CS2x1:Data Structures and Algorithms

Koteswararao Kondepu

k.kondepu@iitdh.ac.in

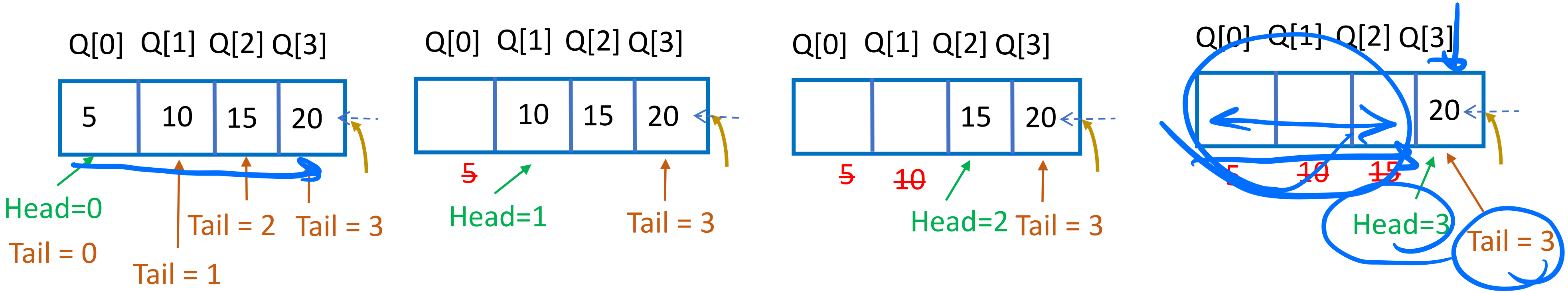
Outline

- Exercise on Circular Queue
- Limitations of Stack and Queue
- Linked list data structures
- Linked list operations
- Linked list exceptions
- Linked list implementation
- Linked list applications

Recap Simple Queue: Limitations (1)

```
Head = Tail = -1

 Enqueue(5)
 Enqueue(10)
 Enqueue(15)
 Enqueue(20)          Dequeue()          Dequeue()          Dequeue()
```



Ineffective: Suffers due to queue is full

✓ **Queue Migration Problem**

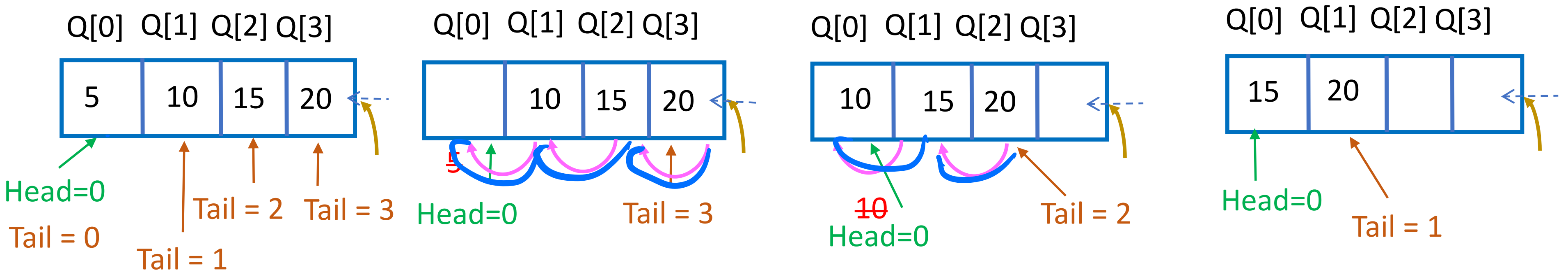
Real Queue: Limitations (2)

```
Head = Tail = -1

Enqueue(5)
Enqueue(10)
Enqueue(15)
Enqueue(20)
```

```
Dequeue()
```

```
Dequeue()
```



Q[0] Q[1] Q[2] Q[3]

5	10	15	20

Head=0

Tail = 0

Tail = 1

Tail = 2

Tail = 3

Q[0] Q[1] Q[2] Q[3]

	10	15	20

Head=0

Tail = 3

Q[0] Q[1] Q[2] Q[3]

10	15	20	

Head=0

Tail = 2

Q[0] Q[1] Q[2] Q[3]

