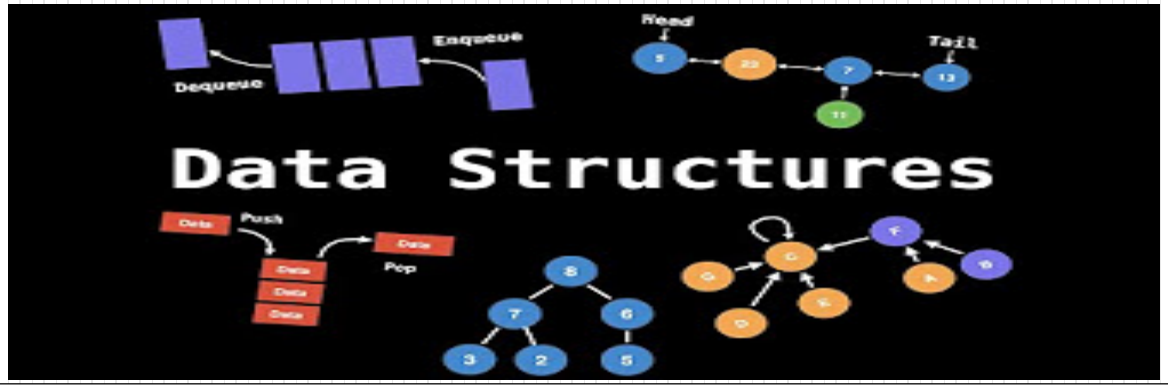
Data Structures

Dr. Bibhudatta Sahoo
Communication & Computing Group
Department of CSE, NIT Rourkela
Email: bdsahu@nitrkl.ac.in, **9937324437**, 2462358



Why data structures?

- **Data structure**, way in which data are **stored** for efficient **search** and **retrieval**.
- Different data structures are suited for different problems. Some data structures are useful for simple general problems, such as retrieving data that has been stored with a specific identifier.
- For example, an online dictionary can be structured so that it can retrieve the definition of a word.
- On the other hand, specialized data structures have been devised to solve complex specific **search problems**.

- **Definition**: A data structure is a way of storing and managing data.

- **Definition:** A **data structure** is a predefined format for efficiently storing, accessing, and processing data in a computer program.

Data structures

- In computer science, a **data structure** is a data **organization**, **management**, and **storage format** that enables efficient access and modification in **main memory**.
- More precisely, a data structure is a collection of **data values**, the **relationships** among them, and the **functions** or **operations** that can be applied to the data.

A = 3 : 108

B = 2 : 100

C : 124

C = A + B

Address	Value
100	2
108	3
116	
124	5

Data structures

- In computer science, a **data structure** is a data **organization**, **management**, and **storage format** that enables efficient access and modification in **main memory**.
- More precisely, a data structure is a collection of **data values**, the **relationships** among them, and the **functions** or **operations** that can be applied to the data.

A = 3 : 108

B = 2 : 100

C : 124

C = A + B

Address	Value	Link
100	2	108
108	3	116
116	9	124
124	5	NULL

File structures

- File Structures is **the Organization of Data in Secondary Storage Device** in such a way that minimize the access time and the storage space.
- A File Structure is a combination of representations for data in files and of operations for accessing the data.
- A File Structure allows applications to read, write and modify data.

Data Structures vs File structures

- Both involves: **representation of data**

  \+

  operations of accessing data

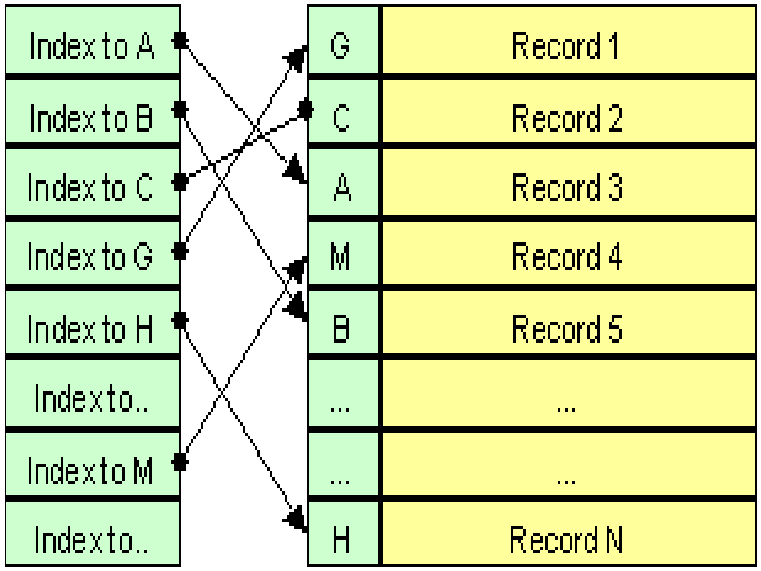
- Difference:

[1] **Data structures deals with the data in main memory**

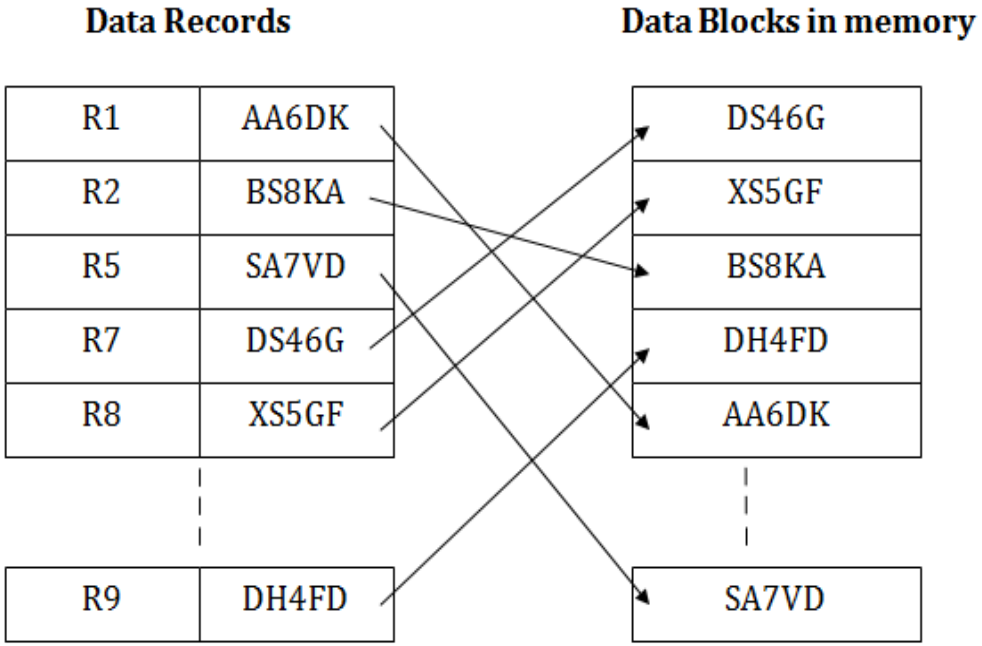
[2] **File Structures deals with the data in secondary memory**

File Structures

- File Structures: **Pile, Sequential, Indexed Sequential, Direct access, Inverted files**; Indexing structures- B-tree and its variations.
- File Structures: Pile, Sequential, Indexed Sequential, Direct access, Inverted files; Indexing structures- B-tree and its variations.



Indexed File Organisation



Data Structures

1. Domain ($\mathcal{D}$): This is the range of values that the data may have. This domain is also termed data object.
2. Function ($\mathcal{F}$): This is the set of operations which may legally be applied to elements of the data object. This implies that for a data structure, we must specify the set of operations.
3. Axioms ($\mathcal{A}$): This is the set of rules with which the different operations belonging to $\mathcal{F}$ can actually be implemented.

Now we can define the term data structure.

A *data structure* D is a triplet, that is, D = ($\mathcal{D}$, $\mathcal{F}$, $\mathcal{A}$) where $\mathcal{D}$ is a set of data object, $\mathcal{F}$ is a set of functions and $\mathcal{A}$ is a set of rules to implement the functions. Let us consider an example.

We know that for the integer data type (int) in the C programming language the structure includes the following types:

$\mathcal{D}$ = (0, ±1, ±2, ±3, ...)

$\mathcal{F}$ = (+, –, *, /, %)

$\mathcal{A}$ = (A set of binary arithmetics to perform addition, subtraction, division, multiplication, and modulo operations.)

Need For Data Structure In Programming

Typically, at any time, the application may face the following hurdles:

#1) Searching Large amounts of Data:

- With a large amount of data being processed and stored, at any given time our program may be required to search a particular data. If the data is too large and not organized properly, it will take a lot of time to get the required data.
- When we use data structures to store and organize data, the retrieval of data becomes faster and easier.

#2) Speed of Processing:

- Disorganized data may result in slow processing speed as a lot of time will be wasted in retrieving and accessing data.
- If we organize the data properly in a data structure while storing, then we will not waste time in activities like retrieving, organizing it every time. Instead, we can concentrate on the processing of data to produce the desired output.

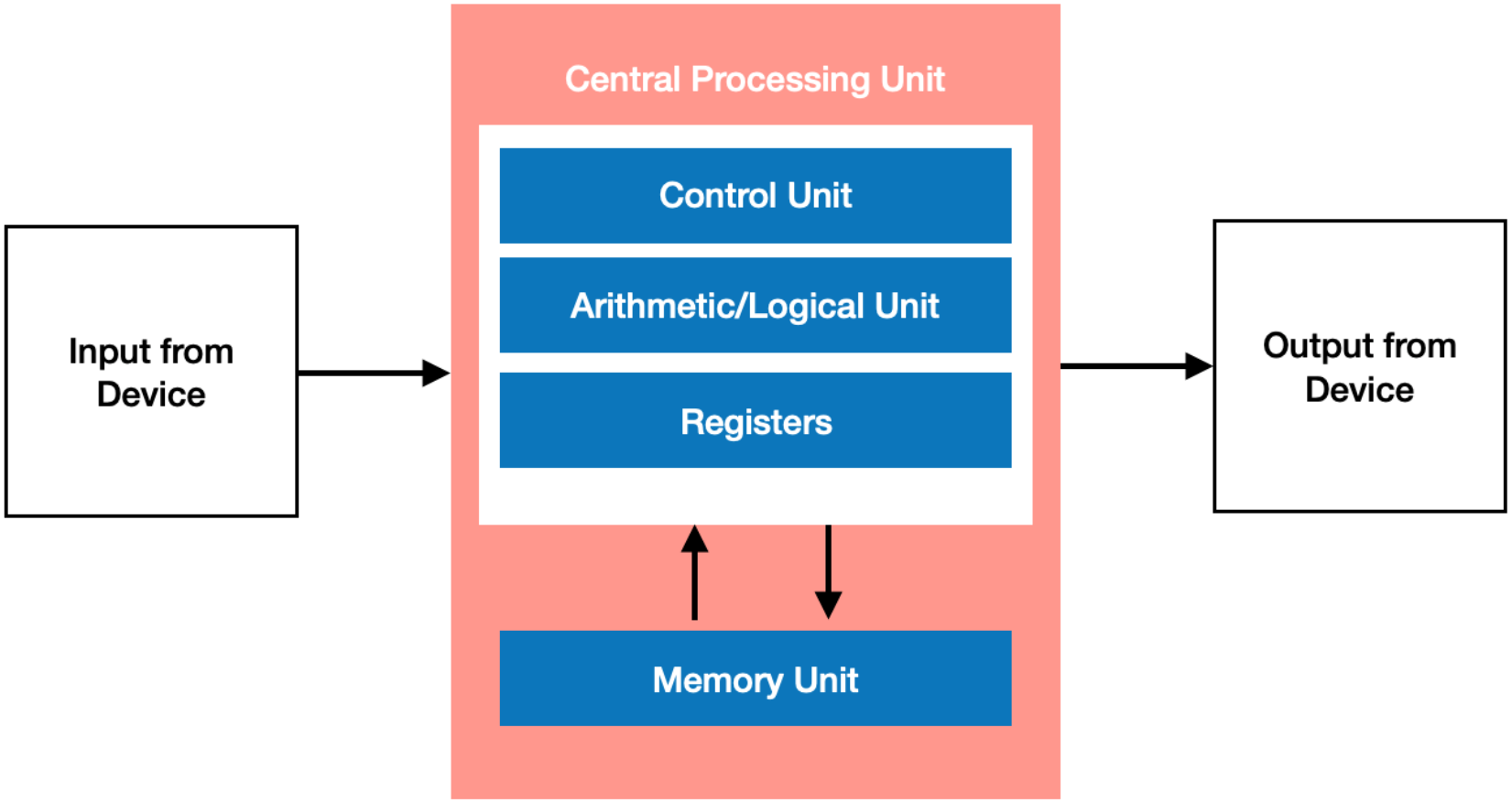
#3) Multiple Simultaneous Requests:

- Many applications these days need to make a simultaneous request to data. These requests should be processed efficiently for applications to run smoothly.
- If our data is stored just randomly, then we will not be able to process all the concurrent requests simultaneously. So it's a wise decision to arrange data in a proper data structure so as to minimize the concurrent requests turnaround time

The Machine architecture

Von Neumann architecture

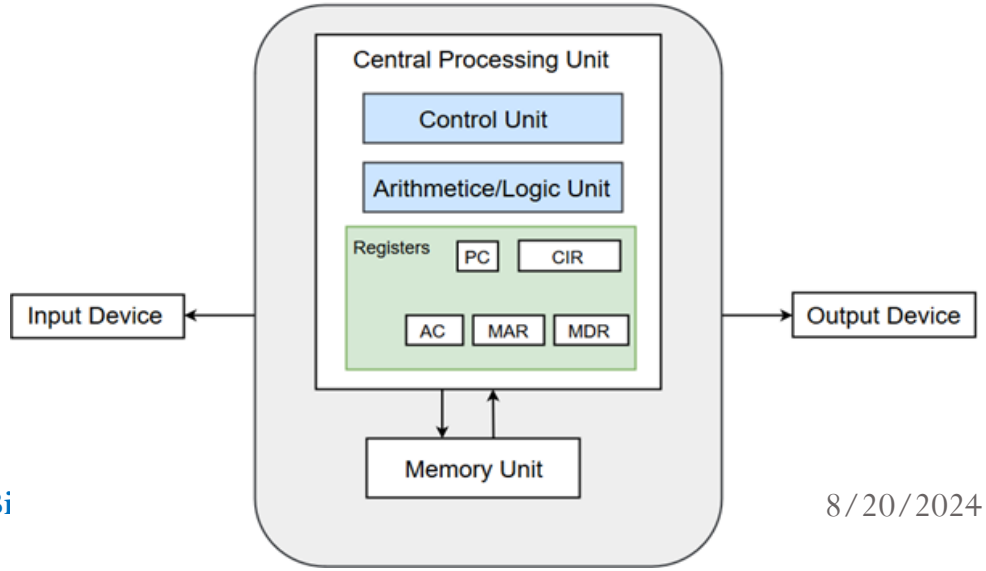
The **von Neumann architecture**—also known as the **von Neumann model** or **Princeton architecture**—is a **computer architecture** based on a 1945 description by John von Neumann and others in the *First Draft of a Report on the EDVAC*



Von-Neumann Model

- Von-Neumann proposed his computer architecture design in 1945 which was later known as Von-Neumann Architecture. It consisted of a Control Unit, Arithmetic, and Logical Memory Unit (ALU), Registers and Inputs/Outputs.
- Von Neumann architecture is based on the stored-program computer concept, where instruction data and program data are stored in the same memory. This design is still used in most computers produced today.

Von-Neumann Basic Structure:



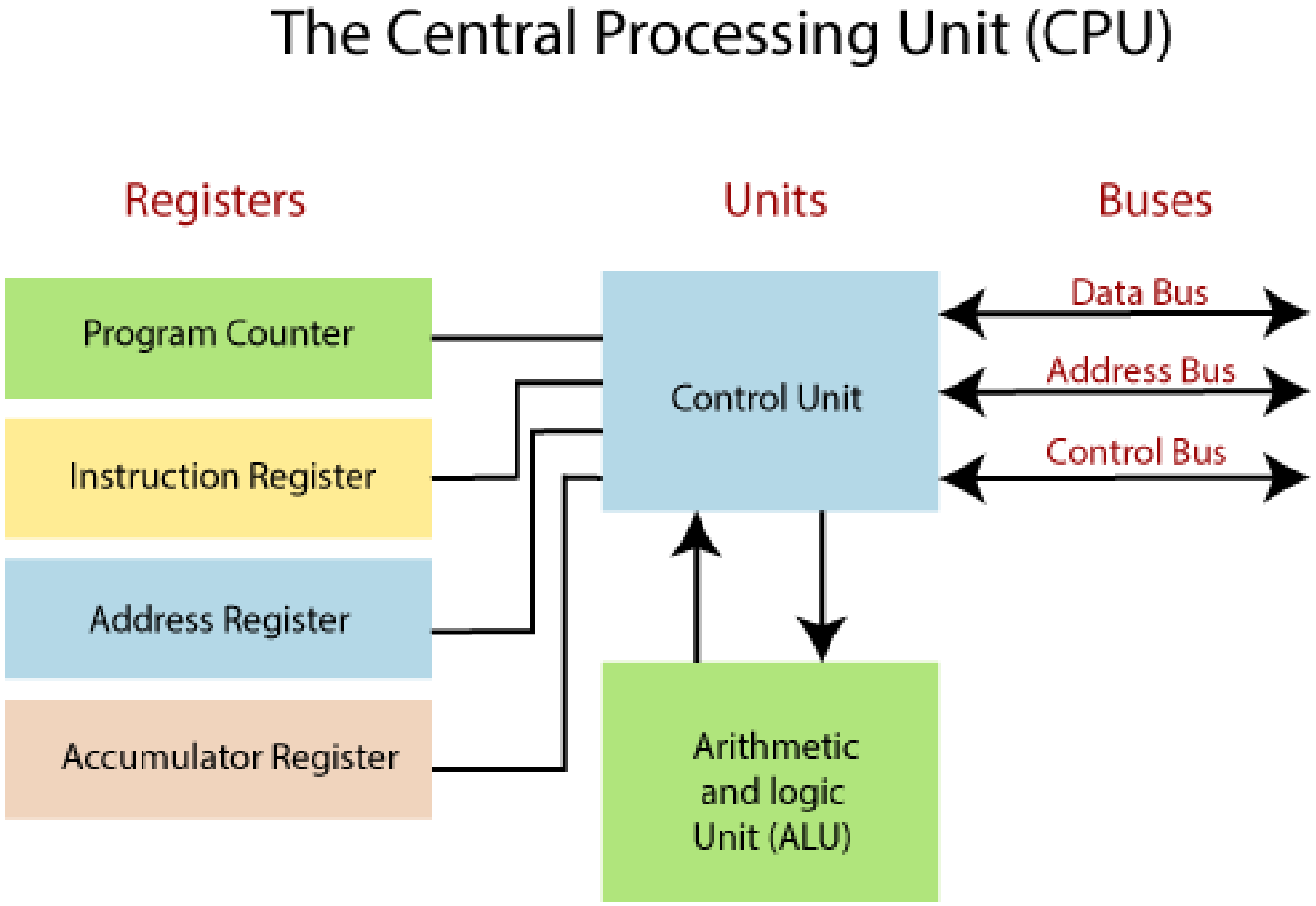
The Von Neumann architecture

- Von Neumann architecture is **the design upon which many general purpose computers are based**. The key elements of von Neumann architecture are: data and instructions are both stored as binary digits. data and instructions are both stored in primary storage.
- The Von Neumann architecture consists of **a single, shared memory for programs and data, a single bus for memory access, an arithmetic/logic unit, and a program control unit**.
- The Von Neumann processor operates fetching and execution cycles seriously.
- This sort of computer executes one instruction at a time in sequence.

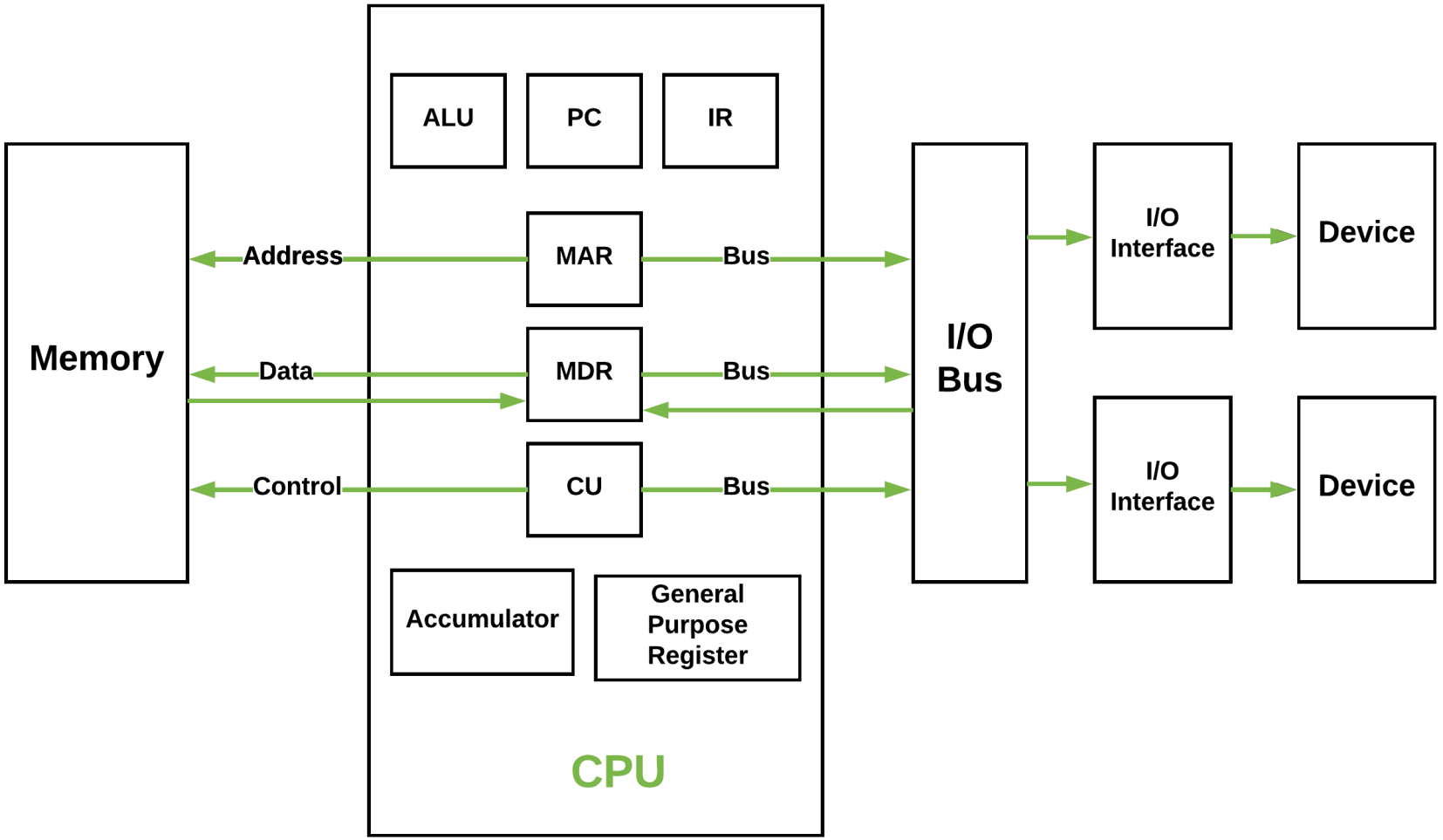
Von Neumann bottleneck

- Whatever we do to enhance performance, we cannot get away from the fact that instructions can only be done one at a time and can only be carried out sequentially. Both of these factors hold back the competence of the CPU.
- This is commonly referred to as the 'Von Neumann bottleneck'. We can provide a Von Neumann processor with more cache, more RAM, or faster components but if original gains are to be made in CPU performance then an influential inspection needs to take place of CPU configuration.

Von-Neumann Model



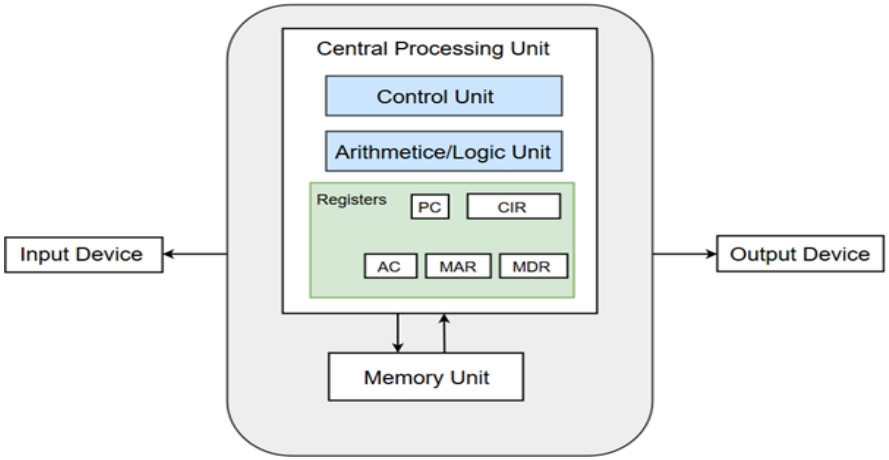
Von-Neumann Model



Von-Neumann Model

- **Control Unit –** A control unit (CU) handles all processor control signals. It directs all input and output flow, fetches the code for instructions and controlling how data moves around the system.
- **Arithmetic and Logic Unit (ALU) –** The arithmetic logic unit is that part of the CPU that handles all the calculations the CPU may need, e.g. Addition, Subtraction, Comparisons. It performs Logical Operations, Bit Shifting Operations, and Arithmetic Operation.

Von-Neumann Basic Structure:



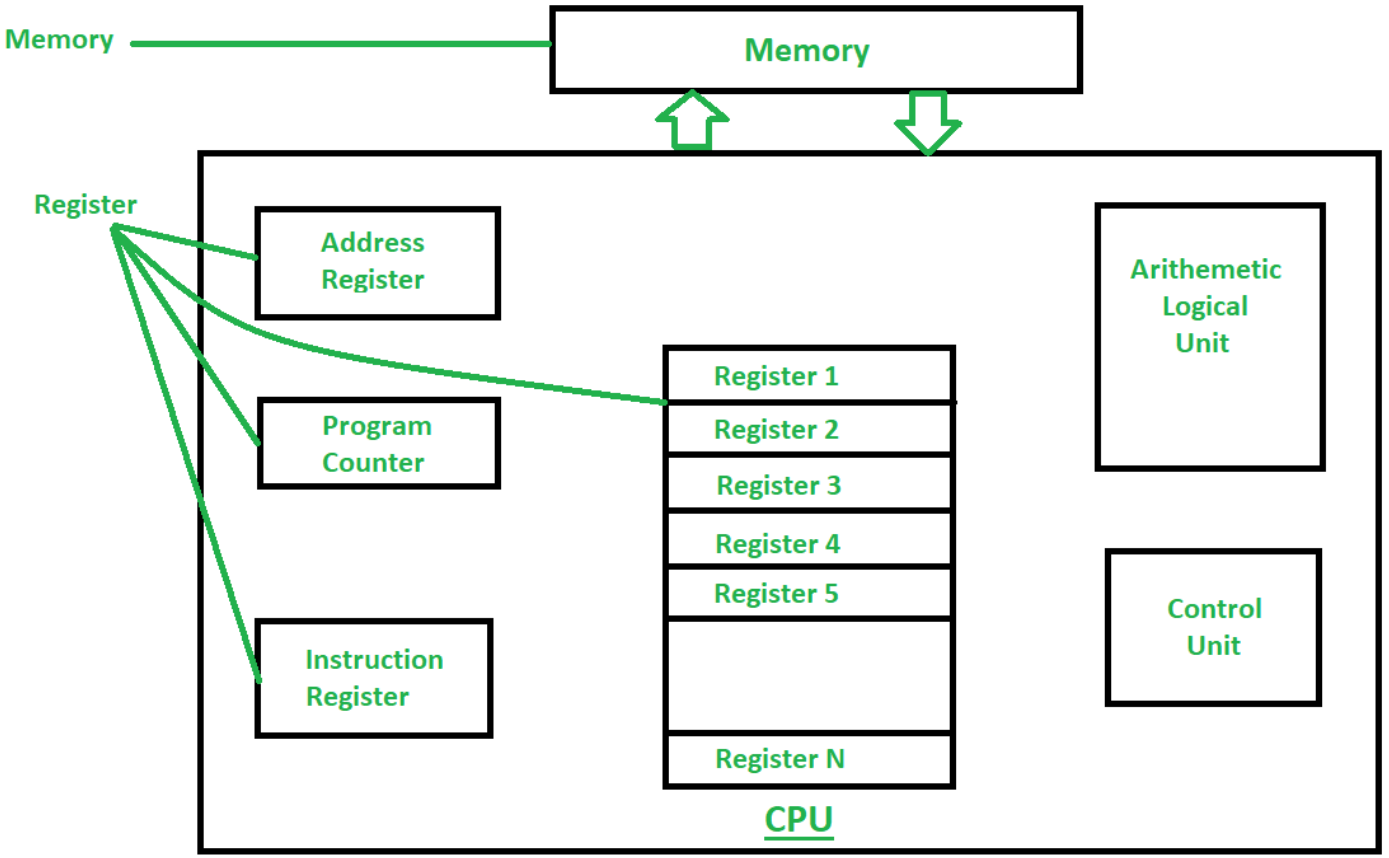
Von-Neumann Model: Main Memory Unit (Registers)

- **Accumulator:** Stores the results of calculations made by ALU.
- **Program Counter (PC):** Keeps track of the memory location of the next instructions to be dealt with. The PC then passes this next address to Memory Address Register (MAR).
- **Memory Address Register (MAR):** It stores the memory locations of instructions that need to be fetched from memory or stored into memory.
- **Memory Data Register (MDR) or MBR:** It stores instructions fetched from memory or any data that is to be transferred to, and stored in, memory.
- **Current Instruction Register (CIR) or IR:** It stores the most recently fetched instructions while it is waiting to be coded and executed.
- **Instruction Buffer Register (IBR):** The instruction that is not to be executed immediately is placed in the instruction buffer register IBR.

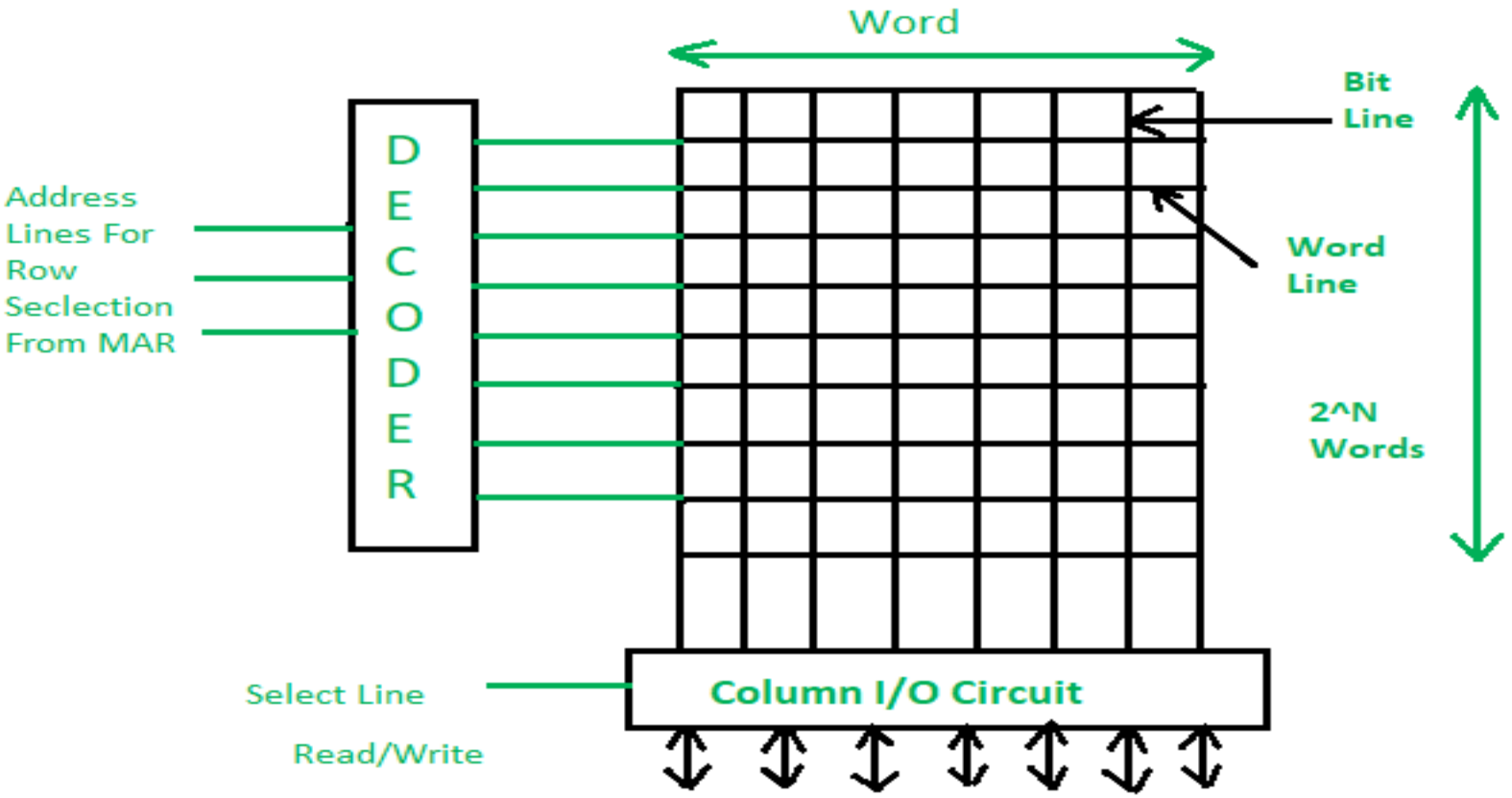
Von-Neumann Model

- **Input/Output Devices –** Program or data is read into main memory from the *input device* or secondary storage under the control of CPU input instruction. *Output devices* are used to output the information from a computer.
- **Buses –** Data is transmitted from one part of a computer to another, connecting all major internal components to the CPU and memory, by the means of Buses. Types:
  - **Data Bus:** It carries data among the memory unit, the I/O devices, and the processor.
  - **Address Bus:** It carries the address of data (not the actual data) between memory and processor.
  - **Control Bus:** It carries control commands from the CPU (and status signals from other devices) in order to control and coordinate all the activities within the computer.

