

Lecture 16

Ordered Structures

- Ordered Structures
- Priority Queues
- Heaps

Ordered Structures

Ordered Structures

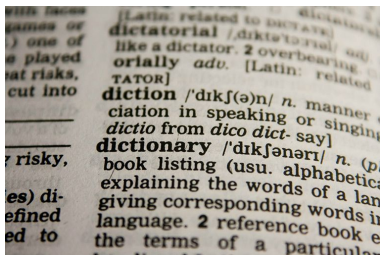
There are many benefits to storing data in a specific order, with some examples listed below.

1. Finding a particular value via binary search in $O(\log(n))$ -time.

In this case, the order or priorities are part of the data.

2. Efficient access to the most important thing in $O(1)$ -time.

In this case, the priorities are not part of the data. Instead the data has an associated priority.



Dictionary and Phone Book 😄

The ordering or priorities are part of the data.



Which graphical elements should have drawing priority?

The priorities are added to the data.

We save time by having the data in order.

But we need to spend time (and possibly space) to put the data in order and/or keep it in order.

Discussion: Ordered Arrays, Linked Lists, Vectors

Our basic linear data structures – arrays, linked lists, vectors – can be used for ordered data.

- We keep the data in sorted order (i.e., the first or highest priority item is first).

Let's focus on the efficiency of the following operations.

- **Insert**. Insert data with a given priority. (The priority may or may not be part of the data).
- **Find**. Return the data with the highest priority.
- **Remove**. Remove the data with the highest priority.



Think about this for 2 minutes.

Then discuss it with your neighbor for 3 minutes.

	Insert	Find	Remove
Array	$O(n)$	$O(1)$	$O(n)$ or $O(1)$ linear or circular
Vector	$O(n)$	$O(1)$	$O(n)$
Linked List	$O(n)$	$O(1)$	$O(1)$

Run-times where n is the number of data values that are currently in the structure.

- **Array**. The existing order helps us quickly find where to insert. Does that help with `insert`'s run-time?
- **Array**. Does it matter if you store the highest value at the front or back of the array?
- **Array**. Does it matter if we use linear indexing or circular index? (see previous lecture on queues)
- **Linked list**. The linked structure helps us insert nodes quickly. Does that help with `insert`'s run-time?
- **Linked list**. Does it matter what type of linked list is used?

Priority Queues

Priority Queue

A priority queue is a data structure that supports the following operations.

- `Insert`. Insert data with a given priority. (The priority may or may not be part of the data).
- `Find`. Return the data with the highest priority.
- `Remove`. Remove the data with the highest priority.

These are the minimum requirements.

In practice, priority queues usually contain many more operations.

	<code>Insert</code>	<code>Find</code>	<code>Remove</code>
Array	$O(n)$	$O(1)$	$O(1)$ or $O(n)$
Vector	$O(n)$	$O(1)$	$O(1)$
Linked List	$O(n)$	$O(1)$	$O(1)$


Run-times

We want to improve upon the $O(n)$ -time run-time for `Insert`.

- $O(1)$ -time for every operation would be fantastic, but perhaps unrealistic.
- $O(\log(n))$ -time is a realistic goal for every operation. How can halving help us?



Heaps

 heap

/hēp/

noun

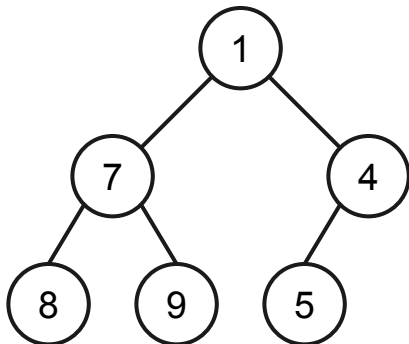
1. a disorderly collection of objects placed haphazardly on top of each other.
"a heap of cardboard boxes"

Similar: [pile](#) [stack](#) [mass](#) [mound](#) [mountain](#) [quantity](#) [load](#) [lot](#) [▼](#)

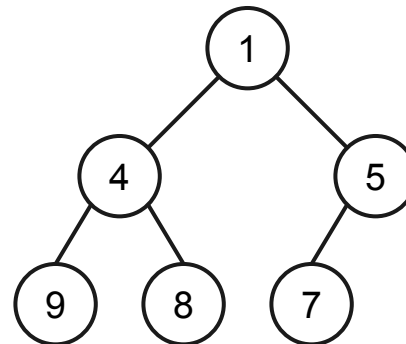
Binary Heaps

A *binary heap* is a binary tree whose node values satisfy two rules:

1. The value of a node is less than or equal to the values of its children.
2. The binary tree is *full* meaning that all levels except the bottom are full and the bottom level has all of the nodes as far to the left as is possible.



