted"?
  - Actually, we only need the smallest element at a fixed position
  - All other elements can be at arbitrary places
  - **`add() / deleteMin()`** could be faster than O(n), if they don't need to keep the entire list sorted
- One such structure is called a heap

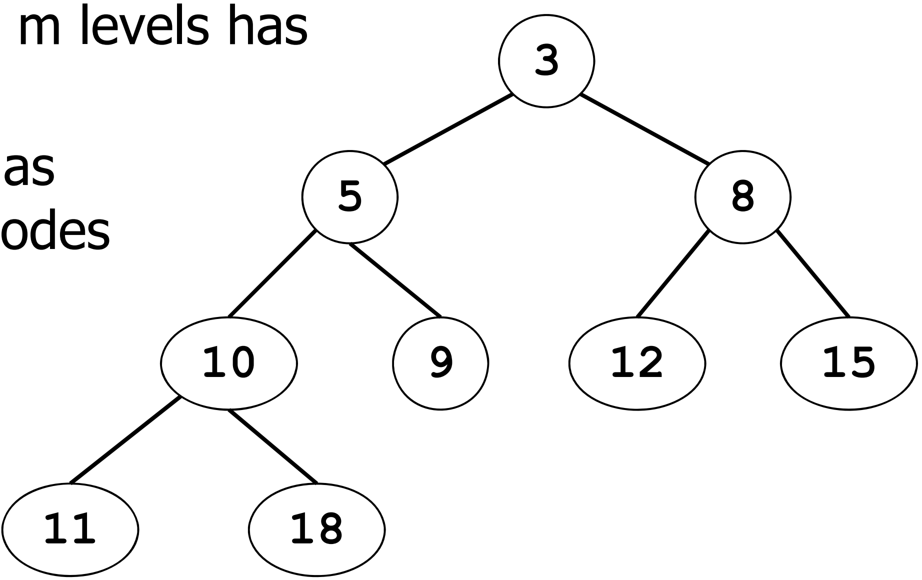
Heaps

- Definition
  A heap is a labeled binary tree for which the following holds
  - *Form-constraint (FC): The tree is complete except the last level*
    - *I.e.: Every node at level l<d-1 has exactly two children*
    - *The last level is filled from left to right*
  - *Heap-constraint (HC): The label of any node is smaller than that of its children*



Properties

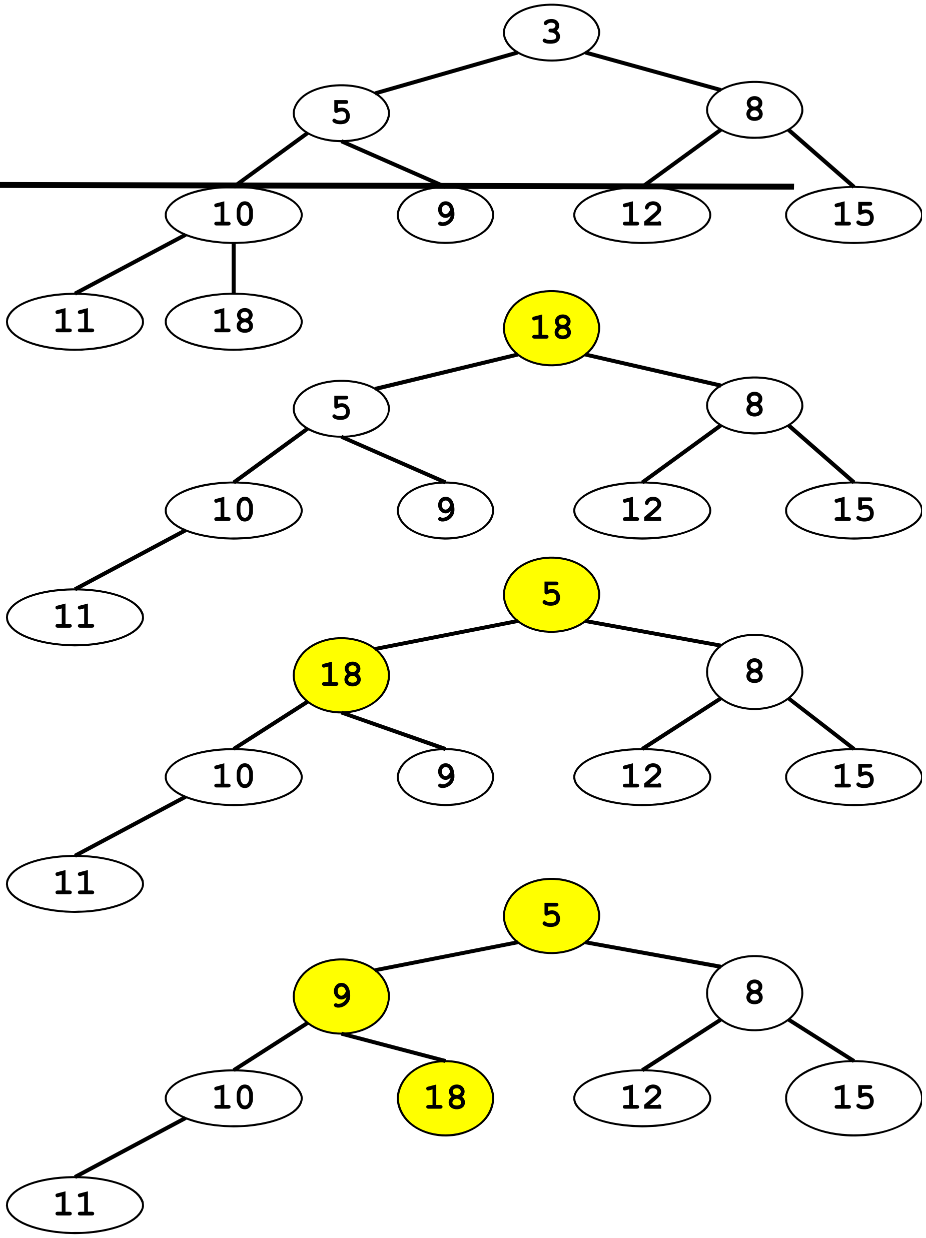
- Order
  - A heap is "a little" sorted: We know the smallest element (root)
  - We know the order for some pairs of elements (parent-child), but for many pairs we don't know which is bigger (e.g. nodes in the same level)
- Size
  - A complete binary tree with m levels has 2^m-1 nodes
  - A heap with m levels thus has between 2^{m-1}+1 and 2^m-1 nodes
  - A heap with n nodes has ceil(log(n+1)) levels



