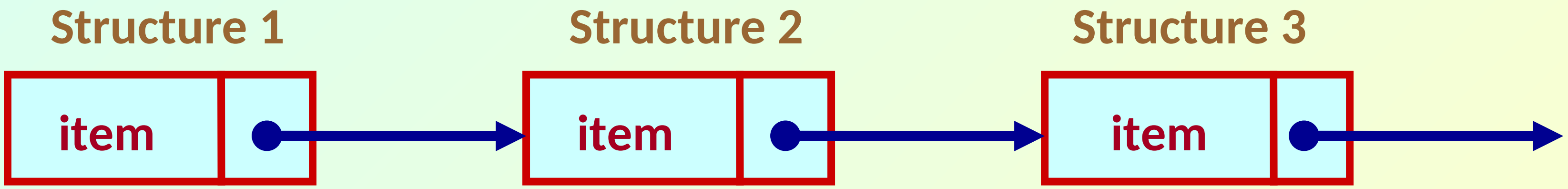
Self-referential Structures
and
Linked List

Linked List :: Basic Concepts

- A list refers to a set of items organized sequentially.
  - An array is an example of a list.
    - The array index is used for accessing and manipulating array elements.
  - Problems with array:
    - The array size has to be specified at the beginning.
    - Deleting an element or inserting an element may require shifting of elements in the array.

Contd.

- A completely different way to represent a list:
  - Make each item in the list part of a structure.
  - The structure also contains a pointer or link to the structure containing the next item.
  - This type of list is called a *linked list*.



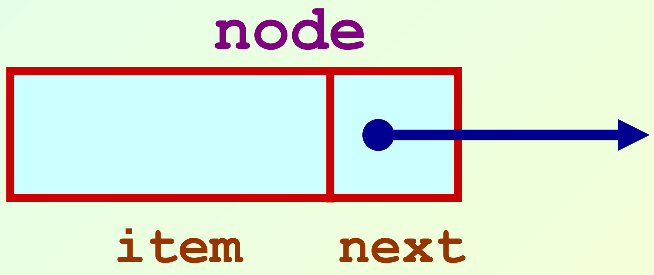
Contd.

- Each structure of the list is called a *node*, and consists of two fields:
  - One containing the data item(s).
  - The other containing the address of the next item in the list (that is, a *pointer*).
- The data items comprising a linked list need not be contiguous in memory.
  - They are ordered by logical links that are stored as part of the data in the structure itself.
  - The link is a pointer to another structure of the same type.

Contd.

- Such a structure can be represented as:

```
struct node
 {
    int item;
    struct node  *next;
 }
```



- Such structures that contain a member field pointing to the same structure type are called ***self-referential structures***.

Contd.

- In general, a node may be represented as follows:

```
struct node_name
      {
         type  member1;
         type  member2;
          ………
         struct node_name *next;
      }
```

Illustration

- Consider the structure:

```
struct stud
    {
        int  roll;
        char name[30];
        int  age;
        struct stud *next;
    }
```

- Also assume that the list consists of three nodes **n1**, **n2** and **n3**.

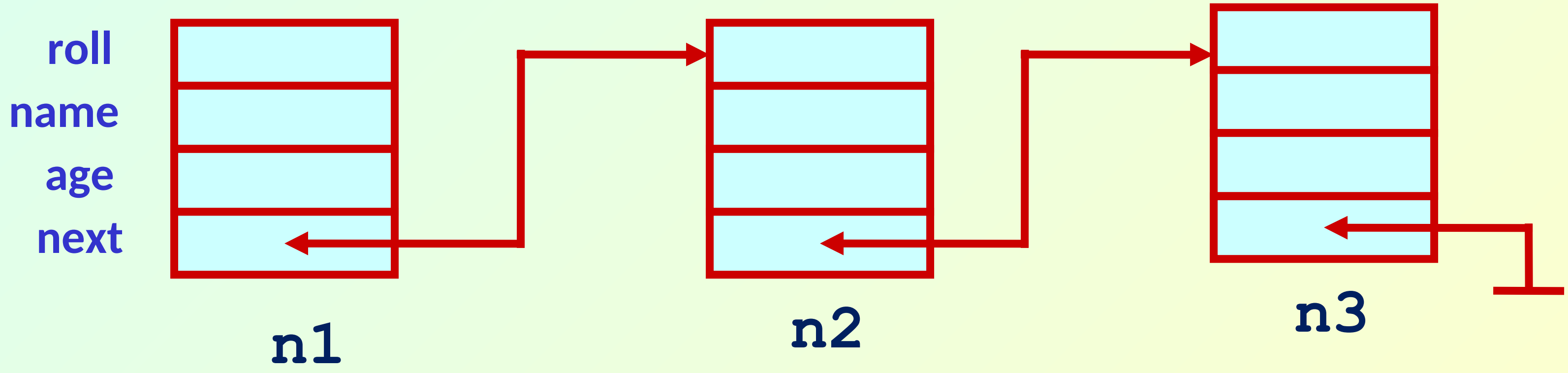
```
struct stud n1, n2, n3;
```

Contd.

- To create the links between nodes, we can write:

```
n1.next = &n2;
n2.next = &n3;
n3.next = NULL;    /* No more nodes follow */
```

- Now the list looks like:



- **Some important observations:**
  - The `NULL` pointer is used to indicate that no more nodes follow, that is, it is the end of the list.
  - To use a linked list, we only need a *pointer to the first element* of the list.
  - Following the chain of pointers, the successive elements of the list can be accessed by *traversing* the list.

