Preliminary notes

Array-contiguous area of memory consisting of equal sized elements

Arrays allow constant time access (we can do arithmetic to figure out the address of all elements)-array_adress+element size*(i-first_index)

data structures are mathematical/logical models or abstract data types

ADTS define data and operations but no implementation

- The ADT list is simply a container of items whose order you indicate and whose position you reference by number.
- You reference list items by their position.

List as an abstract data type

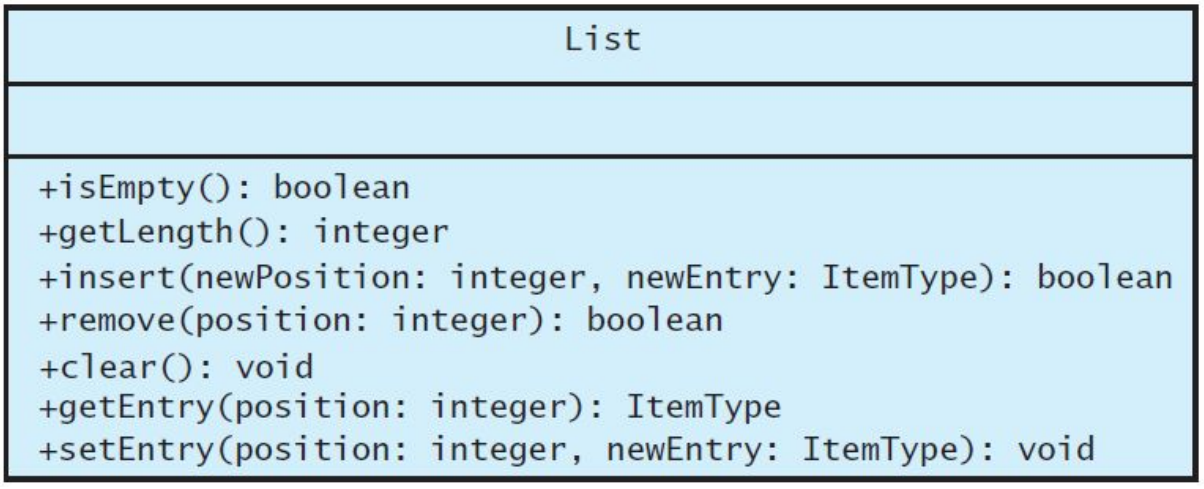
- should store a given number of elements for a given data-type
- should be able to write/modify these elements
- read the element at a position

arrays are a data structure that implements and ADT and one of the first learned in c++

ADT list of operations

• Test whether a list is empty.
• Get the number of entries on a list.
• Insert an entry at a given position on the list.
• Remove the entry at a given position from the list.
• Remove all entries from the list.
• Look at (get) the entry at a given position on the list.
• Replace (set) the entry at a given position on the list.

UML diagram for ADT list



Abstract Data Type : List	
Data	
A finite number of objects, not necessarily distinct, having the same data type and ordered by their positions, as determined by the client.	
Operations	
Pseudocode	Description
isEmpty()	Task: Sees whether this list is empty. Input: None. Output: True if the list is empty; otherwise false.
getLength()	Task: Gets the current number of entries in this list. Input: None. Output: The integer number of entries currently in the list.
insert(newPosition, newEntry)	Task: Inserts an entry into this list at a given position. An insertion before existing entries causes the renumbering of entries that follow the new one. Input: newPosition is an integer indicating the position of the insertion, and newEntry is the new entry. Output: True if 1 □ newPosition □ getLength() + 1 and the insertion is successful; otherwise false.
remove(position)	Task: Removes the entry at a given position from this list. A removal before the last entry causes the renumbering of entries that follow the deleted one. Input: position is the position of the entry to remove. Output: True if 1 □ newPosition □ getLength() and the removal is successful; otherwise false.
clear()	Task: Removes all entries from this list. Input: None. Output: None. The list is empty.
getEntry(position)	Task: Gets the entry at the given position in this list. Input: position is the position of the entry to get; 1 <= position <= getLength() . Output: The desired entry.
setEntry(position, newEntry)	Task: Replaces the entry at the given position in this list. Input: position is the position of the entry to replace; 1 <= position <= getLength() . newEntry is the replacement entry. Output: None. The indicated entry is replaced.

The list operations fall into the three broad categories presented earlier in this book:

• The operation insert adds data to a data collection.

• The operation remove removes data from a data collection.

• The operations isEmpty, getLength , and getEntry ask questions about the data in a data collection.

