What is the time complexity of inserting an element into an unordered set? Group of answer choices O(n) in all cases O(1) on average, O(n) in worst case O(n^2) O(log n)

**ChatGPT said:**

The correct answer is:

**O(1) on average, O(n) in the worst case.**

**Explanation:**

* In an **unordered set**, elements are typically stored using a **hash table**.
* **On average**, inserting an element into a hash table takes **O(1)** time because hashing and inserting into a bucket is constant time.
* **In the worst case**, if there are many hash collisions, all elements could end up in the same bucket (linked list or another structure), leading to **O(n)** time complexity for insertion.

Thus, the best choice is:  
👉 **O(1) on average, O(n) in the worst case.**

4o

**You said:**

Suppose we have the following pseudo-code to count element frequencies in an array: arr = [4, 1, 2, 2, 4, 3] countMap = new Map() for e in arr: if e not in countMap: countMap[e] = 1 else: countMap[e] += 1 print(countMap) Question: What will be the output of countMap? Group of answer choices {1: 1, 2: 2, 3: 1, 4: 1} {4: 1, 1: 1, 2: 2, 3: 1} {4: 2, 1: 2, 2: 1, 3: 1} {4: 2, 1: 1, 2: 2, 3: 1}

**ChatGPT said:**

Let's analyze the pseudo-code and determine the output of countMap.

**Given array:**

ini

CopyEdit

arr = [4, 1, 2, 2, 4, 3]

**Execution of the loop:**

1. e = 4: Not in countMap, so countMap[4] = 1
2. e = 1: Not in countMap, so countMap[1] = 1
3. e = 2: Not in countMap, so countMap[2] = 1
4. e = 2: Already in countMap, increment countMap[2] → countMap[2] = 2
5. e = 4: Already in countMap, increment countMap[4] → countMap[4] = 2
6. e = 3: Not in countMap, so countMap[3] = 1

**Final countMap:**

yaml

CopyEdit

{4: 2, 1: 1, 2: 2, 3: 1}

**Correct Answer:**

👉 **{4: 2, 1: 1, 2: 2, 3: 1}**

4o

**You said:**

Using the hash function h(k) = k mod 7, insert the following keys into a hash table with separate chaining: Keys: [10, 21, 32, 43, 25, 36] What will be stored at index 3? Group of answer choices [10, 21, 32] [43] [10] [43, 25]

**ChatGPT said:**

Let's analyze the hash function and insert the keys into the hash table using **separate chaining**.

**Given hash function:**

h(k)=kmod  7h(k) = k \mod 7h(k)=kmod7

**Compute hash values:**

| **Key** | **Hash value kmod  7k \mod 7kmod7** | **Index** |
| --- | --- | --- |
| 10 | 10mod  7=310 \mod 7 = 310mod7=3 | 3 |
| 21 | 21mod  7=021 \mod 7 = 021mod7=0 | 0 |
| 32 | 32mod  7=432 \mod 7 = 432mod7=4 | 4 |
| 43 | 43mod  7=143 \mod 7 = 143mod7=1 | 1 |
| 25 | 25mod  7=425 \mod 7 = 425mod7=4 | 4 |
| 36 | 36mod  7=136 \mod 7 = 136mod7=1 | 1 |

**Hash table after insertion:**

* **Index 0** → [21]
* **Index 1** → [43, 36] (chaining)
* **Index 3** → [10]
* **Index 4** → [32, 25] (chaining)

**What is stored at index 3?**

👉 **[10]**

**Correct Answer:**

✔ **[10]**

4o

**You said:**

Using the hash function h(k) = k mod 7, insert the keys [14, 28, 41, 56, 21] into a hash table with 7 slots (indexed from 0 to 6) using Linear Probing. After inserting all the keys, which value will be located at index 2? Group of answer choices 28 null 56 41

**ChatGPT said:**

Let's go step by step to insert the keys using **Linear Probing** with the given hash function:

**Given hash function:**

h(k)=kmod  7h(k) = k \mod 7h(k)=kmod7

We use **Linear Probing**, meaning if a collision occurs, we check the next available slot sequentially.