Example: without using function

```
#include <stdio.h>
struct  stud
  {
      int    roll;
      char   name[30];
      int    age;
      struct  stud  *next;
  }

main()
{
    struct  stud  n1, n2, n3;
    struct  stud  *p;

    scanf ("%d %s %d", &n1.roll, n1.name, &n1.age);
    scanf ("%d %s %d", &n2.roll, n2.name, &n2.age);
    scanf ("%d %s %d", &n3.roll, n3.name, &n3.age);
```

```
    n1.next  =  &n2;
    n2.next  =  &n3;
    n3.next  =  NULL;

   /* Now traverse the list and print the elements */

    p = &n1;   /* point to 1st element */
    while (p != NULL)
    {
      printf ("\n %d %s %d", p->roll, p->name, p->age);
      p = p->next;
    }
}
```

A function to carry out traversal

```
#include<stdio.h>
struct  stud
  {
      int   roll;
      char  name[30];
      int   age;
      struct  stud  *next;
  }


void  traverse (struct stud *head)
{
   while (head != NULL)
   {
     printf (”\n %d %s %d”, head->roll, head->name,
                                head->age);
     head = head->next;
   }
}
```

The corresponding main() function

```
main()
{
    struct  stud  n1, n2, n3, *p;

    scanf ("%d %s %d", &n1.roll, n1.name, &n1.age);
    scanf ("%d %s %d", &n2.roll, n2.name, &n2.age);
    scanf ("%d %s %d", &n3.roll, n3.name, &n3.age);

    n1.next  =  &n2;
    n2.next  =  &n3;
    n3.next  =  NULL;

    p = &n1;
    traverse (p);
}
```

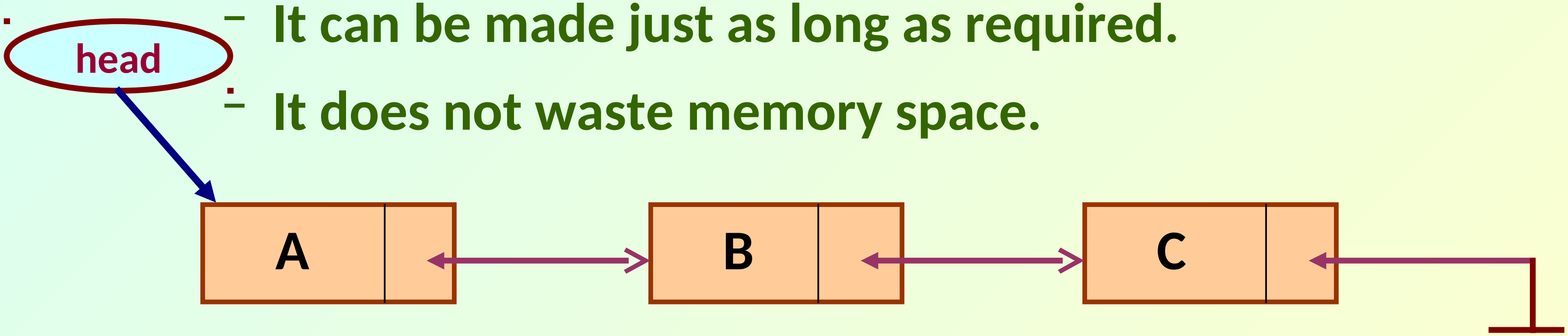
Alternative and More General Way

- Dynamically allocate space for the nodes.
  - Use `malloc()` or `calloc()` for allocating space for every individual nodes.
  - No need for allocating additional space unnecessarily like in an array.

