Linked List

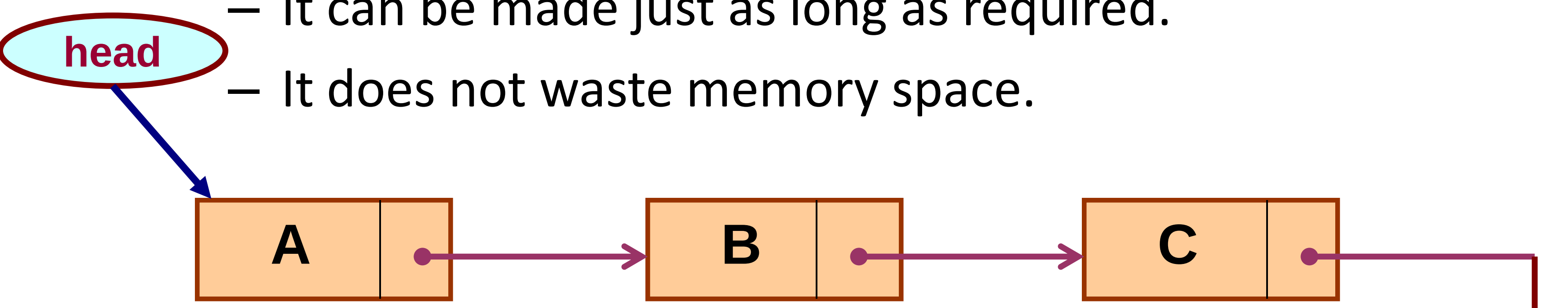
Arrays
Fixed size: Resizing is expensive int a[10];
Insertions and Deletions are inefficient: Elements are usually shifted
Random access i.e., efficient indexing
No memory waste if the array is full or almost full; otherwise may result in much memory waste.
Sequential access is faster [Reason: Elements in contiguous memory locations]

Static & Dynamic Memory Management

- Static Memory Management means allocation and deallocation of memory at compilation time.
- Dynamic memory management refers to allocating and deallocating of memory while program is running(after compilation)
- Advantage of dynamic memory management in handling linklist is that we can create as many nodes as we desire and if some nodes are not required we can deallocate them. Such a deallocated memory can be reallocated for some other nodes.
- Thus the total memory utilization is possible using dynamic memory management

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.

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.

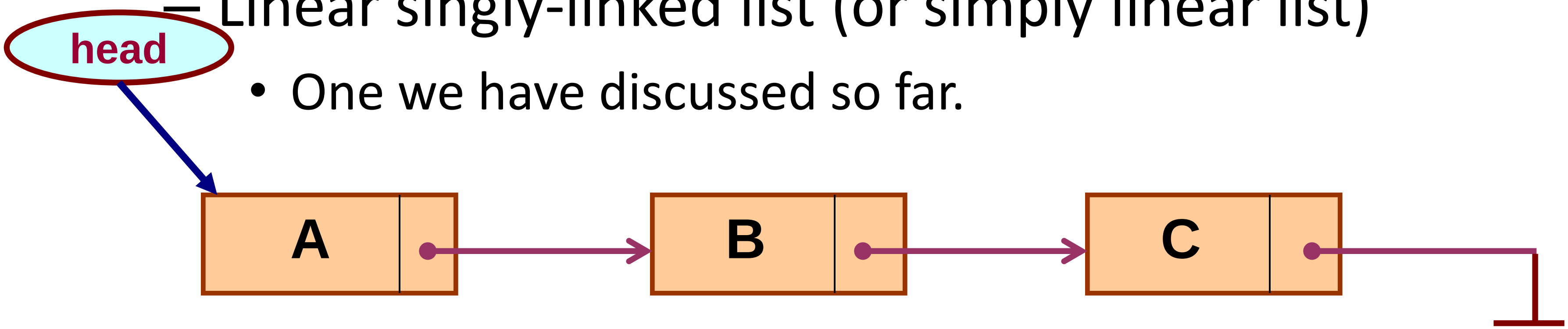
Array versus Linked Lists

- 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.

Types of Lists

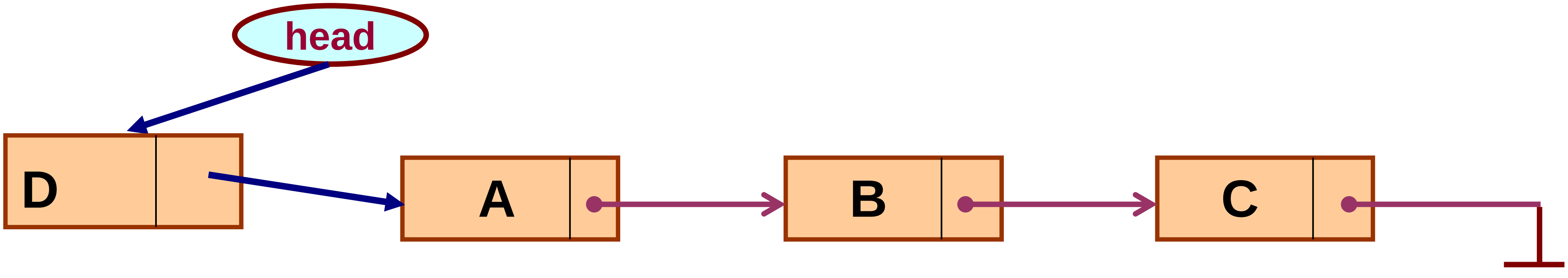
- Depending on the way in which the links are used to maintain adjacency, several different types of linked lists are possible.

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



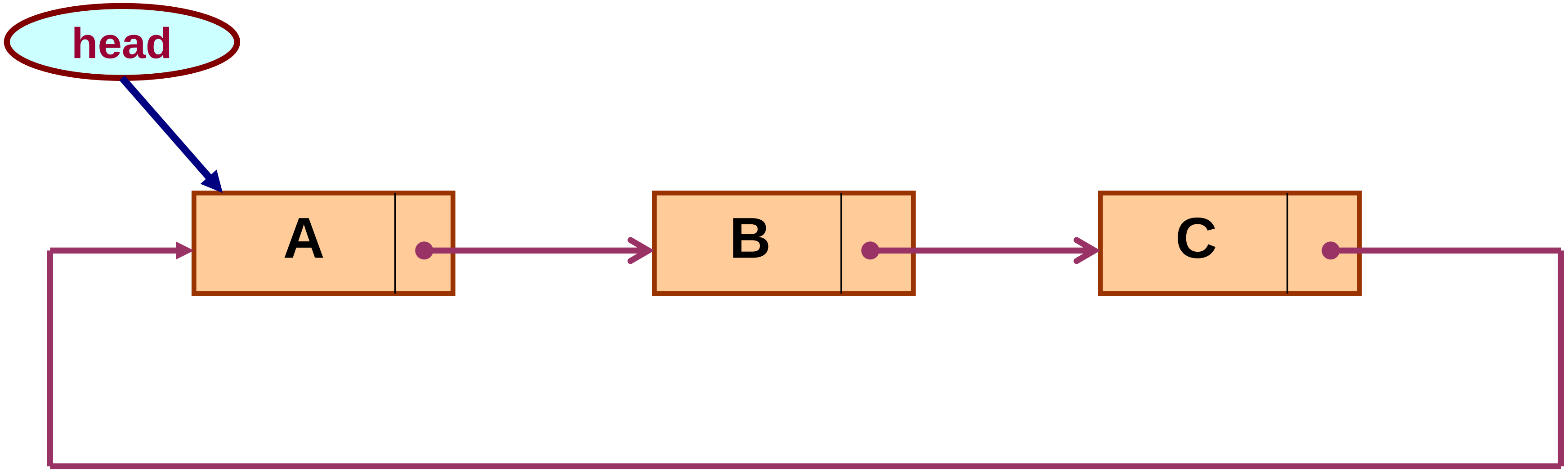
Insertion

- Insert at beginning
- Insert at middle
- Insert at end
- newNode->forw=head; head=NewNode



