CS218: Design and Analysis of algorithms

Amortized analysis

Yan Gu

Monte Carlo algorithm

- Gambles with correctness but not time

```
repeat 300 times:
    k = RandInt(n)
    if A[k] = 1, return k
return "Failed"
```

$$\mathbf{Pr}[\text{failure}] = \frac{\mathbf{1}}{\mathbf{2^{300}}}$$

Worst-case time complexity: $O(1)$

Las Vegas algorithm

- Gambles with time but not correctness

```
repeat:
    k = RandInt(n)
    if A[k] = 1, return k
```

$$\mathbf{Pr}[\text{failure}] = \mathbf{0}$$

- Worst-case time: cannot bound (can be big when super unlucky)
- Expected time: $O(1)$ (2 iterations)

Types of algorithms

	Correctness	Time complexity
Deterministic	Always	Good
Monte Carlo	With good probability	Ideally even better
Las Vegas	Always	Better

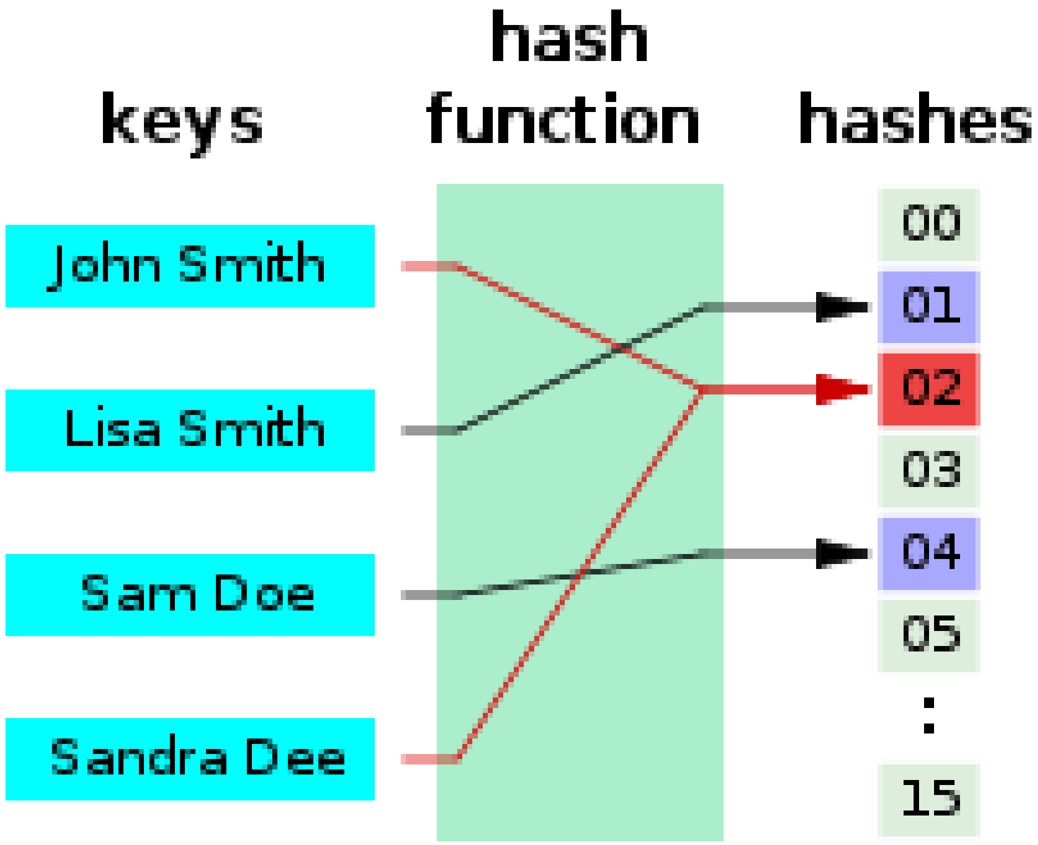
Hashing and Rabin-Karp Algorithm

Hash function

- **Maps arbitrary data to fixed-size values**
  - Usually integers
- **The same data are always mapped to the same value**
- **Different data are unlikely to be mapped to the same value**
  - Collision: two keys are hashed to the same hash value

How to design a hash function

- **E.g., Strings -> integers**
- **How can we map complicated structs**
  - A pair: (i, j) for i, j in [1..100]
  - A triple (i, j, k) for i, j, k in [1..100]
  - A 2×2 matrix?
  - A string?



Application: Rabin-Karp algorithm

- **Substring matching**
- **Given string $X[1..n]$ (text) and $Y[1..m]$ (pattern), we want to check if Y is a substring in X**
  - $X = abcabababc, Y = caba$

- **The naïve solution cost $O(nm)$ time**

- **Knuth–Morris–Pratt algorithm (KMP) can solve this in $O(n)$ time**
  - Hard to understand ☹

Let's try randomization!

- **To check if Y appears in X, we just need to check all X's substring and see if they are the same with Y**
  - Checking if $X[s..e] = Y$ takes $O(m)$ time

- **Let's use hashing!**
  - Check if the hash value of $X[s..e]$ equals to the hash value of Y

Let's try randomization!

