Final Review Session

DMS & Asymptotics

with Aarin & Dhruv!

Asymptotics Definitions

- **Big O**: f(N) = O(g(N)) if f(N) <= Cg(N) when N is large (for **some** constant C)
  – The upper bound in terms of the input size. A function that has Big O in f(x) can grow at most as fast as f(x).
- **Big Omega:** f(N) = Ω(g(N)) if f(N) >= Cg(N) when N is large (for **any** constant C)
  – **The lower bound in terms of the input size.** A function that has Big Omega in f(x) can grow at least as slowly as f(x).
- **Big Theta**: f(N) = Θ(g(N)) if f(N) = O(g(N)) and f(N) = Ω(g(N))
  – **The tightest bound in terms of the input size.** Only exists when the tightest Big O and the tightest Big Omega bound converge to the same value.

Fun Sums

Constant Sequence: $1 + 1 + 1 + 1 + \ldots + 1 = \Theta(N)$ (these are N 1's)

Arithmetic Sequence: $1 + 2 + 3 + 4 + \ldots + N = \Theta(N^2)$

Geometric Sequence: $1 + 2 + 4 + 8 + \ldots + N = \Theta(N)$

Generalized:

Constant Sequence: $c + c + c + \ldots + c = \Theta(N)$

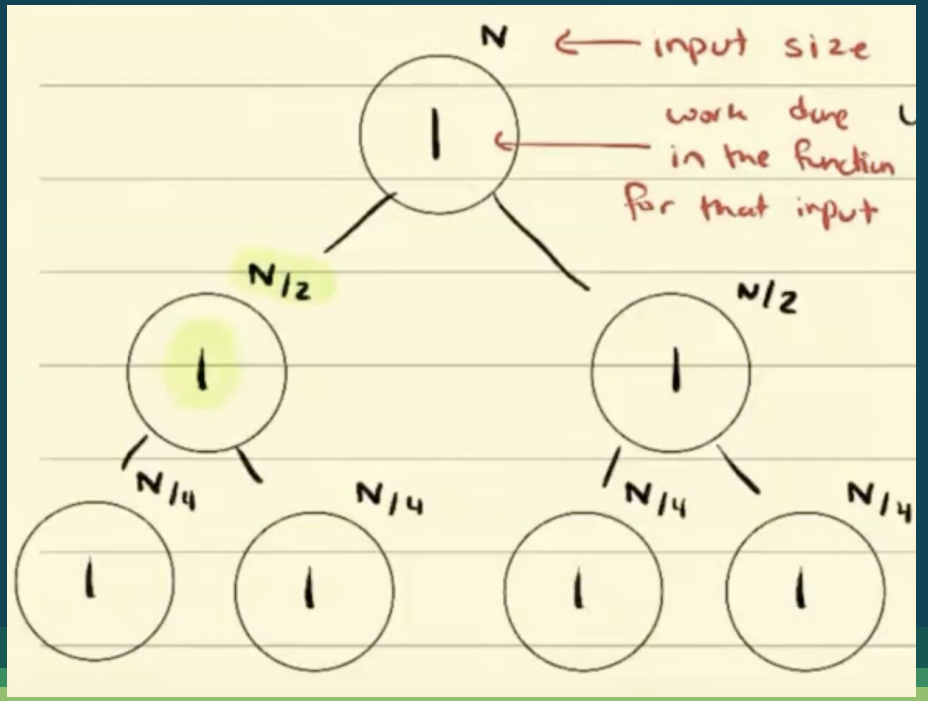
Arithmetic Sequences: $1 + (1+r) + (1+2r) + \ldots + (1+(N-1)r) = \Theta(N^2)$

Geometric Sequence: $1 + r + r^2 + \ldots + N = \Theta(N)$

Recursion

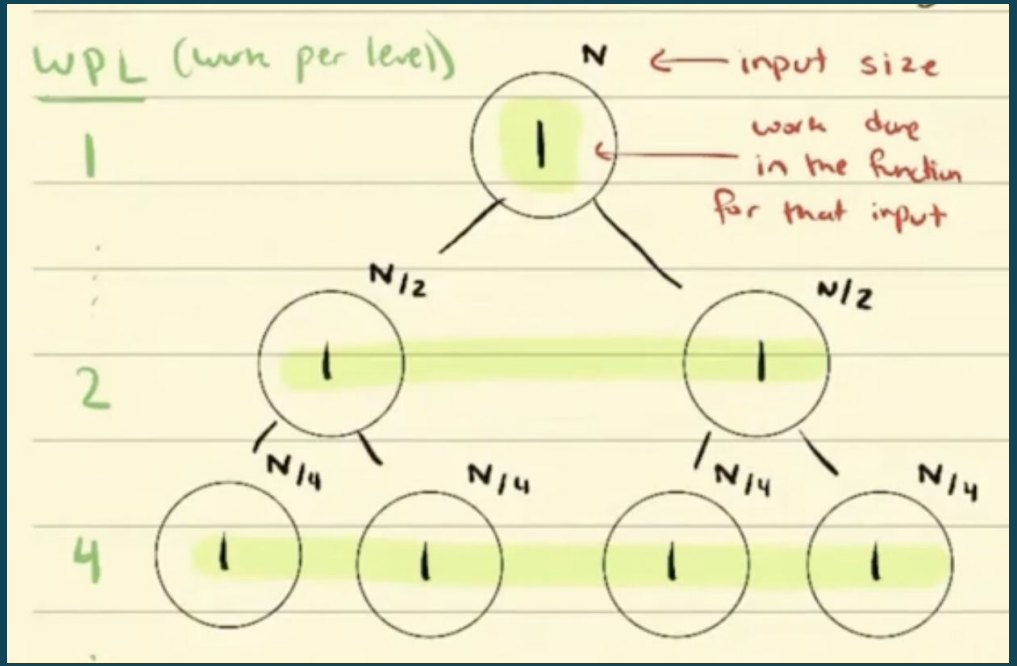
1) Draw out the tree diagram (what subproblems are there?)

```
void f(int N) {
BC  [ if (N > 0) {
        return;
    }
work [ g(N); // g(N) ∈ Θ(1)
RCs  [ f(N/2);
       f(N/2);
}
```



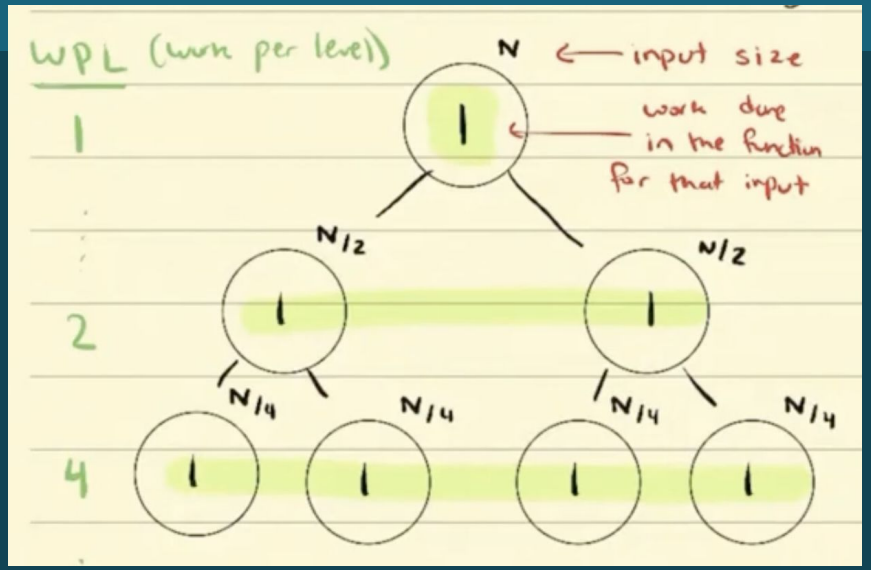
Recursion

2) Divide tree diagram into layers



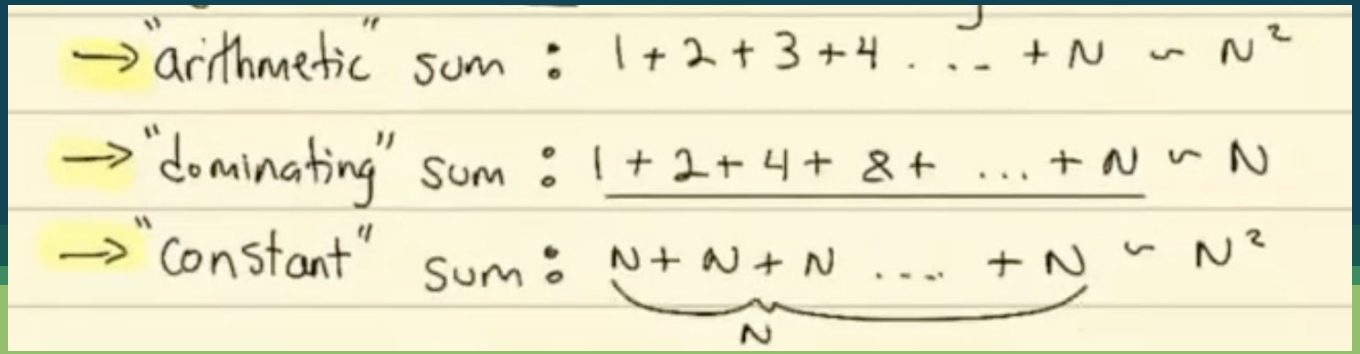
Recursion

3) Recognize which sum we're dealing with:



Runtime: $1 + 2 + 4 + 8 + \ldots + N = \Theta(N)$

Fun Sums Again:



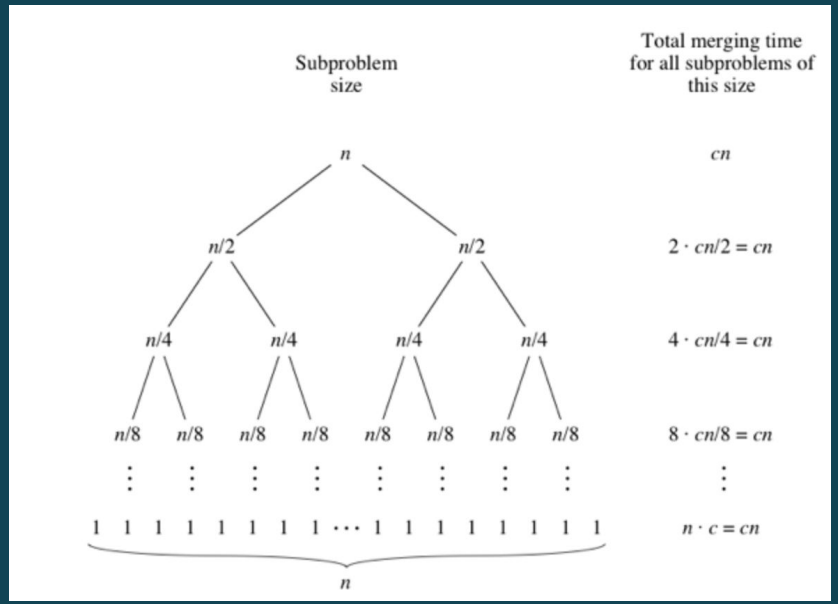
Useful Method: Finding the Tree Height

1) Pattern where every level has the same amount of work
2) Total Work = (# of levels) * (work per level)
3) Find the # of levels by finding the height of a tree
    a) Sometimes we can just look at it
    b) Important Case:

Nodes go from N → N/2 → N/4 → … → 4 → 2 → 1

We rewrite as $2^{\log N} \rightarrow 2^{\log N - 1} \rightarrow 2^{\log N - 2} \rightarrow \ldots \rightarrow 2^2 \rightarrow 2^1 \rightarrow 2^0$

The length of this sequence = logN



Iteration

1. Find Stopping Condition (Ex: x < N)
2. Find mode of iteration (Ex: i++)
3. Use 1 & 2 to determine the number of iterations in a loop

If the number of iterations is constant for “inner” for loops, can just multiply to get the number of iterations →

```
for (int i = 0; i < n; i++) {
   for (int j = 0; j < n; j++) {
      f(i, j); // runs in constant time

   }
}
```

Total Number of Iterations: $\Theta(N^2)$

Iteration

1. Find Stopping Condition (Ex: x < N)
2. Find mode of iteration (Ex: i++)
3. Use 1 & 2 to determine the number of iterations in a loop

If the number of iterations isn't constant for "inner" loops, trying creating a table

→

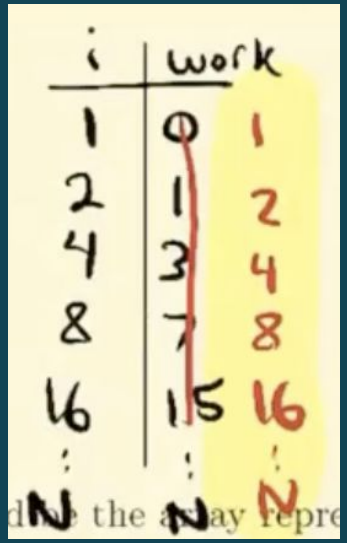
```
private void f1(int N) {
    for (int i = 1; i < N; i++) {
        for (int j = 1; j < i; j++) {
            System.out.println("hello tony");
        }
    }
}
Θ(___)
```

i	work
1	0
2	1
3	2
4	3
⋮	
N	N − 1

Total Number of Iterations: $\Theta(N^2)$ .... since $0 + 1 + 2 + 3 + \ldots + N - 1 = \Theta(N^2)$

Iteration (Extra Example)

```
private void f2(int N) {
    for (int i = 1; i < N; i *= 2) {
        for (int j = 1; j < i; j++) {
            System.out.println("hello hannah");
        }
    }
}
Θ(___)
```



Total Work = $\Theta(N)$ .... since $1 + 2 + 4 + \ldots + N/2 + N = \Theta(N)$

```java
public static void f4(int N) {
    if (N == 0) {return;}
    f4(N / 2);
    f4(N / 2);
    f4(N / 2);
    f4(N / 2);
    g(N); // runs in Θ(N^2) time
}
```

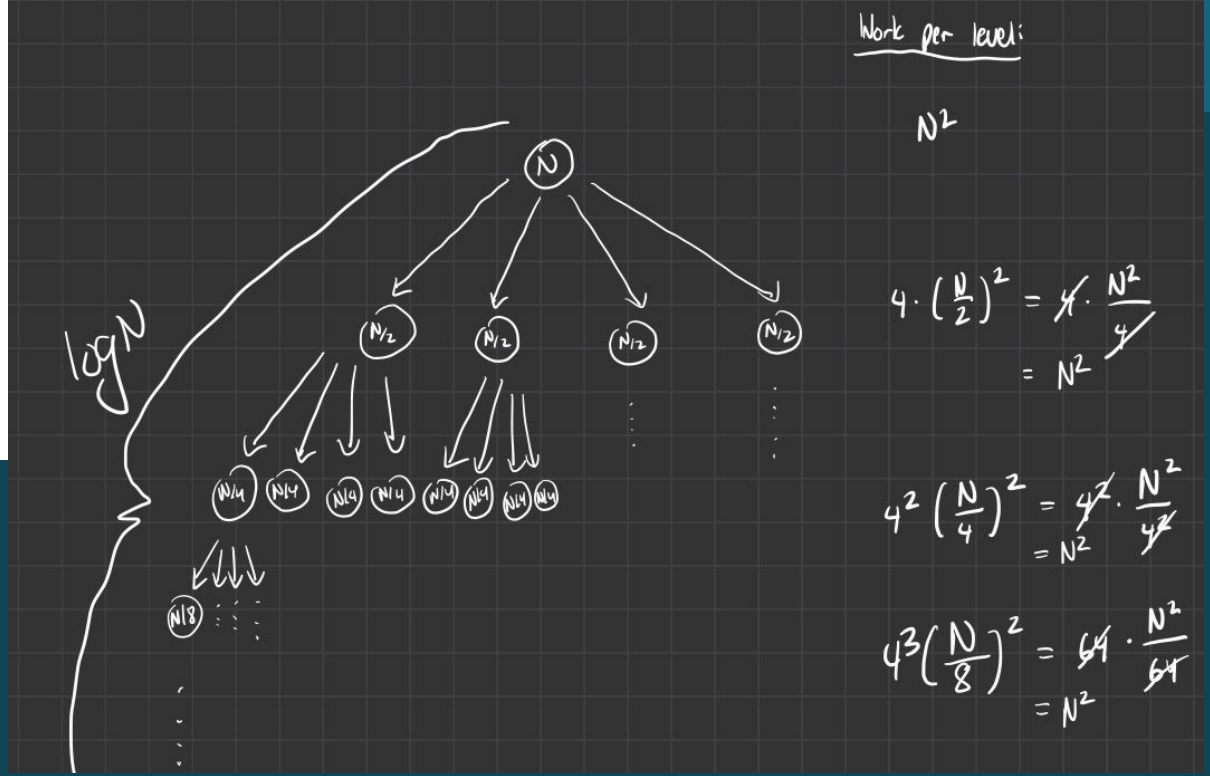
Runtime: $\Theta($ $)$

```java
public static void f5(int N, int M) {
    if (N < 10) {return;}
    for (int i = 0; i <= N % 10; i++) {
        f5(N / 10, M / 10);
        System.out.println(M);
    }
}
```

Runtime: $O($ $)$

```
1  public static void f4(int N) {
2      if (N == 0) {return;}
3      f4(N / 2);
4      f4(N / 2);
5      f4(N / 2);
6      f4(N / 2);
7      g(N); // runs in Θ(N^2) time
8  }
```

Runtime: $\Theta($ $)$



Work per level: $\Theta(N^2)$

# of levels: $\Theta(\log N)$

Final Runtime: $\Theta(N^2 \log N)$

```java
public static void f5(int N, int M) {
    if (N < 10) {return;}
    for (int i = 0; i <= N % 10; i++) {
        f5(N / 10, M / 10);
        System.out.println(M);
    }
}
```

Runtime: $O(\quad)$

We see a big-O bound,
find worst case!

Worst Case → For loop goes through 10 iterations
(i: 0 → 9)

Each call does constant work (print statement)

```
1  public static void f5(int N, int M) {
2      if (N < 10) {return;}
3      for (int i = 0; i <= N % 10; i++) {
4          f5(N / 10, M / 10);
5          System.out.println(M);
6      }
7  }
```

Runtime: $O(\quad)$

Fun with Methods

Method Overloading is done when there are multiple methods with the same name, but different parameters.

```
public void barkAt(Dog d) { System.out.print("Woof, it's another dog!"); }
public void barkAt(CS61BStaff s) { System.out.print("Woof, what is this?"); }
```

Method Overriding is done when a subclass has a method with the exact same function signature as a method in its superclass. It is usually marked with the `@Override` tag(ONLY FOR INSTANCE METHODS)

In Dog class:

```
public void speak() { System.out.print("Woof, I'm a dog!"); }
```

In Corgi Class, which inherits from Dog:

```
@Override
public void speak() { System.out.print("Woof, I'm a corgi!"); }
```

Static vs. Dynamic Type

A variable's static type is specified at declaration, whereas its dynamic type is specified at instantiation (e.g. when using new).

```
Dog d = new Corgi();
```

The static and dynamic type of a variable have to complement each other or else the code will error. For example, a `Dog` is not necessarily a `Corgi`, so `Corgi c = new Dog();` will not compile.

General rule of thumb: Given LHS = RHS, is RHS guaranteed to be a LHS?

NOTE: Though interfaces cannot be instantiated, they can be static types

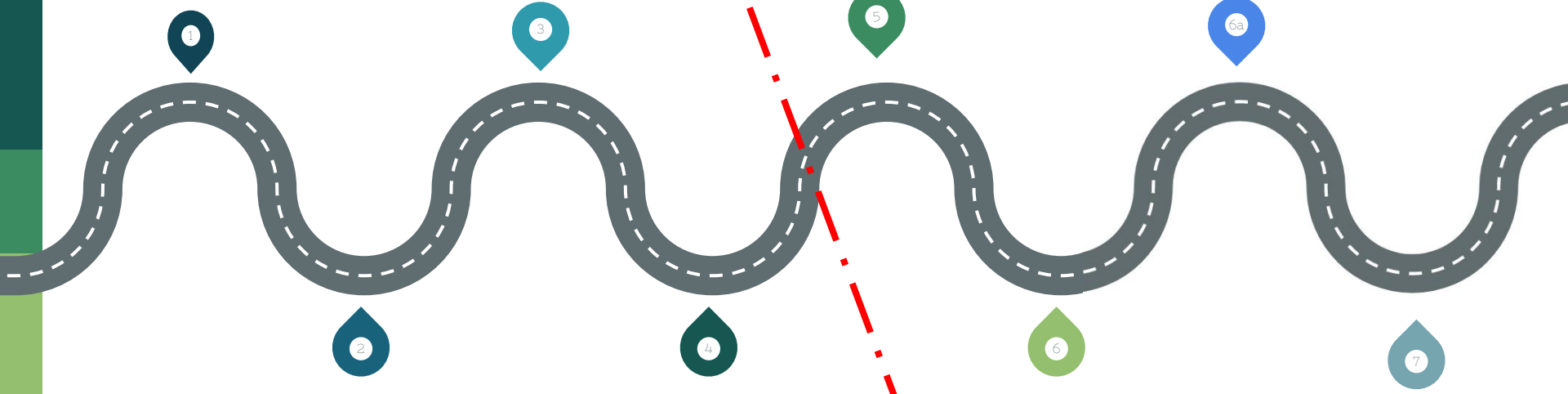
DMS Roadmap

COMPILE TIME → Look at Static Type!

1. Check for valid variable assignments (Does it make sense — Animal A = new Car()
2. Check for valid method calls (look @ static type and superclass(es))
3. Lock in exact method signature while traversing parent classes
4. If nothing found → CE
5. If static method, run it & skip later steps

RUN TIME → Look at Dynamic Type!

6. Check for overridden methods
6a. Does the locked-in method signature have an identical one in the dynamic class or the dynamic class's parent classes?
7. Ensure casted objects can be assigned to their variables



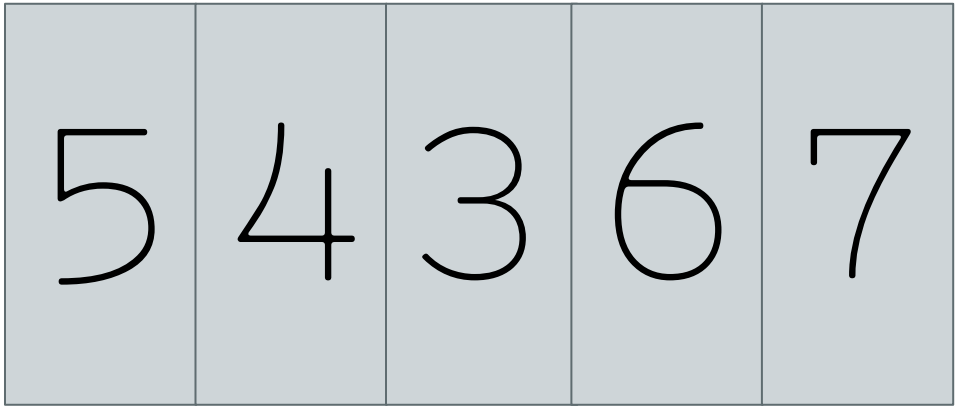
Final Review Session: Sorting

CS61B Data Structures

Let’s “sort” out a few things about sorting...

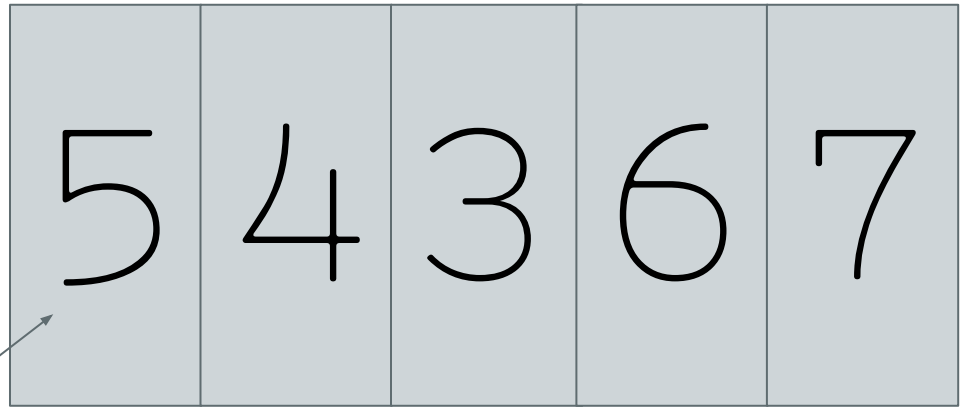
Insertion Sort

- sorts an array by iteratively placing each element in its proper position, one at a time, starting with the first element.
- Uses “swaps” until the element at the current iteration is in the correct position.
- Runtime: $O(N^2)$



Insertion Sort

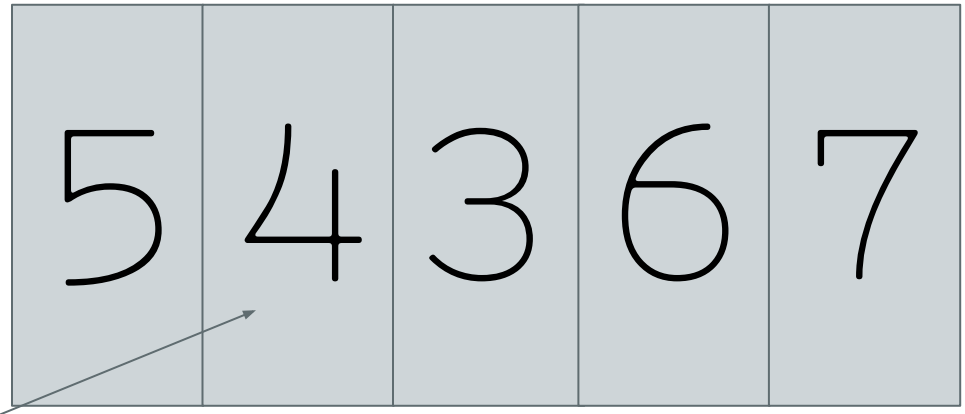
- sorts an array by iteratively placing each element in its proper position, one at a time, starting with the first element.
- Uses “swaps” until the element at the current iteration is in the correct position.
- Runtime: $O(N^2)$



Can’t swap left!

Insertion Sort

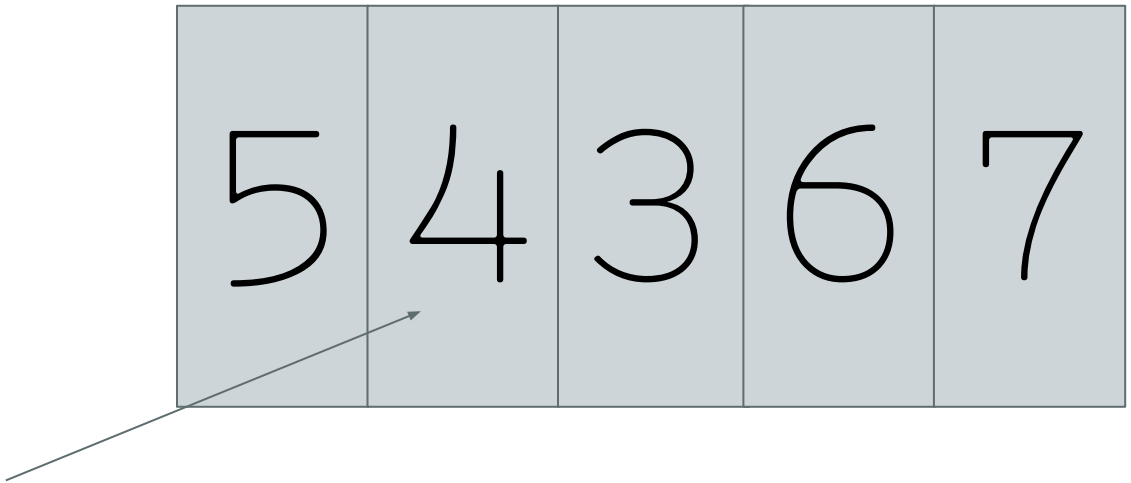
- sorts an array by iteratively placing each element in its proper position, one at a time, starting with the first element.
- Uses “swaps” until the element at the current iteration is in the correct position.
- Runtime: $O(N^2)$



Moved pointer right

Insertion Sort

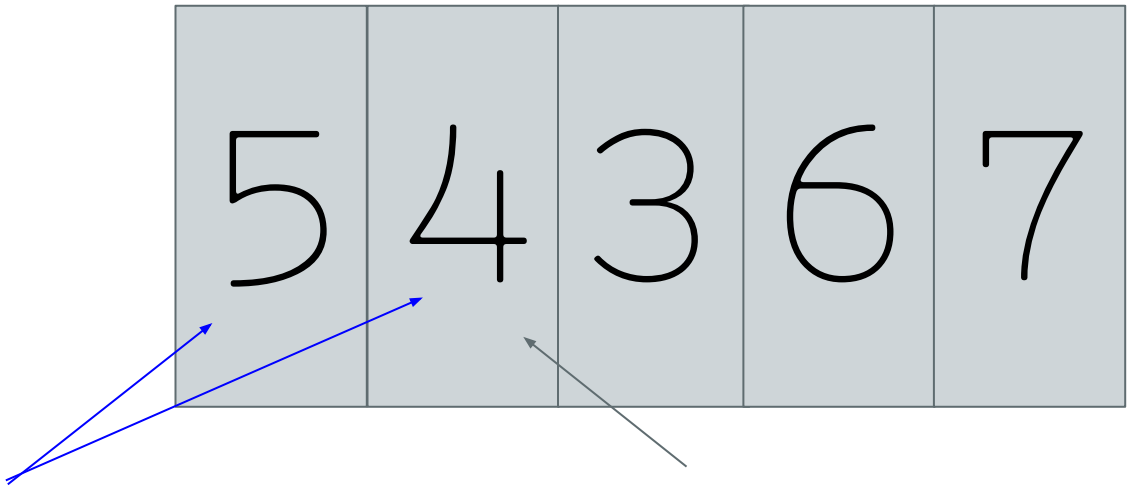
- sorts an array by iteratively placing each element in its proper position, one at a time, starting with the first element.
- Uses “swaps” until the element at the current iteration is in the correct position.
- Runtime: $O(N^2)$



4 is not in the correct position because it is greater than 5!!