15	20		

Head=0

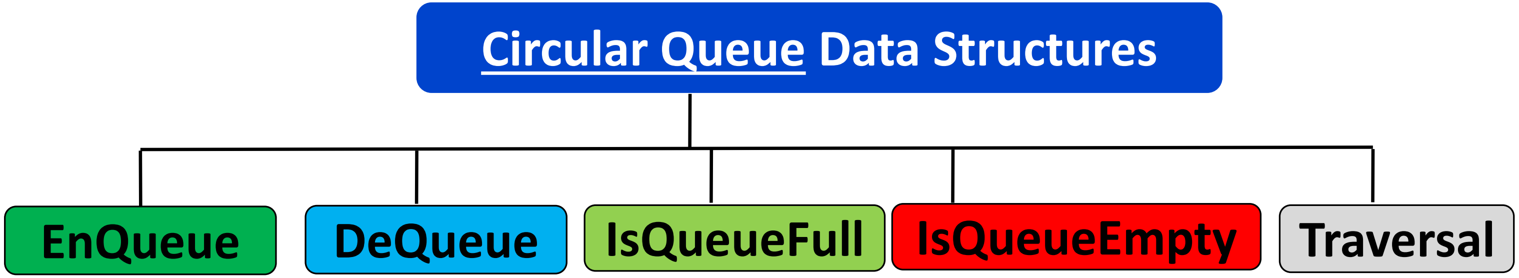
Tail = 1

Deletion Process Inefficient

No Queue Migration

Recap

- Circular Queue (FIFO) Implementation
  - EnQueue ()
  - DeQueue ()
  - IsQueueFull ()
  - IsQueueEmpty ()
  - PrintQueue ()



Circular Queue Data Structures

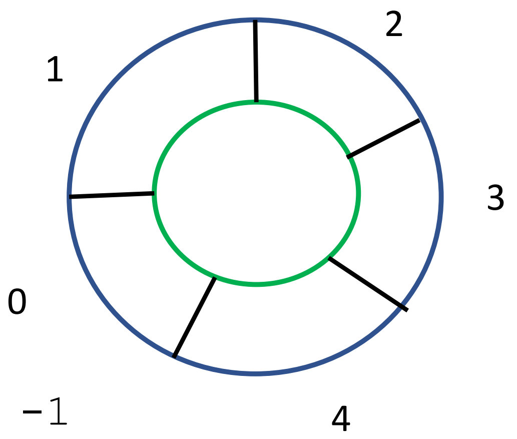
EnQueue

DeQueue

IsQueueFull

IsQueueEmpty

Traversal



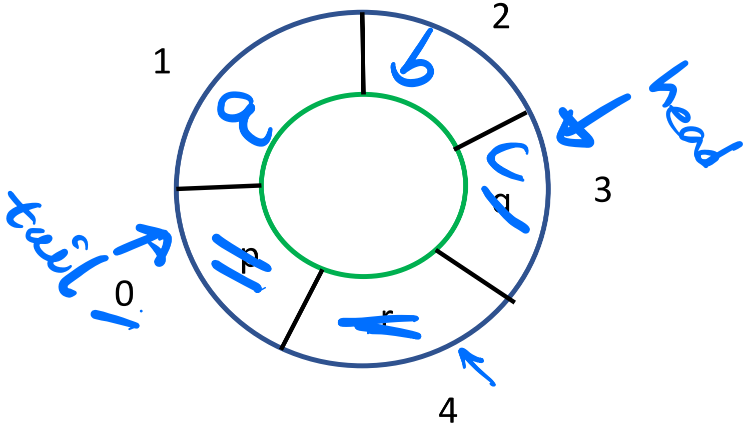
No Queue Migration Problem

Deletion Process Efficient

```
Head = Tail = -1
```

Exercise: Circular Queue (1)

The initial Circular Queue configurations are shown in Figure below.



What is the circular queue content after the following operations: EnQueue (a), DeQueue(), EnQueue (b), DeQueue (), EnQueue (c), DeQueue ()

(a)

Q[0]	Q[1]	Q[2]	Q[3]	Q[4]
	a	b		r

[✓ *(b)*]

Q[0]	Q[1]	Q[2]	Q[3]	Q[4]
–	a	b	c	–

(c)

Q[0]	Q[1]	Q[2]	Q[3]	Q[4]
	a	b		c

Exercise: Circular Queue (2)

Given a circular queue with size 7. What is the final value at index 3, after the following code is executed:

[Note: the circular queue array index starts at 0]

```
for (int k = 1; k <= 7; k++){
EnQueue(k);
}
for (int i = 1; i <= 4; i++){
  int delete;  //to store the dequeued/deleted element
  Dequeue();
  delete=DeQueue();
  EnQueue(delete);
}
```

