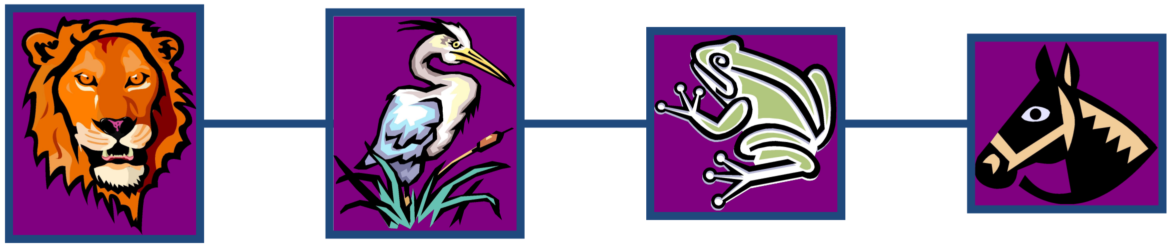
# Linked List

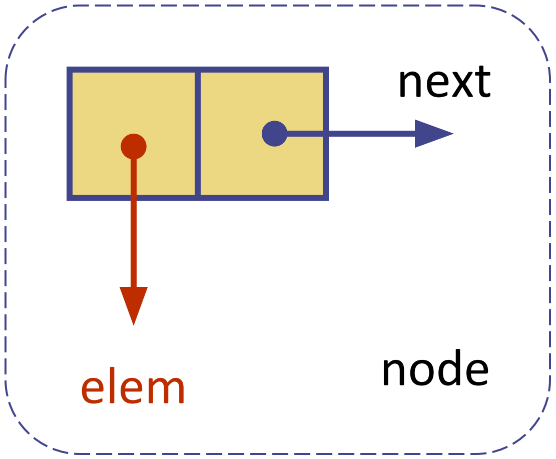
(One-way Linked List)



## Linked List

- A linear collection of data elements (linear data structure).
- Each element is represented by a node.
- Each node is divided into two parts.

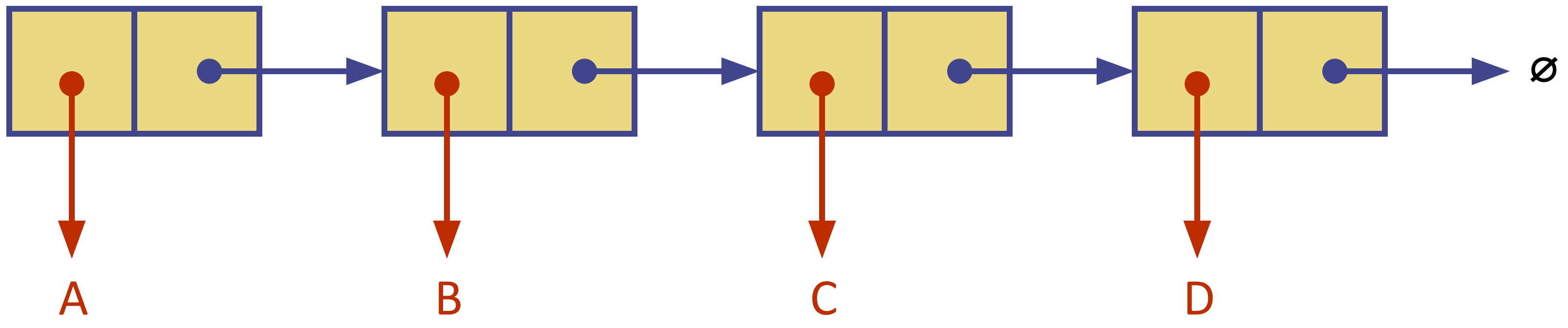
1. Information part contains the information of element.
2. Link part contains the address of the next node in the link.



## Linked List

- A linear collection of data elements (linear data structure).
- Each element is represented by a node.
- Each node is divided into two parts.

  1. Information part contains the information of element.
  2. Link part contains the address of the next node in the link.

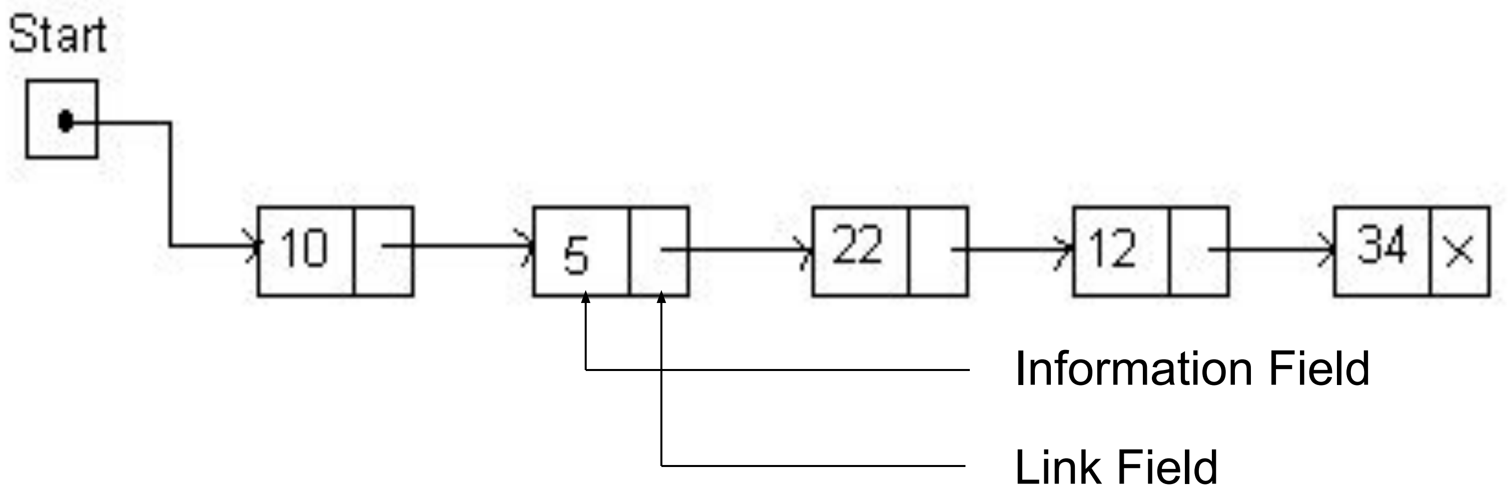


## Linked List

- Linked list has a list pointer variable, **Head** or **Start**, containing the address of the 1st node in the list.
- If Start contains NULL value, it means the list is empty.
- The address field of last node in the list contains NULL representing invalid address.

