DATA STRUCTURE

Elementary Data Organization

Data:- data can be defined as a representation of facts, concepts or instructions in a formalized manner suitable for communication, interpretation or processing by human or electronic machine.

Data is represented with the help of characters like alphabets(A-Z , a-z), digits(0-9) or special character(+,-,*,&,? Etc.) . Data may be a single value or it may be a set of values, must be organized in a particular fashion.

Data Item:- A set of characters which are used together to represent a specific data element e.g. name of a student in a class is represented by the data item NAME.

Types of data items:-

(i) Elementary data items:- these data items can not be further sub divided. For exp. SID

(ii) Group data items:- These data items can be further sub-divided into elementary data items. For example Date. Date may be divided into days, months and years.

Record:- record is a collection of related data items e.g. a payroll record for an employee contains data fields such as name, age, qualification, sex, basic pay, DA, HRA, PF etc.

Or a student record.

Name	Roll no	Class	Marks
Anu	4999	BCA	850

File:- File is a collection of logically related records e.g. a payroll file might consist of the employee pay records for a company.

Introduction
to
Data Structure

Definition

Data Structure is a representation of the logical relationship existing between individual elements of data.

Or

A Data structure is a way of organizing all data items that considers not only the elements stored but also their relationship to each other.

or

A data structure is a class of data that can characterized by its organization and the operations that are defined on it. Hence

Data Structure= Organized Data + Allowed Operations

Data structure mainly specifies the following four things:-

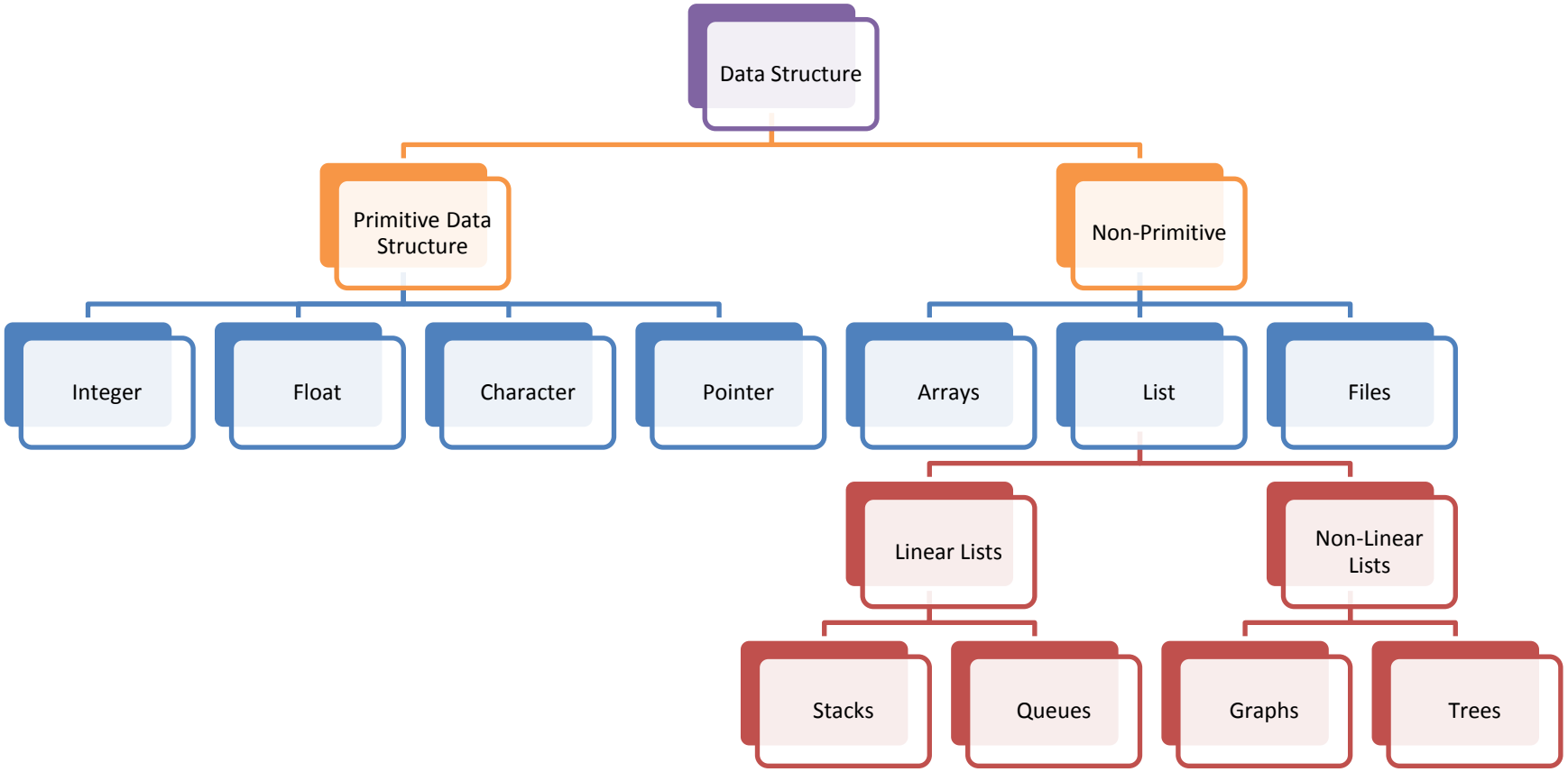
i) Organization of data

ii) Accessing methods

iii) Degree of associativity

iv) Processing alternatives for information

Classification of Data Structure



Primitive Data Structure **:-** The data structure that are atomic or indivisible are called primitive. Example are integer, real, float, Boolean and characters.

Non-Primitive data structure **:-** The data structure that are not atomic are called non primitive or composite. Examples are records, arrays and strings. These are more sophisticated. The non primitive data structures emphasize on structuring f a group of homogenous or heterogeneous data items.

B) Linear Data Structure :- In a linear data structure, the data items are arranged in a linear sequence. Example is array.

Non Linear data structure :- In a non-linear data structure, the data items are not in sequence. Example is tree.

C) Homogenous Data Structure :- In homogenous data structure, all the elements are of same type. Example is array.

Non Homogenous Data Structure :- in non homogenous structures, all the elements are may or may not be of the same types. Example is records.

D) Static Data Structure :- Static structures are ones whose sizes and structures, associated memory location are fixed at compile time.

Dynamic Data Structure :- Dynamic structures are ones which expand or shrink as required during the program execution and there associate memory location change.

Description of various Data Structures

Operations on data structure

- Create
- Selection
- Deletion
- Updation

other Operations

Searching

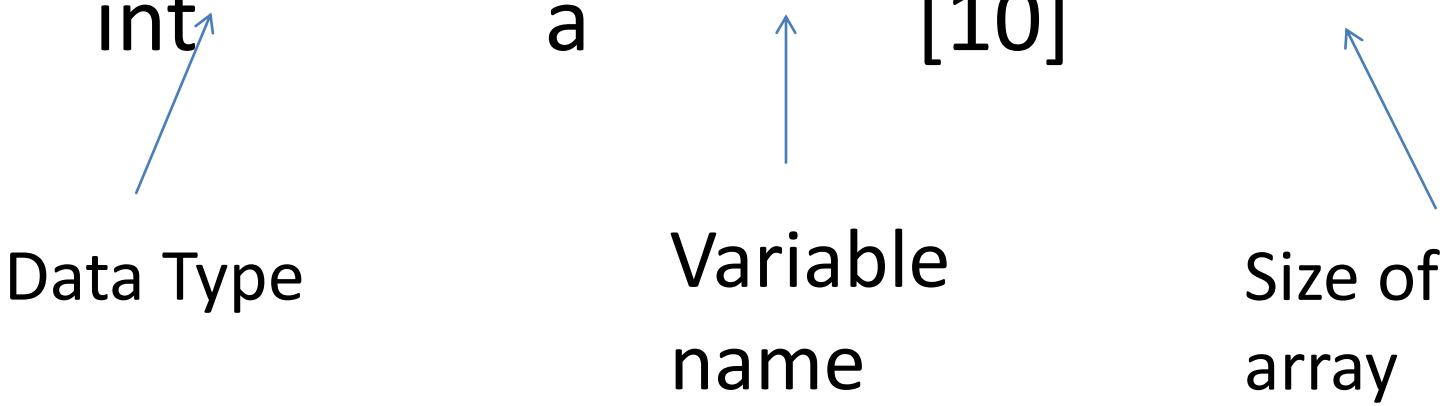
Sorting

Merging

Arrays:- An array is defined as a set of finite number of homogenous elements or data items. It means an array can contain one type of data only, either all integers, all floating point numbers or all characters.3

Exp:- int a[10]

- Array declaration –



Concepts of array:

- The individual element of an array can be accessed by specifying name of the array, followed by index or subscript inside square bracket.

  Exp. to access the 10^{th} element statement will be a[9].

- The first element of the array has index zero[0]. So in an array of 10 elements the first array will be a[0] and last a[9]
- The elements of array will always be stored in consecutive memory location.

- The size of an array can be calculate by this equation

  (upper bound- lower bound)+1

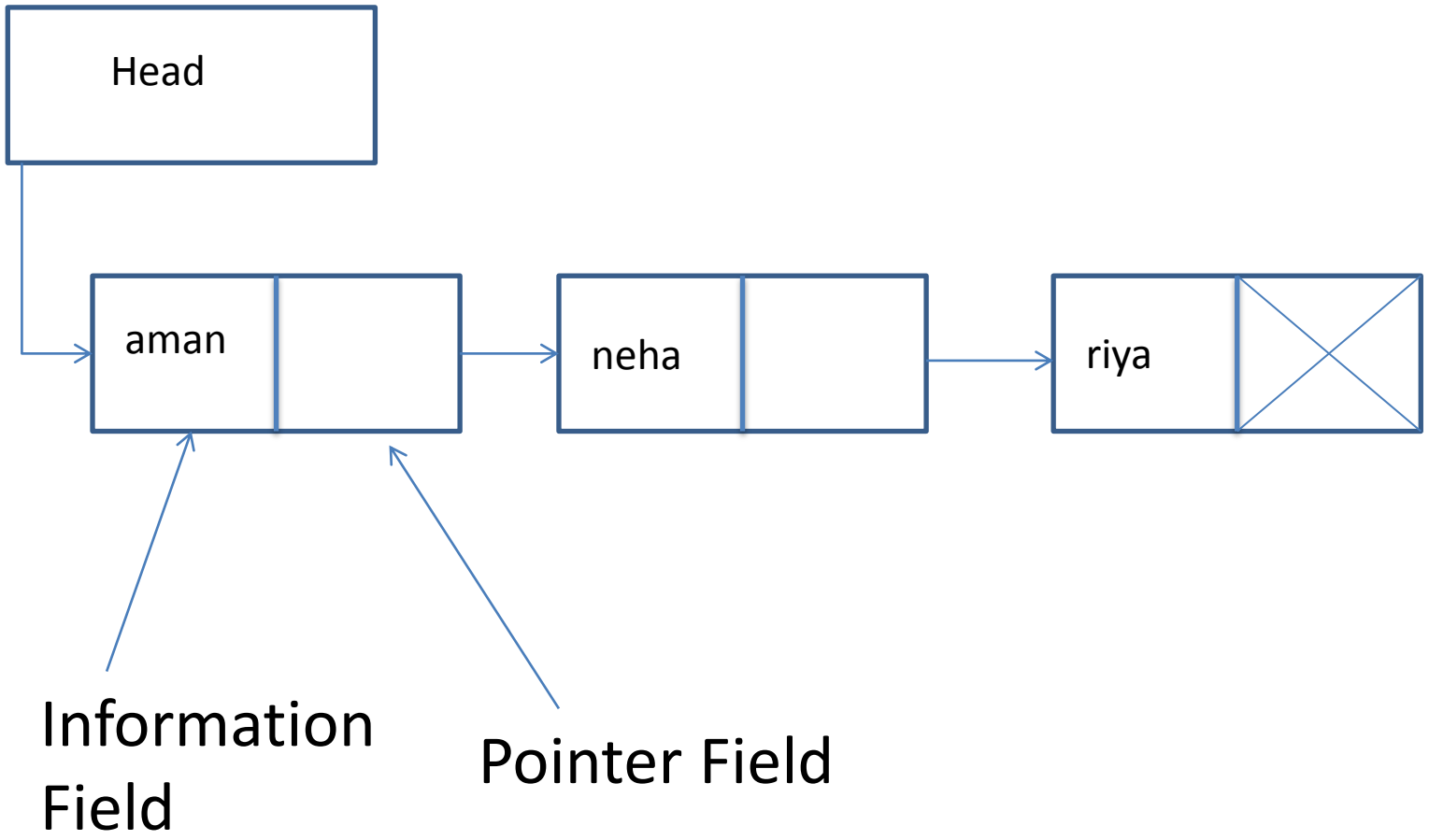
  (9-0)+1

- Arrays can always be read or written through loop. In case of one dimensional array it required one loop for reading and one loop for writing and in case of two dimensional array it requires two loops for each operation.

