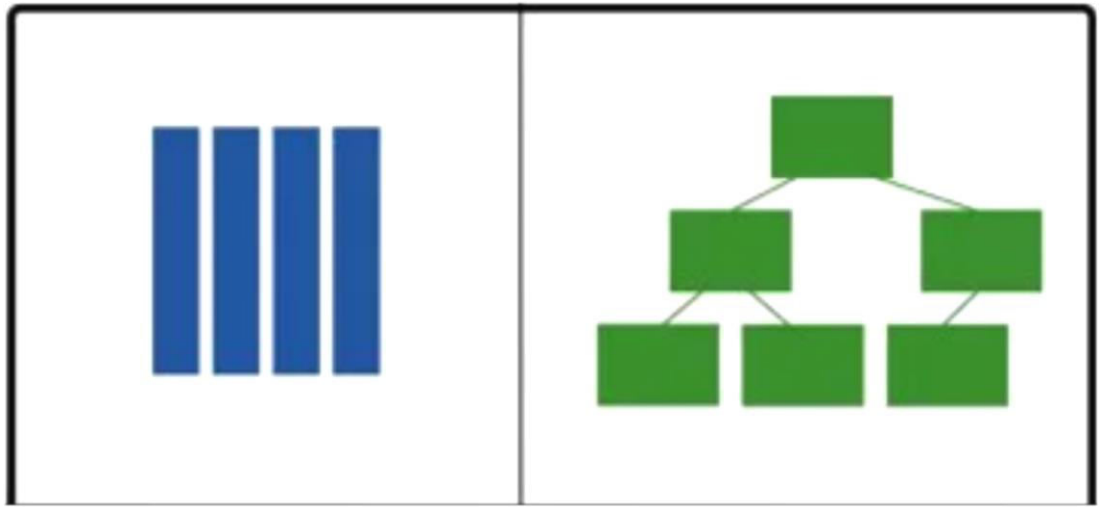
DATA STRUCTURES AND ITS APPLICATIONS

Introduction to Data Structures

Vandana M L
Department of Computer Science and Engineering

Introduction to Data Structures

Data Structure is a scheme of organizing data in the memory of the computer in such a way that various operations can be performed efficiently on this data



PES UNIVERSITY ONLINE

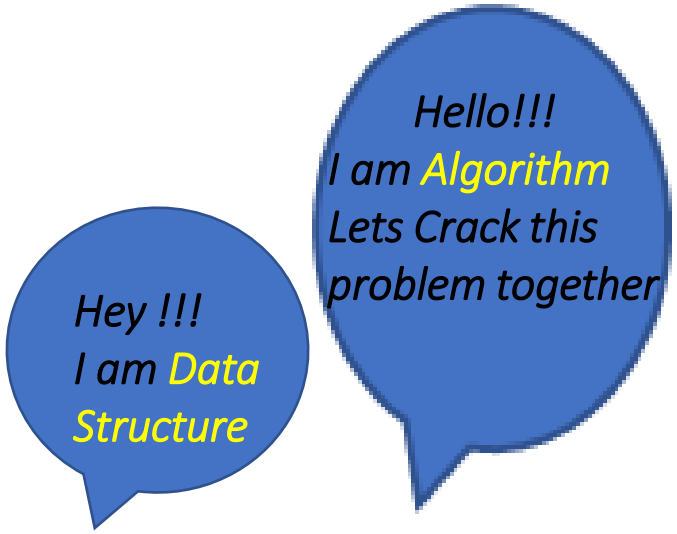
Why Data Structure?

DATA STRUCTURES AND ITS APPLICATIONS

Introduction to Data Structures

Why Data Structures?

- Computer systems deal with large amount of data (text ,image, relational data etc.)
- Data is just the raw material for information, analytics, business intelligence, advertising, etc.
- The way data is organized in memory plays a key role in deciding the time complexity of the algorithms designed for solving the problems
- Data Structures and algorithm go hand in hand



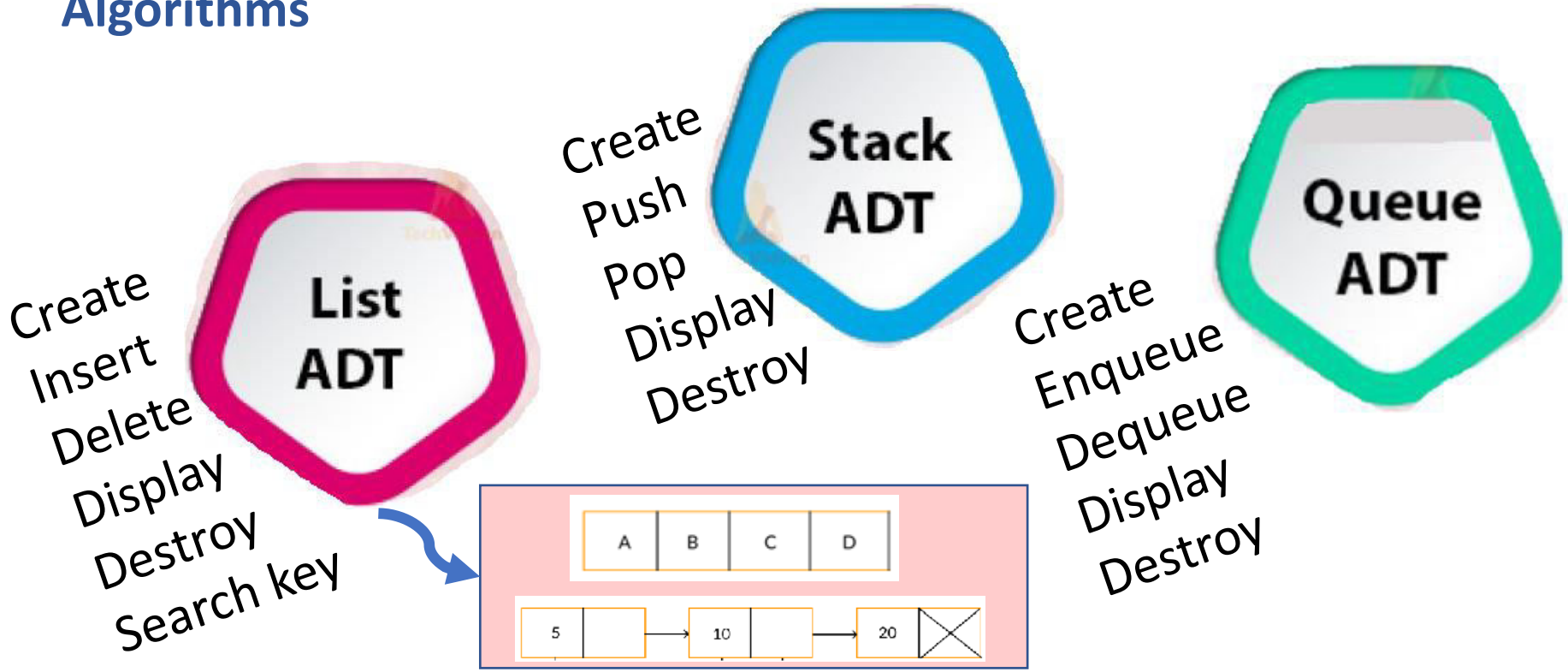
Importance of Data Structures

- Data Structures is most fundamental and building block concept in computer science

- Good knowledge of Data Structures is required to build efficient software systems

Abstract Data Type

- Abstract Data Type is used to represent data and operations associated with an entity from the point of view of user irrespective of implementation
- ADT can be implemented using one or more **Data Structures** and **Algorithms**



DATA STRUCTURES AND ITS APPLICATIONS

Classification of Data Structures

- **Linear Data Structures**

  Stack, Queue, Linked List

- **Non Linear Data Structures**

  Tree , Graph

DATA STRUCTURES AND ITS APPLICATIONS

Classification of Data Structures : Linear Data Structures

Linear Organisation

Stack
Queue
Linked List
Linear List using Array

DATA STRUCTURES AND ITS APPLICATIONS

Classification of Data Structures : Non Linear Data Structures

Non Linear Organisation

Tree

Graph

DATA STRUCTURES AND ITS APPLICATIONS

Few Applications of Linear Data Structures

- **Array**
  - To implement other data structures
  - To store files in memory
- **Linked Lists**
  - To implement other data structures
  - To manipulate large numbers
- **Stacks**
  - Recursion
  - Infix to postfix conversion
- **Queues**
  - Process Scheduling
  - Event handling

DATA STRUCTURES AND ITS APPLICATIONS

Few Applications of Non Linear Data Structures

- **Tree**
  - Auto complete features (Trie)
  - Used by operating systems to maintain the structure of a file system
- **Heaps**
  - Priority Queue implementation
  - Heap Sort
- **Graphs**
  - Computer Networks
  - Shortest Path Problems

Data Structures and Applications

Overview- Course Contents

Unit -1 : Linked Lists

- Memory Allocation Static and Dynamic
- Singly Linked List
- Doubly Linked Lists
- Circularly Linked Lists
- Multi Lists : Sparse Matrix
- Applications :
  - Skip List

Data Structures and Applications

Overview- Course Contents

Unit -2 : Stacks

- Basic Structure of Stack
- Array and Linked Implementation
- Applications :
  - Recursion
  - Conversion of Infix to Postfix
  - Conversion of Infix to Prefix
  - Evaluation of Expression
  - Parentheses Matching

Data Structures and its Applications

Overview

Unit -2 : Queues

- Basic Structure
- Circular Queue, Priority Queue, Dequeue
- Array and Linked Implementation
- Applications :
  - Josephus Problem,
  - CPU Scheduling

Data Structures and Applications

Overview- Course Contents

Unit -3 : Trees

- Definitions, Binary Trees, Binary Search Tree, Threaded Binary trees.
- Operations on Trees
- Implementation of BST,
- Threaded BST

Unit -3 : Heaps

- Heap as a Data Structure
- Array Implementation
- Priority queue as a heap
- Applications : Dictionary Implementation

Data Structures and Applications

Overview- Course Contents

Unit -4 : Balanced Trees and Graphs

- AVL Trees
- Operations on AVL Trees
- Properties of Graphs
- Implementation of Graphs
- Search Operations on Graph
- Applications :
  - Indexing in data bases
  - Representing a Computer Topology

Data Structures and its Applications

Overview- Course Contents

Unit -5 : Suffix Trees

- Tries
- Implementation of Tries
- Operations on Tries : Insert, delete and search
- Applications :
  - Word Prediction
  - URLs Decoding
  - Cryptography

Unit -5 : Hashing

- Hashing Techniques
- Collision resolution
- Double Hashing, Rehashing

Data Structures and its Applications

Overview- Course Contents

Text Book :

Data Structures using C & C++

Yedidyah Langsam, Moshe J. Augenstein, Aaron M. Tenenbaum, 2015, Pearson Education , 2nd Edition.

Reference Book:

Data Structure and Program Design in C

Robert Kruse, C.L Tondo, Bruce P. Leung – 2007, Pearson Education ,2nd Edition.

DATA STRUCTURES AND ITS APPLICATIONS

Memory Allocation

Vandana M L
Department of Computer Science and Engineering

Memory Allocation

- Static Memory Allocation
- Dynamic Memory Allocation

Static Memory Allocation

- allocated by the compiler.
- Exact size and type of memory must be known at compile time
- Memory is allocated in stack area

```
int b;

int c[10] ;
```

Disadvantages of Static Memory Allocation

- Memory allocated can not be altered during run time as it is allocated during compile time
- This may lead to under utilization or over utilization of memory
- Memory can not be deleted explicitly only contents can be overwritten
- Useful only when data size is fixed and known before processing

Dynamic Memory Allocation

- Dynamic memory allocation is used to obtain and release memory during program execution.
- It operates at a low-level
- Memory Management functions are used for allocating and deallocating memory during execution of program
- These functions are defined in “stdlib.h”

Dynamic Memory Allocation Functions:

- Allocate memory - malloc(), calloc(), and realloc()
- Free memory - free()

Dynamic Memory Allocation Functions: malloc()

To allocate memory use

```
void *malloc(size_t size);
```

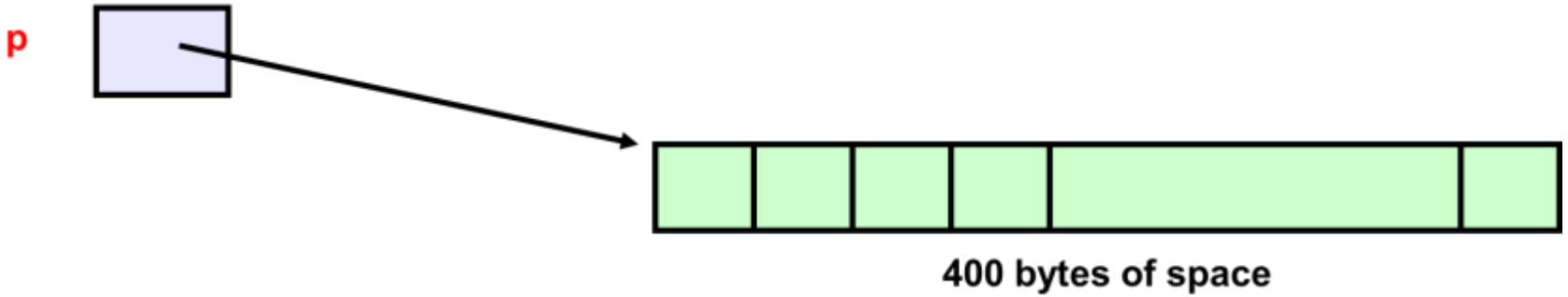
- Takes number of bytes to allocate as argument.
- Use sizeof to determine the size of a type.
- Returns pointer of type void *. A void pointer may be assigned to any pointer.
- If no memory available, returns NULL.

Dynamic Memory Allocation Functions: malloc()

To allocate space for 100 integers:

```
int *p;
if ((p = (int*)malloc(100 * sizeof(int))) == NULL){
   printf("out of memory\n");
   exit();
   }
```

- Note we cast the return value to int*.
- Note we also check if the function returns NULL.



Dynamic Memory Allocation Functions: malloc()

- cptr = (char *) malloc (20);

  Allocates 20 bytes of space for the pointer cptr of type char

- sptr = (struct stud *) malloc(10*sizeof(struct stud));

  Allocates space for a structure array of 10 elements. sptr points to a structure element of type struct stud

Always use sizeof operator to find number of bytes for a data type, as it can vary from machine to machine

Dynamic Memory Allocation Functions: malloc()

- malloc always allocates a block of contiguous bytes
  - The allocation can fail if sufficient contiguous memory space is not available
  - If it fails, malloc returns NULL

```
if  ((p = (int *) malloc(100 * sizeof(int))) == NULL)
 {
      printf ("\n Memory cannot be allocated");
      exit();
 }
```

The n integers allocated can be accessed as *p, *(p+1), *(p+2),…, *(p+n-1) or just as p[0], p[1], p[2], …,p[n-1]

Dynamic Memory Allocation Functions: free()

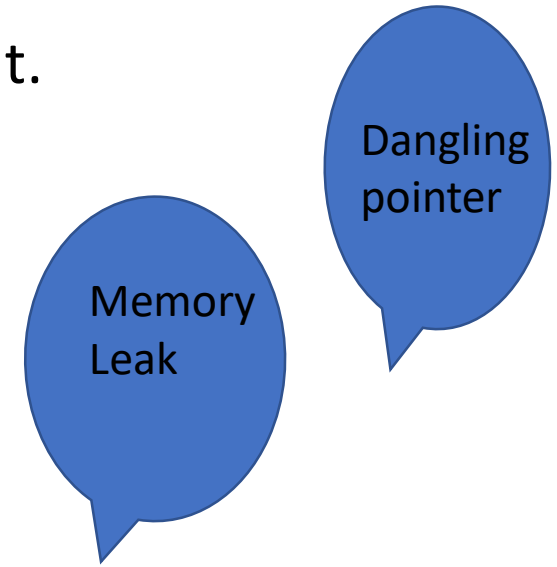
To release allocated memory use

```
free(ptrvariable)
```

- Deallocates memory allocated by malloc().
- Takes a pointer as an argument.

e.g.

```
free(newPtr);
```



Dynamic Memory Allocation Functions: calloc()

Similar to malloc(),

But allocated memory space are zero by default..

calloc() requires two arguments –

```
void *calloc(size_t nitem, size_t size);
```

Example

```
int *p;
p=(int*)calloc(100,sizeof(int));
```

returns a void pointer if the memory allocation is successful,
else it'll return a NULL pointer.

Dynamic Memory Allocation Functions: realloc()

Reallocate a block

Two arguments

- Pointer to the already allocated block
- Size of new block

```
int *ip;
ip = (int*)malloc(100 * sizeof(int));
. . .
/*  need twice as much space */
ip = (int*)realloc(ip, 200 * sizeof(int));
```

DATA STRUCTURES AND ITS APPLICATIONS

Lecture Summary

Memory Allocation

- Static Memory allocation
- Dynamic memory allocation

Apply the concepts to implement C program for the following problem statement

- Multiply two matrices . Allocate the memory for the matrices dynamically

DATA STRUCTURES AND ITS APPLICATIONS

Introduction to Singly Linked List

Vandana M L
Department of Computer Science and Engineering

List Data Structure

List

- Dynamic data structure consists of a collection of elements
- Can be implemented in two ways
  - By contiguous memory allocation : ArrayList
  - By Linked Allocation : Linked List

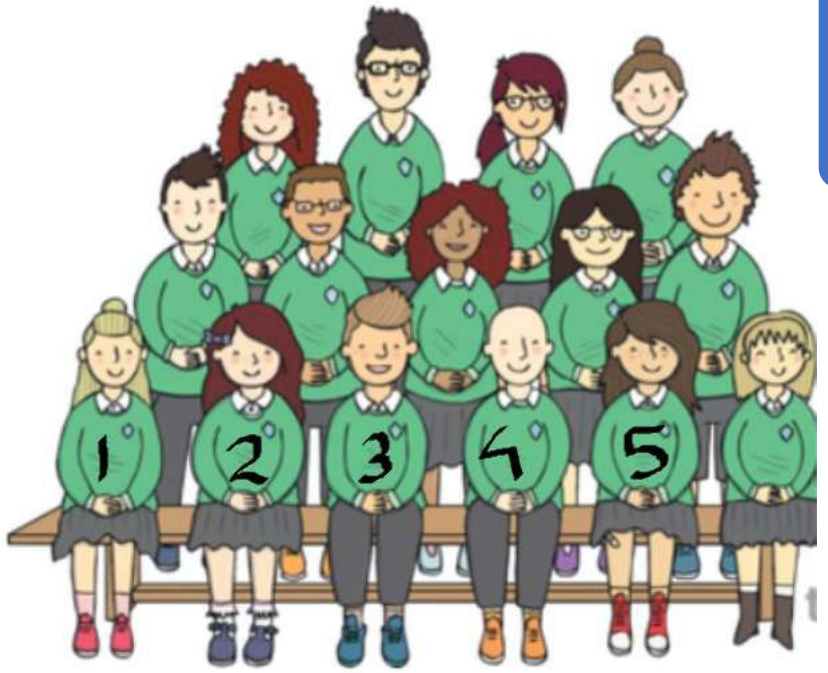
DATA STRUCTURES AND ITS APPLICATIONS

List Data Structure: Operations

- Creating a List
- Inserting an element in a list
- Deleting an element from a list
- Searching a list
- Reversing a list
- Concatenating two lists
- Traversing a list

DATA STRUCTURES AND ITS APPLICATIONS

Understanding Array List (Linear List using Arrays)



Placement

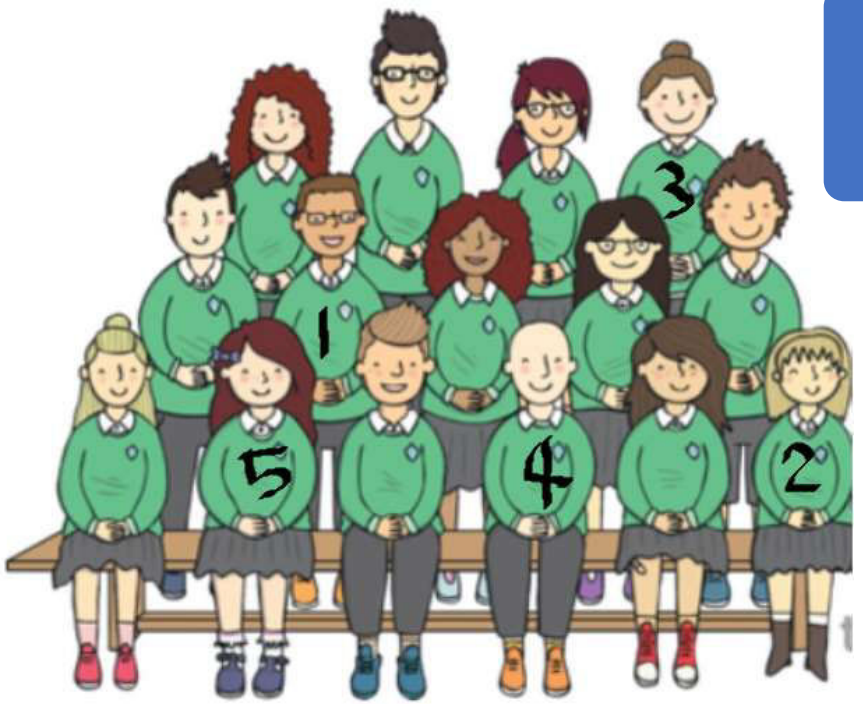
- Sequential

Access

- Sequential

Array List

Understanding Linked List



Placement

- Random

Access

- Sequential

Linked List

DATA STRUCTURES AND ITS APPLICATIONS

Array List Vs Linked List

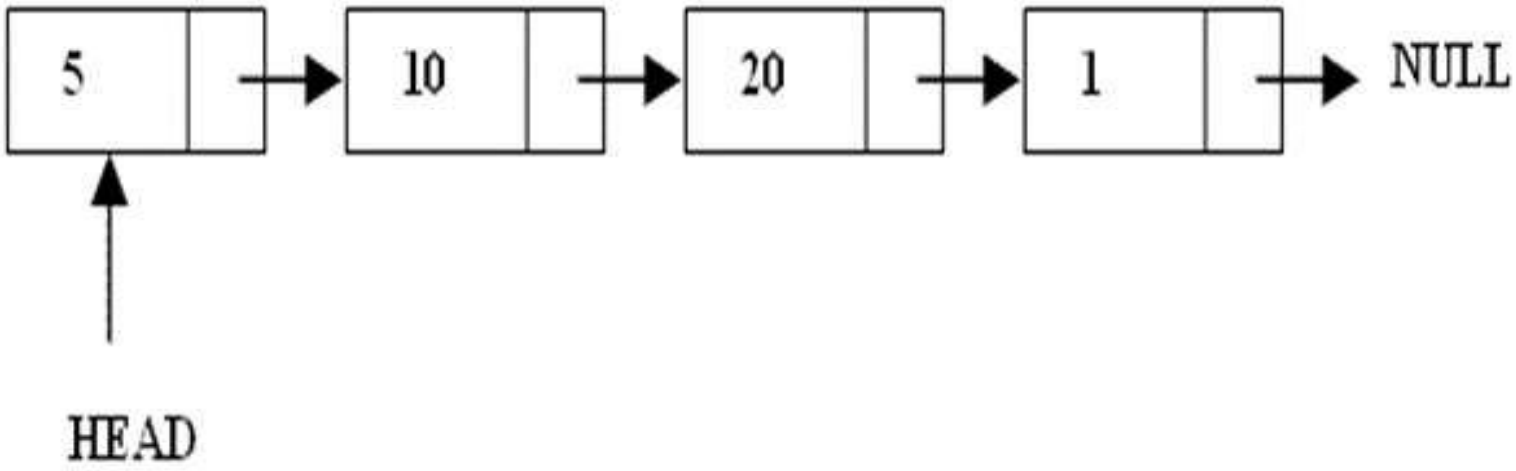
ArrayList	Linked list
Fixed size: Resizing is expensive	Dynamic size
Insertions and Deletions are inefficient	Insertions and Deletions are efficient
Elements in contiguous memory locations	Elements not in contiguous memory locations
May result in memory wasteage if all the allocated space is not used	Since memory is allocated dynamically(as per requirement) there is no wastage of memory.
Sequential and random access is faster	Sequential and random access is slow

Types of Linked List

- Singly Linked List
- Doubly Linked List
- Circular Linked List
- Multi Linked List

Singly Linked List

- A linked list is a linear data structure.
- Nodes make up linked lists.
- Nodes are structures made up of data and a pointer to another node.
- Usually the pointer is called as link.

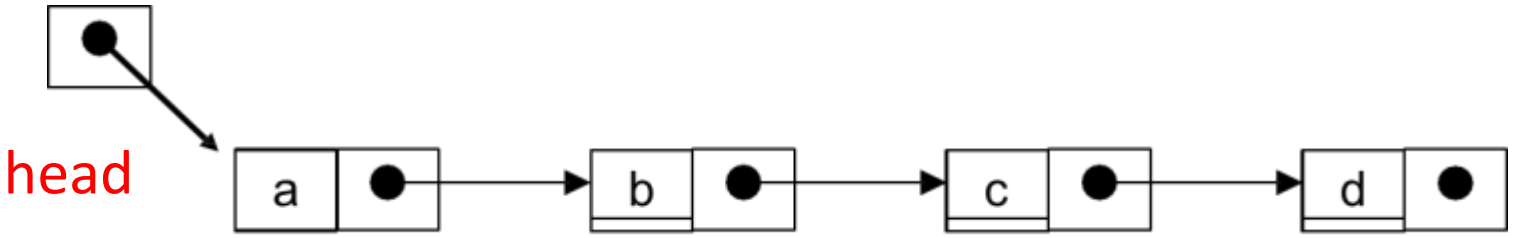


Single Linked List

- 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

DATA STRUCTURES AND ITS APPLICATIONS

Single Linked List :Schematic representation



- Each node contains a value(data) and a pointer to the next node in the list

- Head/start is a pointer which points at the first node in the list

Singly Linked List

Apply the concepts to answer the following questions

- Give structure definition for node of singly linked list used to store employee data (employee no , name, salary ,designation)

DATA STRUCTURES AND ITS APPLICATIONS

Singly Linked List

Vandana M L
Department of Computer Science and Engineering

Singly Linked List: Node Structure

Defining node structure

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

typedef struct node NODE;
```

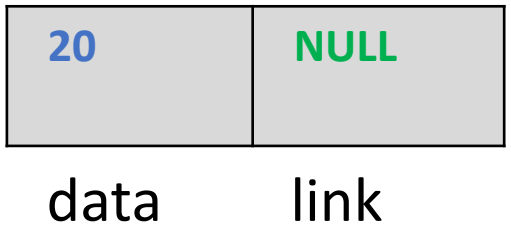
data link

