the real complexities
are hiding

CSE 250
Lecture 35

The Memory Hierarchy

The University at Buffalo, SUNY

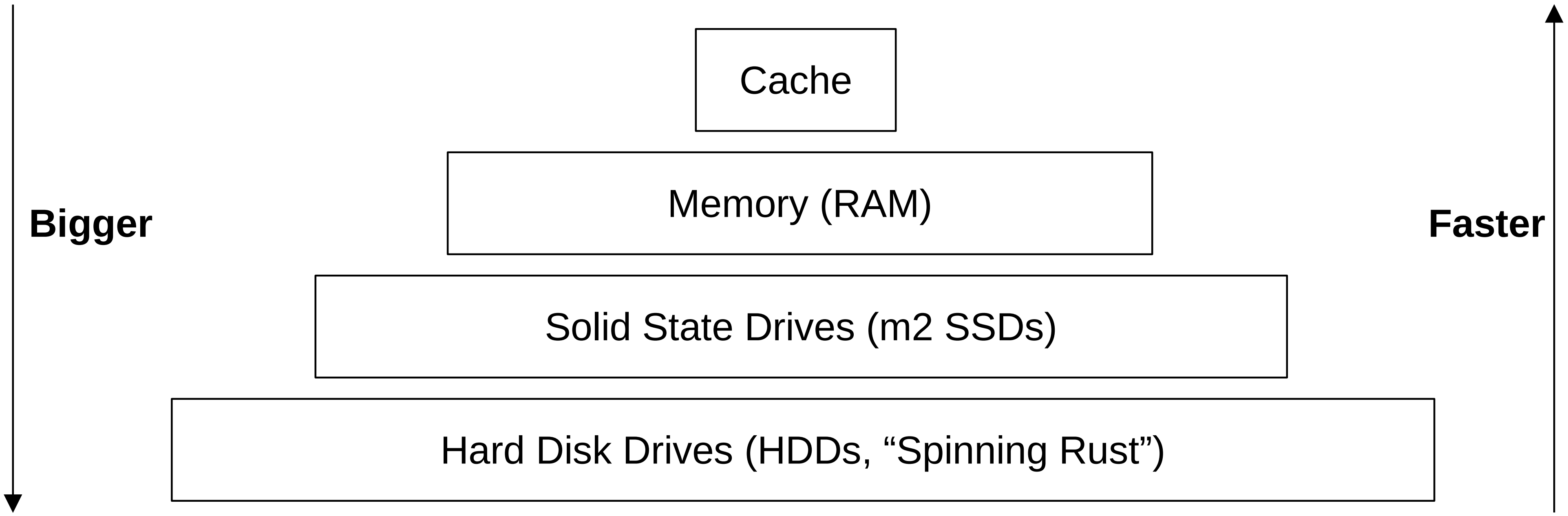
Lies!

- **Lie 1**: Accessing any element of an array of any length is O(1)
  - The “RAM” model of computation
    - Simplified model... but not perfect
  - Real-world Hardware isn’t this simple:
    - The Memory Hierarchy
    - Non-Uniform Memory Access (NUMA)
- **Lie 2**: The constants don’t matter

