CSE-213
(Data Structure)

Lecture on

Recursion, Pointers and Records

Md. Jalal Uddin

Lecturer

Department of CSE

City University

Email: jalalruice@gmail.com

No: 01717011128 (Emergency Call)

Recursion

Recursion: Basic idea

- We have a bigger problem whose solution is difficult to find
- We divide/decompose the problem into smaller (sub) problems
  - Keep on decomposing until we reach to the smallest sub-problem (base case) for which a solution is known or easy to find
  - Then go back in reverse order and build upon the solutions of the sub-problems
- Recursion is applied when the solution of a problem depends on the solutions to smaller instances of the same problem

Recursion

- Recursion in data structure is when a function calls itself indirectly or directly, and the function calling itself is known as a recursive function.
- It's generally used when the answer to a larger issue could be depicted in terms of smaller problems.

Recursive Function

- A function which calls itself

```
int factorial ( int n ) {
    if ( n == 0)        // base case
        return 1;
    else                // general/ recursive case
        return n * factorial ( n - 1 );
}
```

Recursion: Function

- A function is said to be *Recursively defined* if the function definition refers to itself.
- A *Recursive Function* must have the following two properties:
  - There must be certain arguments, called **BASE VALUE**, for which the function does not refer to itself.
  - Each time the function does refer to itself, the argument of the function must be closer to a **BASE VALUE**.
- A *Recursive Function* with two properties is also said to be *well-defined*.

Recursion: Factorial Function

- ✓ In some problems, it may be natural to define the problem in terms of the problem itself.
- ✓ Recursion is useful for problems that can be represented by a **SIMPLER VERSION** of the same problem.
- ✓ Example: the factorial function

```
6! = 6 * 5 * 4 * 3 * 2 * 1
```

We could write:

```
6! = 6 * 5!
```

Recursion: Factorial Function

- ✓ In general, we can express the factorial function as follows:

  `n! = n * (n-1)!`

  Is this correct? Well… almost.

- ✓ The factorial function is **ONLY DEFINED** for *positive* integers. So we should be a bit more precise:

`i)  n! = 1              (if n is equal to 1)`

`ii) n! = n * (n-1)! (if n is larger than 1)`

- ✓ Observe that, this definition of **`n!`** is recursive, since it refers to itself when it uses **`(n-1)!`** , However,
- ✓ **`i)`** the value of **`n!`** is explicitly given when **n=0** (**BASE VALUE**)
- ✓ **`ii)`** the value of **`n!`** for arbitrary **`n`** is defined in terms of a smaller value of n which is closer to the **BASE VALUE**

Recursion: Factorial Function

EXAMPLE: Let's calculate 3! Using the recursive definition.

(1) 3! = 3 . 2!

(2) 2! = 2 . 1!

(3) 1! = 1 . 0!

(4) 0! = 1 **(BASE VALUE)**

(5) 1! = 1 . 1 = 1

(6) 2! = 2 . 1 = 2

(7) 3! = 3 . 2 = 6

- ✓ **Observe that we back track in the reverse order of the original postponded evaluations.**
- ✓ **Recall that this type of postponed processing tends itself to the use of STACKS.**

Recursion: Function

Assume the number typed is 3, that is, numb=3.

```
fac(3) :
3 <= 1 ?                    No.
fac(3) = 3 * fac(2)
   fac(2) :
      2 <= 1 ?              No.
      fac(2) = 2 * fac(1)
          fac(1) :
             1 <= 1 ?       Yes.
             return 1
       fac(2) = 2 * 1 = 2
       return fac(2)
 fac(3) = 3 * 2 = 6
 return fac(3)
```

```
i)   n! = 1              (if n is equal to 1)
ii)  n! = n * (n-1)!     (if n is larger than 1)
```

```
int fac(int numb){
   if(numb<=1)
      return 1;
   else
      return numb * fac(numb-1);
}
```

```
fac(3)  has the value 6
```