A binary heap.



Another binary heap with the same data.

We'll implement a binary heap with the following: where n is the number of nodes currently in the heap

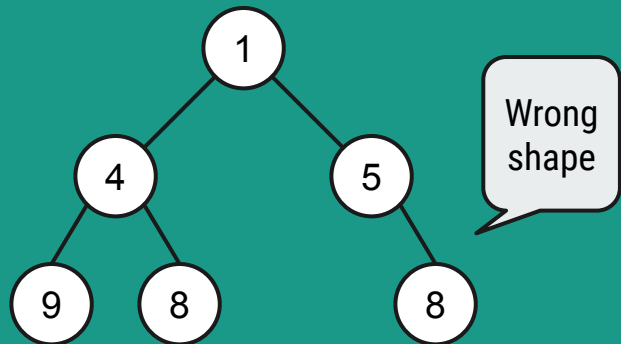
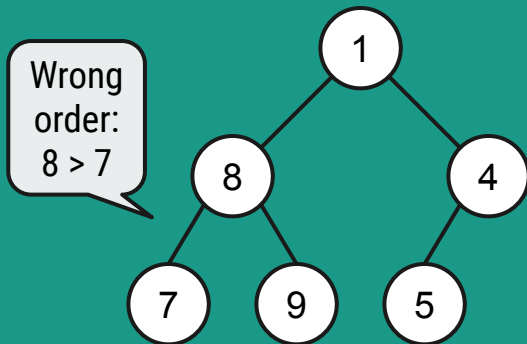
- (a) **Insert**. Insert a new value in $\log(n)$ -time.
- (b) **Find-min**. Return the minimum value in constant-time.
- (c) **Delete-min**. Remove the minimum value in $\log(n)$ -time.

This will allow us to implement a priority queue with $O(\log(n))$ -time operations.

Note: For simplicity we use "values" when discussing heaps; in a priority queue the ordering is based on the "priorities".

Exercise: Identify the Heap(s)?!

Which of the following is a heap?



Operations

Heap Operations

A heap (or binary heap) is a data structure that supports the following operations.

- `insert`. Insert a value.
- `find-min`. Return the minimum value.
- `delete-min`. Remove the minimum value.

These are the minimum requirements.

In practice, heaps often contain many more operations.

	<code>insert</code>	<code>find-min</code>	<code>delete-min</code>
Tree	$O(\log(n))$	$O(1)$	$O(\log(n))$
Array	$O(\log(n))$	$O(1)$	$O(\log(n))$

Our run-time goals for two different implementations of a heap.

A priority queue can use `insert` for `Insert`, `find-min` for `Find`, and `delete-min` for `Remove`. So the heap would spend more time for removing and less time for inserting. Overall, it's an improvement.

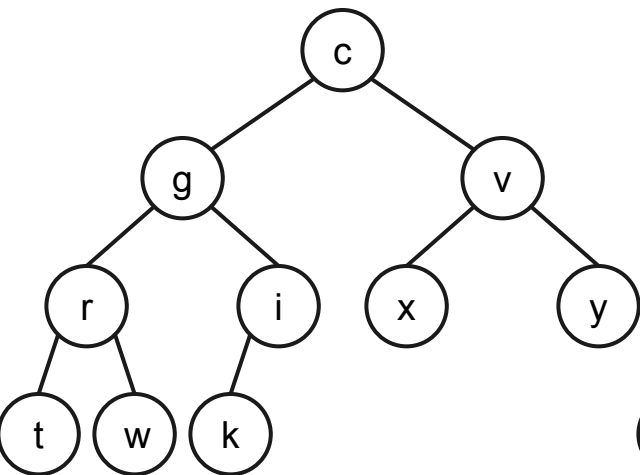
We'll outline how the conceptual steps involved in the operations using a tree structure.

