COM 201 – Data Structures and Algorithms

Arrays, Records and Pointers

Assist. Prof. Özge ÖZTİMUR KARADAĞ

Department of Computer Engineering – ALKÜ

Alanya

Linear vs. Nonlinear Structures

- A data structure is linear if its elements form a sequence, or, a linear list. Linear structures can be represented in memory in two ways:
  - Sequential memory locations, → arrays
  - Represent linear relationship between elements by pointers or links → linked lists

- Nonlinear structures → trees, graphs

Operations on linear structures

- Traversal
- Search
- Insertion
- Deletion
- Sorting
- Merging

Linear Arrays

- A linear array is a list of a finite number n of homogeneous data elements (i.e., data elements of the same type) such that:
  - The elements of the array are referenced respectively by an index set consisting of n consecutive numbers.
  - The elements of the array are stored respectively in successive memory locations.

- The number n of elements is called the length or size of the array.
- In general, the length or the number of data elements of the array can be obtained from the index set by the formula:

  - Length = UB-LB+1

  UB is the largest index, called the upper bound,
  LB is the smallest index, called the lower bound, of the array.
    - When LB=1, length =UB

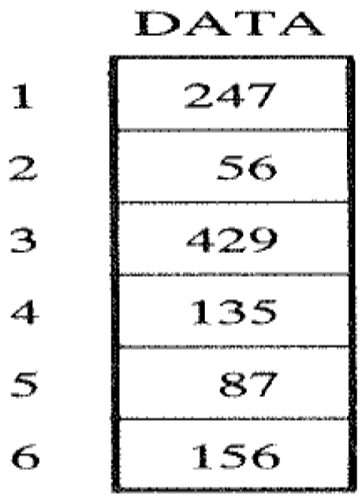
Linear Array

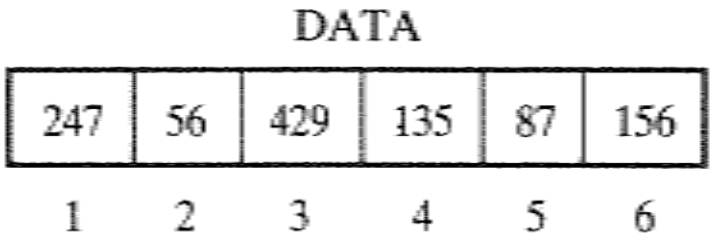
- The elements of an array A may be denoted by the subscript notation
  - $A_1, A_2, A_3, \ldots, A_n$

  or
  - A(1), A(2),…, A(N)

  or
  - A[1], A[2], A[3],…, A[N]

- example: DATA[1]=247, DATA[2]=56, DATA[3]=429, DATA[4]=135, DATA[5]=87, DATA[6]=156





Linear Array

- Each programming language has its own rules for declaring arrays. Each such declaration must give, three items of information:
  - The name of the array
  - The data type of the array
  - The index set of the array

  - Ex: Defining array in C
    - int intArray[20];

  - Ex: Defining array in Java

```
int intArray[]; //declaring array
intArray = new int[20]; // allocating memory to array
```

OR

```
int[] intArray = new int[20]; // combining both statements in one
```

Representation of Linear Arrays in Memory

- Let LA be a linear array in the memory of the computer. Recall that the memory of the computer is simply a sequence of addressed locations.
  - LOC(LA[K])=address of the element LA[K] of the array LA
  - The elements of LA are stored in successive memory cells. Accordingly, the computer does not need to keep track of the address of every element of LA, but needs to keep track only of the address of the first element of LA, denoted by

    Base(LA)

    and called the base address of LA. Using this address Base(LA), the computer calculates the address any element of LA by the following formula:

    LOC(LA[K])=Base(LA)+w(K-lower bound),

    Where w is the number of words per memory cell for the array LA.
  - Observe that the time to calculate LOC(LA[K]) is essentially the same for any value of K.
  - Given any subscript K, one can locate and Access the content of LA[K] without scanning any other element of LA.

