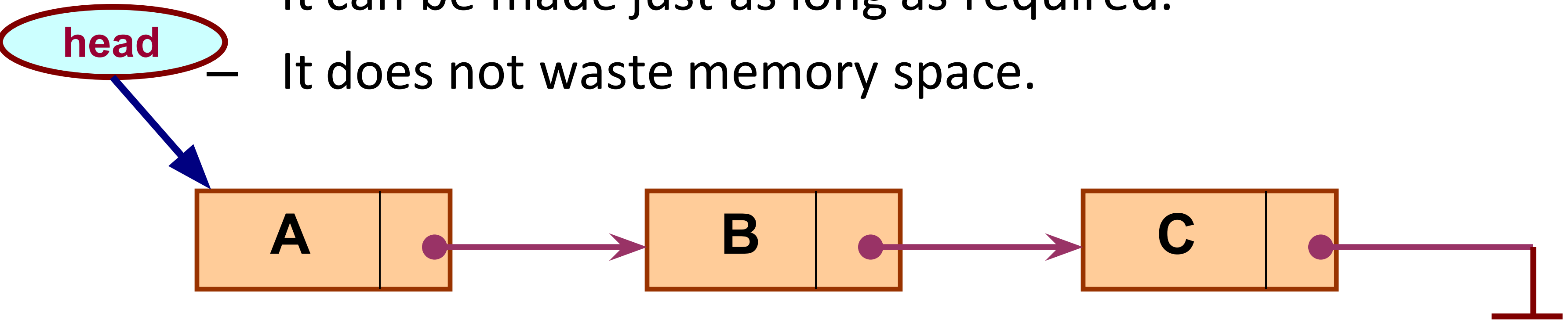
Linked List

Introduction

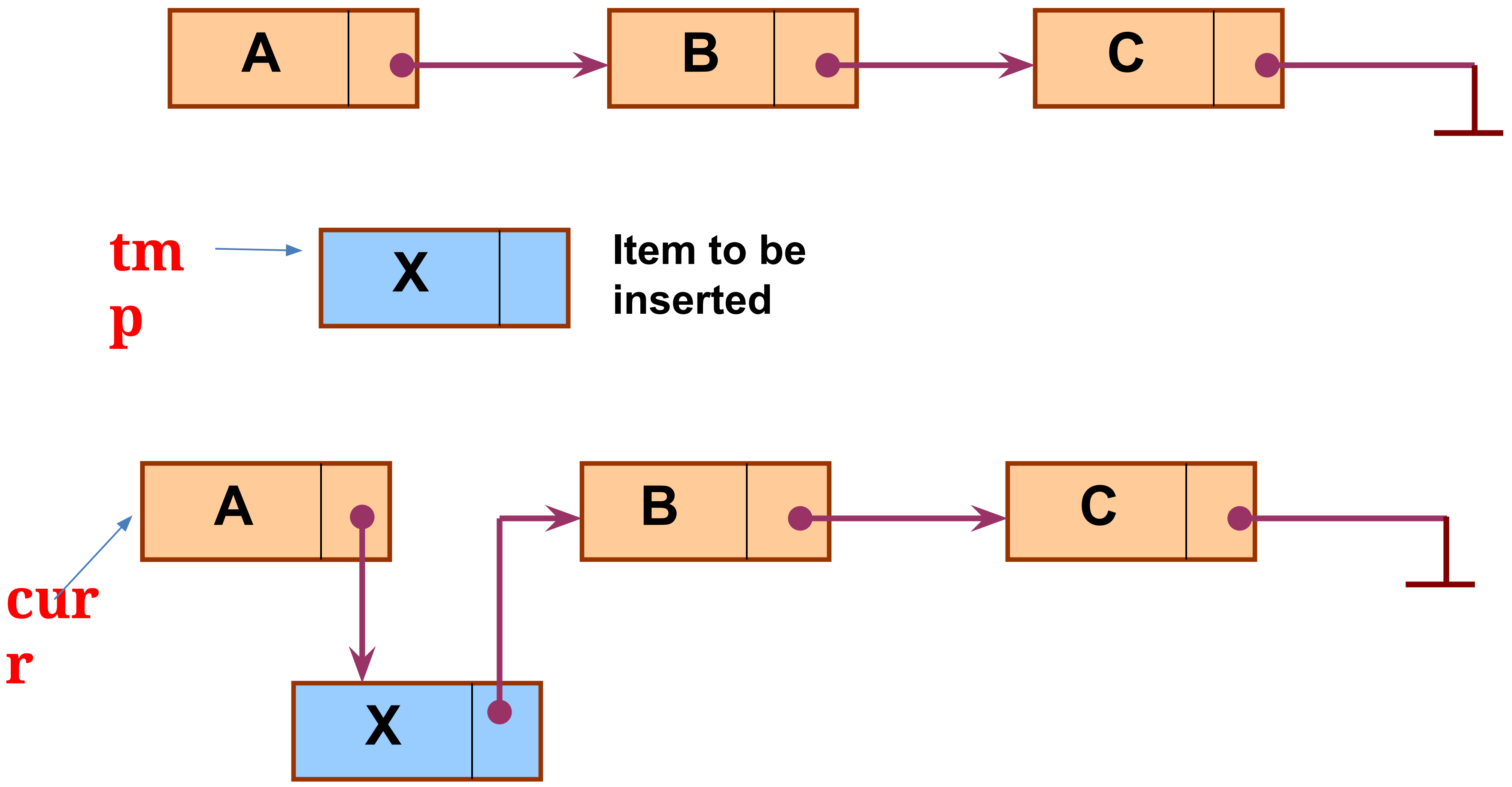
- A linked list is a data structure which can change during execution.
  - Successive elements are connected by pointers.
  - Last element points to NULL.
  - It can grow or shrink in size during execution of a program.
  - It can be made just as long as required.
  - It does not waste memory space.



- Keeping track of a linked list:
  - Must know the pointer to the first element of the list (called *start*, *head*, etc.).

- Linked lists provide flexibility in allowing the items to be rearranged efficiently.
  - Insert an element.
  - Delete an element.