Recursion: Function

For certain problems (such as the factorial function), a recursive solution often leads to short and elegant code. Compare the recursive solution with the iterative solution:

Recursive solution

```
int fac(int numb){
   if(numb<=1)
     return 1;
   else
     return numb*fac(numb-1);
}
```

Iterative solution

```
int fac(int numb){
int product=1;
    while(numb>1){
       product *= numb;
           numb--;
                }
       return product;
                 }
```

Iteration Vs Recursion

Two approaches to writing repetitive algorithms:

1. Iteration
2. Recursion

Recursion vs. Iteration: Computing N!

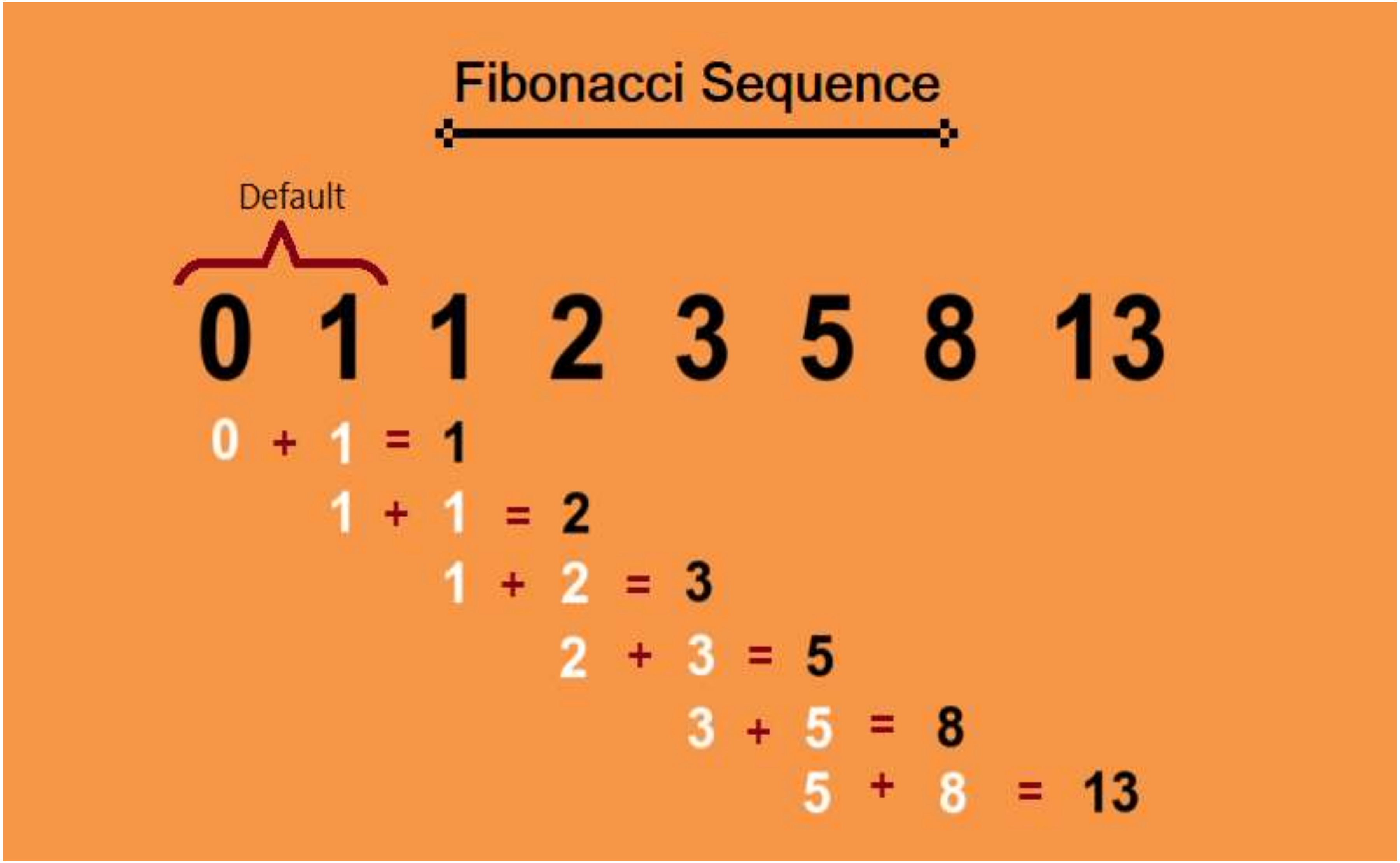
- The factorial of a positive integer n, denoted $n!$, is defined as the product of the integers from 1 to n. For example, $4! = 4\cdot3\cdot2\cdot1 = 24$.
  - Iterative Solution

