COMP 761: Lecture 23 – Heaps

David Rolnick

October 28, 2020

Problem

What is the average-case time for Quicksort? Reminder, the method is:

- 1) Take an element x from somewhere in the list L.
- 2) Go through L one-by-one and *pivot* using x: that is, put anything lower than x in L_1, anything higher than x in L_2.
- 3) Sort L_1 and L_2 recursively.
- 4) Combine L_1, x, L_2, in that order.

(Please don't post your ideas in the chat just yet, we'll discuss the problem soon in class.)

Course Announcements

Course Announcements

- Problem set 4 posted, due Nov. 6

Quicksort: review

Quicksort: review

- **Quicksort** is another divide and conquer approach.

Quicksort: review

- **Quicksort** is another divide and conquer approach.
- 1) Take an element x from somewhere in the list L.

Quicksort: review

- **Quicksort** is another divide and conquer approach.
- 1) Take an element x from somewhere in the list L.
- 2) Go through L one-by-one and *pivot* using x: that is, put anything lower than x in L_1, anything higher than x in L_2.

Quicksort: review

- **Quicksort** is another divide and conquer approach.
- 1) Take an element x from somewhere in the list L.
- 2) Go through L one-by-one and *pivot* using x: that is, put anything lower than x in L_1, anything higher than x in L_2.
- 3) Sort L_1 and L_2 recursively.

Quicksort: review

- **Quicksort** is another divide and conquer approach.
- 1) Take an element x from somewhere in the list L.
- 2) Go through L one-by-one and *pivot* using x: that is, put anything lower than x in L_1, anything higher than x in L_2.
- 3) Sort L_1 and L_2 recursively.
- 4) Combine L_1, x, L_2, in that order.

Quicksort

Quicksort

- Last time: if pick pivot randomly, best-case time is $\Theta(n \log n)$ and worst-case time is $\Theta(n^2)$.

Quicksort

- Last time: if pick pivot randomly, best-case time is $\Theta(n \log n)$ and worst-case time is $\Theta(n^2)$.
- What do we think is the average-case time?

Quicksort

- Last time: if pick pivot randomly, best-case time is $\Theta(n \log n)$ and worst-case time is $\Theta(n^2)$.
- What do we think is the average-case time?
- The time of the algorithm is cX, where X is the number of comparisons between two elements to see which is bigger.

Quicksort

- Last time: if pick pivot randomly, best-case time is $\Theta(n \log n)$ and worst-case time is $\Theta(n^2)$.
- What do we think is the average-case time?
- The time of the algorithm is cX, where X is the number of comparisons between two elements to see which is bigger.
- Suppose the list L contains the numbers $z_1 < z_2 < \cdots < z_n$ in some order.

Quicksort

- Last time: if pick pivot randomly, best-case time is $\Theta(n \log n)$ and worst-case time is $\Theta(n^2)$.
- What do we think is the average-case time?
- The time of the algorithm is cX, where X is the number of comparisons between two elements to see which is bigger.
- Suppose the list L contains the numbers $z_1 < z_2 < \cdots < z_n$ in some order.
- We have $X = \sum_{i<j} X_{ij}$, where X_{ij} is the indicator variable for whether z_i and z_j are compared (1 if they are, 0 if not).

Quicksort

- Last time: if pick pivot randomly, best-case time is $\Theta(n \log n)$ and worst-case time is $\Theta(n^2)$.
- What do we think is the average-case time?
- The time of the algorithm is cX, where X is the number of comparisons between two elements to see which is bigger.
- Suppose the list L contains the numbers $z_1 < z_2 < \cdots < z_n$ in some order.
- We have $X = \sum_{i<j} X_{ij}$, where X_{ij} is the indicator variable for whether z_i and z_j are compared (1 if they are, 0 if not).
- By linearity of expectation:

$$\mathbb{E}[X] = \sum_{i<j} \mathbb{E}[X_{ij}] = \sum_{i<j} p(\text{comparing } z_i \text{ and } z_j)$$

Quicksort

Quicksort

- Now, let's think about when z_i and z_j are compared.

Quicksort

- Now, let's think about when z_i and z_j are compared.
- Example:

$$\text{sort}([5, 2, 6, 1, 4, 3]) = \text{sort}([2, 1, 3]) + [4] + \text{sort}([5, 6]).$$

Quicksort

- Now, let's think about when z_i and z_j are compared.
- Example:

$$\text{sort}([5, 2, 6, 1, 4, 3]) = \text{sort}([2, 1, 3]) + [4] + \text{sort}([5, 6]).$$

- Here, 4 is compared with everything, 3 and 5 will not be compared.

Quicksort

- Now, let's think about when z_i and z_j are compared.
- Example:

$$\text{sort}([5, 2, 6, 1, 4, 3]) = \text{sort}([2, 1, 3]) + [4] + \text{sort}([5, 6]).$$

- Here, 4 is compared with everything, 3 and 5 will not be compared.
- z_i and z_j are compared if and only if we pick z_i or z_j as a pivot *before* picking any of the other numbers between z_i and z_j.

Quicksort

- Now, let's think about when z_i and z_j are compared.
- Example:

$$\text{sort}([5,2,6,1,4,3]) = \text{sort}([2,1,3]) + [4] + \text{sort}([5,6]).$$

- Here, 4 is compared with everything, 3 and 5 will not be compared.
- z_i and z_j are compared if and only if we pick z_i or z_j as a pivot *before* picking any of the other numbers between z_i and z_j.
- Therefore, we have:

$$\begin{aligned}
&p(\text{comparing } z_i \text{ and } z_j) \\
&= p(\text{picking } z_i \text{ first out of the nums between } z_i \text{ and } z_j) \\
&\quad + p(\text{picking } z_j \text{ first out of the nums between } z_i \text{ and } z_j) \\
&= \frac{1}{j-i+1} + \frac{1}{j-i+1} \\
&= \frac{2}{j-i+1}.
\end{aligned}$$

Quicksort

Quicksort

- So we have:

$$\begin{aligned}
\mathbb{E}[X] &= \sum_{i<j} \mathbb{E}[X_{ij}] \\
&= \sum_{i<j} \frac{2}{j-i+1} \\
&= 2\sum_{i} \left(\frac{1}{1} + \frac{1}{2} + \cdots + \frac{1}{n-i+1} \right) \\
&\leq 2n \left(\frac{1}{1} + \frac{1}{2} + \cdots + \frac{1}{n} \right) \\
&= O(n \log n)
\end{aligned}$$

Quicksort

- So we have:

$$\begin{aligned}
\mathbb{E}[X] &= \sum_{i<j} \mathbb{E}[X_{ij}] \\
&= \sum_{i<j} \frac{2}{j-i+1} \\
&= 2\sum_{i} \left(\frac{1}{1} + \frac{1}{2} + \cdots + \frac{1}{n-i+1} \right) \\
&\leq 2n \left(\frac{1}{1} + \frac{1}{2} + \cdots + \frac{1}{n} \right) \\
&= O(n \log n)
\end{aligned}$$

- Where we used $\frac{1}{1} + \frac{1}{2} + \cdots + \frac{1}{n} = O(\log n)$ from last time.