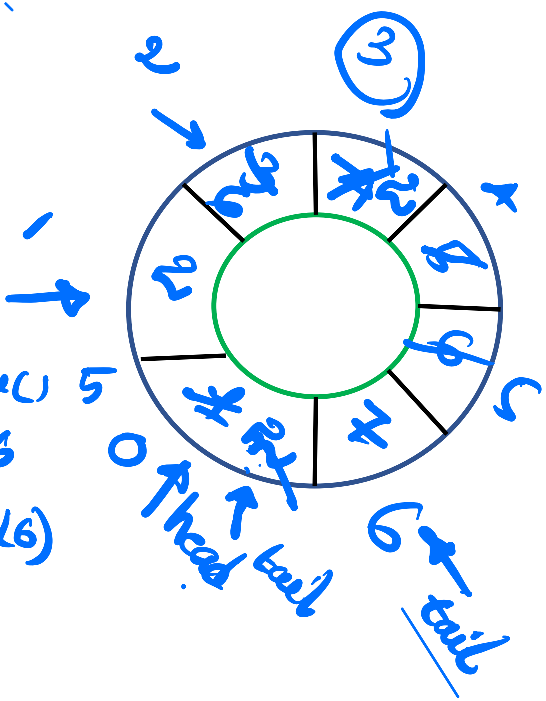
[✓ (a) 2]   (b) 4   (c) 6   (d) 7

i = 1
Dequeue() 1
delete = 2
Enqueue(2)

i = 2
Dequeue() 3
delete = 4
Enqueue(4)

i = 3
Dequeue() 5
delete = 6
Enqueue(6)

i = 4
Dequeue() 7
delete = 2
Enqueue(2)

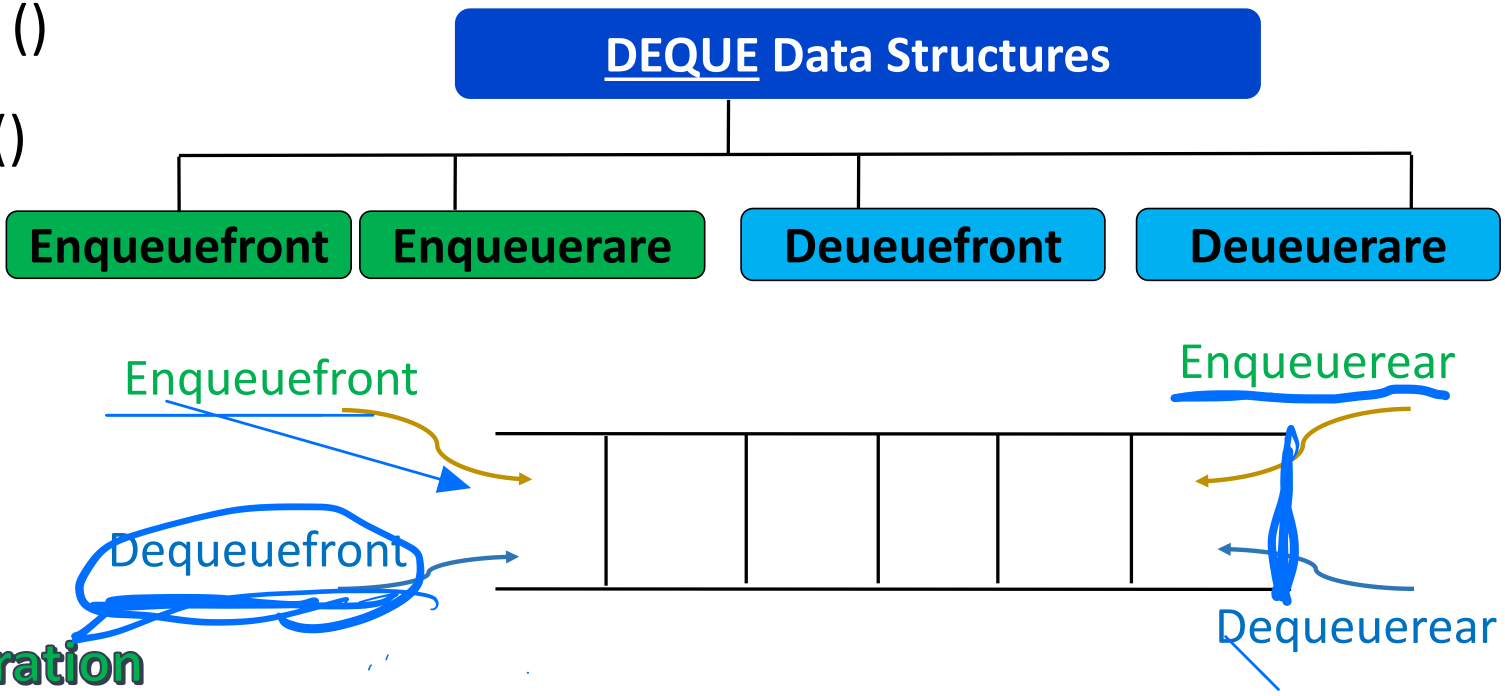


DEQUE Double Ended QUEues

Priority Queue

Enqueue(5, 2)

- Circular Queue (FIFO) Implementation
  - Enqueuefront ()
  - Enqueuerear ()
  - Dequefront ()
  - Dequerear ()
  - IsEmpty ()