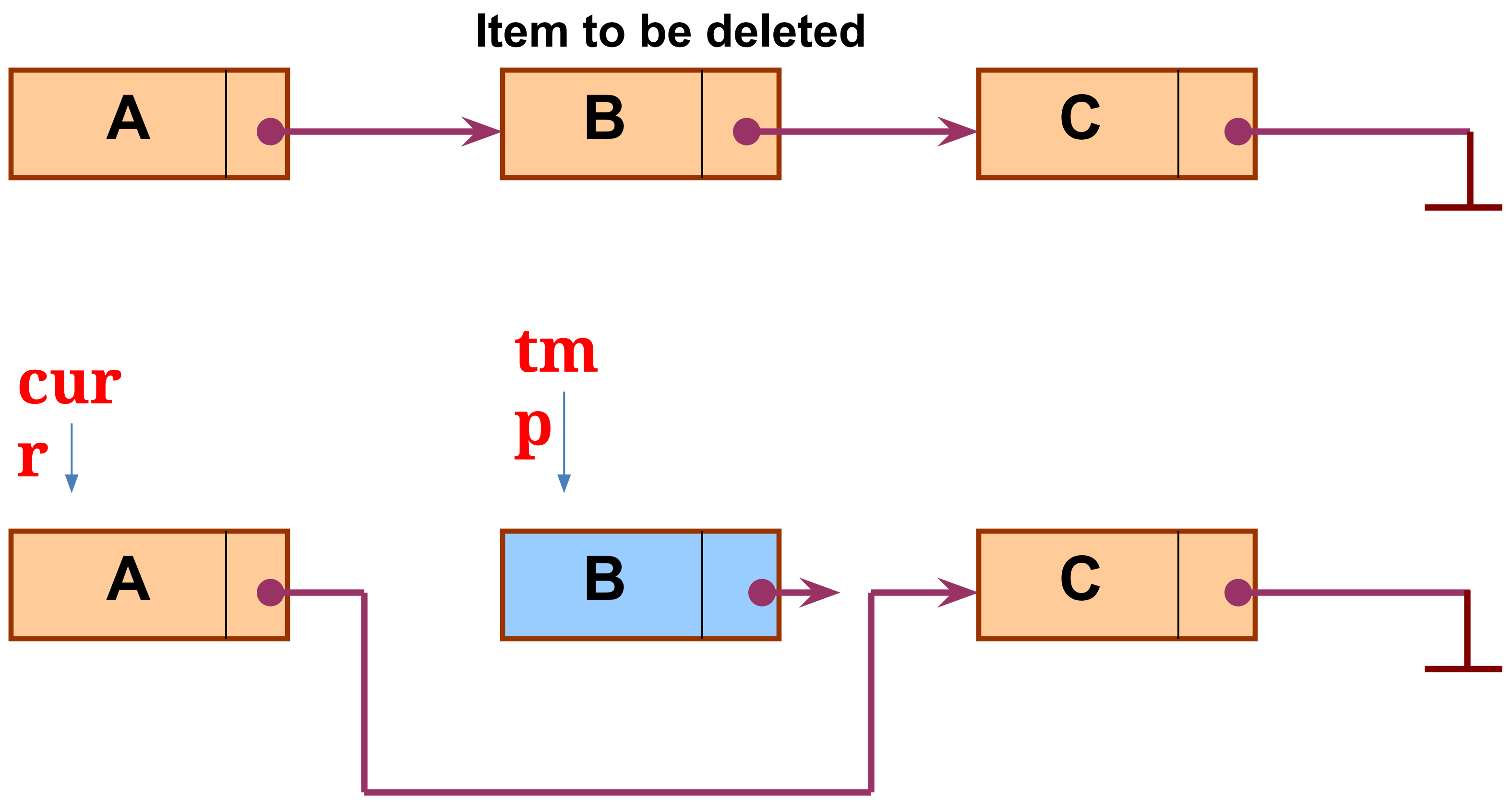
Illustration: Insertion



Pseudo-code for insertion

```
typedef struct nd {
   struct item data;
   struct nd * next;
   } node;

void insert(node *curr)
{
node * tmp;

tmp=(node *)
malloc(sizeof(node));
tmp->next=curr->next;
curr->next=tmp;
}
```

Illustration: Deletion



Pseudo-code for deletion

```
typedef struct nd {
   struct item data;
   struct nd * next;
   } node;

void delete(node *curr)
{
node * tmp;
 tmp=curr->next;
curr->next=tmp->next;
free(tmp);
}
```

In essence ...

- For insertion:
  - A record is created holding the new item.
  - The next pointer of the new record is set to link it to the item which is to follow it in the list.
  - The next pointer of the item which is to precede it must be modified to point to the new item.
- For deletion:
  - The next pointer of the item immediately preceding the one to be deleted is altered, and made to point to the item following the deleted item.

Array versus Linked Lists

- Arrays are suitable for:
  - Inserting/deleting an element at the end.
  - Randomly accessing any element.
  - Searching the list for a particular value.
- Linked lists are suitable for:
  - Inserting an element.
  - Deleting an element.
  - Applications where sequential access is required.
  - In situations where the number of elements cannot be predicted beforehand.

Basic Operations on a List

- Creating a list
- Traversing the list
- Inserting an item in the list
- Deleting an item from the list
- Concatenating two lists into one

Example: Working with linked list

- Consider the structure of a node as follows:

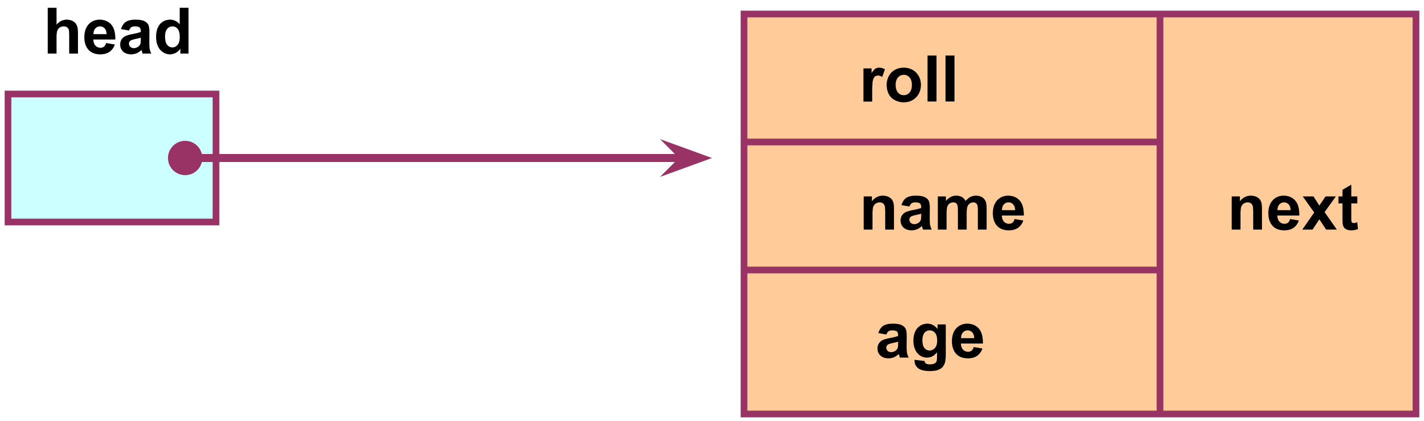
```
struct stud {
              int    roll;
              char   name[25];
              int    age;
              struct stud *next;
            };

  /* A user-defined data type called “node” */

typedef struct stud node;
node *head;
```

Creating a List

How to begin?

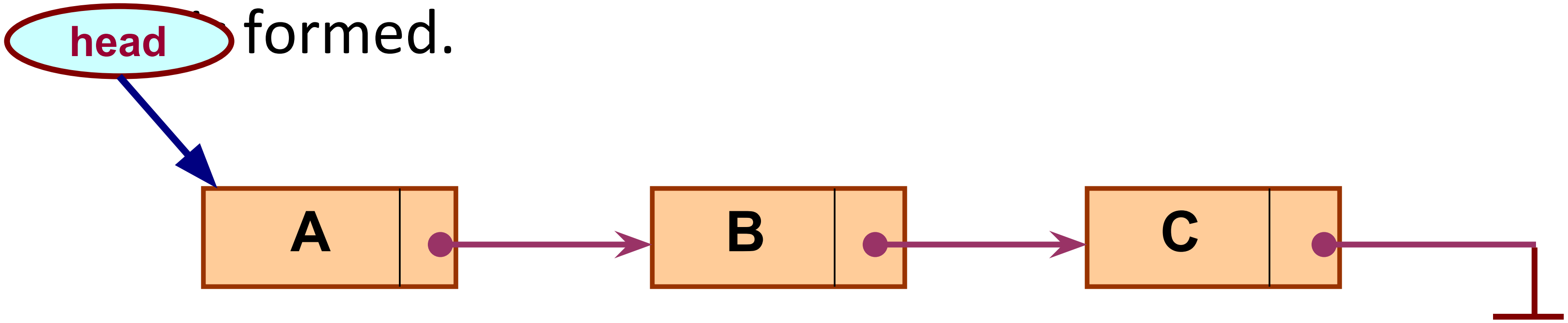
- To start with, we have to create a node (the first node), and make head point to it.

```
head = (node *)
  malloc(sizeof(node));
```



Contd.

- If there are n number of nodes in the initial linked list:
  - Allocate n records, one by one.
  - Read in the fields of the records.
  - Modify the links of the records so that the chain be formed.



```
node *create_list()
{
    int  k, n;
    node  *p, *head;

    printf  ("\n How many elements to enter?");
     scanf ("%d", &n);

    for  (k=0; k<n; k++)
    {
        if (k == 0) {
          head = (node *) malloc(sizeof(node));
          p = head;
         }
         else {
              p->next  = (node *) malloc(sizeof(node));
              p = p->next;
             }

        scanf ("%d %s %d", &p->roll, p->name, &p->age);
    }

    p->next  =  NULL;
    return (head);
}
```

- To be called from `main()` function as:

```
node *head;
………
head = create_list();
```

Traversing the List

What is to be done?

- Once the linked list has been constructed and *head* points to the first node of the list,
  - Follow the pointers.
  - Display the contents of the nodes as they are traversed.
  - Stop when the *next* pointer points to NULL.

```
void display (node *head)
{
  int  count = 1;
  node  *p;

  p = head;
  while (p != NULL)
  {
    printf ("\nNode %d: %d %s %d", count,
                     p->roll, p->name, p->age);
    count++;
    p = p->next;
  }
  printf ("\n");
}
```

- To be called from `main()` function as:

```
node *head;
………
display (head);
```

Inserting a Node in a List

How to do?