The operation setEntry replaces existing data in a data collection, so you can think of it as removing and then adding data.

Display items on list-not an ADT operation

```
// Displays the items on the list aList.
displayList(aList)
      for (position = 1 through aList.getLength())
            {
                  dataItem = aList.getEntry(position)
                  Display dataItem
            }
```

Replacing an item in the absence of setEntry

```
// Replaces the i th entry in the list aList with newEntry.
// Returns true if the replacement was successful; otherwise return
false.
replace(aList, i, newEntry)
      success = aList.remove(i)
      if (success)
            success = aList.insert(i, newItem)
      return success
```

Array-based implementation of ADT

- By defining the data member maxItems for ArrayList , we enable the class implementer to allocate the array items dynamically instead of statically. Thus, the programmer could easily define a constructor that allows the client to choose the size of the array.

Access -read/write element at an index O(1)
Insert-proportional to length of list so O(n)
Remove-proportional to size of list O(n)
Append-time proportional to size of list O(n)

If a list grows and shrinks a lot this will require quite a bit of resources because we will have to go through the process of re-copying everything to the newly sized array

If portions of the array are not used, the memory will be non-eficiently used

There is cost when removing elements from the core of an array since all the elements need to be shift to left to close the gap left.

Linked Lists

- storing in non-contiguous memory blocks
- to link blocks together we use pointers, each node stores the actual data we need plus an address to the next node.

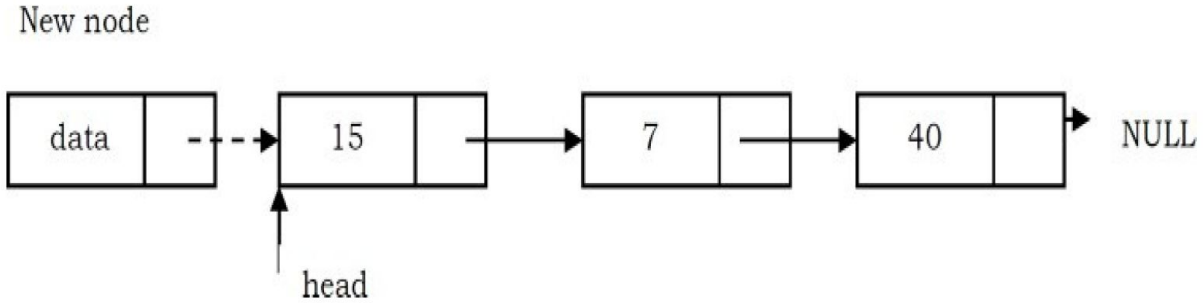
Example:

```
Struct Node
{
        int data; //4bytes
        Node* next; //4bytes
}
```

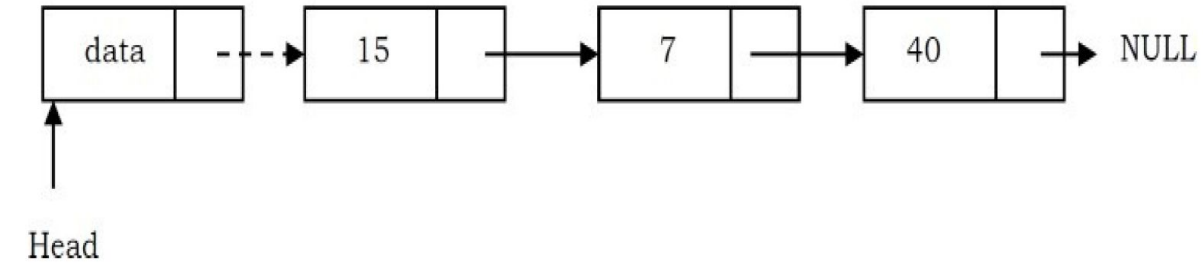
- the first node is called head, the address of first node gives access to the entire list
- to traverse a linked list we start at the head and keep using the pointers to move through
- to insert a node at end, first we create an independent node, after that we fill the address of the previous node (that was pointing to null) to point to the new node and then update the new node address to point to null
- to insert at the middle of the list, we create the node, then we update the address of the previous node to point to the new node, and we make the new node point to the node after itself.
- access to linked list is not constant time (we need to go from head to tail, jumping through the addresses, it is proportional to the size of the list

Insert at beginning:

- only one pointer needs to be changed (new node's next pointer)
- update next pointer of new node to point to current head

