بسم الله الرحمن الرحيم

الفهرس

Lesson 1 : What is Data Structure ? and Why ?

أمثلة على **Data Structures** التي تم دراستها سابقا

- Arrays
- Vector
- String ....

ما هو Program ؟

Program = Algorithms + Data Structure

١. **Algorithms** هو : الخطوات المتبعة في حل البرنامج
٢. Algorithms لابد له من أدوات تساعد على تطبيق الحل وهذه الأدوات تسمى **Data Structure**

وهذه الأدوات تنقسم الى قسمين :

- **أدوات متعلقة باللغة** مثل .... if , for , while (أدوات اللغة)
- **وأدوات تخزن Data وتمثلها في Memory** بحيث تستطيع الوصول إليها أثناء تشغيل البرنامج بسرعة وبكفاءة عالية جدا (**Data Structure**)

مثال للخوارزمية لعمل برنامج يقرأ اسمك ويطبعه على الشاشة

١. طباعة جملة على الشاشة " Please Enter your Name ? "
٢. تعريف متغير من نوع string Name وهو من أنواع (**Data Structure**)
٣. قراءة الاسم من الشاشة وتخزينه في متغير string Name
٤. طباعة الاسم على الشاشة – المخزن في المتغير –

Data Structure : لها علاقة بالتعامل مع البيانات من داخل البرنامج – في الكود –
أو هي وسيلة لتمثيل البيانات داخل البرنامج

طريقة لتنظيم كافة عناصر البيانات وعلاقتها ببعضها البعض من أجل التعامل معها بأفضل طريقة ممكنة

طريقة حلك تتأثر باستخدامك لأنواع **Data Structure** كلما كانت معرفتك بأنواع **Data Structure** كلما استخدمت النوع المناسب منها

إذا لم تعرف جميع أنواع **Data Structure** فستظل حلك محدود ، بحيث تستخدم أنواع من **Data Structure** في البرنامج بحسب ما يناسبها ، فتكون للبرنامج هيكلة للبيانات بحيث تستطيع عمل Storing , Processing ,Retrieving والتعامل معها بكل سلاسة وسهولة

Data Structure ترتب وتدير وتخزن البيانات بطريقة فعالة ، ولها دور في تحسين البرنامج

لابد من معرفة ما هو النوع الملائم من **Data Structure** لبرنامجك ؟ وما فوائد استخدامه ؟ أو مساوئ وهل استخدامه فعال أم لا ؟

Answer	Question
Fales	**Program = Algorithms**
True	**Program = Algorithms + Data Structure**
True	**Data Structure : is a way of organizing all data items and their relationships to each others inside the program in order to deal with them**
False	**Data Structure : does not affects the design of both structural and functional aspects of the program**
True	**Data structure : affects the design of both structural and functional aspects of the program**
True	**Data Structure : is how u organize , manage and store data for efficiency reasons**
True	**A Data structure : is not only used for organizing the data . It is also used for Processing , Retrieving and storing data**
True	**Various types of data structures have their own characteristics , features , applications , and disadvantages**

#Lesson 2 : Differences between Data Structures and Database

Data Structure هي وسيلة لتمثيل البيانات بداخل البرنامج

كل البيانات أو المعلومات تكون مخزنة في RAM – يعني بمجرد إغلاقه لا تستطيع استرجاع البيانات

Database هي كيفية تخزين البيانات في **Database** – خارج البرنامج مخزنة في Hard disk أو في Server (أشبه ب حفظها في ملف – مثل مشروع البنك – لكن على شكل جدول)

Database هي جداول فيها معلومات وبينهم علاقات

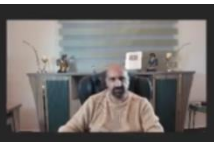
مثال : عند تحويل سطر البيانات في الملف – مبدئيا – هذه هي **Database** – الى أي نوع من **Data Structure** بحيث يستطيع البرنامج قراءته – هذه هي **Data Structure**

DBMS اختصار ل : Database Management System

الفروق بين Data Structure و Database

Data Structure	Database
يتم تخزين البيانات في **RAM** بمجرد إغلاقه لا تستطيع استرجاع البيانات (Temporary)	يتم تخزين البيانات في Hard desk ، وحفظها بشكل دائم (Permanent)
تخزن على شكل خاص	يتم تخزينها على شكل جدول بينهم علاقة
تستخدم لتحقيق الكفاءة وتقليل تعقيدات البرنامج	تستخدم للوصول الى البيانات وإدارة البيانات
لغات البرمجة مثل : C++ , python ... تستخدم Data structure	Structured query language واختصارها SQL وتستخدم لإجراء العمليات على البيانات
أمثلة على Data structure : Array , Linked , List , Stack , Queue , Tree , Graph	أمثلة على Database : MySQL , Oracle , MongoDB , Sybase ....

Data Structure vs Database?



Data Structure : In computer Programming, Data structure is a way of organizing and storing data so as to ease the accessing and modification of data.

Some commonly used data structures are Array, Linked List, Stack, Queue, Heap, Binary Tree and Graph.

Database: is a collection of data that is stored in an organized fashion in a table containing rows and columns using a software package known as Database Management System (DBMS).

DBMS is used to modify, define, manipulate and manage data. Some examples of DBMS are : MySQL, Oracle Database and Microsoft Access.

PROGRAMMING ADVICES

Mohammed Abu-Hadhoud

Data Structures	VS	Database
Data In RAM inside the program. It is a volatile memory (Temporary).		Data In Harddesk outside the program It is a Non-volatile memory (Permenent)
Special format for storing data		Organized collection of data.
used for efficiency and to reduce the complexities of the program		used to access the data and manage it easily
Programming languages C++, Java, Python are used to perform operations using data Structures.		Structured query language (SQL) is used to perform operations on the data
Example : Array, Linked List, Stack, Queue, Tree, Graph.		Example : MySQL, Oracle, MongoDB, Sybase etc.

PROGRAMMING ADVICES

Copyright© 2022
ProgrammingAdivces.com

Mohammed Abu-Hadhoud

Answer	Question
Fales	**Data Structure is the Same as Database**
True	**Data Structure : In computer Programming, Data structure is a way of organizing and storing data so as to ease the accessing and modification of data**
True	**Database: is a collection of data that is stored in an organized fashion in a table containing rows and columns using a software package known as Database Management System (DBMS)**
True	**DBMS is used to modify, define, manipulate and manage data. Some examples of DBMS are : MySQL, Oracle Database and Microsoft Access**
True	**Some commonly used data structures are Array, Linked List, Stack, Queue, Heap, Binary Tree and Graph**
Fales	**Data In Database is stored In RAM inside the program**
True	**Data in data structure is stored In RAM inside the program**
True	**Structured query language (SQL) is used to perform operations on the data in Database**
True	**Programming languages C++, Java, Python are used to perform operations using data Structures**
Fales	**Example of Data Structures : MySQL, Oracle, MongoDB, Sybase etc**
True	**Example Of DBMS (Database) : MySQL, Oracle, MongoDB, Sybase , SQL Server, Access, etc**

Lesson 3 : What are the Classification/Types of Data Structures ?

أنواع Data Structure :

١. Primitive Data Structure بسيطة : أمثلة
- Int , float , char , pointer *
- يتم دمجها في اللغة : تساعدك في تخزينها واسترجاعها من RAM بسهولة
- تعريفها سهل int X = 10
- وتستخدم اللغة machine instruction للتعامل معها في الذاكرة

٢. Non-Primitive Data Structure متطورة معقدة وتسمى **Completes or Advance** :
- ❖ **مشتقة أو مبنية على Primitive Data Structure مثل Array**
- ❖ **تنقسم الى قسمين :**

i. **Linear** Data Structure

١. Linear : **تكون Data فيها مرتبة** ، أي تستطيع الوصول الى جميع عناصرها ب for loop واحدة

٢. Linear لها نوعين

a. **Static** Data Structure :

i. يكون لها حجم ثابت في الذاكرة لا تستطيع تغيير حجمها في Run time ، يتم حجز حجمها بمجرد تعريف نوع من Data Structure مثل array

ii. سهولة الوصول الى عناصرها

b. **Dynamic** Data Structure

i. يكون الحجم فيها غير ثابت ، تستطيع تغيير حجمها أثناء Run time

ii. مثل vector , Stack , Queue , Linked List , Dynamic Array

ii. **Non-Linear** Data Structure

١. Non-Linear : **تكون Data فيها غير مرتبة** ، أي لا تستطيع الوصول الى جميع عناصرها ب for loop واحدة

٢. أمثلة Graph / Tree

❖ **Data Structure إما أن تكون**

iii. **Homogeneous** تحتوي بداخلها على نوع واحد من Data Structure

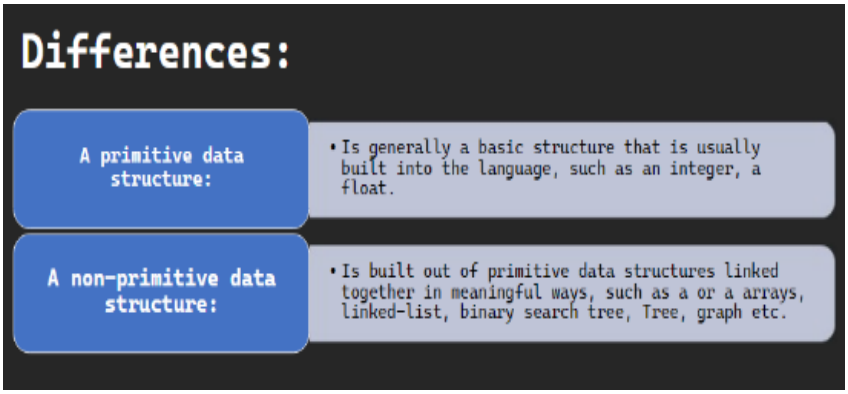
١. مثل int arr[10]

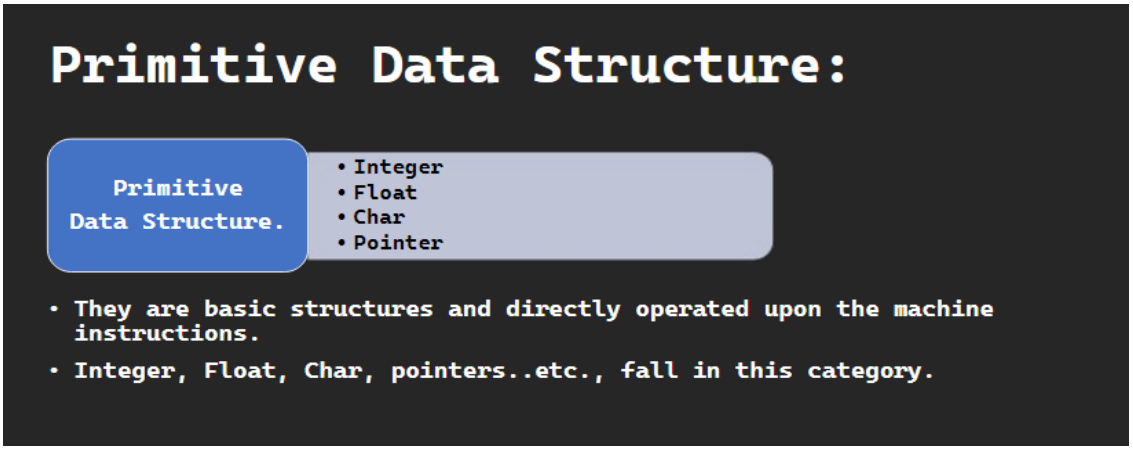
iv. **Heterogeneous** تحتوي بداخلها على أنواع مختلفة من **Data Structure**

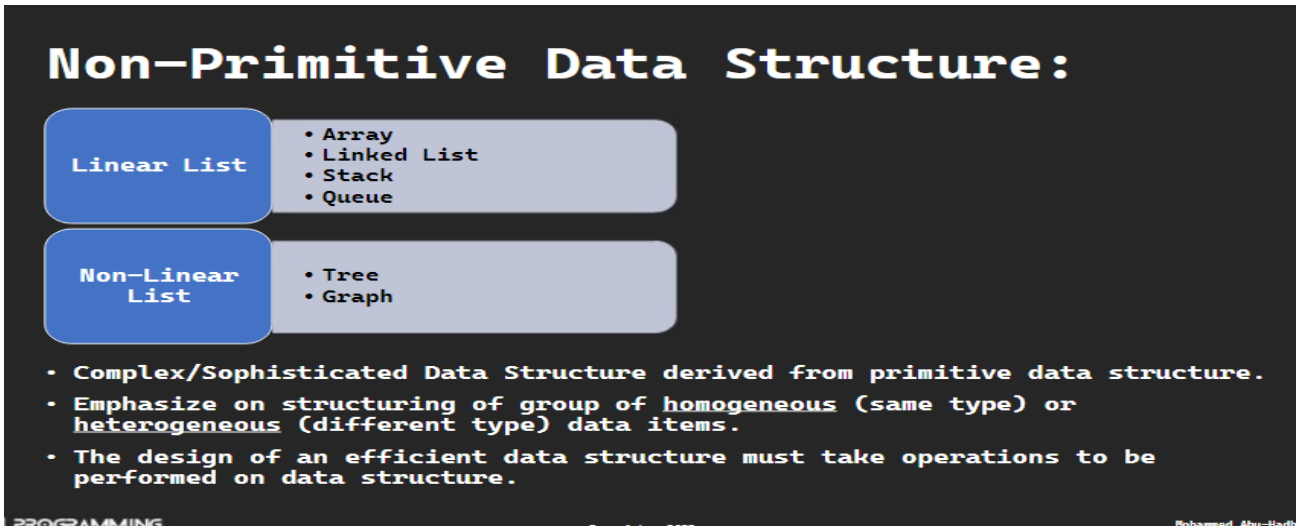
١. مثل Tree , Graph

❖ **العمليات في Data Structure**

١. إنشاء Create
٢. تحديث Update
٣. بحث Search
٤. اختيار Select
٥. ترتيب Sorting
٦. دمج Merging
٧. حذف Delete
٨. تدمير Destroy







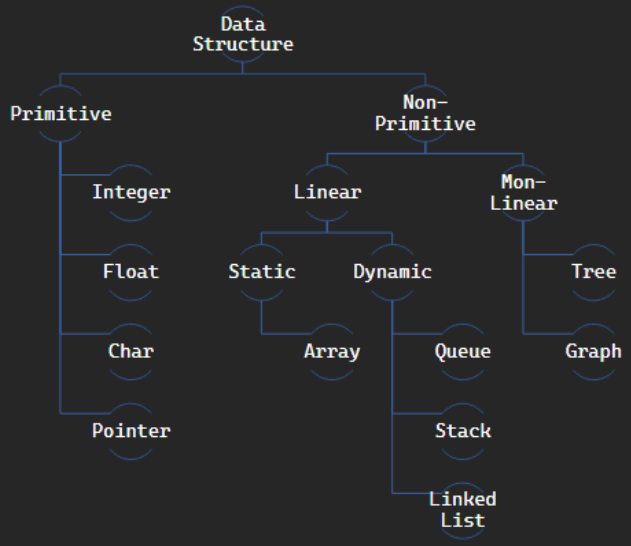
Linear vs Non-Linear Data Structures

Linear Data Structure	• Data structure in which data elements are arranged sequentially or linearly, where each element is attached to its previous and next adjacent elements, is called a linear data structure. • Examples of linear data structures are array, stack, queue, linked list, etc.
Non-Linear Data Structure	• Data structures where data elements are not placed sequentially or linearly are called non-linear data structures. In a non-linear data structure, we can't traverse all the elements in a single run only. • Examples of this data structure are Tree, Graph, etc.

Static vs Non-Static Data Structures

Static data structure	• Static data structure has a fixed memory size. It is easier to access the elements in a static data structure. • An example of this data structure is an array.
Dynamic data structure	• In dynamic data structure, the size is not fixed. It can be randomly updated during the runtime which may be considered efficient concerning the memory (space) complexity of the code. • Examples of this data structure are Stack , Queue, etc.

