Lecture 07-08

- Arrays: The Data Structures

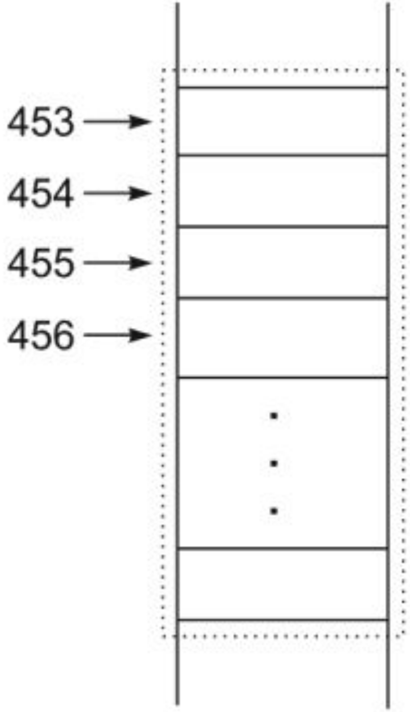
IT205: Data Structures (AY 2023/24 Sem II Sec B) — **Dr. Arpit Rana**

Array: Definition

An array is

- *finite* – contains only a limited number of elements,
- *indexed* – all the elements are stored one-by-one in contiguous locations of the computer memory in a linear ordered fashion, and
- *homogeneous* – all the elements are of the same data type

collection (that's why a composite data structure) of data elements.



Array: Terminology

- *Size (a.k.a. length or dimension)* – the number of elements,
- *Type* – the data type of the elements that it stores
- *Base* – the address of the memory location of the first element.
- *Index* – an integer value used to refer to an element of the array
- *Range of Indices* – indexes of array elements may change from a lower bound (L) to an upper bound (U).

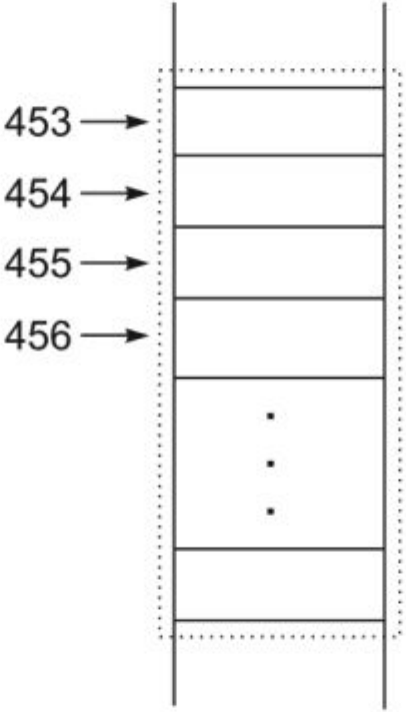
  index start with zero and first element is A1

  - Index (A_i) = L + i - 1, *Size (A)* = U - L +1

- Word – denotes the size of an element; if the size of an element is doubled, it needs two consecutive memory locations to store one word.

One-Dimensional Array

If only ***one index*** is required to reference all the elements in an array, such an array is termed as one-dimensional array or simply an array.

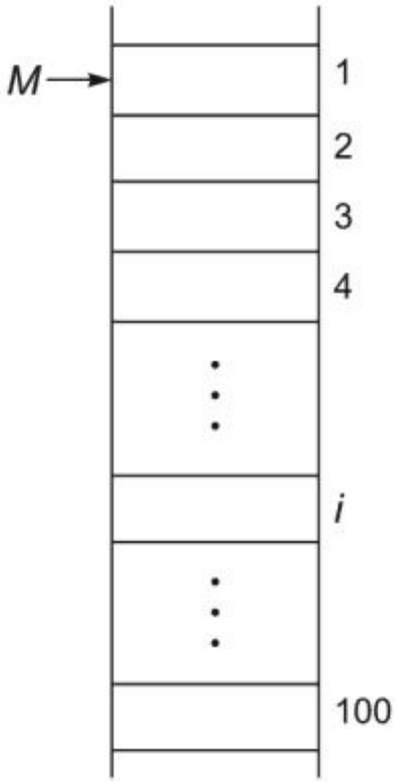


One-Dimensional Array: Physical Representation

Let's suppose an array A[100] is to be stored in a memory and the base is M.