Operation performed on array

1. Creation of an array
2. Traversing of an array
3. Insertion of new elements
4. Deletion of required elements
5. Modification of an element
6. Merging of arrays

Lists:- A list can be defined as a collection of variable number of data items. List are the most commonly used non-primitive data structures. An element of list must contain at least two fields, one for storing data or information and other for storing address of next element.


Head
aman
neha
riya
Information Field
Pointer Field

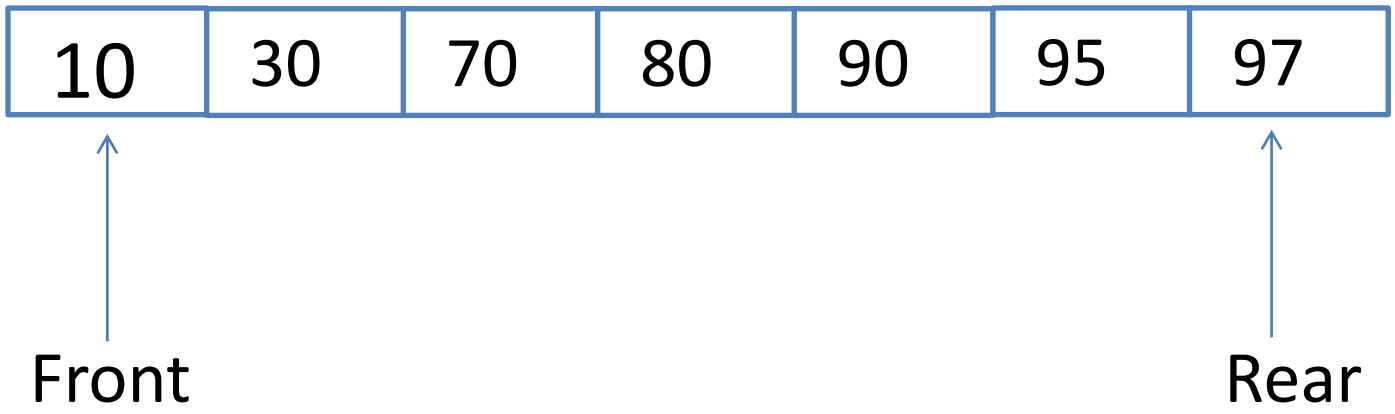
Stack :- A stack is also an ordered collection of elements like array but it has a special feature that deletion and insertion of elements can be done only from one end, called the top of the stack.

- It is a non primitive data structure.
- It is also called as last in first out type of data structure(LIFO).
- Exp. Train, stack of trays etc.
- At the time of insertion or removal of an element base of stack remains same.
- Insertion of element is called Push
- Deletion of element is called Pop

Stack implementation

1. Static implementation (using array)
2. Dynamic implementation(using pointer)

Queue :- Queues are first in first out type of data structures. In a queue new elements are added to the queue from one end called rear end and the elements are always removed from other end called the front end.

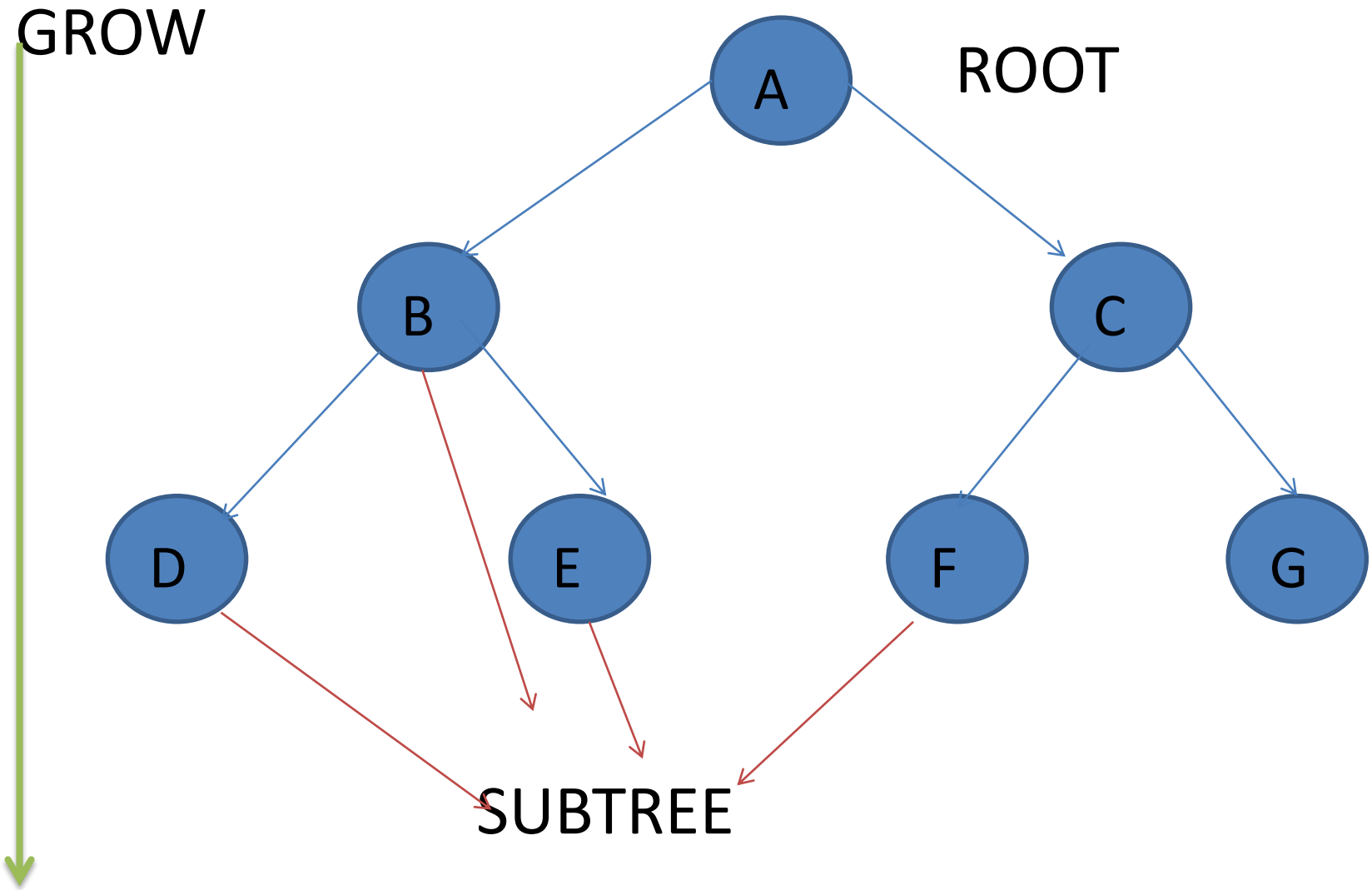

10
30
70
80
90
95
97
Front
Rear

- New elements are added to the queue from one end called REAR end
- Elements are always removed from other end called front end
- Exp. : queue of railway reservation.
- FIFO

Queue Implementation

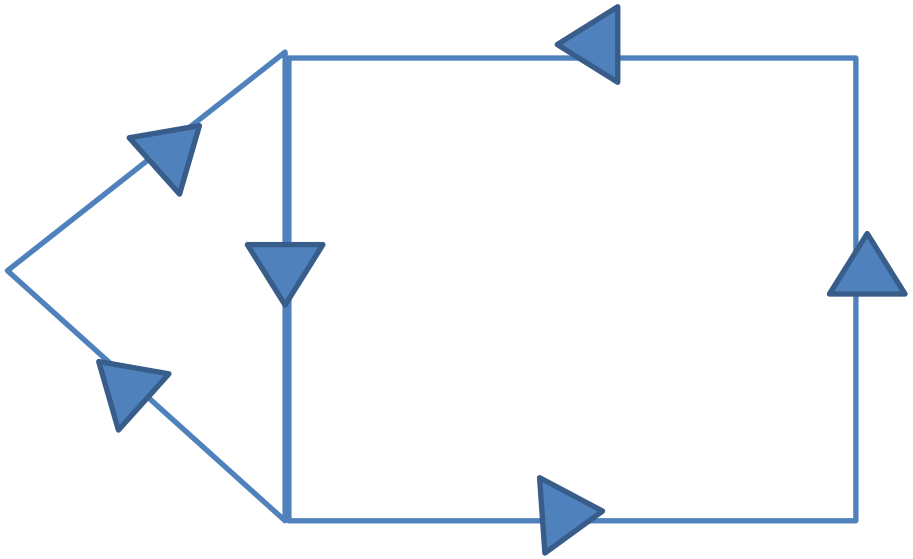
1. Static implementation (using array)
2. Dynamic implementation(using pointer)

Trees:- A tree can be defined as finite set of data items. Tree is non-linear type of data structure in which data items are arranged in a sorted sequence. Trees represent the hierarchical relationship between various elements.


GROW
A
ROOT
B
C
D
E
F
G
SUBTREE

1. There is a special data item at the top of hierarchy called the Root of the tree
2. The remaining data items called the subtree.
3. The tree always grows in length towards bottom in data structure.

Graphs :- Graph is a mathematical non-linear data structure capable of representing many kinds of physical structures. It has found applications in diverse fields like geography, chemistry and engineering sciences.



Types of Graphs

1. Directed Graph
2. Non-directed Graph
3. Connected Graph
4. Non-Connected Graph
5. Simple Graph
6. Multi Graph

Memory Allocation in C

A memory management technique determines how memory should be allocated to the variables declared in the program. There are two technique for memory management.

1. Compile time or Static memory allocation technique.
2. Run time or Dynamic memory allocation technique.

1. **Static memory allocation technique** :- Using this technique memory is reserved by the compiler for variables declared in the program.

- The required amount of memory is allocated to the program element(identifiers name which include variable name, function name, program name etc.) at the start of the program.

- Memory to be allocated to the variable is fixed and is determined by the compiler at the compile time.
- Exp:- if it is a single integer variable it allocates two bytes, array of 10 integer it allocates 20 bytes and for a float it is 4 bytes… . . . .

2. Dynamic memory allocation technique:- the concept of dynamic or run time memory allocation helps us to overcome this problem in arrays, as well as allows us to be able to get the required chunk of memory at run-time.

- Dynamic memory allocation gives flexibility for programmer.
- Efficient use of memory by allocating the required amount of memory whenever needed.

Dynamic allocation
and
De-allocation functions
in
C

1. malloc()
2. calloc()
3. free()
4. realloc()

malloc() Function

- The user should explicitly give the block size it requires for the use
- The malloc() function is like a request to the RAM of the system to allocate memory
- If the request is granted returns a pointer to the first block of that memory

- Type of the pointer it returns is void, it means we can assign it any type of pointer
- if malloc function fails to allocate the required amount of memory, it returns a NULL
- Malloc() function is available in header file<malloc.h>

malloc(number of elements*size of each element)

exp.

Int ptr*

Ptr = malloc(10*sizeof(int));

Type cast

Int ptr*

Ptr = (int*)malloc(10*sizeof(int))

Memory allocation to data structure

```
Struct student
{
int rrn;
Char name[20];
Float per;
};
Struct student *st_ptr;
```

Memory allocation

```
st_ptr = (struct student*) malloc (sizeof(struct student));
```

When this statement is executed a contiguous block of memory of size

Int = 2 bytes

Char = 20 bytes

Float = 4 bytes

Total = 26 bytes

To check the availability of memory

```
Int *ptr;
Ptr = (int *) malloc(5*sizeof(int));
If(ptr==null)
{
Printf("the required amount of memory is not available");
getch();

}
```

Calloc() Function

- Works exactly similar to malloc function
- It needs two arguments.
- First one is number of elements and second is size of element.
- No need of sizeof().

- After allocating the memory the address of the first block is assign to the pointer variable.
- Memory allocated by malloc() function contain garbage value, while memory allocated by calloc function contain all zeros.
- Calloc() function is availble in <stdlib.h> or <alloc.h> in turbo c.

Int *ptr;

Ptr = (int*) calloc(10,2)

Here 10 specify the number of elememnts and two specify the size of each element

Free() Function

- This function is used to de-allocate the previously allocated memory using malloc() or calloc() functions.

- Syntax is:

  free(pointer variable)

- Pointer variable is that variable in which the address of the allocated memory block is assigned. Free() function is used to return the allocated memory to the system RAM.

Realloc() Function

- This function is used to resize the size of memory block, which is already allocated.
- or u can say it is used to modify the size of already allocated memory block.
- You can use this function after alloc() function.