Traversing Linear Arrays

- Let A be a collection of data elements stored in the memory of the computer.
- Following can be accomplished by traversing:
  - To print the contents of each element of A,
  - To count the number of elements of A with a given property.

Traversing Linear Arrays

- Algorithm for traversing a linear array:

(Traversing a Linear Array) Here LA is a linear array with lower bound LB and upper bound UB. This algorithm traverses LA applying an operation PROCESS to each element of LA.

1. [Initialize counter.] Set K := LB.
2. Repeat Steps 3 and 4 while K ≤ UB.
3. [Visit element.] Apply PROCESS to LA[K].
4. [Increase counter.] Set K := K + 1.
   [End of Step 2 loop.]
5. Exit.

(Traversing a Linear Array) This algorithm traverses a linear array LA with lower bound LB and upper bound UB.

1. Repeat for K = LB to UB:
   Apply PROCESS to LA[K].
   [End of loop.]
2. Exit.

Inserting and Deleting

- Inserting: Operation of adding an element to the linear array A
- Deleting: Operation of removing one of the elements from A.
- What to deal with?
- Inserting:
  - If the memory space allocated for the array is large enough, insertion can be done easily.
  - To insert in the middle of the array; on the average, half of the elements must be moved downward to new locations.
- Deleting:
  - Deleting an element at the end of an array can be done easily.
  - Deleting an element somewhere in the middle of the array would require that each element be moved one location upward in order to fill up the array.

Inserting and Deleting

- Inserting:
  - INSERT (LA,N,K,ITEM)
    - Function to insert the element ITEM into the Kth position in Linear array LA with N. elements

(Inserting into a Linear Array) INSERT(LA, N, K, ITEM)
Here LA is a linear array with N elements and K is a positive integer such tha
$K \leq N$. This algorithm inserts an element ITEM into the Kth position in LA

1. [Initialize counter.] Set $J := N$.
2. Repeat Steps 3 and 4 while $J \geq K$.
3. [Move Jth element downward.] Set $LA[J + 1] := LA[J]$.
4. [Decrease counter.] Set $J := J - 1$.
   [End of Step 2 loop.]
5. [Insert element.] Set $LA[K] := ITEM$.
6. [Reset N.] Set $N := N + 1$.
7. Exit

Inserting and Deleting

- Deleting:
  - DELETE(LA,N,K,ITEM) Delete the Kth element from a linear array LA with N elements.

(Deleting from a Linear Array) DELETE(LA, N, K, ITEM)
Here LA is a linear array with N elements and K is a positive integer such that $K \leq N$. This algorithm deletes the Kth element from LA.

1. Set ITEM := LA[K].
2. Repeat for J = K to N − 1:
   [Move J + 1st element upward.] Set LA[J] := LA[J + 1].
   [End of loop.]
3. [Reset the number N of elements in LA.] Set N := N − 1.
4. Exit.

Sorting

- Sorting: Operation of rearranging the element of A so they are in increasing order, so that
  - A[1]<A[2]<….<A[N]
- Bubble Sort:
  - Suppose the list of numbers A[1], A[2],…,A[N] is in memory.
    - Step 1: Compare A[1] and A[2] and arrange them in the desired order, so that A[1]<A[2]. Then compare A[2] and A[3] and arrange them so that A[2]<A[3]. Then compare A[3] and A[4] and arrange them so that A[3]<A[4]. Continue until we compare A[N-1] with A[N] and arrange them so that A[N-1]<A[N].
      - When step 1 is completed, A[N] will contain the largest element.
    - Step 2: Repeat step 1 with one less comparison; that is now we stop after we compare and possible rearrange A[N-2] with A[N-1].
    - Step 3: Repeat step 1 with two fewer comparisons; that is, we stop after we compare and possible rearrange A[N-3] and A[N-2].
    - ….
    - Step N-1: Compare A[1] with A[2] and arrange them so that A[1]<A[2].
  - The Bubble sort algorithm requires n-1 passes for n number of items.

Bubble Sort

(Bubble Sort) BUBBLE(DATA, N)
Here DATA is an array with N elements. This algorithm sorts the elements in DATA.

