**Data Structure**

**Data Structure and Algorithm**

1. [Basic](#ds_basic)
   1. [Introduction](#ds_introduction)
   2. [Recursion](#ds_recursion)
   3. [Algo Run Time Analysis](#ds_run_time_algo)
2. [Physical Data Structure](#physical_ds)
   1. [Array](#array_ds)
      1. [Single Dimension Array](#oned_array)
      2. [Double Dimension Array](#twod_array)
   2. [Linked List](#linked_list)
      1. [Single Linked List](#single_ll)
      2. [Circular Single Linked List](#circular_single_ll)
      3. [Double Linked List](#double_ll)
      4. [Circular Double Linked List](#circular_double_ll)
3. [Logical Data Structure](#logical_ds)
   1. [Stack](#lds_stack)
   2. [Queue](#lds_queue)
   3. Tree
   4. Hashing
   5. Graph
4. Miscellaneous Data Structure
   1. Sorting
5. Algo Techniques
   1. Magic Framework
   2. Greedy Algorithm
   3. Divide and Conquer
   4. Dynamic Programming

**1. Basic**

**1.1. Introduction**

Data Structure is way to organize data in a way that enables it to be processed in an efficient time.

Algorithm is a set of rules to be followed to solve a problem.

Type of DS

Primitive DS Non-Primitives DS

Integer

Float

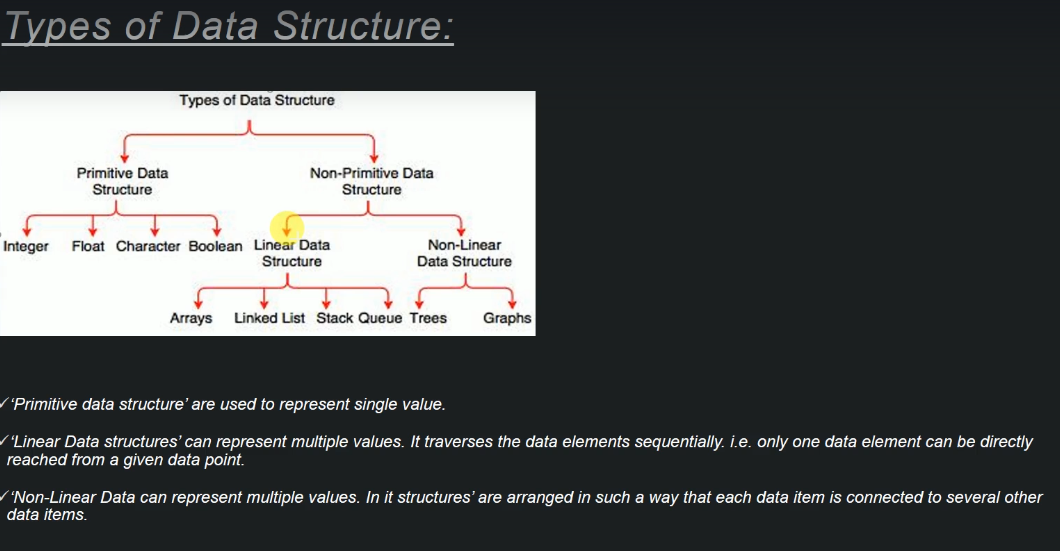
Character Physical DS Logical DS

Boolean Array Stack

Linked List Queue

Tree

Graph



**1.2. Recursion**

* Same operation is performed multiple time with different inputs.
* In every step we try to make problem smaller
* We mandatory need to have a base condition when tells system when to stop the recursion
* Format of a recursive method

Recursive Case: Cases where the function needs to recall itself.

Base Case: Cases where the function doesn’t recall.

sampleRecursion(parameter){  
 if(baseCase is satisfied)  
 return same base case value  
 else  
 return sampleRecursion(modified parameter)  
 }

* How recursion work internally

main(){ ----1  
 bar();  
 sop("main"); ----14  
}  
bar(){ ----3  
 doWork();  
 sop("bar"); ----12  
}  
doWork(){ ----5  
 doMore();  
 sop("doWork"); ----10  
}  
doMore(){ ----7  
 sop("doMore"); ----8  
}  
  
STACK  
-----  
  
| |  
| |  
| doWork() | ----6, 9  
| bar() | ----4, 11  
| main() | ----2, 13  
|-----------|

main(){  
 foo(3);  
}  
foo(n){  
 if(n<1)  
 return   
 else  
 foo(n-1)  
 print "Hello n"  
}  
  
| |  
|foo(1) |  
|foo(2) |  
|foo(3) |  
|main() |  
|--------|

* Program and examples

PrintNumber:

FactorialSeries:

FibonacciSeries:

NumberPalindrome:

<https://github.com/bibhusprasad/011_data_structure/tree/master/src/s1_c2_recursion>

**1.3. Algo Run Time Analysis**

It is a study of given algorithm run time. In layman language we can say “How much time will the given algorithm will take to run”.

Notification for algo run time analysis.

3 notations used to calculate run time analysis.

1. **Omega (Ω)**

* Min Time
* Given lower bound of a given algorithm.
* For any given input running time of a given algorithm will not be less than a given time.

e.g. sort 1000 number

Omega (10): Never take less than 10 sec. it may take 11,12…sec

1. **Big-O (O)**

* Max Time
* Given upper bound of a given time
* For any given input, running time of a given algorithm will not be more than given time

E.g. sort 1000 input

Big-o=O (100): Time taken to solve is less than 100 sec. it may take 99, 98… sec

1. **Theta**

* Decide whether upper bound and lower bound of a given algorithm are same or not.
* For given input running time of a given algorithm will on an average be equal to given time.

