NURTURE
Education Solutions
TOMORROW'S HERE

Subject: Data Structures

Module Number: 02

Module Name: Queues and Linked Lists

Syllabus

Queues: Basic Queue Operations, Representation of a Queue using array, Implementation of Queue Operations using Stack, Applications of Queues- Round Robin Algorithm. Circular Queues, DeQueue and Priority Queues.

Linked Lists: Introduction, single linked list, representation of a linked list in memory, Different Operations on a Single linked list, Reversing a single linked list, Advantages and disadvantages of single linked list, circular linked list, double linked list and Header linked list.

Aim:

The aim of this is to understand the operations and implementations of Queues and Linked Lists.

Objectives:

The objectives of this module are as follows:

- Analyze queue operations to understand their functionality.
- Design an efficient array-based queue.
- Implement queue operations using a stack, demonstrating versatility in data structures.
- Apply queue-based algorithms, like Round Robin scheduling, in real-world situations.
- Explore different queue variations.

Outcomes:

At the end of this module, you are expected to:

- Analyze fundamental queue operations to grasp their functionality.
- Implement a memory-efficient queue using an array.
- Perform various operations on a single linked list, including insertion and deletion.
- Apply algorithms to reverse a single linked list, showcasing problem-solving skills.
- Examine circular linked lists and doubly linked lists for diversified applications.

Table of Contents

Table of Contents (Contd….)

Queues: Basic Queue Operations

In simple language a queue is a simple waiting line which keeps growing if we add the elements to its end and keep shrinking on removal of elements from its front. A queue is a data structure where elements are added at the back and remove elements from the front.

Queues: Basic Queue Operations (Contd….)

In that way a queue is like "waiting in line": For example, while we read a book from a file, it is quite natural to store the read words in a queue so that once reading is complete the words are in the order as they appear in the book.

Queues: Basic Queue Operations (Contd….)

- It is a linear structure which follows a particular order in which the operations are performed. The order is First In First Out (FIFO).
- A good example of queue is any queue of consumers for a resource where the consumer that came first is served first.
- The difference between stacks and queues is in removing. In a stack we remove the item the most recently added; in a queue, we remove the item the least recently added.

Queues: Basic Queue Operations (Contd….)

Operations of Queues

- Enqueue: Adding an element to the rear (end) of the queue.
- Dequeue: Removing the element from the front (front) of the queue.
- Front: Getting the front element of the queue without removing it.
- Rear: Getting the rear element of the queue without removing it.
- IsEmpty: Checking if the queue is empty.
- IsFull: Checking if the queue is full (applicable to a fixed-size array-based implementation).

Queues: Basic Queue Operations (Contd….)

- Queue *Q;
- enqueue(Q, “a”);
- enqueue(Q, “b”);
- enqueue(Q, “c”);
- d=getFront(Q);
- dequeue(Q);
- enqueue(Q, “e”);
- dequeue(Q);

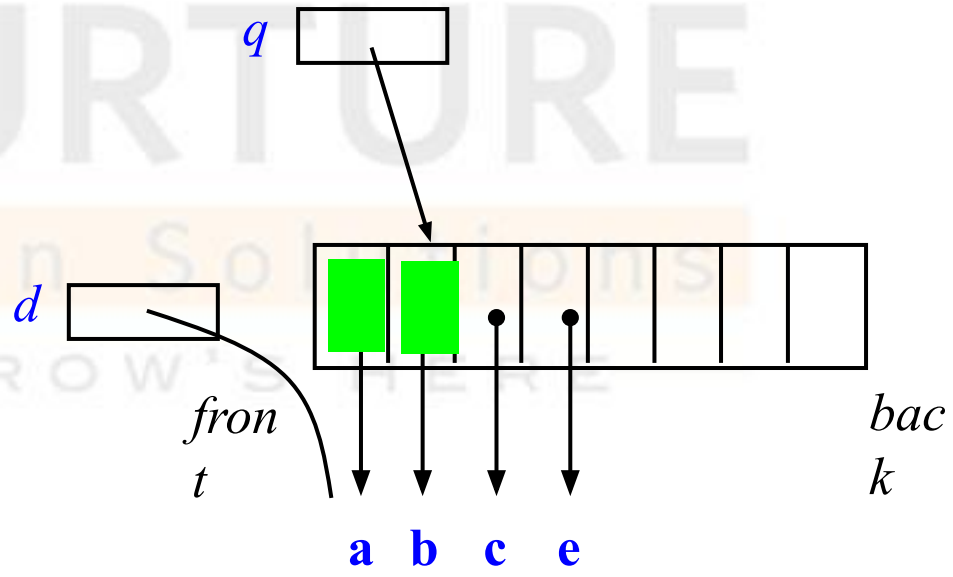


Figure : Operations on Queue

Queues: Basic Queue Operations (Contd….)

Queue ADT:

Queues implement the FIFO (first-in first-out) policy

An example is the printer/job queue!

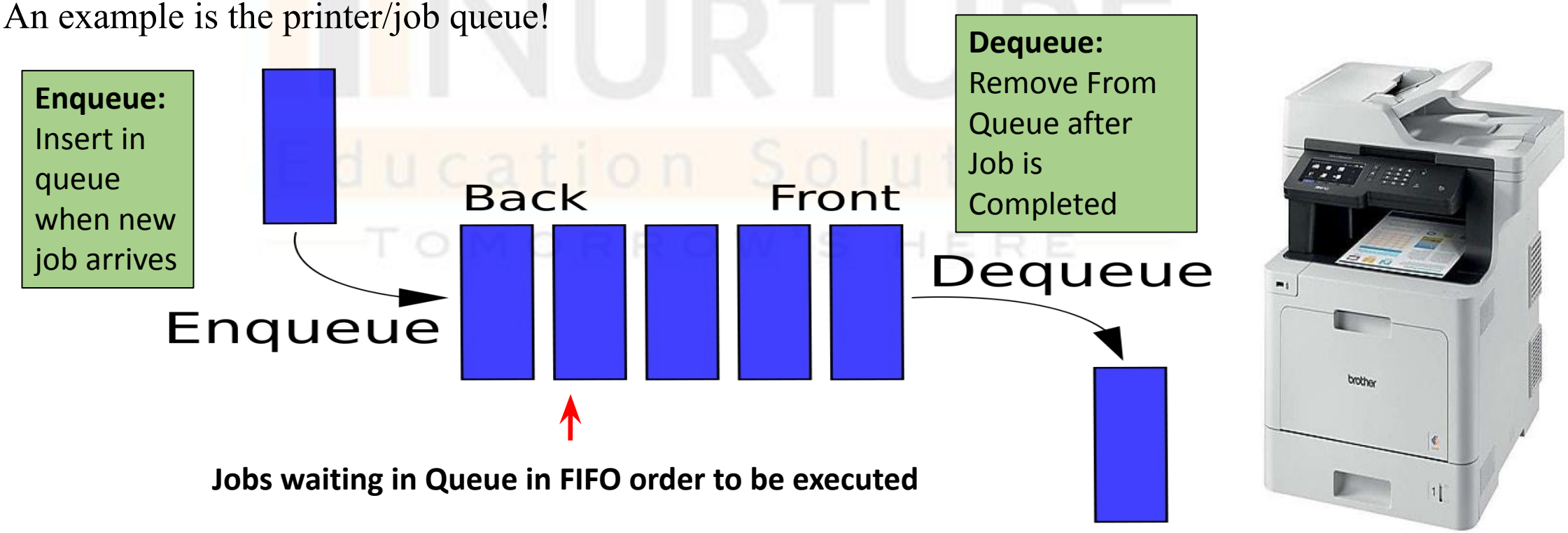


Figure : Printer Job Queue Example

Queues: Basic Queue Operations (Contd….)

The main functions in the Queue ADT are (Q is the queue)

```
void enqueue(o, Q)        // insert o to back of Q
void dequeue(Q);          // remove oldest item
Item getFront(Q);         // retrieve oldest item
boolean isEmpty(Q);       // checks if Q is empty
boolean isFull(Q);        // checks if Q is full
void clear(Q);                  // make Q empty
}
```

Representation of a Queue using array

A queue can be implemented using an array with two indices, front and rear, pointing to the front and rear of the queue, respectively. Elements are enqueued at the rear and dequeued from the front.

Use Array with front and back pointers as implementation of queue

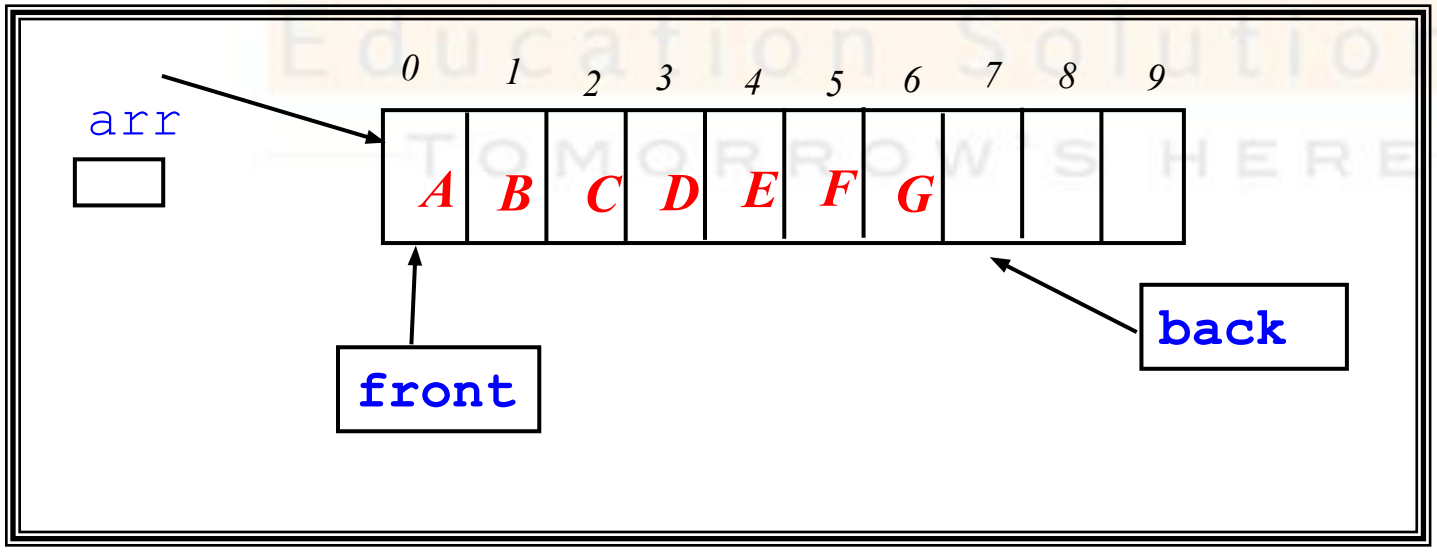


Figure : Array Representation of Queue

Representation of a Queue using array (Contd….)

Here's a basic structure for the queue using an array:

```
#define MAX_SIZE 100
typedef struct {
    int data[MAX_SIZE];
    int front;
    int rear;
} Queue;
```

Implementation of Queue Operations using Stack

Two stacks are used to simulate queue processes. Enqueue operations are performed using one stack (enqueueStack), and dequeue actions are performed using a different stack (dequeueStack). The FIFO behaviour is accomplished by moving the pieces between the two stacks.

Here's a basic structure for the queue using two stacks:

```
#include <stdio.h>
#include <stdlib.h>
#define MAX_SIZE 100
typedef struct {
int data[MAX_SIZE];
int top1; // Top index of the enqueue stack
int top2; // Top index of the dequeue stack} Queue;
```

Implementation of Queue Operations using Stack (Contd….)

```
void initialize(Queue* queue)
{  queue->top1 = -1;
   queue->top2 = -1;}
int isEmpty(Queue* queue)
{ return queue->top1 == -1 && queue->top2 == -1;}
int isFull(Queue* queue)
 {return queue->top1 == MAX_SIZE - 1 || queue->top2 == MAX_SIZE - 1;}
void enqueue(Queue* queue, int value)
 {  if (isFull(queue))
    {  printf("Queue overflow! Cannot enqueue element.\n");
       return;  }
```

Implementation of Queue Operations using Stack (Contd….)

```
if (queue->top1 == -1)
 { queue->top1 = 0; }
   queue->data[++queue->top1] = value;}
int dequeue(Queue* queue)
 { if (isEmpty(queue))
      {  printf("Queue underflow! Cannot dequeue element.\n");
      return -1;}
   if (queue->top2 == -1)
  {   while (queue->top1 != -1)
      { queue->data[++queue->top2] = queue->data[queue->top1--]; }
   }    return queue->data[queue->top2--]; }
```

Applications of Queues- Round Robin Algorithm,Circular Queues, DeQueue and Priority Queues

Round Robin Algorithm:

- The Round Robin algorithm is closely related to the concept of queues, as it uses a queue data structure to manage the execution of processes.
- The algorithm effectively utilizes the properties of the queue data structure to achieve fair and time-shared execution of processes.
- The ready queue manages the processes waiting to be executed, and the cyclic and preemptive behavior of the algorithm is accomplished through queue operations.
- The Round Robin algorithm's application of queues makes it a widely used and efficient CPU scheduling strategy in operating systems and time-sharing environments.