1. Repeat Steps 2 and 3 for K = 1 to N − 1.
2. Set PTR := 1. [Initializes pass pointer PTR.]
3. Repeat while PTR ≤ N − K: [Executes pass.]
   (*a*) If DATA[PTR] > DATA[PTR + 1], then:
   Interchange DATA[PTR] and DATA[PTR + 1].
   [End of If structure.]
   (*b*) Set PTR := PTR + 1.
   [End of inner loop.]
   [End of Step 1 outer loop.]
4. Exit.

Bubble Sort

- Example:

32,51,27,85,66,23,13,57

Bubble Sort

- Complexity of Bubble Sort:
  - The time for a sorting algorithm is measured in terms of the number of comparisons.
  - The number of comparisons f(n) for bubble sort:
    - n-1 comparisons during the first pass
    - n-2 comparisons during the second pass
    - …
    - $f(n) = (n-1) + (n-2) + \cdots + 2 + 1 = \frac{n(n-1)}{2} = \frac{n^2}{2} + O(n) = O(n^2)$

Searching

- Let DATA be a collection of data elements in memory, and suppose a specific ITEM of information is given.
- Searching refers to the operation of finding the location LOC of ITEM in DATA, or printing some message that ITEM does not appear there.
- The complexity of searching algorithms is measured in terms of the number f(n) of comparisons required to find ITEM in DATA where DATA contains n elements.

Searching

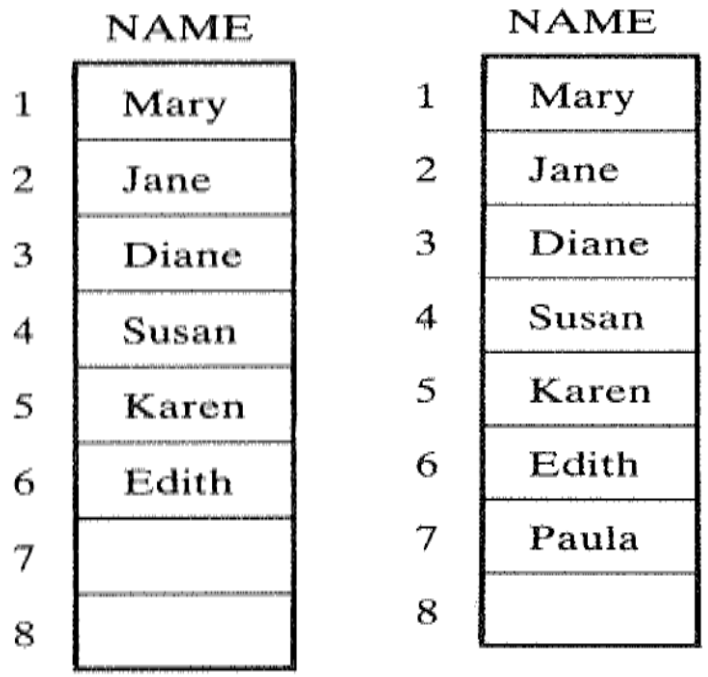
- Linear Search:
  - Suppose DATA is a linear array with n elements, search for a given ITEM in DATA,
  - Compare ITEM with each element of DATA one by one.

(Linear Search) LINEAR(DATA, N, ITEM, LOC)
Here DATA is a linear array with N elements, and ITEM is a given item of information. This algorithm finds the location LOC of ITEM in DATA, or sets LOC := 0 if the search is unsuccessful.

1. [Insert ITEM at the end of DATA.] Set DATA[N + 1] := ITEM.
2. [Initialize counter.] Set LOC := 1.
3. [Search for ITEM.]
   Repeat while DATA[LOC] ≠ ITEM:
       Set LOC := LOC + 1.
   [End of loop.]
4. [Successful?] If LOC = N + 1, then: Set LOC := 0.
5. Exit.

Searching

- Example Linear Search:
  - Search for Paula in the following array with n=6



  - The algorithm will find NAME[7]=Paula, which implies that the Paula is not in the original array.

