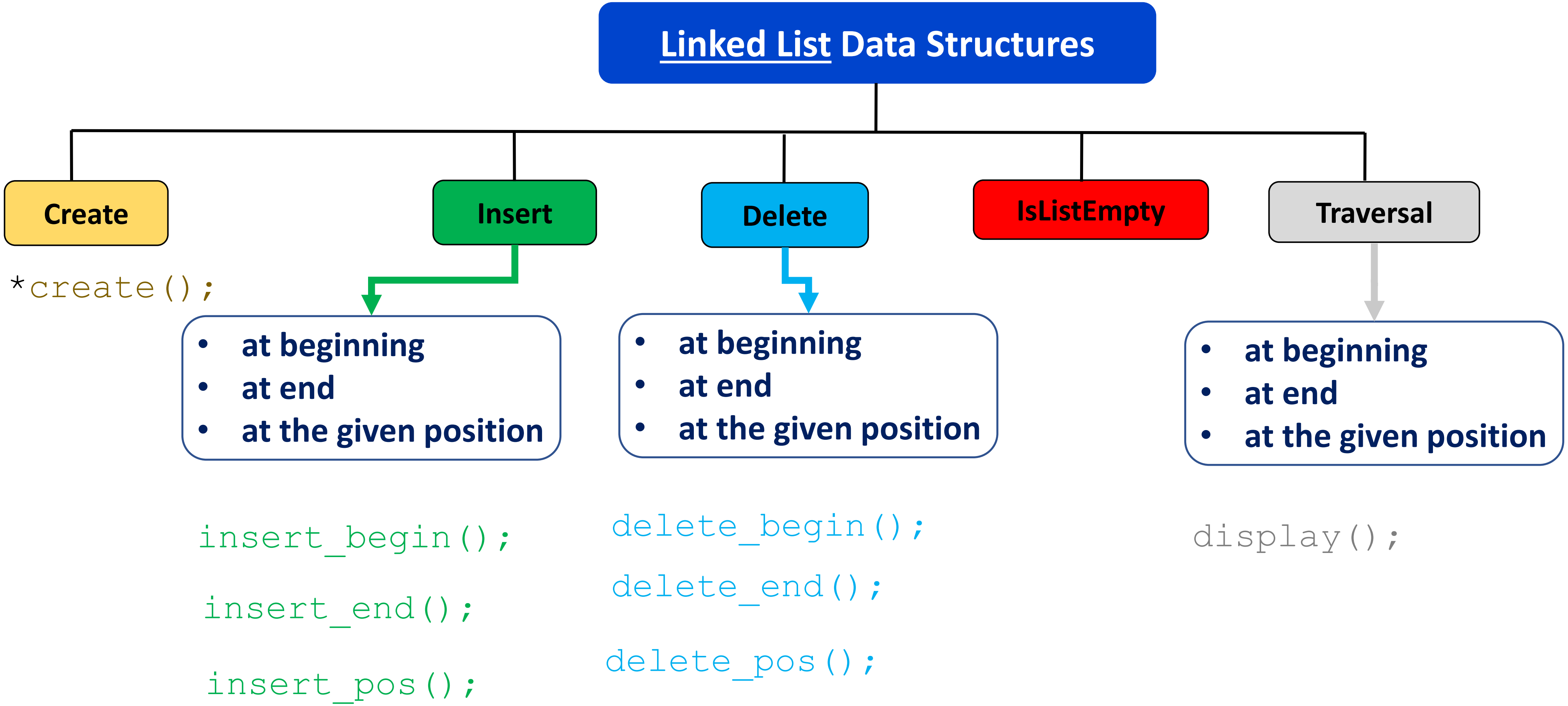
CS2x1:Data Structures and Algorithms

Koteswararao Kondepu

k.kondepu@iitdh.ac.in

Recap: Linked List Operations



Linked List Data Structures

Create

```
*create();
```

Insert

- **at beginning**
- **at end**
- **at the given position**

```
insert_begin();

insert_end();

insert_pos();
```

Delete

- **at beginning**
- **at end**
- **at the given position**

```
delete_begin();

delete_end();

delete_pos();
```

IsListEmpty

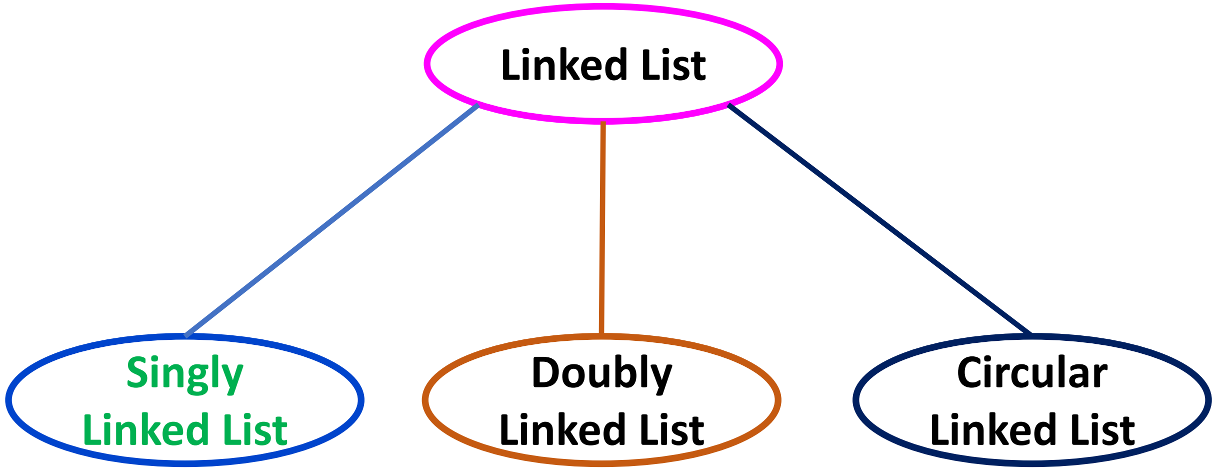
Traversal

- **at beginning**
- **at end**
- **at the given position**

```
display();
```

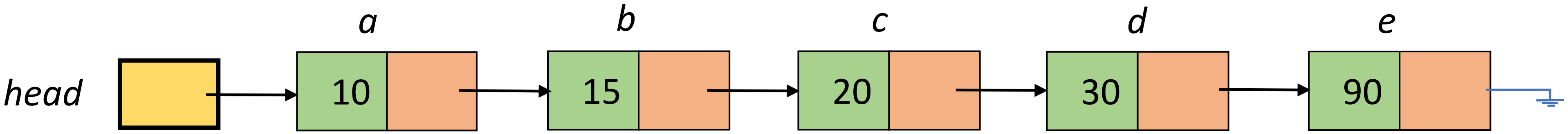
Outline

- Limitation of Singly Linked List
- Exercise on Singly Linked List
- Doubly Linked List Operations
- Exercise on Doubly Linked List
- Circular Linked List Operations
- Exercise on Circular Linked List



Exercise: Singly Linked List (1)

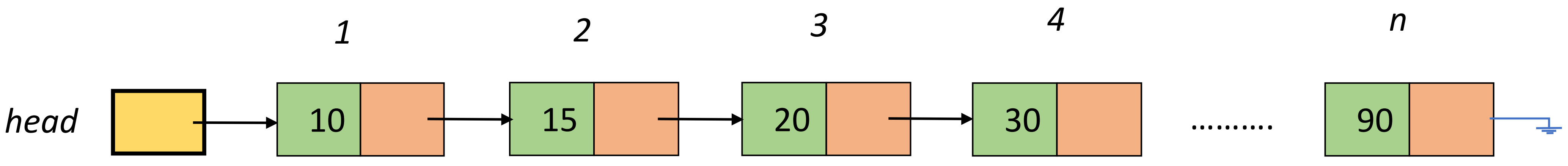
```
struct sll{
        int data;
        struct sll *next;
};
int a[5];
void cs201()
{
    struct sll *sllnewnode=malloc(sizeof(struct sll));
    printf("%d\n", sizeof(a));
    printf("%d\n", sizeof(sllnewnode));
    printf("%d\n", sizeof(sllnewnode)*5);
}
int main() {
cs201();
return 0;
}
```



Linked Lists may use more memory than the arrays

Exercise: Singly Linked List (2)

```
struct sll{
        int data;
        struct sll *next;
};
int a[5]= {10, 15, 20, 25, 30};
```



What is the time taken to access the kth element in the given array and the given linked list?

(i) O(1); O(1)
(ii) O(k); O(k)
(iii) O(1); O(k)
(iv) O(k); O(1)

Nodes are stored ***in-contiguously***, the time required to access individual elements greatly increased within the list

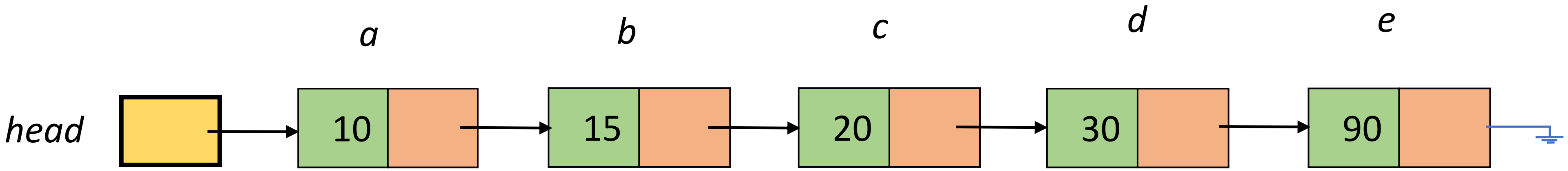
Exercise: Singly Linked List (3)

```
struct sll{
        int data;
        struct sll *next;
};
int a[5]= {10, 15, 20, 25, 30};
```

```
printf("%d\n", a);
printf("%d\n", a+1);
printf("%d\n", a+2);
printf("%d\n", a+3);
printf("%d\n", a+4);
```

	30	4
	25	3
	20	2
	15	1
&a=8288	10	0

Nodes are not stored *in-contiguously*, the time required to access individual elements greatly increased within the list



```
void create(){
    struct sll *aa=malloc(sizeof(struct sll));
    printf("%ld\n", aa);
}
```

