Lecture#1

Data Structures

Dr. Abu Nowshed Chy

Department of Computer Science and Engineering

University of Chittagong

January 05, 2025

Faculty Profile

Course Materials

1. Seymour Lipschutz, "Data Structure", Schaum's Outlines, Tata McGraw Hill
2. Ellis Horowitz, Sartaj Sahni and Sanguthevar Rajasekaran, "Fundamentals of Computer Algorithms", Orient Black Swan.
3. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein, "Introduction to Algorithms", The MIT Press.
4. Kenneth H. Rosen "Discrete Mathematics and Its Applications", Mc Graw Hill
5. Specific material and lecture slides

Goal

"I will, in fact, claim that the difference between a bad programmer and a good one is whether he considers his code or his data structures more important. Bad programmers worry about the code. Good programmers worry about data structures and their relationships."

Linus Torvalds, 2006

Good Computer Program

- A computer program is a series of instructions to carry out a particular task written in a language
- There are a number of features for a good program
  - Run efficiently and correctly
  - Easy to read and understand
  - Easy to debug and modify
  - Easy to maintain
- Program consists of two things: Data Structures and Algorithms

Data Structure

Data Structure

- Data: Data are simply a value or set of values of different type which is called data types like string, integer, char, etc.
- Information: Meaningful data
- Structure: Way of organizing information, so that it is easier to use
- Therefore, we can define data structure as:
  - Its a way of organizing data in such a way so that data can be easier to use

Data Structure

Data Structure = Organized Data + Allowed Operations

Therefore a data structure is made of :

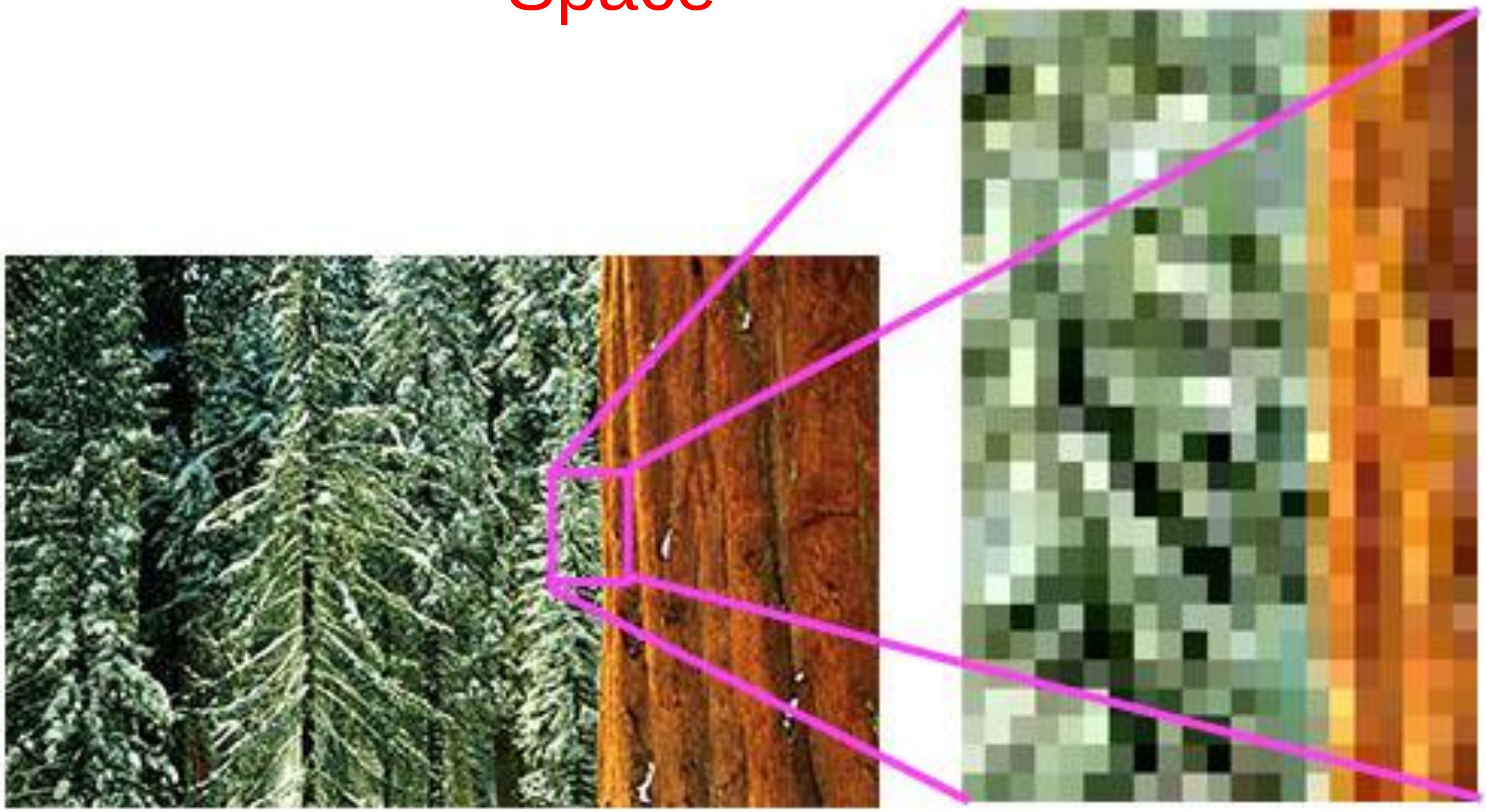
- A set of data values
- A set of functions specifying the operations permitted on the data values
- A set of axioms describing how these operations work

Why Data Structure!!!

- Human requirement with computer are going to complex day by day. To solve the complex requirements in an efficient way we need to know about data structure.
- To provide fastest solution of human requirements
- Provide efficient solution of complex problem:
  - Space
  - Time

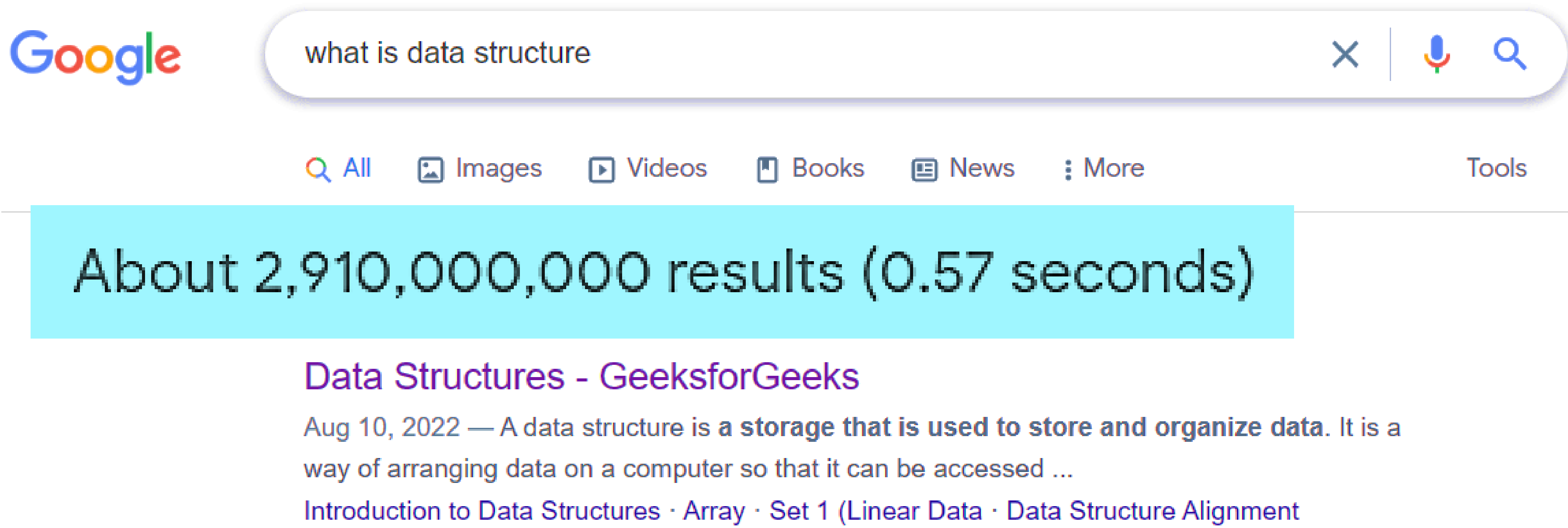
Why Data Structure!!!

Space



Why Data Structure!!!

Time



Why So Many Data Structure!!!

- Ideal data structure: "fast", "elegant", memory efficient
- Generates tensions:
  - Time vs. space
  - Performance vs. elegance
  - Generality vs. simplicity
  - One operation's performance vs. another's

The study of data structures is the study of tradeoffs. That's why we have so many of them!