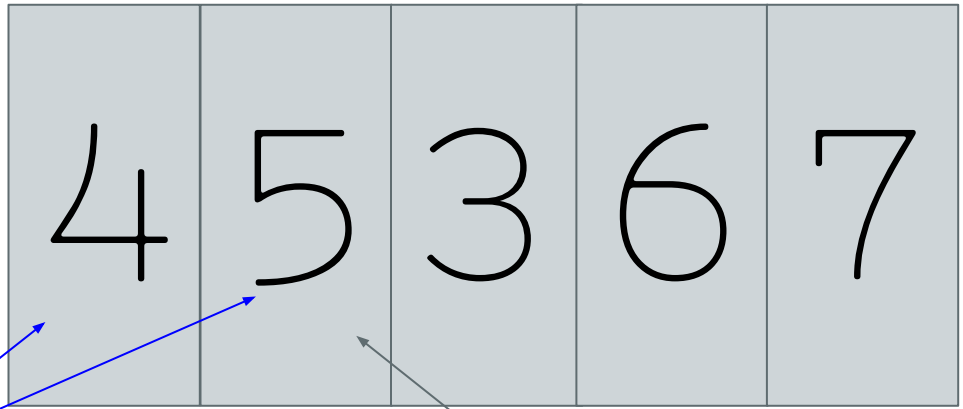
Insertion Sort

- sorts an array by iteratively placing each element in its proper position, one at a time, starting with the first element.
- Uses “swaps” until the element at the current iteration is in the correct position.
- Runtime: $O(N^2)$



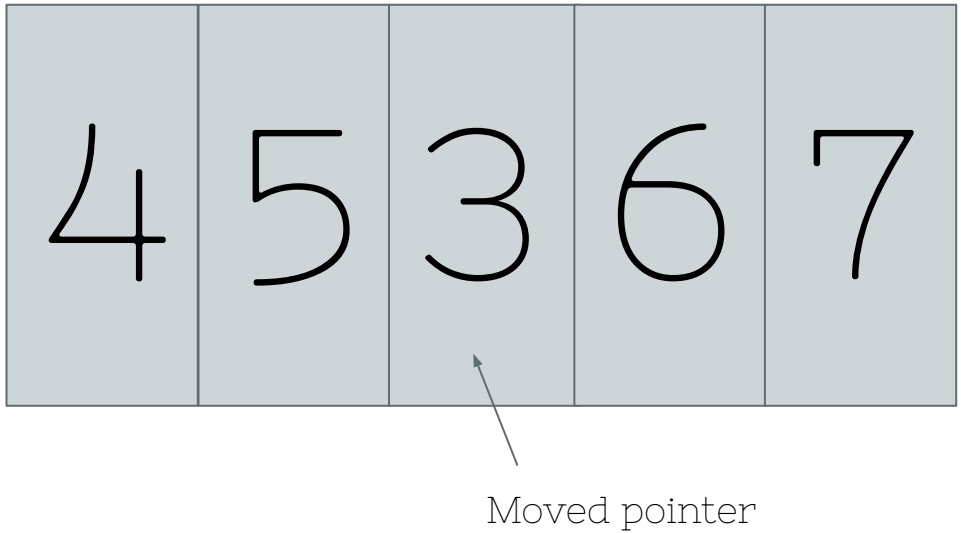
Insertion Sort

- sorts an array by iteratively placing each element in its proper position, one at a time, starting with the first element.
- Uses “swaps” until the element at the current iteration is in the correct position.
- Runtime: $O(N^2)$



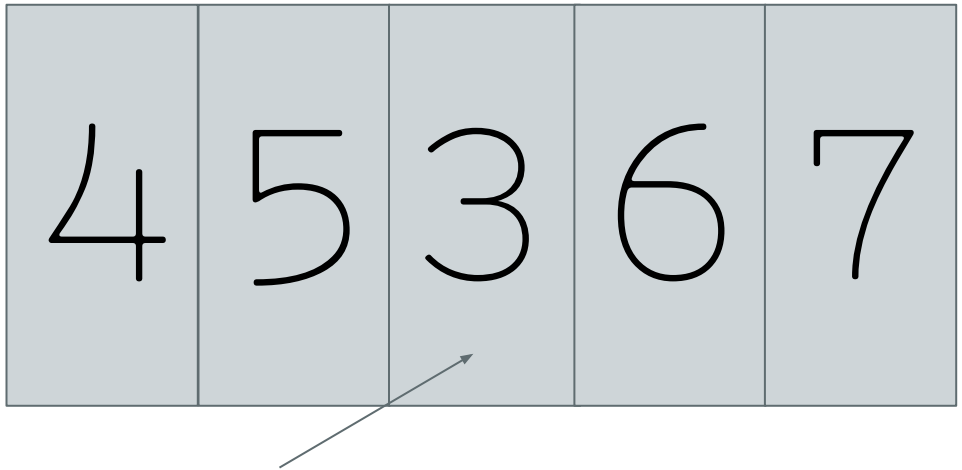
Insertion Sort

- sorts an array by iteratively placing each element in its proper position, one at a time, starting with the first element.
- Uses “swaps” until the element at the current iteration is in the correct position.
- Runtime: $O(N^2)$



Insertion Sort

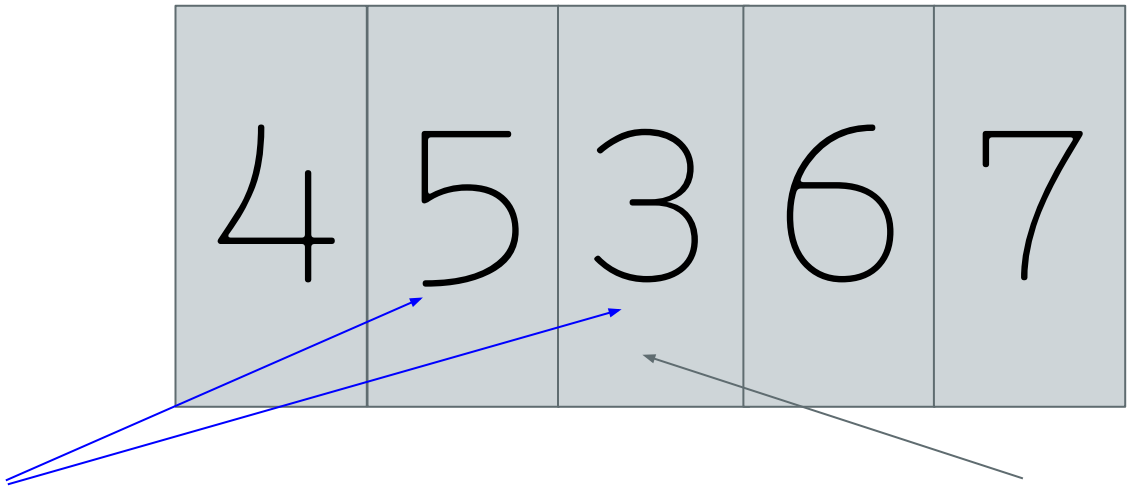
- sorts an array by iteratively placing each element in its proper position, one at a time, starting with the first element.
- Uses “swaps” until the element at the current iteration is in the correct position.
- Runtime: $O(N^2)$



3 is not in the correct position because it is greater than 5!!

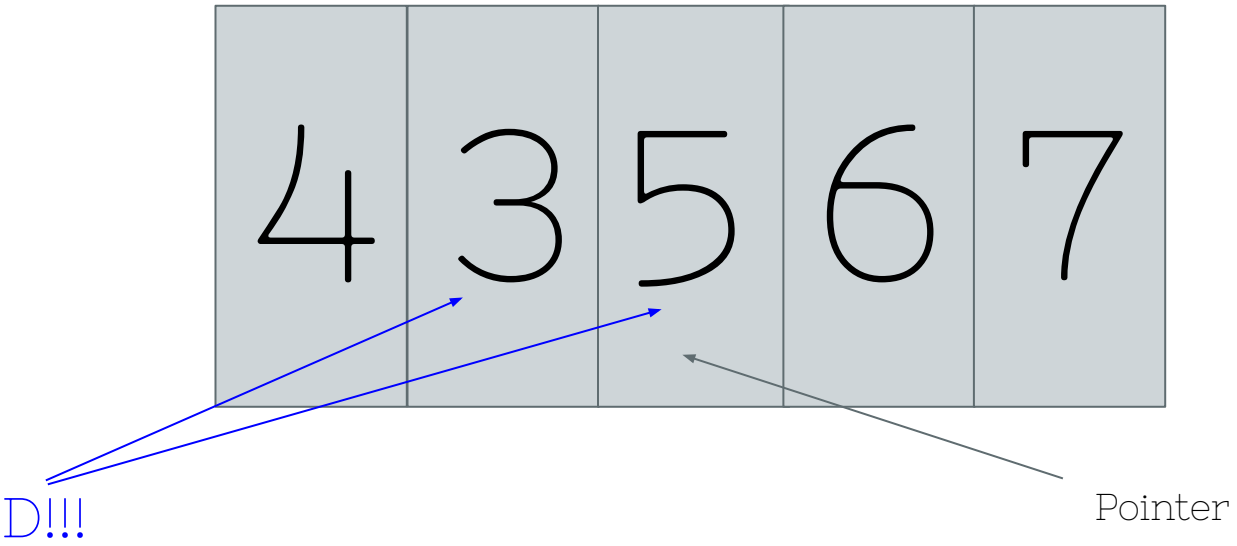
Insertion Sort

- sorts an array by iteratively placing each element in its proper position, one at a time, starting with the first element.
- Uses “swaps” until the element at the current iteration is in the correct position.
- Runtime: $O(N^2)$



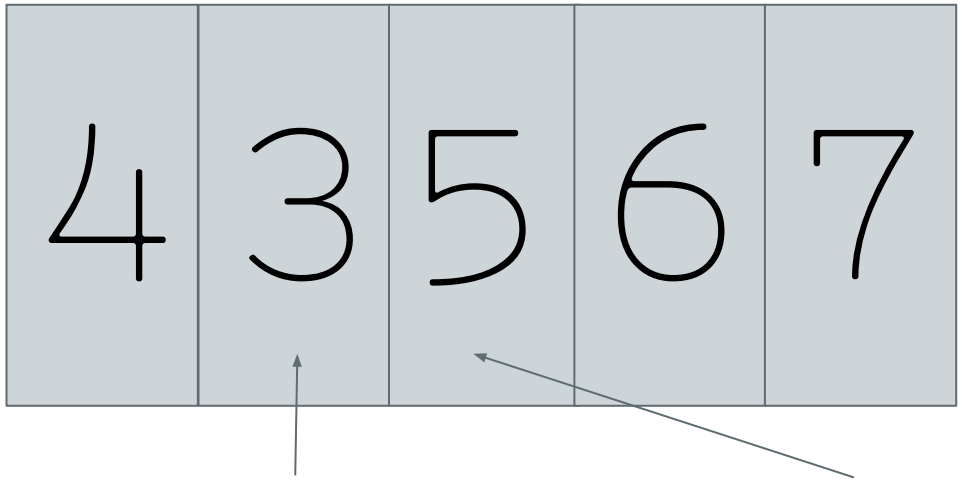
Insertion Sort

- sorts an array by iteratively placing each element in its proper position, one at a time, starting with the first element.
- Uses “swaps” until the element at the current iteration is in the correct position.
- Runtime: $O(N^2)$



Insertion Sort

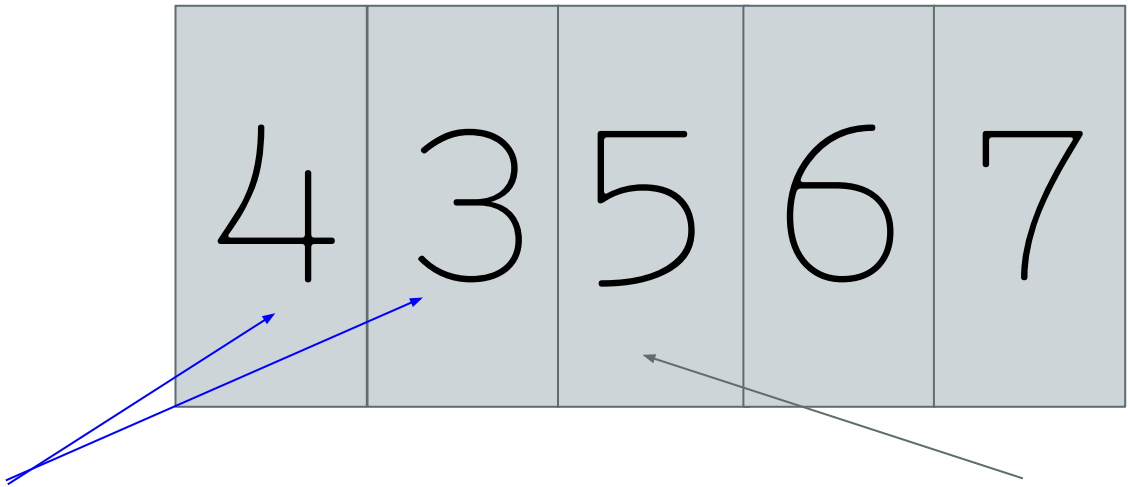
- sorts an array by iteratively placing each element in its proper position, one at a time, starting with the first element.
- Uses “swaps” until the element at the current iteration is in the correct position.
- Runtime: $O(N^2)$



3 is not in the correct position because it is greater than 4!!

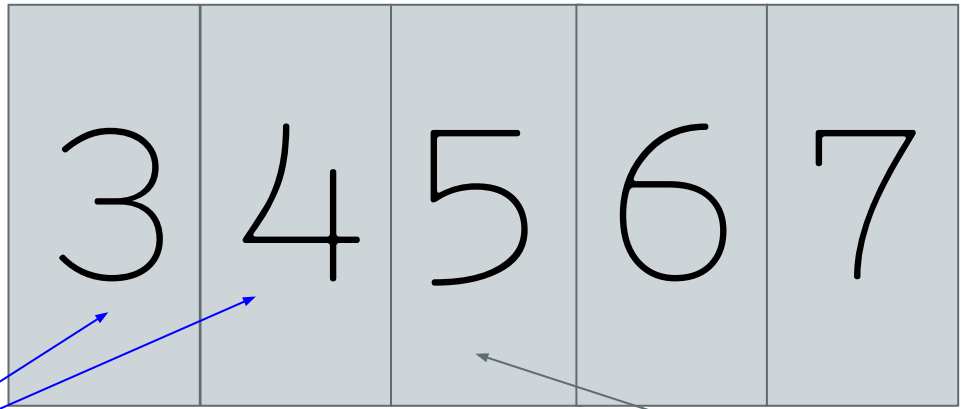
Insertion Sort

- sorts an array by iteratively placing each element in its proper position, one at a time, starting with the first element.
- Uses “swaps” until the element at the current iteration is in the correct position.
- Runtime: $O(N^2)$



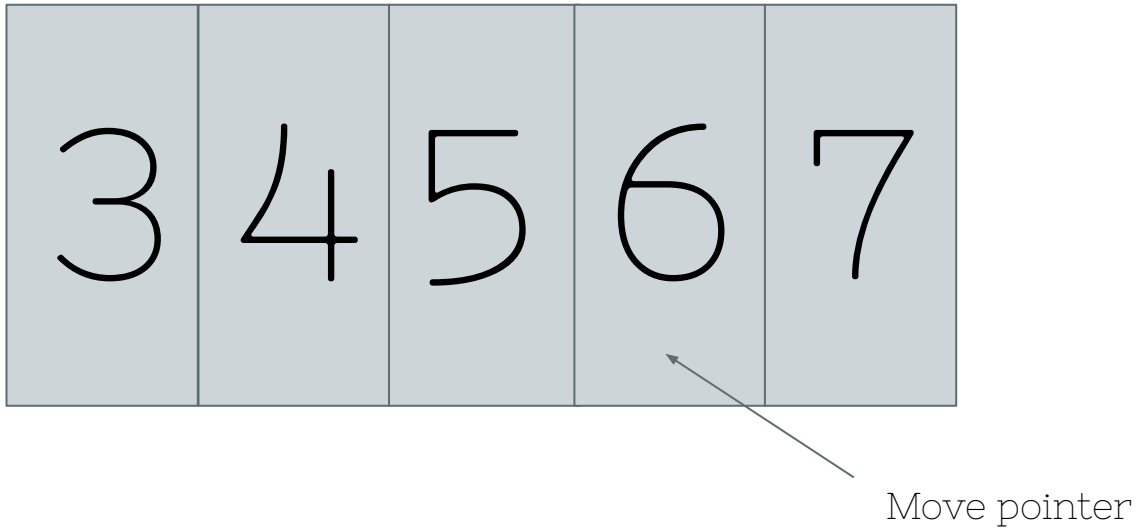
Insertion Sort

- sorts an array by iteratively placing each element in its proper position, one at a time, starting with the first element.
- Uses “swaps” until the element at the current iteration is in the correct position.
- Runtime: $O(N^2)$



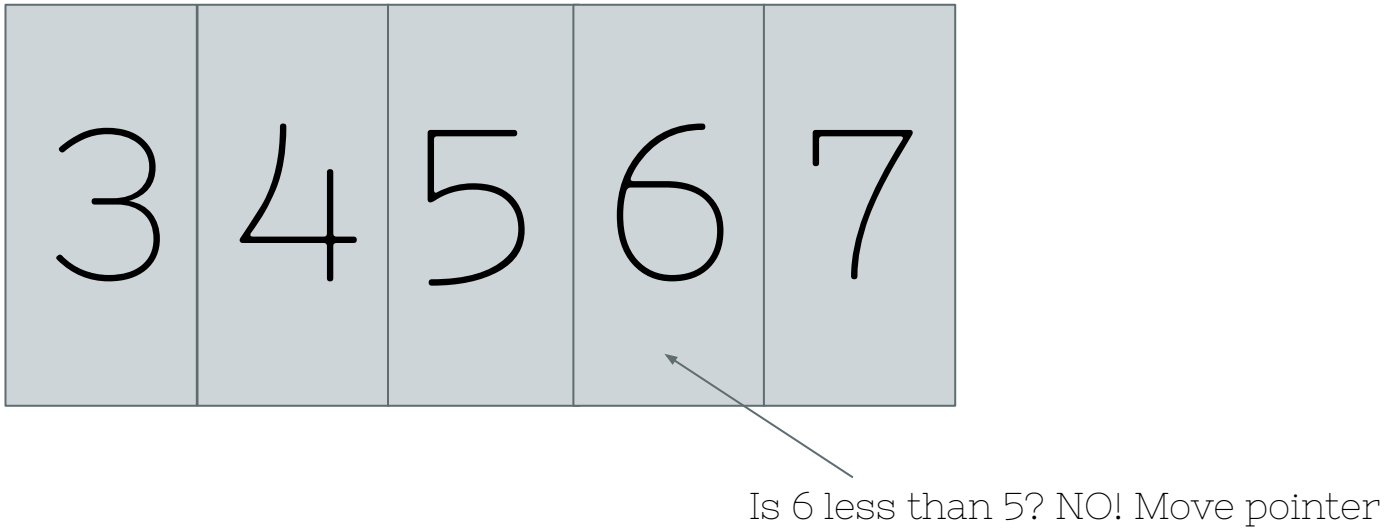
Insertion Sort

- sorts an array by iteratively placing each element in its proper position, one at a time, starting with the first element.
- Uses “swaps” until the element at the current iteration is in the correct position.
- Runtime: $O(N^2)$



Insertion Sort

- sorts an array by iteratively placing each element in its proper position, one at a time, starting with the first element.
- Uses “swaps” until the element at the current iteration is in the correct position.
- Runtime: $O(N^2)$

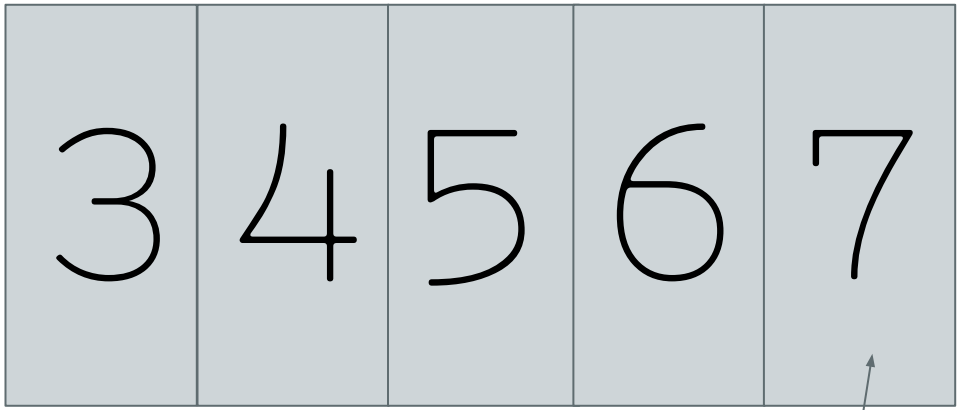


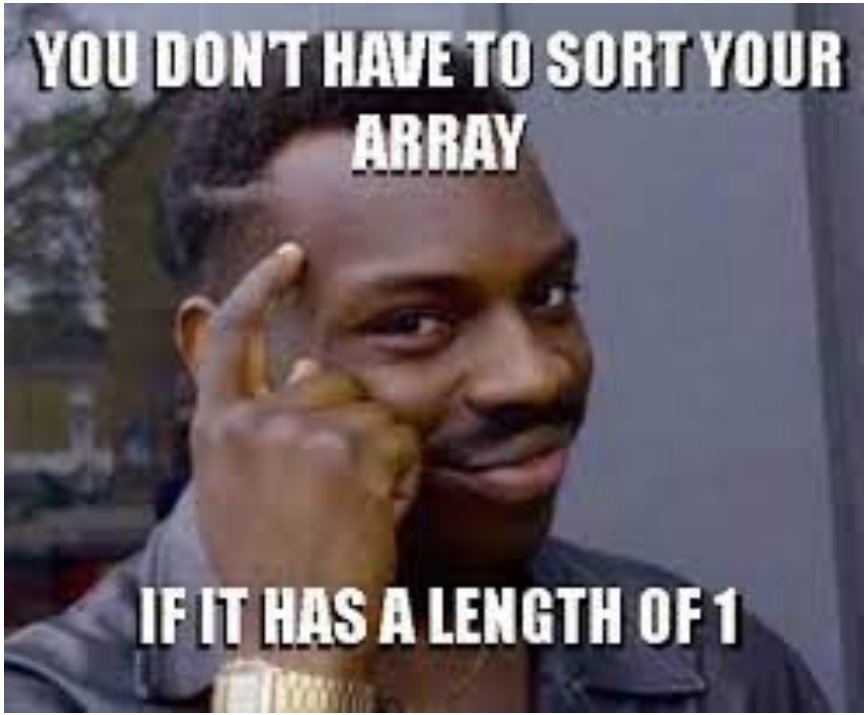
Insertion Sort

- sorts an array by iteratively placing each element in its proper position, one at a time, starting with the first element.
- Uses “swaps” until the element at the current iteration is in the correct position.
- Runtime: $O(N^2)$

Insertion Sort

- sorts an array by iteratively placing each element in its proper position, one at a time, starting with the first element.
- Uses “swaps” until the element at the current iteration is in the correct position.
- Runtime: $O(N^2)$

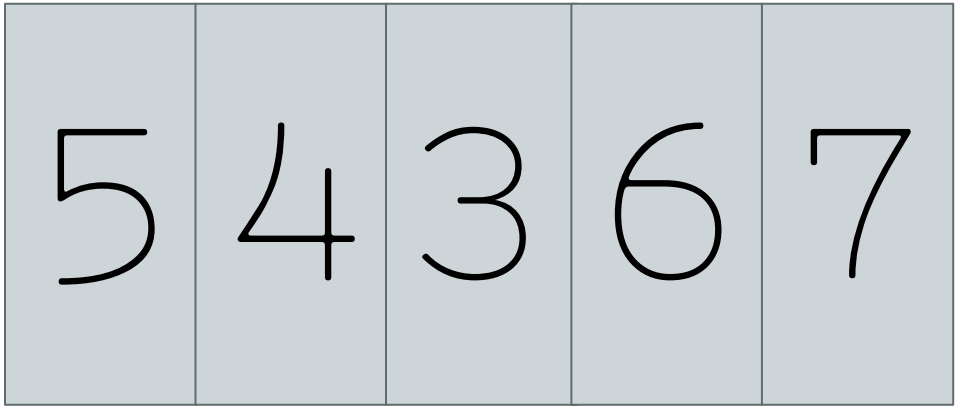




DONE WITH INSERTION SORT

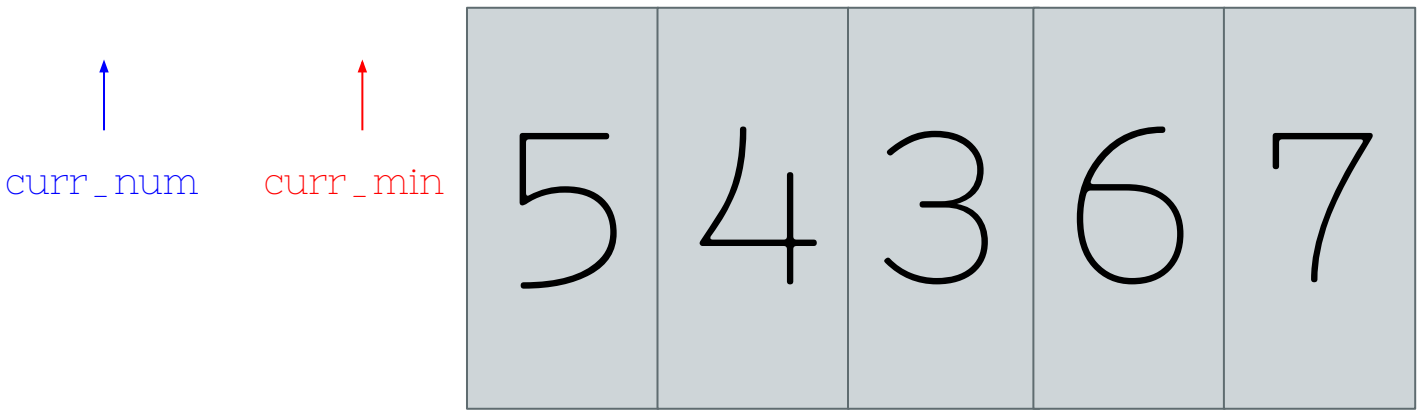
Selection Sort

- sorts an array or a list of items by repeatedly finding the minimum element (or maximum element) from the unsorted part of the list and placing it at the beginning of the sorted part.
- Runtime: $\Theta(N^2)$



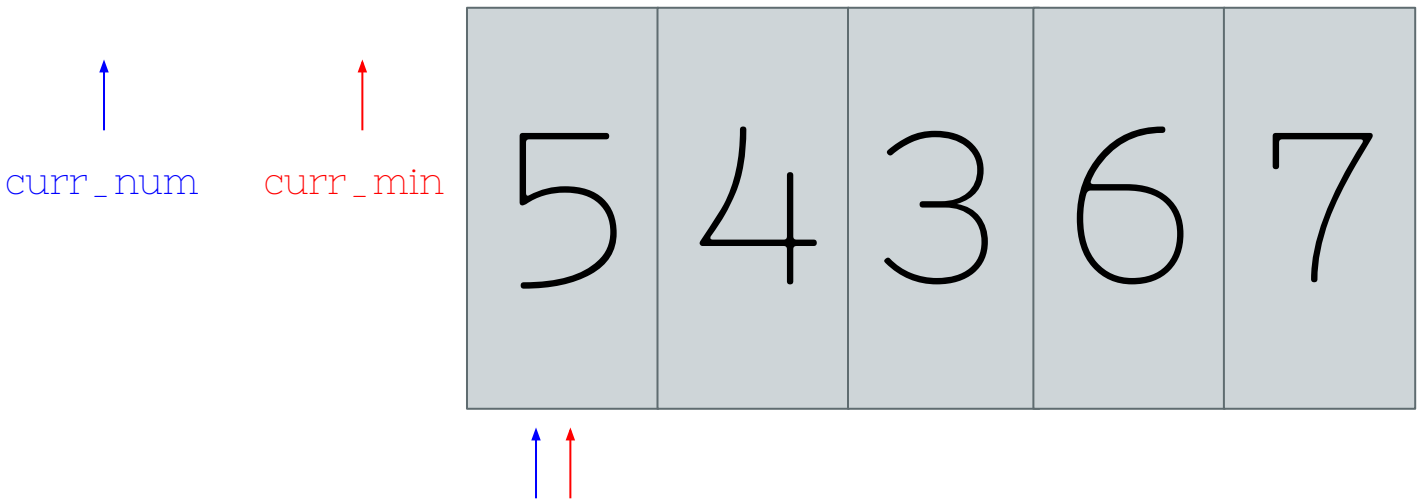
Selection Sort

- sorts an array or a list of items by repeatedly finding the minimum element (or maximum element) from the unsorted part of the list and placing it at the beginning of the sorted part.
- Runtime: $\Theta(N^2)$



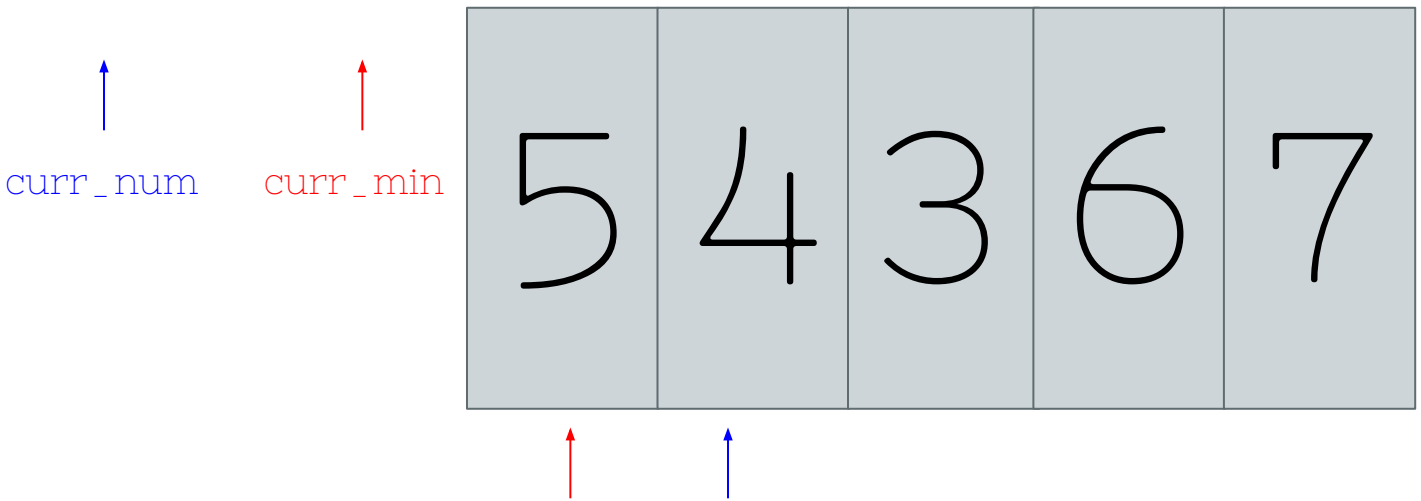
Selection Sort

- sorts an array or a list of items by repeatedly finding the minimum element (or maximum element) from the unsorted part of the list and placing it at the beginning of the sorted part.
- Runtime: **Θ**(N^2)



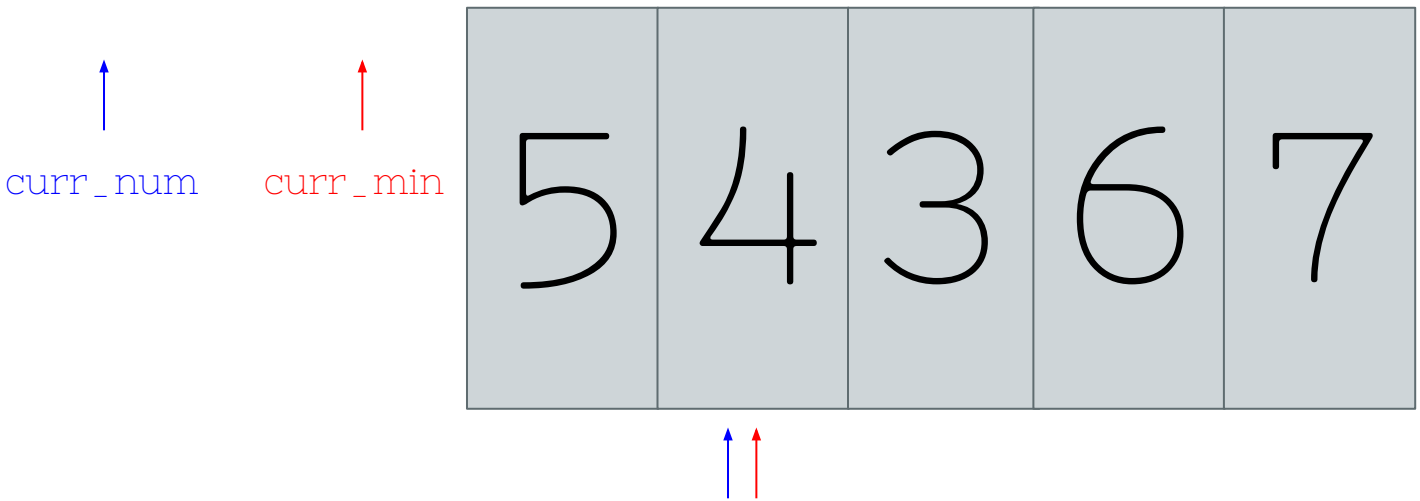
Selection Sort

- sorts an array or a list of items by repeatedly finding the minimum element (or maximum element) from the unsorted part of the list and placing it at the beginning of the sorted part.
- Runtime: $\Theta(N^2)$