- If each element requires one word, the location for an element A[i] in the array can be calculated as -

Address (A[i]) = M + (i - 1)

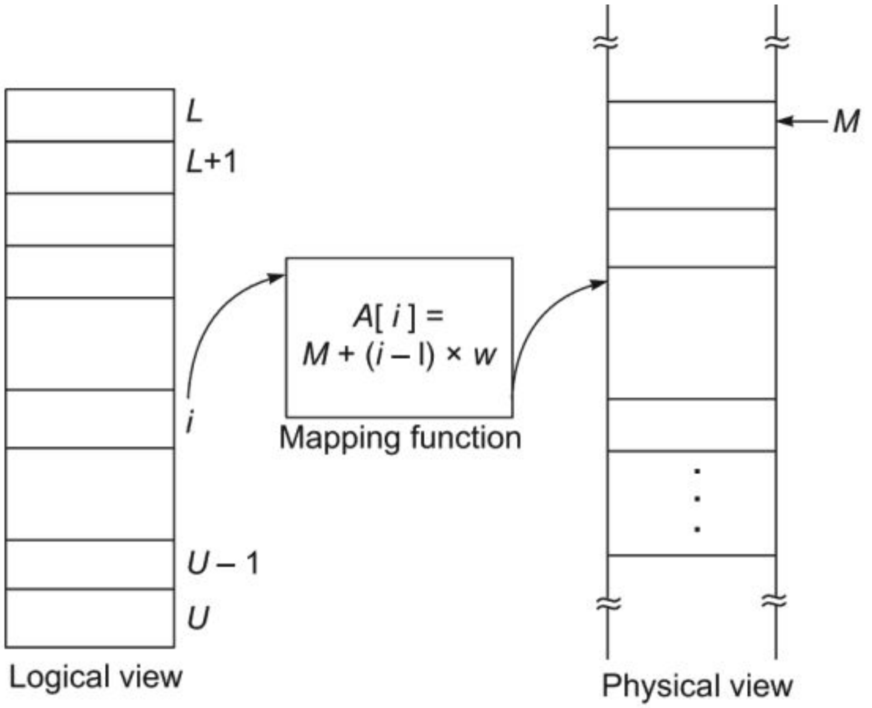


One-Dimensional Array: Physical Representation

- Similarly, an array can be written as A[L … U]. If each element takes w words in memory, then the address A[i] will be -

  Address (A[i]) = M + (i - L) * w

This is an indexing formula which is used to map the logical representation of an array to its physical one.



Exercise

Suppose, an array A[–15 ... 64] is stored in a memory whose starting address is 459. Assume that the word size for each element is 2. Then obtain the following:

(a) How many number of elements are there in the array A? 15+1+64=80

(b) If one word of the memory is equal to 2 bytes, then how much memory is required to store the entire array? 360 bytes

(c) What is the location for A[50]? A[50]=459+(50+15)*2=459+130=589

(d) What is the location of the 10th element? A[-6]=459+(-6+15)*2=478

(e) Which element is located at 589? A[50]

In the indexing formula (in the last slide), we have taken that the array is stored from the lower region of the memory to the higher region. But in some computers the convention is just reverse, that is, starting an array i at a higher region and hence Address (A_i) > Address (A_{i+1}) for all $L < i < U$. Modify the indexing formula for this case.

A[i]=M+(U-i)*W;

Algorithmic Notations

We will use these notations for writing our algorithms.