– Circular linked list

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



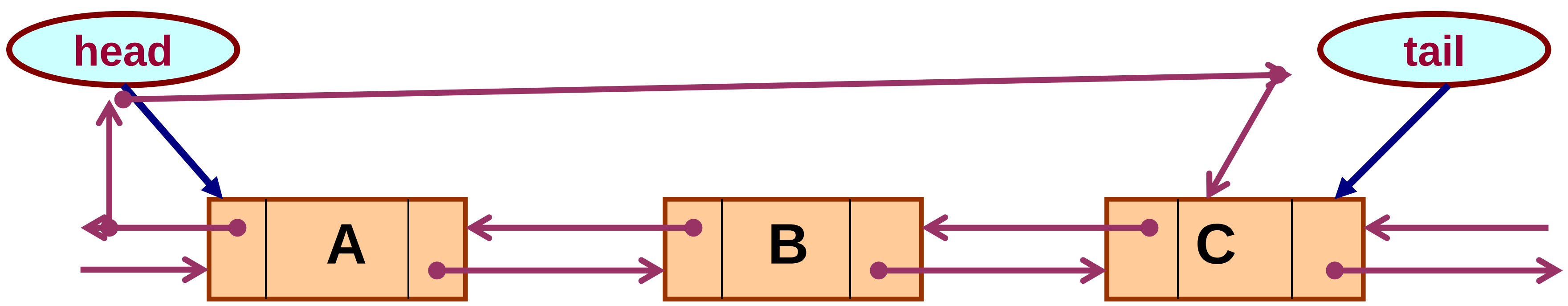
– Circular Doubly linked list

– Prev, forw

A->prev=C

C->forw=A

- 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*.



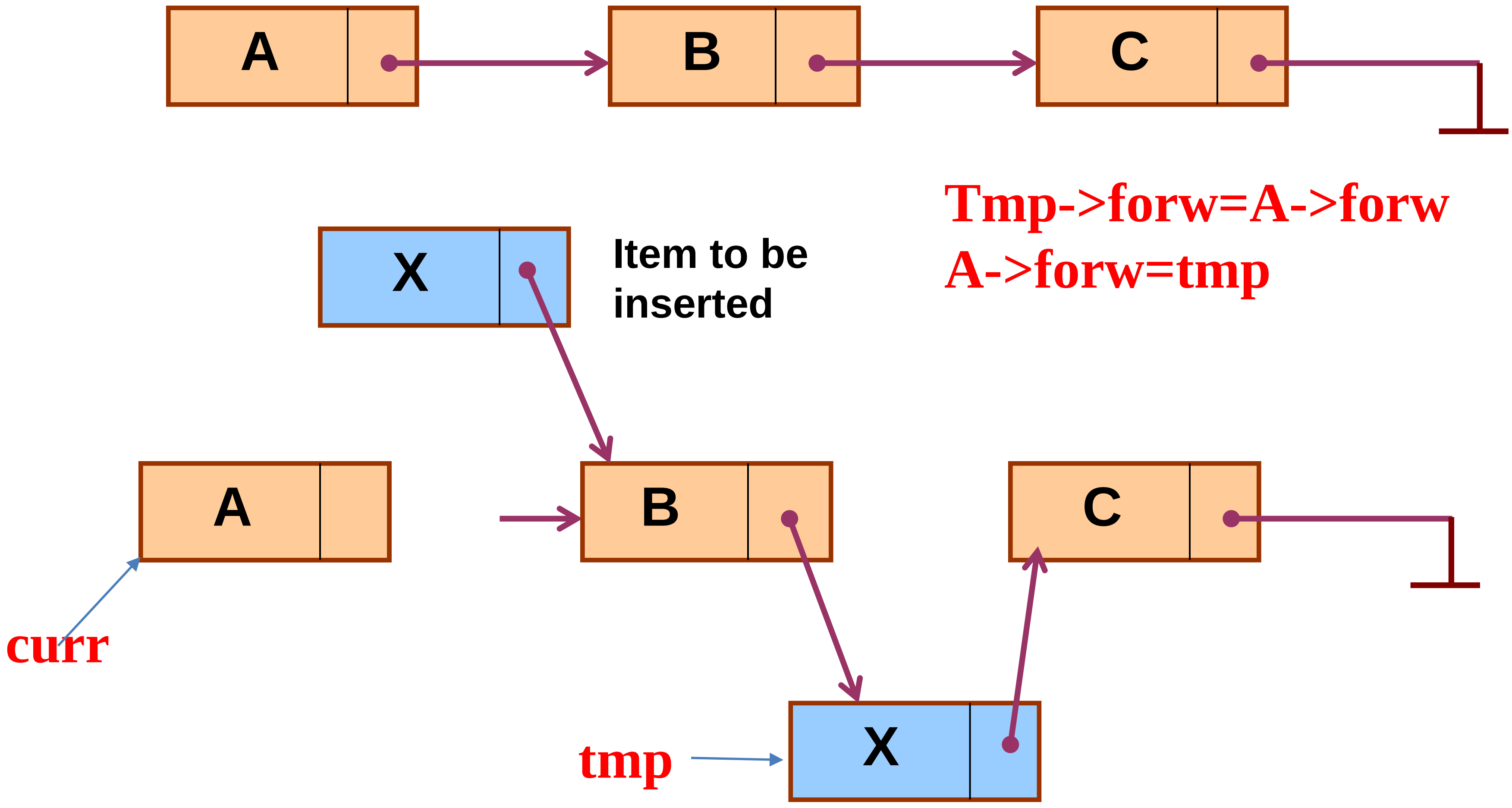
Basic Operations on a List

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

List is an Abstract Data Type

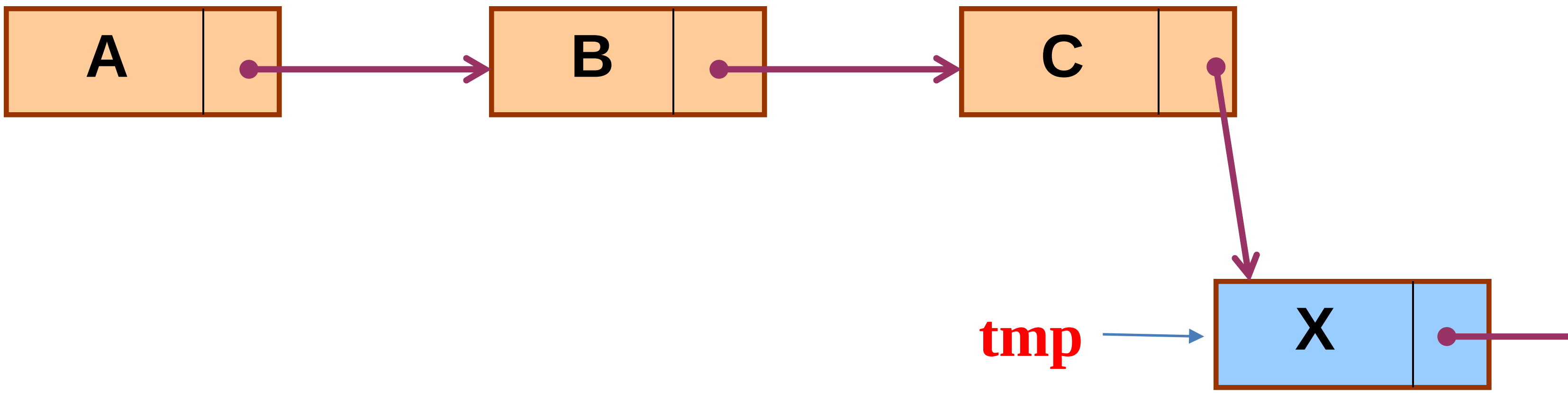
- What is an abstract data type?
  - It is a data type defined by the user.
  - Typically more complex than simple data types like *int*, *float*, etc.
- Why abstract?
  - Because details of the implementation are hidden.
  - When you do some operation on the list, say insert an element, you just call a function.
  - Details of how the list is implemented or how the insert function is written is no longer required.

