

DS 4300

# Large Scale Information Storage and Retrieval

## Foundations

Mark Fontenot, PhD  
Northeastern University

# Searching

- Searching is the most common operation performed by a database system
- In SQL, the SELECT statement is arguably the most versatile / complex.
- Baseline for efficiency is **Linear Search**
  - Start at the beginning of a list and proceed element by element until:
    - You find what you're looking for
    - You get to the last element and haven't found it

# Searching

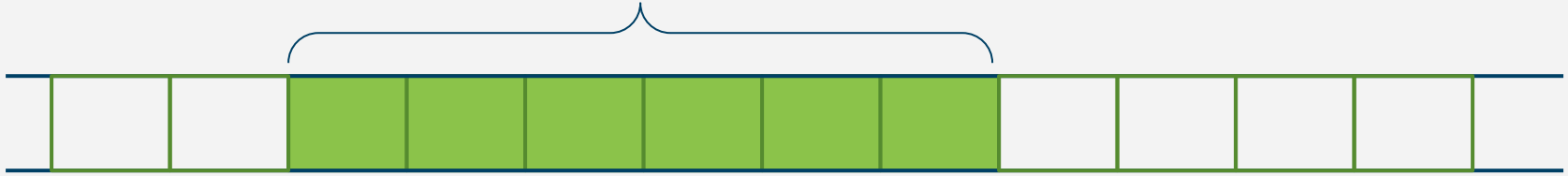
- **Record** - A collection of values for attributes of a single entity instance; a row of a table
- **Collection** - a set of records of the same entity type; a table
  - Trivially, stored in some sequential order like a list
- **Search Key** - A value for an attribute from the entity type
  - Could be  $\geq 1$  attribute

# Lists of Records

- If each record takes up  $x$  bytes of memory, then for  $n$  records, we need  $n \cdot x$  bytes of memory.
- Contiguously Allocated List
  - All  $n \cdot x$  bytes are allocated as a single “chunk” of memory
- Linked List
  - Each record needs  $x$  bytes + additional space for 1 or 2 memory addresses
  - Individual records are linked together in a type of chain using memory addresses

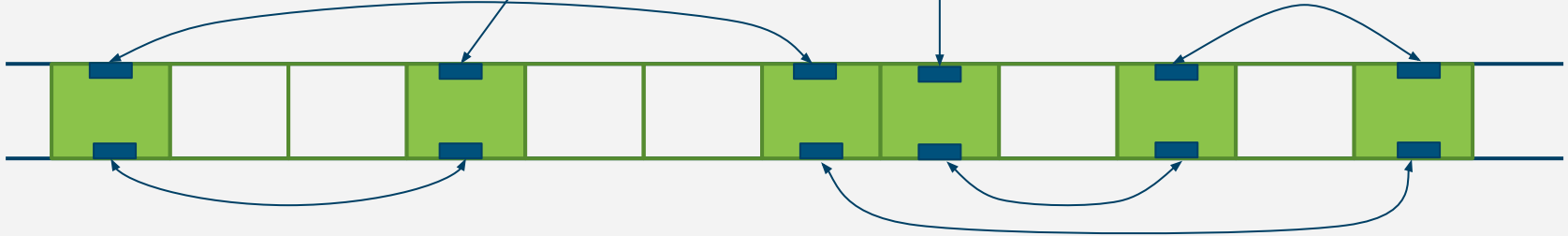
# Contiguous vs Linked

6 Records Contiguously Allocated - **Array**



front

back

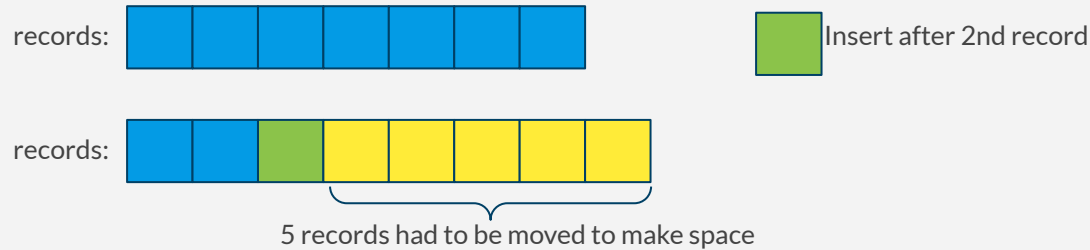


Extra storage for a  
memory address

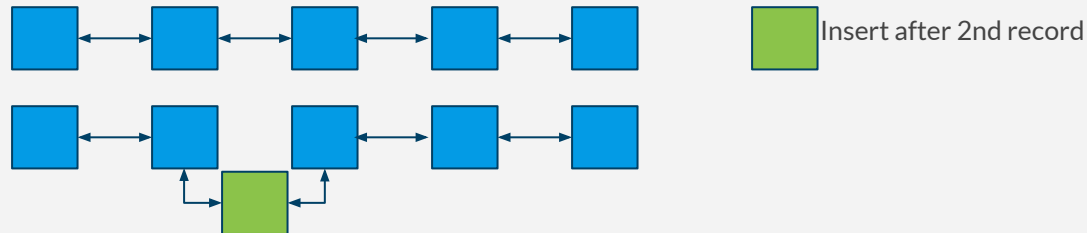
6 Records Linked by memory addresses - **Linked List**