Classifications of Data Structure:



Answer	Question
Fales	**Integer is non-Primitive data structure**
True	**Integer is Primitive data structure**
True	**Array is Non-Primitive Data Structure**
True	**Non-Primitive is derived from Primitive Data Structure**
True	**Heterogenous means Different Types**
True	**Homogenous means Same Types**
Fales	**Linear Data Structure means that data is in non-sequence order**
True	**Linear Data Structure means that data is in sequence order**
Non-Linear, Non-Primitive Data Structure	**Tree and Graph are**
Linear, Non-Primitive Data Structure	**Array is**
Homogenous Data Structure	**Array is**
Dynamic Data Structure	**Stack, Queue, Linked Lists are**

Lesson 4 : Things Affect Program Speed & Efficiency

ما هي العوامل التي تؤثر في سرعة أو أداء البرنامج :

- **Algorithm** الخوارزمية :
  a. الوقت **Time** الذي يتم فيه تشغيل البرنامج
  b. المساحة **Space** التي تأخذها الخوارزمية من الذاكرة أو حجم استهلاك موارد الجهاز
  - كلاها يؤثران على الخوارزمية
  - Time و تسمى : **Complexity Time**
  - Space وتسمى : **Complexity Space**

٢. **Hardware** و تسمى **Performance**
  - CPU , RAM , OS
  - هذه تؤثر على سرعة البرنامج

❖ المطور **Developer** له علاقة فقط في **Algorithm** : أي Time / Space Complexity
❖ كلما كان Time / Space Complexity أقل كلما كان البرنامج أسرع – على حسب البرنامج –

Answer	Question
Hardware (CPU, RAM) Operating System Algorithm	**What Affects Program Speed ?**
Time Space	**What affects Algorithm ?**
Hardware , OS	**Performance has to do with ?**
Time Complexity Space Complexity	**Complexity has to do with ?**
Algorithms	**Developer should focus on ?**

Lesson 5 : Time & Space Complexity - Big O Notation

يوجد أكثر من طريقة (خوارزمية) لحل أي برنامج ، و **Big O هو لقياس الحالة الأسوأ** – للخوارزمية وليس **Hardware** – وتقدير كفاءتها مقارنة بحجم الإدخال

مثال : عندك 1000 طالب وتبحث عن اسم (محمد) – عن طريق for loop – أسوأ احتما هو أن يكون آخر اسم 1000 وهذه هي الحالة الأسوأ في Big O

Big O هو معادلة لقياس العلاقة بين المدخلات وتأثر الخوارزمية بذلك – (سواء Time / Space)

O(n) == Time vs Input = **Complexity Time**

O(n) == Space vs Input = **Complexity Space**

معادلة Big O هي : O(n) – سيدرس لاحقا –

- O اختصار ل : Order of
- (n) اختصار ل : input size

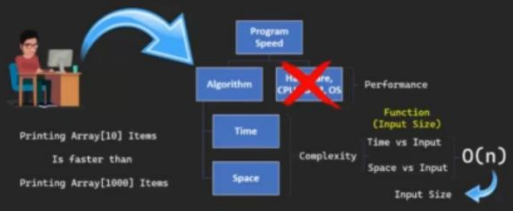
بعض أشهر أنواع – العلاقات في - Big O

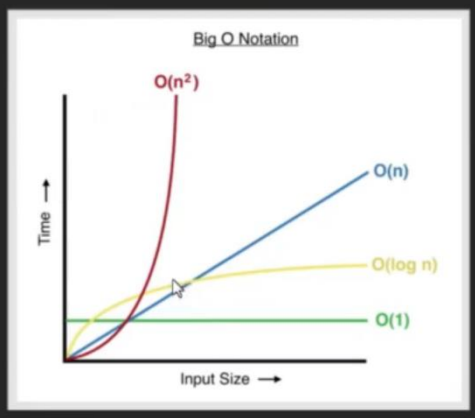
١. **O(1)** معناه ثابت : أي مهما كان حجم input size فإن Time لا يتأثر – سريع
٢. **O(log n)** : عدد التكرار في for loop يكون أقل من input size
٣. **O(n)** معناها : كل زاد Input زاد Time – متوازي –
٤. $O(n^2)$ معناها : كل زاد Input تضاعف Time – مضاعفة / تربيع –

Big O لا يعطيك كم من الوقت تحتاج لتشغيل البرنامج وإنما علاقة بين المدخلات والوقت

Big O ليس له علاقة ب **Hardware**

What is Big O?





PROGRAMMING ADVICES

Mohammed Abu-Hadhoud

Time & Space Complexity

An algorithm's time complexity: specifies how long it will take to execute an algorithm as a function of its input size.

Similarly, an algorithm's space complexity: specifies the total amount of space or memory required to execute an algorithm as a function of the size of the input.

What Big O is NOT?

- Big O is not going to give you an exact answer on how long a piece of code will take to run.
- Does not consider the performance of hardware.

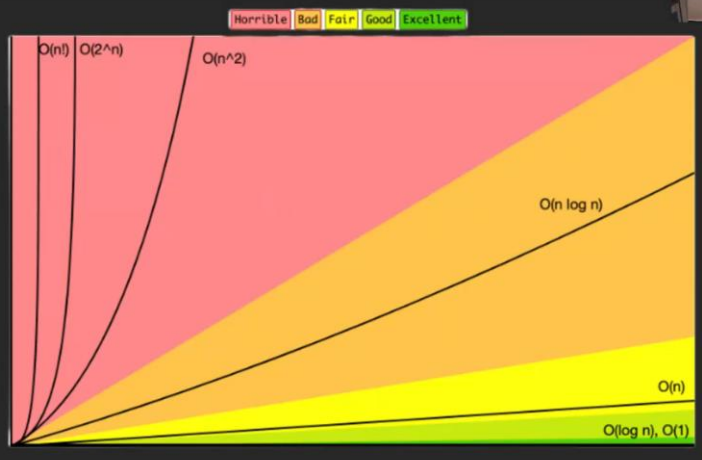
What is Big O Notation?

- The big-O originally stands for “Order Of“.
- Efficiency of Algorithm
- Time Factor of Algorithm
- Space Complexity of Algorithm
- Represented by O(n) where n is the number of Inputs.
- Big O, also known as Big O notation, represents an algorithm's worst-case complexity.
- It uses algebraic terms to describe the complexity of an algorithm.
- Big O defines the runtime required to execute an algorithm, by identifying how the performance of your algorithm will change as the input size grows.
- Relationship between Input Size and Performance.
- But it does not tell you the exact time your algorithm's runtime is.
- Big O notation measures the efficiency and performance of your algorithm using time and space complexity.
- Big O allows us to discuss our code algebraically to get a sense of how quickly it might operate under the strain of large data sets.
- with Big O Notation, we can look at our algorithm and see that it will take O(n) time to run. Big O Notation, written as O(blank), show us how many operations our code will run, and how its runtime grows in comparison to other possible solutions.

PROGRAMMING ADVICES LEARN THE RIGHT WAY

Mohammed Abu-Hadhoud
MBA, PMOC, PgMP®, PMP®, PMI-RMP®, CM, ITILF, MCPD, MCSD
26+ years of experience

What is Big O?



PROGRAMMING ADVICES LEARN THE RIGHT WAY

Mohammed Abu-Hadhoud
MBA, PMOC, PgMP®, PMP®, PMI-RMP®, CM, ITILF, MCPD, MCSD
26+ years of experience

Answer	Question
True	**An algorithm's time complexity: specifies how long it will take to execute an algorithm as a function of its input size**
True	**An algorithm's space complexity: specifies the total amount of space or memory required to execute an algorithm as a function of the size of the input**
True	**The big-O originally stands for “Order Of"**
True	**Big O tells us about the Efficiency of Algorithm**
True	**Big O, also known as Big O notation, represents an algorithm's worst-case complexity**
False	**Big O represents an algorithm's best-case complexity**
True	**Big O uses algebraic terms to describe the complexity of an algorithm**
True	**Big O describes the relationship between Input Size and Time/Spaces**
False	**Big O tells you the exact time your algorithm's runtime is**
True	**Big O does not tell you the exact time your algorithm's runtime is**
True	**with Big O Notation, we can look at our algorithm and see that it will take O(n) time to run. Big O Notation, written as O(blank), show us how many operations our code will run, and how its runtime grows in comparison to other possible solutions**

Lesson 6 : Big O(1) : Constant Time Function

Big O(1) : يعني أن هذه الخوارزمية ليست مربوطة بعدد المدخلات Input size ، مهما كان عدد المدخلات يبقى ثابت وهو أسرع شيء ، وتسمى **Time Complexity**

مثال : إنشاء برنامج لإرجاع آخر حرف من النص string ؟ يوجد أكثر من حل منها

١) هذه الخوارزمية لا يتغير فيها **Time Complexity** لأنها مهما أخذت من مدخلات تظل بها نفس عدد الخطوات **Time** لإنهاء البرنامج – غير مربوطة بعدد الأحرف في النص –

```
char GetLastCharacter(string S1)
{
    return S1[S1.length() - 1];
}
```

٢) هذه الخوارزمية مربوطة بعدد الأحرف في النص Input size ، فتوجد علاقة بين Input size – و for loop على حسب عدد حروف النص فيزيد Time – هذه O(n) –

```
char GetLastCharacter2(string S1)
{
    int n = S1.length() - 1;

    for (int i = 0; i <= n; i++)
    {
        if (i == n)
        {
            return S1[n];
        }
    }
}
```

لحساب Big O لأي خوارزمية : حساب عدد خطوات لإنهاء الخوارزمية

- لسهولة الحساب نفرض أن كل خطوة = Time
- كل خطوة تأخذ نفس Time مثال : "return" == Time "+" , "-" .....

❖ تطبيق على الخوارزمية الأولى Big O(1) – **لعدد الخطوات** –

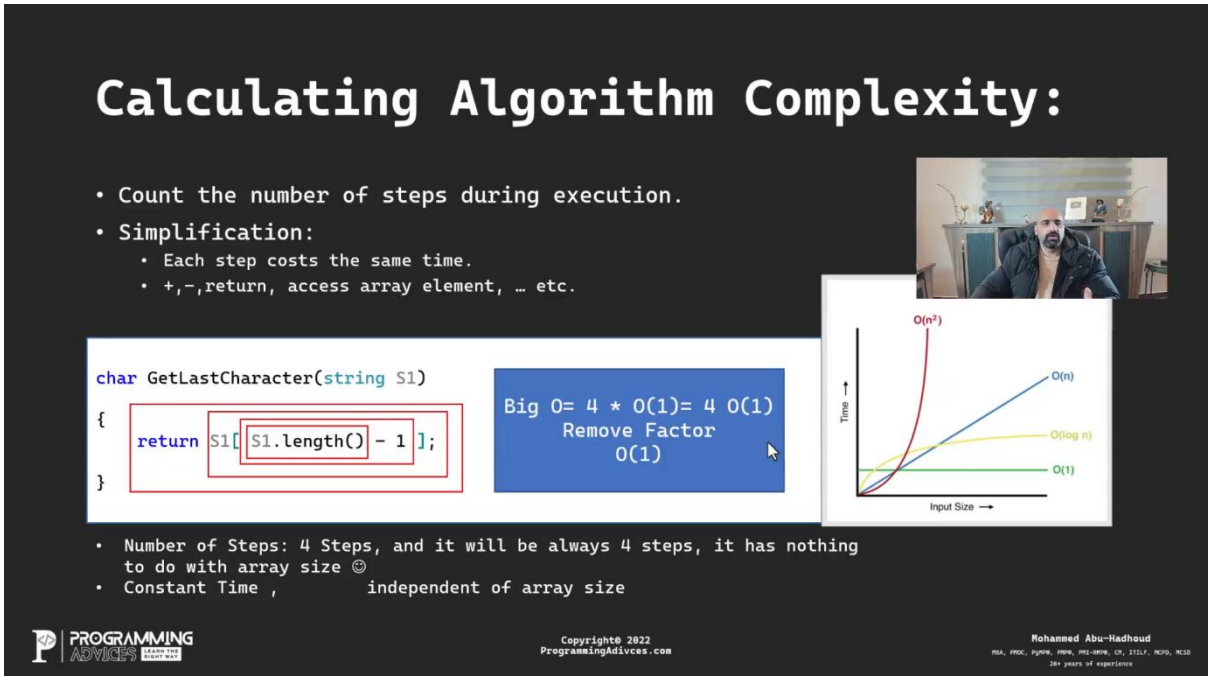
١. `S1.length()`
٢. `S1.length() - 1`
٣. `S1[S1.length() - 1];`
٤. `return S1[S1.length() - 1];`

❖ هذا هو Big O(1) لأن الخطوات تكون ثابتة مهما كان عدد Input size

❖ ولهاذا يسمى Constant Time Function

معادلة Big O(1)

Big O = عدد الخطوات "4" * O(1) = O(1) 4 "يتم حذف العامل = 4 " == O(1)



Answer	Question
True	**O(1) Means Constant Time Function ?**
False	**O(1) is affected by Input Size ?**
True	**O(1) is not affected by Input Size, so it always takes the same time**
True	**O(1) is the fastest notation, which means that your algorithm is excellent**

Lesson 7 : Big O(n) : Linear Time Function

Big O(n) : يعني أن هذه الخوارزمية مربوطة بعدد المدخلات Input size ، كلما كان عدد المدخلات أكبر كلما زاد Time ، تكون الزيادة ثابتة – بشكل متوازي (خطي / **Linear**) –

هذه الخوارزمية مربوطة بعدد الأحرف في النص Input size ، فتوجد علاقة بين Input size – و for loop على حسب عدد حروف النص فيزيد Time – هذه O(n) –

```
char GetLastCharacter2(string S1)
{
    int n = S1.length() - 1;

    for (int i = 0; i <= n; i++)
    {
        if (i == n)
        {
            return S1[n];
        }
    }
}
```

- تطبيق على الخوارزمية Big O(n) – **لعدد الخطوات** –
- عدد الخطوات خارج Loop

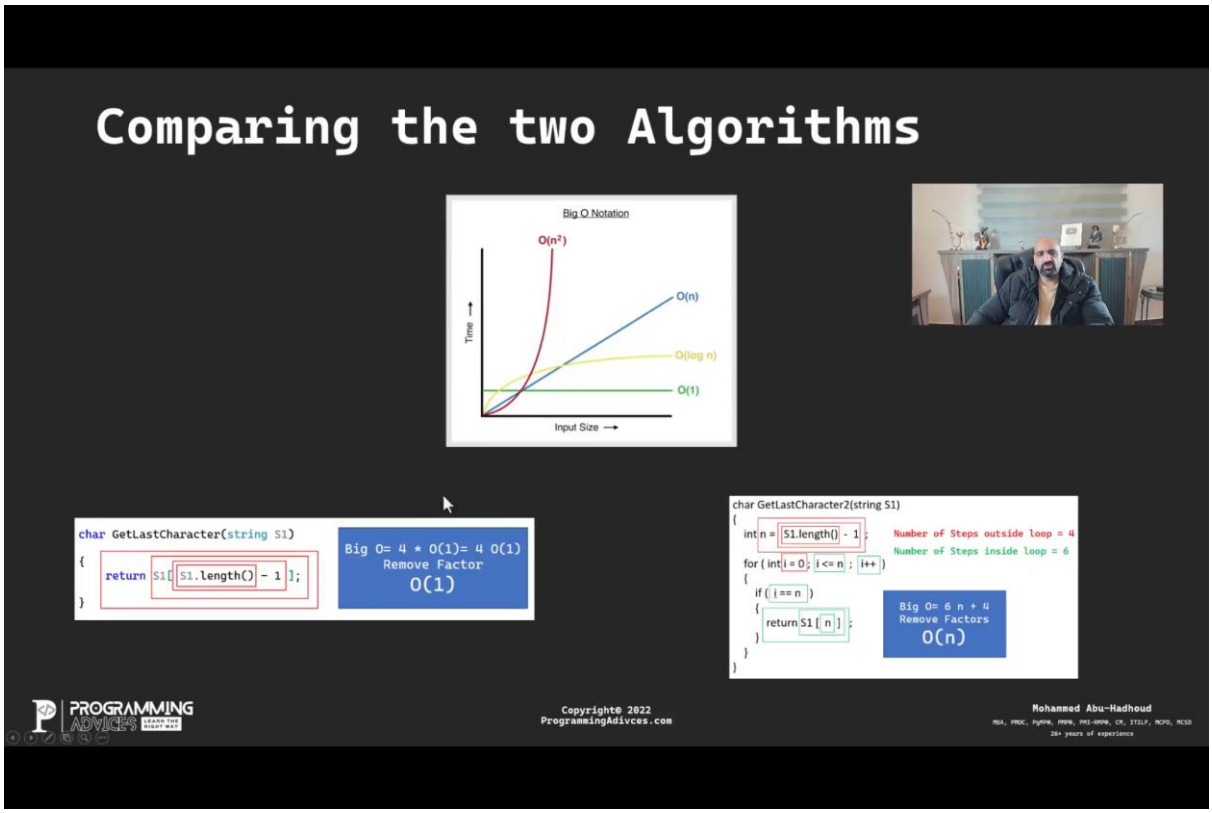
١. `S1.length()`
٢. `S1.length() - 1`
٣. `int n = S1.length() - 1;`
٤. `int i = 0;`

- عدد الخطوات داخل Loop

١. `i <= n`
٢. `i++`
٣. `i == n`
٤. `[n];`
٥. `S1[n];`
٦. `return S1[n];`

معادلة Big O(n)

Big O = عدد الخطوات داخل Loop "6" * n المدخل + عدد الخطوات خارج Loop "4" == O(n)



Answer	Question
False	**O(n) is a constant time function**
True	**O(n) is a linear time function**
False	**(1)O(n) is faster than O**
True	**O(1) is faster than O(n)**
True	**O(n) is affected by Input Size Linearly**

Lesson 8 : Big O(n^2) : Quadratic Time Function

Big O(n^2) : يعني أن هذه الخوارزمية مربوطة بعدد المدخلات Input size ، كلما كان عدد المدخلات أكبر كلما زاد Time ضعف ، تكون الزيادة مضاعفة – بشكل تربيع – يسمي **Quadratic**

هذه الخوارزمية مربوطة بعدد Input size ، فتوجد علاقة بين Input size – و for loop1 و for loop2 على حسب عدد المدخل فيتضاعف Time – هذه O(n^2) –

```
int MultiplicationSum(short n)
{
    int Sum = 0;

    for (short i = 1; i <= n; i++)
    {
        for (short j = 1; j <= n; j++)
        {
            Sum = Sum + (i * j);
        }
    }
    return Sum;
}
```

- تطبيق على الخوارزمية Big O(n^2) – **لعدد الخطوات** –
- عدد الخطوات داخل Loop 2 – الثانية / الداخلية –

١. j <= n;
٢. j++
٣. (i * j);
٤. Sum + (i * j);
٥. Sum = Sum + (i * j);

- حساب Loop 2 ب Big O هو
  - 5 * n = n

- عدد الخطوات داخل Loop 1 – الأولى –

١. i <= n;
٢. i++
٣. short j = 1;
٤. زائد + عدد خطوات for loop 2 = 5

- حساب Loop 1 ب Big O هو
  - (عدد تكرار for loop 2 = n + عدد خطوات for loop 1 = 3) * n
  - = 3n + المدخل "n" تربيعه بعدد for loop = n^2
  - الجواب = n^2

❖ عدد الخطوات خارج Loop

١. `Sum = 0;`
٢. `short i = 1;`
٣. `Sum`
٤. `return Sum`

- الحساب
  - عدد الخطوات خارج loop = $n^2 + 4$
  - n^2 =

مثال : الصلاحيات في مشروع البنك Permissions ؟

١. تم حله بتمثيل كل صلاحية برقم Binary وتم استخدام && Bit - O(1) –

```
bool CheckAccessPermission(enPermissions Permission)
{
    if (this->Permissions == enPermissions::eAll)
        return true;

    if ((Permission & this->Permissions) == Permission)
        return true;
    else
        return false;

}
```