- The problem is to insert a node *before a specified node*.
  - Specified means some value is given for the node (called *key*).
  - In this example, we consider it to be `roll`.
- Convention followed:
  - If the value of roll is given as *negative*, the node will be inserted at the *end* of the list.

Contd.

- When a node is added at the beginning,
  - Only one next pointer needs to be modified.
    - *head* is made to point to the new node.
    - New node points to the previously first element.
- When a node is added at the end,
  - Two next pointers need to be modified.
    - Last node now points to the new node.
    - New node points to NULL.
- When a node is added in the middle,
  - Two next pointers need to be modified.
    - Previous node now points to the new node.
    - New node points to the next node.

```
void insert (node **head)
{
    int  k = 0, rno;
    node *p, *q, *new;

    new = (node *) malloc(sizeof(node));

    printf ("\nData to be inserted: ");
      scanf ("%d %s %d", &new->roll, new->name, &new->age);
    printf ("\nInsert before roll (-ve for end):");
      scanf ("%d", &rno);

    p = *head;

    if (p->roll == rno)      /* At the beginning */
    {
        new->next = p;
        *head = new;
    }
```

```
  else
    {
   while ((p != NULL) && (p->roll != rno))

        {
            q = p;
            p = p->next;
        }

        if  (p == NULL)        /* At the end */

        {
            q->next = new;
            new->next = NULL;
        }
    else if  (p->roll  == rno)
                             /* In the middle */

                {
                    q->next = new;
                    new->next = p;
                }
    }
}
```

The pointers q and p always point to consecutive nodes.

- To be called from `main()` function as:

```
node *head;
………
insert (&head);
```

Deleting a node from the list

What is to be done?

- Here also we are required to delete a specified node.
  - Say, the node whose `roll` field is given.
- Here also three conditions arise:
  - Deleting the first node.
  - Deleting the last node.
  - Deleting an intermediate node.

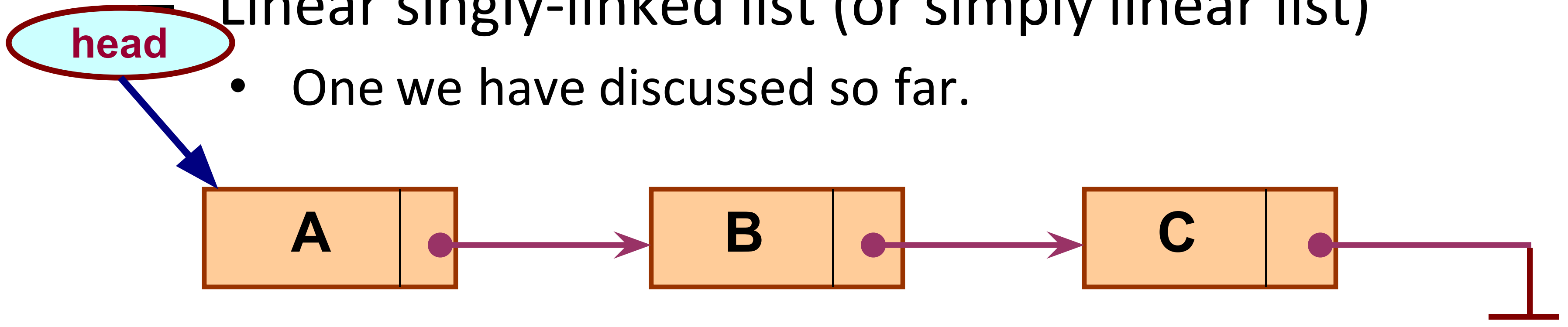
```
void  delete (node **head)
{
    int  rno;
    node  *p, *q;

    printf ("\nDelete for roll :");
      scanf ("%d", &rno);

    p = *head;
    if  (p->roll == rno)
             /* Delete the first element */
    {
        *head = p->next;
        free (p);
    }
```

```
  else
     {
           while  ((p != NULL) && (p->roll != rno))

           {
               q = p;
               p  =  p->next;
           }

           if  (p == NULL)      /* Element not found
*/
              printf ("\nNo match :: deletion
failed");

           else if (p->roll == rno)
                          /* Delete any other element
*/
                {
                    q->next  =  p->next;
                    free (p);
                }
     }
```

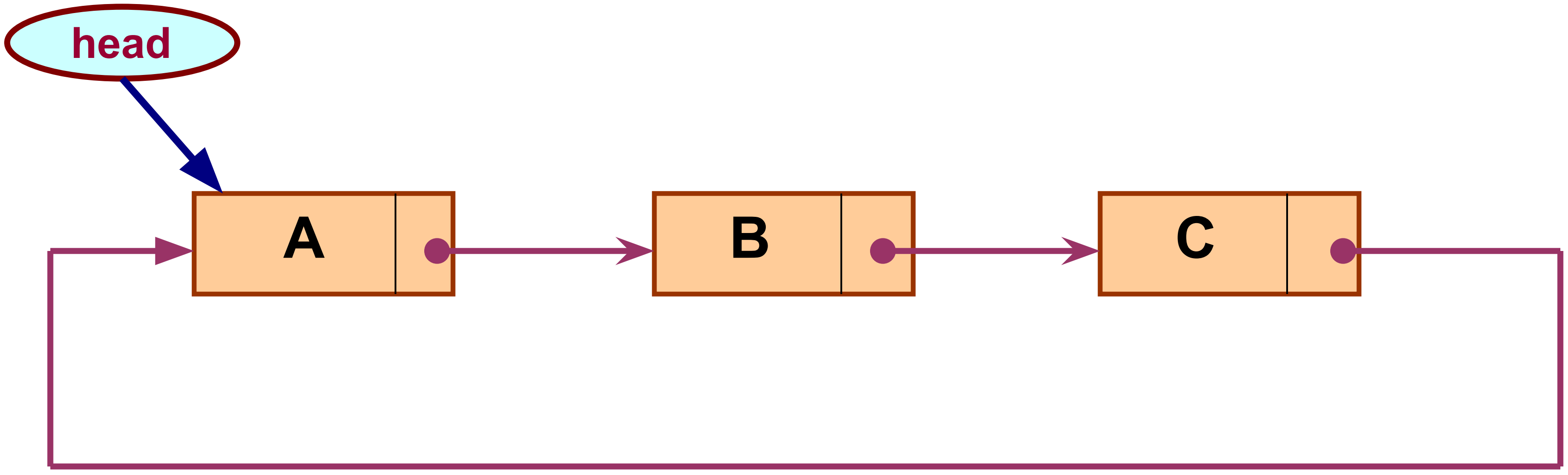
Types of Lists

- Depending on the way in which the links are used to maintain adjacency, several different types of linked lists are possible.

  – Linear singly-linked list (or simply linear list)
    - One we have discussed so far.



- Circular linked list
  - The pointer from the last element in the list points back to the first element.



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

struct Node *addToEmpty(struct Node *last, int data)
{
   // This function is only for empty list
   if (last != NULL)
     return last;

   // Creating a node dynamically.
   struct Node *temp =
         (struct Node*)malloc(sizeof(struct Node));

   // Assigning the data.
   temp -> data = data;
   last = temp;
     // Creating the link.
   last -> next = last;

   return last;
}
```

```
node *create_list()
{
    int  k, n;
    node  *p, *head;

    printf  ("\n How many elements to enter?");
     scanf ("%d", &n);

    for  (k=0; k<n; k++)
    {
        if (k == 0) {
          head = (node *) malloc(sizeof(node));
          p = head;
         }
         else {
               p->next  = (node *) malloc(sizeof(node));
               p = p->next;
              }

        scanf ("%d %s %d", &p->roll, p->name, &p->age);
    }

    p->next  =  head;
    return (head);
}
```

```
void display (node *head)
{
  int  count = 1;
  node  *p;

  p = head;
do
  {
    printf ("\nNode %d: %d %s %d", count,
                   p->roll, p->name, p->age);
    count++;
    p = p->next;
  }
while(p != head)
  printf ("\n");
}
```

