

# CSE 2320

Week of 08/03/2020

Instructor : Donna French

# Dynamic Programming

Let's look at a familiar recursive problem

Calculate the  $n$ th Fibonacci value.

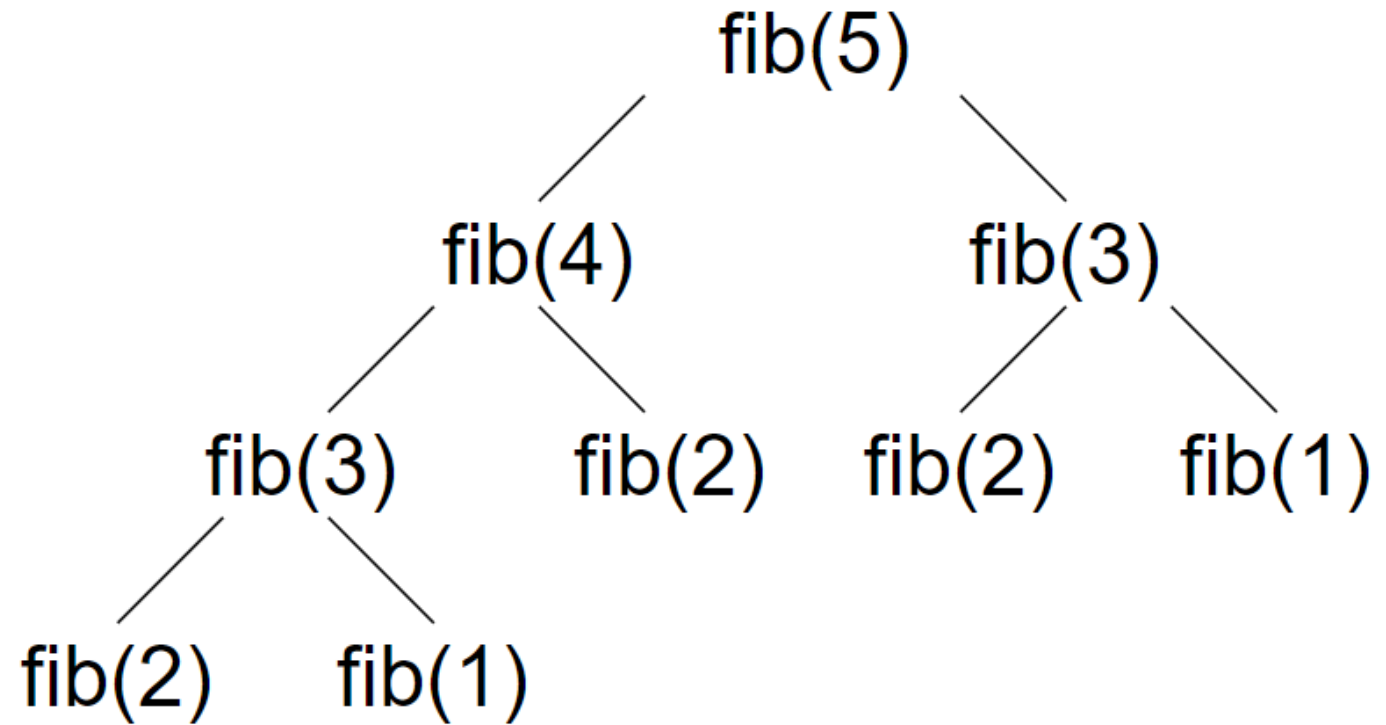
The recursive algorithm is

```
if n <= 2, then set f = 1
else f = fib(n-1) + fib(n-2)
return f
```

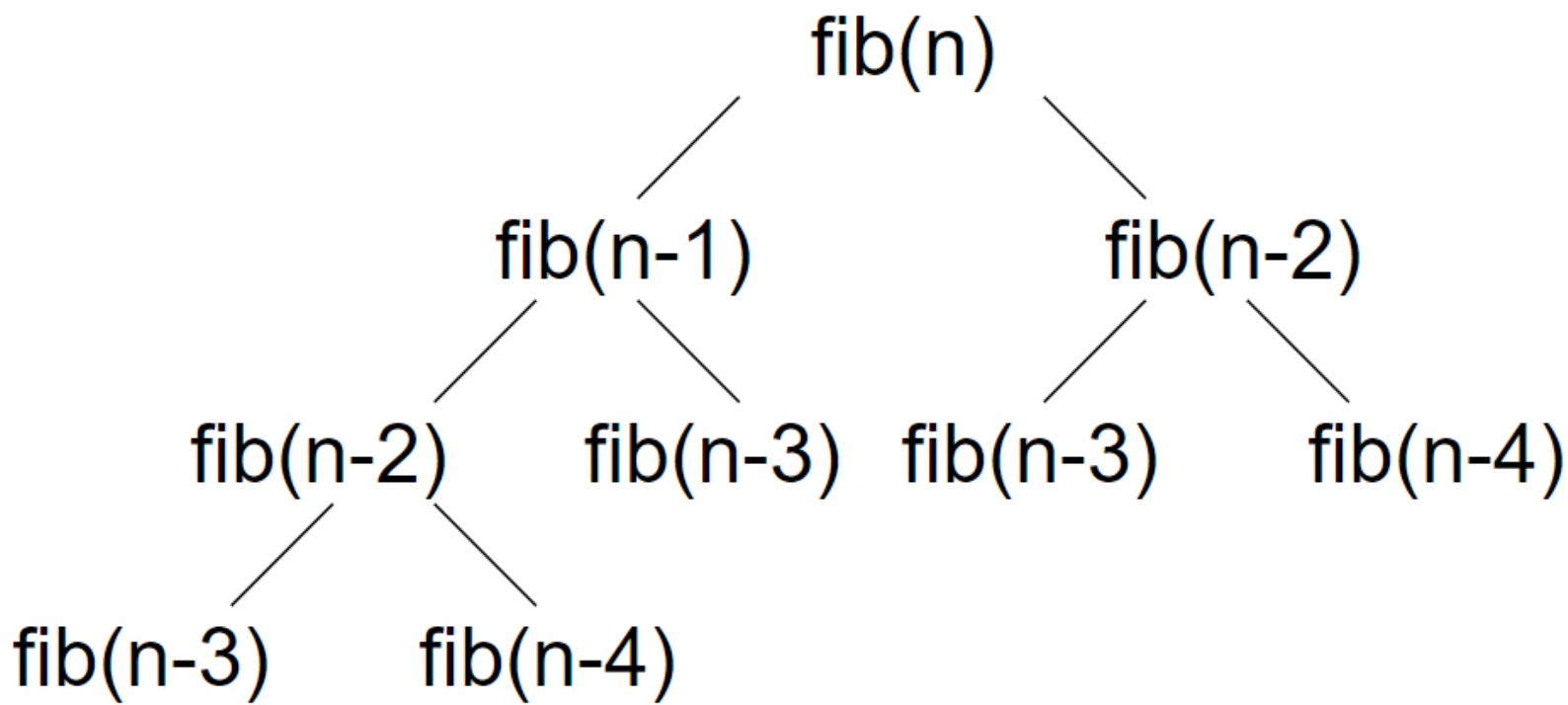
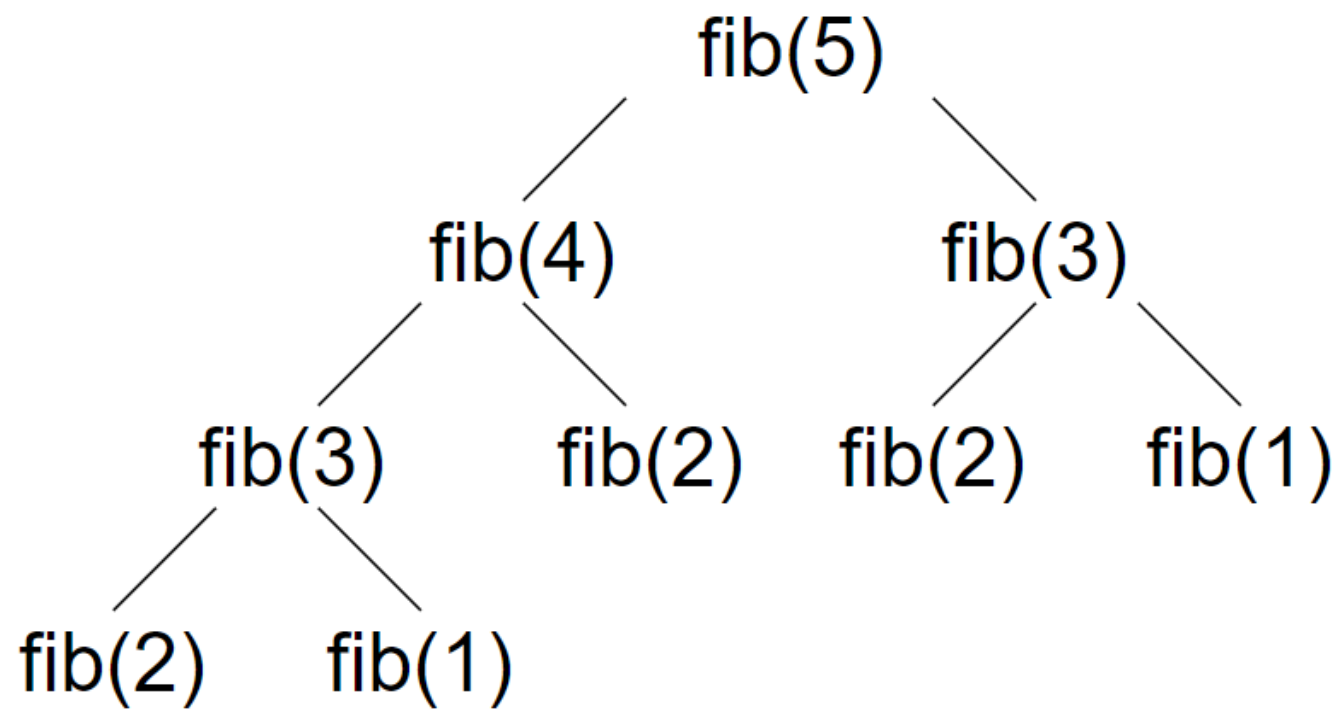
n	Fibonacci
0	0
1	1
2	1
3	2
4	3
5	5
6	8
7	13
8	21
9	34
10	55
11	89

# Dynamic Programming

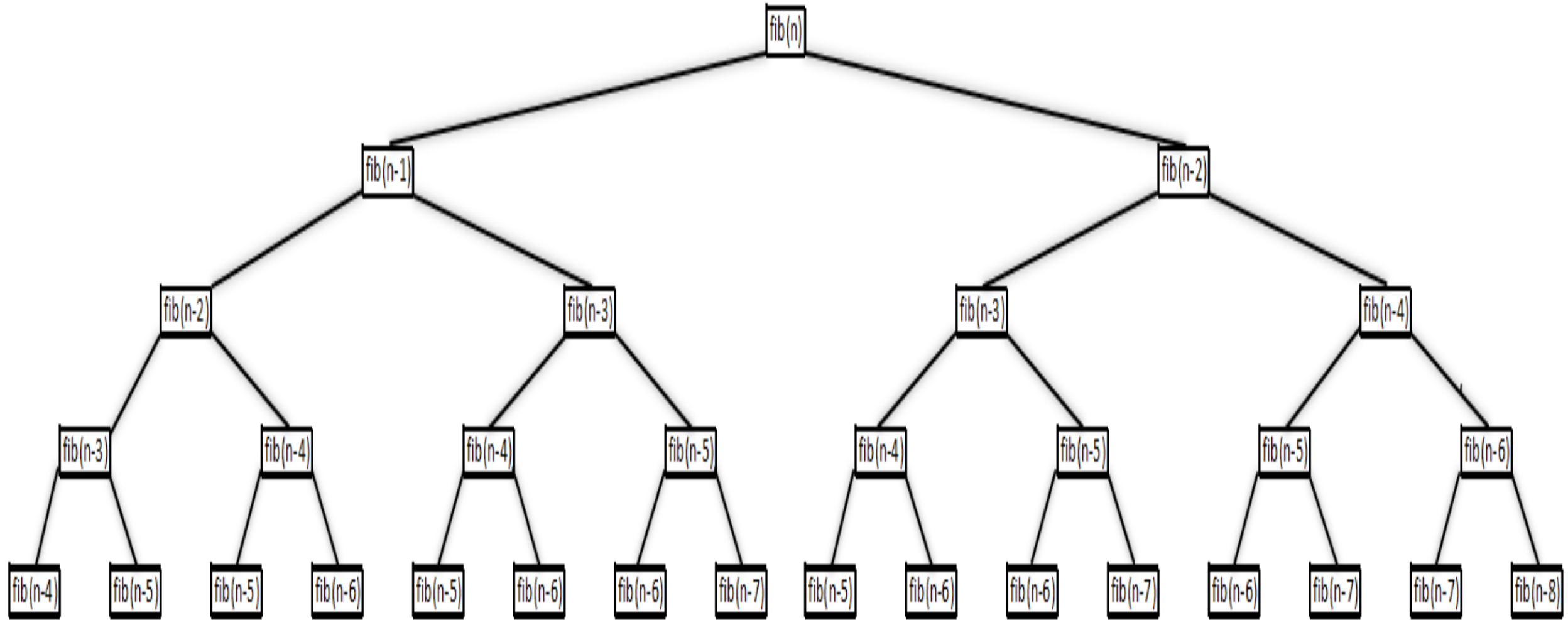
```
if n <= 2,  
then set f = 1  
else f = fib(n-1) + fib(n-2)  
  
return f
```



# Dynamic Programming



# Dynamic Programming



# Dynamic Programming

General, powerful algorithm design technique

Dynamic Programming is approximately "careful brute force"

Trying to get to polynomial time

Break a problem into subproblems, solve those subproblems and reuse those solutions.

# Dynamic Programming

Dynamic Programming (DP) is an algorithmic technique for solving an optimization problem by breaking it down into simpler subproblems and utilizing the fact that the optimal solution to the overall problem depends upon the optimal solution to its subproblems.

# Dynamic Programming

There are two patterns used in solving DP problems

1. Tabulation – Bottom up
2. Memoization – Top Down



Dynamic Programming

# MEMOIZATION

**MEMOIZATION?**

**MEMORIZATION?**



# Dynamic Programming

There are two patterns used in solving DP problems

1. Tabulation – Bottom up

I will study the theory of Dynamic Programming in CSE2320, then I will practice some problems using DP and hence I will master Dynamic Programming.

2. Memoization – Top Down

To master Dynamic Programming, I would have to practice DP problems and to practice problems, I would have to study some theory of Dynamic Programming from CSE2320.

# Dynamic Programming

## **Tabulation Method – Bottom Up Dynamic Programming**

As the name itself suggests starting from the bottom and cumulating answers to the top.

Let's apply this method to the Fibonacci problem.

What is the bottom/base case of Fibonacci?

# Dynamic Programming

## Tabulation Method – Bottom Up Dynamic Programming

What is the bottom/base case of Fibonacci?

Let's look at the classic definition of Fibonacci...

if  $n \leq 2$ ,  
then set  $f = 1$

else  $f = \text{fib}(n-1) + \text{fib}(n-2)$

return  $f$

So for  $n = 0$  or  $n = 1$  or  $n = 2$ ,  
1 is always returned

# Dynamic Programming

## Tabulation Method – Bottom Up Dynamic Programming

	n	Fibonacci
	0	0
	1	1
if (n == 0    n == 1    n == 2)	2	1
{	3	2
return 1;	4	3
}	5	5
	6	8
	7	13
	8	21
	9	34
	10	55
	11	89

# Dynamic Programming

## Tabulation Method – Bottom Up Dynamic Programming

```
if (n == 0 || n == 1)
{
    return n;
}

for (i = 2; i <= n; i++)
{

```

n	Fibonacci
0	0
1	1
2	1
3	2
4	3
5	5
6	8
7	13
8	21
9	34
10	55
11	89

# Dynamic Programming

## Tabulation Method – Bottom Up Dynamic Programming

```
if (n == 0 || n == 1)
{
    return n;
}

for (i = 2; i <= n; i++)
{
}
```

n	Fib	
0	0	
1	1	
2	1	0 + 1
3	2	1 + 1
4	3	1 + 2
5	5	2 + 3
6	8	3 + 5

# Dynamic Programming

## Tabulation Method – Bottom Up Dynamic Programming

```
if (n == 0 || n == 1)
{
    return n;
}

int First = 0, Second = 1;
for (i = 2; i <=n; i++)
{
    Fib = First + Second;
    First = Second;
    Second = Fib;
}

return Second;
```

n	Fib	
0	0	
1	1	
2	1	0 + 1
3	2	1 + 1
4	3	1 + 2
5	5	2 + 3
6	8	3 + 5



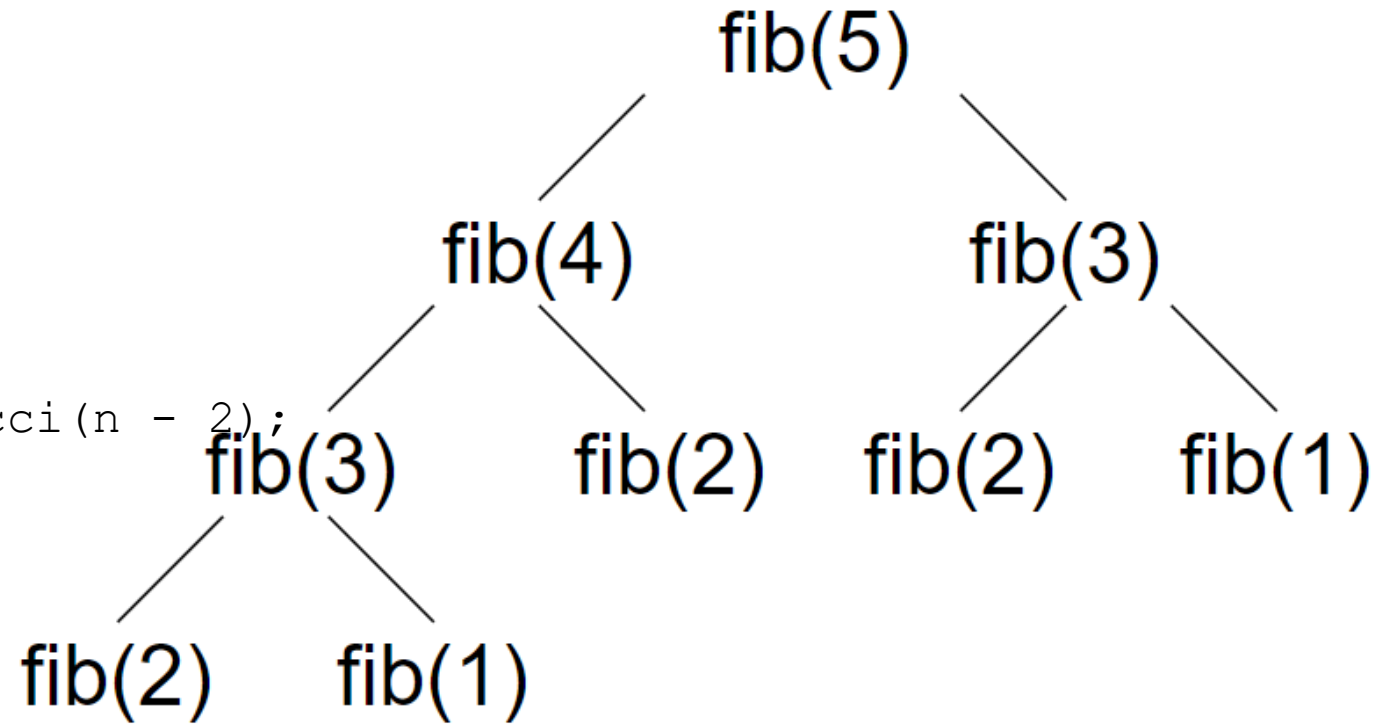
# Dynamic Programming

## Memoization

In computing, memoization or memoisation is an optimization technique used primarily to speed up computer programs by storing the results of expensive function calls and returning the stored result when the same inputs occur again.

