CS151 Intro to Data Structures

Merge Sort
Quick Sort

Announcements

HW7 released tonight due April 21st

Lab9 due April 21st

Outline

Hash Maps
Homework Discussion

MergeSort
QuickSort

Handling Collisions

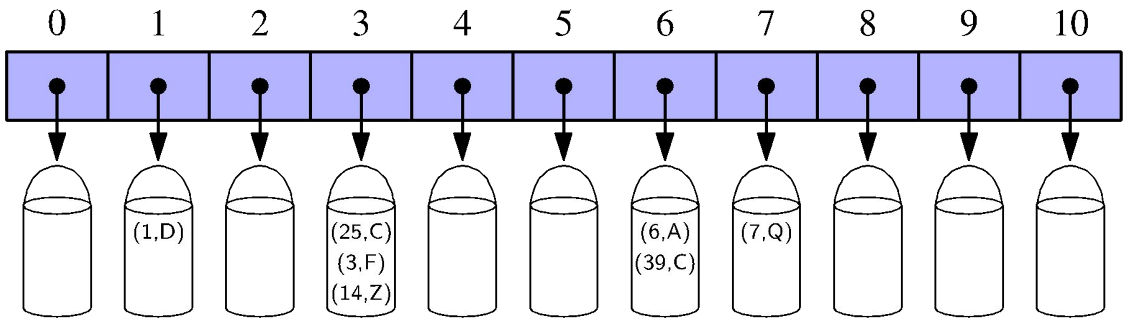
Handling Collisions

A hash function does not guarantee one-to-one mapping – no hash function does

One approach **chaining**:

When more than one key hash to the same index, we have a bucket

Each index holds a collection of entries



Collision Handling

Collisions occur when elements with different keys are mapped to the same cell

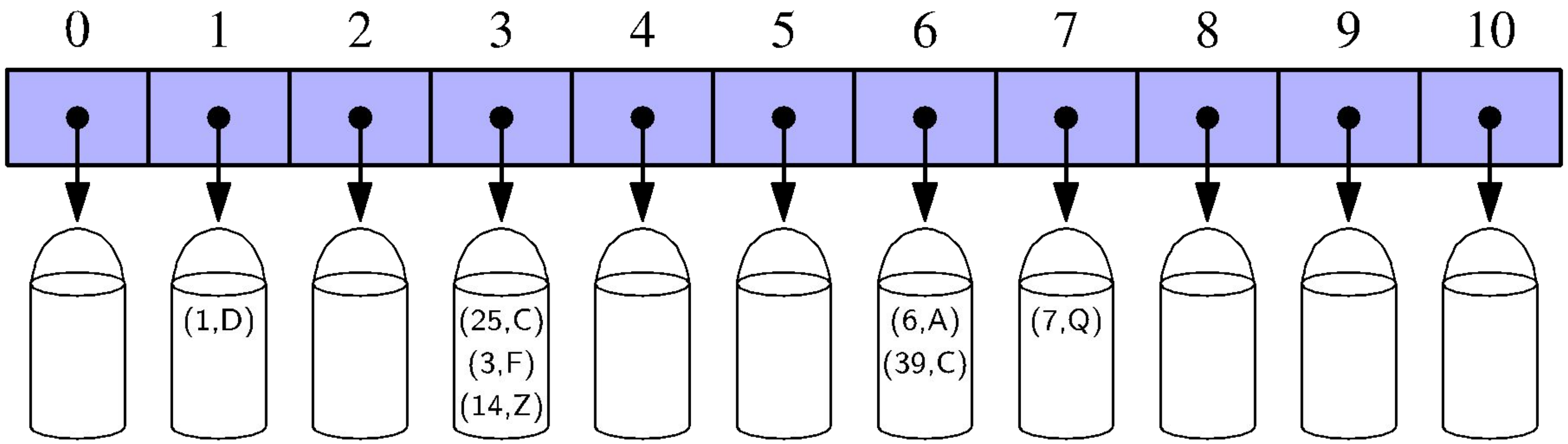
Separate Chaining: let each cell in the table point to a linked list of entries that map there

Simple, but requires additional memory besides the table

Let’s implement a ChainHashMap

What data structure should we use for the buckets?

- LinkedList!