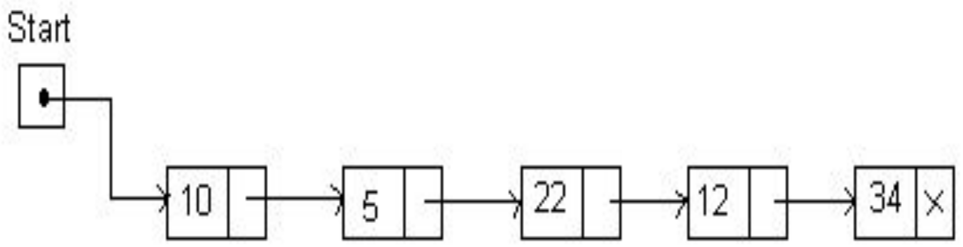
- Example:

The following figure shows a linked list having 5 nodes.



## Memory Representation of Linked List

- A linked list can be maintained in memory using two linear arrays.
    1. Info
    2. Link
- Info[K] represents the information part of a node in the list.
- Link[K] represents the nex tpointer field of a node in the list.
- The beginning of the list is denoted by the variable Start.



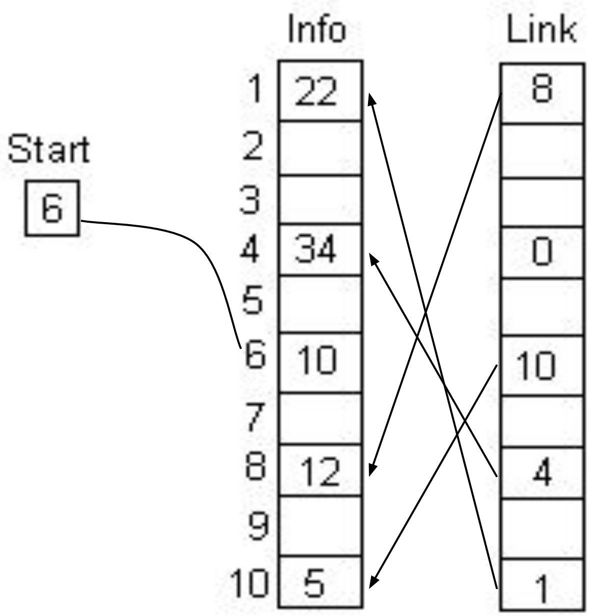
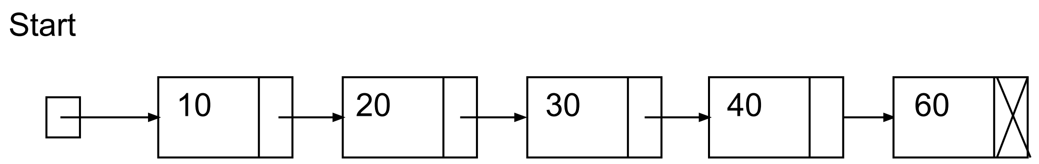


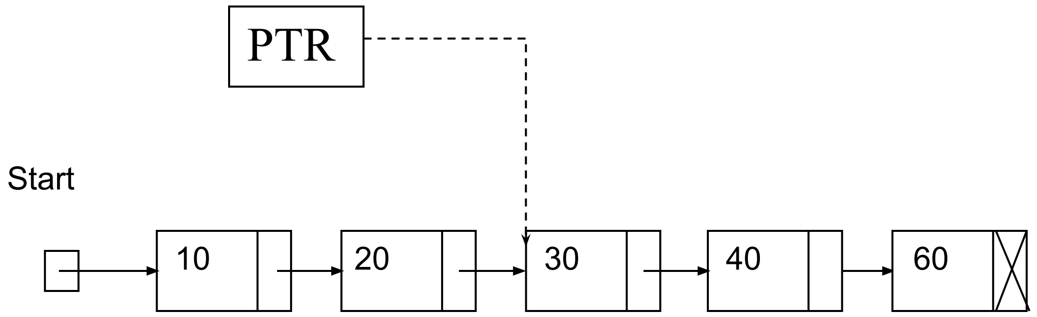
Figure: Linked list & its memory representation

# Some Notation



- LIST- name of the linked list.
- START – points to the first node of the linked list
- PTR- points to the node that is currently being processed
- INFO[PTR] – value of the current node.
- PTR:=LINK[PTR] – moves the pointer to next node (Update pointer)