Quicksort

- So we have:

$$\begin{aligned}
\mathbb{E}[X] &= \sum_{i<j} \mathbb{E}[X_{ij}] \\
&= \sum_{i<j} \frac{2}{j-i+1} \\
&= 2\sum_{i} \left(\frac{1}{1} + \frac{1}{2} + \cdots + \frac{1}{n-i+1} \right) \\
&\leq 2n \left(\frac{1}{1} + \frac{1}{2} + \cdots + \frac{1}{n} \right) \\
&= O(n \log n)
\end{aligned}$$

- Where we used $\frac{1}{1} + \frac{1}{2} + \cdots + \frac{1}{n} = O(\log n)$ from last time.
- Since best-case running time is $\Theta(n \log n)$, the average running time is not just $O(n \log n)$, it is $\Theta(n \log n)$.

Rooted trees

Rooted trees

- A *rooted tree* is a tree (in the sense of graph theory) where one node is identified as the root.

Rooted trees

- A *rooted tree* is a tree (in the sense of graph theory) where one node is identified as the root.
- For any node, just one path to the root (since tree).

Rooted trees

- A *rooted tree* is a tree (in the sense of graph theory) where one node is identified as the root.
- For any node, just one path to the root (since tree).
- So each node has a *depth* = the number of edges between it and the root.

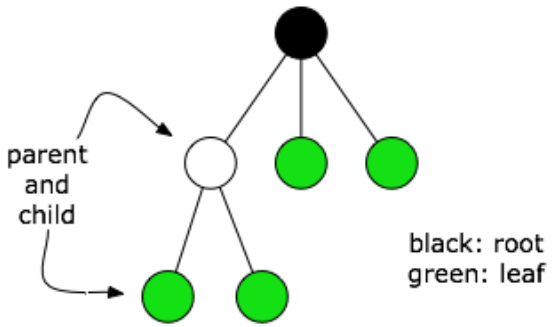


Figure: tree data structure

Rooted trees

- A *rooted tree* is a tree (in the sense of graph theory) where one node is identified as the root.
- For any node, just one path to the root (since tree).
- So each node has a *depth* = the number of edges between it and the root.

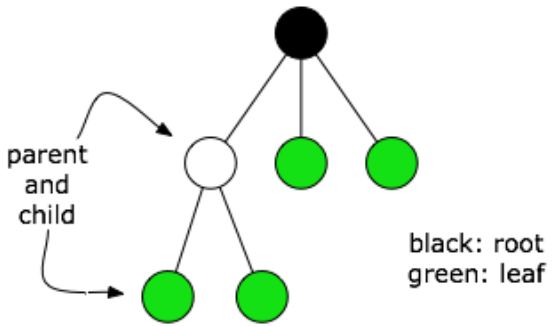


Figure: tree data structure

- The root has depth 0.

Rooted trees

- A *rooted tree* is a tree (in the sense of graph theory) where one node is identified as the root.
- For any node, just one path to the root (since tree).
- So each node has a *depth* = the number of edges between it and the root.

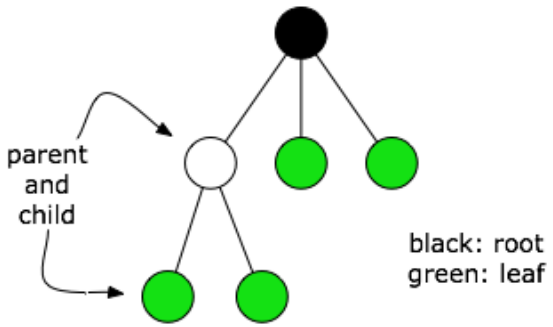


Figure: tree data structure

- The root has depth 0.
- Every node of depth $d > 0$ has a *parent*, which is its unique neighbor at depth $d - 1$.

Rooted trees

- A *rooted tree* is a tree (in the sense of graph theory) where one node is identified as the root.
- For any node, just one path to the root (since tree).
- So each node has a *depth* = the number of edges between it and the root.

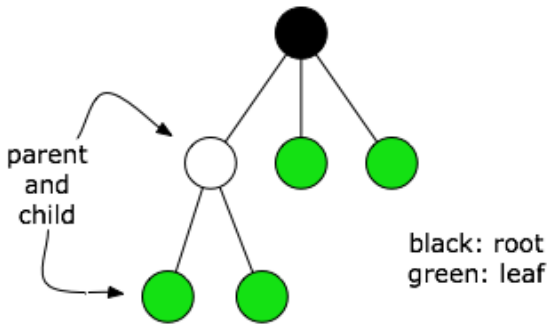


Figure: tree data structure

- The root has depth 0.
- Every node of depth $d > 0$ has a *parent*, which is its unique neighbor at depth $d - 1$.
- If w is the parent of v, then v is said to be a *child* of w.

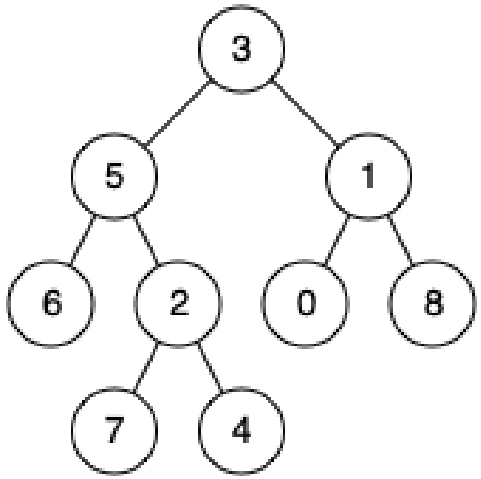
Binary trees

Binary trees

- A *binary tree* is a rooted tree where every node has at most two children, which are designated *left* and *right* children.

