

CHAPTER 4

Arrays, Records and Pointer

Introduction

- Array: Array is one kind of linear structure where the linear relationship between elements are maintained by means of Sequential memory location.
- Operations performed on any linear structure(Array or linked list) are as follows:
 - Traversal: Processing every element.
 - Search: Finding the location of the element with a given search parameter such a value or a key.
 - Insertion: Adding a new element to the list.
 - Deletion: Removing an existing element from the list.
 - Sorting: Arranging the elements in some order, ascending or descending
 - Merging: Combining two lists into a single one.

Linear Array

- A linear array is a list of finite no. n of homogeneous (same type) data elements. Such that:
 - The elements of the array are references respectively by an index set consisting of n consecutive numbers.
 - The elements are stored respectively in successive memory locations.
- The no. of elements of an array can be found by the following formula:

$$\text{Length} = \text{UB} - \text{LB} + 1$$

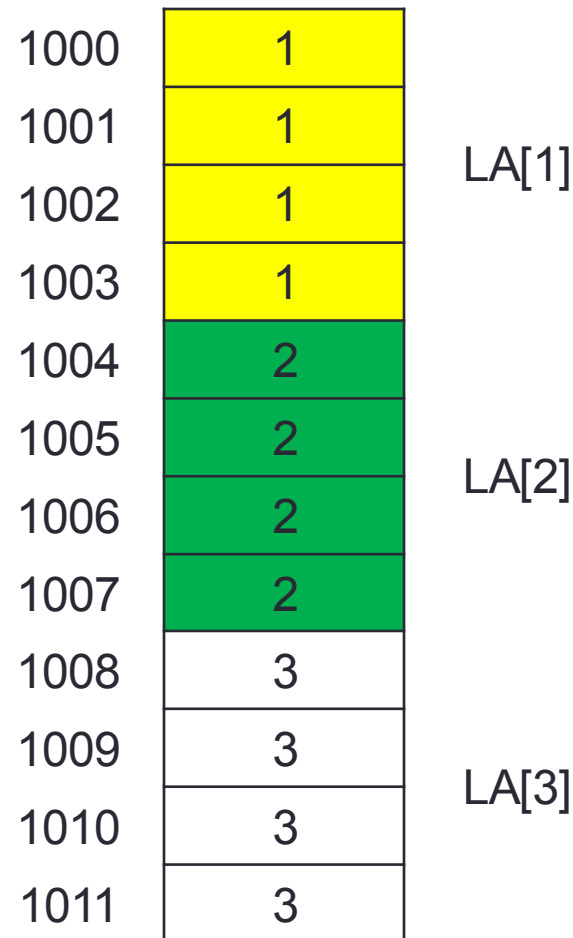
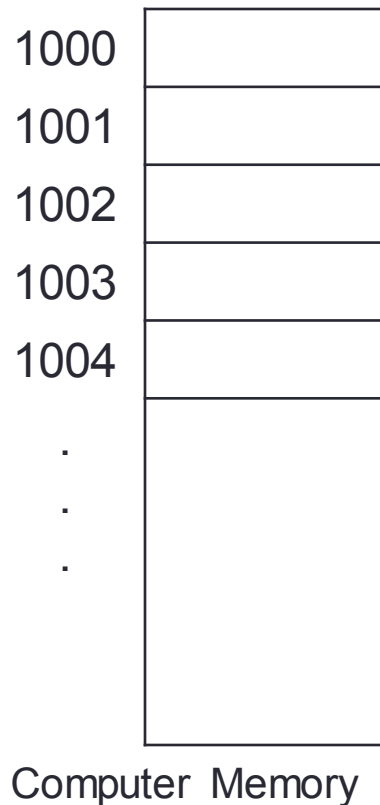
where UB is Upper bound and LB is Lower bound index.

- Ex: If array Data has upper index 360 and lower index 100 then $\text{length} = 360 - 100 + 1 = 261$

Representation of Linear Array in Memory

- Array elements are stored in sequential memory location. So it is not necessary to remember the location of every elements.
- Only the first element's address is need to be memorized.
- The Successive elements address can be computed as follows:
 - $LOC(LA[k]) = Base(LA) + w(k - LB)$
where $LOC(LA[k])$ is the location of the element $LA[k]$, $Base(LA)$ is the address of the first element and w is the no. of words per memory cell.