- **To check if Y appears in X, we just need to check all X's substring and see if they are the same with Y**
  - Checking if $X[s..e] = Y$ takes $O(m)$ time

- **Let's use hashing!**
  - For a string using characters a, b, c
  - H_1: using a=1, b=2, c=3. adding everything up

$$Y = bcab, H_1(Y) = 8$$

a	b	c	a	b	a	b	c

abca	1+2+3+1=7	abab	1+2+1+2=6
bcab	2+3+1+2=8	babc	2+1+2+3=8
caba	3+1+2+1=7		

Let's try randomization!

$Y = bcab,\ H_1(Y) = 8$

- **How to compute the hash value quickly?**

a	b	c	a	b	a	b	c

abca	1+2+3+1=7
bcab	2+3+1+2=8
caba	3+1+2+1=7
abab	1+2+1+2=6
babc	2+1+2+3=8

Hash value:

1+2+3+1=7
7-1+2=8
8-2+1=7
7-3+2=6
6-1+3=8

$O(n)$ time!

Fewer collisions?

$Y = bcab,\ H_2(Y) = 2312$

- H_2: treat a=1,b=2,c=3, use a decimal number

a	b	c	a	b	a	b	c

$O(n)$ time!

abca	1231
bcab	2312
caba	3121
abab	1212
babc	2123

Hash value:

1231

(1231-1000)*10+2=2312

(2312-2000)*10+1=3121

(3121-3000)*10+2=1212

(1212-2000)*10+3=2123

A simple version

```
num (c) {return c-98;} // a=1, b=2, c=3

check_match (X, Y) {
  hy = 0;
  for (i = 1..m) hy = hy*10 + num(Y[i]); //compute hash value of Y
  hx = 0;
  for (i = 1..n) {
    if (i < m) hx = hx*10 + num(X[i]); // process the first (m-1) characters
    else {
       if (i==m) hx = hx*10 + num(X[i]);
       else hx = (hx - X[i-m+1] * pow(10, m-1))*10 + num(X[i]); // compute new hash value
       if (hx == hy)
          if (check(X[i-m+1 .. i], Y))  return true;
    }
 } }
```

More characters?

- **What happens if we have 26 letters? Or even more?**
  - Use base-26 (or base-x with x > 26)

- **If Y has 100 characters?**
- **Cannot use an integer to store?**

- **We can use $\boldsymbol{H_3(s) = H_2(s) \,\%\, p}$ for some big prime $\boldsymbol{p}$**
  - Still, the same strings will be mapped to the same value
  - Different strings are likely to be mapped to different values
- $\boldsymbol{(a+b) \,\%\, p = (a \,\%\, p) + (b \,\%\, p)}$
- $\boldsymbol{(a \times b) \,\%\, p = (a \,\%\, p) \times (b \,\%\, p)}$

The cost of the algorithm?

- $O(n)$ time to compute and compare all hash values
- But if two hash values are the same, we need to verify the strings are equal or not
  - $O(m)$ time
- In the worst case, all comparisons succeed, we need $O(nm)$ time

- However, the probability of two different strings is mapped to the same value is $1/p$, expected cost is $O\left(\frac{mn}{p} + n\right)$
- We can use a large p to decrease the cost
- We can also use two independent hash functions to even lower the chance of collision

CS218: Design and Analysis of algorithms

Amortized analysis

Yan Gu

Amortized analysis

- A somehow different angle to analyzing algorithms (very different from the “standard” way of worst-case analysis)
- May be new for many of you, so you may need to take some time after class to understand the idea