Binary trees

- A *binary tree* is a rooted tree where every node has at most two children, which are designated *left* and *right* children.
- Often nodes in a binary tree are used to store data.



Binary trees

Binary trees

- What can the maximum depth be for a binary tree with n nodes?

Binary trees

- What can the maximum depth be for a binary tree with n nodes?
- It could just be a path, with each node the parent of the next, so could have depth $n - 1$.

Binary trees

- What can the maximum depth be for a binary tree with n nodes?
- It could just be a path, with each node the parent of the next, so could have depth $n - 1$.
- What is the minimum depth?

Binary trees

- What can the maximum depth be for a binary tree with n nodes?
- It could just be a path, with each node the parent of the next, so could have depth $n - 1$.
- What is the minimum depth?
- The tree could be perfectly balanced with 2 nodes of depth 1, 4 of depth 2, etc.

Binary trees

- What can the maximum depth be for a binary tree with n nodes?
- It could just be a path, with each node the parent of the next, so could have depth $n - 1$.
- What is the minimum depth?
- The tree could be perfectly balanced with 2 nodes of depth 1, 4 of depth 2, etc.
- In this case, if depth is d, we have

$$\begin{aligned} n &= 1 + 2 + 4 + \ldots + 2^d \\ &= \frac{2^{d+1} - 1}{2 - 1} \\ &= 2^{d+1} - 1, \end{aligned}$$

so the minimum depth is $d = \Theta(\log n)$.

Heaps

Heaps

- A *max-heap* is a binary tree in which the key stored in any node is greater than the value stored by both its children.

Heaps

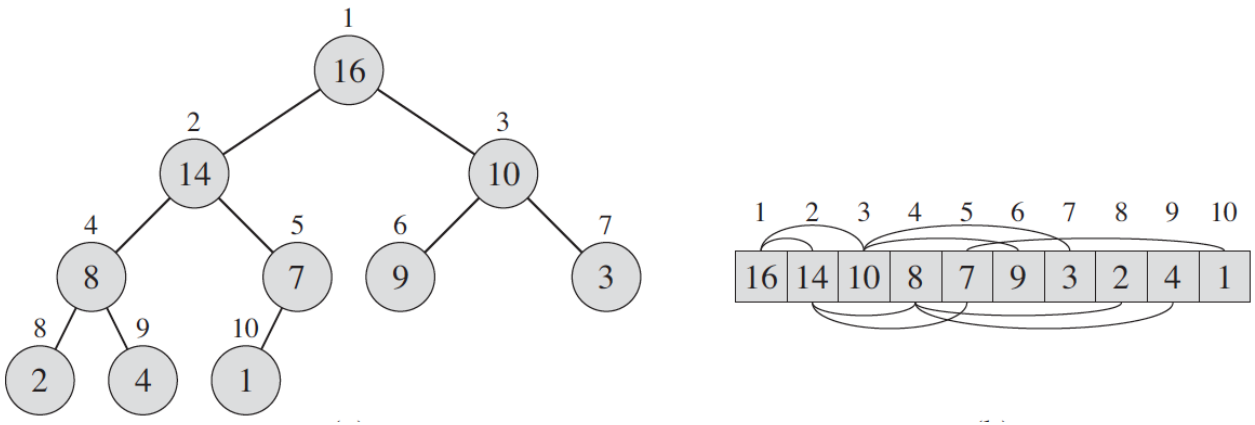
- A *max-heap* is a binary tree in which the key stored in any node is greater than the value stored by both its children.
- A *min-heap* is the same, just with min instead of max.

Heaps

- A *max-heap* is a binary tree in which the key stored in any node is greater than the value stored by both its children.
- A *min-heap* is the same, just with min instead of max.
- We'll assume heaps are max-heaps here, but same logic will apply to min-heaps.

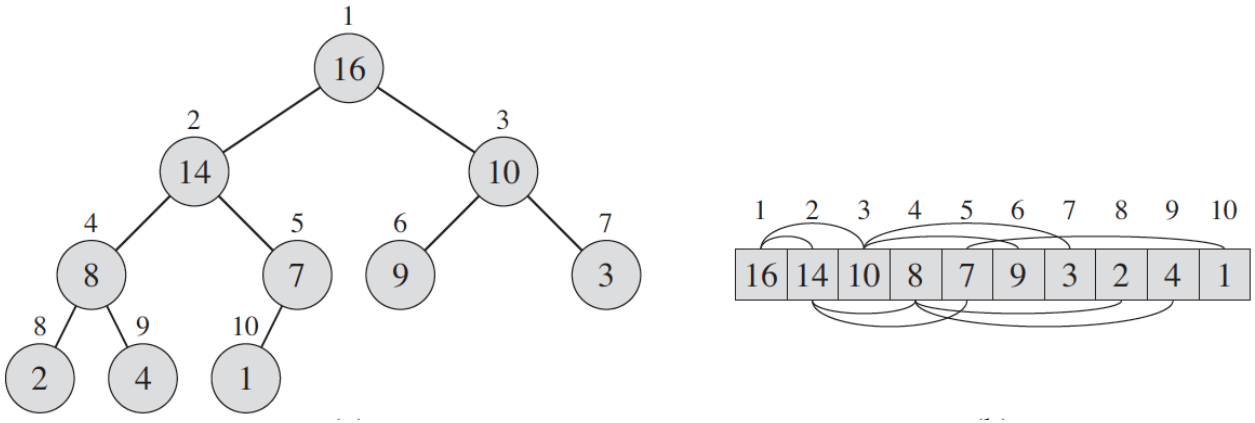
Heaps

- A *max-heap* is a binary tree in which the key stored in any node is greater than the value stored by both its children.
- A *min-heap* is the same, just with min instead of max.
- We'll assume heaps are max-heaps here, but same logic will apply to min-heaps.
- The root node always must have the largest key.



