Linked List

Linked List :: Basic Concepts

- A list refers to a set of items organized sequentially.
  - An array is an example of a list.
    - The array index is used for accessing and manipulating array elements.
  - Problems with array:
    - The array size has to be specified at the beginning.
    - Deleting an element or inserting an element may require shifting of elements in the array.

Linked list

- Linear data structure
- Is an ordered collection of finite, homogeneous data elements called nodes where the linear order is maintained by means of links or pointers

Node: an element in a linked list

A node consists of two fields

- ***Data***(to store the actual information)
- ***Link*** (to point to the next node)

In linked list the adjacency between the elements are maintained by means of links or pointers. A link or pointer is the address of the subsequent element

Difference between array and linked list

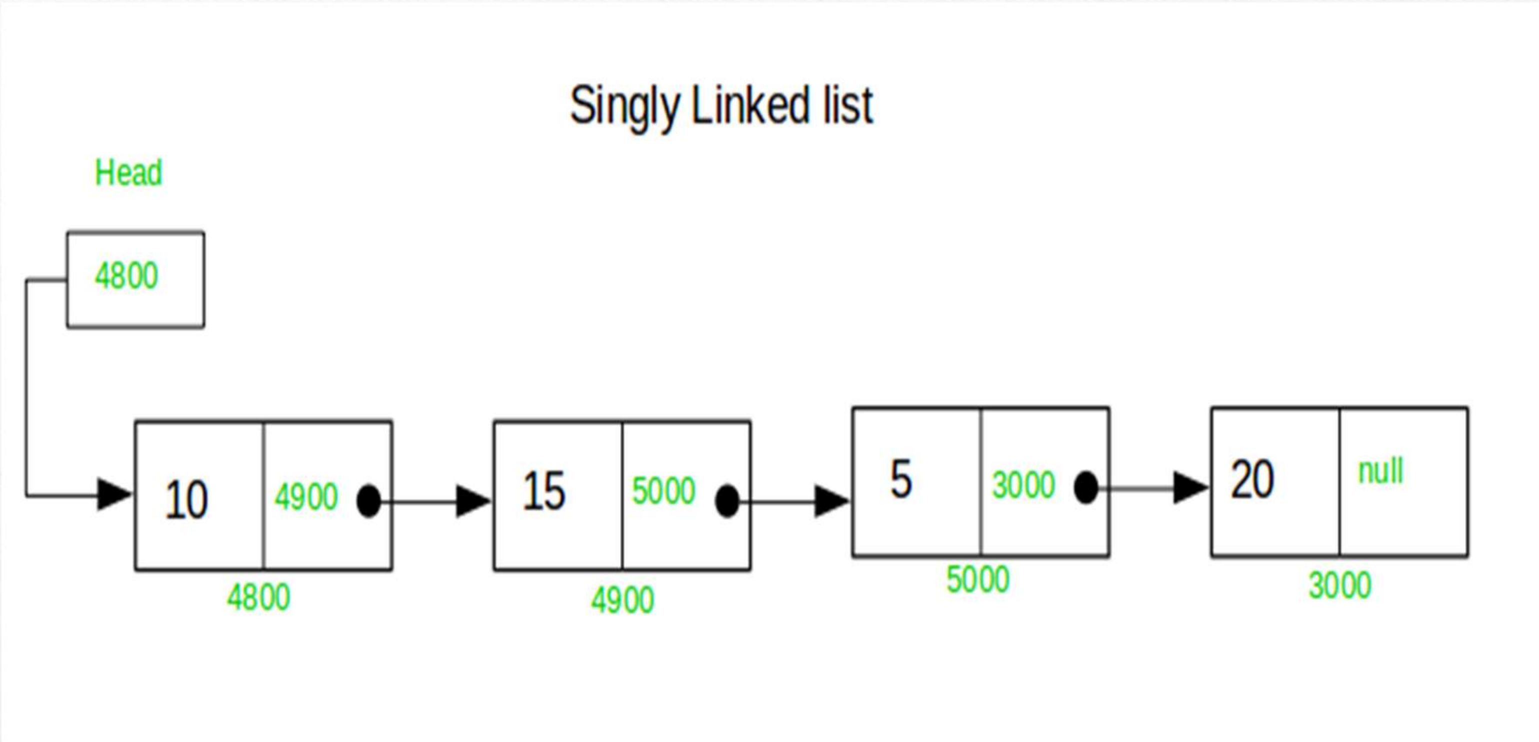
- *Array*

- Elements are stored in consecutive memory locations.
- Memory allocated at compile time
- Once memory is allocated ,it can not be extended any more.
- There for array is known as static data structure.

- *Linked list*

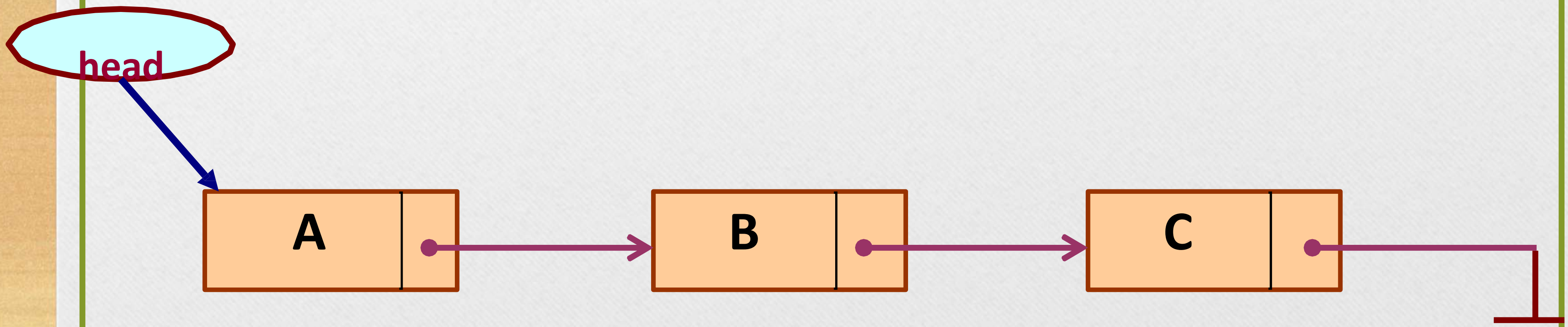
- Elements are stored in different memory location.
- Memory is allocated at runtime
- Once memory is allocated , it can be varied or extended during it use.
- There for linked list is known as dynamic data structure.

Array	Linked list
Operation like insertion, deletion..takes more time in an array	Operation like insertion, deletion takes less time in linkedlist
Memory space is only for data stored	Extra memory space for pointer in every node
It is easier and faster to access the element in an array with the help of index	Time consuming as we have to start traversing from the first element (random access not allowed)


Singly Linked list
Head
4800
10
4900
15
5000
5
3000
20
null
4800
4900
5000
3000

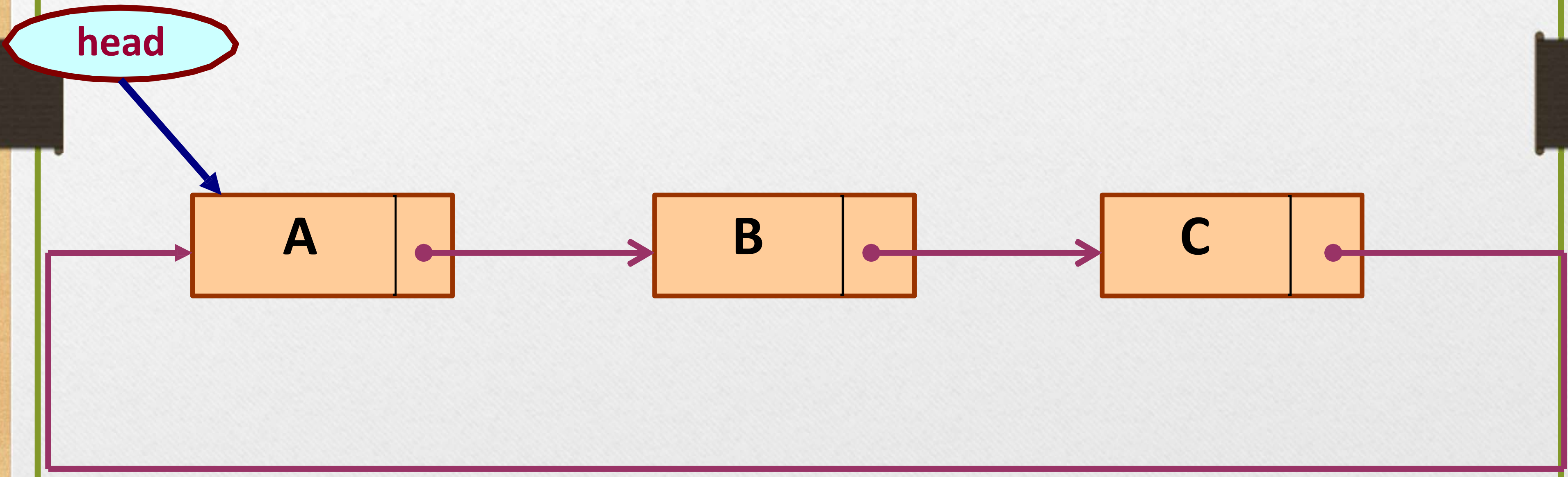
Types of Lists

- Depending on the way in which the links are used to maintain adjacency, several different types of linked lists are possible.
  - Linear singly-linked list (or simply linear list)