Complexity of Linear Search

- Worst case: search through the entire array DATA, f(n)=n+1
- Average case: search half of the array f(n)=n/2

- Complexity of linear search is proportional to the number of elements in the array.

Binary Search

- Suppose DATA is an array which is stored in increasing numerical order.

(Binary Search) BINARY(DATA, LB, UB, ITEM, LOC)
Here DATA is a sorted array with lower bound LB and upper bound UB, and ITEM is a given item of information. The variables BEG, END and MID denote, respectively, the beginning, end and middle locations of a segment of elements of DATA. This algorithm finds the location LOC of ITEM in DATA or sets LOC = NULL.

1. [Initialize segment variables.]
   Set BEG := LB, END := UB and MID = INT((BEG + END)/2).
2. Repeat Steps 3 and 4 while BEG ≤ END and DATA[MID] ≠ ITEM.
3. If ITEM < DATA[MID], then:
   Set END := MID − 1.
   Else:
   Set BEG := MID + 1.
   [End of If structure.]
4. Set MID := INT((BEG + END)/2).
   [End of Step 2 loop.]
5. If DATA[MID] = ITEM, then:
   Set LOC := MID.
   Else:
   Set LOC := NULL.
   [End of If structure.]
6. Exit.

Example

- Binary Search
  - DATA=11, 22, 30, 33, 40, 44, 55, 60, 66, 77, 80, 88, 99

    - ITEM=40

Complexity of Binary Search

- The complexity is measured by the number f(n) of comparisons to locate ITEM in DATA where DATA contains n elements. Each comparison reduces the sample size in half. We require at most f(n) comparisons to locate ITEM where
  - $2^{f(n)} > n$ or equivalently $f(n) = \lfloor log_2 n \rfloor + 1$

  - Ex. DATA contains 1000000 elements.
    - $2^{20} > 1000000$
    - 20 comparisons will be enough to find the location of an item in a data array with 1000000 elements.

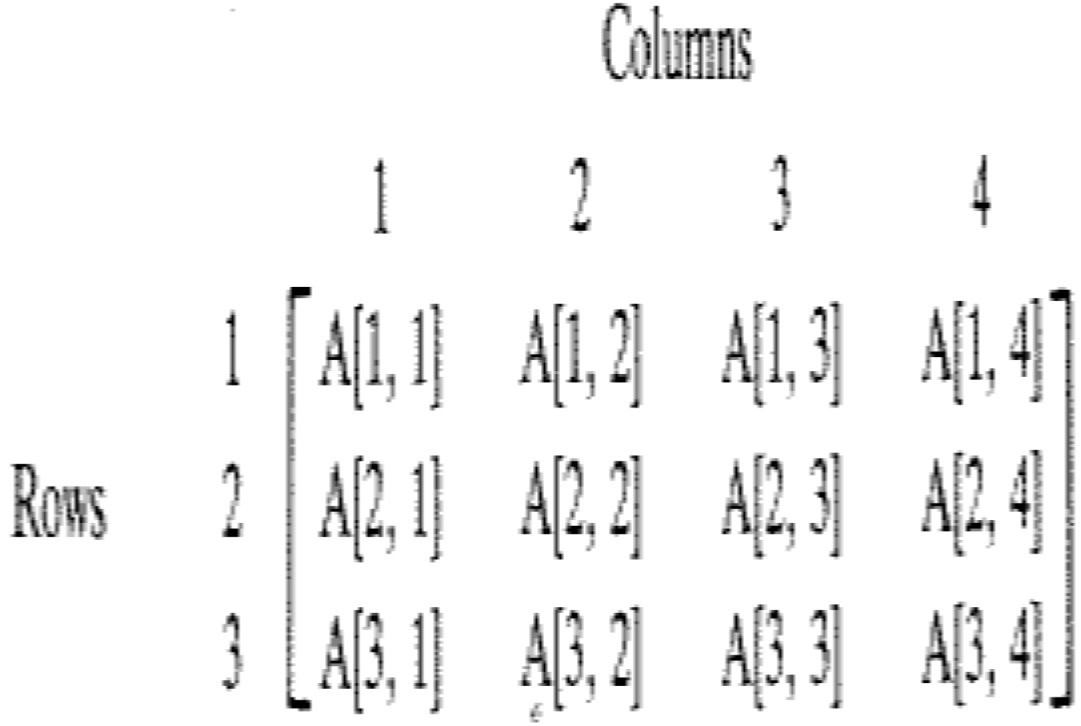
Searching

- Limitations of Binary Search:
  - It requires two conditions:
    - The list must be sorted,
    - One must have direct Access to the middle element in any sublist.