Heaps

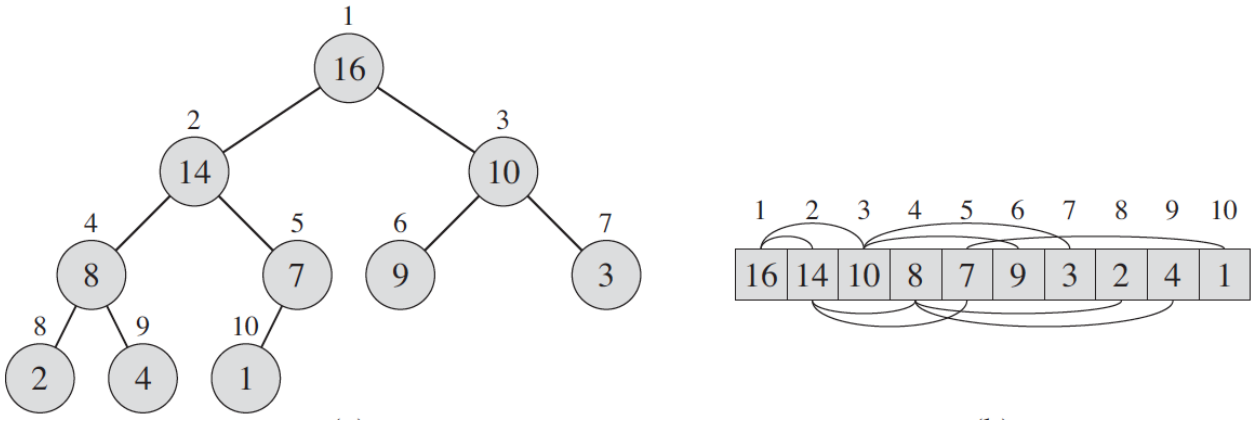
- A *max-heap* is a binary tree in which the key stored in any node is greater than the value stored by both its children.
- A *min-heap* is the same, just with min instead of max.
- We'll assume heaps are max-heaps here, but same logic will apply to min-heaps.
- The root node always must have the largest key.



- We also assume in a heap that each *row* (set of nodes of a single depth) is full except possibly the last one, so depth = $\Theta(\log n)$.

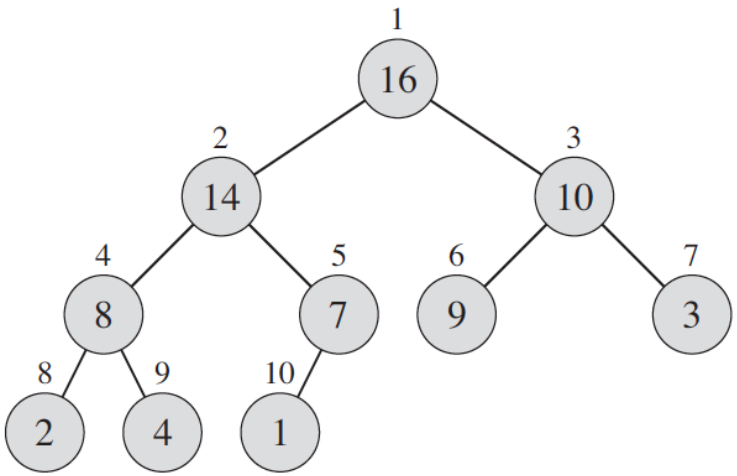
Heaps

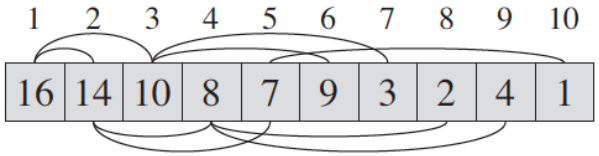
- A *max-heap* is a binary tree in which the key stored in any node is greater than the value stored by both its children.
- A *min-heap* is the same, just with min instead of max.
- We'll assume heaps are max-heaps here, but same logic will apply to min-heaps.
- The root node always must have the largest key.



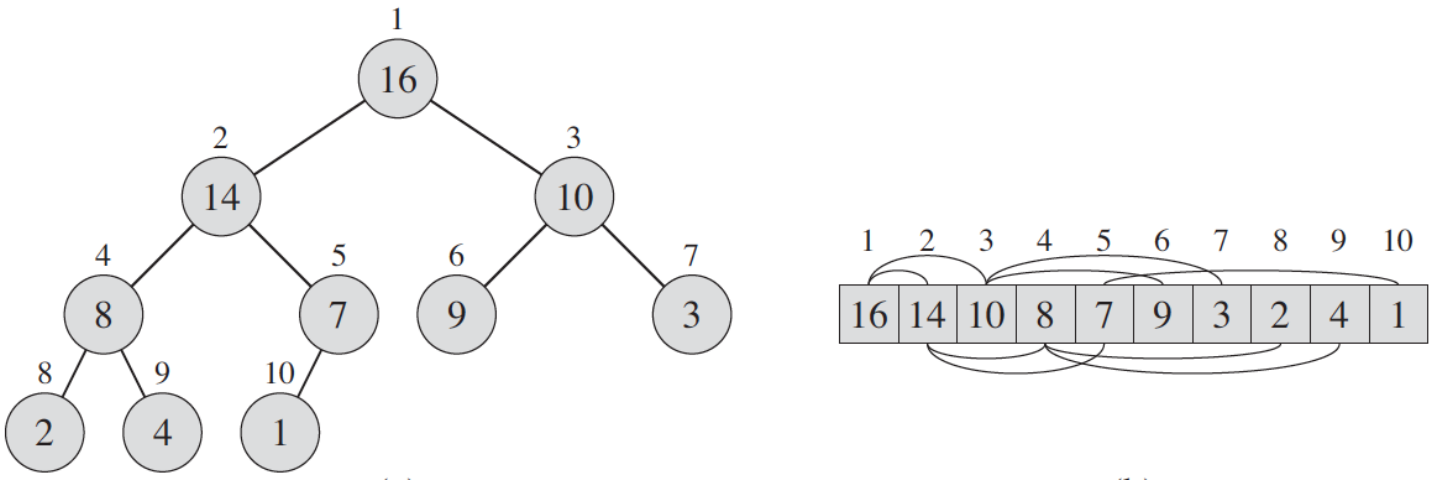
- We also assume in a heap that each *row* (set of nodes of a single depth) is full except possibly the last one, so depth = $\Theta(\log n)$.
- And that the last row has all its nodes as far left as possible (easy by swapping left and right).