```
void traverse(struct Node *last)
{
    struct Node *p;
      // If list is empty, return.
    if (last == NULL)
    {
        cout << "List is empty." << endl;
        return;
    }
      // Pointing to first Node of the list.
    p = last -> next;
      // Traversing the list.
    do
    {
        cout << p -> data << " ";
        p = p -> next;
      }
    while(p != last->next);
  }
```

Inserting a Node in a Circular List

- When a node is added at the beginning
- When a node is added at the end
- When a node is added in the middle

Inserting At Beginning of the list

We can use the following steps to insert a new node at beginning of the circular linked list...

Step 1: Create a **newNode** with given value.

Step 2: Check whether list is **Empty** (**head** == **NULL**)

Step 3: If it is **Empty** then, set **head** = **newNode** and **newNode → next** = **he ad** .

Step 4: If it is **Not Empty** then, define a Node pointer '**temp**' and initialize with '**head**'.

Step 5: Keep moving the '**temp**' to its next node until it reaches to the last node (until '**temp → next** == **head**').

Step 6: Set '**newNode → next** =**head**', '**head** = **newNode**' and '**temp → next** = **head**'.

Inserting At End of the list

Step 1: Create a **newNode** with given value.

Step 2: Check whether list is **Empty** (**head** == **NULL**).

Step 3: If it is **Empty** then, set **head** = **newNode** and **newNode** → **next** = **head**.

Step 4: If it is **Not Empty** then, define a node pointer **temp** and initialize with **head**.

Step 5: Keep moving the **temp** to its next node until it reaches to the last node in the list (until **temp → next** == **head**).

Step 6: Set **temp →**
next = **newNode** and **newNode → next** = **head**.

Inserting At Specific location in the list (After a Node)

Step 1: Create a **newNode** with given value.

Step 2: Check whether list is **Empty** (**head** == **NULL**)

Step 3: If it is **Empty** then, set **head** = **newNode** and **newNode →** **next** = **head**.

Step 4: If it is **Not Empty** then, define a node pointer **temp** and initialize with **head**.

Step 5: Keep moving the **temp** to its next node until it reaches to the node after which we want to insert the newNode (until **temp1 → data** is equal to **location**, here location is the node value after which we want to insert the newNode).

Step 6: Every time check whether **temp** is reached to the last node or not. If it is reached to last node then display **'Given node is not found in the list!!! Insertion not possible!!!'** and terminate the function. Otherwise move the **temp** to next node.

Step 7: If **temp** is reached to the exact node after which we want to insert the newNode then check whether it is last node (temp → next == head).

Step 8: If **temp** is last node then set **temp → next** = **newNode** and **newNode → next** = **head**.

Step 8: If **temp** is not last node then set **newNode → next** = **temp → next** and **temp → next** = **newNode**.

Deletion

In a circular linked list, the deletion operation can be performed in three ways those are as follows...

- Deleting from Beginning of the list
- Deleting from End of the list
- Deleting a Specific Node

Deletion at The begining

Step 1: Check whether list is **Empty** (**head** == **NULL**)

Step 2: If it is **Empty** then, display **'List is Empty!!! Deletion is not possible'** and terminate the function.

Step 3: If it is **Not Empty** then, define two Node pointers **'temp1'** and '**temp2**' and initialize both '**temp1**' and '**temp2**' with **head**.

Step 4: Check whether list is having only one node (**temp1 → next** == **head**)

Step 5: If it is **TRUE** then set **head** = **NULL** and delete **temp1** (Setting **Empty** list conditions)

Step 6: If it is **FALSE** move the **temp1** until it reaches to the last node. (until **temp1 →** **next** == **head**)

Step 7: Then set **head** = **temp2 → next**, **temp1 → next** = **head** and delete **temp2**.

Deletion at the end

Step 1: Check whether list is **Empty** (**head** == **NULL**)

Step 2: If it is **Empty** then, display **'List is Empty!!! Deletion is not possible'** and terminate the function.

Step 3: If it is **Not Empty** then, define two Node pointers **'temp1'** and '**temp2'** and initialize '**temp1**' with **head**.

Step 4: Check whether list has only one Node (**temp1 → next** == **head**)

Step 5: If it is **TRUE**. Then, set **head** = **NULL** and delete **temp1**. And terminate from the function. (Setting **Empty** list condition)

Step 6: If it is **FALSE**. Then, set '**temp2 = temp1** ' and move **temp1** to its next node. Repeat the same until **temp1** reaches to the last node in the list. (until **temp1 → next** == **head**)

Step 7: Set **temp2 → next** = **head** and delete **temp1**.

Deletion at any other position

Step 1: Check whether list is **Empty** (**head** == **NULL**)

Step 2: If it is **Empty** then, display **'List is Empty!!! Deletion is not possible'** and terminate the function.

Step 3: If it is **Not Empty** then, define two Node pointers **'temp1'** and '**temp2**' and initialize '**temp1**' with **head**.

Step 4: Keep moving the **temp1** until it reaches to the exact node to be deleted or to the last node. And every time set '**temp2 = temp1**' before moving the '**temp1**' to its next node.

Step 5: If it is reached to the last node then display **'Given node not found in the list! Deletion not possible!!!'**. And terminate the function.

Step 6: If it is reached to the exact node which we want to delete, then check whether list is having only one node (**temp1 → next** == **head**)

Step 7: If list has only one node and that is the node to be deleted then set **head** = **NULL** and delete **temp1** (**free(temp1)**).

Step 8: If list contains multiple nodes then check whether **temp1** is the first node in the list (**temp1 == head**).

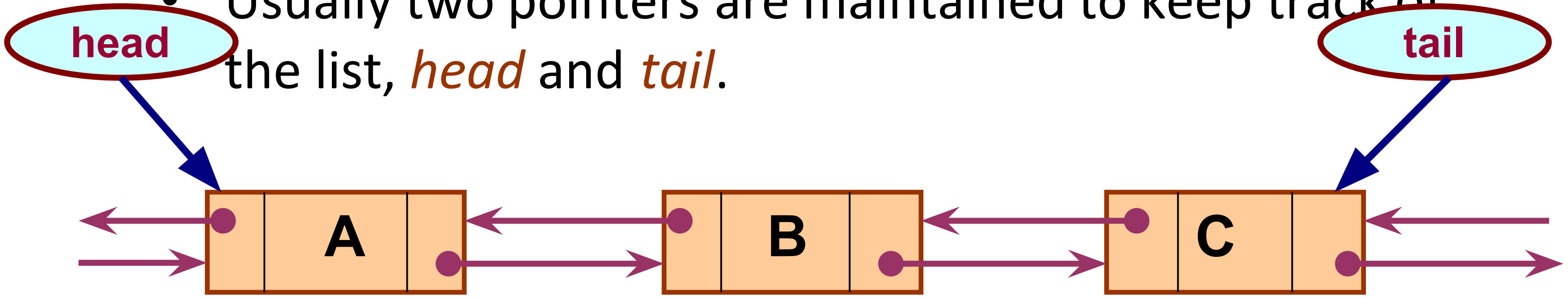
Step 9: If **temp1** is the first node then set **temp2** = **head** and keep moving **temp2** to its next node until **temp2** reaches to the last node. Then set **head = head → next**, **temp2 → nex**t = **head** and delete **temp1**.

Step 10: If **temp1** is not first node then check whether it is last node in the list (**temp1 → next == head**).

Step 11: If **temp1** is last node then set **temp2 → next** = **head** and delete **temp1** (**free(temp1)**).

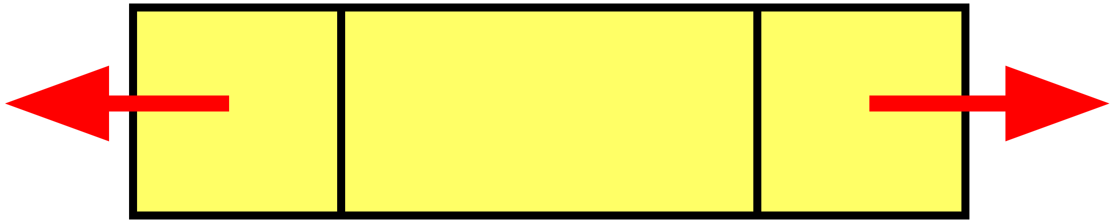
Step 12: If **temp1** is not first node and not last node then set **temp2 → next** = **temp1 → next** and delete **temp1** (**free(temp1)**).

- Doubly linked list
  - Pointers exist between adjacent nodes in both directions.
  - The list can be traversed either forward or backward.
  - Usually two pointers are maintained to keep track of the list, *head* and *tail*.



Node data

- info: the user's data
- next, back: the address of the next and previous node in the list



- In a double linked list, we perform the following operations...
- Insertion
- Deletion
- Display

- In a double linked list, the insertion operation can be performed in three ways as follows...
- Inserting At Beginning of the list
- Inserting At End of the list
- Inserting At Specific location in the list

Insertion at the begining

Step 1: Create a **newNode** with given value and **newNode → previous** as **NULL**.

Step 2: Check whether list is **Empty** (**head** == **NULL**)

Step 3: If it is **Empty** then, assign **NULL** to **newNode → next** and **newNode** to **head**.

Step 4: If it is **not Empty** then, assign **head** to **newNode → next** head-> previous to newNode and **newNode** to **head**.

Insertion at the End

Step 1: Create a **newNode** with given value and **newNode → next** as **NULL**.

Step 2: Check whether list is **Empty** (**head** == **NULL**)

Step 3: If it is **Empty**, then assign **NULL** to **newNode → previous** and **newNode** to **head**.

Step 4: If it is **not Empty**, then, define a node pointer **temp** and initialize with **head**.

Step 5: Keep moving the **temp** to its next node until it reaches to the last node in the list (until **temp → next** is equal to **NULL**).

Step 6: Assign **newNode** to **temp → next** and **temp** to **newNode → previous**.

Insertion at a specifed location

Step 1: Create a **newNode** with given value.

Step 2: Check whether list is **Empty** (**head** == **NULL**)

Step 3: If it is **Empty** then,

assign **NULL** to **newNode → previous**

& **newNode → next**

and **newNode** to **head**.

Step 4: If it is **not Empty** then,

define two node pointers **temp1** & **temp2** and
initialize **temp1** with **head**.

Step 5: Keep moving the **temp1** to its next node until it reaches to the node after which we want to insert the newNode (until **temp1 → data** is equal to **location**, here location is the node value after which we want to insert the newNode).

Step 6: Every time check whether **temp1** is reached to the last node. If it is reached to the last node then display **'Given node is not found in the list!!! Insertion not possible!!!'** and terminate the function. Otherwise move the **temp1** to next node.

Step 7: Assign

temp1 → next to **temp2**,

newNode to **temp1 → next**,

temp1 to **newNode → previous**,

temp2 to **newNode → next** and

newNode to **temp2 → previous**.

Deletion

- In a double linked list, the deletion operation can be performed in three ways
- Deleting from Beginning of the list
- Deleting from End of the list
- Deleting a Specific Node

Deleting from Beginning of the list

Step 1: Check whether list is **Empty** (**head** == **NULL**)

Step 2: If it is **Empty** then, display **'List is Empty!!! Deletion is not possible'** and terminate the function.

Step 3: If it is not Empty then, define a Node pointer **'temp'** and initialize with **head**.

Step 4: Check whether list is having only one node (**temp → previous** is equal to **temp → next**)

Step 5: If it is **TRUE**, then set **head** to **NULL** and delete **temp** (Setting **Empty** list conditions)

Step 6: If it is **FALSE**, then assign **temp → next** to **head**, **NULL** to **head → previous** and delete **temp**.

Deleting from End of the list

Step 1: Check whether list is **Empty** (**head** == **NULL**)

Step 2: If it is **Empty**, then display **'List is Empty!!! Deletion is not possible'** and terminate the function.

Step 3: If it is not Empty then, define a Node pointer **'temp'** and initialize with **head**.

Step 4: Check whether list has only one Node (**temp → previous** and **temp → next** both are **NULL**)

Step 5: If it is **TRUE**, then assign **NULL** to **head** and delete **temp**. And terminate from the function. (Setting **Empty** list condition)

Step 6: If it is **FALSE**, then keep moving **temp** until it reaches to the last node in the list. (until **temp → next** is equal to **NULL**)

Step 7: Assign **NULL** to **temp → previous → next** and delete **temp**.

Deleting a Specific Node

Step 1: Check whether list is **Empty** (**head** == **NULL**)

Step 2: If it is **Empty** then, display **'List is Empty!!! Deletion is not possible'** and terminate the function.

Step 3: If it is not Empty, then define a Node pointer **'temp'** and initialize with **head**.

Step 4: Keep moving the **temp** until it reaches to the exact node to be deleted or to the last node.

Step 5: If it is reached to the last node, then display **'Given node not found in the list! Deletion not possible!!!'** and terminate the function.

Step 6: If it is reached to the exact node which we want to delete, then check whether list is having only one node or not

Step 7: If list has only one node and that is the node which is to be deleted then
set **head** to **NULL** and delete **temp** (**free(temp)**).

Step 8: If list contains multiple nodes, then check whether **temp** is the first node in the list (**temp == head**).

Step 9: If **temp** is the first node, then move the **head** to the next node (**head = head → next**), set **head** of **previous** to **NULL** (**head → previous = NULL**) and delete **temp**.

Step 10: If **temp** is not the first node, then check whether it is the last node in the list (**temp → next == NULL**).

Step 11: If **temp** is the last node then set **temp** of **previous** of **next** to **NULL** (**temp → previous → next = NULL**) and delete **temp** .

Step 12: If **temp** is not the first node and not the last node, then
set **temp** of **previous** of **next** to **temp** of **next**
(temp → previous → next = temp → next),
temp of **next** of **previous** to **temp** of **previous**
(temp → next → previous = temp → previous)
and delete **temp** .

Adding two polynomials using Linked List

- Given two polynomial numbers represented by a linked list. Write a function that add these lists means add the coefficients who have same variable powers.
- Input: 1st number = 5x^2 + 4x^1 + 2x^0

  2nd number = 5x^1 + 5x^0

  Output: 5x^2 + 9x^1 + 7x^0
- Input: 1st number = 5x^3 + 4x^2 + 2x^0

  2nd number = 5x^1 + 5x^0

  Output: 5x^3 + 4x^2 + 5x^1 + 7x^0