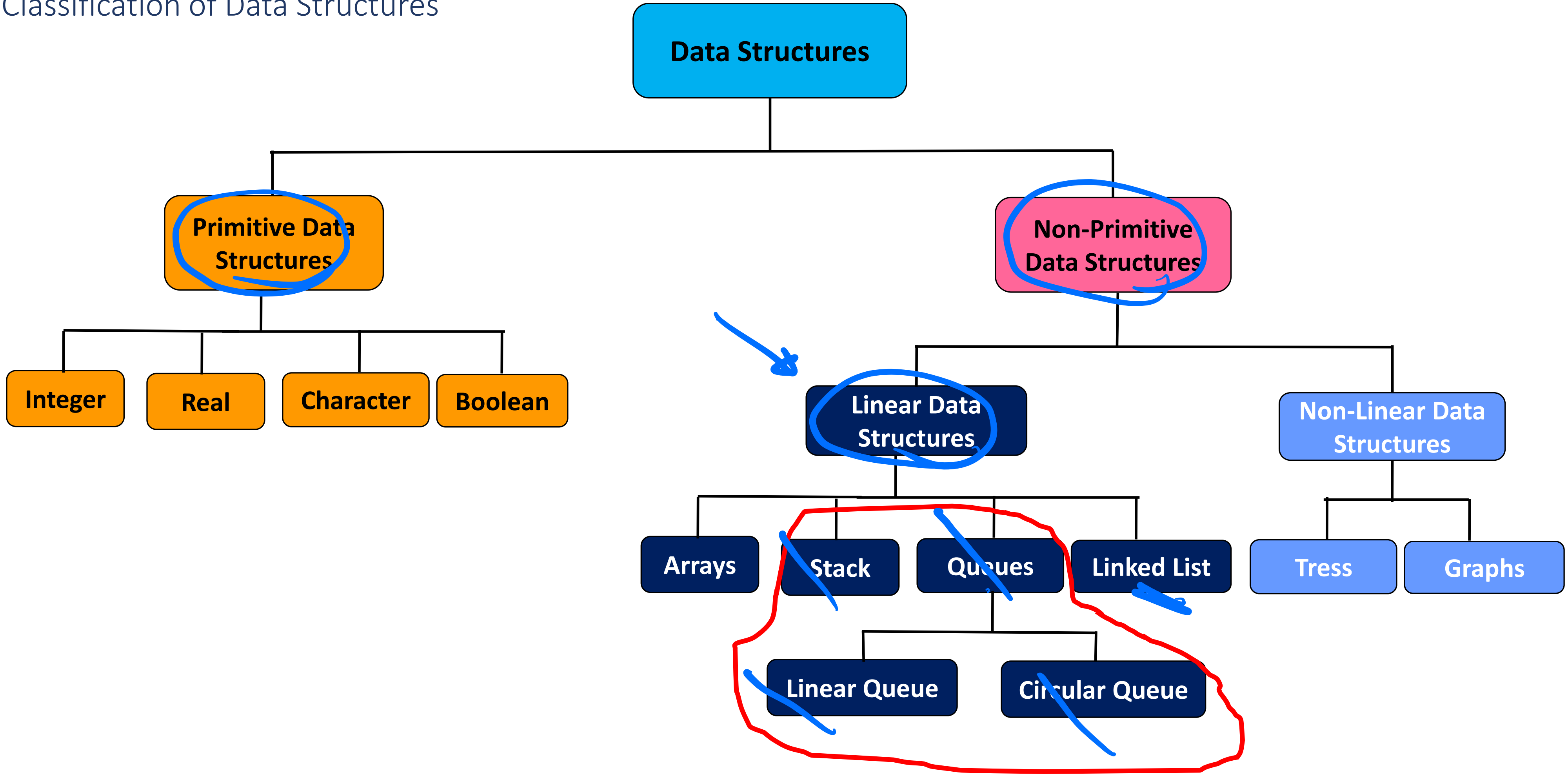
No Queue Migration

Deletion Process efficient

Elements can be added/removed at both ends

Recap

Classification of Data Structures



Linked List

Advantages:

(i) Dynamic in Size: Increase ↑ or Decrease ↓ → maximum size need not to be known in advance