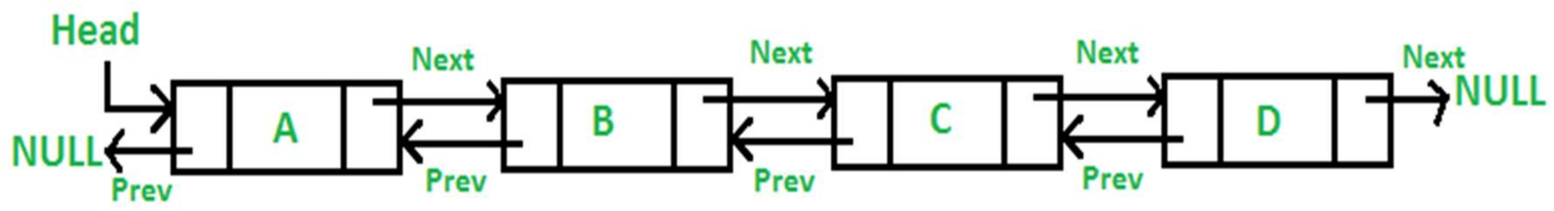


– Circular linked list

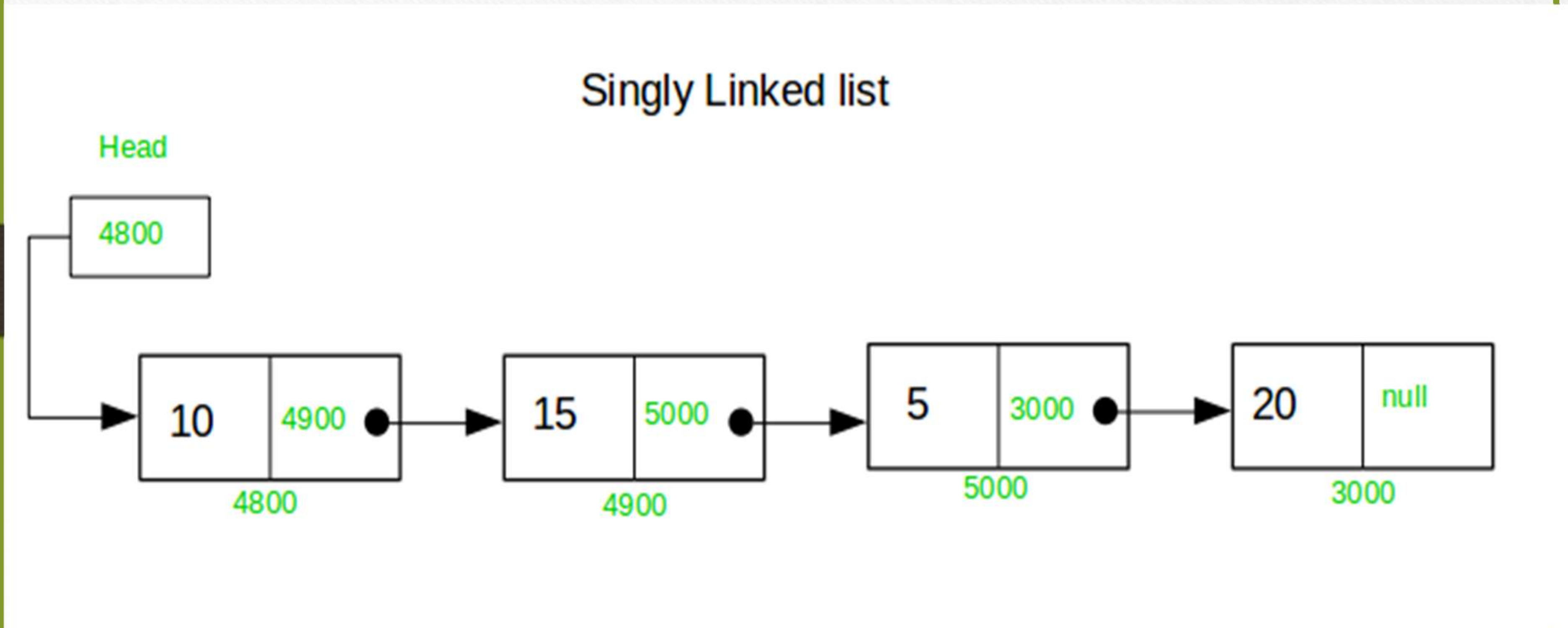
- **The pointer from the last element in the list points back to the first element.**



– **Doubly linked list**
  - **Pointers exist between adjacent nodes in both directions.**
  - **The list can be traversed either forward or backward.**
  - **Usually two pointers are maintained to keep track of the list, head and tail.**


Head
Next
Next
Next
Next
NULL
A
B
C
D
NULL
Prev
Prev
Prev
Prev

SINGLY LINKED LIST



Singly Linked List

- A singly linked list is a linked list in which **each node contain only one link pointing to the next node in the list.**
- In a singly linked list , the **first node always pointed by a pointer** called **HEAD**.
- If the **link** of the node points to **NULL** , then that indicates the **end of the list**.
- Here one can move from left to right only. So it is also called one-way list

Representation of a linked list in memory

- Two ways:

1. Static representation using array

1. **Dynamic representation using free pool storage**

Static representation

- Two arrays are maintained:
  - One for data and other for links.

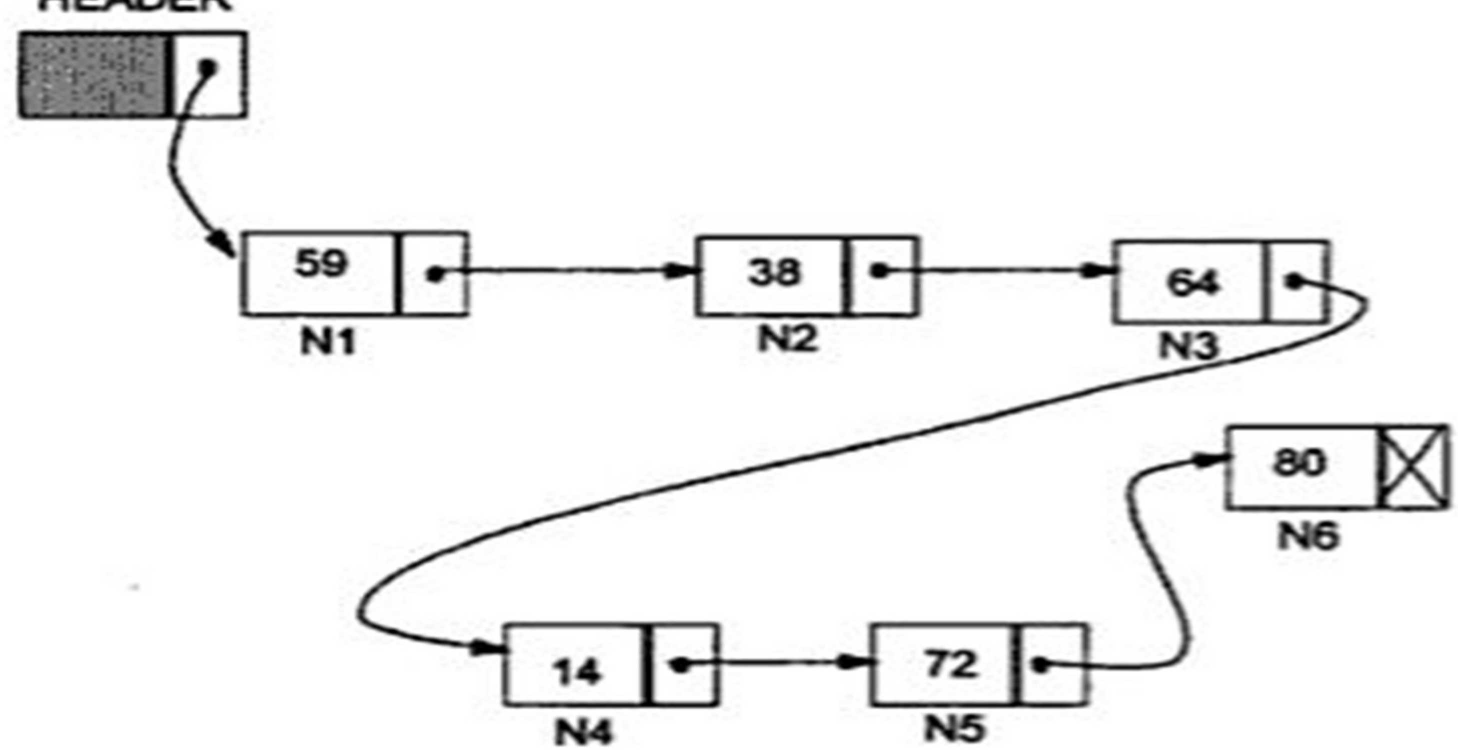
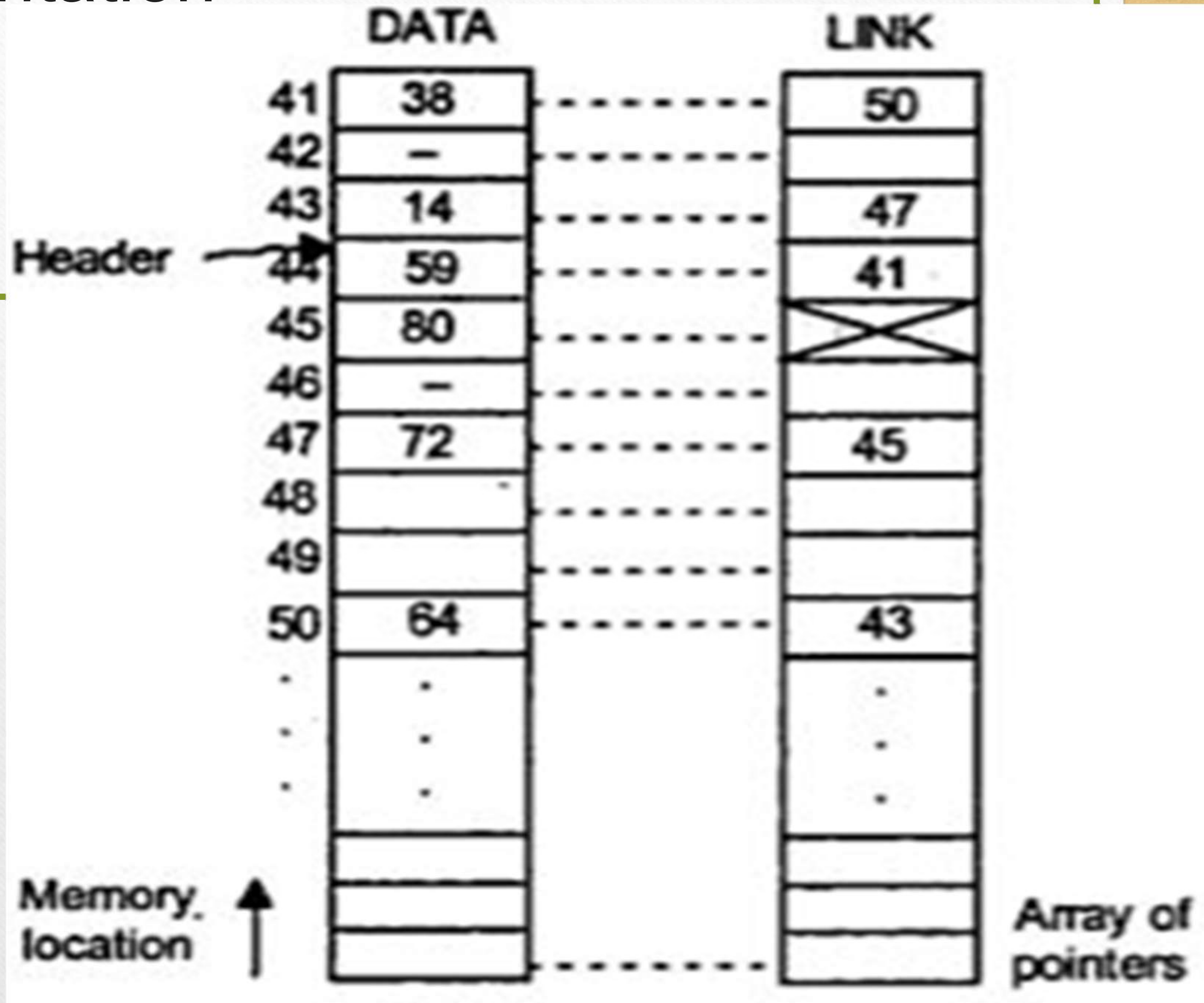


Fig. 3.2 A single linked list with 6 nodes.

