Minimum Spanning Trees

Arash Rafiey

September 26, 2017

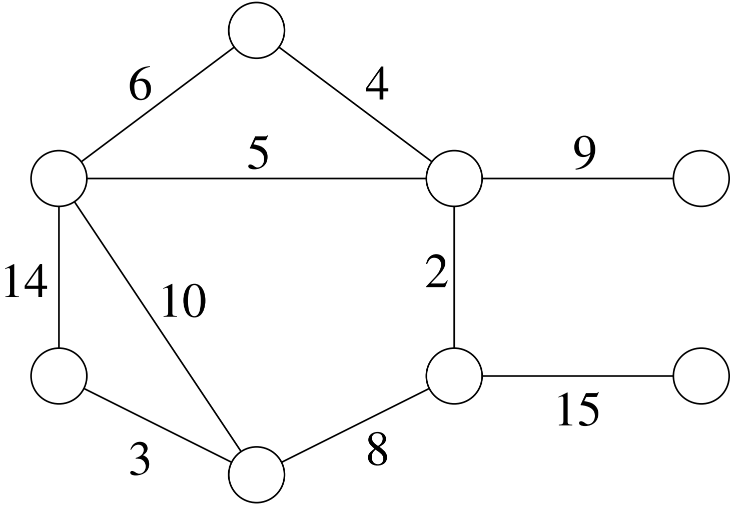
Minimum spanning trees (MST)

One of the most famous greedy algorithms

- Given undirected graph $G = (V, E)$, connected
- Weight function $w : E \to \mathbb{R}$
- Spanning tree: tree that connects all nodes, hence $n = |V|$ nodes and $n - 1$ edges
- MST $T : w(T) = \sum_{(u,v)\in T} w(u, v)$ minimized

Application ?

- Chip design
- Communication infrastructure in networks


6
4
5
9
14
10
2
15
3
8

Growing a minimum spanning tree

First, generic algorithm. It manages set of edges A, maintains invariant:

Prior to each iteration, A is subset of some MST.

Growing a minimum spanning tree

First, generic algorithm. It manages set of edges A, maintains invariant:

Prior to each iteration, A is subset of some MST.

At each step, determine edge (u, v) that can be added to A, i.e. **without violating invariant**, i.e., $A \cup \{(u, v)\}$ is also subset of some MST. We then call (u, v) a **safe edge**.

Growing a minimum spanning tree

First, generic algorithm. It manages set of edges A, maintains invariant:

Prior to each iteration, A is subset of some MST.

At each step, determine edge (u, v) that can be added to A, i.e. **without violating invariant**, i.e., $A \cup \{(u, v)\}$ is also subset of some MST. We then call (u, v) a **safe edge**.

We will describe Kruskal's and Prim's algorithms. They differ in how they specify rules to determine safe edges.

In Kruskal's algorithm, A is a **forest**; while in Prim's algorithm, A is a **single tree** (other components are single nodes).

In Kruskal's algorithm, A is a **forest**; while in Prim's algorithm, A is a **single tree** (other components are single nodes).

Kruskal's algorithm

Finds a safe edge to add to growing forest by finding minimum-weight edge e that connects any two trees (already created).

In Kruskal's algorithm, A is a **forest**; while in Prim's algorithm, A is a **single tree** (other components are single nodes).

Kruskal's algorithm

Finds a safe edge to add to growing forest by finding minimum-weight edge e that connects any two trees (already created).

In Kruskal's algorithm, A is a **forest**; while in Prim's algorithm, A is a **single tree** (other components are single nodes).

Kruskal's algorithm

Finds a safe edge to add to growing forest by finding minimum-weight edge e that connects any two trees (already created).

Kruskal's is **greedy** because at each step it adds an edge of least possible weight.

We will use `Disjoint-Set` data structure.

Each set contains the nodes in a tree of the current forest.

We will use `Disjoint-Set` data structure.

Each set contains the nodes in a tree of the current forest.

We will use the following operations:

- `Make-Set`(u) initializes a new set containing just node u.