Hashing and hash table

Hash function

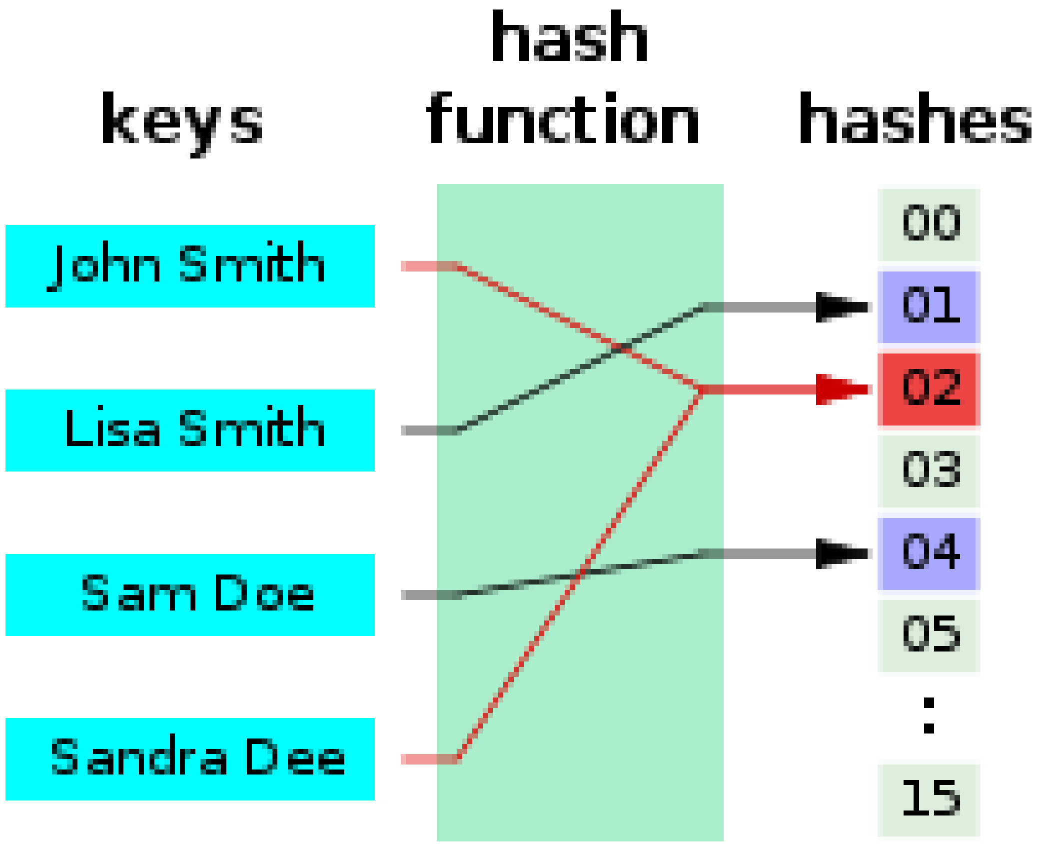
- **Maps arbitrary data to fixed-size values**
  - Usually integers
- **The same data are always mapped to the same value**
- **Different data are unlikely to be mapped to the same value**
  - Collision: two keys are hashed to the same hash value

A short recall of hash table

- Maintain a list of unordered elements, and support quick insert/delete and lookup
- Used in unordered_set/unordered_map (C++), HashMap (Java), Dictionary (Python)

Hash table and hash functions

- E.g., Strings -> integers
- A["John Smith"] = false
  - A[2] = false (John Smith)
- A["Lisa Smith"] = true
  - A[1] = true (Lisa Smith)
- X = A["John Smith"]
  - X = A[2]
- A["Sandra Dee"] = false
  - A[2] ?



Ways to deal with collisions

- **When an element is mapped to a index i, but finds out that position i has been taken by another element?**

- **Open addressing / closed hashing**
  - Find another empty position (e.g., linear probing: try the next position)
  - It can also be using other ways to find the next empty position, not necessarily try the next position (e.g., probe quadratically)

- **Closed addressing / open hashing**
  - Throw the element still to position i
  - All elements in position i will be further organized as another data structure, e.g., a linked list

Open addressing vs. closed addressing

Open Addressing

Characteristic structure (colors denote "home" bucket):

0:
1: 1
2: 2
3: 2
4: 2
5: 4
6:
7: 7
8: 7
9: 9

Closed Addressing

Characteristic structure (colors denote "home" bucket):

0:
1: 1
2: 2 2 2
3:
4: 4
5:
6:
7: 7 7
8:
9: 9

Source: https://programming.guide/hash-tables-open-vs-closed-addressing.html

Simple uniform hashing strategy

- **For each element with key x, find a random position $h_1(x)$;**
  - if there's a collision, try another (i.e., $h_2(x)$)

- **Say insert key to be 5**
- **Then insert key to be 88**

- **What's the expected number of retries?**

13	-	-	45	-	5	-	-	74	-	88	20

Analyzing expected number of retries

- **Assuming at least half of the elements in the hash table are empty**
  - Recall this is the "load factor" r of a hash table, and presumably should be less than ½
- **What's the probability that we need the first try?**
- **What's the probability that we need the first retry?**
- **What's the probability that we need the second retry?**
- **What's the probability that we need the k-th retry?**

- **Total number of retries** $\leq 1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \cdots = 2 = O(1)$

13	**-**	**-**	**45**	**-**	**-**	**-**	**-**	**74**	**-**	**-**	**20**

Conclusion for hash table analysis

- Using this simple strategy, we can show that insert and lookup has constant ($O(1)$) cost in expectation
- Similar for other probing strategies

What happens if the hash table is too full?

In the above analysis, it’s very important that the hash table load factor is a below 1/2

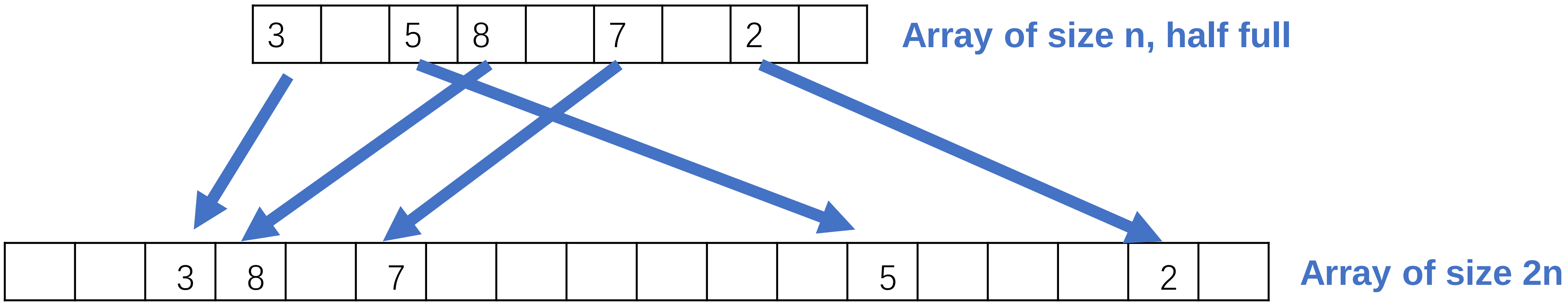
- That’s why we know that an insertion succeeds with probability ½
- So the cost is 1/2+1/4+1/8+… = O(1)