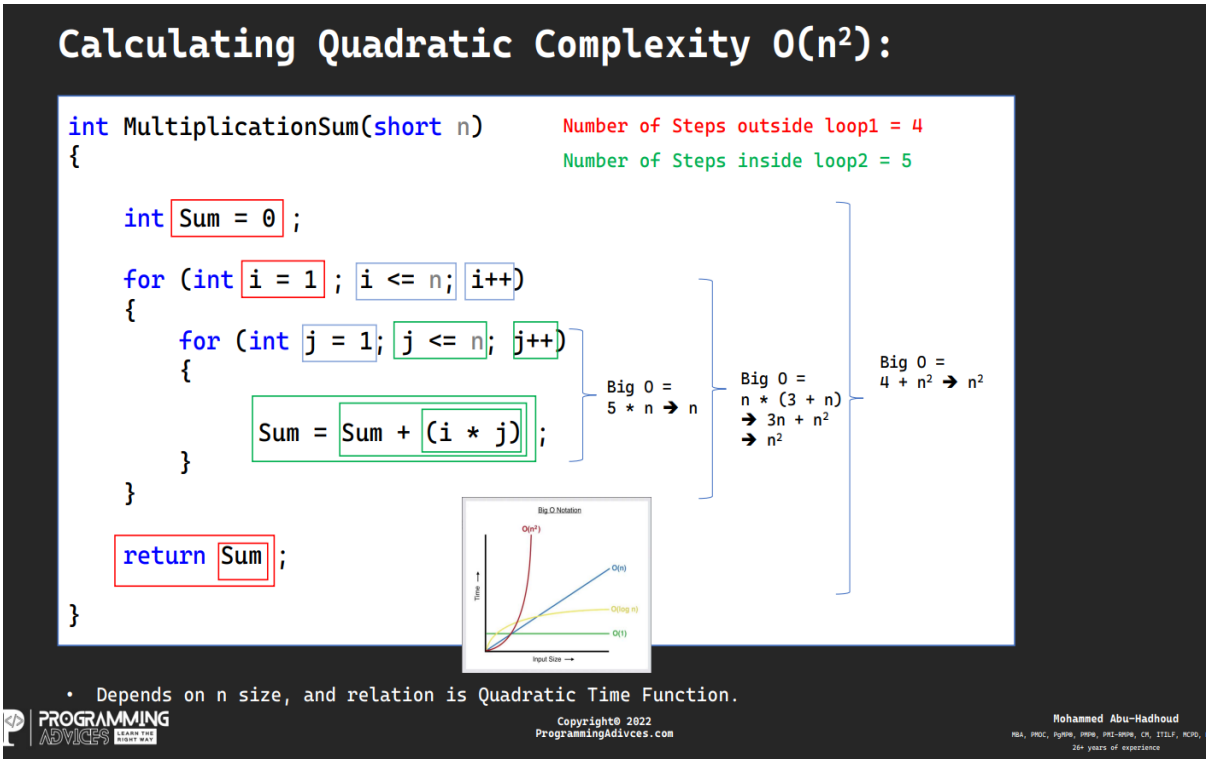
٢. استخدام 2 for loop – $O(n^2)$ –

a. for loop 1 : من 1 الى عدد User
b. for loop 2 بداخل for loop 1 : من 1 الى عدد Permissions

- مع المقارنة هل User له صلاحية ؟

كلما زاد عدد for loop و تكراره كان البرنامج أبطأ – قد تكون مضطر لاستخدامه –

أسوأ نوع في Big O هو : O(n*n)



Answer	Question
True	**Big O(n^2) is Quadratic Time Function**
True	**O(n) is much faster than O(n^2)**
False	**O(1) is slower than O(n^2)**
True	**O(1) < O(n) < O(n^2) < O(n^3)**
True	**if you have f= 3 + 5n + 6n^2 , then big O is O(n^2)**

Lesson 9 : Big O(Log n) : Logarithmic Time Function

Big O(log n) : يعني أن هذه الخوارزمية مربوطة بعدد المدخلات Input size لكن يتقلص عدد التكرار في for loop بمقدار النصف مثلا – لا يتم تكرار for loop بعدد المدخلات – يسمي **Logarithmic**

مثال : لتناقص عدد التكرار عن المدخلات بمقدار النصف لكل مرة تكرار

```
void fun1(short n)
{
    short x = n;

    while (n > 0)
    {
        x = x / 2;

        cout << x << endl;
    }
}
```

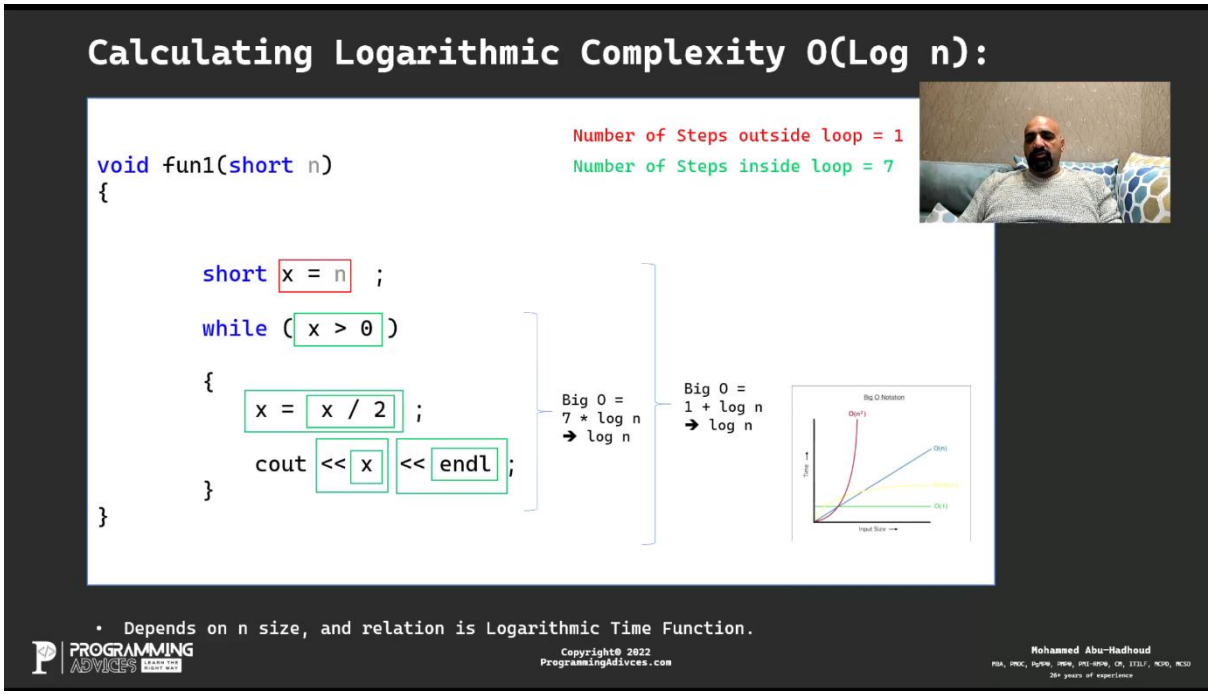
- تطبيق على الخوارزمية Big O(log n) – **لعدد الخطوات** –
- عدد الخطوات داخل while Loop
  1. `(n > 0)`
  2. `x / 2;`
  3. `x = x / 2;`
  4. `x`
  5. `cout << x`
  6. `endl`
  7. `cout << endl`
  - الحساب
    - log = log + 7

- عدد الخطوات خارج while Loop
  1. `short x = n;`
  - الحساب
    - log = log + 1

مثال عملي على **Big O(log n)** هو Binary search – ستدرس لاحقا –

مثال : البحث عن رقم 5 في array مخزن فيها أرقام من 1 الى 100 بشكل مرتب

تقسم array / 2 = 50 ثم تقسم 50 / 2 = 25 ثم 25 / 2 = 12 ثم 12 / 2 = 6
فتبحث من 1 الى 6 – تعرف في أي نطاق تبحث عن 5 –



Answer	Question
True	**Log2 always half itself**
True	**Log means that the number of iterations in the loop is always less than n**
True	**O(log n) is much faster than O(n)**

#Lesson 10 : Important Question :-) , Work Smart

مثال :

```
short FindNumberAlgorhim1(short arr1[10], short Number)

{
    int n = 10;
    short pos = -1;

    for (int i = 0; i <= n; i++)
    {
        if (arr1[i] == Number)
        {
            pos = i;
        }
    }

    return pos;

}

short FindNumberAlgorhim2(short arr1[10], short Number)

{
    int n = 10;

    for (int i = 0; i <= n; i++)
    {
        if (arr1[i] == Number)
        {
            return i;
        }
    }

    return -1;

}
```

إذا تم حل مشكلة بخوارزميتين مختلفتين من النوع O(n) فهل لهما نفس السرعة ؟ سؤال في Interview

- الخوارزمية الأولى بطيئة : لأنها حتى لو وجدت الرقم المطلوب لن تخرج من for loop بل ستستمر الى نهاية العناصر
- الخوارزمية الثانية سريعة : بمجرد عثورها على الرقم المطلوب تخرج من Function
- **وتتساويا من ناحية السرعة – والوقت – إذا كان الرقم المطلوب هو آخر عنصر**
  - إذا تساويا في Big O لا يعني أنهما قد تساويا في نفس السرعة
  - Big O لا يجاوب على الوقت المستغرق للخوارزمية , وإنما على نوعها - O(n) –

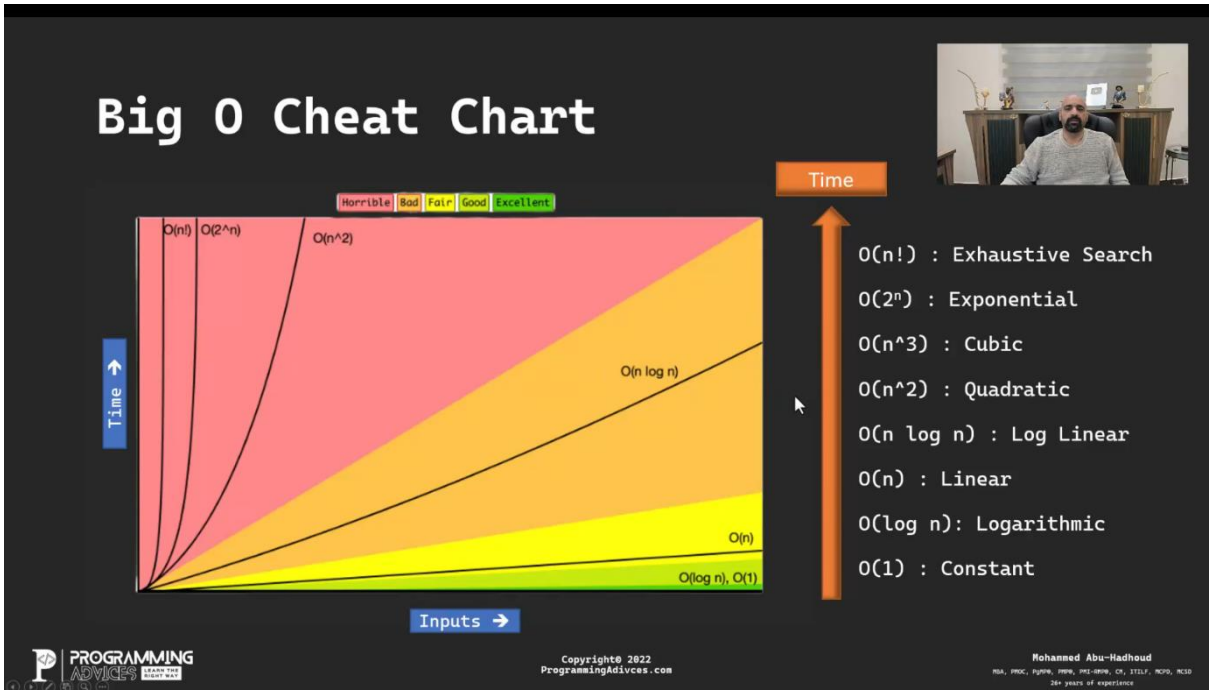
#Lesson 11 : Big O Comparison and Conclusion

Big O هو تعبير جبري في الرياضيات ليبين كيف حالة الخوارزمية ، فيقارن ما بين Input & Time

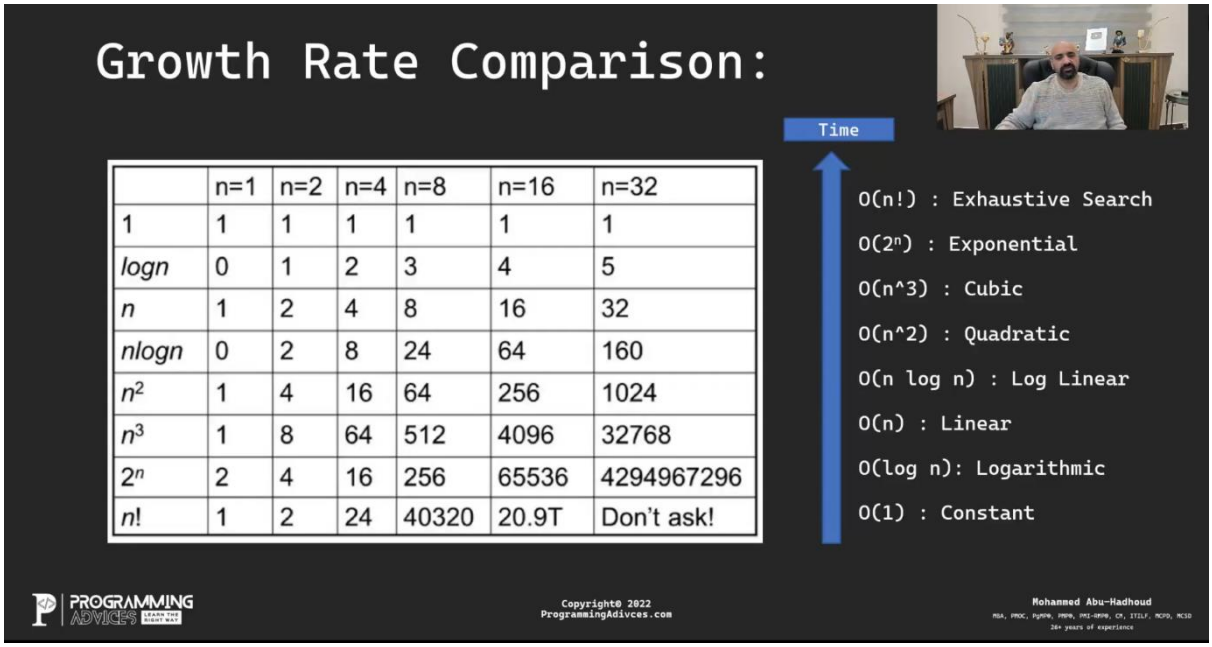
Input مرتبط ب Space في RAM كلما زاد Input زادت المساحة Space للبرنامج

الأفضل أن تكون حالة أو نوع الخوارزمية : O(1) ثم O(log n) ثم O(n) فما عدا ذلك يكون للضرورة

ترتيب أنواع Big O على حسب Time



مثال على الزمن الذي تحتاجه الخوارزمية مقارنة مع Input – كل Input بثانية لإتمامه لتقريب الفهم

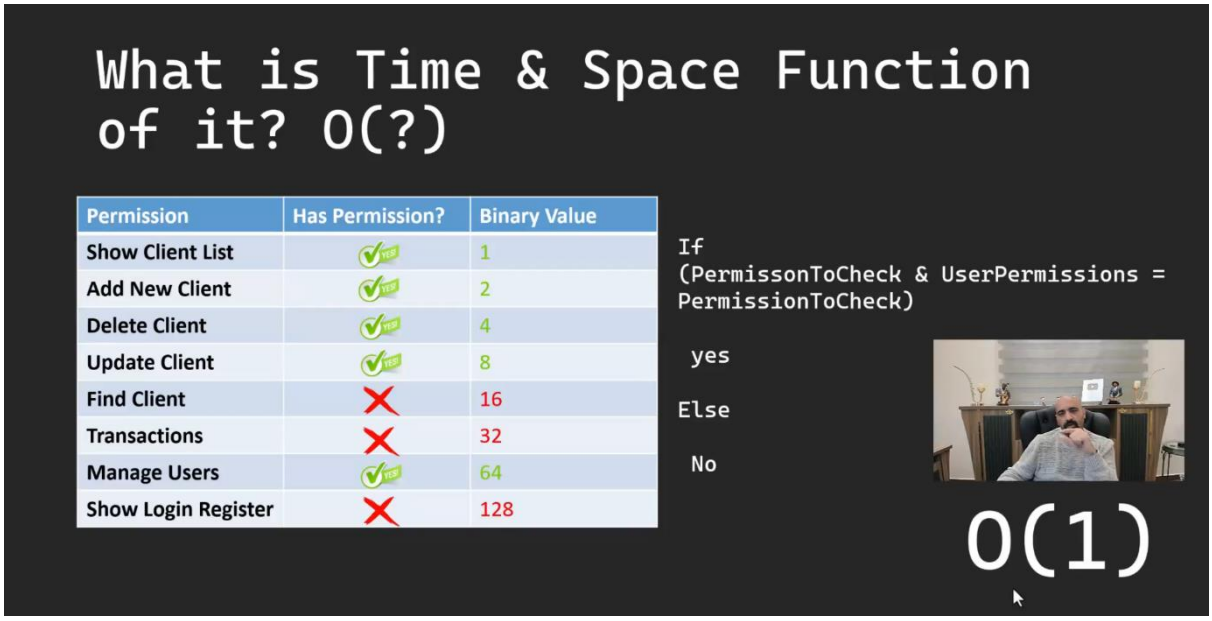


	n=1	n=2	n=4	n=8	n=16	n=32
1	1	1	1	1	1	1
logn	0	1	2	3	4	5
n	1	2	4	8	16	32
nlogn	0	2	8	24	64	160
n^2	1	4	16	64	256	1024
n^3	1	8	64	512	4096	32768
2^n	2	4	16	256	65536	4294967296
n!	1	2	24	40320	20.9T	Don't ask!

