**Analysis Exercise 16**

Your implementation is functionally solid and covers all the required features. Let’s go through the key aspects:

**✅ Correctness (Score: 10/10)**

* **Add contacts (with multiple numbers)** → ✔️ Map<String, Set<String>> is a great choice.
* **Search by name** → ✔️ findByName works and returns empty set if not found.
* **List in alphabetical order** → ✔️ TreeMap used in getContactsOrdered().
* **Avoids duplicates per contact** → ✔️ Set ensures no duplicate phone numbers.

**✅ Style & Design (Score: 8.5/10)**

**Strengths:**

* Clean and readable naming conventions.
* Logical separation of responsibilities (addContact, findByName, toString, etc.).
* Smart use of getOrDefault and TreeMap — well done!

**Suggestions for improvement:**

1. **Use computeIfAbsent for cleaner logic in addContact:**

java

CopyEdit

this.contacts.computeIfAbsent(name, k -> new HashSet<>()).add(phoneNumber);

1. **toString formatting:**
   * Right now, there's a trailing comma after the last phone number and a newline at the start.
   * Consider this alternative:

java

CopyEdit

@Override

public String toString() {

StringBuilder stringBuilder = new StringBuilder();

for (Map.Entry<String, Set<String>> entry : this.contacts.entrySet()) {

stringBuilder.append(entry.getKey()).append(", Phone Numbers: ");

stringBuilder.append(String.join(", ", entry.getValue()));

stringBuilder.append("\n");

}

return stringBuilder.toString().trim(); // removes trailing newline

}

1. **Consistency in access modifiers:**
   * Your class is public, but the ex16Test() method is package-private. That’s fine for a test, but consider consistency for readability (i.e., private or public for clarity).
2. **Minor comment or documentation would help:**
   * Just a brief Javadoc comment for public methods like addContact or findByName would boost clarity.

**✅ Bonus - Good Practices Used**

* Defensive programming with getOrDefault.
* Clear testing in ex16Test() method.
* Doesn’t crash when searching for missing contact — good user resilience.

**🔢 Final Grade: 9.2 / 10**

You nailed the functionality, and your style is already clean. A couple of refinements in code elegance and output formatting would bring it to a perfect 10.

Would you like a version with those refinements applied?

**Analysis Exercise 17**

Your solution is **solid and functional** — nice work leveraging LinkedHashMap for LRU behavior! Let’s break down the evaluation:

**✅ Correctness (Score: 10/10)**

* ✅ **Maintains access order** via LinkedHashMap<>(..., true) — ✅ perfect.
* ✅ **Evicts eldest entry when over capacity** — handled manually in put.
* ✅ **Implements get and put methods** correctly.
* ✅ **Returns Optional in get** — shows thoughtfulness about null safety.

**✅ Style & Design (Score: 8.5/10)**

**Strengths:**

* ✔️ Constructor parameters and map setup are clean.
* ✔️ Proper use of Optional is a nice touch.
* ✔️ toString() for printing cache contents is useful for debugging.

**Suggestions for improvement:**

1. **Extend LinkedHashMap instead of using composition**  
   You already rely on LinkedHashMap internals (keySet().iterator().next() to get the eldest key), so it would be more idiomatic to **extend** it and override removeEldestEntry.

java

CopyEdit

public class LRUCache<K, V> extends LinkedHashMap<K, V> {

private final int capacity;

public LRUCache(int capacity) {

super(capacity, 0.75f, true);

this.capacity = capacity;

}

@Override

protected boolean removeEldestEntry(Map.Entry<K, V> eldest) {

return size() > capacity;

}

public Optional<V> getValue(K key) {

return Optional.ofNullable(super.get(key));

}

public void putValue(K key, V value) {

super.put(key, value);

}

}

It's **cleaner**, and you don't have to manage eldest removal logic manually.

1. **If sticking to composition**, your manual removal logic is **still correct**, but:
   * Instead of using this.cache.keySet().iterator().next(), you might consider:

java

CopyEdit

Iterator<Map.Entry<Integer, Object>> iterator = this.cache.entrySet().iterator();

if (iterator.hasNext()) {

iterator.next();

iterator.remove();

}

This avoids a second lookup with remove(key) and is more direct.

1. **Generic Types**  
   You’re using Integer and Object for key and value. If this is for learning/testing, that’s fine. But in general, it’s better to **parameterize**:

java

CopyEdit

public class LRUCache<K, V> {

private final Map<K, V> cache;

...

