Data Structures & Algorithms

Prepared by: Mohamed Ayman

Algorithm Engineer at Valeo

Deep Learning Researcher and Teaching Assistant

at The American University in Cairo (AUC)

spring 2020

sw.eng.MohamedAyman@gmail.com

facebook.com/cs.MohamedAyman

linkedin.com/in/cs-MohamedAyman

github.com/cs-MohamedAyman

codeforces.com/profile/Mohamed_Ayman

Lecture 2

Arrays
Dynamic Length

Course Roadmap

Part 1: Linear Data Structures

Lecture Agenda

We will discuss in this lecture the following topics

1- Introduction to Arrays

2- Insertion Operation

3- Deletion Operation

4- Search Operation

5- Traverse Operation

6- Time Complexity & Space Complexity

Let's
STARTUP

Lecture Agenda

Section 1: Introduction to Arrays

Section 2: Insertion Operation

Section 3: Deletion Operation

Section 4: Search Operation

Section 5: Traverse Operation

Section 6: Time Complexity & Space Complexity

Introduction to Arrays

➢ An array data structure, is a data structure consisting of a collection of elements, each identified by at least one array index or key. An array is stored such that the position of each element can be computed from its index tuple by a mathematical formula. The simplest type of data structure is a linear array, also called one-dimensional array. The memory address of the first element of an array is called first address, foundation address, or base address.

➢ Arrays are among the oldest and most important data structures, and are used by almost every program. They are also used to implement many other data structures, such as lists and strings. They effectively exploit the addressing logic of computers. In most modern computers and many external storage devices, the memory is a one-dimensional array of words, whose indices are their addresses. Processors, especially vector processors, are often optimized for array operations.

➢ Arrays are useful mostly because the element indices can be computed at run time. Among other things, this feature allows a single iterative statement to process arbitrarily many elements of an array. For that reason, the elements of an array data structure are required to have the same size and should use the same data representation. The term array is often used to mean array data type, a kind of data type provided by most high-level programming languages that consists of a collection of values or variables that can be selected by one or more indices computed at run-time.

Introduction to Arrays

➢ **Array is a container which can hold a fix number of items** and these items should be of the same type. Most of the data structures make use of arrays to implement their algorithms. Following are the important terms to understand the concept of Array.

• Element: Each item stored in an array is called an element.

• Index: Each location of an element in an array has a numerical index, which is used to identify the element.

➢ **Advantages of Arrays**

• Reading an array element is simple and efficient. This is because any element can be instantly read using indexes (base address calculation behind the scene) without traversing the whole array.

• All the elements of an array can be accessed using a single name (array name) along with the index, which is readable, user-friendly and efficient rather than storing those elements in different-2 variables.

• Array is a foundation of other data structures. Like Linked List, Stack, Queue etc. are implemented using array.

➢ **Disadvantages of Arrays**

• While using array, we must need to make the decision of the size of the array in the beginning, so if we are not aware how many elements we are going to store in array, it would make the task difficult.

• The size of the array is fixed so if at later point, if we need to store more elements in it then it can't be done. On the other hand, if we store less number of elements than the declared size, the remaining allocated memory is wasted.
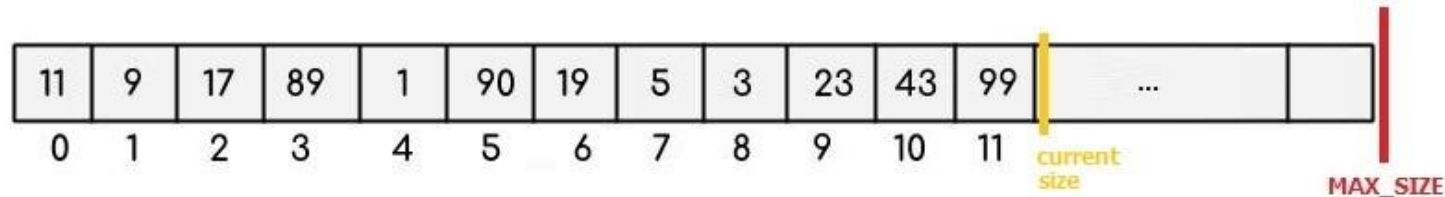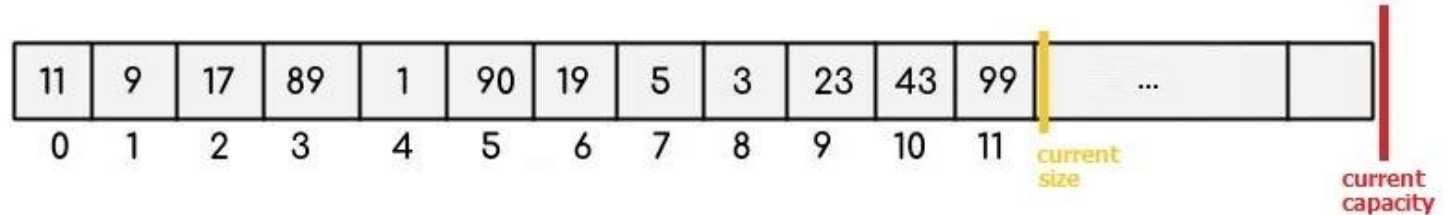
Introduction to Arrays