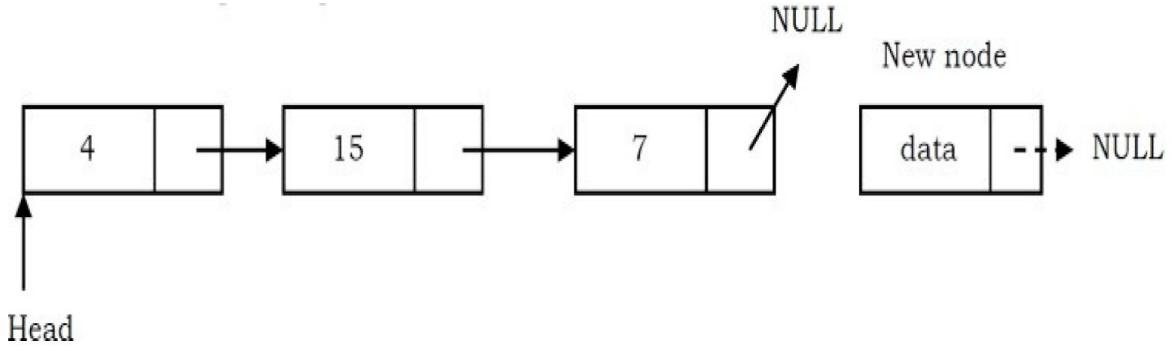


- update head pointer to point to new node

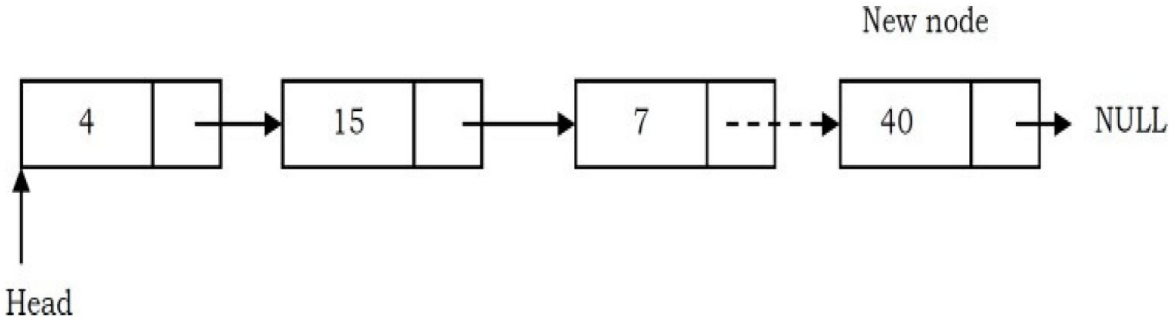


Insert at end:

- need to modify 2 pointers (the pointer of the last node and the pointer of the new node
- set new node's pointer to NULL

