



SC2001/CE2101/CZ2101: Algorithm Design and Analysis

Introduction to Sorting

Instructor: Asst. Prof. LIN Shang-Wei

Courtesy of Dr. Ke Yiping, Kelly's slides

Learning Objectives

At the end of this lecture, students should be able to:

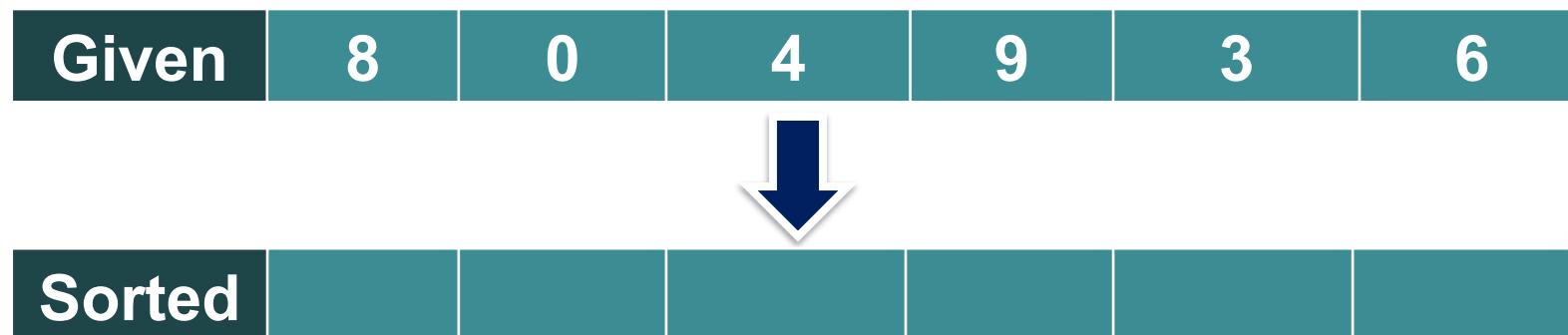
- Define what sorting is
- Explain why we learn sorting
- Analyse the objective and evaluation of sorting algorithms

What is Sorting?

Definition (sorting in ascending order):

- Given a set of records r_1, r_2, \dots, r_n with key values k_1, k_2, \dots, k_n , arrange records in order s such that records $r_{s1}, r_{s2}, \dots, r_{sn}$ have keys with property $k_{s1} \leq k_{s2} \leq \dots \leq k_{sn}$.

Example:

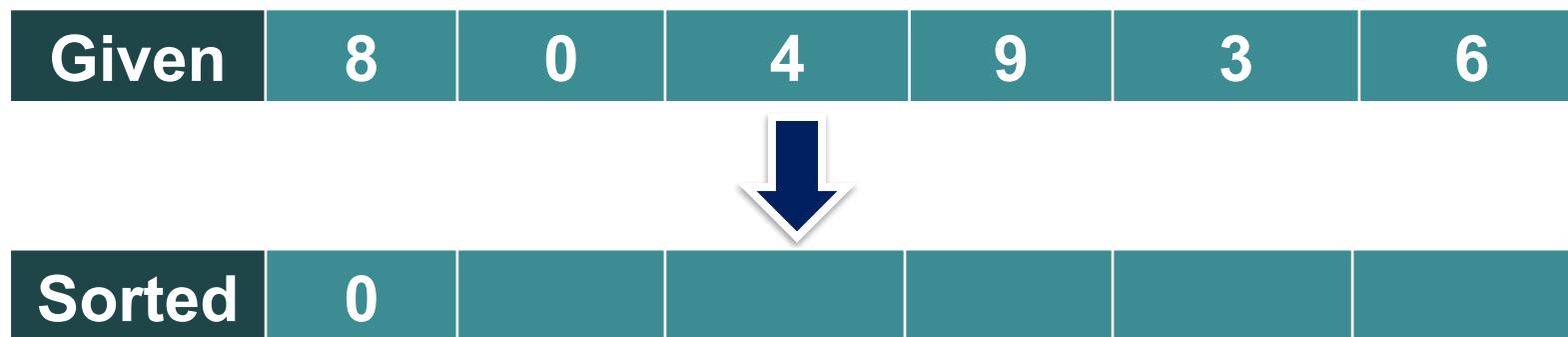


What is Sorting?

Definition (sorting in ascending order):

- Given a set of records r_1, r_2, \dots, r_n with key values k_1, k_2, \dots, k_n , arrange records in order s such that records $r_{s1}, r_{s2}, \dots, r_{sn}$ have keys with property $k_{s1} \leq k_{s2} \leq \dots \leq k_{sn}$.

Example:

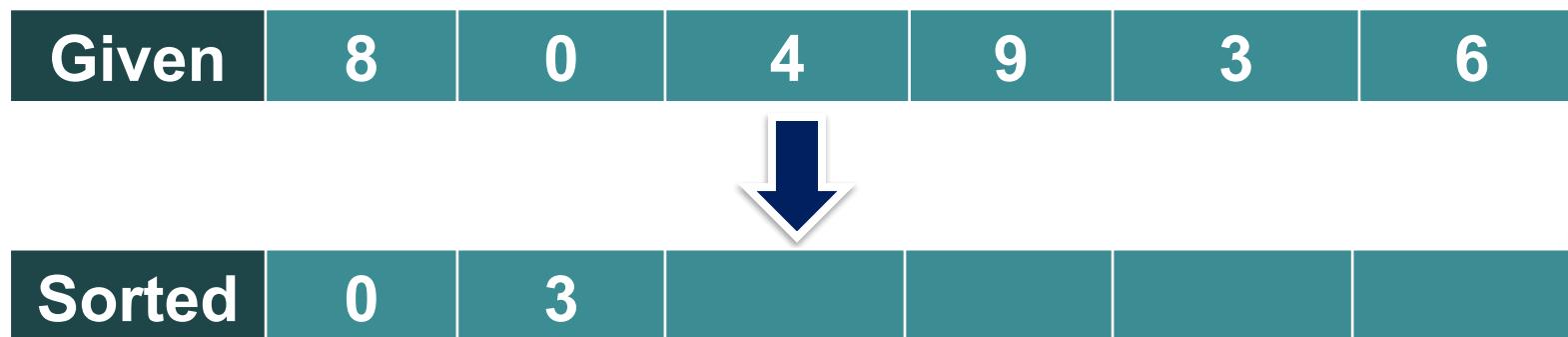


What is Sorting?

Definition (sorting in ascending order):

- Given a set of records r_1, r_2, \dots, r_n with key values k_1, k_2, \dots, k_n , arrange records in order s such that records $r_{s1}, r_{s2}, \dots, r_{sn}$ have keys with property $k_{s1} \leq k_{s2} \leq \dots \leq k_{sn}$.

Example:

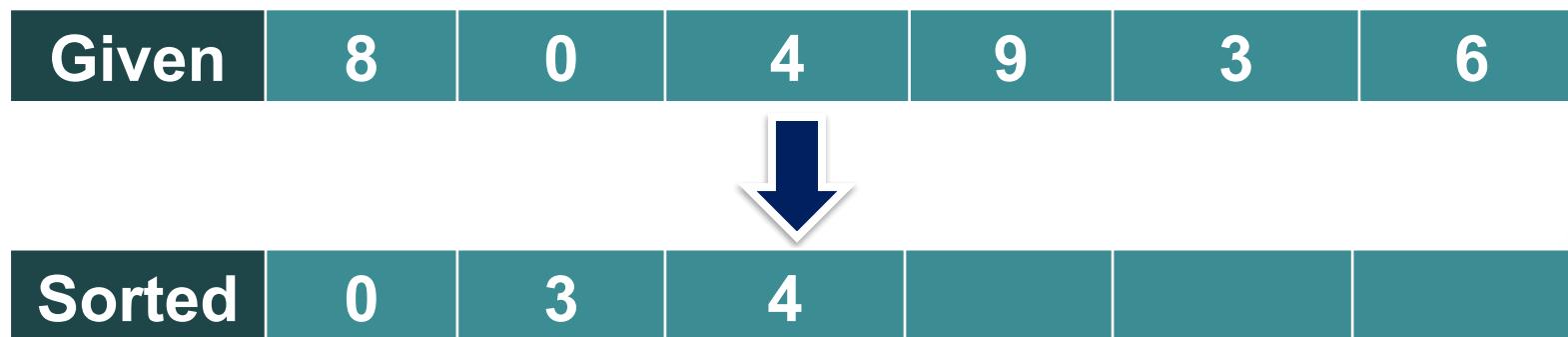


What is Sorting?

Definition (sorting in ascending order):

- Given a set of records r_1, r_2, \dots, r_n with key values k_1, k_2, \dots, k_n , arrange records in order s such that records $r_{s1}, r_{s2}, \dots, r_{sn}$ have keys with property $k_{s1} \leq k_{s2} \leq \dots \leq k_{sn}$.

Example:



What is Sorting?

Definition (sorting in ascending order):

- Given a set of records r_1, r_2, \dots, r_n with key values k_1, k_2, \dots, k_n , arrange records in order s such that records $r_{s1}, r_{s2}, \dots, r_{sn}$ have keys with property $k_{s1} \leq k_{s2} \leq \dots \leq k_{sn}$.

Example:

| Given | 8 | 0 | 4 | 9 | 3 | 6 |
|--------|---|---|---|---|---|---|
| | | | | | | |
| Sorted | 0 | 3 | 4 | 6 | | |



What is Sorting?

Definition (sorting in ascending order):

- Given a set of records r_1, r_2, \dots, r_n with key values k_1, k_2, \dots, k_n , arrange records in order s such that records $r_{s1}, r_{s2}, \dots, r_{sn}$ have keys with property $k_{s1} \leq k_{s2} \leq \dots \leq k_{sn}$.

Example:

| Given | 8 | 0 | 4 | 9 | 3 | 6 |
|--------|---|---|---|---|---|---|
| | | | | | | |
| Sorted | 0 | 3 | 4 | 6 | 8 | |



What is Sorting?

Definition (sorting in ascending order):

- Given a set of records r_1, r_2, \dots, r_n with key values k_1, k_2, \dots, k_n , arrange records in order s such that records $r_{s1}, r_{s2}, \dots, r_{sn}$ have keys with property $k_{s1} \leq k_{s2} \leq \dots \leq k_{sn}$.

Example:

| Given | 8 | 0 | 4 | 9 | 3 | 6 |
|--------|---|---|---|---|---|---|
| | | | | | | |
| Sorted | 0 | 3 | 4 | 6 | 8 | 9 |

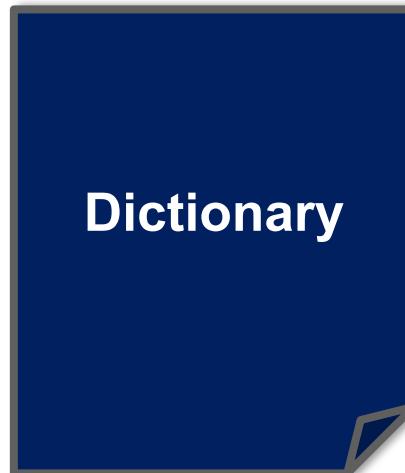




Why do we learn sorting?

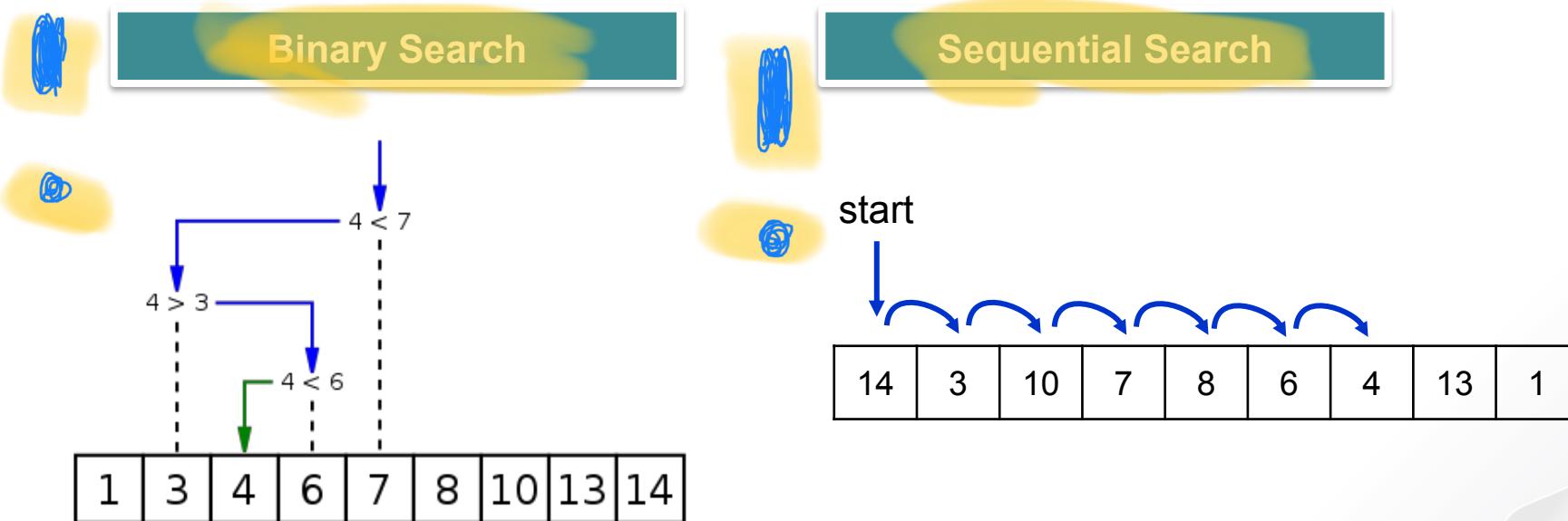
Why do we learn sorting?

- Things must be kept in some order if we want to find them quickly.



Why do we learn sorting?

- Things must be kept in some order if we want to find them quickly.
- How to arrange things in order? Sorting algorithms.
- Sorting is a basic building block for many algorithms.



Reference: T. (2015, April 19). Binary search in a sorted array. Retrieved May 18, 2016, from https://commons.wikimedia.org/wiki/File:Binary_search_into_array.png#/media/File:Binary_search_into_array.png.

Why do we learn sorting?

- Things must be kept in some order if we want to find them quickly.

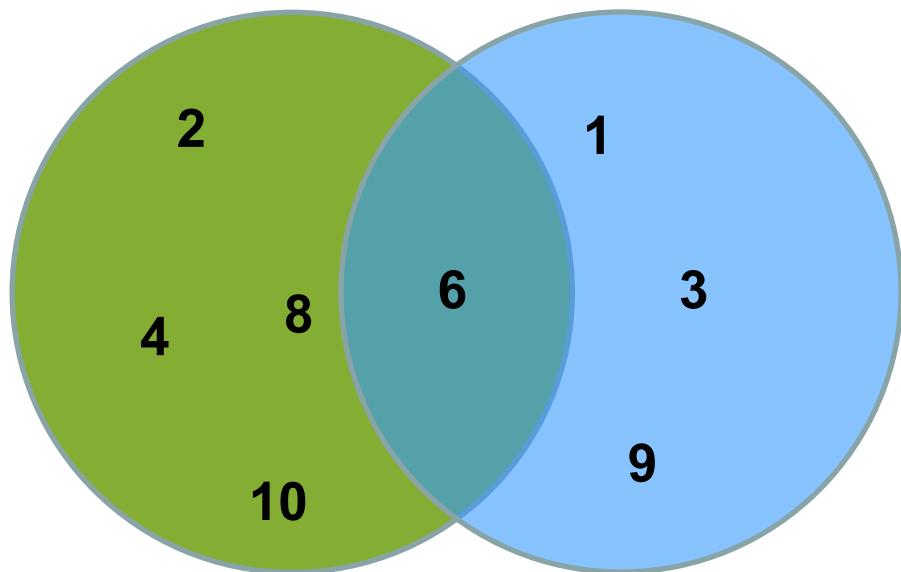
- How to arrange things in order? Sorting algorithms.
- Sorting is a basic building block for many algorithms.

- Most thoroughly studied problem in Computer Science.
- To learn ideas in Algorithm Design derived from techniques in sorting.

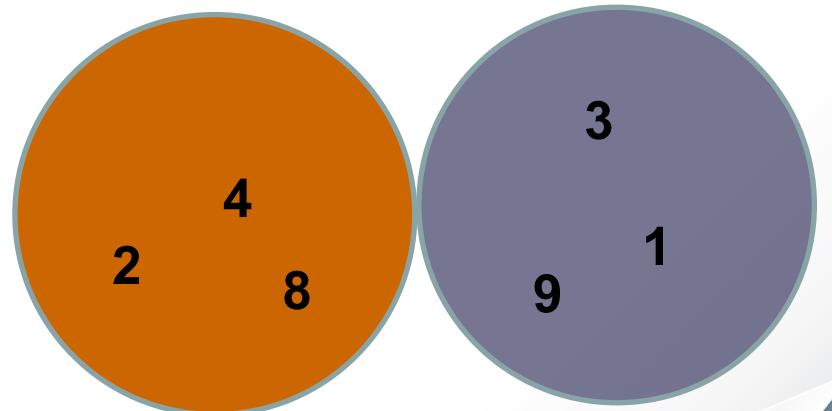
Example: Disjoint Sets

- **Problem:**

- Determine whether two sets (both of size n) are disjoint.



Disjoint

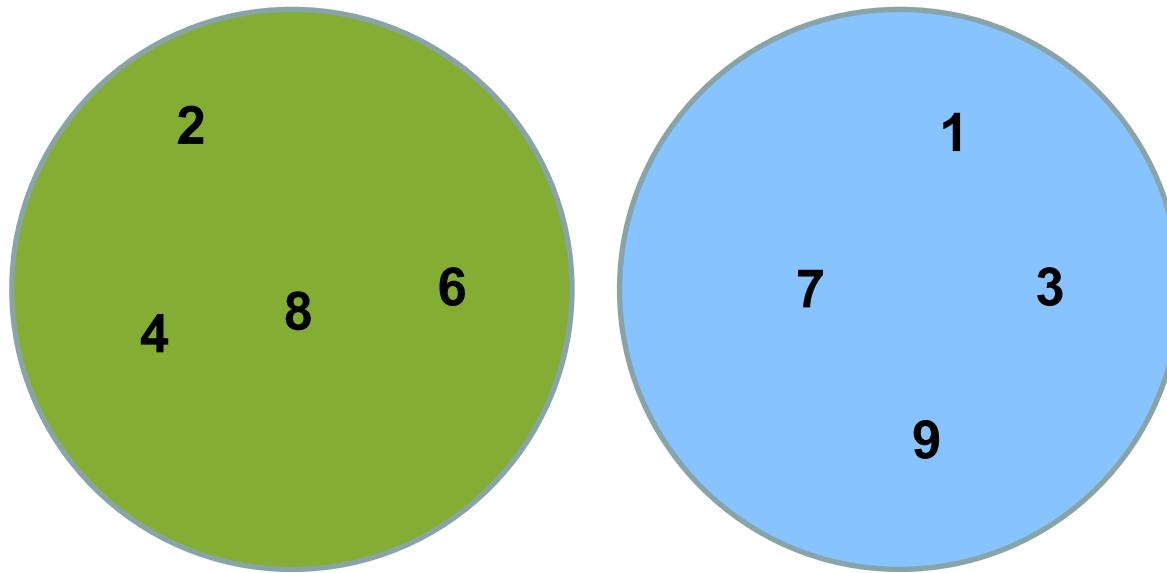


Example: Disjoint Sets

- **Problem:**

Determine whether two sets (both of size n) are disjoint.

- **Solution 1:** Compare each element of the 1st set with each element of the 2nd set. That is, n^2 comparisons.



Example: Disjoint Sets

Problem:

Determine whether two sets (both of size n) are disjoint.

- **Solution 1:** Compare each element of the 1st set with each element of the 2nd set. That is, n^2 comparisons.
- **Solution 2:**

Step 1: We first **sort the first set into ascending order**. This takes $O(nlgn)$ effort using Mergesort or Heapsort.

Step 2: For each element in the 2nd set, **we use Binary Search to find it in the 1st set**. This takes $O(nlgn)$ time.

Comparison of Performance

Solution 1: $O(n^2)$

Solution 2: $O(n \lg n)$

Savings:

| | | | | | |
|-----------|---|-------|--------|--------|---------|
| n | = | 64 | 128 | 256 | 512 |
| n^2 | = | 4,096 | 16,384 | 65,536 | 262,144 |
| $n \lg n$ | = | 384 | 896 | 2,048 | 4,608 |

Comparison of Performance

The data items to be sorted:

- Given a (very large) list of records.
- Each record has the following form: key; rest info of record:

```
class ALIST {  
    KeyType    key;  
    DataType   data;  
};
```

- Key domain is an ordered set.
- **Objective:** To arrange records in ‘ascending’ or ‘descending’ order.

Comparison of Performance

The data items to be sorted:

- Given a (very large) list of records.
- Each record has the following form: key; rest of record:

```
class ALIST {  
    KeyType    key;  
    DataType   data;  
};
```

- Key domain is an ordered set.
- **Objective:** To arrange records in ‘ascending’ or ‘descending’ order.

Comparison of Performance

- Sorting can be classified into **internal sorting** and **external sorting**.

- We focus on internal sorting only,
i.e., all records are in (high speed) main memory during sorting.

↳ happen in my
(small enough to be held in my)

Sorting involves two basic actions:

- 1) key comparisons between two records
- 2) swapping records around

 **Goal:** Use **minimum** working space and do as **few** key comparisons as possible.

Summary

- Sorting is to arrange a set of records so that their key values are in ascending or descending order.
- It is important to learn sorting, because:
 - Sorting has important applications
 - Ideas of sorting can be used for other algorithms
- Objective is to design sorting algorithms with:
 - Minimum usage of memory
 - Minimum number of key comparisons or swaps



SC2001/CE2101/CZ2101: Algorithm Design and Analysis

Insertion Sort

Instructor: Asst. Prof. LIN Shang-Wei

Courtesy of Dr. Ke Yiping, Kelly's slides



Learning Objectives

At the end of this lecture, students should be able to:

- Explain the **incremental approach** as a strategy of algorithm design
- Describe how **insertion sort** algorithm works, by manually running its pseudo code on a toy example
- Analyse the **time complexities** of Insertion sort in the best case, worst case and average case

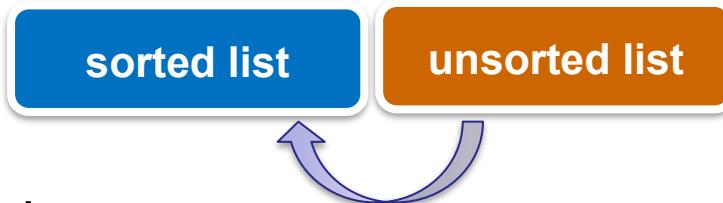
Insertion Sort of a Hand of Cards



Insertion Sort

The incremental approach

- An intuitive, primitive sorting method
- A form of insertion into an **ordered** list
- Given an unordered set of objects, repeatedly remove an entry from the set and insert it into a new **ordered** list
- Ensure that the new list is **ordered at all times**
- Each insertion requires movements of certain entries in the ordered list



Insertion Sort (Pseudo Code)

The incremental approach

```
void InsertionSort (ALIST slot[ ], int n)
{
    // input slot is an array of n records;
    // assume n > 1;
    for (int i=1; i < n; i++)
        for (int j=i; j > 0; j--) {
            if (slot[j].key < slot[j-1].key)
                swap(slot[j], slot[j-1]);
            else break;
        }
}
```

Insertion Sort (Pseudo Code)

The incremental approach

```
void InsertionSort (ALIST slot[ ], int n)
```

{ // input slot is an array of n records;

6 29 45 12 64 16
 ⑨ ⑧ ⑦

6 12 29 45 64 16
 ⑩ ⑪ ⑫ ⑬

6 12 29 45 16 64
 ⑭ ⑮ ⑯

6 12 29 16 45 64
 ⑰ ⑱ ⑲

6 12 } 16 29 45 64
 ⑳ ⑳ ⑳



45 29 06 64 12 16

12 29 45 64 16

12 29 45 64 16

12 29 45 64 16

12 29 45 64 16

```
class ALIST {  

    KeyType key;  

    DataType data;  

};
```

Insertion Sort (Pseudo Code)

The incremental approach

```
void InsertionSort (ALIST slot[ ], int n)
```

```
{ // input slot is an array of n records;
```

```
    // assume n > 1;
```

```
    for (int i=1; i < n; i++)
```

```
        for (int j=i; j > 0; j--) {
```

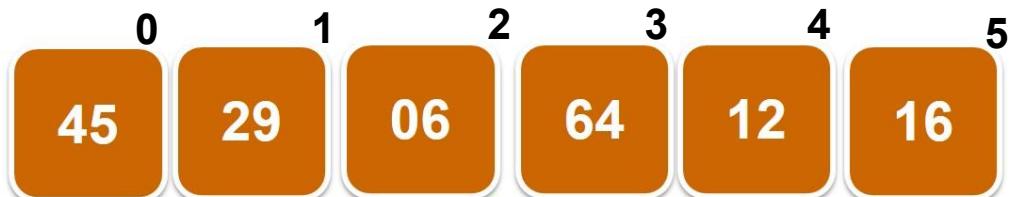
```
            if (slot[j].key < slot[j-1].key)
```

```
                swap(slot[j], slot[j-1]);
```

```
            else break;
```

```
}
```

```
}
```



Insertion Sort (Pseudo Code)

The incremental approach

```
void InsertionSort (ALIST slot[ ], int n)
```

```
{ // input slot is an array of n records;
```

```
    // assume n > 1;
```

```
    for (int i=1; i < n; i++)
```

```
        for (int j=i; j > 0; j--) {
```

```
            if (slot[j].key < slot[j-1].key)
```

```
                swap(slot[j], slot[j-1]);
```

```
            else break;
```

```
}
```

```
}
```



Insertion Sort (Pseudo Code)

The incremental approach

```
void InsertionSort (ALIST slot[ ], int n)
{ // input slot is an array of n records;
  // assume n > 1;
  for (int i=1; i < n; i++)
    for (int j=i; j > 0; j--) {
      if (slot[j].key < slot[j-1].key)
        swap(slot[j], slot[j-1]);
      else break;
    }
}
```

Insertion Sort (Pseudo Code)

The incremental approach

```
void InsertionSort (ALIST slot[ ], int n)
{ // input slot is an array of n records;
  // assume n > 1;
  for (int i=1; i < n; i++)  Pick up a new item from slot[ ]
    for (int j=i; j > 0; j--) {
      if (slot[j].key < slot[j-1].key)
        swap(slot[j])
      else break;
    }
}
```

sorted list

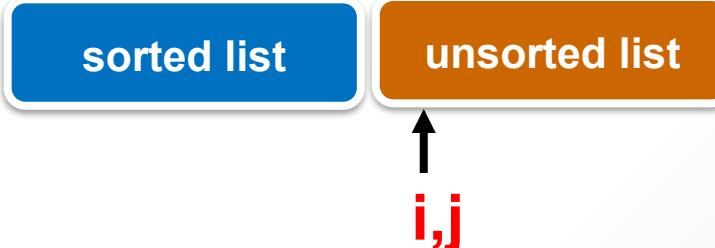
unsorted list



Insertion Sort (Pseudo Code)

The incremental approach

```
void InsertionSort (ALIST slot[ ], int n)
{ // input slot is an array of n records;
  // assume n > 1;
  for (int i=1; i < n; i++)
    for (int j=i; j > 0; j--) { Find the correct position to insert
      if (slot[j].key < slot[j-1].key) the item.
      swap(slot[j], slot[j-1]);
      else break;
    }
}
```



Insertion Sort (Pseudo Code)

The incremental approach

```
void InsertionSort (ALIST slot[ ], int n)
```

```
{ // input slot is an array of n records;
```

```
    // assume n > 1;
```

```
    for (int i=1; i < n; i++)
```

```
        for (int j=i; j > 0; j--) {
```

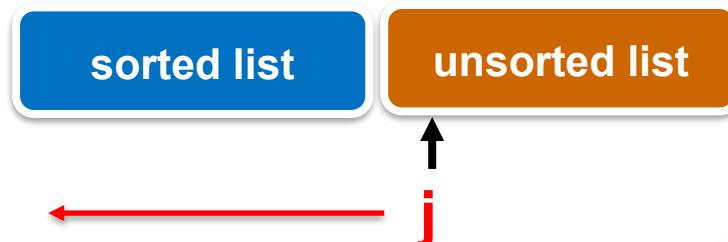
```
            if (slot[j].key < slot[j-1].key)
```

```
                swap(slot[j], slot[j-1]);
```

```
            else break;
```

```
}
```

```
}
```



Insertion Sort (Pseudo Code)

The incremental approach

```
void InsertionSort (ALIST slot[ ], int n)
```

```
{ // input slot is an array of n records;
```

```
    // assume n > 1;
```

```
    for (int i=1; i < n; i++)
```

```
        for (int j=i; j > 0; j--) {
```

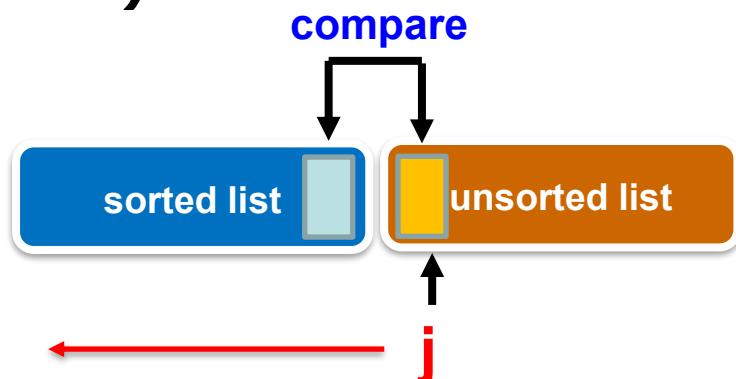
```
            if (slot[j].key < slot[j-1].key)
```

```
                swap(slot[j], slot[j-1]);
```

```
            else break;
```

```
}
```

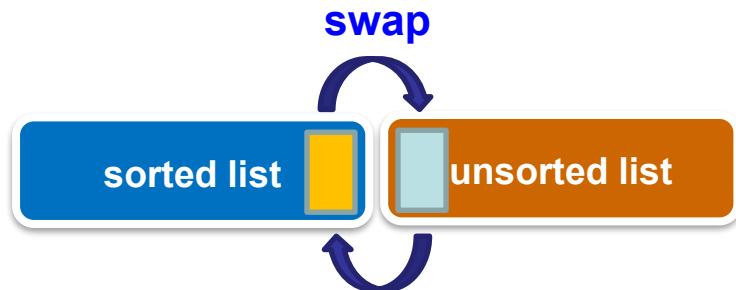
```
}
```



Insertion Sort (Pseudo Code)

The incremental approach

```
void InsertionSort (ALIST slot[ ], int n)
{ // input slot is an array of n records;
  // assume n > 1;
  for (int i=1; i < n; i++)
    for (int j=i; j > 0; j--) {
      if (slot[j].key < slot[j-1].key)
        swap(slot[j], slot[j-1]);
      else break;
    }
}
```



Insertion Sort (Pseudo Code)

The incremental approach

```
void InsertionSort (ALIST slot[ ], int n)
```

```
{ // input slot is an array of n records;
```

```
    // assume n > 1;
```

```
    for (int i=1; i < n; i++)
```

```
        for (int j=i; j > 0; j--) {
```

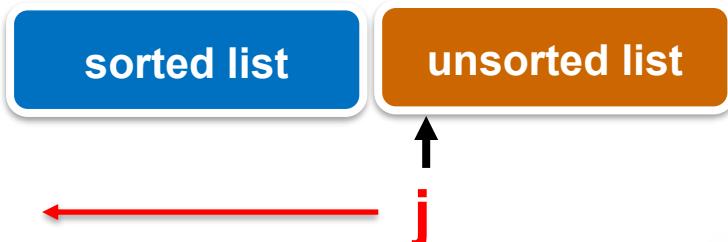
```
            if (slot[j].key < slot[j-1].key)
```

```
                swap(slot[j], slot[j-1]);
```

```
            else break;
```

```
}
```

```
}
```



Insertion Sort (Pseudo Code)

The incremental approach

```
void InsertionSort (ALIST slot[ ], int n)
{ // input slot is an array of n records;
  // assume n > 1;
  for (int i=1; i < n; i++)
    for (int j=i; j > 0; j--) {
      if (slot[j].key < slot[j-1].key)
        swap(slot[j], slot[j-1]);
      else break; ← What does it mean?
    }
}
```

The correct position was found!!

Insertion Sort (Pseudo Code)

The incremental approach

```
void InsertionSort (ALIST slot[ ], int n)
```

```
{ // input slot is an array of n records;
```

```
    // assume n > 1;
```

```
    for (int i=1; i < n; i++)
```

```
        for (int j=i; j > 0; j--) {
```

```
            if (slot[j].key < slot[j-1].key)
```

```
                swap(slot[j], slot[j-1]);
```

```
            else break;
```

```
}
```

```
}
```





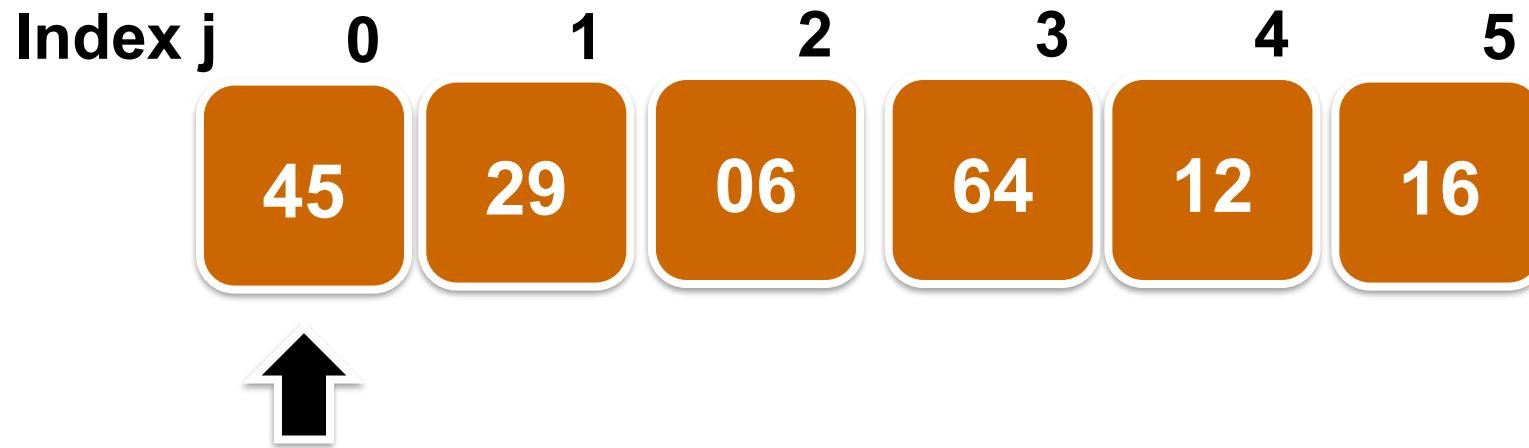
Insertion Sort Example

Insertion Sort Example

Sort in ascending order

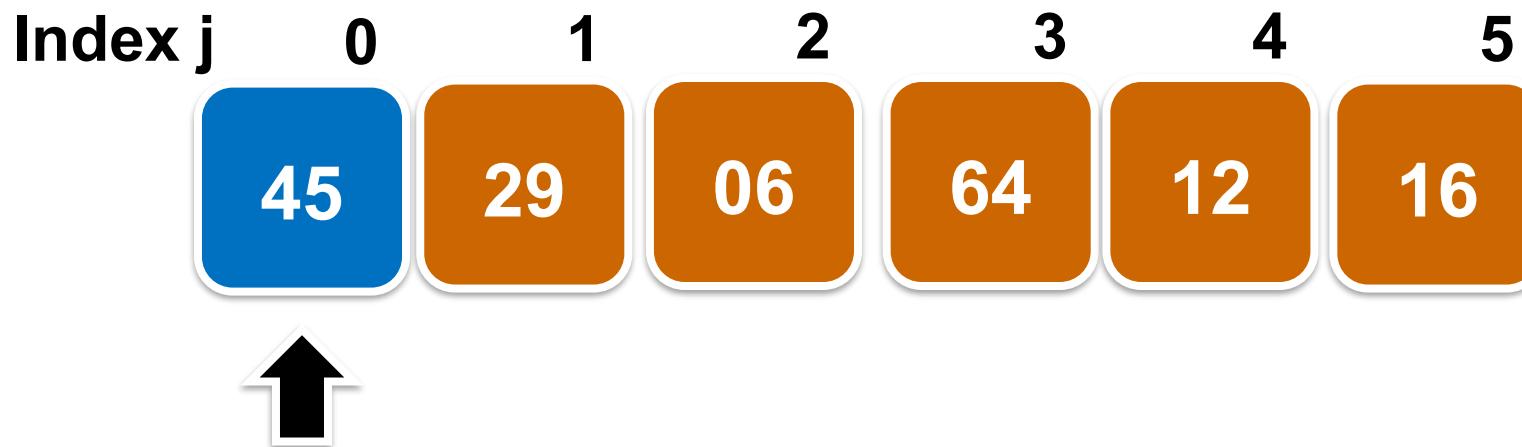
| Index j | 0 | 1 | 2 | 3 | 4 | 5 |
|---------|----|----|----|----|----|----|
| | 45 | 29 | 06 | 64 | 12 | 16 |

Insertion Sort Example

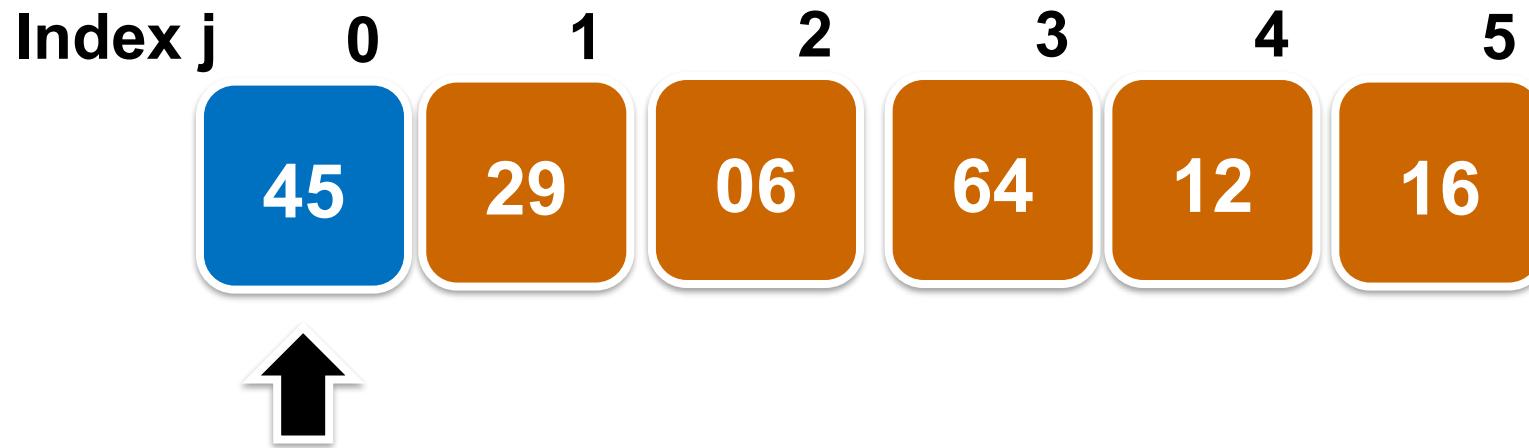


Insertion Sort Example

$i = 0$



Insertion Sort Example



Insertion Sort Example

If (slot[j].key < slot[j-1].key)

(slot[**1**].key < slot[**0**].key)

29 < **45** ✓

1
2
3
4
5

Index j 0 1 2 3 4 5

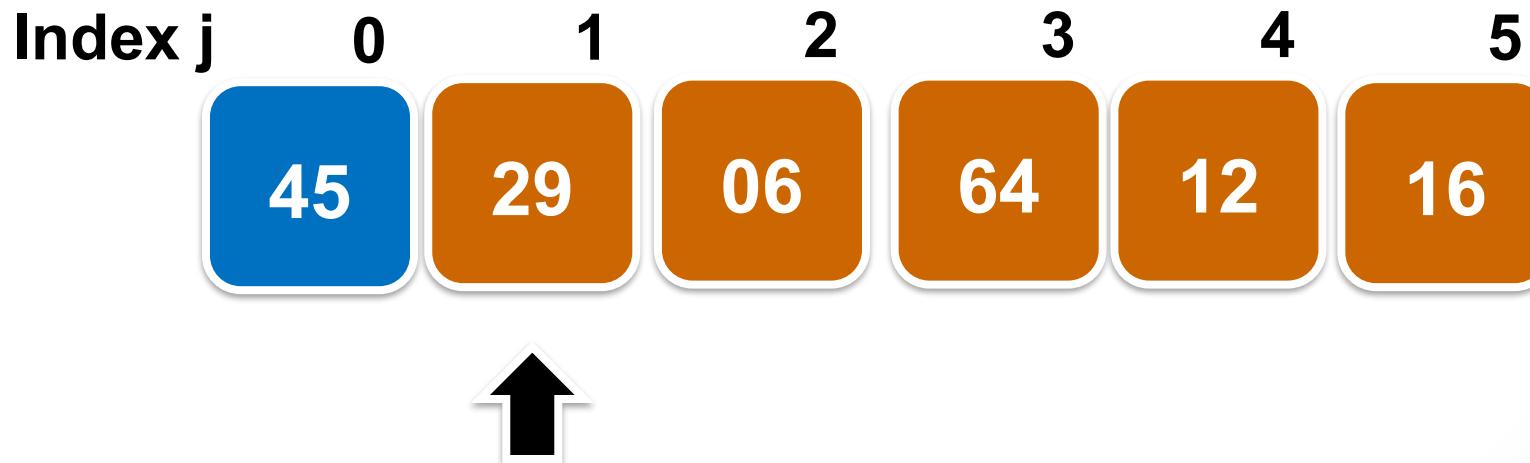
| | | | | | |
|----|----|----|----|----|----|
| 45 | 29 | 06 | 64 | 12 | 16 |
|----|----|----|----|----|----|



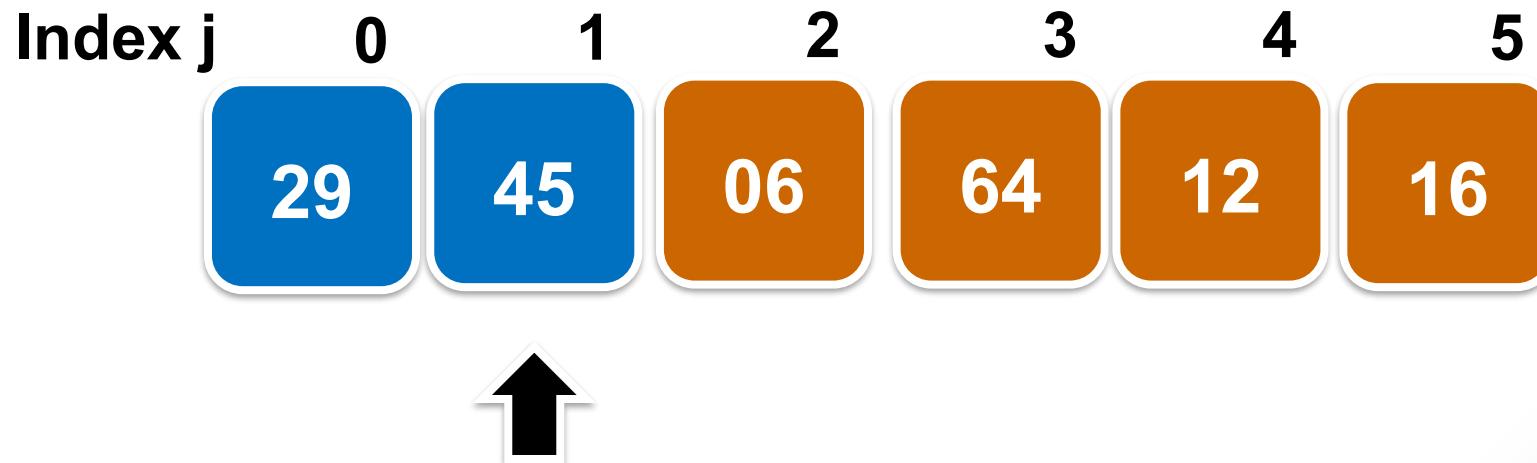
Insertion Sort Example

swap(slot[j], slot[j-1]);
swap(slot[1], slot[0]);

j > 1
j > 1 *break*



Insertion Sort Example



Insertion Sort Example

If (slot[j].key < slot[j-1].key)

(slot[**2**].key < slot[**1**].key)

06 < **45** ✓

i
j
j-1

| Index j | 0 | 1 | 2 | 3 | 4 | 5 |
|---------|---|---|---|---|---|---|
|---------|---|---|---|---|---|---|



Insertion Sort Example

swap(slot[j], slot[j-1]);
swap(slot[2], slot[1]);

j, 2
j, 2

Index j 0 1 2 3 4 5



Insertion Sort Example

↓ ↓
↓ ↓
↓ ↓

If (slot[j].key < slot[j-1].key)

(slot[1].key < slot[0].key)

06 < 29 ✓

| Index j | 0 | 1 | 2 | 3 | 4 | 5 |
|---------|---|---|---|---|---|---|
|---------|---|---|---|---|---|---|



Insertion Sort Example

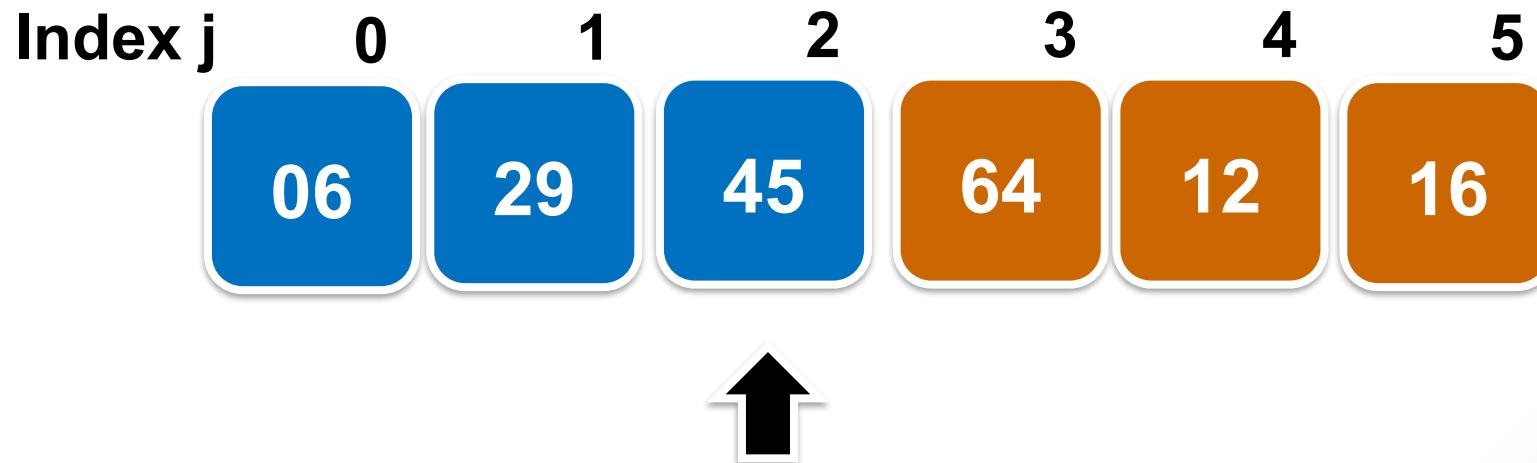
-
i = 2
-
j = 1
-
j = 0
up slot 0x

swap(slot[j], slot[j-1]);
swap(slot[1], slot[0]);

Index j 0 1 2 3 4 5



Insertion Sort Example



Insertion Sort Example

i = 3
j = 3
Up to slot

If (slot[j].key < slot[j-1].key)

(slot[3].key < slot[2].key)

64 < 45 ✗

Index j 0 1 2 3 4 5

| | | | | | |
|----|----|----|----|----|----|
| 06 | 29 | 45 | 64 | 12 | 16 |
|----|----|----|----|----|----|



Insertion Sort Example

$i = 4$

$j = 4$

If ($\text{slot}[j].key < \text{slot}[j-1].key$)

($\text{slot}[4].key < \text{slot}[3].key$)

12 < 64 ✓

| Index j | 0 | 1 | 2 | 3 | 4 | 5 |
|---------|---|---|---|---|---|---|
|---------|---|---|---|---|---|---|

06

29

45

64

12

16



Insertion Sort Example

j = 4
j = 4

swap(slot[j], slot[j-1]);
swap(slot[4], slot[3]);

Index j 0 1 2 3 4 5



Insertion Sort Example

i ↗
j ↘

If (slot[j].key < slot[j-1].key)

(slot[3].key < slot[2].key)

12 < 45 ✓

| Index j | 0 | 1 | 2 | 3 | 4 | 5 |
|---------|---|---|---|---|---|---|
|---------|---|---|---|---|---|---|



Insertion Sort Example

$i \geq 4$
 $-$
 $\swarrow \searrow$

swap(slot[j], slot[j-1]);
swap(slot[3], slot[2]);

Index j 0 1 2 3 4 5



Insertion Sort Example

*j: 4
j: 2*

If (slot[j].key < slot[j-1].key)

(slot[**2**].key < slot[**1**].key)

12 < **29** ✓

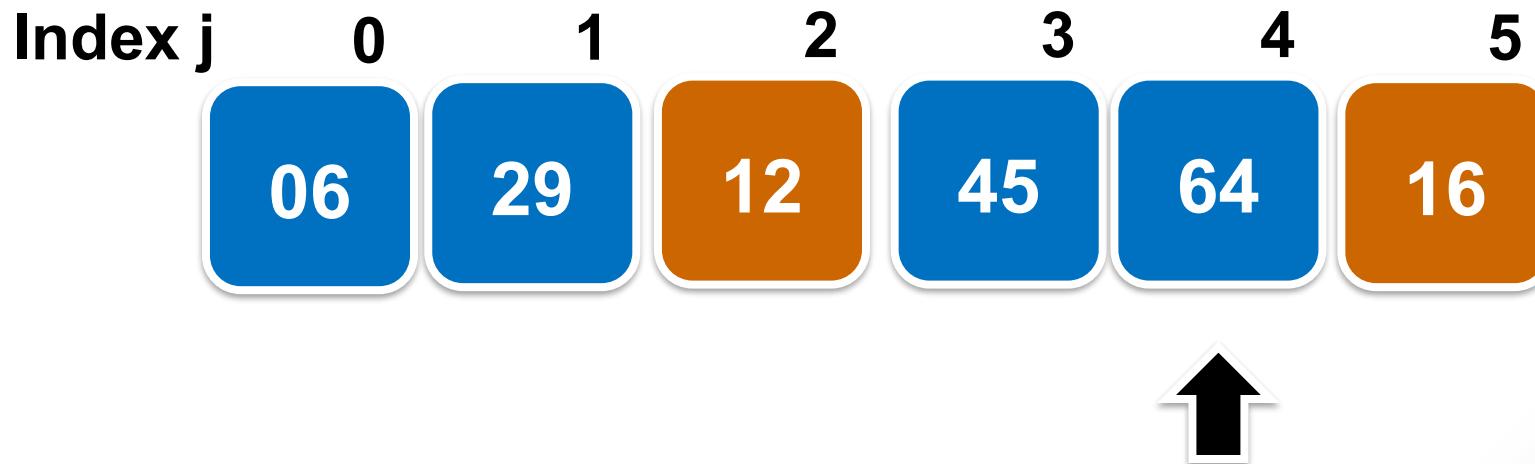
| Index j | 0 | 1 | 2 | 3 | 4 | 5 |
|---------|---|---|---|---|---|---|
|---------|---|---|---|---|---|---|



Insertion Sort Example

swap(slot[j], slot[j-1]);
swap(slot[2], slot[1]);

j = 2



Insertion Sort Example

$i = 4$

$j = 1$

Up to slot

If (slot[j].key < slot[j-1].key)

(slot[1].key < slot[0].key)

12 < 06 ✗

| Index j | 0 | 1 | 2 | 3 | 4 | 5 |
|---------|----|----|----|----|----|----|
| | 06 | 12 | 29 | 45 | 64 | 16 |

06

12

29

45

64

16



Insertion Sort Example

j : 5
-
j : 4

If (slot[j].key < slot[j-1].key)

(slot[5].key < slot[4].key)

16 < 64 ✓

| Index j | 0 | 1 | 2 | 3 | 4 | 5 |
|---------|---|---|---|---|---|---|
|---------|---|---|---|---|---|---|



Insertion Sort Example



swap(slot[j], slot[j-1]);
swap(slot[5], slot[4]);

Index j 0 1 2 3 4 5



Insertion Sort Example

*i ↗
j ↘
j+1 ↙*

If (slot[j].key < slot[j-1].key)

(slot[4].key < slot[3].key)

16 < 45 ✓

| Index j | 0 | 1 | 2 | 3 | 4 | 5 |
|---------|----|----|----|----|----|----|
| | 06 | 12 | 29 | 45 | 16 | 64 |



Insertion Sort Example

-
 $i = 5$
-
 $j = 4$

swap(slot[j], slot[j-1]);
swap(slot[4], slot[3]);

Index j 0 1 2 3 4 5

06

12

29

45

16

64



Insertion Sort Example



If (slot[j].key < slot[j-1].key)

(slot[**3**].key < slot[**2**].key)

16 < **29** ✓

| Index j | 0 | 1 | 2 | 3 | 4 | 5 |
|---------|----|----|----|----|----|----|
| | 06 | 12 | 29 | 16 | 45 | 64 |



Insertion Sort Example

-
j ↗ {
... ↘ }
j ↘

swap(slot[j], slot[j-1]);
swap(slot[3], slot[2]);

| Index j | 0 | 1 | 2 | 3 | 4 | 5 |
|---------|----|----|----|----|----|----|
| | 06 | 12 | 29 | 16 | 45 | 64 |



Insertion Sort Example

If (slot[j].key < slot[j-1].key)
(slot[2].key < slot[1].key)
16 < 12 ✗

| Index j | 0 | 1 | 2 | 3 | 4 | 5 |
|---------|----|----|----|----|----|----|
| | 06 | 12 | 16 | 29 | 45 | 64 |



Insertion Sort Example

Sorted in ascending order

| Index j | 0 | 1 | 2 | 3 | 4 | 5 |
|---------|----|----|----|----|----|----|
| | 06 | 12 | 16 | 29 | 45 | 64 |



Insertion Sort Algorithm (Recap)

Insertion Sort Algorithm

- Original unsorted set and final sorted list are both in array slot[].

06

12

29

45

64

16

Insertion Sort Algorithm

- Original unsorted set and final sorted list are both in array slot[].
- Since sorting is performed directly on original array without any working storage, swapping and shifting are essential.

06

12

29

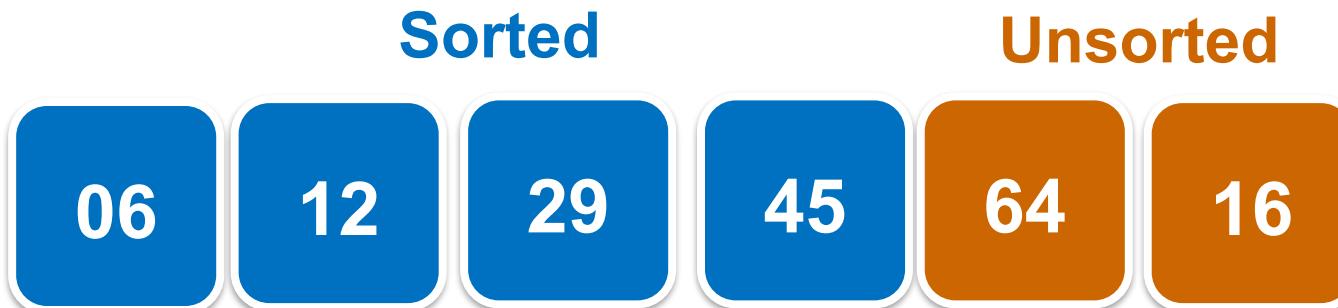
45

64

16

Insertion Sort Algorithm

- Original unsorted set and final sorted list are both in array slot[].
- Since sorting is performed directly on original array without any working storage, swapping and shifting are essential.
- During sorting, slot[] contains sorted portion on the ‘left’ and unsorted portion on the ‘right’; sorted portion grows while unsorted portion shrinks.



Insertion Sort Algorithm

- In the outer ‘for’ loop, i begins with 1 because the ordered list begins with one element (slot[0]); hence slot[1] is the first element from the unordered list.

```
for (int i=1; i < n; i++)  
    for (int j=i; j > 0; j--) {  
        if (slot[j].key < slot[j-1].key)  
            swap(slot[j], slot[j-1]);  
        else break;  
    }
```

Insertion Sort Algorithm

- At each iteration, number at slot[i] is inserted into the new ordered list.

```
for (int i=1; i < n; i++)  
    for (int j=i; j > 0; j--) {  
        if (slot[j].key < slot[j-1].key)  
            swap(slot[j], slot[j-1]);  
        else break;
```

Insertion Sort Algorithm

- The inner ‘for’ loop finds the correct position in the ordered list by swapping slot[j] with slot[j-1] as long as the key of slot[j-1] is > the key of slot[j].

```
for (int i=1; i < n; i++)  
    for (int j=i; j > 0; j--) {  
        if (slot[j].key < slot[j-1].key)  
            swap(slot[j], slot[j-1]);  
        else break;
```

Insertion Sort Algorithm

- The inner ‘for’ loop finds the correct position in the ordered list by swapping slot[j] with slot[j-1] as long as the key of slot[j-1] is > the key of slot[j].

```
for (int i=1; i < n; i++)  
    for (int j=i; j > 0; j--) {  
        if (slot[j].key < slot[j-1].key)  
            swap(slot[j], slot[j-1]);  
        else break;
```



Complexity of Insertion Sort

Complexity of Insertion Sort

Number of key comparisons:

- There are $n - 1$ iterations (**the outer loop**)
- Best case: 1 key comparison/ iteration, total: $n - 1$
- Already sorted: [06] [12] [16] [29] [45] [64]
- Worst case: i key comparisons for the i th iteration
- Reversely sorted: [64] [45] [29] [16] [12] [06]

Total: $1 + 2 + 3 + \dots + (n-1) = \sum_{i=1}^{n-1} i = \frac{(n-1)n}{2} = \frac{n^2 - n}{2}$

Insertion Sort Performance

- **Average case:** the i th iteration may have $1, 2, \dots, i$ key comparisons, each with $1/i$ chance.

The average no. of comparisons in the i th iteration:

$$\frac{1}{i} \sum_{j=1}^i j = \frac{1}{i} (1 + 2 + \dots + i)$$

Summation for the $n-1$ iterations:

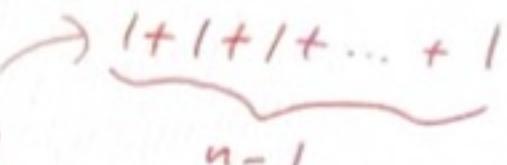
$$\begin{aligned} 1 + \frac{1}{2}(1+2) + \frac{1}{3}(1+2+3) + \dots + \frac{1}{n-1}(1+\dots+n-1) &= \sum_{i=1}^{n-1} \left(\frac{1}{i} \sum_{j=1}^i j \right) \\ &= \sum_{i=1}^{n-1} \left(\frac{1}{i} \frac{i(i+1)}{2} \right) = \frac{1}{2} \sum_{i=1}^{n-1} (i+1) = \frac{1}{2} \left(\frac{(n-1)(n+2)}{2} \right) = \Theta(n^2) \end{aligned}$$

Hint for analysing the average case of insertion sort

Posted on: Friday, August 19, 2022 10:10:54 PM SGT

Dear students,

Some of you are asking how to do the last step on p. 57 of the slides about insertion sort.
The hint is given in the following:

$$\begin{aligned}\sum_{i=1}^{n-1} (i+1) &= \sum_{i=1}^{n-1} i + \sum_{i=1}^{n-1} 1 \\&= \frac{(n-1) \cdot n}{2} + (n-1) \\&= \frac{n^2 - n + 2n - 2}{2} = \frac{n^2 + n - 2}{2} \\&= \frac{(n-1)(n+2)}{2}\end{aligned}$$


Best Regards,
Lin, S.-W.



Insertion Sort Performance

😊 Strengths:

- 👉 Good when the unordered list is almost sorted.
- 👉 Need minimum time to verify if the list is sorted.
- 👉 Fast with linked storage implementation: no movement of data.

😢 Weaknesses:

- 👉 When an entry is inserted, it may still not be in the final position yet.
- 👉 Every new insertion necessitates movements for some inserted entries in ordered list.
- 👉 When each slot is large (e.g., a slot contains a large record of 10Mb), movement is expensive.
- 👉 Less suitable with contiguous storage implementation.

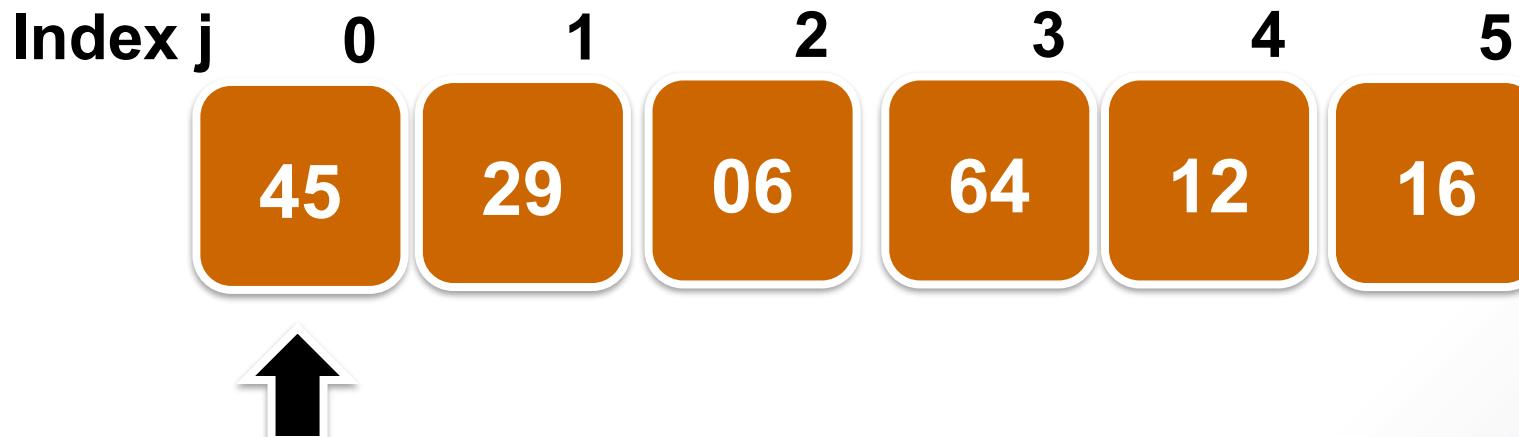
Summary

- Insertion sort uses the incremental approach.
- **Main idea:** Repeatedly pick up an element x to insert into a sorted sub-array on the left side, by comparing x with its left neighbour. If they are out of order, swap them; otherwise, insert x there.

| Index j | 0 | 1 | 2 | 3 | 4 | 5 |
|---------|----|----|----|----|----|----|
| | 45 | 29 | 06 | 64 | 12 | 16 |

Summary

- Insertion sort uses the incremental approach.
- **Main idea:** Repeatedly pick up an element x to insert into a sorted sub-array on the left side, by comparing x with its left neighbour. If they are out of order, swap them; otherwise, insert x there.



