

DATA STRUCTURE AND ALGORITHMS

Jamaica Pingol

Bachelor of Science in Information Technology

2 – 1 N

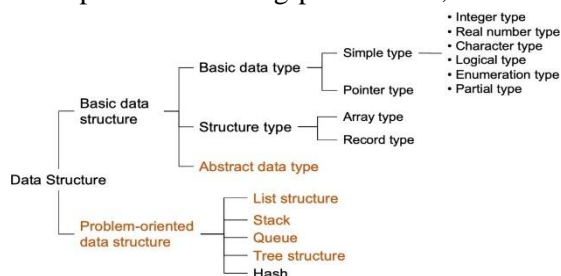
Lesson 1: Introduction to Data Structures



- Data is represented in **memory** (data structure) and how it is represented in **disk** (file structure)
- Data structure is a way of collecting and organizing data in such a way that we can perform operations on these data in an effective and sometimes efficient way.
- Data structure refers to a scheme for organizing data, or in other words a data structure is an arrangement of data in computer's memory in such a way that it could make the data quickly available to the processor for required calculations.

Classification of Data Structures

- Notion of data structures involves:
 1. A set of data elements;
 2. the way data elements are logically related;
 3. and a set of allowable operations on the data elements.
- Data structures are structures programmed to store data, so that various operations can be performed on it easily.
- **Data type** = set of values from which a variable, constant, function, or other expression may take its value. For example, integer takes on whole numbers, real numbers represent fixed-point and floating-point values, character takes on



alphanumeric values, Boolean represents a true or false value, while date/time represents a range of values depicting date and time.

- Data type can be thought of as a set of the "same kind" of data processed by the computer.

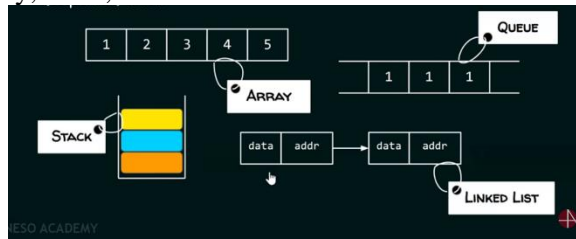
Categories of Data Structures

Primitive Data Structure	Simple Data Structure	Complex Data Structure			
		Compound Data Structure			File Organization
		Linear	Non-Linear		
			Binary	N-ary	
Integer	Array	List	Binary Tree	Graph	Sequential
Char	String	Stack	Binary Search Tree	General Tree	Relative
Real	Record	Queue	Search Tree	M-way Search Tree	Indexed
Boolean			AVL Tree	Heap	Sequential
				B-Tree	Multi-Key

- **Types of data structure:**

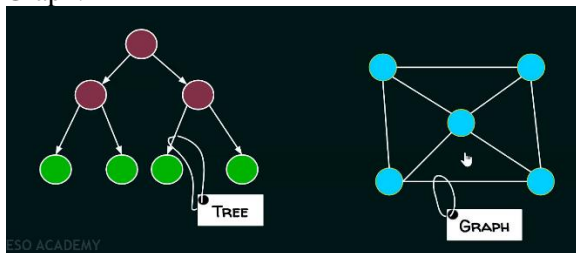
Linear Data Structures = a data structure is linear when all the elements are arranged in a linear (sequential order). When each element of the data structure has only one predecessor and one successor, then we can say that -- that data structure is a linear data structure.

Example: Queue, Array, stack, and linked list.



Non Linear Data structures = when all the elements are not arranged in a linear (sequential) order. There is no linear arrangement of the elements.

Examples: Trees and Graph.



Basic Data Type or Primitive Data Structures represent a set of individual data and is frequently used to create a program. Sometimes called **atomic data structure** as they represent a form where data can no longer be divided or have no parts. This group can be further divided into **Simple Type** and **Pointer Type**.

- **Simple type** is the most basic data type which is usually declared according to the syntax rule of a programming language. Most common data types fall under this group.
- **Integer type** – represents integers or whole numbers where the maximum or minimum value is the unit of data that a computer can process at one time and is determined by the length of one word.
- **Real number type** – represent fixed-point and floating-point numbers
- **Character type** – comprised of alphabets, numerals, and symbols as characters. A character code is expressed as a binary number inside a computer.
- **Logical type** – sometimes referred to as Boolean type where the values are used in performing logical operations such as AND, OR, and NOT.
- **Enumeration type** – a data type that enumerates all possible values of variables.
- **Partial type** – used to specify an original-value subset by constraining existing data types, that is identifying upper and lower limits of a variable.