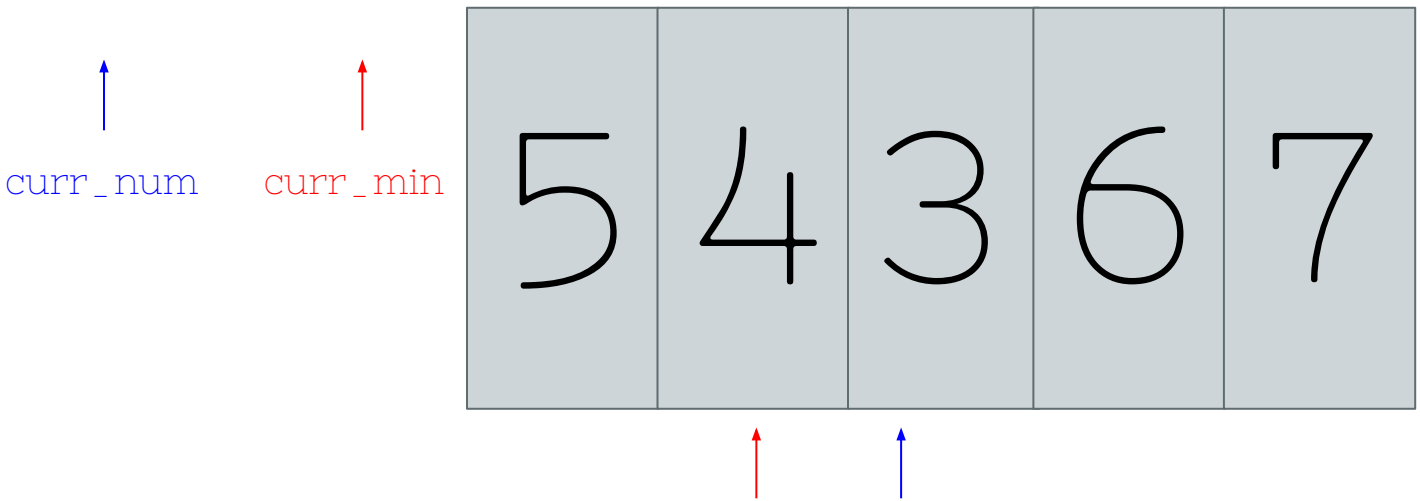
Selection Sort

- sorts an array or a list of items by repeatedly finding the minimum element (or maximum element) from the unsorted part of the list and placing it at the beginning of the sorted part.
- Runtime: $\Theta(N^2)$



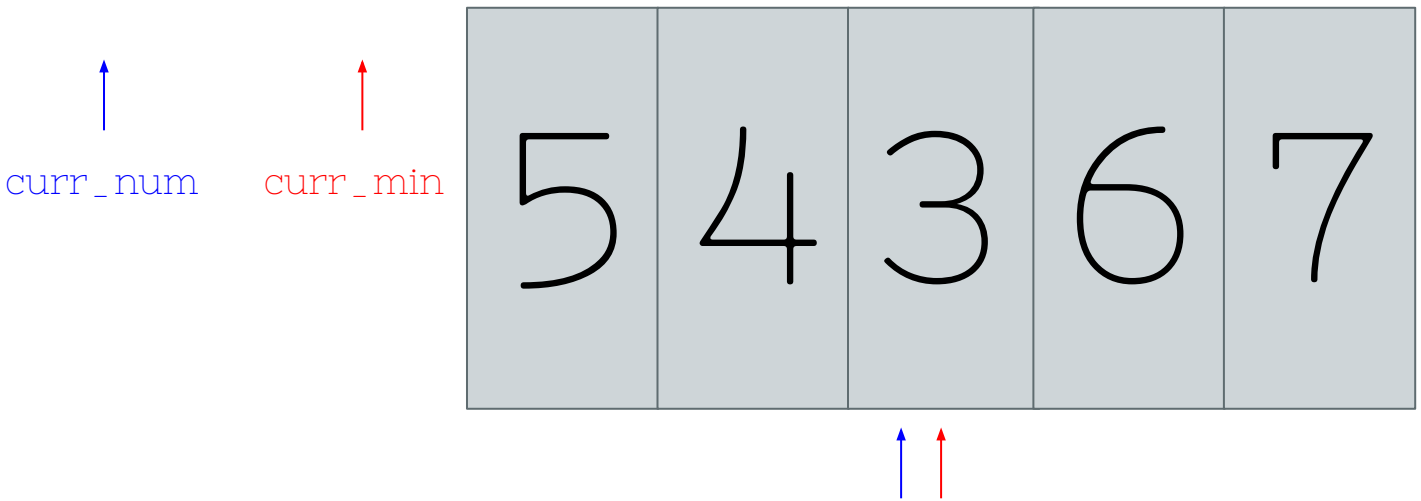
Selection Sort

- sorts an array or a list of items by repeatedly finding the minimum element (or maximum element) from the unsorted part of the list and placing it at the beginning of the sorted part.
- Runtime: **Θ**(N^2)



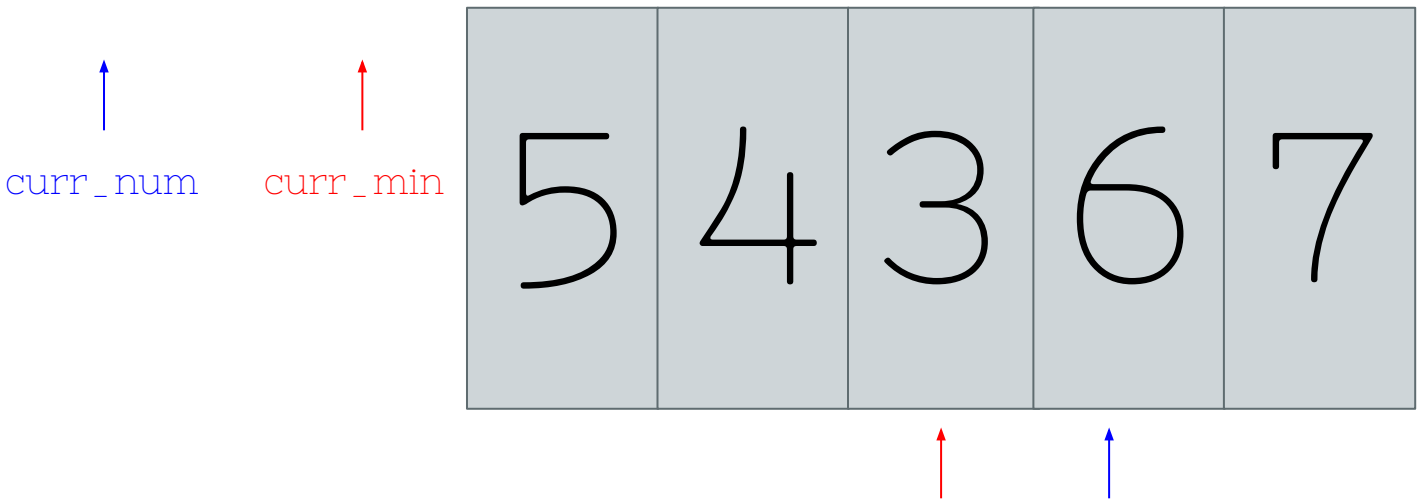
Selection Sort

- sorts an array or a list of items by repeatedly finding the minimum element (or maximum element) from the unsorted part of the list and placing it at the beginning of the sorted part.
- Runtime: **Θ**(N^2)



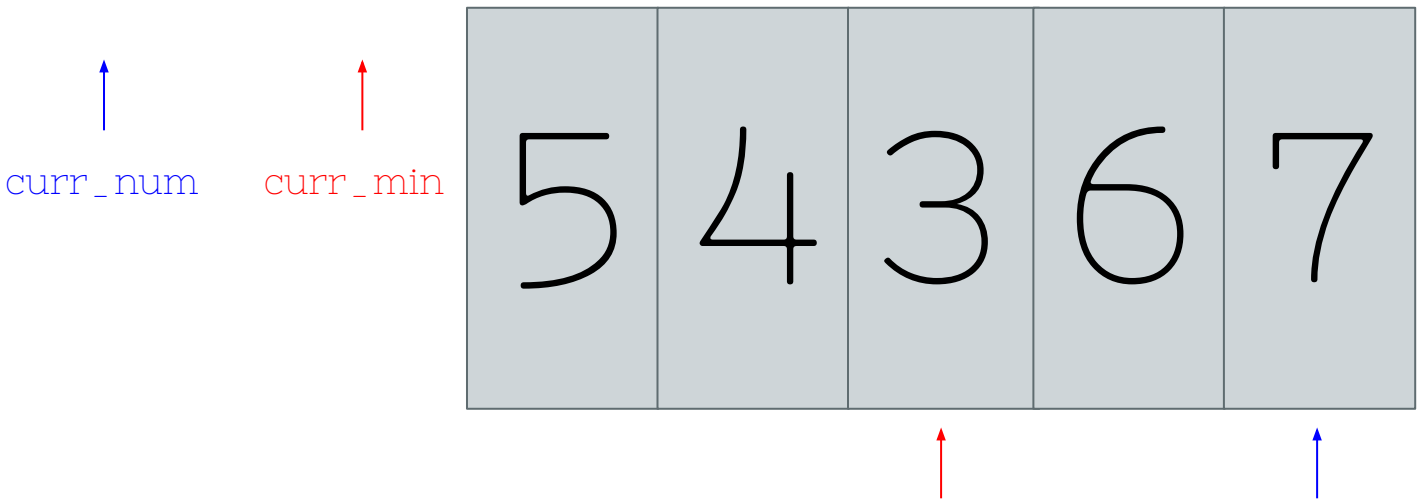
Selection Sort

- sorts an array or a list of items by repeatedly finding the minimum element (or maximum element) from the unsorted part of the list and placing it at the beginning of the sorted part.
- Runtime: $\Theta(N^2)$



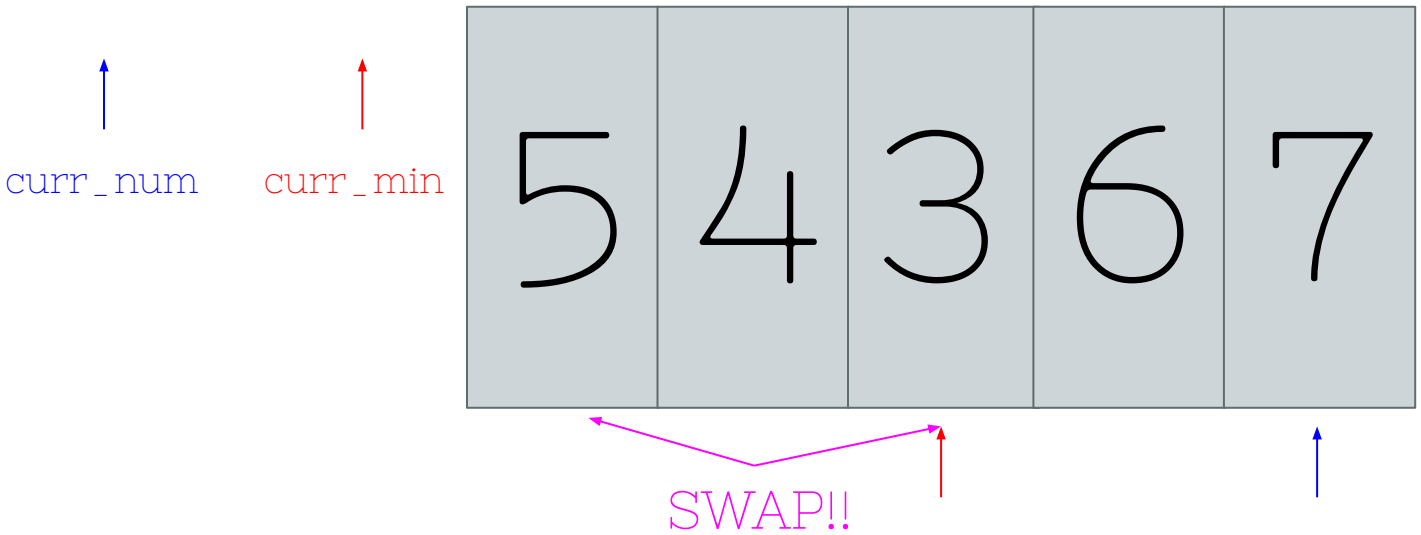
Selection Sort

- sorts an array or a list of items by repeatedly finding the minimum element (or maximum element) from the unsorted part of the list and placing it at the beginning of the sorted part.
- Runtime: **Θ**(N^2)



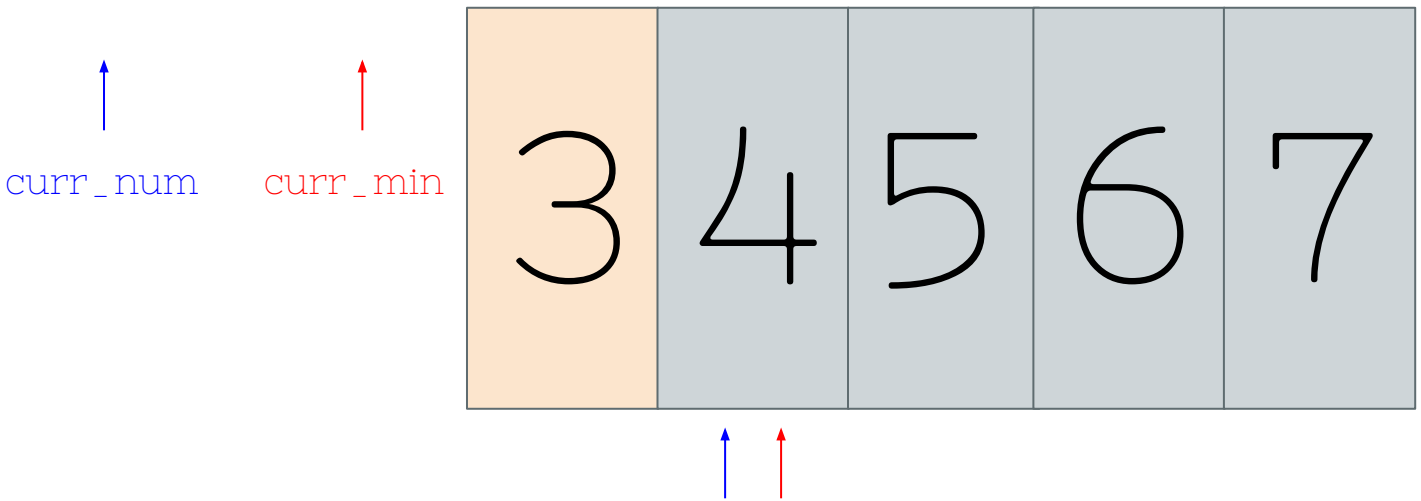
Selection Sort

- sorts an array or a list of items by repeatedly finding the minimum element (or maximum element) from the unsorted part of the list and placing it at the beginning of the sorted part.
- Runtime: **Θ**(N^2)



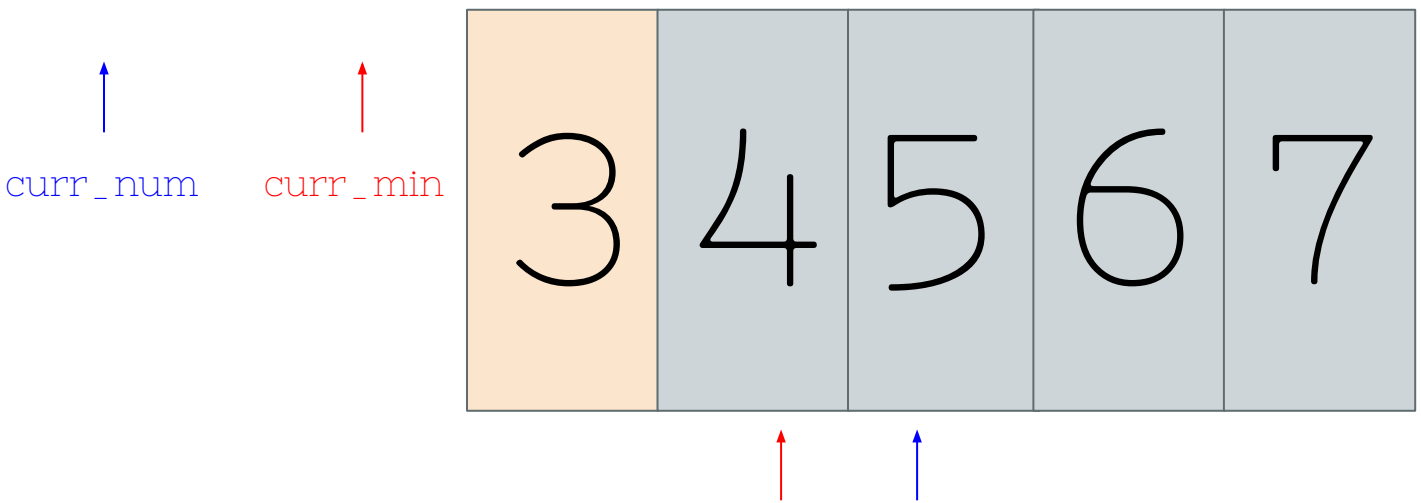
Selection Sort

- sorts an array or a list of items by repeatedly finding the minimum element (or maximum element) from the unsorted part of the list and placing it at the beginning of the sorted part.
- Runtime: **Θ**(N^2)



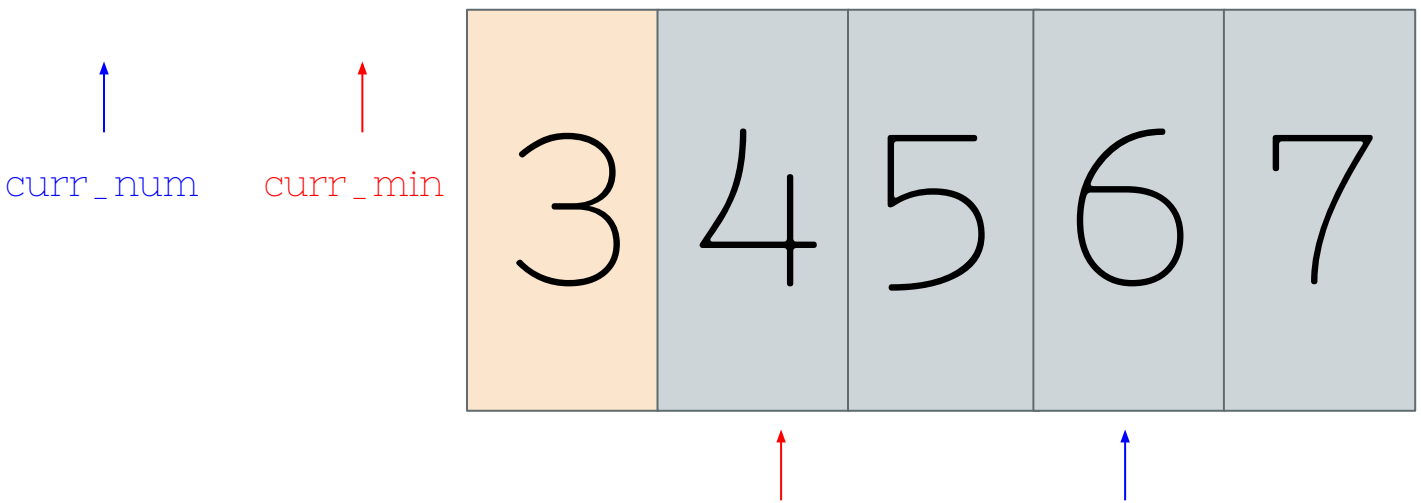
Selection Sort

- sorts an array or a list of items by repeatedly finding the minimum element (or maximum element) from the unsorted part of the list and placing it at the beginning of the sorted part.
- Runtime: $\Theta(N^2)$



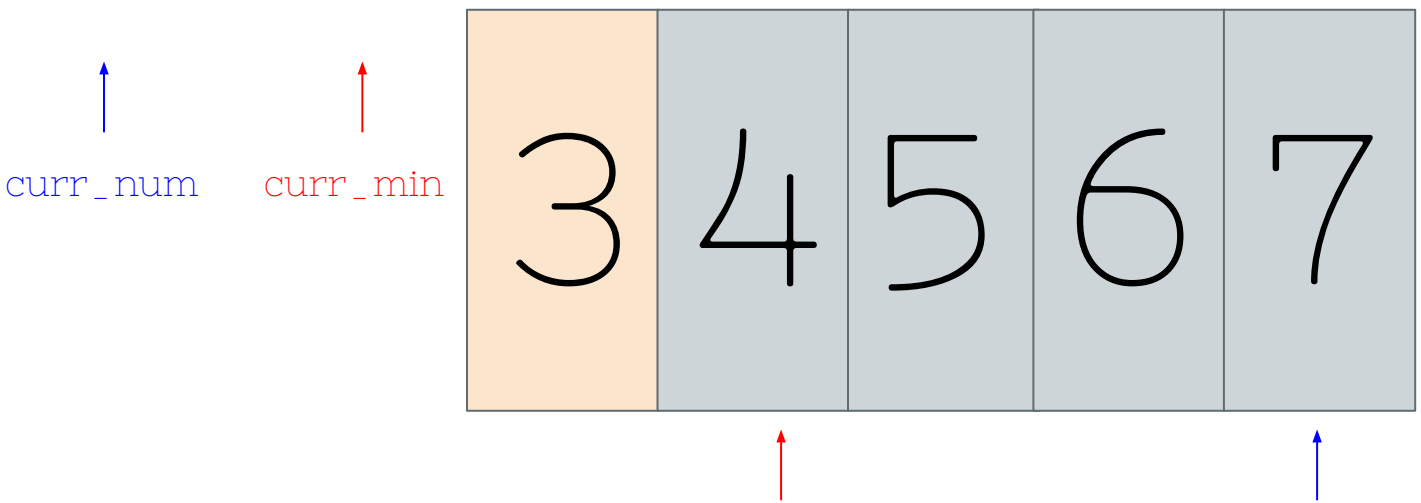
Selection Sort

- sorts an array or a list of items by repeatedly finding the minimum element (or maximum element) from the unsorted part of the list and placing it at the beginning of the sorted part.
- Runtime: $\Theta(N^2)$



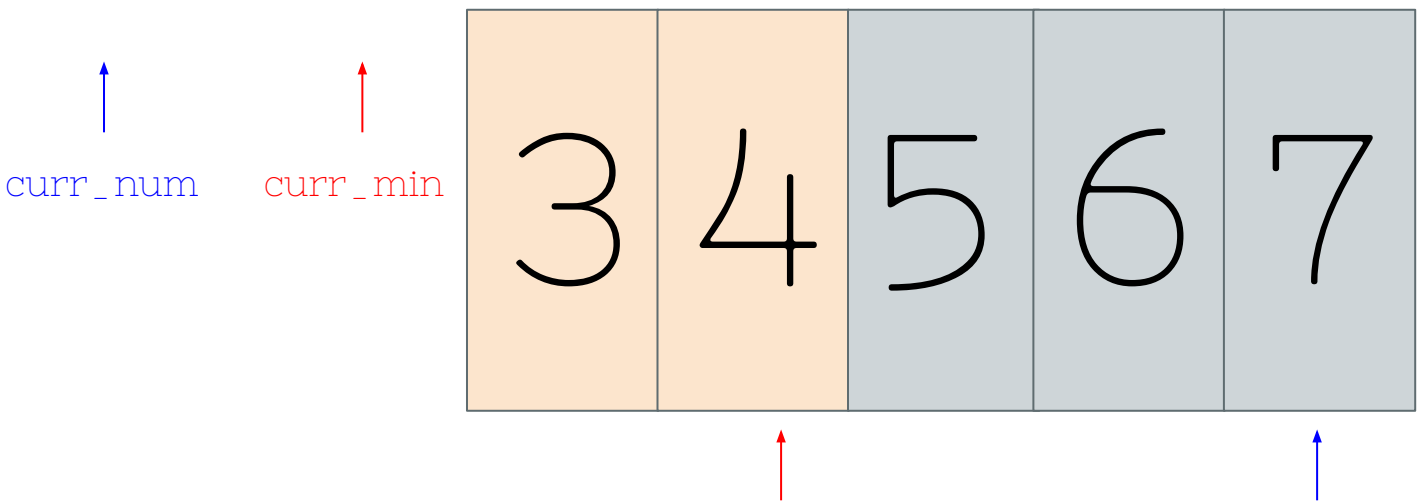
Selection Sort

- sorts an array or a list of items by repeatedly finding the minimum element (or maximum element) from the unsorted part of the list and placing it at the beginning of the sorted part.
- Runtime: $\Theta(N^2)$



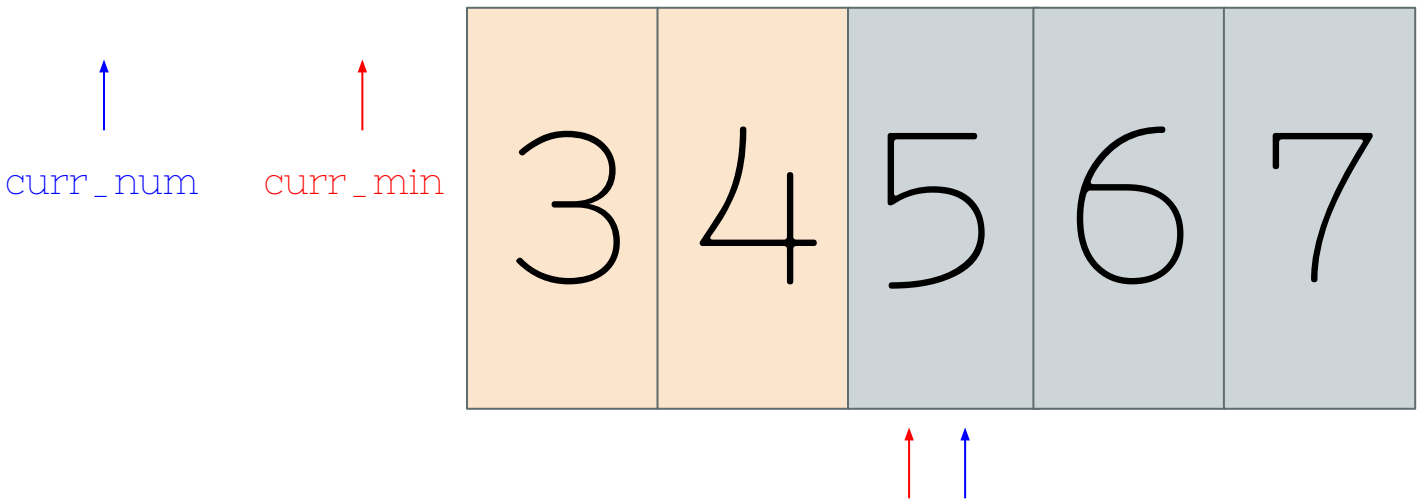
Selection Sort

- sorts an array or a list of items by repeatedly finding the minimum element (or maximum element) from the unsorted part of the list and placing it at the beginning of the sorted part.
- Runtime: **Θ**(N^2)



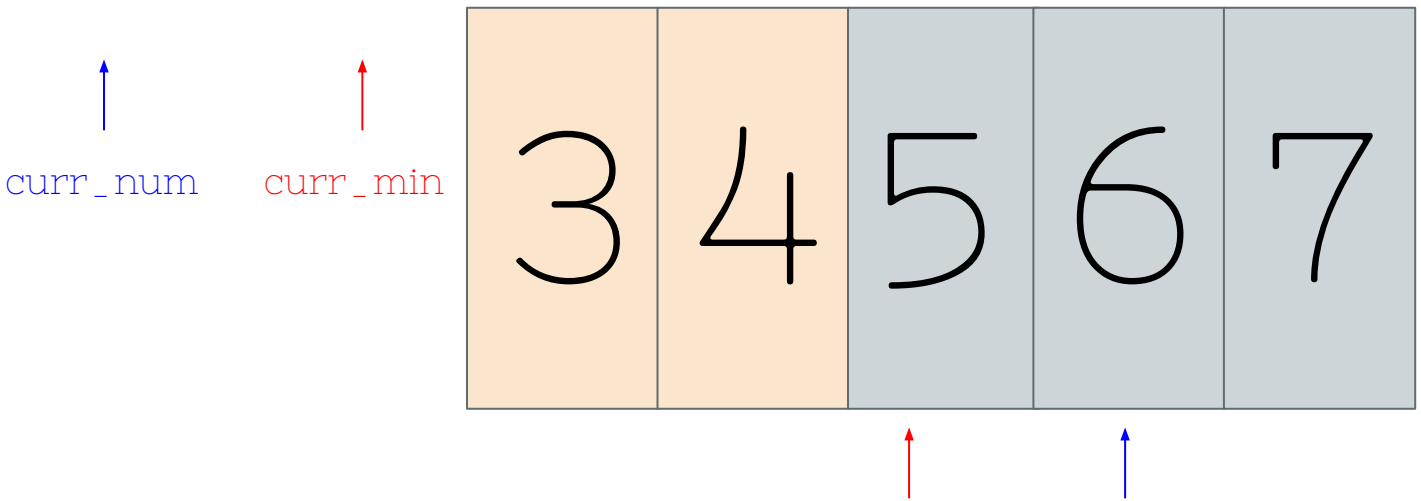
Selection Sort

- sorts an array or a list of items by repeatedly finding the minimum element (or maximum element) from the unsorted part of the list and placing it at the beginning of the sorted part.
- Runtime: **Θ**(N^2)



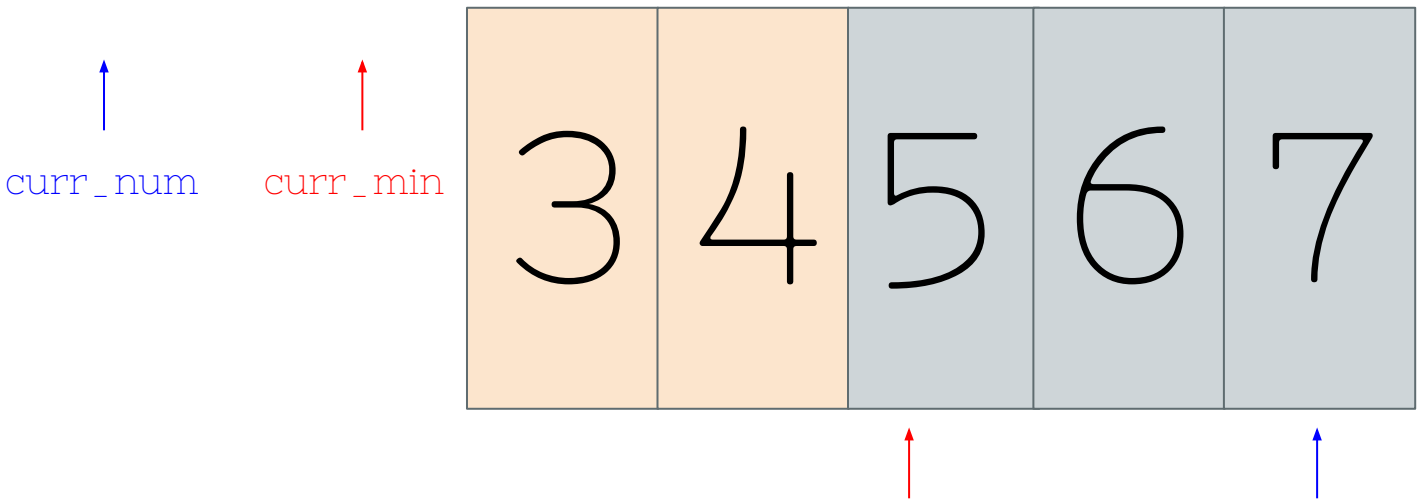
Selection Sort

- sorts an array or a list of items by repeatedly finding the minimum element (or maximum element) from the unsorted part of the list and placing it at the beginning of the sorted part.
- Runtime: **Θ**(N^2)



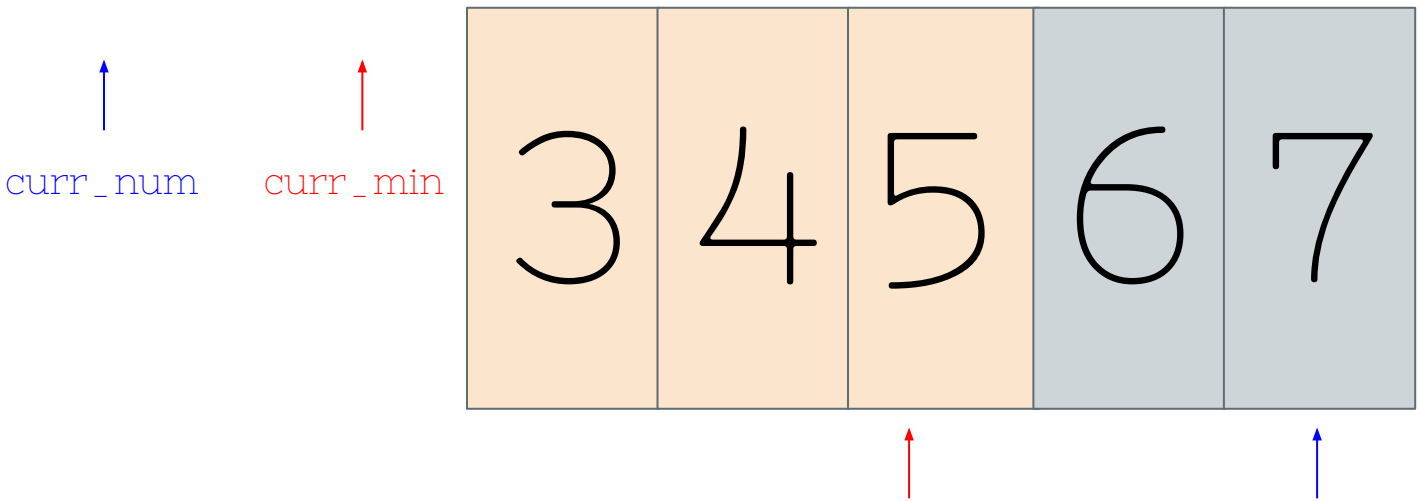
Selection Sort

- sorts an array or a list of items by repeatedly finding the minimum element (or maximum element) from the unsorted part of the list and placing it at the beginning of the sorted part.
- Runtime: $\Theta(N^2)$