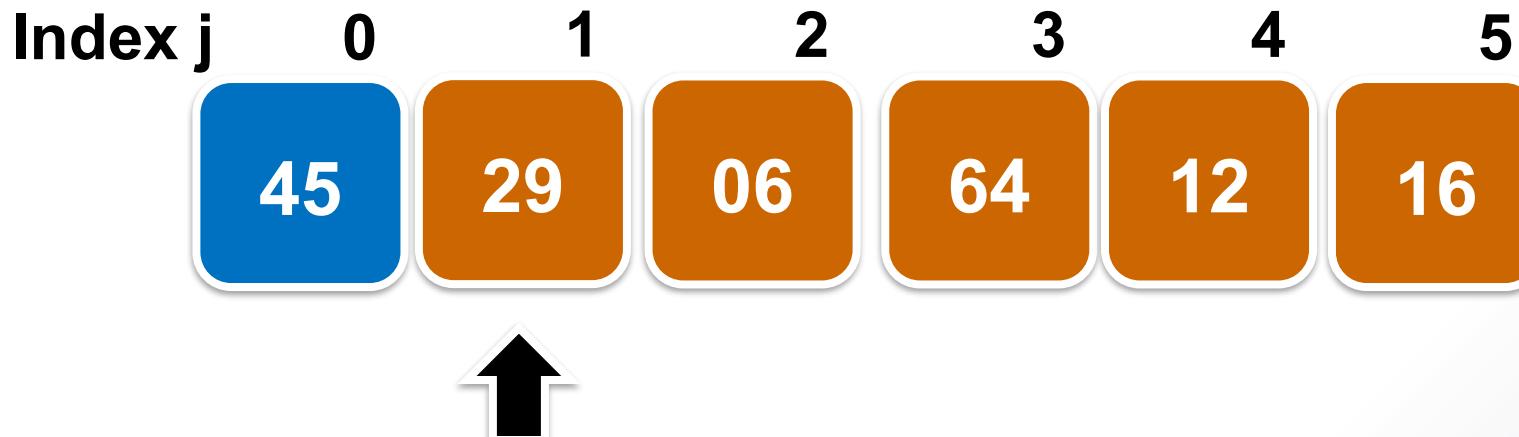
Summary

- Insertion sort uses the incremental approach.
- **Main idea:** Repeatedly pick up an element x to insert into a sorted sub-array on the left side, by comparing x with its left neighbour. If they are out of order, swap them; otherwise, insert x there.

| Index j | 0 | 1 | 2 | 3 | 4 | 5 |
|---------|----|----|----|----|----|----|
| | 45 | 29 | 06 | 64 | 12 | 16 |

Summary

- Insertion sort uses the incremental approach.
- **Main idea:** Repeatedly pick up an element x to insert into a sorted sub-array on the left side, by comparing x with its left neighbour. If they are out of order, swap them; otherwise, insert x there.



Summary

- Insertion sort uses the incremental approach.
- **Main idea:** Repeatedly pick up an element x to insert into a sorted sub-array on the left side, by comparing x with its left neighbour. If they are out of order, swap them; otherwise, insert x there.
- **Time complexity analysis:**
 - Best case: $\Theta(n)$, when input array is already sorted.
 - Worst case: $\Theta(n^2)$, when input array is reversely sorted.
 - Average case: $\Theta(n^2)$.



SC2001/CE2101/CZ2101: Algorithm Design and Analysis

Mergesort

Instructor: Asst. Prof. LIN Shang-Wei

Courtesy of Dr. Ke Yiping, Kelly's slides

Learning Objectives

At the end of this lecture, students should be able to:

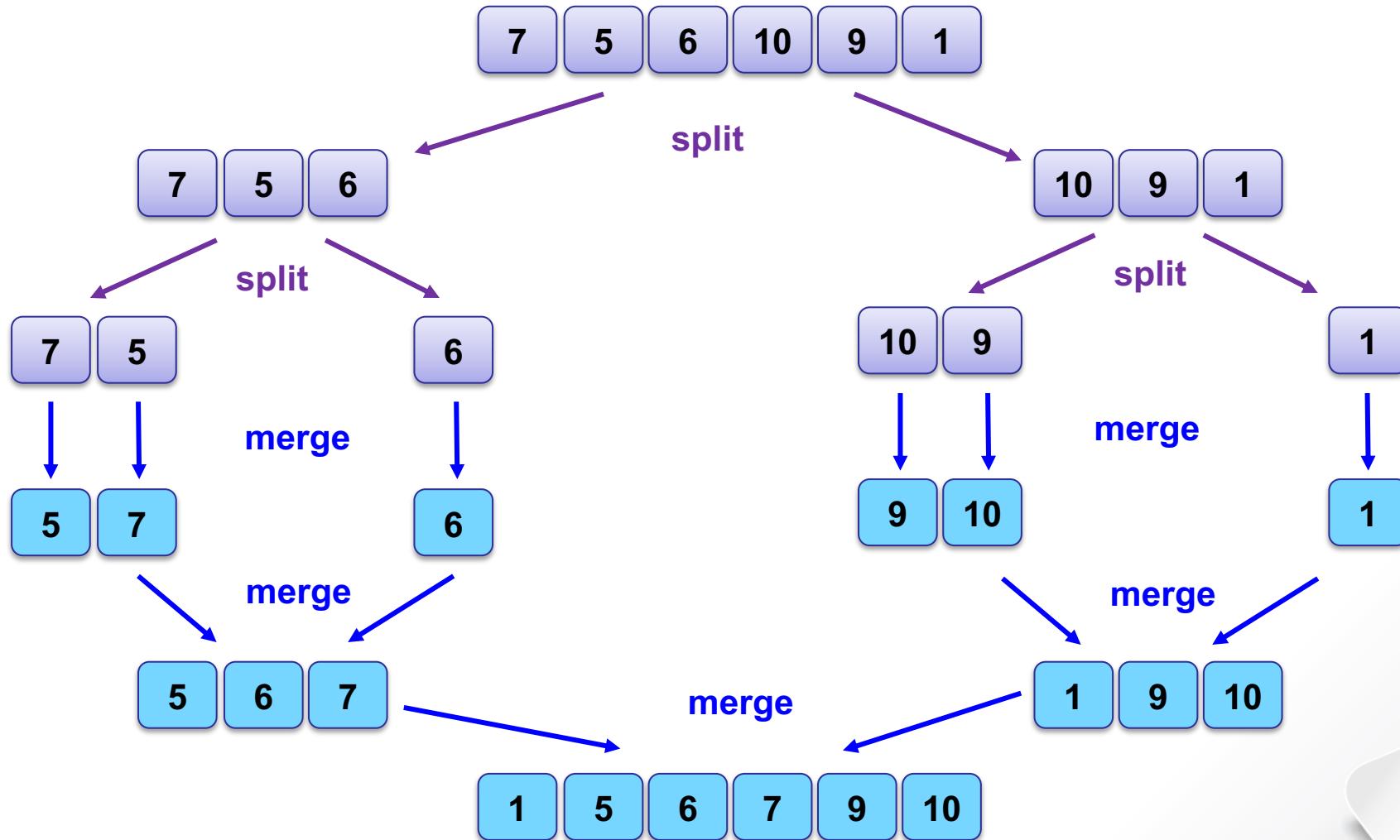
- Explain the approach of **Divide and Conquer**
- Describe how Mergesort works by:
 - Recalling the pseudo code
 - Manually executing the algorithm on a toy input array
- Analyse the **time complexity** of Mergesort, by using:
 - Recurrence equation
 - Recursion tree



Mergesort

(Divide and Conquer Approach)

Mergesort in a nutshell



Mergesort

The Divide and Conquer approach

The skeleton of this approach:

```
solve (problem of size n)
{
    if (n <= minimum size)
        solve the problem directly;
    else {
        divide the problem into p1, p2, ..., pk;
        for each sub-problem ps
            solutions = solve (ps);
        combine all solutions;
    }
}
```

Mergesort

The Divide and Conquer approach

The skeleton of this approach:

```
solve (problem of size n)
{
    if (n <= minimum size)
        solve the problem directly;
    else {
        divide the problem into p1, p2, ..., pk;
        for each sub-problem ps
            solutions = solve (ps);
        combine all solutions;
    }
}
```

Mergesort

The Divide and Conquer approach

The skeleton of this approach:

```
solve (problem of size n)
{
    if (n <= minimum size)
        solve the problem directly;
    else {
        divide the problem into p1, p2, ..., pk;
        for each sub-problem ps
            solutions = solve (ps);
        combine all solutions;
    }
}
```

Mergesort

The Divide and Conquer approach

The skeleton of this approach:

```
solve (problem of size n)
{
    if (n <= minimum size)
        solve the problem directly;
    else {
        divide the problem into p1, p2, ..., pk;
        for each sub-problem ps
            solutions = solve (ps);
        combine all solutions;
    }
}
```

Mergesort

The Divide and Conquer approach

The skeleton of this approach:

```
solve (problem of size n)
{
    if (n <= minimum size)
        solve the problem directly;
    else {
        divide the problem into p1, p2, ..., pk;
        for each sub-problem ps
            solutions = solve (ps);
        combine all solutions;
    }
}
```

Mergesort

The Divide and Conquer approach

The skeleton of this approach:

solve (problem of size n)

{ if ($n \leq$ minimum size)

 solve the problem directly;

 else {

 divide the problem into p_1, p_2, \dots, p_k ;

for each sub-problem p_s

solution_s = solve (p_s);

 combine all solution_s;

 }

}

Mergesort

The Divide and Conquer approach

The skeleton of this approach:

```
solve (problem of size n)
{
    if (n <= minimum size)
        solve the problem directly;
    else {
        divide the problem into p1, p2, ..., pk;
        for each sub-problem ps
            solutions = solve (ps);
        combine all solutions;
    }
}
```

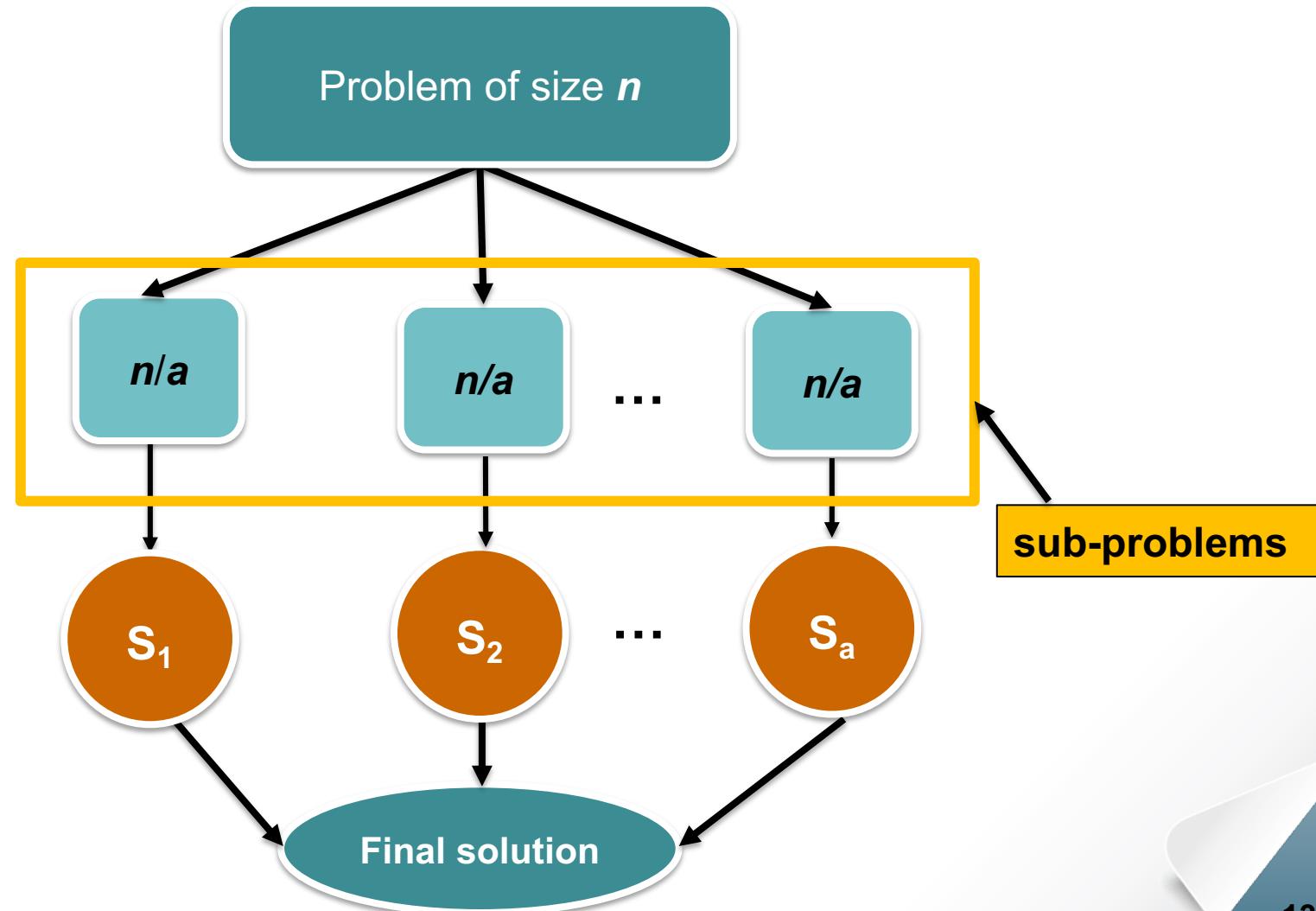
Mergesort

The Divide and Conquer approach

The skeleton of this approach:

```
solve (problem of size n)
{
    if (n <= minimum size)
        solve the problem directly;
    else {
        divide the problem into p1, p2, ..., pk;
        for each sub-problem ps
            solutions = solve (ps);
        combine all solutions;
    }
}
```

Mergesort





Mergesort (Algorithm)

Mergesort (Algorithm)

```
mergeSort(list) {  
    if (length of list > 1) {  
        Partition list into two (approx.) equal sized  
        lists, L1 & L2;  
        mergeSort (L1);  
        mergeSort (L2);  
        merge the sorted L1 & L2;  
    }  
}
```

Mergesort (Algorithm)

```
mergeSort(list) {
```

```
    if (length of list > 1) {
```

Partition list into two (approx.) equal sized

lists, L1 & L2;

```
        mergeSort (L1);
```

```
        mergeSort (L2);
```

merge the sorted L1 & L2;

```
}
```

```
}
```

Mergesort (Algorithm)

```
mergeSort(list) {
```

```
    if (length of list > 1) {
```

Partition list into two (approx.) equal sized

lists, L1 & L2;

```
        mergeSort (L1);
```

```
        mergeSort (L2);
```

merge the sorted L1 & L2;

```
}
```

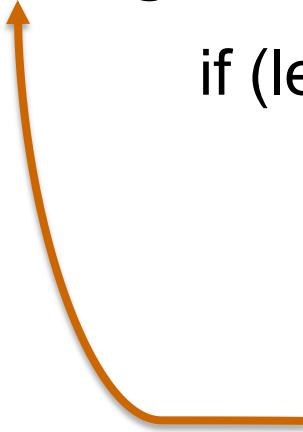
```
}
```

Mergesort (Algorithm)

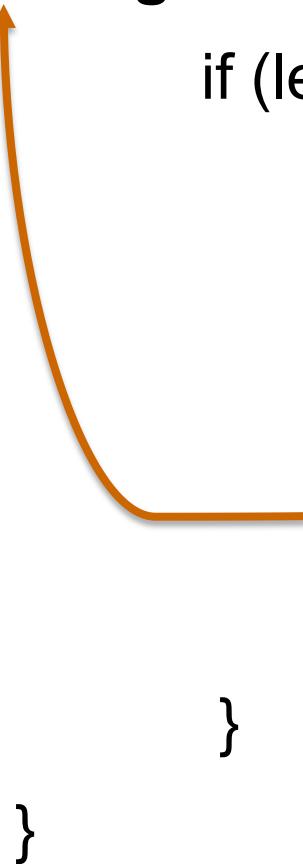
```
mergeSort(list) {  
    if (length of list > 1) {  
        Partition list into two (approx.) equal sized  
        lists, L1 & L2;  
        mergeSort (L1);  
        mergeSort (L2);  
        merge the sorted L1 & L2;  
    }  
}
```

Mergesort (Algorithm)

```
mergeSort(list) {  
    if (length of list > 1) {  
        Partition list into two (approx.) equal sized  
        lists, L1 & L2;  
        mergeSort (L1);  
        mergeSort (L2);  
        merge the sorted L1 & L2;  
    }  
}
```



Mergesort (Algorithm)



```
mergeSort(list) {  
    if (length of list > 1) {  
        Partition list into two (approx.) equal sized  
        lists, L1 & L2;  
        mergeSort (L1);  
        mergeSort (L2);  
        merge the sorted L1 & L2;  
    }  
}
```

Mergesort (Algorithm)

```
mergeSort(list) {  
    if (length of list > 1) {  
        Partition list into two (approx.) equal sized  
        lists, L1 & L2;  
        mergeSort (L1);  
        mergeSort (L2);  
        merge the sorted L1 & L2;  
    }  
}
```

Mergesort (Algorithm)

```
mergeSort(list) {  
    if (length of list > 1) {  
        Partition list into two (approx.) equal sized  
        lists, L1 & L2;  
        mergeSort (L1);  
        mergeSort (L2);  
        merge the sorted L1 & L2;  
    }  
}
```



Mergesort (Overview of Pseudo Code)

Mergesort

```
void mergesort(int n, int m)
```

```
{ int mid = (n+m)/2;
```

```
if (m-n <= 0)
```

```
    return;
```

```
else if (m-n > 1) {
```

```
    mergesort(n, mid);
```

```
    mergesort(mid+1, m);
```

```
}
```

```
merge(n, m);
```

```
}
```

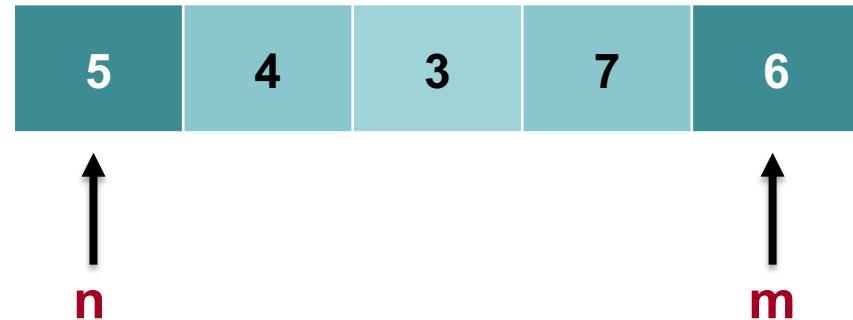
| | | | | |
|---|---|---|---|---|
| 5 | 4 | 3 | 7 | 6 |
|---|---|---|---|---|

4 → left + 2 elements

Mergesort

```
void mergesort(int n, int m)
```

```
{  int mid = (n+m)/2;  
  if (m-n <= 0)  
      return;  
  else if (m-n > 1) {  
      mergesort(n, mid);  
      mergesort(mid+1, m);  
  }  
  merge(n, m);  
}
```



Mergesort

```
void mergesort(int n, int m)
```

```
{ int mid = (n+m)/2;
```

```
if (m-n <= 0)
```

```
    return;
```

```
else if (m-n > 1) {
```

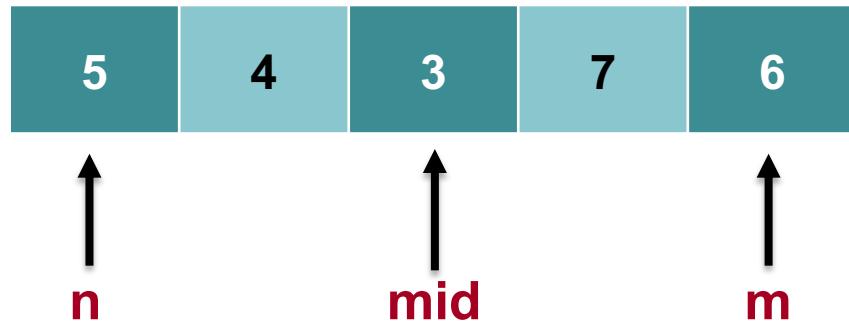
```
    mergesort(n, mid);
```

```
    mergesort(mid+1, m);
```

```
}
```

```
    merge(n, m);
```

```
}
```

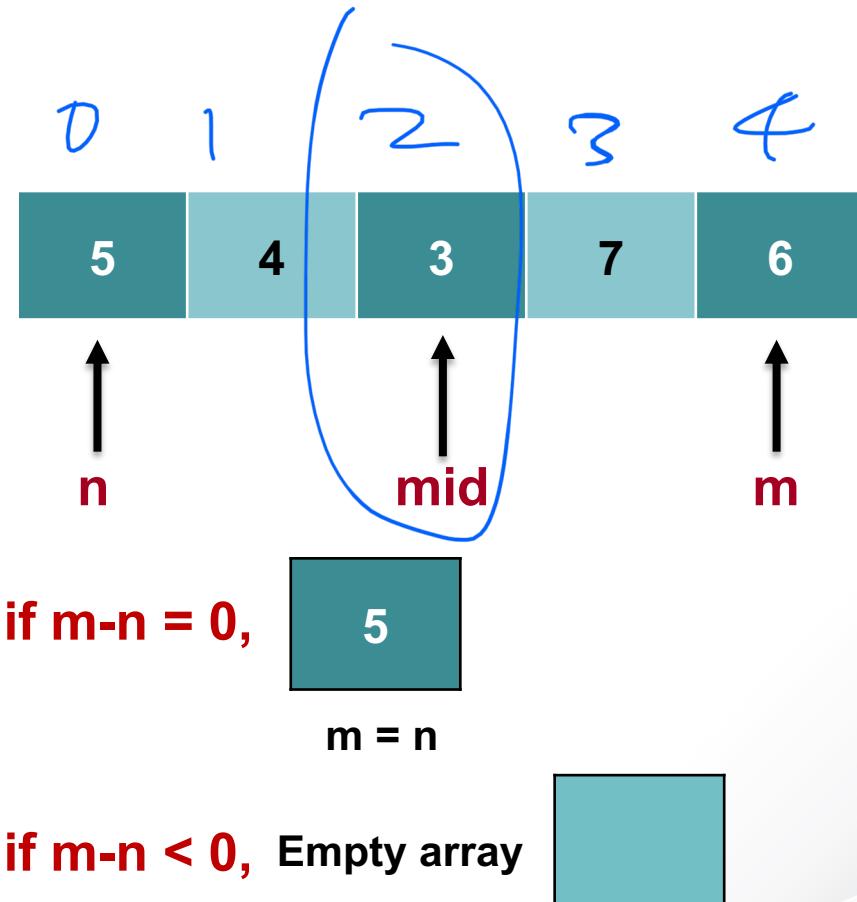


Mergesort

```

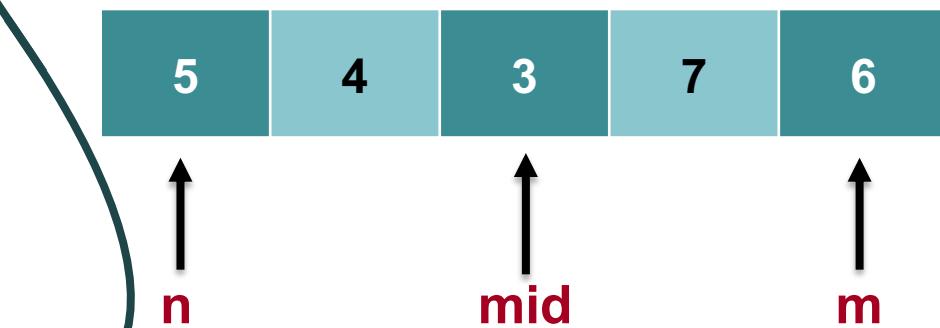
void mergesort(int n, int m)
{   int mid = (n+m)/2; (0+4)/2
    if (m-n <= 0) mid = 2
        return;
    else if (m-n > 1) {
        mergesort(n, mid);
        mergesort(mid+1, m);
    }
    merge(n, m);
}

```



Mergesort

```
void mergesort(int n, int m)
{
    int mid = (n+m)/2;
    if (m-n <= 0)
        return;
    else if (m-n > 1) {
        mergesort(n, mid);
        mergesort(mid+1, m);
    }
    merge(n, m);
}
```

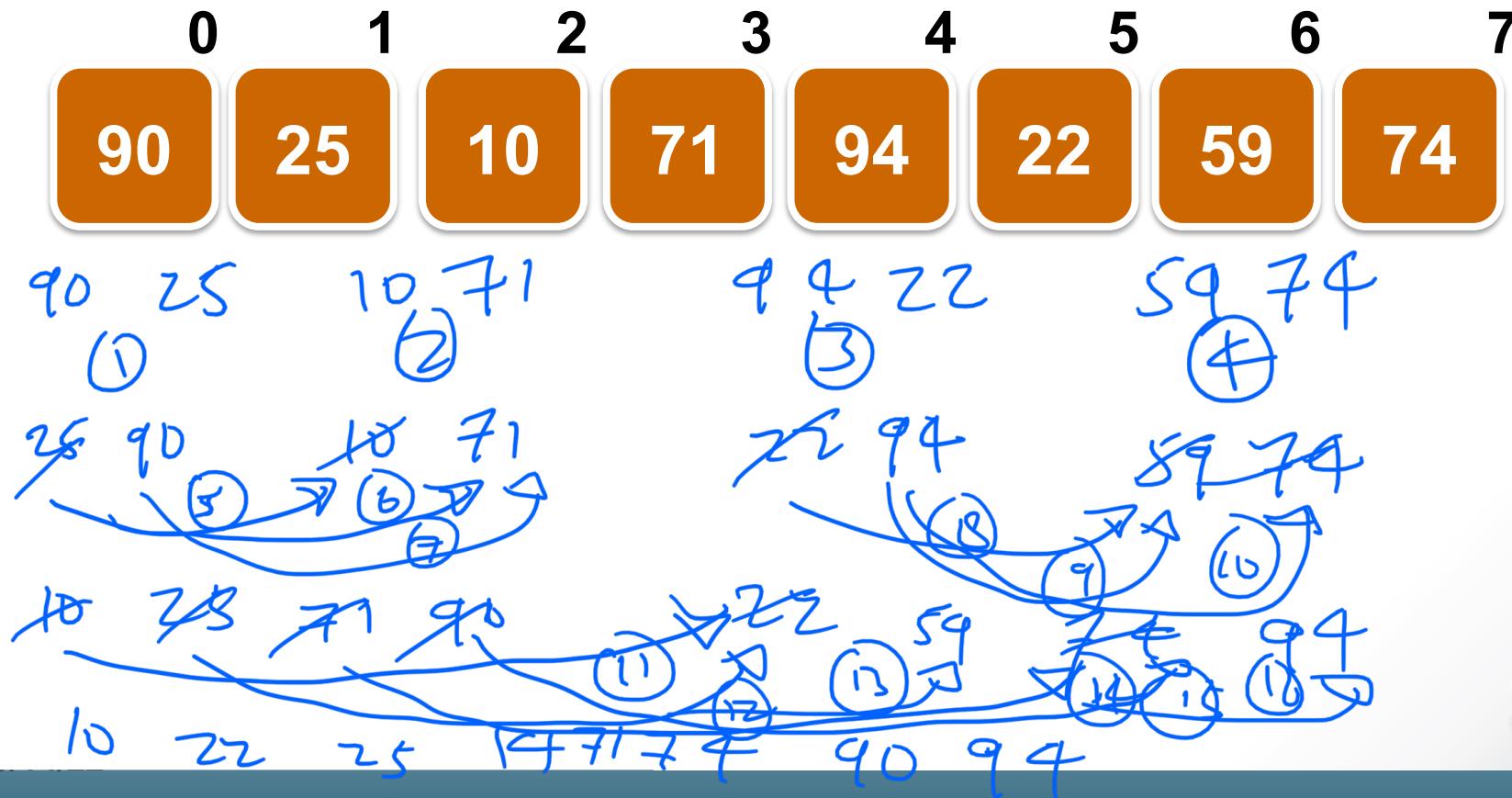




Mergesort (Example)

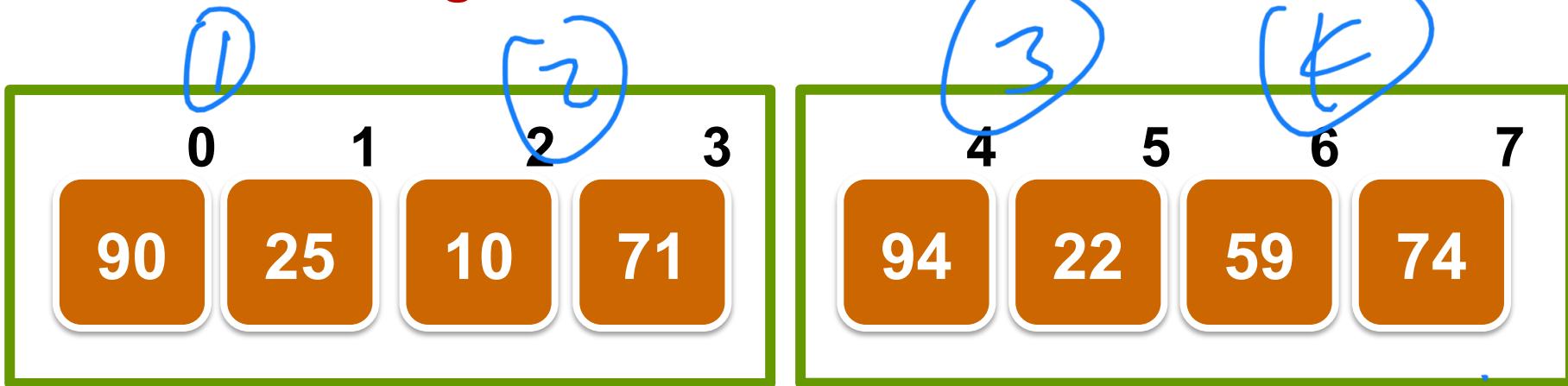
Mergesort

Sort in ascending order



Mergesort

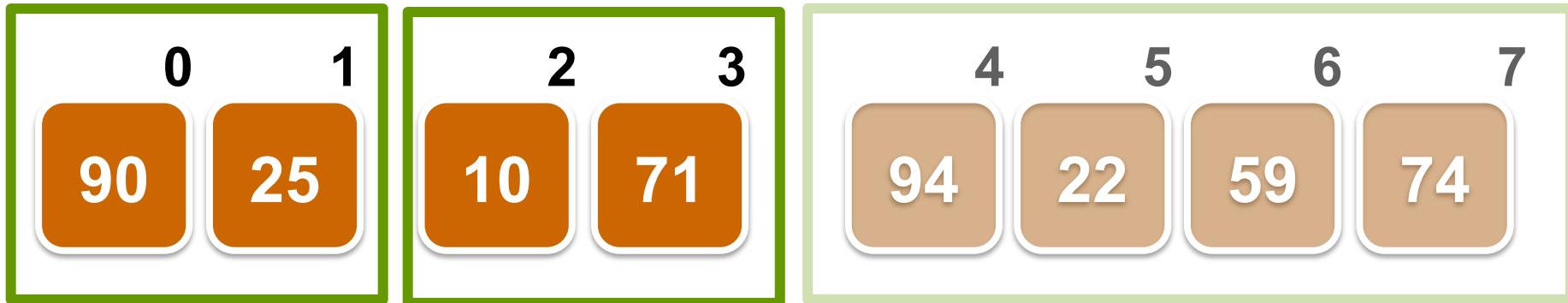
Sort in ascending order



Mergesort

Sort in ascending order

$$m-n \approx 1$$



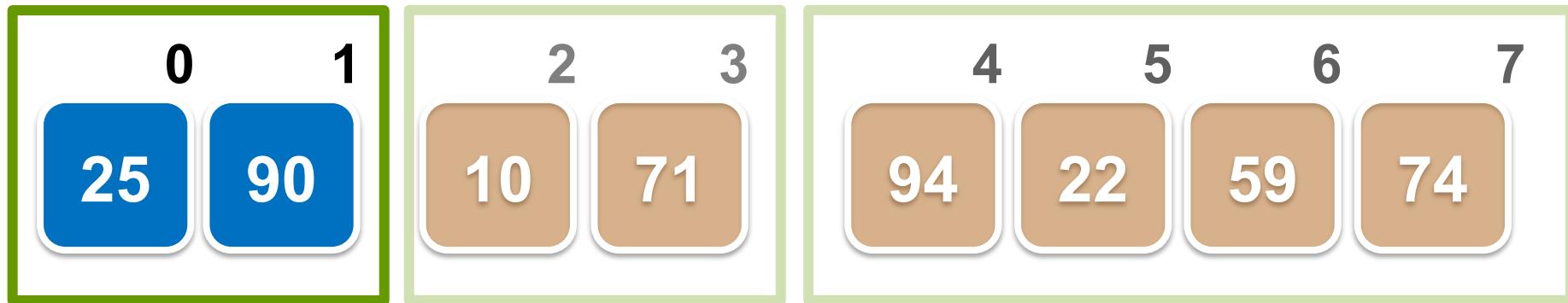
Mergesort

Sort in ascending order



Mergesort

Sort in ascending order



1 key comparison in merging

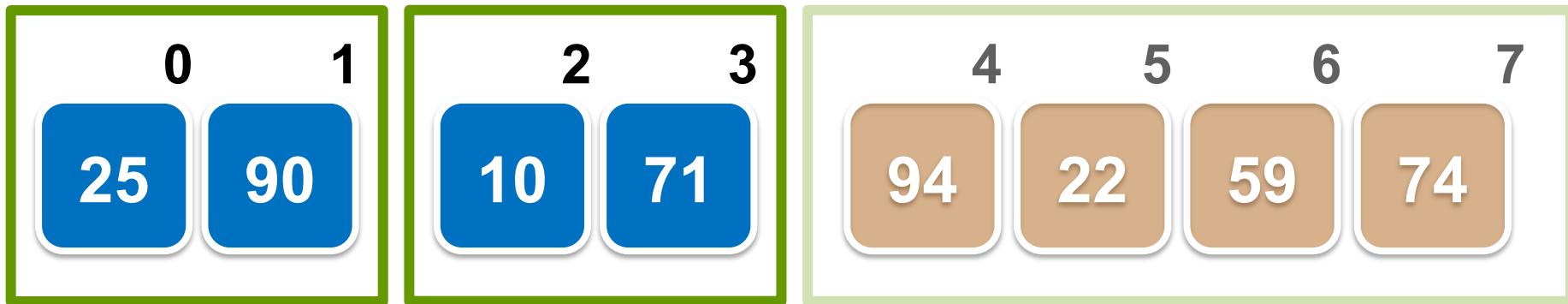
Mergesort

Sort in ascending order



Mergesort

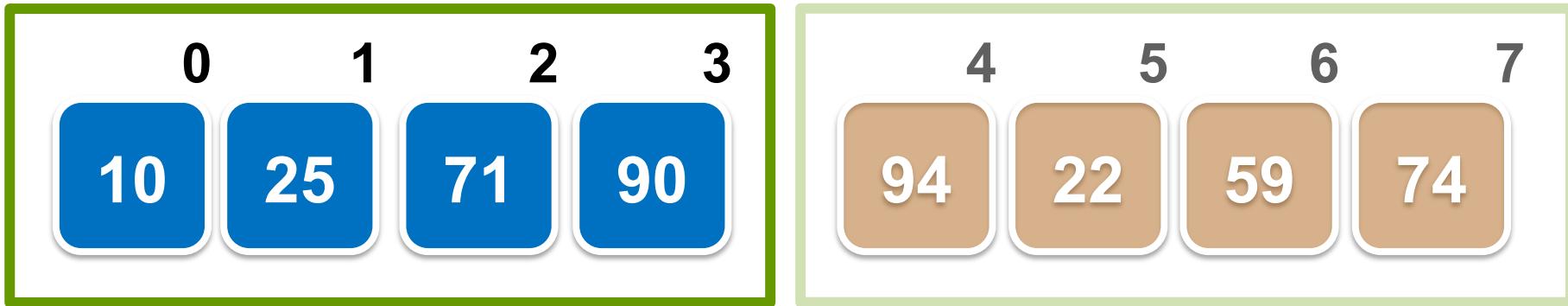
Sort in ascending order



1 key comparison in merging

Mergesort

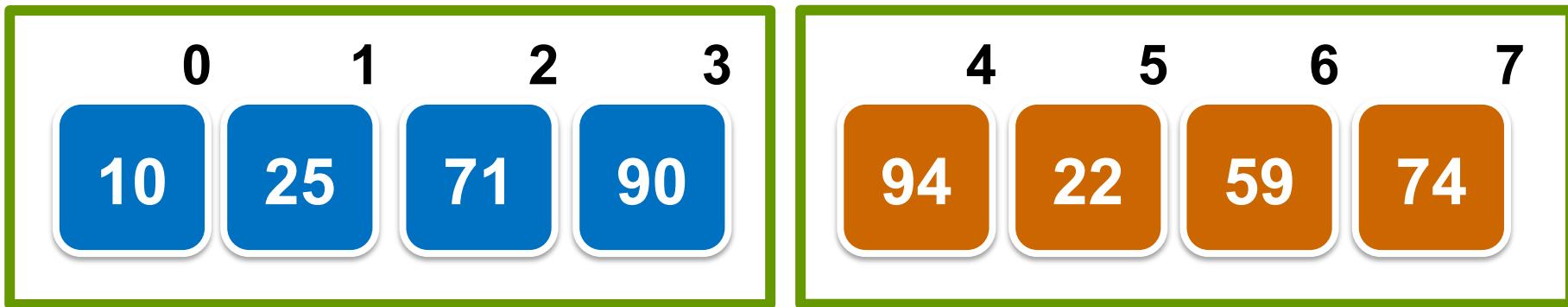
Sort in ascending order



3 key comparison in merging

Mergesort

Sort in ascending order



3 key comparison in merging

Mergesort

Sort in ascending order



Mergesort

Sort in ascending order



Mergesort

Sort in ascending order



1 key comparison in merging

Mergesort

Sort in ascending order



Mergesort

Sort in ascending order



1 key comparison in merging

Mergesort

Sort in ascending order

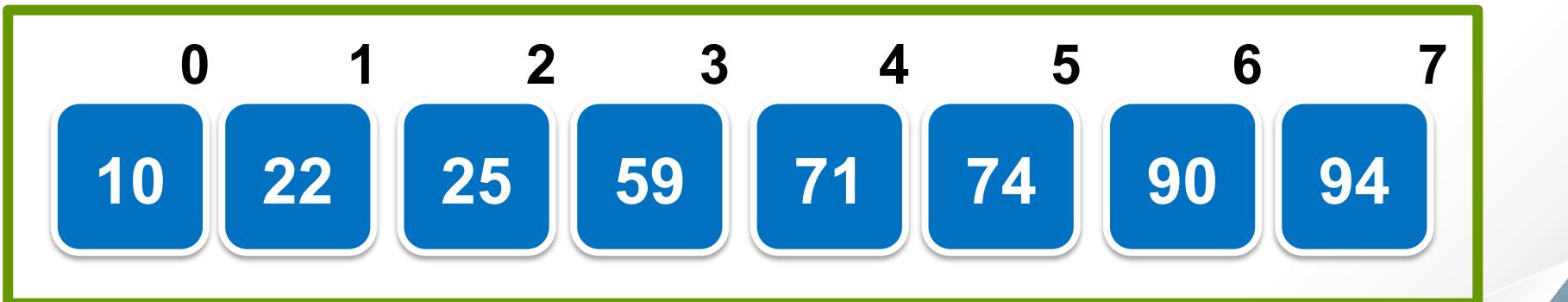


3 key comparison in merging

Mergesort



Sorted in ascending order



7 key comparisons in merging



Merge (Pseudo Code)

Merge (Pseudo Code)

```
void merge(int n, int m) {  
    if (m-n <= 0) return;  
    divide the list into 2 halves; // both halves are sorted  
    while (both halves are not empty) {  
        compare the 1st elements of the 2 halves; // 1 comparison  
        if (1st element of 1st half is smaller)  
            1st element of 1st half joins the end of the merged list;  
        else if (1st element of 2nd half is smaller)  
            move the 1st element of 2nd half to the end of the  
            merged list;
```



Merge (Pseudo Code)

```
void merge(int n, int m) {  
    if (m-n <= 0) return;  
    divide the list into 2 halves; // both halves are sorted  
    while (both halves are not empty) {  
        compare the 1st elements of the 2 halves; // 1 comparison  
        if (1st element of 1st half is smaller)  
            1st element of 1st half joins the end of the merged list;  
        else if (1st element of 2nd half is smaller)  
            move the 1st element of 2nd half to the end of the  
            merged list;
```



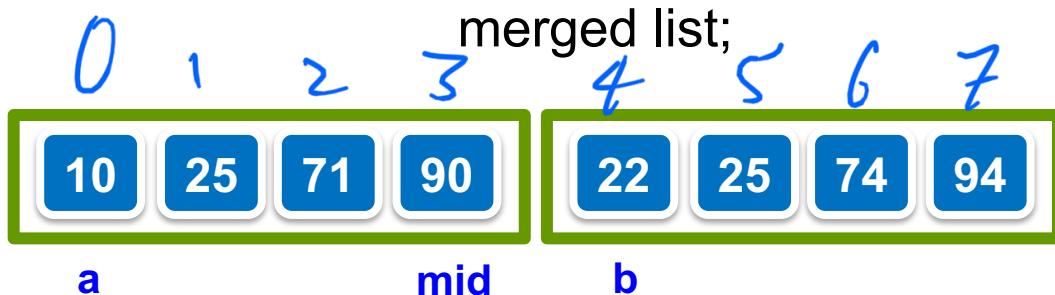
Merge (Pseudo Code)

```

void merge(int n, int m) {
    if (m-n <= 0) return;
    divide the list into 2 halves; // both halves are sorted
    while (both halves are not empty) {
        compare the 1st elements of the 2 halves; // 1 comparison
        if (1st element of 1st half is smaller)
            1st element of 1st half joins the end of the merged list;
        else if (1st element of 2nd half is smaller)
            move the 1st element of 2nd half to the end of the
    }
}

```

$\text{mid} = \frac{(n+m)}{2}$
 $\leq \equiv \geq$



Merge (Pseudo Code)

```
void merge(int n, int m) {
    if (m-n <= 0) return;
    divide the list into 2 halves; // both halves are sorted
    while (both halves are not empty) {
        compare the 1st elements of the 2 halves; // 1 comparison
        if (1st element of 1st half is smaller)
            1st element of 1st half joins the end of the merged list;
        else if (1st element of 2nd half is smaller)
            move the 1st element of 2nd half to the end of the
            merged list;
```



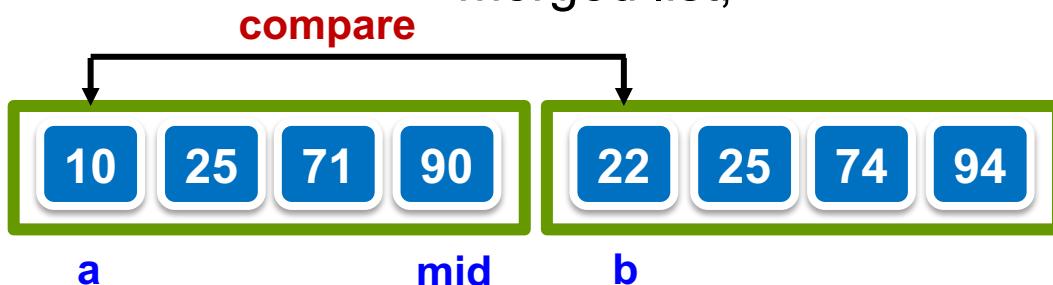
a

mid

b

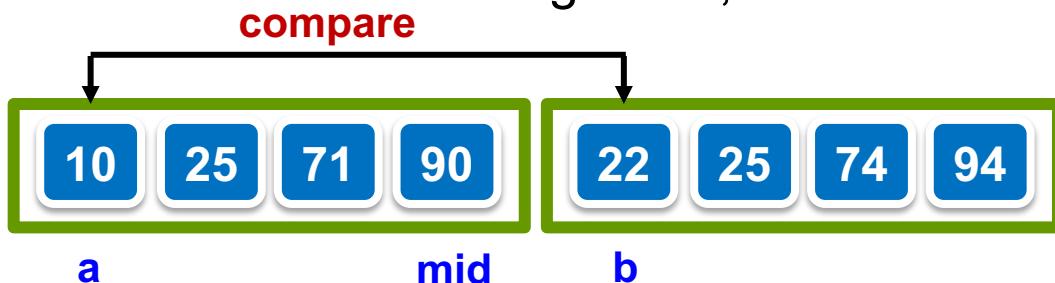
Merge (Pseudo Code)

```
void merge(int n, int m) {  
    if (m-n <= 0) return;  
    divide the list into 2 halves; // both halves are sorted  
    while (both halves are not empty) {  
        compare the 1st elements of the 2 halves; // 1 comparison  
        if (1st element of 1st half is smaller)  
            1st element of 1st half joins the end of the merged list;  
        else if (1st element of 2nd half is smaller)  
            move the 1st element of 2nd half to the end of the  
            merged list;
```



Merge (Pseudo Code)

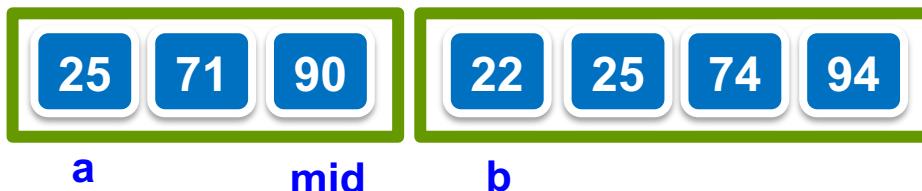
```
void merge(int n, int m) {  
    if (m-n <= 0) return;  
    divide the list into 2 halves; // both halves are sorted  
    while (both halves are not empty) {  
        compare the 1st elements of the 2 halves; // 1 comparison  
        if (1st element of 1st half is smaller)  
            1st element of 1st half joins the end of the merged list;  
        else if (1st element of 2nd half is smaller)  
            move the 1st element of 2nd half to the end of the  
            merged list;
```



Merge (Pseudo Code)

```
void merge(int n, int m) {  
    if (m-n <= 0) return;  
    divide the list into 2 halves; // both halves are sorted  
    while (both halves are not empty) {  
        compare the 1st elements of the 2 halves; // 1 comparison  
        if (1st element of 1st half is smaller)  
            1st element of 1st half joins the end of the merged list;  
        else if (1st element of 2nd half is smaller)  
            move the 1st element of 2nd half to the end of the  
            merged list;  
    }  
}
```

10



Merge (Pseudo Code)

```
void merge(int n, int m) {  
    if (m-n <= 0) return;  
    divide the list into 2 halves; // both halves are sorted  
    while (both halves are not empty) {  
        compare the 1st elements of the 2 halves; // 1 comparison  
        if (1st element of 1st half is smaller)  
            1st element of 1st half joins the end of the merged list;  
        else if (1st element of 2nd half is smaller)  
            move the 1st element of 2nd half to the end of the  
            merged list;
```



Merge (Pseudo Code)

```
else { // the 1st elements of the 2 halves are equal

    if (they are the last elements) break;

    1st element of 1st half joins end of the merged list;

    move the 1st element of 2nd half to the end of the
    merged list;

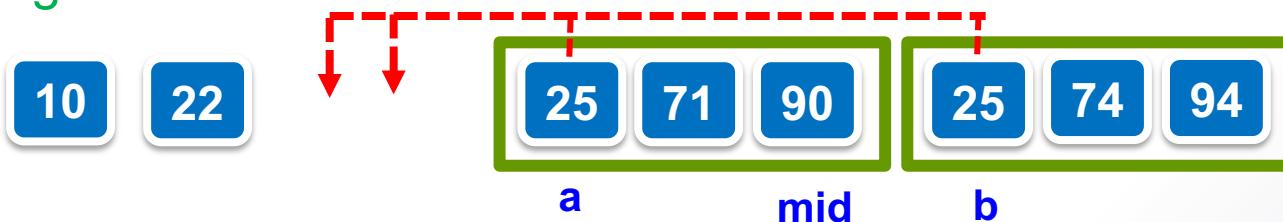
}

} // end of while loop;

} // end of merge
```

Merge (Pseudo Code)

```
else { // the 1st elements of the 2 halves are equal  
    if (they are the last elements) break;  
    1st element of 1st half joins end of the merged list;  
    move the 1st element of 2nd half to the end of the  
    merged list;  
}  
} // end of while loop;  
} // end of merge
```



Merge (Pseudo Code)

```
else { // the 1st elements of the 2 halves are equal
    if (they are the last elements) break;
    1st element of 1st half joins end of the merged list;
    move the 1st element of 2nd half to the end of the
    merged list;
}
} // end of while loop;
} // end of merge
```

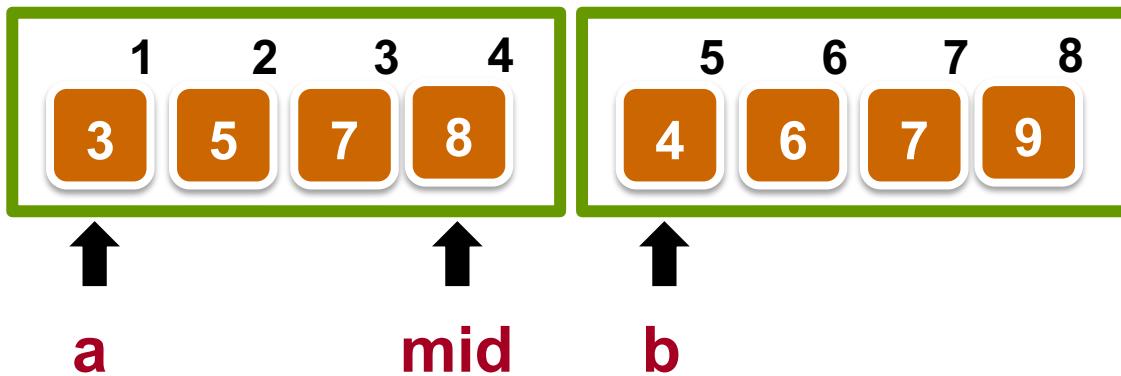
Challenge:
How to do it without auxiliary storage for the merged list?



Merge (Case Scenarios)

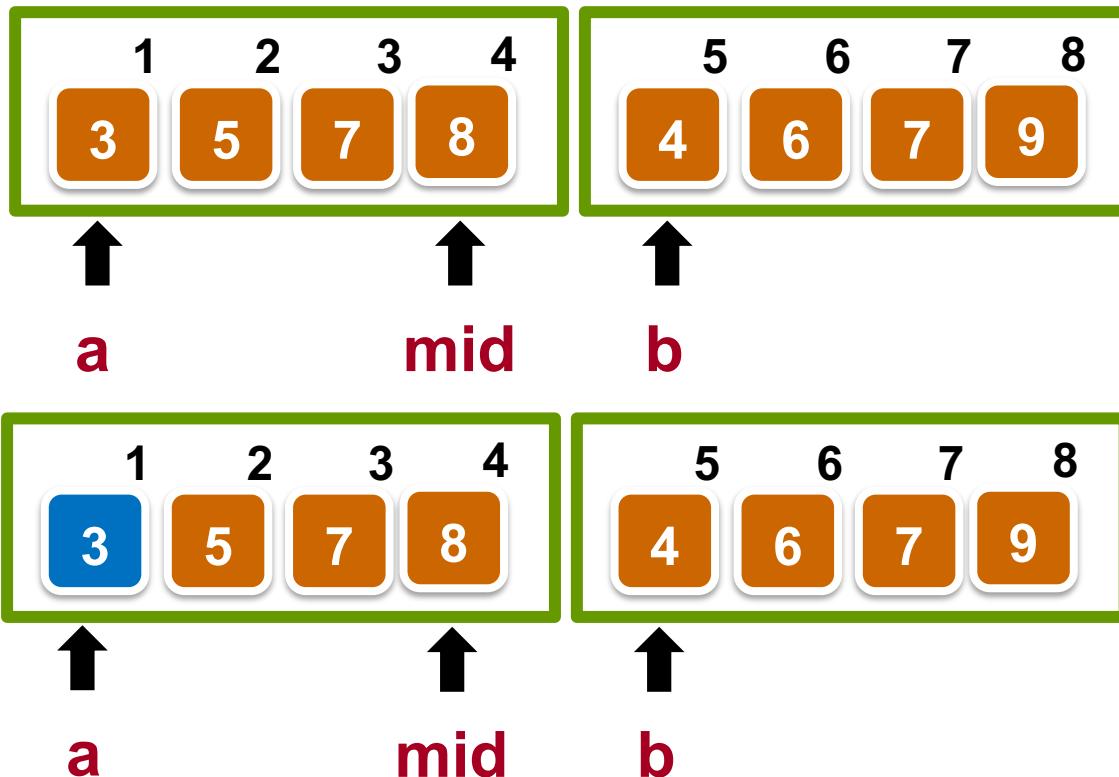
Merge (Case Scenarios)

Case 1: 1st element of 1st half is smaller



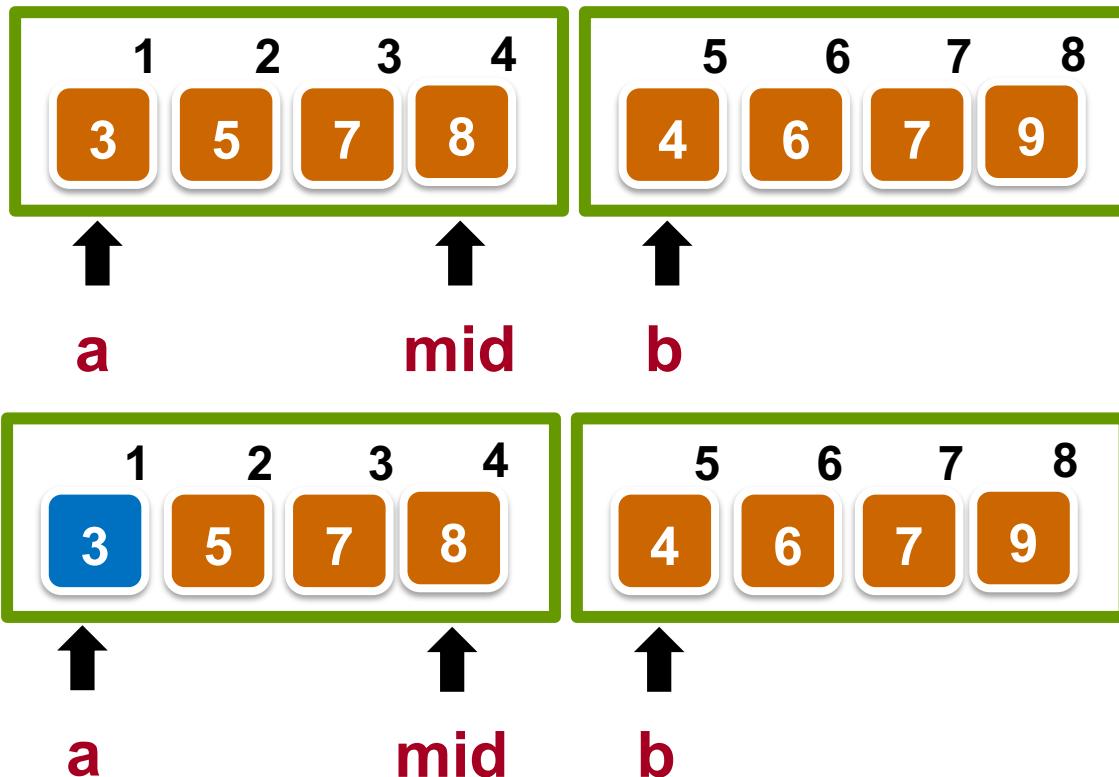
Merge (Case Scenarios)

Case 1: 1st element of 1st half is smaller



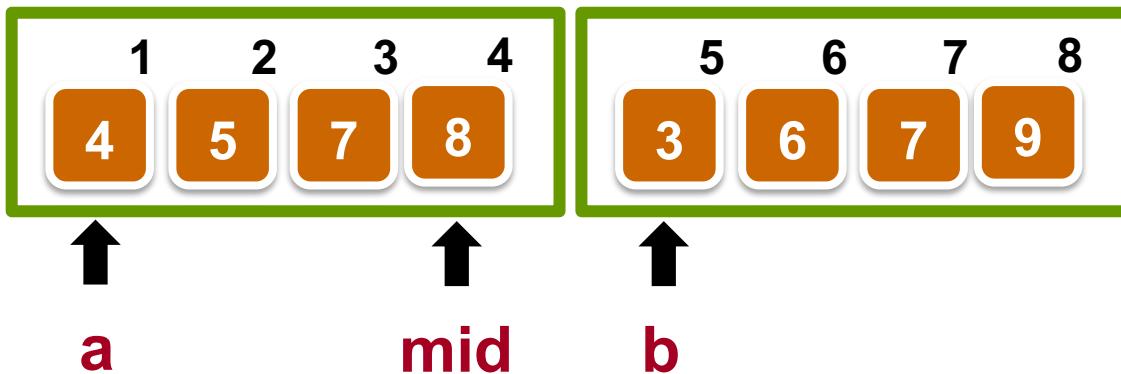
Merge (Case Scenarios)

Case 1: 1st element of 1st half is smaller



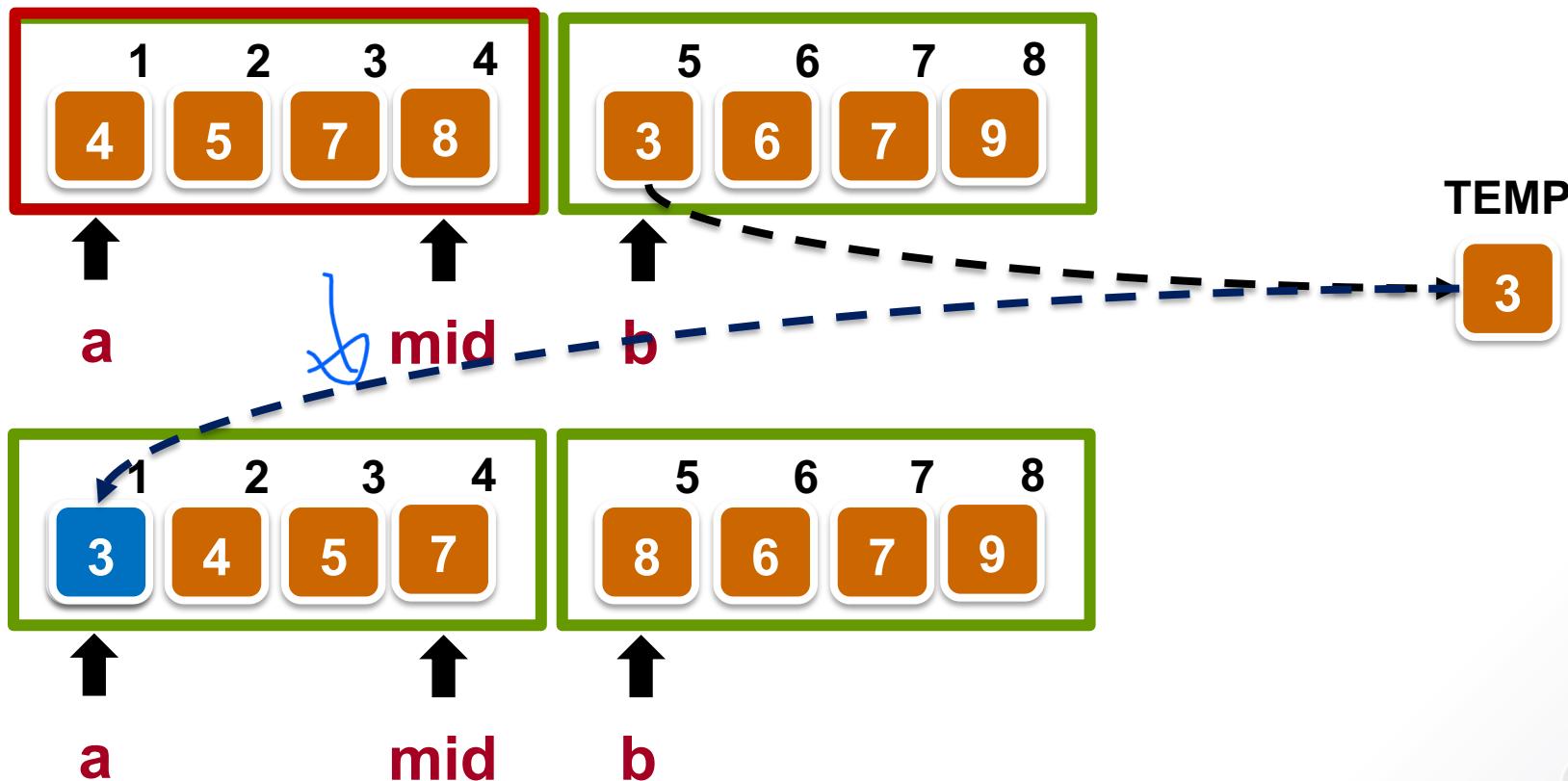
Merge (Case Scenarios)

Case 2: 1st element of 2nd half is smaller



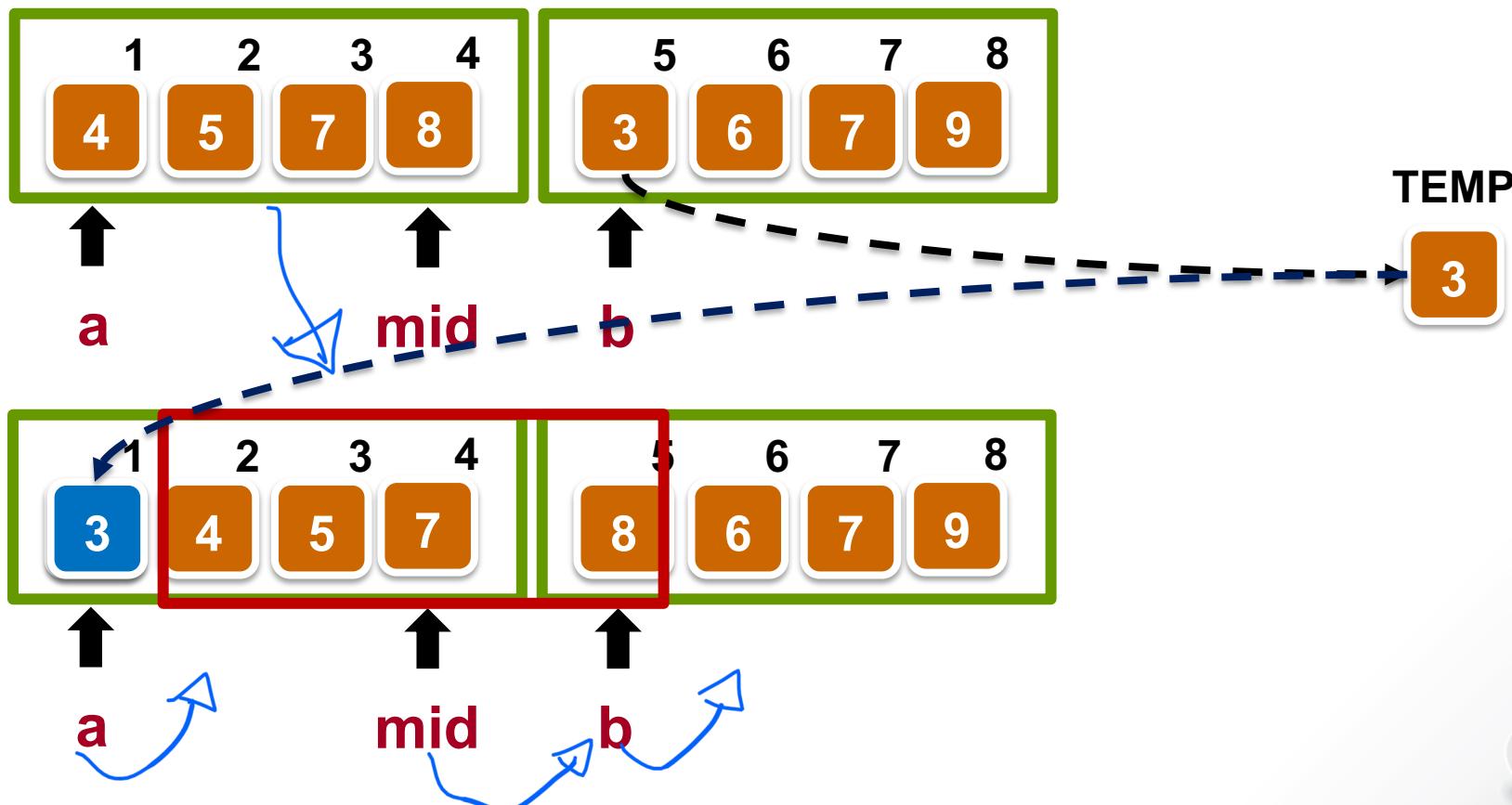
Merge (Case Scenarios)