I. If the allocated memory block is insufficient for current application.

II. If the allocated memory is much more than what is required by the current application.

III. It provides more precise and efficient utilization of memory.

Syntax:

Ptr_var =realloc(ptr_var,new_size);

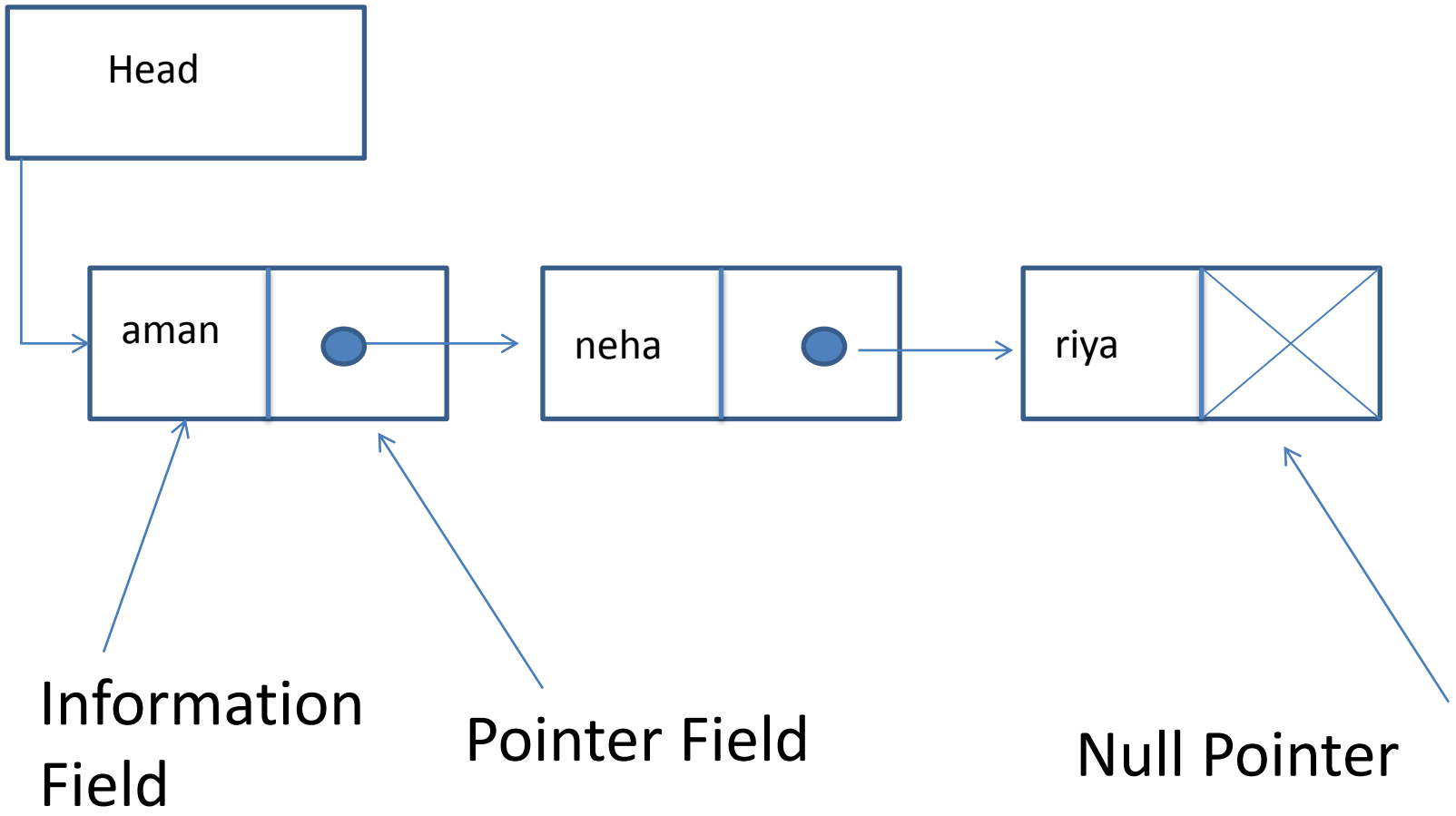
Ptr-var is the pointer holding the starting address of already allocated memory block new size is the size in bytes you want the system allocate now.

- New size may be larger or smaller.

Linked list

Linked Lists:- A list can be defined as a collection of variable number of data items. List are the most commonly used non-primitive data structures.

A linked list consisting of elements called nodes where each node is composed of two parts : information part and a link part or a pointer part.


Head
aman
neha
riya
Information Field
Pointer Field
Null Pointer

- Advantages:

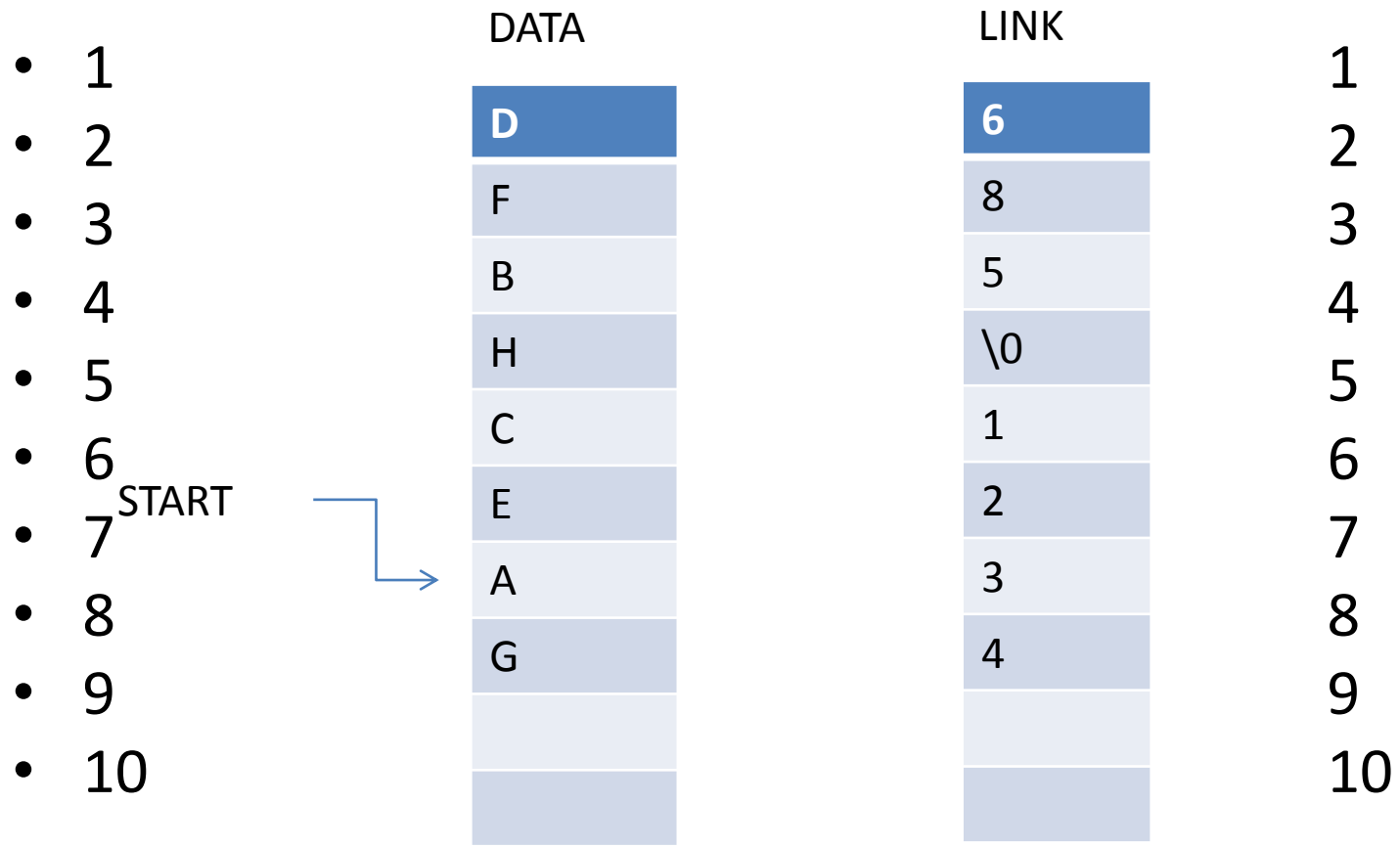
1. Linked lists are dynamic data structure. They can grow or shrink during the execution of a program
2. Efficient memory utilization . Memory is not pre allocated. Memory is allocated whenever it is required and it is deallocated when it is no longer required.
3. Insertion and deletions are easier and efficient. Linked lists provide flexibility in inserting a data item at a specified position and deletion of data from the given position.

- Disadvantages:

1. More memory : if the number of fields are more, then more memory space is needed.
2. Access to an arbitrary data item is little cumbersome and also time consuming.

- Key Terms :

1. Data Field
2. Link Field
3. Null Pointer
4. External Pointer(use to access entire list)
5. Empty List (external pointer = 0)


DATA
LINK
1
2
3
4
5
6
7
8
9
10
START
D
F
B
H
C
E
A
G
6
8
5
\0
1
2
3
4
1
2
3
4
5
6
7
8
9
10

- DATA [7] = ‘A’ link [7] = 3
- DATA [3] = ‘B’ link [3] = 5
- DATA [5] = ‘C’ link [5] = 1
- DATA [1] = ‘D’ link [1] = 6
- DATA [6] = ‘E’ link [6] = 2
- DATA [2] = ‘F’ link [2] = 8
- DATA [8] = ‘G’ link [8] = 4
- DATA [4] = ‘H’ link [7] = \0 OR NULL

Operations on linked list

1. Creation
2. Insertion
3. Deletion
4. Traversing
5. Searching
6. Concatenation
7. Display

Types of linked list

1. Singular Linked List
2. Doubly Linked List
3. Circular Linked List
4. Circular doubly Linked List

Singular linked list

- It is a dynamic data structure.
- It may grow or shrink.
- Linked list can be created using structures,, pointers and dynamic memory allocation function malloc.
- We consider head as an external pointer. This helps in creating and accessing other nodes in the linked list.

Structure definition and head creation.

- Struct node
- {
- Int num;
- Struct node *ptr; /* pointer to node*/
- };
- Typedef struct node NODE; /* type definition making it abstract data type*/
- NODE *start; /*pointer to the node of linked list*/
- Start = (node*)malloc(sizeof (NODE));/*dynamic memory allocation*/

- Exp:-
- Struct node
- {
- int num;
- Struct node *ptr;
- };
- Void create()
- {
- Typedef struct node NODE;
- NODE *start, *first, *temp;
- Int count = 0;
- Char choice;
- First = null;
- Do
- {
- Start = (NODE*)malloc (sizeof(NODE));
- Printf(“enter the data item\n”);
- Scanf(“%d”, &start->num);
- }

```
if(first !=null)
{
temp->ptr = start;
temp = start;
}
else
{
first = temp = start;
}
fflush(stdin);
Printf(“do you want to continue(type y or n)?\n”);
Scanf(“%c”, &choice);
}
While((choice ==‘y’)||(choice ==‘Y’));
}
```

Inserting nodes

- To insert an element into the following three things could be done:

1. Allocating a node
2. Assigning the data
3. Adjusting the pointers

- Instances for inserting a new node

1. Insertion at the beginning of the list.
2. Insertion at the end of the list.
3. Insertion at the specified position within the list.