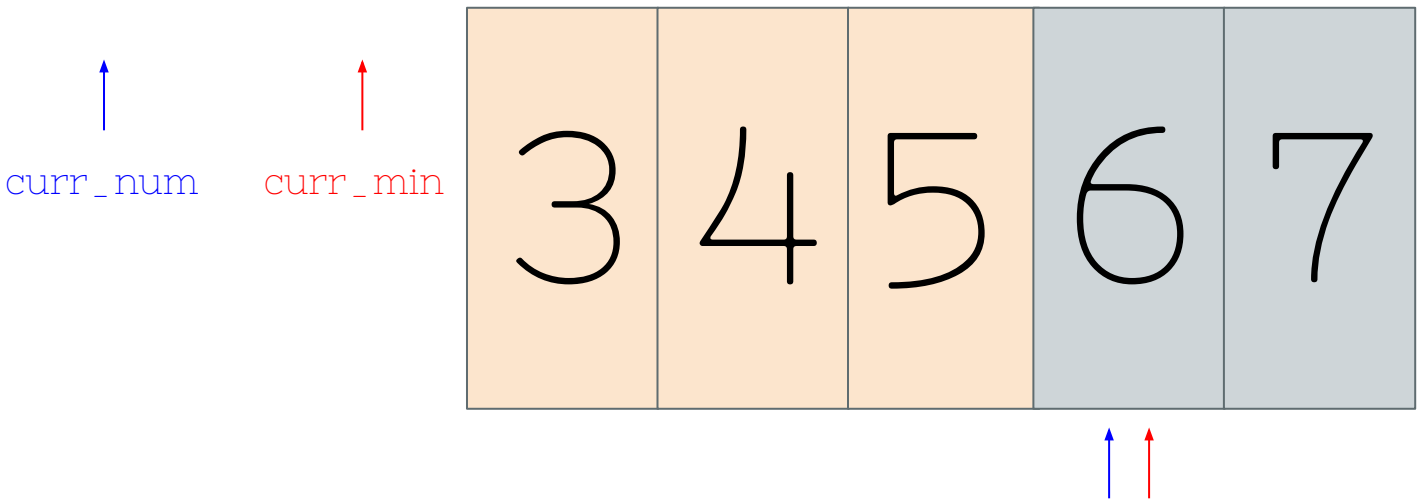
Selection Sort

- sorts an array or a list of items by repeatedly finding the minimum element (or maximum element) from the unsorted part of the list and placing it at the beginning of the sorted part.
- Runtime: **Θ**(N^2)



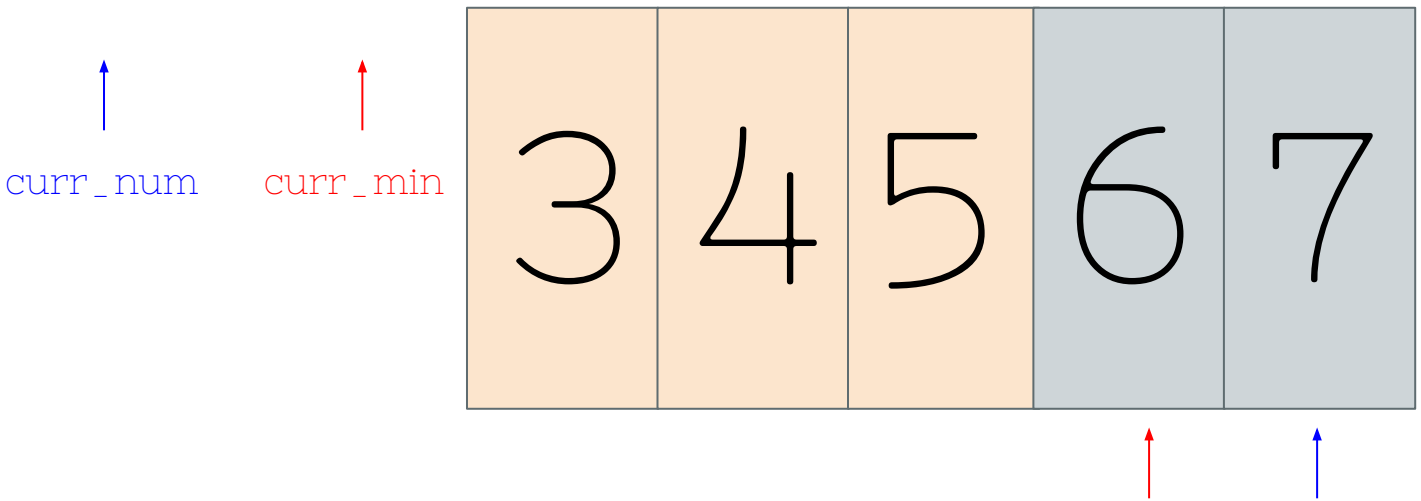
Selection Sort

- sorts an array or a list of items by repeatedly finding the minimum element (or maximum element) from the unsorted part of the list and placing it at the beginning of the sorted part.
- Runtime: $\Theta(N^2)$



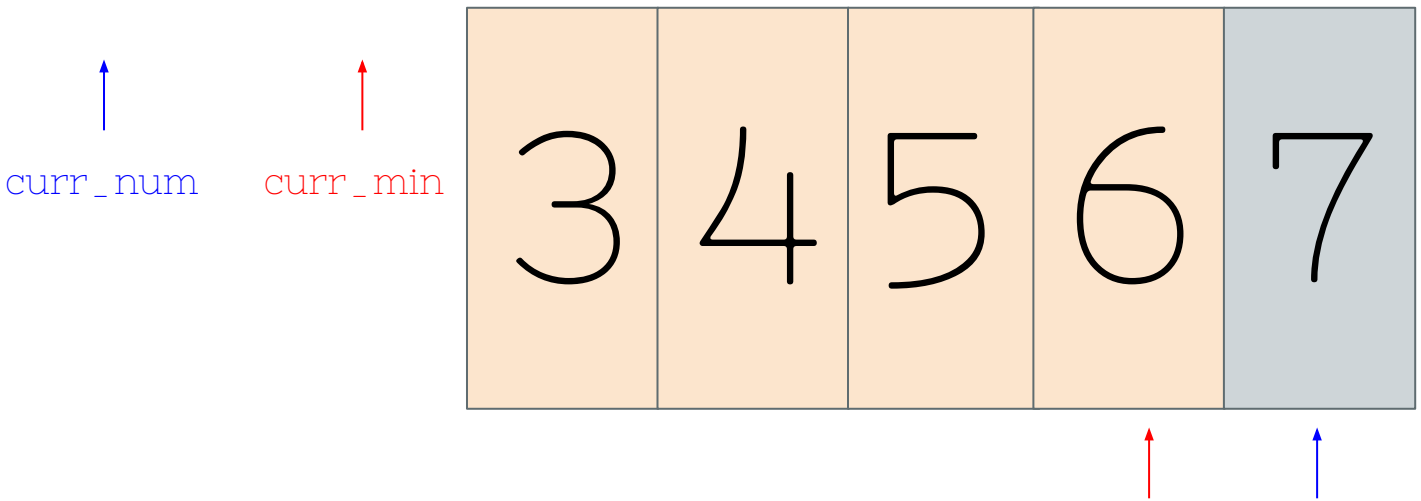
Selection Sort

- sorts an array or a list of items by repeatedly finding the minimum element (or maximum element) from the unsorted part of the list and placing it at the beginning of the sorted part.
- Runtime: $\Theta(N^2)$



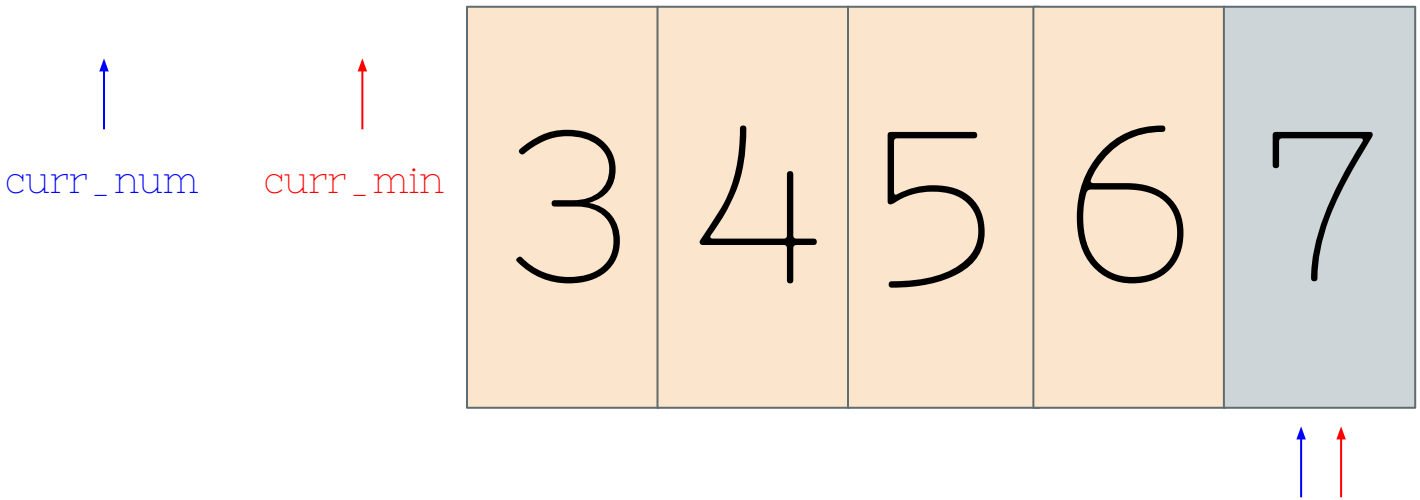
Selection Sort

- sorts an array or a list of items by repeatedly finding the minimum element (or maximum element) from the unsorted part of the list and placing it at the beginning of the sorted part.
- Runtime: **Θ**(N^2)



Selection Sort

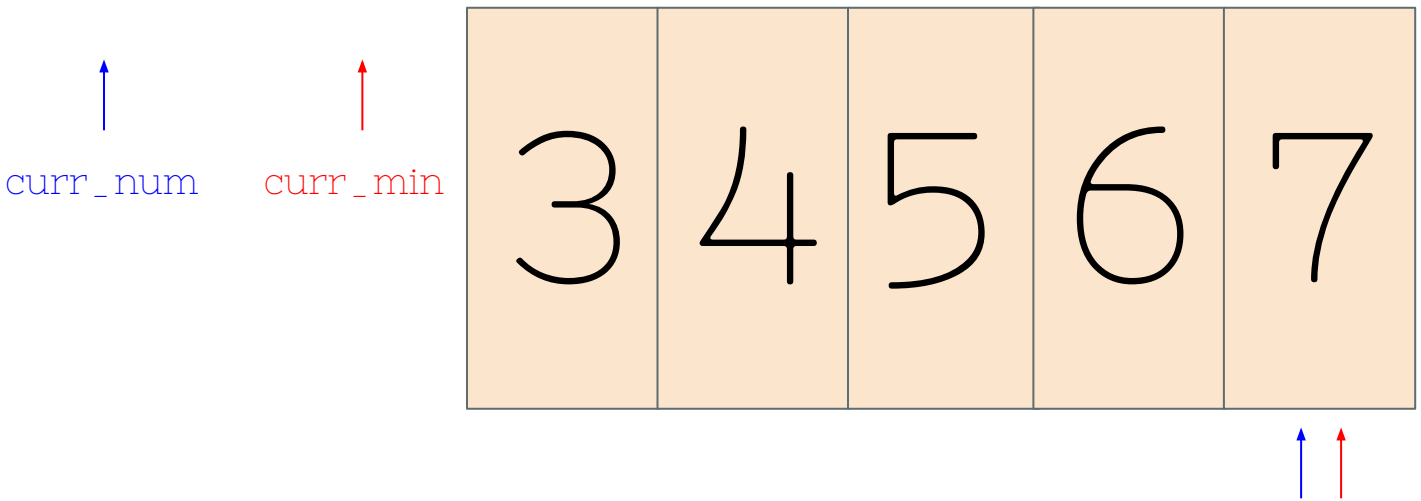
- sorts an array or a list of items by repeatedly finding the minimum element (or maximum element) from the unsorted part of the list and placing it at the beginning of the sorted part.
- Runtime: $\Theta(N^2)$



Can't move curr_num! End of sort!

Selection Sort

- sorts an array or a list of items by repeatedly finding the minimum element (or maximum element) from the unsorted part of the list and placing it at the beginning of the sorted part.
- Runtime: $\Theta(N^2)$



DONE WITH SELECTION SORT

Problem 4 a i) and ii)

4 Mechanical Sorting

a) Suppose that we have the array below.

For each subpart, assume we are working with the original state of the array. For each algorithm, assume it is implemented as explained in lab. By a swap, we mean the exchange of two elements within the same array:

```
void swap(int[] x, int a, int b) {
    int temp = x[a];
    x[a] = x[b];
    x[b] = temp;
}
```

Note that if indices `a` and `b` are equal, i.e. if we try to swap an element with itself, it does **not** count as a swap.

i) Run minimum-based selection sort. After how many swaps does 6 *first* go in front of 7?

◯ 1 ◯ 2 ◯ 3 ◯ 4 ◯ 5 ◯ 6 ◯ 7 ◯ 8 ◯ 9 ◯ 10

ii) Run insertion sort. After how many swaps does 6 *first* go in front of 7?

◯ 1 ◯ 2 ◯ 3 ◯ 4 ◯ 5 ◯ 6 ◯ 7 ◯ 8 ◯ 9 ◯ 10

4 Mechanical Sorting

a) Suppose that we have the array below.

For each subpart, assume we are working with the original state of the array. For each algorithm, assume it is implemented as explained in lab. By a swap, we mean the exchange of two elements within the same array:

```
void swap(int[] x, int a, int b) {
    int temp = x[a];
    x[a] = x[b];
    x[b] = temp;
}
```

Note that if indices `a` and `b` are equal, i.e. if we try to swap an element with itself, it does **not** count as a swap.

i) Run minimum-based selection sort. After how many swaps does 6 *first* go in front of 7?

○ 1 ○ 2 ○ 3 ● 4 ○ 5 ○ 6 ○ 7 ○ 8 ○ 9 ○ 10

ii) Run insertion sort. After how many swaps does 6 *first* go in front of 7?

○ 1 ○ 2 ○ 3 ○ 4 ○ 5 ○ 6 ○ 7 ○ 8 ○ 9 ○ 10

4a i)

[4, 2, 1, 7, 3, 6, 0]

[0, 2, 1, 7, 3, 6, 4]

[0, 1, 2, 7, 3, 6, 4]

[0, 1, 2, 3, 7, 6, 4]

[0, 1, 2, 3, 4, 6, 7]

4 Mechanical Sorting

a) Suppose that we have the array below.

For each subpart, assume we are working with the original state of the array. For each algorithm, assume it is implemented as explained in lab. By a swap, we mean the exchange of two elements within the same array:

```
void swap(int[] x, int a, int b) {
    int temp = x[a];
    x[a] = x[b];
    x[b] = temp;
}
```

Note that if indices `a` and `b` are equal, i.e. if we try to swap an element with itself, it does **not** count as a swap.

i) Run minimum-based selection sort. After how many swaps does 6 *first* go in front of 7?

○ 1 ○ 2 ○ 3 ● 4 ○ 5 ○ 6 ○ 7 ○ 8 ○ 9 ○ 10

ii) Run insertion sort. After how many swaps does 6 *first* go in front of 7?

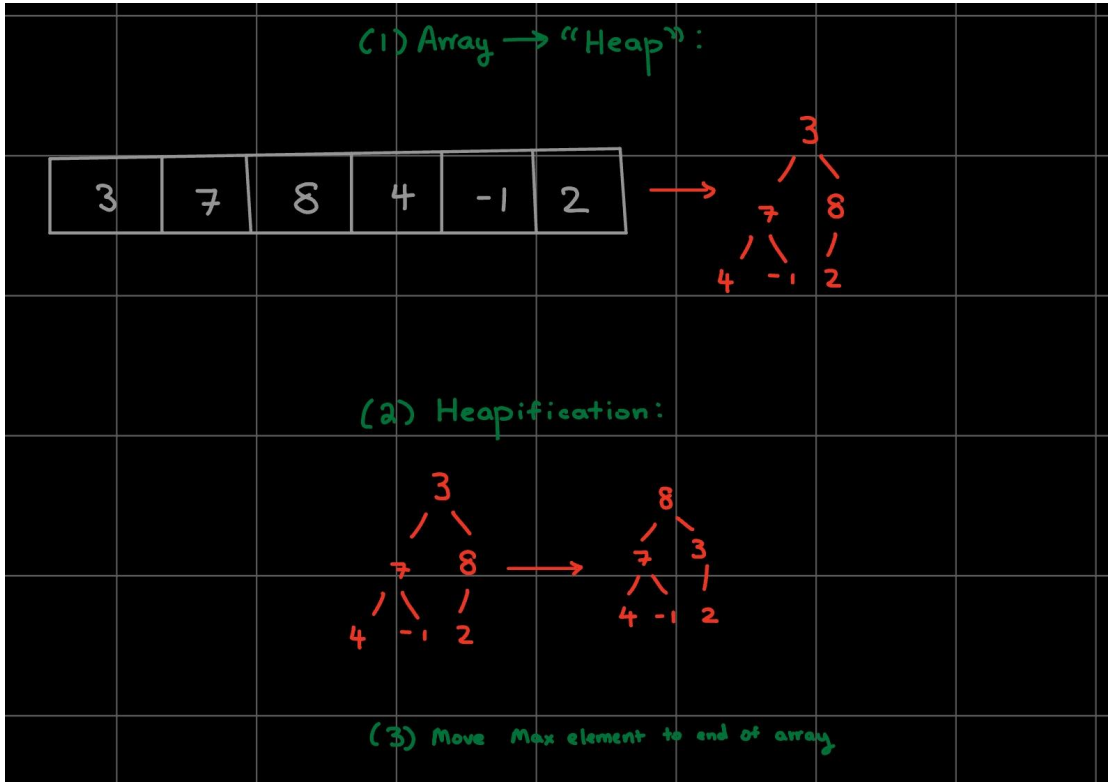
○ 1 ○ 2 ○ 3 ○ 4 ○ 5 ● 6 ○ 7 ○ 8 ○ 9 ○ 10

4a ii)

[4, 2, 1, 7, 3, 6, 0]
[2, 4, 1, 7, 3, 6, 0]
[2, 1, 4, 7, 3, 6, 0]
[1, 2, 4, 7, 3, 6, 0]
[1, 2, 4, 3, 7, 6, 0]
[1, 2, 3, 4, 7, 6, 0]
[1, 2, 3, 4, 6, 7, 0]

Heap Sort

- Sort Elements using RemoveMax Operation
- Runtime: O(N*logN)



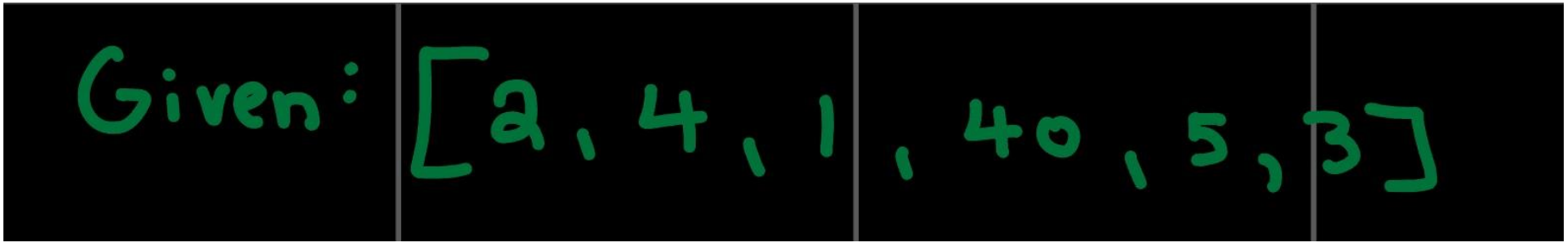
LSD Radix Sort

Steps:

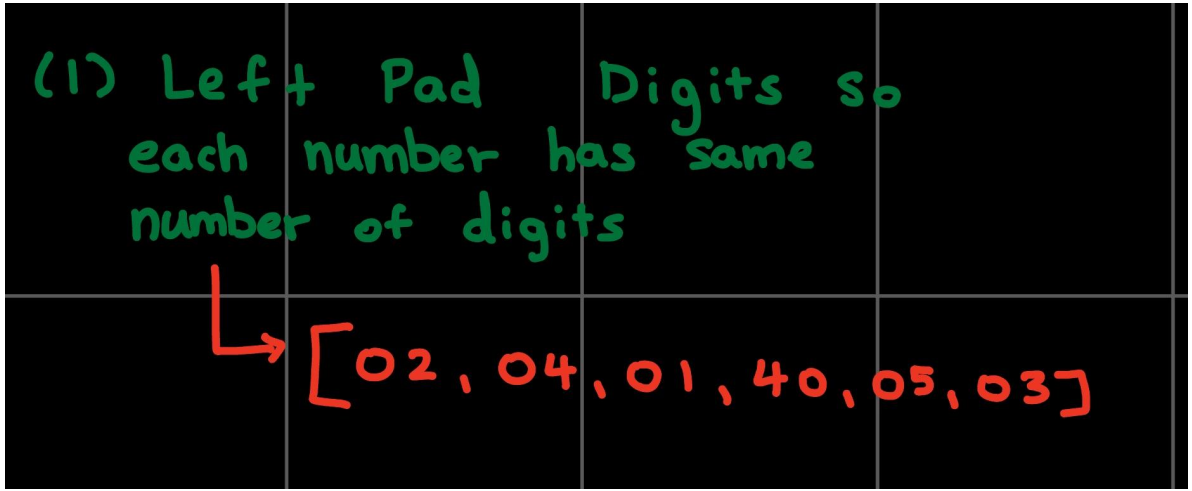
1. Left-Pad Numbers with 0's such that each number has an equivalent number of digits
2. Group by Rightmost Digit (Least Significant Digit or LSD)
3. Group by Second Rightmost Digit
4. ... Until you Group by the Leftmost Digit

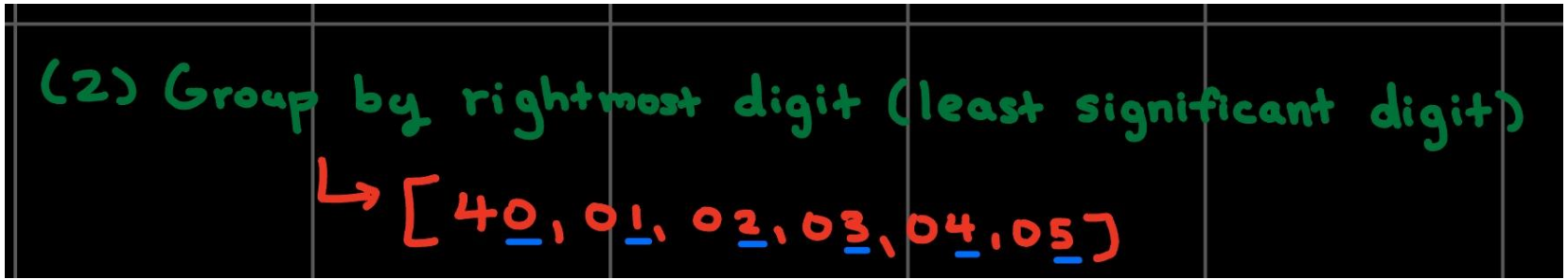
Runtime:
O(WN + WR)
N: Number of Elements
R: Alphabet Size
W: Item Width

LSD Radix Sort Example

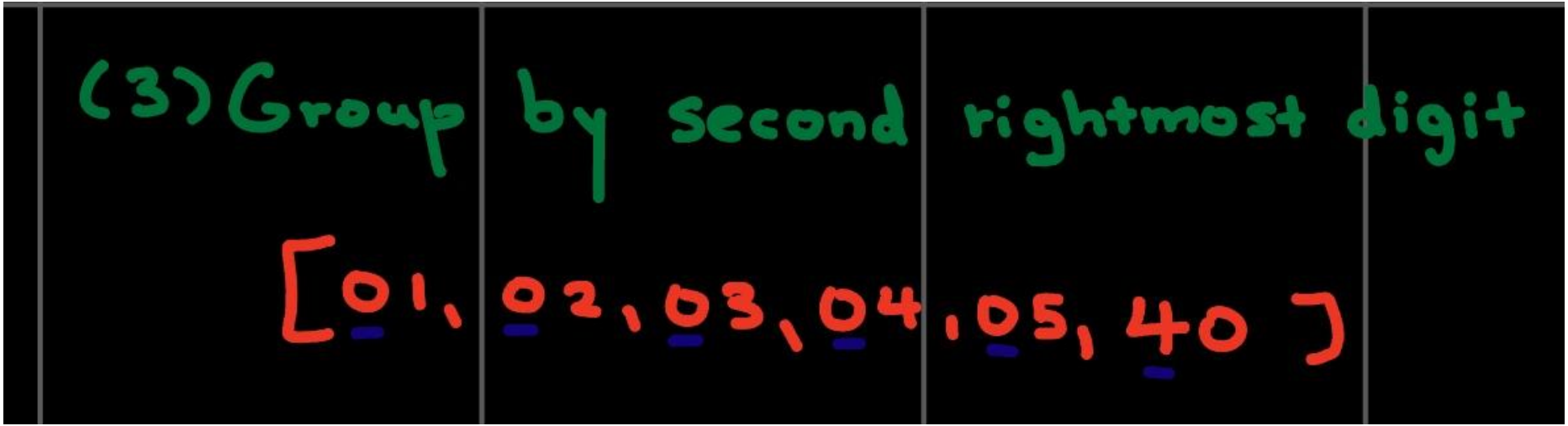


LSD Radix Sort Example (Continued)





LSD Radix Sort Example (Continued)



MSD Radix Sort

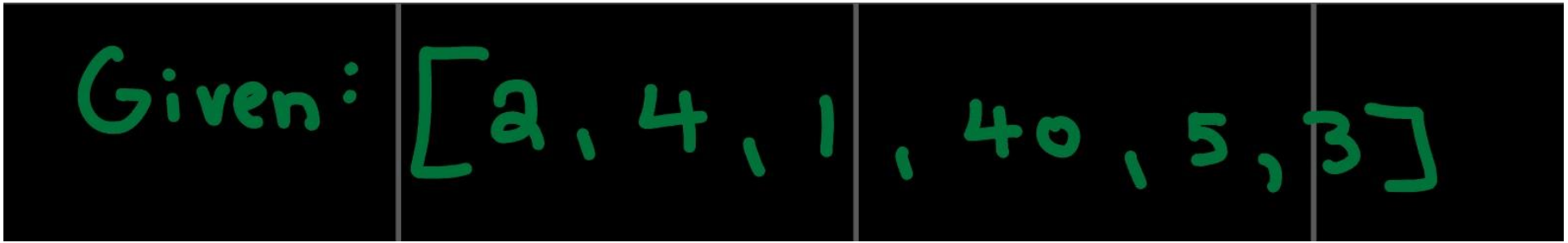
Steps:

1. Left-Pad Numbers with 0's such that each number has an equivalent number of digits
2. Group by Leftmost Digit (Most Significant Digit or MST)
3. Group by Second Leftmost Digit
4. ... Until you Group by the Rightmost Digit

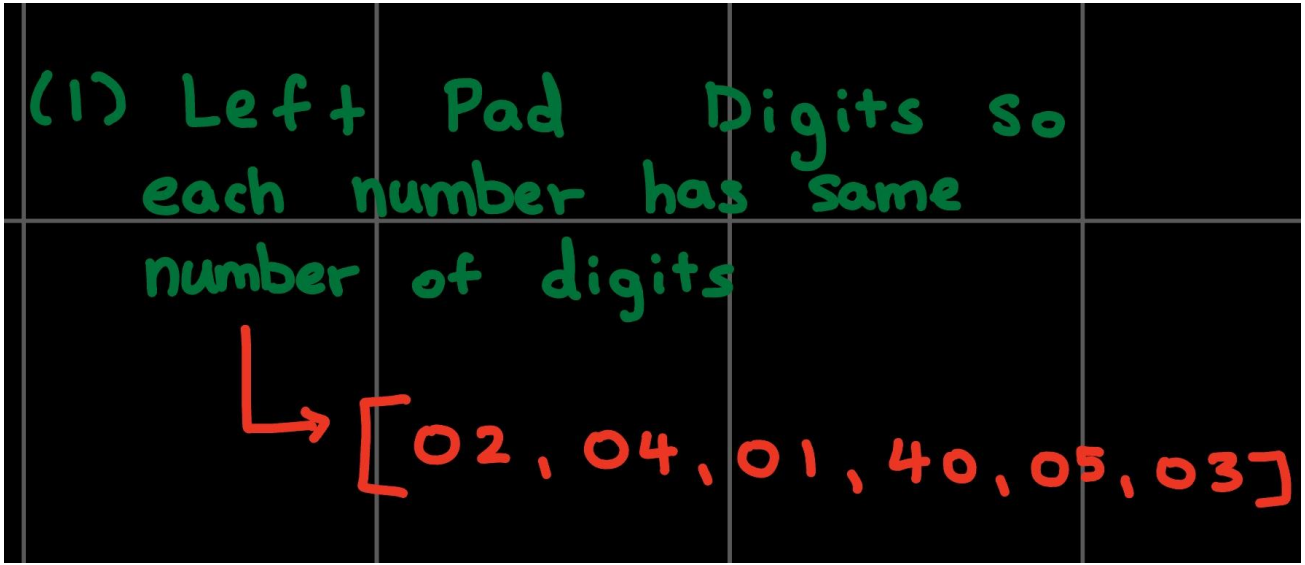
Runtime:
O(WN + WR)
N: Number of Elements
R: Alphabet Size
W: Item Width

Best Case Runtime
$\Omega(N + R)$
N: Number of Elements
R: Alphabet Size

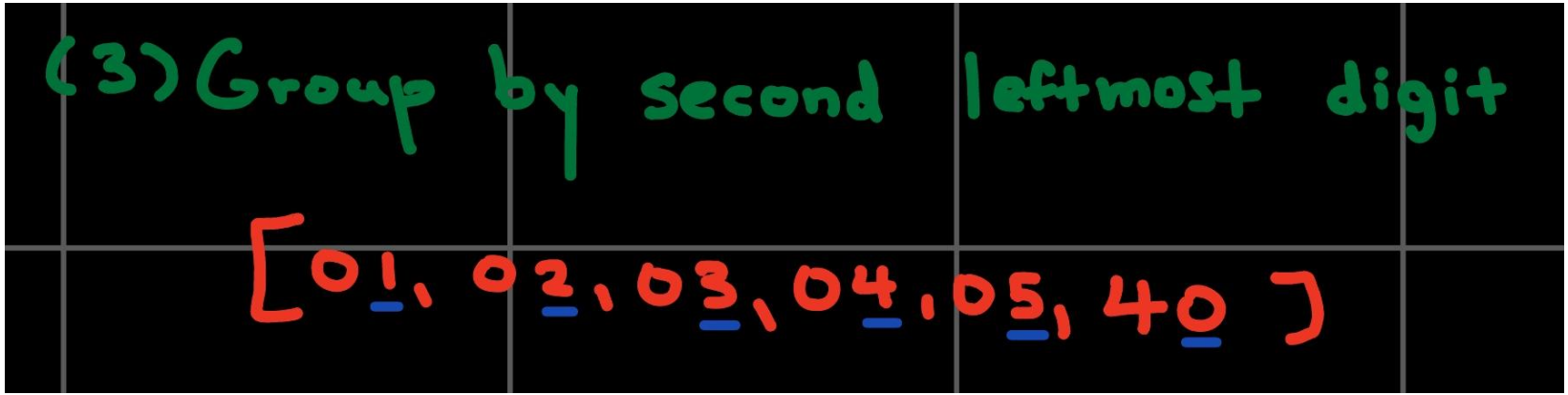
MSD Radix Sort Example

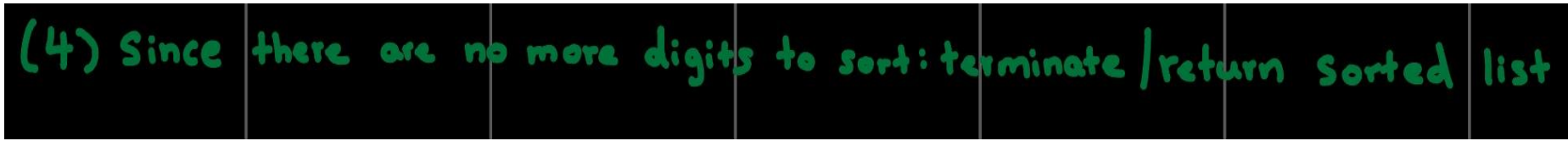


MSD Radix Sort Example (Continued)



MSD Radix Sort Example (Continued)

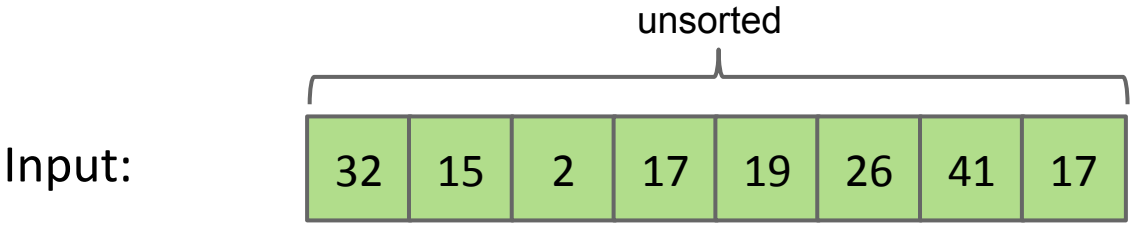




Merge Sort

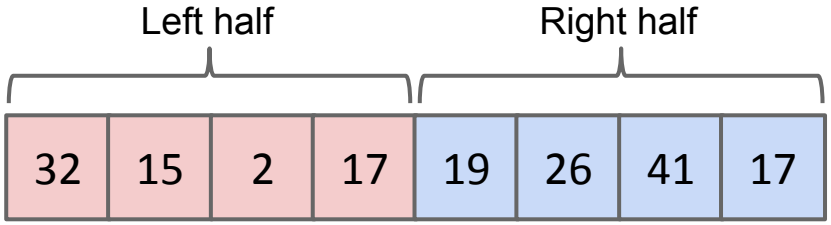
Top-Down merge sorting N items:

- Split items into 2 roughly even pieces.
- Mergesort each half.
- Merge the two sorted halves to form the final result.