The actual implementation is made easier by instead using an array in a clever way.

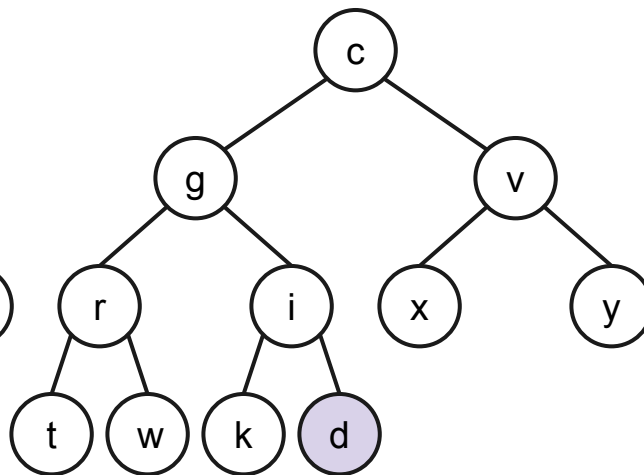
Insert into Heap

How can we implement the `insert` operation in a heap?

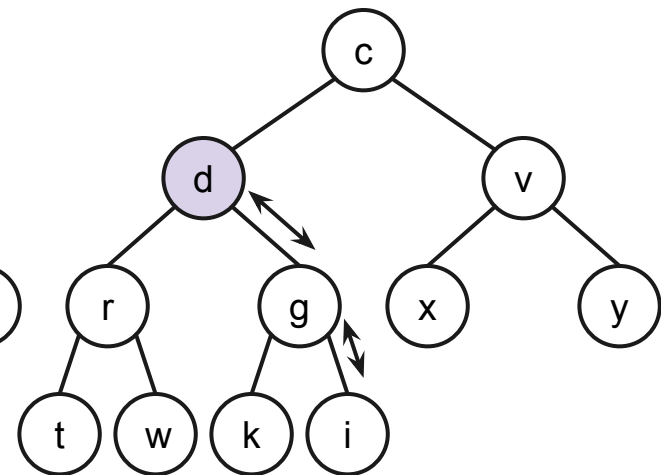
- We start by inserting the value into the next location, so the heap's shape is correct.
- Then we fix the relative orders by “bubbling” the value up as needed.



Before inserting d.



The heap has the right shape but d is not in the right place.



Fix the heap by bubbling upward.
Notice that each swap doesn't cause other ordering problems.

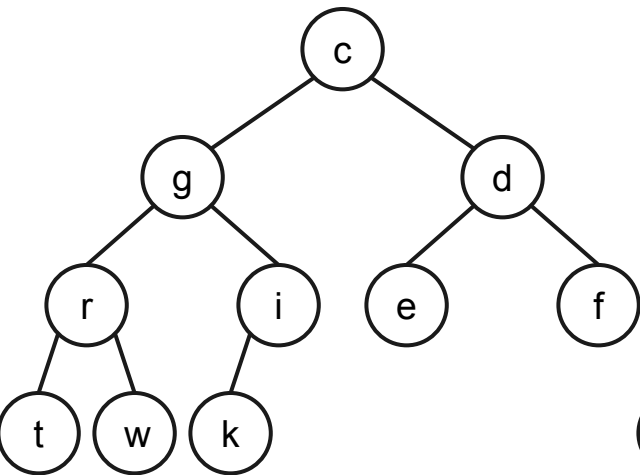
The `insert` operation can be implemented in worst-case $O(\log n)$ -time.

- The heap always has $\log n$ height, where n is the number of elements currently in the heap.
- We'd still need to figure out exactly how to store and modify this type of tree structure.