Illustration: Insertion



Tmp->forw=A->forw
A->forw=tmp

Illustration: Insertion at end



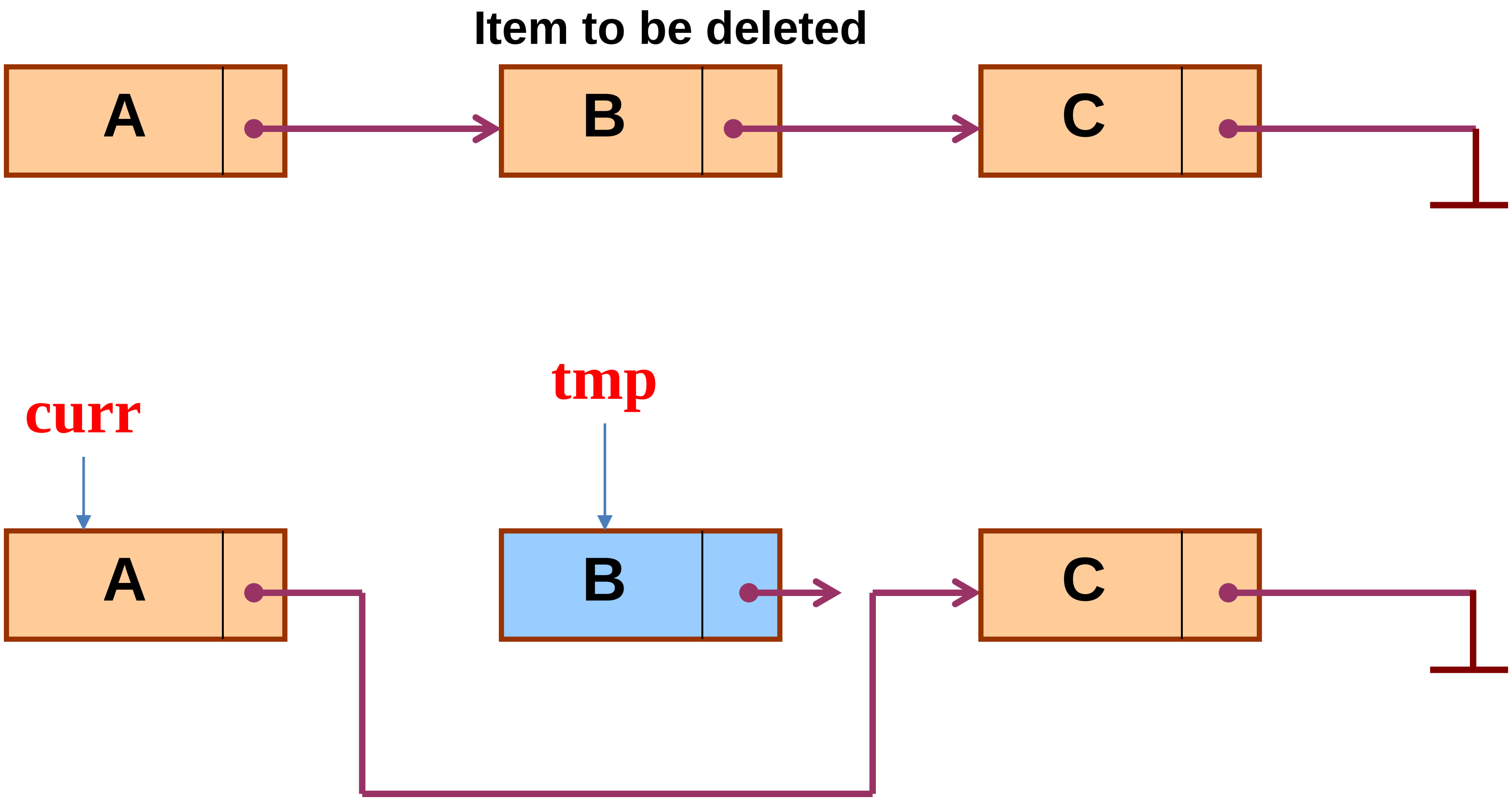
Tmp->next=NULL;
C->next=tmp;

Pseudo-code for insertion

```
typedef struct nd {
   int Roll_NO;
   struct nd * next;
   } node;

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

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

```
struct nd
{
  int rollno;
  struct nd* next;
}

DLL
struct nd
{
struct nd *prev;
int data;
struct nd *next;
}
```

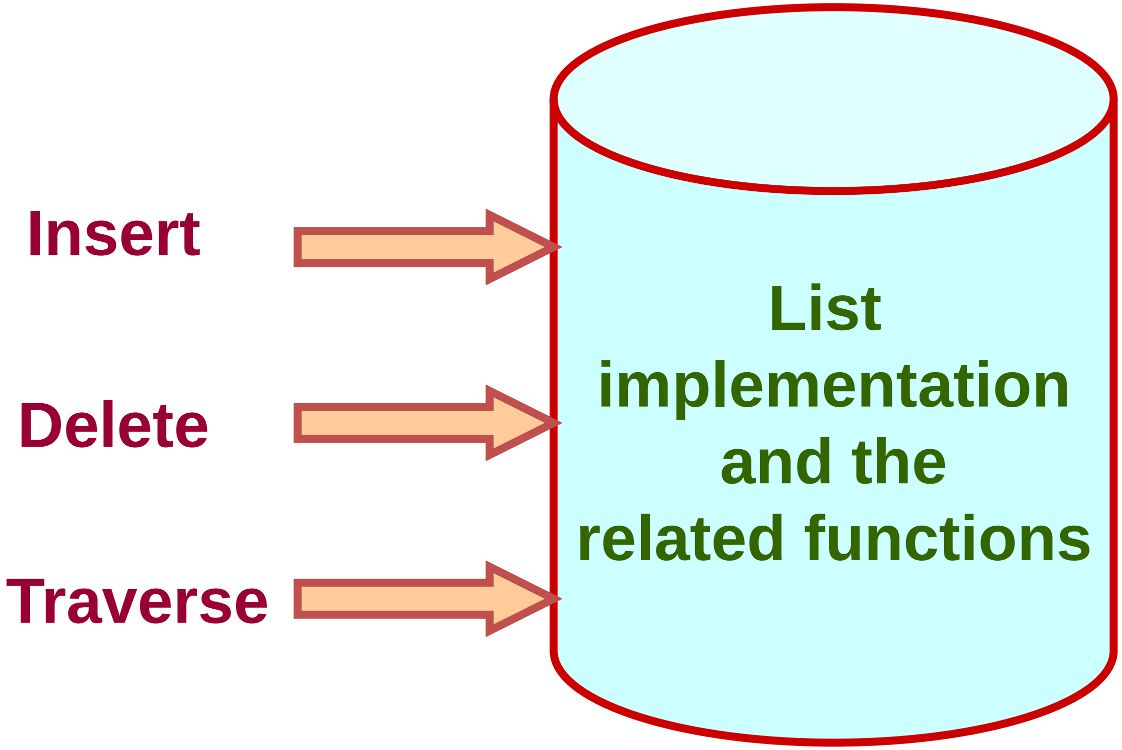
Illustration: Deletion



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);
}
```

Conceptual Idea



Example: Working with linked list

- Consider the structure of a node as follows:

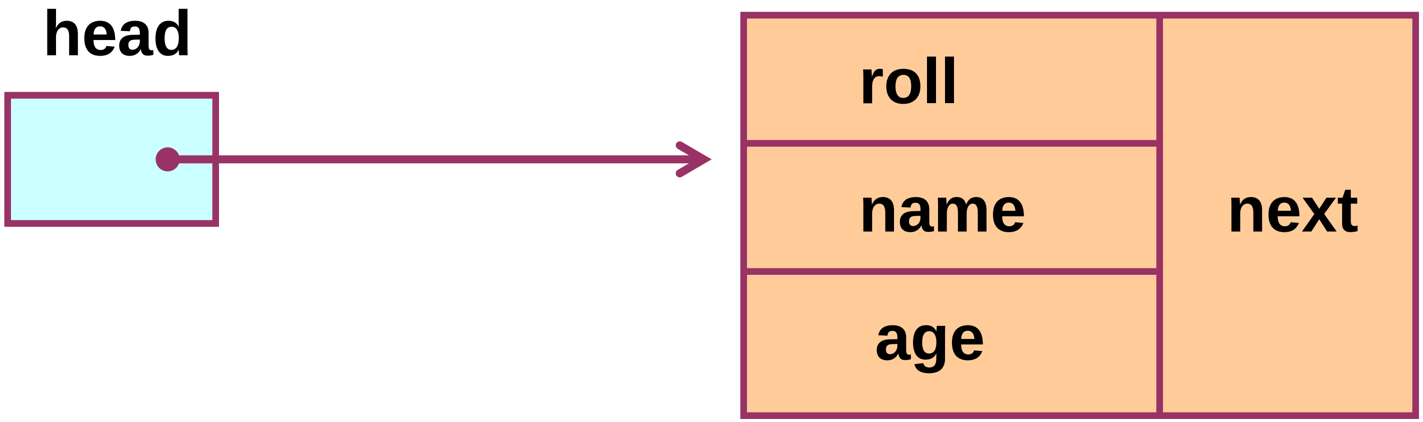
```
struct stud {
              int   roll;
              char  name[25];
              int   age;
              struct stud *next;
            };

  /* A user-defined data type called "node" */
typedef struct stud node;
node *head;
```

Creating a List

How to begin?

