CSE 3318

Week of 07/31/2023

Instructor : Donna French

```
Pichu|1.00|4.4|B|Tiny Mouse|Static|1|172
Pikachu|1.04|13.2|B|Mouse|Static|2|025
Raichu|2.07|66.1|B|Mouse|Static|3|026
Charmander|2.00|18.7|B|Lizard|Blaze|1|004
Charmeleon|3.7|41.9|B|Flame|Blaze|2||005
Charizard|5.7|199.5|B|Flame|Blaze|3|006
Squirtle|1.08|19.8|B|Tiny Turtle|Torrent|1|007
Wartortle|3.03|49.6|B|Turtle|Torrent|2|008
Blastoise|5.03|188.5|B|Shellfish|Torrent|3|009
Igglybuff|1.00|2.2|B|Balloon|Cute Charm, Competitive|1|174
Jigglypuff|1.08|12.1|B|Balloon|Cute Charm,Competitive|2|039
Wigglytuff|3.03|26.5|B|Balloon|Cute Charm,Competitive|3|040
Ponyta|3.03|66.1|B|Fire Horse|Run Away, Flash Fire|1|077
Rapidash|5.07|209.4|B|Fire Horse|Run Away, Flash Fire|2|078
Psyduck|2.07|43.2|B|Duck|Damp, Cloud Nine|1|054
Golduck|5.07|168.9|B|Duck|Damp. Cloud Nine|2|055
Vaporeon|3.03|63.9|B|Bubble Jet|Water Absorb|2|134
Eevee|1.00|14.3|B|Evolution|Run Away, Adaptability|1|133
Jolteon|2.07|54.0|B|Lightning|Volt Absorb|3|135
Meowth|1.04|9.2|B|Scratch Cat|Pickup, Technician|1|052
Persian|3.03|70.5|B|Classy Cat|Technician, Limber|2|053
Bulbasaur|2.04|15.2|B|Seed|Overgrow|1|001
Ivysaur|3.03|28.7|B|Seed|Overgrow|2|002
Venusaur|6.07|220.5|B|Seed|Overgrow|3|003
Smoochum|1.04|13.2|F|Kiss|Oblivious, Forewarn|1|238
Jynx|4.07|89.5|F|Human Shape|Oblivious, Forewarn|2|124
Growlithe|2.04|41.9|B|Puppy|Intimidate, Flash Fire|1|058
Arcanine|6.03|341.7|B|Legendary|Intimidate, Flash Fire|2|059
Finneon|1.04|15.4|B|Wing Fish|Swift Swim, Storm Drain|1|456
Lumineon|3.11|52.9|B|Neon|Swift Swim, Storm Drain|2|457
Ditto|1.00|8.8|U|Transform|Limber|0|132
Nidoran|1.04|15.4|F|Poison Pin|Poison Point, Rivalry|1|029
Nidorina|2.07|44.1|F|Poison Pen|Poison Point, Rivalry|2|030
```

```c
typedef struct Pokemon
{
    char *name;
    float height;
    float weight;
    char gender;
    char *category;
    char *abilities;
    int evolution;
    int national_pokedex_number;
    struct Pokemon *next_ptr;
}
POKEMON;
```

Coding Assignment 6

CSE 3318

Coding Assignment 6 is over creating a hash table and using it to do the 3 basic dictionary actions - insert, delete and search.

Your hash table will be created using an integer array with separate chaining to resolve conflicts.

Step 1

For this assignment, you need to choose a subject for your dictionary. For my version of the assignment, I chose Pokémon – you may not also use Pokémon – you need to pick your own subject. Pick a subject that allows you to gather enough information to fulfill the minimum requirements of the assignment.

Structure

You will need to create a `struct` to hold each entry of the dictionary. My Pokémon example is

```
typedef struct Pokemon
{
    char *name;
    float height;
    float weight;
    char gender;
    char *category;
    char *abilities;
    int evolution;
    int national_pokedex_number;
    struct Pokemon *next_ptr;
}
POKEMON;
```

In this `struct`, the `char` pointers are used instead of `char` arrays since it is not known how much space those entries will need. Memory to hold those fields will be dynamically allocated using `malloc()` and the pointer returned by `malloc()` will be stored in the struct.

For your assignment, you will need a `struct` with a minimum of 5 fields – the parts shown in **bold** are required.

```
typedef struct YourStructName
{
   char *chararrayfield;
   float floatfield;
   char charfield;
   int intfield;
   struct YourStructName *next_ptr;
}
YOURSTRUCTNAME;
```

Your `struct` must be `typedef`ed and you will be expect to use the `typedef` throughout the program. You may add more fields if you want your dictionary to hold more information. Your member names must adequately describe the data being stored in the variable. Put your typedef in a text file and name it `Code6_xxxxxxxxxx_struct.txt` where `xxxxxxxxxx` is your student id.

Input File

Your input file must contain a minimum of 30 pipe delimited records that represent your dictionary entries. Here's the first 10 lines of my Pokédex file.

```
Pichu|1.00|4.4|B|Tiny Mouse|Static|1|172
Pikachu|1.04|13.2|B|Mouse|Static|2|025
Raichu|2.07|66.1|B|Mouse|Static|3|026
Charmander|2.00|18.7|B|Lizard|Blaze|1|004
Charmeleon|3.7|41.9|B|Flame|Blaze|2||005
Charizard|5.7|199.5|B|Flame|Blaze|3|006
Squirtle|1.08|19.8|B|Tiny Turtle|Torrent|1|007
Wartortle|3.03|49.6|B|Turtle|Torrent|2|008
Blastoise|5.03|188.5|B|Shellfish|Torrent|3|009
Igglybuff|1.00|2.2|B|Balloon|Cute Charm, Competitive|1|174
```

The fields in each entry must be pipe delimited and must be listed in the same order as your `struct`. Save this as a text file named `Code6_xxxxxxxxxx_InputFile.txt` where `xxxxxxxxxx` is your student id.

Hashing Function

Pick one of the methods discussed in class for creating your hash function (numeric keys, alphanumeric keys, folding). Pick which field(s) from your `struct` will be used for the hash function. Remember that whatever method you choose should result in a value that you can MOD with the hash table define to get an index that will always be within your array.

Write a small C program that prompts for input and calls your hash function and prints the result of the hash. You are proving that your hash function will work. Here's an example – your version may be slightly different depending on your input. THIS CODE IS ONLY FOR SHOWING THAT YOUR HASH FUNCTION WORKS. It is ONLY an example – modify it as needed to show that your hash function will work.

```
#include <stdio.h>
#define HASHTABLESIZE 10
int MyHashFunction(pass in value being used in hash)
{
   perform hash method
   return value % HASHTABLESIZE;
}

int main(void)
{
   char HashValue[20];
   printf("Enter value ");
   scanf("%s", HashValue);
   printf("The hash value for %s is %d\n”, HashValue, MyHashFunction(HashValue));
   return 0;
}
```

Compile your version and run and confirm that your function works. Save this code as `Code6_xxxxxxxxxx_HashFunction.c` where `xxxxxxxxxx` is your student id.

Step 2

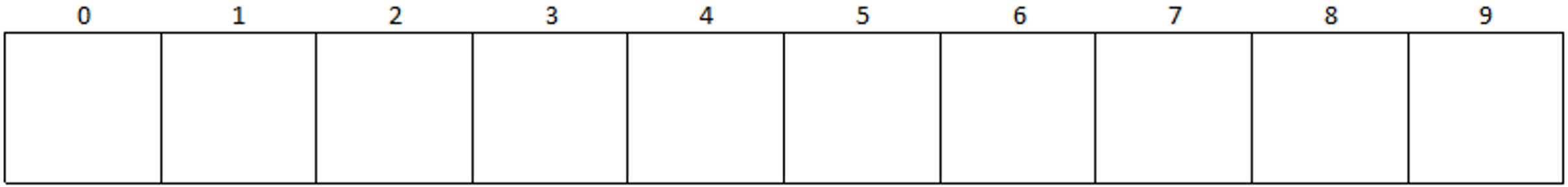
Submit your input file, `struct` and hashing function for approval. Submit these to me by email. If you want to zip the files, you can, but you can just send them as attachments. No pictures or screenshots will be accepted.

Three files must be submitted via email

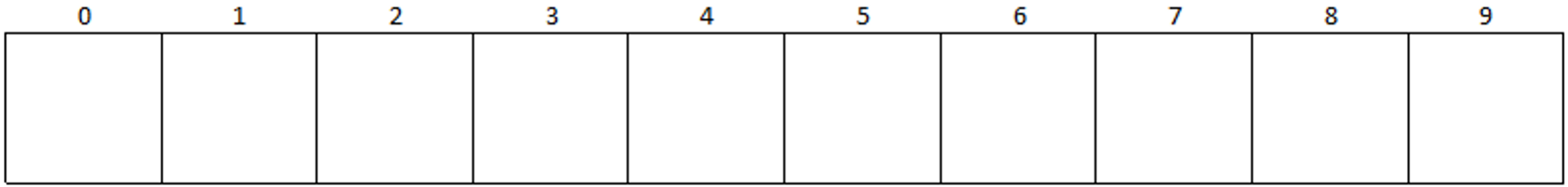
```
Code6_xxxxxxxxxx_struct.txt
Code6_xxxxxxxxxx_InputFile.txt
Code6_xxxxxxxxxx_HashFunction.c
```

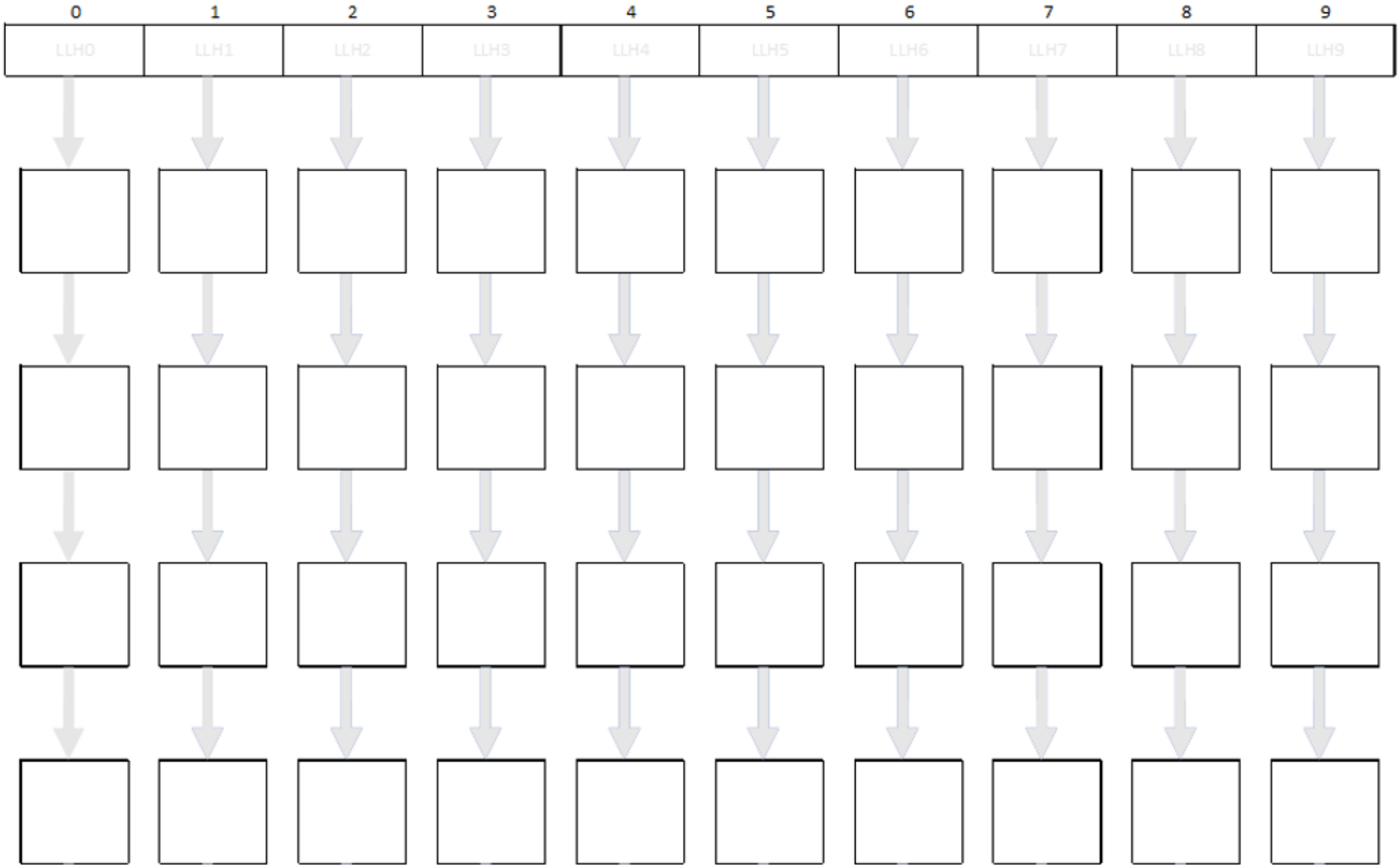
Once you get approval, you will receive the rest of the instructions for the assignment. If you do not get approval, any code you turn in for Coding Assignment 6 will be assigned a grade of 0 regardless.

Part A – Complete the hash version using the same values/Part A data



Part B - Create the hash table using the values from the Part B data. Use linear probing for resolving conflicts.



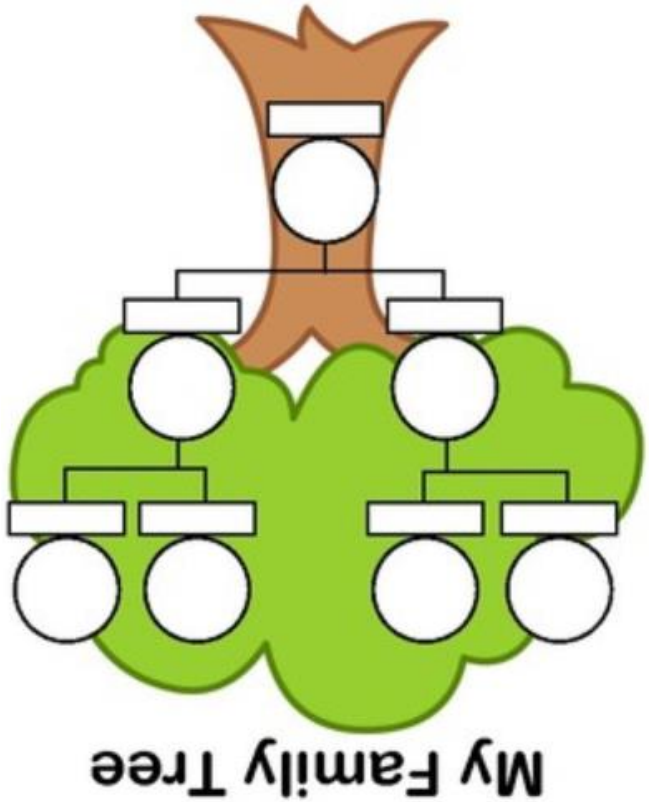

0
1
2
3
4
5
6
7
8
9
LLH0
LLH1
LLH2
LLH3
LLH4
LLH5
LLH6
LLH7
LLH8
LLH9

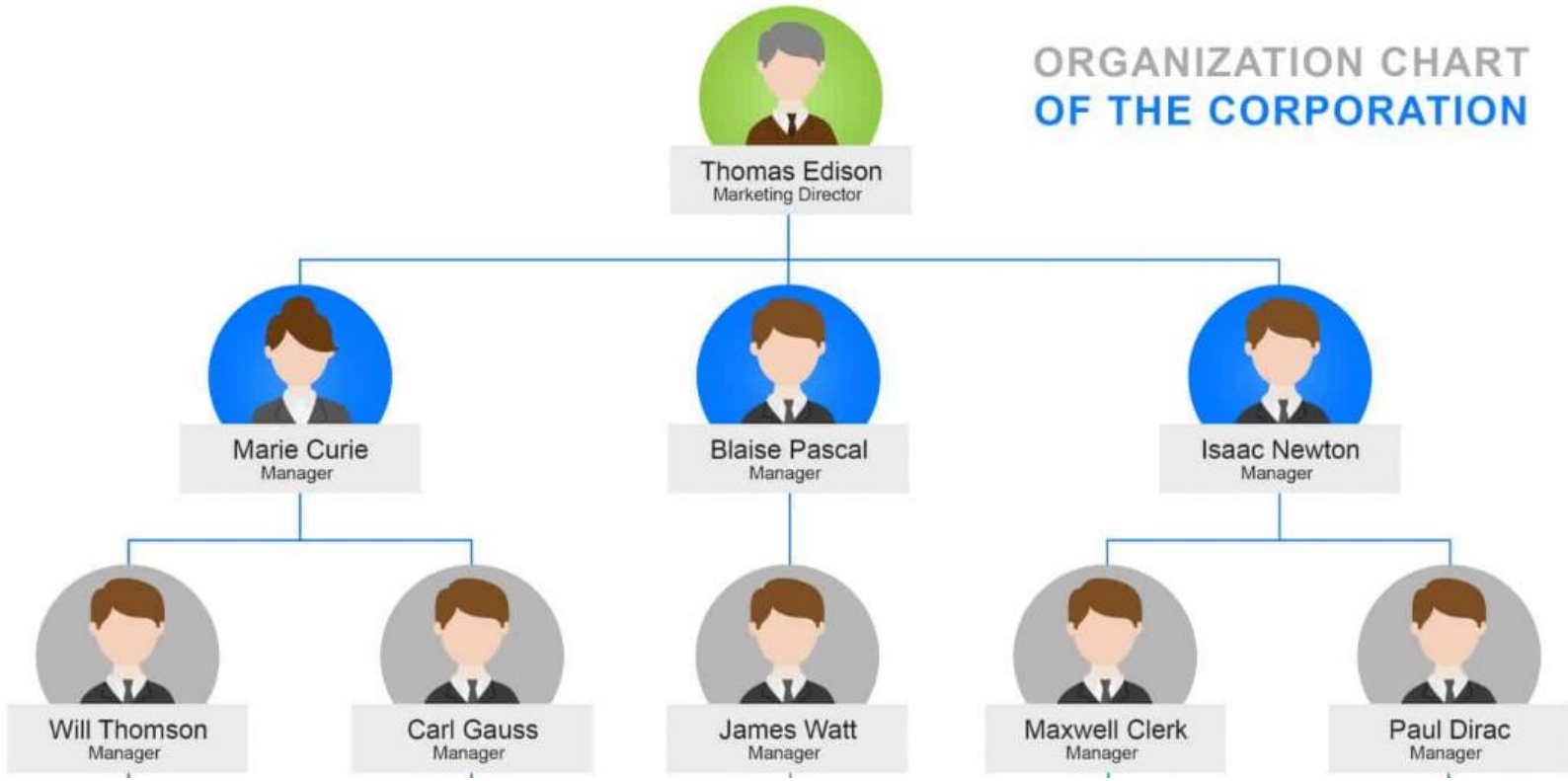
Tree Data Structure

- Linked lists, stacks, and queues are all **linear** structures
  - one node follows another
  - each node contains a pointer to the next node

- Trees are **non-linear** structures
  - more than one node can follow another
  - each node contains pointers to an arbitrary number of nodes
  - the number of nodes that follow can vary from one node to another.

- Trees organize data hierarchically instead of linearly.


My Family Tree


ORGANIZATION CHART
OF THE CORPORATION
Thomas Edison
Marketing Director
Marie Curie
Manager
Blaise Pascal
Manager
Isaac Newton
Manager
Will Thomson
Manager
Carl Gauss
Manager
James Watt
Manager
Maxwell Clerk
Manager
Paul Dirac
Manager

Binary Tree

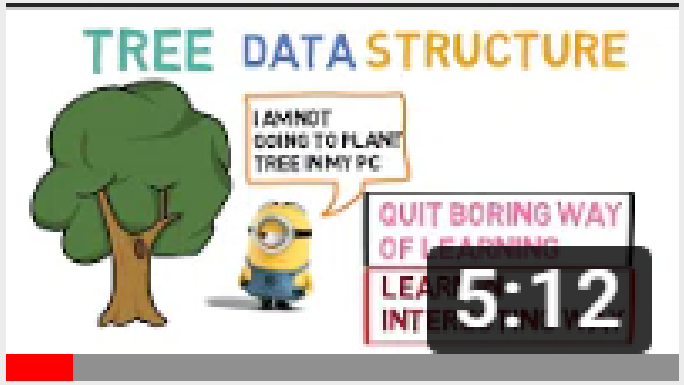
What is a binary tree?

A binary tree is a non-linear tree-like data structure consisting of nodes where each node has up to two child nodes, creating the branches of the tree.

The two children are usually called the left and right nodes.

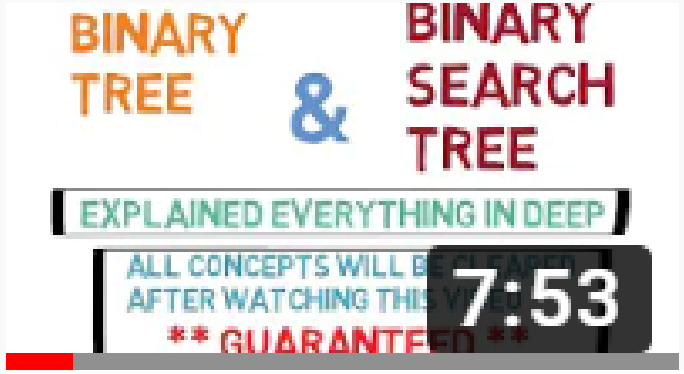
Parent nodes are nodes with children. Parent nodes can be child nodes themselves.

Binary trees are used to implement binary search trees and binary heaps. They are also often used for sorting data as in a heap sort.



Introduction to Tree Data Structure

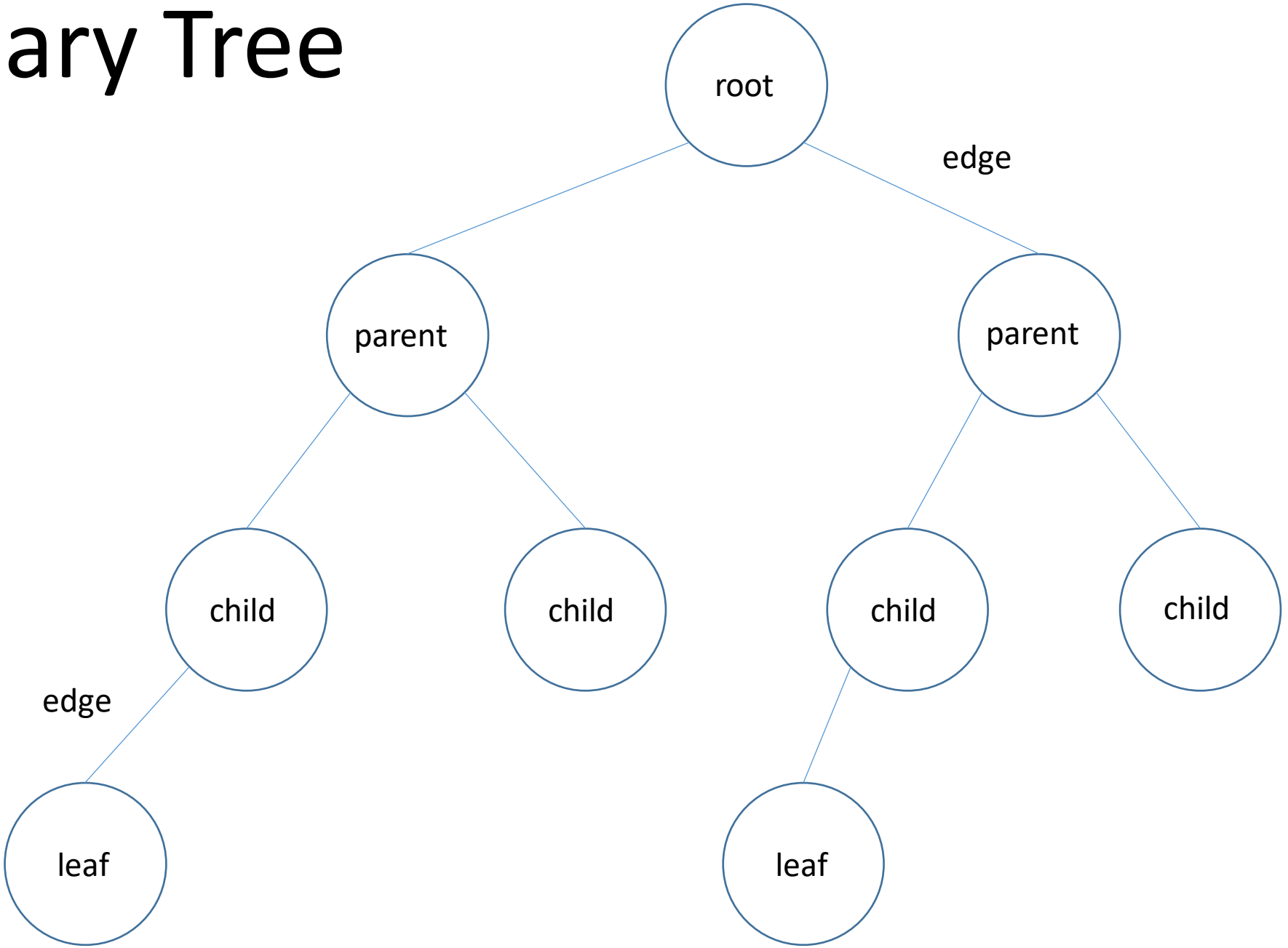
Codearchery



Binary Tree and Binary Search Tree in Data Structure

Codearchery

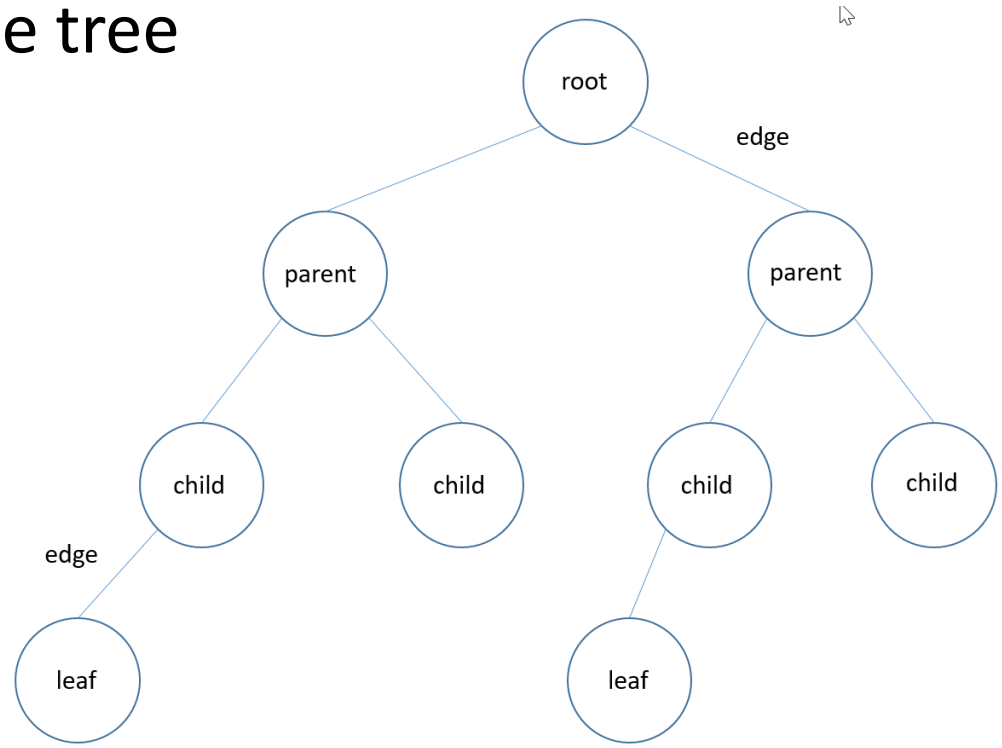
Binary Tree


root
edge
parent
parent
child
child
child
child
edge
leaf
leaf

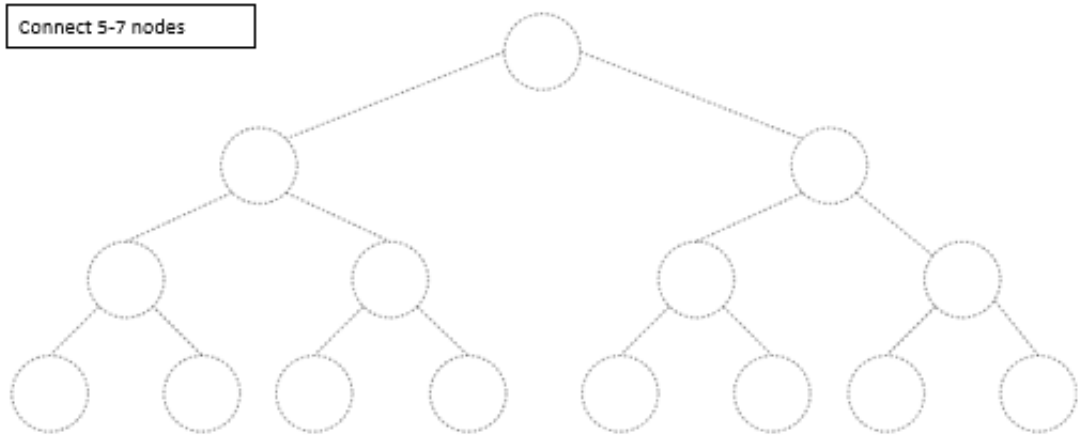
Binary Tree

Tree Vocabulary

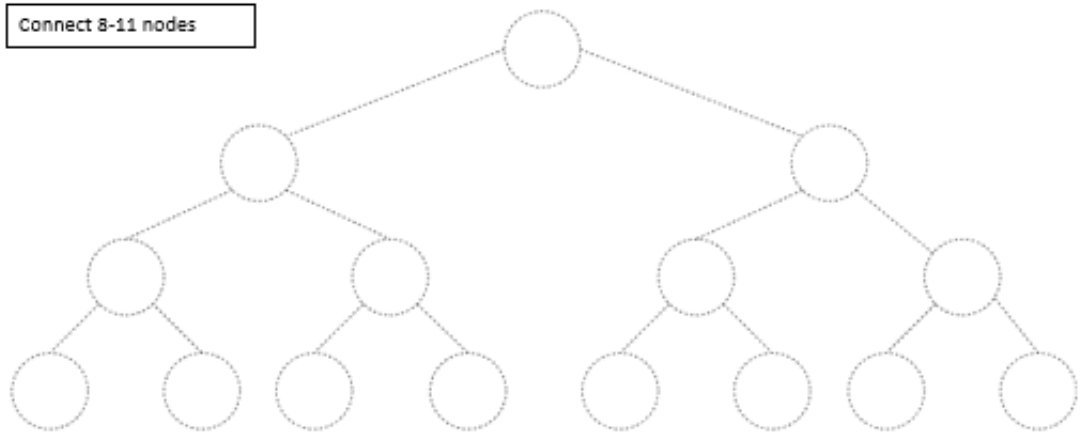
topmost node	root of the tree
node directly under another node	child
node directly above another node	parent
node with no children	leaf
link between two nodes	edge
length of the path from the root	depth
length of the path from the node to the deepest leaf reachable from it	height



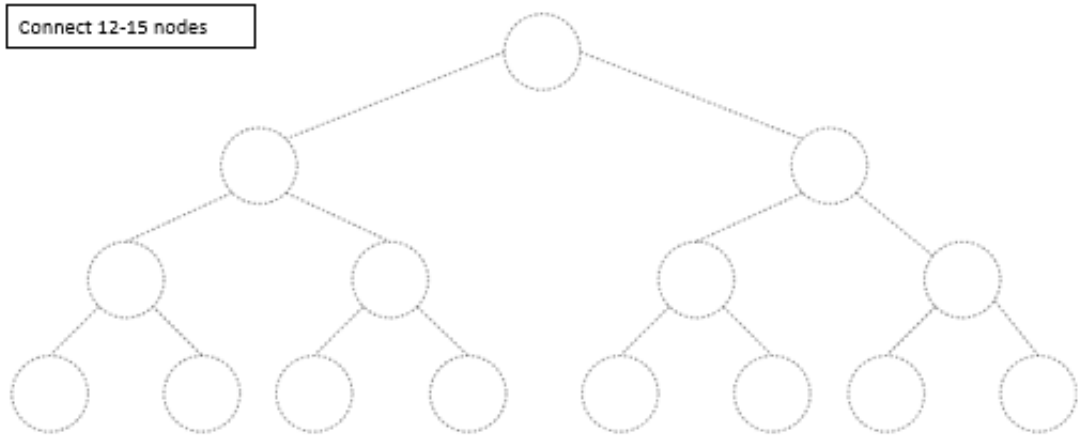
Connect 5-7 nodes



Connect 8-11 nodes



Connect 12-15 nodes



Binary Trees

Before we talk about heaps, let's talk about some versions of binary trees.

Perfect Binary Tree

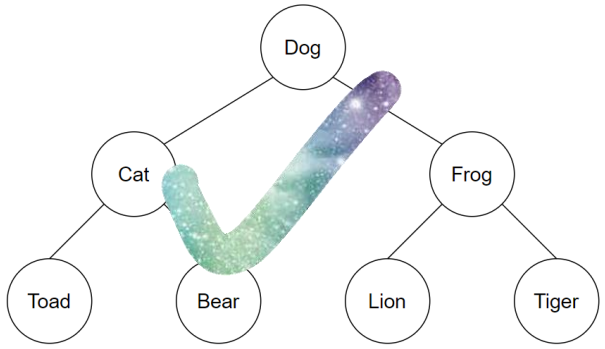
Full Binary Tree

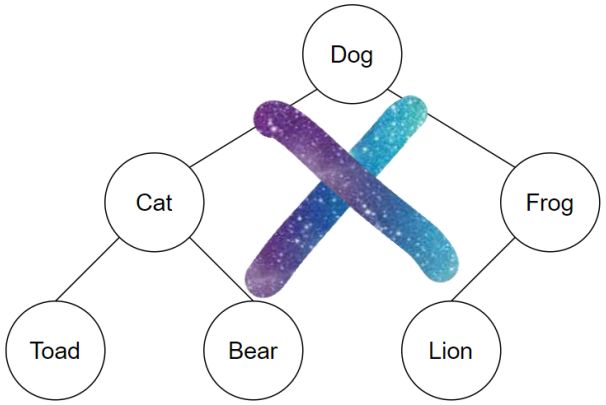
Complete Binary Tree

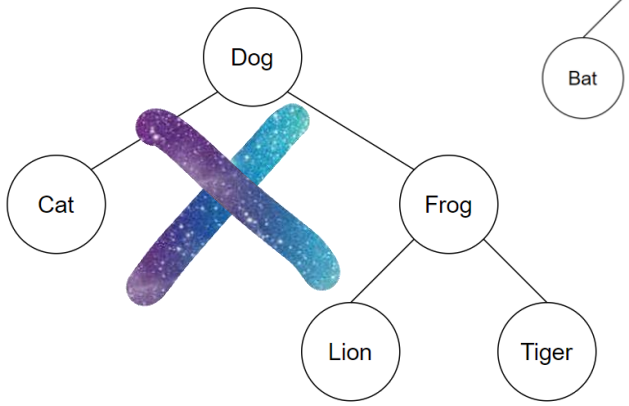
Binary Trees

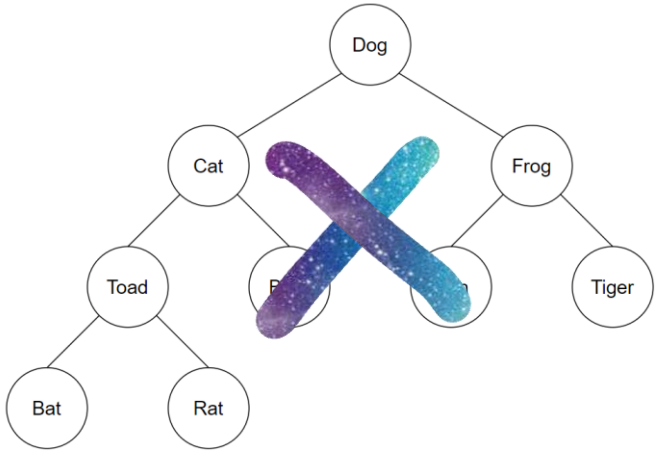
Perfect Binary Tree

A Binary tree is a **Perfect Binary Tree** if all the internal nodes have two children and all leaf nodes are at the same level.