Algorithm <Name of the operation>
Input: <Specification of input data for the operation>
Output: <Specification of output after the successful performance of the operation>
Remark: <If the operation assumes other data structure for its implementation or something important>

Steps:

```
. . . . . . . . . . . . . . . . . .
If <condition> then                         // Comment on this step, if any is applicable
        . . . . . . . . . . . . . . . .
        . . . . . . . . . . . . . . . .
Else
        . . . . . . . . . . . . . . . . . . .
        . . . . . . . . . . . . . . . . . .
EndIf
. . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . .
While <condition> do
        . . . . . . . . . . . . . . . . . .
        . . . . . . . . . . . . . . . . . .
EndWhile
. . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . .
/* Comment on the following few steps, if any is applicable */
For <loop condition> do
        . . . . . . . . . . . . . . . . . .
        . . . . . . . . . . . . . . . . . .
EndFor
. . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . .
Stop
```

Operations on Arrays

Traversing: to visit all elements in an array. In the algorithm below, the Process() method defines an action on an element of the array.

Algorithm TraverseArray
Input: An array A with elements.
Output: According to *Process*( ).
Data structures: Array A[L ... U]. // L and U are the lower and upper bounds // of array index

Steps:

1. $i = L$ // Start from the first location L
2. **While** $i \leq U$ **do**
3. **Process**(A[i])
4. $i = i + 1$ // Move to the next location
5. **EndWhile**
6. **Stop**

Operations on Arrays

in decreasing order

Sorting: to sort all elements of an array in a specific order (ascending or descending).

in increasing order

Algorithm SortArray

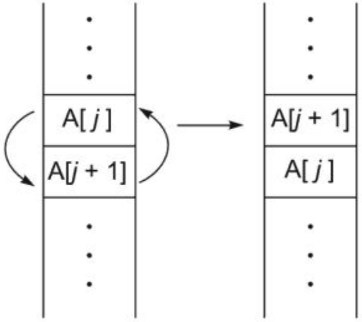
Input: An array with integer data.

Output: An array with sorted elements in an order according to *Order*( ).

Data structures: An integer array A[*L*...*U*]. // *L* and *U* are the lower and upper bounds of // array index

Steps:

```
i = U
While i ≥ L do
    j = L                                          // Start comparing from first
    While j < i do
        If Order(A[j], A[j + 1]) = FALSE           // If A[j] and A[j + 1] are not in order
            Swap(A[j], A[j + 1])                   // Interchange the elements (see Figure 2.4)
        EndIf
        j = j + 1
    EndWhile
    i = i − 1
EndWhile
Stop
```



Swapping operation

Exercise

Write algorithms to

(a) Sort an array of integers in ascending order. helloc2.cpp
(b) Sort an array of integers in descending order. helloc3.cpp
(c) Sort an array of string of characters in lexicographic order. helloc4.cpp
(d) Sort an array of an abstract data type. complex number sorting .

Hint: You have to think of only a small modification to procedure *Order*(...) in each case.

Operations on Arrays

Searching: to search an element of interest in an array.

Algorithm SearchArray

Input: *KEY* is the element to be searched.

Output: Index of *KEY* in A or a message on failure.

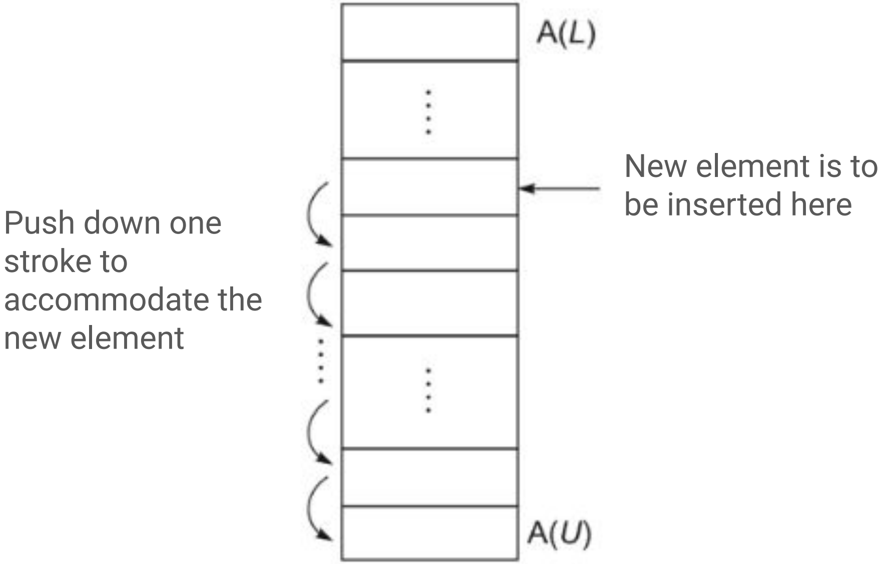
Data structures: An array A[*L* ... *U*]. //*L* and *U* are the lower and upper bounds of array index