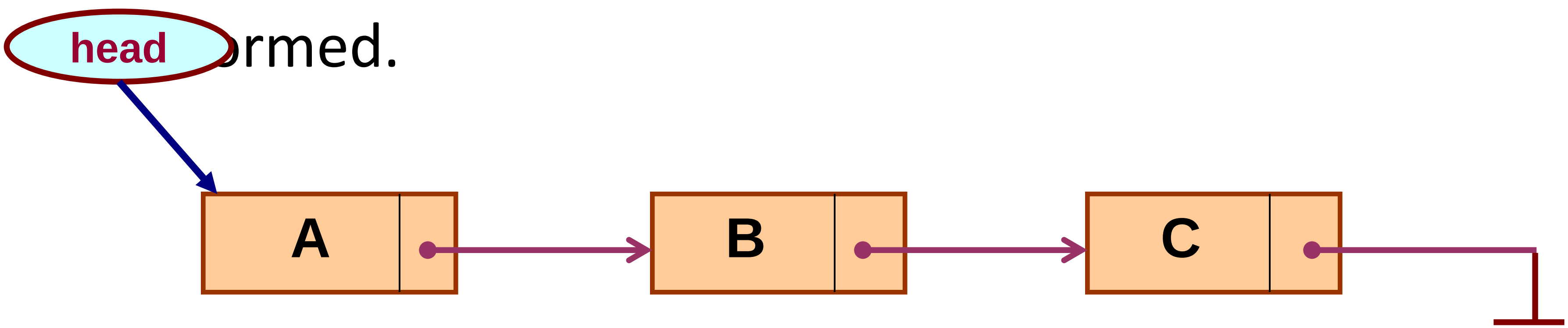
- To start with, we have to create a node (the first node), and make `head` point to it.

```
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 is 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.

```c
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);
             }
    }
}
```

Few Exercises to Try Out

- Write a function to:
  - Concatenate two given list into one big list.

    ```
    node  *concatenate (node *head1, node *head2);
    ```

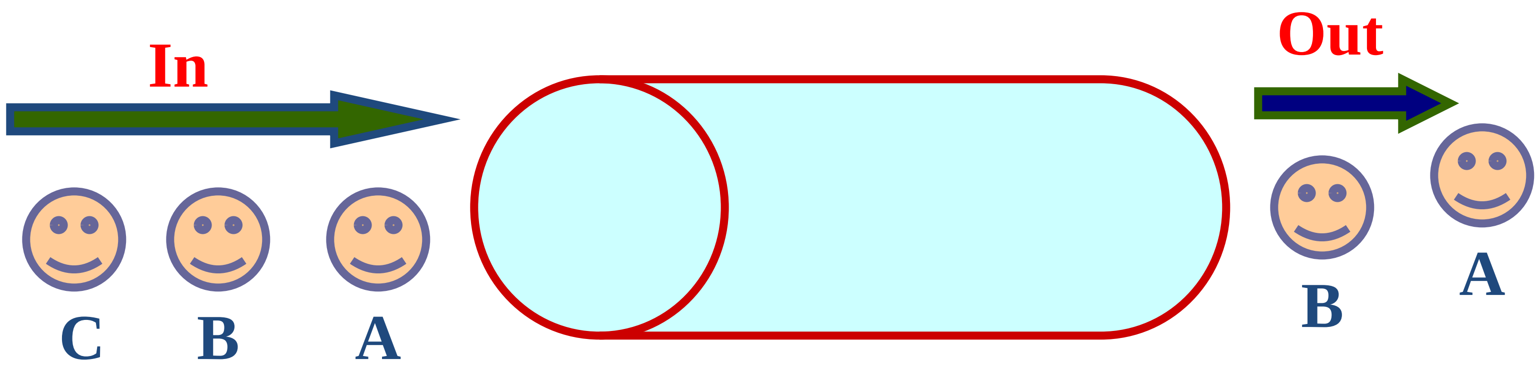
  - Insert an element in a linked list in sorted order. The function will be called for every element to be inserted.

    ```
    void  insert_sorted (node **head,  node *element);
    ```

  - Always insert elements at one end, and delete elements from the other end (first-in first-out QUEUE).

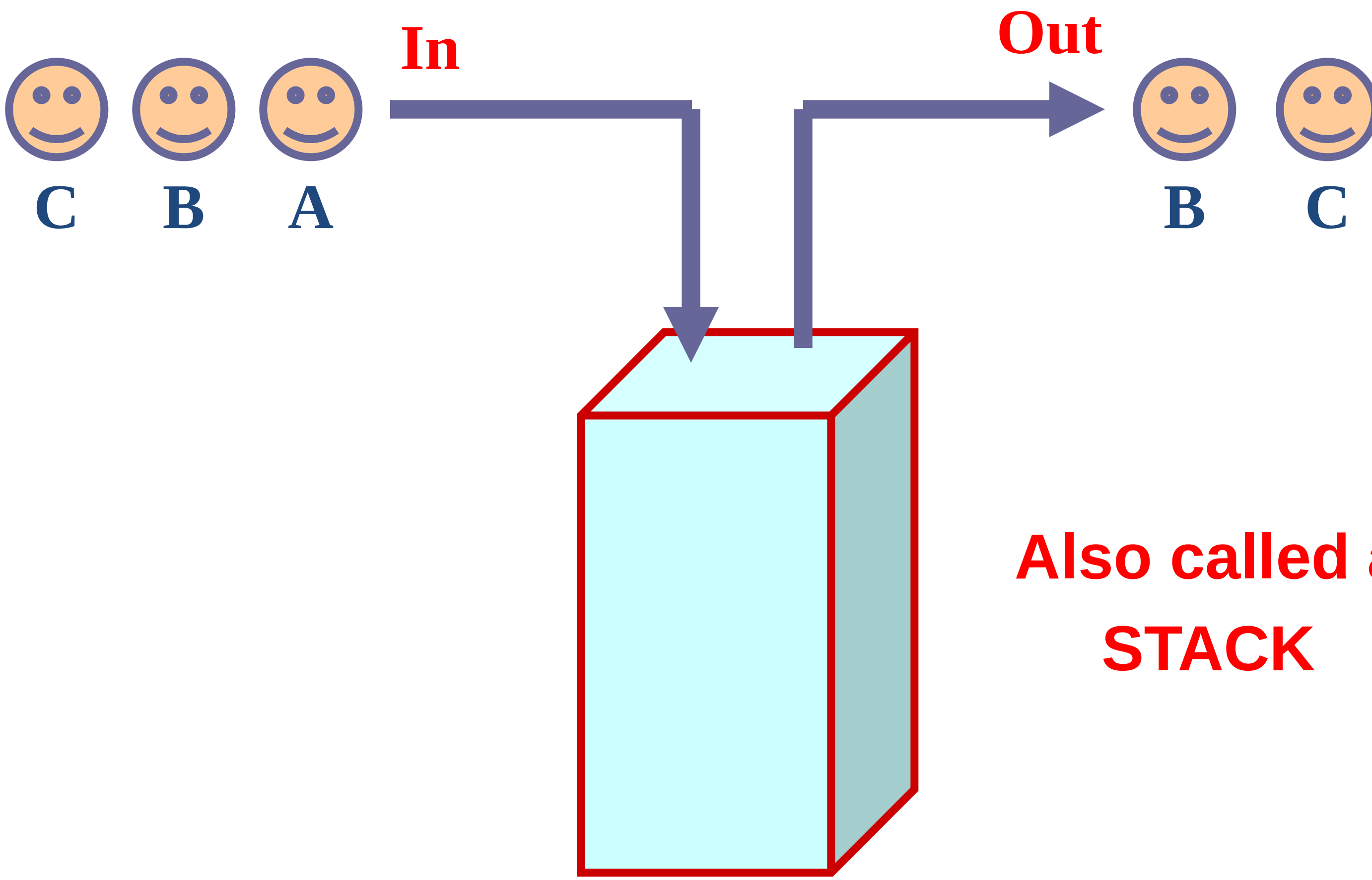
    ```
    void  insert_q (node **head,  node *element)
    node  *delete_q (node **head)   /* Return the deleted node */
    ```

A First-in First-out (FIFO) List



Also called a QUEUE

A Last-in First-out (LIFO) List



Also called a STACK

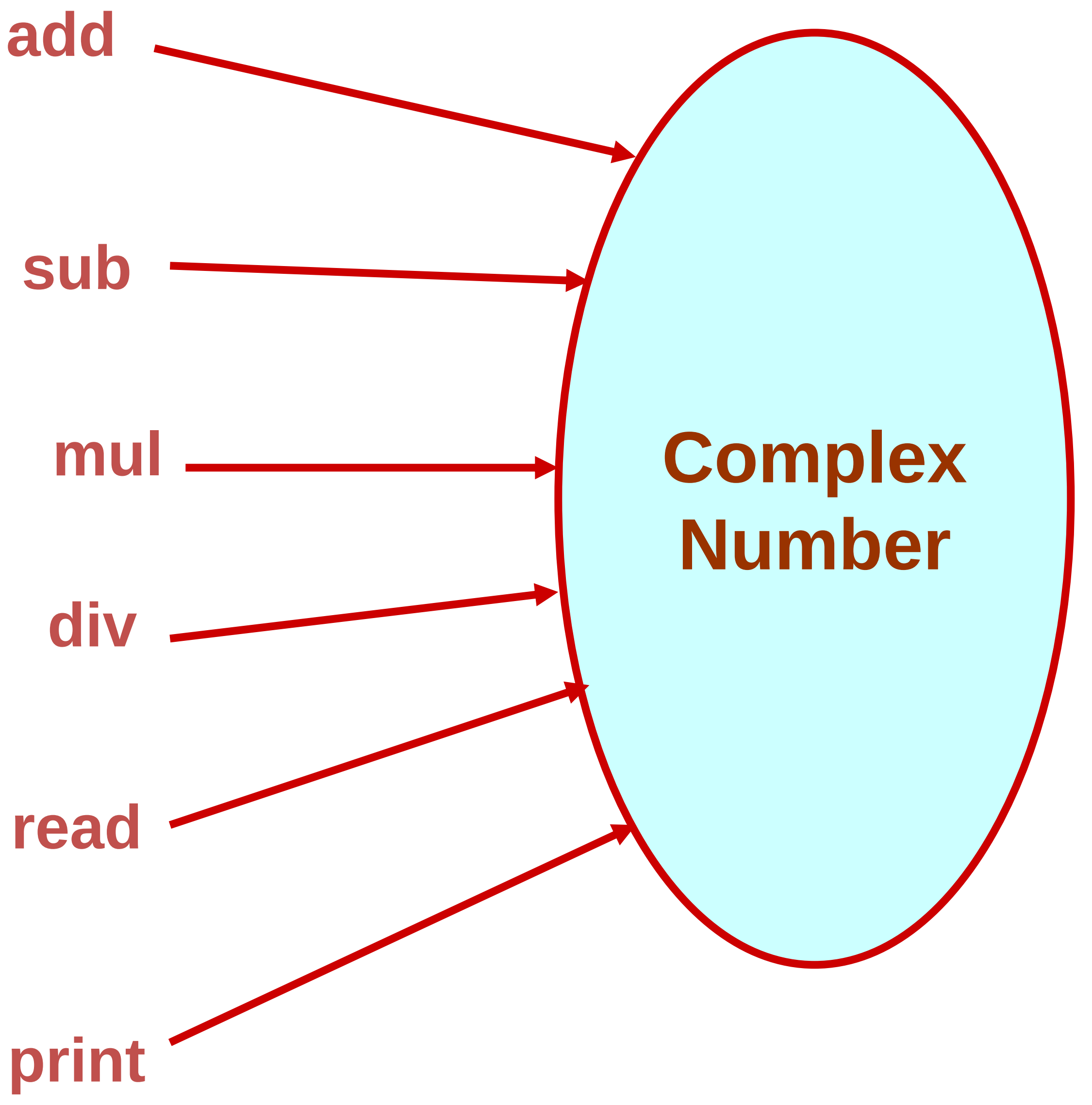
Abstract Data Types

Example 1 :: Complex numbers