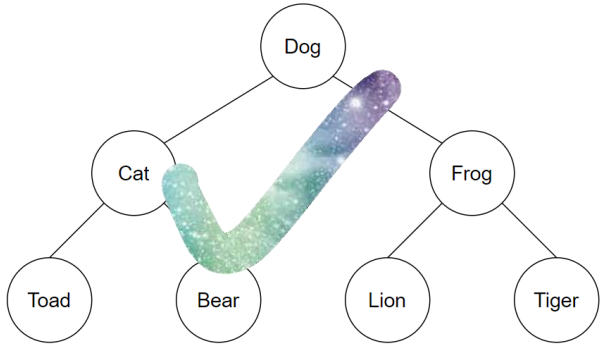


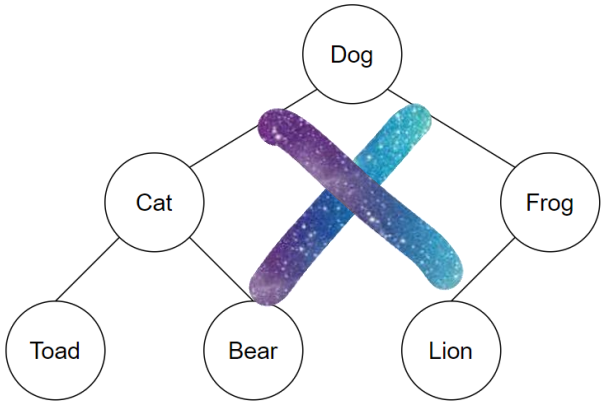
Binary Trees

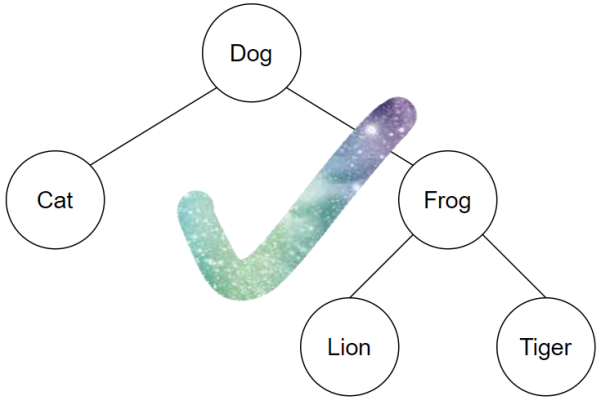
Full Binary Tree

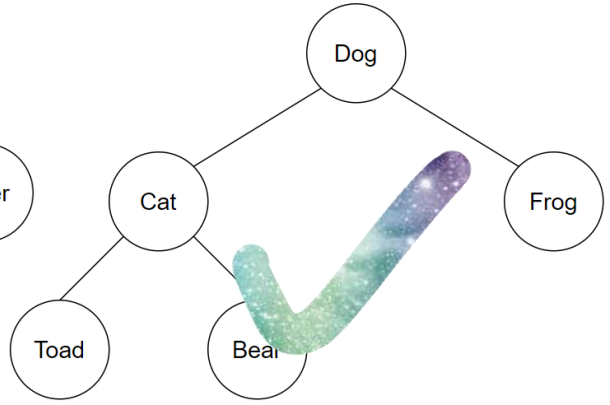
A Binary Tree is a **full binary tree** if every node has 0 or 2 children.

We can also say a full binary tree is a binary tree in which all nodes except leaf nodes have two children.





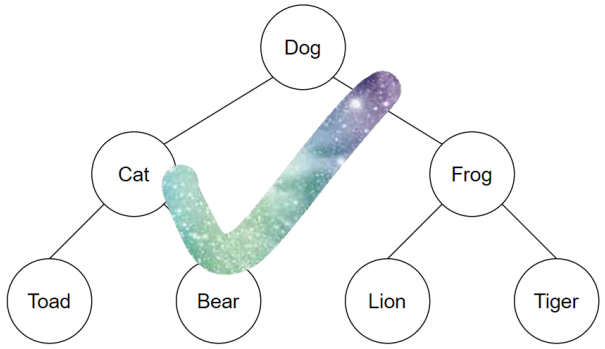


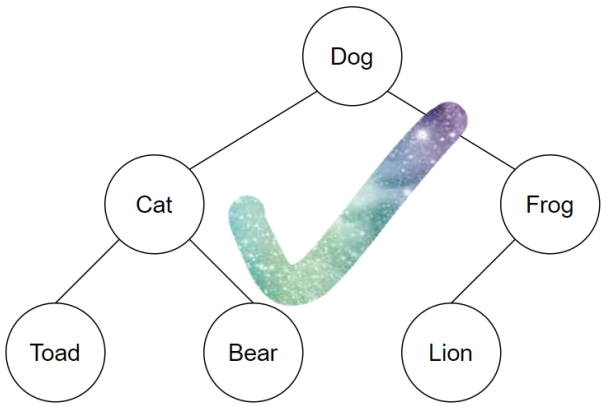


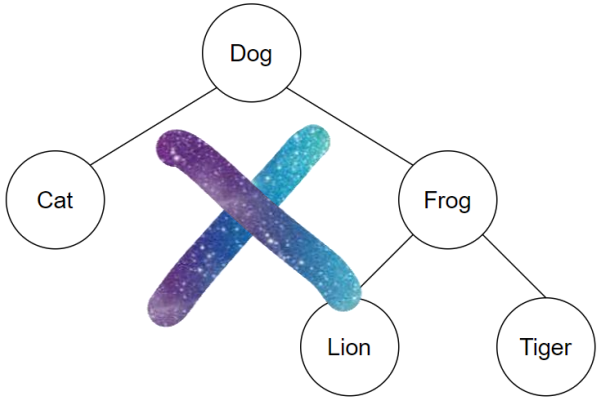
Binary Trees

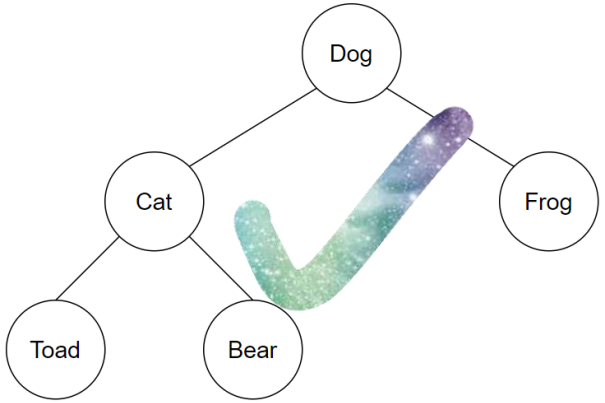
Complete Binary Tree

A Binary Tree is a **Complete Binary Tree** if all the levels are completely filled except possibly the last level and the last level has all keys as left as possible







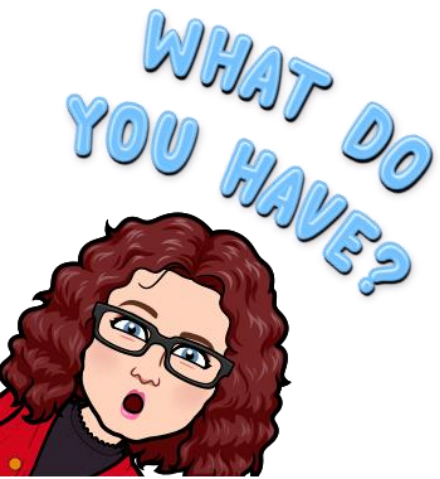


Binary Trees

A Binary Tree is a **complete binary tree** if all the levels are completely filled except possibly the last level and the last level has all keys as left as possible

A Binary Tree is a **full binary tree** if every node has 0 or 2 children.

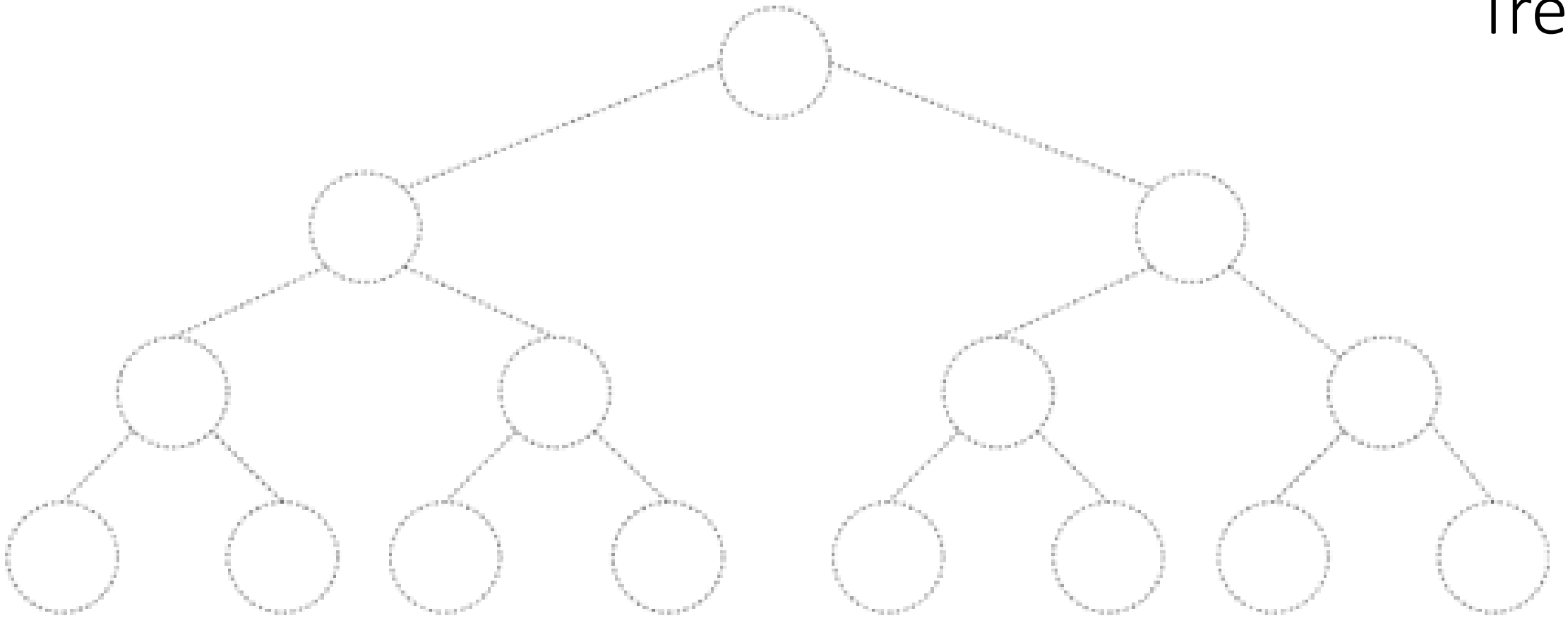
A Binary tree is a **Perfect Binary Tree** if all the internal nodes have two children and all leaf nodes are at the same level.



n = 2-3
n = 4-7
n = 8-15

Levels = 2
Levels = 3
Levels = 4

Complete Binary Tree



Binary Heap

A binary **heap** is a complete binary tree that satisfies the heap property.

Complete Binary Tree - all the levels are completely filled except possibly the last level and the last level has all keys as left as possible

Heap is one of the most efficient implementations of an abstract data type called a priority queue.

A priority queue is an abstract data type similar to regular queue data structure in which each element additionally has a "priority" associated with it.

In a priority queue, an element with high priority is served before an element with low priority.

Priority Queue

Imagine you are an Air Traffic Controller (ATC) working in the control tower of an airport.

Aircraft X is ready to land and the runway is free, so you give them permission to land.

Aircraft Y radios in that they will be arriving at the airport within the next 5 minutes.

You also know that the runway will be occupied/unavailable for any other plane for at least 15 minutes while a plane is landing.

You tell Aircraft Y to go into a holding pattern when they arrive because the runway will be occupied for at least another 10 minutes after their arrival.

You have Aircraft X and Y in a queue.

Priority Queue

If another aircraft arrives, they will be put in the queue behind Aircraft Y and told to maintain a holding pattern while waiting their turn.

Five minutes later, Aircraft Z radios in that they are 7 minutes away from the airport but critically low on fuel due to going around a big storm system.

They can't land immediately because the runway will still be occupied by Aircraft X.

So you put them in the queue because they have to wait on Aircraft X (which is already landing) but they have priority over Aircraft Y even though Aircraft Y arrived first.

Binary Heap

A binary heap is a binary tree with the following properties…

1. Complete tree

2. Heap property which means the binary heap is either a

   Min Heap
   or
   Max Heap

Binary Heap

Min Heap

The value at root of the tree must be smallest value present in the Binary Heap.

This property must be recursively true for all nodes in the binary tree.

Any parent node should be smaller than its child nodes.

Binary Heap

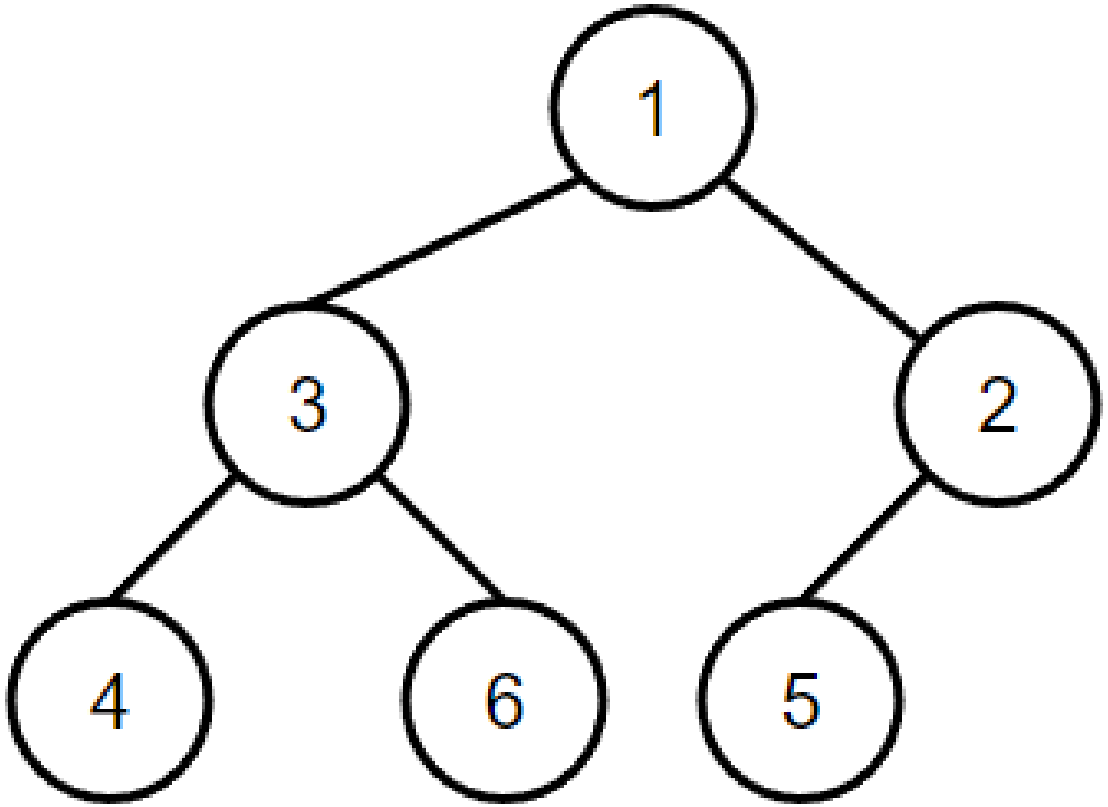
Is this a Min Heap?

1. Is it a complete tree?

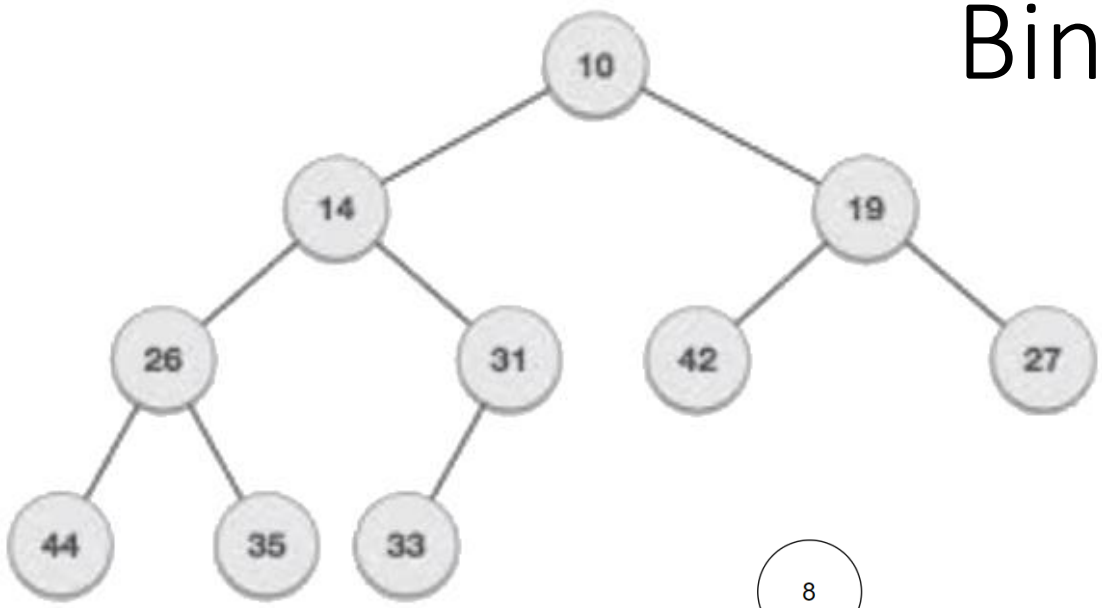
   all the levels are completely filled except possibly the last level and the last level has all keys as left as possible

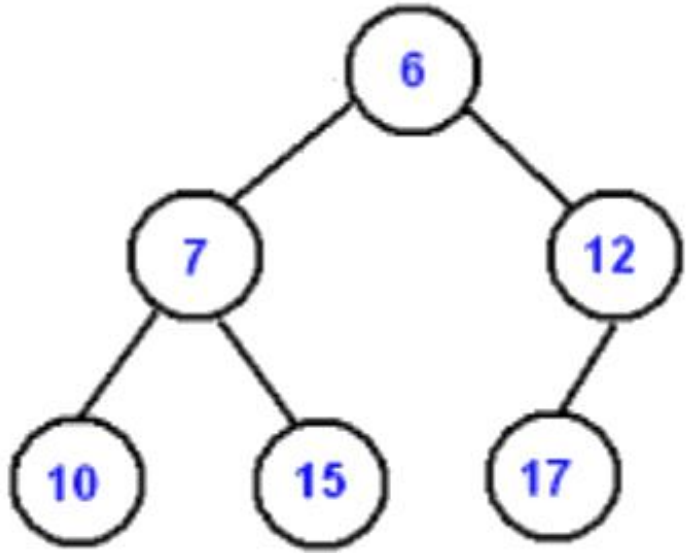
2. Is the value at the root of the tree the smallest value in the heap?

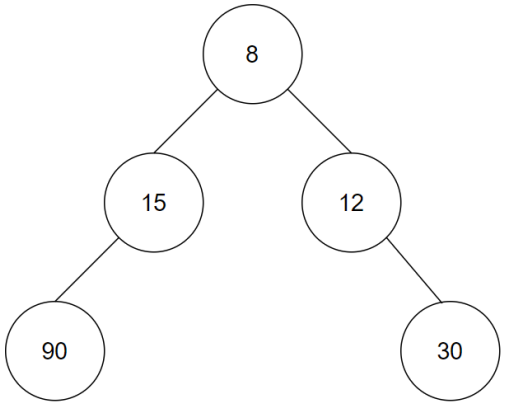
3. Are all parent nodes smaller than their children?

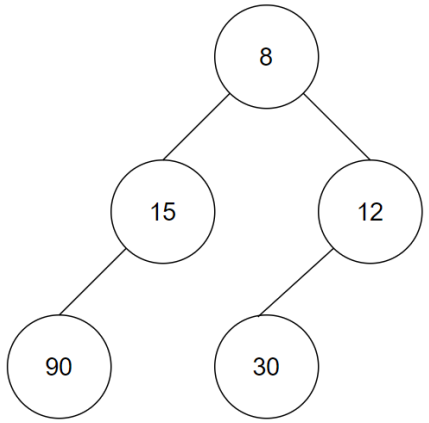


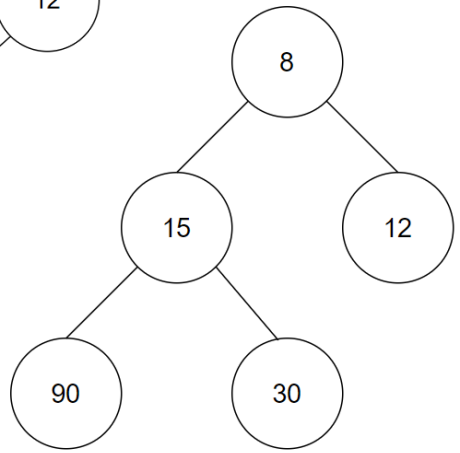
Binary Heap

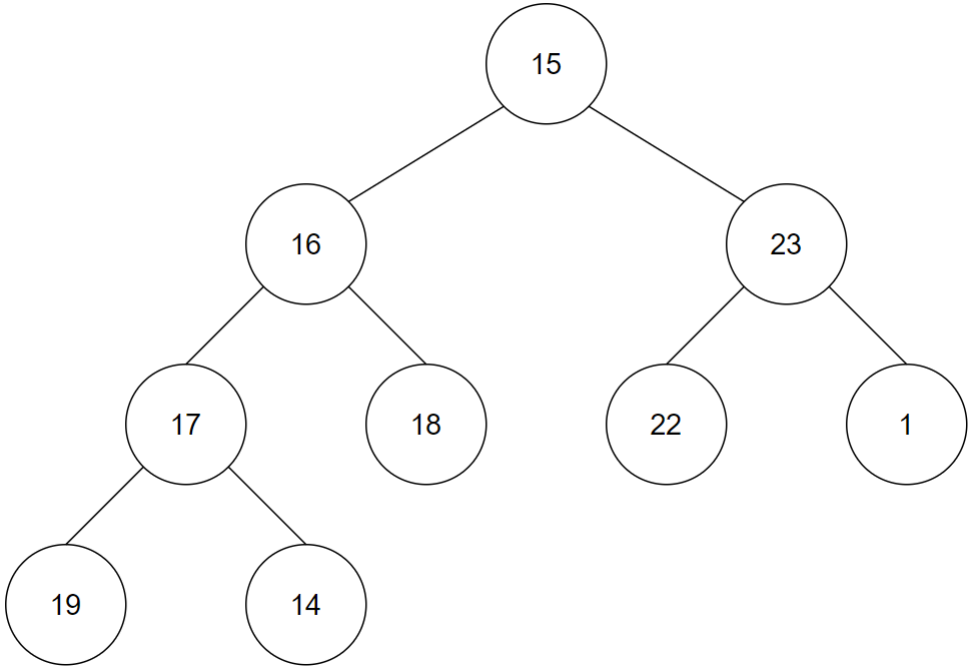

10
14
19
26
31
42
27
44
35
33


6
7
12
10
15
17


8
15
12
90
30


8
15
12
90
30


8
15
12
90
30


15
16
23
17
18
22
1
19
14

Binary Heap

Max Heap

The value at root of the tree must be largest value present in the Binary Heap.

This property must be recursively true for all nodes in the binary tree.

Any parent node should be larger than its child nodes.

Binary Heap

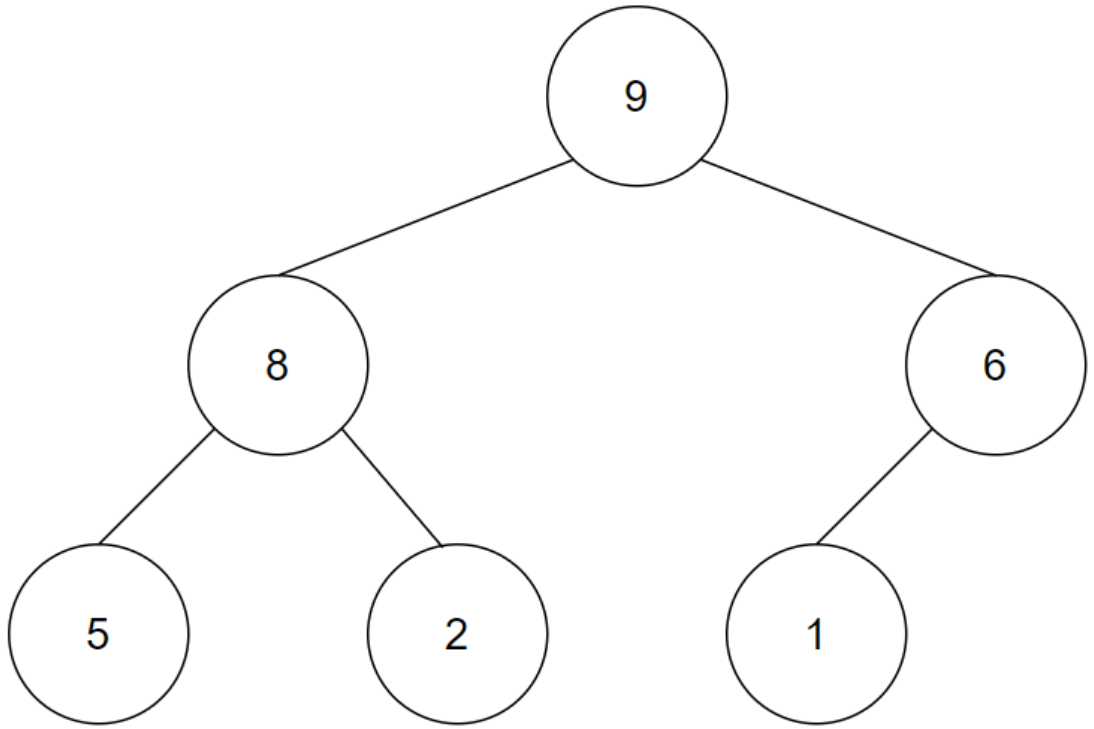
Is this a Max Heap?

1. Is it a complete tree?

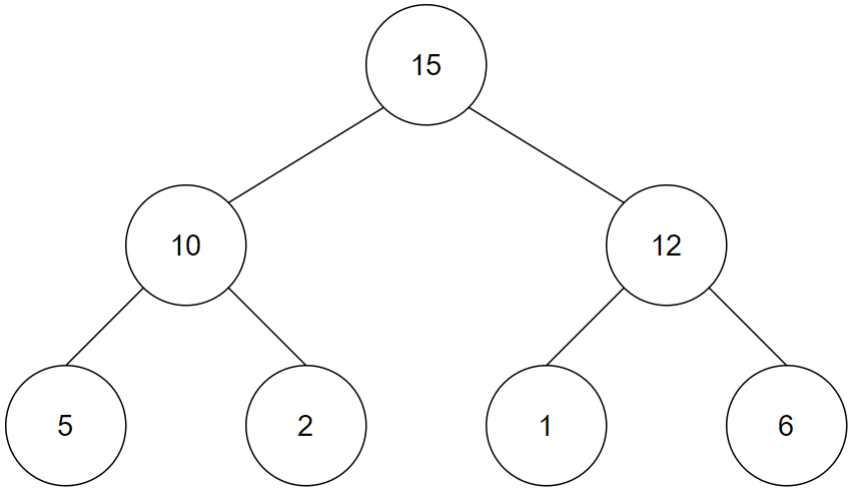
   all the levels are completely filled except possibly the last level and the last level has all keys as left as possible

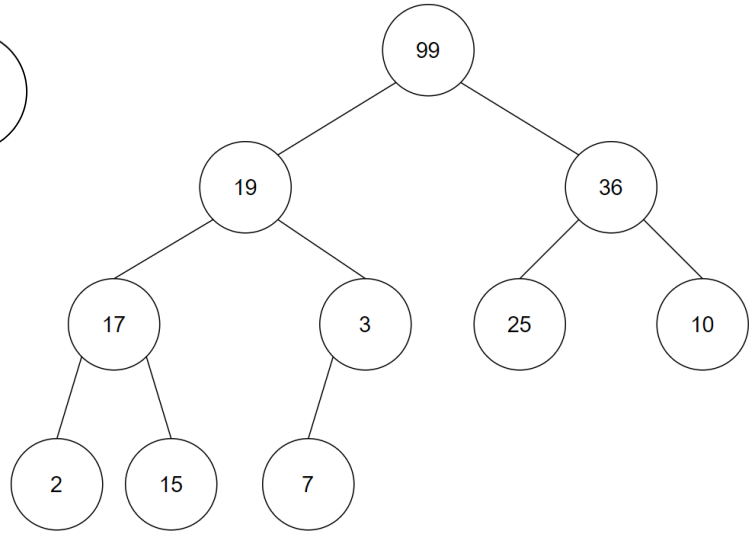
2. Is the value at the root of the tree the largest value in the heap?

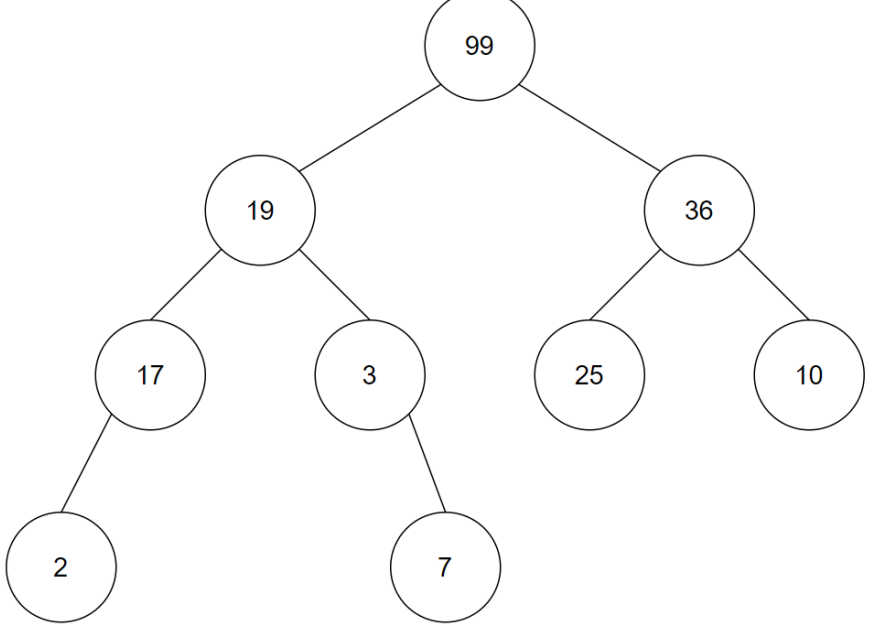
3. Are all parent nodes larger than their children?

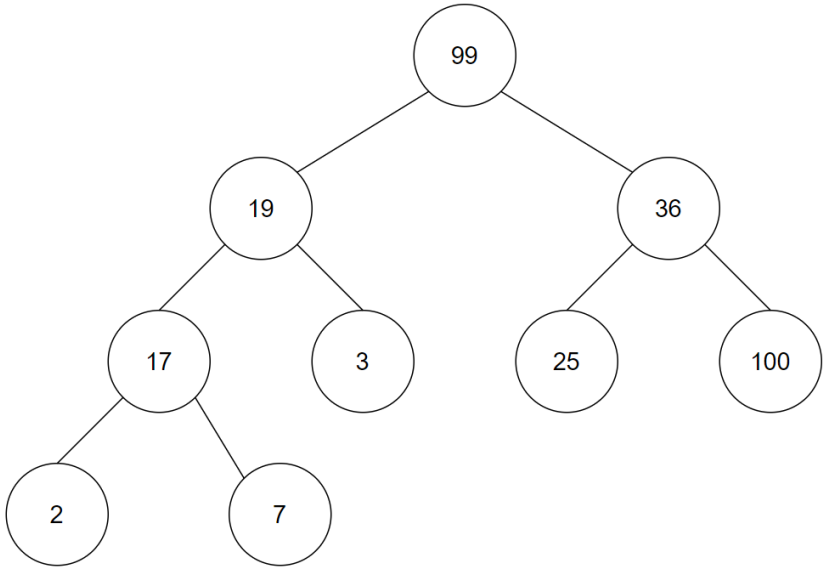


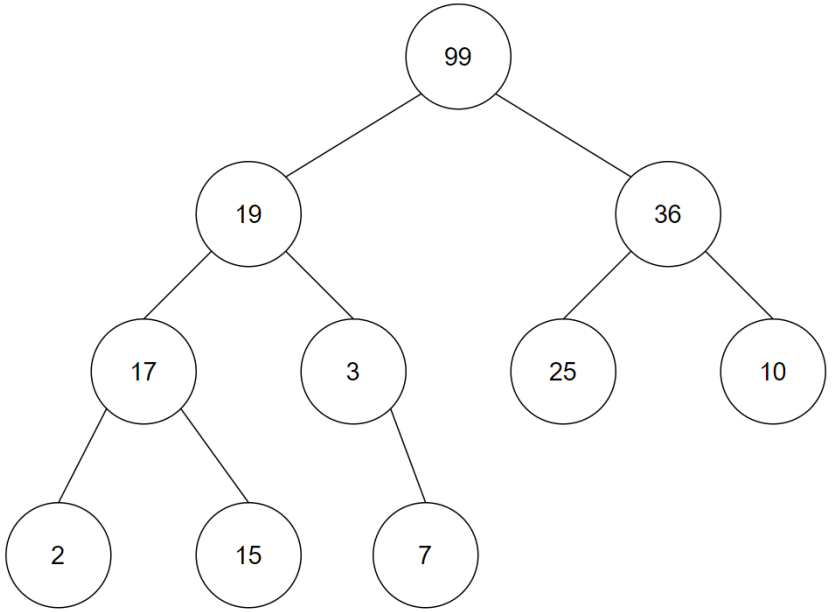
Binary Heap


15
10
12
5
2
1
6


99
19
36
17
3
25
10
2
15
7


99
19
36
17
3
25
10
2
7


99
19
36
17
3
25
100
2
7


99
19
36
17
3
25
10
2
15
7

Binary Heap

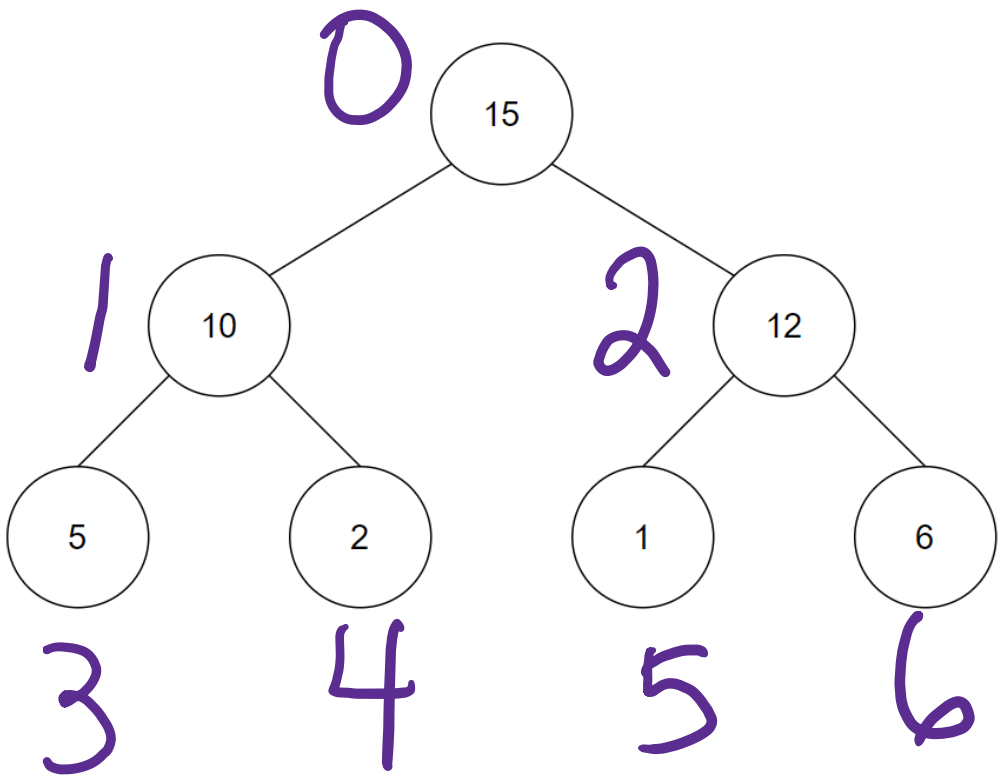
Now, here's the good news.

A binary heap is typically represented by an array.

So how do we turn a tree into an array?

Binary Heap

Let's look at it graphically and then we'll formalize the process.