Example

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 5 | 13 | 3 | … | 55 | 41 | 23 | … | 19 | 1 | 10 | 17 |

Find a given number in that array

We search in each index. It may be in 1st index or it may be last index.so time take is n\*1.

So, Omega (1): it may be in 1st cell i.e. not less than minimum time

Big-O(n): it may be in last cell i.e. max time

Theta: (n/2): average time.

**Time Complexity Name Example**

O (1) constant Adding an element at front of a limited list

O (log n) logarithm Find an element on sorted array

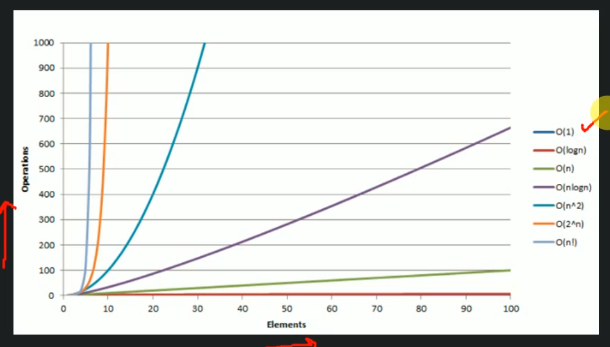
O (n) liner find an element on un sorted array

O (n log n) liner logarithm Marge sort

O (n2) Quadratic shortest path between 2 nodes in a graph

O (n3) Cubic Matrix multiplication

O (2n) Exponential Tower of Hanoi Tower



How to calculate ‘Algorithm time complexity’

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 5 | 13 | 3 | … | 55 | 41 | 23 | … | 19 | 1 | 10 | 17 |

findBiggestNumber(int arr[])  
 biggestNumber = arr[0] ----------------O(1)  
 loop: i = 1 to length(arr) - 1 --------O(n) } -------O(n)  
 if arr[i] > biggestNumber }-------O(1) }  
 biggestNumber = arr[i] }  
 return biggestNumber ----------------O(1)  
  
 Time Complexity : O(1) + O(n) + O(1) = O(n)

**Time Complexity of recursive algorithm #1**

findBiggestNumber(arr[], size) -----T(n)  
 static highest = Integer.MIN -----O(1)  
 if size equals-1 -----O(1)  
 return highest -----O(1)  
 else -----O(1)  
 if arr[size] > highest -----O(1)  
 update highest -----O(1)  
 return findBiggestNumber(arr, size-1) -----T(n-1)  
  
  
 T(n) = O(1)+T(n-1) -----equation 1  
 T(-1) = O(1) -----Base condition  
 T(n-1) = O(1) + (T(n-1)-1) -----equation 2  
 T(n-2) = O(1) + (T(n-2)-1) -----equation 3  
  
 T(n) = 1+T(n-1)  
 = 1+1+(T(n-1)-1) -----from equation 2  
 = 2+T(n-2)  
 = 2+1+T(n-2)-1) -----from equation 3  
 = 3+T(n-3)  
 = K+T(n-K) -----K as constant  
 = (n+1)+T(n-(n+1)) -----replace k with n+1  
 = n+1+T(-1) -----Base condition  
 = n+1+1  
 = O(n)

**Time Complexity of recursive algorithm #2**

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 10 | 20 | 30 | 40 | 50 | 60 | 70 | 80 | 90 | 100 | 110 |

Given sorted array of 11. find time complexity for search no 110

BinarySearch(int findNumber, int arr[], start, end) -----T(n)  
 if(start equals end) -----O(1)  
 if(arr[start] equals findNumber) -----O(1)  
 return start -----O(1)  
 else -----O(1)  
 return error number not present -----O(1)  
  
 mid = findMid(arr[],start,end) -----O(1)  
 if mid > findNumber -----O(1)  
 BinarySearch(int findNumber, int arr[], start, mid) -----T(n/2)  
 else if mid < findNumber -----O(1)  
 BinarySearch(int findNumber, int arr[], mid, end) -----T(n/2)  
 else if mid equals findNumber -----O(1)  
 return mid -----O(1)  
  
 T(n) = T(n/2)+1 -----equation 1  
 T(1) = 1 -----base condition  
 T(n/2) = T(n/4) + 1 -----equation 2  
 T(n/4) = T(n/8) + 1 -----equation 3  
  
  
 T(n) = T(n/2)+1  
 = (T(n/4)+1)+1)  
 = T(n/4) + 2  
 = (T(n/8)+1)+2)  
 = T(n/8) + 3  
 = T(n/2k) + k  
 = T(1) + log n replace k with log n  
 = O(log n)

**2. Physical Data Structure**

**2.1. Array Data Structure**

* Array can store data of specified data type
* It has contagious memory location
* Every cell of an array has unique index
* Index start with 0
* Size of array need to specified mandatory and cannot modified

**One-dimensional Array**

Having one row

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 10 | 20 | 30 | 40 | 50 |

**Two-dimensional Array**

m\*n array i.e. m rows and n column. 2\*5 example

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 10 | 20 | 30 | 40 | 50 |
| 100 | 200 | 300 | 400 | 500 |

**Three-dimensional Array**

m\*n\*l i.e. depth \* row \* column e.g. rube cube.