Von Neuman Architecture

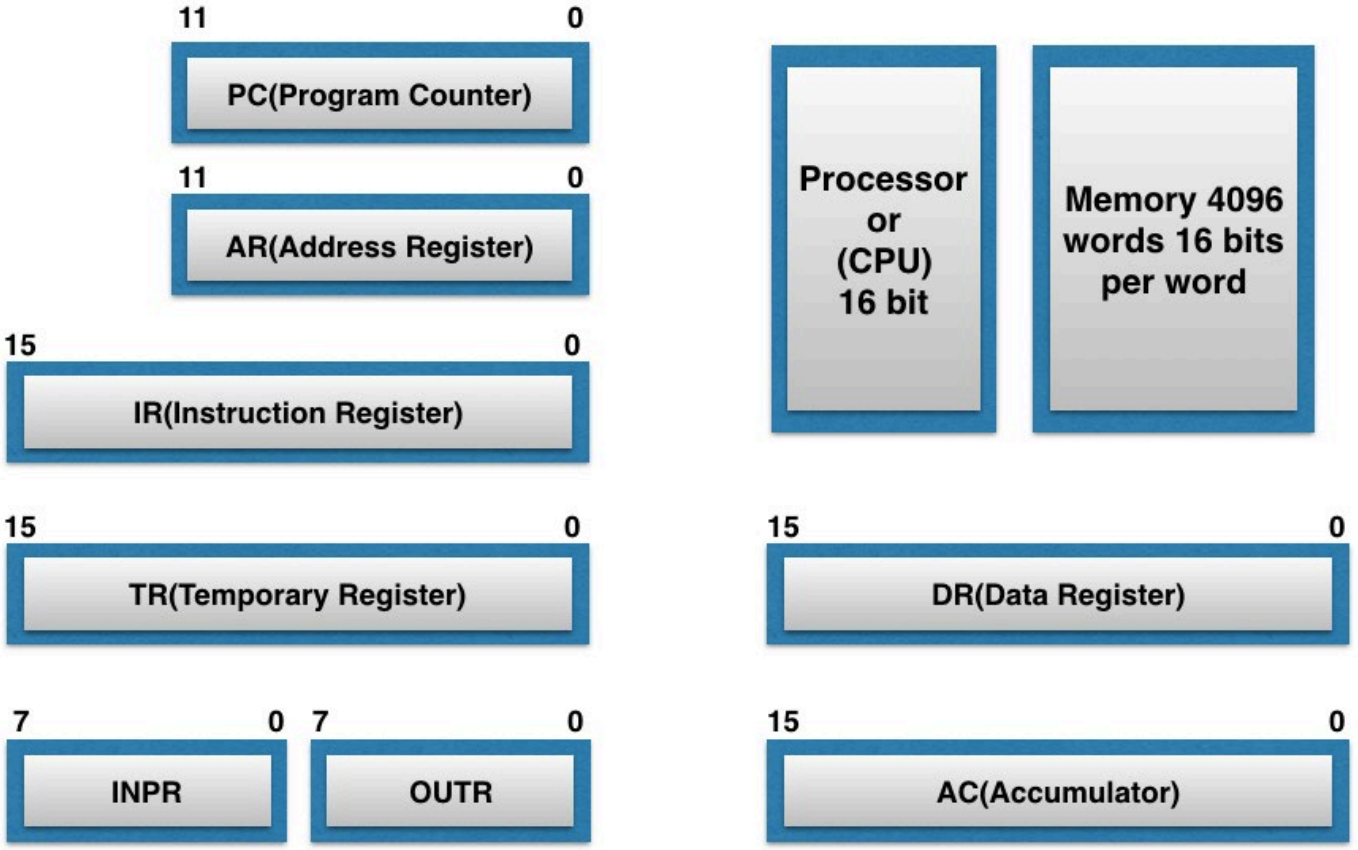


2D Memory organization



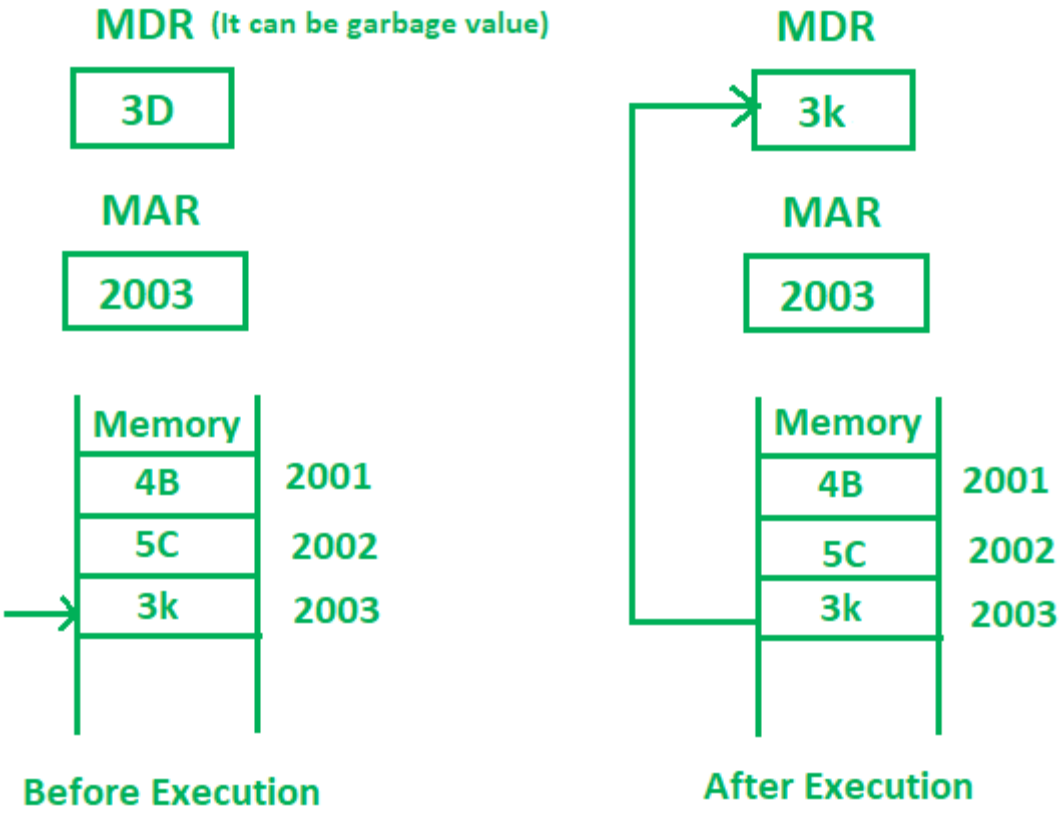
2D Memory Organization

Von-Neumann Model



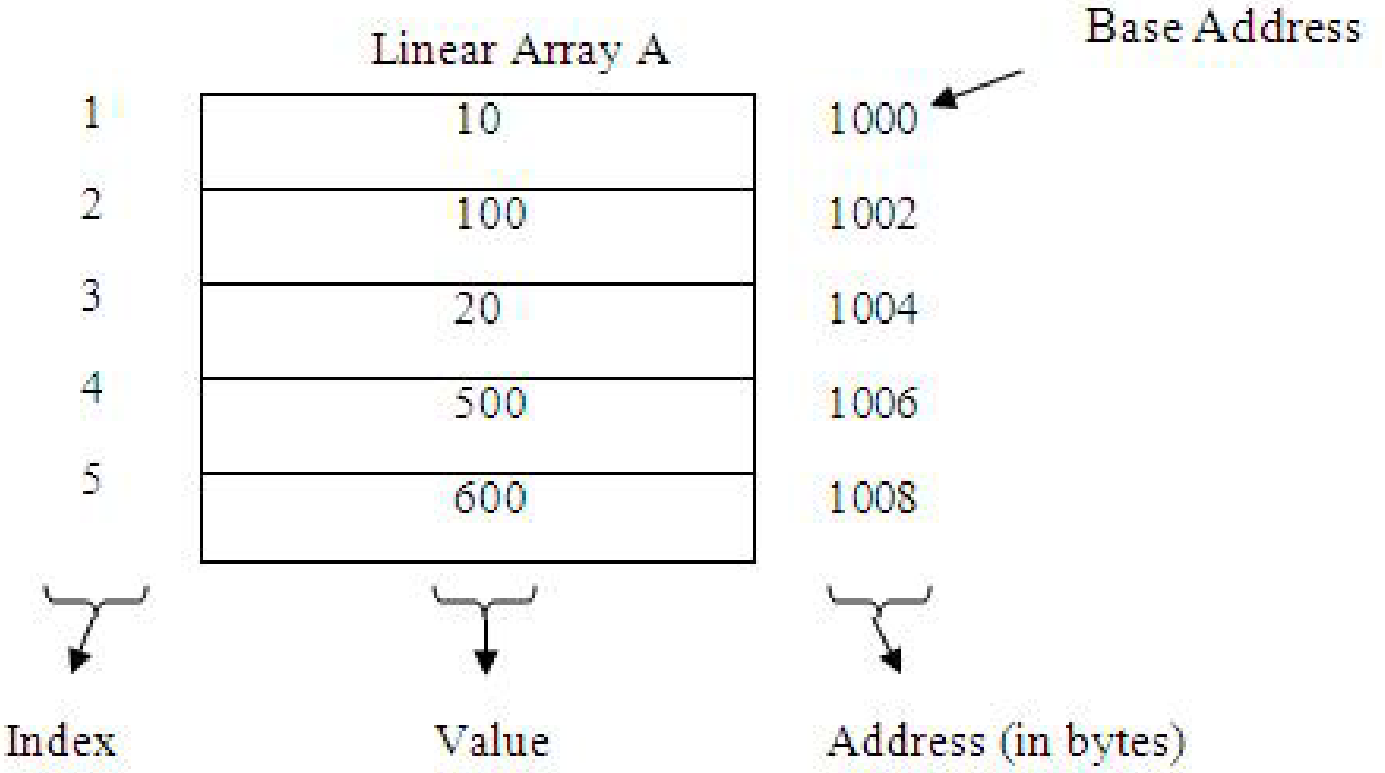
Basic Computer Registers and Memory

Memory Read Operation

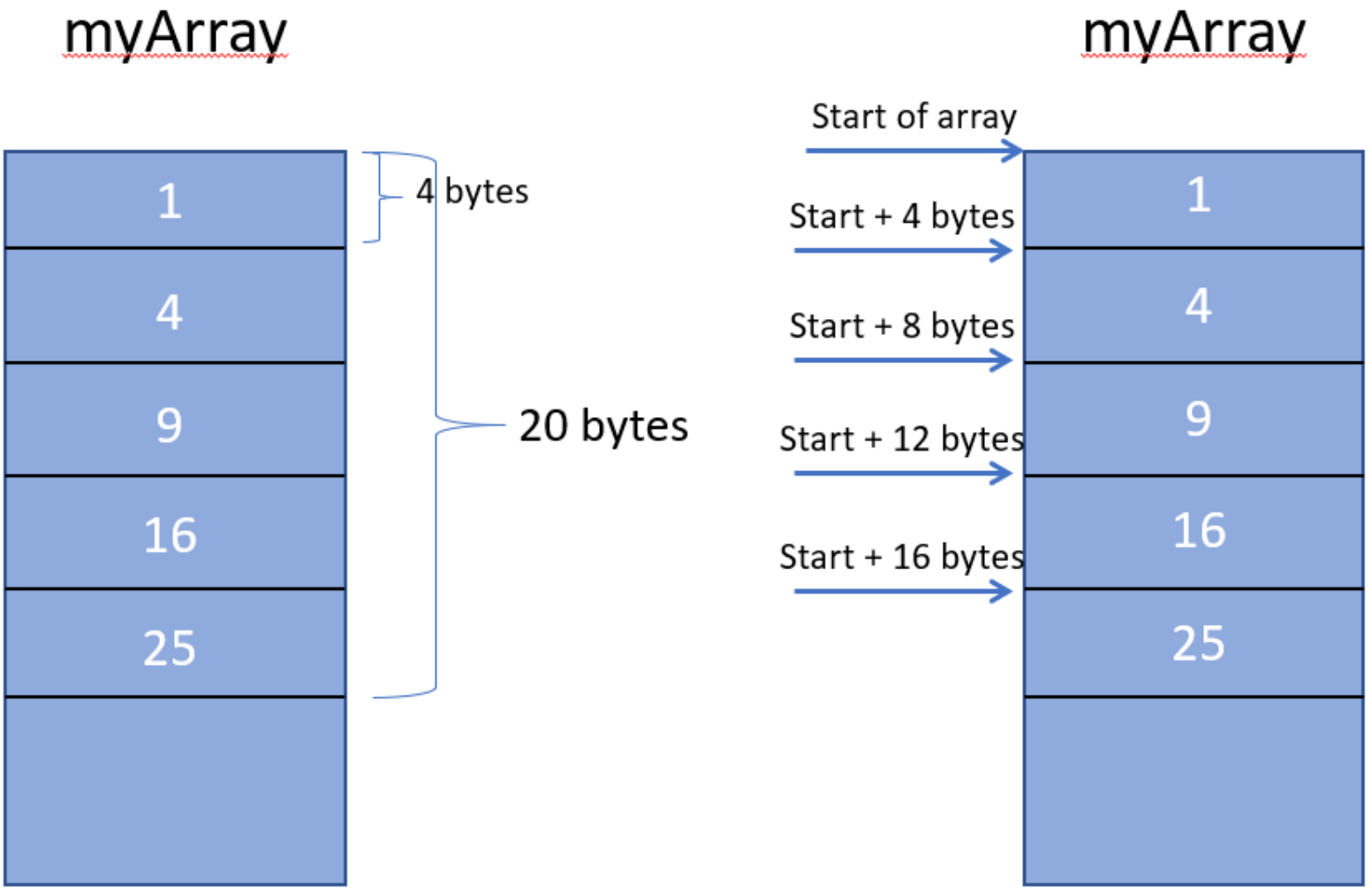


Memory Read Operation

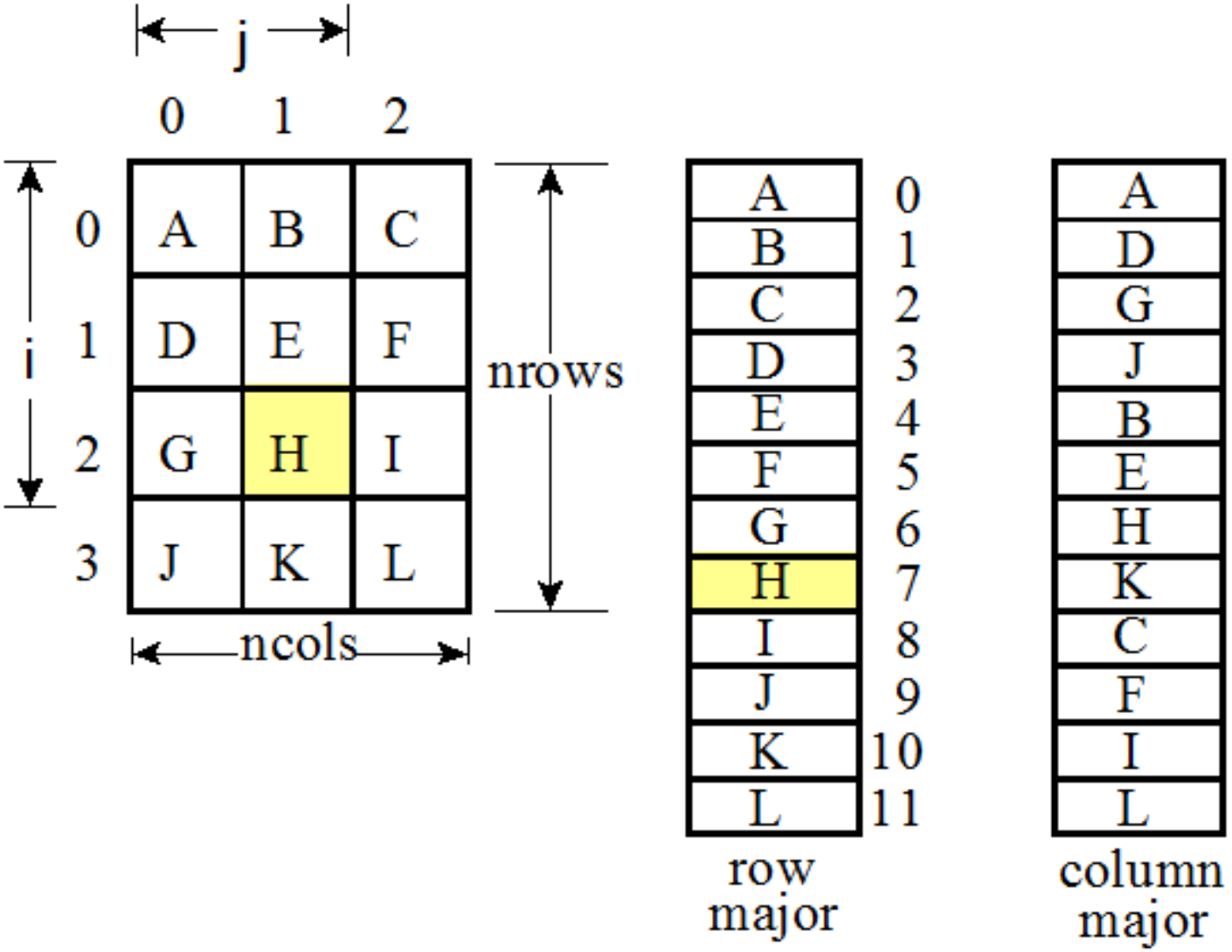
Representation of Array in main memory



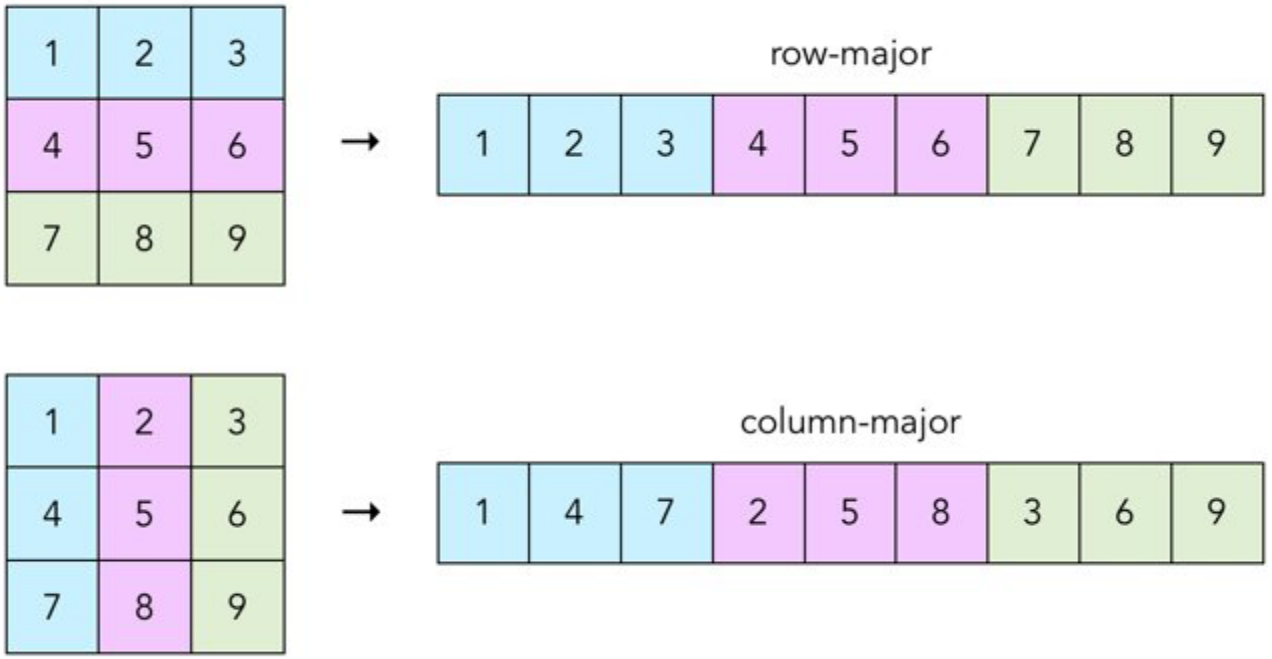
Representation of Array in main memory



Matrix representation in array

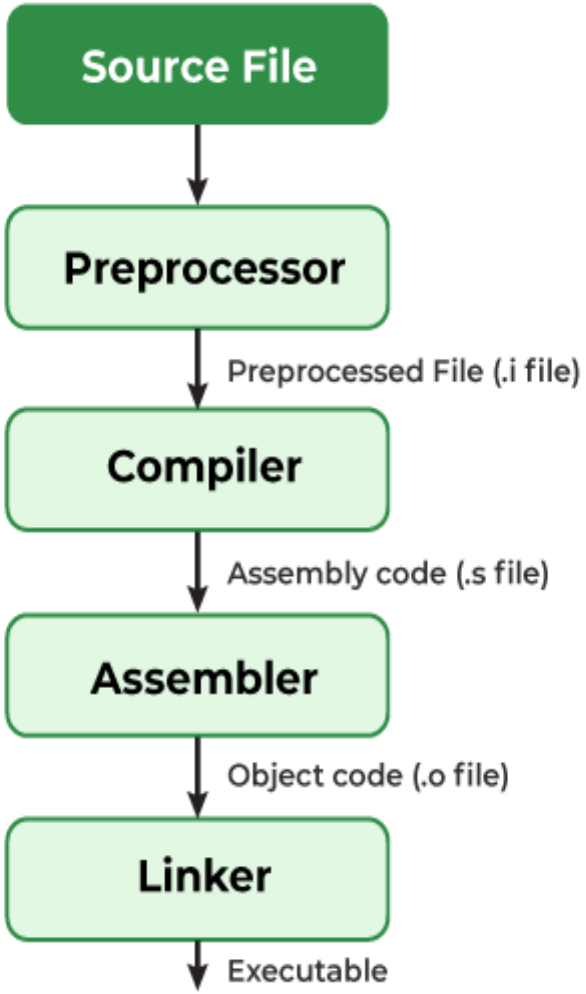


Matrix representation in array



Execution of

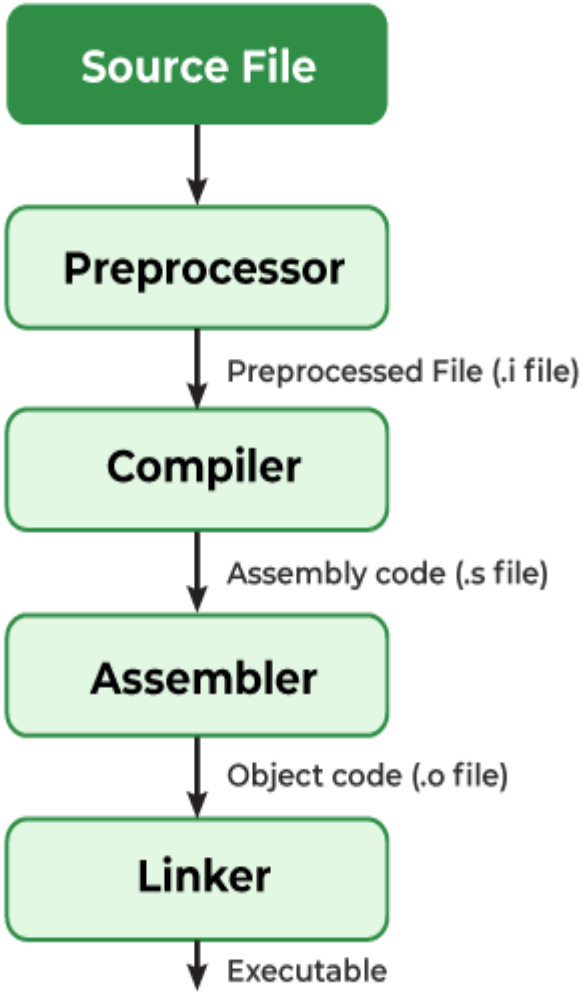
Compilation Process in C

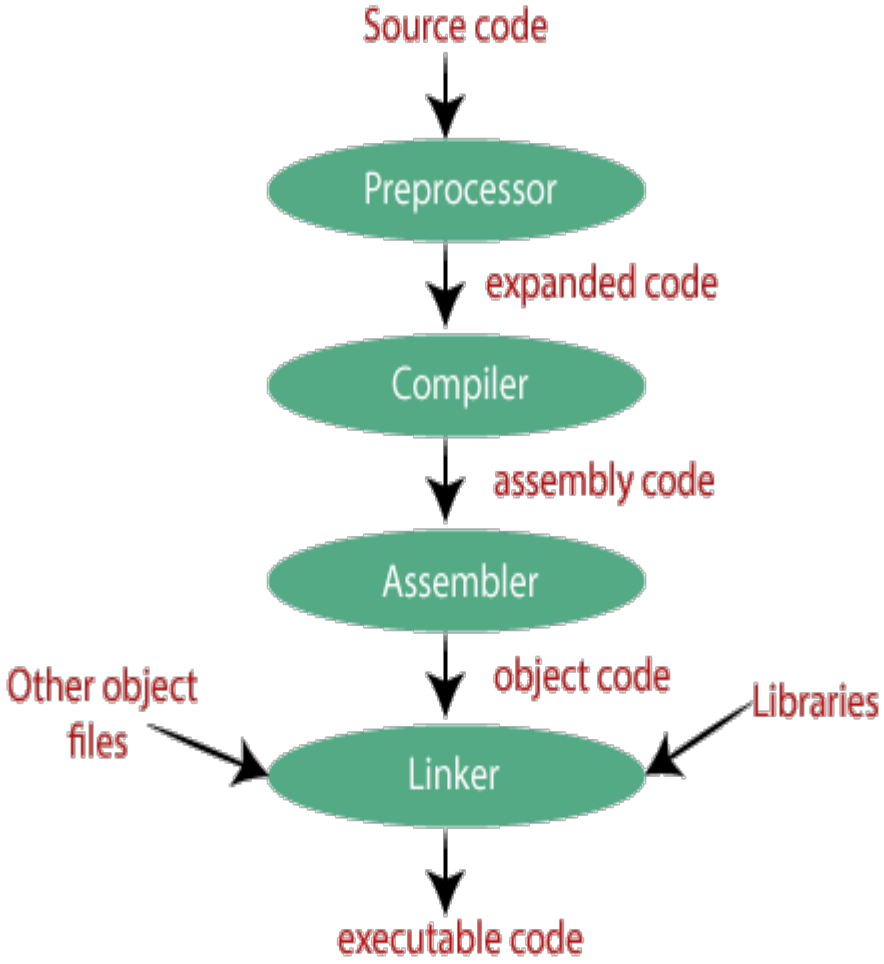



```
#include<stdio.h>
main()
{
printf("Hello javaTpoint");
return 0;
}
```

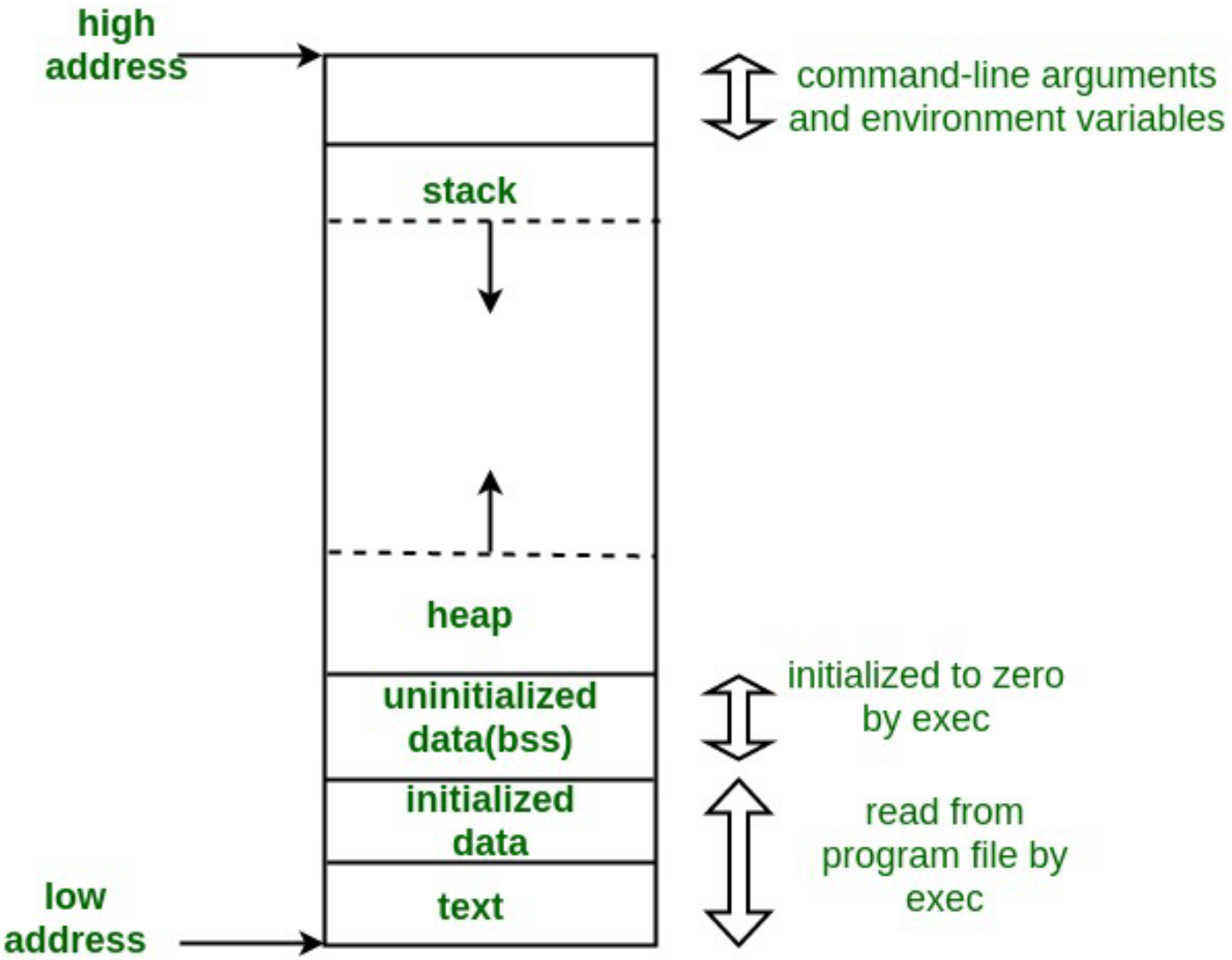
```
01000000000000
01111111111111
01010101101010
00000011111111
00000111111111
00000010101011
```

Compilation Process in C



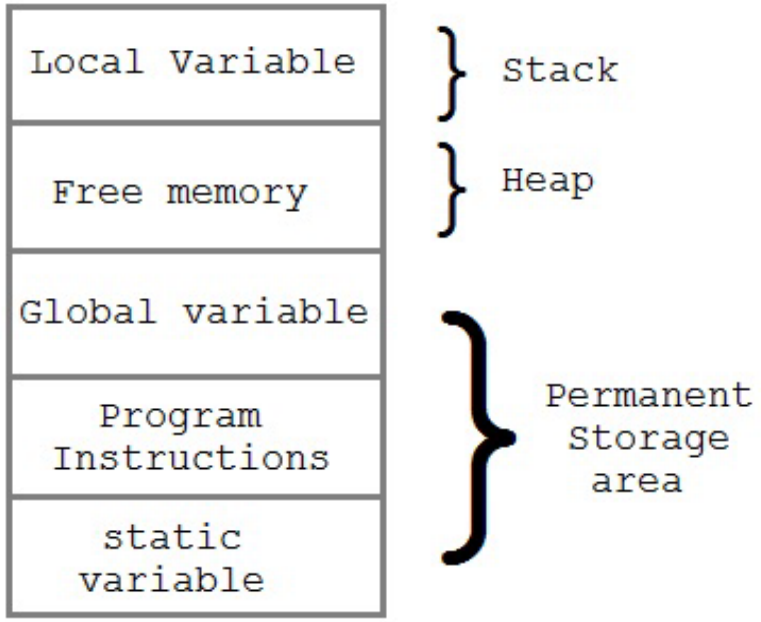


Memory layout of C program

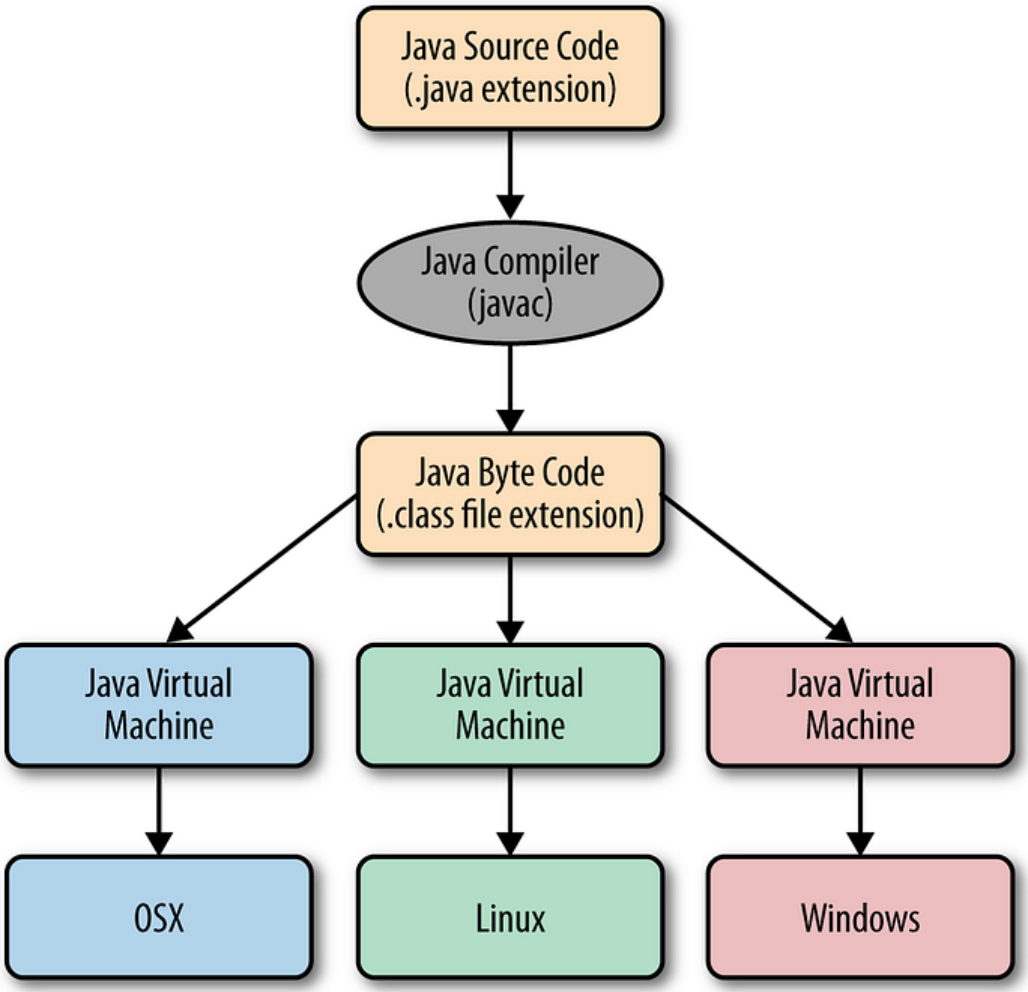


Memory allocation in C

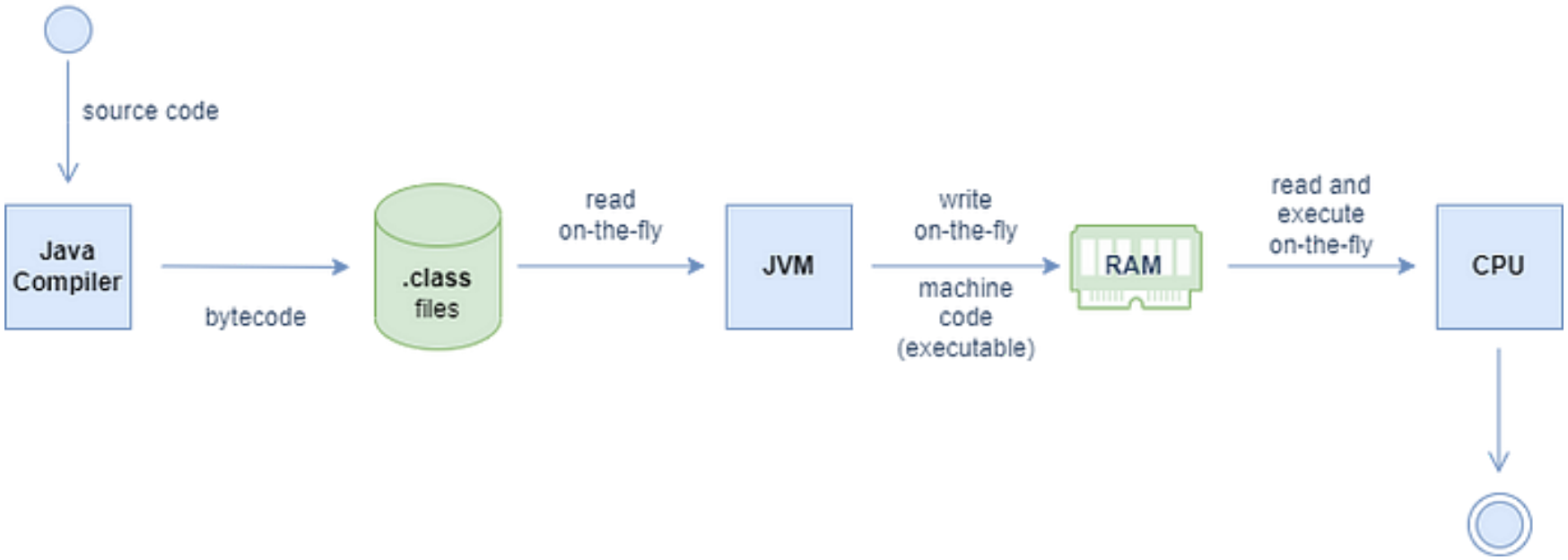
- **Stack memory** is allocated during compilation time execution and this is known as **static memory allocation**.
- **Heap memory** is allocated at run-time compilation. This is know as **dynamic memory allocation**.



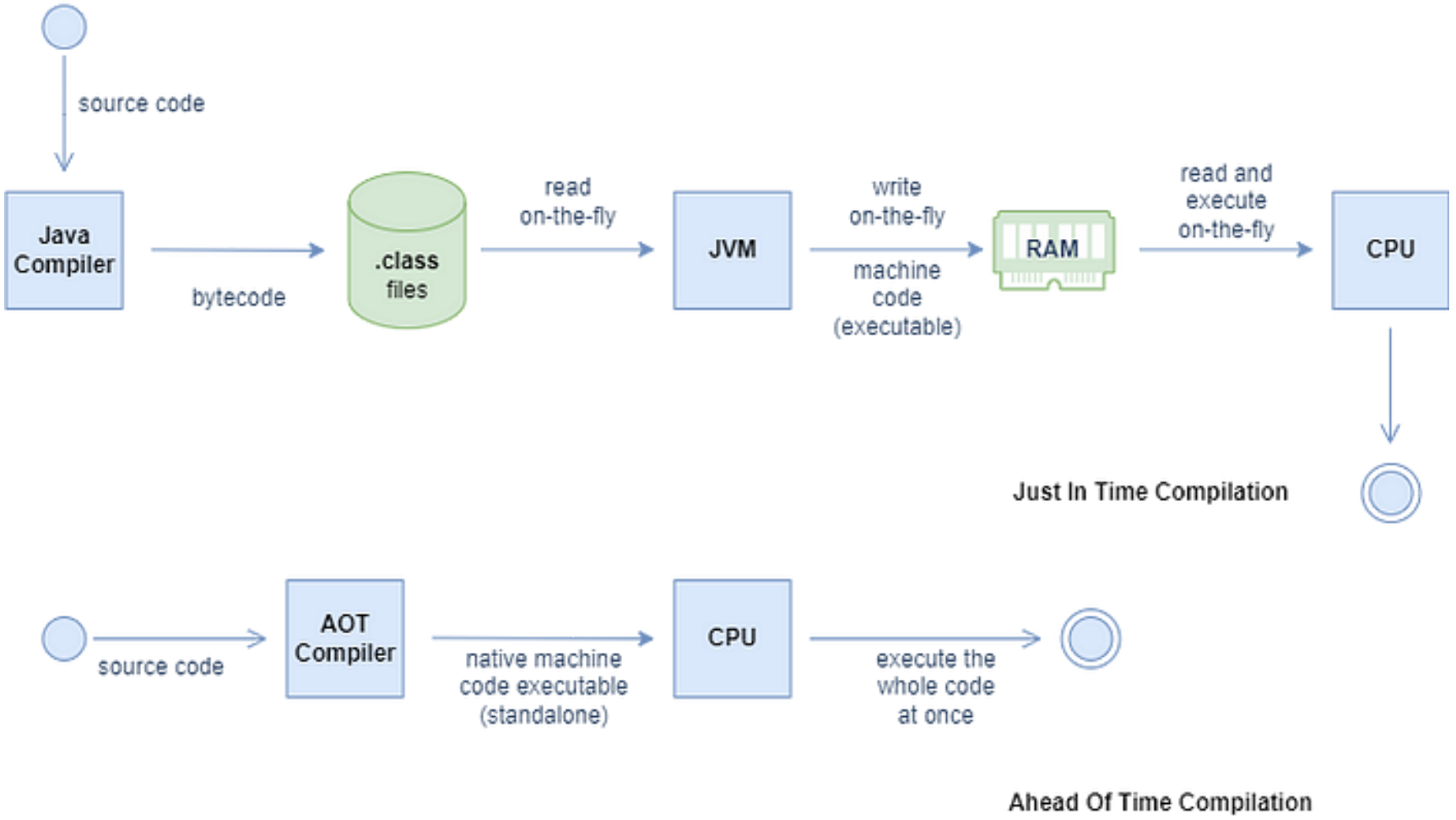
Java Compile Process



Java program compilation process



Java program compilation process



Java virtual machine (JVM) Memory Structure

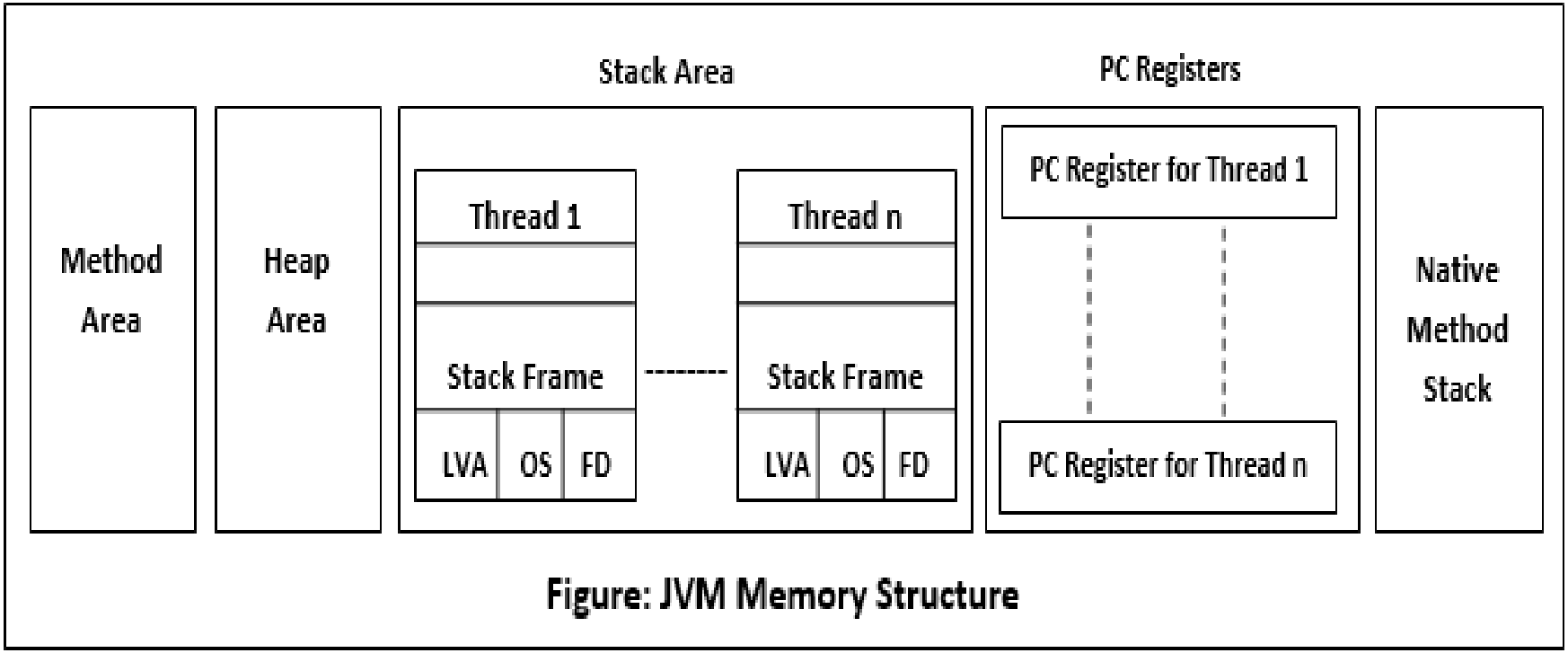
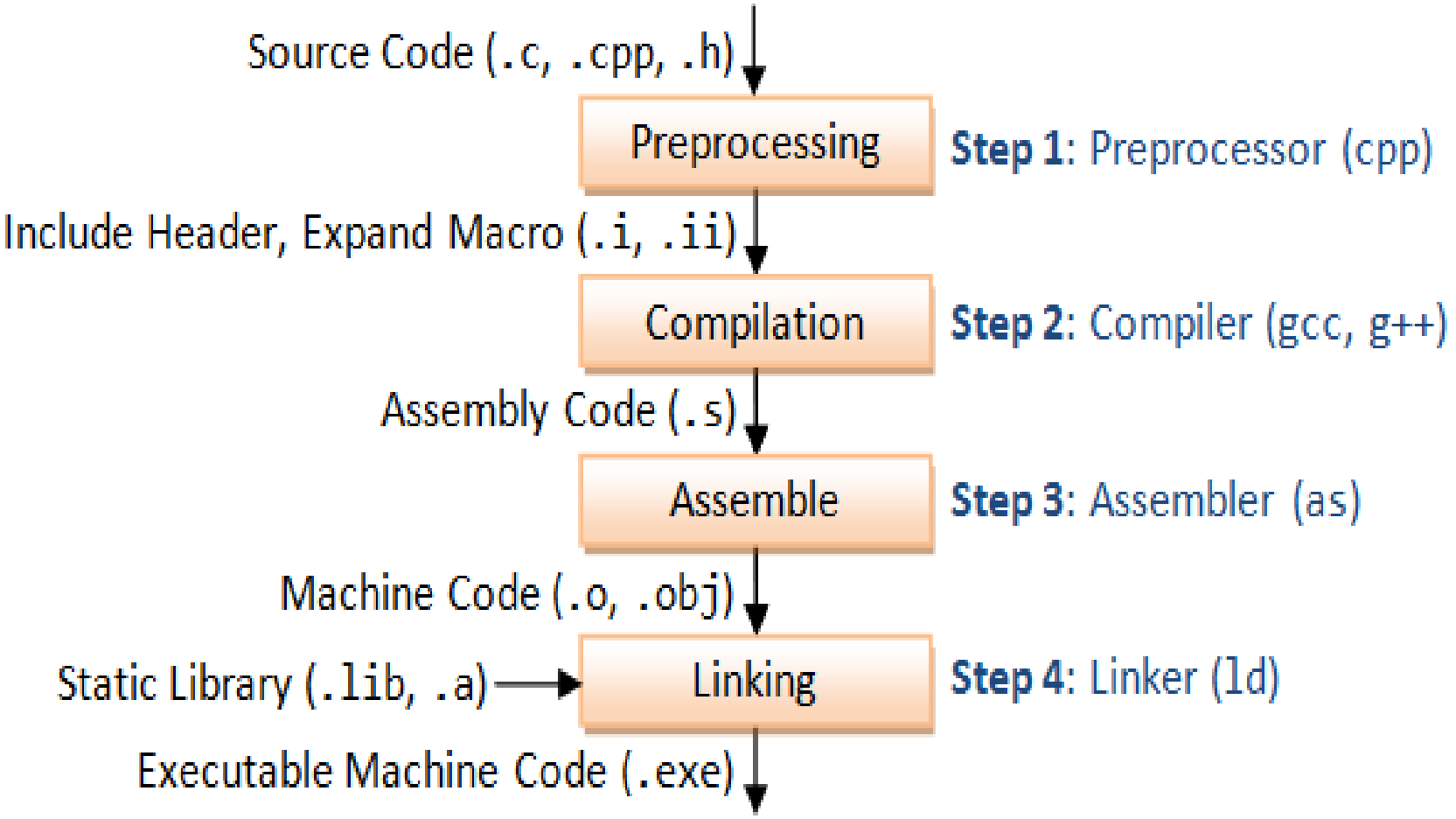
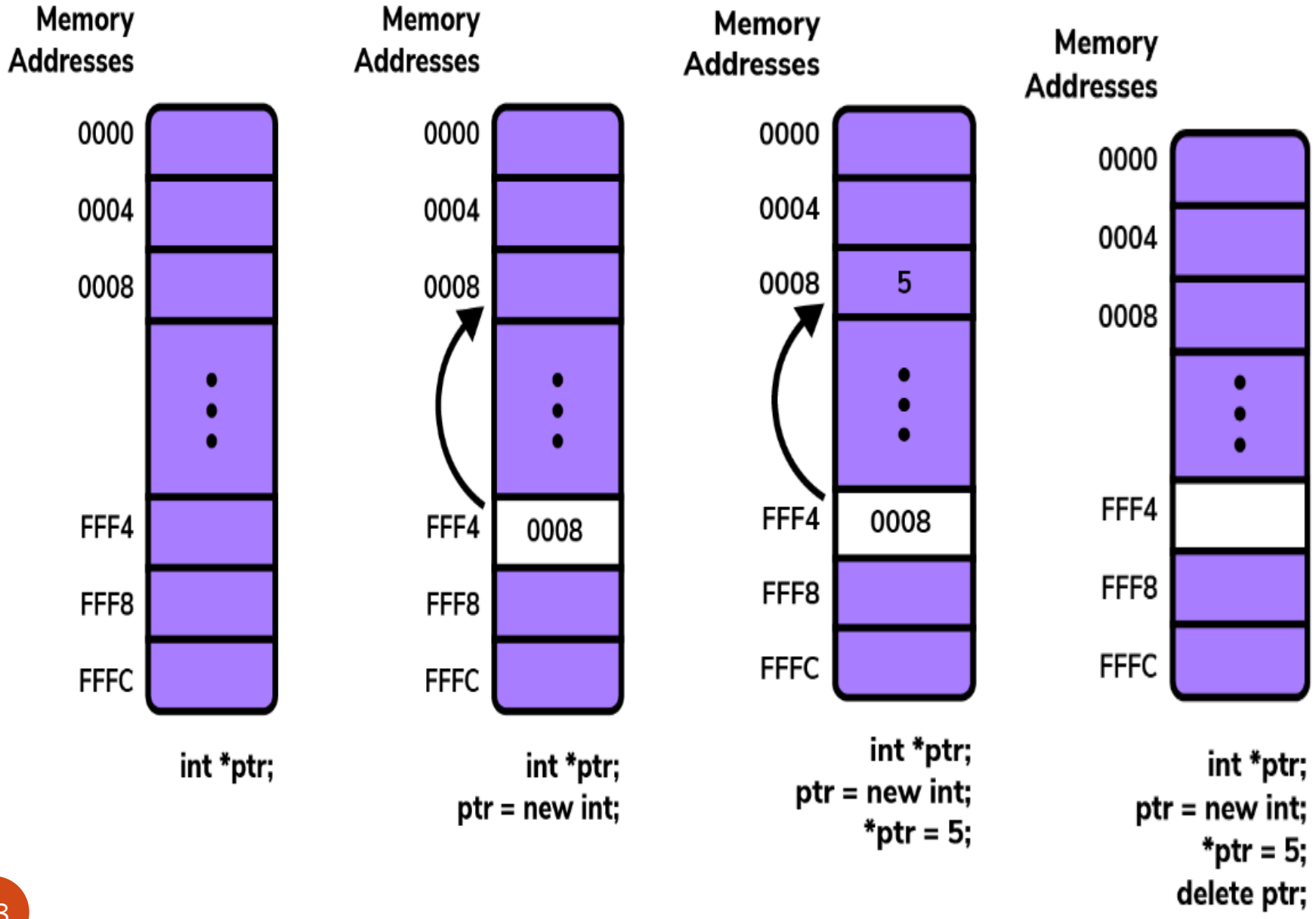