**Insert keys into the hash table:**

**Step 1: Insert 14**

h(14)=14mod  7=0h(14) = 14 \mod 7 = 0h(14)=14mod7=0

* Insert **14** at index **0**.

**Step 2: Insert 28**

h(28)=28mod  7=0h(28) = 28 \mod 7 = 0h(28)=28mod7=0

* Collision at index **0** (occupied by **14**).
* Linear probing → Check index **1**, it's empty → Insert **28** at index **1**.

**Step 3: Insert 41**

h(41)=41mod  7=6h(41) = 41 \mod 7 = 6h(41)=41mod7=6

* Insert **41** at index **6**.

**Step 4: Insert 56**

h(56)=56mod  7=0h(56) = 56 \mod 7 = 0h(56)=56mod7=0

* Collision at index **0** (**14** is there).
* Linear probing → Check index **1** (occupied by **28**).
* Linear probing → Check index **2**, it's empty → Insert **56** at index **2**.

**Step 5: Insert 21**

h(21)=21mod  7=0h(21) = 21 \mod 7 = 0h(21)=21mod7=0

* Collision at index **0** (**14** is there).
* Linear probing → Check index **1** (**28** is there).
* Linear probing → Check index **2** (**56** is there).
* Linear probing → Check index **3**, it's empty → Insert **21** at index **3**.

**Final Hash Table:**

| **Index** | **Value** |
| --- | --- |
| 0 | 14 |
| 1 | 28 |
| 2 | 56 |
| 3 | 21 |
| 4 | null |
| 5 | null |
| 6 | 41 |

**Answer:**

Value at index **2** = **56**.

👉 **Correct answer: 56**.

4o

**You said:**

What will be the state of the map after the following operations? put(1, A) put(2, B) put(3, C) remove(2) get(2) Group of answer choices (1,A), (2,null), (3,C) (1,A), (3,C) (1,A), (2,B), (3,C) (1,A), (2,B)

**ChatGPT said:**

Let's analyze the sequence of operations on the map (typically implemented as a **hash map** or **dictionary**):

**Operations Breakdown:**

1. **put(1, A)** → Inserts key **1** with value **A** → Map: { 1: A }
2. **put(2, B)** → Inserts key **2** with value **B** → Map: { 1: A, 2: B }
3. **put(3, C)** → Inserts key **3** with value **C** → Map: { 1: A, 2: B, 3: C }
4. **remove(2)** → Removes key **2** from the map → Map: { 1: A, 3: C }
5. **get(2)** → Tries to retrieve key **2**, but it was removed, so it returns **null or does not exist**.

**Final State of the Map:**

{ 1: A, 3: C }

**Correct Answer:**

👉 **(1, A), (3, C)**

4o

**You said:**

Consider a map where keys are integers and values are characters. What will be the final state of the map after executing the following operations? put(10, A) put(15, B) put(20, C) put(10, D) remove(15) put(20, E) get(15) Group of answer choices (10,D), (15,B), (20,E) (10,A), (15,null), (20,E) (10,A), (15,B), (20,C) (10,D), (20,E)

**ChatGPT said:**

Let's analyze the sequence of operations step by step on a **map (dictionary-like structure)** where keys are **integers** and values are **characters**.

**Operations Breakdown:**

1. **put(10, A)** → Inserts key **10** with value **A**  
   **Map:** { 10: A }
2. **put(15, B)** → Inserts key **15** with value **B**  
   **Map:** { 10: A, 15: B }
3. **put(20, C)** → Inserts key **20** with value **C**  
   **Map:** { 10: A, 15: B, 20: C }
4. **put(10, D)** → Updates key **10** with value **D** (overwrites **A**)  
   **Map:** { 10: D, 15: B, 20: C }
5. **remove(15)** → Removes key **15** from the map  
   **Map:** { 10: D, 20: C }
6. **put(20, E)** → Updates key **20** with value **E** (overwrites **C**)  
   **Map:** { 10: D, 20: E }
7. **get(15)** → Tries to retrieve key **15**, but it was removed → returns **null or not found**