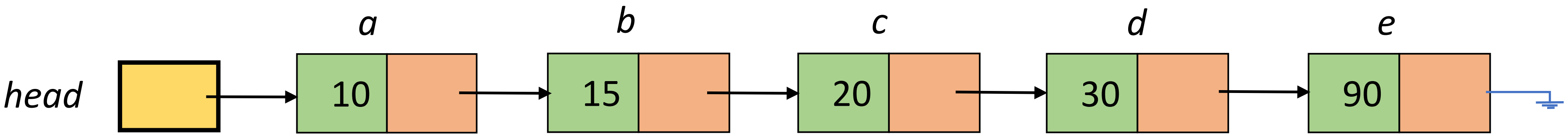
```
create();
create();
create();
create();
create();
```

Exercise: Singly Linked List (4)

```
struct sll{
        int data;
        struct sll *next;
};
int a[5]= {10, 15, 20, 25, 30};

  printf("%d\n", *(a+4));
  printf("%d\n", *(a+3));
  printf("%d\n", *(a+2));
  printf("%d\n", *(a+1));
  printf("%d\n", *a);
```

	30	4
	25	3
	20	2
	15	1
&a=8288	10	0

Difficulties arises in linked-list when it comes to reverse traversing



```
void create(){
    struct sll *aa=malloc(sizeof(struct sll));
    printf("%ld\n", aa);
}
```

```
create();
create();
create();
create();
create();
```

```
struct node *tail
tail = head;
while ( tail → next != NULL)
        tail = tail → next
```

Exercise: Singly Linked List (4)

Consider the following function to traverse a linked:

```
void traverse(struct Node *head)
{
   while (head->next != NULL)
   {
       printf("%d  ", head->data);
       head = head->next;
   }
}
```

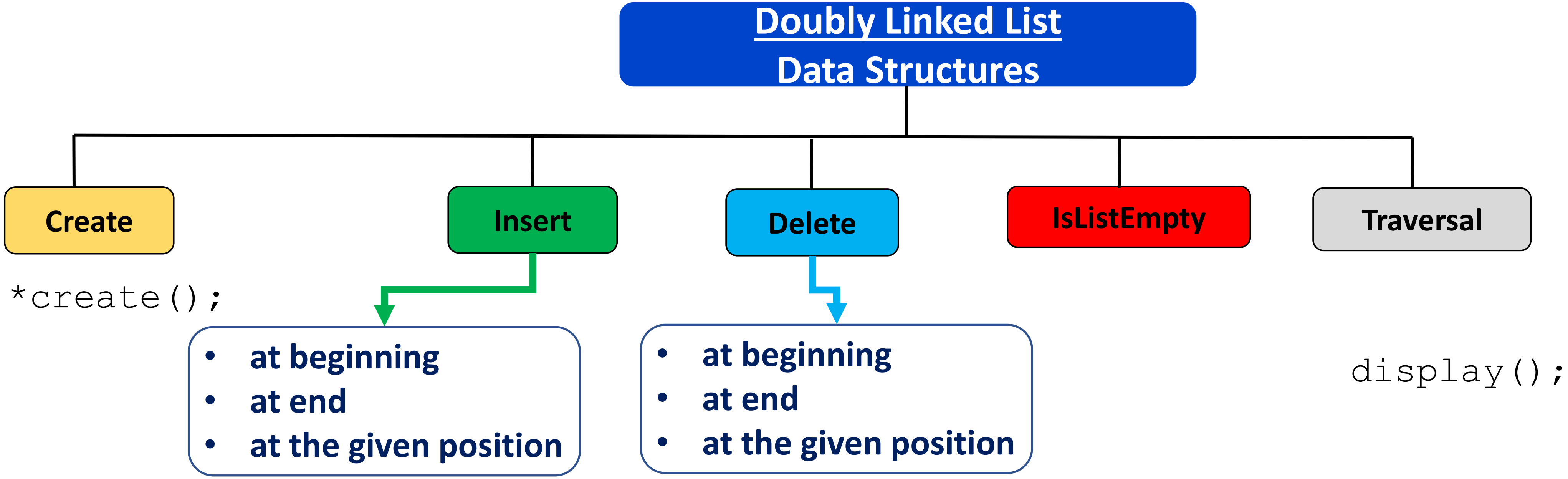
Which of the following is FALSE about above function?

A. The function may crash when the linked list is empty
B. The function doesn't print the last node when the linked list is not empty
C. The function is implemented incorrectly because it changes head
D. All of the above

Limitations: Singly Linked List

- Limitations:
  a) Linked Lists may use more memory than the arrays
  b) Nodes in a linked-list are accessed in order from beginning, thus the linked lists are inherently sequential access → no direct access
  c) Nodes are stored *in-contiguously*, the time required to access individual elements greatly increased within the list
  d) Difficulties arises in linked-list when it comes to reverse traversing

Doubly Linked List: Operations



Doubly Linked List
Data Structures

Create

```
*create();
```

Insert

- **at beginning**
- **at end**
- **at the given position**

Delete

- **at beginning**
- **at end**
- **at the given position**

IsListEmpty

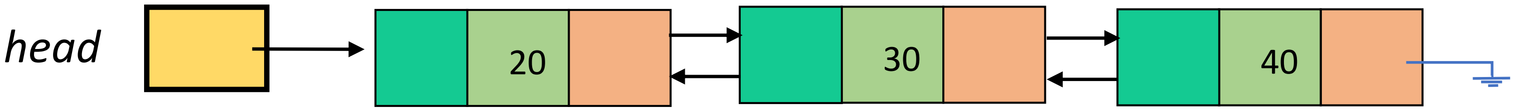
Traversal

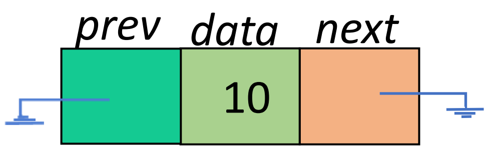
```
display();
```

```
insert_begin();

insert_end();

insert_pos();
```

```
delete_begin();

delete_end();

delete_pos();
```

```
struct node {
        struct node *prev;
        int data;
        struct node *next;
  }
```

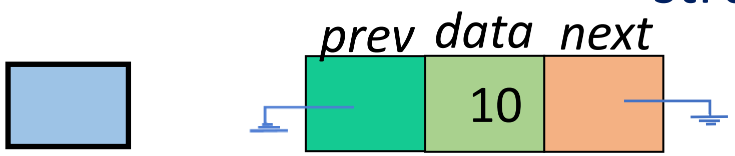




Doubly Linked List: Insert at the beginning or Insert at the head

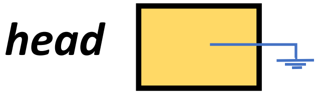
Steps:

(i) Creating a node with data



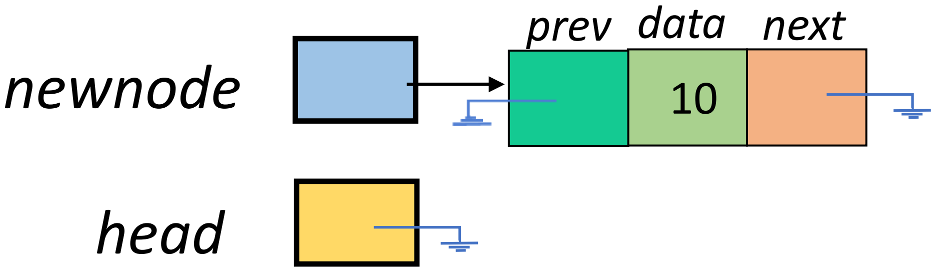
```
struct node *newnode = malloc (sizeof (struct node));
newnode → data = 10;   //Entering data
newnode → next = NULL; //making node next to NULL
newnode → prev = NULL;
```

```
struct node {
        struct node *prev;
        int data;
      struct node *next;
}
```



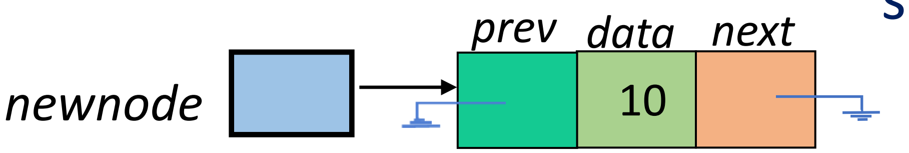
struct node *head = NULL

(ii) Adding a node to an empty linked list



```
if (head == NULL)
    head = newnode
```

Doubly Linked List: Insert at the beginning or Insert at the head



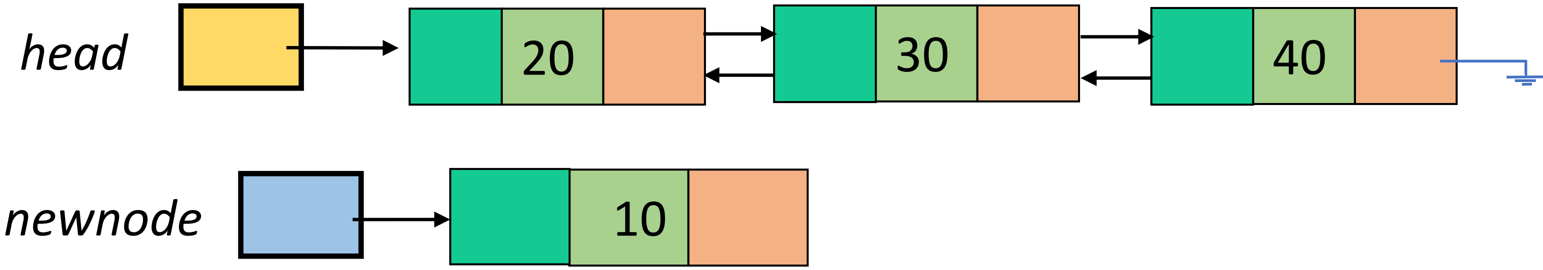
```
struct node {
        struct node *prev;
        int data;
      struct node *next;
}
```

