Linked List

Linked Lists

- Each element of the same type
- Each element stored in a node, along with a reference to its successor
- Each node allocated dynamically and accessed by reference
- No limit on length, subject to available computer memory
- Elements may be linked backwards as well as forwards

List Overview

- Basic operations of linked lists
  - Insert, find, delete, print, etc.
- Variations of linked lists
  - Circular linked lists
  - Doubly linked lists

Linked Lists

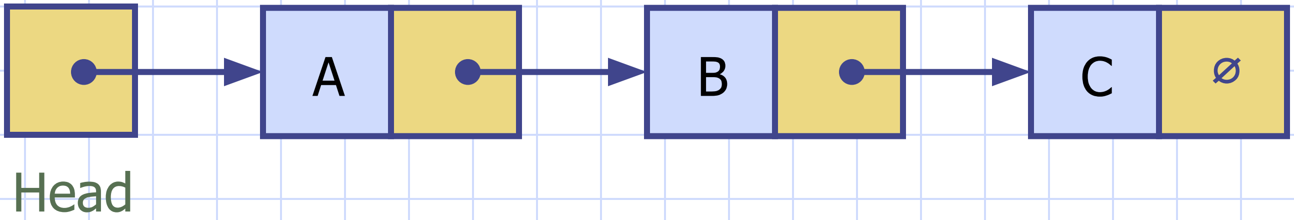
- A linked list is a very flexible dynamic data structure: items may be added to it or deleted from it at anywhere.
- Use a linked list instead of an array when
  - the number of data elements is unpredictable

Advantages of Linked List over Array

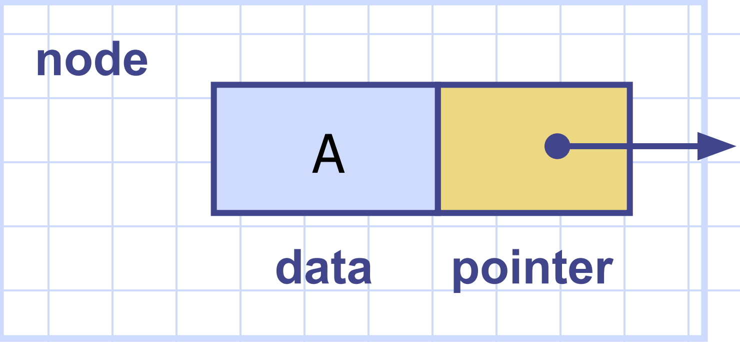
- Array have a fixed dimension.Once the size of an array is decided it can not be increased during execution.
- Array elements are always stored in contiguous memory locations .sometimes it happens that enough contiguous memory locations might not be available for the array that we are trying to create.
- Insertion and deletion in array is complex because it requires shifting of elements in array.

- It is a dynamic data structure,i.e.a linked list can grow and shrink in size during its lifetime.
- The nodes of linked list (elements) are stored at different memory locations it hardly happens that we fall short of memory when required.
- Insertion and deletion in linked list is easy because it does not requires shifting of elements in it.

Linked Lists

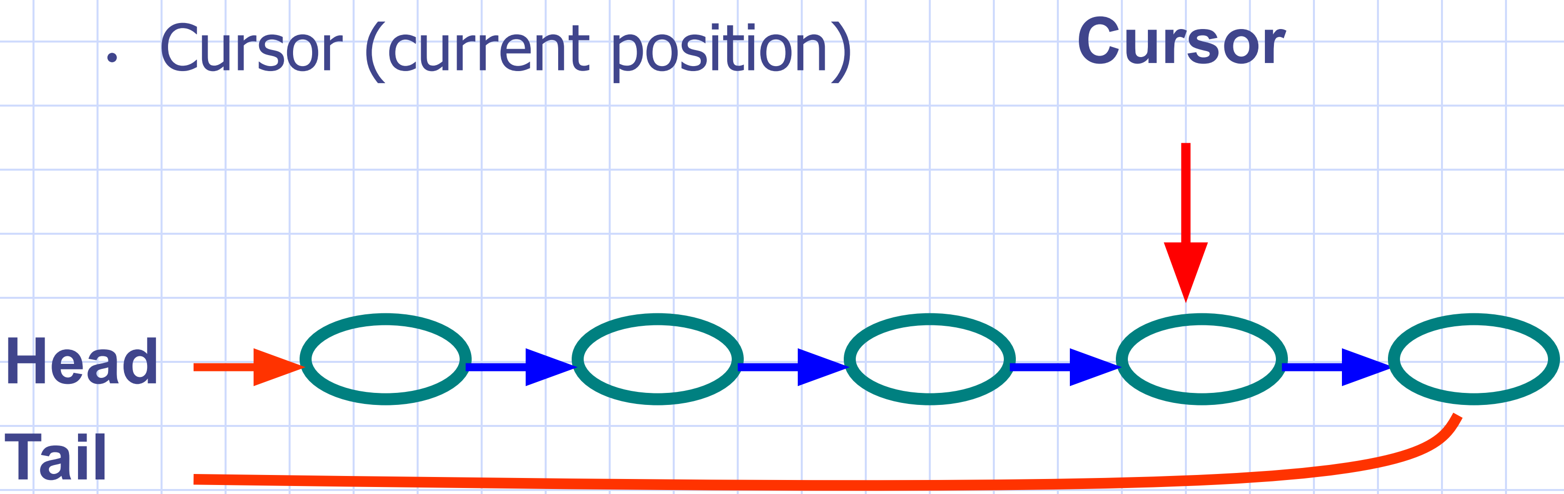


- A *linked list* is a series of connected *nodes*
- Stores a collection of items non-contiguously.
- Each node contains at least
  - A piece of data (any type)
  - Pointer to the next node in the list
- *Head*: pointer to the first node
- Must know where first item is
- The last node points to `NULL`