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot (n-1) \cdot (n-2) \cdots 3 \cdot 2 \cdot 1 & \text{if } n \geq 1 \end{cases}$$

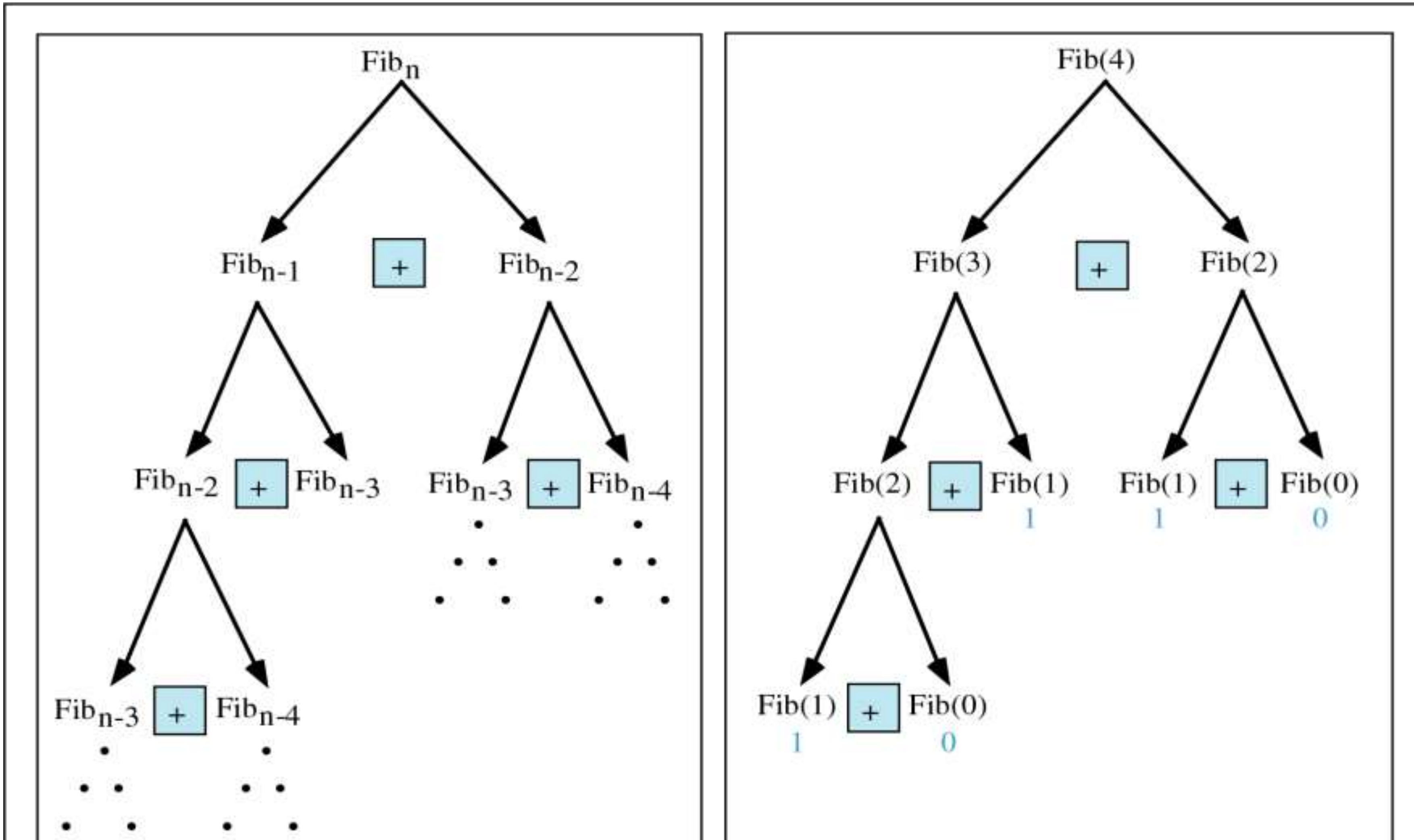
  - Recursive Solution

$$\text{factorial}(n) = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot \text{factorial}(n-1) & \text{if } n \geq 1 \end{cases}$$

Recursion: Fibbonacc



Recursion: Fibbonacci



(a) Fib(n)　　(b) Fib(4)

Recursion: Fibbonacc

```
// Calculate Fibonacci numbers using recursive function.
// A very inefficient way, but illustrates recursion well
int fib(int number)
{
        if (number == 0) return 0;
