Data Structures & Algorithms

Prepared by: Mohamed Ayman

Machine Learning Researcher

spring 2019

sw.eng.MohamedAyman@gmail.com

facebook.com/sw.eng.MohamedAyman

linkedin.com/in/eng-mohamed-ayman

codeforces.com/profile/Mohamed_Ayman

Complexity Analysis

Agenda

1- Introduction to Data Structures

2- Characteristics of Data Structures

3- Execution Time Cases

4- Basics Operation

5- Common Asymptotic Notation

6- Array Data Structure


Let's
STARTUP

Agenda

1- Introduction to Data Structures

2- Characteristics of Data Structures

3- Execution Time Cases

4- Basics Operation

5- Common Asymptotic Notation

6- Array Data Structure

Introduction to Data Structure

- Data structure is a way to store and organize data in order to support efficient insertions, queries, searches, updates, and deletions. Although a data structure in itself does not solve the given programming problem, the algorithm operating on it does, using the most efficient data structure for the given problem may be a difference between passing or exceeding the problem's time limit.

- There are many ways to organize the same data and sometimes one way is better than the other on different context.

Characteristics of Data Structure

1 - Correctness:

Data structure implementation should implement its interface correctly.

2 - Time Complexity:

Running time or the execution time of operations of data structure must be as small as possible.

3 - Space Complexity:

Memory usage of a data structure operation should be as little as possible.

- The foundation terms of a data structure:

  1- Interface: It represents the set of operations that a data structure supports.

  2- Implementation: It provides the internal representation of a data structure.

Agenda

- ✓ 1- Introduction to Data Structures
- ✓ 2- Characteristics of Data Structures
- 3- Execution Time Cases
- 4- Basics Operation
- 5- Common Asymptotic Notation
- 6- Array Data Structure

Execution Time Cases

There are three cases which are usually used to compare various data structure's execution time in a relative manner.

- Best Case:

  Minimum time required for program execution. (Ω Notation)

- Average Case:

  Average time required for program execution. (θ Notation)

- Worst Case:

  Maximum time required for program execution. (Big O Notation)

Worst Case (Big O Notation)

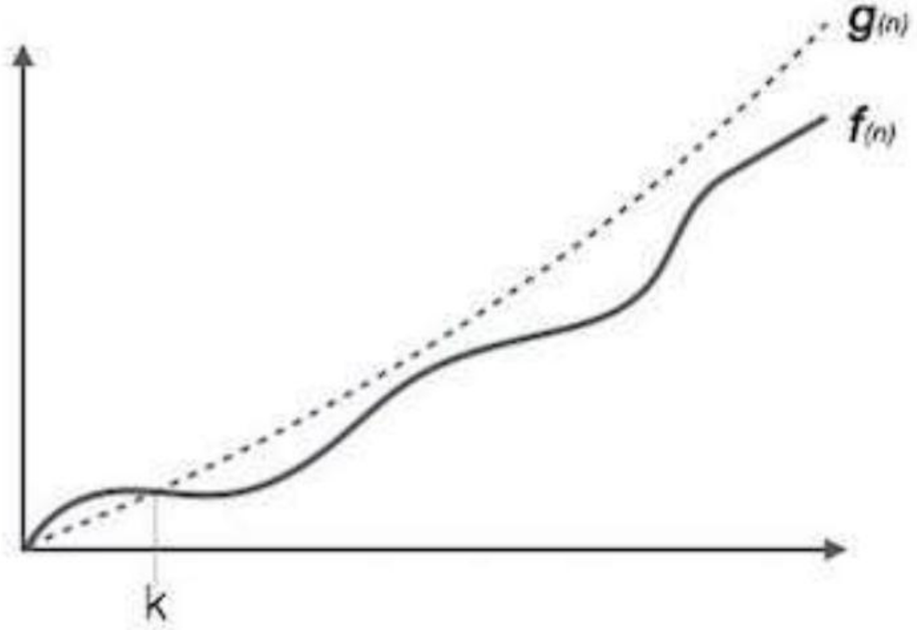
- The notation **O**(n) is the formal way to express the upper bound of an algorithm's running time.