Operations

- Assume we store our PQ as a heap
- Clearly, `getMin()` is possible in O(1)
  - Keep a pointer to the root
- But …
  - How can we perform `deleteMin()` – such that the new structure again is a heap?
  - How can we add an element to a heap – such that the new structure again is a heap?
  - How can we turn a list into a heap?

DeleteMin()



- We first remove the root
  - Creates two heaps
  - We must connect them again
- We take the „last“ node, place it in root, and “sift” it down the tree
  - Last node: right-most in the last level
  - Sifting down: Exchange with smaller of both children as long as at least one child is smaller than the node itself

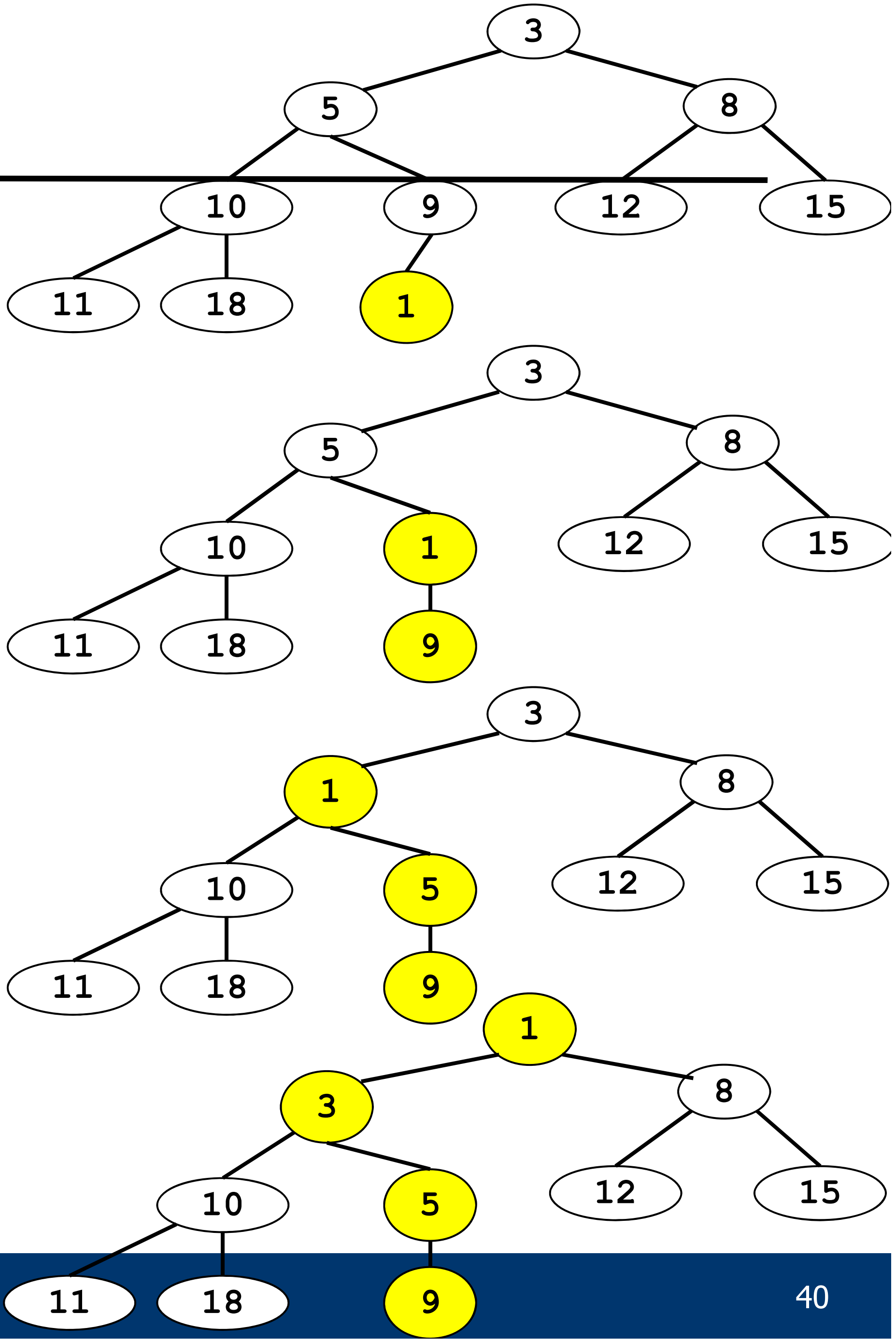
Analysis - Correctness

- We need to show that FC and HC still hold
- HC: Look at the tree after we moved a node k. k may
  - … be smaller than its children. Then HC holds and we are done
  - … be larger than at least one child k2. Assume that k2 is the smaller of the two children (k1, k2) of k. We next swap k and k2. The new parent (k2) now is smaller than its children (k1, k), so the HC holds
  - After the last swap, k has no children – HC holds