Heaps



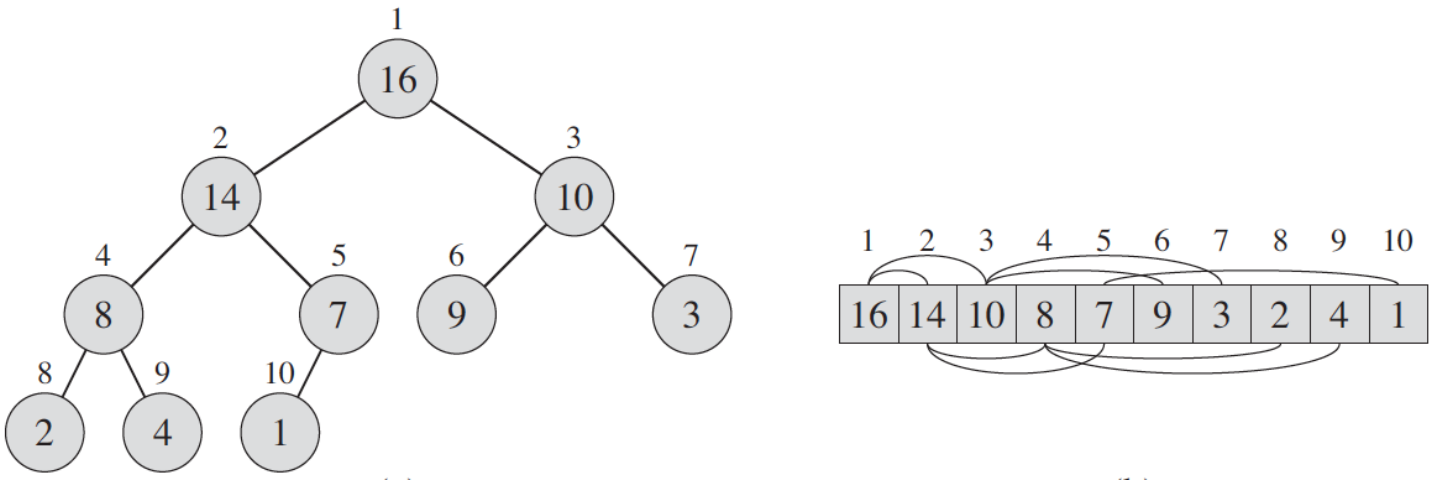


Heaps



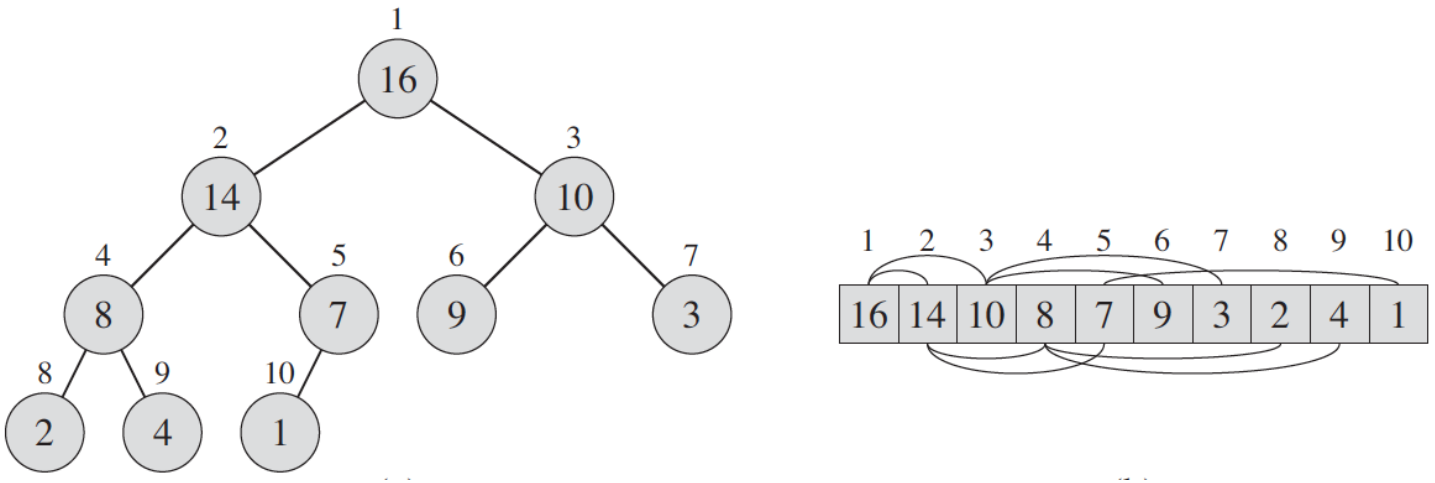
- Here, nodes have a *key* (value stored in them), and also an *index* (starting from the root and counting across each row in turn).

Heaps



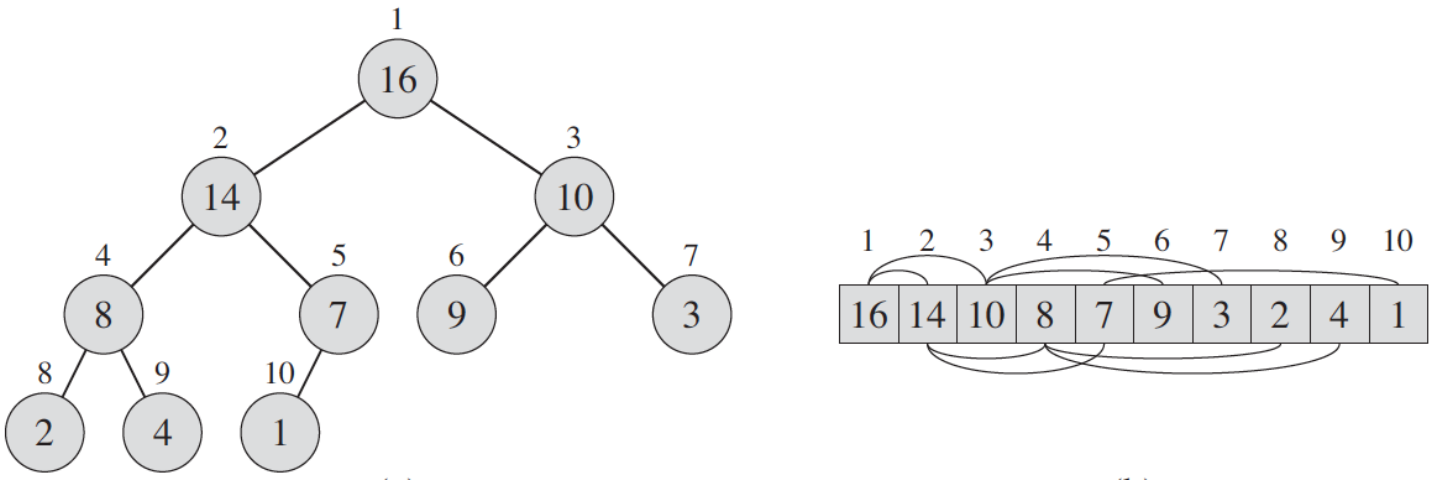
- Here, nodes have a *key* (value stored in them), and also an *index* (starting from the root and counting across each row in turn).
- Indexing makes it easy to move about in the tree with bitwise operations.