- It measures the worst case time complexity or the longest amount of time an algorithm can possibly take to complete.

- The Big O notation asymptotically bounds a function from above and below. When we have only an asymptotic upper bound, we use O-notation.

- For a given function g(n), we denote by O(g(n)) (pronounced "big O of g of n" or sometimes just "O of g of n") the set of functions

Worst Case (Big O Notation)



For example: for a function f(n)

O(f(n)) = { g(n) : there exists C > 0 and k such that g(n) $\leq$ c * f(n) for all n > k }

Best Case (Ω Notation)

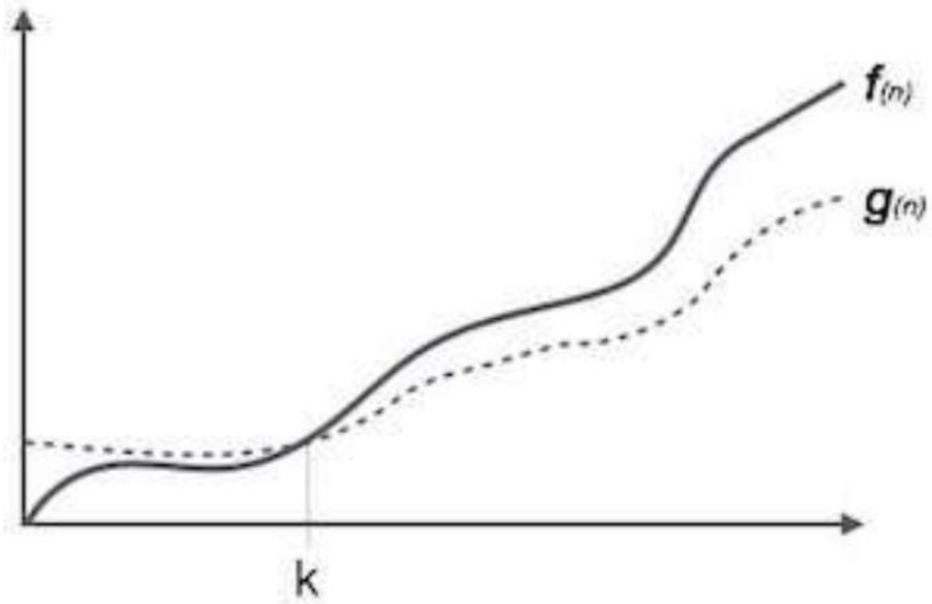
- The notation Ω(n) is the formal way to express the lower bound of an algorithm's running time.

- It measures the best case time complexity or the best amount of time an algorithm can possibly take to complete.

- Ω notation provides an asymptotic lower bound. For a given function g(n), we denote by Ω (g(n))

- For a given function g(n), we denote by O(g(n)) (pronounced "big-omega of g of n" or sometimes just "omega of g of n") the set of functions

Best Case (Ω Notation)