- FC: We remove one node, then we sift down
  - Removing last node doesn't affect FC as we remove in the last level
  - Sifting does not change the topology of the tree (we only swap)

Analysis - Complexity

- Recall that a heap with n nodes has ceil(log(n+1)) levels
- During sifting, we perform at most one comparison and one swap in every level
- Thus: O(ceil(log(n+1))) = O(log(n))

Add() on a Heap



- Cannot simply add on top
- Idea: We add new element somewhere in last level and sift up
  - We might need a new level
  - Sifting up: Compare to parent and swap if parent is larger

Analysis

- Correctness
  - HC
    - If parent has only one child, HC holds after each swap
    - Assume a parent k has children k1 and k2, k2 was swapped there in the last move, and k2<k. Since HC held before, k<k1, thus k2<k<k1. We swap k2 and k, and thus the new parent is smaller than its children. On the other hand, if k2≥k, HC holds immediately (and we don't swap).
  - FC: See **`deleteMin()`**
- Complexity: O(log(n))
  - See **`deleteMin()`**

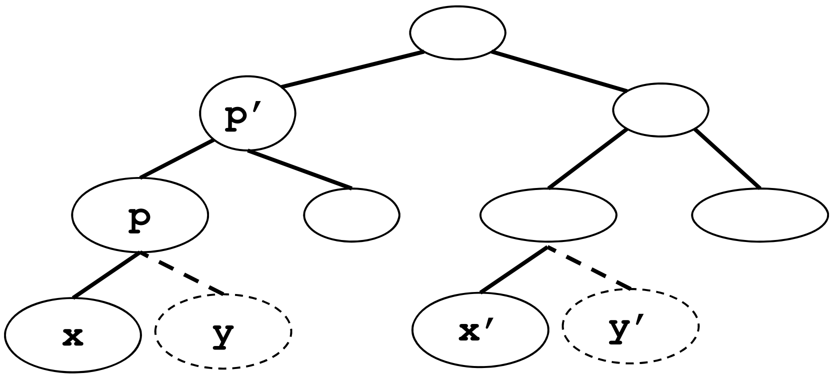
How to Find the Next Free / Last Occupied Node