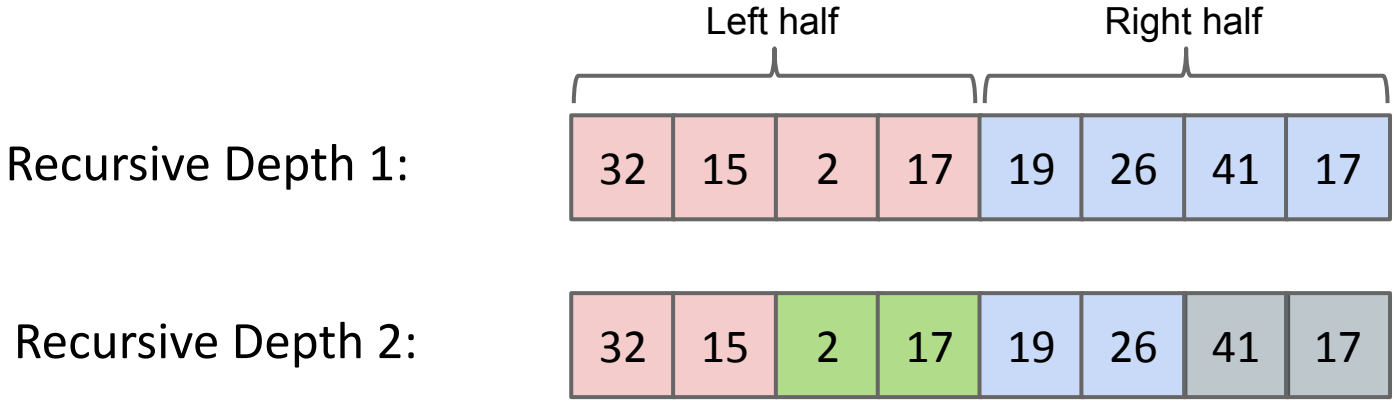


Merge Sort Recursive Step

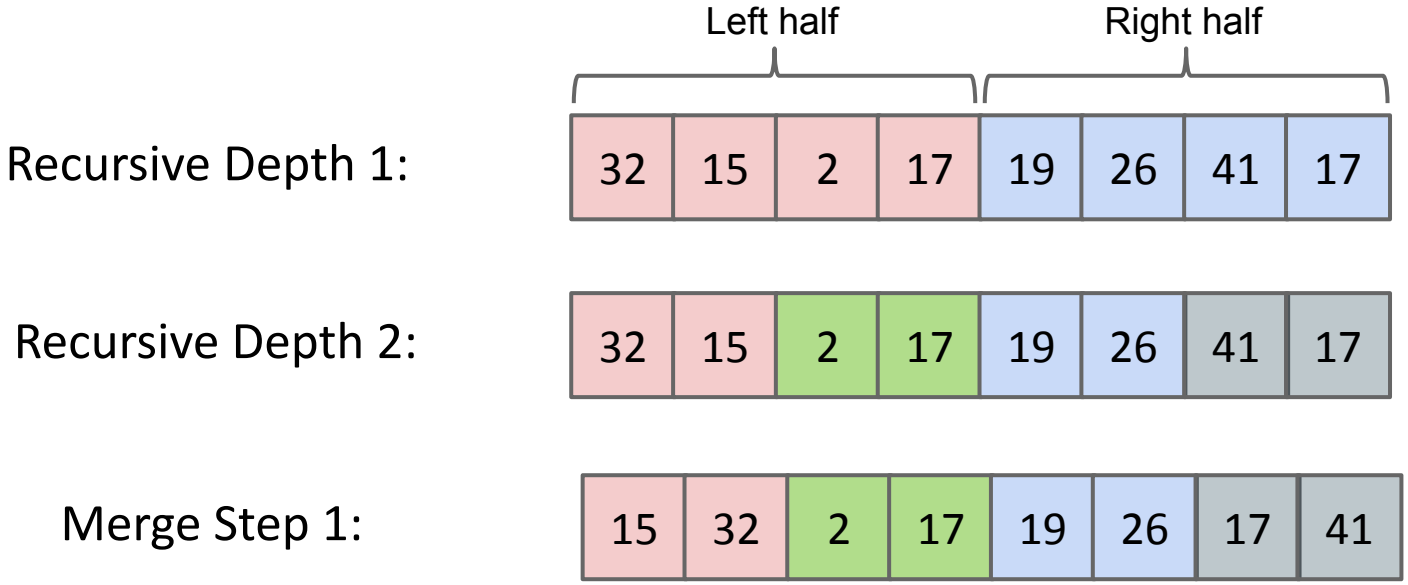
Recursive Depth 1:



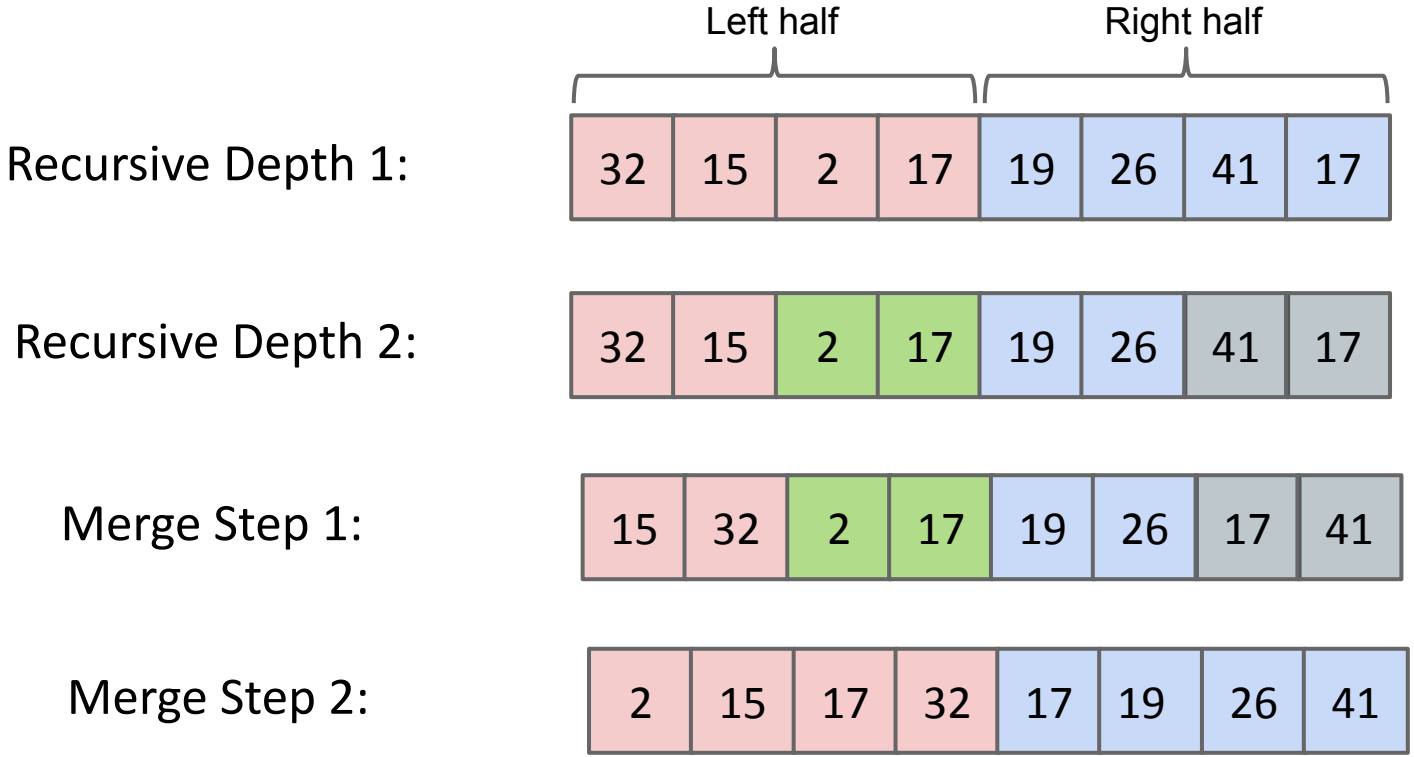
Merge Sort Recursive Step



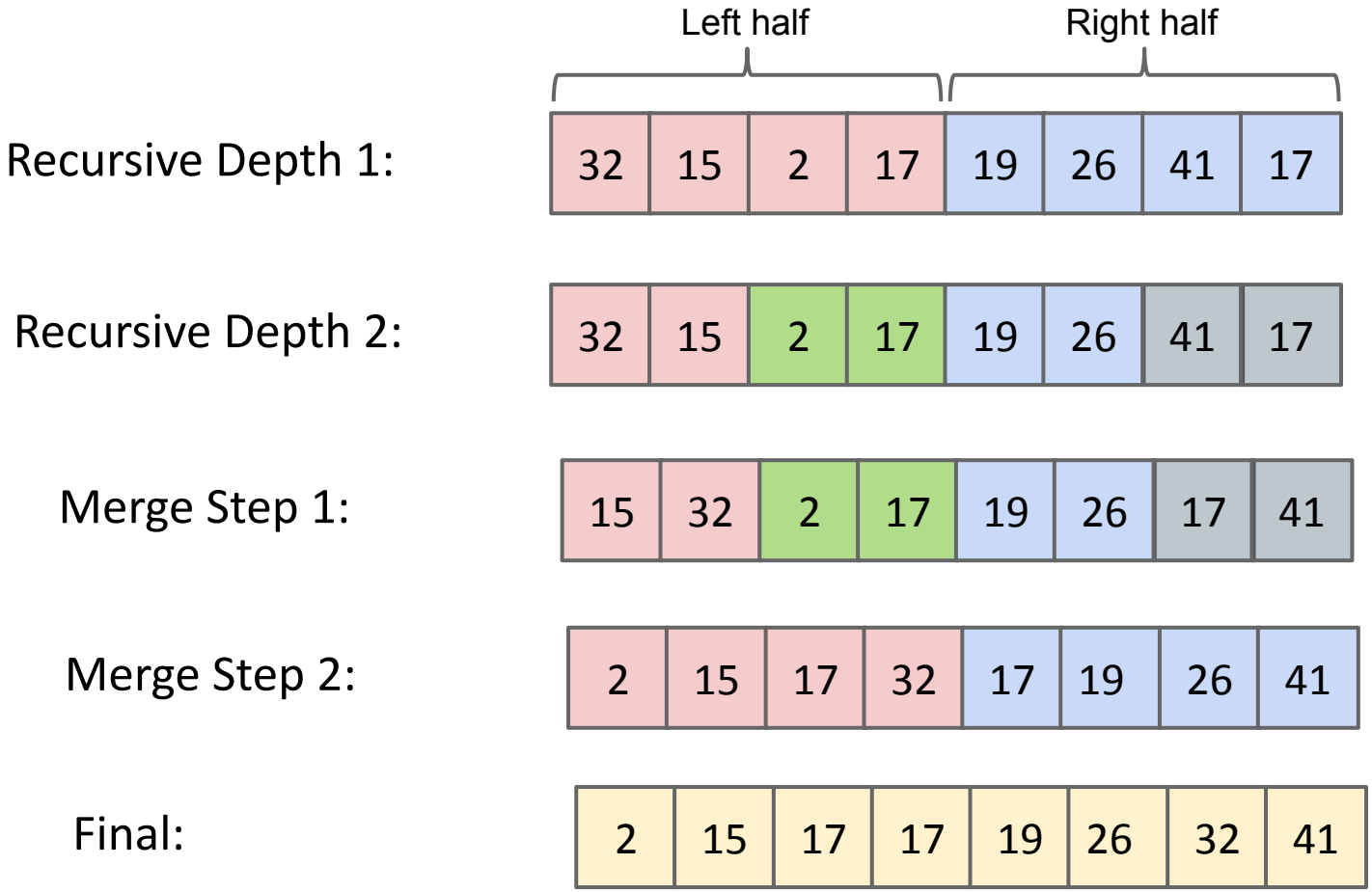
Merge Sort Recursive Step



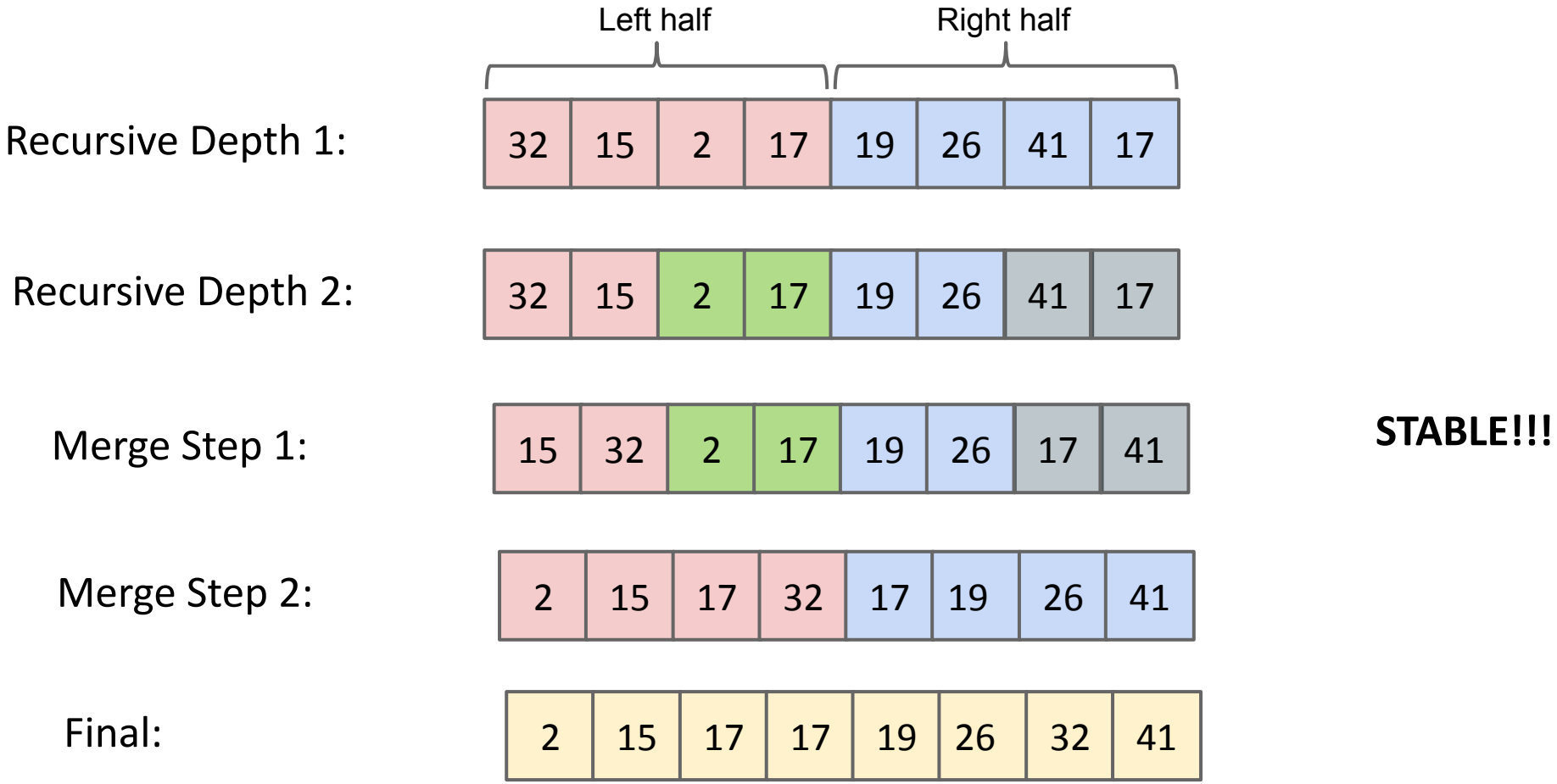
Merge Sort Recursive Step



Merge Sort Recursive Step



Merge Sort Recursive Step

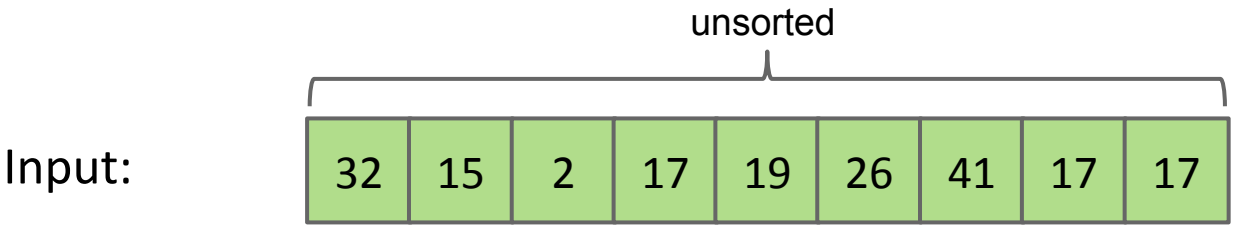


STABLE!!!

Quick Sort

Quick sorting N items:

- Partition on leftmost item.
- Quicksort left half.
- Quicksort right half.



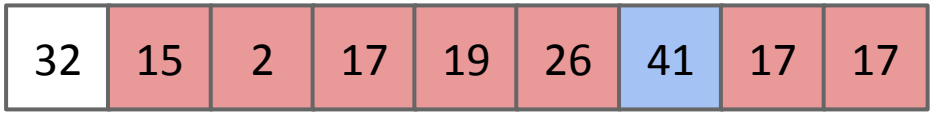
Quick Sort

Quick sorting N items:

- **Partition on leftmost item (32).**
- Quicksort left half.
- Quicksort right half.

partition(32)

Input:

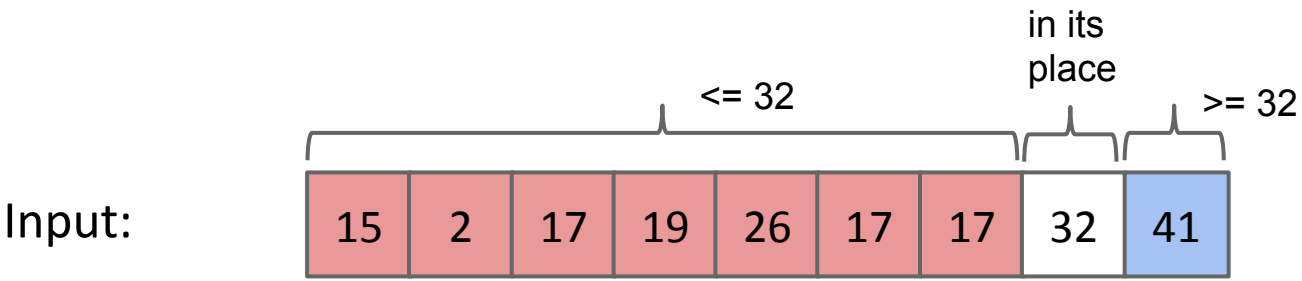


Quick Sort

Quick sorting N items:

partition(32)

- **Partition on leftmost item (32).**
- Quicksort left half.
- Quicksort right half.

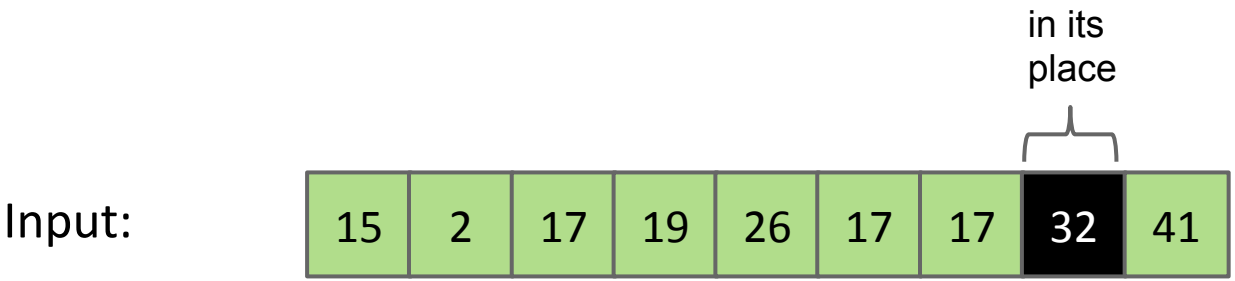


Quick Sort

partition(32)

Quick sorting N items:

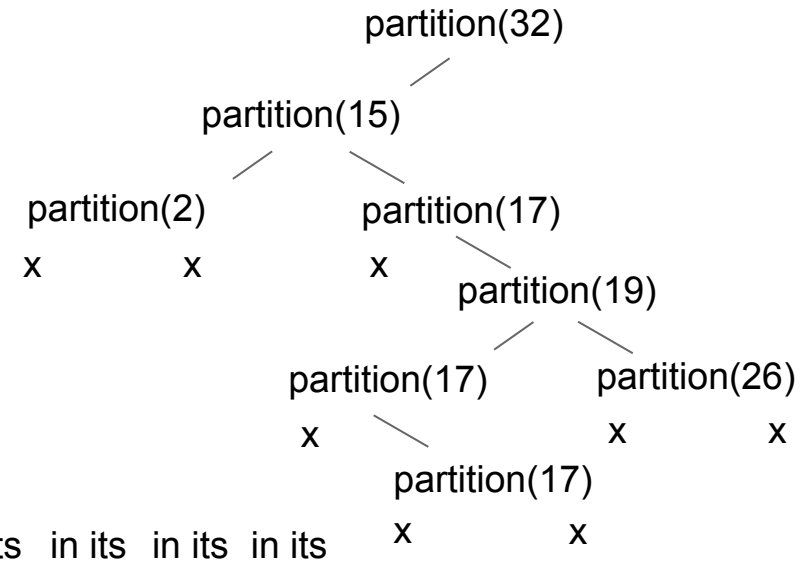
- **Partition on leftmost item (32) (done).**
- Quicksort left half.
- Quicksort right half.

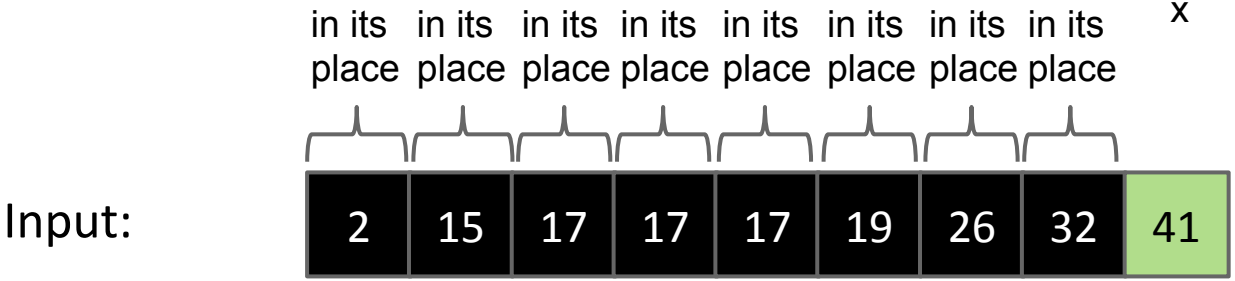


Quick Sort

Quick sorting N items:

- Partition on leftmost item (32) (done).
- **Quicksort left half (details not shown).**
- Quicksort right half.

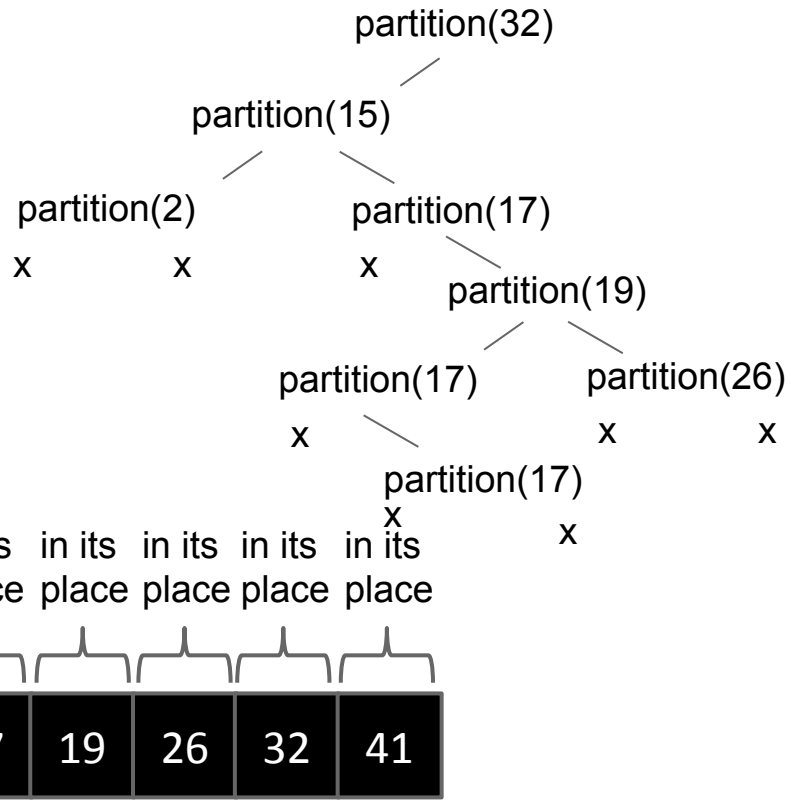




Quick Sort

Quick sorting N items:

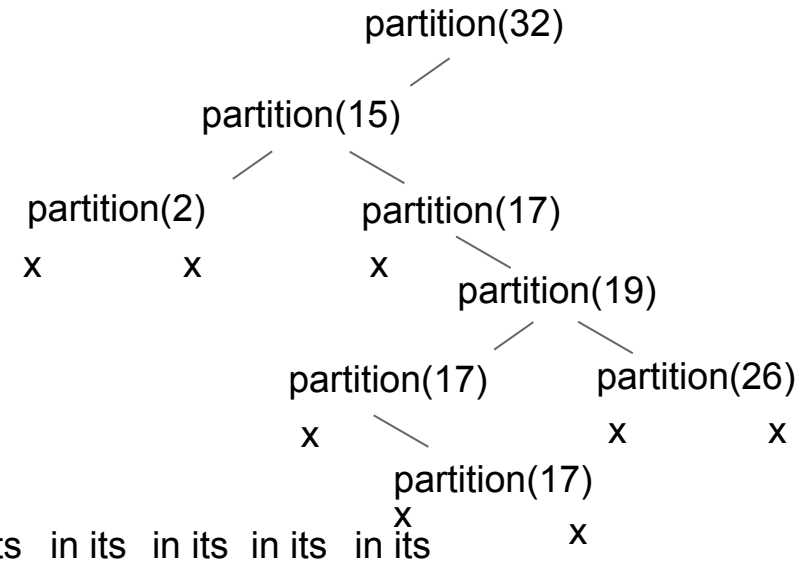
- Partition on leftmost item (32) (done).
- Quicksort left half (details not shown).
- **Quicksort right half (details not shown).**



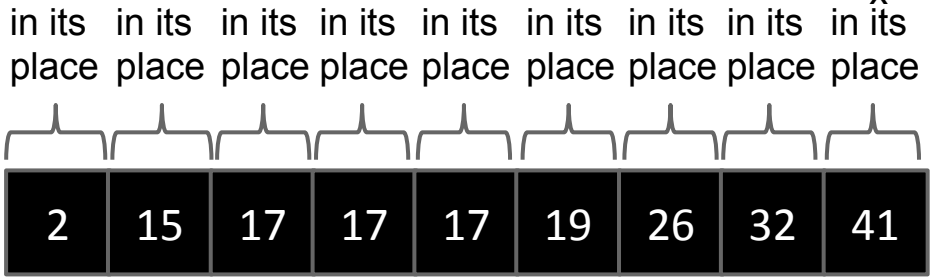
Quick Sort

Quick sorting N items:

- Partition on leftmost item (32) (done).
- Quicksort left half (details not shown).
- **Quicksort right half (details not shown).**



NOT STABLE!



4c

Suppose we want to sort all the `Olympians` in ascending order by their `medalCount` breaking ties alphabetically by `event` and then `name`. For simplicity, assume `names` are unique. For instance, if we have the Olympians below:

To achieve this, you may call each of `insertionSort`, `selectionSort`, and `mergeSort` exactly once. `sortingKey` should be one of "name", "event", or "medalCount". The signatures of the methods are below:

4c

Sorting Order: **Ascending order by `medalCount`**, breaking ties by **`event`** then **`name`**

Available Sorts: 1) **`insertionSort`**, 2) **`selectionSort`** , and 3) **`mergeSort`**

```
public List<Olympian> sortOlympians(List<Olympian> olympians)
{
    olympians = ____________________________________________;
    olympians = ____________________________________________;
    olympians = ____________________________________________;
    return olympians;
}
```

4c

Sorting Order: **Ascending order by `medalCount`**, breaking ties by **`event`** then **`name`**

Available Sorts: 1) **`insertionSort`**, 2) **`selectionSort`** , and 3) **`mergeSort`**

```
public List<Olympian> sortOlympians(List<Olympian> olympians)
{
    olympians = <any sort>;
    olympians = <stable sort>;
    olympians = <stable sort>;
    return olympians;
}
```

4c

Sorting Order: **Ascending order by `medalCount`**, breaking ties by **`event`** then **`name`**

Available Sorts: 1) **`insertionSort`**, 2) **`selectionSort`** , and 3) **`mergeSort`**

```
public List<Olympian> sortOlympians(List<Olympian> olympians)
{
    olympians = selectionSort(olympians, "name");
    olympians = [insertionSort or mergeSort](olympians, "event");
    olympians = [insertionSort or mergeSort](olympians, "medalCount");
    return olympians;
}
```

Final Review Session: Data Structures

CS61B Data Structures

CS61B Data Structures and Implementations: What can they do? Do they do things? Let's find out!

Agenda

01 Abstract Data Types

02 Disjoint Sets

03 Balanced Trees

04 Heaps

05 Hashing

06 Overview

A Quick Overview of Abstract Data Types

ADTs: What the heck are they and Why do we use them so much?