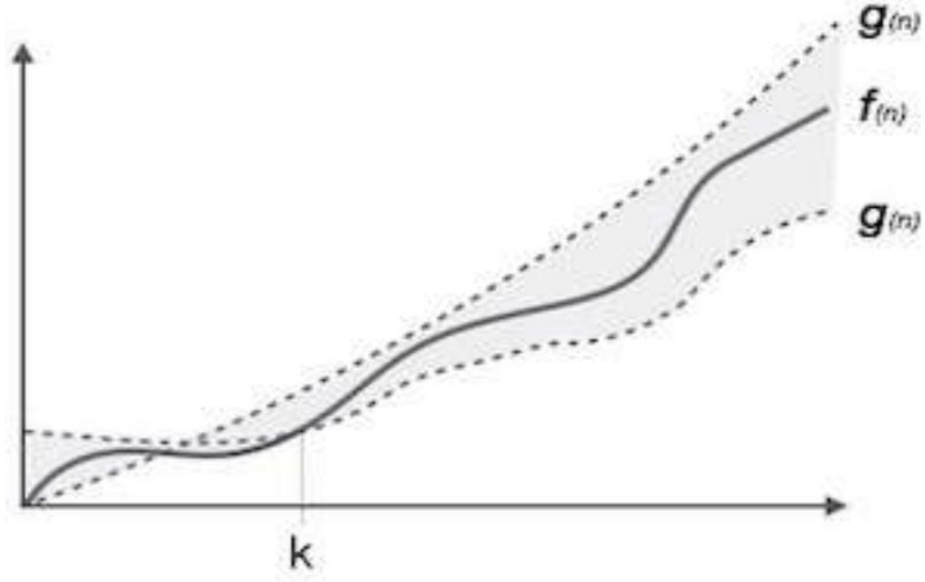


For example: for a function f(n)

$\Omega(f(n)) \geq \{ g(n) : \text{there exists } c > 0 \text{ and } k \text{ such that } g(n) \leq c * f(n) \text{ for all } n > k \}$

Average Case (θ Notation)

- The notation θ(n) is the formal way to express both the lower bound and the upper bound of an algorithm's running time.



For example: for a function f(n)

θ(f(n)) = { g(n) if and only if g(n) = O(f(n)) and g(n) = Ω(f(n)) for all n > k }

Basics Operation

- From the data structure point of view, following are some important categories of operations:

- Traversal: print all items in data structure
- Search: find an item in data structure
- Insert: add new item in data structure
- Delete: remove an item from data structure

Agenda

- ✓ 1- Introduction to Data Structures
- ✓ 2- Characteristics of Data Structures
- ✓ 3- Execution Time Cases
- ✓ 4- Basics Operation
- 5- Common Asymptotic Notation
- 6- Array Data Structure

Common Asymptotic Notation

- Sort the following functions in ascending order:

$O(n\,\hat{}\,3)$

$O(\log n)$

$O(n)$

$n\,\hat{}\,O(1)$

$O(1)$

$O(n\,\hat{}\,2)$

$O(n \log n)$

$2\,\hat{}\,O(n)$

$O(\sqrt{n})$

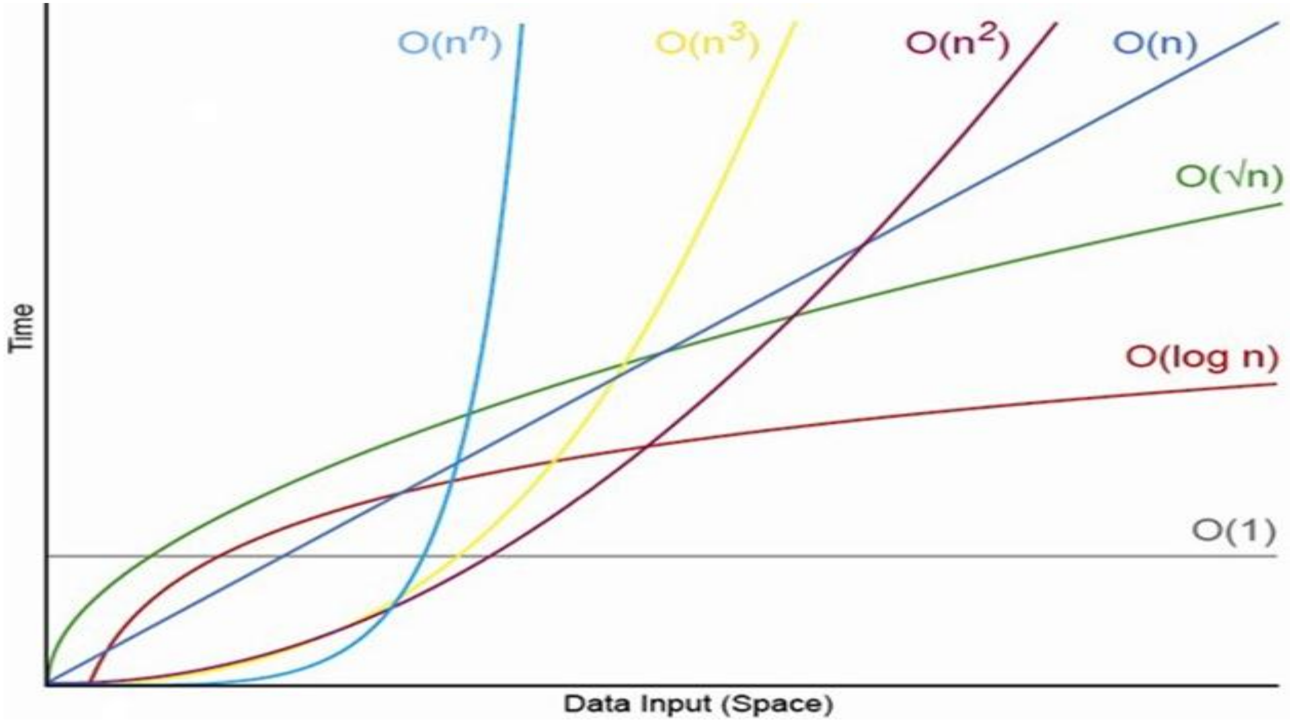
Common Asymptotic Notation

- Following is a list of some common asymptotic notations:

Constant	=>	O(1)
Logarithmic	=>	O(log n)
Square root	=>	O($\sqrt{n}$)
Linear	=>	O(n)
n * log n	=>	O(n log n)
Quadratic	=>	O(n ^ 2)
Cubic	=>	O(n ^ 3)
Polynomial	=>	n ^ O(1)
Exponential	=>	2 ^ O(n)

Common Asymptotic Notation

This graph show behavior of each function



Agenda

- ✔ 1- Introduction to Data Structures
- ✔ 2- Characteristics of Data Structures
- ✔ 3- Execution Time Cases
- ✔ 4- Basics Operation
- ✔ 5- Common Asymptotic Notation
- 6- Array Data Structure

Array Data Structure

- It 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.

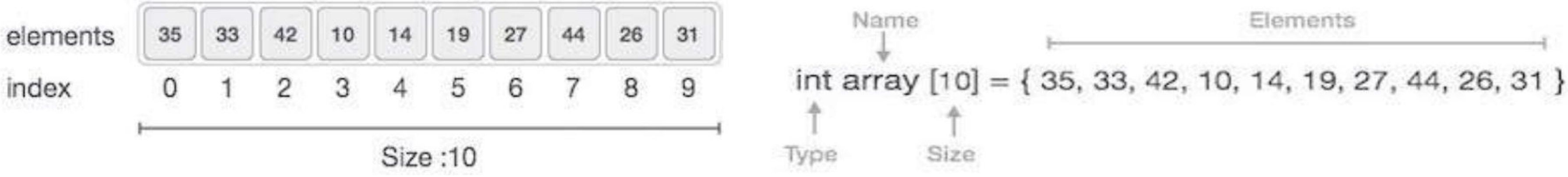
- Following are the important terms to understand the concept of Array.

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

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

- It can be declared in various ways in different languages.

Let's take C/C++ array declaration:



Insertion Operation

- 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.

- Complexity of insertion : O(n)

Insertion Operation Algorithm

- Let Array be a linear unordered array of MAX elements.
- Let LA be a Linear Array (unordered) with N elements and K is a positive integer such that 0 <= K <= N
- Following is the algorithm where ITEM is inserted into the K-th position of LA.

```
1. Start
2. Set J=N
3. Set N = N+1
4. Repeat steps 5 and 6 while J >= K
5. Set LA[J+1] = LA[J]
6. Set J = J-1
7. Set LA[K] = ITEM
8. Stop
```

Insertion Operation in C++

Link: repl.it/repls/UselessInconsequentialScreencast

```cpp
// This function insert item at index k in array
void insert_element(int item, int k) { // O(n)
    // check for invalid index
    if(k < 0 || k > n)
        return;
    // loop to shif values till reach index k
    int j = n;
    while (j >= k) {
        arr[j+1] = arr[j];
        j = j - 1;
    }
    arr[k] = item;
    // update new size
    n = n + 1;
}
```

Deletion Operation

- Deletion refers to removing an existing element from the array and re-organizing all elements of an array.