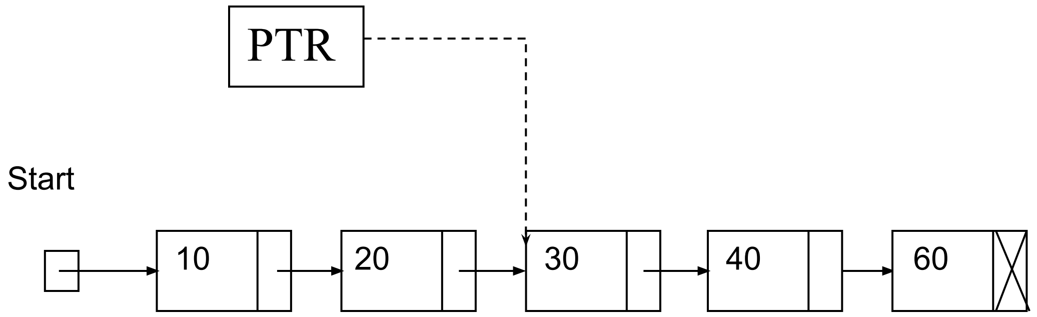
# Some Notation



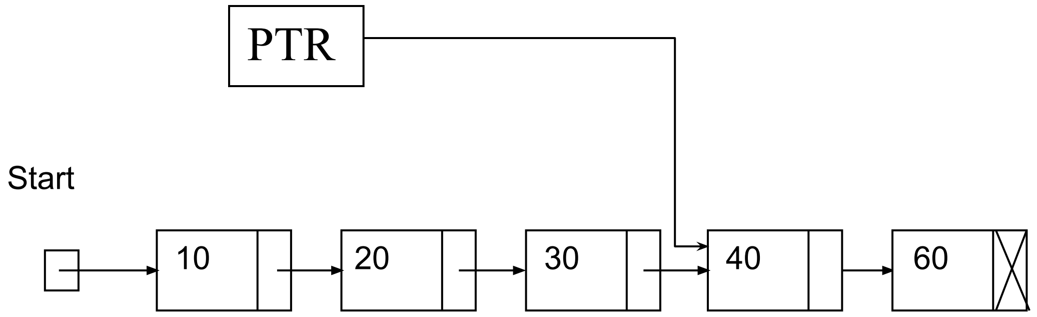
- INFO [PTR] – 30

# Some Notation

- PTR:=LINK[PTR] – moves the pointer to next node (Update pointer)



- After performing PTR:=LINK[PTR] –Update pointer)



# Singly-linked lists vs. 1D-arrays

| **ID-array** | **Singly-linked list** |
|---|---|
| Fixed size: Resizing is expensive | Dynamic size |
| Insertions and Deletions are inefficient: Elements are usually shifted | Insertions and Deletions are efficient: No shifting |
| Random access i.e., efficient indexing | No random access<br>□ Not suitable for operations requiring accessing elements by index such as sorting |
| No memory waste if the array is full or almost full; otherwise may result in much memory waste. | Extra storage needed for references; however uses exactly as much memory as it needs |
| Sequential access is faster because of greater locality of references [Reason: Elements in contiguous memory locations] | Sequential access is slow because of low locality of references [Reason: Elements not in contiguous memory locations] |

## Linked Lists versus arrays

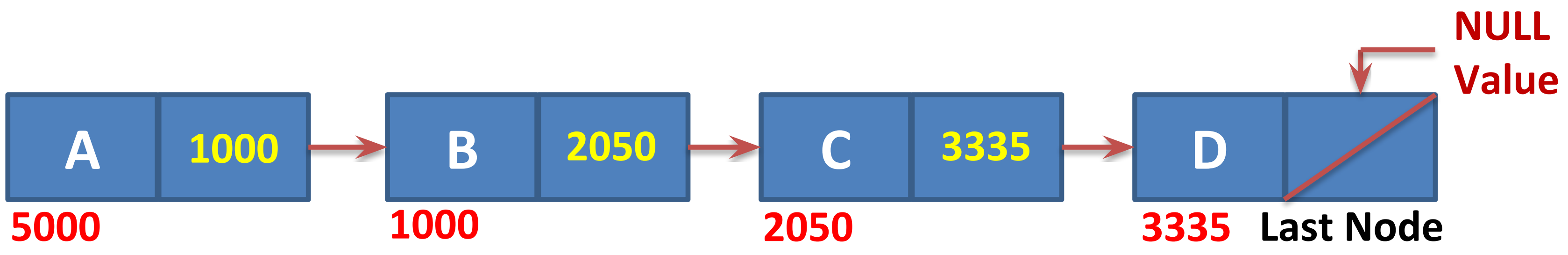
Linked lists have several advantages over arrays.

- Elements can be inserted into linked lists indefinitely, while an array will eventually either fill up or need to be resized, an expensive operation that may not even be possible if memory is fragmented.
- An array from which many elements are removed may become wastefully empty or need to be made smaller.

## Linked lists have several disadvantages over arrays.

- Arrays allow random access, while linked lists allow only sequential access to the elements.
- Singly-linked lists can only be traversed in one direction. This makes linked lists unsuitable for applications where it is useful to look up an element by its index quickly, such as heapsort.
- Another disadvantage of linked lists is the extra storage needed for references.