➢ **Following are the basic operations** supported by an array.

- Insertion: Adds an element at the given index.
- Deletion: Removes an element at the given index.
- Search: Searches an element using the given index or by the value.
- Traverse: Print all the array elements one by one.

➢ **Array Types**

1. Array Static Length

11	9	17	89	1	90	19	5	3	23	43	99	...	
0	1	2	3	4	5	6	7	8	9	10	11		

current size   MAX_SIZE

2. Array Dynamic Length

11	9	17	89	1	90	19	5	3	23	43	99	...	
0	1	2	3	4	5	6	7	8	9	10	11		

current size   current capacity

Reserve Approach - Dynamic Array

➢ **A Dynamic array automatically grows** when we try to make an insertion and there is no more space left for the new item. Usually the area doubles in size.

➢ **A simple dynamic array** can be constructed by allocating an array of fixed-size, typically larger than the number of elements immediately required. The elements of the dynamic array are stored contiguously at the start of the underlying array, and the remaining positions towards the end of the underlying array are reserved, or unused.

➢ **When all space is consumed, and an additional element is to be added,** the underlying fixed-sized array needs to be increased in size. Typically resizing is expensive because you have to allocate a bigger array and copy over all of the elements from the array you have overgrow before we can finally append our item.

➢ **When we enter an element in array** but array is full then you create a function, this function creates a new array double size or as you wish and copy all element from the previous array to a new array and return this new array. Also, we can reduce the size of the array. and add an element at a given position, remove the element at the end default and at the position also.