```
struct cplx {
              float  re;
              float  im;
            }
typedef struct cplx complex;

complex *add (complex a, complex b);
complex *sub (complex a, complex b);
complex *mul (complex a, complex b);
complex *div (complex a, complex b);
complex *read();
void print (complex a);
```

Structure definition

Function prototypes

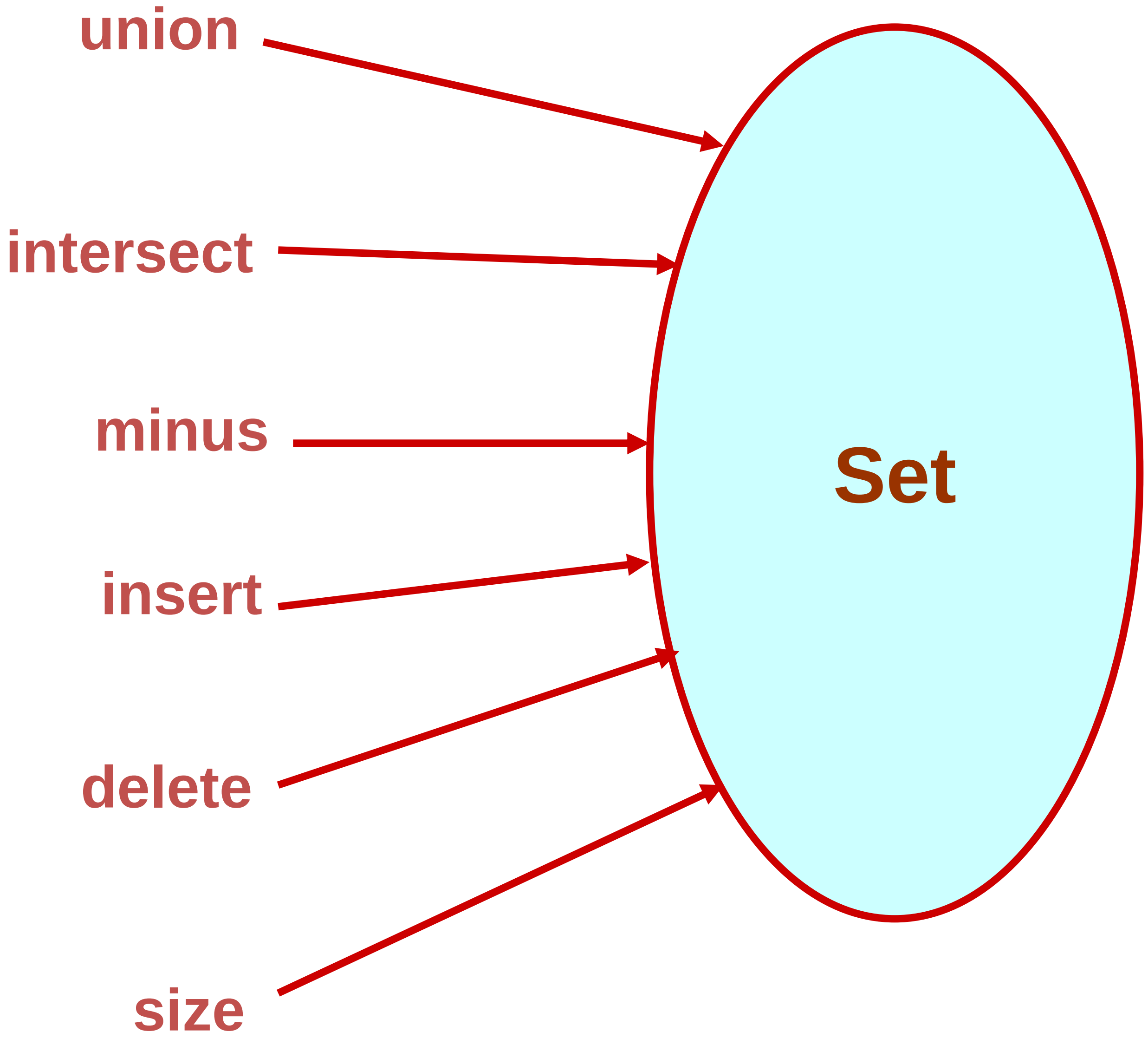

add
sub
mul
div
read
print
Complex Number

Example 2 :: Set manipulation