(ii) No need of swapping if an element is Inserted or Deleted

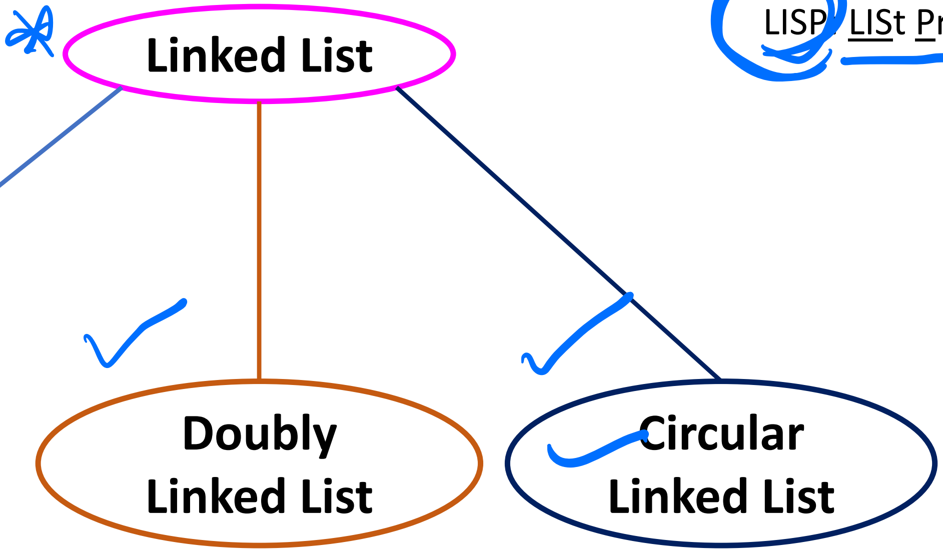
(iii) Memory assignment during the execution/run time

(iv) Important *******: Linked Lists are *linear* for accessing, and *non-linear for strong in memory*

(v) IS it Flexible in rearranging elements?

LISP: LISt Processor

1958

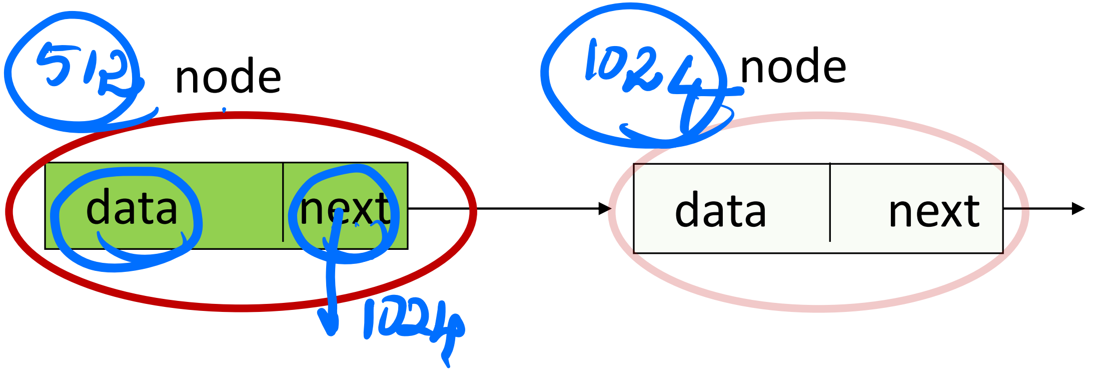


Linked List: Notations

***Assumptions**: Everyone familiar with how to define structure with pointers*

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

10



- **Node is a self-referential structure**
- **Nodes are logically adjacent, physically scattered**
- **Array is a physically adjacent and is contiguous.**

Revisit: structures with pointers

What is the output of the below program?

```
#include <stdio.h>
int main()
{
int a[5] = {1,2,3,4,5};
int *ptr;
ptr = (int *)(&a+1);
printf("%d %d\n", *(a+1), *(ptr-1));
}
```

&a+1

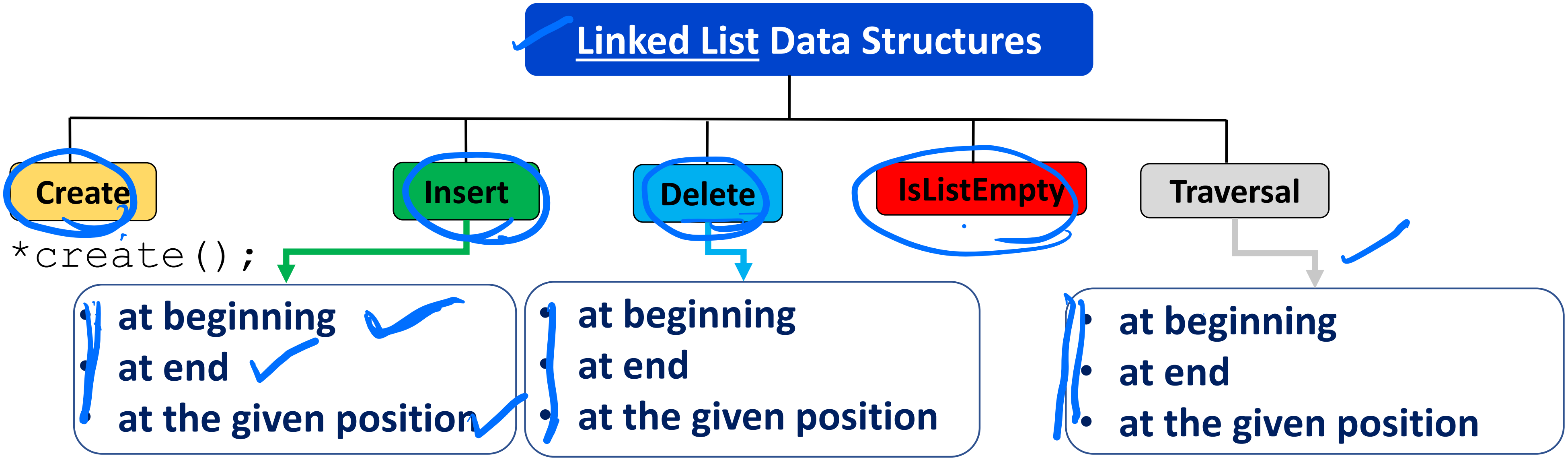
&(a+1)