Heaps



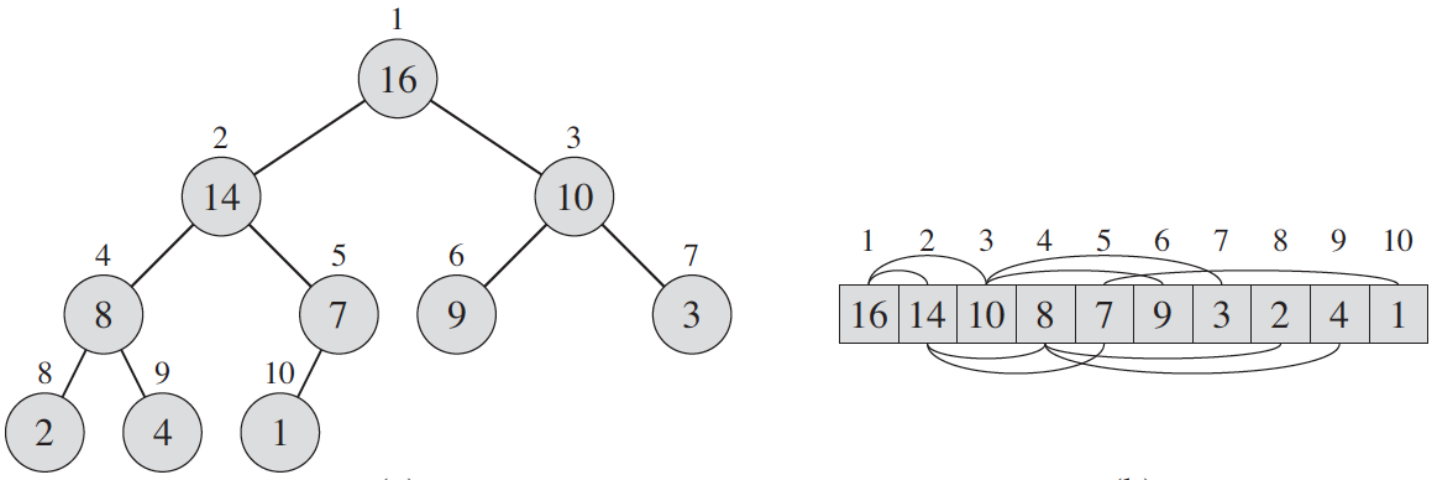
- Here, nodes have a *key* (value stored in them), and also an *index* (starting from the root and counting across each row in turn).
- Indexing makes it easy to move about in the tree with bitwise operations.
- How to get from a node to its left child?

Heaps



- Here, nodes have a *key* (value stored in them), and also an *index* (starting from the root and counting across each row in turn).
- Indexing makes it easy to move about in the tree with bitwise operations.
- How to get from a node to its left child?
- Just double the index!

Heaps



- Here, nodes have a *key* (value stored in them), and also an *index* (starting from the root and counting across each row in turn).
- Indexing makes it easy to move about in the tree with bitwise operations.
- How to get from a node to its left child?
- Just double the index!
- For the right child, double the index and add 1.

Making a heap

Making a heap

- How do we make a heap from a list?

Making a heap

- How do we make a heap from a list?
- Let's start by throwing the numbers into the right shape in some order.

Making a heap

- How do we make a heap from a list?
- Let's start by throwing the numbers into the right shape in some order.
- We need to reorder them so each parent is larger than its children.

Making a heap

- How do we make a heap from a list?
- Let's start by throwing the numbers into the right shape in some order.
- We need to reorder them so each parent is larger than its children.
- Let's make this happen starting with the bottom row.

Making a heap

- How do we make a heap from a list?
- Let's start by throwing the numbers into the right shape in some order.
- We need to reorder them so each parent is larger than its children.
- Let's make this happen starting with the bottom row.
- Nodes in the bottom row have no children, so that row is fine.

Making a heap

- How do we make a heap from a list?
- Let's start by throwing the numbers into the right shape in some order.
- We need to reorder them so each parent is larger than its children.
- Let's make this happen starting with the bottom row.
- Nodes in the bottom row have no children, so that row is fine.
- Now for the next row up.

Making a heap

- How do we make a heap from a list?
- Let's start by throwing the numbers into the right shape in some order.
- We need to reorder them so each parent is larger than its children.
- Let's make this happen starting with the bottom row.
- Nodes in the bottom row have no children, so that row is fine.
- Now for the next row up.
- If any number is less than one of its children, we can swap it with the larger of its children.

Making a heap

- How do we make a heap from a list?
- Let's start by throwing the numbers into the right shape in some order.
- We need to reorder them so each parent is larger than its children.
- Let's make this happen starting with the bottom row.
- Nodes in the bottom row have no children, so that row is fine.
- Now for the next row up.
- If any number is less than one of its children, we can swap it with the larger of its children.
- Similar process for rows higher up.

Making a heap

Making a heap

- Assume nodes depth $> k$ are already sorted correctly.