```
struct polydata
{
  int coeff;
  int expo;
};

typedef struct node
 {
polydata data;
struct node *link;
}polynode;
```

coef expo link

Creating a node

- Allocate memory for the node dynamically
- If the memory is allocated successfully
  set the data part to user defined value
  set the link part to NULL



Inserting a node

There are 3 cases

- Insertion at the beginning
- Insertion at the end
- Insertion at a given position

Insertion at the beginning

- Create a node

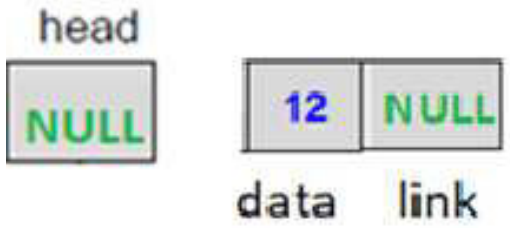
If the list is empty

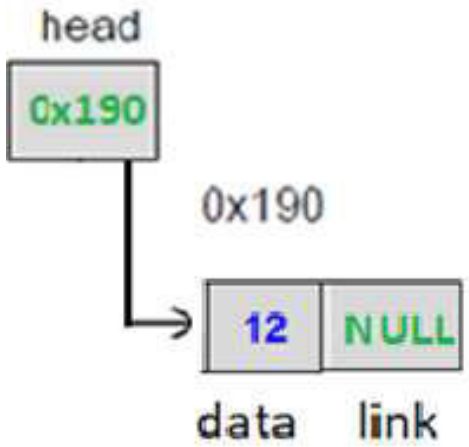
- make the head pointer point towards the new node;

Else

- Make the next pointer of the node point towards the first node of the list
- Make the head pointer point towards this new node

Insertion at the beginning



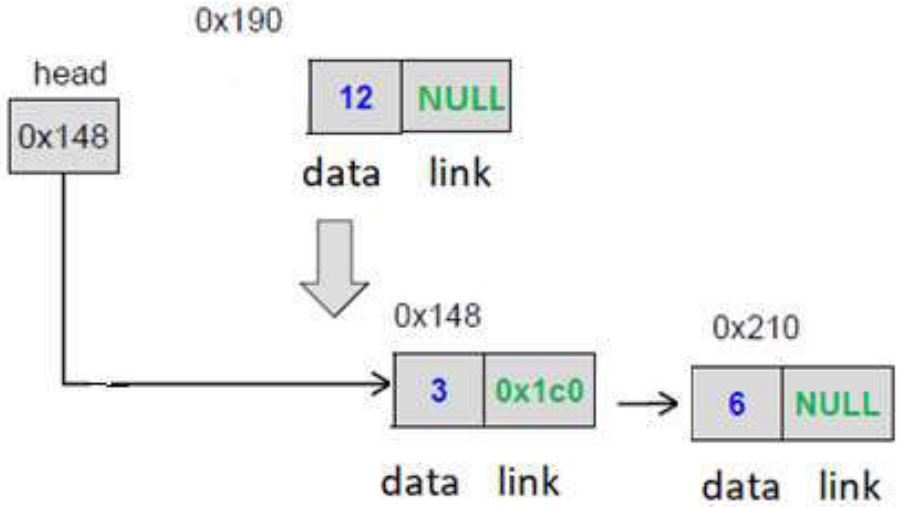


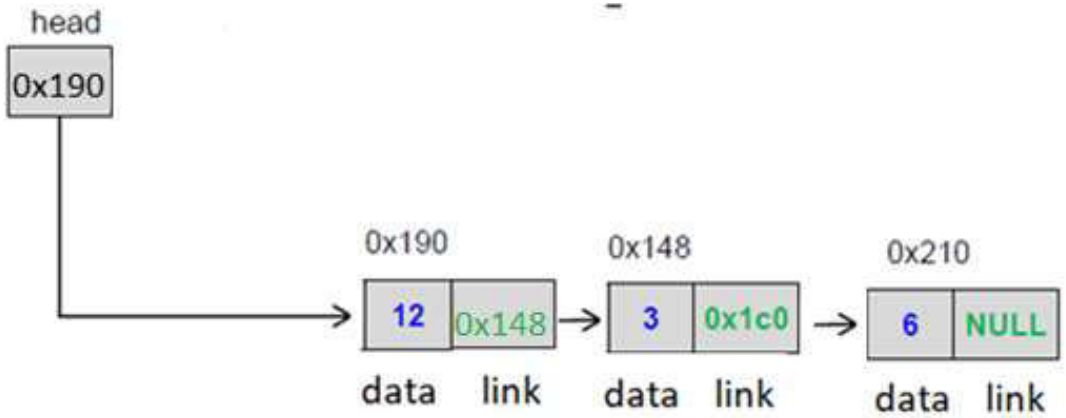
CASE1

DATA STRUCTURES AND ITS APPLICATIONS

Singly Linked List operations

Insertion at the beginning





CASE2

Insertion at the end of the

- Create a node

If the list is empty

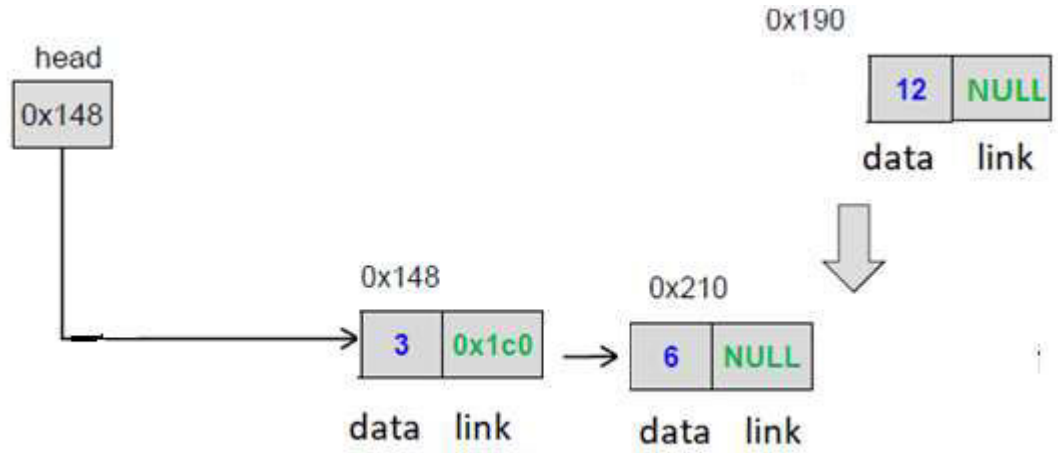
- make the head pointer point towards the new node;

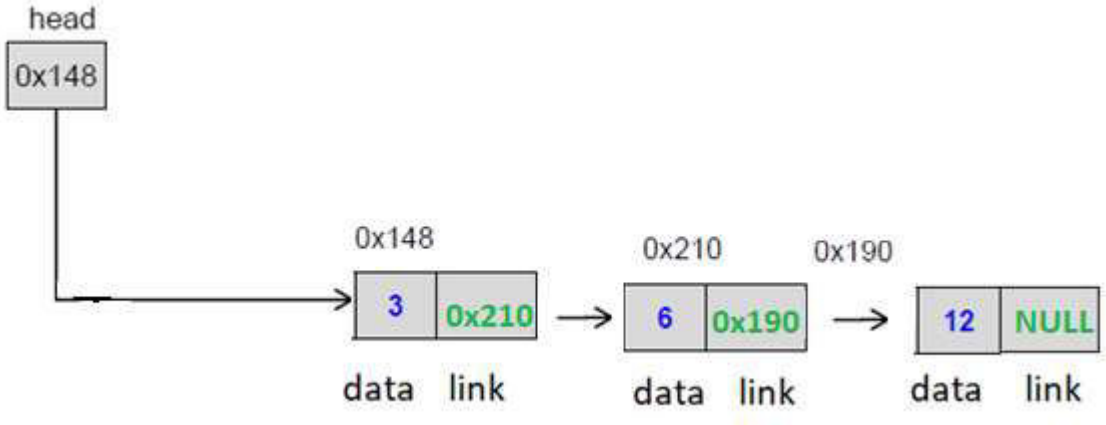
Else

- Traverse the linked list to find out the last node
- Make the link pointer of the last node to point to the new node

Singly Linked List operations

Insertion at the end of the





Singly Linked List operations

Insertion at the given position

- Create a node

If the list is empty or if insertion is to be done at first position

- Same steps as insert front

Else

- Traverse the linked list to reach given position
- Keep track of the previous node

    If it is an intermediate position

    - Change previous node link to point to the newnode
    - Newnode to point to the next node

    Else

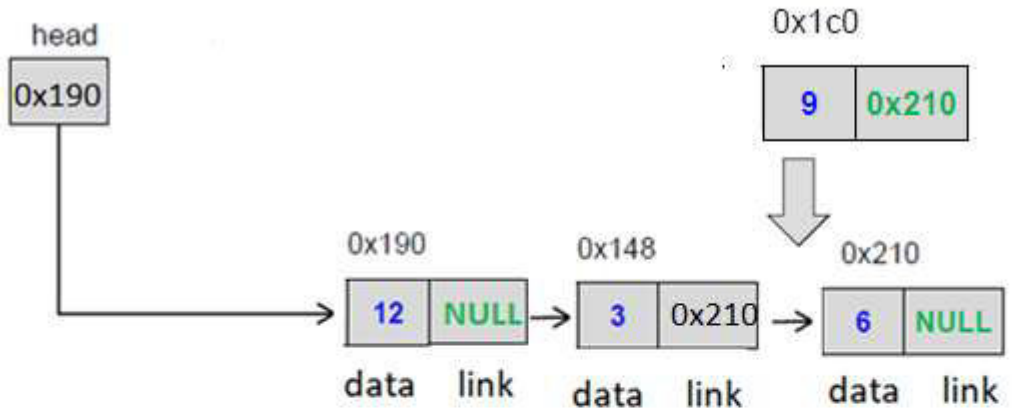
        If it is last position

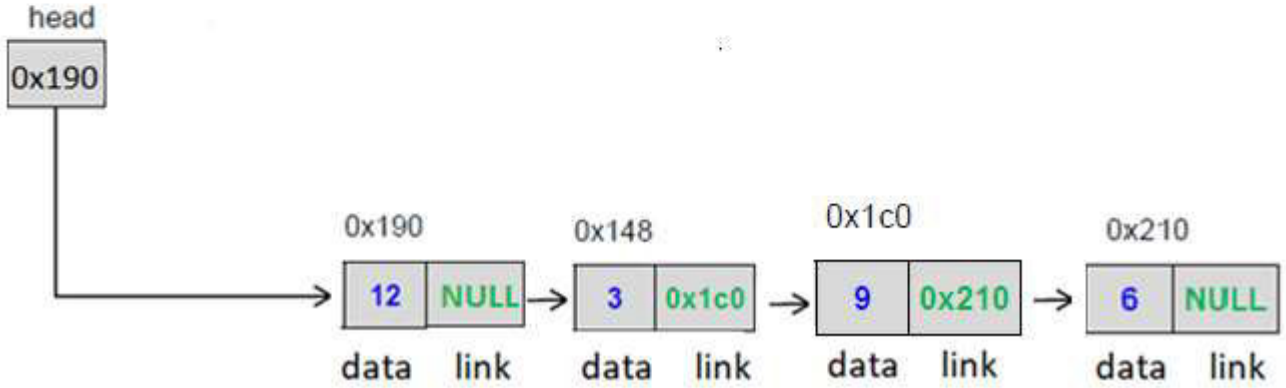
        - Same steps as insert at end

        Else

            Print “Invalid position”

Insertion at the given position





Singly Linked List insert operation

Apply the concepts to implement following operations on a circular singly linked list

- Count number of nodes
- Concatenate two lists
- Sum of all the node values in the list

DATA STRUCTURES AND ITS APPLICATIONS

Singly Linked List

Vandana M L
Department of Computer Science and Engineering

Deleting a node

There are 3 cases

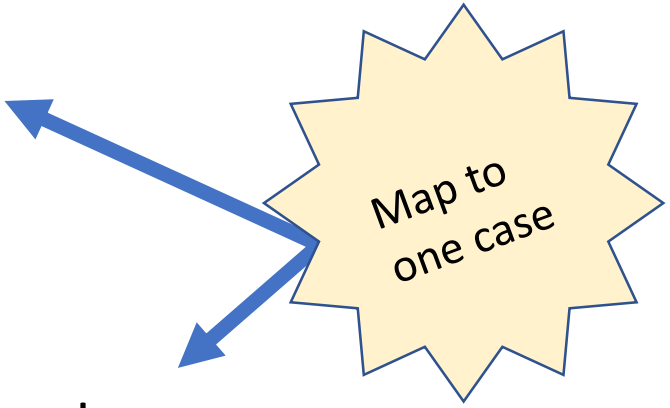
- Deleting first node
- Deleting last node
- Deleting a node at a given position

Deleting first node

Case 1: Linked list is empty

Case 2: Linked list with a single node

- delete the node
- set head to NULL

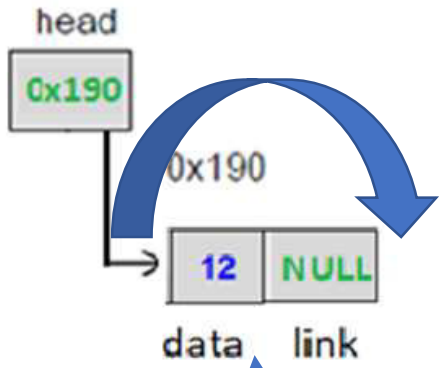


Case3:Linked list has more than one node

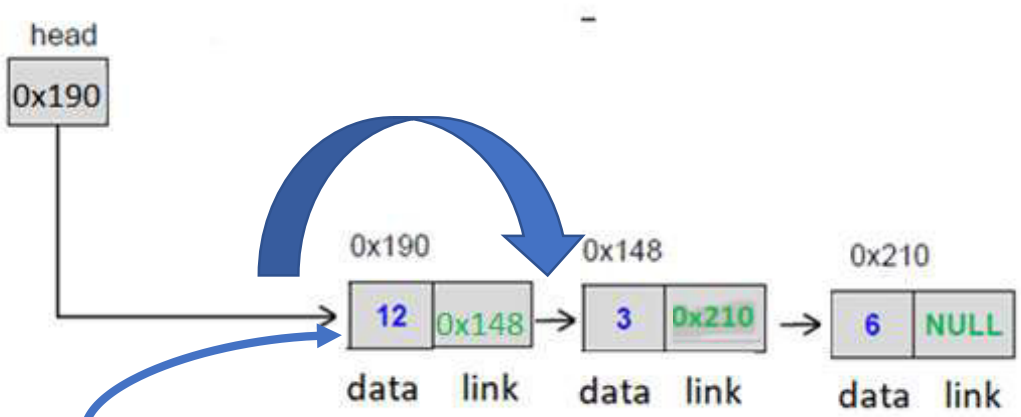
- Change head to point to second node
- Delete the first node

Deleting first node

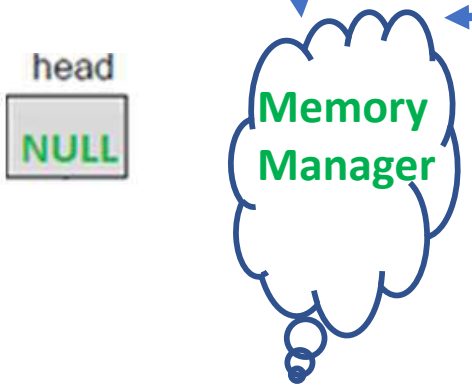
Only one node in list

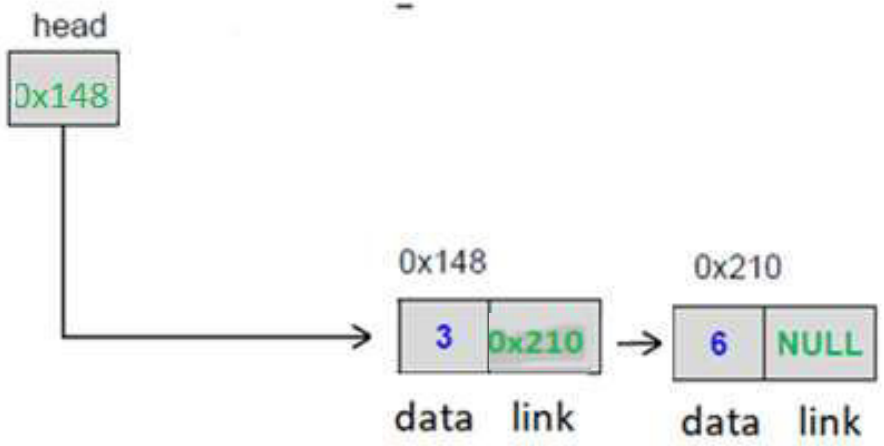


More than one node



Free this





Deleting last node

Case 1: Linked list is empty

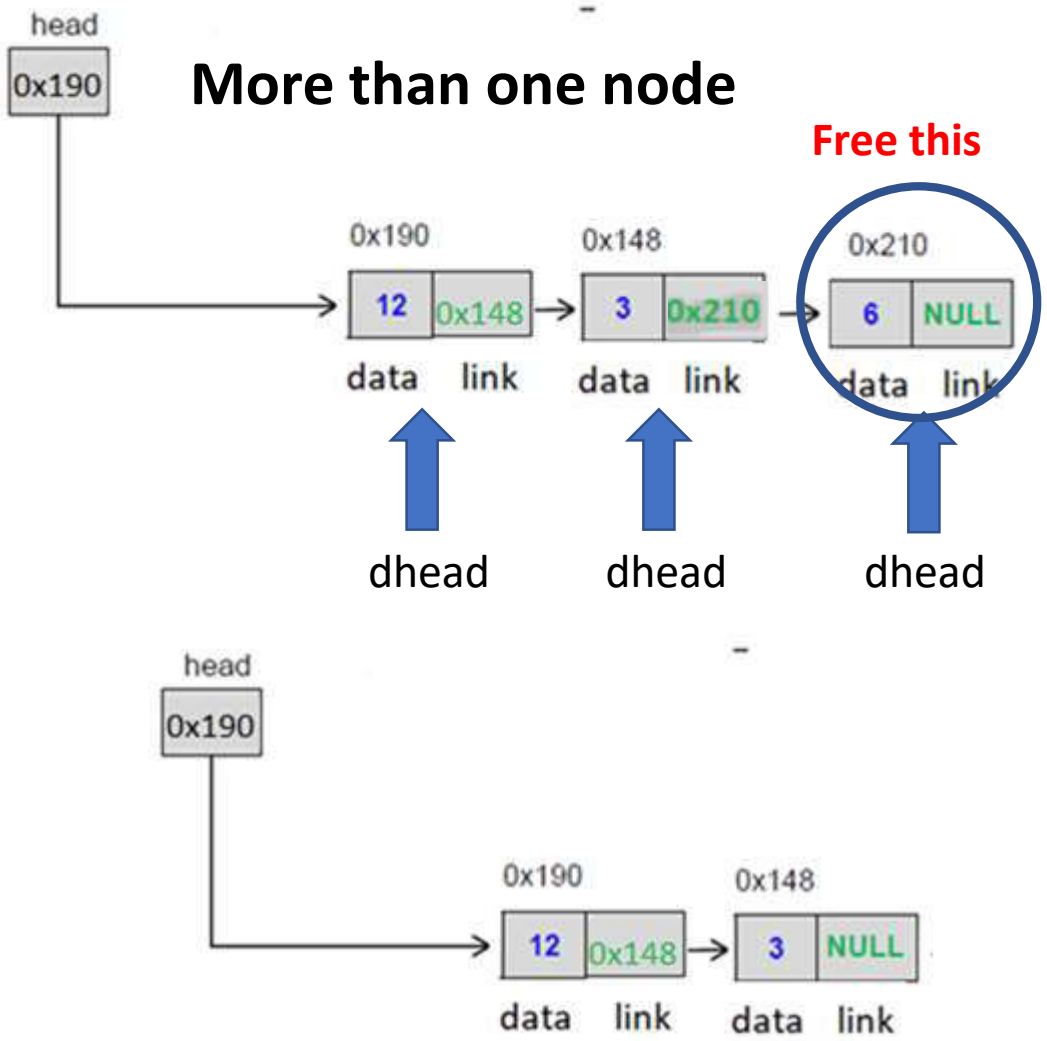
Case 2: Linked list with a single node

- delete the node
- set head to NULL

Case3:Linked list has more than one node

- Traverse the linked list to point to second last node
- Delete the last node
- Set link field of second last node to NULL

Deleting last node



Deleting node from a given position

If the linked list is not empty

If position is 1

- Delete from the front of the linked list

Else

If position is a valid position

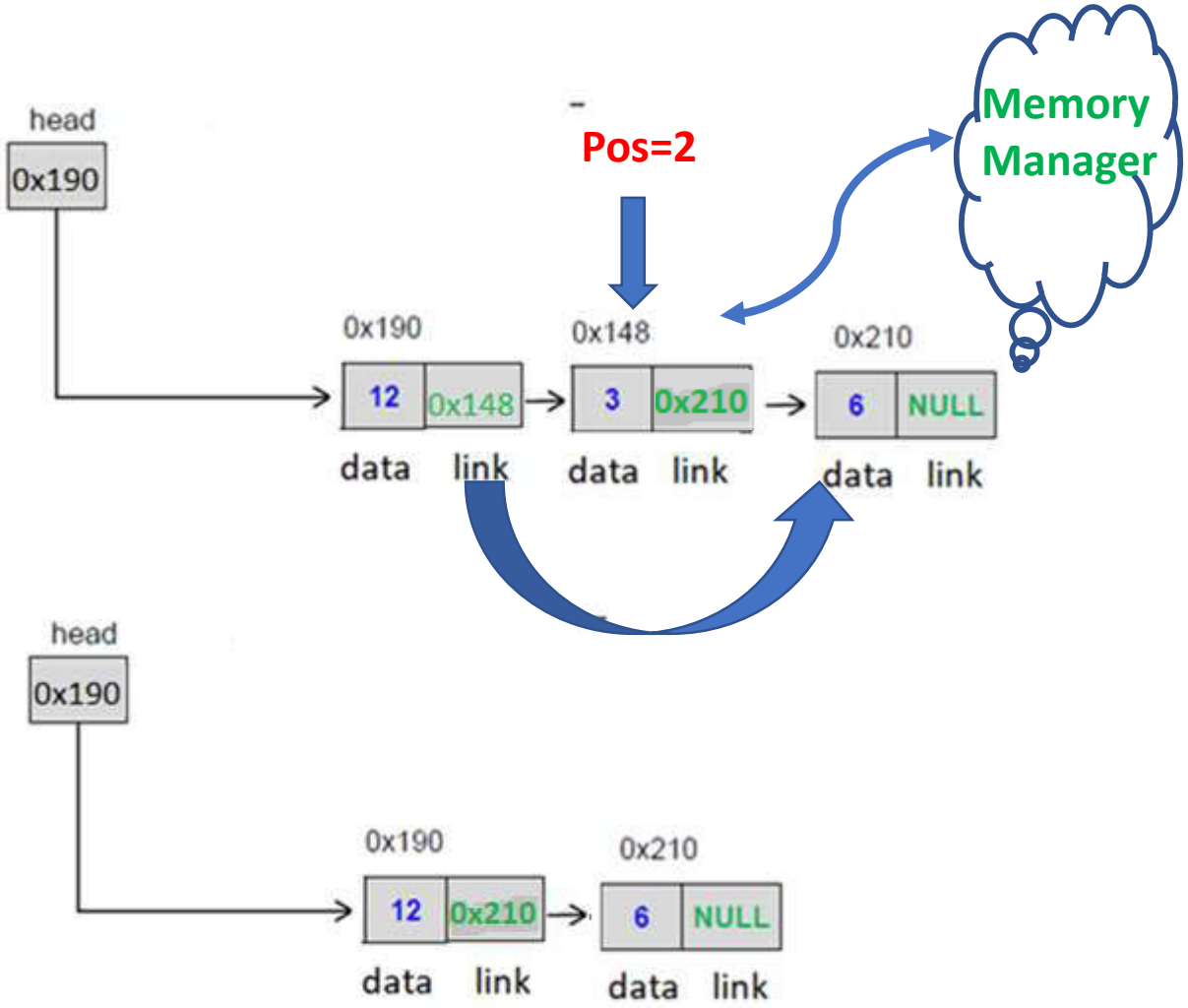
- Traverse linked list to get the desired position
- keep track of previous node
- set previous node link field to link field of current node
- delete the current node

Else

- print invalid position

Singly Linked List Operations

Deleting node from a given position



Singly Linked List delete operation

Apply the concepts to implement following operations for a singly linked list

- Delete a node with given key value
- Delete all alternate nodes
- Delete all the nodes (erase the linked list)

DATA STRUCTURES AND APPLICATIONS

Doubly Linked List

Vandana M L
Department of Computer Science and Engineering

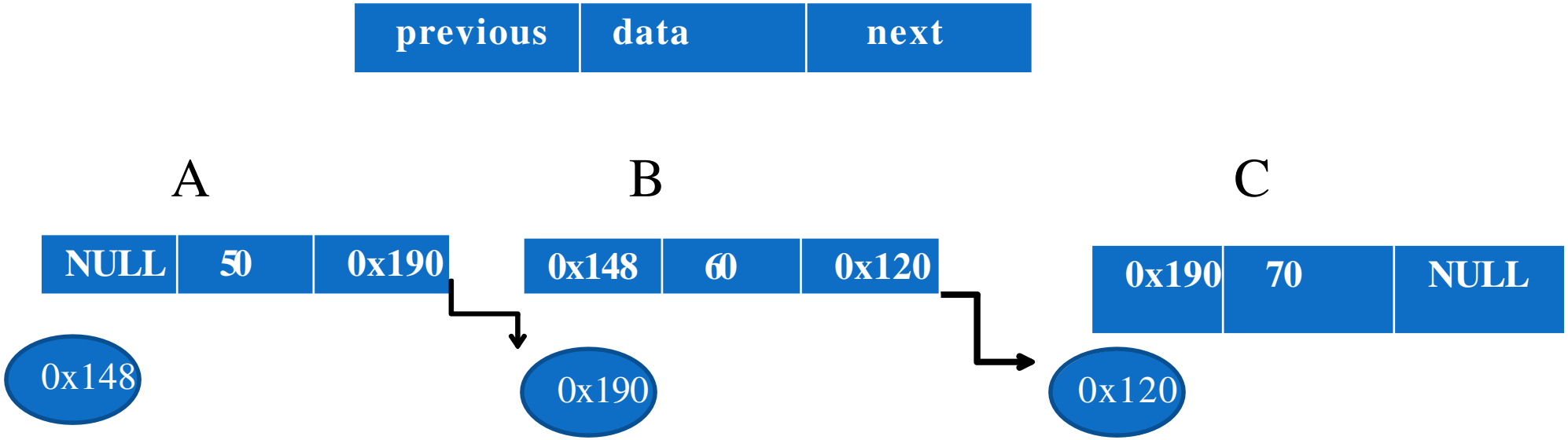
Doubly Linked List

A doubly linked list contain three fields:

- Data
- link to the next node
- link to the previous node.

DATA STRUCTURES AND APPLICATIONS

Doubly Linked List :Node Structure



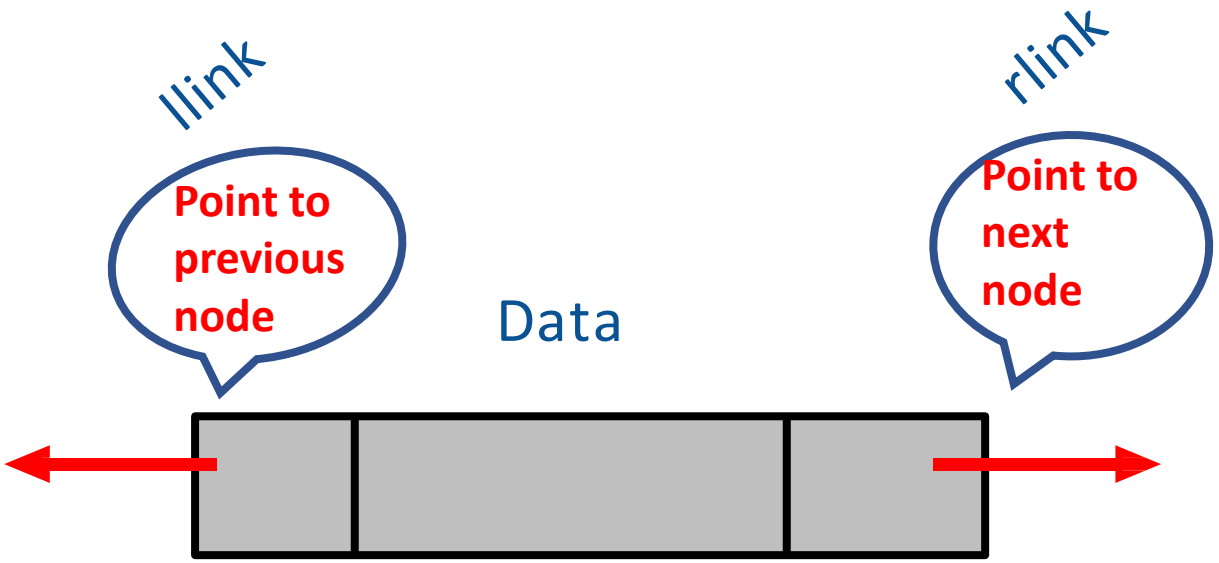
Doubly Linked List Vs Singly Linked List

- Advantages:
  - Can be traversed in either direction (may be essential for some programs)
  - Some operations, such asdeletion and inserting before a node, become easier
- Disadvantages:
  - Requires more space
  - List manipulations are slower (because more links must be changed)
  - Greater chance of having bugs (because more links must be manipulated)

DATA STRUCTURES AND APPLICATIONS

Doubly Linked List Node definition

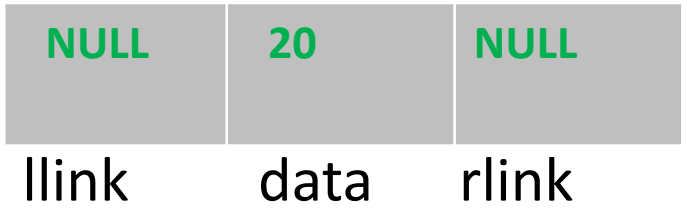
```
struct node
{
  int data;
  node*llink;
  node*rlink;
};
```



Creating a node

- Allocate memory for the node dynamically

- If the memory is allocated successfully
  set the data part to user defined value
  set the llink (address of previous node) and rlink (address of next node) part to NULL



Inserting a node

There are 3 cases

- Insertion at the beginning
- Insertion at the end
- Insertion at a given position

Insertion at the beginning

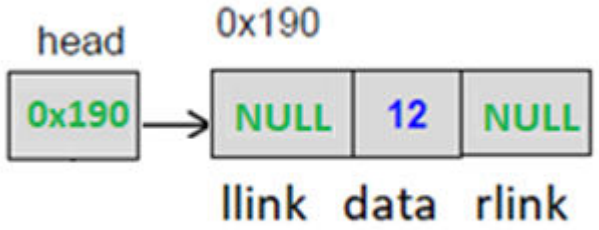
What all will change

If the linked list empty(case 1)
Head/Start pointer
else (case2)

- Head/Start pointer
- New front's llink and rlink
- Old front's llink

Doubly Linked List Implementation

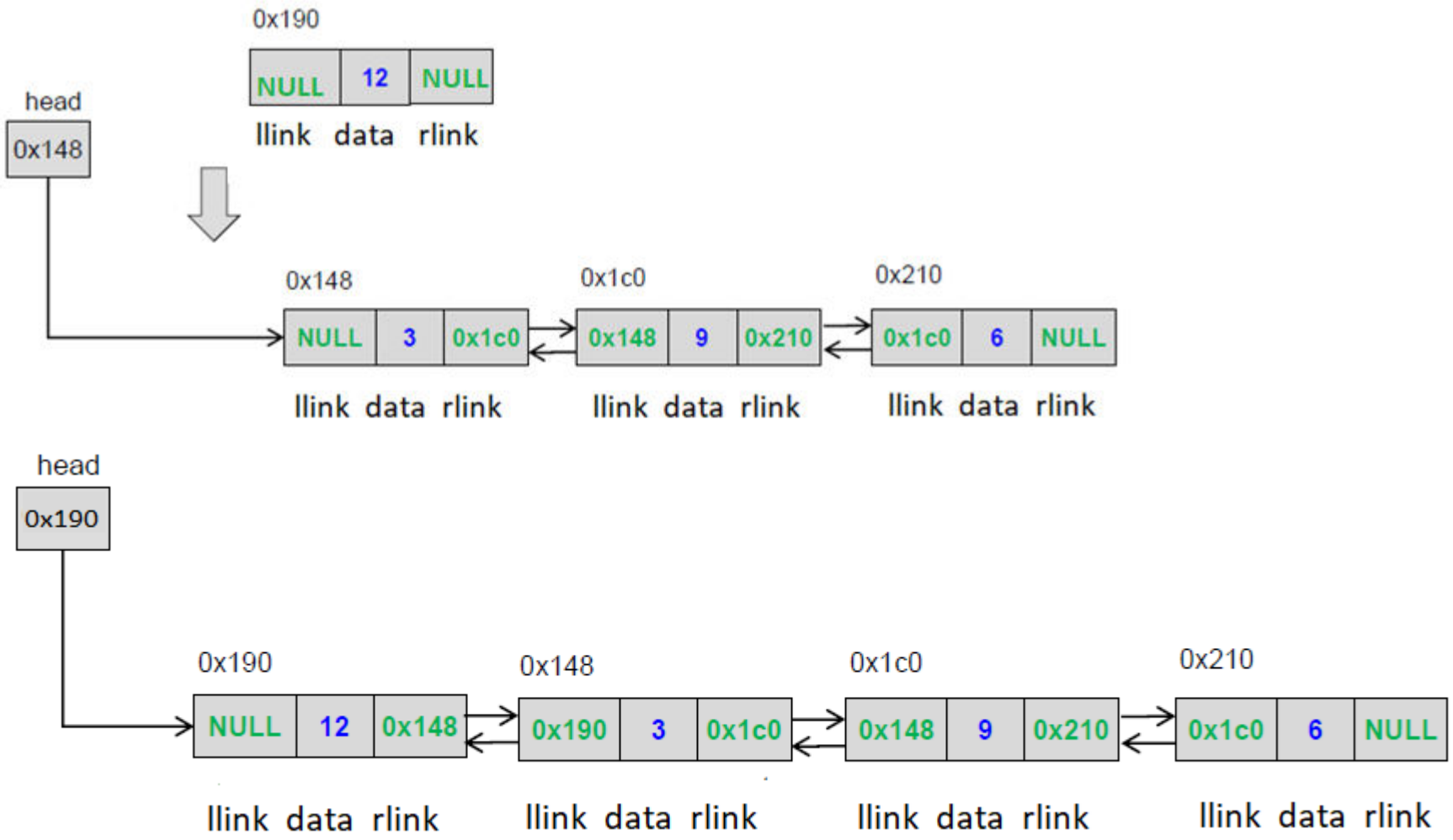
Insertion at the beginning (Case1)



DATA STRUCTURES AND APPLICATIONS

Doubly Linked List Implementation

Insertion at the beginning(Case 2)



Insertion at the end

What all will change

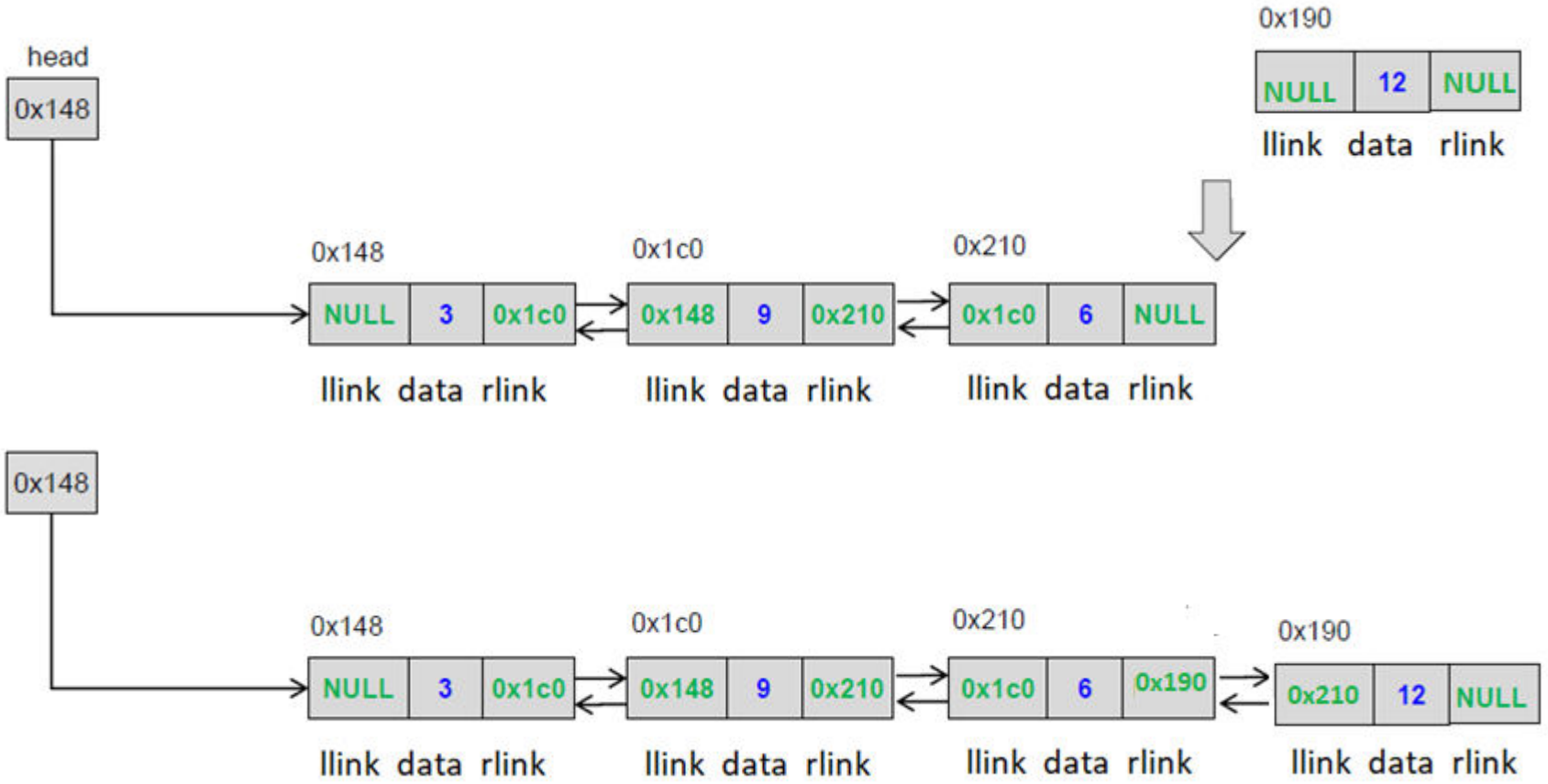