Figure: JVM Memory Structure

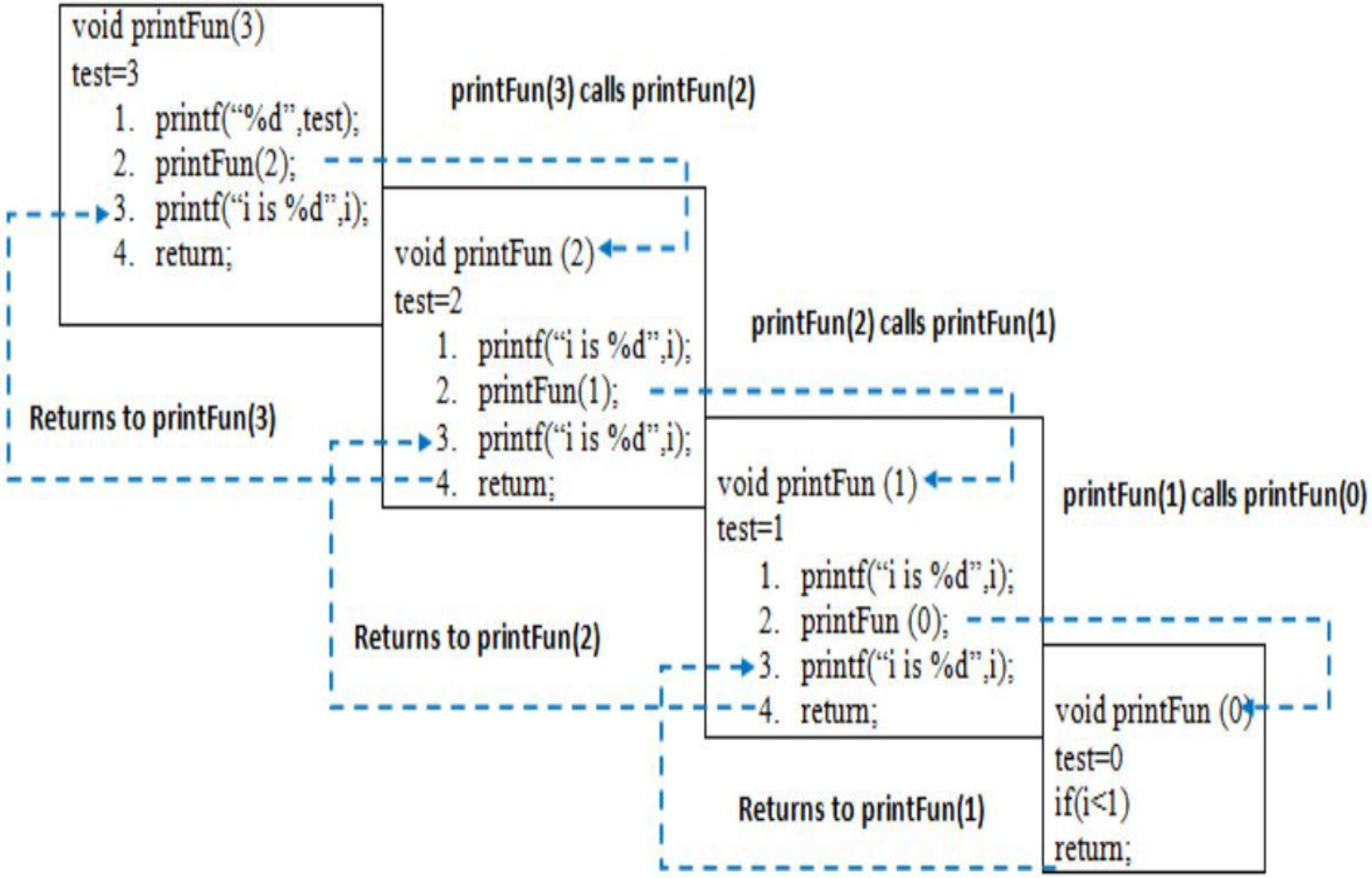
The Process of Compilation in C++



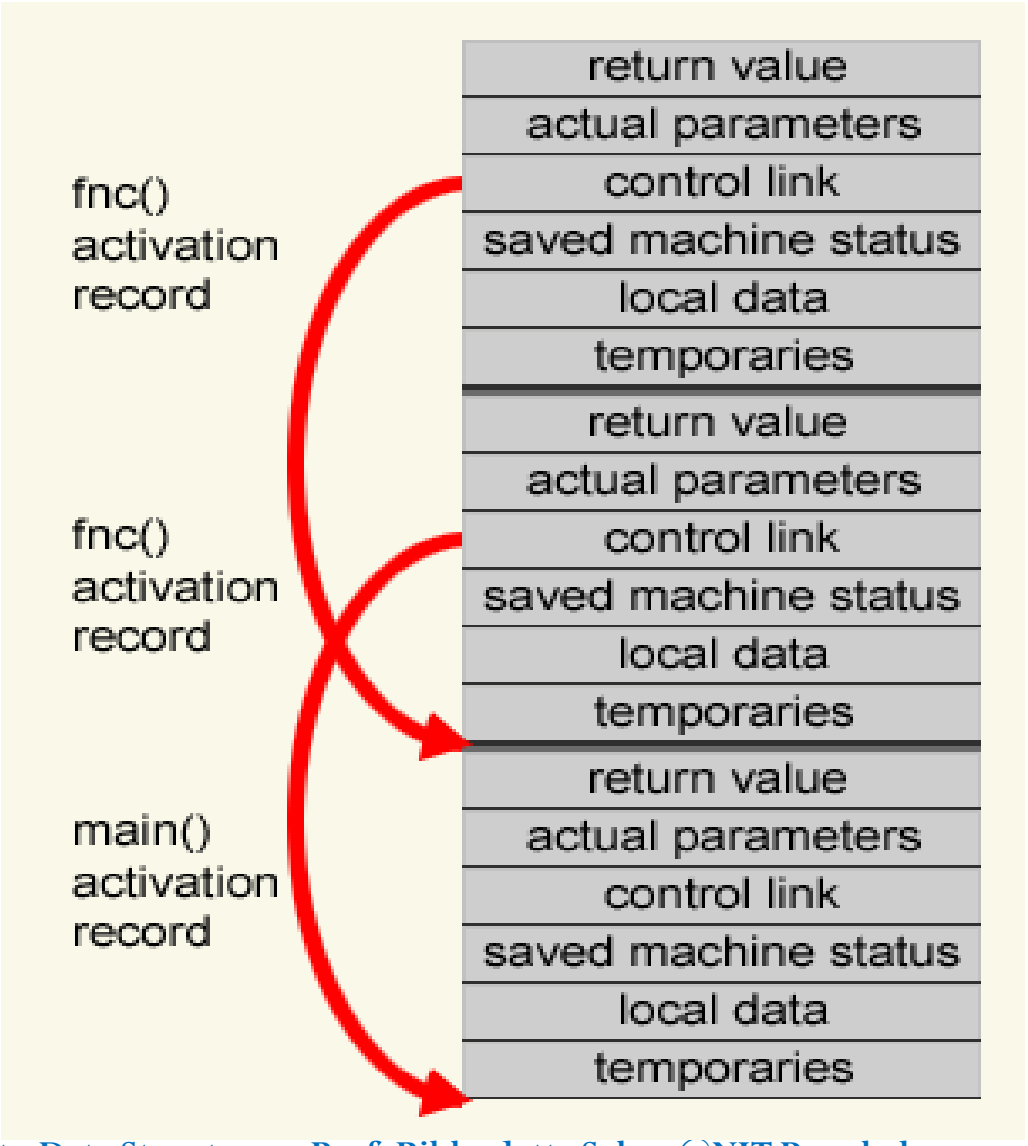
Basics of the C++ memory model



Recurssion call



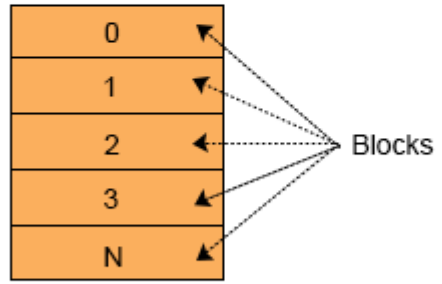
Activation record of main() program and function



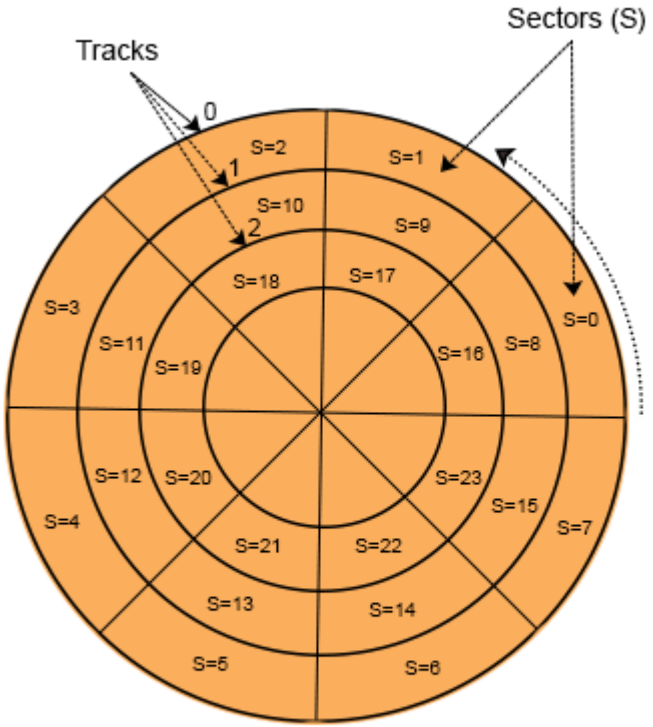
Organization DASD
Direct Access Storage Device

A category of storage devices in computing that allows data to be accessed directly from any location on the device, typically in a random-access manner

File blocks and disk sectors.



File-block Size = Disk-Sector size



Hard Disk with 3-tracks
each track contains 8-sectors

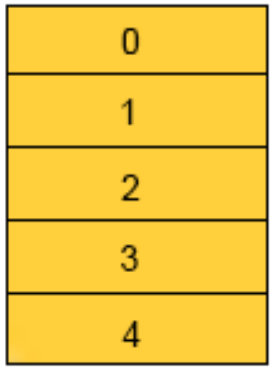
Two Files allocation into the hard disk

Directory

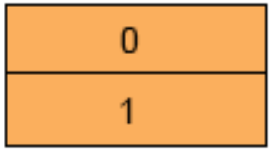
File	Start	Length
File-1	16	5
File-2	21	2

S= Sector
B= Block

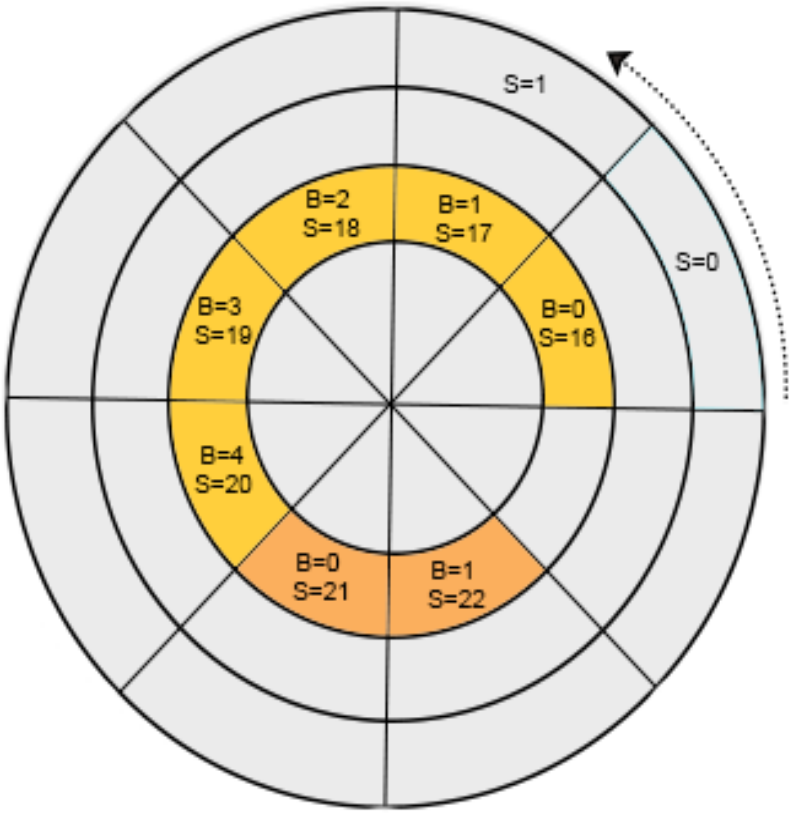
File-1 blocks
File-2 blocks
Index blocks
Filled Blocks
By other files



File-1
with 5 blocks



File-2
with 2 blocks



Hard Disk with 3-tracks
each track contains 8-sectors

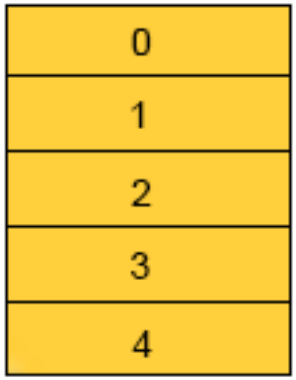
Allocation of File-1 with 5 blocks and File-2 with 2 block

Directory

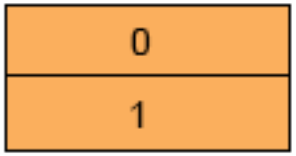
File	Start
File-1	16
File-2	17

S= Sector
B= Block

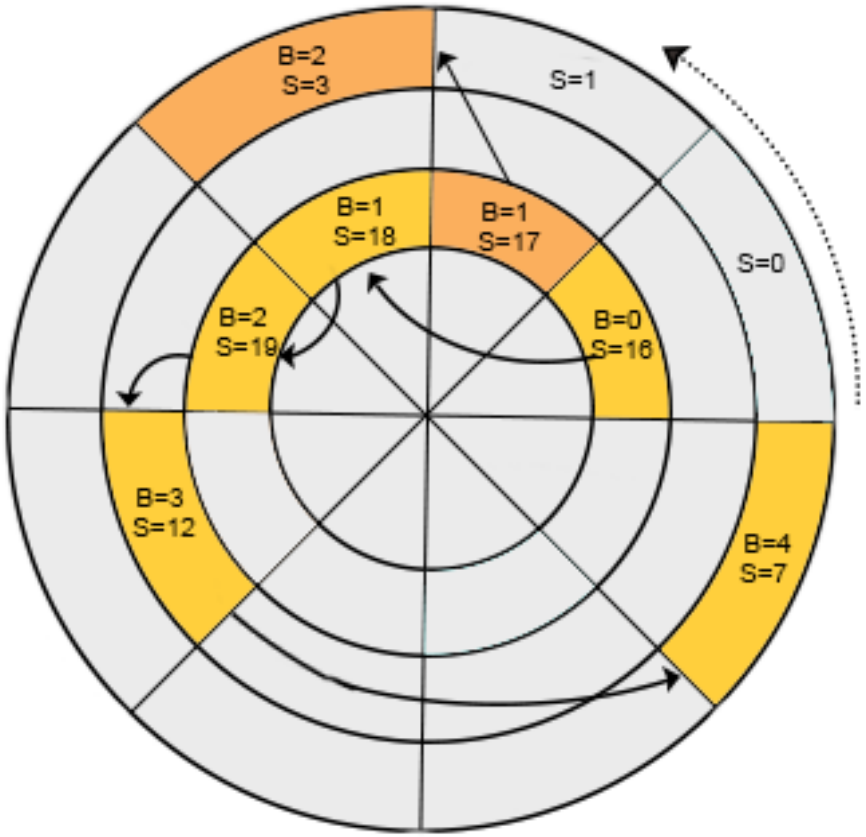
File-1 blocks
File-2 blocks
Filled Blocks
By other files



File-1
with 5 blocks

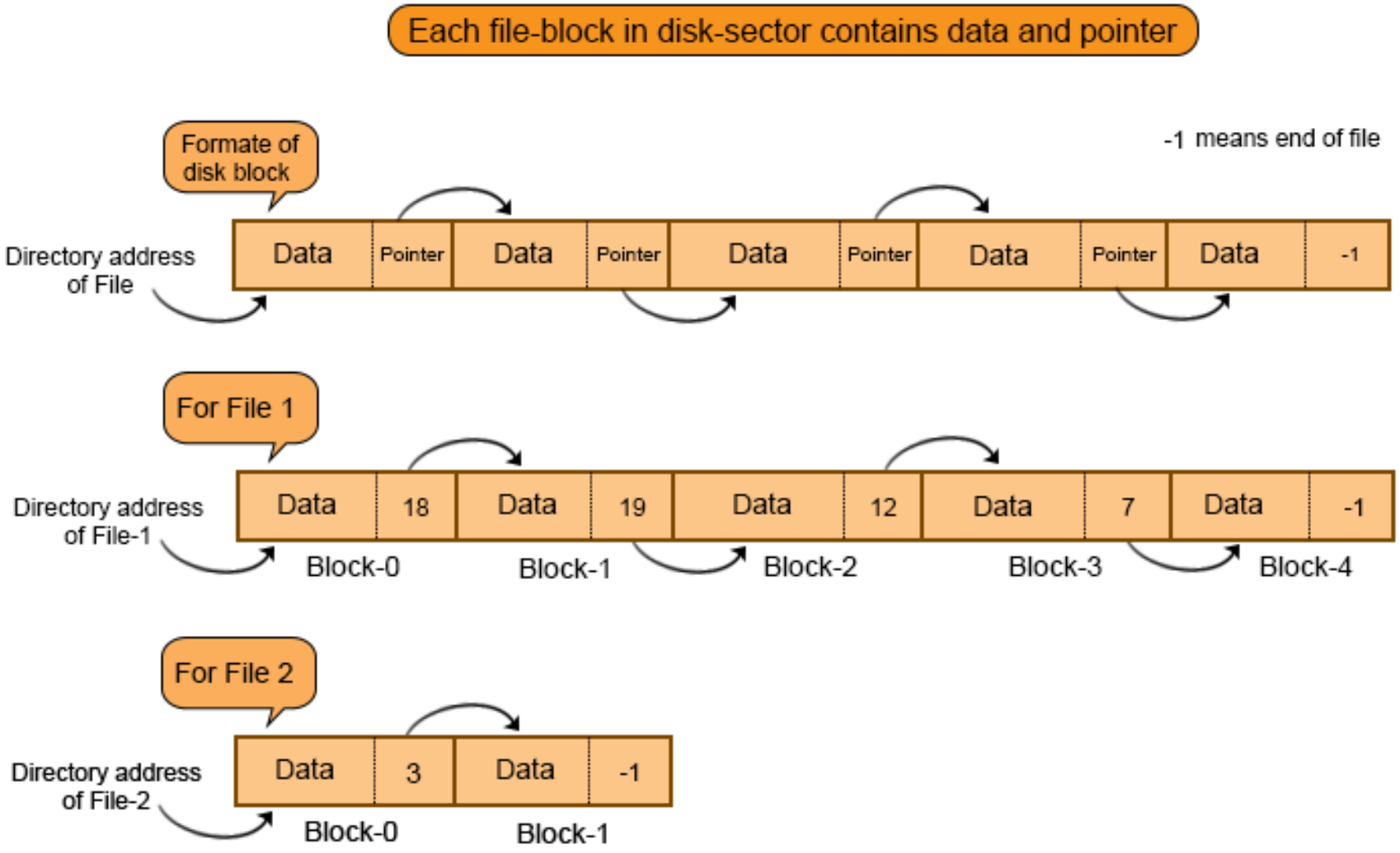


File-2
with 2 blocks



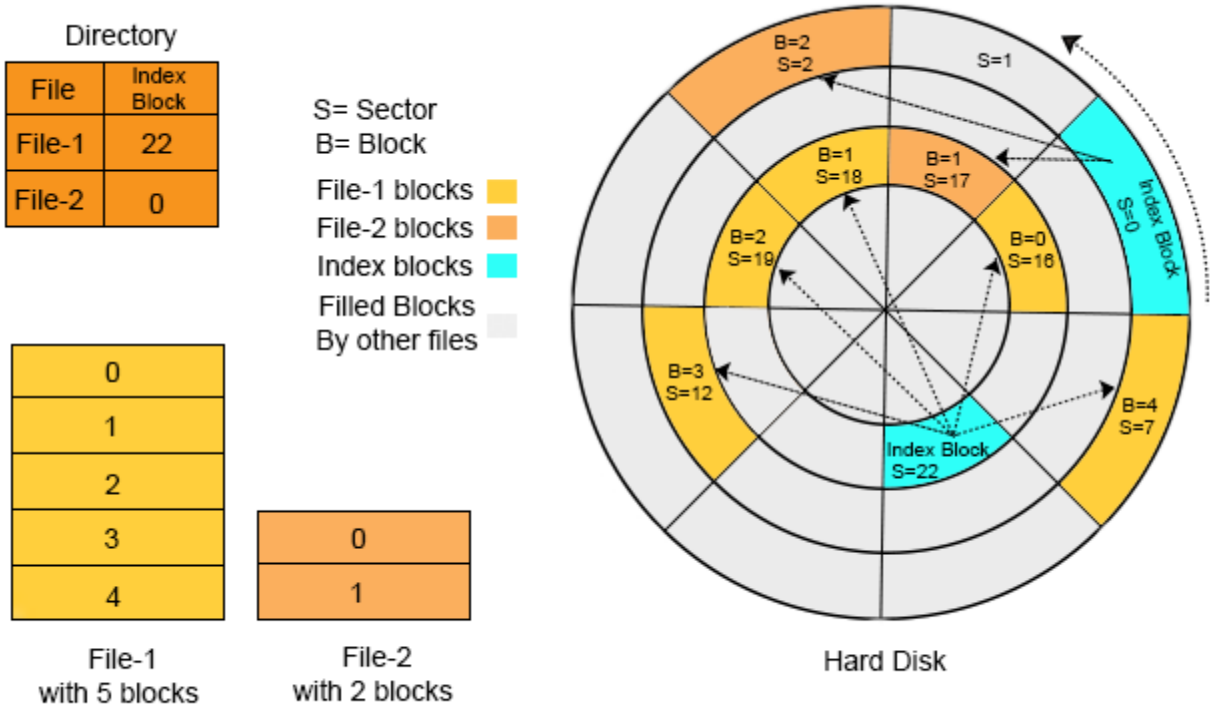
Hard Disk

Allocation of File-1 with 5 blocks and File-2 with 2 block



Indexed Allocation

In index allocation scheme, Directory contains an index block address for each file. This Indexed block does not contain the file data, but it just contains the pointers to all blocks allocated to that particular file in the hard disk.



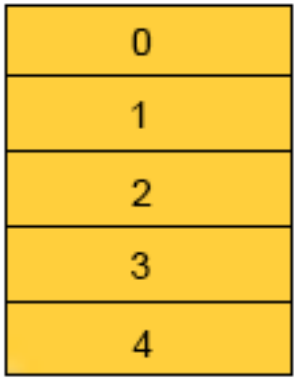
Indexed Allocation

Directory

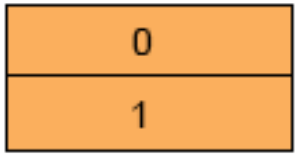
File	Index Block
File-1	22
File-2	0

S= Sector
B= Block

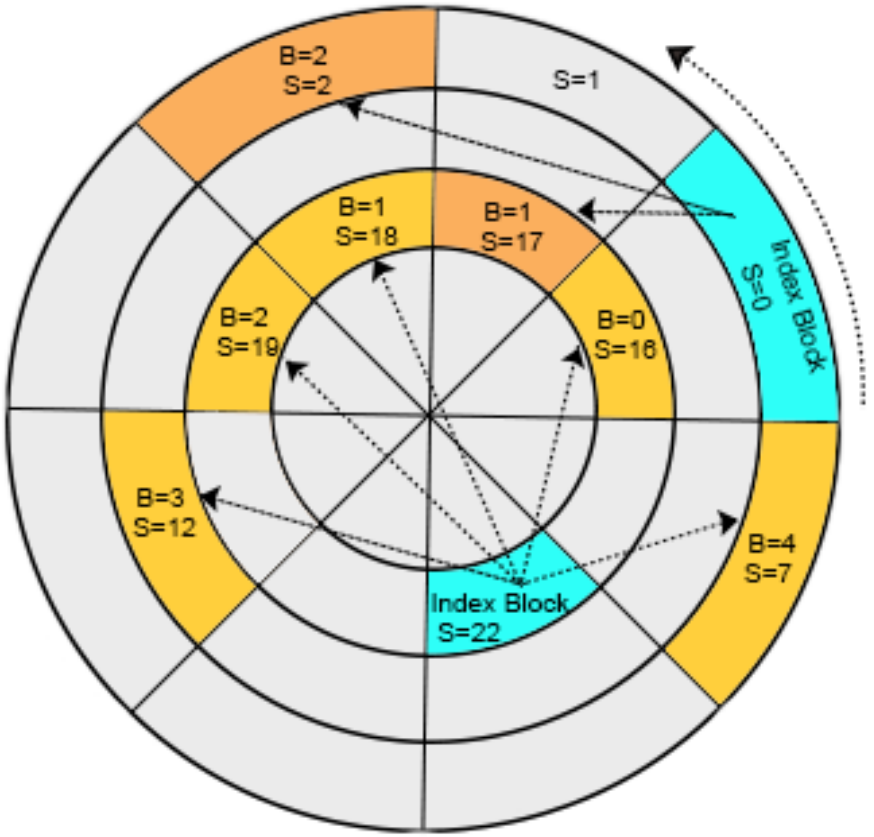
File-1 blocks
File-2 blocks
Index blocks
Filled Blocks
By other files



File-1
with 5 blocks



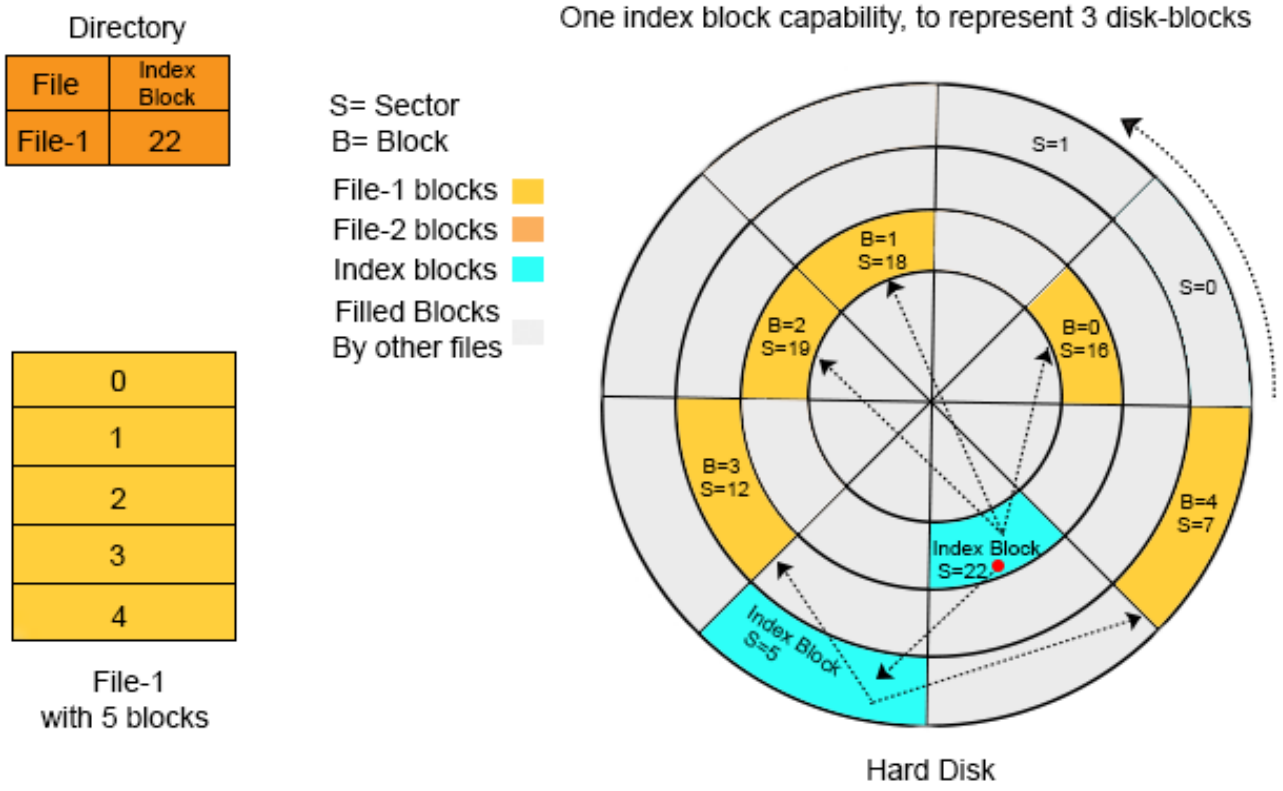
File-2
with 2 blocks



Hard Disk

Example: Indexed Allocation

- The file having 5 blocks. One index block size is capable of representing the 3 disk block, hence require two index block to represent 5 blocks of file in hard disk.



Example: Indexed Allocation

Directory

File	Index Block
File-1	22

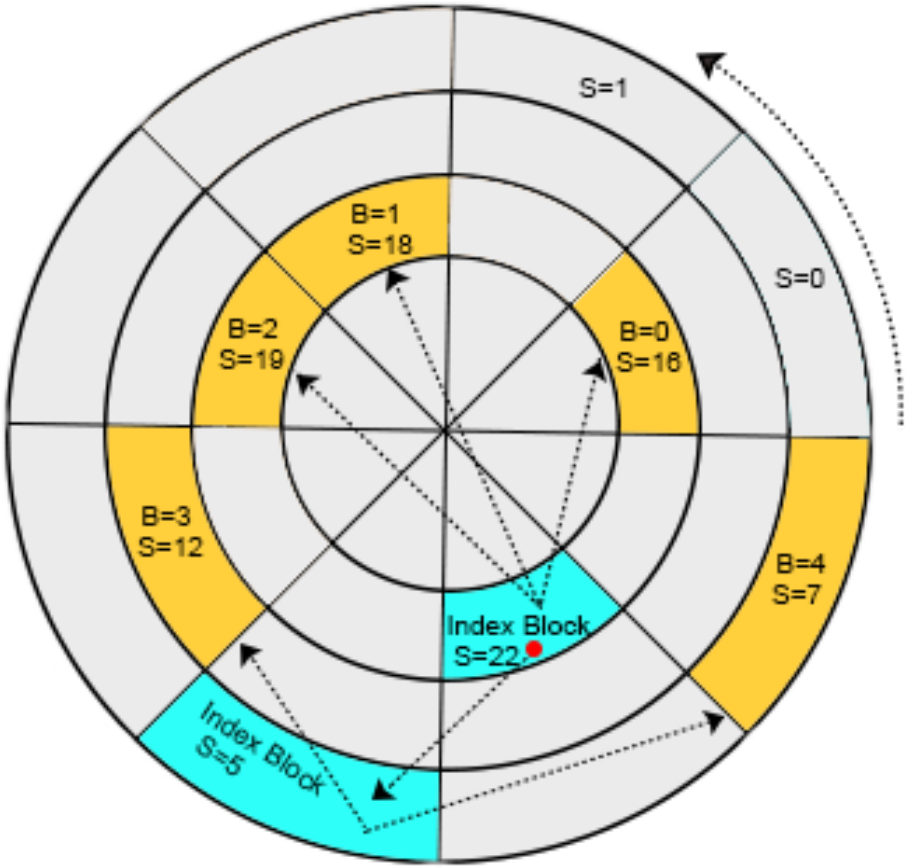
S= Sector
B= Block

File-1 blocks
File-2 blocks
Index blocks
Filled Blocks
By other files

0
1
2
3
4

File-1
with 5 blocks

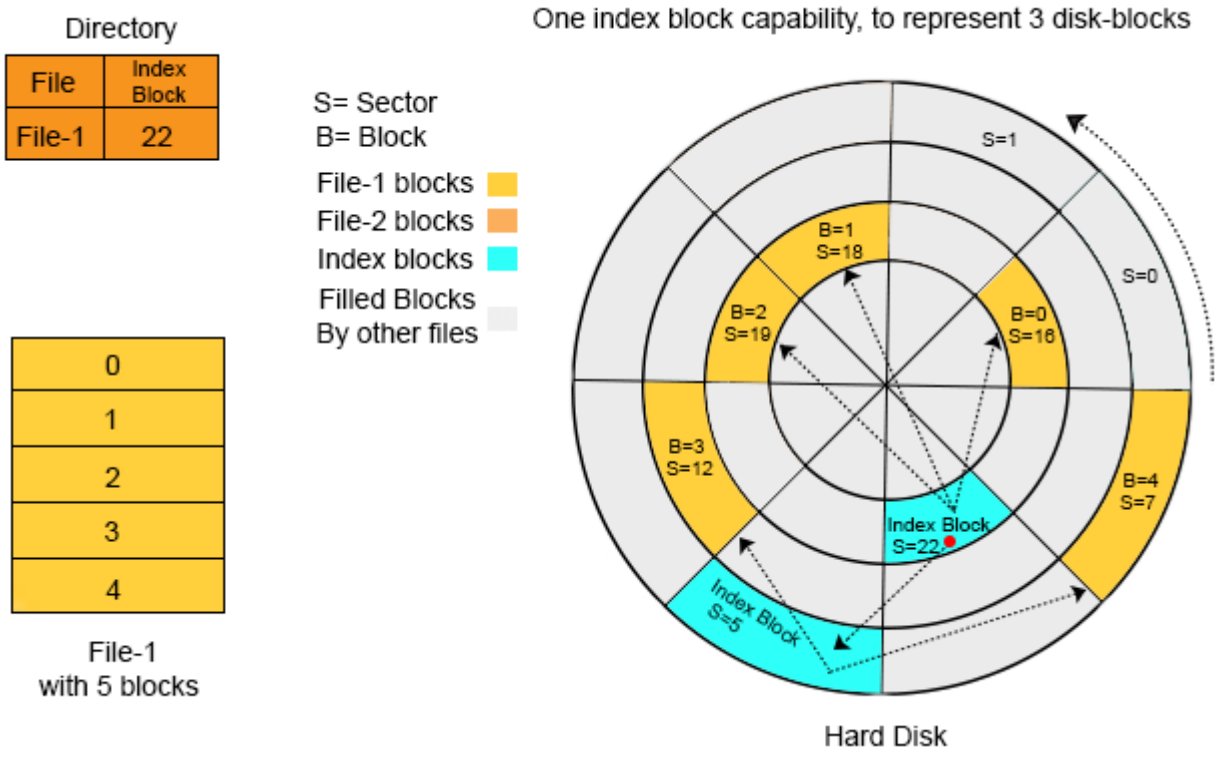
One index block capability, to represent 3 disk-blocks



Hard Disk

Indexed file organization on DASD

- Each block of data on a DASD volume is assigned a unique address that represents a distinct location, resulting in faster, more efficient data access.



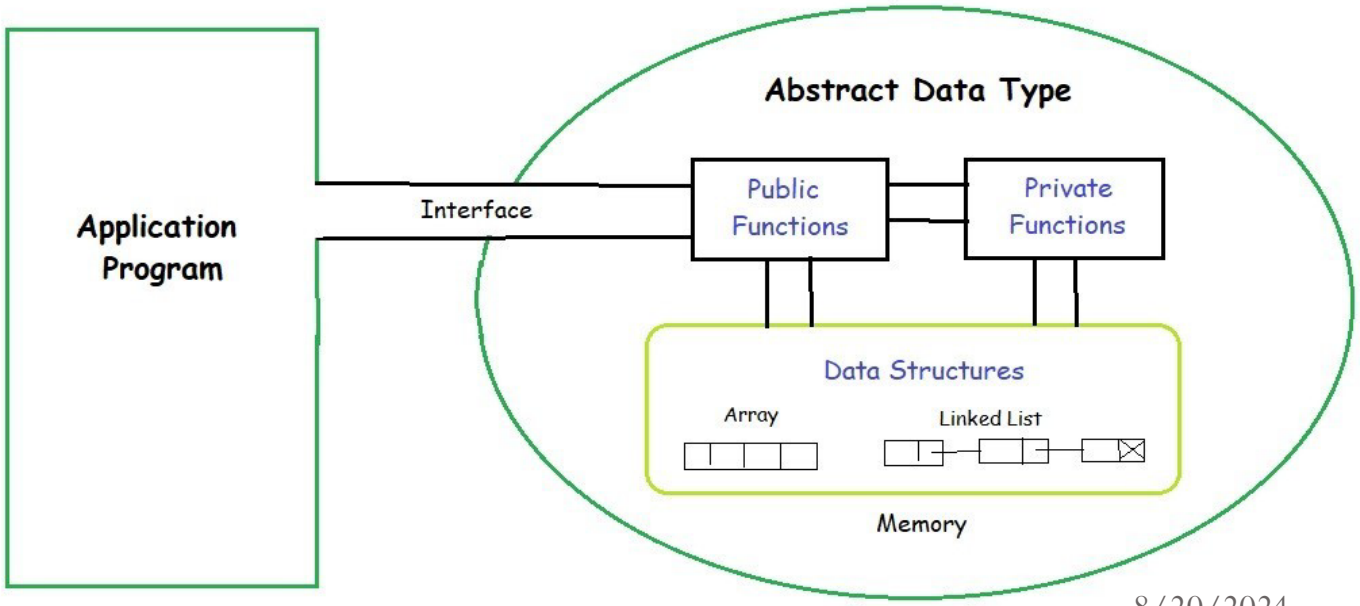
Abstract Data Types (ADT)

Abstract data types (ADT).

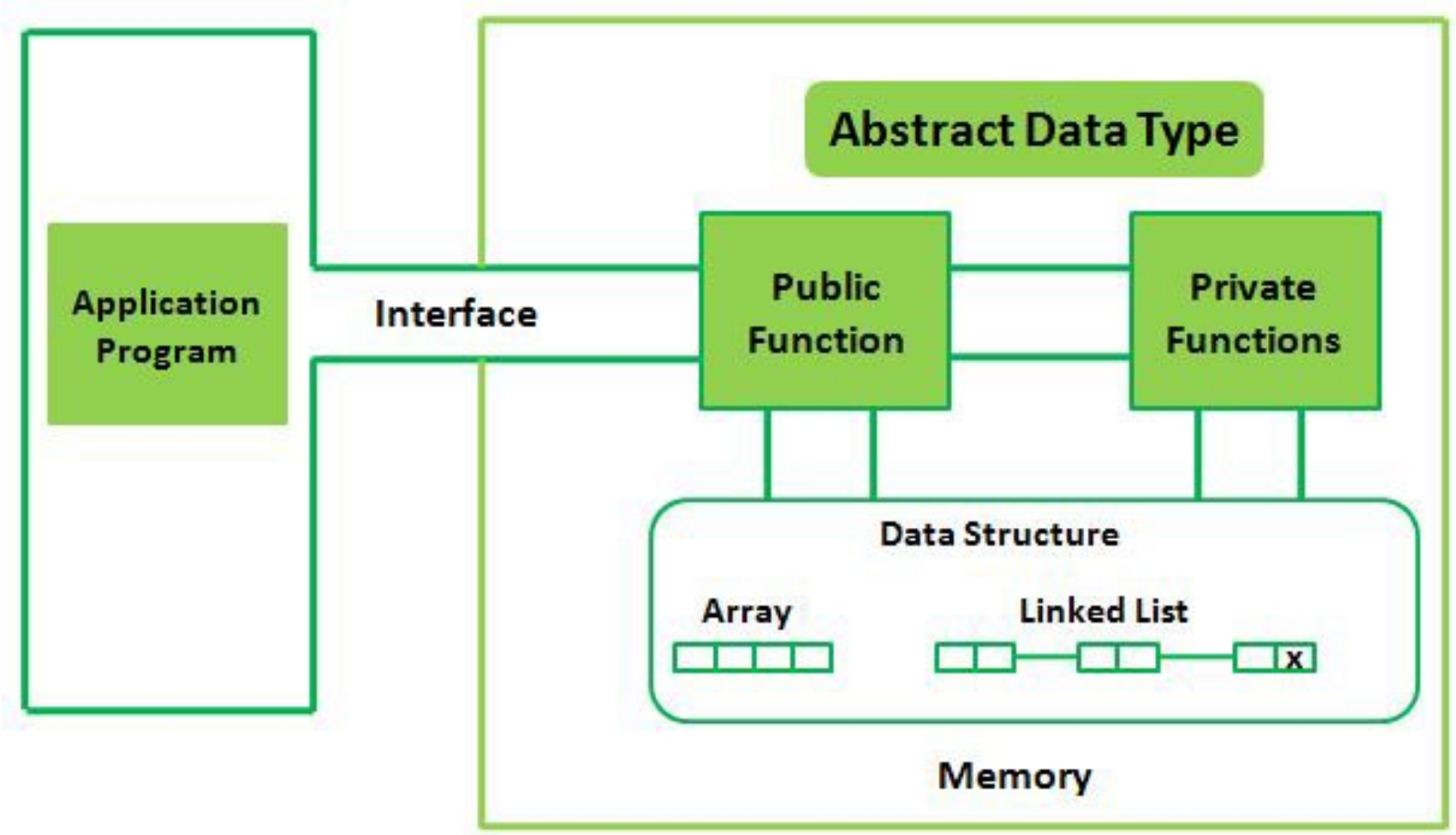
- Data structures serve as the basis for **abstract data types (ADT)**.
- The ADT defines the **logical form** of the data type. The **data structure** implements the physical form of the **data type**.
- Abstract Data type (ADT) is a type (or class) for objects whose behaviour is defined by a set of value and a set of **operations**.
- The definition of ADT only mentions what operations are to be performed but not how these operations will be implemented.
- It does not specify how data will be organized in memory and what algorithms will be used for implementing the operations.
- It is called "**abstract**" because it gives an implementation-independent view.
- The process of providing only the essentials and hiding the details is known as abstraction.

ADT

- The user of **data type** does not need to know how that data type is implemented, for example, we have been using Primitive values like int, float, char data types only with the knowledge that these data type can operate and be performed on without any idea of how they are implemented.
- ADT as a black box which hides the inner structure and design of the data type.



Abstract Data Type



Use of ADT

- For example, when writing application code, we don't care how strings are represented: we just declare variables of type string, and manipulate them by using string operations.

- Once an abstract data type has been designed, the programmer responsible for implementing that type is concerned only with choosing a suitable data structure and coding up the methods.

- On the other hand, application programmers are concerned only with using that type and calling its methods without worrying much about how the type is implemented.

The array as an ADT

ADT(Abstract Data type) :- It represent the data and set of operation on data.