# Dynamic Programming

```
unsigned long long int fibonacci(unsigned int n)
{
    if (n == 0 || n == 1)
    {
        return n;
    }
    else
    {
        return fibonacci(n - 1) + fibonacci(n - 2);
    }
}
```

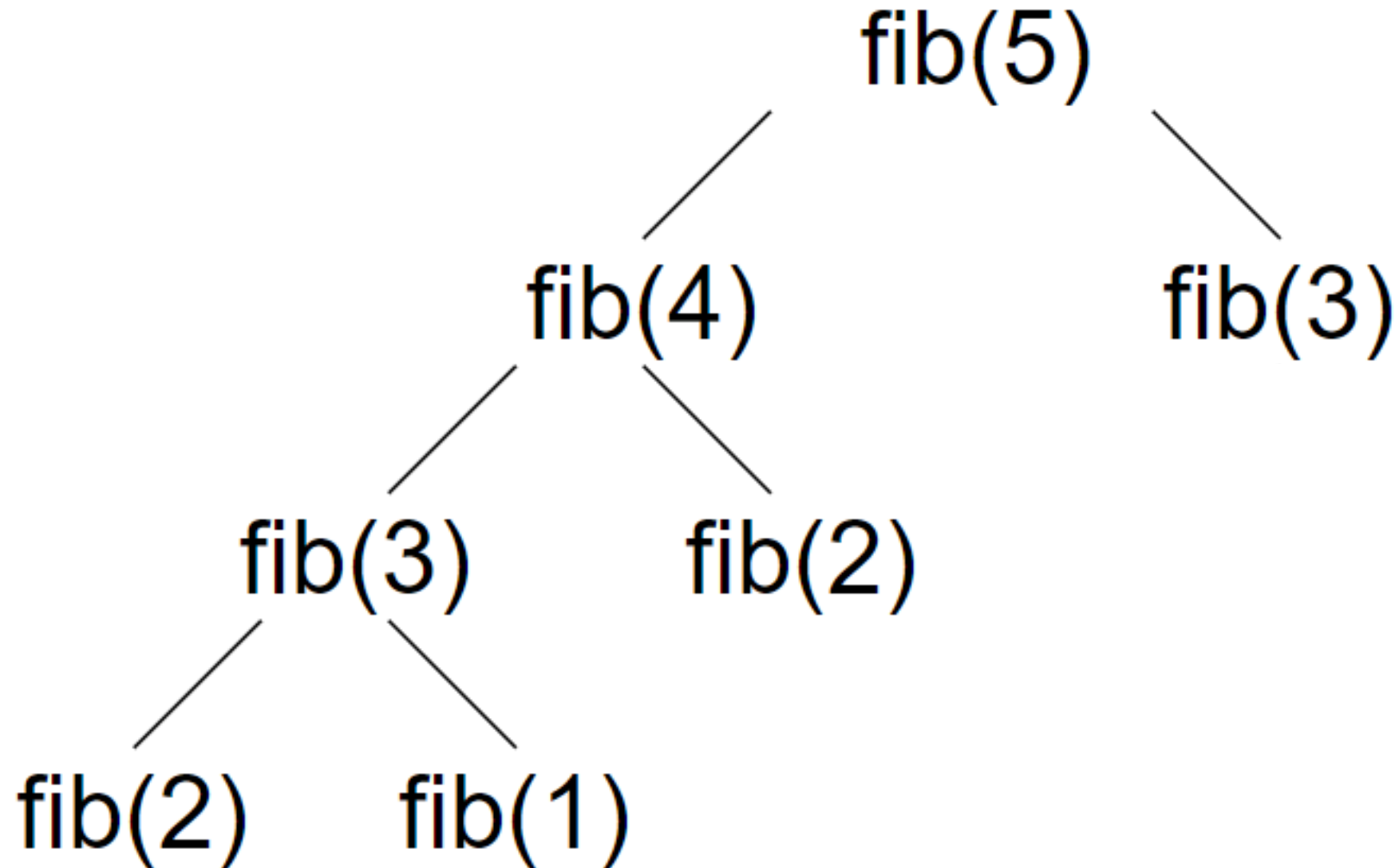


# Dynamic Programming

$\text{fib}(3) = 2$

$\text{fib}(4) = 3$

$\text{fib}(5) = 5$



# Dynamic Programming

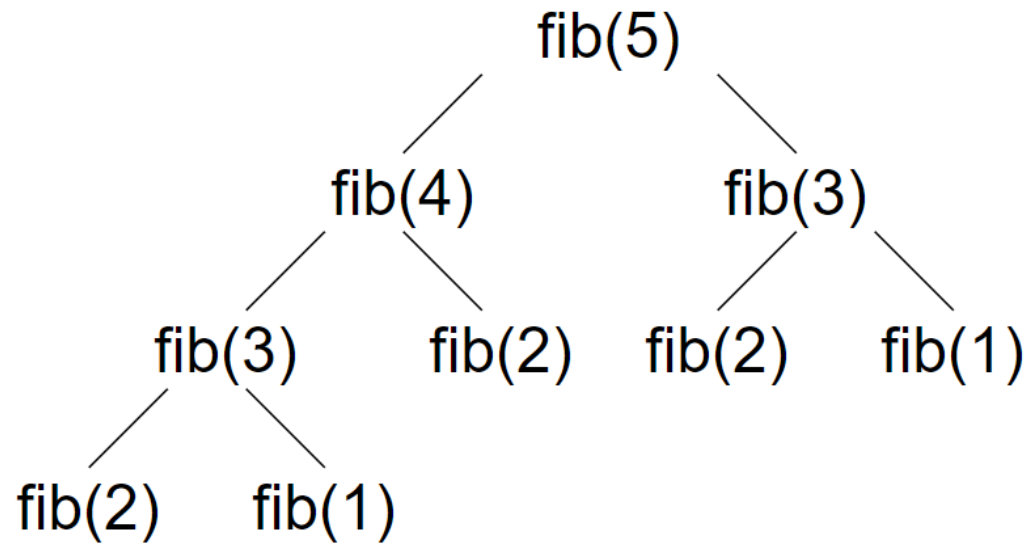
The memoized program for a problem is like the recursive version with a small modification that it looks into a lookup table before computing solutions.

We initialize a lookup array with all initial values as NIL.

Whenever we need the solution to a subproblem, we first look into the lookup table.

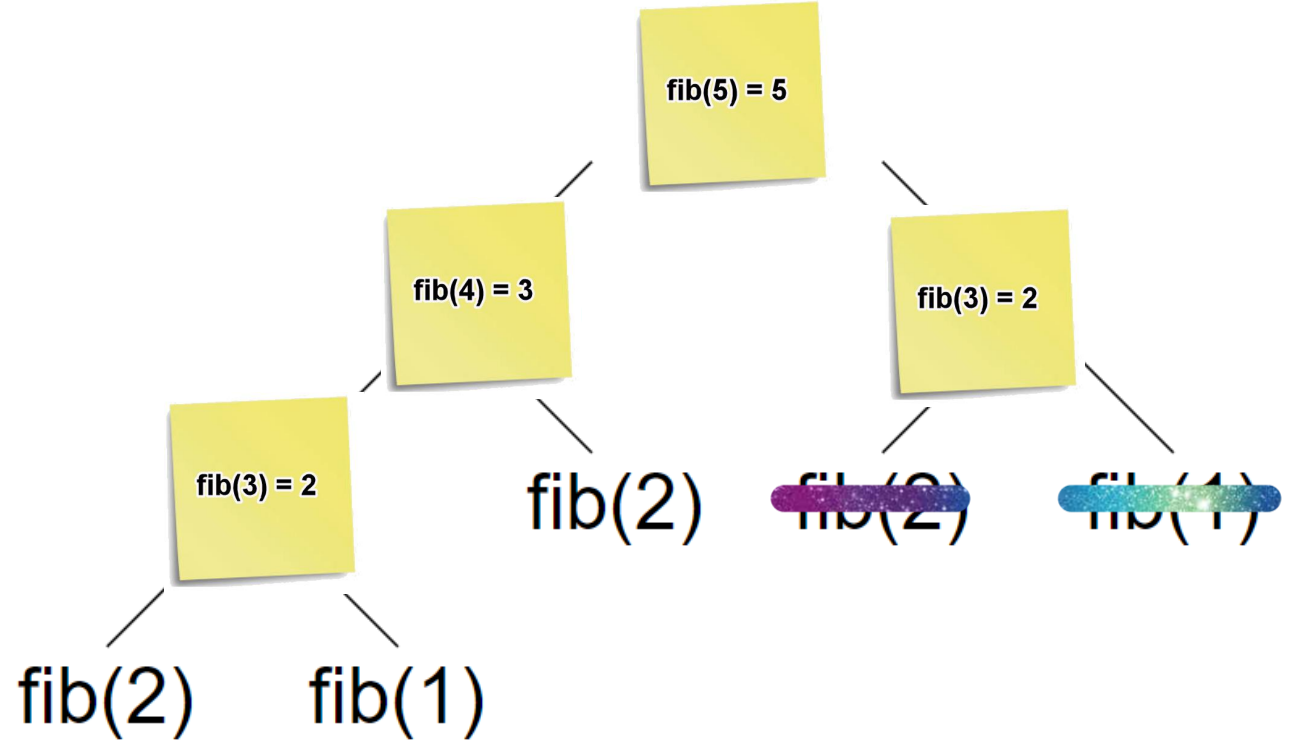
If the precomputed value is there then we return that value, otherwise, we calculate the value and put the result in the lookup table so that it can be reused later.

```
unsigned long long int fibonacci(unsigned int n)
{
    if (n == 0 || n == 1)
    {
        return n;
    }
    else
    {
        return fibonacci(n - 1) + fibonacci(n - 2);
    }
}
```



```
unsigned long long fibonacci(unsigned long long n, unsigned long long Memo[])  
{  
    unsigned long long result = 0;  
  
    if (n == 1 || n == 2)  
    {  
        result = 1;  
    }  
    else if (Memo[n] != '\0')  
    {  
        result = Memo[n];  
    }  
    else  
    {  
        result = fibonacci(n - 1, Memo) + fibonacci(n - 2, Memo);  
        Memo[n] = result;  
    }  
    return result;  
}
```

The diagram illustrates the recursive calls for calculating the 5th Fibonacci number, fib(5). The nodes are represented by yellow boxes containing the function name and its value. The root node is fib(5) = 5. It branches into fib(4) = 3 and fib(3) = 2. fib(4) = 3 branches into fib(3) = 2 and fib(2). The first fib(3) = 2 branches into fib(2) and fib(1). The second fib(3) = 2 branches into fib(2) and fib(1). The third fib(2) branches into fib(1) and fib(0). The fourth fib(2) branches into fib(1) and fib(0). The fifth fib(2) branches into fib(1) and fib(0). The sixth fib(2) branches into fib(1) and fib(0). The seventh fib(2) branches into fib(1) and fib(0).



# Dynamic Programming

There are two main properties of a problem that suggests it can be solved using Dynamic Programming.

1. Overlapping Subproblems
2. Optimal Substructure

# Dynamic Programming

## Overlapping Subproblems

Like Divide and Conquer, Dynamic Programming combines solutions to sub-problems.

Dynamic Programming is mainly used when solutions of same subproblems are needed again and again.



# Dynamic Programming

In dynamic programming, computed solutions to subproblems are stored/memoized so that these don't have to be recomputed.

Dynamic Programming is not useful when there are no common (overlapping) subproblems because there is no point storing the solutions if they are not needed again.

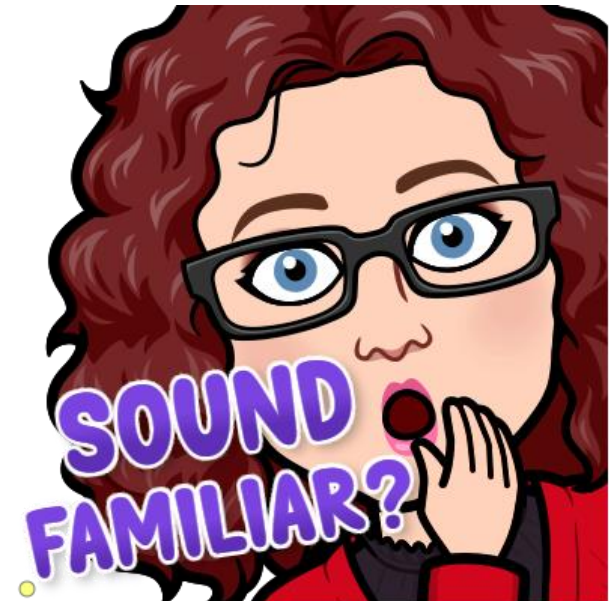
For example, Binary Search doesn't have common subproblems.

# Dynamic Programming

## Optimal Substructure

A given problem has Optimal Substructure Property if the optimal solution of the given problem can be obtained by using optimal solutions of its subproblems.

Greedy Algorithms have the same property.



# Dynamic Programming

## Optimal Substructure

A given problem has Optimal Substructure Property if the optimal solution of the given problem can be obtained by using optimal solutions of its subproblems.

DP techniques exploit this property to split the problems into smaller subproblems and solve them instead.

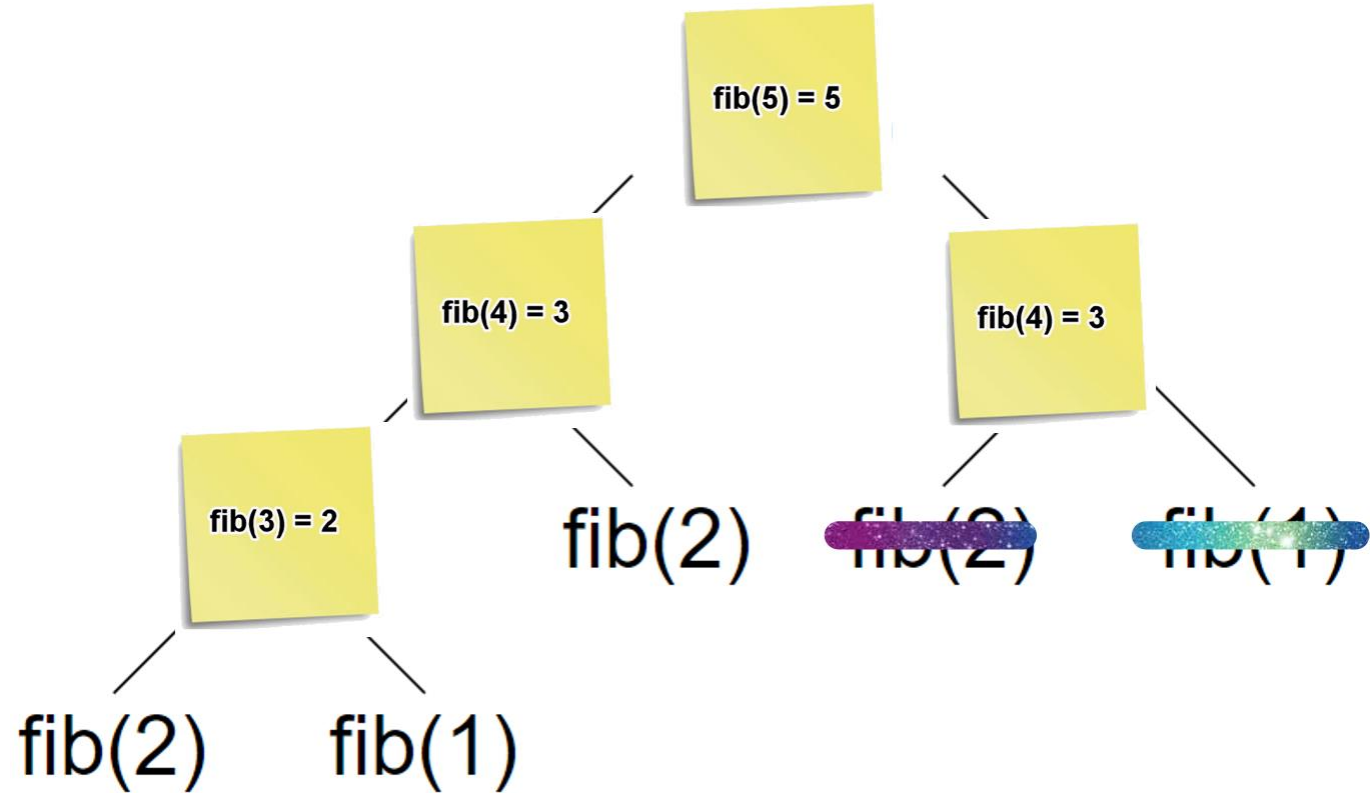
After the subproblems become sufficiently small, solving them becomes trivial.

# Dynamic Programming

DP  $\approx$  recursion + memoization

Memoize (remember) and re-use solutions to subproblems that help solve the problem.

For any given value  $k$ ,  $\text{fib}(k)$  only recurses the first time it is called.



# Dynamic Programming

For any given value  $k$ ,  $\text{fib}(k)$  only recurses the first time it is called.

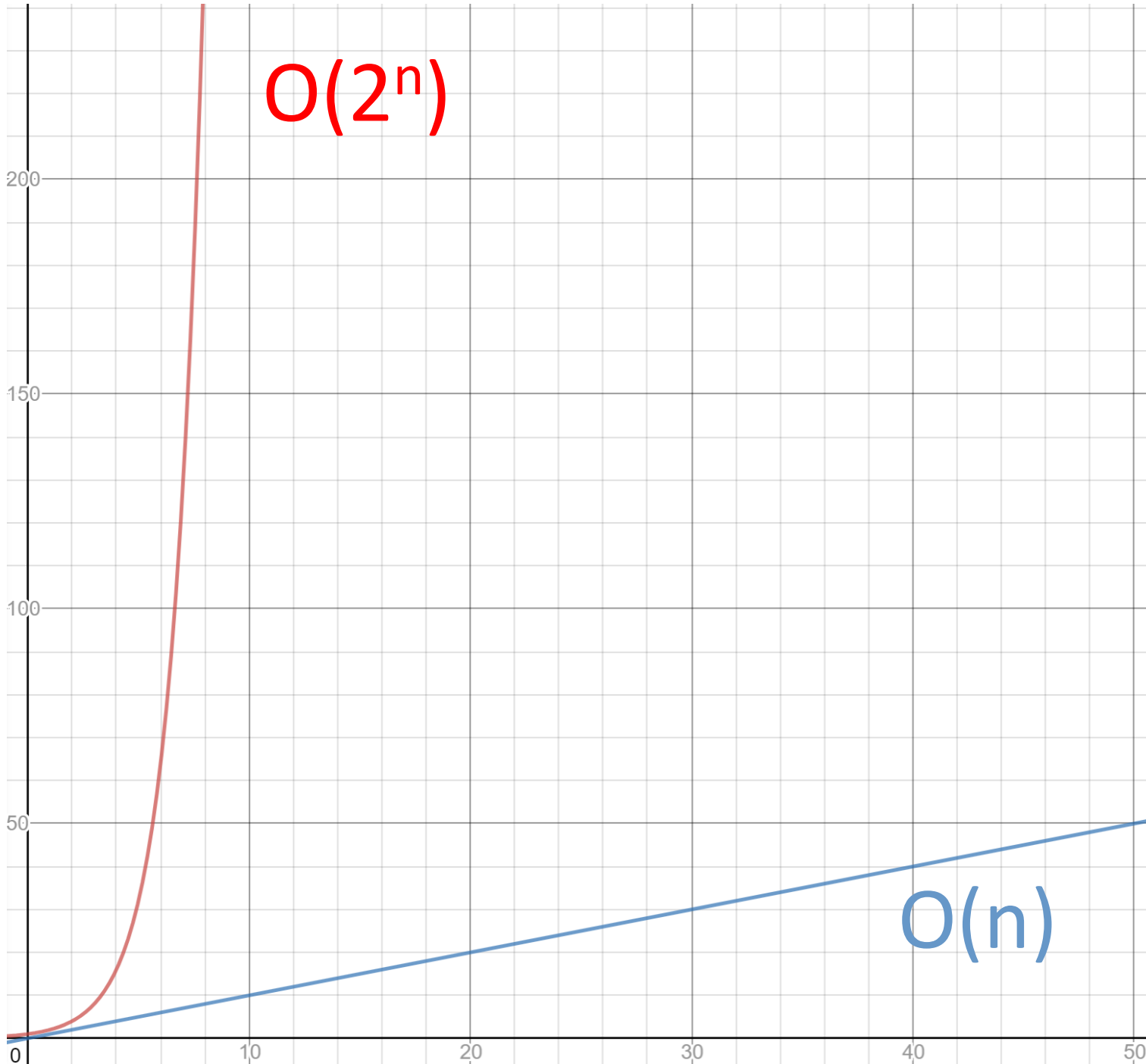
Memoized calls cost  $\Theta(1)$

The number of non-memorized calls is  $n$  given that we are calculating  $\text{fib}(n)$ .

The amount of nonrecursive work per call is  $\Theta(1)$ .

Total time of memoized version of Fibonacci is  $\Theta(n)$





1,125,899,906,842,624

$O(n)$

$O(2^n)$

# Hashing

If you need the definition of a word in the dictionary, how do you find it?

Do you start from page 1 and search until you find it?

$O(n)$

Do you divide the dictionary in half and decide if you need to look in the right half or left half and keep repeating that process until you find the word?

$O(\log_2 n)$

***I HOPE NOT!***





# Hashing

## Dictionary

Abstract Data Type - class of objects whose logical behavior is defined by a set of values and a set of operations

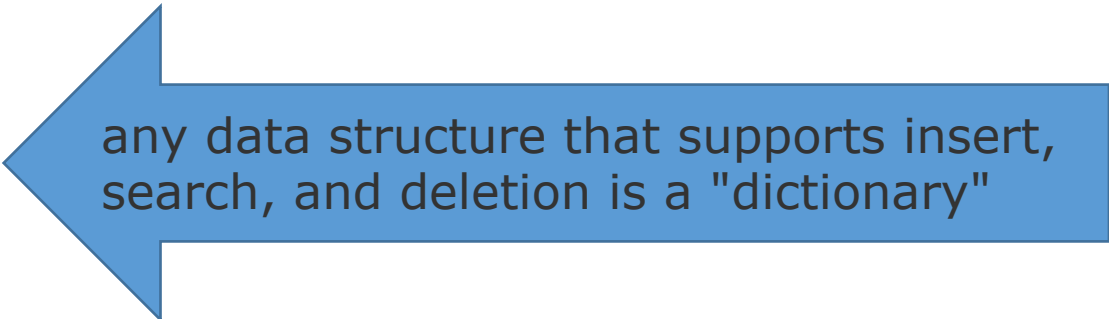
Maintains a set of items each with its own key

3 basic actions for a dictionary

Insert item

Delete item

Search for item



any data structure that supports insert, search, and deletion is a "dictionary"

# Hashing

## Dictionary

3 basic actions for a dictionary

- Insert item

  - Overwrite any existing key

- Delete item

- Search for item

  - Return the item with the given key or report that it does not exist

# Hashing

For those that have studied C++, you might be wondering...

