

CH-231-A

Algorithms and Data Structures

ADS

Lecture 19

Dr. Kinga Lipskoch

Spring 2022

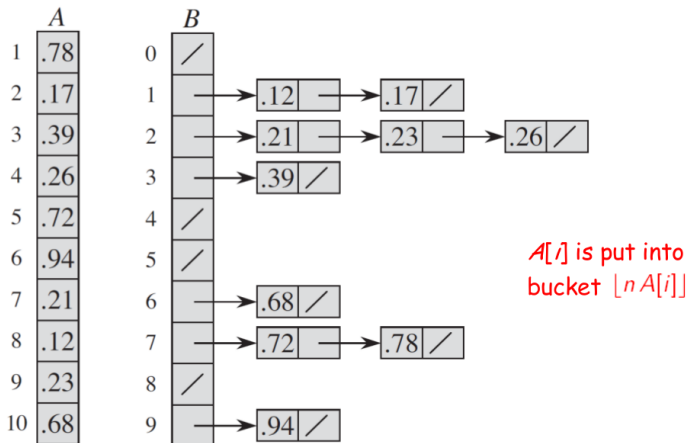
Bucket Sort: Motivation

- ▶ Can we use the idea of Radix Sort to sort any numbers, i.e., without assuming them to be integers?
- ▶ In order to do this efficiently, we make a new assumption:
 - ▶ The to-be-sorted elements shall distribute uniformly and independently over the interval $[0, 1)$.
- ▶ Remark:
 - ▶ Interval $[0, 1)$ is not a real restriction, as we can normalize the elements to this interval in linear time.
 - ▶ However, uniform distribution and independence are restrictions and we will see that we need this to assure good expected running time.

Bucket Sort: Idea

- ▶ Assuming that we have to sort n numbers, we split the interval $[0, 1)$ into n subintervals or buckets.
- ▶ Then, we can distribute the n numbers to the n buckets.
- ▶ Assuming uniform distribution, we can conclude that we have only few numbers falling into each bucket.

Bucket Sort: Example $n = 10$



Bucket Sort: Pseudocode

BUCKET-SORT(A)

- 1 let $B[0..n-1]$ be a new array
- 2 $n = A.length$
- 3 **for** $i = 0$ **to** $n - 1$
- 4 make $B[i]$ an empty list
- 5 **for** $i = 1$ **to** n
- 6 insert $A[i]$ into list $B[\lfloor nA[i] \rfloor]$
- 7 **for** $i = 0$ **to** $n - 1$
- 8 sort list $B[i]$ with insertion sort
- 9 concatenate the lists $B[0], B[1], \dots, B[n-1]$ together in order

Bucket Sort: Time Complexity

BUCKET-SORT(A)

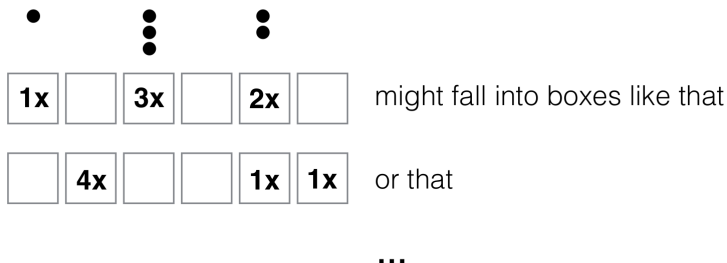
```
1  let  $B[0 \dots n - 1]$  be a new array
2   $n = A.length$ 
3  for  $i = 0$  to  $n - 1$ 
4      make  $B[i]$  an empty list
5  for  $i = 1$  to  $n$ 
6      insert  $A[i]$  into list  $B[\lfloor n A[i] \rfloor]$ 
7  for  $i = 0$  to  $n - 1$ 
8      sort list  $B[i]$  with insertion sort
9  concatenate the lists  $B[0], B[1], \dots, B[n - 1]$  together in order
```

Time complexity:

$$T(n) = \Theta(n) + \sum_{i=0}^{n-1} O(n_i^2),$$

where n_i denotes the number of elements in bucket i .

Bucket Sort: Average Case



Bucket Sort: Expected Time Complexity (1)

- ▶ $E[T(n)] = \Theta(n) + \sum_{i=0}^{n-1} O(E[n_i^2])$
- ▶ What is $E[n_i^2]$?
- ▶ Let X_{ij} be the event that $A[j]$ falls into bucket i .
- ▶ Then, $n_i = \sum_{j=1}^n X_{ij}$
- ▶ Use assumptions of uniform distribution and independence.