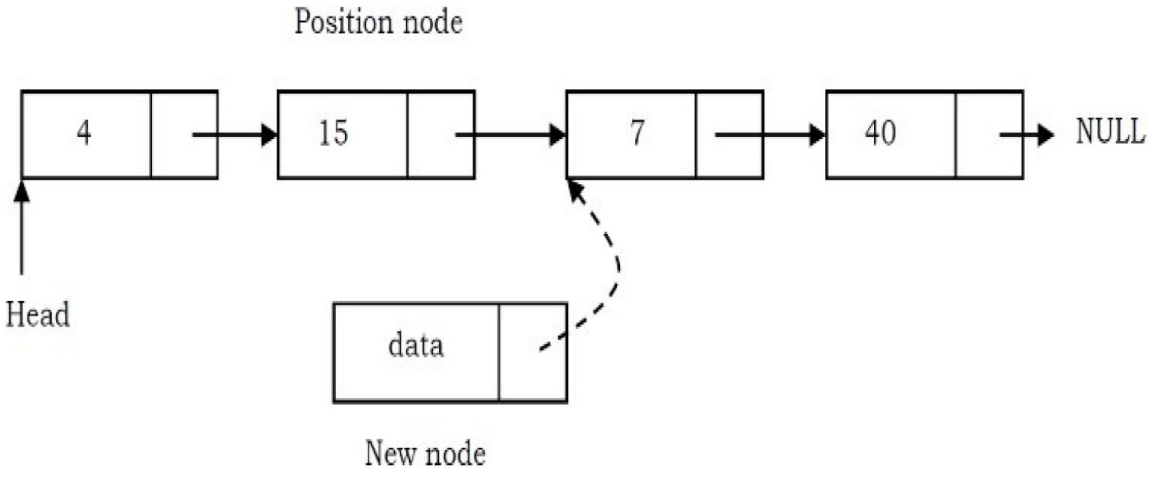


- set old last node's pointer to new node

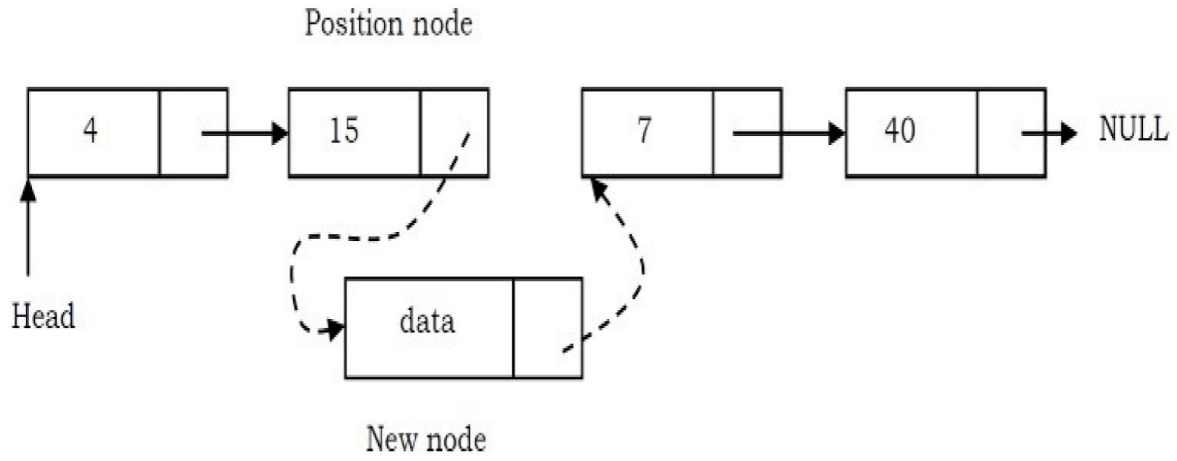


Insert at middle:

- If we want to add an element at position 3 then we stop at position 2. That means we traverse 2 nodes and insert the new node. For simplicity let us assume that the second node is called position node. The new node points to the next node of the position where we want to add this node.

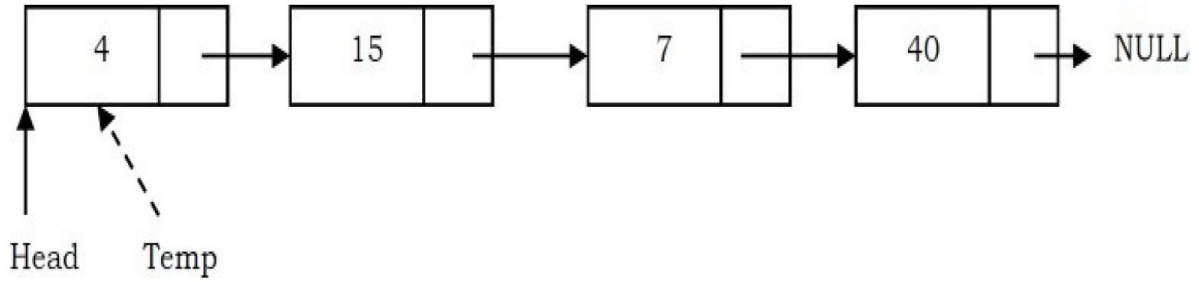


- Position node's next pointer now points to the new node.

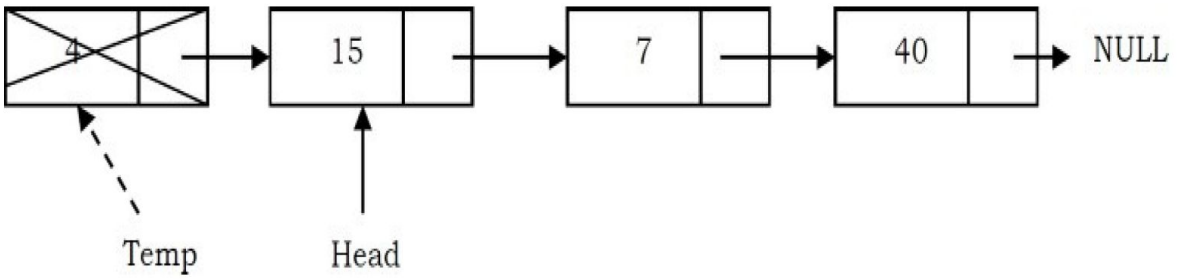


Delete the first

- Create a temporary node which will point to the same node as that of head.