**Q. How array represent in memory**

A. arr[5] = {10,20,30,40,50};

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 10 | 20 | 30 | 40 | 50 |

arr[2][3] = {{10,20,30},{40,50,60}};

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 10 | 20 | 30 | 40 | 50 | 60 |

arr[2][3][3] = {{{10,20,30},{40,50,60},{70,80,90}},

{{100,110,120},{130,140,150},{160,170,180}}};

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 10 | 20 | 30 | 40 | 50 | 60 | 70 | 80 | 90 | 100 | 110 | 120 | 130 | 140 | 150 | 160 | 170 | 180 |

**2.1.1. One Dimensional Array**

**Time complexity: Declaring, Instantiating, Initializing 1D array.**

Declare  
 dataType []arr; -----O(1)  
 e.g. int []arr;  
  
Instantiation  
 arr = new dataType[size]; -----O(1)  
 e.g. arr = new int[5];  
  
Initialization  
 arr[0] = 10; -----O(1)  
 arr[1] = 20; -----O(1)  
 arr[2] = 30; -----O(1)  
 arr[3] = 40; -----O(1)  
 arr[4] = 50; -----O(1) i.e. O(n)  
  
Declare, Instantiation and Initialization  
int arr[] = {10,20,30,40,50}; -----O(1)

**Time complexity: Inserting a value in 1D array**

insert(arr, valueToBeInserted, location)  
 if arr[location] occupied -----O(1)  
 return error; -----O(1)  
 else -----O(1)  
 arr[location] = valueToBeInserted -----O(1) i.e. O(1)

Total time complexity: O(1)  
Space complexity: O(1)

**Time complexity: Traversing in 1D array**

traverseArray(arr)  
 loop i = 0 to arr size -----O(n)  
 print arr[i] -----O(1)  
  
Total time complexity: O(n)  
Space complexity: O(1)

**Time complexity: Accessing given cell 1D array**

accessingCell(arr, cellNo){  
 if(cellNo > arr.size) ----O(1)  
 return error cell no is higher than array size ----O(1)  
 else ----O(1)  
 print arr[cellNo] ----O(1)  
}  
  
Total time complexity: O(1)  
Space complexity: O(1)

**Time complexity: Search a given value in an 1D array**

searchInAnArray(arr, value){  
 loop i=0 to arr size -----O(n)  
 if arr[i] equals value -----O(1)  
 return i; -----O(1)  
 else -----O(1)  
 return error value not present -----O(1)  
}  
  
Total time complexity: O(n)  
Space complexity: O(1)

**Time complexity: Deletion a value in an 1D array**

deletion(arr, location){  
 if arr[location] is occupied -----O(1)  
 arr[location] = Integer.MIN -----O(1)  
 else -----O(1)  
 return location is already empty -----O(1)  
}  
  
Total time complexity: O(1)  
Space complexity: O(1)

**Time Complexity of 1D array**

|  |  |  |
| --- | --- | --- |
| **Operations** | **Time Complexity** | **Space Complexity** |
| Create an empty array | O(1) | O(n) |
| Inserting a value in an array | O(1) | O(1) |
| Traversing in a given array | O(n) | O(1) |
| Accessing a given cell | O(1) | O(1) |
| Searching a given value | O(n) | O(1) |
| Delete a given cell value | O(1) | O(1) |
| Delete a given value | O(n) | O(1) |

**Example**

<https://github.com/bibhusprasad/011_data_structure/tree/master/src/s2_c1_array>

**2.1.2. Two Dimensional Array**

**Time complexity: Declaring, Instantiating, Initializing 2D array.**

Declare  
 dataType [][]arr; -----O(1)  
 e.g. int [][]arr;  
  
Instantiation  
 arr = new dataType[row][column]; -----O(1)  
 e.g. arr = new int[2][3];  
  
Initialization  
 arr[0][0] = 10; -----O(1)  
 arr[0][1] = 20; -----O(1)  
 arr[0][2] = 30; -----O(1)  
 arr[1][0] = 40; -----O(1)  
 arr[1][1] = 50; -----O(1)  
 arr[1][2] = 60; -----O(1) i.e. O(m\*n)  
  
Declare, Instantiation and Initialization  
int arr[][] = {{10,20,30},{40,50,60}} -----O(1)

**Time complexity: Inserting a value in 2D array**

insert(arr, valueToBeInserted, row, column)  
 if arr[row][column] occupied -----O(1)  
 return error location is already occupied; -----O(1)  
 else -----O(1)  
 arr[row][column] = valueToBeInserted -----O(1) i.e. O(1)  
  
Total time complexity: O(1)  
Space complexity: O(1)

**Time complexity: Traversing in 2D array**

traverseArray(arr, row, column)  
 loop i = 0 to row -----O(m)  
 loop j = 0 to column -----O(n)  
 print arr[i][j] -----O(1)  
  
Total time complexity: O(mn)  
Space complexity: O(1)

**Time complexity: Accessing given cell 2D array**

accessingCell(arr, rowNo, columnNO){  
 return arr[row][column] -----O(1)

Total time complexity: O(1)  
Space complexity: O(1)

**Time complexity: Search a given value in an 2D array**

searchInAnArray(arr, value){  
 loop i=0 to row -----O(m)  
 loop j=0 to column -----O(n)  
 if arr[i][j] equals value -----O(1)  
 print(i, j); -----O(1)  
 else -----O(1)  
 return error value not present -----O(1)  
}  
  