Representation of Linear Array in Memory



LA array with $w=4$

Traversing a Linear Array

- Traversing means accessing or processing each element of an array exactly once. This can be printing elements, summing the elements, finding sine value etc.
- The most common algorithm for Traversing an array LA with LB and UB is given below.
 1. Set $k := LB$
 2. Repeat steps 3 and 4 while $k \leq UB$
 3. Apply Process to $LA[k]$
 4. Set $k := k + 1$
 5. Exit

Inserting an Element

- Inserting an element to an array is not easy all the time.
- If we want to add at the end, we have to make sure that there is available memory at the end of the array.
- If we want to add in the middle or other place than the end then we have to move the elements downwards to store the element.
- This is time consuming and so we move towards Linked list which give us an opportunity to insert elements any place without moving huge data.

Inserting an Element

A
B
C
D
E

A
B
C
D
E
P

A
B
C
D
E

A
P
B
C
D
E

Easy to add P after E

Difficult to add P after A

Inserting an Element

- This Algorithm add Item to kth position.
- INSERT(LA, N, K, Item)
 1. Set $j := N$
 2. Repeat Steps 3 and 4 while $j \geq K$
 3. Set $LA[j+1] := LA[j]$
 4. Set $j := j - 1$
 5. Set $LA[k] = \text{Item}$
 6. Set $N := N + 1$
 7. Exit

Deleting an Element

- Deleting an element is not also easy all the time.
- We can delete the last element without any effort.
- However, elements other than the last one incurs massive works.
- We have to move the elements upwards in order to fill the deleted element

Deleting an Element

A
B
C
D
E

A
B
C
D

A
B
C
D
E

A
C
D
E

Easy to Remove E

Difficult Remove B

Deleting an Element

This algorithm deletes the kth element and store it to Item.

DELETE(LA, N, K, Item)

1. Set $\text{Item} := \text{LA}[\text{K}]$
2. Repeat for $j := \text{K}$ to $\text{N}-1$:
 Set $\text{LA}[j] := \text{LA}[j+1]$
3. Set $\text{N} := \text{N}-1$
4. Exit

Sorting

- Sorting refers to the process of rearranging elements to either increasing order or to decreasing order.
- There are a numbers of such sorting algorithms.
- **Bubble sort** is one of them.
- Others sorting algorithms are: **Insertion sort, Selection sort, Merge sort, Radix sort** etc.
- This algorithm finds the largest number from the array of n elements and place it to the end in the first pass.
- Then it continue this operation for numbers $n-1$, then for $n-2$ then for $n-3$ until for 1.
- In this way it is possible to get the n sorted data.

Bubble Sort

- Bubble(data, n)
- This algorithm sorts n numbers in ascending order.
 1. Repeat steps 2 and 3 for k=1 to n-1
 2. Set ptr:=1
 3. Repeat while ptr<=n-k
 - a. If data[ptr]>data[ptr+1] then
Interchange data[ptr] and data[ptr+1]
 - b. Set ptr:=ptr+1
 4. Exit

Complexity of Bubble sort is $n(n+1)/2$

Bubble Sort (Example)

- Suppose we have five numbers 19, 13, 20, 8, 2. we can display the interchanges in the following way.
- Pass 1:
13, 19, 20, 8, 2
13, 19, 20, 8, 2
13, 19, 8, 20, 2
13, 19, 8, 2, 20
- Pass 2:
13, 19, 8, 2, 20
13, 8, 19, 2, 20
13, 8, 2, 19, 20
- Pass 3:
8, 13, 2, 19, 20
8, 2, 13, 19, 20
- Pass 4:
2, 8, 13, 19, 20

Searching


- Searching refers to finding the location of an item in a list.
- Search time depends on many factors.
 - The arrangement of the data (ascending or descending or unsorted).
 - The algorithm used to search.
 - The search criteria/ search item, etc.
- The most common Search algorithm is Linear Search.
- Other Search algorithms are: Binary Search, Binary Search Tree, etc.
- In Linear Search, data are searched from the beginning of the list until the desired data can be found or the list search ends.