Types of Data Structures

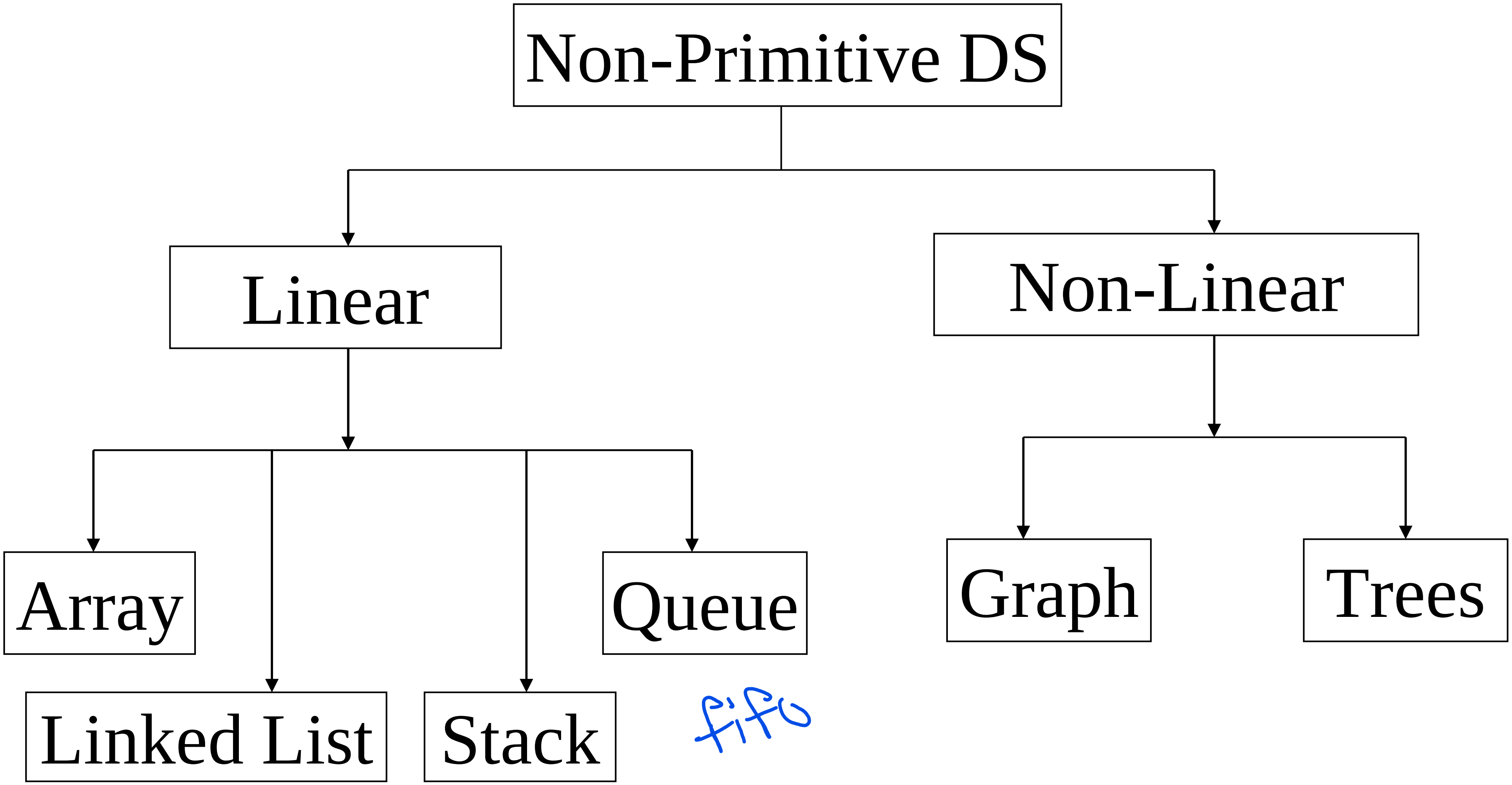
- Data structure
  - Primitive DS
    - Integer
    - Float
    - Character
    - Pointer
  - Non-Primitive DS

ancient,
Directly operated by
machine instruction

Primitive Data Structure

- There are basic structures and directly operated upon by the machine instructions.
- A primitive data structure used to represent standard data types of any one of the computer languages.
- Integer, Floating-point number, Character, String, etc. are fall in this category.

Types of Data Structures



Non-Primitive Data Structure

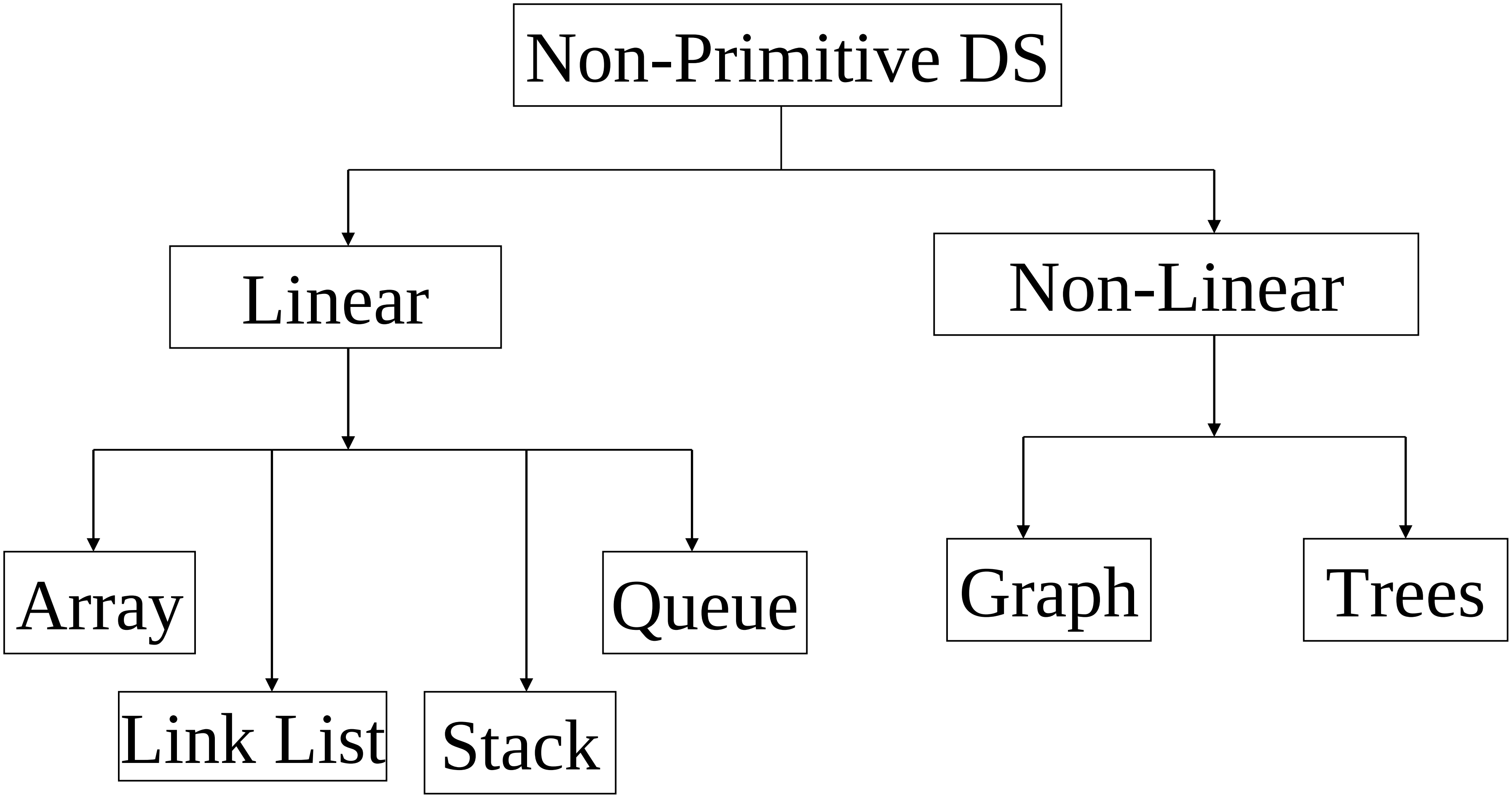
- There are more sophisticated data structures.
- These are derived from the primitive data structures.
- The non-primitive data structures emphasize on structuring of a group of homogeneous (same type) or heterogeneous (different type) data items.
- Array, Stack, Queue, Tree, Graph are example of non-primitive data structures.
- The design of an efficient data structure must consider the operations to be performed on that data structure.

Linear Vs. Non-linear DS

- Linear Data Structures: A linear data structures traverses the data elements sequentially, in which only one data element can directly be reached. E.g. Array, Linked List

- Non-Linear Data Structures: Every data item is attached to several other data items in a way that is specific for reflecting relationships. The data items are not arranged in a sequential structure. E.g. Tree, Graph

Types of Data Structures



Array

- An array is a collection of data items, all of the same type, accessed using a common name.
- A linear array Student contains the name of six students

int Student [6]

Array Index	
0	Dalia
1	Sumona
2	Mubtasim
3	Anamul
4	Ibtisam
5	Jarin

Student [0] = Dalia

Student [1] = Sumona

Student [2] = Mubtasim

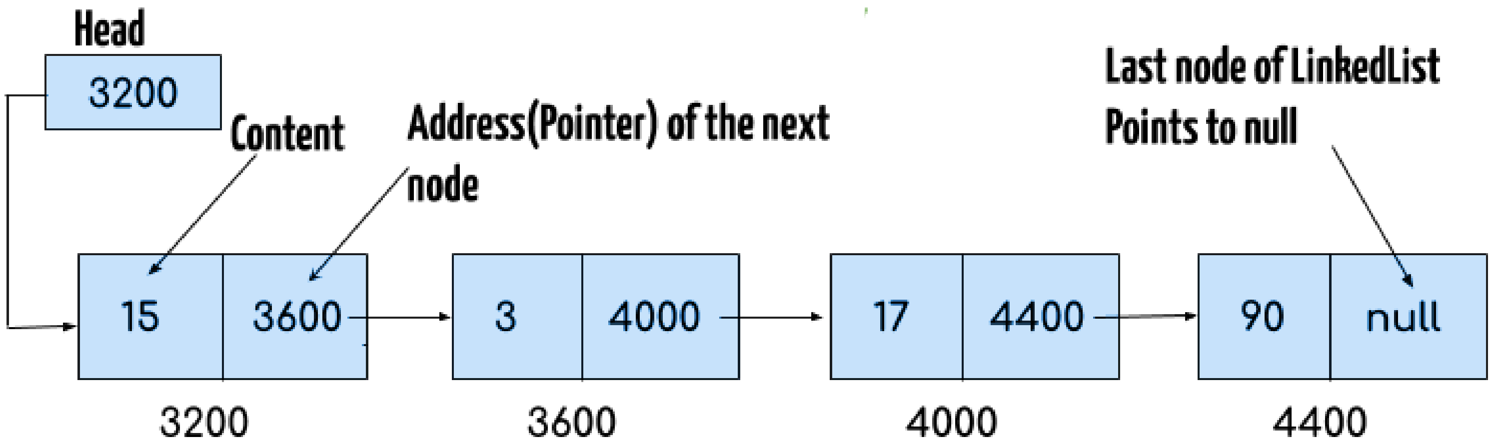
Student [3] = Anamul

Student [4] = Ibtisam

Student [5] = Jarin

Linked List

- A linked list, or one way list, is a linear collection of data elements, called nodes
- Each node of the list contains the data item and a pointer to the next node