- Actually, any constant works

- But, how can we guarantee that?

Resizing hash table

- When the load factor of the hash table is more than ½
- i.e., when we have more than n/2 elements in the hash table of size n
- Resize the table



What is the cost of an insertion?

- **Worst case cost: $\boldsymbol{O(n)}$?**
  - n is the current table size

- **That's too expensive…?**

- **Is the worst-case cost of n insertions $O(n^2)$?**

What is the cost of an insertion?

- **Worst case cost: $O(n)$?**
  - n is the current table size
- **However, this happens very rarely - at least every $O(n)$ insertions!**

	Initial total size	Current #slots filled	#insertions before resizing	Resizing cost
Phase 1	k	0	k/2	k/2
Phase 2	2k	k/2	k/2	k
Phase 3	4k	k	k	2k
Phase 4	8k	2k	2k	4k
……				

Happens after k/2 insertions

Happens after k/2 insertions

Happens after k insertions

Happens after 2k insertions

- **All the rest of the insertions cost $O(1)$!**
- **Every t insertions, we have an insertion of cost $O(t)$**
- **Somehow "on average", the cost is still a constant?**

(Assume unit cost per insertion and per rehash)

Amortized Analysis

Amortized analysis

- **Some operations are expensive, some of them are cheap**
- **The amortized analysis considers both the costly and less costly operations together over the whole sequence of operations**

- **For a resizable hash table:**
  - In every k insertions, we have an insertion cost O(k)
  - All other insertions have cost O(1)
  - In any sequence of k insertions, the total cost is O(k)!

Amortized analysis

- You must perform a series of operations
- We analyze the cost per operation

- If we look at a sequence of k operations, the total cost is f(k)
- Then the “amortized” cost for each operation is f(k)/k

- Some operations may be expensive
- But we can charge its work to some previous operations

- A common way to think about it: a bad case happen only after enough number of good cases happen!

What is the cost of an insertion?

	Initial total size	Current #slots filled	#insertions before resizing	Resizing cost
Phase 1	k	0	k/2	k/2
Phase 2	2k	k/2	k/2	k
Phase 3	4k	k	k	2k
Phase 4	8k	2k	2k	4k
……				

Happens after k/2 insertions
Happens after k/2 insertions
Happens after k insertions
Happens after 2k insertions

- The total cost of the first k/2 insertions is k/2 + k/2
- The total cost of the first k insertions is < 3k
- The total cost of the first 2k insertions is < 6k
- The total cost of every t consecutive insertion is O(t)
- The amortized cost of an insertion is a constant!

Binary Counter

Binary counter

- **You need to flip the scoreboard from 0 to n, how many times you need to flip it?**
  - You need to flip every digit

- **Let's first consider a simple case, what if it's a binary counter?**

Binary counter

- Binary number
- Count from 0 to n
- You pay 1 dollar when you flip a bit (0 to 1 or 1 to 0)
- How much do we need to pay?
- 000 -> 001 -> 010 -> 011 -> 100 -> 101 -> 110 -> 111

- Let’s guess!
    - A. $O(\log n)$
    - B. $O(\sqrt{n})$
    - C. $O(n)$
    - D. $O(n \log n)$
    - E. $O(n^2)$

Binary counter

- **0000**
- **0001**
- **0010**
- **0011**
- **0100**
- **0101**
- **0110**
- **0111**

- **1000**
- **1001**
- **1010**
- **1011**
- **1100**
- **1101**
- **1110**
- **1111**

- **The last digit changes every time**
  - n times
- **The second last digit changes every other time**
  - n/2 times
- **The third last digit changes every 4 times**
  - n/4 times
- **…**

- **In total $O(n)$ time**

How many flips we need for every increment?

- xxxxx0 -> xxxxx1
- xx01111 -> xx10000
- Not a fixed number – can be really bad!
- Sometimes we need to flip a lot of bits… a bad case?
- Let’s “amortize” the cost!

Piggy bank

- **Let’s consider every bit**

- **Every time, when it change from 0 to 1, pay 2 dollars!**
    - 1 dollar for changing this bit from 0 to 1 immediately
    - 1 dollar to save in its piggy bank, use it to pay when it’s changed back from 1 to 0

- **So we only need to pay the costs for 0 -> 1!**

Binary counter

	A[3]	A[2]	A[1]	A[0]	Paid
Counter	**0**	**0**	**0**	**0**	
Bank balance	0	0	0	0	

Only pay when a bit changes **from 0 to 1**, but pay $2!