If the linked list empty(same as case 1 of insert at front)
Head/Start pointer(case 2)
else

- Last node's rlink
- New node's llink

DATA STRUCTURES AND APPLICATIONS

Doubly Linked List Implementation

Insertion at the end



Doubly Linked List Implementation

Insertion at the given position

- Create a node

If the list is empty

- make the start pointer point towards the new node;

Else

- Traverse the linked list to reach given position
- Keep track of the previous node

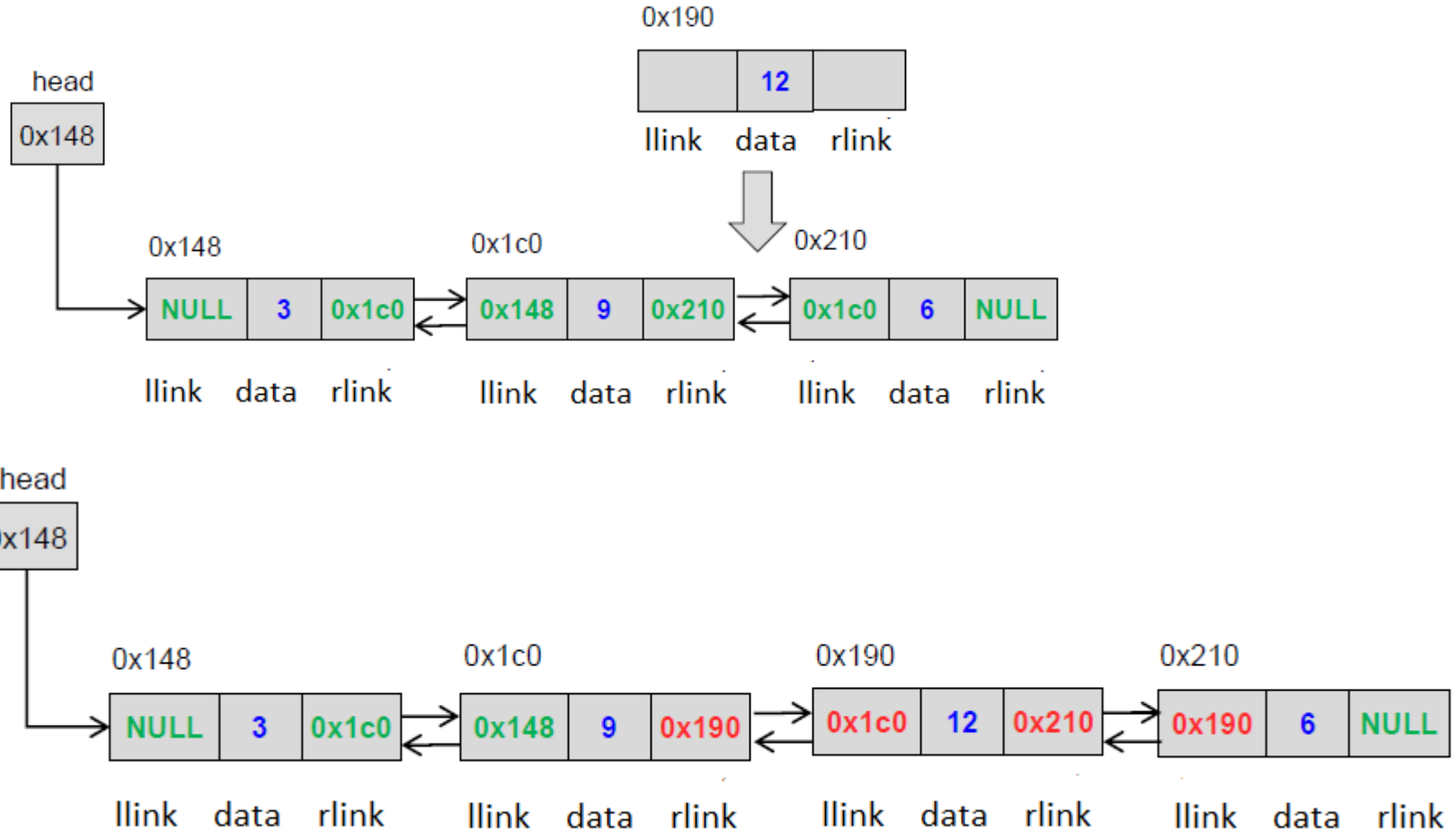
If it is an intermediate position

- Change previous node rlink to point to the newnode
- Newnode's llink to point to previous node and rlink to point to the next node
- Next node llink to point to the newnode

DATA STRUCTURES AND APPLICATIONS

Doubly Linked List Implementation

Insertion at the given position



Deleting a node

There are 3 cases

- Deleting first node
- Deleting last node
- Deleting a node at a given position

Deleting a node

There are 3 cases

- Deleting first node
- Deleting last node
- Deleting a node at a given position

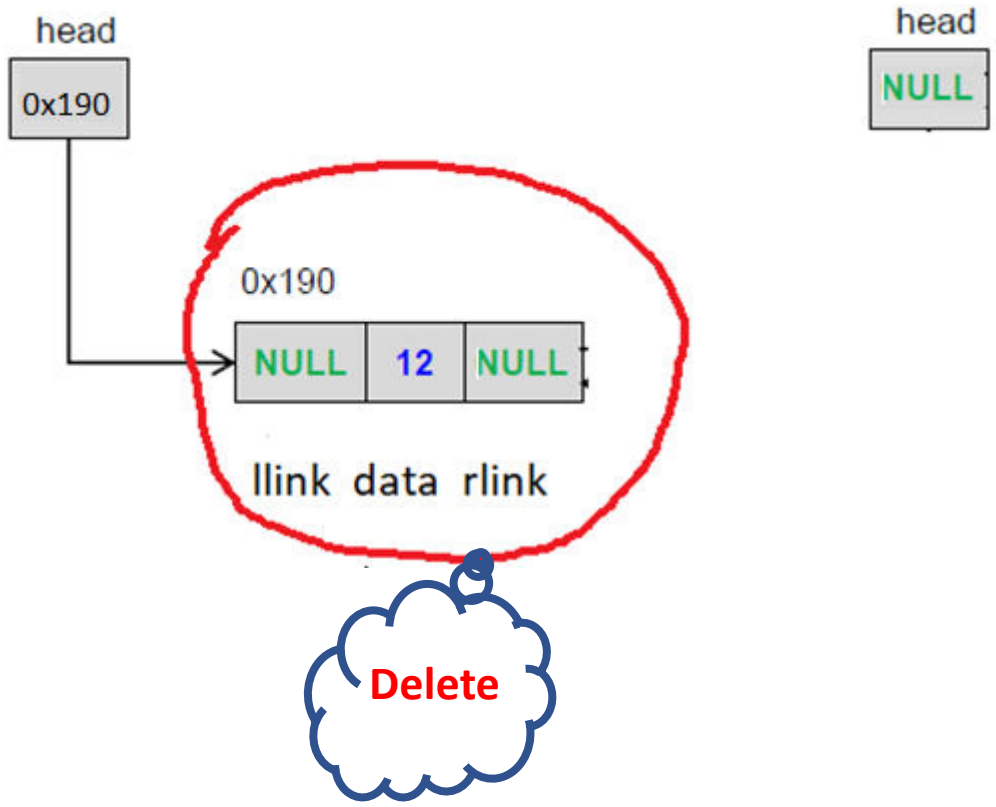
Deleting first node

What will change??

- Case I : Empty Linked List
- Case II : Linked list with a single node
  first node gets freed up
  head points to NULL
- Case III : Linked List with more than one node
  Second node llink gets changed to NULL
  first node gets freed off
  head points to second node

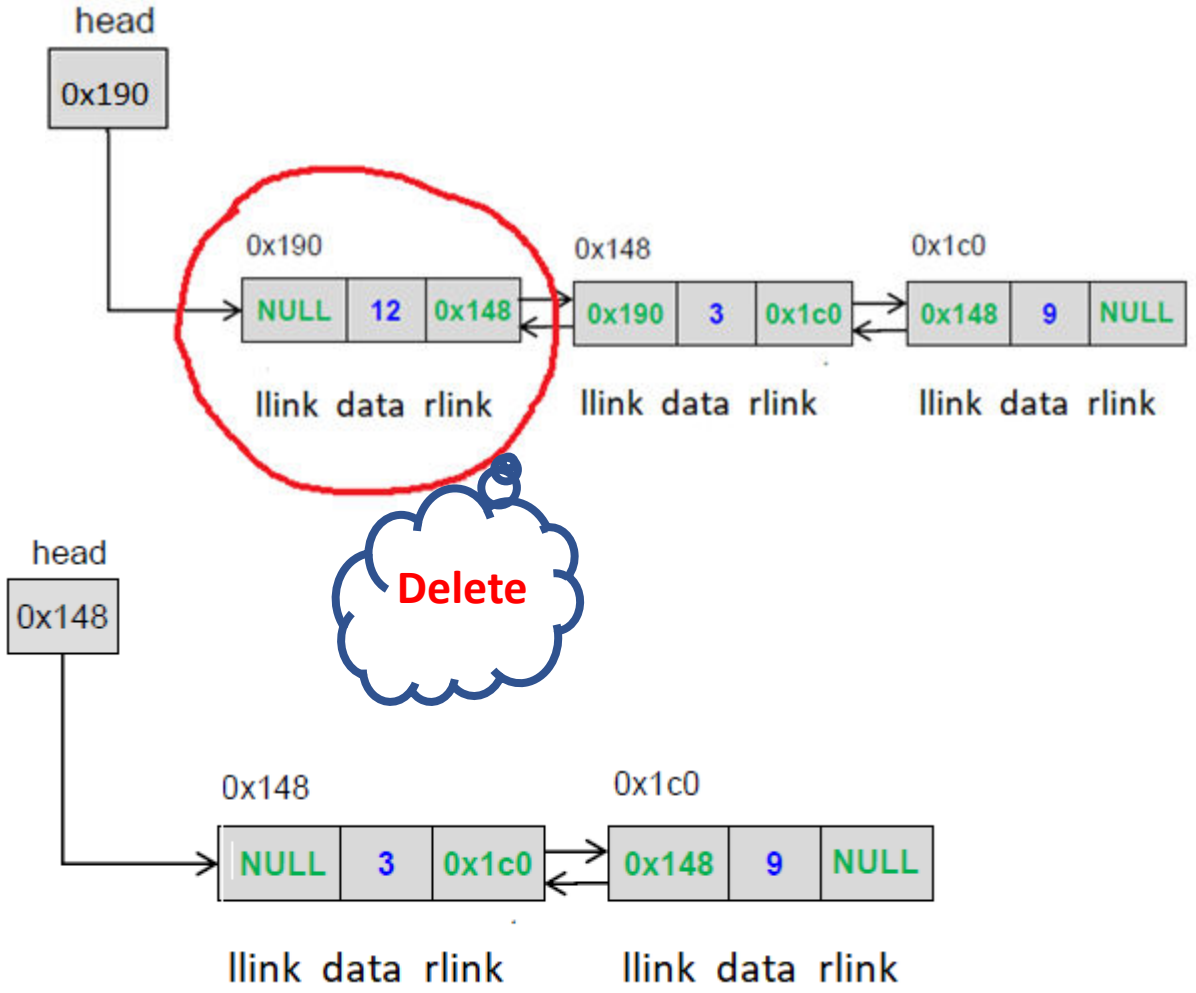
Deleting first node

- Case II : Linked list with a single node



Deleting first node

- Case III : Linked List with more than one node



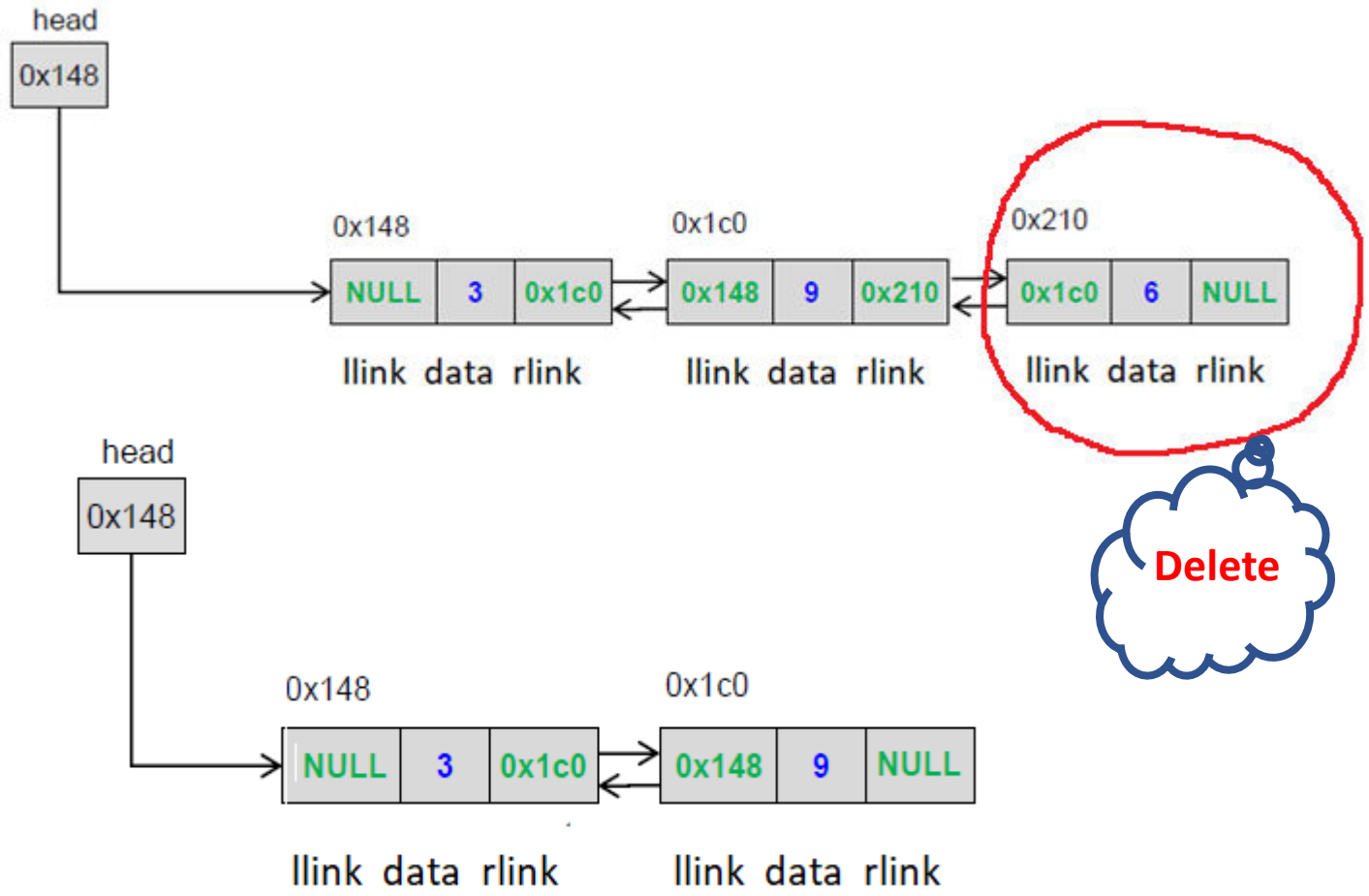
Deleting last node

What will change??

- Case I : Empty Linked List
- Case II : Linked list with a single node
  first node gets freed up
  head points to NULL
- Case III : Linked List with more than one node
  Second last node rlink point to NULL
  last node gets freed up

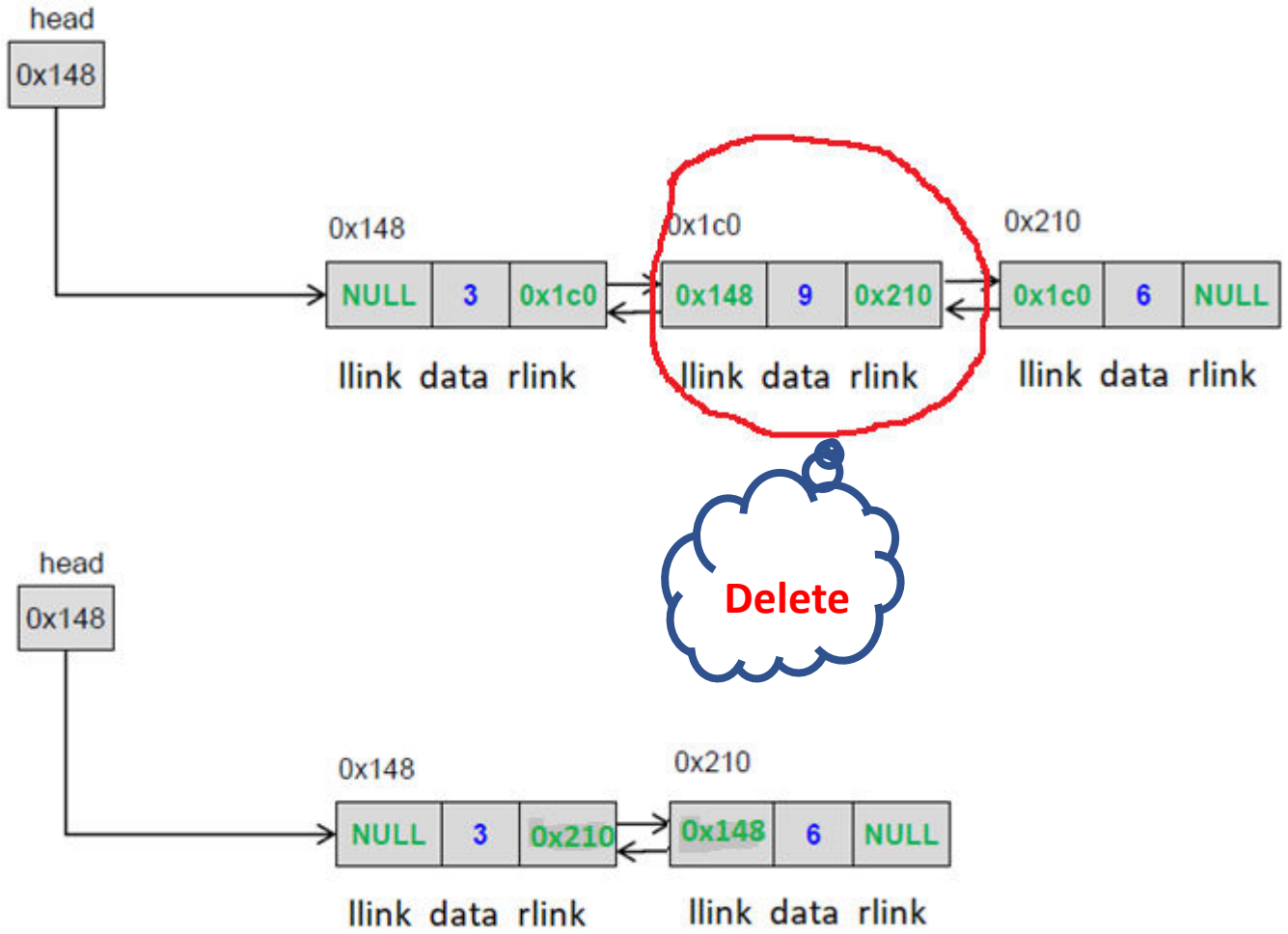
Deleting last node

- Case II : Linked List with more than one node



Deleting a node at intermediate position

- Case II : Linked List with more than one node



Doubly Linked List insert operation

Apply the concepts to implement following operations for a Doubly linked list

- reverse a doubly linked list
- Find the node pairs with a given sum in a doubly linked list
- Insert a node after a node with a given value
- Remove duplicate nodes from a doubly linked list

DATA STRUCTURES AND ITS APPLICATIONS

Doubly Linked List

Vandana M L
Department of Computer Science and Engineering

Deleting a node

There are 3 cases

- Deleting first node
- Deleting last node
- Deleting a node at a given position

Deleting a node

There are 3 cases

- Deleting first node
- Deleting last node
- Deleting a node at a given position

Deleting first node

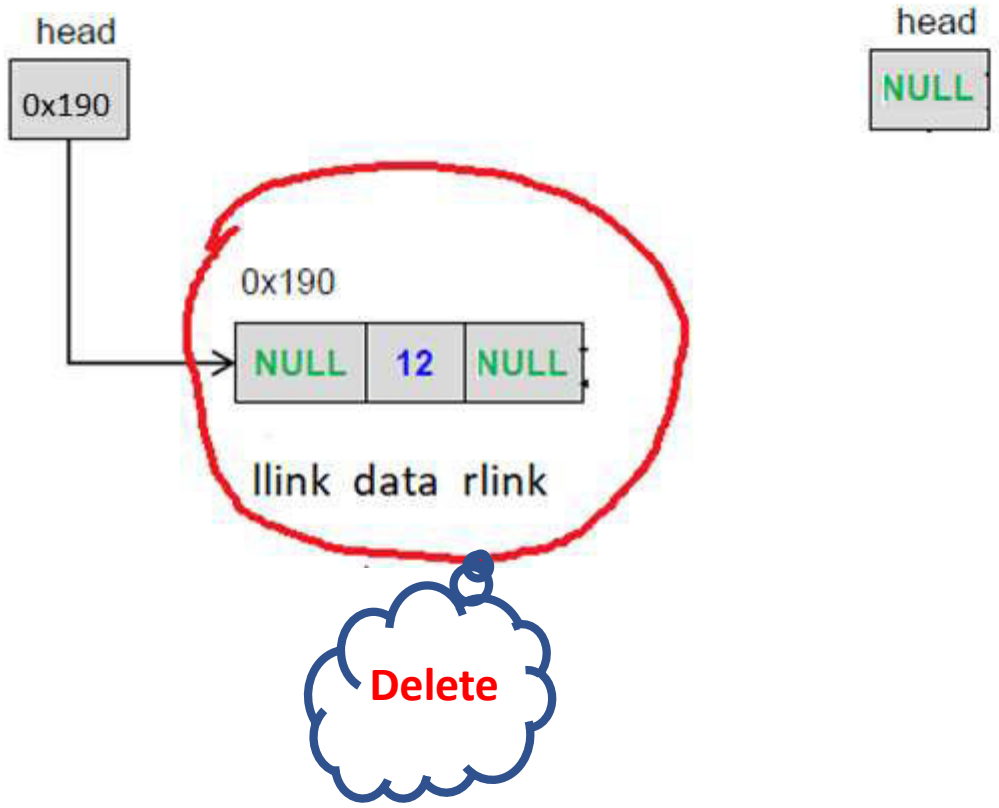
What will change??

- Case I : Empty Linked List
- Case II : Linked list with a single node
  first node gets freed up
  head points to NULL
- Case III : Linked List with more than one node
  Second node llink gets changed to NULL
  first node gets freed off

Doubly Linked List Operations

Deleting first node

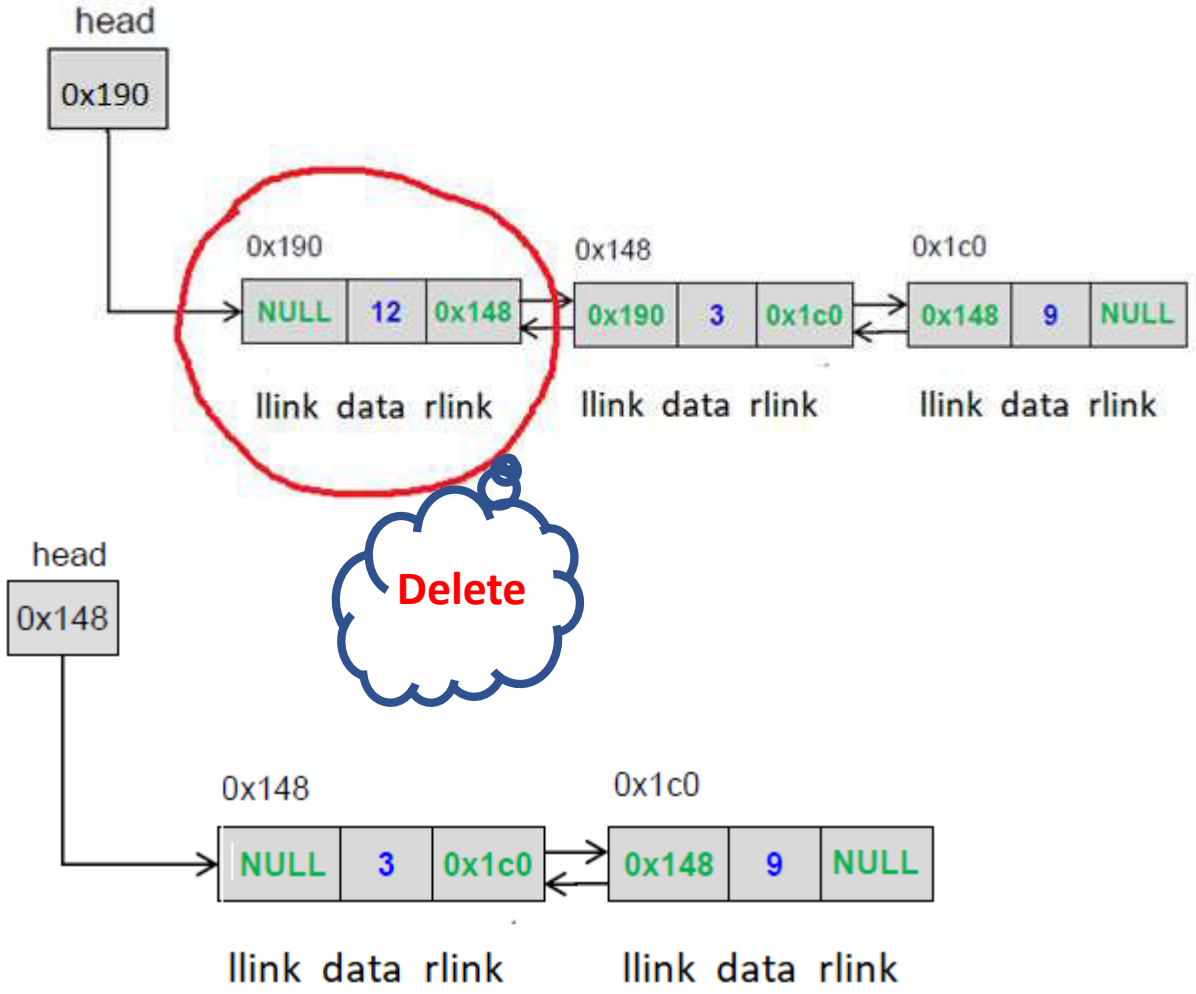
- Case II : Linked list with a single node



Doubly Linked List Implementation

Deleting first node

- Case III : Linked List with more than one node



Deleting last node

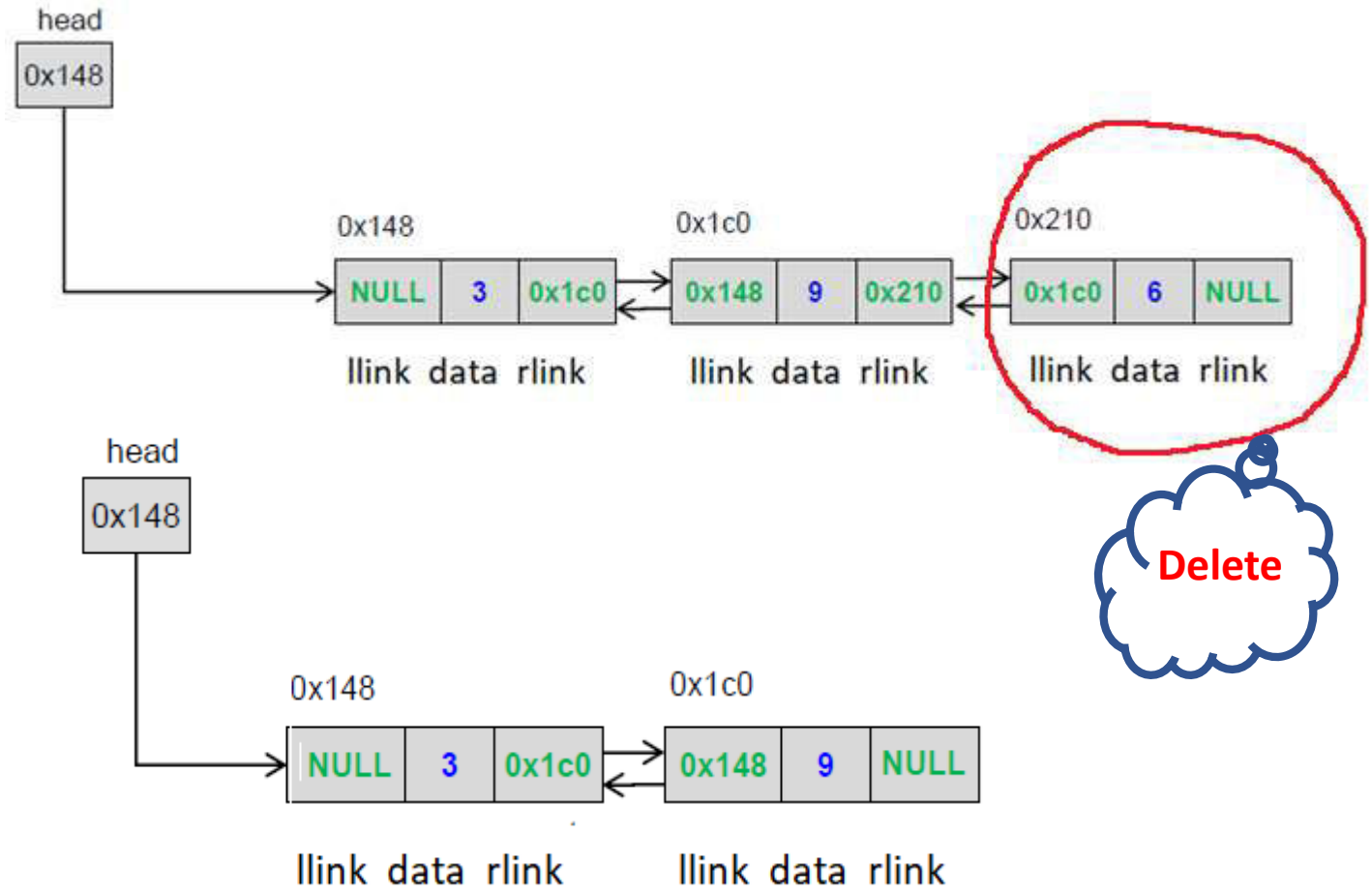
What will change??

- Case I : Empty Linked List
- Case II : Linked list with a single node
  first node gets freed up
  head points to NULL
- Case III : Linked List with more than one node
  Second last node rlink point to NULL
  last node gets freed up

Doubly Linked List Implementation

Deleting last node

- Case II : Linked List with more than one node



Deleting a node at intermediate position

- Traverse list to find the desired position, keep track of the previous node

If position is found

If position is 1

- Delete from front

else

If it is last position

- Delete from end

else

if intermediate position

- Change previous node rlink to rlink of current node
- Change llink of node following current node to previous node
- Delete current node

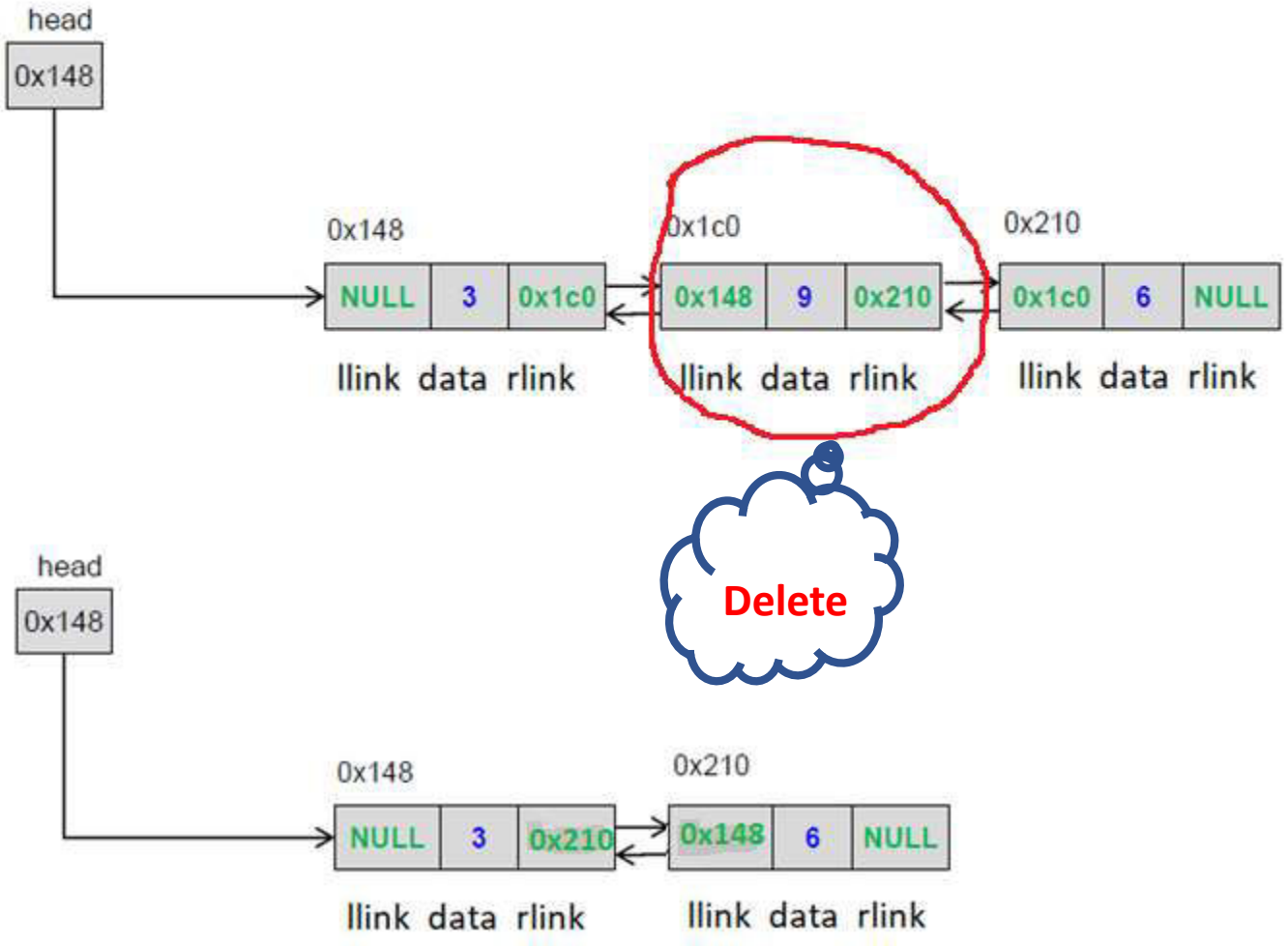
else

invalid position

Doubly Linked List Operations

Deleting a node at intermediate position

- Case II : Linked List with more than one node



Doubly Linked List insert operation

Apply the concepts to implement following operations for a Doubly linked list

- reverse a doubly linked list
- Remove duplicate nodes from a doubly linked list
- Delete a node with a given key value from doubly linked list

DATA STRUCTURES AND ITS APPLICATIONS

Circular Singly Linked List

Vandana M L
Department of Computer Science and Engineering

Circular Linked List

Circular linked list is a linked list where all nodes are connected to form a circle.

- Circular Singly Linked List
- Circular Doubly Linked List

  With additional head node
  Without additional head node

DATA STRUCTURES AND ITS APPLICATIONS

Circular Linked List Operations

Insert a node

- Insert at front
- Insert at end
- Insert at a position
- Ordered insertion

Delete node

- Delete front node
- Delete end node
- Delete a node from position
- Delete a node with a given value

Additional

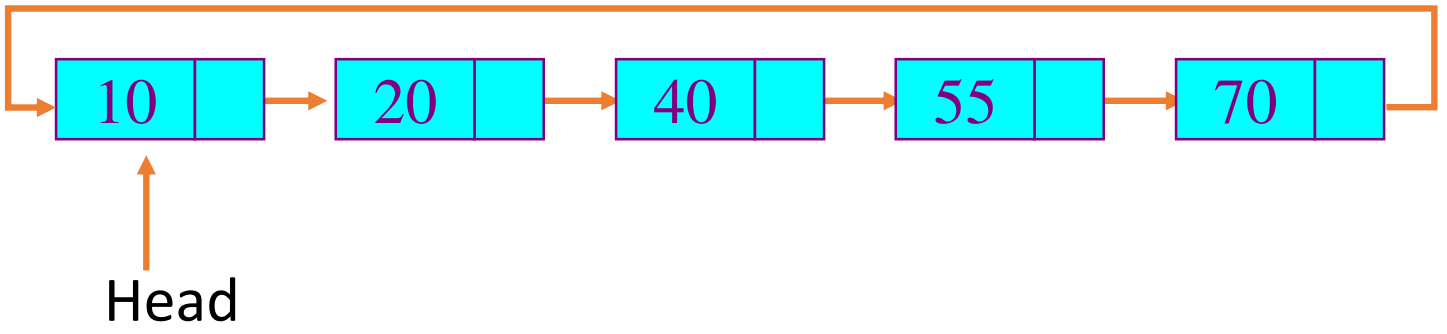
- Display list
- Concatenate two list
- reverse a list

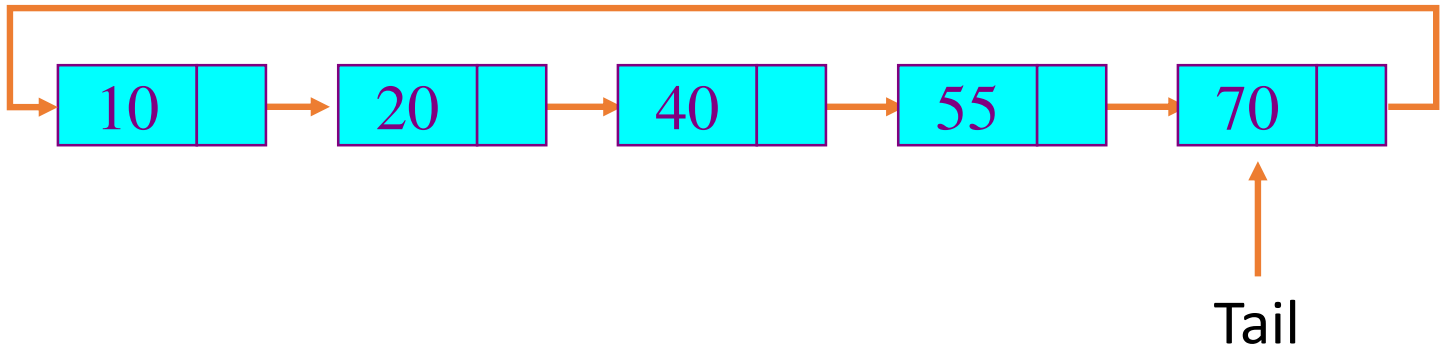
Circular Linked List: Applications

- Useful for implementation of queue, eliminates the need to maintain two pointers as in case of queue implementation using arrays
- Circular linked lists are useful for applications to repeatedly go around the list like playing video and sound files in “looping” mode
- Advanced data structures like Fibonacci Heap Implementation
- Plays a key role in linked implementation of graphs

Circular Singly 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 Tail pointer is often used instead of a Head pointer





Circular Singly Linked List Node Definition