Total time complexity: O(mn)  
Space complexity: O(1)

**Time complexity: Deletion a value in an 2D array**

deletion(arr, row, column){  
 if arr[row][column] is occupied -----O(1)  
 arr[row][column]= Integer.MIN -----O(1)  
 else -----O(1)  
 return location is already empty -----O(1)  
}  
  
Total time complexity: O(1)  
Space complexity: O(1)

**Time Complexity of 2D array**

|  |  |  |
| --- | --- | --- |
| **Operations** | **Time Complexity** | **Space Complexity** |
| Create an empty array | O(1) | O(mn) |
| Inserting a value in an array | O(1) | O(1) |
| Traversing in a given array | O(mn) | O(1) |
| Accessing a given cell | O(1) | O(1) |
| Searching a given value | O(mn) | O(1) |
| Delete a given cell value | O(1) | O(1) |
| Delete a given value | O(mn) | O(1) |

**Example**

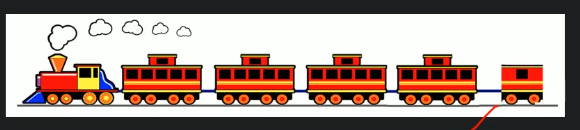
<https://github.com/bibhusprasad/011_data_structure/tree/master/src/s2_c1_array>

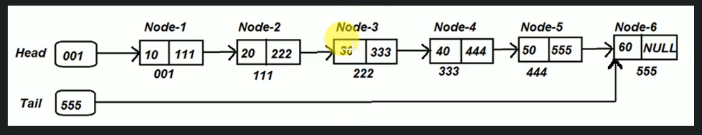
**2.2. Linked List**

A linked list is a liner data structure where each element is a separate object. Each element is also called as node. Node comprises two thing one is data other is reference of next node.

Linked list is variable in size

e.g Train





* There must be a head and tail present in Linked list
* Node contain data and next node reference
* It is liner means it travers sequentially i.e. if you want to travers from note 1 to 5 then you can direct go to node 5. You need to go via node 2,3,4.

Difference between Linked List and Array

* Separate Object vs One object
* Delete an Object vs Delete a value
* Variable size vs Fixed in size
* Traversing access(traverse node by node) vs Random access(direct access any cell)

Components of Linked List

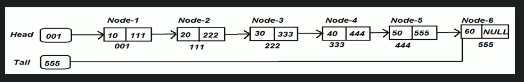
* Node: Contains data and reference of next node
* Head: Reference to first node of list
* Tail: Reference to last node of list

Types of Linked List

* Single Linked List

In Single linked list each node contains the data and reference of next node.

It is most basic LL which gives the flexibility to add/remove nodes at run time.

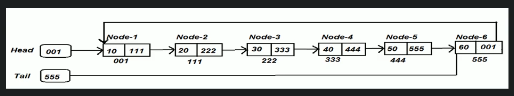


* Circular Single Linked List

In Circular linked list same as SLL. In case CSLL last node contain the address of node 1. Where in SLL last node contain the address null.

When we want to loop through the list indefinitely until the list exist.

e.g. LUDO game. Here 4 players is present. 1st p1 turn the p2 then p3 then p4. Again from p4 , p1 turn will come. IN SLL it is not possible.

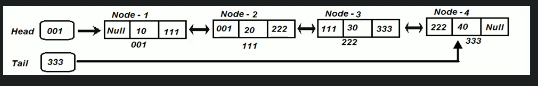


* Double Linked List

In DLL each node contains two reference, i.e. previous and next node reference.

When we want to in both directions depending upon our requirement.

e.g. Music player. Here we can able to access both next song and previous song.

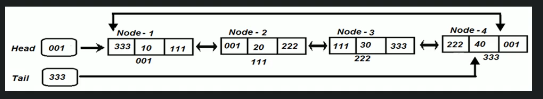


* Circular Double Linked List

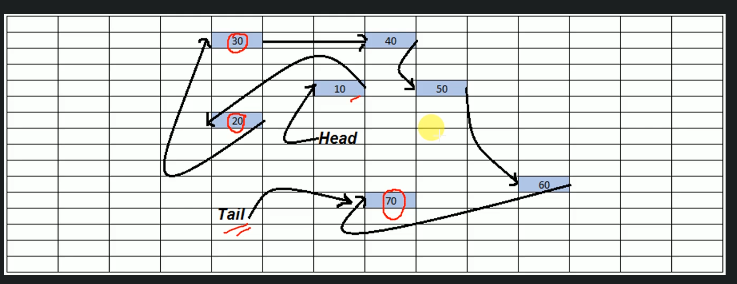
In CDLL same as DLL, the only change is that end node contains the reference of first node and vice versa.

Want to loop in through list in indefinitely in both forward and backward direction.

e.g. Alt+Tab button in windows OS.