Linked List in more detail

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

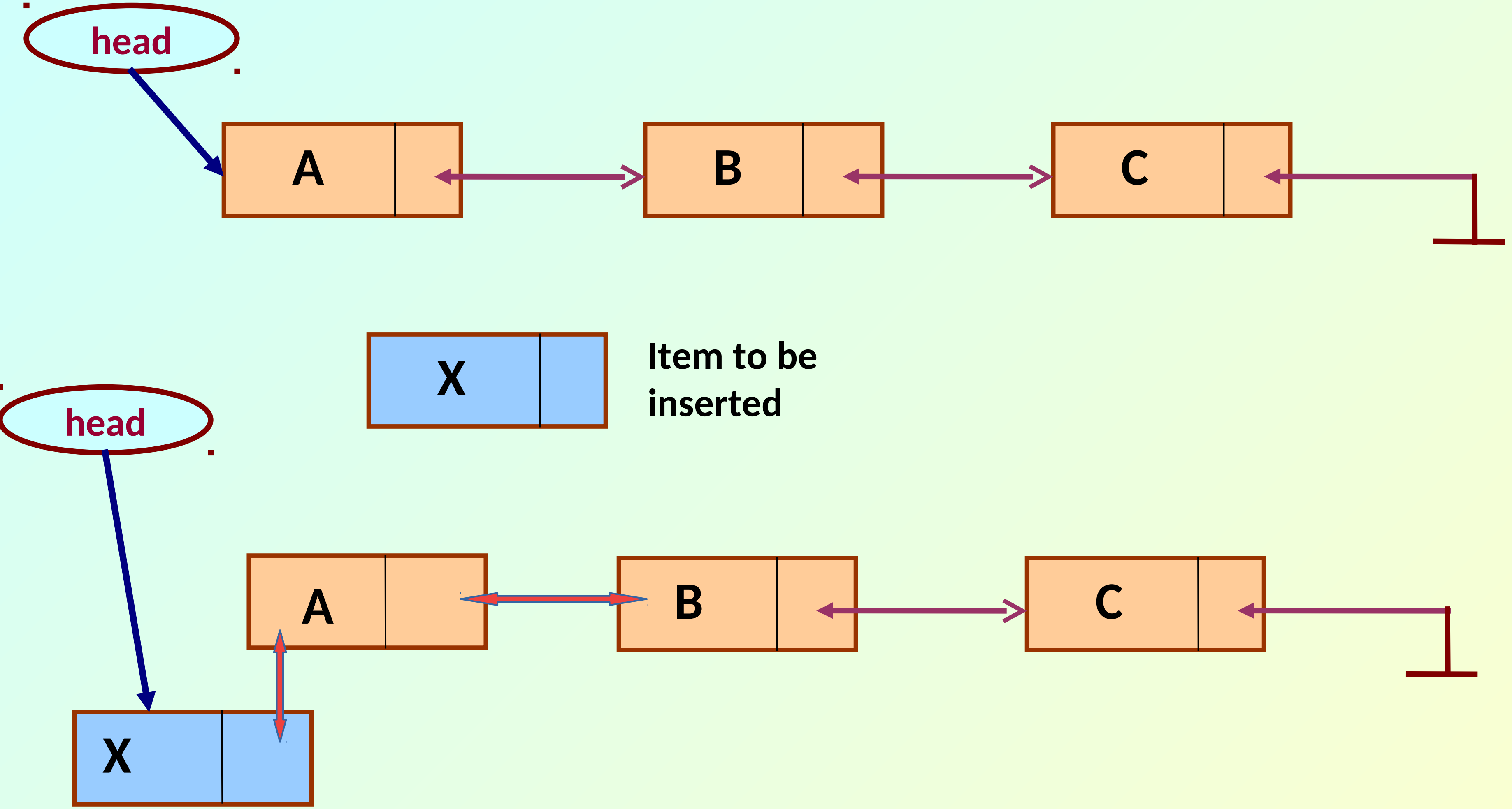
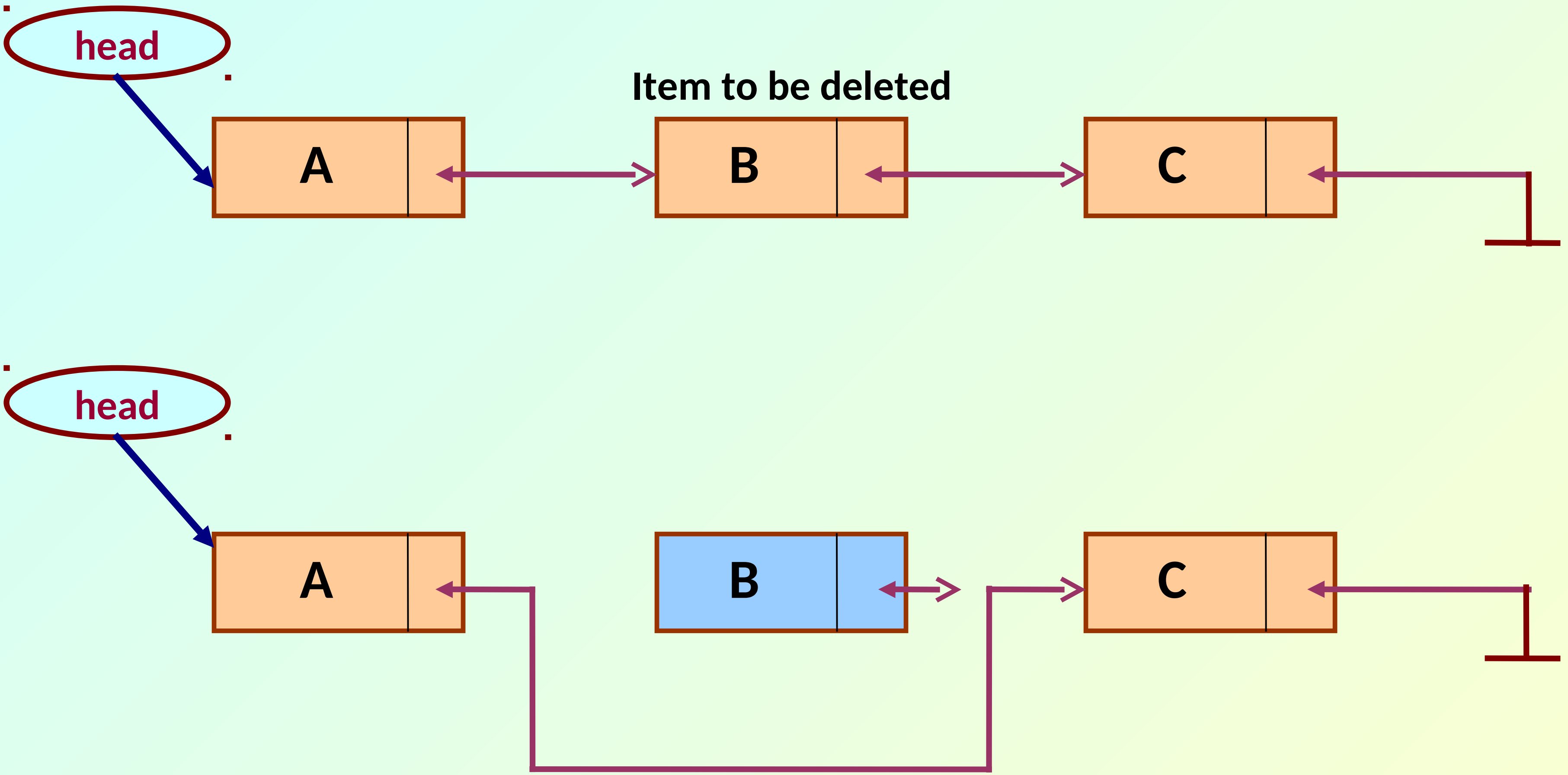


Illustration: Deletion



In essence …

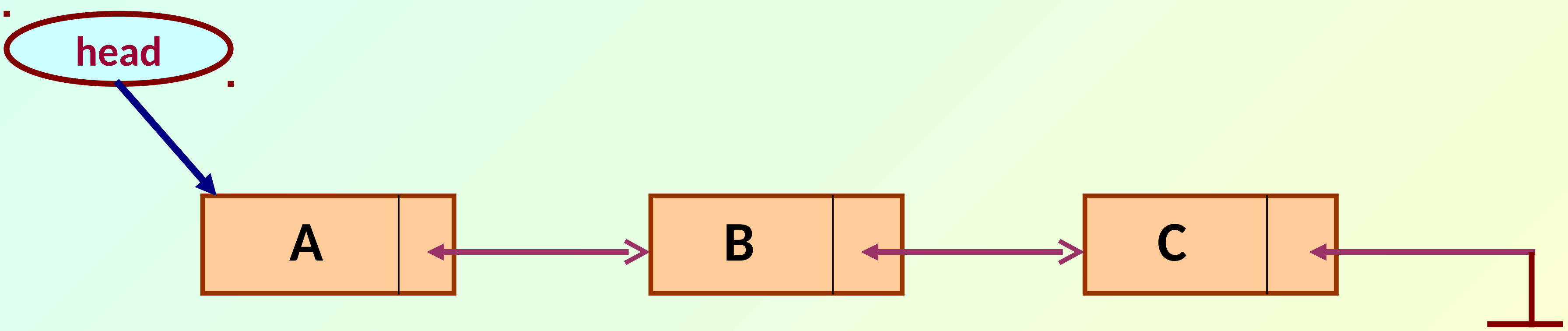
- **For insertion:**
  - A record is created holding the new item.
  - The *next* pointer of the new record is set to link it to the item which is to follow it in the list.
  - The *next* pointer of the item which is to precede it must be modified to point to the new item.
- **For deletion:**
  - The *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.