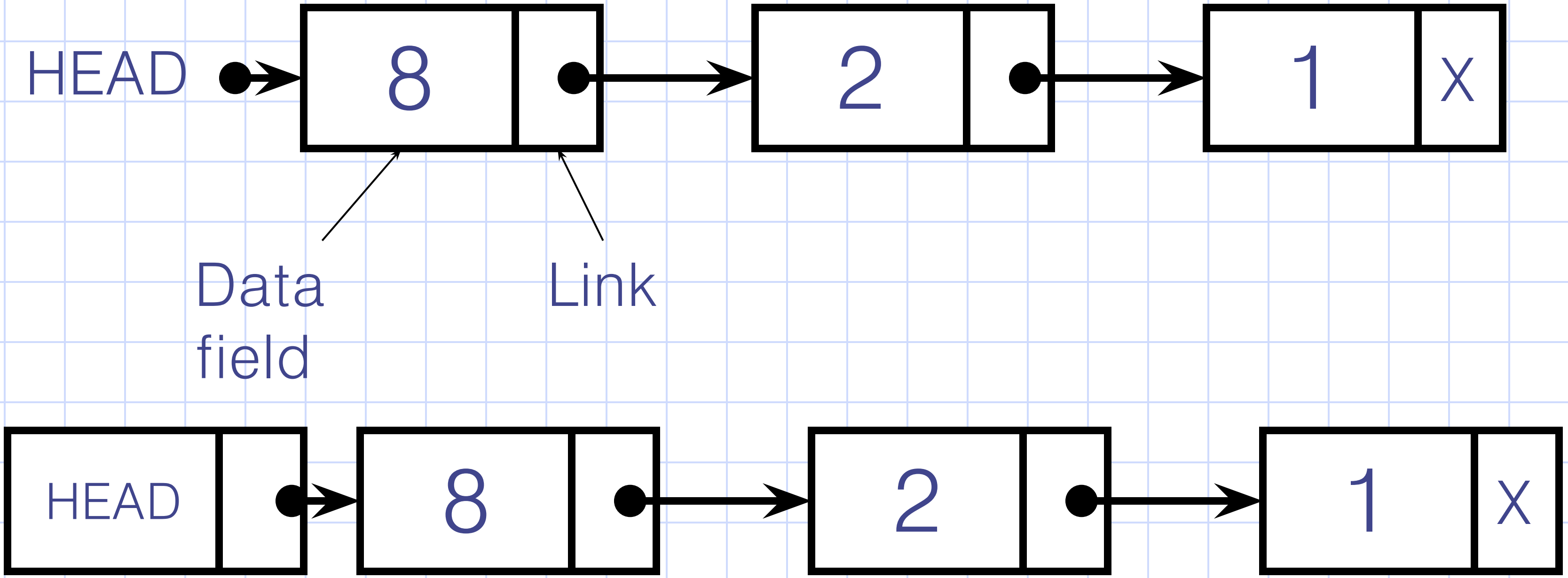
Linked List

- Properties
  - Elements in linked list are ordered
  - Element has successor
- State of List
  - Head
  - Tail
  - Cursor (current position)



Linked List Implementation

- 2 types of implementing ‘head’



Linked list creation algorithm

- Step:1[Initially list empty]

  start =NULL
- Step:2 [Allocate space to newly created node]

  node = create a node
- Step:3 [Assign value to the information part of the node]

  info[node] =data
- Step:4[Assign null to the address part for signaling end of the list]

  next[node] =start
- Step:5[Assign address of first node to start variable]

  start = node
- Step:6 [Return the created node]

  return(start)

Code for create SLL

```
void create(){
    newnode=(struct node *)malloc(sizeof(struct node));
    if(newnode==NULL){
     printf("\nNo enough memory available..");
     exit(0);
    }
    newnode->next=NULL;
    printf("\tEnter no:");
    scanf("%d",&newnode->data);
}
```

Algorithm to traverse nodes

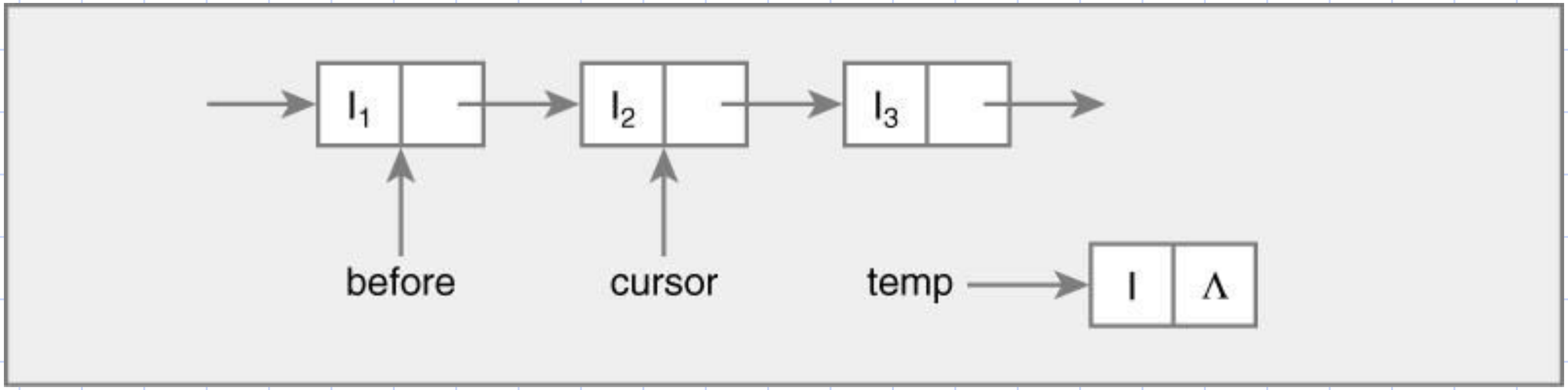
- Step:1[Initialize pointer variable current_node]
  current_node = first
- Step:2 [Perform the travering operation]
  Repeat while current_node != NULL
  Process Info[current_node]
- Step:3 [Move pointer to next node]
  current_node = next[current_node]
- Step:4 Exit

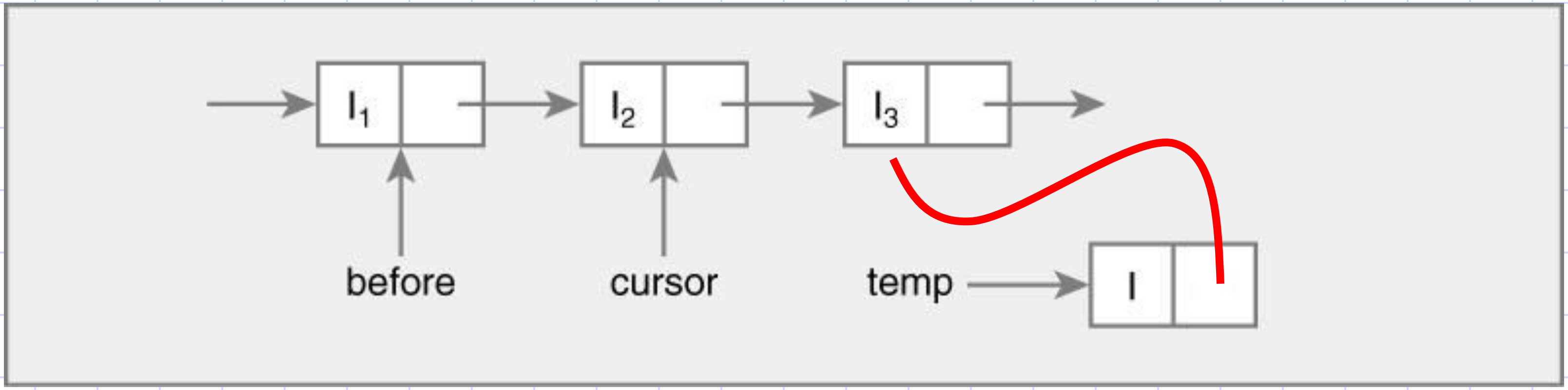
```
void display(){
    temp=head;
    printf("\n\nThe List Contents");
    if(head==NULL){
     printf("There is no link");
     return;
    }
    while(temp!=NULL){
     printf("\n\t\t\t|  %d  |",temp->data);
     printf("\n\t\t\t-------");
         temp=temp->next;
    }
    }
```