- **Pointer type** – are addresses that are allocated in a main memory unit. Pointer types are used to refer to variables, file records, or functions.

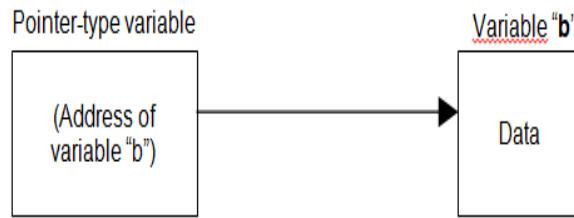


Figure 2 Pointer type

Structure Type or Simple Data Structure is a data structure that contains a basic data type or any of the defined data types as its elements. Arrays, strings, and records are examples of Structure Type.

- **Array type or array** is simply a finite set of elements having the same type referenced under a common name.
- If this type is a **character type**, we refer to it as a *string* or a collection of character elements.
- **Record type** = set of elements but this time of different data types referenced under a common name.

Abstract Data Type is a part of Basic Data Structure but represents those under Problem-oriented Data Structure. Abstract Data Type or ADT is almost always synonymous to Data Structures but represents more of a logical description (abstract) rather than actual implementation. ADT is basically a mathematical model where a set of data values and its associated operations are precisely specified independent of any particular implementation.

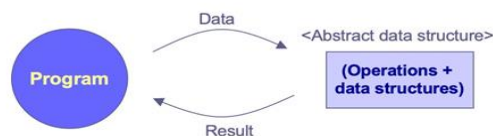


Figure 3 Abstract Data Type representation |

Logical and Physical Forms

Every data items have both a logical and physical form.

1. Logical form: definition of the data item within an ADT (e.g. integers in mathematical sense: $+$, $-$, $*$, $/$ (operations) $7/2 = 3.5$ $7 \setminus 2 = 3$)
2. Physical form: implementation of the data item (e.g. 16 or 32 bit integers)

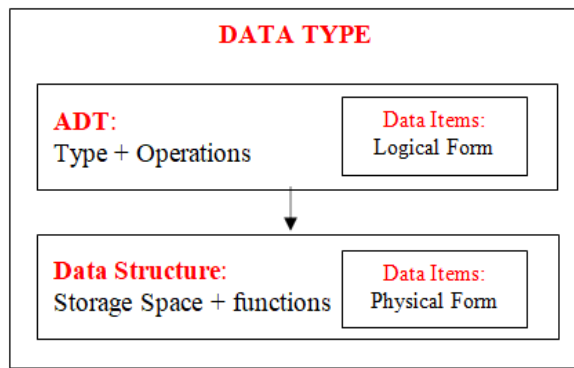


Figure 4 Logical and Physical Form of a Data Type

Classification of Data Structures based on Characteristics

Characteristic	Description
Linear	In linear data structures, the data items are arranged in a linear sequence. Example: Array
Non-Linear	For non-linear data structures, the data items are not in sequence. Example: Tree, Graph
Homogeneous	Homogeneous data structures represent a structure whose elements are of the same type. Example: Array
Non-Homogeneous	In Non-homogeneous data structure, the elements may or may not be of the same type. Example: Structures
Static	Static data structures are those whose sizes and structures associated memory locations are fixed, at compile time. Example: Array
Dynamic	Dynamic structures are those which expands or shrinks depending upon the program need and its execution. Also, their associated memory locations changes. Example: Linked List created using pointers.

Types of data structures - Data structure types are determined by what types of operations are required or what kind of algorithms are going to be applied. These types include:

1. **Arrays** - An array stores a collection of items at **adjoining memory location**. Items that are the same type **get stored together** so that the position of each element can be calculated or retrieved easily. Arrays can be fixed or flexible in length.
2. **Stacks** - A stack stores a collection of items in the **linear order** that **operations are applied**. This order could be last in first out (LIFO) or first in first out (FIFO).
3. **Queues** - A queue stores a collection of items **similar to a stack**; however, the operation order can **ONLY** be **first in first out**.
4. **Linked lists** - A linked list stores a collection of items in a **linear order**. Each element or node in a linked list contains a data item as well as a reference, or link, to the next item in the list.
5. **Trees** - A tree stores a collection of items in an abstract, **hierarchical way**. Each node is linked to other nodes and can have multiple sub-values, also known as children.
6. **Graphs** - A graph stores a collection of items in a **non-linear fashion**. Graphs are made up of a finite set of nodes, also known as **vertices**, and lines that connect them, also known as **edges**. These are useful for representing real-life systems such as computer networks.
7. **Hash tables** - A hash table, or hash map, stores a collection of items in an **associative array** that plots keys to values. A hash table uses a hash function to convert an index into an array of buckets that contain the desired data item.

DATA STRUCTURE AND ALGORITHMS

Jamaica Pingol

Bachelor of Science in Information Technology

2 – 1 N

Lesson 2: Algorithm Concept



- **Algorithm Analysis or Analysis of Algorithm** is the theoretical study of computer program's performance and resource usage. In this unit, we present the idea of algorithms and how performance and resource usage affect the effectiveness of an algorithm.
- An **algorithm** is a finite set of instructions or logic, written in order, to accomplish a certain predefined task.
- Algorithm is not the complete code or program, it is just the core logic (solution) of a problem, which can be expressed either as an informal high-level description as pseudocode or using a flowchart.
- **Big O notation** is a mathematical notation that describes the limiting behavior of a function when the argument tends towards a particular value or infinity. It is a member of a family of notations invented by Paul Bachmann, Edmund Landau, and others, collectively called Bachmann-Landau notation or asymptotic notation.

Every algorithm must satisfy the following criteria:

1. **Input** - An algorithm has zero or more input quantities which are externally supplied taken from a set of objects called the **domain of the algorithm**.
2. **Output** - It has one or more output quantities which generate a set called the **range of the algorithm**.
3. **Definiteness** - Each instruction must be clear and unambiguous, meaning each step of an algorithm must be **precisely defined**.
4. **Finiteness** - If we traced out the instructions of an algorithm, then for **all cases** the algorithm will **terminate** after a finite number of steps.
5. **Correctness or Effectiveness** - Every instruction must be sufficiently basic that it can in principle, be carried out by a person by manual means and must generate a **correct output**.

Time complexity of Algorithms

In order to analyze an algorithm, we need to determine its (algorithm) efficiency, which measures the amount of resources consumed in solving a problem of size in terms of time and space. A way of doing this is by **benchmarking** which is implementing the algorithm, running using some specific input and measuring time taken.

The best option is to perform **asymptotic analysis** where we evaluate the performance of an algorithm in terms of the input size (problem size) in relation with the processing time (or space) required to solve the problem. In here we look for three cases to examine:

- **Best Case** – if the algorithm is executed, the *fewest number of instructions* are executed.
- **Average Case** – executing the algorithm produces *path lengths* that will on *average be the same*.
- **Worst Case** – executing the algorithm produces *path lengths* that are always a *maximum*.

For any defined problem, there can be N number of solutions. This is true in general.

One solution to this problem can be running a loop for n times, starting with the number n and adding n to it every time.

```
/*
we have to calculate the square of n
*/
for i=1 to n
do
n = n + n
// when the loop ends n will hold its square
return n
```

Or we can simply use a mathematical operator* (multiplication) to find the square.

```
/*
we have to calculate the square of n
*/
return n*n
```

- Time complexity of an algorithm signifies the total time required by the program to run till its completion.
- The time complexity of algorithms is most commonly expressed using the big O notation which is an asymptotic notation to represent the time complexity.

Types of Notation for Time Complexity

1. **Big Oh** denotes "fewer than or the same as" <expression> iterations.
 2. **Big Omega** "more than or the same as" <expression> iterations.
 3. **Big Theta** denotes "the same as" <expression> iterations.
 4. **Little Oh** denotes "fewer than" <expression> iterations.
 5. **Little Omega** denotes "more than" <expression> iterations.
- **O(expression)** is the set of functions that **grow slower** than or at the same rate as expression. It indicates the **maximum time** required by an algorithm for all input values. It represents the **worst case** of an algorithm's time complexity.
 - **Omega(expression)** is the set of functions that **grow faster** than or at the same rate as expression. It indicates the **minimum time** required by an algorithm for all input values. It represents the **best case** of an algorithm's time complexity.
 - **Theta(expression)** consist of all the functions that lie in **both O(expression)** and **Omega(expression)**. It indicates the **average bound** of an algorithm and it represents the **average case** of an algorithm's time complexity.

