**🔎 Efficiency of Search Operations in Hash Tables**

**Python’s dict (Open Addressing) vs. Separate Chaining (Closed Addressing)**

**1. Python’s Implementation (Open Addressing)**

* **Collision Resolution**: Variant of **double hashing**
* **Search Efficiency**:
  + **Average Case**: O(1)
    - Direct access on no collision
    - Probing sequence if collision occurs
  + **Worst Case**: O(n)
    - Many collisions → long probe sequence
* **Key Factors**:
  + Load factor ≤ **2/3** (auto-resize when exceeded)
  + Pseudo-random probing sequence
  + Uses **tombstones** for deletions

**2. Closed Addressing (Separate Chaining)**

* **Collision Resolution**: Buckets hold linked lists (or sometimes balanced trees)
* **Search Efficiency**:
  + **Average Case**: O(1 + α)
    - α = n/m (load factor)
    - Hash computation + traversal of list
  + **Worst Case**: O(n)
    - All items land in the same bucket → linear search
* **Key Factors**:
  + Load factor can be > 1 (no strict resizing needed)
  + Simple deletion (remove from list directly)

**3. Efficiency Comparison**

|  |  |  |
| --- | --- | --- |
| **Aspect** | **Python (Open Addressing)** | **Closed Addressing (Separate Chaining)** |
| **Cache Performance** | ✅ Excellent (contiguous arrays) | ❌ Poor (pointer chasing, cache misses) |
| **Memory Overhead** | ✅ Lower (no pointer storage) | ❌ Higher (linked list nodes + pointers) |
| **Search - Best Case** | ✅ O(1) | ✅ O(1) |
| **Search - Worst Case** | ❌ O(n) | ❌ O(n) |
| **Load Factor Handling** | ✅ Auto-resize at ~66% | ✅ Can support higher α |
| **Deletion Handling** | ❌ Complex (tombstones) | ✅ Simple (direct removal) |

**4. Python-Specific Optimizations**

Python’s dict is **not just a naïve open addressing table** — it has several engineering optimizations:

1. **Combined table + entries** structure (efficient memory layout).
2. **Compact representation** for small dicts.
3. **Custom hash functions** for common types (str, int, etc.).
4. **Cache locality–friendly layout** (contiguous memory).

**Example (fast search):**

d = {'key1': 'value1', 'key2': 'value2'}  
value = d['key1'] # Extremely efficient lookup

**5. Performance Considerations**

**✅ Open Addressing (Python’s dict) is best when:**

* Small/medium collections
* Good hash function quality
* Memory-constrained environments
* Applications sensitive to cache performance

**✅ Closed Addressing (Separate Chaining) is better when:**

* Very high load factors
* Lots of frequent deletions
* Poorly distributed hash outputs
* Extremely large datasets (predictable performance tradeoffs)

**6. Real-World Performance**

* Python’s **open addressing** (dict) usually **outperforms separate chaining** for everyday workloads because:
  + Better cache performance
  + Lower memory overhead
  + Automatic resizing keeps load factor low
* However, **separate chaining** can be more **stable/predictable** when handling adversarial cases or pathological hash distributions.

**🎯 Conclusion**

Python’s choice of **open addressing with double hashing** provides:

* **Faster average-case lookups**
* **Cache-friendly design**
* **Lower memory cost**

At the cost of:

* More complex deletion logic
* Occasional need for resizing

For **general-purpose usage (like Python dicts)**, open addressing wins in practice.  
For **specialized environments with high deletion rates or poor hash functions**, separate chaining may be more robust.

👉 Do you want me to also include **a diagram/visual illustration** (showing probe sequences vs. chains) so it’s easier to compare at a glance?