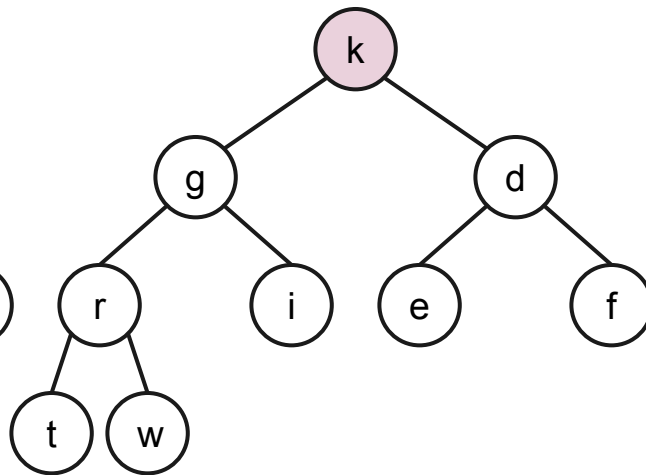
Deletion from Heap

How can we implement the `delete-min` operation in a heap? Note that deleting its last value is easy.

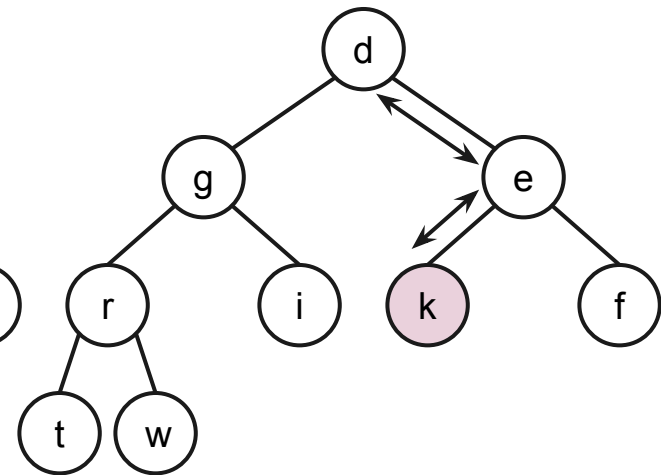
- We swap the top value with the last value, then delete it. The heap is in the correct shape.
- Then we fix the relative orders by “bubbling” the swapped value down as needed.



Before deleting `c`.
We will swap `c` and `k`, then delete `c`.



The heap has the right shape,
but `k` is not in the correct location.



Fix the heap by bubbling `k` down as needed.
Only swap with the smaller child.

The `delete-min` operation can be implemented in worst-case $O(\log n)$ -time.

- The heap always has $\log n$ height, where n is the number of elements currently in the heap.
- We'd still need to figure out exactly how to store and modify this type of tree structure.

Tree Implementation

We conceptualized a heap using a tree-like structure. and our approach to the `insert` and `delete-min` operations are based on this structure.

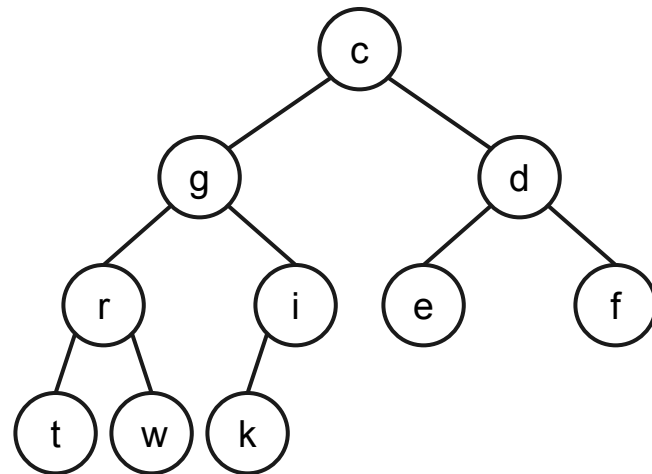
It is possible to implement a heap using this tree-like structure. The implementation uses nodes and links (i.e., references) in a similar manner to linked lists.