# Linked Storage Representation

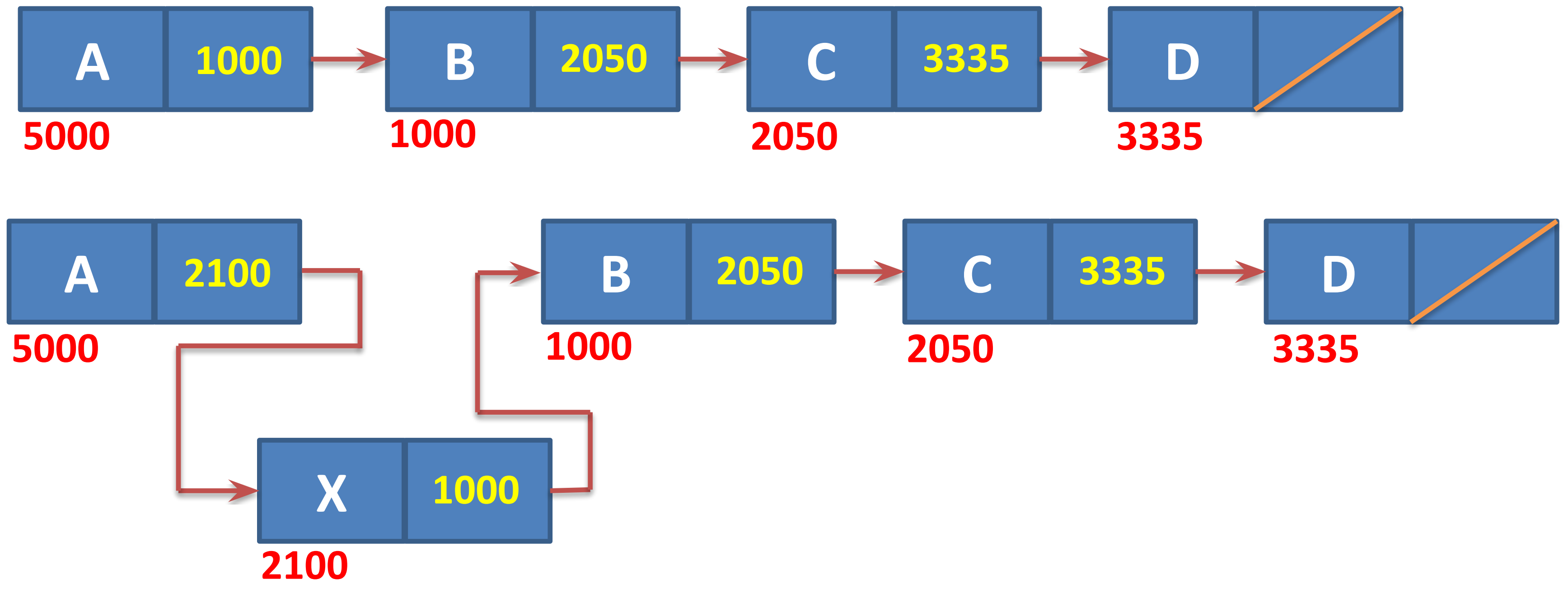


**A linked List**

- The linked allocation method of storage can result in both efficient use of computer storage and computer time.
  - A linked list is a **non-sequential collection** of data items.
  - Each **node** is **divided** into **two parts**, the **first part** represents the **information** of the element and the **second part** contains the **address of the next mode**.
  - The **last node** of the list does not have successor node, so **null value** is stored as the address.
  - It is possible for a list to have no nodes at all, such a list is called empty list.

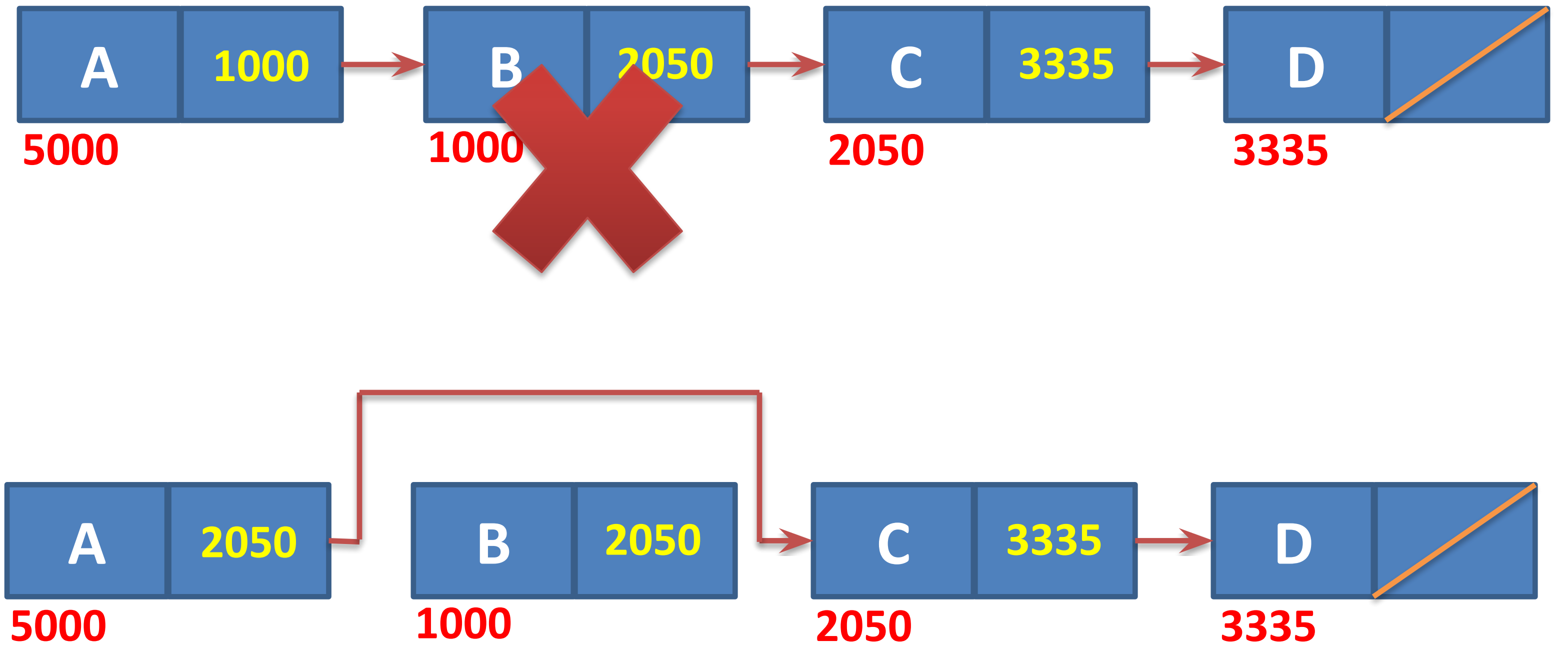
# Pros & Cons of Linked Allocation

- **Insertion Operation**
  - we have an n elements in list and it is required to insert a new element between the first and second element, what to do with sequential allocation & linked allocation?
  - Insertion operation is more efficient in Linked allocation.