- Let’s say I got a driver's license and needed a car
  - Problem is I’m not that technically capable in telling cars apart
  - But I know that I need a vehicle that can drive
    - It can have other additional operation (like a rocket booster in the batmobile) but to qualify as a car it needs to be able to drive.
  - This is an abstraction barrier for the user
- Abstract Data Types are defined by their operations, not their implementation
  - For a class to fall into being an ADT they must have all of the operations the ADT expects
    - Ex. an ArrayList could not be a list if there was not a .size() function
  - But this does not limit the class in it’s methods in anyway as there can be additional functions beyond that

ADTs: What the heck are they and Why do we use them so much?

- This allows the implementation of the operation to be open to you!
  - But why should we do that?
    - It allows for different runtimes
    - It allows for the use of different Data Structures to implement at your leisure

Lists

```
public interface List<T>
{
        int size();

        void add(T element);

        boolean contains (T element);

        boolean remove (T element);

        T get(T element);


        String toString();

 }
```

- I don't know how an instance of a List ADT is implemented but I know it has these operations, that I can use as the user.
- So if you find yourself implementing an interface be sure to have all of the needed methods implemented.
- Be sure to use the interface as a needed blueprint for your class!

A Quick Overview of Balanced Trees

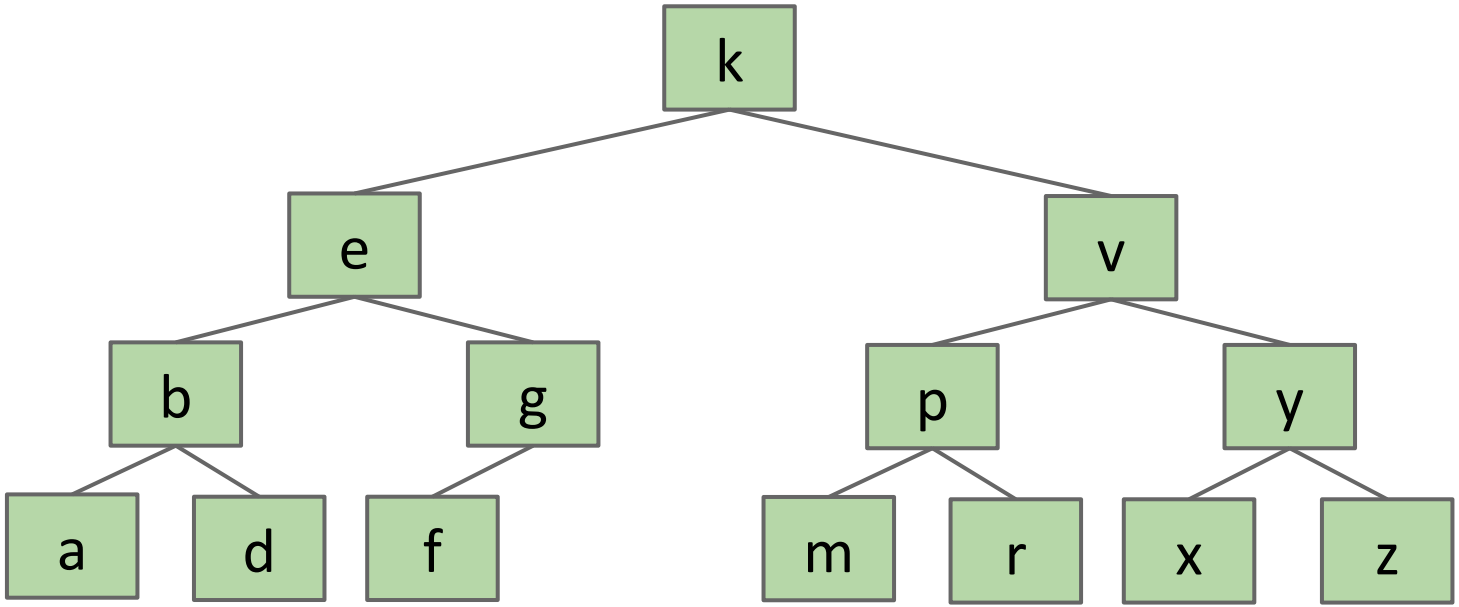
Balanced Trees: What the heck are they and Why do we use them so much?

- Binary Search Trees are trees where each node has 2 or less children s.t. the left child is less than the parent, and the right child is more than the parent
  - Why do we want it balanced?
    - Well when we traverse the tree if it is perfectly balanced, the left tree has ½ of the elements of the tree. So by choosing one child to traverse to we eliminate the need to look at half the elements! Then when we traverse again we eliminate the need to look at ¼ of the elements and so on and so one
    - So if we have a balanced tree traversal takes only log(N) with log(N) levels.
    - So the reason we use it is to be organized and find a value fast!

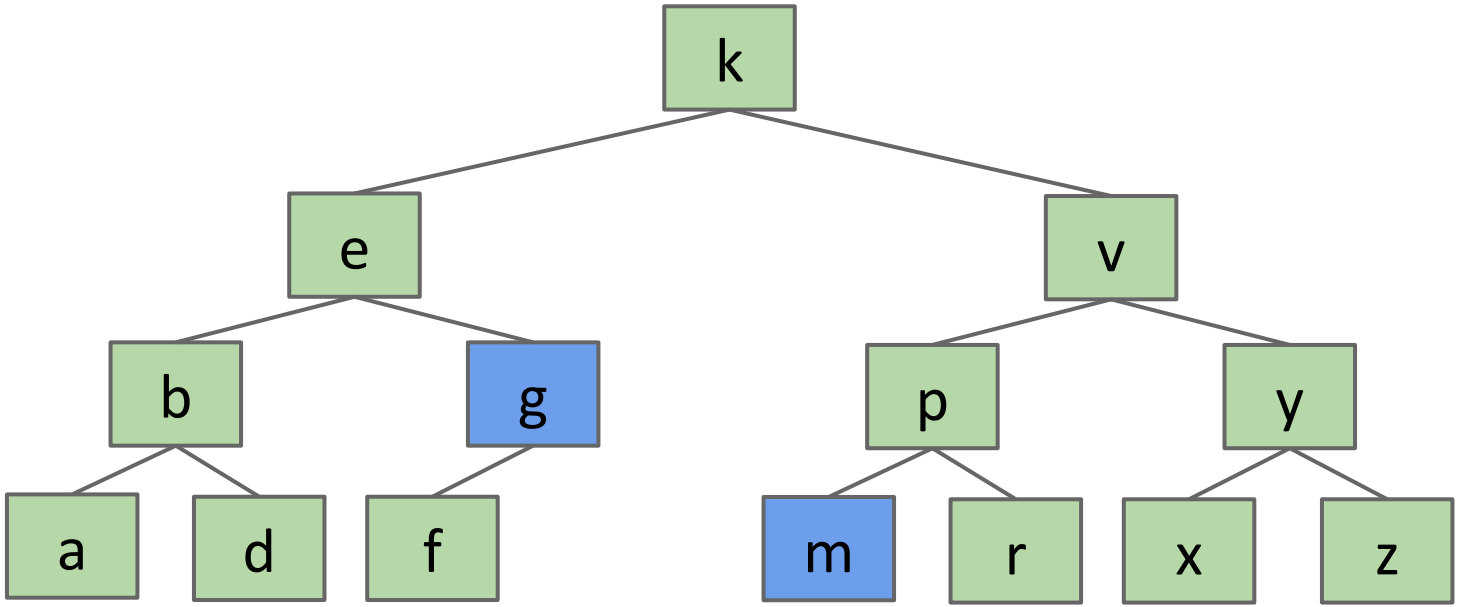
Balanced Trees: Operations

- Search and insertion should just be traversing the tree, but what about deleting?
  - We really use Hibbard Deletion which finds the element which is most similar to the element to replace it
    - Ie finding the largest value of left tree or the smallest value of the right tree

Balanced Trees: What if we deleted K what could we replace it with?



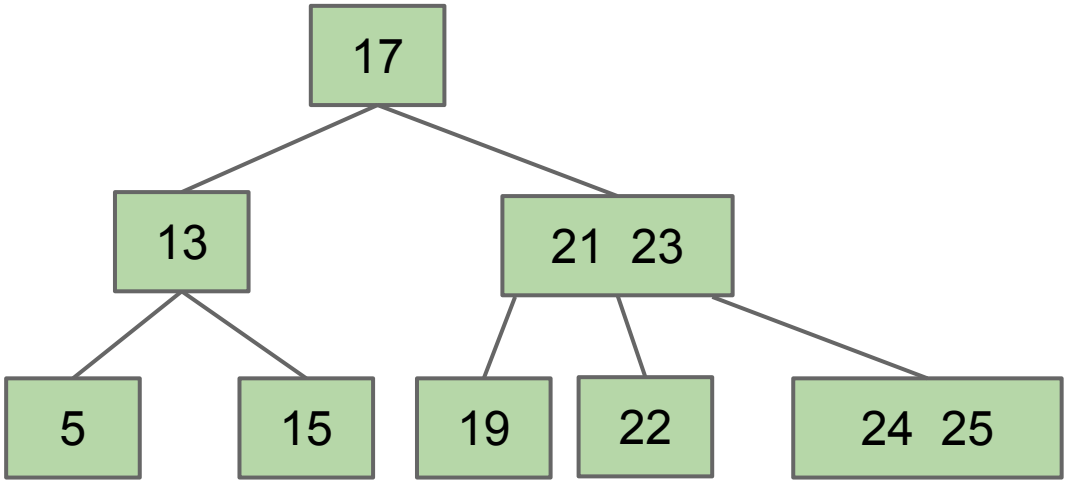
Balanced Trees: What if we deleted K what could we replace it with?



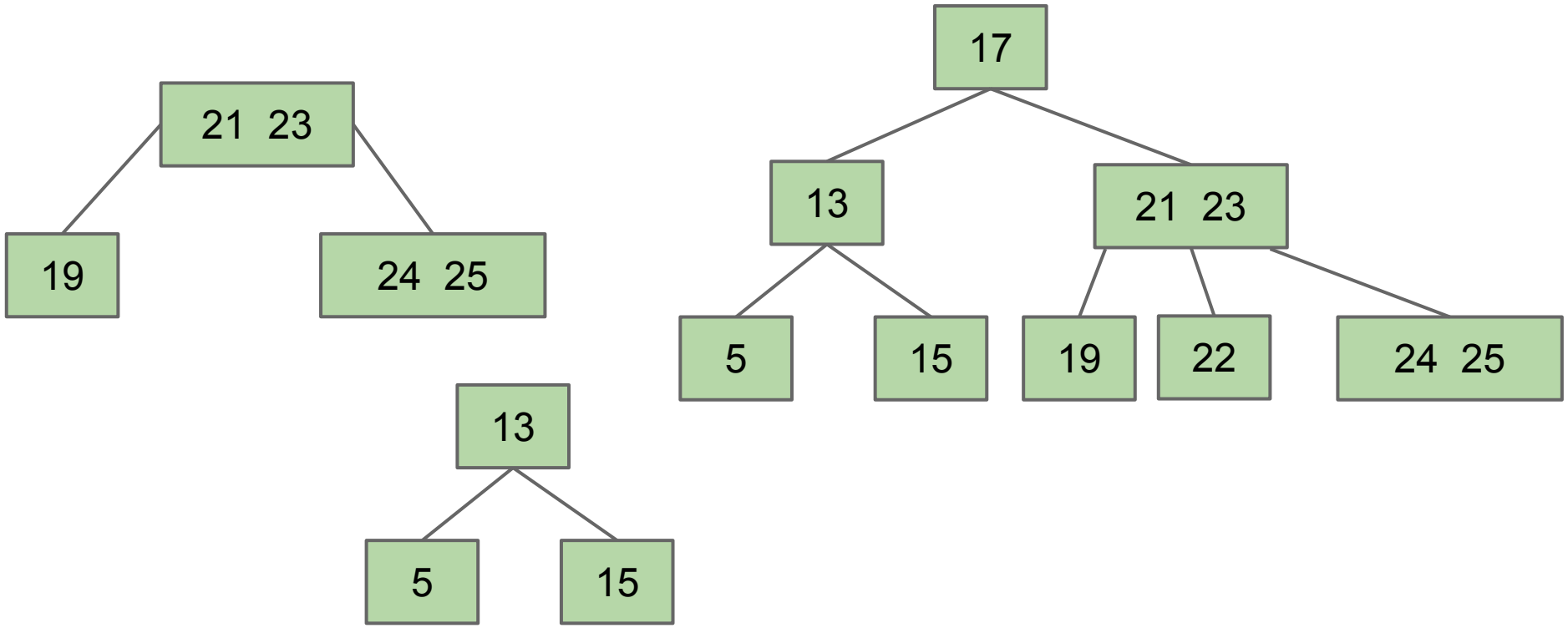
B-Trees: What the heck are they and Why do we use them so much?

- We can't always rely on the user to put things in the right order
  - That's why we have those worst case runtimes!
  - So what if we made a tree that was user proof and would always be balanced?
- That's why we will create B-Trees
  - But what the heck are they??
    - These will be trees like we've seen before but this time each node can hold more than one element
    - Ie a 2-3 tree can have a node hold 2 elements though other B-trees can hold more than that
    - If a node is holding too many elements we will push up the middle element to ensure that the tree remains balanced!

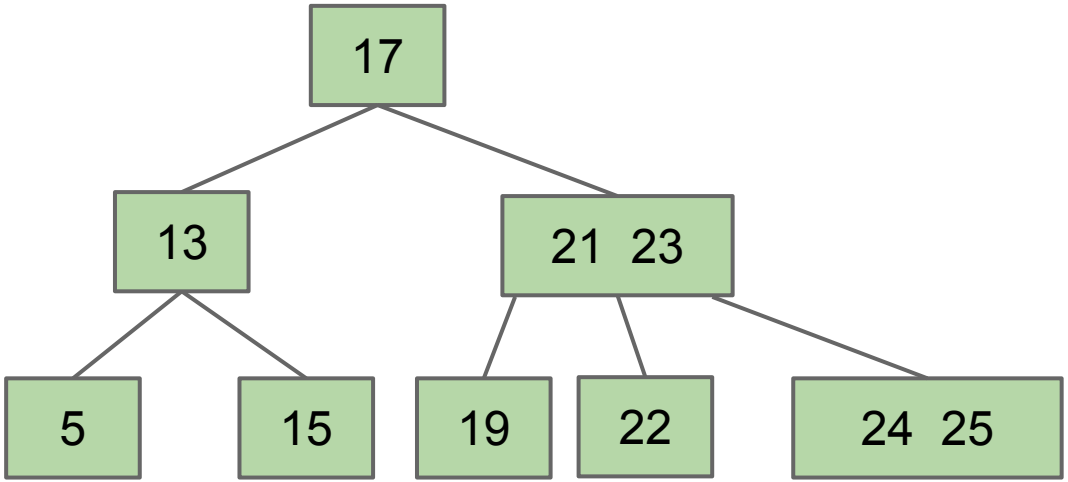
B-Trees: What the heck do they look like?



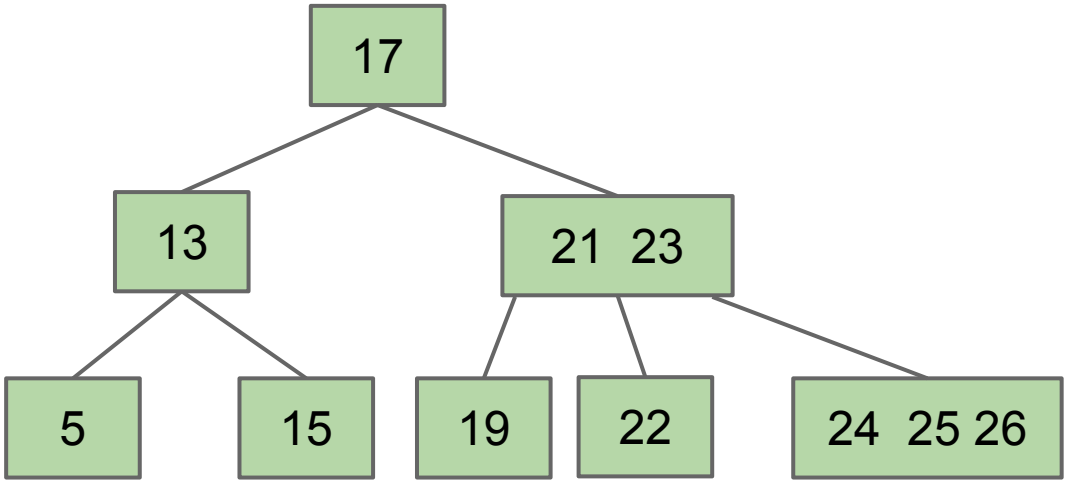
B-Trees: What the heck do they not look like?



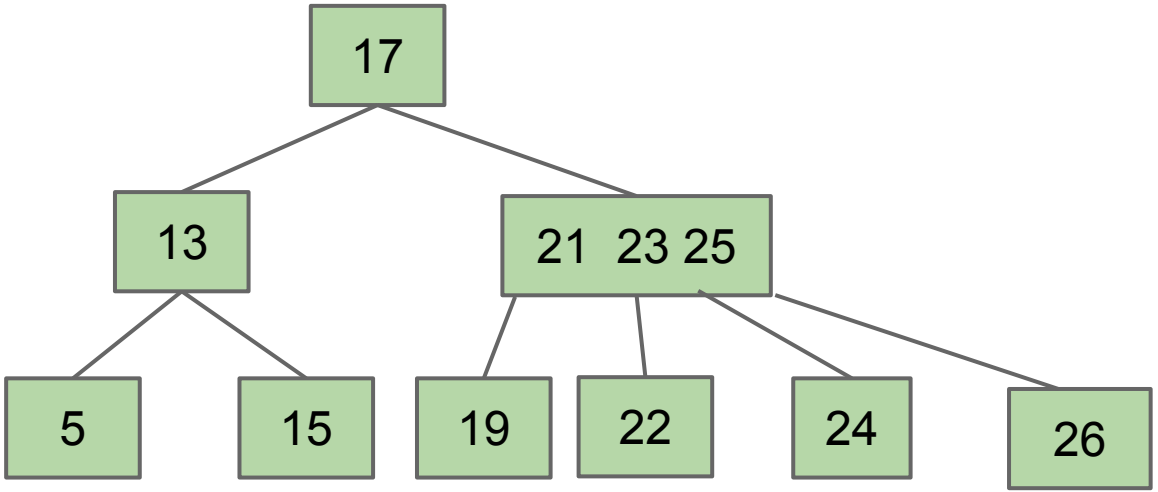
B-Trees: What the heck do they look like if we insert 26?



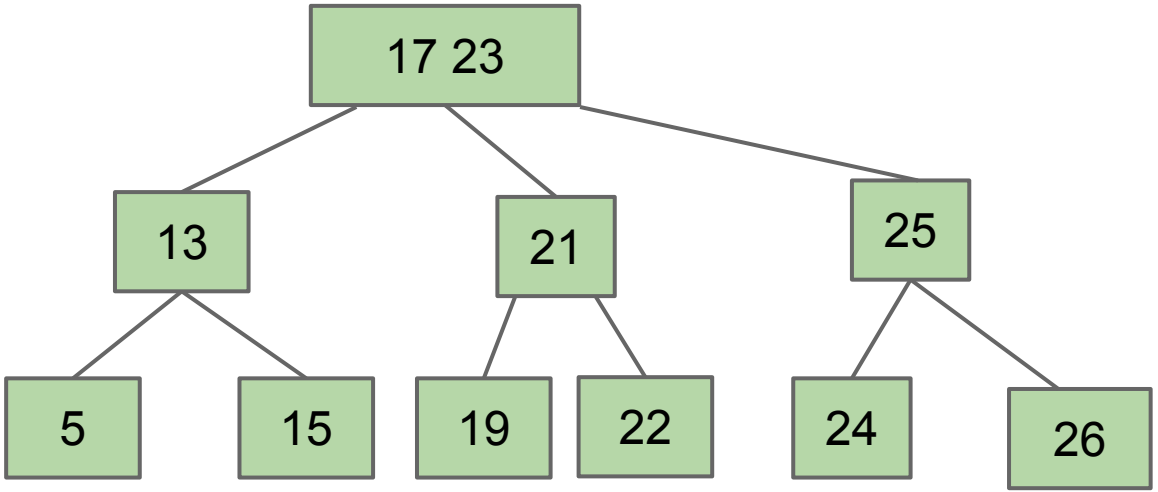
B-Trees: What the heck do they look like if we insert 26?



B-Trees: What the heck do they look like if we insert 26?



B-Trees: What the heck do they look like if we insert 26?



B-Trees

- Insert new elements at the leaf level
- Height has a worst case of Θ (log(N))
- All operations will take O(log(N))
- Has splitting, in which case we push up the middle element
- Deletion out of scope
- Harder to implement in actual code

BST

- Insert new elements at the leaf level
- Height will be at worst case of Θ(N)
- In the best case operations will take Θ(log(N)) but in the worst case it will take Θ(N)
- Has no splitting
- Deletion is in scope
- Easier to implement in code

B-Trees

- Insert new elements at the leaf level
- Height has a worst case of Θ (log(N))
- All operations will take O(log(N))
- Has splitting, in which case we push up the middle element
- Deletion out of scope
- **Harder to implement in actual code**

BST

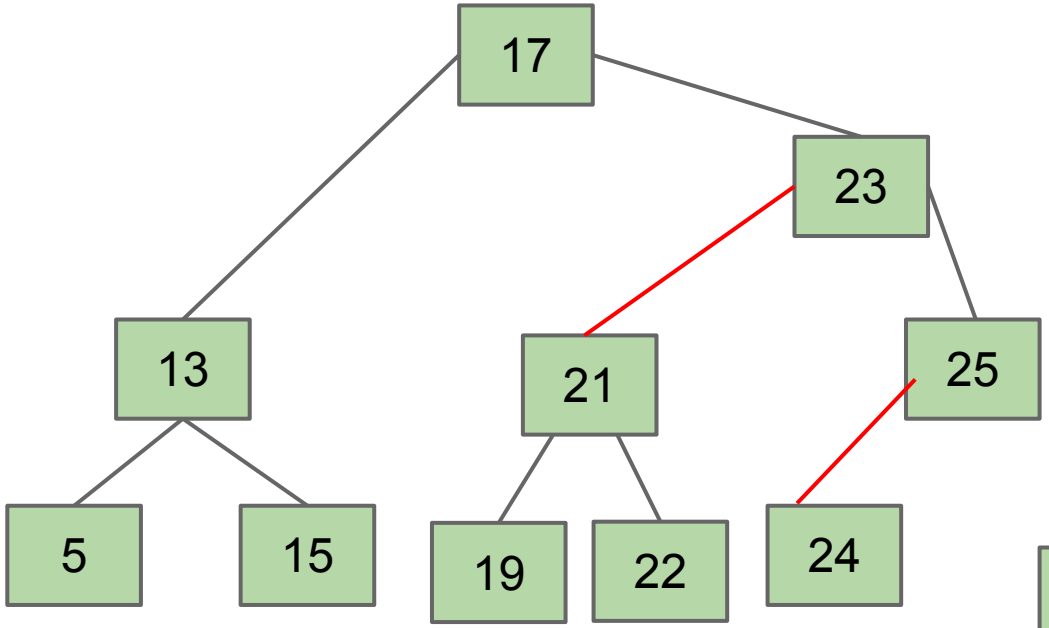
- Insert new elements at the leaf level
- Height will be at worst case of Θ(N)
- In the best case operations will take Θ(log(N)) but in the worst case it will take Θ(N)
- Has no splitting
- Deletion is in scope
- Easier to implement in code

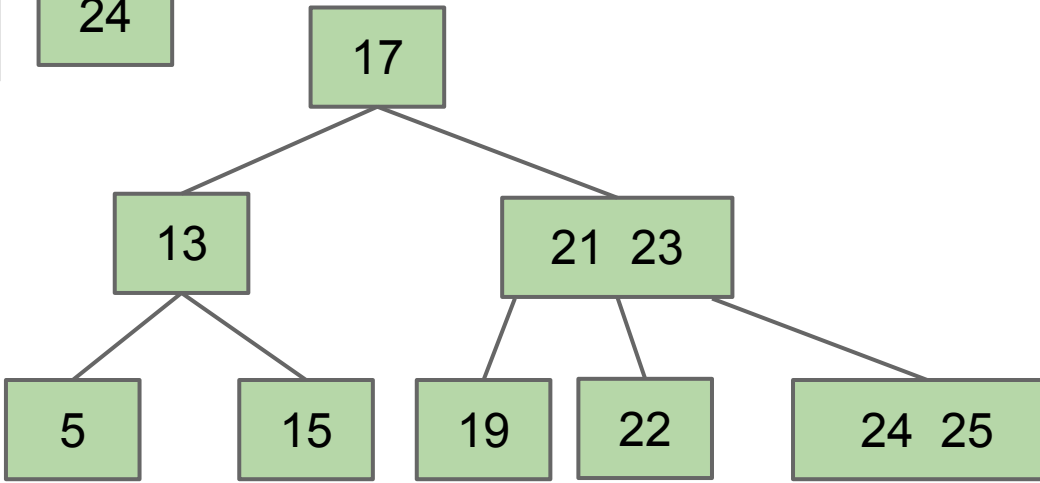
Red Black Trees: What the heck are they and Why do we use them so much?

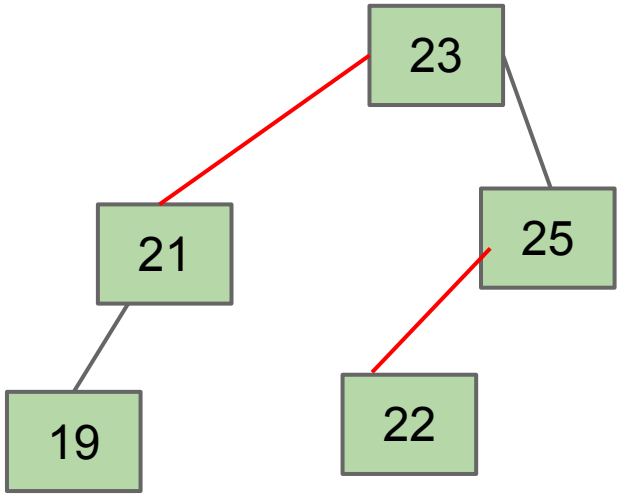
- We want a balanced tree but that's really hard to implement. So we should think of something that is easy to implement **Red Black Trees**
  - These are similar to B-Trees in structure except that instead of multiple elements being stored in one node they are connected through left leaning red child.
  - This is easy to implement and self balancing!
  - Most importantly they keep the O(log(N)) height

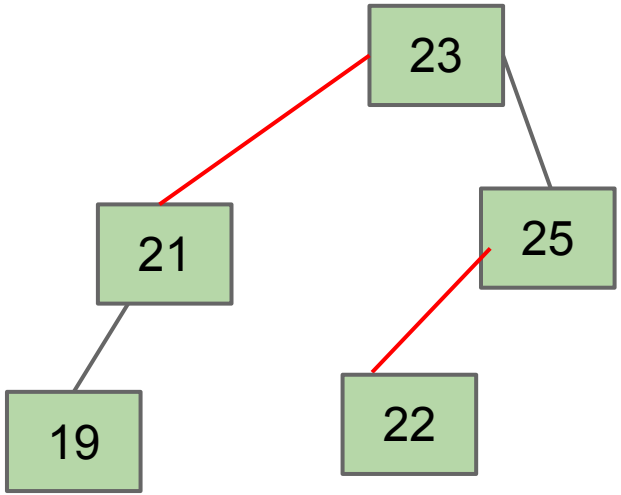
The Rules of Adding

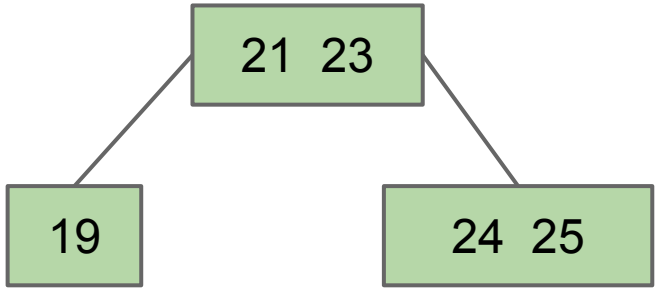
- We still add at the leaf level except we have a red branch
- We may need to rotate the nodes but when?
  - (1) The red branches must be left leaning
  - (2) If there are 2 consecutive red left leaning branches then we must shift the nodes rightward
  - (3) if a parent has 2 red children then the colors must be switch


17
23
13
21
25
5
15
19
22
24


17
13
21 23
5
15
19
22
24 25

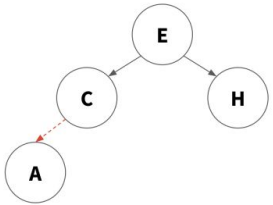

23
21
25
19
22


23
21
25
19
22


21 23
19
24 25

3 (Extra) What Color Am I?

For each of the situations below in a valid LLRB tree, indicate whether the node is red, black, or either red or black. An example of what we mean by `value` and `color`:



In the above LLRB, the node with value A has a color of red while the node with value H has a color of black.

(a) The largest value in a tree with more than one node.

(b) The smallest value in a tree with more than one node.

(c) A node whose parent is red.

(d) A node whose children are the same color.

(e) A freshly inserted node after the insertion operation is completed.

(a) The largest value in a tree with more than one node.

(a) The largest value in a tree with more than one node.

Black. The largest value is always a right child, so it must be black.

(b) The smallest value in a tree with more than one node.

(b) The smallest value in a tree with more than one node.

Either. A two node 2-3 tree has a red smallest child. A three node (insert 1,2,3) 2-3 tree has a black smallest child.

(c) A node whose parent is red.

(c) A node whose parent is red.

Black. There cannot be two consecutive red nodes

(d) A node whose children are the same color.

(d) A node whose children are the same color.

Either. A node with two children of the same color (which must be black) can be either red or black

(e) A freshly inserted node after the insertion operation is completed.

(e) A freshly inserted node after the insertion operation is completed.

Either. When the insertion operation for a BST has completed, the node may be either red or black depending on rotations and color flips.

LLRB

- Insert new elements at the leaf level
- Height has a worst case of Θ (log(N))
- All operations will take O(log(N))
- Has Red and Black branches if there is too much we “push up” the middle node
- Deletion out of scope
- Easier to Implement

BST

- Insert new elements at the leaf level
- Height will be at worst case of Θ(N)
- In the best case operations will take Θ(log(N)) but in the worst case it will take Θ(N)
- Has no splitting
- Deletion is in scope
- Easier to implement in code

B-Trees

- Insert new elements at the leaf level
- Height has a worst case of Θ (log(N))
- All operations will take O(log(N))
- Has splitting, in which case we push up the middle element
- Deletion out of scope
- Harder to implement in actual code

BST

- Insert new elements at the leaf level
- Height will be at worst case of Θ(N)
- In the best case operations will take Θ(log(N)) but in the worst case it will take Θ(N)
- Has no splitting
- Deletion is in scope
- Easier to implement in code

A Quick Overview of Disjoint Sets

```
public interface DisjointSet {
    void connect (x, y); // Connects nodes x and y (you may also see union)
    boolean isConnected(x, y); // Returns true if x and y are connected
}
```

QuickFind uses an array of integers to track which set each element belongs to.

{1, 3, 5}, {0, 2}, {4}

QuickUnion stores the parent of each node rather than the set to which it belongs and merges sets by setting the parent of one root to the other.

{3, -1, 1, -1, 2, 3} OR {3, 1, 1, 3, 2, 0}

WeightedQuickUnion does the same as QuickUnion except it decides which set is merged into which by size (merge smaller into larger), reducing stringiness.

{3, -3, 1, -3, 2, 3}

WeightedQuickUnion with Path Compression sets the parent of each node to the set's root whenever find() is called on it.