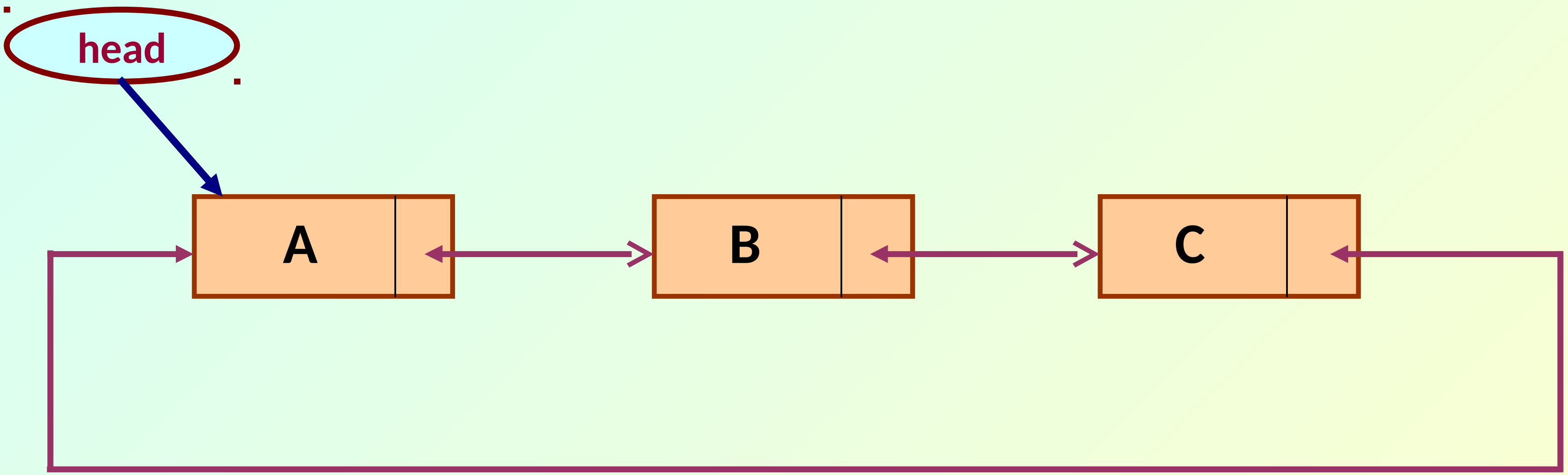
Types of Lists

- 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.



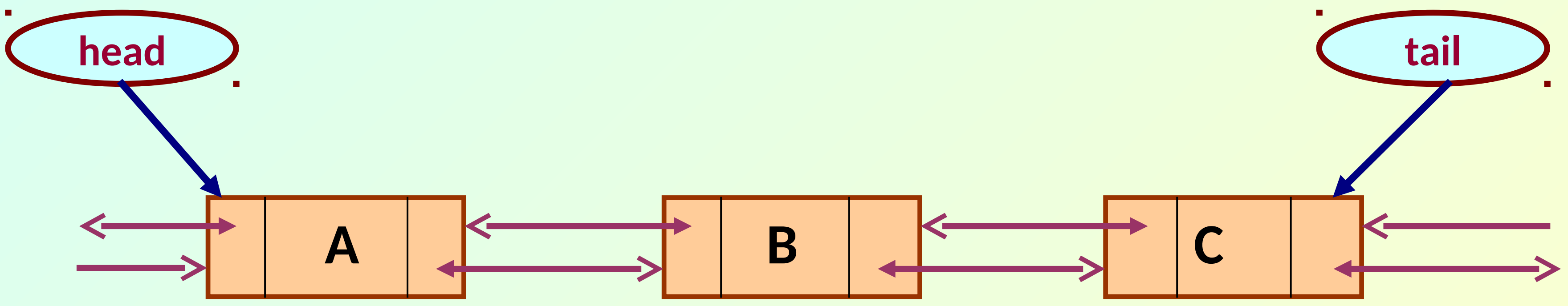
– **Circular linked list**

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



– **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.**



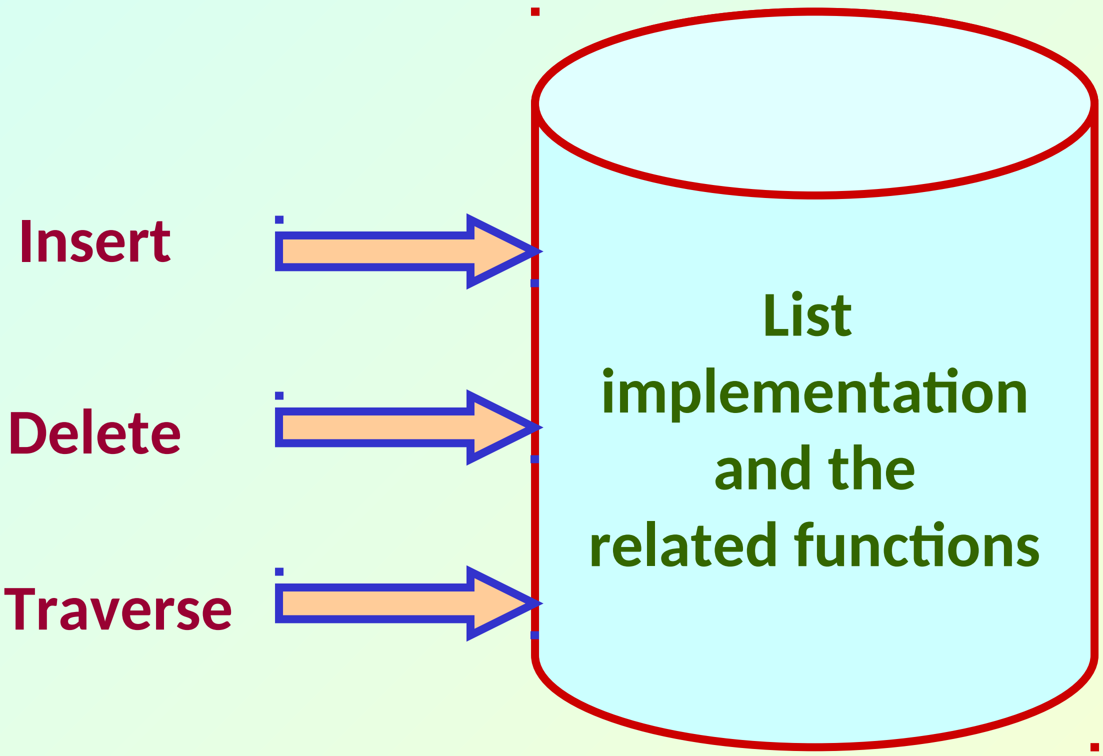
Basic Operations on a List

- 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.