```
struct node {
              int element;
              struct node *next;
            }
typedef struct node set;

set *union (set a, set b);
set *intersect (set a, set b);
set *minus (set a, set b);
void insert (set a, int x);
void delete (set a, int x);
int size (set a);
```

Structure definition

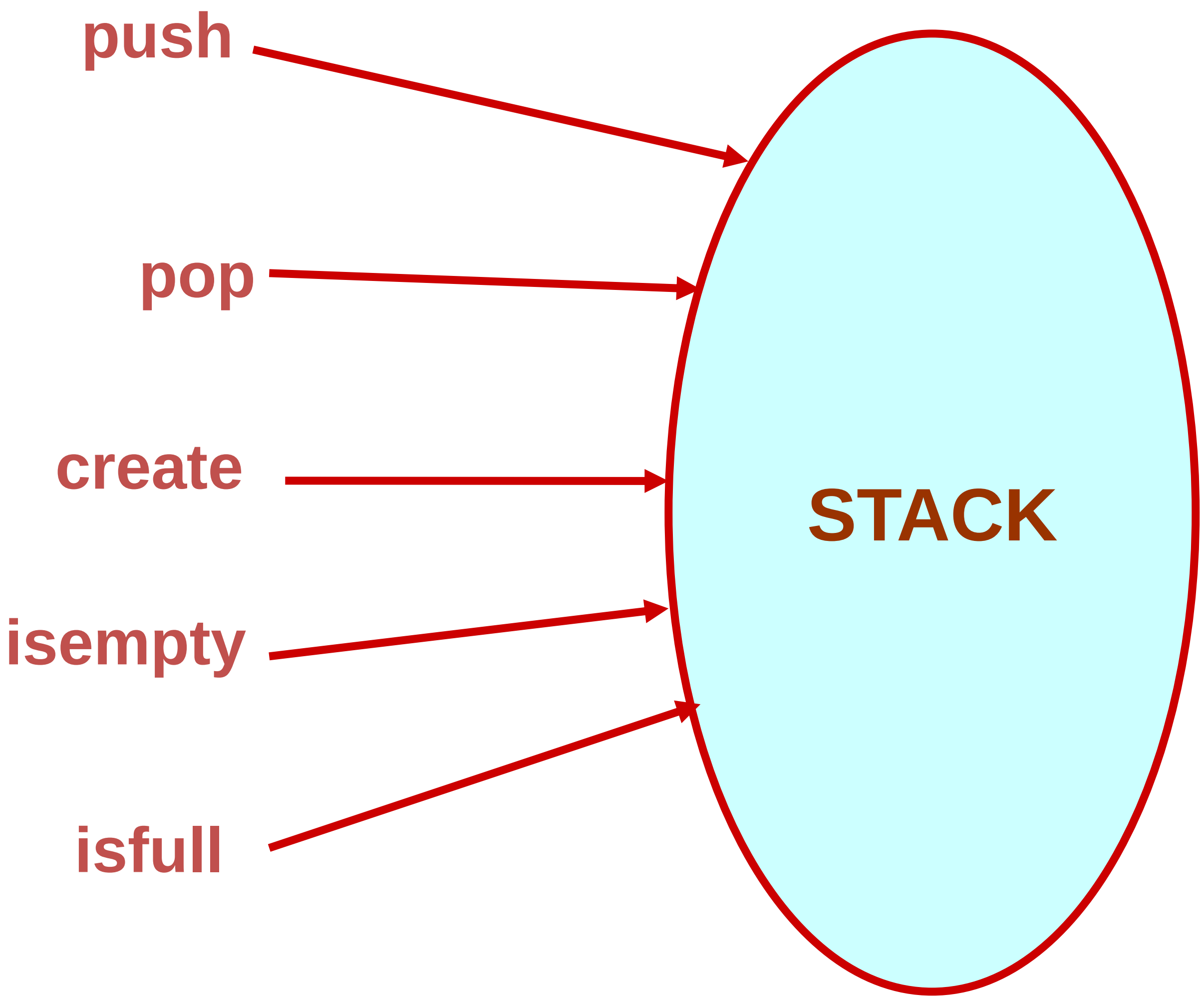
Function prototypes


union
intersect
minus
insert
delete
size
Set

Example 3 :: Last-In-First-Out STACK

Assume:: stack contains integer elements

```
void push (stack *s, int element);
                    /* Insert an element in the stack */
int pop (stack *s);
                    /* Remove and return the top element */
void create (stack  *s);
                    /* Create a new stack */
int isempty (stack *s);
                    /* Check if stack is empty */
int isfull (stack *s);
                    /* Check if stack is full */
```


push
pop
create
isempty
isfull
STACK

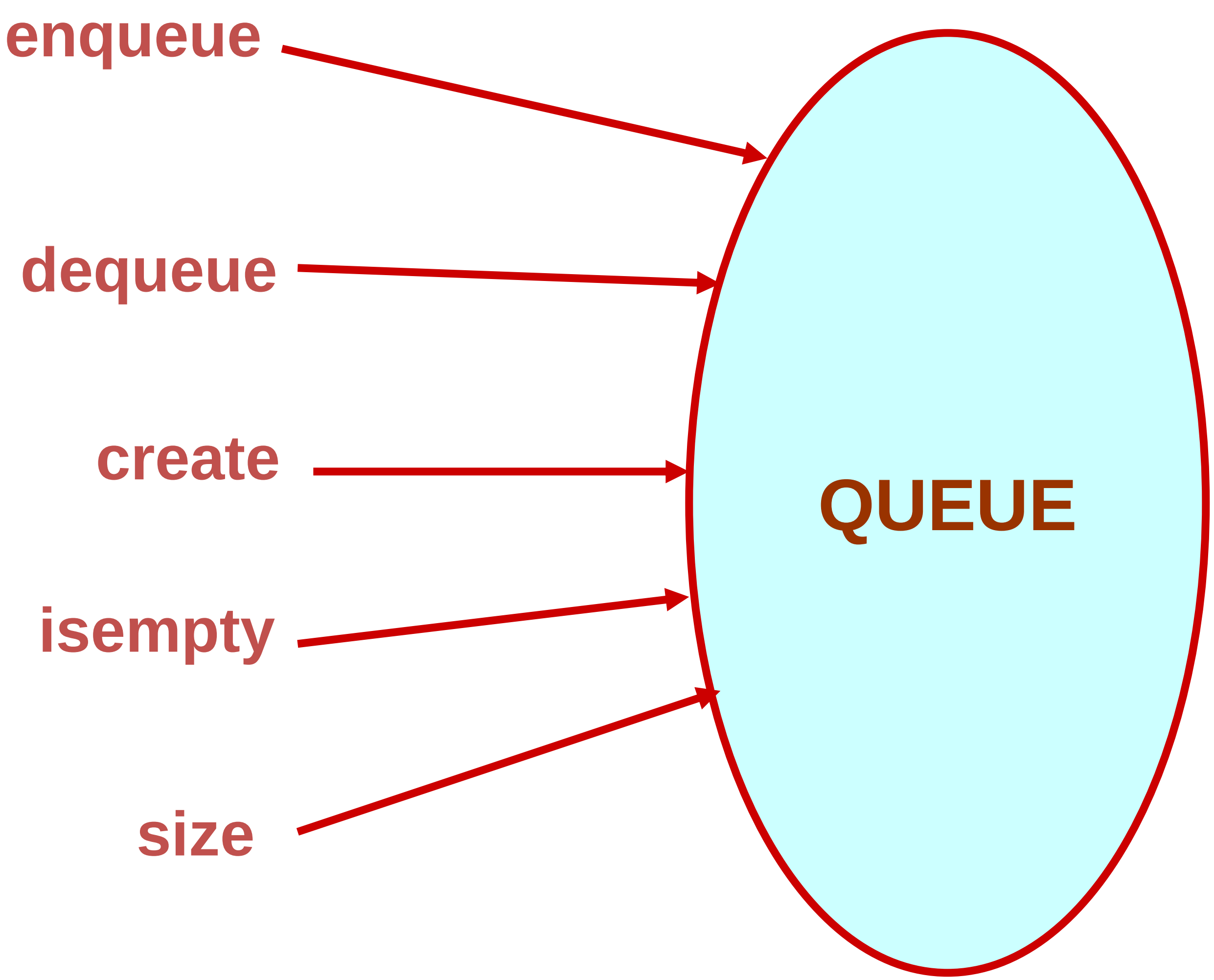
Contd.

- We shall look into two different ways of implementing stack:
  - Using arrays
  - Using linked list

Example 4 :: First-In-First-Out QUEUE

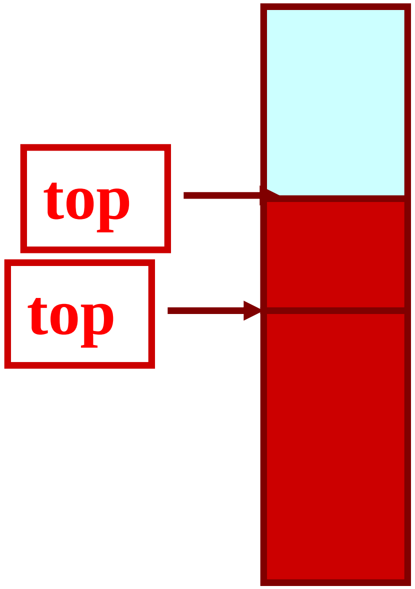
Assume:: queue contains integer elements