Case 2: 1st element of 2nd half is smaller



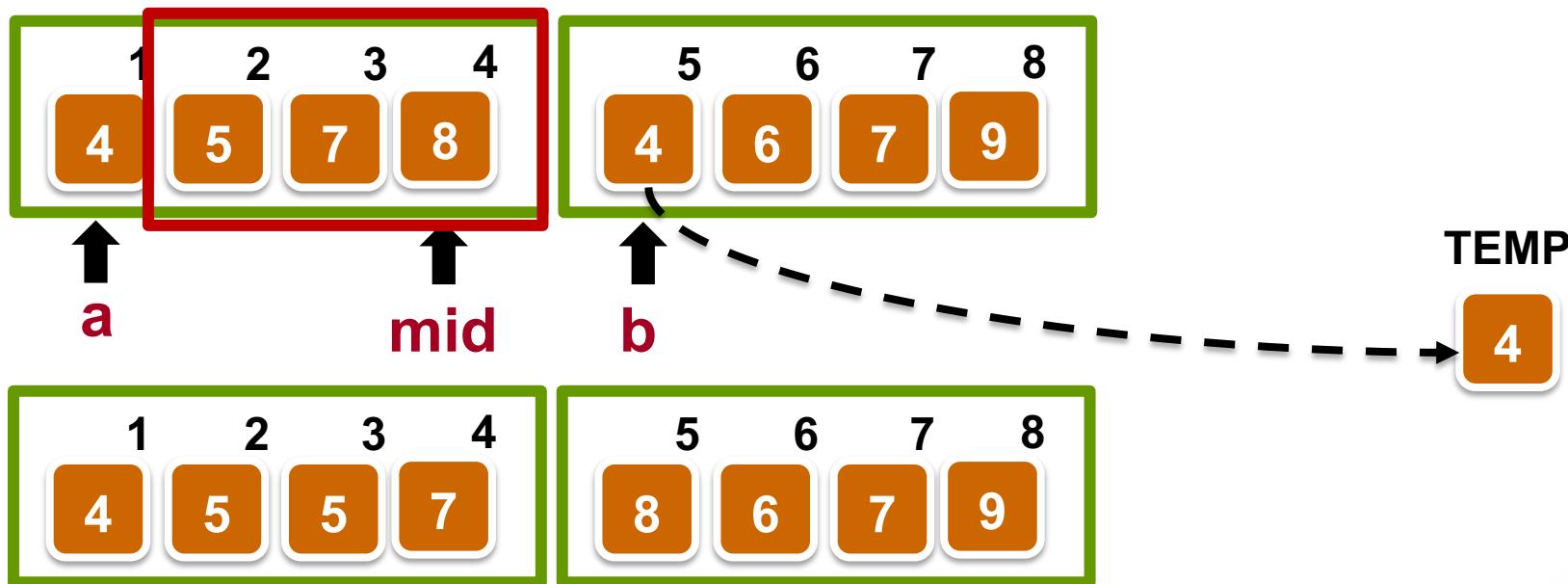
Merge (Case Scenarios)

Case 2: 1st element of 2nd half is smaller



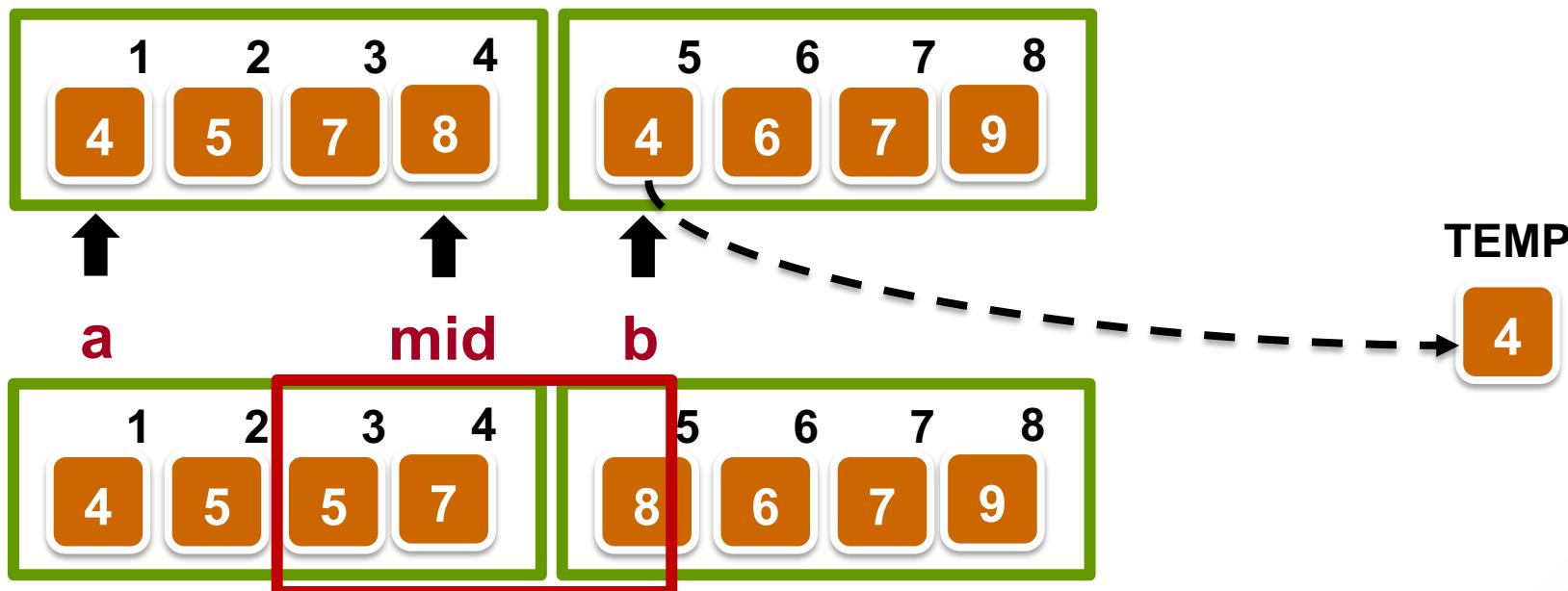
Merge (Case Scenarios)

Case 3: 1st element of 2nd half is equal



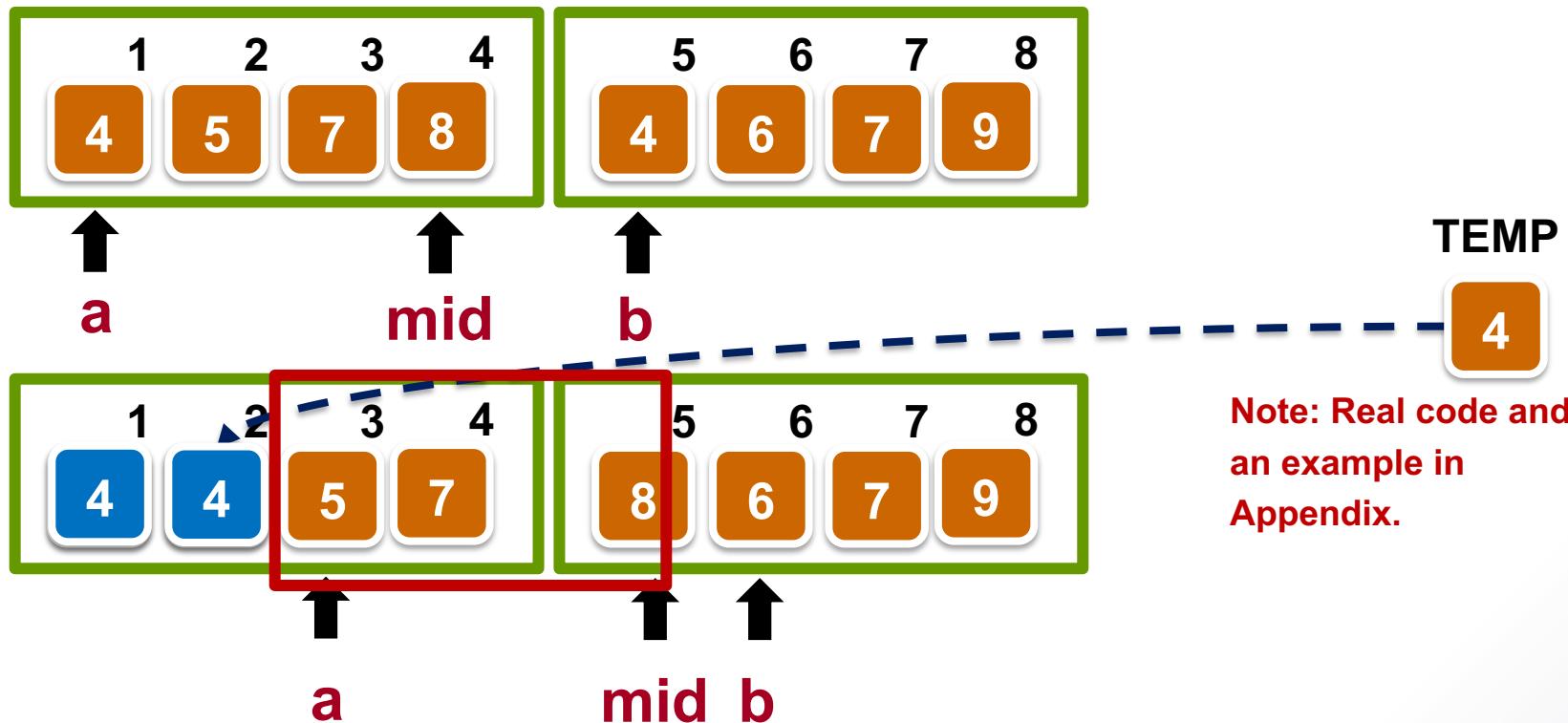
Merge (Case Scenarios)

Case 3: 1st element of 2nd half is equal



Merge (Case Scenarios)

Case 3: 1st element of 2nd half is equal



Can we just copy the first 4 twice?



Mergesort Algorithm (Recap)

Mergesort Algorithm (Recap)

- Since merging is performed directly on the original array, swapping and shifting are needed
- `mergesort()` partitions a contiguous array of elements between index `n` and `m` into two subarrays

```
void mergesort(int n, int m)
{
    int mid = (n+m)/2;
    if (m-n <= 0)
        return;
    else if (m-n > 1) {
        mergesort(n, mid);
        mergesort(mid+1, m);
    }
    merge(n, m);
}
```

Mergesort Algorithm (Recap)

- Since merging is performed directly on the original array, swapping and shifting are needed
- `mergesort()` partitions a contiguous array of elements between index n and m into two subarrays
- Recursively partitions until $m-n \leq 0$, then merge the resulting two subarrays

```
void mergesort(int n, int m)
{
    int mid = (n+m)/2;
    if (m-n <= 0)
        return;
    else if (m-n > 1)
        ....
}
```

Mergesort Algorithm (Recap)

- Since merging is performed directly on the original array, swapping and shifting are needed
- `mergesort()` partitions a contiguous array of elements between index `n` and `m` into two subarrays
- Recursively partitions until `m-n<=0`, then merge the resulting two subarrays
- `merge()` function merges two sub-arrays of elements between index `n` and ‘`mid`’, and between ‘`mid+1`’ and `m`

```
void mergesort(int n, int m)
{
    int mid = (n+m)/2;
    if (m-n <= 0)
        .....
    merge(n, m);
}
```

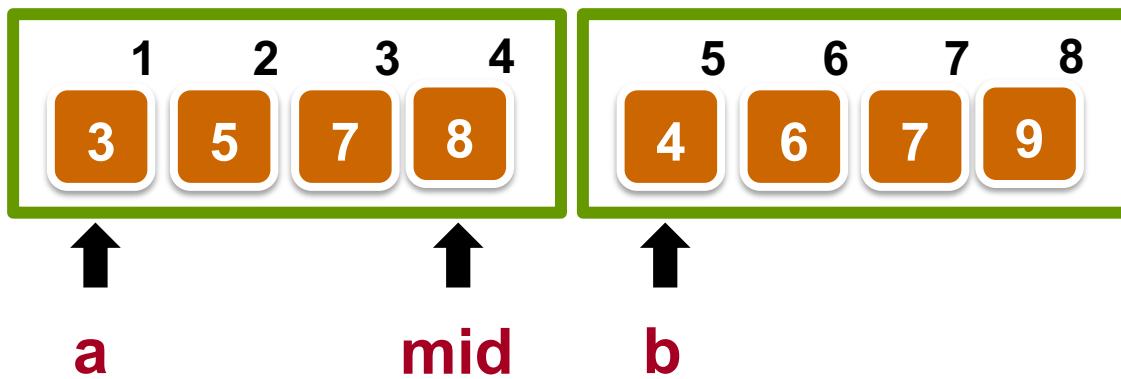
Mergesort Algorithm (Recap)

- Since merging is performed directly on the original array, swapping and shifting are needed
- `mergesort()` partitions a contiguous array of elements between index `n` and `m` into two subarrays
- Recursively partitions until `m-n<=0`, then merge the resulting two subarrays
- `merge()` function merges two sub-arrays of elements between index `n` and ‘`mid`’, and between ‘`mid+1`’ and `m`
- During merging, one element from each subarray is compared and the smaller one is inserted into new list

```
void mergesort(int n, int m)
{
    .....
    merge(n, m);
}
```

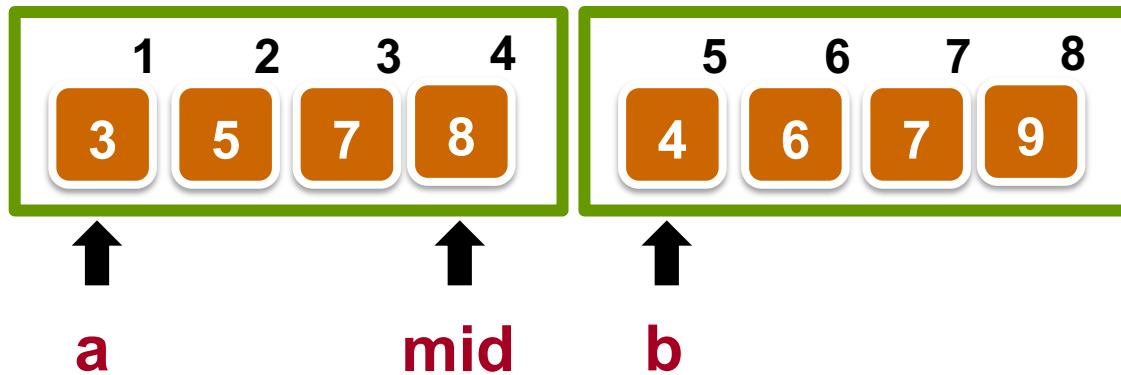
Mergesort Algorithm (Recap)

- Left subarray runs from n to ‘mid’ with a as running index; right subarray runs from $mid+1$ to m with b as running index



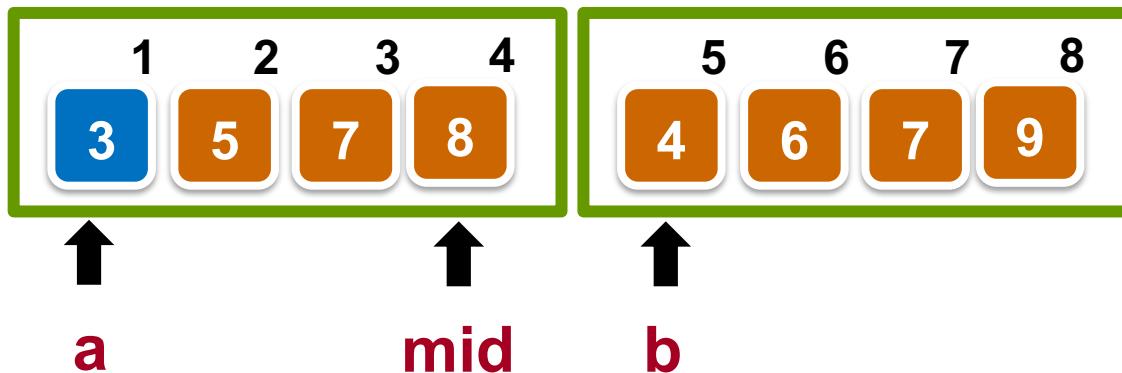
Mergesort Algorithm (Recap)

- Left subarray runs from n to ‘mid’ with a as running index; right subarray runs from $mid+1$ to m with b as running index
- $\text{slot}[a]$ is the head element of left subarray, $\text{slot}[b]$ is the head element of right subarray



Mergesort Algorithm (Recap)

- Left subarray runs from n to ‘mid’ with a as running index; right subarray runs from $mid+1$ to m with b as running index
- $\text{slot}[a]$ is the head element of left subarray, $\text{slot}[b]$ is the head element of right subarray
- During merging, both left and right subarrays shrink towards the right to make space for the newly merged array



Mergesort Algorithm (Recap)

Case 1: if $\text{slot}[a] < \text{slot}[b]$, there is nothing much to do since smaller element already in correct position (with regard to the merged array)



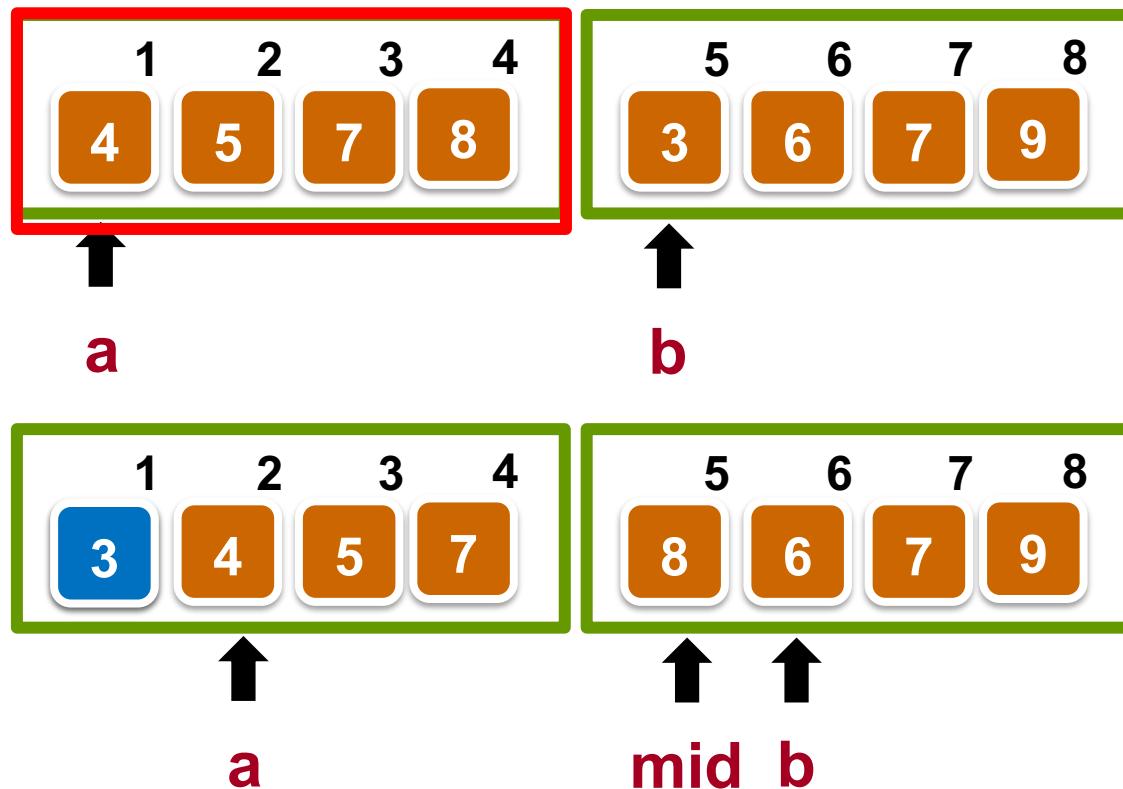
↑
a **mid** **b**



↑
a **mid** **b**

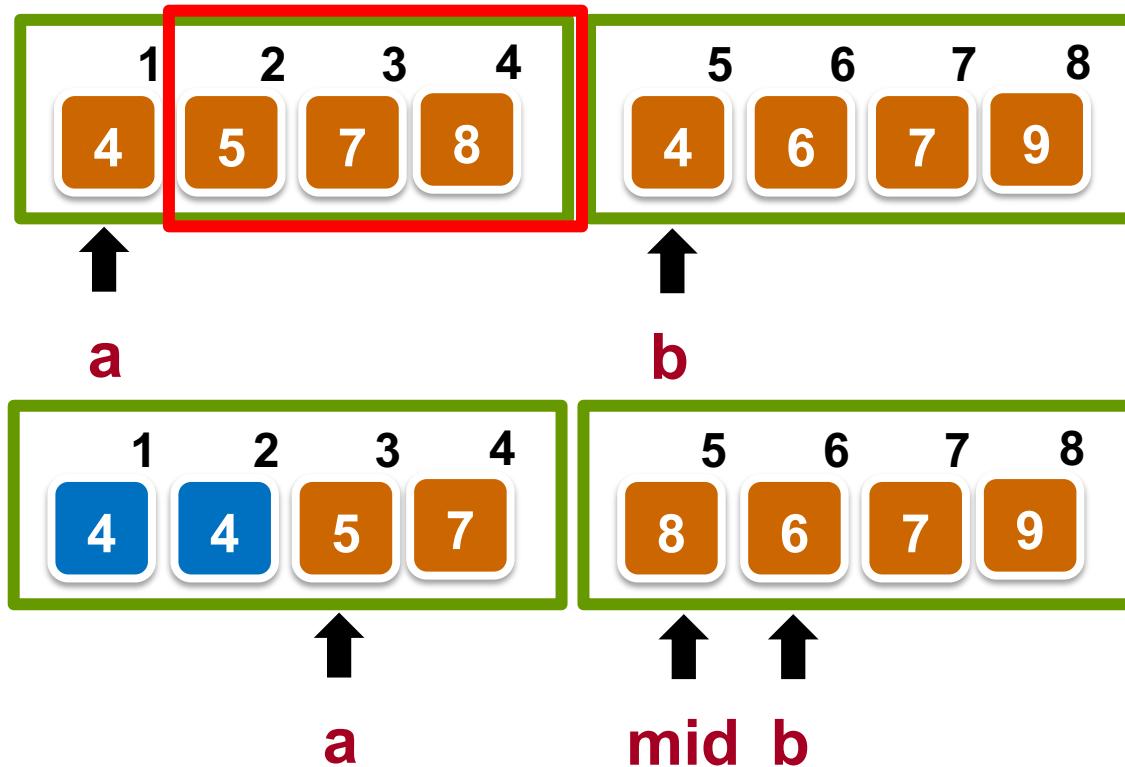
Mergesort Algorithm (Recap)

Case 2: if $\text{slot}[a] > \text{slot}[b]$, then Right-shift (by one) elements of left subarray from index a to ‘ mid ’ and insert element at $\text{slot}[b]$ into $\text{slot}[a]$

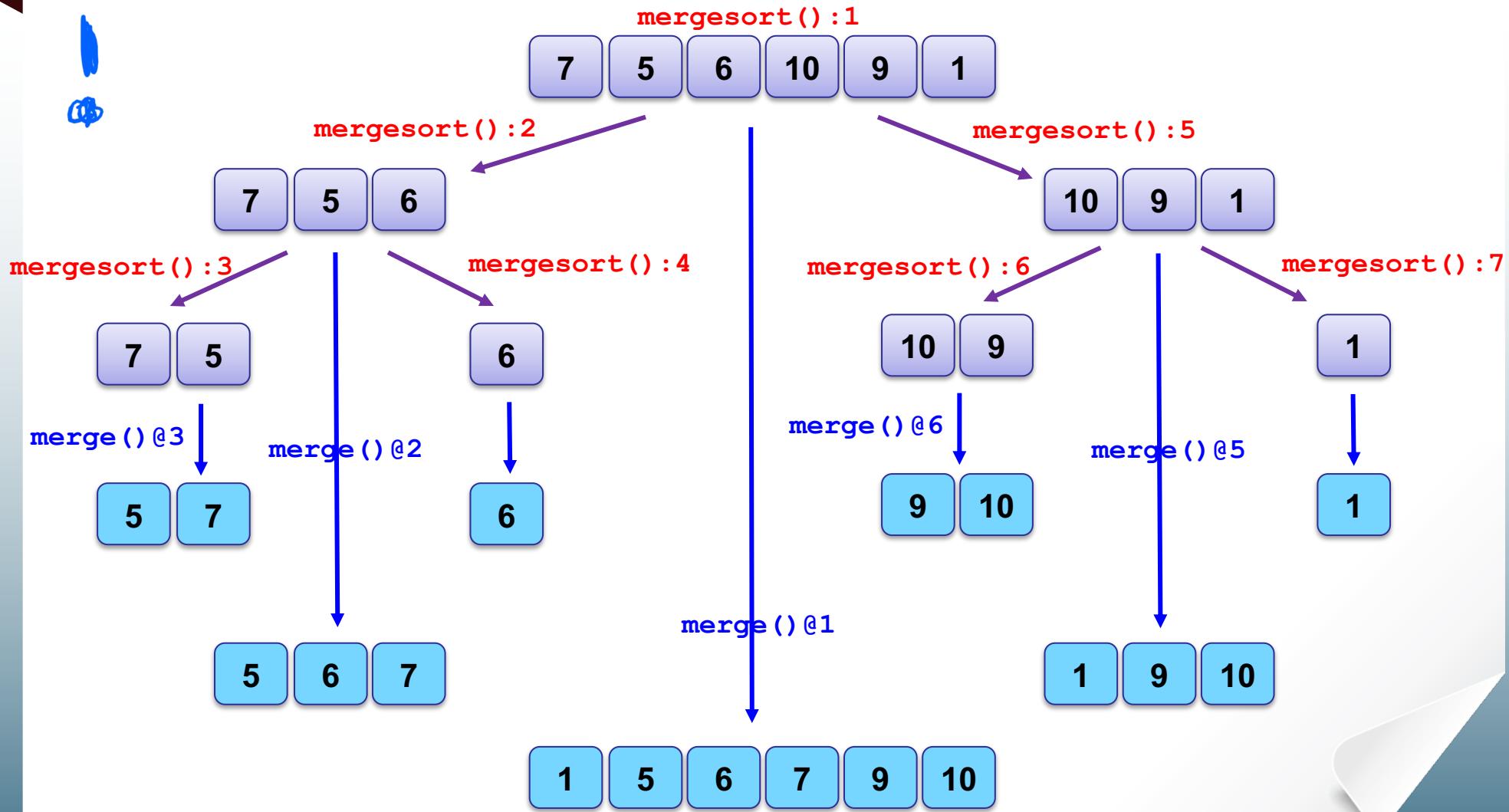


Mergesort Algorithm (Recap)

Case 3: if `slot[a] == slot[b]`, then `slot[a]` is in the correct position. So, move `slot[b]` next to beside `slot[a]`, by Right-shifting and swapping



Call Graph of Mergesort

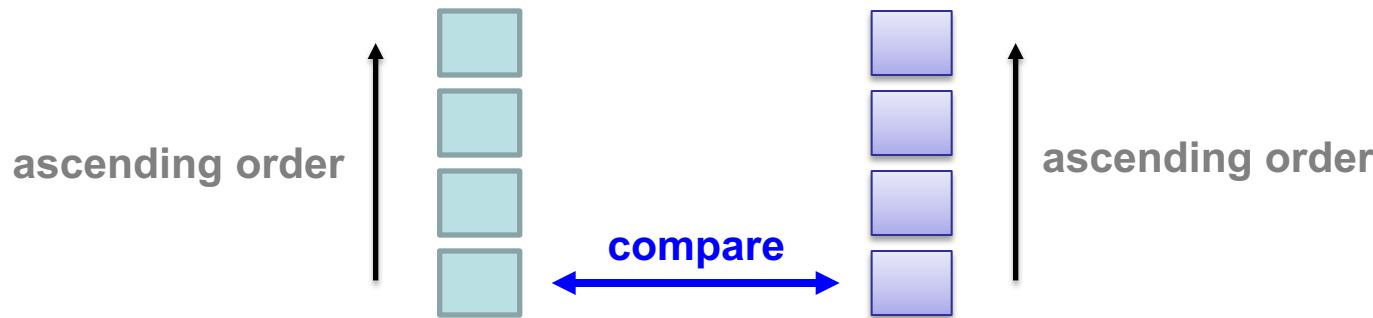




Complexity of Mergesort

Complexity of merge()

- After each comparison of keys from the two sub-lists, at least one element is moved to the new merged list and never compared again



- After the last key comparison, at least two elements will be moved into the merged list
- Thus, to merge two sub-lists of n elements in total, the number of key comparisons needed is at most (worst case) $n - 1$
 - How about best case?

$$\text{best} = \frac{n}{2}$$

7 5 8 6
3 4 2 1

Complexity of Mergesort

```
void mergesort(int s, int e) // s=start, e=end
{
    int mid = (s+e)/2;
    if (e-s <= 0) return;
    else if (e-s > 1) {
        mergesort(s, mid);
        mergesort(mid+1, e);
    }
    merge(s, e);
}
```

$$\longrightarrow W(1) = 0$$

$$\longrightarrow W(n/2)$$

$$\longrightarrow W(n/2)$$

$$\longrightarrow \text{Worst case: } n-1$$

best case $\frac{n}{2}$

$W(n)$

Complexity of Mergesort

Mergesort performance (assume $n = 2^k$)

$$\log_2(n) = \ln(n)/\ln(2)$$

Worst case :

$$W(1) = 0,$$

$$W(n) = W(n/2) + W(n/2) + n-1$$

Or

$$2W(n/2) + n - 1$$

$$W(2^k) = 2W(2^{k-1}) + 2^k - 1$$

$$= 2(2W(2^{k-2}) + 2^{k-1} - 1) + 2^k - 1$$

$$= 2^2W(2^{k-2}) + 2^k - 2 + 2^k - 1$$

$$= 2^2(2W(2^{k-3}) + 2^{k-2} - 1) + 2^k - 2 + 2^k - 1$$

$$= 2^3W(2^{k-3}) + 2^k - 2^2 + 2^k - 2 + 2^k - 1$$

...

$$= 2^kW(2^{k-k}) + k2^k - (1 + 2 + 4 + \dots + 2^{k-1})$$

$$= k2^k - (2^k - 1)$$

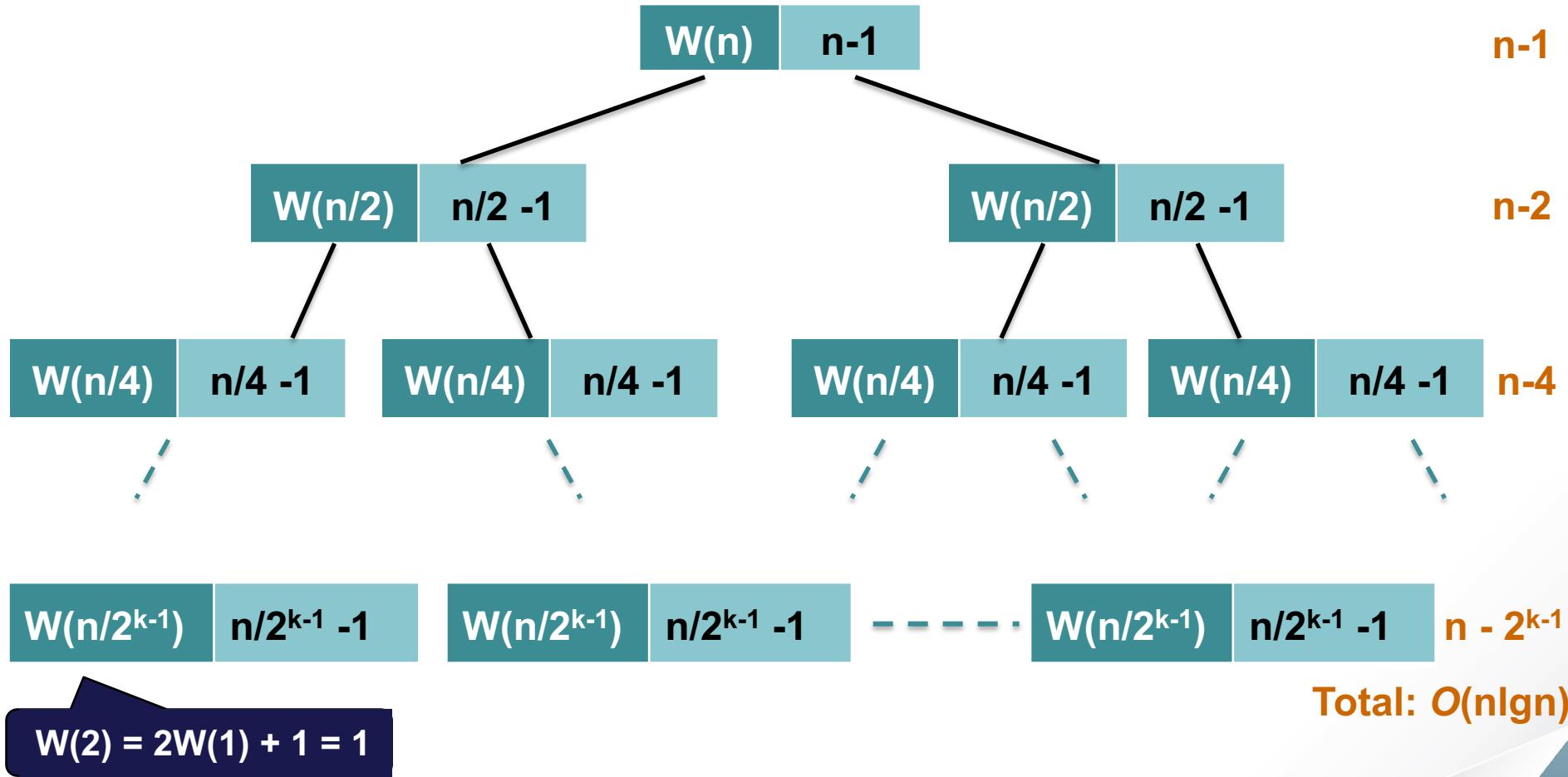
$$= n \lg n - (n - 1)$$

$$= O(n \lg n)$$

$$k = \lg n$$

Geometric series

Visually :Recursion Tree



Height of tree is $k = O(\lg n)$

Evaluation of Mergesort

😊 Strengths:

- 👉 Simple and good runtime behavior
- 👉 Easy to implement when using linked list

😢 Weaknesses:

- 👉 Difficult to implement for contiguous data storage such as array without auxiliary storage (requires data movements during merging)

Summary

- Mergesort uses the **Divide and Conquer** approach.
 - It recursively divide a list into two halves of approximately equal sizes, until the sub-list is too small (no more than two elements).
 - Then, it recursively merges two sorted sub-lists into one sorted list.
- The worst-case running time for **merging** two sorted lists of total size n is $n - 1$ key comparisons.
- The running time of Mergesort is $O(nlgn)$.



SC2001/CE2101/CZ2101: **Algorithm Design and Analysis**

Appendix

(Merge operation in Mergesort)

Instructor: Asst. Prof. LIN Shang-Wei

Courtesy of Dr. Ke Yiping, Kelly's slides

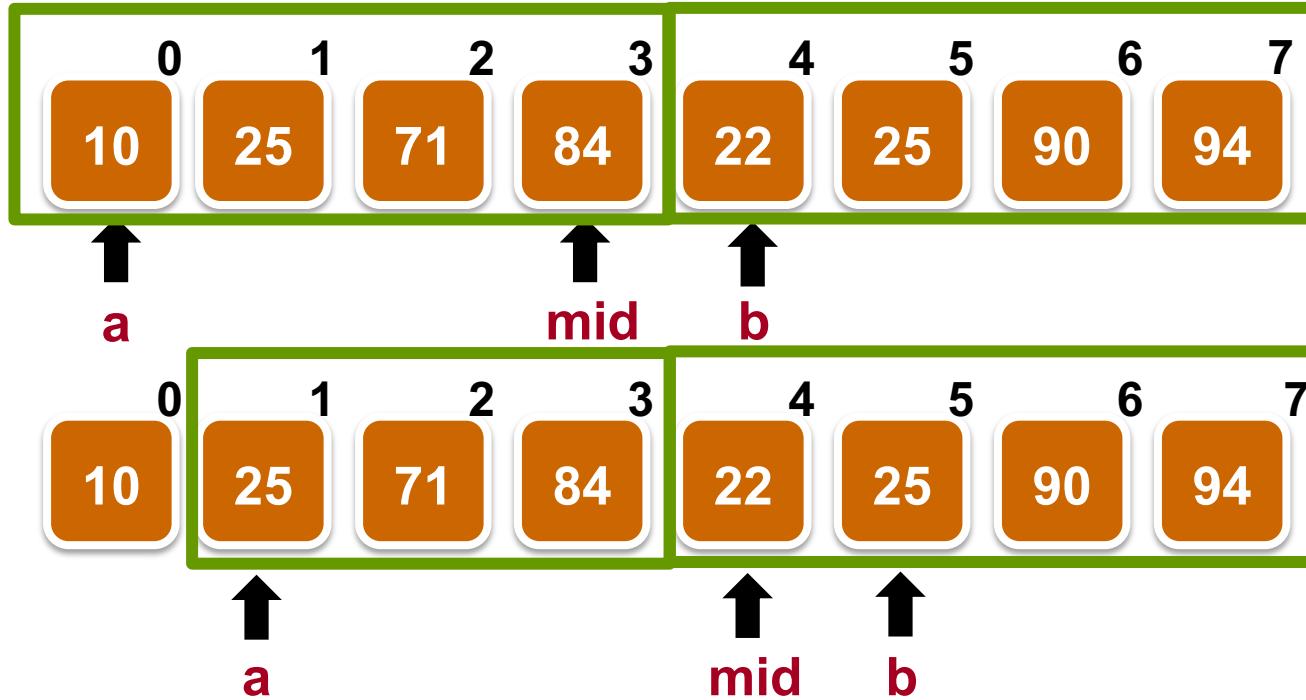
Merge Function

```
void merge(int n, int m)
{
    int mid = (n+m)/2;
    int a = n, b = mid+1, i, tmp;
    if (m-n <= 0) return;
    while (a <= mid && b <= m) {
        cmp = compare(slot[a], slot[b]);
        if (cmp > 0) { //slot[a] > slot[b]
            tmp = slot[b++];
            for (i = ++mid; i > a; i--)
                slot[i] = slot[i-1];
        }
    }
}
```

Merge Function

```
        slot[a++] = tmp;
    } else if (cmp < 0) //slot[a] < slot[b]
        a++;
    else { //slot[a] == slot[b]
        if (a == mid && b == m)
            break;
        tmp = slot[b++];
        a++;
        for (i = ++mid; i > a; i--)
            slot[i] = slot[i-1];
        slot[a++] = tmp;
    }
} // end of while loop;
} // end of merge
```

Merge Operation



a : the 1st element of the 1st half

mid : the last element of the 1st half

b : the 1st element of the 2nd half

Parameters for merge:

n:0, m: 7

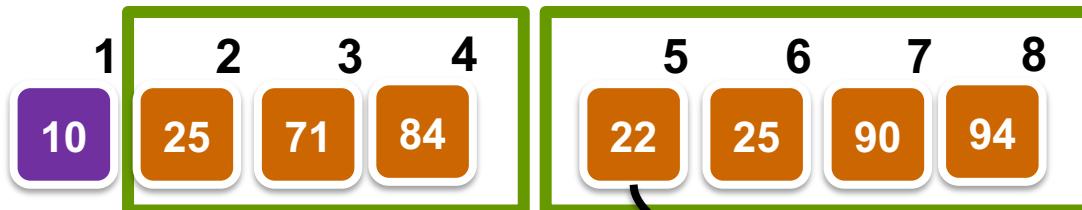
mid = $(0+7)/2 = 3;$

a = n; **b** = mid+1;

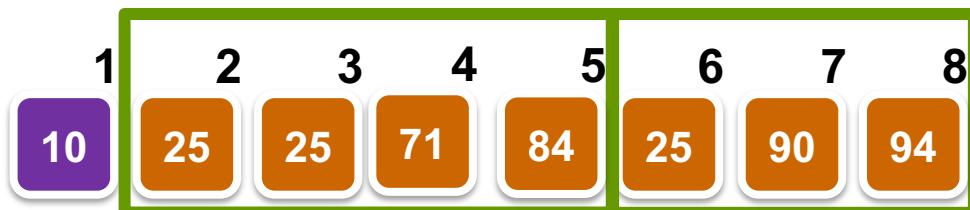
Comparison:

slot[a] < slot[b]

Merge Operation

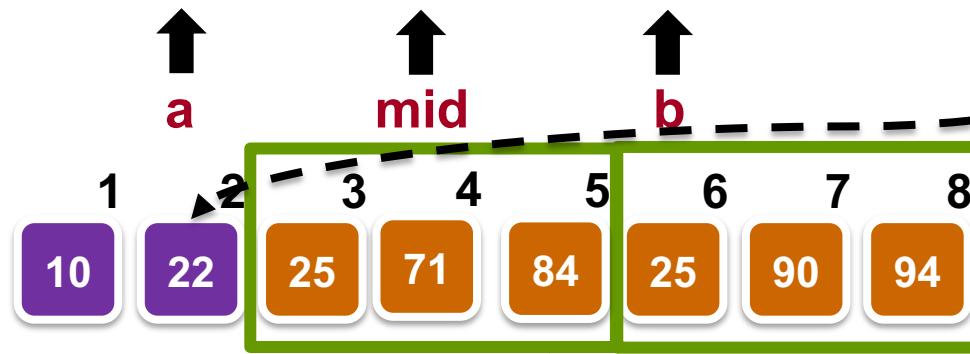


Comparison:
 $\text{slot}[a] > \text{slot}[b]$



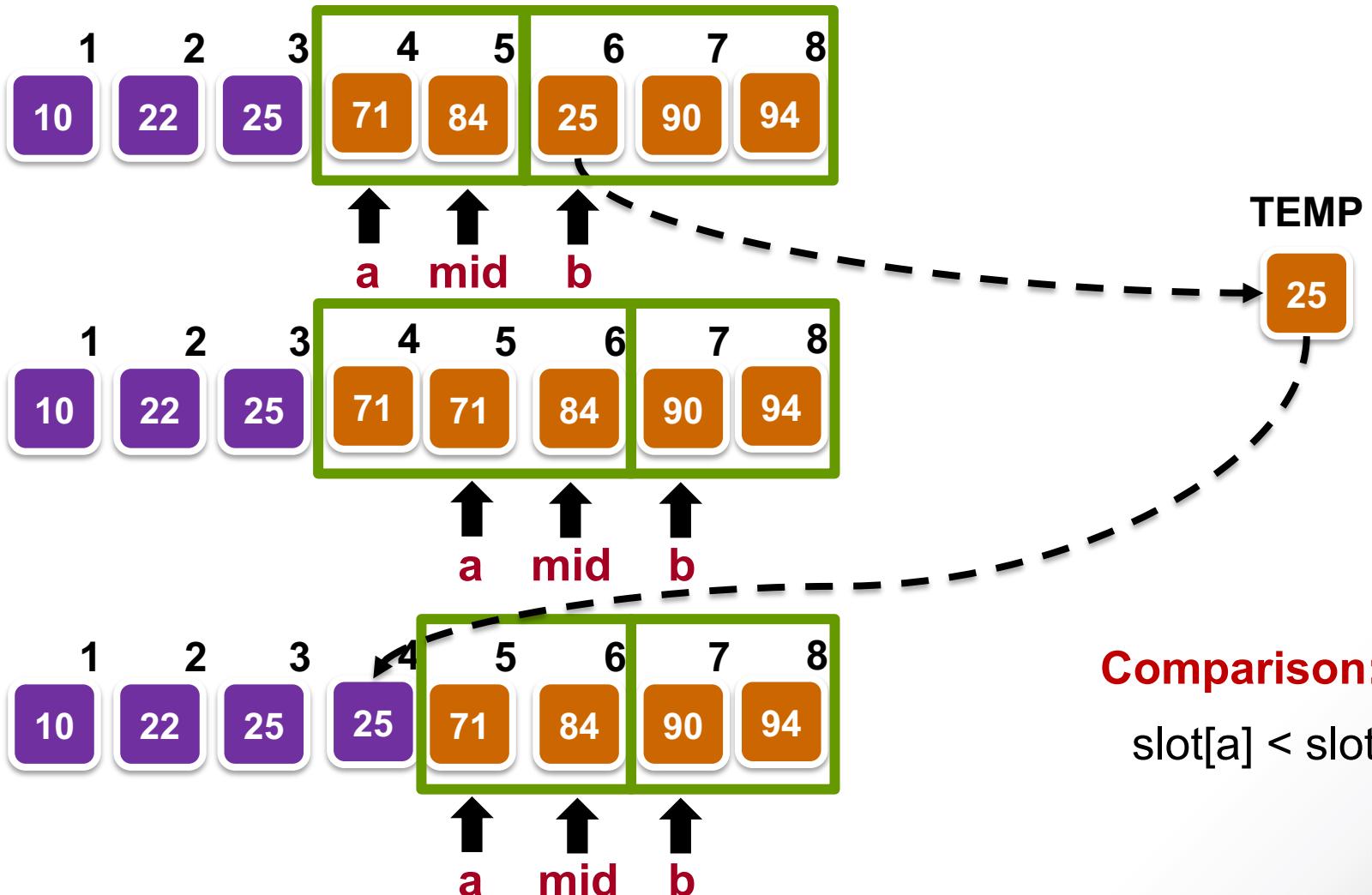
TEMP

22

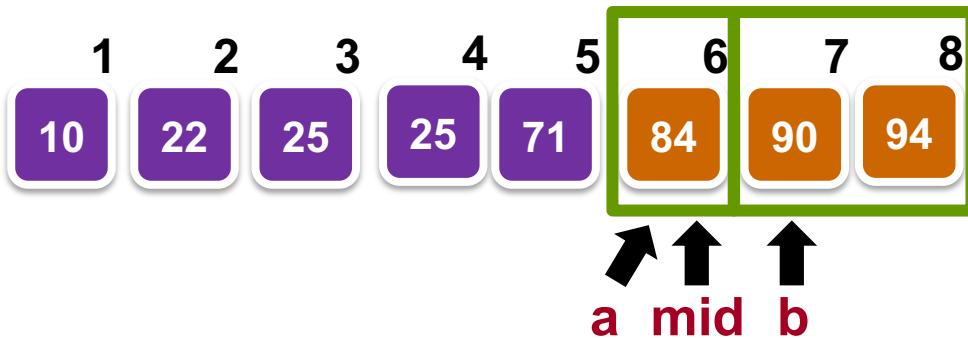


Comparison:
 $\text{slot}[a] == \text{slot}[b]$

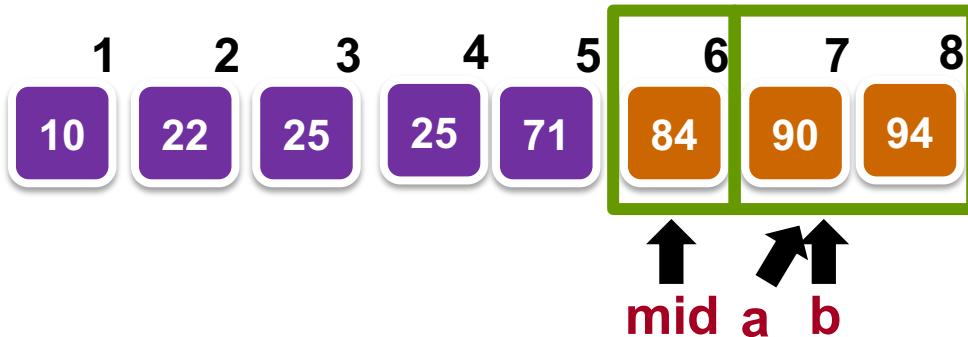
Merge Operation



Merge Operation



Comparison:
 $\text{slot}[a] < \text{slot}[b]$



Merge operation completed



SC2001/CE2101/CZ2101: Algorithm Design and Analysis

Quicksort

Instructor: Asst. Prof. LIN Shang-Wei

Courtesy of Dr. Ke Yiping, Kelly's slides

Learning Objectives

At the end of this lecture, students should be able to:

- Explain how “Divide and Conquer” approach is used in Quicksort
- Explain the pseudo code of Quicksort
- Manually execute Quicksort on an example input array
- Analyse time complexities of Quicksort in the best, average and worst cases

Quicksort

- Fastest general purpose in-memory sorting algorithm in the average case
- Implemented in Unix as `qsort()` which can be called in a program (see ‘man qsort’ for details)
- Main steps
 - Select one element in array as **pivot**
 - Partition list into two sublists with respect to pivot such that all elements in left sublist are less than pivot; all elements in right sublist are greater than or equal to pivot
 - Recursively partition until input list has one or zero element
- No merging is required because the pivot found during partitioning is already at its final position

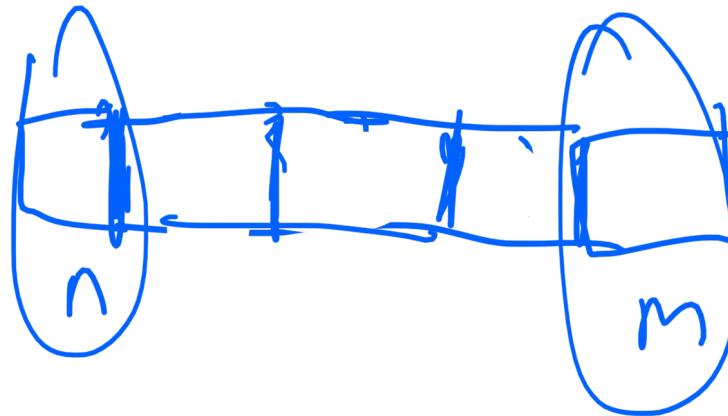




Quicksort (Pseudo Code)

Quicksort (Pseudo Code)

```
void quicksort(int n, int m)
{
    int pivot_pos;
    if (n >= m)
        return;
    pivot_pos = partition(n, m);
    quicksort(n, pivot_pos - 1);
    quicksort(pivot_pos + 1, m);
}
```



Quicksort (Pseudo Code)

```
void quicksort(int n, int m)
{
    int pivot_pos;
    if (n >= m)
        return;
    pivot_pos = partition(n, m);
    quicksort(n, pivot_pos - 1);
    quicksort(pivot_pos + 1, m);
}
```

Quicksort (Pseudo Code)

```
void quicksort(int n, int m)
{
    int pivot_pos;
    if (n >= m)
        return;
    pivot_pos = partition(n, m);
    quicksort(n, pivot_pos - 1);
    quicksort(pivot_pos + 1, m);
}
```

Quicksort (Pseudo Code)

```
void quicksort(int n, int m)
{
    int pivot_pos;
    if (n >= m)
        return;
    pivot_pos = partition(n, m);
    quicksort(n, pivot_pos - 1);
    quicksort(pivot_pos + 1, m);
}
```

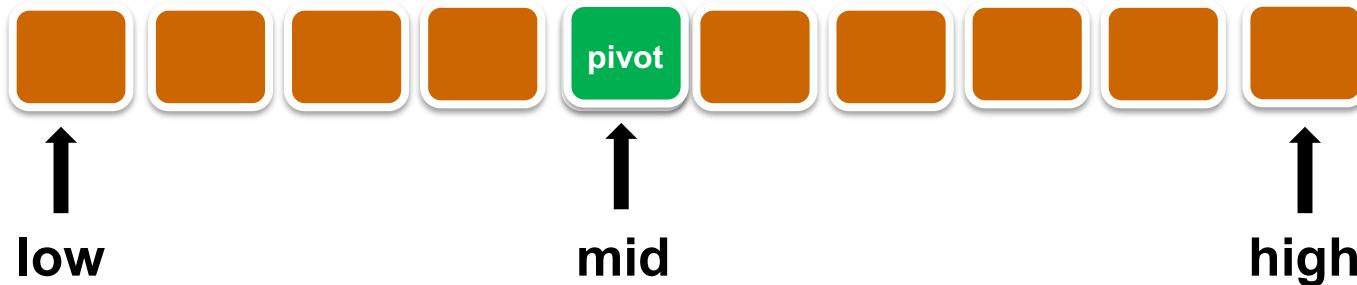
Quicksort (Pseudo Code)

```
void quicksort(int n, int m)
{
    int pivot_pos;
    if (n >= m)
        return;
    pivot_pos = partition(n, m);
    quicksort(n, pivot_pos - 1);
    quicksort(pivot_pos + 1, m);
}
```



Partition Routine in Quicksort

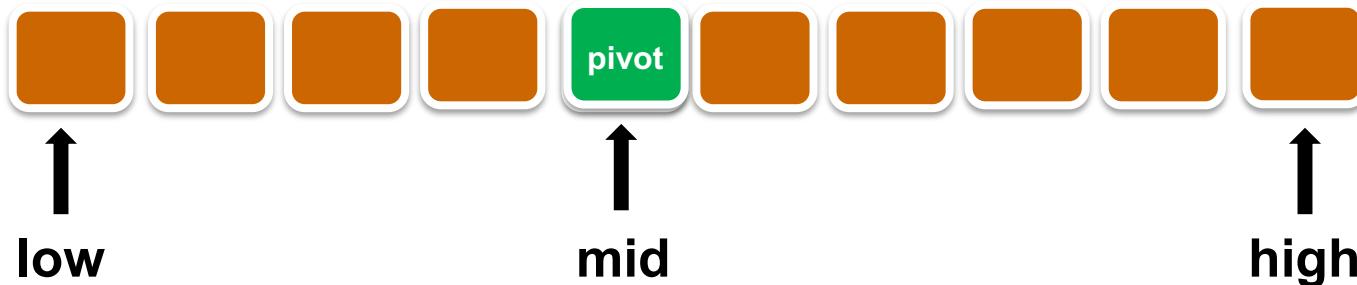
Partition Routine in Quicksort



int partition(int low, int high)

```
{  
    int i, last_small, pivot;  
    int mid = (low+high)/2;  
    swap(low, mid);  
    pivot = slot[low];  
    last_small = low;
```

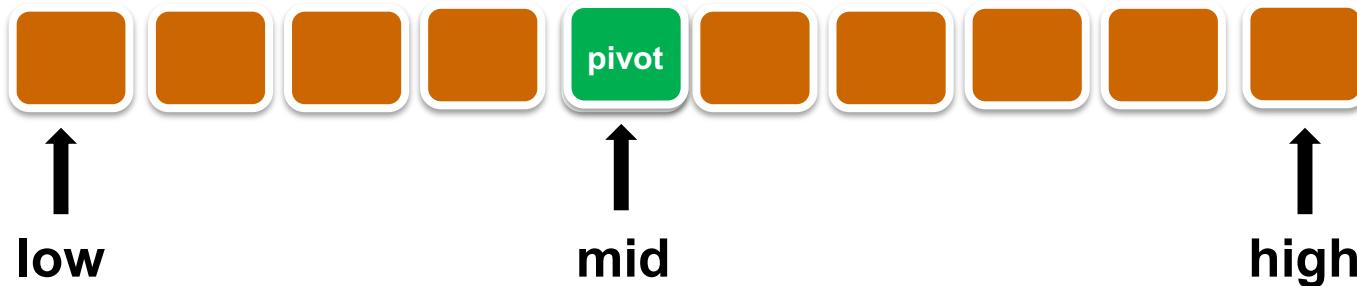
Partition Routine in Quicksort



int partition(int low, int high)

```
{  
    int i, last_small, pivot;  
    int mid = (low+high)/2;  
    swap(low, mid);  
    pivot = slot[low];  
    last_small = low;
```

Partition Routine in Quicksort

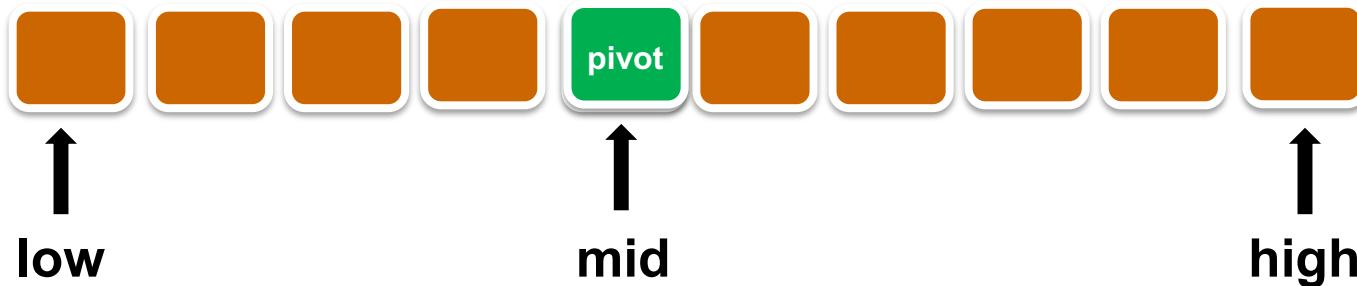


```
int partition(int low, int high)
```

```
{
```

```
    int i, last_small, pivot;  
    int mid = (low+high)/2;  
    swap(low, mid);  
    pivot = slot[low];  
    last_small = low;
```

Partition Routine in Quicksort



```
int partition(int low, int high)
```

```
{
```

```
    int i, last_small, pivot;
```

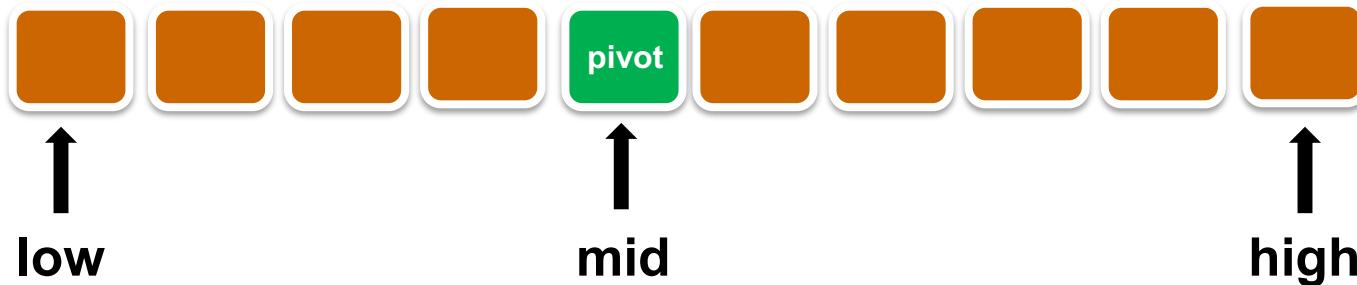
```
    int mid = (low+high)/2;
```

```
    swap(low, mid);
```

```
    pivot = slot[low];
```

```
    last_small = low;
```

Partition Routine in Quicksort



```
int partition(int low, int high)
```

```
{
```

```
    int i, last_small, pivot;
```

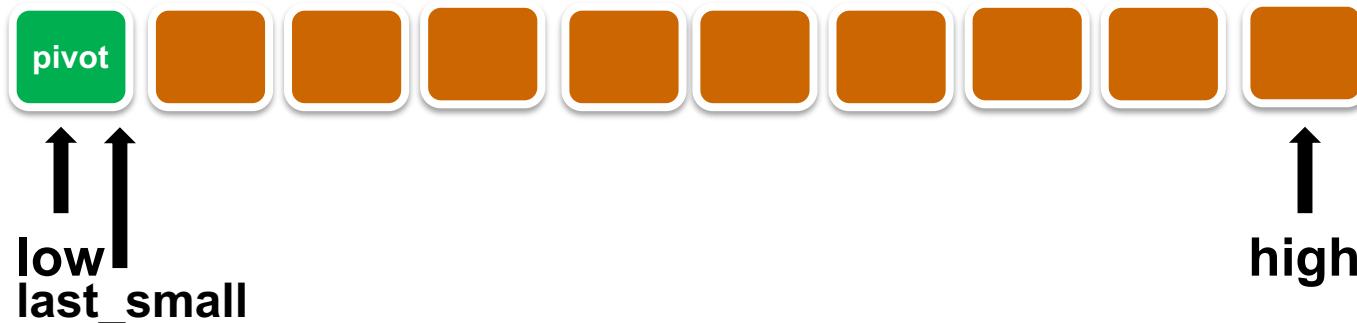
```
    int mid = (low+high)/2;
```

```
swap(low, mid);
```

```
    pivot = slot[low];
```

```
    last_small = low;
```

Partition Routine in Quicksort



```
int partition(int low, int high)
```

```
{
```

```
    int i, last_small, pivot;
```

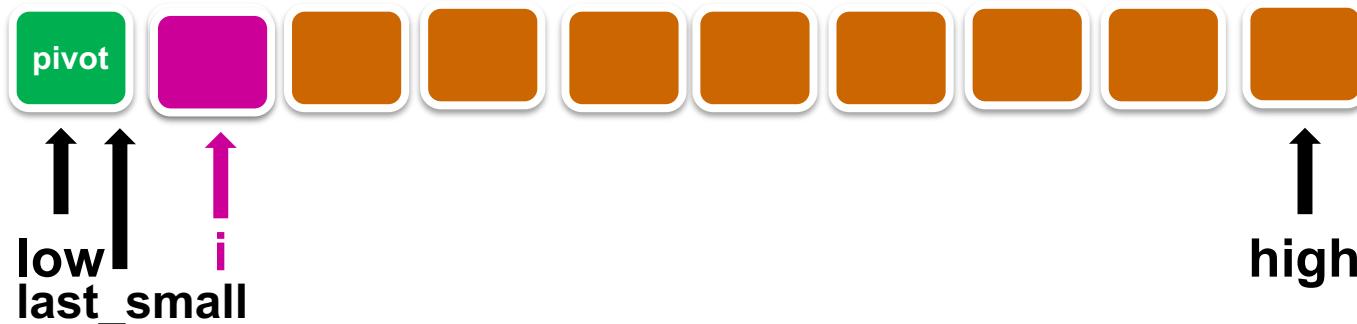
```
    int mid = (low+high)/2;
```

```
    swap(low, mid);
```

```
pivot = slot[low];
```

```
last_small = low;
```

Partition Routine in Quicksort



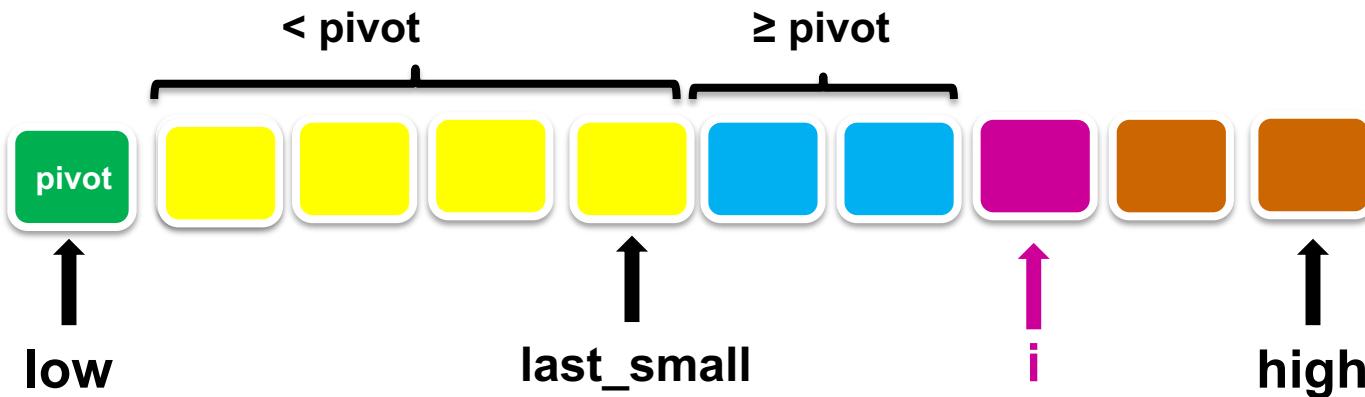
```
int partition(int low, int high)
```

```
{.....
```

```
    for (i = low+1; i <= high; i++)
        if (slot[i] < pivot)
            swap(++last_small, i);
    swap(low, last_small);
    return last_small;
```

```
}
```

Partition Routine in Quicksort



```
int partition(int low, int high)
```

```
{.....
```