    - This mens that one must use a sorted array to hold data. But keeping data in a sorted array is normally very expensive when there are many insertions and deletions. Other data structures such as linked list or Binary search tree can be more effective in that case.

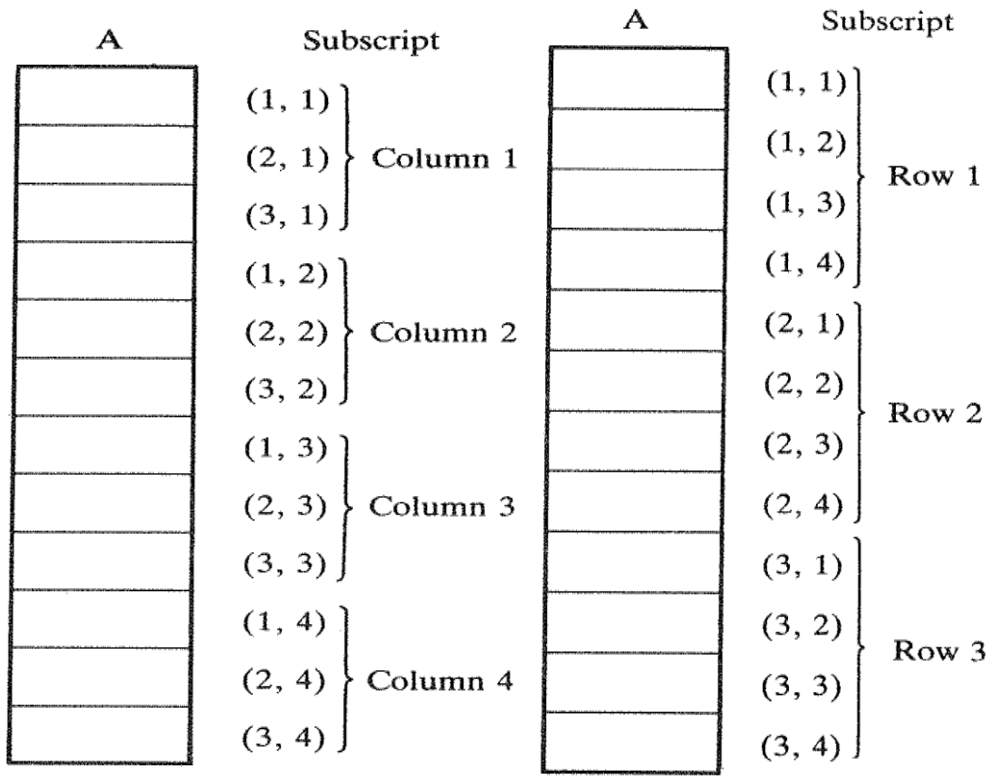
Multidimensional Arrays

- Two-dimensional mxn array A



Multidimensional Arrays

- Representation of two-dimensional arrays in memory
  - Column-major order
  - Row-major order



(*a*) Column-major order. (*b*) Row-major order.

Multidimensional Arrays

- For linear array computer keep track of the address LOC(LA[K]) by this formula:
  - LOC(LA[K])=Base(LA)+w(K-1)
  - w is the number of words per memory cell for the array LA and 1 is the lower bound of the index set LA.
- For two-dimensional mxn array A,
  - Column-major order LOC(A[J,K])=Base(A)+w[M(K-1)+(J-1)]
  - Row-major order LOC(A[J,K])=Base(A)+w(N(J-1)+(K-1)]

Multidimensional Arrays

- Example:
  - Consider the array SCORE with size 25x4. Suppose Base(SCORE)=200 and there are w=4 words per memory cell. Programming language stores two-dimensional arrays using row-major order. Find the address of SCORE[12,3]

  - LOC(SCORE[12,3])=200+4[4(12-1)+(3-1)]=200+4[46]=384

Pointers; Pointer Arrays

- Let DATA be any array.
- A variable P is called a pointer if P 'points' to an element in DATA, i.e is P points the address of an element in DATA.