- Based on the requirement, element can be deleted at the beginning, end, or any given index of array.

- Complexity of deletion : O(n)

Deletion Operation Algorithm

- Consider LA is a linear array with N elements and K is a positive integer such that 0 <= K < N

- Following is the algorithm to delete an element available at the K-th position of LA.

```
1.  Start
2.  Set J=K
3.  Repeat steps 4 and 5 while J < N
4.  Set LA[J] = LA[ J+1]
5.  Set J = J+1
6.  Set N = N-1
7.  Stop
```

Deletion Operation in C++

Link: repl.it/repls/UselessInconsequentialScreencast

```cpp
// This function delete item at index k in array
void delete_element(int k) { // O(n)
    // check for invalid index
    if(k < 0 || k >= n)
        return;
    // loop to shif values till reach end of array
    int j = k;
    while (j < n) {
        arr[j] = arr[j+1];
        j = j + 1;
    }
    // update new size
    n = n - 1;
}
```

Search Operation

- You can perform a search for an array element based on its value or its index.

- Complexity of search : O(n)

Search Operation Algorithm

- Consider LA is a linear array with N elements and K is a positive integer such that 0 <= K < N
- Following is the algorithm to find an element with a value of ITEM using sequential search.

```
1. Start
2. Set J=0
3. Repeat steps 4 and 5 while J < N
4. IF LA[J] is equal ITEM THEN GOTO STEP 6
5. Set J = J +1
6. PRINT J, ITEM
7. Stop
```

Search Operation in C++

Link: repl.it/repls/UselessInconsequentialScreencast

```cpp
// This function search about item in array
int search_element(int item) { // O(n)
    int j = 0;
    while (j < n) {
        if(arr[j] == item)
            break;
        j = j + 1;
    }
    return j;
}
```

Update Operation

- Update operation refers to updating an existing element from the array at a given index.

- Complexity of update : O(1)

Update Operation Algorithm

- Consider LA is a linear array with N elements and K is a positive integer such that 0 <= K < N

- Following is the algorithm to update an element available at the K-th position of LA.

```
1. Start
2. Set LA[K] = ITEM
3. Stop
```

Update Operation in C++

Link: repl.it/repls/UselessInconsequentialScreencast

```cpp
// This function update item at index k in array
void update_element(int item, int k) { // O(1)
    // check for invalid index
    if(k < 0 || k >= n)
        return;
    // update index k with new item
    arr[k] = item;
}
```

Traversal Operation

- Traversal operation refers to print all elements from the array.

- Complexity of update : O(n)

Traversal Operation Algorithm

- Consider LA is a linear array with N elements

- Following is the algorithm to print all elements of LA.

```
1. Start
2. Set J=0
3. Repeat steps 4 and 5 while J < N
4.  PRINT LA[J]
5. Set J = J +1
6. Stop
```

Traversal Operation in C++

Link: repl.it/repls/UselessInconsequentialScreencast

```cpp
// This function print array values
void print_array() { // O(n)
    int i;
    for(i = 0 ; i < n ; i++)
        cout << arr[i] << ' ';
    cout << '\n';
}
```

Agenda

- ✓ 1- Introduction to Data Structures
- ✓ 2- Characteristics of Data Structures
- ✓ 3- Execution Time Cases
- ✓ 4- Basics Operation
- ✓ 5- Common Asymptotic Notation
- ✓ 6- Array Data Structure


DO
MORE.