        if (number == 1) return 1;
        return (fib(number-1) + fib(number-2));
}
```

Recursive definition:

```
F(0) = 0;
F(1) = 1;
F(number)=F(number-1)+ F(number-2);
```

Pointers

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

Pointer Arrays

An array **PTR** is called a *pointer array* if each element of **PTR** is a pointer

Pointer and Pointer array are used to facilitate the processing the information in DATA

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		

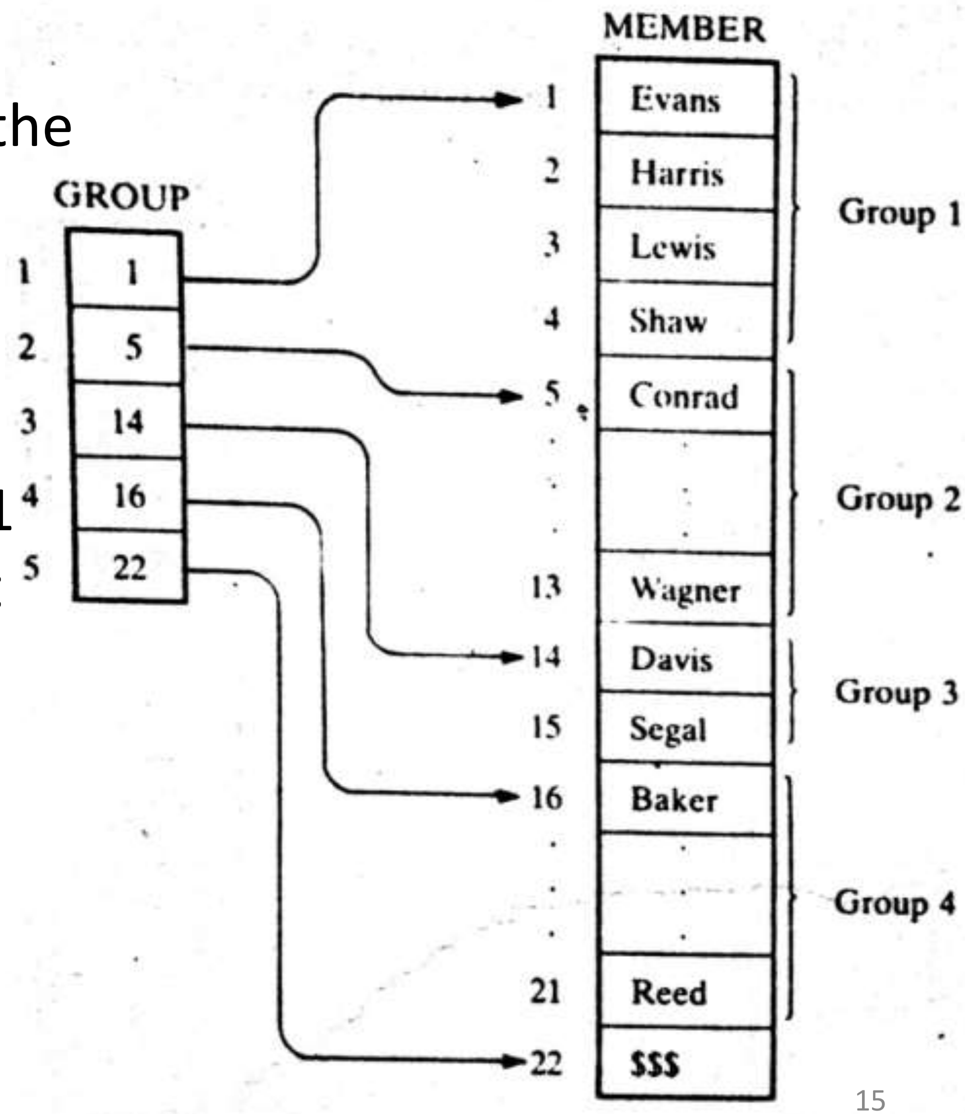
How the membership list can be stored in memory keeping track of the different groups?