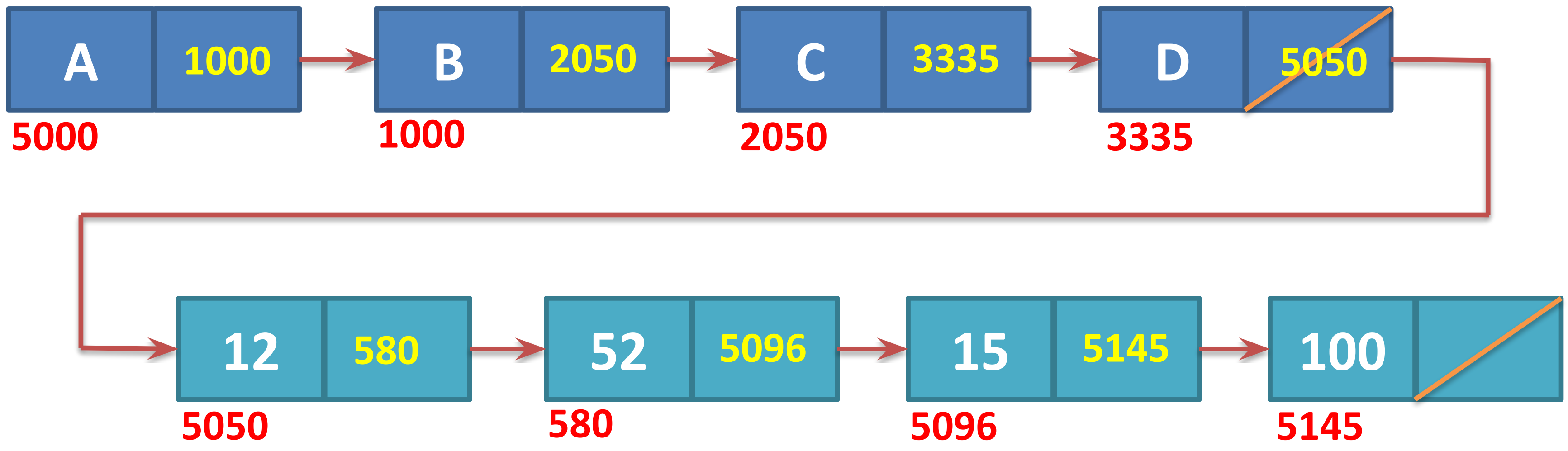
# Pros & Cons of Linked Allocation

- **Deletion Operation**
  - Deletion operation is more efficient in Linked Allocation



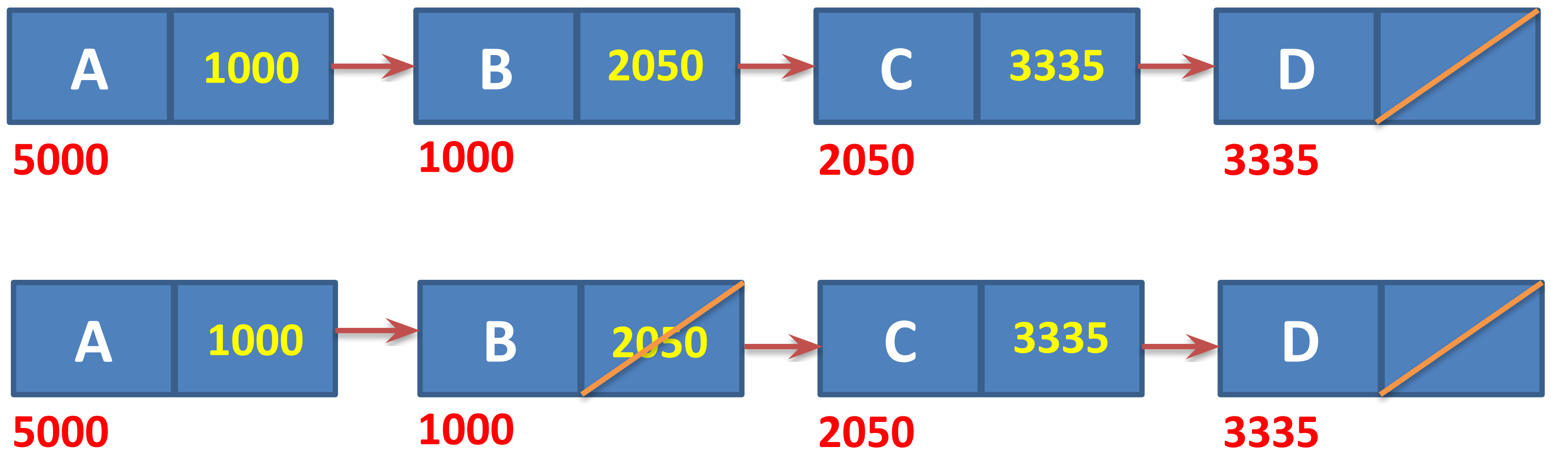
# Pros & Cons of Linked Allocation

- **Search Operation**
  - **If particular node in the list is required**, it is necessary to follow links from the first node onwards until the desired node is found, in this situation **it is more time consuming** to go through linked list than a sequential list.
  - Search operation is more time consuming in Linked Allocation.
- **Join Operation**
  - Join operation is more efficient in Linked Allocation.