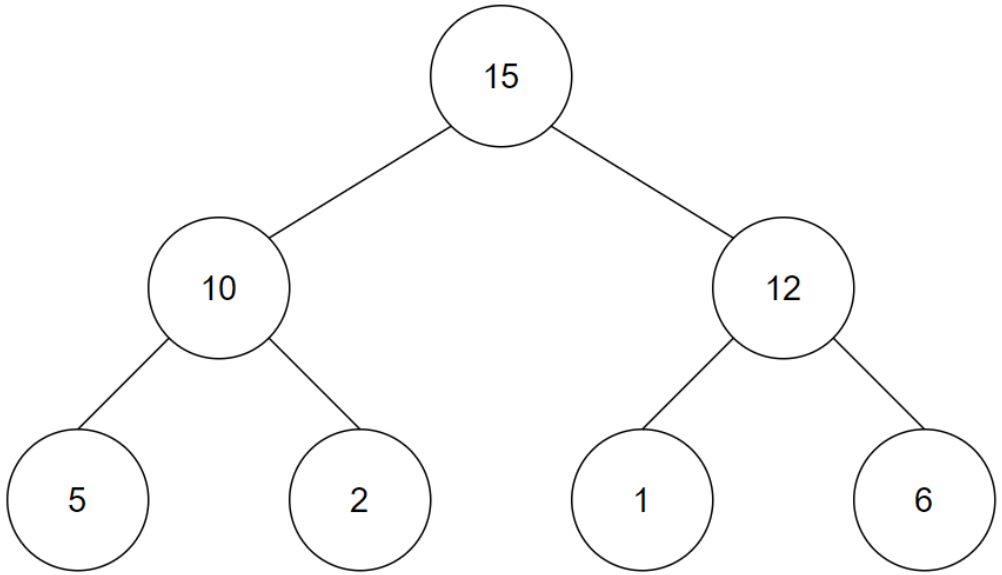


0	1	2	3	4	5	6

Binary Heap

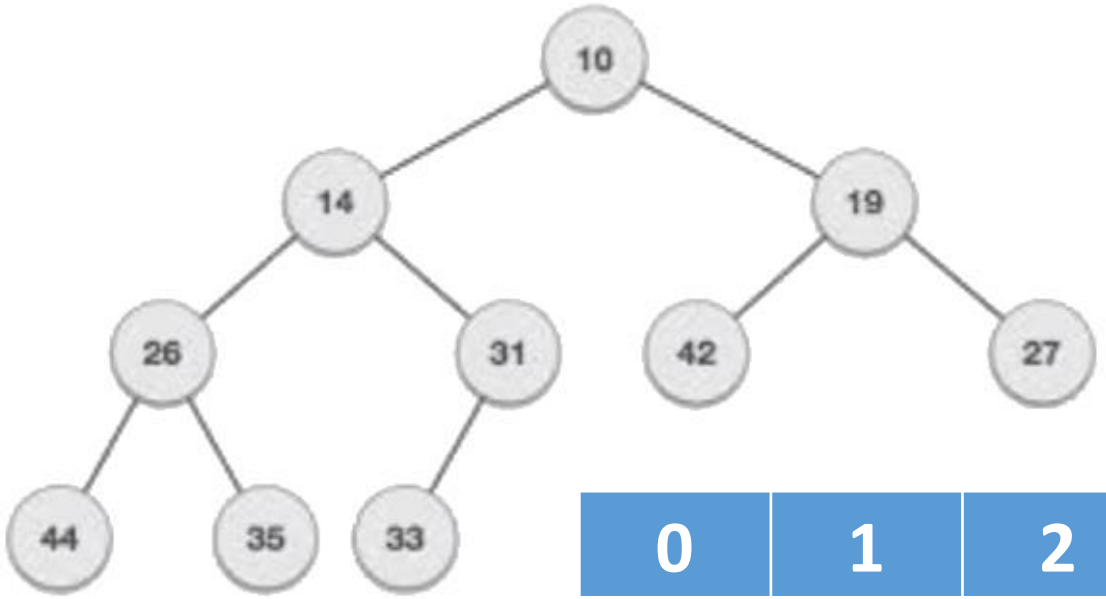
Let's start with the array and see if we correctly recreate the binary heap.

0	1	2	3	4	5	6
15	10	12	5	2	1	6



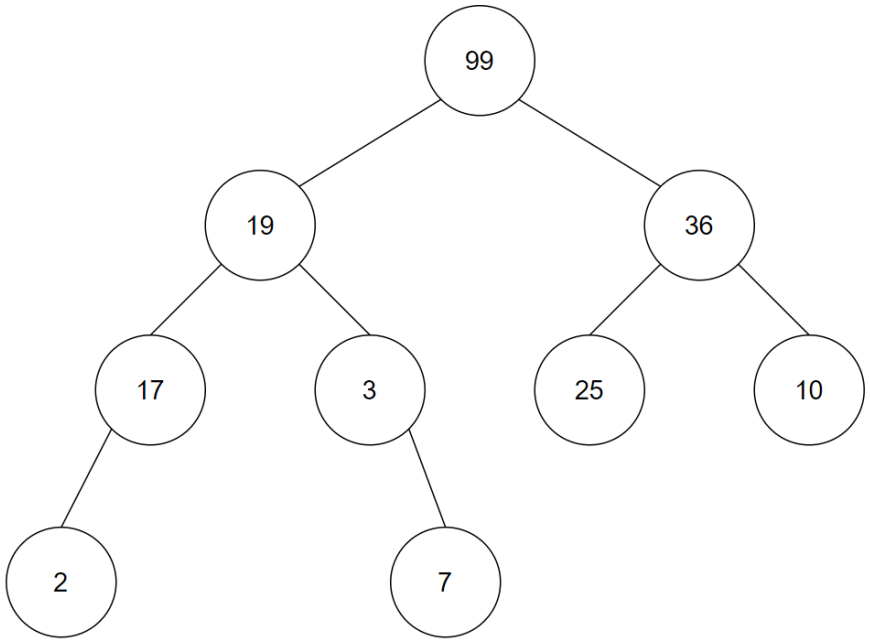
Binary Heap

That was for a max heap – let's try a min heap…



0	1	2	3	4	5	6	7	8	9

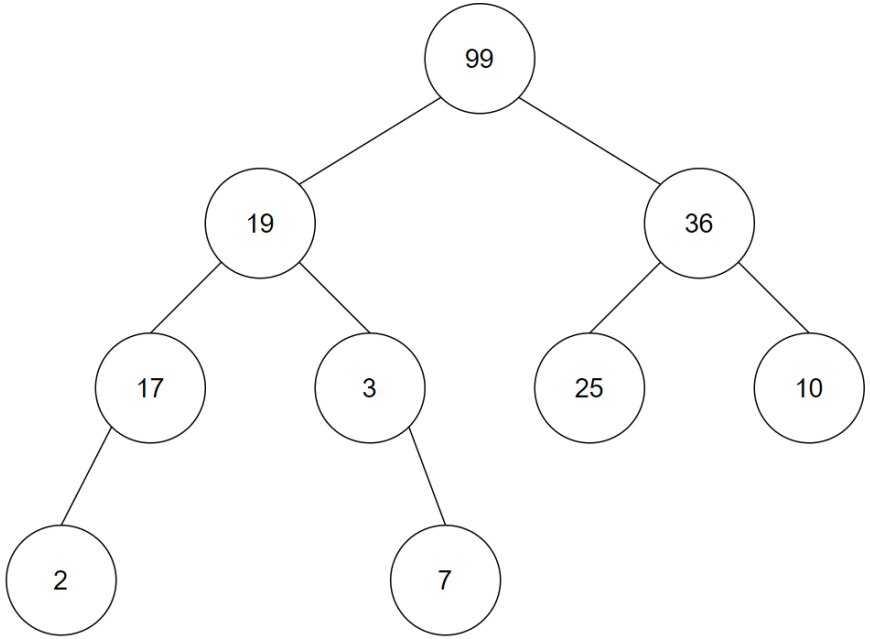
Binary Heap



What happens if we follow this process with a tree that has the values in the right locations (max parents/root) but was not a complete tree?

all the levels are completely filled except possibly the last level and the last level has all keys as left as possible

Binary Heap



What if we just left blanks?

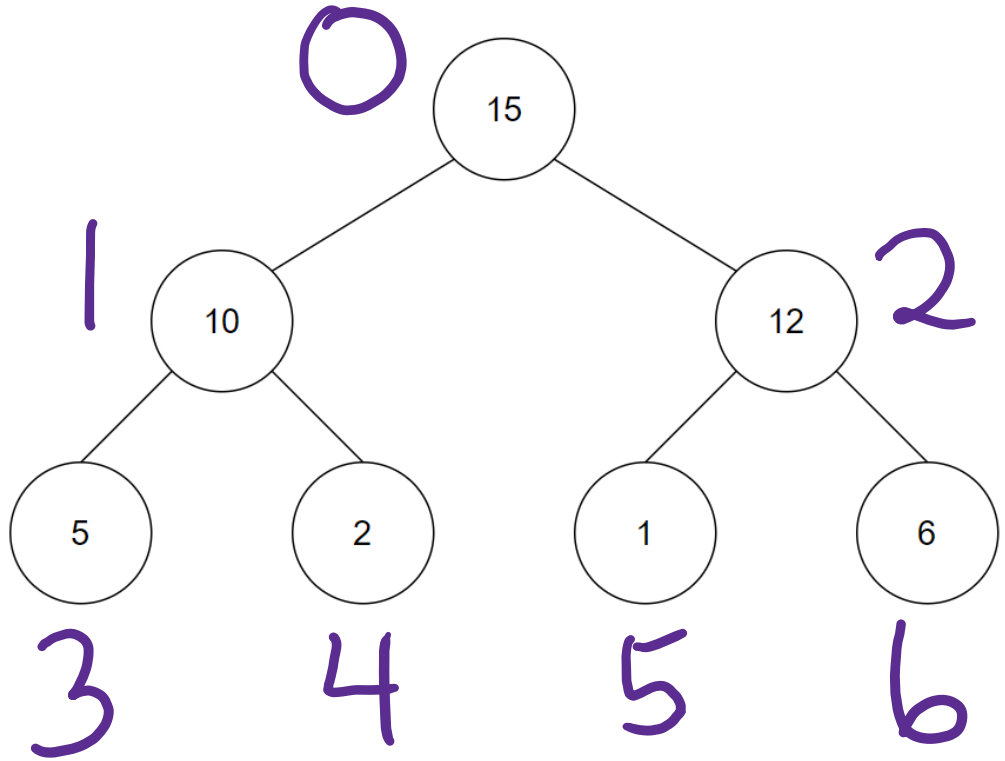
0	1	2	3	4	5	6	7	8	9	10

Binary Heap

This traversal method is called Level Order. We go through each level in order.

0	1	2	3	4	5	6
15	10	12	5	2	1	6

That order corresponds to the array indexes.



Binary Heap

So root always goes to ARRAY[0].

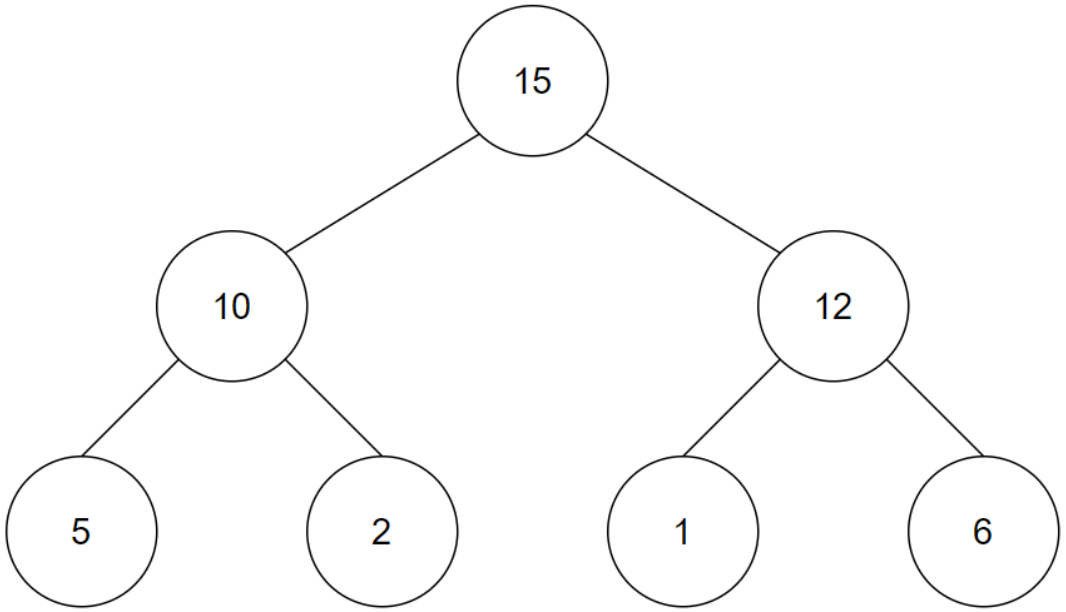
When using zero indexed arrays, we can state that for a index of i

left child = 2i + 1

right child = 2i + 2

Left child of root (i=0) = 2i + 1 = 1

Right child of root (i=0) = 2i + 2 = 2



0	1	2	3	4	5	6
15	10	12	5	2	1	6

Binary Heap

left child = 2i + 1

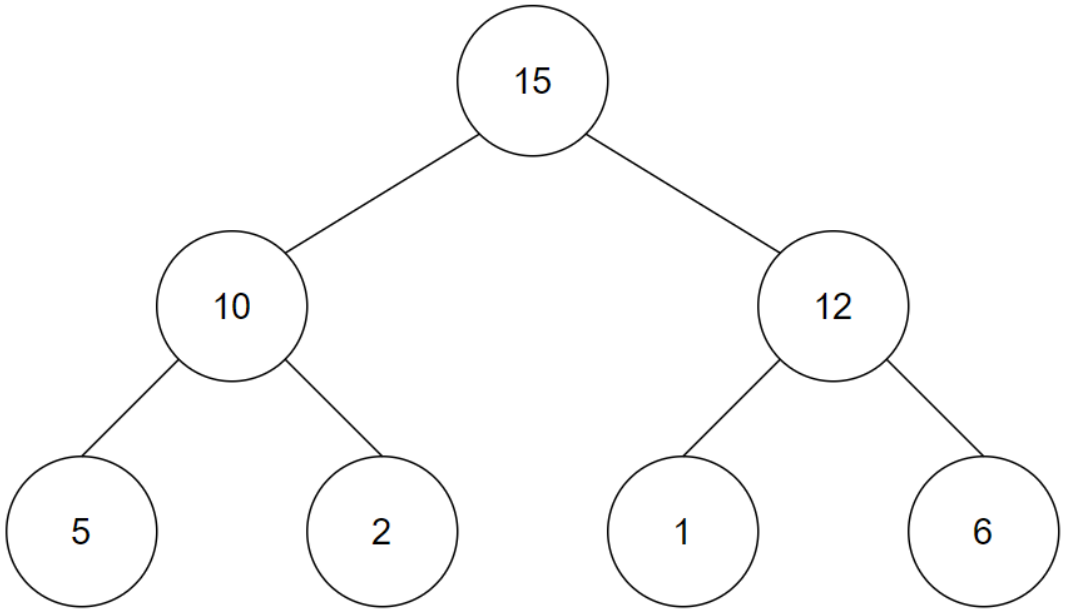
right child = 2i + 2

Left child of (10) (i=1) = 2i + 1 = 3

Right child of (10) (i=1) = 2i + 2 = 4

Left child of (12) (i=2) = 2i + 1 = 5

Right child of (12) (i=2) = 2i + 2 = 6



0	1	2	3	4	5	6
15	10	12	5	2	1	6

Adding to a Max Heap

How do we add a value?

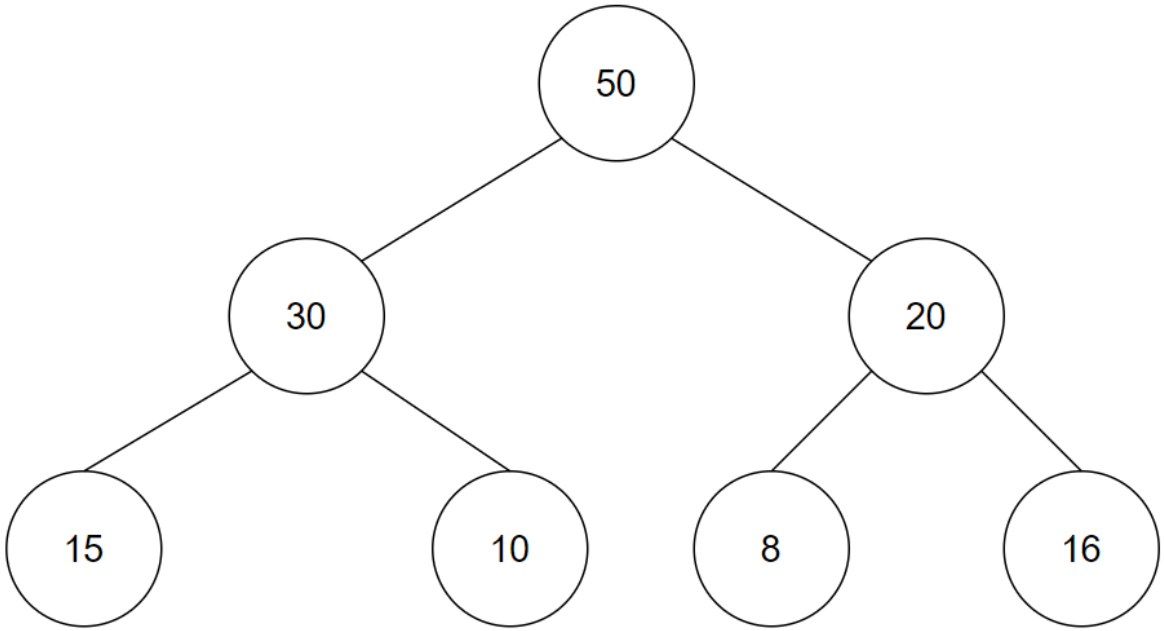
Let's say we want to add 60?

60 is greater than all of the other values in the tree so it should become the root.

But how do we make it the root?

Where does 50 go?

How do we shift around the parents and children?



0	1	2	3	4	5	6
50	30	20	15	10	8	16

Adding to a Max Heap

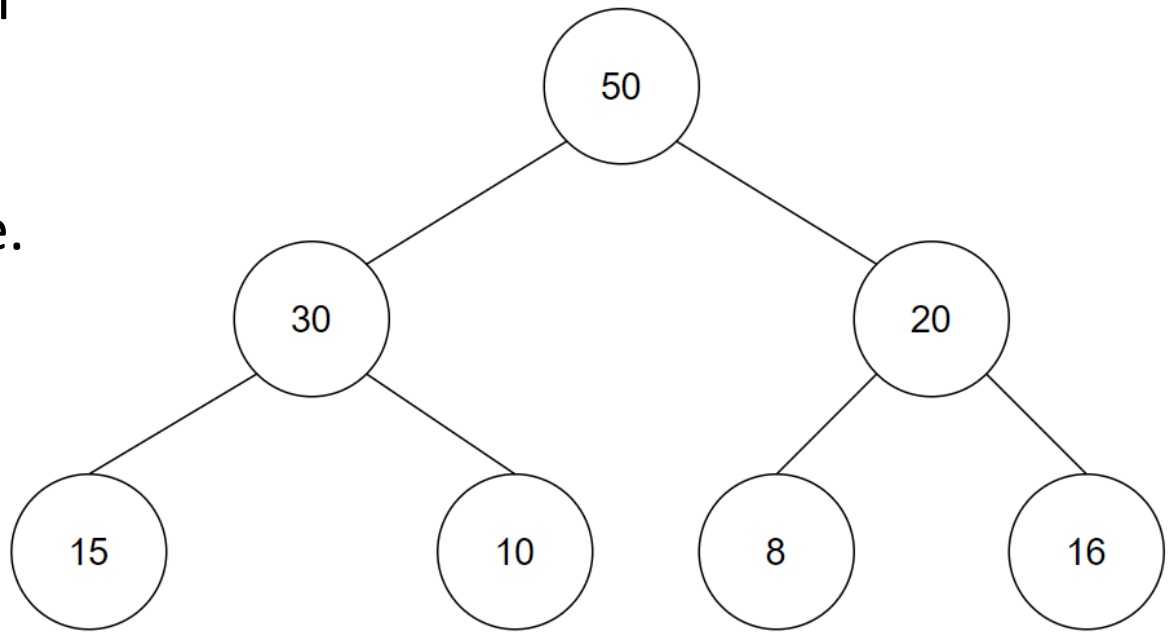
Adding a value to heap is called inserting or pushing.

So let's look at the array instead of the tree.

Where would it make sense to add an element to the array?

At the end would be the easiest

should be room
won't have to move anyone else



0	1	2	3	4	5	6
50	30	20	15	10	8	16

Adding to a Max Heap

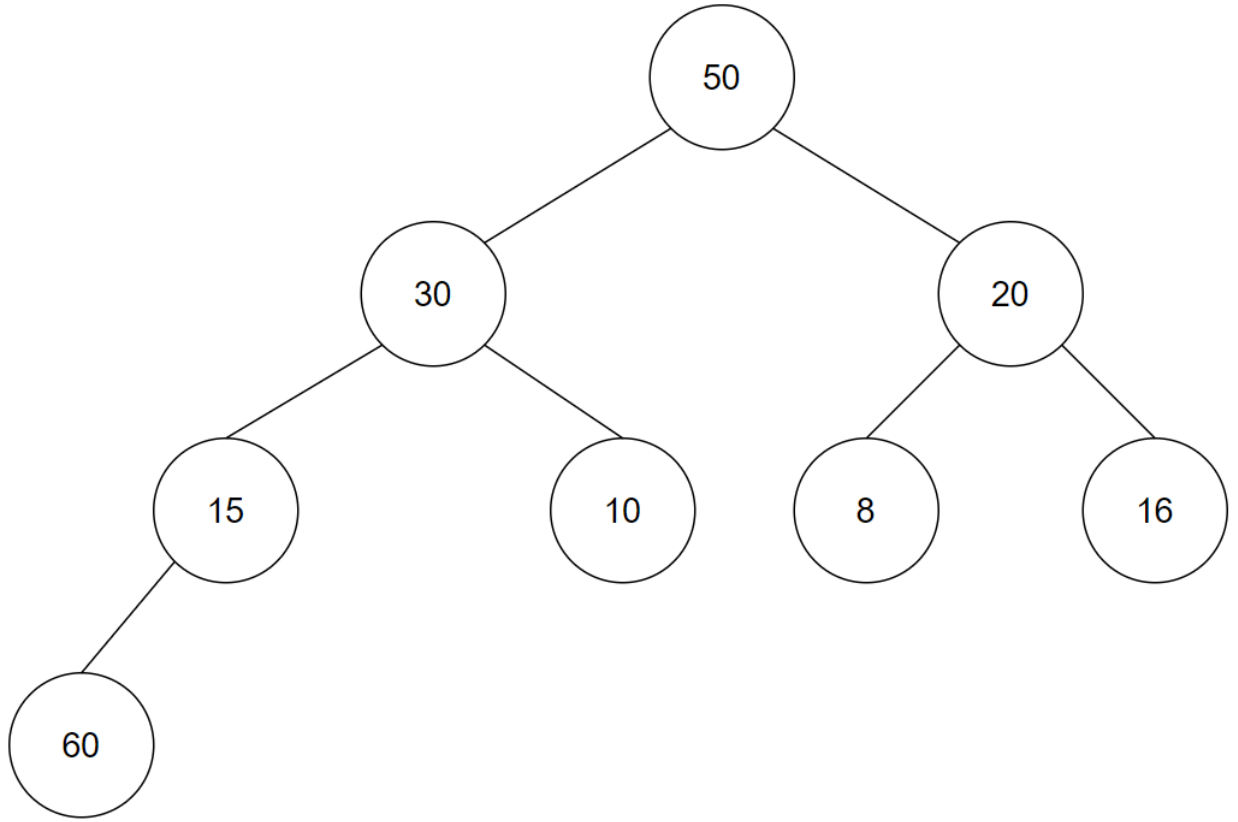
What does that do to our tree?

Where would the 60 go according to the array?

Once we put 60 there, we have violated the heap property

60 is largest value in the tree but it is not the root.

So we need to move it…



0	1	2	3	4	5	6	7
50	30	20	15	10	8	16	60

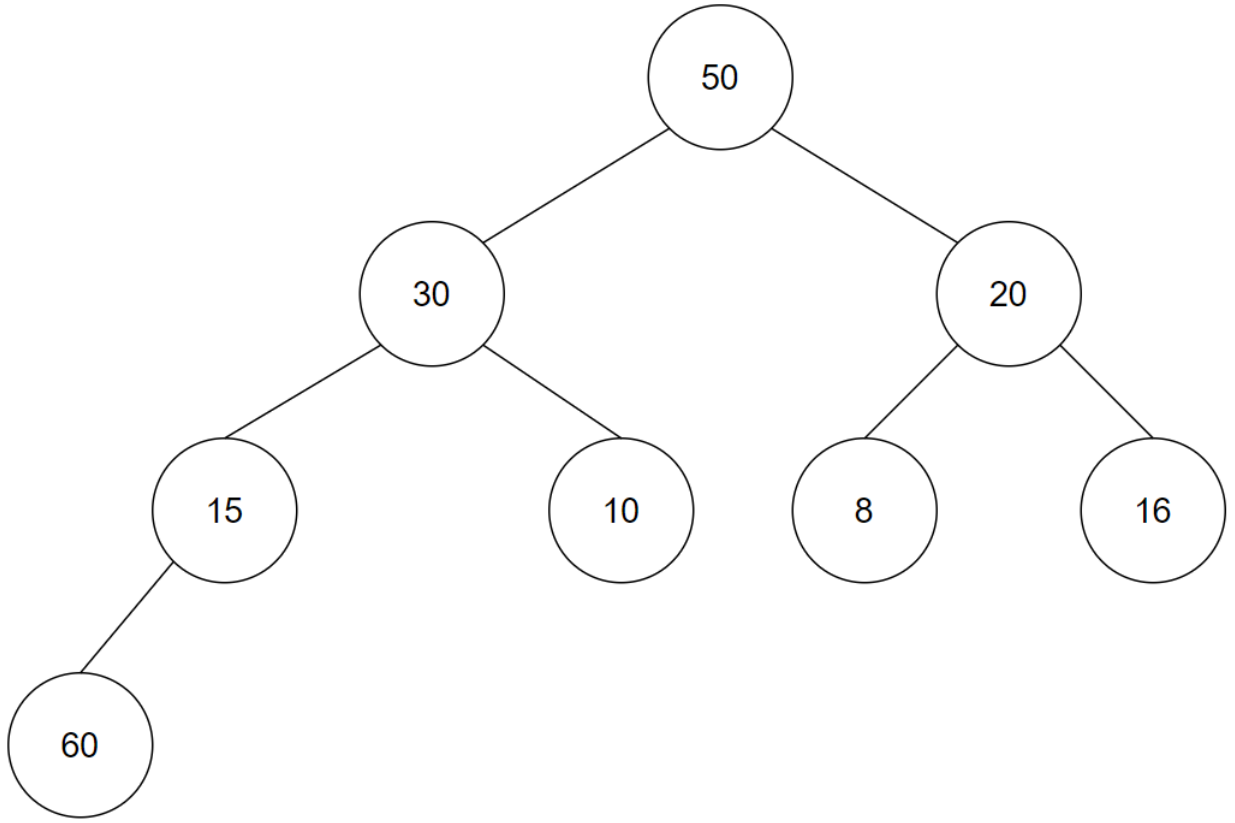
Adding to a Max Heap

We could swap 15 and 60.

And then swap 30 and 60.

And then swap 50 and 60.

Have we restored the heap property?

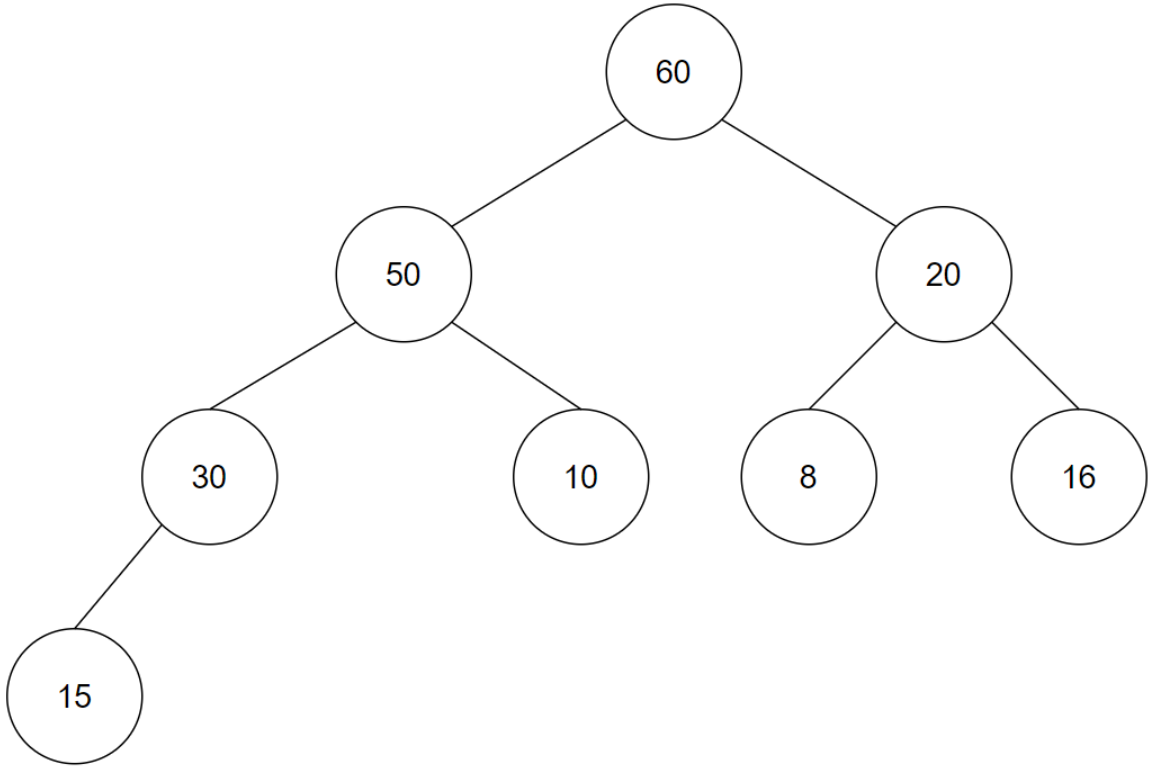


0	1	2	3	4	5	6	7
50	30	20	15	10	8	16	60

Adding to a Max Heap

Yes!

We have a complete tree and the root is the greatest value and every child is less than its parent.



0	1	2	3	4	5	6	7
60	50	20	30	10	8	16	15

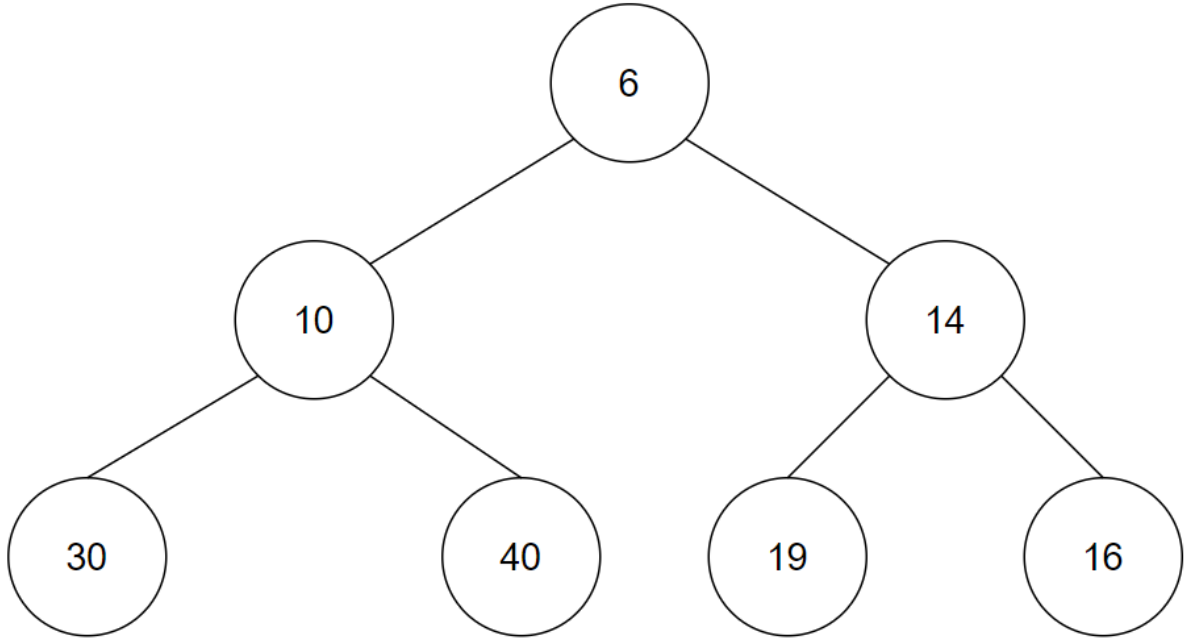
Adding to a Min Heap

How do we add a value?

Let's say we want to add 2?

2 is smaller than all of the other values in the tree so it should become the root.

Do we follow the same process as we did with max heap?



0	1	2	3	4	5	6
6	10	14	30	40	19	16

Adding to a Min Heap

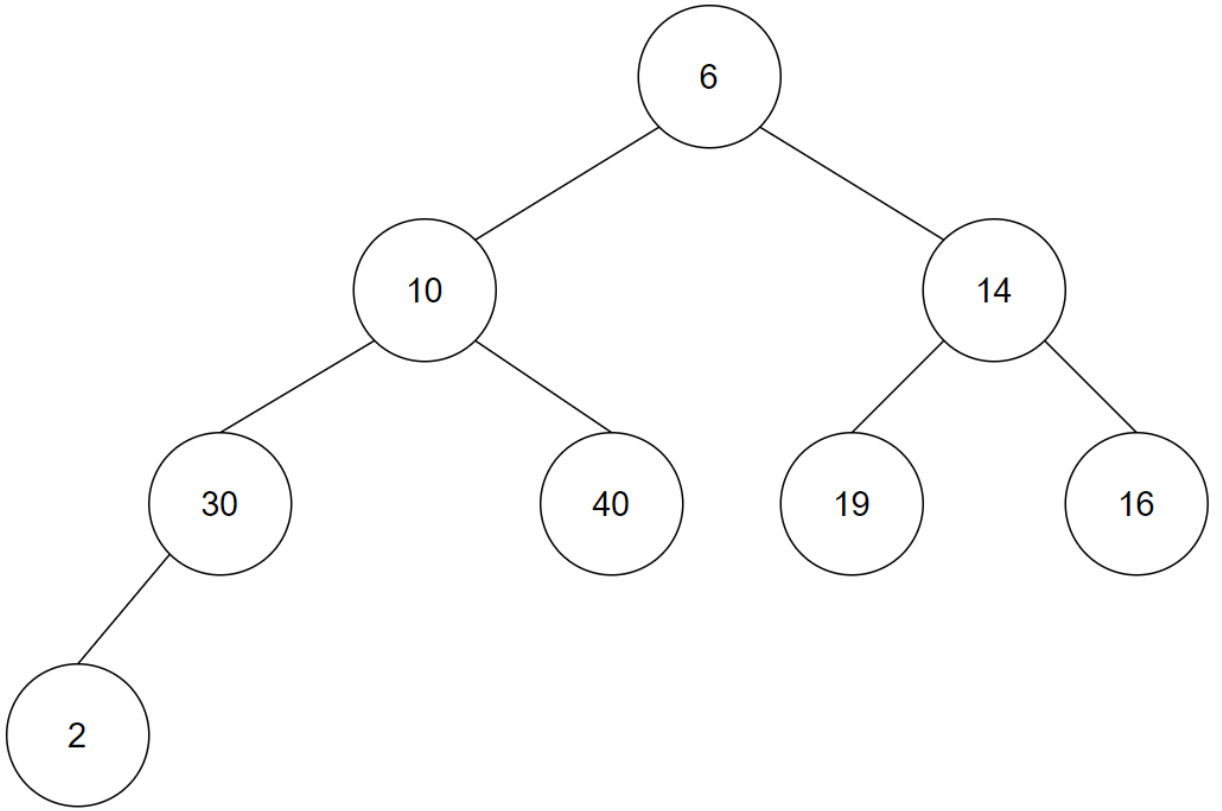
Let's add the 2 just like we did 60.

Now we need to swap 2 and 30.

Now swap 2 and 10.

Now swap 6 and 2.

Have we restored the heap property?

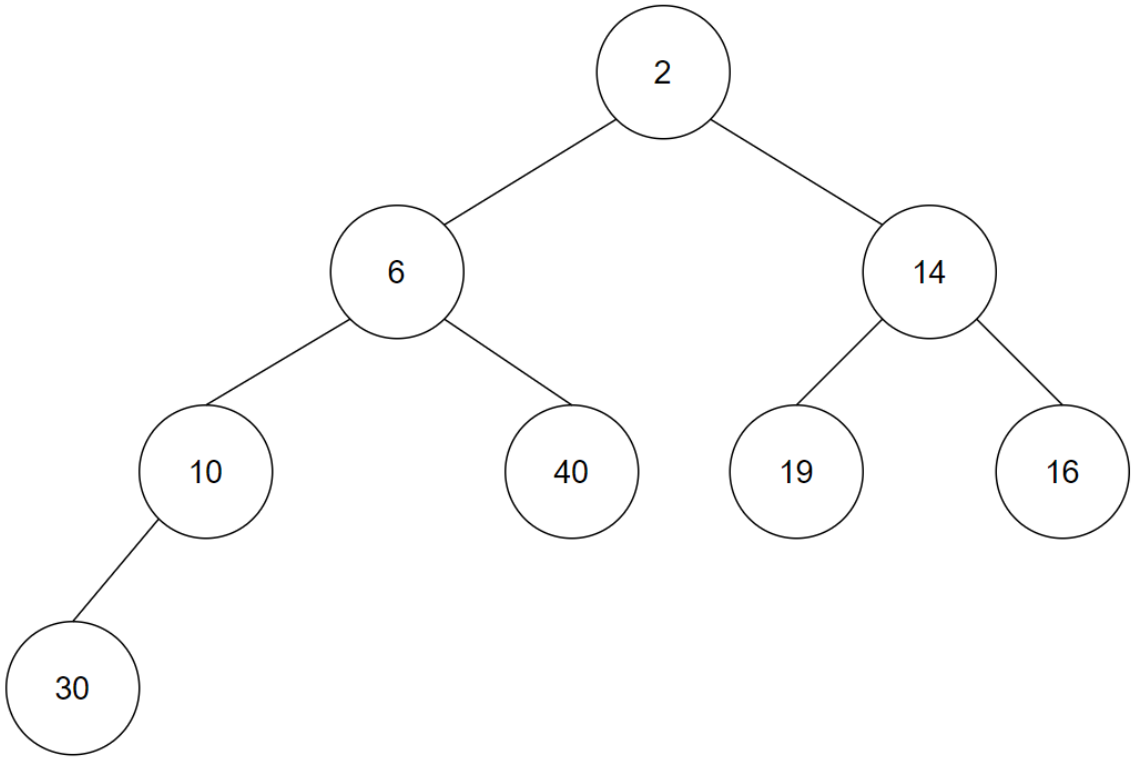


0	1	2	3	4	5	6	7
6	10	14	30	40	19	16	2

Adding to a Min Heap

Yes!