Running Time Calculations

- **Complexity Theory** – also called as computational complexity. It is measure of the amount of computing resources (time and space) that a particular algorithm consumes when it runs.
- **Algorithm analysis** is an important part of computational complexity theory, which provides theoretical estimation for the required resources of an algorithm to solve a **computational problem**.
- **Complexity Theory** or **computational complexity** is the measure of the **amount of computing resources** (time and space) that a particular algorithm consumes when it runs.
- **Time Complexity** is the amount of time the *algorithm is completed*. It is commonly expressed using the **big O notation**.

How to Calculate the Time Complexity

1. Break your algorithm/function into individual operations
2. Calculate the Big O of each operation
3. Add up the Big O of each operation together
4. Remove the constants
5. Find the highest order term

EXAMPLE 1 (Time Complexity): Array with 5 elements

```
Example 1:  
arr_X 1 20 5 4 9  
sum=0;  
for(i=0; i<n; i++)  
{  
    sum=sum+arr_X[i];  
}
```

1. Break your algorithm/function into individual operations

- Take note that each line is an individual operations

```
Example 1:  
arr_X 1 20 5 4 9  
sum=0; line 1  
for(i=0; i<n; i++) 2  
{  
    sum=sum+arr_X[i]; 3  
}
```

- **1st line:** Answer it 1 for time complexity. Kasi minsan lang po siya maexecute pag narun siya sa program

sum=0; → **1**

- **2nd line:** Dahil for loop ito, may mga part siya like the condition and the increment, paulit-ulit siyang pupuntahan. We have to break this into 3 parts:

for(i=0; i<n; i++)

- **i=0 initialization** -----> **1 (since minsan lang siya mag initialize)**
- **i<n; i++ condition and increment** (ilang beses nga ba natin babalikan ito sa program)

2. Calculate the Big O of each operation

Example 1:

arr_X

1	20	5	4	9
---	----	---	---	---

sum=0; -----> 1

for(i=0; i<n; i++) -----> $1 + (n+1) + n$

{
 sum=sum+arr_X[i];
}

i=0

i=1

i=2

i=3

i=4

i=5

Example 1:

arr_X

1	20	5	4	9
---	----	---	---	---

sum=0; -----> 1

for(i=0; i<n; i++) -----> $1 + n + 1 + n$

{
 sum=sum+arr_X[i]; -----> n
}

- **3rd line:** n (since nasa loob siya ng loop, based siya sa n time)

3. Add up the Big O of each operation together

Example 1:

arr_X

1	20	5	4	9
---	----	---	---	---

sum=0; -----> 1

for(i=0; i<n; i++) -----> $1 + n + 1 + n$

{
 sum=sum+arr_X[i]; -----> $\frac{n}{\text{no. of elements in an array}}$
}

(Add the n, it will become 3n)

Example 1:

arr_X

1	20	5	4	9
---	----	---	---	---

sum=0; -----> 1

for(i=0; i<n; i++) -----> $1 + n + 1 + n$

{
 sum=sum+arr_X[i]; -----> $\frac{n}{\text{no. of elements in an array}}$
}

3n+3

ADD THEM

(Add the constants = 3)

4. Remove the constants

Example 1:

```
arr_X [ 1 20 5 4 9 ]
sum=0; → 1
for(i=0; i<n; i++) → 1+n+1+n
{
    sum=sum+arr_X[i]; → n
}
```

~~$n+3$~~

5. Find the highest order term

Since ang natira nalang is si n , wala na siyang pagkukumpara, so we can say the time complexity of this example is $O(n)$ read as “Big O of n ”.

EXAMPLE 1 (Space Complexity): Array with 5 elements

Compute the Space Complexity- These are variables used in algorithm

*yung mga ginamit sa code and ilan siya beses na ginamit

```
arr_X [ 1 20 5 4 9 ]
sum=0;
for(i=0; i<n; i++)
{
    sum=sum+arr_X[i];
}
```

Arr_X = n
sum = 1
i = 1
n = 1
 ~~$n+3$~~ $O(n)$

Operation is based on the length of data to be executed

We can say the space- complexity of this example is $O(n)$ read as “Big O of n ”.