`std::map`

is that a dictionary?

Yes!

`std::map` is the C++ standard library implementation of a dictionary

# Hashing

A **map** is a set where each element is a pair, called a key/value pair.

The key is used for sorting and indexing the data and must be unique.

The value is the actual data.

Duplicate keys are *not* allowed—a single value can be associated with each key.

This is called a one-to-one mapping.

# Hashing

A map of students where **id number** is the key and **name** is the value can be represented graphically as

Notice that keys are arranged in ascending order.

Maps always arrange its keys in sorted order.

Here the keys are of string type; therefore, they are sorted lexicographically.

1120217	Nikhilesh
1120236	Navneet
1120250	Vikas
1120255	Doodrah

Keys

values

# Hashing

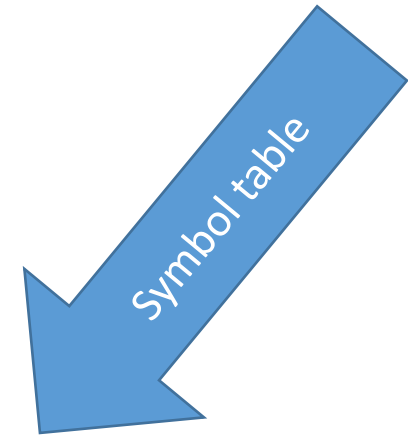
So what is a limitation of this implementation of a dictionary?



# Hashing

Dictionaries have a lot of uses

- Databases use them for lookups
- Spell check
- Username and password verification
- Compilers and interpreters (translate variable name to memory address)
- Network router
- Cryptography



# Hash Table

A **hash table** is a data structure which implements an associative array abstract data type which is a structure that can map keys to values.

In hashing, large keys are converted into small keys by using a **hash function**. The hash function employs an algorithm to compute an index into the array. These small keys are then stored in this array which is called the **hash table**.

The idea of hashing is to distribute entries uniformly across an array.



# Hash Table

Let's say we need to store student id's and their corresponding pin numbers.

1000012345	87345
1000023456	93727
1000046536	87323
1000085848	34522
1000100349	82234
1002002020	73721

# Hash Table

We could store that information in an array of structures.

0	1	2	3	4	5	6	7	8	9
1000012345 87345	1000023456 93727	1000046536 87323	1000085848 34522	1000100349 82234	1002002020 73721				

To find an item, we would perform a linear search – start at the beginning of the array and check each cell for a match.

Since this array has 10 cells, the performance would be based on the worst case which is the value we are looking for is in the last cell so  $O(10)$  or  $O(n)$  where  $n$  is the size of the array.

# Hash Table

Instead of storing each student ID/pin in an array, let's use a **Hash Function** to determine which array element is used.

Given an input key, the hash function uses an algorithm to map the input key to an index in the array.

```
int GetHashCode(unsigned int student_id)
{
    return ((student_id) % MaxHashTableSize);
}
```

# Hash Table

```
int GetHashCode(unsigned int student_id)
{
    return (student_id % MaxHashTableSize);
}
```

Input : 1000012345

Output :  $1000012345 \% 10 = 5$

So we put 1000012345+87345 in array element 5.

0	1	2	3	4	5	6	7	8	9
					1000012345 87345				

# Hash Table

```
int GetHashCode(unsigned int student_id)
{
    return (student_id % MaxHashTableSize);
}
```

Input : 1000023456

Output :  $1000023456 \% 10 = 6$

So we put 1000023456+93727 in array element 6.

0	1	2	3	4	5	6	7	8	9
					1000012345 87345	1000023456 93727			

# Hash Table

```
int GetHashCode(unsigned int student_id)
{
    return (student_id % MaxHashTableSize);
}
```

Input : 1000085848

Output :  $1000085848 \% 10 = 8$

So we put 1000085848+34522 in array element 8.

0	1	2	3	4	5	6	7	8	9
					1000012345 87345	1000023456 93727		1000085848 34522	

# Hash Table

```
int GetHashCode(unsigned int student_id)
{
    return (student_id % MaxHashTableSize);
}
```

Input : 1000100349

Output :  $1000100349 \% 10 = 9$

So we put 1000100349+82234 in array element 9.

0	1	2	3	4	5	6	7	8	9
					1000012345 87345	1000023456 93727		1000085848 34522	1000100349 82234

# Hash Table

```
int GetHashCode(unsigned int student_id)
{
    return (student_id % MaxHashTableSize);
}
```

Input : 1002002020

Output :  $1002002020 \% 10 = 0$

So we put 1002002020+73721 in array element 0.

0	1	2	3	4	5	6	7	8	9
1002002020 73721					1000012345 87345	1000023456 93727		1000085848 34522	1000100349 82234



# Hash Table

Now, when we want to search for a student ID, we give the student id to the hash function and it returns which array element to look in.

```
int GetHashCode(unsigned int student_id)
{
    return (student_id % MaxHashTableSize);
}
```

Input : 1000012345

Output :  $1000012345 \% 10 = 5$

Array Element 5 has student ID 1000012345 and pin 87345.

0	1	2	3	4	5	6	7	8	9
1002002020 73721					1000012345 87345	1000023456 93727		1000085848 34522	1000100349 82234

# Hash Table

Array with linear search –  $O(n)$  where  $n$  is the size of the array.

0	1	2	3	4	5	6	7	8	9
1000012345 87345	1000023456 93727	1000046536 87323	1000085848 34522	1000100349 82234	1002002020 73721				

## Hash Table

Hash function returns the exact array element containing information –  $O(1)$

0	1	2	3	4	5	6	7	8	9
1002002020 73721					1000012345 87345	1000023456 93727		1000085848 34522	1000100349 82234

# Hash Table – Hash Function

To achieve a good hashing mechanism, it is important to have a good hash function with the following basic requirements:

- Easy to compute - It should be easy to compute and must not become an algorithm in itself.
- Uniform distribution - It should provide a uniform distribution across the hash table and should not result in clustering.
- Minimize collisions: Collisions occur when pairs of elements are mapped to the same hash value. These should be avoided.

# Hash Table - Collisions

```
int GetHashCode(unsigned int student_id)
{
    return (student_id % MaxHashTableSize);
}
```

Input : 1000046536

Output :  $1000046536 \% 10 = 6$

So we would put 1000046536+87323 in array element 6 but another record is already there.

0	1	2	3	4	5	6	7	8	9
1002002020 73721					1000012345 87345	1000023456 93727		1000085848 34522	1000100349 82234



# Hash Table – Hash Function

Irrespective of how good a hash function is, **collisions** are bound to occur.

Therefore, to maintain the performance of a hash table, it is important to manage collisions through various collision resolution techniques.

There are multiple techniques for dealing with collisions

- Open Addressing

- Chaining

# Collision Resolution

## Open Addressing

In open addressing, all item are stored in the hash table itself.

Each array cell contains either a record or NULL.

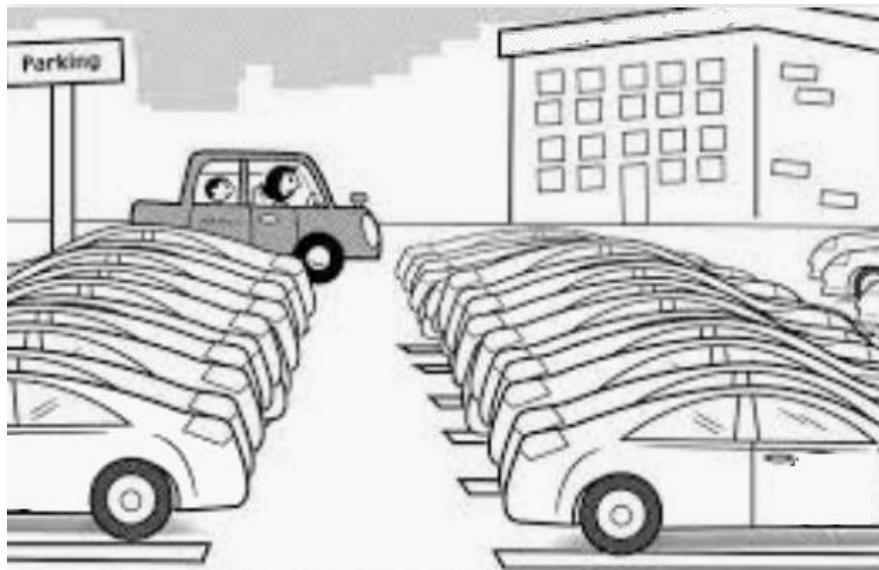
When inserting an item, if the array cell is already occupied, then open addressing takes the next cell.

# Collision Resolution

## Open Addressing

It keeps moving to the next cell until it finds an empty one.

If it reaches the end of the array, it starts over at the beginning and keeps looking

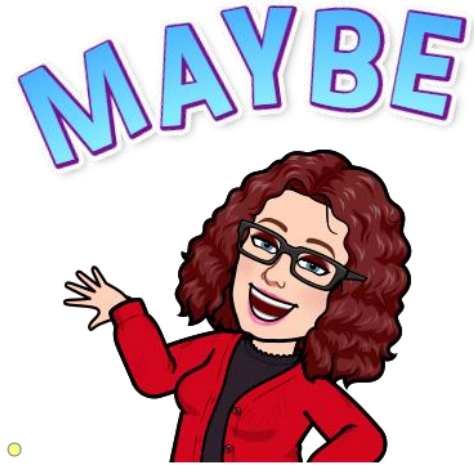






# Collision Resolution

## Open Addressing



This method either requires enough array space for every item to be stored or it needs to be able to reallocate the array in order to expand.

When searching for an element, the hash function takes the search to the array cell indicated by the hash, but then must perform a linear search to ensure that the correct item was found.

The array cells of the hash table are examined one by one until the desired item is found or the end of the array is found (meaning the item is not present).

# Collision Resolution

## Open Addressing



### Linear Probing

In linear probing, we linearly probe for next open cell.

The main problem with linear probing is clustering, many consecutive elements form groups and it starts taking time to find a free slot or to search an element.

# Collision Resolution

## Open Addressing

### Quadratic Probing

Takes the original hash index and adds successive values of an arbitrary quadratic polynomial until an open cell is found.

### Double Hashing

Perform a second hash to find a new cell

# Collision Resolution

## Open Addressing

Linear probing is easy to compute and has the best cache performance but suffers from clustering.

Quadratic probing lies between the two in terms of cache performance and clustering.

Double hashing has poor cache performance but no clustering. Double hashing requires more computation time as two hash functions need to be computed.

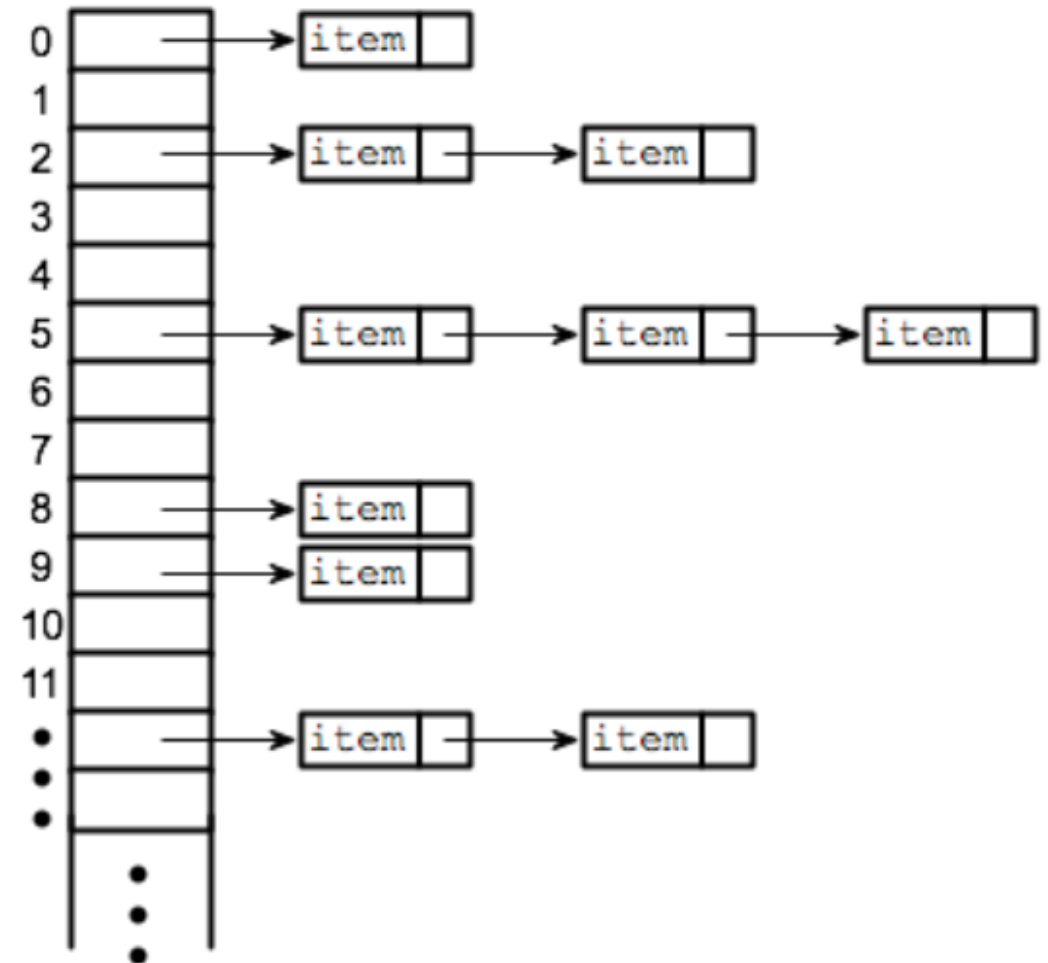
# Collision Resolution

## *Separate chaining (open hashing)*

Separate chaining is one of the most commonly used collision resolution techniques.

It is usually implemented using linked lists.

In separate chaining, each element of the hash table is a linked list.

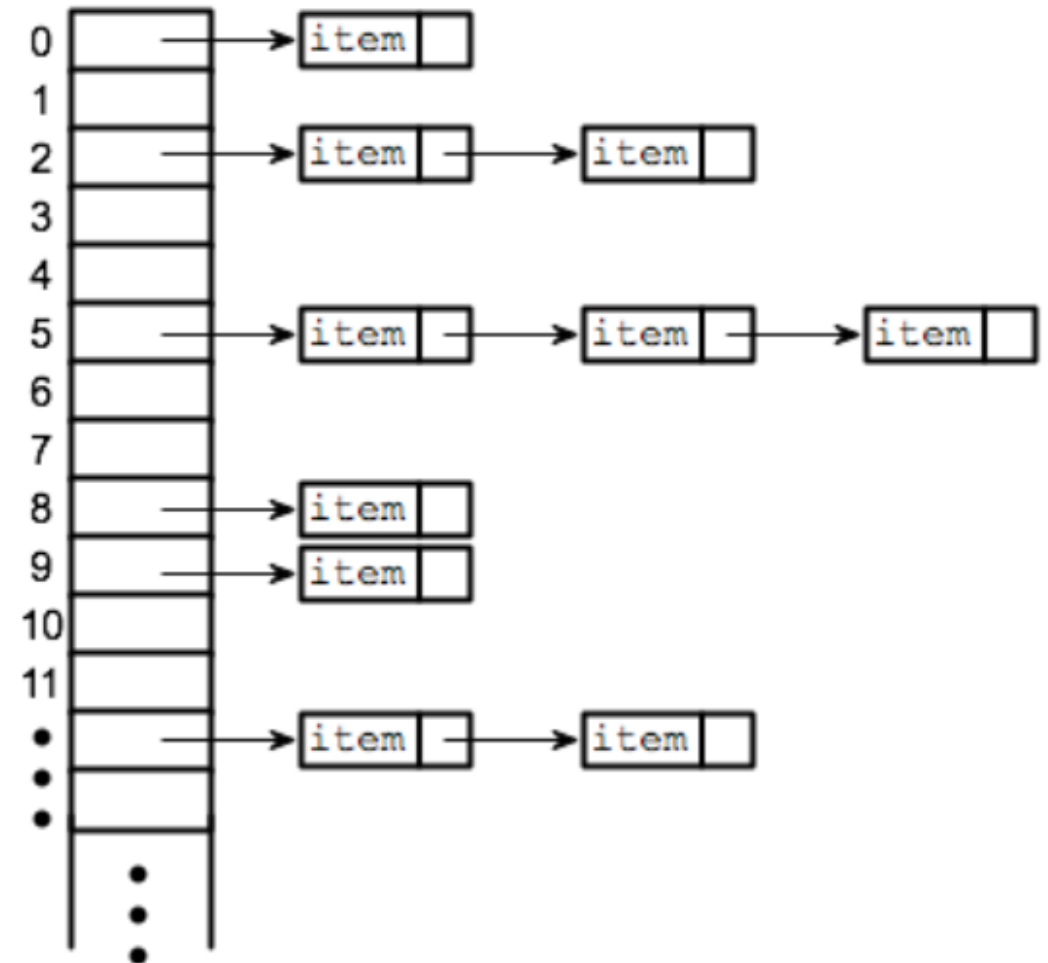


# Collision Resolution

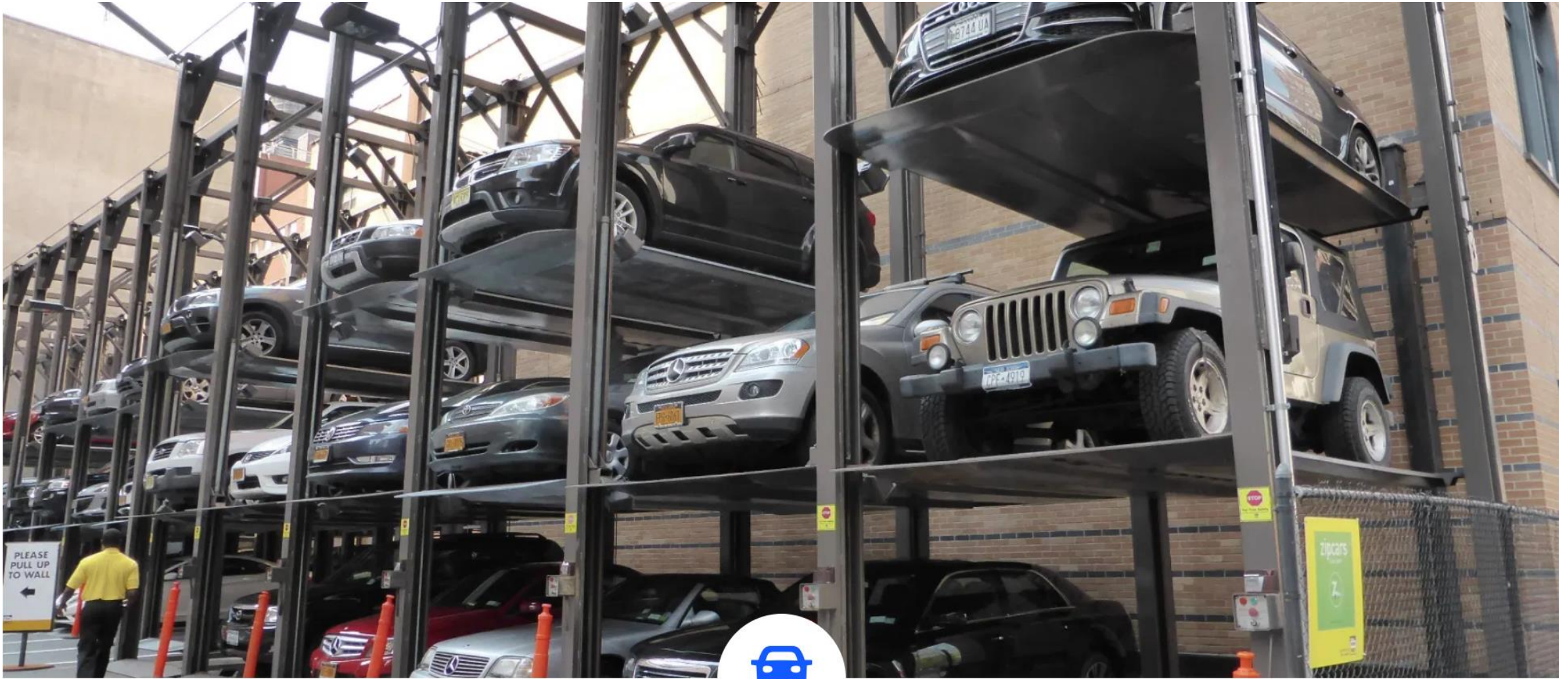
## *Separate chaining (open hashing)*

To store an element in the hash table you must insert it into a specific linked list.

If there is any collision (i.e. two different elements have same hash value) then store both elements in the same linked list.