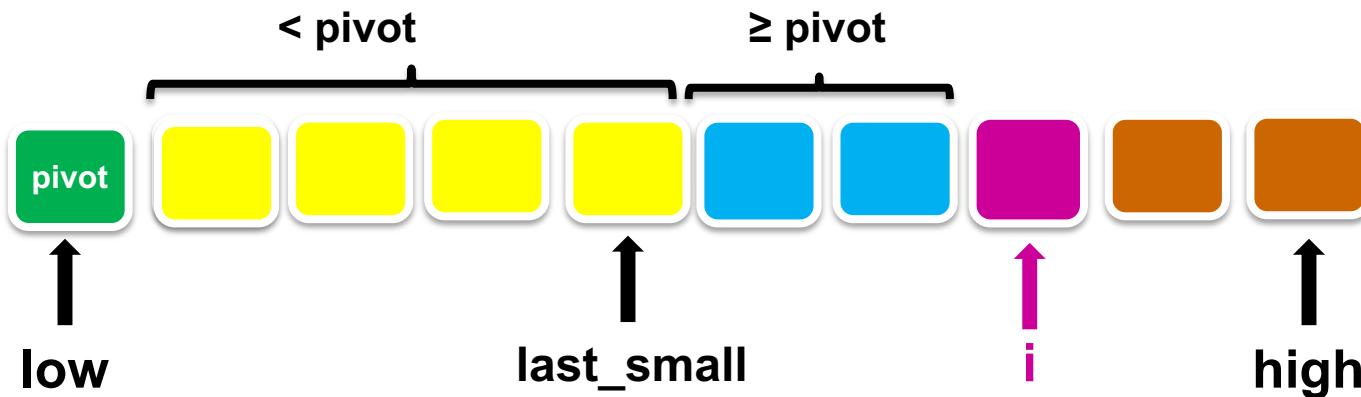
```
    for (i = low+1; i <= high; i++)
        if (slot[i] < pivot)
            swap(++last_small, i);
```

```
    swap(low, last_small);
```

```
    return last_small;
```

```
}
```

Partition Routine in Quicksort



int partition(int low, int high)

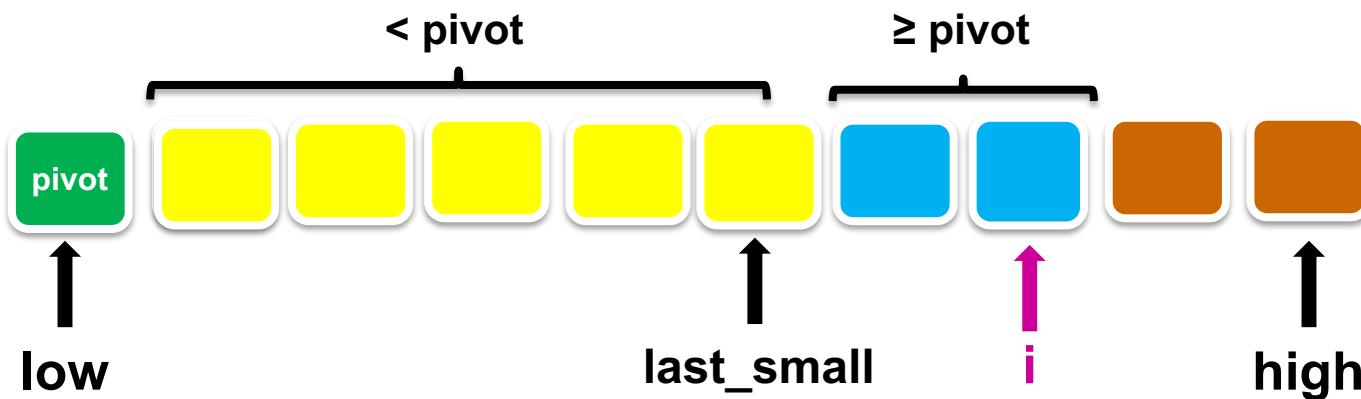
{.....

```
for (i = low+1; i <= high; i++)
    if (slot[i] < pivot) ✓
        swap(++last_small, i);
```

```
swap(low, last_small);
return last_small;
```

}

Partition Routine in Quicksort



```
int partition(int low, int high)
```

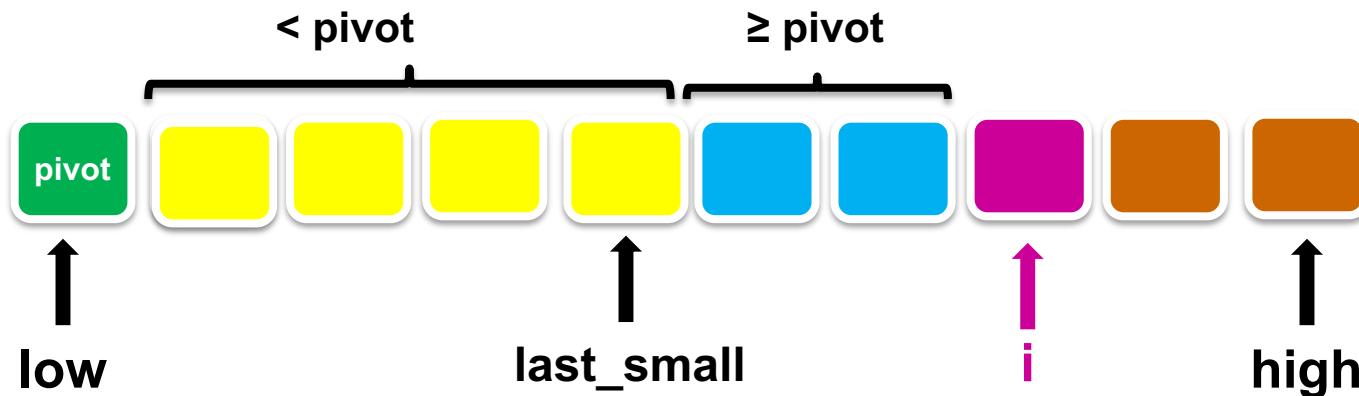
```
{.....
```

```
    for (i = low+1; i <= high; i++)
        if (slot[i] < pivot) ✓
            swap(++last_small, i);
```

```
    swap(low, last_small);
    return last_small;
```

```
}
```

Partition Routine in Quicksort

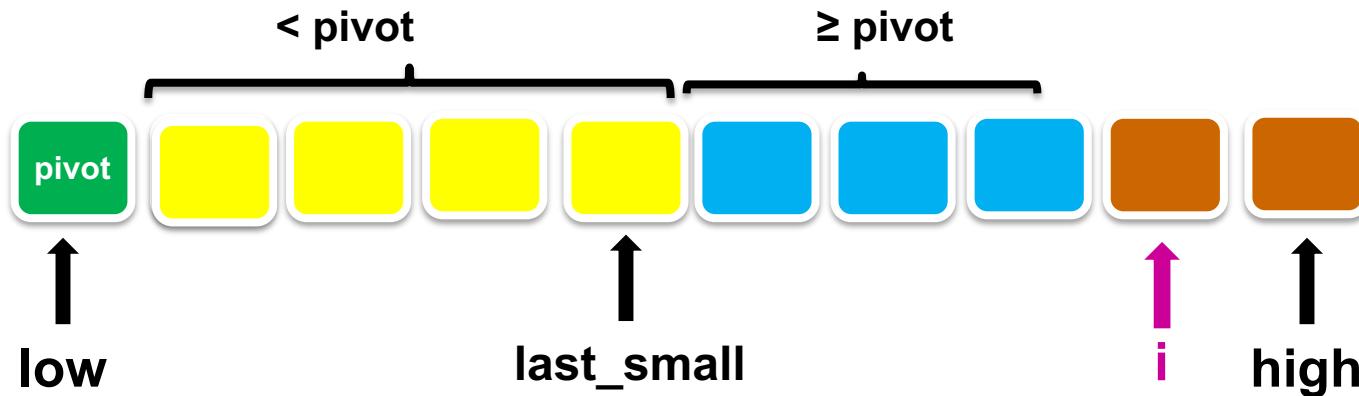


```
int partition(int low, int high)
```

```
{.....
```

```
    for (i = low+1; i <= high; i++)
        if (slot[i] < pivot) ✗
            swap(++last_small, i);
    swap(low, last_small);
    return last_small;
}
```

Partition Routine in Quicksort

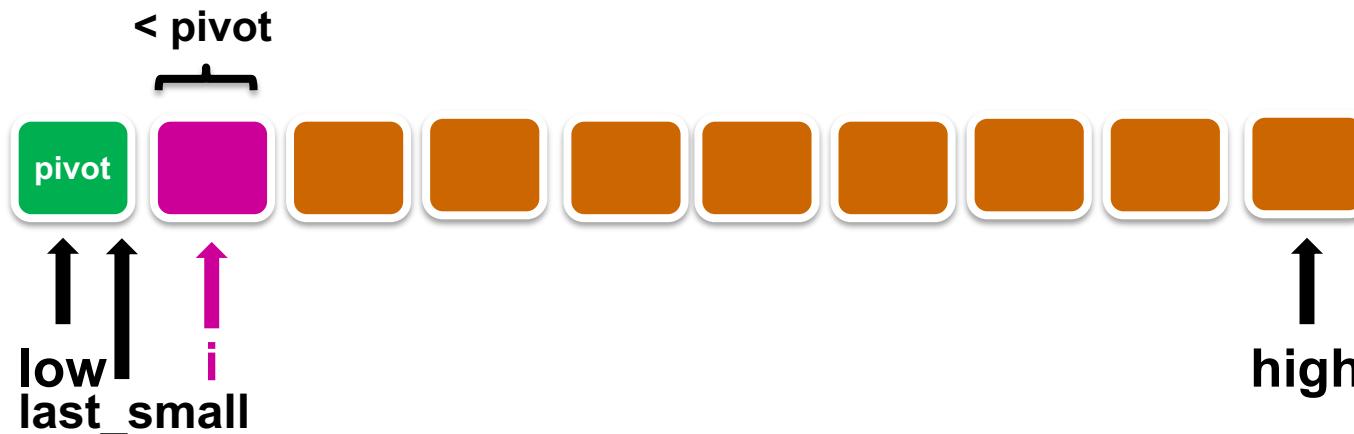


```
int partition(int low, int high)
```

```
{.....
```

```
    for (i = low+1; i <= high; i++)
        if (slot[i] < pivot) ✗
            swap(++last_small, i);
    swap(low, last_small);
    return last_small;
}
```

Partition Routine in Quicksort



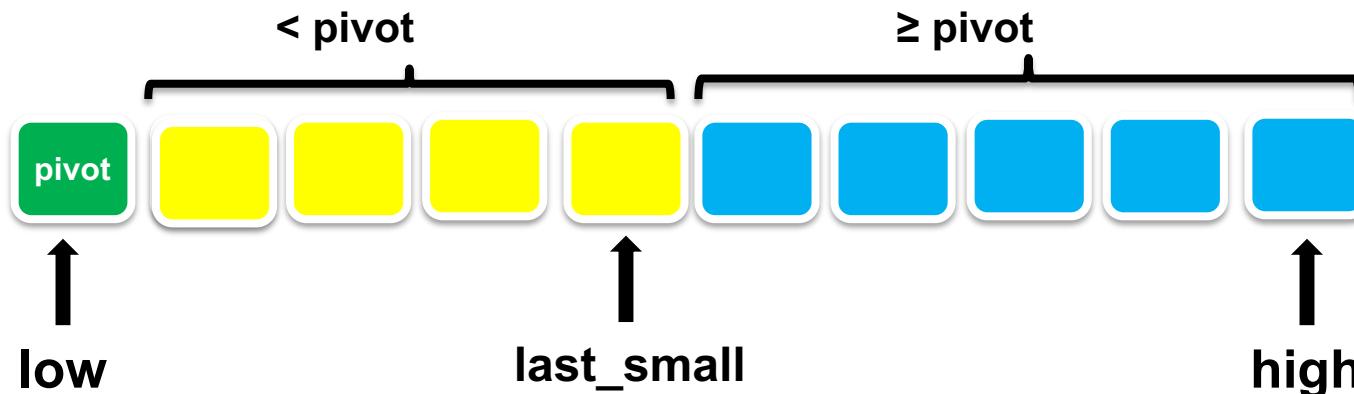
int partition(int low, int high)

{.....

```
    for (i = low+1; i <= high; i++)
        if (slot[i] < pivot) ✓
            swap(++last_small, i);
    swap(low, last_small);
    return last_small;
```

}

Partition Routine in Quicksort



```
int partition(int low, int high)
```

```
{.....
```

```
    for (i = low+1; i <= high; i++)
        if (slot[i] < pivot)
            swap(++last_small, i);
    swap(low, last_small);
    return last_small;
```

```
}
```

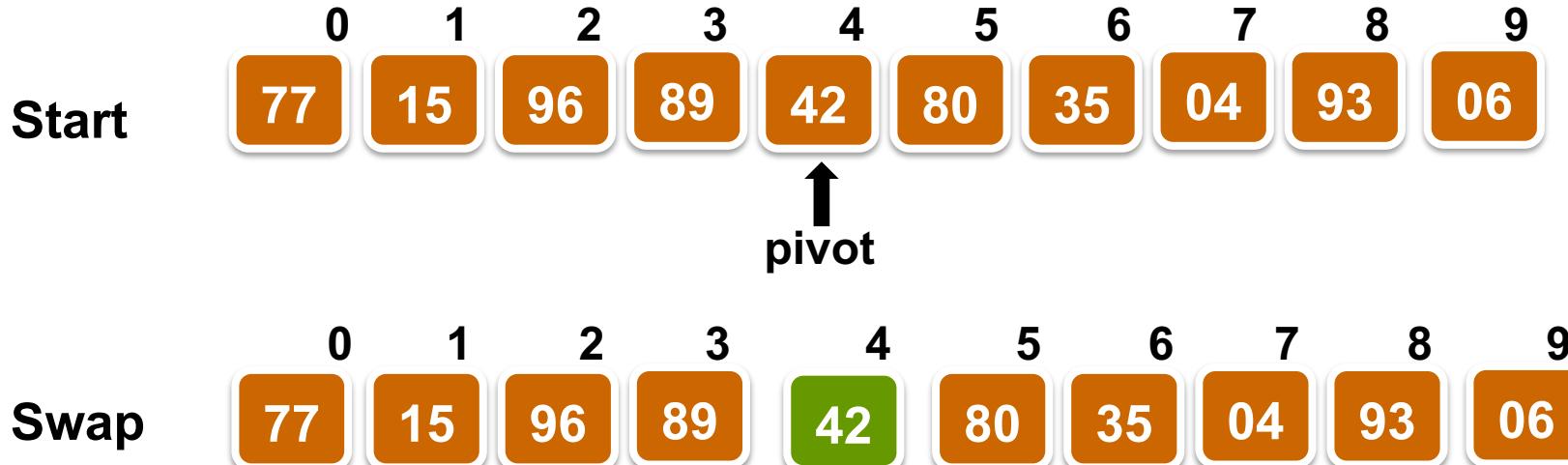
Note:

- ☞ Loop terminates when *i* reaches **high**;
- ☞ swap **pivot** from position **low** to position **last_small**, to obtain the final position of pivot element.



Quicksort (Example)

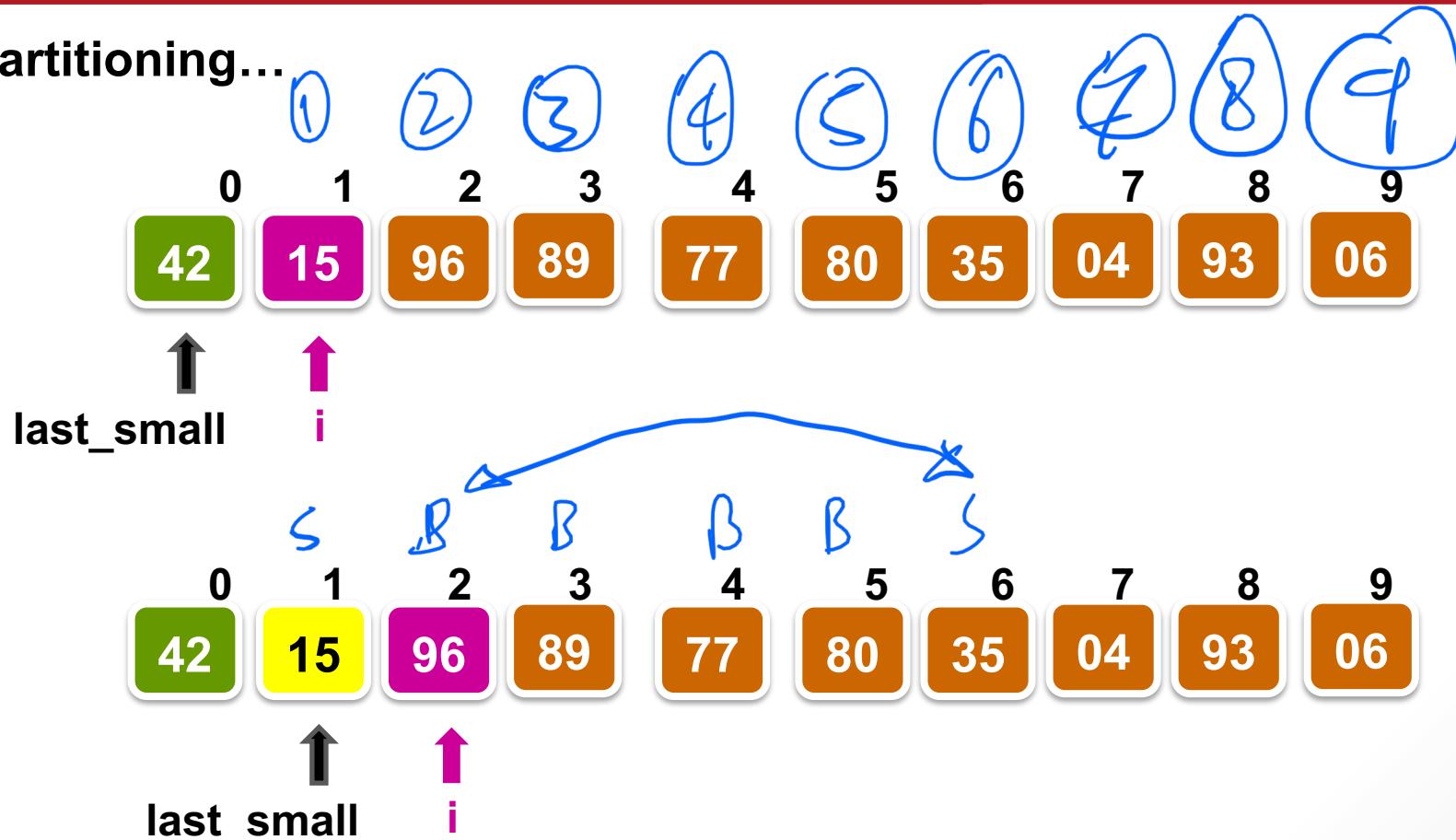
Quicksort (Example)



Partition the elements ...

Quicksort (Example)

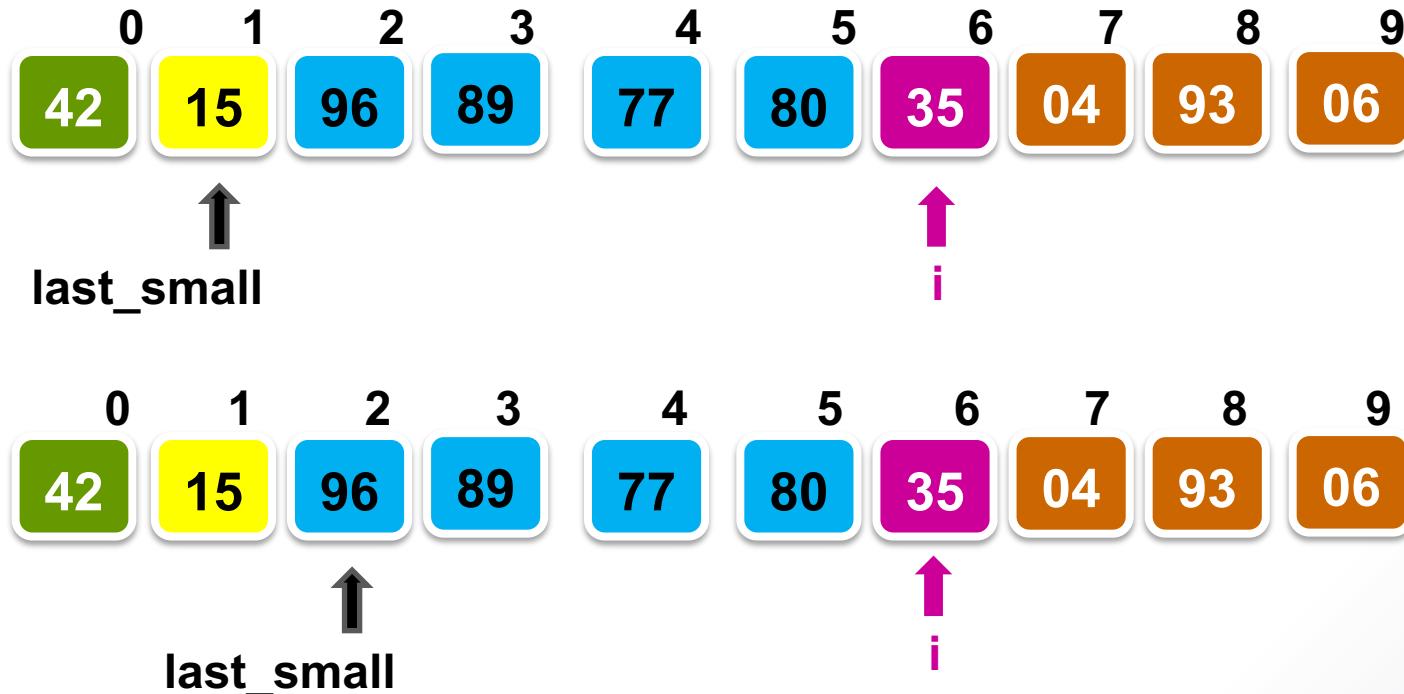
Partitioning...



Quicksort (Example)

Partitioning...

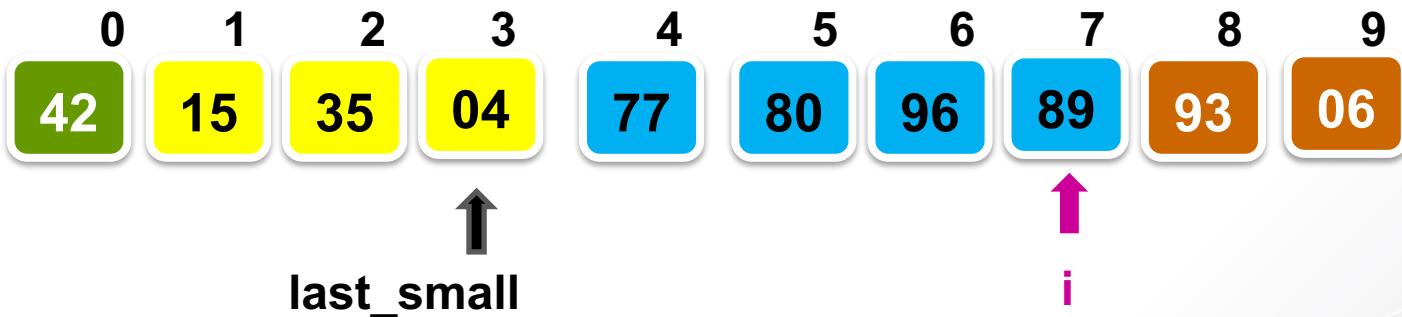
Carry on checking if (item \geq pivot) ...



Quicksort (Example)

Partitioning...

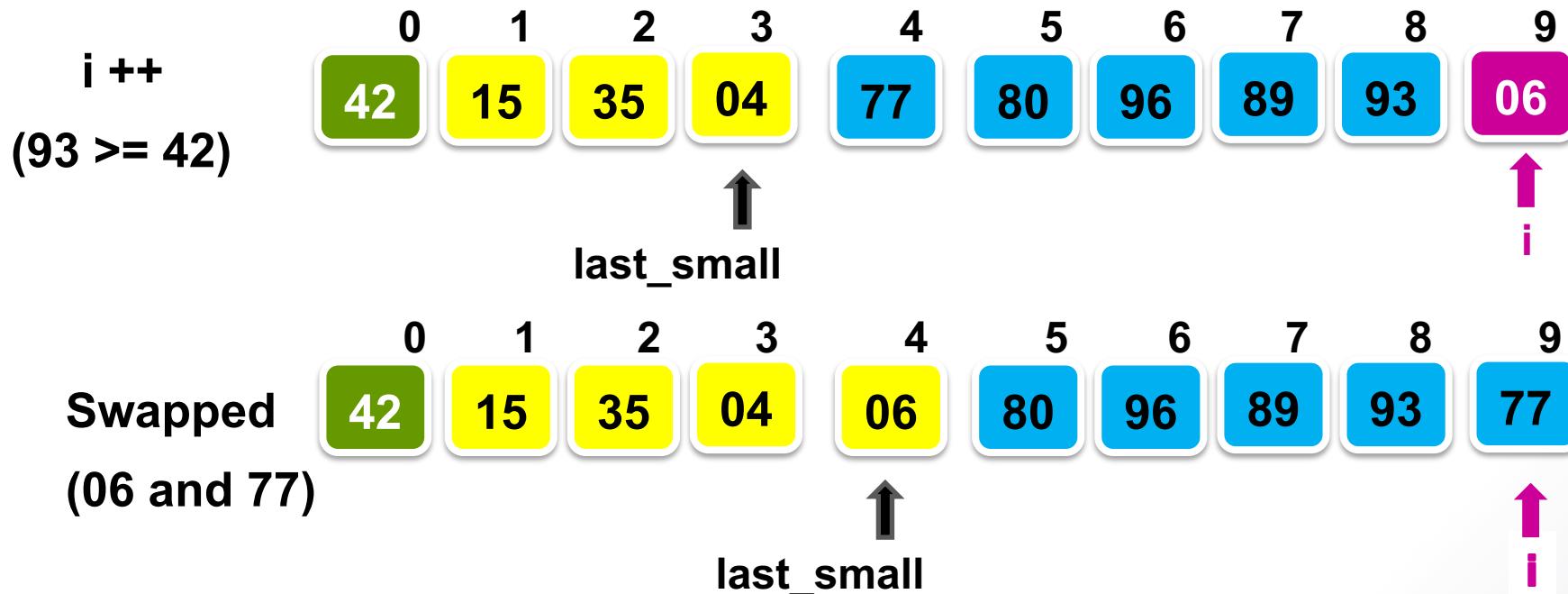
Carry on checking if (item \geq pivot) ...



Quicksort (Example)

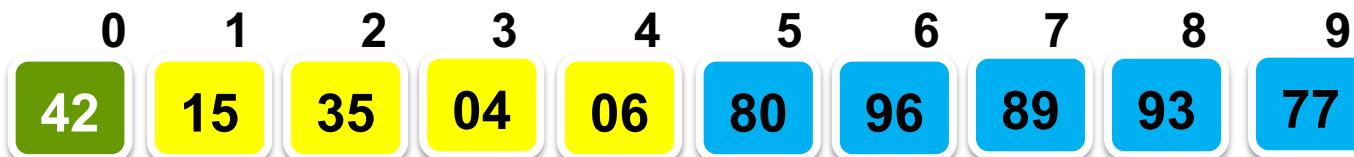
Partitioning...

Carry on checking if (item \geq pivot) ...



Quicksort (Example)

Finally, swap “last_small” (i.e. the final position where the pivot should be) with pivot.



We have done **9** comparisons in the partition.

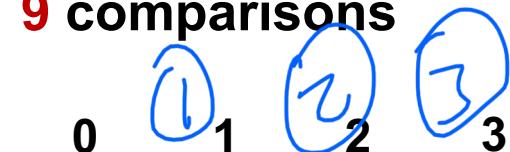
Quicksort (Example)



After partitioning...



9 comparisons



Step 2:

Swap



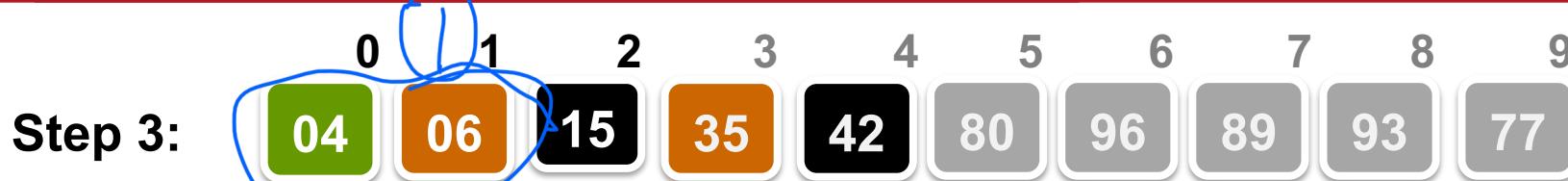
Recursively call
Quicksort (low,pivot_pos-1);
Ignore RHS for time being

Insert

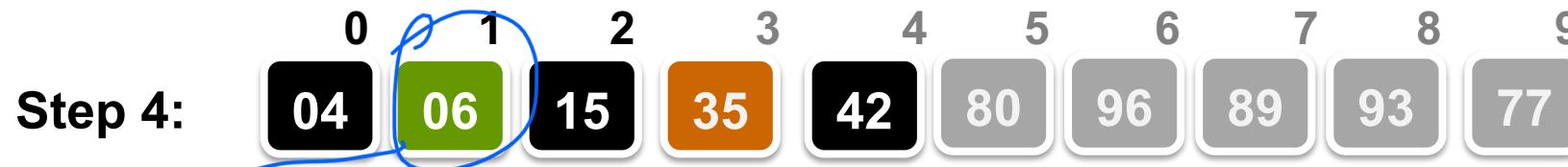


3 comparisons

Quicksort (Example)



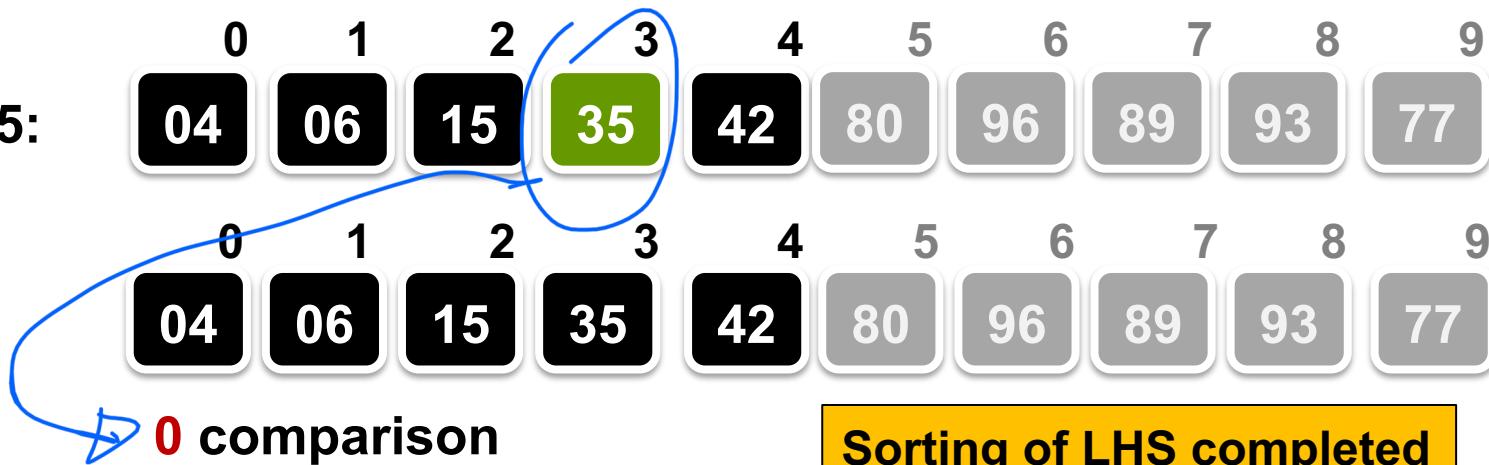
1 comparison



0 comparison

Quicksort (Example)

Step 5:



Quicksort (Example)

Dealing with right half of the array:

① ② ③ ④

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|----|----|----|----|----|----|----|----|----|----|
| 04 | 06 | 15 | 35 | 42 | 89 | 96 | 80 | 93 | 77 |

Step 6:

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|----|----|----|----|----|----|----|----|----|----|
| 04 | 06 | 15 | 35 | 42 | 77 | 80 | 89 | 93 | 96 |

4 comparisons

⑤

Step 7:

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|----|----|----|----|----|----|----|----|----|----|
| 04 | 06 | 15 | 35 | 42 | 77 | 80 | 89 | 93 | 96 |

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|----|----|----|----|----|----|----|----|----|----|
| 04 | 06 | 15 | 35 | 42 | 77 | 80 | 89 | 93 | 96 |

1 comparison

Quicksort (Example)

Dealing with right half of the array:



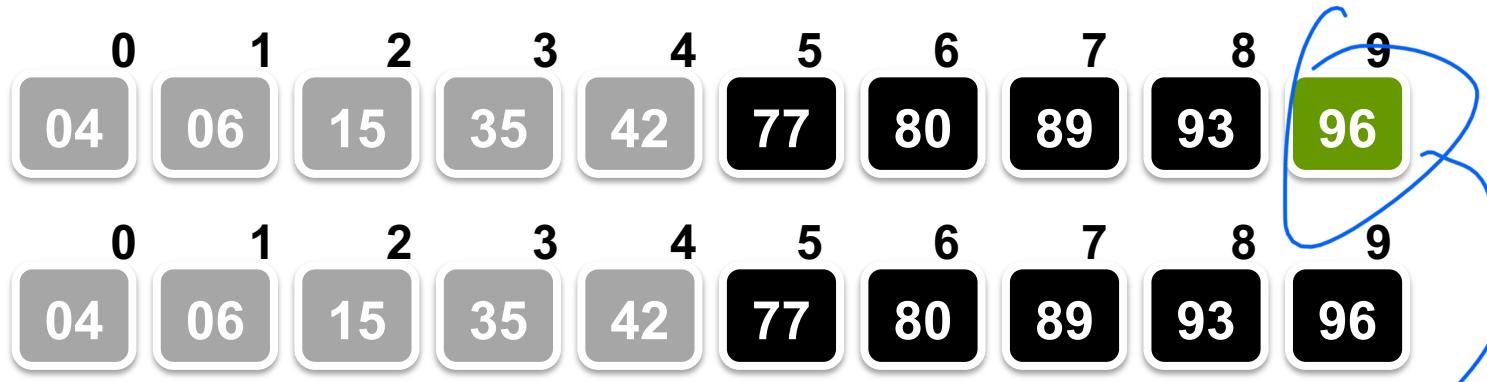
0 comparison



1 comparison

Quicksort (Example)

Step 10:

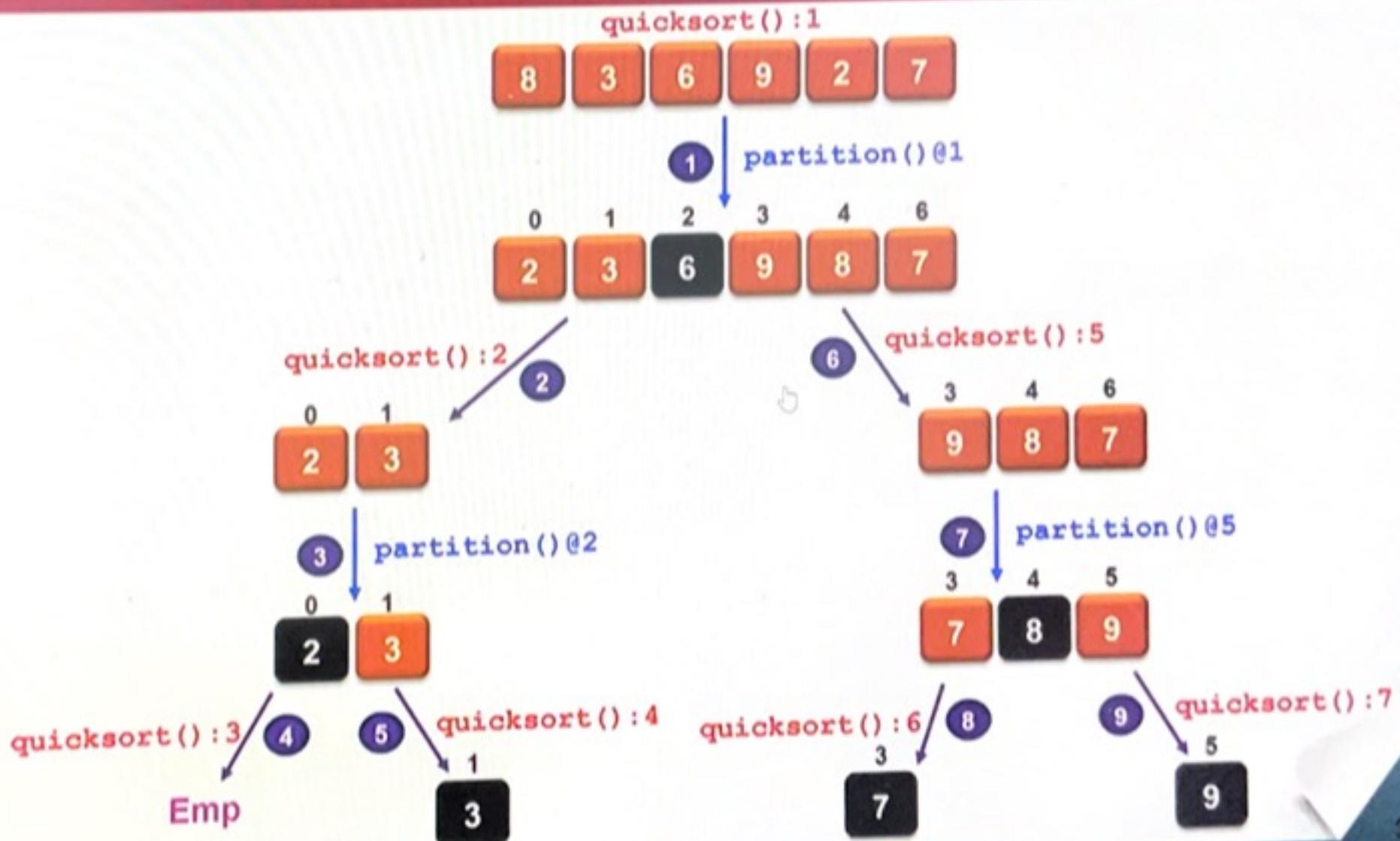


0 comparison

Final outcome:



Execution Order of Quicksort



Comments on Quicksort

- **Which element of array should be pivot?** In this implementation, we take the middle element as pivot (other choices possible).
- Use `quicksort(0, size - 1)` to invoke quick sort; ‘size’ is the number of elements in array `slot[]`.
- During partitioning, the middle element (pivot) is moved to the 1st position (i.e. `slot[0]`).
- A ‘for’ loop goes through the rest of array to split it into two portions.

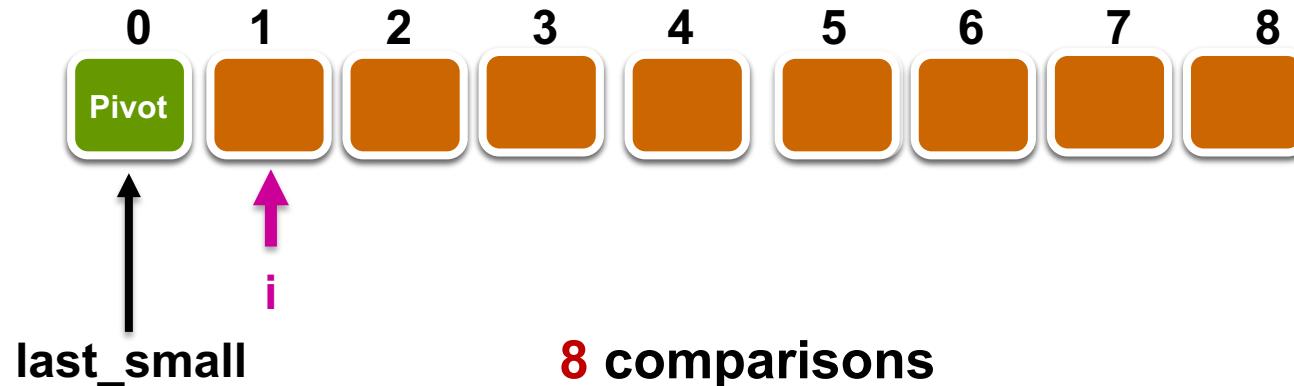


pivot is bottleneck

Quicksort's Performance

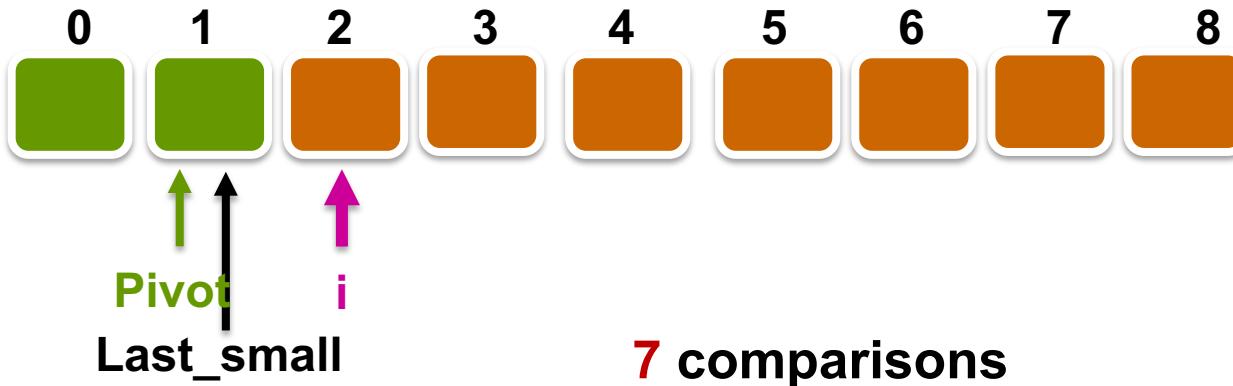
Quicksort's Performance

Worst-case



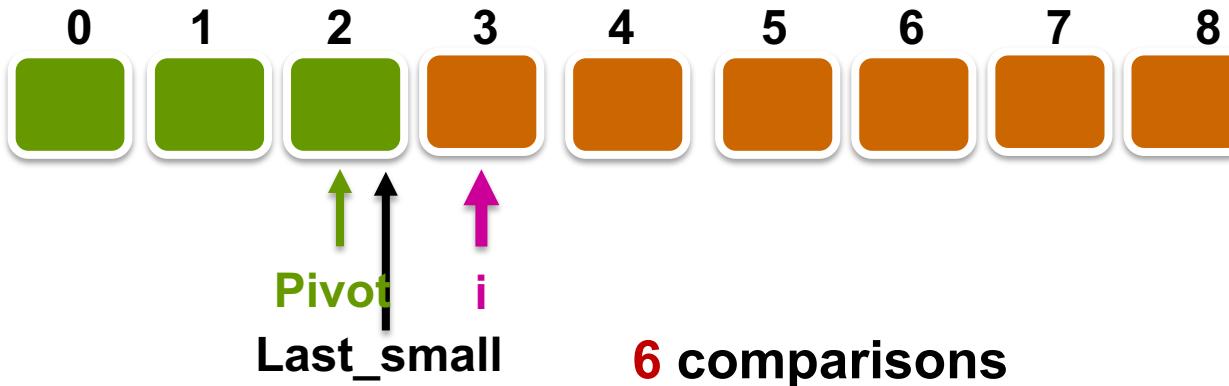
Quicksort's Performance

Worst-case



Quicksort's Performance

Worst-case



Quicksort's Performance

Worst case happens when the pivot does a bad job at splitting the array **evenly**, if pivot is the smallest or the largest key each time, then the total no. of key comparisons is $O(n^2)$.

$$\sum_{k=2}^n (k-1) = \sum_{k=1}^{n-1} k = \frac{n(n-1)}{2} \approx n^2$$

Quicksort's Performance

Best case happens when the pivot happens to divide the array into two sub-arrays of **equal length**, in **every partitioning**.

For simplicity, let's assume:

- $n = 2^k$, i.e. $k = \lg n$.
- Each step, the pivot divides the array of length n into two sub-arrays each of length approximately $n/2$.

$$\begin{aligned} \sum_{i=0}^{n-1} a_i &= a^0 + a^1 + \dots + a^{k-1} \\ \text{Sum of even num} &= n(n+1) \\ \text{Sum of odd num} &= n^2 \\ \sum_{i=0}^{n-1} i &= \frac{n(n-1)}{2} \\ \sum_{i=0}^{n-1} i &= n(n+1)/2 \end{aligned}$$

check $\frac{a^k - 1}{a - 1}$

Quicksort's Performance

The recurrence equation is:

$$T(1) = 0,$$

recursion

$$T(n) = 2T(n/2) + cn,$$

per comparison $\approx n - 1$

$$T(n) = 2(2T(n/4) + cn/2) + cn$$

$$= 2^2 T(n/4) + 2cn$$

$$= 2^3 T(n/8) + 3cn$$

...

$$= \underline{2^k} T(\underline{n/2^k}) + kcn$$

$$= nT(1) + cn\lg n = cn\lg n$$

$$\therefore T(n) = \Theta(n \lg n)$$

Because $n = 2^k$, i.e. $k = \lg n$,
and $T(1) = 0$

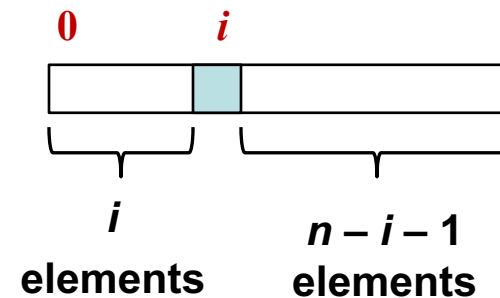
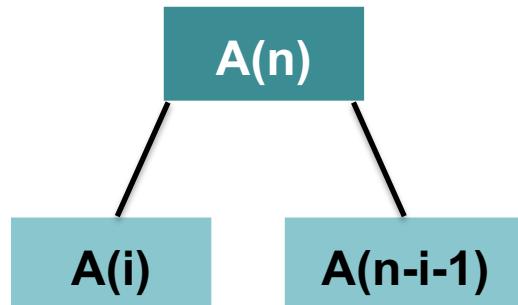
Quicksort's Performance

Average case: assume that the keys are distinct and that all permutations of the keys are equally likely.

k = no. of elements in the range of the array being sorted,

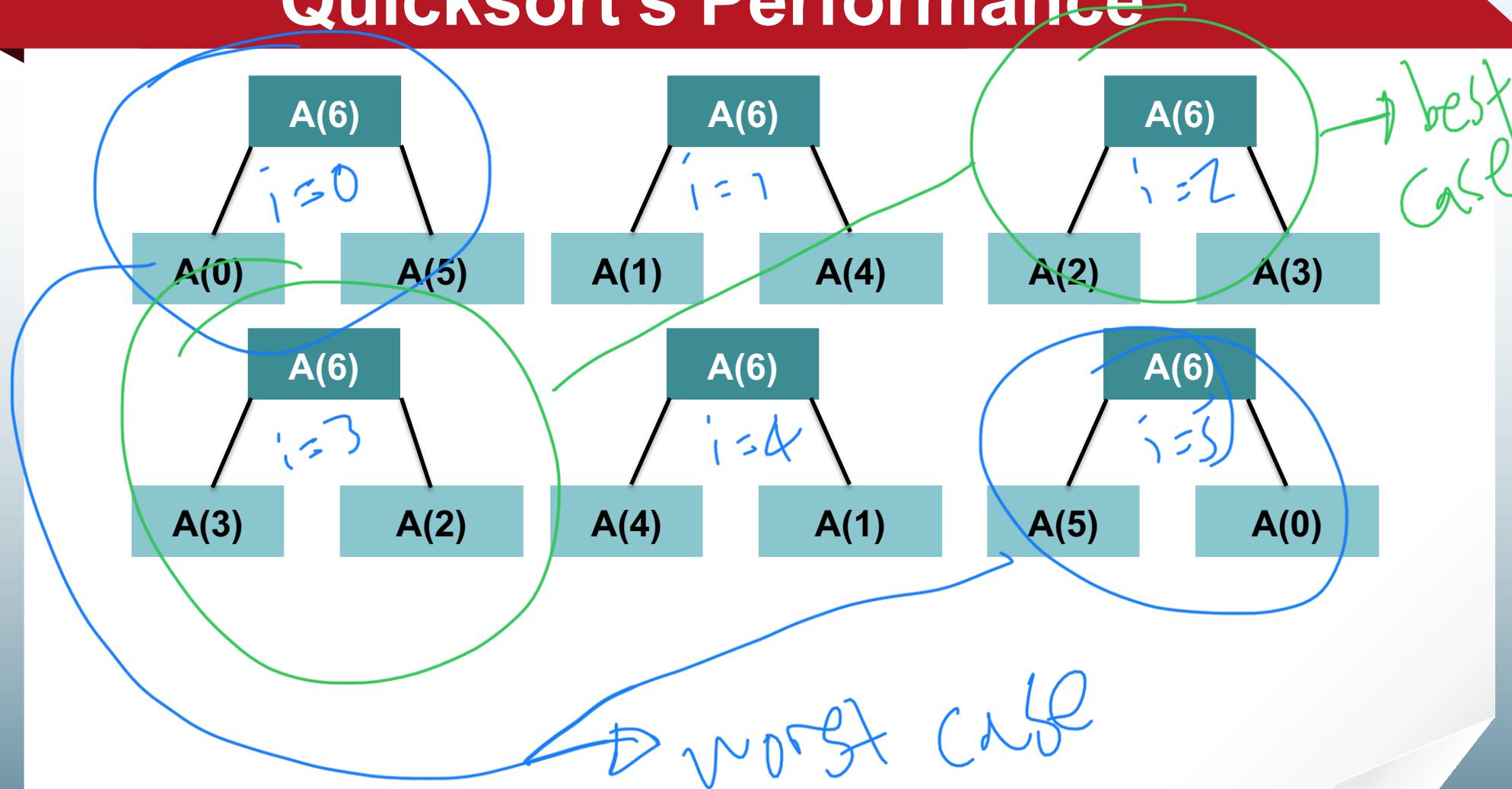
$A(k)$ = no. of comparisons done for this range,

i = final position of the pivot, counting from 0,



i = PIVOT

Quicksort's Performance



Quicksort's Performance

Thus,

$$A(6) = 5 + \frac{1}{6}(A(0) + A(5) + A(1) + A(4) + A(2) + A(3) + \dots + A(5) + A(0))$$

$$A(0) = A(1) = 0$$

$$A(n) = n - 1 + \frac{1}{n} \sum_{i=0}^{n-1} [A(i) + A(n-i-1)] = \Theta(n \lg n)$$

Proof is not required

Quicksort's Performance

😊 Strengths:

- 👉 Fast on average
- 👉 No merging required
- 👉 Best case occurs when pivot always splits array into equal halves

😢 Weaknesses:

- 👉 Poor performance when pivot does not split the array evenly
- 👉 Quicksort also performs badly when the size of list to be sorted is small
- 👉 If more work is done to select pivot carefully, the bad effects can be reduced

Summary

- Quicksort uses the “Divide and Conquer” approach.
- Partition function splits an input list into two sub-lists by comparing all elements with the pivot:
 - Elements in the left sub-list are $<$ pivot and
 - Elements in the right sub-list are \geq pivot.
- Quicksort is called recursively on each sub-list.
- The worst-case time complexity of Quicksort is $\Theta(n^2)$.
- The best-case and average-case time complexities of Quicksort are both $\Theta(n \lg n)$.

Quicksort (A as array, low as int, high as int)

if (low < high)

 pivot_location = **Partition** (A, low, high)

Quicksort (A, low, pivot_location)

Quicksort (A, pivot_location + 1, high)

Partition (A as array, low as int, high as int)

 pivot = A[low]

 leftwall = low

 for i = low + 1 to high

 if (A[i] < pivot) then

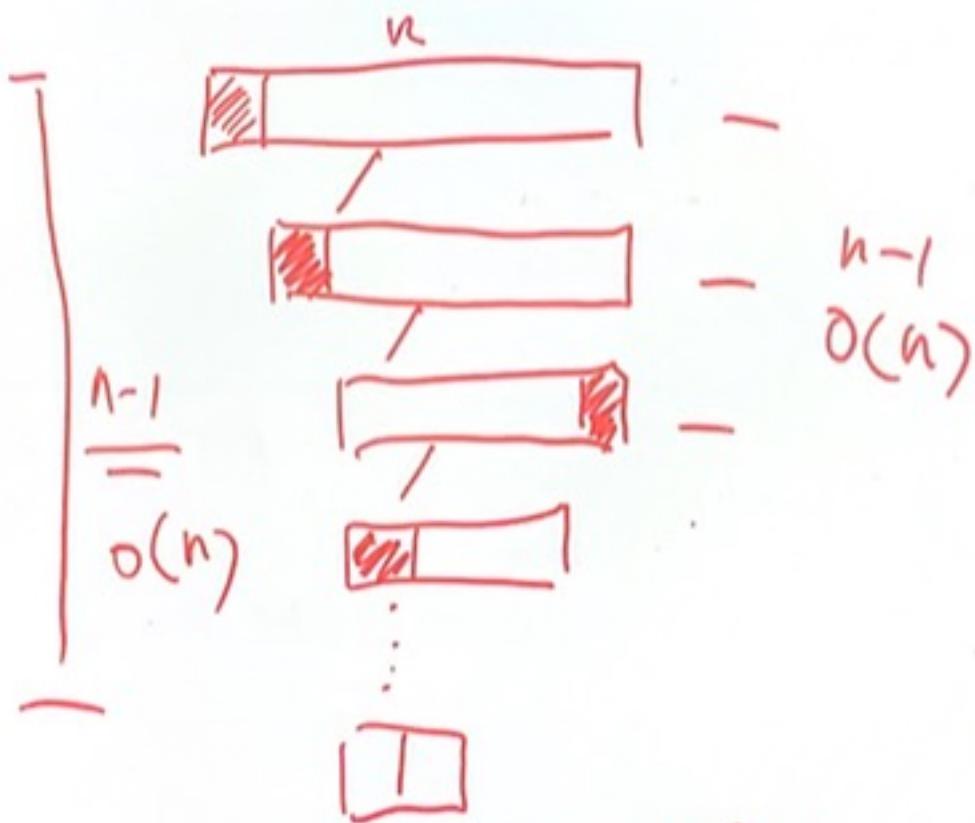
swap (A[i], A[leftwall])

 leftwall = leftwall + 1

swap (pivot, A[leftwall])

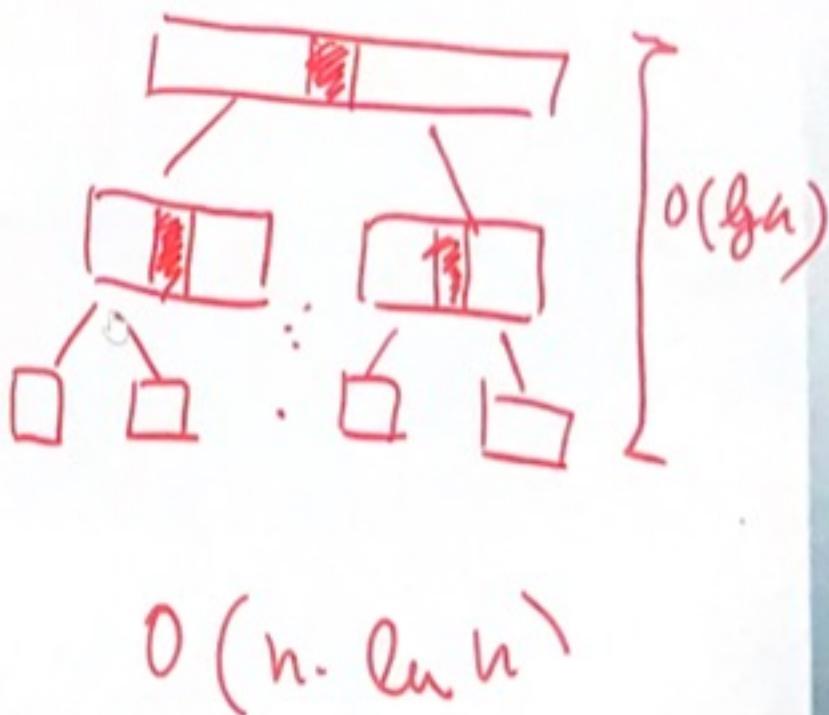
 return (leftwall)

Worst Case



$$O(n) \cdot O(n) = O(n^2)$$

Best Case





SC2001/CE2101/CZ2101: **Algorithm Design and Analysis**

Heapsort

Instructor: Asst. Prof. LIN Shang-Wei

Courtesy of Dr. Ke Yiping, Kelly's slides

Learning Objectives

At the end of this lecture, students should be able to:

- Explain the definition and properties of a heap
- Describe how Heapsort works
- Explain how to construct a heap from an input array
- Analyse the time complexity of Heapsort



Introduction to Heapsort

Heapsort

Heapsort is based on a heap data structure.

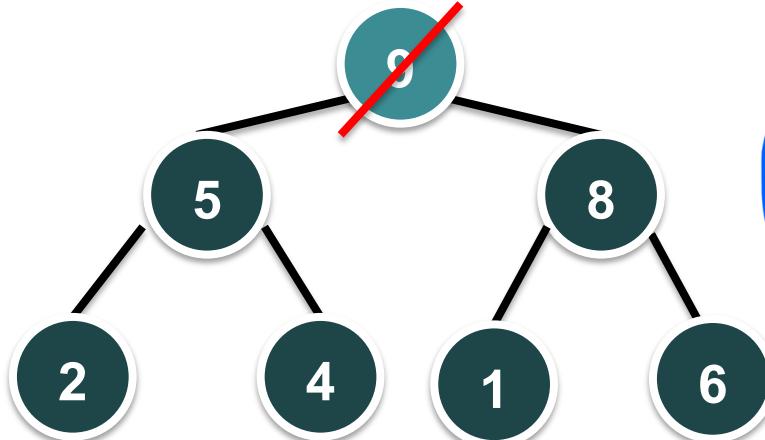
- **The definition of a heap includes:**

- a description of the structure.
- a condition on the data in the nodes (of a binary tree) called **partial order tree property**.

! Partial order tree property

A tree T is a **(maximising)** partial order tree if and only if each node has a key value **greater than or equal to** each of its child nodes (if it has any).

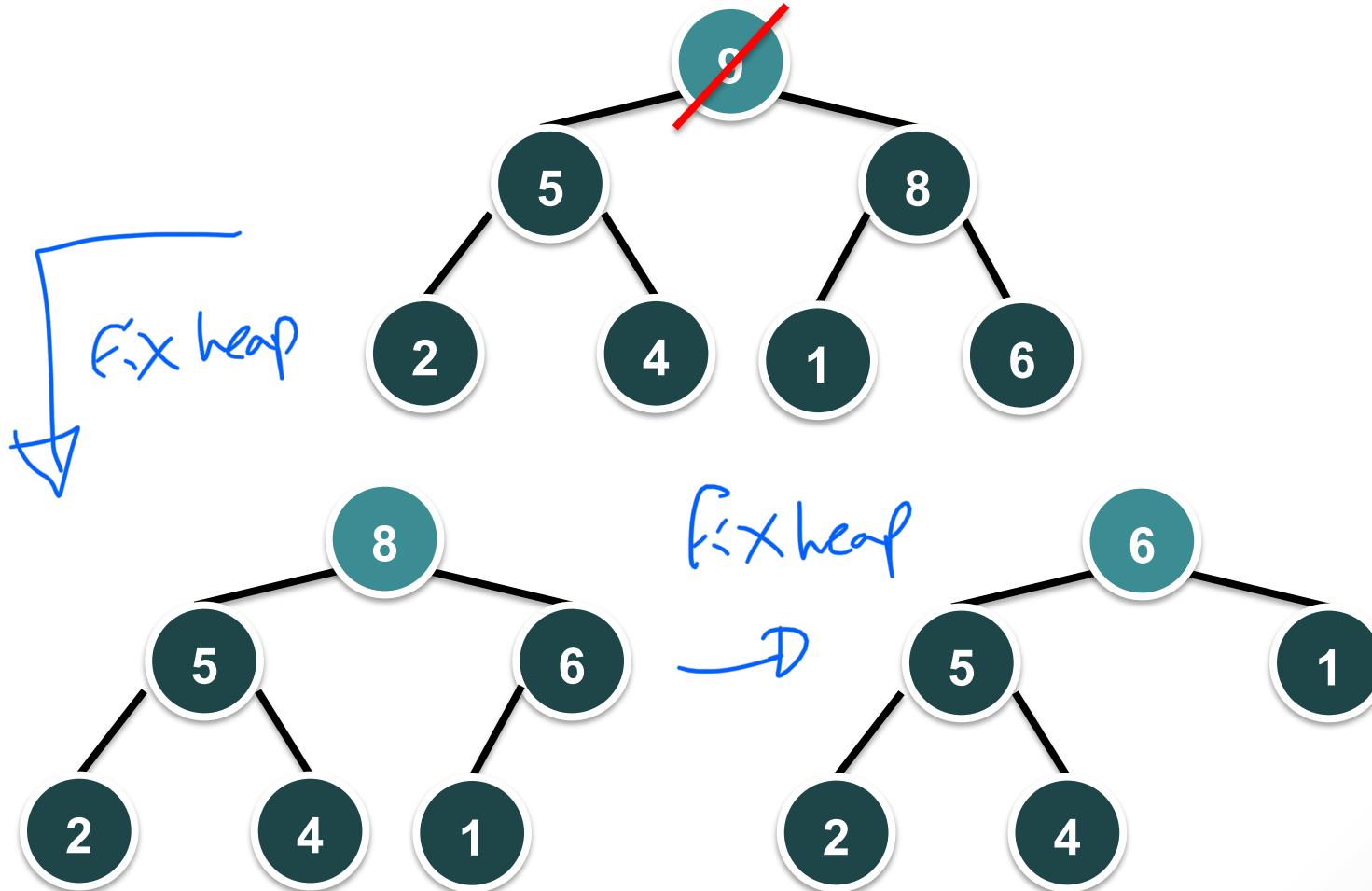
Partial Order Tree Property



↗ in heap
Parent \triangleright left + right
but left, n right
doesn't hv to

P \geq C maximizing Partial order tree
P \leq C minimizing

Partial Order Tree Property



Heapsort

algo

Heapsort is based on a heap data structure.

- **The definition of a heap includes:**

- a description of the structure.
 - a condition on the data in the nodes (of a binary tree) called **partial order tree property**.

- **Partial order tree property**

A tree T is a (**maximising**) partial order tree if and only if each node has a key value **greater than or equal to** each of its child nodes (if it has any).

- For a **minimising** partial order tree, the key value of every parent node is **less than or equal to** the value of each of its child nodes.