Each node object contains the following information:

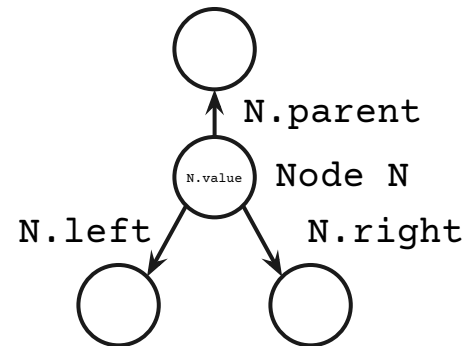
- Its value.
- A reference to its parent.
- A reference to its left-child.
- A reference to its right-child.

We also maintain a reference to the heap's top node.

- This is often referred to as the *root*.
- It is the only node whose parent is set to `null`.



How do we implement this structure?



Each node contains a value and references to its parent, left-child, and right-child.

Array Implementation

Array Implementation

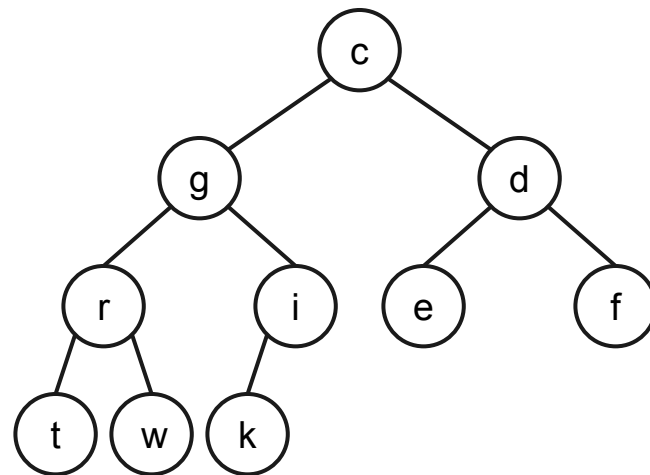
We conceptualized a heap using a tree-like structure, and our approach to the `insert` and `delete-min` operations are based on this structure.

Could implement a heap with an array? Links become implicit.

Goals and requirements for an array implementation:

- Simpler than the tree implementation.
- Navigate the structure with the same speed.
 - From a given node in the tree structure, we can access the parent, as well as the left-child and right-child, in $O(1)$ -time by using the associated links.
 - From a given index in the array, we would need to be able to access the corresponding indices of the parent, as well as the left-child and right-child in $O(1)$ -time.
- Must avoid additional work (e.g., moving values).

We'll consider two ways of organizing the array.



How can we store the heap's values in an array?
What order should they be placed in?

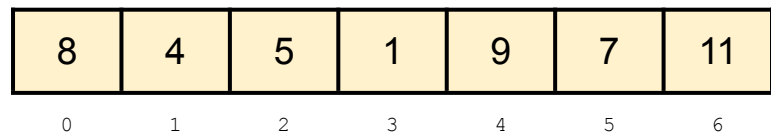
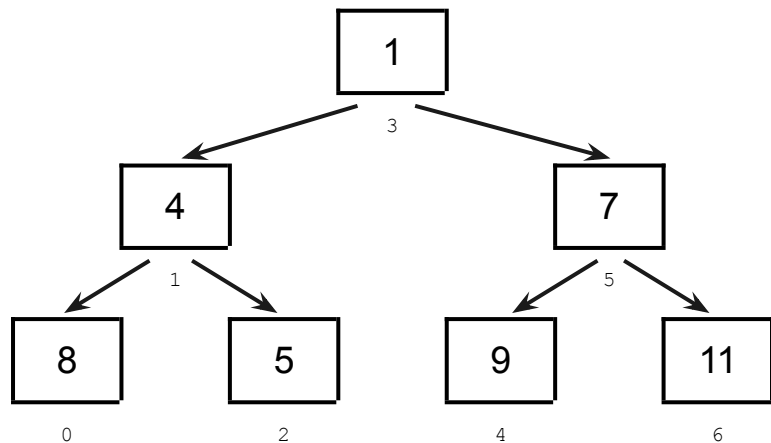
t	r	w	g	i	k	c	e	d	f	...
---	---	---	---	---	---	---	---	---	---	-----