```
struct Node
{
    int coeff;
    int pow;
    struct Node *next;
};
```

Reverse a singly linked list

- Given pointer to the head node of a linked list, the task is to reverse the linked list. We need to reverse the list by changing links between nodes.

- Input : Head of following linked list

  1->2->3->4->NULL
- Output : Linked list should be changed to,

  4->3->2->1->NULL
- Input : Head of following linked list

  1->2->3->4->5->NULL

  Output : Linked list should be changed to,

  5->4->3->2->1->NULL
- Input : NULL Output : NULL
- Input : 1->NULL Output : 1->NULL

- *Initialize three pointers prev as NULL, curr as head and next as NULL.*
- *Iterate trough the linked list.*
- *In loop, do following.*

  next = curr->next

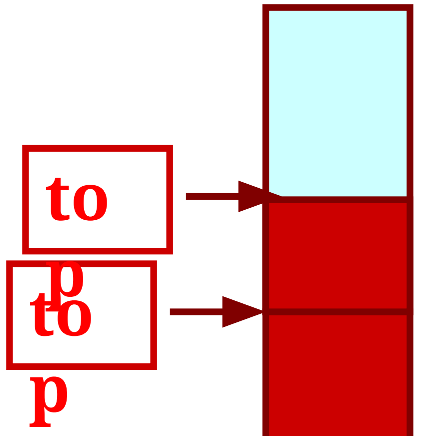
  curr->next = prev
- *Move prev and curr one step forward*

  prev = curr

  curr = next

Stack Implementations: Using Array and Linked List

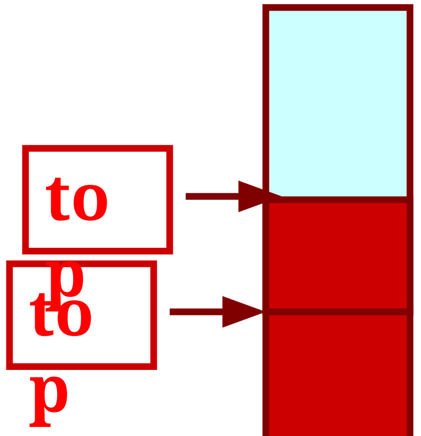
STACK USING ARRAY



PUS
H

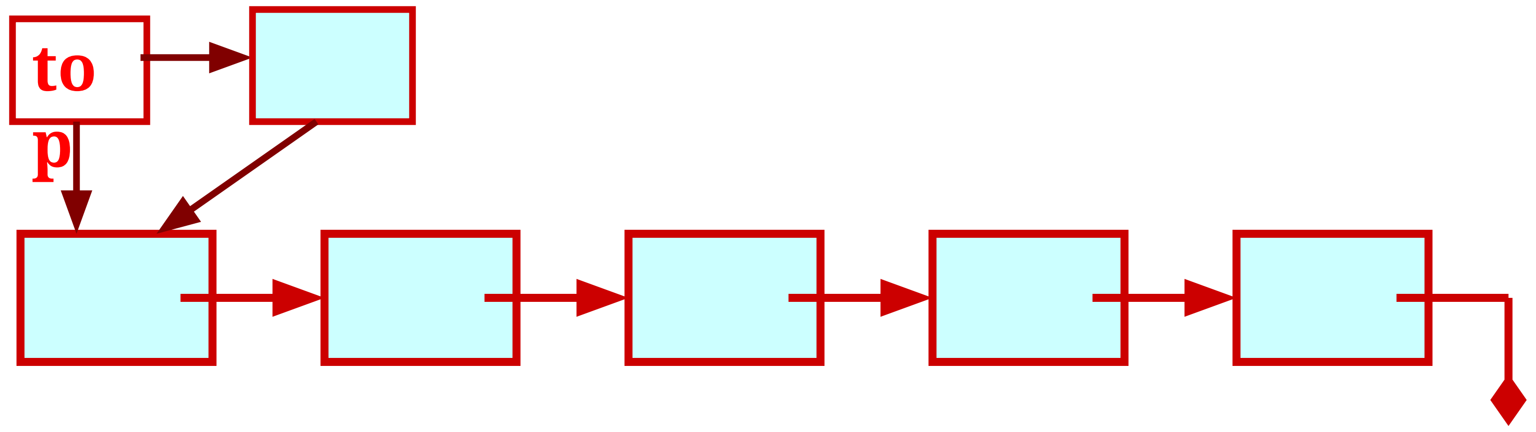
STACK USING ARRAY

PO
P



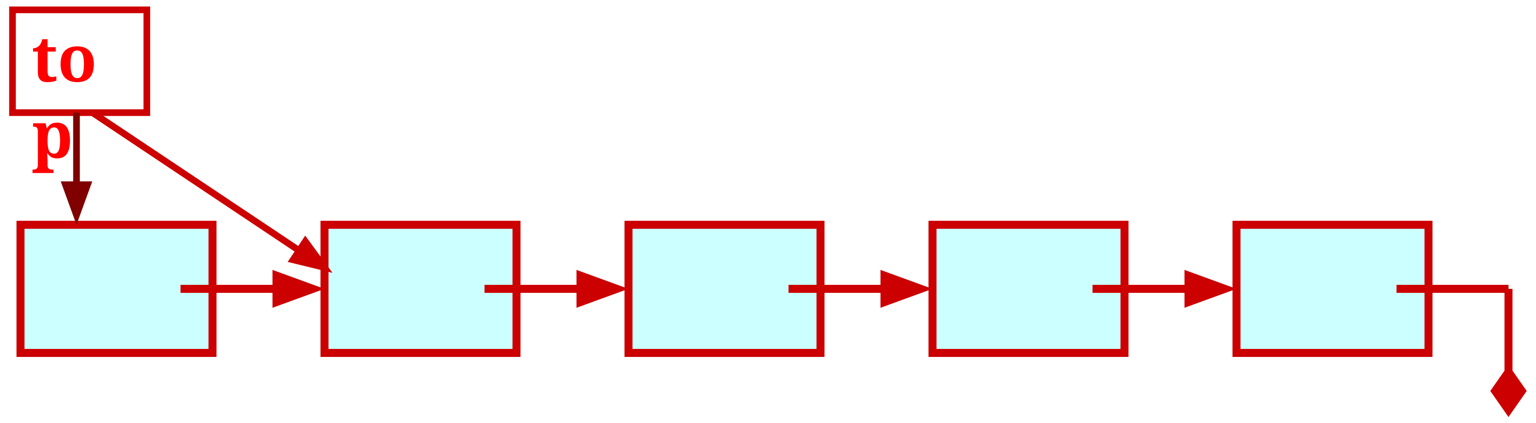
Stack: Linked List Structure

PUSH OPERATION



Stack: Linked List Structure

POP
OPERATION



Basic Idea

- We would:
  - Declare an array of fixed size (which determines the maximum size of the stack).
  - Keep a variable which always points to the “top” of the stack.
    - Contains the array index of the “top” element.

Basic Idea

- In the array implementation, we would:
  - Declare an array of fixed size (which determines the maximum size of the stack).
  - Keep a variable which always points to the “top” of the stack.
    - Contains the array index of the “top” element.
- In the linked list implementation, we would:
  - Maintain the stack as a linked list.
  - A pointer variable `top` points to the start of the list.
  - The first element of the linked list is considered as the stack top.

Declaration