POINTER ARRAYS

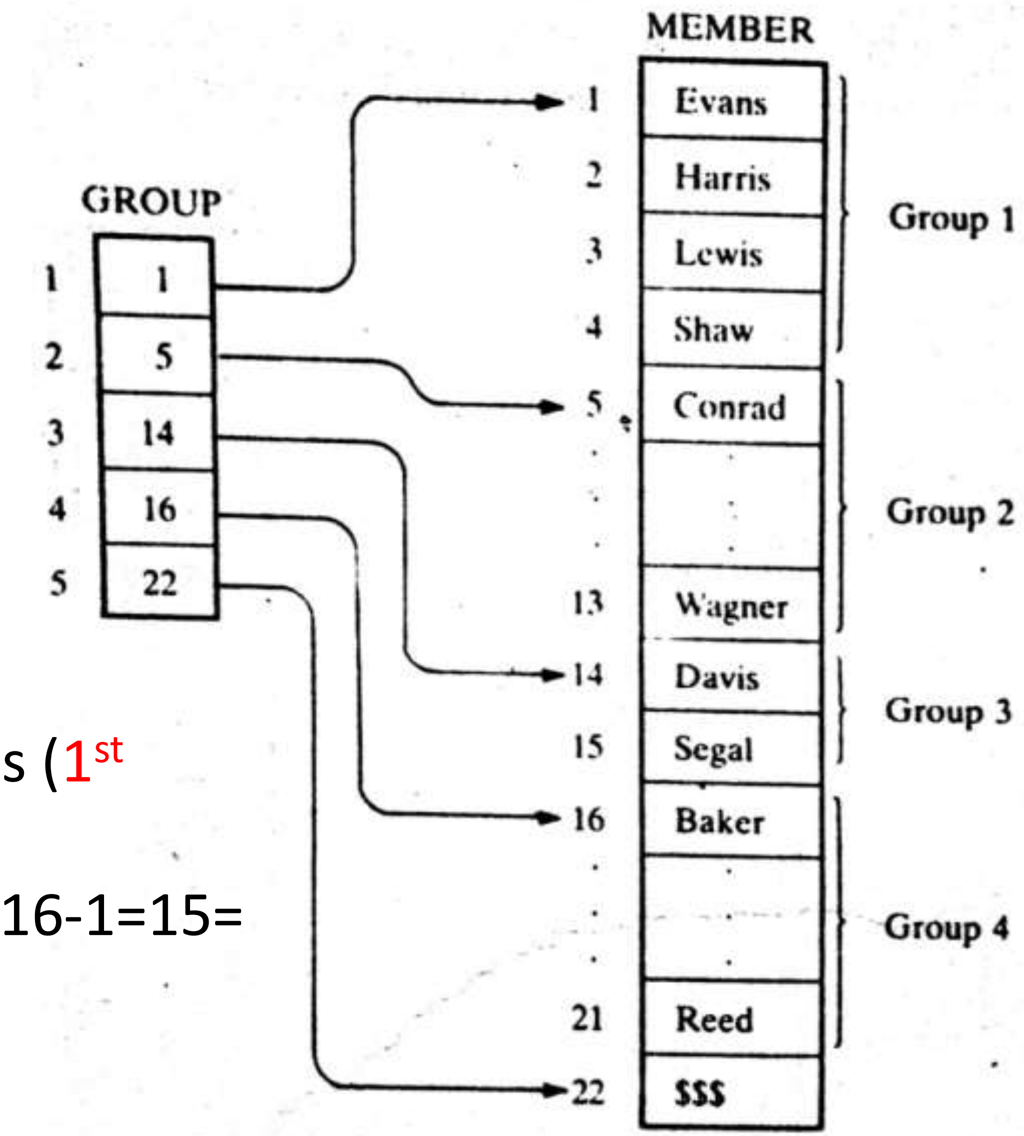
Pointer arrays is introduced in the last two space-efficient data structure.