#Lesson 12 : Binary Data Structure: Real Examples

Binary هو نوع من أنواع Data Structure لأنك تستطيع تخزين بيانات برقم واحد مثل الصلاحيات – لن تجد هذا الدرس في أي محتوى آخر –

مثال للصلاحيات Permission في مشروع البنك ، مثال 79 ماذا يوجد فيه من صلاحيات للمستخدم



- 79 هو مجموع الأرقام – Binary – التي يسمح للمستخدم الوصول إليها
- 79 هو رقم له حجم صغير في الذاكرة ، ولكنه مخزن فيه كل صلاحيات للمستخدم
- 79 مثال على **Space Complexity** مساحة صغيرة ل int
  - اذا تم حلها بطريقة أخرى : فستكون لكل صلاحية مساحة في الذكرة
- 79 مثال على **Time Complexity** : للتحقق من صلاحية المستخدم بشرط واحد فقط

```
if ((Permission & this->Permissions) == Permission)
    return true;
else
    return false;
```

  - بطريقة **Binary** يكون O(1) سريع جدا
  - اذا تم حلها بطريقة أخرى : ستتحقق من الصلاحية ب for loop فيكون O(n)

#Lesson 13 : Array is One of the Most Important Data Structures

Array هي : سلسلة من المتغيرات لها نفس النوع Array (int , string , struct , Object , char)
وبما أن المتغيرات التي بداخل **Array** لها نفس النوع فهي homogeneous data structure

تستطيع الوصول الى أول عنصر في Array عن طريق Indexes = 0 (هذا في C++ يوجد لغات أخرى يبدأ Indexes فيها من 1)

وكل عنصر في Array له Address خاص به في الذاكرة

❖ هل Array لها حجم ثابت ؟

على حسب اللغة المستخدمة مثلا في لغة C يكون له حجم ثابت لا يمكنك تغييره
أما في لغة C++ يمكن أن يكون لك Dynamic Array

❖ ما هي العمليات Operations في Array ؟

- تسمح لك بالوصول الى موقع العنصر بسرعة : x[5]
- عمليات البحث والإدراج سهلة
- ينمكن الوصول الى العاصر باستخدام for loop

Array سهلة التعامل معها لذا هي من أهم أنواع Data Structure

Array يبنى عليها أنواع أخرى من Data Structure

ما هو **Time Complexity** ل Array ؟

عند تعبئة Array بهذه الطريقة x[5] = 10 تكون O(1)

عند تعبئة أو البحث عن عنصر معين في Array ب for Loop تكون O(n)

```
    int arr[10];

    for (int i = 0; i < 10 ; i++)
    {
        cin << arr[i] ;
    }
short FindNumberAlgorhim2(short arr1[10], short Number)
{
    int n = 10;
    for (int i = 0; i <= n; i++)
    {
        if (arr1[i] == Number)
        {
            return i;
        }
    }
    return -1;}
```

Array Data Structure:

Array: is a variable that can store multiple values of the same type.

Array: An array is a collection of data items stored at contiguous memory locations.

The idea is to store multiple items of the (homogeneous - **same type**) together.

Array Data Structure:

```
int x[5] = { 22, 18, 2, 55, 520 };
```

Is Array Always of Fixed Size?

Depends on they language for example: In C language, the array has a fixed size meaning once the size is given to it, it cannot be changed i.e. you can't shrink it nor can you expand it.

In C++ you can have dynamic arrays.

Operations on The Array?

- Arrays allow random access to elements. This makes accessing elements by position faster.
- Hence operation like searching, insertion, and access becomes really efficient.
- Array elements can be accessed using the loops.

Time Complexity on The Array?

- **Insertion:** We try to insert a value to a particular array index position, as the array provides random access it can be done easily using the assignment operator.

```
arr[2] = 10;
```

- Time Complexity:
  - O(1) to insert a single element
  - O(N) to insert all the array elements [where N is the size of the array]

Time Complexity on The Array?

- **Access elements in Array:** Accessing array elements become extremely important, in order to perform operations on arrays.

```
return arr[2];
```

- Time Complexity:
  - O(1) to insert a single element
  - O(N) to access all the array elements [where N is the size of the array]

Time Complexity on The Array?

- **Searching in Array:** We try to find a particular value in the array, in order to do that we need to access all the array elements and look for the particular value.

```
// searching for value 55 in the array;

Loop from i = 0 to 10:
    check if  arr[i] = 55:
        return true;
```

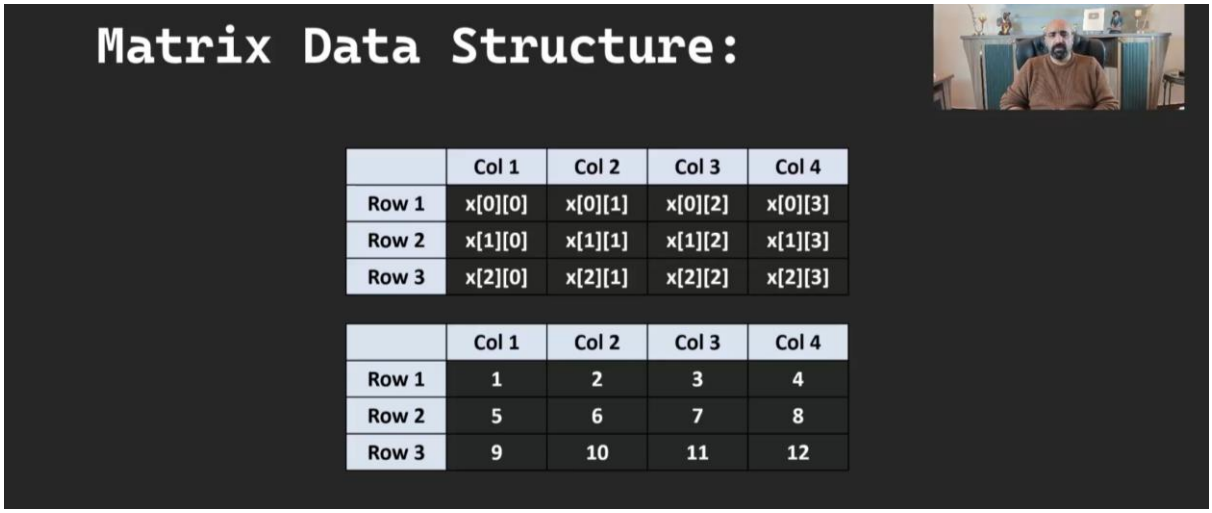
- Time Complexity:
  - O(N) [where N is the size of the array]

Applications of The Array?

- Array stores data elements of the same data type.
- Arrays are used when the size of the data set is known.
- Used in solving matrix problems.
- Applied as a lookup table in computer.
- Databases records are also implemented by the array.
- Helps in implementing sorting algorithm.
- The different variables of the same type can be saved under one name.
- Arrays can be used for CPU scheduling.
- Used to Implement other data structures like Stacks, Queues, Heaps, Hash tables, etc.

#Lesson 14 : Matrix Data Structure

Matrix هي عبارة عن Two Dimensional Array : **arr[Row] [Col]**
أو هي تمثيل لمجموعة من الأرقام مرتبة حسب ترتيب الصفوف والأعمدة



	Col 1	Col 2	Col 3	Col 4
Row 1	x[0][0]	x[0][1]	x[0][2]	x[0][3]
Row 2	x[1][0]	x[1][1]	x[1][2]	x[1][3]
Row 3	x[2][0]	x[2][1]	x[2][2]	x[2][3]

	Col 1	Col 2	Col 3	Col 4
Row 1	1	2	3	4
Row 2	5	6	7	8
Row 3	9	10	11	12

تستطيع الوصول الى أي عنصر داحل Matrix عن طريق Index مثل رقم 6 = x[1][1]

❖ ما هو **Time Complexity** ل Array ؟

عند تعبئة Matrix أو البحث فيها أو تحديثها بهذه الطريقة x[1] [0] = 20 تكون **O(1)**

عند تعبئة أو البحث عن عنصر معين في Matrix ب Two for Loop تكون **$O(n^2)$**

```
void FindNumberAlgorhimMatrix(short mat[3][4], short Number)
{
    for (int i = 0; i < 3; i++)
    {
        for (int j = 0; j < 4; j++)
        {
            if (mat[i][j] == Number)
            {
            cout << "\nIndex " << Number << " is = [" << i << "][" << j << "]\n";
                break;
            }
        }
    }
}
int main()
{
    short matrix[3][4] =
    {
        { 1,2,3,4 },
        { 5,6,7,8 },
        { 9,10,11,12 } };

    FindNumberAlgorhimMatrix(matrix, 7);
}
```

Matrix Data Structure:

Matrix: A matrix represents a collection of numbers arranged in an order of rows and columns.

Time Complexity on The Matrix?

- **Insertion:** We try to insert a value to a particular array index position, as the array provides random access it can be done easily using the assignment operator.

```
arr[2][1] = 10;
```

- Time Complexity:
  - O(1) to insert a single element
  - $O(N^2)$ to insert all the matrix elements [where N is the size of the array]

Time Complexity on The Matrix?

- **Access elements in Matrix:** Accessing Matrix elements become extremely important, in order to perform operations on arrays.

```
return arr[2][1];
```

- Time Complexity:
  - O(1) to insert a single element
  - $O(N^2)$ to access all the matrix elements [where N is the size of the array]

Time Complexity on The Array?

- **Searching in Array:** We try to find a particular value in the matrix, in order to do that we need to access all the matrix elements and look for the particular value.

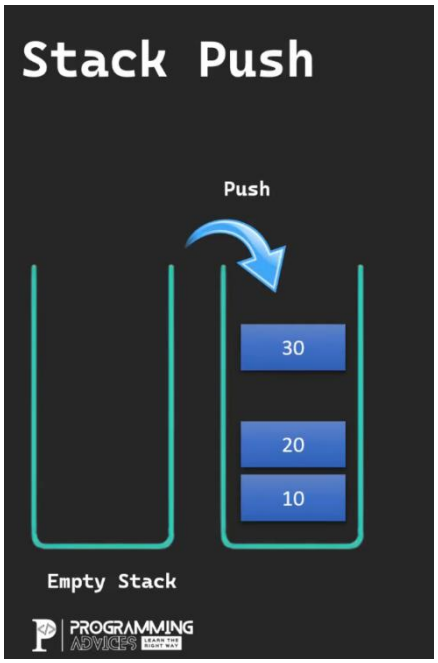
```
// searching for value 55 in the matrix;

Loop from i = 0 to 10:
Loop from j = 0 to 10:
    check if  arr[i][j] = 55:
        return true;
```

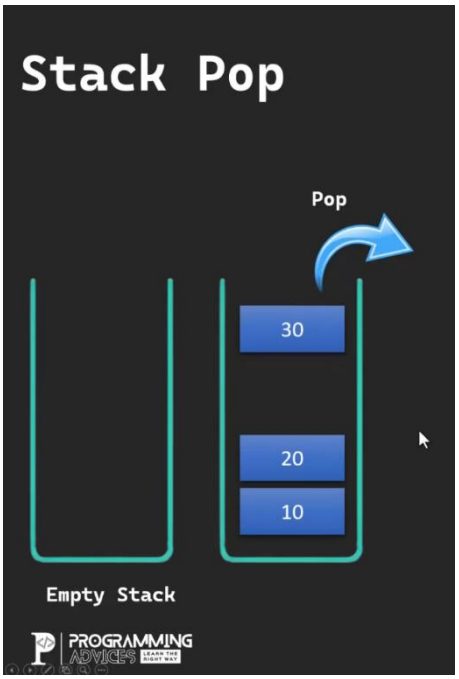
- Time Complexity:
  - $O(N^2)$ [where N is the size of the array]

#Lesson 15 : Important: What is Stack Data Structure ?

Stack Data Structure هو أول عنصر دخل Stack هو آخر عنصر يخرج من Stack ، وآخر عنصر دخل Stack هو أول عنصر يخرج من Stack (مبدأ LIFO اختصار ل : Last in First Out)

عند إضافة Push أكثر من عنصر في Stack ، لا تستطيع إخراج Pop أول عنصر تمت إضافته في Stack وإنما يخرج Pop آخر عنصر تمت إضافته





- Vector Data Structure مبني على Stack Data Structure
- Stack Data Structure مبني على Array Data Structure
- من استخدامات Stack هي Call Stack – Function – OR Call Hierarchy
  بمعنى آخر Call Stack مبنية على Stack Data Structure

#Lesson 16 : STL : Stack

يوجد في C++ مكتبة متقدمة تقدم خدمات جاهزة لتكون أسرع في البرمجة وهي : **STL**

STL اختصار ل : Standard Template Library

مثال لبعض المكتبات في **STL** : Stack , Vector

لابد من استدعاء مكتبة Stack : `#include <stack>`

طريقة استدعاء **Stack** يكون مثل Vector

```
#include <iostream>
#include <stack>

using namespace std;

int main()
{

    // create a stack of int
    stack <int> stkNumbers;

    // push into stack
    // big O = O(1)
    stkNumbers.push(10);
    stkNumbers.push(20);
    stkNumbers.push(30);
    stkNumbers.push(40);
    stkNumbers.push(50);

    // we can access the element by getting the top and popping
    // until the stack is empty
        // big O = O(1)
    cout << "\ncount=" << stkNumbers.size() << endl;

    cout << "Numbers are:\n";
    // big O = O(1)
    while (!stkNumbers.empty())
    {
        // big O = O(n) = loop
            // big O  to program = O(n)

        // print top element
        // stack ترجع آخر رقم دخل : top()
            // big O = O(1)
        cout << stkNumbers.top() << "\n";

        // pop top element from stack
            // big O = O(1)
        stkNumbers.pop();
    }
    system("pause>0");
    return 0;
]
```

Answer	Question
push()	**To add an Item to a stack you use**
pop()	**To remove the top item from stack you use**
top()	**To access the last item in stack you use**
size()	**To get the count of the items in stack you use**
empty()	**To check if the stack has items or not you use**
True	**LIFO means Last In First Out**

#Lesson 17 : Stack Swap

Stack Swap : هو تبديل Stack1 مكان Stack2 وتبديل Stack2 مكان Stack1

```cpp
#include <iostream>
#include <string>
#include <stack>
using namespace std;

int main()
{
    // stack container declaration
    stack<int> MyStack1;
    stack<int> MyStack2;

    // pushing elements into first stack
    MyStack1.push(10);
    MyStack1.push(20);
    MyStack1.push(30);
    MyStack1.push(40);

    // pushing elements into 2nd stack
    MyStack2.push(50);
    MyStack2.push(60);
    MyStack2.push(70);
    MyStack2.push(80);

    // using swap() function to swap elements of stacks
    MyStack1.swap(MyStack2);

    // printing the first stack
    cout << "MyStack1 = ";
    while (!MyStack1.empty()) {
        cout << MyStack1.top() << " ";
        MyStack1.pop();
    }

    // printing the second stack
    cout << endl << "MyStack2 = ";
    while (!MyStack2.empty()) {
        cout << MyStack2.top() << " ";
        MyStack2.pop();
    }

    system("pause>0");
    return 0;
}
```

Result

MyStack1 = 80 70 60 50

MyStack2 = 40 30 20 10

#Lesson 18 : Vector Data Structure

Vector Data Structure مبني على Stack Data Structure

Vector شبيه جدا ب Stack

والفرق بينهما

- ❖ Vector أوسع من Stack ويوجد به كثير من Method
- ❖ Vector تعامله معاملة Array مثال : `vNumber[2]`
  - تستطيع الوصول الى العناصر بشكل مباشر

```
#include <iostream>
#include <vector>

using namespace std;

int main()
{
    vector <int> vNumber;

    vNumber.push_back(10);
    vNumber.push_back(20);
    vNumber.push_back(30);
    vNumber.push_back(40);


    cout << "\nElemant 3 = " << vNumber[2] << endl; // print = 30

    system("pause>0");
    return 0;
}
```

- ❖ أما في stack لا تستطيع الوصول إلى العناصر بشكل مباشر

#Lesson 19 : Important: What is Queue Data Structure ?

Queue Data Structure هو أن أول عنصر يدخل Queue عن طريق Push وهو أول عنصر يخرج Queue عن طريق Pop (Queue تأخذ مبدأ FIFO : First In First Out)
عكس Stack (تأخذ مبدأ LIFO اختصار ل : Last in First Out)

#Lesson 20 : STL : Queue

يوجد في C++ مكتبة متقدمة تقدم خدمات جاهزة لتكون أسرع في البرمجة وهي : **STL**

STL اختصار ل : Standard Template Library
مثال لبعض المكتبات في **STL** : Vector , Stack , **Queue**

لابد من استدعاء مكتبة **Queue**: `#include <queue>`

طريقة استدعاء **Queue** يكون مثل Vector & Stack

```
#include <iostream>
#include <queue>

using namespace std;

int main()
{
    // Queue container declaration
    queue<int> MyQueue;

    // pushing elements into first stack
    MyQueue.push(10);
    MyQueue.push(20);
    MyQueue.push(30);
    MyQueue.push(40);

    cout << "\nCount: " << MyQueue.size();
    cout << "\nFront: " << MyQueue.front();
    cout << "\nBack: " << MyQueue.back() << endl;

    cout << "\nMyQueue = ";
    while (!MyQueue.empty())
    {
        cout << MyQueue.front() << " ";
        MyQueue.pop();

    }

    system("pause>0");
    return 0;
}
```

Result

Count: 4

Front: 10

Back: 40

MyQueue = 10 20 30 40

Answer	Question
True	**FIFO: means First In First Out**
front()	**If you want to print the first item in Queue, you use**
back()	**If you want to print the last item in Queue, you use**
size()	**If you want to print the count of items in Queue, you use**
push()	**If you want add item in Queue, you use**
pop()	**If you want to remove item from Queue, you use**
False	**Queue is using LIFO**
True	**Queue is using FIFO**

Queue Swap : هو تبديل Queue1 مكان Queue2 وتبديل Queue2 مكان Queue1

مثل Stack Swap

```
    // queue container declaration
    queue<int> MyQueue1;
    queue<int> MyQueue2;

    // pushing elements into first
queue
    MyQueue1.push(10);
    MyQueue1.push(20);
    MyQueue1.push(30);
    MyQueue1.push(40);

    // pushing elements into 2nd queue
    MyQueue2.push(50);
    MyQueue2.push(60);
    MyQueue2.push(70);
    MyQueue2.push(80);

    // using swap() function to swap
elements of queues
    MyQueue1.swap(MyQueue2);
```

```
    // printing the first queue
    cout << "MyQueue1 = ";
    while (!MyQueue1.empty()) {
        cout <<
MyQueue1.front() << " ";
        MyQueue1.pop();
    }

    // printing the second
queue
    cout << endl << "MyQueue2 =
";
    while (!MyQueue2.empty()) {
        cout <<
MyQueue2.front() << " ";
        MyQueue2.pop();
    }


    system("pause>0");
    return 0;
}
```

#Lesson 21 : What is Linked List ?