Data:-

1. Array Space
2. Size
3. Length

Operation:-

1. Display()
2. Add(x)/append(x)
3. Insert(index x)
4. Delete(index)
5. Search(x)
6. Get(index)
7. Set(index x)
8. Max()/Min()
9. Reverse()
10. Shift()/rotate

Abstract Data Type (ADT) STACK

AbstractDataType *stack*
{
instances
linear list of elements; one end is called the *bottom*; the other is the *top*;

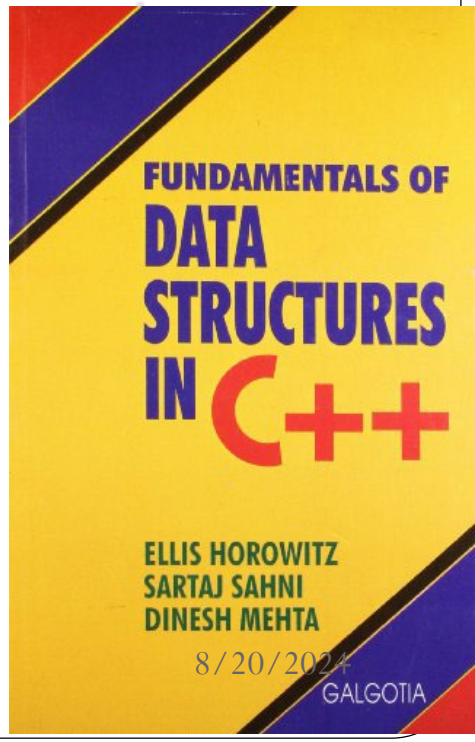
operations
empty() : Return **true** if the stack is empty, return **false** otherwise;

size() : Return the number of elements in the stack;

top() : Return the top element of the stack;

pop() : Remove the top element from the stack;

push(x) : Add element x at the top of the stack;
}

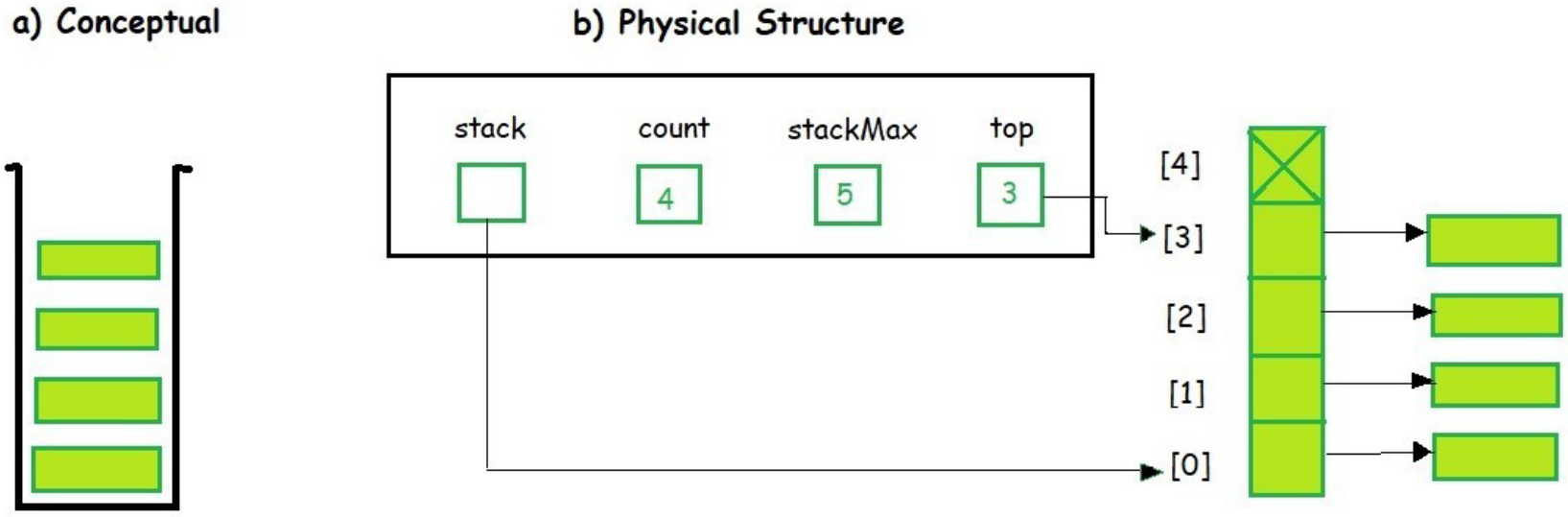


Templet : class STACK

```
template<class T>
class stack
{
   public:
      virtual ~stack() {}
      virtual bool empty() const = 0;
                  // return true iff stack is empty
      virtual int size() const = 0;
                  // return number of elements in stack
      virtual T& top() = 0;
                  // return reference to the top element
      virtual void pop() = 0;
                  // remove the top element
      virtual void push(const T& theElement) = 0;
                  // insert theElement at the top of the stack
};
```

ADT stack

- In Stack ADT Implementation instead of data being stored in each node, the pointer to data is stored.
- The program allocates memory for the *data* and *address* is passed to the stack ADT.
- The head node and the data nodes are encapsulated in the ADT. The calling function can only see the pointer to the stack.
- The stack head structure also contains a pointer to *top* and *count* of number of entries currently in stack.



How to create data structure: STACK

- How can one get to know which data structure to be used for a particular ADT (STACK)?
- An ADT tells **what** is to be done and data structure tells **how** it is to be done. In other words, we can say that ADT gives us the blueprint while data structure provides the **implementation part**.
- A stack can be implemented by means of **Array, Structure, Pointer, and Linked List**.
- The different implementations are compared for time and space.
- The Stack ADT can be implemented by both **Arrays** and **linked list**.
- Suppose the array is providing **time efficiency** while the linked list is providing **space efficiency**, so the one which is the best suited for the current user's requirements will be selected.

Abstract data type specification of a linear list

AbstractDataType *linearList*
{

instances

ordered finite collections of zero or more elements

operations

empty() : return `true` if the list is empty, `false` otherwise

size() : return the list size (i.e., number of elements in the list)

get(*index*): return the *index*th element of the list

indexOf(x): return the index of the first occurrence of x in the list, return -1 if x is not in the list

erase(*index*): remove/delete the *index*th element, elements with higher index have their index reduced by 1

insert(*index*, x): insert x as the *index*th element, elements with index $\geq$ *index* have their index increased by 1

output(): output the list elements from left to right

}

Abstract class specification of a linear list

```
template<class T>
class linearList
{
   public:
      virtual ~linearList() {}
      virtual bool empty() const = 0;
                  // return true iff list is empty
      virtual int size() const = 0;
                  // return number of elements in list
      virtual T& get(int theIndex) const = 0;
                  // return element whose index is theIndex
      virtual int indexOf(const T& theElement) const = 0;
                  // return index of first occurence of theElement
      virtual void erase(int theIndex) = 0;
                  // remove the element whose index is theIndex
      virtual void insert(int theIndex, const T& theElement) = 0;
                  // insert theElement so that its index is theIndex
      virtual void output(ostream& out) const = 0;
                  // insert list into stream out
};
```

The abstract data type queue

```
AbstractDataType queue
{
    instances
        ordered list of elements; one end is called the front; the other is the back;

    operations
        empty() : Return true if the queue is empty, return false otherwise;

        size() : Return the number of elements in the queue;

        front() : Return the front element of the queue;

        back() : Return the back element of the queue;

        pop() : Remove an element from the front of the queue;

        push(x) : Add element x at the back of the queue;
}
```

The abstract class queue

```
template<class T>
class queue
{
   public:
      virtual ~queue() {}
      virtual bool empty() const = 0;
                  // return true iff queue is empty
      virtual int size() const = 0;
                  // return number of elements in queue
      virtual T& front() = 0;
                  // return reference to the front element
      virtual T& back() = 0;
                  // return reference to the back element
      virtual void pop() = 0;
                  // remove the front element
      virtual void push(const T& theElement) = 0;
                  // add theElement at the back of the queue
};
```

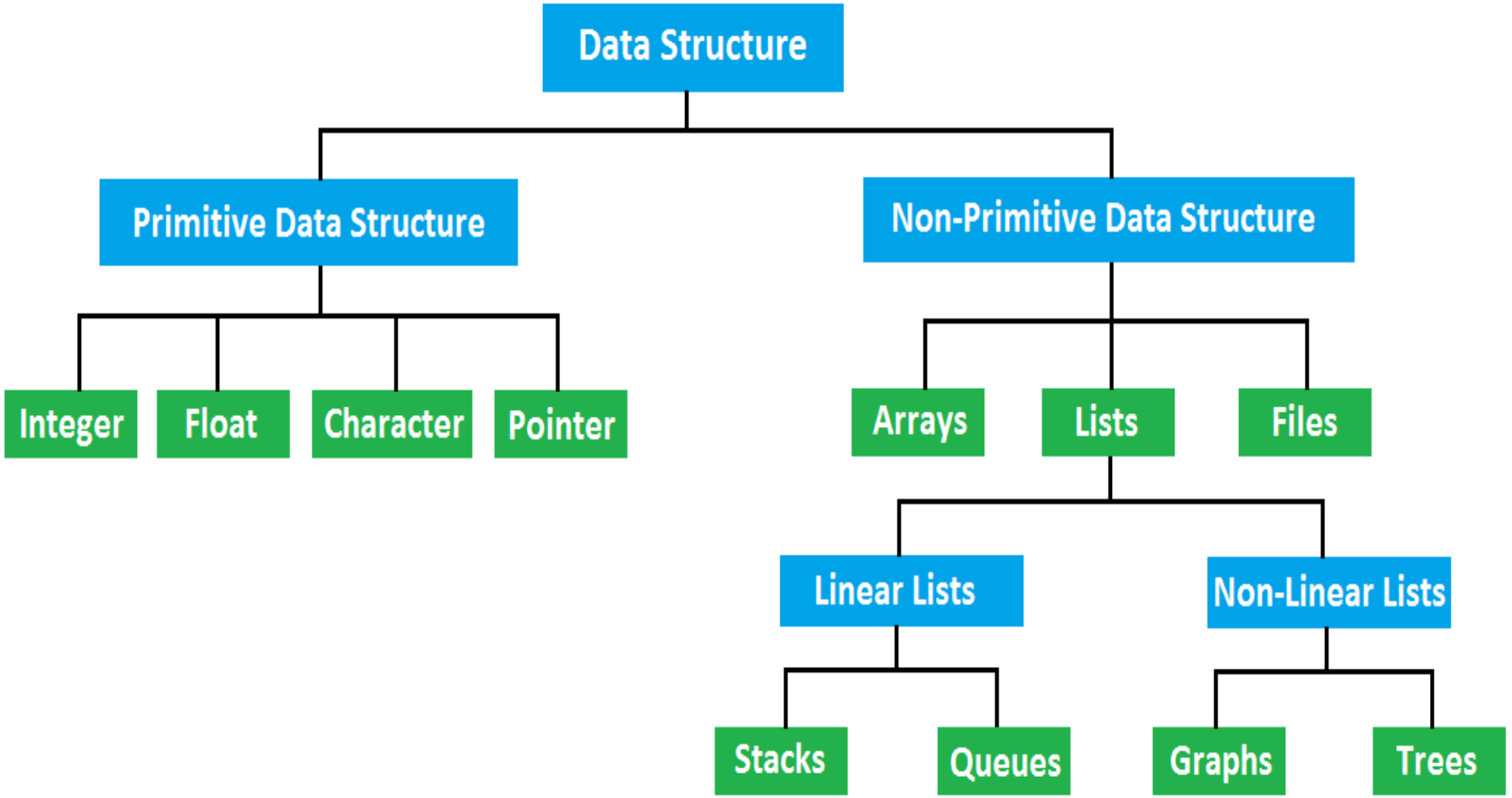
Summary

- **Abstract data type (ADT) is a concept or model of a data type.**
- In ADT, a user doesn't have to bother about how that data type has been implemented. Moreover, ADT also takes care of the implementation of the functions on a data type.
- In ADT, the user will have predefined functions on each data type ready to use for any operation. Generally, in ADT, a user knows what to do without disclosing how to do it. These kinds of models are defined in terms of their data items and associated operations.
- In every programming language, we implement ADTs using different methods and logic. For example, in C, ADTs are implemented mostly using structure. On the other hand, in C++ or JAVA, they're implemented using class. **However, operations are common in all languages.**
- ADTs are a popular and important data type. Generally, ADTs are mathematical or logical concepts that can be implemented on different machines using different languages.
- ADT are very flexible and don't dependent on languages or machines.

Classification of data structure

A data structure is a specialized format for organizing, processing, retrieving and storing data.

Classification of data structure



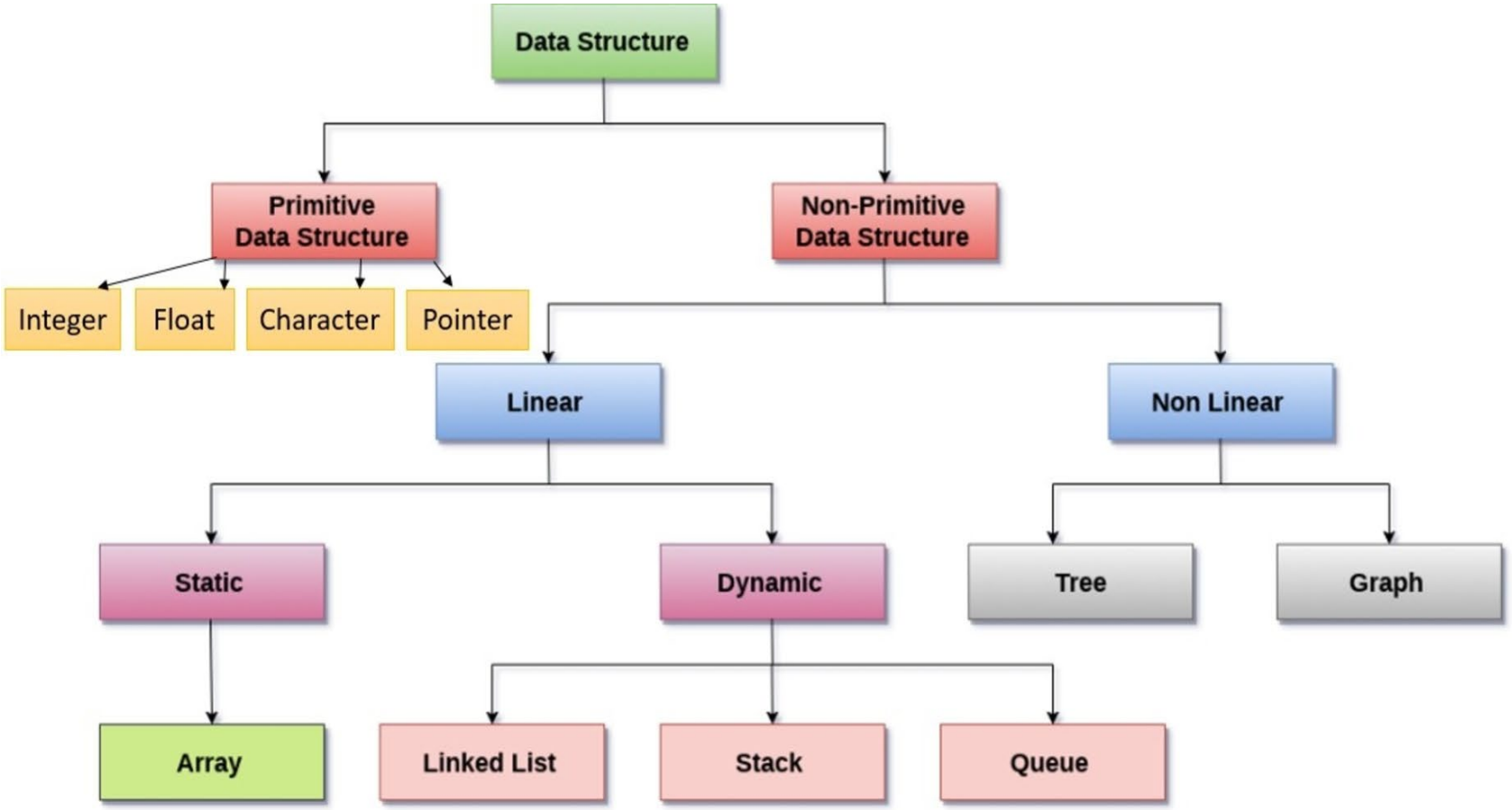
Primitive Data Structures

- **Basic:** Primitive data structures are the most basic and fundamental data structures that are directly built into a programming language.
- **Simple Representation:** They have a simple and straightforward representation, such as integers, floating-point numbers, and Booleans.
- **Limited Functionality:** Primitive data structures are limited in their functionality and can only store and manipulate simple data.
- **Built-in Types:** Primitive data structures are typically built-in types in most programming languages and do not require any special implementation.
- **Fast Access:** Primitive data structures are stored in contiguous memory locations, which allows for fast access to their elements.
- **Efficient Storage:** Since primitive data structures have a simple representation, they use less memory compared to non-primitive data structures.
- **Commonly used:** Primitive data structures are commonly used in programming to store basic information and are used as building blocks for more complex data structures.

Non-Primitive Data Structures

- **Complex:** Non-primitive data structures are more complex and sophisticated data structures that are built using primitive data structures.
- **Advanced Functionality:** They provide more advanced functionality and enable the efficient storage and manipulation of large amounts of data.
- **Used to solve complex problems:** Non-primitive data structures are used to solve complex problems that cannot be solved using primitive data structures alone.
- **Dynamic:** Non-primitive data structures can grow or shrink in size during runtime, whereas primitive data structures have a fixed size.
- **More Memory:** Non-primitive data structures use more memory compared to primitive data structures due to their complex representation.
- **Higher Abstraction:** Non-primitive data structures provide a higher level of abstraction compared to primitive data structures and hide the underlying implementation details.
- **Efficient Access:** Non-primitive data structures are designed to provide efficient access to elements based on the problem they are used to solve.

Classification of data structure

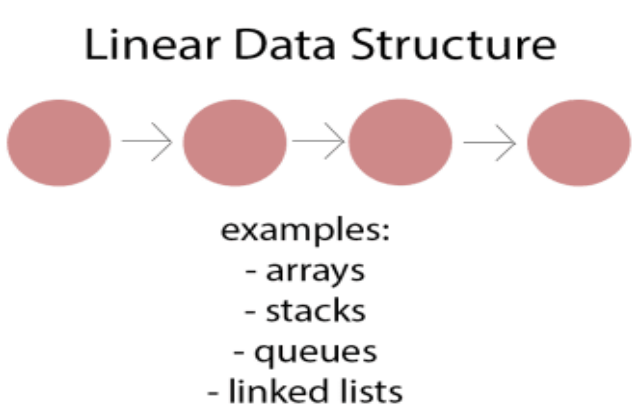


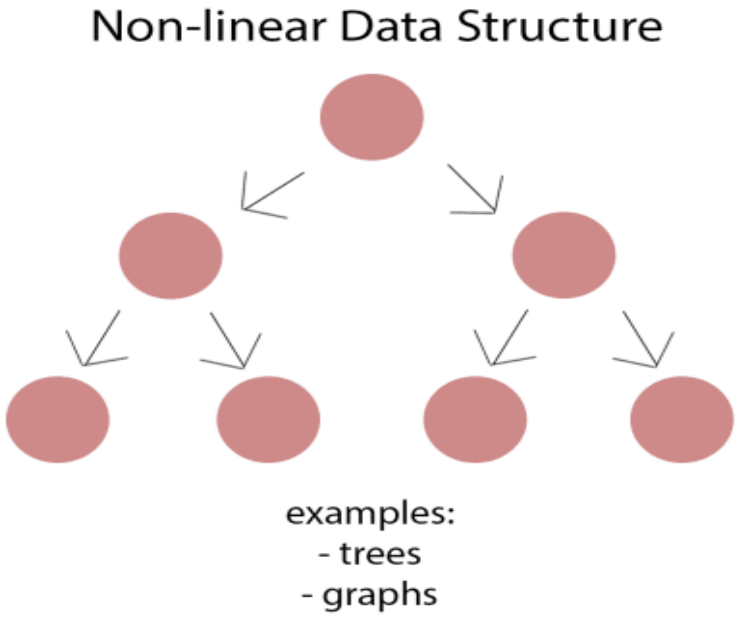
Classification of Data Structure

- **Primitive structures:** For example integer, real, character, Boolean. These data types are simple as these cannot be further sub-divided.
- **Non-primitive structures:** For example: Arrays, Trees, Linked List etc. These are complex data types. Based on the arrangement of data, the primitive structures can be further divided into two types: linear and non-linear data structure.
- **Linear data structure:** In linear data structure, the data is arranged in linear fashion or sequential order. Some of the examples are arrays, linked lists, stacks, queues.
- **Non-linear data structure:** In non-linear data structure, the data is not arranged in sequence. Some of the examples are trees, graphs.

Classification of Data Structure

- The main difference between linear and non linear data structures is that linear data structures arrange data in a sequential manner while nonlinear data structures arrange data in a hierarchical manner, creating a relationship among the data elements.

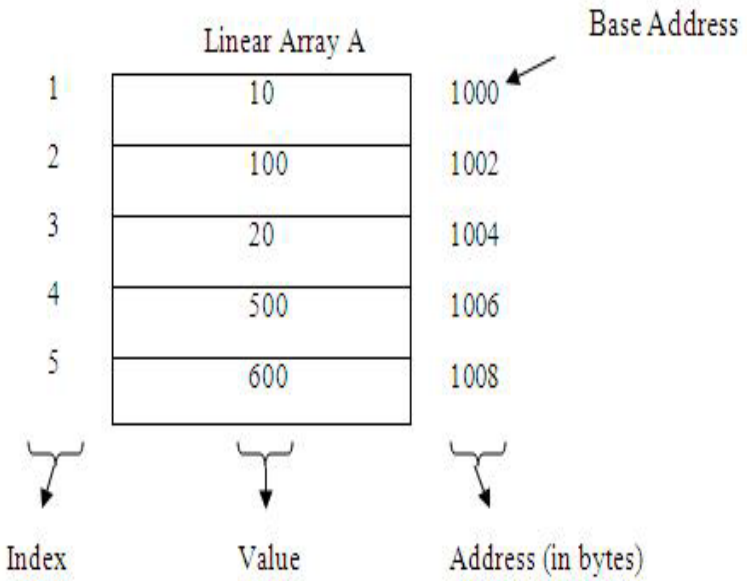




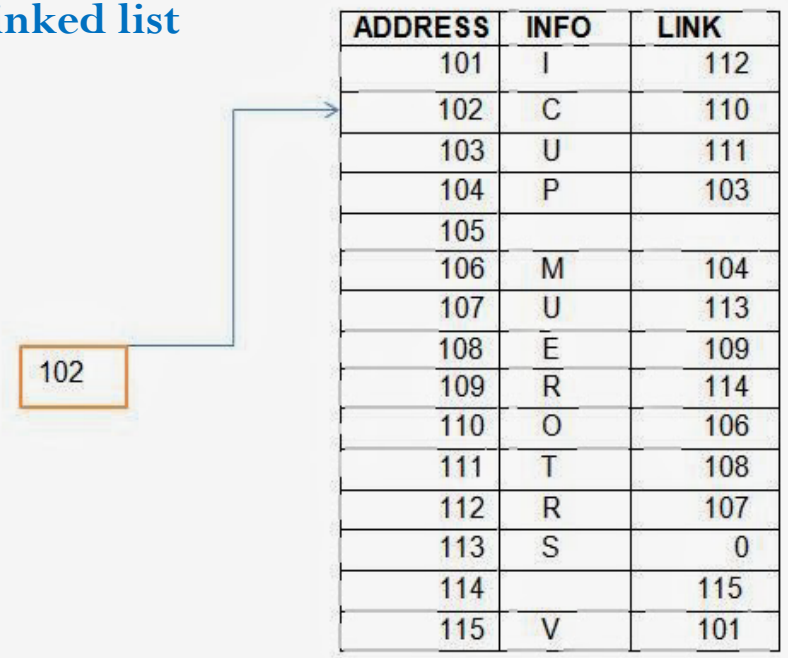
Linear data structure

A **linear data structure** has data elements connected to each other so that elements are arranged in a sequential manner and each element is connected to the element in front of it and behind it.

In a **linear data structure,** there is a **linear relationship** between two consecutive **data element**.



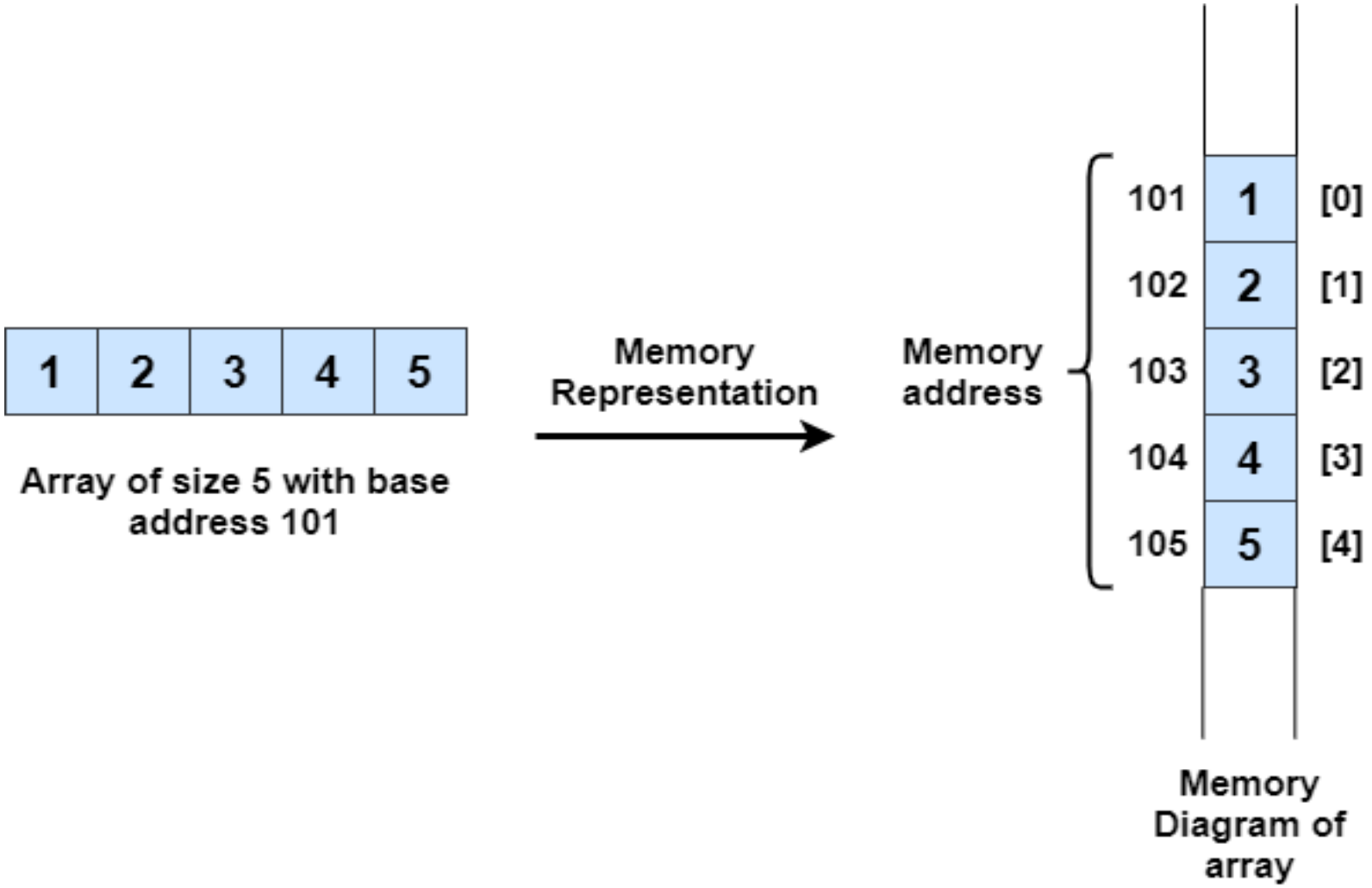
Linked list



ADDRESS	INFO	LINK
101	I	112
102	C	110
103	U	111
104	P	103
105		
106	M	104
107	U	113
108	E	109
109	R	114
110	O	106
111	T	108
112	R	107
113	S	0
114		115
115	V	101

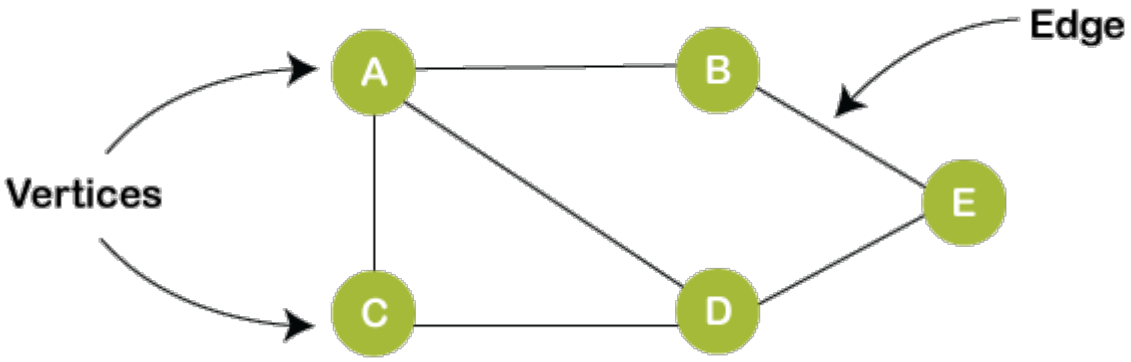
COMPUTER VIRUS

Linear data structure



Non-linear data structure

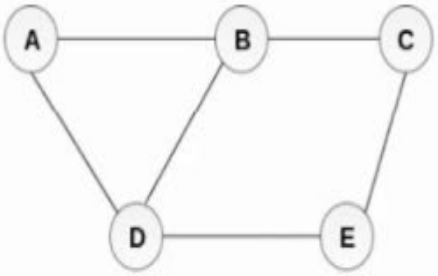
- A non-linear data structure is also another type of data structure in which the data elements are not arranged in a contiguous manner.
- As the arrangement is **non-sequential**, so the data elements cannot be traversed or accessed in a single run.
- In non-linear data structure, an element can be connected to more than two elements.



Non-linear data structure

- Non-linear data structures are data organization in a hierarchical or network-based format, where elements can have one or more relationships with other elements.
- Non-linear data structure allows for more flexible and efficient data organization, especially for representing relationships and hierarchical data.
- Non-linear data structures are more efficient in utilizing computer memory than linear data structures.
- Examples of non-linear data structures include trees, graphs, and heaps. These types of data structures are useful for applications that involve large amounts of data or complex data relationships, such as computer graphics, databases, and artificial intelligence.

Non-linear data structure: Graph

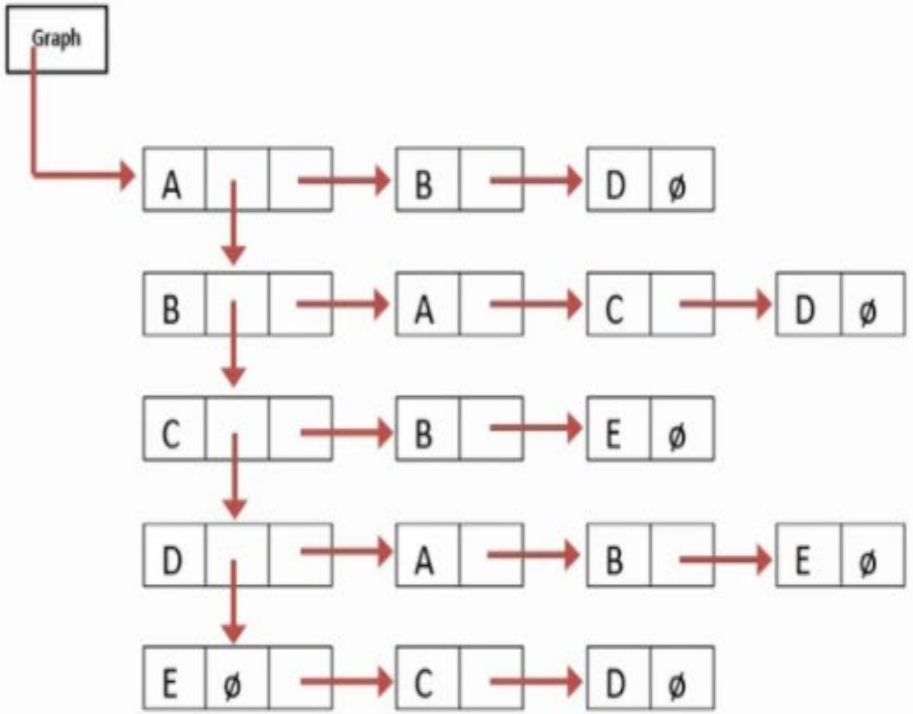


Let V_G = {A, B, C, D, E}

Adjacency matrix will be

	A	B	C	D	E
A	0	1	0	1	0
B	1	0	1	1	0
C	0	1	0	0	1
D	1	1	0	0	1
E	0	0	1	1	0

Adjacency List will be



Graph in main memory

	1	2	3	4
1	A[1,1]	A[1,2]	A[1,3]	A[1,4]
2	A[2,1]	A[2,2]	A[2,3]	A[2,4]
3	A[3,1]	A[3,2]	A[3,3]	A[3,4]

ELEMENTS	ADDRESS	
A[1,1]	200	Column 1
A[2,1]	201	
A[3,1]	202	
A[1,2]	203	Column 2
A[2,2]	204	
A[3,2]	205	
A[1,3]	206	Column 3
A[2,3]	207	
A[3,3]	208	
A[1,4]	209	Column 4
A[2,4]	210	
A[3,4]	211	

Concept:

Column Major Order (CMO)

Graph in main memory

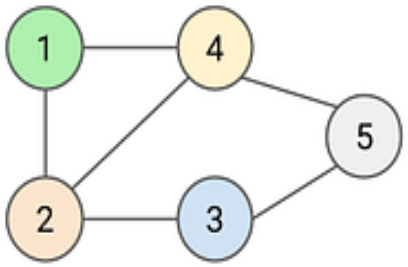
	1	2	3	4
1	A[1,1]	A[1,2]	A[1,3]	A[1,4]
2	A[2,1]	A[2,2]	A[2,3]	A[2,4]
3	A[3,1]	A[3,2]	A[3,3]	A[3,4]

Concept:

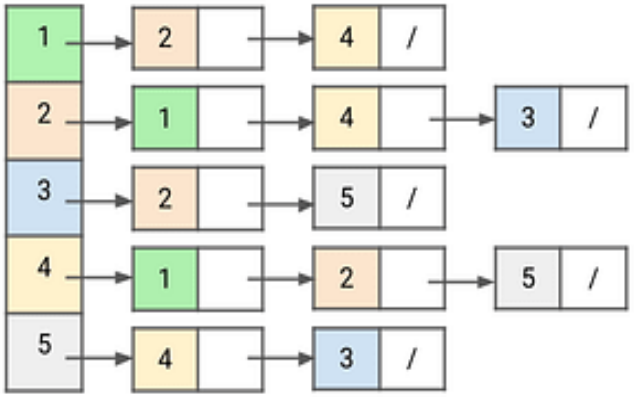
Row Major Order (RMO)

ELEMENTS	ADDRESS	
A[1,1]	200	Row 1
A[1,2]	201	
A[1,3]	202	
A[1,4]	203	
A[2,1]	204	Row 2
A[2,2]	205	
A[2,3]	206	
A[2,4]	207	
A[3,1]	208	Row 3
A[3,2]	209	
A[3,3]	210	
A[3,4]	211	

Non-linear data structure: Graph



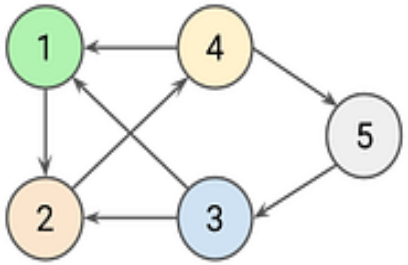
Undirected Graph



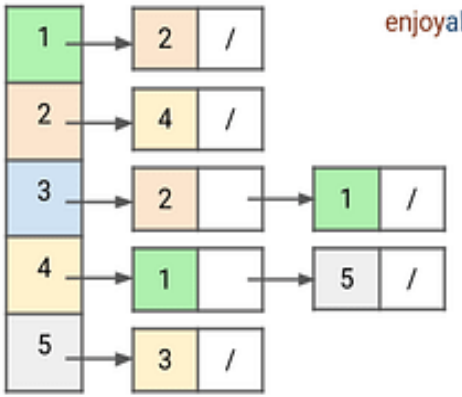
Adjacency List Representation

	1	2	3	4	5
1	0	1	0	1	0
2	1	0	1	1	0
3	0	1	0	0	1
4	1	1	0	0	1
5	0	0	1	1	0

Adjacency Matrix Representation



Directed Graph



enjoyalgorithms.com

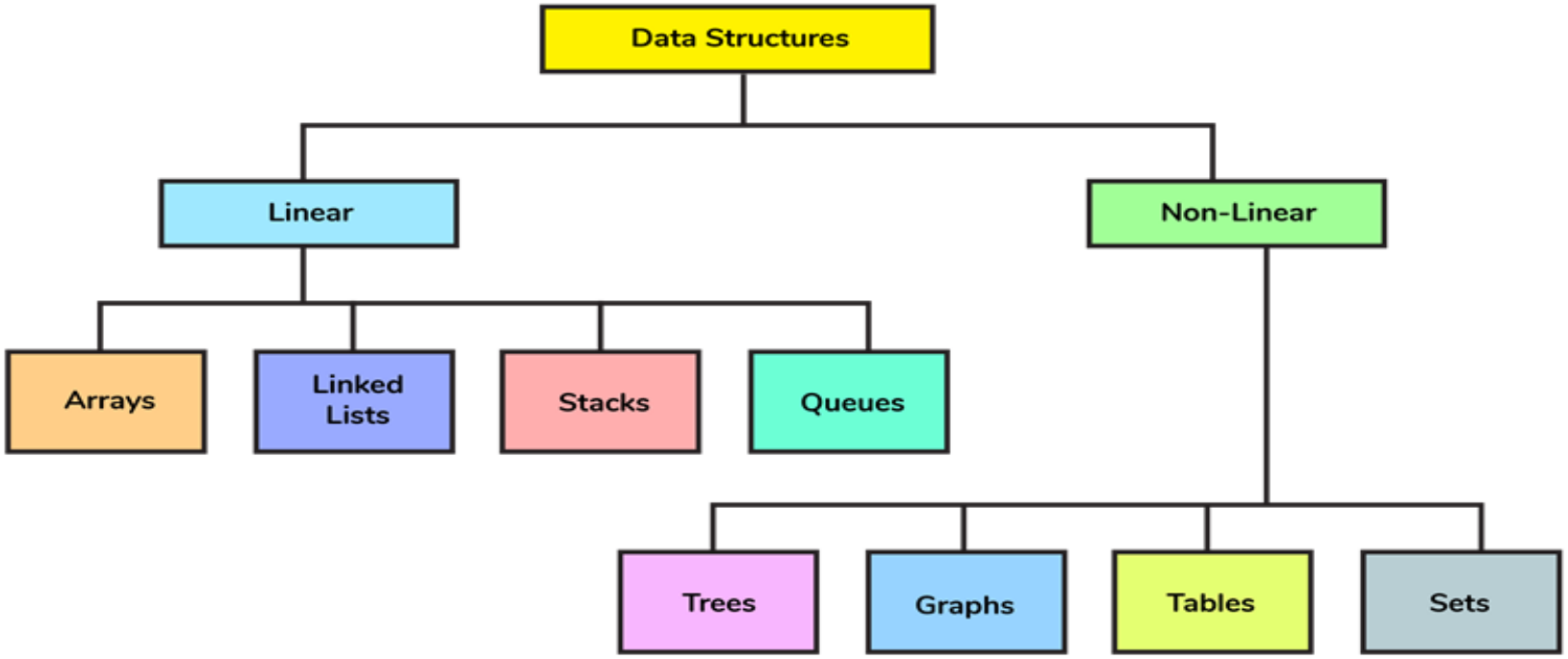
	1	2	3	4	5
1	0	1	0	0	0
2	0	0	0	1	0
3	1	1	0	0	0
4	`1	0	0	0	1
5	0	0	1	0	0

LINEAR DATA STRUCTURES VERSUS NON LINEAR DATA STRUCTURES

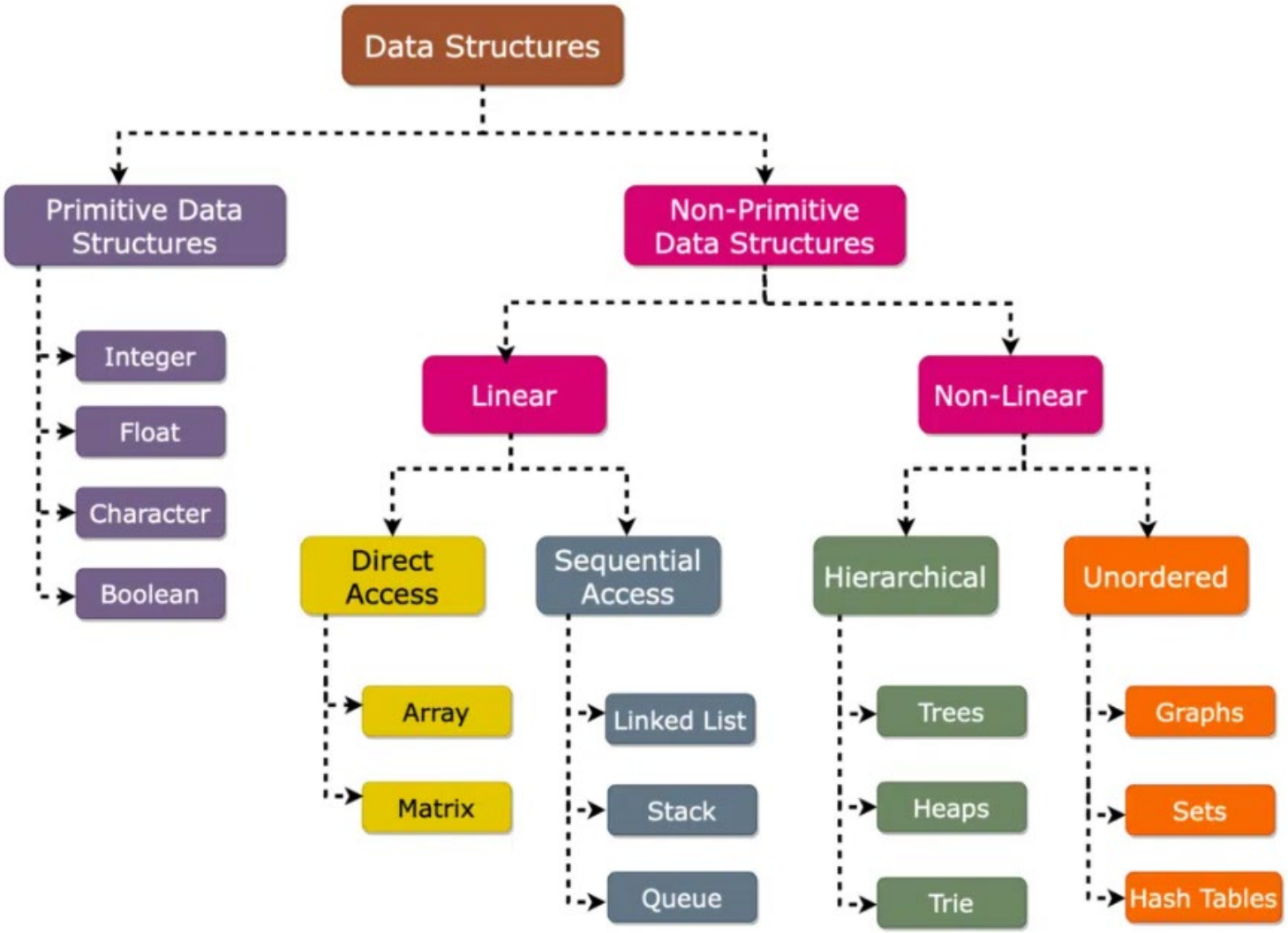
LINEAR DATA STRUCTURES	NON LINEAR DATA STRUCTURES
A type of data structure that arranges the data items in an orderly manner where the elements are attached adjacently	A type of data structure that arranges data in sorted order, creating a relationship among the data elements
Memory utilization is inefficient	Memory utilization is efficient
Single-level	Multi-level
Easier to implement	Difficult to implement
Ex: Array, linked list, queue, stack	Ex: tree, graph

Classification of Non-primitive Data Structure

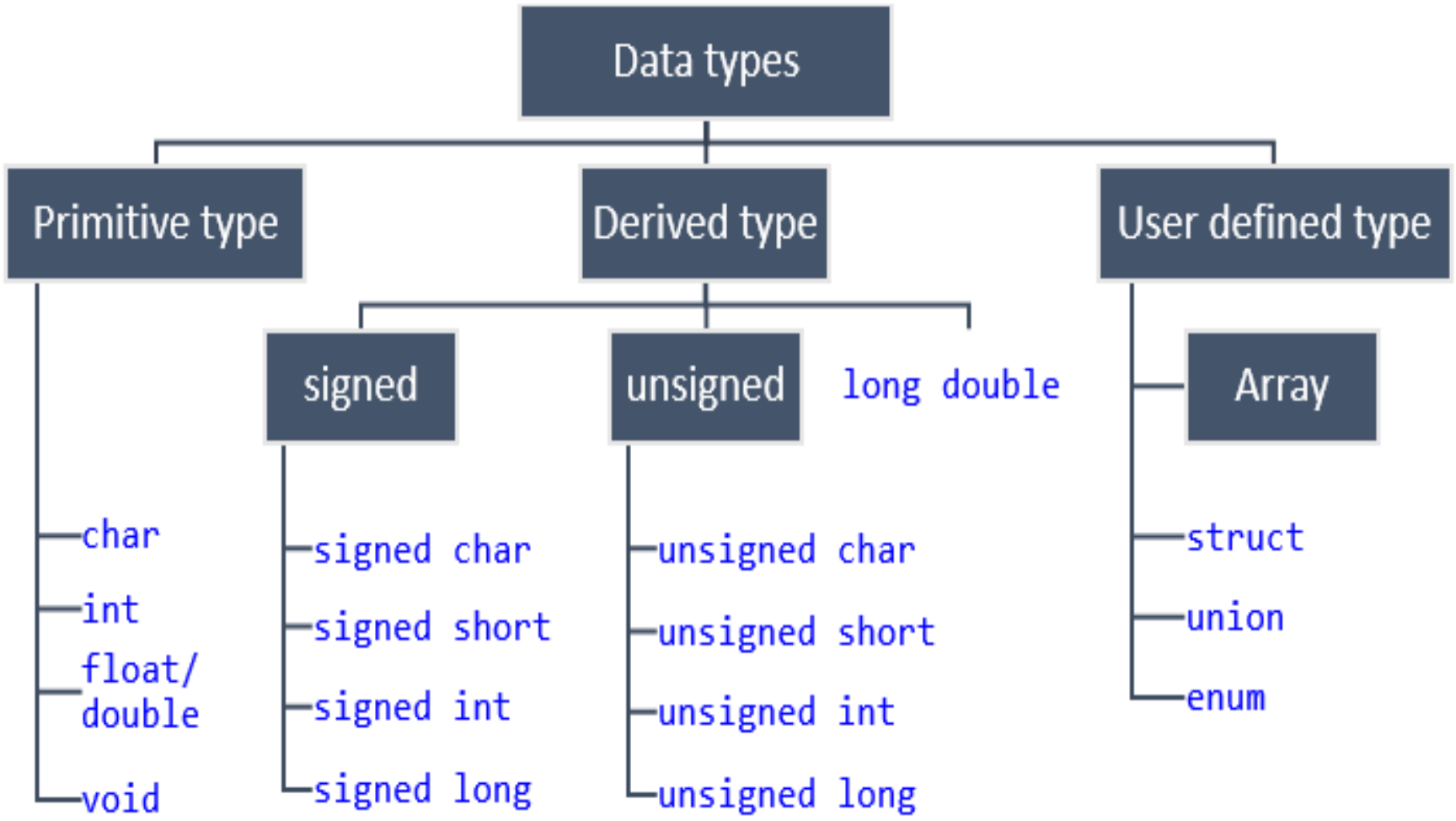
Linear data structures allow traversing through the items sequentially. On the other hand, in a **nonlinear data structure**, each element is attached to one or more elements creating a relationship among the items.