Inserting a new node

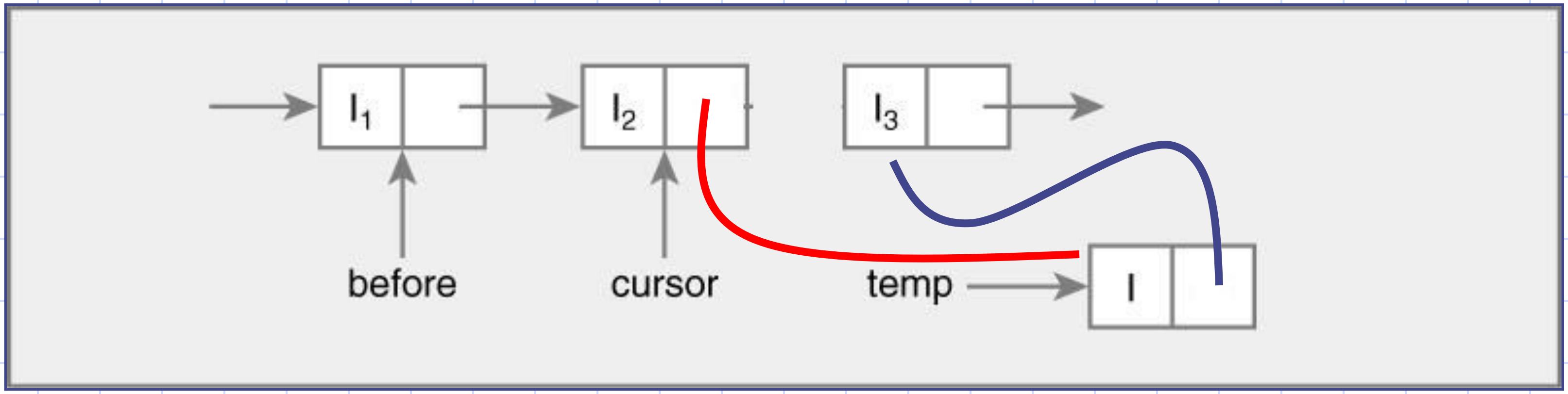
- Possible cases of `InsertNode`
  1. Insert into an empty list
  2. Insert in front
  3. Insert at back
  4. Insert in middle
- But, in fact, only need to handle two cases
  - Insert as the first node (Case 1 and Case 2)
  - Insert in the middle or at the end of the list (Case 3 and Case 4)

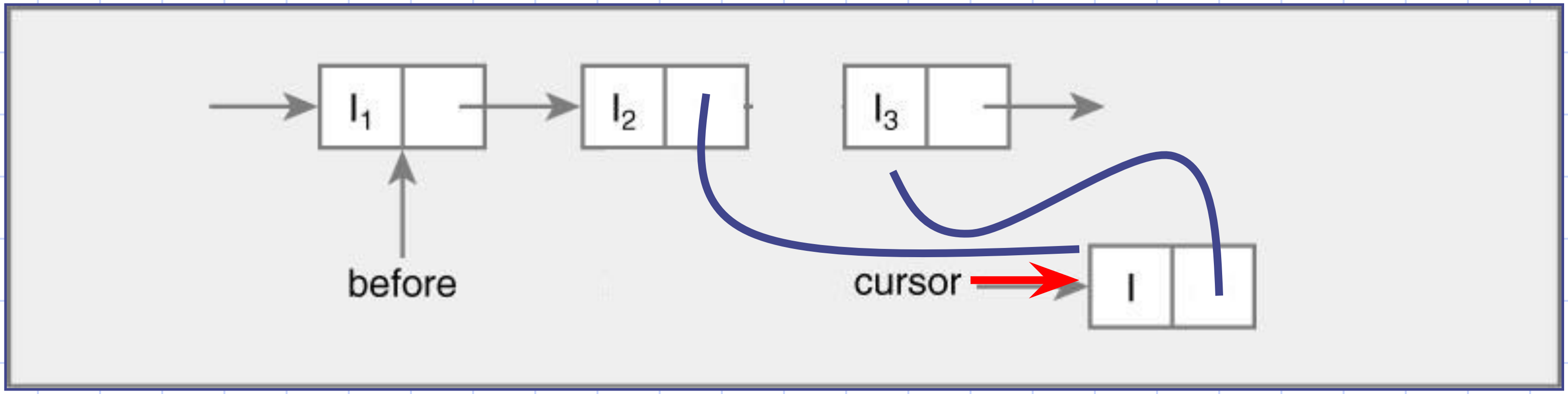
Linked List – Insert (After Cursor)



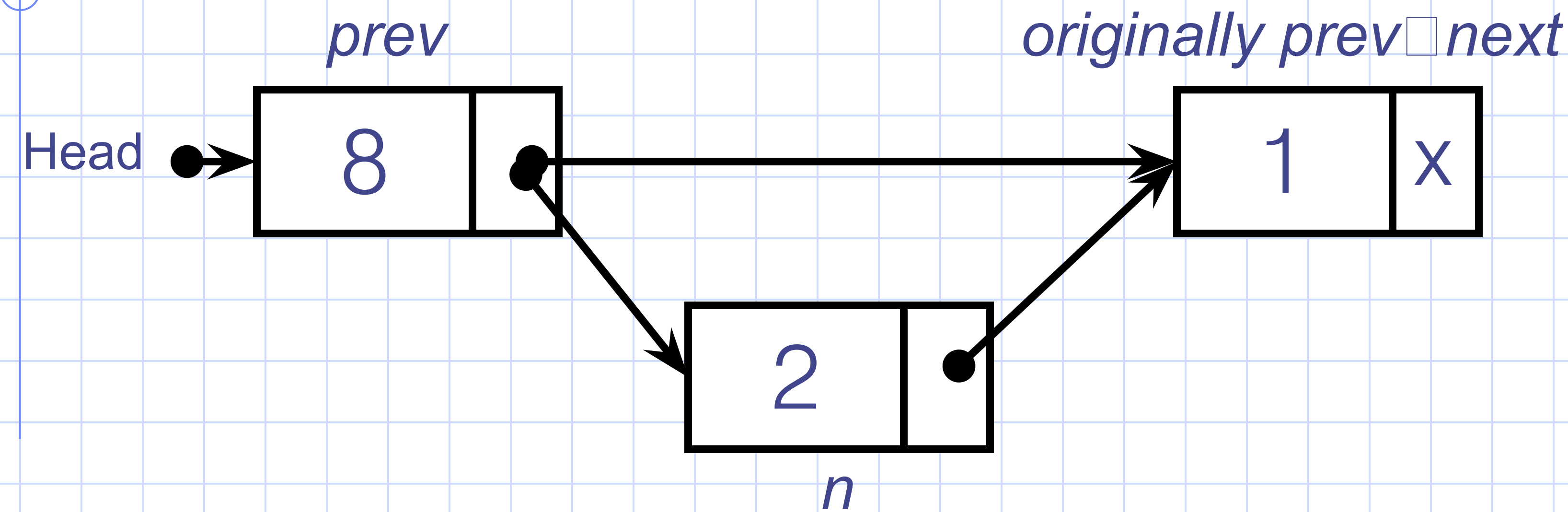


Linked List – Insert (After Cursor)





Primitive Functions – Insert



Insert a node at the beginning

- Step:1[Check for free space]