We have a complete tree and the root is the smallest value and every child is greater than its parent.



0	1	2	3	4	5	6	7
2	6	14	10	40	19	16	30

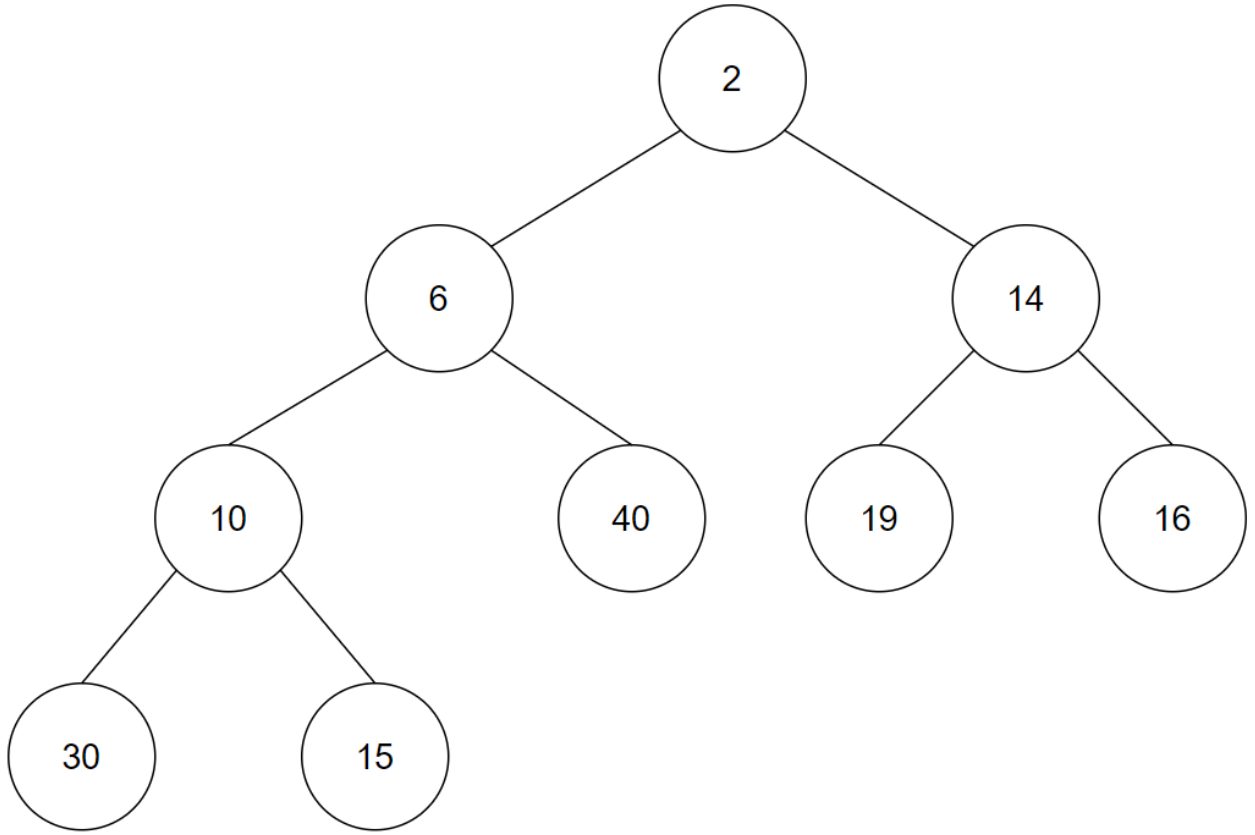
Adding/Inserting with Heap

What if my heap looks like this and I want to add 1.

Where do I add it?

Follow the same process as before…

Add the new value at the end of the array.



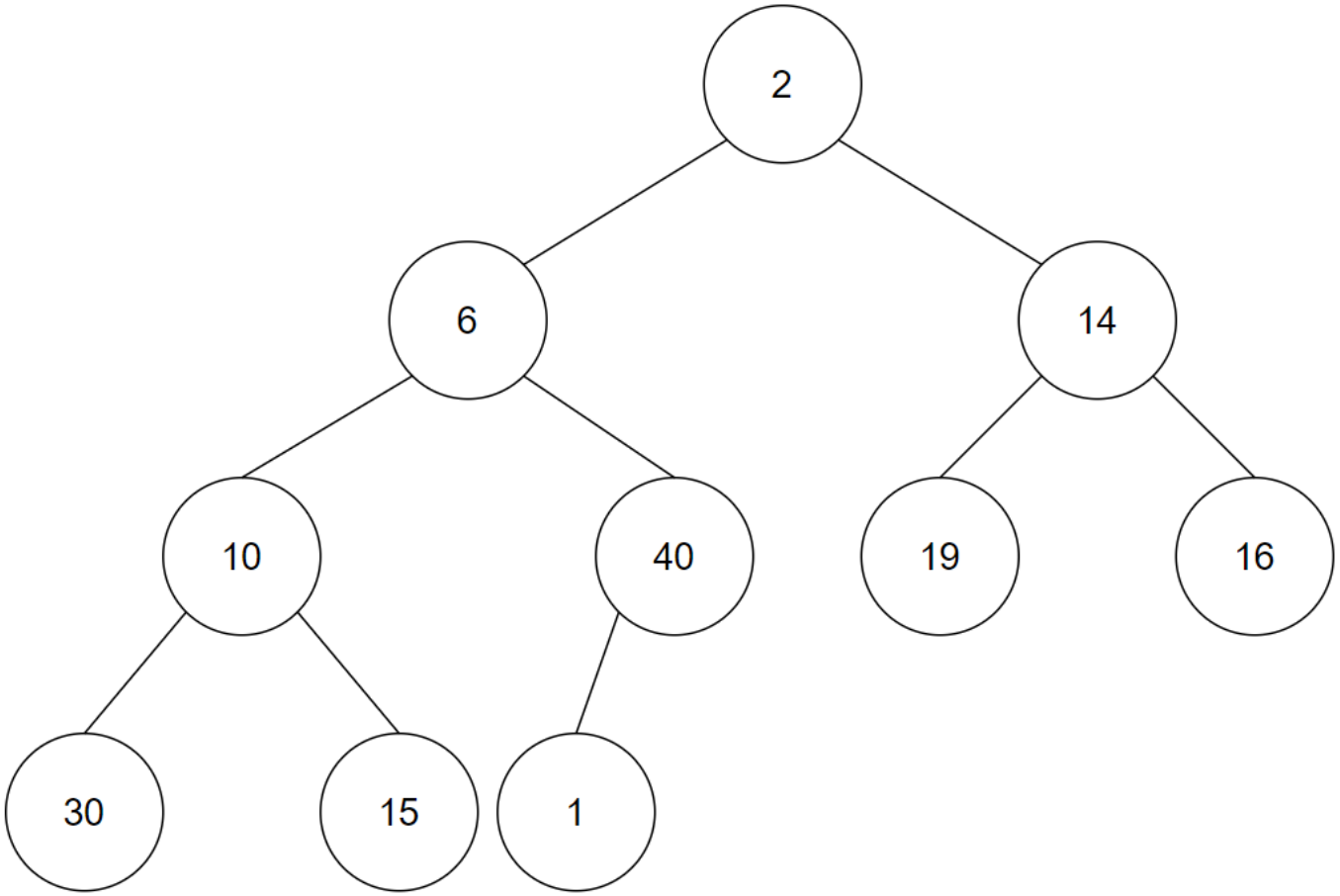
0	1	2	3	4	5	6	7	8	9
2	6	14	10	40	19	16	30	15	

Adding/Inserting with Heap

Now swap 40 and 1

Swap 1 and 6

Swap 1 and 2



0	1	2	3	4	5	6	7	8	9
2	6	14	10	40	19	16	30	15	1

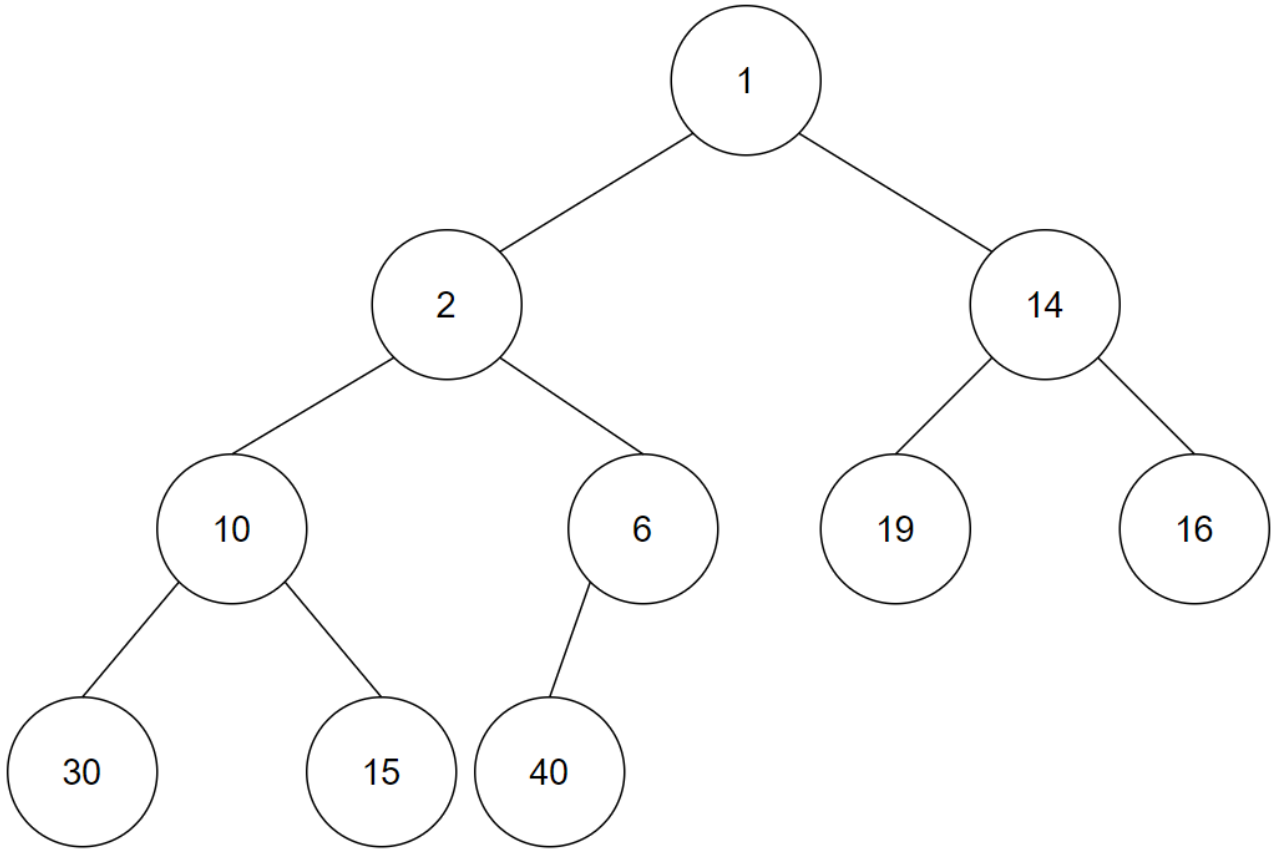
Adding/Inserting with Heap

Do we still have a min heap?

Complete tree and every parent is less than its children.

Root is the smallest.

Yes!



0	1	2	3	4	5	6	7	8	9
1	2	14	10	6	19	16	30	15	40

Deleting from a Heap

If you had a stack of cans like this one, how would you remove a can?

Without making a mess or knocking them over?

This is not a game of Jenga

so we would take the top can.

Deleting from a Heap

We delete from a heap the same way – we take the top element.

For a max heap, the top element/root is the biggest element in the heap.

For a min heap, the top element/root is the smallest element in the heap.

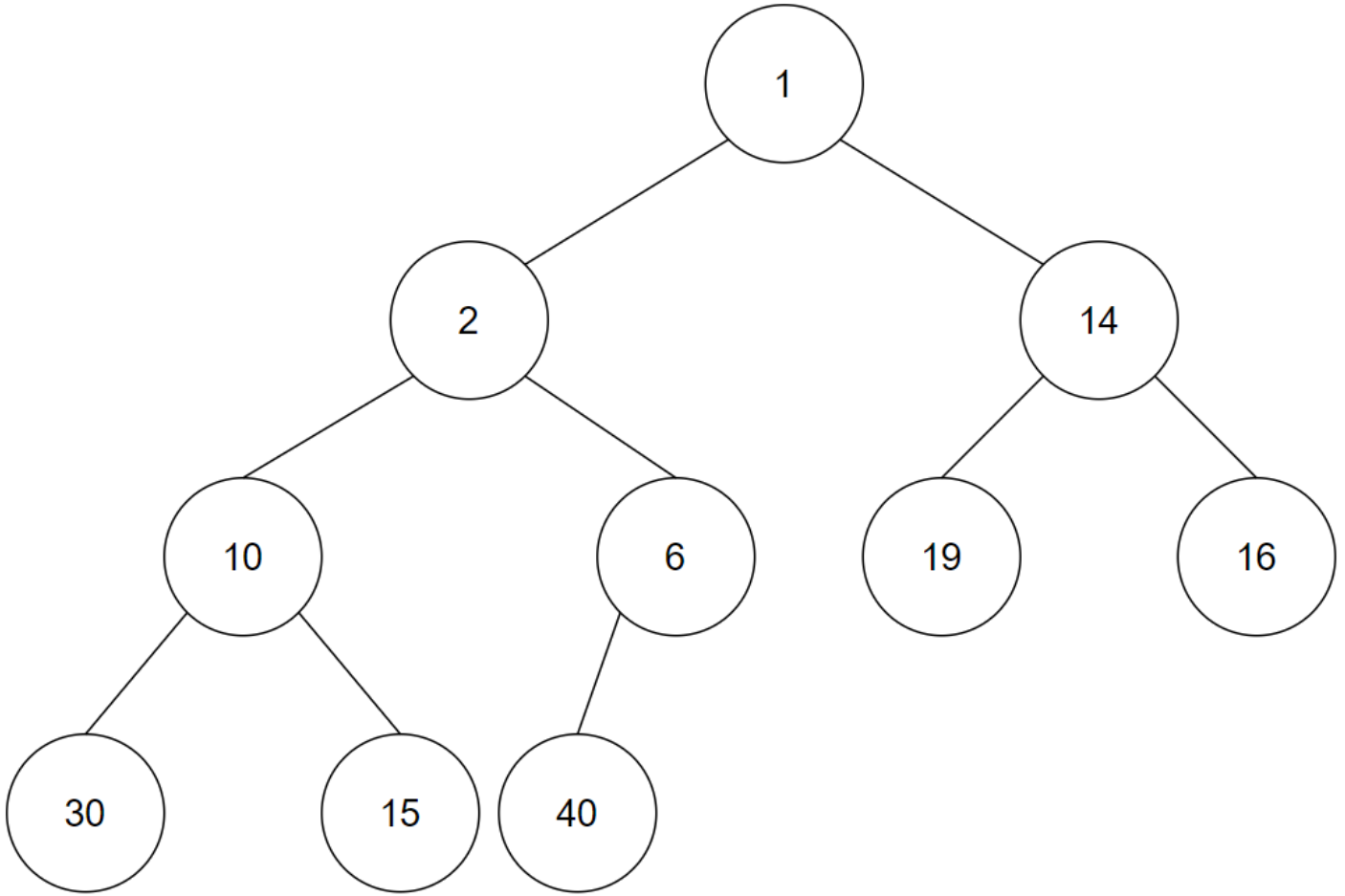
Deleting from a Min Heap

So what happens when we remove the root?

We need a new root?

Do we pick one of the root's children?

No.



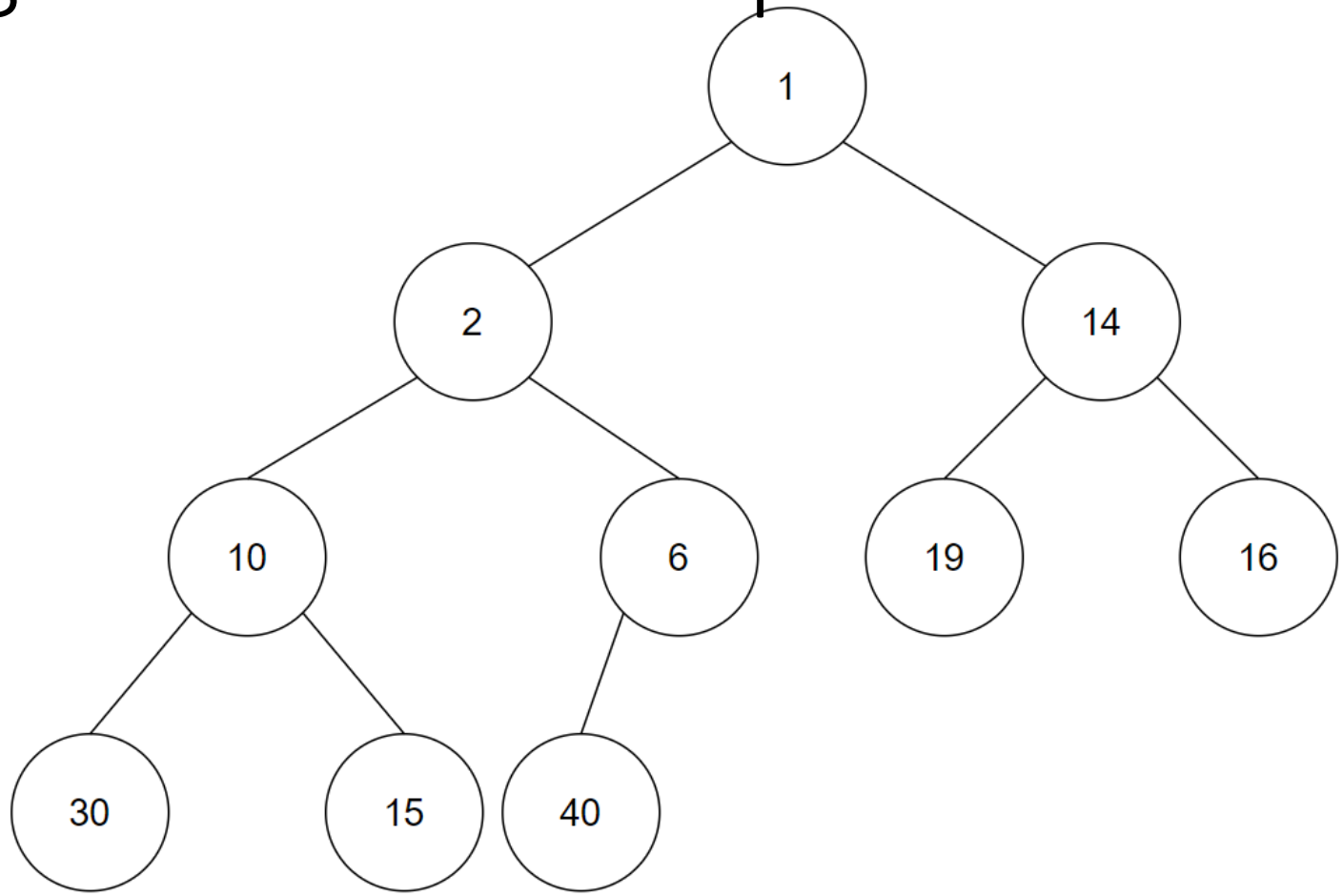
Deleting from a Min Heap

Let's look at the array again.

If we take the value out of the 0th element, what is the easiest replacement value from the array?

If we take element 1, then we'll just have another gap.

We've already said that we don't like gaps in our arrays.



0	1	2	3	4	5	6	7	8	9
	2	14	10	6	19	16	30	15	40

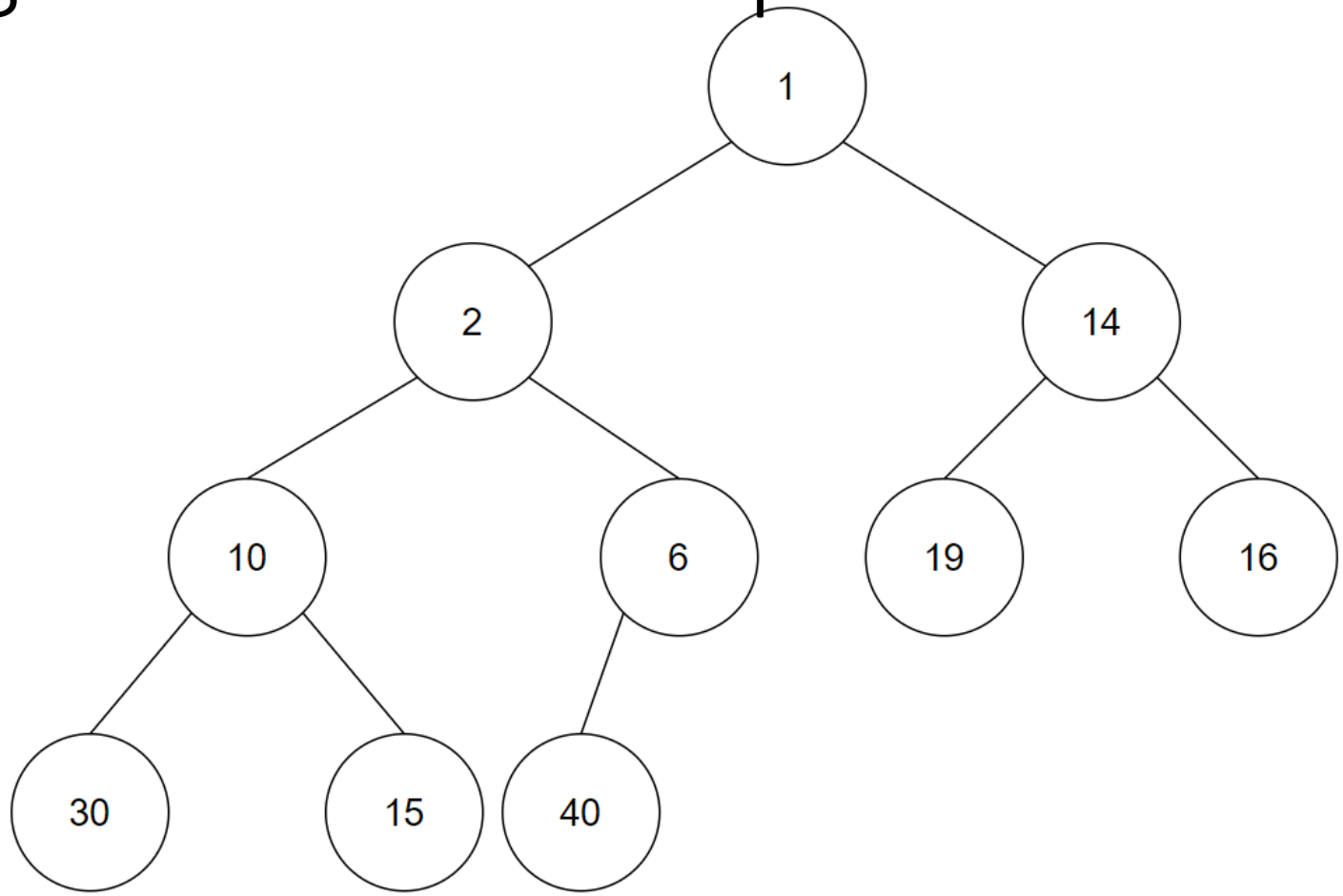
Deleting from a Min Heap

What if we move the last element to the 0^{th} spot?

It's OK to have spaces at the END of the array.

What will the heap look like?

Do we have a problem?



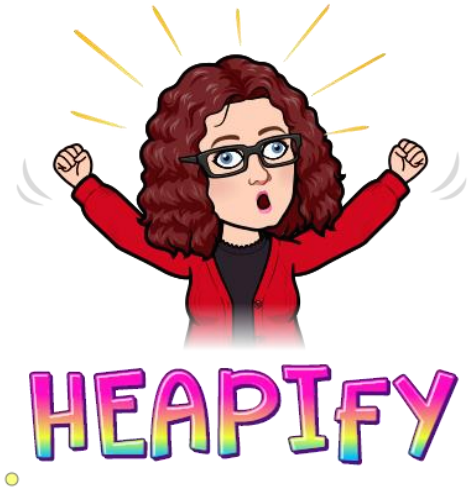
0	1	2	3	4	5	6	7	8	9
40	2	14	10	6	19	16	30	15	

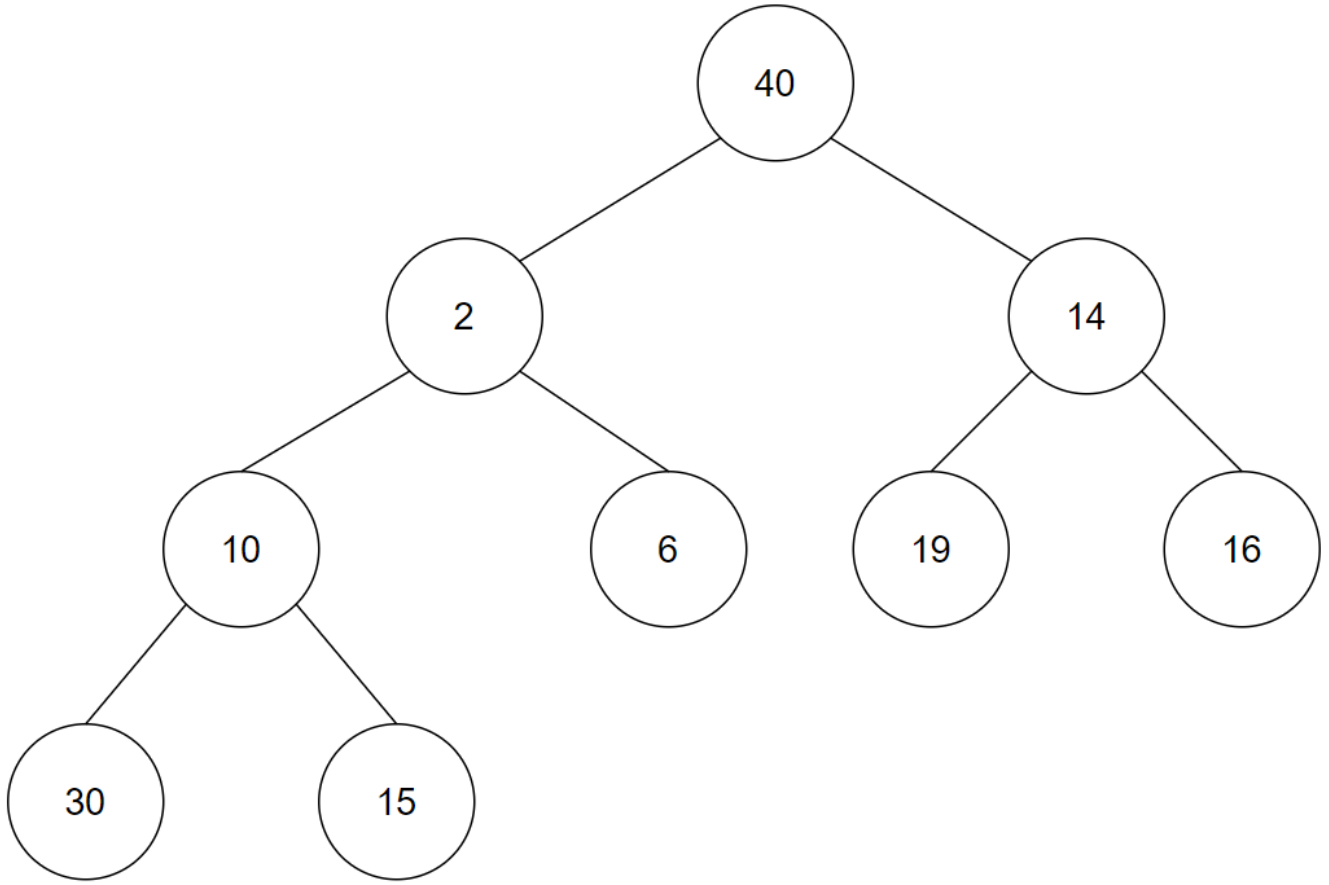
Deleting from a Min Heap

Do we have a problem?

We have lost our heap property again.

The tree is still complete but 40 is too big to be the root of this min heap.





0	1	2	3	4	5	6	7	8	9
40	2	14	10	6	19	16	30	15	

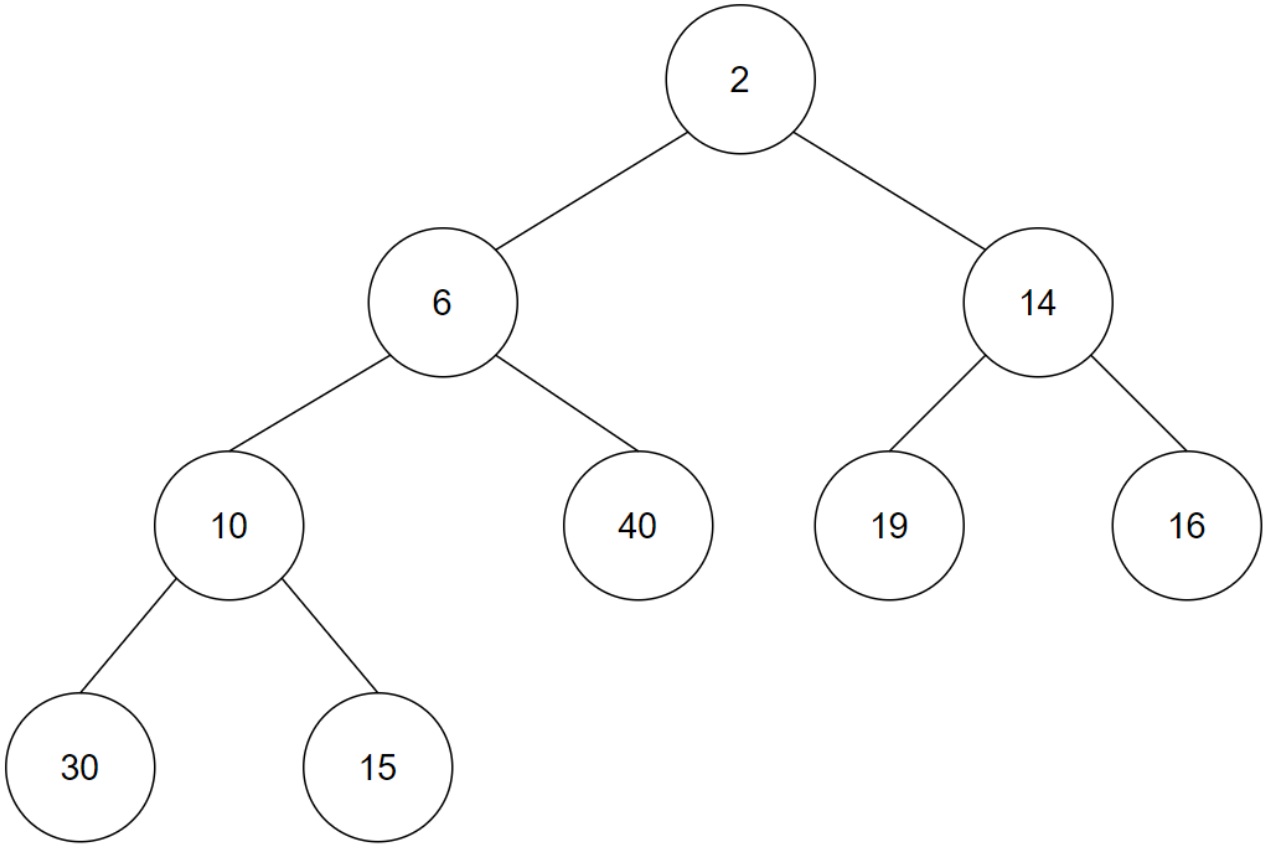
Deleting from a Min Heap

We are back to a min heap

Complete tree

Parents smaller than children

Root is smallest value.



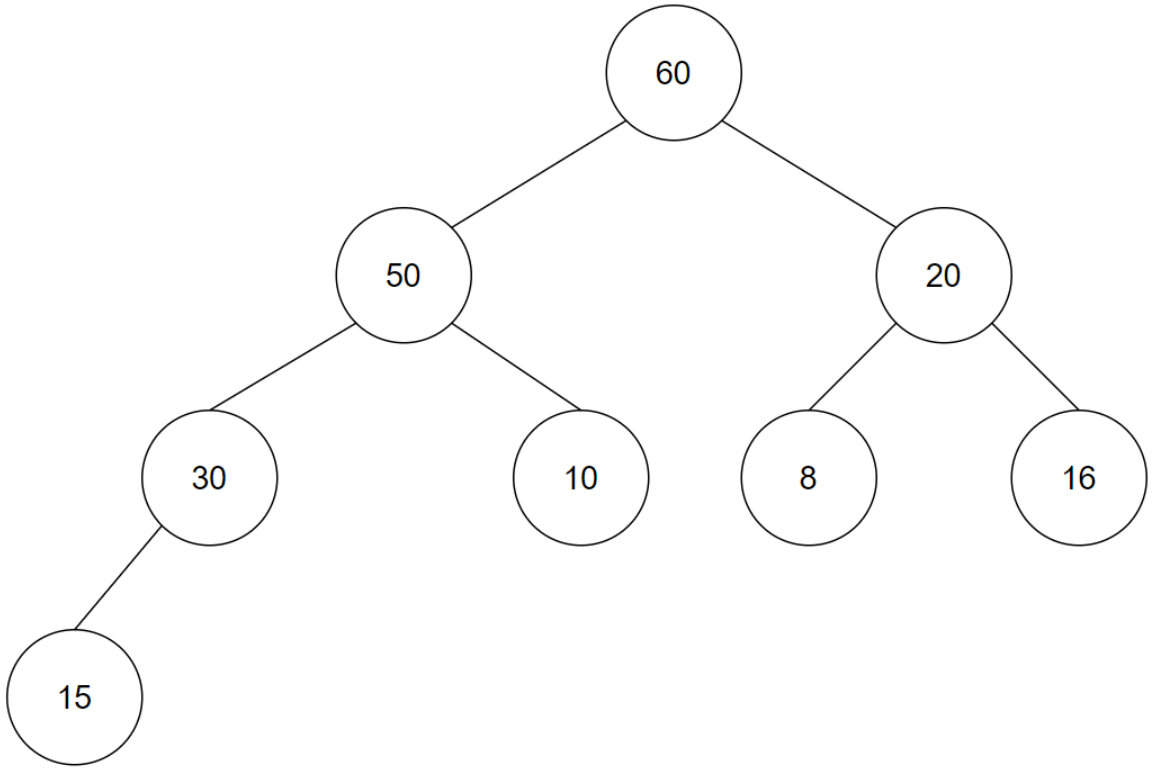
0	1	2	3	4	5	6	7	8	9
2	6	14	10	40	19	16	30	15	

Deleting from a Max Heap

Let's try the same technique with a max heap.

We want remove 60.

We take 60 out and swap it with the last value of 15.