How linked list store in memory



Common Operation of linked list

Creation of linked list

Insertion of linked list

Traverse of linked list

Searching in a linked list

Deletion of a node in linked list

Deletion of a linked list

**2.2.1 Single Linked List:**

**Time Complexity: Single Linked List: Creation**

CreateSingleLinkedList(nodeValue)   
 create a head and tail pointer and initialize with ZERO ---- O(1)  
 create a blank node ---- O(1)  
 node.value=nodeValue ---- O(1)  
 node.next=null ---- O(1)  
 head=node ---- O(1)  
 tail=node ---- O(1)  
   
 Time Complexity O(1)  
 Space Complexity O(1)

**Time Complexity: Single Linked List: Insertion**

* **Insert at Start**
* **Insert at last**
* **Insert at specified location**

insertLinkedList(head, nodeValue, location)  
 create a blank node -----O(1)  
 node.value = nodeValue -----O(1)

if(!existLinkedList(head)) -----O(1)  
 return error //Linked list not present -----O(1)  
 else if(location equlas 0)   
 //insert at first position -----O(1)  
 node.next = head -----O(1)  
 head=node -----O(1)  
 else if(location equals last)  
 //insert at last -----O(1)  
 node.next = null -----O(1)  
 last.next = node -----O(1)  
 last = node -----O(1)  
 //to keep track of last node   
 else //insert at specific location -----O(1)  
 loop: tempNode = 0 to location -1 -----O(n)  
 //loop till we reach specified node and end the loop  
 node.next = tempNode.next -----O(1)  
 tempNode.next=node -----O(1)  
   
Time complexity ----- O(n)  
Space complexity ----- O(1)

**Time Complexity: Single Linked List: Traverse**

traverseLinkedList(head)  
 if head == NULL -----O(1)  
 then return null -----O(1)  
 loop: haed to tail -----O(n)  
 print currentNode.value -----O(1)  
  
Time complexity -----O(n)  
Space complexity -----O(1)

**Time Complexity: Single Linked List: Searching**

searchNode(node, nodeValue)  
 loop: tempNode = start to tail ----O(n)  
 if(tempNode.value equals nodeValue) ----O(1)  
 print tempNode.value ----O(1)  
 return value found ----O(1)  
 return value not found ----O(1)  
   
Time complexity -----O(n)  
Space complexity -----O(1)

**Time Complexity: Single Linked List: Deletion of a node**

deleteOfNode(head, location)  
 if(!existLinkedList(head)) ----O(1)  
 return error ----O(1)  
 else if(location equals 0) ----O(1)  
 head = head.next ----O(1)  
 if this is the only element

in list then update tail = null; ----O(1)  
 else if(location >= last) ----O(1)  
 if (current node is only node in list)

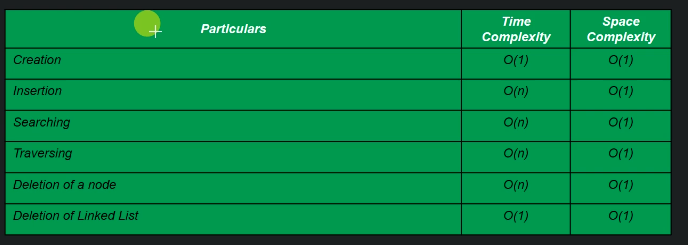
then head=tail=null; return ----O(1)  
 loop till 2nd last node(tmpNode) ----O(1)  
 tail = tmpNode; ----O(1)  
 tmpNode.next = null; ----O(1)  
 else ----O(1)  
 loop.tmpNode=start to location-1 ----O(n)  
 tmpNode.next=tempNode.next.next ----O(1)  
   
Time complexity -----O(n)  
Space complexity -----O(1)

**Time Complexity: Single Linked List: Deletion a linked list**

deleteLikedList(head, tail)  
 head = null ----O(1)  
 tail = null ----O(1)

Time complexity -----O(1)  
Space complexity -----O(1)

**Time Complexity of Single Linked List array**

****

<https://github.com/bibhusprasad/011_data_structure/tree/master/src/s2_c2_linkedlist>

<https://github.com/bibhusprasad/011_data_structure/tree/master/src/s2_c2_node>

**2.2.2. Circular Single Linked List:**

**Time Complexity: Circular Single Linked List: Creation**

createCircularSingleLinkedList(nodeValue)  
 create a head and tail pointer and initialize with ZERO ---- O(1)  
 create a blank node ---- O(1)  
 node.value=nodeValue ---- O(1)  
 node.next=node ---- O(1)  
 head=node ---- O(1)  
 tail=node ---- O(1)  
  
Time Complexity O(1)  
Space Complexity O(1)

**Time Complexity: Circular Single Linked List: Insertion**

* **Insert at Start**
* **Insert at last**
* **Insert at specified location**

insertLinkedList(head, nodeValue, location)  
 create a blank node -----O(1)  
 node.value = nodeValue -----O(1)  
  
 if(!existLinkedList(head)) -----O(1)  
 return error //Linked list not present -----O(1)  
 else if(location equals 0)  
 //insert at first position -----O(1)  
 node.next = head -----O(1)  
 head = node -----O(1)  
 tail.next=head -----O(1)  
 else if(location equals last)  
 //insert at last -----O(1)  
 node.next = head -----O(1)  
 tail.next = node -----O(1)  
 last = node -----O(1)  
 //to keep track of last node  
 else //insert at specific location -----O(1)  
 loop: tempNode = 0 to location -1 -----O(n)  
 //loop till we reach specified node and end the loop  
 node.next = tempNode.next -----O(1)  
 tempNode.next=node -----O(1)  
  
Time complexity ----- O(n)  
Space complexity ----- O(1)

**Time Complexity: Circular Single Linked List: Traverse**

traverseLinkedList(head)  
 if head == NULL -----O(1)  
 then return null -----O(1)  
 loop: haed to tail -----O(n)  
 print currentNode.value -----O(1)  
  