```
#include <iostream>
using namespace std;

struct Node{
   int data;
   struct Node* next;
};
typedef struct node csll_node;
```

Insertion at the beginning

Insert at the front of linked list

- Create a node

If the list is empty

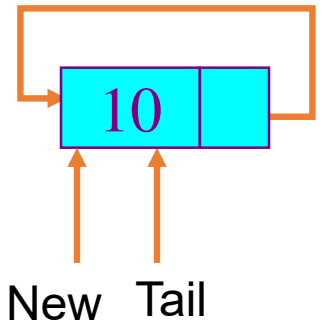
- make the tail pointer point towards the new node

Else

- Change the new node link field to point to the first node
- Change the last node link to point to the new node

Circular Singly Linked List Operations

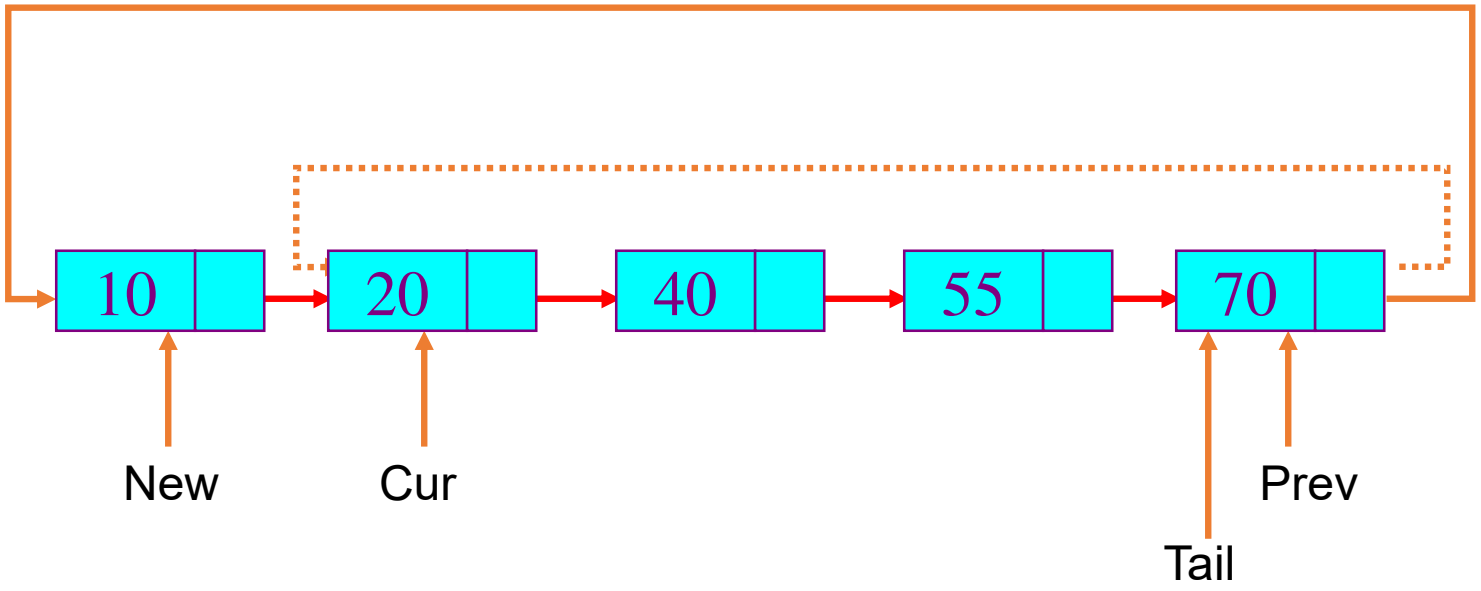
Insertion into an empty list



Insert to head of a Circular Linked List

New->next = Cur; ⟹ New->next = Tail->next;

Prev->next = New; ⟹ Tail->next = New;

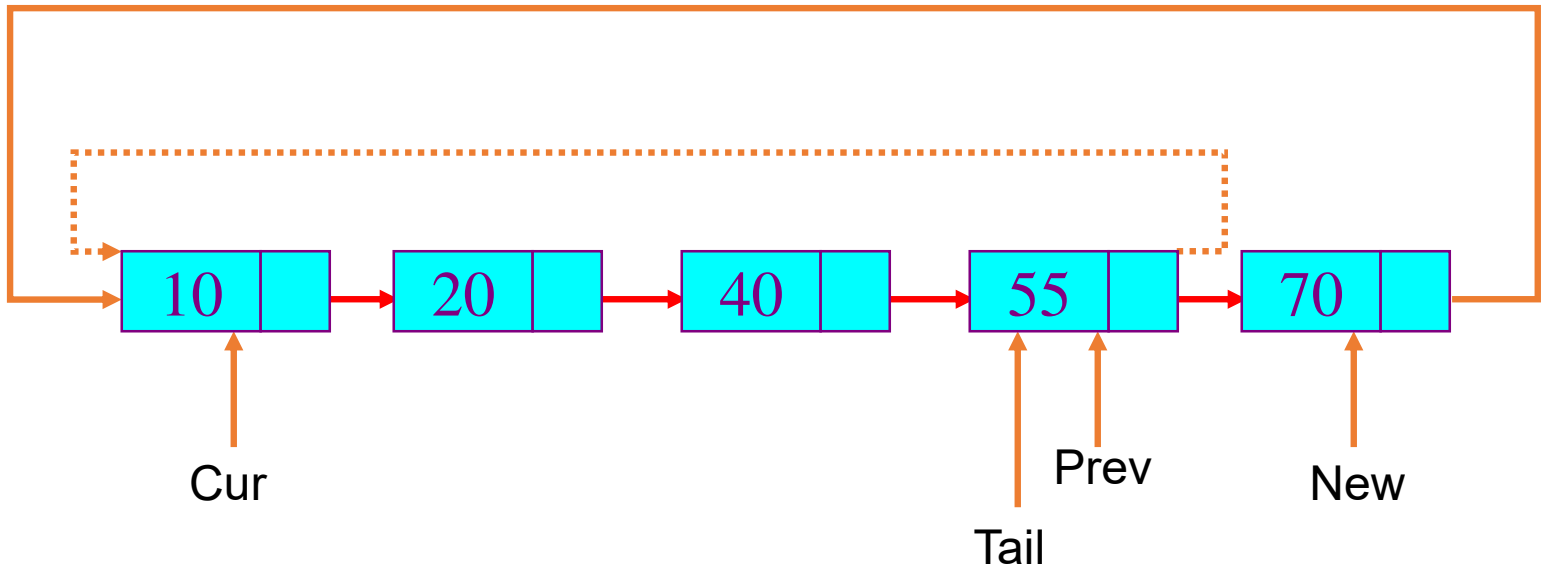


Insert to the end of a Circular Linked List

New->next = Cur; ⟹ New->next = Tail->next;

Prev->next = New; ⟹ Tail->next = New;

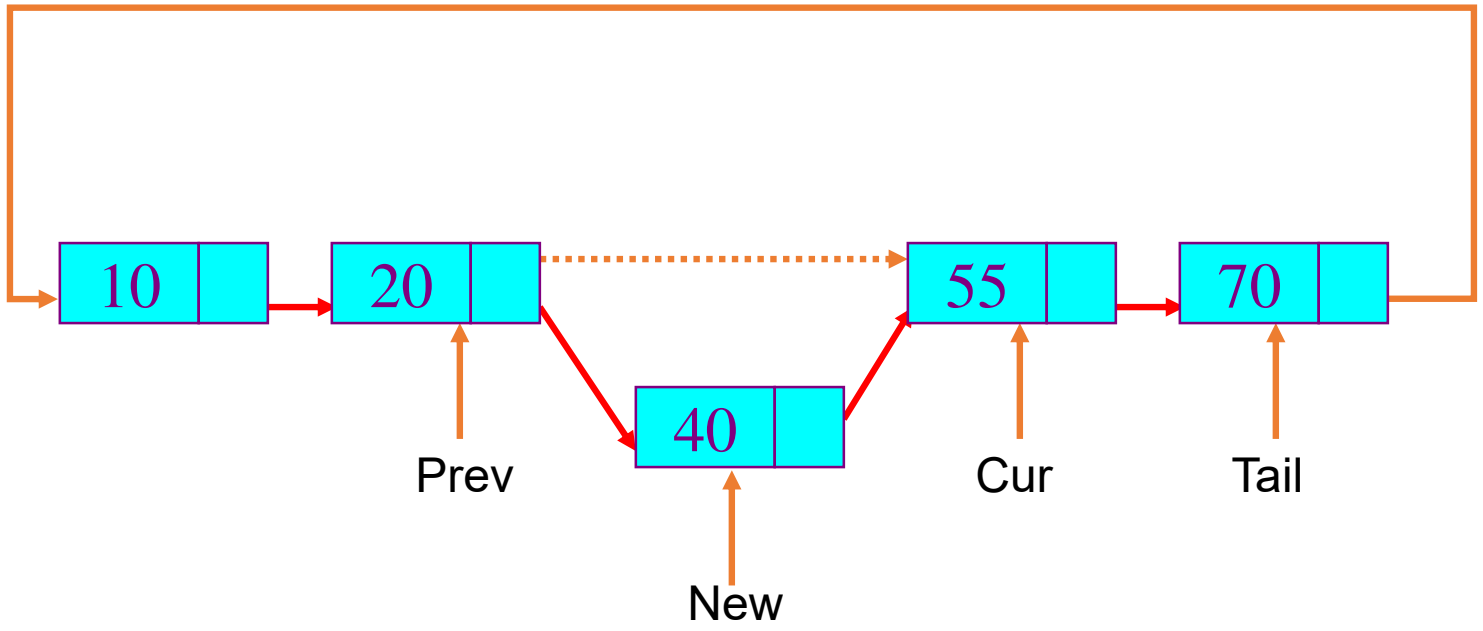
Tail = New;



Insert to the middle of Circular Linked List

New->next = Cur;

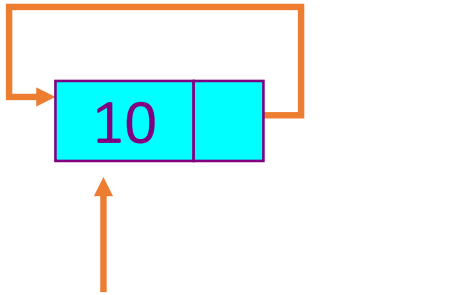
Prev->next = New;



Delete a node from a single-node Circular Linked List

Tail = NULL;

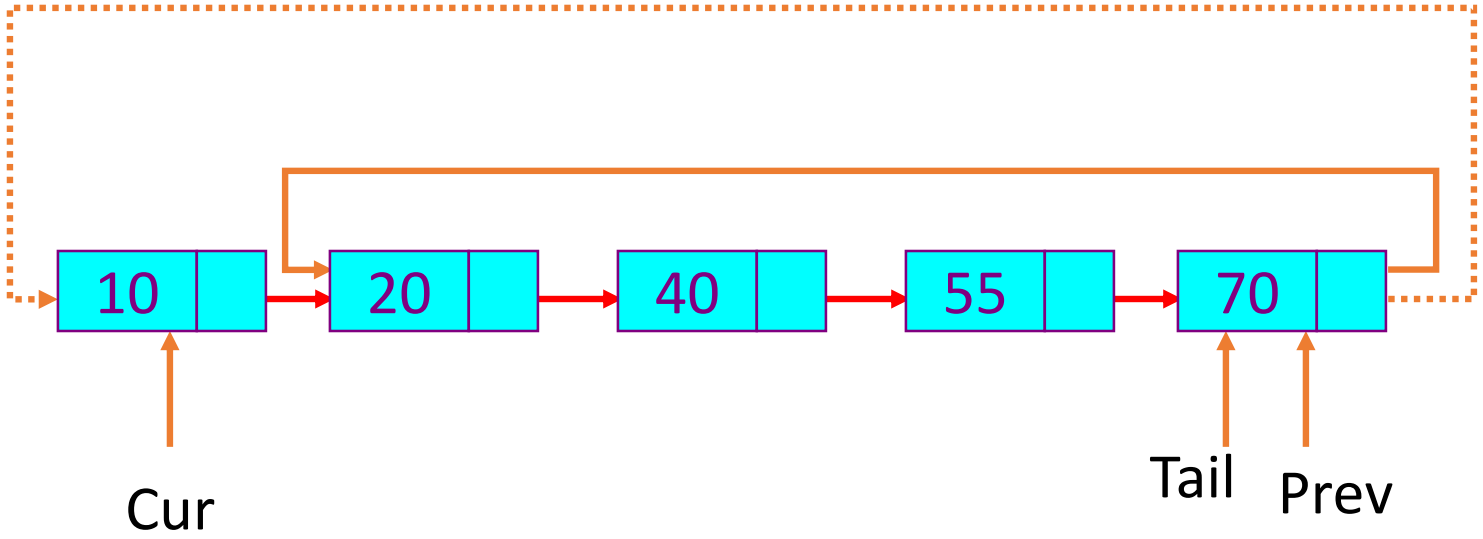
free(Cur);



Tail = Cur = Prev

Delete the head node from a Circular Linked List

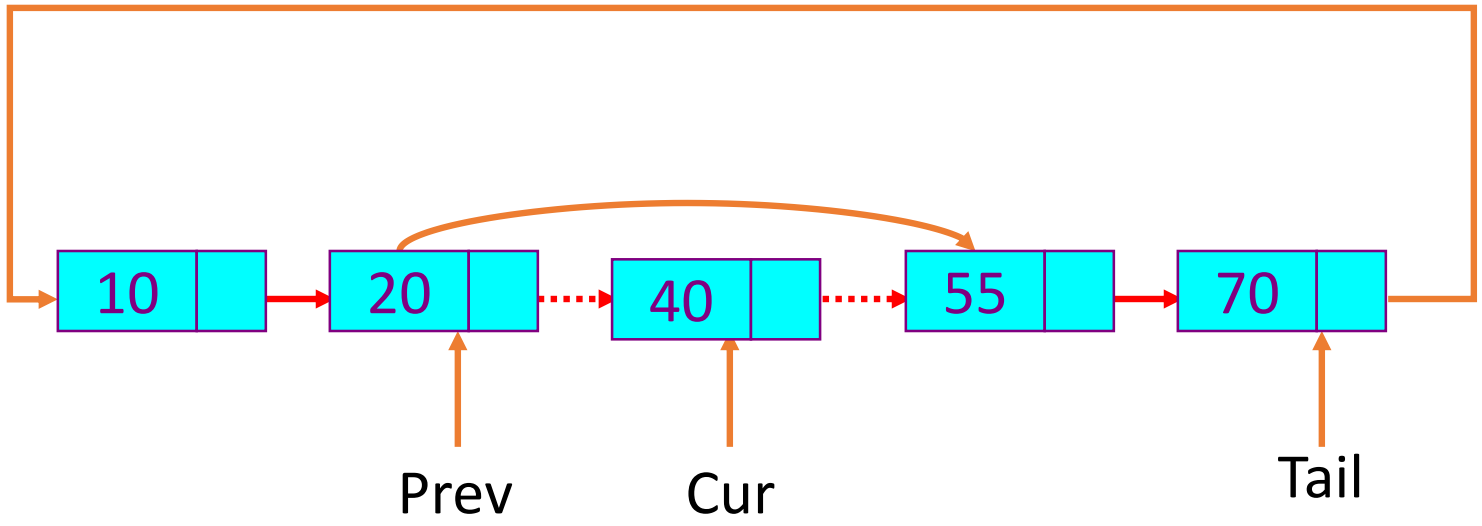
```
Prev->next = Cur->next;          // same as: Tail->next = Cur->next