*re-watch video on youtube

https://www.youtube.com/watch?v=hZNDJHK9uAY&ab_channel=ITSInfoTechSkills

DATA STRUCTURE AND ALGORITHMS

Jamaica Pingol

Bachelor of Science in Information Technology

2 – 1 N






Lesson 3: Algorithm, Pseudo Codes, Flowchart, Array



Algorithm = finite sequence of instructions, typically used to solve a class of specific problems or to perform a **computation**. It is used as specifications for performing calculations and data processing. It is usually in terms of **Central Processing Unit** time and memory requirements.

Pseudo Codes are textual presentations of a flowchart. It is close to a natural language and the control structures impose logic. It may become part of the **program documentation**. It could also be **translated** into a program.

Flowchart is a **visual representation** of the sequence of steps and decisions needed to perform a process. Each step in a sequence is noted with a diagram shape. Steps are linked by connecting lines and directional arrows.

Basic Symbols		
Symbol	Name	Function
	Start/end	An oval represents a start or end point
	Arrows	A line is a connector that shows relationships between the representative shapes
	Input/Output	A parallelogram represents input or output
	Process	A rectangle represents a process
	Decision	A diamond indicates a decision

Array is a **container** which can hold a **fixed** number of items and these items should be of the **same type**. Most of the data structures make use of arrays to implement their algorithms. Here are the important terms for arrays:

- **Element** – Each **item** stored in an array is called an element.
- **Index** – Each **location** of an element in an array has a numerical index, which is used to identify the element.

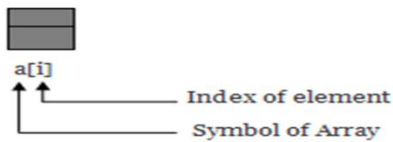
Basic Operations

Following are the basic operations supported by an array.

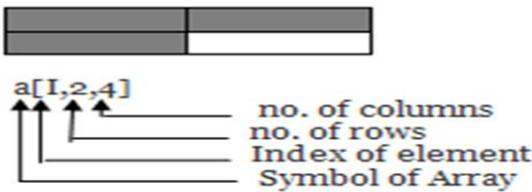
- **Traverse** – print all the array elements one by one.
- **Insertion** – Adds an element at the given index.
- **Deletion** – Deletes an element at the given index.
- **Search** – Searches an element using the given index or by the value.
- **Update** – Updates an element at the given index.

Two Types of Array

1. Single Array



2. Two Dimensional Array



Address calculation in single dimension array

Calculation Formula for single dimension array

$$\text{Address of } A[x] = B + W(x - LB)$$

B = base address

x = subscript

W = storage size (in bytes)

LB = lower bound of subscript, if not specified assumed 0 (zero).

Example

LB
4

given the base address of an array $A[1300 \dots 1700]$ as 1020 and size of each element is 2 bytes in the memory. Find the address of $A[1700]$.

B = 1020

x = 1700

W = 2

LB = 1300

$$\begin{aligned} \text{Formula} \cdot \text{Address of } A[x] &= B + W(x - LB) \\ &= 1020 + 2(1700 - 1300) \\ &= 1020 + 2(400) \\ &= 1020 + 800 \\ &= 1820 \end{aligned}$$

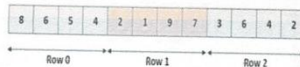
Address calculation in Two dimensional array

Address of an element of any array say $A[i][j]$ is calculated by two forms: Row major system and column Major system.

		Column Index			
		0	1	2	3
Row Index	0	8	6	5	4
	1	2	1	9	7
	2	3	6	4	2

Two-Dimensional Array

Row-Major (Row Wise Arrangement)



Column-Major (Column Wise Arrangement)

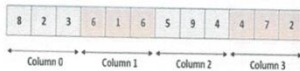


Figure 12 Alternatives to Linearization of Two-dimension Array

Formula for Row and Column wise

$A [I] [J]$
 row \leftarrow I \rightarrow column \leftarrow J

Row major system: address of $A [I] [J]$ Row major wise $B + W \times [N \times (I - L_r) + (J - L_c)]$

Column major system: address of $A [I] [J]$ Column major wise $B + W \times [(I - L_r) + M \times (J - L_c)]$

Where:

B = Base address

W = Storage size

I = Row subscript

L_r = Lower limit or start row index of matrix or 0

J = Column subscript

L_c = Lower limit of column or 0

M = Number of rows

N = Number of column

Formula for number of rows and column

$$\text{Number of rows (M)} = (U_r - L_r) + 1$$

$$\text{Number of column (N)} = (U_c - L_c) + 1$$

Note: $A [L_r \dots U_r, L_c \dots U_c] / A [20] [30] / A [40] [60]$

Example: An array $X [-15 \dots 10, 15 \dots 40]$ requires 1 byte of storage.
If beginning location is 1500 determine the location of $X [15] [20]$

Solution: Number of Rows $M = (U_r - L_r) + 1$
 $= 10 - (-15) + 1$
 $= 25 + 1$
 $M = 26$

Number of columns $N = (U_c - L_c) + 1$
 $= (40 - 15) + 1$
 $= 25 + 1$
 $N = 26$

Column Major wise calculation and Row Major wise calculation

Given: $B = 1500$ $J = 20$ $L_r = -15$ $M = 26$
 $W = 1$ $I = 15$ $L_c = 15$ $N = 26$

Column Major wise: $B + W \times [(I - L_r) + M \times (J - L_c)]$
 $= 1500 + 1 \times [(15 - (-15)) + 26 \times (20 - 15)]$
 $= 1500 + 1 \times [30 + 26 \times 5]$
 $= 1500 + 1 \times [160]$
 $= 1500 + 160$
 Column = 1660

$$\begin{aligned}
 \text{Row Major Wise} &: B + W + L \times N \times (I - L_r) + (J - L_c) \\
 &= 1500 + 1 \times 26 \times (15 - (-15)) + (20 - 15) \\
 &= 1500 + 1 \times [26 \times (30) + 5] \\
 &= 1500 + 1 \times [780 + 5] \\
 &= 1500 + 1 \times [785] \\
 &= 1500 + 785 \\
 \text{ROW} &= 2285
 \end{aligned}$$

Finding location of single array

Example:

0	1	2	3	4		100
2	9	5	4	6	+ given	102
104	106	108	110	112		104 2
						106 4
						108 5
						110 4
						112 6

Where: B = Base address

L_B = lower bound

W = memory

x

$$\text{Formula: } A[x] = B + W \times (x - L_B)$$

$$\begin{aligned}
 \text{Address of } A[3] &= B + W \times (x - L_B) \\
 110 &= 104 + 2 \times (3 - 0) \\
 &= 104 + 2 \times (3) \\
 &= 104 + 6 \\
 110 &= 110
 \end{aligned}$$

$$B = 104$$

$$W = 2$$

$$x = 3$$

$$L_B = 0$$

Finding location of two dimensional array

Two Dimensional Array

Where:

B = Base Address

W = Storage size of an element

L_r = Start row index of matrix/Lower bound of row = 0

L_c = Start column index of matrix/Lower bound of column = 0

M = number of rows in matrix = 3

N = number of columns in matrix = 4

i = row Subscript of element whose address to be found

j = column Subscript of element whose address to be found

	0	1	2	3
0	5	7	4	3
1	1	0	9	8
2	6	2	1	4

$$A[L_r \dots U_r][L_c \dots U_c]$$

DATA STRUCTURE AND ALGORITHMS

Jamaica Pingol

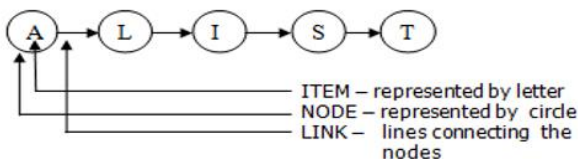
Bachelor of Science in Information Technology

2 – 1 N

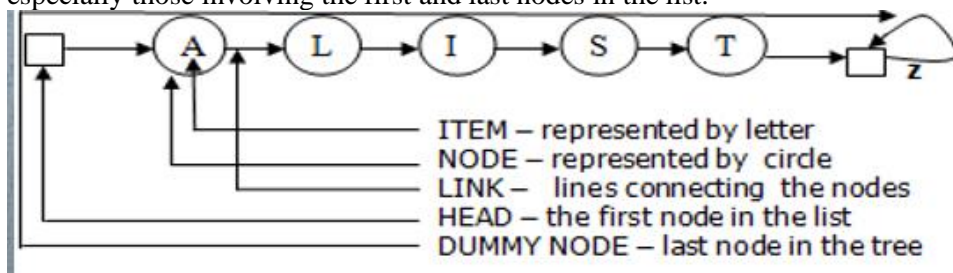
Lesson 4: Array with Implementation of Linked List