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;
            };
```

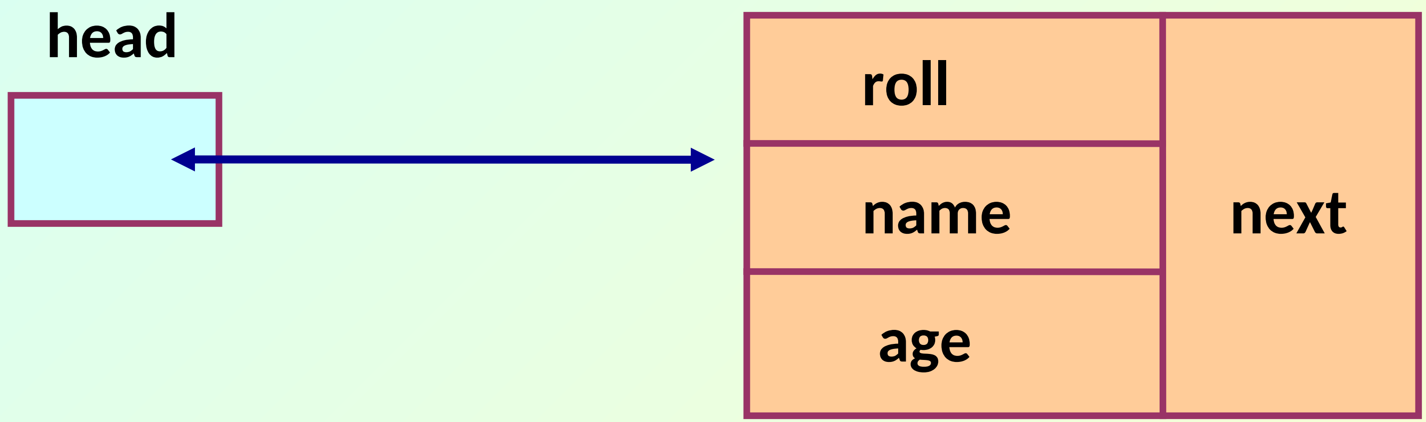
```
  /* A user-defined data type called “node” */

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

Creating a List

How to begin?

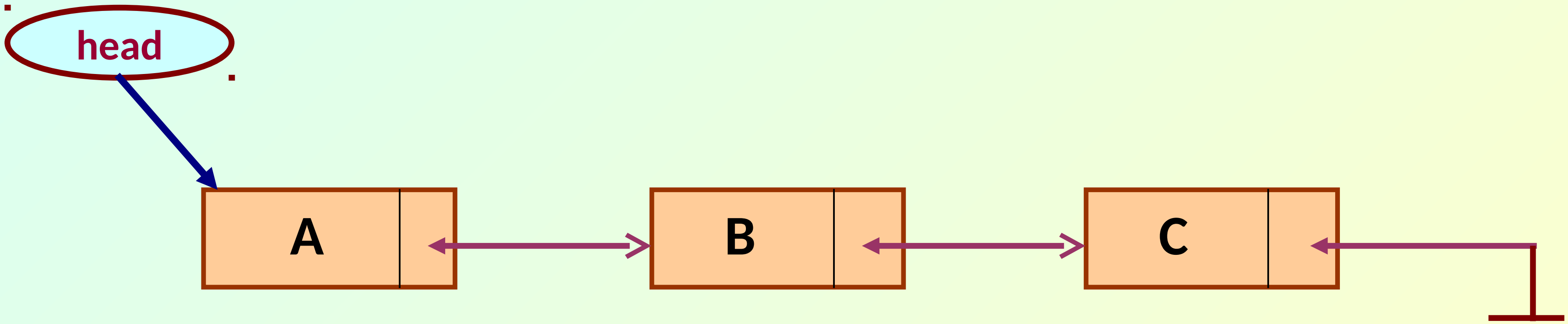
- To start with, we have to create a node (the first node), and make ***head*** point to it.

```
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.**



```c
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.

a) 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.

b) 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`.

c) 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.

Why is the argument a pointer to pointer?

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

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

    printf ("\nEnter data 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.

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

A sample main() function

```
int main()
{
    node *head;

    head = create_list();
    display(head);

    insert(&head);
    display(head);

    delete(&head);
    display(head);
}
```

Few Exercises to Try Out

- Write functions to:
  1. Concatenate two given lists into one big list.
     - node *concatenate (node *head1, node *head2);
  2. 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);
  3. 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 */

More Exercises

4. **Implement a circular linked list, and write functions to insert, delete, and traverse nodes in the list.**
5. **Represent a polynomial as a linked list, where every node will represent a term of the polynomial ($a_n x^n$), and will contain the values of ‘n’ and ‘a_n’. Write a function to add two given polynomials.**

Abstract Data Types

Definition

- An abstract data type (ADT) is a specification of a set of data and the set of operations that can be performed on the data.
- Such data type is abstract in the sense that it is independent of various concrete implementations.
- Some examples follow.