The pointer array contains the locations of the.....

- ✓ Different groups, or
- ✓ First element in the different groups.
- ✓ GROUP[L] and GROUP[L+1]-1 contain respectively, the first and last element in group L.



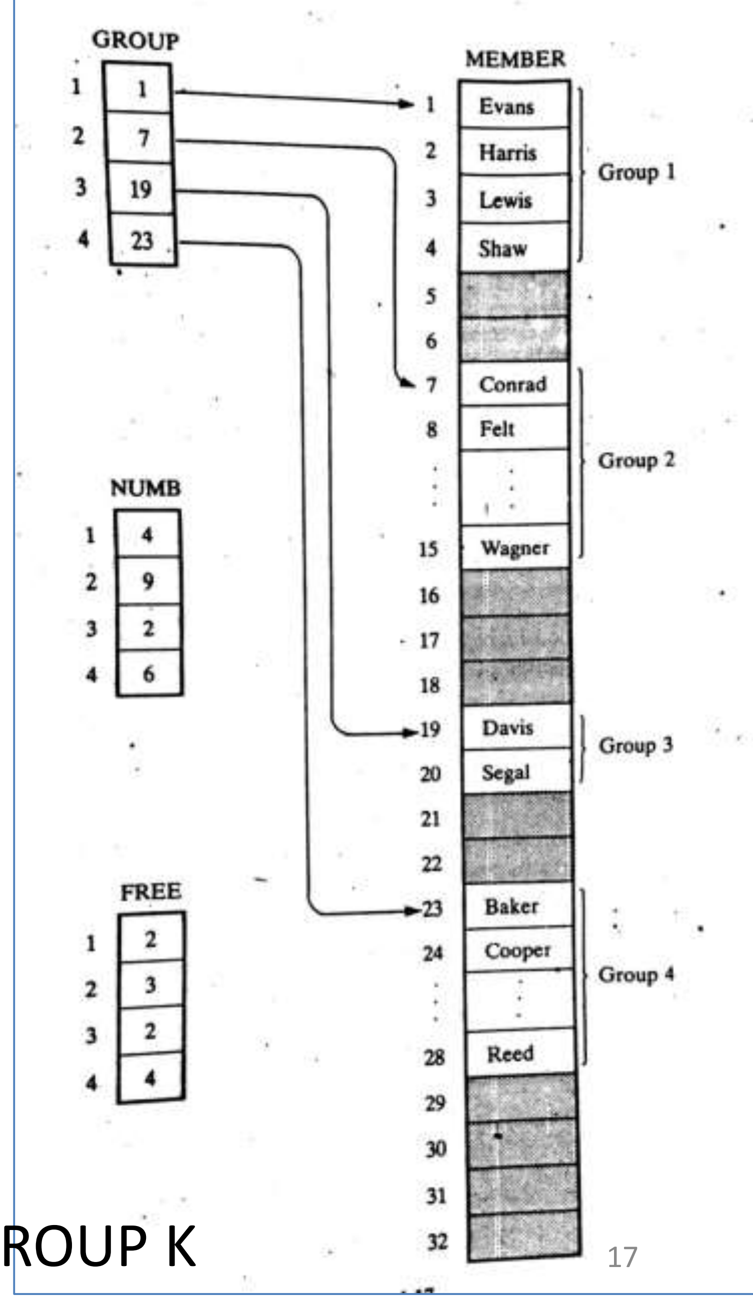
POINTER ARRAYS: Example



Suppose L=3
GROUP[L]=GROUP[3]= 14= Davis (1st Element of grp 3)
GROUP[L+1]-1=GROUP[3+1]-1=16-1=15= Segel (last element of grp 3)

POINTER ARRAYS: Extended

- Here unused memory cells are indicated by the shading.
- Observe that now there are some empty cells between the groups.