Practice


Let's
STARTUP

Complexity Examples O(1)

```c
void fun()
{
    int i;
    for(i=1;i<=100;i++)
    {
        printf("*");
    }
}
//  O(1)
```

```c
void fun()
{
    int i,j;
    for(i=1;i<=100;i++)
    {
        for(j=1;j<=100;j++)
        {
            printf("*");
        }
    }
}
//  O(1)
```

Complexity Examples O(n)

```
void fun(int n)
{
     int i;
     for(i=1;i<=n;i++)
     {
          printf("*");
     }
}
//   O(n)
```

```
void fun(int n)
{
     int i,j;
     for(i=1;i<=n;i++)
     {
          for(j=1;j<=100;j++)
          {
               printf("*");
          }
     }
}
//   O(n)
```

Complexity Examples O(n^2)

```
void fun(int n)
{
    int i,j,k;
    for(i=1;i<=n;i++)
    {
        for(j=1;j<=i;j++)
        {
            printf("*");
        }
    }
}
//  O(n^2)
```

```
void fun(int n)
{
    int i,j,k;
    for(i=1;i<=n;i++)
    {
        for(j=1;j<=n;j++)
        {
            printf("*");
        }
    }
}
//  O(n^2)
```

```
void fun(int n)
{
    int i,j,k;
    for(i=1;i<=n;i++)
    {
        for(j=1;j<=i;j++)
        {
            for(k=1;k<=100;k++)
            {
                printf("*");
            }
        }
    }
}
//  O(n^2)
```

Complexity Examples $O(n^3)$

```
void fun(int n)
{
    int i,j,k;
    for(i=1;i<=n;i++)
    {
        for(j=1;j<=i;j++)
        {
            for(k=1;k<=j;k++)
            {
                printf("*");
            }
        }
    }
}
//  O(n^3)
```

```
void fun(int n)
{
    int i,j,k;
    for(i=1;i<=n;i++)
    {
        for(j=1;j<=n;j++)
        {
            for(k=1;k<=n;k++)
            {
                printf("*");
            }
        }
    }
}
//  O(n^3)
```

Complexity Examples O(log n)

```
void fun(int n)
{
    int i=1;
    while(i<=n)
    {
        printf("*");
        i=i*2;
    }
}
//  O(log n)
```

```
void fun(int n)
{
    int i;
    for(i=1;i<=n;i++)
    {
        int m=n, j=1;
        while(j<=m)
        {
            printf("*");
            j=j*2;
        }
    }
}
//  O(nlog n)
```

Complexity Examples O(log n), O(n log n)

```
void fun(int n)
{
    int i=1;
    while(i<=n)
    {
        printf("*");
        i=i*2;
    }
}
//  O(log n)
```

```
void fun(int n)
{
    int i;
    for(i=1;i<=n;i++)
    {
        int m=n, j=1;
        while(j<=m)
        {
            printf("*");
            j=j*2;
        }
    }
}
//  O(nlog n)
```

Complexity Examples $O(\sqrt{n})$

```
void fun(int n)
{
    int i=1, s=1;
    while(s<=n)
    {
        i=i+1;
        s=s+i;
        printf("*");
    }
}
//  O(√n)
```

```
void fun(int n)
{
    int i;
    for(i=1;i*i<=n;i++)
    {
        printf("*");
    }
}
//  O(√n)
```

Complexity Examples $O(\sqrt{n}\ \log n)$

```
void fun(int n)
{
	int i,j,k;
	for(i=1;i*i<=n;i++)
	{
		for(k=1;k<=n;k*=2)
		{
			printf("*");
		}
	}
}
//  O(√n logn)
```

Complexity Examples $O(n\,(\log n)\hat{}2)$

```
void fun(int n)
{
    int i,j,k;
    for(i=n/2;i<=n;i++)
    {
        for(j=1;j<=n;j*=2)
        {
            for(k=1;k<=n;k*=2)
            {
                printf("*");
            }
        }
    }
}
//  O(n (logn)^2)
```

Complexity Examples O(n^2 log n)

```c
void fun(int n)
{
    int i,j,k;
    for(i=n/2;i<=n;i++)
    {
        for(j=1;j<=n/2;j++)
        {
            for(k=1;k<=n;k*=2)
            {
                printf("*");
            }
        }
    }
}
//  O(n^2 logn)
```


DO
MORE.