0	1	2	3	4	5	6	7
60	50	20	30	10	8	16	15

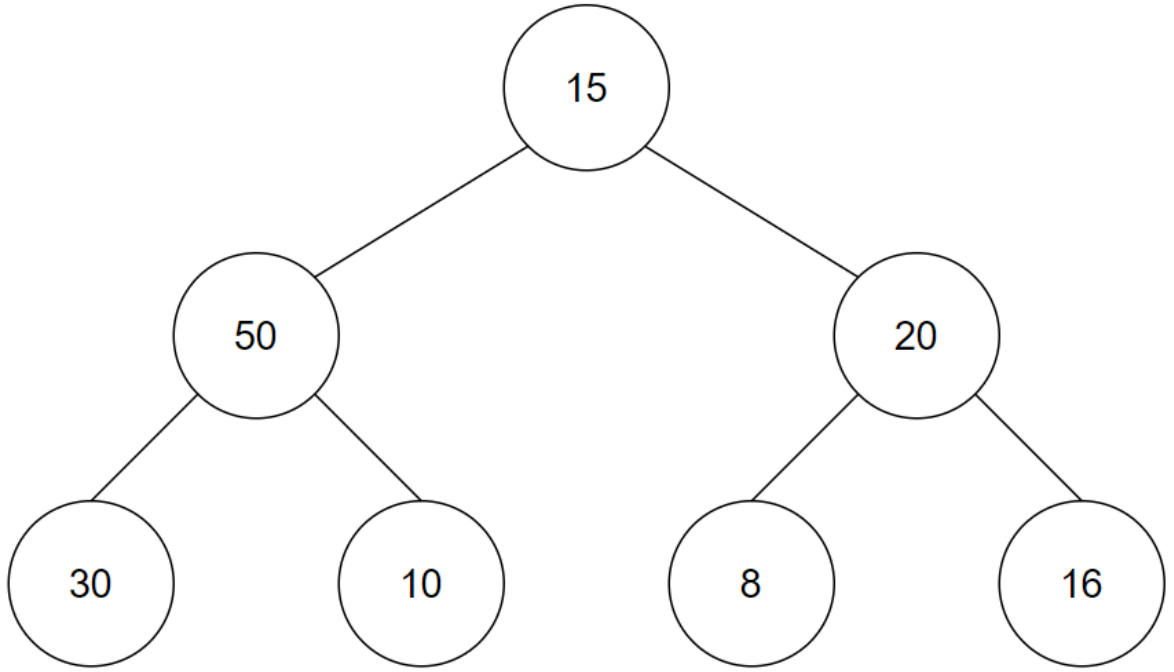
Deleting from a Max Heap

Now let's fix our heap.

Swap 15 and 50

Swap 15 and 30

Are we OK now?

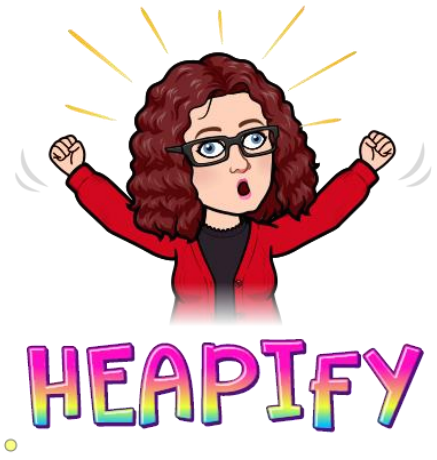


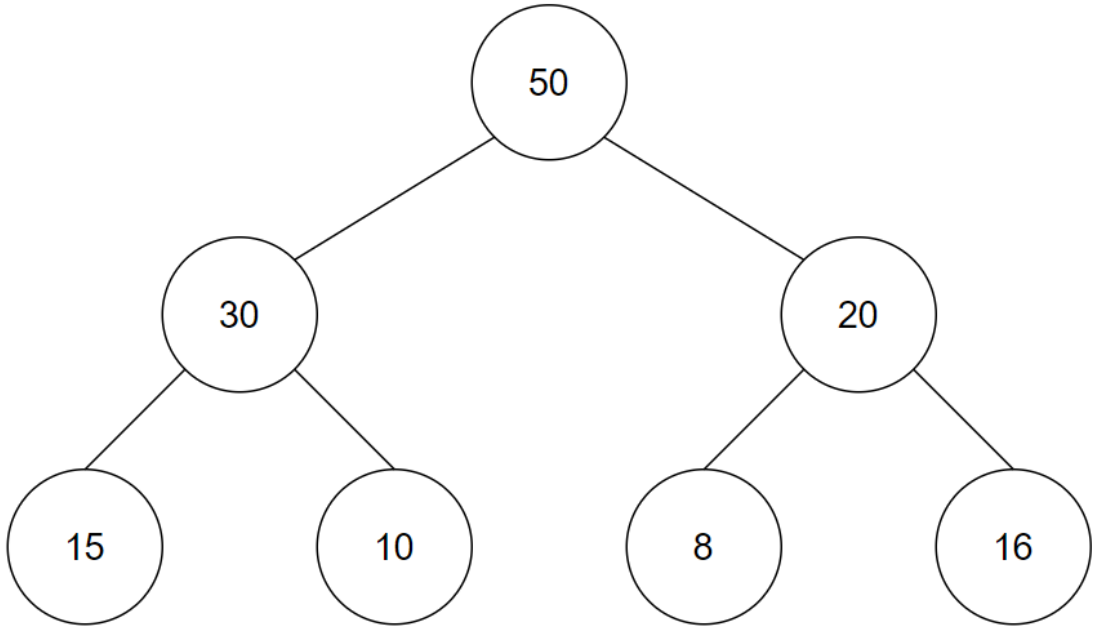
0	1	2	3	4	5	6	7
15	50	20	30	10	8	16	

Deleting from a Max Heap

Yes!

Our tree is complete and we have maintained our heap property.

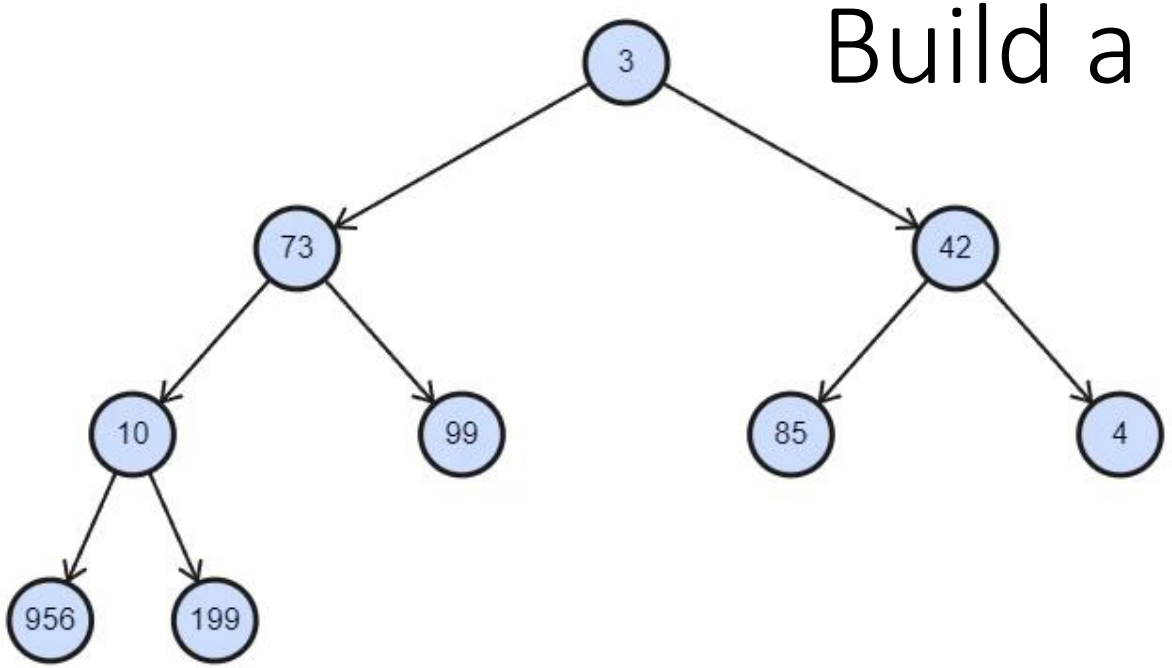


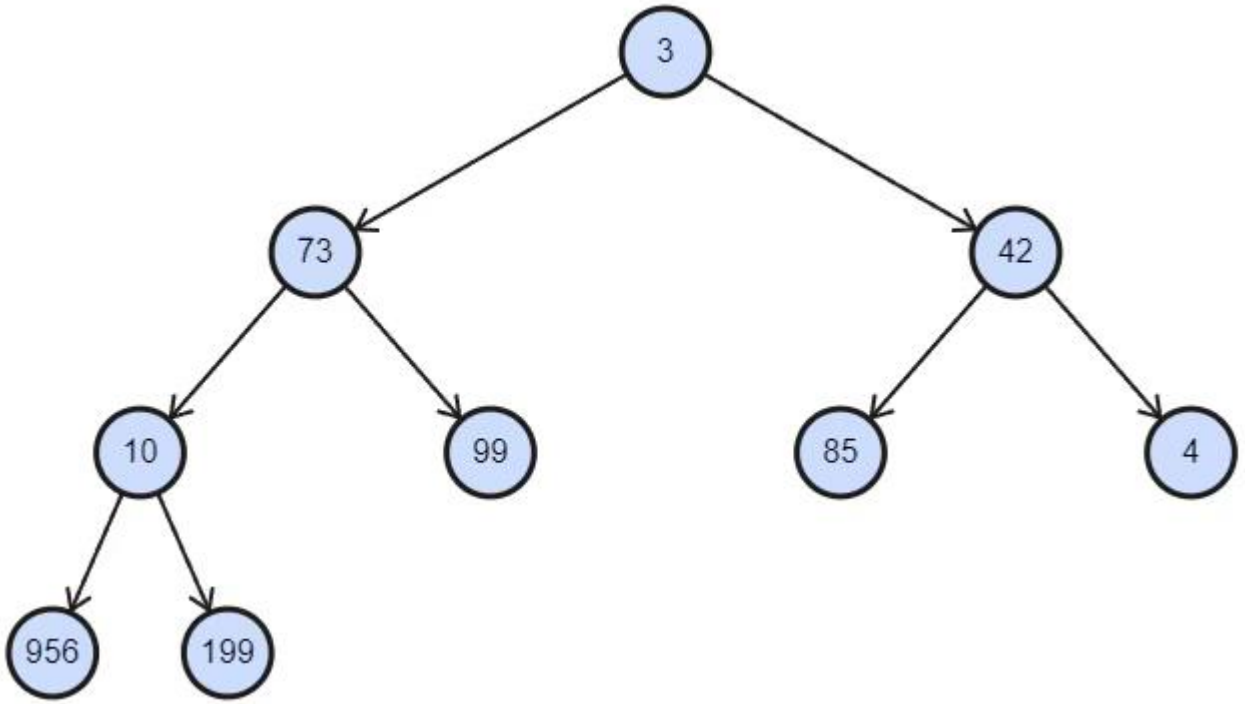


0	1	2	3	4	5	6	7
50	30	20	15	10	8	16	

3	73	42	10	99	85	4	956	199

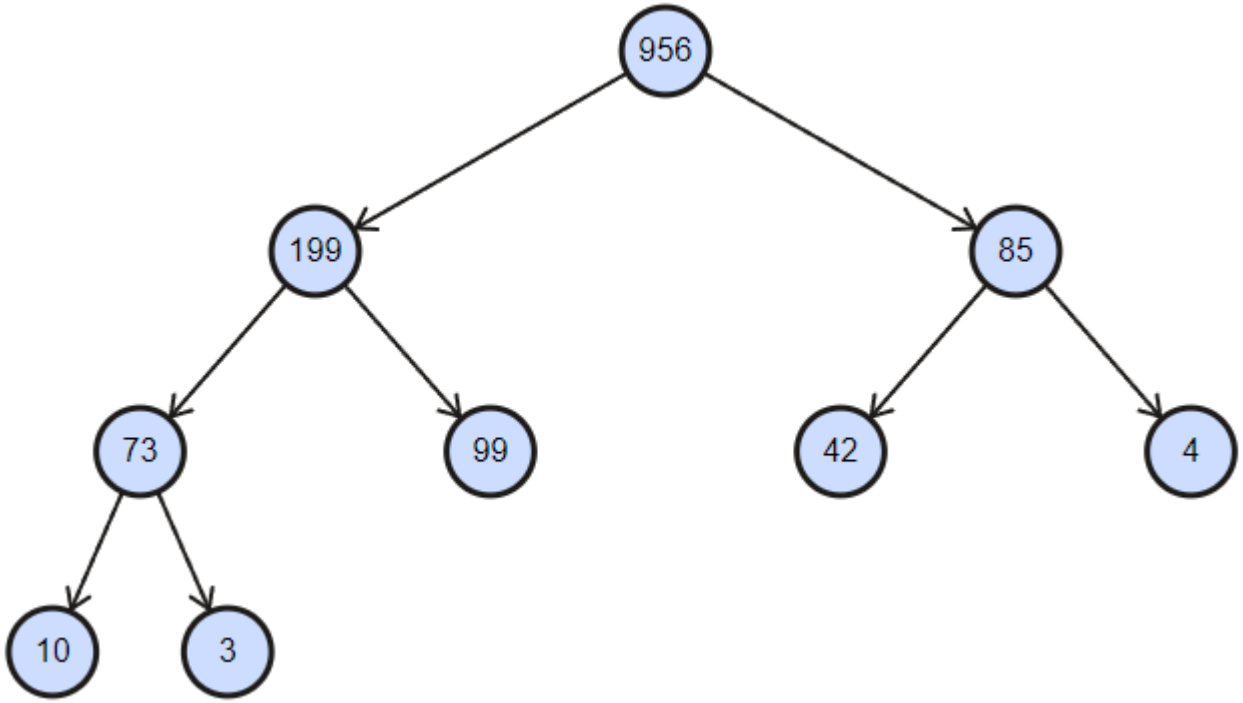
Build a Max Heap



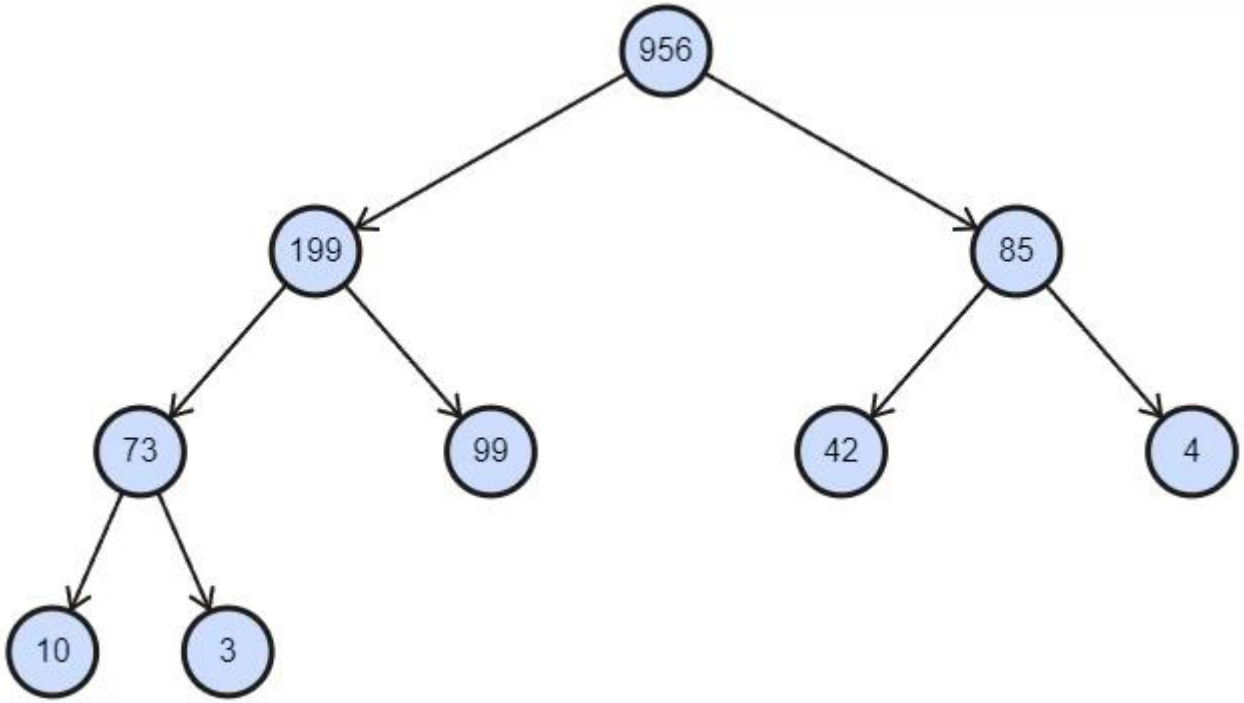


956	199	85	73	99	42	4	10	3

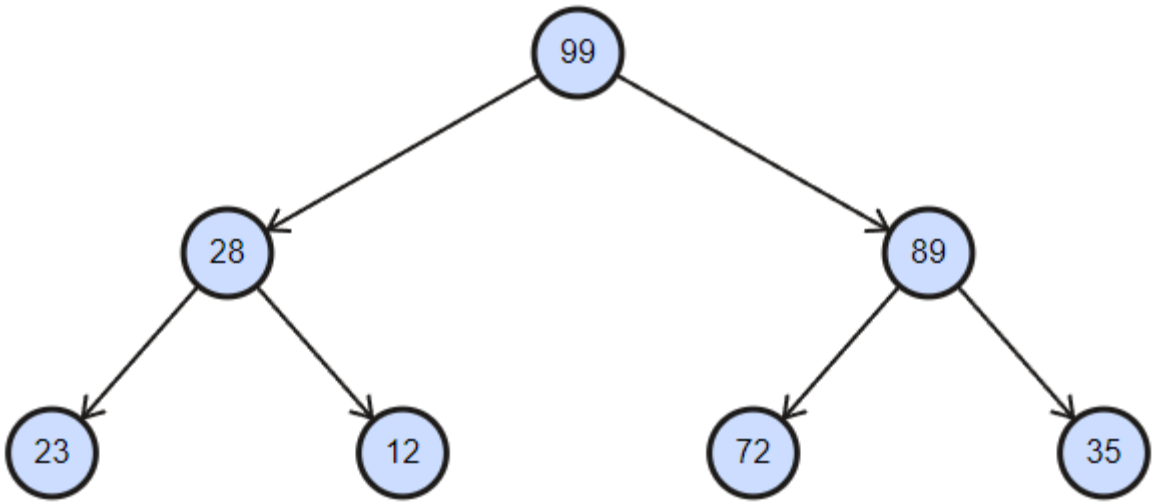
Insert a New Value in a Max Heap



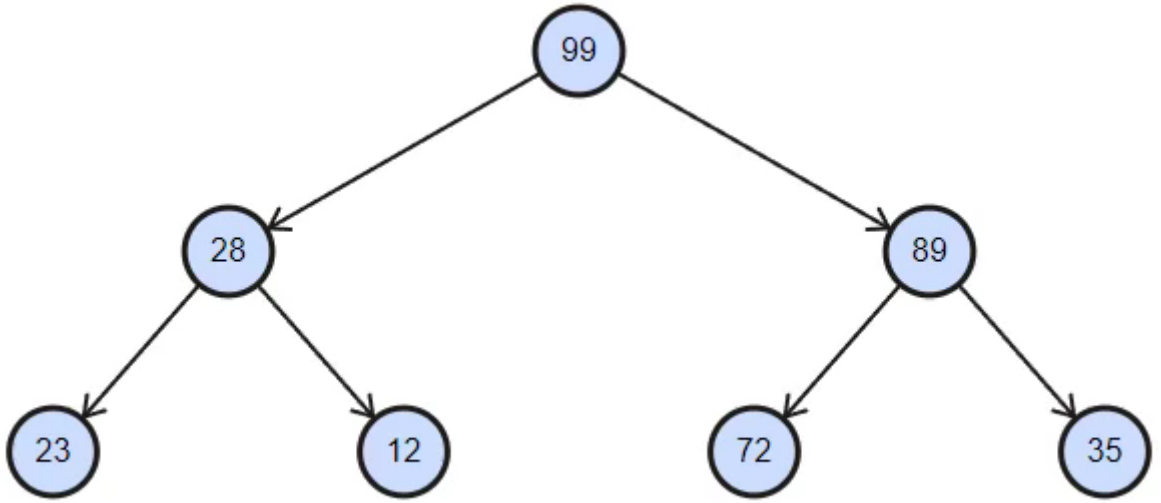
Insert 200



Delete from Max Heap

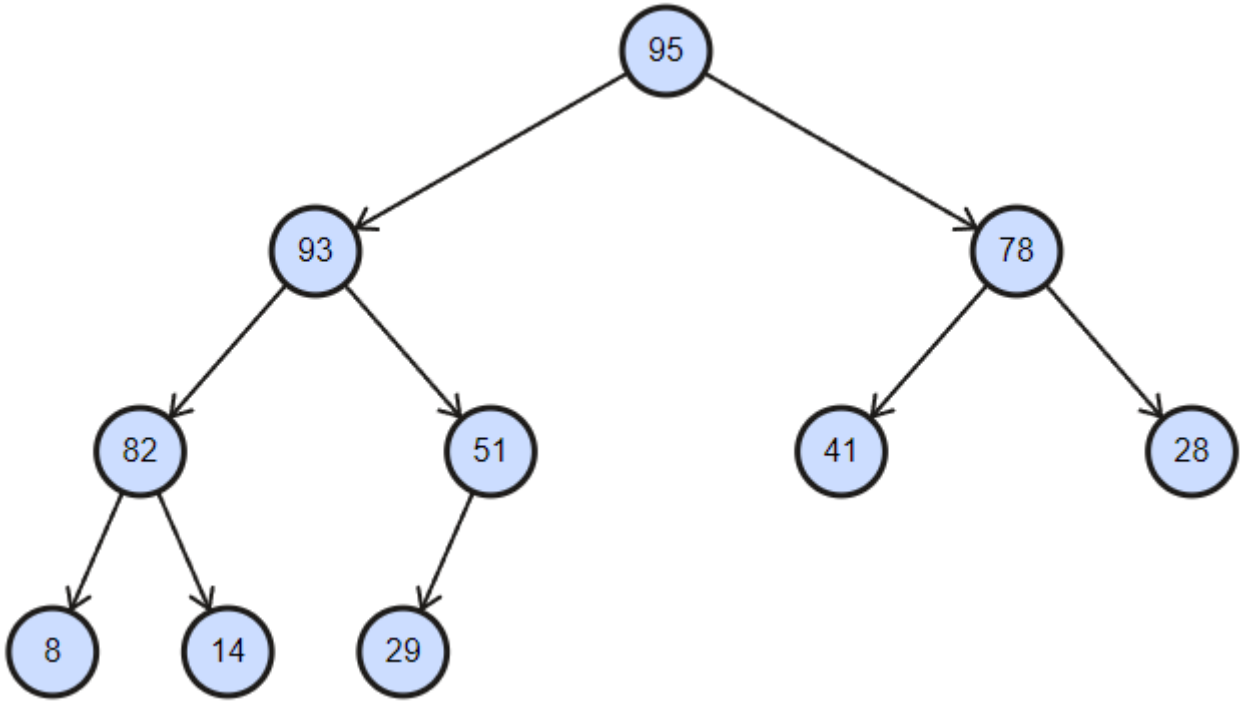


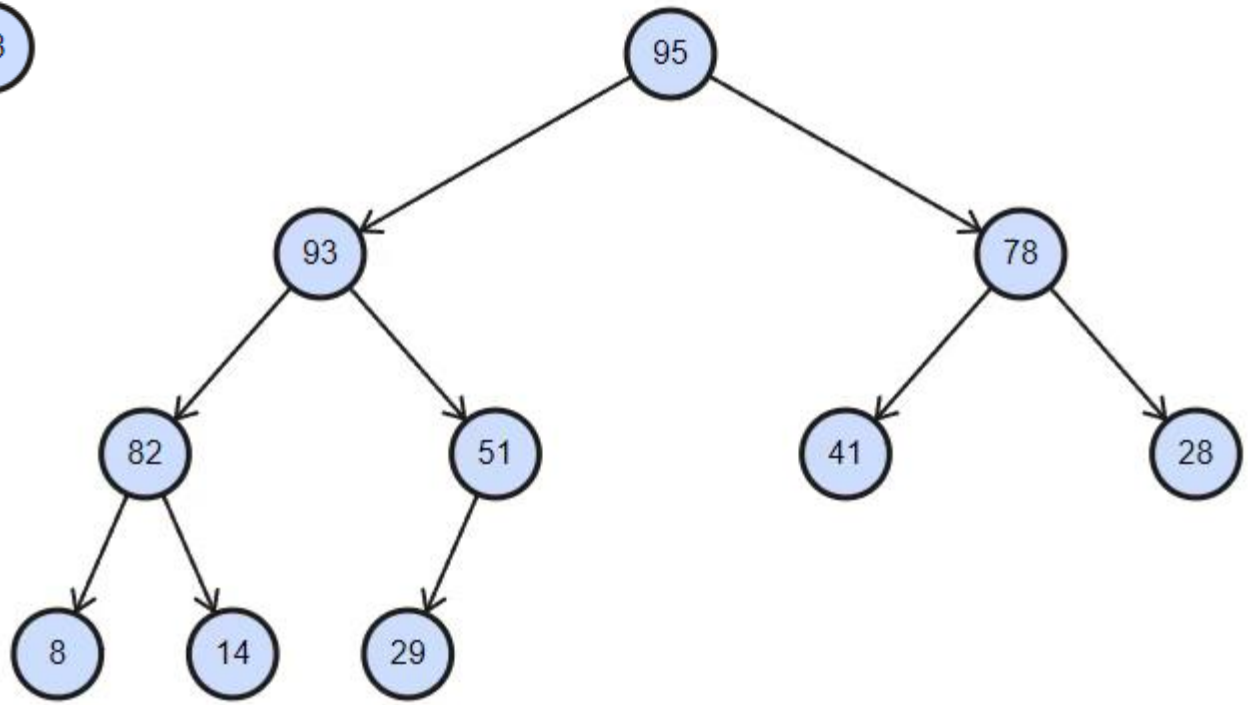
Delete root



95	93	78	82	51	41	28	8	14	29

Delete from Max Heap



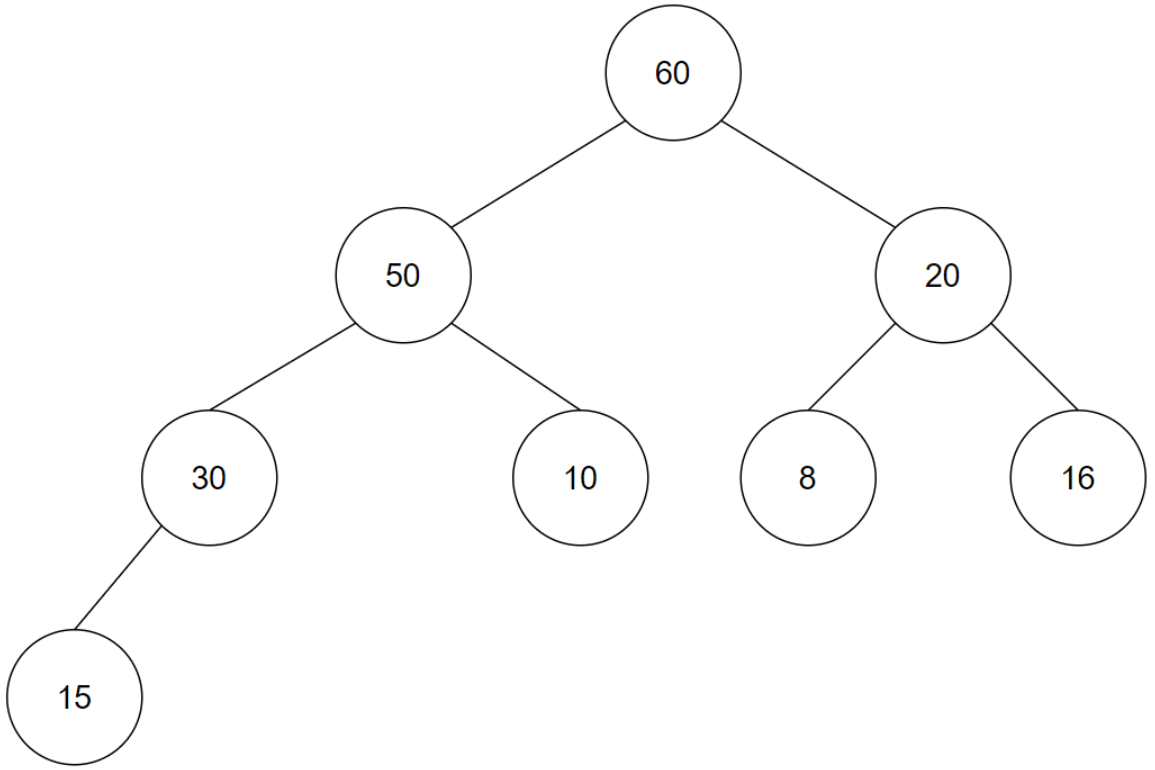


Sorting Using a Max Heap

What happens if we keep deleting the root from the heap?

When we delete the root, we are just taking it out of index 0 – we are not removing array element 0 – doing so just allows us to put the last element of the array in index 0.

What if we stash that 60 in that spot we just emptied?



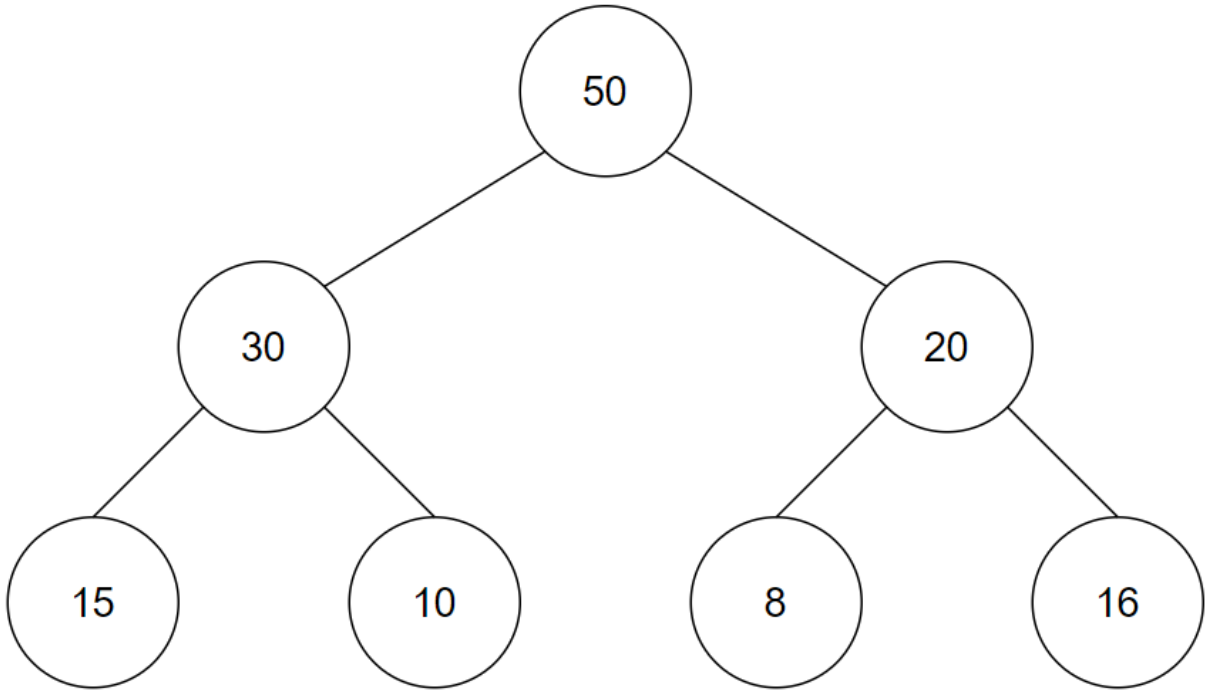
0	1	2	3	4	5	6	7
60	50	20	30	10	8	16	15

Sorting Using a Max Heap

So after heapifying, our heap is back to a good state and we make note that our array ends at element 6.

We are just using the space in element 7 to store that 60.

So what happens if we repeat this process?



0	1	2	3	4	5	6	7
50	30	20	15	10	8	16	60

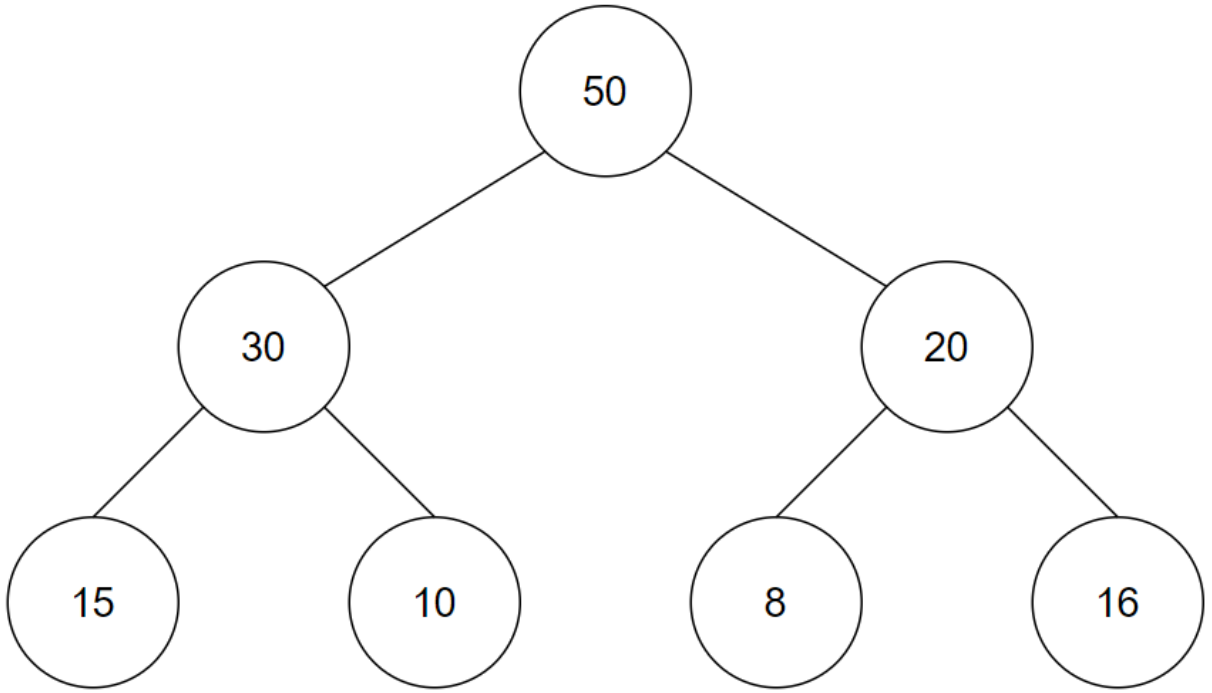
Sorting Using a Max Heap

We swap the 50 and the 16.

Heapify time!!

We make note that the end of our array is now element 5.

This also means that we only have 6 elements in our heap.

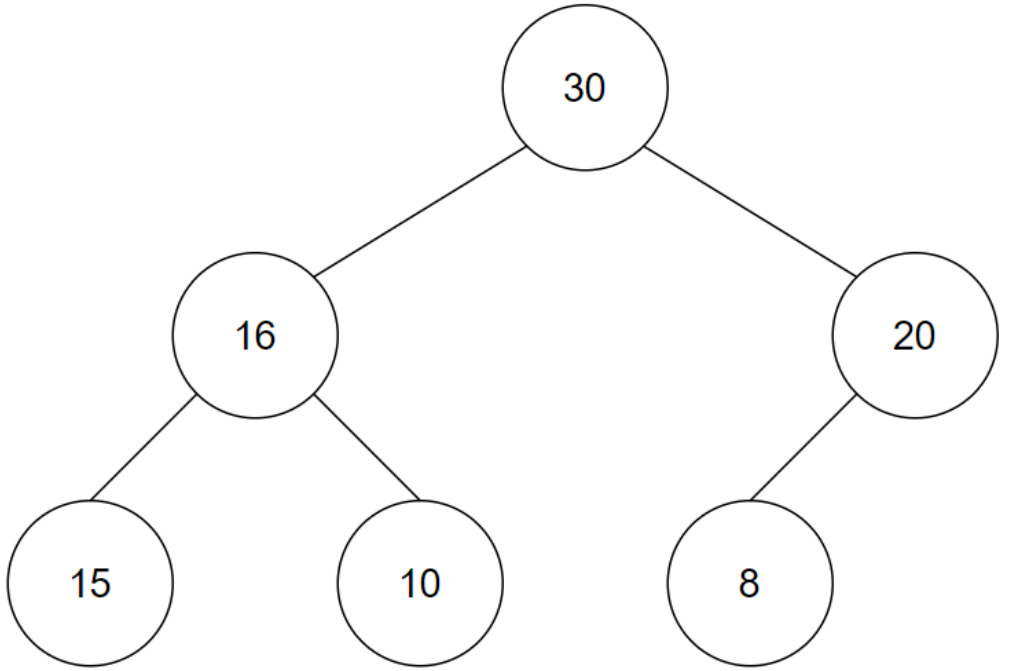


0	1	2	3	4	5	6	7
16	30	20	15	10	8	50	60

Sorting Using a Max Heap

Repeat.

Swap 30 and 8 and heapify.