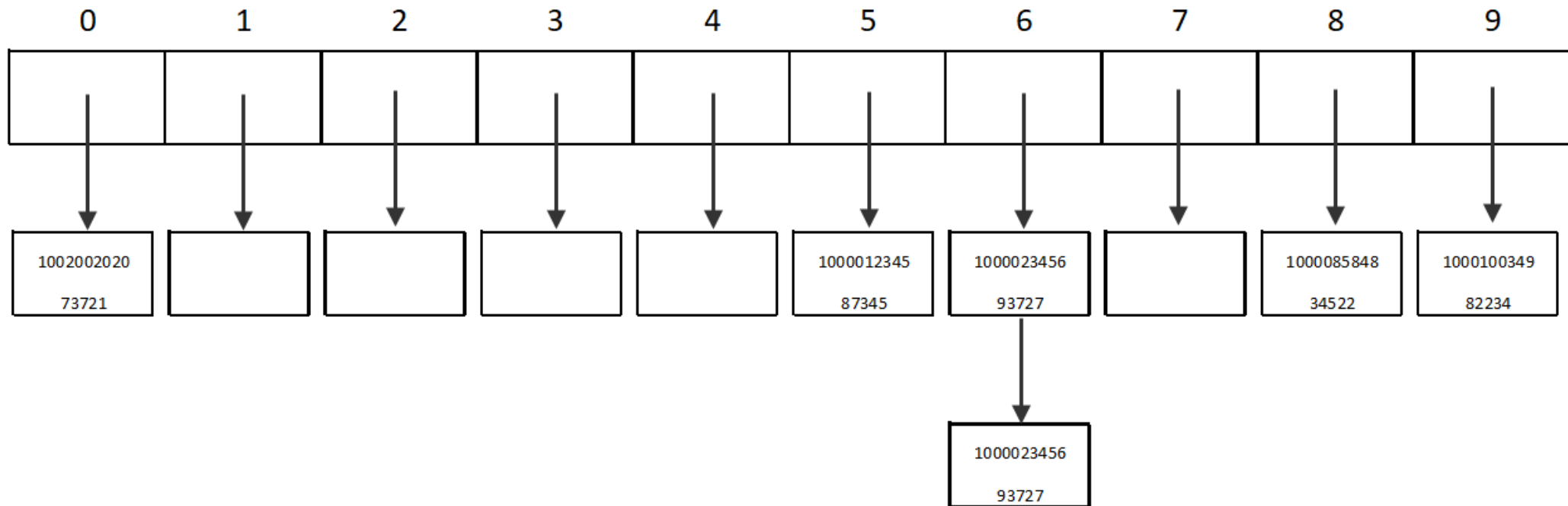






# Collision Resolution

Separate chaining turns each array element into its own linked list. Any collisions are handled by adding the new element on to the end of the linked list for that index.





# Collision Resolution

For separate chaining, the worst-case scenario is when all the entries are inserted into the same linked list. The lookup procedure may have to scan all its entries, so the worst-case cost is proportional to the number ( $n$ ) of entries in the table so  $O(n)$  which is the same as linear search.

A good hash function can prevent this worst-case scenario and make hash table look up always faster than linear search.

Hash tables, on average, are more efficient than search trees or any other table lookup structure. You can access an element in  **$O(1)$**  time. This is why hash tables are heavily used in many situations.

# Hashing

The capacity of the hash table is the size of the array.

The load factor is the number of keys stored in the hash table divided by the capacity.

The size should be chosen so that the load factor is less than 1.

For instance, if we want to implement a German-English dictionary with 50,000 German words, we need a hash table that is larger than 50,000.

# Hashing

Since the number of keys in the hash table is less than the capacity of the hash table, assuming that the keys are evenly distributed across indices, there will be few collisions, and most of the linked lists will be of length 1.

A few will be of length 2; a very few will be of length 3, and so on.

The probability that there is any linked list that is very much longer than the load factor is very small.

Separate Chaining	Open Addressing
Chaining is simpler to implement.	Open Addressing requires more computation.
Does not run out of room.	Array may fill up
Less sensitive to the hash function and load factor.	Requires extra care to avoid clustering and the load factor.
Mostly used when it is unknown how many and how frequently keys may be inserted or deleted.	Used when the frequency and number of keys is known.
Cache performance is not good as keys are stored using linked list.	Provides better cache performance as everything is stored in the same table.
Waste of space - some parts of hash table are never used	A cell can be used even if an input doesn't map to it.
Uses extra space for linked list pointers.	No links – everything in table

# Hashing

Jan	Tim	Mia	Sam	Leo	Ted	Bea	Lou	Ada	Max	Zoe
<b>0</b>	<b>1</b>	<b>2</b>	<b>3</b>	<b>4</b>	<b>5</b>	<b>6</b>	<b>7</b>	<b>8</b>	<b>9</b>	<b>10</b>

If we had this array of names and we wanted to look up name, how would we find it?

We would have to start at the first element and search.

What is the time complexity of a linear search?

$O(n)$

What if we sorted and used a binary search?

$O(\log_2 n)$

# Hashing

Jan	Tim	Mia	Sam	Leo	Ted	Bea	Lou	Ada	Max	Zoe
<b>0</b>	<b>1</b>	<b>2</b>	<b>3</b>	<b>4</b>	<b>5</b>	<b>6</b>	<b>7</b>	<b>8</b>	<b>9</b>	<b>10</b>

How about we use hashing?

$$\text{Jan} = J (74) + a (97) + n (110) = 281$$

What can we use for a hashing function?

Now MOD by size of array

$$281 \% 11 = 6$$

How about ASCII values?

# Hashing

Bea	Tim					Jan				
0	1	2	3	4	5	6	7	8	9	10

$$\text{Tim} = T(84) + i(105) + m(109) = 298 \quad \text{Bea} = B(66) + e(101) + a(97) = 264$$

Now MOD by size of array

$$298 \% 11 = 1$$

Now MOD by size of array

$$264 \% 11 = 0$$

# Hashing

If we apply the hash function to the rest of the names, our hash table will look like...

Bea	Tim	Leo	Sam	Mia	Zoe	Jan	Lou	Max	Ada	Ted
<b>0</b>	<b>1</b>	<b>2</b>	<b>3</b>	<b>4</b>	<b>5</b>	<b>6</b>	<b>7</b>	<b>8</b>	<b>9</b>	<b>10</b>

Now, how do we look up a name?



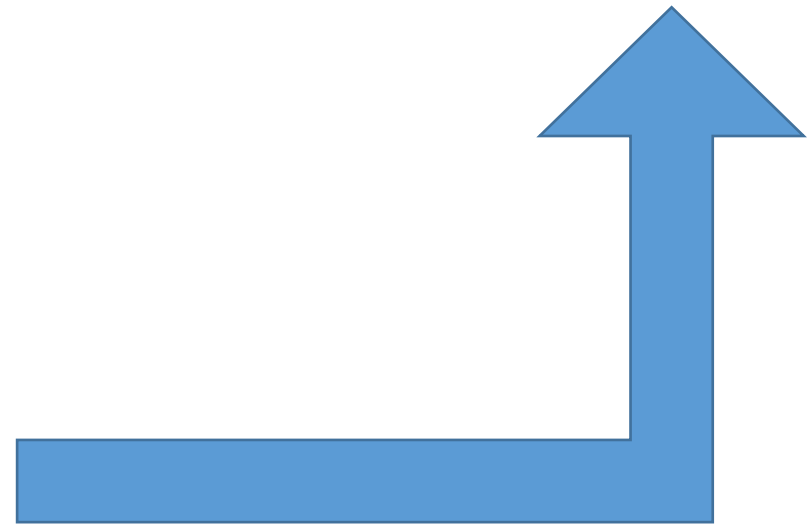
# Hashing

Bea	Tim	Leo	Sam	Mia	Zoe	Jan	Lou	Max	Ada	Ted
0	1	2	3	4	5	6	7	8	9	10

Let's say we want to find Ada.

We apply the hash function

$$\text{Ada} = A(65) + d(100) + a(97) = 262 \% 11 = 9$$



Time complexity?

$\Theta(1)$

# Hash Functions/Algorithms

## Numeric keys

Divide the key by the number of cells in the array and take the remainder (use mod)

## Alphanumeric keys

Divide the sum of the ASCII values by the number of cells in the array and take the remainder (use mod)

## Folding Method

Divide the key into equal parts and add the parts together and then mod

Phone number – 2147722387

$214 \Rightarrow 7$  and  $772 \Rightarrow 16$  and  $2387 = 20$

$7+16+20 = 43$

$43 \bmod \text{\#of cells in array}$

# Hashing with Open Addressing

Let's look at how Open Addressing handles collisions.

Bea	Tim			Mia	Zoe					
0	1	2	3	4	5	6	7	8	9	10

What happens when we try to add Sue?

$$S(83) + u(117) + e(101) = 301 \% 11 = 4$$

Mia is already in cell 4. So what does Open Addressing do with this collision?

# Hashing with Open Addressing

Open Addressing with linear probing would add Sue at cell 6.

Bea	Tim			Mia	Zoe	Sue				
0	1	2	3	4	5	6	7	8	9	10

What happens when we try to add Len?

$$L(76) + e(101) + n(110) = 287 \% 11 = 1$$

Tim is already in cell 1. So what does Open Addressing do with this collision?

# Hashing with Open Addressing

Open Addressing with linear probing would add Len at cell 2.

Bea	Tim	Len	Moe	Mia	Zoe	Sue	Lou			
0	1	2	3	4	5	6	7	8	9	10

Moe would hash to cell 3 so no issue.

Lou would hash to cell 7 so no issue.

Rae would hash to cell 5 but Zoe is already there.

What happens with Open Addressing?

# Hashing with Open Addressing

Open Addressing with linear probing would add Rae at cell 8.

Bea	Tim	Len	Moe	Mia	Zoe	Sue	Lou	Rae	Max	Tod
<b>0</b>	<b>1</b>	<b>2</b>	<b>3</b>	<b>4</b>	<b>5</b>	<b>6</b>	<b>7</b>	<b>8</b>	<b>9</b>	<b>10</b>

Max would hash to cell 8 which is already occupied by Rae.


Open Addressing with linear probing would add Max at cell 9.

Tod would hash to cell 9 which is already occupied by Max.

Open Addressing with linear probing would add Tod at cell 10.

# Hashing with Open Addressing

Bea	Tim	Len	Moe	Mia	Zoe	Sue	Lou	Rae	Max	Tod
0	1	2	3	4	5	6	7	8	9	10



So how do we look up Rae?

We hash it

$$R(82) + a(97) + e(101) = 280 \% 11 = 5$$

So we would look for Rae at cell 5.

# Hashing with Separate Chaining

Let's do the same example but with Separate Chaining instead of Open Addressing.

Instead of storing the names in the actual array cells like with Open Addressing....

Bea	Tim			Mia	Zoe					
0	1	2	3	4	5	6	7	8	9	10

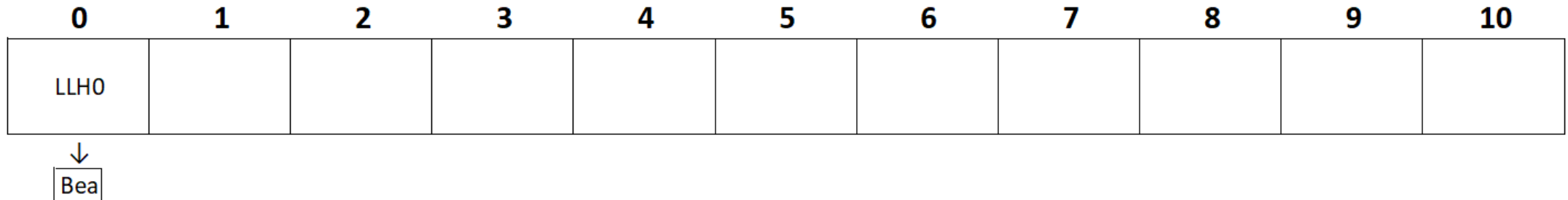


# Hashing with Separate Chaining

When a key hashes to an array cell, a linked list is created.

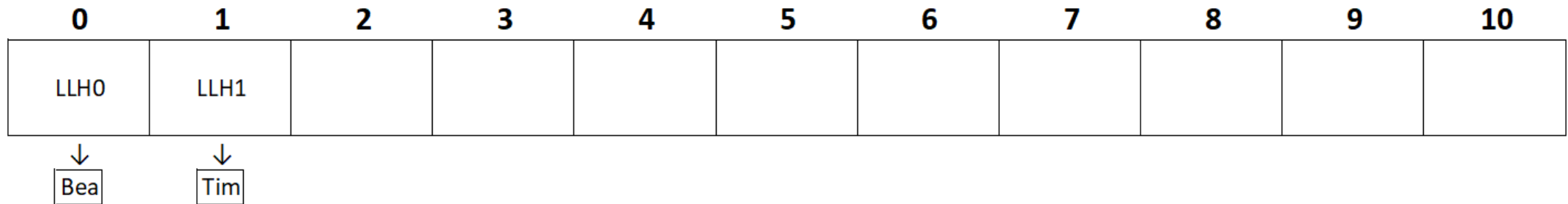
The linked list head is stored in the array – the array is now a `char` pointer array rather than just a `char` array.

Bea hashes to cell 0.

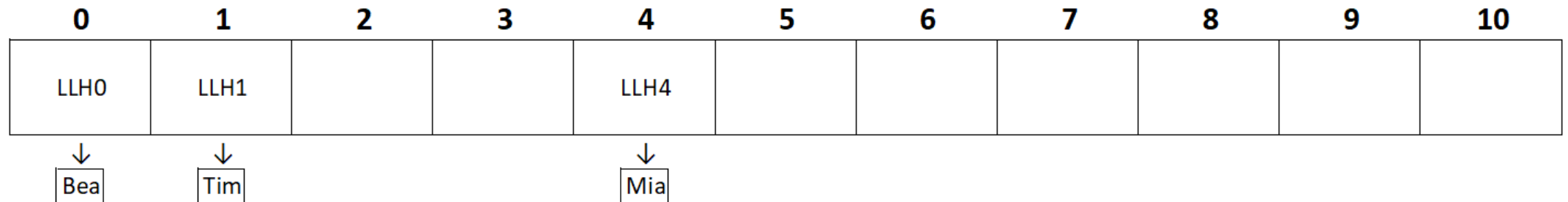


# Hashing with Separate Chaining

Tim hashes to cell 1.

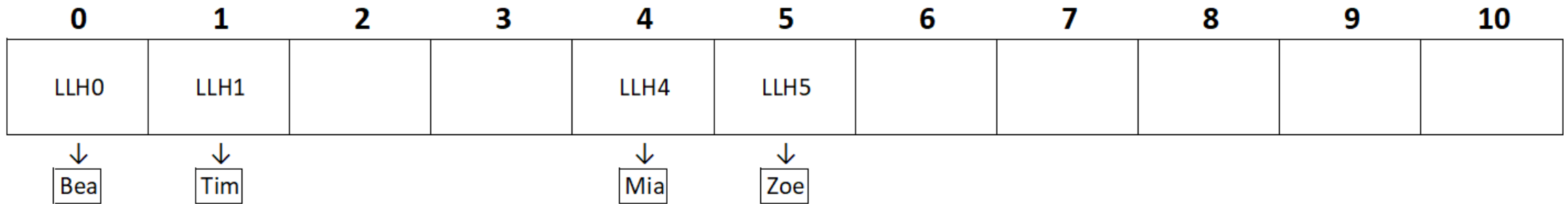


Mia hashes to cell 4.

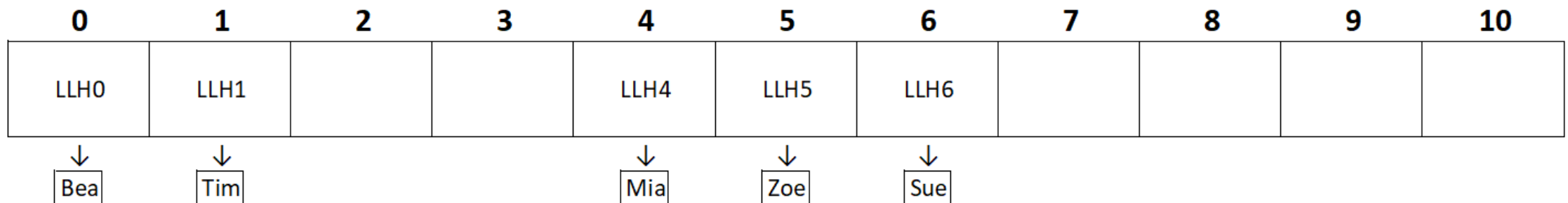


# Hashing with Separate Chaining

Zoe hashes to cell 5.

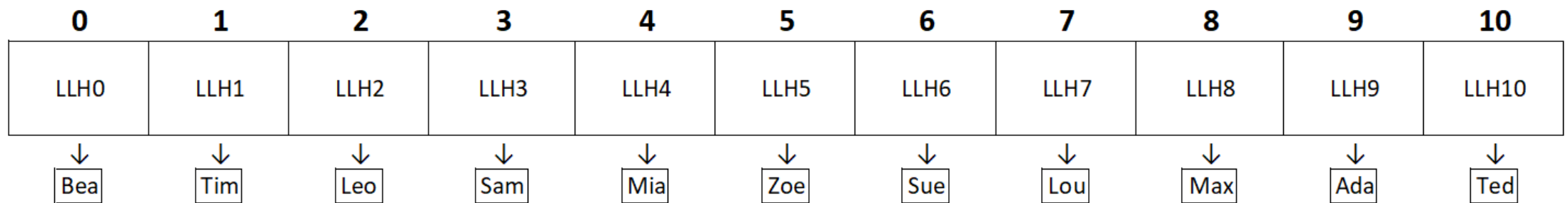


Sue hashes to cell 6.

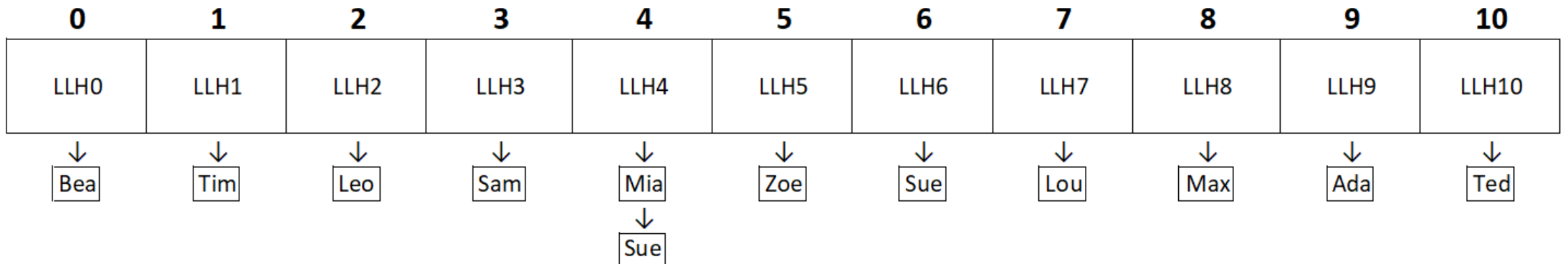


# Hashing with Separate Chaining

We can add the rest...

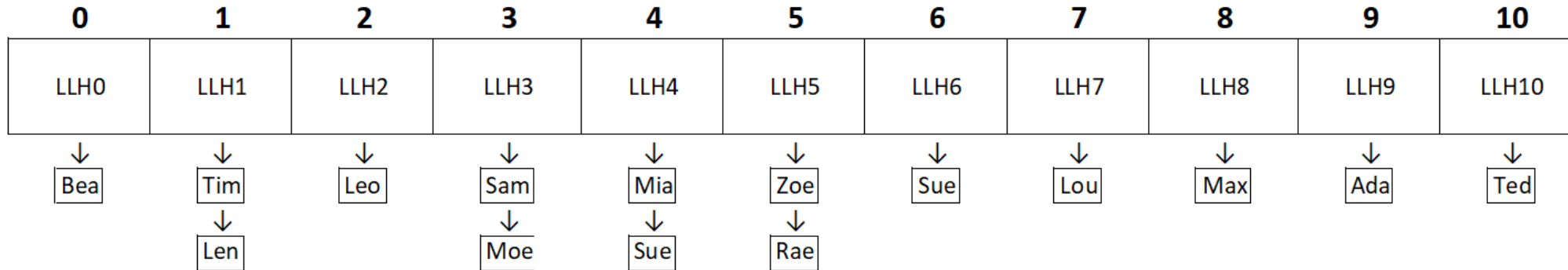


What happens when we try to add Sue which hashes to 4?