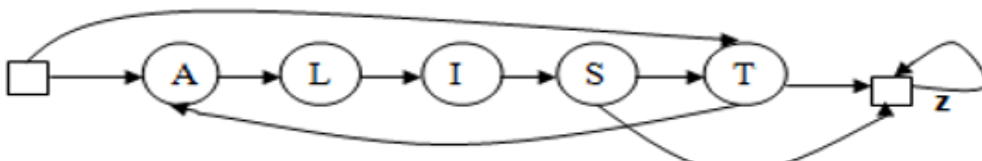
Linked list data structure provides **better memory management** than arrays. Because the linked list is allocated memory at run time, so there is **no waste of memory**. Performance wise linked list is **slower than array** because there is no direct access to linked list elements. Linked list is proved to be a useful data structure when the number of elements to be stored is **not known ahead of time**. **Simple Linked List** is an explicit arrangement of each item.

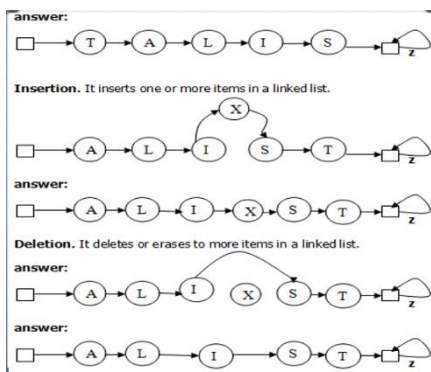


A **Simple Linked List with its Dummy Node** makes a certain manipulation with the links especially those involving the first and last nodes in the list.



Rearranging a Linked List makes a structural change by the changing just through the links.





Definition of Terms:

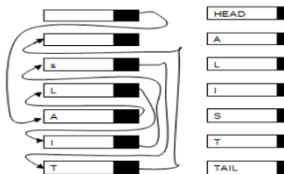
- **Storage allocation** – It is a place for storing information.
- **Key [Next [head]]** – refers to the information associated with the first item on the list
- **Key[next[next[head]]]** – refers to the second item on the list
- **Parallel arrays** – the structure can be built “on top of” the data
- **Key** – contains the data
- **Next** – All the structure in the parallel array

INSTRUCTIONS:

HEAD	HEAD	HEAD	HEAD	4	NEXT[4] = 0
TAIL	S	S	S	6	KEY[4] = HEAD
S	L	L	L	1	NEXT[3] = A
L	A	A	A	5	KEY[3] = L
A	TAIL	TAIL	TAIL	2	NEXT[2] = I
I	I	I	I	3	KEY[2] = S
T	T	T	T	1	NEXT[1] = T
					KEY[1] = TAIL

Example Given:

Head	4
tail	1
S	6
L	5
A	3
I	2
T	1



answer:



HEAD	HEAD	HEAD	HEAD	4	NEXT[4] = 0
TAIL	S	S	S	6	KEY[4] = HEAD
S	L	L	T	1	NEXT[3] = A
L	A	I	L	5	KEY[3] = L
A	TAIL	A	I	2	NEXT[2] = I
I	I	TAIL	A	3	KEY[2] = S
T	T	T	TAIL	1	NEXT[1] = T
					KEY[1] = TAIL

INSTRUCTIONS: (7 DISTINCT LETTERS)

1. Insert PROJECT IN AN EMPTY LIST
2. Insert PROJ after Head
3. Insert E after R
4. Insert C after O
5. Insert t after J

HEAD	HEAD	HEAD	HEAD	HEAD	HEAD	5	NEXT[5] = 0
TAIL	TAIL	P	P	P	P	6	KEY[5] = HEAD
	P	R	R	R	R	8	NEXT[4] = O
	R	O	E	E	E	2	KEY[4] = P
	O	J	O	O	O	4	NEXT[3] = J
	J	TAIL	J	C	C	7	KEY[3] = R
	E	E	TAIL	J	J	3	NEXT[2] = E
	C	C	C	TAIL	T	1	KEY[2] = C
	T	T	T	T	T	1	NEXT[1] = T
							KEY[1] = TAIL