Steps:

```
1.  i = L, found = 0, location = 0          // found = 0 indicates search is not finished and
                                               unsuccessful
2.  While (i ≤ U) and (found = 0) do        // Continue if all or any one condition do(es) not satisfy
3.      If Compare(A[i], KEY) = TRUE then   // If key is found
4.          found = 1                       // Search is finished and successful
5.          location = i
6.      Else
7.          i = i + 1                       // Move to the next
8.      EndIf
9.  EndWhile
10. If found = 0 then
11.         Print "Search is unsuccessful : KEY is not in the array"
12. Else
13.         Print "Search is successful : KEY is in the array at location", location
14. EndIf
15. Return(location)
16. Stop
```

Operations on Arrays

Insertion: to insert an element into an array provided that the array is not full.



Operations on Arrays

Algorithm InsertArray

Input: *KEY* is the item, *LOCATION* is the index of the element where it is to be inserted.
Output: Array enriched with *KEY*.
Data structures: An array A[*L* ... *U*]. // *L* and *U* are the lower and upper bounds of

Steps:

```
1.  If A[U] ≠ NULL then          // NULL indicates that room is available for a new entrant
2.      Print "Array is full: No insertion possible"
3.      Exit                                                        // End of execution
4.  Else
5.      i = U                                                   // Start pushing from bottom
6.      While i > LOCATION do
7.          A[i] = A[i–1]
8.          i = i – 1
9.      EndWhile
10.     A[LOCATION] = KEY                           // Put the element at the desired location
11. EndIf
12. Stop
```

Exercise

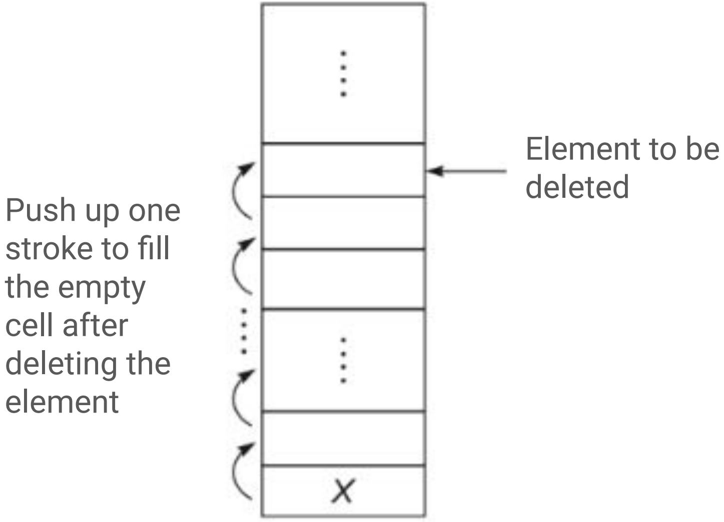
The algorithm InsertArray only checks the last element for vacancy. But an array may be empty from any i^{th} position ($L \le i \le U$); in that case the numbers of push down can be reduced instead of pushing down the entire trailing part. Modify the algorithm InsertArray when the last element is at the i^{th} location ($i \le U$). ex1.cpp

How can an empty array be defined? Verify which of the aforementioned algorithms work well with an empty array. If not, modify the algorithm(s) so that they can work even for empty array(s).

Operations on Arrays

Deletion: to delete a particular element from an array.

It is a general practise to keep no intermediary location as empty.