```
#define MAXSIZE 100

struct lifo
{
   int st[MAXSIZE];
   int  top;
};
typedef struct lifo
                stack;
stack s;
```

ARRAY

```
struct lifo
{
   int value;
   struct lifo *next;
};
typedef struct lifo
                stack;

stack *top;
```

LINKED LIST

Stack Creation

```
void create (stack *s)
{
    s->top = -1;

    /* s->top points to

       last element
       pushed in;
       initially -1 */
}
```

ARRAY

```
void create (stack **top)
{
    *top = NULL;

    /* top points to NULL,
       indicating empty
       stack            */
}
```

LINKED LIST

Pushing an element into the stack

```
void push (stack *s, int element)
  {
     if (s->top == (MAXSIZE-1))
     {
         printf (“\n Stack overflow”);
         exit(-1);
     }
     else
     {
         s->top ++;
         s->st[s->top] = element;
     }
  }
```

ARRAY

```
void push (stack **top, int element)
{
    stack *new;

    new = (stack *) malloc(sizeof(stack));
    if (new == NULL)
    {
       printf (“\n Stack is full”);
       exit(-1);
    }

    new->value = element;
    new->next = *top;
    *top = new;
}
```

LINKED LIST

Popping an element from the stack

```
int pop (stack *s)
  {
     if (s->top == -1)
     {
        printf (“\n Stack underflow”);
        exit(-1);
     }
     else
     {
        return (s->st[s->top--]);
     }
  }
```

ARRAY

```
int pop (stack **top)
{
   int t;
   stack *p;

   if (*top == NULL)
   {
      printf ("\n Stack is empty");
      exit(-1);
   }
   else
   {
      t = (*top)->value;
      p = *top;
      *top = (*top)->next;
      free (p);
      return t;
   }
}
```

LINKED LIST

Checking for stack empty

```
int isempty (stack *s)
{
   if (s->top == -1)
          return 1;
   else
          return (0);
}
```

ARRAY

```
int isempty (stack *top)
{
   if (top == NULL)
        return (1);
    else
        return (0);
}
```

LINKED LIST

Checking for stack full

```
int isfull (stack *s)
{
   if (s->top == 

  (MAXSIZE-1))
        return 1;
   else
        return (0);
}
```

ARRAY

- Not required for linked list implementation.
- In the `push()` function, we can check the return value of `malloc()`.
  - If -1, then memory cannot be allocated.

LINKED LIST

Example main function :: array

```
#include <stdio.h>
#define MAXSIZE 100

struct lifo
{
   int st[MAXSIZE];
   int  top;
};
typedef struct lifo stack;

main()
{
  stack A, B;
  create(&A);  create(&B);
  push(&A,10);
  push(&A,20);
```

```
  push(&A,30);
  push(&B,100);  push(&B,5);

  printf (“%d %d”, pop(&A),
              pop(&B));

  push (&A, pop(&B));

  if (isempty(&B))
    printf (“\n B is empty”);
}
```

Example main function :: linked list