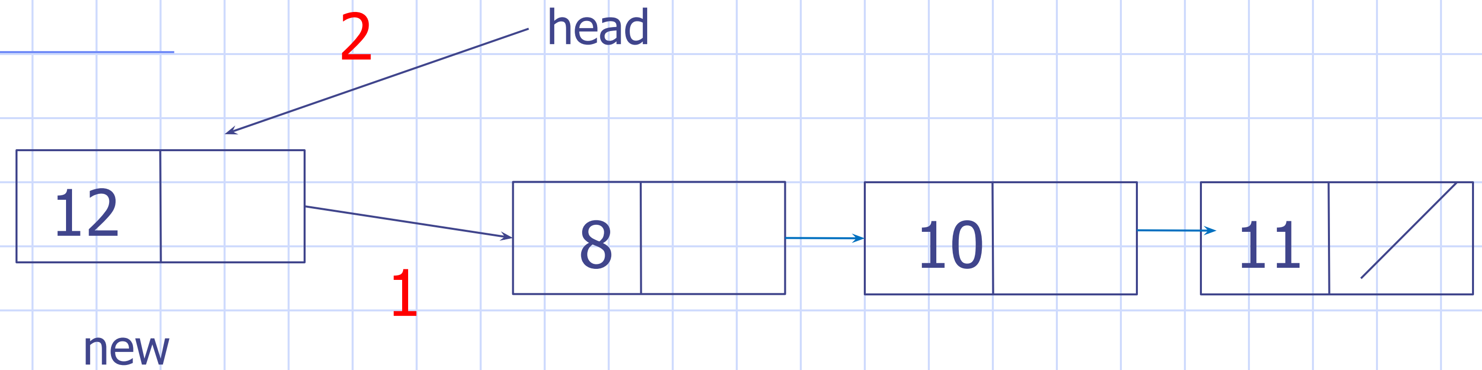
  If new = NULL output “OVERFLOW” and Exit
- Step:2[Allocate free space]

  new = create new node
- Step3:[Read value of information part of a new node]

  info[new] = value
- Step4:[Link address part of the currently created node with the address of start]

  next[new] =start
- Step:5[Now assign address of newly created node to the start]

  start = new
- Step:6 Exit




```
void addbeg(){
create();
    if(head==NULL){
        head=newnode;
        return;
    }
    newnode->next=head;
    head=newnode;
}
```

Insert a node at the last

- Step:1[Check for free space]

  If new = NULL output “OVERFLOW” and Exit
- Step:2[Allocate free space]

  new = create new node
- Step:3[Read value of information part of a new node]

  info[new] = value
- Step:4[Move the pointer to the end of the list]

  Repeat while node != NULL

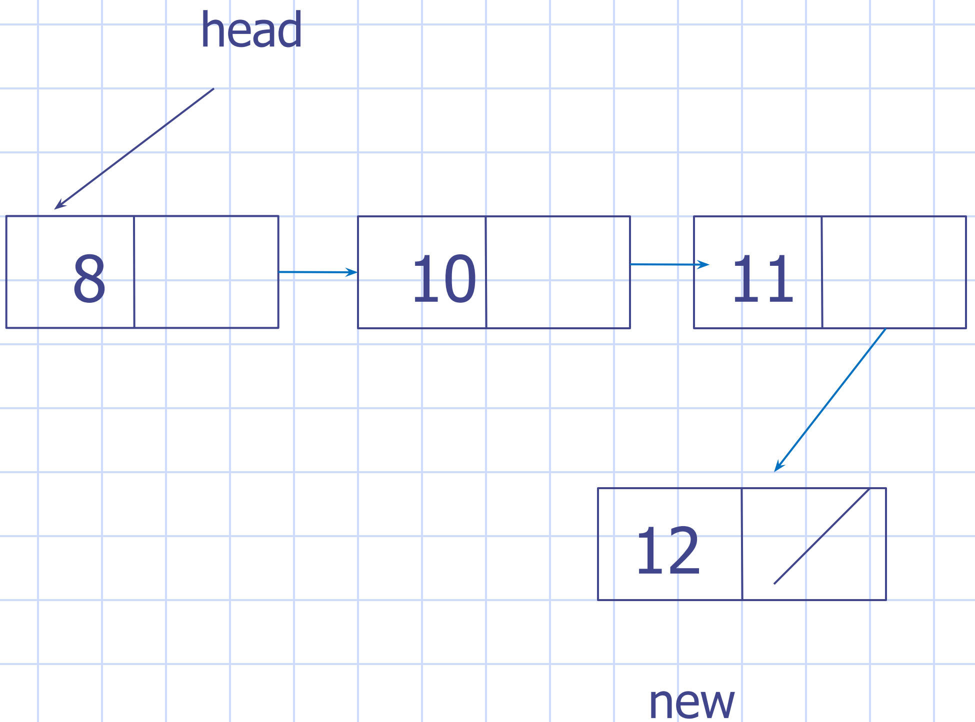
  node = next[node]

  previous =next[previous]
- Step:5[Link currently created node with the last node of the list]

  next[new] = node

  next[previous] = new
- Step:6 Exit

```
void addlast(){
    create();
    if(head==NULL){
        head=newnode;
        return;
    }
    temp=head;
    while(temp->next!=NULL)
        temp=temp->next;
    temp->next=newnode;
}
```



Insert a node at Specific location

- Step:1[Check for availability of space]

  If new=NULL output “OVERFLOW” and Exit

- Step:2[Initialization]

  node_number=0;

  node=start.next [points to first node of the list]

  previous = address of start[Assign address of start to previous]

- Step:3[Read node number that we want to insert]

  input insert_node

- Step:4[Perform insertion operation]

  Repeat through step 5 while node != NULL

- Step:5[Check if insert_node is equal to the node_number]

  If node_number+1 =insert_node

   next[new] = node

   previous->next =new

   info[new]=value

   return

  else[move the pointer forward]

   node=next[node]

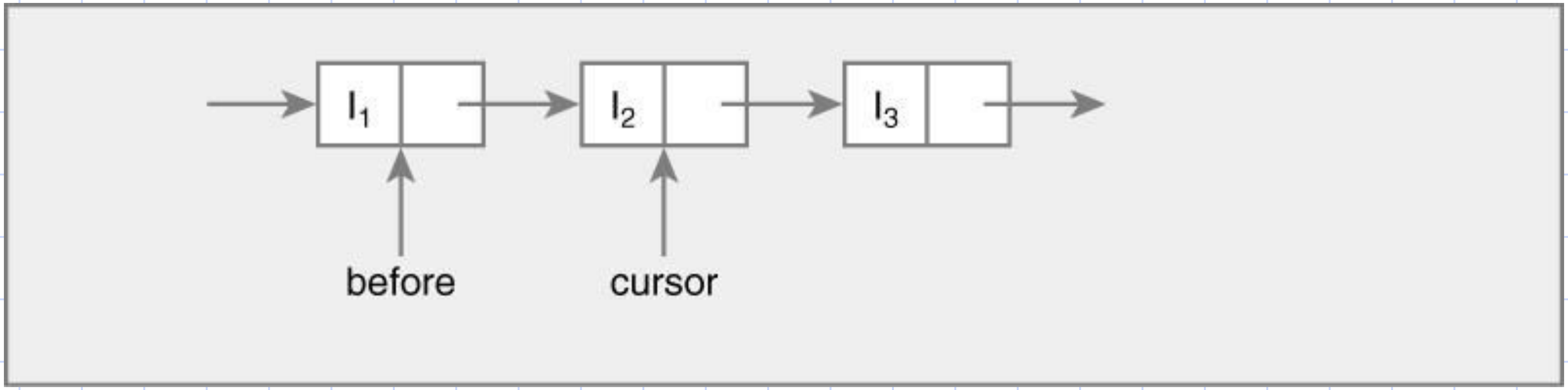
   previous = next[previous]