```
void enqueue (queue *q, int element);
                /* Insert an element in the queue */
int dequeue (queue *q);
                /* Remove an element from the queue */
queue *create();
                /* Create a new queue */
int isempty (queue *q);
                /* Check if queue is empty */
int size (queue *q);
                /* Return the no. of elements in queue */
```


enqueue
dequeue
create
isempty
size
QUEUE

Stack Implementations: Using Array and Linked List

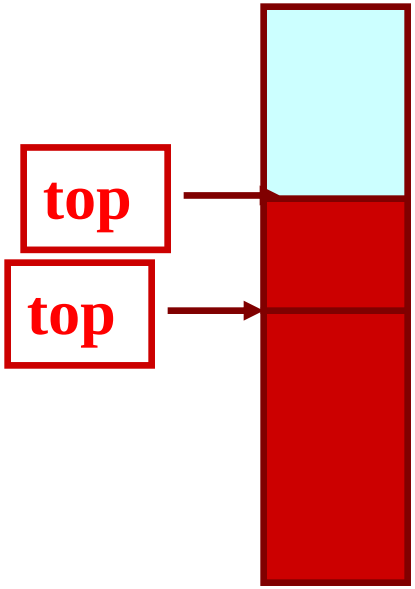
STACK USING ARRAY



PUSH

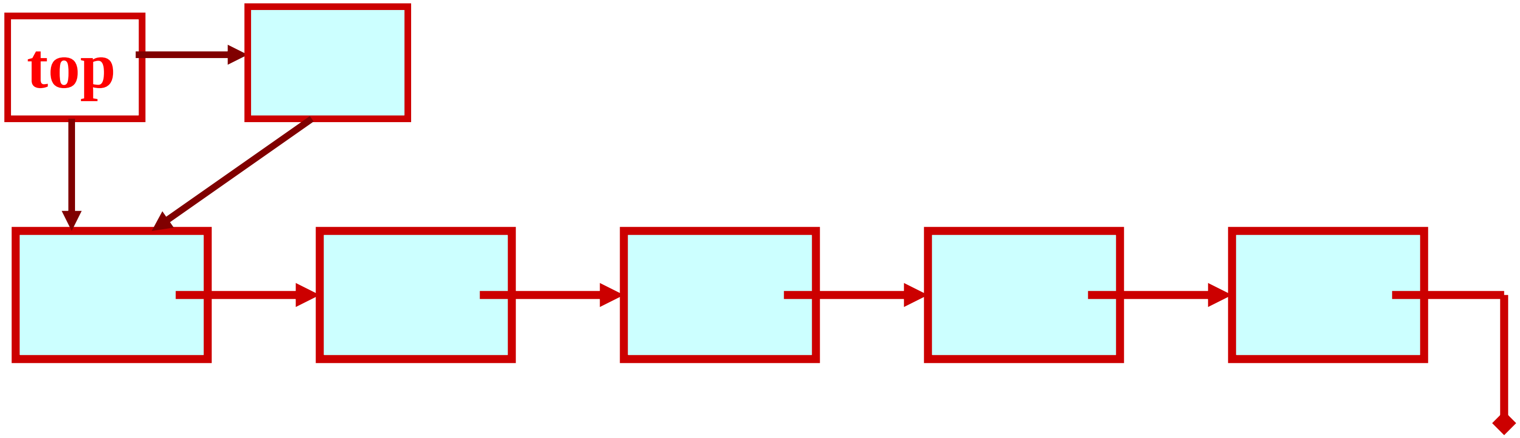
STACK USING ARRAY

POP



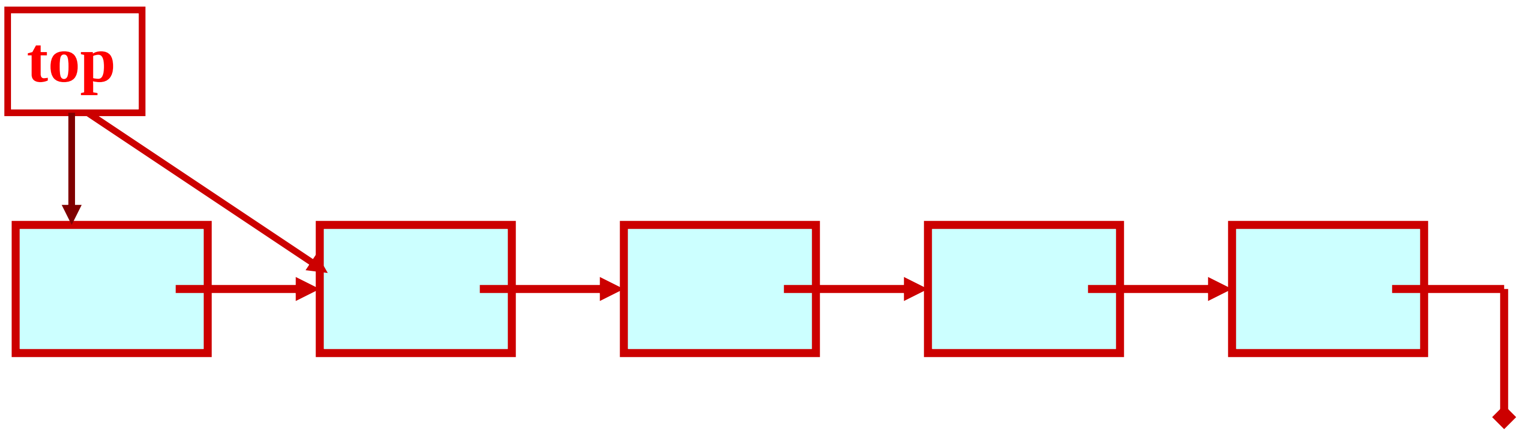
Stack: Linked List Structure

PUSH OPERATION



Stack: Linked List Structure

POP OPERATION



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;
```

