

# Lecture # 3

## Linked List

- **Link List** is a **Data Structure** in which elements are explicitly ordered, that is each item contains within itself the address of next item.
- The **array** implementation has the serious drawback and that is we must specify size at the **construction time** though it is **simple** and **fast**.
- Murphy's Law:
  - Construct an array with space for  $n = \text{twice your estimate of largest collection}$
  - Tomorrow you will need  $n+1$

# Linked List

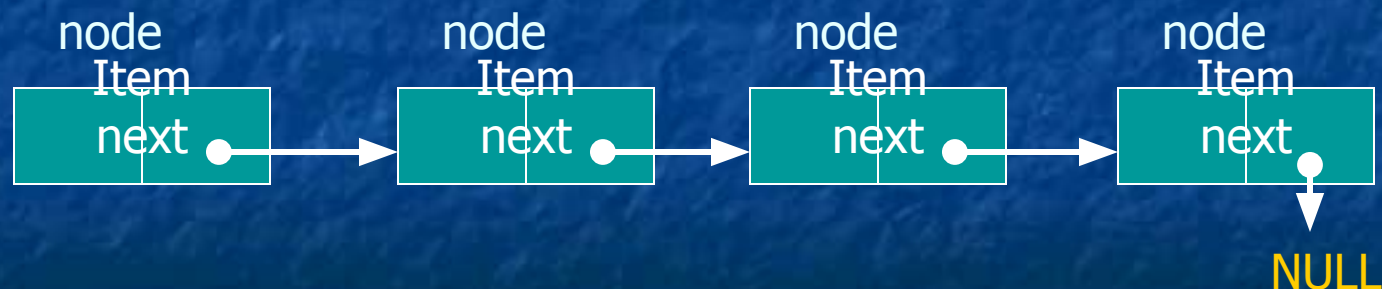
- Create a structure called a *Node*.



- The object field will hold the actual *list* element.
- The next field in the structure will hold the *starting location* of the *next* node.
- Chain the nodes together to form a *linked* list.



- A very common source of problems in program maintenance is the need to increase the capacity of a program to handle larger collections.
- In a **linked list**, each **item** is allocated space as it is added to the list. A **link** is kept with each item to the **next item** in the list.



- Each node of the list has **two** elements:
  - The item being stored in the list and
  - A pointer to the next item in the list
- The last node in the list contains a **NULL** pointer to indicate that it is the **end** or **tail** of the list.
- As items are added to a list, memory for a node is **dynamically allocated**. Thus the number of items that may be added to a list is limited only by the amount of memory available.
- **Handle of the LL** : The variable ( or handle) which represents the list, is simply a pointer to the node at the **head** of the list.

# Linked List

- Picture of our list (2, 6, 7, 8, 1) stored as a linked list:





```
struct node
{
    char name[15];
    node *next;
};
```