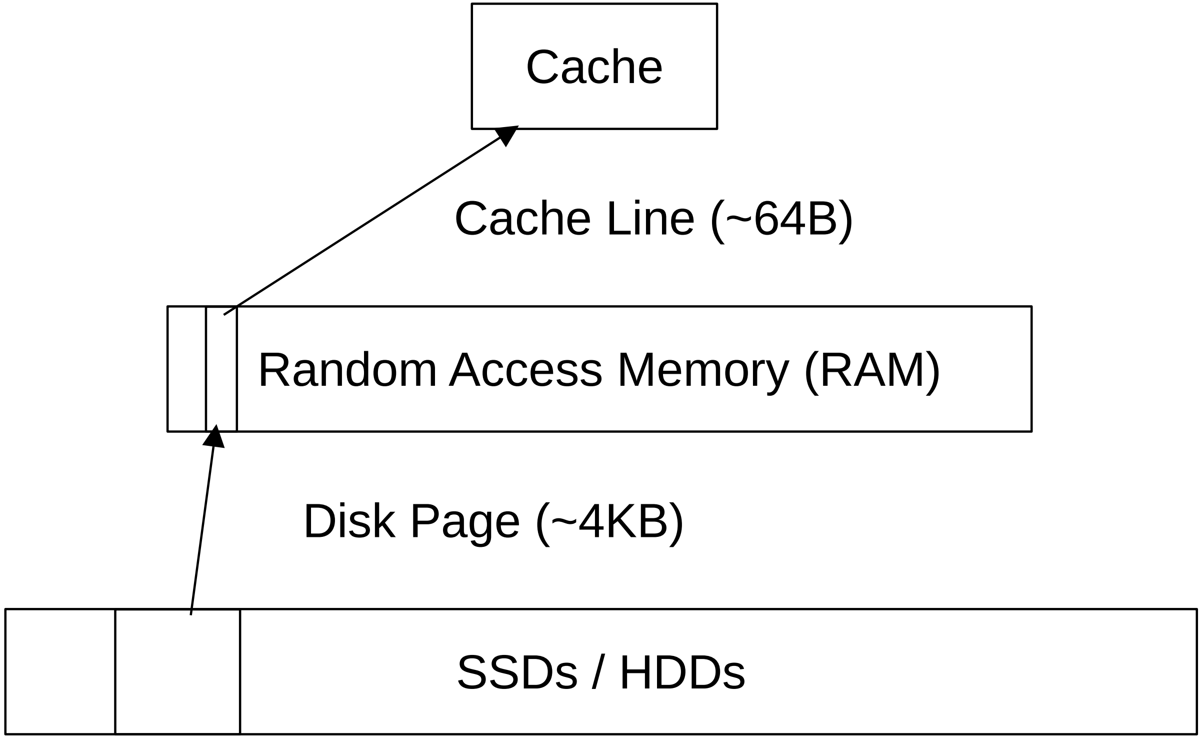
Algorithm Bounds

- Runtime Bounds
  - The algorithm takes O(…) time.
- Memory Bounds
  - The algorithm needs O(…) storage
- IO Bounds
  - The algorithm performs O(…) accesses to slower memory

The Memory Hierarchy (simplified)



The Memory Hierarchy (simplified)



Reading an Array Entry

Tiny constant

- Is the array entry in cache?
  - Yes
    - Return it (1-4 clock cycles)
  - No
    - Is the array entry in real memory

So-so constant

HUGE constant

      - Yes
        - Load it into cache (10s of clock cycles)
      - No
        - Load it out of virtual memory (100s of clock cycles)

Reading an Array Entry

It matters whether we're reading from cache, memory, or disk!

Today: Memory vs Disk

Ground Rules: Disk vs RAM

- All data starts off in a file on disk
  - Need to load data into RAM before accessing it.
  - Load data in 4KB chunks (“pages”).
  - The amount of available RAM is finite.
  - Deallocating a page is one instruction.
    - ... unless it was modified and needs to be written back.
- 3 features describe an algorithm:
  - Number of instructions (runtime complexity)
  - Number of data loads (IO complexity)
  - Number of pages of RAM required (memory complexity)

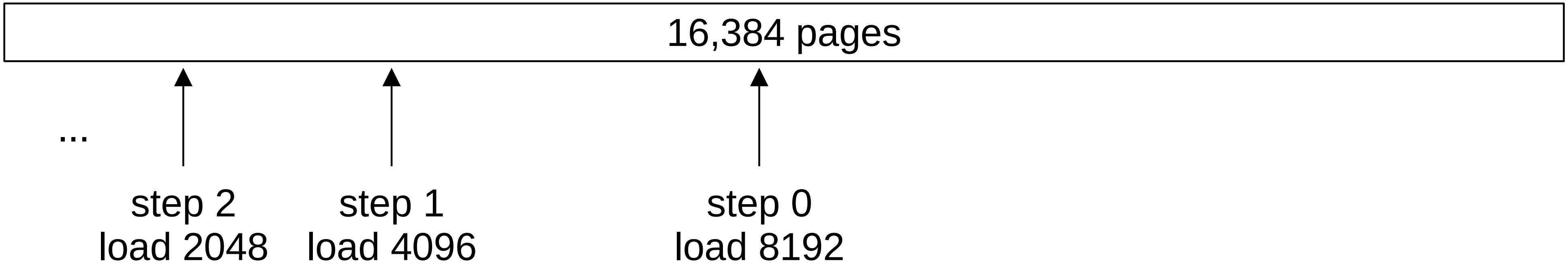
Similar rules apply to any pair of levels of the memory hierarchy.