**Final State of the Map:**

{ 10: D, 20: E }

**Correct Answer:**

👉 **(10, D), (20, E)**

4o

**You said:**

What is the worst-case time complexity of the following remove(k) algorithm implemented with a singly linked list? p = head; while p is not null do if p.element.key = k then t = p→element.value; remove(p); return t; p = p→next; return null; Group of answer choices O(n log n) O(n) O(1) O(log n)

**ChatGPT said:**

**Analyzing the remove(k) Algorithm:**

The given algorithm searches for a node with key **k** in a **singly linked list** and removes it.

**Step-by-Step Complexity Analysis:**

1. **Worst-case scenario:** The key **k** is **not in the list**, or it is at the **end of the list**.
2. **Traversal:** The algorithm iterates through the list **one node at a time** using p = p→next, which results in a maximum of **n iterations** (where n is the number of elements in the list).
3. **Removal:** Once the node is found, the removal takes **O(1)** time.
4. **Best-case scenario:** If **k** is found at the head of the list, the algorithm terminates in **O(1)** time.

**Worst-Case Time Complexity:**

* In the worst case, the algorithm **must traverse the entire list**, which is **O(n)**.
* The removal operation itself is **O(1)**, but it only occurs after traversal.

**Final Answer:**

✅ **O(n)**

4o

**You said:**

Consider this pseudo-code that implements a Map of Sets, where each key maps to a Set of integer values: mapOfSets M M['group1'] = emptySet() M['group1'].insert(10) M['group1'].insert(15) M['group2'] = emptySet() M['group2'].insert(15) M['group2'].insert(20) print(M) What is the appropriate way to describe the structure printed by print(M)? Group of answer choices A map containing keys 'group1' and 'group2', each associated with distinct sets A single set containing all values from both groups A map with duplicated keys A list of tuples (group, value)

**ChatGPT said:**

**Analyzing the Pseudo-code:**

The given code implements a **Map of Sets**, meaning each key maps to a **set of integer values**.

**Step-by-step Execution:**

1. **Initialize M** → A map (dictionary-like structure).
2. **M['group1'] = emptySet()**
   * group1 is associated with an empty set.
3. **Insert elements into group1:**
   * M['group1'].insert(10) → group1 = {10}
   * M['group1'].insert(15) → group1 = {10, 15}
4. **M['group2'] = emptySet()**
   * group2 is associated with another empty set.
5. **Insert elements into group2:**
   * M['group2'].insert(15) → group2 = {15}
   * M['group2'].insert(20) → group2 = {15, 20}
6. **Final Structure of M:**

bash

CopyEdit

{

'group1': {10, 15},

'group2': {15, 20}

}

1. **print(M) Output:**  
   A map (dictionary) where:
   * 'group1' maps to {10, 15}
   * 'group2' maps to {15, 20}

**Answer Choices Analysis:**

✅ **Correct answer:**  
👉 **"A map containing keys 'group1' and 'group2', each associated with distinct sets"**

**Why Not the Other Choices?**

❌ **"A single set containing all values from both groups"**

* Incorrect because we have **two separate sets**, not one merged set.

❌ **"A map with duplicated keys"**

* Incorrect because maps **do not allow duplicate keys**. Each key ('group1', 'group2') is unique.

❌ **"A list of tuples (group, value)"**

* Incorrect because the structure is a **map of sets**, not a list of key-value pairs.

4o

**You said:**

Using the hash function h(k) = k mod 5, insert the keys [12, 17, 9, 10] into a hash table with 5 slots (indexed from 0 to 4) using Linear Probing. After inserting the keys in the given order, which value will be located at index 2 in the hash table? Group of answer choices 12 10 9 17

**ChatGPT said:**

Let's go step by step using **Linear Probing** to insert the given keys into a hash table with **5 slots** (indexed from 0 to 4) using the hash function:

h(k)=kmod  5h(k) = k \mod 5h(k)=kmod5

**Step 1: Insert 12**

h(12)=12mod  5=2h(12) = 12 \mod 5 = 2h(12)=12mod5=2

* Insert **12** at index **2**.