Answer:



Calibri

DATA STRUCTURE AND ALGORITHMS

Jamaica Pingol

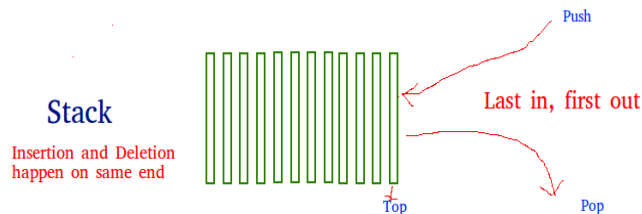
Bachelor of Science in Information Technology

2 – 1 N

Lesson 5: Stacks



Stacks - is a linear data structure which follows a particular order in which the operations are performed. The order may be LIFO (Last In First Out) or FILO (First In Last Out).



- **Push:** Adds an item in the stack. If the stack is full, then it is said to be an Overflow condition.
- **Pop:** Removes an item from the stack. The items are popped in the reversed order in which they are pushed. If the stack is empty, then it is said to be an Underflow condition.
- **Peek or Top:** Returns top element of stack.
- **isEmpty:** Returns true if stack is empty, else false.

Stack lastElement() method in Java with Example

The **Java.util.Stack.lastElement()** method in Java is used to retrieve or fetch the last element of the Stack. It returns the element present at the last index of the Stack.

Syntax:

`Stack.lastElement()`

Parameters: The method does not take any parameter.

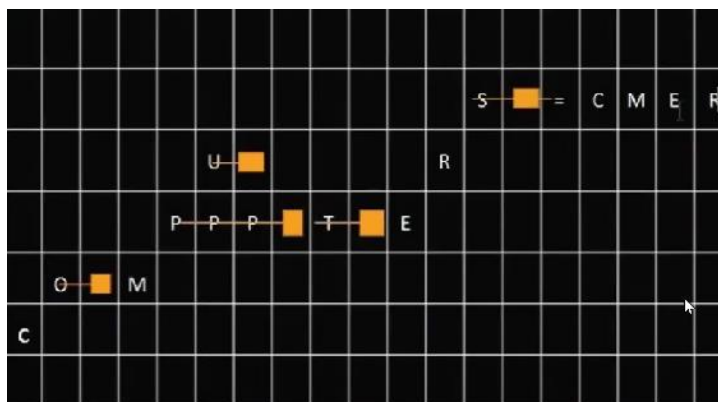
Return Value: The method returns the **last element** present in the Stack

Stacks (Last in First Out)

Note: **Letters** represent information or data while **asterisk(*)** represents deletion operation

Given:

CO*MPUT*ERS***



DATA STRUCTURE AND ALGORITHMS

Jamaica Pingol

Bachelor of Science in Information Technology

2 – 1 N

Lesson 6: Queues



Queue is an abstract data structure, somewhat similar to Stacks. Unlike stacks, a queue is open at both its ends.

One end is always used to insert data (enqueue) and the other is used to remove data (dequeue). Queue follows First-In-First-Out methodology, i.e., the data item stored first will be accessed first.

There are many real-life examples of a Queues.

- First in First out (FIFO)
- Last in Last out (LIFO)
- Fall-in line in the cashier when you are paying your tuition fee
- When you call to a customer service
- When you display bread in bakery
- When you dispose food in a restaurant

Queue Representation

As we now understand that in queue, we access both ends for different reasons. The following diagram given below tries to explain queue representation as data structure – As in stacks, a queue



Methods of Queue Interface

- **boolean add(E e):** This method adds the specified element at the end of Queue. Returns true if the element is added successfully or false if the element is not added that basically happens when the Queue is at its max capacity and cannot take any more elements.
- **E element():** This method returns the head (the first element) of the Queue.
- **boolean offer(object):** This is same as add() method.
- **E remove():** This method removes the head(first element) of the Queue and returns its value.

- **E poll():** This method is almost same as remove() method. The only difference between poll() and remove() is that poll() method returns null if the Queue is empty.
- **E peek():** This method is almost same as element() method. The only difference between peek() and element() is that peek() method returns null if the Queue is empty.

Queues(First in First Out)

Note: **Letters** represent information or data while **asterisk(*)** represents deletion operation

Given:

CO*MPUT*ERS***