Applications of Queues- Round Robin Algorithm,Circular Queues, DeQueue and Priority Queues (Contd….)

Types of Queues:

- Circular Queue – Elements are represented in a circular fashion. Insertion is done at very first location if last location is full.
- Double Ended Queue – Elements can be inserted or deleted from both ends.
- Priority Queue- Each element is assigned with a priority. An element with highest priority is processed first. Two elements with highest priority is in FIFO order.

Applications of Queues- Round Robin Algorithm,Circular Queues, DeQueue and Priority Queues (Contd….)

Circular Queue:

Circular Queue is a linear data structure in which the operations are performed based on FIFO (First In First Out) principle and the last position is connected back to the first position to make a circle. It is also called 'Ring Buffer'.

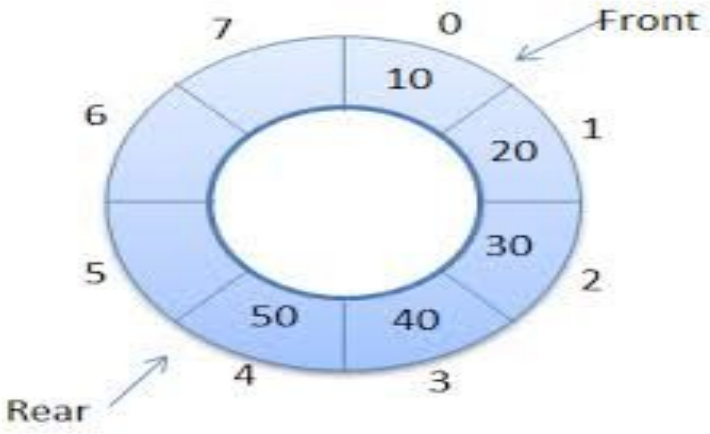


Figure : Representation of Circular Queue

Applications of Queues- Round Robin Algorithm,Circular Queues, DeQueue and Priority Queues (Contd….)

Circular Queues: Operations of Circular Queue

- Front: Get the front item from queue.
- Rear: Get the last item from queue.
- enQueue(value) This function is used to insert an element into the circular queue. In a circular queue, the new element is always inserted at Rear position.

Steps: 1. Check whether queue is Full – Check ((rear == SIZE-1 && front == 0) || (rear == front-1)).

2. If it is full then display Queue is full. If queue is not full then, check if (rear == SIZE – 1 && front != 0) if it is true then set rear=0 and insert element.

Applications of Queues- Round Robin Algorithm,Circular Queues, DeQueue and Priority Queues (Contd….)

- deQueue() This function is used to delete an element from the circular queue. In a circular queue, the element is always deleted from front position.

Applications of Circular Queue

- Memory Management: The unused memory locations in the case of ordinary queues can be utilized in circular queues.
- Traffic system: In computer controlled traffic system, circular queues are used to switch on the traffic lights one by one repeatedly as per the time set.
- CPU Scheduling: Operating systems often maintain a queue of processes that are ready to execute or that are waiting for a particular event to occur.

Applications of Queues- Round Robin Algorithm,Circular Queues, DeQueue and Priority Queues (Contd….)

Dequeue:

- Dequeue is a generalized version of Queue data structure that allows insert and delete at both ends.

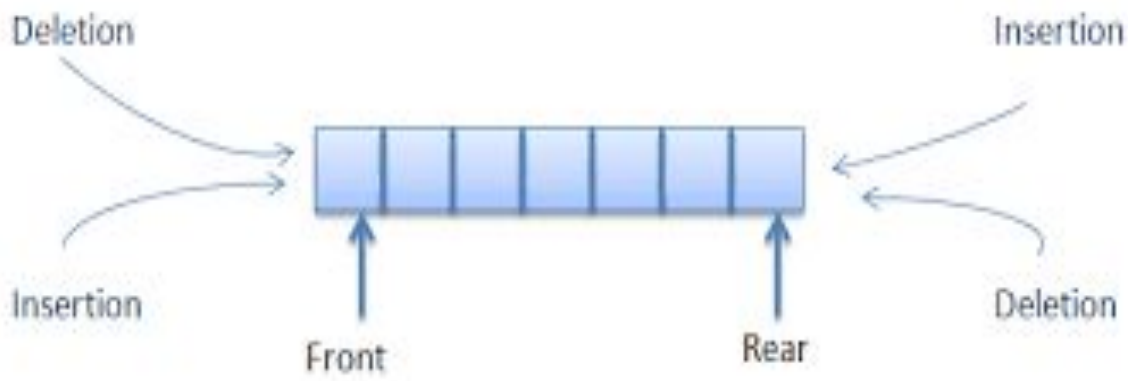


Figure : Representation of Double Ended Queue

Applications of Queues- Round Robin Algorithm,Circular Queues, DeQueue and Priority Queues (Contd….)

Operations of Dequeue

insertFront(): Adds an item at the front of Deque.

insertLast(): Adds an item at the rear of Deque.

delete Front(): Deletes an item from front of Deque.

deleteLast(): Deletes an item from rear of Deque.

- Following operations are also supported

getFront(): Gets the front item from queue

getRear(): Gets the last item from queue

isEmpty(): Checks whether Deque is empty or not

IsFull(): Checks whether Deque is full or not.

Applications of Queues- Round Robin Algorithm,Circular Queues, DeQueue and Priority Queues (Contd….)

Applications of Dequeue

Supports both stack and queue operations, it can do the following:

- The De-queue data structure supports clockwise and anticlockwise rotations in O(1) time which can be useful in certain applications.
- The problems where elements need to be removed and or added both ends can be efficiently solved using Dequeue.
- CPU Scheduling
- Graph algorithms like Dijkstra's Shortest Path Algorithm, Prim's Minimum Spanning Tree, etc.
- All queue applications where priority is involved

Applications of Queues- Round Robin Algorithm,Circular Queues, DeQueue and Priority Queues (Contd….)

Priority Queue:

Priority Queue is an extension of queue with the following properties:

1. Every item has a priority associated with it.
2. An element with high priority is dequeued before an element with low priority.
3. If two elements have the same priority, they are served according to their order in the queue.

Applications of Queues- Round Robin Algorithm,Circular Queues, DeQueue and Priority Queues (Contd….)

Operations of Priority Queue

- insert(item, priority): Inserts an item with given priority.
- getHighestPriority(): Returns the highest priority item.
- deleteHighestPriority(): Removes the highest priority item.

Applications of Queues- Round Robin Algorithm,Circular Queues, DeQueue and Priority Queues (Contd….)

Applications of General Queue

- When a resource is shared among multiple consumers.

  Examples include CPU scheduling, Disk Scheduling.

- When data is transferred asynchronously (data not necessarily received at same rate as sent) between two processes.

  Examples include IO Buffers, pipes, file IO, etc.

Round Robin Scheduling Algorithm:

Round Robin is a preemptive scheduling algorithm commonly used in operating systems for time-sharing environments. It ensures that each process gets a fixed time slice before moving to the next process. If a process doesn't complete within its time slice, it's moved to the back of the queue.

```
#include <stdio.h>
void roundRobin(int processes[], int n, int burst_time, int quantum) {
    int remaining_time[n];
    for (int i = 0; i < n; i++) {
        remaining_time[i] = burst_time;
    }
```

Round Robin Scheduling Algorithm:

```
int time = 0;
    while (1) {
        int done = 1;
        for (int i = 0; i < n; i++) {
            if (remaining_time[i] > 0) {
                done = 0;
                if (remaining_time[i] > quantum) {
                    time += quantum;
                    remaining_time[i] -= quantum;
                } else {
```

Round Robin Scheduling Algorithm:

```
    time += remaining_time[i];
                remaining_time[i] = 0;
            }
    printf("Process %d finished at time %d\n", i + 1, time);
        }
    }
    if (done)
        break;
  }
}
```

Round Robin Scheduling Algorithm:

```
int main() {
    int processes[] = {1, 2, 3};
    int n = sizeof(processes) / sizeof(processes[0]);
    int burst_time[] = {10, 5, 8};
    int quantum = 2;
    roundRobin(processes, n, burst_time, quantum);
    return 0;
}
```

Circular Queue:

A circular queue is a variation of the basic queue with its front and rear connected, allowing the elements to wrap around.

```
#define MAX_SIZE 5
typedef struct {
    int items[MAX_SIZE];
    int front;
    int rear;
    int size;
} CircularQueue;
```

Circular Queue (Contd….)

```
void enqueue(CircularQueue *q, int data) {
    if ((q->rear + 1) % MAX_SIZE == q->front) {
        printf("Queue is full.\n");
    } else {
        q->rear = (q->rear + 1) % MAX_SIZE;
        q->items[q->rear] = data;
        q->size++;
    }
}
```

Circular Queue (Contd….)

```
int dequeue(CircularQueue *q) {
    if (q->front == q->rear) {
        printf("Queue is empty.\n");
        return -1;
    } else {
        q->front = (q->front + 1) % MAX_SIZE;
        int data = q->items[q->front];
        q->size--;
        return data;
    }
}
```

Circular Queue

Here's a simple algorithm for a double-ended queue (deque) application: a task manager. In this application, tasks can be added to the front or rear of the deque, and they can be completed and removed from both ends of the deque. This algorithm outlines the steps to perform basic operations on the deque in the context of a task manager application.

Create a Deque: Initialize a deque data structure to manage tasks. The deque will have a front and rear pointer initially set to -1 to indicate an empty deque.

Add Task to Front: To add a task to the front of the deque, perform the following steps:

- Check if the deque is full.
- If the deque is empty, set both the front and rear pointers to 0.
- Otherwise, update the front pointer according to the circular buffer logic.
- Add the task to the deque at the position indicated by the front pointer.

Circular Queue

Add Task to Rear: To add a task to the rear of the deque, perform similar steps as adding to the front, but update the rear pointer instead.

Remove and Complete Task from Front: To remove and complete a task from the front of the deque, perform the following steps:

- Check if the deque is empty.
- Retrieve the task at the front position.
- If the deque becomes empty after removal, update both the front and rear pointers to -1.
- Otherwise, update the front pointer according to the circular buffer logic.

Remove and Complete Task from Rear: To remove and complete a task from the rear of the deque, perform similar steps as removing from the front, but update the rear pointer instead.

Circular Queue

- **Print Deque:** Optionally, provide a function to print the tasks in the deque from front to rear, showing their order.
- **Free Memory:** When the application is done using the deque, ensure that the memory allocated for the deque structure and any associated nodes is properly freed to avoid memory leaks.
- **Main Program Loop:** In the main program loop, provide a menu for the user to interact with the task manager. Options could include adding tasks to the front or rear, completing tasks from the front or rear, printing the task list, and exiting the application.

Linked Lists: Introduction

- Linked List can be considered as train where Engine is head pointer, Coaches are Nodes, Connectors connecting the coaches are pointers.
- Engine is connected to first coach can be considered as head pointing to the first Node.
- Last coach do not connects with any coach further, it can be considered as Last Node in which link part(connector part) points to NULL.
- Passengers inside the coaches can be considered as Data part of the Nodes.
- As the coaches may be easily added or removed from the train as per the requirement so is the case with Linked List.

Linked Lists: Introduction (Contd….)