Array Data Structure - Initialization

Link: repl.it/repls/UselessInconsequentialScreencast

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

const int N = 1e5+3;
int n;
int arr[N];
```

Array Data Structure - Insert

Link: repl.it/repls/UselessInconsequentialScreencast

```
int main() {
    // n = 5
    cout << "please enter array size : \n";
    cin >> n;

    // arr = 10 20 30 40 50
    cout << "please enter array elements : \n";
    for (int i = 0 ; i < n ; i++)
        cin >> arr[i];

    cout << '\n';
    cout << "array elements : \n";
    print_array();
    // 10 20 30 40 50
```

Array Data Structure - Insert

Link: repl.it/repls/UselessInconsequentialScreencast

```
please enter array size :
5
please enter array elements :
10
20
30
40
50

array elements :
10 20 30 40 50
```

Array Data Structure - Delete

Link: repl.it/repls/UselessInconsequentialScreencast

```
    cout << '\n';
    delete_element(0);
    cout << "array elements after delete element at position 0 : \n";
    print_array();
    // 10 20 30 70 40 50 90 60

    delete_element(n-1);
    cout << "array elements after delete element at position " << n-1 << " : \n";
    print_array();
    // 10 20 30 70 40 50 90

    delete_element(3);
    cout << "array elements after delete element at position 3 : \n";
    print_array();
    // 10 20 30 40 50 90
```

Array Data Structure - Delete

Link: repl.it/repls/UselessInconsequentialScreencast

```
array elements after delete element at position 0 :
10 20 30 70 40 50 90 60
array elements after delete element at position 6 :
10 20 30 70 40 50 90
array elements after delete element at position 3 :
10 20 30 40 50 90
```

Array Data Structure - Search

Link: repl.it/repls/UselessInconsequentialScreencast

```
    cout << '\n';
    cout << "position of element 40 is : ";
    cout << search_element(40) << "\n";
    // 3

    cout << "position of element 90 is : ";
    cout << search_element(90) << "\n";
    // 5
```

```
position of element 40 is : 3
position of element 90 is : 5
```

Array Data Structure - Update

Link: repl.it/repls/UselessInconsequentialScreencast

```
    cout << '\n';
    update_element(100,0);
    cout << "array elements after update element at position 0 with value 100 : \n";
    print_array();
    // 100 20 30 40 50 90

    update_element(900,n-1);
    cout << "array elements after update element at position " << n-1 << " with value 900 : \n";
    print_array();
    // 100 20 30 40 50 900

    update_element(400,3);
    cout << "array elements after update element at position 3 with value 400 : \n";
    print_array();
    // 100 20 30 400 50 900
}
```

Array Data Structure - Update

Link: repl.it/repls/UselessInconsequentialScreencast

```
array elements after update element at position 0 with value 100 :
100 20 30 40 50 90
array elements after update element at position 5 with value 900 :
100 20 30 40 50 900
array elements after update element at position 3 with value 400 :
100 20 30 400 50 900
```


DO
MORE.

Assignment

References

[01]	Online Course YouTube Playlists	https://bit.ly/2Pq88rN
[02]	Introduction to Algorithms Thomas H. Cormen	https://bit.ly/2ONhuSn
[03]	Competitive Programming 3 Steven Halim	https://nus.edu/2z4OvyK
[04]	Fundamental of Algorithmics Gilles Brassard and Paul Bartley	https://bit.ly/2QuvwbM
[05]	Analysis of Algorithms An Active Learning Approach	http://bit.ly/2EgCCYX
[06]	Data Structures and Algorithms Annotated Reference	http://bit.ly/2c37XEv
[07]	Competitive Programmer's Handbook	https://bit.ly/2APAbeG
[08]	GeeksforGeeks	https://geeksforgeeks.org
[09]	Codeforces Online Judge	http://codeforces.com
[10]	HackerEarth Online Judge	https://hackerearth.com
[11]	TopCoder Online Judge	https://topcoder.com

Questions ?