Collision Handling Approach #2

Open Addressing and Probing

When a collision occurs, find an empty slot nearby to store the colliding element

Open Addressing and Probing

- Example: $h(x) = x\%13$
- insert 18(5), 41(2), 22(9), 44(5), 59(7), 32(6), 31(5), 73(8)

Keep "*probing*"

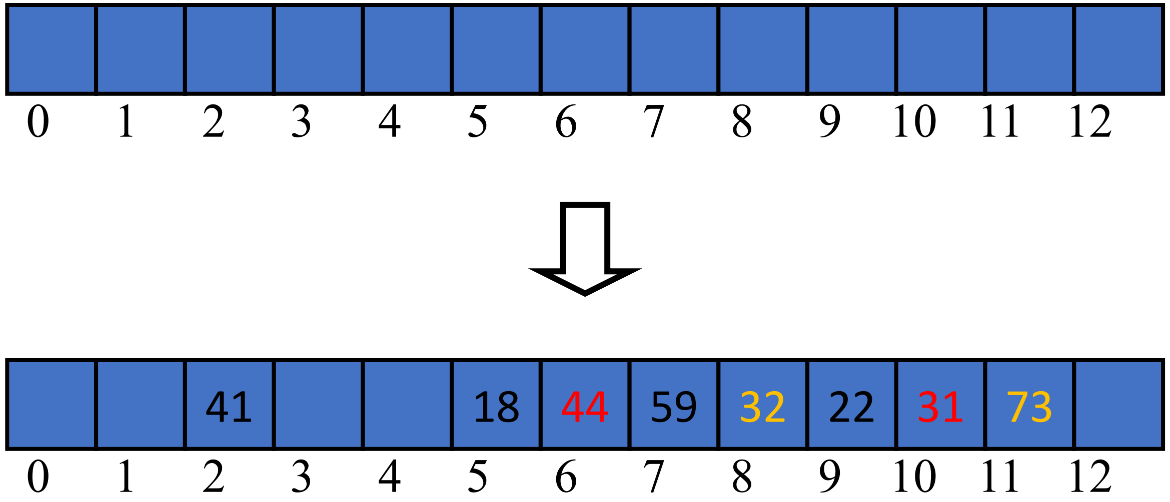
(h(k)+1)%n

(h(k)+2)%n

....

(h(k)+i)%n

until you find an empty slot!



ProbeHashMap

Let’s implement a ProbeHashMap

Open Addressing and Probing

Linear Probing (what we just implemented):

- Keep "*probing*" until you find an empty slot

  (h(k)+1) % n
  (h(k)+2) % n
  ....
  (h(k)+i) % n

- Colliding items cluster together – future collisions to cause a longer sequence of probes

Open Addressing and Probing

Quadratic Probing:

- Keep "*probing*" until you find an empty slot

  (h(k)+f(1)) % n

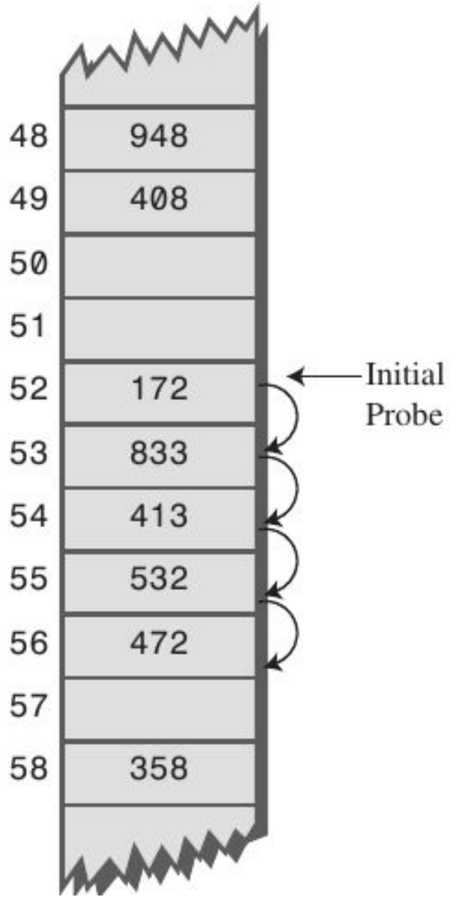
  (h(k)+f(2)) % n

  ....