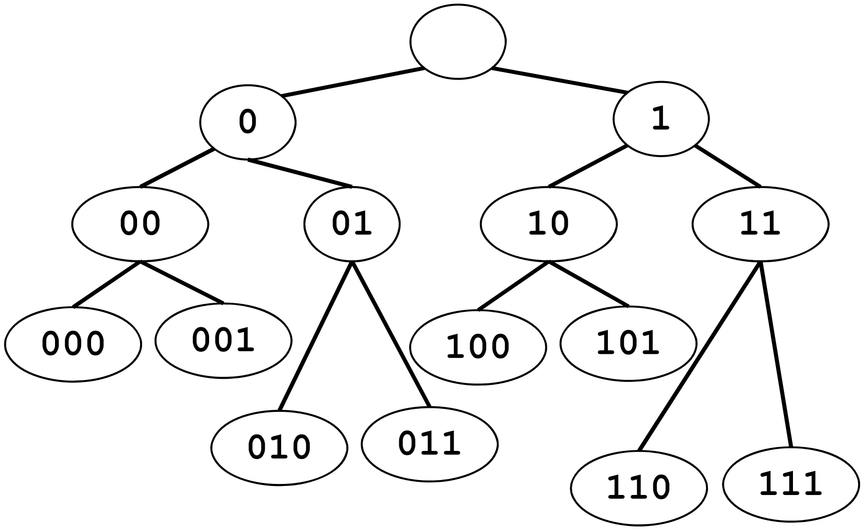
- What do we need to find?
  - For **`deleteMin`**, we use the right-most leaf on the last level
  - For **`add`**, we add the leaf right to the last leaf
- We actually need the parent k
  - From n, we can compute in O(1) the position p of the last leaf in the last level: p = n – 2^(floor(log(n)))
    - Or log(n+1) for **`add`**
  - The parent k of the node at p has index floor(p/2)'th in level d-1
  - The parent k' of k has index floor(p/4)'th in level d-2
  - …
  - Now, in each node, we can decide whether to go left or right
  - Fast trick: Use the binary representation of p

Illustration

- For **`deleteMin`**, we need x (or x'); for **`add`**, we need y (or y')
  - p(x)=0, p(y)=1, p(x')=4, p(y')=5
  - Binary: 000, 001, 100, 101
- Go through bitstring from left-to-right
- Next bit=0: Go left
- Next bit=1: Go right

- Allows finding k in O(log(n))





Summary

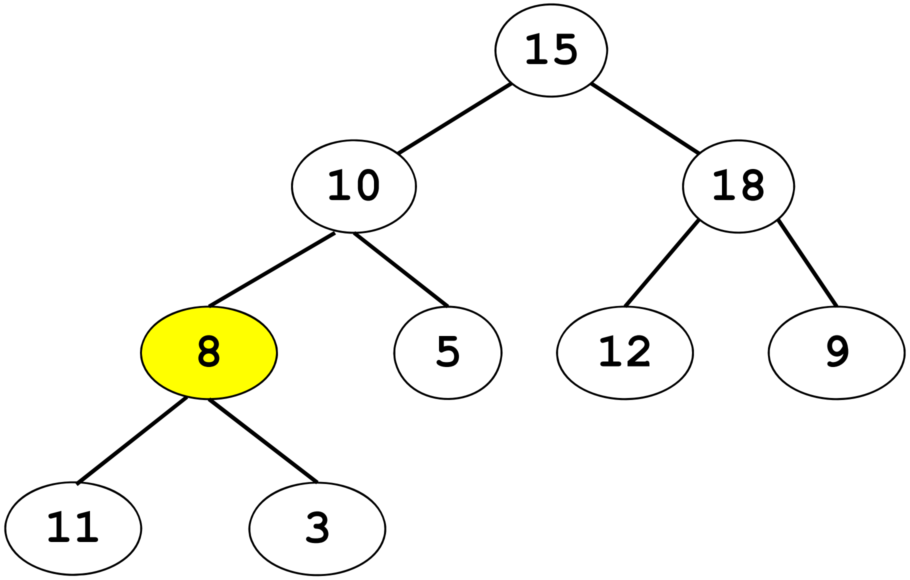
	Linked list	**Sorted linked list**	**Heap**
getMin()	O(n)	O(1)	O(1)
deleteMin()	O(1)	O(1)	O(log(n))
add()	O(1)	O(n)	O(log(n))
merge()	O(1)	O(n1+n2)	O(log(n1)*log(n2))
Space	n add. pointer	n add. pointer	n add. pointer