Reserve Algorithm - Dynamic Array

- *Reserve Algorithm*:
1. *Initializae a new temporary array*
2. *Copy the items in the old array to the new one*
3. *Delete the old array*
4. *Set the new temporary array to be the array*
5. *Update the capacity of the array*
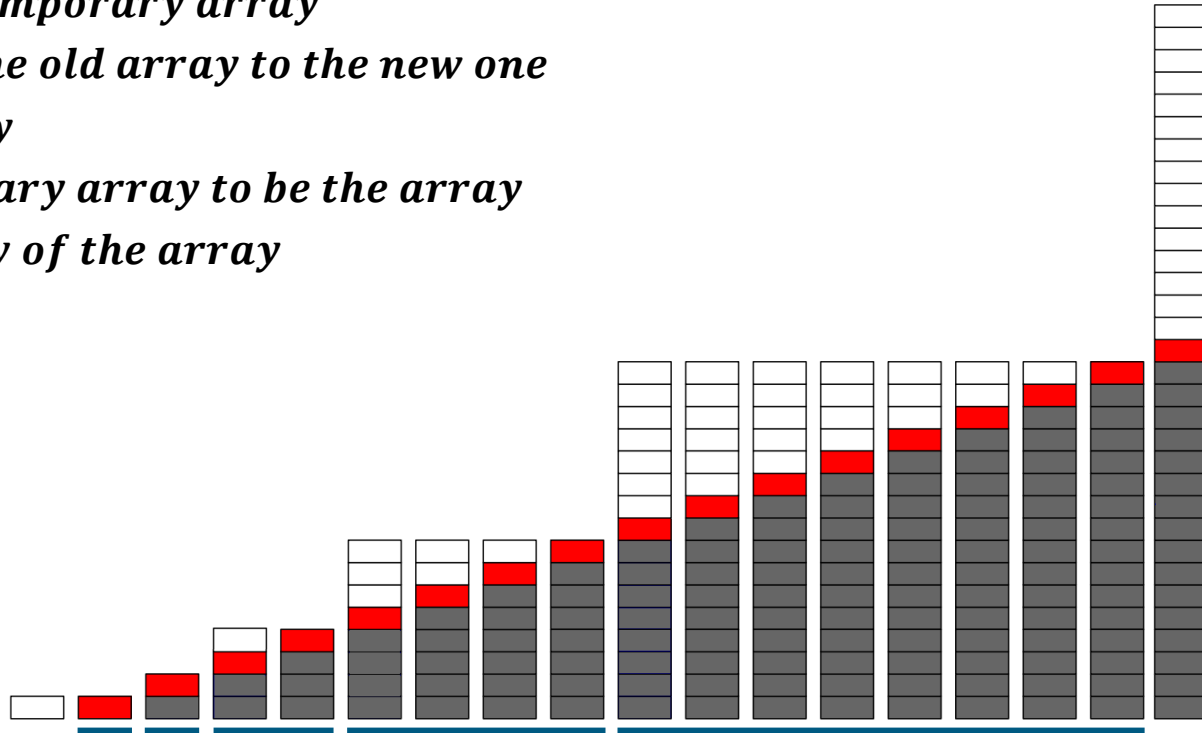
Reserve Method – Dynamic Array

```cpp
// Initialize an array with dynamic length
int n;
int capacity;
int* arr;

// This function updates the capacity of the stack
void reserve(int new_capacity) {
    // Initialize a new stack with the new capacity
    int* temp = new int[new_capacity];
    // copy the elements in the current stack to the new stack
    for (int i = 0; i < n; i++)
        temp[i] = arr[i];
    // delete the old stack
    delete[] arr;
    // set the temp stack with new capacity to be the stack
    arr = temp;
    // set the current capacity of the stack to be the new capacity
    capacity = new_capacity;
}
```

Array-Dynamic-Length.cpp

Lecture Agenda

✔ Section 1: Introduction to Arrays

Section 2: Insertion Operation

Section 3: Deletion Operation

Section 4: Search Operation

Section 5: Traverse Operation

Section 6: Time Complexity & Space Complexity

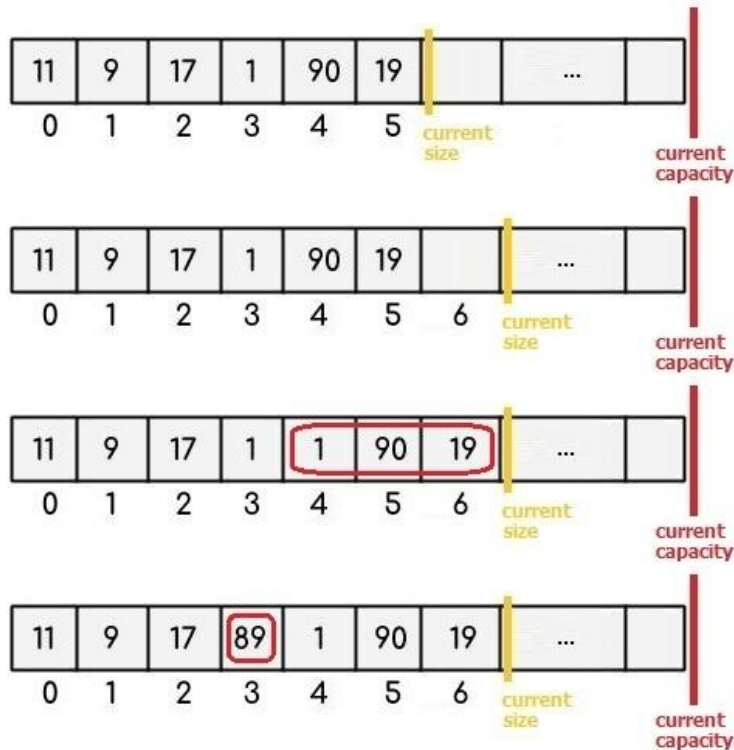# Insertion Operation – Dynamic Array

- **Insert operation** is to insert one or more data elements into an array. Based on the requirement, a new element can be added at the beginning, end, or any given index of array.