Static representation

- **Two parallel arrays of equal size are allocated which should be sufficient to store the entire linked list**



Dynamic representation

- The efficient way of representing a linked list is using the free pool of storage.
- There is a
  - *memory bank* : Collection of free memory spaces &
  - *memory manager*: a program
- Whenever a node is required, the request is placed to the memory manager.
- It will search the memory bank for the block. If found, it will be granted.
- *Garbage collector*: Another program that returns the unused node to the memory bank.

Dynamic representation

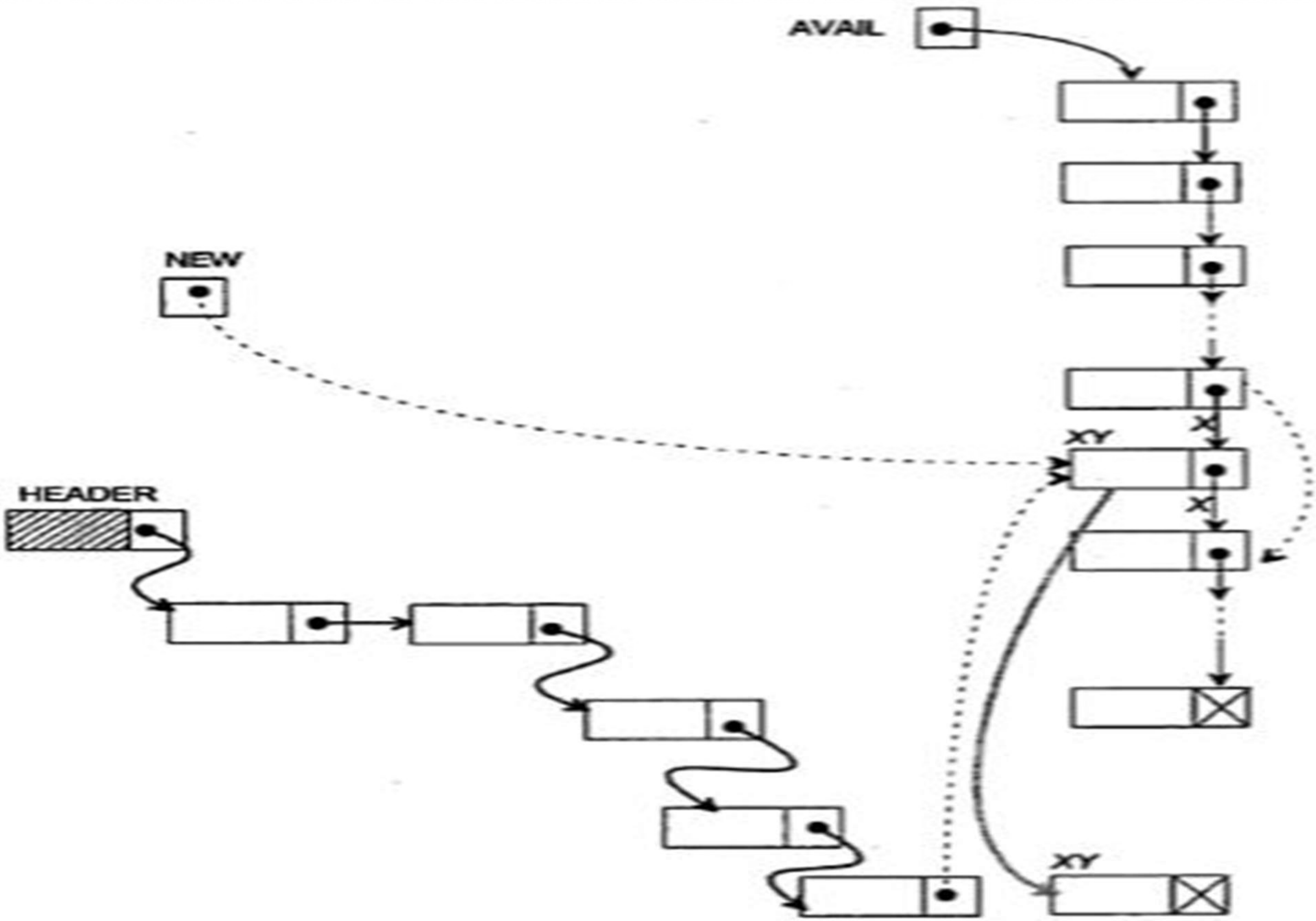


Fig. 3.4(a) Allocation of a node from memory bank to a linked list.

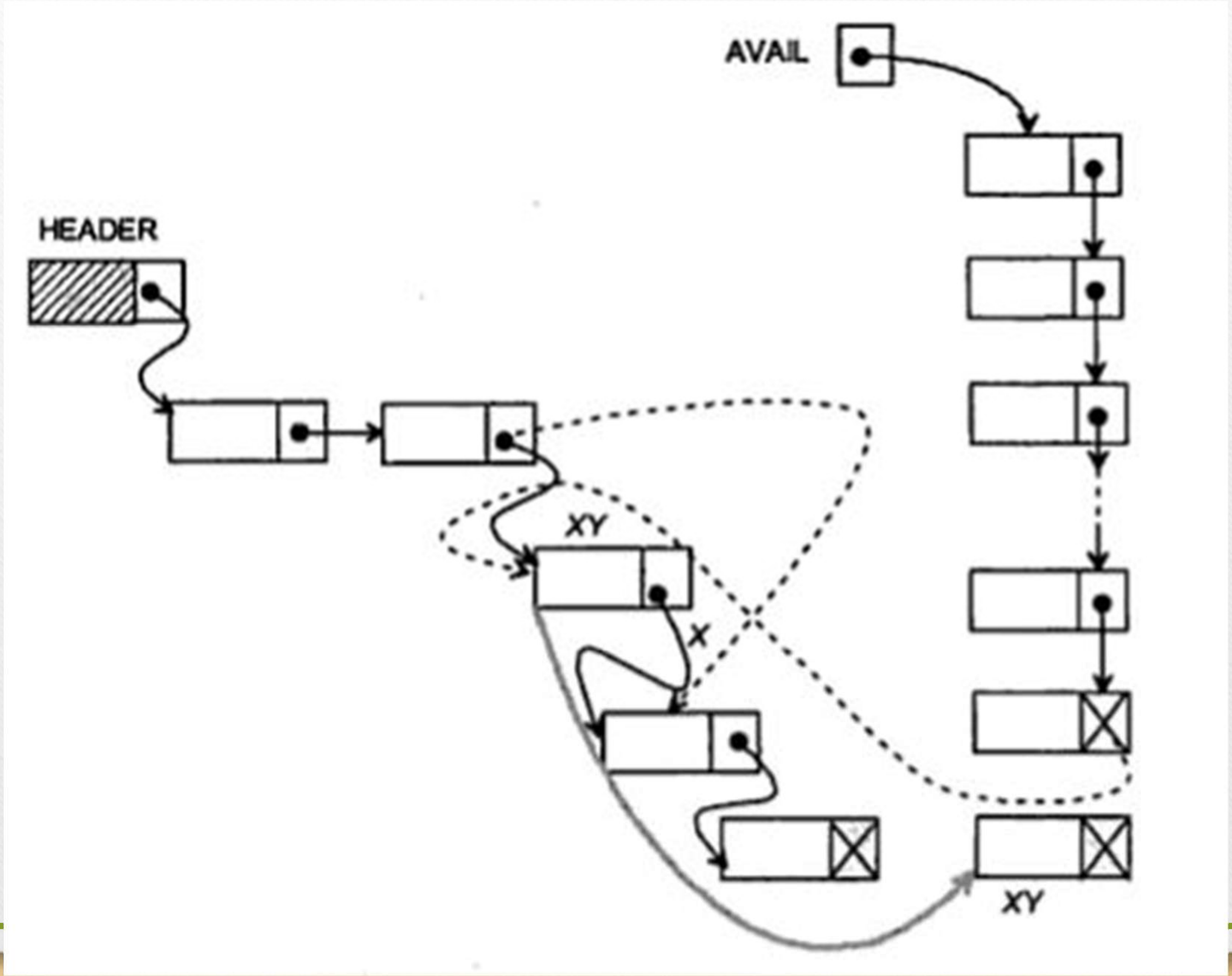
- A list of variable memory space is there whose pointer is stored in Avail
- For a request of a node, the list Avail is searched for the block of right size

---

- If Avail is null or block of right size is not available the memory will return the size accordingly
- Suppose a block is found and let it be ‘xy’, then the memory manager will return the pointer of xy to the caller in a temporary buffer say ‘New’
- The newly availed node xy can be inserted at any position in the linked list by changing the pointers of the concerned nodes.

Dynamic representation

- Returning a node to memory bank



Operations on a singly linked list are

1. **Traversing (Display)** all the elements of the list.
2. **Inserting** a node into the list.

    i. insert an element at the beginning of the list

    ii. insert an element at the end of the list

    iii. insert an element at the specified position in the list

3. **Deleting** a node from the list

    i. Delete an element from the beginning of the list

    ii. Delete an element at the end of the list

    iii. Delete an element at the specified position in the list

4. **Copying** the list to make a duplicate of it.

5. **Merging** the linked list with another one to make a larger list.

6. **Searching** for an element in the list.

7. **Count** the number of elements.

Creating a List

Representation of a node using structure

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

Creation of a new node

P=(struct node *)malloc(sizeof
struct node);

- A node is created named ‘p’

- P→ data=8
- P→ link=null

Procedure Getnode(node)

Procedure GetNode

Input: NODE is the type of the data for which a memory has to be allocated.