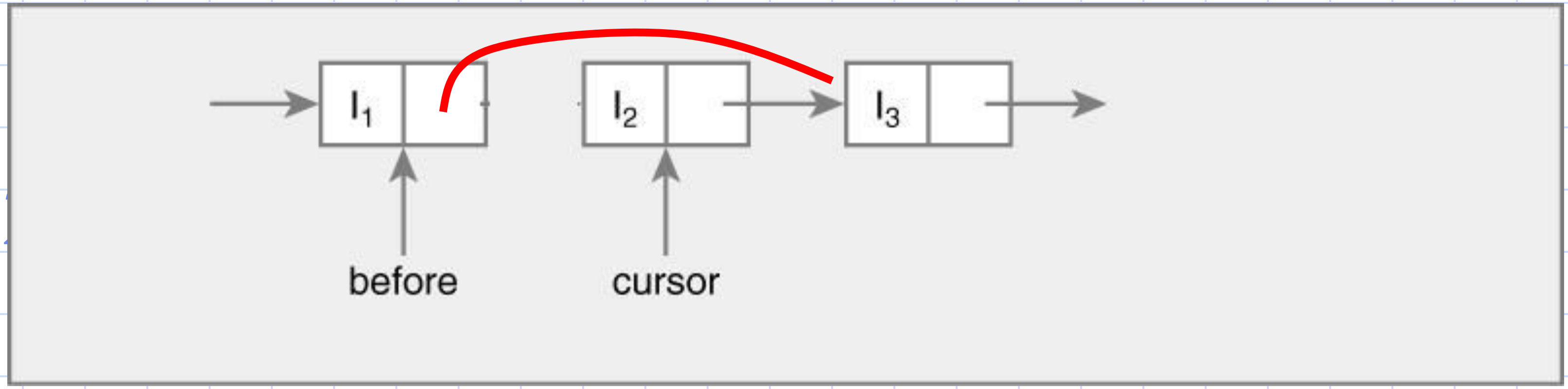
  node_number = node_number + 1

- Step:6 Exit

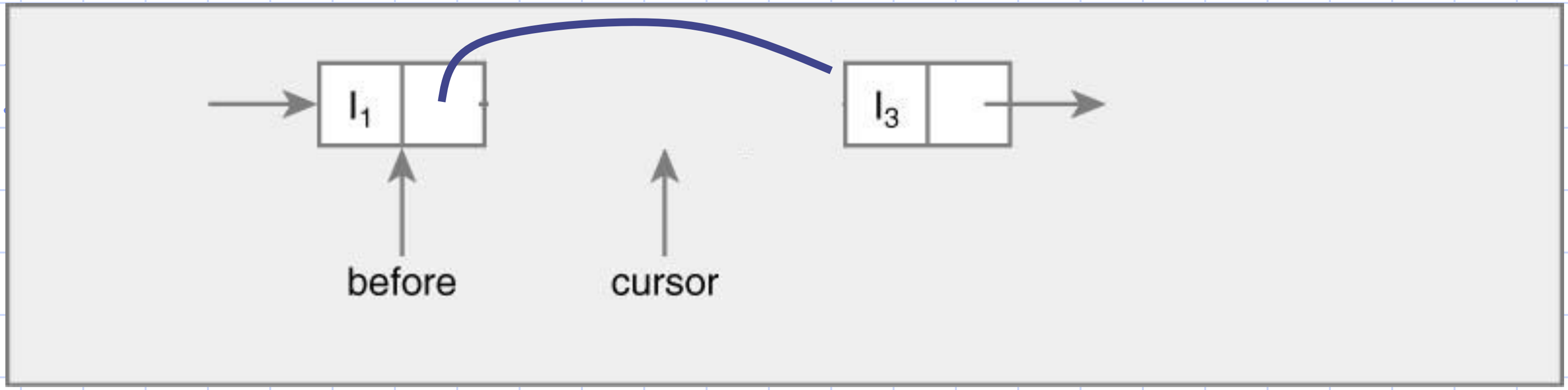
```
void addspec(int loc){
    int cnt=2;
    struct node *pred;
    temp=head;
    if(loc==1)
     addbeg();
    else
    {
    while(temp->next!=NULL && cnt!=loc){
     cnt++;
     temp=temp->next;  }
     create();
    newnode->next=temp->next;
    temp->next=newnode;
    }  }
```

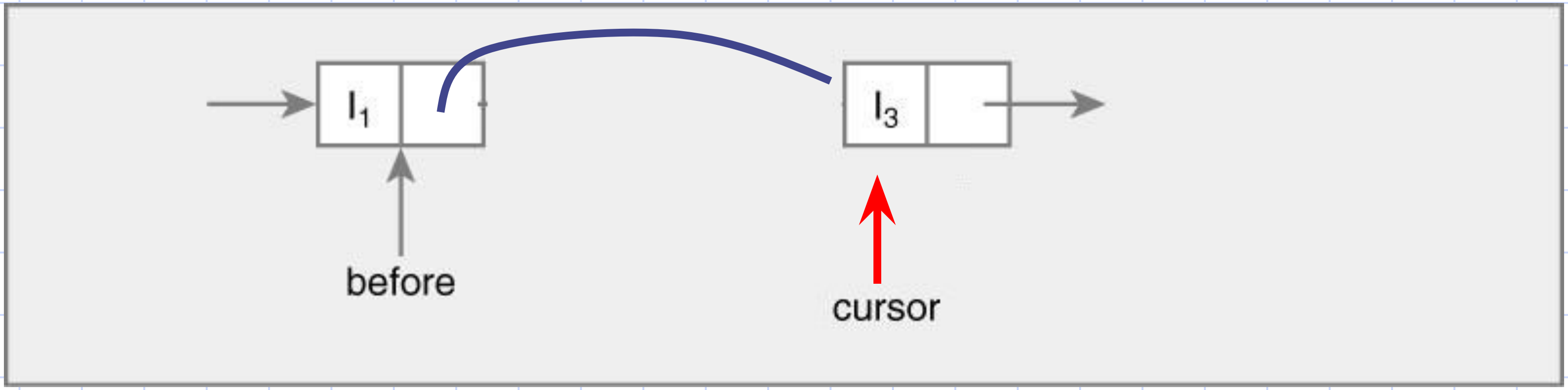
Linked List – Delete (Cursor)



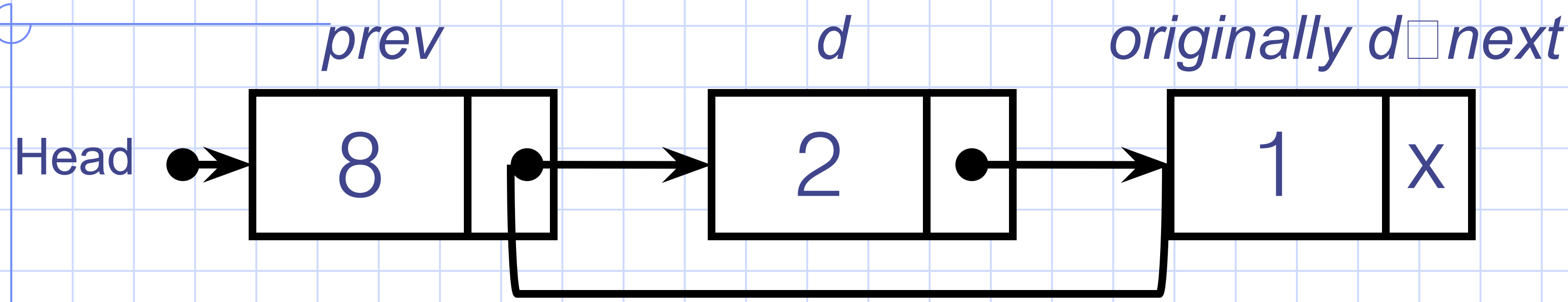


Linked List – Delete (Cursor)





Primitive Functions – Delete



Algorithm for Deletion of first node

- Step:1[Initializaion]

  node=start.next[points to the first node in the list]

  previous =assign address of start
- Step:2[Perform deletion operation]