5x4 = 20

2

5

Linked List: Operations



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

- at beginning
- at end
- at the given position

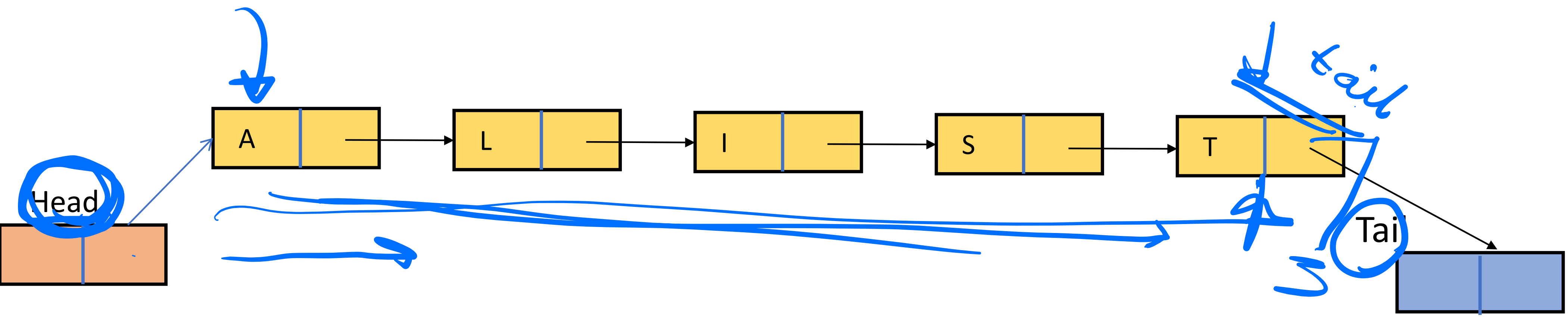
```
insert_begin();
insert_end();
insert_pos();
```

```
delete_begin();
delete_end();
delete_pos();
```

```
display();
get_begin();
get_end();
get_pos();
```

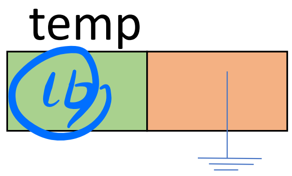
Linked List: Best practice

head = head→next



Linked List: Creation

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



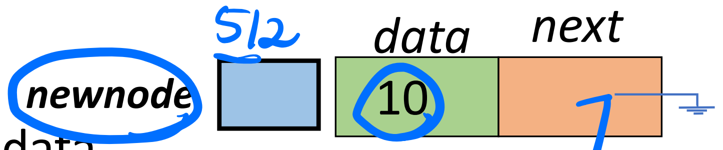
tem

78

```
void create(){
      struct node *temp;
      temp=(struct node *)malloc(sizeof(struct node));
      if(temp==NULL){
            printf("\n Out of Memory Space: \n");
            exit(0);
      }
      printf("\n Enter the data value for the node:");
      scanf("%d",&temp->info);
      temp->next=NULL;
}
```

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

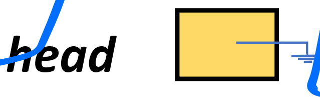
***Steps*:**



(i) Creating a node with data

```
struct node *newnode = malloc(sizeof(struct node));
newnode → data = 10;
newnode → next = NULL;
```

