2, 3 - Arrays, Records & Pointers

Multidimensional Array

Pointer Array

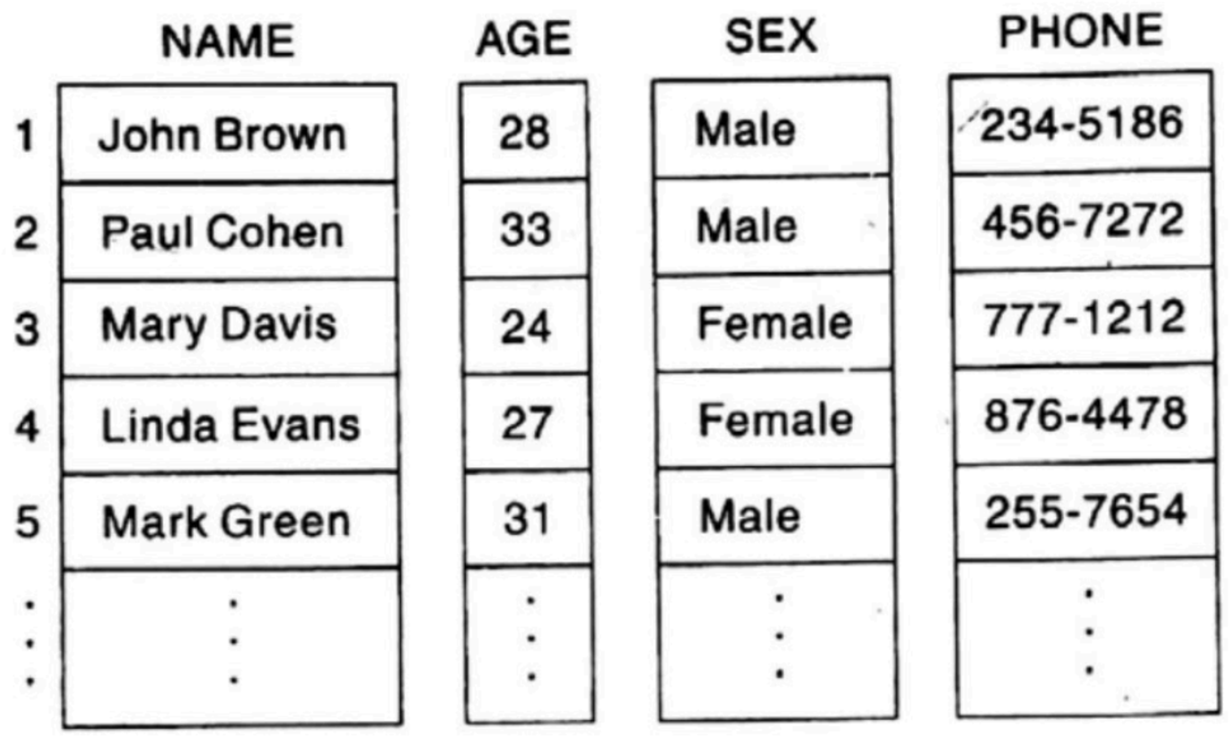
Same as normal array just store the pointer of a element

Record Structure

- Collection of non-homogenous data.
- Indexed by attribute name, not by position.
  Example:

```
struct Student {
    string name;
    int roll;
    float marks;
};
```

Representation of records in Memory: Parallel Arrays



	NAME	AGE	SEX	PHONE
1	John Brown	28	Male	234-5186
2	Paul Cohen	33	Male	456-7272
3	Mary Davis	24	Female	777-1212
4	Linda Evans	27	Female	876-4478
5	Mark Green	31	Male	255-7654
⋮	⋮	⋮	⋮	⋮

Matrices

- The time complexity of Matrix Multiplication ---> $O(n^3)$

Sparse Matrices

One kind of matrix where most of the elements are zero.

Data Structure

Classifications:

- **Linear** --> elements form a sequence (i.e. array, linked list)
- **Non-linear** --> elements are not arranged sequentially.

Operations:

- **Traversal** --> processing each element in the list
- **Search** --> finding the location of the element based on given value.
- Insertion -- > adding a new element to the list.
- **Deletion** --> removing an element from the list.
- **Sorting** --> arranging the element in some type of order.
- **Merging** --> combining to lists into a single list.

Linear Arrays

A list of a finite number of homogeneous data elements.

Representation of Linear Arrays in Memory

Formula:

$$\boxed{LOC(LA[k]) = BASE(LA) + w \times (k - \text{lower bound})}$$

Here,

- $LOC(LA[k]) =$ address of the element $LA[k]$ of the array LA.
- $BASE(LA) =$ Base address of LA.
- $w =$ size of each element in memory (in bytes).
- $\text{lower bound} =$ the index of the first element of the array.

Traversing Linear Arrays

- Loop from starting index to the last index.

- Time Complexity --> $O(n)$.

Inserting

- First move the last element to it's next position. Do this from last to $k+1\,th$ position. Now the $k-th$ position is free to add new element.
- Time Complexity --> $O(n)$.

Deleting

- First, remove the element at the $k-th$ position. Then, shift all elements after k one position to the left to fill the empty spot. This frees up the last position in the array.
- Time Complexity --> $O(n)$.

Bubble Sort

- Repeatedly compare adjacent elements and swap them if they are in the wrong order. After each pass, the largest element to the end. Repeat for the remaining unsorted part of the array until the array is sorted.
- Time Complexity:
    - Worst case --> $O(n^2)$.
    - Best case --> $O(n)$.
- Space Complexity --> $O(1)$.