Linear Search

- For a linear list of size n , the search time depends on whether the item is found at the beginning end or at ending end.
- For best case the time is 1, for worst case n . So the complexity is $(n+1)/2$

Linear(data, n , item, loc)

1. Set data[n+1]=item
2. Set loc:=1
3. Repeat while data[loc]<>item
4. Set loc:=loc+1
5. If loc==n+1 then set loc:=0
6. Exit



is not equal

Binary Search

- Binary search can only be employed to a list of sorted data.
- It divides the total list into two parts and make a decision to which part the item may reside.
- Taking the desired part it then divide again and do the same thing.
- It perform the same operation until it finds the item or the search completes.
- Since it divides the list into two all the time the complexity is very low: $(\log_2 n) + 1$

Binary Search

- Binary(data, ub, lb, item, loc)
 1. 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
 - a. set end:=mid-1
 - b. else set beg:=mid+1
 4. Set mid:=int((beg+end)/2)
 5. If data[mid]=item then
 - a. set loc:=mid
 - b. Else loc:=null
 6. exit

Binary Search Example

There are 13 data: 11, 22, 30, 33, 40, 44, 55, 60, 66, 77, 80, 88, 99. Initially beg:=1 end:=13 and mid:=7, say item=40 the steps for searching:

1. 11, 22, 30, 33, 40, 44, 55, 60, 66, 77, 80, 88, 99
2. Since $40 < 55$ so (beg=1 end=6 mid=3)
3. 11, 22, 30, 33, 40, 44, 55, 60, 66, 77, 80, 88, 99
4. Since $40 \geq 30$ so (beg=4 end=6 mid=5)
5. 11, 22, 30, 33, 40, 44, 55, 60, 66, 77, 80, 88, 99
6. Since $40 = 40$ (item=data[mid]), the search complete and the result location is 6 i.e. final mid value)

Multi Dimensional Array

- So far we talked about one dimensional array.
- Some times we need 2 or 3 dimension to represent data. For example if we want to deal with matrix we need 2D.
- 2D array: It is a collection of $m \cdot n$ elements, where each element is specified by a pair of integers (such as j, k) called the subscripts.
- The elements of a 2D array with m rows and n columns looks like as below.

$$\begin{array}{ccccccc} A[1, 1] & A[1, 2] & \dots\dots & A[1, n] \\ A[2, 1] & A[2, 2] & \dots\dots & A[2, n] \\ \dots\dots & \dots\dots & \dots\dots & \dots\dots \\ A[m, 1] & A[m, 2] & \dots\dots & A[m, n] \end{array}$$

Representation of 2D Arrays in Memory

- The elements of a 2D array is stored in memory in two way;
- row by row (first row then second row and so on) or
- Column by column (first column then second column and so on)

	1, 1	Row 1
	1, 2	
	2, 1	Row 2
	2, 2	
	3, 1	Row 3
	3, 2	
	4, 1	Row 4
	4, 2	
Row Major Order		

	1, 1	Column 1
	2, 1	
	3, 1	
	4, 1	
	1, 2	Column 2
	2, 2	
	3, 2	
	4, 2	
Column Major Order		

2D Arrays

- If the elements of an Array A with m rows and n columns are stored in row major order, the location of any elements $A[i, j]$ can be found from the formula.
 - $\text{loc}(A[i, j]) = \text{Base}(A) + w[n(i-1) + (j-1)]$
- Similarly for column major order the location is
 - $\text{loc}(A[i, j]) = \text{Base}(A) + w[m(j-1) + (i-1)]$
- Example: Consider 25X4 matrix array score; base value is 200, w=4 what is the address of score[12, 3].
 - For row major: $200 + 4[4(12-1) + (3-1)] = 384$
 - For column major: $200 + 4[25(3-1) + (12-1)] = 444$