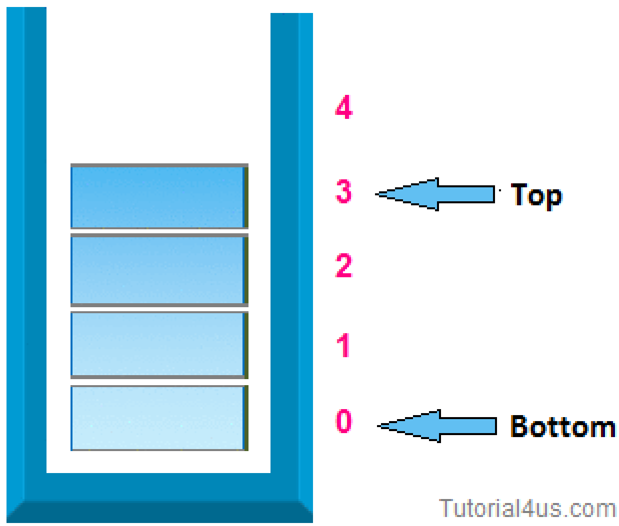
Stack

- A stack is a data structure in which items can be inserted only from one end and deleted from the same end.
- Stacks are a special form of collection with LIFO semantics
- Two methods

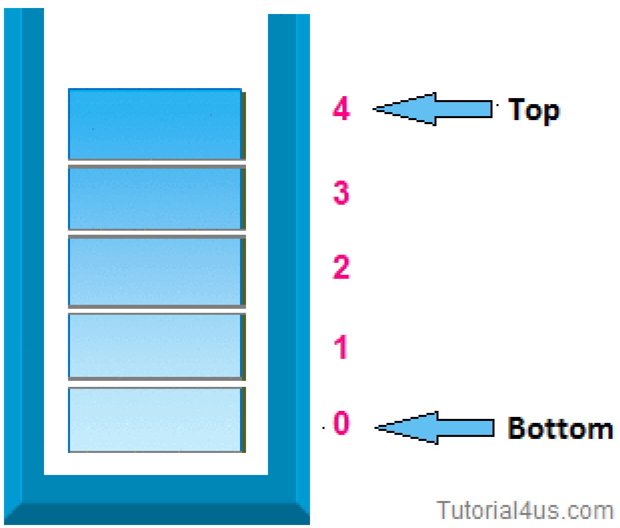
  PUSH → add item to the top of the stack

  POP → remove an item from the top of the stack
- It could be thought of just like a stack of plates placed on table, a person always takes off a plate from the top and the new plates are placed on to the stack at the top.

Stack



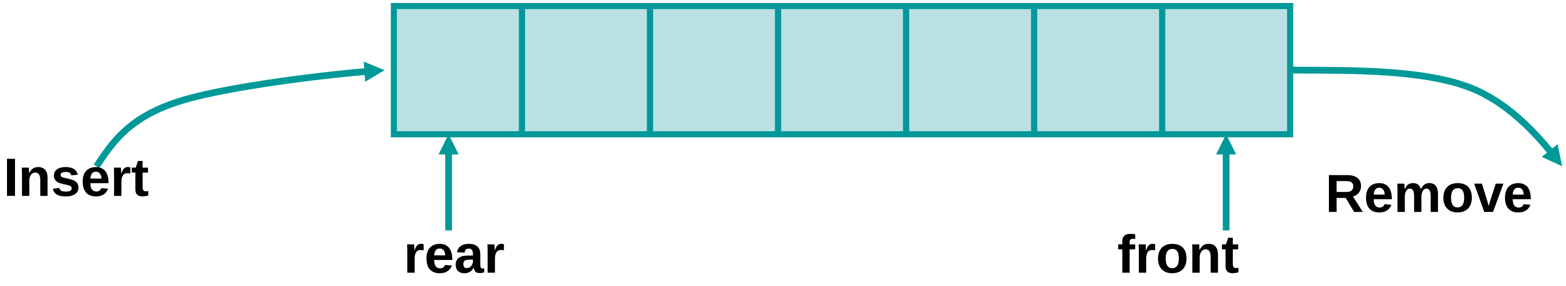
PUSH Operation



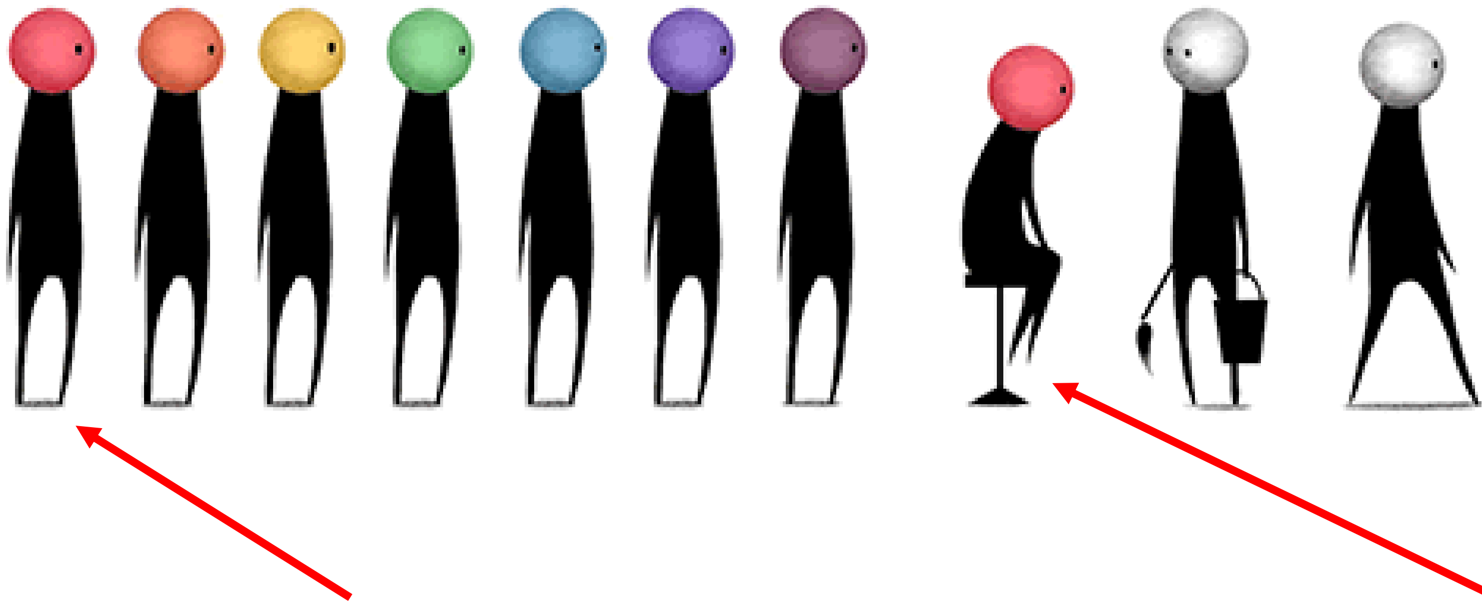
POP Operation

Queue

- A queue is a two ended data structure where insertion is done at one end and deletion is performed at the other end
- Queue are a special form of collection with FIFO semantics
- The insertion end is called rear and the deletion end is called front