Instances for inserting a new node

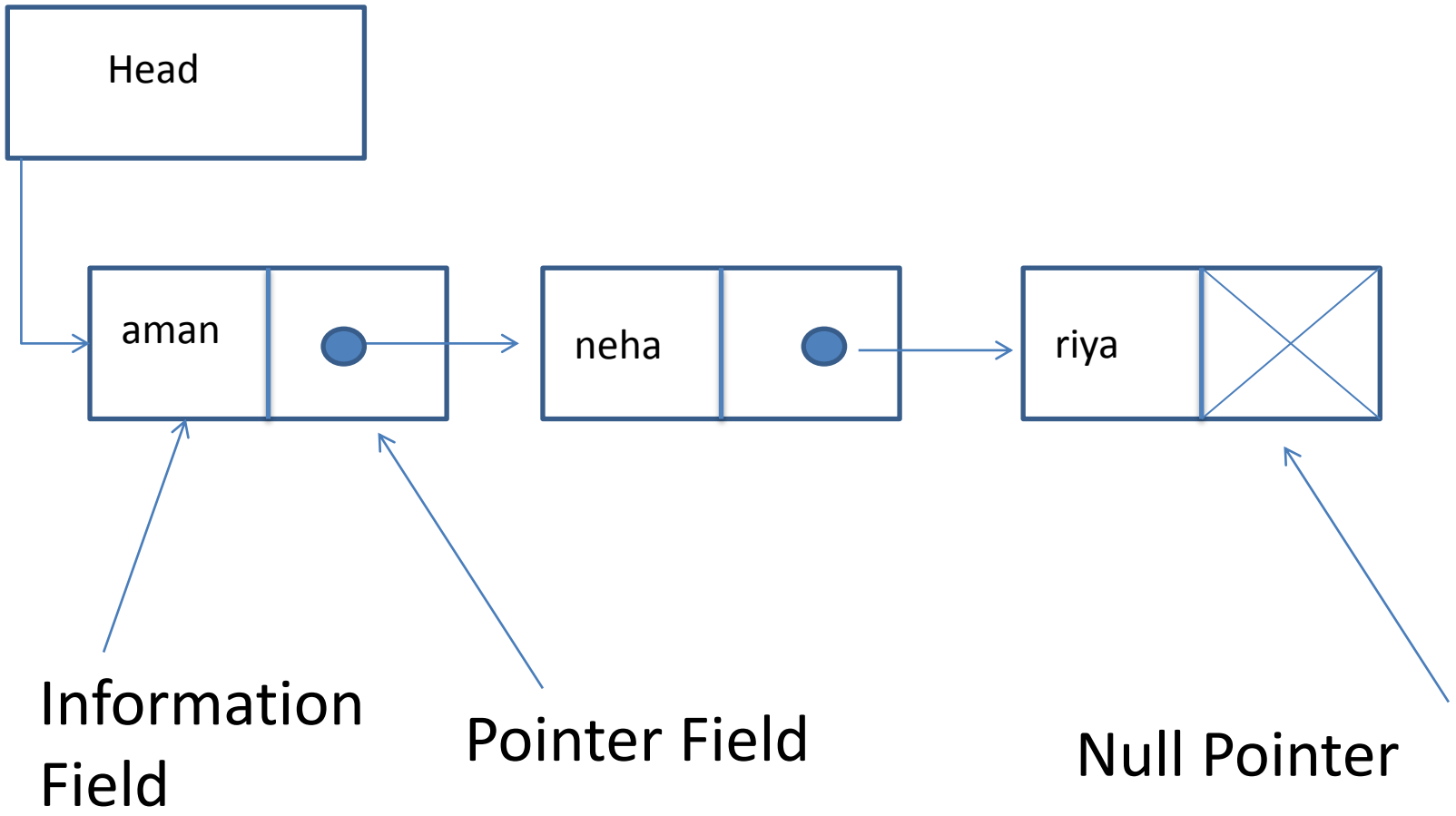
- To insert an element into the follwing three things could be done:

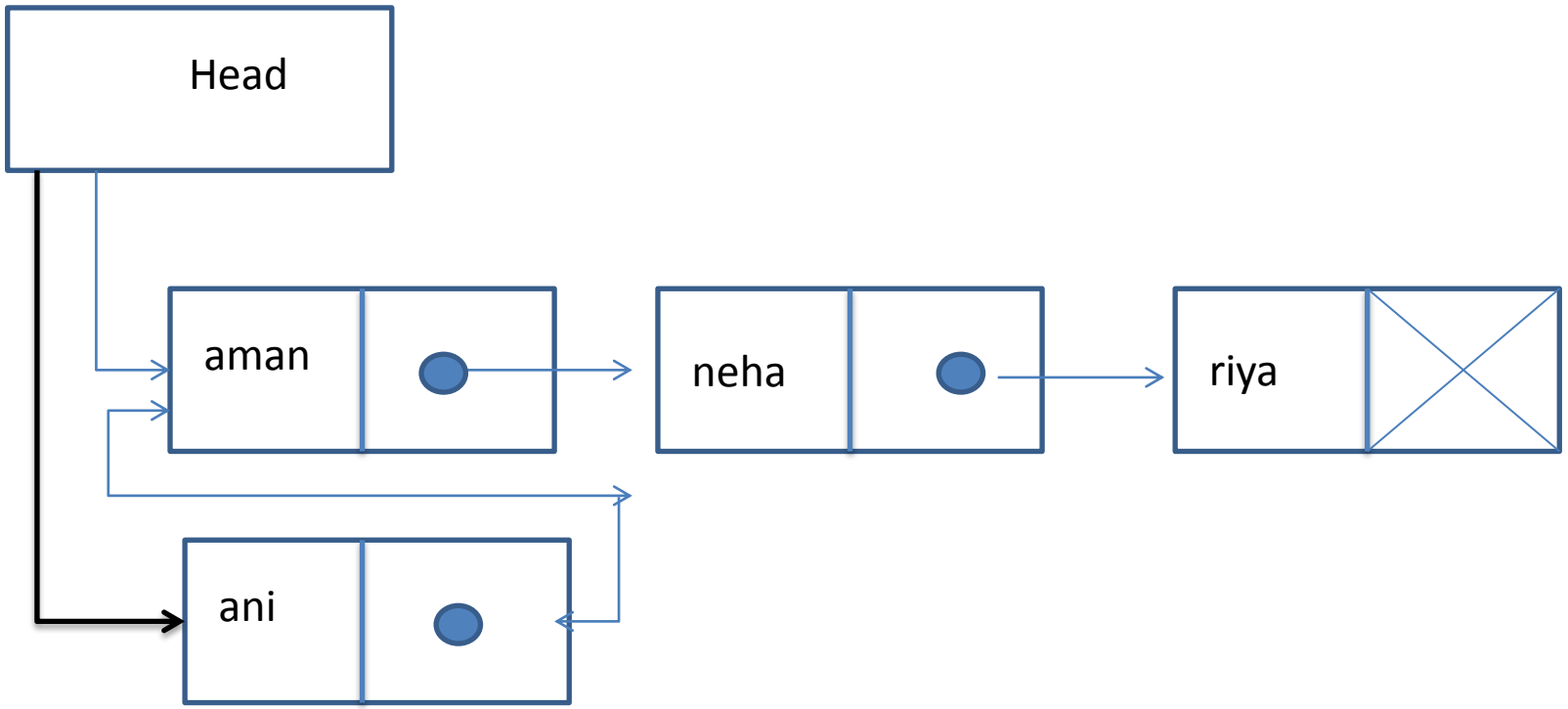
1. Insertion at the beginning of the list.
2. Insertion at the end of the list.
3. Insertion at the specified position within the list.

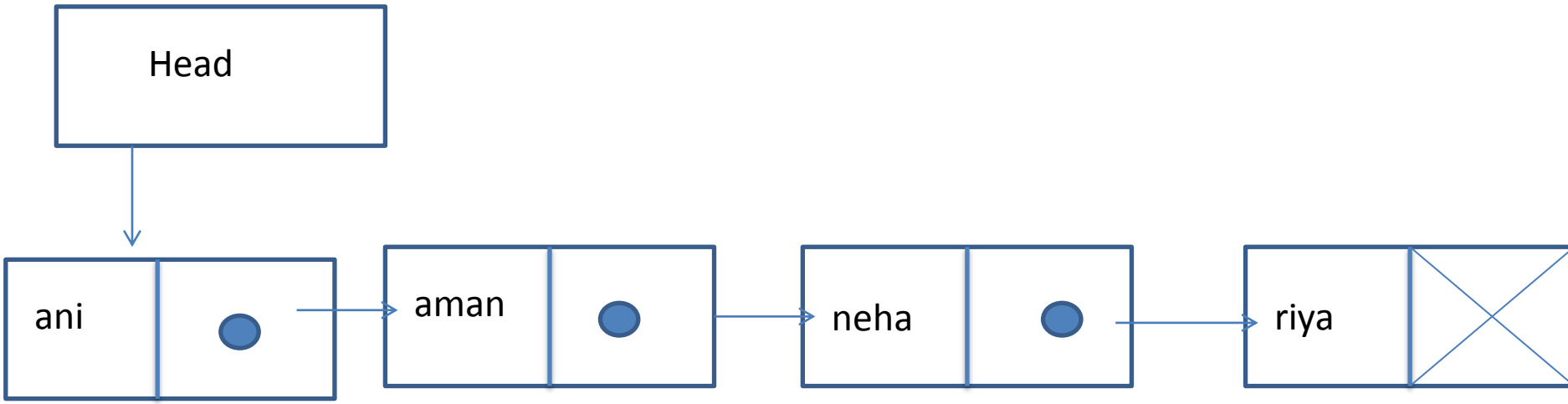
Steps for deciding the position of insertion

1. If the linked list is empty or the node to be inserted appears before the starting node then insert thet node at the beginning of the linked list.
2. If the node to be inserted appears after the last node in the linked list then insert that node at the end of the linked list.
3. If the above two conditions do not hold true then insert the new at the specified position within the linked list.

Inserting a node at the beginning


Head
aman
neha
riya
Information Field
Pointer Field
Null Pointer


Head
aman
neha
riya
ani


Head
ani
aman
neha
riya

Algorithm

- INSERT_FIRST (STAR, ITEM)
- This algorithm inserts item as the first Node of the linked ist pointed by Start

1. [check for overflow?]
   If PTR =NULL, then
   Print, overlow
   Exit
   Else
   PTR = (NODE*)malloc (size of (node))
   // create new node from memory and assign its address to PTR
   End if
2. set PTR -> INFO = item
3. set PTR -> NEXT = START
4. set START = PTR