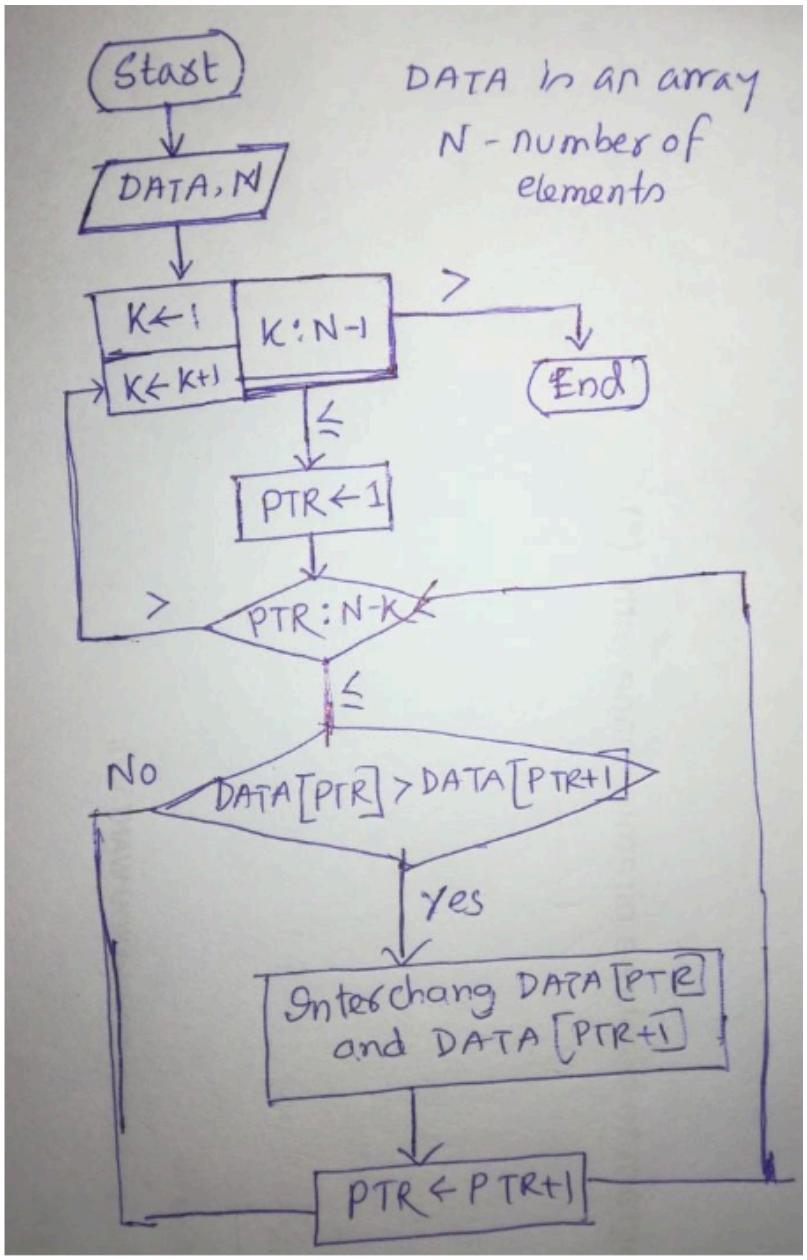
Algorithm:

1. Repeat steps 2 & 3 for $K = 1$ to $N - 1$.

2.
   ```
       Set $PTR := 1$.
   ```

3.
   ```
       Repeat while $PTR \le N - K$:
   1. If $DATA[PTR] > DATA[PRT+1]$, then:
          Interchange $DATA[PTR]$ and $DATA[PTR + 1]$.
   2. set $PTR := PTR+1$.
   ```
4. Exit.

Flowchart:



Searching

Linear Search

- Start from the first element and compare each element sequentially with the target. If the element is found return its position, else continue to the next element. If the end of the array is reached element not found.
- **Time Complexity:** $O(n)$.

Algorithm:

1. Set $DATA[N+1] := ITEM$.
2. Set $LOC := 1$.

3. Repeat while $DATA[LOC] \neq ITEM$:
   Set $LOC := LOC + 1$.
4. If $LOC = N + 1$, then Set $LOC := 0$.
5. Exit.

Binary Search

- Works on a sorted array.
- Repeatedly divide the array into two halves:
  1. Find the **middle element**.
  2. If middle = target → **found**.
  3. If middle > target → search in **left half**.
  4. If middle < target → search in **right half**.
     Repeat until the element is found or the search space becomes empty.
- **Time Complexity:** $O(log_2 n)$.
- **Space Complexity:** $O(1)$.

Algorithm:

1. Set $BEG := LB$, $END := UB$ and $MID = (BEG + END)/2$.
2. Repeat steps 3 and 4 while $BEG \leq END$ and $DATA[MID] \neq ITEM$.
3.
```
      IF $ITEM<DATA[MID]$, then:
    Set $END:=MID - 1$.
Else:
    Set $BEG:=MID+1$.
```
4.
```
      Set $MID:=(BEG+END) / 2$.
```
5. Exit.

Limitation:

- Requires a sorted list to work.
- Maintaining a sorted array is costly if there are frequent insertions and deletions. In such cases, **other data structures** (e.g., linked lists, trees) may be preferred.