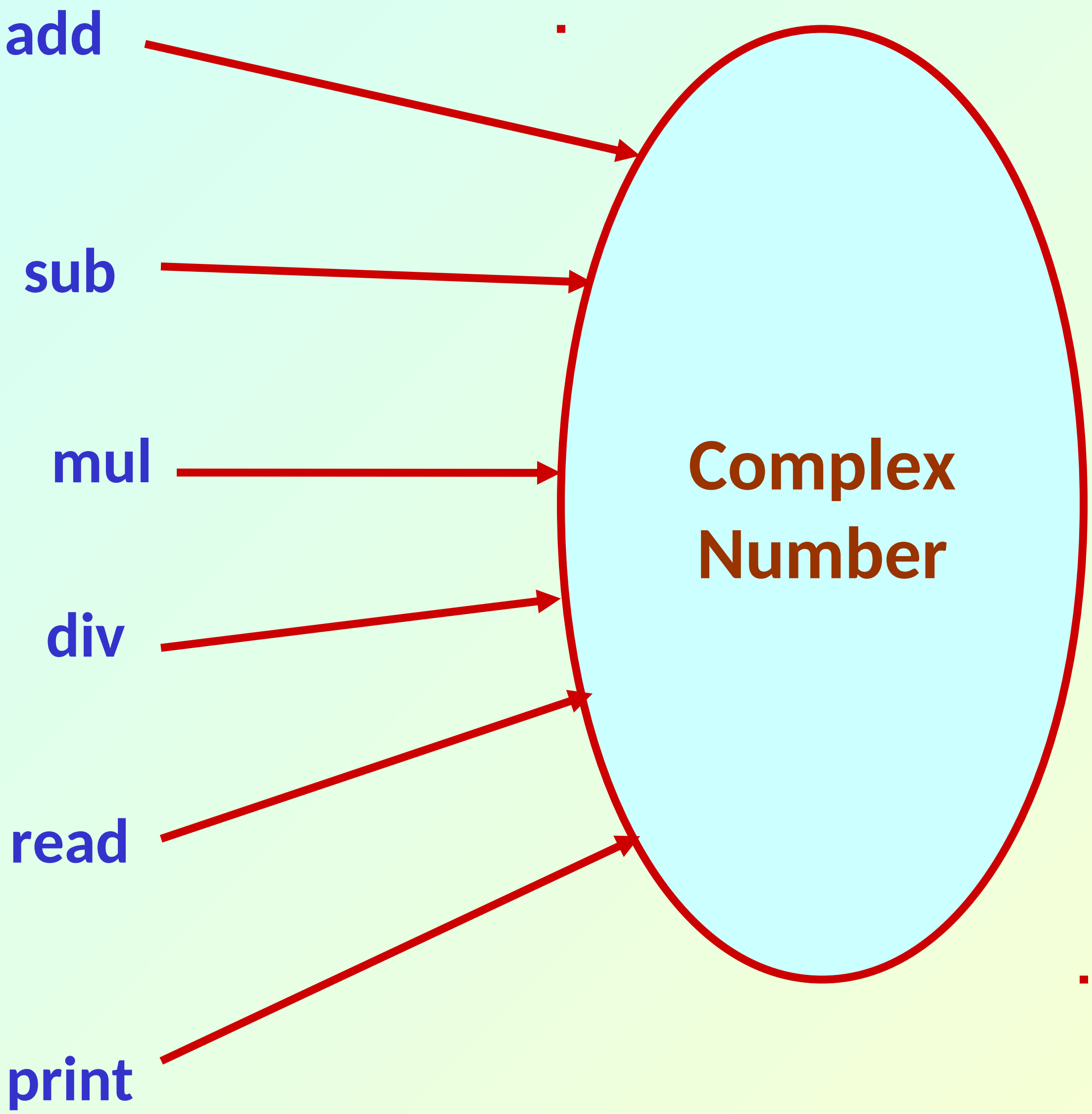
Example 1 :: Complex numbers

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

Structure definition

```
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);
```

Function prototypes


add
sub
mul
div
read
print
Complex Number

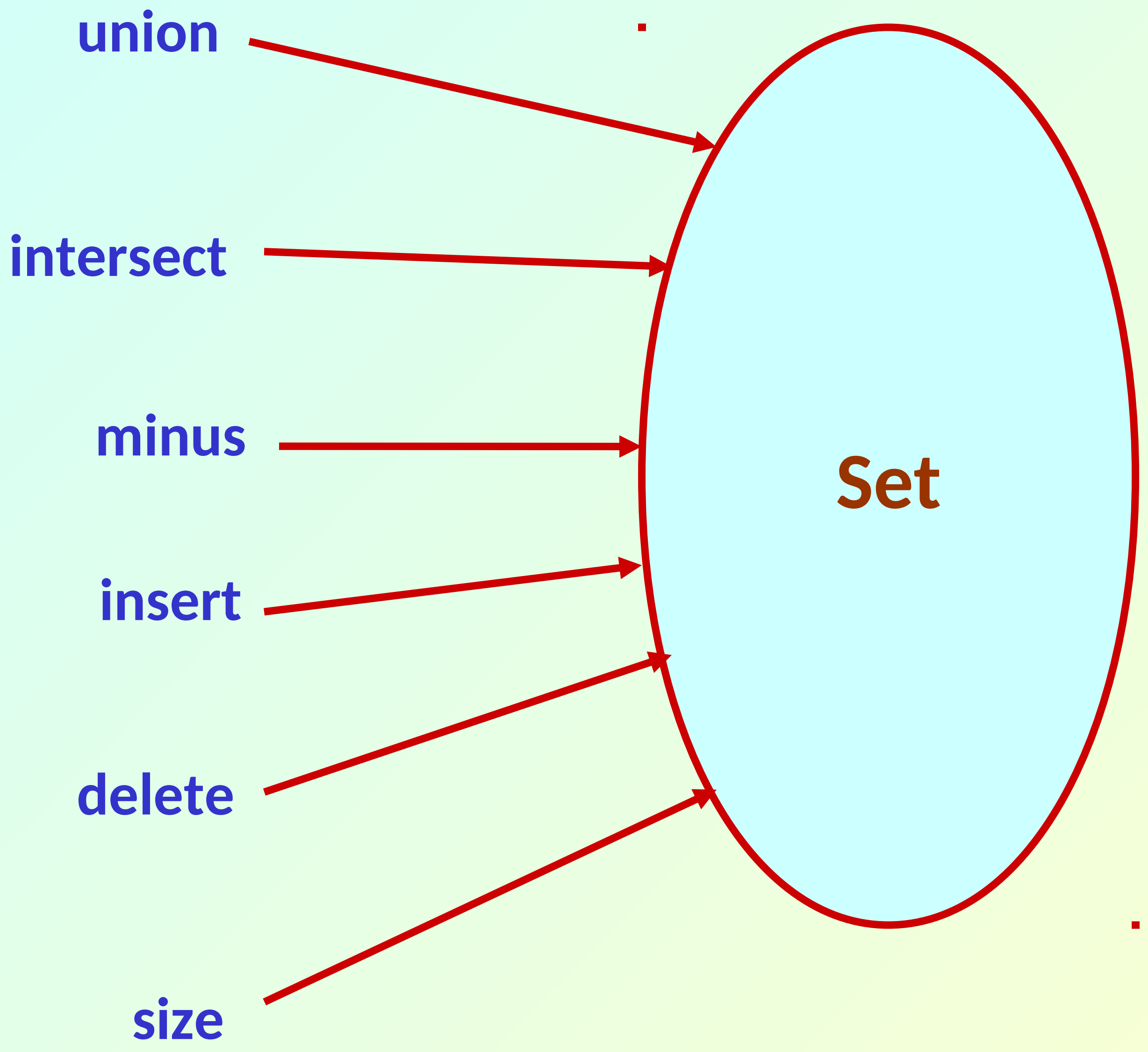
Example 2 :: Set manipulation

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

Structure definition

```
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);
```