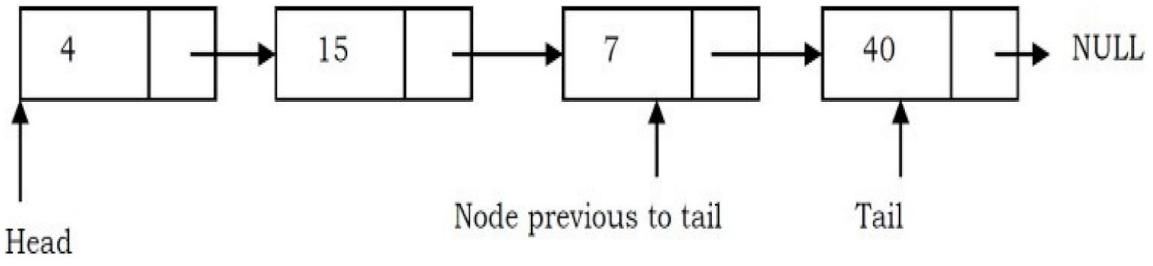


- move the head nodes pointer to the next node and dispose of the temporary node.

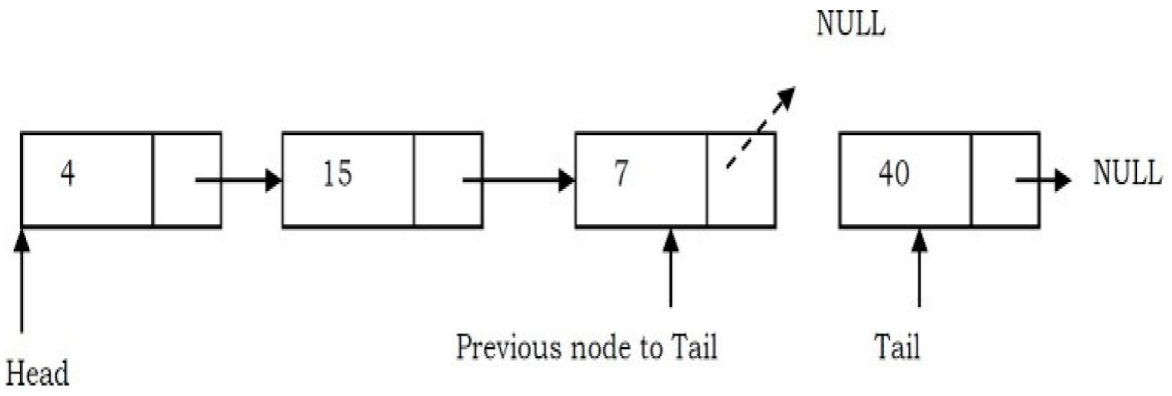


Deleting the Last Node

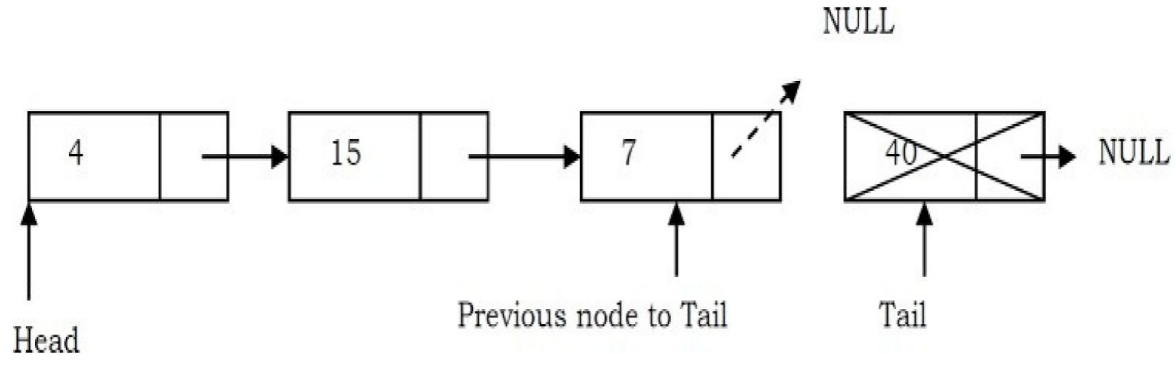
- Traverse the list and while traversing maintain the previous node address also. By the time we reach the end of the list, we will have two pointers, one pointing to the tail node and the other pointing to the node before the tail node.