**Hash Table:**

makefile

CopyEdit

Index: 0 1 2 3 4

Values: - - 12 - -

**Step 2: Insert 17**

h(17)=17mod  5=2h(17) = 17 \mod 5 = 2h(17)=17mod5=2

* Collision at **index 2** (occupied by 12).
* **Linear Probing:** Check **index 3** → it's empty.
* Insert **17** at **index 3**.

**Hash Table:**

makefile

CopyEdit

Index: 0 1 2 3 4

Values: - - 12 17 -

**Step 3: Insert 9**

h(9)=9mod  5=4h(9) = 9 \mod 5 = 4h(9)=9mod5=4

* Insert **9** at **index 4**.

**Hash Table:**

makefile

CopyEdit

Index: 0 1 2 3 4

Values: - - 12 17 9

**Step 4: Insert 10**

h(10)=10mod  5=0h(10) = 10 \mod 5 = 0h(10)=10mod5=0

* Insert **10** at **index 0**.

**Final Hash Table:**

makefile

CopyEdit

Index: 0 1 2 3 4

Values: 10 - 12 17 9

**Answer:**

Value at **index 2** = **12**.

👉 **Correct answer: 12**.

4o

**You said:**

What is the worst-case complexity of retrieving an element in a list-based map? Group of answer choices O(log n) O(n log n) O(1) O(n)

**ChatGPT said:**

**Understanding List-Based Map Retrieval:**

A **list-based map** is typically implemented as an **unsorted linked list or array of key-value pairs**.

**Retrieval Process:**

* To find an element, we must **scan through the list** to locate the key.
* In the **worst case**, the key is either at the **end** of the list or **not present** at all, requiring us to traverse the **entire list**.

**Worst-Case Time Complexity:**

* **Best case:** If the key is at the beginning → **O(1)**.
* **Worst case:** If the key is at the end or missing → **O(n)**.

**Correct Answer:**

✅ **O(n)**

4o

**You said:**

giải thích chi tiết về độ phức tạp thuật toán của unordered set

**ChatGPT said:**

**Giải thích chi tiết về độ phức tạp thuật toán của unordered\_set**

Trong **C++**, unordered\_set là một cấu trúc dữ liệu dựa trên **bảng băm (hash table)**. Nó hỗ trợ các thao tác như **thêm (insert), xóa (erase), và tìm kiếm (find)** với **độ phức tạp trung bình O(1)**, nhưng trong trường hợp xấu nhất có thể là **O(n)**.

**1. Cấu trúc nội bộ của unordered\_set**

* unordered\_set sử dụng **bảng băm (hash table)** để lưu trữ dữ liệu.
* Các phần tử được ánh xạ vào **các bucket (nhóm)** dựa trên giá trị của **hàm băm (hash function)**.
* Nếu hai phần tử có cùng giá trị băm, chúng sẽ được đặt trong cùng một bucket (**xảy ra xung đột - collision**).
* Để giải quyết xung đột, unordered\_set thường sử dụng **danh sách liên kết (chaining)** hoặc **open addressing**.

**2. Phân tích độ phức tạp thuật toán của unordered\_set**

| **Thao tác** | **Trung bình (Average)** | **Trường hợp xấu nhất (Worst Case)** |
| --- | --- | --- |
| **Insert** | O(1) | O(n) (khi có quá nhiều xung đột) |
| **Erase** | O(1) | O(n) |
| **Find** | O(1) | O(n) |
| **Iterate** | O(n) | O(n) |

**2.1. Insert (Thêm phần tử)**

* Khi thêm một phần tử x, unordered\_set sử dụng **hàm băm** để tính toán vị trí bucket.
* Nếu bucket đó chưa có phần tử nào → **Chèn vào O(1)**.
* Nếu bucket đã chứa một phần tử khác (xung đột) → Thêm vào danh sách liên kết của bucket → **Vẫn O(1) trong trung bình**, nhưng **O(n) trong trường hợp xấu nhất**.