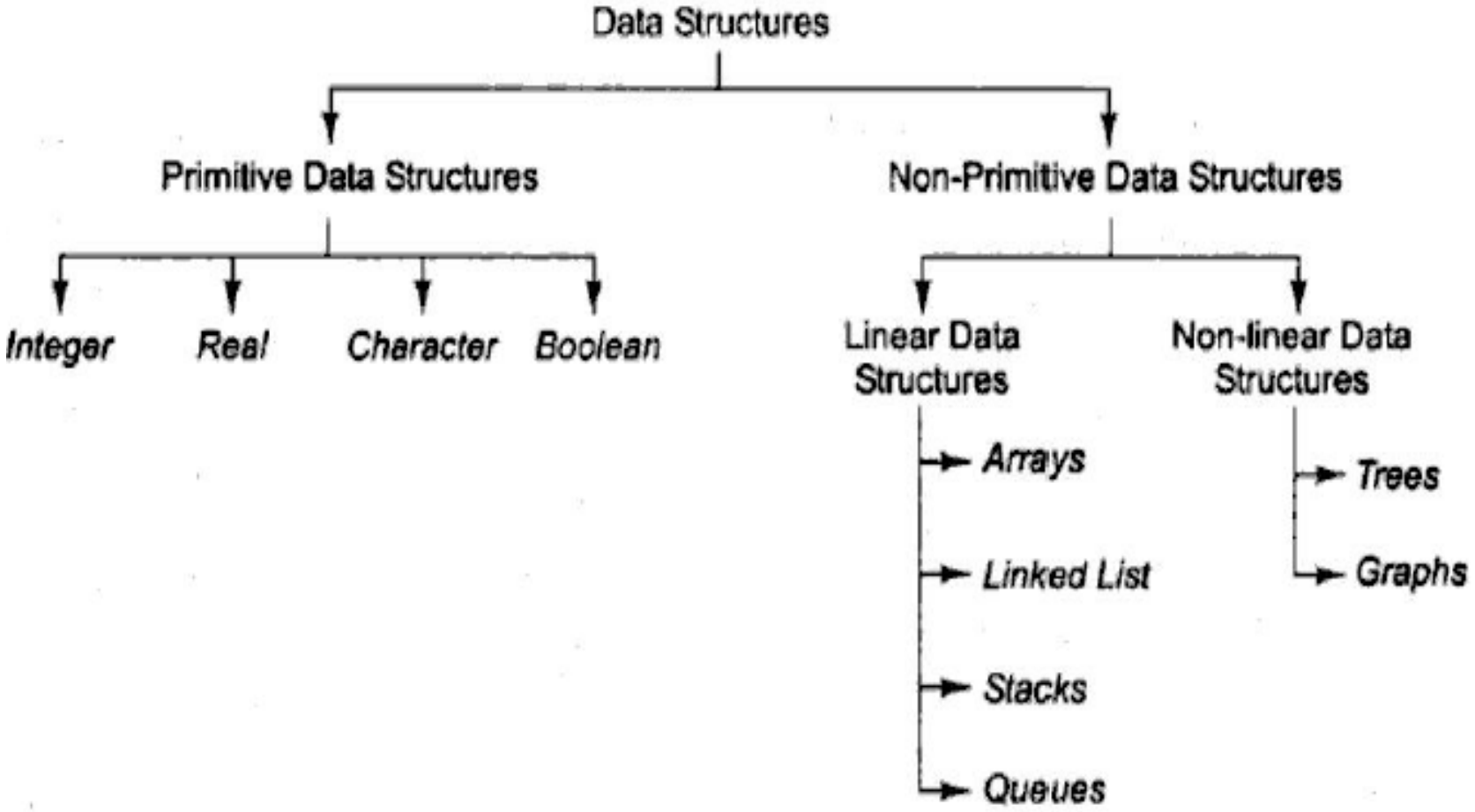
Classification of Non-primitive Data Structure



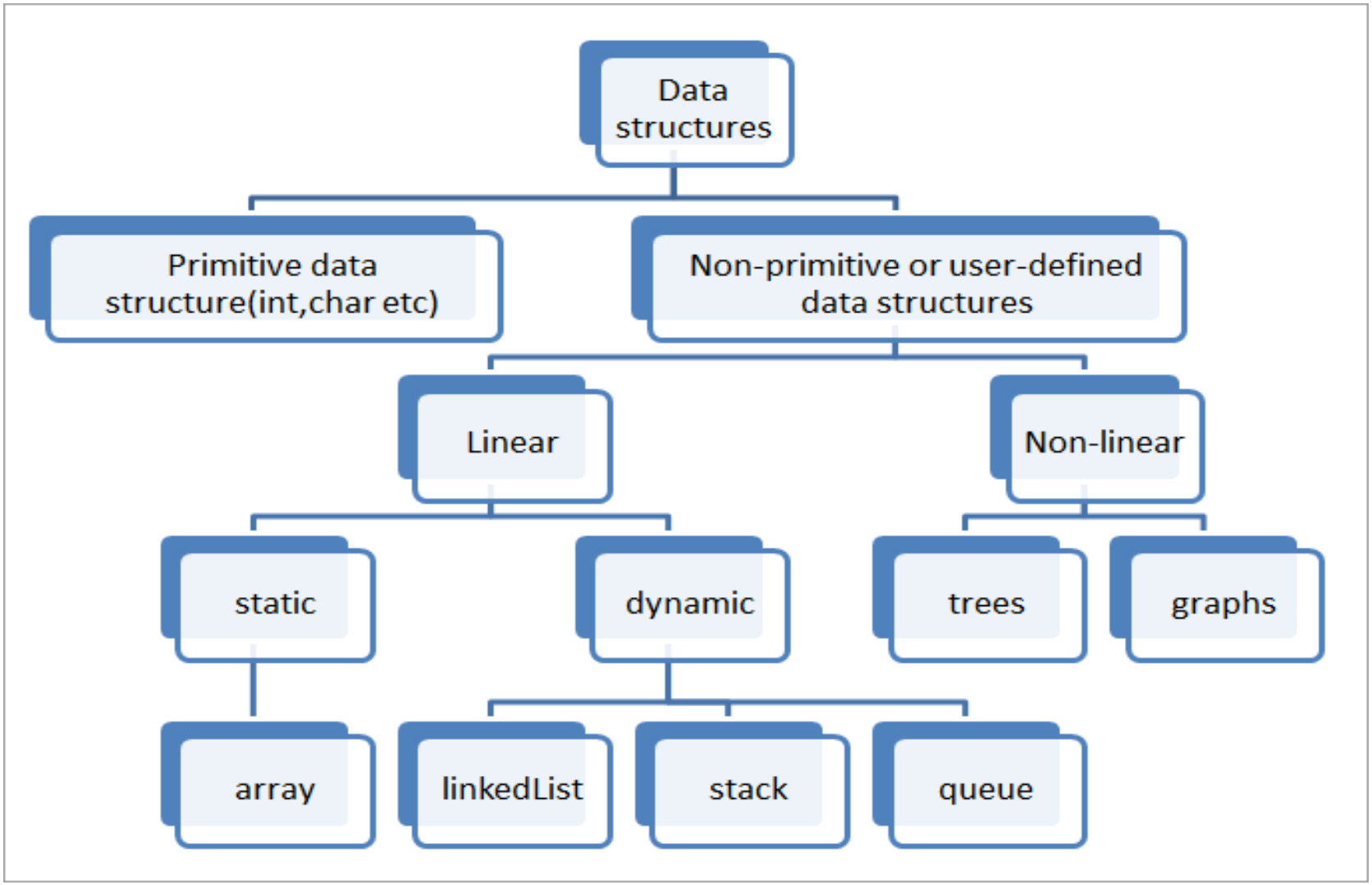
Data types in Programming language C



Classification of Data Structure



Data structures classification in C++



Static and dynamic data structures

- Linear data structures are further divided into static and dynamic data structures.

What is a Static Data structure?

- Static data structures usually have a fixed size and once their size is declared at compile time it cannot be changed.
- The content of the data structure can be modified but without changing the memory space allocated to it.

What is Dynamic Data Structure?

- Dynamic data structures can change their size dynamically and accommodate themselves.
- In Dynamic data structure the size of the structure in not fixed and can be modified during the operations performed on it.
- Dynamic data structures are designed to facilitate change of data structures in the run time.

Static Data Structure

- **Static data structures** are characterized by their **fixed size** and predefined memory allocation.
- This means that the amount of memory allocated to the structure is determined at **compile time** and remains constant throughout the program's execution.

Features of Static Data Structures:

- **Memory Allocation:** For static data structures, static memory is allocated at the compile time by the compiler, which is stored in the stack memory of the program.
- **Memory Deallocation:** Memory allocated to the static data structures deallocates when they go out of scope or the program ends.
- **Continuous Memory Allocation:** As we have discussed above, continuous memory is allocated to the static data structures, which means there is no need to store the structural information of the data structure or explicit data variable to store the information of the memory location.

Advantages of Static Data Structures:

- Static data structures have a fixed size and structure, making them simpler to implement and understand.
- Since the size is known in advance, the memory can be allocated efficiently.
- The behavior of static data structures is more predictable, making them suitable for real-time systems.
- Accessing elements in static data structures is generally faster than in dynamic data structures.

Disadvantages of Static Data Structures:

- The size and structure of static data structures are fixed, which can lead to inefficient memory usage if the data requirements change.
- Static data structures have a predefined size, limiting their ability to handle large and varying amounts of data.
- If the data size is smaller than the allocated memory, the unused memory is wasted.
- Expanding the size of a static data structure requires complex procedures, such as creating a new, larger structure and copying the data.

Data structures are essential in AI:

Data structures play a crucial role in artificial intelligence (AI) for organizing, managing, and processing data efficiently.

1. Efficiency in Handling Large Data Volumes: AI systems often deal with massive amounts of data. Efficient data structures like trees, graphs, and hash tables enable quick data access, search, insertion, and deletion, which are crucial for the performance of AI models.

2. Graphs for Networked Data: Graphs are fundamental in AI for modeling networked data. This includes social networks, recommendation systems, and semantic networks. They help in understanding and exploiting relationships between entities.

Data structures are essential in AI:

3. Trees for Decision Processes: Trees, particularly decision trees, are used in machine learning for building prediction models. They help in breaking down a dataset into smaller subsets while an associated decision tree is incrementally developed at the same time.

4. Arrays and Matrices for Mathematical Computations: Arrays and matrices are key in numerical operations which are central to AI algorithms, especially in areas like deep learning where matrix multiplication is a frequent operation.

Data structures are essential in AI:

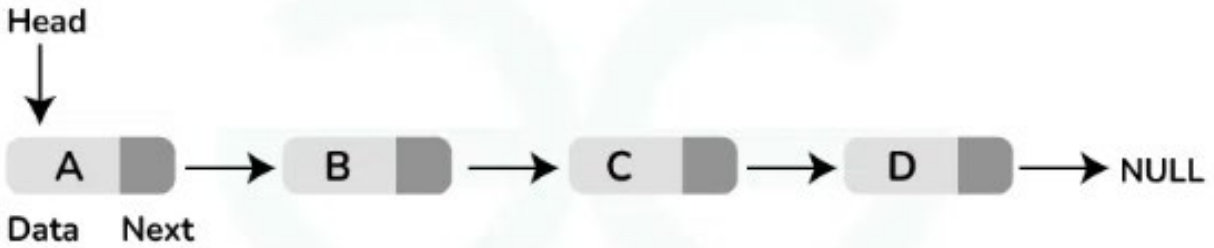
5. **Stacks and Queues for Control Structures**: AI algorithms, such as those used in depth-first search (DFS) or breadth-first search (BFS), rely on stacks and queues to manage the states to be explored.

6. **Heaps for Optimization Problems:** Heaps are crucial for managing prioritized queues, useful in numerous AI applications such as pathfinding algorithms in AI-driven games or in the optimization routines of machine learning algorithms.

7. **Hash Tables for Fast Access Data**: Hash tables are used for creating feature vectors from large datasets, allowing for quick access and manipulation of data, which is essential for real-time AI applications.

Dynamic Data Structure

- **Dynamic data structures** are flexible in size and can grow or shrink as needed during program execution.
- Dynamic data structures's adaptability makes them suitable for handling data of varying sizes or when the size is unknown beforehand.
- Examples of dynamic data structures include **linked lists, stacks, queues,** and **trees.**



Features of Dynamic Data Structures:

- **Dynamic Memory Allocation:** Dynamic data structures allocate memory at runtime, rather than being pre-allocated at compile-time. This memory is stored in the program's heap.
- **Flexible Memory Usage:** Unlike static data structures, the memory used by dynamic data structures is not limited to a fixed size. It can expand or contract as needed during program execution.
- **Manual Memory Management:** The memory allocated to dynamic data structures does not automatically deallocate when the structure goes out of scope. The programmer is responsible for manually deallocating this memory, often using functions like free() or delete(), to avoid memory leaks.
- **Non-Contiguous Memory:** The memory locations used by dynamic data structures are not guaranteed to be contiguous. This means additional metadata must be stored to track the locations of each element within the data structure.

Advantages of Dynamic Data Structures:

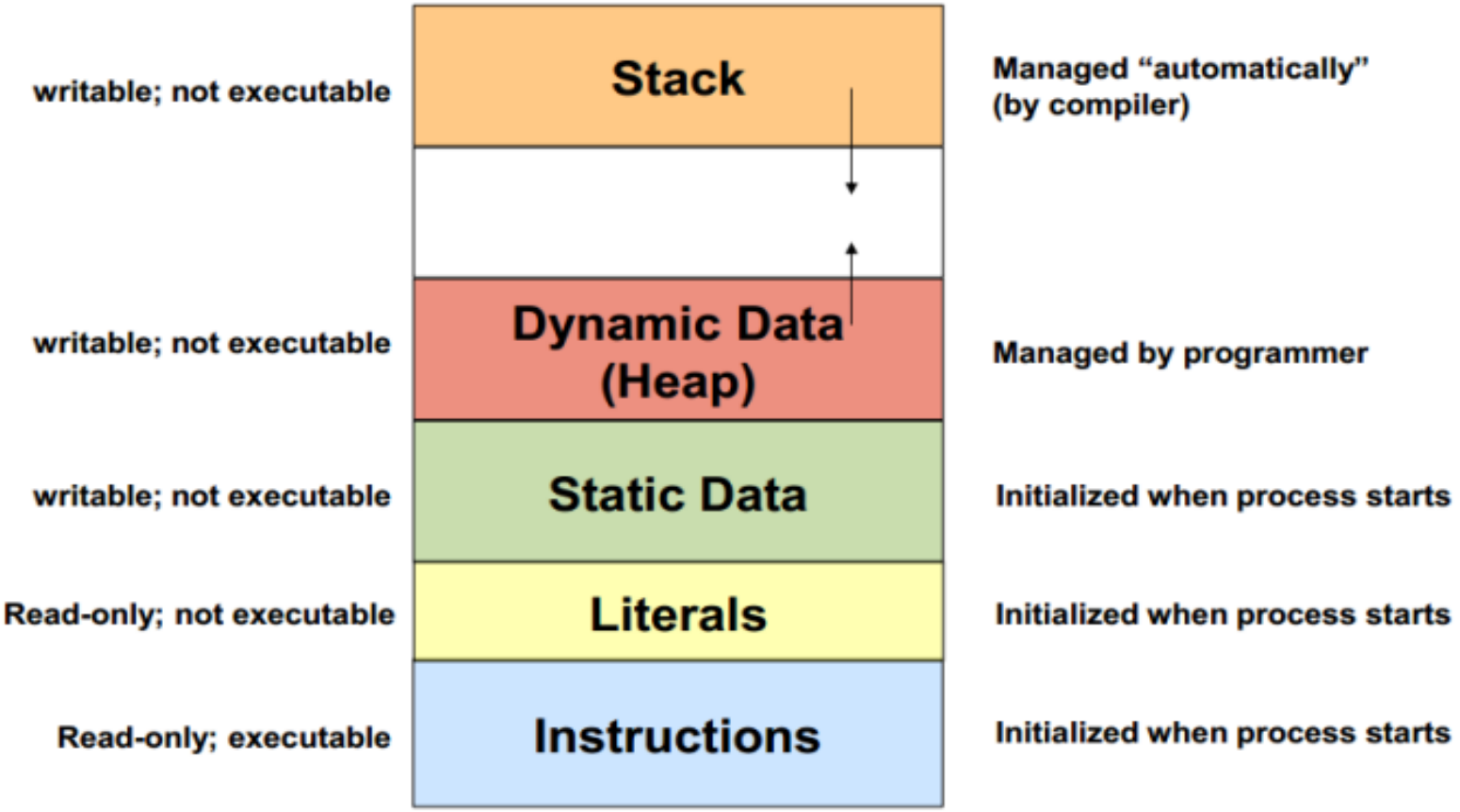
- Dynamic data structures can grow or shrink in size as needed, allowing for efficient use of memory.
- Dynamic data structures used to allocate and deallocate memory dynamically, preventing wastage of memory.
- Dynamic data structures can handle large and varying amounts of data without pre-defining the size.
- Dynamic data structures provide a simpler programming interface compared to static data structures.

Disadvantages of Dynamic Data Structures:

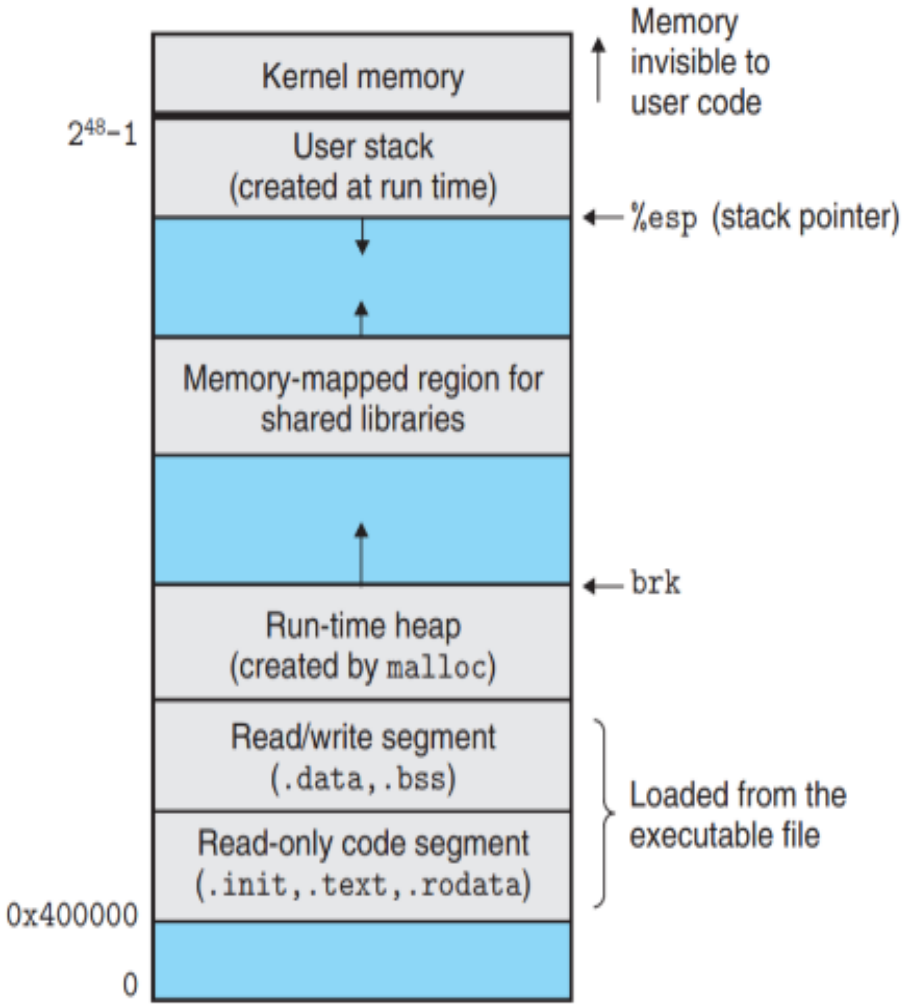
- Dynamic memory allocation and deallocation can incur additional overhead compared to static data structures.
- Dynamic data structures can be more complex to implement and maintain than static data structures.
- Improper handling of dynamic memory can lead to memory leaks, which can cause performance issues.
- The performance of dynamic data structures can be less predictable than static data structures, especially in real-time systems.

Data Structures in runtime environment

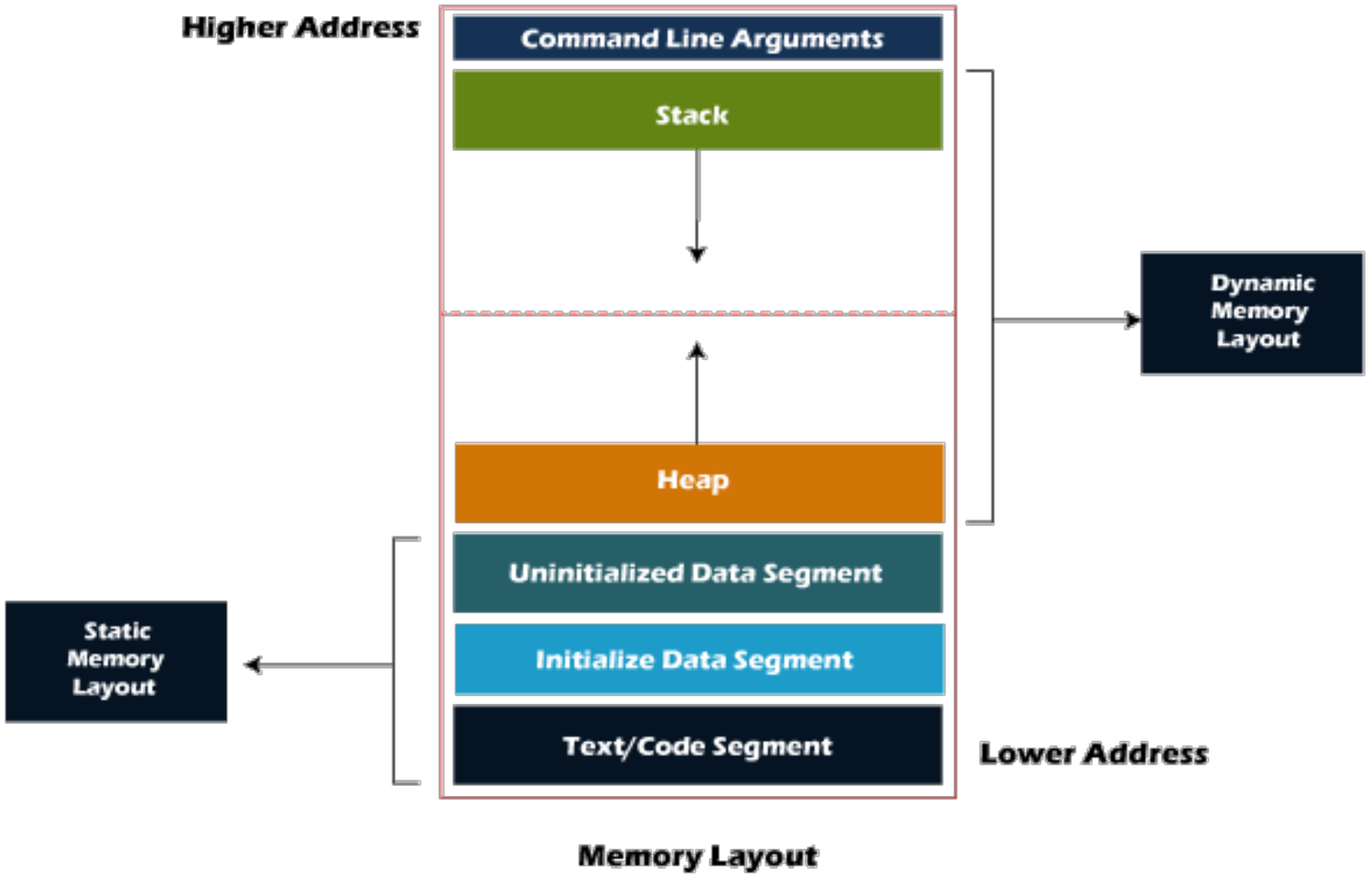
Heap is memory allocated at run-time for new items to **dynamic data structure**;



Memory map with LINUX OS



Memory Layout in C Program



Dynamic vs static data structure

- Dynamic data structure is a collection of data in memory that can grow or shrink in size. Example: Python's list.
- Static data structure is fixed in size at compilation time and cannot be resized to hold more items or to remove items to free up memory. Example: a static array in VB. Python does not have a built-in array type.
- Size of static structure fixed at compile-time whereas size of dynamic structure can change at run-time;
- Static structures can waste storage space/memory if the number of data items stored is small relative to the size of the structure whereas dynamic structures only take up the amount of storage space required for the actual data.
- Dynamic data structure is normally implemented using pointers which require extra memory space.
- Data insertion with dynamic data structure involves adjusting pointers while static data structure requires moving data items which can be more complex.

Static vs dynamic data structures

Static data structures	Dynamic data structures
Inefficient as memory is allocated that may not be needed	Efficient as the amount of memory varies as needed
Fast access to the data as the memory location is fixed when the program is written	Slower access to each element as the memory location is allocated at run time
Memory locations allocated will also be sequential which increases the access speed	Memory addresses are usually fragmented which reduces access speed
Structures are a fixed size which make them easy and predictable to work with	Structures vary in size so it is harder to find the length
The relationship between the different bits of data does not change	The relationship between different elements of data will change as the program is run
Only uses a couple of pointers so it can use less memory for large structures or if all locations are used	Uses a lot more pointers than a static structure so can be inefficient in some cases

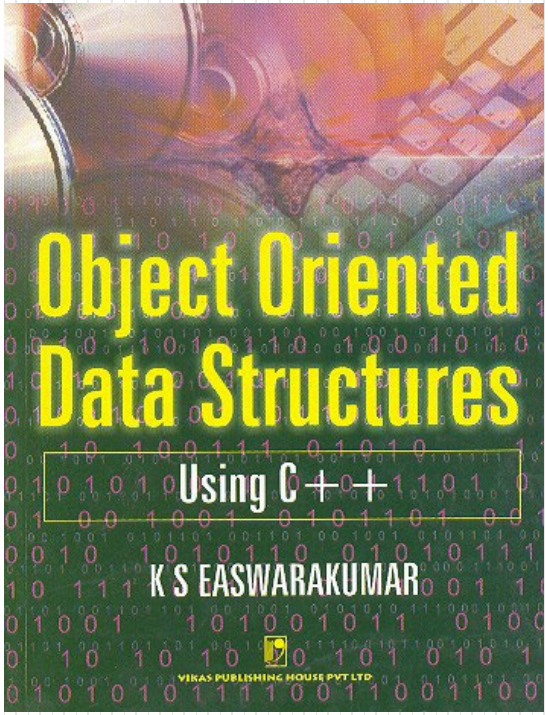
Operations on data structure

- **Create:-** The create operation results in reserving memory for program elements. This can be done by declaration statement. Creation of data structure may take place either during compile-time or run-time. malloc() function of C language is used for creation.
- **Destroy:-** Destroy operation destroys memory space allocated for specified data structure. free() function of C language is used to destroy data structure.
- **Selection:-** Selection operation deals with accessing a particular data within a data structure.
- **Updation:-** It updates or modifies the data in the data structure.
- **Insertion:** Adding a new record in the data structure.
- **Deletion:** Remove a record from the data structure.
- **Traverse:** Accessing each record in the data structure exactly once.

Operations on data structure

- **Searching:-** It finds the presence of desired data item in the list of data items, it may also find the locations of all elements that satisfy certain conditions.
- **Sorting:-** Sorting is a process of arranging all data items in a data structure in a particular order, say for example, either in ascending order or in descending order.
- **Merging:-** Merging is a process of combining the data items of two different sorted list into a single sorted list.
- **Splitting:-** Splitting is a process of partitioning single list to multiple list.
- **Traversal:-** Traversal is a process of visiting each and every node of a list in systematic manner.

Object-Oriented design & implementation of Data Structures

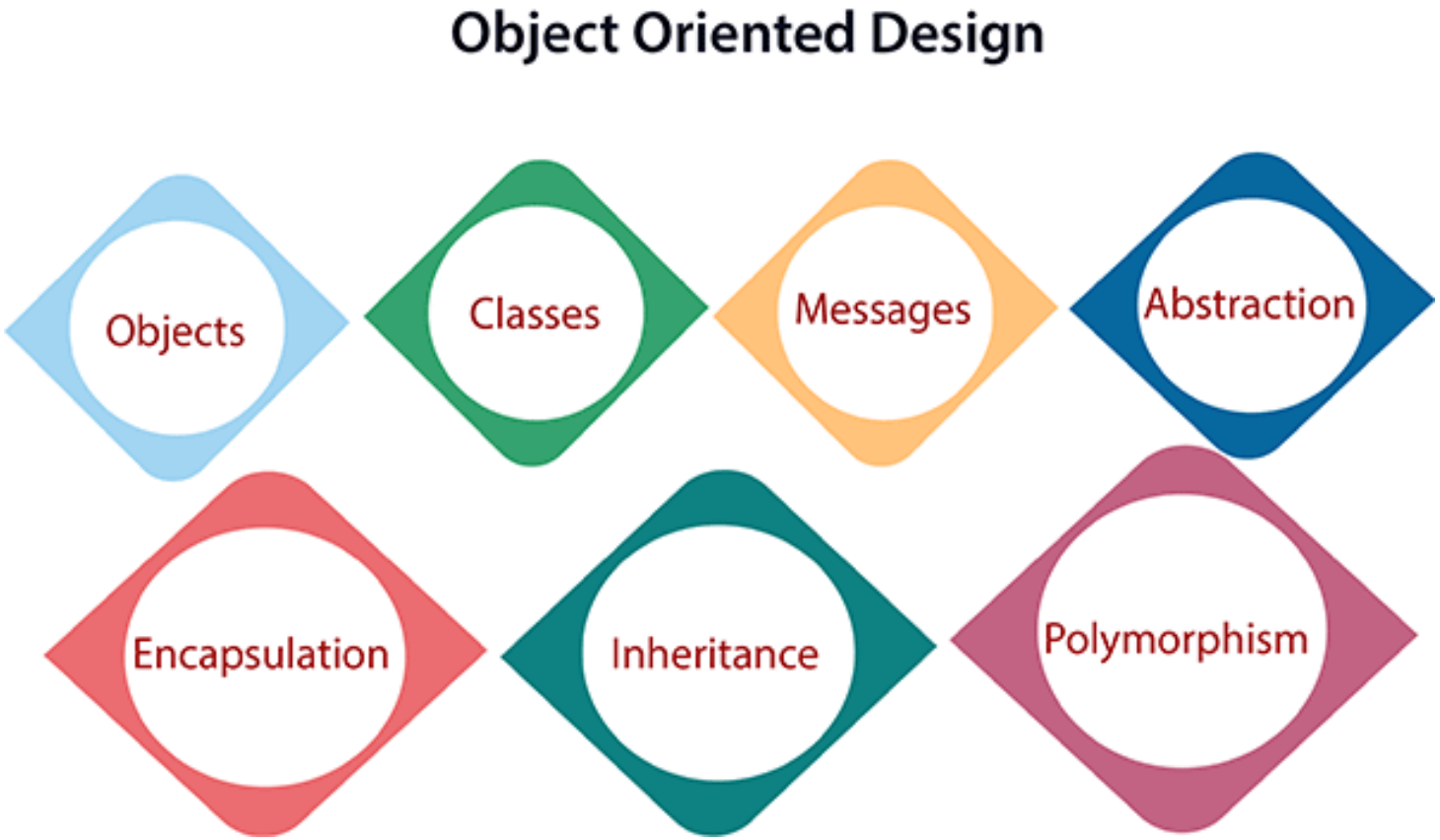


Object-Oriented Data Structures

- Data structure is a collection of data that has specific ways of accessing, storing, and organizing the data.
- A data structure also defines any relationship between multiple pieces of data.
- Data structure is **a collection of data that has specific ways of accessing, storing, and organizing the data**. A data structure also defines any relationship between multiple pieces of data.
- Object-oriented programming (OOP) is a programming paradigm based on the concept of objects, which are **data structures that contain data, in the form of fields (or attributes) and code, in the form of procedures, (or methods)**.

Object Oriented Design

Object-oriented design (OOD) is the process of using an object-oriented methodology to design a computing system or application.



Object Oriented Design

- **Objects:** All entities involved in the solution design are known as objects. For example, person, banks, company, STACK, QUEUE and users are considered as objects. Every entity has some attributes associated with it and has some methods to perform on the attributes.
- **Classes:** A class is a generalized description of an object. An object is an instance of a class. A class defines all the attributes, which an object can have and methods, which represents the functionality of the object.

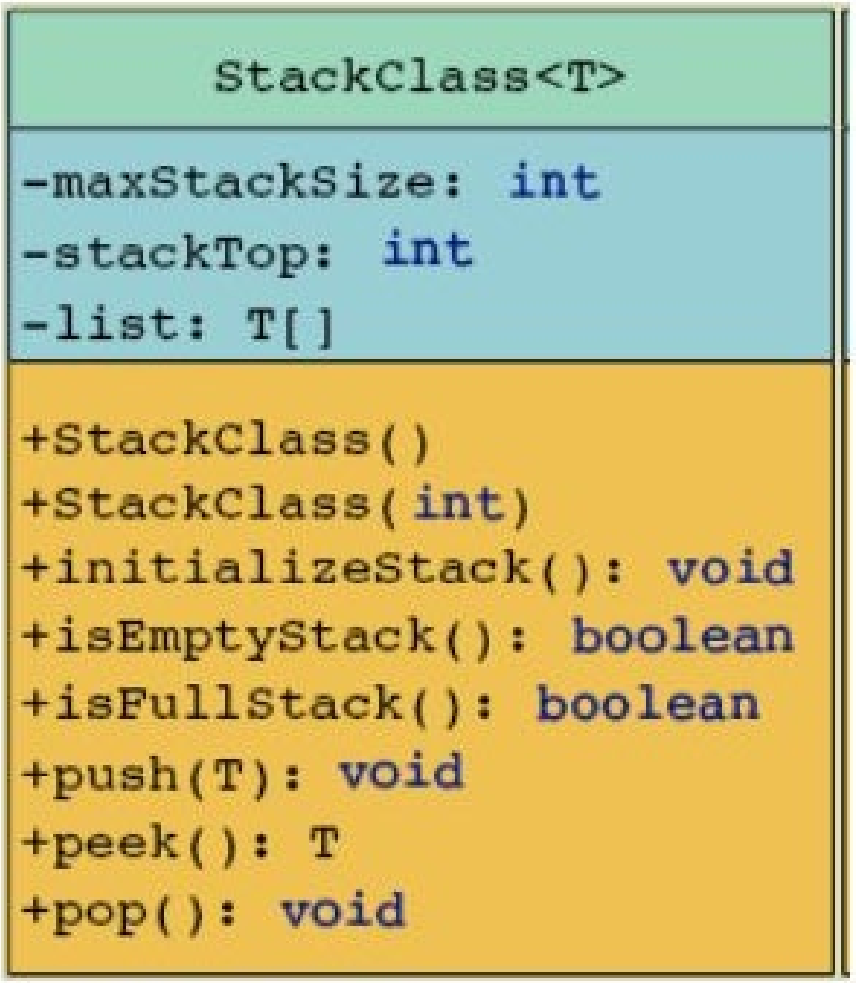
Object Oriented Design

- **Messages:** Objects communicate by message passing. Messages consist of the integrity of the target object, the name of the requested operation, and any other action needed to perform the function. Messages are often implemented as procedure or function calls.
- **Abstraction** In object-oriented design, complexity is handled using abstraction. Abstraction is the removal of the irrelevant and the amplification of the essentials.
- **Encapsulation:** Encapsulation is also called an information hiding concept. The data and operations are linked to a single unit. Encapsulation not only bundles essential information of an object together but also restricts access to the data and methods from the outside world.

Object Oriented Design

- **Inheritance:** OOD allows similar classes to stack up in a hierarchical manner where the lower or sub-classes can import, implement, and re-use allowed variables and functions from their immediate superclasses. This property of OOD is called an inheritance. This makes it easier to define a specific class and to create generalized classes from specific ones.
- **Polymorphism:** OOD languages provide a mechanism where methods performing similar tasks but vary in arguments, can be assigned the same name. This is known as polymorphism, which allows a single interface is performing functions for different types. Depending upon how the service is invoked, the respective portion of the code gets executed.

UML class diagram of class StackClass



Declaration of class currency in C++

```
class currency
{
   public:
      // constructor
      currency(signType theSign = plus,
               unsigned long theDollars = 0,
               unsigned int theCents = 0);
      // destructor
      ~currency() {}
      void setValue(signType, unsigned long, unsigned int);
      void setValue(double);
      signType getSign() const {return sign;}
      unsigned long getDollars() const {return dollars;}
      unsigned int getCents() const {return cents;}
      currency add(const currency&) const;
      currency& increment(const currency&);
      void output() const;
   private:
      signType sign;              // sign of object
      unsigned long dollars;      // number of dollars
      unsigned int cents;         // number of cents
};
```

The C++ Standard Template Library (STL)

The C++ Standard Template Library (STL)

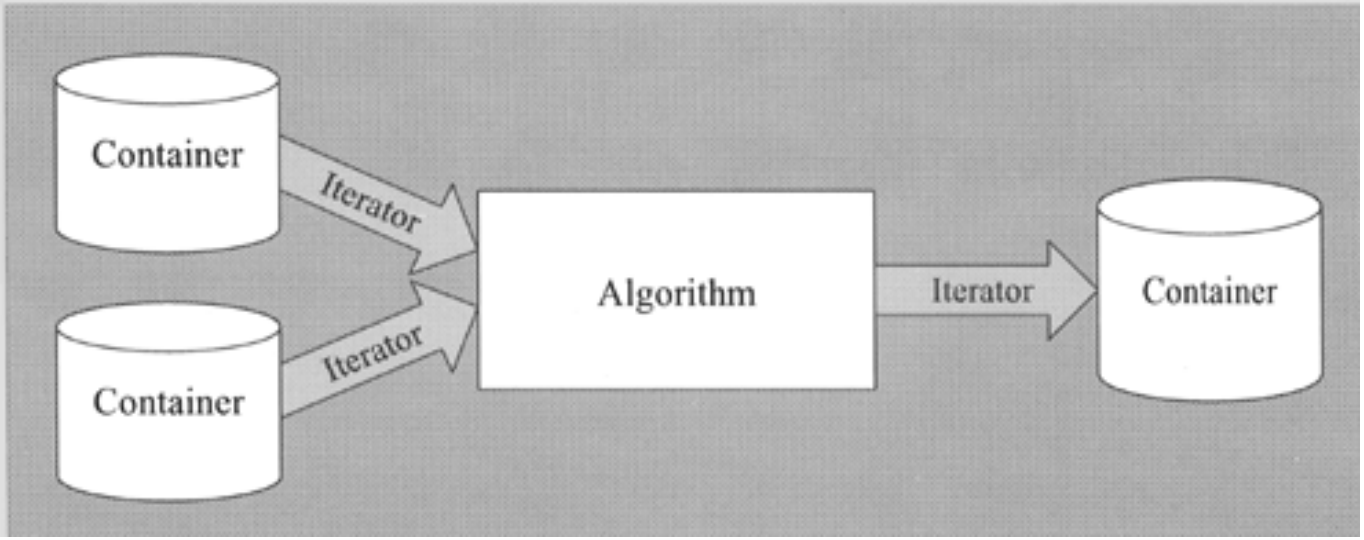
- The C++ Standard Library is a reference for C++ programmer to help them at every steps of their projects related to system programming.
- Standard Template Library provides **Algorithms and Containers**. Containers contain data and algorithms operate on the data which is in containers.
- The aim of object oriented language is combining algorithm and data into a class which is called object.
- **STL goes opposite way of OOP language**. It separates algorithms and data structures.
- Purpose of that is code reuse. We want one algorithm apply on different containers. And we want one container to use different algorithms.