- **_Insertion Algorithm_**:

If the array is full increase the capacity

1. $j = n$
2. $arr[j + 1] = arr[j]$
3. $j = j - 1$
4. **Repeat Step 2 if** $j \geq idx$
5. $arr[idx] = item$
6. $n = n + 1$

- Insert 89 at index 3

Insertion Operation – Dynamic Array

```cpp
// This function inserts an element at the given index in the array
void insert_element(int item, int idx) {
    // check for invalid index
    if (idx < 0 || idx > n)
        return;
    // check if we need to update the capacity of the array
    if (n == capacity)
        reserve(2 * capacity + 1);
    // loop to shif values till reach the given index
    int j = n;
    while (j >= idx) {
        arr[j+1] = arr[j];
        j = j - 1;
    }
    // insert the new element
    arr[idx] = item;
    // update the size of the array
    n = n + 1;
}
```

Array-Dynamic-Length.cpp

© Prepared by: Mohamed Ayman

Lecture Agenda

✔ Section 1: Introduction to Arrays

✔ Section 2: Insertion Operation

Section 3: Deletion Operation

Section 4: Search Operation

Section 5: Traverse Operation

Section 6: Time Complexity & Space Complexity
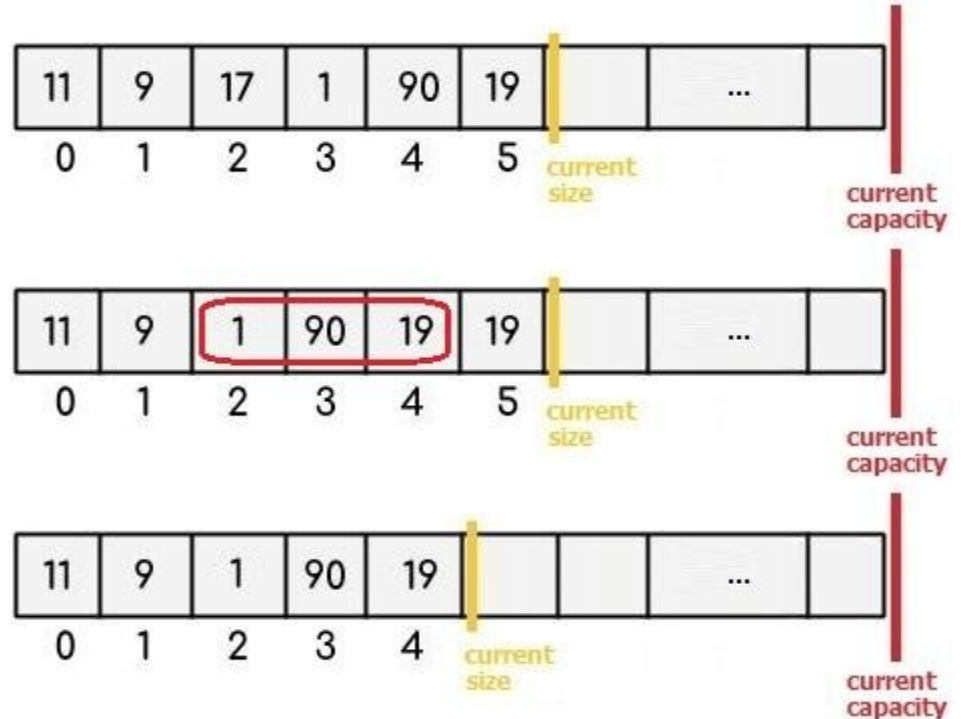
Deletion Operation - Dynamic Array

- **Deletion operation** refers to removing an existing element from the array and reorganizing all elements of an array.