0	1	2	3	4	5	6	7
30	16	20	15	10	8	50	60

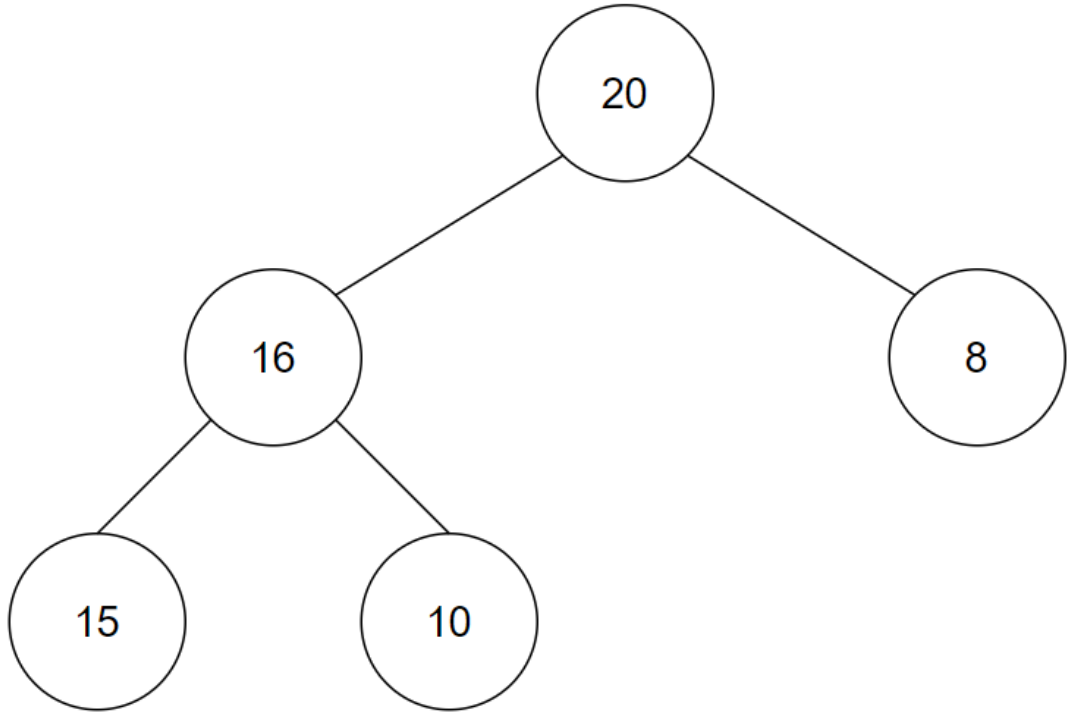
Sorting Using a Max Heap

Repeat.

Swap 20 and 10

Why not swap 20 and 8?

Heapify.



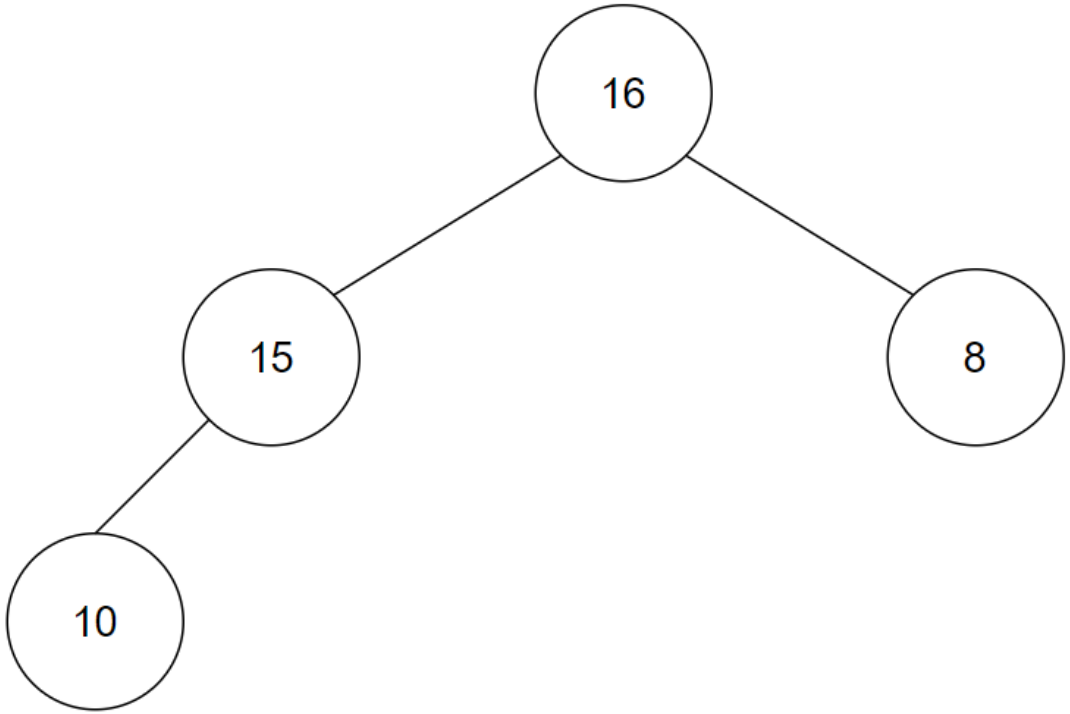
0	1	2	3	4	5	6	7
20	16	8	15	10	30	50	60

Sorting Using a Max Heap

Repeat.

Swap 16 and 10.

Heapify.



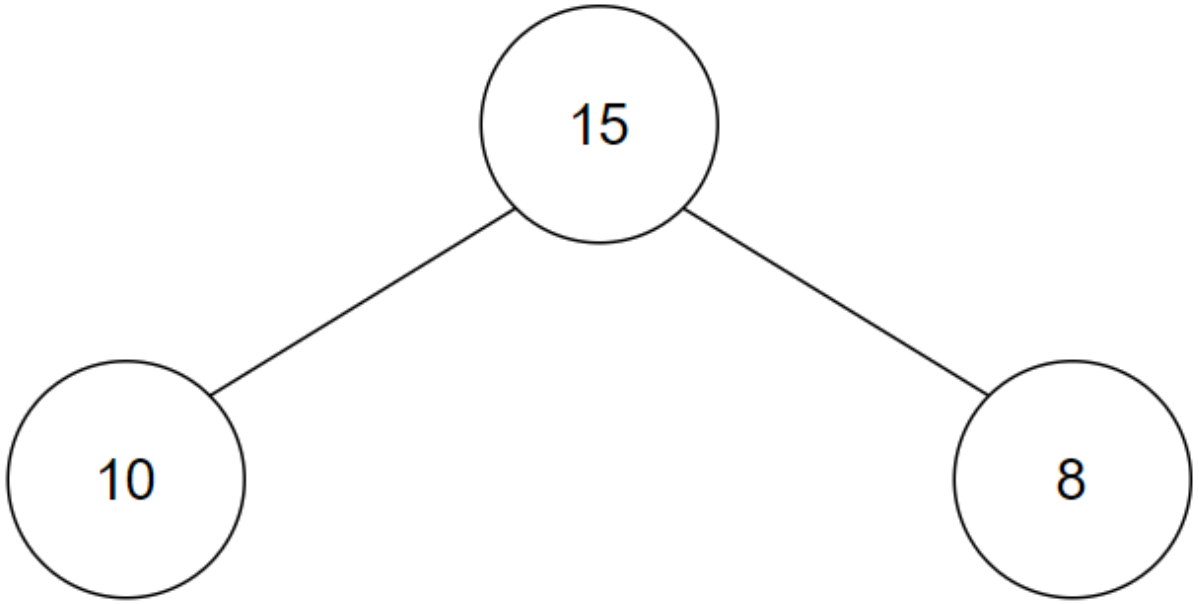
0	1	2	3	4	5	6	7
16	15	8	10	20	30	50	60

Sorting Using a Max Heap

Repeat.

Swap 8 and 15.

Heapify



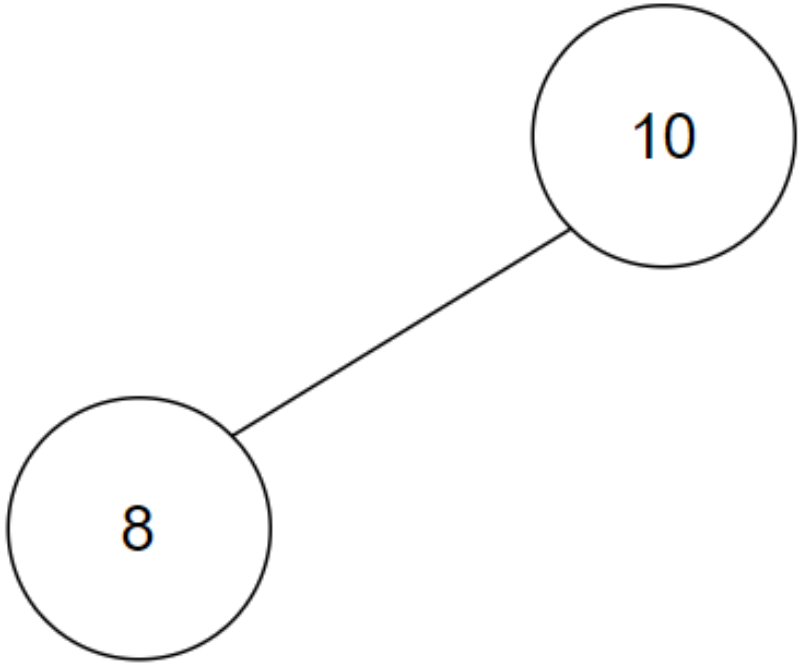
0	1	2	3	4	5	6	7
15	10	8	16	20	30	50	60

Sorting Using a Max Heap

Repeat.

Swap 10 and 8.

Do we need to heapify?

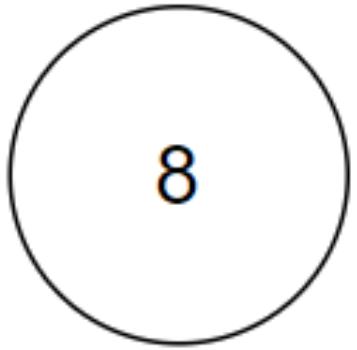


0	1	2	3	4	5	6	7
10	8	15	16	20	30	50	60

Sorting Using a Max Heap

No.

Final node maintains the heap property.

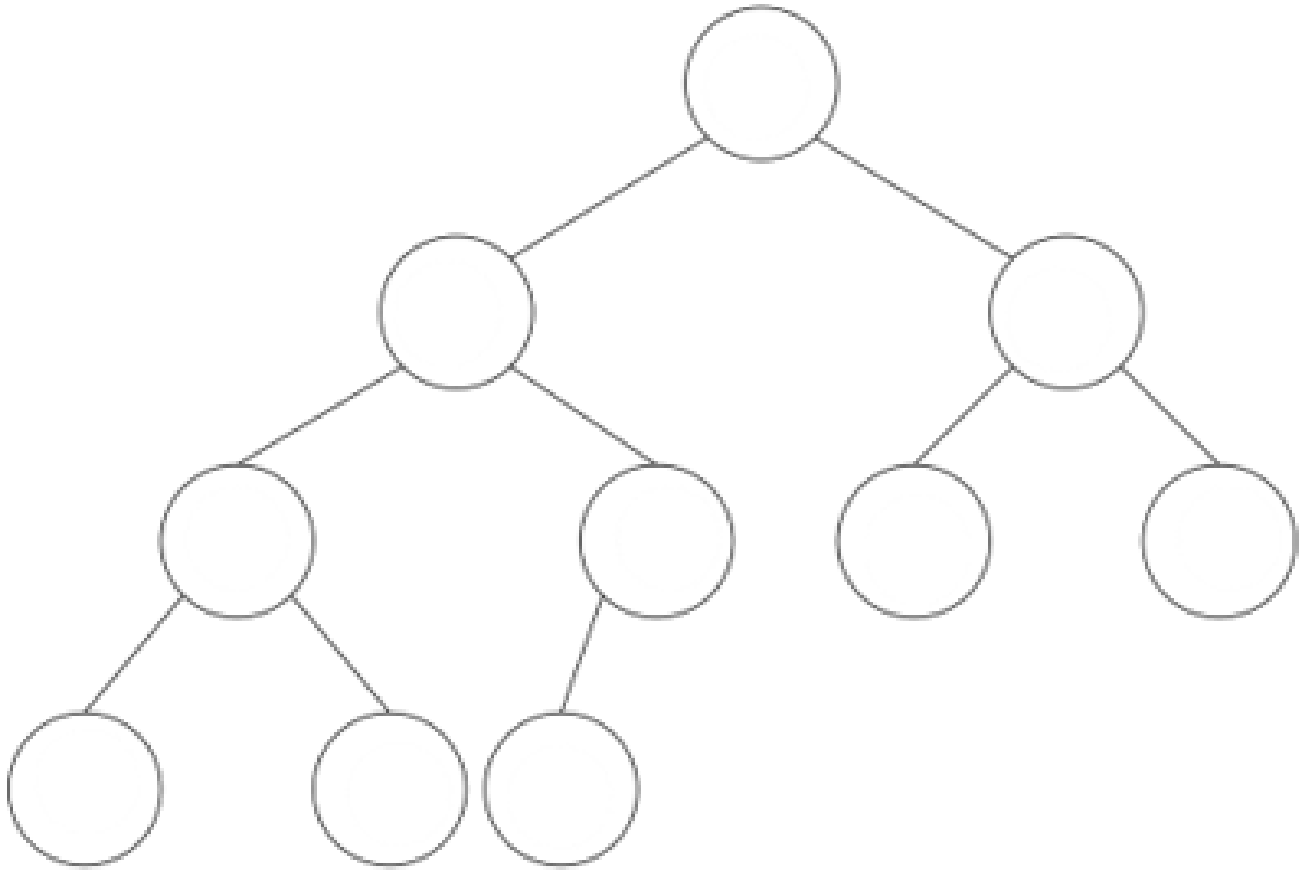


What is in our array now?

It is sorted.

0	1	2	3	4	5	6	7
8	10	15	16	20	30	50	60

1. Create the tree given the array.
2. Sort the array using a min heap delete



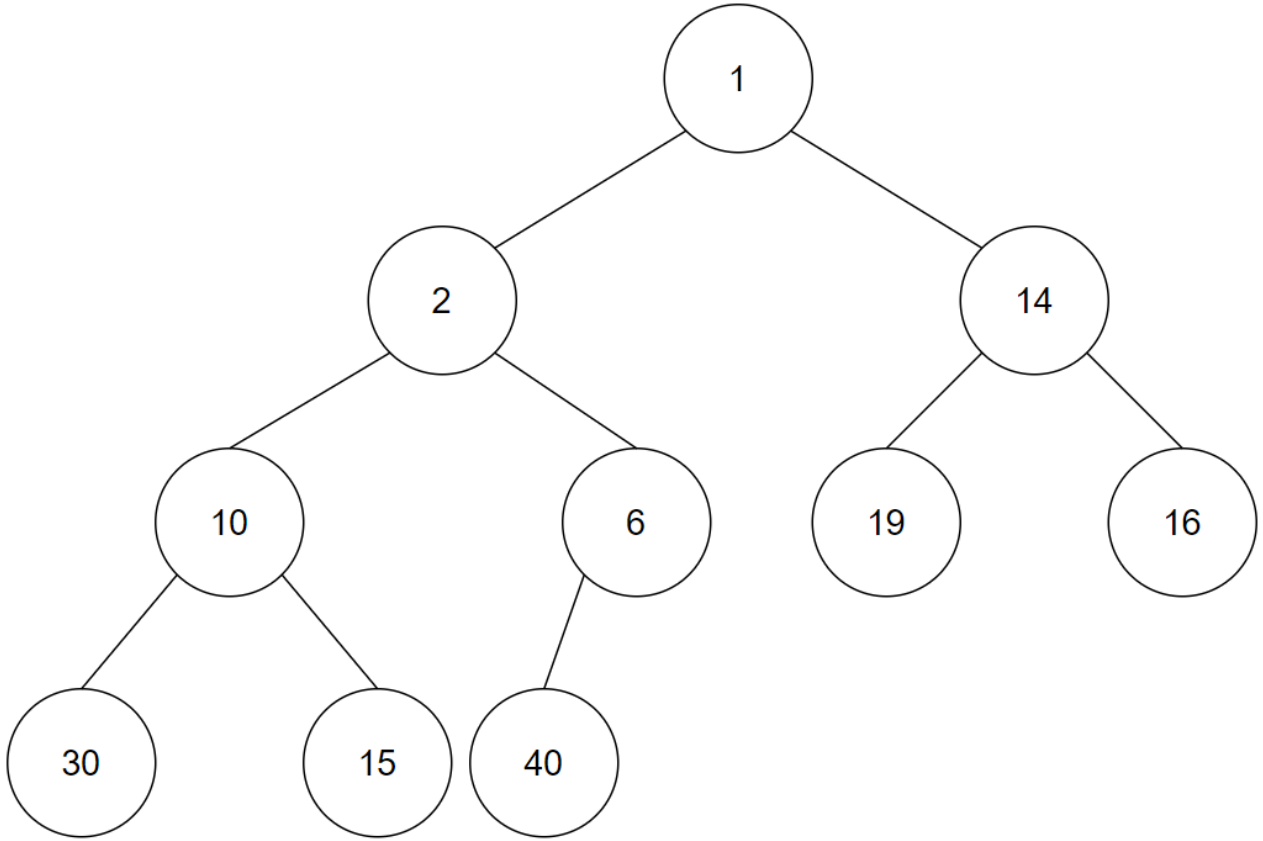
0	1	2	3	4	5	6	7	8	9
1	2	14	10	6	19	16	30	15	40

Sorting Using a Min Heap

Does the process work differently for a Min Heap?

Will the result be different?

Let's try it.

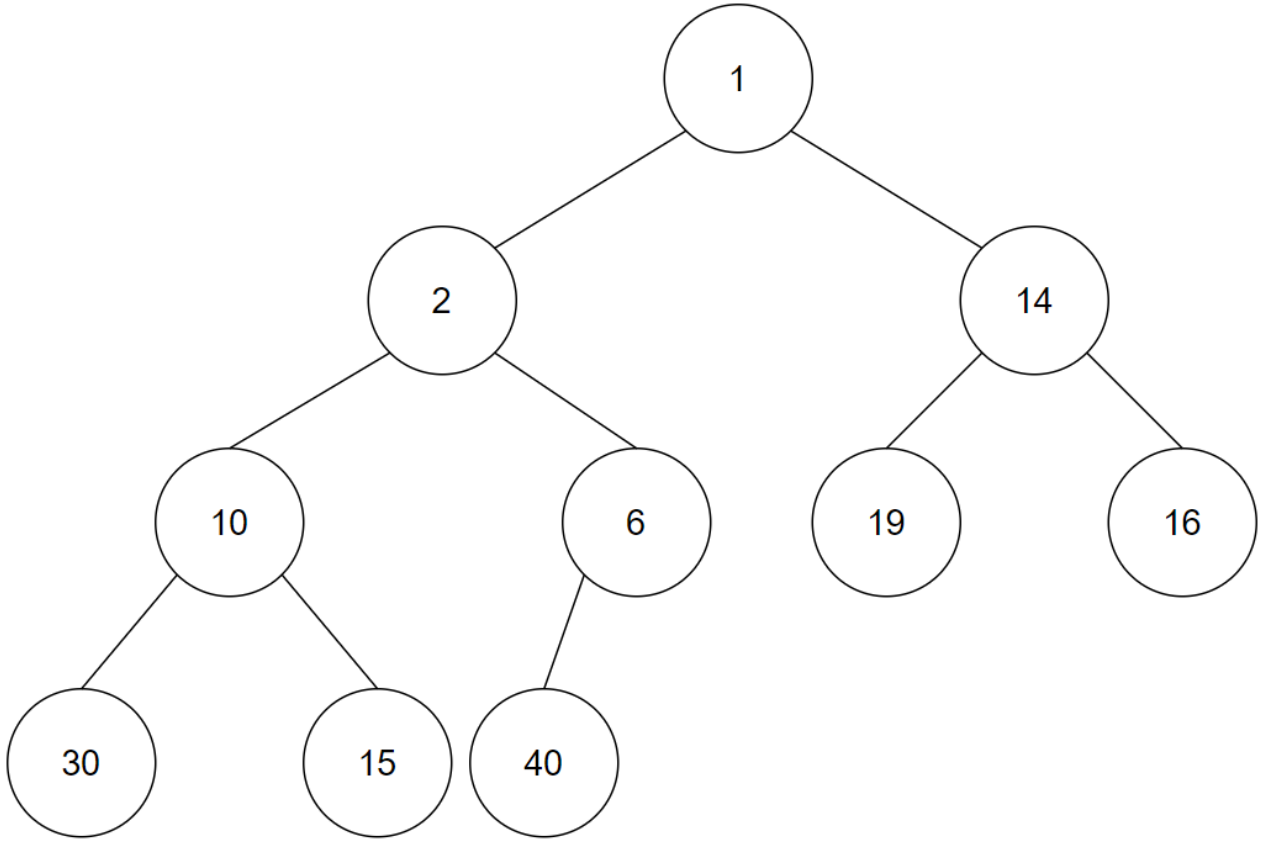


0	1	2	3	4	5	6	7	8	9
1	2	14	10	6	19	16	30	15	40

Sorting Using a Min Heap

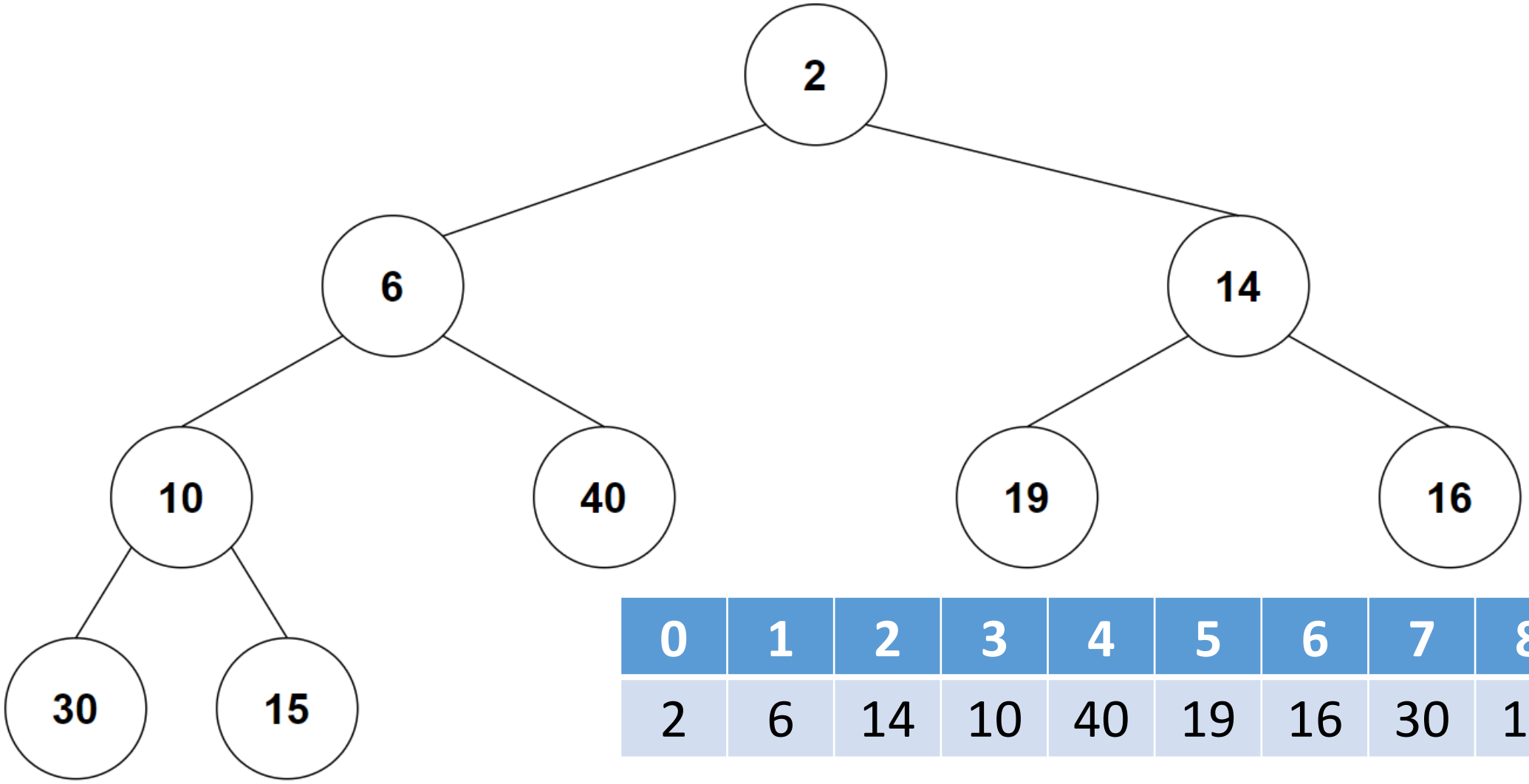
Swap 1 and 40

Heapify!



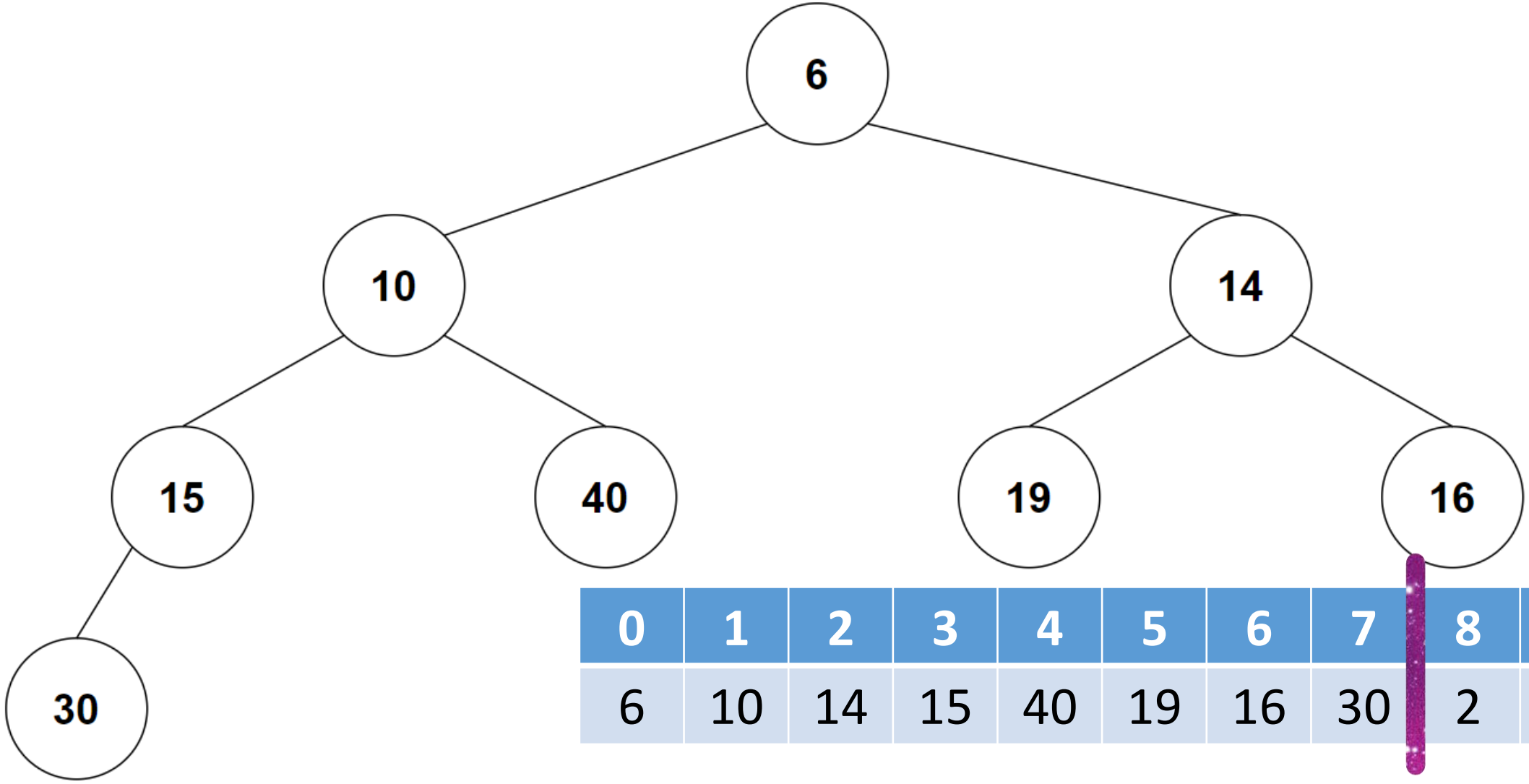
0	1	2	3	4	5	6	7	8	9
40	2	14	10	6	19	16	30	15	1

Sorting Using a Min Heap



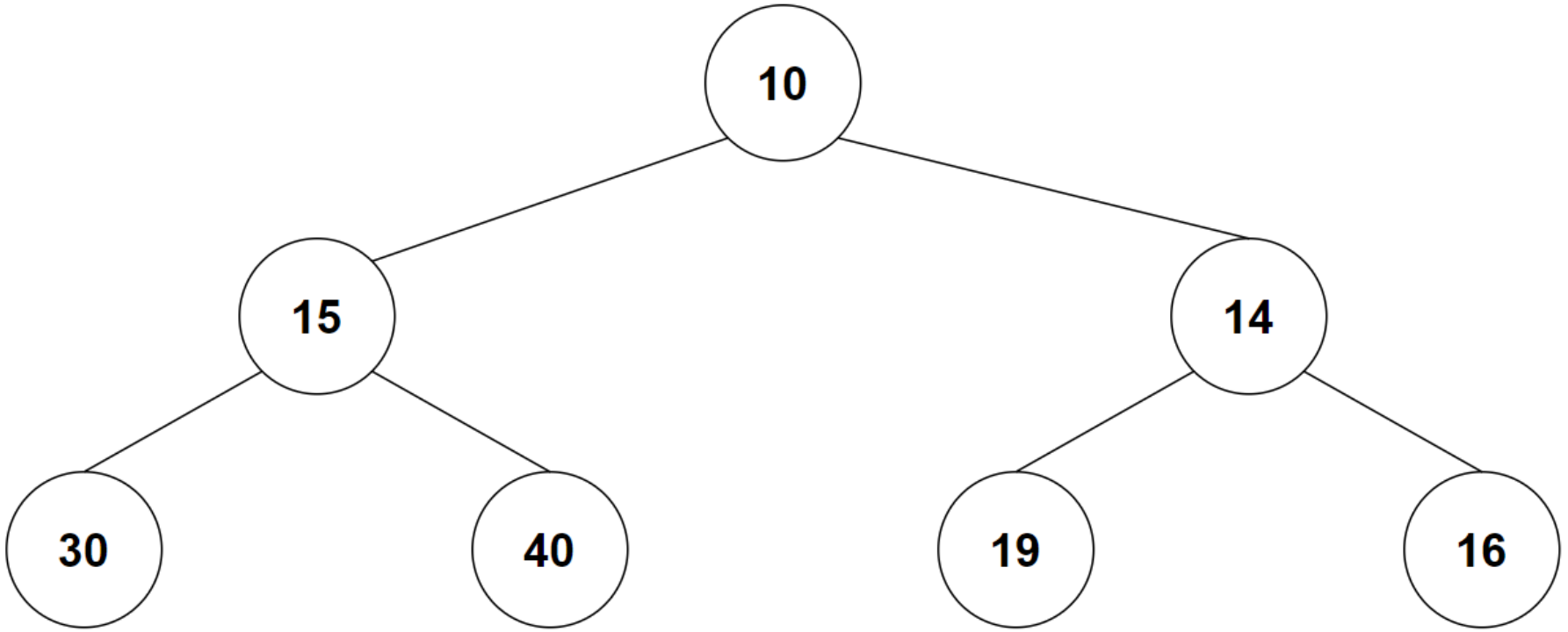
0	1	2	3	4	5	6	7	8	9
2	6	14	10	40	19	16	30	15	1

Sorting Using a Min Heap



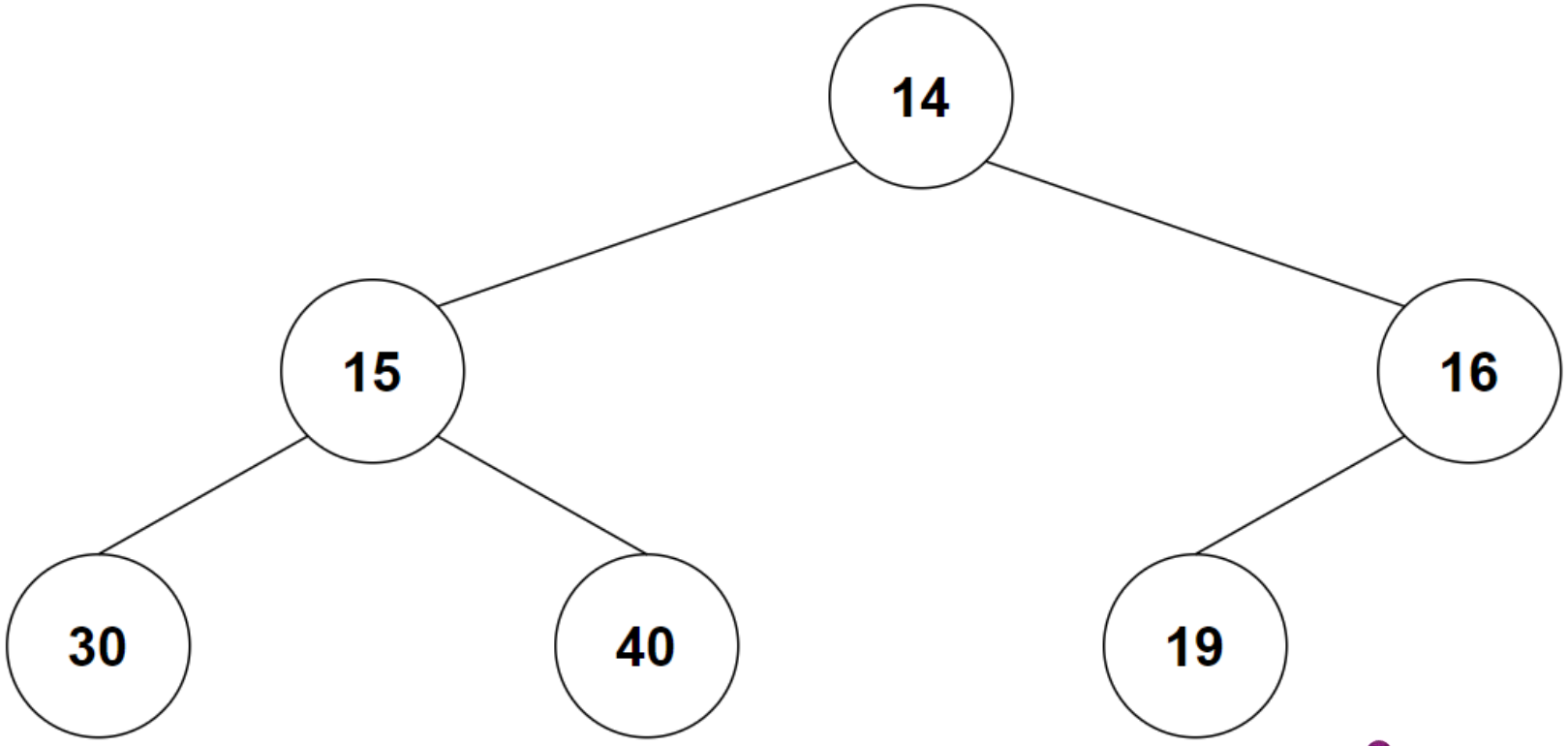
0	1	2	3	4	5	6	7	8	9
6	10	14	15	40	19	16	30	2	1