$1 for changing this bit from 0 to 1

$1 in bank. Later use it when it's changed back from 1 to 0

Binary counter

	A[3]	A[2]	A[1]	A[0]	Paid
Counter	**0**	**0**	**0**	**0**	
Bank balance	0	0	0	0	
Counter	**0**	**0**	**0**	**1**	
Bank balance	0	0	0	1	2
Counter	**0**	**0**	**1**	**0**	
Bank balance	0	0	1	0	2
Counter	**0**	**0**	**1**	**1**	
Bank balance	0	0	1	1	2
Counter	**0**	**1**	**0**	**0**	
Bank balance	0	1	0	0	2
Counter	**0**	**1**	**0**	**1**	
Bank balance	0	1	0	1	2

	A[3]	A[2]	A[1]	A[0]	Paid
Counter	**0**	**1**	**0**	**1**	
Bank balance	0	1	0	1	2
Counter	**0**	**1**	**1**	**0**	
Bank balance	0	1	1	0	2
Counter	**0**	**1**	**1**	**1**	
Bank balance	0	1	1	1	2
Counter	**1**	**0**	**0**	**0**	
Bank balance	1	0	0	0	2

Only pay when a bit changes **from 0 to 1**, but pay $2!

- $1 for changing this bit from 0 to 1
- $1 in bank. Later use it when it's changed back from 1 to 0

Piggy bank

- **Let's consider every bit**

- **Every time, when it change from 0 to 1, pay 2 dollars!**
  - 1 dollar for changing this bit from 0 to 1 immediately
  - 1 dollar to save in its piggy bank, use it to pay when it's changed from 1 to 0

- **So we only need to pay the costs for 0 -> 1!**

- **How many 0s will be changed to 1 every time?**

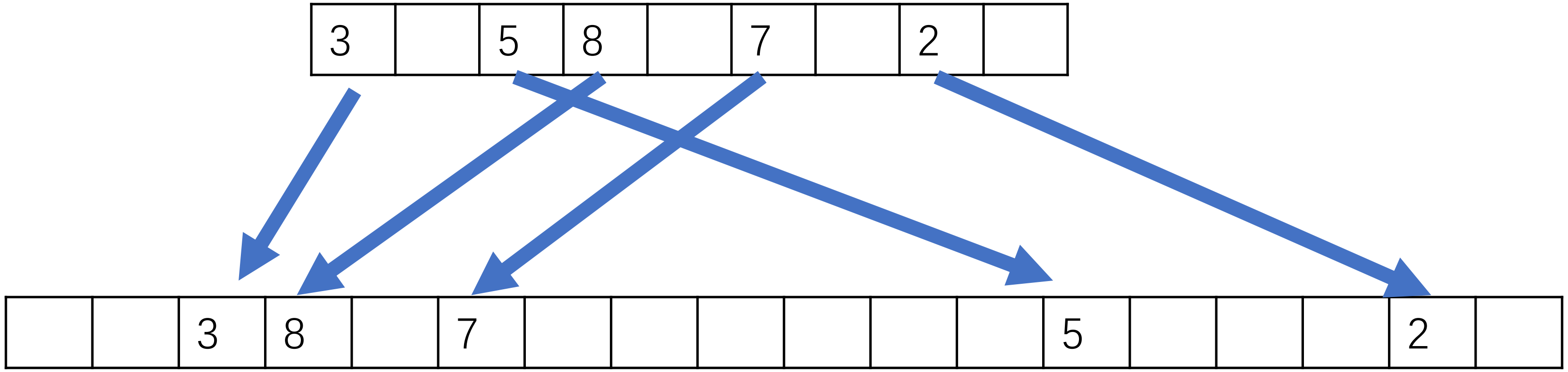
Piggy bank

- So we only need to pay the costs for 0 -> 1!
- How many 0s will be changed to 1 every time?
- There will be only one 0 changed to 1 in each increment!
- Why? Because every time we encounter a 0, we change it to 1, and stop
- In total the cost is $O(n)$

“Piggy bank” analysis for resizable hash tables

Resizing hash table

- When the load factor of the hash table is more than ½
- i.e., when we have more than n/2 elements in the hash table of size n
- Resize the table



What is the cost of an insertion?

- **Although every insertion may need $O(n)$ cost**

- **Resizing happens only after $O(n)$ insertions!**

- **Let every insertion "pay" more dollars!**
  - One for the insertion itself (used immediately)
  - More for moving the elements to the next hash table (save in the bank)
  - How much do you need?

  - When we resize, we have enough money in our piggy bank

What is the cost of an insertion?