Linked List Data Structure هي Linear Data Structure – خطية – مثل Array

Linked List Data Structure هي سلسلة من Data تؤشر على بعض

Linked List تشبه Array ولكنها ليست Array

Linked List هي مجموعة من Nodes وكل Node ينقسم الى جزئين Data و Pointer يؤشر على Node التالية

Singly Linked List

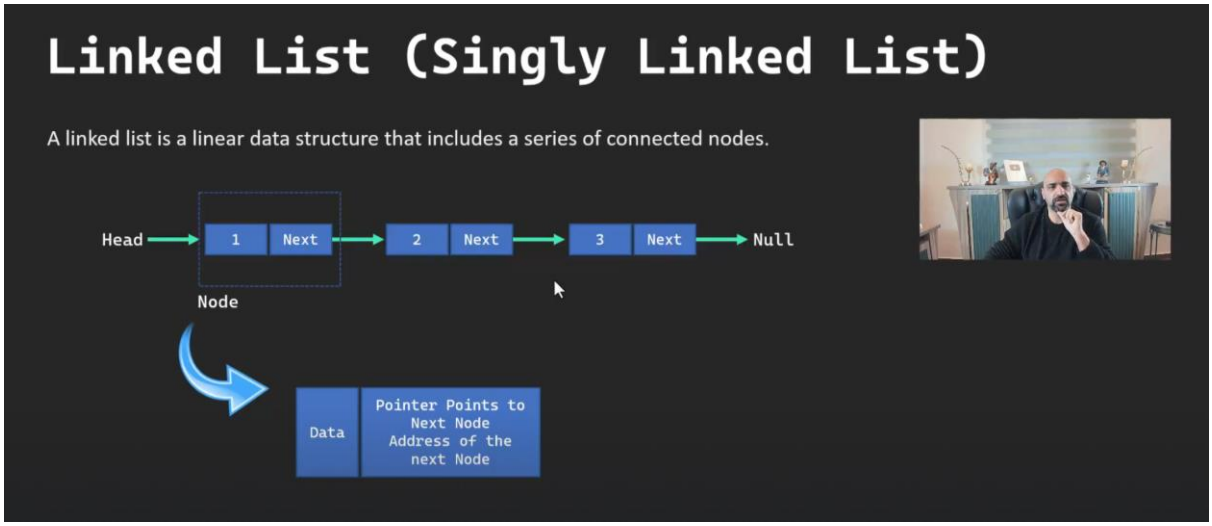
- **Head** يؤشر على مكان أول Node - البداية –
- **Null** : نهاية السلسلة – فراغة –

- **Node** ينقسم الى قسمين رئيسيتين
  - **Data** : سواء كانت ... int , string , Object , Structure أو أي نوع من Datatype
  - **Pointer** (Next : مؤشر يؤشر على Address ل Node التالية
    - و تؤشر Node الأخيرة على لا شيء – فراغ – من Address ويسمى Null

Dynamic Array : إضافة Data في Run Time من غير معرفة مسبقة بحجم Data المضافة وكانت Dynamic Array لا توجد في لغة C لهذا استخدموا Linked List

Linked List هي Data Structure تستطيع في Run Time إضافة وحذف Data وربط هذه Data بعلاقة كل Node يؤشر على Node التالية

Linked List يبنى عليها أنواع أخرى من Data Structure مثل Stack & Queue



Answer	Question
True	**A linked list is a linear data structure that includes a series of connected nodes**
True	**In singly linked list each node consists of two parts: Data and Pointer to the address of next node**
True	**Linked List is a data structure used to build other data structures like Stack, Queue , and others**
True	**Linked list allows you to add data dynamically on run time**

#Lesson 22 : Singly Linked List Implementation

Linked List هي مجموعة من Nodes وكل Node ينقسم الى جزئين Data و Pointer يؤشر على Node التالية

Node تستطيع تمثيلها ب Class و Structure

❖ مثال على Class

```
#include <iostream>
using namespace std;

// Creating a node
class Node

{
public:
    int value; // Data
    Node* next; // التالية Node ل Address يتم تخزين = Pointer
};
int main()
{
    Node* head;

    // Class Node من Object 3 إنشاء
    Node* Node1 = NULL;
    Node* Node2 = NULL;
    Node* Node3 = NULL;

    // allocate 3 nodes in the heap
    // Data & Address = Pointer موجودين في الذاكرة ولكن بدون
    Node1 = new Node();
    Node2 = new Node();
    Node3 = new Node();

    // Assign value values
    Node1->value = 1;
    Node2->value = 2;
    Node3->value = 3;

    // Connect nodes
    Node1->next = Node2; // Node1 value = 1 && Node1 * next = Address Node2
    Node2->next = Node3;  // Node2 value = 2 && Node2 * next = Address Node3
    Node3->next = NULL;   // Node3 value = 3 && Node3 * next = Address NULL
    // Linked List نهاية NULL

    // print the linked list value
    head = Node1; // head  = Node1 value = 1 && Node1 * next = Address Node2

    while (head != NULL)
    {
        cout << head->value << endl;
        head = head->next; // Node1 * next = Address Node2
        // Node2 * next = Address Node3
        // Node3 * next = Address NULL
    }
}
```

❖ مثال على Structure

```
#include <iostream>
using namespace std;

struct stNODE
{
    int value;
    stNODE* Next;
};

int main()
{
    stNODE* HEAD;

    stNODE* stNODE4 = NULL;
    stNODE* stNODE5 = NULL;
    stNODE* stNODE6 = NULL;

    stNODE4 = new stNODE();
    stNODE5 = new stNODE();
    stNODE6 = new stNODE();


    stNODE4->value = 4;
    stNODE5->value = 5;
    stNODE6->value = 6;

    stNODE4->Next = stNODE5;
    stNODE5->Next = stNODE6;
    stNODE6->Next = NULL;

    HEAD = stNODE4;
    while (HEAD != NULL)
    {
        cout << HEAD->value << endl;
        HEAD = HEAD->Next;
    }


    system("pause>0");
    return 0;
}
```

#Lesson 23 : Operations - Insert At Beginning

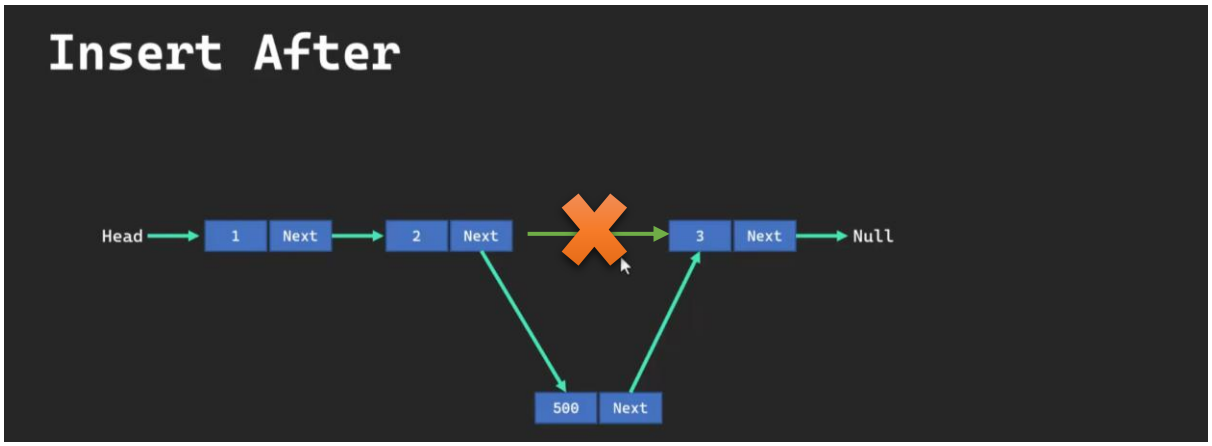
إضافة Node الى بداية أو أول Singly Linked List

```
#include <iostream>
using namespace std;

// Creating a node
class Node

{
public:
    int value; // Data
    Node* next; // التالية Node ل Address يتم تخزين = Pointer
};

void InsertAtBeginning(Node*& head, int value)
{

    // Allocate memory to a node
    Node* new_node = new Node();

    // insert the data
    new_node->value = value; // 5 --> 4 --> 3 --> 2 --> 1
    new_node->next = head;   // *5 --> *4 --> *3 --> *2 --> *1 --> NULL
    // value ل وليس Class Node ل Address *1 المقصود ب

    // Move head to new node
    head = new_node;
    // main في لتخزينها head في يتم تخزينها
}

// Print the linked list
void PrintList(Node* head)

{
    while (head != NULL)
    {
        cout << head->value << " ";
        head = head->next;
    }
}

int main()
{

    Node* head = NULL;

    InsertAtBeginning(head, 1);
    InsertAtBeginning(head, 2);
    InsertAtBeginning(head, 3);
    InsertAtBeginning(head, 4);
    InsertAtBeginning(head, 5);

    PrintList(head);
}
```

#Lesson 24 : Operations - Find

```cpp
#include <iostream>
using namespace std;

class Node
{
public:
    int value; // Data
    Node* next; // التالية Node ل Address يتم تخزين = Pointer
};
void InsertAtBeginning(Node*& head, int value)
{
    Node* new_node = new Node();

    new_node->value = value; // 5 --> 4 --> 3 --> 2 --> 1
    new_node->next = head;   // *5 --> *4 --> *3 --> *2 --> *1 --> NULL

    head = new_node;
}
void PrintList(Node* head)
{
    while (head != NULL)
    {
        cout << head->value << " ";
        head = head->next;
    }
}

Node* Find(Node* head, int Value)
{
    while (head != NULL) {

        if (head->value == Value)
            return head;

        head = head->next;
    }
    return NULL;
}
int main() {
    Node* head = NULL;

    InsertAtBeginning(head, 1);
    InsertAtBeginning(head, 2);
    InsertAtBeginning(head, 3);
    InsertAtBeginning(head, 4);
    InsertAtBeginning(head, 5);

    PrintList(head);

    Node* N1 = Find(head, 2);

    if (N1 != NULL)
        cout << "\nNode Found :-)\n";
    else
        cout << "\nNode Is not found :-(\n";
}
```

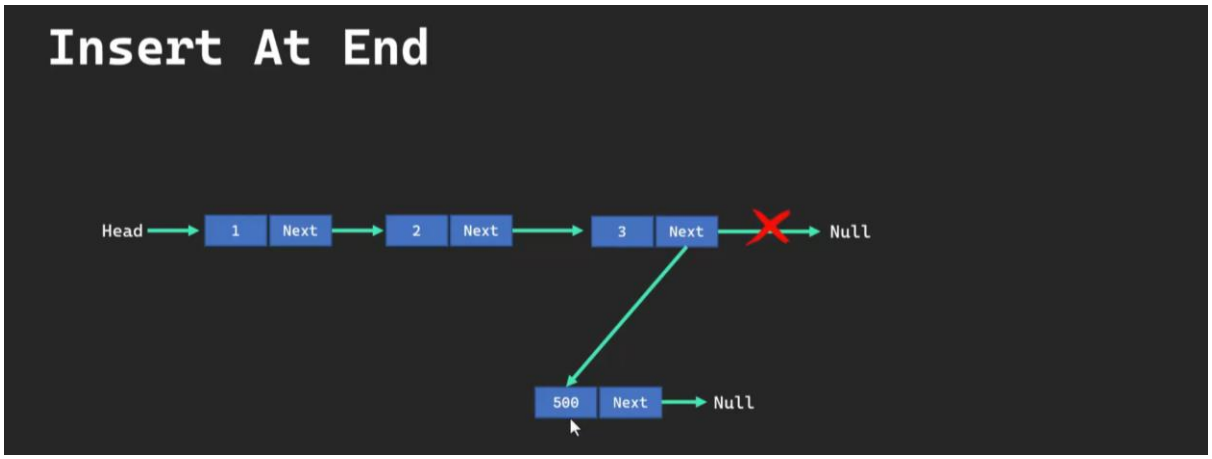
#Lesson 25 : Operations – Insert After



مثل الكود السابق + الكود التالي :

```
#include <iostream>
using namespace std;

void InsertAfter(Node* prev_node, int value)
{
    if (prev_node == NULL)
    {
        cout << "the given previous node cannot be NULL";
        return;
    }

    Node* new_node = new Node();

    new_node->value = value; // 500
    new_node->next = prev_node->next; // next address *500 = address *1
    prev_node->next = new_node; // next address *2 = address *500
    // value ل وليس Class node ل Address *1 المقصود ب
}

int main()
{
    Node* N1 = Find(head, 2);

    InsertAfter(N1, 500);

    PrintList(head);

    cout << "\n\n";

     N1 = Find(head, 500);
     InsertAfter(N1, 600);
     PrintList(head);

    system("pause>0");
    return 0;
}
```

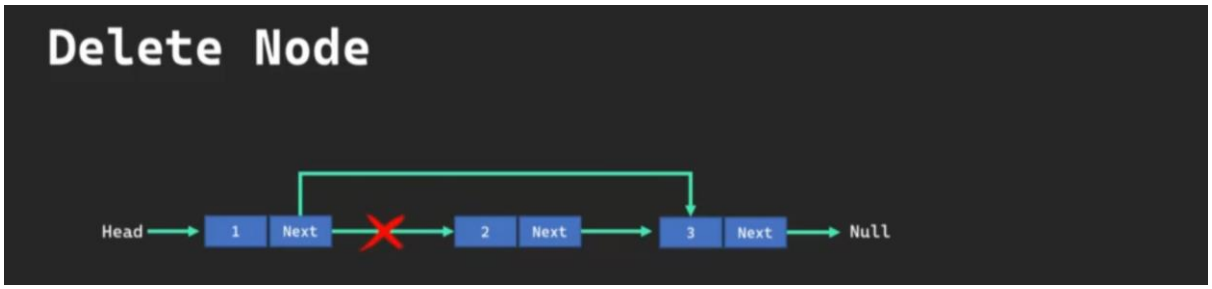
#Lesson 26 : Operations – Insert At End



مثل الكود السابق + الكود التالي :

```
#include <iostream>
using namespace std;
// Insert at the end
void InsertAtEnd(Node*& head, int Value) {

    Node* new_node = new Node();

    new_node->value = Value;
    new_node->next = NULL;

    if (head == NULL) {
        head = new_node;
        return;
    }

    Node* LastNode = head;
    while (LastNode->next != NULL)
    {
        LastNode = LastNode->next;
    }

    LastNode->next = new_node;
    return;
}
int main()
{
     Node* head = NULL;

     InsertAtEnd(head, 1);
     InsertAtEnd(head, 2);
     InsertAtEnd(head, 3);
     InsertAtBeginning(head, 0);

     InsertAtEnd(head, 9);

     PrintList(head);

    system("pause>0");
    return 0; }
```

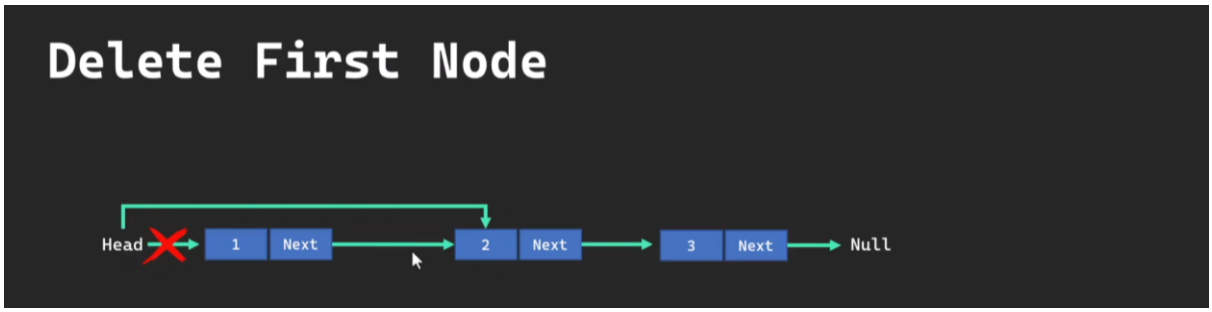
#Lesson 27 : Operations – Delete Node



مثل الكود السابق + الكود التالي :

```
#include <iostream>
using namespace std;
// Delete a node
void DeleteNode(Node*& head, int Value) {

    Node* Current = head, * Prev = head;

    if (head == NULL)
    {
        return;
    }

    if (Current->value == Value)
    {
        head = Current->next;
        delete Current;//free from memory
        return;
    }

    // Find the key to be deleted
    while (Current != NULL && Current->value != Value)
    {
        Prev = Current;
        Current = Current->next;
    }

    // If the value is not present
    if (Current == NULL)
        return;

    // Remove the node
    Prev->next = Current->next;
    delete Current;//free from memory

}
int main()
{
     DeleteNode(head, 2);

     PrintList(head);

     system("pause>0");
    return 0;
}
```