Disjoint Sets Asymptotics

```
public interface DisjointSet {
     void connect (x, y); // Connects nodes x and y (you may also see union)
     boolean isConnected(x, y); // Returns true if x and y are connected
}
```

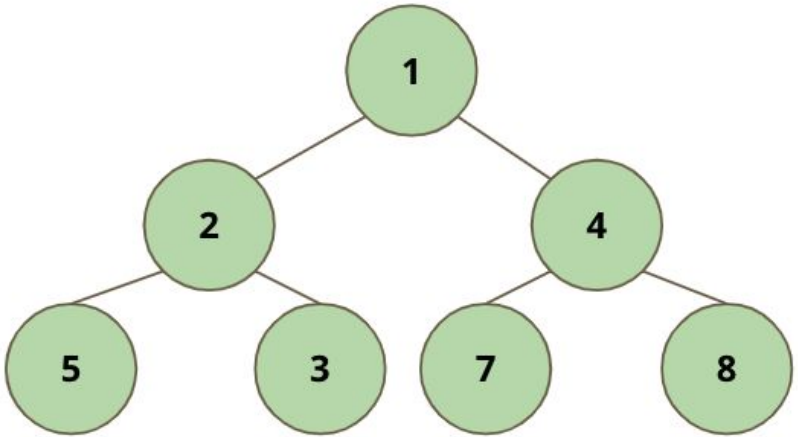
Implementation	Constructor	connect()	isConnected()
QuickUnion	Θ(N)	O(N)	O(N)
QuickFind	Θ(N)	O(N)	O(1)
Weighted Quick Union	Θ(N)	O(log N)	O(log N)
WQU with Path Compression	Θ(N)	O(log N) Θ(1)*	O(log N) Θ(1)*

* we don't really talk about QU/QF in application, more to show the asymptotic motivation for WQU

A Quick Overview of Heaps

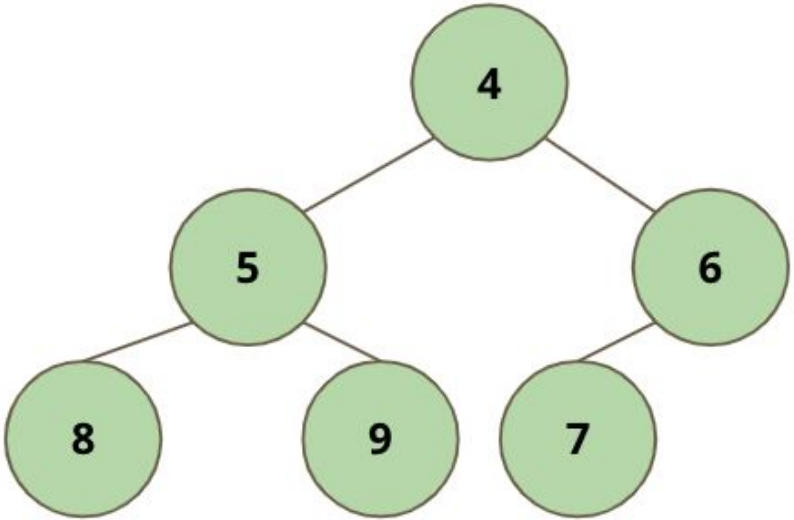
What is a heap?

- **Min-heap:** every parent is less than or equal to both of its children
- **Max-heap:** every parent is greater than or equal to both of its children



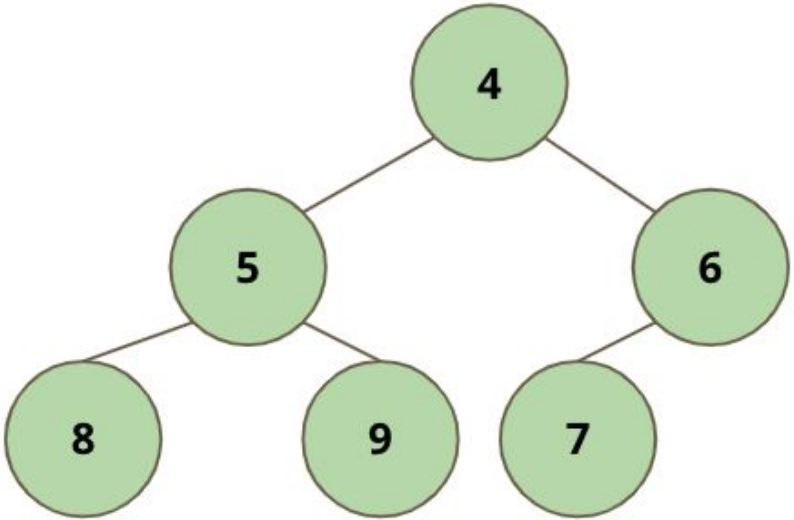
- stored in an array
- 0th index is null
- [-, 1, 2, 4, 5, 3, 7, 8]
- **what kind of tree traversal is this?**
  - level-order traversal!
- parent of arr[i] = arr[i/2]
- heaps can be used to implement priority queues

How the fricken frack do we insert?



let’s say we insert **1**:

How the fricken frack do we insert?

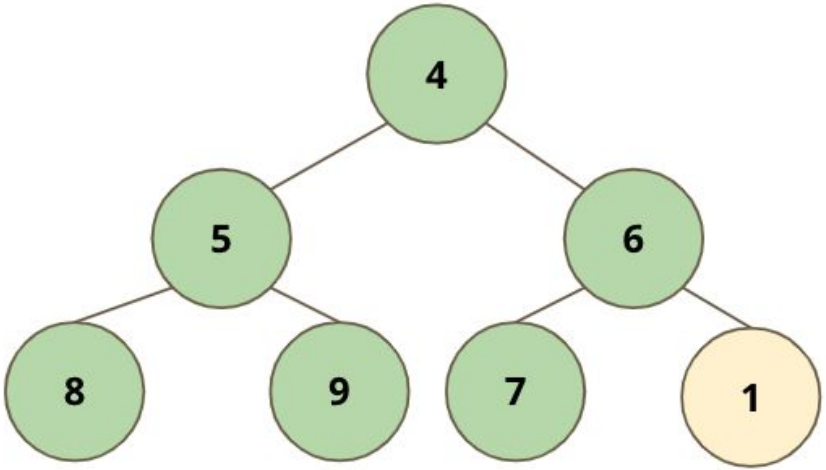


Min heap!

[-, 4, 5, 6, 8, 9, 7]

How the fricken frack do we insert?

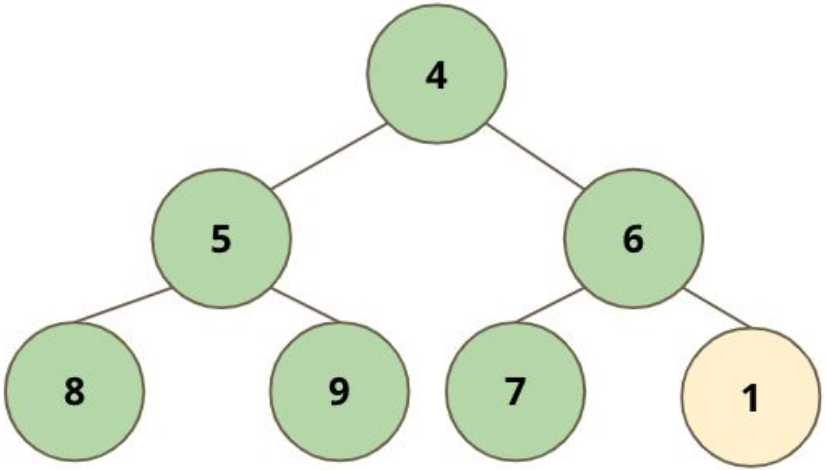
1. Add 1 to end of array and end of heap



[-, 4, 5, 6, 8, 9, 7, 1]

How the fricken frack do we insert?

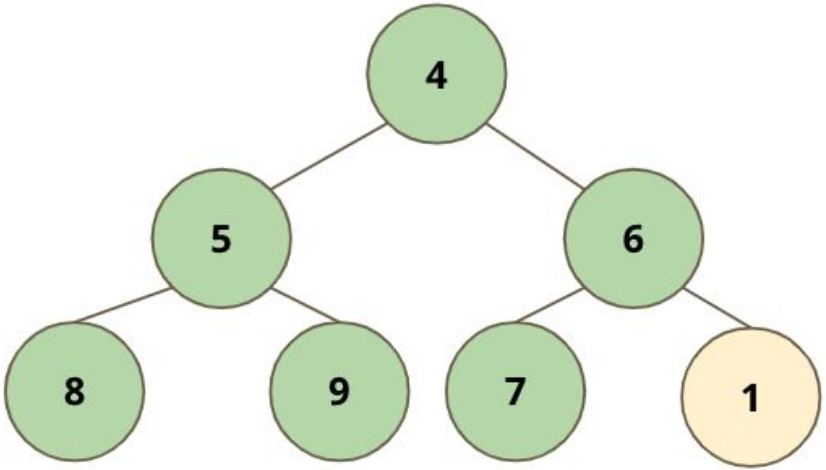
1. Add 1 to end of array and end of heap
2. Check if added element satisfies heap condition (min p <= c; max p>= c)



[-, 4, 5, 6, 8, 9, 7, 1]

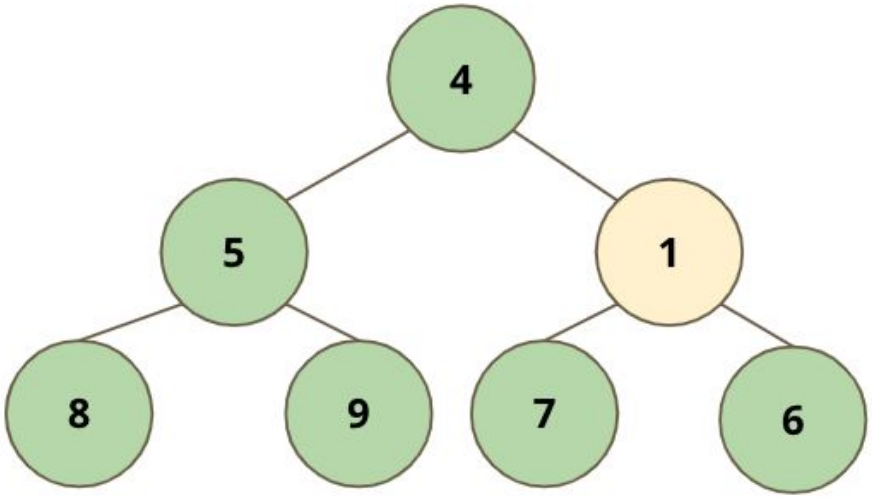
How the fricken frack do we insert?

1. Add 1 to end of array and end of heap
2. Check if added element satisfies hea



How the fricken frack do we insert?

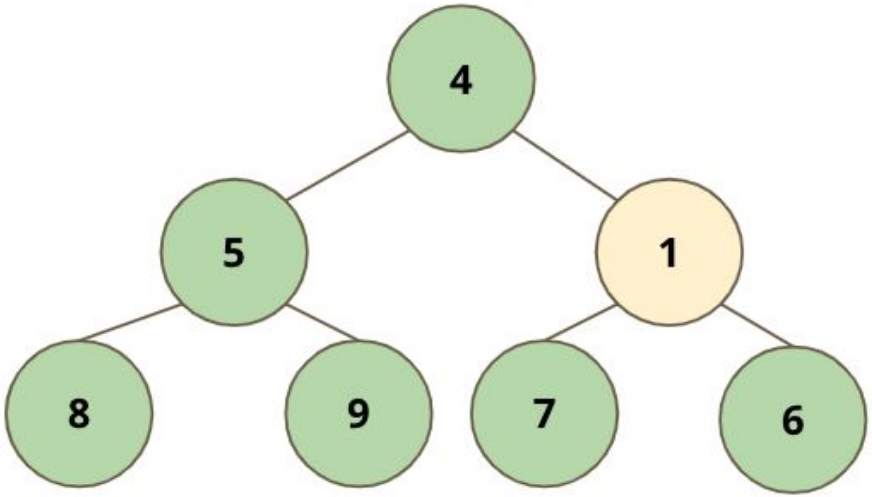
1. Add 1 to end of array and end of heap
2. Check if added element satisfies heap condition (min p <= c; max p>= c)
3. If it does not satisfy, swap child and parent in heap and array



[-, 4, 5, 1, 8, 9, 7, 6]

How the fricken frack do we insert?

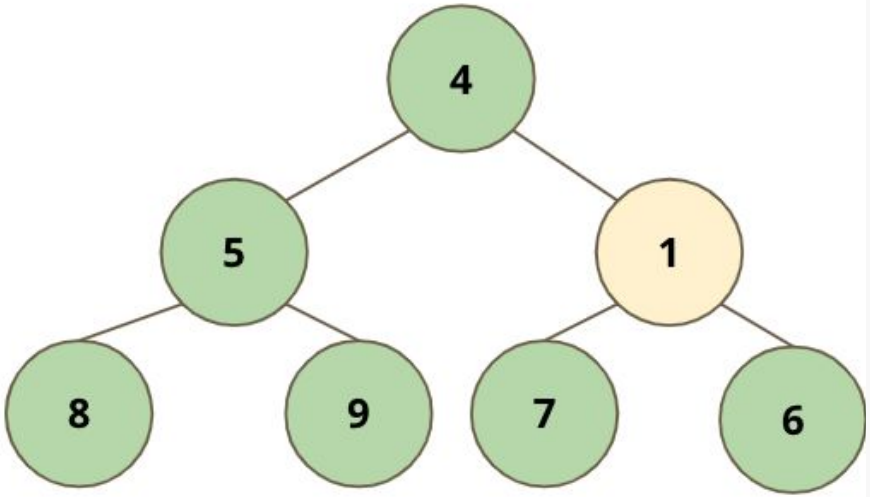
1. Add 1 to end of array and end of heap
2. Check if added element satisfies heap condition (min p <= c; max p>= c)
3. If it does not satisfy, swap child and parent in heap and array
4. Repeat step 2 and 3 until no more swaps are needed



[-, 4, 5, 1, 8, 9, 7, 6]

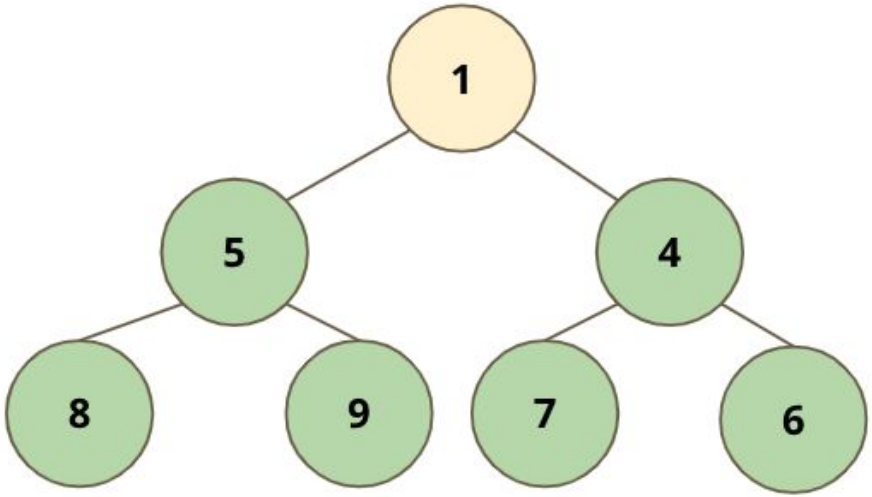
How the fricken frack do we insert?

1. Add 1 to end of array and end of heap
2. Check if added element satisfies hea
3. If it does not satisfy, swap child and p
4. Repeat step 2 and 3 until no more sv



How the fricken frack do we insert?

1. Add 1 to end of array and end of heap
2. Check if added element satisfies heap condition (min p <= c; max p>= c)
3. If it does not satisfy, swap child and parent in heap and array
4. Repeat step 2 and 3 until no more swaps are needed



[-, 1, 5, 4, 8, 9, 7, 6]

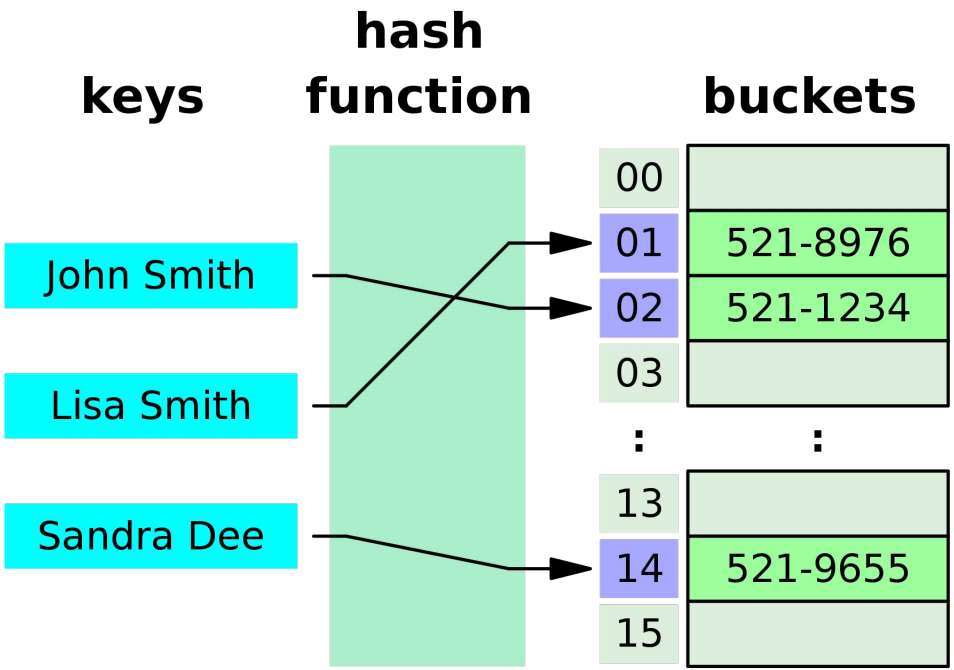
Heap asymptotics!

Operation	Worst case	Best case
insert	Θ(logN)	Θ(1)
findMin/Max	Θ(1)	Θ(1)
removeMin/ Max	Θ(logN)	Θ(1)

A Quick Overview of Hashing

Whats Hashenning?

- Normally, you would have to iterate through the data structure to find your element, but now you can go straight to it
- Hash function: what we use to decide the “bucket” it goes into
    - Find hashcode for our object, modulo by bucket size
        - Item.hashcode % numBuckets
- Some complications arise from collisions and overflow entries, but we solve that using external chaining
- Resizing with load factor: N = number of elements, M = number of buckets
- When N / M >= loadfactor,
- increase the number of buckets
- rehash all the elements and put back into new set of buckets



Hash, Mash or Pass?

What makes a valid hashcode?

- Must be an integer
- Hashcode for same object must be consistent
- If two objects are “equal”, they have the same hashcode
    - Also true for reverse

What makes a good hashcode?

- Distributes elements evenly

Hash, Mash or Pass?

```java
public int hashCode(String name) {
    return 2023;
}

public int hashCode(String name) {
    return (int) name.charAt(0) // ASCII value of the first char in NAME
}

public int hashCode(String name) {
    return size;
}

public int hashCode(String name) {
    return name.hashCode();
}
```

Hash, Mash or Pass?

```
public int hashCode(String name)
    return 2023;
}
```

Has all qualities for valid hashcode, but always results in collisions

```
public int hashCode(String name) {
    return (int) name.charAt(0) // ASCII value of the first char in NAME
}

public int hashCode(String name) {
    return size;
}

public int hashCode(String name) {
    return name.hashCode();
}
```

Hash, Mash or Pass?

```
public int hashCode(String name)
    return 2023;
}
```

Has all qualities for valid hashcode, but always results in collisions

```
public int hashCode(String name) {
    return (int) name.charAt(0) // ASCII value of the first char in NAME
}
```

Valid but if there are many names starting with the same letter we would get many collisions.

```
public int hashCode(String name) {
    return size;
}
```

```
public int hashCode(String name) {
    return name.hashCode();
}
```

Hash, Mash or Pass?

```
public int hashCode(String name)
    return 2023;
}
```

Has all qualities for valid hashcode, but always results in collisions

```
public int hashCode(String name) {
    return (int) name.charAt(0) // ASCII value of the first char in NAME
}
```

Valid but if there are many names starting with the same letter we would get many collisions.

```
public int hashCode(String name) {
    return size;
}
```

Size is always changing so a name won't always have the same hash code.

```
public int hashCode(String name) {
    return name.hashCode();
}
```

Hash, Mash or Pass?

```
public int hashCode(String name)
    return 2023;
}
```

Has all qualities for valid hashcode, but always results in collisions

```
public int hashCode(String name) {
    return (int) name.charAt(0) // ASCII value of the first char in NAME
}
```

Valid but if there are many names starting with the same letter we would get many collisions.

```
public int hashCode(String name) {
    return size;
}
```

Size is always changing so a name won't always have the same hash code.

```
public int hashCode(String name) {
    return name.hashCode();
}
```

Java's built-in String hash function may be used here, which we know is fairly unique for most strings

Cooking with Asymptotics

Worst case (lots of collisions)	Best case (items perfectly evenly distributed)
Θ(N)	Θ(1)

Runtime is based off how good your hashcode is!

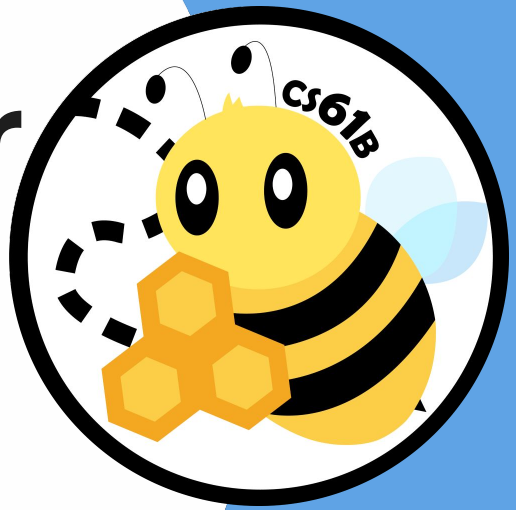
Quick Overview

- This is a lot(in fact like most of the class) so don't feel too overwhelmed by the amount of content take it one step at a time
- The main approach for examine to know the rules of each data structure and decide which one may be best at which time!
- Also it's just a test! I doesn't define you, your worth or what you have learned!

Final Review Session

Thank you for coming!

There is no data structure with the space complexity to hold your potential



Graphs!

with Claire & Jay!

Trees

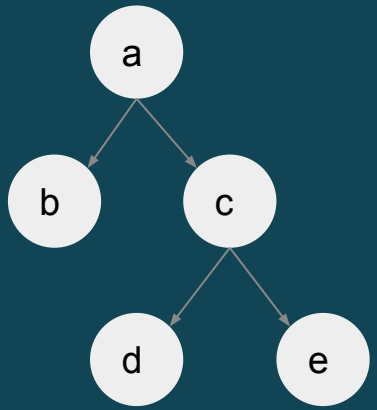
Trees are structures that follow a few basic rules:

1. If there are N nodes, there are N-1 edges
2. There is exactly 1 path from every node to every other node (assuming the tree is undirected)
3. The above two rules means that trees are fully connected and contain no cycles

A parent node points towards its child.

The root of a tree is a node with no parent nodes.

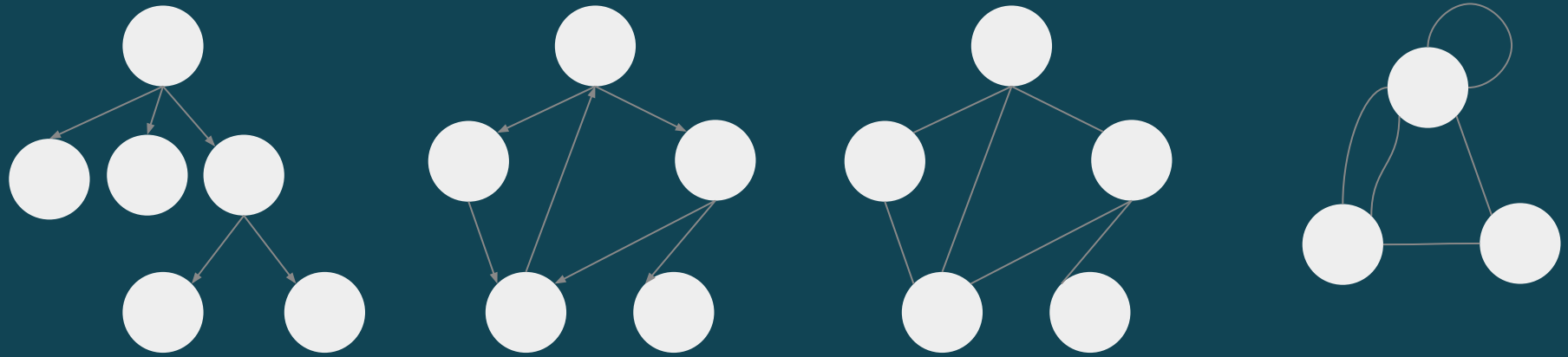
A leaf of a tree is a node with no child nodes.



Graphs

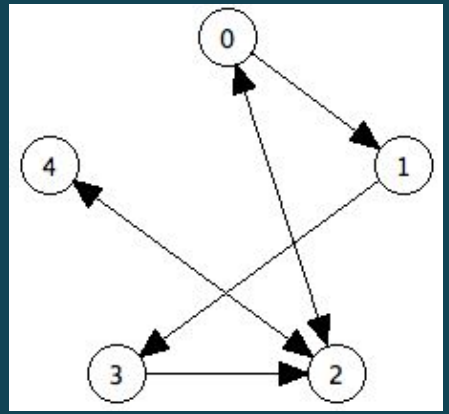
Graphs are similar to what we know about trees but more general!

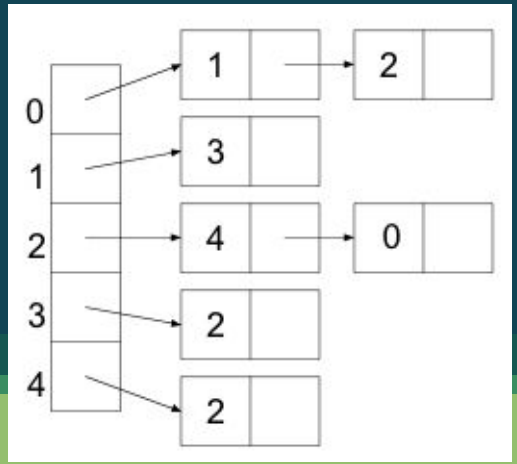
1. Graphs allow cycles
2. Simple graphs don't allow parallel edges (2 or more edges connecting the same two nodes) or self edges (an edge from a vertex to itself)
3. Graphs may be directed or undirected



Representing Graphs - Adjacency List

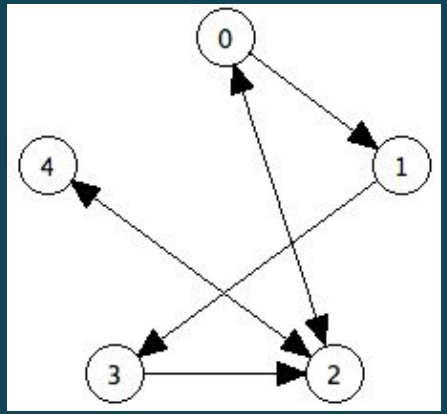
- Each position in the array represents a **vertex** in the graph
- Each of these positions point to a **list**
- The lists are called **adjacency lists** = each element in the list represents a neighbor of the vertex





Representing Graphs - Adjacency Matrix

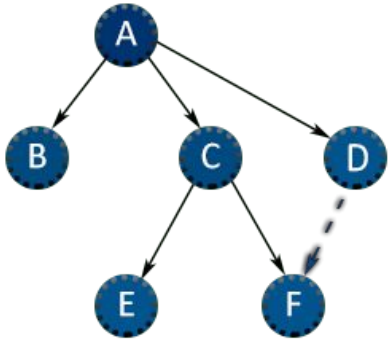
- Each vertex has a row and a column in the matrix
- The value in that table says true or false whether or not that edge exists.



	0	1	2	3	4
0	F	T	T	F	F
1	F	F	F	T	F
2	T	F	F	F	T
3	F	F	T	F	F
4	F	F	T	F	F

Breadth First Search (BFS)

BFS = visiting nodes based off of their distance to the source, or starting point. For graphs, we visit all of our immediate children before continuing on to any of our grandchildren



A B C D E F

- Level order traversal
- Visit the nearest level of unvisited neighbors before going deeper.

Pre-order Depth First Search (DFS)

DFS = exploring a neighbor's entire subgraph before moving on to the next neighbor

Preorder:

For each node,

1. visit the node
2. recursively visit each neighbor.

```
public void dfsPreorder(Node n) {
    visit(n);
    for (Node neighbor : n.neighbors()) {
        if (!neighbor.visited) {
            dfsPreorder(neighbor);
        }
    }
}
```

Post-order Depth First Search (DFS)

DFS = exploring a neighbor's entire subgraph before moving on to the next neighbor

Postorder:

For each node,

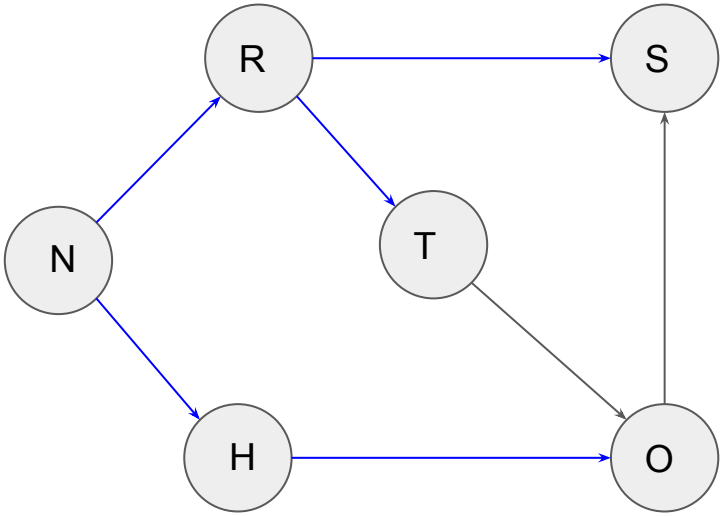
1. recursively visit each neighbor.
2. visit the node

```
public void dfsPostorder(Node n) {
    for (Node neighbor : n.neighbors()) {
        if (!neighbor.visited) {
            dfsPostorder(neighbor);
        }
    }
    visit(n);
}
```

Shortest Path Tree (SPT) of a Graph

- “Tree” = Acyclic
- “Shortest Path” = Minimum Total Distance traveled between starting vertex and the target vertices
- SPT: |V| – 1 edges

- **Shortest Unweighted Paths**: BFS
- **Shortest Weighted Paths**: Dijkstra, A*



SPT

Dijkstra's Algorithm

*messes up with negative edges

- Outputs the shortest path tree from a starting vertex to **all** other vertices in the graph

1. Create a priority queue.
2. Add starting vertex to the priority queue with priority 00. Add all other vertices to the priority queue with priority ∞∞.
3. While the priority queue is not empty: pop a vertex out of the priority queue:
    - When you visit vertex v:
        - For each edge from v to w, assume we take that edge for now, and write that distance.
        - If we find a better path, update our preferred edge and distance (this is known as "relaxing" the edge).

Hug's Example

A*

- finds the shortest path between an initial and a final point

Difference from Dijkstra's:

1. Has a goal vertex G; Uses heuristics to guide toward the goal
2. Different “priority” for a vertex V: distance already traveled + estimated travel distance in the future (heuristic)
3. Not every vertex will be “visited”

Hug Demo

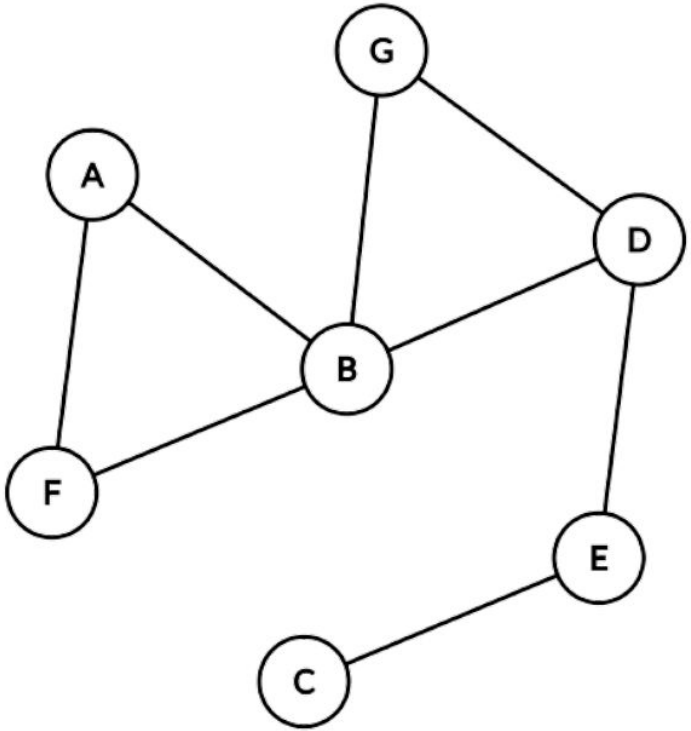
A* Heuristic

- An estimated distance between current vertex to goal vertex

- ”Good” heuristics
  - **Admissible**: always underestimates the total travel distance
  - **Consistent**: always underestimates both the total travel distance and the travel distance between any two “stops” (vertices).