# Hashing with Separate Chaining

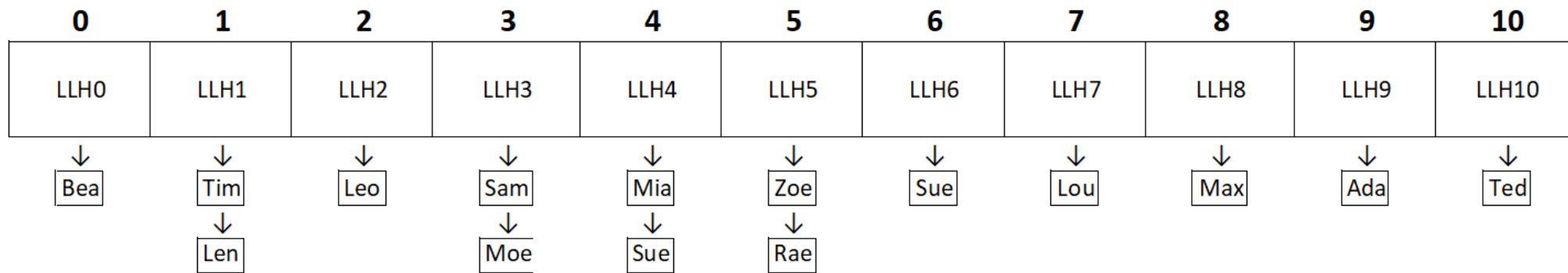
What happens when we add Len(1), Moe(3), Lou(7)



Any entries that hashes to match an existing entries gets added at the end of the linked list.

# Hashing with Separate Chaining

When an item is hashed and needs to be added to the hash table,



we need to check if it is the first item to be added at that location.

If it is, then we create a linked list head, malloc a new node, put the data in the node and store the linked list head in the array.

If it is not the first, then we use the linked list head stored in the array cell and traverse to the end of the list and add a newly malloced node.

# Hash Table – Uses

- *Associative arrays*: Hash tables are commonly used to implement many types of in-memory tables. They are used to implement associative arrays (arrays whose indices are arbitrary strings or other complicated objects).
- *Database indexing*: Hash tables may also be used as disk-based data structures and database indices (such as in dbm).
- *Caches*: Hash tables can be used to implement caches i.e. auxiliary data tables that are used to speed up the access to data, which is primarily stored in slower media.
- *Object representation*: Several dynamic languages, such as Perl, Python, JavaScript, and Ruby use hash tables to implement objects.
- Hash Functions are used in various algorithms to make their computing faster

# Binary Trees

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

Perfect Binary Tree

Full Binary Tree

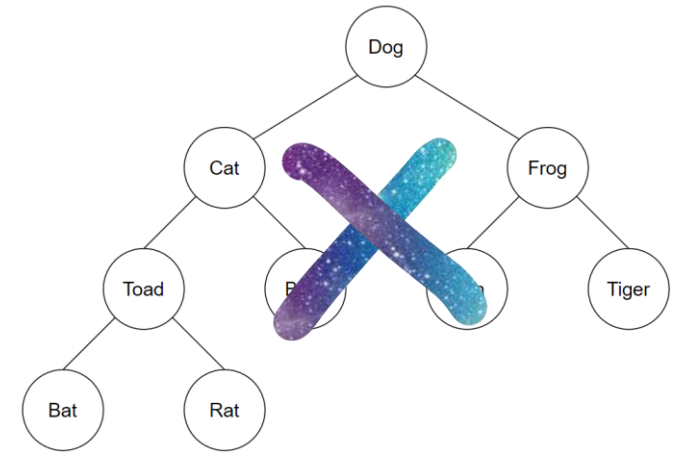
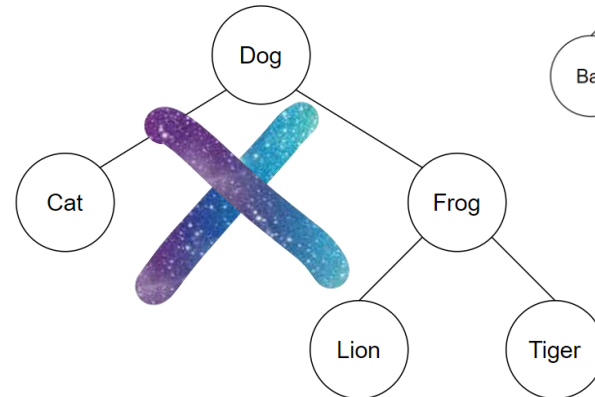
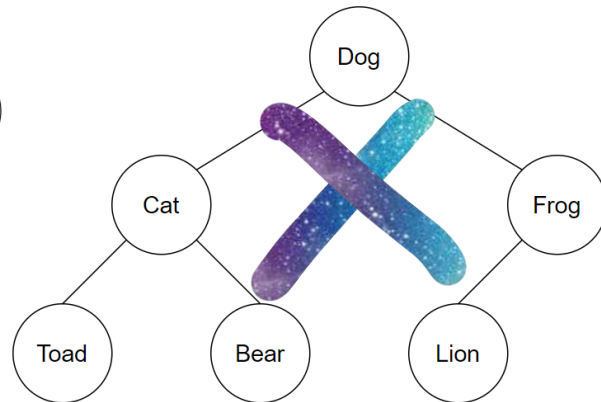
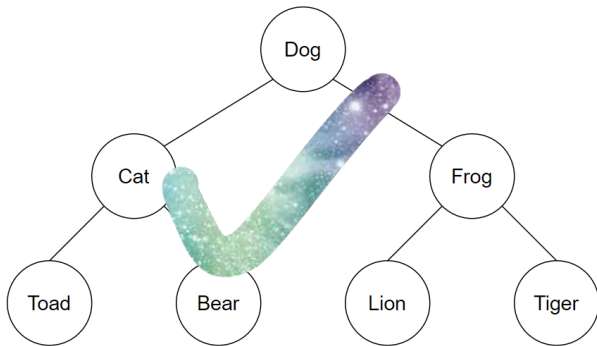
Complete Binary Tree



# 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.

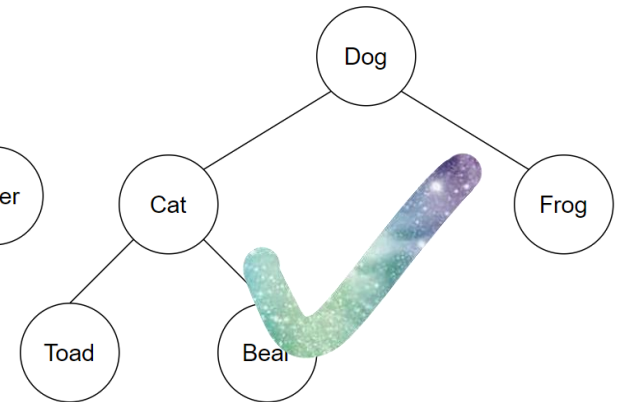
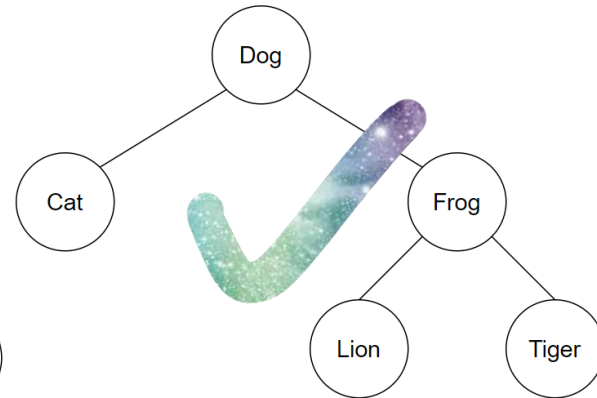
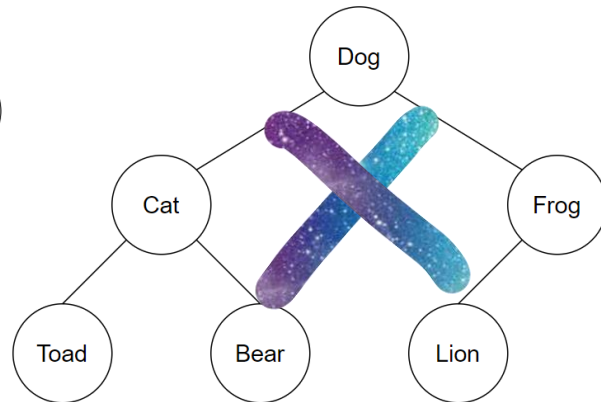
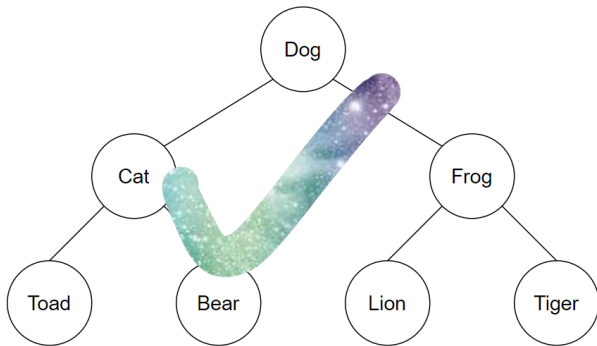


# Binary Trees

## Full Binary Tree

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

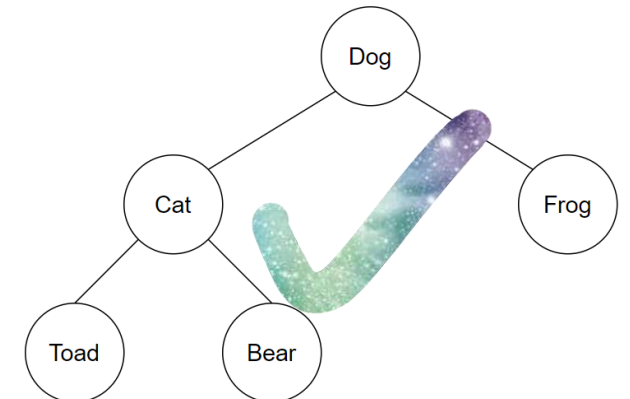
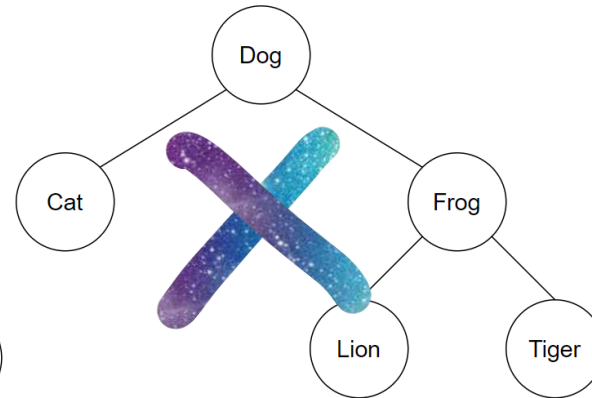
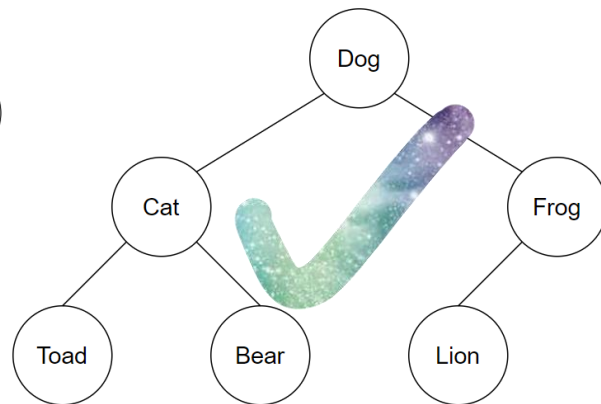
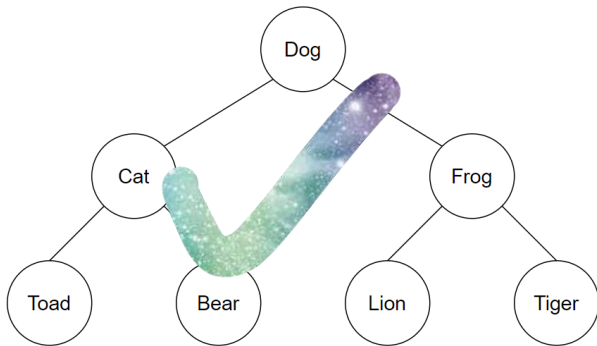
We can also say a full binary tree is a binary tree in which all nodes except leaf nodes have two children.



# 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 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

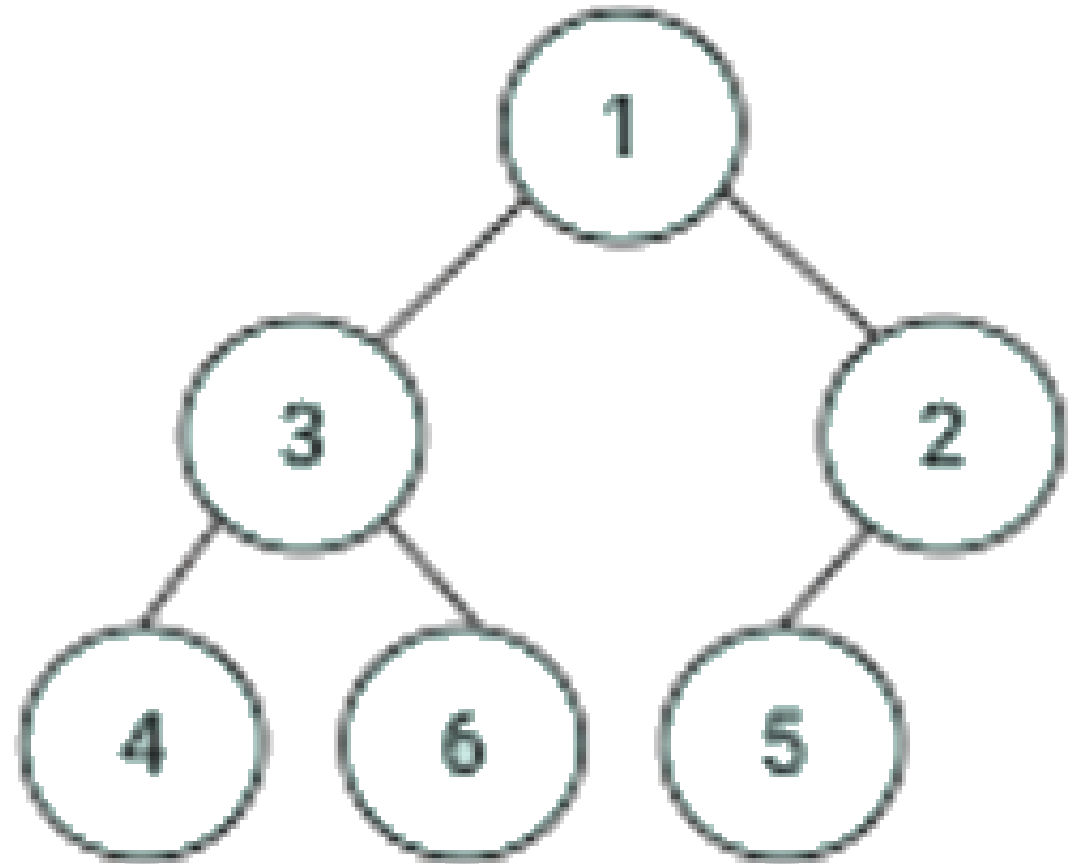
Is this a Min Heap?

1. Is it a complete tree?

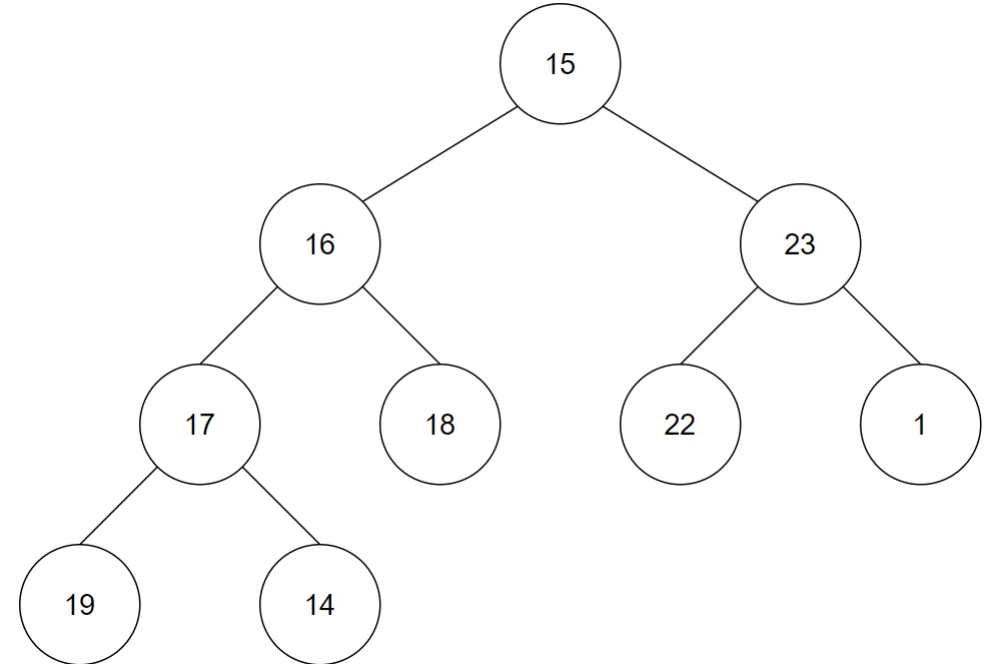
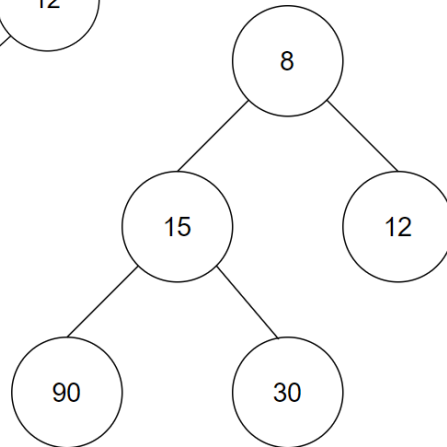
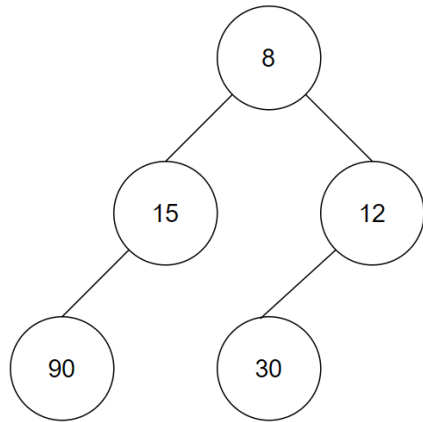
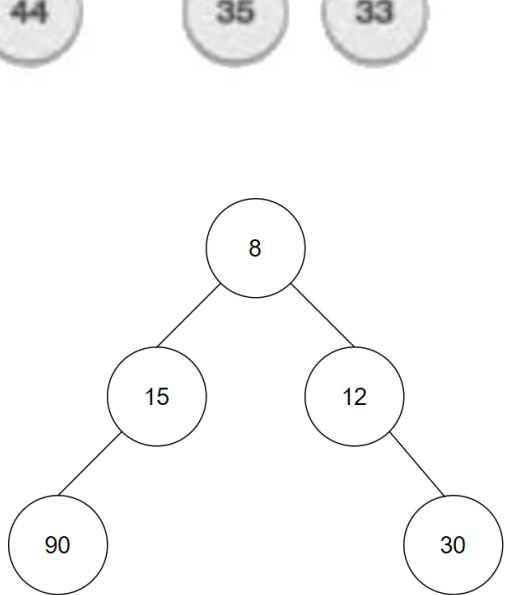
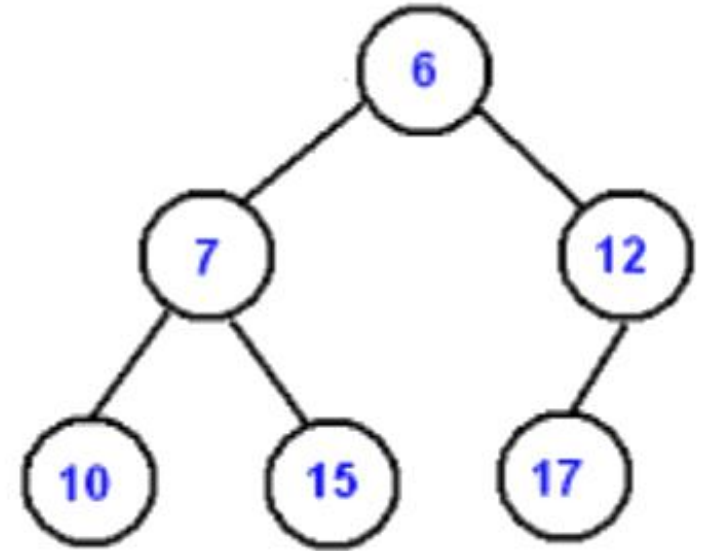
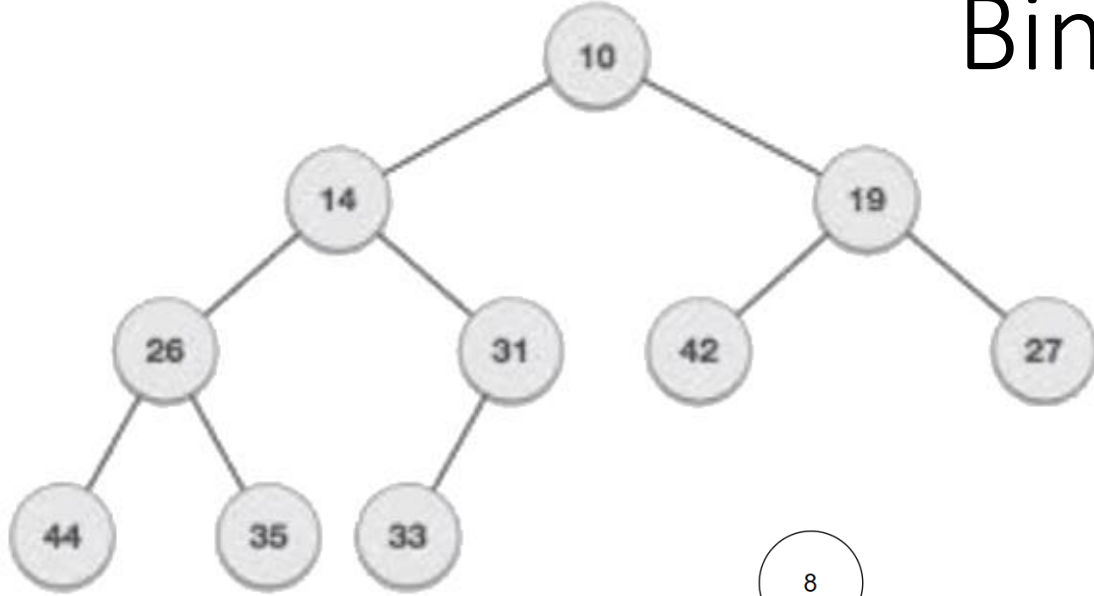
all the levels are completely filled except possibly the last level and the last level has all keys as left as possible