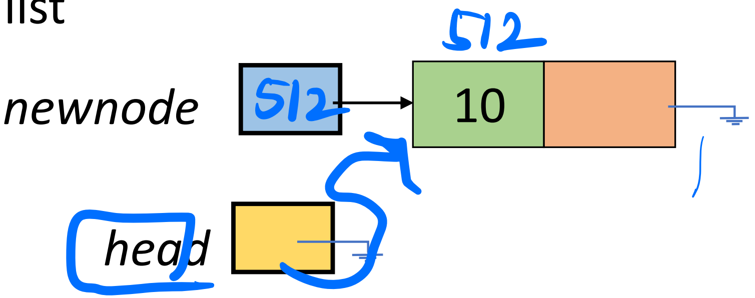
```
struct node {
            int data;
            struct node
*next;
}
```



*struct node *head = NULL*

(ii) Adding a node to an empty linked list

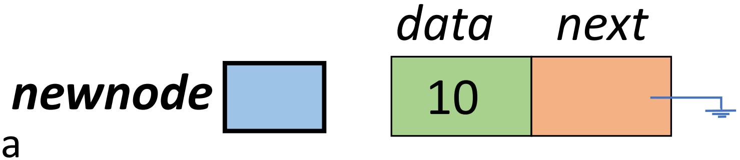
```
if (head == NULL)
    head = newnode
```



head = newnod

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

Steps:



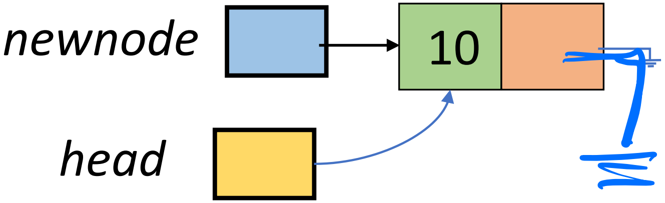
```
struct node {
        int data;
        struct node *next;
}
```

(i) Creating a node with data

```
struct node *newnode = malloc(sizeof(struct node));
newnode → data = 10;
newnode → next = NULL;
```

✓ (ii) Adding a node to an empty linked list

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



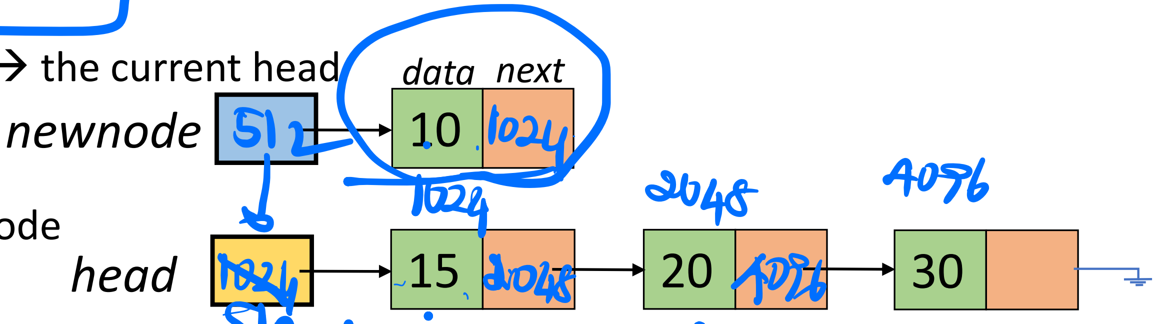
(iii) Adding a node to the beginning of a linked list

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

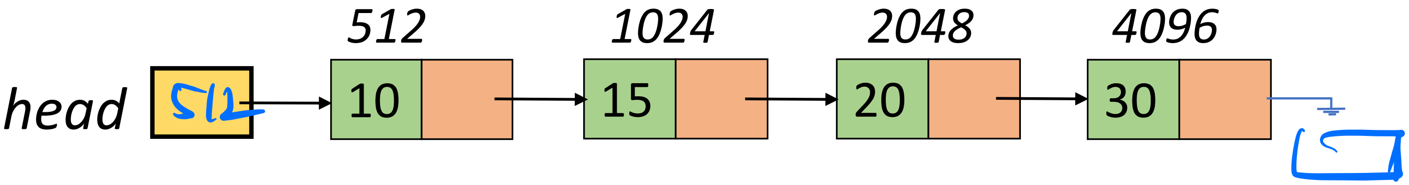
```
newnode → next = head
```

b) Update the head pointer to the new node

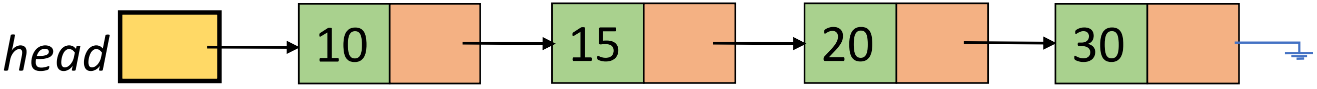
```
head = newnode
```



Linked List: Example



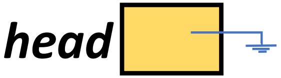
- *head → data* = ? 10
- *head → next* = ? 1024
- *head → next → next → data* = ? 20
- *?* 2048 *→ next → data* = ? 30
- *head → next → next → next → data* = ? 30
- *head → next → next → next → next → data* = ? ⟷ undefined / error