We will use `Disjoint-Set` data structure.

Each set contains the nodes in a tree of the current forest.

We will use the following operations:

- `Make-Set`(u) initializes a new set containing just node u.
- `Find-Set`(u) returns representative element from set that contains u (so we can check whether two nodes u, v belong to same tree).

We will use `Disjoint-Set` data structure.

Each set contains the nodes in a tree of the current forest.

We will use the following operations:

- `Make-Set`(u) initializes a new set containing just node u.
- `Find-Set`(u) returns representative element from set that contains u (so we can check whether two nodes u, v belong to same tree).
- `Union`(u, v) combines two sets (the one containing u with the one containing v).

We will use `Disjoint-Set` data structure.

Each set contains the nodes in a tree of the current forest.

We will use the following operations:

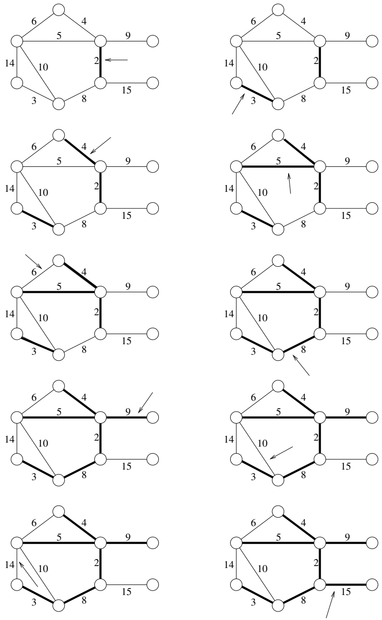
- `Make-Set`(u) initializes a new set containing just node u.
- `Find-Set`(u) returns representative element from set that contains u (so we can check whether two nodes u, v belong to same tree).
- `Union`(u, v) combines two sets (the one containing u with the one containing v).

Time complexity depends on the actual implementation of Disjoint-set data structure. It can be implemented with running time $O(\log n)$.

MST-Kruskal's Algorithm ($G = (V, E)$, function w on E)

1. $A := \emptyset$
2. **for** each node $v \in V[G]$
3.   `Make-Set`(v)
4. sort edges of E into nondecreasing order by weight w
5. **for** each edge $(u, v) \in E$, taken in the order
6.   **if** `Find-Set`$(u) \neq$ `Find-Set` (v)
7. $A := A \cup \{(u, v)\}$
8.   `Union`(u, v)

–in the loop 5–8, for each edge we check whether it belongs to the same component (tree); if not: it's a cheapest edge ($=$ save edge) connecting 2 components (edges are sorted, hence all consecutive edges have a weight at least the weight of the current edge)



Running time

We assume that all `Disjoint-Set` operations, can be done in $\mathcal{O}(\log|V|)$ time

- initializing A takes $\mathcal{O}(1)$
- sorting edges takes $\mathcal{O}(|E|\log|E|)$; since $|E| \leq |V|^2$, we have $\log|E| = \mathcal{O}(\log|V|)$; hence sorting takes: $\mathcal{O}(|E|\log|V|)$
- the initialization loop performs $|V|$ `Make-Set` operation; the main **for** loop performs $\mathcal{O}(E)$ `Find-Set` and `Union` operations;
  together it takes $\mathcal{O}((|V|+|E|)\log|V|)$
- since G is connected, $|E| \geq |V|-1$, so `Disjoint-Set` operations take $\mathcal{O}(|E|\cdot\log|V|)$
- the total running time is $\mathcal{O}(|E|\log|V|)$

Prim's algorithm

- Like Kruskal's, a special case of the generic algorithm.

Prim's algorithm

- Like Kruskal's, a special case of the generic algorithm.
- The set A always forms a **single tree** (as opposed to a forest in Kruskal's).

Prim's algorithm

- Like Kruskal's, a special case of the generic algorithm.
- The set A always forms a **single tree** (as opposed to a forest in Kruskal's).
- The tree starts from a single (arbitrary) node r (root) and grows until it spans all of V.