# Linked List

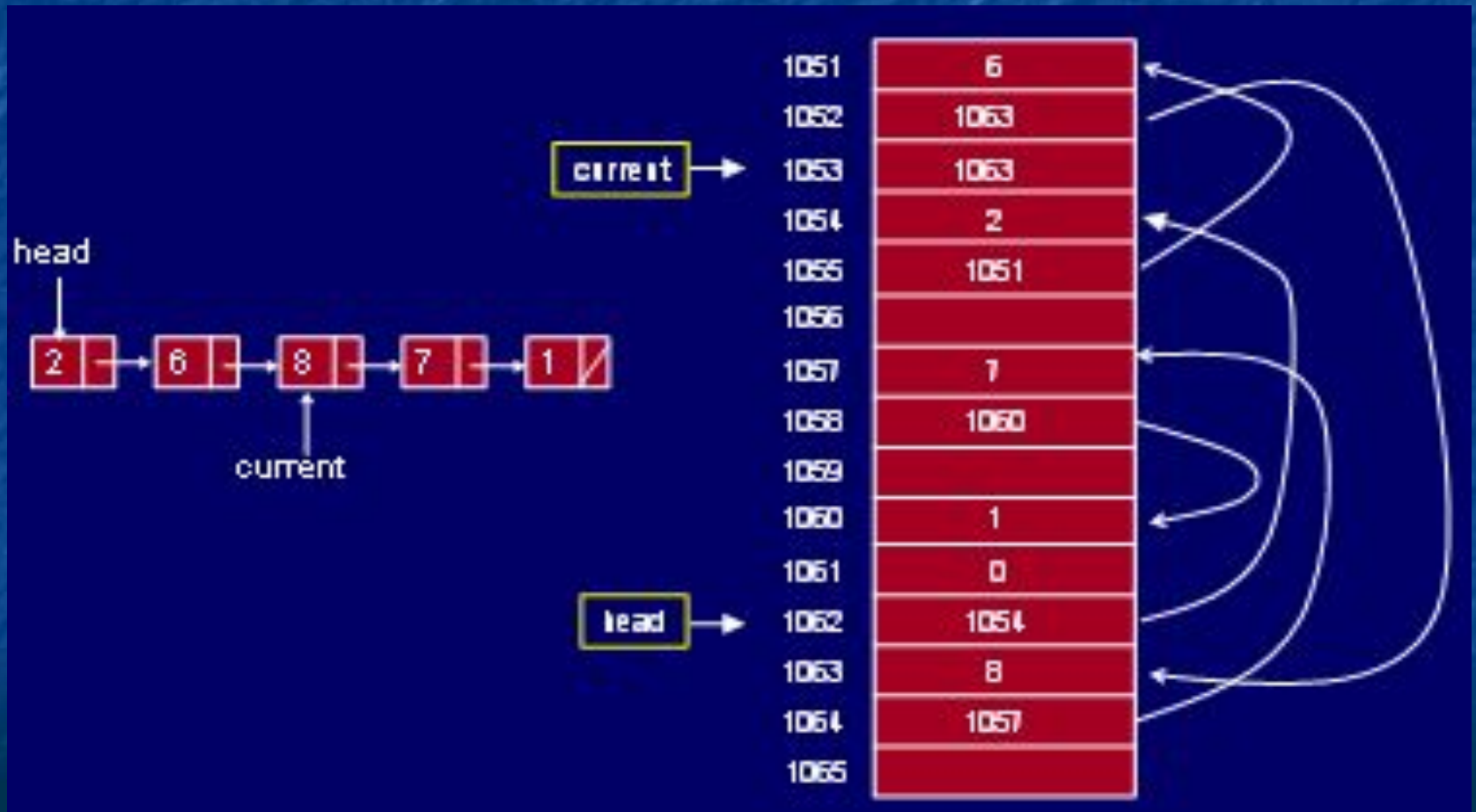
Note some features of the list:

- Need a *head* to point to the first node of the list. Otherwise we won't know where the start of the list is.
- The *current* here is a pointer, not an index.
- The *next* field in the last node points to *nothing*. We will place the memory address NULL which is guaranteed to be inaccessible.