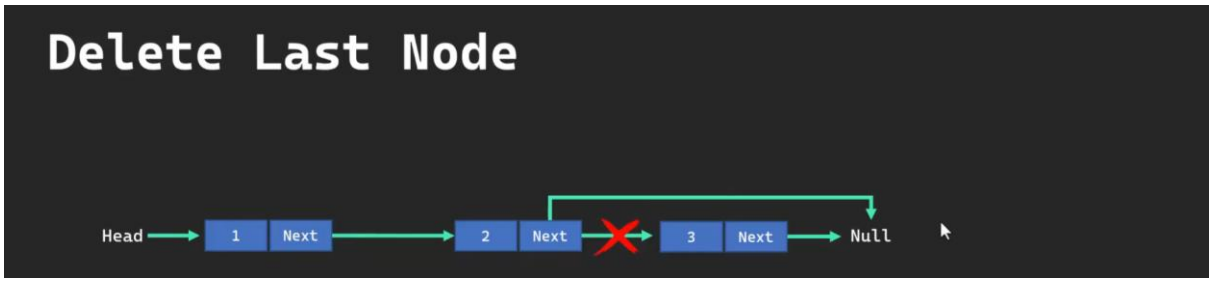
#Lesson 28 : Operations – Delete First Node



مثل الكود السابق + الكود التالي :

```
#include <iostream>
using namespace std;
// Delete First Node
void DeleteFirstNode(Node*& head) {

    Node* Current = head;

    if (head == NULL)
    {
        return;
    }

    head = Current->next;
    delete Current;//free from memory
    return;

}

int main()
{
    Node* head = NULL;

     InsertAtEnd(head, 1);
     InsertAtEnd(head, 2);
     InsertAtEnd(head, 3);
     InsertAtEnd(head, 4);
     InsertAtEnd(head, 5);
     InsertAtEnd(head, 6);
     PrintList(head);


     DeleteFirstNode(head);

     PrintList(head);



     system("pause>0");
    return 0;
}
```

#Lesson 29 : Operations – Delete Last Node



مثل الكود السابق + الكود التالي :

```
#include <iostream>
using namespace std;
// Delete Last node
void DeleteLastNode(Node*& head)
{
    Node* Current = head, * Prev = head;

    if (head == NULL)
    {
        return;
    }

    if (Current->next == NULL) {
        head = NULL;
        delete Current;//free from memory
        return;
    }

    // Find the key to be deleted
    while (Current != NULL && Current->next != NULL) {
        Prev = Current;
        Current = Current->next;
    }

    // Remove the node
    Prev->next = NULL;
    delete Current;//free from memory
}


int main()
{
     DeleteLastNode(head);

     PrintList(head);



     system("pause>0");
    return 0;
}
```

All Operations – LinkedList (Singly Linked List)

```
#include <iostream>
using namespace std;

// Creating a node
class Node
{
public:
    int value; // Data
    Node* next; // التالية Node ل Address يتم تخزين = Pointer
};

void InsertAtBeginning(Node*& head, int value)
{

    // Allocate memory to a node
    Node* new_node = new Node();

    // insert the data
    new_node->value = value; // 5 --> 4 --> 3 --> 2 --> 1
    new_node->next = head;   // *5 --> *4 --> *3 --> *2 --> *1 --> NULL
    // value ل وليس Class Node ل Address *1 المقصود ب

    // Move head to new node
    head = new_node;
    // main في لتخزينها head في يتم تخزينها
}

Node* Find(Node* head, int Value)
{

    while (head != NULL) {

        if (head->value == Value)
            return head;


        head = head->next;
    }

    return NULL;


}


// Print the linked list
void PrintList(Node* head)

{
    while (head != NULL)
    {
        cout << head->value << " ";
        head = head->next;
    }
}
```

```
// Insert a node after a node
void InsertAfter(Node* prev_node, int value)
{
    if (prev_node == NULL)
    {
        cout << "the given previous node cannot be NULL";
        return;
    }

    Node* new_node = new Node();

    new_node->value = value; // 500
    new_node->next = prev_node->next; // next address *500 = address *1
    prev_node->next = new_node; // next address *2 = address *500
    // value ليس ل Class node ل Address *1 المقصود ب
}

// Insert at the end
void InsertAtEnd(Node*& head, int Value) {

    Node* new_node = new Node();

    new_node->value = Value;
    new_node->next = NULL;

    if (head == NULL) {
        head = new_node;
        return;
    }

    Node* LastNode = head;
    while (LastNode->next != NULL) // NULL !=  next على * تؤشر لا الأخيرة Node هل
    {
        LastNode = LastNode->next;
    }

    LastNode->next = new_node;
    return;
}


// Delete a node
void DeleteNode(Node*& head, int Value) {

    Node* Current = head, * Prev = head;

    if (head == NULL)
    {
        return;
    }

    if (Current->value == Value)
    {
        head = Current->next;
        delete Current;//free from memory
        return;
    }
```

```
    // Find the key to be deleted
    while (Current != NULL && Current->value != Value)
    {
        Prev = Current;
        Current = Current->next;
    }

    // If the value is not present
    if (Current == NULL)
        return;

    // Remove the node
    Prev->next = Current->next;
    delete Current;//free from memory

}

// Delete First Node
void DeleteFirstNode(Node*& head) {

    Node* Current = head;

    if (head == NULL)
    {
        return;
    }

    head = Current->next;
    delete Current;//free from memory
    return;

}

// Delete Last node
void DeleteLastNode(Node*& head)
{
    Node* Current = head, * Prev = head;

    if (head == NULL)
    {
        return;
    }

    if (Current->next == NULL) // NULL ==  next على * تؤشر  الأخيرة Node هل
    {
        head = NULL;
        delete Current;//free from memory
        return;
    }

    // Find the key to be deleted
    while (Current != NULL && Current->next != NULL)
// NULL !=  next على * تؤشر لا الأخيرة Node هل
    {
        Prev = Current;
        Current = Current->next;
    }

    // Remove the node
    Prev->next = NULL;
    delete Current;//free from memory
}
```

```
// Print the linked list
void PrintList(Node* head)

{
    while (head != NULL)
    {
        cout << head->value << " ";
        head = head->next;
    }
    cout << "\n";

}

int main()
{

    Node* head = NULL;

    //==============================

    InsertAtBeginning(head, 1);
    InsertAtBeginning(head, 2);
    InsertAtBeginning(head, 3);
    InsertAtBeginning(head, 4);
    InsertAtBeginning(head, 5);

    cout << "\n Insert At Beginning : \n";
    PrintList(head);

    //==============================

    Node* N1 = Find(head, 2);

    if (N1 != NULL)
        cout << "\nNode Found :-)\n";
    else
        cout << "\nNode Is not found :-(\n";

    //==============================

    cout << "\n Insert After : \n";

    InsertAfter(N1, 500);

    PrintList(head);


     N1 = Find(head, 500);
     InsertAfter(N1, 600);
     PrintList(head);


     //==============================

     cout << "\n Insert At End : \n";

     InsertAtEnd(head, 9);

     PrintList(head);

     //==============================
```

Result

: Insert At Beginning

1 2 3 4 5

(-: Node Found

: Insert After

1 500 2 3 4 5

1 600 500 2 3 4 5

: Insert At End

9 1 600 500 2 3 4 5

: Delete Node

6 5 4 3 2 1

6 5 4 3 1

: Delete First Node

6 5 4 3

: Delete Last Node

5 4 3

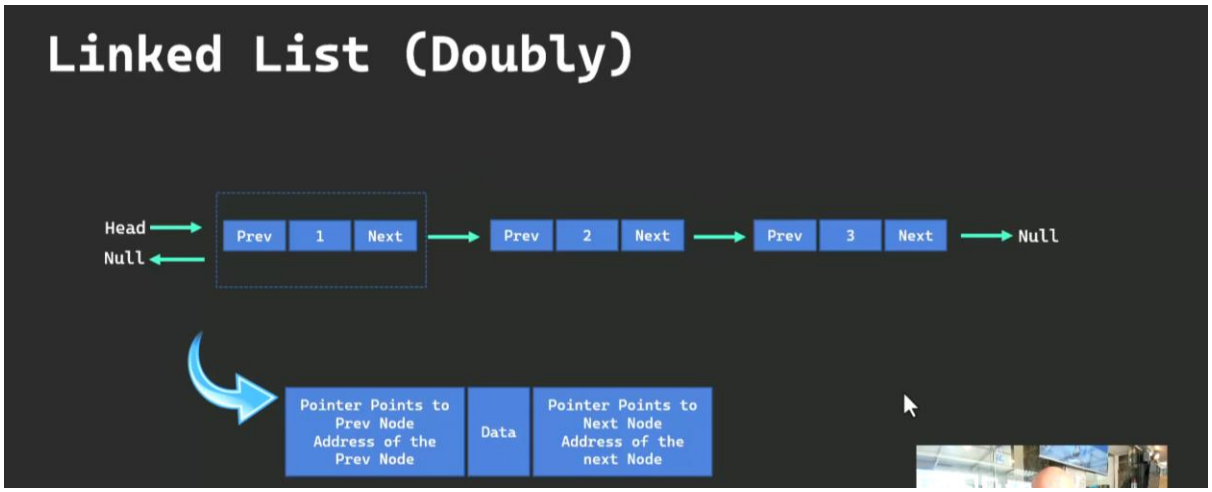
```
    cout << "\n Delete Node : \n";

    head = NULL;

    InsertAtEnd(head, 1);
    InsertAtEnd(head, 2);
    InsertAtEnd(head, 3);
    InsertAtEnd(head, 4);
    InsertAtEnd(head, 5);
    InsertAtEnd(head, 6);
    PrintList(head);


    DeleteNode(head, 2);

    PrintList(head);

    //==============================


    cout << "\n Delete First Node : \n";

    DeleteFirstNode(head);

    PrintList(head);


    //==============================

    cout << "\n Delete Last Node : \n";

    DeleteLastNode(head);

    PrintList(head);

    //==============================


    system("pause>0");
   return 0;
}
```

#Lesson 30 : What is Doubly Linked List ?

الفرق بين Singly Linked List و Doubly Linked List ؟

- ❖ Singly Linked List هي مجموعة من Nodes وكل Node ينقسم الى جزئين Data و Pointer يؤشر على Node التالية

- ❖ **Doubly Linked List** هي مجموعة من Nodes وكل Node ينقسم الى ثلاثة أجزاء
  - **Data** : سواء كانت ... Structure , Object , string , int أو أي نوع من Datatype
  - (**Pointer**) Next : مؤشر يؤشر على Address ل Node التالية
    - و تؤشر Node الأخيرة على لا شيء – فراغ – من Address ويسمى Null
  - (**Pointer**) Previous : مؤشر يؤشر على Address ل Node السابقة
- ❖ **وتمتاز** عن Singly Linked List : بأنها تستطيع المرور على العناصر فيها من طريقين
  - من البداية Head مثل Singly Linked List
  - ومن النهاية NULL



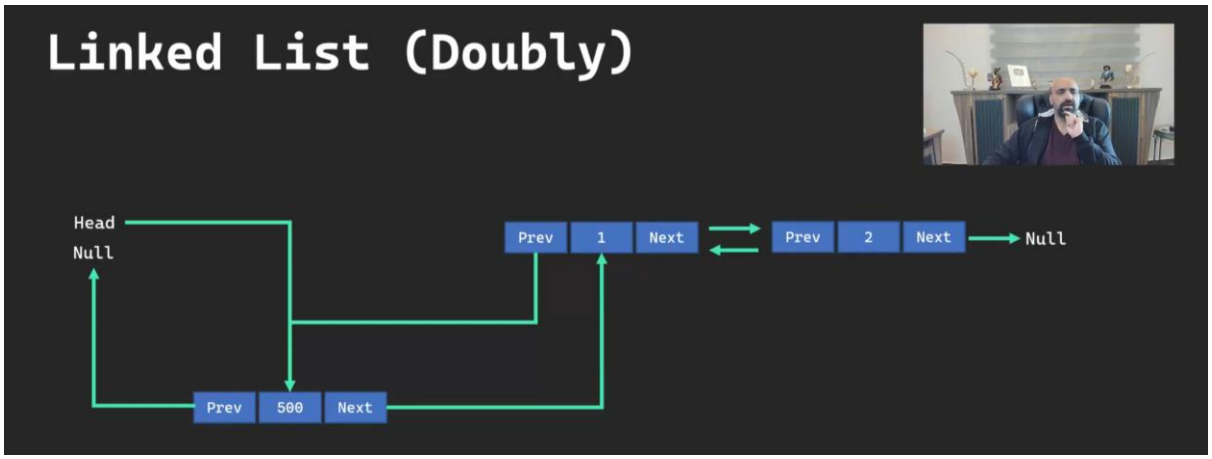
Answer	Question
True	**In Doubly Linked List Each node consists of a data value, a pointer to the next node, and a pointer to the previous node**
True	**In Doubly Linked List: Traversal can occur in both ways**
False	**In singly linked list: Traversal can occur in both ways**
True	**In Doubly Linked List: It requires more space because of an extra pointer**

#Lesson 31 : Doubly Linked List Implementation

في **Doubly Linked List** كل Node تؤشر على Node التالية next، و تؤشر على Node السابقة prev

```
#include <iostream>
using namespace std;

// Creating a node
class Node

{
public:
    int value; // Data
    Node* next; // التالية Node ل Address يتم تخزين = Pointer
    Node* prev;
};

int main()
{
    Node* head;

    Node* Node1 = NULL;
    Node* Node2 = NULL;
    Node* Node3 = NULL;

    // allocate 3 nodes in the heap
    Node1 = new Node();
    Node2 = new Node();
    Node3 = new Node();

    // Assign value values
    Node1->value = 1;
    Node2->value = 2;
    Node3->value = 3;

    // Connect nodes
    Node1->next = Node2;
    Node1->prev = NULL;

    Node2->next = Node3;
    Node2->prev = Node1;

    Node3->next = NULL;
    Node3->prev = Node2;

    // print the linked list value
    head = Node1;


    while (head != NULL) {
        cout << head->value << endl;
        head = head->next;
    }

     system("pause>0");

    return 0;
}
```

#Lesson 32 : Operations -Insert At Beginning

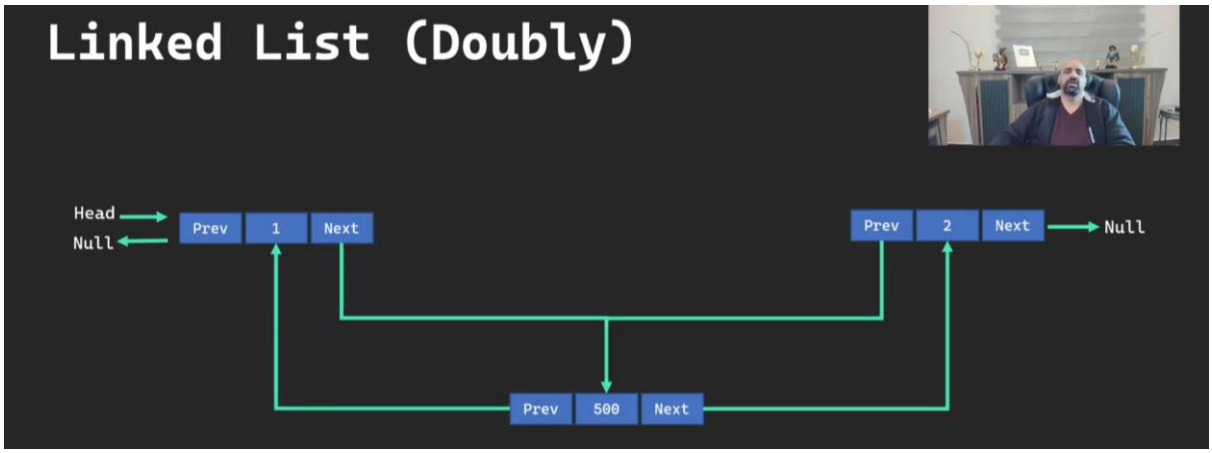



```
#include <iostream>
using namespace std;

void InsertAtBeginning(Node*& head, int value) {


    /*
        1-Create a new node with the desired value.
        2-Set the next pointer of the new node to the current head of the list.
        3-Set the previous pointer of the current head to the new node.
        4-Set the new node as the new head of the list.
    */

    Node* newNode = new Node();
    newNode->value = value;
    newNode->next = head;
    newNode->prev = NULL;

    if (head != NULL) {
        head->prev = newNode;
    }
    head = newNode;
}

int main()
{
    Node* head = NULL;

    InsertAtBeginning(head, 5);
    InsertAtBeginning(head, 4);
    InsertAtBeginning(head, 3);
    InsertAtBeginning(head, 2);
    InsertAtBeginning(head, 1);

    cout << "\nLinked List Content:\n";
    PrintList(head);
    PrintListDetails(head);

    system("pause>0");
    return 0;
}
```

#Lesson 33 : Operations – Find Node

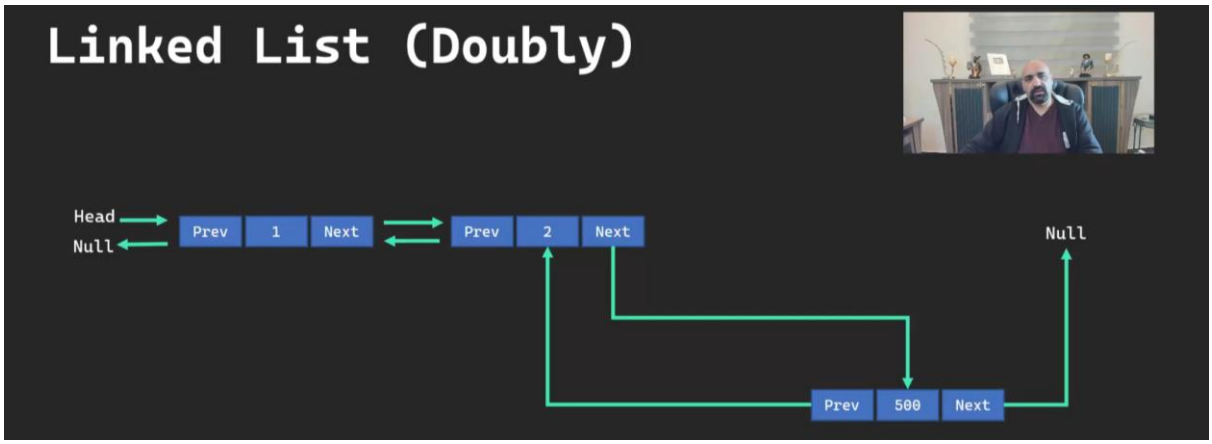
```
#include <iostream>
using namespace std;

Node* Find(Node* head, int Value)
{

    while (head != NULL) {

        if (head->value == Value)
            return head;


        head = head->next;
    }

    return NULL;


}

// Print the linked list
void PrintList(Node* head)

{
    cout << "NULL <--> ";
    while (head != NULL) {
        cout << head->value << " <--> ";
        head = head->next;
    }
    cout << "NULL";

}

int main()
{
    Node* head = NULL;

    InsertAtBeginning(head, 5);
    InsertAtBeginning(head, 4);
    InsertAtBeginning(head, 3);
    InsertAtBeginning(head, 2);
    InsertAtBeginning(head, 1);

    PrintList(head);

    Node* N1 = Find(head, 2);

    if (N1 != NULL)
        cout << "\nNode Found :-)\n";
    else
        cout << "\nNode Is not found :-(\n";



    system("pause>0");
    return 0;
}
```