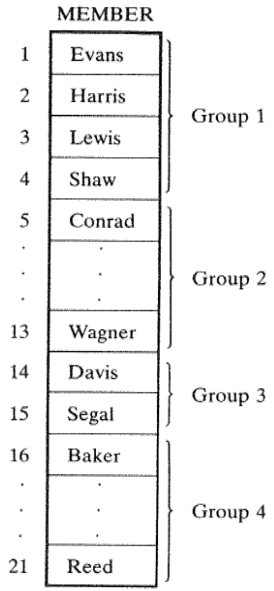
- How to store this data?
  - Two-dimensional array?
  - linear- array?

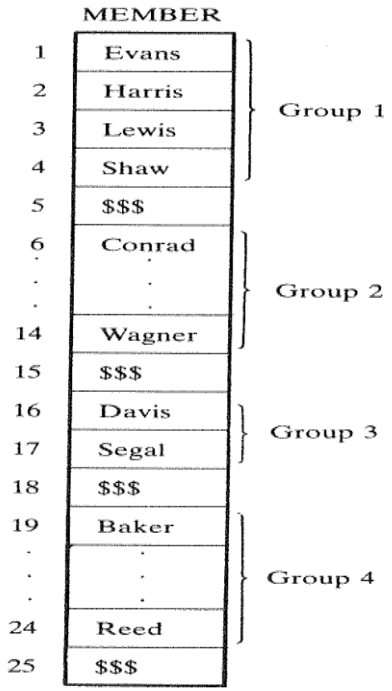
Group 1	Group 2	Group 3	Group 4
Evans	Conrad	Davis	Baker
Harris	Felt	Segal	Cooper
Lewis	Glass		Ford
Shaw	Hill		Gray
	King		Jones
	Penn		Reed
	Silver		
	Troy		
	Wagner		

Pointers; Pointer Arrays

Group 1	Group 2	Group 3	Group 4
Evans	Conrad	Davis	Baker
Harris	Felt	Segal	Cooper
Lewis	Glass		Ford
Shaw	Hill		Gray
	King		Jones
	Penn		Reed
	Silver		
	Troy		
	Wagner		

- -How to store this data?
  - Two-dimensional array?
    - Not space-efficient
  - Linear- array?
    - No way to find names in a specified group



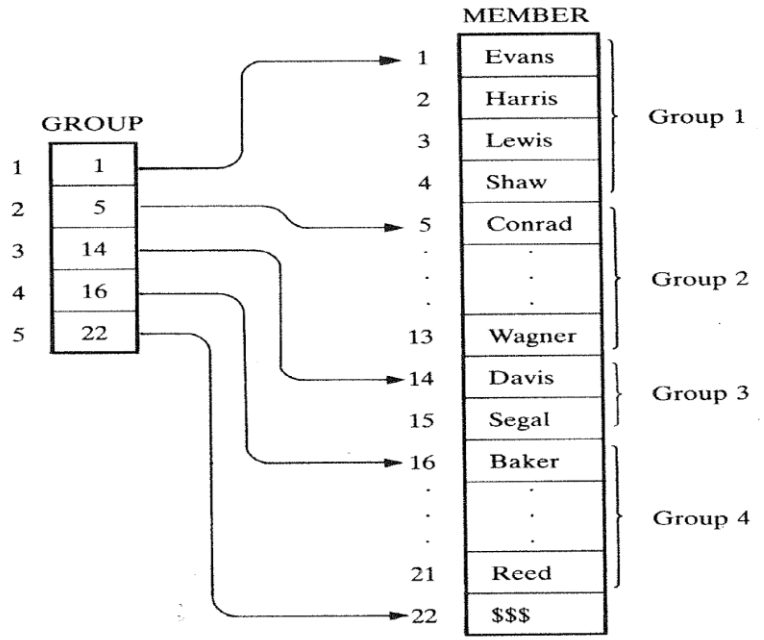


Any suggestions?