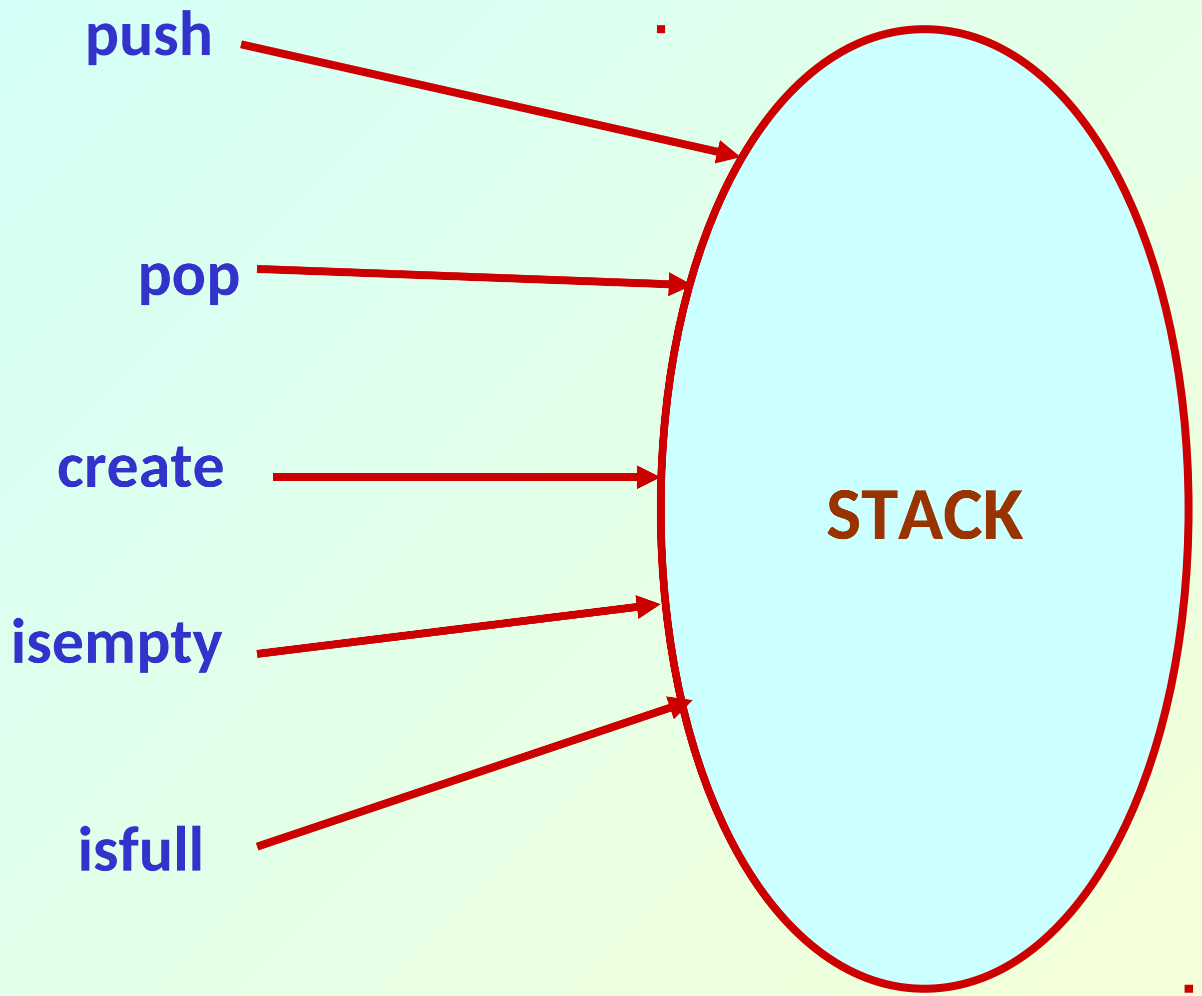
Function prototypes


union
intersect
minus
insert
delete
size
Set

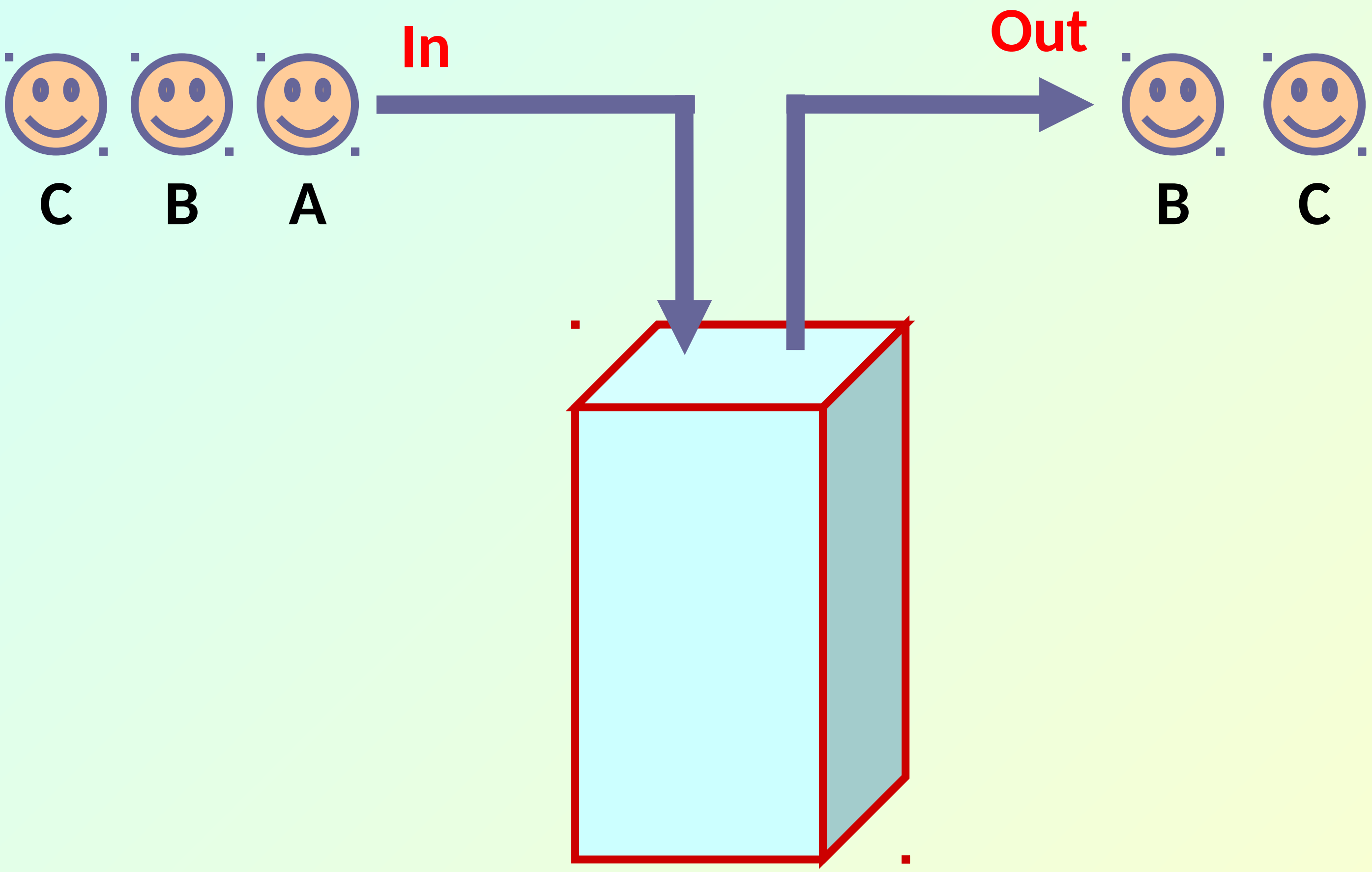
Example 3 :: Last-In-First-Out STACK

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

Visualization of a Stack