```
#include <stdio.h>
struct lifo
{
   int value;
   struct lifo *next;
};
typedef struct lifo stack;
```

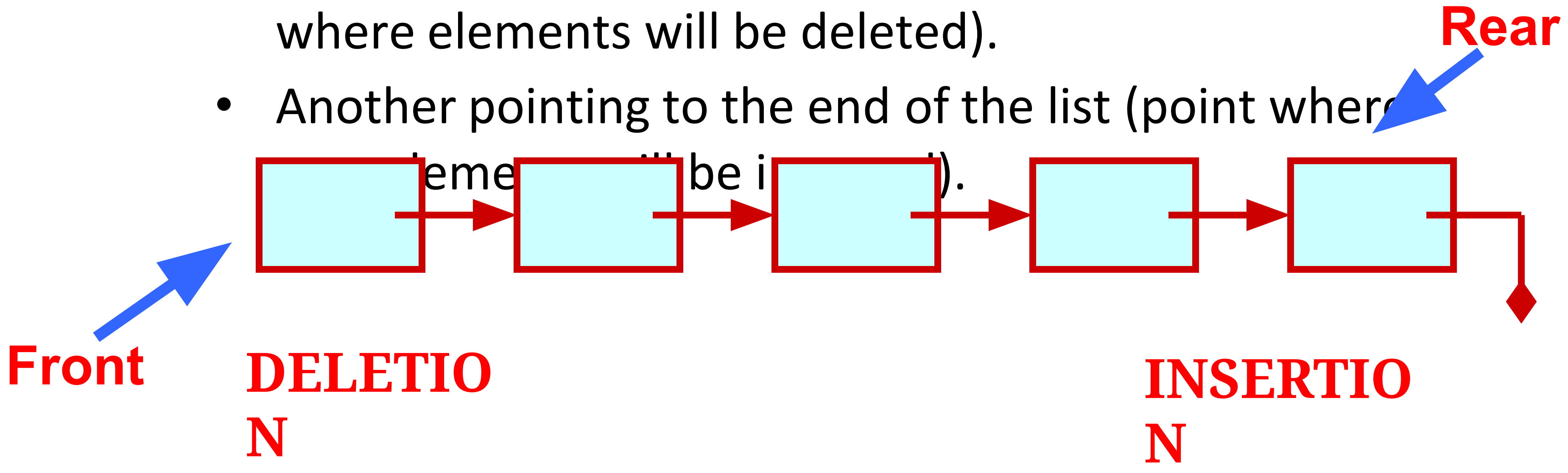
```
main()
{
  stack *A, *B;
  create(&A); create(&B);
  push(&A,10);
  push(&A,20);
```

```
  push(&A,30);
  push(&B,100);
  push(&B,5);

  printf (“%d %d”,
        pop(&A), pop(&B));

  push (&A, pop(&B));

  if (isempty(B))
    printf (“\n B is
  empty”);
}
```

Queue Implementation using Linked List

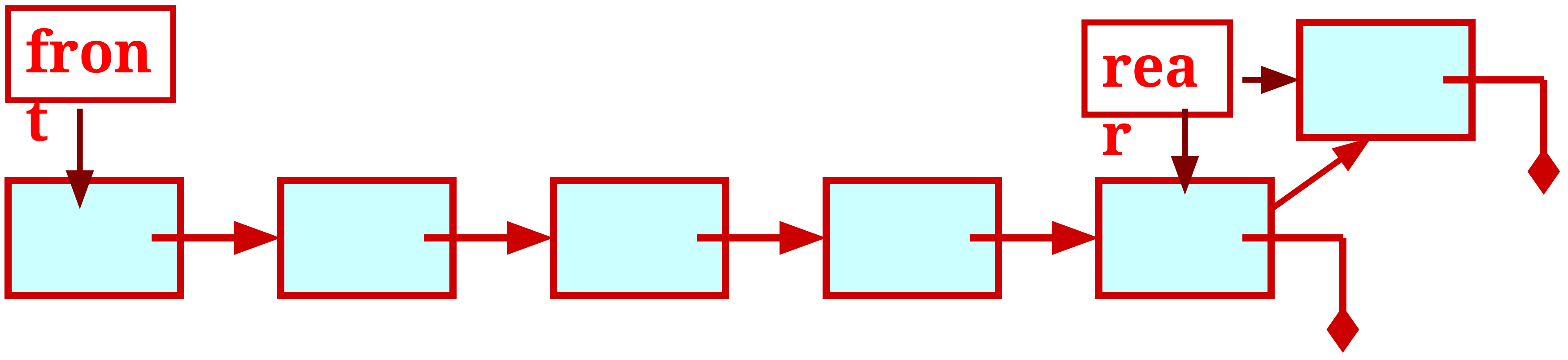
Basic Idea

- Basic idea:
  - Create a linked list to which items would be added to one end and deleted from the other end.
  - Two pointers will be maintained:
    - One pointing to the beginning of the list (point from where elements will be deleted).
    - Another pointing to the end of the list (point where [illegible]leme[illegible]ll be i[illegible]).



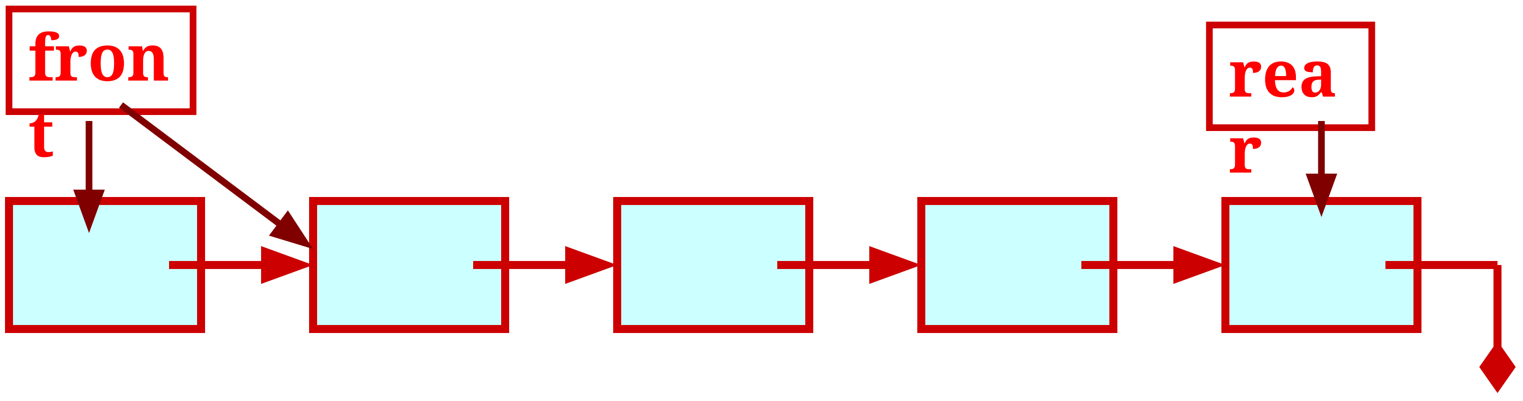
QUEUE: LINKED LIST STRUCTURE

ENQUEU
E



QUEUE: LINKED LIST STRUCTURE

DEQUEU
E



QUEUE using Linked List

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

struct node{
              char name[30];
              struct node *next;
           };

typedef struct node _QNODE;

typedef struct {
     _QNODE *queue_front, *queue_rear;
      } _QUEUE;
```

```
_QNODE *enqueue (_QUEUE *q,
char x[])
{
_QNODE *temp;
temp= (_QNODE *)
        malloc (sizeof(_QNODE))
if (temp==NULL){
printf("Bad allocation \n");
return NULL;
}
strcpy(temp->name,x);
temp->next=NULL;
```

```
if(q-
>queue_rear==NULL)
{
q->queue_rear=temp;
q->queue_front=
     q->queue_rear;
}
else
{
q->queue_rear-
>next=temp;
q->queue_rear=temp;
}
return(q->queue_rear);
```

```
char  *dequeue(_QUEUE *q,
char x[])
{
_QNODE *temp_pnt;

if(q->queue_front==NULL){
q->queue_rear=NULL;
printf("Queue is empty \n");
return(NULL);
}
```

```
else{
strcpy(x,q->queue_front-
>name);
temp_pnt=q->queue_front;
q->queue_front=
        q->queue_front->next;
free(temp_pnt);
if(q->queue_front==NULL)
q->queue_rear=NULL;
return(x);
  }
}
```

```
void init_queue(_QUEUE *q)
{
 q->queue_front= q-
>queue_rear=NULL;
}

int isEmpty(_QUEUE *q)
{
 if(q==NULL) return 1;
 else return 0;
}
```

```
main()
{
int i,j;
char command[5],val[30];
_QUEUE q;

 init_queue(&q);

command[0]='\0';
printf("For entering a name use 'enter
<name>'\n");
printf("For  deleting  use 'delete' \n");
printf("To end the session use 'bye' \n");
while(strcmp(command,"bye")){
scanf("%s",command);
```

```
if(!strcmp(command,"enter")) {
scanf("%s",val);
if((enqueue(&q,val)==NULL))
printf("No more pushing please
\n");
else printf("Name entered %s
\n",val);
}
```