Place a marker to indicate the end of each group. Few extra memory cells are used, but it is possible to Access any particular group.

Pointer Arrays

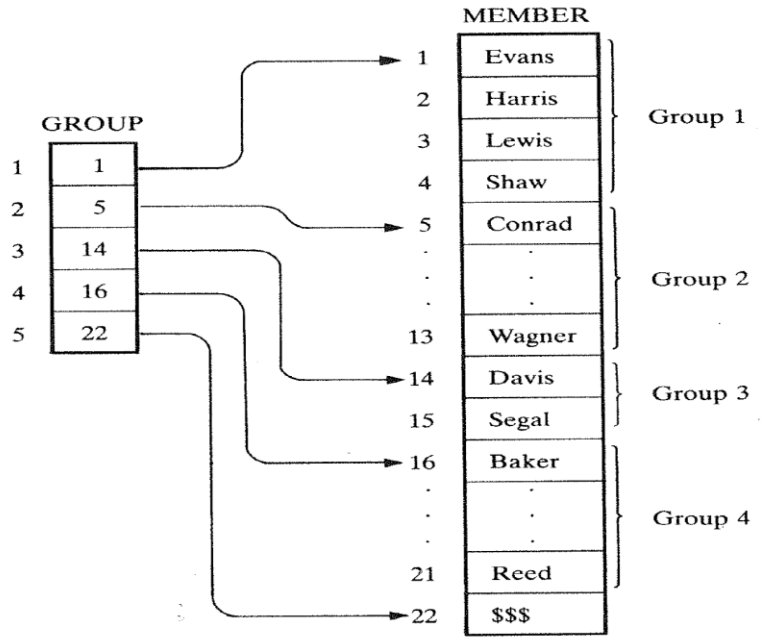
- GROUP is a pointer array which contains the locations of the different groups



GROUP[L] contains the first element in group L
GROUP[L+1]-1 contains the last element in group L

Pointer Arrays

- Example: to print the names in the Lth group??



GROUP[L] contains the first element in group L
GROUP[L+1]-1 contains the last element in group L

Set FIRST := GROUP[L] and LAST := GROUP[L + 1] – 1.
Repeat for K = FIRST to LAST:
    Write: MEMBER[K].
[End of loop.]
Return.

Records

- A record is a collection of related data items, each of which is called a field or attribute,
- A file is a collection of similar records.
- A record differs from a linear array in the following ways:
  - A record may be a collection of nonhomogeneous data, the data items in a record may have different data types,
  - The data items in a record are indexed by attribute names, so there may not be a natural ordering of its elements.

Records

- Structures in C:
  - Definition of the struct

```
struct Person {
  char name[50];
  int citNo;
  float salary;
};
```

  - Creating a struct variable:

```
struct Person {
  // code
};

int main() {
  struct Person person1, person2, p[20];
  return 0;
}
```

```
struct Person {
   // code
} person1, person2, p[20];
```

- Accessing members of a struct:

```
person1.citNo = 1984;
person1. salary = 2500;
```

Sample Question - 1

- Write a program in C which accepts 10 integers and sorts them using the Bubble Sort Algorithm.
  - Extend your program to accept new integers after sorting.
  - Use linear array in your program. And traverse the array by
    - array indexing
    - Pointers
    - To search for a given integer, if the integer is not in the array, insert it to the sorted array.

Sample Question - 2

- Write a C program which accepts the name, age and salary information of 10 employees.
  - Store this data in a struct array.
  - Sort the employee by their age.
  - Write a function ‘get_salary’ which returns the salary for a given employee name. (implement this function first by array indexing, then by pointer.)

References

- Shaum’s Outline of Theory and Problems of Data Structures, Seymour Lipschutz, McGraw-Hill.