Contd.

- We shall later 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 *createq();
```

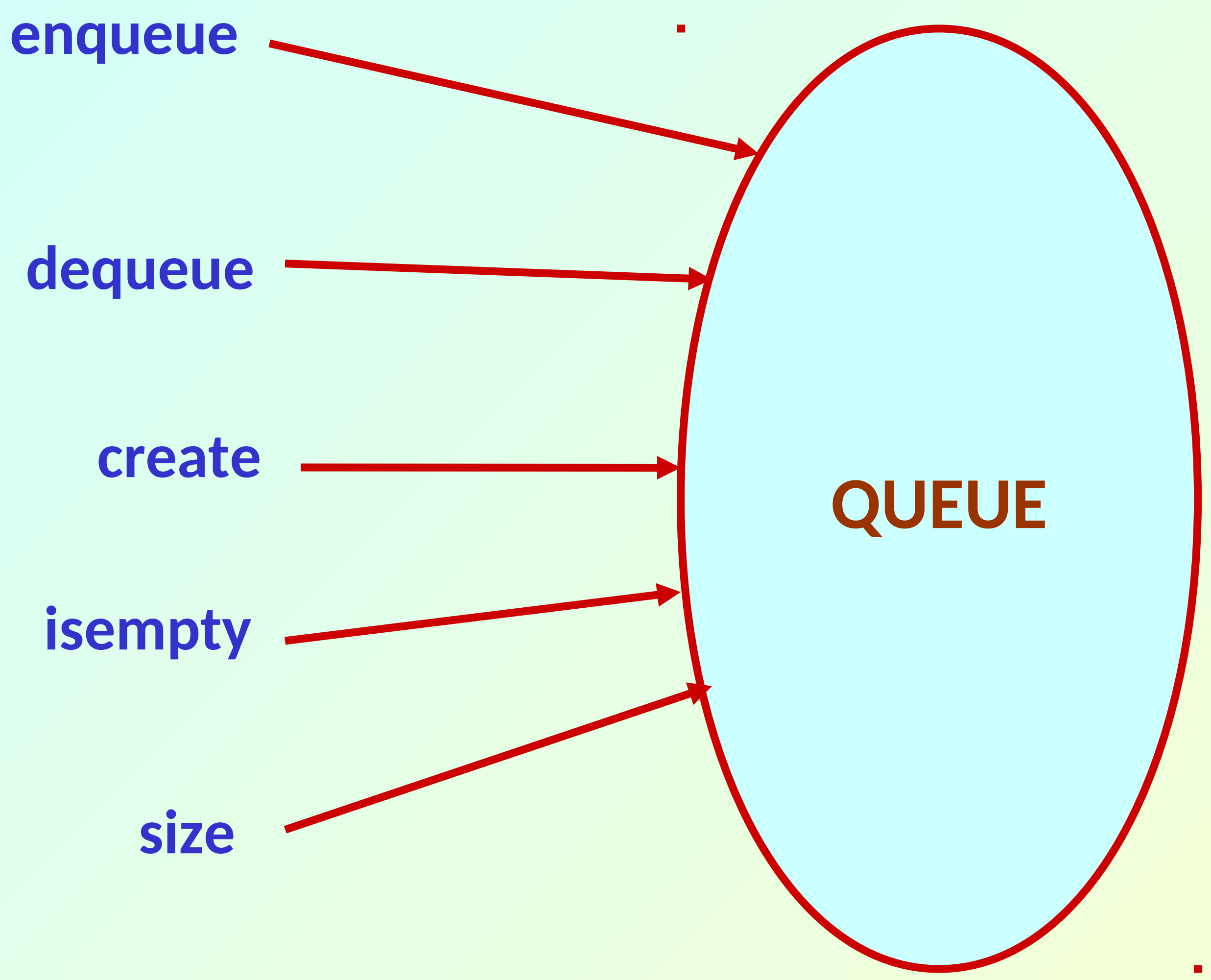
/* Create a new queue */

```
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

Visualization of a Queue