General Multi Dimensional Array

- An n dimensional array $m_1 \times m_2 \times \dots \times m_n$ is a collection of $m_1.m_2.\dots.m_n$ data elements in which each element is specified by a list of n integers- such as k_1, k_2, \dots, k_n , called the subscript.
- As a 2D array we can also store elements as a row major order or column major order.
- For row major order the location can be found as:
- $\text{Base}(C) + w[(\dots((E_1 L_2 + E_2) L_3 + E_3) L_4 + \dots + E_{n-1}) L_n + E_n]$
- For column major order the location can be found as:
- $\text{Base}(C) + w[(((\dots(E_n L_{n-1} + E_{n-1}) L_{n-2}) + \dots + E_3) L_2 + E_2) L_1 + E_1]$

General Multi Dimensional Array

- Where L_i is the no. of elements in the index set can be computed as $L_i = UB - LB + 1$
- (For a given subscript K_i), the effective index E_i of L_i is the no. of indices preceding K_i in the index set and E_i can be computed as:
- $E_i = K_i - \text{lower bound}$.
- Example: Maze(2:8, -4:1, 6:10) is a 3D array with base=200, w=4, calculate Maze[5, -1, 8] address in a row major order and column major order.

General Multi Dimensional Array

Solution:

- $L_1=8-2+1=7$, $L_2=1-(-4)+1=6$, $L_3=10-6+1=5$,
- $E_1=5-2=3$, $E_2=-1-(-4)=3$, $E_3=8-6=2$, $w=4$, $\text{base}=200$.
- For row major order:
 - $\text{Base} + w[(E_1L_2+E_2)L_3+E_3] = 200+4[(3.6+3)5+2]=628$.
- For column major order:
 - $\text{Base} + w[(E_3L_2+E_2)L_1+E_1] = 200+4[(2.6+3)7+3]=632$.

Pointer and Pointer Array

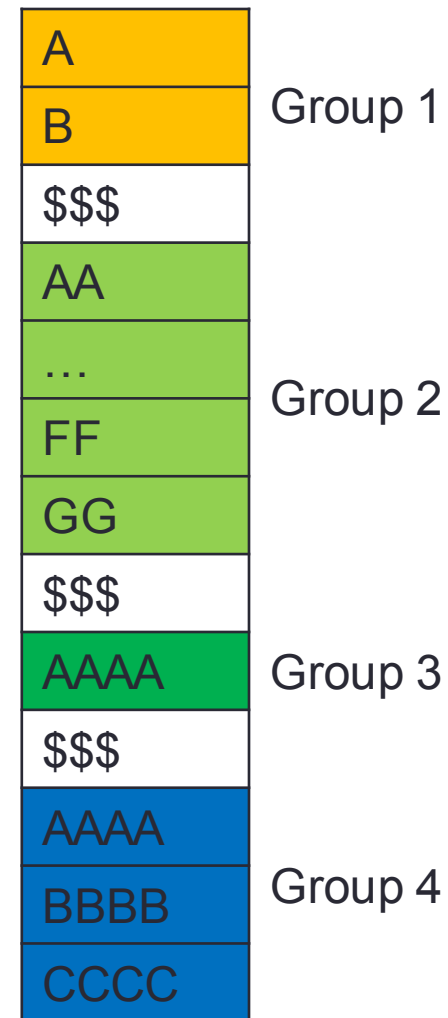
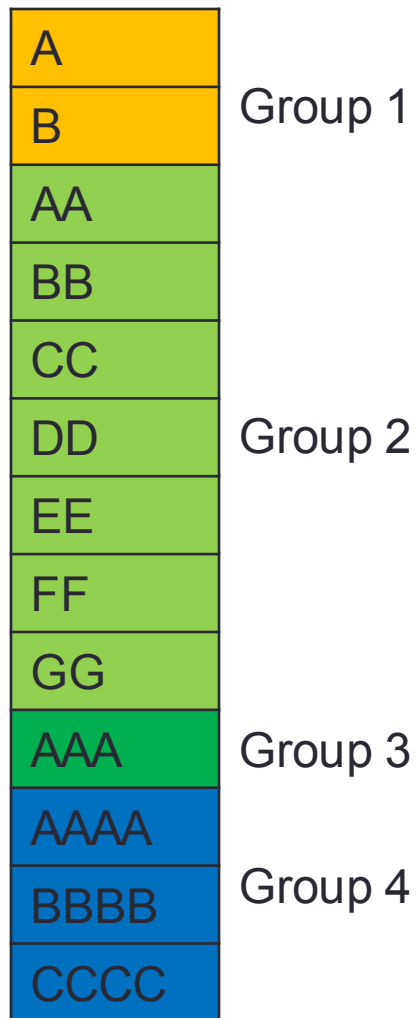
- Pointer: A variable P is called a pointer if P points to an element in Data; where Data is an array.
- Pointer Array: An array PTR is called a pointer array if each element of PTR is a pointer.
- Let us consider the list below.