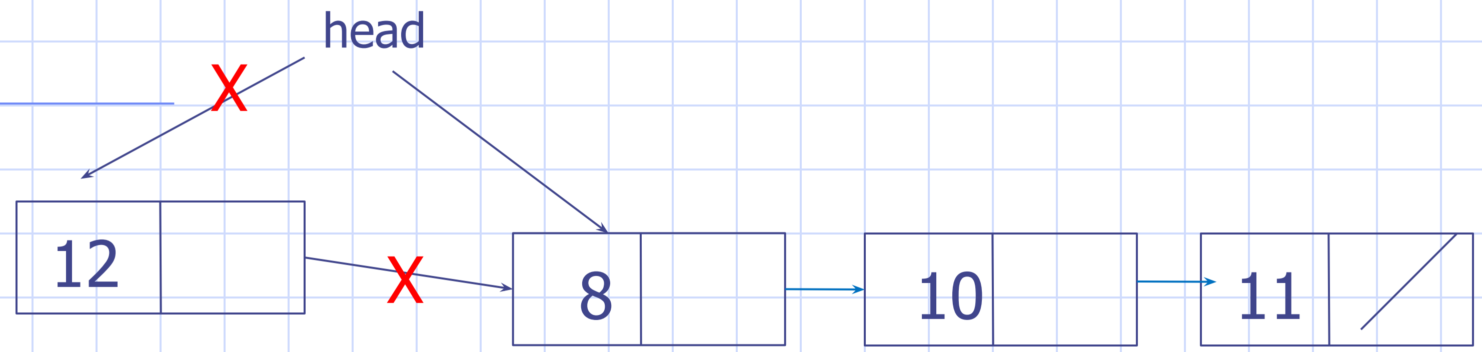
  if node = NULL

  output “Underflow” and Exit

  Else[delete first node]

  next[previous]=next[node][move pointer to next node in the list]

  Free the space associated with node
- Step:3 Exit




```
void deletfirst(){
    temp=head;
   if(head->next!=Null)
    head=head->next;
   else
     printf("only one node in list");
    free(temp);
}
```

Algorithm for Deletion of last node

- Step:1[Initializaion]

  node=start.next[points to the first node in the list]

  previous =assign address of start

  node_number =0
- Step:2[Check list is empty or not]

  if node = NULL

  output “Underflow” and Exit
- Step:3[Scan the list to count the number of nodes in the list]

  Repeat while node !=NULL

  previous=node

  node =next[node]

  node_number = node_number+1

- Step 4:

    Next[prev]=NULL

[END]

```
void deletend(){
    struct node *pred;
    temp=head;
    while(temp->next != NULL){
     pred=temp;
     temp=temp->next;
    }
    pred->next=NULL;
    free(temp);
}
```

Deletion of a desired node

- Step:1[Initialization]

  node=start.next [points to first node of the list]

  previous = address of start
- Step:2[Initialize node counter]

  node_number=1;
- Step:3 [Read node number]

  delete_node=value
- Step:4[Check list is empty or not]

  If node=NULL output "UNDERFLOW" and Exit

- Step:5[Perform deletion operation]

  Repeat through step 6 while node != NULL

  if node_number =delete_node

   i) next[previous] = next[node][make link of previous node to the next node]

   ii) delete(node) [Delete current node]

   iii) Exit

  else[move the pointer forward]

    Previous=node;

        node=next[node]

  Step:6 node_number = node_number + 1

- Step:7 Exit

```
void deletpos(int pos){
    int cnt=1;
    struct node *pred;
    temp=head;
    if(pos==1)
     deletfirst();
    else
    {
    while(temp->next!=NULL && cnt!=pos){
     cnt++;
     pred=temp;
     temp=temp->next;
    }
    pred->next=temp->next;
    free(temp);
    }
}
```

Searching algorithm

- Step:1[Initialization]

  node=start.next[points to the first node of the list]

  previous=address of first node in the lsit

  node_number=1
- Step:2[set the flag]

  flag=0
- Step:3[Read the information of a node to which we want to search]

  search_node=value
- Step:4[Check the list]

  if node =NULL

  Output “List is empty” and Exit

- Step:5[Perform search operation]

  Repeat through step6 while node!= NULL

  If info[node]= search_node

  i) output "search is succesfull" and position is equal to value of node_number

  // ii) node = next[node]

  // iii) previous =next[node]

  iv) flag=1

  else

  i) node= next[node]

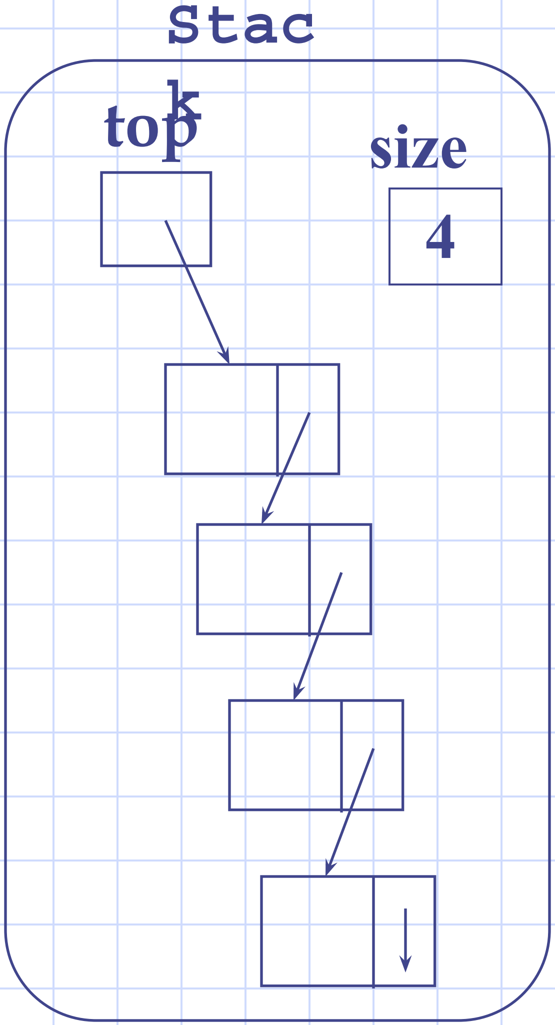
  ii) previous = next[previous]
- Step:6[Increment the value of node_number]

  node_number = node_number+1
- Step:7[check the flag]

  If flag=0 output "search is successful"
- Step:8 Exit

Singly list

Linked list as Stack data structure



Stack link

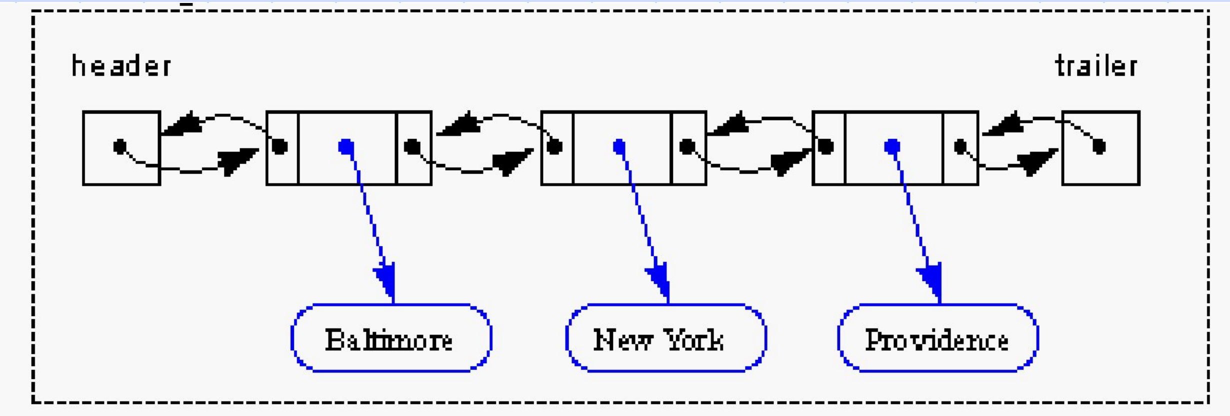
Linked Lists

- Types of linked lists:
  - *singly linked list*
    - begins with a pointer to the first node
    - terminates with a null pointer
    - only traversed in one direction
  - *circular, singly linked*
    - pointer in the last node points back to the first node
  - *doubly linked list*
    - two “start pointers”- first element and last element
    - each node has a forward pointer and a backward pointer
    - allows traversals both forwards and backwards
  - *circular, doubly linked list*
    - forward pointer of the last node points to the first node and backward pointer of the first node points to the last node