#Lesson 34 : Operations – Insert After




```
#include <iostream>
using namespace std;

void InsertAfter(Node* current, int value) {


    /*  1 - Create a new node with the desired value.
         2-Set the next pointer of the new node to the next node of the current
node.
         3-Set the previous pointer of the new node to the current node.
         4-Set the next pointer of the current node to the new node.
         5-Set the previous pointer of the next node to the new node(if it
exists).
    */

    Node* newNode = new Node();
    newNode->value = value;
    newNode->next = current->next;
    newNode->prev = current;

    if (current->next != NULL) {
        current->next->prev = newNode;//current = 2 next --> 3 prev <-- (2) = 500
    }
    current->next = newNode;
}

int main()
{
    cout << "\n Insert After : \n";

     N1 = Find(head, 2);

    InsertAfter(N1, 500);

    cout << "\n\n\nLinked List Content after InsertAfter:\n";
    PrintList(head);
    PrintListDetails(head);

    system("pause>0");
    return 0;
}
```

#Lesson 35 : Operations – Insert At End

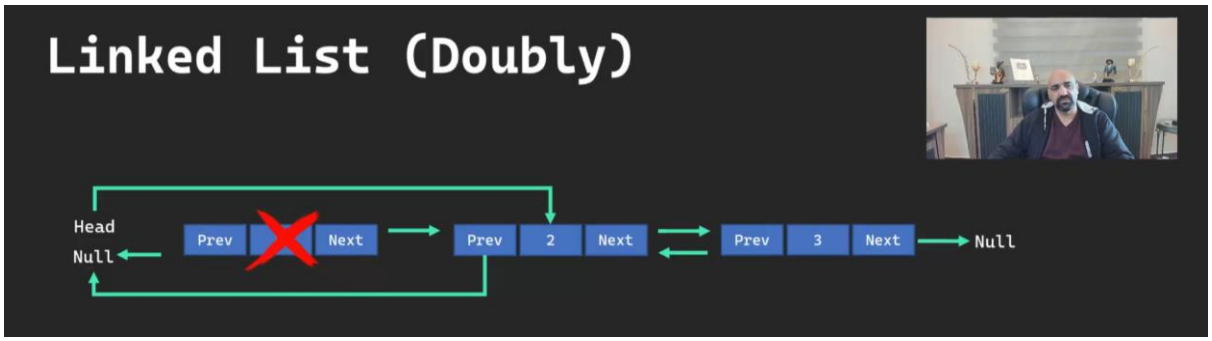



```
#include <iostream>
using namespace std;
void InsertAtEnd(Node* head, int value) {

    /*
        1-Create a new node with the desired value.
        2-Traverse the list to find the last node.
        3-Set the next pointer of the last node to the new node.
        4-Set the previous pointer of the new node to the last node.
    */

    Node* newNode = new Node();
    newNode->value = value;
    newNode->next = NULL;

    if (head == NULL) {
        newNode->prev = NULL;
        head = newNode;
    }
    else {
        Node* current = head;
        while (current->next != NULL) {
            current = current->next;
        }
        current->next = newNode;
        newNode->prev = current;
    }
}
int main()
{
    cout << "\nLinked List Content:\n";
    PrintList(head);
    PrintListDetails(head);

    InsertAtEnd(head, 500);

    cout << "\n\n\nLinked List Content after InsertAtEnd:\n";
    PrintList(head);
    PrintListDetails(head);
    system("pause>0");
    return 0;
}
```

#Lesson 36 : Operations – Delete Node

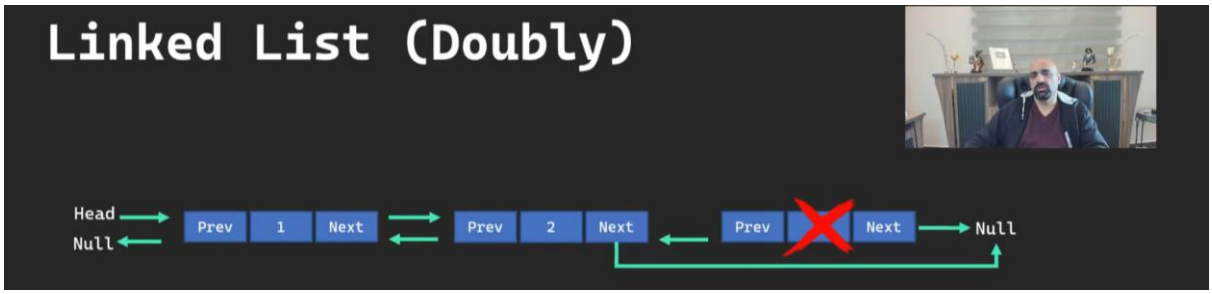

```
#include <iostream>
using namespace std;

void DeleteNode(Node*& head, Node*& NodeToDelete) {


    /*
        1-Set the next pointer of the previous node to the next pointer of the
current node.
        2-Set the previous pointer of the next node to the previous pointer of
the current node.
        3-Delete the current node.
    */
    if (head == NULL || NodeToDelete == NULL) {
        return;
    }
    if (head == NodeToDelete) {
        head = NodeToDelete->next;
    }
    if (NodeToDelete->next != NULL) {
        NodeToDelete->next->prev = NodeToDelete->prev;
    }
    if (NodeToDelete->prev != NULL) {
        NodeToDelete->prev->next = NodeToDelete->next;
    }
    delete NodeToDelete;
}

int main()
{
    cout << "\nLinked List Content:\n";
    PrintList(head);
    PrintListDetails(head);

    //Traverse the list to find the node to be deleted.
    N1 = Find(head, 4);

    DeleteNode(head, N1);

    cout << "\n\n\nLinked List Content after delete:\n";
    PrintList(head);
    PrintListDetails(head);

    system("pause>0");
    return 0;
}
```

#Lesson 37 : Operations – Delete First Node




```cpp
#include <iostream>
using namespace std;

void DeleteFirstNode(Node*& head) {

    /*
        1-Store a reference to the head node in a temporary variable.
        2-Update the head pointer to point to the next node in the list.
        3-Set the previous pointer of the new head to NULL.
        4-Delete the temporary reference to the old head node.
    */

    if (head == NULL) {
        return;
    }
    Node* temp = head;
    head = head->next;

    if (head != NULL) {
        head->prev = NULL;
    }
    delete temp;
}


int main()
{
    DeleteFirstNode(head);


    cout << "\n\n\nLinked List Content after delete:\n";
    PrintList(head);
    PrintListDetails(head);

    system("pause>0");
    return 0;
}
```

#Lesson 38 : Operations – Delete Last Node




```
#include <iostream>
using namespace std;

void DeleteLastNode(Node*& head) {

    /*
        1-Traverse the list to find the last node.
        2-Set the next pointer of the second-to-last node to NULL.
        3-Delete the last node.
    */

    if (head == NULL) {
        return;
    }
    if (head->next == NULL) {
        delete head;
        head = NULL;
        return;
    }
    Node* current = head;
    while (current->next->next != NULL) {
        current = current->next;
    }
    Node* temp = current->next;
    current->next = NULL;
    delete temp;
}


int main()
{
    cout << "\nLinked List Content:\n";
    PrintList(head);
    PrintListDetails(head);

    DeleteLastNode(head);

    cout << "\n\n\nLinked List Content after delete last node:\n";
    PrintList(head);
    PrintListDetails(head);


    system("pause>0");
    return 0;
}
```

All Operations – Doubly Linked List

```
#include <iostream>
using namespace std;

void InsertAtBeginning(Node*& head, int value) {


    /*
        1-Create a new node with the desired value.
        2-Set the next pointer of the new node to the current head of the list.
        3-Set the previous pointer of the current head to the new node.
        4-Set the new node as the new head of the list.
    */

    Node* newNode = new Node();
    newNode->value = value;
    newNode->next = head;
    newNode->prev = NULL;

    if (head != NULL) {
        head->prev = newNode;
    }
    head = newNode;
}

void PrintNodeDetails(Node* head)
{

    if (head->prev != NULL)
        cout << head->prev->value;
    else
        cout << "NULL";

    cout << " <--> " << head->value << " <--> ";

    if (head->next != NULL)
        cout << head->next->value << "\n";
    else
        cout << "NULL";

}

Node* Find(Node* head, int Value)
{

    while (head != NULL) {

        if (head->value == Value)
            return head;


        head = head->next;
    }

    return NULL;


}
```

Result

:Linked List Content

NULL <--> 1 <--> 2 <--> 3 <--> 4 <--> 5 <--> NULL

NULL <--> 1 <--> 2

3 <--> 2 <--> 1

4 <--> 3 <--> 2

5 <--> 4 <--> 3

4

NULL <--> 5 <-->

(-: Node Found

: Insert After

:Linked List Content after Insert After

NULL <--> 1 <--> 2 <--> 500 <--> 3 <--> 4 <--> 5 <--> NULL

NULL <--> 1 <--> 2

500 <--> 2 <--> 1

3 <--> 500 <--> 2

4 <--> 3 <--> 500

5 <--> 4 <--> 3

NULL <--> 5 <--> 4

```
// Print the linked list
void PrintListDetails(Node* head)

{
    cout << "\n\n";
    while (head != NULL) {
        PrintNodeDetails(head);
        head = head->next;
    }
}


// Print the linked list
void PrintList(Node* head)

{
    cout << "NULL <--> ";
    while (head != NULL) {
        cout << head->value << " <--> ";
        head = head->next;
    }
    cout << "NULL";

}
```

:Linked List Content after Insert at End

NULL <--> 1 <--> 2 <--> 500 <--> 3 <--> 4 <--> 5 <--> 500 <--> NULL

NULL <--> 1 <--> 2

500 <--> 2 <--> 1

3 <--> 500 <--> 2

4 <--> 3 <--> 500

5 <--> 4 <--> 3

500 <--> 5 <--> 4

NULL <--> 500 <--> 5

```
void InsertAfter(Node* current, int value) {


    /*  1 - Create a new node with the desired value.
         2-Set the next pointer of the new node to the next node of the current
node.
         3-Set the previous pointer of the new node to the current node.
         4-Set the next pointer of the current node to the new node.
         5-Set the previous pointer of the next node to the new node(if it
exists).
    */

    Node* newNode = new Node();
    newNode->value = value;
    newNode->next = current->next;
    newNode->prev = current;

    if (current->next != NULL) {
        current->next->prev = newNode;//current = 2 next --> 3 prev <-- (2) = 500
    }
    current->next = newNode;
}


void InsertAtEnd(Node* head, int value) {

    /*
        1-Create a new node with the desired value.
        2-Traverse the list to find the last node.
        3-Set the next pointer of the last node to the new node.
        4-Set the previous pointer of the new node to the last node.
    */

    Node* newNode = new Node();
    newNode->value = value;
    newNode->next = NULL;
```

```
    if (head == NULL) {
        newNode->prev = NULL;
        head = newNode;
    }
    else {
        Node* current = head;
        while (current->next != NULL) {
            current = current->next;
        }
        current->next = newNode;
        newNode->prev = current;
    }
}

void DeleteNode(Node*& head, Node*& NodeToDelete) {

    /*
        1-Set the next pointer of the previous node to the next pointer of the
current node.
        2-Set the previous pointer of the next node to the previous pointer of
the current node.
        3-Delete the current node.
    */
    if (head == NULL || NodeToDelete == NULL) {
        return;
    }
    if (head == NodeToDelete) {
        head = NodeToDelete->next;
    }
    if (NodeToDelete->next != NULL) {
        NodeToDelete->next->prev = NodeToDelete->prev;
    }
    if (NodeToDelete->prev != NULL) {
        NodeToDelete->prev->next = NodeToDelete->next;
    }
    delete NodeToDelete;
}

void DeleteFirstNode(Node*& head) {

    /*
        1-Store a reference to the head node in a temporary variable.
        2-Update the head pointer to point to the next node in the list.
        3-Set the previous pointer of the new head to NULL.
        4-Delete the temporary reference to the old head node.
    */

    if (head == NULL) {
        return;
    }
    Node* temp = head;
    head = head->next;

    if (head != NULL) {
        head->prev = NULL;
    }
    delete temp;
}
```

:Linked List Content after delete

NULL <--> 1 <--> 2 <--> 500 <--> 3 <--> 5 <--> 500 <--> NULL

NULL <--> 1 <--> 2

500 <--> 2 <--> 1

3 <--> 500 <--> 2

5 <--> 3 <--> 500

500 <--> 5 <--> 3

NULL <--> 500 <--> 5

```
void DeleteLastNode(Node*& head) {

    /*
        1-Traverse the list to find the last node.
        2-Set the next pointer of the second-to-last node to NULL.
        3-Delete the last node.
    */

    if (head == NULL) {
        return;
    }
    if (head->next == NULL) {
        delete head;
        head = NULL;
        return;
    }
    Node* current = head;
    while (current->next->next != NULL) {
        current = current->next;
    }
    Node* temp = current->next;
    current->next = NULL;
    delete temp;
}


int main()
{
    Node* head = NULL;

    InsertAtBeginning(head, 5);
    InsertAtBeginning(head, 4);
    InsertAtBeginning(head, 3);
    InsertAtBeginning(head, 2);
    InsertAtBeginning(head, 1);

    cout << "\nLinked List Content:\n";
    PrintList(head);
    PrintListDetails(head);

    //==============================

    Node* N1 = Find(head, 2);

    if (N1 != NULL)
        cout << "\nNode Found :-)\n";
    else
        cout << "\nNode Is not found :-(\n";


    //==============================

    cout << "\n Insert After : \n";

     N1 = Find(head, 2);

    InsertAfter(N1, 500);

    cout << "\n\nLinked List Content after Insert After:\n";
    PrintList(head);
    PrintListDetails(head);

    //==============================
```

:Linked List Content after First delete

NULL <--> 2 <--> 500 <--> 3 <--> 5 <--> 500 <--> NULL

NULL <--> 2 <--> 500

3 <--> 500 <--> 2

5 <--> 3 <--> 500

500 <--> 5 <--> 3

NULL <--> 500 <--> 5

:Linked List Content after delete last node

NULL <--> 2 <--> 500 <--> 3 <--> 5 <--> NULL

NULL <--> 2 <--> 500

3 <--> 500 <--> 2

5 <--> 3 <--> 500

NULL <--> 5 <--> 3

```
    cout << "\nLinked List Content:\n";
    PrintList(head);
    PrintListDetails(head);

    InsertAtEnd(head, 500);

    cout << "\n\n\nLinked List Content after InsertAtEnd:\n";
    PrintList(head);
    PrintListDetails(head);

    //==============================


    cout << "\nLinked List Content:\n";
    PrintList(head);
    PrintListDetails(head);

    //Traverse the list to find the node to be deleted.
    N1 = Find(head, 4);

    DeleteNode(head, N1);

    cout << "\n\n\nLinked List Content after delete:\n";
    PrintList(head);
    PrintListDetails(head);

    //==============================

    cout << "\nLinked List Content:\n";
    PrintList(head);
    PrintListDetails(head);

    DeleteFirstNode(head);


    cout << "\n\n\nLinked List Content after First delete:\n";
    PrintList(head);
    PrintListDetails(head);

    //==============================


    cout << "\nLinked List Content:\n";
    PrintList(head);
    PrintListDetails(head);

    DeleteLastNode(head);

    cout << "\n\n\nLinked List Content after delete last node:\n";
    PrintList(head);
    PrintListDetails(head);

    //==============================




    system("pause>0");
    return 0;
}
```

#Lesson 39 : What is Circular Linked List ?

أنواع Linked List

١. Singly Linked List
٢. Doubly Linked List
٣. **Circular** Singly Linked List
٤. **Circular** Doubly Linked List

Circular : دائرة لا نهائية - تكرر Linked List ب Loop لانهائي

١. عند تحويل Singly Linked List – العادية – الى **Circular** يتم عمل التالي :
   - ❖ تحويل مؤشر next Node الأخيرة التي كانت = NULL الى Node الأولى

٢. عند تحويل Doubly Linked List – العادية – الى **Circular** يتم عمل التالي :
   - ❖ تحويل مؤشر next Node الأخيرة التي كانت = NULL الى Node الأولى
   - ❖ تحويل مؤشر prev Node الأولى التي كانت = NULL الى Node الأخيرة

Circular Linked List مفيدة في بعض التطبيقات مثل

- ❖ **Buffer** : مثال تقريبي : أن buffer يستطيع تخزين 100 عنصر ، وعند امتلاء buffer يتم إلغاء أول عنصر وتخزين عنصر جديد مكانه مبدأ (FIFO) . (يكون Circular buffer)
- ❖ عند تشغيل البرنامج في C++ تظهر شاشة سوداء وتسمى output screen أو **console** ، النتائج التي تظهر هي من buffer – البيانات المخزنة في buffer –
  - مثال اطبع كل الاحتمالات من aaa الى zzz
    - إذا رجعت الى بداية الشاشة لن تجد aaa لأنه قد تمت طباعتها ثم تم حذفها لأن buffer قد امتلئ واستبدل مكانها ببيانات أخرى ...