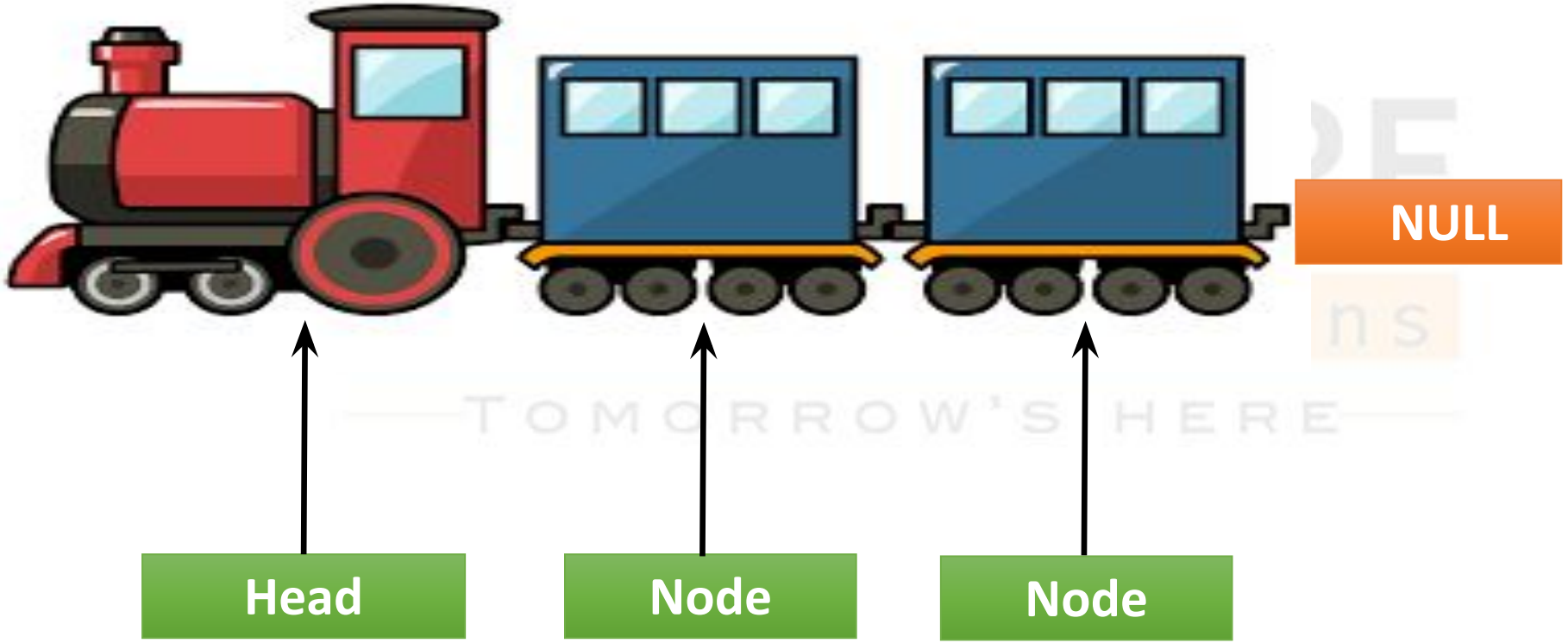


Figure : Linked List representation by real life example

Linked Lists: Introduction (Contd….)

Linked List Basics:

- Linked lists and arrays are similar since they both store collections of data.
- The array's features all follow from its strategy of allocating the memory for all its elements in one block of memory.
- Linked lists allocate memory for each element separately and only when necessary.
- Linked lists are appropriate when the number of data elements to be represented in the data structure at once is unpredictable.
- Linked lists are dynamic, so the length of a list can increase or decrease as necessary.
- Each node does not necessarily follow the previous one physically in the memory.
- Linked lists can be maintained in sorted order by inserting or deleting an element at the proper point in the list.

Linked Lists: Introduction (Contd….)

Types of Linked List: There are four basic types of linked list

- Single Linked List
- Doubly linked list
- Circular linked list
- Circular doubly linked list

Single Linked List

We can think of linked lists as a network of people in which one person has some data and also has a note of another person's address of location.

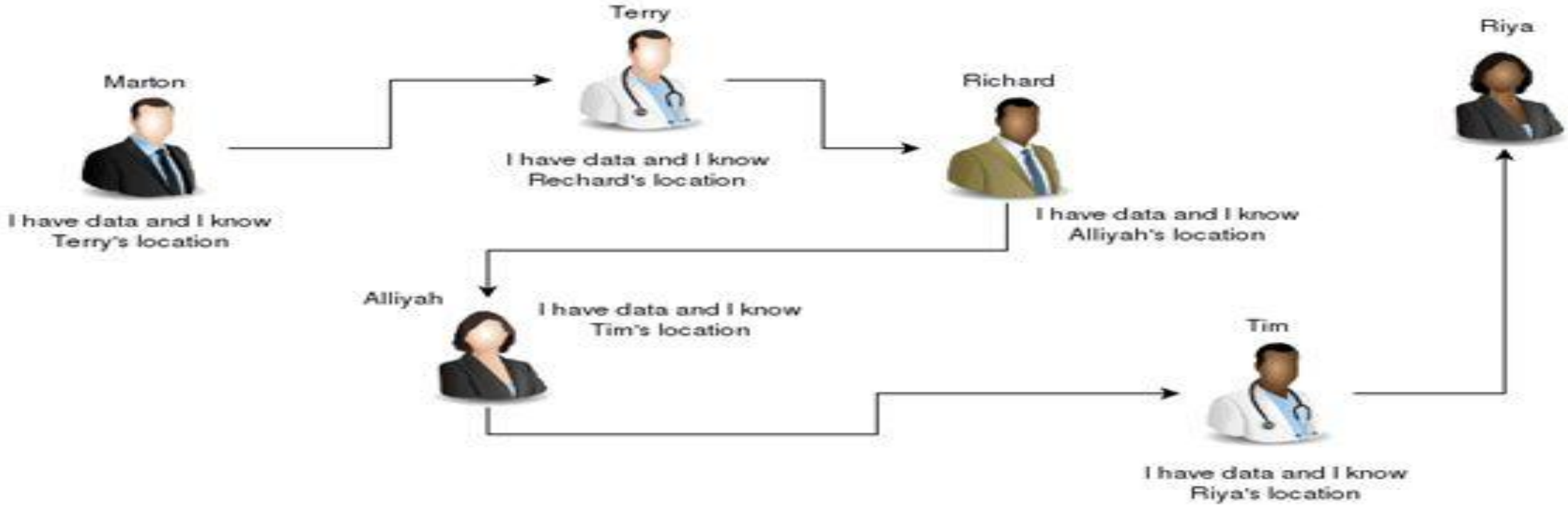


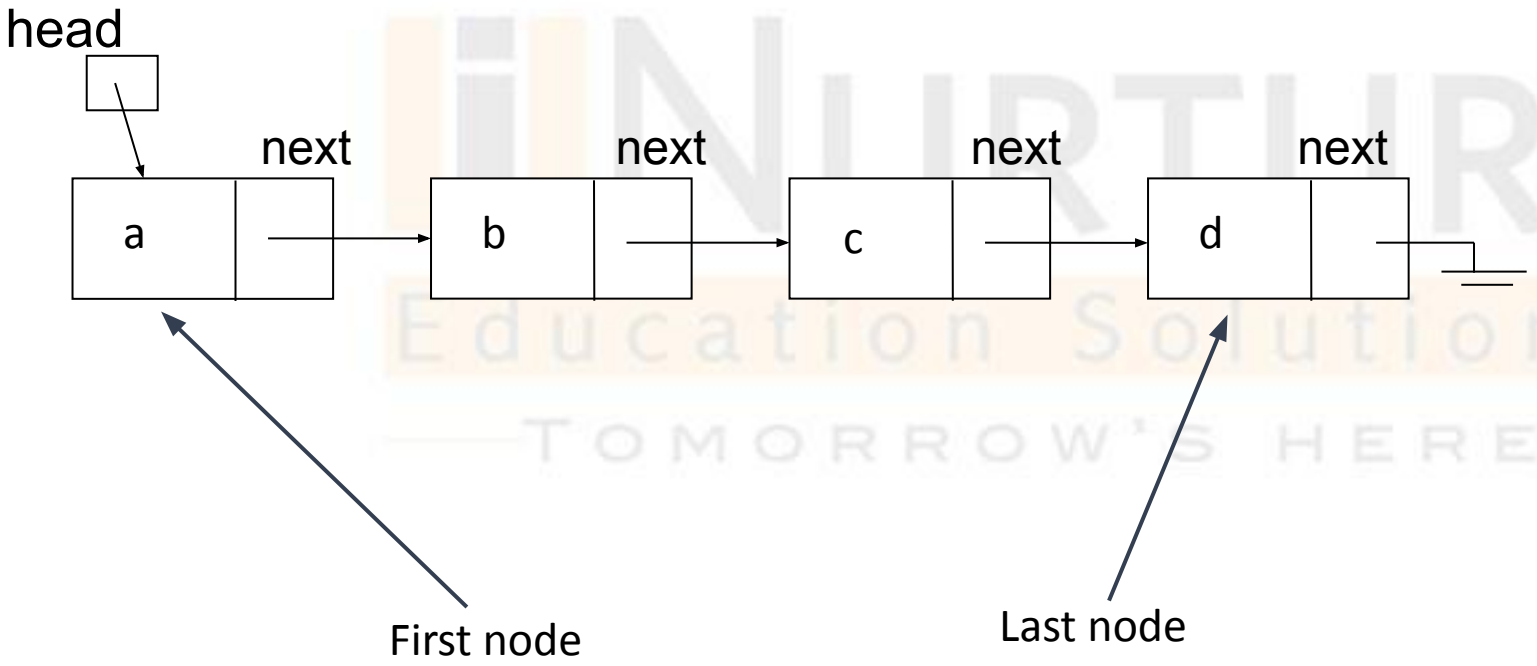
Figure : Network of People an analogy to Single Linked List
Source: *https://www.c-sharpcorner.com*

Single Linked List (Contd….)

- A single linked list is one in which all nodes are linked together in some sequential manner.
- Each node has only one link part .
- Each link part contains the address of the next node in the list .
- Link part of the last node contains NULL value which signifies the end of the node.

Single Linked List (Contd….)

Representation of a linked list in memory



- Empty Linked list is a single pointer having the value of NULL.head = NULL;

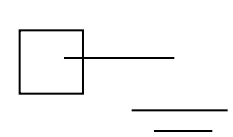


Figure: Schematic representation of Single Linked List

Different Operations on a Single linked list

Insertion: New nodes can be inserted at the beginning, end, or any position in the list. To insert a node, we update the next pointer of the new node and the previous node accordingly.

Deletion: Nodes can be removed from the list by updating the next pointers of the adjacent nodes to bypass the node to be deleted. The memory occupied by the deleted node is then freed.

Traversal: To access the data in each node, we start from the head (first node) and follow the next pointers until we reach the end of the list (where the next pointer is NULL).

Searching: We can search for a specific element by traversing the list and comparing the data in each node until we find the desired element or reach the end of the list.

Different Operations on a Single linked list (Contd….)

Insertion in a linked list

tmp = new Node;

tmp->element = x;

tmp->next = current->next;

current->next = tmp;

Or simply (if Node has a constructor initializing its members):

current->next = new Node(x,current->next);

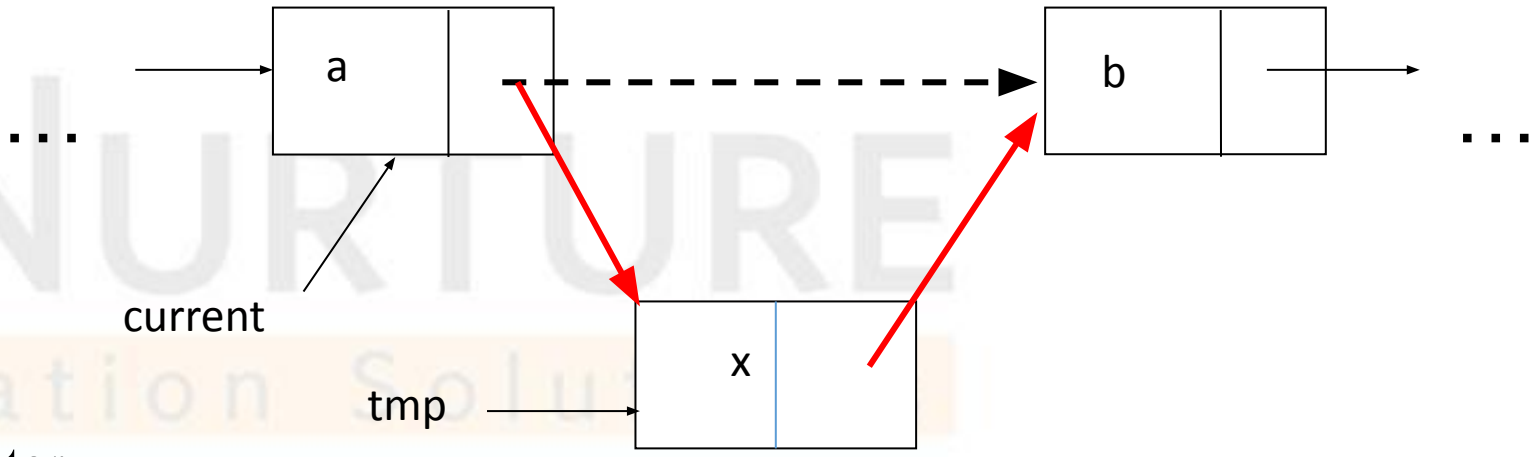


Figure : Insertion operation on Single Linked List

Different Operations on a Single linked list (Contd….)