Binary Search

- 2^{20} (~1M) Records, 64 bytes each (8 byte key, 56 byte value)
  - 64 MB of data, 16,384 4k pages, 64 records/page
- Binary Search: ~$\log(2^{20})$ = 20 steps

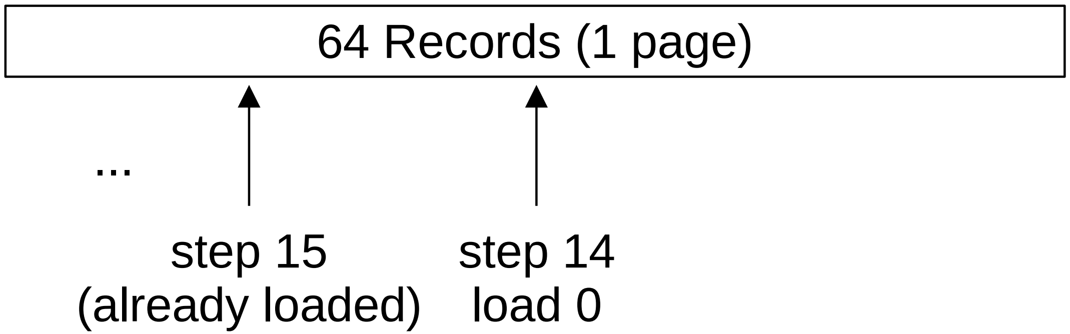
Binary Search

- 2^{20} (~1M) Records, 64 bytes each (8 byte key, 56 byte value)
  - 64 MB of data, 16,384 4k pages, 64 records/page
- Example: Binary Search (Answer: At position 0)



Binary Search

- 2^{20} (~1M) Records, 64 bytes each (8 byte key, 56 byte value)
  - 64 MB of data, 16,384 4k pages, 64 records/page
- Example: Binary Search (Answer: At position 0)



Binary Search

- 2^{20} (~1M) Records, 64 bytes each (8 byte key, 56 byte value)
  - 64 MB of data, 16,384 4k pages, 64 records/page
- Example: Binary Search (Answer: At position 0)
  - Steps 0-14 each load 1 page (15 pages loaded)
    - slooooow...
  - Steps 15-19 access the same page as step 14
    - fast!

What's the memory complexity?

How does it scale with the # of records?

Complexity

- **n** records total
- **R** record size (in Bytes)
- **P** page size (in Bytes)
- $\mathbf{C} = \lfloor {}^{\mathbf{R}}/_{\mathbf{P}} \rfloor$ records per page

Binary Search Complexity

- Overall binary search runtime:
  - log(n) steps
- Behavior goes through two stages
  - **Stage 1**: Each request goes to a new page (e.g., 0-13)
    - log(n) - log(**C**) (= log(n) - log($^{\mathbf{R}}/_{\mathbf{P}}$)) steps
  - **Stage 2**: One load for all requests (e.g., 14-19)
    - log(**C**) steps

Binary Search: Complexity

- Memory Complexity
  - **Stage 1**
    - Each page is never used again, can discard immediately
  - **Stage 2**
    - All use the same page
  - We're interested in the maximum memory use <u>at one time</u>.

The "Working Set" size is 1 page

Binary Search: Complexity

- 1 page always has 64 records
  - The last 6 binary search steps are all on the same page
- With Scaling n...
  - 2^{21} records (32GB): 21 binary search steps, 16 loads
  - 2^{22} records (64GB): 22 binary search steps, 17 loads
  - 2^{23} records (128GB): 23 binary search steps, 18 loads

Binary Search: Complexity

- IO Complexity:
  - **Stage 1**:
    - Each step does one load: O(log(n) - log(C)) = O(log(n))
  - **Stage 2**:
    - Exactly one load for the entire step: O(1)
  - Total IO is the sum of the IOs of the component steps

IO Complexity scales as $\log_2(n)$

How do we improve Binary Search?