free(cur);
```



Delete a middle node Cur from a Circular Linked List

Prev->next = Cur->next;

Free(Cur);



Circular Singly Linked List operations

Apply the concepts to implement following operations for a singly linked list

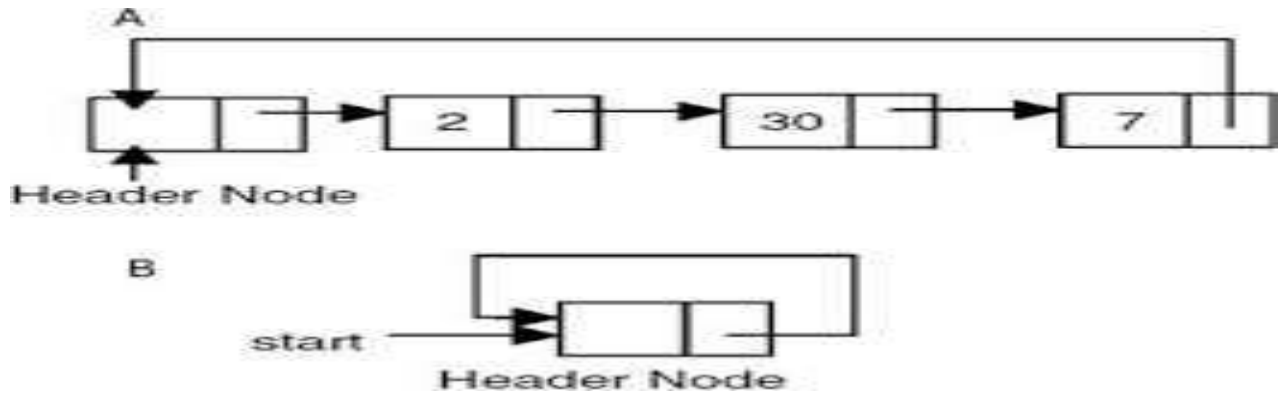
- insert a node after a given node(pointer)
- Insert a node after a node with a given value

DATA STRUCTURES AND ITS APPLICATIONS

Circular Linked Lists

Dinesh Singh
Department of Computer Science & Engineering

Circular Linked Lists



- The first node in the list is the header node.
- The address part of the last node points to the header node
- Circular list does not have a natural first or the last node
- An External pointer points to the header node and the one following this node is the first node.

Operations on Circular Linked Lists

Implementation of some operations on Circular linked Lists with header node

- Insert at the head of a list
- Insert at the end of the List
- Delete a Node given its value

Note: head is a pointer to the header node and the following node is the first node

Creating Header node

```
struct node *create_head()
 {
   struct node *temp;
   temp=(struct node*)malloc(sizeof(struct node));
   temp->data=0; // keeps the count of nodes in the list
   temp->next=temp;
   return temp;
  }
```

Operations on Circular Linked Lists

Algorithm to insert a node at the head of the list
insert_head(p,x)
//p pointer to header node, x element to be inserted
//x gets inserted after the header node

allocate memory for the node
initialise the node
//insert the new node after the header node
Copy the value of the next part of the header node into the next part of the new node
Copy the address of the new node into next part of the header node

Operations on Circular Linked Lists with header node

```
void insert_head(struct node *p,int x)
//p points to the header node, x element to be inserted
 {
  struct node *temp;
  //create node and initialise
   temp=(struct node*)malloc(sizeof(struct node));
   temp->data=x;
// next part of new node points to the node after the header node
   temp->next=p->next;
   p->next=temp; //next part of header node points to the new node
   p->data++;
  }
```

Operations on Circular Linked Lists with header node

Algorithm to insert a node at the end of the list
insert_tail(p,x)
//p pointer to header node, x element to be inserted
allocate memory for the node
initialise the node

move to the last node
//insert the new node after the last node
Copy the address of the header node into next of new node
Copy the address of the new node into the next of last node

Operations on Circular Linked Lists with header node

Algorithm to insert a node at the end of the list

```
void insert_tail(struct node *p,int x)
 {
  struct node *temp,*q;
   temp=(struct node*)malloc(sizeof(struct node));
   temp->data=x;

   q=p->next; // go the first node

   while(q->next!=p) // move to the last node
    q=q->next;

 temp->next=p;// copy address of header node into next of new node
 q->next=temp; // copy the address of new node into next of the last node
 p->data++; // increment the count of nodes in the list
 }
```

Operations on Circular Linked Lists with header node

Algorithm to delete a node given its value
delete_node(p,x)
//p pointer to header node, x element to be deleted

move forward until the node to be deleted is found
or header node is reaches
If(node found)
  delete the node by adjusting the pointers
else
 node not found // if header node is reached

Operations on Circular Linked Lists with header node

```
void delete_node(struct node *p, int x)
 {
  //p points to the header node, x is element to be inserted
   struct node *prev,*q;

   q=p->next; // go to the first node
   prev=p;
  //move forward until the data is found or header node is reached
   while((q!=p)&&(q->data!=x))
   {
    prev=q; // keep track of the previous node
    q=q->next;
   }
```

Operations on Circular Linked Lists with header node


```
if(q==p) // header node reached
    printf("Node not found..\n");
  else
  {
   prev->next=q->next; //delete the node
   free(q);
   p->data--; // decrement the count of nodes in the list
  }
}
```

DATA STRUCTURES AND ITS APPLICATIONS

Circular Doubly Linked List

Vandana M L
Department of Computer Science and Engineering

Circular Doubly Linked List

Node Structure Definition

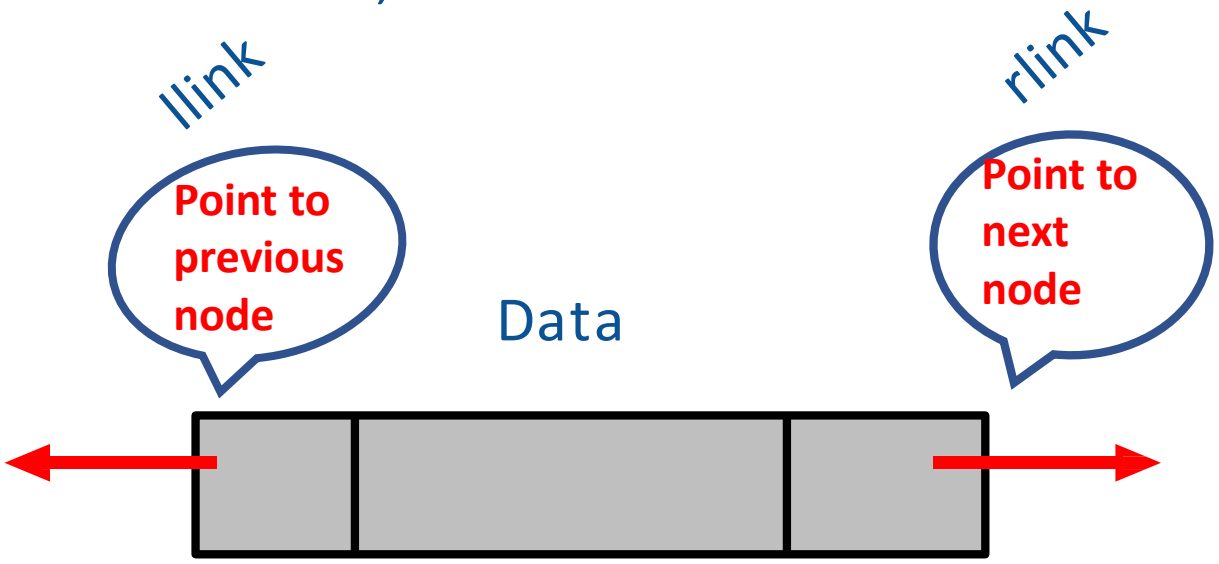
A doubly linked list node contain three fields:

- Data
- link to the next node
- link to the previous node.

Circular Doubly Linked List

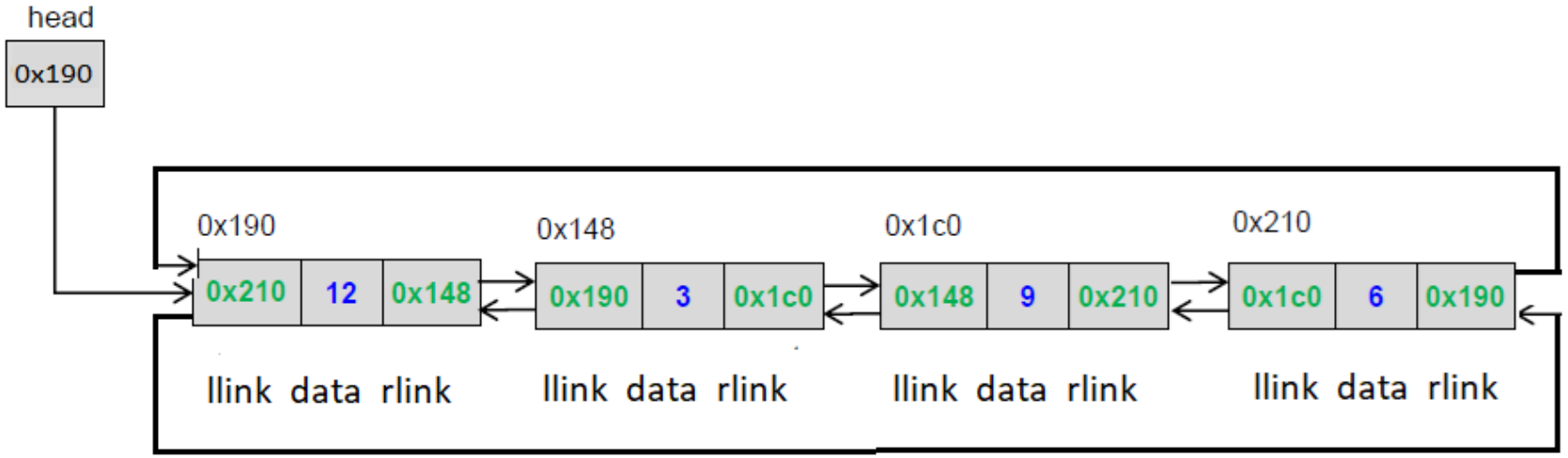
Node Structure Definition