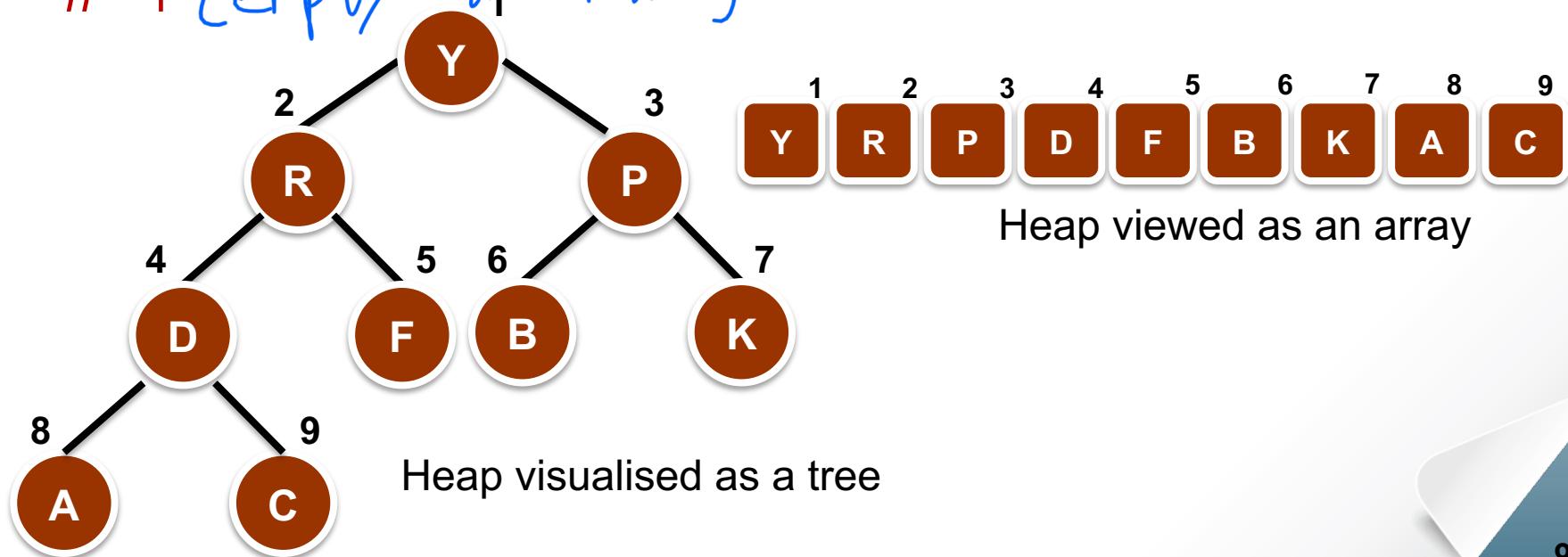


Heap Structure

Heap Structure

A binary tree T with height h is a heap structure if and only if it satisfies the following conditions:

- T is complete at least through depth $h - 1$
- all leaves are at depth h or $h - 1$
- all paths to a leaf of depth h are to the left of all paths to a leaf of depth $h - 1$ (empty left of $h-1$)

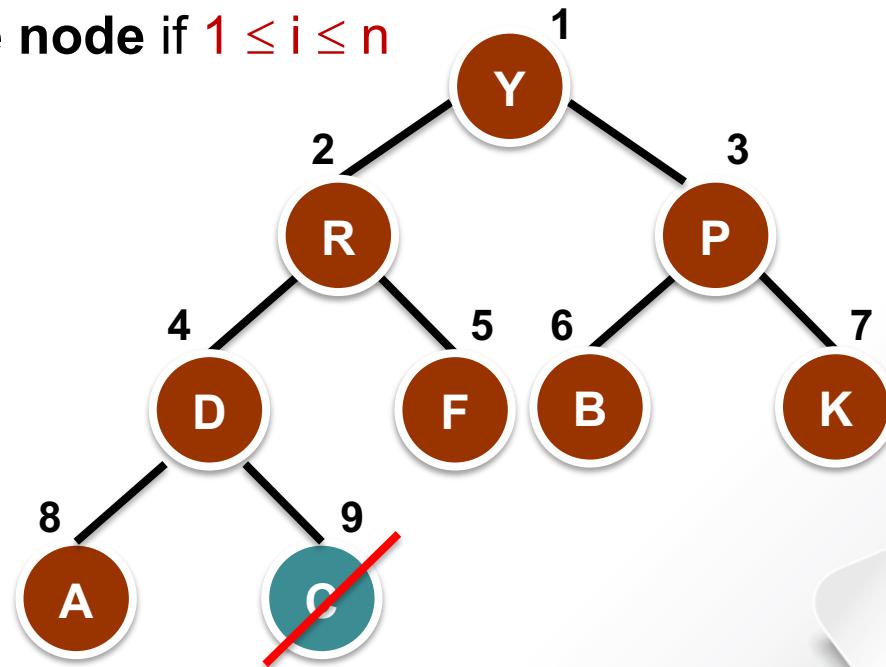


Heap Structure

- This means that every successive level of the tree must fill up from left to right. Further, an entire level must be full before any nodes at that level can have children nodes.
- Implementing the tree with n nodes by an array:**
 - Entry i in the array is a **tree node** if $1 \leq i \leq n$

$n = \cancel{9}$

$n = 8$

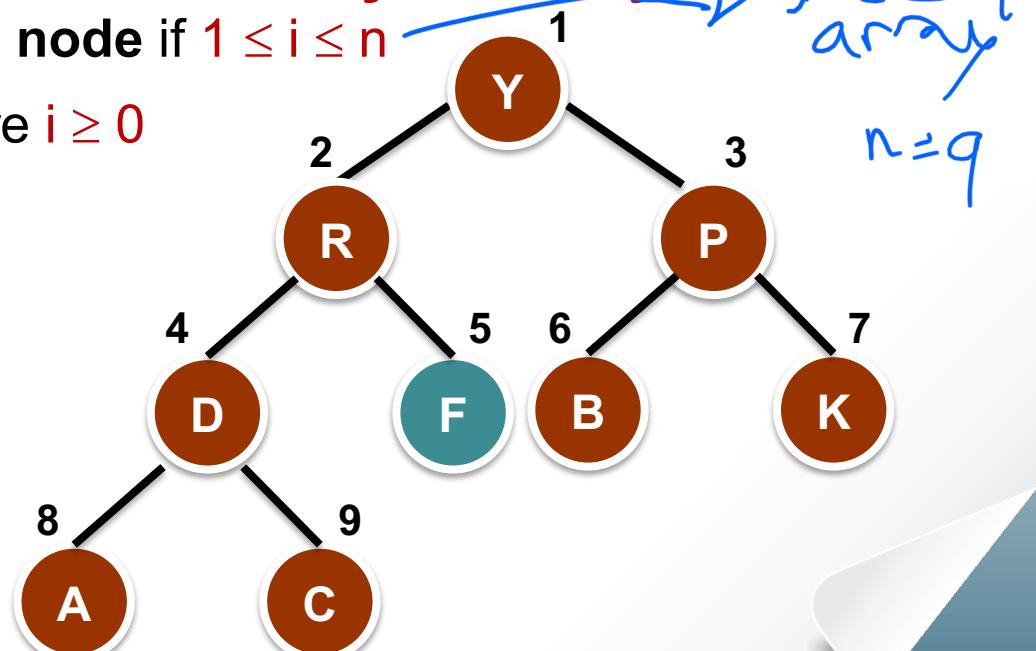


Heap Structure

- This means that every successive level of the tree must fill up from left to right. Further, an entire level must be full before any nodes at that level can have children nodes.
- Implementing the tree with n nodes by an array:**
 - Entry i in the array is a **tree node** if $1 \leq i \leq n$
 - Parent (i): return $\lfloor i/2 \rfloor$ where $i \geq 0$

$i = 5$

Parent (5) = $\lfloor 5/2 \rfloor = 2$

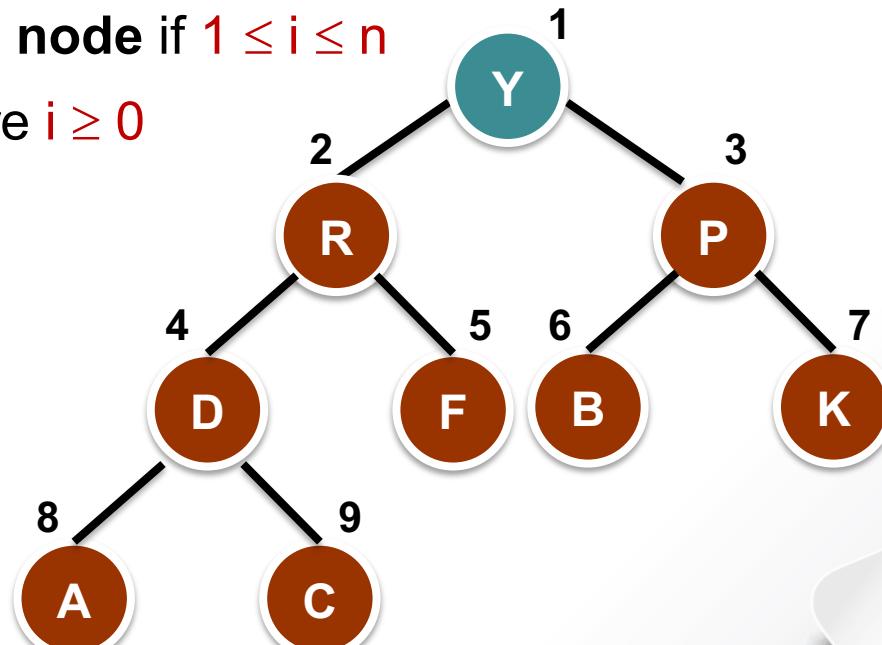


Heap Structure

- This means that every successive level of the tree must fill up from left to right. Further, an entire level must be full before any nodes at that level can have children nodes.
- Implementing the tree with n nodes by an array:**
 - Entry i in the array is a **tree node** if $1 \leq i \leq n$
 - Parent (i): return $\lfloor i/2 \rfloor$ where $i \geq 0$

$i = 1$

Parent (1) = $\lfloor 1/2 \rfloor = 0$



Heap Structure

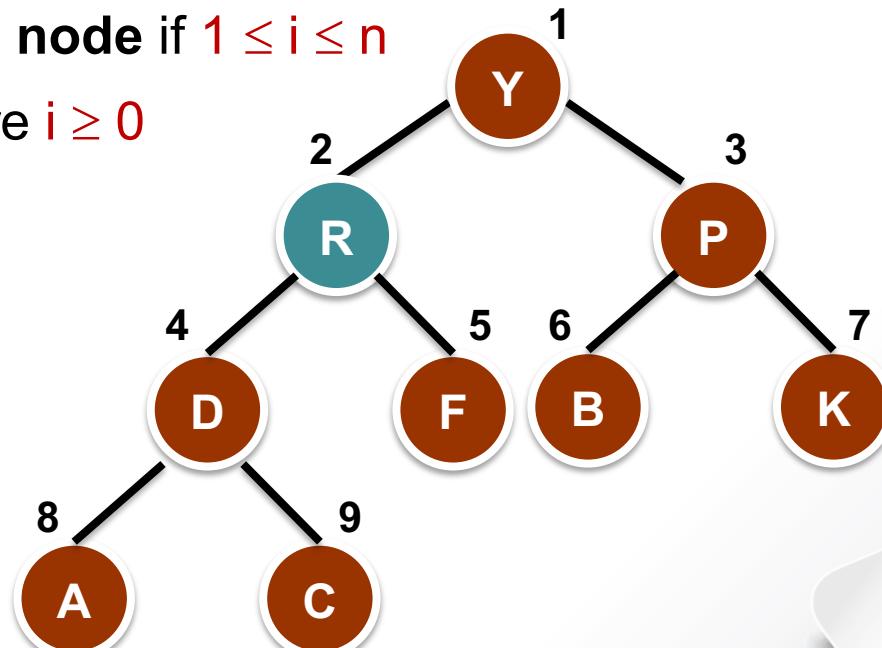
- This means that every successive level of the tree must fill up from left to right. Further, an entire level must be full before any nodes at that level can have children nodes.
- Implementing the tree with n nodes by an array:**

- 1) Entry i in the array is a **tree node** if $1 \leq i \leq n$
- 2) Parent (i): return $\lfloor i/2 \rfloor$ where $i \geq 0$
- 3) Left subtree of i : return $2i$

$i = 2$

Left subtree (i)

$$= 2 \times 2 = 4$$



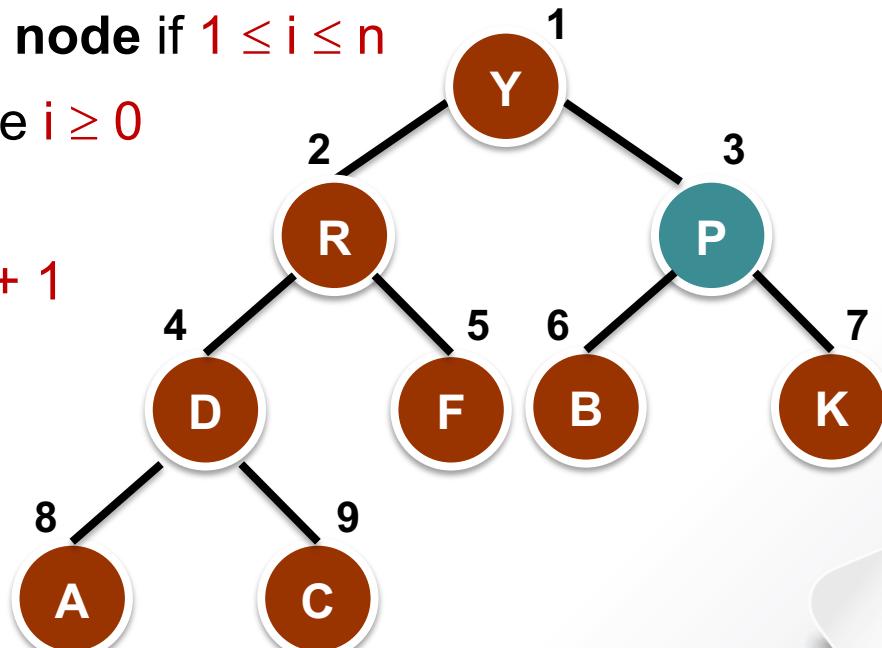
Heap Structure

- This means that every successive level of the tree must fill up from left to right. Further, an entire level must be full before any nodes at that level can have children nodes.
- Implementing the tree with n nodes by an array:**
 - Entry i in the array is a **tree node** if $1 \leq i \leq n$
 - Parent (i): return $\lfloor i/2 \rfloor$ where $i \geq 0$
 - Left subtree of i : return $2i$
 - Right subtree of i : return $2i + 1$

$i = 3$

Right subtree (i):

$$= (2 \times 3) + 1 = 7$$



Heap Structure

- This means that every successive level of the tree must fill up from left to right. Further, an entire level must be full before any nodes at that level can have children nodes.
- Implementing the tree with n nodes by an array:**

1) Entry i in the array is a **tree node** if $1 \leq i \leq n$

2) Parent (i): return $\lfloor i/2 \rfloor$ where $i \geq 0$

3) Left subtree of i : return $2i$

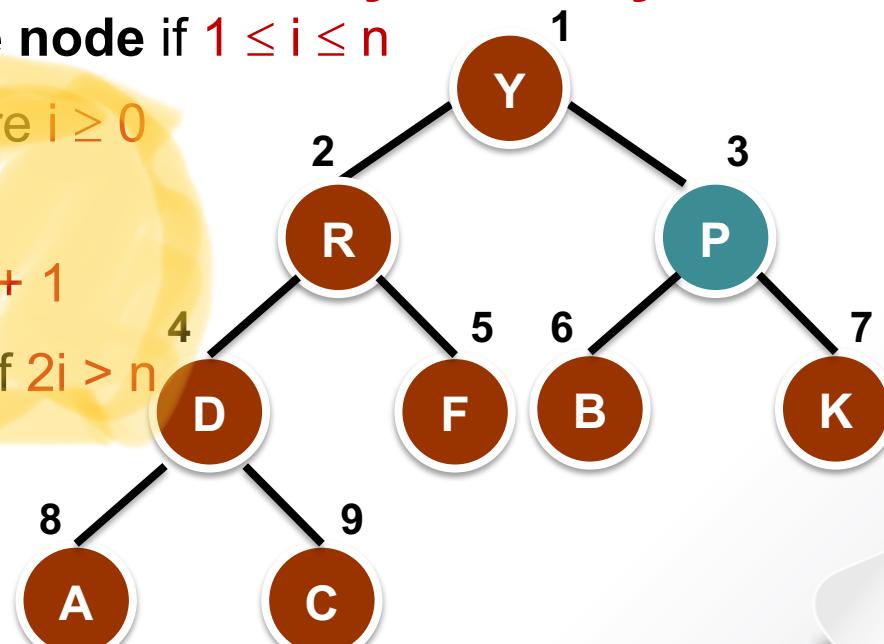
4) Right subtree of i : return $2i + 1$

5) $\text{array}[i]$ is a leaf if and only if $2i > n$

$$i = 3, \quad n = 9$$

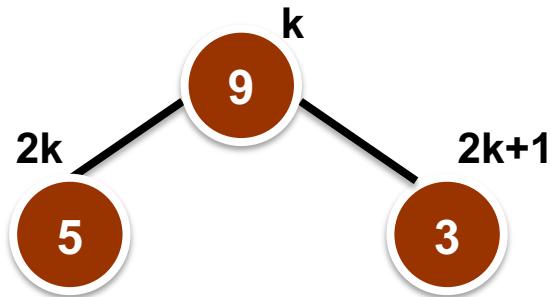
2 x 3 is not > 9

Node 3 is not a leaf node.



Heap Structure

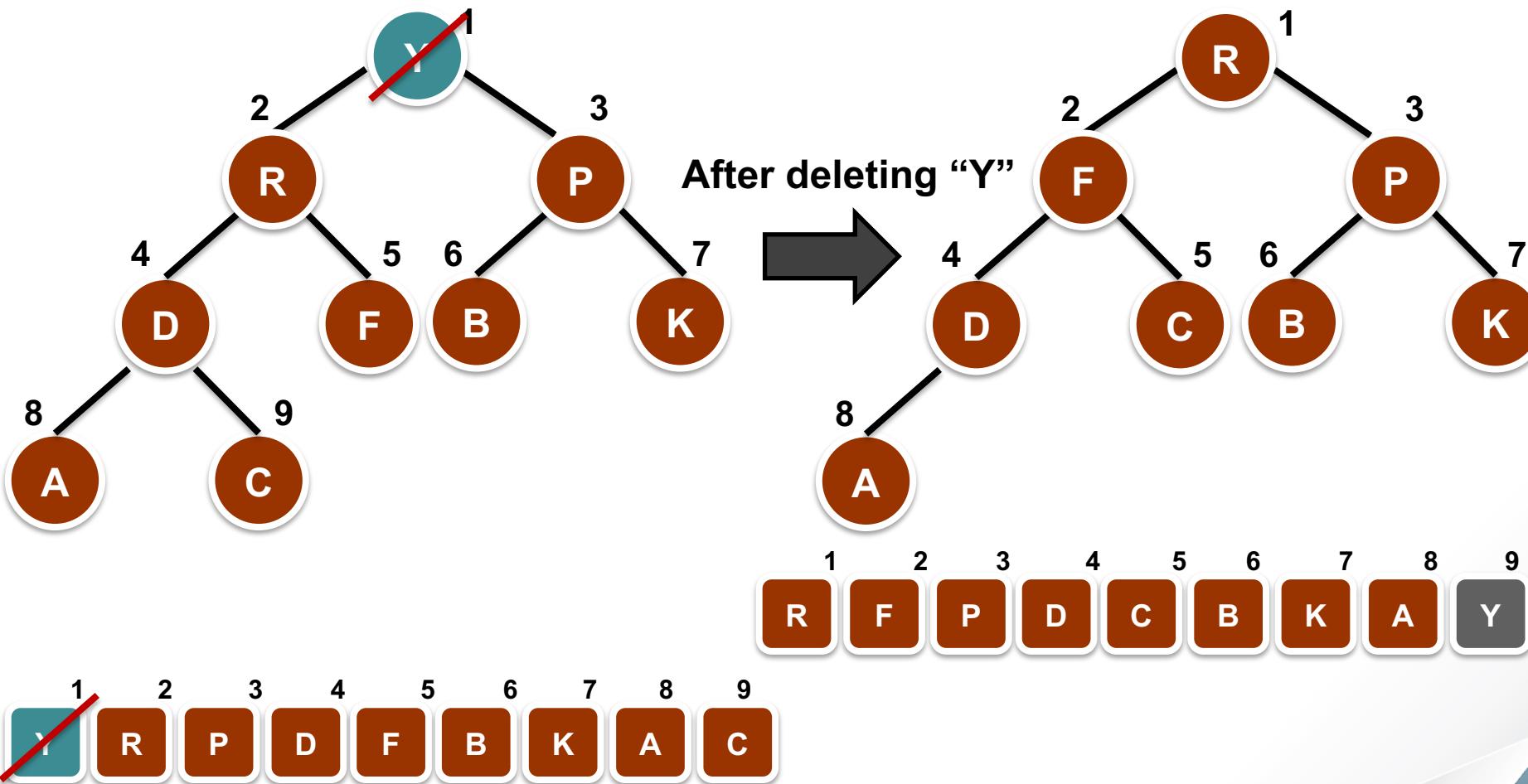
- Therefore the partial order tree property requires that for all positions k in the list, the key at k is at least as large as the keys at $2k$ and $2k + 1$ (if these positions exist).



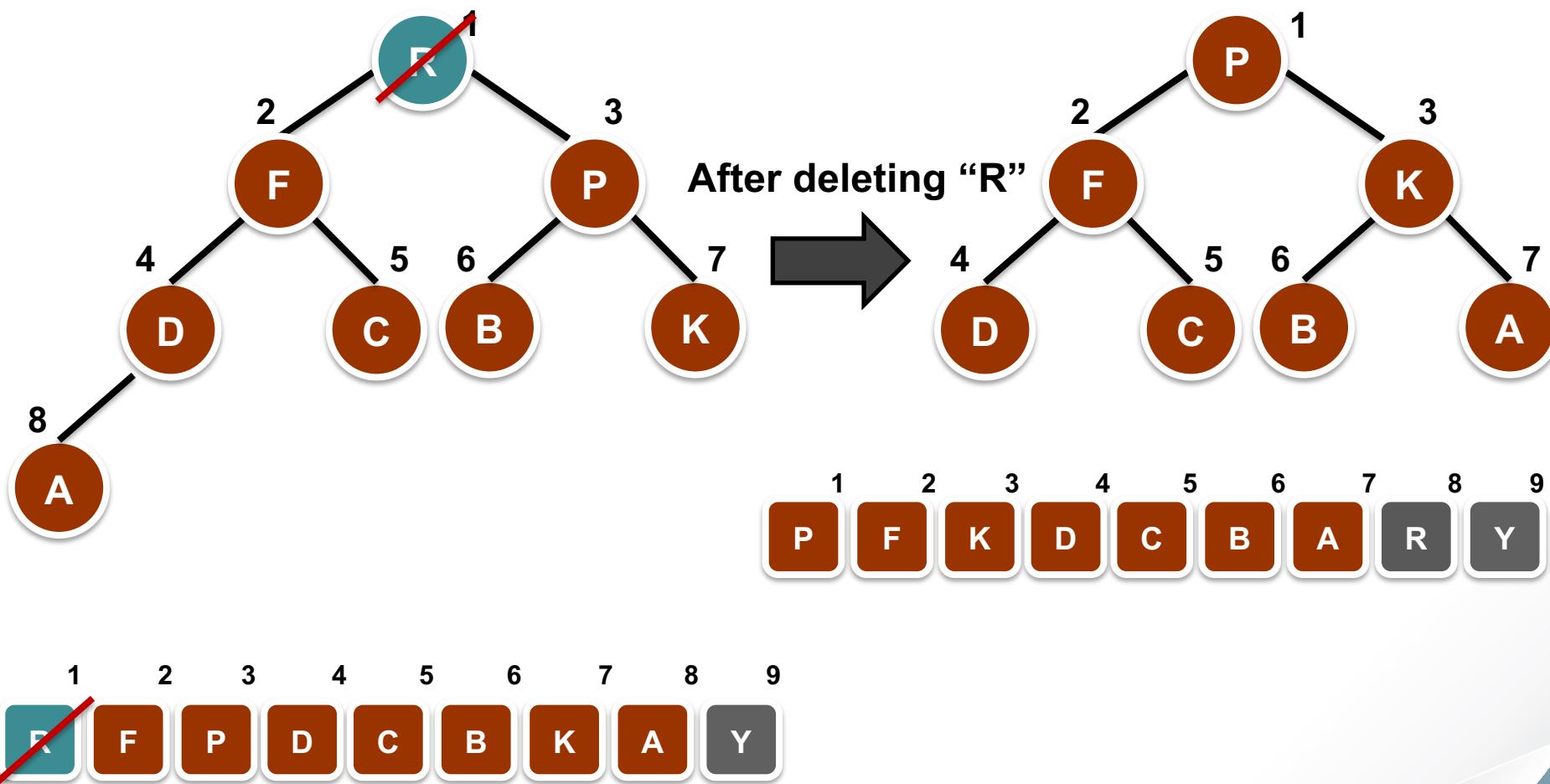


Heapsort (Example)

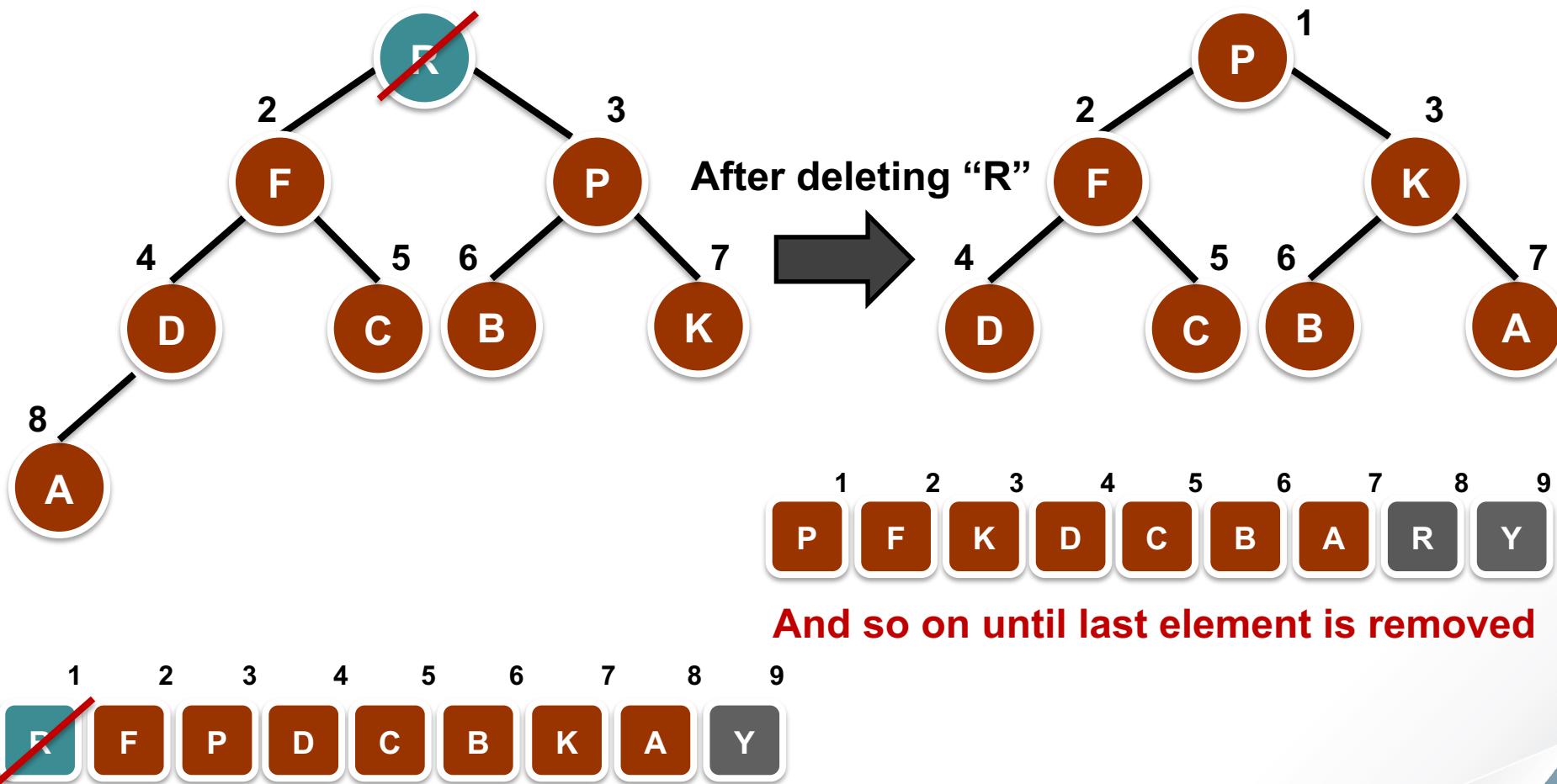
Heapsort (Example)



Heapsort (Example)



Heapsort (Example)





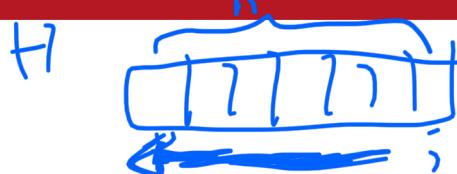
Heapsort Method

Heapsort Method

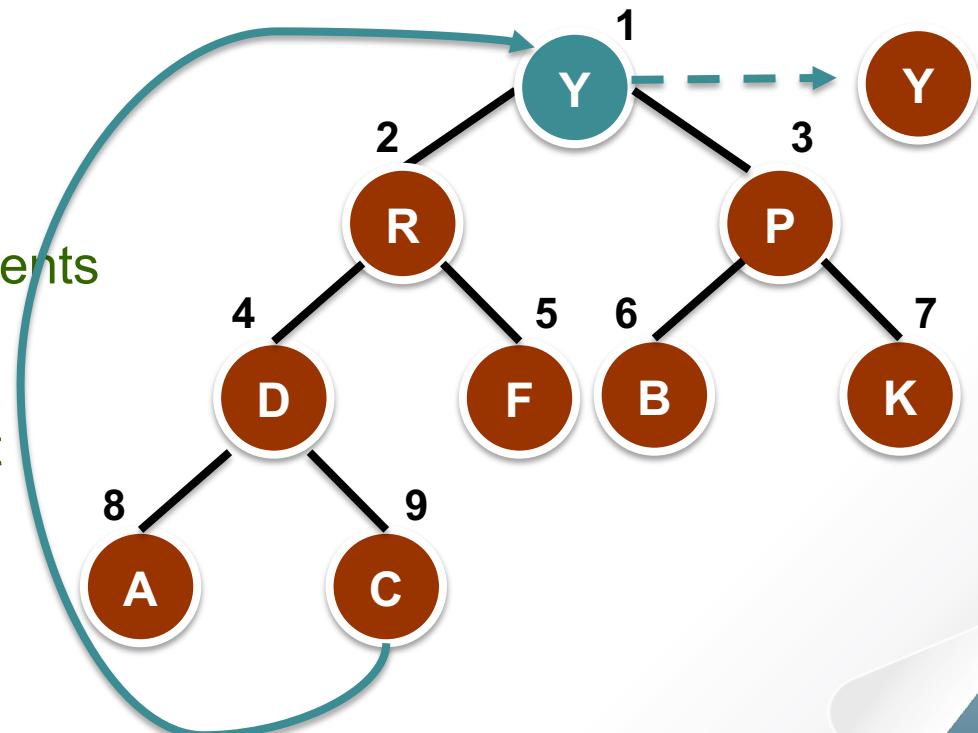
heapSort (array, n)

```
{
    construct heap H from array with n elements;

    for (i = n; i >= 1; i--)
    {
        curMax = getMax(H);
        deleteMax(H);
        // as result, H has i - 1 elements
        array[i] = curMax;
        // insert curMax in sorted list
    }
}
```



Take out last
and re-insert



Heapsort Method

deleteMax(H)

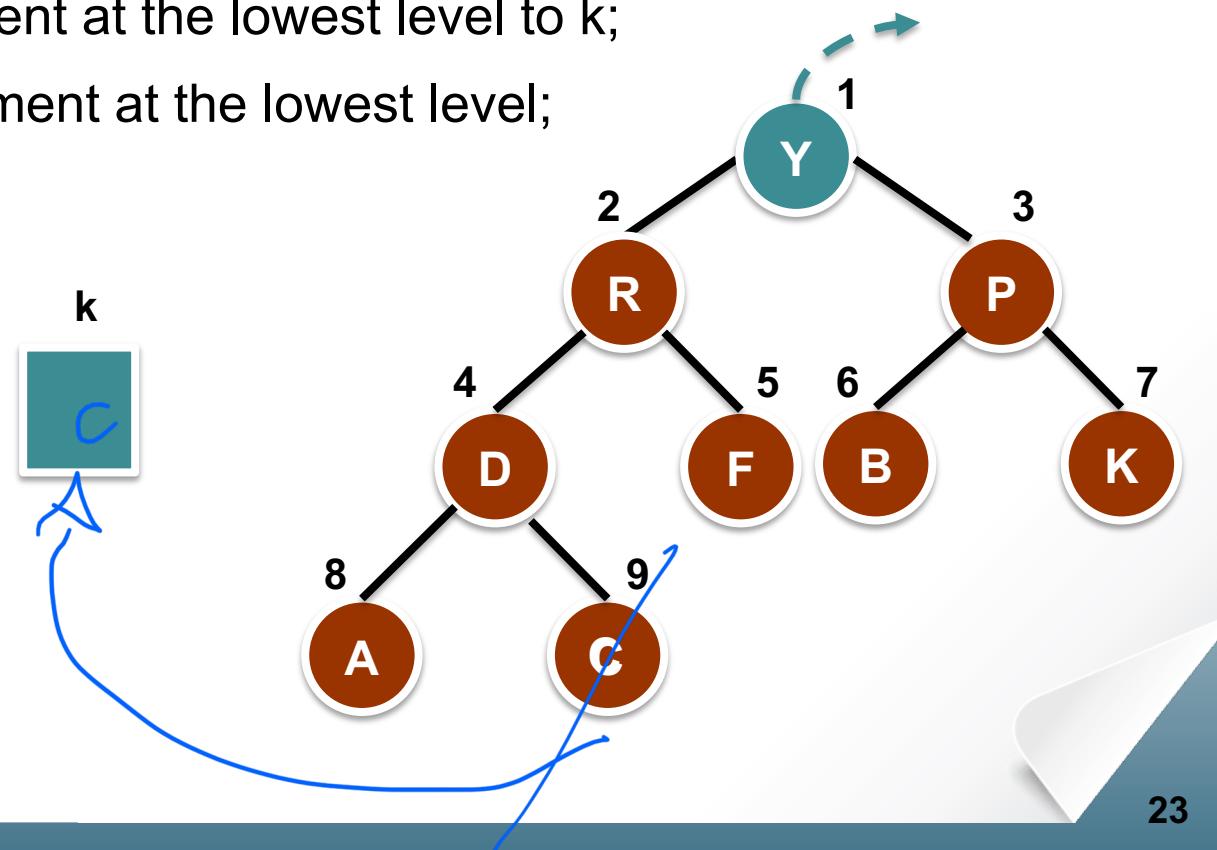
{

copy the rightmost element at the lowest level to k;

delete the rightmost element at the lowest level;

fixHeap(H, k);

}



fixHeap

fixHeap(H, k) { // recursive

 if (H is a leaf)

 insert k in root of H;

 else {

 compare left child with right child;

 largerSubHeap = the larger child of H;

 if (k >= key of root(largerSubHeap))

 insert k in root of H;

 else {

 insert root(largerSubHeap) in root of H;

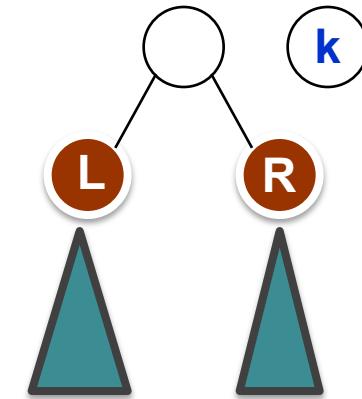
 fixHeap(largerSubHeap, k);

 }

 }

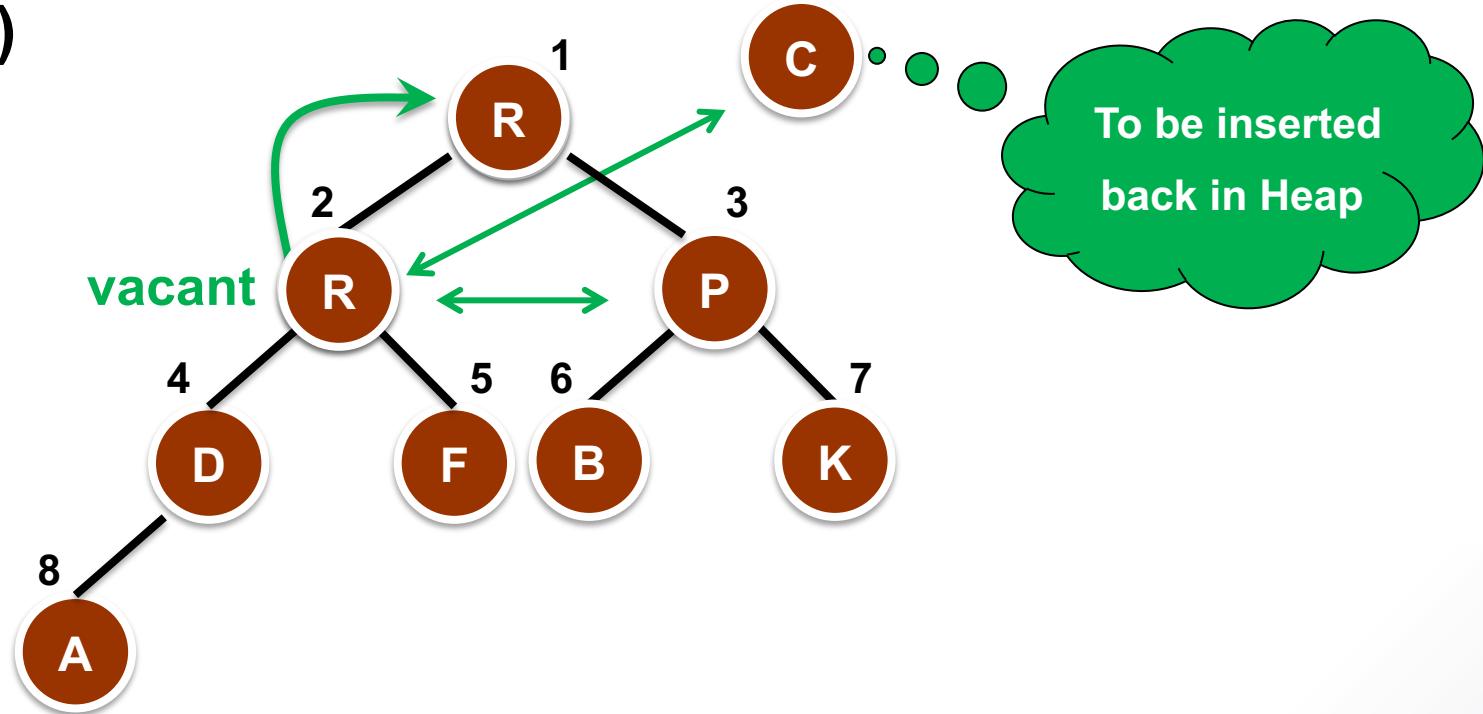
}

Something we want to insert



fixHeap

fixHeap(H,C)

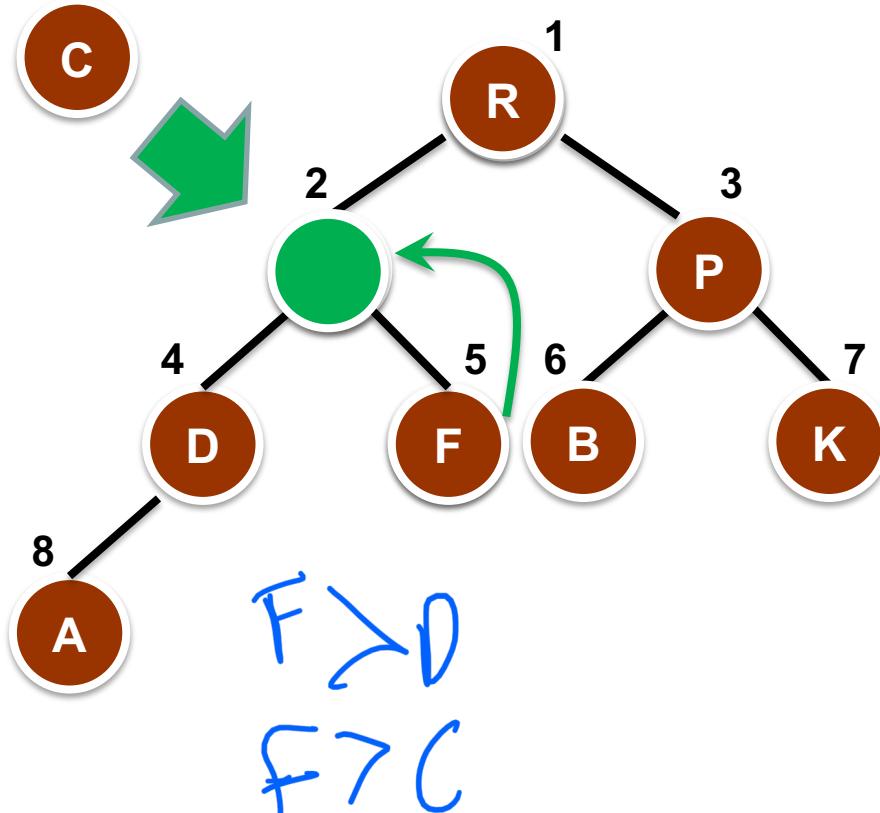


R > P and R is also > C; so R is inserted into Root, and the original slot of R becomes vacant.

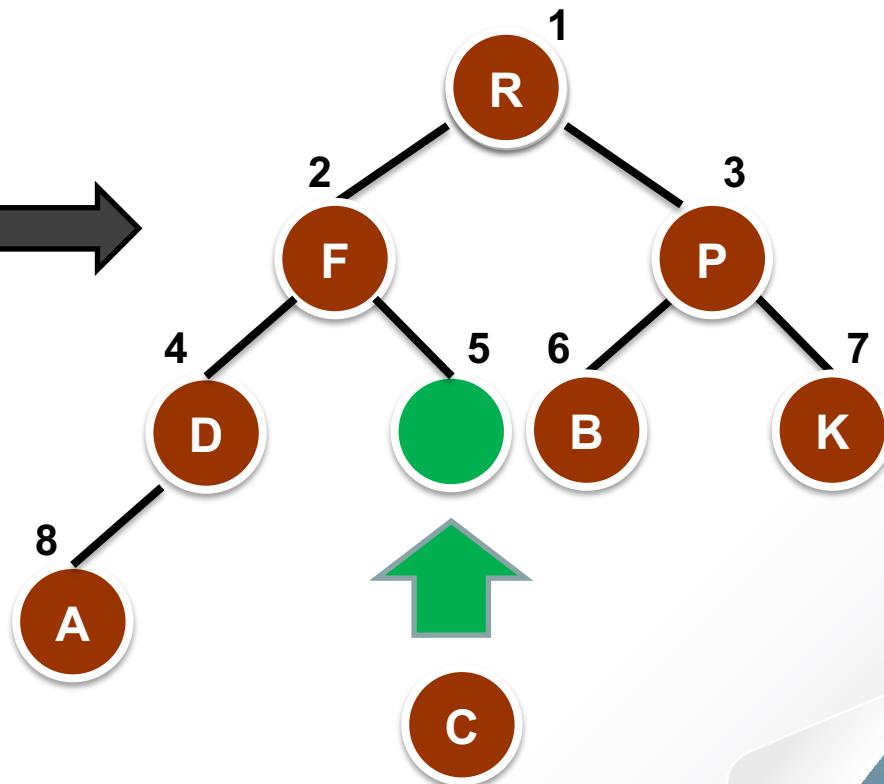
fixHeap is called again to reinsert C into the sub-heap.

fixHeap

Call fixHeap



At this point, the subtree is a leaf.
Hence, C is inserted.



Call fixHeap

fixHeap

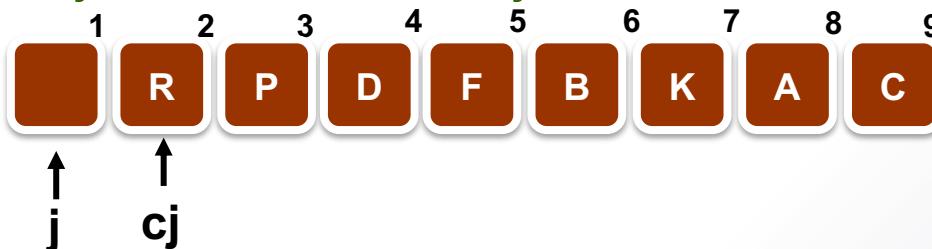
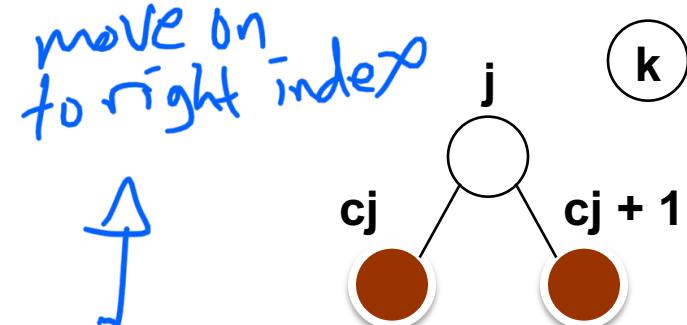
```

fixHeap(H, k)          // iterative
{
    int j = 1,           // root of the heap
    index ← cj = 2;      // left child of the root
    while (cj <= currentSize) → n
    { // cj should be the larger child of j
        if (cj < currentSize && H[cj] < H[cj+1]) cj++;
        if (k >= H[cj]) break; // should put k in H[j]
        H[j] = H[cj];        // move larger child to H[j]
        j = cj;              // move down one level
        cj = 2 * j;          // cj is the left child of j
    }
    H[j] = k;
}

```

move on to right index

↑





Heap Construction

Heap Construction

Construct a heap from an array

Start by putting all elements of the array in a heap structure in arbitrary order; then, “heapifying” the heap structure.

constructHeap(array, H)

{

 put all elements of array into a heap structure H in
 arbitrary order;

 heapifying(H);

}

Uses the *fixheap* function
mentioned earlier

Heap Construction

heapifying(H)

{

```
if (H is not a leaf) {  
    heapifying(left subtree of H);  
    heapifying(right subtree of H);  
    k = root(H);  
    fixHeap(H, k);
```

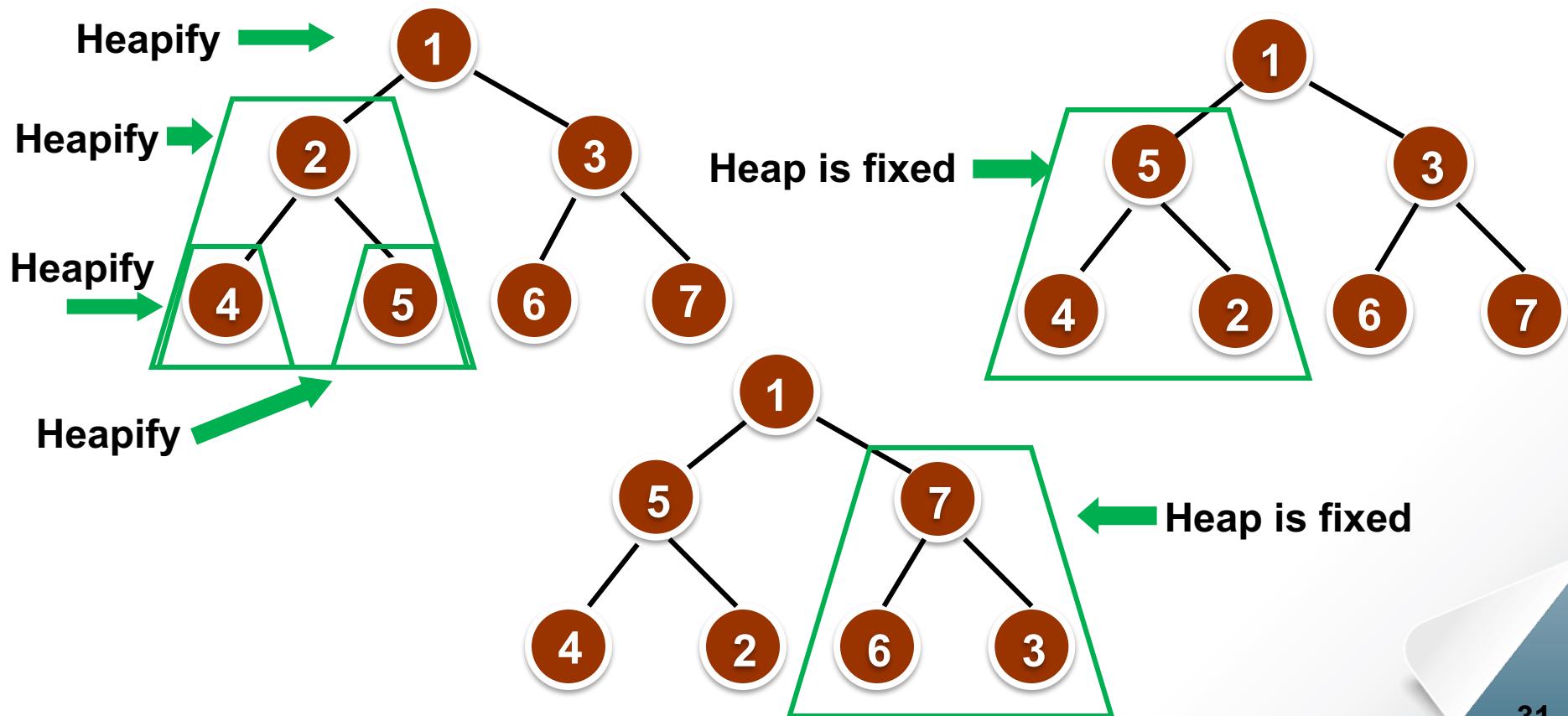
}

}

Post-order traversal
of a binary tree

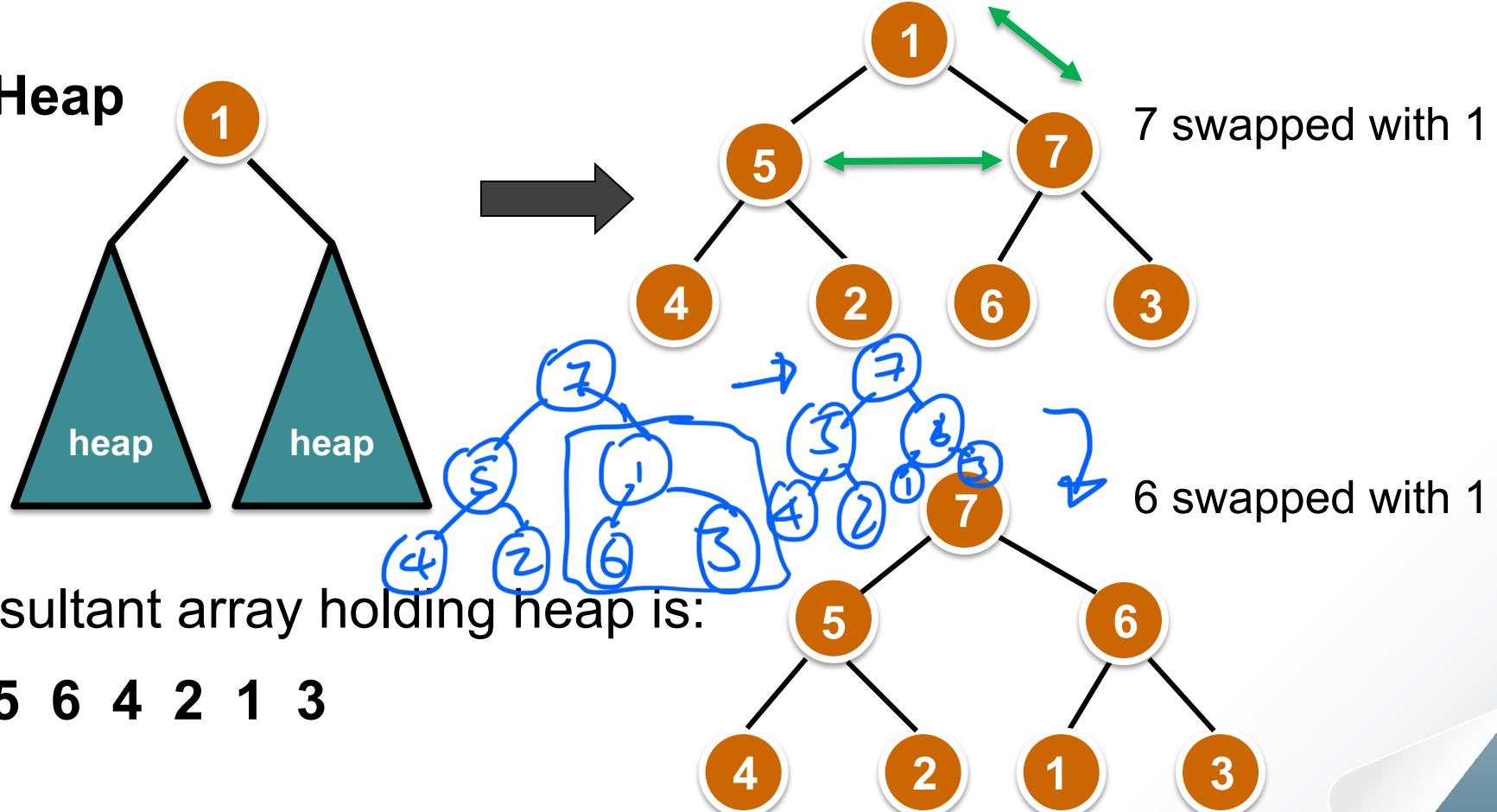
Heap Construction

Assume elements in initial arbitrary order: 1 2 3 4 5 6 7



Heap Construction

fixHeap





Time Complexity of Heapsort

$O(n \log n)$

build-max-heap : $O(n)$

heapsify : $O(\log n)$, called $n-1$ times

Time Complexity of fixHeap

```

fixHeap(H, k)          // recursive
{
    if (H is a leaf)      // Heap has just one node      O(1)
        insert k in root of H;                      O(1)

    else {
        1 Comparison LargerSH = Sub-Heap at larger child of H's root;      O(1)
        1 Comparison if (k >= LargerSH's root key)      O(1)
            insert k in root of H;                      O(1)

        else {
            insert LargerSH's root key in root of H;      O(1)
            fixHeap(LargerSH, k);
        }
    }
}

```

STH

*worst case
 $\geq 2 \times \text{Height}$*

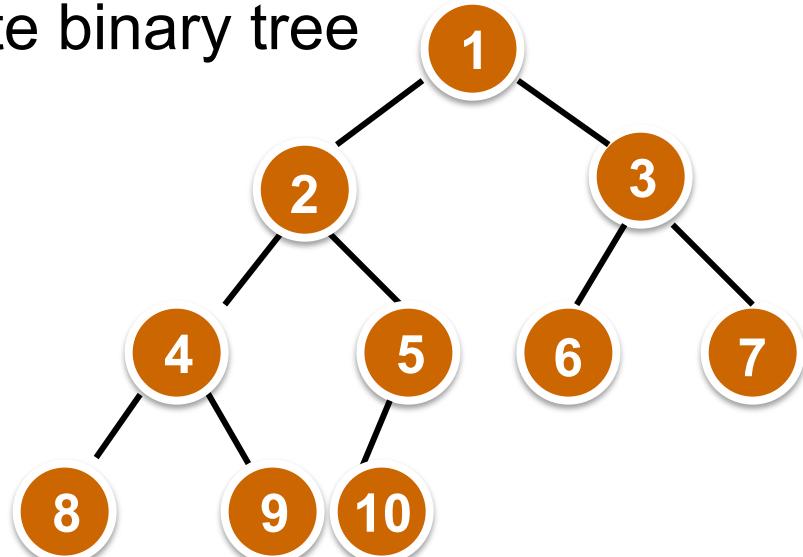
Each recursive call moves down a level

Total no. of key comparisons $\leq 2 \times \text{tree height}$

Time Complexity of fixHeap

Recall: A heap is a nearly complete binary tree

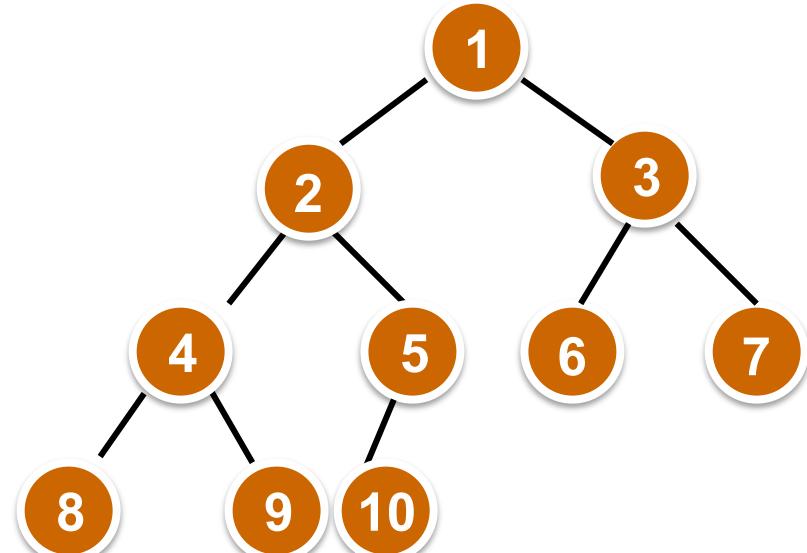
Note: A complete binary tree of k levels has $2^k - 1$ nodes (prove by mathematical induction)



Time Complexity of fixHeap

A heap with

- 1 level has $\leq 1 (= 2^1 - 1)$ node;
- 2 levels has $\leq 3 (= 2^2 - 1)$ nodes;
- 3 levels has $\leq 7 (= 2^3 - 1)$ nodes;
- 4 levels has $\leq 15 (= 2^4 - 1)$ nodes;
- $k - 1$ levels has $\leq 2^{k-1} - 1$ nodes;
- k levels has $\leq 2^k - 1$ nodes.



Time Complexity of fixHeap

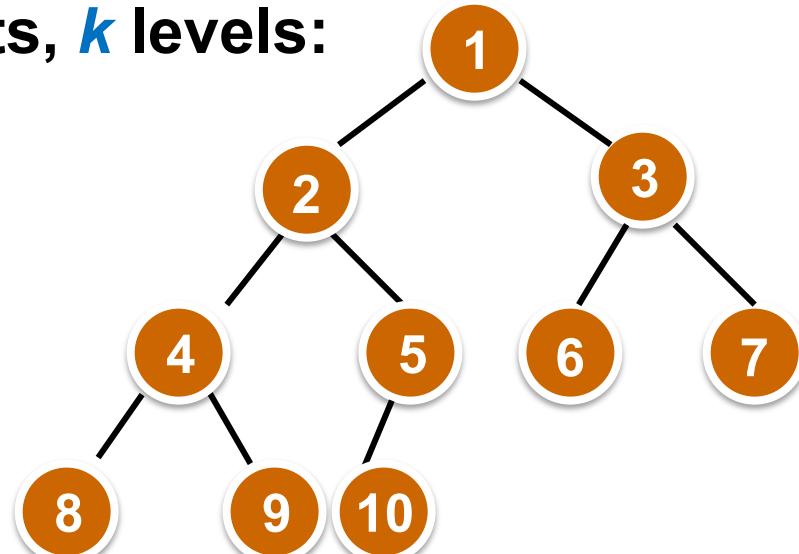
Assume the heap has n elements, k levels:

$$2^{k-1} - 1 < n \Rightarrow 2^{k-1} \leq n$$

$$2^{k-1} \leq n \leq 2^k - 1$$

$$\Rightarrow k - 1 \leq \lg n < k$$

$$\Rightarrow k - 1 = \lfloor \lg n \rfloor$$



Height of a heap with n nodes is $O(\lg n)$.

Worst-case time complexity of fixHeap is $O(\lg n)$.

Time Complexity of heapifying

```
heapifying(H)                                W(n)
{
    if (H is not a leaf)                      O(1)
    {
        heapifying(left subtree of H);       W((n-1)/2)
        heapifying(right subtree of H);      W((n-1)/2)
        k = root(H);                      O(1)
        fixHeap(H, k);                   2 Ign
    }
}
```

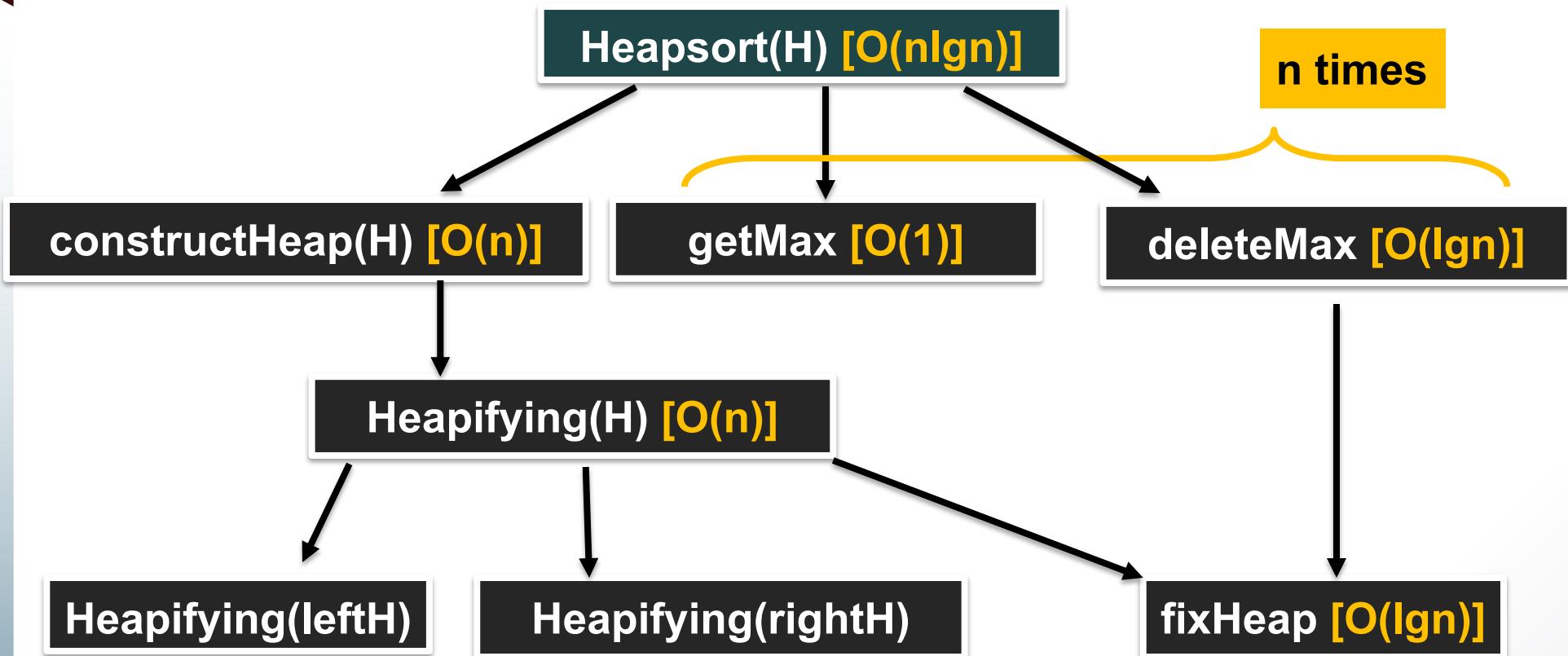
Time Complexity of heapifying

- Assume a heap is a **full** binary tree, i.e. $n = 2^d - 1$ for some non-negative integer d . The worst-case time complexity of `heapify()`, i.e. $W(n)$, satisfies:

$$W(n) = 2W((n-1)/2) + 2\lg n$$

- Solving this equation gives $W(n) = \mathcal{O}(n)$ comparisons of keys in the worst-case.
(How to solve the recurrence equation is not required)

Heapsort Performance



Priority Queues

Priority Queues (Optional, for self-learning)

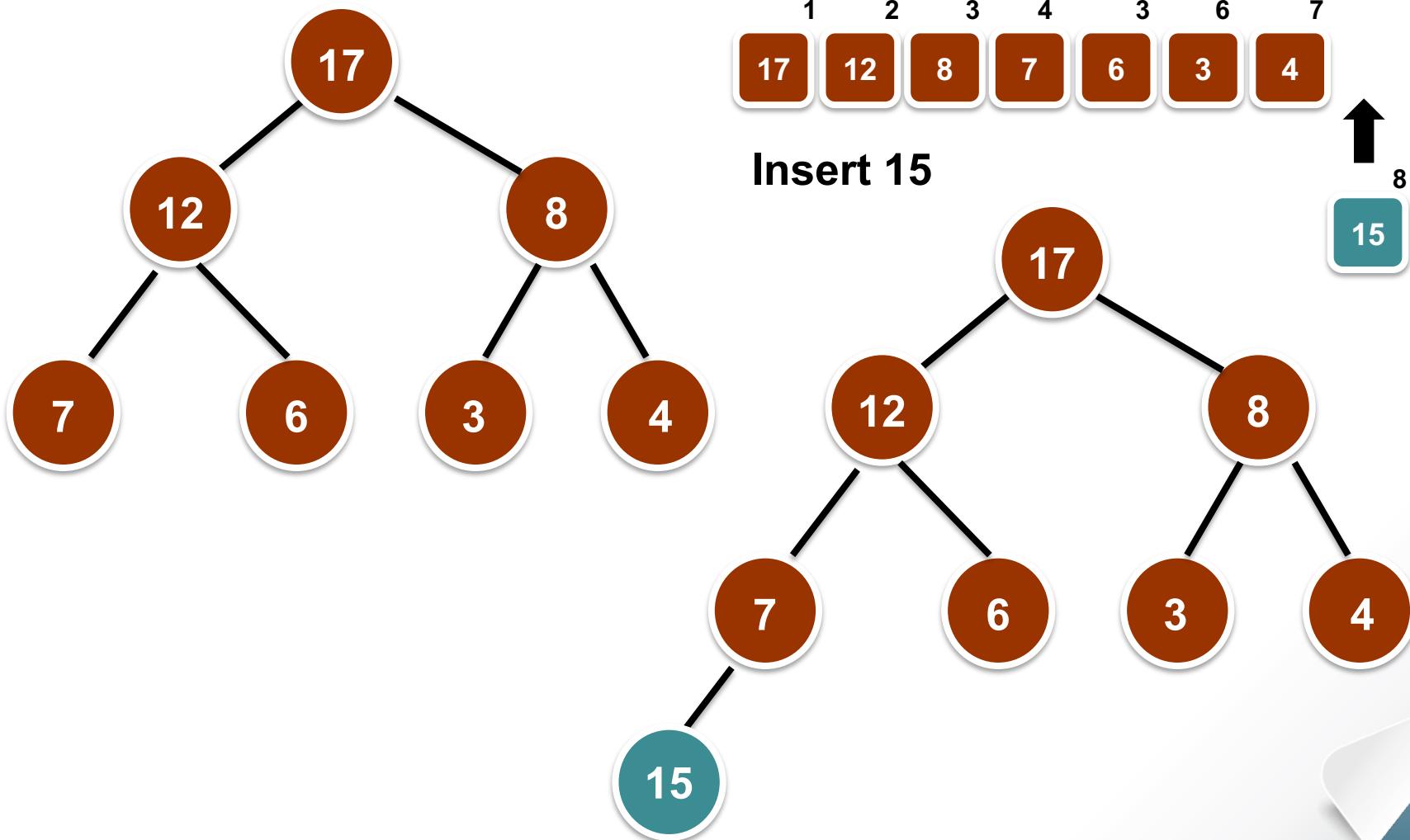
- A priority queue is a data structure for maintaining a set S of elements, each with a key value. This key is considered as the ‘priority’ of the element in S .
- Priority queues are frequently used in job scheduling, simulation systems etc.
- A priority queue supports the following operations:
 - **insert(x)** inserts the element x into a priority queue pq .
 - **Maximum(pq)** returns largest key from pq .
 - **extractMax(S)** removes largest key and re-arranges pq .
- Using a heap allows an efficient way of implementing a priority queue.