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



# 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

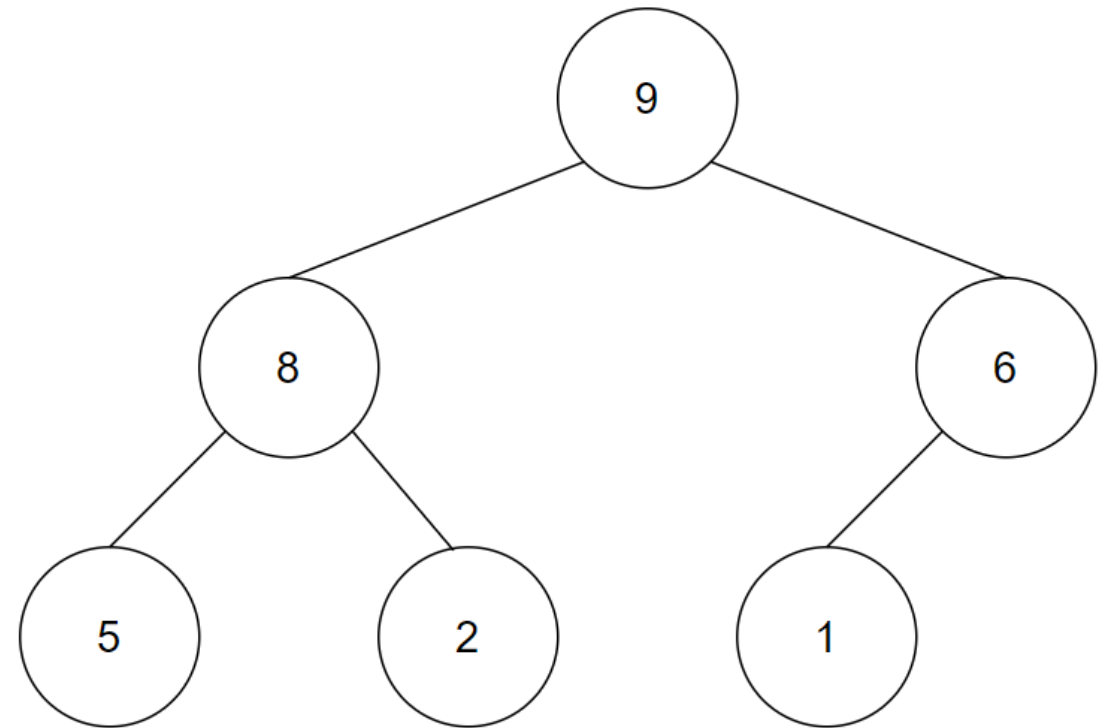
Is this a Max Heap?

1. Is it a complete tree?

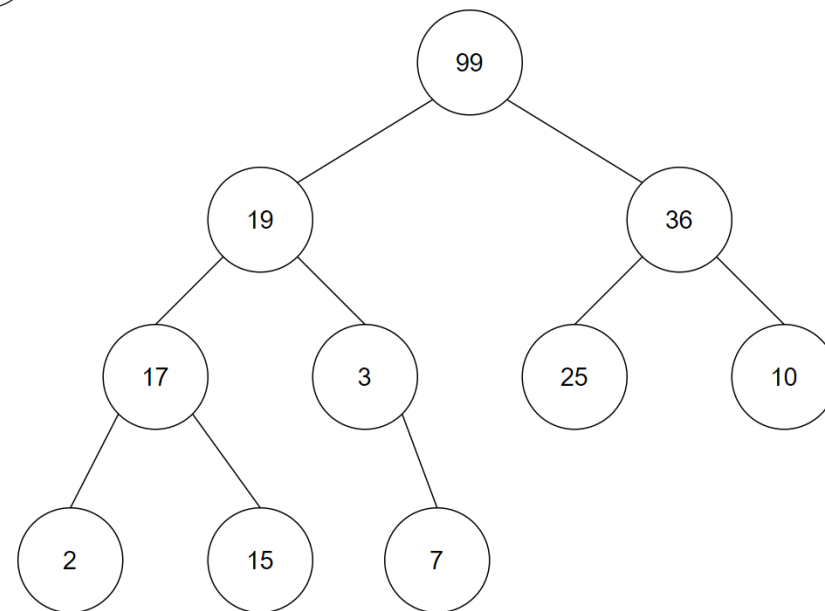
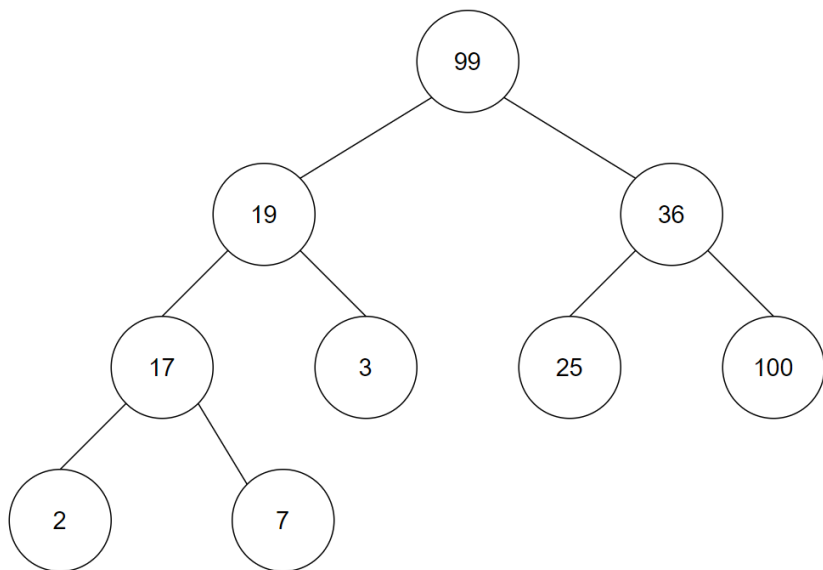
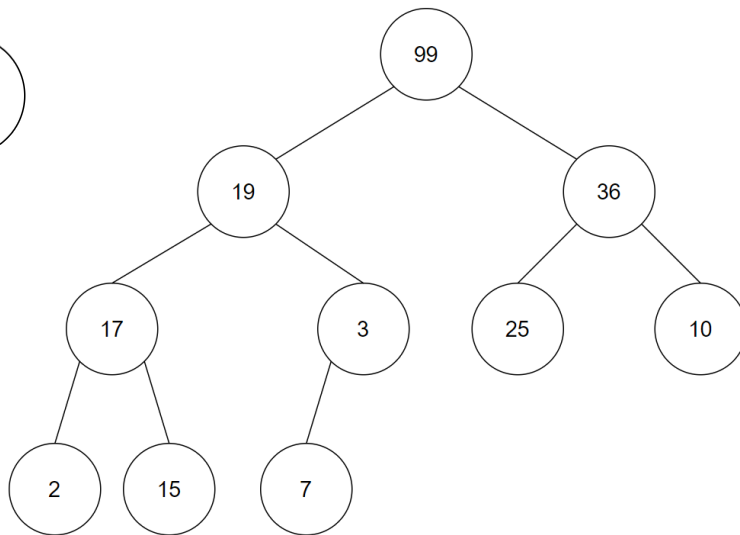
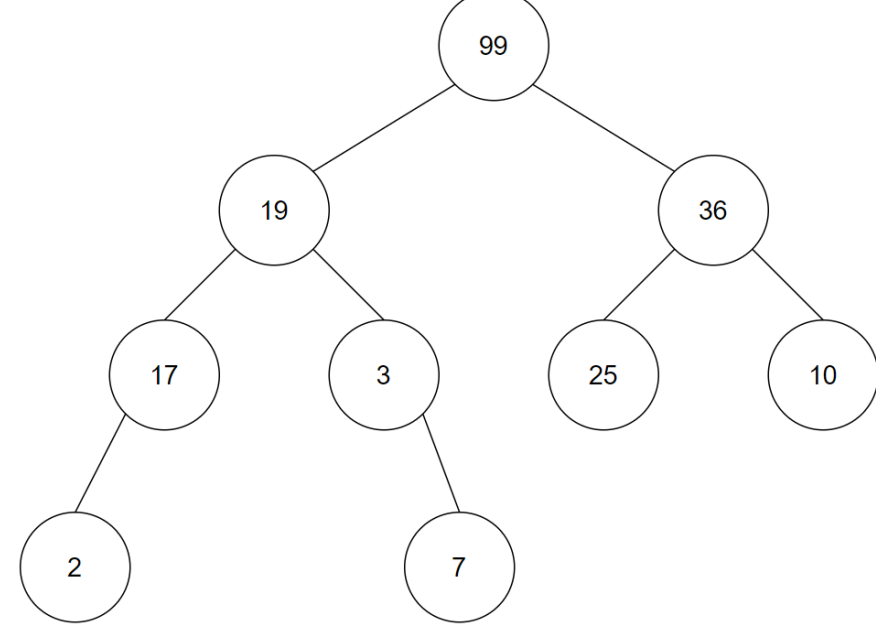
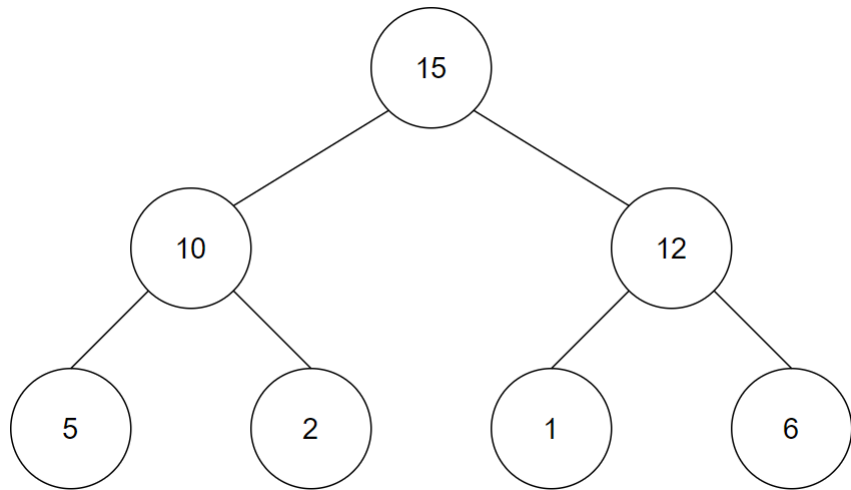
all the levels are completely filled except possibly the last level and the last level has all keys as left as possible

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



# 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?

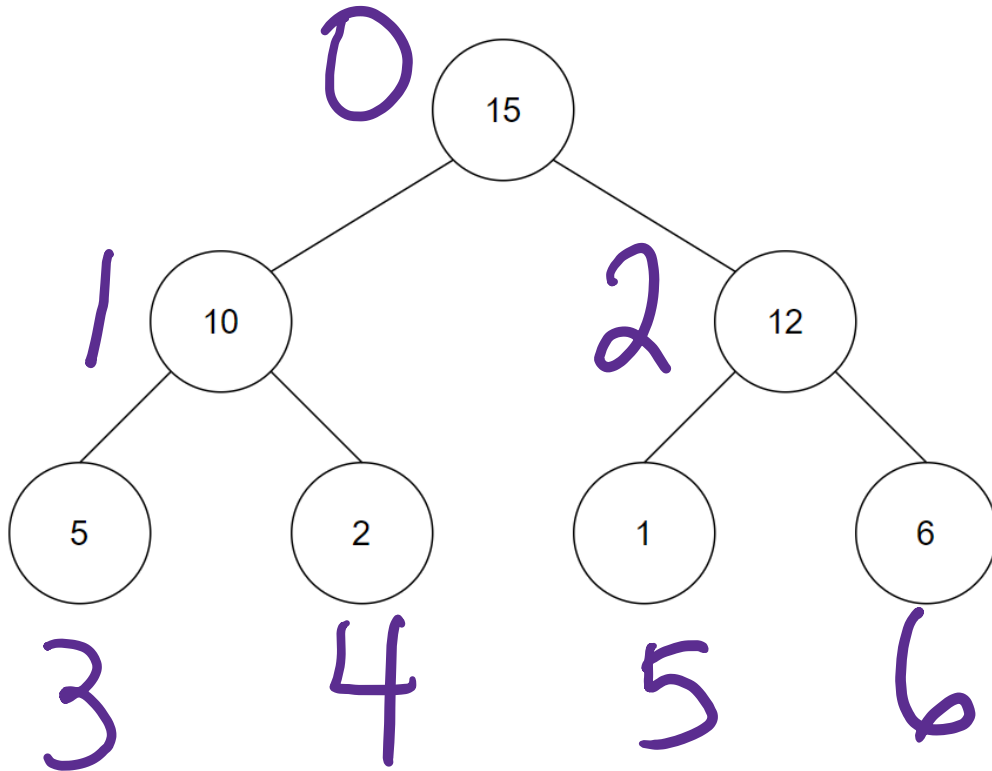


**THERE'S A  
BETTER WAY.**



# Binary Heap

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

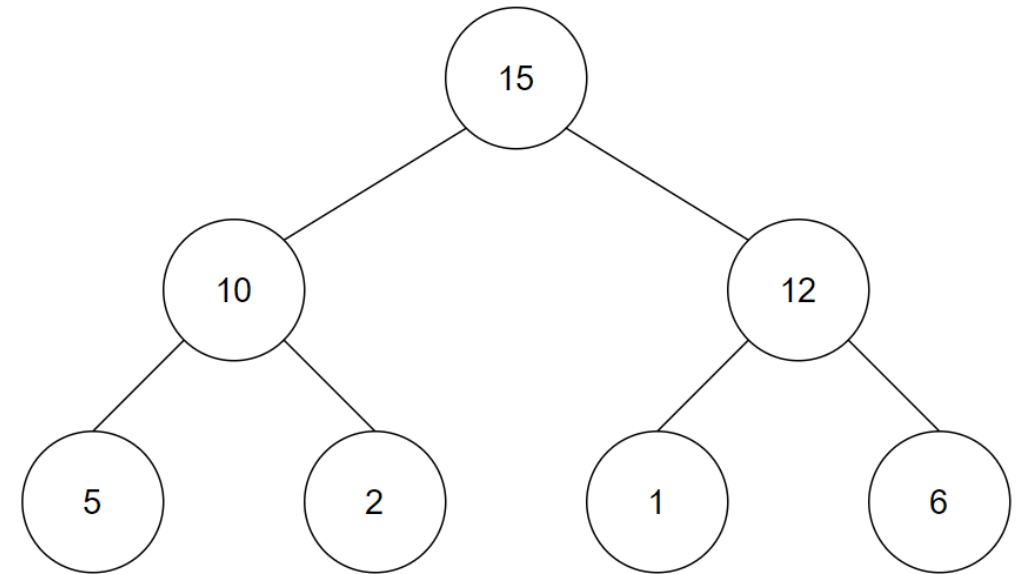


0	1	2	3	4	5	6
15	10	12	5	2	1	6

# Binary Heap

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

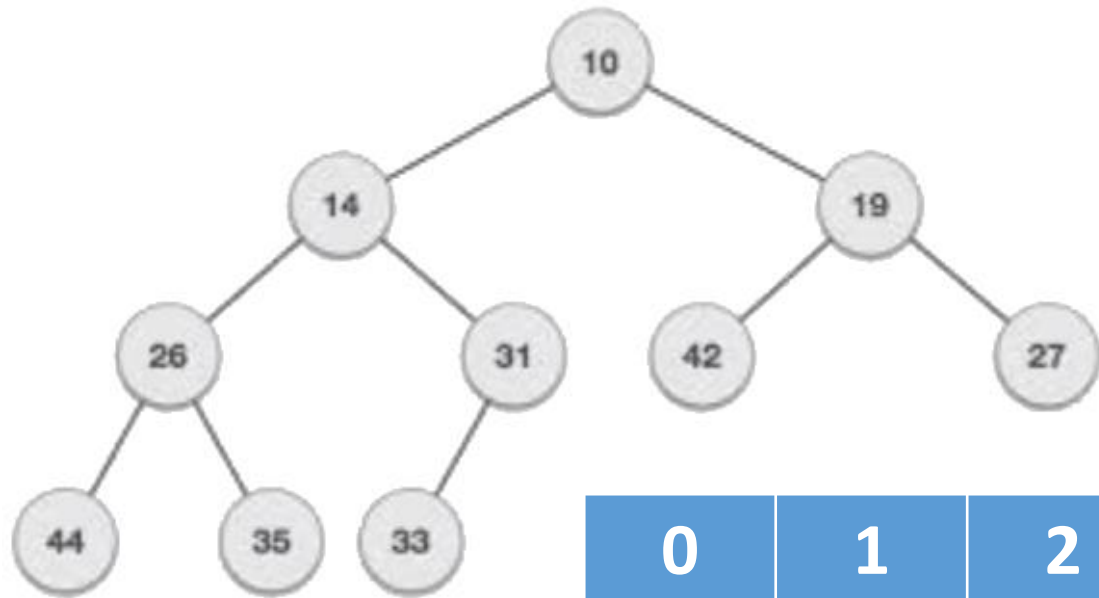
0	1	2	3	4	5	6
15	10	12	5	2	1	6





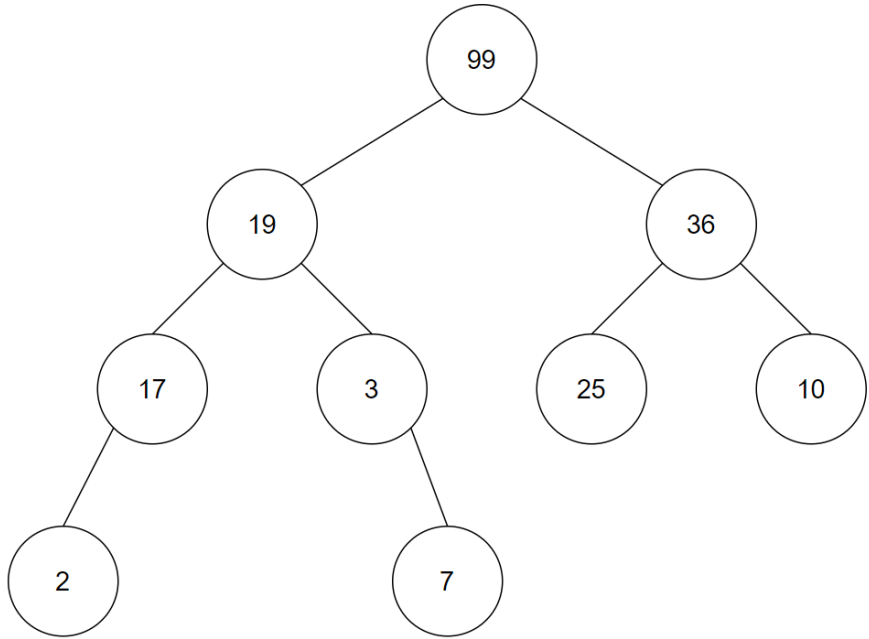
# Binary Heap

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



0	1	2	3	4	5	6	7	8	9
10	14	19	26	31	42	27	44	35	33

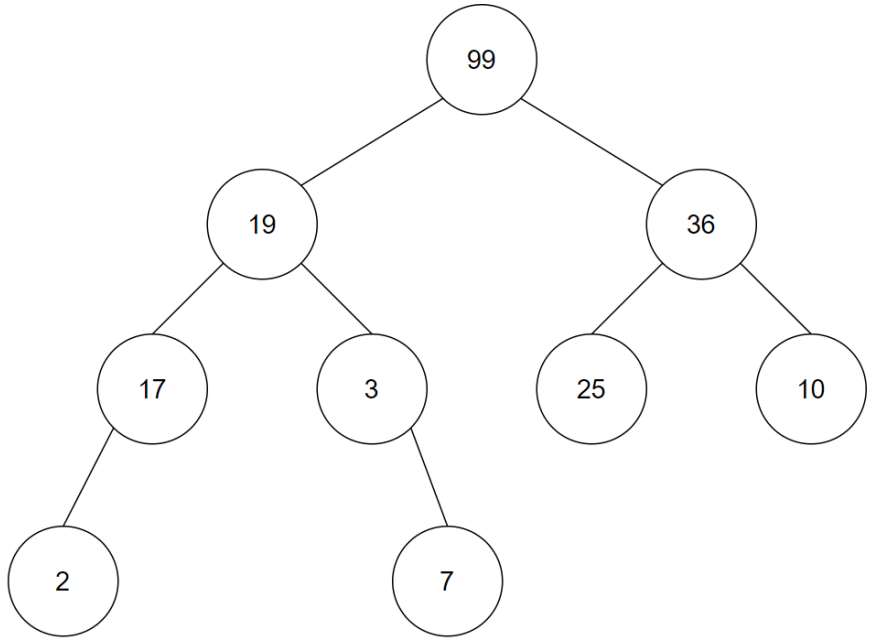
# Binary Heap



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?