- Update previous node's next pointer with NULL.

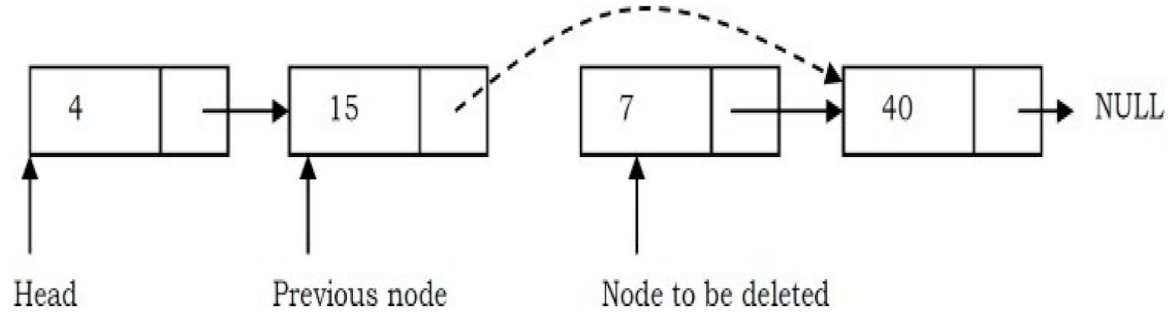


- Dispose of the tail node.

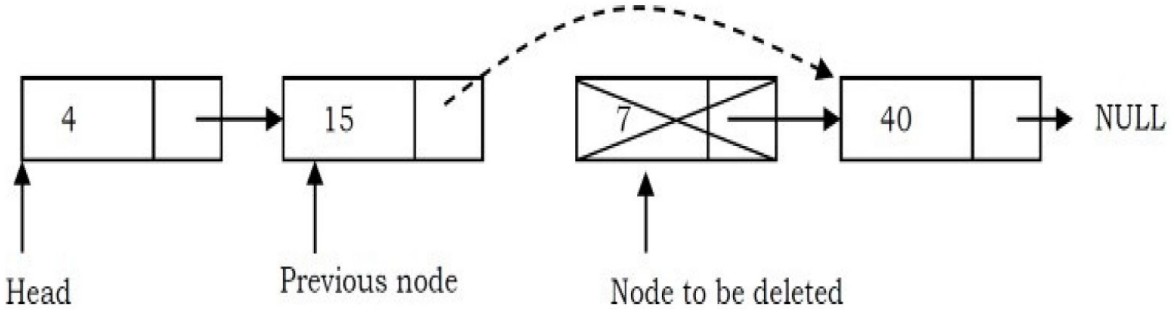


Deleting an Intermediate Node

- Head and tail links
  are not updated in this case. Such a removal can be done in two steps:
  Similar to the previous case, maintain the previous node while traversing the list.
  Once we find the node to be deleted, change the previous node's next pointer to the next pointer of the node to be deleted.



- Dispose of the current node to be deleted.



Array vs Linked list

	Advantages	Disadvantages
Arrays	Simple and easy to use	Preallocates all needed memory up front and wastes memory space for indices in the array that are empty.
	Faster access to the elements (constant access)	Fixed size: The size of the array is static (specify the array size before using it).
	O(1) to access any element in the array	One block allocation: To allocate the array itself at the beginning, sometimes it may not be possible to get the memory for the complete array (if the array size is big).
	greatly benefits from modern CPU caching methods	Complex position-based insertion: To insert an element at a given position, we may need to shift the existing elements. This will create a position for us to insert the new element at the desired position. If the position at which we want to add an element is at the beginning, then the shifting operation is more expensive.
Linked Lists	can be expanded in constant time	O(n) for access to an element in the list in the worst case
		If the last item is deleted, the last but one must then have its pointer changed to hold a NULL reference, this requires list traversal
		waste memory in terms of extra reference points

Parameter	Linked list	Array	Dynamic Array
Indexing	O(n)	O(1)	O(1)
Insert/delete beginning	O(1)	O(n) if array not full to shift elements	O(n)
Insert end	O(n)	O(1) if array not full	O(1) array not full O(n) array is full
Delete end	O(n)	O(1)	O(n)
Insert middle	O(n)	O(n) if array not full, no need to shift elements	O(n)
Delete middle	O(n)	O(n) if array not full, no need to shift	O(n)

		elements	
Extra space	O(n) pointers	0	O(n)