Priority Queues

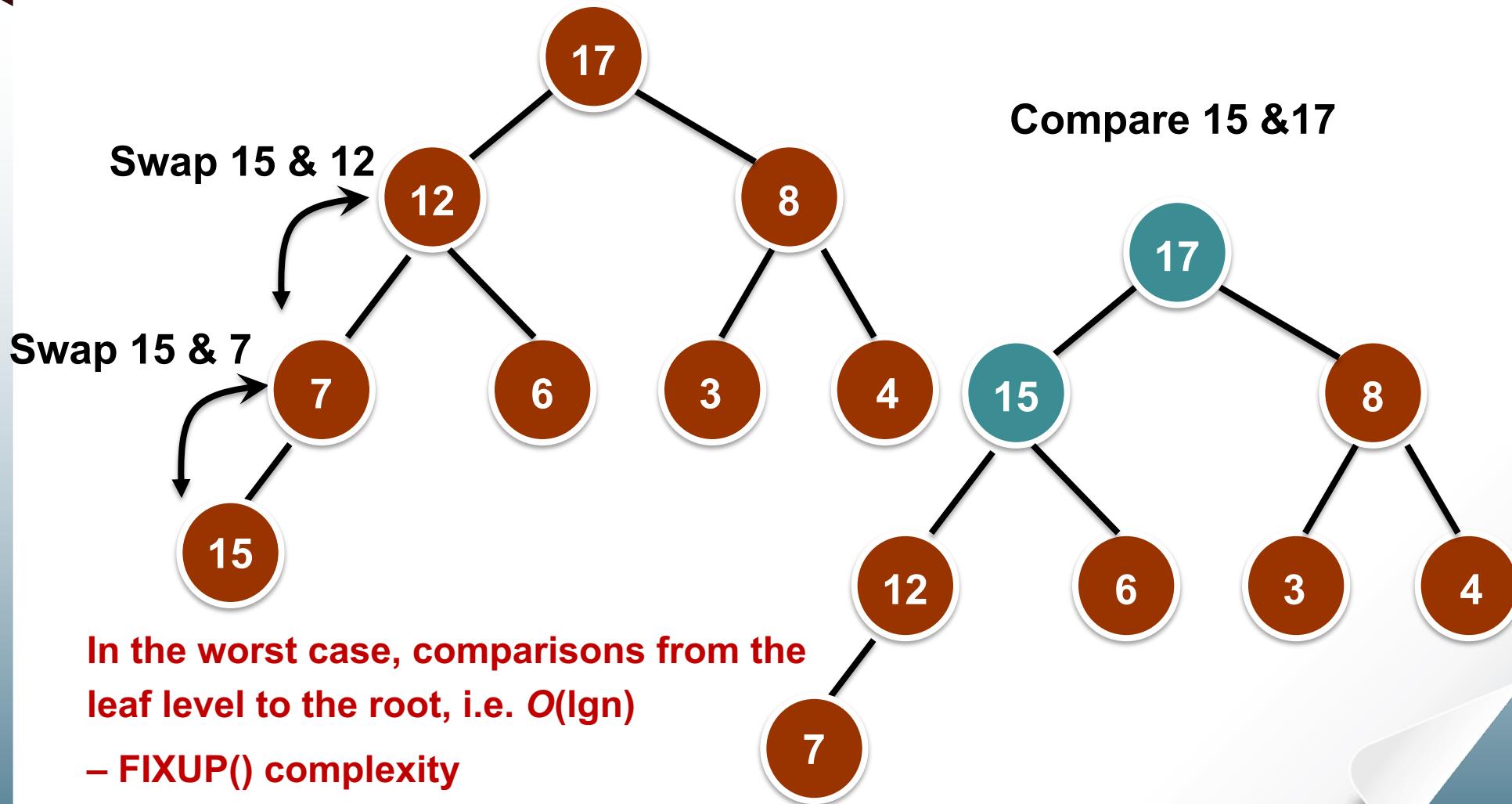
Class pq // Java code

```
{    private:  
        ALIST pq;  
        int N;          // size of priority queue  
    public:  
        // initialisation & other methods such as EMPTY omitted  
        void insert (item i)  
        { pq[++N] = i; fixUp(pq,N); }  
        item extractMax()  
        { swap(pq[1], pq[N]); fixDown(pq, 1, N - 1);  
         return pq[ N -- ]; }  
    }
```

Action of Fixup



Action of Fixup



Action of Fixup

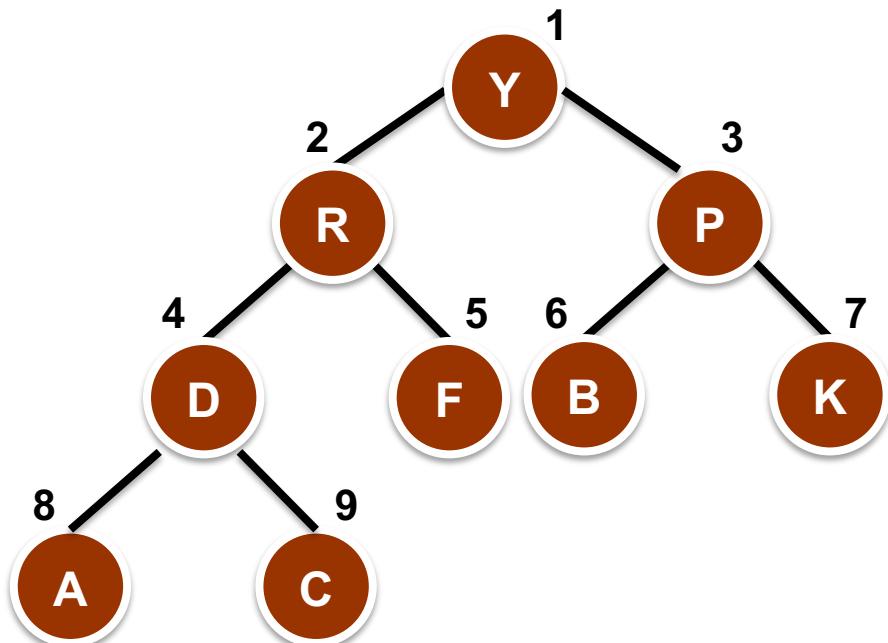
- The running time of `insert()` on an n -element heap is $O(\lg n)$ – same as `fixUp()`
- The running time of `extractMax()` on an n -element heap is $O(\lg n)$ – same as `fixHeap()`
- The running time of `Maximum(pq)` (i.e. `getMax()`) on an n -element heap is $O(1)$ - simply gets `pq[1]`
- So a heap can support any priority queue operation on a set of n elements in $O(\lg n)$ time



Heapsort (Summary)

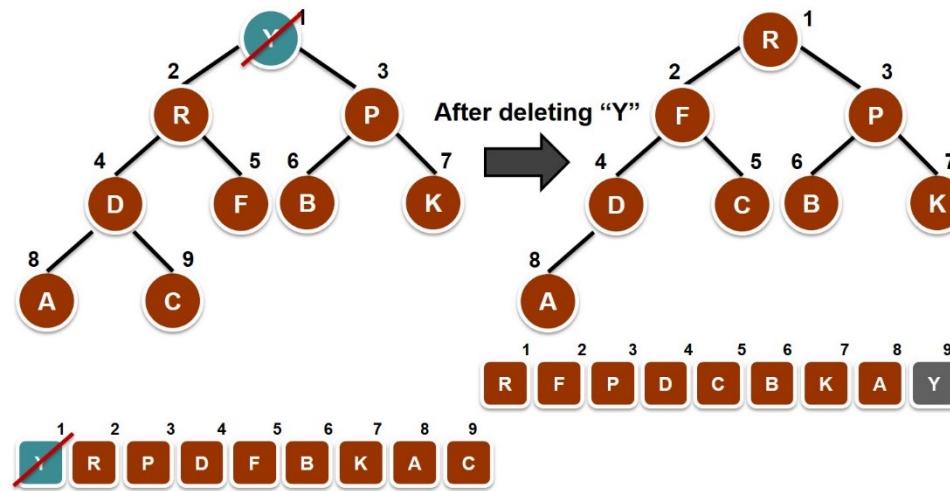
Summary

- Heapsort is a sorting algorithm using the data structure of heap.
- A (**maximising**) heap is an almost complete binary tree that satisfies the (**maximising**) partial order tree property.



Summary

- Heapsort is a sorting algorithm using the data structure of heap.
- A (**maximising**) heap is an almost complete binary tree that satisfies the (**maximising**) partial order tree property.
- Heapsort works by repeatedly deleting the root (maximum node) of the heap, and repair the damage (**fixHeap**).



Summary

- Heapsort is a sorting algorithm using the data structure of heap.
- A (**maximising**) heap is an almost complete binary tree that satisfies the (**maximising**) partial order tree property.
- Heapsort works by repeatedly deleting the root (maximum node) of the heap, and repair the damage (**fixHeap**).
- Heap is constructed by recursively calling **fixHeap** in a post-order traversal of the binary tree.
- **In worst-case**, **heapsort** takes time $\Theta(n \lg n)$, and **heap construction** takes linear time $\Theta(n)$.



Comparison of Sorting Algorithms

Comparison of Sorting Algorithms

Time complexity comparison:

| | Best | Average | Worst |
|-----------|------------|------------|------------|
| Insertion | n | n^2 | n^2 |
| Merge | $n \log n$ | $n \log n$ | $n \log n$ |
| Quick | $n \log n$ | $n \log n$ | n^2 |
| *Radix | n | n | n |
| Heap | $n \log n$ | $n \log n$ | $n \log n$ |

* Radix sort is not required

Empirical Comparison

Compared by time (in milliseconds)

| Insertion | 0.1 | 168 | 342 | 23,382 |
|-----------|-----|-----|-----|--------|
| Merge | 2.0 | 2.3 | 2.2 | 30 |
| Quick | 0.7 | 0.9 | 0.7 | 12 |
| *Radix | 1.6 | 1.6 | 1.6 | 18 |
| Heap | 3.4 | 3.5 | 3.6 | 49 |

Up Down 100,000 numbers

Reference: Shaffer, C. A. (2001). *A practical introduction to data structures and algorithm analysis*. Upper Saddle River, NJ: Prentice Hall.

'UP' and **'DOWN'** columns show the performance for inputs of size 10,000 where the numbers are in ascending (sorted) and descending (reversely sorted) order. Figures are timings obtained using workstation running UNIX.



SC2001/CE2101/ CZ2101: Algorithm Design and Analysis

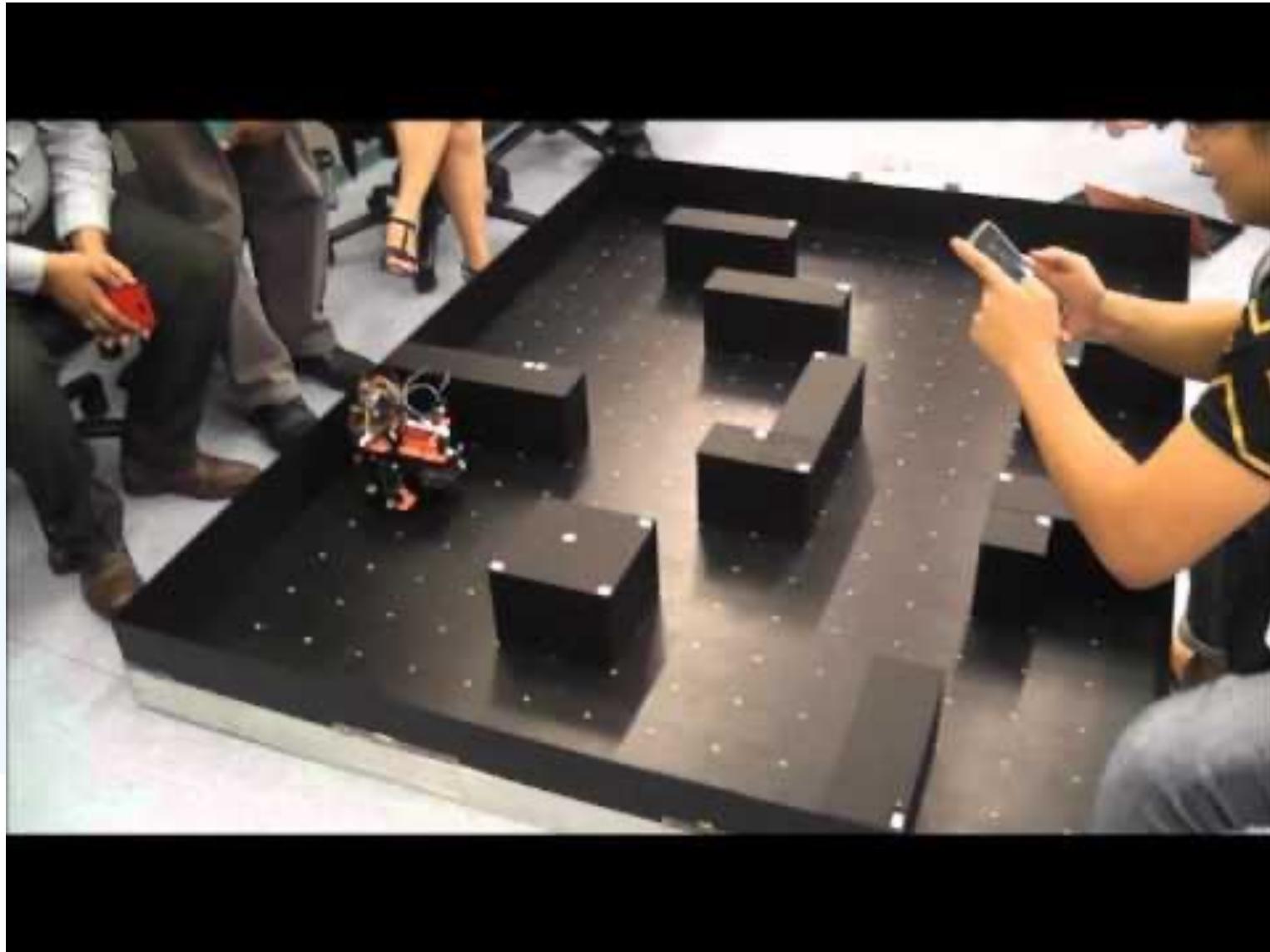
**Greedy Algorithms;
Dijkstra's Algorithm; Prim's Algorithm**

Instructor: Asst. Prof. LIN Shang-Wei

Courtesy of Dr. Ke Yiping, Kelly's slides



Greedy Algorithms



Learning Objectives

At the end of this lecture, students should be able to:

- Explain the strategy of Greedy algorithms
- Solve single-source shortest paths problem using Dijkstra's algorithm
- Prove the correctness of Dijkstra's algorithm
- Describe Prim's algorithm for finding minimum spanning trees (MSTs)
- Prove the correctness of Prim's algorithm

Greedy Algorithms

- In optimisation problems, the algorithm needs to make a series of choices whose overall effect is to minimise the total cost, or maximise the total benefit, of some system.
- There is a class of algorithms, called the **greedy algorithms**, in which we can find a solution by using only knowledge available at the time when the next choice (or guess) must be made.
- Each individual choice is the best within the knowledge available at the time.

Greedy Algorithms

- Each individual choice is not very expensive to compute.
- A choice cannot be undone, even if it is found to be a bad choice later.
- Greedy algorithms cannot guarantee to produce the optimal solution for a problem.



Dijkstra's Algorithm

Dijkstra's Algorithm

Shortest Path Problem:

The problem of finding the shortest path from one vertex in a graph to another vertex. "Shortest" may be the least number of edges, or the least total weight, etc.

Dijkstra's Algorithm:

This is an algorithm to find the shortest paths from a single source vertex to all other vertices in a **weighted, directed** graph. All weights must be **nonnegative**.

Dijkstra's Algorithm

Dijkstra's algorithm keeps two sets of vertices:

- **S** ---- the set of vertices whose shortest paths from the source node have already been determined [they form the tree]
- **V – S** ---- the remaining vertices

The other data structures needed are:

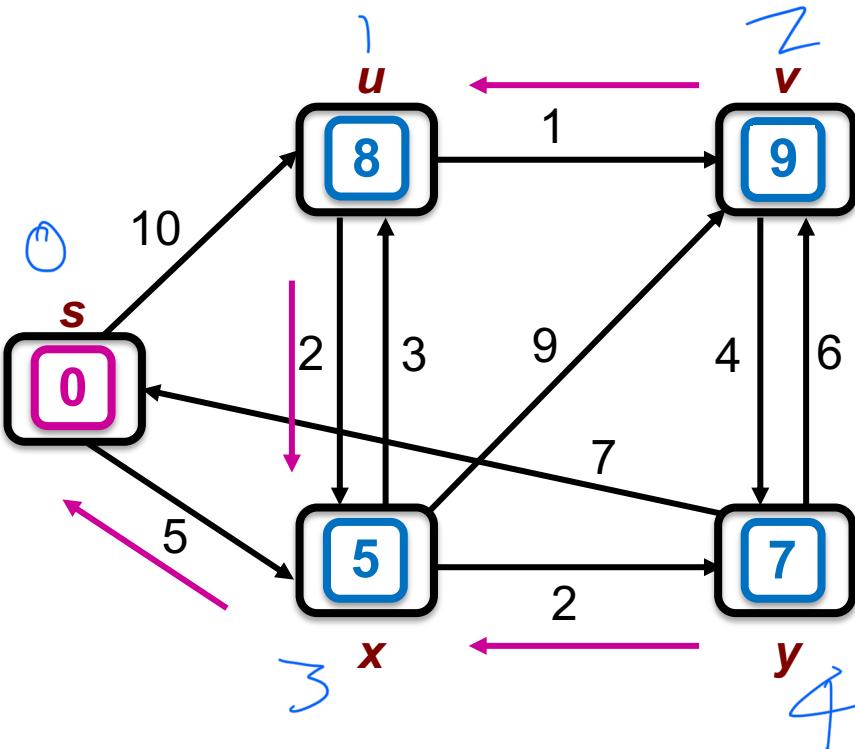
- **d** ---- array of estimates for the lengths of shortest paths from source node to all vertices
- **pi** ---- an array of predecessors for each vertex

Basic Steps

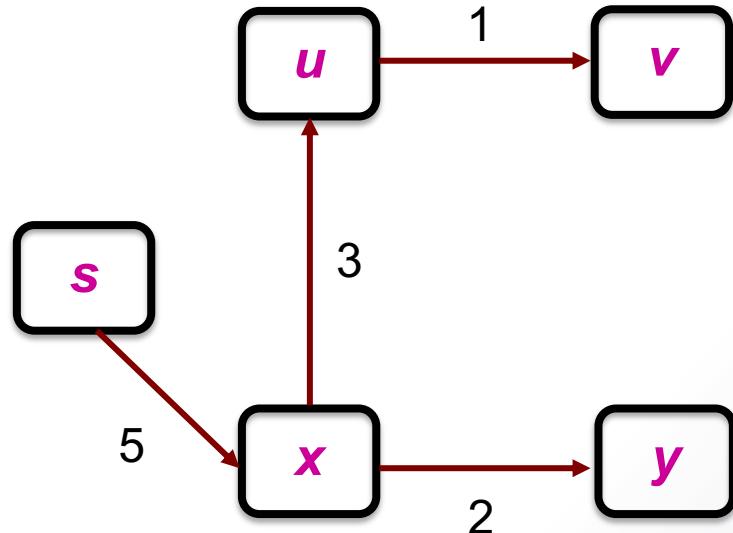
The basic steps are:

1. Initialise \mathbf{d} and \mathbf{pi}
2. Set \mathbf{S} to empty
3. While there are still vertices in $\mathbf{V} - \mathbf{S}$
 - i. Move \mathbf{u} , the vertex in $\mathbf{V} - \mathbf{S}$ that has the shortest path estimate from source, to \mathbf{S}
 - ii. For all the vertices in $\mathbf{V} - \mathbf{S}$ that are connected to \mathbf{u} , update their estimates of shortest distances to the source

A Toy Example

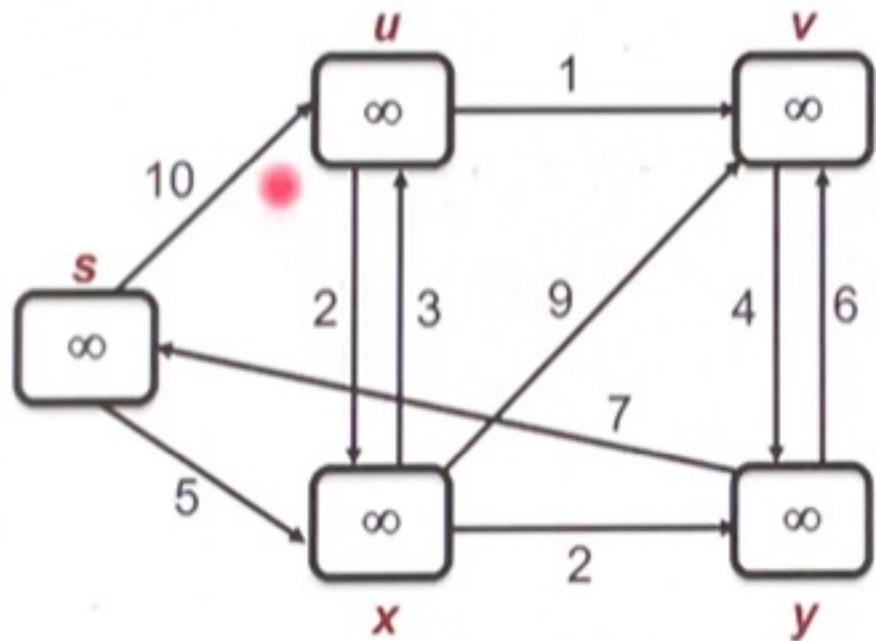


Shortest paths from *s* to other vertices



A Toy Example

S: { } 

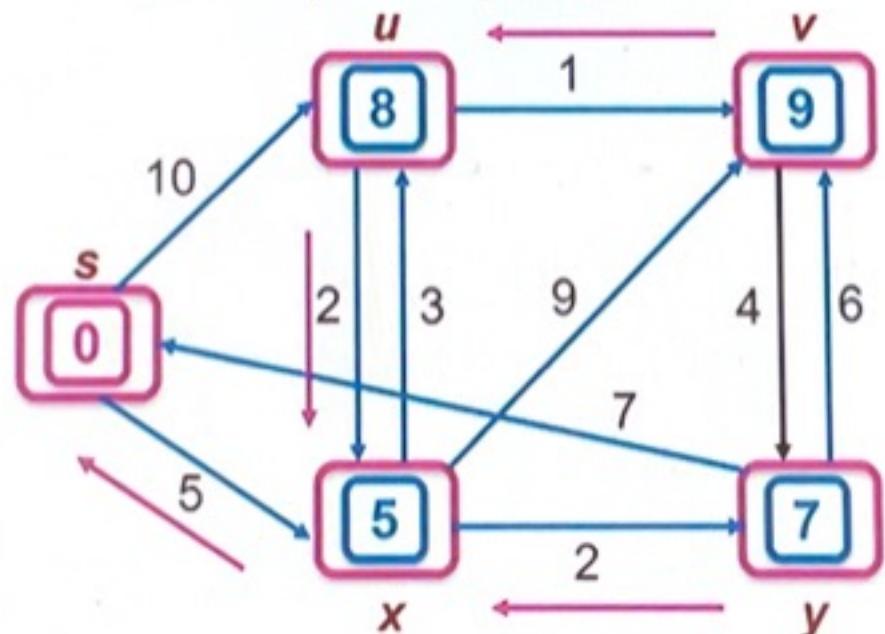


| d | s | x | u | v | y |
|---|---|---|---|---|---|
| | | | | | |

| pi | s | x | u | v | y |
|----|---|---|---|---|---|
| | | | | | |

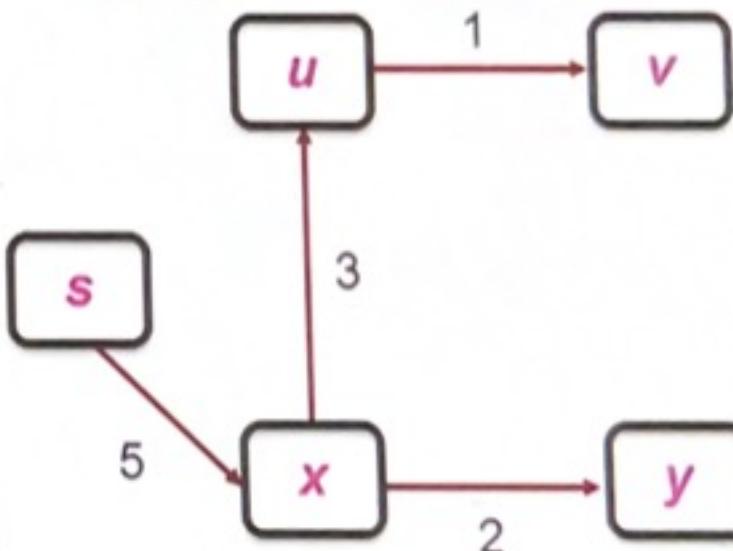
A Toy Example

$S: \{s, x, y, u, v\}$



| d | s | x | u | v | y |
|---|---|---|---|---|---|
| | 0 | 5 | 8 | 9 | 7 |

Shortest paths from s to other vertices



| pi | s | x | u | v | y |
|----|---|---|---|---|---|
| | ? | 5 | x | u | x |

Pseudocode of Dijkstra's Algorithm

Dijkstra_ShortestPath (Graph G, Node source) {

 for each vertex v {

$d[v] = \text{infinity}$;

$\text{pi}[v] = \text{null pointer}$;

$S[v] = 0$;

$\text{if } S[v] = 1 \text{ if } v \text{ is in } S$
 $\text{else } S[v] = 0 \text{ if } v \text{ is not in } S$

 }

$d[\text{source}] = 0$;

 put all vertices in priority queue, Q, in $d[v]$'s increasing order;

 while not Empty(Q) {

$u = \text{ExtractCheapest}(Q)$;

$S[u] = 1$; /* Add u to S */

Pseudocode of Dijkstra's Algorithm

for each vertex v adjacent to u

if ($S[v] \neq 1$ and $d[v] > d[u] + w[u, v]$) {

remove v from Q ;

$d[v] = d[u] + w[u, v]$; ↳ alt way to V
existing calculation
distance

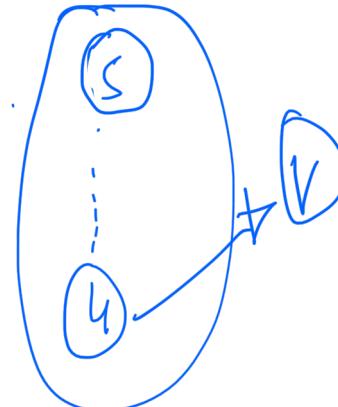
$\pi[v] = u$;

insert v into Q according to its $d[v]$;

}

} // end of while loop

}



Worst case time complexity of Dijkstra's algorithm is $O(|V|^2)$ (analysis not required).

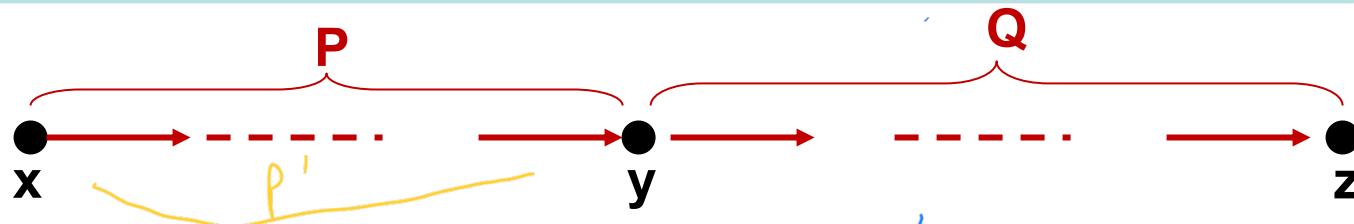
Proof of Correctness

Property of Shortest Path

if A then
 $\bar{xz} = \min$

B
 $\bar{xy} = \min \wedge \bar{yz} = \min$

Lemma 1: In a weighted graph G, suppose that a shortest path from x to z consists of a path P from x to y followed by a path Q from y to z. Then P is a shortest path from x to y and Q is a shortest path from y to z.

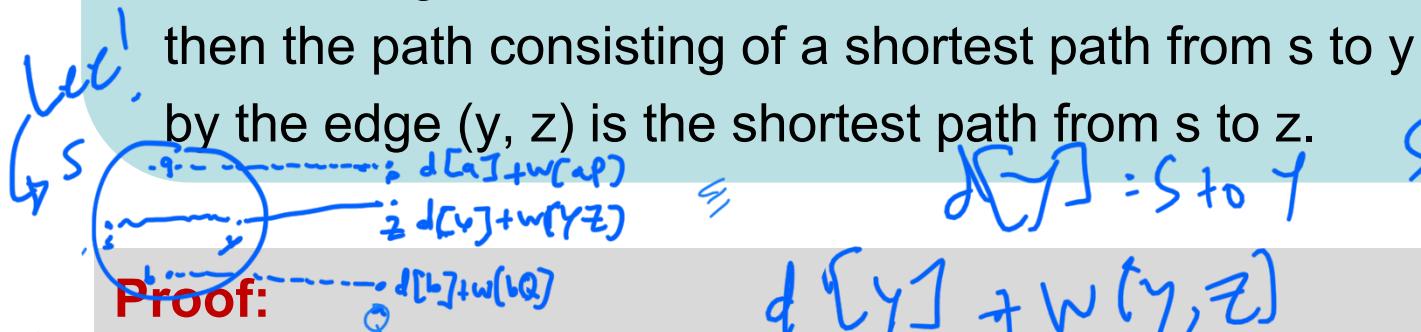
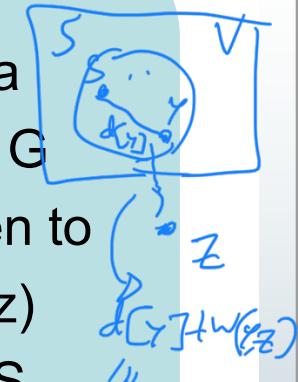


Proof (By Contradiction):

Assume that P is not the shortest path from x to y. Then there will be another path from x to y, P' which is shorter than P. As a result P' followed by Q will be a path **shorter** than P followed by Q. But it was known that P followed by Q is the **shortest** path. Contradiction. Same can be said about Q.

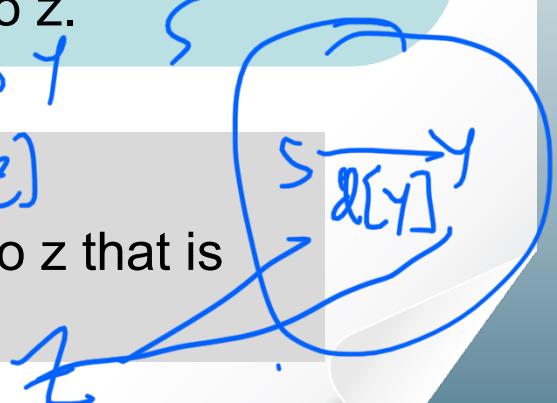
Greedy choice is optimal

Theorem D1: Let $G = (V, E, W)$ be a weighted graph with nonnegative weights. Let S be a subset of V and let s be a member of S . Assume that $d[y]$ is the shortest distance in G from s to y , for each y in S . Let z be the next vertex chosen to go into S . If edge (y, z) is chosen to minimise $d[y] + W(y, z)$ over all edges with one vertex in S and one vertex in $V - S$, then the path consisting of a shortest path from s to y followed by the edge (y, z) is the shortest path from s to z .



Proof:

We will show that there is no other path from s to z that is shorter.



Greedy choice is optimal

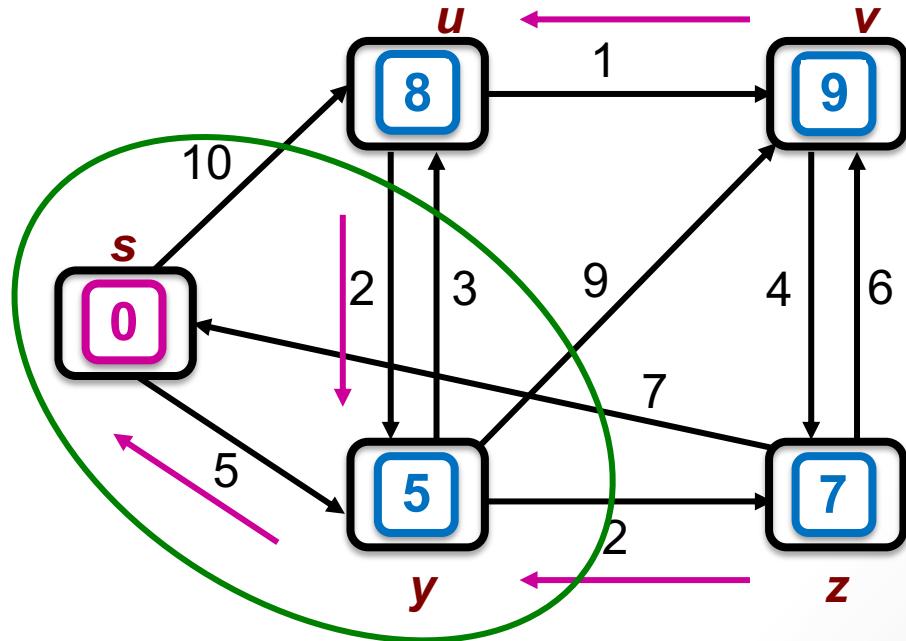
Proof of Theorem D1 (continued)

$P: s \rightarrow y \rightarrow z$ (shortest path for z)

$$W(P) = d[y] + W(y, z)$$

$P': s \rightarrow y \rightarrow u \rightarrow \dots \rightarrow z$
(an alternative shortest path)

$$W(P') = d[y] + W(y, u) + \text{distance from } u \text{ to } z$$



Because $d[y] + W(y, u) \geq d[y] + W(y, z)$,
and distance from u to z is nonnegative,
therefore $W(P) \leq W(P')$.

Edge (y, z) is chosen to minimise $d[y] + W(y, z)$ over all edges with one vertex in S and one vertex in $V - S$

Greedy choice is optimal

Proof of Theorem D1 (continued)

Let P be a shortest path from s to y followed by edge (y, z)

Let $W(P) =$ the distance travelled along P

Let $P' =$ any shortest path different from P , i.e., $P' = s, z_1, \dots, z_k, \dots, z$

Assume that z_k is the first vertex in P' not in set S .

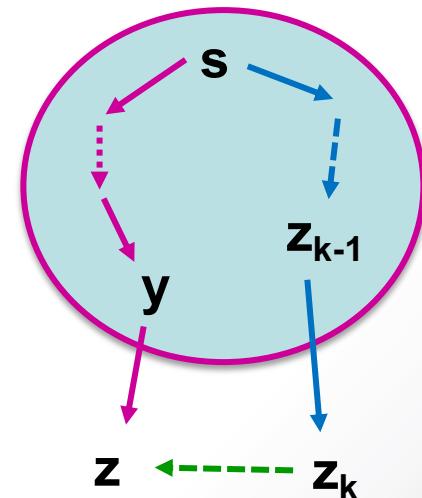
$$W(P) = d[y] + W(y, z)$$

$$W(P') = d[z_{k-1}] + W(z_{k-1}, z_k) + \text{distance from } z_k \text{ to } z$$

Note that: $d[z_{k-1}] + W(z_{k-1}, z_k) \geq d[y] + W(y, z)$

Since **distance from z_k to z** is non-negative,
therefore, $W(P) \leq W(P')$.

$W(\cdot)$ = length



Theorem D2 and Proof

Theorem D2: Given a directed weighted graph G with nonnegative weights and a source vertex s , Dijkstra's algorithm computes the shortest distance from s to each vertex of G that is reachable from s .

Proof (By induction):

We will show by induction that as each vertex v is added into set S , $d[v]$ is the shortest distance from s to v .

Basis:

The algorithm assigns $d[s]$ to zero when the source vertex s is added to S . So $d[s]$ is the shortest distance from s to s when S has the first vertex in it.

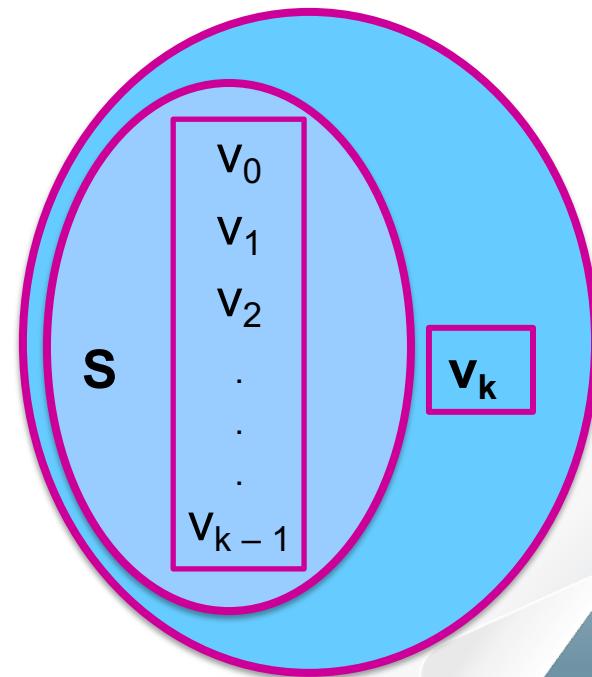
Theorem D2 and Proof (Continued)

Inductive Hypothesis:

Assume the theorem is true when S has k vertices. That is, assume $v_0, v_1, v_2, \dots, v_{k-1}$ are added where $d[v_1], d[v_2] \dots$ are the shortest distances.

When v_k is chosen by Dijkstra's algorithm, it means an edge (v_i, v_k) , where $i \in \{0, 1, 2, \dots, k-1\}$, is chosen to minimise $d[v_i] + W(v_i, v_k)$ among all edges with one vertex in S and one vertex not in S .

By Theorem D1, $d[v_k]$ is the shortest distance from source to v_k . So the theorem is true when S has $k+1$ vertices.



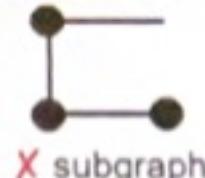
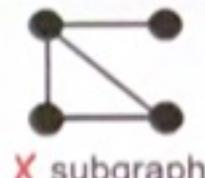
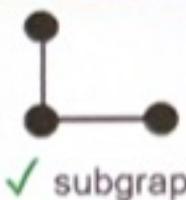
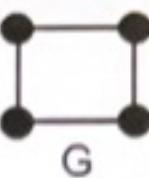


Wk 5

Minimum Spanning Tree

Minimum Spanning Tree

Definition of Subgraph



A subgraph of a graph $G = (V, E)$ is a graph $G' = (V', E')$ such that $V' \subseteq V$ and $E' \subseteq E$ and $E' \subseteq V' \times V'$

→ each edge must connect to vertex

Definition of Spanning Tree

A connected, acyclic subgraph containing all the vertices of a graph.

Definition of Minimum Spanning Tree

not unique

A minimum-weight spanning tree in a weighted graph.

Minimum Spanning Tree

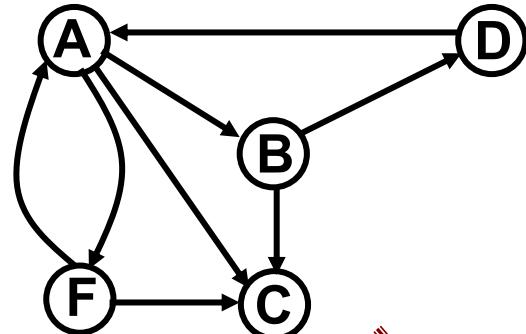
Definition of **Spanning Tree**

A connected, acyclic subgraph containing all the vertices of a graph.

Definition of **Minimum Spanning Tree**

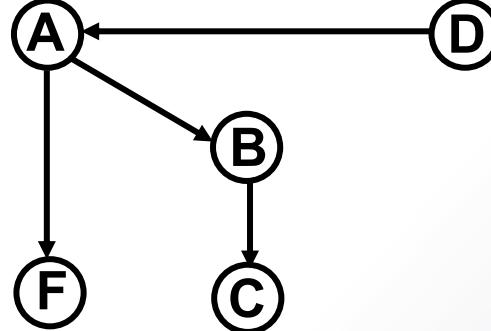
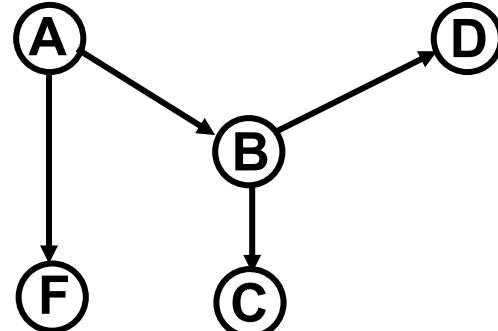
A minimum-weight spanning tree in a weighted graph.

Spanning Tree

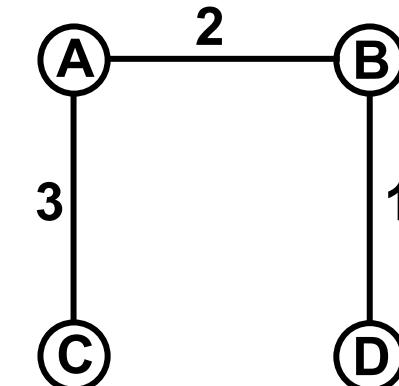
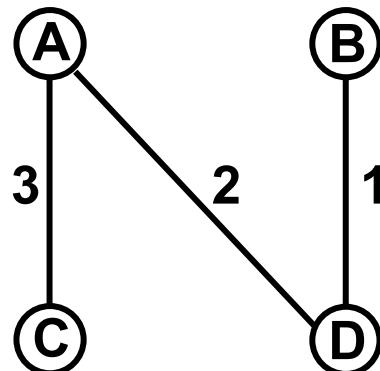
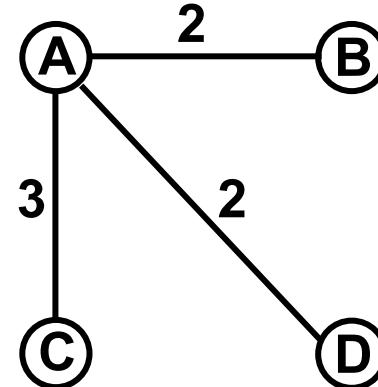
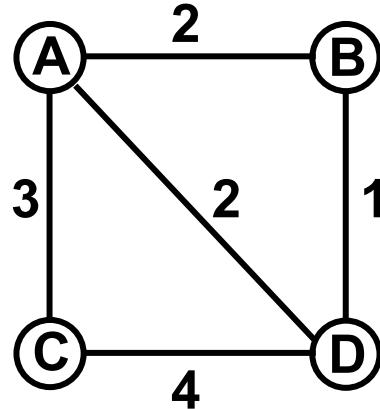


Spanning Tree

Another Spanning Tree



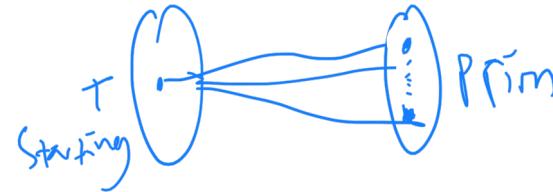
Minimum Spanning Tree



Main Idea of Prim's Algorithm

Prim's Algorithm

- It works on undirected graph.
- It builds upon a single partial minimum spanning tree, at each step adding an edge connecting the vertex nearest to but not already in the current partial minimum spanning tree.
- At first a vertex is chosen, this vertex will be the first node in T .
- Set P is initialised: $P = \text{set of vertices not in tree } T \text{ but are adjacent to some vertices in } T$.



Main Idea of Prim's Algorithm

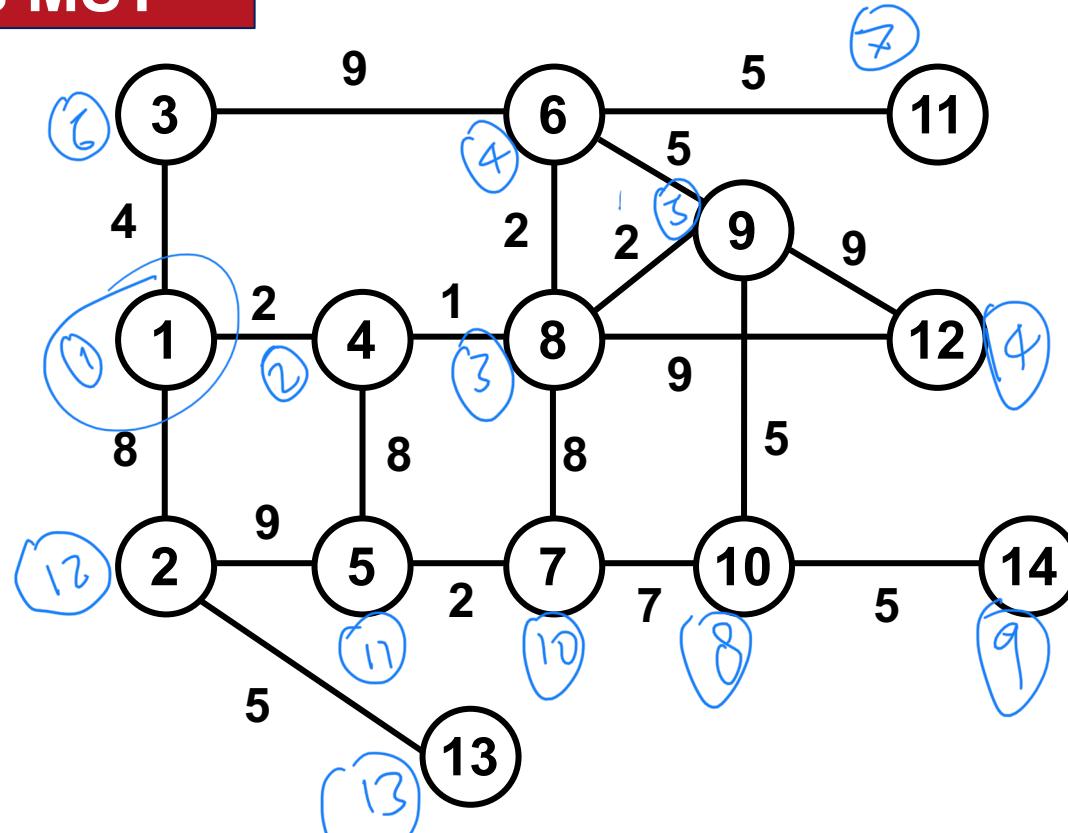
Prim's Algorithm (Cont.)



- In every iteration in the Prim's Algorithm, a new vertex u from set P will be connected to the tree T . The vertex u will be deleted from the set P . The vertices adjacent to u and not already in P will be added to P .
- When all vertices are connected into T , P will be empty. This means the end of the algorithm.
- The new vertex in every iteration will be chosen by using **greedy** method, i.e. among all vertices in P which are connected to some vertices already inserted in the tree T but themselves are not in T , we choose one with the minimum cost.

An Example of Prim's Algorithm

Prim's MST



Black vertices: unseen vertices

Pink vertices: tree vertices

Blue vertices: fringe vertices

3 subsets of vertices

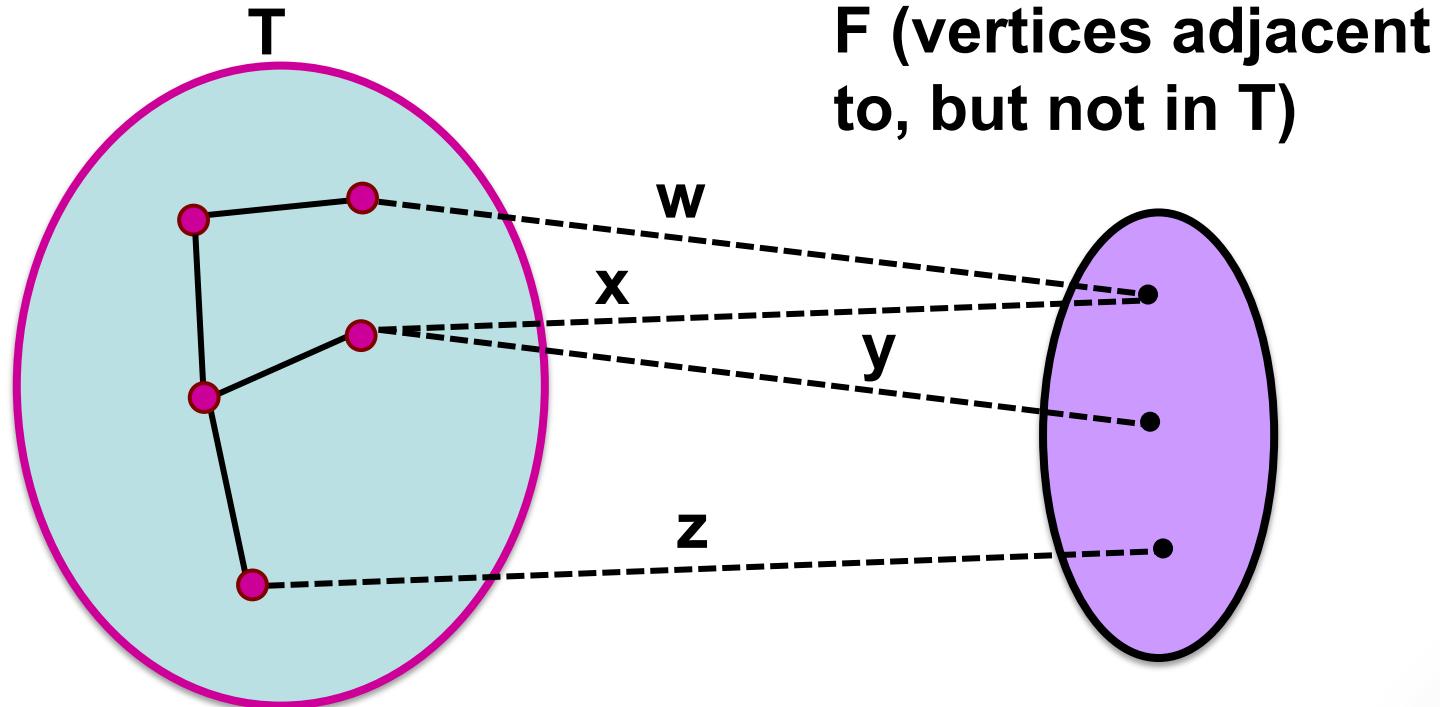
Prim's Algorithm classifies vertices into three disjoint categories:

- **Tree vertices** - in the tree being constructed so far
- **Fringe vertices** - not in the tree but adjacent to some vertices in the tree
- Unseen vertices - all others

Greedy choice of Prim's Algo

- Key step in the algorithm is the selection of a vertex from the fringe (which, of course, depends on the weights on incident edges).
- Prim's Algorithm always chooses a minimum weight edge from **tree** vertex to **fringe** vertex.

Main Idea of Prim's Algorithm



Choose $\min(w, x, y, z)$

Pseudocode of Prim's Algo

```
primMST(G, s, n) // outline of Prim's algorithm
{
    Initialise all vertices as unseen.
    Reclassify s as tree vertex.
    Reclassify all vertices adjacent to s as fringe.
    While (there are fringe vertices)
    {
        Select an edge of minimum weight between a tree vertex t and a fringe vertex v;
        Reclassify v as tree; add edge tv to the tree;
        Reclassify all unseen vertices adjacent to v as fringe.
    }
}
```

Implementing Prim's Algo

Data Structures Used:

- Array **d**: distance of a fringe vertex from the tree
- Array **pi**: vertex connecting a fringe vertex to a tree vertex
- Array **S**: whether a vertex is in the minimum spanning tree being built
- Priority queue **pq**: queue of fringe vertices in the order of the distances from the tree

At the end of the algorithm, array **pi** has the minimum spanning tree.

Implementing Prim's Algo

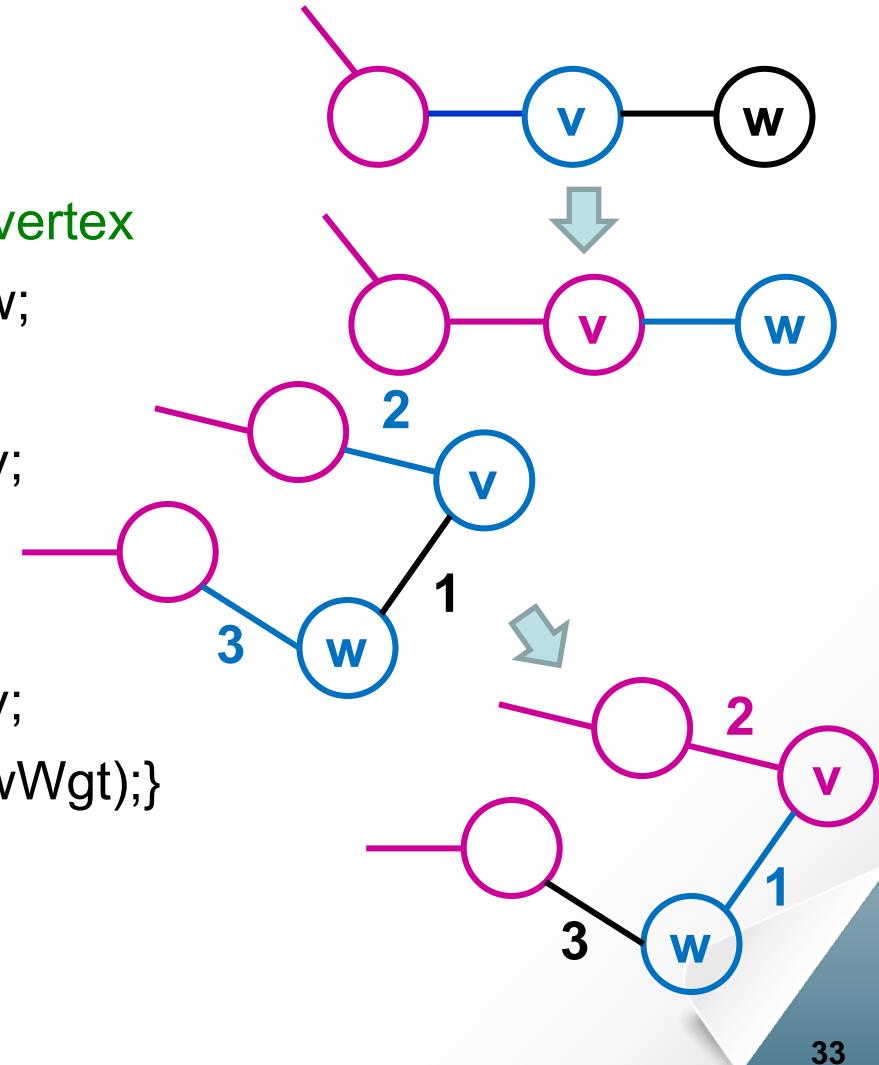
```
primMST(G, s, n) {  
    initialise priority queue pq as empty;  
    for each vertex v {  
        d[v] = infinity; S[v] = 0;  
        pi[v] = null pointer; }  
    d[s] = 0; S[s] = 1;  
    insert(pq, s, 0);  
    while (pq is not empty) {  
        u = getMin(pq); deleteMin(pq);  
        S[u] = 1;  
        updateFringe(pq, G, u); }  
}
```

Update Fringe Set of Vertices

```

III
updateFringe(pq, G, v) {
    for all vertices w adjacent to v {
        if (S[w] != 1) { //if w is not a tree vertex
            newWgt = weight of edge vw;
            if (d[w] == infinity) {
                d[w] = newWgt; pi[w] = v;
                insert(pq, w, newWgt);
            } else if (newWgt < d[w]) {
                d[w] = newWgt; pi[w] = v;
                decreaseKey(pq, w, newWgt);}
            } // if w is not a tree vertex
    } // for all vertices
}

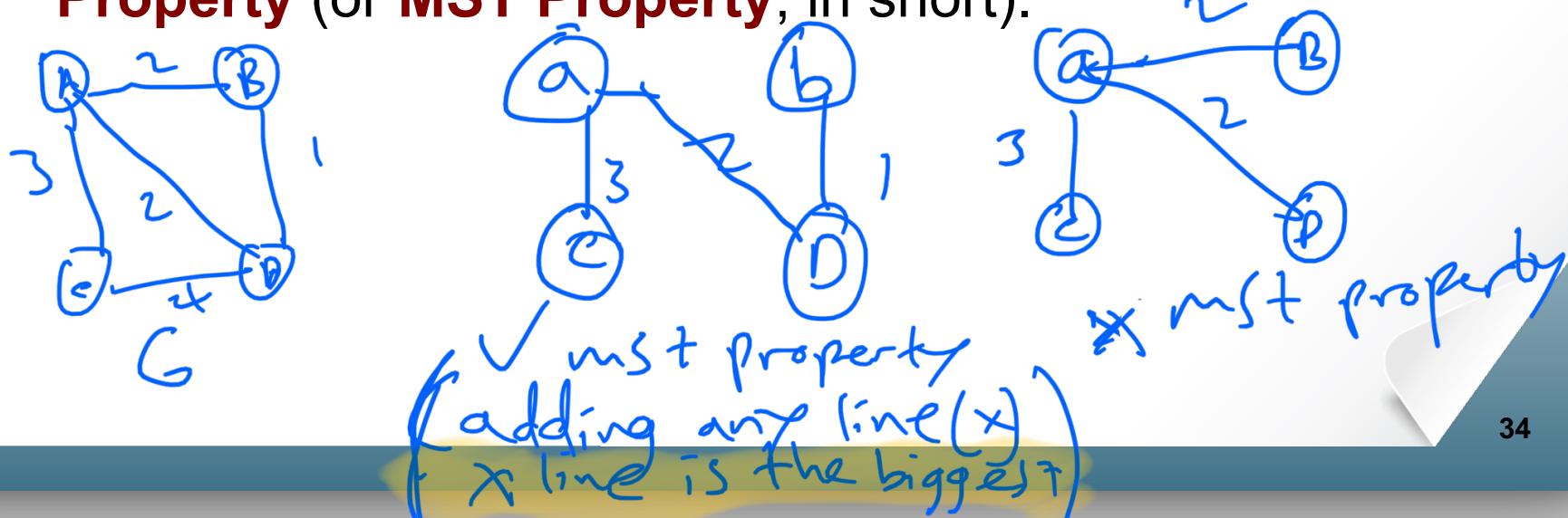
```



MST Property

Minimum Spanning Tree Property

Let T be a spanning tree of G , where $G = (V, E, W)$ is a connected, weighted graph. Suppose that for every edge (u, v) of G that is not in T , if (u, v) is added to T it creates a cycle such that (u, v) is a maximum-weight edge on that cycle. Then T has the **Minimum Spanning Tree Property** (or **MST Property**, in short).



Lemma 1 and Proof

Lemma 1: In a connected weighted graph $G = (V, E, W)$, if T_1 and T_2 are two spanning trees that have the MST property, then they have the same total weight.

Proof by induction on k , the number of edges in T_1 but not T_2 (there are also k edges in T_2 but not in T_1).

Basis:

$k = 0$; i.e. $T_1 = T_2$. Therefore, they have the same weight.

Proof of Lemma 1 (continued)

Inductive hypothesis: For $k > 0$, assume the lemma holds when there are j differing edges where $0 \leq j < k$.

Let uv be the minimum weight edge among the differing edges (assume uv is in T_2 but not T_1).

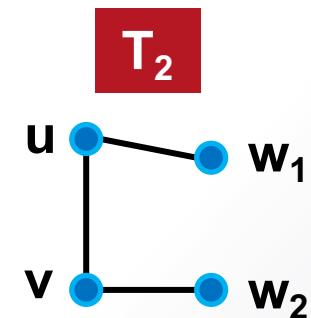
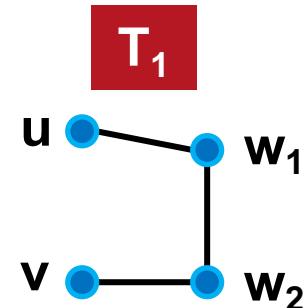
Look at unique path in T_1 from u to v .

Suppose it is made up of w_0, w_1, \dots, w_p where $w_0 = u, \dots, w_p = v$.

This path must contain some edge different from T_2 's.

Let $w_i w_{i+1}$ be this differing edge.

By MST property of T_1 , $w_i w_{i+1}$ cannot be $> uv$'s weight.



Proof of Lemma 1 (continued)

But since uv was chosen to be the minimum weight among differing edges, $w_i w_{i+1}$ cannot have weight less than uv .

Therefore, $W(w_i w_{i+1}) = W(uv)$.

Add uv to T_1 (creating a cycle). Remove $w_i w_{i+1}$ leaving tree T'_1 (which has the same weight as T_1).

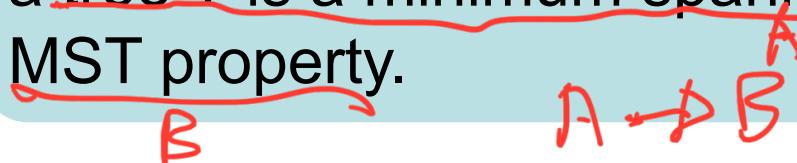
But T'_1 and T_2 differ only on $k - 1$ edges.

$$\begin{aligned} W(T'_1) &= W(T_2) \\ W(T_1) &= W(T'_1) \end{aligned}$$

So by inductive hypothesis, T'_1 and T_2 have the same total weight. Therefore, T_1 and T_2 have same weight.