c	g	d	r	i	e	f	t	w	k	...
---	---	---	---	---	---	---	---	---	---	-----

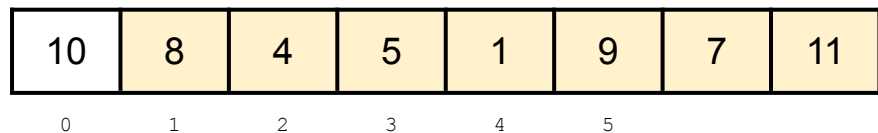
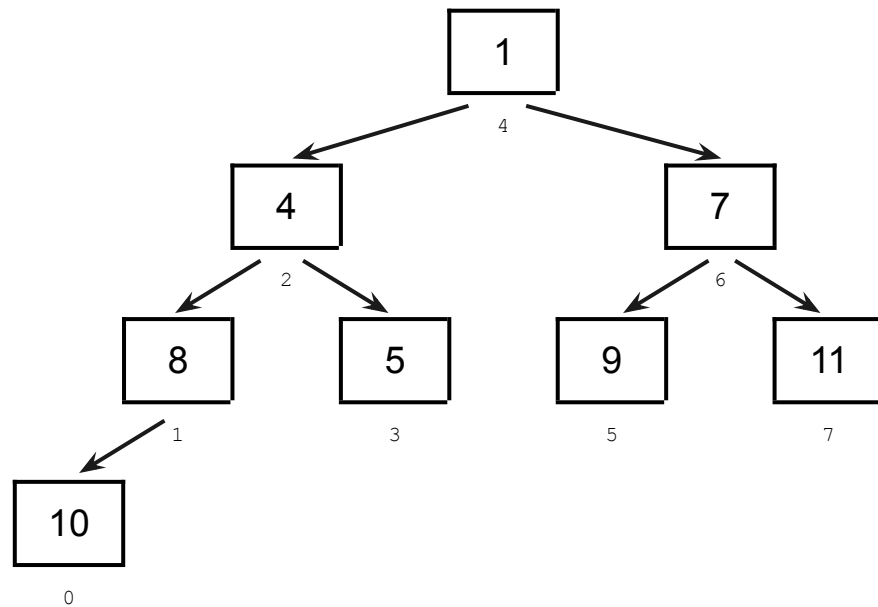
Two different ways of organizing the array.

Array Organization #1: Squish Downard (Incorrect)

Let's first try organizing the array by squishing the tree's nodes downward into the array.



A heap of $n = 7$ values stored in an array with implicit links.

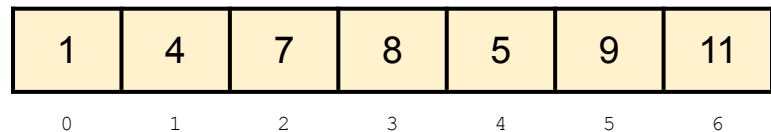
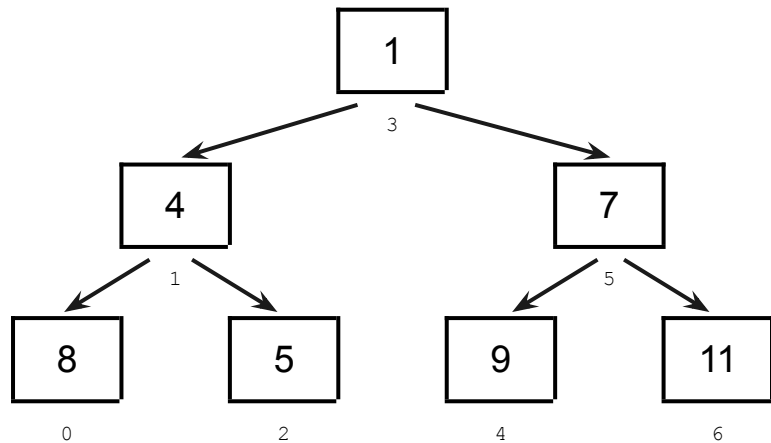


Adding one more value causes every value to move.