Prim's algorithm

- Like Kruskal's, a special case of the generic algorithm.
- The set A always forms a **single tree** (as opposed to a forest in Kruskal's).
- The tree starts from a single (arbitrary) node r (root) and grows until it spans all of V.
- At each step, a cheapest edge is added to tree A that connects A to isolated node of $G_A = (V, A)$.

Prim's algorithm

- Like Kruskal's, a special case of the generic algorithm.
- The set A always forms a **single tree** (as opposed to a forest in Kruskal's).
- The tree starts from a single (arbitrary) node r (root) and grows until it spans all of V.
- At each step, a cheapest edge is added to tree A that connects A to isolated node of $G_A = (V, A)$.
- This adds only edges safe for A, hence on termination, A is an MST.

Prim's algorithm

- Like Kruskal's, a special case of the generic algorithm.
- The set A always forms a **single tree** (as opposed to a forest in Kruskal's).
- The tree starts from a single (arbitrary) node r (root) and grows until it spans all of V.
- At each step, a cheapest edge is added to tree A that connects A to isolated node of $G_A = (V, A)$.
- This adds only edges safe for A, hence on termination, A is an MST.
- Strategy is greedy, always pick a cheapest possible edge.

Min-priority queue

Min-priority queue is a data structure that maintains a set of elements S.

Min-priority queue

Min-priority queue is a data structure that maintains a set of elements S.

Each element $v \in S$ has an associated value $key(v)$ (denotes the priority of v). The lower key the more priority.

Min-priority queue

Min-priority queue is a data structure that maintains a set of elements S.

Each element $v \in S$ has an associated value $key(v)$ (denotes the priority of v). The lower key the more priority.

Min-priority queue supports addition, deletion and selection of the element with the smallest key.

Min-priority queue

Min-priority queue is a data structure that maintains a set of elements S.

Each element $v \in S$ has an associated value $key(v)$ (denotes the priority of v). The lower key the more priority.

Min-priority queue supports addition, deletion and selection of the element with the smallest key.

A min-priority queue for a set of n elements can be implemented in such a way that each of the addition, deletion and returning the smallest key element takes $O(\log n)$.

Prim's Algorithm using priority queue

The crucial point is **efficiently selecting new edges**. We store all nodes that are **not** in the tree, in a min-priority queue Q. We have to assign priorities (**keys**) to nodes:

For $v \in V$,

- $key[v]$ is

  $\boxed{\text{the minimum weight of any edge connecting } v \text{ to a node in tree } A}$

  $key[v] = \infty$ if there is no such edge.
- $\pi[v]$ is the parent of v in tree.

During the algorithm, the set A from generic algorithm is kept implicitly as

$$A = \{(v, \pi[v]) : \; v \in V - \{r\} - Q\}$$

When the algorithm terminates, the min-priority queue Q is empty, hence A contains an MST for G:

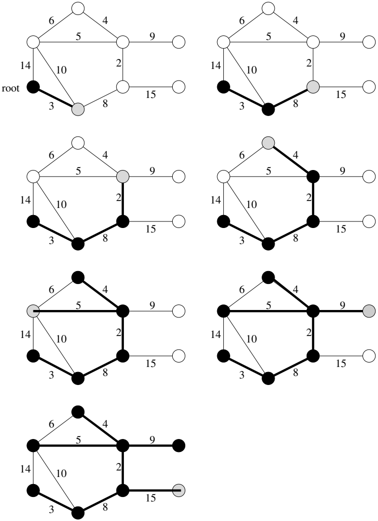
$$A = \{(v, \pi[v]) : \; v \in V - \{r\}\}$$

Prim's Algorithm ($G = (V, E)$, function w, root node $r \in V$)

1. **for** each $u \in V$
2. $key[u] := \infty$
3. $\pi[u] := NIL$
4. $key[r] := 0$
5. $Q := V$
6. **while** $Q \neq \emptyset$
7. $u :=$ `Extract-Min`(Q)
8.   **if** $(u \neq r)$ add $(u, \pi[u])$ into A **else** add r to A.
9.   **for** each edge vu
10.   **if** $v \in Q$ and $w(u, v) < key[v]$
11. $\pi[v] := u$
12. $key[v] := w(u, v)$ `/* Decrease-Key */`

Lines 1–5

- set the key of each node to ∞ (except root r whose key is set to 0 so that it will be processed first)
- set parent of each node to NIL
- initialize min-priority queue Q (all nodes)


root

Running time

Depends on how the min-priority queue Q is implemented.