Operations on Arrays

Algorithm DeleteArray
Input: *KEY* the element to be deleted.
Output: Slimed array without *KEY*.
Data structures: An array A[*L*...*U*]. // *L* and *U* are the lower and upper bounds of // array index

Steps:

```
1.  i = SearchArray(A, KEY)          // Perform the search operation on A and return
2.  If (i = 0) then                  // the location
3.      Print "KEY is not found: No deletion"
4.      Exit                         // Exit the program
5.  Else
6.      While i < U do
7.          A[i] = A[i + 1]          // Replace the element by its successor
8.          i = i + 1
9.      EndWhile
10. EndIf
11. A[U] = NULL                      // The bottom-most element is made empty (see Figure 2.6)
12. U = U – 1                        // Update the upper bound now
13. Stop
```

ex3.cpp

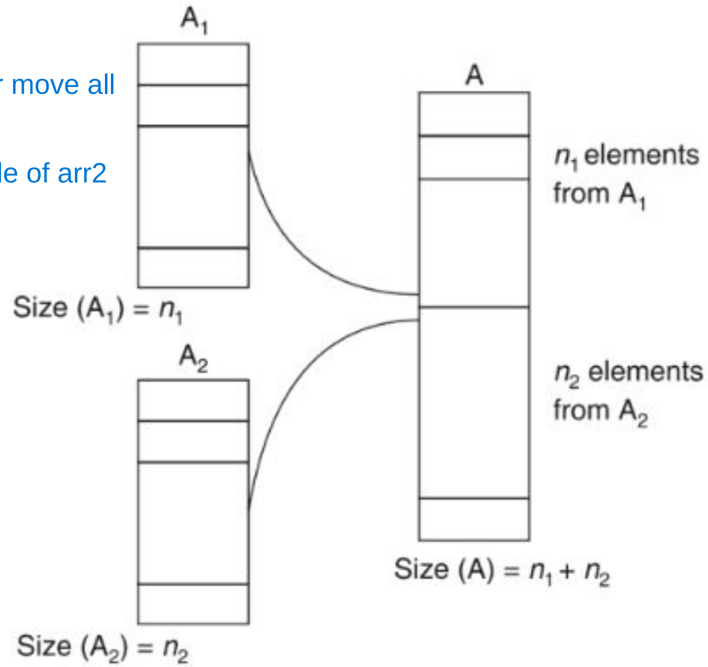
Operations on Arrays

Merging: to merge to different arrays into a single array.

1.first move all element of arr1 and after move all element of arr2 in arr3

2.move first ele of arr1 and then first ele of arr2 and repeat this process

first method is good because in first method we need only 2 array at time in second method we need 3 array at time



Operations on Arrays

Algorithm Merge

Input: Two arrays $A_1[L_1 \ldots U_1]$, $A_2[L_2 \ldots U_2]$.

Output: Resultant array $A[L \ldots U]$, where $L = L_1$, and $U = U_1 + (U_2 - L_2 + 1)$ when A_2 is appended after A_1.

Data structures: Array structure.

Steps:

1. $i_1 = L_1,\ i_2 = L_2,$ // *Initialization* of control variables
2. $L = L_1,\ U = U_1 + U_2 - L_2 + 1$ // *Initialization* of lower and upper bounds // L and U are the two bounds of resultant array A
3. $i = L$
4. **AllocateMemory** (**Size**$(U - L + 1)$) // Allocate memory for the array A
5. **While** $i_1 \le U_1$ **do** // To copy array A_1 into the first part of A
6. $\quad A[i] = A_1[i_1]$
7. $\quad i = i + 1,\ i_1 = i_1 + 1$
8. **EndWhile**
9. **While** $i_2 \le U_2$ **do** // To copy array A_2 into the last part of A
10. $\quad A[i] = A_2[i_2]$
11. $\quad i = i + 1,\ i_2 = i_2 + 1$
12. **EndWhile**
13. **Stop**

ex4.cpp

Exercise

Modify the algorithm *Merge* for the following cases of (i) if both A_1 and A_2 are empty; (ii) either A_1 is empty or A_2 is empty.

ex5.cpp

Calculate the *time complexity* of the following operations:

1. Traversing 0(n)
2. Sorting 0(n^2)
3. Searching 0(n)
4. Insertion 0(n)
5. Deletion 0(n)
6. Merging 0(n+m)

n=Number of element in array1

m=Number of element in array2

Next Lecture

- Multidimensional Arrays