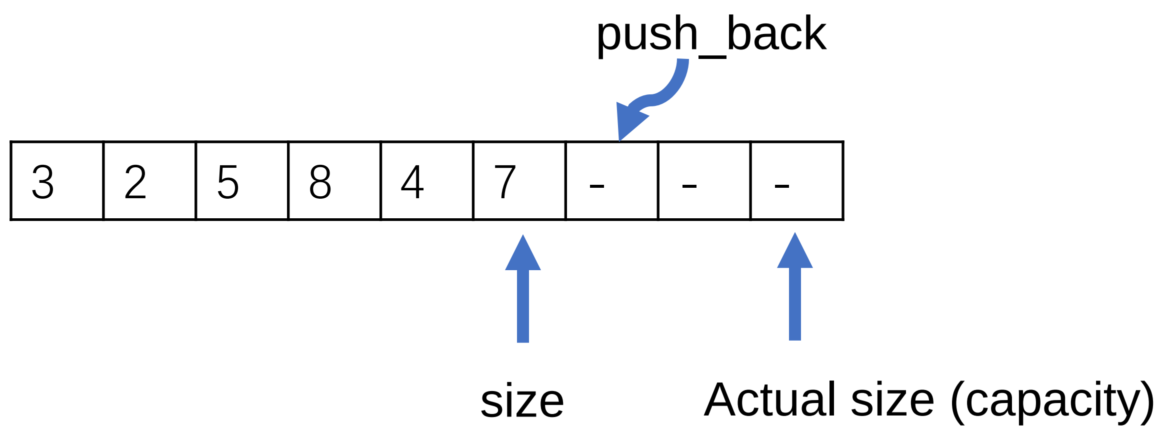
	Initial total size	Current #slots filled	#insertions before resizing	Saved money	Resizing cost
Phase 1	k	0	k/2	k/2 * 2 = k	k/2
Phase 2	2k	k/2	k/2	k/2 * 2 = k	k
Phase 3	4k	k	k	k * 2 = 2k	2k
Phase 4	8k	2k	2k	2k * 2 = 4k	4k
……					

- **Let every insertion "pay" 3 dollars!**
  - One for the insertion itself (used immediately)
  - One for moving the element itself to the next hash table (save in the bank)
  - One for moving some other element to the next hash table (save in the bank)

  - When we resize, we have enough money in our piggy bank

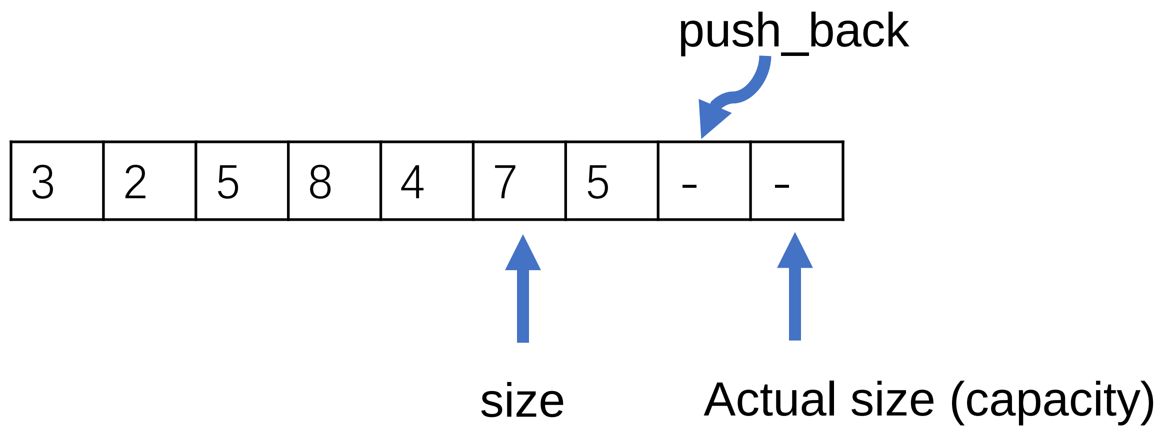
Similar data structures

- **Resizable arrays**
- **When we write code, usually we specify the array size when we allocate memory, but what happens if we do not know the upper bound of the array size?**
    - E.g., “std::vector” in C++
    - push_back: add to the end
    - size(): get the current size



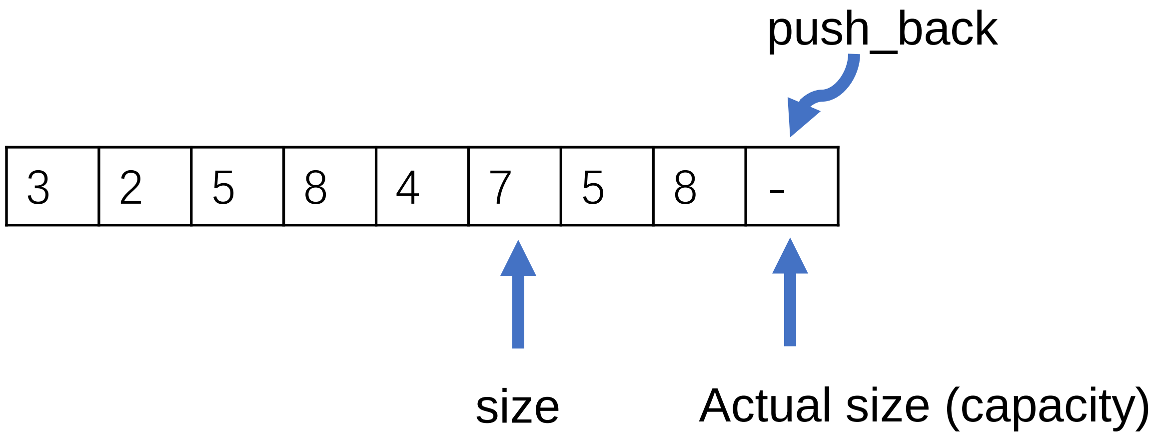
Similar data structures

- **Resizable arrays**
- **When we write code, usually we specify the array size when we allocate memory, but what happens if we do not know the upper bound of the array size?**
  - E.g., “std::vector” in C++
  - push_back: add to the end
  - size(): get the current size