Steps:

(iii) Adding a node to the beginning of a linked list

a) Update the next pointer of new node → the current head, and the current head previous → new node

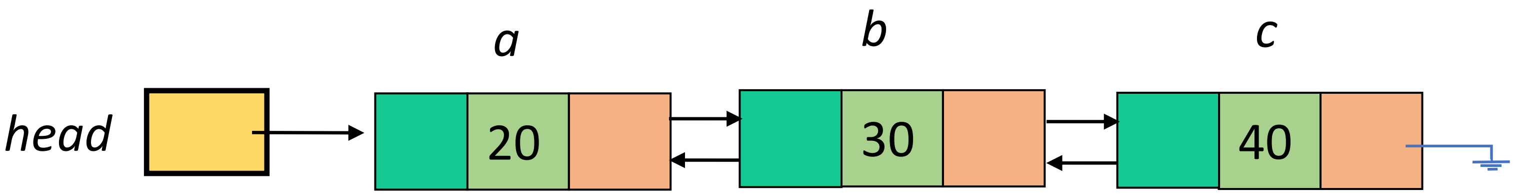
newnode →next = head
head →prev = newnode



b) Update the head pointer to the new node

head = newnode

Doubly Linked List: Example

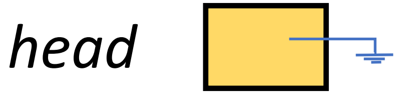


- *head →data = ?*
- *head →next →next →data = ?*
- *c →prev →prev →data = ?*
- *head →next →next →next →data = ?*
- *c →prev →prev →prev →data = ?*

Doubly Linked List: traversal or display

Steps:

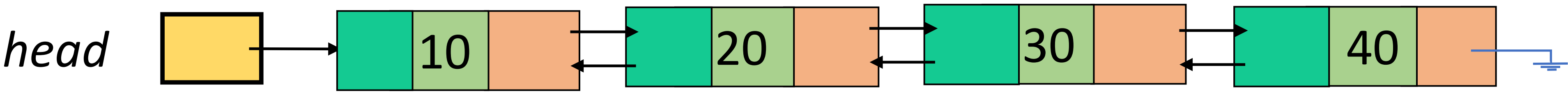
(i) Check if the linked list empty or not



*struct node *head = NULL*

```
if (head == NULL)
    printf ("Linked List is Empty\n");
```

(ii) List Traversal: Each node present in the list must be visited and display the data value

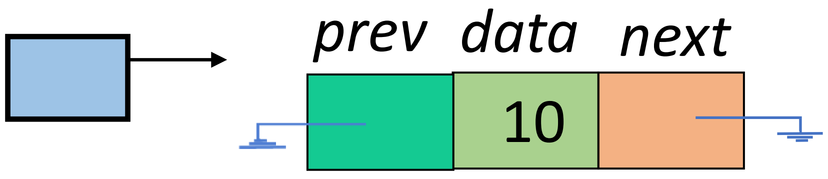


1 *struct node *traversal*

2 *traversal = head;*

3
```
while ( traversal != NULL)
        display the element: traversal → data
        traversal = traversal →next
```

Doubly Linked List: Insert at the end or Insert at the tail



```
struct node {
            struct node *prev;
            int data;
            struct node *next;
      }
```

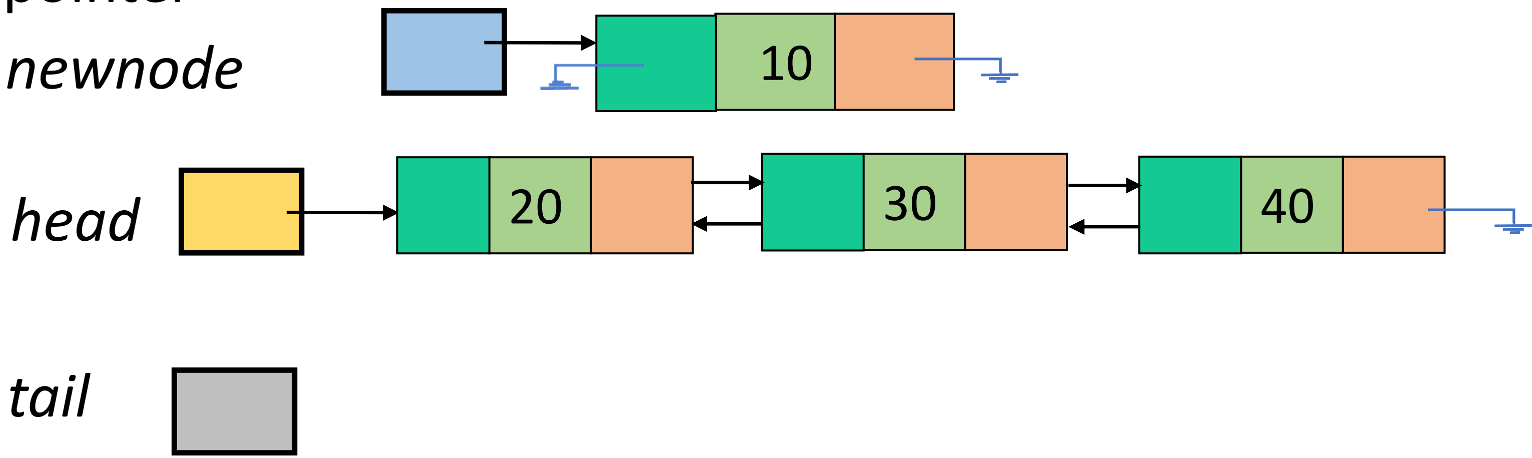
Steps:

(i) Creating a node with data

(ii) Adding a node to at the end of a linked list

a) Traversal the list till the tail pointer

```
struct node *tail

tail = head;
while ( tail → next != NULL)
        tail = tail → next
```



b) tail node pointer points to the new node

tail → next = newnode

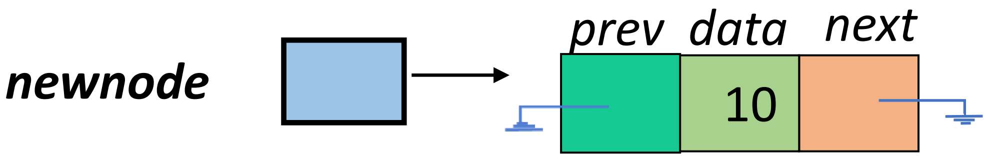
c) **new node *prev* pointer points to the tail node**

newnode → prev = tail

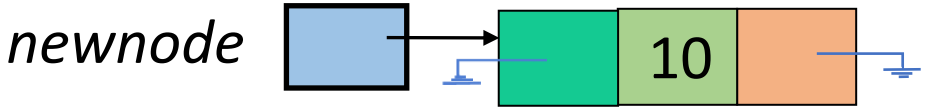
Doubly Linked List: Insert at the given position

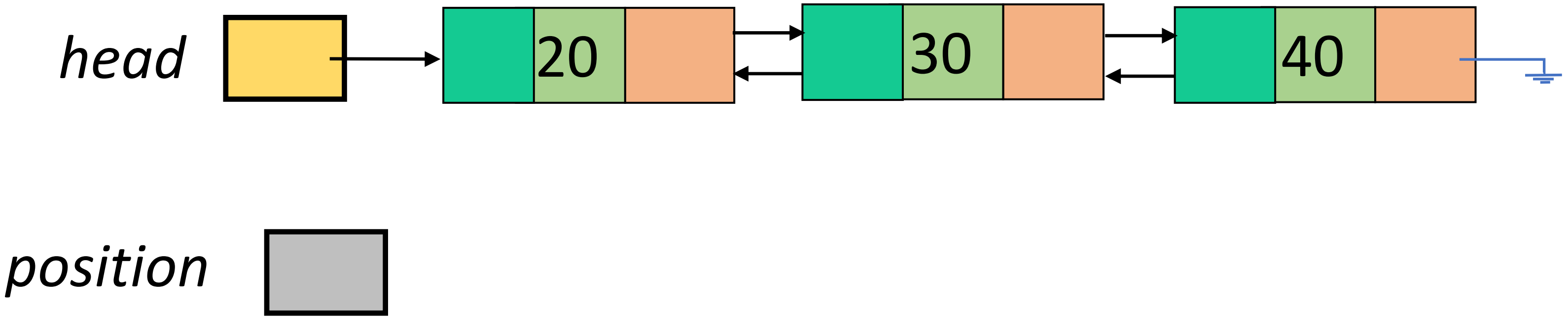
Steps:

(i) Creating a node with data



(ii) Adding a node to at the given position

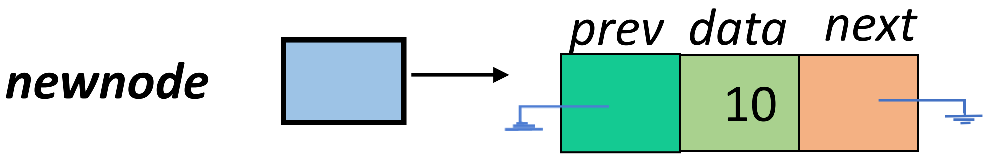




Doubly Linked List: Insert at the given position

Steps:

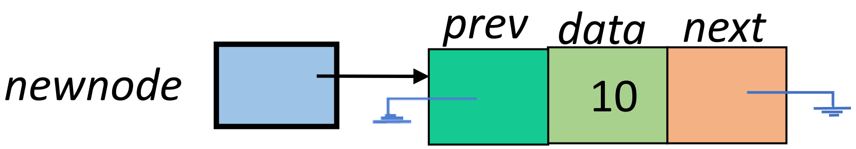
(i) Creating a node with data

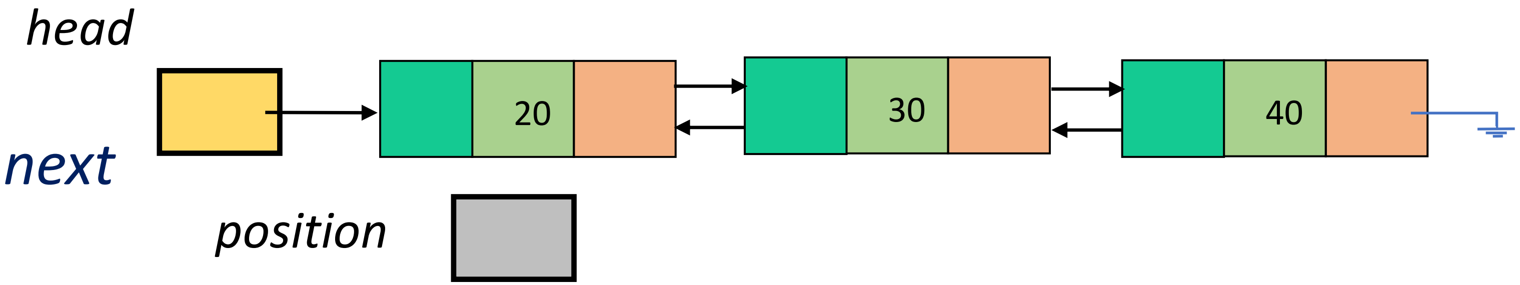


(ii) Adding a node to at the given position

a) Traversal the list till the *position – 1*

```
struct node *position
    position = head
    i = 0
    while (i<pos-1)
        position = position → next
        i++;
```





b) Point new node *prev* to the position node and *next* to the next node of the position node

1 *newnode → prev= position;*

2 *newnode → next = position → next*

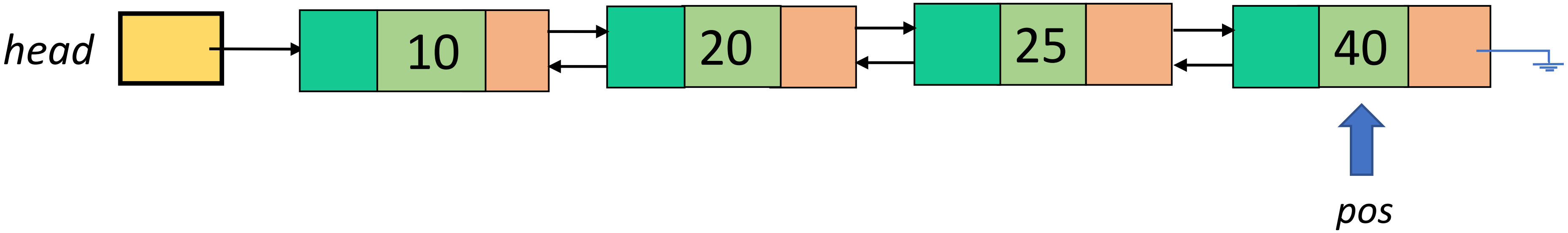
c) Point position next prev to the newnode and position next to the new node

3 *position→ next→ prev=newnode;*

4 *position → next = newnode*

Exercise: Doubly Linked List (1)

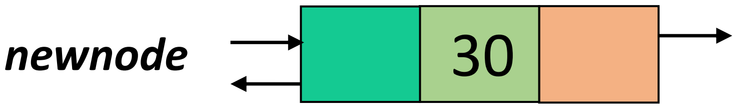
```
struct node {
        struct node *prev;
        int data;
        struct node *next;
}
```



Q: Insert the given newnode after the provided position?

Note: (i) multiple options are possible

(ii) the order of the given instructions are to be followed



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

(a) newnode → next = pos
pos → next = newnode
newnode → prev = NULL

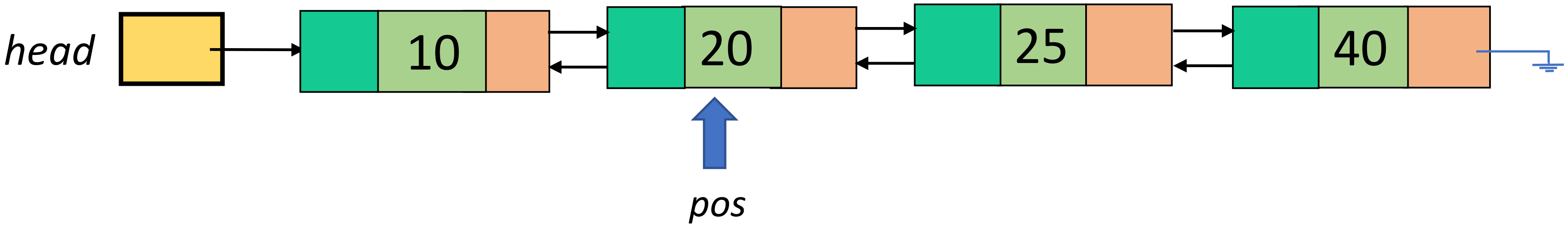
(b) pos → next = newnode
newnode → prev = pos
newnode → next = NULL

(c) newnode → prev = pos
pos → next = newnode
newnode → next = NULL

(d) newnode → prev = NULL
newnode → next = pos
pos → next = newnode

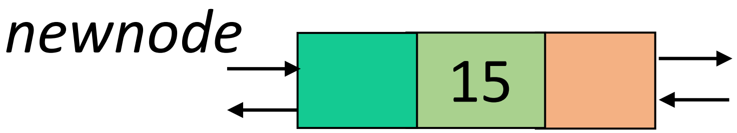
Exercise: Doubly Linked List (2)

```
struct node {
        struct node *prev;
        int data;
        struct node *next;
}
```



Q: Insert the given newnode <u>before the provided position</u>?

Note: the order of the given instructions are to be followed



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

(a) newnode → next = pos
newnode → prev = pos → prev
pos → prev → next = newnode
pos → prev = newnode

(b) pos → prev → next = newnode
pos → prev = newnode
newnode → prev = pos → prev
pos → prev → next = newnode

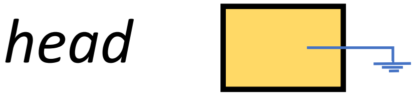
(c) pos → prev → next = newnode
newnode → prev = pos → prev
pos → prev = newnode
pos → prev → next = newnode

(d) newnode → prev = pos
newnode → next = pos → next
pos → next → prev = newnode
pos → next = newnode

Doubly Linked List: delete at the beginning or delete at the head

Steps:

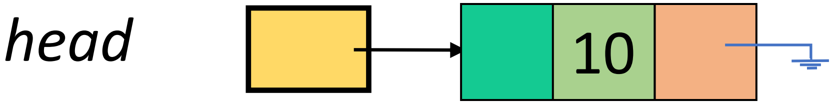
(i) Deleting a node from the empty list



```
If (head == NULL)
    printf("List is Empty\n")
```

(ii) Deleting a node at the beginning of linked list when only one node exisit

```
struct node *delbegin
```

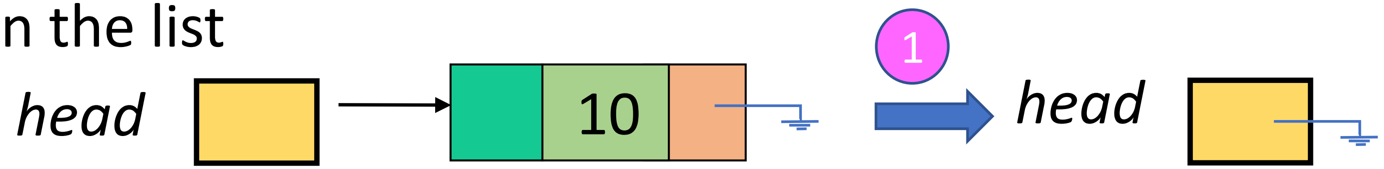


a) Point the head node to the delete pointer

```
delbegin = head
```

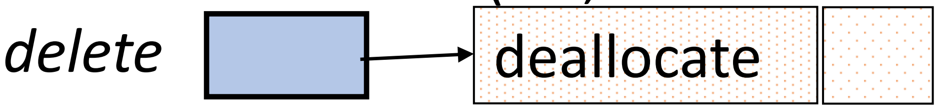
b) delete the node in case of single node in the list

```
If (head → next == NULL)
    head  = head → next
```



c) physically deleting the node from the node (i.e., return the allocated node memory to head)

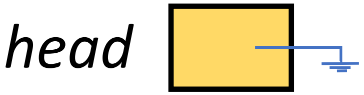
```
free (delbegin)
```



Doubly Linked List: delete at the beginning or delete at the head

Steps:

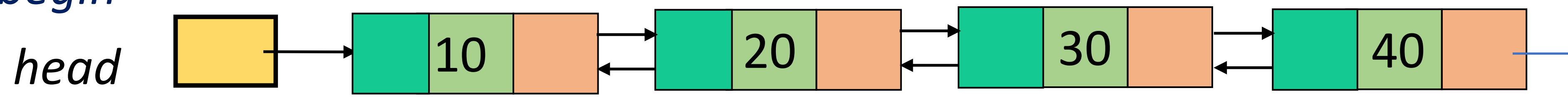
(i) Deleting a node from the empty list



(ii) Deleting a node at the beginning of linked list

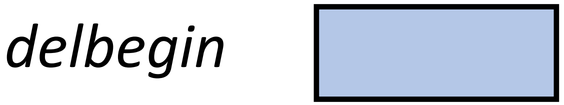
a) Point the head node to the delete pointer

*struct node *delbegin*



b) point head to next node of the head and *head prev* to NULL

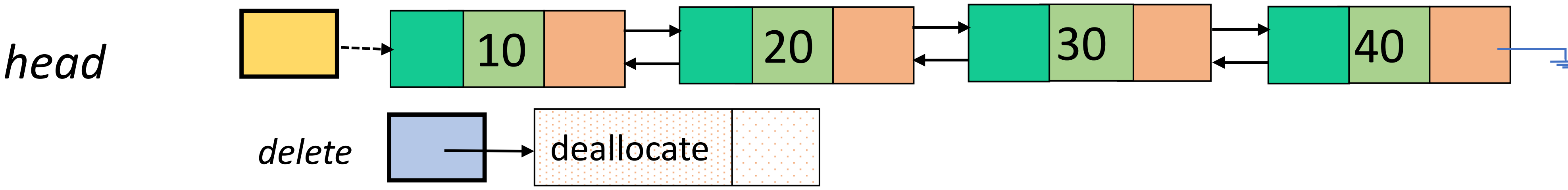
delbegin = head



1 *head = head → next*

2 *head→ prev = NULL*

c) physically deleting the node from the node (i.e., return the allocated node memory to head)



