Chapter 05 — Searching Records: Two Complete TA Guides

XANDRIAN RESEARCH — RECONSTRUCTED SCHOLARLY EDITION
Edition: 26 October 2026. New scholarly reconstruction, prepared for a curated public reading collection. The original historic object could not be authenticated.

Reconstructed by Sol on 2026-10-26 from: the surviving `CH05_TA_GUIDE.md` and `CH05_TA_FIELD_GUIDE.md`, plus the published *PB-2000C searching* paper.

Document identity: This is a new scholarly reconstruction with a new identity. It is not the inaccessible historical Google Drive object, and it must not be presented as a byte-identical recovery. The two archived guides follow in full, separated by document boundary. Their historical "private teaching support" headers are retained verbatim even though Alex has authorized creating a public scholarly edition.

Part I — Historical TA Guide (complete)

CS617 Chapter 05

TA Guide - Searching Records

Private teaching support. Use this with the classroom PDF. The student-facing lab states the observable behavior; this guide preserves the reasoning, edge cases, and A-level analysis needed for consistent grading and tutoring.

---

1. Teaching intent

The conceptual progression is:

```
keys -> records -> ordered records -> indexed records
```

Random generation, status output, and hex display are supporting machinery. The core lesson is that the **comparison object** and the **movement object** are not necessarily the same thing.

For the core representation:

```
record = key + payload
record size = k + d
```

where `k >= 2`. The logical key is a random `uint16`; extra key bytes are zero padding under the established course byte-order convention.

---

2. Core storage and generation

The core working-data budget is 16 KiB. A generation request is safe only when its worst case fits:

```
r * (k + d) <= 16384
```

`r` is the number of generation attempts, not a guarantee of `r` accepted records. Each attempt generates a key, searches for a duplicate, rejects duplicates without retrying, and appends the key/payload only when unique.

After generation:

```
accepted + duplicate_rejections = r
```

A new generation replaces the collection and marks it unordered.

---

3. Search instrumentation

Count **logical candidate keys examined**, not HD61700 compare instructions.

Reference bounds for `n > 0`:

```
linear search: <= n candidates
unsuccessful linear search: exactly n candidates
standard binary search: <= floor(log2(n)) + 1 candidates
```

The important teaching point is observational: the two searches return the same logical result but can examine very different numbers of candidates.

Binary search requires an ordered collection. The implementation may sort once on the first binary-search request and remember that the unchanged collection is now ordered. A subsequent binary search should not sort again. Linear search is valid before or after sorting.

---

4. Sorting records

Students already know Shell sort from the preceding chapter. The new invariant is:

> Compare by key, but keep the entire logical record intact when its position changes.

Common failure:

```
keys sorted correctly
payload bytes left in original positions
```

That is not a correct record sort. Tests should verify association, not merely key order.

---

5. Display boundary

The physical LCD is 4 x 32. The record dump gets two display lines and should remain bounded to roughly 60 printable characters total across those lines. Grade readability and bounded output, not one exact syntax.

---

6. A-level indexed representation

The A ROM implements the same visible application behavior with a different representation:

```
stationary payload area
sorted index entries = key + 2-byte reference
```

Storage per logical item:

```
core     = k + d
indexed  = d + (k + 2) = k + d + 2
```

Therefore the A design spends two extra bytes per logical item. Under the same 16 KiB combined payload/index budget:

```
r * (k + d + 2) <= 16384
```

The object moved during sorting is:

```
core:    k + d bytes
indexed: k + 2 bytes
```

so indexed movement is smaller exactly when:

```
d > 2
```

At `d=2` movement size is equal; at `d=1` the index entry is larger than the original record. The intended A-level conclusion is:

> Indexing is a tradeoff, not an automatic improvement.

A correct A ROM should demonstrate that payload bytes remain stationary, index order changes, every reference still resolves to the correct payload, and linear/binary search returns the same logical record as the core representation.