Queue



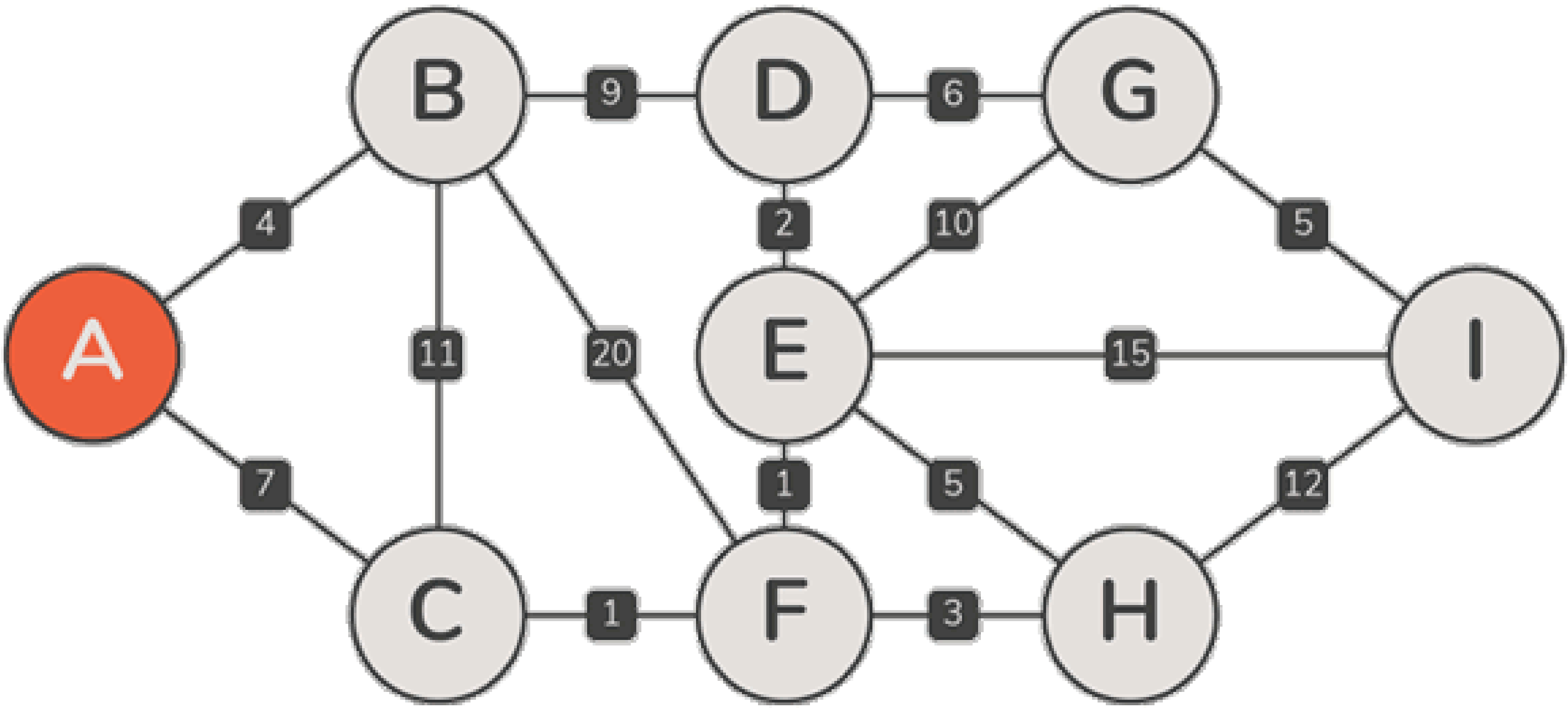
Insertion End → Rear

Deletion End → Front

Graph

- A graph is a set of items connected by edges. Each items is called a vertex or node.
- An edge connects a pair of vertices and may have weight such as length, cost and another measuring instrument or relationships according to the graph.
- Vertices on the graph are shown as point or circles and edges are drawn as arcs or line segment.

Graph

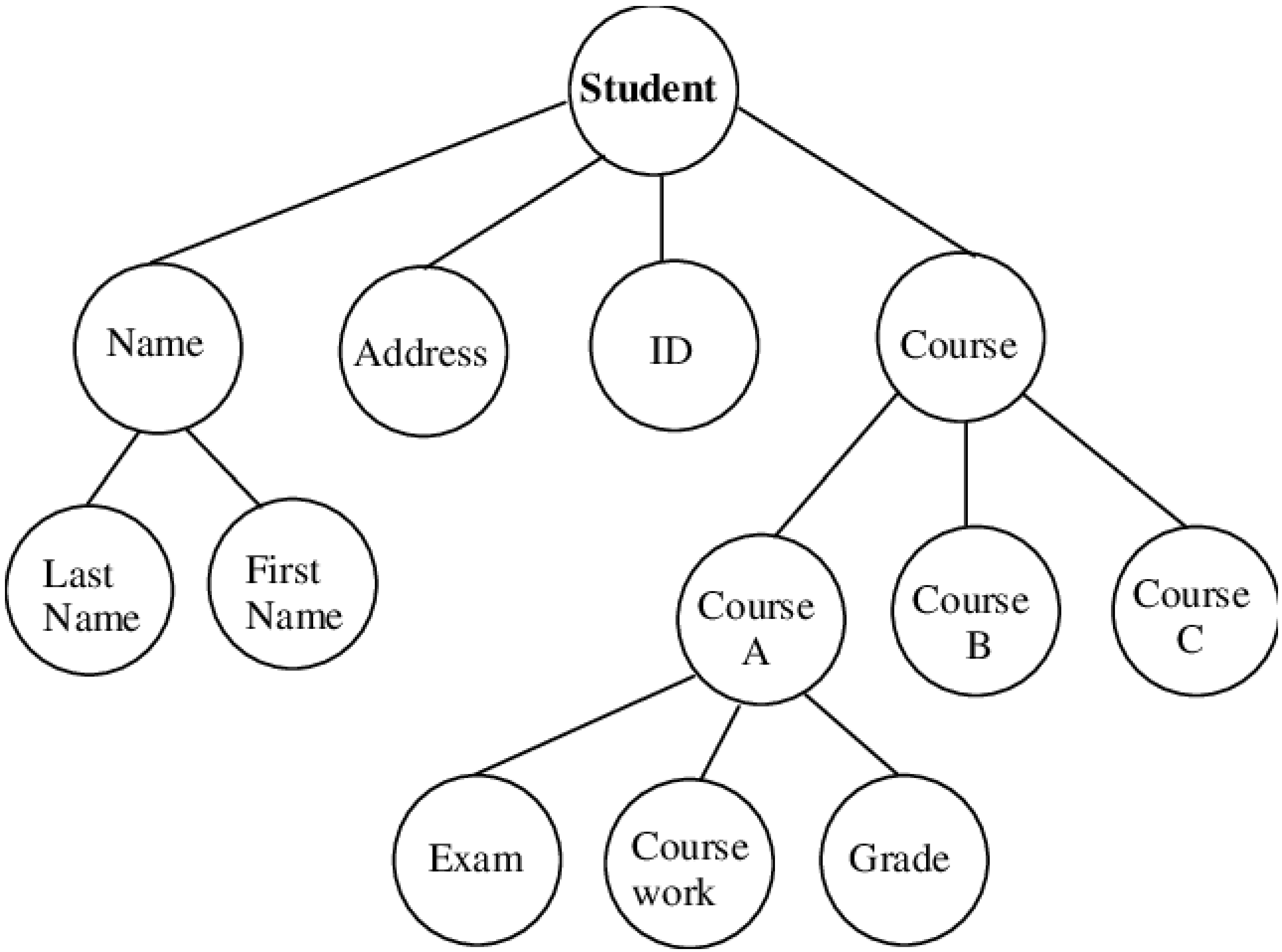


Finding Shortest Path

Tree

- A tree can be defined as finite set of data items (nodes).
- Tree is non-linear type of data structure in which data items are arranged or stored in a sorted sequence.
- Tree represent the hierarchical relationship between various elements.

Tree



Student
Name
Address
ID
Course
Last Name
First Name
Course A
Course B
Course C
Exam
Course work
Grade