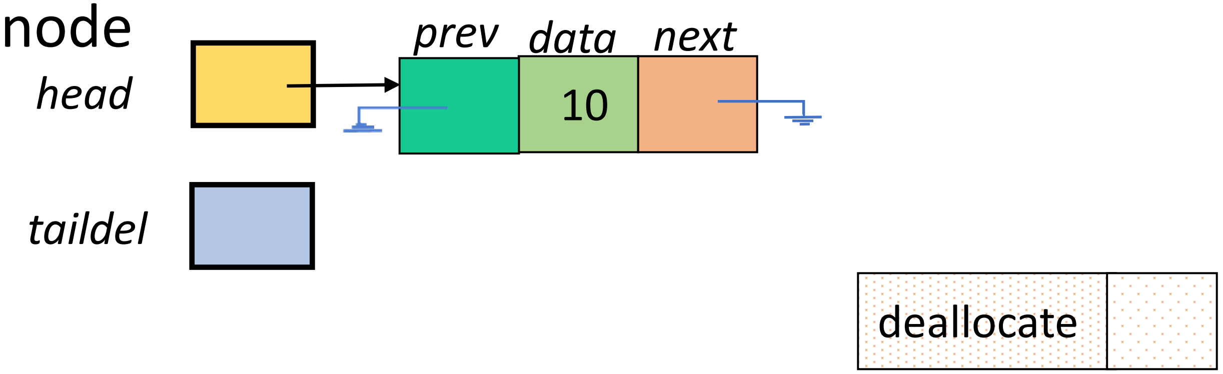
free (delete)

Doubly Linked List: delete at the end or delete at the tail

Steps:

(i) Deleting a node from the linked list with one node

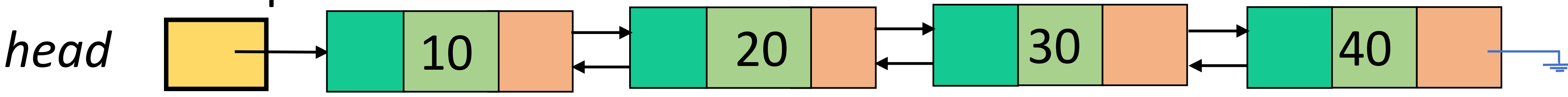
```
struct node *taildel
taildel = head
If (head → next == NULL)
     head = NULL
     free (taildel)
```



(ii) Deleting a node at the end of linked list

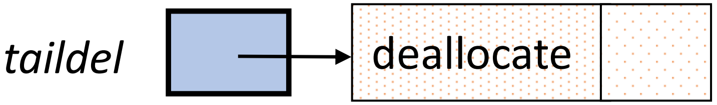
a) Point the head to the delete pointer

```
taildel = head
```



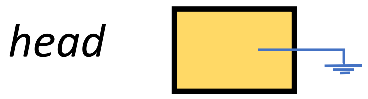
b) Traversal to the tail node

```
while ( tail → next != NULL)
          tail = tail → next
taildel→ prev → next = NULL
```

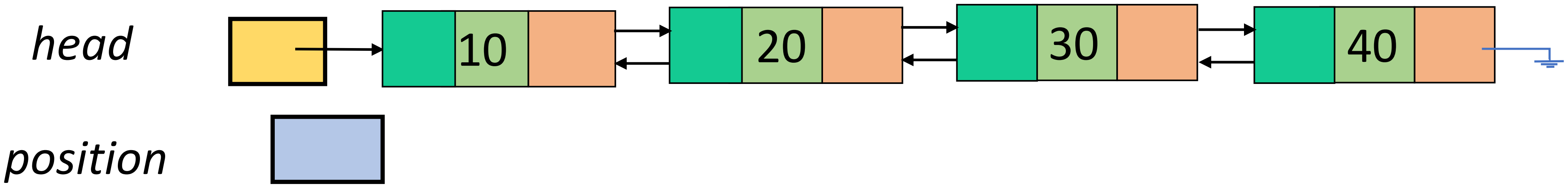


c) free tail node *free (taildel)*

Linked List: delete at the given position


head

Steps:


head

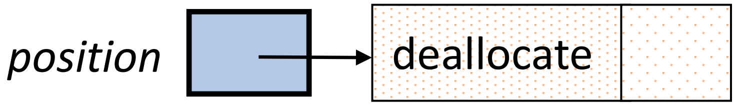
10

20

30

40

position


position

deallocate

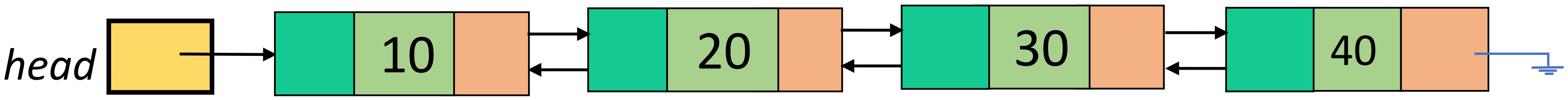
Linked List: delete at the given position

Steps:

(iii) Deleting a node at the given position

a) Traversal the list till the *position – 1*

```
struct node *position
position = head
i = 0
while (i<pos)
    positionprevnode = position
    position = position → next
    i++;
```



① *position → next → prev = position → prev;*

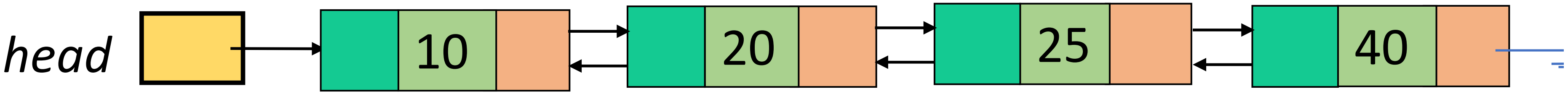
② *position → prev → next = position → next*

free (position)

Note: assume the position is not the first and last position

deallocate

Exercise: Doubly Linked List (3)



```
struct node {
        struct node *prev;
        int data;
        struct node *next;
}
```

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

Q: Map the node delete options?

Note: the order of the given instructions are to be followed

(a) pos → prev → next = NULL

(b) head = head → next
head → prev = NULL

(c) pos → prev → next = pos → next
pos → next → prev = pos → prev

(d) pos → next → next → pre = pos
pos → next = pos → next → next

(i) To delete the first node of the given list

(ii) To delete the last node of the given list

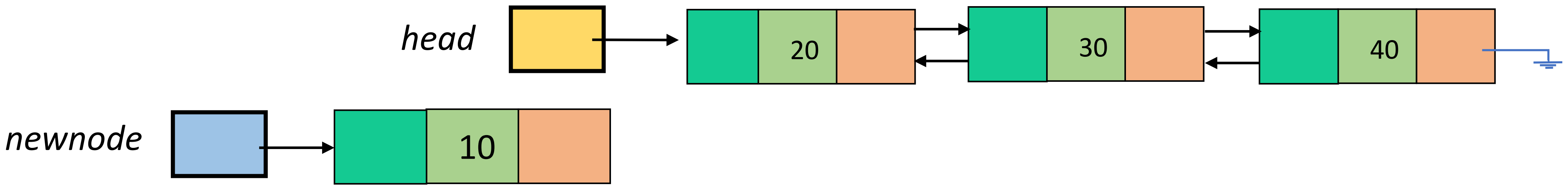
(iii) To delete the node after the given position (pos)

(iv) To delete the node from the given position (pos)

Exercise: Doubly Linked List (4)

Q: Fill the following table with the number of pointer operations need to be changed for each doubly linked list operation ?

Operations	begin	end	Middle (pos)
Insert			
delete			



(a) newnode → next = head
head → prev = newnode
newnode → prev = NULL
head = newnode

(b) newnode → prev = pos
pos → next = newnode
newnode → next = NULL

(c) newnode → next = pos
newnode → prev = pos → prev
pos → prev → next = newnode
pos → prev = newnode

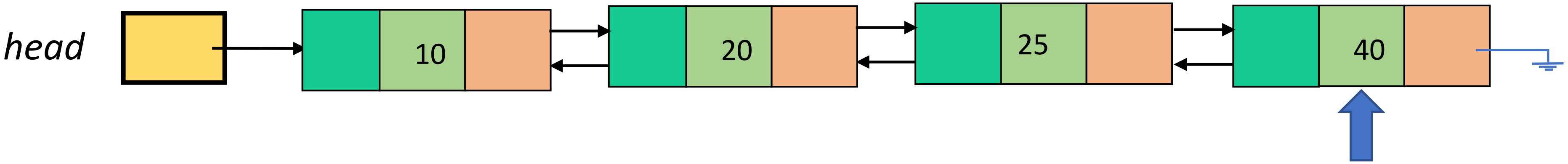
Limitations: Doubly Linked List

```
struct node {
        struct node *prev;
        int data;
        struct node *next;
}
```