- Accordingly, a new element may be inserted in a new group without necessarily moving the elements in any other group.
- Using the data structure, one requires an array NUMB which gives the number of elements in each group.
- Observe that GROUP[K+1]-GROUP[K] is the total number of space available for group K. Hence



FREE[K]=GROUP[K+1]-GROUP[K]-NUMB[K]

Gives the number of empty cells following GROUP K

POINTER ARRAYS: Extended, Example

Suppose, we want to print only the number of FREE cells of GROUP 2. Then

FREE[K]=GROUP[K+1]-GROUP[K]-NUMB[K]

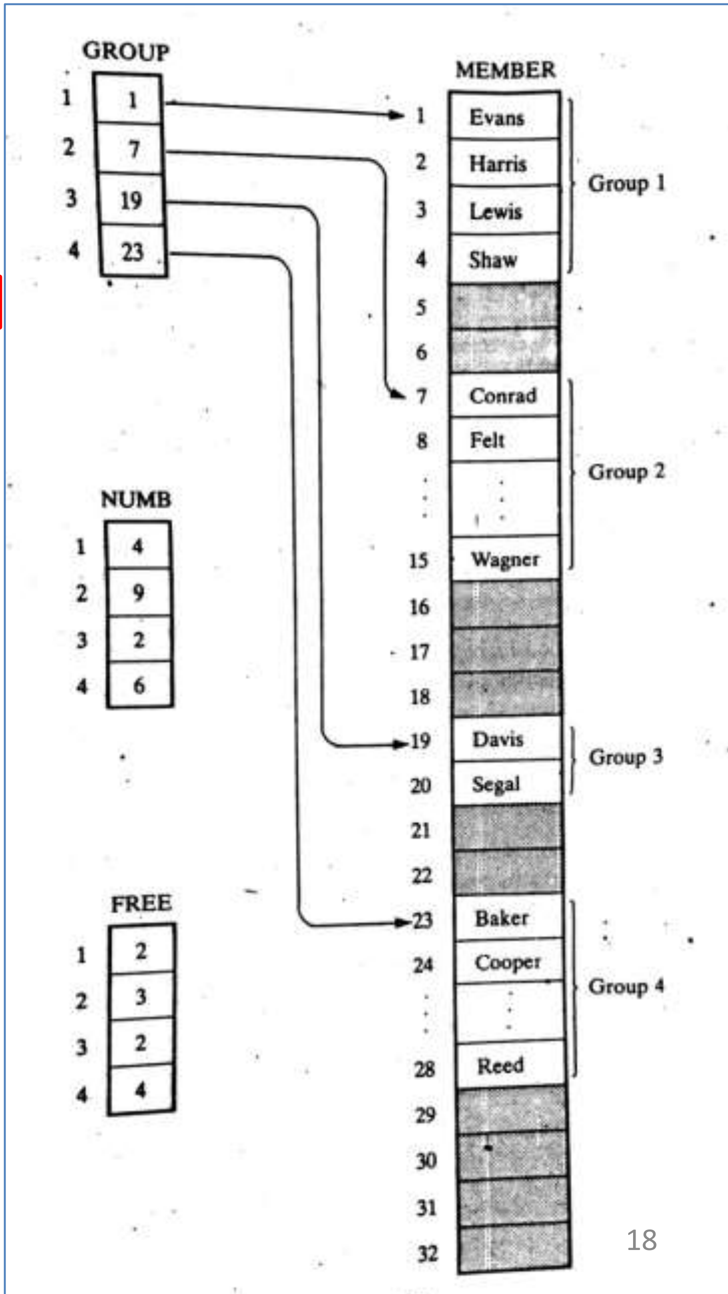
FREE[2]=GROUP[2+1]-GROUP[2]-NUMB[2]