Doubly Linked List

- Keep a pointer to the next and the previous element in the list

```
typedef struct node *pnode;
    typedef struct node {
            char data [10];
            pnode next;
            pnode prev;
    }
```

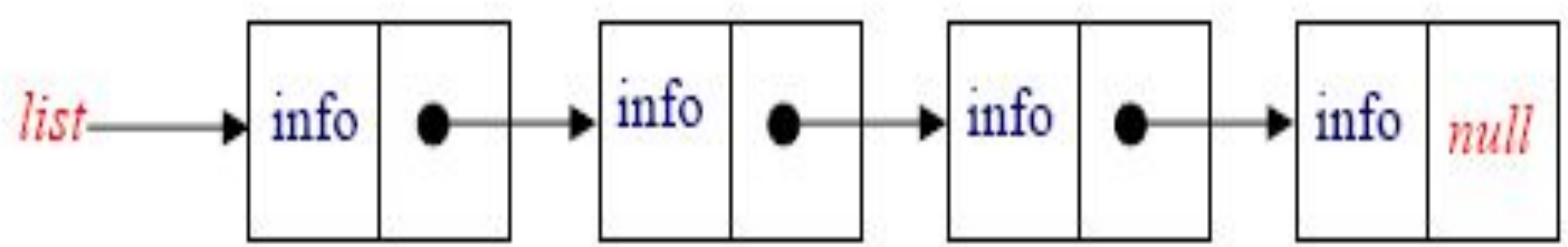


Doubly link

Circular Linked Lists

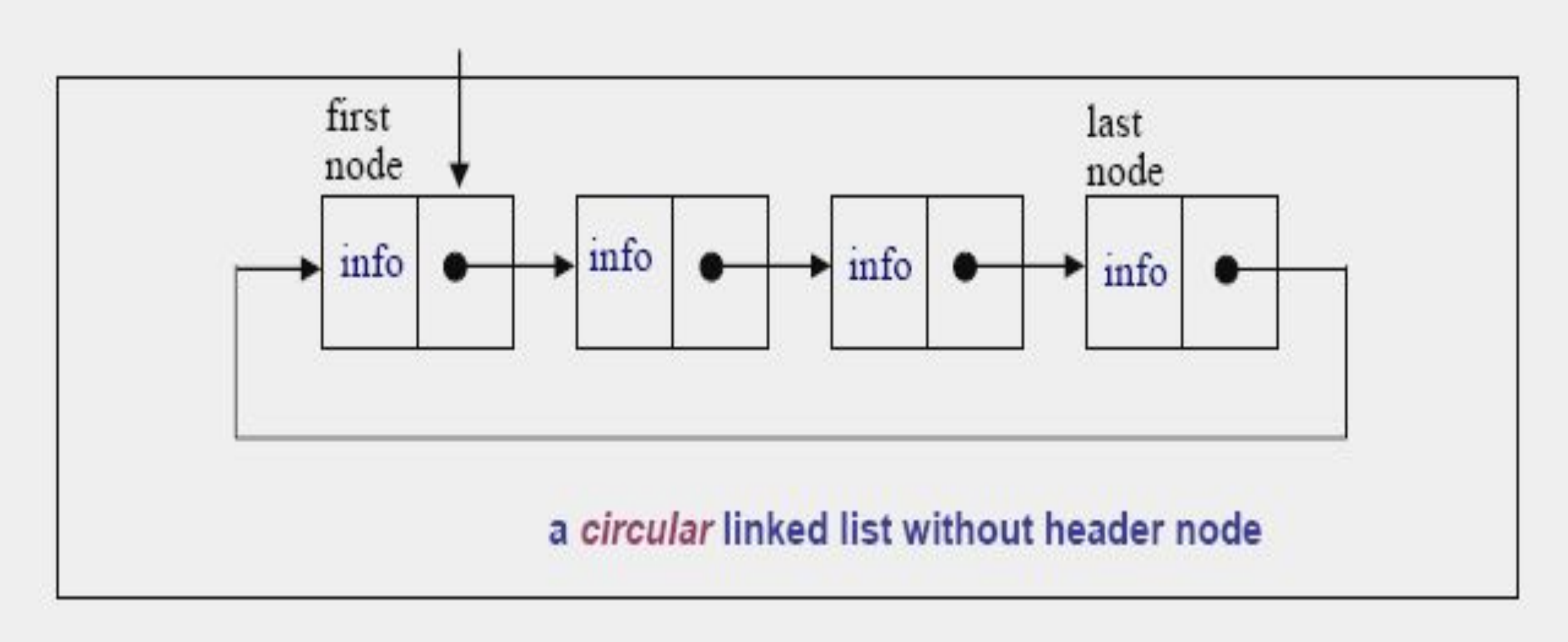
- In linear linked lists if a list is traversed (all the elements visited) an external pointer to the list must be preserved in order to be able to reference the list again.
- Circular linked lists can be used to help the traverse the same list again and again if needed. A circular list is very similar to the linear list where in the circular list the pointer of the last node points not NULL but the first node.

