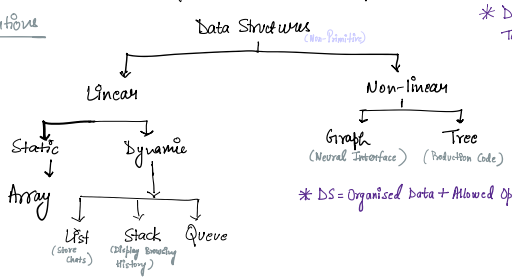
Introduction

17 March 2024 22:54

Introduction to Data Structures-
↳ Data organization, storage & management format
Data Structure is a structure of Data
* Data Presented in Tables

*Applications



* DS = Organised Data + Allowed Operations

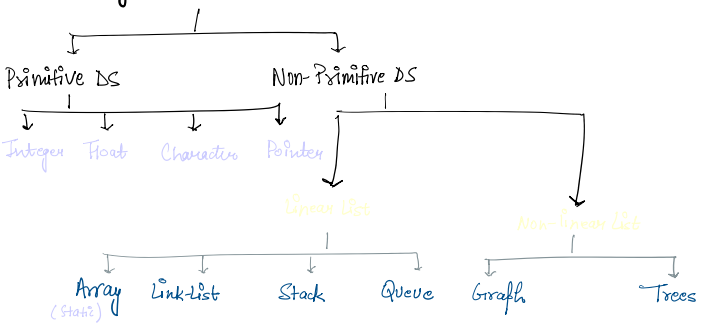
* Every piece of data that is stored in a computer is kept in a memory cell with a specific address.
(i) C - contain address reserved for certain data types
(ii) Python - No Reservation

OPERATIONS - (on non-primitive DS)
Creation, Modification, Deletion, Storage, Traversing, Duplication, Sorting, Filtering, Concatenation, Splitting, Searching, Merging

* Difference btw Linear & Non-linear DS -

LINEAR	NON-LINEAR
① Elements are stored in sequential manner	① Complex Relations
② Fixed starting & ending point	② No clear beginning or end
③ Accessing elements involves traversing across a single path.	③ Accessing elements involves traversing through various branches & paths.
④ Array, List, stack, queue	④ Tree, Graph

Classification of Data Structure



ARRAYS (index & values)

-> Set of finite number of homogeneous elements (same data items) (can contain one type of data only)
-> Declaration of array - `int arr [10]`
(Data Type, Name of Array, Length of Array)
-> Array - 0 1 2 3 4 5 6 7 8 9 / 7 9 11 32 55 69 75 2 97
(Elements of array are always stored in consecutive memory location)
-> No. of Elements in Array = (Upperbound - Lowerbound) + 1
* In above case - (9-0)+1 = (10)
-> Common Operations performed on Arrays -
(i) Reading an Array
(ii) Writing an Array
(iii) Insertion of New Element
(iv) Deletion of Required Element
(v) Modification of an Element
(vi) Merging of Arrays

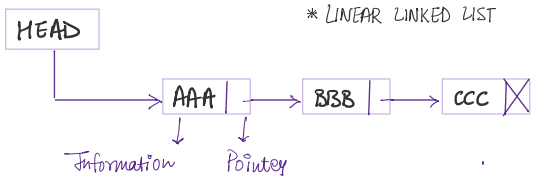
```
For (i=0, i<=9, i++)
    scanf ("%d", &arr[i]);
```

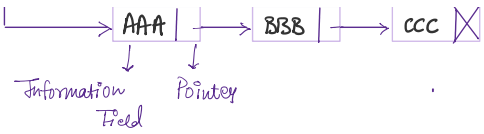
```
For (i=0, i<=9, i++)
    printf ("%d", arr[i]);
```

* Multi-dimensional Arrays- "Array of Arrays"

LISTS

(Collection of variable no. of data items)
(Collection of Nodes)



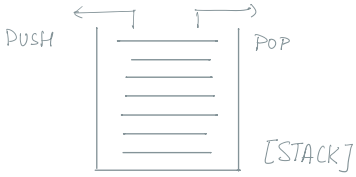


TYPES OF LINKED LIST

① Single Linked List
② Doubly Linked List
③ Single Circular Linked List
④ Doubly Circular Linked List

STACK

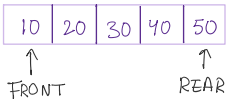
* LIFO – Last In First Out



→ Stack can be implemented in two ways –
(i) Using Arrays (Static Implementation)
(ii) Using Pointers (Dynamic Implementation)