```
struct node
{
   int data;
   struct node*llink;
   struct node*rlink;
};
```



DATA STRUCTURES AND ITS APPLICATIONS

Circular Doubly Linked List: Example



Creating a node

- Allocate memory for the node dynamically
- If the memory is allocated successfully
  - set the data part
  - set the llink and rlink to NULL

Inserting a node

There are 3 cases

- Insertion at the beginning
- Insertion at the end
- Insertion at a given position

Insertion at the beginning

What all will change

Case 1: linked list empty

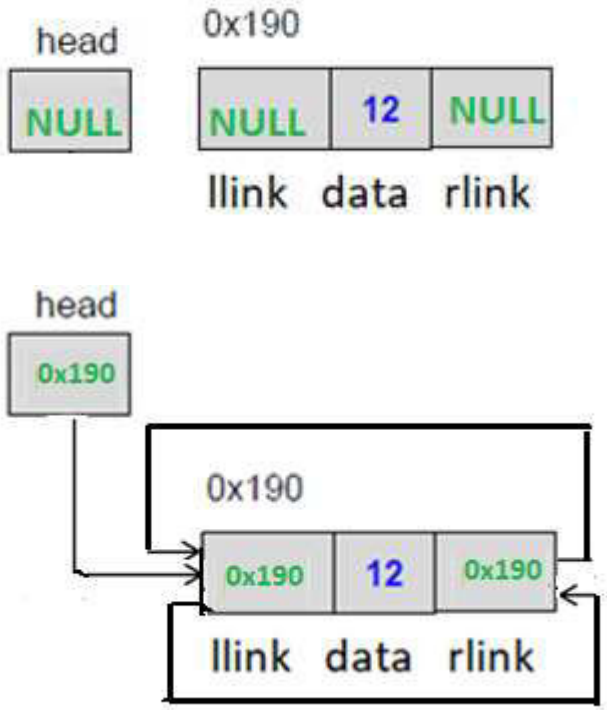
- Head pointer

Case 2: linked list is not empty

- Head pointer
- New front node's rlink and llink
- Old front node's llink
- Last node's rlink

Circular Doubly Linked List Operations

Insertion at the beginning (Case1)

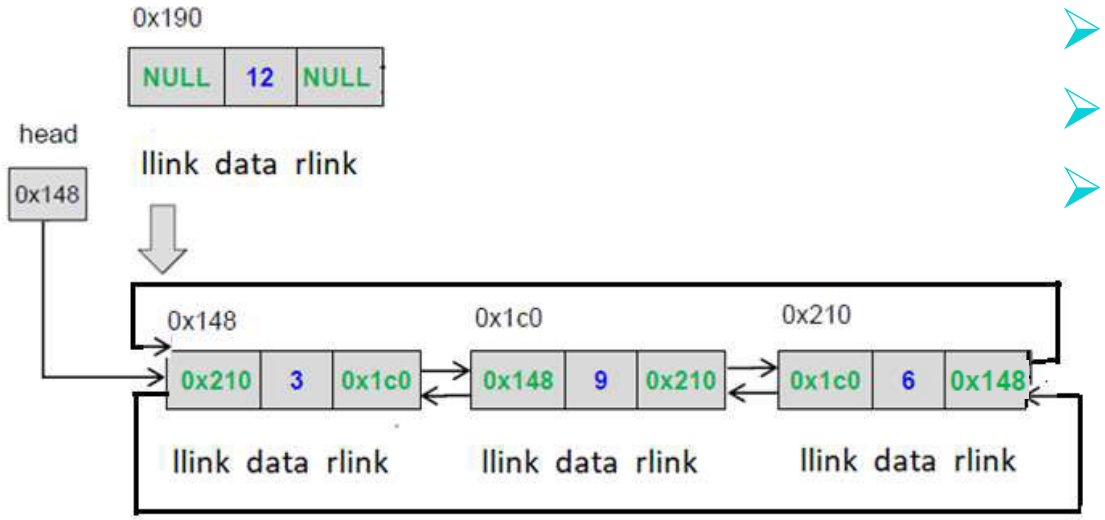


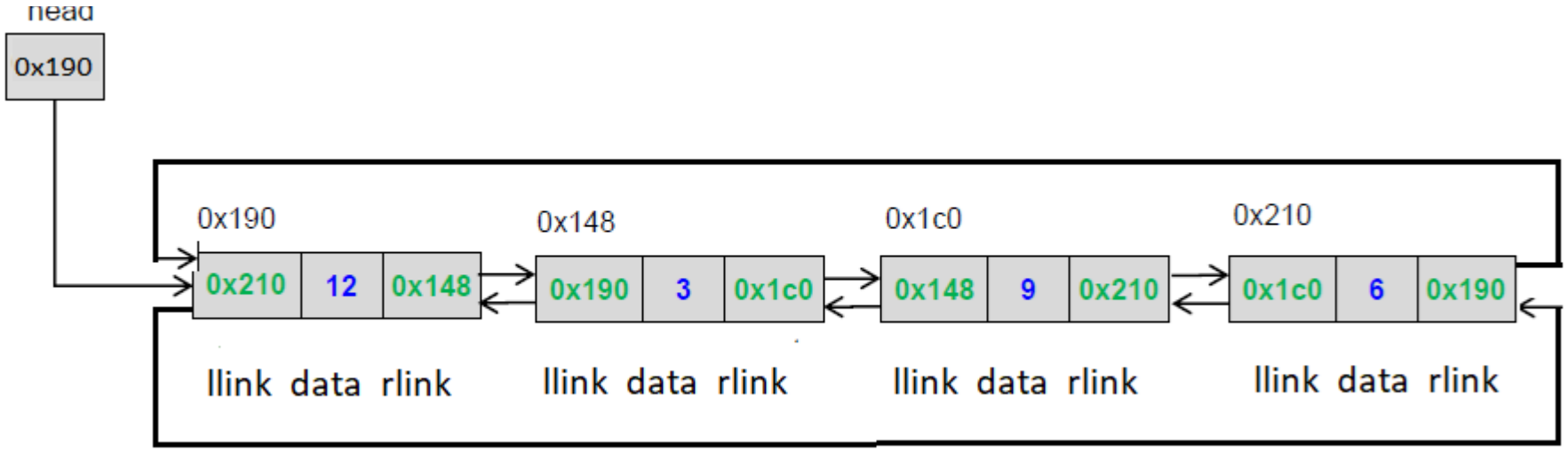
DATA STRUCTURES AND ITS APPLICATIONS

Circular Doubly Linked List Operations

Insertion at the beginning(Case 2)

- Head pointer
- New front node's rlink and llink
- Old front node's llink
- Last node's rlink





Insertion at the end

What all will change

Case 1: linked list empty

- Head pointer

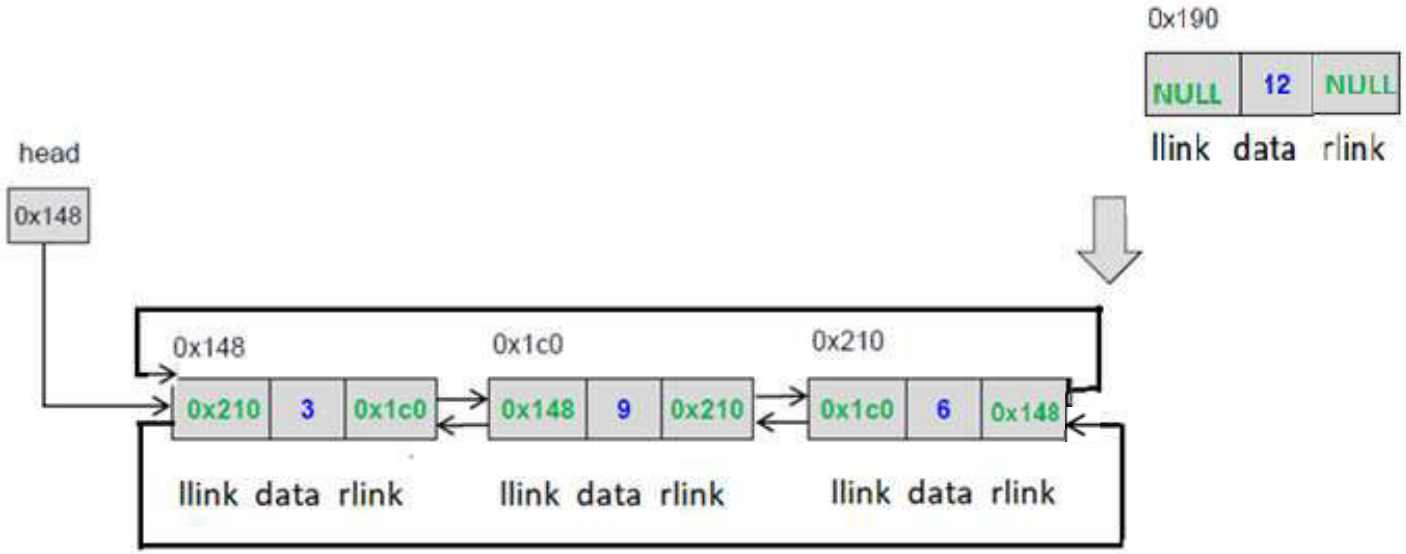
Case 2: linked list is not empty else

- Last node's rlink
- First node llink
- New node's llink and rlink

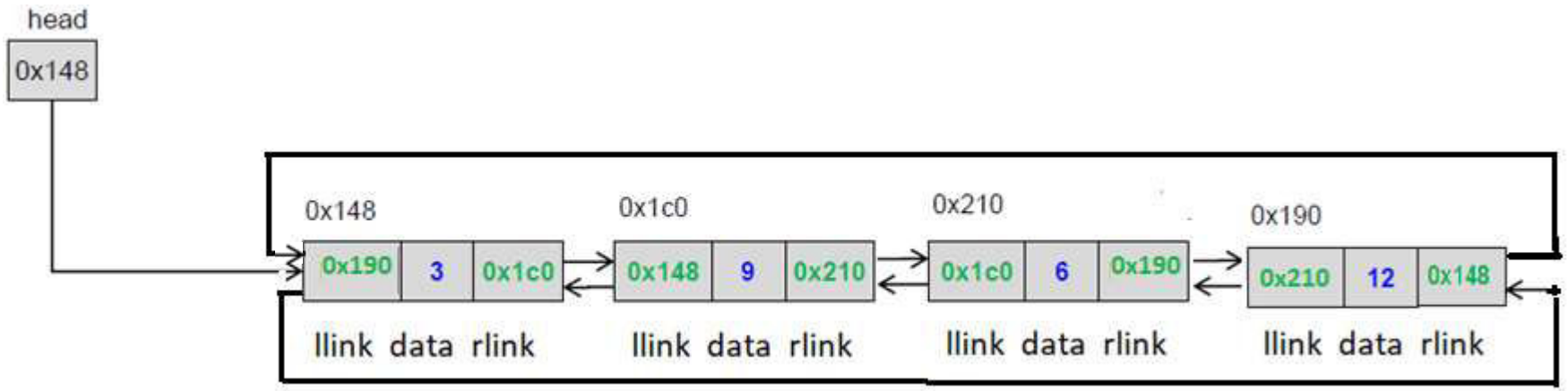
DATA STRUCTURES AND ITS APPLICATIONS

Circular Doubly Linked List Operations

Insertion at the end



node's rlink
: node llink
v node's llink and rlink



Insertion at the given position

- Create a node

If the list is empty

- make the start pointer point towards the new node;

Else

if it is first position

- Insert at front

else

- Traverse the linked list to reach given position
- Keep track of the previous node

If it is valid position

intermediate position

- Change link fields of current previous and intermediate node

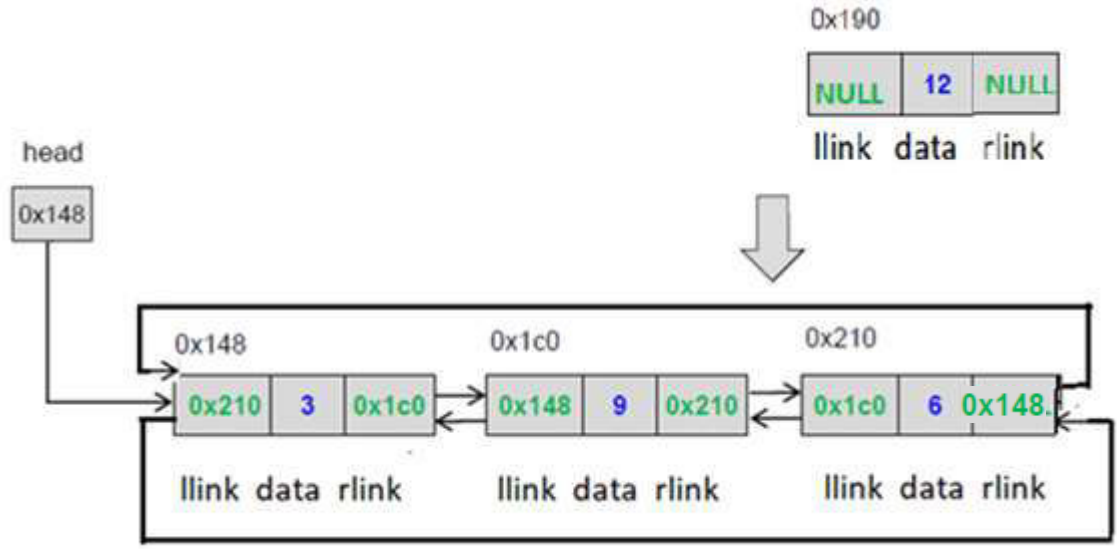
last position

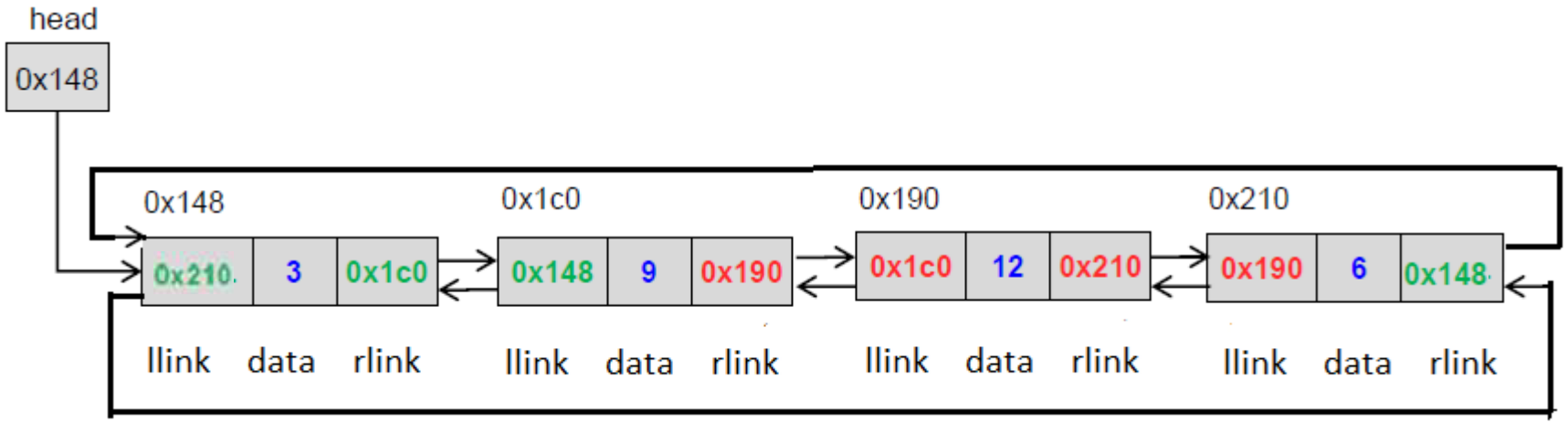
- insert at end

DATA STRUCTURES AND ITS APPLICATIONS

Circular Doubly Linked List Operations

Insertion at the given position





Deleting a node

There are 3 cases

- Deleting first node
- Deleting last node
- Deleting a node at a given position

Deleting a node

There are 3 cases

- Deleting first node
- Deleting last node
- Deleting a node at a given position

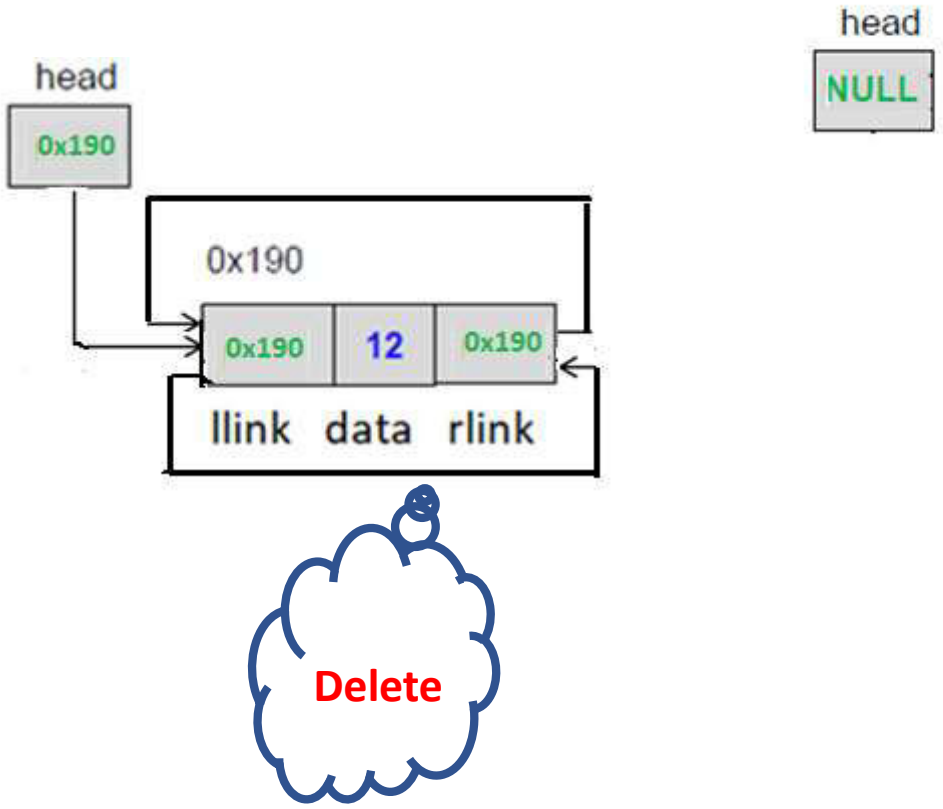
Deleting first node

What will change??

- Case I : Empty Linked List
- Case II : Linked list with a single node
  first node gets freed up
  head points to NULL
- Case III : Linked List with more than one node
  Second node llink
  last node rlink
  first node gets freed off
  head pointer points to second node

Deleting first node

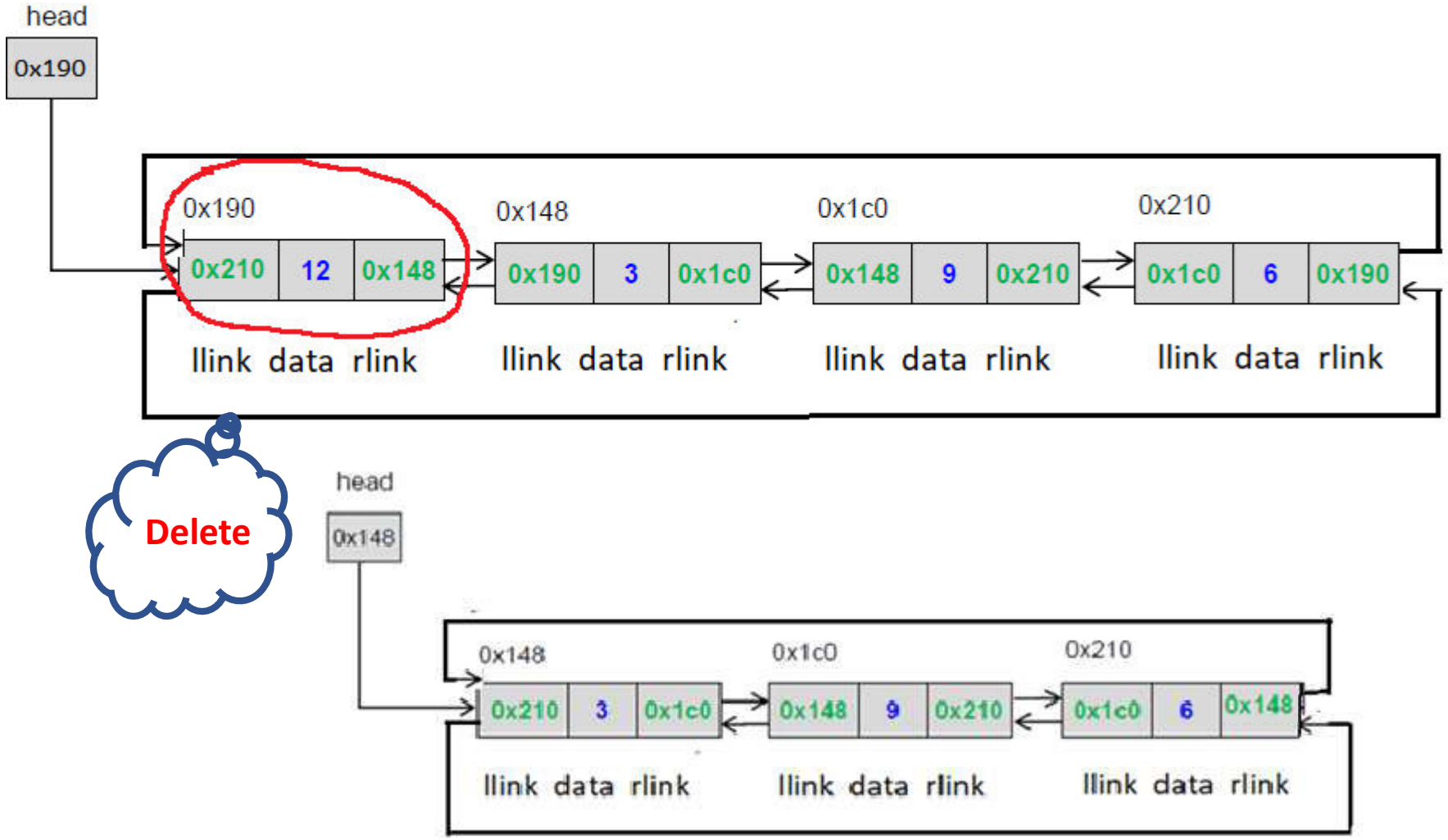
- Case II : Linked list with a single node



Circular Doubly Linked List Operations

Deleting first node

- Case III : Linked List with more than one node



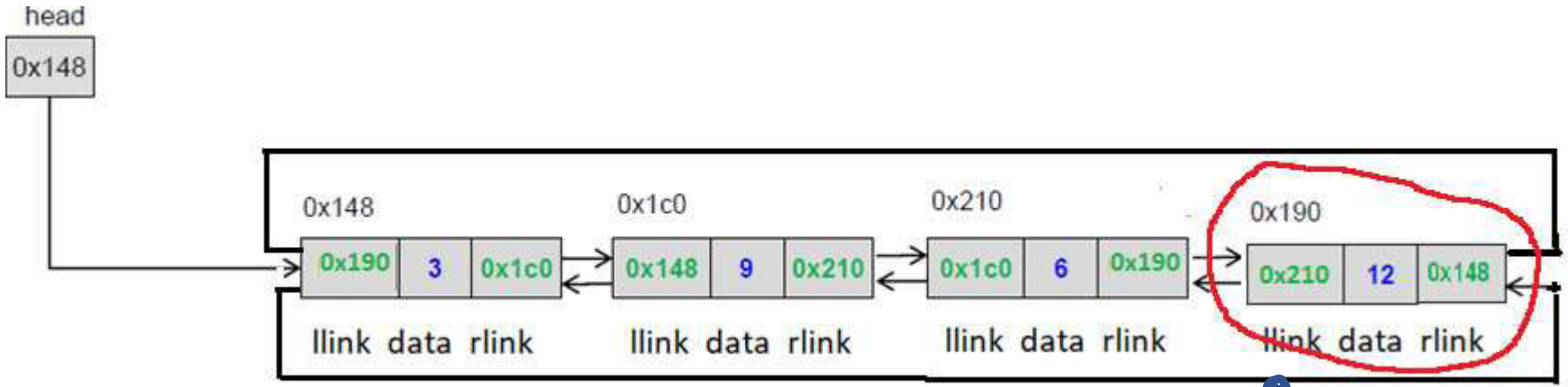
Deleting last node

What will change??

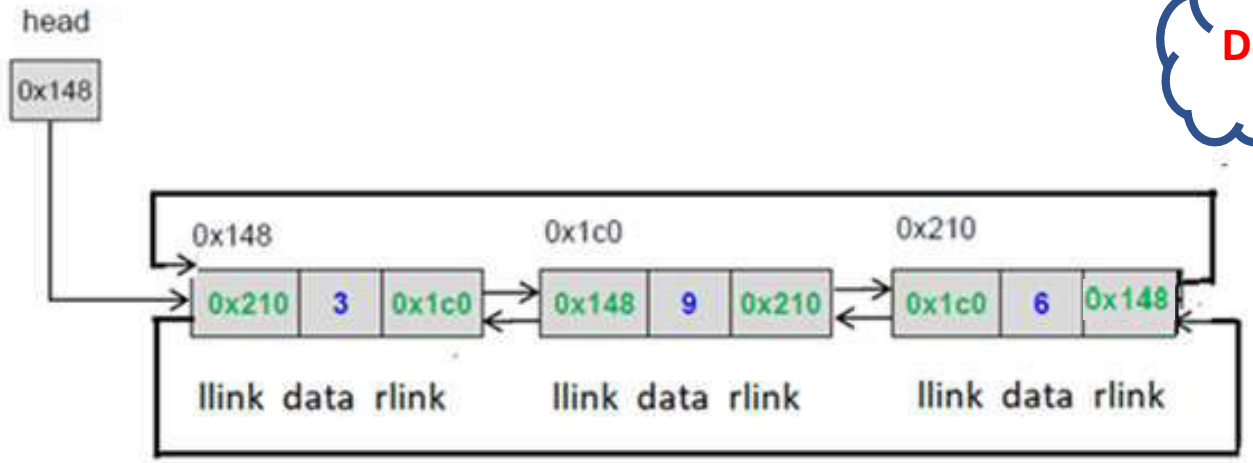
- Case I : Empty Linked List
- Case II : Linked list with a single node
  first node gets freed up
  head points to NULL
- Case III : Linked List with more than one node
  Second last node rlink
  first node llink
  last node gets freed up

Deleting last node

- Case II : Linked List with more than one node

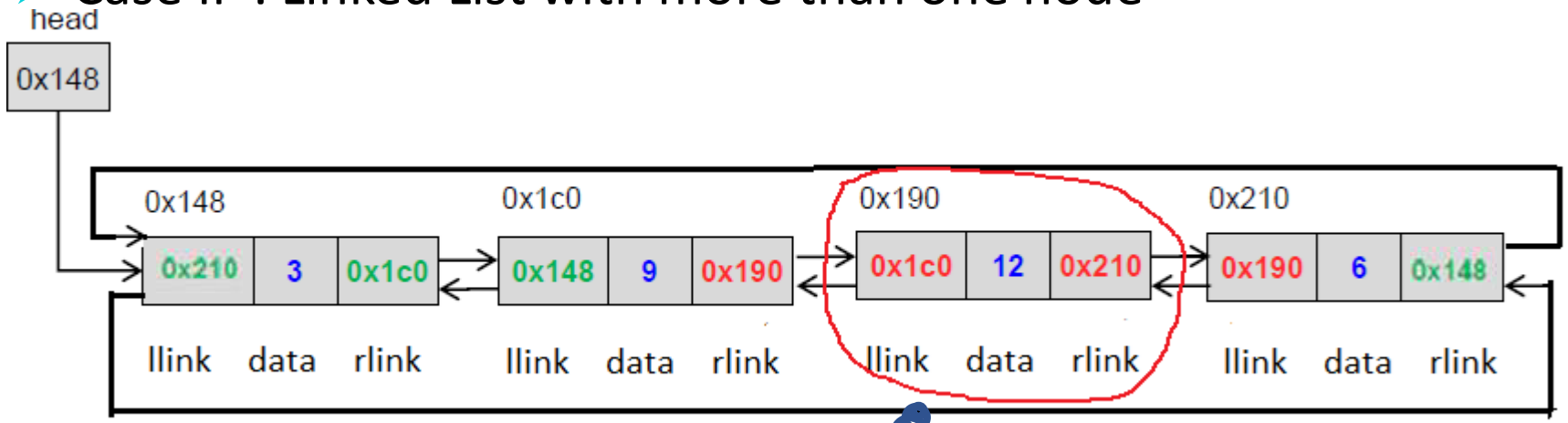


Delete

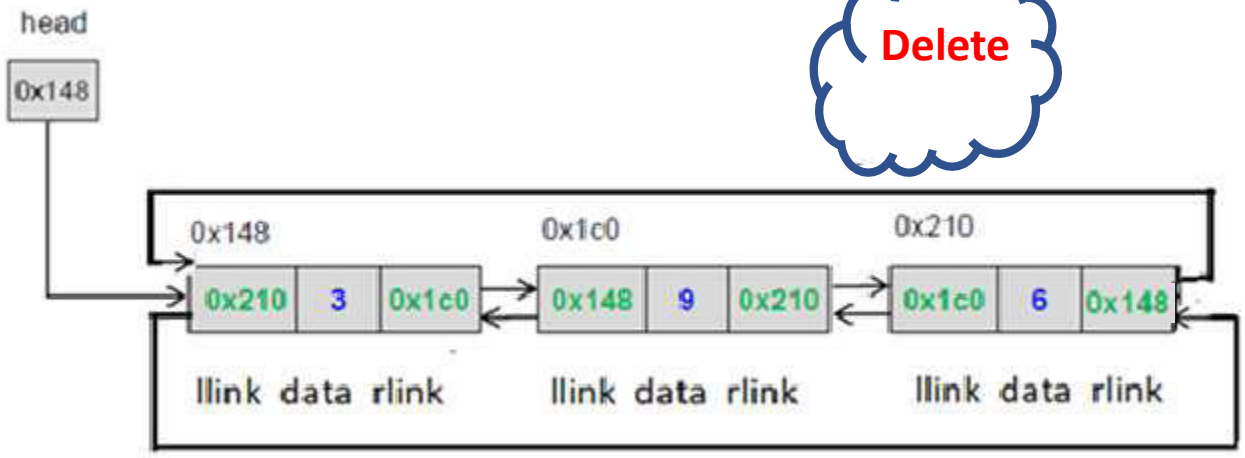


Deleting a node at intermediate position

- Case II : Linked List with more than one node



Delete



Circular doubly Linked List operations

Apply the concepts to implement following operations for a doubly circular linked list

- Reverse list using recursion
- Search given element in the list
- Find the largest value in the list

DATA STRUCTURES AND ITS APPLICATIONS

Multilist Representation

Prof. Vandana M L
Department of Computer Science and Engineering

Sparse Matrix

Matrix ??

Two Dimensional data

1 1 3 0 4

1 3 5 1 0

9 0 5 1 0

Sparse Matrix??

More zero elements than non zero elements

0 0 3 0 0

0 0 5 1 0

0 0 0 0 0

Sparse Matrix Representation

- 2D Matrix

  results in lot of memory wastage as non zero elements are also stored
- Triple Notation

  Array representation
- Multilist Representation

  Linked representation hence size can be changed dynamically

DATA STRUCTURES AND ITS APPLICATIONS

Sparse Matrix Representation: Triple Notation

In triple notation sparse matrix is represented as an array of tuple values.
Each tuple consists of
<rowno columnno Value>

The first block in array block holds information regarding
<total no of rows, total no of columns ,value>

$$\begin{bmatrix} 2 & 0 & 0 & 0 \\ 4 & 0 & 0 & 3 \\ 0 & 0 & 0 & 0 \\ 8 & 0 & 0 & 1 \\ 0 & 0 & 6 & 0 \end{bmatrix}$$

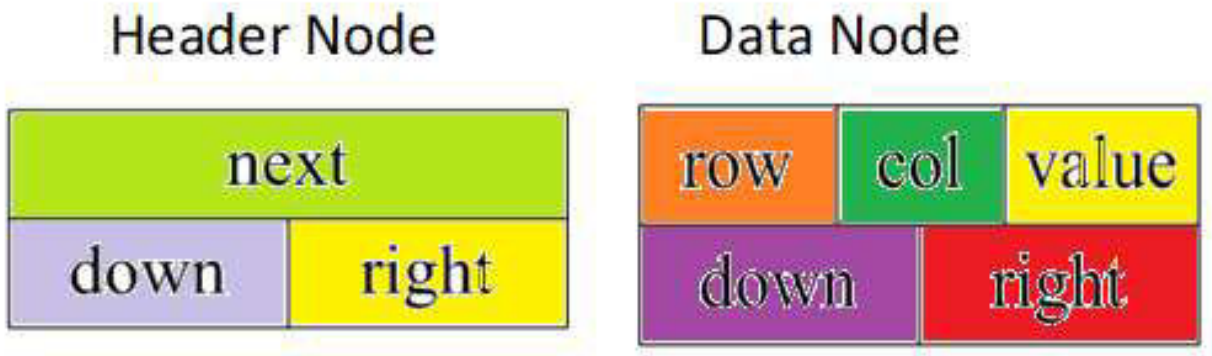
Triple Notation

Row No	Column No	Value
5	4	6
0	0	2
1	0	4
1	3	3
3	0	8
3	3	1
4	2	6

Sparse Matrix Representation: Linked representation

Node Structure

Two types of nodes are used



Sparse Matrix Representation: Linked representation

Node Structure Definition

```
#define MAX_SIZE 50 /* size of largest
matrix */
typedef enum {head, entry} tagfield;
typedef struct matrixNode * matrixPointer;
typedef struct entryNode {
int row;
int col;
int value; };
```