#Lesson 40 : What is ADT ?

ADT اختصار ل : Abstract Data Type

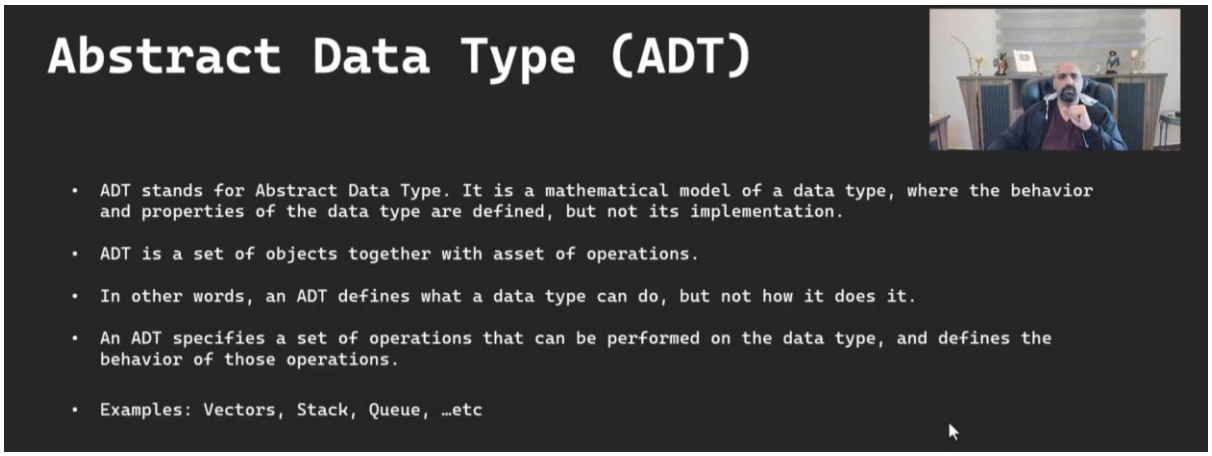
Abstract Data Type هي عبارة عن Class فيها Methods التي تساعدك على التعامل مع Data Structure

Abstract Data Type الهدف منه هو تسهيل حياة المبرمج
Abstract Data Type تم إخفاء Function التي تبيني Data Type وإظهار Members التي تفيدك فقط ، من غير معرفة كيف تم بناءه

أي Class هي **Abstract Data Type** إذا كان يوجد فيها Methods ، وغيرها

أمثلة على Abstract Data Type مثل مكتبة STL التي منها

- **Vector** : تستطيع الاستفادة من Members التي بداخلها مثل ... push , pop , size
- Stack , Queue ...
- ومثل بناء Class خاص بك مثل
  - Class **Employee** تم استخدام Abstract لمن يستعمل هذا Class من خارج Class (إخفاء Function & Procedure التي لا يتم تفيد مستخدم هذه Class)



#Lesson 41 : What is Map ? Dictionary

Map من أنواع Data Structure من مكتبة STL

Map تشبه Array مثل المرور على العناصر ب for loop أو الوصول الى عنصر معين ب Index

وفي **Map** تستطيع الوصول الى العناصر عن طريق **Key**

Map يتم تمثيلها بشيئين : value و Key
مثال : Key هو اسم الطالب ، و Value هو علامة الطالب

الفرق بين Map و Array هو :

- Array يكون الوصول إليها عن طريق Index يكون دائما رقم
- **Map : يكون الوصول إليها عن طريق Key**

Key : لا بد أن يكون فريد – لا يتم تكراره –

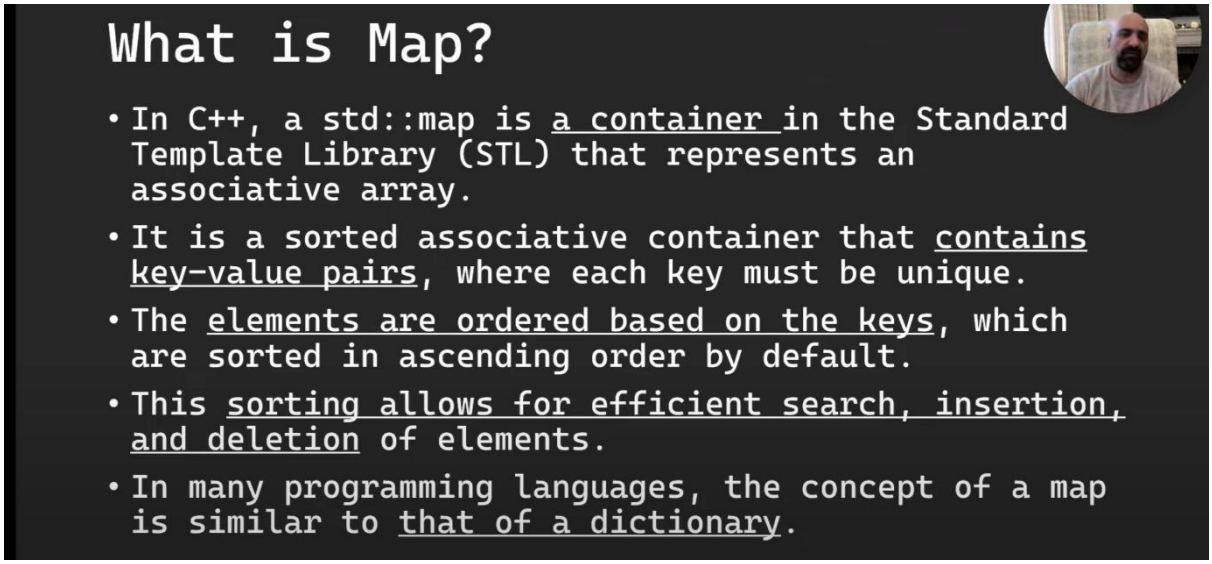
في C++ اسمها Map وفي جميع اللغات الأخرى مثل C# & Python تسمى Dictionary

يتم ترتيب العناصر على حسب Key بشكل تصاعدي بشكل افتراضي – مثل الترتيب الأبجدي –

وهذا الترتيب يفيد في البحث والإدراج والحذف وإيجادها – يكون البحث عن ايجادها جدا سريع لأنه يستخدم Algorithm Balance3 وغيرها

Answer	Question
A container in the Standard Template Library (STL) that represents an associative array	**What is a std::map in C++ ?**
Ascending order	**What is the default sorting order of std::map ?**
The value associated with the key will be updated	**What happens if you attempt to insert a duplicate key in a std::map**

```cpp
#include <iostream>
#include <map>

using namespace std;
int main()
{
// Declare a map with string keys and int values
    map<string, int> studentGrades;


    // Inserting grades for three students
    studentGrades["Ali"] = 77;     // Assigning a grade of 77 to the student
"Ali"
    studentGrades["Ahmed"] = 85;   // Assigning a grade of 85 to the student
"Ahmed"
    studentGrades["Fadi"] = 95;    // Assigning a grade of 95 to the student
"Fadi"


    // Printing all map values
    cout << "\nPrinting all map values..\n\n";

    // Iterating over the map
    for (const auto& pair : studentGrades) {
        cout << "Student: " << pair.first << ", Grade: " << pair.second << endl;
    }
    // Finding the grade for a specific student
    string studentName = "Fadi";
    if (studentGrades.find(studentName) != studentGrades.end()) {
        cout << studentName << "'s grade: " << studentGrades[studentName] <<
endl;
    } else {
        cout << "Grade not found for " << studentName << endl;
    }

    system("pause>0");
    return 0;
}
```



What is map?

In C++, a `std::map` is a container in the Standard Template Library (STL) that represents an associative array. It is a sorted associative container that contains key-value pairs, where each key must be unique. The elements are ordered based on the keys, which are sorted in ascending order by default. This sorting allows for efficient search, insertion, and deletion of elements.

Here are some key characteristics and features of `std::map`:

- Associative Container:
  - `std::map` is an associative container, meaning it associates a key with a value.
- Sorted Order:
  - The elements are sorted based on the keys. By default, sorting is done using the `<` operator.
- Unique Keys:
  - Each key in a `std::map` must be unique. If you attempt to insert a key that already exists, the associated value will be updated.
- Balanced Binary Search Tree:
  - Internally, `std::map` is usually implemented as a balanced binary search tree, such as a red-black tree. This ensures efficient search, insertion, and deletion operations.
- Efficient Lookups:
  - Searching for an element based on its key has a time complexity of O(log n), making it efficient for large datasets.
- Dynamic Sizing:
  - `std::map` dynamically adjusts its size as elements are inserted or removed.
- Key-Value Pairs:
  - Each element in the map is a key-value pair, where the key and value can be of any data type.
- Iterator Support:
  - It provides iterators that allow you to traverse the elements in sorted order.

In many programming languages, the concept of a `map` is similar to that of a `dictionary`. Both represent an associative container that stores key-value pairs, allowing efficient retrieval, insertion, and deletion of elements based on their keys.

#Lesson 42 : Union

يجب الحذر عند استعمال Union لكي لا يحدث خطأ في البرنامج

Syntex ل Union مثل Structure

Union من أنواع Data Structure مثله مثل Structure

باستخدام **Union** تستطيع تخزين أكثر من Data Type في مكان واحد في الذاكرة

عند تعريف struct يكون لكل واحد من Data Type له مكان خاص له في الذاكرة ، مثال :

```
struct MyStruct
{
    int x;
    float r;
    char s;
};
```

أما **Union** يكون كل Data Type **لهم مكان واحد فقط** مشترك فيما بينهم في الذاكرة ويكون حجم **Union** لأكبر Data Type موجود فيه

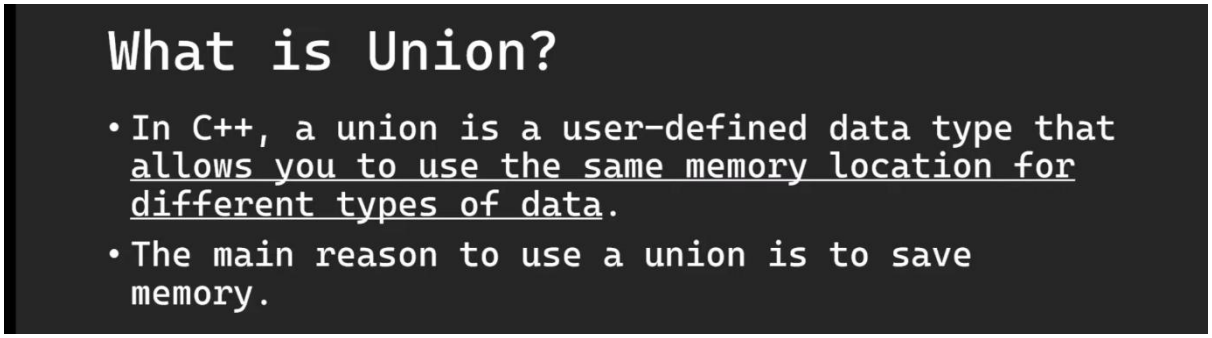
يتم استخدام **Union** لكي يتم توفير مساحة في الذاكرة

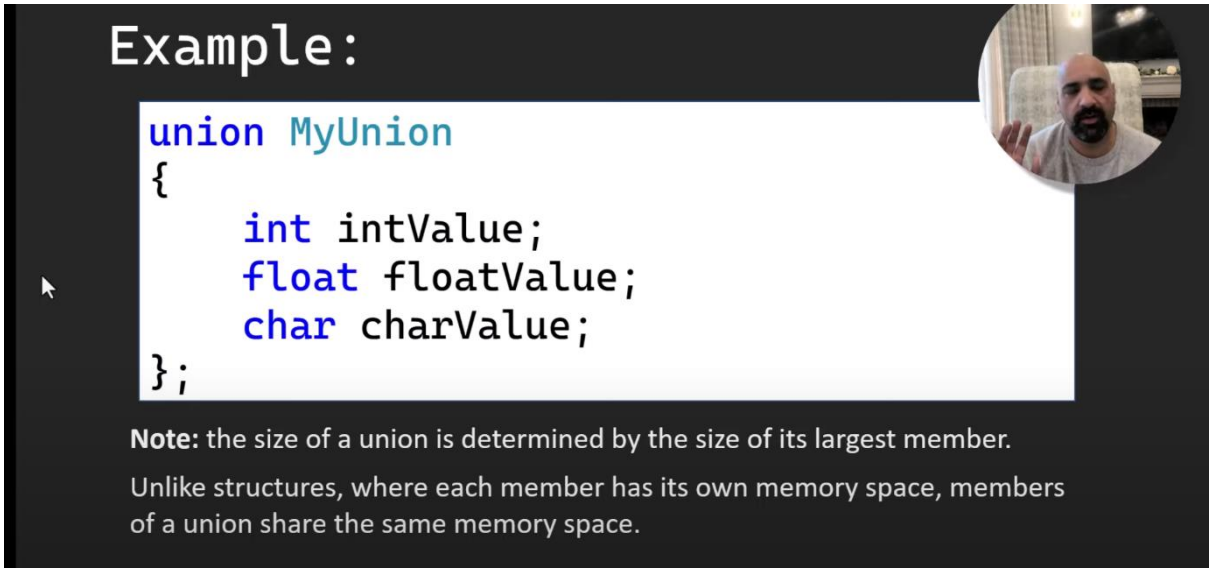
عند استخدام **Union** لابد من تذكر آخر عملية قمت بها في **Union** وإلا حدث خطأ في Run Time

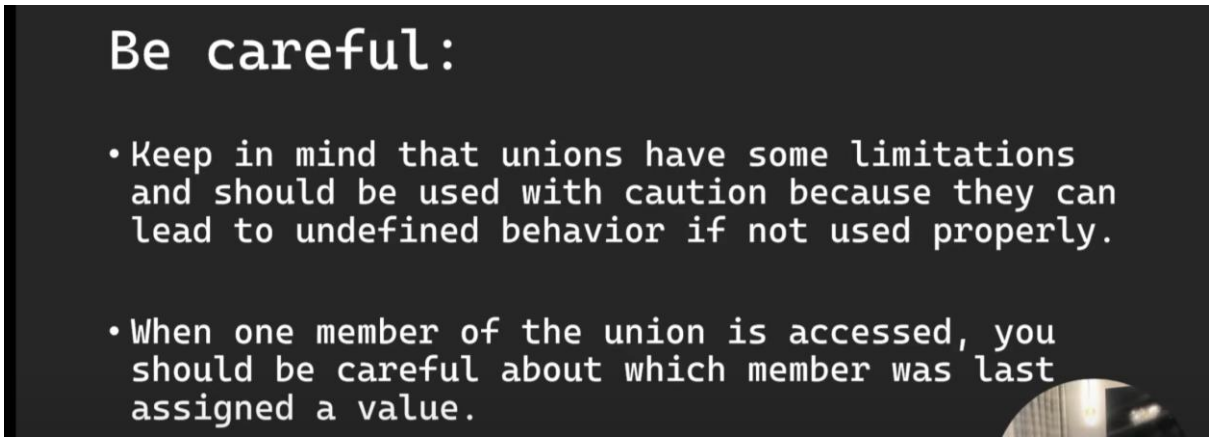
لأنه يوجد بها أكثر من Data Type داخل **Union** فيجب تذكر آخر عملية int , float , char...

```
#include <iostream>

using namespace std;
union MyUnion
{
    int intValue;
    float floatValue; // union ل float يتم حجز حجم
    char charValue;
};

int main()
{
    MyUnion myUnion;

    myUnion.intValue = 42;
    cout << "Integer value: " << myUnion.intValue << endl;

    myUnion.floatValue = 3.14f;
    cout << "Float value: " << myUnion.floatValue << endl;

    myUnion.charValue = 'A';
    cout << "Char value: " << myUnion.charValue << endl;

    system("pause>0");
    return 0;
}
```

Answer	Question
A user-defined data type that allows you to use the same memory location for different types of data	**What is a union in C++ ?**
Members of a union share the same memory space	**How are members of a union stored in memory ?**
It modifies the shared memory space of the union	**What happens when you assign a value to one member of a union ?**
The size of its largest member	**What is the size of a union determined by ?**
The value might not make sense for that type, resulting in undefined behavior	**What can happen if you access a member of a union that wasn't the last one written to ?**







In C++, a union is a user-defined data type that allows you to use the same memory location for different types of data. Unlike structures, where each member has its own memory space, members of a union share the same memory space.

Here's a simple example of a union in C++:

```
#include <iostream>

union MyUnion {
    int intValue;
    float floatValue;
    char charValue;
};

int main() {
    MyUnion myUnion;

    myUnion.intValue = 42;
    std::cout << "Integer value: " << myUnion.intValue << std::endl;

    myUnion.floatValue = 3.14f;
    std::cout << "Float value: " << myUnion.floatValue << std::endl;

    myUnion.charValue = 'A';
    std::cout << "Char value: " << myUnion.charValue << std::endl;

    return 0;
}
```

In this example, `MyUnion` is a union that can store an integer, a float, or a character. When you assign a value to one of the members, it modifies the shared memory space, and accessing another member will give you the interpretation of that data based on its type.

Keep in mind that unions have some limitations and should be used with caution because they can lead to undefined behavior if not used properly. When one member of the union is accessed, you should be careful about which member was last assigned a value. Also, the size of a union is determined by the size of its largest member.

- Undefined Behavior:
    - When you use a union, it's important to be mindful of which member was last assigned a value. Accessing a member that wasn't the last one written to results in undefined behavior. This is because the union shares the same memory space for all its members. If you read from a member that was not the last one written to, the value might not make sense for that type.
- Careful Member Access:
    - If you assign a value to one member of the union and then access another member, you might get unexpected results. For example, if you store an integer and then read it as a floating-point number, the interpretation of the bits will be different.
- Size Determined by Largest Member:
    - The size of a union is determined by the size of its largest member. This means that if you have a union with an `int`, a `float`, and a `char`, the size of the union will be the size of the `float` since it's the largest member. This can lead to inefficient memory usage if you're not careful with the design of your union.

الفهرس

هذا وصلى الله على نبينا محمد وعلى اله وصحبه أجمعين ، وآخر دعوانا أنِ الحمد لله رب العالمين