QUEUE

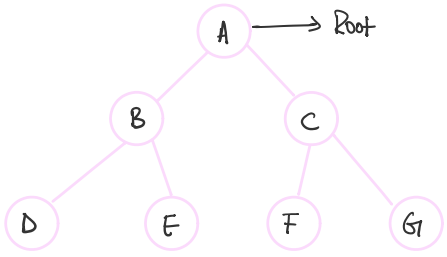
* FIFO – First In First Out



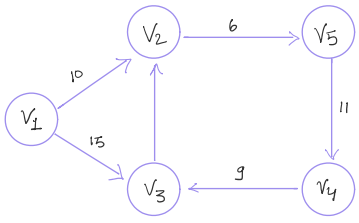
→ Queues can be implemented in two ways – Arrays & Pointers

TREES

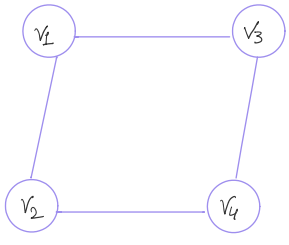
* Organises the data into branches, which relates the information.



GRAPHS



(i) Directed & Weighted Graph



(ii) Undirected Graph

Types of Graphs –
(i) Directed Graph
(ii) Undirected Graph
(iii) Simple Graph
(iv) Weighted Graph
(v) Connected Graph
(vi) Non-connected Graph

ROW MAJOR ORDER –

	Col0	Col1	Col2
Row0	1	2	3
Row1	4	5	6

m*n
2*3 Matrix

i = row, j = column
m & n = matrix indexes
Base Address = 50
Size of int = 4 Bytes

(for 1D Array)

= ((i*n)+j) * size of each element + Base Address

= (0*3)+1) *4+50

= 54

(for 2D Array)

= B.A + W * ((I-LR) *N + (J-LC))

= 100 + 2*(2*5+3)

= 100 + 2*(13) = 126

COLUMN MAJOR ORDER

00	10	01	11	02	12	→ Array Index
1	4	2	5	3	6	
50	54	58	62	66	70	→ Memory Address

= ((j * m) + i) * size of each element + Base Address

INSERTION

Initialize value of I
Set i = len
Repeat for i = len down to pos
[Shift elements down by pos 1]
Set a[i+1] = a[i]
[End loop]
[Insert element at required position]
Set a[pos] = num
[Reset len] set len = len+1
Display new list of array
end

```
for (i=len;i>pos;i--)
{
a[i+1] = a[i];
}
a[pos] = num ;
```

DELETION

Set item = a[pos]
Repeat for j = pos to n-1
[Shift elements by pos 1 upward]
Set a[j] = a[j+1]
[End Loop]
reset n = n-1
Display new list of array
end

```
Item = a[pos]
for(j= pos ;j<=n-1;j++)
{
a[j]= a[j+1];
}
n= n-1
```

TRAVERSING OF ARRAY

(Let LB the lowerbound & UB the upperbound)

→ [Initialise counter]
→ Set i at lower bound LB
→ Repeat for i = LB to UB
→ [visit element]
→ display a[i]
→ [end of loop]
→ exit

MERGE ARRAY

```
int i ,j ;
j= 0
for (i=0, i<m, i++)
{
C[j] = a[i];
```

```
for (i=0, i<m, i++)
{
C[j] = a[i];
for (i=0, i<n, i++)
{
C[j] = b[i];
J= j+1;
}
}
J= j+1
}
```

POINTERS

→ Pointers serve as a derived data type that stores the memory address of another variable.

→ A pointer holds the location where the data resides in the computer's memory.

→ Powerful tools for managing memory & optimizing data structures.

→ Allow us to work with memory addresses directly, which is essential for efficient programming.

→ To declare a pointer, use the (*) dereference operator before its name

```
int * ptr;
```

→ Wild Pointers – pointer pointing to some random memory address as it's not ~~int~~ initialised.

POINTER INITIALISATION -

→ Assign some initial value to the pointer variable.

→ Use the (&) **address of operator** to get the memory address of a variable & then store it in the pointer variable.

```
int v = 20;
int * ptr;
ptr = &v;

or
int * ptr = &v;      // single line declaration

int n;
char c;
int * ptrn;          // * ptrn give the content of the location pointed to by ptr

c= 'x'
n= 15;
ptrn = &n            // If n is a variable, then &n is the address of the variable
```

ADVANTAGES -

① Efficiency : Pointers enhance performance in various scenarios;
- → To access a memory location
- → Traversing data structures like strings, lookup tables, control tables & trees.
- → Manipulating arrays efficiently
- → Handling repetitive processes.
- → Allocation & dynamic memory & its distribution

DISADVANTAGES -

- → A bit difficult to understand
- → Can cause several errors; segmentation errors or unrequired memory access.
- → If a pointer has an incorrect value, it may corrupt the memory.
- → Pointers may also cause memory leakage.
- → Relatively slower than the variables.

ANALYSIS OF ALGORITHM (coined by Donald Knuth)

Worst - Case	Max. no. of steps taken on any instance of size a.
Best - Case	Min. no. of steps taken on any instance of size a.
Average - Case	Avg. no. of steps taken on any instance of size a.