= 19-7-9

=3

For GROUP 4?

Try now



RECORDS

- ✓ A ***record*** is a collection of related data items, each of which is called a field or attribute, and
- ✓ a ***file*** is a collection or similar records.
- ✓ Although, a record is a collection of data items, it differs from a linear array in the following ways……………
  - ➢ A record may be a collection of nonhomogeneous data;
  - ➢ The data items in a record are indexed by attribute names, so there may not be a natural ordering of its elements.

RECORDS: Structure Example

1. Newborn
   2. Name
   2. Sex
   2. Birthday
      3. Month
      3. Day
      3. Year
   2. Father
      3. Name
      3. Age
   2. Mother
      3. Name
      3. Age

Under the relationship of group item to sub- item, the data items in a record form a hierarchical structure which can be described by mean of “Level” numbers

Name	Sex	Birthday			Father		Mother	
					Name	Age	Name	Age

Indexing Items in a Record

- ✓ Suppose we want to access some data item in a record.
- ✓ We can not simply write the data name of the item since the same may appear in different places in the record. For example.....

1. Newborn
    2. Name
    2. Sex
    2. Birthday
        3. Month
        3. Day
        3. Year
    2. Father
        3. Name
        3. Age
    2. Mother
        3. Name
        3. Age

➢ In order to specify a particular item,

- ❖ we may have to *qualify* the name by using appropriate group item names in the structure.
- ❖ This *qualification* is indicated by using decimal points (periods) to separate group items from subitems.
- ❖ Example: Newborn.Father.Age or Father.Age

Fully qualified reference

Indexing Items in a Record

1. Newborn

- **2. Name**
- **2. Sex**
- **2. Birthday**
  - **3. Month**
  - **3. Day**
  - **3. Year**
- **2. Father**
  - **3. Name**
  - **3. Age**
- **2. Mother**
  - **3. Name**
  - **3. Age**

1. Newborn(20)

- **2. Name**
- **2. Sex**
- **2. Birthday**
  - **3. Month**
  - **3. Day**
  - **3. Year**
- **2. Father**
  - **3. Name**
  - **3. Age**
- **2. Mother**
  - **3. Name**
  - **3. Age**

Newborn is defined to be a file with 20 records

✓ The Name of the sixth newborn to be referenced by writing …….

Newborn. Name[6]

✓ The age of the father of the 6th newborn may be referenced by writing…..

Newborn.Father.Age[6]

Representation of RECORDS in memory

Since records may contain nonhomogeneous data, the element of a record can not be stored in an array.

See Example: 4.18, 4.20, 4.21

Teach Yourself with example

Try to understand the SOLVED Problems