0	1	2	3	4	5	6	7	8
99	19	36	17	3	25	10	2	7

# Binary Heap



What if we just left blanks?

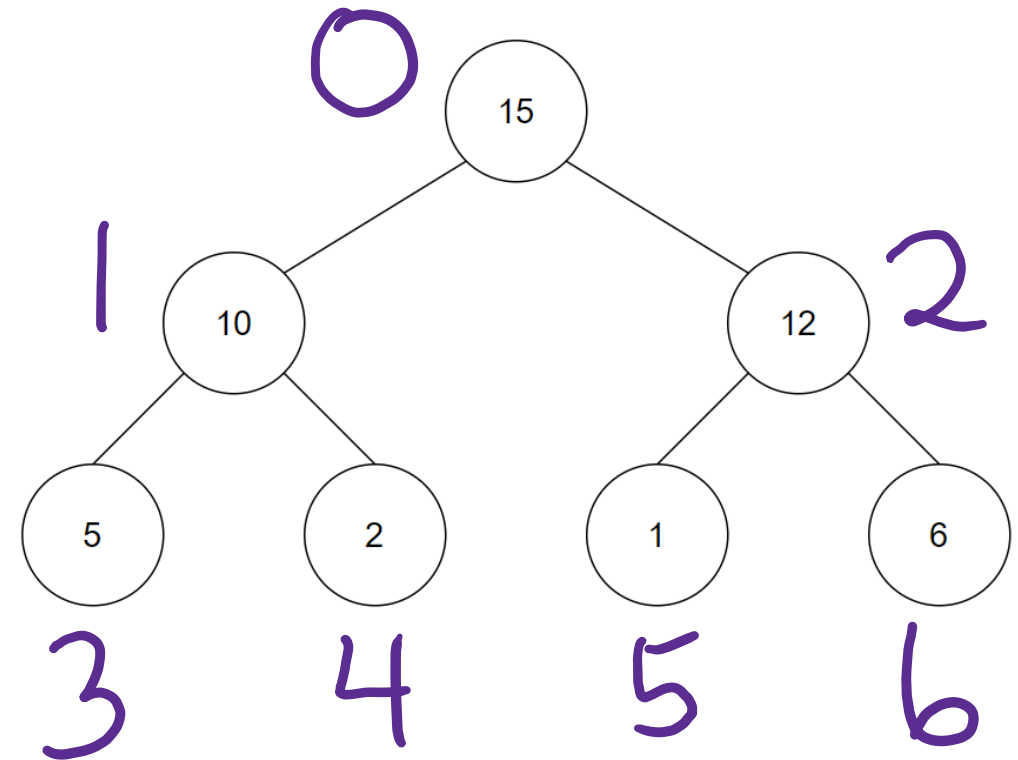
0	1	2	3	4	5	6	7	8	9	10
99	19	36	17	3	25	10	2			7

# 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].

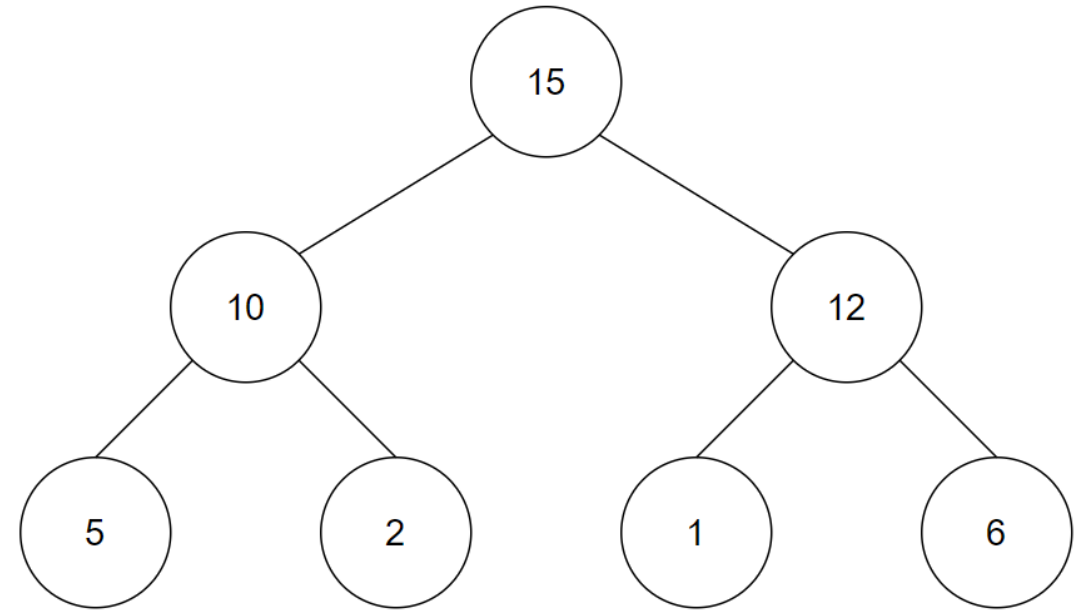
When using zero indexed arrays, we can state that for a index of  $i$

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 =  $2i + 1$

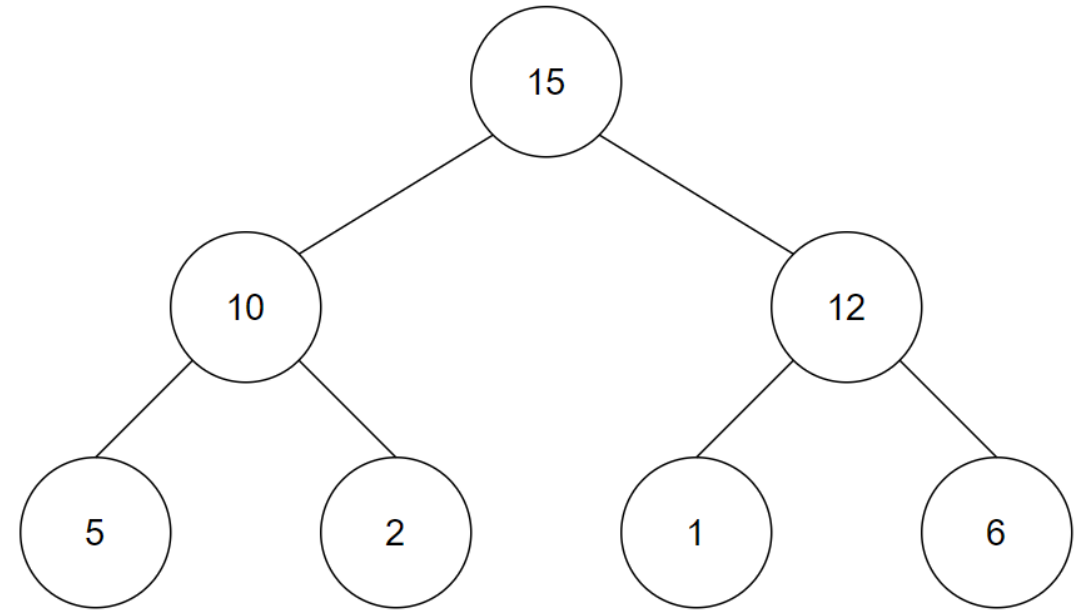
right child =  $2i + 2$

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?

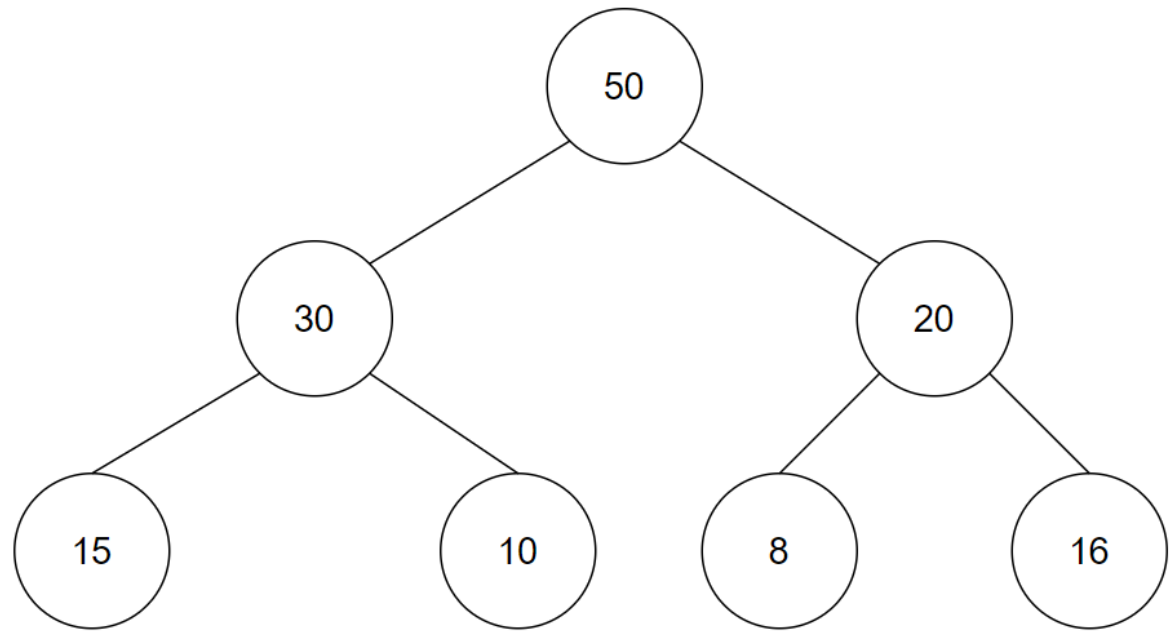
Let's say we want to add 60?

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

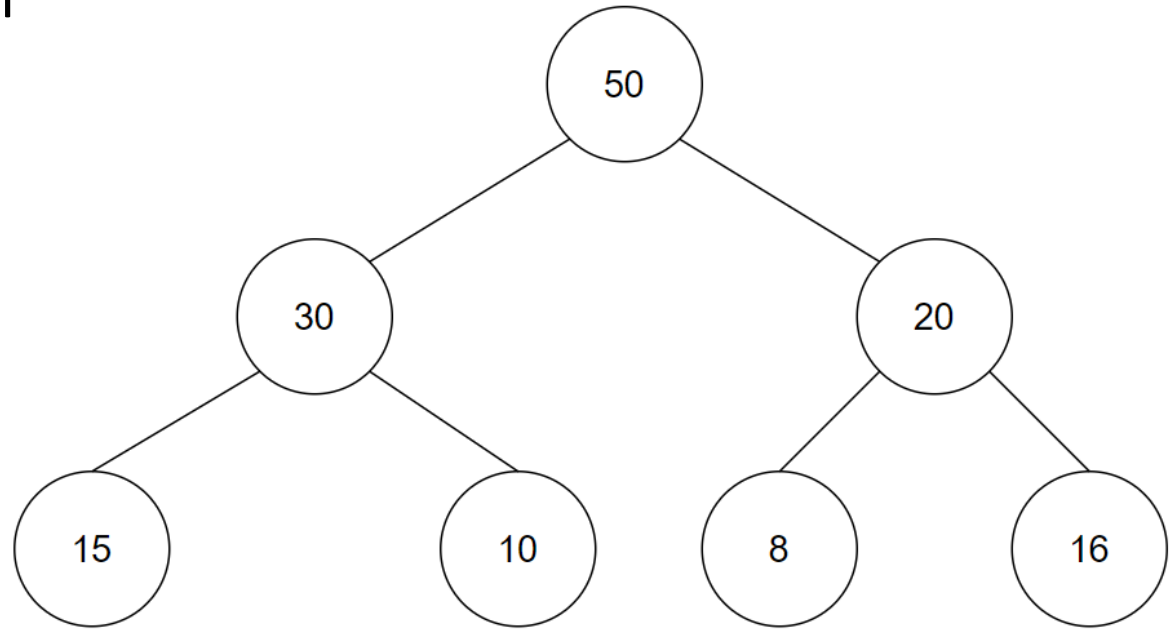
Adding a value to heap is called inserting or pushing.

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

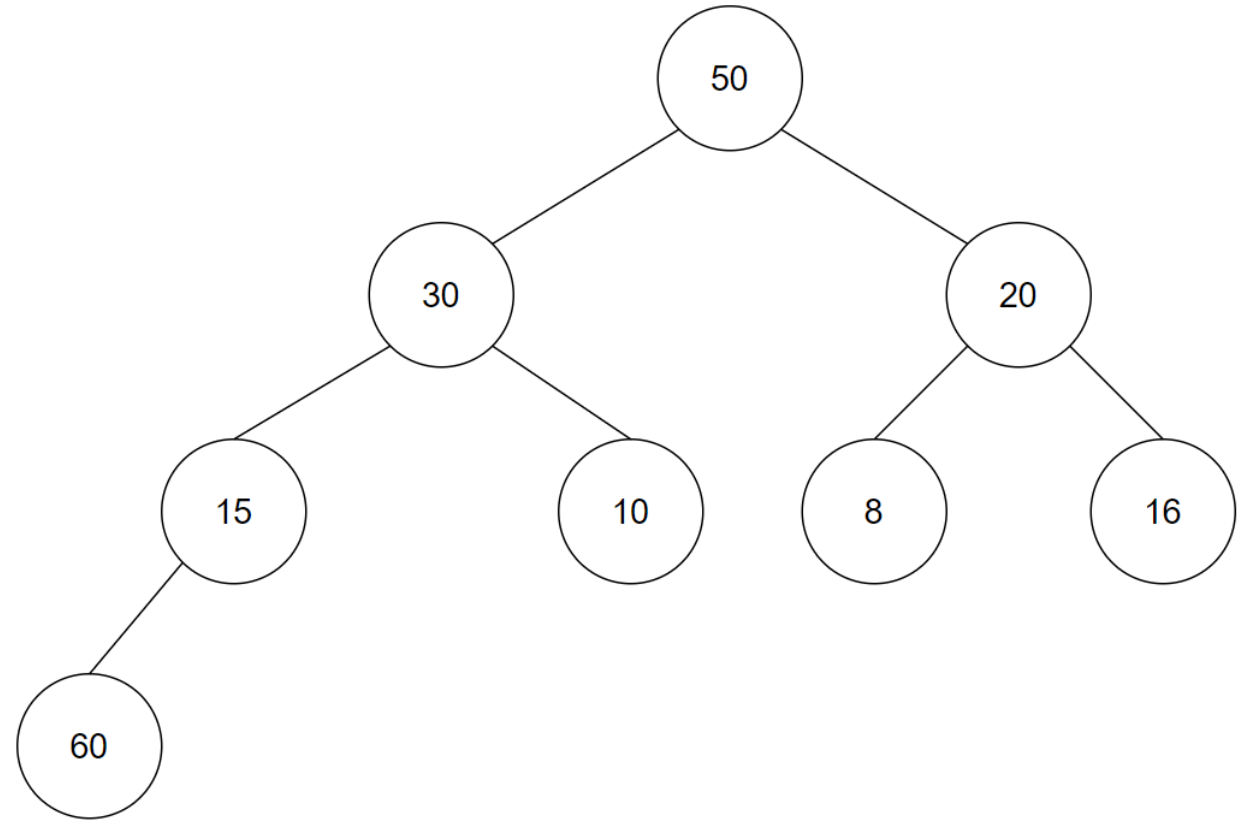
What does that do to our tree?

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

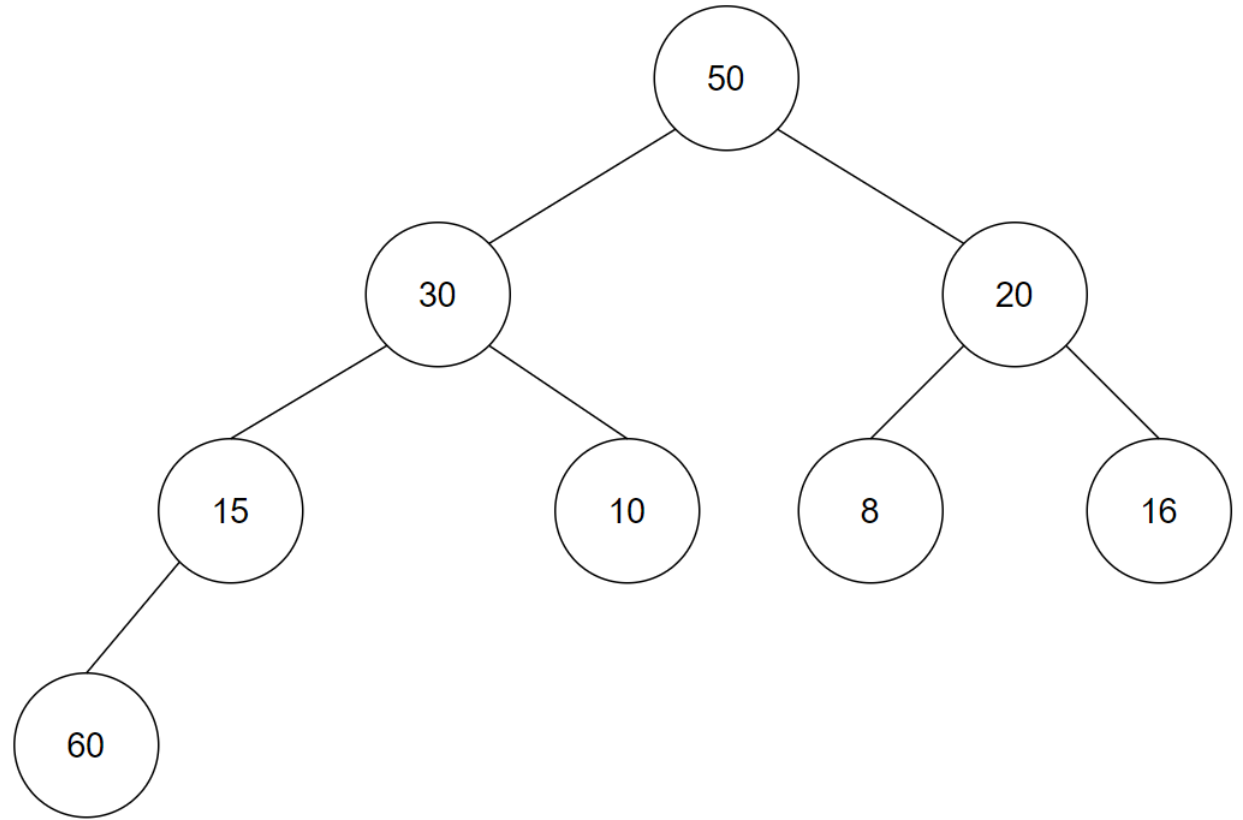
# Adding to a Max Heap

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

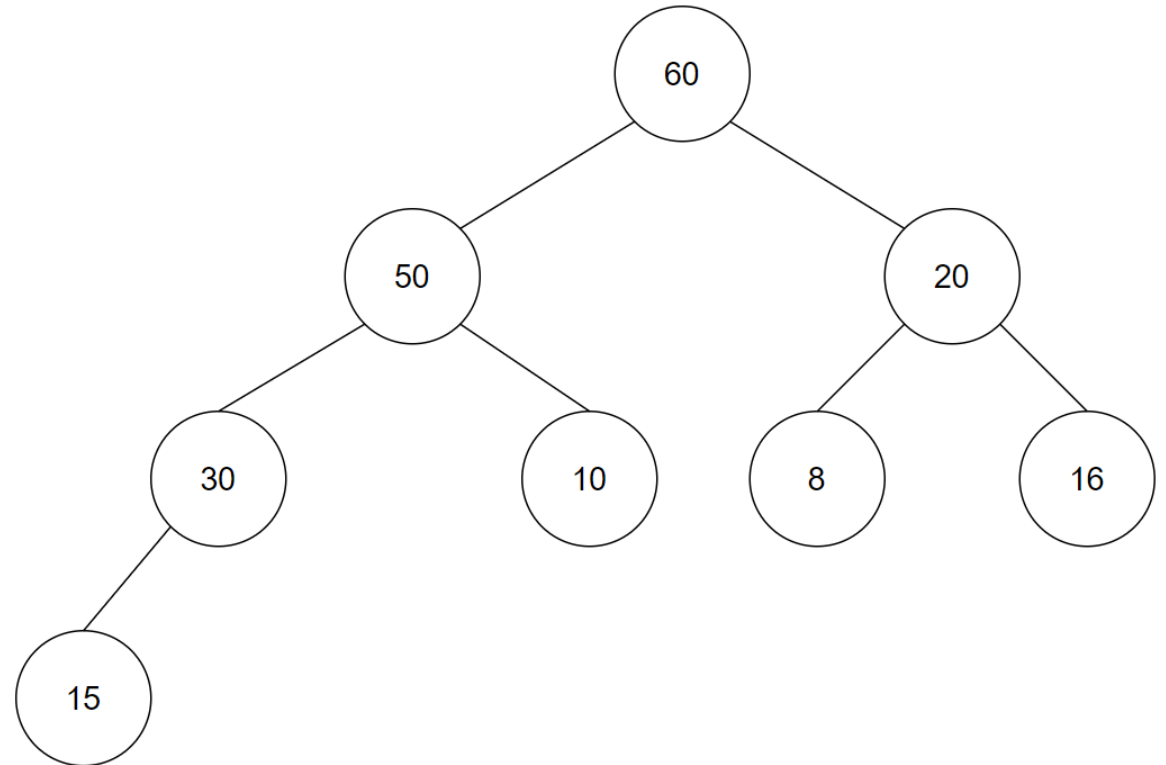
# Adding to a Max Heap

Yes!

