**Data Structures**

Way of organizing data to handle them efficiently.

**Types Of Data Structures**

**1. Array:**

* Collection of items stored in contiguous memory.
* Collection of elements which are of same datatype.
* Memory Allocation – Fixed size. One Memory Block is allocated for entire array.
* Index can be used to access individual elements

Disadvantages:

1. Specifying array size before using
2. One block allocation
3. Complex position based insertion

**2. Linked List: (Linear DS)**

* Elements are not stored at contiguous memory location.
* It consists of nodes – Data field and reference
* Data field contains the element and reference contains the memory location of next element
* Last Element points to NULL
* Can grow or shrink in size during execution
* In initial phase, we can start with space for one element and add on new elements when needed.

Advantages:

1. Dynamic size
2. Ease of insertion or deletion

Disadvantages:

1. Random access is not allowed .We have to access element sequentially from first node.
2. Extra memory space for pointers.

**3. Stack : (Linear DS)**

* Follows particular order in which operations are performed
* LIFO (Last In First Out) or (First In Last Out)
* Operations- push,pop,top,isEmpty
* Insertion and deletion at one end.
* Trying to pop out an element in empty stack is underflow
* Trying to push an element in full stack is overflow
* Order in which the data is inserted is important

Pros and Cons

* very fast access
* don't have to explicitly de-allocate variables
* space is managed efficiently by CPU, memory will not become fragmented
* local variables only

**4. Queue:**

* Insertion and deletion happens in two different end.
* Ordered List in which insertion are done at one end and deletion is done in other end
* First element to be inserted is the first one to be deleted.
* Element is inserted in queue-Enqueue
* Element is removed from Queue – Dequeue
* Dequeuing an empty queue is called underflow
* Enqueueing the full queue is called overflow

Pros And Cons

* variables can be accessed globally
* no limit on memory size
* (relatively) slower access
* no guaranteed efficient use of space, memory may become fragmented over time as blocks of memory are allocated, then freed
* you must manage memory (you're in charge of allocating and freeing variables)
* variables can be resized using realloc()

Interpreter/Compiler

|  |  |
| --- | --- |
| Interpreter | Compiler |
| Translates program one statement at a time. | Scans the entire program and translates it as a whole into machine code. |
| It takes less amount of time to analyze the source code but the overall execution time is slower. | It takes large amount of time to analyze the source code but the overall execution time is comparatively faster. |
| No intermediate object code is generated, hence are memory efficient. | Generates intermediate object code which further requires linking, hence requires more memory. |
| Continues translating the program until the first error is met, in which case it stops. Hence debugging is easy. | It generates the error message only after scanning the whole program. Hence debugging is comparatively hard. |
| Programming language like Python, Ruby use interpreters. | Programming language like C, C++ use compilers. |