Deletion from a linked list

```
Node *deletedNode = current->next;
current->next = current->next->next;
delete deletedNode;
```

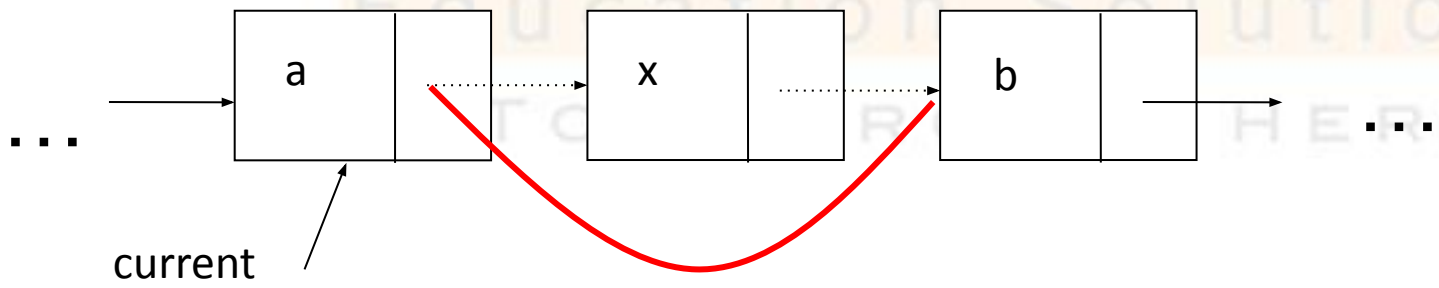


Figure : Deletion operation on Single Linked List

Different Operations on a Single linked list (Contd….)

Traversing a linked list

```
Node *pWalker;
int count = 0;
cout <<“List contains:\n”;
for (pWalker=pHead; pWalker!=NULL;
 pWalker = pWalker->next)
{     count ++;
   cout << pWalker->element << endl;  }
```

Different Operations on a Single linked list (Contd….)

Searching a node in a linked list

```
pCur = pHead;
// Search until target is found or we reach
// the end of list
while (pCur != NULL &&
      pCur->element != target)
{ pCur = pCur->next;}
//Determine if target is found
if (pCur) found = 1;
else found = 0;
```

Different Operations on a Single linked list (Contd….)

Special Cases (1)

Inserting before the first node (or to an empty list):

```
tmp = new Node;
tmp->element = x;
if (current == NULL)
{ tmp->next = head;
   head = tmp;}
 else {// Adding in middle or at end
      tmp->next = curent->next;
      current->next = tmp;}
```

Different Operations on a Single linked list (Contd….)

Special Cases (2)

```
Node *deletedNode;
if (current == NULL){
   // Deleting first node
   deletedNode = head;
   head = head ->next;}
else{  // Deleting other nodes
   deletedNode = current->next;
   current->next = deletedNode ->next;}
delete deletedNode;
```

Different Operations on a Single linked list (Contd….)

Header Nodes

One problem with the basic description: it assumes that whenever an item x is removed (or inserted) some previous item is always present.

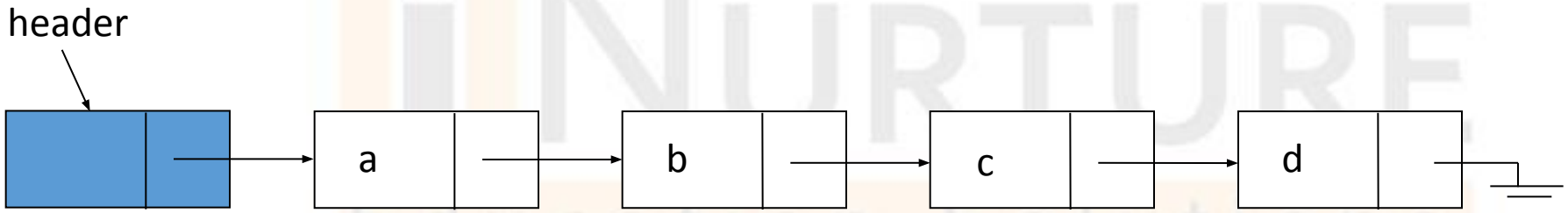
Consequently removal of the first item and inserting an item as a new first node become special cases to consider.

In order to avoid dealing with special cases: introduce a header node (dummy node).

A header node is an extra node in the list that holds no data but serves to satisfy the requirement that every node has a previous node.

Different Operations on a Single linked list (Contd….)

List with a header node



Empty List

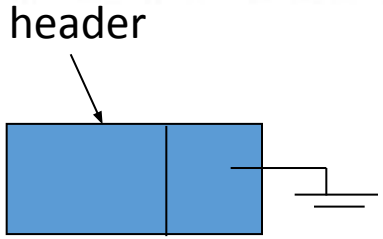


Figure : Linked list with elements and Empty List

Reversing a Single Linked List

- To reverse a single linked list, Need to change the direction of the next pointers for each node.
- Start with three pointers: one pointing to the current node, one to the previous node (initially NULL), and one to the next node.
- Update the next pointer of the current node to point to the previous node and then move the three pointers one step forward until we reach the end of the list.
- The last node becomes the new head of the reversed list.

Advantages and disadvantages of single linked list

Advantages:

Dynamic Size: Linked lists can grow or shrink as needed since memory allocation is done dynamically, unlike arrays with fixed sizes.

Insertion and Deletion: Insertion and deletion operations are efficient in linked lists, especially in the middle of the list, as they involve updating pointers rather than shifting elements.

Memory Utilization: Linked lists efficiently utilize memory as they allocate memory only for the data and the next pointer, without requiring contiguous memory.

Advantages and disadvantages of single linked list

Disadvantages:

Random Access: Accessing an element at a specific index in the list requires sequential traversal from the head, resulting in slower random access compared to arrays.

Extra Memory: Each node in the linked list requires additional memory to store the next pointer, which can lead to higher memory overhead compared to arrays.

Circular linked list

We may understand the circular linked list by taking example of clock as clock arms return to same number from where it starts we can understand number displayed on clock as nodes where elements are stored. Each node points to next node and the last node points to first node.

One application of circular linked list is implementation of Round Robin Scheduling policy in operating system. Where each process runs for equal time slice then CPU is scheduled to next process in list. This goes on until last process get its share of time, executes and then gain CPU is allocated to first process in the queue.

Clock as an analogy to traversal in Circular Linked List

Source: https://mashable.com

Circular linked list (Contd….)

- A Circular Linked List is a special type of Linked List.
- It supports traversing from the end of the list to the beginning by making the last node point back to the head of the list. A Rear pointer is often used instead of a Head pointer.

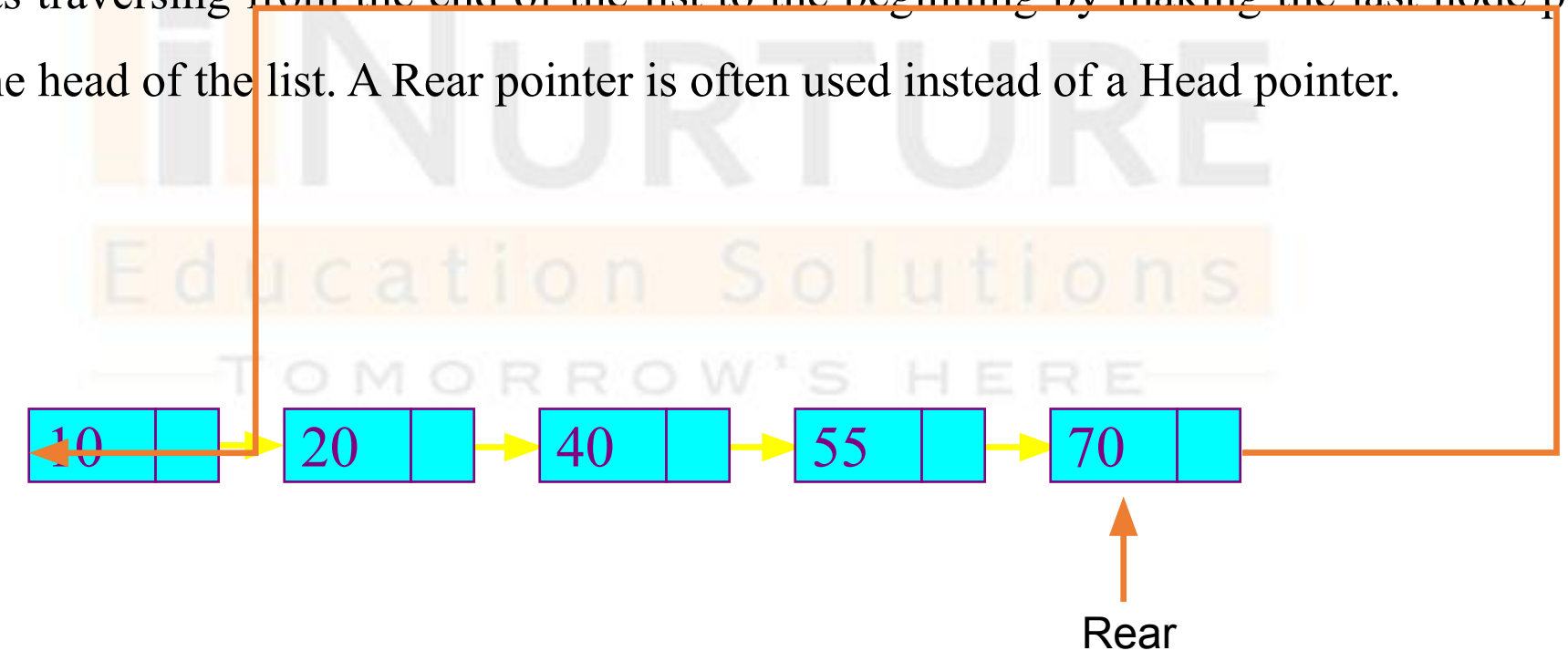


Figure: Schematic representation of circular linked list

Circular linked list (Contd….)

Motivation

Circular linked lists are usually sorted.

Circular linked lists are useful for playing video and sound files in "looping" mode.

They are also a stepping stone to implementing graphs, an important topic in comp171.

Circular Linked List Definition

```
#include <iostream>
using namespace std;
struct Node
{int data;
Node* next;};
typedef Node* NodePtr;
```

Circular linked list (Contd….)

Circular Linked List Operations

- insertNode(NodePtr& Rear, int item)

  //add new node to ordered circular linked list

- deleteNode(NodePtr& Rear, int item)

  //remove a node from circular linked list

- print(NodePtr Rear)

  //print the circular linked list once

Circular linked list (Contd….)

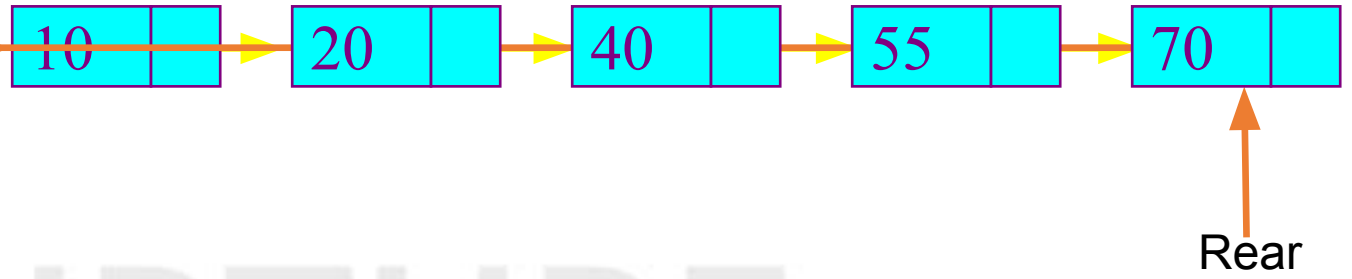


Figure : Traversal **operation on Circular Linked List**

Traverse the list

```
void print(NodePtr Rear){
        NodePtr Cur;
        if(Rear != NULL)
{       Cur = Rear->next;
         do{    cout << Cur->data << " ";
                Cur = Cur->next;
          }while(Cur != Rear->next);
          cout << endl; }
 }
```

Circular linked list (Contd….)

