Code lines	What it achieves	Notes (only if necessary)

`out.append("true" if ok else "false")`
`out.append(val if val is not None else "null")`
`out.append(",".join(parts) if parts else "")`
	Emits judge-friendly string outputs	Keep output types as strings unless prompt says otherwise

`ts = int(q[1]); size = int(q[3]); ttl = int(q[5])`	Normalizes numeric inputs that may arrive as strings	int() handles '7' and 7

`db = {}`
`db.setdefault(key, {})`
`db[key][field] = payload`
	Builds the standard nested map store	setdefault avoids repetitive if key not in db

`from collections import defaultdict`
`user_files = defaultdict(set)`
`user_tasks = defaultdict(list)`
	Maintains inverted indexes without guard code	Convert to plain dict only if a grader is picky (rare)

`fields = db.get(key)` `payload = fields.get(field) if fields else None`	Safe nested lookup without KeyError	Prefer this to try/except for speed/clarity
`del db[key][field]` `if not db[key]: del db[key]`	Deletes a field and removes empty parent container	Prevents “zombie keys” in later scans
`def is_live(exp_at, t):` `  return exp_at is None or t < exp_at`	Centralizes TTL liveness condition	If prompt uses <= boundary, change it here once

`p = fields.get(field)`
`if p and not is_live(p['exp'], ts):`
`  del fields[field]`
`  if not fields: del db[key]`
`  p = None`
	Lazy expiration on access	Apply same idea in SCAN / SEARCH too

`fields[field] = {'val': v, 'exp': None}` `fields[field] = {'val': v, 'exp': ts + ttl}`	Overwrite logic for non-expiring vs TTL writes	Keep a consistent payload shape everywhere
`matches = [f for f in fields if f.startswith(prefix)]`	Filters by field/name prefix	Usually fast enough under OA constraints
`matches = [n for n in files if n.endswith(suffix)]`	Filters by suffix	Combine with prefix via and

`res = []`
`for f, p in fields.items():`
`  if is_live(p['exp'], ts) and p['val'].startswith(vpref):`
`  res.append(f)`
	Filters by value-prefix while respecting TTL	Use items() to avoid extra dict lookups

`res.sort()` `ans = ','.join(f"{f}({fields[f]['val']})" for f in res)`	Lexicographic sorting + canonical field(value) formatting	Return '' if res empty
`items.sort(key=lambda x: (-x.size, x.name))`	Multi-key sort: size desc, name asc	Common for SEARCH-style outputs

`cands = [(size, name) for name, size in mp.items()]`
`cands.sort()`
`victim = cands.pop()`
	Eviction victim: largest size, tie lexicographically largest	Sorting asc then popping is a clean trick

`need = (usage + add) - cap`
`to_del = plan(need)`
`if sum(sz(f) for f in to_del) < need: return 'false'`
`for f in to_del: delete(f)`
`add_file()`
	Two-phase plan-then-apply for atomic operations	Prevents partial deletions on failure

`changed = []`
`changed.append((name, old_size))`
`for name, old in reversed(changed): restore(name, old)`
	Lightweight rollback using diffs	Great for “decompress may fail” patterns

`import copy`
`snaps[sid] = copy.deepcopy(state)`
`state = copy.deepcopy(snaps[sid])`
	Snapshot/restore without reference aliasing	Shallow copies are a classic bug

`sid = f"snap{len(snaps)+1}"`	Monotonic snapshot id generation	Deterministic and fast
`new_size = (size + 1) // 2`	Computes ceil(size/2)	Works for integers only

`rest = original`
`if owner and usage - cur + rest > cap: return 'false'`
`usage += (rest - cur)`
`cur = rest`
	Transactional size increase under quota	Don’t mutate state until after the check

`usage[user] += size; user_files[user].add(name)` `usage[user] -= size; user_files[user].discard(name)`	Keeps usage and ownership index consistent	Always update both structures together

`dst = File(name=new, size=src.size, owner=src.owner,`
`  compressed=src.compressed,`
`original_size=src.original_size)`
	Deep-copy metadata for COPY	Avoid shared references to mutable objects

`start = page * page_size` `page_items = items[start:start+page_size]`	Pagination via slicing	Confirm 0- vs 1-indexing in prompt