Heaps can also be kept efficiently in an array – no extra space, but limit to heap size

Creating a Heap

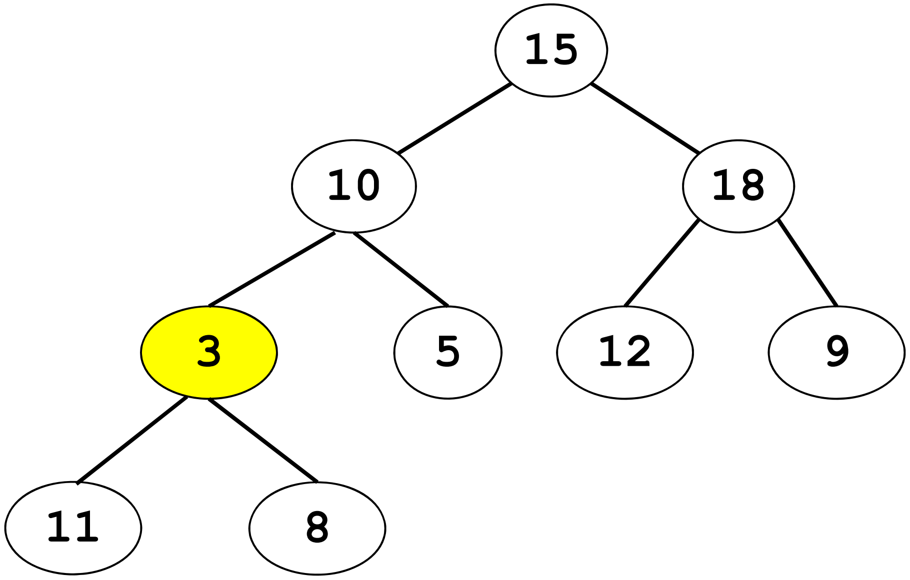
- We start with an unsorted list with n elements
- Naïve algorithm: Start with empty heap and perform n additions
  - Obviously requires O(n*log(n))
- Better: Bottom-Up-Sift-Down
  - Build a tree from the n elements fulfilling the FC (but not HC)
    - Simple fill a tree level-by-level – this is in O(n)
  - Sift-down all nodes on the second-last level
  - Sift-down all nodes on the third-last level
  - …
  - Sift down root

Illustration



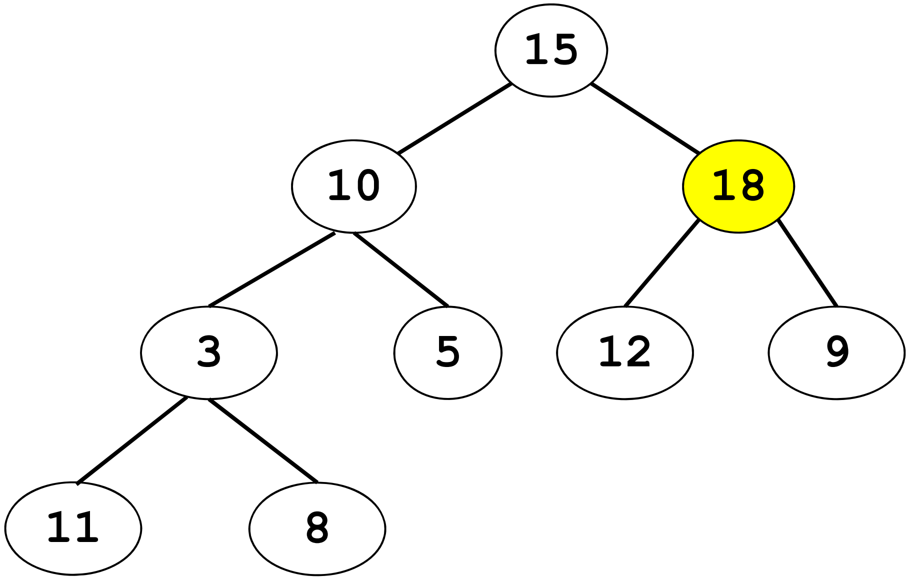
- Start with right most inner node at second-to-last level: 8