# Pros & Cons of Linked Allocation

- Split Operation
  - Split operation is more efficient in Linked Allocation



## Pros & Cons of Linked Allocation

- Linked list require **more memory** compared to array because along with value it stores pointer to next node.
- Linked lists are among the simplest and most common data structures. They can be used to implement other data structures like stacks, queues, and symbolic expressions, etc…

## Operations & Type of Linked List

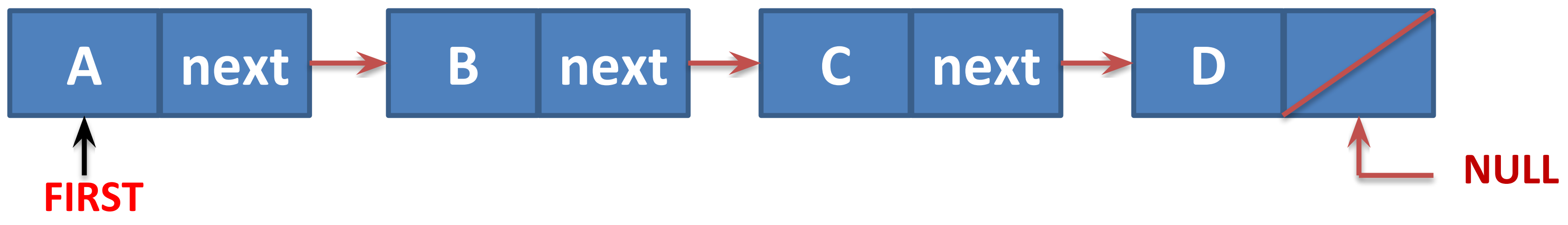
### Operations on Linked List

- Insert
  - Insert at first position
  - Insert at last position
  - Insert into ordered list
- Delete
- Traverse list (Print list)
- Copy linked list

### Types of Linked List

- Singly Linked List
- Circular Linked List
- Doubly Linked List

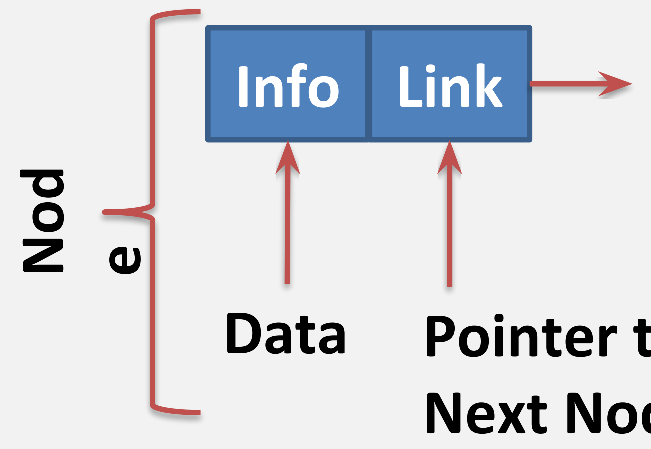
# Singly Linked List



- It is basic type of linked list.
- Each node contains data and pointer to next node.
- Last node's pointer is null.
- First node address is available with pointer variable **FIRST**.
- **Limitation** of singly linked list is **we can traverse only in one direction**, forward direction.

# Node Structure of Singly List

**Typical Node**

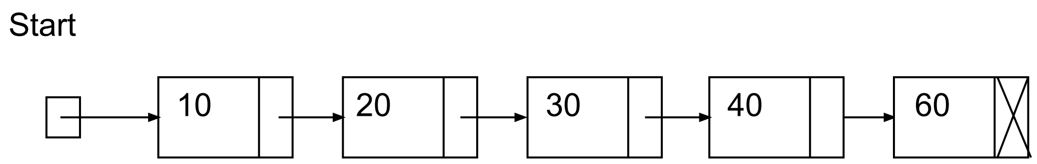


**Accessing Part of Node**
Info (Node)
Link (Node)

**C Structure to represent a node**

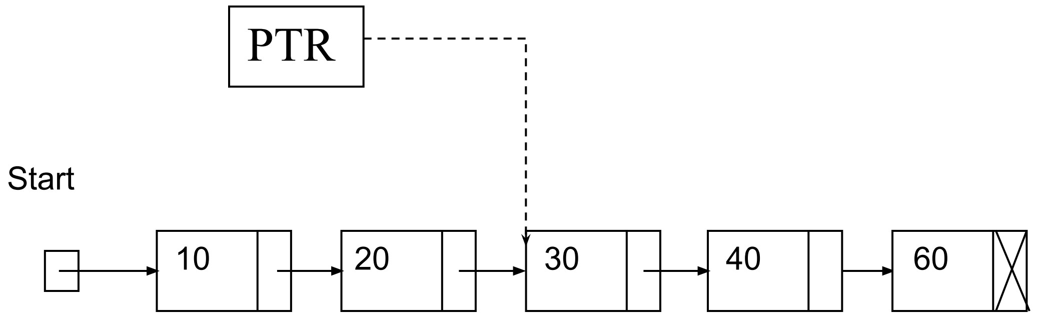
```
struct node
{
            int info;
            struct node *link;
};
```

# Some Notation



- LIST- name of the linked list.
- START – points to the first node of the linked list
- PTR- points to the node that is currently being processed
- INFO[PTR] – value of the current node.
- PTR:=LINK[PTR] – moves the pointer to next node (Update pointer)