Insert into an empty list

NotePtr New = new Node;

New->data = 10;

Rear = New;

Rear->next = Rear;

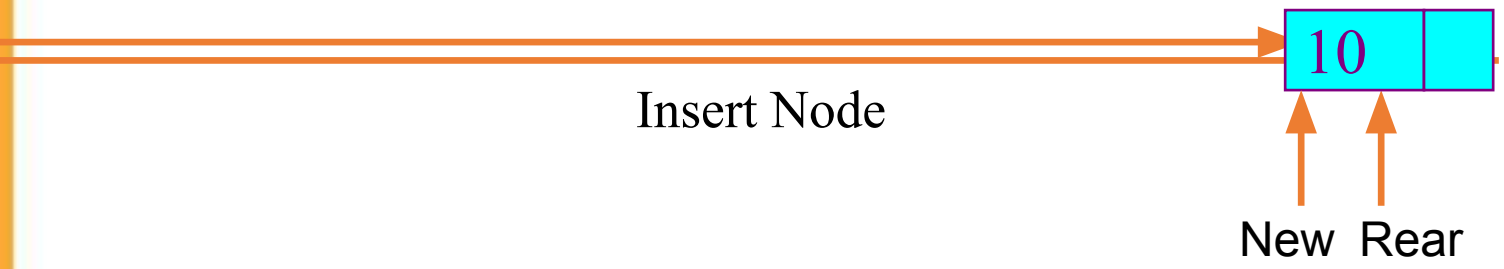


Figure : Adding first node to Circular Linked List

Circular linked list (Contd….)

Insert to head of a Circular Linked List

New->next = Cur; // same as: New->next = Rear->next;

Prev->next = New; // same as: Rear->next = New;

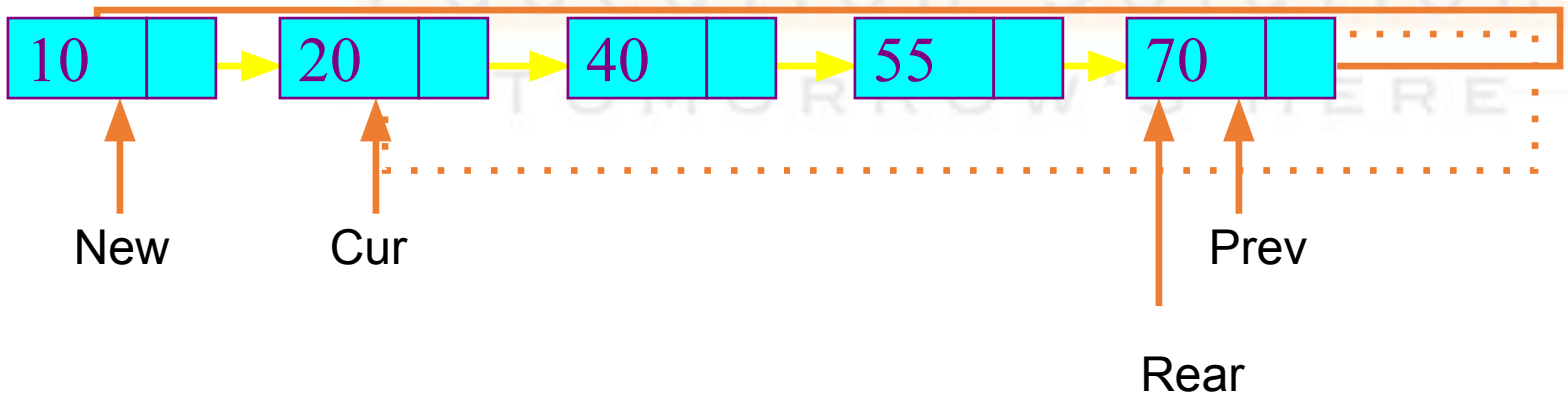


Figure : Insertion at head in Circular Linked List

Circular linked list (Contd….)

Insert to middle of a Circular Linked List between Prev and Cur

New->next = Cur;

Prev->next = New;

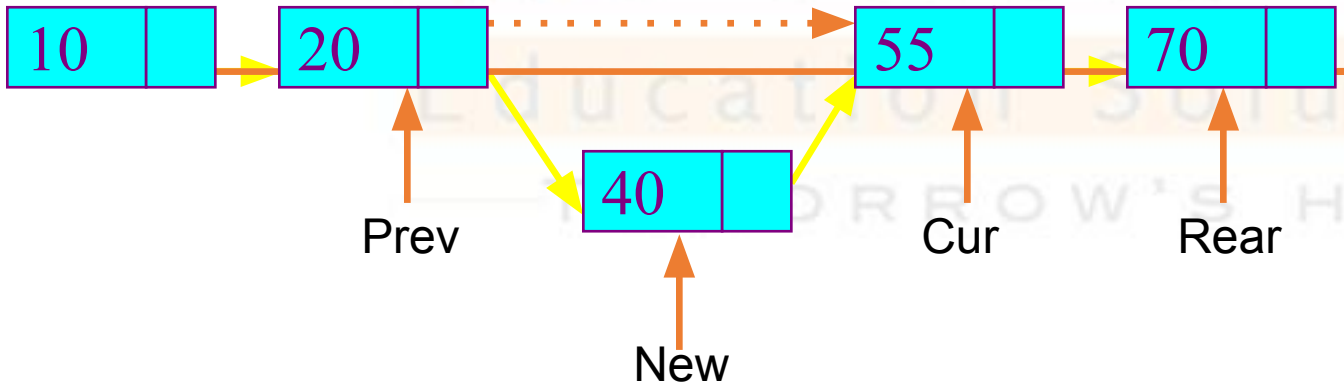


Figure : Insertion at middle in Circular Linked List

Circular linked list (Contd….)

Insert to end of a Circular Linked List

New->next = Cur; // same as: New->next = Rear->next;

Prev->next = New; // same as: Rear->next = New;

Rear = New;

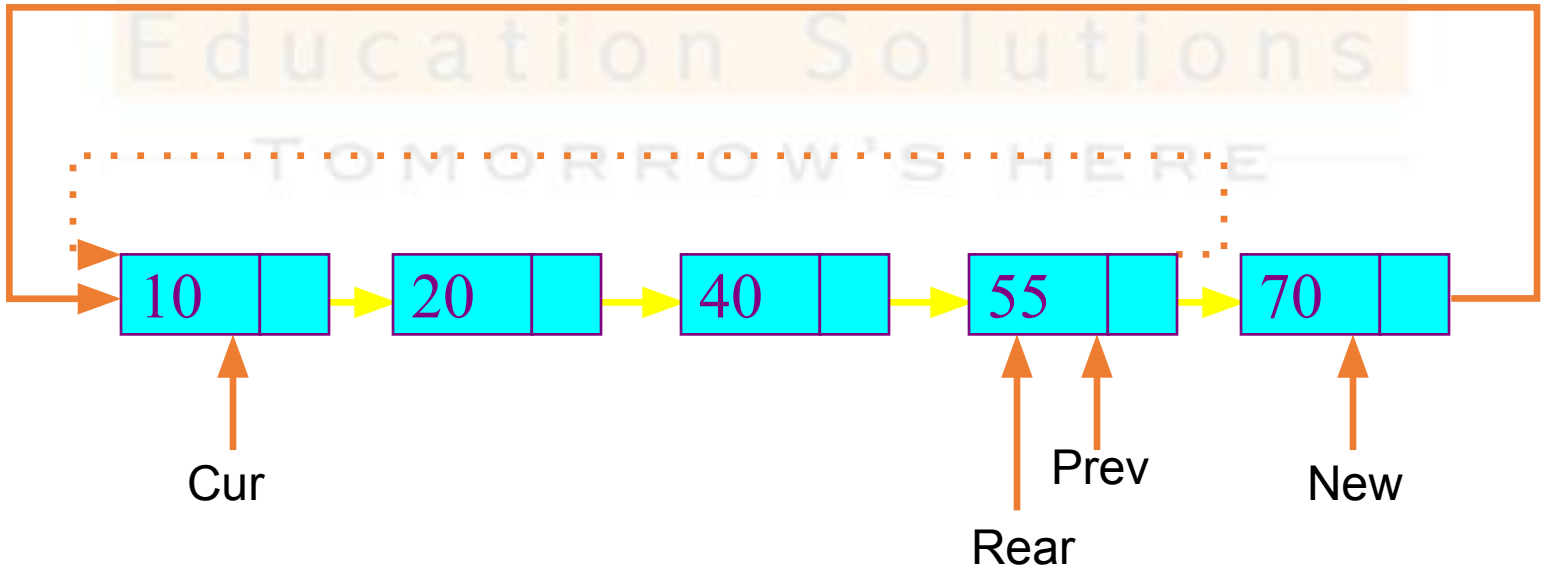


Figure : Insertion at end in Circular Linked List

Circular linked list (Contd….)

```
void insertNode(NodePtr& Rear, int item)
{
    NodePtr  New, Cur, Prev;
    New = new Node;
    New->data = item;
    if(Rear == NULL)
  {  // insert into empty list
        Rear = New;
        Rear->next = Rear;
        return;    }
```

Circular linked list (Contd….)

```
Prev = Rear;
        Cur = Rear->next;
        do{                     // find Prev and Cur
if(item <= Cur->data)
                break;
            Prev = Cur;
            Cur = Cur->next;
        }while(Cur != Rear->next);
        New->next = Cur;        // revise pointers
        Prev->next = New;
```

Circular linked list (Contd….)

```
if(item > Rear->data)  //revise Rear pointer if adding to end
        Rear = New;
}
```

Circular linked list (Contd….)

Delete a node from a single-node Circular Linked List

Rear = NULL;

delete Cur;

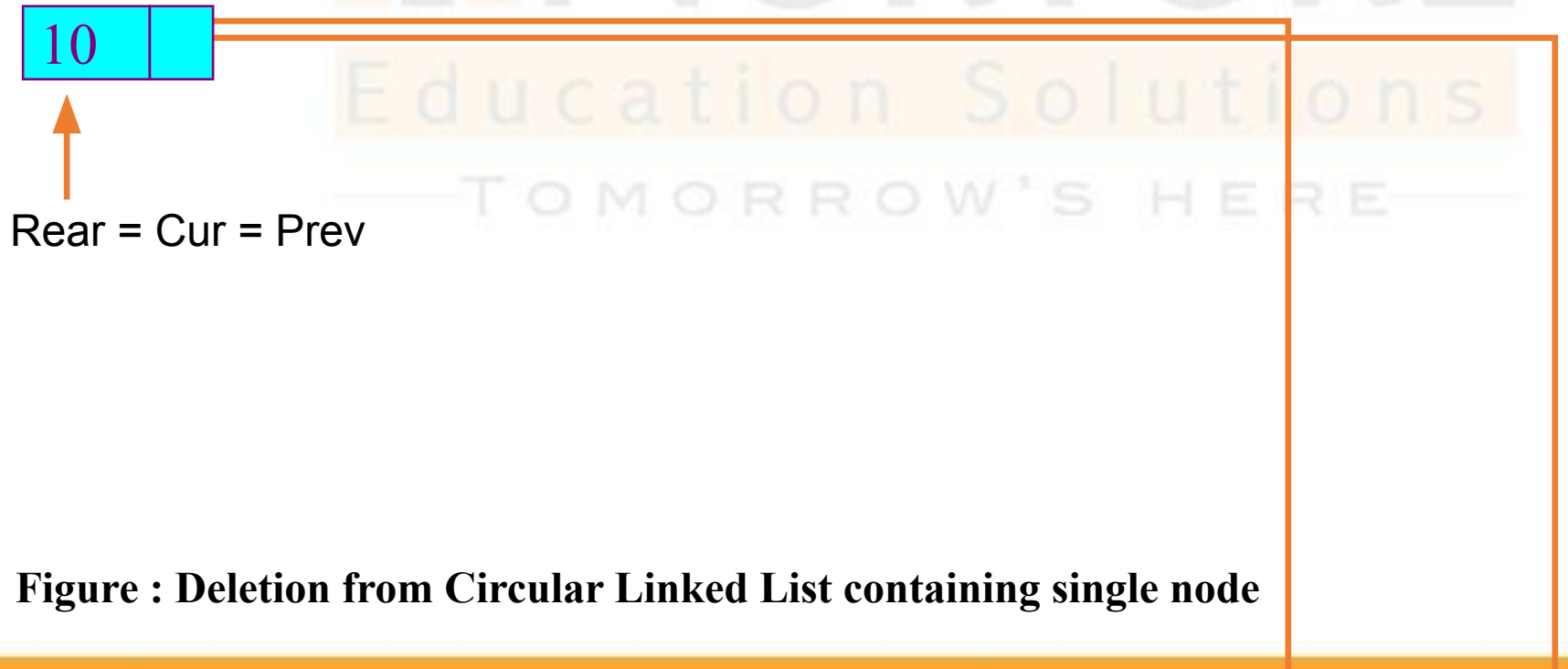


Figure : Deletion from Circular Linked List containing single node

Circular linked list (Contd….)