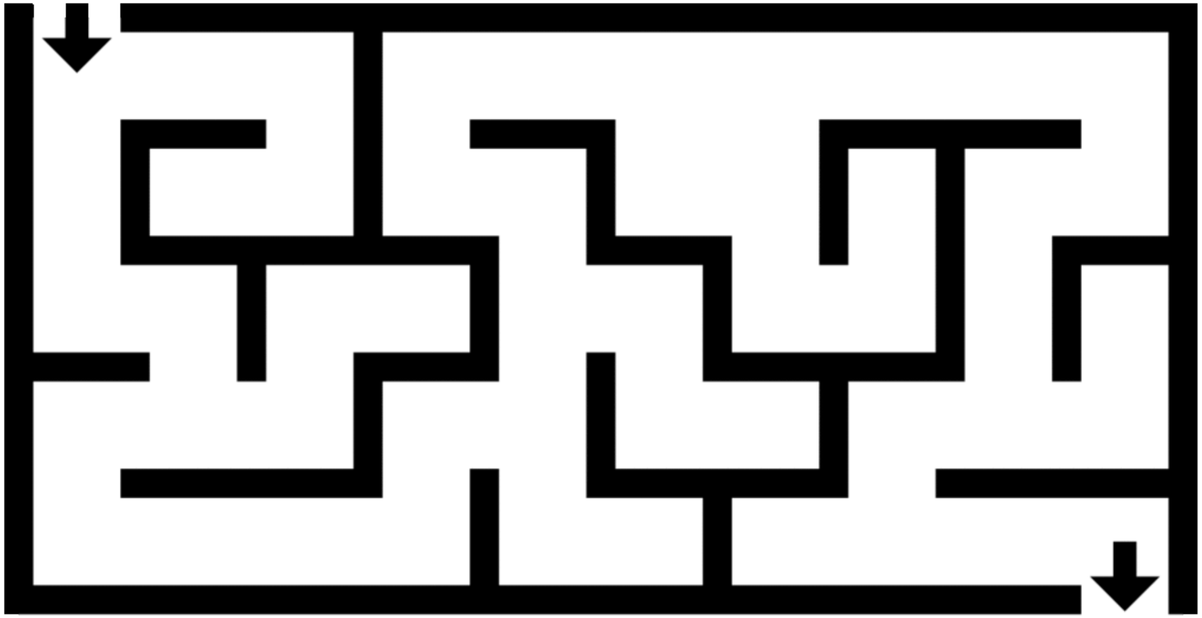
Data Structure Operations

- There are six basic operations that can be performed on data structure:-

1. Traversing
2. Searching
3. Inserting
4. Deleting
5. Sorting
6. Merging

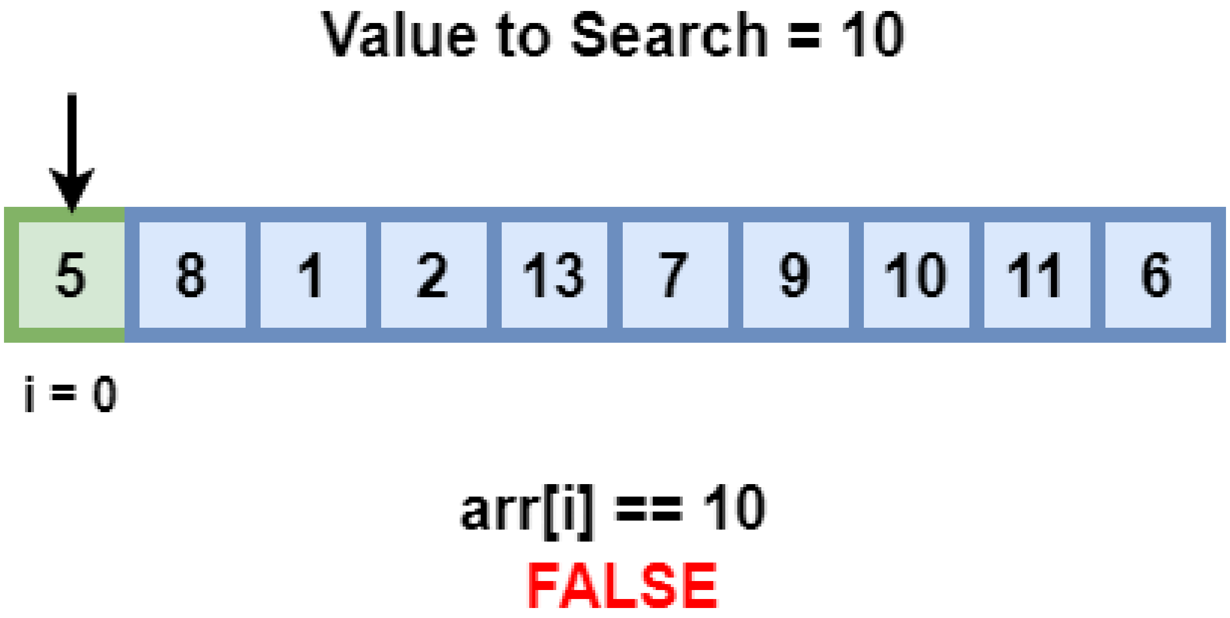
Traversing

- Traversing means accessing and processing each element in the data structure exactly once. The accessing and processing is sometimes called “Visiting” the record.



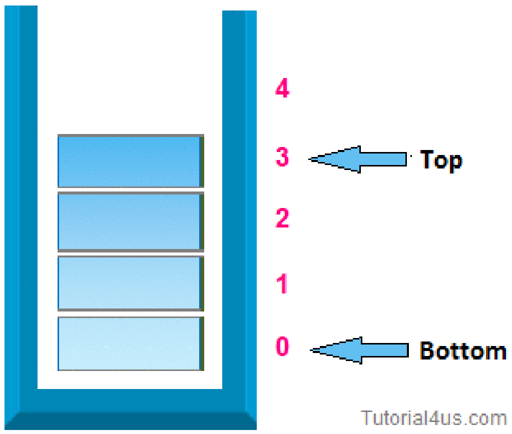
Searching

- Finding the location of the record with a given key value or finding the locations of all record which satisfy one or more conditions



Inserting

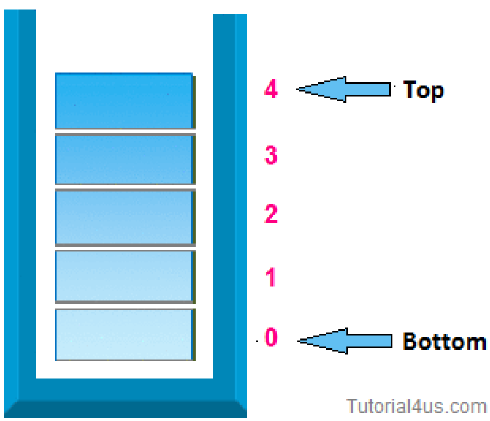
- Inserting an element is adding an element in the data structure at any position. After insert operation the number of elements are increased by one.



PUSH Operation

Deleting

- Deleting an element is removing an element in the data structure at any position. After deletion operation the number of elements are decreased by one.



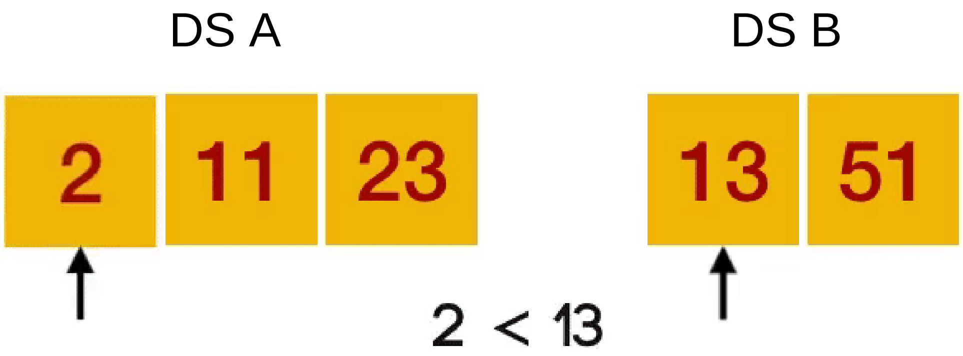
POP Operation

Sorting

- Sorting is the process of arranging a list of elements in a sequential order. The sequential order may be descending order or an ascending order according to the requirements of the data structure.

Merging

- The process of combining the elements of two data structures into a single data structure is called merging.



Selecting a Data Structure

If you are a programmer and want to choose a data structure for your program, consider these factors:

- Analyze the problem to determine the resource constraints.
- Determine the basic operations that must be supported.
- The size of the data.
- The size of the storage.
- The data dynamics, such as changing or editing the data

Algorithms

Algorithm & Its Characteristics

Algorithm refers to the logic of a program and a step-by-step description of how to arrive at the solution of a given problem.

In order to qualify as an algorithm, a sequence of instruction must have the following characteristics:

- Each instruction should be precise and unambiguous
- Each instruction should be complete in a finite time
- Any instructions should not be repeated infinitely
- After executing the instructions, the desired results must be obtained

Representation of an Algorithm

An algorithm can be represented in any of the following ways:

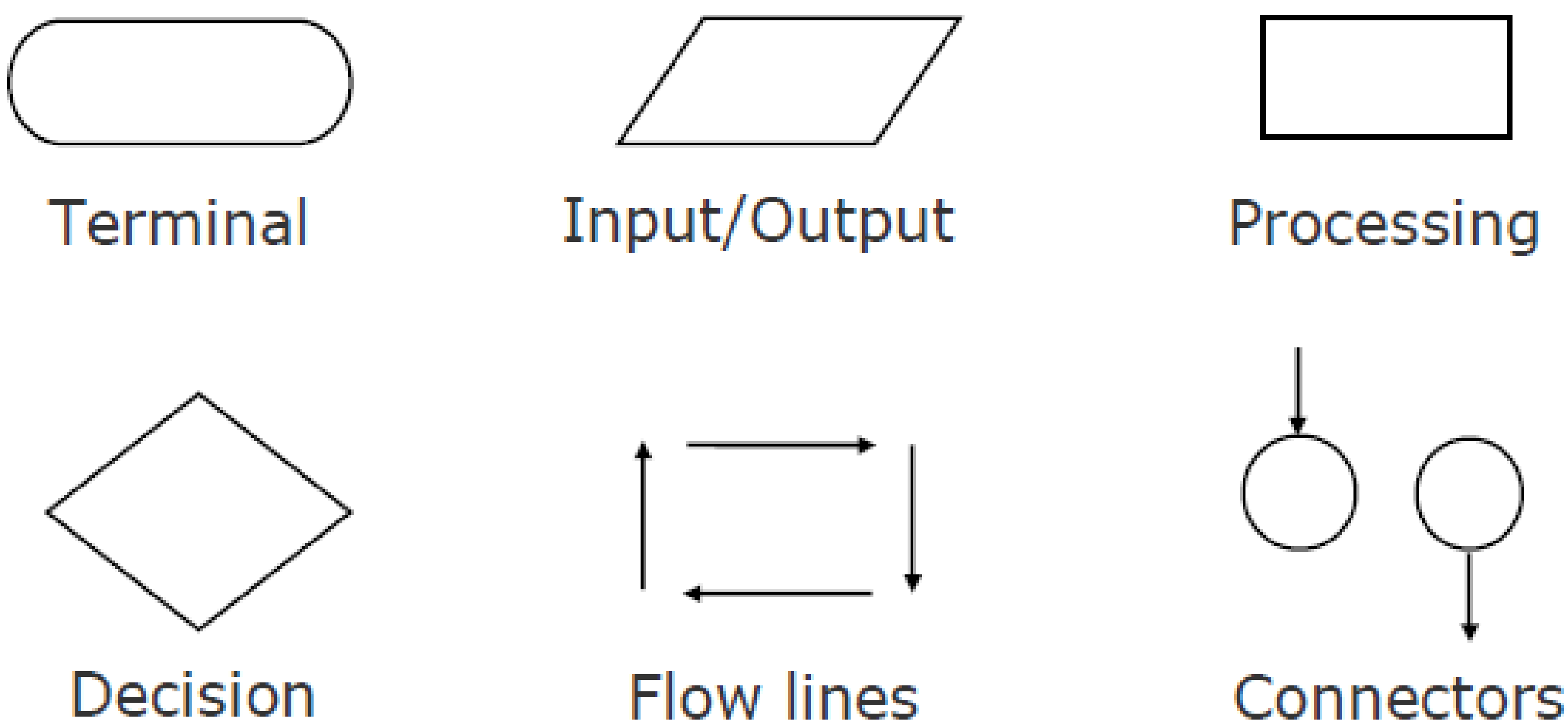
- As programs
- As flowcharts
- As pseudocodes

When an algorithm is represented in the form of a programming language, it becomes a program.