Standard Template Library (STL)

- Standard Template Library (STL) is a collection of classes and functions, which is written in the core language. functions like lists, stacks, arrays, etc.
- The C++ programs in this section deals with the implementation of Standard Template Library like vectors, maps, set, queue, priority queue, strings, Sorting, stacks, arrays, deque, lists, Set_Intersection, Set_Difference, Set_Symmetric_difference, Prev_Permutataion and Next_Permutation in STL.

Containers

- Containers have different interfaces from each other. So if we want to apply one algorithm on different containers, we need to provide different implementations for that.

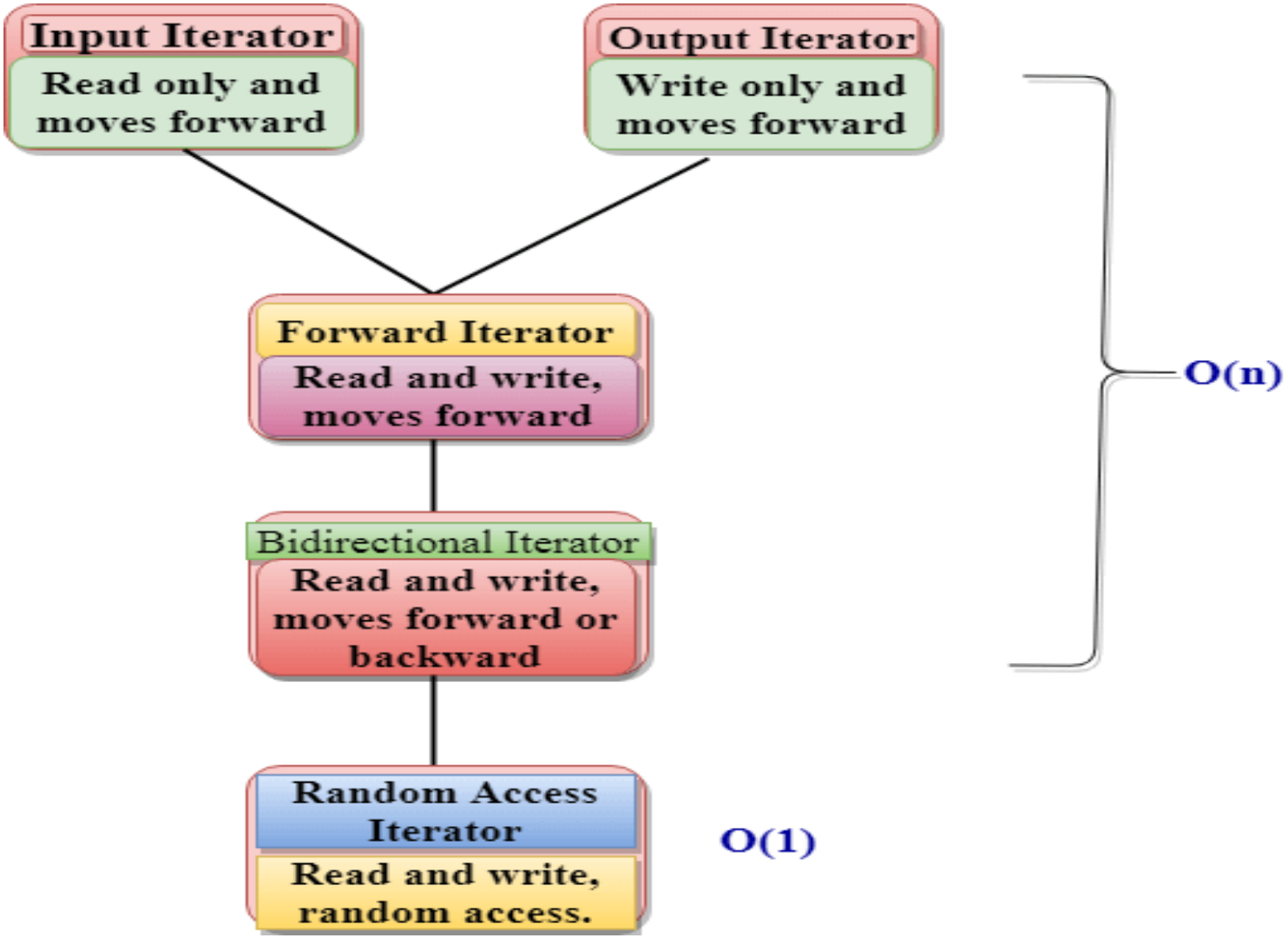


- If we want to apply one algorithm on different containers, we need to provide different implementations for that. So if we have N algorithms and M containers we need to provide N*M implementations.

Iterators

- To solve this problem STL library provides another group of modules called **Iterators**.
- Each container required to provide a common interface defined by iterators.
- Iterator can iterate each item inside a container. So the algorithm instead of working on containers directly it only works on the iterators.
- So the algorithm doesn't know about on which container it is working. It only knows about the iterator.
- This will very useful to reuse the code instead of writing N*M implementations (above mentioned example). Now we only need to provide N+M implementations.

Iterators


Input Iterator
Read only and moves forward
Output Iterator
Write only and moves forward
Forward Iterator
Read and write, moves forward
Bidirectional Iterator
Read and write, moves forward or backward
Random Access Iterator
Read and write, random access.
O(n)
O(1)

Stack in C++ STL

- Stacks are a type of container adaptors with LIFO(Last In First Out) type of working, where a new element is added at one end (top) and an element is removed from that end only.

- **template <class Type, class Container = deque<Type> > class stack**;

The functions associated with stack

empty() – Returns whether the stack is empty – Time Complexity : O(1)

size() – Returns the size of the stack – Time Complexity : O(1)

top() – Returns a reference to the top most element of the stack – Time Complexity : O(1)

push(g) – Adds the element 'g' at the top of the stack – Time Complexity : O(1)

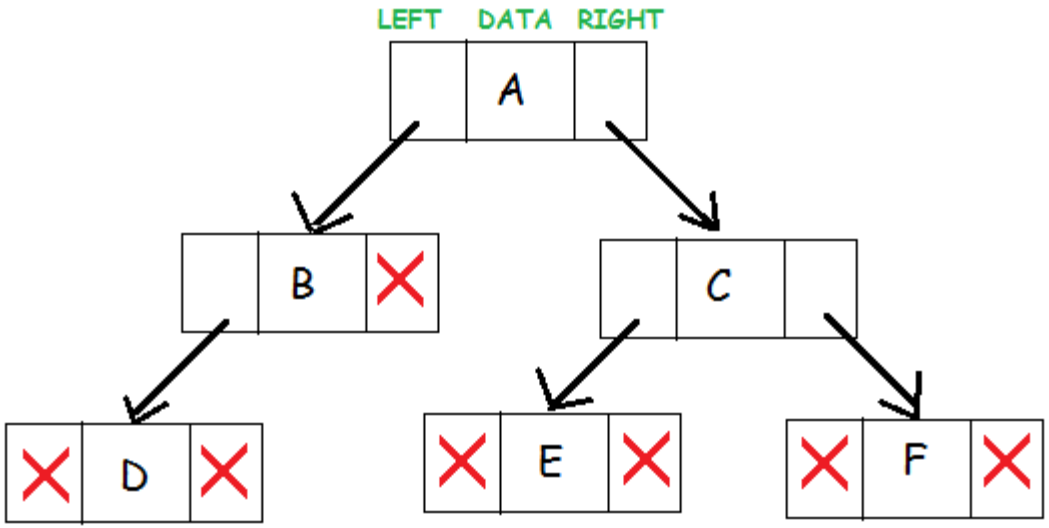
pop() – Deletes the top most element of the stack – Time Complexity : O(1)

Example

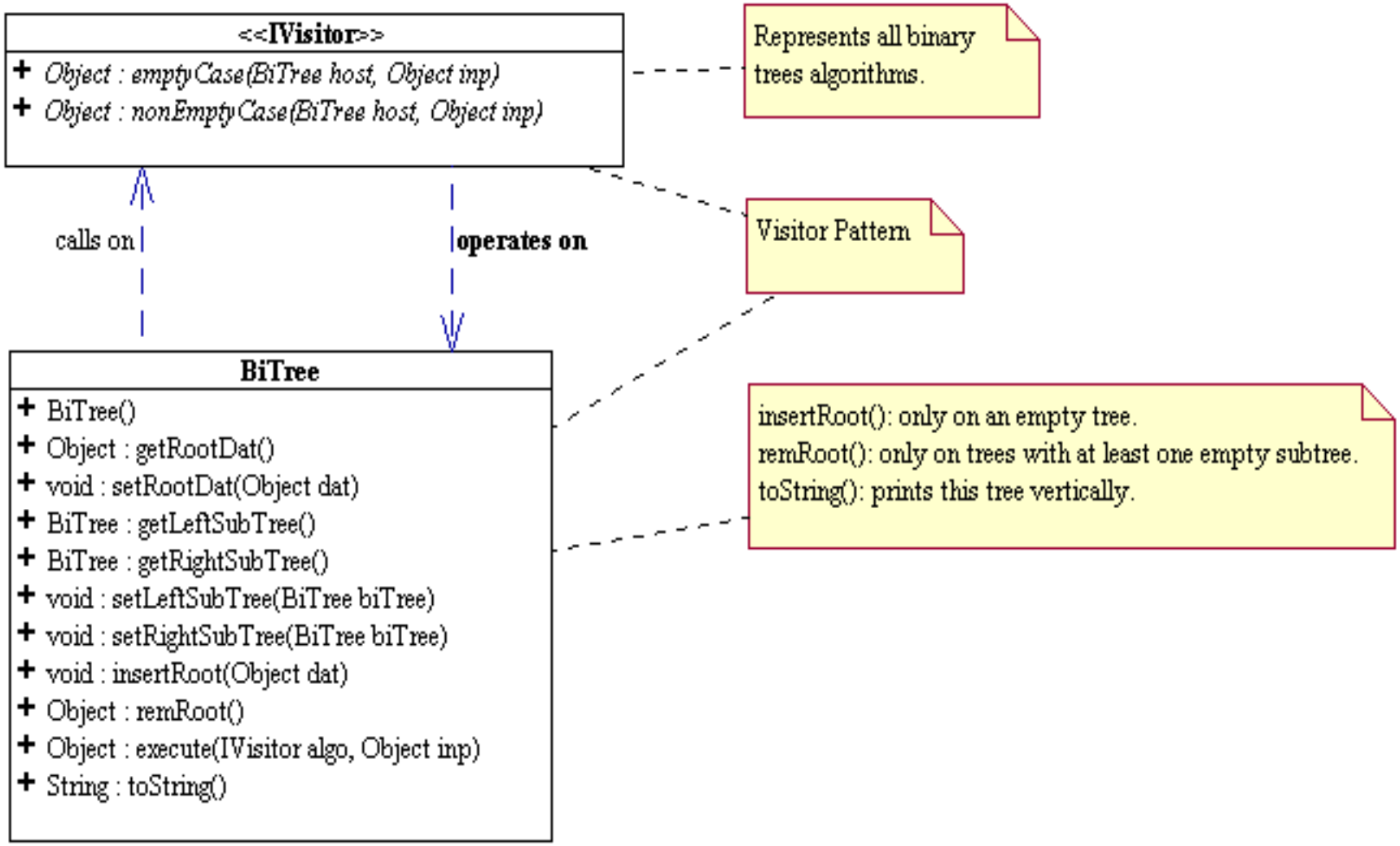
```
#include <iostream>
#include <stack>
using namespace std;
int main() {
        stack<int> stack;
        stack.push(21);
        stack.push(22);
        stack.push(24);
        stack.push(25);

                stack.pop();
        stack.pop();

        while (!stack.empty()) {
                cout << ' ' << stack.top();
                stack.pop();
        }
}
```

Binary Tree Structure

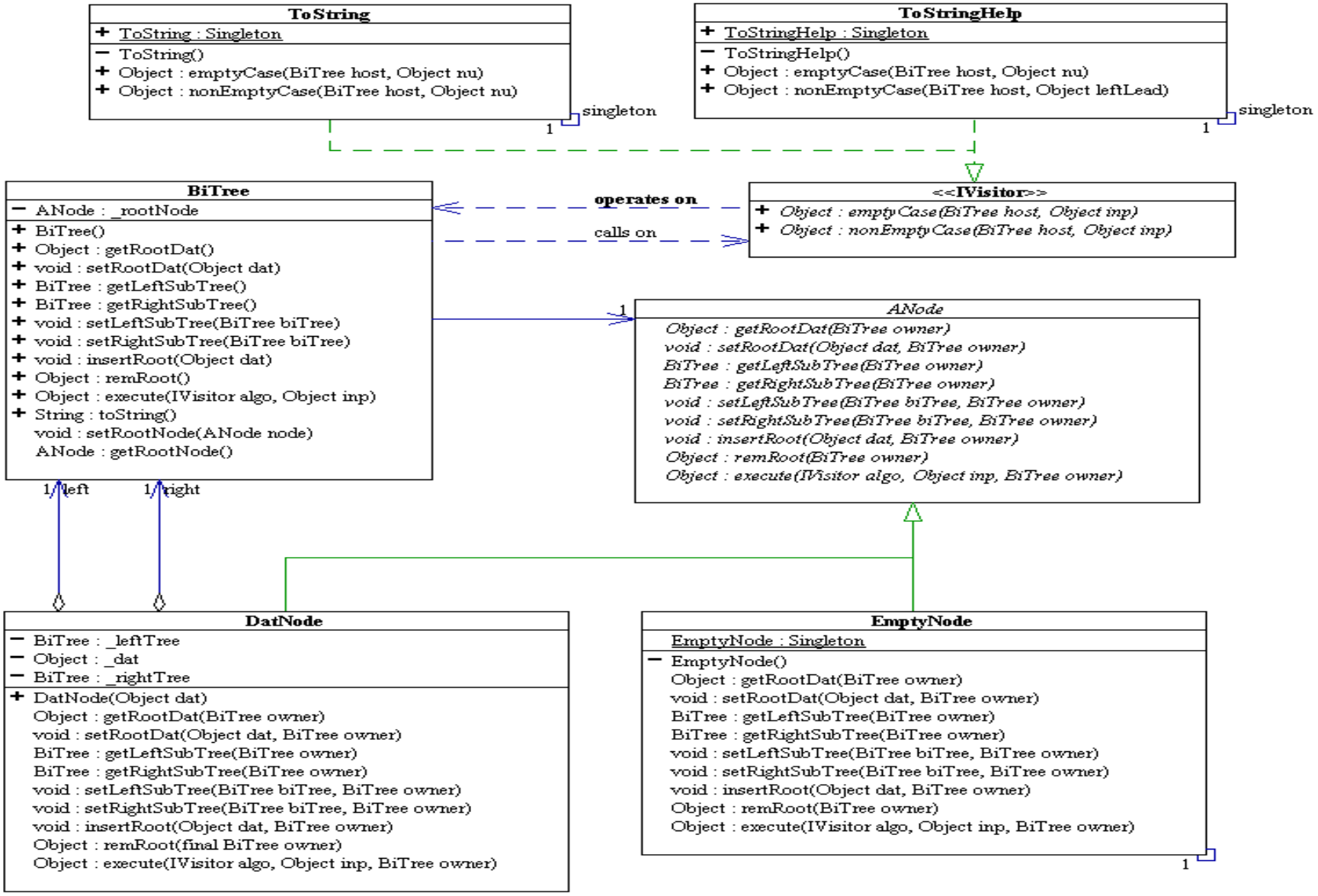
- **Binary Tree as a Recursive Data Structure**. A recursive data structure is **a data structure that is partially composed of smaller or simpler instances of the same data structure**.
- A binary tree is either **empty** or consists of a node called the **root** together with two binary trees called the **left-subtree** and the **right- subtree**.



The public interface of the binary tree framework



The implementation details of binary tree as UML class diagram



ADT *binaryTree*

AbstractDataType *binaryTree*
{
instances
collection of elements; if not empty, the collection is partitioned into a root, left subtree, and right subtree; each subtree is also a binary tree;

operations

empty() : return **true** if empty, return **false** otherwise;

size() : return number of elements/nodes in the tree;

preOrder(*visit*) : preorder traversal of binary tree; *visit* is the visit function to use;

inOrder(*visit*) : inorder traversal of binary tree;

postOrder(*visit*) : postorder traversal of binary tree;

levelOrder(*visit*) : level-order traversal of binary tree;
}

C++ abstract class *binaryTree*

```
template<class T>
class binaryTree
{
   public:
      virtual ~binaryTree() {}
      virtual bool empty() const = 0;
      virtual int size() const = 0;
      virtual void preOrder(void (*) (T *)) = 0;
      virtual void inOrder(void (*) (T *)) = 0;
      virtual void postOrder(void (*) (T *)) = 0;
      virtual void levelOrder(void (*) (T *)) = 0;
};
```

C++ STL and binary search trees

The C++ Standard Template Library provides these **containers** (i.e., data structures):

vector	queue	map
deque	priority_queue	multimap
list	set	bitset
stack	multiset	

Of these, **set** is one that is implemented using a balanced binary search tree (typically a red-black tree)

Containers

- A container is **an object that stores a collection of objects of a specific type**. For example, if we need to store a list of names, we can use a vector .
- A container is a holder object that stores a collection of other objects (its elements). They are implemented as class templates, which allows a great flexibility in the types supported as elements.
- The container manages the storage space for its elements and provides member functions to access them, either directly or through iterators (reference objects with similar properties to pointers).
- C++ STL provides different types of containers based on our requirements.

Container class templates

Sequence containers:

array
Array class (class template)

vector
Vector (class template)

deque
Double ended queue (class template)

forward_list
Forward list (class template)

list
List (class template)

Container adaptors:

stack
LIFO stack (class template)

queue
FIFO queue (class template)

priority_queue
Priority queue (class template)

Container class templates

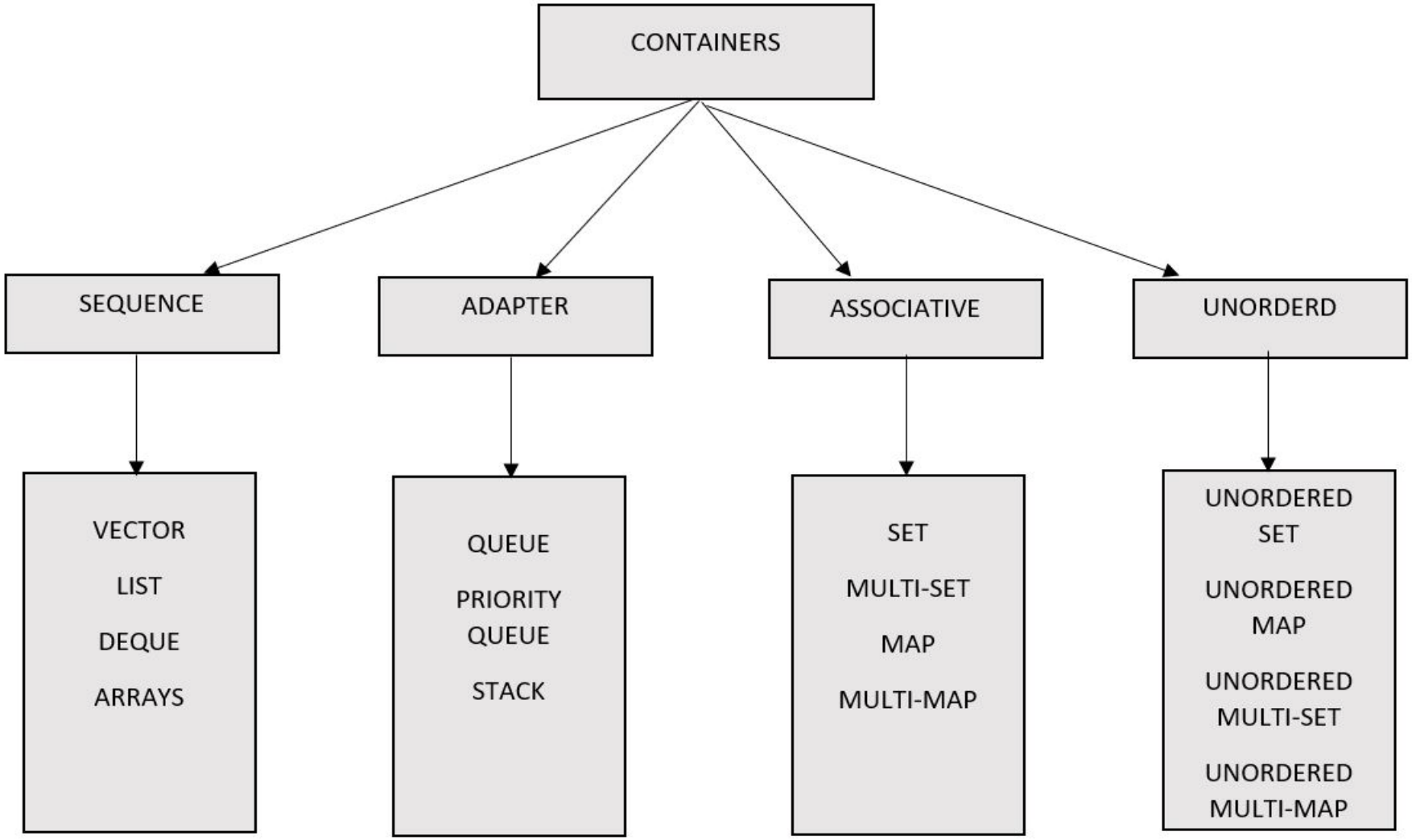
Associative containers:

set
Set (class template)
multiset
Multiple-key set (class template)
map
Map (class template)
multimap
Multiple-key map (class template)

Unordered associative containers:

unordered_set
Unordered Set (class template)
unordered_multiset
Unordered Multiset (class template)
unordered_map
Unordered Map (class template)
unordered_multimap
Unordered Multimap (class template)

Container class templates C++



Choice of Programming Language: TIOBE Index for August 2023

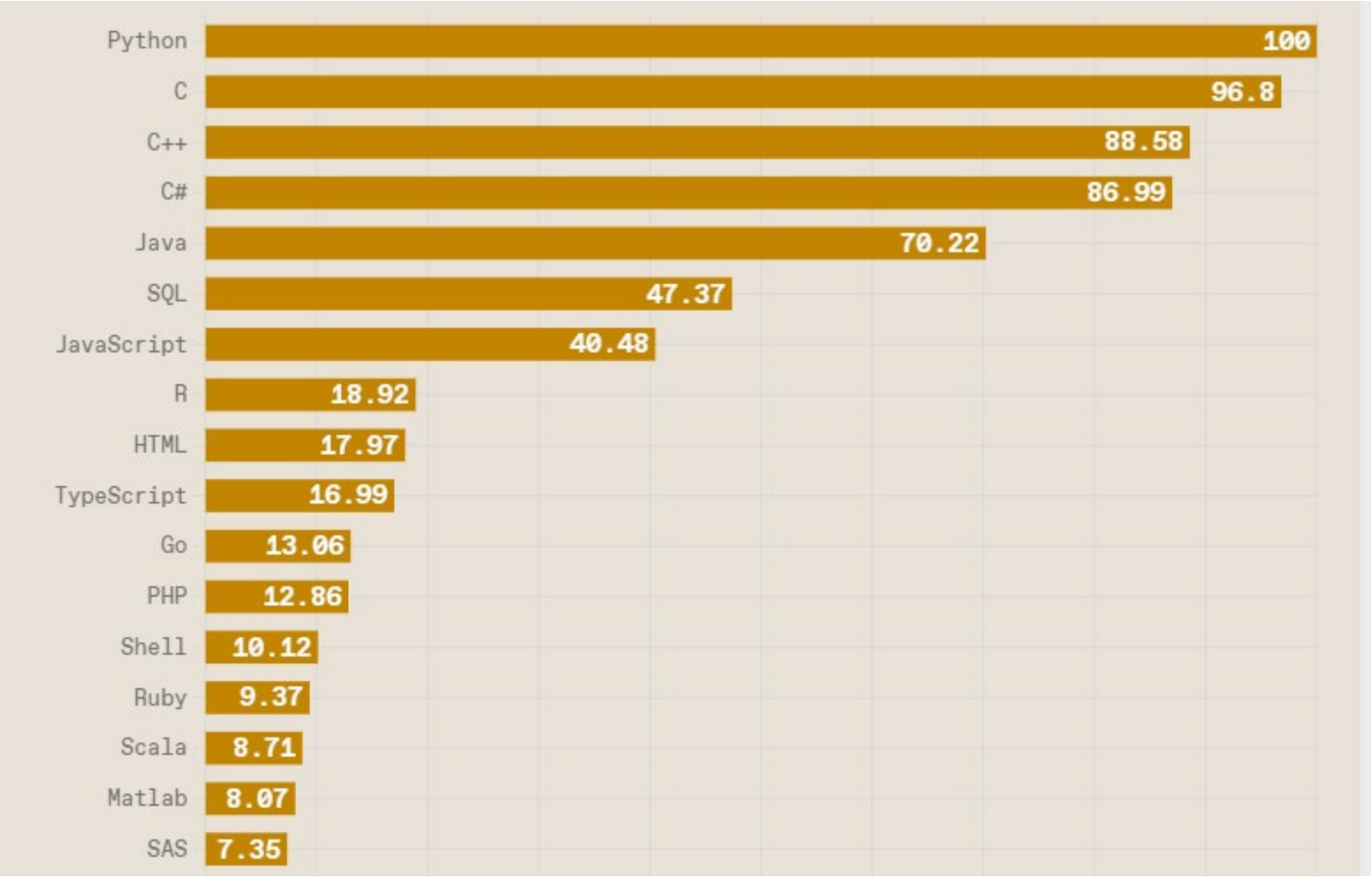
Aug 2023	Aug 2022	Programming Language	Ratings
1	1	Python	13.33%
2	2	C	11.41%
3	4	C++	10.63%
4	3	Java	10.33%
5	5	C#	7.04%
6	8	JavaScript	3.29%
7	6	Visual Basic	2.63%
8	9	SQL	1.53%
9	7	Assembly language	1.34%
10	10	PHP	1.27%

PYPL PopularitY of Programming Language

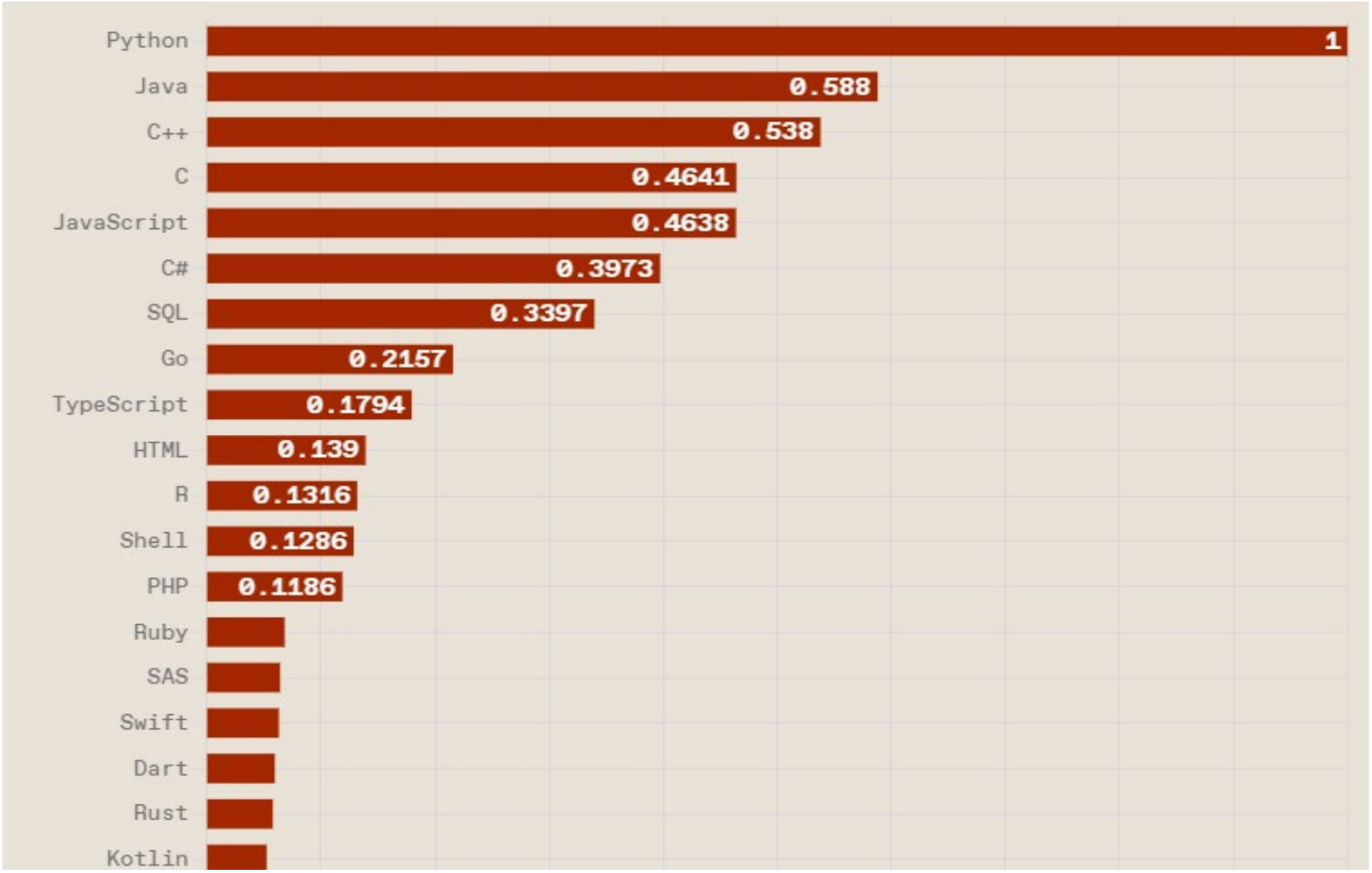
Worldwide, Aug 2023 :

Rank	Change	Language	Share	1-year trend
1		Python	28.04 %	+0.3 %
2		Java	15.78 %	-1.3 %
3		JavaScript	9.27 %	-0.2 %
4		C#	6.77 %	-0.2 %
5		C/C++	6.59 %	+0.4 %
6		PHP	5.01 %	-0.4 %
7		R	4.35 %	+0.0 %
8		TypeScript	3.09 %	+0.3 %
9	↑↑	Swift	2.54 %	+0.5 %
10		Objective-C	2.15 %	+0.1 %

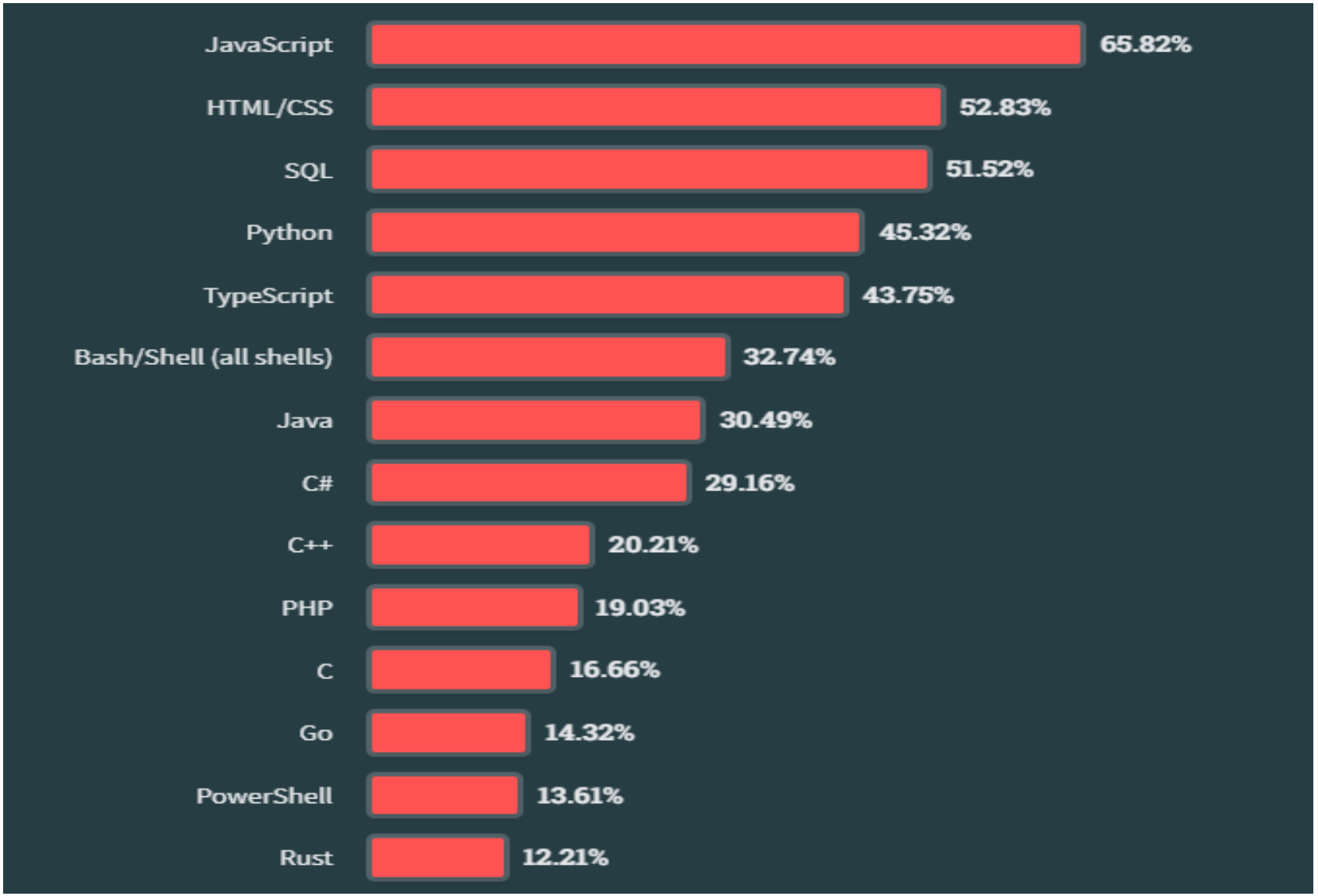
IEEE Spectrum's Top Programming Languages 2022



IEEE Spectrum's Top Programming Languages 2023



Stackoverflow Developer Surveys L based on Professional Developers

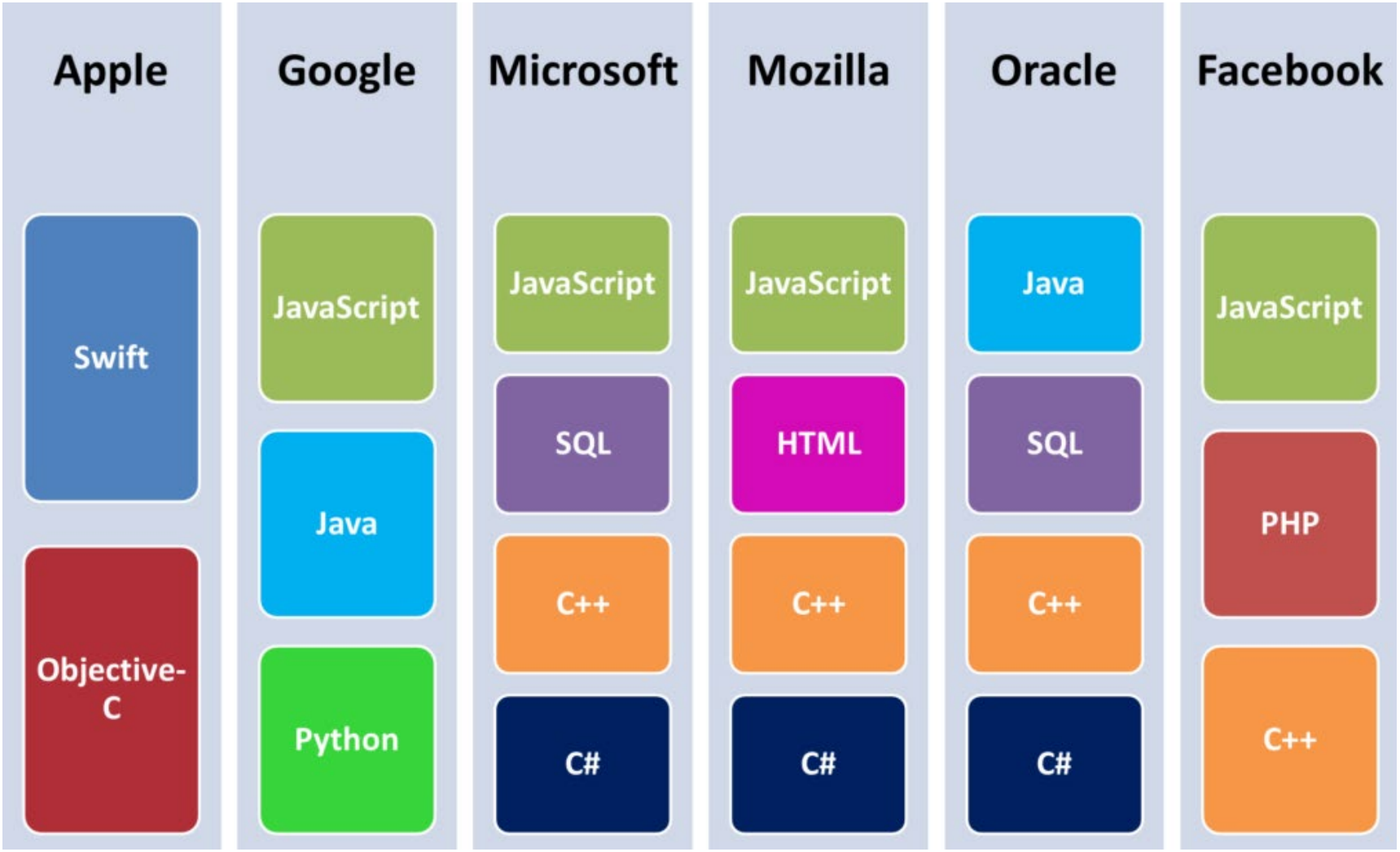


Which programming language is in demand in 2023?

Programming Language to learn based on your career goals

Language	Development					
	Front-end web	Back-end web	Mobile	Game	Desktop Application	System Programming
JavaScript	✓	✓	✓			
Elm	✓					
TypeScript	✓			✓		
Scala		✓			✓	
Python		✓			✓	
Go		✓			✓	✓
Ruby		✓				
Swift			✓			
Java			✓			
Objective C			✓			
Unity				✓		
Rust						✓

Top languages used by major companies



Top Programming Languages For Cloud Computing

JavaScript	Node.js	Python
C	GoLang	Java
.NET	PHP	Ruby on Rails

Popular cloud programming languages and frameworks

JavaScript
A dynamic programming language used for interactive web development

Node.js
An open source JavaScript runtime environment ideal for microservices development and deployment

C
A general-purpose, procedural programming language widely used to develop applications and OSes

Python
A high-level, general-purpose programming language popular with data scientists and AI development teams

Java
A class-based, object-oriented and server-side programming language popular for cloud-native applications, Android applications and IoT

.NET
An open source and cross-platform development framework ideal for web and mobile apps, as well as microservices

Go
An open source, compiled programming language used for back-end services, such as distributed networking

Swift
A general-purpose programming language used for mobile application development, specifically iOS

Unity
A development platform and game engine popular for game development and virtual reality applications

Kotlin
A statically-typed, general-purpose programming language used for mobile application development, specifically Android

Rust
A low-level, memory-efficient programming language often used for embedded and bare-metal development

Laboratory Assignment

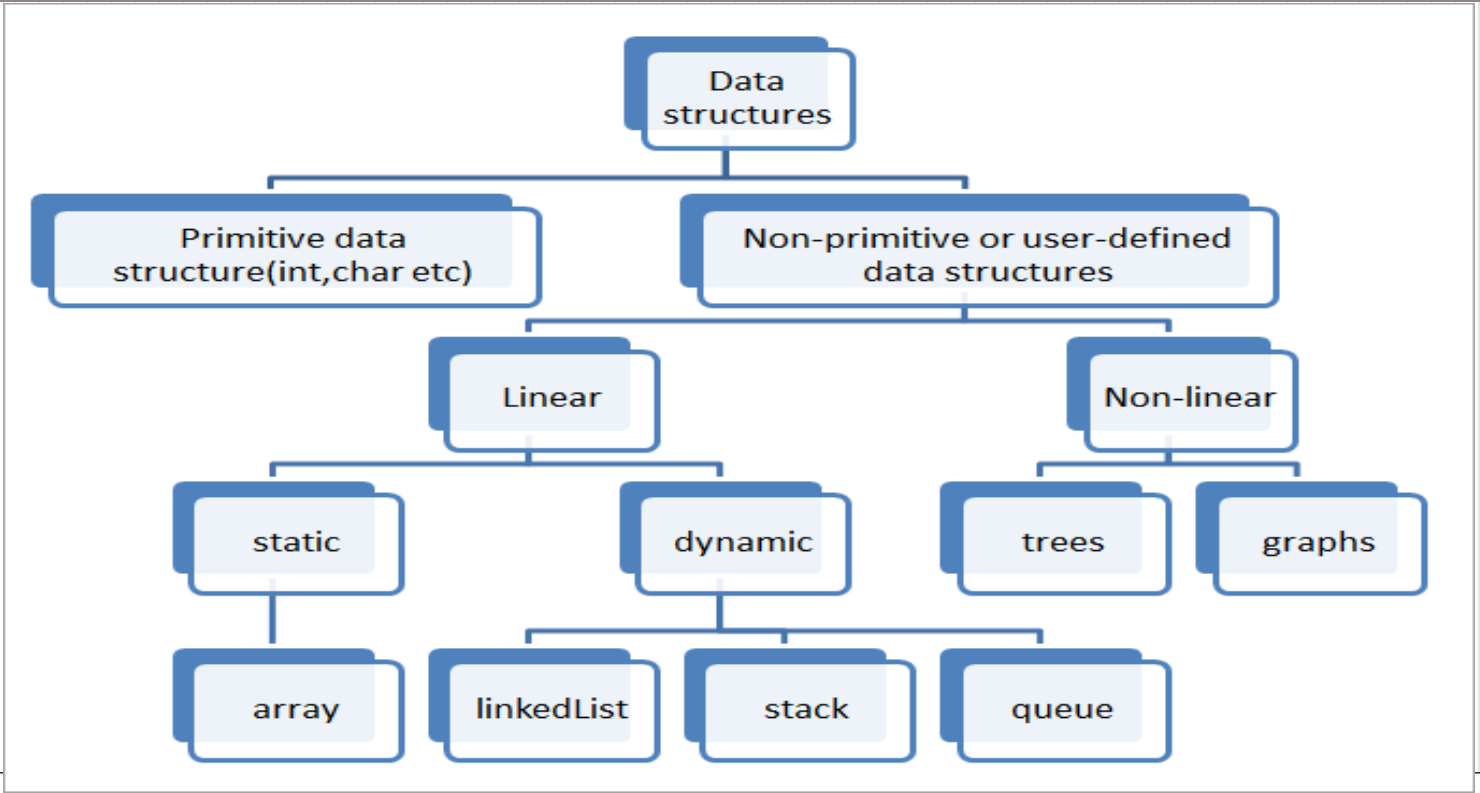
- **Q.18. Create a Binary Tree data structure, to be used for different application**
- This assignment is an example of the kind of computation one does on binary trees. You have to use C++ for the development of a program that does the following. The program takes commands from users, as usual; the following commands are supported:
- exit: quits the program
- newtree: asks the user to enter an *inorder sequence* and a *preorder sequence* (of strings) of a binary tree. As we have discussed in class, these two orders are sufficient to uniquely reconstruct the tree. Once the user has entered the two sequences, the tree is the "current tree", and the old tree (from a previous typing of newtree command) is discarded.
- preorder: prints the current tree in preorder
- inorder: prints the current tree in inorder
- postorder: prints the current tree in postorder
- levelorder: prints the current tree in levelorder
- ver: prints the current tree vertically in a "pretty format."
- hor: prints the current tree horizontally in a "pretty format"
- sym: prints the current tree symmetrically in a "pretty format"

Thanks for Your Attention!