Delete Note

Delete the head node from a Circular Linked List

Prev->next = Cur->next; // same as: Rear->next = Cur->next delete Cur;

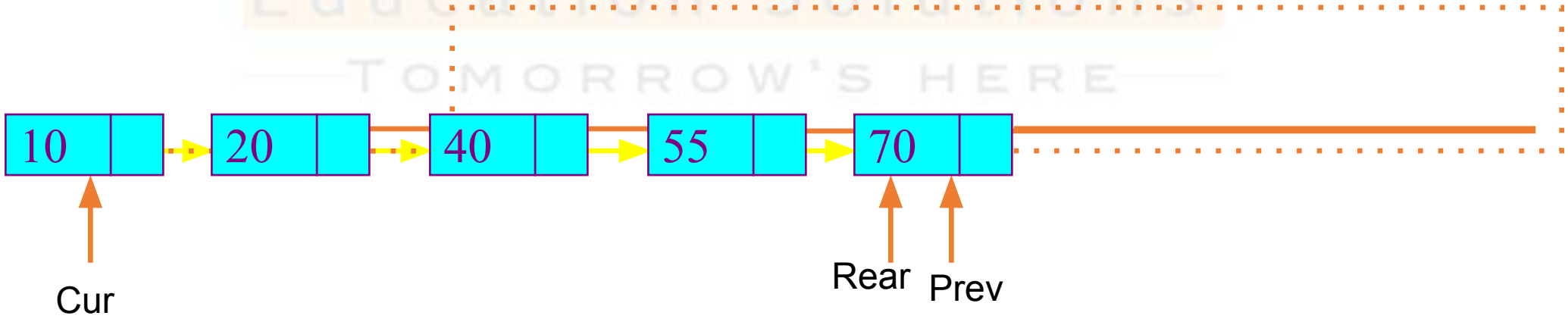


Figure : Deletion from Circular Linked List at beginning

Circular linked list (Contd….)

Delete the end node from a Circular Linked List

Prev->next = Cur->next; // same as: Rear->next;

delete Cur;

Rear = Prev;

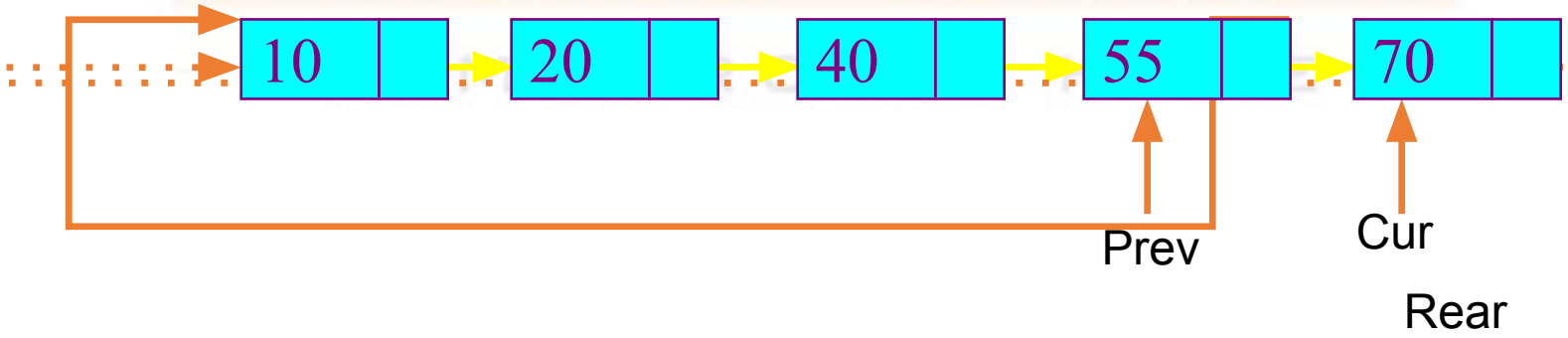


Figure : Deletion from Circular Linked List at the end

Circular linked list (Contd….)

Delete the end node from a Circular Linked List

Prev->next = Cur->next;

delete Cur;

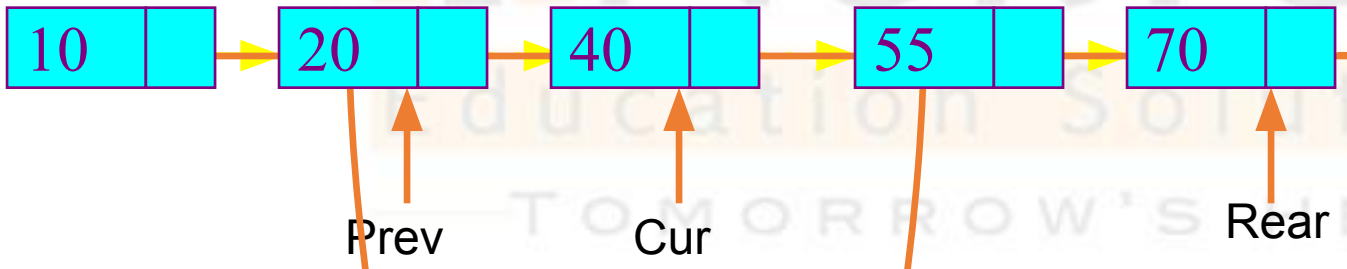


Figure: Deletion from Circular Linked List at middle

Circular linked list (Contd….)

```
void deleteNode(NodePtr& Rear, int item){
NodePtr Cur, Prev;
if(Rear == NULL){
        cout << "Trying to delete empty list" << endl;
        return;    }
    Prev = Rear;
    Cur = Rear->next;
    do{   // find Prev and Cur
        if(item <= Cur->data)  break;
        Prev = Cur;
        Cur = Cur->next; }while(Cur != Rear->next);
```

Circular linked list (Contd….)

```
if(Cur->data != item){// data does not exist
        cout << "Data Not Found" << endl;
        return;
    }
if(Cur == Prev){        // delete single-node list
        Rear = NULL;
        delete Cur;
        return;
    }
    if(Cur == Rear)         // revise Rear pointer if deleting end
        Rear = Prev;
```

Circular linked list (Contd….)

```
    Prev->next = Cur->next;    // revise pointers
        delete Cur;
    }
```

Circular linked list (Contd….)

```
void main(){
    NodePtr Rear = NULL;
    insertNode(Rear, 3);
    insertNode(Rear, 1);
    insertNode(Rear, 7);
    insertNode(Rear, 5);
    insertNode(Rear, 8);
    print(Rear);
    deleteNode(Rear, 1);
    deleteNode(Rear, 3);
```

Circular linked list (Contd….)

```
deleteNode(Rear, 8);
        print(Rear);
        insertNode(Rear, 1);
        insertNode(Rear, 8);
        print(Rear);
 }
```

Result is:

1 3 5 7 8

5 7

1 5 7 8

Doubly linked list

In Doubly linked list each node not only contains address of next node but also previous node, thus we can traverse in both the directions.

Also the as first node does not have any previous node so it contains NULL in address field pointing to previous node. Last node does not have next node so it contains NULL in address field pointing to next node.

Figure : Two one way bridges as an analogy to bidirectional traversal in Doubly Linked List

Doubly linked list (Contd….)

- Doubly linked list is a linked data structure that consists of a set of sequentially linked records called nodes.
- Each node contains three fields: one is data part which contain data only. Two other fields is links part that are point or references to the previous or to the next node in the sequence of nodes.
- The beginning and ending nodes' previous and next links, respectively, point to some kind of terminator, typically a sentinel node or null to facilitate traversal of the list.
- It is a way of going both directions in a linked list, forward and reverse.
- Many applications require a quick access to the predecessor node of some node in list

Doubly linked list (Contd….)

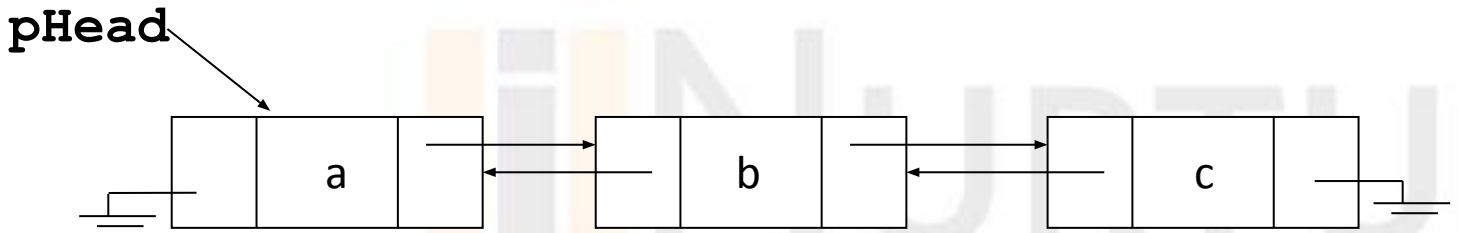


Figure : Schematic representation of Doubly Linked List

Doubly linked list (Contd….)

Advantages over Singly-linked Lists

- Quick update operations such as: insertions, deletions at both ends (head and tail), and also at the middle of the list.
- A node in a doubly-linked list store two references:
- A next link; that points to the next node in the list, and
- A prev link; that points to the previous node in the list.

Doubly linked list (Contd….)

- A doubly linked list provides a natural implementation of theList ADT.
- Nodes implement Position and store:
  - element
  - link to the previous node
  - link to the next node
- Special trailer and header nodes

Doubly linked list (Contd….)

Doubly Linked list is created using a Node structure having three parts two pointers one for previous node one for next node and third part is element which stores the data part.

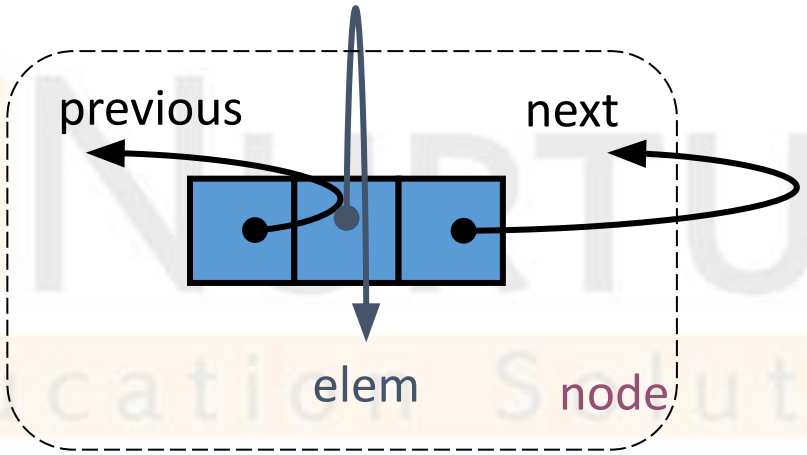
Like in the Figure given shows a doubly linked list with four nodes.

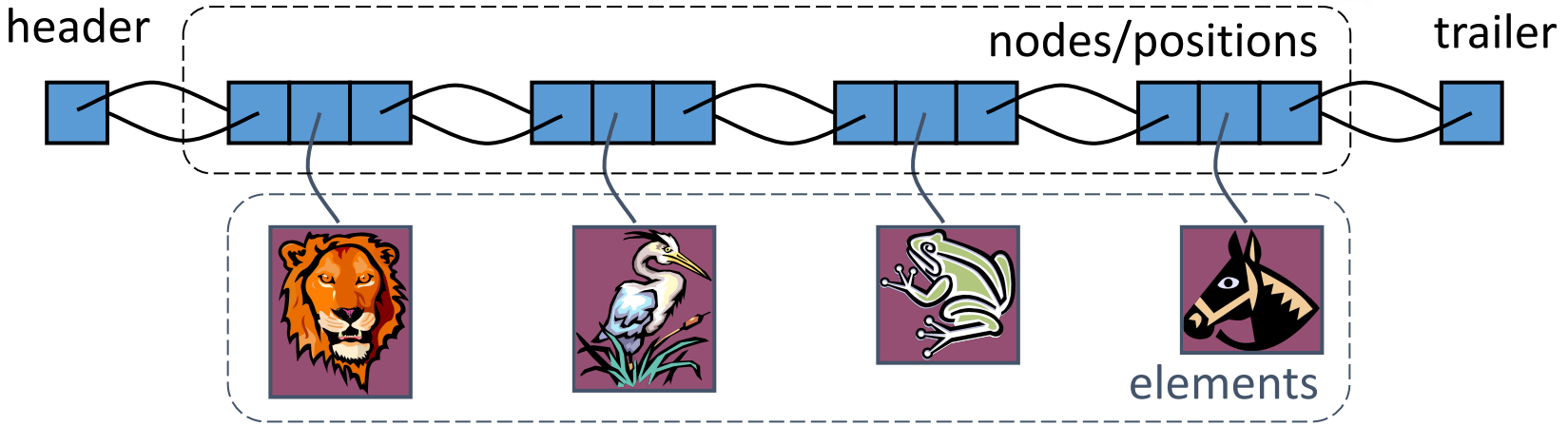
Head pointer points to first node as there is no previous node before first node therefore it points to header pointer. And Next node pointer points to next node and elements contains 'Lion' as data part .

Likewise for other nodes its shown.

In doubly linked list we can traverse in both direction such that from head to tail using next pointer in each node from tail to head using previous pointer in each node.