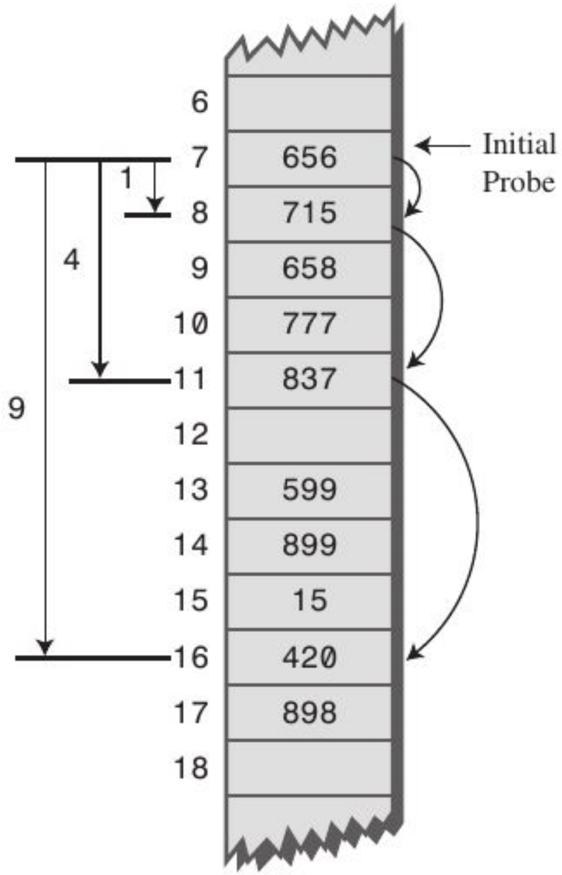
  (h(k)+f(i)) % n

  where $f(i) = i^2$

Linear Probing vs Quadratic Probing



Linear Probing



Quadratic Probing

- Quadratic probing still creates large clusters!
- Unlike linear probing, they are clustered away from the initial hash position
- If the primary hash index is x, probes go to x+1, x+4, x+9, x+16, x+25 and so on, this results in ***Secondary Clustering***

Approach #3: **Double Hashing**

Let's try to avoid clustering.

To probe, let's use a **second hash function**

- Keep "*probing*" until you find an empty slot

  (h(k)+f(1)) % n

  (h(k)+f(2)) % n

  ....

  (h(k)+f(i)) % n

Where f(i) = i * **h'(k)**

Approach #3: **Double Hashing**

Keep "*probing*" until you find an empty slot

$$(h(k)+f(1)) \ \% \ n$$

$$(h(k)+f(2)) \ \% \ n$$

....

$$(h(k)+f(i)) \ \% \ n$$

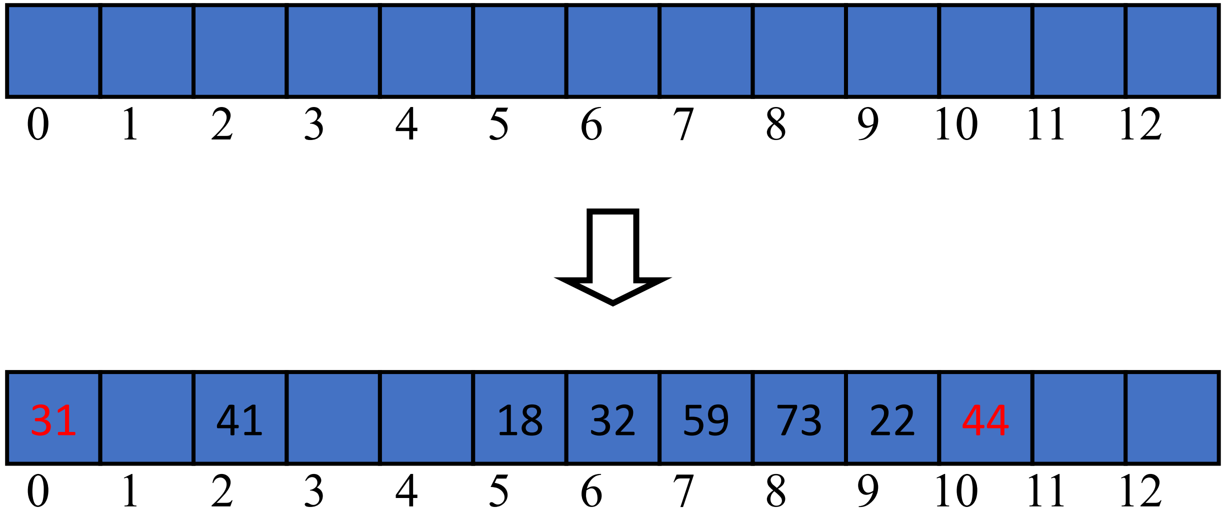
Where $f(i) = i * \mathbf{h'(k)}$

A common choice for $\mathbf{h'(k)} = q - (k \ \% \ q)$
where q is prime and $< n$

Example

k	$h(k)$	h'(k)	Probes		
18	5	3	5		
41	2	1	2		
22	9	6	9		
44	5	5	5	10	
59	7	4	7		
32	6	3	6		
31	5	4	5	9	0
73	8	4	8		

- Insert 18, 41, 22, 44, 59, 32, 31, 73



probe:

(h(k) + f(k)) % n

h(k) = k % 13

f (k) = i * **h'(k)**

h'(k) = 7 - k % 7

Performance Analysis

	ChainHashMap Best Case	ChainHashMap Worst Case	ProbeHashMap Best Case	ProbeHashMap Worst Case
get				
put				
remove				

Which is better in practice?

Open Addressing vs Chaining

- Probing is significantly faster in practice
- locality of references – much faster to access a series of elements in an array than to follow the same number of pointers in a linked list

Performance Analysis

	ArrayMap	HashMap with good hashing and good probing
get		
put		
remove		

Performance of Hashtable

	array	linked list	BST (balanced)	HashTable
search				
insert				
remove				

Load Factor

- HashMaps have an underlying array... what if it gets full?
  - For ChainHashMap collisions increase
  - For ProbeHashMap we need to resize!