- for initialization, time $\mathcal{O}(|V|)$
- body of the **while** loop is executed $|V|$ times, each `Extract-Min` takes $\mathcal{O}(\log|V|)$, hence total time for all calls to `Extract-Min` is $O(|V|\log|V|)$
- **for** loop in lines 8–11 is executed $\mathcal{O}(E)$ times altogether, since sum of lengths of all adjacency lists is $2|E|$
- test for membership in Q on line 10, can be implemented in constant time $\mathcal{O}(1)$ (keeping a membership bit for every node)
- line 11 performs `Decrease-Key` operation, each takes $\mathcal{O}(\log|V|)$ time, hence the total time spent here is $\mathcal{O}(|E|\log|V|)$
- the total time: $\mathcal{O}(|V|\log|V| + |E|\log|V|) = \mathcal{O}(|E|\log|V|)$

Correctness of Prim's Algorithm

Let G be a connected, weighted graph.

At every iteration of Prim's algorithm, an edge must be found that connects a node in a constructed subtree to a node outside the subgraph and since G is connected the output of Prim's algorithm is a tree say Y.

Let Y_1 be a minimum spanning tree of graph P. If $Y_1 = Y$ then Y is a minimum spanning tree. Otherwise :

Let e be the first edge added during the construction of tree Y that is not in tree Y_1.

Let V be the set of nodes connected by the edges added before edge e.

One endpoint of edge $e \in V$ and the other is not.

Since tree Y_1 is a spanning tree of graph G, there is a path Q in tree Y_1 joining the two endpoints of e.

As we travel along Q, we must encounter an edge f joining a node in set V to one that is not in set V.

Now, at the iteration when edge e was added to tree Y, edge f could also have been added and it would be added instead of edge e if its weight was less than e, and since edge f was not added, we conclude that

$w(f) \geq w(e)$.

Let tree Y_2 be the graph obtained by removing edge f from and adding edge e to tree Y_1.

It is easy to show that :

1) Tree Y_2 is connected,
2) Tree Y_2 has the same number of edges as tree Y_1,
3) the total weights of the edges in Y_2 is not larger than that of tree Y_1.

Therefore Y_2 is also a minimum spanning tree of graph G and it contains edge e and all the edges added before it during the construction of set V.

Repeat the steps above and we will eventually obtain a minimum spanning tree of graph G that is identical to tree Y. This shows Y is a minimum spanning tree.

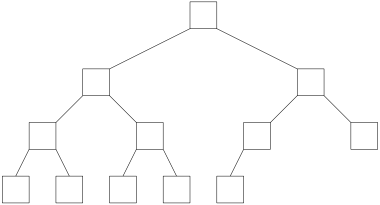
Heap

A **heap** (data structure) is a **linear array** that stores a nearly complete **tree**.

Only talking about **binary heaps** that store **binary trees**.

nearly complete trees:

- all levels except possibly the lowest one are filled
- the bottom level is filled from left to right up to some point



Want to store trees like that to facilitate search in the tree.

Suppose that array A stores (or represents) a binary heap
Two major attributes:

- length(A) is **number of elements** in array A
- heap-size(A) is **number of elements in heap** stored within array A

Although $A[1], \ldots, A[\text{length}(A)]$ can contain **valid numbers**, only elements $A[1], \ldots, A[\text{heap-size}(A)]$ actually store **elements of the heap**, heap-size(A) $\leq$ length(A).

