# Linked lists



Linked list is data structure with following properties::

- Successive elements are connected by pointers
- Last element points to NULL
- Can grow or shrink in size
- Does not waste memory as in arrays when number of elements are not unknown

# Why linked list?

Comparing it with arrays::

Arrays==>> pre-allocation
fixed size
complex position based insertion

\+ point - random access - O(1)

Linked lists - can be expanded in constant time
need based memory allocation
\- point => access time is O(n)
\- point => pointer overheads & pointer arithmetic

# Comparison of array and linked list

| Parameter | Array | Linked List |
|---|---|---|
| Indexing (Selecting specific) | O(1) | O(n) |
| Insertion/Deletion in the beginning | O(n) | O(1) |
| Insertion at Ending | O(1) | O(n) |
| Deletion at Ending | O(1) | O(n) |
| Insertion in middle | O(n) | O(n) |
| Deletion in middle | O(n) | O(n) |

================================================================

## Singly Linked Lists

List contains elements made of data component & pointer.
Pointer points to next element, if any, else, points to NULL

Elements are called as nodes.

E.g.,

```
struct ListNode {

     double weight;
     struct ListNode *next;
};
```

Basic Operations
----------------



Traversing the list (similar to iterating through the List ADT)
Inserting an item in the list
Deleting  an item from the list

Traversing
----------

```
int ListLength(struct ListNode *head) {
    struct ListNode *current = head;
  int count = 0;

  while(current) {
   count++;
   current = current->next;
  }

  return count;
}
```

Time complexity ?  Space Complexity?

O(n)
O(1)

Inserting a node
----------------

Different situations::
1) insertion at the beginning of the list
2) insertion at the end of the list
3) insertion in between two nodes

(1)
head is pointing to, say, element e.
say new element is n.
Set n's pointer to point to e.
Set head to point to n.

(2)
last element l is pointing to NULL
new element n's pointer too point to NULL
using head, traverse the list, access the last element l
Set l's pointer to point to new element n

(3)
say, we want to add new element n at some existing node, say x.
first traverse the list and reach upto element y which is just before x
(i.e.,y's next pointer is pointing to x)
Set new element n's next pointer to point to y (using the value of y's next pointer)
Set y's next pointer to point to n

Deleting a node
---------------

Different situations::
1) Deletion of the first element

2) Deletion of the last element
3) Deletion of element which is in between two elements of the list



(1)

Create a temporary node which points to the same element as that of head

Set head's next pointer to set to its next element

Set free the temporary variable

(2)

Traverse the list and while doing it so keep track of last but one element too

So we should have two pointers at this junction with us, one pointing to the tail and other pointing to the previous element just before the tail element

Set previous node's next pointer to NULL

Destroy the tail node (call to free)

(3)

Traverse the list and while doing it so keep track of element which is to be deleted (say current) and as well as its previous element

So we should have two pointers at this junction with us,
one pointing to the element to be deleted (current) and
other pointing to its previous element

Set previous node's next pointer to the next pointer of the element to be deleted

Destroy the current element

Lab Assignment::

Write a program to create a singly linked list of integers.

Write a program to delete (destroy) singly linked list of integers.

```
define it the iterator, point to first element using head
  while(it) {
 an = it->next;
 free(it);
 it = an;
}
*head = NULL;
```

===========================================================================

o Doubly Linked Lists

Explain concept using next and previous element
Explain operations of insertion and deletion.

Key difference with singly linked list provided to the students.

===========================================================================

o Circular Linked Lists

o Node-based storage with arrays

Refer to Presentation