Time complexity -----O(n)  
Space complexity -----O(1)

**Time Complexity: Circular Single Linked List: Searching**

searchNode(node, nodeValue)  
 loop: tempNode = start to tail ----O(n)  
 if(tempNode.value equals nodeValue) ----O(1)  
 print tempNode.value ----O(1)  
 return value found ----O(1)  
 return value not found ----O(1)  
   
Time complexity -----O(n)  
Space complexity -----O(1)

**Time Complexity: Circular Single Linked List: Deletion of a node**

deleteOfNode(head, location)  
 if(!existLinkedList(head)) ----O(1)  
 return error ----O(1)  
 else if(location equals 0) ----O(1)  
 head = head.next ----O(1)  
 tail.next = head ----O(1)  
 if this is the only element in list then update tail = null;  
 ----O(1)  
 else if(location >= last) ----O(1)  
 if (current node is only node in list)  
 then head=tail=null ----O(1)  
 loop till 2nd last node(tmpNode) ----O(1)  
 tail = tmpNode; ----O(1)  
 tmpNode.next = head; ----O(1)  
 else ----O(1)  
 loop.tmpNode=start to location-1 ----O(n)  
 tmpNode.next=tempNode.next.next ----O(1)  
  
Time complexity -----O(n)  
Space complexity -----O(1)

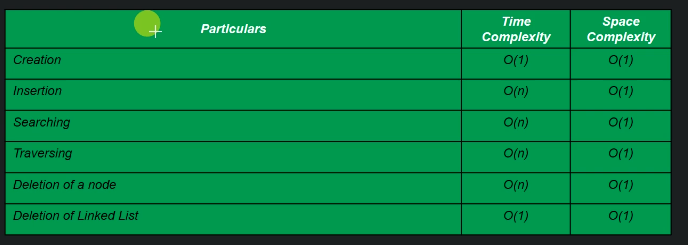
**Time Complexity: Circular Single Linked List: Deletion a linked list**

deleteLikedList(head, tail)  
 head = null ----O(1)

tail.next = null -----O(1)  
 tail = null ----O(1)

Time complexity -----O(1)  
Space complexity -----O(1)

**Time Complexity of Circular Single Linked List array**

****

<https://github.com/bibhusprasad/011_data_structure/tree/master/src/s2_c2_linkedlist>

<https://github.com/bibhusprasad/011_data_structure/tree/master/src/s2_c2_node>

**2.2.3. Double Linked List:**

**Time Complexity: Double Linked List: Creation**

CreateDoubleLinkedList(nodeValue)  
 create a blank node ---- O(1)  
 node.value=nodeValue ---- O(1)  
 head=node ---- O(1)  
 tail=node ---- O(1)  
 node.next=node.previous=null ---- O(1)  
  
Time Complexity O(1)  
Space Complexity O(1)

**Time Complexity: Double Linked List: Insertion**

* **Insert at Start**
* **Insert at last**
* **Insert at specified location**

insertLinkedList(head, nodeValue, location)  
 create a blank node -----O(1)  
 node.value = nodeValue -----O(1)  
  
 if(!existLinkedList(head)) -----O(1)  
 return error //Linked list not present -----O(1)  
 else if(location equlas 0)  
 //insert at first position -----O(1)  
 node.next = head -----O(1)  
 node.previous = null -----O(1)  
 head.previous = node -----O(1)  
 head = node -----O(1)  
 else if(location equals last)  
 //insert at last -----O(1)  
 node.next = null -----O(1)  
 node.previous = last -----O(1)  
 last.node = next -----O(1)  
 last = node -----O(1)  
 //to keep track of last node  
 else //insert at specific location -----O(1)  
 loop: tempNode = 0 to location -1 -----O(n)  
 //loop till we reach specified node and end the loop  
 node.next = tempNode.next -----O(1)  
 node.previous = tempNode -----O(1)  
 tempNode.next=node -----O(1)  
 node.next.previous = node ----O(1)  
  
Time complexity ----- O(n)  
Space complexity ----- O(1)

**Time Complexity: Double Linked List: Traverse**

traverseLinkedList(head)  
 if head == NULL -----O(1)  
 then return null -----O(1)  
 loop: haed to tail -----O(n)  
 print CurrentNode.value -----O(1)  
  
Time complexity -----O(n)  
Space complexity -----O(1)

**Time Complexity: Double Linked List: Reverse Traverse**

traverseLinkedList(head)  
 if head == NULL -----O(1)  
 then return null -----O(1)  
 loop: tail to head -----O(n)  
 print CurrentNode.value -----O(1)  
  
Time complexity -----O(n)  
Space complexity -----O(1)

**Time Complexity: Double Linked List: Searching**

searchNode(node, nodeValue)  
 loop: tempNode = head to tail ----O(n)  
 if(tempNode.value equals nodeValue) ----O(1)  
 print tempNode.value ----O(1)  
 return value found ----O(1)  
 return value not found ----O(1)  
   
Time complexity -----O(n)  
Space complexity -----O(1)