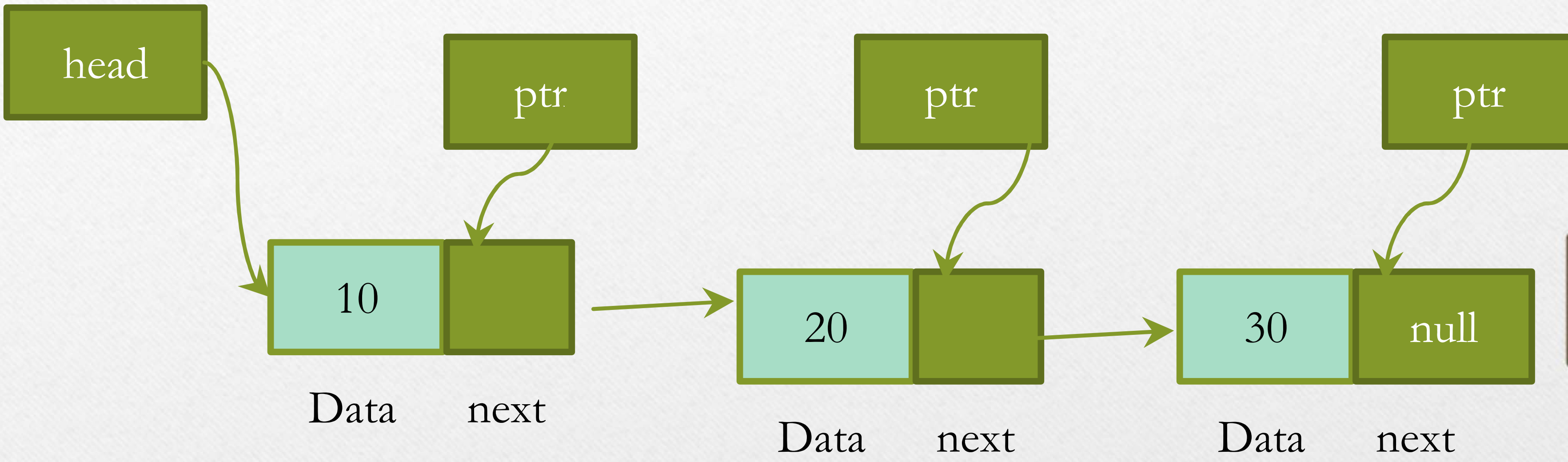
Output: Return a message if the allocation fails else the pointer to the memory block allocated.

Steps:

```
1.  If (AVAIL = NULL)                    // AVAIL is the pointer to the pool of free storage
2.      Return(NULL)
3.      Print "Insufficient memory: Unable to allocate memory"
4.  Else                                              // Sufficient memory is available
5.      ptr = AVAIL // Start from the location, where AVAIL points
6.      While (SizeOf(ptr) ≠ SizeOf(NODE)) and (ptr→LINK ≠ NULL) do
                // Till the desired block is found or the search reaches the end of the pool
7.          ptr1 = ptr                             // To keep the track of the previous block
8.          ptr = ptr→LINK                                          // Move to the next block
9.      EndWhile
10.     If (SizeOf(ptr) = SizeOf(NODE))           // Memory block of right size is found
11.         ptr1→LINK = ptr→LINK                                  // Update the AVAIL list
12.         Return(ptr)
13.     Else
14.         Print "The memory block is too large to fit"
15.         Return(NULL)
16.     EndIf
17. EndIf
18. Stop
```

Traversing a single linked list

- Here we visit every node in the list starting from the first node to the last one.



Algorithm Traverse_SL

Input: *HEADER* is the pointer to the header node.

Output: According to the *Process*( )

Data structures: A single linked list whose address of the starting node is known from the *HEADER*.

Steps:

1. ptr = HEADER→LINK // ptr is to store the pointer to a current node
2. **While** (ptr ≠ NULL) **do** // Continue till the last node
3.   **Process**(ptr) // Perform *Process*( ) on the current node
4.   ptr = ptr→LINK // Move to the next node
5. **EndWhile**
6. **Stop**

Note: A simple operation, namely *Process*() may be devised to print the data content of a node, or count the total number of nodes, etc.

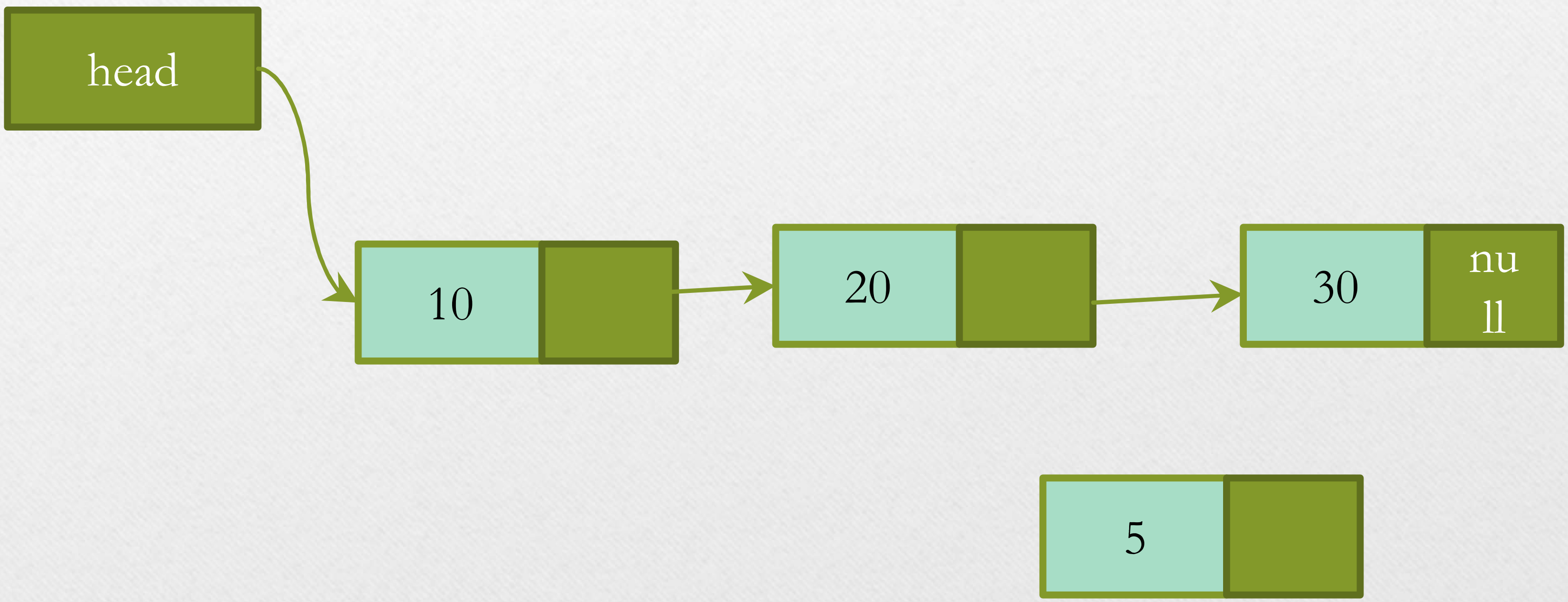
Insertion

- There are various positions where a node can be inserted:

1. Inserting at the front (as a first element)
2. Inserting at the end(as a last element)
3. Inserting at any other position

Insertion- At the beginning

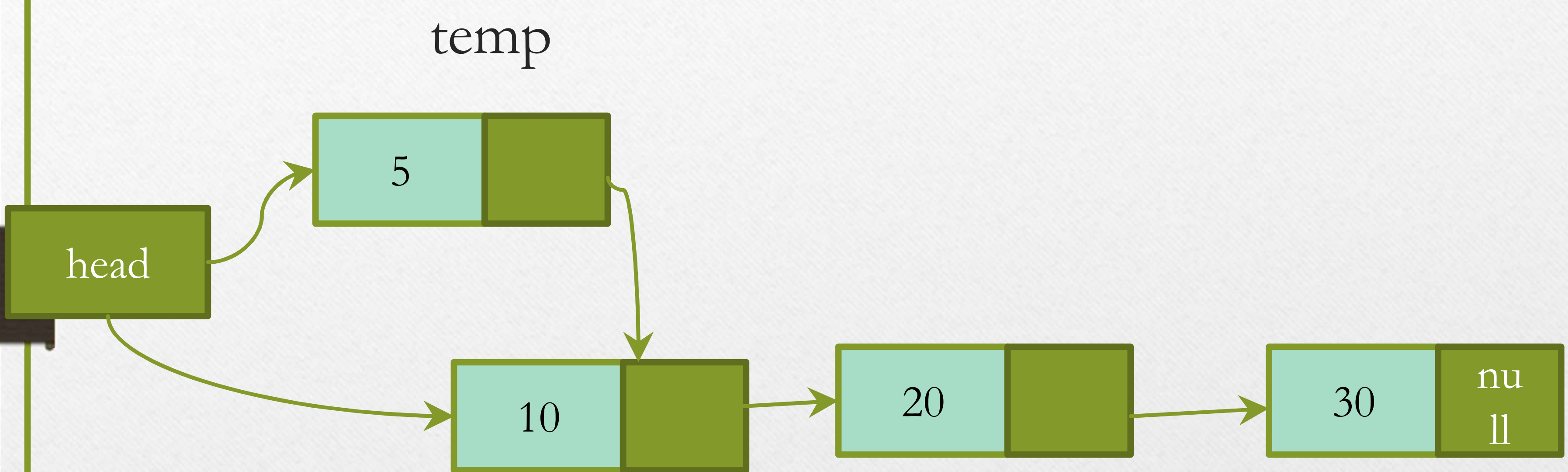
- Consider a linked list with 3 nodes



- We want to insert a new node temp at the beginning of the list

Insertion- At the beginning

- Now temp is inserted at 1st position



temp-> link=head

head=temp

Inserting a node at the front of a single linked list.

- The algorithm InsertFront_SL is used to insert a node at the front of a single linked list.

Algorithm InsertFront_SL

Input: HEADER is the pointer to the header node and X is the data of the node to be inserted.

Output: A single linked list with a newly inserted node at the front of the list.