- ***Limitations:***

a) Each node requires an extra pointer → more space

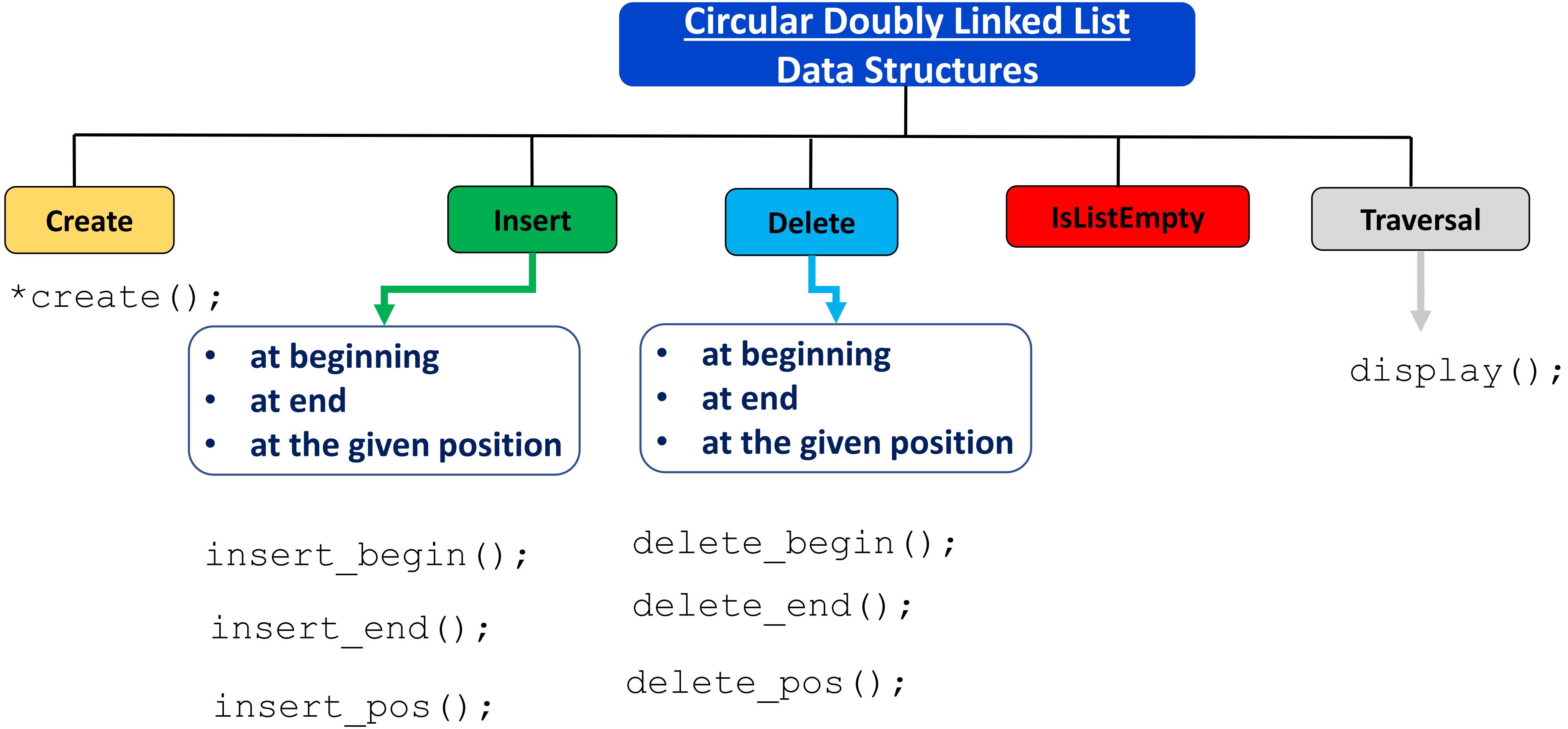
b) More operations required to add new node or delete node → requires an additional care to avoid loops



c) Traversal to the last node → find a last node in the given linked list

d) Requires an additional time for ***reverse traversing***

Circular Doubly Linked List: Operations



Circular Doubly Linked List
Data Structures

Create

`*create();`

Insert

- at beginning
- at end
- at the given position

```
insert_begin();

insert_end();

insert_pos();
```

Delete

- at beginning
- at end
- at the given position

```
delete_begin();

delete_end();

delete_pos();
```

IsListEmpty

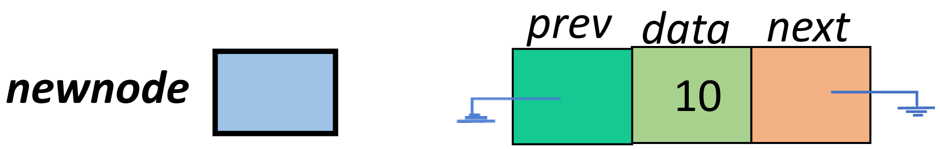
Traversal

`display();`

Circular Doubly Linked List: Insert at the beginning or Insert at the head

Steps:

(i) Creating a node with data



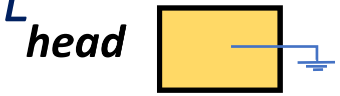
```
struct node {
        struct node *prev;
        int data;
        struct node *next;
}
```

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

newnode → prev = NULL;

newnode → data = 10;   //Entering data

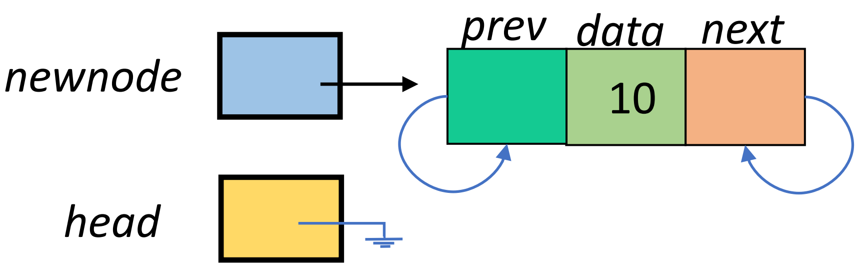
newnode → next = NULL; //making node next to NULL



*struct node *head = NULL*

(ii) Adding a node to an empty linked list

```
if (head == NULL)
    head = newnode
newnode → prev = head;
newnode → next =head;
```



Exercise: Operation → Adding a node to an empty list

Q: Map the following linked lists "to insert an element to empty list":

(i) newnode → prev = head;
newnode → next = head;

(ii) newnode → prev = NULL;
newnode → next =NULL;

(iii) newnode → next =NULL;

(iv) newnode → next =head;

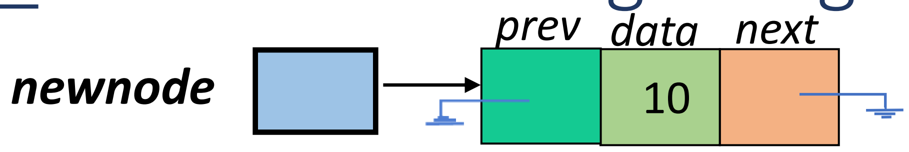
(a) Single Linked List

(b) Circular Single Linked List

(c) Doubly Linked List

(d) Circular Doubly Linked List

Circular Doubly Linked List: Insert at the beginning or Insert at the head



```
struct node {
        struct node *prev;
        int data;
        struct node *next;
}
```

Steps:

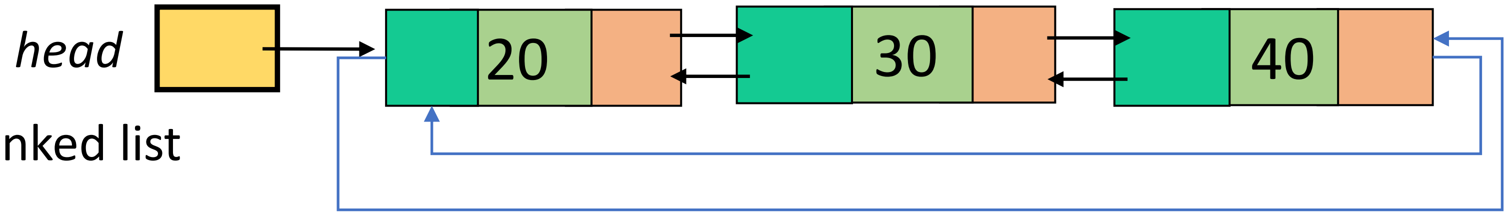
(i) Creating a node with data

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

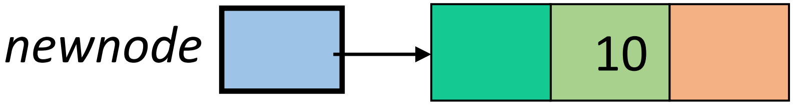
newnode → prev =NULL;

newnode → data = 10;

newnode → next = NULL;



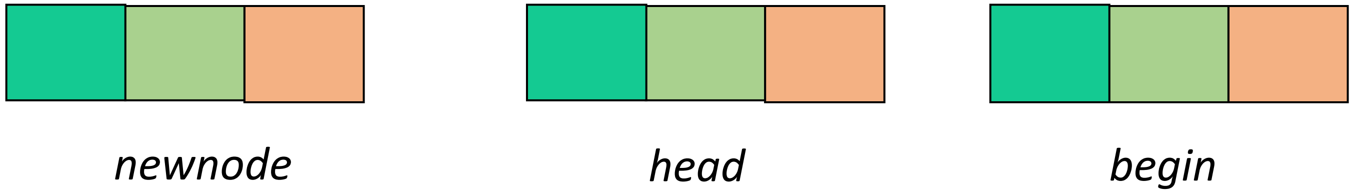
(ii) Adding a node to the beginning of a linked list



Circular Doubly Linked List: Insert at the beginning or Insert at the head

Steps:

(i) Traversal the list till the last node pointing to the head



*struct node *begin*

begin = head → prev

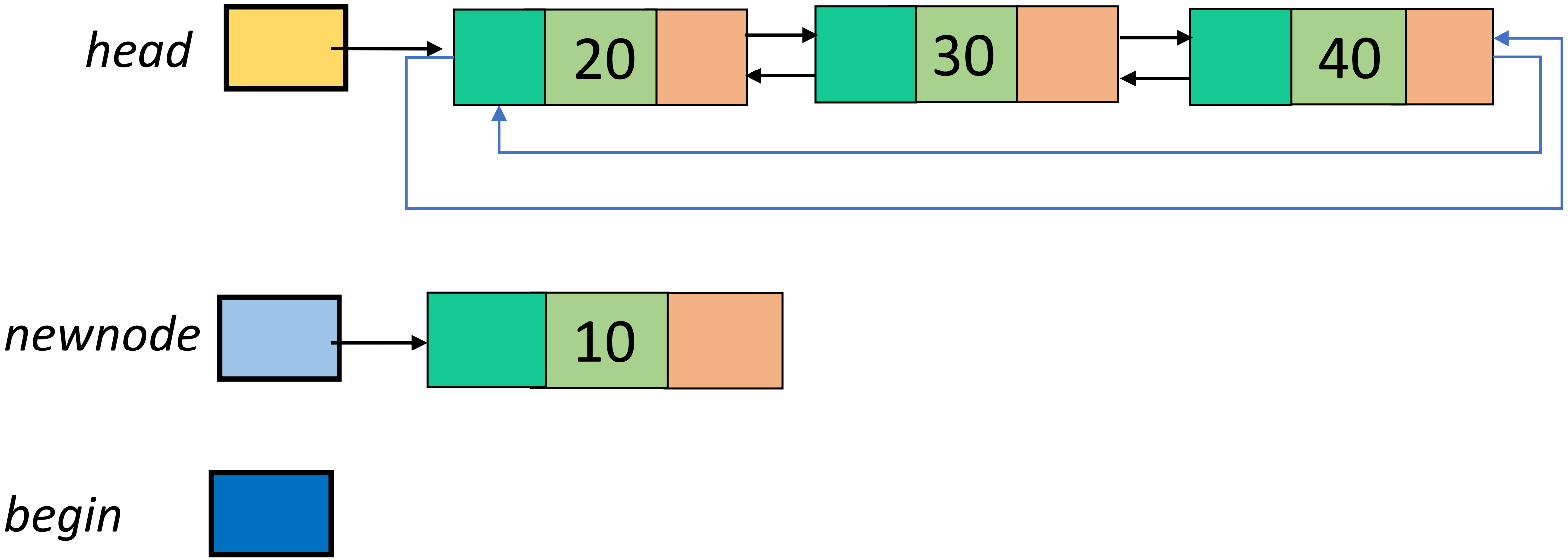
1 *begin->next = newnode;*

2 *newnode->prev=begin;*

3 *head->prev=newnode;*

4 *newnode->next=head;*

5 *head=newnode;*

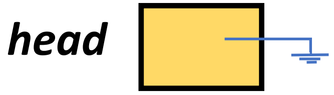


Circular Doubly Linked List: traversal or display

Steps:

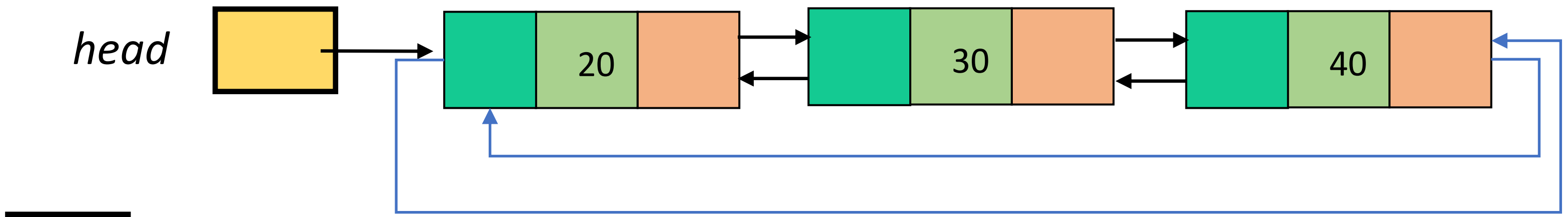
(i) Check if the linked list empty or not

```
if (head == NULL)
    printf ("Linked List is Empty\n");
```



*struct node *head = NULL*

(ii) List Traversal: Each node present in the list must be visited and display the data value



1 *struct node *traversal*

2 *traversal = head;*

```
3  while ( traversal → next != head)
           display the element: traversal → data
           traversal = traversal →next
4  display the last element: traversal → data
```

traversal != NULL ✗

Circular Doubly Linked List: Insert at the end or Insert at the tail

Steps:

(i) Traversal the list till the last node pointing to the head

*struct node *tail*

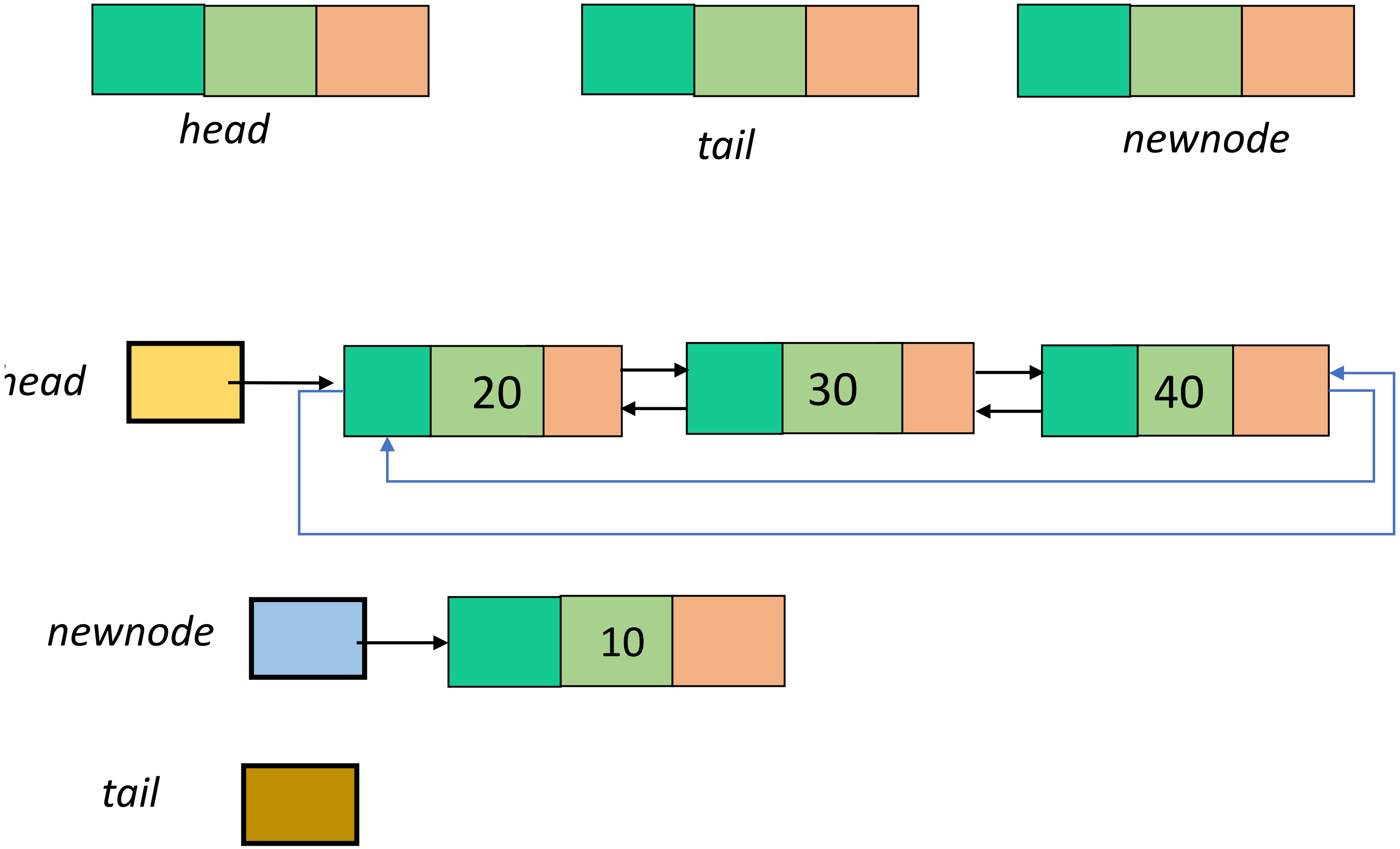
tail = head → prev

1 *tail->next = newnode;*

2 *newnode->prev= tail;*

3 *head->prev=newnode;*

4 *newnode->next=head;*



Circular Doubly Linked List: Insert at the given position

Steps:

(i) Traversal the list till the last node pointing to the head

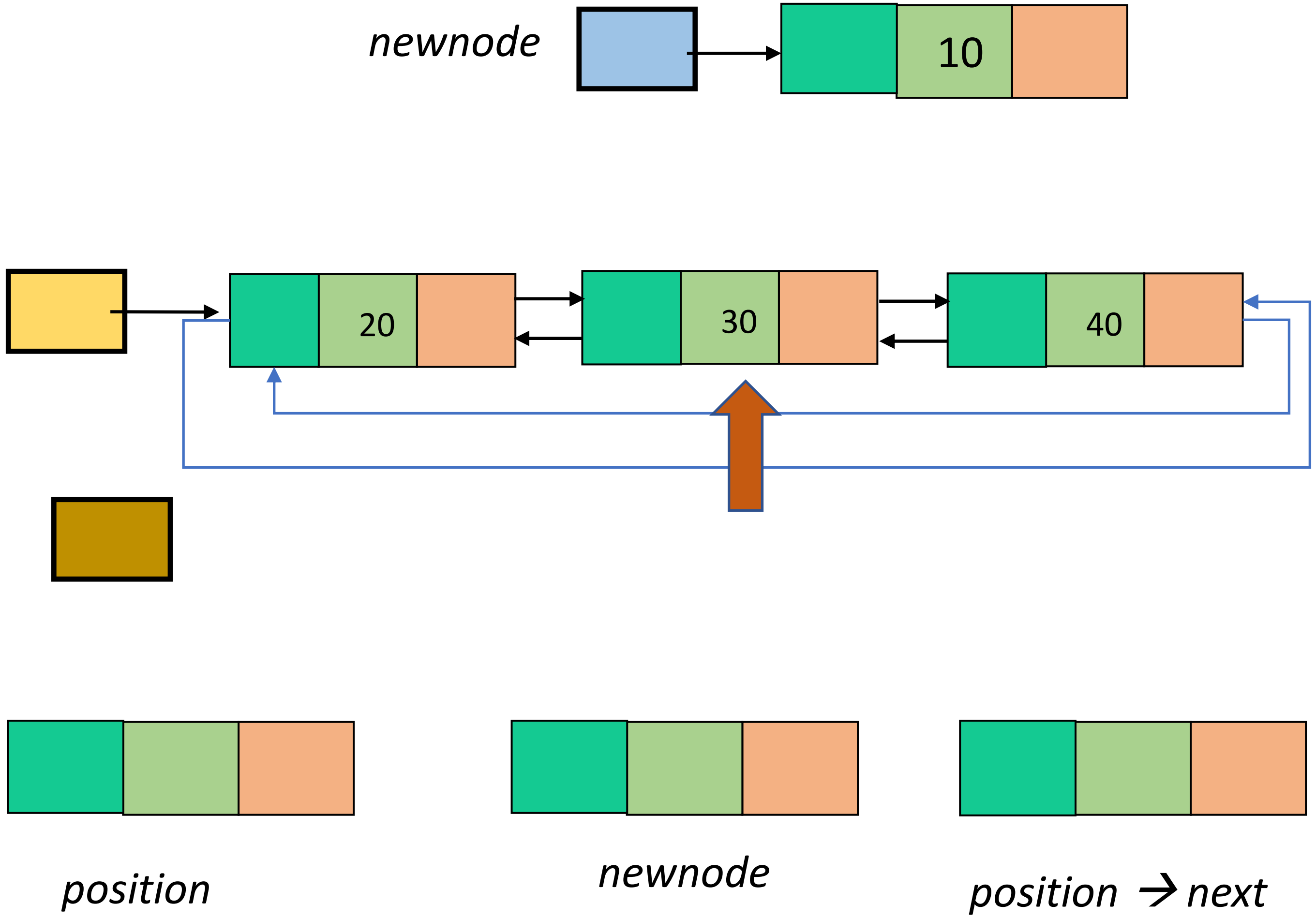
*struct node *position*

1 *newnode → prev = position;*

2 *newnode → next=position → next;*

3 *position → next → prev = newnode;*

4 *position → next= newnode;*



Circular Doubly Linked List: delete at the begin

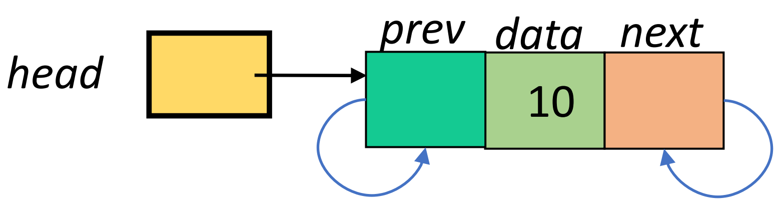
Steps:

(i) If list is empty

```
if (head == NULL)
printf (“List is empty\n”);
```

(ii) If the list contain only one node

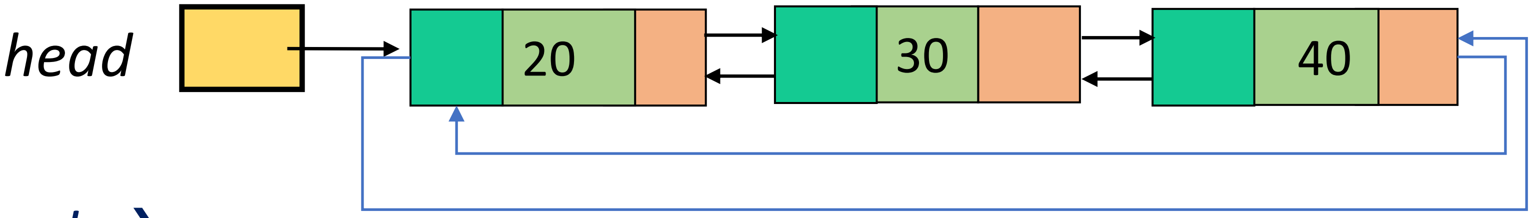
```
struct node *delbegin
if (head → next == head )
     delbegin=head;
     head == NULL;
     //print the deleted node data
    free (delbegin)
```



Circular Doubly Linked List: delete at the begin

Steps:

(iii) If the list contain more than one node



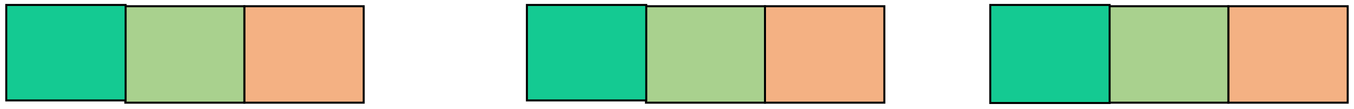
*struct node *delbegin*

1 *head → next → prev = head → prev;*

2 *head → prev → next= head → next;*

3 *head = head → next;*

4 *free (delbegin)*



head *head → next* *head → prev*

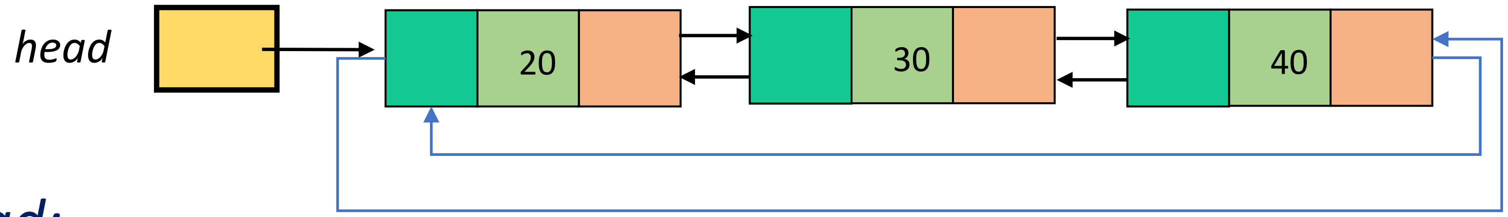
Circular Doubly Linked List: delete at the end

Steps:

(iv) If the list contain more than one node

*struct node *deltail*

deltail = head → prev



1 *deltail → prev → next = head;*

2 *head → prev = deltail → prev;*

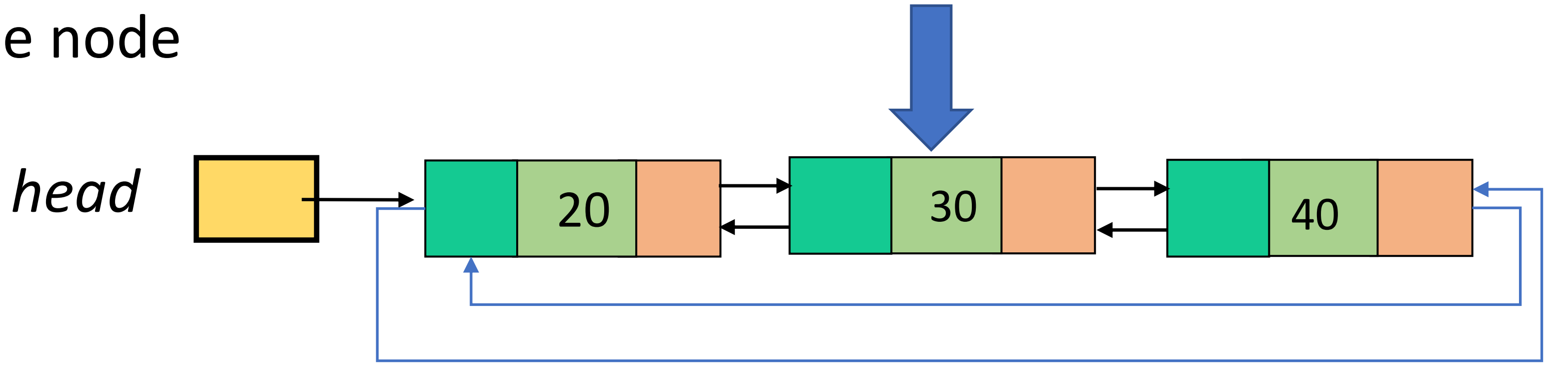
3 *free (deltail)*

Circular Doubly Linked List: delete at the position

Steps:

(v) If the list contain more than one node

*struct node *deltail*



① *position → prev → next = position → next;*

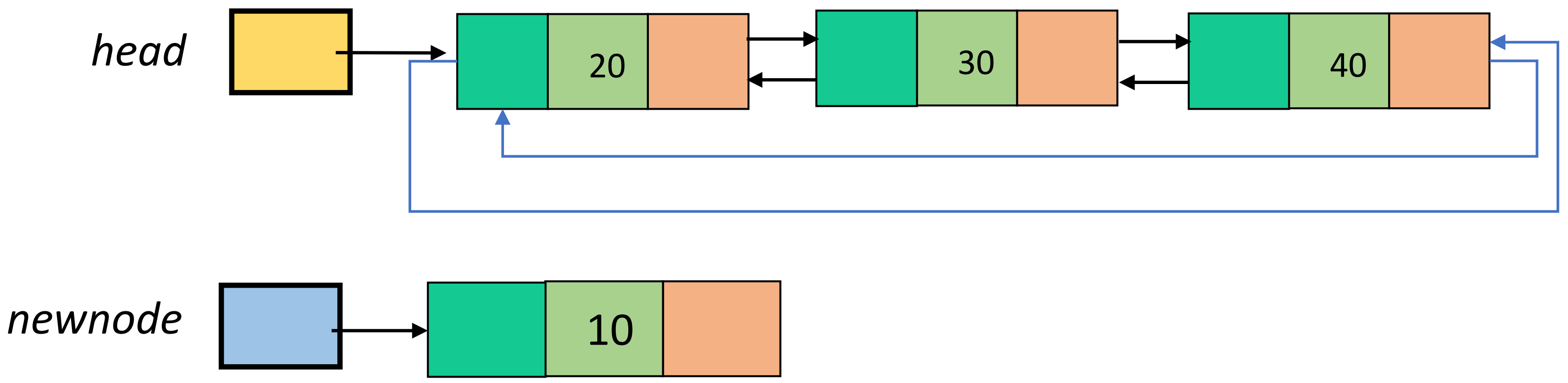
② *position → next → prev = position → prev;*

③ *free (deltail)*

Exercise: Circular Linked List (1)

Q: Fill the following table with the number of pointer operations need to be changed for each doubly linked list operation ?

Operations	begin	end	Middle (pos)
Insert			
delete			



thank you!

email:
k.kondepu@iitdh.ac.in

NEXT Class: 08/05/2023