Circular Linked Lists

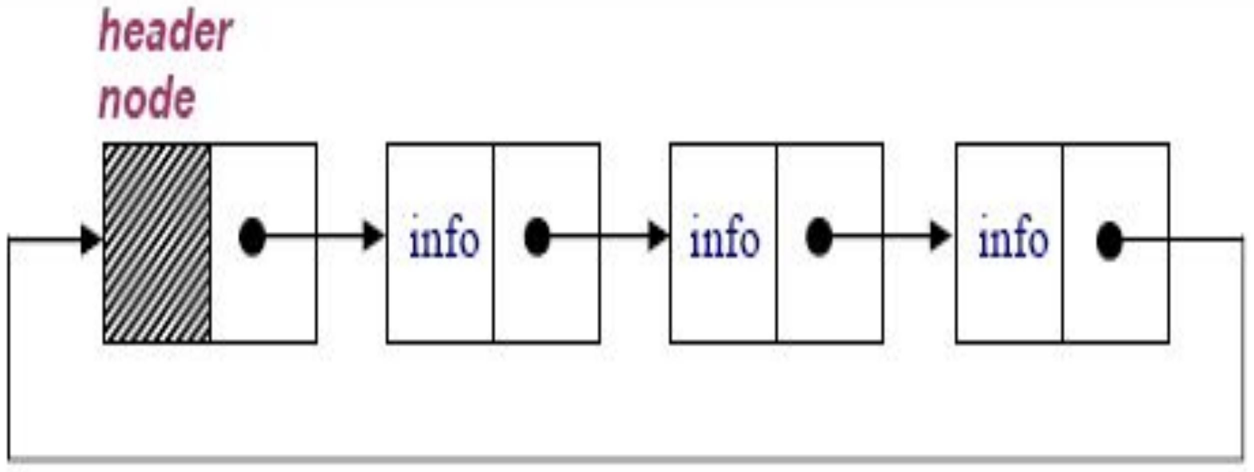


A Linear Linked List

Circular Linked Lists



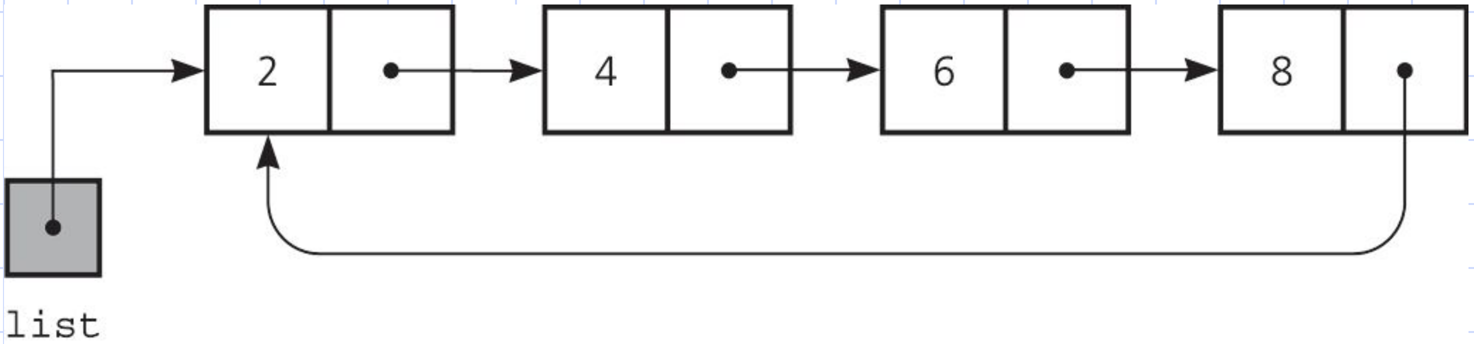
Circular Linked Lists



a *circular* linked list with a *header node*

Circular Linked Lists

- Last node references the first node
- Every node has a successor
- No node in a circular linked list contains *NULL*



Circular Linked Lists

- In a circular linked list there are two methods to know if a node is the first node or not.
  - Either an external pointer, ***list***, points the first node or
  - A ***header node*** is placed as the first node of the circular list.
- The header node can be separated from the others by having a ***sentinel value*** as the info part.

Circular linked list structure

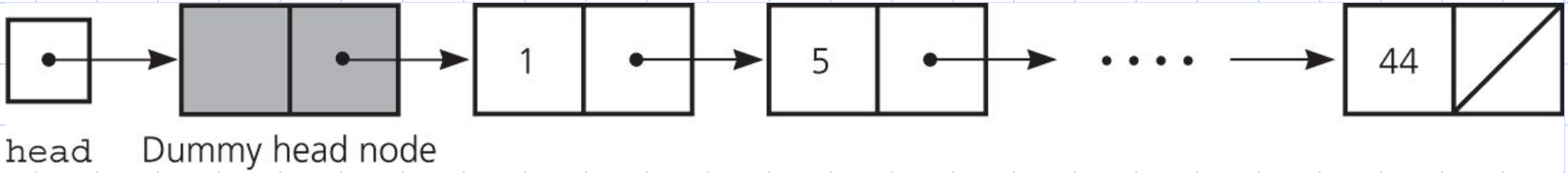
- The structure definition of the circular linked lists and the linear linked list is the same:

```
struct node{
    int info;
    struct node *next;
};
typedef struct node *NODEPTR;
```

Circular_list

Dummy Head Nodes

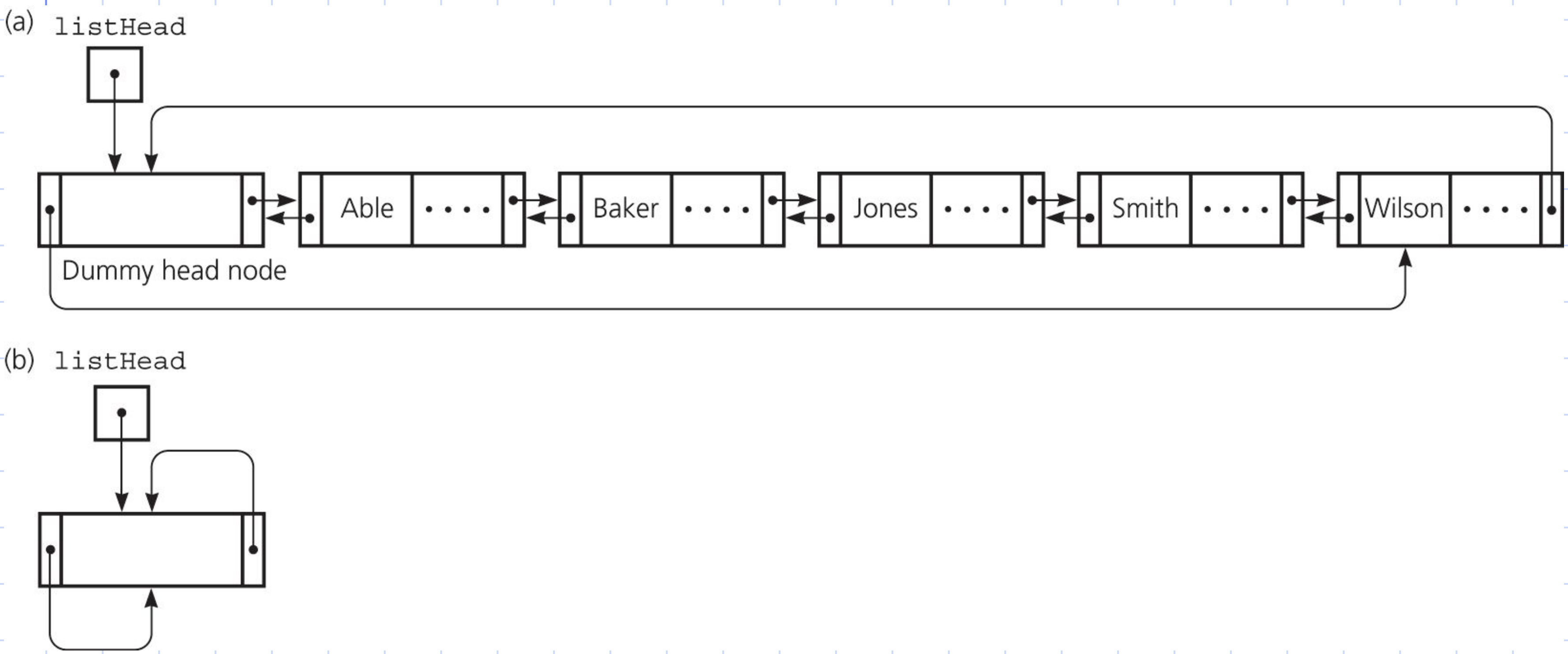
- Dummy head node
  - Always present, even when the linked list is empty
  - Insertion and deletion algorithms initialize `prev` to reference the dummy head node, rather than *NULL*



Circular Doubly Linked Lists

- Each node points to both its predecessor and its successor
- Circular doubly linked list
  - `precede` pointer of the dummy head node points to the last node
  - `next` reference of the last node points to the dummy head node
  - No special cases for insertions and deletions

Circular Doubly Linked Lists



(a) A circular doubly linked list with a dummy head node

(b) An empty list with a dummy head node