# Linked List

## Actual picture in memory:

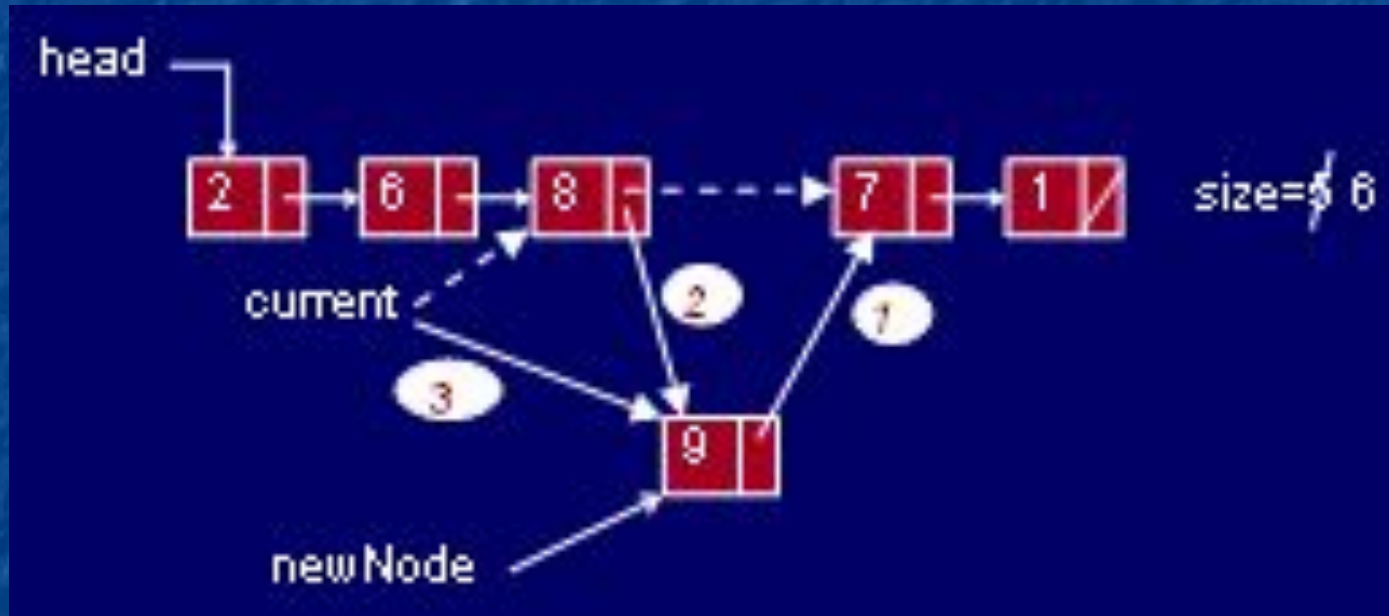


# Operations of Link List

- Adding (Inserting) to a Link List (LL) : The simplest strategy for adding an item to a list is to:
  1. Allocate **space** for a **new** node.
  2. **Copy** the **item value** into it.
  3. **Make** the **new node's next pointer** point to the current **head** of the **list** and
  4. Make the **head** of the list point to the newly allocated node.

This strategy is fast and efficient but each item is added to the head of the list.

# Adding element into link list





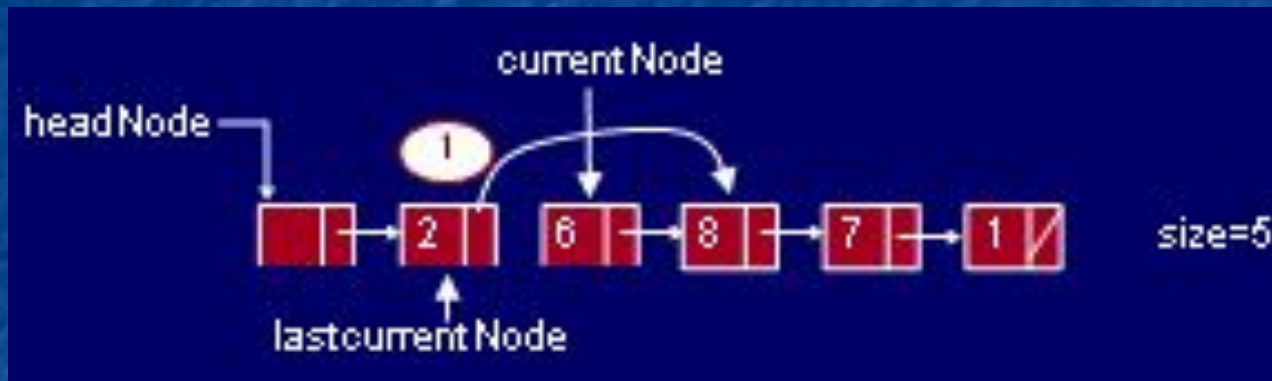
- Searching a Link List :

- To search a list of  $n$  objects or elements the search function takes  $n$  operations to search a data item in the LL in **worst case**, since it may have to search the **entire link list**.

## ■ Deletion from Link List :

- This removes a data item's node (block) from the start of link list. Pointer must be pointing to that node which has to be deleted and then pointer will have to be **updated**.
- But if we wish to delete the particular node (block) with given key to match, then we have to search for that node (block) in the entire Link List.

# removing element from link list





# Implementation of Link List

- Discussion on *White Board* from *C++* point of view
  - Searching
  - Addition
  - Deletion etc ....

# Assignment # 2

- Implement a Linked List (LL) discussed above including following operations.
  - Insertion of node at any place in LL.
  - Deletion of node at any place in LL
  - Searching for a node at any place in LL
  - Printing whole Linked List (LL)
  - Getting a value at a specific position in LL
  - Length of the LL
  - Checking emptiness of the LL
- Due Date: next week same day (In Office Hours)