```
typedef struct matrixNode {
    matrixPointer down;
    matrixPointer right;
    tagfield tag;
    union
    {
    matrixPointer next;
    entryNode entry;
    } u;
};
```

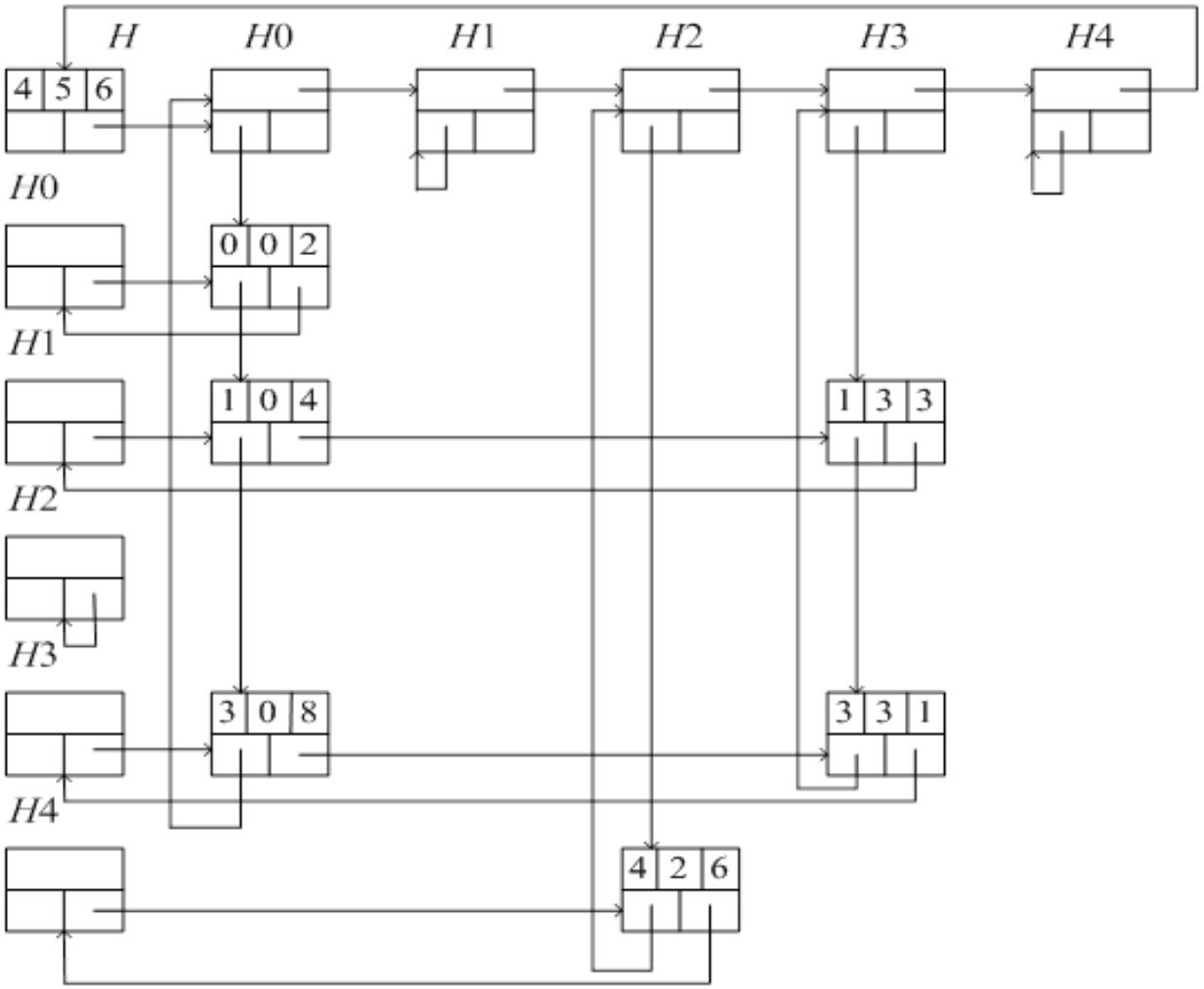
Sparse Matrix Representation: Linked representation

Example

$$\begin{bmatrix} 2 & 0 & 0 & 0 \\ 4 & 0 & 0 & 3 \\ 0 & 0 & 0 & 0 \\ 8 & 0 & 0 & 1 \\ 0 & 0 & 6 & 0 \end{bmatrix}$$

DATA STRUCTURES AND ITS APPLICATIONS

Sparse Matrix Representation: Linked representation



Courtesy: “Fundamentals of Data Structures” By Ellis Horowitz and Sartaj Sahni

DATA STRUCTURES AND ITS APPLICATIONS

Sparse Matrix Representation Summary

Sparse matrix representation

- Triple
- Linked Representation

Concepts can be applied to implement the following operations

- Create_SparseMatrix()
- Transpose_of_SparseMatrix()
- Add_SparseMatrices()
- Multiple_SparseMatrices()

DATA STRUCTURES & ITS APPLICATIONS

UNIT 1: Skip List

Kusuma K V
Department of Computer Science & Engineering

Performance considerations: lists

- Size():
  - O(1) if size is stored explicitly, else O(n)
- IsEmpty():
  - O(1)
- FindElement(k):
  - O(n)
- InsertItem(k, e):
  - O(1) (assumes item already found)
- Remove(k):
  - O(1) (assumes item already found)

Skip List

- Skip Lists support O(log n)
  - Insertion
  - Deletion
  - Search
- A relatively recent data structure
  - W. Pugh in 1989
- “A probabilistic alternative to balanced trees”

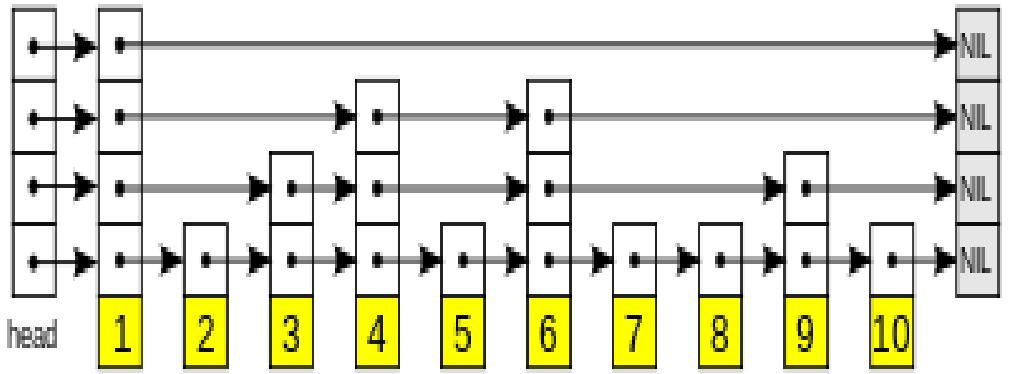
Skip List

- A skip list is a probabilistic data structure that allows O(logn) search complexity as well as O(logn) insertion complexity within an ordered sequence of n elements
- Thus it can get the best features of a sorted array (for searching) while maintaining a linked list-like structure that allows insertion, which is not possible in an array

https://en.wikipedia.org/wiki/Skip_list

Skip List

- Fast search is made possible by maintaining a linked hierarchy of sub sequences, with each successive subsequence skipping over fewer elements than the previous one
- Searching starts in the sparsest subsequence until two consecutive elements have been found, one smaller and one larger than or equal to the element searched for
- Via the linked hierarchy, these two elements link to elements of the next sparsest subsequence, where searching is continued until finally we are searching in the full sequence



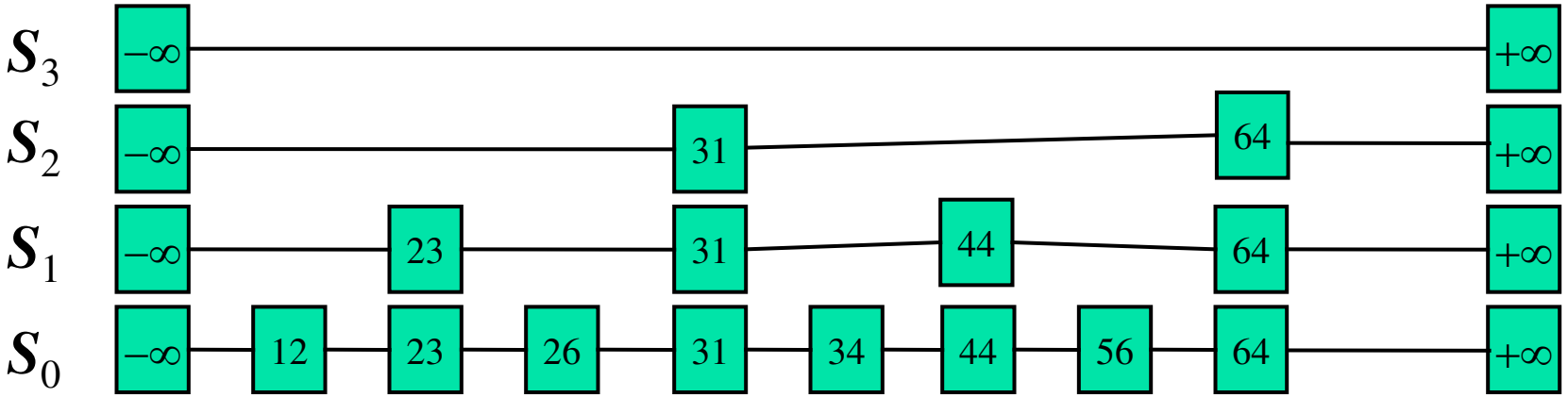
https://en.wikipedia.org/wiki/Skip_list

Perfect Skip List

- A skip list is a collection of lists at different levels
- The lowest level (0) is a sorted, singly linked list of all nodes
- The first level (1) links alternate nodes
- The second level (2) links every fourth node
- In general, level i links every 2^ith node
- In total, $\lceil \log_2 n \rceil$ levels (i.e. $O(\log_2 n)$ levels)
- Each level has half the nodes of the one below it

Perfect Skip List

- Example of a Perfect Skip List



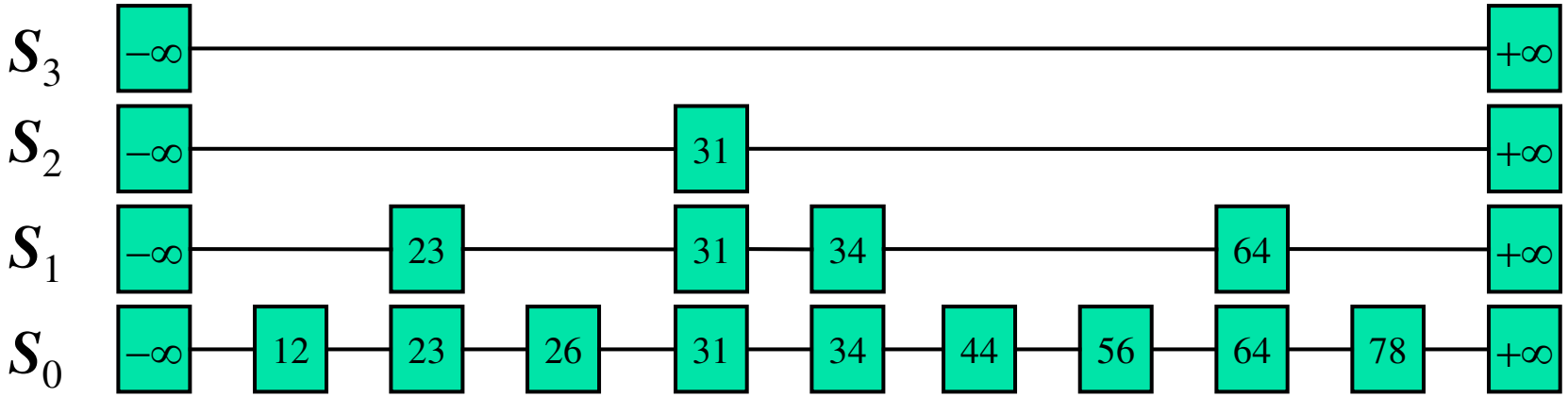
Insertions/Deletions

When we add a new node, our beautifully precise structure might become invalid

- We may have to change the level of every node
- One option is to move all the elements around
- But it takes O(n) time, which is back where we began
- Is it possible to achieve a net gain?

Randomized Skip List

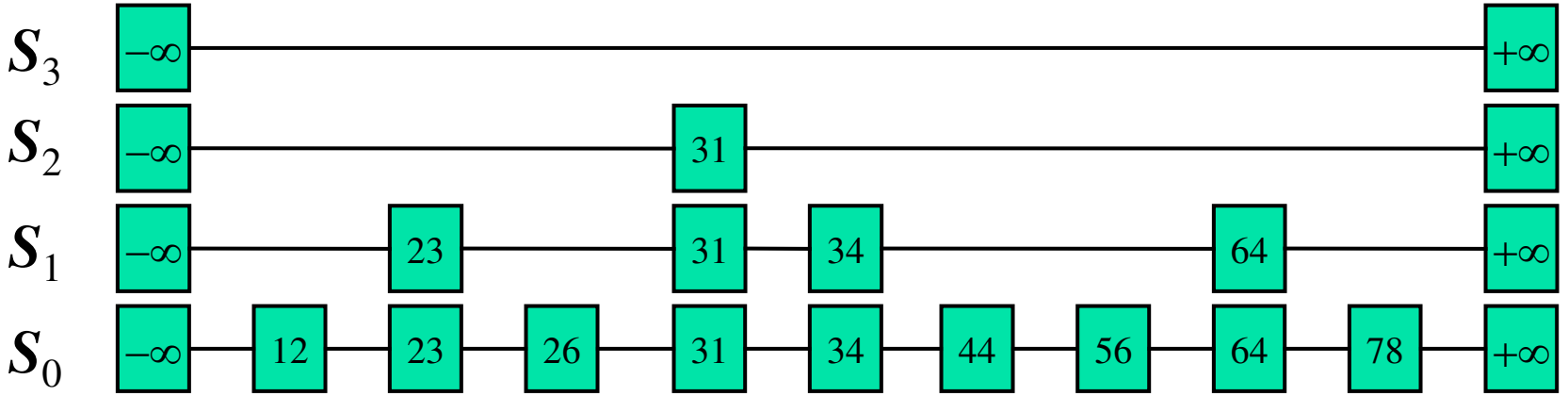
Example of a Randomized Skip List



Skip List

Skip List definition

- A skip list for a set S of distinct (key, element) items is a series of lists $S_0, S_1, \ldots, S_h$ such that:
  - Each list S_i contains the special keys $+\infty$ and $-\infty$
  - List S_0 contains the keys of S in non decreasing order
  - Each list is a subsequence of the previous one, i.e.,
    $$S_0 \supseteq S_1 \supseteq \ldots \supseteq S_h$$
  - List S_h contains only the two special keys

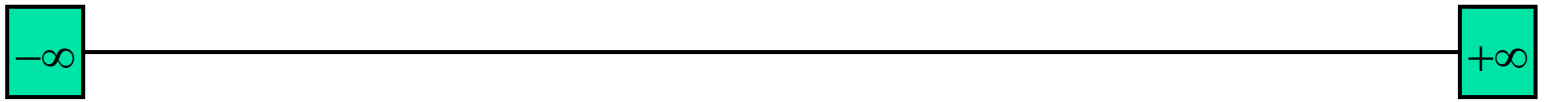


Skip List

Initialization

A new list is initialized as follows:

- A node NIL ($+\infty$) is created and its key is set to a value greater than the greatest key that could possibly used in the list
- Another node NIL ($-\infty$) is created, value set to lowest key that could be used



DATA STRUCTURES & ITS APPLICATIONS

References

- Presentation Slides: Advanced Data Structures
  Dr. RamaMoorthy Srinath, Professor, CSE, PESU

Note: Apart from the links already mentioned on Slides

DATA STRUCTURES & ITS APPLICATIONS

UNIT 1: Skip List

Kusuma K V
Department of Computer Science & Engineering

Skip List

Search

We search for a key x in a skip list as follows:

- We start at the first position of the top list
- At the current position p, we compare x with y i.e., key(after(p))

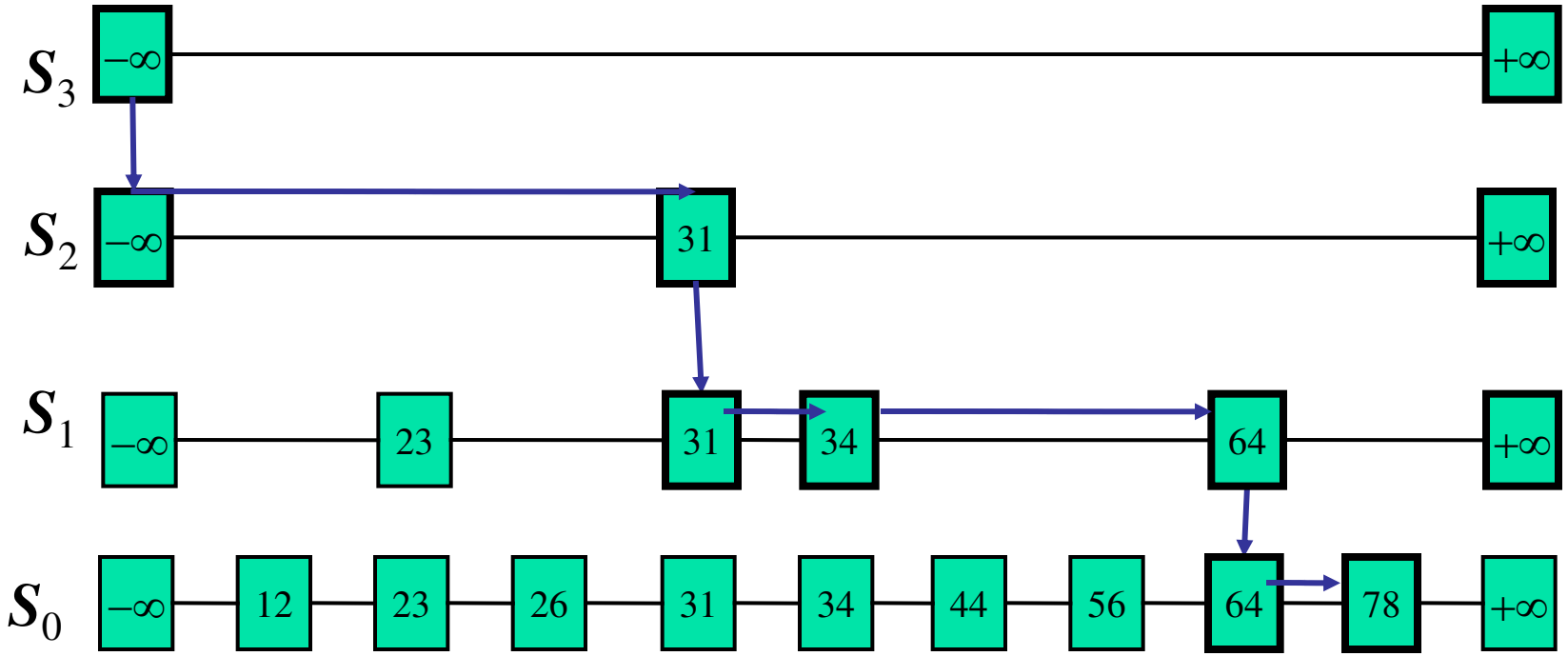
  x = y: we return element(after(p))

  x > y: we “scan forward”

  x < y: we “drop down”
- If we try to drop down past the bottom list, we return NO_SUCH_KEY
- Example: search for 78

Skip List

Searching in Skip List Example



Skip List

Insertion

- The insertion algorithm for skip lists uses randomization to decide how many references to the new item should be added to the skip list
- We then insert the new item in this bottom-level list at its appropriate position. After inserting the new item at this level we “flip a coin”
  - If the flip comes up tails, then we stop right there.
  - If the flip comes up heads, we move to next higher level and insert in this level at the appropriate position

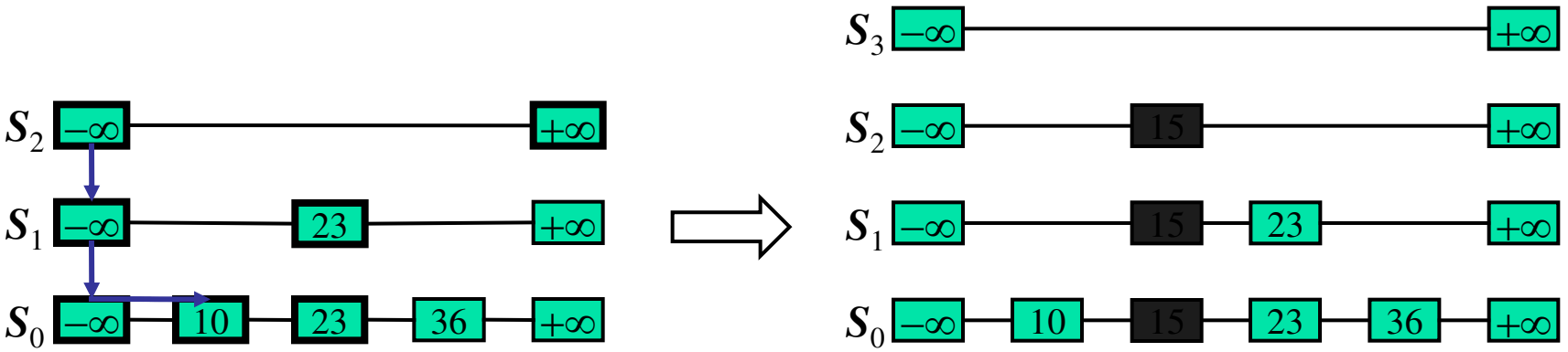
Randomized Algorithms

- A randomized algorithm performs coin tosses (i.e., uses random bits) to control its execution
- Its running time depends on the outcomes of the coin tosses
- We analyze the expected running time of a randomized algorithm under the following assumptions
  - the coins are unbiased, and
  - the coin tosses are independent
- The worst-case running time of a randomized algorithm is large but has very low probability (e.g., it occurs when all the coin tosses give “heads”)

Skip List

Insertion in Skip List Example

- Suppose we want to insert 15
- Do a search, and find the spot between 10 and 23
- Suppose the coin comes up “head” two times



Skip List

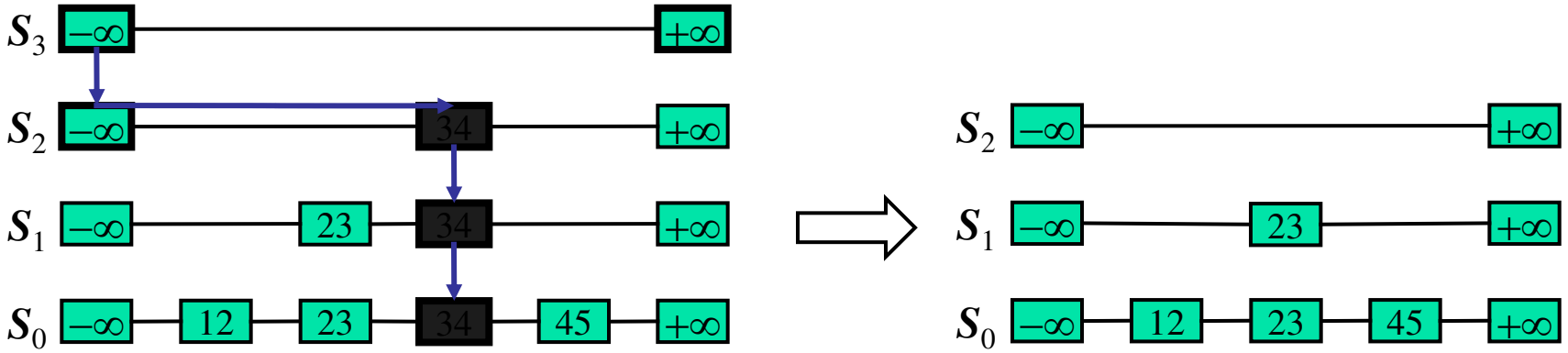
Deletion

- We begin by performing a search for the given key k.
- If a position p with key k is not found, then we return the NO_SUCH_KEY element
- Otherwise, if a position p with key k is found (it would be found on the bottom level), then we remove all the position above p
- If more than one upper level is empty, remove it

Skip List

Deletion in Skip List Example

- Suppose we want to delete 34
- Do a search, find the spot between 23 and 45
- Remove all the position above p



Skip List

Starting with an empty skip list, insert these keys, with these "randomly generated" levels

- 5, with level 1
- 26, with level 1
- 25, with level 4
- 6, with level 3
- 21, with level 0
- 3, with level 2
- 22, with level 2

Note that the ordering of the keys in the skip list is determined by the value of the keys only; the levels of the nodes are determined by the random number generator only

Skip List

5 with level 1, 26 with level 1, 25 with level 4, 6 with level 3
21 with level 0, 3 with level 2, 22 with level 2

DATA STRUCTURES & ITS APPLICATIONS

References

- Presentation Slides: Advanced Data Structures
  Dr. RamaMoorthy Srinath, Professor, CSE, PESU

Note: Apart from the links already mentioned on Slides

DATA STRUCTURES AND ITS APPLICATIONS

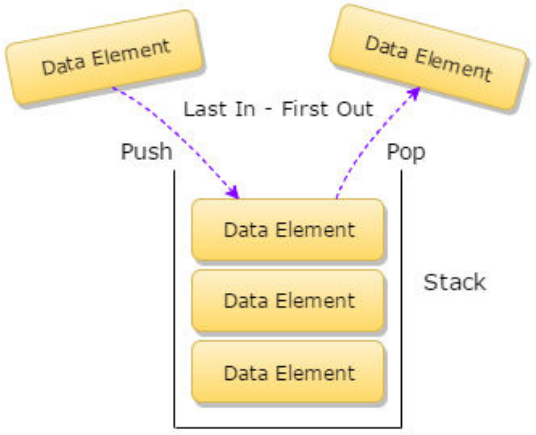
Stacks

Dinesh Singh
Department of Computer Science & Engineering

Data Structures and its Applications

Stacks - Definition

- A Stack is a data Structure in which all the insertions and deletions of entries are at one end. This end is called the TOP of the stack.
- When an item is added to a stack it is called push into the stack
- When an item is removed it is called pop from the stack.
- The Last item pushed onto a stack is always the first that will be popped from the stack.
- This property is called the *last in, first out* or LIFO for short



Stacks – Representation in C

A stack in C is declared as a structure containing two objects :

- An array to hold the elements of the stack
- An Integer to indicate the position of the current stack top within the array
- Stack of integers can be done by the following declaration

```
#define STACKSIZE 100
struct stack
{
  int top;
  int items[STACKSIZE]
};
```

Once this is done, actual stack can be declared by

```
struct stack s;
```

Stacks – Representation in C

Items need not be restricted to integers, items can be of any type.

A stack can contain items of different types by using C unions.

```
#define STACKSIZE 100
#define INT 1
#define FLOAT 2
#define STRING 3
struct stackelement {
    int etype;
    union{
        int ival;
        float fval;
        char *pavl; //pointer to string
    } element;
};
```

Stacks – Representation in C

```
struct stack
{
  int top;
  struct stackelement items[STACKSIZE];
};
```

- The above declaration defines a stack whose items can either be integers, floating point numbers or string depending on the value of etype (previous slide).

Data Structures and its Applications

Stacks – Implementation of operations of stack

Operations on stack

- Inserting an element on to the stack : push
- Deleting an element from the stack : pop
- Checking the top element : peep
- Checking if the stack is empty : empty
- Checking if the stack is full : overflow

Representation of stack will be as follows

```
#define STACKSIZE 100
struct stack
{
  int top;
  int items[STACKSIZE]
};
```

Stacks – Implementation of operations of stack

```
void push(struct stack *ps, int x)
/*ps is pointer to the structure representing stack, x is integer to be inserted
top is integer that indicates the position of the current stack top within the
      array, items is an integer array that represents stack, STACK_SIZE is the
      maximum size of the stack */
{
   if (ps->top == STACKSIZE -1) //check if the stack is full
         printf("STACK FULL Cannot insert..");
  else
   {
      ++(ps->top); //increment top
      ps->items[ps->top]=x; //insert the element at a location top
    }
}
```

Stacks – Implementation of operations of stack

```
int pop(struct stack *ps )
/*ps is pointer to the structure representing stack, top is integer that indicates
      the position of the current stack top within the array , items is an integer
      array that represents stack, STACK_SIZE is the maximum size of the stack */
  {
   if (ps->top == -1) // check if the stack is the empty
          printf("STACK EMPTY Cannot DELETE..");
  else
   {
      x=ps->items[ps->top]; //delete the element
      --(ps->top); //decrement top
     return x;
    }
}
```

Stacks – Implementation of operations of stack

```
int display(struct stack *ps )
/*ps is pointer to the structure representing stack, top is integer that indicates
      the position of the current stack top within the array , items is an integer
      array that represents stack, STACK_SIZE is the maximum size of the stack */
  {
   if (ps->top == -1) // check if the stack is the empty
          printf("STACK EMPTY ");
  else
   {
      for (i=ps->top;i>=0;i--) // displays the elements from top
         printf("%d",ps->items[i]);
 }
}
```

Stacks – Implementation of operations of stack

```
int peep(struct stack *ps )
{
   if (ps->top == -1)
          printf("STACK EMPTY ..");
  else
   {
      x=ps->items[ps->top]; //get the element
      return x;
    }
}
```

Stacks – Implementation of operations of stack

```
int empty(struct stack *ps )
{
    if (ps->top == -1)
            return 1;
   return 0;
}
int overflow(struct stack *ps)
{
    if (ps->top==STACKSIZE-1)
      return 1;
  return 0;
}
```

Stacks – Implementation of operations of stack

implementation of stack operations where the items array and top are separate variables (Not part of structure)

```
void push(int *s, int *top, int x)
{
  if(*top==STACKSIZE-1)//check if the stack is full
  {
    printf("stack overflow..cannot insert");
   return 0;
 }
 else
  {
    ++*top; //increment the top
    s[*top]=x; //insert the element
  }
 return 1;
}
```

Stacks – Implementation of operations of stack

- Implementation of stack operations where the items and the top are separate variables (Not part of structure)

```
int pop(int *s, int *top)
{
  if(*top==-1)//check if the stack is empty
  {
    printf("Stack empty .. Cannot delete");
   return -1;
 }
 else
  {
    x=s[*top]; //insert the element
    --*top; //decrement top
   return x; // return the deleted element
  }
}
```

Stacks – Implementation of operations of stack

- Implementation of stack operations where the items and the top are separate variables (Not part of structure)

```
display(int *s, int *top)
{
   if(*top==-1)
    printf("Empty stack");
 else
  {
   for(i=*top;i>=0;i--) // display the elements from the top
     printf("%d",s[i]);
  }
}
```

Stacks – Application of Stack

- Write an algorithm to determine if an input character string is of the form x C y where x is a string consisting of the letters ‘A’ and ‘B’ and where y is the reverse of x. At each point you may read only the character of the string

```
 int check(t)
//the function returns 1 if string t is of the form x C y, else returns 0
//uses stack s and its operations push and pop
 {
    i=0;
    while(t[i]!=‘C’) //push all the characters of the string into the stack
until C  is encountered
    {
      push(&s, t[i]);
      i=i+1;
    }
```

```
      //pop the contents of the stack and compare with t[i] until the end of  the string
while(t[i]!='\0')
{
   x=pop(&s);
   if(t[i]!=x) //if the character popped out is not equal to the character read from the string
      return 0; // not of the form xCy
   i=i+1;
 }
 return 1; // string of the form
 }
```

DATA STRUCTURES AND ITS APPLICATIONS

Stacks – Linked List Implementation

Dinesh Singh
Department of Computer Science & Engineering

Data Structures and its Applications

Stacks – linked list implementation

- A stack can be easily implemented through the linked list. In the Implementation by a linked list, the stack contains a top pointer. which is “head” of the stack. The pushing and popping of items happens at the head of the list.

Structure of the stack

```
struct node
{
   int data;
   struct node *next;
 }
struct node *top
top=NULL;
```

Data Structures and its Applications

Stacks – linked list implementation

Note :

- Items of the stack represented as the linked list.
- Each item is a node
- Top is a pointer that points to the first node (top of the stack)
- Top is initially NULL (Empty stack)
- Insertion and deletion happens at the front of the list
- stack size is not limited.

Operations on the stack

- Push : Inserting an element at the front of the list
- Pop : delete an element from the front of the list
- Display : displaying the list

Stacks – linked list implementation

```
//implements the push operation
void push(int x, struct node **top)
   {
        struct node *temp;
        temp=(struct node*)malloc(sizeof(struct node));
        temp->data=x;
        temp->next=*top; // insert in front of the list
        *top=temp; // make top points to the new top node
    }
```

Stacks – linked list implementation

```
//implements the pop operation
//returns top element, -1 if stack is empty
int pop(struct node **top)
    {
        int x;
       struct node *q;

       if(*top==NULL)
        {
          printf("Empty Stack\n");
           return -1;
         }
```

```
//implements the pop operation

else
        {
          q=*top;
           x=q->data; // get the top element
          *top=q->next; // make top point to the next top
          free(q); // free the memory of the node
           return(x);
         }
    }
```

Stacks – linked list implementation

```
//implements the display  operation
void display(struct node *top)
    {
       if(top==NULL)
         printf("Empty Stack\n");
       else
        {
         while(top!=NULL)
         {
           printf("%d->“,top->data);
           top=top->next;
         }
        }
    }
```

Stacks – linked list implementation

Another representation of structure of stack

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

struct stack s;
s.top=NULL;
```

Stacks – linked list implementation

```
//implements the push operation
void push(int x, struct stack * s)
//s is pointer to structure stack
   {
        struct node *temp;
        temp=(struct node*)malloc(sizeof(struct node));
        temp->data=x;
        temp->next=s->top; // insert in front of the list
        s->top=temp; // make top points to the new top node
    }
```

Stacks – linked list implementation

```
//implements the pop operation
//returns top element, -1 if stack is empty
int pop(struct stack *s)
    {
        int x;
       struct node *q;

       if(s->top==NULL)
        {
          printf("Empty Stack\n");
           return -1;
         }
```


```
//implements the pop operation

else
        {
          q=s->top;
           x=q->data; // get the top element
          s->top=q->next; // make top point to the next top
          free(q); // free the memory of the node
           return(x);
         }
    }
```

Stacks – linked list implementation

```
//implements the display  operation
void display(struct stack *s)
   {
      struct node *q;
      if(s->top==NULL)
        printf("Empty Stack\n");
      else
       {
        q=s->top;
        while(q!=NULL)
        {
          printf("%d->“,q->data);
          q=q->next;
         }
       }
   }
```

Stacks – Application

Write an Algorithm to print a string in the reverse order

```
//prints the text in a reverse order
reverse(t)
{
 i=0;
 //push all the characters on to the stack
 while(t[i]!='\0')
 {
    push(s,t[i]);
    i=i+1;
 }
```

Stacks – Application

pop all the characters from the stack until the stack is empty

```
while(!empty(s))
 {
    x= pop(s);
    print(x)
   }
}
```

DATA STRUCTURES AND ITS APPLICATIONS

Application of stacks– Functions, nested functions and Recursion

Dinesh Singh
Department of Computer Science & Engineering

Application of stacks – Functions, nested functions

Activation record :

- When functions are called, The system (or the program) must remember the place where the call was made, so that it can return there after the function is complete.
- It must also remember all the local variables, processor registers, and the like, so that information will not be lost while the function is working.
- This information is considered as large data structure . This structure is sometimes called the invocation record or the activation record for the function call.

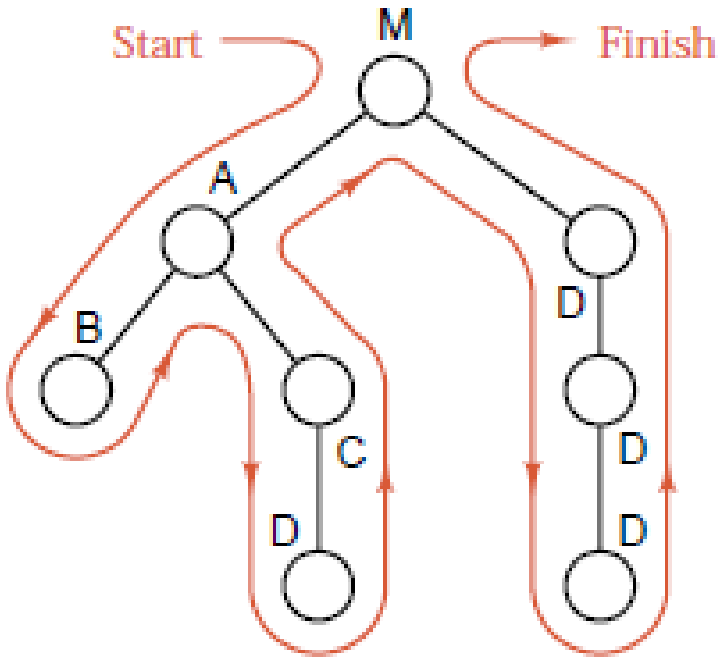
Application of stacks – Functions, nested functions and Recursion

- Suppose that there are three functions called A,B and C. M is the main function.
- Suppose that A invokes B and B invokes C. Then B will not have finished its work until C has finished and returned. Similarly, A is the first to start work, but it is the last to be finished, not until sometime after B has finished and returned.
- Thus the sequence by which function activity proceeds is summed up as the property last in, first out.
- The machine's task of assigning temporary storage area (activation records) used by functions would be in same order (LIFO).
- Since LIFO property is used, the machine allocates these records in the stack
- Hence a stack plays a key role in invoking functions in a computer system.