Linked List: traversal or display

***Steps*:**

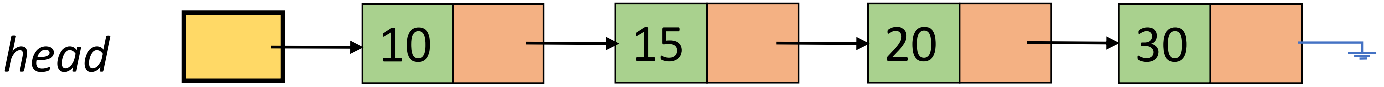
(i) Check if the linked list empty or not



*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

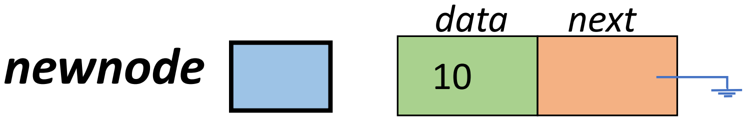


```
1  struct node *traversal
```

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

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

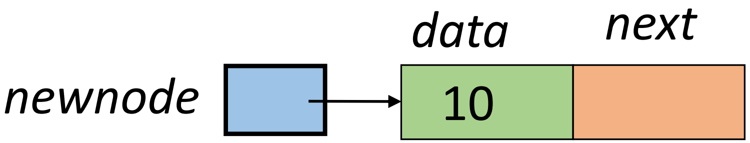
***Steps*:**



(i) Creating a node with data

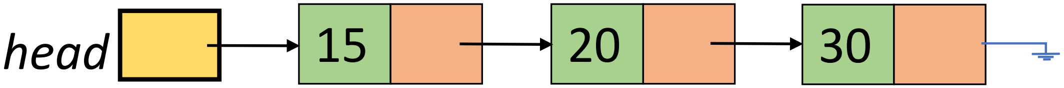
```
struct node *newnode = malloc(sizeof(struct node));
newnode → data = 10;
newnode → next = NULL;
```

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



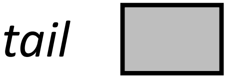
(ii) Adding a node to at the end of a linked list

```
tail = head;
```



a) Traversal the list till the tail pointer

```
struct node *tail
```



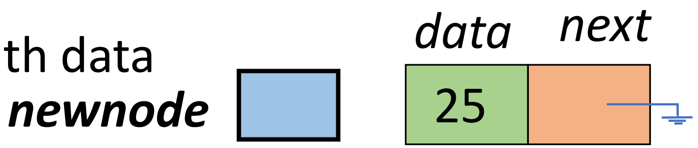
```
while ( tail → next != NULL)
        tail = tail → next
```

b) tail node pointer points to the new node *tail → next = newnode*

Linked List: Insert at the given position

***Steps*:**

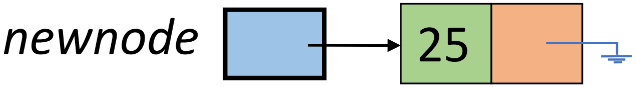
(i) Creating a node with data



```
struct node *newnode = malloc(sizeof(struct node));
newnode → data = 10;
newnode → next = NULL;
```

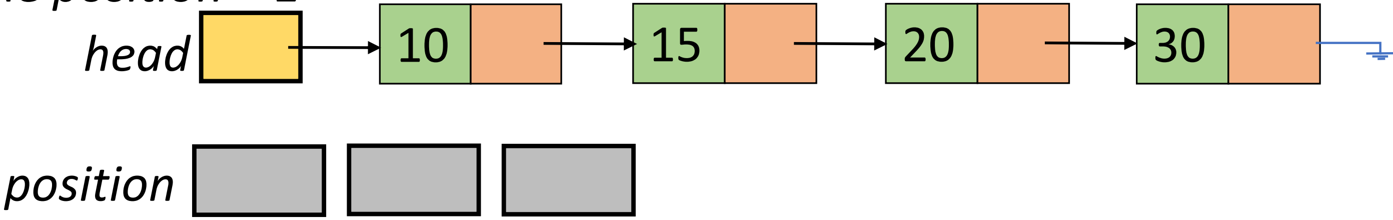
```
struct node {
        int data;
        struct node *next;
}
```

(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)
   position = position → next
   i++;
```



b) Point *newnode* → next to the position-node → next

```
newnode → next = position → next
```

c) Point position-node → next to the *newnode*

```
position → next = newnode
```

thank you!

email:
k.kondepu@iitdh.ac.in

NEXT Class: 28/04/2023