- Load Factor = # of elements stored / capacity

- A common strategy is to resize the hash map when the load factor exceeds a predefined threshold (often 0.75)
  - tradeoff between memory and runtime

Outline

Homework Discussion

MergeSort

Homework 7

- NYPD “Stop Question and Frisk” dataset
- How to work with large data

From Wikipedia, the free encyclopedia

A ***Terry* stop** in the United States allows the police to briefly detain a person based on reasonable suspicion of involvement in criminal activity.[1][2] Reasonable suspicion is a lower standard than probable cause which is needed for arrest. When police stop and search a pedestrian, this is commonly known as a **stop and frisk**. When police stop an automobile, this is known as a traffic stop. If the police stop a motor vehicle on minor infringements in order to investigate other suspected criminal activity, this is known as a **pretextual stop**. Additional rules apply to stops that occur on a bus.[3]

Homework 7

- How many times was the same person stopped for questioning?

MergeSort

What sorting algorithms have we seen thus far?

1. Selection sort
   a. How does it work?
   b. Runtime complexity
2. Heap sort
   a. How does it work?
   b. Runtime complexity?

Divide and Conquer algorithm

1. **Divide**: recursively break down the problem into sub-problems
2. **Conquer:** recursively solve the sub-problems
3. **Combine:** combine the solutions to the sub-problems until they are a solution to the entire problem

Binary search is a divide and conquer algorithm

Usually involves recursion

Merge Sort

1. **Divide**: Divide the unsorted list into lists with only one element

2. **Conquer**: merge them back together in a sorted manner

3. **Combine:** merge the sorted sequences

Merge Sort

https://youtu.be/4VqmGXwpLqc?si=WpYuXYLtJOuhvd77&t=24

Merge Sort

Sort a sequence of numbers A, $|A| = n$

Base: $|A| = 1$, then it's already sorted

General

- divide: split A into two halves, each of size $\frac{n}{2}$ ($\left\lfloor\frac{n}{2}\right\rfloor$ and $\left\lceil\frac{n}{2}\right\rceil$)
- conquer: sort each half (by calling mergeSort recursively)
- combine: merge the two sorted halves into a single sorted list