Making a heap

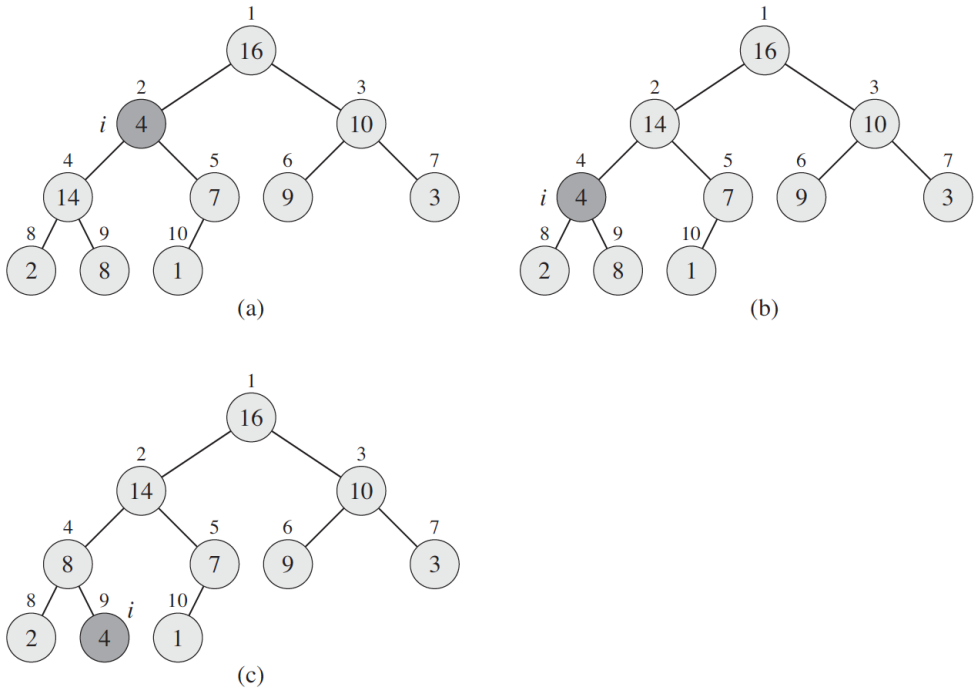
- Assume nodes depth $> k$ are already sorted correctly.
- For each node at depth k, if smaller than one of its children, swap with that child.

Making a heap

- Assume nodes depth $> k$ are already sorted correctly.
- For each node at depth k, if smaller than one of its children, swap with that child.
- Where the key is now, swap again if necessary.

Making a heap

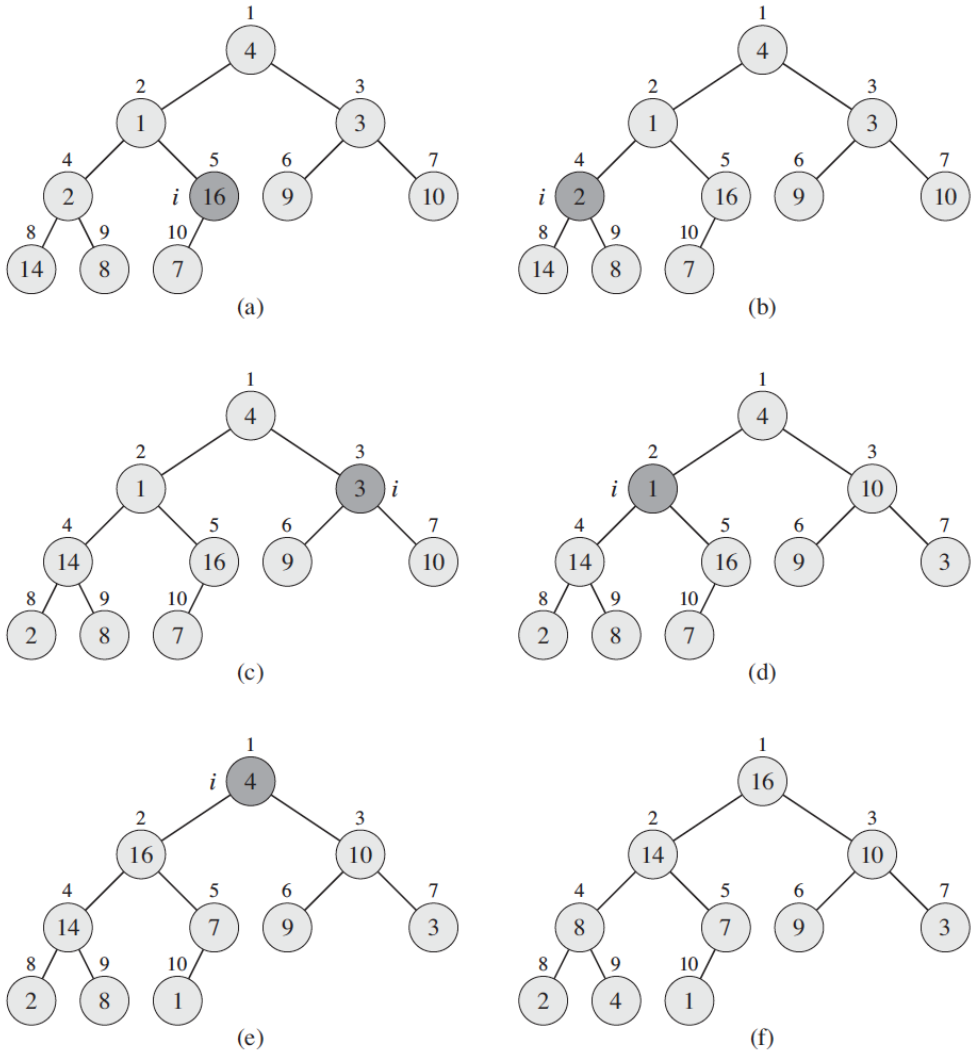
- Assume nodes depth $> k$ are already sorted correctly.
- For each node at depth k, if smaller than one of its children, swap with that child.
- Where the key is now, swap again if necessary.
- Continue swapping until the max property is satisfied or the key reaches a leaf node.



Making a heap

Making a heap

- We can do this to order the whole heap!



Making a heap

Making a heap

- How long does it take?

Making a heap

- How long does it take?
- We have to fix every node in the heap.

Making a heap

- How long does it take?
- We have to fix every node in the heap.
- How long does it take to fix a node at depth k?

Making a heap

- How long does it take?
- We have to fix every node in the heap.
- We have to swap a node at depth k up to $d - k$ times, if d is the maximum depth.

Making a heap

- How long does it take?
- We have to fix every node in the heap.
- We have to swap a node at depth k up to $d - k$ times, if d is the maximum depth.
- How many nodes are at depth k?