Theorem 1 and Proof

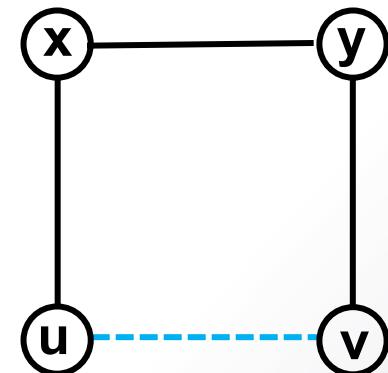
Theorem 1: In a connected weighted graph $G = (V, E, W)$, a tree T is a minimum spanning tree if and only if T has the MST property.



Proof (Only if): Assume T is an MST for graph G .

Suppose T does not satisfy the MST property, i.e. there is some edge uv that is not in T such that adding uv creates a cycle, in which some other edge xy has weight $W(xy) > W(uv)$.

Then, by removing xy and adding uv , we create a new spanning tree whose total weight is $< W(T)$; This contradicts the assumption that T is an MST.



Proof of Theorem 1 (continued)

Theorem 1: In a connected weighted graph $G = (V, E, W)$, a tree T is a minimum spanning tree if and only if T has the MST property.

Proof (Only if): Assume T is an MST for graph G .

(Cont.)

(If) Assume T has MST property.

If T_{\min} is an MST, then T_{\min} has MST property by the first half of the proof.

By Lemma 1, $W(T) = W(T_{\min})$, so T is also an MST.

Prim's Algorithm is Optimal

Lemma 2: Let $G = (V, E, W)$ be a connected weighted graph; Let T_k be the tree with k vertices constructed by Prim's Algorithm, for $k = 1, 2, \dots, n$; and let G_k be the subgraph induced by the vertices of T_k . Then T_k has the MST property in G_k . (**Proof is not required**)

Theorem 2: Prim's Algorithm outputs a minimum spanning tree.

Proof:

- From Lemma 2, T_n has the MST property.
- By Theorem 1, T_n is a minimum spanning tree.

Priority Queue for MST (Optional)

- Inserted by order of priority (not chronological, as in 'normal' queues – FIFO)
- Elements to be inserted have a 'key' - contains the priority; element with highest priority will be selected first. [priority can be largest value (e.g. if we're computing max profit) or smallest value (e.g. if we're interested in min cost)]
- Think of pq as a sequence of pairs: $(id_1, w_1), (id_2, w_2), \dots, (id_k, w_k)$. The order is in increasing w_i and id is a unique identifier for an element

Methods of Priority Queue (Optional)

The Priority Queue consists of:

Create: Constructor to set up PQ

isEmpty; getMin; getPriority: Access functions

insert; deleteMin; decreaseKey: Manipulation procedures

Insert(pq, id, w): Inserts (id, w) into an existing pq - position depends on w

decreaseKey(pq, id, neww): Rearranges pq based on new wt of element id

getMin(pq): Returns id_1 ;

getPriority(pq): Returns weight of min element

Summary

- Greedy algorithm is a general strategy to solve optimization problems
- Dijkstra's algorithm finds single-source shortest paths in a weighted graph of nonnegative edge weights
- Prim's algorithm finds the minimum spanning trees in weighted graphs
- Both are greedy algorithms, and use priority queue



week 6

SC2001/CE2101/ CZ2101: Algorithm Design and Analysis

**Union-Find Programs and
Kruskal's Algorithm**

Instructor: Asst. Prof. LIN Shang-Wei

Courtesy of Dr. Ke Yiping, Kelly's slides

Contents

- Dynamic Equivalence Relations
 - Three basic operations
- Union-Find Programs
 - Improve algorithms step by step
 - QuickFind
 - QuickUnion
 - Weighted QuickUnion with Path Compression (WQUPC)
- Kruskal's Algorithm
 - Pseudocode
 - Correctness
 - Complexity

Learning Objectives

At the end of this lecture, students should be able to:

- Understand the concept of dynamic equivalence relations
- Understand and analyze various union-find programs
- Solve minimum spanning tree (MST) problem using Kruskal's algorithm
- Prove the correctness of Kruskal's algorithm



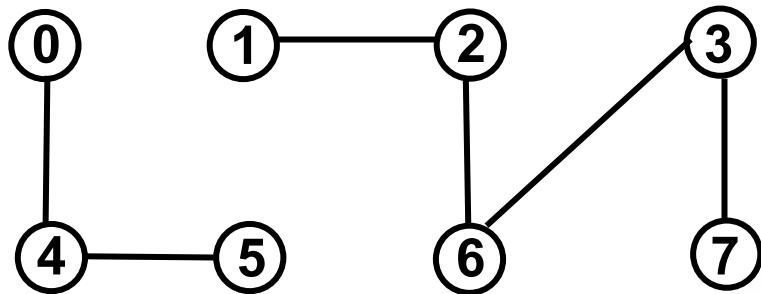
Dynamic Equivalence Relations

Dynamic Equivalence Relations

- An equivalence relation \mathbf{R} on a set \mathbf{S} is a binary relation, such that for every a , b , and c in \mathbf{S} :
 - Reflexivity: $R(a, a)$
 - Symmetry: if $R(a, b)$, then $R(b, a)$
 - Transitivity: if $R(a, b)$ and $R(b, c)$, then $R(a, c)$
- Dynamic equivalence relation: the equivalence relation will change with a number of operations
- Given a set of N objects in \mathbf{S} , define three operations:
 - Initialize the objects
 - Connect two objects: add them into relation \mathbf{R}
 - Is there a path connecting two objects?

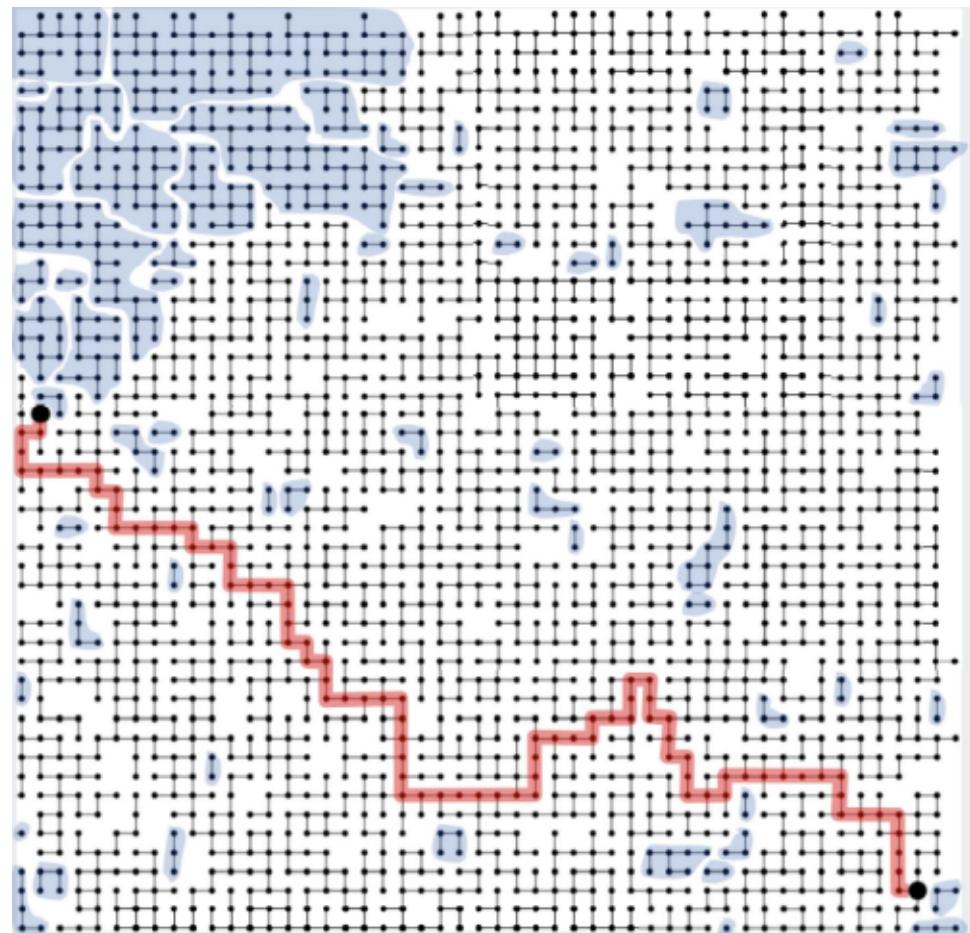
Running Example

- Initialize 8 objects: 0, ..., 7
- Connect 4 and 5
- Connect 1 and 2
- Connect 6 and 3
- Connect 0 and 4
- Are 1 and 7 connected? ✗
- Are 0 and 5 connected? ✓
- Connect 2 and 6
- Connect 3 and 7
- Are 1 and 7 connected? ✓



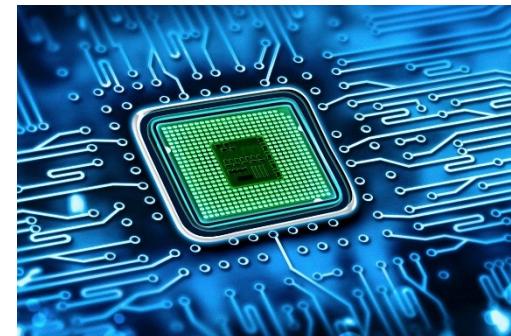
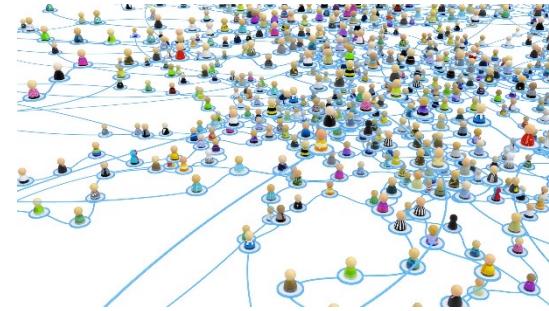
A Larger Example

- Are the two big dots connected?
- Yes
- 63 components



Applications

- Friends in a social network
- Webpages on the Internet
- Statements in a Python program
- Computers in a computer network
- Transistors in a computer chip



Challenges

- Dynamics:
 - Operations may be intermixed.
- Number of operations M can be huge.
- Number of objects N can be huge.

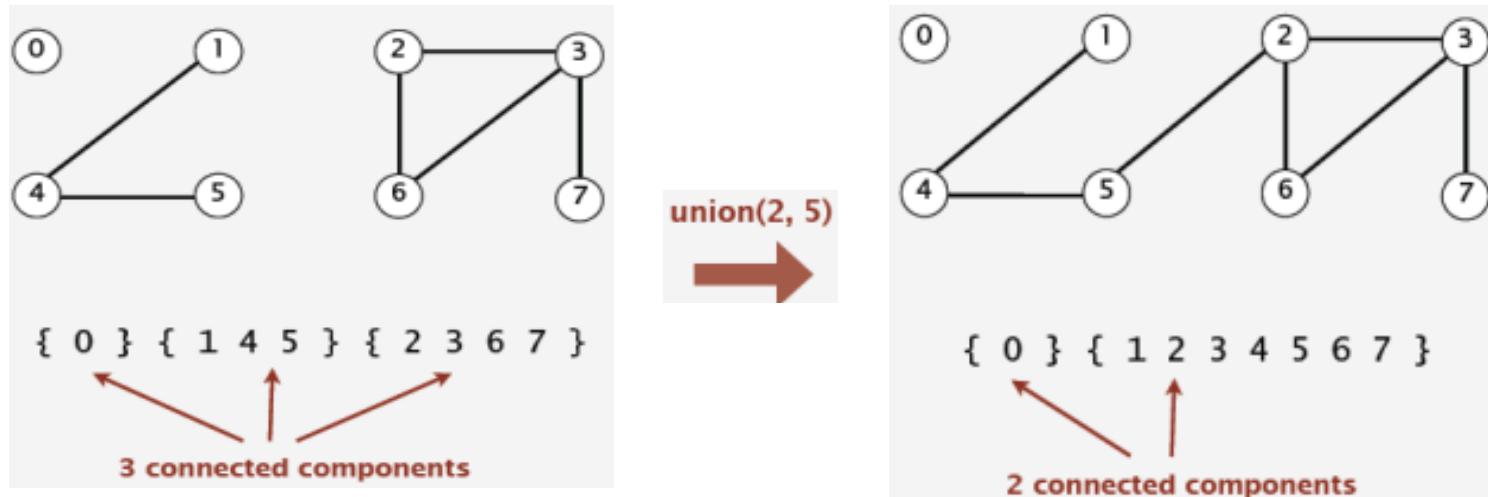
- Goal: design efficient data structure



Union-Find Programs

Union-Find

- Equivalence class: connected components
- Three operations to implement the dynamic equivalence relation
 - $\text{find}(p)$: which component does an object p belong to?
 - $\text{connected}(p, q)$: are two objects p and q connected?
 - $\text{union}(p, q)$: connect p and q (compute the union of their components)



Union-Find APIs

- A class defined for union-find (in Java)

```
public class UF
{
    UF(int N)                                initialize union-find data structure
                                                with N singleton objects (0 to N - 1)
    void union(int p, int q)                  add connection between p and q
    int find(int p)                          component identifier for p (0 to N - 1)
    boolean connected(int p, int q)          are p and q in the same component?
}
```

```
public boolean connected(int p, int q)
{   return find(p) == find(q); }
```

1-line implementation of connected()

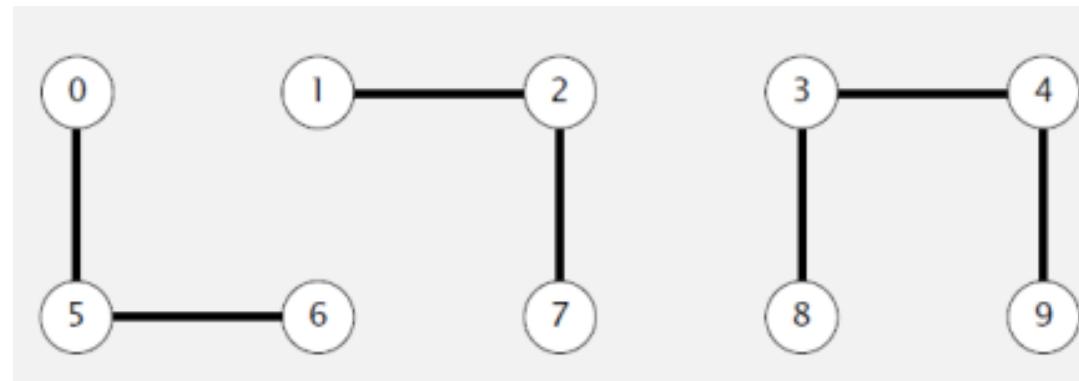


QuickFind Algorithm

QuickFind [Eager Approach]

■ Data structure:

- Integer array $\text{id}[]$ of length N
- Interpretation: $\text{id}[p]$ is the ID of the component that p belongs to



| | | | | | | | | | | |
|---------------|---|---|---|---|---|---|---|---|---|---|
| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| $\text{id}[]$ | 0 | 1 | 1 | 8 | 8 | 0 | 0 | 1 | 8 | 8 |

QuickFind Functions

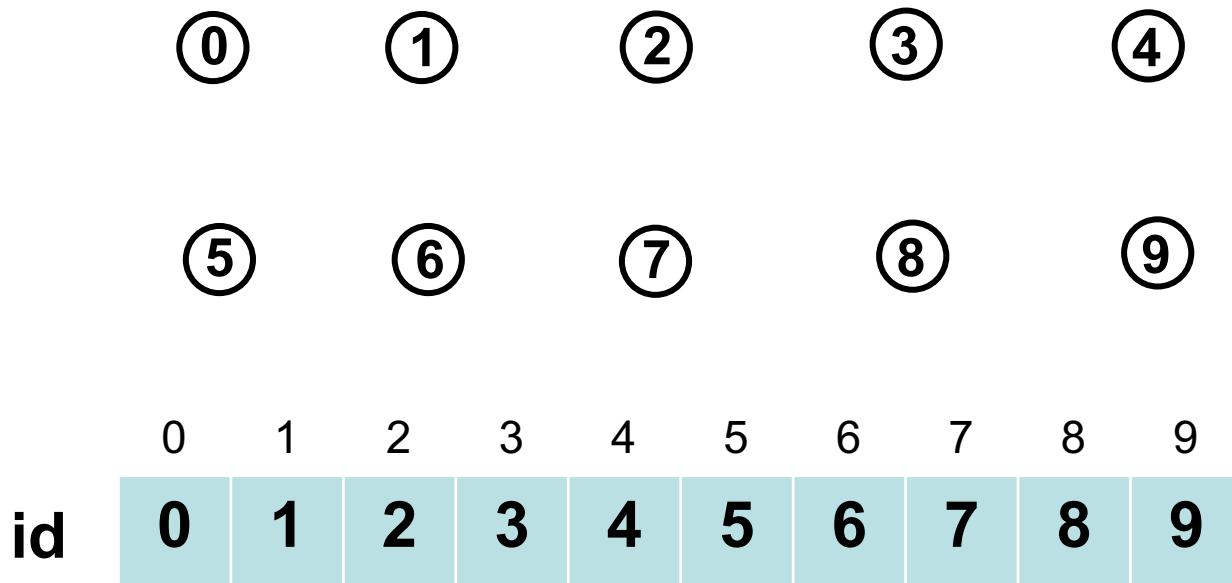
- $\text{find}(p)$: what is the id of p ?
 - $\text{id}[6] = 0$; $\text{id}[1] = 1$
- $\text{connected}(p, q)$: do p and q have the same id?
 - 6 and 1 are not connected
- $\text{union}(p, q)$: change all entries whose id equals $\text{id}[p]$ to $\text{id}[q]$
 - $\text{union}(6, 1)$

| | | | | | | | | | | |
|---------------|---|---|---|---|---|---|---|---|---|---|
| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| $\text{id}[]$ | 0 | 1 | 1 | 8 | 8 | 0 | 0 | 1 | 8 | 8 |

| | | | | | | | | | | |
|---------------|---|---|---|---|---|---|---|---|---|---|
| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| $\text{id}[]$ | 1 | 1 | 1 | 8 | 8 | 1 | 1 | 1 | 8 | 8 |

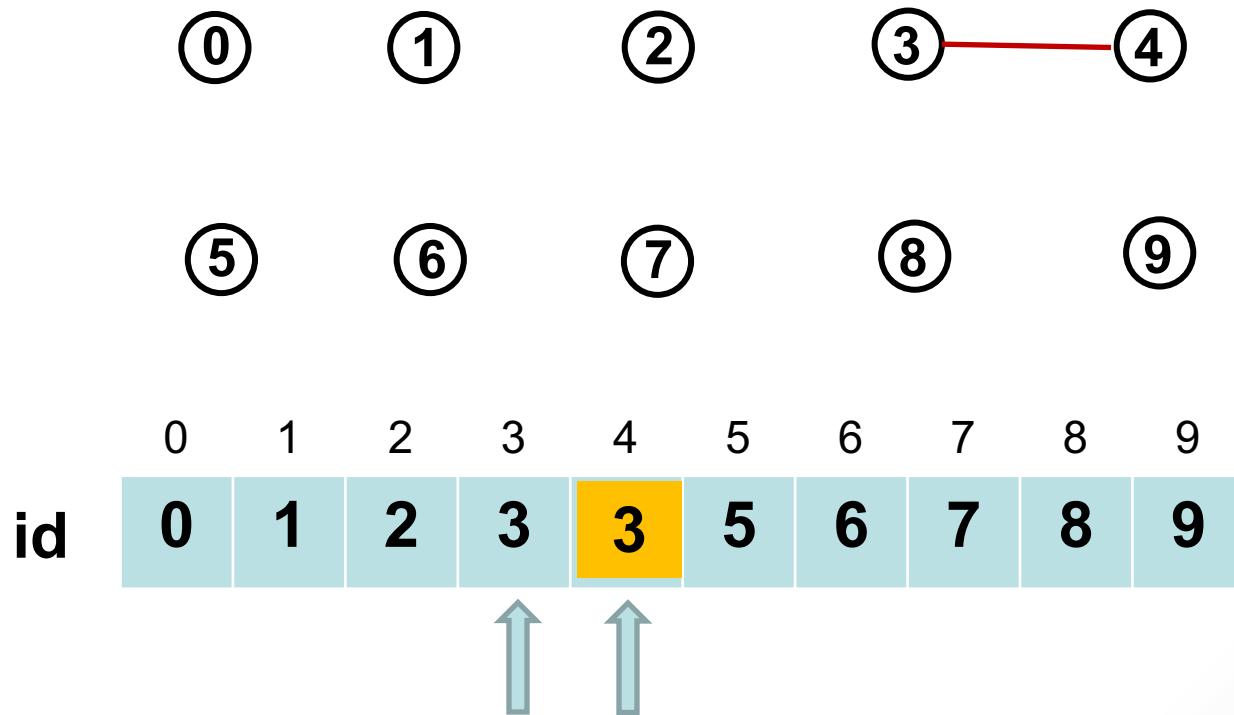
problem: many values can change

QuickFind Demo



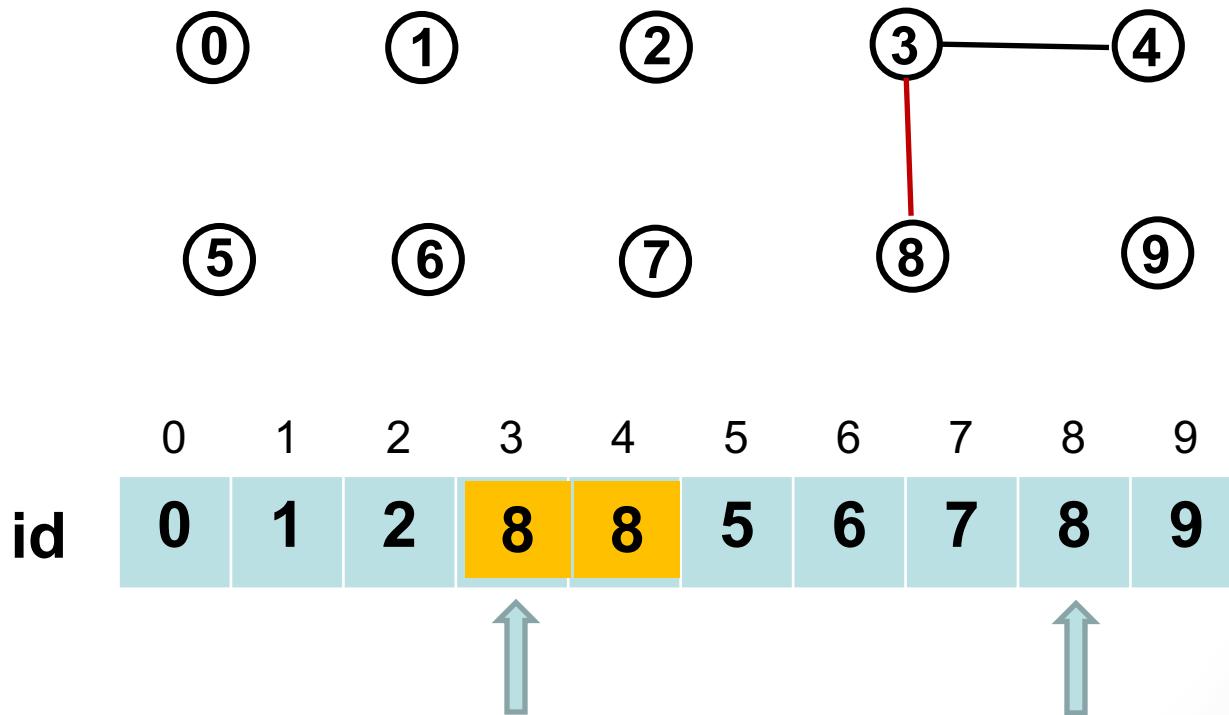
QuickFind Demo

union(4, 3)



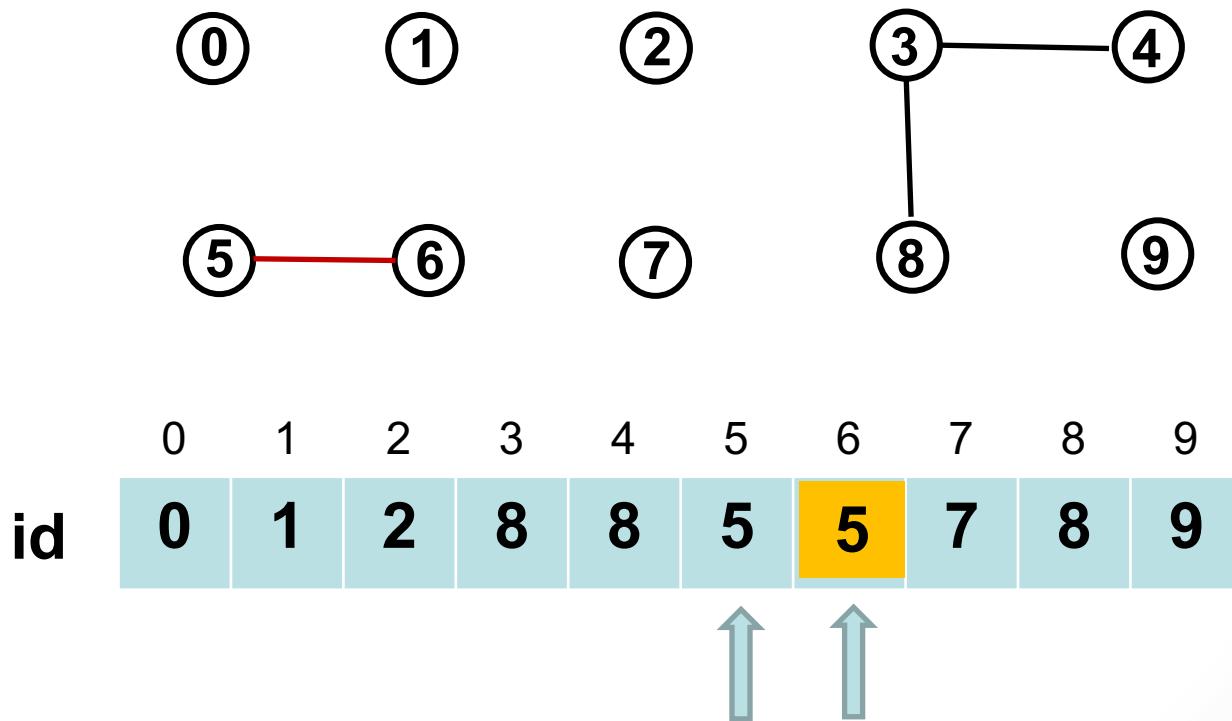
QuickFind Demo

union(3, 8)



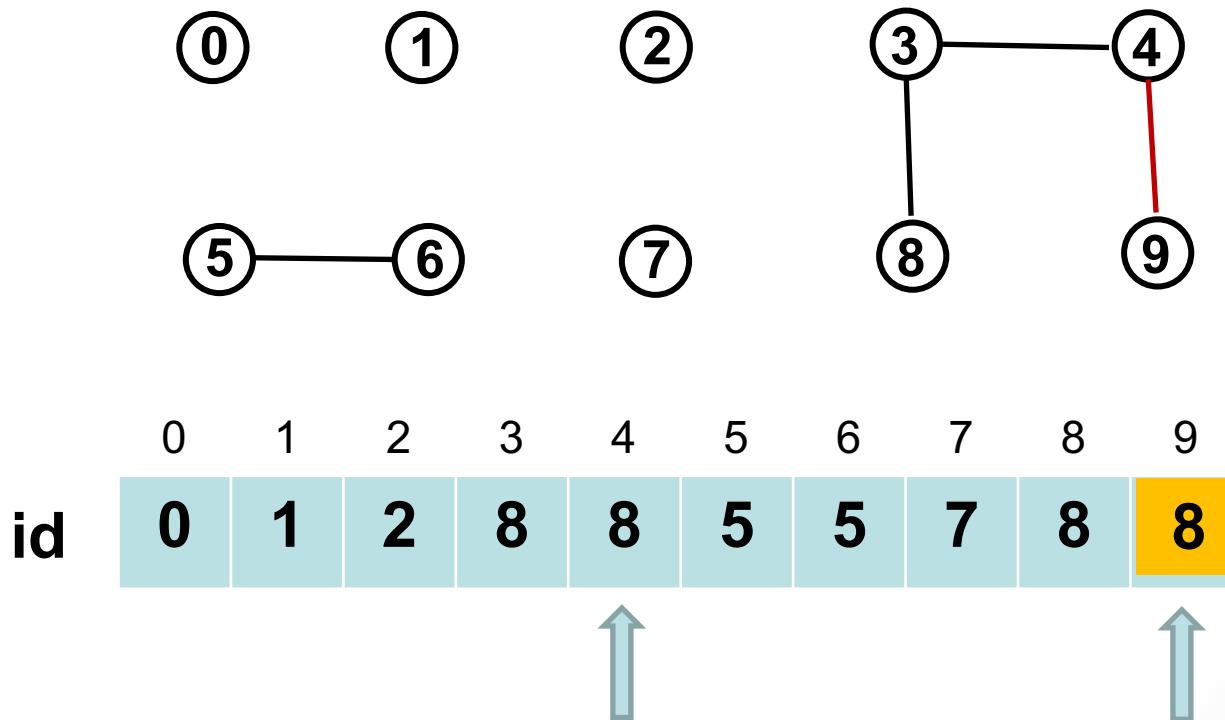
QuickFind Demo

union(6, 5)



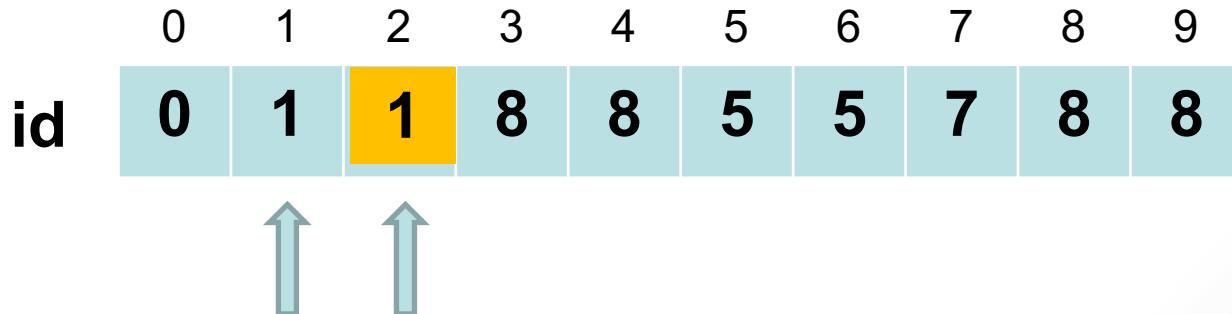
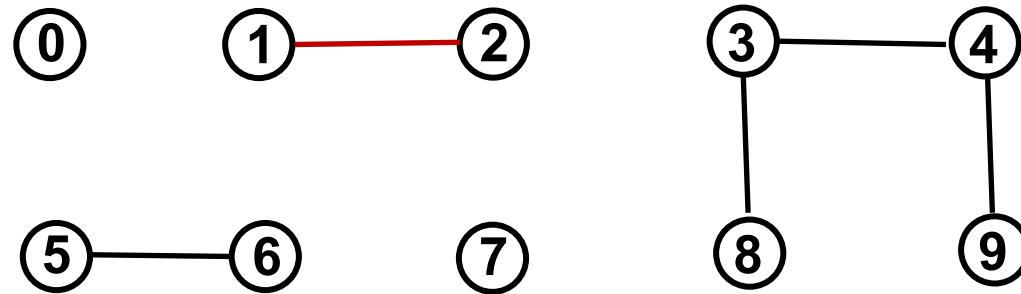
QuickFind Demo

union(9, 4)



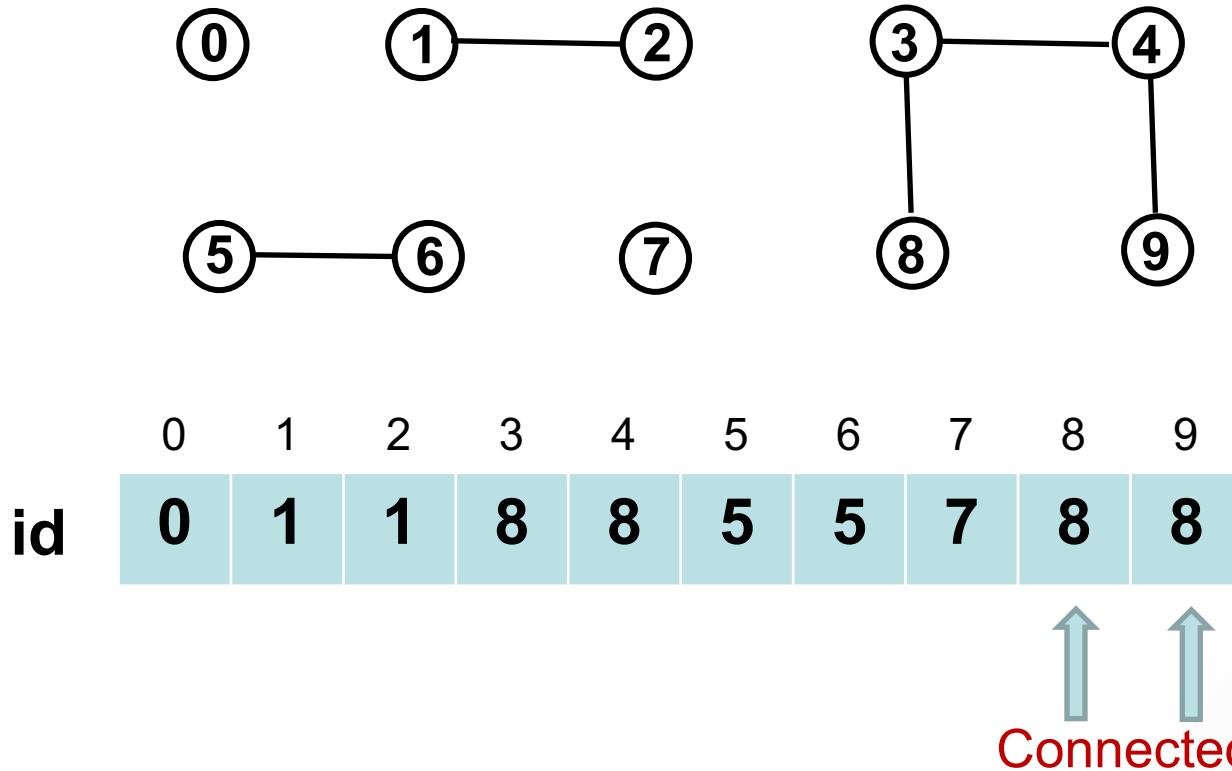
QuickFind Demo

union(2, 1)



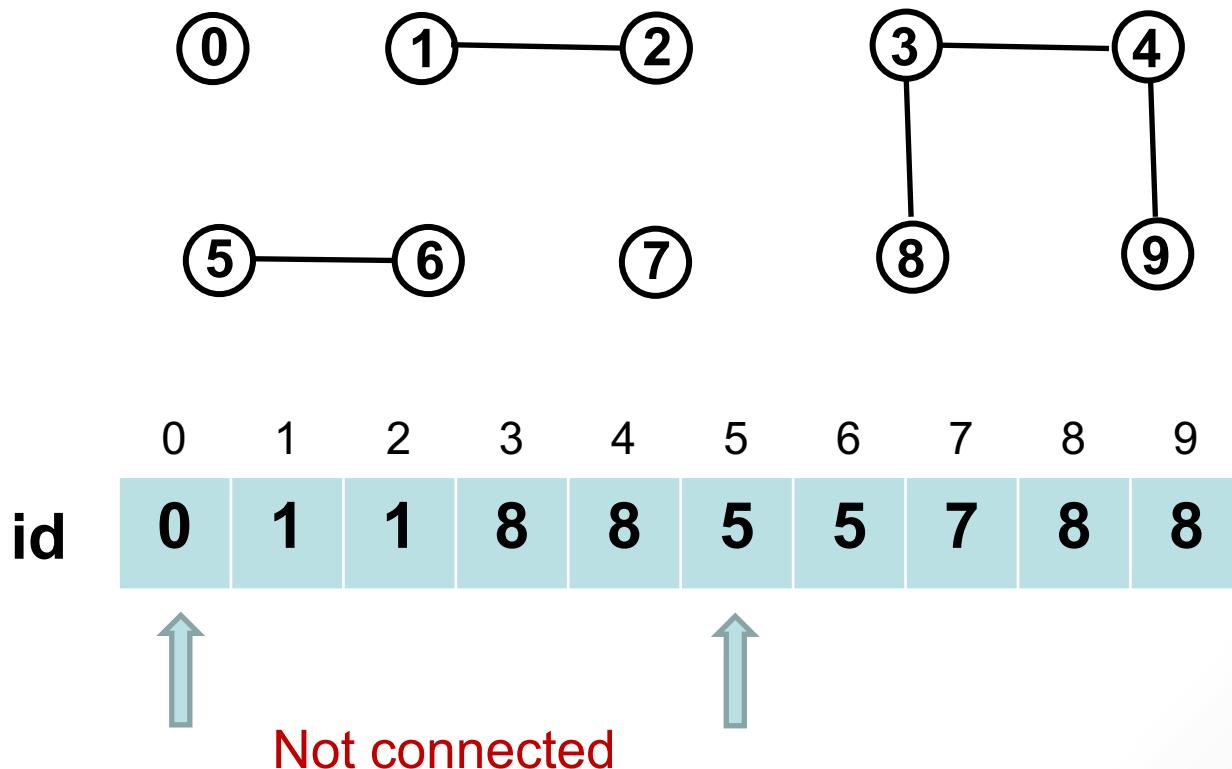
QuickFind Demo

connected(8, 9)



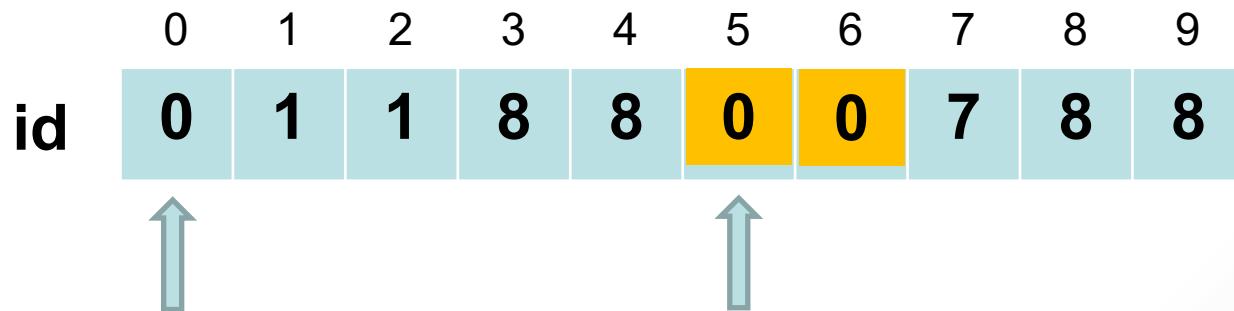
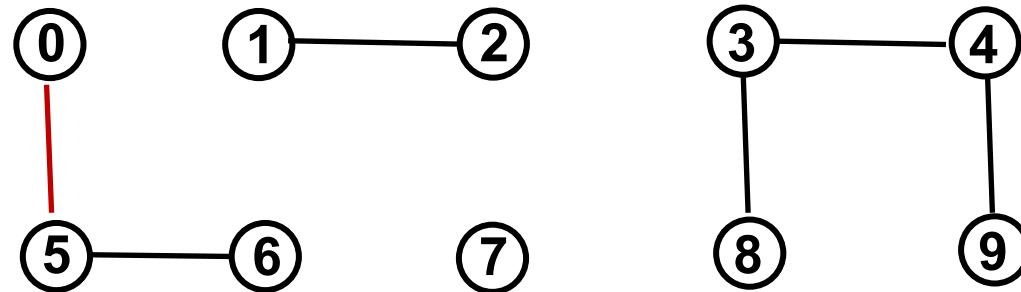
QuickFind Demo

connected(5, 0)



QuickFind Demo

union(5, 0)



QuickFind Implementation (Java)

```
public class QuickFindUF
{
    private int[] id;

    public QuickFindUF(int N)
    {
        id = new int[N];
        for (int i = 0; i < N; i++)
            id[i] = i;
    }

    public int find(int p)
    {
        return id[p];
    }

    public void union(int p, int q)
    {
        int pid = id[p];
        int qid = id[q];
        for (int i = 0; i < id.length; i++) ←
            if (id[i] == pid) id[i] = qid;
    }
}
```

set id of each object to itself
(N array accesses)

return the id of p
(1 array access)

change all entries with $\text{id}[p]$ to $\text{id}[q]$
(at most $2N + 2$ array accesses)

QuickFind - Time Complexity

- Measured by: number of array accesses (read or write)

| Algorithm | Initialization | union | find | connected |
|-----------|----------------|--------|--------|-----------|
| QuickFind | $O(N)$ | $O(N)$ | $O(1)$ | $O(1)$ |

- union() is too expensive:
 - For N union operations on N objects, it takes $O(N^2)$ time.

Quadratic is not scalable

- Rough standard (in 2020)
 - 10^{11} operations per second
 - 10^{11} words of main memory (400GB)
 - Access all words in ~1 second
- Huge problem for QuickFind
 - 10^{11} union operations on 10^{11} objects
 - QuickFind takes more than 10^{22} operations.
 - 3000+ years to complete!
- Quadratic algorithms don't scale with technology
 - We need better algorithms!

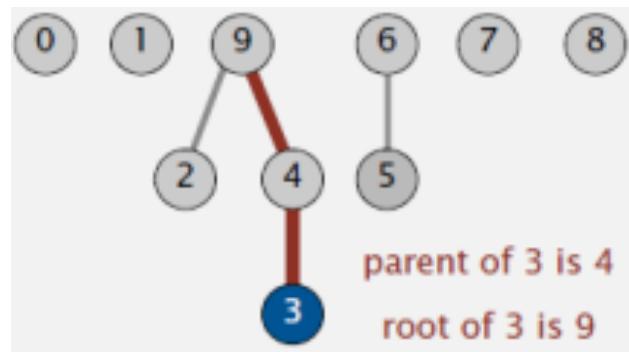


QuickUnion Algorithm

QuickUnion [Lazy Approach]

■ Data structure:

- Integer array $\text{id}[]$ of length N
- Interpretation: $\text{id}[p]$ is the parent of p
- Root of p is $\text{id}[\text{id}[\text{id}[\dots\text{id}[i]\dots]]]$: component ID
 - Keep id-ing until the value doesn't change (cycles?)

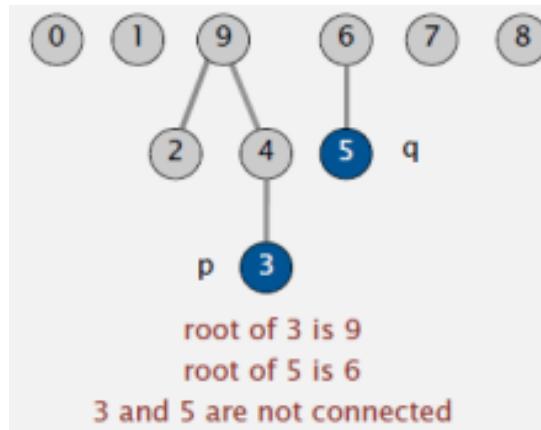


| | | | | | | | | | | |
|-------------|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | |
| id[] | 0 | 1 | 9 | 4 | 9 | 6 | 6 | 7 | 8 | 9 |

QuickUnion Functions

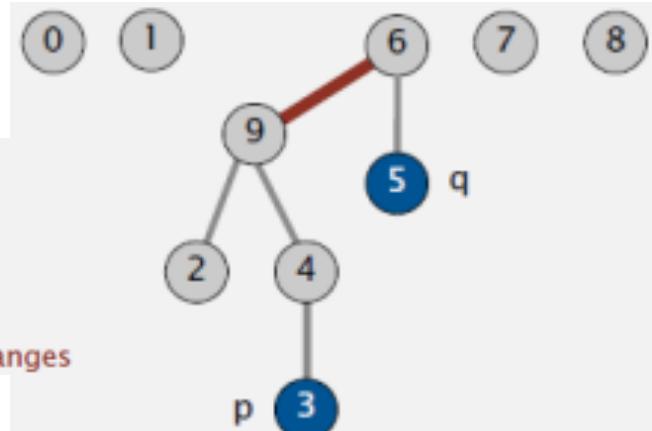
- $\text{find}(p)$: what is the **root** of p ?
- $\text{connected}(p, q)$: do p and q have the same **root**?
- $\text{union}(p, q)$: set the id of p 's **root** to the id of q 's **root**.

| | | | | | | | | | | | | |
|---------------|---|---|---|---|---|---|---|---|---|---|---|---|
| $\text{id}[]$ | 0 | 1 | 2 | 3 | 4 | 9 | 6 | 6 | 6 | 7 | 8 | 9 |
| | 0 | 1 | 9 | 4 | 9 | 6 | 6 | 6 | 6 | 7 | 8 | 9 |



| | | | | | | | | | | | | |
|---------------|---|---|---|---|---|---|---|---|---|---|---|---|
| $\text{id}[]$ | 0 | 1 | 2 | 3 | 4 | 9 | 6 | 6 | 7 | 8 | 9 | |
| | 0 | 1 | 9 | 4 | 9 | 6 | 6 | 6 | 6 | 7 | 8 | 9 |

↑
only one value changes



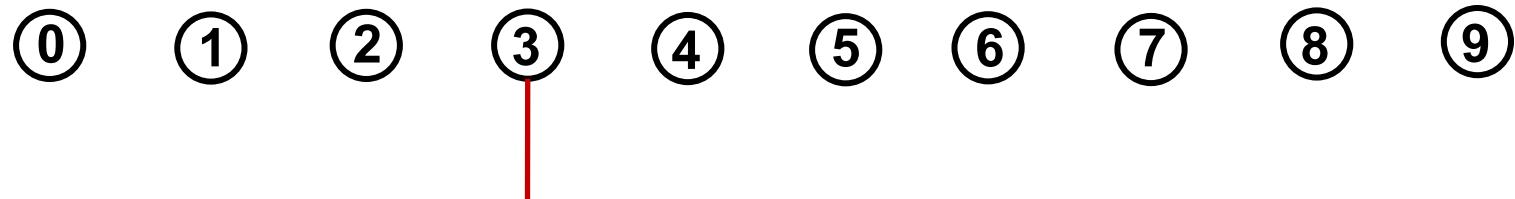
QuickUnion Demo

① ② ③ ④ ⑤ ⑥ ⑦ ⑧ ⑨

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|-----------|---|---|---|---|---|---|---|---|---|---|
| id | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

QuickUnion Demo

union(4, 3)



| id | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|----|---|---|---|---|---|---|---|---|---|---|
| | 0 | 1 | 2 | 3 | 3 | 5 | 6 | 7 | 8 | 9 |

QuickUnion Demo

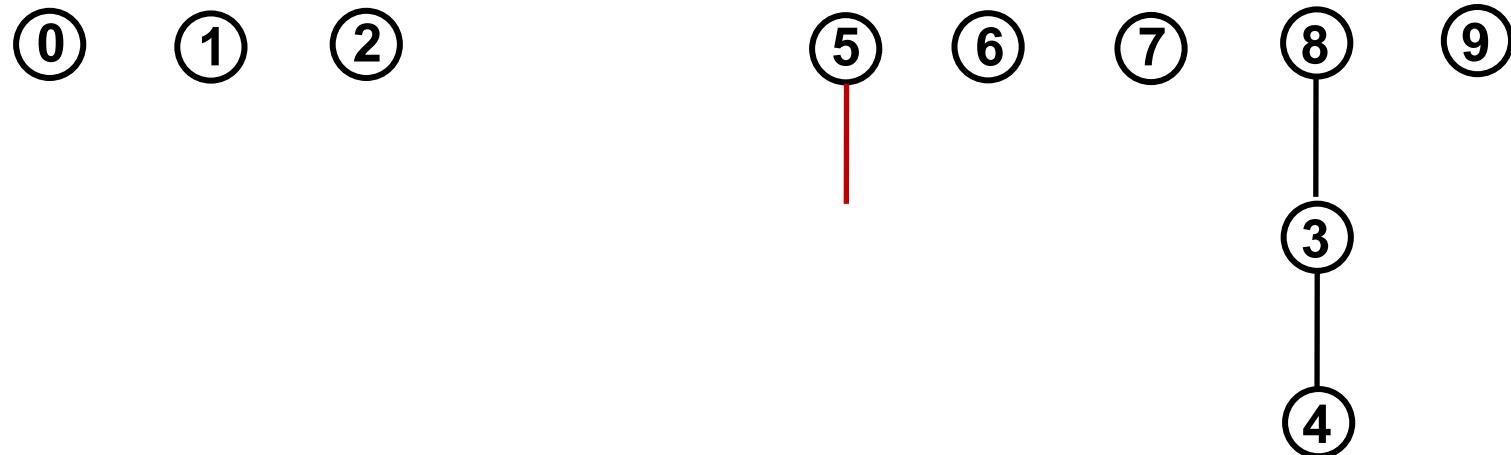
union(3, 8)



| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|-----------|---|---|---|---|---|---|---|---|---|---|
| id | 0 | 1 | 2 | 8 | 3 | 5 | 6 | 7 | 8 | 9 |

QuickUnion Demo

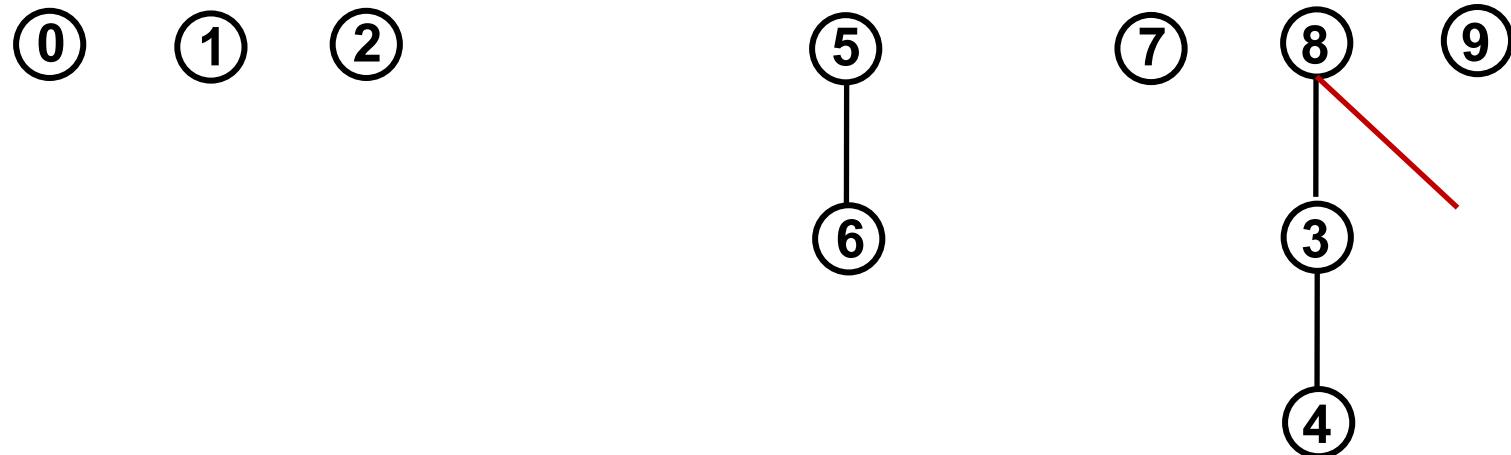
union(6, 5)



| id | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|----|---|---|---|---|---|---|---|---|---|---|
| | 0 | 1 | 2 | 8 | 3 | 5 | 5 | 7 | 8 | 9 |

QuickUnion Demo

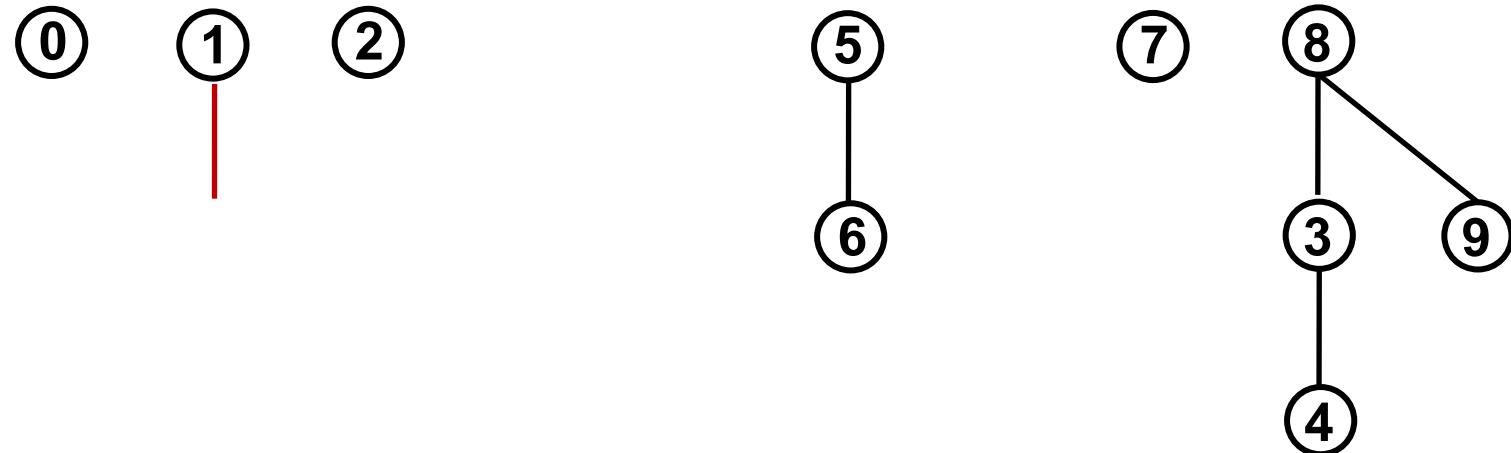
union(9, 4)



| | | | | | | | | | | |
|-----------|---|---|---|---|---|---|---|---|---|---|
| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| id | 0 | 1 | 2 | 8 | 3 | 5 | 5 | 7 | 8 | 8 |

QuickUnion Demo

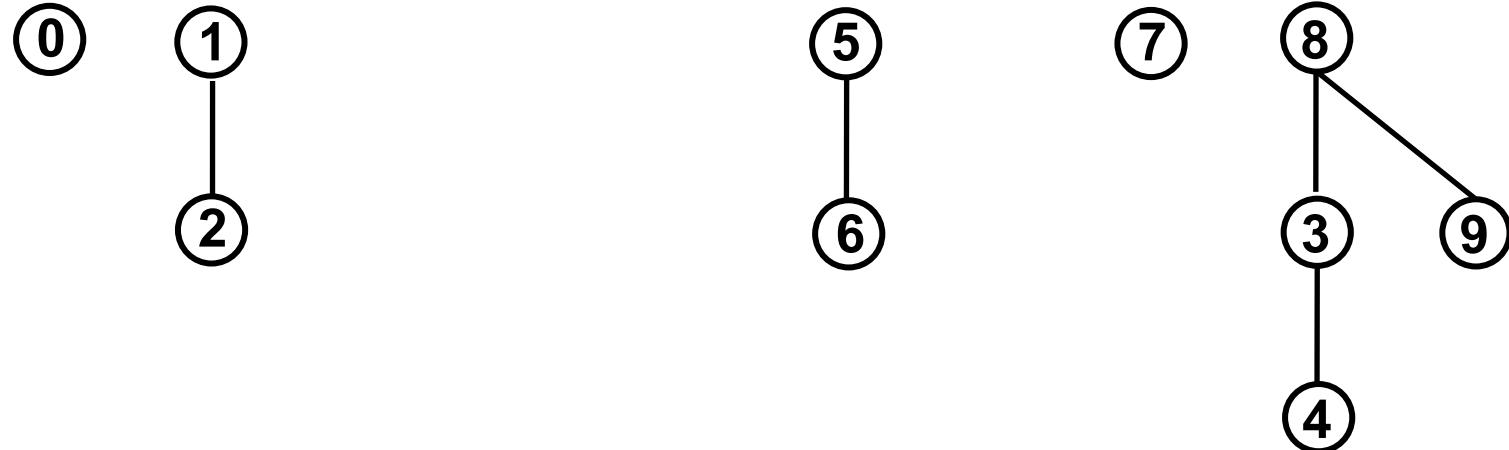
union(2, 1)



| id | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|----|---|---|---|---|---|---|---|---|---|---|
| | 0 | 1 | 1 | 8 | 3 | 5 | 5 | 7 | 8 | 8 |

QuickUnion Demo

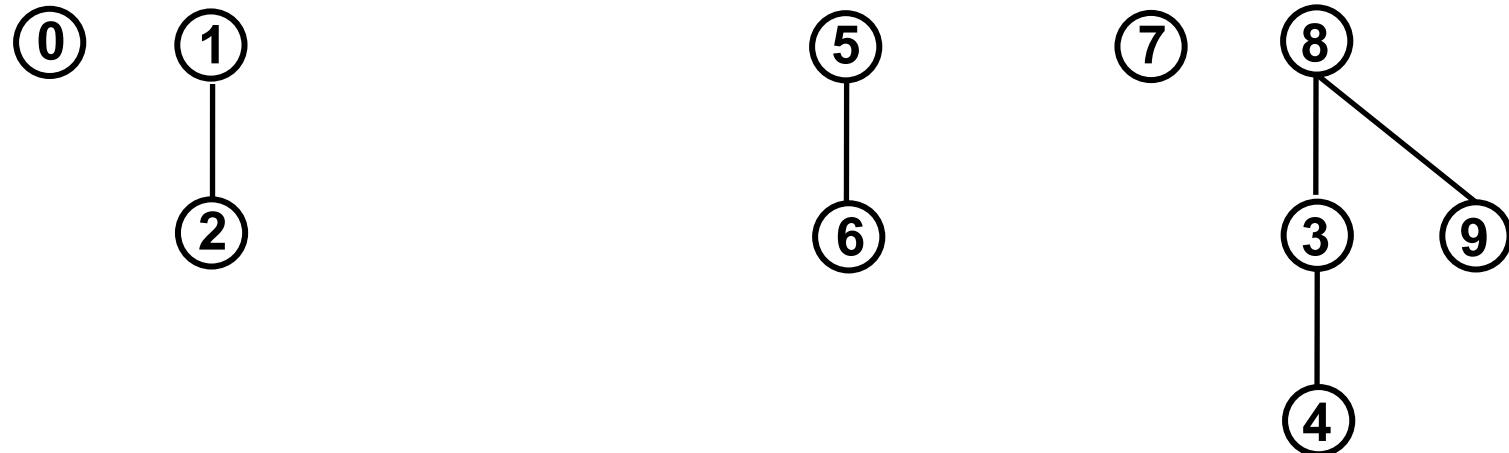
connected(8, 9) ✓



| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|-----------|---|---|---|---|---|---|---|---|---|---|
| id | 0 | 1 | 1 | 8 | 3 | 5 | 5 | 7 | 8 | 8 |

QuickUnion Demo

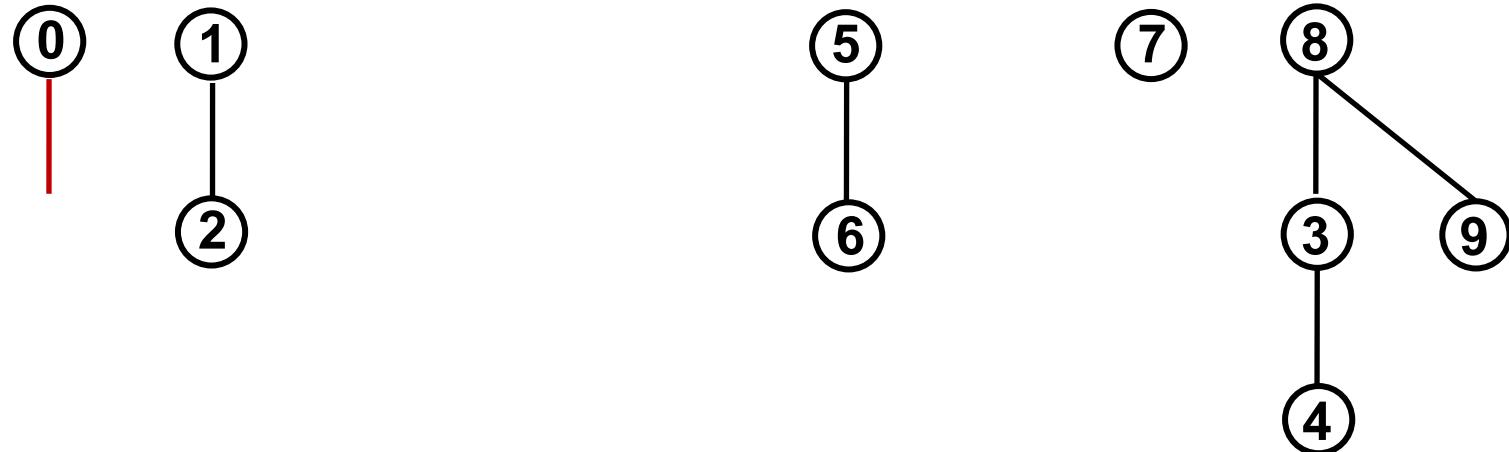
connected(5, 4) ✗



| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|-----------|---|---|---|---|---|---|---|---|---|---|
| id | 0 | 1 | 1 | 8 | 3 | 5 | 5 | 7 | 8 | 8 |

QuickUnion Demo

union(5, 0)



| | | | | | | | | | | |
|-----------|---|---|---|---|---|---|---|---|---|---|
| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| id | 0 | 1 | 1 | 8 | 3 | 0 | 5 | 7 | 8 | 8 |

QuickUnion Implementation (Java)

```
public class QuickUnionUF
{
    private int[] id;

    public QuickUnionUF(int N)
    {
        id = new int[N];
        for (int i = 0; i < N; i++) id[i] = i; ←
    }

    public int find(int i)
    {
        while (i != id[i]) i = id[i]; ←
        return i;
    }

    public void union(int p, int q)
    {
        int i = find(p);
        int j = find(q);
        id[i] = j; ←
    }
}
```

set id of each object to itself
(N array accesses)

chase parent pointers until reach root
(depth of i array accesses)

change root of p to point to root of q
(depth of p and q array accesses)

QuickUnion - Time Complexity

- Measured by: number of array accesses (read or write)

| Algorithm | Initialization | union | find | connected | |
|------------|----------------|--------|--------|-----------|------------|
| QuickFind | $O(N)$ | $O(N)$ | $O(1)$ | $O(1)$ | |
| QuickUnion | $O(N)$ | $O(N)$ | $O(N)$ | $O(N)$ | worst case |

- QuickFind:
 - union() is too expensive.
 - Trees are flat, but need many id updates to keep them flat
- QuickUnion:
 - Trees can be tall.
 - find() and connected() are too expensive.



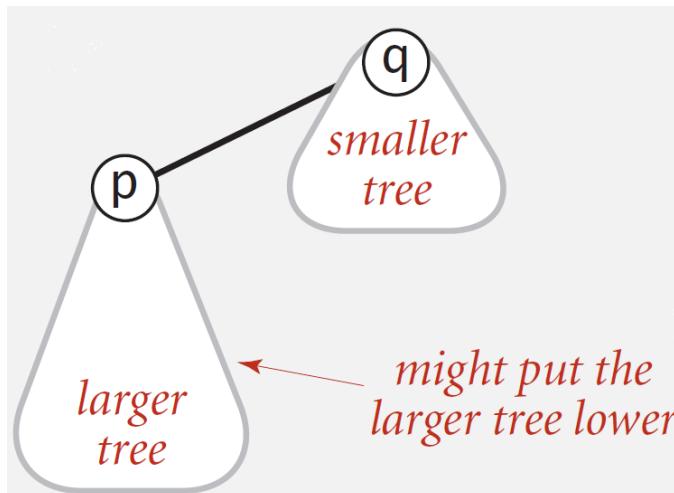
WQUPC Algorithm

1. Weighted QuickUnion

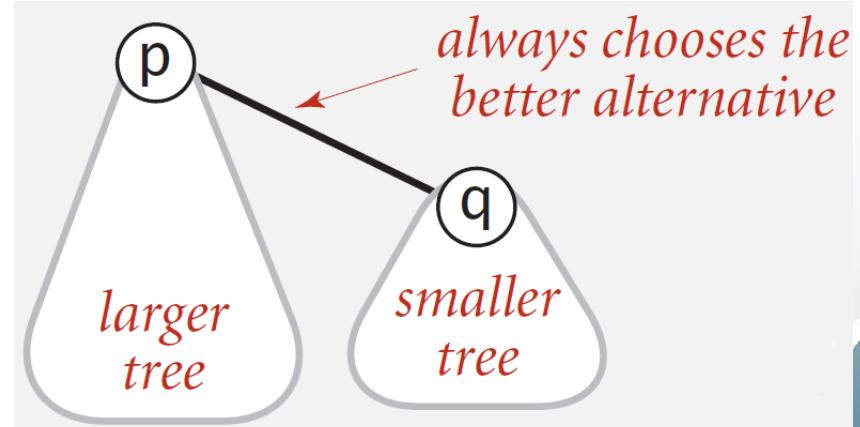
- Idea:

- Modify QuickUnion to avoid tall trees.
- Keep track of size of each tree (number of objects).
- Balance by linking root of smaller tree to root of larger tree.
- Other alternatives: ?