Group 1	Group 2	Group 3	Group 4
A	AA	AAA	AAAA
B	BB		BBBB
	CC		CCCC
	DD		
	EE		
	FF		
	GG		

Pointer and Pointer Array

- If we store the list, we can use a 2D array of size 4X7.
- Off Course it wastes $28-13=15$ memory cell.
- The space can be properly utilized if a 1D array is employed and data are stored one group after another.
- Here the main problem is to access any specific group.
- This can be overcome if we store a sentinel (such as \$\$\$) just after each group while storing data in the 1D array.
- With the sentinel we can find individual group; there off course still exist a problem; we have to traverse from the beginning.
- The best solution is to use a pointer array.

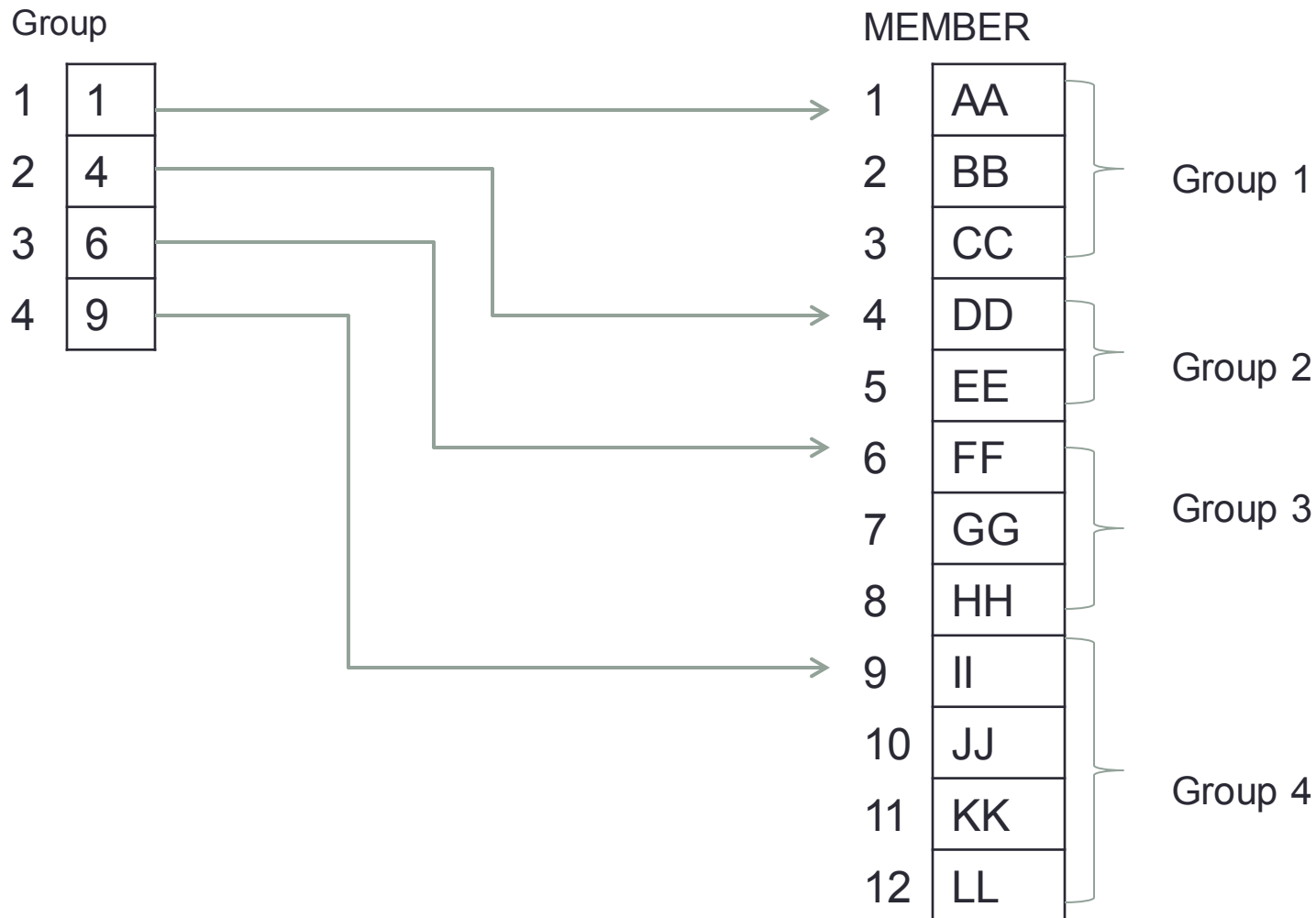
Pointer and Pointer Array



Pointer Array

- A pointer array is used to point each group of elements.
- The pointer array size depends on the no. of groups.
- Say Group is a pointer array and Member is a 1D array then Group[L] points to the first element of group L, Group[L+1] points to the first element of group L+1.
- Group[L] and Group[L+1]-1 contains respectively the first and last element of group L.
- Algorithm to display each elements of group L.
 1. Set first:=Group[L] and last:=Group[L+1]-1
 2. Repeat for k=first to last
 3. Write Member[k]
 4. Return