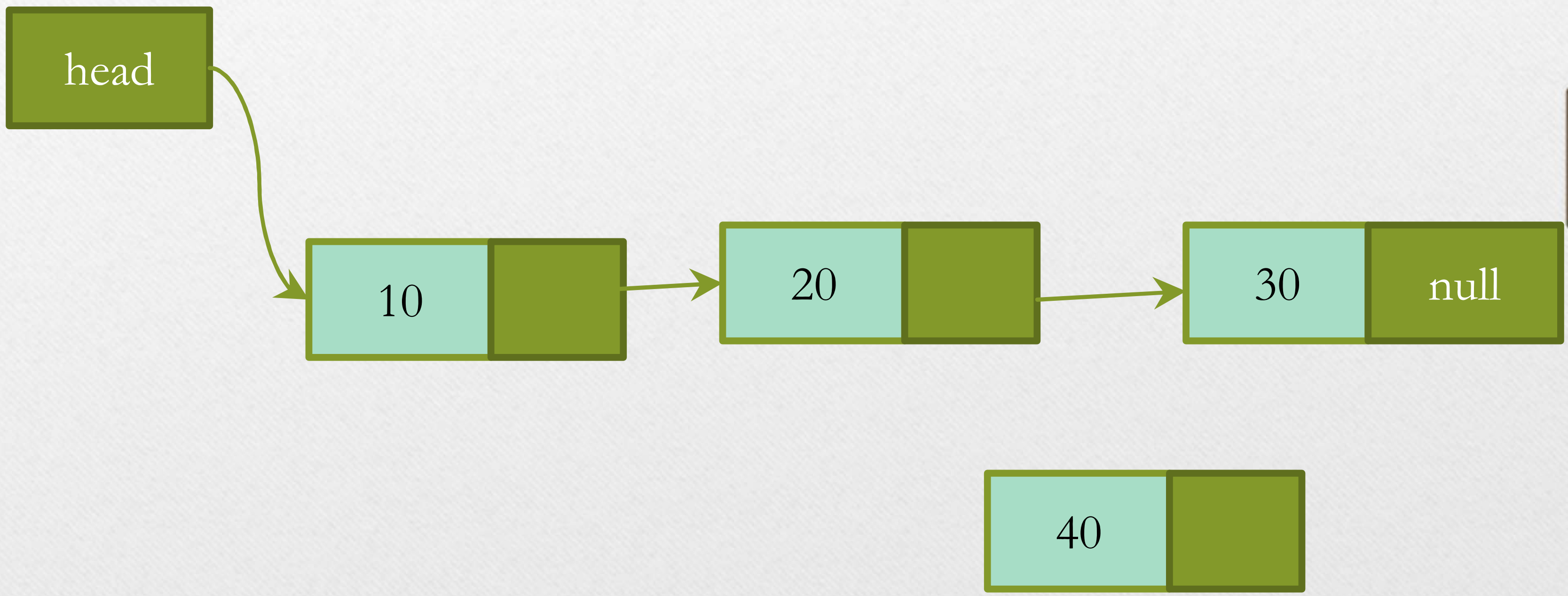
Data Structures: A single linked list whose address of the starting node is known from the HEADER.

Steps:

1. new = **GetNode**(NODE) // Get a memory block of type NODE and store its pointer in new
2. **If** (new = NULL) **then** // Memory manager returns NULL on searching the memory bank
3. **Print** "Memory underflow: No insertion"
4. *Exit* // Quit the program
5. **Else** // Memory is available and get a node from memory bank
6. new→LINK = HEADER→LINK // Change of pointer 1 as shown in Figure 3.5(a)
7. new→DATA = *X* // Copy the data *X* to newly availed node
8. HEADER→LINK = new // Change of pointer 2 as shown in Figure 3.5(a)
9. **EndIf**
10. **Stop**

Insertion- At the end

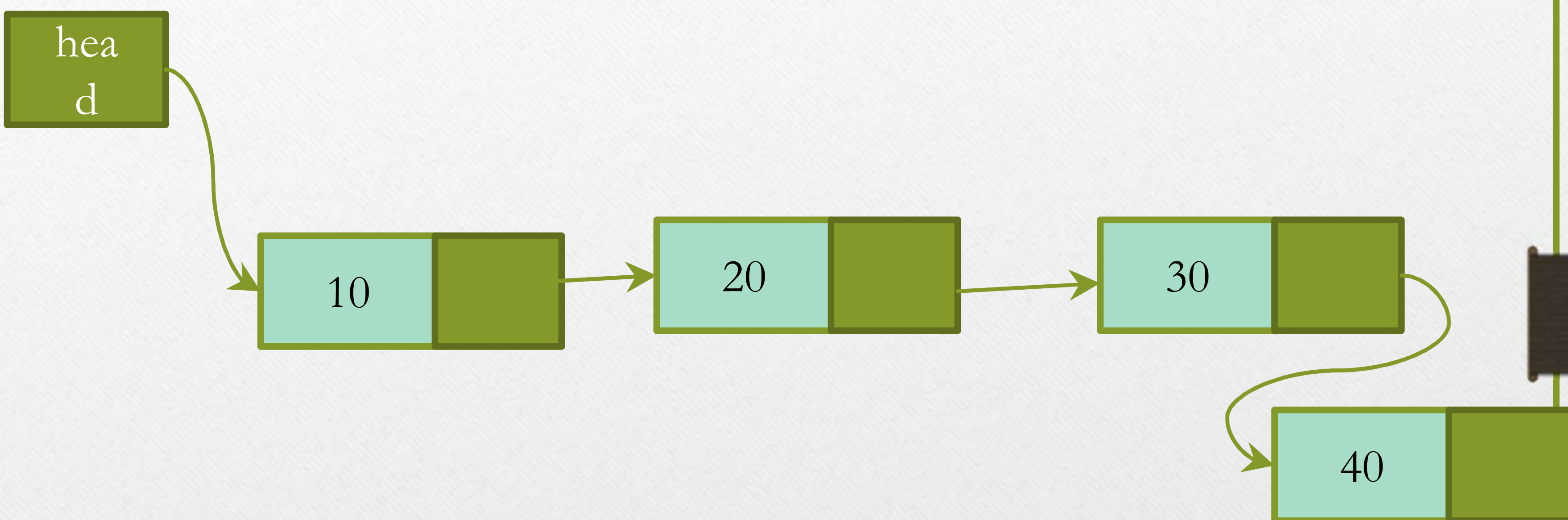
- Consider a linked list with 3 nodes



- We want to insert a new node temp   at the end of the list

Insertion- At the end

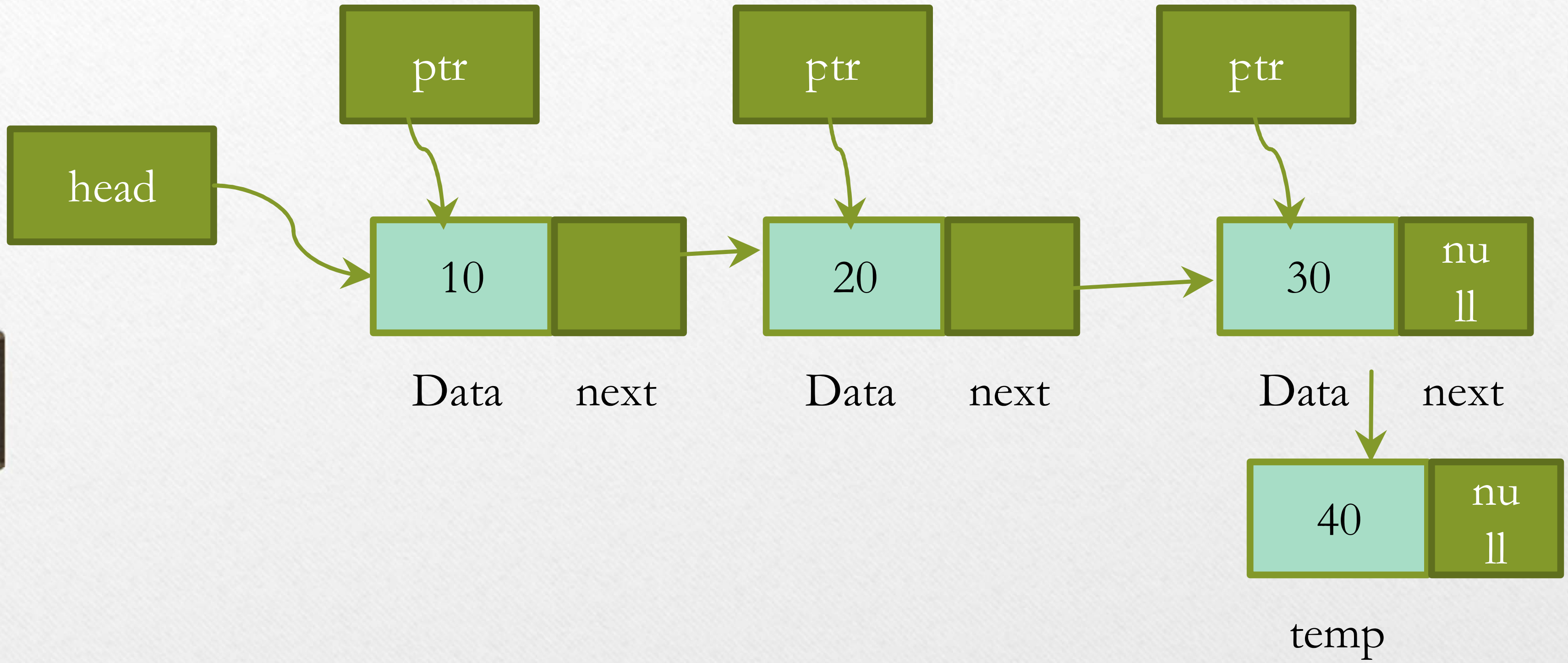
- List should be traversed first.



- Now temp is inserted at the end

Insertion- At the end

Here first we need to traverse the list to get the last node.



Inserting a node at the end of a single linked list.

- The algorithm InsertEnd_SL is used to insert a node at the end of a single linked list.

Algorithm InsertEnd_SL

Input: HEADER is the pointer to the header node and X is the data of the node to be inserted.

Output: A singe linked list with a newly inserted node having data X at the end of the list.

Data Structures: A single linked list whose address of the starting node is known from the HEADER.

Algorithm InsertEnd_SL

Input: *HEADER* is the pointer to the header node and *X* is the data of the node to be inserted.

Output: A single linked list with a newly inserted node having data *X* at the end of the list.

Data structures: A single linked list whose address of the starting node is known from the *HEADER*.