- **Deletion Algorithm**:

1. $j = idx$
2. $arr[j] = arr[j+1]$
3. $j = j + 1$
4. **Repeat Step 2 if $j < n$**
5. $n = n - 1$

If the array is half full decrease the capacity

- Delete 17 in index 2

Deletion Operation – Dynamic Array

```cpp
// This function deletes an element at index idx in the array
void delete_element(int idx) {
    // check for invalid index
    if (idx < 0 || idx >= n)
        return;
    // loop to shif values till reach end of the array
    int j = idx;
    while (j < n) {
        arr[j] = arr[j+1];
        j = j + 1;
    }
    // update the size of the array
    n = n - 1;
    // check if we need to update the capacity of the array
    if (n < capacity / 2)
        reserve(capacity / 2);
}
```

Array-Dynamic-Length.cpp

Lecture Agenda

✔ Section 1: Introduction to Arrays

✔ Section 2: Insertion Operation

✔ Section 3: Deletion Operation

Section 4: Search Operation

Section 5: Traverse Operation
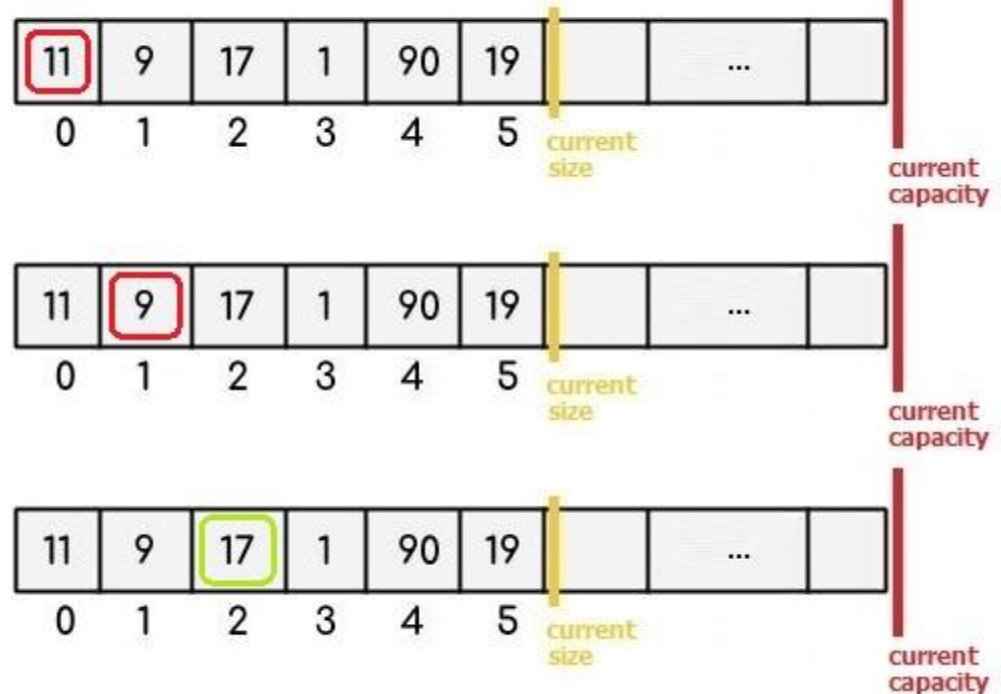
Section 6: Time Complexity & Space Complexity

Search Operation – Dynamic Array

- **Search operation** perform a search for an array element based on its value or its index.

- *Search Algorithm*:
1. $j = 0$
2. $if\ arr[j] == item\ then\ item\ found$
3. $j = j + 1$
4. $Repeat\ Step\ 2\ if\ j < n$
5. $item\ not\ found$

- Search for 17

Search Operation – Dynamic Array

```cpp
// This function searches for an element in the array
bool search_element(int item) {
    int j = 0;
    // loop to find the given element in the array
    while (j < n) {
        if (arr[j] == item)
            // return that the element in the array
            return true;
        j = j + 1;
    }
    // return that the element not in the array
    return false;
}
```