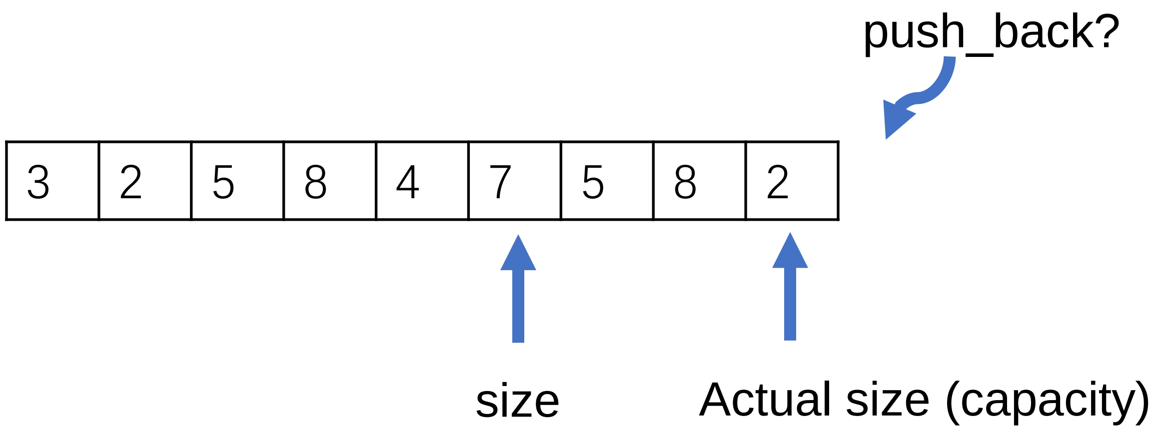
Similar data structures

- **Resizable arrays**
- **When we write code, usually we specify the array size when we allocate memory, but what happens if we do not know the upper bound of the array size?**
  - E.g., “std::vector” in C++
  - push_back: add to the end
  - size(): get the current size