Steps:

1. new = **GetNode**(NODE) // Get a memory block of type NODE and return its pointer as new
2. **If** (new = NULL) **then** // Unable to allocate memory for a node
3. **Print** "Memory is insufficient: Insertion is not possible"

```
4.   Exit                                          // Quit the program
5. Else              // Move to the end of the given list and then insert
6.   ptr = HEADER                          // Start from the HEADER node
7.   While (ptr→LINK ≠ NULL) do                       // Move to the end
8.     ptr = ptr→LINK                // Change pointer to the next node
9.   EndWhile
10.  ptr→LINK = new                // Change the link field of last node:
                                         Pointer 1 as in Figure 3.5(b)
11.  new→DATA = X               // Copy the content X into the new node
12. EndIf
13. Stop
```

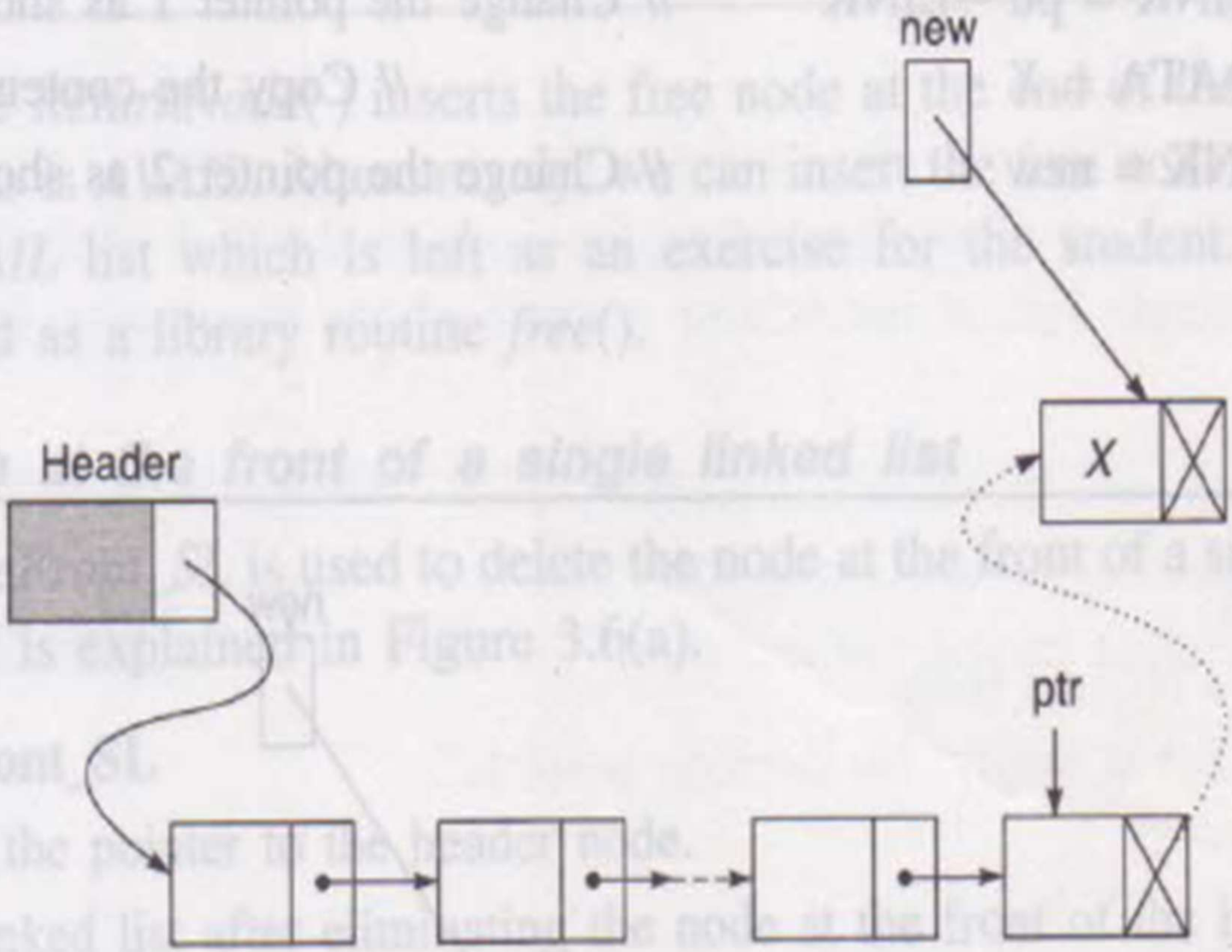
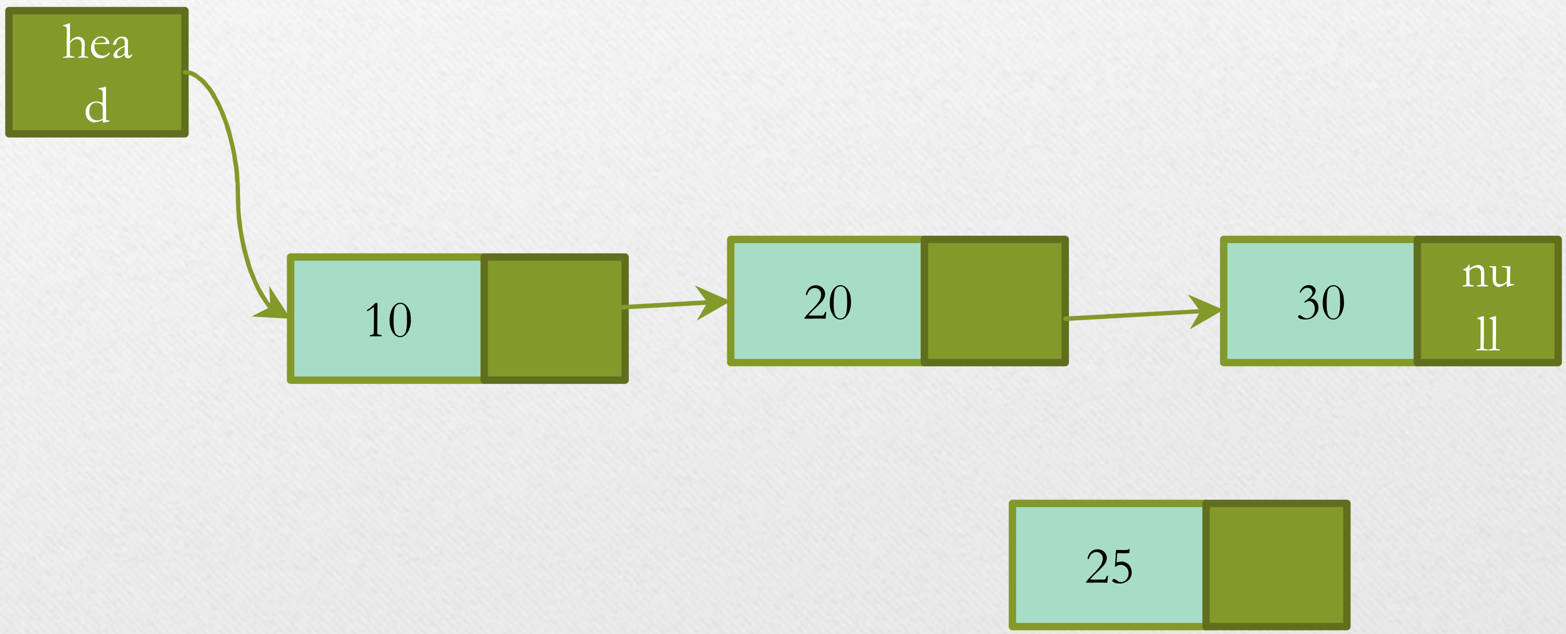


Figure 3.5(b) Inserting a node at the end of a single linked list.

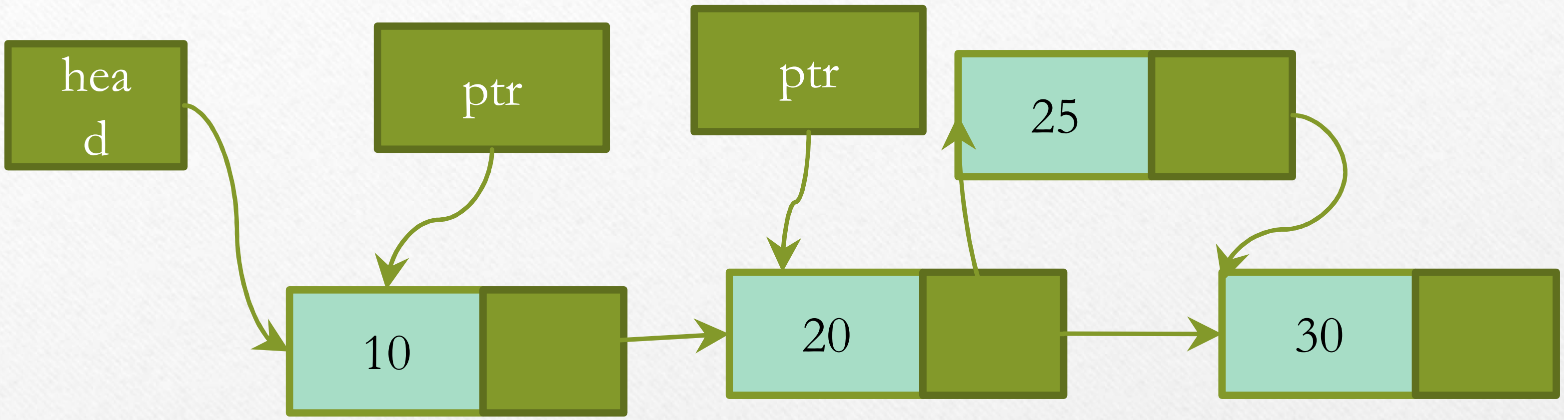
Insertion- At any position in the list

- Consider a linked list with 3 nodes



- We want to insert a new node temp after node 20

Insertion- After a node



Now temp is inserted after **node 20**

Algorithm InsertAny_SL

Input: *HEADER* is the pointer to the header node, *X* is the data of the node to be inserted, and *KEY* being the data of the key node after which the node has to be inserted.

Output: A single linked list enriched with newly inserted node having data *X* after the node with data *KEY*.

Data structures: A single linked list whose address of the starting node is known from the *HEADER*.

