DS4300: FOUNDATIONS

Searching

- Searching is the most common operation in a database system.
- **SQL SELECT Statement:** The most versatile and complex SQL command used for querying data.
- **Baseline Efficiency: Linear Search**
  - Starts at the beginning of a list.
  - Proceeds element by element until:
    - The desired item is found.
    - The end of the list is reached without finding the item.
- **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 >= 1 attribute

List of Records

- If each record takes up x bytes of memory, then for n records, we need ***n*x*** bytes of memory.
- *Contiguously Allocated List*
  - All ***n*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 Allocation

- ***Contiguous***
  - Data is stored in **one continuous block** of memory.
  - Example: **Arrays** in programming.
  - **Pros:**
    - Fast access for random access
  - **Cons:**
    - Slow for random insertions except at the end of the array
- ***Linked***
  - Data is stored in **separate, non-adjacent memory locations**, with pointers connecting them.
  - Example: **Linked Lists** in programming.
  - **Pros:**
    - Faster for random insertion
  - **Cons:**
    - Slower access

Binary Search

- **Input =** Array of values in sorted order, target value
- **Output =** The location (index) of where the target is located or some value indicating the target was not found
- ***Sample code:***

```
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
```

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; log2n comparisons.
  - Therefore, in the worst case, binary search is O(log2n) time complexity.

Back to Database Searching

- Assume data is stored on disk by column *id*'s value
- Searching for a specific *id* = fast
- A linear scan is the only option when searching for a specific *specialVal*.
- Data can't be stored on a disk sorted by both *id* and s*pecialVal* simultaneously.
- Sorting by both would require duplicating data, which is space-inefficient.
- ***Solution:*** Use an external data structure to enable faster searching for *specialVal* instead of relying on a linear scan
- ***Options***:
  - An array of tuples (specialVal, rowNumber) sorted by specialVal

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