Illustration



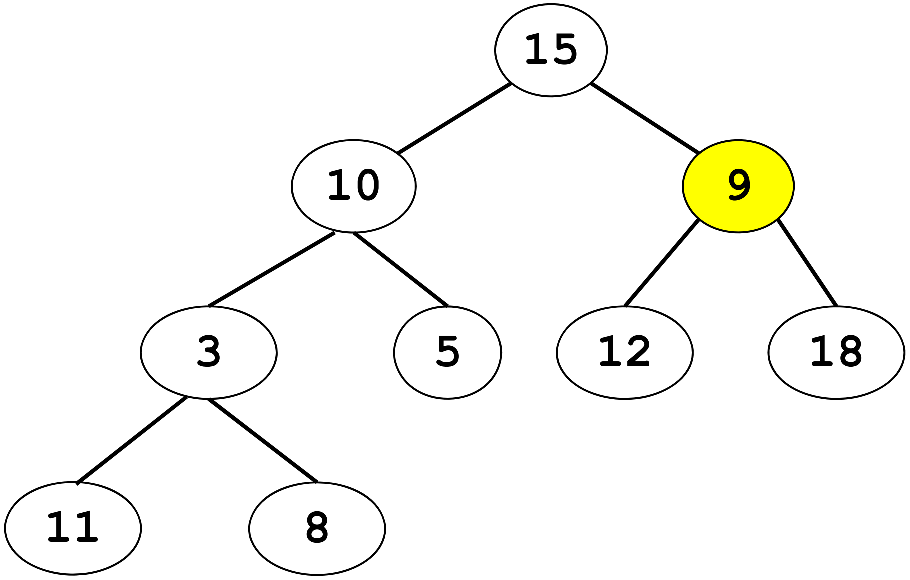
- Sift down 8 (swap with smallest child)

Illustration



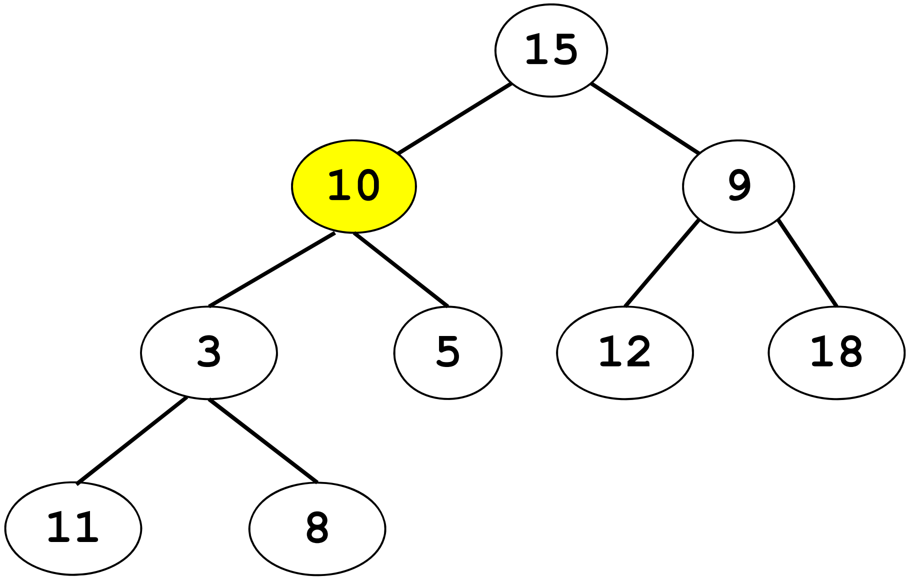
- Done with second-to-last level
- Next, work on third-to-last level, from right to left