Array-Dynamic-Length.cpp

Lecture Agenda

✔ Section 1: Introduction to Arrays

✔ Section 2: Insertion Operation

✔ Section 3: Deletion Operation

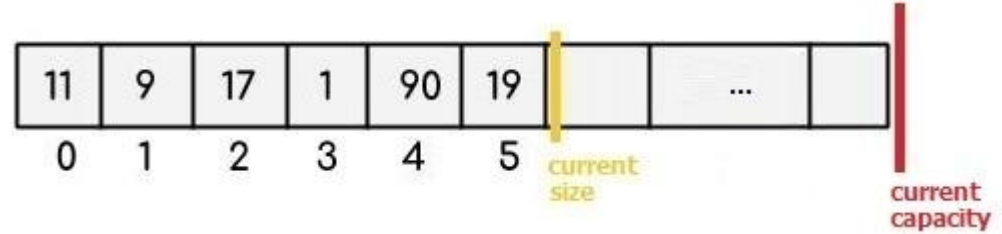✔ Section 4: Search Operation

Section 5: Traverse Operation

Section 6: Time Complexity & Space Complexity
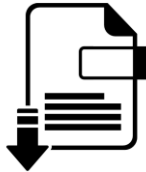
Traverse Operation – Dynamic Array

- **Traverse operation** is to traverse through the elements of an array.

- Traverse this array

- ***Traverse Algorithm***:

1. $i = 0$
2. $print\ arr[i]$
3. $i = i + 1$
4. $Repeat\ Step\ 2\ if\ i < n$

```cpp
// This function prints the contents of the array
void print_array() {
    int i = 0;
    // loop to print the elements in the array
    while (i < n) {
        cout << arr[i] << ' ';
        i = i + 1;
    }
}
```

Array-Dynamic-Length.cpp

Functionality Testing – Dynamic Array

➢ Initialize a global array

```cpp
#include <bits/stdc++.h>
using namespace std;

// Initialize an array with dynamic length
int n;
int capacity;
int* arr;
```

➢ In the Main function:

```cpp
cout << "Array capacity: " << capacity << '\n';
cout << "Array size: " << n << '\n';
cout << "Array items: ";
print_array();
cout << '\n';
```

➢ Expected Output:

```
Array capacity: 0
Array size: 0
Array items:
```

Array-Dynamic-Length.cpp

Functionality Testing – Dynamic Array

➢ In the Main function:

```cpp
cout << "adding the following elements 10 20 30 40 50\n";
insert_element(10, n);
insert_element(20, n);
insert_element(30, n);
insert_element(40, n);
insert_element(50, n);
cout << "the above elements have been added to the array\n";
cout << "Array capacity: " << capacity << '\n';
cout << "Array size: " << n << '\n';
cout << "Array items: ";
print_array();
cout << '\n';
```

➢ Expected Output:

```
adding the following elements 10 20 30 40 50
the above elements have been added to the array
Array capacity: 7
Array size: 5
Array items: 10 20 30 40 50
```

Array-Dynamic-Length.cpp

Functionality Testing – Dynamic Array

➢ In the Main function:

```cpp
cout << "add element 60 at position " << n << " : \n";
insert_element(60, n);
cout << "Array capacity: " << capacity << '\n';
cout << "Array size: " << n << '\n';
cout << "Array items: ";
print_array();
cout << '\n';
cout << "add element 20 at position 0 : \n";
insert_element(20, 0);
cout << "Array capacity: " << capacity << '\n';
cout << "Array size: " << n << '\n';
cout << "Array items: ";
print_array();
cout << '\n';
```

➢ Expected Output:

```
add element 60 at position 5 :
Array capacity: 7
Array size: 6
Array items: 10 20 30 40 50 60
add element 20 at position 0 :
Array capacity: 7
Array size: 7
Array items: 20 10 20 30 40 50 60
```

Array-Dynamic-Length.cpp

Functionality Testing – Dynamic Array

➢ **In the Main function:**

```cpp
cout << "add element 70 at position 4 : \n";
insert_element(70, 4);
cout << "Array capacity: " << capacity << '\n';
cout << "Array size: " << n << '\n';
cout << "Array items: ";
print_array();
cout << '\n';
cout << "add element 90 at position " << n-1 << " : \n";
insert_element(90, n-1);
cout << "Array capacity: " << capacity << '\n';
cout << "Array size: " << n << '\n';
cout << "Array items: ";
print_array();
cout << '\n';
```