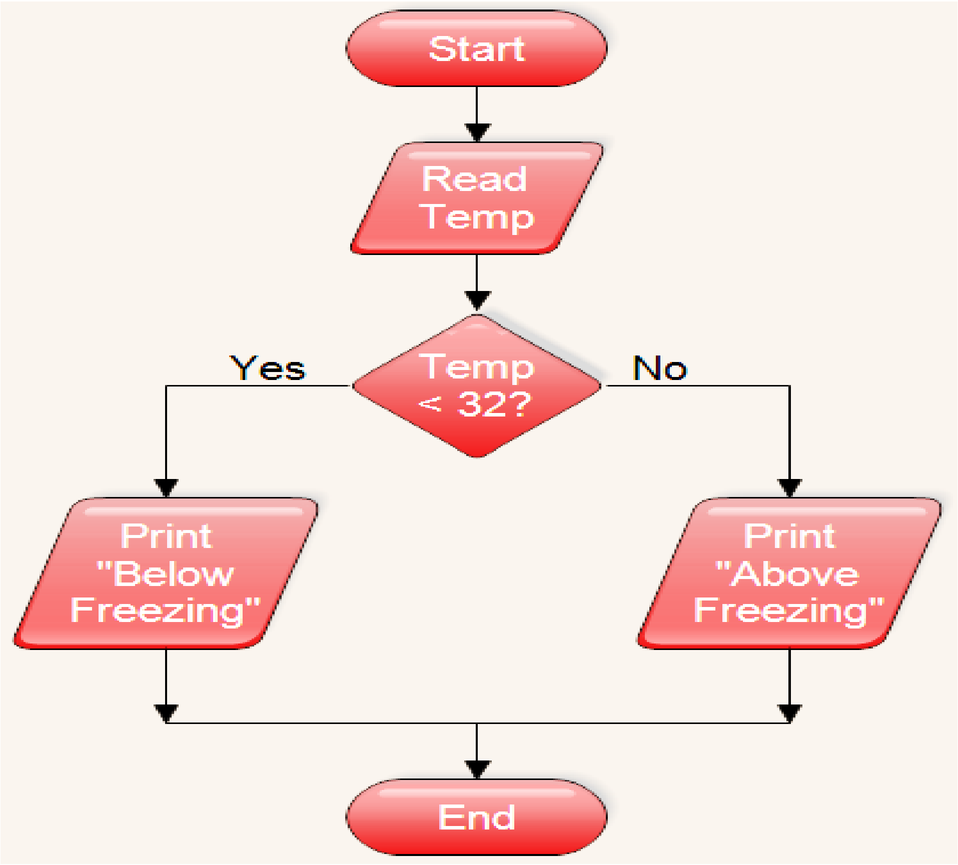
Thus any program is an algorithm, although the reverse is not true.

Flowchart

A flowchart is a pictorial representation of an algorithm.



Flowchart Example



Pseudocode

A program planning tool where algorithm or program logic is written in an ordinary natural language using a structure that resembles computer instructions.

```
PROGRAM MakeACupOfTea:
  Organise everything together;
  Plug in kettle;
  Put teabag in cup;
  Put water into kettle;
  Wait for kettle to boil;
  Add water to cup;
  Remove teabag with spoon/fork;
  Add milk and/or sugar;
  Serve;
END.
```

Pseudocode


"You can't just copy-pase pseudocode
into a program and expect it to work"
that's where
you're wrong
kiddo
python

Areas of Study

- How to devise algorithms: an understanding of the algorithmic structures and their representation in the form of Pseudocode or flowcharts
- How to validate algorithms: determining the correctness
- How to analyze algorithms: determining the time and storage of an algorithm requires.
- How to test a program: debugging
- Expressing the algorithm: To implement the algorithm using a programming language

Analysis of Algorithm

Analysis of an algorithm refers to the task of determining how much computing time and storage an algorithm requires.

- An efficient algorithm uses as few resources as possible
  - Time
  - Space (memory)
- Often, have trade-off time/space
  - Faster if use more space
  - Smaller if takes longer
- Time is the most important issue

Time Complexity of an Algorithm

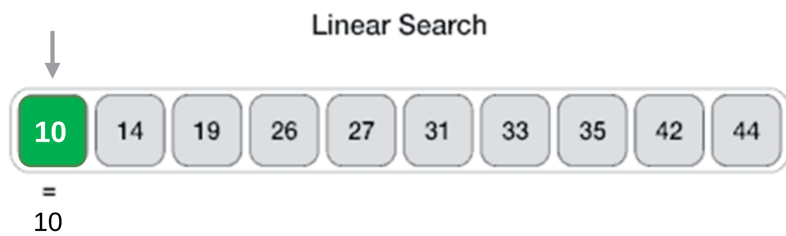
- Time complexity of an algorithm is the amount of time (or the number of steps) needed by a program to complete its task (i.e. to execute a particular algorithm)
- Compilation Time: Time taken to compile an algorithm
- Run Time: It is the time to execute the compiled program. The run time of an algorithm depend upon the number of instructions present in the algorithm. Therefore, it is in the control of the programmer.

Time Complexity of an Algorithm

- Hardly ever true that algorithm takes same time on different instances of same size
- Choice: best, worse or average case analysis
- Best case analysis
  - Time taken on best input of size *n*
- Worst case analysis
  - Time taken on worst input of size *n*
- Average case analysis
  - Time taken as *average* of time taken on inputs of size *n*

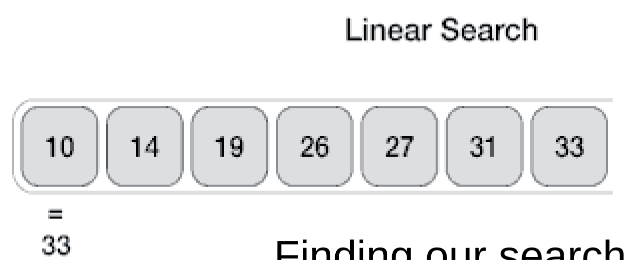
Time Complexity of an Algorithm

- Best case



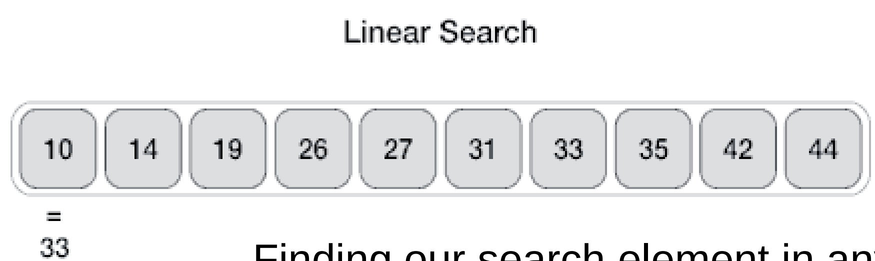
Finding our search element at the first position

- Worst case



Finding our search element at the last position

- Average case



Finding our search element in any position

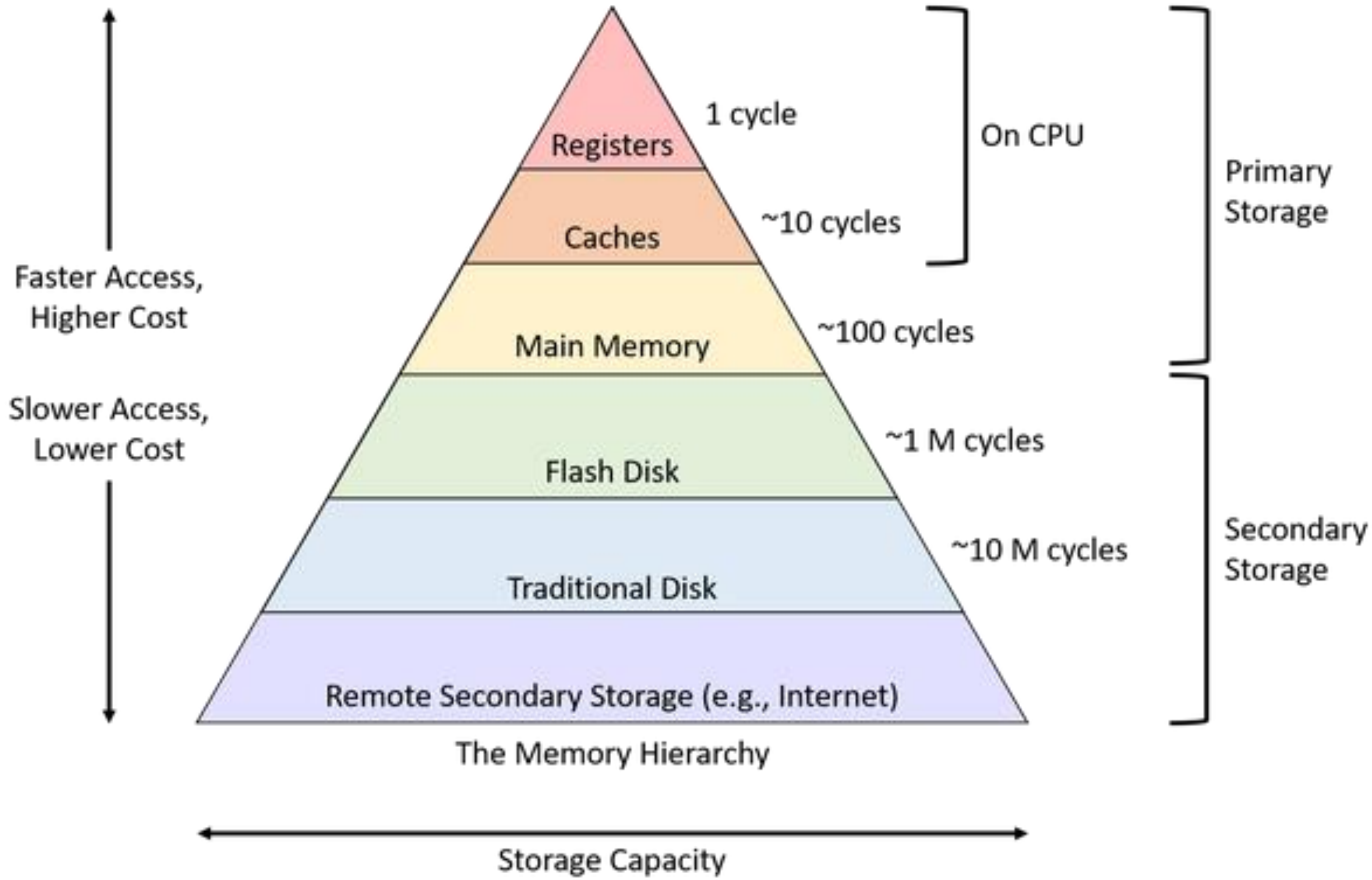
Which Case to Choose?