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



# HEAPIFY



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

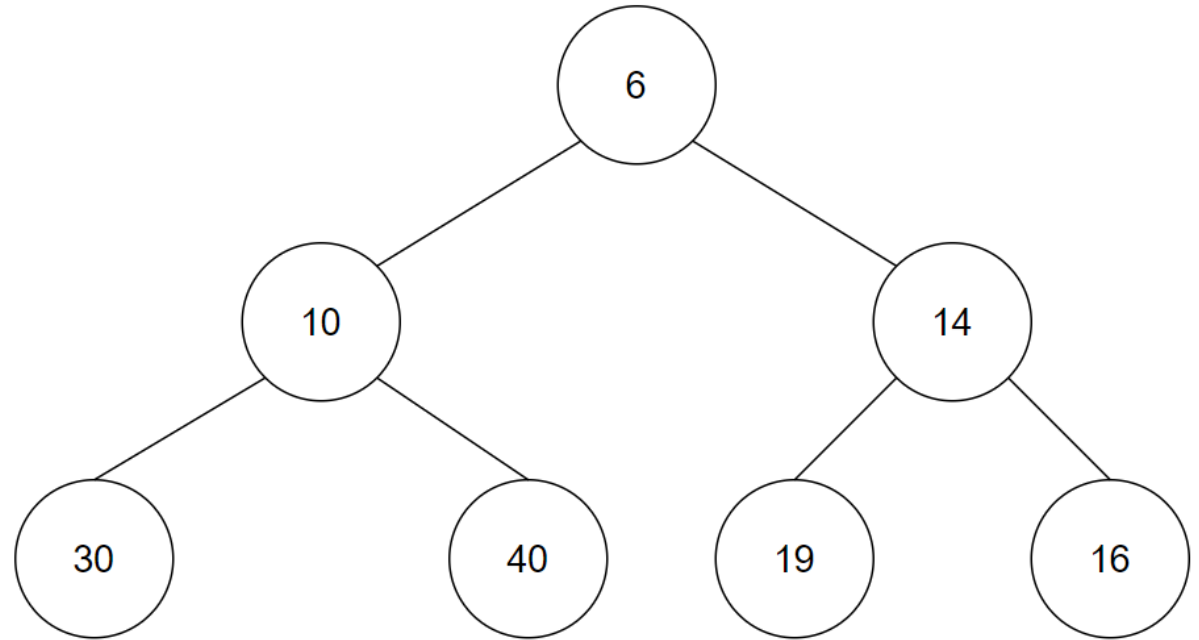
# Adding to a Min Heap

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

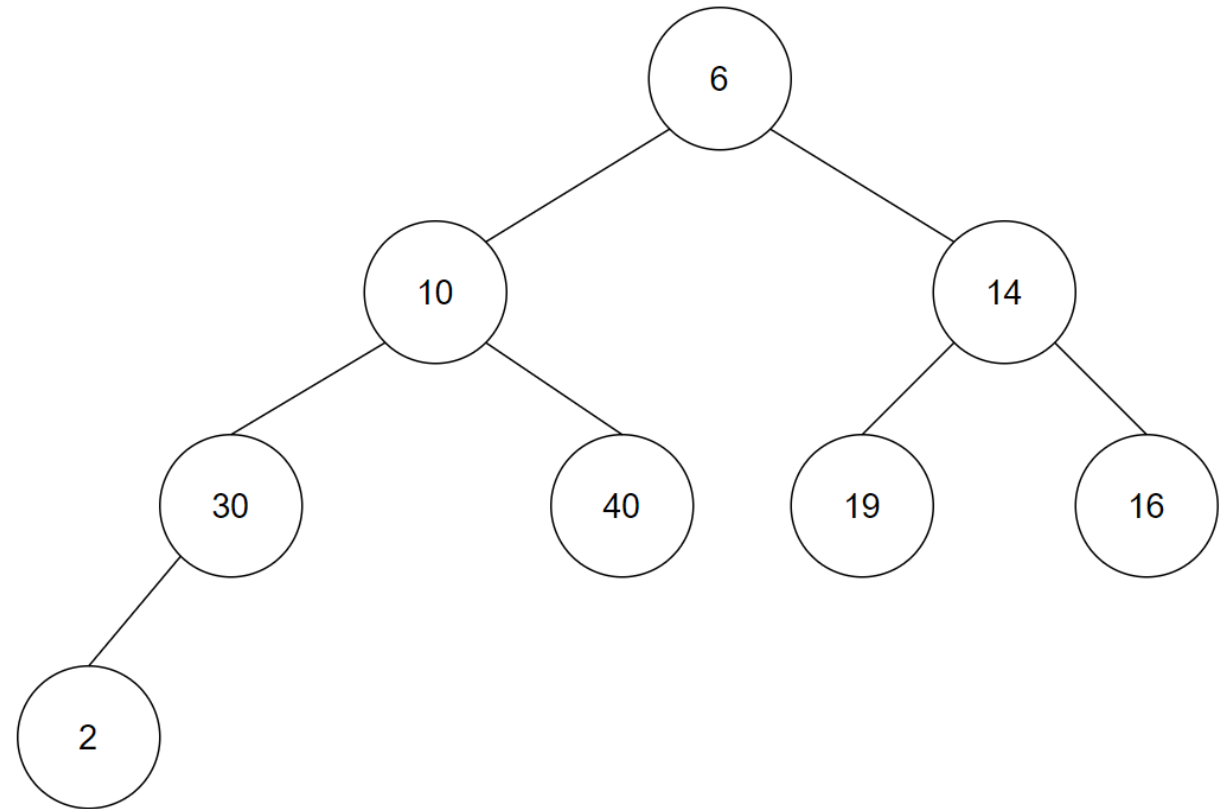
Let's add the 2 just like we did 60.

Now we need to swap 2 and 30.

Now swap 2 and 10.

Now swap 6 and 2.

Have we restored the heap property?

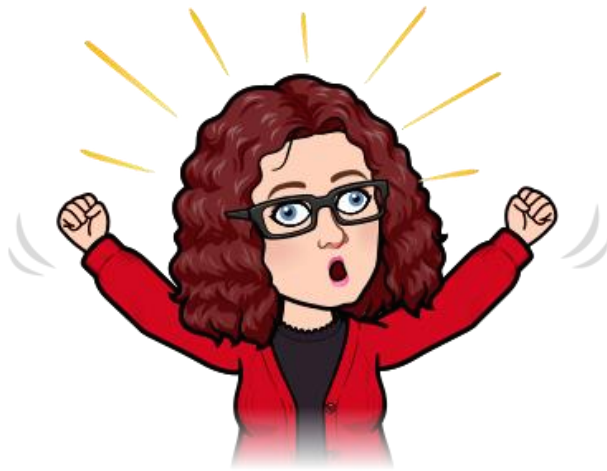


0	1	2	3	4	5	6	7
6	10	14	30	40	19	16	2

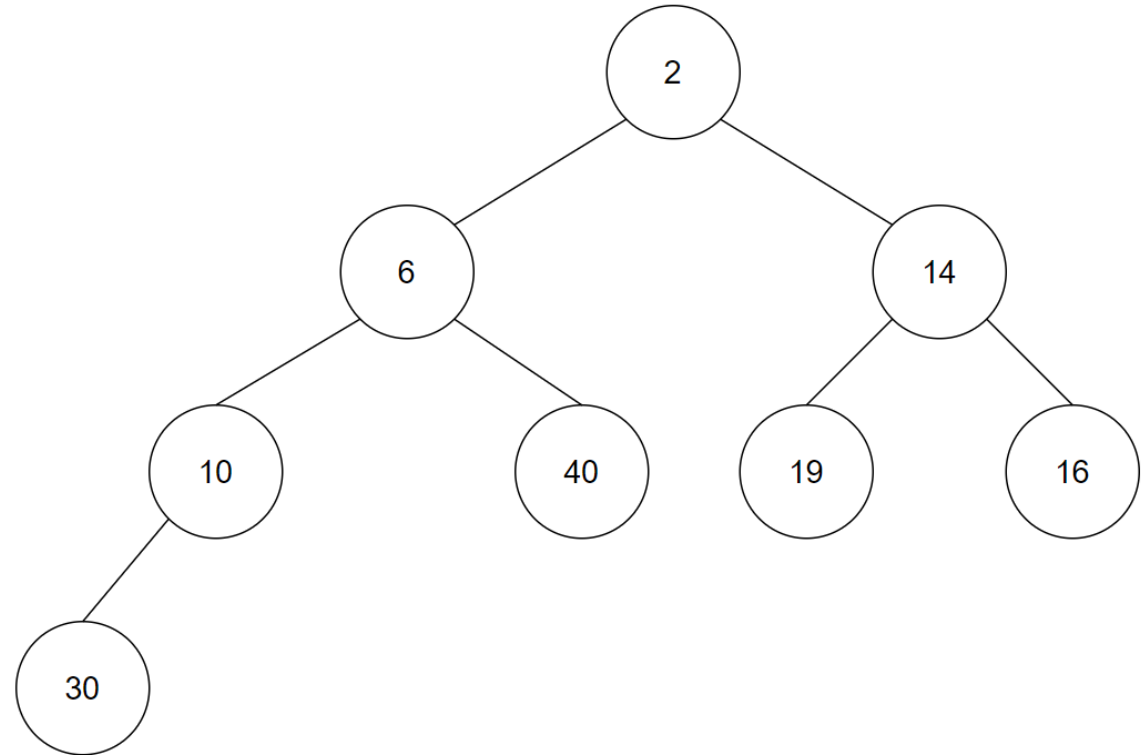
# Adding to a Min Heap

Yes!

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



# HEAPIFY



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

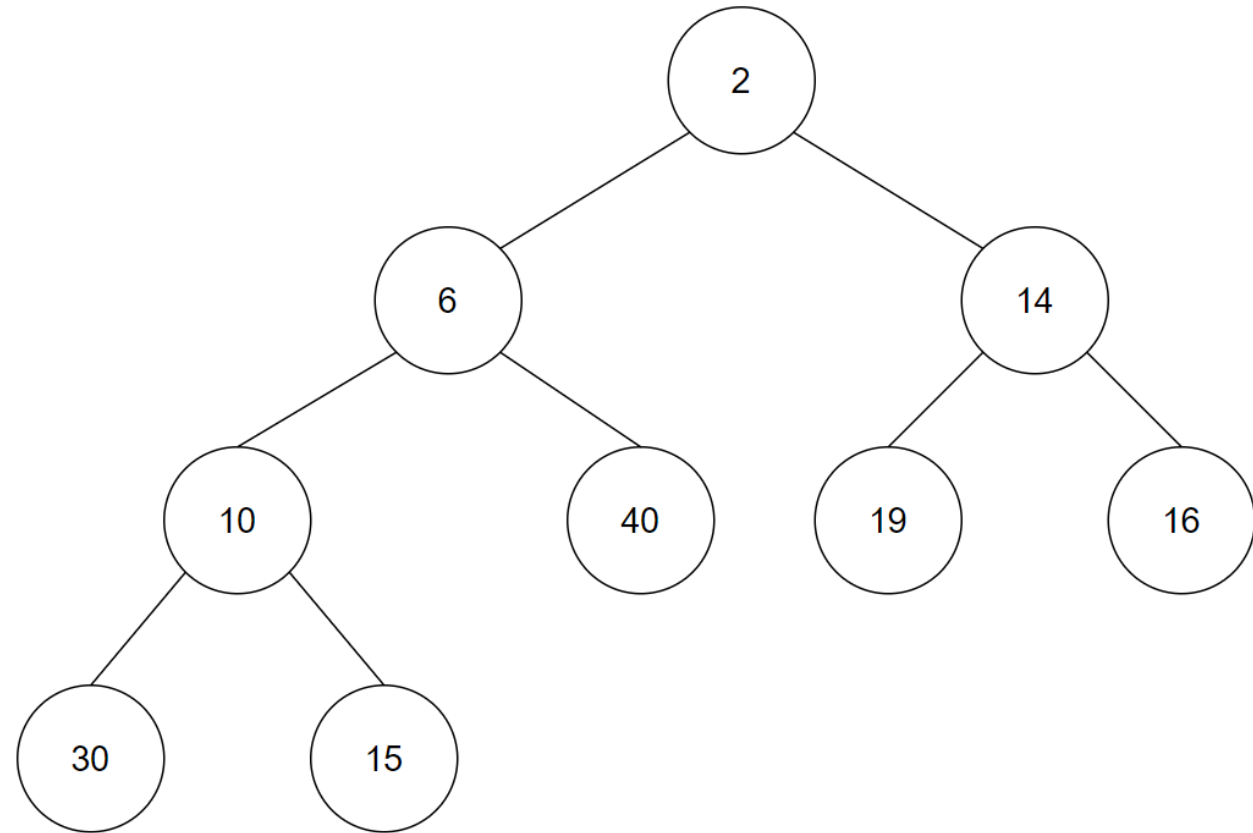
# Adding/Inserting with Heap

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.



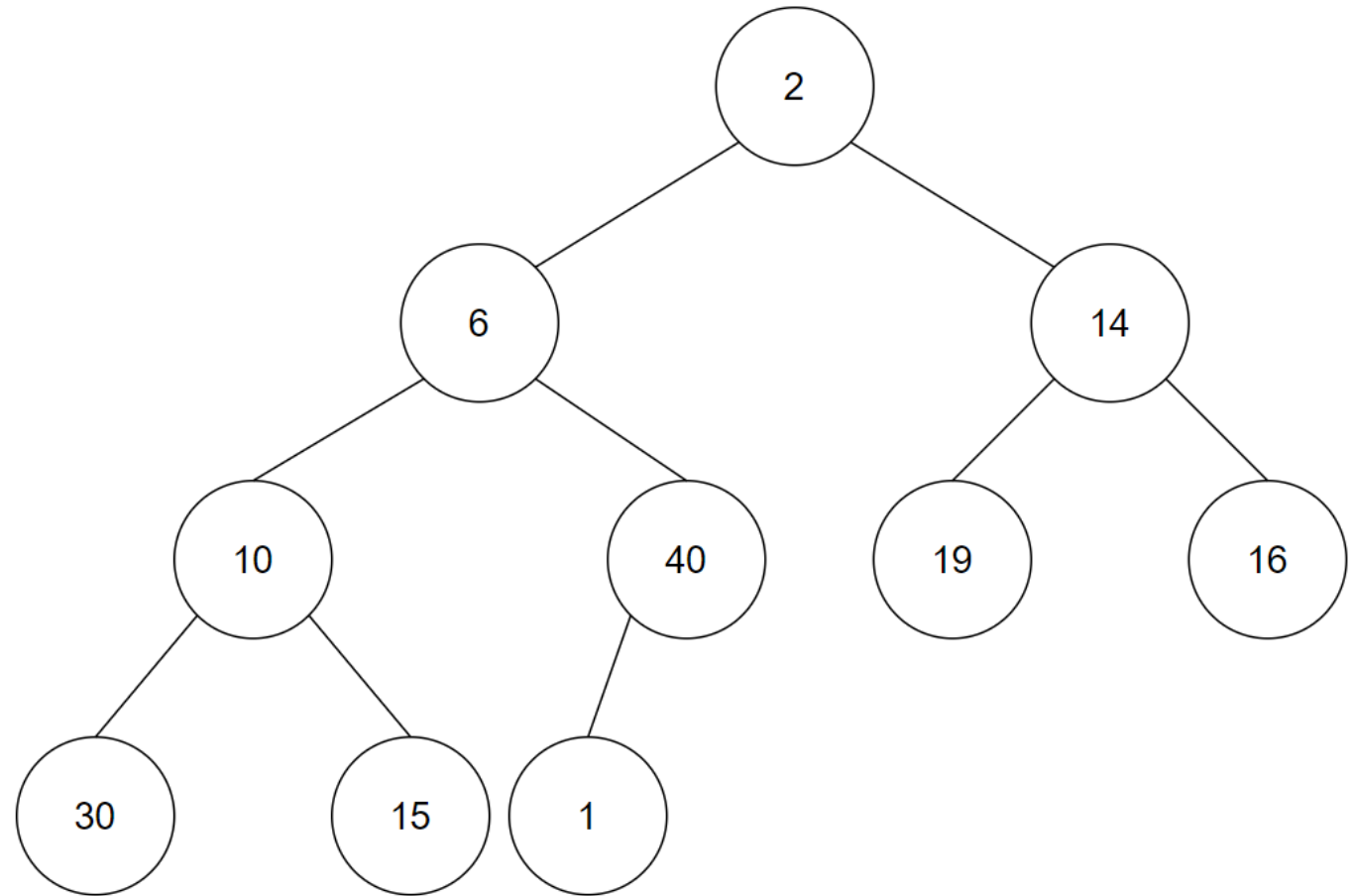
0	1	2	3	4	5	6	7	8	9
2	6	14	10	40	19	16	30	15	

# 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



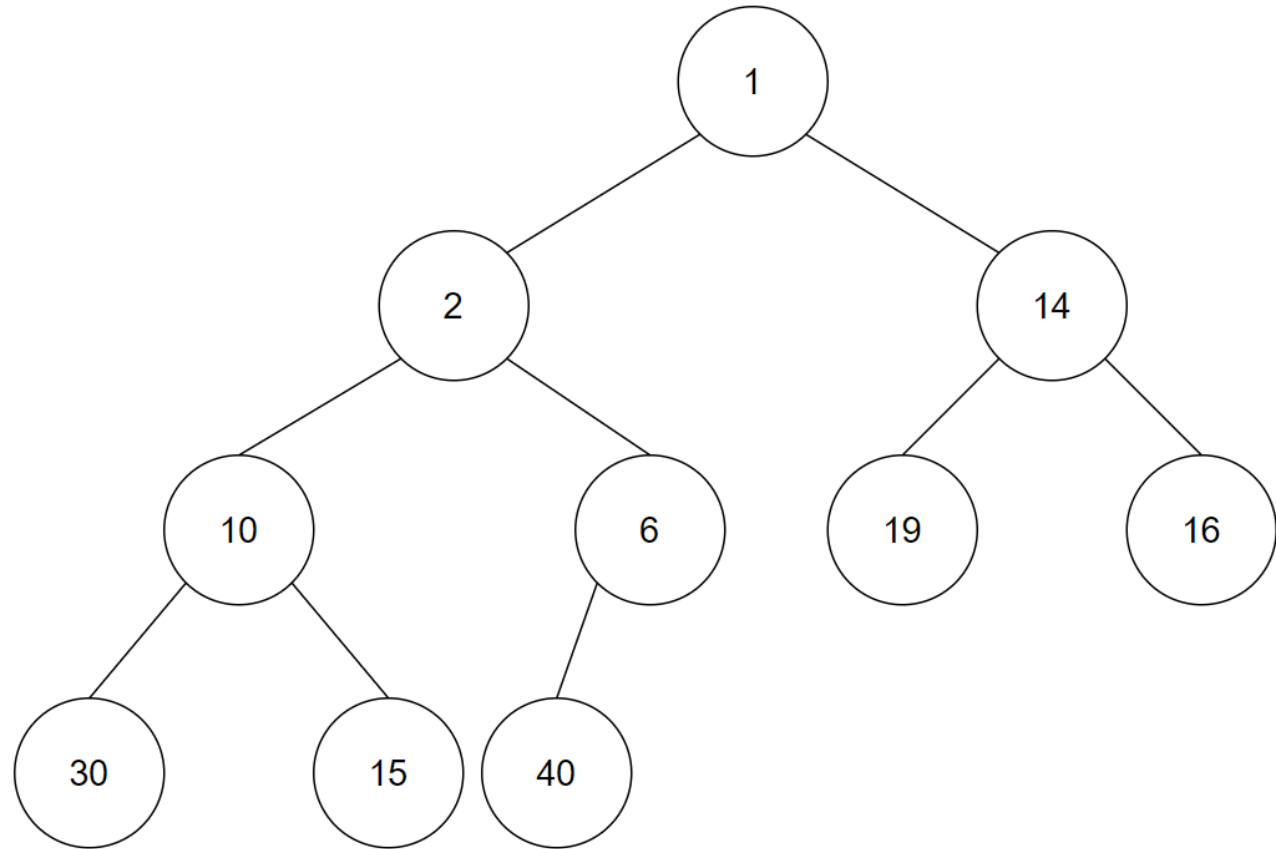
# Adding/Inserting with Heap

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