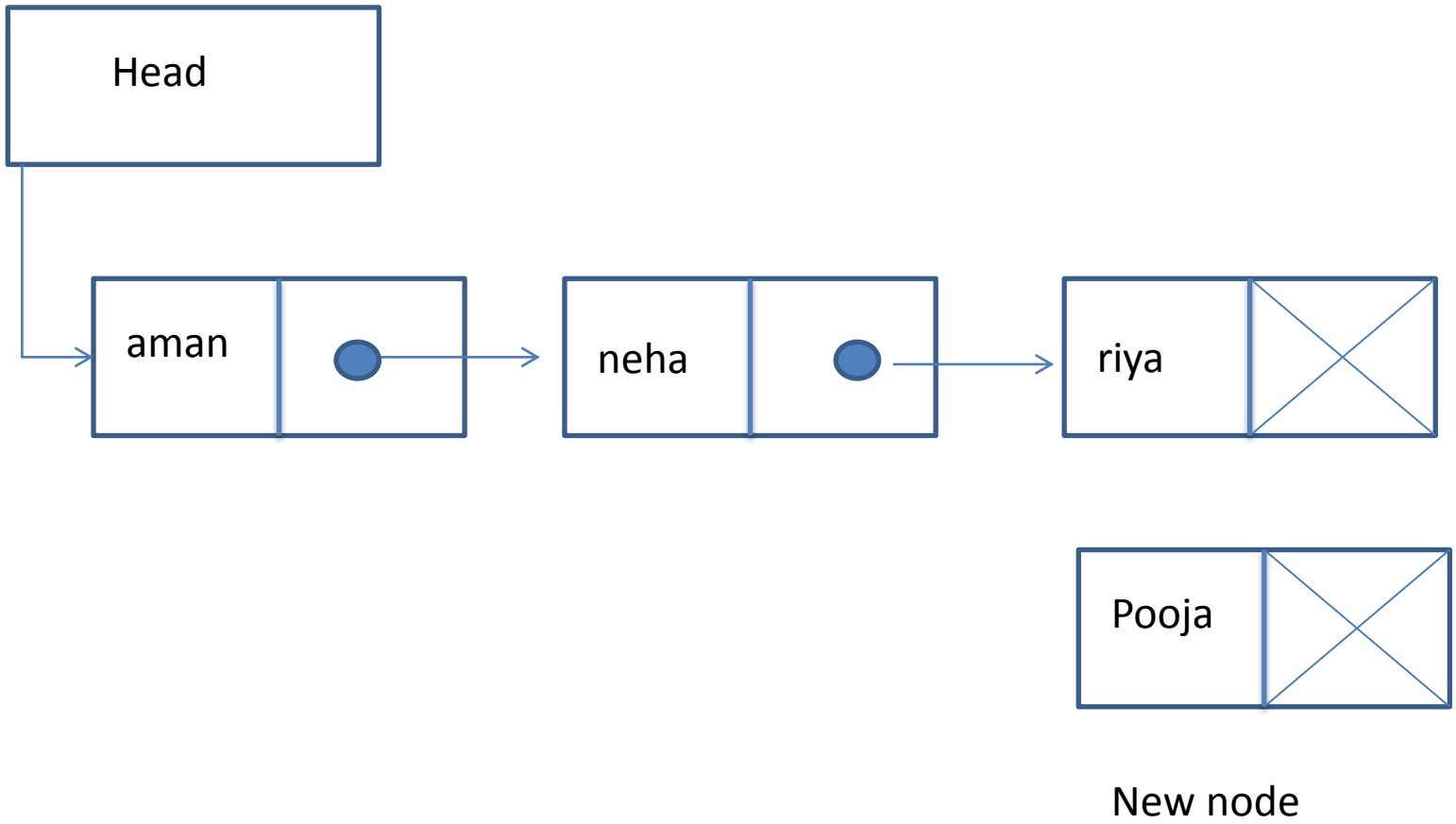
function

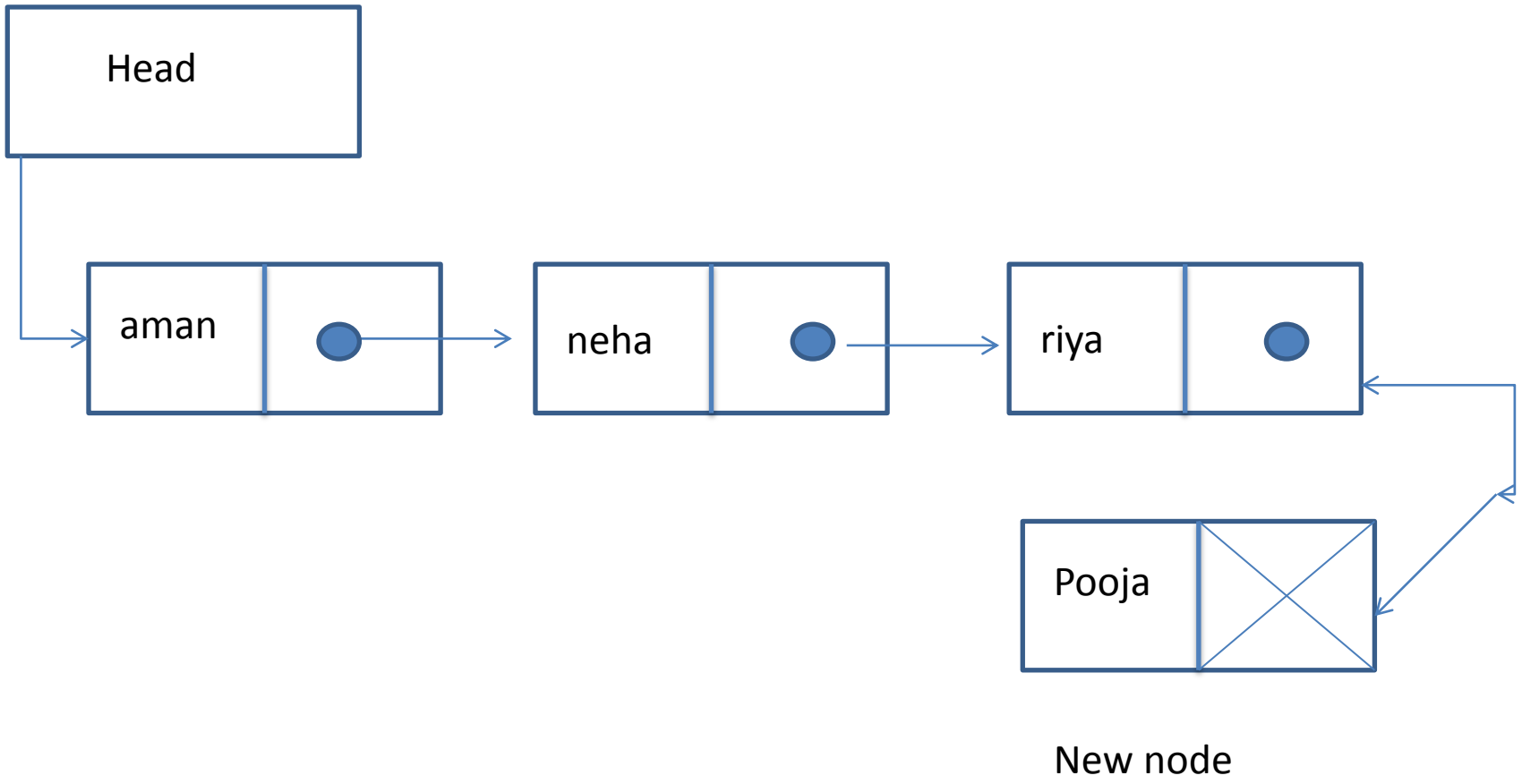
- Void insertatbegin(int item)
- { NODE *p;
- P = (NODE*)malloc(sizeof(NODE));
- P->num = item;
- If (start ==NULL)
- P->next = NULL;
- Else
- P->next = start;
- Start = p;
- }

Inserting a node at the end

- Insert-last(START, ITEM)
- This algorithm inserts an item at the last of the linked list

1. [check for overflow]
2. Set PTR -> info = item
3. Set PTR -> next = NULL
4. If START = NULL set START = P
5. SET LOC = START
6. Repeat step 7 until Loc -> next != NULL
7. Set Loc = Loc -> next
8. Set loc -> next = p


Head
aman
neha
riya
Pooja
New node


Head
aman
neha
riya
Pooja
New node

Algorithm

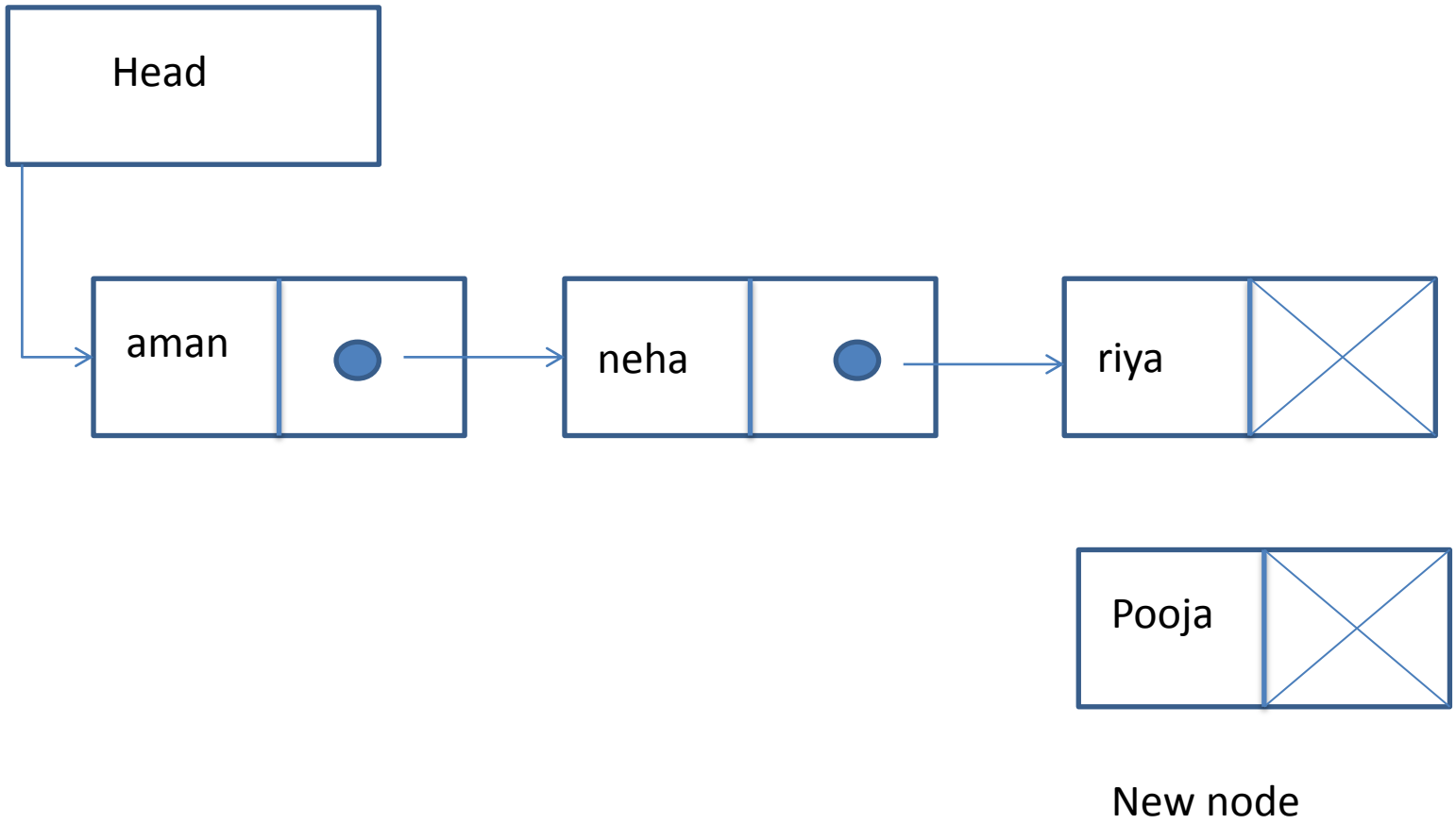
- Insert-last(START, ITEM)
- This algorithm inserts an item at the last of the linked list

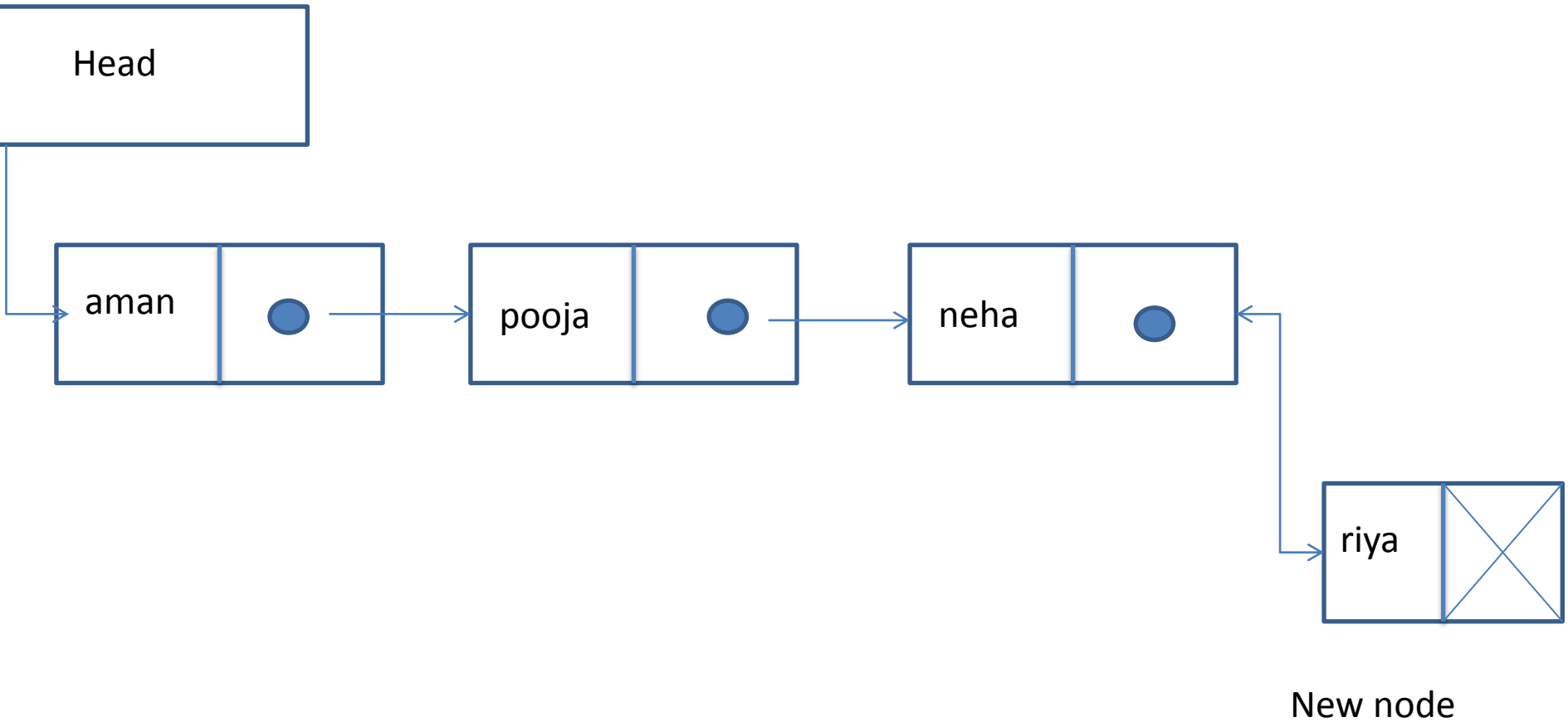
1. [check for overflow]
2. Set PTR -> info = item
3. Set PTR -> next = NULL
4. If START = NULL set START = P
5. SET LOC = START
6. Repeat step 7 until Loc -> next != NULL
7. Set Loc = Loc -> next
8. Set loc -> next = p

function

- Void insertatend(int item)
- {NODE *p, *loc;
- P = (NODE*)malloc(sizeof(NODE));
- P-> num = item;
- P-> next = NULL;
- If (start == NULL)
- { start = p; }
- else
- { loc = start;
- While (loc->next != NULL)
- { loc = loc ->next;}
- loc-> next = p;
- }

Inserting a new node at the specified position


Head
aman
neha
riya
Pooja
New node


Head
aman
pooja
neha
riya
New node

Algorithm

- Insert-location(START, ITEM, LOC)
- This algorithm inserts an item at the specified position in the linked list

1. [check for overflow]

   If PTR == NULL, then

   Print ‘overflow’

   Exit

   Else

   Ptr = (Node *) malloc (size of (node))

   End if

2. Set PTR-> INFO = item

3. If start = NULL then
   Set start = p
   Set p -> next = NULL
   End if
4. Initialise the counter(i) and pointers
   (Node * temp)
   Set i = 0
   Set temp = START
5. Repeat steps 6 and 7 until i<loc
6. Set temp = temp -> next
7. Set i = i+1
8. Set p -> next = temp -> next
9. set temp -> next = p

function

```
Void insertsp(int item, int loc)
{
NODE *p, *tmp;
Int k;
For (k=0; tmp = start; k<loc;k++)
{
Tmp = tmp->next;
If (tmp == NULL)
{
Printf("node in the list is less than one");
Return;
}}
P = (NODE*)malloc(sizeof(NODE));
P-> info = item;
P-> next = loc->next;
Loc ->next = p;
}
```

Deleting nodes

- Deleting a node from linked list has the following three instances:

1. Deleting the first node of the linked list
2. Deleting the last node of linked list
3. Deleting the specified node within the linked list.