QuickUnion



Weighted QuickUnion



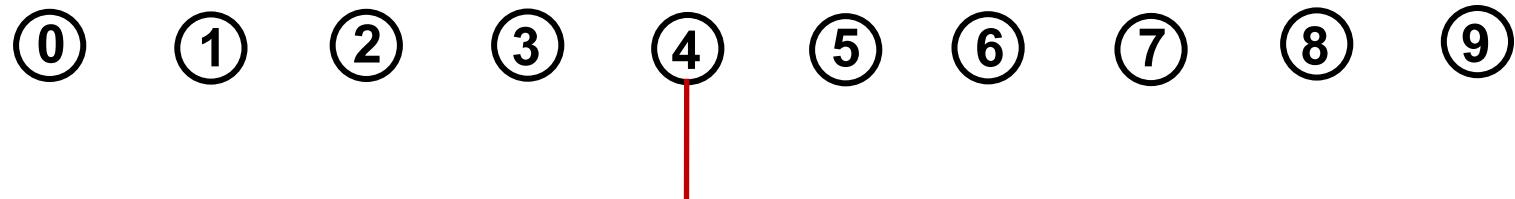
Weighted QuickUnion Demo

① ② ③ ④ ⑤ ⑥ ⑦ ⑧ ⑨

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|-----------|---|---|---|---|---|---|---|---|---|---|
| id | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

Weighted QuickUnion Demo

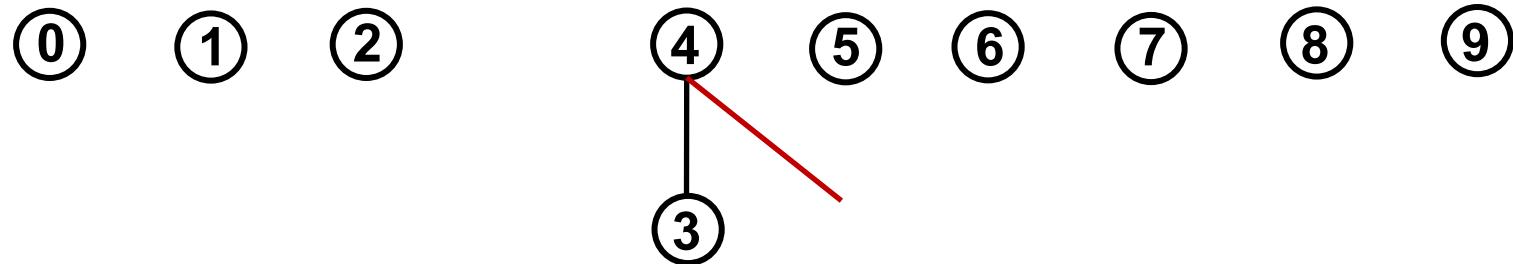
union(4, 3)



| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|-----------|---|---|---|---|---|---|---|---|---|---|
| id | 0 | 1 | 2 | 4 | 4 | 5 | 6 | 7 | 8 | 9 |

Weighted QuickUnion Demo

union(3, 8)

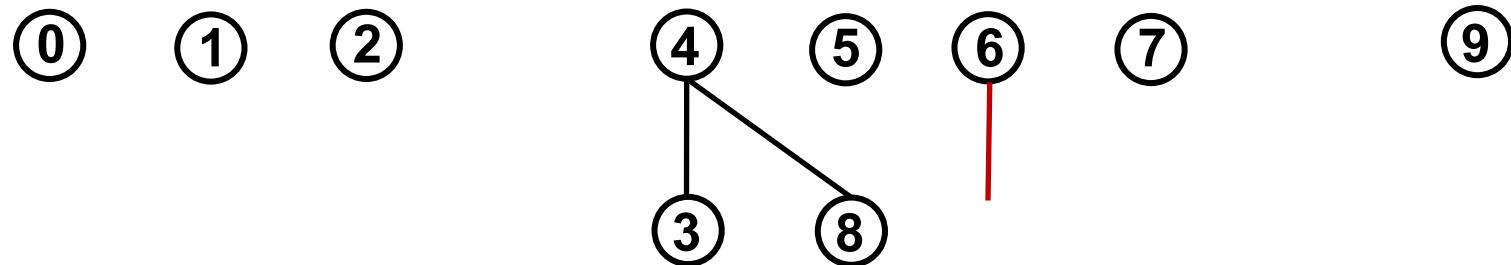


Weighting makes 8 point to 4

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|-----------|---|---|---|---|---|---|---|---|---|---|
| id | 0 | 1 | 2 | 4 | 4 | 5 | 6 | 7 | 4 | 9 |

Weighted QuickUnion Demo

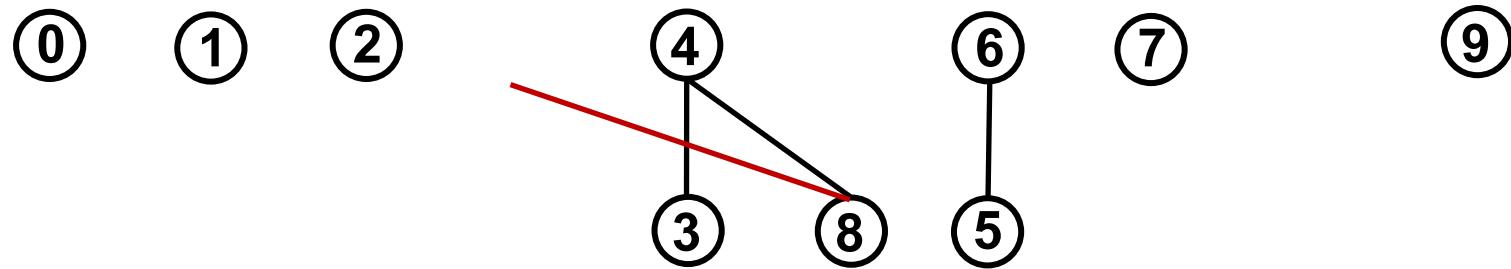
union(6, 5)



| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|-----------|---|---|---|---|---|---|---|---|---|---|
| id | 0 | 1 | 2 | 4 | 4 | 6 | 6 | 7 | 4 | 9 |

Weighted QuickUnion Demo

union(9, 4)

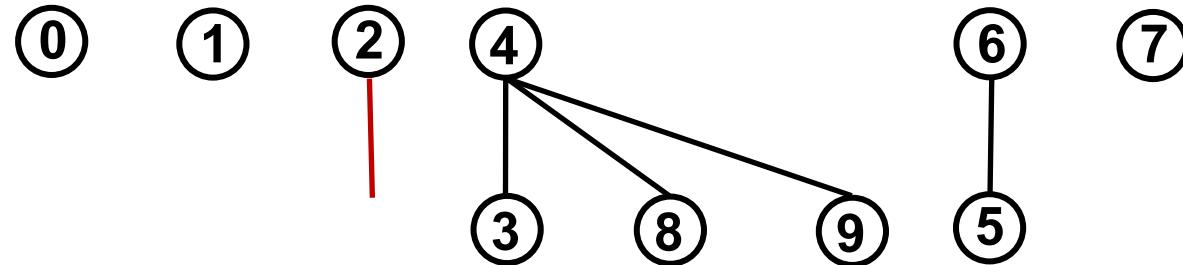


Weighting makes 9 point to
4

| | | | | | | | | | | |
|-----------|---|---|---|---|---|---|---|---|---|---|
| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| id | 0 | 1 | 2 | 4 | 4 | 6 | 6 | 7 | 4 | 4 |

Weighted QuickUnion Demo

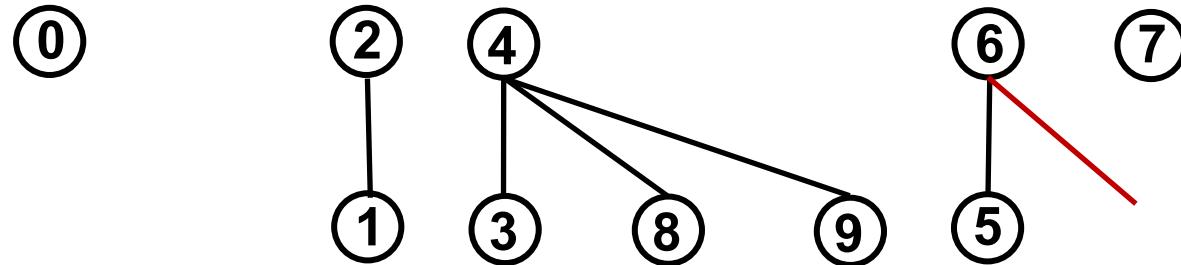
union(2, 1)



| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|-----------|---|---|---|---|---|---|---|---|---|---|
| id | 0 | 2 | 2 | 4 | 4 | 6 | 6 | 7 | 4 | 4 |

Weighted QuickUnion Demo

union(5, 0)

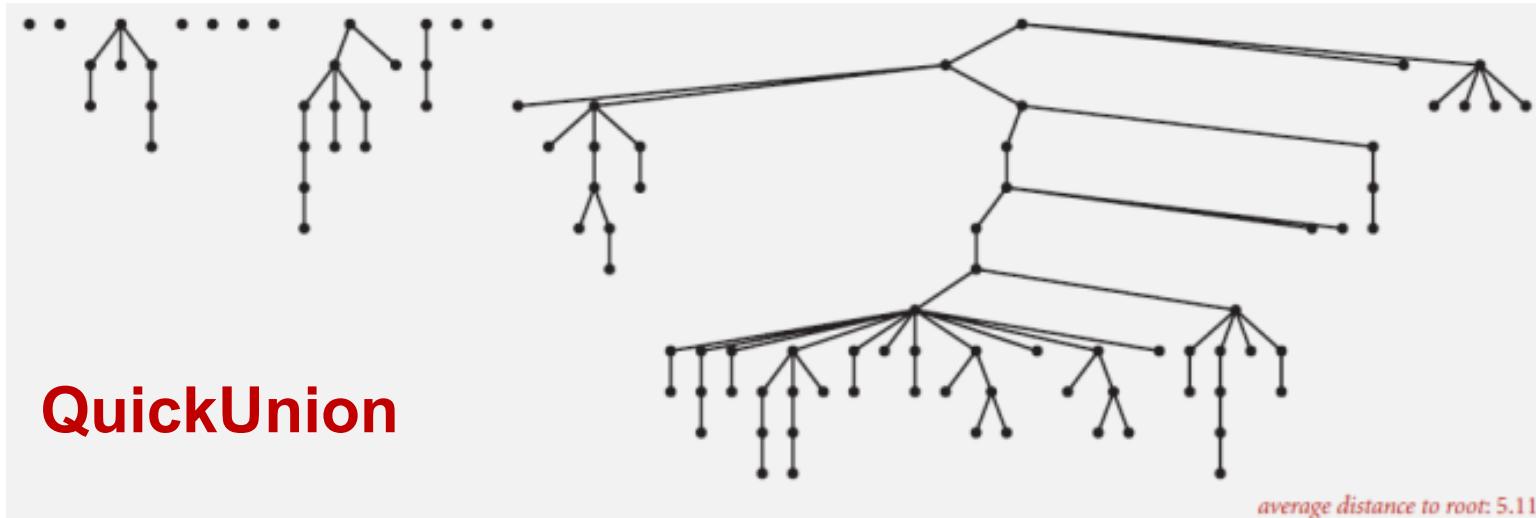


Weighting makes 0 point to
6

| id | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|----|---|---|---|---|---|---|---|---|---|---|
| id | 6 | 2 | 2 | 4 | 4 | 6 | 6 | 7 | 4 | 4 |

QuickUnion vs Weighted QuickUnion

On 100 objects, 88 union operations



Weighted QuickUnion

Weighted QuickUnion Implementation

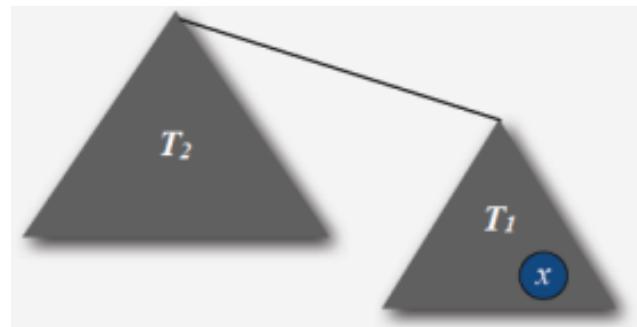
- Data structure:
 - Extra array $sz[i]$ to count no. of objects in the tree rooted at i .
- **find()** and **connect()**
 - Same as QuickUnion
- **union()**
 - Make smaller tree lower
 - Update the array $sz[]$

roots

```
int i = find(p);
int j = find(q);
if (i == j) return;
if (sz[i] < sz[j]) { id[i] = j; sz[j] += sz[i]; }
else { id[j] = i; sz[i] += sz[j]; }
```

Weighted QuickUnion - Time Complexity

- Running time depends on the depth of a node.
- **Proposition.** Depth of any node x is at most $\log_2 N$.
- Proof:
 - What causes the depth of a node x to increase?
 - Increases by 1 when tree T_1 containing x is merged into another tree T_2
 - Size of the tree containing x at least doubles since $|T_2| \geq |T_1|$
 - Size of the tree containing x can double at most $\log_2 N$ times. (?)



Weighted QuickUnion - Time Complexity

- Running time depends on the depth of a node.
- Proposition.** Depth of any node x is at most $\log_2 N$.

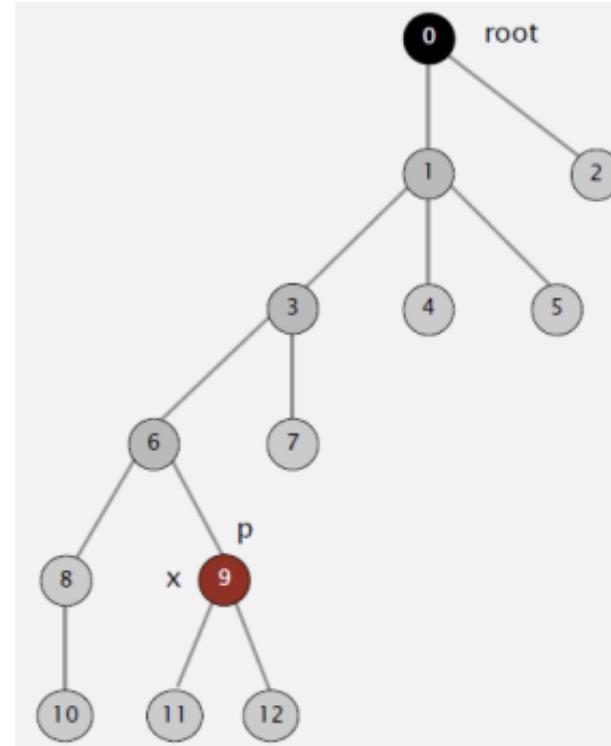
| Algorithm | Initialization | union | find | connected |
|----------------------------|----------------|-------------|-------------|-------------|
| QuickFind | $O(N)$ | $O(N)$ | $O(1)$ | $O(1)$ |
| QuickUnion | $O(N)$ | $O(N)$ | $O(N)$ | $O(N)$ |
| Weighted QuickUnion | $O(N)$ | $O(\log N)$ | $O(\log N)$ | $O(\log N)$ |

- Stop here??**
 - No, easy to improve further.

2. Weighted QuickUnion with Path Compression

- Idea:

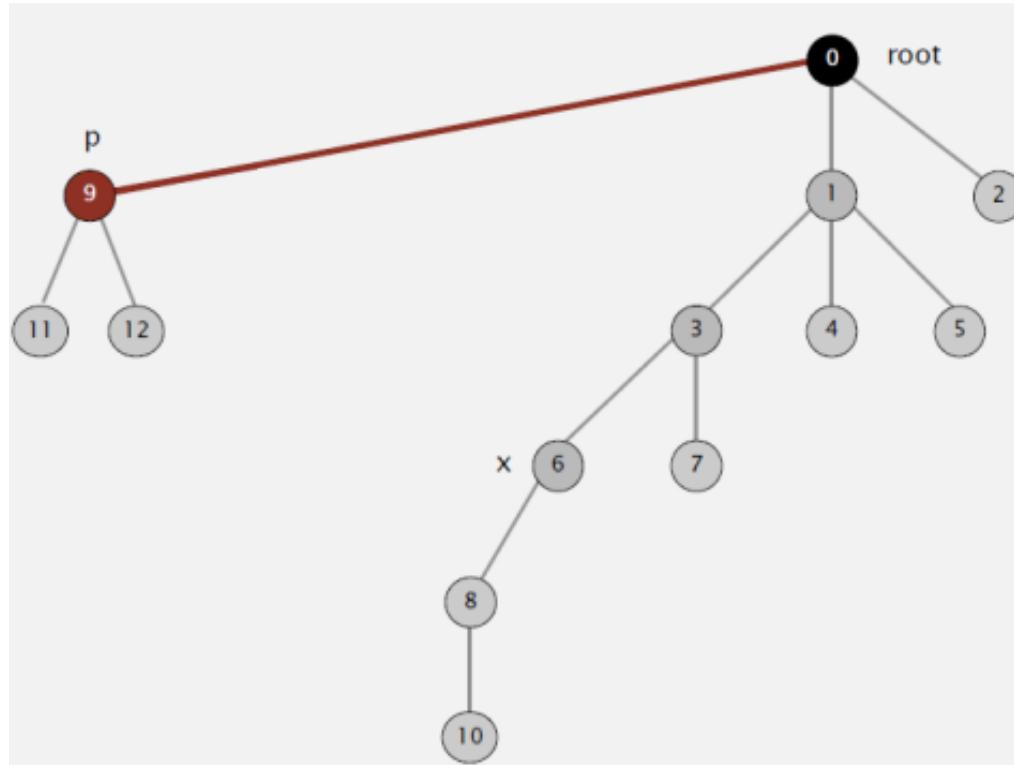
- Right after computing the root of p , set the $\text{id}[]$ of each node on the path to that root



2. Weighted QuickUnion with Path Compression

- Idea:

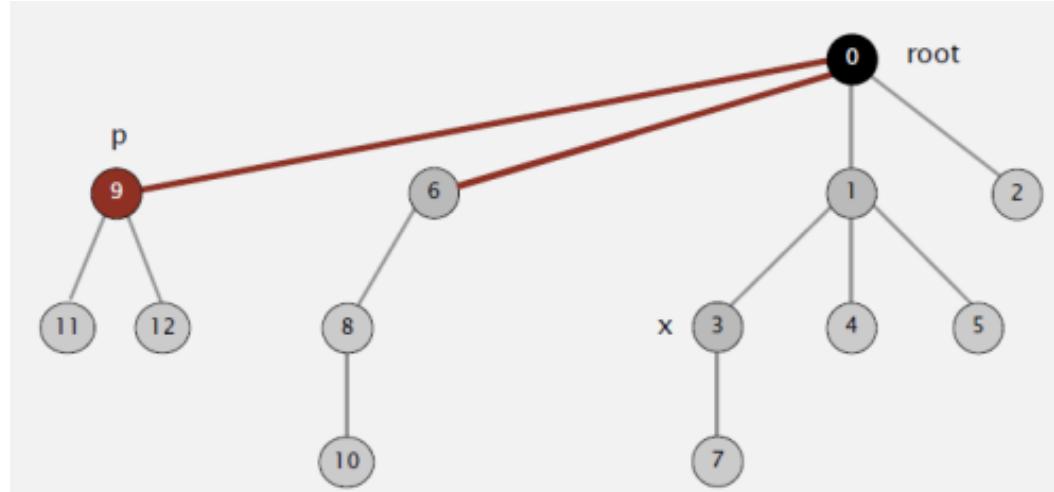
- Right after computing the root of p , set the $\text{id}[]$ of each node on the path to that root



2. Weighted QuickUnion with Path Compression

- Idea:

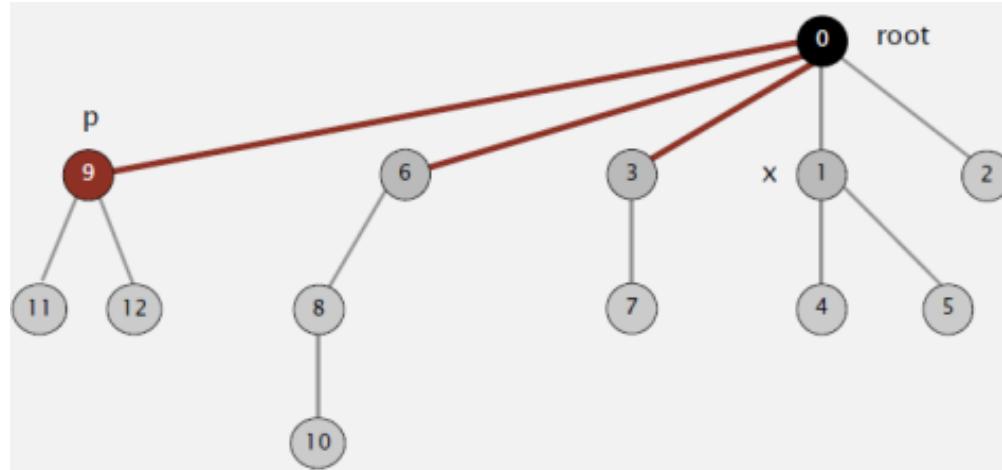
- Right after computing the root of p , set the $\text{id}[]$ of each node on the path to that root



2. Weighted QuickUnion with Path Compression

- Idea:

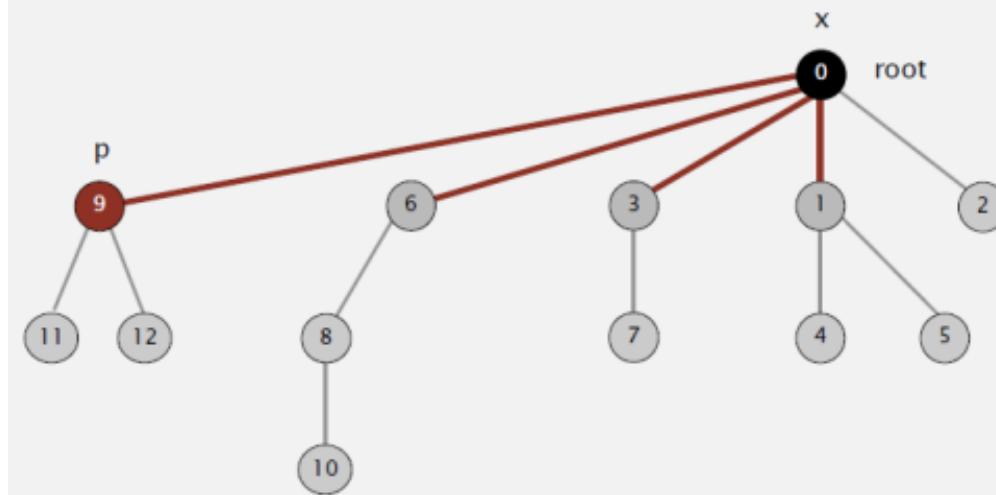
- Right after computing the root of p , set the $\text{id}[]$ of each node on the path to that root



2. Weighted QuickUnion with Path Compression

- Idea:

- Right after computing the root of p , set the $\text{id}[]$ of each node on the path to that root



WQUPC Implementation (Java)

- Two-pass implementation
 - Add second loop to find() to set the id[] of each examined node to the root. (As in previous example)
- Simpler one-pass variant (path halving)
 - Make every other node in path point to its grandparent

```
public int find(int i)
{
    while (i != id[i])
    {
        id[i] = id[id[i]]; ← only one extra line of code !
        i = id[i];
    }
    return i;
}
```

- In practice: keeps tree almost completely flat

WQUPC: Time Complexity

Proposition. [Hopcroft-Ulman, Tarjan] Starting from an empty data structure, any sequence of M union-find ops on N objects makes $\leq c(N + M \lg^* N)$ array accesses.

- Analysis can be improved to $N + M \alpha(M, N)$.
- Simple algorithm with fascinating mathematics.

 2^{16}

| N | $\lg^* N$ |
|--------|-----------|
| 1 | 0 |
| 2 | 1 |
| 4 | 2 |
| 16 | 3 |
| 65536 | 4 |
| 265536 | 5 |

iterated \lg function

- [Fredman-Saks] No linear-time algorithm exists.
- WQUPC:
 - In theory: not quite linear
 - In practice: linear

Summary: Union-Find

- Weighted QuickUnion (w/wo path compression) makes it possible to solve problems that could not otherwise be addressed.
- Complexity for M union-find operations on N objects

| Algorithms | Worst-Case Time |
|---|------------------|
| QuickFind | $M N$ |
| QuickUnion | $M N$ |
| Weighted QuickUnion | $N + M \log N$ |
| Weighted QuickUnion with Path Compression | $N + M \log^* N$ |

- [10¹¹ unions with 10¹¹ objects]
 - WQUPC reduces time from 3000 years to 6 seconds
 - Supercomputer won't help much; good algorithm makes it feasible!

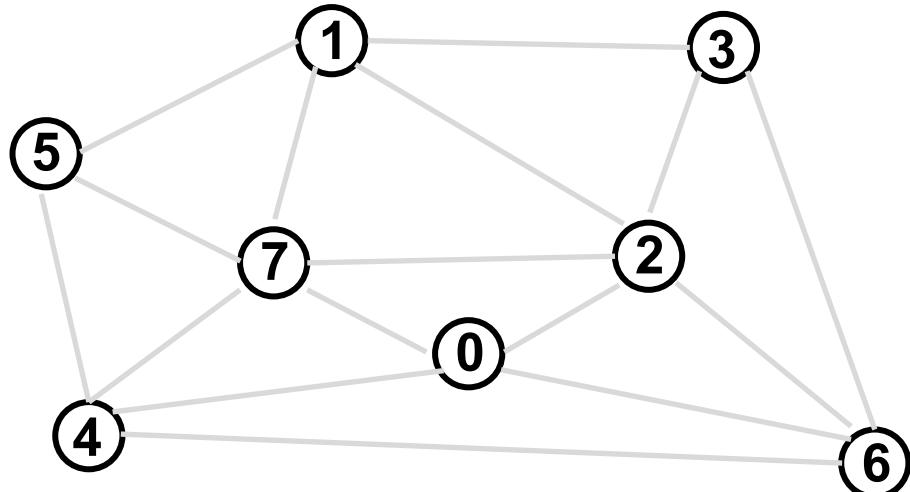


Kruskal's Algorithm

Kruskal's Algorithm

- To compute the minimum spanning tree (MST)
- Uses the **greedy** strategy
- Main Idea:
 - Consider edges in increasing order of weight
 - Add next edge to tree T unless it would create a cycle

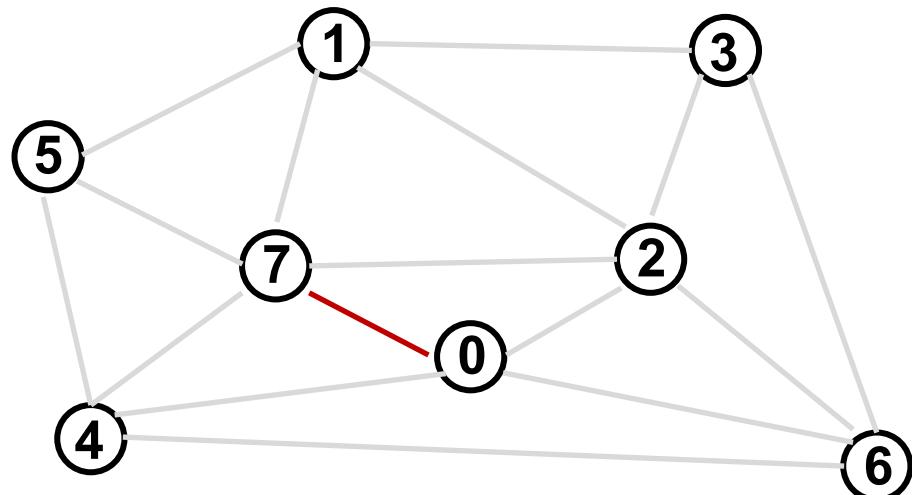
Kruskal's Algorithm - Example



Graph edges
in weight
increasing
order

| | |
|-----|------|
| 0-7 | 0.16 |
| 2-3 | 0.17 |
| 1-7 | 0.19 |
| 0-2 | 0.26 |
| 5-7 | 0.28 |
| 1-3 | 0.29 |
| 1-5 | 0.32 |
| 2-7 | 0.34 |
| 4-5 | 0.35 |
| 1-2 | 0.36 |
| 4-7 | 0.37 |
| 0-4 | 0.38 |
| 6-2 | 0.40 |
| 3-6 | 0.52 |
| 6-0 | 0.58 |
| 6-4 | 0.93 |

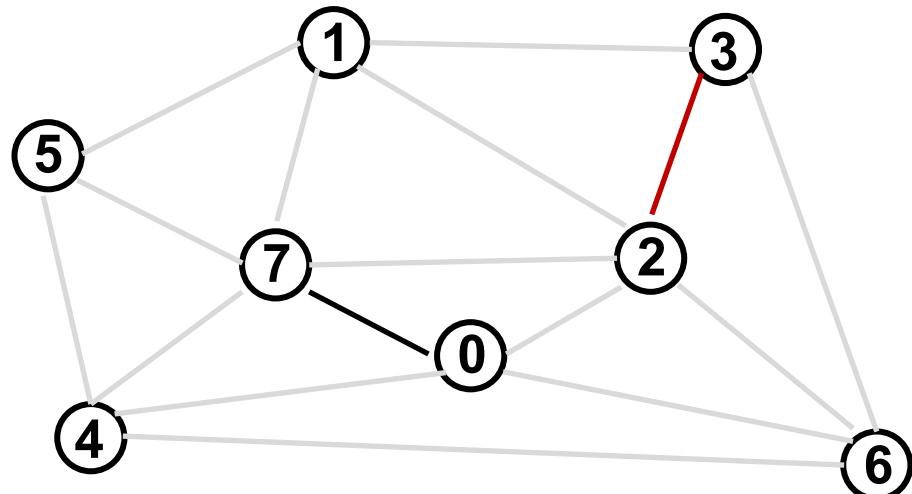
Kruskal's Algorithm - Example



Graph edges
in weight
increasing
order

| | |
|-----|------|
| 0-7 | 0.16 |
| 2-3 | 0.17 |
| 1-7 | 0.19 |
| 0-2 | 0.26 |
| 5-7 | 0.28 |
| 1-3 | 0.29 |
| 1-5 | 0.32 |
| 2-7 | 0.34 |
| 4-5 | 0.35 |
| 1-2 | 0.36 |
| 4-7 | 0.37 |
| 0-4 | 0.38 |
| 6-2 | 0.40 |
| 3-6 | 0.52 |
| 6-0 | 0.58 |
| 6-4 | 0.93 |

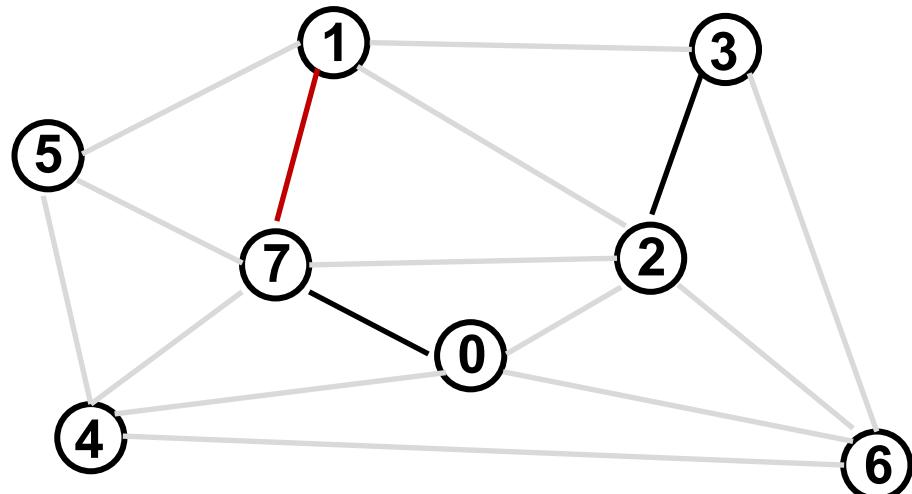
Kruskal's Algorithm - Example



Graph edges
in weight
increasing
order

| | |
|-----|-------------|
| 0-7 | 0.16 |
| 2-3 | 0.17 |
| 1-7 | 0.19 |
| 0-2 | 0.26 |
| 5-7 | 0.28 |
| 1-3 | 0.29 |
| 1-5 | 0.32 |
| 2-7 | 0.34 |
| 4-5 | 0.35 |
| 1-2 | 0.36 |
| 4-7 | 0.37 |
| 0-4 | 0.38 |
| 6-2 | 0.40 |
| 3-6 | 0.52 |
| 6-0 | 0.58 |
| 6-4 | 0.93 |

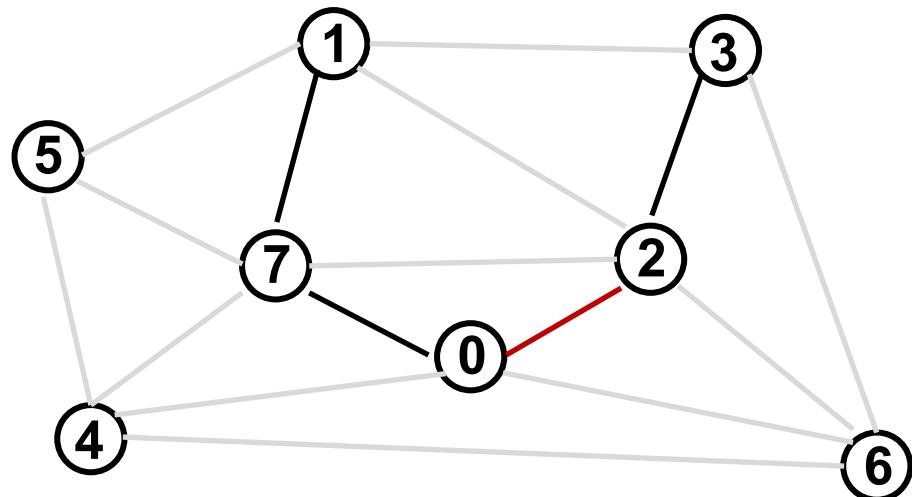
Kruskal's Algorithm - Example



Graph edges
in weight
increasing
order

| | |
|------------|-------------|
| 0-7 | 0.16 |
| 2-3 | 0.17 |
| 1-7 | 0.19 |
| 0-2 | 0.26 |
| 5-7 | 0.28 |
| 1-3 | 0.29 |
| 1-5 | 0.32 |
| 2-7 | 0.34 |
| 4-5 | 0.35 |
| 1-2 | 0.36 |
| 4-7 | 0.37 |
| 0-4 | 0.38 |
| 6-2 | 0.40 |
| 3-6 | 0.52 |
| 6-0 | 0.58 |
| 6-4 | 0.93 |

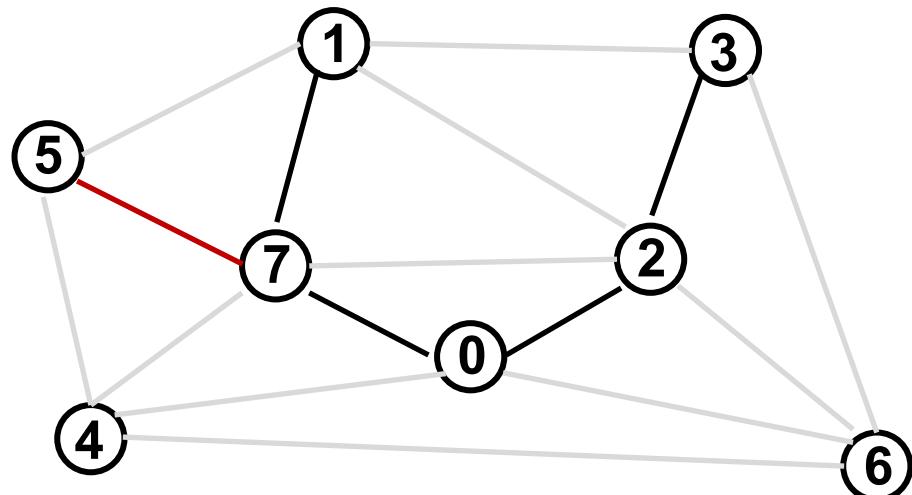
Kruskal's Algorithm - Example



Graph edges
in weight
increasing
order

| | |
|------------|-------------|
| 0-7 | 0.16 |
| 2-3 | 0.17 |
| 1-7 | 0.19 |
| 0-2 | 0.26 |
| 5-7 | 0.28 |
| 1-3 | 0.29 |
| 1-5 | 0.32 |
| 2-7 | 0.34 |
| 4-5 | 0.35 |
| 1-2 | 0.36 |
| 4-7 | 0.37 |
| 0-4 | 0.38 |
| 6-2 | 0.40 |
| 3-6 | 0.52 |
| 6-0 | 0.58 |
| 6-4 | 0.93 |

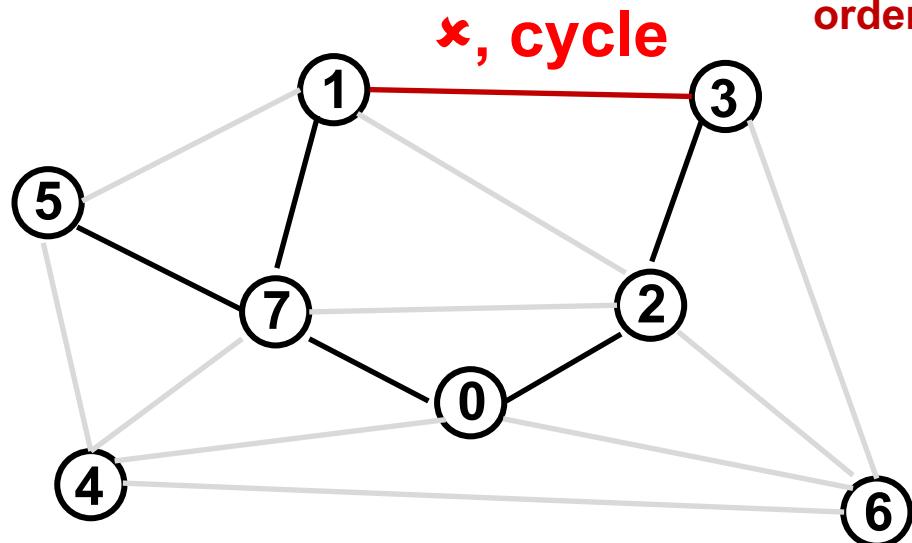
Kruskal's Algorithm - Example



Graph edges
in weight
increasing
order

| | |
|------------|-------------|
| 0-7 | 0.16 |
| 2-3 | 0.17 |
| 1-7 | 0.19 |
| 0-2 | 0.26 |
| 5-7 | 0.28 |
| 1-3 | 0.29 |
| 1-5 | 0.32 |
| 2-7 | 0.34 |
| 4-5 | 0.35 |
| 1-2 | 0.36 |
| 4-7 | 0.37 |
| 0-4 | 0.38 |
| 6-2 | 0.40 |
| 3-6 | 0.52 |
| 6-0 | 0.58 |
| 6-4 | 0.93 |

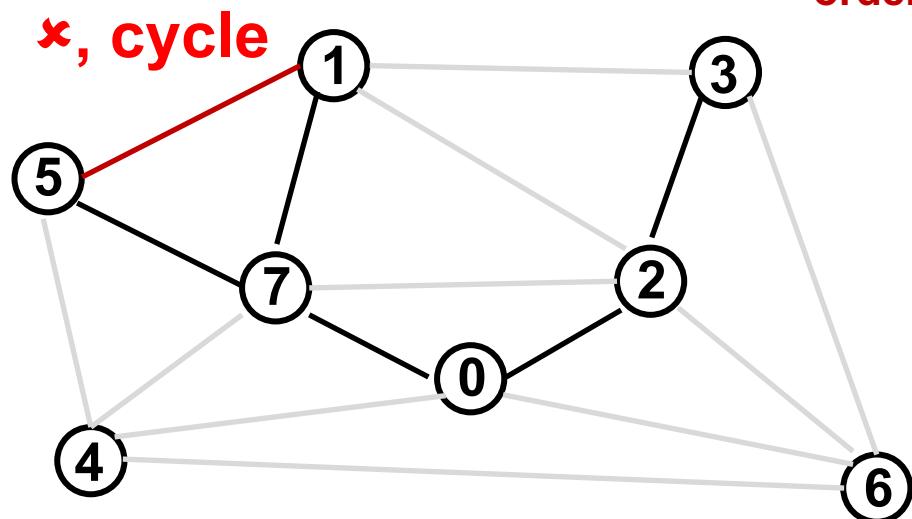
Kruskal's Algorithm - Example



Graph edges
in weight
increasing
order

| | |
|------------|-------------|
| 0-7 | 0.16 |
| 2-3 | 0.17 |
| 1-7 | 0.19 |
| 0-2 | 0.26 |
| 5-7 | 0.28 |
| 1-3 | 0.29 |
| 1-5 | 0.32 |
| 2-7 | 0.34 |
| 4-5 | 0.35 |
| 1-2 | 0.36 |
| 4-7 | 0.37 |
| 0-4 | 0.38 |
| 6-2 | 0.40 |
| 3-6 | 0.52 |
| 6-0 | 0.58 |
| 6-4 | 0.93 |

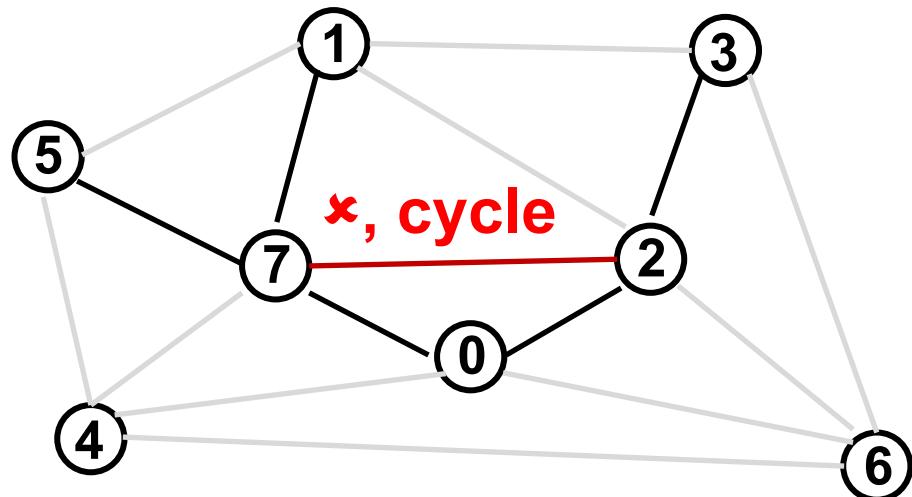
Kruskal's Algorithm - Example



Graph edges
in weight
increasing
order

| | |
|------------|-------------|
| 0-7 | 0.16 |
| 2-3 | 0.17 |
| 1-7 | 0.19 |
| 0-2 | 0.26 |
| 5-7 | 0.28 |
| 1-3 | 0.29 |
| 1-5 | 0.32 |
| 2-7 | 0.34 |
| 4-5 | 0.35 |
| 1-2 | 0.36 |
| 4-7 | 0.37 |
| 0-4 | 0.38 |
| 6-2 | 0.40 |
| 3-6 | 0.52 |
| 6-0 | 0.58 |
| 6-4 | 0.93 |

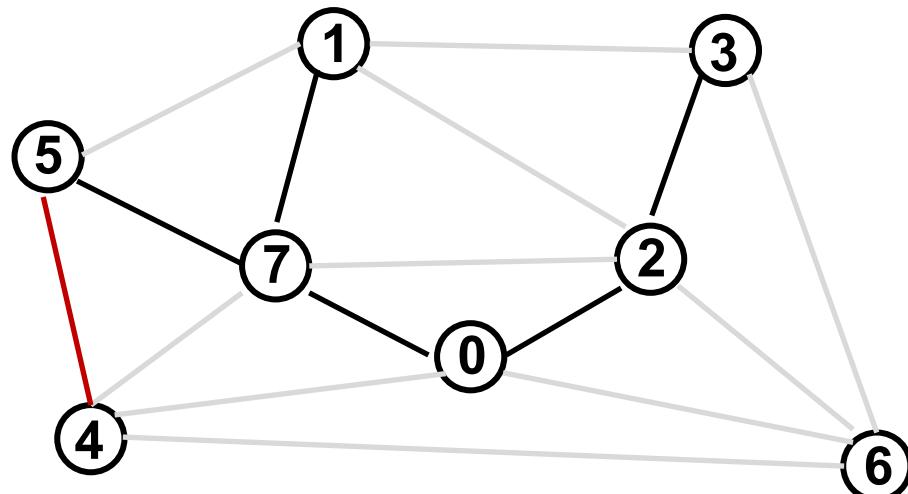
Kruskal's Algorithm - Example



Graph edges
in weight
increasing
order

| | |
|------------|-------------|
| 0-7 | 0.16 |
| 2-3 | 0.17 |
| 1-7 | 0.19 |
| 0-2 | 0.26 |
| 5-7 | 0.28 |
| 1-3 | 0.29 |
| 1-5 | 0.32 |
| 2-7 | 0.34 |
| 4-5 | 0.35 |
| 1-2 | 0.36 |
| 4-7 | 0.37 |
| 0-4 | 0.38 |
| 6-2 | 0.40 |
| 3-6 | 0.52 |
| 6-0 | 0.58 |
| 6-4 | 0.93 |

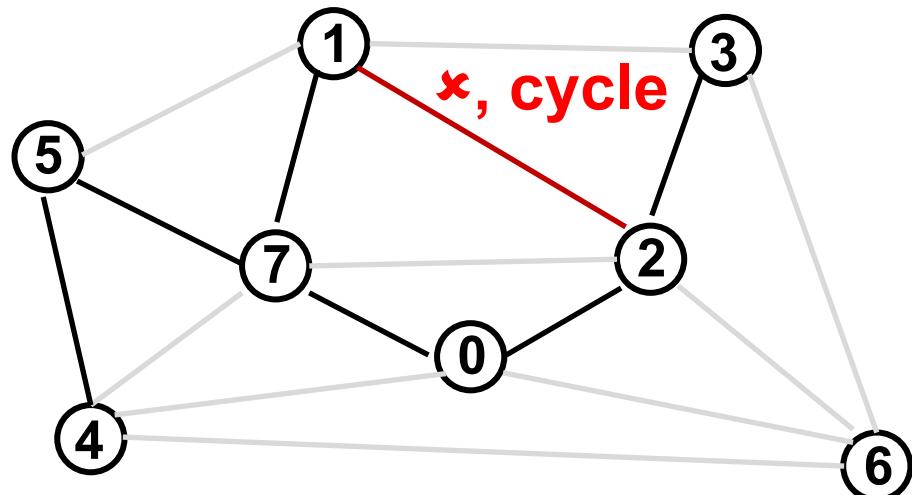
Kruskal's Algorithm - Example



Graph edges
in weight
increasing
order

| | |
|------------|-------------|
| 0-7 | 0.16 |
| 2-3 | 0.17 |
| 1-7 | 0.19 |
| 0-2 | 0.26 |
| 5-7 | 0.28 |
| 1-3 | 0.29 |
| 1-5 | 0.32 |
| 2-7 | 0.34 |
| 4-5 | 0.35 |
| 1-2 | 0.36 |
| 4-7 | 0.37 |
| 0-4 | 0.38 |
| 6-2 | 0.40 |
| 3-6 | 0.52 |
| 6-0 | 0.58 |
| 6-4 | 0.93 |

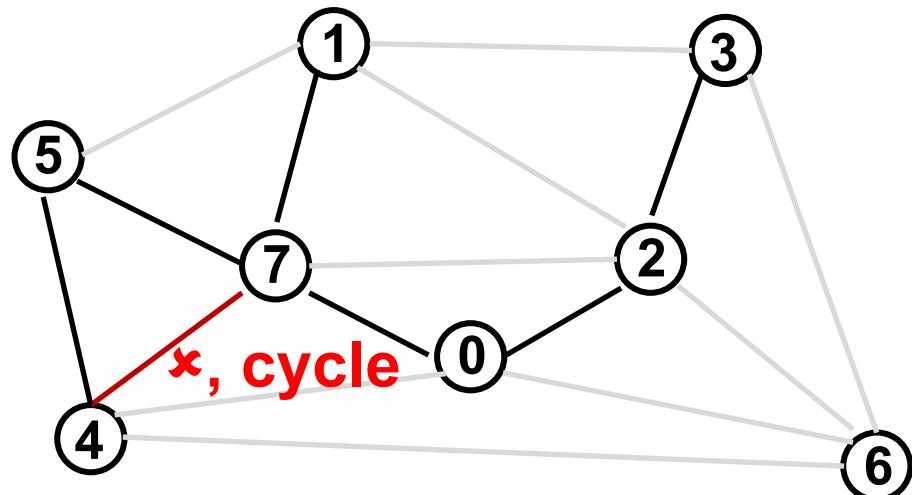
Kruskal's Algorithm - Example



Graph edges
in weight
increasing
order

| | |
|------------|-------------|
| 0-7 | 0.16 |
| 2-3 | 0.17 |
| 1-7 | 0.19 |
| 0-2 | 0.26 |
| 5-7 | 0.28 |
| 1-3 | 0.29 |
| 1-5 | 0.32 |
| 2-7 | 0.34 |
| 4-5 | 0.35 |
| 1-2 | 0.36 |
| 4-7 | 0.37 |
| 0-4 | 0.38 |
| 6-2 | 0.40 |
| 3-6 | 0.52 |
| 6-0 | 0.58 |
| 6-4 | 0.93 |

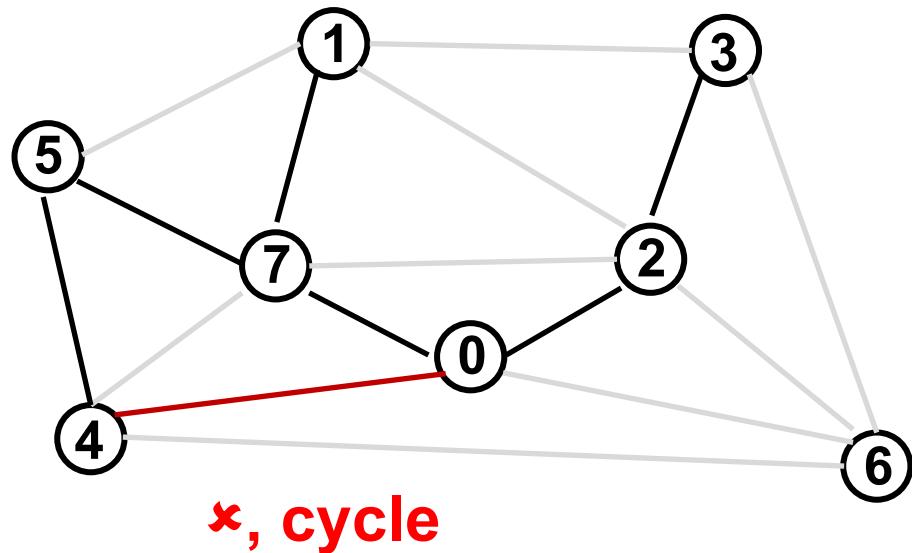
Kruskal's Algorithm - Example



Graph edges
in weight
increasing
order

| | |
|------------|-------------|
| 0-7 | 0.16 |
| 2-3 | 0.17 |
| 1-7 | 0.19 |
| 0-2 | 0.26 |
| 5-7 | 0.28 |
| 1-3 | 0.29 |
| 1-5 | 0.32 |
| 2-7 | 0.34 |
| 4-5 | 0.35 |
| 1-2 | 0.36 |
| 4-7 | 0.37 |
| 0-4 | 0.38 |
| 6-2 | 0.40 |
| 3-6 | 0.52 |
| 6-0 | 0.58 |
| 6-4 | 0.93 |

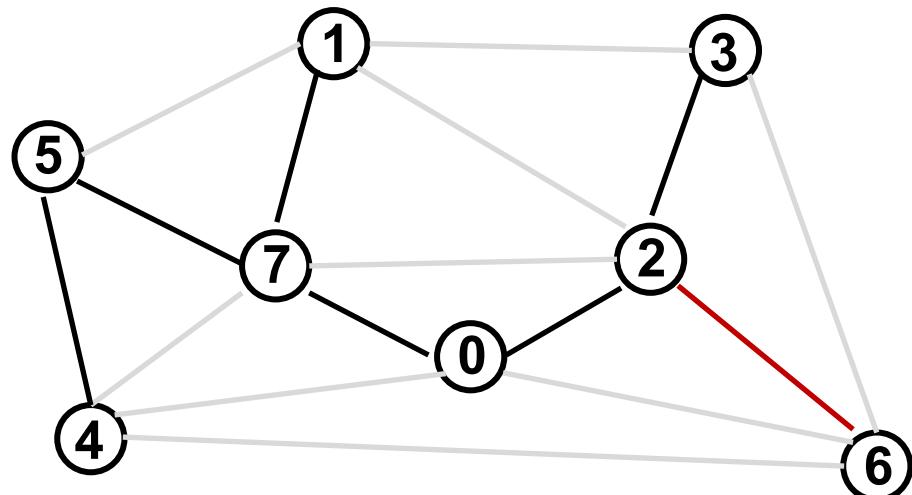
Kruskal's Algorithm - Example



Graph edges
in weight
increasing
order

| | |
|------------|-------------|
| 0-7 | 0.16 |
| 2-3 | 0.17 |
| 1-7 | 0.19 |
| 0-2 | 0.26 |
| 5-7 | 0.28 |
| 1-3 | 0.29 |
| 1-5 | 0.32 |
| 2-7 | 0.34 |
| 4-5 | 0.35 |
| 1-2 | 0.36 |
| 4-7 | 0.37 |
| 0-4 | 0.38 |
| 6-2 | 0.40 |
| 3-6 | 0.52 |
| 6-0 | 0.58 |
| 6-4 | 0.93 |

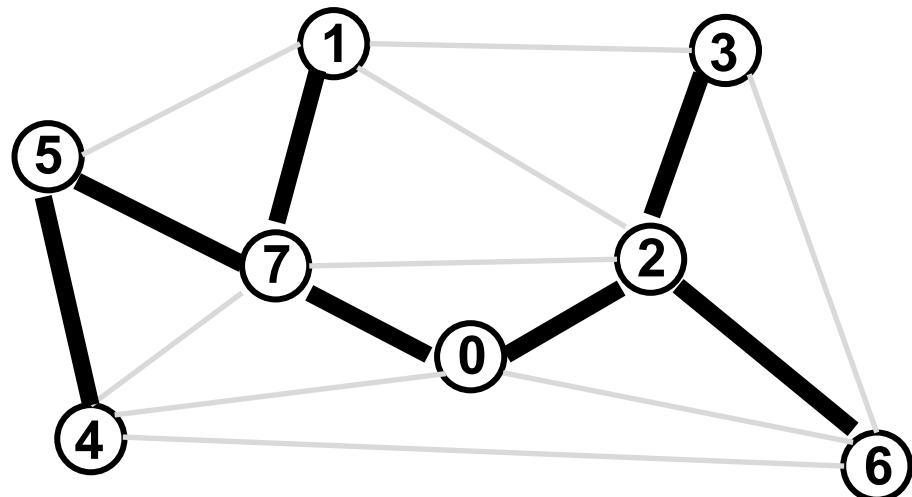
Kruskal's Algorithm - Example



Graph edges
in weight
increasing
order

| | |
|------------|-------------|
| 0-7 | 0.16 |
| 2-3 | 0.17 |
| 1-7 | 0.19 |
| 0-2 | 0.26 |
| 5-7 | 0.28 |
| 1-3 | 0.29 |
| 1-5 | 0.32 |
| 2-7 | 0.34 |
| 4-5 | 0.35 |
| 1-2 | 0.36 |
| 4-7 | 0.37 |
| 0-4 | 0.38 |
| 6-2 | 0.40 |
| 3-6 | 0.52 |
| 6-0 | 0.58 |
| 6-4 | 0.93 |

Kruskal's Algorithm - Example



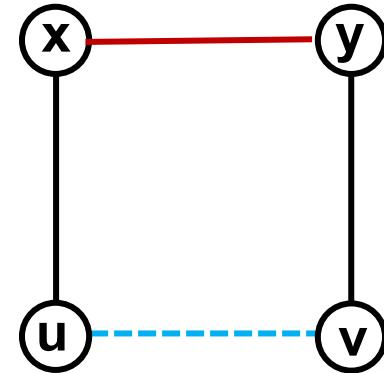
Graph edges
in weight
increasing
order

N-1 edges
Done!

| | |
|------------|-------------|
| 0-7 | 0.16 |
| 2-3 | 0.17 |
| 1-7 | 0.19 |
| 0-2 | 0.26 |
| 5-7 | 0.28 |
| 1-3 | 0.29 |
| 1-5 | 0.32 |
| 2-7 | 0.34 |
| 4-5 | 0.35 |
| 1-2 | 0.36 |
| 4-7 | 0.37 |
| 0-4 | 0.38 |
| 6-2 | 0.40 |
| 3-6 | 0.52 |
| 6-0 | 0.58 |
| 6-4 | 0.93 |

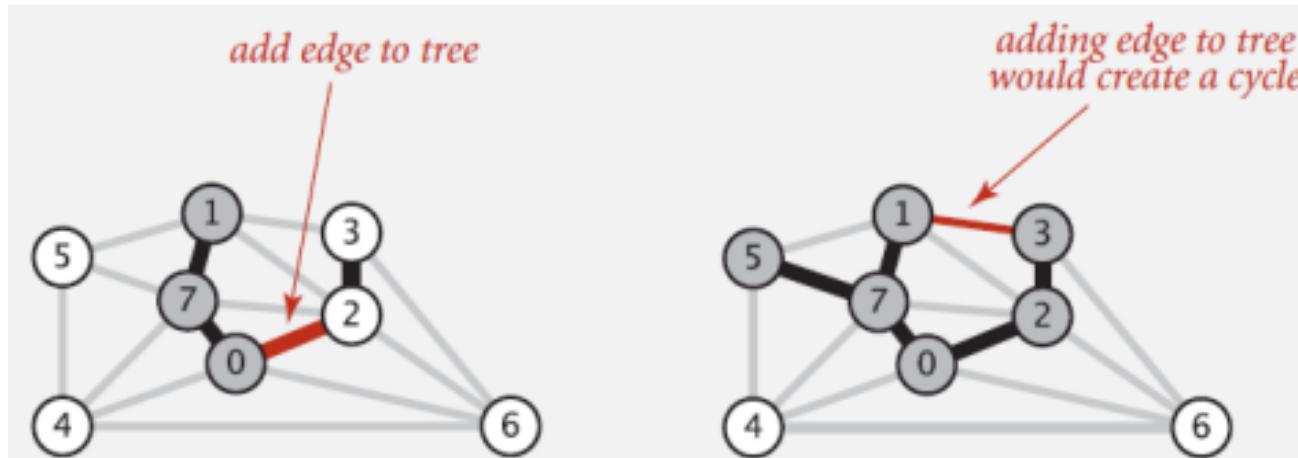
Kruskal's Algorithm: Correctness

- **Proposition.** [Kruskal 1956] Kruskal's algorithm correctly computes the MST.
- Proof by contradiction:
 - Suppose the tree T produced by Kruskal's is not an MST, i.e., it doesn't satisfy the MST property.
 - There is some edge $u-v$ not in T such that adding $u-v$ creates a cycle, in which some other edge $x-y$ has weight $W(x-y) > W(u-v)$.
 - As $W(x-y) > W(u-v)$, edge $x-y$ must be processed after edge $u-v$ in Kruskal's.
 - At the time when $u-v$ is processed, it must be added into T because it doesn't form a cycle.
 - This contradicts that $u-v$ is not in T .



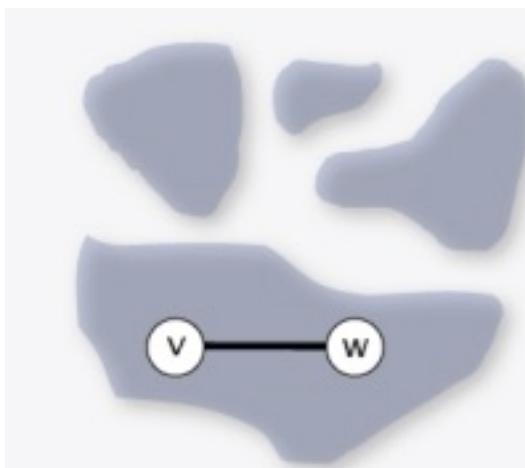
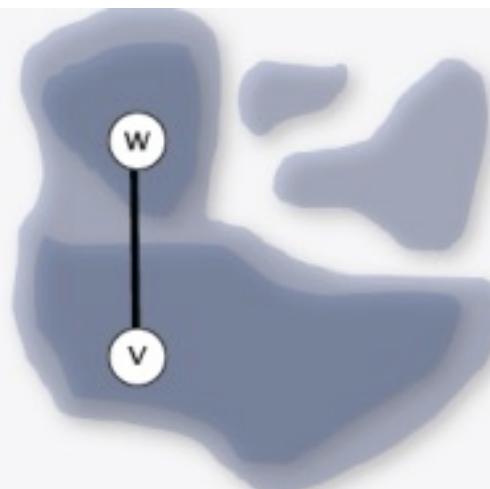
Kruskal's Algorithm: Implementation

- Challenge:
 - How to check if adding an edge $v-w$ to tree T will create a cycle?
- Possible solutions:
 - DFS: run DFS from v to check if it could reach w . $O(|V|)$
 - Union-find: $\text{connected}(v, w) = ?$ $O(\log^* |V|)$



Kruskal's Algorithm: Implementation

- Use the union-find data structure:
 - Use the id array to keep track of connected components in T .
 - If $\text{connected}(v, w) = \text{TRUE}$, then adding $v-w$ will create a cycle.
 - Else, add $v-w$ to T by calling $\text{union}(v, w)$.

Case 1: adding $v-w$ creates a cycleCase 2: add $v-w$ to T and merge sets containing v and w

Kruskal's Algorithm: Implementation (Java)

```

public class KruskalMST
{
    private Queue<Edge> mst = new Queue<Edge>();

    public KruskalMST(EdgeWeightedGraph G)
    {
        MinPQ<Edge> pq = new MinPQ<Edge>(G.edges()); O(|E|) ← build priority queue (or sort)

        UF uf = new UF(G.V()); O(|V|)
        while (!pq.isEmpty() && mst.size() < G.V()-1)
        {
            Edge e = pq.delMin(); O(|E| log|E|) ← greedily add edges to MST
            int v = e.either(), w = e.other(v);
            if (!uf.connected(v, w)) O(|E| log*|V|) ← edge v-w does not create cycle
            {
                uf.union(v, w); O(|V| log*|V|) ← merge sets
                mst.enqueue(e); O(|V|) ← add edge to MST
            }
        }
    }

    public Iterable<Edge> edges()
    { return mst; }
}

```

VEI iteration

Overall: $O(|E| \log|E|)$

Kruskal's Algorithm: Summary

- Kruskal's algorithm finds the minimum spanning trees in weighted graphs
- It is a greedy algorithm.
- It uses the union-find data structure for efficient implementation.
- Its time complexity is $O(|E| \log|E|)$.