➢ **Expected Output:**

```
add element 70 at position 4 :
Array capacity: 15
Array size: 8
Array items: 20 10 20 30 70 40 50 60

add element 90 at position 7 :
Array capacity: 15
Array size: 9
Array items: 20 10 20 30 70 40 50 90 60
```

Array-Dynamic-Length.cpp

© Prepared by: Mohamed Ayman

Functionality Testing – Dynamic Array

➢ In the Main function:

```cpp
cout << "delete element at position 0 : \n";
delete_element(0);
cout << "Array capacity: " << capacity << '\n';
cout << "Array size: " << n << '\n';
cout << "Array items: ";
print_array();
cout << '\n';
cout << "delete element at position " << n-1 << " : \n";
delete_element(n-1);
cout << "Array capacity: " << capacity << '\n';
cout << "Array size: " << n << '\n';
cout << "Array items: ";
print_array();
cout << '\n';
```

➢ Expected Output:

```
delete element at position 0 :
Array capacity: 15
Array size: 8
Array items: 10 20 30 70 40 50 90 60

delete element at position 7 :
Array capacity: 15
Array size: 7
Array items: 10 20 30 70 40 50 90
```

Array-Dynamic-Length.cpp

Functionality Testing – Dynamic Array

➢ In the Main function:

```cpp
cout << "delete element at position 3 : \n";
delete_element(3);
cout << "Array capacity: " << capacity << '\n';
cout << "Array size: " << n << '\n';
cout << "Array items: ";
print_array();
cout << '\n';

if (search_element(40))
    cout << "element " << 40 << " in the array\n";
else
    cout << "element " << 40 << " not in the array\n";

if (search_element(100))
    cout << "element " << 100 << " in the array\n";
else
    cout << "element " << 100 << " not in the array\n";
```

➢ Expected Output:

```
delete element at position 3 :
Array capacity: 7
Array size: 6
Array items: 10 20 30 40 50 90
element 40 in the array
element 100 not in the array
```

Array-Dynamic-Length.cpp

© Prepared by: Mohamed Ayman

Lecture Agenda

✔ Section 1: Introduction to Arrays

✔ Section 2: Insertion Operation

✔ Section 3: Deletion Operation

✔ Section 4: Search Operation

✔ Section 5: Traverse Operation

Section 6: Time Complexity & Space Complexity

Time Complexity & Space Complexity

➤ Time Analysis

	Worst Case	Average Case
• Insert at the begin	Θ(n)	θ(n)
• Insert at the end	Θ(n)	θ(1)
• Insert at specific position	Θ(n)	θ(n)
• Delete at the begin	Θ(n)	θ(n)
• Delete at the end	Θ(n)	θ(1)
• Delete at specific position	Θ(n)	θ(n)
• Search	Θ(n)	θ(n)
• Traverse	Θ(n)	θ(n)

Lecture Agenda

✔ Section 1: Introduction to Arrays

✔ Section 2: Insertion Operation

✔ Section 3: Deletion Operation

✔ Section 4: Search Operation

✔ Section 5: Traverse Operation

✔ Section 6: Time Complexity & Space Complexity

Practice

Practice

1- Rearrange an array such that arr[i] = i

2- Reverse an array or string

3- Rearrange array such that arr[i] >= arr[j] if i is even and arr[i] <=arr[j] if i is odd such that j < i

4- Rearrange positive and negative numbers in O(n) time and O(1) extra space

5- Rearrange array in alternating positive & negative items with O(1) extra space

6- Move all zeroes to end of array

7- Minimum swaps required to bring all elements less than or equal to k together

8- Rearrange array such that even positioned are greater than odd

9- Rearrange an array in order - smallest, largest, 2nd smallest, 2nd largest, ..