**Time Complexity: Double Linked List: Deletion of a node**

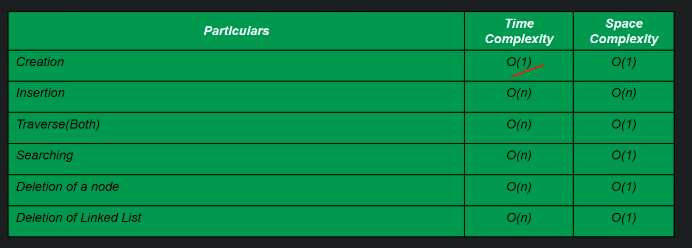
deleteOfNode(head, location)  
 if(!existLinkedList(head)) ----O(1)  
 return error ----O(1)  
 else if(location equals 0) ----O(1)  
 head = head.next ----O(1)  
 head.previous = null; ----O(1)  
 if this is the only element in list  
 then update head=tail=null; ----O(1)  
 else if(location >= last) ----O(1)  
 if (current node is only node in list)  
 then head=tail=null ----O(1)  
 tail = tail.previous; ----O(1)  
 tail.next = null; ----O(1)  
 else ----O(1)  
 loop.tmpNode=start to location-1 ----O(n)  
 tmpNode.next=tempNode.next.next ----O(1)  
 tmpNode.next.previous = tempNode ----O(1)

Time complexity -----O(n)  
Space complexity -----O(1)

**Time Complexity: Double Linked List: Deletion a linked list**

deleteLikedList(head, tail)  
 loop: temp head to tail ---- O(n)  
 temp.previous = null; ---- O(1)  
 head = tail = null; ---- O(1)  
  
Time complexity -----O(n)  
Space complexity -----O(1)

**Time Complexity of Circular Single Linked List array**



<https://github.com/bibhusprasad/011_data_structure/tree/master/src/s2_c2_linkedlist>

<https://github.com/bibhusprasad/011_data_structure/tree/master/src/s2_c2_node>

**2.2.4. Circular Double Linked List:**

**Time Complexity: Circular Double Linked List: Creation**

CreateDoubleLinkedList(nodeValue)  
 create a blank node ---- O(1)  
 node.value=nodeValue ---- O(1)  
 head=node ---- O(1)  
 tail=node ---- O(1)  
 node.next=node.previous=node ---- O(1)  
  
Time Complexity O(1)  
Space Complexity O(1)

**Time Complexity: Circular Double Linked List: Insertion**

* **Insert at Start**
* **Insert at last**
* **Insert at specified location**

insertLinkedList(head, nodeValue, location)  
 create a blank node -----O(1)  
 node.value = nodeValue -----O(1)  
  
 if(!existLinkedList(head)) -----O(1)  
 return error //Linked list not present -----O(1)  
 else if(location equlas 0)  
 //insert at first position -----O(1)  
 node.next = head -----O(1)

node.previous = tail -----O(1)   
 head.previous = node -----O(1)  
 head = node -----O(1)

tail.next = node -----O(1)  
 else if(location equals last)  
 //insert at last -----O(1)  
 node.next = head -----O(1)  
 node.previous = tail -----O(1)  
 tail.next = node -----O(1)  
 tail = node -----O(1)  
 //to keep track of last node  
 else //insert at specific location -----O(1)  
 loop: tempNode = 0 to location -1 -----O(n)  
 //loop till we reach specified node and end the loop  
 node.next = tempNode.next -----O(1)  
 node.previous = tempNode -----O(1)  
 tempNode.next=node -----O(1)  
 node.next.previous = node ----O(1)  
  
Time complexity ----- O(n)  
Space complexity ----- O(1)

**Time Complexity: Circular Double Linked List: Traverse**

traverseLinkedList(head)  
 if head == NULL -----O(1)  
 then return null -----O(1)  
 loop: haed to tail -----O(n)  
 print CurrentNode.value -----O(1)  
  
Time complexity -----O(n)  
Space complexity -----O(1)

**Time Complexity: Circular Double Linked List: Reverse Traverse**

traverseLinkedList(head)  
 if head == NULL -----O(1)  
 then return null -----O(1)  
 loop: tail to head -----O(n)  
 print CurrentNode.value -----O(1)  
  
Time complexity -----O(n)  
Space complexity -----O(1)

**Time Complexity: Double Linked List: Searching**

searchNode(node, nodeValue)  
 loop: tempNode = start to tail ----O(n)  
 if(tempNode.value equals nodeValue) ----O(1)  
 print tempNode.value ----O(1)  
 return value found ----O(1)  
 return value not found ----O(1)  
   
Time complexity -----O(n)  
Space complexity -----O(1)

**Time Complexity: Double Linked List: Deletion of a node**

deleteOfNode(head, location)  
 if(!existLinkedList(head)) ----O(1)  
 return error ----O(1)  
 else if(location equals 0) ----O(1)  
 head = head.next ----O(1)  
 head.previous = null; ----O(1)

tail.next = head; ----O(1)  
 if this is the only element in list  
 then update head.next=head.previous=head=tail=null; ----O(1)  
 else if(location >= last) ----O(1)  
 if (current node is only node in list)  
 then head.next=head.previous head=tail=null ----O(1)  
 tail = tail.previous; ----O(1)

tail.next = head; ----O(1)  
 head.previous=tail; ----O(1)  
 else ----O(1)  
 loop.tmpNode=start to location-1 ----O(n)  
 tmpNode.next=tempNode.next.next ----O(1)  
 tmpNode.next.previous = tempNode ----O(1)