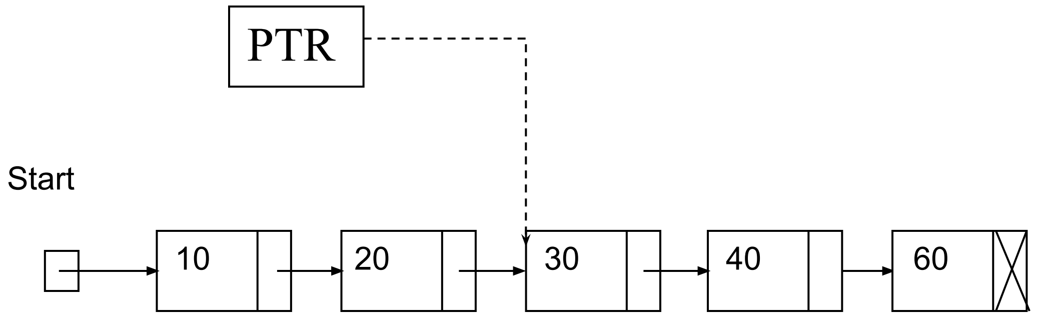
## Some Notation



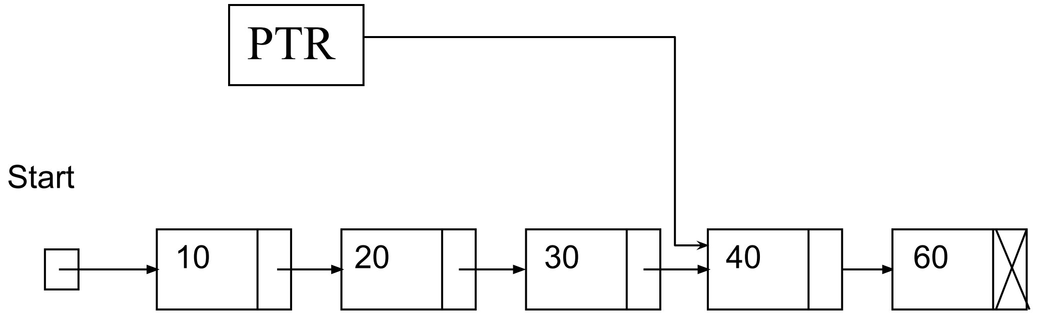
- INFO [PTR] – 30

# Some Notation

- PTR:=LINK[PTR] – moves the pointer to next node (Update pointer)



- After performing PTR:=LINK[PTR] –Update pointer)



## Traversing A Linked List

- PTR- points to the node that is currently being processed
- INFO[PTR] – value of the current node.
- PTR:=LINK[PTR] – moves the pointer to next node (Update pointer)

1. Initialize pointer PTR to START
2. Process Current node INFO[PTR]
3. Update pointer to next node
4. Continue until PTR equal to NULL

```
Traverse_List (List, Info, Link, Start)
1.  Set PTR := Start
2.  While PTR ≠ NULL repeat steps 3 and 4
3.      Apply process to Info[PTR]
4.      PTR := Link[PTR]
5.  Return.
```

## Traversing A Linked List

### Counting The No. of Elements in a Linked List

**Count_Node_List (List, Info, Link, Start)**

```
1.  Set PTR := Start and Num := 0
2.  While PTR ≠ NULL repeat steps 3 and 4
3.      Num := Num + 1
4.      PTR := Link[PTR]
5.  Return.
```

## Searching a Linked List

1. List is unsorted
2. List is sorted

## Searching an Item from an Unsorted List

Suppose we have a linked list LIST, we want to find whether an ITEM is in the LIST or not.

1. Initialize pointer PTR to START
2. Traverse through the list using PTR
3. We Have to check if PTR==NULL
4. Then we have to check INFO[PTR]==ITEM

## Searching a Linked List

1. List is unsorted
2. List is sorted

## Searching an Item from an Unsorted List

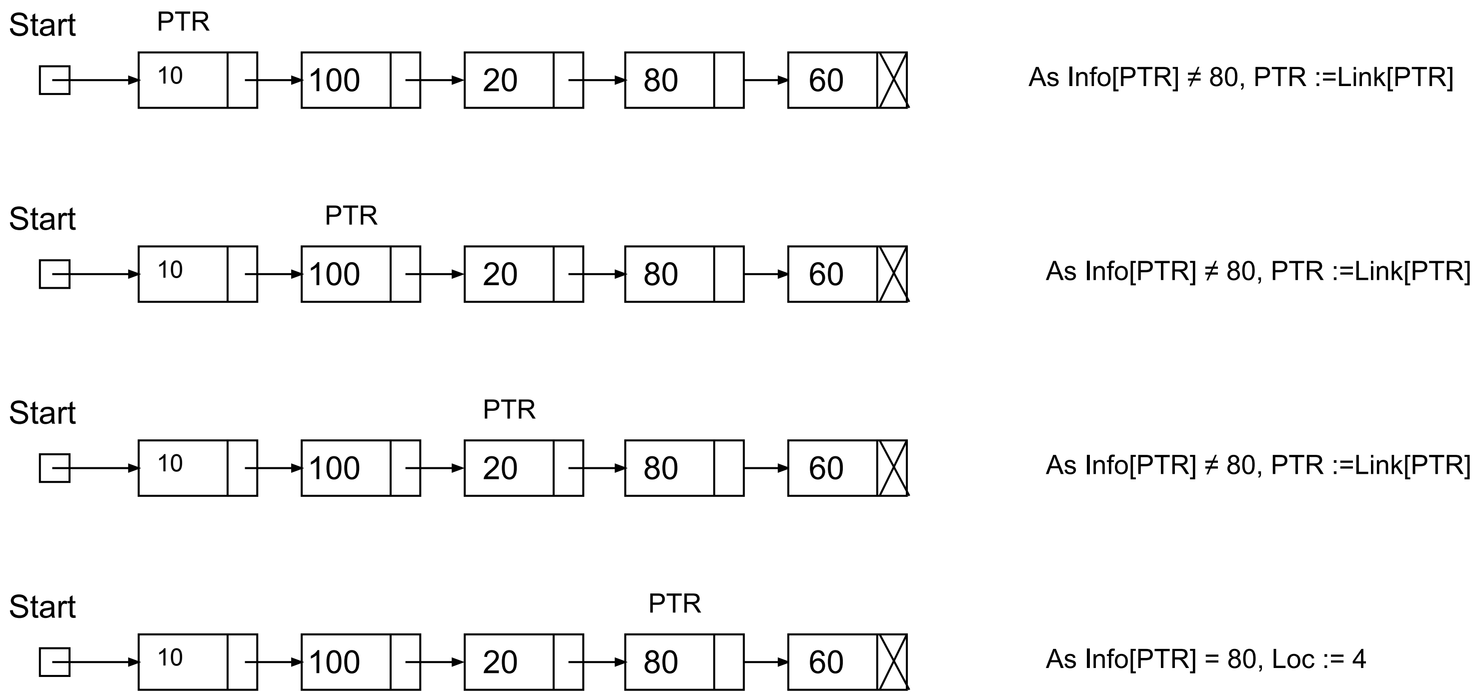
**Search_UnsortedList (Info, Link, Start, Item, Loc)**

```
1.  Set PTR := Start  and Loc := NULL
2.  While PTR ≠ NULL do steps 3 to 5
3.      If Item = Info[PTR]   then
4.          Loc := PTR , print: item found and return.   /…Successful Search
5.      Else PTR := Link[PTR]
6.  If Loc = NULL then Print: 'Item not in the list.'      /…Unsuccessful Search
7.  Return
```