- The steps involved in deleting the node from the linked list are as follows:

Instances for deleting a node

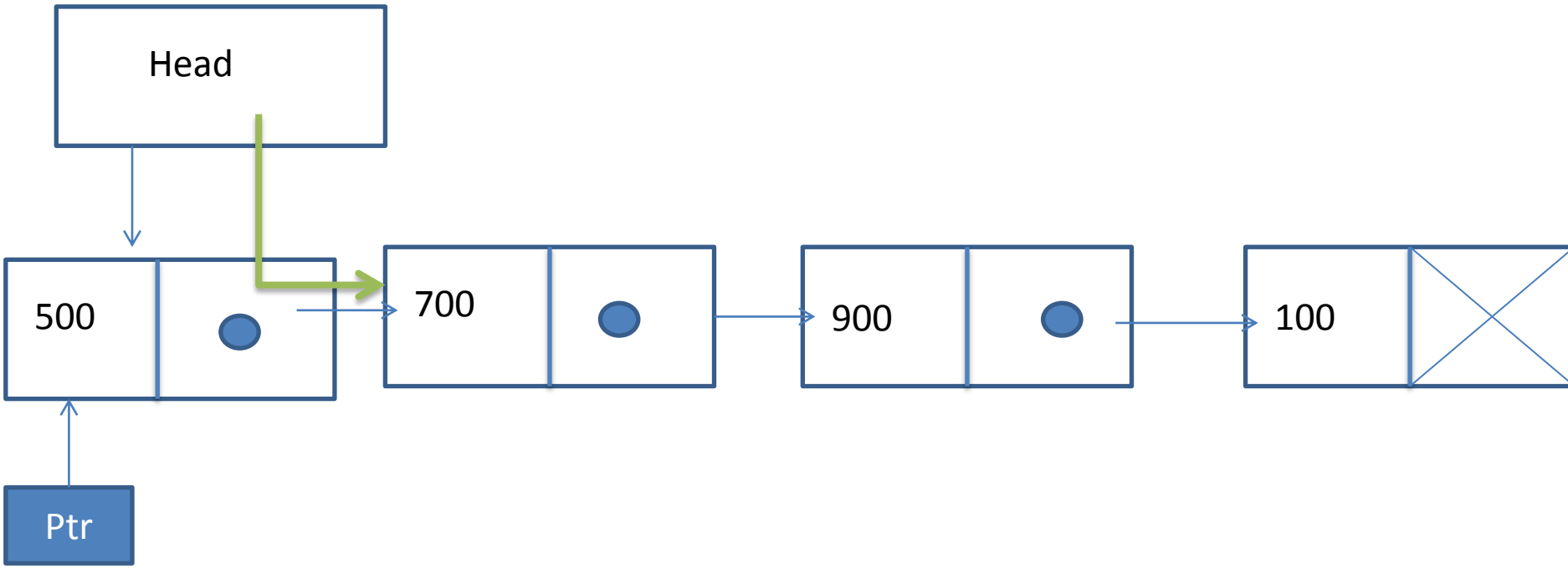
- deleting a node from linked list has three instances:

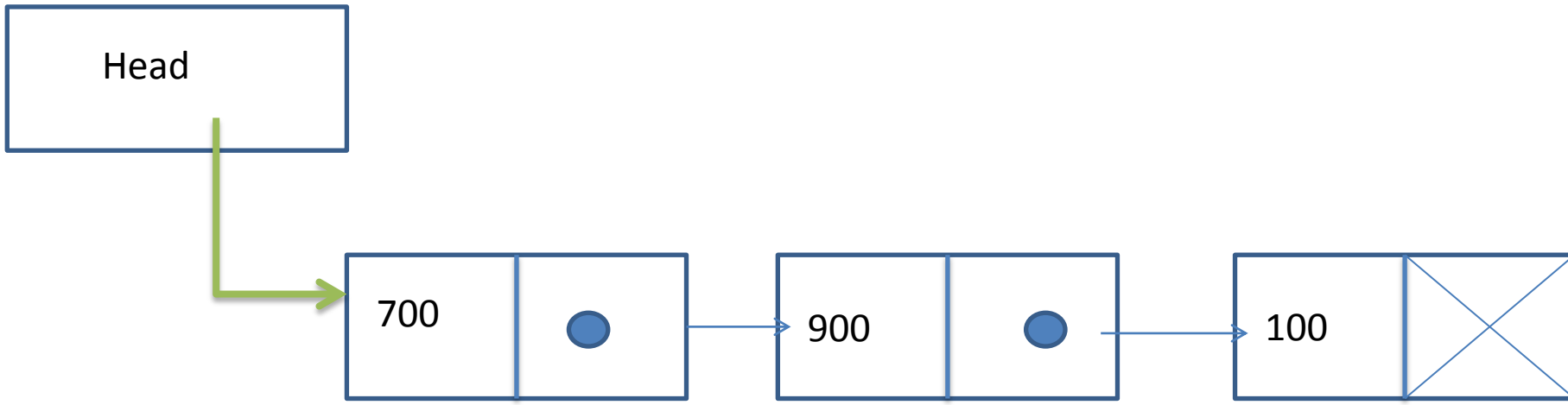
1. Deleting the first node of the linked list
2. Deleting the last node of the linked list
3. Deleting the specified node within the linked list

- Steps involved in deleting the node from the linked list:

1. If the linked list is empty then display the message “deletion is not possible”
2. If the node to be deleted is the first node then set the pointer head to point to the second node in the linked list.
3. If the node to be deleted is the last node, then go locating the last but one node and set its link field to point to null pointer
4. If the situation is other than the above three, then delete the node from the specified position within the linked list.

Deleting the first node


Head
500
700
900
100
Ptr


Head
700
900
100

algorithm

- Delete first (START)

1. [check for under flow]

   If START = NULL then

   Print (‘linked list empty’)

   Exit

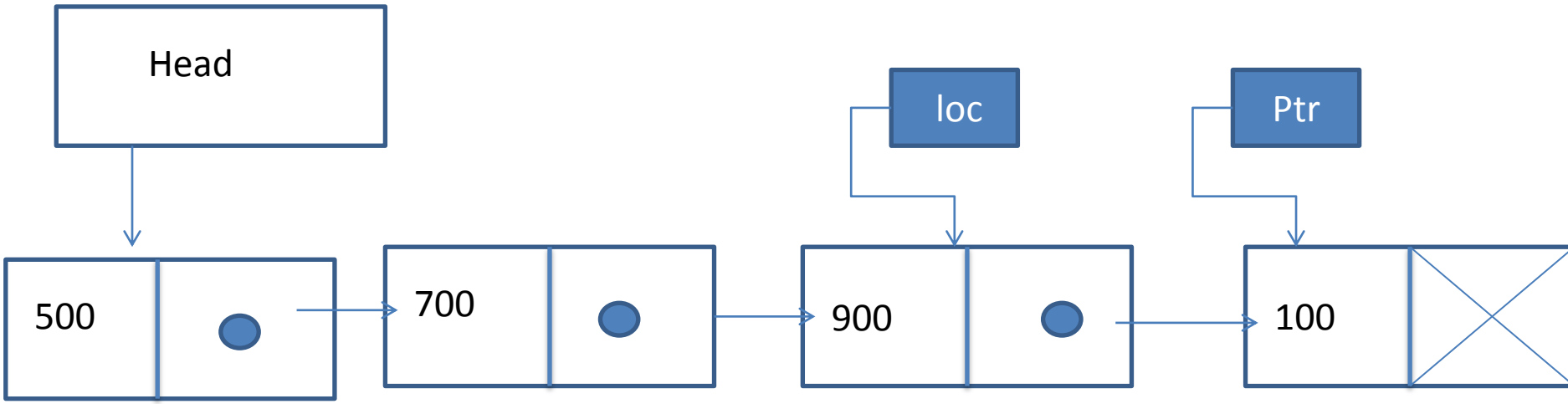
   End if

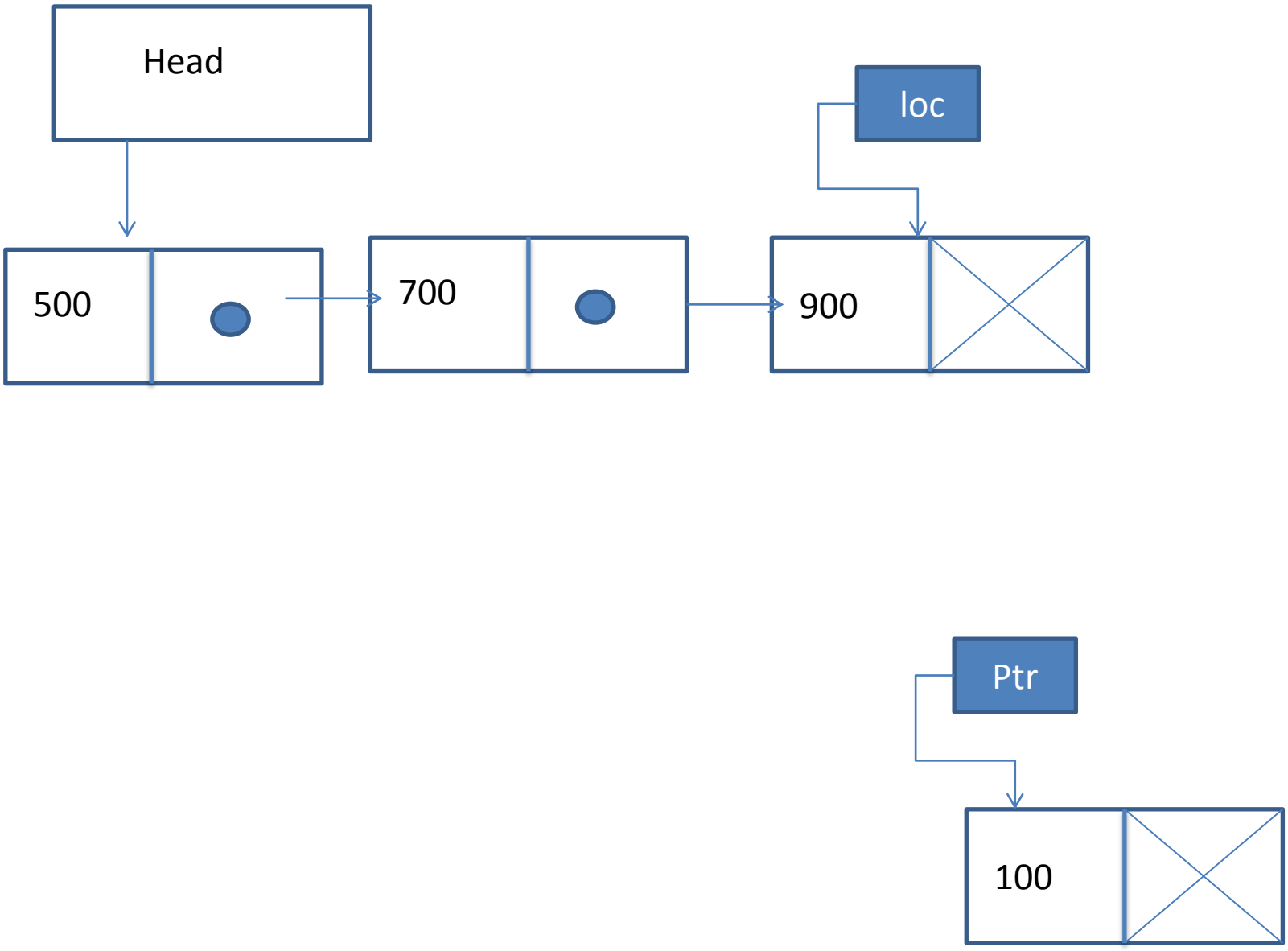
2. set ptr = START
3. set START = START -> NEXT
4. print, element deleted is, Info
5. free(ptr)

function

```
Void del_beg()
{
NODE *p;
If (start == NULL)
{
Return;
}
Else
{
P = start;
Start = start->next;
Print("element deleted is %d", p->num);
Free(p);
}
}
```