- **Observation 1**:
  - 64 MB of 2^{20} x sizeof(key + data)

    vs
  - 2^{20} x 8B = 8 MB of keys
- **Observation 2**:
  - We don’t need to know which array index the record is at
    - ... only the page it’s on
    - ... and each page stores a contiguous range of keys

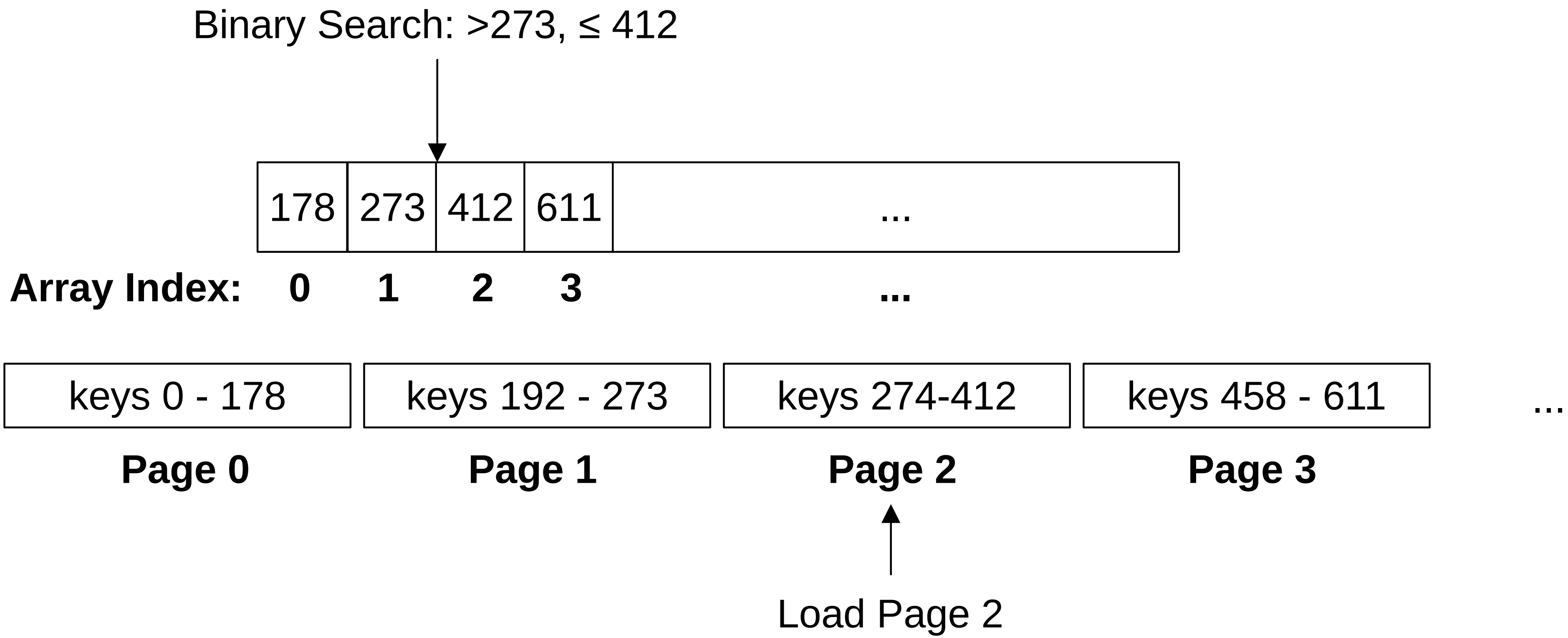
Fence Pointers

- **Idea**: Precompute the greatest key in each page in memory
  - n records; 64 records/page; $^{n}/_{64}$ keys
  - e.g., $n=2^{20}$ records; Needs 2^{14} keys
    - 2^{20} 64 byte records = 64 MB
    - 2^{14} 8 byte records = 2^{19} bytes = 512 <u>***K***</u>B
  - Call this a “Fence Pointer Table”