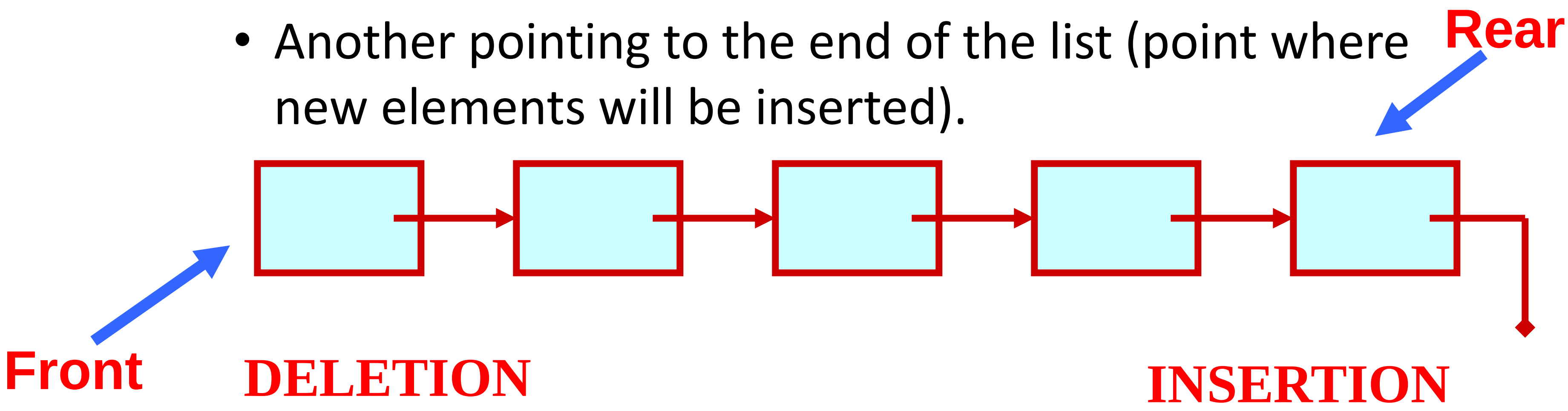
```
main()
{
  stack *A, *B;
  create(&A); create(&B);
  push(&A,10);
  push(&A,20);
```

```
  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

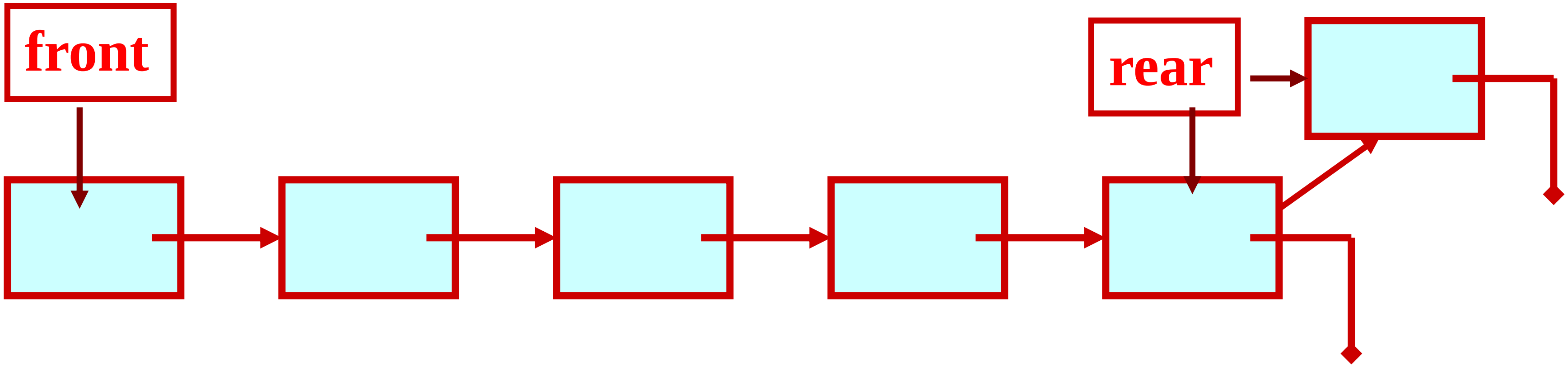
Basic Idea

- 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 new elements will be inserted).



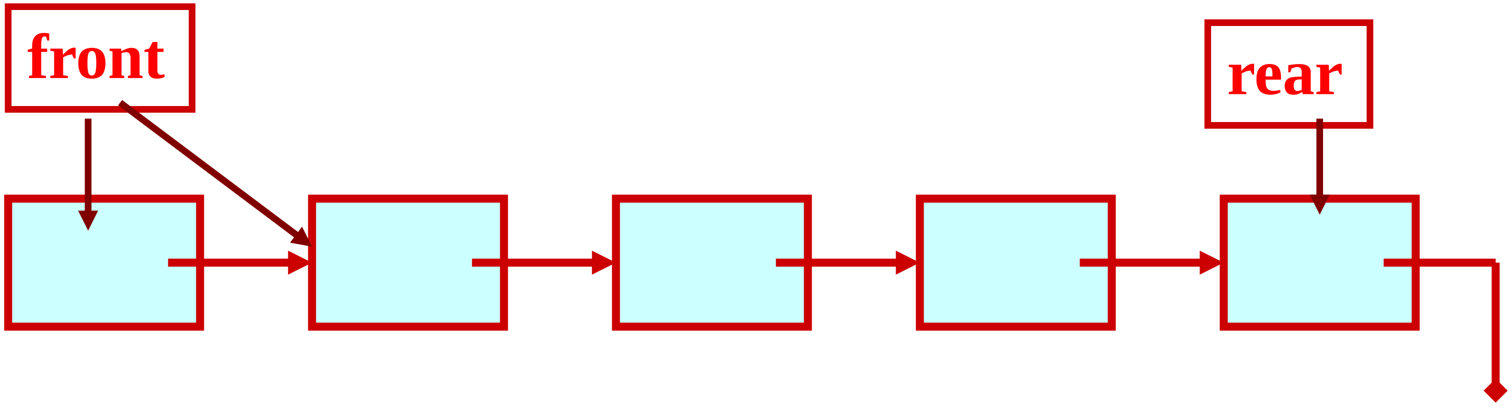
QUEUE: LINKED LIST STRUCTURE

ENQUEUE



QUEUE: LINKED LIST STRUCTURE

DEQUEUE



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);
```

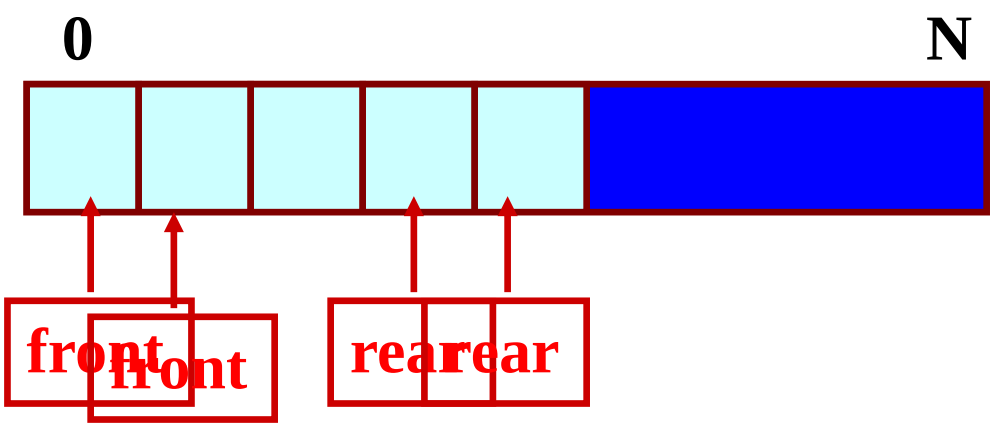
```
if(!strcmp(command,"enter")) {
scanf("%s",val);
if((enqueue(&q,val)==NULL))
printf("No more pushing please \n");
else printf("Name entered %s \n",val);
}
```

```
if(!strcmp(command,"delete")) {
if(!isEmpty(&q))
printf("%s \n",dequeue(&q,val));
else printf("Name deleted %s \n",val);
}
} /* while */
printf("End session \n");
}
```

Problem With Array Implementation

ENQUEUE **DEQUEUE**

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");
}
```