Example

6	8	4	1	7	2	5	3

Example

6	8	4	1	7	2	5	3

6	8	4	1

7	2	5	3

Example

6	8	4	1	7	2	5	3

6	8	4	1

7	2	5	3

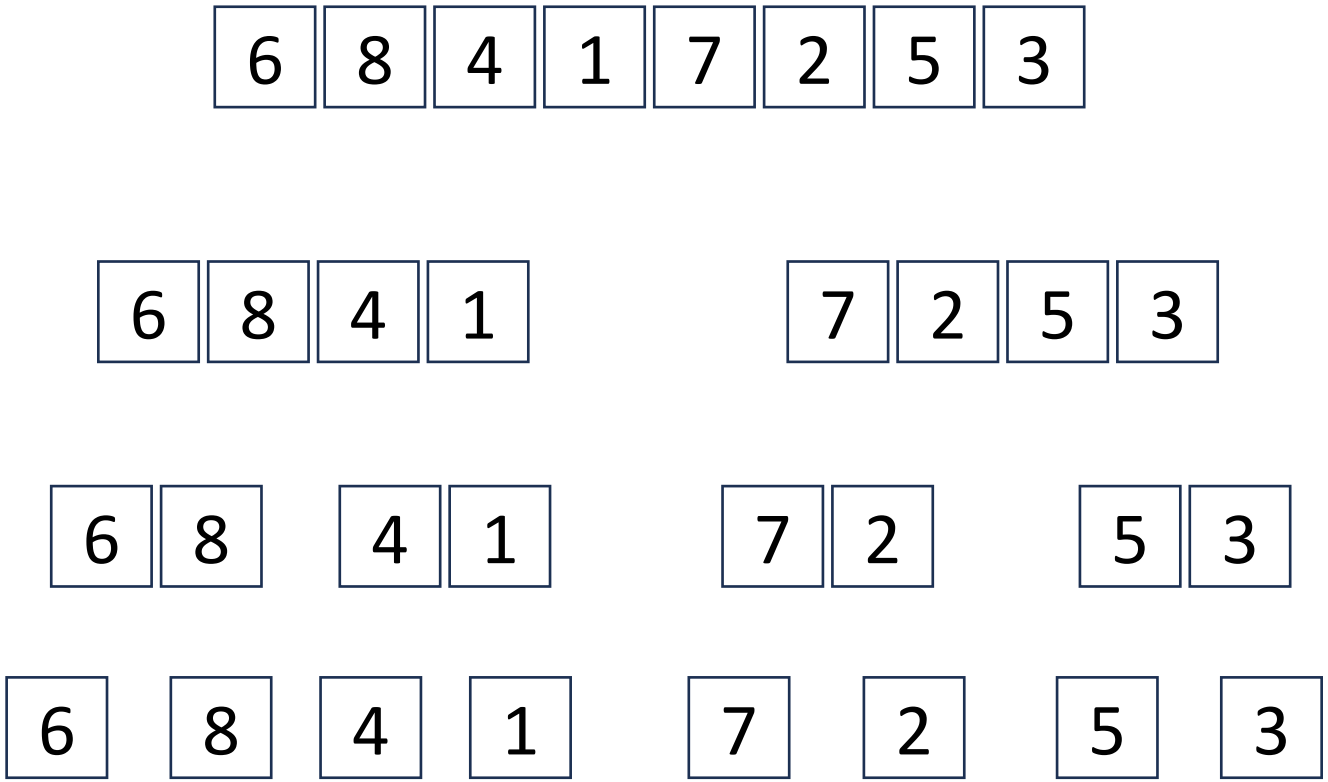
6	8

4	1

7	2

5	3

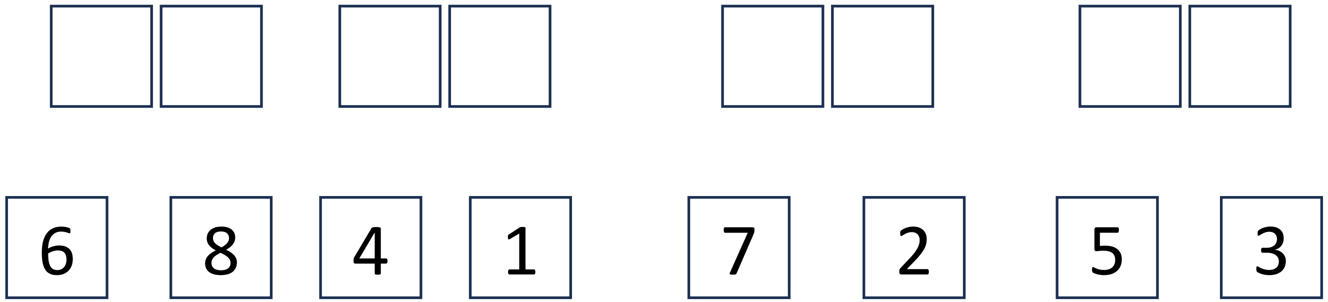
Example



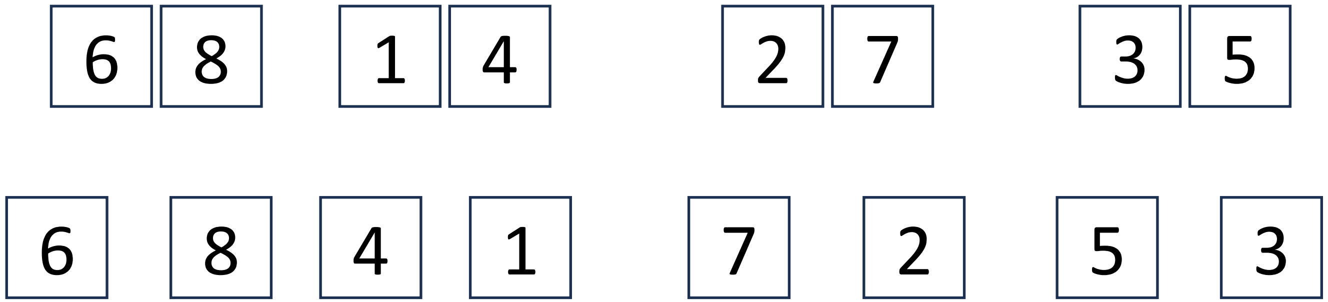
Example

6	8	4	1	7	2	5	3

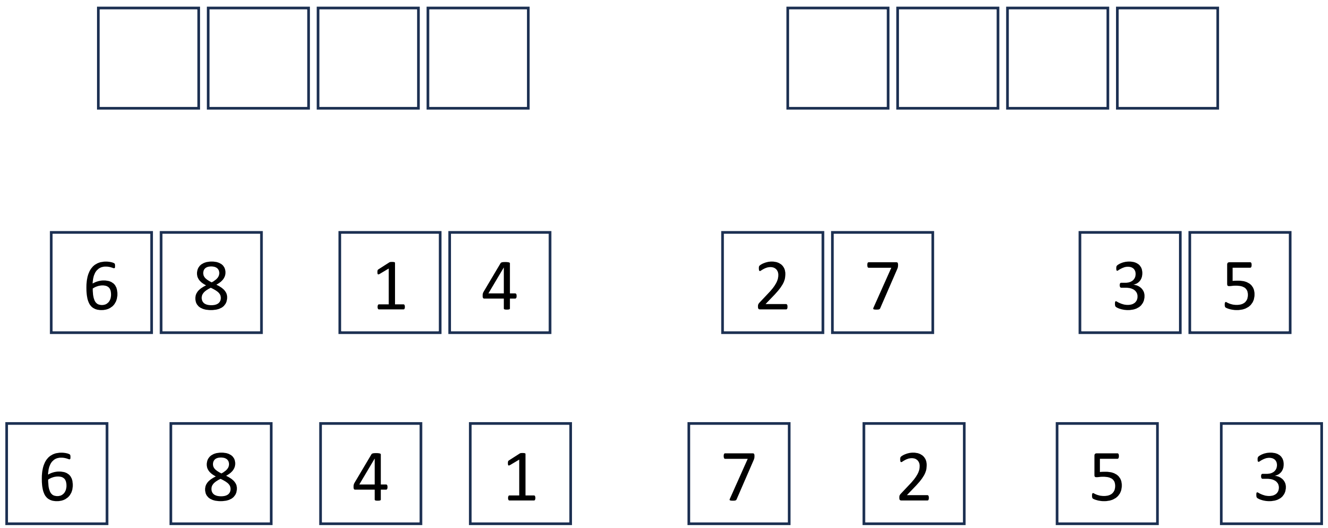
Example



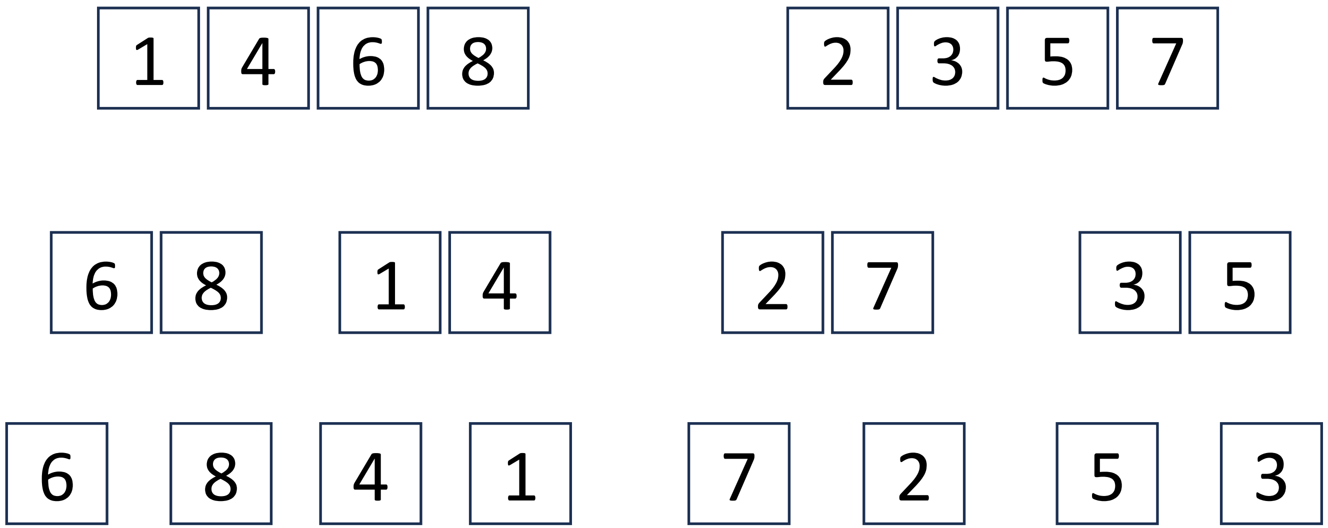
Example



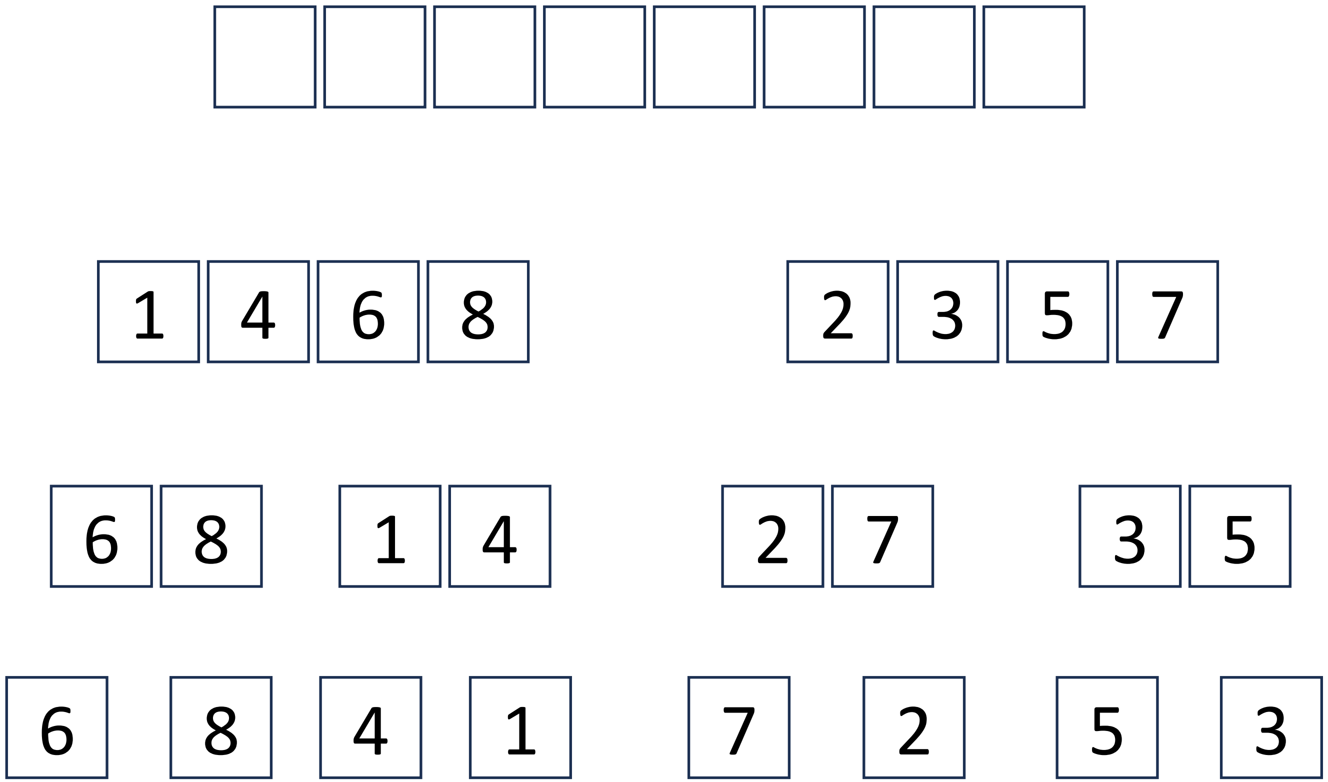
Example



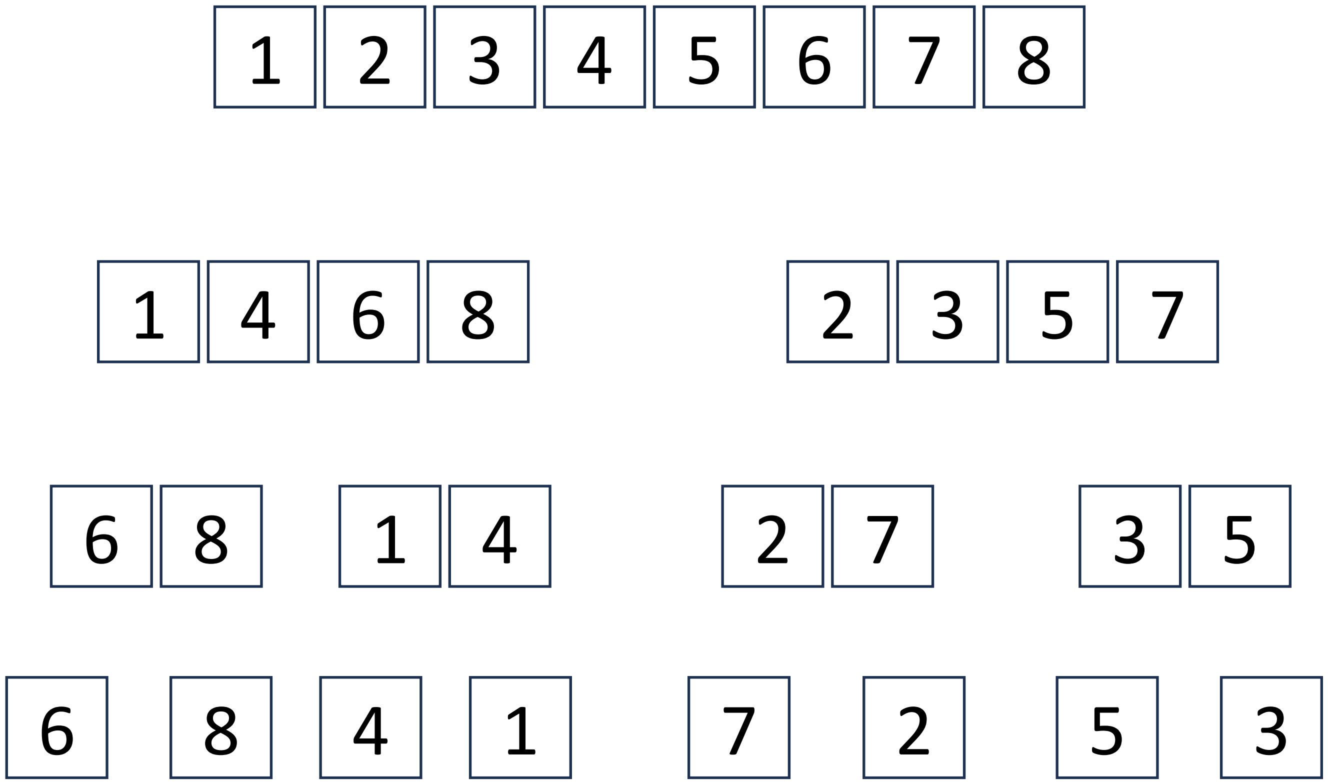
Example



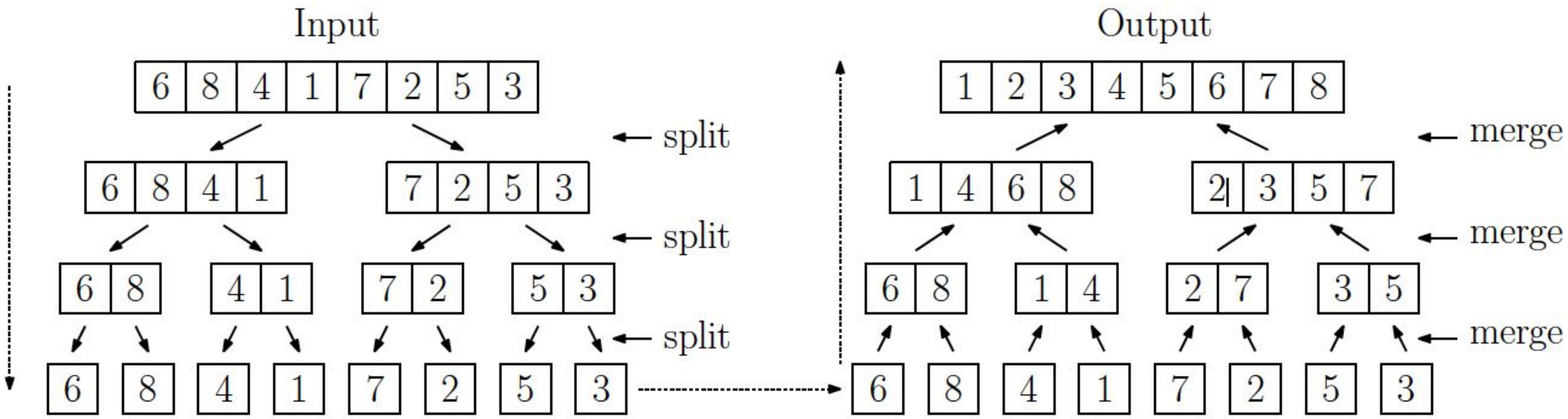
Example