5 Graph Traversals

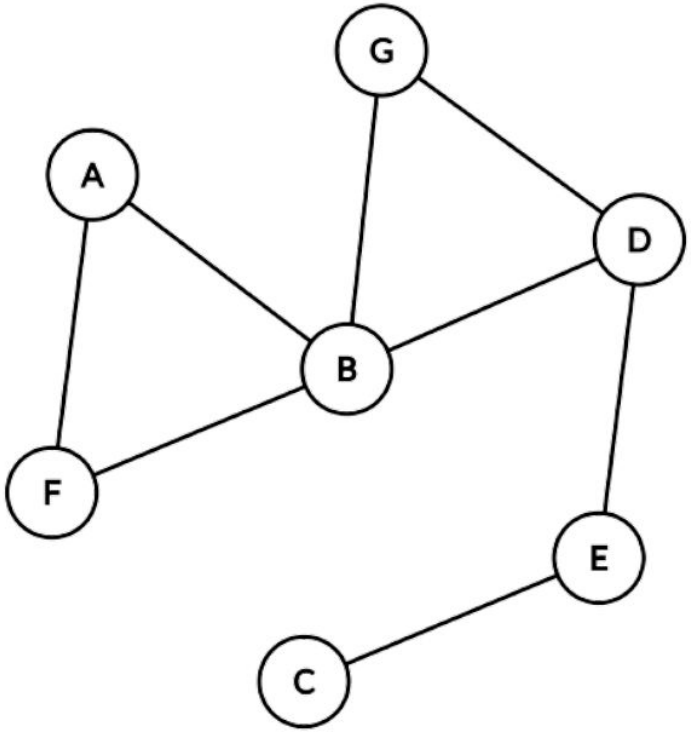
We are given the following undirected graph:



(a) Write the nodes visited in preorder DFS traversal starting from A. Break ties alphabetically.

5 Graph Traversals

We are given the following undirected graph:

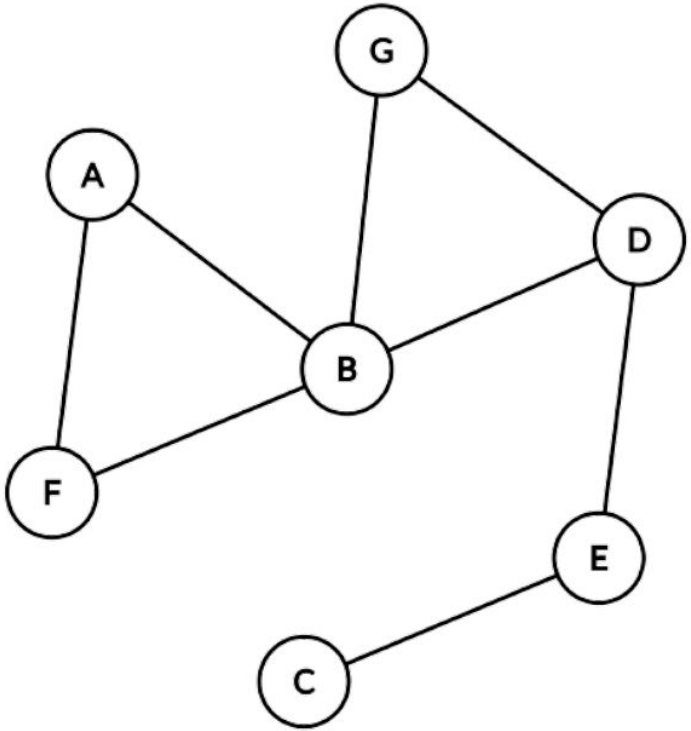


(a) Write the nodes visited in preorder DFS traversal starting from A. Break ties alphabetically.

A, B, D, E, C, G, F

5 Graph Traversals

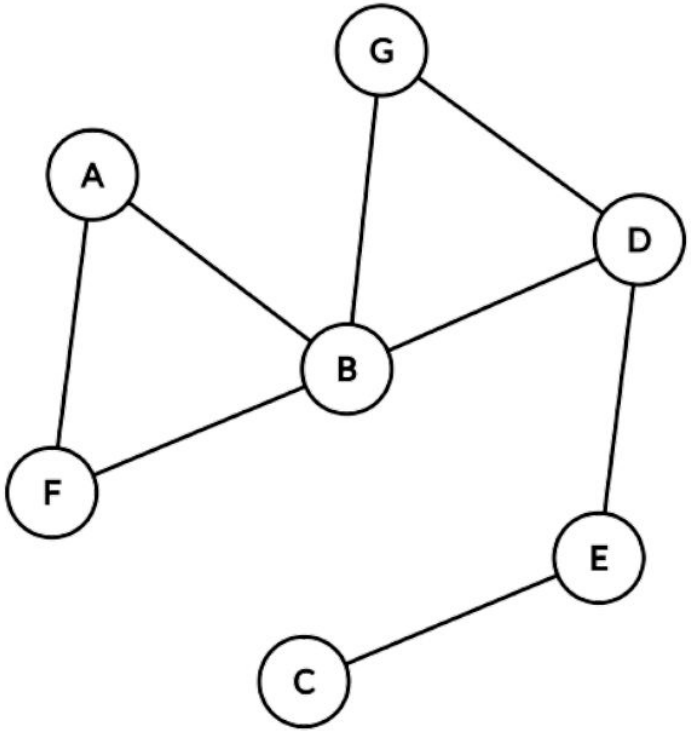
We are given the following undirected graph:



(b) Write the nodes visited in postorder DFS traversal starting from A. Break ties alphabetically.

5 Graph Traversals

We are given the following undirected graph:

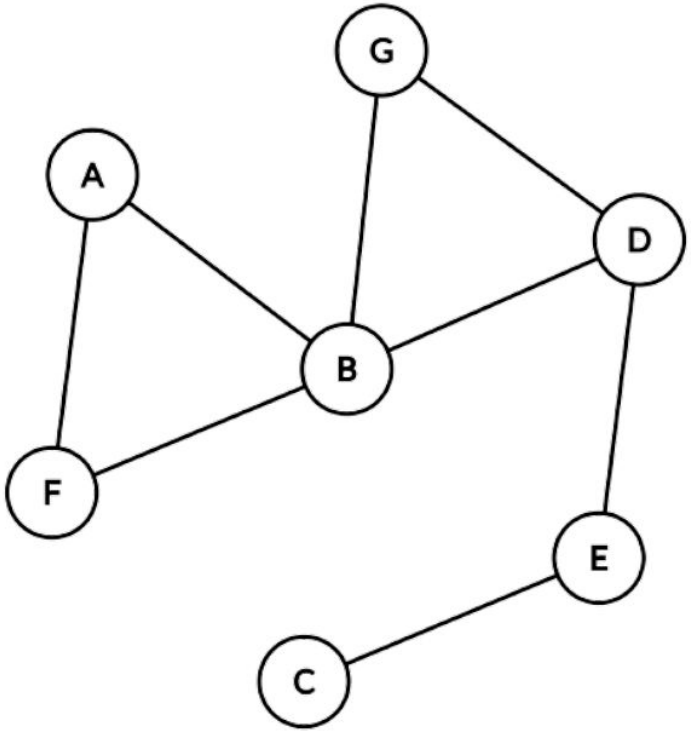


(b) Write the nodes visited in postorder DFS traversal starting from A. Break ties alphabetically.

C, E, G, D, F, B, A

5 Graph Traversals

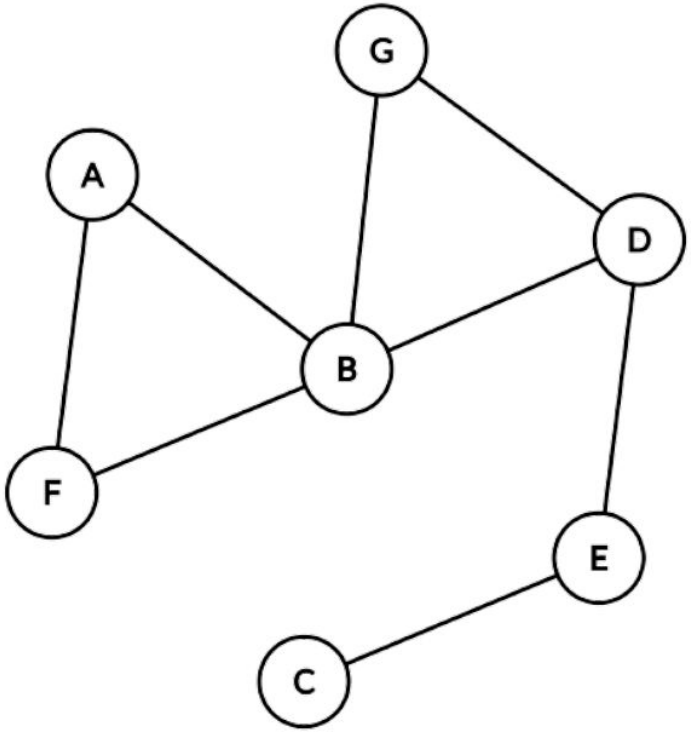
We are given the following undirected graph:



(c) Write the nodes visited in BFS traversal starting from A. Break ties alphabetically.

5 Graph Traversals

We are given the following undirected graph:

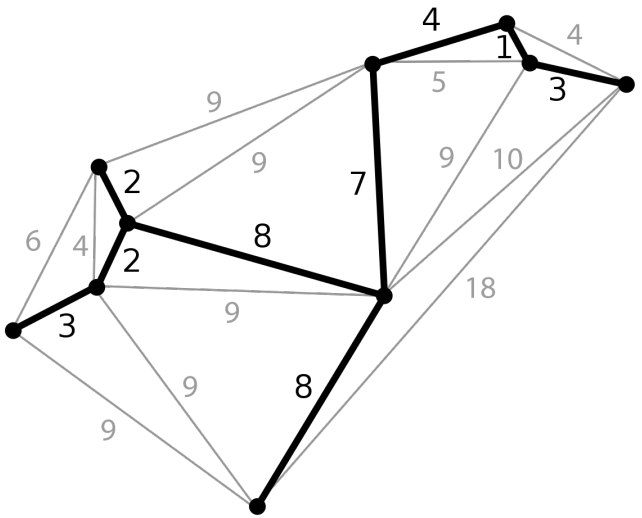


(c) Write the nodes visited in BFS traversal starting from A. Break ties alphabetically.

A, B, F, D, G, E, C

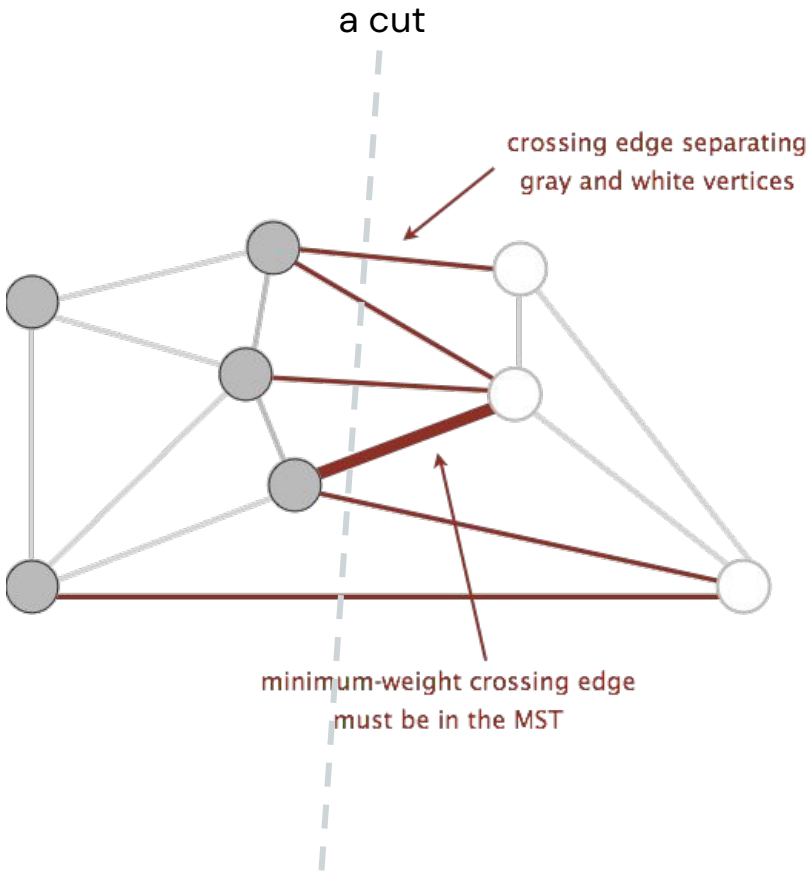
Minimum Spanning Trees (MSTs)

- A **subset of edges** on an undirected graph which connect **all the vertices** together using **minimum total edge weight**
- # edges in MST = V – 1
- Not necessarily unique! (break ties alphabetically, unless stated otherwise)
- Should not be confused with SPT (not same in general)



Cut Property

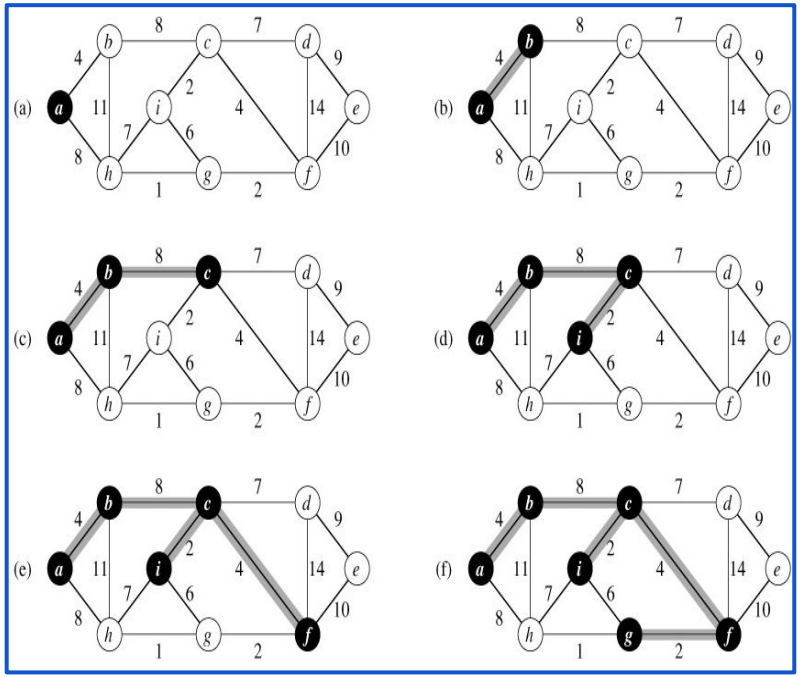
- **Cut:** an assignment of a graph's nodes to two non-empty sets
- **Crossing edge:** an edge that connects the two sets in a cut, it "crosses" a cut.
- Given any cut, **minimum weight crossing edge** is in the MST



Prim's Algorithm

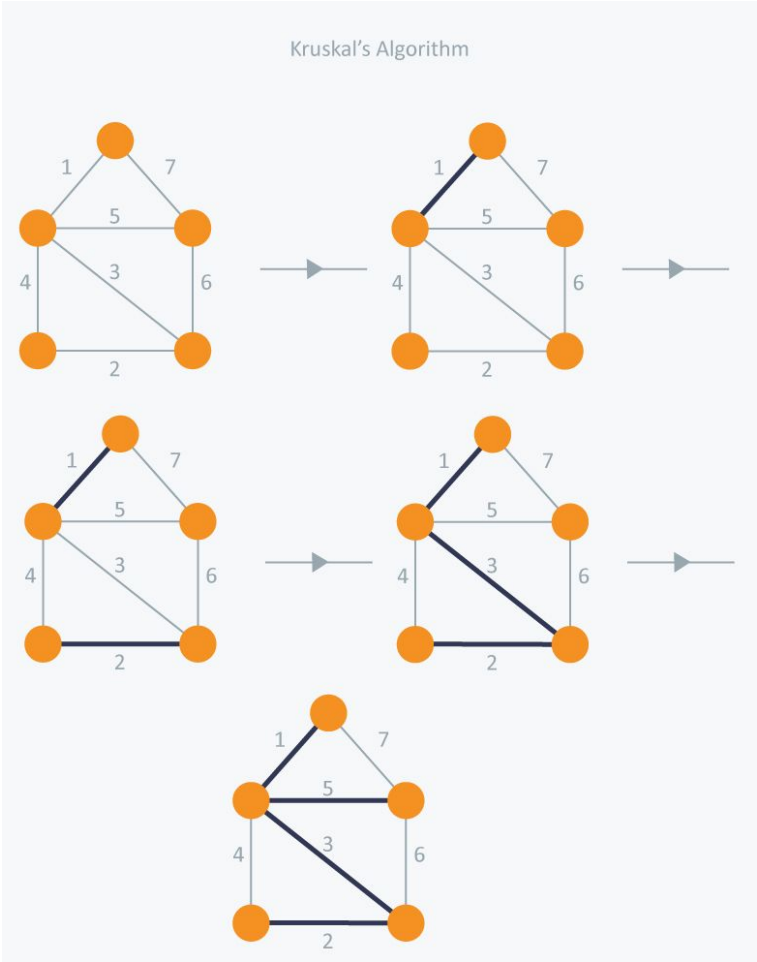
Demo Demo with PQ

- Choose a start node (usually provided)
- Continue adding neighboring edges with the lowest weight to MST
  - Neighboring: edge connects one node in the MST with one not in the MST (no cycles allowed!)
- Repeat until all vertices are connected (# edges = V - 1)
- Data structure: **Priority queue**
- O(E log V)



Kruskal's Algorithm

- Continue adding edge with lowest weight, skipping if it creates a cycle
- Repeat until all nodes are connected (# edges = V – 1)
- Data structure: **WQU** (or any Disjoint Set)
- O(E log E)



Demo Demo with WQU

6 MST Adaptations

For each subpart below, determine whether the algorithm presented on the described graph will always, sometimes, or never find a valid MST. Assume the original graph is

- simple
- connected
- undirected
- has **at least one cycle**
- and **the weights of all the edges are distinct**.

Assume ties are broken randomly in each algorithm presented. Recall that an MST is simply a set of edges. Parts are independent.

a) For this part, additionally assume the original graph has at least one negative edge. Run Kruskal's as normal.

◯ Always finds a valid MST.
◯ Sometimes finds a valid MST.
◯ Never finds a valid MST.

6 MST Adaptations

For each subpart below, determine whether the algorithm presented on the described graph will always, sometimes, or never find a valid MST. Assume the original graph is

- simple
- connected
- undirected
- has **at least one cycle**
- and **the weights of all the edges are distinct**.

Assume ties are broken randomly in each algorithm presented. Recall that an MST is simply a set of edges. Parts are independent.

a) For this part, additionally assume the original graph has at least one negative edge. Run Kruskal's as normal.

◯ Always finds a valid MST.
◯ Sometimes finds a valid MST.
◯ Never finds a valid MST.

Solution:
√ **Always finds a valid MST.**
◯ Sometimes finds a valid MST.
◯ Never finds a valid MST.

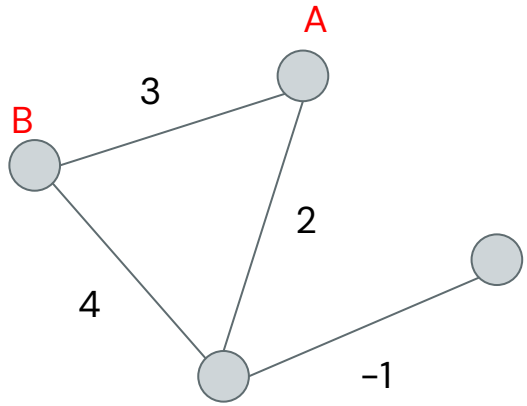
MST algorithms aren't affected by having negative edges.

b) For this part, additionally assume the graph has **exactly one cycle**. Run DFS from a random vertex breaking ties by edge weight. Output the DFS tree.

- ◯ Always finds a valid MST.
- ◯ Sometimes finds a valid MST.
- ◯ Never finds a valid MST.

b) For this part, additionally assume the graph has **exactly one cycle**. Run DFS from a random vertex breaking ties by edge weight. Output the DFS tree.

- ◯ Always finds a valid MST.
- ◯ Sometimes finds a valid MST.
- ◯ Never finds a valid MST.



Solution:

- ◯ Always finds a valid MST.
- √ **Sometimes finds a valid MST.**
- ◯ Never finds a valid MST.

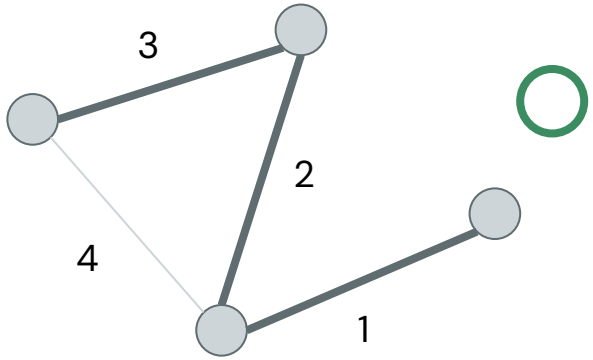
Starting from A gives {−1, 2, 4} while starting from B gives {−1, 2, 3} (MST).

c) Run Dijkstra's from every vertex in the graph for **one step** so that from each vertex we have found *one* edge. Add these edges to a set. Output the set.

- ◯ Always finds a valid MST.
- ◯ Sometimes finds a valid MST.
- ◯ Never finds a valid MST.

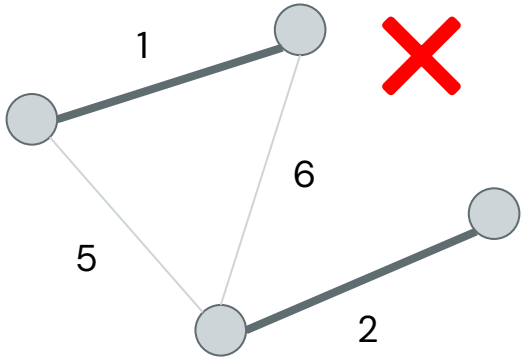
c) Run Dijkstra's from every vertex in the graph for **one step** so that from each vertex we have found *one* edge. Add these edges to a set. Output the set.

- ○ Always finds a valid MST.
- ○ Sometimes finds a valid MST.
- ○ Never finds a valid MST.



Solution:

- ○ Always finds a valid MST.
- √ **Sometimes finds a valid MST.**
- ○ Never finds a valid MST.



Graph could become disconnected (if more than one pair choose each other)

Each of the remaining subparts will *modify* the original graph, e.g. adding edges or changing edge weights. Your task is to determine whether the algorithm presented on the modified graph will always, sometimes, or never find a valid MST of the **original, unmodified graph**. Assume the modifications are **in addition** to the assumptions listed in the beginning. Parts are independent.

d) Find the minimal edge **M** in the graph, and modify the original graph by adding |**M**| (the edge weight of **M**) to all edges in graph. Run Kruskal's on modified graph.

- ◯ Always finds a valid MST of the original, unmodified graph.
- ◯ Sometimes finds a valid MST of the original, unmodified graph.
- ◯ Never finds a valid MST of the original, unmodified graph.

Each of the remaining subparts will *modify* the original graph, e.g. adding edges or changing edge weights. Your task is to determine whether the algorithm presented on the modified graph will always, sometimes, or never find a valid MST of the **original, unmodified graph**. Assume the modifications are **in addition** to the assumptions listed in the beginning. Parts are independent.

d) Find the minimal edge **M** in the graph, and modify the original graph by adding |**M**| (the edge weight of **M**) to all edges in graph. Run Kruskal's on modified graph.

- ◯ Always finds a valid MST of the original, unmodified graph.
- ◯ Sometimes finds a valid MST of the original, unmodified graph.
- ◯ Never finds a valid MST of the original, unmodified graph.

Solution:

- √ **Always finds a valid MST of the original, unmodified graph.**
- ◯ Sometimes finds a valid MST of the original, unmodified graph.
- ◯ Never finds a valid MST of the original, unmodified graph.

Adding |M| to all edges doesn't change priority (imagine running Kruskal's)

e) For this part, additionally assume the original graph has at least one negative edge. Modify the original graph by multiplying each **negative** edge by -1. Run Kruskal's on the modified graph.

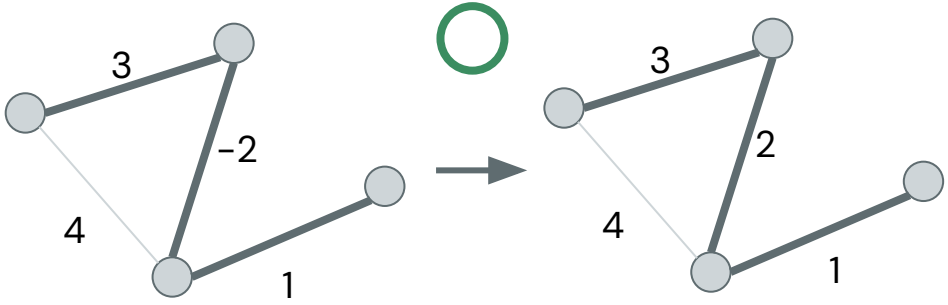
- ○ Always finds a valid MST of the original, unmodified graph.
- ○ Sometimes finds a valid MST of the original, unmodified graph.
- ○ Never finds a valid MST of the original, unmodified graph.

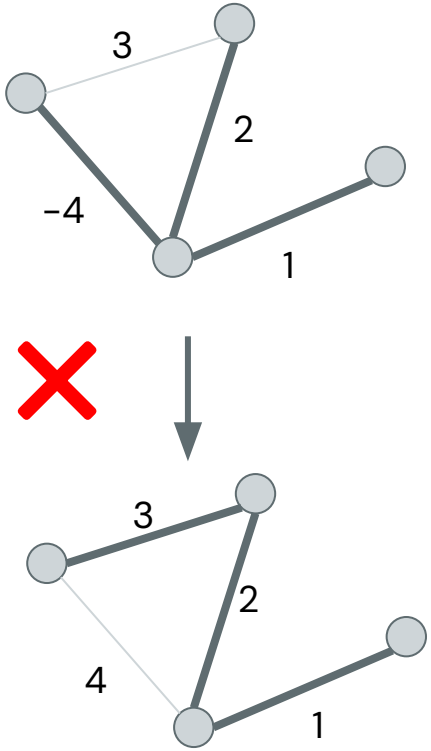
e) For this part, additionally assume the original graph has at least one negative edge. Modify the original graph by multiplying each **negative** edge by -1. Run Kruskal's on the modified graph.

- ○ Always finds a valid MST of the original, unmodified graph.
- ○ Sometimes finds a valid MST of the original, unmodified graph.
- ○ Never finds a valid MST of the original, unmodified graph.

Solution:

- ○ Always finds a valid MST of the original, unmodified graph.
- √ **Sometimes finds a valid MST of the original, unmodified graph.**
- ○ Never finds a valid MST of the original, unmodified graph.





This depends on the magnitude of the negative edges.

f) For this part, additionally assume the original graph is fully connected *and* every vertex is incident to at least one negative edge. Recall a fully connected graph means an edge exists between every pair of vertices. Modify the original graph by multiplying each **negative** edge by -1. Run Kruskal's on the modified graph.

- ◯ Always finds a valid MST of the original, unmodified graph.
- ◯ Sometimes finds a valid MST of the original, unmodified graph.
- ◯ Never finds a valid MST of the original, unmodified graph.

g) For this part, additionally assume the original graph is fully connected *and* every vertex is incident to at least one negative edge. Modify the original graph by multiplying each edge by -1. Run Kruskal's on the modified graph.

- ◯ Always finds a valid MST of the original, unmodified graph.
- ◯ Sometimes finds a valid MST of the original, unmodified graph.
- ◯ Never finds a valid MST of the original, unmodified graph.

f) For this part, additionally assume the original graph is fully connected *and* every vertex is incident to at least one negative edge. Recall a fully connected graph means an edge exists between every pair of vertices. Modify the original graph by multiplying each **negative** edge by -1. Run Kruskal's on the modified graph.

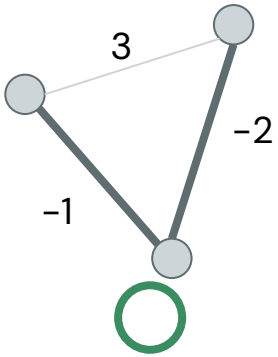
Solution:

- ◯ Always finds a valid MST of the original, unmodified graph.
- √ **Sometimes finds a valid MST of the original, unmodified graph.**
- ◯ Never finds a valid MST of the original, unmodified graph.

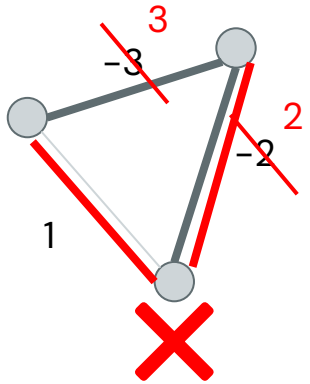
g) For this part, additionally assume the original graph is fully connected *and* every vertex is incident to at least one negative edge. Modify the original graph by multiplying each edge by -1. Run Kruskal's on the modified graph.

Solution:

- ◯ Always finds a valid MST of the original, unmodified graph.
- ◯ Sometimes finds a valid MST of the original, unmodified graph.
- √ **Never finds a valid MST of the original, unmodified graph.**



vs



f) again depends on magnitude.

g) flipping signs of all edges = min becomes max.

h) For this part, additionally assume the graph isn't fully connected. Modify the graph by the following: for every pair of vertices without an edge between them, add an edge with weight equal to the weight of the **largest** edge. Run Kruskal's on the modified graph.

- ◯ Always finds a valid MST of the original, unmodified graph.
- ◯ Sometimes finds a valid MST of the original, unmodified graph.
- ◯ Never finds a valid MST of the original, unmodified graph.

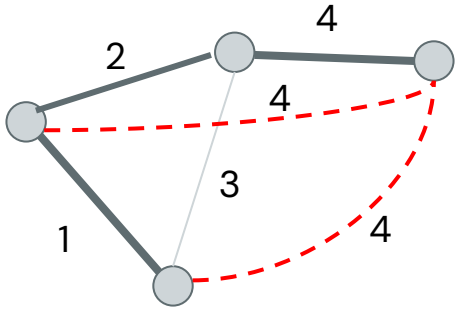
i) For this part, additionally assume the graph isn't fully connected. Modify the graph by the following: for every pair of vertices without an edge between them, add an edge with weight equal to the weight of the **smallest** edge. Run Kruskal's on the modified graph.

- ◯ Always finds a valid MST of the original, unmodified graph.
- ◯ Sometimes finds a valid MST of the original, unmodified graph.
- ◯ Never finds a valid MST of the original, unmodified graph.

h) For this part, additionally assume the graph isn't fully connected. Modify the graph by the following: for every pair of vertices without an edge between them, add an edge with weight equal to the weight of the **largest** edge. Run Kruskal's on the modified graph.

Solution:

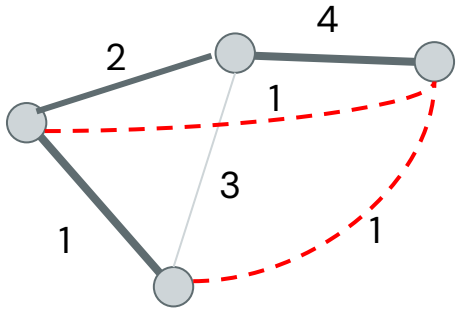
- ○ Always finds a valid MST of the original, unmodified graph.
- √ **Sometimes finds a valid MST of the original, unmodified graph.**
- ○ Never finds a valid MST of the original, unmodified graph.



i) For this part, additionally assume the graph isn't fully connected. Modify the graph by the following: for every pair of vertices without an edge between them, add an edge with weight equal to the weight of the **smallest** edge. Run Kruskal's on the modified graph.

Solution:

- ○ Always finds a valid MST of the original, unmodified graph.
- ○ Sometimes finds a valid MST of the original, unmodified graph.
- √ **Never finds a valid MST of the original, unmodified graph.**



h) could end up choosing newly added edge over original due to tiebreaking

g) newly added edges will always be prioritized & chosen (imagine running Kruskal's)