Doubly linked list (Contd….)





Doubly linked list (Contd….)

Sentinel Nodes

- To simplify programming, two special nodes have been added at both ends of the doubly-linked list.
- Head and tail are dummy nodes, also called sentinels, do not store any data elements.
- Head: header sentinel has a null-prev reference (link).
- Tail: trailer sentinel has a null-next reference (link).

Doubly linked list (Contd….)

What do we see from a Doubly-linked List?

A doubly-linked list object would need to store the following:

1. Reference to sentinel head-node;
2. Reference to sentinel tail-node; and
3. Size-counter that keeps track of the number of nodes in the list (excluding the two sentinels).

Doubly linked list (Contd….)

Empty Doubly-Linked List:

Using sentinels, we have no null-links; instead, we have:

head.next = tail

tail.prev = head

Single Node List:

Size = 1

This single node is the first node, and also is the last node:

first node is head.next

last node is tail.prev

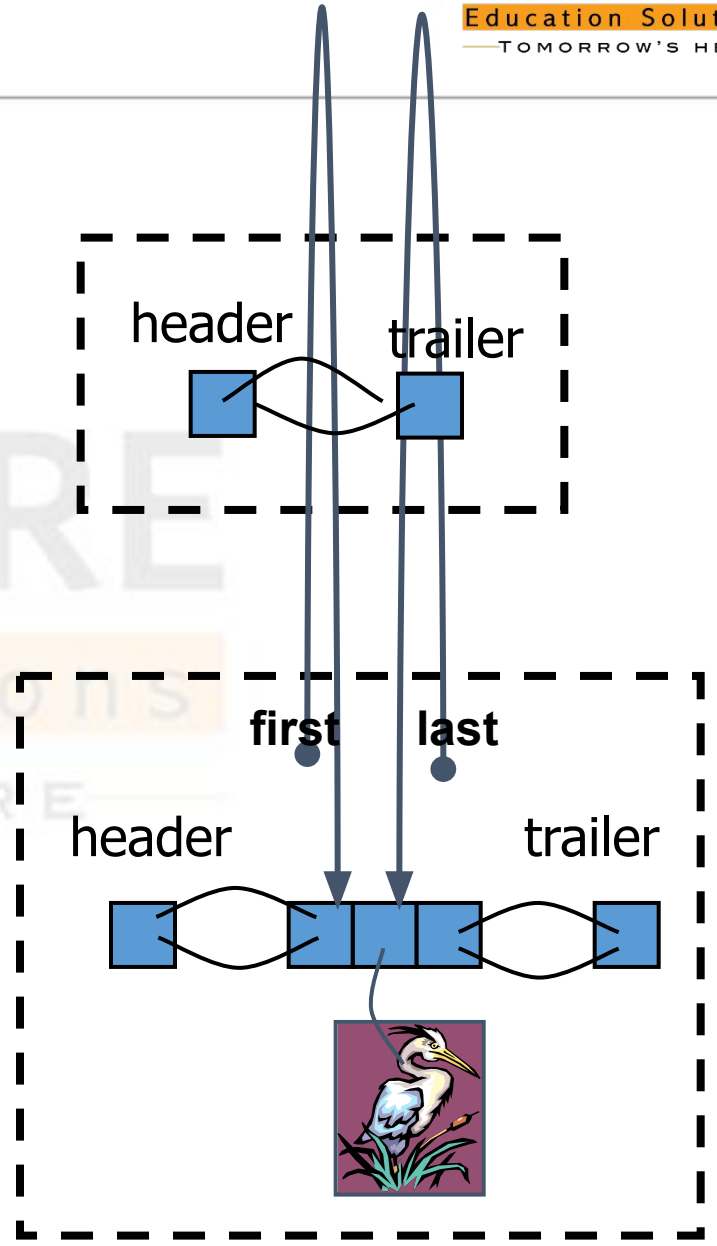


Figure : Doubly linked list in Empty state and with Single Node

Doubly linked list (Contd….)

Insertion into a Doubly-Linked List

1. AddFirst Algorithm

To add a new node as the first of a list:

Algorithm addFirst()

new(T)

T.data ← y

T.next ← head.next

T.prev ← head

head.next.prev ← T {Order is important}

head.next ← T

Size++ This Algorithm is valid also in case of empty list.

Doubly linked list (Contd….)

2. AddLast Algorithm

To add a new node as the last of list:

Algorithm addLast()

new(T)

T.data ← y

T.next ← tail

T.prev ← tail.prev

tail.prev.next ← T {Order is important}

tail.prev ← T

Size++

This Algorithm is valid also in case of empty list.

Doubly linked list (Contd….)

Removal from a Doubly-Linked List

3. RemoveLast Algorithm

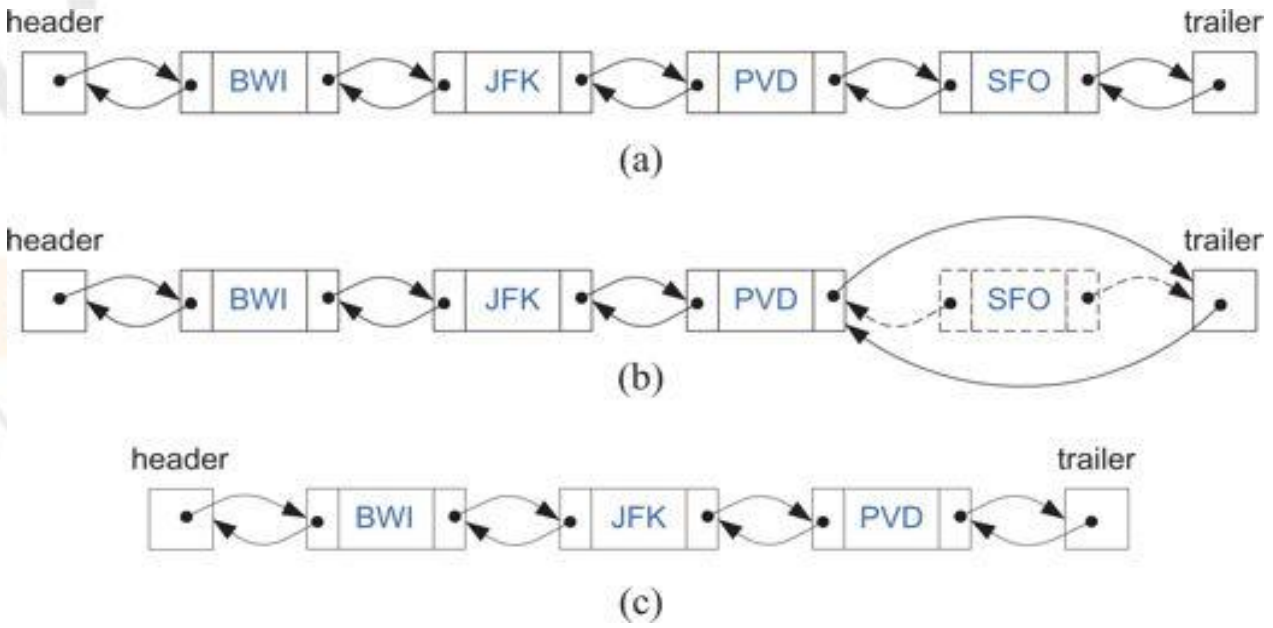


Figure : Steps showing deletion operation on doubly linked list

Notice that before removal, we must check for empty list. If not, we will remove the last node in the list, as shown in Figure above.

Doubly linked list (Contd….)

Algorithm remove Last()

```
If size = 0 then output "error"          else { T ← tail.prev
                                                y ← T.data
                                                T.prev.next ← tail
                                                tail.prev ← T.prev
                                                delete(T)          {garbage collector}
                                               size--
                                               return y
                                              }
```

This algorithm is valid also in case of a single node, size=1, in which case we'll get an empty list. Algorithm is one statement.

Doubly linked list (Contd….)

Insertion

- We visualize operation Add After(p, X), which returns position q

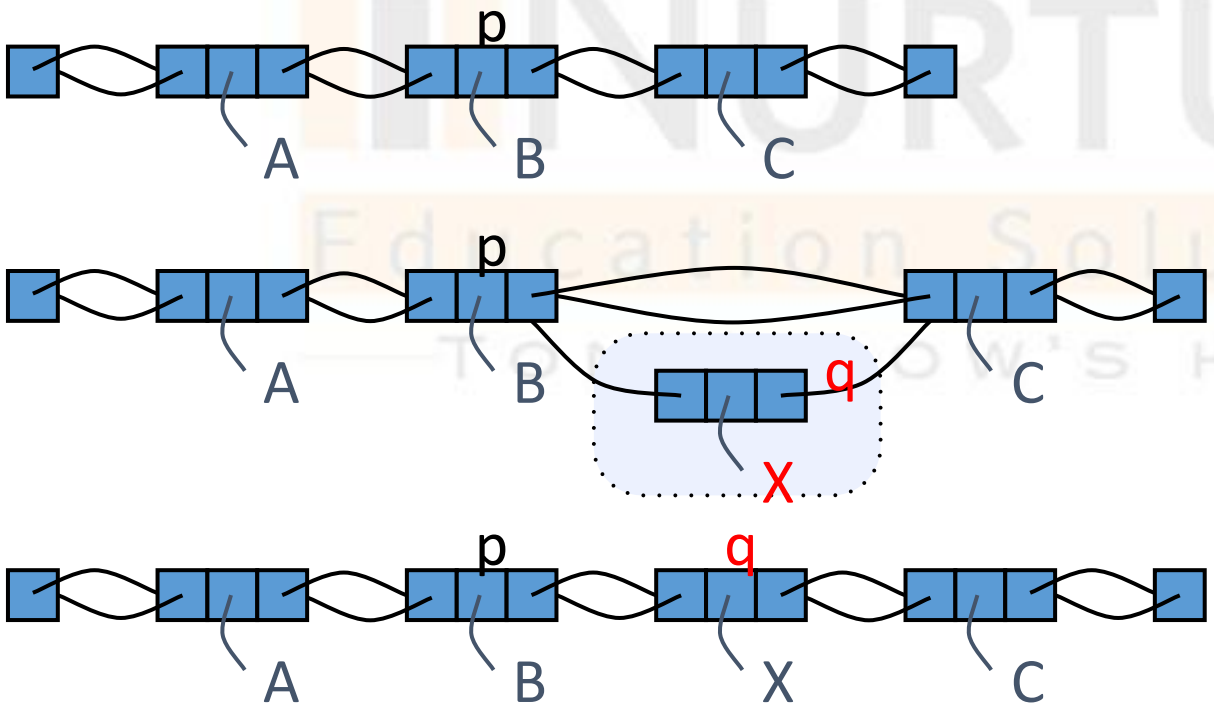


Figure : Steps showing insertion operation on doubly linked list

Doubly linked list (Contd….)

Insertion Algorithm

Algorithm insertAfter(p,e):

Create a new node v

v.setElement(e)

v.setPrev(p) {link v to its predecessor}

v.setNext(p.getNext()){link v to its successor}

(p.getNext()).setPrev(v) {link p's old successor to v}

p.setNext(v) {link p to its new successor, v}

return v {the position for the element e}

Doubly linked list (Contd….)