Deleting the last node


Head
loc
Ptr
500
700
900
100


Head
loc
500
700
900
Ptr
100

- Delete last (START)

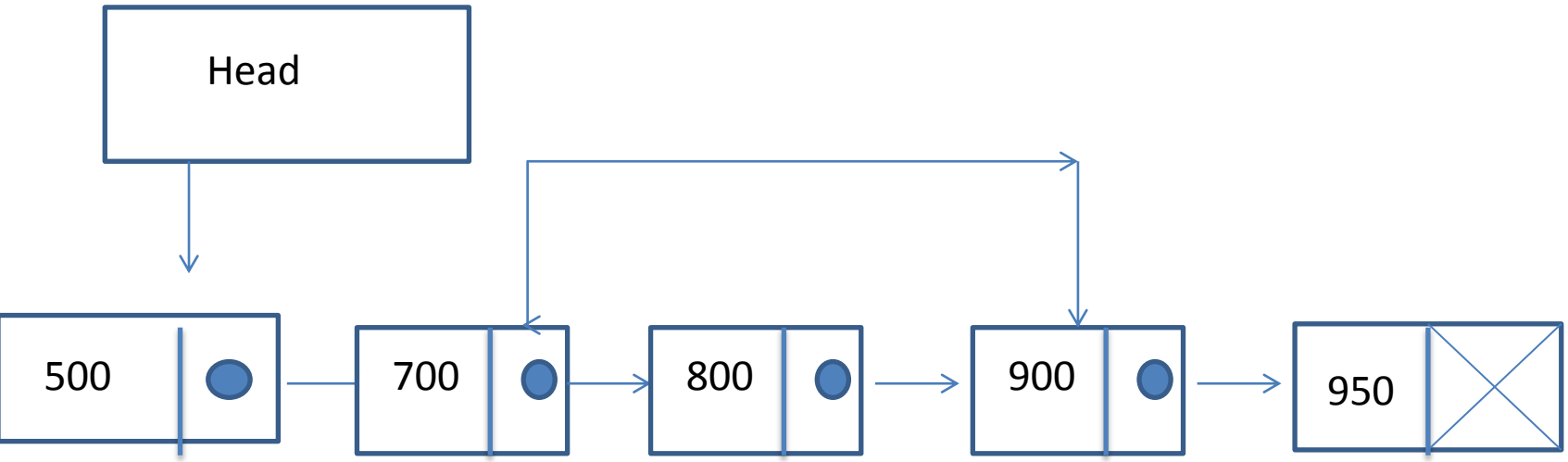
1. [check for underflow]
   If START = NULL, then
   Print('linked list empty')
   Exit
   End if
2. If START -> next = NULL, then
   Set PTR = START
   Set START = NULL
   Print element deleted is ptr -> info
   Free (PTR)
3. Set PTR = START
4. Repeat steps 5 and 6 till PTR->NEXT != NULL
5. Set LOC = PTR
6. SET PTR = PTR->NEXT
7. Set LOC ->next = NULL
8. Free(PTR)

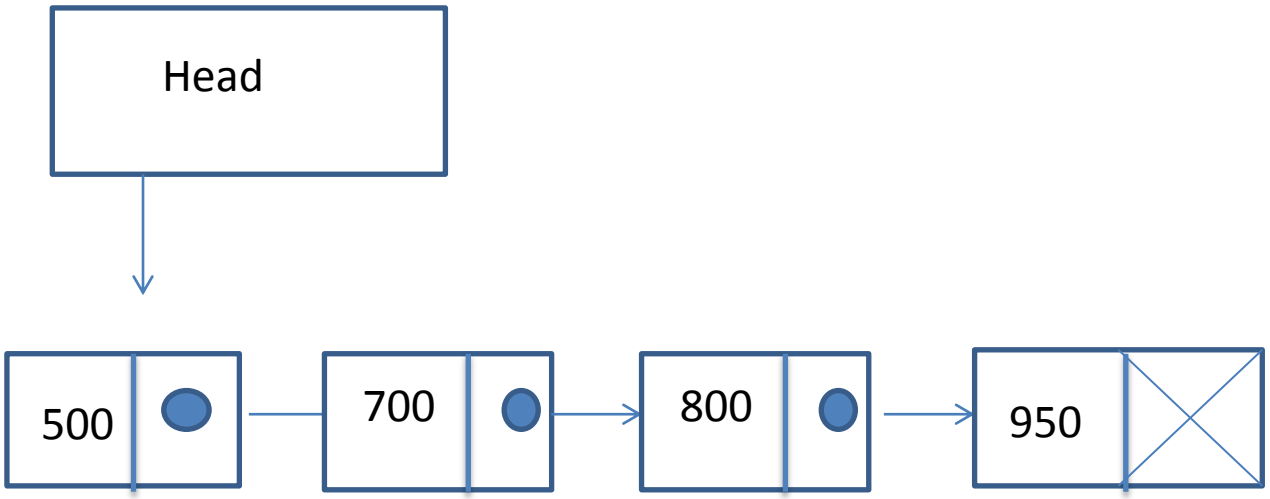
Function

```
Void del_end()
{
NODE *p, *loc;
If (start == NULL)
{return ;}
Else if ( start -> next == NULL)
{
P = start;
Start = NULL;
Printf("element deleted is %d", p->num);
Free(p);
}
```

```
Else
{
Loc = start;
P = start->next;
While (p->next == NULL)
{
Loc = ptr;
P =p->next;
}
Loc ->next = NULL
Free(P);
}
}
```

Deleting the node from specified position


Head
500
700
800
900
950


Head
500
700
800
950

- Algorithm :
- Delete_location(START, LOC)

1. [check for under flow]
   If PTR == NULL, then
   printf ”underflow”
   Exit
2. [initialize the counter, i and pointer]
   Node *tmp, node *ptr;
   Set I = 0
   Set *ptr = START
3. Repeat step 4 to 9 until i<=LOC
4. Set tmp = ptr
5. Set ptr = ptr->next
6. Set i = i +1

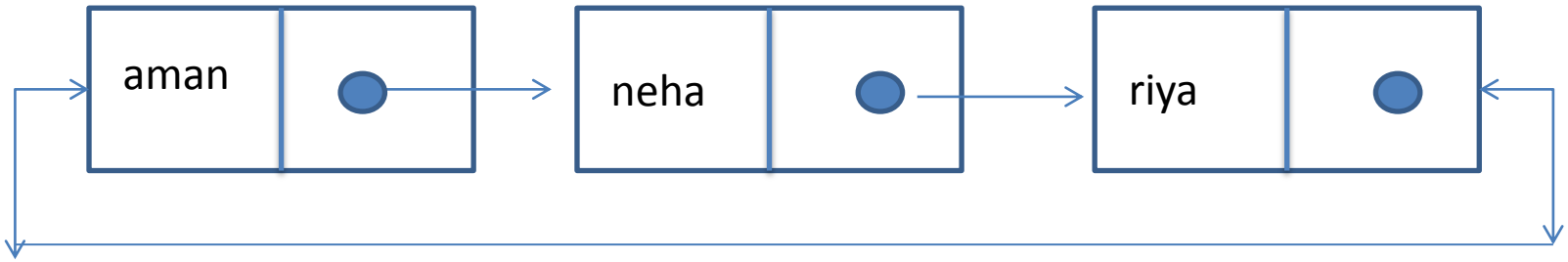
7. Print “element deleted is”
8. Set tmp -> next = ptr ->new
9. Free(ptr)

```
del_loc()
{
Node *p, *tmp;
Int I, loc;
printf*("enter the position to delete");
Scanf("%d",&loc);
If (start == null)
{
printf("empty list");
}
Else
{ ptr = start;
For(i=1;i<=loc;i++)
{ tmp = ptr; ptr = ->next;}
Printf("no. deleted is %d", ptr->num);
Tmp ->next = ptr ->next;
Free(ptr):
}}
```

Circular linked list

- It is just a singly linked list in which the link field of the last node contains the address of the first node of the list . The link field of the last node does not point to the NULL rather it points back to the beginning of the linked list.
- It has no end.

- Inserting a node at the beginning
- Inserting a node at the end
- Deleting a node from the beginning
- Deleting a node from the end


aman
neha
riya

- structure for the circular linked list :
- Struct node
- {
- Int num;
- Struct node *next;
- };
- Typedef struct Node node;
- Node *start = NULL;
- Node *last = NULL;

Inserting a node at the beginning

- Algorithm:

1. [check for overflow]
   If ptrr = NULL, then
   Print, overflow
   Exit
   Else
   Ptr = (Node*)malloc(sizeof(NODE));
2. If start == NULL then
   Set ptr ->num = item
   Set ptr ->next = ptr
   Set start = ptr
   Set last = ptr
   End if
3. Set ptr -> num = item
4. Set ptr ->next = start
5. Set start = ptr
6. Set last -> next = ptr

```
Function:
{
Node *p;
P= (node *)malloc(sizeof(node));
Printf("enter the no.");
Scanf("%d",&p->num);
If(start ==NULL)
{
P->next = p;
Start = p;
Last = p;
}
Else
{
P->next = start;
Start = p;
Last ->next = p;
}
}
```