Sorting Using a Min Heap



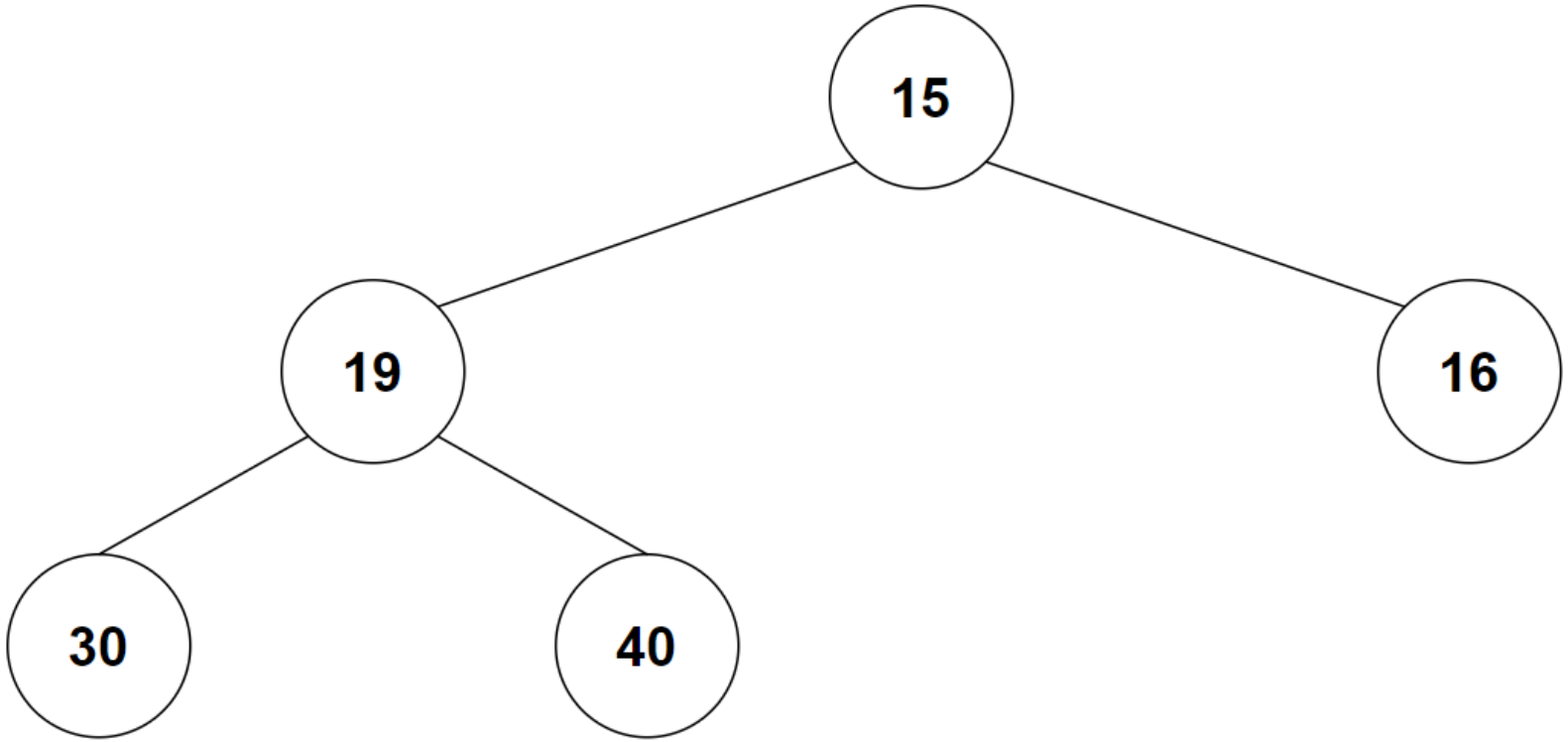
0	1	2	3	4	5	6	7	8	9
10	15	14	30	40	19	16	6	2	1

Sorting Using a Min Heap



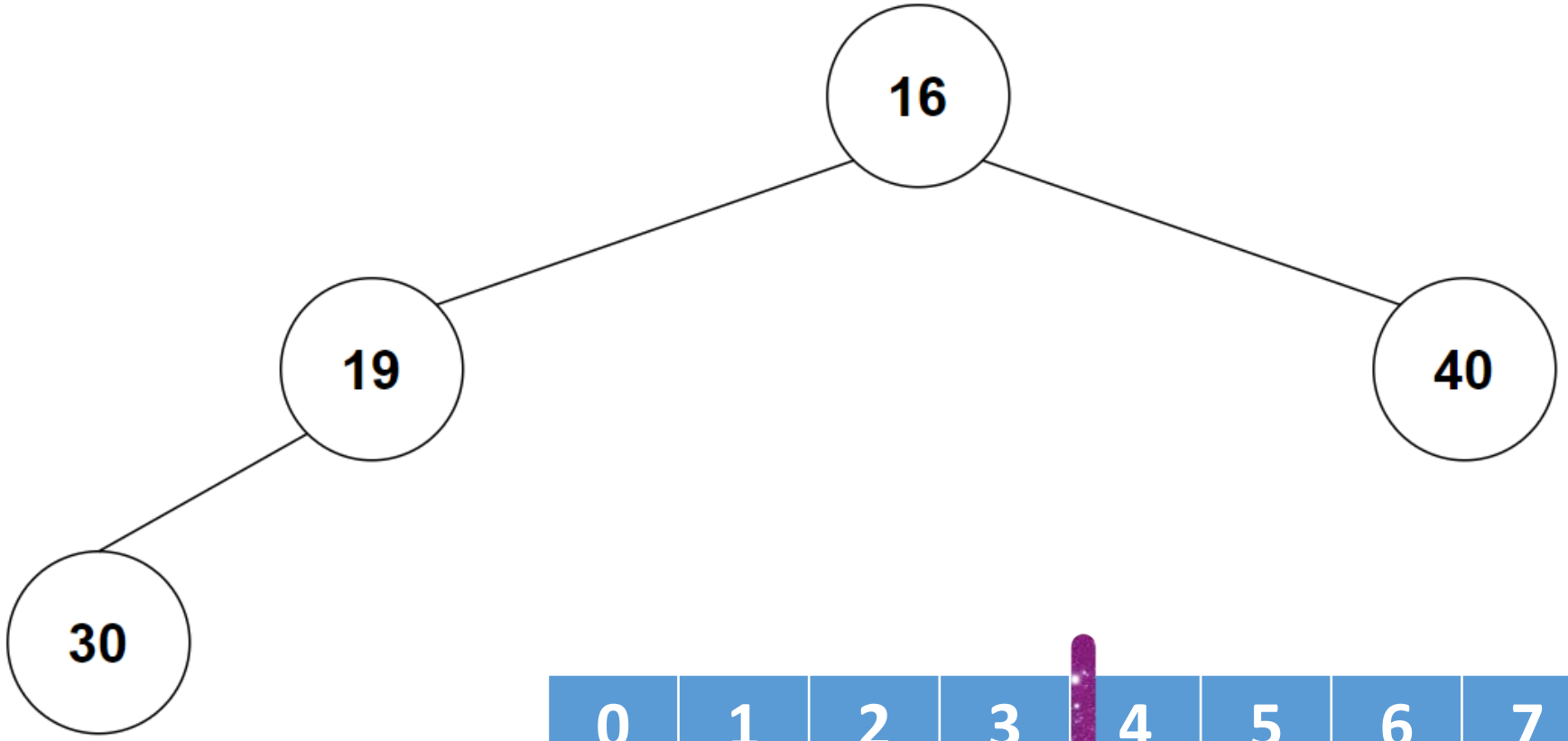
0	1	2	3	4	5	6	7	8	9
14	15	16	30	40	19	10	6	2	1

Sorting Using a Min Heap



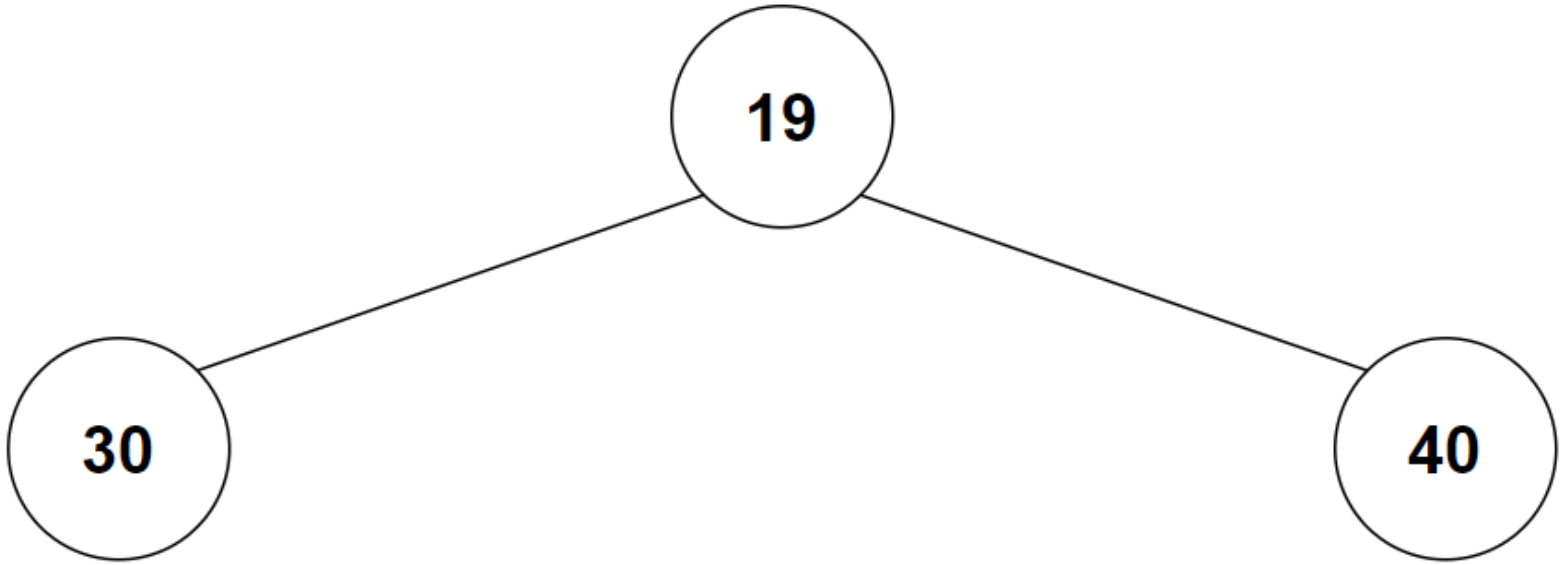
0	1	2	3	4	5	6	7	8	9
15	19	16	30	40	14	10	6	2	1

Sorting Using a Min Heap



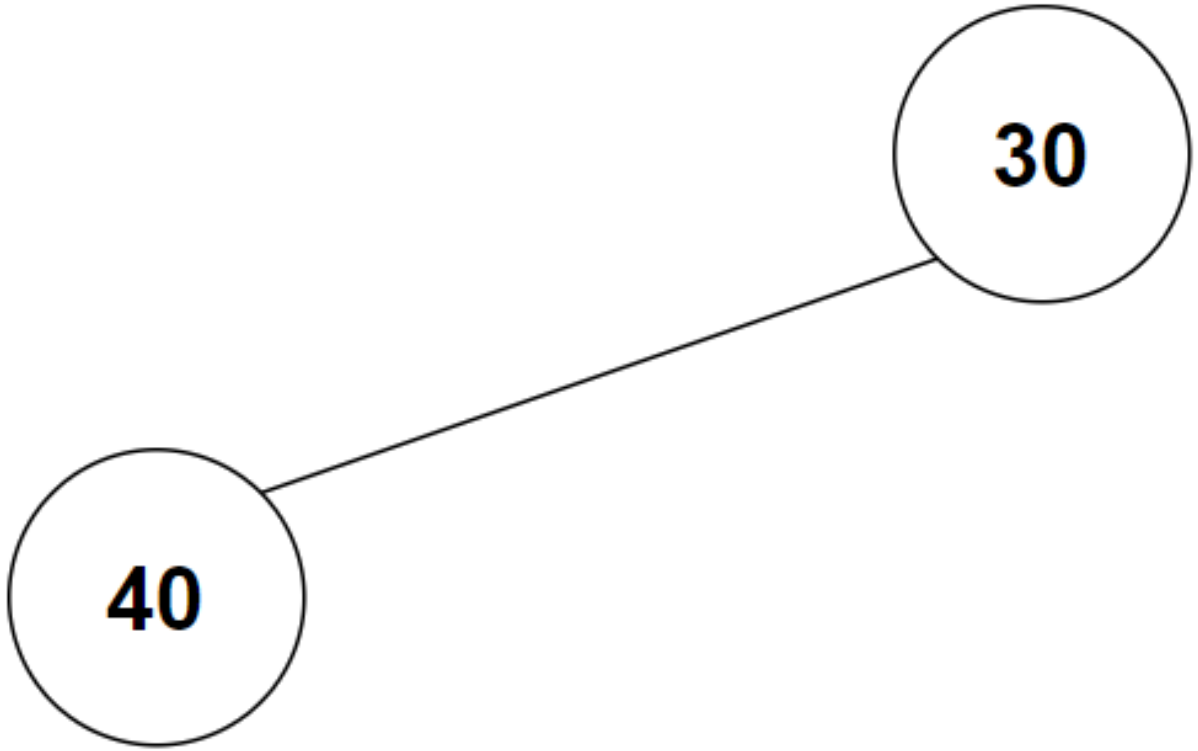
0	1	2	3	4	5	6	7	8	9
16	19	40	30	15	14	10	6	2	1

Sorting Using a Min Heap



0	1	2	3	4	5	6	7	8	9
19	30	40	16	15	14	10	6	2	1

Sorting Using a Min Heap



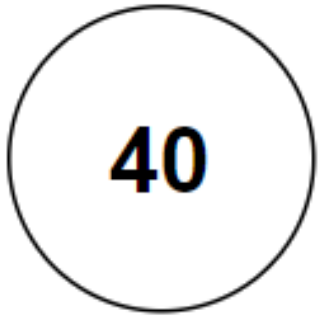
0	1	2	3	4	5	6	7	8	9
30	40	19	16	15	14	10	6	2	1

Sorting Using a Min Heap

Summary

Binary Max Heap sorted the list ascending order.

Binary Min Heap sorted the list in descending order.

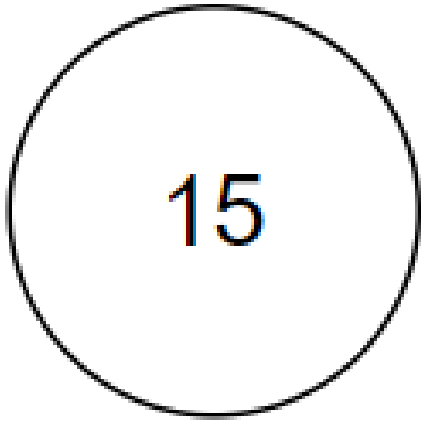


0	1	2	3	4	5	6	7	8	9
40	30	19	16	15	14	10	6	2	1

Creating a Max Heap

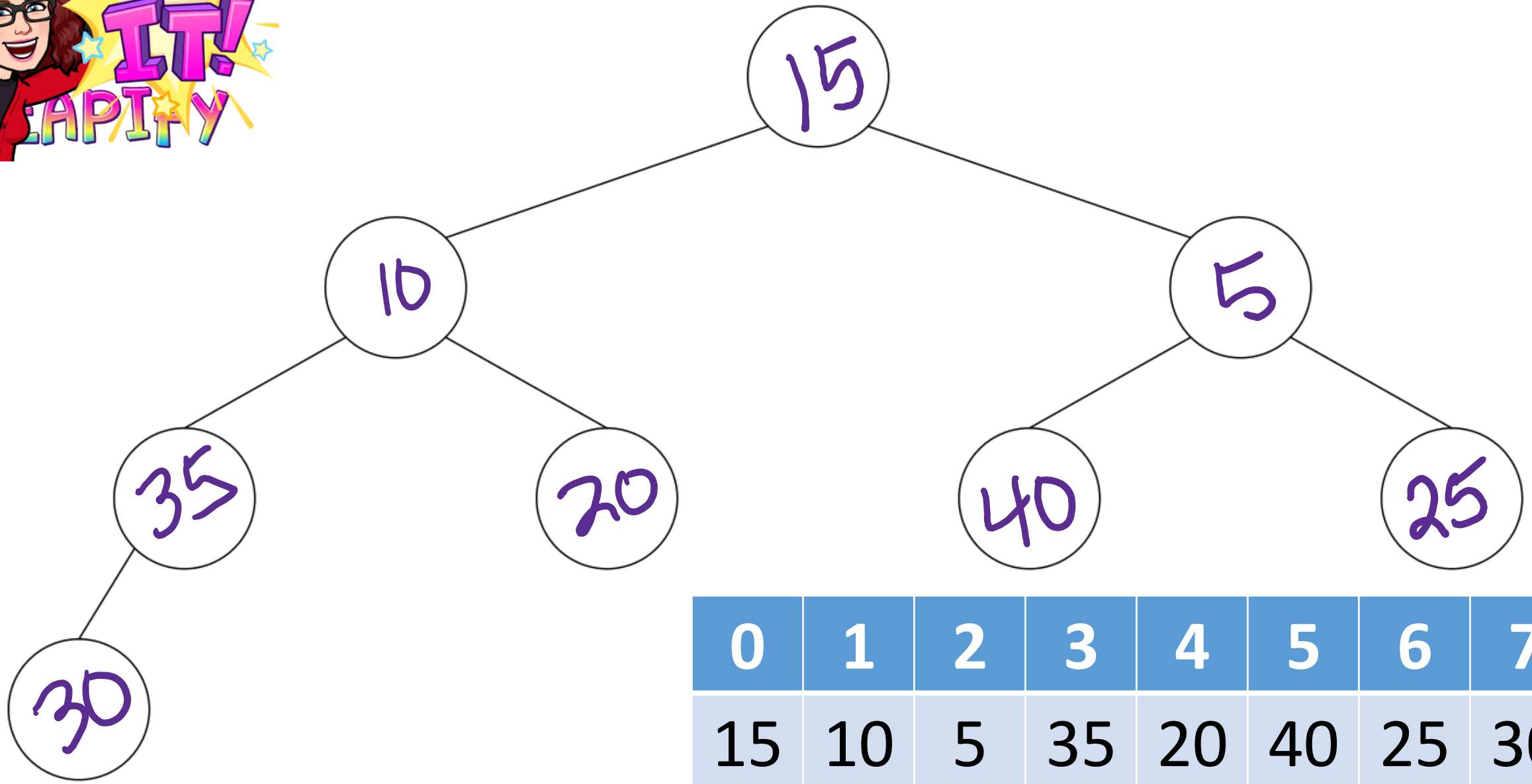
Given this array, create a max heap

0	1	2	3	4	5	6	7
15	10	5	35	20	40	25	30



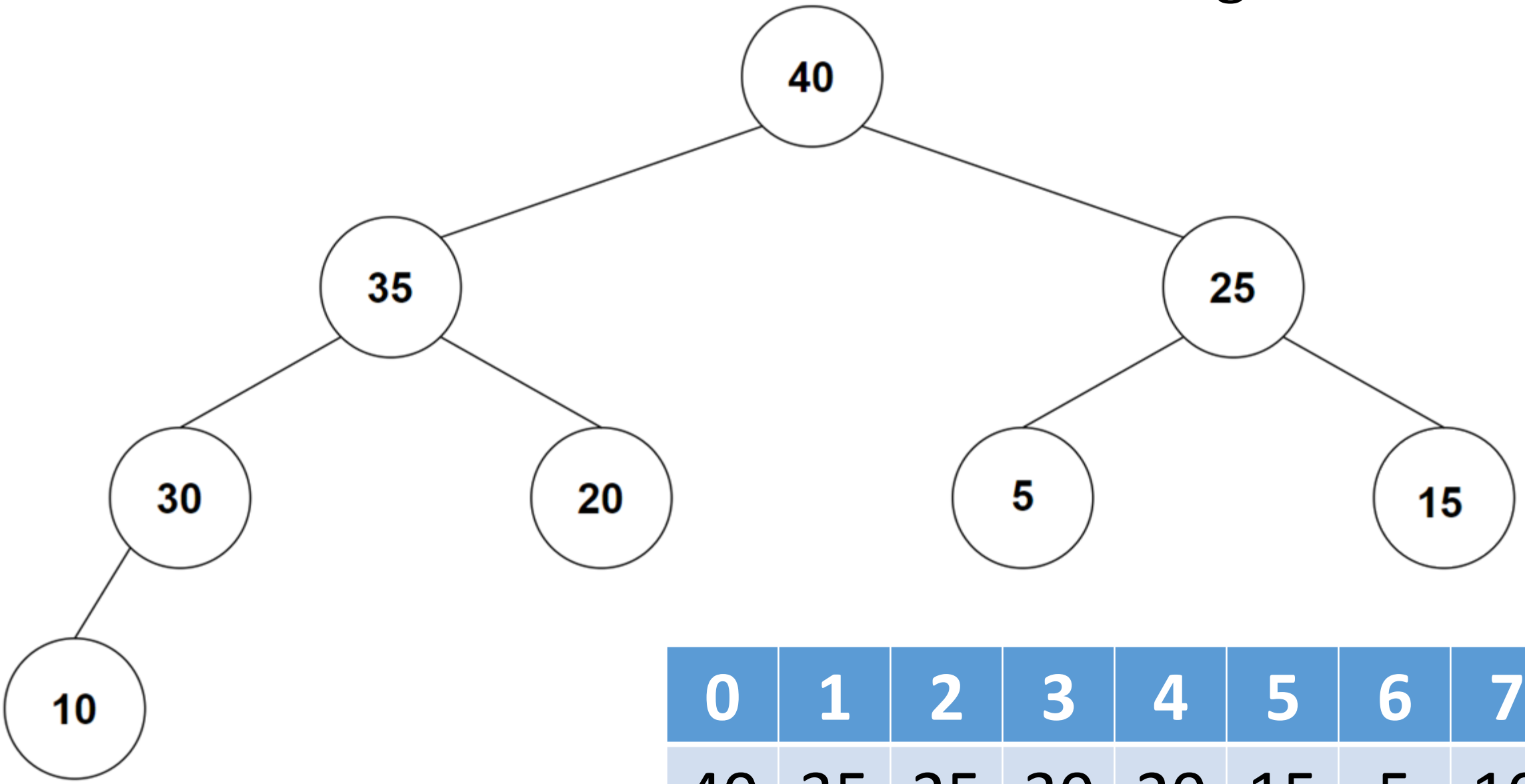
Take the 0^{th} element and make it the root.

Creating a Max Heap



0	1	2	3	4	5	6	7
15	10	5	35	20	40	25	30

Creating a Max Heap

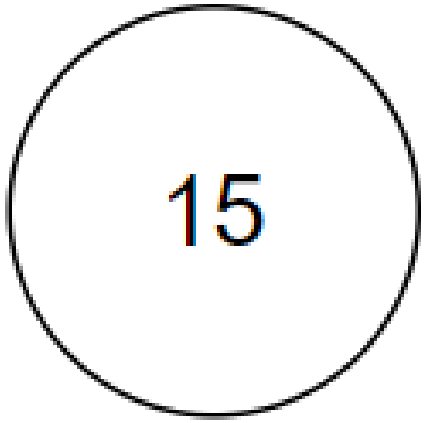


0	1	2	3	4	5	6	7
40	35	25	30	20	15	5	10

Creating a Min Heap

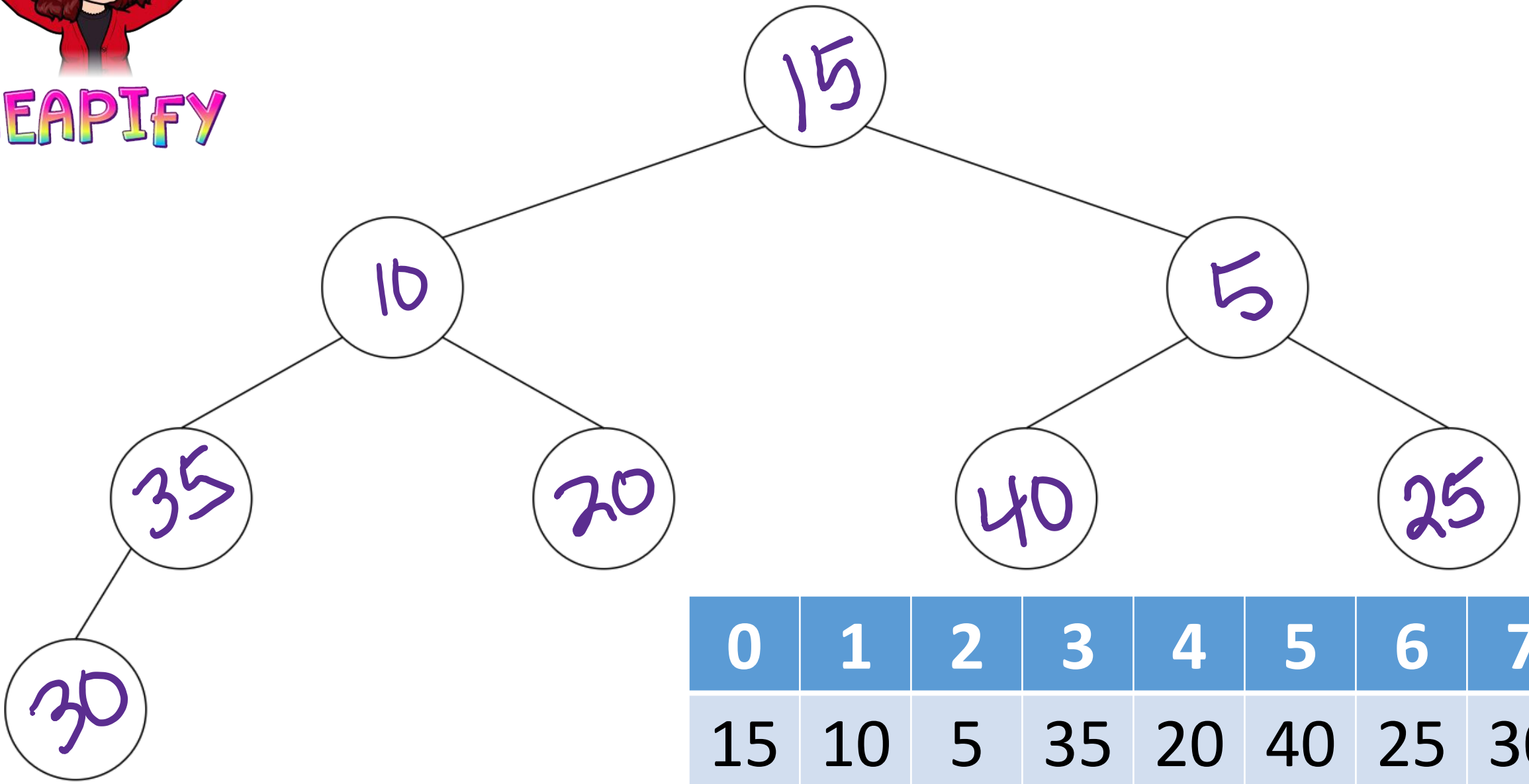
Given this array, create a min heap

0	1	2	3	4	5	6	7
15	10	5	35	20	40	25	30



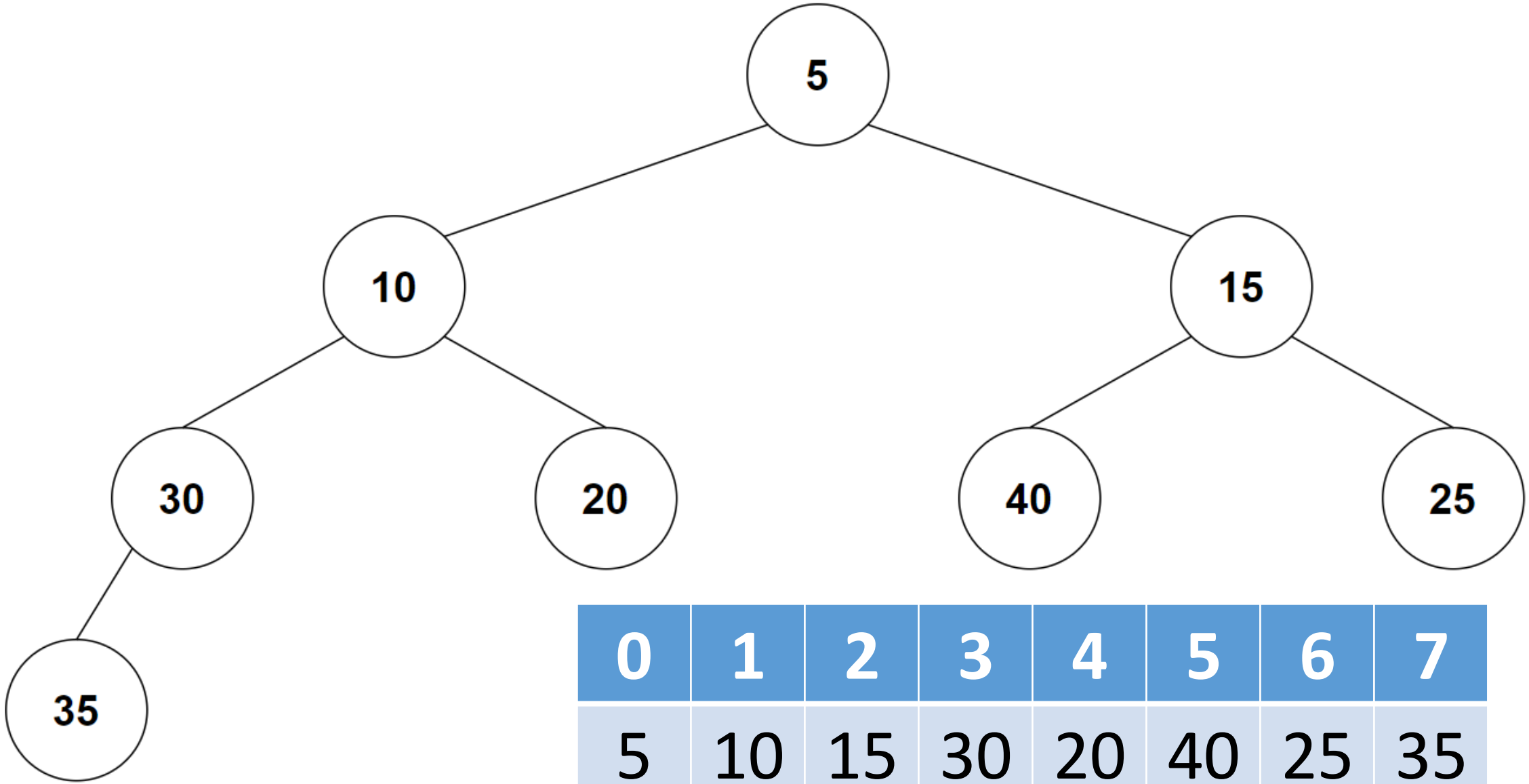
Take the 0^{th} element and make it the root.

Creating a Min Heap



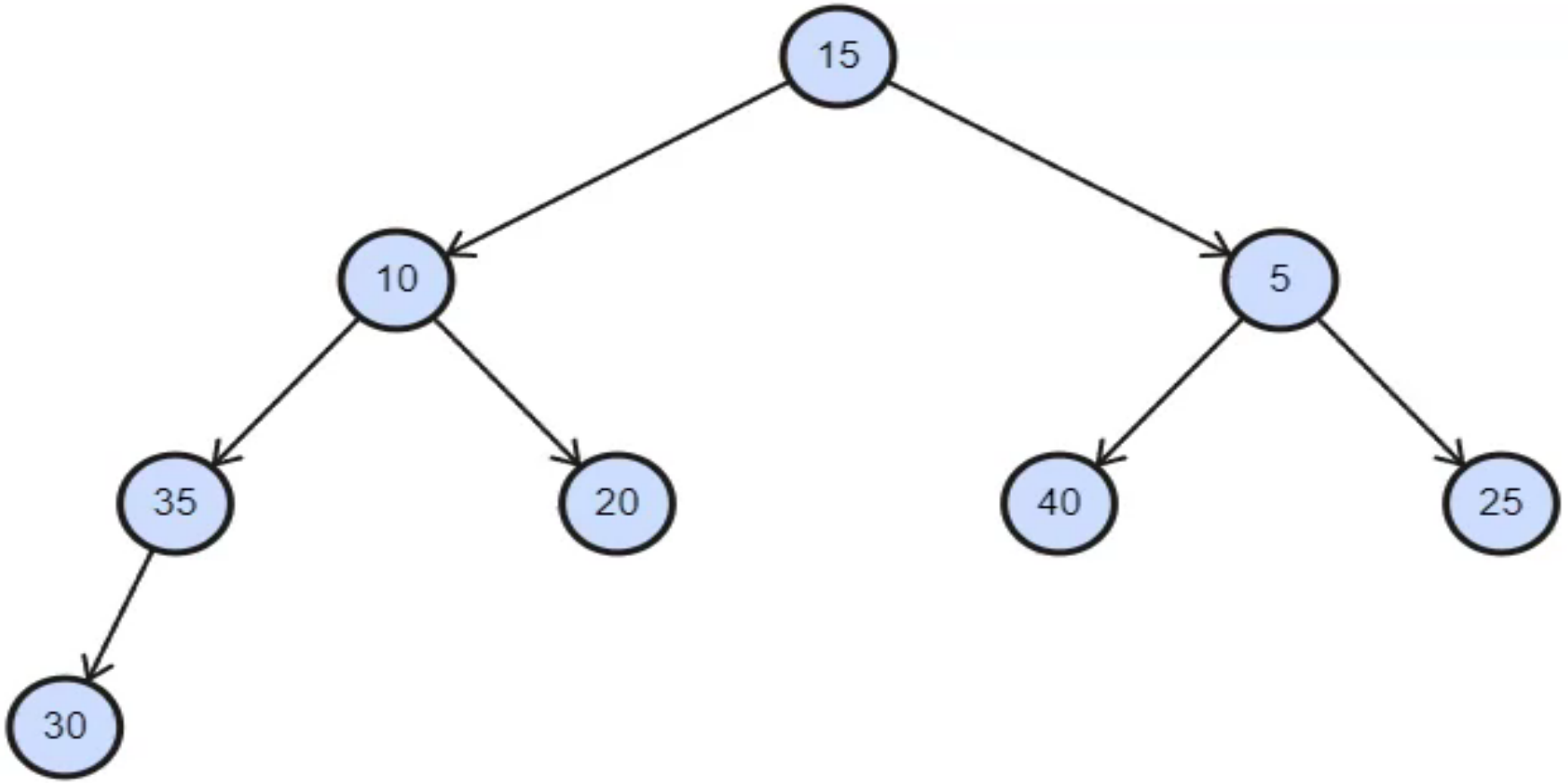
0	1	2	3	4	5	6	7
15	10	5	35	20	40	25	30

Creating a Min Heap



0	1	2	3	4	5	6	7
5	10	15	30	20	40	25	35

Sorting Using a Max Heap



Time Complexity

What is the time complexity of building a binary heap?

How many elements did we insert?

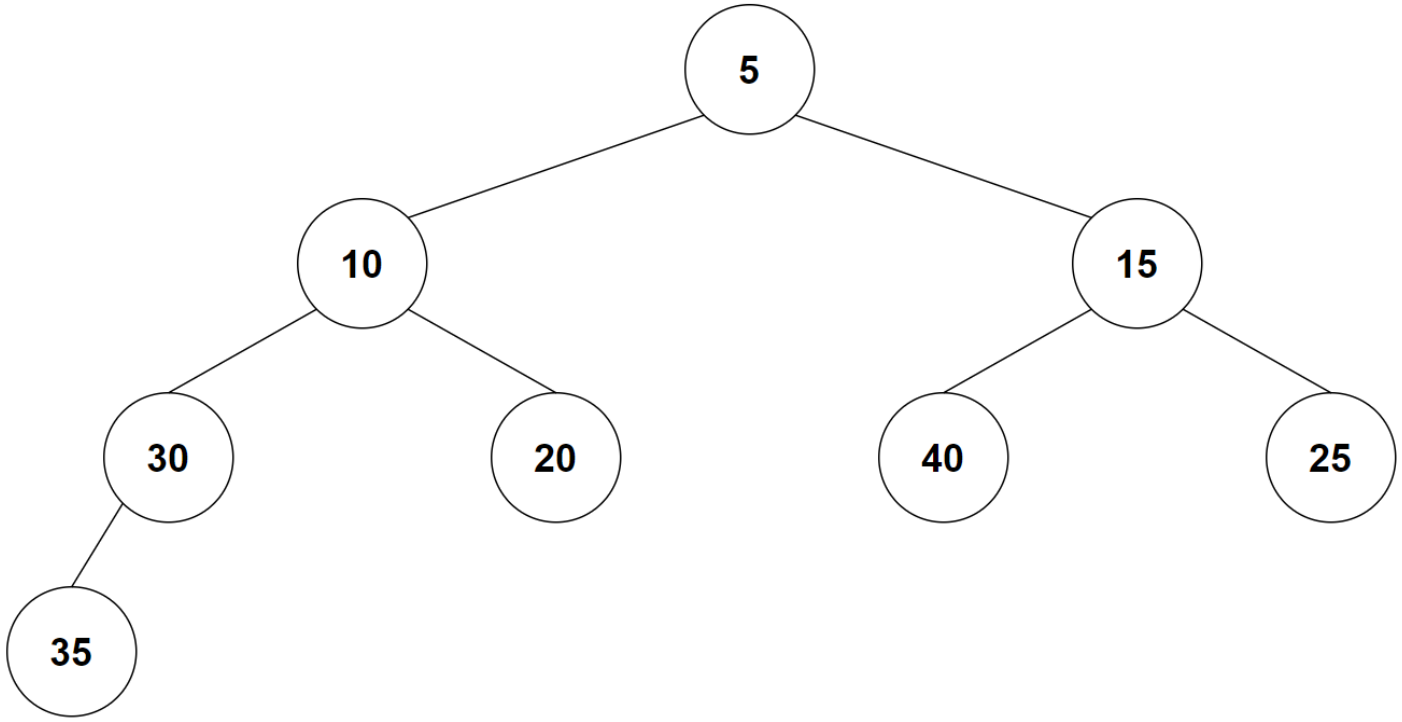
n

The amount of time needed to insert an element into a heap depends on the height of the complete binary tree.

$\log_2 n$

So for worst case, we assume that every element is moved up/down the full height of the tree.

$n\log_2 n$



0	1	2	3	4	5	6	7
5	10	15	30	20	40	25	35

Time Complexity

What is the time complexity of deleting a binary heap?

How many elements did we delete?