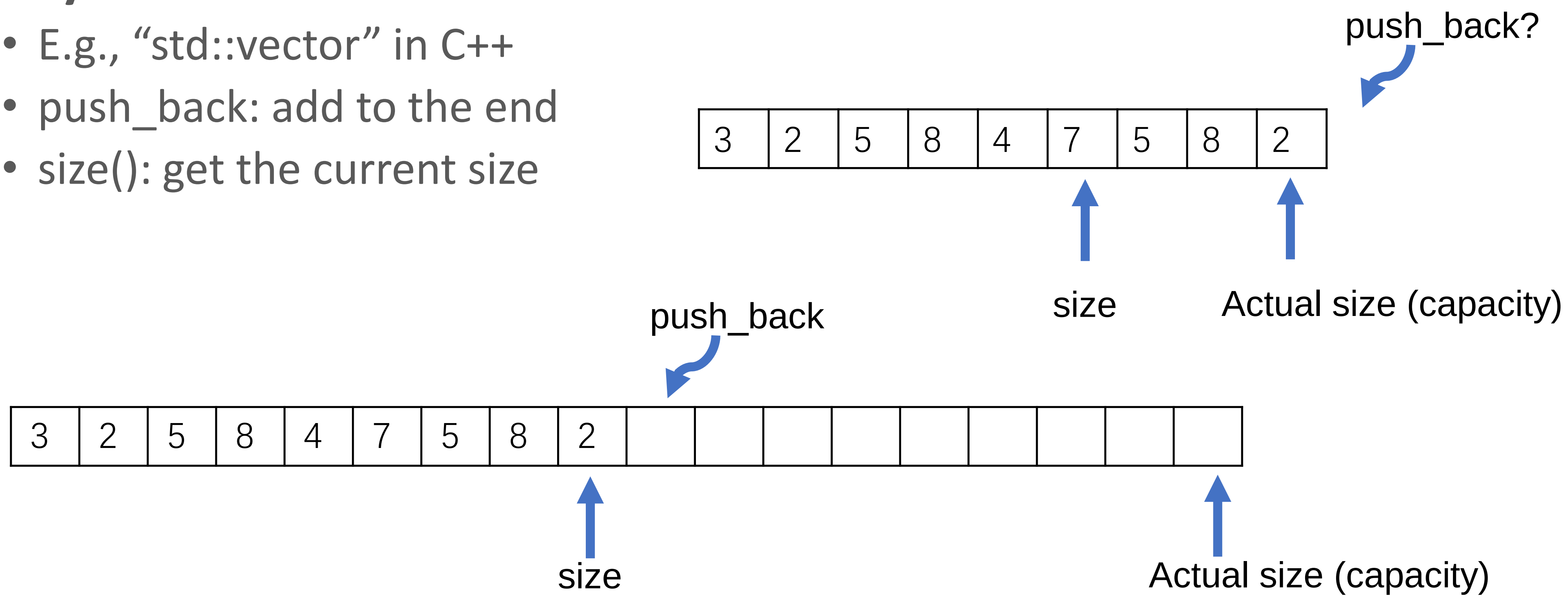
Similar data structures

- **Resizable arrays**
- **When we write code, usually we specify the array size when we allocate memory, but what happens if we do not know the upper bound of the array size?**
  - E.g., “std::vector” in C++
  - push_back: add to the end
  - size(): get the current size



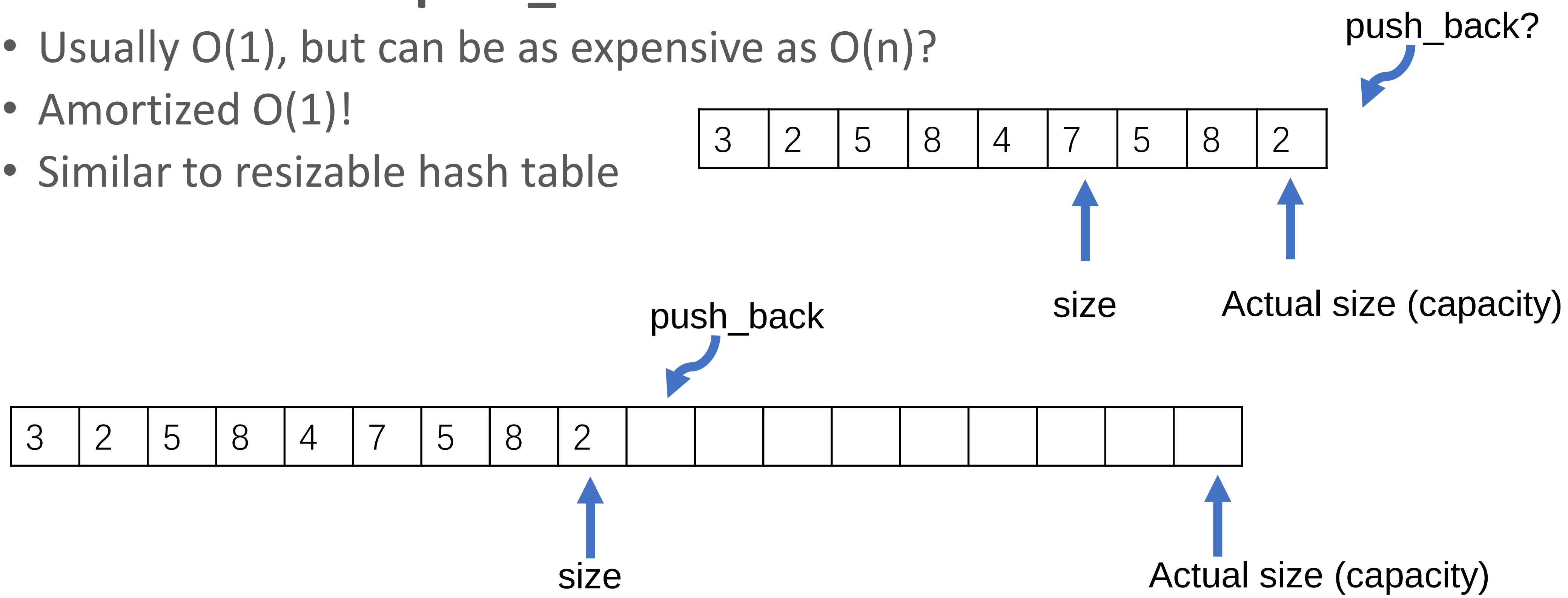
Similar data structures

- **Resizable arrays**
- **When we write code, usually we specify the array size when we allocate memory, but what happens if we do not know the upper bound of the array size?**
  - E.g., “std::vector” in C++
  - push_back: add to the end
  - size(): get the current size



Resizable arrays

- **Are we wasting space?**
  - Yes, but up to a constant factor!
  - When we store k elements, we use at most 2k space
- **What is the cost of push_back**
  - Usually O(1), but can be as expensive as O(n)?
  - Amortized O(1)!
  - Similar to resizable hash table



Summary for amortized analysis

- **Some operations are expensive, some of them are cheap**
- **The amortized analysis considers the cost of the “average” cost per operation on a sequence of operations**
  - Direct analysis
  - Piggy bank analysis
  - Potential analysis

- **Examples:**
  - Array-based data structures (e.g., vector, hash-tables)
  - Cost analysis for Fibonacci heap, Splay tree, weight-balanced tree, scapegoat tree, and many other data structures
  - Many other algorithms such as parallel scheduler

The next 2 weeks

- Graph algorithms