Making a heap

- How long does it take?
- We have to fix every node in the heap.
- We have to swap a node at depth k up to $d - k$ times, if d is the maximum depth.
- Up to 2^k nodes at depth k, so total $\#$ moves is:

Making a heap

- How long does it take?
- We have to fix every node in the heap.
- We have to swap a node at depth k up to $d - k$ times, if d is the maximum depth.
- Up to 2^k nodes at depth k, so total # moves is:

$$\leq \sum_{k=0}^{d} (d-k)2^k$$

Making a heap

- How long does it take?
- We have to fix every node in the heap.
- We have to swap a node at depth k up to $d - k$ times, if d is the maximum depth.
- Up to 2^k nodes at depth k, so total # moves is:

$$\leq \sum_{k=0}^{d} (d-k)2^k$$

$$= 1 + (1+2) + (1+2+4) + \cdots + (1+2+\cdots+2^{d-1})$$

Making a heap

- How long does it take?
- We have to fix every node in the heap.
- We have to swap a node at depth k up to $d - k$ times, if d is the maximum depth.
- Up to 2^k nodes at depth k, so total # moves is:

$$\begin{aligned}
&\leq \sum_{k=0}^{d} (d-k)2^k \\
&= 1 + (1+2) + (1+2+4) + \cdots + (1+2+\cdots+2^{d-1}) \\
&= (2-1) + (4-1) + (8-1) + \cdots + (2^d - 1)
\end{aligned}$$

Making a heap

- How long does it take?
- We have to fix every node in the heap.
- We have to swap a node at depth k up to $d - k$ times, if d is the maximum depth.
- Up to 2^k nodes at depth k, so total # moves is:

$$
\begin{aligned}
&\leq \sum_{k=0}^{d} (d-k)2^k \\
&= 1 + (1+2) + (1+2+4) + \cdots + (1+2+\cdots+2^{d-1}) \\
&= (2-1) + (4-1) + (8-1) + \cdots + (2^d - 1) \\
&= (2+4+8+\cdots+2^d) - d
\end{aligned}
$$

Making a heap

- How long does it take?
- We have to fix every node in the heap.
- We have to swap a node at depth k up to $d - k$ times, if d is the maximum depth.
- Up to 2^k nodes at depth k, so total # moves is:

$$
\begin{aligned}
&\leq \sum_{k=0}^{d} (d-k)2^k \\
&= 1 + (1+2) + (1+2+4) + \cdots + (1+2+\cdots+2^{d-1}) \\
&= (2-1) + (4-1) + (8-1) + \cdots + (2^d - 1) \\
&= (2+4+8+\cdots+2^d) - d \\
&= 2^{d+1} - 2 - d
\end{aligned}
$$

Making a heap

- How long does it take?
- We have to fix every node in the heap.
- We have to swap a node at depth k up to $d - k$ times, if d is the maximum depth.
- Up to 2^k nodes at depth k, so total # moves is:

$$
\begin{aligned}
&\leq \sum_{k=0}^{d} (d-k)2^k \\
&= 1 + (1+2) + (1+2+4) + \cdots + (1+2+\cdots+2^{d-1}) \\
&= (2-1) + (4-1) + (8-1) + \cdots + (2^d - 1) \\
&= (2+4+8+\cdots+2^d) - d \\
&= 2^{d+1} - 2 - d \\
&= O(2^d) = O(n).
\end{aligned}
$$

Making a heap

- How long does it take?
- We have to fix every node in the heap.
- We have to swap a node at depth k up to $d - k$ times, if d is the maximum depth.
- Up to 2^k nodes at depth k, so total # moves is:

$$
\begin{aligned}
&\leq \sum_{k=0}^{d} (d-k)2^k \\
&= 1 + (1+2) + (1+2+4) + \cdots + (1+2+\cdots+2^{d-1}) \\
&= (2-1) + (4-1) + (8-1) + \cdots + (2^d - 1) \\
&= (2+4+8+\cdots+2^d) - d \\
&= 2^{d+1} - 2 - d \\
&= O(2^d) = O(n).
\end{aligned}
$$

- So $O(n)$ time to build the heap.

Priority queues

Priority queues

- A *max-priority queue* is a data structure S that allows you to perform the following operations
  - Insert(S, x), inserting key x into S.
  - Maximum(S), returns the maximum of S.
  - ExtractMax(S), removes the maximum from S.
  - IncreaseKey(S, i, x) takes the element at index i and increases it to value x (assuming the key was smaller before).

Priority queues

- A *max-priority queue* is a data structure S that allows you to perform the following operations
  - Insert(S, x), inserting key x into S.
  - Maximum(S), returns the maximum of S.
  - ExtractMax(S), removes the maximum from S.
  - IncreaseKey(S, i, x) takes the element at index i and increases it to value x (assuming the key was smaller before).
- Similarly, a *min-priority queue* is a data structure allowing Insert, Minimum, ExtractMin, and DecreaseKey.

Priority queues

- A *max-priority queue* is a data structure S that allows you to perform the following operations
  - Insert(S, x), inserting key x into S.
  - Maximum(S), returns the maximum of S.
  - ExtractMax(S), removes the maximum from S.
  - IncreaseKey(S, i, x) takes the element at index i and increases it to value x (assuming the key was smaller before).
- Similarly, a *min-priority queue* is a data structure allowing Insert, Minimum, ExtractMin, and DecreaseKey.
- Priority queues are useful across algorithms.

Priority queues

- A *max-priority queue* is a data structure S that allows you to perform the following operations
  - Insert(S, x), inserting key x into S.
  - Maximum(S), returns the maximum of S.
  - ExtractMax(S), removes the maximum from S.
  - IncreaseKey(S, i, x) takes the element at index i and increases it to value x (assuming the key was smaller before).
- Similarly, a *min-priority queue* is a data structure allowing Insert, Minimum, ExtractMin, and DecreaseKey.
- Priority queues are useful across algorithms.
- A max-heap can be used to implement a max-priority queue!

Priority queues

- A *max-priority queue* is a data structure S that allows you to perform the following operations
  - Insert(S, x), inserting key x into S.
  - Maximum(S), returns the maximum of S.
  - ExtractMax(S), removes the maximum from S.
  - IncreaseKey(S, i, x) takes the element at index i and increases it to value x (assuming the key was smaller before).
- Similarly, a *min-priority queue* is a data structure allowing Insert, Minimum, ExtractMin, and DecreaseKey.
- Priority queues are useful across algorithms.
- A max-heap can be used to implement a max-priority queue!
- (Likewise, a min-heap can be used to implement a min-priority queue.)

Next time!

Graph algorithms I