References

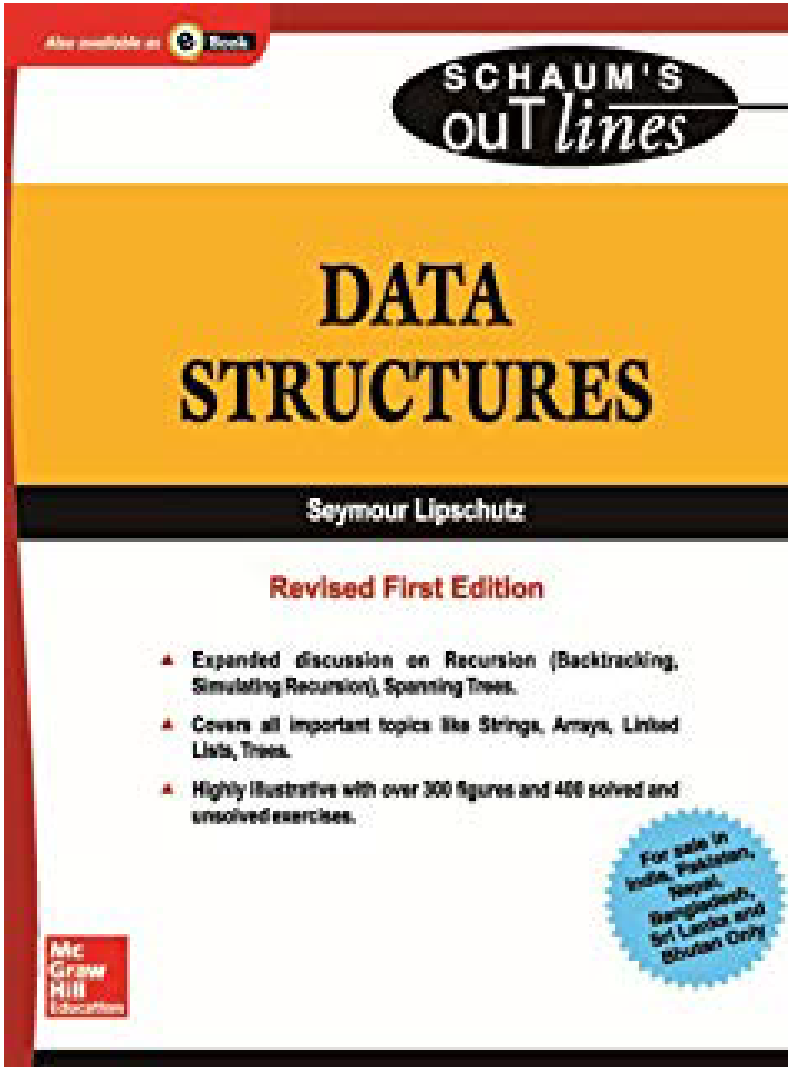
- https://www.sanfoundry.com/cpp-program-implement-stack-stl/
- https://www.knowledgehut.com/blog/cloud-computing/cloud-computing-programming-languages
- https://altitudeaccelerator.ca/best-programming-language-coding-basics-startups/
- https://www.javatpoint.com/memory-layout-in-c
- https://www.geeksforgeeks.org/static-and-dynamic-data-structures-in-java-with-examples/#features-of-static-data-structures
- https://www.enjoyalgorithms.com/blog/graph-representation-in-data-structures

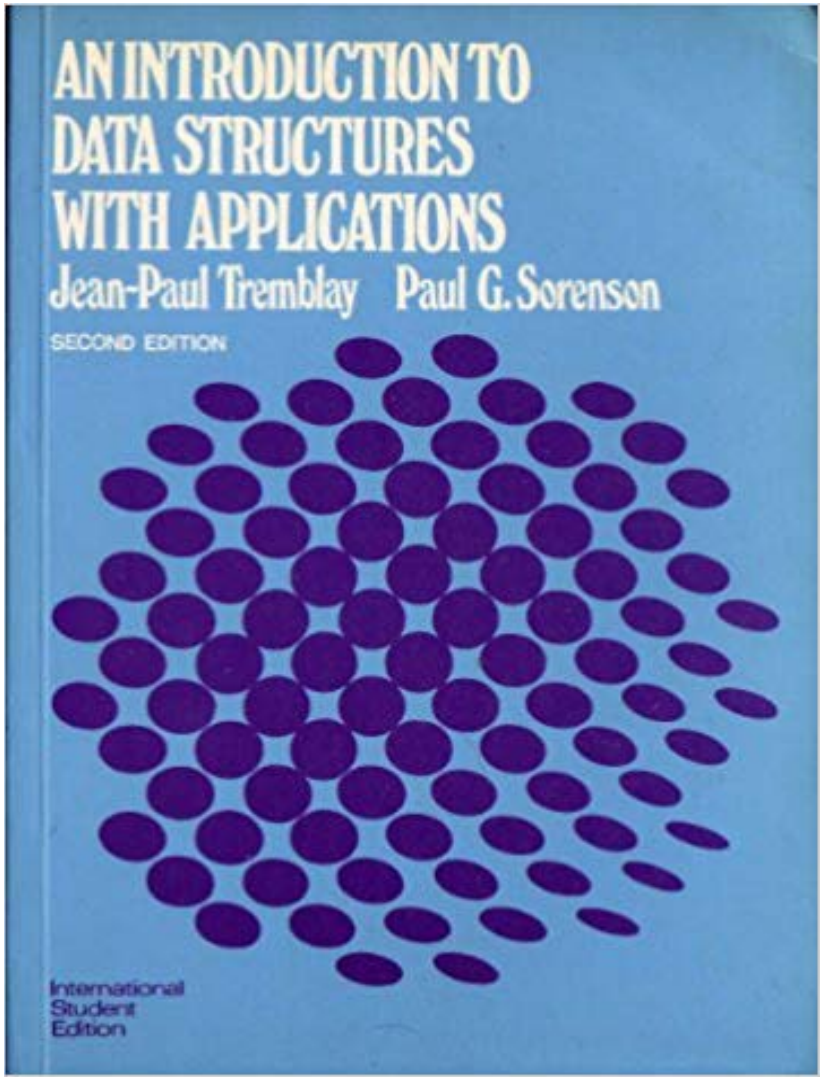
Data Structures References



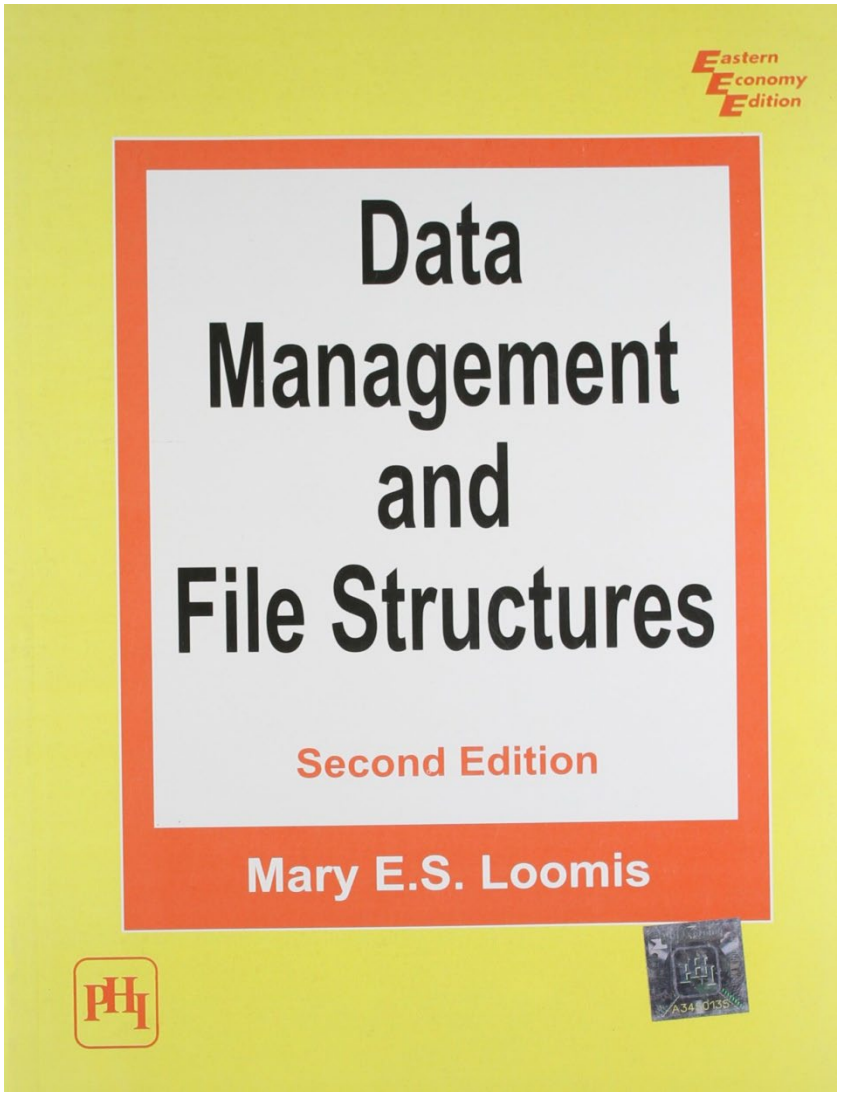
Suggested Reading

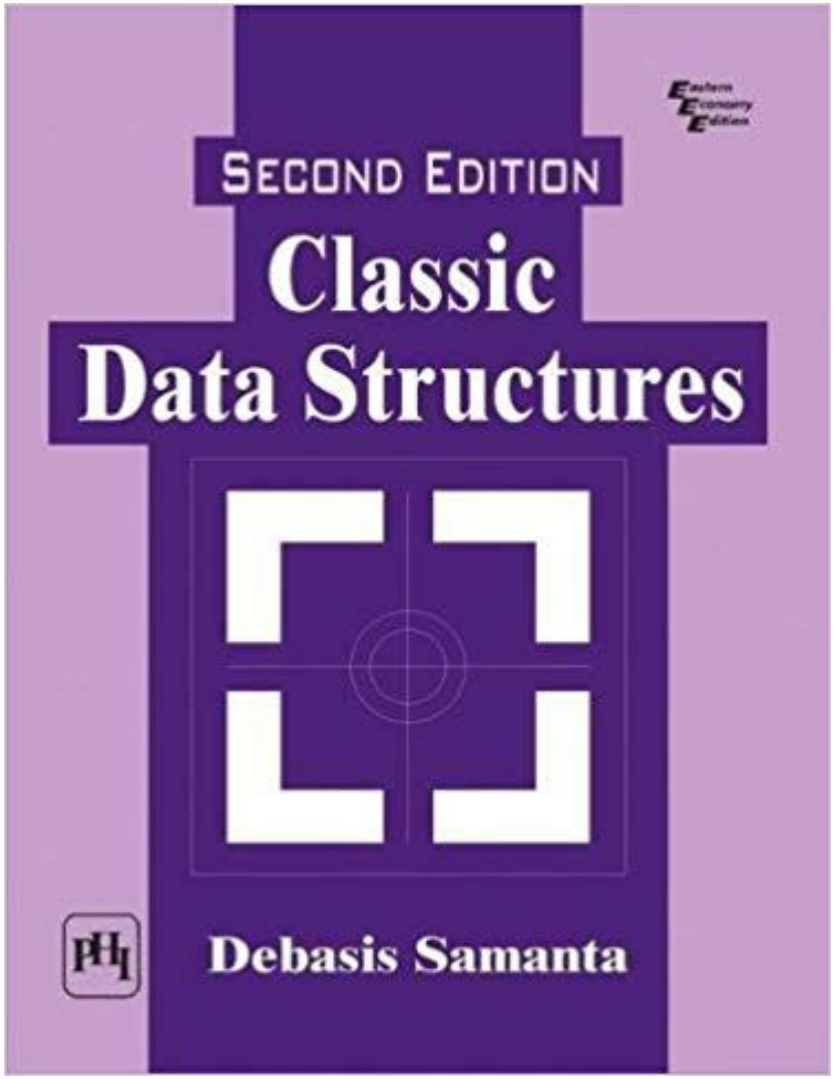
Data Structure References



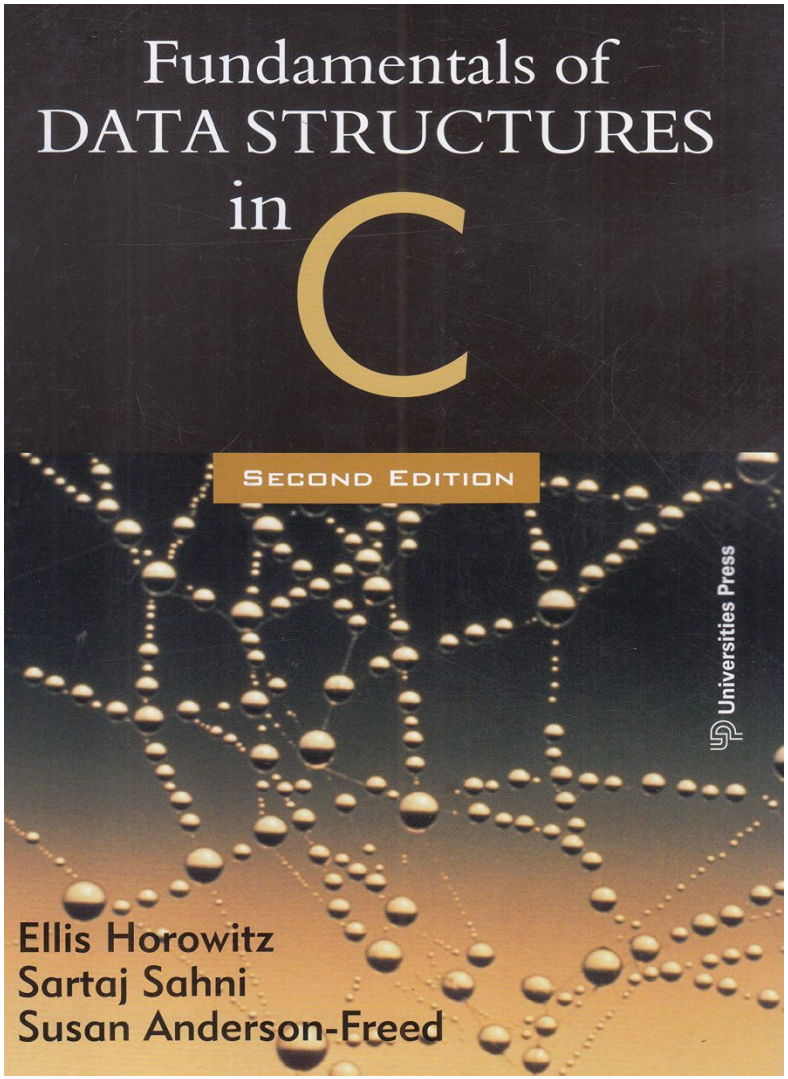


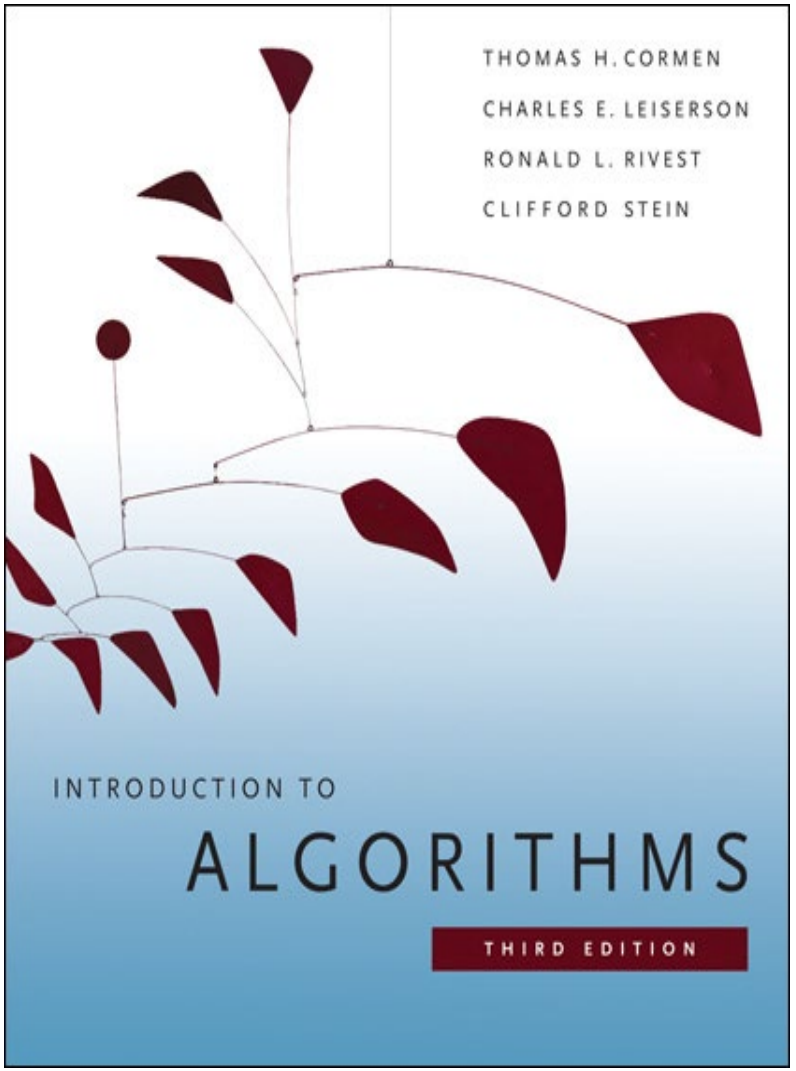
Data Structure References



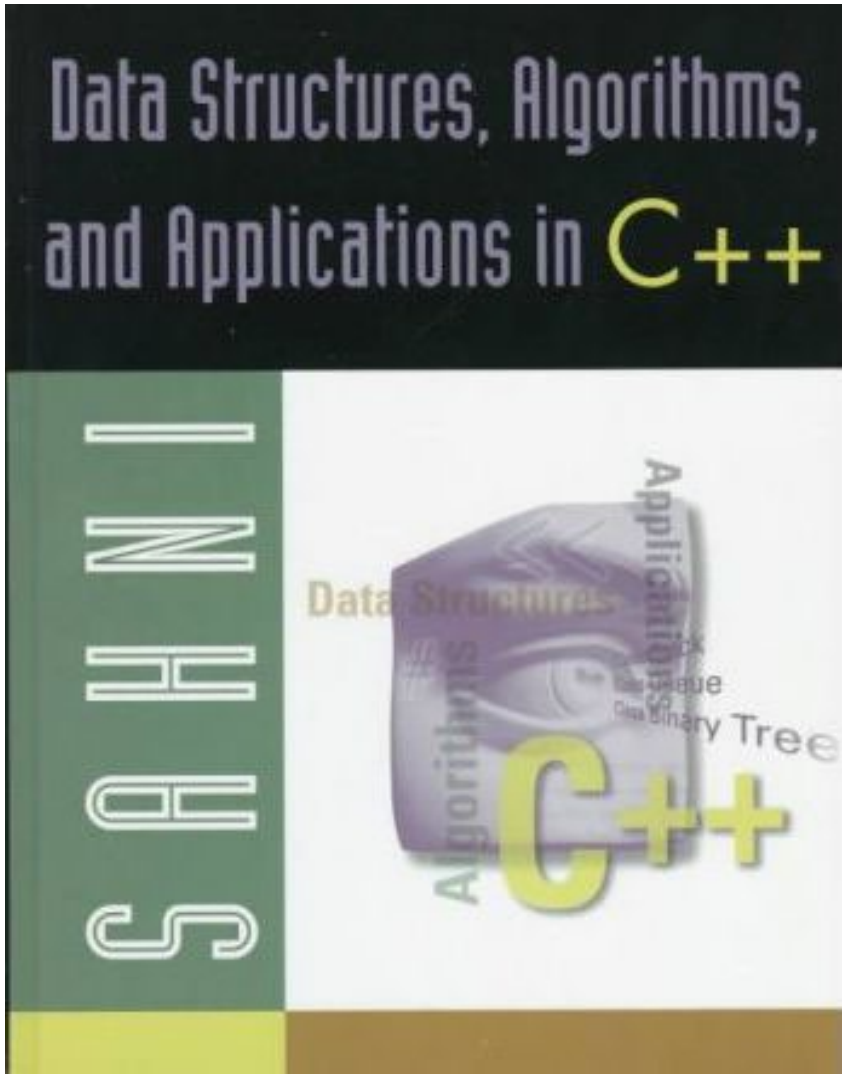


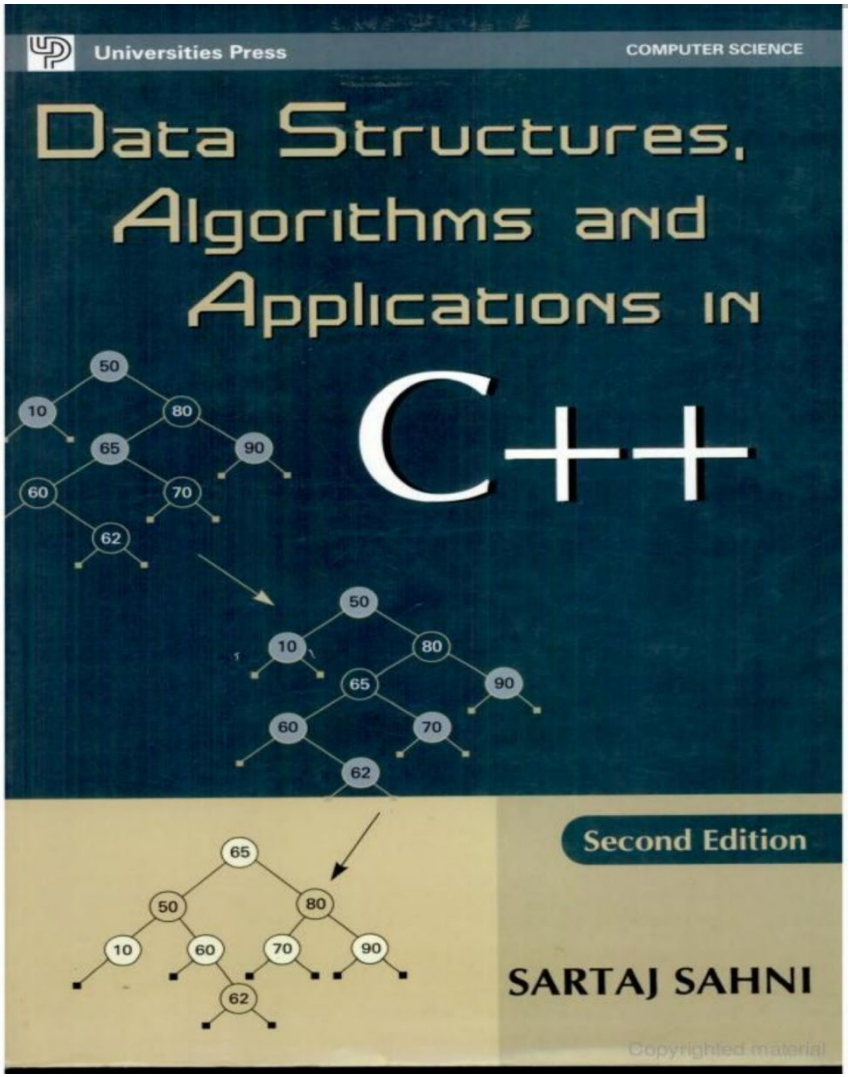
Data Structure References



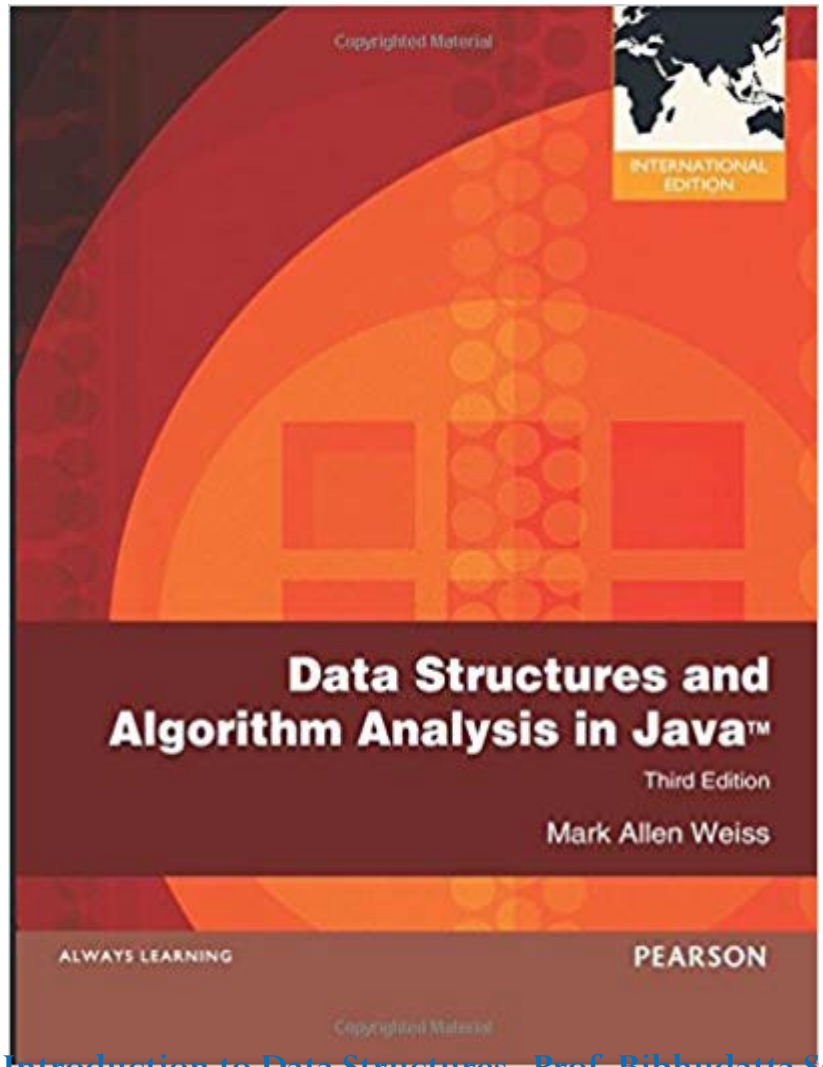


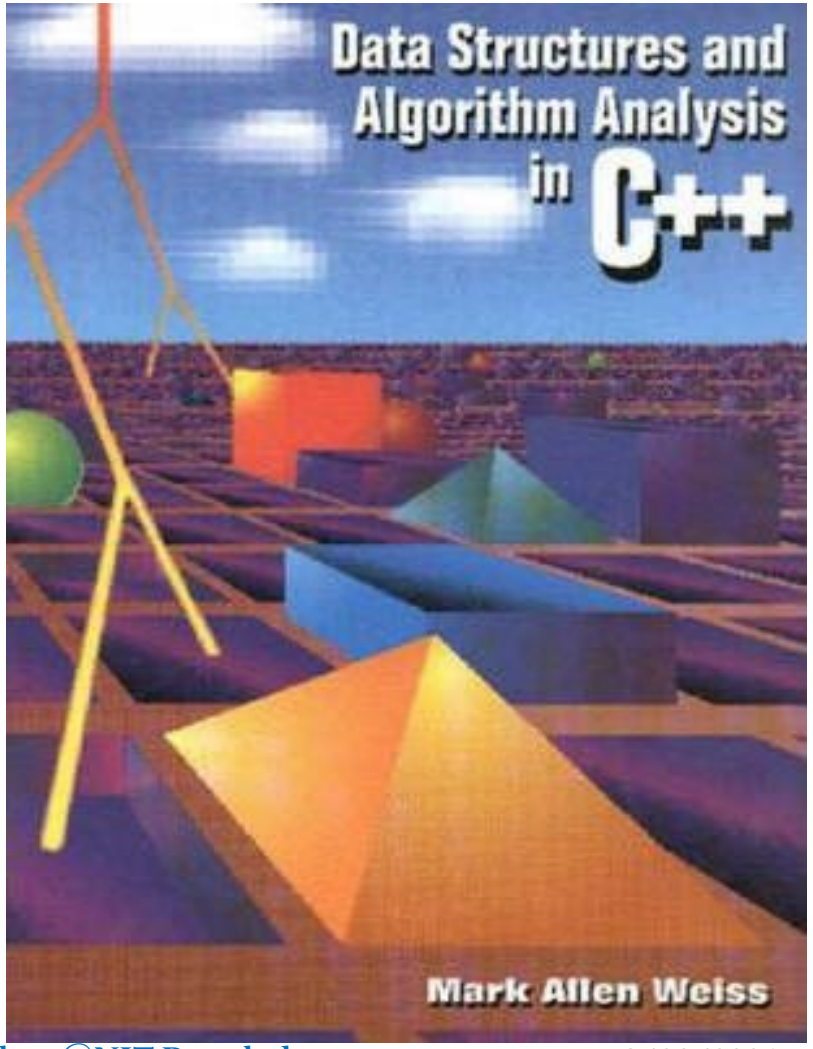
Sartaj Sahni , “Data Structures and Algorithms and Applications in C++



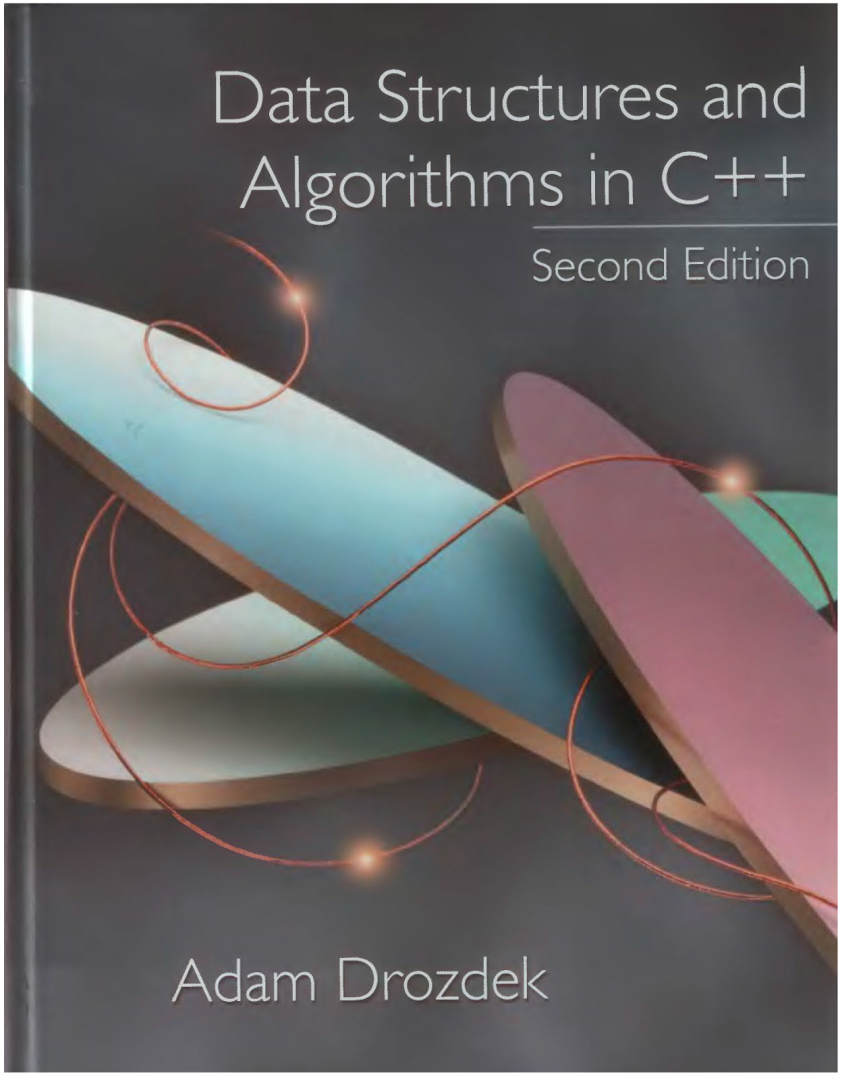


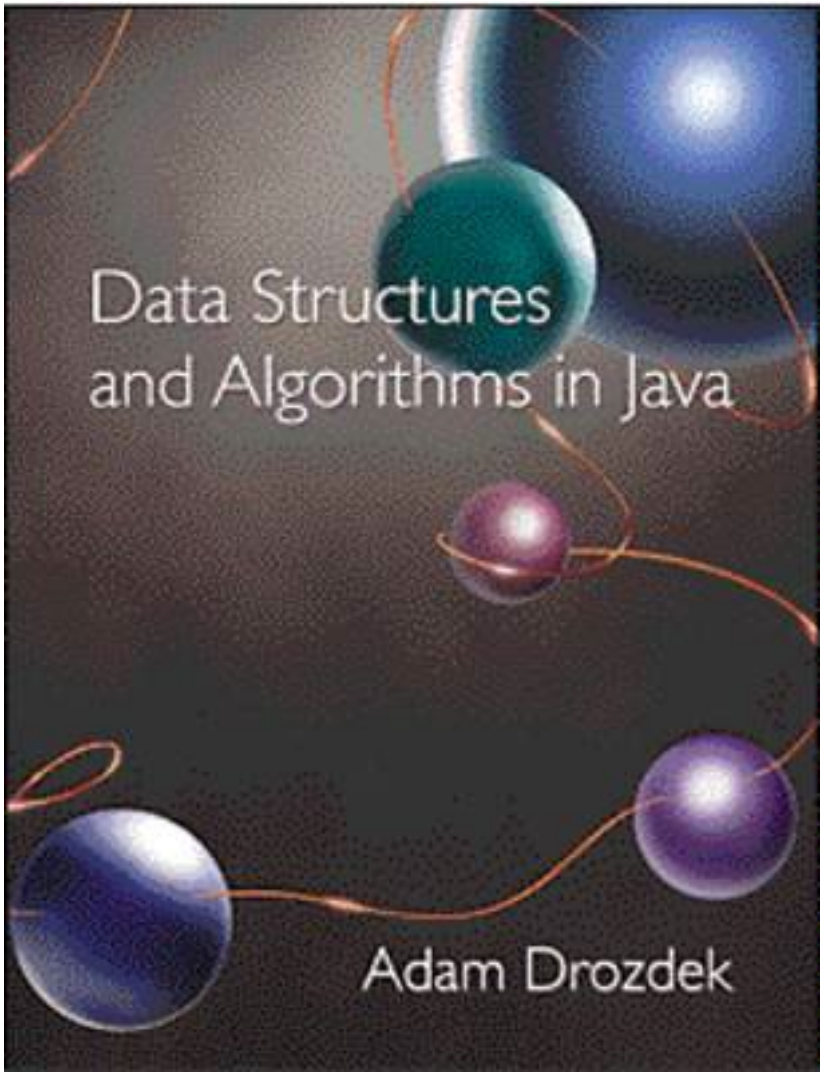
Mark Allen Weiss , Data Structures and Algorithm Analysis in Java / C ++



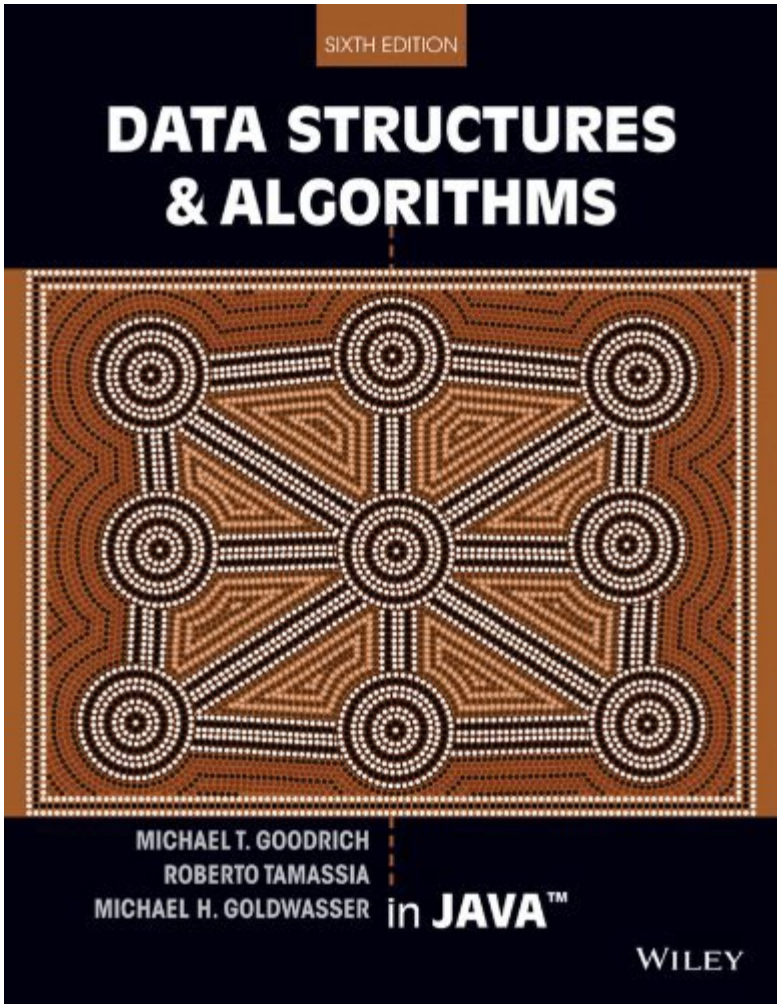


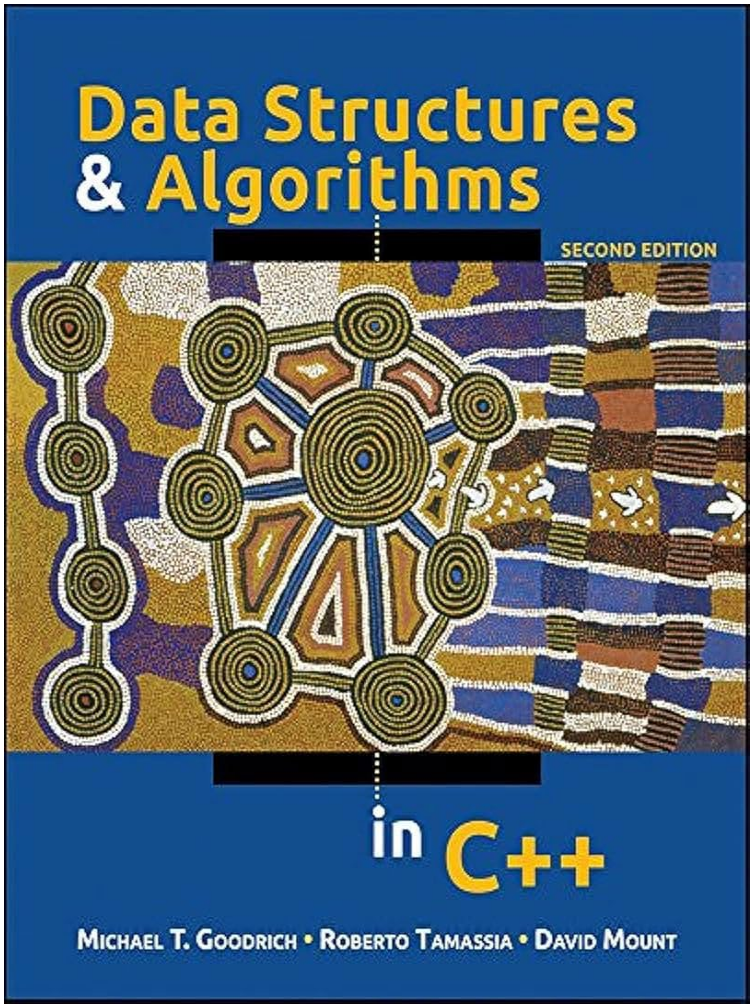
Adam Drozdek , “Data Structures and Algorithms in C++ / Java





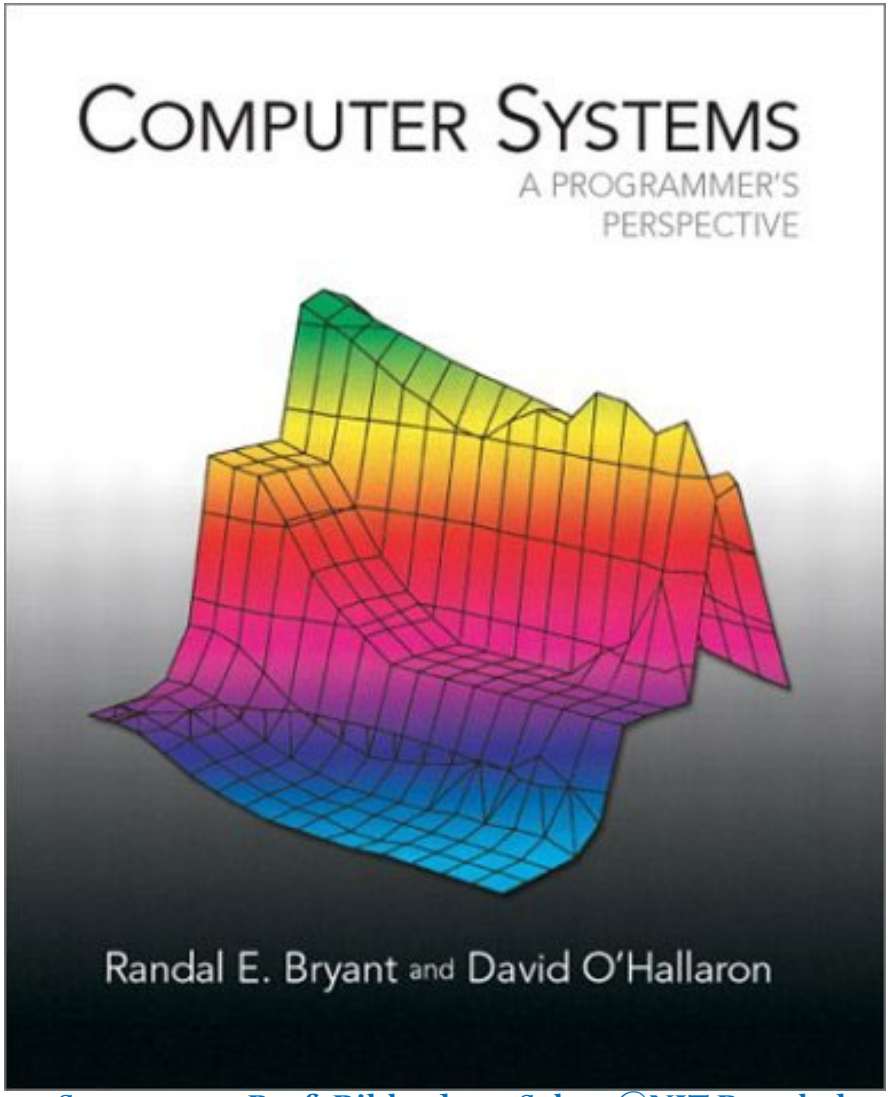
Goodrich, Tamasia, Goldwasser :Data Structures and Algorithm Analysis in Java / C ++