Pointer Array Example

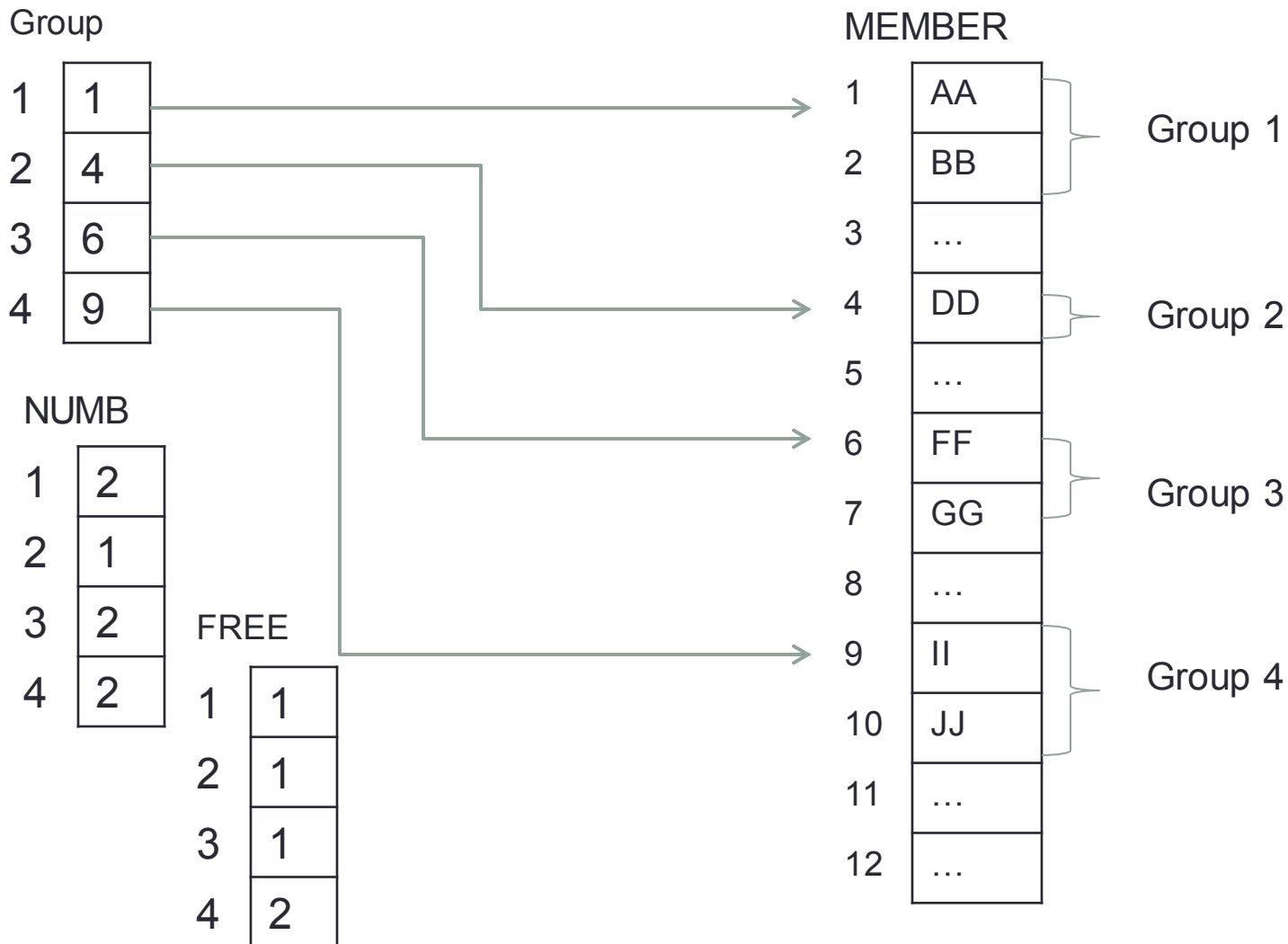


Pointer Array Other Version

- Actually in previous example of pointer array data need to be moved downward or upward if anyone wish to insert or delete any element respectively.
- A slightly different version however provide us some empty cells after each group which reduces the no. of data movement.
- In this method, an array NUMB is used which gives the no. of elements in each group.
- There may be another array FREE which gives the no. of free cells between two groups. Off course we can calculate the free space from the following formula.

$$\text{FREE}[k] = \text{GROUP}[k+1] - \text{GROUP}[k] - \text{NUMB}[k]$$

Another Pointer Array Example



Records

- A record is a collection of related data items each of which is called a field or attribute.
- Each data item itself may be a group item composed of sub items; those items which are indecomposable are called items or atoms or scalars.
- Although a record is a collection of related data items, it differ from a linear array in the following ways:
 1. A record may be a collection of nonhomogeneous data (data items may have different data types).
 2. The data items in a record are indexed by attribute names, so there may not be a natural ordering of its elements.

Sparse Matrices

- Matrices with a high proportion of zero entries are called Sparse Matrices.
- It is two types:
- (Lower) Triangular Matrix: All elements above the main diagonal are zero or, equivalently, nonzero entries can only occur on or below the main diagonal.