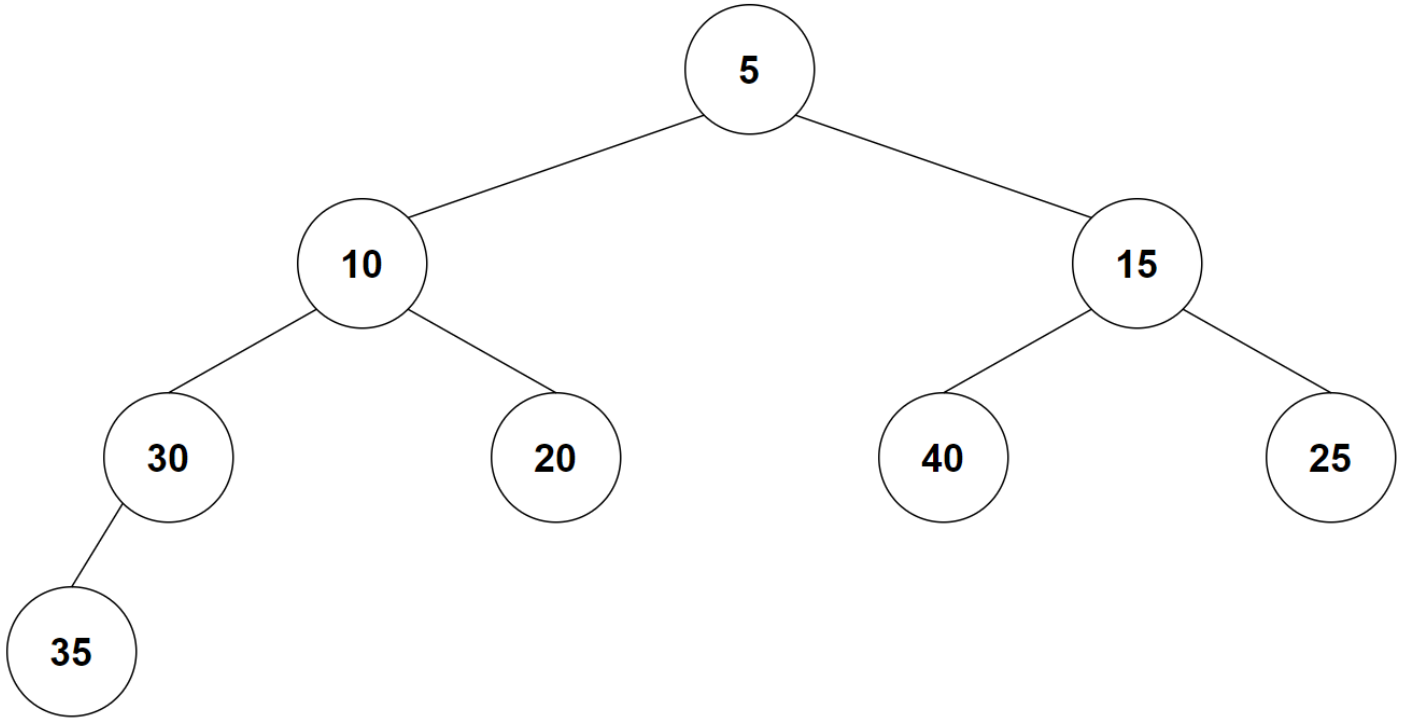
n

The amount of time needed to delete an element into a heap depends on the height of the complete binary tree.

$\log_2 n$

So for worst case, we assume that every element is moved up/down the full height of the tree.

$n\log_2 n$



0	1	2	3	4	5	6	7
5	10	15	30	20	40	25	35

Time Complexity

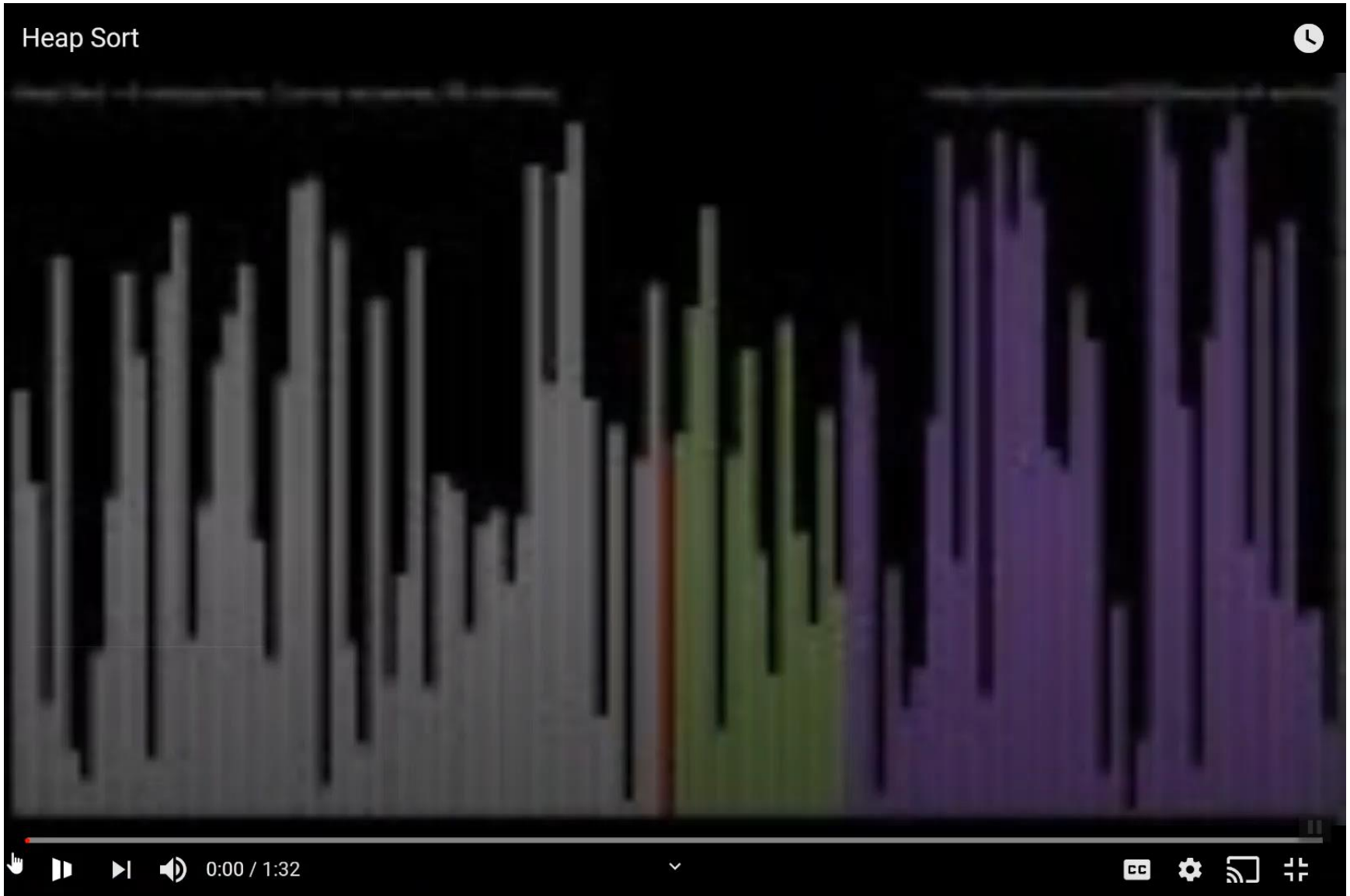
What is the time complexity of sorting using a binary heap?

We sorted by building a binary heap with the data and then deleting it to get the sort.

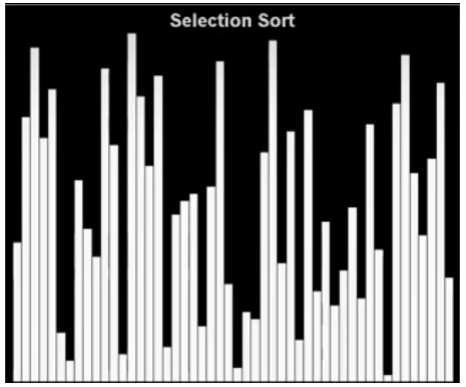
Building	+	Deleting	=	Sorting
$n\log_2 n$	+	$n\log_2 n$	=	$2*n\log_2 n$

$2n\log_2 n = O(n\log_2 n)$ for Heap Sort

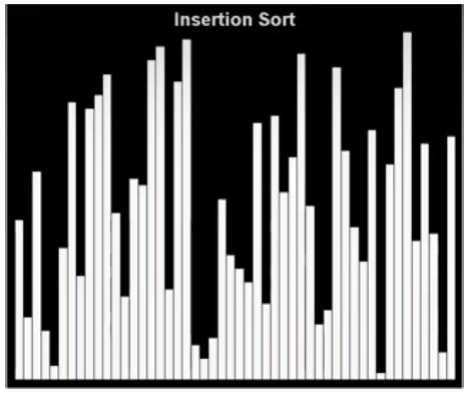
Visual Heap Sort



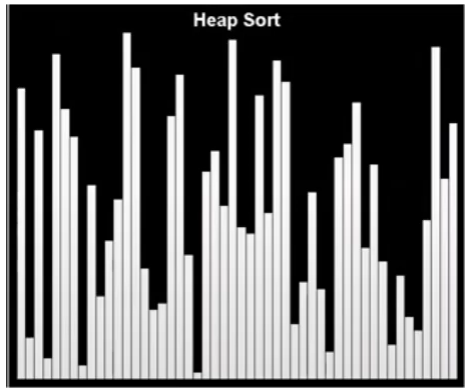
Sorts



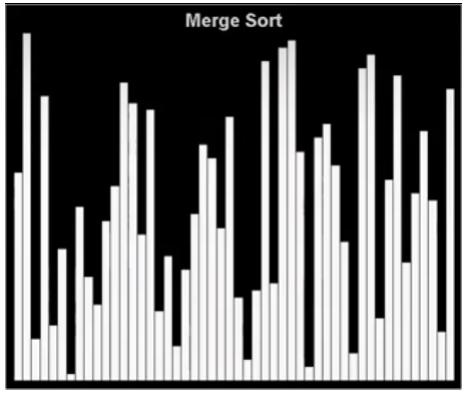
$O(n^2)$



$O(n^2)$

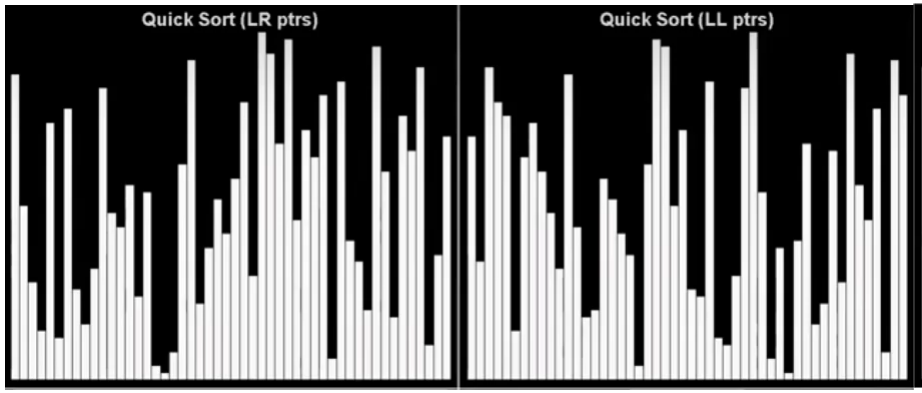


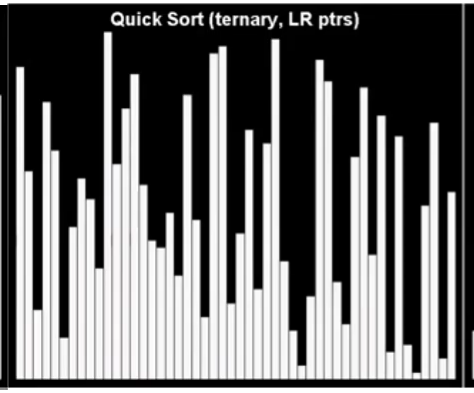
$O(n\log_2 n)$

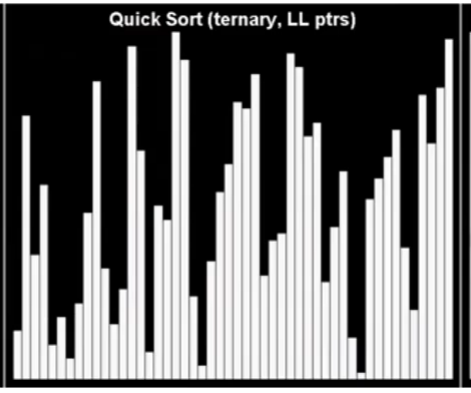


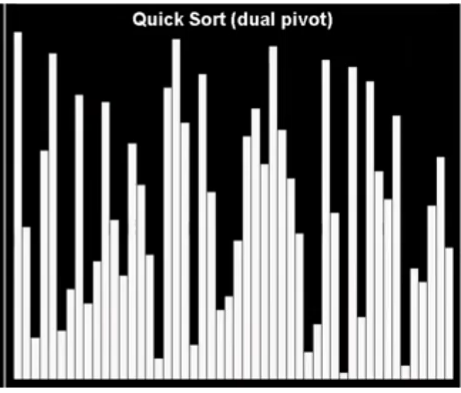
$O(n\log_2 n)$

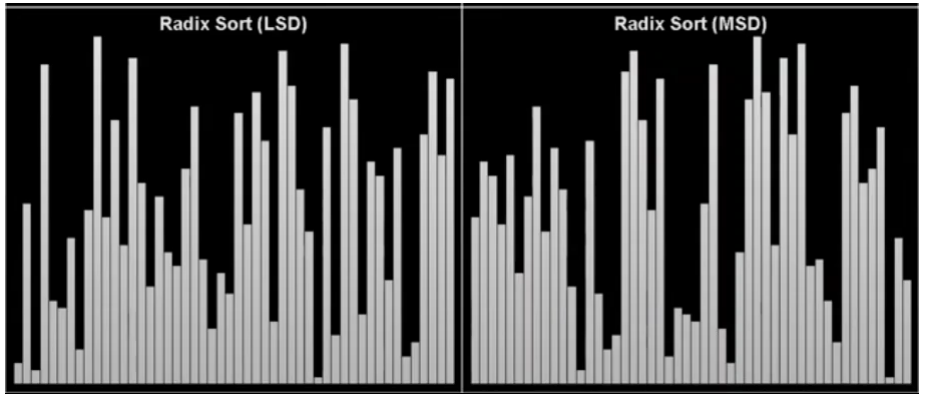
$O(n\log_2 n)$
$O(n^2)$

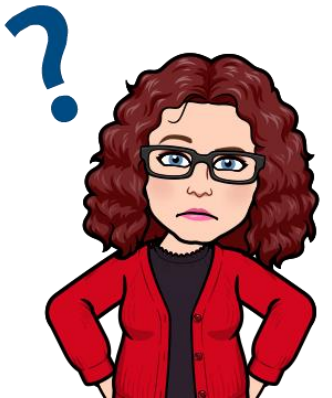












Radix Sort

Let's start with a list of numbers to sort

645, 86, 182, 35, 601, 8

Step 1

All numbers to be sorted need to be the same length.

Radix Sort

645, 86, 182, 35, 601, 8

How do we make 8 and 35 and 86 the same length as 648, 182 and 608?

645, 086, 182, 035, 601, 008

We add leading zeros/front pad our numbers with zeroes.

No difference in value between 001 and 1.

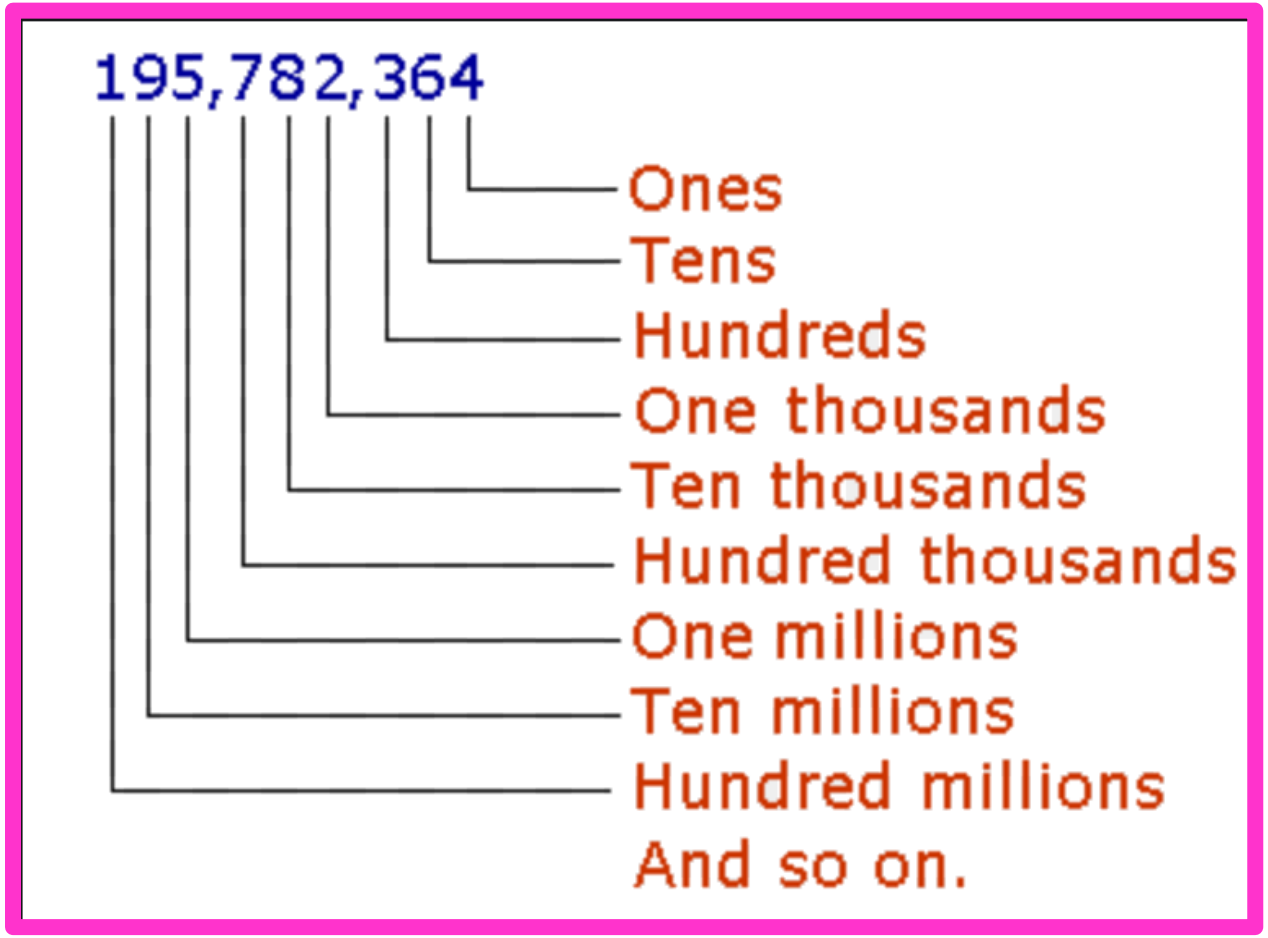
Big difference between 1 and 100 – add leading zeros to not change value.

Radix Sort

645, 086, 182, 035, 601, 008

Now we arrange our list by place value starting with ones.

601, 182, 645, 035, 086, 008

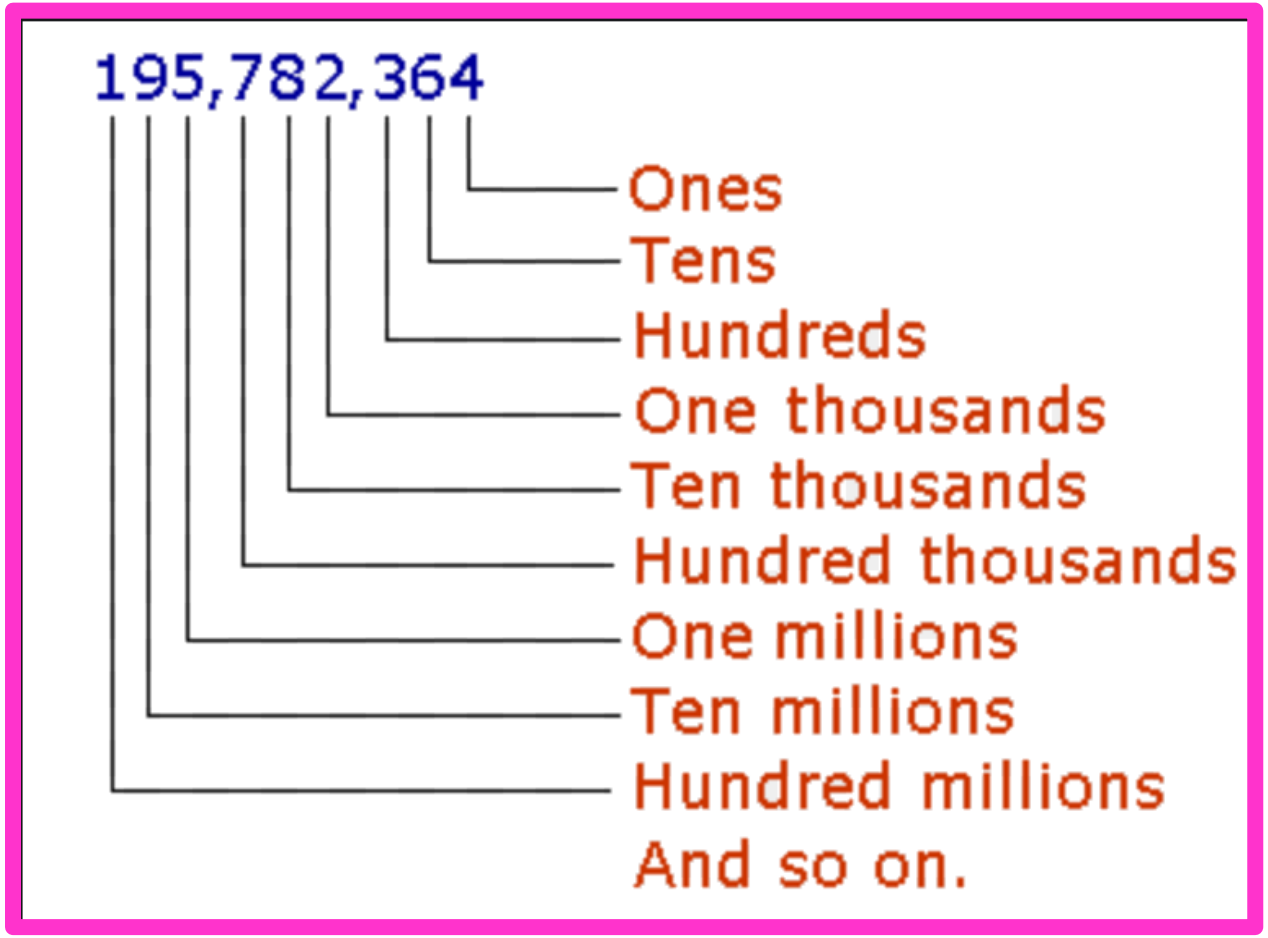


Radix Sort

601, 182, 645, 035, 086, 008

Now we arrange our list by place value tens.

601, 008, 035, 645, 182, 086

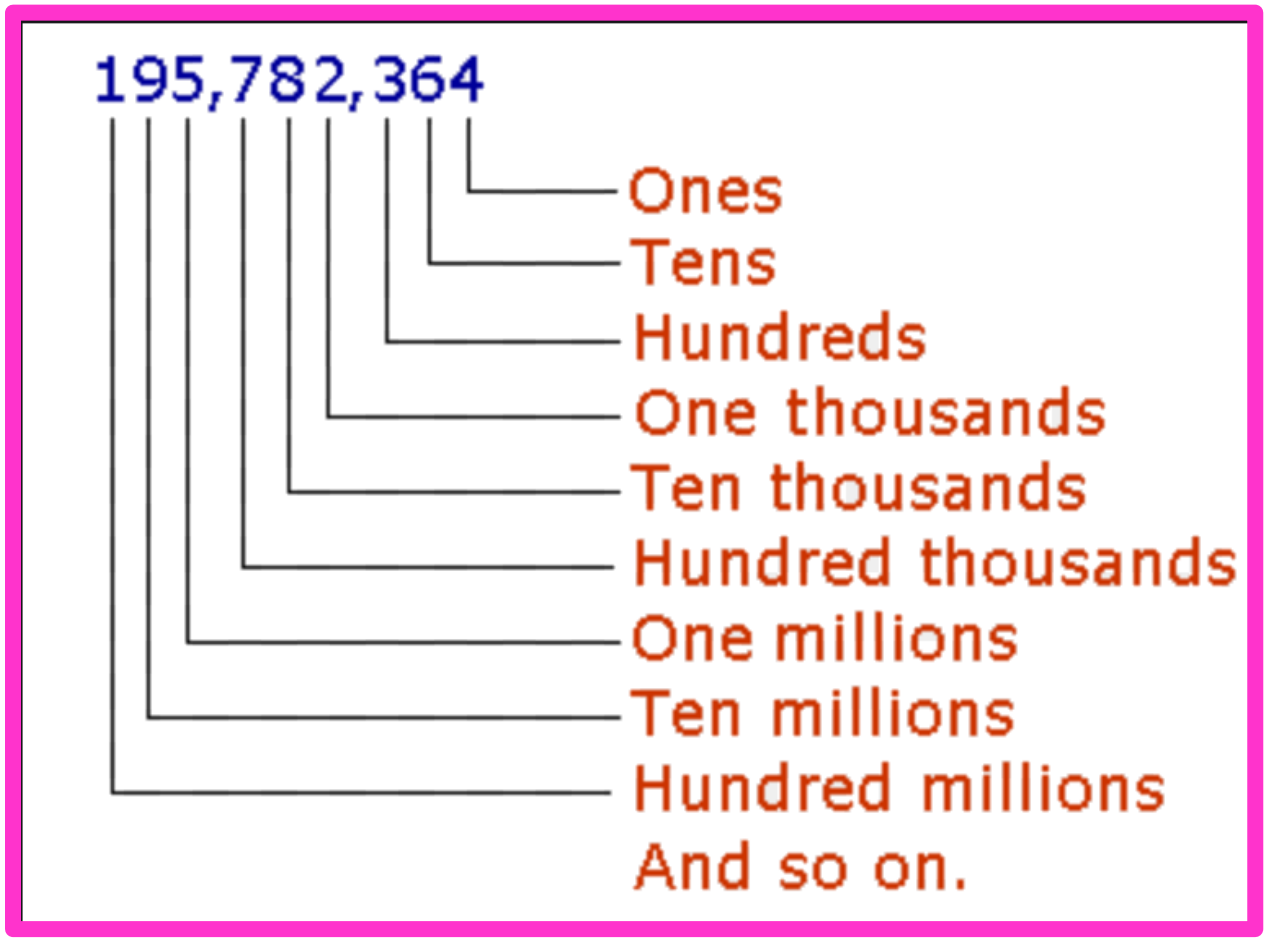


Radix Sort

601, 008, 035, 645, 182, 086

Now we arrange our list by place value hundreds.

008, 035, 086, 182, 601, 645



Radix Sort

645, 086, 182, 035, 601, 008

601, 182, 645, 035, 086, 008

601, 008, 035, 645, 182, 086

008, 035, 086, 182, 601, 645

Radix Sort

Radix Definition

The **base** of a system of numeration

So "Radix Sort" means a sort that uses the **base** of a system of numeration.

So can we use Radix Sort on numbers in bases other than 10?

Radix Sort

Let's start with a list of binary numbers

110, 11, 101, 1, 100, 111 , 10

Add leading zeros

110, 011, 101, 001, 100, 111 , 010

110, 011, 101, 001, 100, 111 , 010

Using ones, tens and hundreds place value is more applicable to the decimal system.

So let's talk about Most Significant Bit (MSB) and Least Significant Bit (LSB).

Radix Sort

MSB

the bit in a binary number which is of the greatest numerical value
the leftmost/first bit of a number

LSB

the bit in a binary number which is of the lowest numerical value.
the rightmost/last bit of a number

Radix Sort

110, 011, 101, 001, 100, 111 , 010

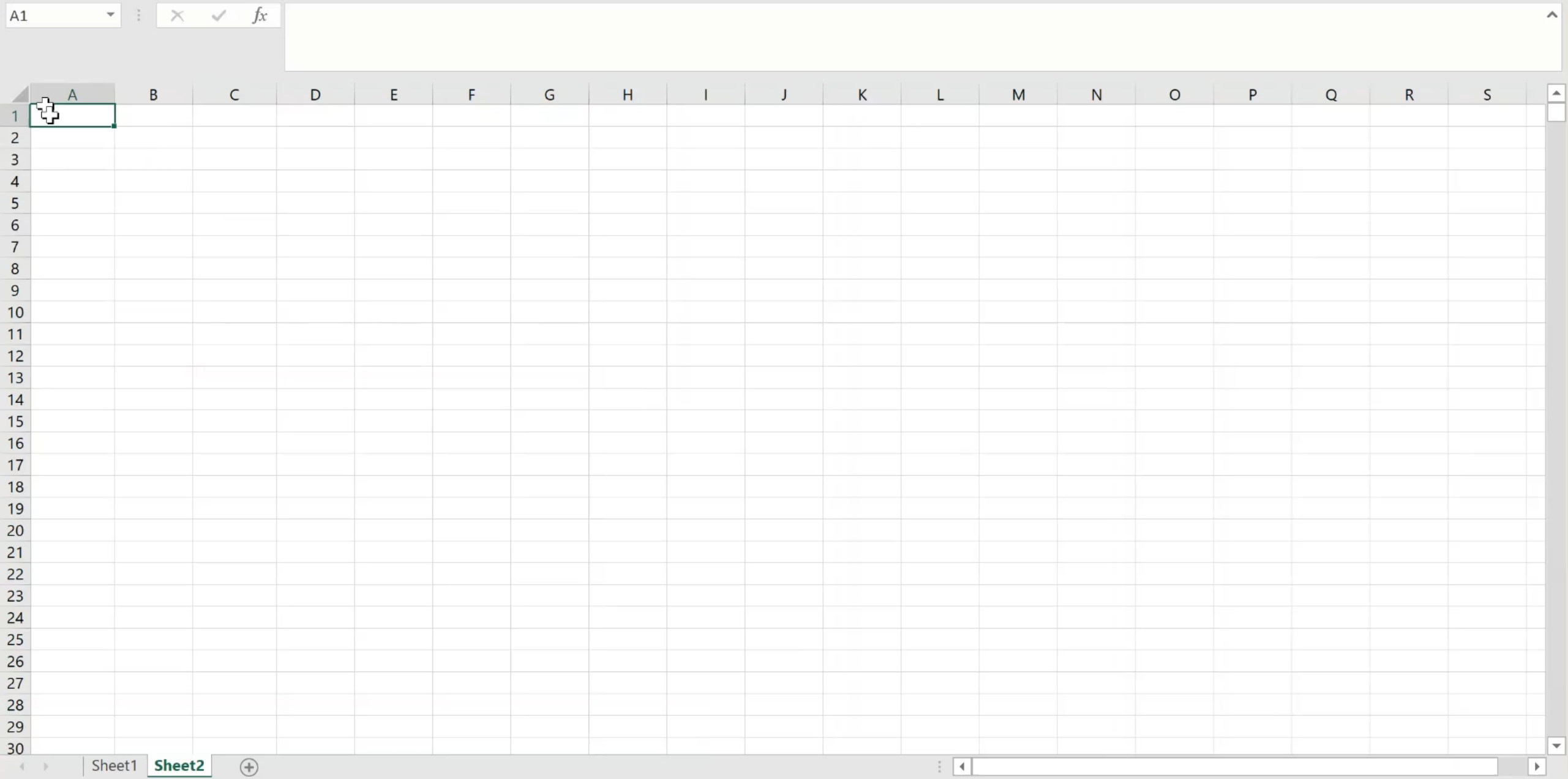
Let's arrange them starting with LSB and work towards the MSB.

110, 100, 010, 011, 101, 001, 111

100, 101, 001, 110, 010, 011, 111

001, 010, 011, 100, 101, 110, 111

Radix Sort


A1
fx
A B C D E F G H I J K L M N O P Q R S
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30
Sheet1
Sheet2

Radix Sort Time Complexity

So what is the time complexity of Radix Sort?

Let's establish a few terms

d is the number of digits in the numbers in the list – use the number of digits in the biggest number in the list.

n is the number of numbers in the list

b is the base/radix we are using

Radix Sort Time Complexity

For every number in our list, *n*, we checked each digit.

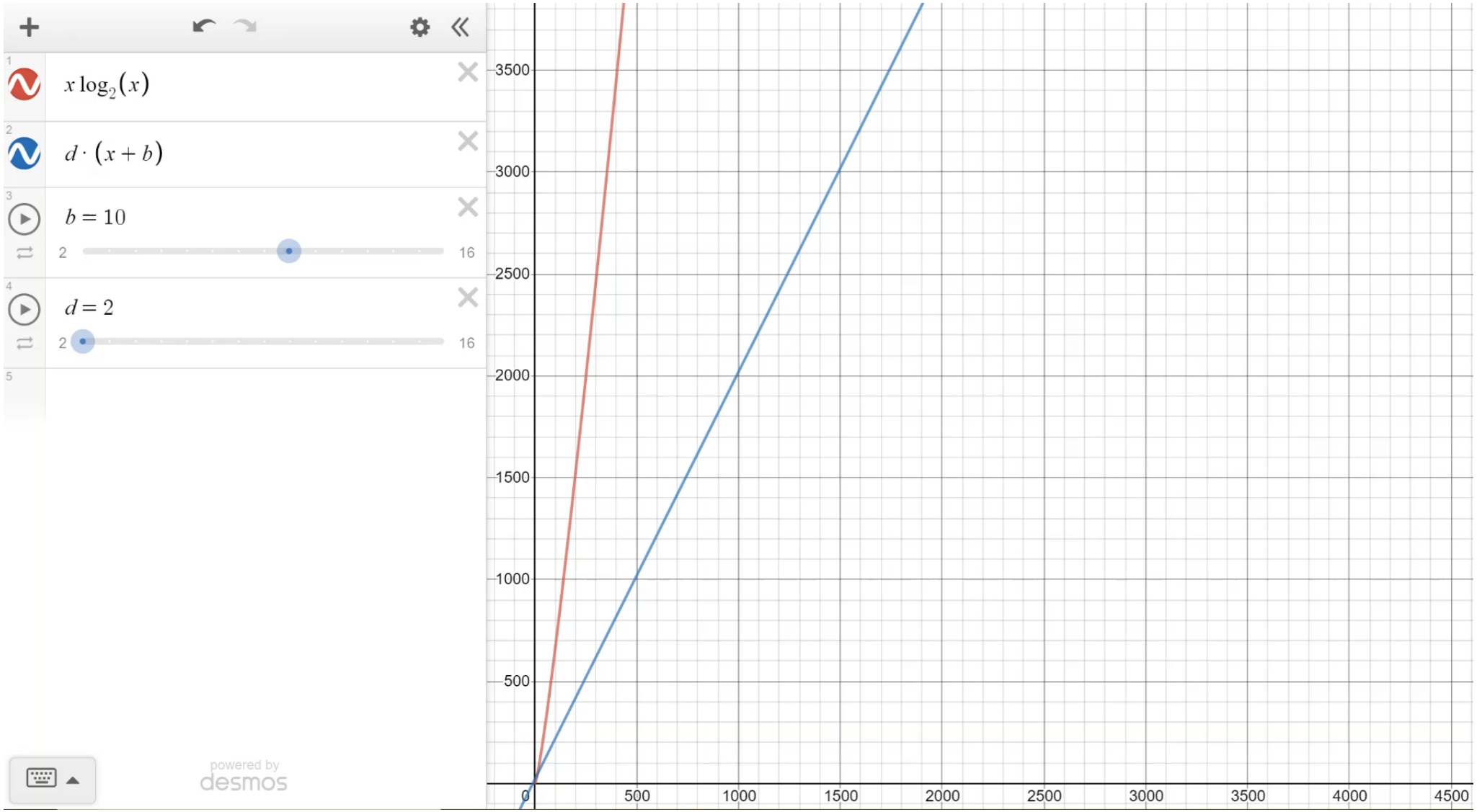
$d*n$

When we had a base 10 number, we had to check 10 digits -> 0-9.
When we had a base 2 number, we had to check 2 digits -> 0 and 1.

So the base influenced how much time we spent on each number.

$O(d * (n+b))$

Radix Sort Time Complexity



Radix Sort Time Complexity

$O(d * (n+b))$

As we saw in the graph, changing the base did not influence the run time very much.

You will also see the time complexity of radix sort written as

$O(nk)$

where k is the number of digits

Radix Sort

Interesting Thoughts on Radix Sort

Radix Sort does not use any comparisons or swaps.

The time complexity of Radix Sort maintains its coefficient (number of digits) because of the significant impact of that value on the time complexity.

$O(nk)$

Even asymptotic notation does not wave away the impact of the number of digits.

Radix Sort

So if Radix Sorts has a better run time than most other sorts, why isn't it used more?

The biggest limitation of Radix Sort is that it is limited to data that only consists of digits or letters.

Radix Sort also does not actually move any elements around in the storage container (like an array) – it makes a new version of the storage container; therefore, has a higher space complexity than some other "slower" sorts. O(n+k).