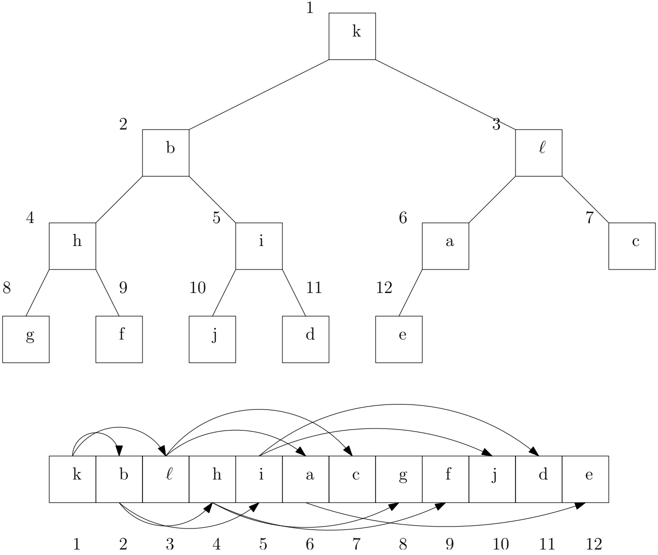
Assignment of tree vertices to array elements:

Very easy:

- root is $A[1]$
- given index i of some node, we have
  - $\textsc{Parent}(i) = \lfloor i/2 \rfloor$
  - $\textsc{Left}(i) = 2i$
  - $\textsc{Right}(i) = 2i + 1$

Implementation straightforward:

- $i \to 2i$
  left-shift by one of bit string representing i
- $i \to 2i + 1$
  left-shift by one plus adding a 1 to the last bit
- $i \to \lfloor i/2 \rfloor$
  right-shift by one


1
k
2
b
3
ℓ
4
h
5
i
6
a
7
c
8
g
9
f
10
j
11
d
12
e
k b ℓ h i a c g f j d e
1 2 3 4 5 6 7 8 9 10 11 12

This particular vertex numbering isn't the only requirement for the thing to be a proper heap

Two kinds: **min-heaps** and **max-heaps**

Both cases, values in nodes satisfy **heap property**

- **max-heap** with **max-heap property**:
  for every node i (other than root)

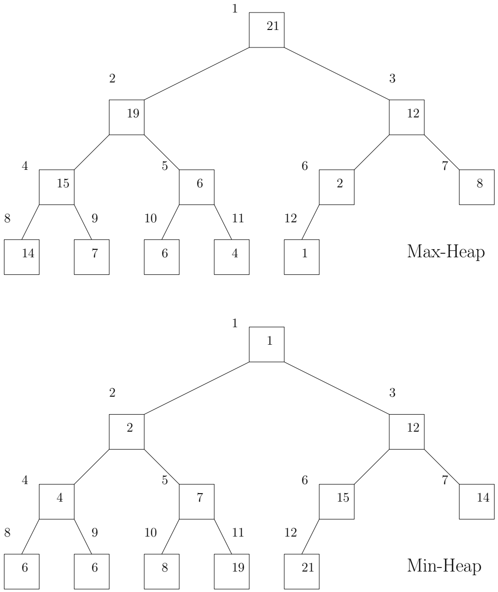
  $$A[\textsc{Parent}(i)] \geq A[i]$$

  meaning: value of node is **at most** value of parent, largest value is stored at root
- **min-heap** with **min-heap property**:
  for every node i (other than root)

  $$A[\textsc{Parent}(i)] \leq A[i]$$

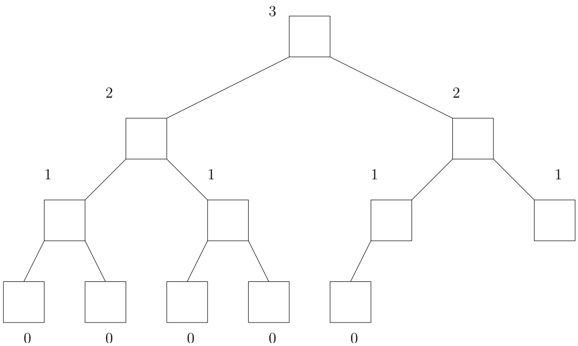
  meaning: value of node is **at least** value of parent, smallest value is stored at root