# Pros and Cons

- Arrays are faster for random access, but slow for inserting anywhere but the end



- Linked Lists are faster for inserting anywhere in the list, but slower for random access



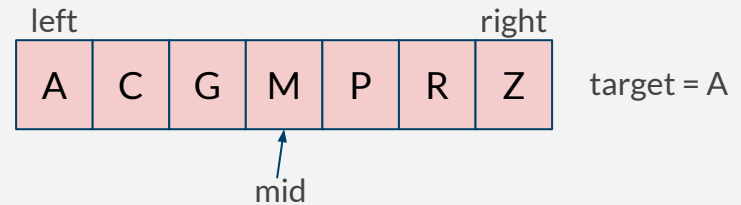
# Observations:

- Arrays
  - fast for random access
  - slow for random insertions
- Linked Lists
  - slow for random access
  - fast for random insertions

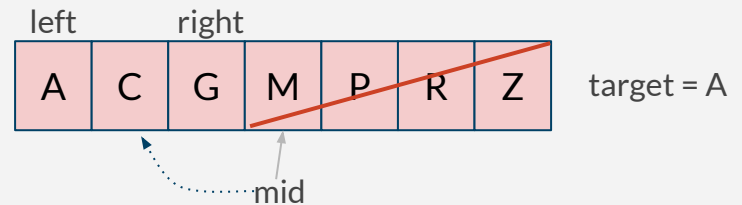
# Binary Search

- Input: array of values in sorted order, target value
- Output: the location (index) of where target is located or some value indicating target was not found

```
def binary_search(arr, target)
    left, right = 0, len(arr) - 1
    while left <= right:
        mid = (left + right) // 2
        if arr[mid] == target:
            return mid
        elif arr[mid] < target:
            left = mid + 1
        else:
            right = mid - 1
    return -1
```



Since `target < arr[mid]`, we reset `right` to `mid - 1`.





# Time Complexity

- Linear Search

- Best case: target is found at the first element; only 1 comparison
- Worst case: target is not in the array;  $n$  comparisons
- Therefore, in the worst case, linear search is  $O(n)$  time complexity.

- Binary Search

- Best case: target is found at  $mid$ ; 1 comparison (inside the loop)
- Worst case: target is not in the array;  $\log_2 n$  comparisons
- Therefore, in the worst case, binary search is  $O(\log_2 n)$  time complexity.

# Back to Database Searching

- Assume data is stored on disk by column id's value
- Searching for a specific id = fast.
- But what if we want to search for a specific *specialVal*?
  - Only option is linear scan of that column
- Can't store data on disk sorted by both id and specialVal (at the same time)
  - data would have to be duplicated → space inefficient

id	specialVal
1	55
2	87
3	50
4	108
5	122
6	149
7	145
8	120
9	50
10	83
11	128
12	117
13	119
14	119
15	51
16	85
17	51
18	145
19	73
20	73

# Back to Database Searching

- Assume data is stored on disk by column id's value
- Searching for a specific value
- But, if we search by *specialVal*, we need to scan the entire column
- Cache the *specialVal* values in memory
  - data would have to be duplicated → space inefficient

We need an external data structure to support faster searching by *specialVal* than a linear scan.

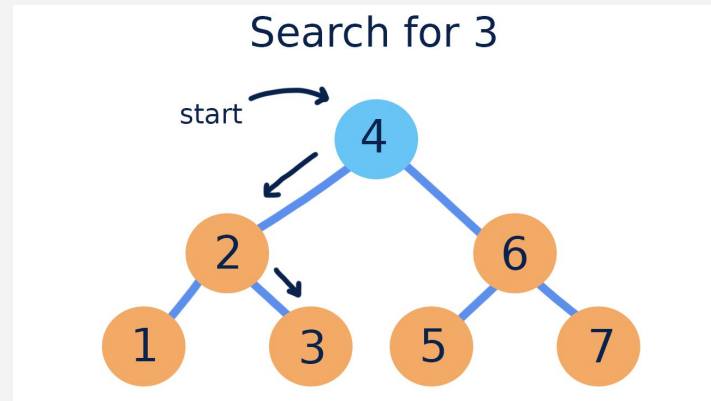
id	specialVal
1	55
2	87
3	50
4	108
5	122
6	149
7	145
8	120
9	50
10	83
11	128
12	117
13	119
14	119
15	51
16	85
17	51
18	145
19	73
20	73

# What do we have in our arsenal?

- 1) An array of tuples (specialVal, rowNumber) sorted by specialVal
  - a) We could use Binary Search to quickly locate a particular specialVal and find its corresponding row in the table
  - b) But, every insert into the table would be like inserting into a sorted array - slow...
- 2) A linked list of tuples (specialVal, rowNumber) sorted by specialVal
  - a) searching for a specialVal would be slow - linear scan required
  - b) But inserting into the table would theoretically be quick to also add to the list.

# Something with Fast Insert and Fast Search?

- Binary Search Tree - a binary tree where every node in the left subtree is less than its parent and every node in the right subtree is greater than its parent.



# To the Board!