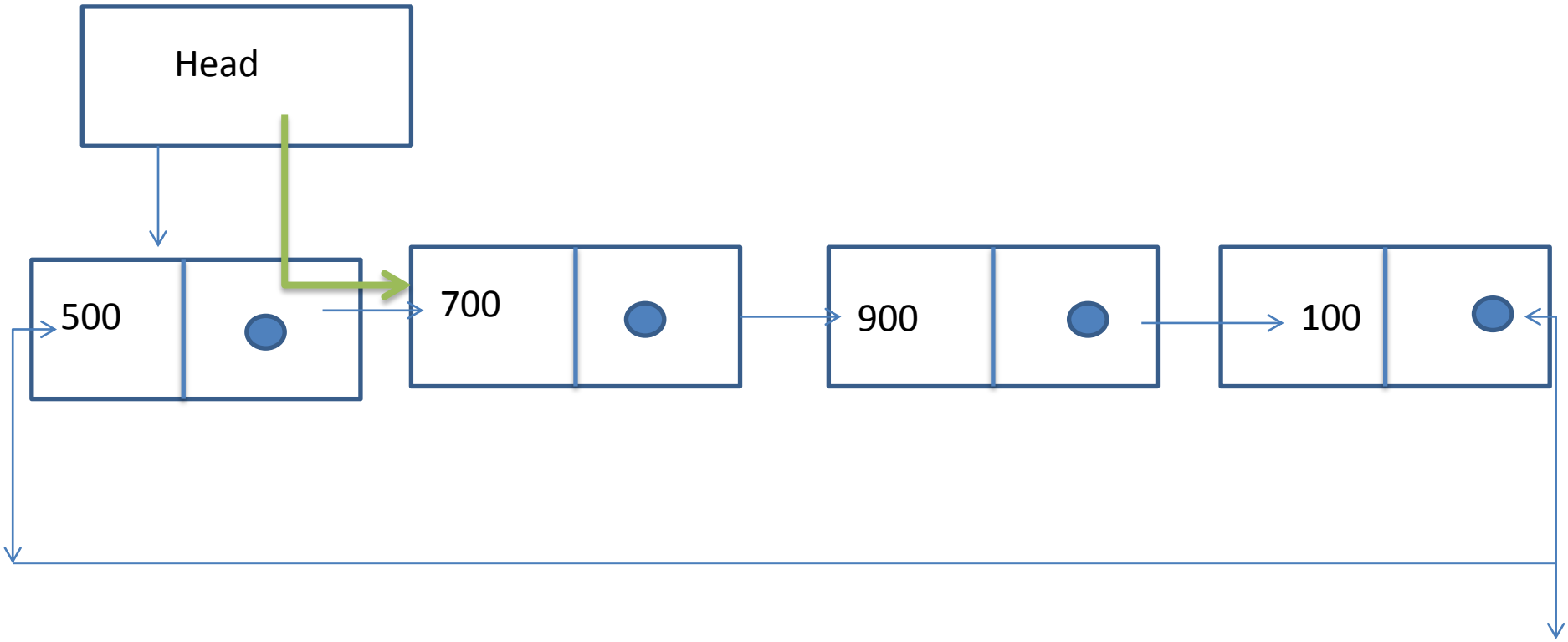
Inserting a node at the end

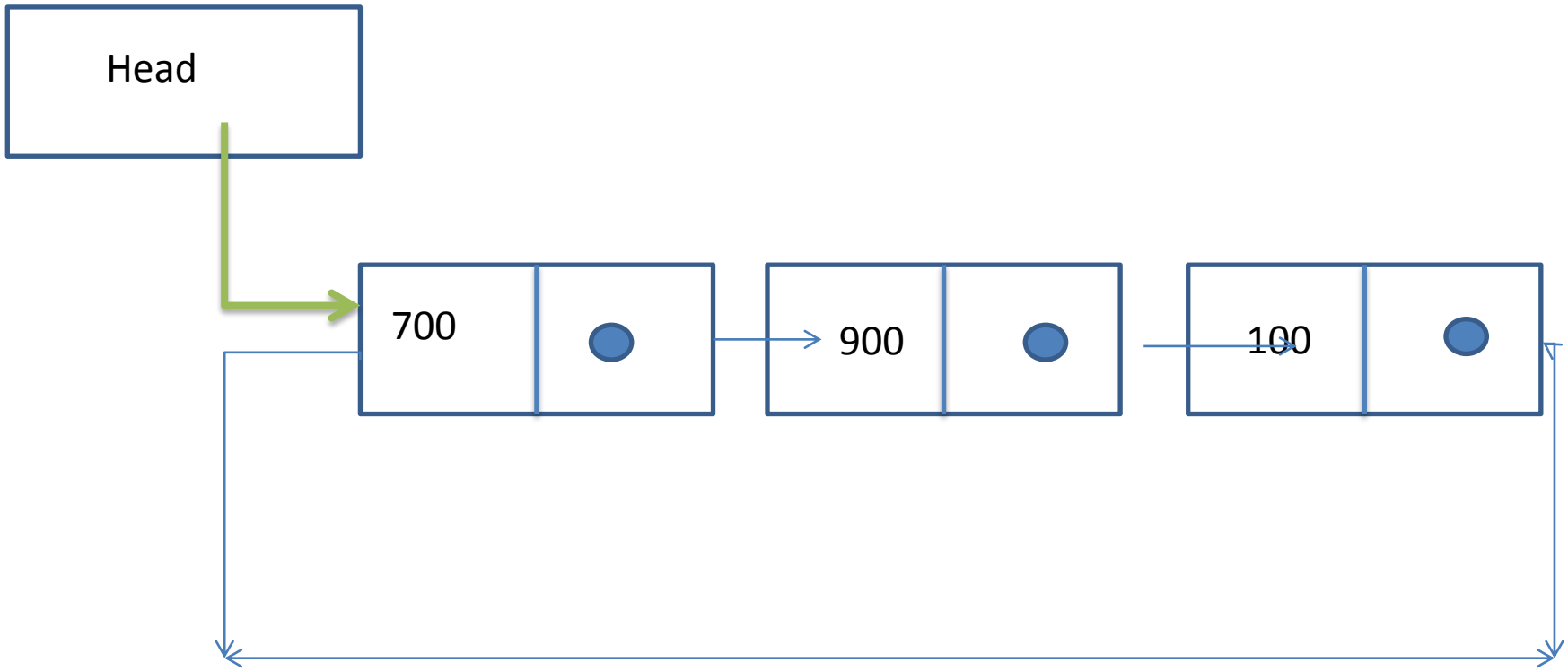
- Algorithm:

1. [check for overflow]
   If ptrr = NULL, then
   Print, overflow
   Exit
   Else
   Ptr = (Node*)malloc(sizeof(NODE));
2. If start == NULL then
   Set ptr ->num = item
   Set ptr ->next = ptr
   Set last = ptr
   Set start = ptr
   End if
3. Set ptr -> num = item
4. Set last ->next = ptr
5. Set last = ptr
6. Set last -> next = start

```
Node *last(node *start)
{
Node *p;
P=(node*)malloc(sizeof(node));
Printf("enter the no.");
Scanf("%d",&p->num);
If(start ==NULL)
{
P->next =p;
Start = last = p;
}
Else
{
Last ->next = p;
Last = p;
Last->next = start;
}
Return (start);
}
```

Deleting a node from the beginning


Head
500
700
900
100

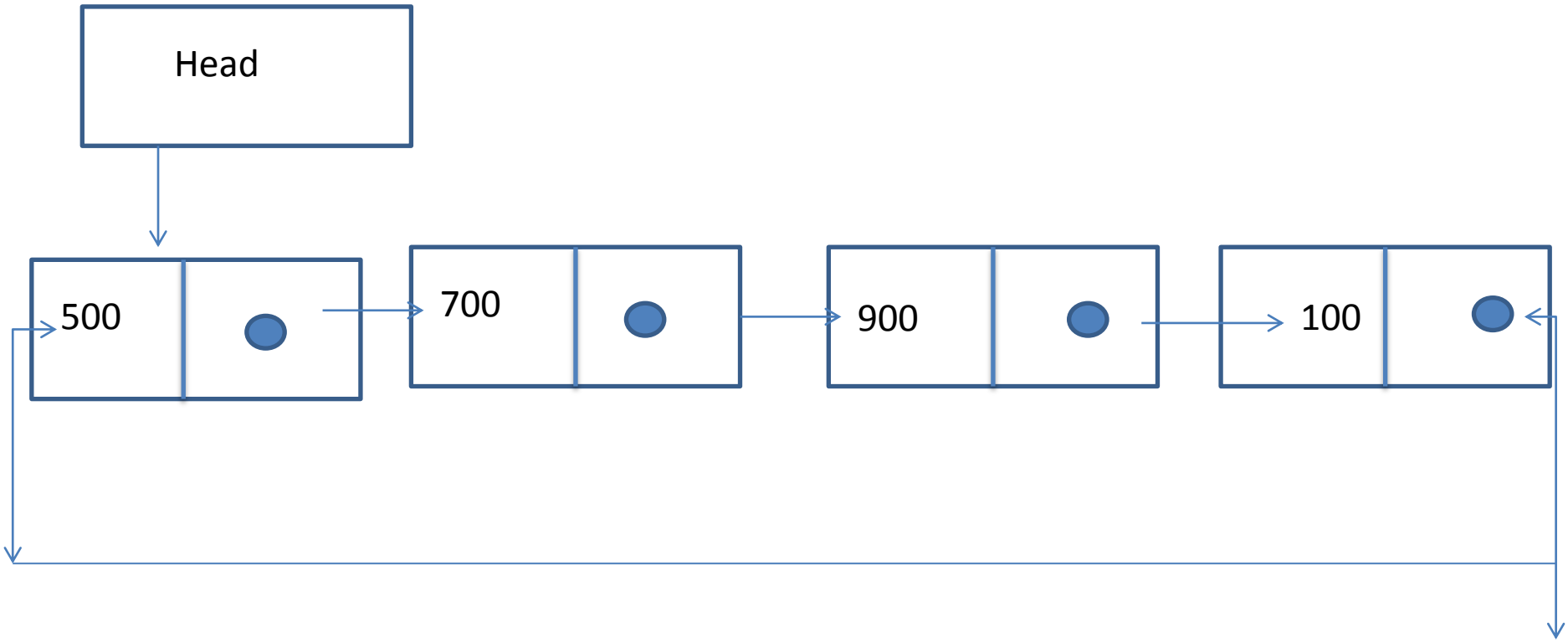

Head
700
900
100

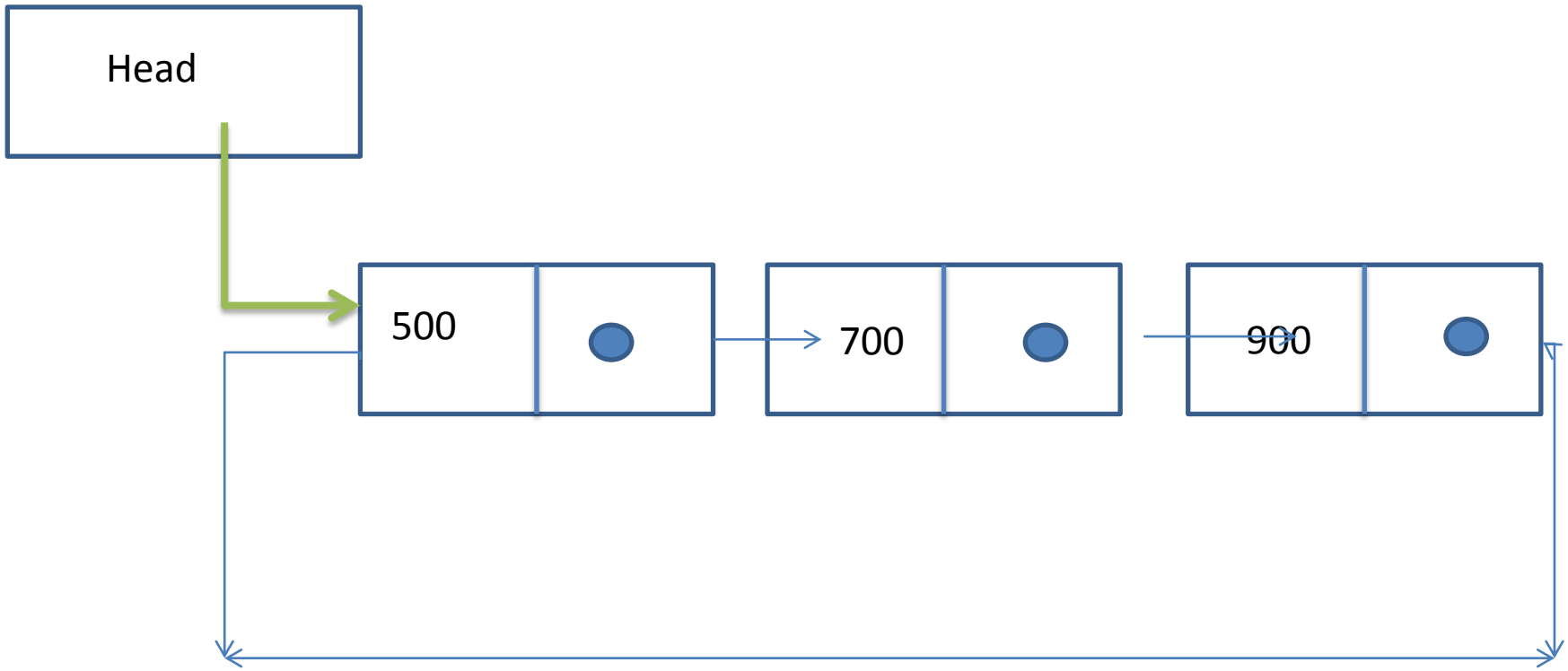
- Algorithm:
- Delete_first()

1. [check for under flow]
   If start = NULL
   Print ‘circular list empty’
   Exit
   End if
2. Set ptr = start
3. Set start = start->next
4. Print , element deleted
5. Set last->next = start
6. Free(ptr);

- Function:
- Node *delete_first(node *start)
- {
- Node *p
- P = start
- If (p == NULL)
- {
- Printf(“list empty”);
- }
- Else
- {
- P = start;
- Start = start ->next;
- Printf(“element deleted”);
- Last->next = p;
- }
- Return start;
- }

Deleting a node from the end


Head
500
700
900
100


Head
500
700
900

- Algorithm:

1. [check for underflow]
   If start = NULL then
   Print ‘circular list is empty’
   Exit
   End if
2. Set ptr = start
3. Repeat steps 4 and 5 until
   Ptr!=start
4. Set ptr1= ptr
5. Set ptr = ptr->next
6. Print “element deleted”
7. Set ptr1->next = ptr->next
8. Set last = ptr1
9. Return start

```
Function:
Node *del_last(node *start)
{
Node *p, *q;
P = start;
If (p==NULL)
{
Printf(“list empty”);
}
Else
{
While(p->next != last)
{
q= p;
P= p->next;
}
Printf(“element deleted”);
q->next= p->next;
Last = q;
}
Return start;
}
```

```
Node *insertfirst(node *start)
{
Node *p;
P = (node *)malloc(sizeof(node));
Printf("enter the no.");
Scanf("%d",&p->num);
If(start==NULL)
{
P->next = p;
Start =p;
Last = p;
}
Else
{
P->next = start;
Start = p;
Last->next = p;
}
}
```

Doubly linked list

- A doubly linked list is one in which all nodes are linked together by multiple number of links which help in accessing both the successor node and predecessor node from the given node position. It provides bi-directional traversing.
- Inserting a node at the beginning
- Inserting a node at the end
- Deleting a node from the beginning
- Deleting a node from the end

Circular doubly linked list

- A circular linked list is one which has both the successor pointer and predecessor pointer in circular manner.
- Inserting a node at the beginning
- Inserting a node at the end
- Deleting a node from the begging
- Deleting a node from the end