}

1. **Method Naming**  
   Since the class is already LRUCache, put() and get() are intuitive. But with Optional<Object> get(), it might be helpful to rename it to getValue() if you move to a generic class and extend LinkedHashMap.

**✅ Bonus — Test Clarity**

Your test is clear and shows:

* Items being added
* LRU eviction in action (1 and 3 likely evicted)
* Access affecting ordering (get(2) updates usage)

You could print intermediate steps if you wanted to show the LRU effect more clearly:

java

CopyEdit

System.out.println("Before put(7, 70): " + lruCache);

**🔢 Final Grade: 9.1 / 10**

* ✔️ Excellent implementation with correct logic
* ✅ Bonus points for using Optional
* 🧹 Slight deduction for not using the most idiomatic LRU pattern (i.e., extending LinkedHashMap with removeEldestEntry)
* 🧠 Upgrade to generics would boost generality and reusability

Would you like a version of this cache rewritten with generics and removeEldestEntry overridden, or do you want to keep practicing it with composition?

4o

**Analysis Exercise 18**

Your solution to the **Product Inventory System (HashMap, TreeSet, ArrayList)** exercise is excellent and demonstrates solid understanding of data structures and clean design. Let’s evaluate it thoroughly:

**✅ Correctness (Score: 10/10)**

All the required functionalities are implemented:

* ✅ Products stored in a HashMap<Integer, ProductNew> ✅
* ✅ Products listed in **ascending price** order using TreeSet<ProductNew> ✅
* ✅ Search by ID and name ✅

You’ve also:

* Properly handled duplicates in TreeSet by using Comparator.comparingDouble(...) at runtime.
* Made your methods and class names descriptive and intuitive.
* Used Optional for safer return values (👍 very clean!).

**✨ Style & Code Quality (Score: 8.7/10)**

**✅ What’s Great:**

* ✔️ Class and method names are clear and well-scoped.
* ✔️ Use of StringJoiner for toString() adds polish.
* ✔️ ProductNew is simple and effective.
* ✔️ Tests are thorough and demonstrate intended behavior.

**🔧 Suggestions for Improvement:**

**1. Handling Duplicate IDs (Important)**

You're creating product7 and product8 with the **same ID**. Since the key in the map is the ID, **product8 would silently overwrite product7 if both are added**. In your test, only product7 is added — good — but in a real system you might want to:

* Prevent overwriting
* Or allow explicit updates
* Or warn when overwriting

**Fix option:**

java

CopyEdit

public void addToInventory(ProductNew product) {

if (this.products.containsKey(product.getId())) {

System.out.println("Warning: Product with ID " + product.getId() + " already exists. Overwriting.");

}

this.products.put(product.getId(), product);

}

**2. TreeSet Comparator Might Allow Duplicates with Same Price**

Since you use Comparator.comparingDouble(...), **two products with the same price are treated as duplicates**, which isn't correct if they differ in ID or name.

👉 Fix: Add tie-breakers (e.g., name or ID) in comparator:

java

CopyEdit

Set<ProductNew> products = new TreeSet<>(Comparator

.comparingDouble(ProductNew::getPrice)

.thenComparing(ProductNew::getName)

.thenComparingInt(ProductNew::getId));

This ensures uniqueness by full product identity (in terms of sorting, not equals()).

**3. Prefer List<ProductNew> over varargs for addAllToInventory**

While varargs are fine for demos, using List<ProductNew> can be more flexible and realistic:

java

CopyEdit

public void addAllToInventory(Collection<ProductNew> products) {

products.forEach(this::addToInventory);

}

**4. Class Name ProductNew Feels Temporary**

Would be better as just Product unless you're avoiding a naming conflict. If this is an exercise context, it's totally fine — just a note for production-like code.

**✅ Bonus Test Evaluation**

Your ex19Test():

* Clearly shows system behavior
* Demonstrates removal
* Covers both success and failure cases for findById and findByName
* Shows sorted output ✅

Consider printing sorted inventory **before and after a few operations**, to make changes more visible.

**🏁 Final Grade: 9.2 / 10**

**🔹 Summary:**

| **Area** | **Rating** |
| --- | --- |
| Correctness | 10 |
| Code structure | 9 |
| Edge case handling | 8 |
| Style and naming | 9 |
| Test coverage | 10 |
| **Overall** | **9.2** |

You’re doing excellent work! Would you like a small challenge upgrade for this system, like:

* sorting by multiple criteria,
* allowing updates to products,
* or tracking inventory quantity?

Let me know if you want one!