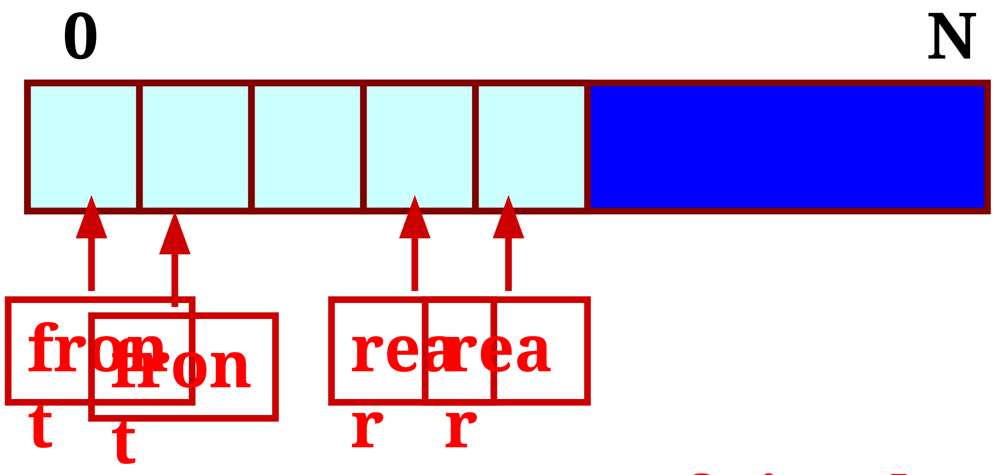
```
if(!strcmp(command,"delete"))
{
if(!isEmpty(&q))
printf("%s \n",dequeue(&q,val));

else printf("Name deleted %s
\n",val);
}
} /* while */
```

Problem With Array Implementation

ENQUEU
E

DEQUEU
E

Effective queuing storage area of array gets reduced.



Use of circular array indexing

Queue: Example with Array Implementation

```
#define MAX_SIZE
100

typedef struct { char name[30];
            } _ELEMENT;

 typedef struct {
            _ELEMENT q_elem[MAX_SIZE];
             int rear;
             int front;
             int full,empty;
             } _QUEUE;
```

Queue Example: Contd.

```
void init_queue(_QUEUE
*q)
{q->rear= q->front= 0;
 q->full=0; q->empty=1;
}

 int  IsFull(_QUEUE *q)
{return(q->full);}

 int  IsEmpty(_QUEUE
*q)
{return(q->empty);}
```

Queue Example: Contd.

```
 void AddQ(_QUEUE *q, _ELEMENT ob)
 {
   if(IsFull(q)) {printf("Queue is Full \n");
return;}

   q->rear=(q->rear+1)%(MAX_SIZE);
   q->q_elem[q->rear]=ob;

   if(q->front==q->rear) q->full=1; else q-
>full=0;
   q->empty=0;

  return;
 }
```

Queue Example: Contd.

```
 _ELEMENT  DeleteQ(_QUEUE *q)
  {
   _ELEMENT temp;
    temp.name[0]='\0';

      if(IsEmpty(q)) {printf("Queue is EMPTY\n");
return(temp);}

      q->front=(q->front+1)%(MAX_SIZE);
      temp=q->q_elem[q->front];

      if(q->rear==q->front) q->empty=1; else q->empty=0;
      q->full=0;

    return(temp);
  }
```

Queue Example: Contd.

```
#include <stdio.
h>
#include
<stdlib.h>
#include
<string.h>
```

```
main()
{
int i,j;
char command[5];
_ELEMENT ob;
_QUEUE A;

  init_queue(&A);

  command[0]='\0';
  printf("For adding a name use 'add
[name]'\n");
  printf("For deleting  use 'delete' \n");
  printf("To end the session use 'bye' \n");
```

Queue Example: Contd.

```
while (strcmp(command,"bye")!=0){
    scanf("%s",command);

      if(strcmp(command,"add")==0) {
        scanf("%s",ob.name);
        if (IsFull(&A))
       printf("No more insertion please
\n");
       else {
        AddQ(&A,ob);
         printf("Name inserted %s \n",ob.
name);
              }
                                        }
```

Queue Example: Contd.

```
      if (strcmp(command,"delete")==0) {
          if (IsEmpty(&A))
              printf("Queue is empty \n");
              else {
                      ob=DeleteQ(&A);
                       printf("Name deleted %s \n",ob.
name);
                      }
                                                   }
         }  /* End of while */
  printf("End session \n");
}
```

Concatenation of 2 linked lists

1)If head of the first link list exists and second does not then we can simply point the starting pointer of the resulting list to starting node of first link list.

2) If head of the second link list exists and first does not then we can simply point the starting pointer of the resulting list to starting node of second link list.

3)If both link list exist , then

1) make the starting pointer of the resulting list to head or starting node of first link list

(2)traverse the entire first link list.

(3) last node of first link list point it to the starting node of second link list

Merge 2 sorted linked lists

(1) Create a new head pointer to an empty linked list.

(2) Check the first value of both linked lists.

(3) Whichever node from L1 or L2 is smaller, append it to the new list and move the pointer to the next node.

(4) Continue this process until you reach the end of a linked list.

Example

- L1 = 1 -> 3 -> 10   L2 = 5 -> 6 -> 9
- L3 = null

Compare the first two nodes in both linked lists: (1, 5), 1 is smaller so add it to the new linked list and move the pointer in L1.

- L1 = 3 -> 10   L2 = 5 -> 6 -> 9
- L3 = 1

Compare the first two nodes in both linked lists: (3, 5), 3 is smaller so add it to the new linked list and move the pointer in L1

- . L1 = 10 L2 = 5 -> 6 -> 9 L3 = 1 ->

Compare the first two nodes in both linked lists: (10, 5), 5 is smaller so add it to the new linked list and move the pointer in L2.

- L1 = 10   L2 = 6 -> 9
- L3 = 1 -> 3 -> 5

Compare the first two nodes in both linked lists: (10, 6), 6 is smaller so add it to the new linked list and move the pointer in L2

- . L1 = 10 L2 = 9
- L3 = 1 -> 3 -> 5 -> 6

Compare the first two nodes in both linked lists: (10, 9), 9 is smaller so add it to the new linked list and move the pointer in L2.

- L1 = 10 L2 = null
- L3 = 1 -> 3 -> 5 -> 6 -> 9

Because L2 points to null, simply append the rest of the nodes from L1 and we have our merged linked list.

- L3 = 1 -> 3 -> 5 -> 6 -> 9 -> 10

Using Recursion

```
// Data Structure to store a linked list node
struct Node
{
    int data;
    struct Node* next;
};
```

- // Takes two lists sorted in increasing order, and merge their nodes together to make one big sorted list which is returned

```
struct Node* SortedMerge(struct Node* a, struct Node* b)
{
 struct Node* result = NULL;
 // Base cases
    if (a == NULL)
        return b;
        else if (b == NULL)
        return a;
       // Pick either a or b, and recur
    if (a->data <= b->data)
    {
        result = a;
        result->next = SortedMerge(a->next, b);
    }
```

-

```
else
    {
        result = b;
        result->next = SortedMerge(a, b->next);
    }

    return result;
}
```

Using Reference

```
struct Node* SortedMerge  (struct Node* a, struct Node* b)
{
   struct Node* result = NULL;
   struct Node** lastPtrRef = &result; // point to the last result pointer

   while (1)
   {
      if (a == NULL)
      {
         *lastPtrRef = b;
         break;
      }
      else if (b == NULL)
      {
         *lastPtrRef = a;
         break;
      }
```

```
 if (a->data  <=  b->data)
          MoveNode(lastPtrRef, &a);
       else
          MoveNode(lastPtrRef, &b);

 // tricky: advance to point to the next ".next"
field
       lastPtrRef = &((*lastPtrRef)->next);
    }
```

- return result;
- }

```c
void MoveNode(struct Node** destRef, struct Node** sourceRef)
{
  /* the front source node  */
  struct Node* newNode = *sourceRef;
  assert(newNode != NULL);

  /* Advance the source pointer */
  *sourceRef = newNode->next;

  /* Link the old dest off the new node */
  newNode->next = *destRef;

  /* Move dest to point to the new node */
  *destRef = newNode;
}
```

MoveNode() function takes the node from the front of the

source, and move it to the front of the dest.

It is an error to call this with the source list empty.

Before calling MoveNode():

source == {1, 2, 3}

dest == {1, 2, 3}

Affter calling MoveNode():

source == {2, 3}

dest == {1, 1, 2, 3} */