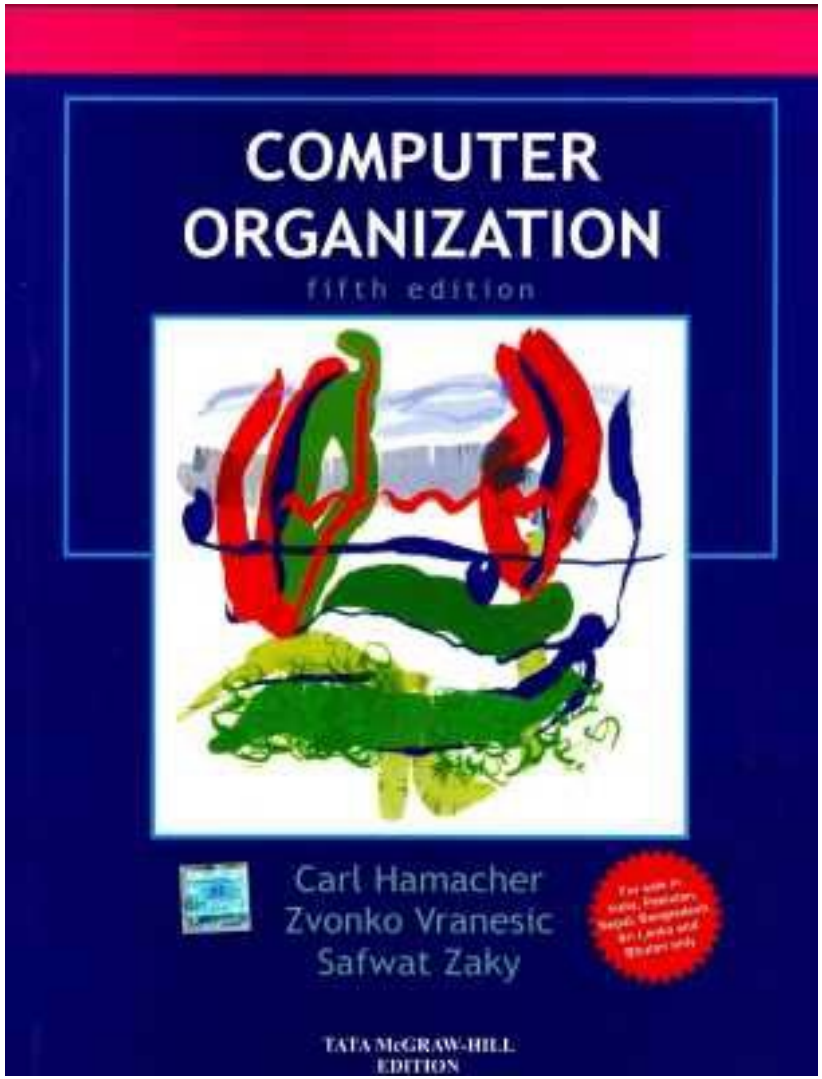


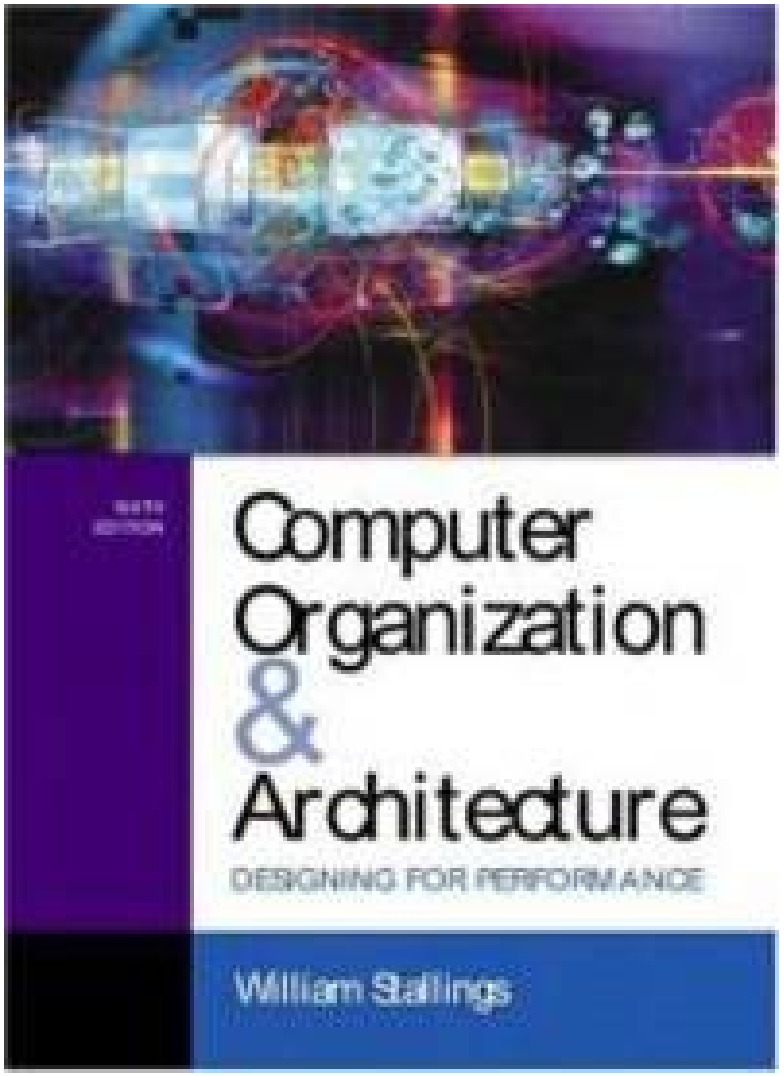
Computer Systems

Computer Systems

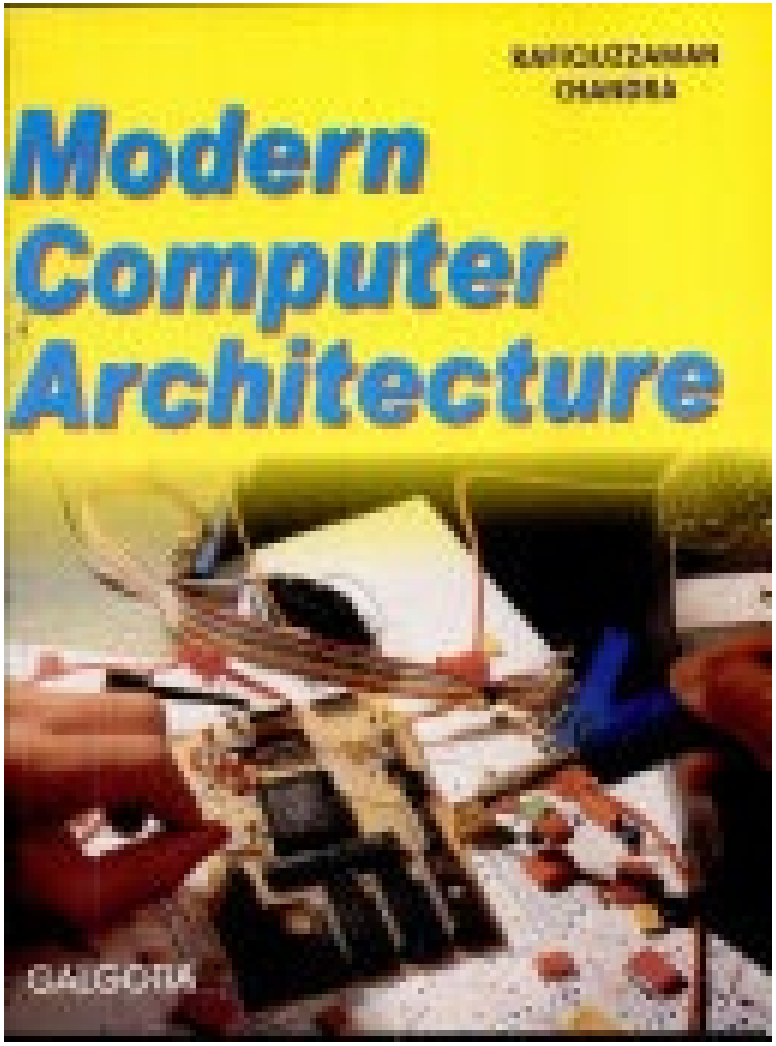


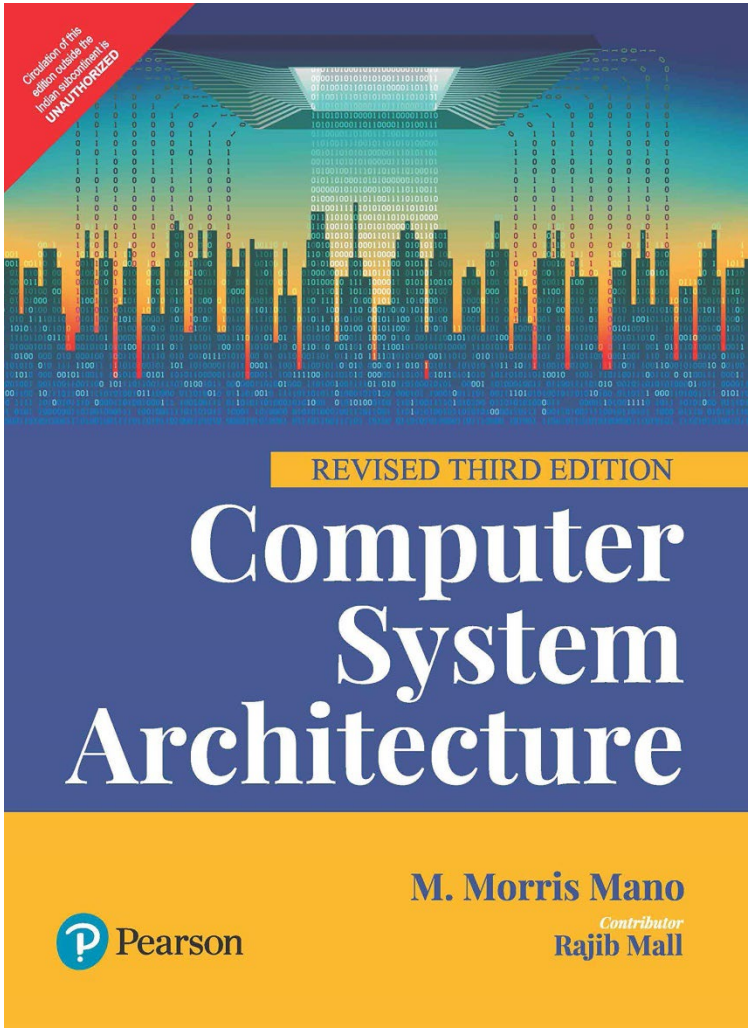
Computer Organization & Architecture





Computer Organization & Architecture





List of DOS Debug commands:

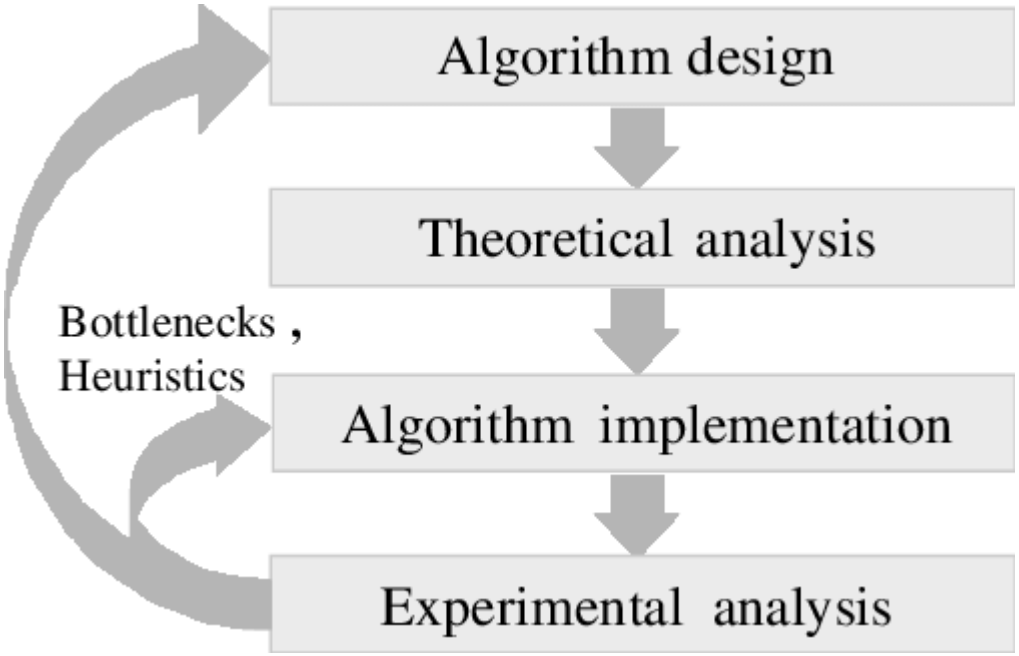
?	Displays a list of debug commands
a	Assembles 8086/8087/8088 mnemonics
c	Compares two portions of memory
d	Displays the contents of a portion of memory
e	Enters data into memory starting at a specified address
f	Fills a range of memory with specified values
g	Runs the executable file that is in memory
h	Performs hexadecimal arithmetic
i	Displays one byte value from a specified port
l	Loads the contents of a file or disk sectors into memory
m	Copies the contents of a block of memory
n	Specifies a file for an l or w command, or specifies the parameters for the file you are testing
o	Sends a single byte value to an output port

List of DOS Debug commands:

p	Executes a loop, a repeated string instruction, a software interrupt, or a subroutine
q	Stops the Debug session
r	Displays or alters the contents of one or more registers
s	Searches a portion of memory for a specified pattern of one or more byte values
t	Executes one instruction and then displays the contents of all registers, the status of all flags, and the decoded form of the instruction that Debug will execute next
u	Disassembles bytes and displays the corresponding source statements
w	Writes the file being tested to a disk
xa	Allocates expanded memory
xd	Deallocates expanded memory
xm	Maps expanded memory pages
xs	Displays the status of expanded memory

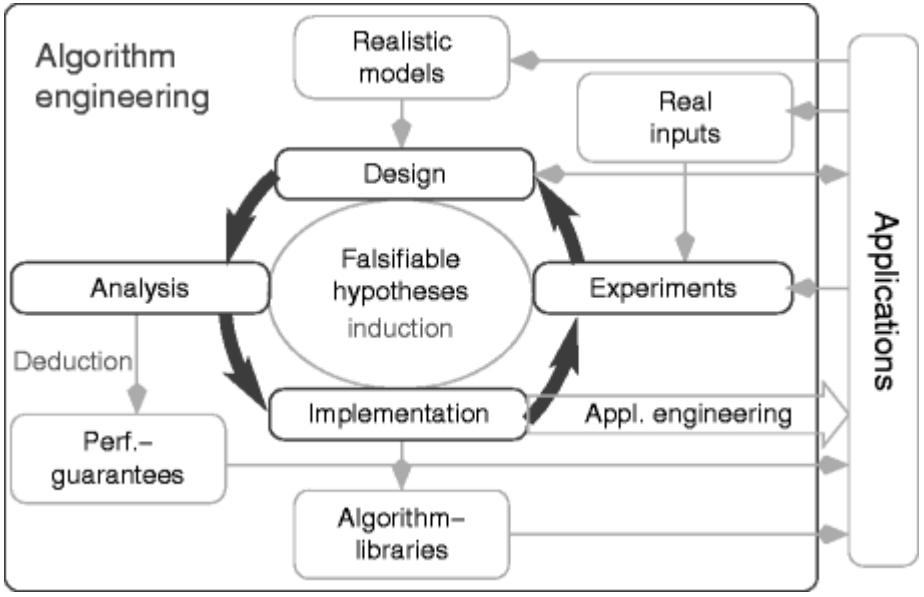
Algorithm engineering

- **Algorithm engineering** focuses on the design, analysis, implementation, optimization, profiling and experimental evaluation of computer algorithms, bridging the gap between **algorithm** theory and practical applications of algorithms in software **engineering**.
- It is a general methodology for **algorithmic** research.



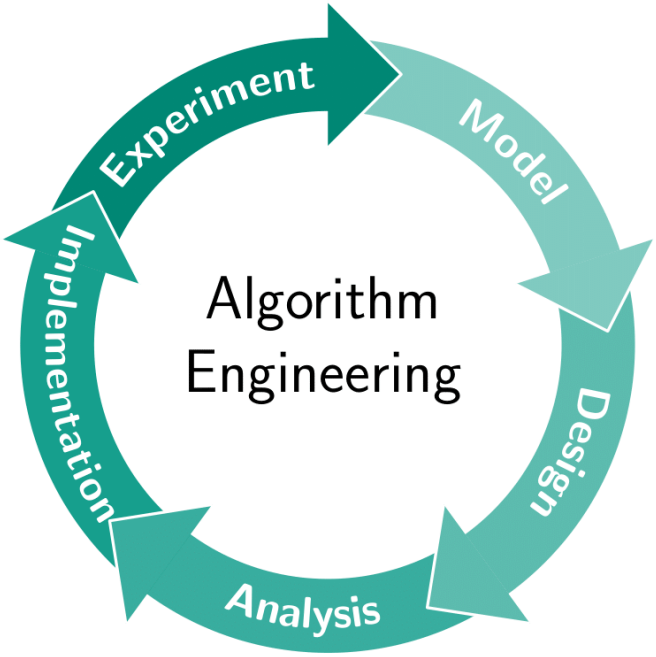
Algorithm engineering

- Algorithmics is the subdiscipline of computer science that studies the systematic development of efficient algorithms.
- Algorithm Engineering (AE) is a methodology for algorithmic research that views design, analysis, implementation, and experimental evaluation of algorithms as a cycle driving algorithmic research.



Algorithm engineering cycle

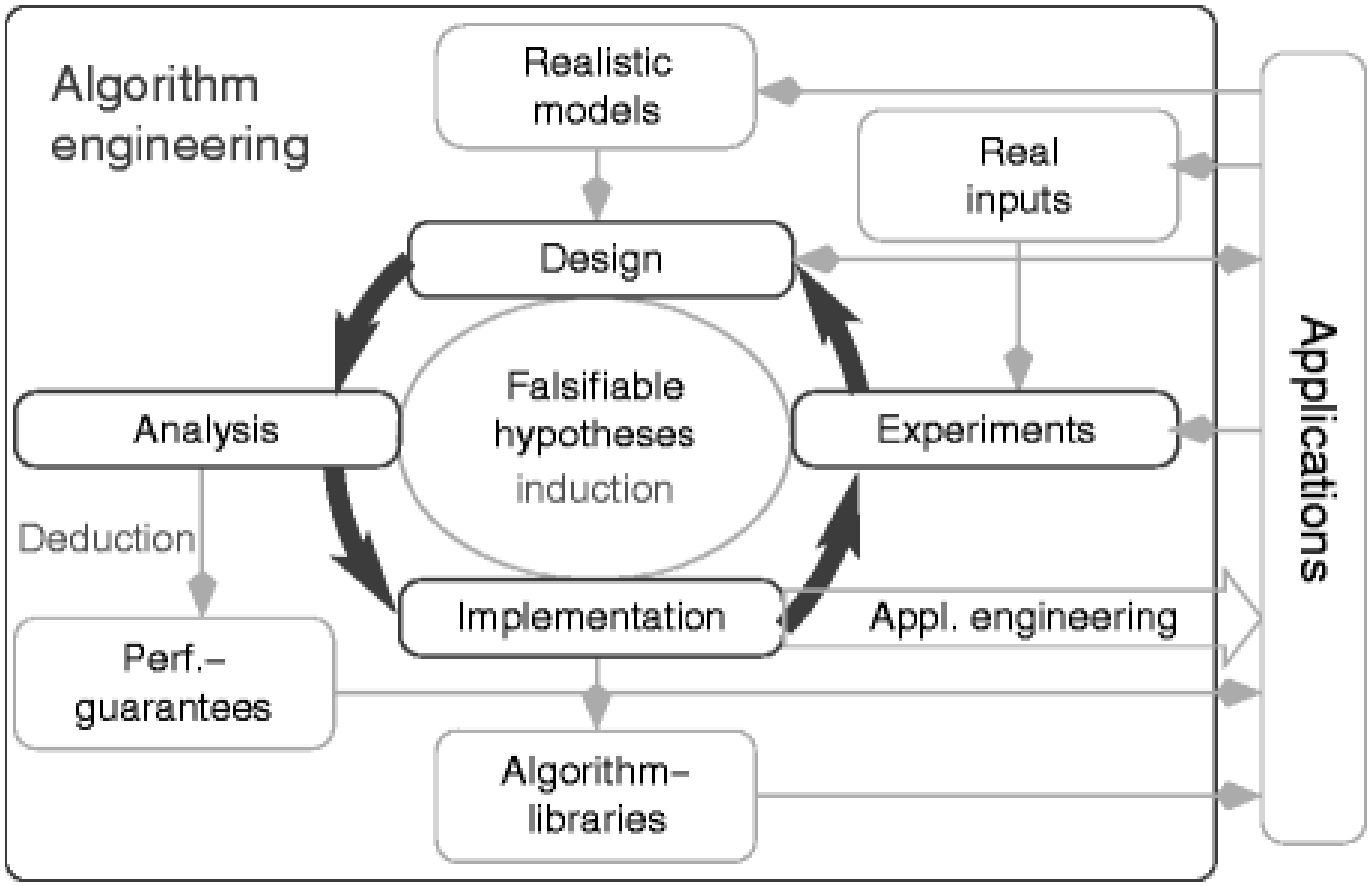
- Algorithm engineering consists of the design, analysis, implementation and experimental evaluation of practicable algorithms.



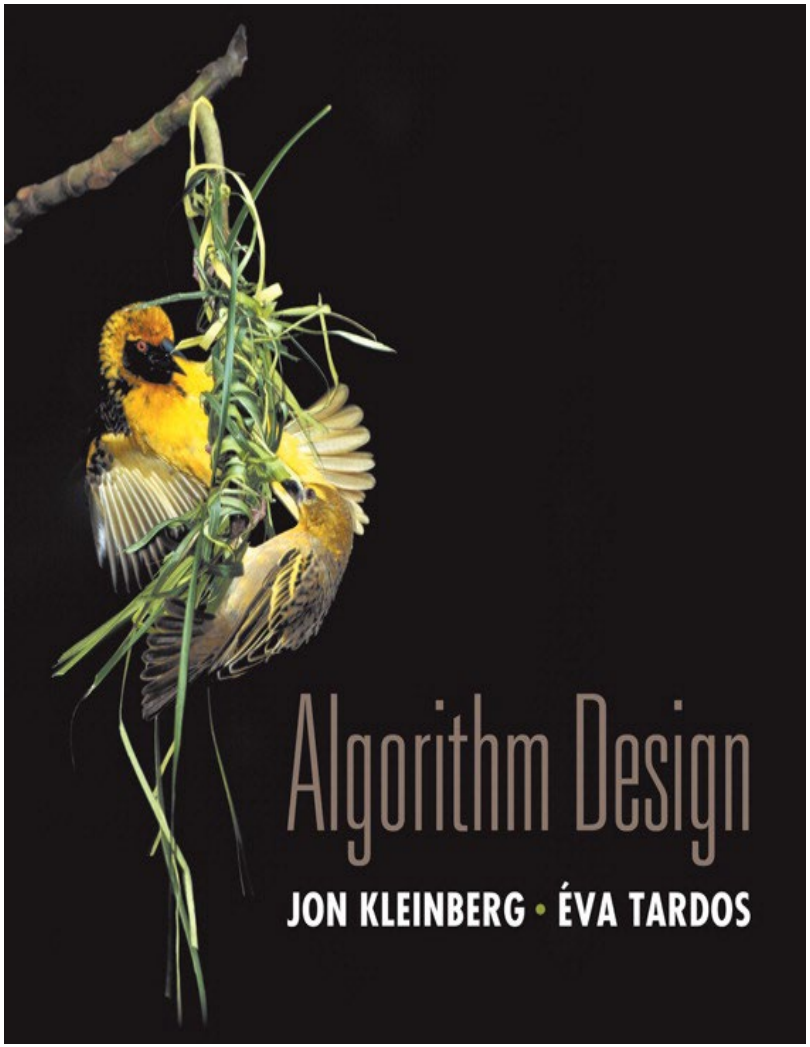
- Research in this direction can be done both in a lab environment and in a real world application. Experience shows that there are significant differences in methodology between these two settings.

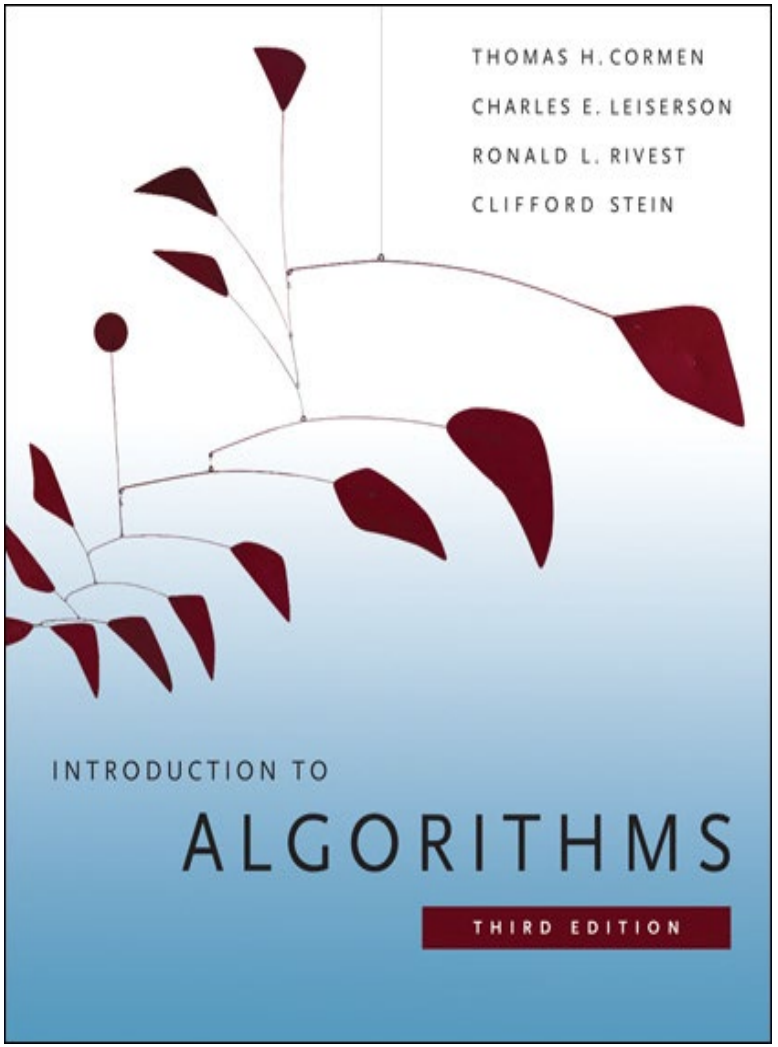
Algorithm engineering

- Algorithm engineering as a cycle of design, analysis, implementation, and experimental evaluation driven by falsifiable hypotheses

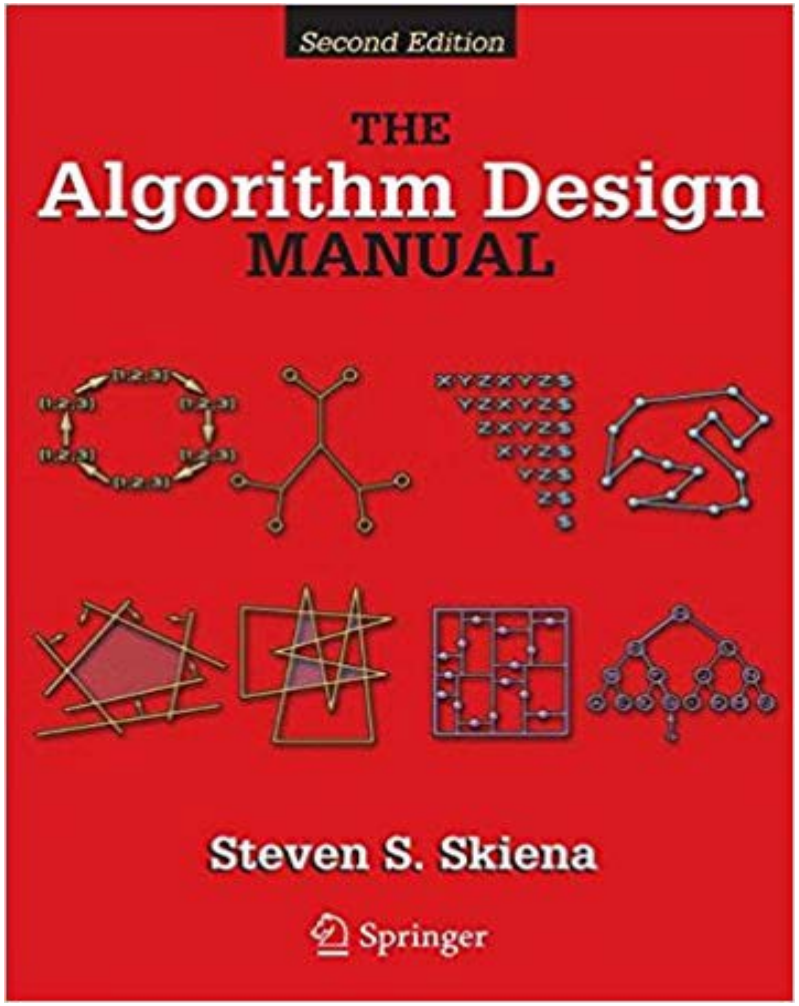


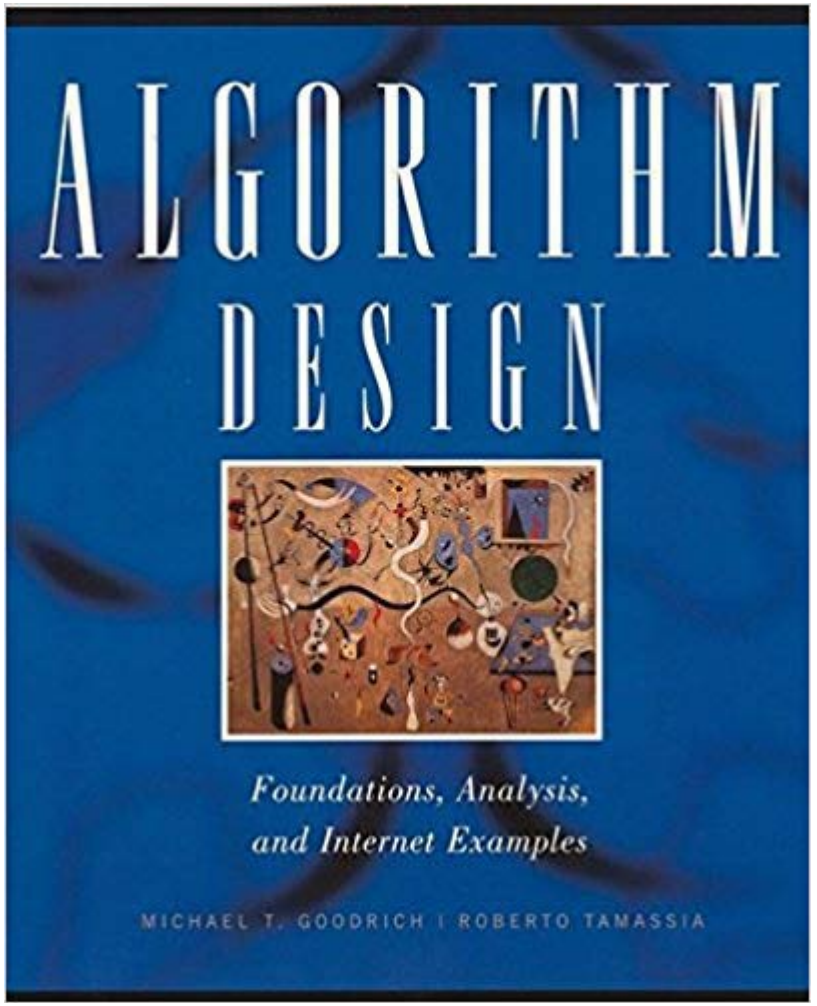
Data Structure References



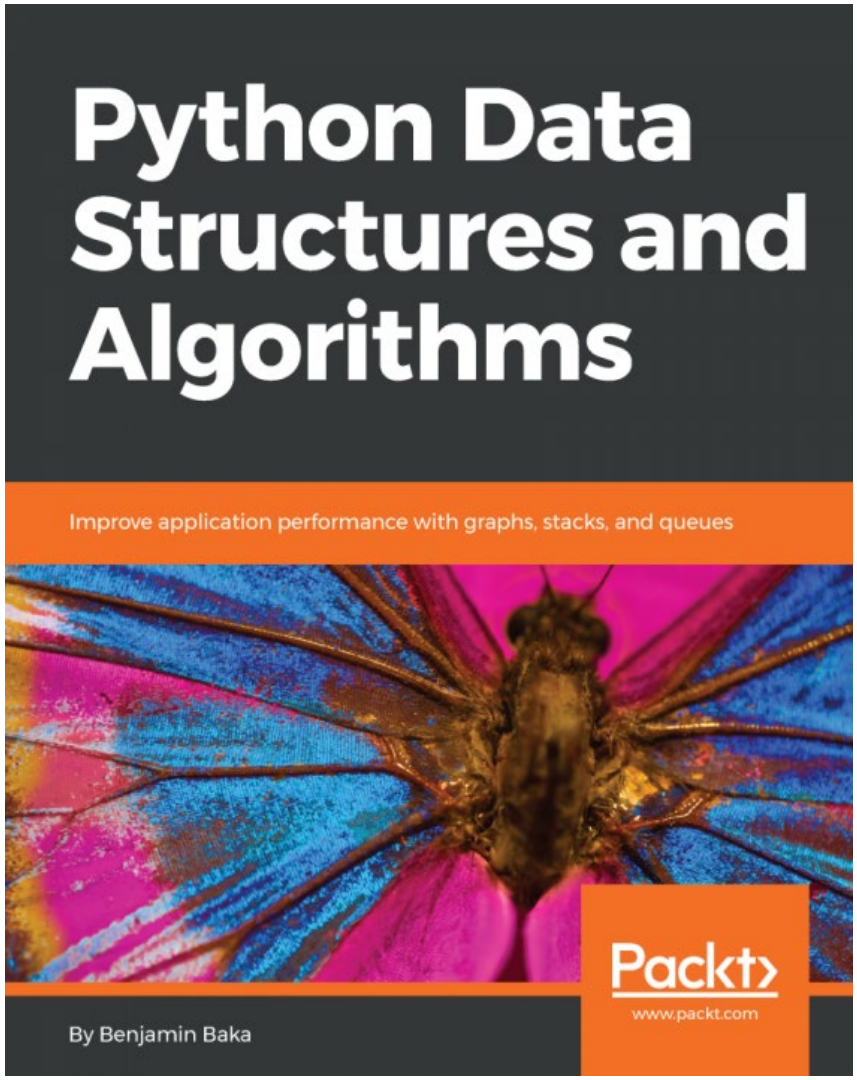


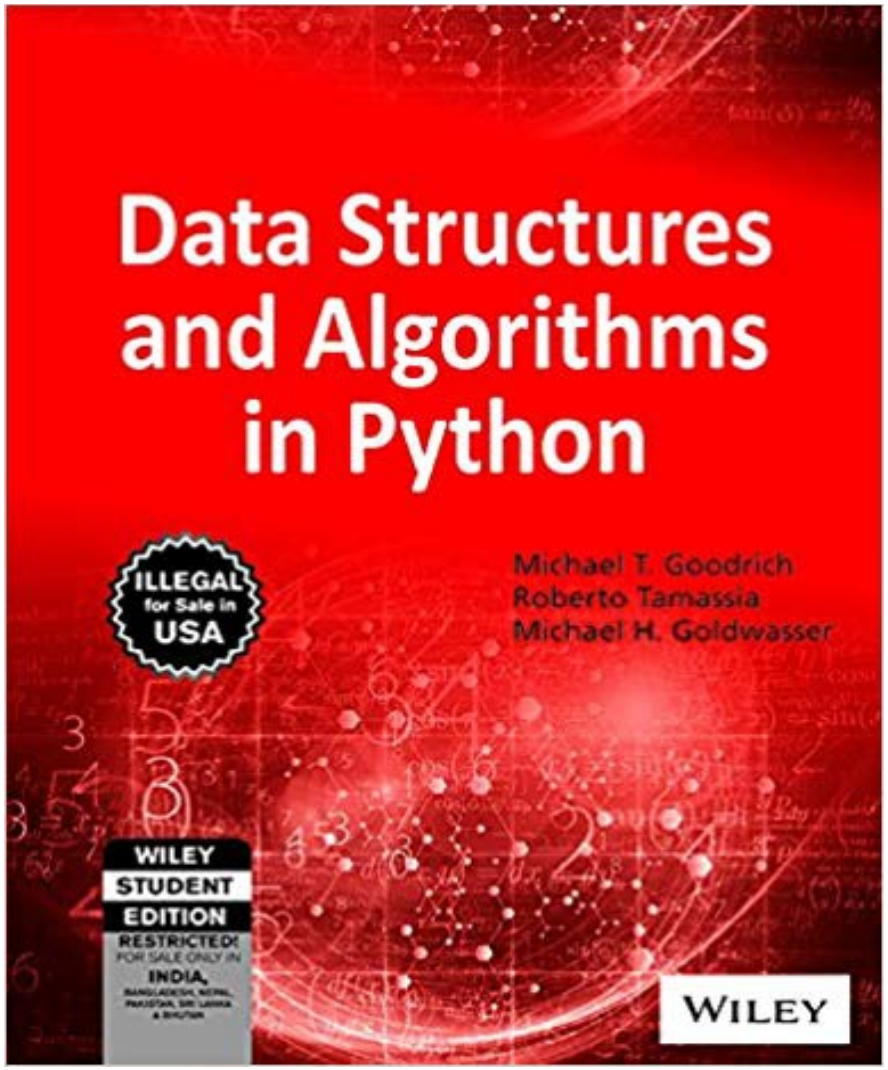
Data Structure References



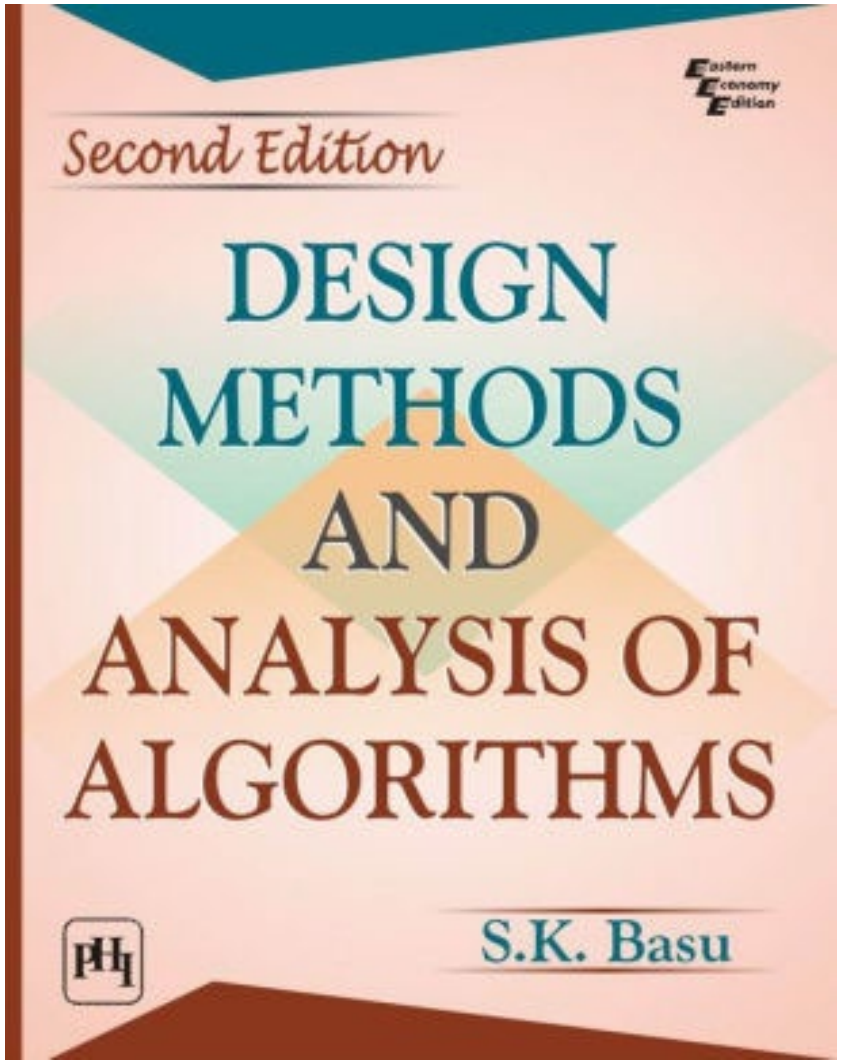


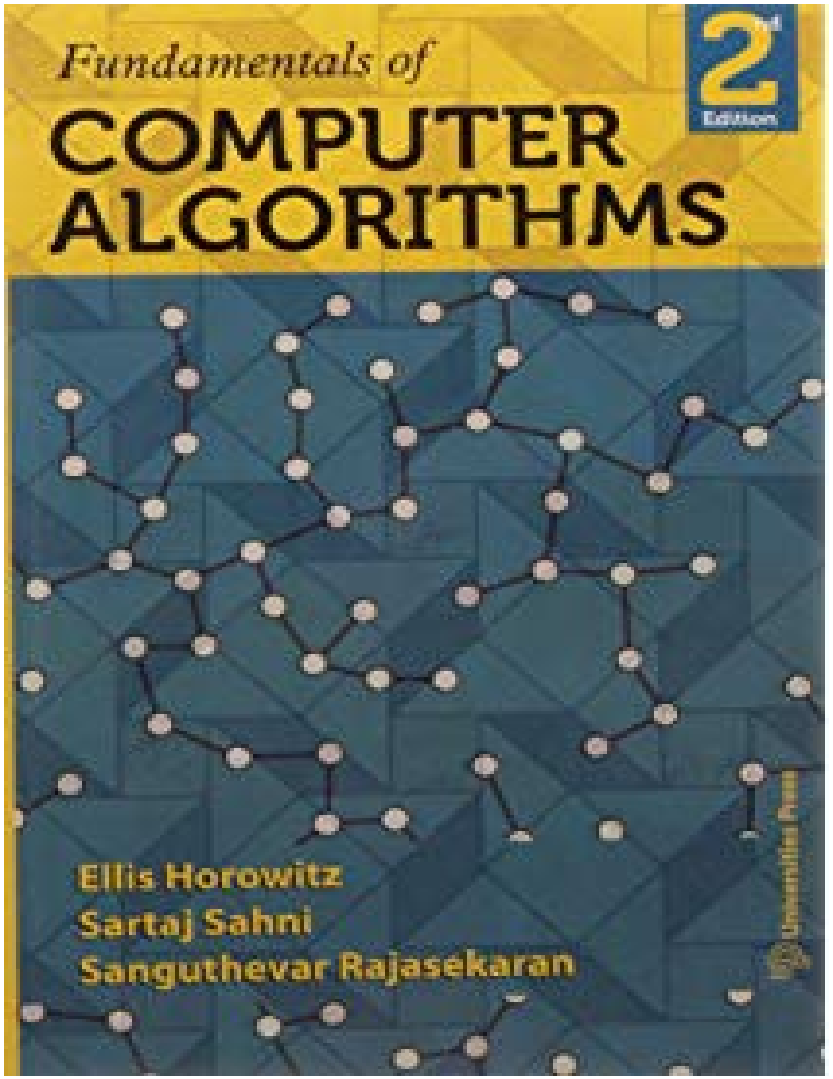
Data Structure References





Data Structure References





Exercises

- Define a data structure.
- Explain why data structures are important in computer science.
- Differentiate between data structures and file structures.
- What are the key properties of a data structure?
- What is an Abstract Data Type (ADT)?

External Memory Data Structures

- Many modern applications store and process datasets much larger than the main memory of even state-of-the-art high-end machines.
- There is a massive and dynamically changing datasets often need to be stored in data structures on external storage devices, and in such cases the Input/Output (or I/O) communication between internal and external memory can become a major performance bottleneck.
- The need to explore the recent advances in the development of worst-case I/O-efficient external memory data structures.

Data Structures