- Usually interested in worst case scenario:
  - The most operations that might be made for some problem size
- Worst case is only *safe* analysis – guaranteed upper bound (best case too optimistic)
- Average case analysis harder
  - Usually have to assume some probability distribution of the data
  - E.g. if looking for a specific letter in a random list of letters, might expect letter to appear 1/26 of time

Space Complexity

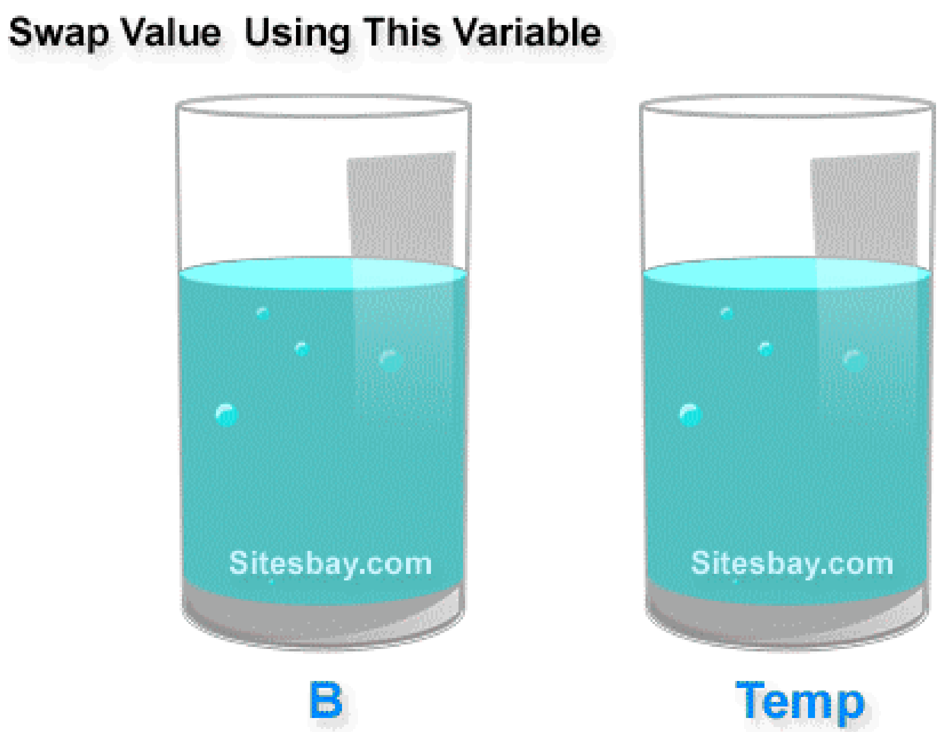
- Space complexity of a program is the amount of memory it consumes until it completes its execution
- A fixed amount of memory occupied by the space for the program and space occupied by the variables used in the program

Memory Hierarchy Diagram



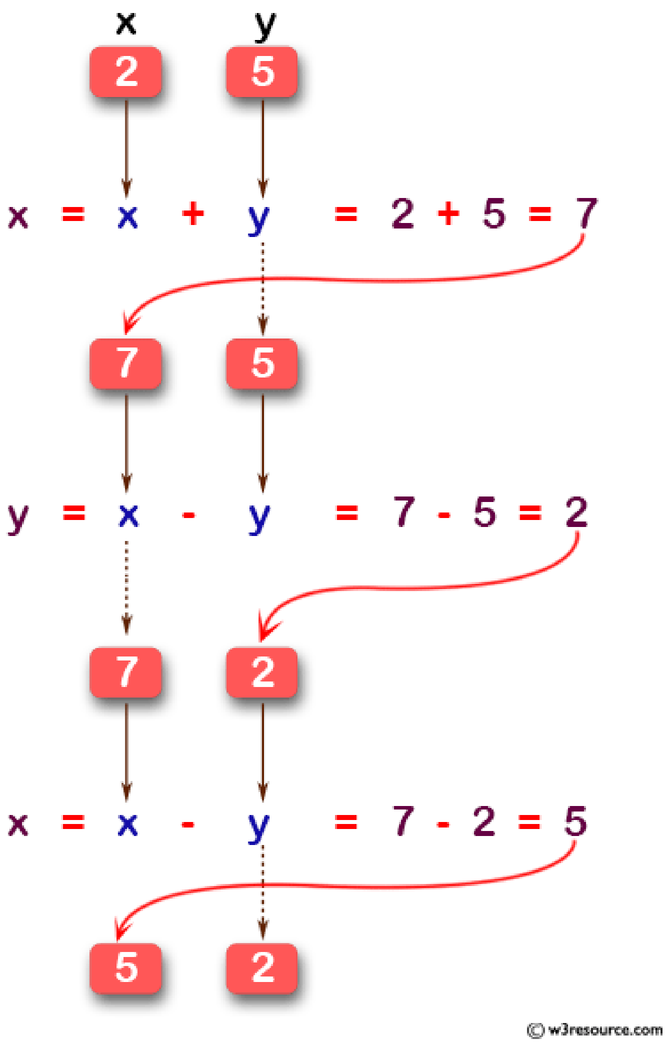
Space Complexity

Program to Swap Two Numbers



Space Complexity

Program to Swap Two Numbers
Without Third Variable



Space Complexity

Programming

```
a = a + b;
b = a - b;
a = a - b;
```

```
a,b = b,a
```

How to Choose Algorithms?

- What metric should we used to judge algorithms:
  - Length of the program (lines of code)
  - Ease of programming (i.e. bugs, maintenance)
  - Memory required
  - Running time
- Running time is the dominant standard
  - Quantifiable and easy to compare
  - Often the critical bottleneck

How do Estimate the Running Time?

- Characterize the size of the input
  - Input is an array containing at least *n* integers
  - Thus, size of input is *n*
- Count how many operations (steps) are taken in the algorithm for an input of size *n*
  - 1 step is an elementary operation
  - +, <, a[j] (indexing into an array), =, …

How do we Analyze an Algorithm?

A simple Example :

```
// Input: int A[N], array of N integers
// Output: Sum of all numbers in array A

int Sum(int A[], int N)
{
   int s=0;
   for (int i=0; i< N; i++)
      s = s + A[i];
   return s;
}
```

How should we analyse this?

How do we Analyze an Algorithm?

```
int sum(int A[], int N) {
    int s=0;
    for(int i=0; i<N; i++ )
        s = s + A[i] ;
    return(s);
}
```

- only happen once (#3)
- happen once *for each iteration* of loop (#5)

How do we Analyze an Algorithm?

- only happen once (so 3 such operations)
- happen once *for each iteration* of loop (5 operations)
- Therefore, in loop total operations: (5 * *n*)
- Total operations of this code segment: 5*n* + 3
- Complexity function: f(n) = 5*n* + 3 (Linear Equation)

O(*n*)

How does size affect running time?

- $5n + 3$ is an estimate of running time for different values of n

n	Operations
10	53
100	503
1000	5003
1000000	5000003

- As n grows, number of operations grows in *linear* proportion to n, for this sum function

Another Example

```
int sum=0;
for (int j=0; j<100; j++)
    sum = sum + j;
```

- Loop executes 100 times
- 4 = O(1) steps per iteration
- Total time here is

  100 * O(1) = O(100* 1) = O(1)

- Thus, faster than previous loop, for values up to 100

Another Example

```
int Sum(int A[], int N)
{
  int s=0;
  for (int i=0; i< N; i++)
    s = s + A[i];
  return s;
}
```

$O(n)$

```
int sum=0;
for (int j=0; j<100; j++)
  sum = sum + j;
```

$O(1)$

Take the higher order

Total time complexity is:

$= O(n) + O(1)$

$= O(n)$

```
printf("Sum is now %d",sum);
```

What is the time complexity is? $O(1)$

Another Example

```
for (i=0; i<n; i++) {
   for (j=0; j<m; j++) {
      sequence of statements
  }
}
```

- Outer loop executes *n* times
- For every outer, inner executes *m* times
- Thus, inner loop total = *n* * *m* times
- Complexity is O(*n***m*)

Practice Test#1

```
for (i=0; i<n; i++) {
   for (j=0; j<n; j++) {
      sequence of statements
   }
}
```

- What is the time complexity of the above code segment?