## Example:

Suppose item 10 is in memory location 1.
item 100 is in memory location 2.
item 20 is in memory location 3.
item 80 is in memory location 4.
item 60 is in memory location 5.

**Item to be searched is 80.**



Start
PTR
10
100
20
80
60
As Info[PTR] ≠ 80, PTR :=Link[PTR]
As Info[PTR] ≠ 80, PTR :=Link[PTR]
As Info[PTR] ≠ 80, PTR :=Link[PTR]
As Info[PTR] = 80, Loc := 4

## Searching a Linked List

1. List is unsorted

2. List is sorted

## Searching an Item from a Sorted List

Suppose we have a linked list LIST, we want to find whether an ITEM is in the LIST or not.

1. Initialize pointer PTR to START
2. Traverse through the list using PTR
3. We Have to check if PTR==NULL
4. We can exit if INFO[PTR] > ITEM

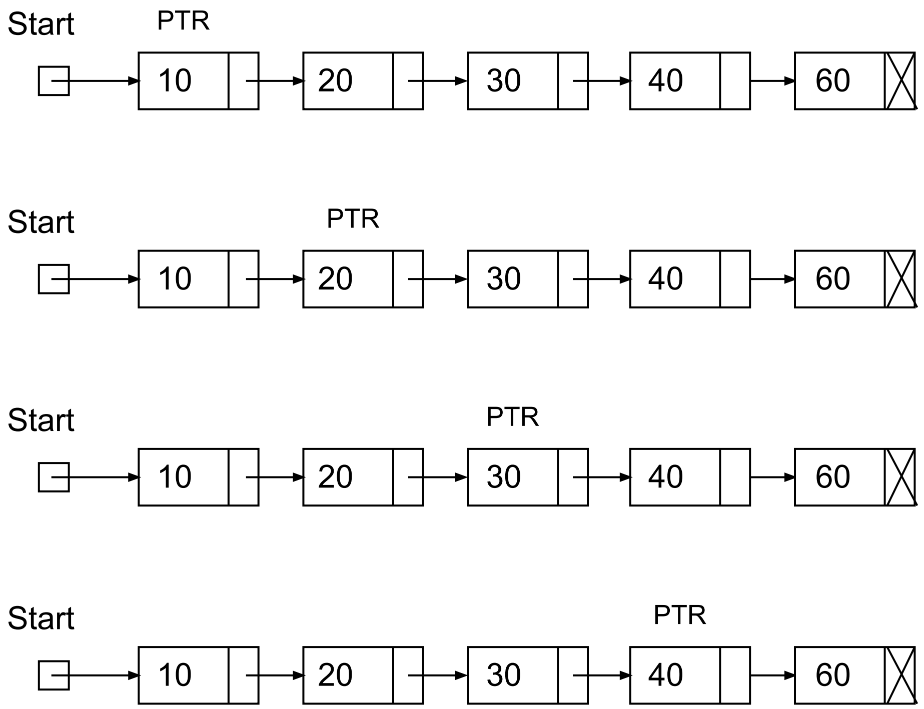
## Searching an Item from a Sorted List

```
Sch_SortedList (Info, Link, Start, Item, Loc)  / List is sorted in ascending order
1.   Set PTR := Start  and Loc := NULL
2.   While PTR ≠ NULL do steps 3 to 7
3.        if Item = Info[PTR]
4.             then Loc := PTR, print: Item found and return.    /…Successful Search
5.        else if Item > Info[PTR]
6.             then PTR := Link[PTR]
7.        else Exit.
8.   If Loc = NULL then Print: ‘Item not in the list.’       /…Unsuccessful Search
9.   Return
```

## Example:

Suppose item 10 is in memory location 1.
item 20 is in memory location 2.
item 30 is in memory location 3.
item 40 is in memory location 4.
item 60 is in memory location 5.

**Item to be searched is 40.**



As Info[PTR] ≠ 40, PTR :=Link[PTR]

As Info[PTR] ≠ 40, PTR :=Link[PTR]

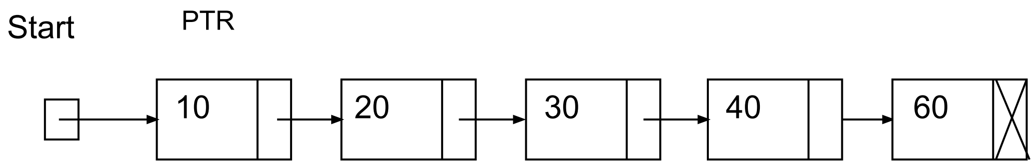
As Info[PTR] ≠ 40, PTR :=Link[PTR]

As Info[PTR] = 40, Loc := 4

## Example:

Suppose item 10 is in memory location 1.
item 20 is in memory location 2.
item 30 is in memory location 3.
item 40 is in memory location 4.
item 60 is in memory location 5.

**Item to be searched is 6.**



As item 6 < Info[PTR] and the given list is already sorted in ascending order, so item can not be in the list.

So, Loc = NULL