Suppose given values 1,2,4,6,6,7,8,12,14,15,19,21



Note: given fixed set of values, there are many possible proper min-heaps and max-heaps (except for what's at root)

If viewing heap as "ordinary" tree, define **height** of vertex as # of edges on longest simple downward path from vertex to some leaf



The height of a heap is the height of its root.
A heap of n elements is based on a complete binary tree, therefore its height is $\Theta(\log n)$

What are the minimum and maximum numbers of elements in a heap of height h? Prove that an n-element heap has height $\lfloor \log_2 n \rfloor$.

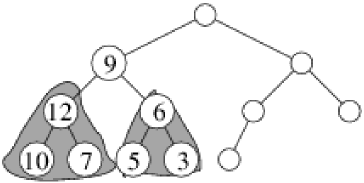
Important basic procedures for max-heaps

1. **Max-Heapify**, runs in time $O(\log n)$ time, a key procedure that maintains the *max-heap property* (if we change a value in the root of a heap, this procedure will correct the heap, so that it's again a heap)
2. **Build-Max-Heap**, runs in time $O(n)$, produces a max-heap from unsorted data

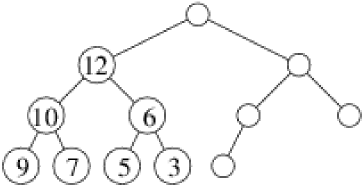
Also **Max-Heap-Insert**, **Heap-Extract-Max**, **Heap-Increase-Key** (run in time $O(\log n)$), **Heap-Maximum**, (run in time $O(1)$), used when the heap is used to implement a *priority queue*

Maintains the max-heap property

Inputs are an array A and an index i
Assumption: sub-trees rooted in $\text{LEFT}(i)$ and $\text{RIGHT}(i)$ are proper max-heaps, but $A[i]$ may be smaller than its children



Task of **Max-Heapify** is to let $A[i]$ float down in the max-heap below it so that heap rooted in i becomes proper max-heap



Max-Heapify (A, i)

$\ell := \textsc{Left}(i)$
$r := \textsc{Right}(i)$

if $\ell \leq$ heap-size(A) and $A[\ell] > A[i]$
  largest $:= \ell$
else
  largest $:= i$

if $r \leq$ heap-size(A) and $A[r] > A[$largest$]$
  largest $:= r$

if largest $\neq i$
  exchange $A[i] \leftrightarrow A[$largest$]$
  Max-Heapify$(A,$ largest$)$

Idea:

Lines 1–2 are just for convenience

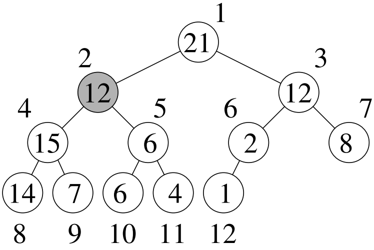
Lines 3–8 find the largest of elements $A[i]$, $A[\ell]$, and $A[r]$

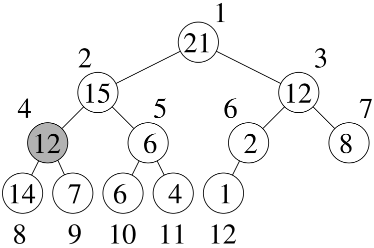
Lines 9–11

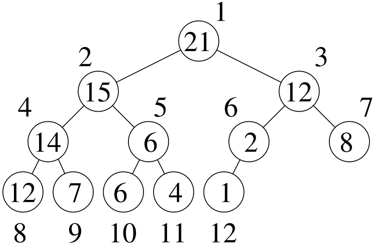
1. first check if there's anything to be done at all,
2. and if yes,
    1. move the "misplaced element" one level down,
    2. and make a recursive call one level deeper on this element

We know that after the exchange we have the largest of $A[i]$, $A[\ell]$, and $A[r]$ in position i, so among these three, everything is OK. However, further down may still be problems.