Practice Test#2

```
for (j=0; j<n; j++)
    for (k=0; k<n; k++)
        sum = sum + j*k;

for (l=0; l<n; l++)
    sum = sum -1;

printf(“Sum is now %d”,sum);
```

$n^2 + n$

$O(n^2)$

- What is the time complexity of the above code segment?

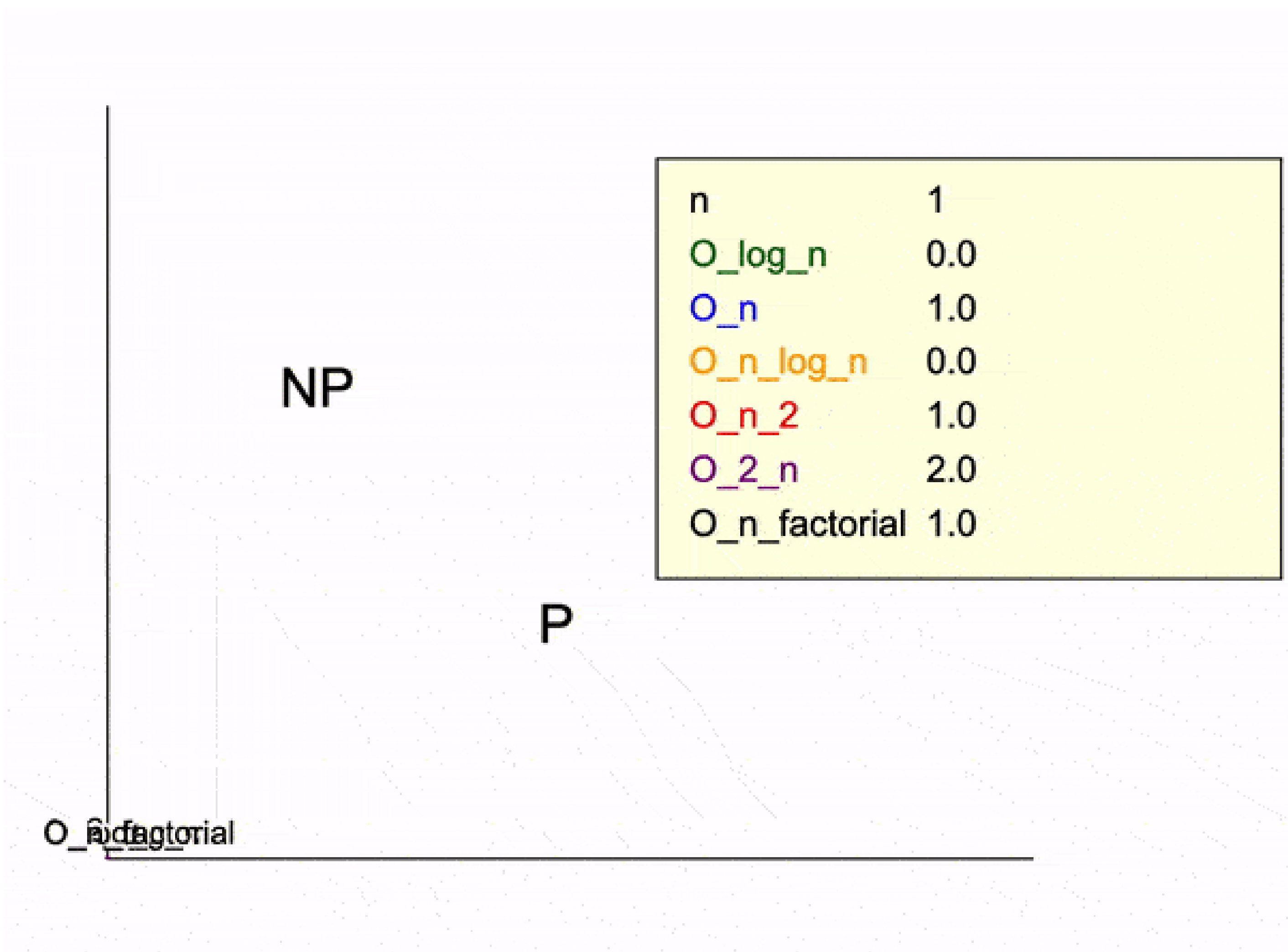
Common Orders of Growth

- Let n be the input size, and b and k be constants

$O(k) = O(1)$	Constant Time
$O(\log_b n) = O(\log n)$	Logarithmic Time
$O(n)$	Linear Time
$O(n \log n)$	
$O(n^2)$	Quadratic Time
$O(n^3)$	Cubic Time
...	
$O(k^n)$	Exponential Time
$O(n!)$	Exponential Time

Increasing Complexity

Common Orders of Growth



Time Limit Exceeded (TLE)

- Number of operations that are allowed to perform within a second
- Most of the sites these days allow 10^8 operations per second
- After figuring out the number of operations that can be performed, search for the right complexity by looking at the constraints given in the problem

Time Limit Exceeded (TLE)

Example:

Given an array A[ ] and a number x, check for a pair in A[ ] with the sum as x. where N is:

1) $1 <= N <= 10^3$

2) $1 <= N <= 10^5$

3) $1 <= N <= 10^8$

Time Limit Exceeded (TLE)

1) $1 <= N <= 10^3$

2) $1 <= N <= 10^5$

3) $1 <= N <= 10^8$

For Case 1:

A naive solution that is using two for-loops works as it gives us a complexity of $O(N^2)$, which even in the worst case will perform 10^6 operations which are well under 10^8. Of course $O(N)$ and $O(N \log N)$ is also acceptable in this case.

Time Limit Exceeded (TLE)

1) $1 <= N <= 10^3$

2) $1 <= N <= 10^5$

3) $1 <= N <= 10^8$

For Case 2:

We have to think of a better solution than $O(N^2)$, as in worst case, it will perform 10^{10} operations as N is 10^5. So complexity acceptable for this case is either O(NlogN) which is approximately 10^6 (10^5 * ~10) operations well under 10^8 or O(N).

Time Limit Exceeded (TLE)

```
1) 1 <= N <= 10^3
2) 1 <= N <= 10^5
3) 1 <= N <= 10^8
```

For Case 3:

Even O(N logN) gives us TLE as it performs ~10^9 operations which are over 10^8. So the only solution which is acceptable is O(N) which in worst case will perform 10^8 operations.

Time Limit Exceeded (TLE)

Common time complexities

Let n be the main variable in the problem.

- If $n \leq 12$, the time complexity can be O(n!).
- If $n \leq 25$, the time complexity can be $O(2^n)$.
- If $n \leq 100$, the time complexity can be $O(n^4)$.
- If $n \leq 500$, the time complexity can be $O(n^3)$.
- If $n \leq 10^4$, the time complexity can be $O(n^2)$.
- If $n \leq 10^6$, the time complexity can be O(n log n).
- If $n \leq 10^8$, the time complexity can be O(n).
- If $n > 10^8$, the time complexity can be O(log n) or O(1).

Time Limit Exceeded (TLE)

Examples of each common time complexity

- O(n!) [Factorial time]: Permutations of 1 ... *n*
- $O(2^n)$ [Exponential time]: Exhaust all subsets of an array of size *n*
- $O(n^3)$ [Cubic time]: Exhaust all triangles with side length less than *n*
- $O(n^2)$ [Quadratic time]: Slow comparison-based sorting (eg. Bubble Sort, Insertion Sort, Selection Sort)
- O(n log n) [Linearithmic time]: Fast comparison-based sorting (eg. Merge Sort)
- O(n) [Linear time]: Linear Search (Finding maximum/minimum element in a 1D array), Counting Sort
- O(log n) [Logarithmic time]: Binary Search, finding GCD (Greatest Common Divisor) using Euclidean Algorithm
- O(1) [Constant time]: Calculation (eg. Solving linear equations in one unknown)

Why Measure Efficiency!!

- Many ways to solve a problem.
  - but which way (i.e. algorithm) is better

- Moore's Law : Number of transistors on CPU doubles every year (more precisely every 18 months)
  - i.e. System performance doubles (well almost) every 18 months or so.

- If this is so and with current speeds of cpu's these days, then why bother worrying about how efficient our code is?

A Classic Optimization Problem

- The Travelling Salesman problem:
  "*A travelling salesman has to visit 100 different locations in a town what is the shortest route that he can take?*"

- Total number of distinct routes possible : $100! \approx 30^{100}$

- What does this mean in terms of running time?
  A supercomputer capable of checking 100 billion routes per second can check roughly about 10^{20} routes in the space of one year.

- Millions of years needed to check all routes!

Code Optimization

- Scale down the Time Complexity of a code segment

```
float sum = 0;
for (int i = 1; i<=N; i++)
       sum = sum + i;
```

$O(n)$

Considering → N=7

```
float sum = 0;
sum = sum + 1;
sum = sum + 2;
sum = sum + 3;
sum = sum + 4;
sum = sum + 5;
sum = sum + 6;
sum = sum + 7;
```

Code Optimization

- Scale down the Time Complexity of a code segment

```
float sum = 0;
for (int i = 1; i<=N; i++)
       sum = sum + i;
```

$O(n)$

```
Sum = (N*(N+1))/2
```

$O(1)$


Thank You!