Bucket Sort: Estimate $E[n_i^2]$ (1)

$$\begin{aligned} E[n_i^2] &= E \left[\left(\sum_{j=1}^n X_{ij} \right)^2 \right] = E \left[\sum_{j=1}^n \sum_{k=1}^n X_{ij} X_{ik} \right] \\ &= E \left[\sum_{j=1}^n X_{ij}^2 + \sum_{j=1}^n \sum_{\substack{k=1 \\ k \neq j}}^n X_{ij} X_{ik} \right] \\ &= \sum_{j=1}^n E[X_{ij}^2] + \sum_{j=1}^n \sum_{\substack{k=1 \\ k \neq j}}^n E[X_{ij} X_{ik}] \\ &= \sum_{j=1}^n E[X_{ij}^2] + \sum_{j=1}^n \sum_{\substack{k=1 \\ k \neq j}}^n E[X_{ij}] E[X_{ik}] \end{aligned}$$

Bucket Sort: Estimate $E[n_i^2]$ (2)

$$E[X_{ij}] E[X_{ik}] = \frac{1}{n} \cdot \frac{1}{n} = \frac{1}{n^2}.$$

$$E[X_{ij}^2] = 1^2 \cdot \frac{1}{n} + 0^2 \cdot \left(1 - \frac{1}{n}\right) = \frac{1}{n}.$$

$$\begin{aligned} E[n_i^2] &= \sum_{j=1}^n E[X_{ij}^2] + \sum_{j=1}^n \sum_{\substack{k=1 \\ k \neq j}}^n E[X_{ij}] E[X_{ik}] \\ &= \sum_{j=1}^n \frac{1}{n} + \sum_{j=1}^n \sum_{\substack{k=1 \\ k \neq j}}^n \frac{1}{n^2} \\ &= \frac{n}{n} + n(n-1) \frac{1}{n^2} \\ &= 2 - \frac{1}{n}. \end{aligned}$$

Bucket Sort: Expected Time Complexity (2)

- ▶ $E[T(n)] = \Theta(n) + \sum_{i=0}^{n-1} O(E[n_i^2])$
- ▶ Based on the previous estimation we have the following
- ▶ $E[T(n)] = \Theta(n) + n \cdot O(2 - 1/n) = \Theta(n)$

Searching Problem

- ▶ Given a sorted sequence.
- ▶ Find an element in that sequence.
- ▶ Example:

- ▶ Sequence

3	5	7	8	9	12	15
---	---	---	---	---	----	----

- ▶ Find element 9.
- ▶ Brute-force approach (going through the sequence from start until we find the 9) runs in $O(n)$.

Binary Search

Idea: Use a Divide & Conquer strategy.

1. **Divide:** Check middle element.
2. **Conquer:** Recursively search one subarray.
3. **Combine:** Nothing to be done.

Binary Search: Example (Find 9)

3	5	7	8	9	12	15
---	---	---	---	---	----	----

3	5	7	8	9	12	15
---	---	---	---	---	----	----

3	5	7	8	9	12	15
---	---	---	---	---	----	----

3	5	7	8	9	12	15
---	---	---	---	---	----	----

3	5	7	8	9	12	15
---	---	---	---	---	----	----

3	5	7	8	9	12	15
---	---	---	---	---	----	----

Binary Search: Time Complexity

$$T(n) = 1T(n/2) + \Theta(1)$$

$$a = 1, b = 2$$

$$n^{\log_b a} = n^{\log_2 1} = 1$$

$$f(n) = \Theta(1)$$

$$\text{Case 2: } T(n) = \Theta(\lg n)$$

Summary

- ▶ Sorting problem:
 - ▶ Comparison sorts:
 - ▶ InsertionSort: $\Theta(n)$ [best], $\Theta(n^2)$ [average & worst].
 - ▶ Merge Sort: $\Theta(n \lg n)$.
 - ▶ Heap Sort: $\Theta(n \lg n)$ – heap as a data structure
 - ▶ Quicksort: $\Theta(n \lg n)$ [best & average], $\Theta(n^2)$ [worst].
 - ▶ Decision trees: Worst case does not get better than $\Theta(n \lg n)$.
 - ▶ Sorting in linear time:
 - ▶ Counting Sort: small integers
 - ▶ Radix Sort: large integers
 - ▶ Bucket Sort: any numbers, but uniform distribution.
- ▶ Searching Problem:
 - ▶ Linear Search: $\Theta(1)$ [best], $\Theta(n)$ [average & worst]
 - ▶ Binary Search: $\Theta(1)$ [best], $\Theta(\lg n)$ [average & worst]

Data Structure

Definition:

A data structure is a way to store and organize data in order to facilitate access and modification.