Application of stacks – Functions, nested functions

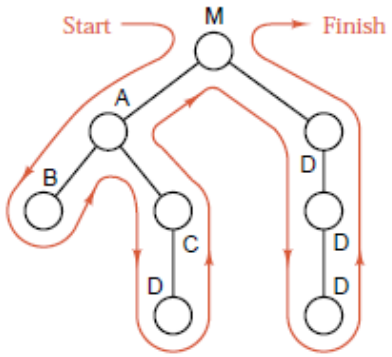
Example :

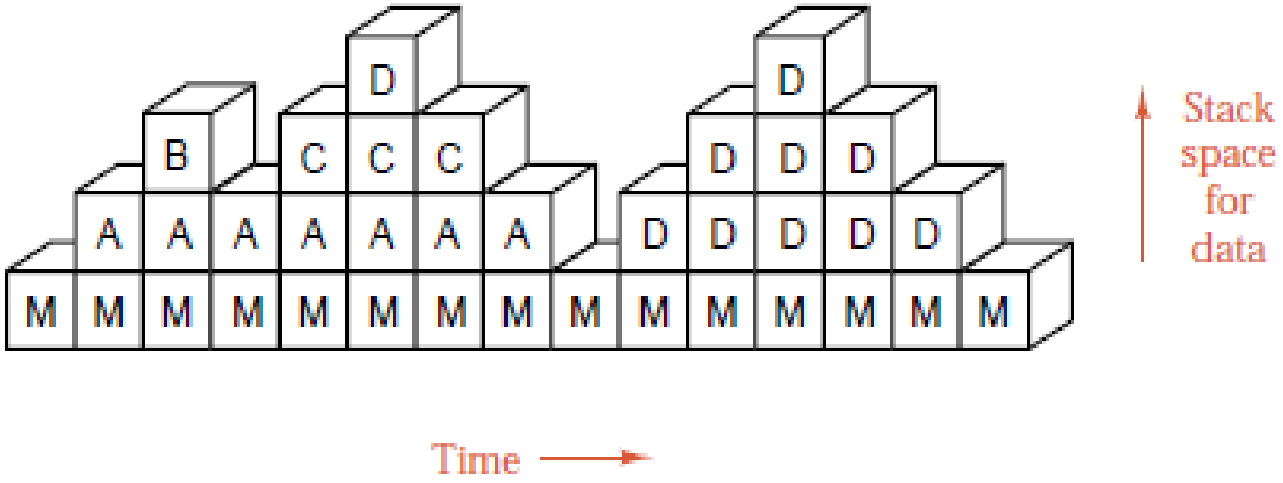
Consider the following showing the order in which the functions are invoked



Application of stacks – Functions, nested functions

The Sequence of stack frames of the activation records of the function calls is given below. Each vertical column shows the contents of the stack at a given time, and changes to the stack are portrayed by reading through the frames from left to right.





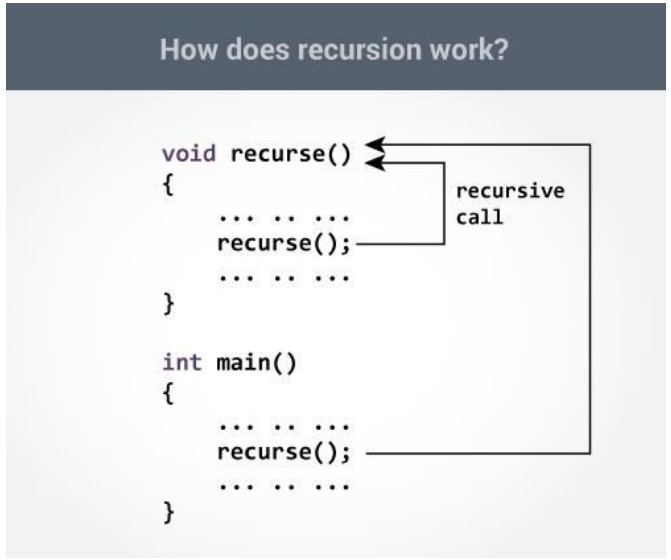
Data Structures and its Applications

Application of stacks – Recursion

Recursion :

- Recursion is a computer programming technique involving the use of a procedure, subroutine or a function.

- The process in which a function calls itself directly or indirectly is called recursion and the corresponding function is called as recursive function.



Application of stacks – Recursion

How a particular problem is solved using recursion?

- The idea is to represent a problem in terms of one or more smaller problems.

- A way is found to solve these smaller problems and then build up a solution to the entire problem.

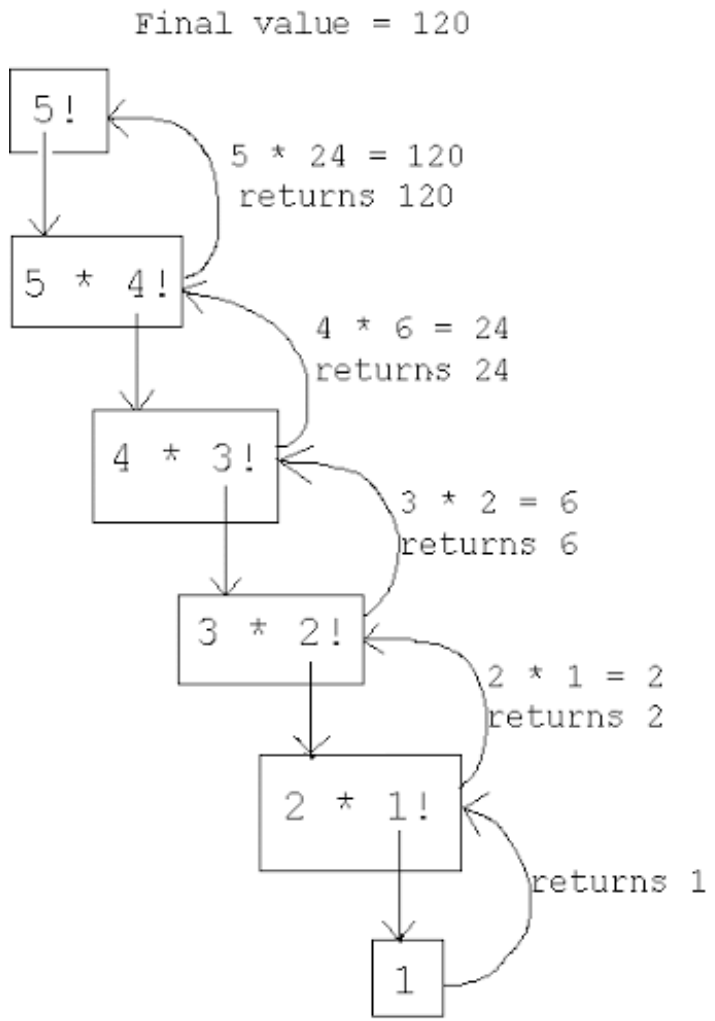
- The sub problems are in turn broken down into smaller sub problems until the solution to the smallest sub problem is known.

- The solution to the smallest sub problem is called the base case.

- In the recursive program, the solution to thebase case is provided

Application of stacks – Recursion

Example 1: Factorial of a Number n



Recursive definition :

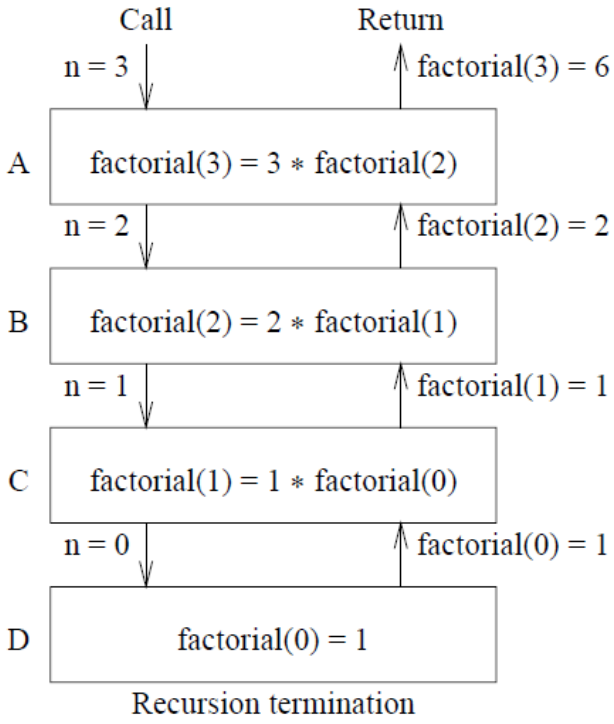
n! = 1 if n=1

n!= n * (n-1)! If n>1

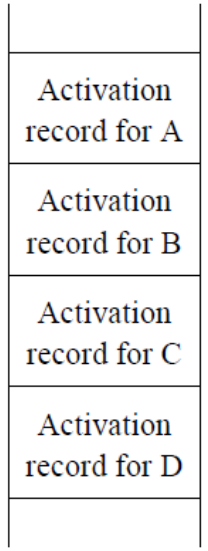
Application of stacks – Recursion

Recursive function to compute factorial of n

```
factorial(n)
{
  int f;
  if(n==0)
       return 1;
   f=n*factorial(n-1);
return f;
}
```



(a)



(b)

Application of stacks – Recursion

Recursive function to find the product of a*b

a*b= a if b=1;

a*b = a*(b-1)+a if b >1;

To evaluate 6 * 3

6*3 = 6*2 + 6 = 6*1 + 6 + 6 = 6 + 6 + 6 = 18

```
multiply(int a, int b)
{
   int p;
    if (b==1)
     return a
   p= multiply(a,b-1) + a;
  return p;
}
```

Application of stacks – Recursion

Fibonacci Sequence

The fibonacci sequence is the sequence of integers
1, 1, 2, 3, 5, 8, 13, 21, 34...

Each element is the sum of two preceding elements

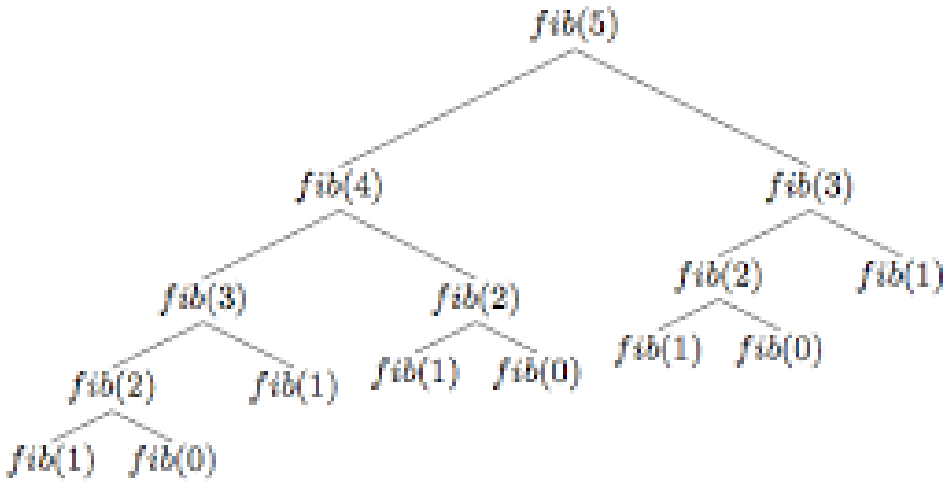
If we consider fib(0) = 1 and fib(1)=1
then recursive definition to compute the nth element in the sequence is

fib(n) = n if n=0 or n=1
fib(n) = fib(n-1) + fib(n-2) for n >=2

Application of stacks – Recursion

Recursive function to compute the nth element in the Fibonacci Sequence

```
 fib(int n)
{
   int x,y;
   if ( (n==0) || (n==1
     return n;
  x= fib(n-1) ;
  y=fib(n-2);
 return x+y;
}
```



Application of stacks – Recursion

Recursive function to find the sum of elements of an array

```
int sum(int *a, int n)
//a is pointer to the array, n is the index of the last element of the array
{
  int s;
  if(n==0) // base condition
   return a[0];
  s= sum(a,n-1) + a[n]; // compute sum of n-1 elements and add the nth element
  return s;
}
```

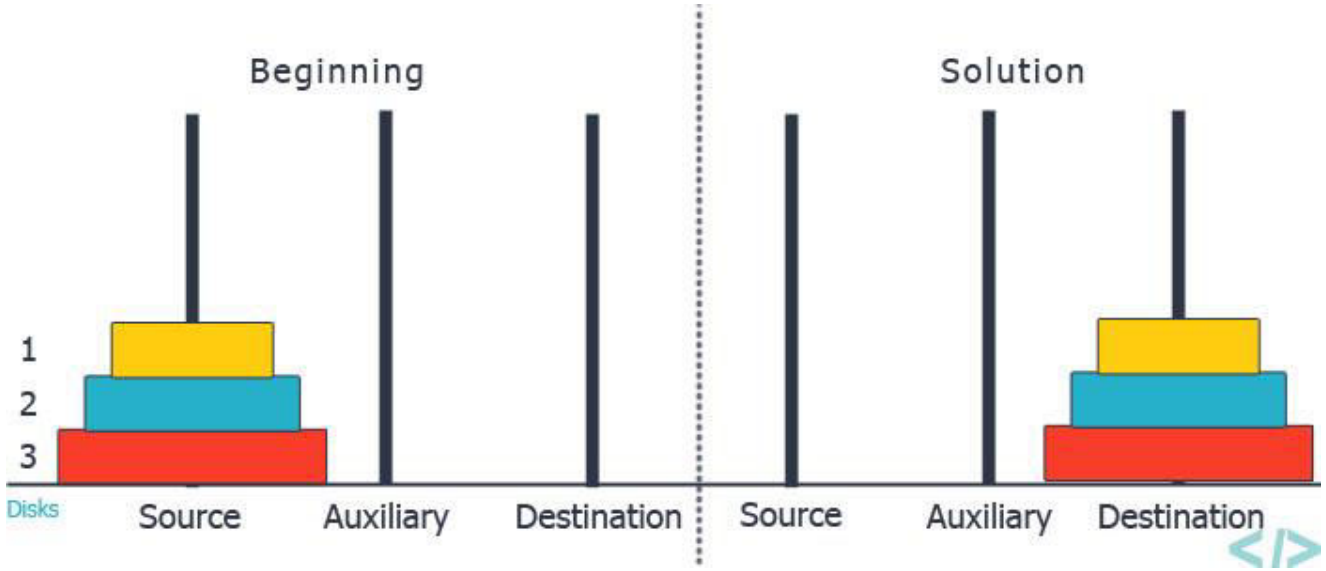
Application of stacks – Recursion

Recursive function to display the elements of the linked list in the reverse order

```
int display(struct node *p)
{
  if(p->next!=NULL)
     display(p->next)
  printf("%d ",p->data);
}
```

Tower of Hanoi

Three Pegs A, B and C exists. N disks of differing diameters are placed on peg A. The Larger disk is always below a smaller disk. The objective is to move the N disks from Peg A to Peg C using Peg B as the auxillary peg



Application of stacks – Recursion

Tower of Hanoi – recursive solution

If a solution to n-1 disks is found, then the problem would be solved. Because in the trivial case of one disk, the solution would be to move the single disk from Peg A to Peg C.

To move n disks from A to C , the recursive solution would be as follows

1. If n=1 move the single disk from A to C and stop
2. Move the top n-1 disks from A to B using C as auxillary
3. Move the remaining disk from A to C
4. Move n-1 disks from B to C using A as the auxillary

Application of stacks – Recursion

Recursive function for Tower of Hanoi

```
void tower(int n,char src,char tmp,char dst)
 {
  if(n==1)
  {
   printf("\nMove disk %d from %c to %c",n,src,dst);
   return;
  }
  tower(n-1,src,dst,tmp);
  printf("\nMove disk %d form %c to %c",n,src,dst);
  tower(n-1,tmp,src,dst);
  return;
 }
```

Application of stacks – Recursion

Recursive function for Tower of Hanoi

Solution for 4 disks

Move disk 1 from peg A to peg B
Move disk 2 from peg A to peg C
Move disk 1 from peg B to peg C
Move disk 3 from peg A to peg B
Move disk 1 from peg C to peg A
Move disk 2 from peg C to peg B
Move disk 1 from peg A to peg B
Move disk 4 from peg A to peg C
Move disk 1 from peg B to peg C
Move disk 2 from peg B to peg A
Move disk 1 from peg C to peg A
Move disk 3 from peg B to peg C
Move disk 1 from peg A to peg B
Move disk 2 from peg A to peg C
Move disk 1 from peg B to peg C

Solution for moving 3 disks from A to B
Move 4^{th} disk from A to C
Solution for moving 3 disks from B to C

DATA STRUCTURES AND ITS APPLICATIONS

Infix to Postfix and Prefix Expressions – Implementation

Dinesh Singh
Department of Computer Science & Engineering

Infix , Postfix and Prefix Expressions

- Consider the sum of A and B expressed as A + B
- This representation is called infix.
- There are two alternate notations for expressing sum of A and B using the symbols A , B and + , these are
  - Prefix : + A B
  - Postfix : A B +
- The prefixes “Pre” , “post” and “in” refers to the relative position of the operator with respect to the two operands.
- In prefix, operators precedes the two operands
- In postfix, the operator follows the two operands
- In infix, the operator is in between the two operands

Infix , Postfix and Prefix Expressions

Conversion of Infix to Postfix

- For Example : consider the expression A + B * C
- The evaluation of the above expression requires the knowledge of precedence of operators
- A + B * C can be expressed as A + (B * C) as multiplication takes precedence over addition
- Applying the rules of precedence the above infix expression can be converted to postfix as follows

A + B * C = A + (B * C)

= A + (BC *) convert the multiplication

= A (B C *) + convert the addition

= A B C * +

Infix , Postfix and Prefix Expressions

Conversion of Infix to Prefix

- For Example : consider the expression A + B * C
- Applying the rules of precedence the above infix expression can be converted to prefix as follows

A + B * C = A + (B * C)

= A + (* B C) convert the multiplication

= + A (* B C) + convert the addition

= + A * B C

Infix , Postfix and Prefix Expressions

Applying the rules of precedence the table shows the conversion of Infix to Postfix and Prefix Expression

Infix	Postfix	Prefix
A + B * C + D	A B C * + D +	+ + A * B C D
(A + B) * (C + D)	A B + C D + *	* + A B + C D
A * B + C * D	A B * C D * +	+ * A B * C D
A + B + C + D	A B + C + D +	+ + + A B C D
A $ B * C – D + E / F / (G+H)	A B $ C * D – E F / G H + / +	+ - * $ A B C D / /E F + G H
((A + B)*C – (D – E)) $ (F + G)	A B + C * D E – - F G + $	$ - * + A B C – D E + F G
A – B / (C * D $ E)	A B C D E $ * / -	- A / B * C $ D E

$ is the exponentiation operator and its precedence is from right to left

For example A $ B $ C = A $ (B $ C)

Infix to postfix conversion - algorithm

```
opstk is the empty stack
while(not end of input)
{
  symb = next input character
// if the input symbol is an operand , add it to the postfix string
 if(symb is an operand)
    add symb to postfix string
else
   {
     // pop the contents of the stack while the precedence of
    // top of the stack is greater than the precedence of the symbol scanned
    //prcd function returns true in this case
      while(!empty(opstk ) and ( prcd(stacktop(opstk),symb))
       {
          topsymb=pop(opstk)
         add topsymb to postfix string
       }
```

Infix to postfix conversion - algorithm

```
   /* if prcd function returns false, then
if  the stack is empty and symbol is not ‘)’, push symb on to the stack*/
        if( empty(opstk) || symb!=‘)’)
            push(opstk,symb)
      else
          topsymb=pop(opstk); // if symb is ‘)’ ,  pop ‘ ( ‘ from the stack
}
while(!empty(opstk)
 {
   topsymb=pop(opstk)
   add topsymb to postfix string
 }
}
```

Infix to postfix conversion - algorithm

- prcd(OP1,OP2) is a function that compares the precedence of the top of the stack(OP1) and the input symbol (OP2) and returns TRUE if the precedence is greater else returns FALSE.
- Some of the return values of prcd function
  - prcd(‘ * ‘ , ‘ + ‘) returns TRUE : prcd(‘ * ’, ’ / ’) returns TRUE
  - prcd(‘ + ‘, ‘ * ‘) returns FALSE : prcd(‘ / ’, ’ * ’) returns TRUE
  - prcd(‘ + ‘ , ‘ + ‘) returns TRUE :
  - prcd(‘ + ‘ , ‘ – ‘) returns TRUE
  - prcd(‘ – ‘ , ‘ – ‘) returns TRUE
  - prcd(‘ $ ‘ , ‘$’) returns FALSE : exponentiation operator, associatively from right to left
  - prcd(‘ (‘, op) , prcd(op ‘ (‘) returns FALSE for any operator
  - prcd (op , ‘)‘) returns TRUE for any operator
  - Prcd(‘ (‘ , ‘) ‘) returns FALSE

Infix to postfix conversion - algorithm

- The motivation behind the conversion algorithm is the desire to output the operators in the order in which they are to be executed.
- If an incoming operator is of greater precedence than the one on top of the stack, this new operator is pushed on to the stack.
- If on the other hand , the precedence of the new operator is less than the one on the top of the stack, the operator on the top of the stack should be executed first.
- Therefore the top of the stack is popped out and added to the postfix and the incoming symbol is compared with the new top and so on.
- Parenthesis in the input string override the order of operations
- When a left parenthesis is scanned, it is pushed on to the stack
- When its associated right parenthesis is found, all the operators between the two parenthesis are placed on the output string. Because they are to be executed before any operators appearing after the parenthesis

Infix to postfix conversion – Trace of the algorithm

Trace of the algorithm for A + B * C

symb	postfix string	opstk	Remarks
A	A	Empty	symb is operand , add 'A'on to the postfix string
+	A	+	symb is operator, and stack empty, push + on to the stack
B	AB	+	symb is operand , Add 'B' to the postfix string
*	AB	+*	symb is operator, precedence of + (stack top) is < than precedence of *, therefore push * on to the stack

Infix to postfix conversion – Trace of the algorithm

Trace of the algorithm for A + B * C

symb	postfix string	opstk	Remarks
C	ABC	+ *	symb is operand , add C on to the postfix string
End of input	ABC*	+	Pop * from the stack and add to the postfix string
-	ABC*+	Empty	Pop + from the stack and add to the postfix string

Data Structures and its Applications

Infix to postfix conversion – Trace of the algorithm

PES UNIVERSITY ONLINE

Trace of the algorithm for (A + B)* C

symb	postfix string	opstk	Remarks
(	-	Empty	Stack empty, push '(' on to the stack
A	A	(	Symb is an operand, add 'A' to the post fix string
+	A	(+	Precedence of top of the stack '(' is less than '+' , push '+' on to the stack
B	AB	(+	Symb is an operand, add B to the postfix string
)	AB+	(	Precedence of stack top '+' is > than ')' Pop '+' from the stack and add to the postfix string
	AB+	empty	Precedence of stack top stack top '(' is not greater than ')' and symb is ')', therefore pop '(' from the stack

Infix to postfix conversion – Trace of the algorithm

Trace of the algorithm for (A + B)* C

symb	postfix string	opstk	Remarks
*	AB+	*	Stack empty, push '*' on to the stack
C	AB+C	*	Symb is an operand, add 'C' to the postfix string
End of string	AB+C*	Empty	End of input, pop * from the stack and add to the postfix.

Infix to posfix conversion – another algorithm

```
void convert_postfix(char *infix,char*postfix)
 {
    i=0;
   char ch;
   j=0;
   push(s,&top,'#');
   while(infix[i]!='\0')
   {
     ch=infix[i];
   //while the prcedence of top of stack is greater than the
//precedence of the input symbol, pop and add to the postfix
    while(stack_prec(peep(s,top))>input_prec(ch))
        postfix[j++]=pop(s,&top);
```

Infix to posfix conversion – another algorithm

```
if(input_prec(ch)!=stack_prec(peep(s,top)))
        push(s,&top,ch);
      else
        pop(s,&top);
    i++;
   }
   while(peep(s,top)!='#')
     postfix[j++]=pop(s,&top);
   postfix[j]='\0';
 }
```

Infix to posfix conversion – another algorithm

Precedence table

Operator	Input Precedence	Stack Precedence
+ , -	1	2
* , /	3	4
$	6	5
Operands	7	8
)	0	- : Never pushed on to stack
(	9	0
#	-	-1

Infix to prefix conversion – algorithm

Example 1:
Input : A * B + C / D
Output : + * A B/ C D

Example 2 :
Input : (A - B/C) * (D/K-L)
Output : *-A/BC-/DKL

1: Reverse the infix expression i.e A+B*C will become C*B+A.
Note while reversing each '(' will become ')' and each ')' becomes '('.

2: Obtain the postfix expression of the modified expression i.e CB*A+.

3: Reverse the postfix expression. Hence in our example prefix is +A*BC.

DATA STRUCTURES AND ITS APPLICATIONS

Evaluation of Postfix expression and Parenthesis matching

Dinesh Singh
Department of Computer Science & Engineering

Evaluation of Postfix Expression - Algorithm

- Each operator in a postfix string refers to the previous two operands.
- Each time an operand is read , it is pushed on to the stack
- When an operator is reached, its operands will be the top two elements on to the stack.
- The two elements are popped out, the indicated operation is performed on them and result is pushed on the stack so that it will be available for use as an operand of the next operator.

Evaluation of Postfix Expression - Algorithm

```
opndstk is the empty stack
while(not end of the input) // scan the input string
{
 symb=next input character;
 if (symb is an operand)
   push(opndstk,symb)
else
   {
    opnd2=pop(opndstk);
    opnd1=pop(opndstk);
    value = result of applying symb to opnd1 and opn2;
   push(opndstk, value);
  }
 return(pop(opndstk));
}
```

Evaluation of Postfix Expression – Trace of the Algorithm

Infix : 3 + 5 * 4

Postfix expression : 3 5 4 * +

Symb	Opnd1	Opnd2	Value	opndstk
3	-			3
5				3, 5
4				3 , 5 , 4
*	5	4	20	3 , 20
+	3	20	60	60

Data Structures and its Applications

Evaluation of Postfix Expression - implementation

```
int postfix_eval(char* postfix)
 {
  int i,top,r;
  int s[10];//stack
  top=-1;
  i=0;

  while(postfix[i]!='\0')
  {
    char ch=postfix[i];
    if(isoper(ch))
    {
      int op1=pop(s,&top);
      int op2=pop(s,&top);
```

Evaluation of Postfix Expression - implementation

```
switch(ch)
      {
        case '+':r=op1+op2;
                 push(s,&top,r);
                 break;
        case '-':r=op2-op1;
                  push(s,&top,r);
                  break;
         case '*':r=op1*op2;
                 push(s,&top,r);
                  break;
        case '/':r=op2/op1;
                  push(s,&top,r);
                 break;
       }//end switch
    }//end if
```

Evaluation of Postfix Expression - implementation

```
else
      push(s,&top,ch-'0');//convert charcter to
integer and push
     i++;
    }  //end while
  return(pop(s,&top));
 }
```

Parenthesis Matching – overview of the Algorithm

Examples

1. (( )) : Valid Expression
2. ((( )) : Invalid Expression (Extra opening parenthesis)
3. (( ))) : Invalid Expression (Extra closing parenthesis)
4. ((}) : Invalid Expression (Parenthesis mismatch)
5. (( )] : Invalid Expression (Parenthesis mismatch)

Parenthesis Matching – overview of the Algorithm

1.Read the input symbol from the input expression
2. If the input symbol is one of the open parenthesis (‘(‘ , ‘ { ‘ or ‘ [‘), it is pushed on to the stack
3. If the input symbol is of closing parenthesis, stack is popped and the popped parenthesis is compared with the input symbol, if there is a mismatch in the type of the parenthesis, return 0 (Mismatch of parenthesis)
4. If there is a match in the parenthesis , the next input symbol is read.
5. If during this process, if the stack becomes empty, return 0 (Extra closing parenthesis)
6. If at the end of the expression, if the stack is not empty, return 0 (Extra opening parenthesis)
7. If at the end of the input expression, if the stack is empty, return 1. (Parenthesis are matching)

Data Structures and its Applications

Parenthesis Matching - Implementation

```
int match(char *expr)
 {
   int i,top;
   char s[10],ch,x;//stack
   i=0;
    top=-1;

   while(expr[i]!='\0')
   {
     ch=expr[i];
     switch(ch)
     {
        case '(':
        case '{':
        case '[':push(s,&top,ch);
                 break;
```

Parenthesis Matching

```
case ')':if(!isempty(top))
                {
                  x=pop(s,&top);
                  if(x=='(')
                    break;
                  else
                    return 0;//mismatch of parenthesis
                }
                else
                  return 0;//extra closing parenthesis
```

Parenthesis Matching

```
case '}':if(!isempty(top))
         {
          x=pop(s,&top);
          if(x=='{')
            break;
          else
            return 0;//mismatch of parenthesis
         }
         else
          return 0;//extra closing parenthesis
```

Data Structures and its Applications

Parenthesis Matching

```
case ']':if(!isempty(top))
                {
                  x=pop(s,&top);
                  if(x=='[')
                    break;
                  else
                    return 0;//mismatch of parenthesis
                }
                else
                  return 0;//extra closing parenthesis

          }//end switch
          i++;
      }//end while
     if(isempty(top))
        return 1;
      return 0;//extra opening parenthesis
  }
```

THANK YOU

Dinesh Singh
Department of Computer Science & Engineering

dineshs@pes.edu

+91 8088654402