---

7. Edge cases

Core tests should include:

```
empty collection
one record
first and last key
missing below minimum
missing above maximum
missing between stored keys
duplicate generation attempt
```

```
repeated binary search without re-sort
linear search before and after sorting
key/payload association after record movement
```

A-level tests additionally include:

```
payload stationary while index moves
references remain valid after sort
duplicate detection through index
linear and binary search through index keys
combined payload/index memory within 16 KiB
same logical result as the core ROM
```

---

8. B / A / Apple Polisher

```
B-level         correct record generation/search/sort application
A-level         same behavior through stationary payload + key/reference
index
Apple Polisher  exceptional modularity, reuse, and assembly craftsmanship
```

The Apple Polisher is independent of the B/A distinction. Reward natural boundaries, small understandable interfaces, useful reuse between the two ROMs, and little unnecessary coupling. Do not reward file count.

A useful engineering question is:

> If one search routine were replaced tomorrow, how much unrelated source would have to change?

---

9. Socratic hints

Records: “You already know how to sort keys. What must stay attached when a key moves?”

Binary search: “What property of the collection lets you discard half of the remaining candidates?”

Duplicates: “What is the simplest search you already know that can tell whether a generated key is present?”

Indexing: “If the payload were much larger than the key, why should sorting the key require moving all of the payload bytes?”

Tradeoff: “The index saves movement. What new bytes must be stored for every logical record?”

Part II — Historical TA Field Guide (complete)

CS617 Chapter 05

TA Field Guide - Searching Records

Private teaching support. This is the office-hours companion to `CH05_TA_GUIDE.md`.

---

1. First questions

When a student brings a failing search application, establish:

```
How many logical records exist?
Is the collection currently ordered?
What exactly is a record in this implementation?
What does the comparison counter count?
What was the last operation: gen, linear, binary, or sort?
```

Separate state errors from search-algorithm errors before reading low-level code.

---

2. Symptom triage

Binary search misses a key that linear search finds

Ask whether the collection was actually sorted under the same key ordering used by binary search. Then ask whether the binary interval update can discard the target incorrectly at either endpoint.

Binary search works once but behaves strangely after new generation

Inspect the `ordered` state. A new collection must invalidate any previous sorted-state claim.

Repeated binary search sorts every time

The behavior may be correct but misses an explicit efficiency/state requirement. Ask:

> What fact about an unchanged collection could you remember after the first sort?

Keys are sorted but payloads are wrong

This is the canonical record-movement failure. Ask:

What is the object being compared, and what is the object that must move?

Do not prescribe a byte-copy loop.

Duplicate count seems impossible

Check the experiment definition: `r` is attempts, not accepted count, and duplicates are rejected without replacement retries. Verify:

```
accepted + rejected_duplicates = r
```

Search comparison counts look too large

Ask whether the student is counting machine compare instructions instead of logical candidate keys examined.

Search comparison counts are too small

Look for hidden reuse of previous search state or a collection that was accidentally left sorted/generated differently from what the report claims.

A-level index returns the right key but wrong payload

Suspect reference maintenance rather than search order. Ask the student to choose one payload before sorting, record its address/reference, sort the index, and prove that the same reference still resolves to the same bytes.

A-level student claims the index is always better

Ask for both storage and movement equations. The key question is not “does it use an index?” but “what does the index buy, and what does it cost?”

Hex dump overruns the display

Keep this a UI-boundary issue. The assignment requires bounded readable output, not a full monitor. Do not let formatting consume the lab.

---

3. Hint escalation

Level 1 - state the invariant

```
"What must remain attached to this key?"
"What must be true before binary search is legal?"
```

Level 2 - identify persistent state

```
"What changes after a successful sort?"
"What event makes that fact false again?"
```

Level 3 - quantify the tradeoff

```
"How many bytes move in the core design? In the index design?"
```

Level 4 - reveal the TA inequality

Use `d > 2` only after the student has set up the movement comparison independently.

---

4. Red flags

Intervene when you see:

```
binary search performed on an unordered collection
key and payload separated by sorting
counter defined as processor compare instructions instead of candidate keys
new generation that leaves stale ordered state
A-level payload moved despite claiming stationary data
references regenerated or rewritten during index sort without justification
"index is always faster/better" stated without a cost model
```

---

5. Quick oral check

Ask:

```
Why can binary search discard half the interval?
What exactly does one search comparison mean in this lab?
After sorting records, what must still be true about every payload?
What event invalidates the ordered flag?
What extra storage does the A-level index cost per logical record?
When does the indexed movement become smaller than full-record movement?
```