Examples we have seen so far:

- ▶ Array
- ▶ Heap
- ▶ Max-priority queue
- ▶ Linked list

Array (1)

- ▶ **Definition:**

An array is a random-access data structure consisting of a collection of elements, each identified by an index or key.

- ▶ The simplest type of data structure is a linear array, where the indices are one-dimensional.
- ▶ A dynamic array refers to an array which can change its size.

Array (2)

Examples of operations:

- ▶ Getting or setting the value at a particular index:
 - ▶ constant time
- ▶ Iterating over the elements in order:
 - ▶ linear time
- ▶ Inserting or deleting an element:
 - ▶ beginning – linear time
 - ▶ middle – linear time
 - ▶ end – constant time

Dynamic Set

- ▶ In the following, we assume that we are interested in storing and handling dynamic sets.
- ▶ Dynamic sets are sets of elements that can change their size.
- ▶ Elements are identified by a key from a totally ordered set.

Dynamic Set: Operations

Two categories of operations:

- ▶ Queries return the information of a stored object.
- ▶ Modify operations alter the set.

Examples for Queries

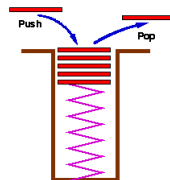
- ▶ *Search*(S, k):
 - ▶ returns element $x \in S$ with $\text{key}[x] = k$ (nil if not existent).
- ▶ *Minimum*(S):
 - ▶ returns element $x \in S$ with smallest $\text{key}[x]$.
- ▶ *Maximum*(S):
 - ▶ returns element $x \in S$ with largest $\text{key}[x]$.
- ▶ *Successor*(S, x):
 - ▶ returns for element $x \in S$ the next-larger element in S (nil if x is element with largest key).
- ▶ *Predecessor*(S, x):
 - ▶ returns for element $x \in S$ the next-smaller element in S (nil if x is element with smallest key).

Examples for Modify Operations

- ▶ *Insert*(S, x):
 - ▶ adds element x to dynamic set S (S grows).
- ▶ *Delete*(S, x):
 - ▶ deletes element x from dynamic set S (S shrinks).

Stack

- ▶ Elementary dynamic data structure.
- ▶ Implements idea of dynamic set.
- ▶ Idea follows that of a coin stacker.
- ▶ Delete operation is called **pop**.
- ▶ Insert operation is called **push**.
- ▶ LIFO principle (Last In First Out):
The element that is returned by the **pop** operation is the last one that has been added (via **push**).



Stack Operations

- ▶ Queries:
 - ▶ *Stack-Empty*(S):
True iff stack S is empty.
 - ▶ ...
- ▶ Modify operations:
 - ▶ *Push*(S, x):
Add element x on top of stack S and push other elements down.
 - ▶ *Pop*(S):
If stack is non-empty, remove top-most element and return it.

Stack: Implementation as an Array

$S.top$ is the index of the top of the stack

STACK-EMPTY(S)

```
1  if  $S.top == 0$   
2      return TRUE  
3  else return FALSE
```

PUSH(S, x)

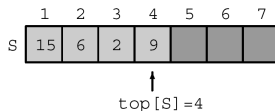
```
1   $S.top = S.top + 1$   
2   $S[S.top] = x$ 
```

POP(S)

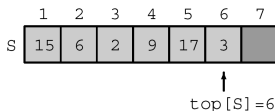
```
1  if STACK-EMPTY( $S$ )  
2      error "underflow"  
3  else  $S.top = S.top - 1$   
4      return  $S[S.top + 1]$ 
```

Stack: Example (Array Implementation)

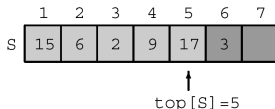
- Stack with four elements:



- Performing operations $\text{Push}(S, 17)$ and $\text{Push}(S, 3)$:



- Performing operation $\text{Pop}(S)$ returning entry 3:



Stack Operations: Complexity

STACK-EMPTY(S)

```
1  if  $S.top == 0$   
2      return TRUE  
3  else return FALSE
```

PUSH(S, x)

```
1   $S.top = S.top + 1$   
2   $S[S.top] = x$ 
```

POP(S)

```
1  if STACK-EMPTY( $S$ )  
2      error "underflow"  
3  else  $S.top = S.top - 1$   
4      return  $S[S.top + 1]$ 
```

Complexity:

when implemented as an array all operations are $O(1)$.

Stack Operations: Underflow and Overflow

- ▶ If we want to perform a *Pop*-operation on the empty stack, we have a **stack-underflow** situation.
- ▶ We may also have a **stack-overflow** situation, if we assume that the stack has a maximum amount of entries and then we try to perform a *Push*-operation (not considered in the array implementation).