- Tridiagonal Matrix: Non zero entries can only occur on the diagonal or on elements immediately above or below the diagonal
- Since in case of Sparse matrix nonzero elements are high we can save memory space by discarding the zero elements in some way (discussed later)

Sparse Matrix Example

$$\begin{pmatrix} 4 & & & & \\ -2 & 3 & & & \\ 2 & 4 & -5 & & \\ 9 & 8 & -2 & -5 & \\ 2 & 4 & 5 & 8 & 9 \end{pmatrix}$$

Triangular Matrix

$$\begin{pmatrix} 4 & -3 & & & \\ -2 & 3 & 6 & & \\ & 4 & -5 & 7 & \\ & & -2 & -5 & -5 \\ & & & 8 & 9 \end{pmatrix}$$

Tridiagonal Matrix

Memory Save of a Sparse Matrix

- We can save almost half the memory requirement of a n square Sparse matrix A if we attempt to save only the nonzero elements as indicated by the figure.
- For this reason, we can use an one dimensional array B . That is, we let $B[1]=a_{1,1}$, $B[2]=a_{2,1}$, $B[3]=a_{2,2}$, $B[4]=a_{3,1}$ and so on.
- In this way B will contain $1+2+3+4+\dots+n=(1/2)n(n+1)$ element rather than n^2 .
- We need a formula that gives us the integer L in terms of j and k where $B[L]=a_{j,k}$.
- L is the no. of elements in the list up to and including $a_{j,k}$.

Relation between L and j,k

- There are $1+2+3+4+\dots+(j-1)=(1/2)j(j-1)$ elements in the rows above $a_{j,k}$, and there are k elements in row j up to and including $a_{j,k}$.
- Accordingly $L=(1/2)j(j-1)+k$ yields the index that accesses the value $a_{j,k}$ from the linear array B.