10- Reorder an array according to given indexes

11- Rearrange positive and negative numbers with constant extra space

12- Arrange given numbers to form the biggest number

13- Rearrange an array in maximum minimum form

14- Move all negative numbers to beginning and positive to end with constant extra space

15- Distinct adjacent elements in an array

Practice

16- Move all negative elements to end in order with extra space allowed

17- Rearrange array such that even index elements are smaller and odd index elements are greater

18- Positive elements at even and negative at odd positions

19- Replace every array element by multiplication of previous and next

20- Find a sorted subsequence of size 3 in linear time

21- Largest subarray with equal number of 0s and 1s

22- Maximum Product Sub-array

23- Replace every element with the greatest element on right side

24- Maximum circular subarray sum

25- Maximize sum of consecutive differences in a circular array

26- Find Index of 0 to be replaced with 1 to get longest continuous sequence of 1s in a binary array

27- Three way partitioning of an array around a given range

28- Generate all possible sorted arrays from alternate elements of two given sorted arrays

29- Minimum number of swaps required for arranging pairs adjacent to each other

30- Replace two consecutive equal values with one greater

Assignment

Implement STL Vector

- Vectors are same as dynamic arrays with the ability to resize itself automatically when an element is inserted or deleted, with their storage being handled automatically by the container. Vector elements are placed in contiguous storage so that they can be accessed and traversed using iterators. In vectors, data is inserted at the end. Inserting at the end takes differential time, as sometimes there may be a need of extending the array. Removing the last element takes only constant time because no resizing happens. Inserting and erasing at the beginning or in the middle is linear in time.

- Vectors are sequence containers representing arrays that can change in size. Just like arrays, vectors use contiguous storage locations for their elements, which means that their elements can also be accessed using offsets on regular pointers to its elements, and just as efficiently as in arrays. But unlike arrays, their size can change dynamically, with their storage being handled automatically by the container.

Implement STL Vector

- Internally, vectors use a dynamically allocated array to store their elements. This array may need to be reallocated in order to grow in size when new elements are inserted, which implies allocating a new array and moving all elements to it. This is a relatively expensive task in terms of processing time, and thus, vectors do not reallocate each time an element is added to the container.

- Instead, vector containers may allocate some extra storage to accommodate for possible growth, and thus the container may have an actual capacity greater than the storage strictly needed to contain its elements (i.e., its size). Libraries can implement different strategies for growth to balance between memory usage and reallocations, but in any case, reallocations should only happen at logarithmically growing intervals of size so that the insertion of individual elements at the end of the vector can be provided with amortized constant time complexity. Therefore, compared to arrays, vectors consume more memory in exchange for the ability to manage storage and grow dynamically in an efficient way.

More Info: cplusplus.com/reference/vector/vector/
More Info: en.cppreference.com/w/cpp/container/vector
More Info: geeksforgeeks.org/vector-in-cpp-stl/

Implement STL Vector

- Member functions: (constructor) Construct vector (public member function)

   (destructor) Vector destructor (public member function)

   (operator=) Assign content (public member function)

- Iterators: (begin) Return iterator to beginning (public member function)

   (end) Return iterator to end (public member function)

   (rbegin) Return reverse iterator to reverse beginning (public member function)

   (rend) Return reverse iterator to reverse end (public member function)

   (cbegin) Return const_iterator to beginning (public member function)

   (cend) Return const_iterator to end (public member function)

   (crbegin) Return const_reverse_iterator to reverse beginning (public member function)

   (crend) Return const_reverse_iterator to reverse end (public member function)

Implement STL Vector

- Capacity:
  (size) Return size (public member function)

  (max_size) Return maximum size (public member function)

  (resize) Change size (public member function)

  (capacity) Return size of allocated storage capacity (public member function)

  (empty) Test whether vector is empty (public member function)

  (reserve) Request a change in capacity (public member function)

  (shrink_to_fit) Shrink to fit (public member function)

- Element access:
  (operator[]) Access element (public member function)

  (at) Access element (public member function)

  (front) Access first element (public member function)

  (back) Access last element (public member function)

Implement STL Vector

- Modifiers: (assign) Assign vector content (public member function)

  (push_back) Add element at the end (public member function)

  (pop_back) Delete last element (public member function)

  (insert) Insert elements (public member function)

  (erase) Erase elements (public member function)

  (swap) Swap content (public member function)

  (clear) Clear content (public member function)

More Info: cplusplus.com/reference/vector/vector/
More Info: en.cppreference.com/w/cpp/container/vector