Deletion

- We visualize remove(p), where p = last()

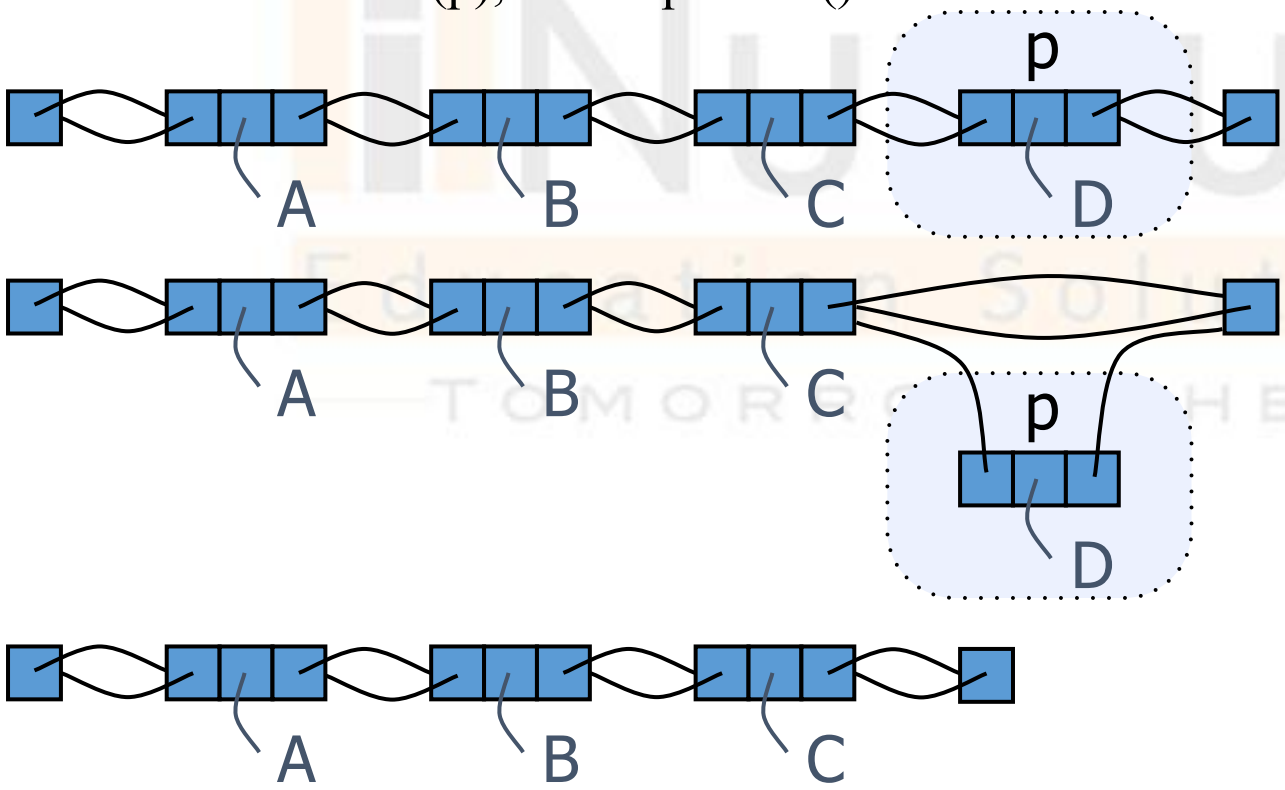


Figure : Steps showing deletion operation on doubly linked list

Doubly linked list (Contd….)

Deletion Algorithm

Algorithm remove(p):

t = p.element {a temporary variable to hold the return value}

(p.getPrev()).setNext(p.getNext()) {linking out p}

(p.getNext()).setPrev(p.getPrev())

p.setPrev(null) {invalidating the position p}

p.setNext(null)

return t

Doubly linked list (Contd….)

Doubly Linked List

Advantages:

- Convenient to traverse the list backwards.
- Simplifies insertion and deletion because you no longer have to refer to the previous node.

Disadvantage:

- Increase in space requirements.

Header Linked List

A header node is a special node that is found at the beginning of the list. A list that contains this type of node, is called the header-linked list. This type of list is useful when information other than that found in each node is needed.

For example:

Suppose there is an application in which the number of items in a list is often calculated. Usually, a list is always traversed to find the length of the list. However, if the current length is maintained in an additional header node that information can be easily obtained.

Header Linked List (Contd….)

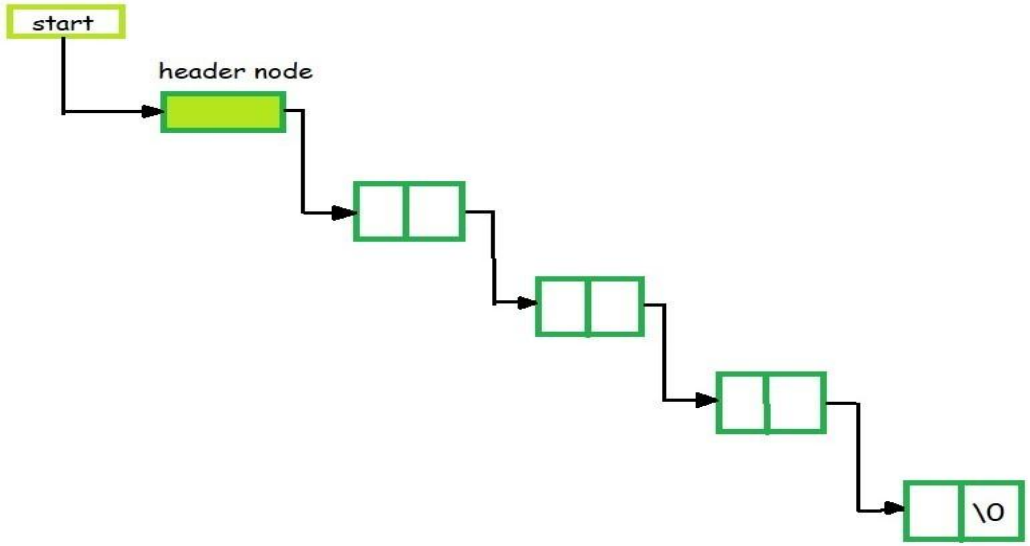
Types of Header Linked List:

- Grounded Header Linked List
- Circular Header Linked List

Header Linked List (Contd….)

Grounded Header Linked List

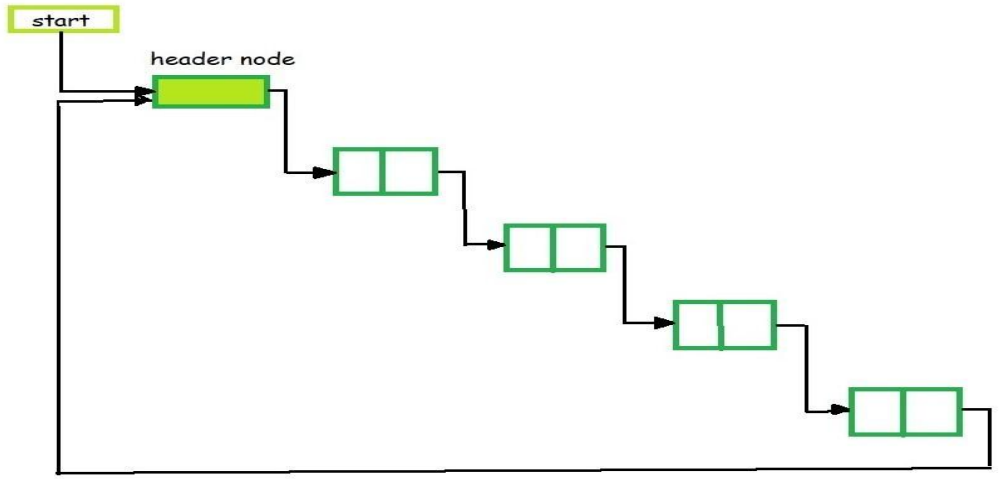
It is a list whose last node contains the NULL pointer. In the header linked list the start pointer always points to the header node. start -> next = NULL indicates that the grounded header linked list is empty. The operations that are possible on this type of linked list are Insertion, Deletion, and Traversing.



Header Linked List (Contd….)

Circular Header Linked List

A list in which last node points back to the header node is called circular linked list. The chains do not indicate first or last nodes. In this case, external pointers provide a frame of reference because last node of a circular linked list does not contain the NULL pointer. The possible operations on this type of linked list are Insertion, Deletion and Traversing.



Header Linked List (Contd….)

Header Linked Lists are not as commonly used as regular linked lists, but they simplify list operations by ensuring that a starting point (the header node) always exists, even when the list is empty. This consistent starting point eliminates the need for special cases when performing operations on an empty list, making code implementation more straightforward and reducing the chances of errors.

Header Linked List (Contd….)

Header Node:

- The header node is the first node in the linked list, placed before the actual data nodes.
- It is a special node that does not hold any meaningful data; its purpose is to act as a sentinel or marker for the start of the linked list.
- The header node contains only two fields:

1. Data: Empty or dummy data (usually NULL or a specific value to distinguish it from valid data).
2. Next Pointer: Points to the first "real" data node in the list.

Header Linked List (Contd….)

Empty List:

- In a regular linked list, when the list is empty, the head pointer points to NULL, indicating that there are no elements in the list.
- In a Header Linked List, even when the list is empty, the header node still exists. The next pointer of the header node points to NULL, indicating the end of the list.

Insertion and Deletion:

- Header Linked Lists support the same insertion and deletion operations as regular linked lists.
- When inserting a new node, the new node is linked to the existing nodes by updating the next pointers of the appropriate nodes, similar to regular linked lists.
- When deleting a node, the next pointer of the previous node is updated to bypass the node to be deleted, and the memory of the deleted node is freed.

Header Linked List (Contd….)

Traversal and Search:

- Traversing a Header Linked List involves starting from the header node and following the next pointers to visit each data node in the list.
- Searching for a specific element is done similarly by iterating through the list starting from the header node and comparing the data in each node.

Memory Overhead:

- The introduction of the header node adds a small memory overhead to the linked list as it requires additional memory allocation for the header node.
- The overhead is typically insignificant compared to the benefits it provides, especially in cases where an empty list needs to be handled differently.

Summary

- Queues are data structures that support basic operations like insertion (enqueue), deletion (dequeue), peeking at the front element, checking if the queue is empty, and determining if it is full.
- They can be represented using arrays, dynamic memory allocation, or linked lists and are used in various scenarios, including Round Robin Algorithm for CPU scheduling, Circular Queues for event-driven systems, DeQueue for task scheduling, and Priority Queues for task prioritization.
- Linked lists are dynamic data structures consisting of nodes that contain data and pointers. In a single linked list, nodes are linked in a unidirectional chain, with each node pointing to the next node.

Summary (Contd….)

- Operations on a single linked list include insertion, deletion, traversal, and reversal. Single linked lists offer dynamic size, efficient insertion and deletion, and memory utilization advantages.
- Circular linked lists form a closed loop with the last node pointing back to the head, while double linked lists have both next and previous pointers for efficient traversal in both directions.

Self Assessment Question

1. A linked list is a linear data structure with contiguous memory allocation for elements to be stored. State whether true or false.

   a. True

   b. False

Answer: b

Self Assessment Question

2. Queue is a ……..Data structure

a. FIFO

b. LIFO

c. Linear

d. None of the above

Answer: a

Self Assessment Question

3. Which Queue data structure allows insertion and deletion at both ends?

a. Circular

b. Priority

c. Dequeue

d. None of the above

Answer: b

Self Assessment Question

4. In priority queue two elements with same priority is processed……in order.

a. LIFO

b. FIFO

c. Same order

d. None of the above

Answer: b

Self Assessment Question

5. Which of these is an application of linked list?

a. To implement file system

b. Chaining

c. Implementing non-binary trees

d. All the above

Answer: d

Self Assessment Question

6. Which of the following is not a disadvantage to the usage of array?

a. Fix a size

b. Insertion based on position

c. Random access

d. Static allocation

Answer: c

Assignment

1. Compare the array-based and linked list-based representations of a queue in terms of memory utilization and time complexity for insertion and deletion.
2. Describe the basic operations of a queue and explain their significance in various real-world applications
3. Define linked lists and discuss their advantages over arrays for dynamic data storage. Provide examples of scenarios where linked lists are preferred.
4. Implement a singly linked list in Java and perform operations like insertion, deletion, and traversal.

Assignment (Contd….)

5. Compare and contrast single linked lists, double linked lists, and circular linked lists. Discuss the specific advantages and use cases of each type.
6. Explain the concept of a header linked list and illustrate how it simplifies operations on an empty list.

Document Link

Topic	URL	Notes
Linked list	https://www.geeksforgeeks.org/linked-list-set-1-introduction/ https://www.tutorialspoint.com/data_structures_algorithms/linked_list_algorithms.htm	Include notes on linked list
Sparse matrix	https://www.geeksforgeeks.org/sparse-matrix-representation/	Include note on sparse matrix implementation using linked list
Polynomials manipulation	https://www.geeksforgeeks.org/adding-two-polynomials-using-linked-list/	Include notes on polynomials manipulation using linked list

Video Link

Topic	URL	Notes
Queue and its basic operations	https://www.youtube.com/watch?v=XuCbpw6Bj1U	This link contains the queue and its basic operations
Types of queue data structure	4.1 Queue in Data Structure | Introduction to Queue | Data Structures Tutorials - YouTube	This link contains the Queue data Structure
Types of queue data structure	https://www.youtube.com/watch?v=4xLh68qokxQ	This link contains the Dequeue
Applications of Queue	Queue as ADT | Enqueue & Dequeue | Data Structures | Lec-13 | Bhanu Priya - YouTube	This link contains features, Queue ADT and applications of Queue

Video Link (Contd….)

Topic	URL	Notes
Linked Lists	https://www.youtube.com/watch?v=pBrz9HmjFOs	This video explains about linked list

E-Document Link

Topic	URL	Notes
Types of Queue	https://www.mlsu.ac.in/econtents/326_05Queues.pdf	This link contains the types of Queue
Operations of Queue	https://www2.seas.gwu.edu/~mtdiab/Courses/CS1112/slides/QUEUES-ADT.pdf	This link contains the features of queue and operations of queue
Applications of Queue	https://courses.cs.washington.edu/courses/cse143/02au/slides/18b-SAndQ-applications.pdf	This link contains applications of Queue