Steps:

1. new = **GetNode**(NODE) // Get a memory block of type NODE and returns its pointer as new
2. **If** (new = NULL) **then** // Unable to allocate memory for a node
3. **Print** "Memory is insufficient: Insertion is not possible"

```
4.      Exit                                                              // Quit the program
5.   Else
6.      ptr = HEADER                                              //Start from the HEADER node
7.      While (ptr→DATA ≠ KEY) and (ptr→LINK ≠ NULL) do                  // Move to the node
                                       //having data as KEY or at the end if KEY is not in the list
8.          ptr = ptr→LINK
9.      EndWhile
10.     If (ptr→LINK = NULL) then                                  // Search fails to find the KEY
11.         Print "KEY is not available in the list"
12.         Exit
13.     Else
14.         new→LINK = ptr→LINK              // Change the pointer 1 as shown in Figure 3.5(c)
15.         new→DATA = X                                  // Copy the content into the new node
16.         ptr→LINK = new                   // Change the pointer 2 as shown in Figure 3.5(c)
17.     EndIf
18.  EndIf
19.  Stop
```

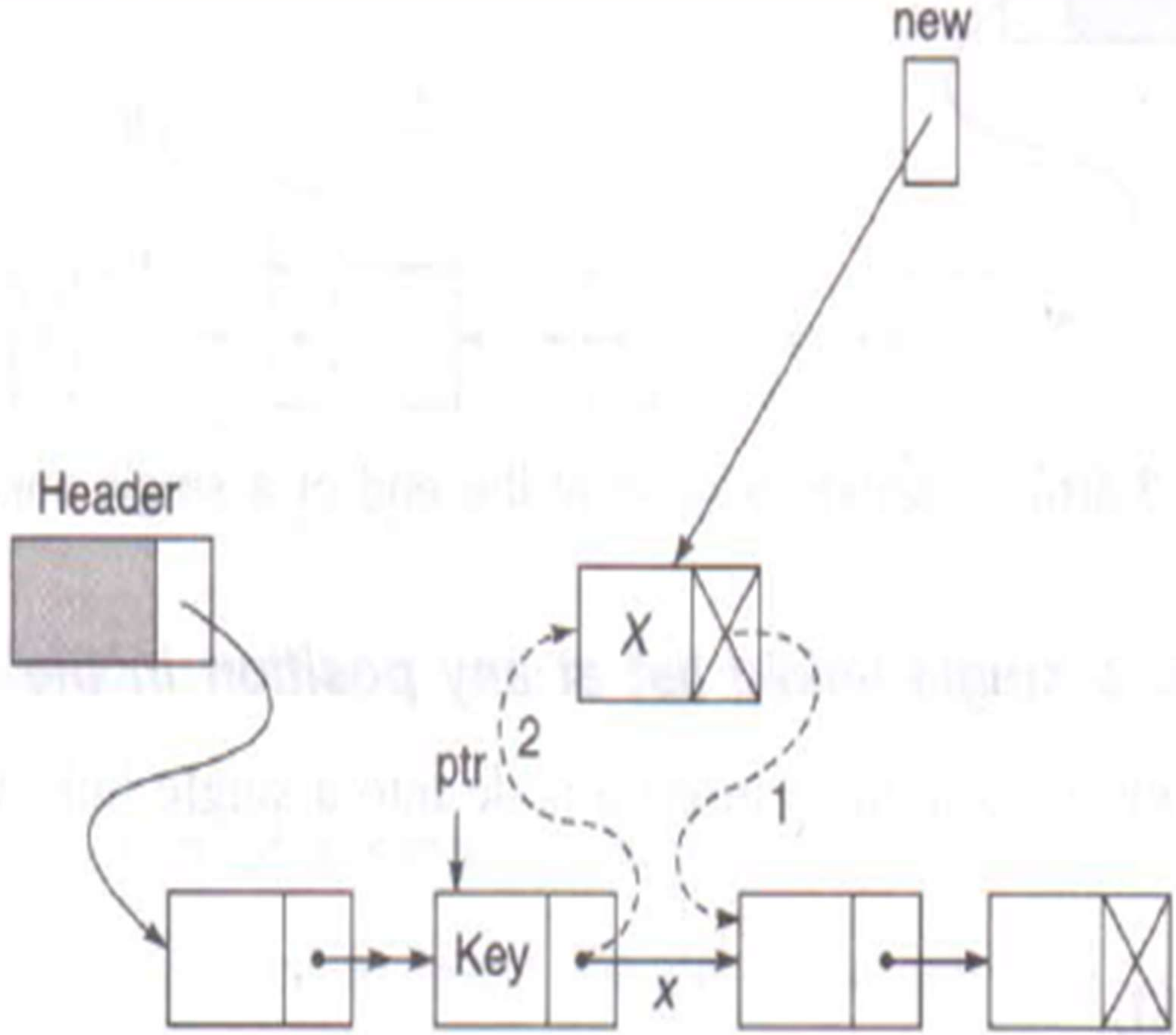


Figure 3.5(c) Inserting a node at any position in a single linked list.

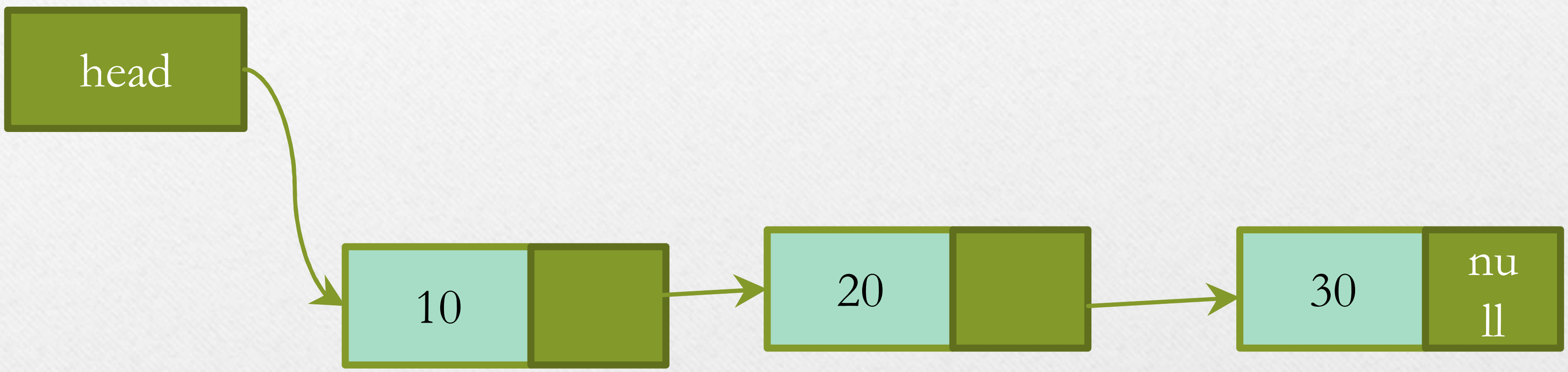
Deletion

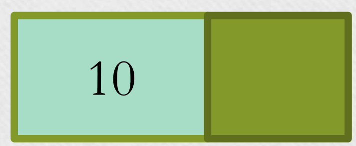
- In a linked list, an element can be deleted:

1. From the 1st location
2. From the last location
3. From any position in the list

Deletion- From the beginning

- Consider a linked list with 3 nodes

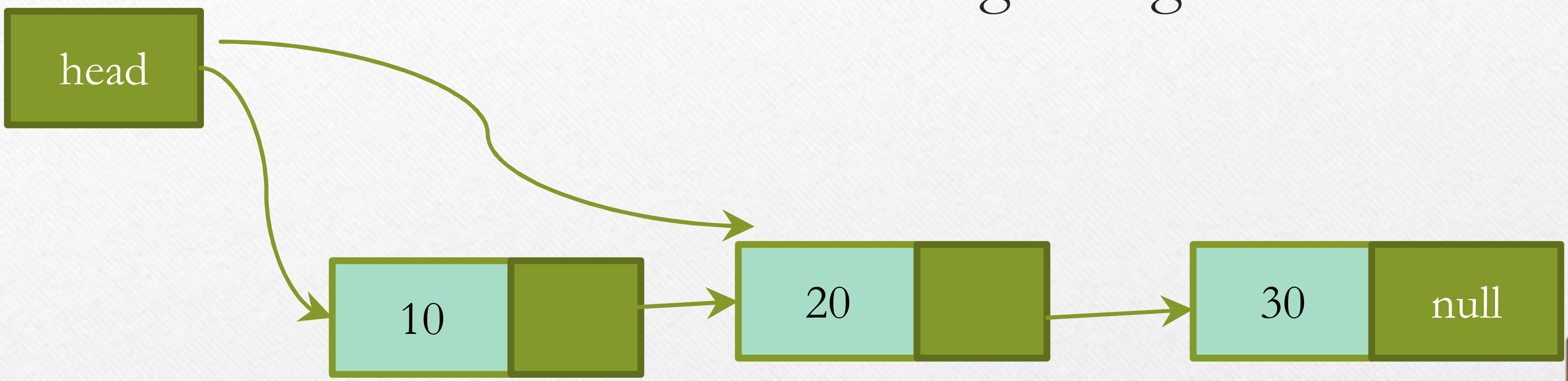




- We want to delete a node from the beginning of the list

Deletion- From the beginning

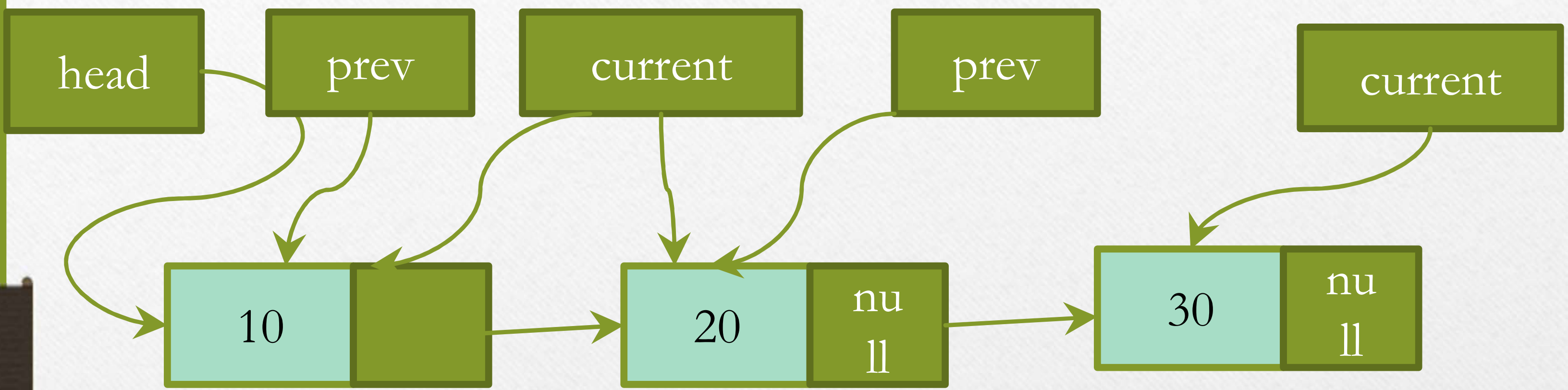
- Now node 10 is deleted from the beginning