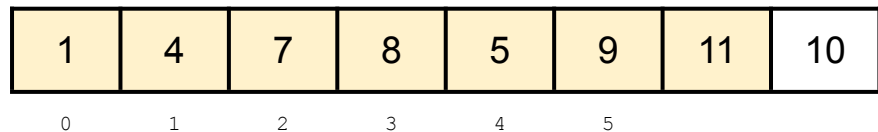
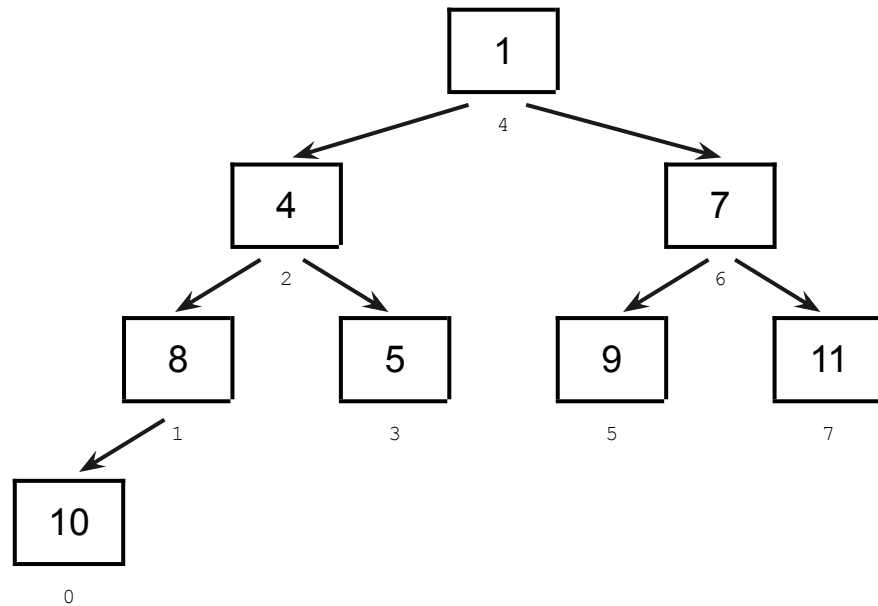
Bad news: When the heap grows from size $2^n - 1$ to 2^n every value in the array may need to move.

Array Organization #2: Top-to-Bottom Left-to-Right (Correct)

Next let's try organizing the array row-by-row from top-to-bottom and left-to-right on each row.



A heap of $n = 7$ values stored in an array with implicit links.



Adding one more value doesn't force any values to move.

Good news: There is no added work associated with adding values. Can we navigate efficiently?

Discussion: Navigating the Array by Computing Parent and Child Indices

To actually implement these heap operations with an array, we need to be able to (efficiently) convert between the indices of each node and its parent / children.

If the current index is j , then what is the index of the following:

- Its left-child?
- Its right-child?
- Its parent?

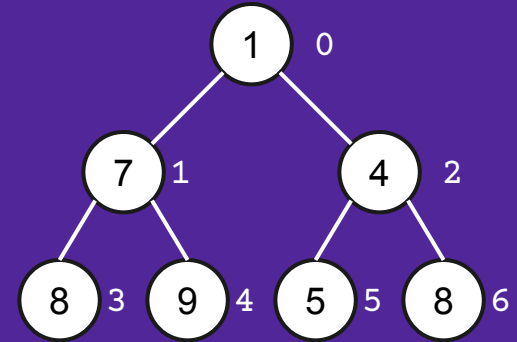


Think about this for 1 minutes.

Then discuss it with your neighbor for 1 minute.

Answers:

- The left-child of index j is $2j + 1$. For example, the left-child of index 2 is index $2j + 1 = 5$.
- The right-child of index j is $2j + 2$. For example, the right-child of index 2 is index $2j + 2 = 6$.
- The parent of index j is index $\lfloor (j - 1) / 2 \rfloor$ where $\lfloor \cdot \rfloor$ denotes floor (i.e. round down as in Java). For example, the parent of index 6 is $\lfloor (j - 1) / 2 \rfloor = \lfloor (6 - 1) / 2 \rfloor = \lfloor 5 / 2 \rfloor = 2$.



A binary heap and its array indices.

Applications

Applications

Heaps have many applications including:

1. Heap sort in $O(n \log(n))$ -time.
2. Efficient implementation of priority queues.