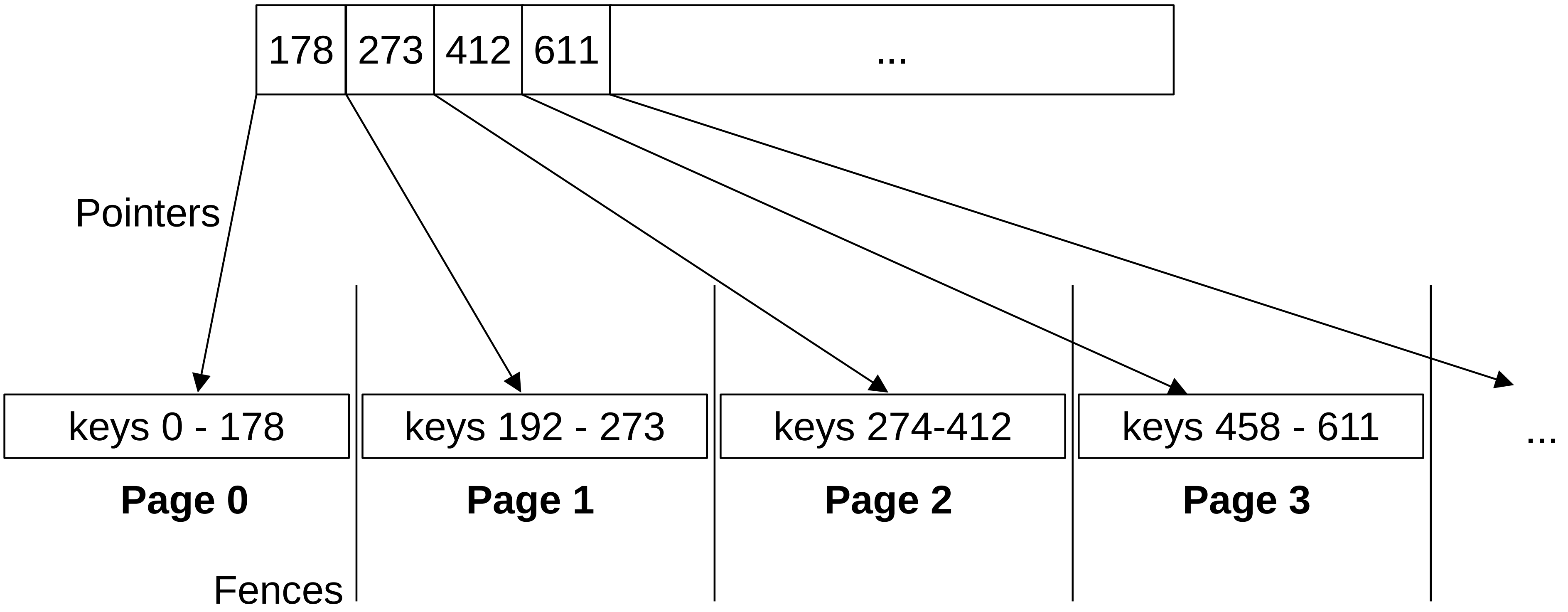
RAM: 2^{14} = 16,384 keys (Fence Pointer Table)

Disk: 16,384 pages (Actual Data)

Example


Binary Search: >273, ≤ 412
178
273
412
611
...
Array Index:
0
1
2
3
...
keys 0 - 178
keys 192 - 273
keys 274-412
keys 458 - 611
...
Page 0
Page 1
Page 2
Page 3
Load Page 2

Example (Why “fence pointer”?)



Fence Pointers

- **Step 1**: Binary Search on the Fence Pointer Table
  - All in-memory (IO complexity = 0)
- **Step 2**: Load page
  - One load (IO complexity = 1)
- **Step 3**: Binary search within page
  - All in-memory (IO complexity = 0)
- Total IO Complexity: O(1)

Fence Pointers

- Memory Complexity:
  - Need the entire fence pointer table in memory **at all times**
    - O(n / C) pages = O(n)
  - Steps 2, 3 load one more page
  - **Total**: O(n+1) = O(n)

O(n) is... not ideal