Illustration



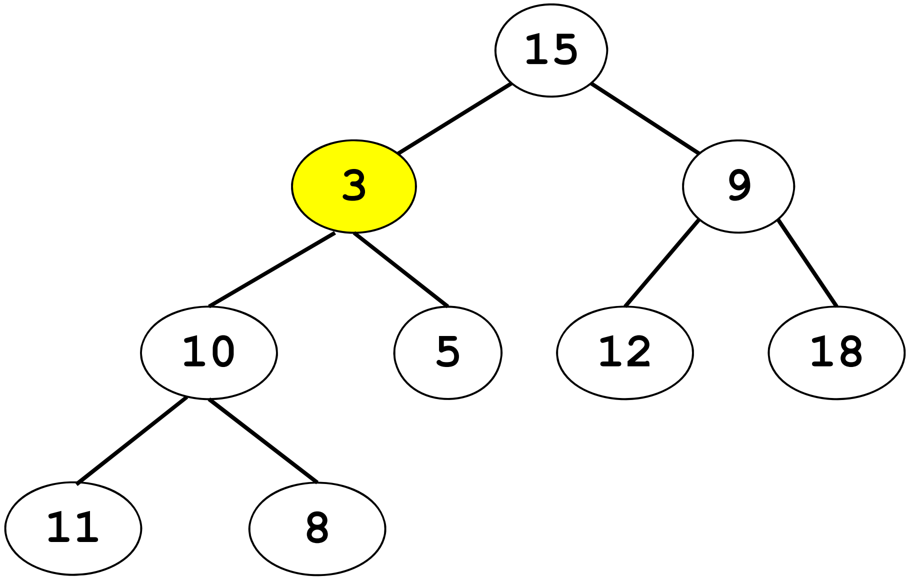
- Sift down 18

Illustration



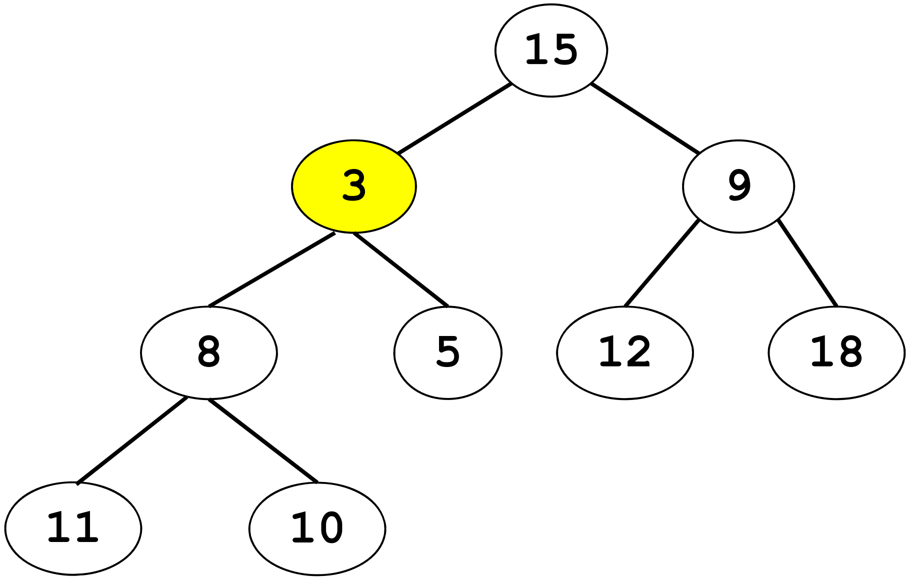
- Sift down 10

Illustration



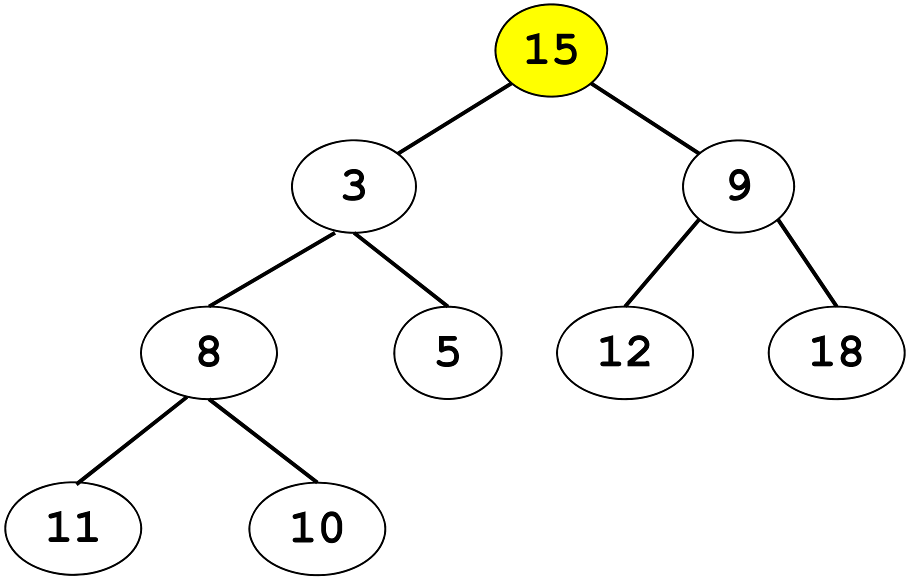
- Sift down 10

Illustration



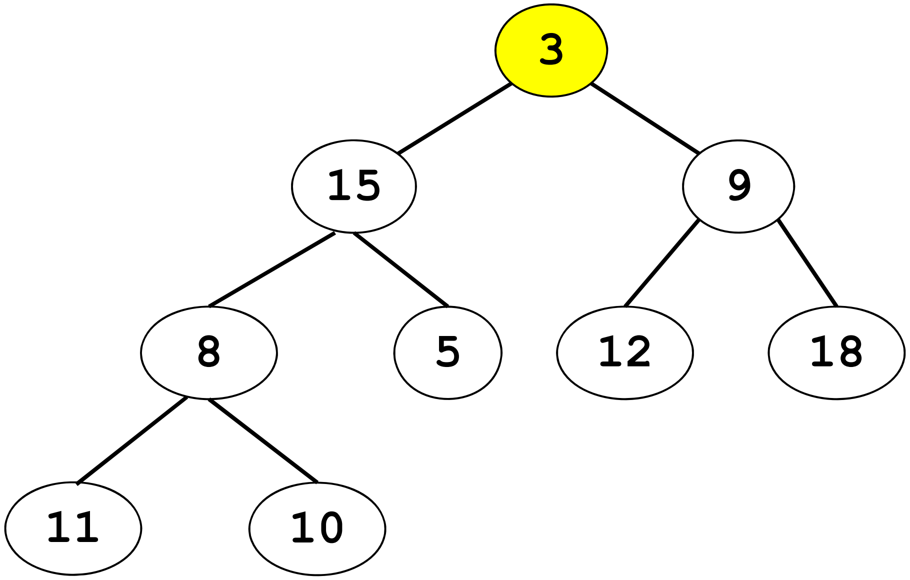
- Sift down 10

Illustration



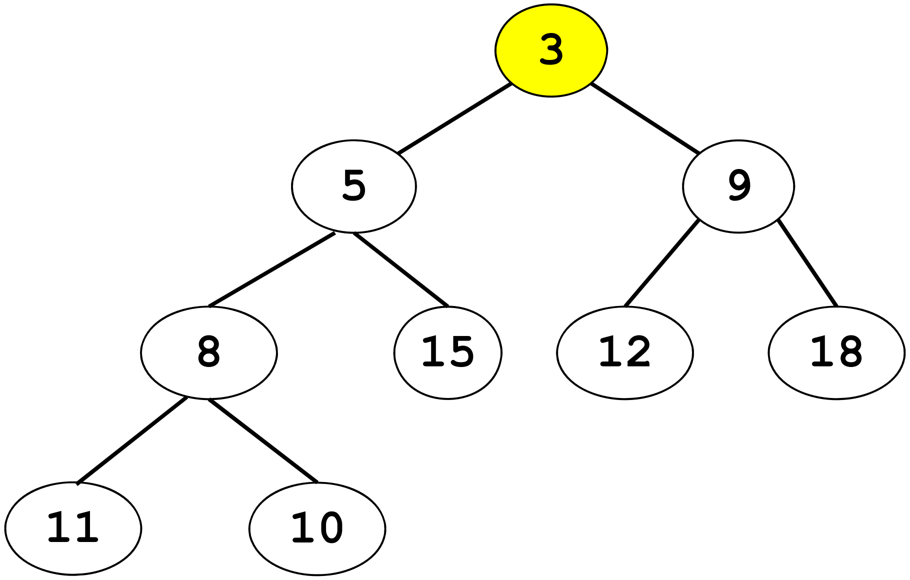
- Sift down 15

Illustration



- Sift down 15

Illustration



- Done

Analysis

- Correctness
  - After finishing one level, all subtrees starting in this level are heaps because sifting-down ensures FC and HC (see **`deleteMin()`**)
  - Thus, when we are done with the first level (root), we have a heap
- Analysis
  - We look at the cost per level h (1 … log(n)=d)
  - For every node at level h, we need at most d-h operations
  - At level h≠d, there are 2^{h-1} nodes
    - For nodes at level d, we don't do anything
  - Over all levels, this yields

$$T(n) = \sum_{h=1}^{d-1} 2^{h-1} * (d-h) = \sum_{h=1}^{d-1} h * 2^{d-h-1} = 2^{d-1} \sum_{h=1}^{d-1} \frac{h}{2^h} \leq n \sum_{h=1}^{\infty} \frac{h}{2^h} = n * 2 = O(n)$$

Heap Sort

- Heaps also are a suitable data structure for sorting
- Heap-Sort (a classical sorting algorithm)
  - Given an unsorted list, first create a heap in O(n)
  - Repeat
    - Take the smallest element and store in array in O(1)
    - Re-build heap in O(log(n))
      - Call `deleteMin( root)`
  - Until heap is empty – after n iterations
- Thus: O(n*log(n))
  - Average-case only slightly better
- Can be implemented in-place when heap is stored in array
  - See [OW93] for details