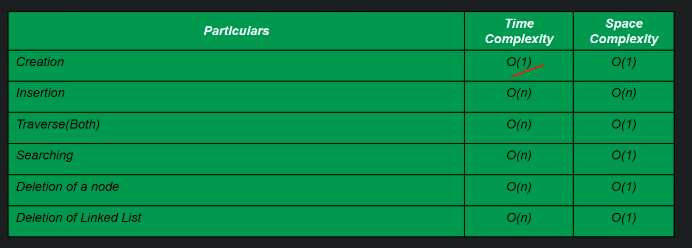
Time complexity -----O(n)  
Space complexity -----O(1)

**Time Complexity: Double Linked List: Deletion a linked list**

deleteLikedList(head, tail)

tail.next = null ---- O(n)  
 loop: temp head to tail ---- O(n)  
 temp.previous = null; ---- O(1)  
 head = tail = null; ---- O(1)  
  
Time complexity -----O(n)  
Space complexity -----O(1)

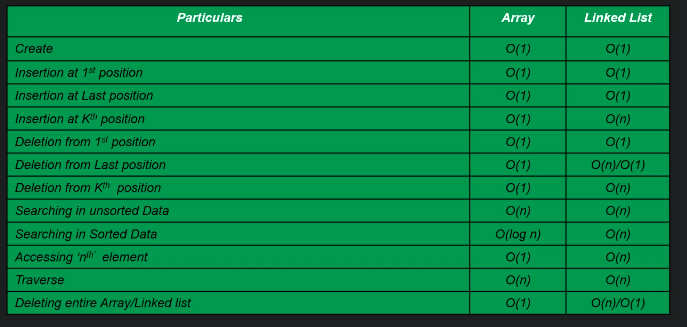
**Time Complexity of Circular Single Linked List array**



<https://github.com/bibhusprasad/011_data_structure/tree/master/src/s2_c2_linkedlist>

<https://github.com/bibhusprasad/011_data_structure/tree/master/src/s2_c2_node>

**Different between Array and Linked List**



**Practical Use of linked list**

Alt tab button in windows and windows photo viewer

**3. Logical Data Structure**

**3.1. Stack**

* Insertion/deletion follows LIFO (Last In First Out)
* When we need to create last incoming data first algo
* E.g. browser back button.
* Common operations in stack are
  + createStack()
  + push()
  + pop()
  + peek()
  + isEmpty()
  + isFull()
  + deleteStack()
* We can implement stack in two ways using array and using linked list

**Implement Stack using Array**

Props: Easy to implement

Cons: Fixed in size

**Time Complexity: createStack()**

createStack(int size)  
 create black array od 'size' ----O(1)  
 Initialize variable 'topStack' to -1 ----O(1)  
   
Time complexity : O(1)  
Space complexity : O(1)

**Time Complexity: push()**

pushOperation(value)  
 if stack is full ----O(1)  
 return error; ----O(1)  
 else ----O(1)  
 insert value at top of the array ----O(1)  
 update topStack++; ----O(1)  
   
Time complexity : O(1)  
Space complexity : O(1)

**Time Complexity: pop()**

popOperation(value)  
 if stack is empty ----O(1)  
 return error; ----O(1)  
 else ----O(1)  
 print top of the stack ----O(1)  
 update topStack--; ----O(1)  
  
Time complexity : O(1)  
Space complexity : O(1)

**Time Complexity: peek()**

peekOperation(value)  
 if stack is empty ----O(1)  
 return error; ----O(1)  
 else ----O(1)  
 print top of the stack ----O(1)  
  
Time complexity : O(1)  
Space complexity : O(1)

**Time Complexity: isEmpty()**

isEmptyOperation()  
 if topStack = -1 ----O(1)  
 return true; ----O(1)  
 else ----O(1)  
 return false ----O(1)  
  
Time complexity : O(1)  
Space complexity : O(1)

**Time Complexity: isFull()**

isFullOperation()  
 if topStack equals array.size ----O(1)  
 return true; ----O(1)  
 else ----O(1)  
 return false ----O(1)  
  
Time complexity : O(1)  
Space complexity : O(1)

**Time Complexity: delete()**

deleteOperation()  
 arr = null ----O(1)  
  
Time complexity : O(1)  
Space complexity : O(1)

**Implement Stack using Linked List**

Props: Variable in size

Cons: Moderate in implementation

**Time Complexity: createStack()**

createStack()  
 create an object of SingleLinkedList class ----O(1)  
  
Time complexity : O(1)  
Space complexity : O(1)

**Time Complexity: push()**

pushOperation(value)  
 create a node ----O(1)  
 node.value = value; ----O(1)  
 node.next = head; ----O(1)  
 head = node; ----O(1)  
  
Time complexity : O(1)  
Space complexity : O(1)

**Time Complexity: pop()**

popOperation()  
 if isEmpty() ----O(1)  
 return error; ----O(1)  
 else  
 tmpNode = head ----O(1)  
 head = head.next ----O(1)  
 return tmpNOde.value ----O(1)  
  
Time complexity : O(1)  
Space complexity : O(1)

**Time Complexity: peek()**

peekOperation()  
 return header.value ----O(1)  
  
Time complexity : O(1)  
Space complexity : O(1)

**Time Complexity: delete()**

deleteOperation()  
 head = null ----O(1)  
  
Time complexity : O(1)  
Space complexity : O(1)

**When to use and when to avoid**

LIFO principle, cannot easily corrupted, random access be not possible

<https://github.com/bibhusprasad/011_data_structure/tree/master/src/s3_c1_stack>

**3.2. Queue**