Also note that we check $\ell \leq$ heap-size(A) and $r \leq$ heap-size(A), so that we don't go checking outside of the heap – the recursion will end if these tests fail


1
21
2
12
3
12
4
15
5
6
6
2
7
8
14
7
6
4
1
8
9
10
11
12


1
21
2
15
3
12
4
12
5
6
6
2
7
8
14
7
6
4
1
8
9
10
11
12


1
21
2
15
3
12
4
14
5
6
6
2
7
8
12
7
6
4
1
8
9
10
11
12

Running time of **Max-Heapify** on node of height h (counted from bottom!) is $O(h)$ and $h = O(\log n)$

Building a Heap

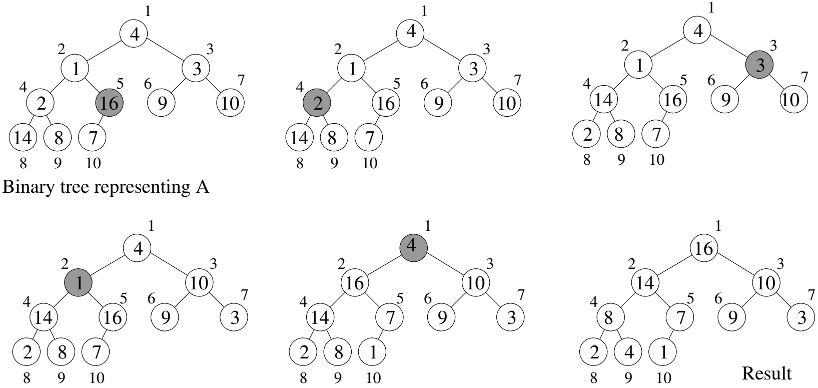
Easy using **Max-Heapify**
Suppose we have given an unordered array $A[1], \ldots, A[n]$ with $n = \text{length}(A)$
One can show that the elements

$$A[\lfloor n/2 \rfloor + 1],\ A[\lfloor n/2 \rfloor + 2],\ \ldots,\ A[n]$$

are the *leaves* of a heap..
Thus, it's OK to initially consider them as 1-element heaps, and run **Max-Heapify** "on top" of them, once for each non-leaf element (1-element heaps are always proper heaps!).

Build-Max-Heap(A)
1. heap-size(A) := length(A)
2. **for** $i := \lfloor \text{length}(A)/2 \rfloor$ **downto** 1
3. Max-Heapify(A, i)

4	1	3	2	16	9	10	14	8	7	Input array A



Build-Max-Heap Running time

Simple bound: calls to Max-Heapify cost $O(\log n)$, there are $O(n)$ of them, thus $O(n \log n)$.

HeapSort

It is easy to write down the algorithm for Heapsort.
Idea as follows:

1. Given unsorted array A, build heap on A, using Build-Max-Heap
2. Extract largest element (is in $A[1]$), and move it to the end of array (swap $A[1]$ and $A[n]$)
3. Decrease the size of the heap by 1
4. The root might not satisfy the heap property (that's where the element formerly in $A[n]$ now is), hence, using Max-Heapify, correct the heap
5. Extract the 2nd-largest element (again, in $A[1]$), and so on...

Heapsort(A)

1. Build-Max-Heap(A)
2. **for** $i :=$ length(A) **downto** 2
3. exchange $A[1] \leftrightarrow A[i]$
4. heap-size(A) $:=$ heap-size(A) $- 1$
5. Max-Heapify($A, 1$)

Heapsort(A)

1. Build-Max-Heap(A)
2. **for** $i :=$ length(A) **downto** 2
3. exchange $A[1] \leftrightarrow A[i]$
4. heap-size(A) $:=$ heap-size(A) $- 1$
5. Max-Heapify($A, 1$)

Running time: $O(n \log n)$ (Build-Max takes $O(n)$, and then $O(n)$ rounds with $O(\log n)$ each).