Example



Example - summary



Merge - how do we sort two sorted lists?

```
Algorithm merge(A, B)
  S = []

  while(!A.isEmpty() and !B.isEmpty())
   if A[0] < B[0]
      S.add(A.removeFirst())
   else
      S.add(B.removeFirst())

  while (!A.isEmpty())
      S.add(A.removeFirst())
  while (!B.isEmpty())
      S.add(B.removeFirst())
  return S
```

runtime complexity?

O(n)

where n is A.length + B.length

Merge Sort Implementation

Runtime of MergeSort

Runtime of merging two sorted two lists A, B where |A| + |B| = n :

O(n)

How many times do we merge two sorted lists?

log n times

So total runtime is:

O(n * log(n))

Quicksort

Quicksort

- Divide and conquer
- **Divide:** select a *pivot* and create three sequences:
  a. L: stores elements less than the pivot
  b. E: stores elements equal to the pivot
  c. G: stores elements greater than the pivot
- **Conquer:** recursively sort L and G
- **Combine:** L + E + G is a sorted list

Quick Sort

Sort [2, 6, 5, 3, 8, 7, 1, 0]

1. choose a pivot
2. swap pivot to the end of the array
3. Find two items:
   a. left which is larger than our pivot
   b. right which is smaller than our pivot

4. swap left and right
5. repeat 3 and 4 until right < left
6. swap left and pivot
7. Sort L E and R recursively

Quick Sort - Choosing a pivot

What if we chose our pivot to be 1?

We want a pivot that divides our list as evenly as possible.

Median-of-three: look at the first, middle, and last elems in the array, and pick the middle element.

Quicksort runtime complexity

Bad pivot:

$O(n^2)$

Good pivot:

$O(n \log n)$

Summary of Sorting Algorithms

Algorithm	Time
selection-sort	
heap-sort	
merge-sort	
quick-sort	