1. If (head==NULL)
2. Print 'list empty'
3. Else
4. head=head-> link

Deletion- From the end

- Consider a linked list with 3 nodes



- We want to delete a node 

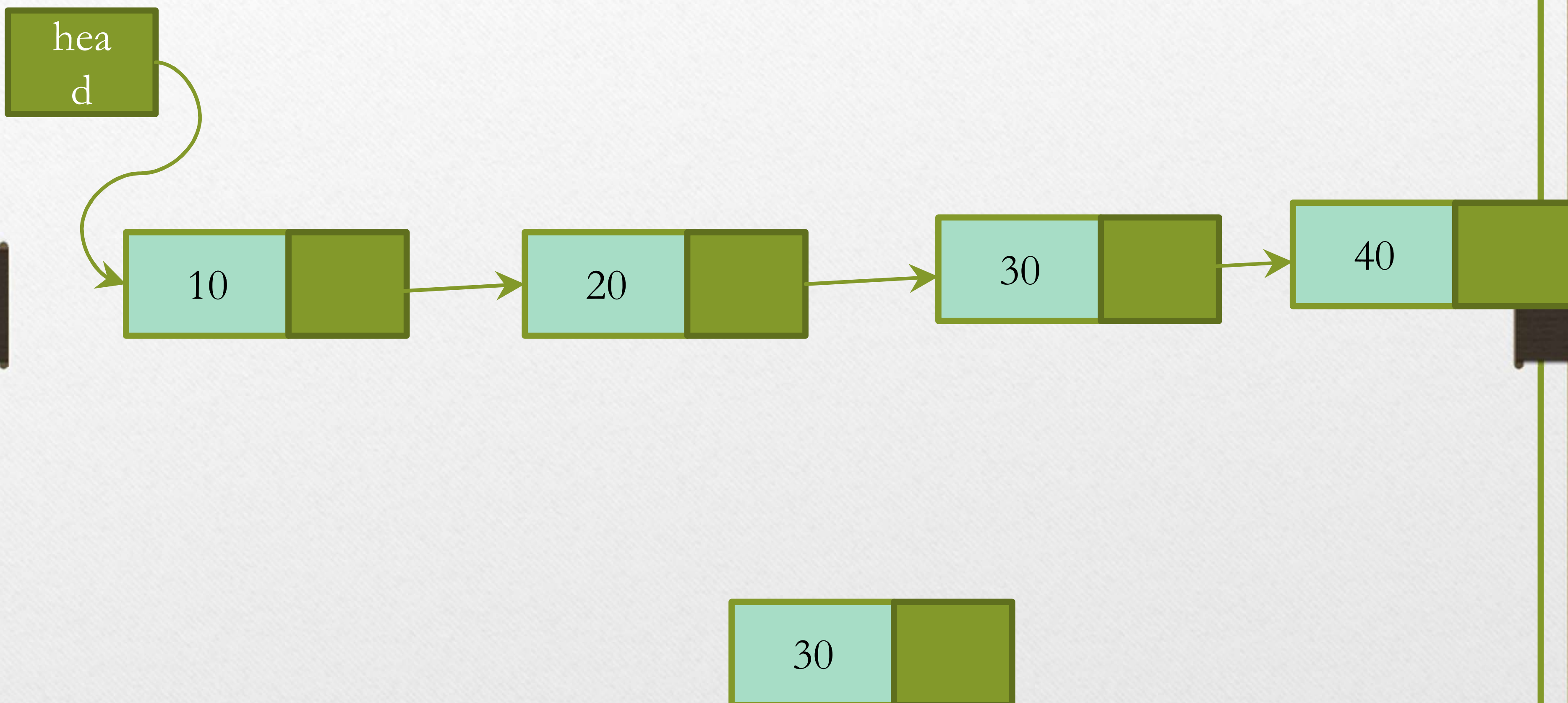
from the end of the list

Deletion- From the end

1. Set current=head, prev=head
2. Repeat while(current-> link!=null)
3.   prev= current
4.   current=current->link
5. prev->link=null

Deletion- From any position

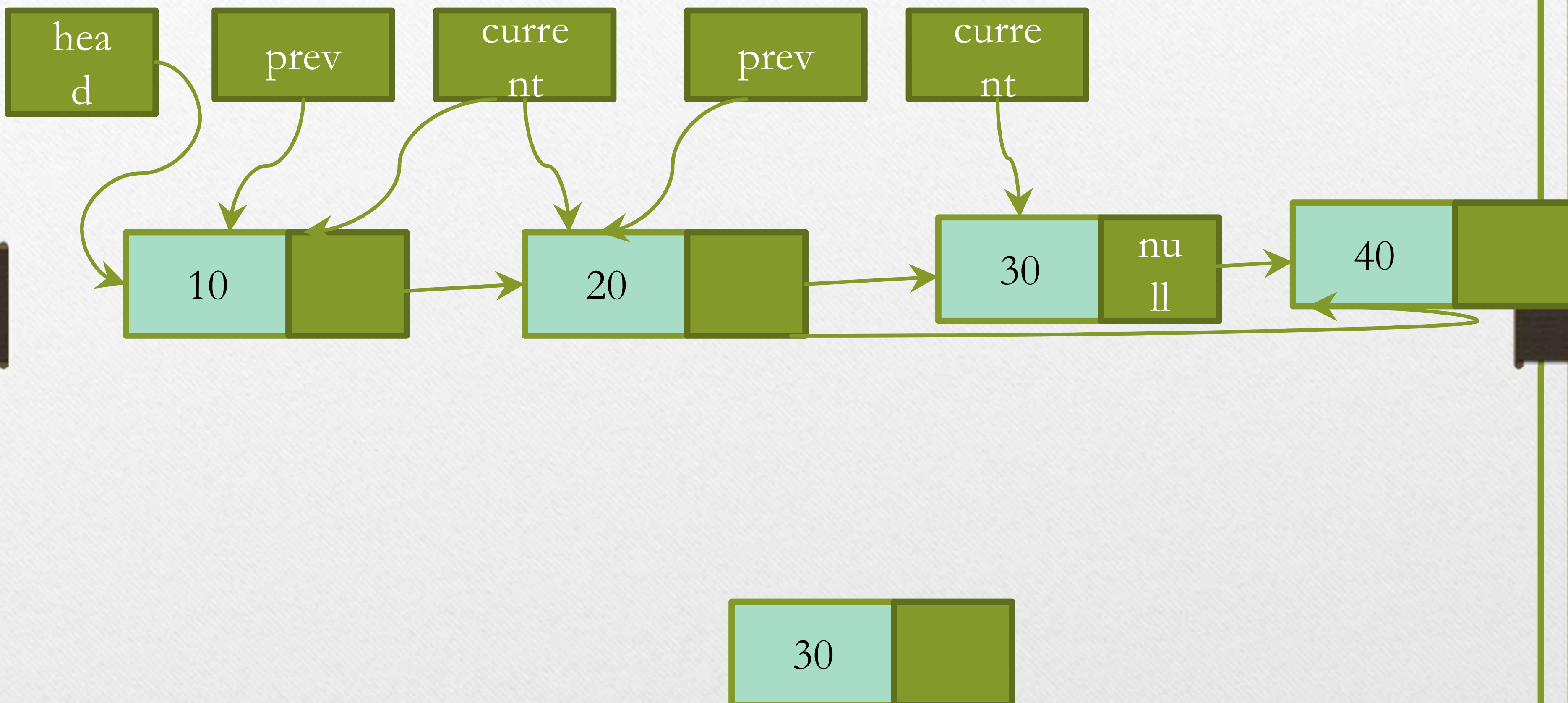
- Consider a linked list with 4 nodes



- We want to delete a node from the middle of the list

Deletion- From intermediate location

- Consider a linked list with 4 nodes



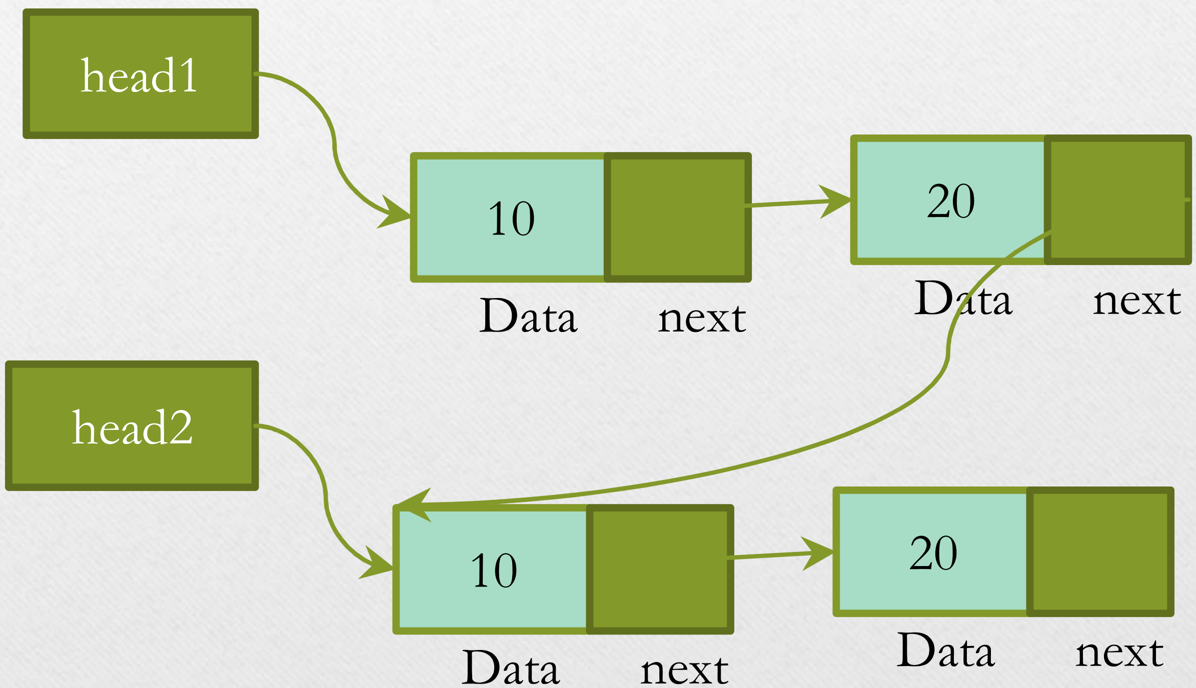
- We want to delete a node from the middle of the list

Delete a node key

1. Read the value key that is to be deleted //30
2. Set current=head, prev=head
3. Repeat while(current->link!=null)&& (current-> data!=key)
4. prev= current
5. current=current->link
6. If (current==null)
7. print “element not found”
8. Else
9. prev->link=current->link

Merging

- Two linked list L1 and L2.
- Merge L2 after L1



Merging

1. Set ptr= head1
2. While(ptr->link!= NULL)
3. ptr=ptr->link
4. ptr->link=head2
5. Head=head1
6. Stop