Priority queues

They are different: there's no "real" FIFO rule anymore.

A **priority queue** maintains set S of elements, each with a **key** (priority).

Two kinds: **max-priority queues** and **min-priority queues**, usually implemented by **max-heaps** and **min-heaps**.

Max-priority queue

Operations:

- **Insert**(S, x) inserts element x into set S
- **Maximum**(S) returns element of S with largest key
- **Extract-Max**(S) removes and returns element of S with largest key
- **Increase-Key**(S, x, k) increases x's key to new value k, assuming k is at least as large as x's old key

Min-priority queue is used via operations: **Insert**, **Minimum**, **Extract-Min**, and **Decrease-Key**.

Max-Heap Operations, Return Max

- using max-heaps, we know that the largest element is in $A[1]$: we have $O(1)$ access to largest element
- removing/inserting elements and increasing keys means that we (basically) can call **Max-Heapify** or a similar procedure (fixing the heap from bottom up) at the right place (relatively efficient operation $\mathcal{O}(\log n)$)

Min-priority queues analogous.

- using max-heaps, we know that the largest element is in $A[1]$: we have $O(1)$ access to largest element
- removing/inserting elements and increasing keys means that we (basically) can call **Max-Heapify** or a similar procedure (fixing the heap from bottom up) at the right place (relatively efficient operation $\mathcal{O}(\log n)$)

Min-priority queues analogous.

Implementation

Heap-Maximum(A)
1. **return** $A[1]$

implements **Maximum** operation in $O(1)$ time.

How to we **remove** the largest element from the queue/heap so that we will still have a proper max-heap?

Heap-Extract-Max(A)

1. **if** heap-size$(A) < 1$ **error** "heap underflow"
2. max $:= A[1]$
3. $A[1] := A[$heap-size$(A)]$
4. heap-size$(A) :=$ heap-size$(A) - 1$
5. **Max-Heapify**$(A, 1)$
6. **return** max

Running time is $O(\log n)$

Max-Heap Operations, Increase key

Suppose that the element which key is to be increased is identified by index i

We first update the key $A[i]$

Clearly, this can destroy the max-heap property, thus we need to find a new place for this element

The idea is to move the updated element as far up toward the root as necessary

Heap-Increase-Key(A, i, key)

1. **if** key $< A[i]$
2. **error** "new key is smaller than current key"
3. $A[i] :=$ key
4. **while** $i > 1$ and $A[\text{Parent}(i)] < A[i]$
5. exchange $A[i] \leftrightarrow A[\text{Parent}(i)]$
6. $i := \text{Parent}(i)$

Running time is $O(\log n)$ (height of tree)

Max-Heap Operations, Insert

Inserting is now easy:

1. add a new element to the end of heap
2. set its key to desired value

Heap-Insert(A, key)

1. heap-size(A) := heap-size$(A) + 1$
2. A[heap-size(A)] $:= -\infty$
3. **Heap-Increase-Key**$(A, \text{heap-size}(A), \text{key})$

Inserting is now easy:

1. add a new element to the end of heap
2. set its key to desired value

Heap-Insert(A, key)

1. $\text{heap-size}(A) := \text{heap-size}(A) + 1$
2. $A[\text{heap-size}(A)] := -\infty$
3. **Heap-Increase-Key**$(A, \text{heap-size}(A), \text{key})$

Running time is $O(\log n)$

Max-Heap Operations, Insert

Inserting is now easy:

1. add a new element to the end of heap
2. set its key to desired value

Heap-Insert(A, key)

1. $\text{heap-size}(A) := \text{heap-size}(A) + 1$
2. $A[\text{heap-size}(A)] := -\infty$
3. **Heap-Increase-Key**$(A, \text{heap-size}(A), \text{key})$

Running time is $O(\log n)$

Conclusion: all considered operations can be implemented to run in time $O(\log n)$, **Maximum**() even in $O(1)$

The lower bound for sorting an arbitrary array of values of size n is $\theta(n \log n)$.

WHY ?