**2.2. Find (Tìm kiếm phần tử)**

* Tương tự **insert()**, unordered\_set tính toán vị trí của phần tử trong bảng băm.
* Nếu phần tử nằm đúng vị trí của nó trong bucket → **O(1)**.
* Nếu có xung đột và phần tử nằm sâu trong danh sách liên kết → **O(n) trong trường hợp xấu nhất**.

**2.3. Erase (Xóa phần tử)**

* unordered\_set tìm phần tử cần xóa và loại bỏ nó khỏi bucket.
* Nếu không có xung đột → **O(1)**.
* Nếu có xung đột → Phải tìm phần tử trong danh sách liên kết → **O(n) trong trường hợp xấu nhất**.

**2.4. Iterate (Duyệt qua tất cả phần tử)**

* unordered\_set không đảm bảo thứ tự của các phần tử.
* Duyệt tất cả phần tử luôn là **O(n)** vì phải kiểm tra toàn bộ bảng băm.

**3. Trường hợp xấu nhất (Worst Case) xảy ra khi nào?**

* Khi **tất cả phần tử bị ánh xạ vào cùng một bucket** (tức là mọi giá trị băm đều giống nhau).
* Khi đó, unordered\_set hoạt động như một **danh sách liên kết**, và tất cả các thao tác find(), insert(), erase() sẽ có độ phức tạp **O(n)** thay vì **O(1)**.
* Điều này thường xảy ra khi:
  + **Hàm băm kém chất lượng**, tạo quá nhiều xung đột.
  + **Tập dữ liệu lớn**, dẫn đến nhiều phần tử trong cùng một bucket.
  + **Không mở rộng kích thước bảng băm kịp thời** (rehashing).

**4. Cách giảm xung đột và tối ưu hiệu suất**

**4.1. Dùng hàm băm tốt**

* Dùng các hàm băm chuẩn như std::hash<> trong C++ thay vì tự viết hàm băm yếu.

**4.2. Điều chỉnh kích thước bảng băm (rehashing)**

* unordered\_set sẽ tự động mở rộng bảng băm khi số phần tử vượt ngưỡng **load factor** (thường là 0.75).
* Khi rehash xảy ra, tất cả phần tử sẽ được phân bố lại → tránh tình trạng quá tải trong một bucket.

**4.3. Tránh sử dụng khóa có giá trị băm giống nhau**

* Khi sử dụng các kiểu dữ liệu phức tạp làm key (ví dụ std::pair, std::string), đảm bảo rằng chúng có giá trị băm phân bố đều.

**5. So sánh unordered\_set với set**

| **Đặc điểm** | **unordered\_set (Hash Table)** | **set (Red-Black Tree)** |
| --- | --- | --- |
| Cấu trúc nội bộ | Bảng băm | Cây đỏ-đen (Balanced BST) |
| Độ phức tạp trung bình | **O(1)** | **O(log n)** |
| Độ phức tạp xấu nhất | **O(n)** | **O(log n)** |
| Duyệt phần tử | Không theo thứ tự | Theo thứ tự tăng dần |
| Thích hợp cho | **Tìm kiếm nhanh** | **Duyệt có thứ tự** |

🔹 **Dùng unordered\_set khi cần hiệu suất truy vấn nhanh (O(1)) và không quan tâm thứ tự phần tử.**  
🔹 **Dùng set khi cần duyệt phần tử theo thứ tự (O(log n)).**

**6. Kết luận**

* **Trong trường hợp trung bình**, unordered\_set có độ phức tạp **O(1)** cho **insert, erase, find**.
* **Trong trường hợp xấu nhất**, khi có quá nhiều xung đột, độ phức tạp trở thành **O(n)**.
* **Cần chọn hàm băm tốt và đảm bảo rehash đúng cách để tránh xung đột**.

👉 Nếu mục tiêu là **truy vấn nhanh**, unordered\_set là một lựa chọn tuyệt vời! 🚀

4o

Top of Form

Bottom of Form