Abstract Data Types and Linked Lists

Abstract Data Types

Definition

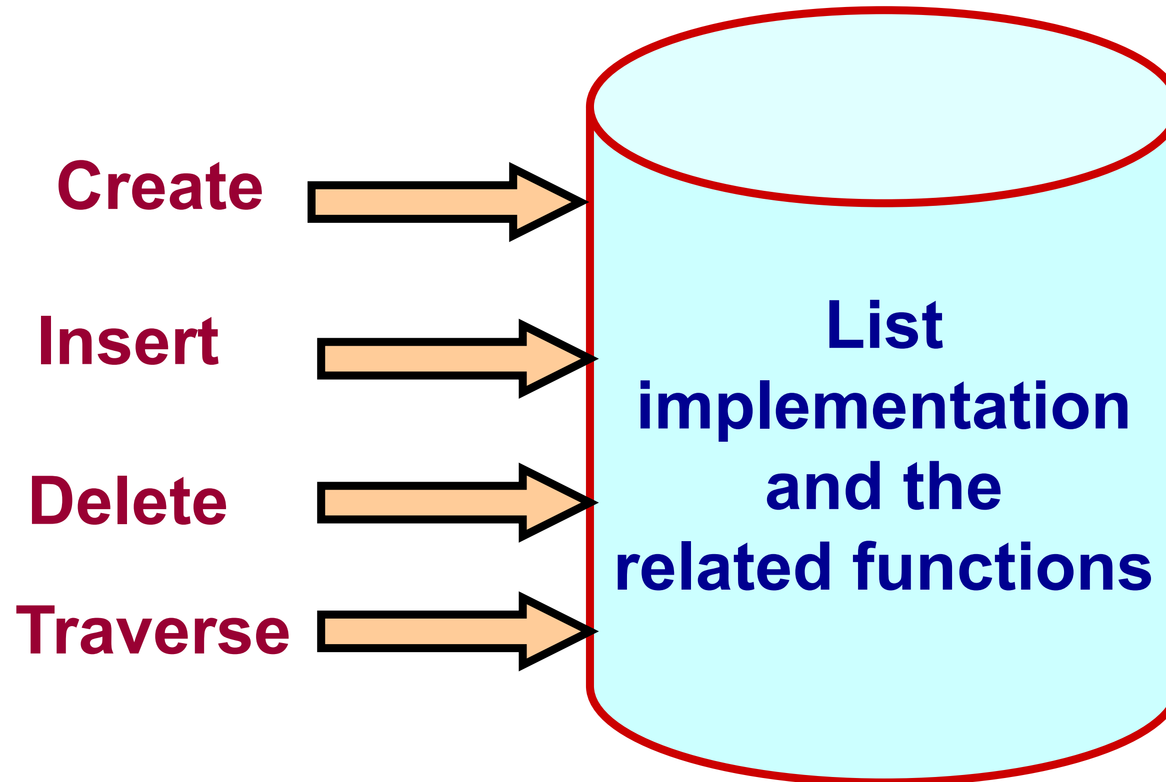
An abstract data type (ADT) is a specification of a set of data and the set of operations that can be performed on the set of data.

Such a data type is abstract in the sense that it is independent of various concrete implementations.

- To perform some operation on the ADT, the user just calls a function.
- Details of how the ADT is implemented or how the functions are written is not required to be known by the user.
- Even if the underlying implementations of the functions are changed, as long as the function interfaces remain the same, the user can continue using the ADT in the same way.

Some examples follow.

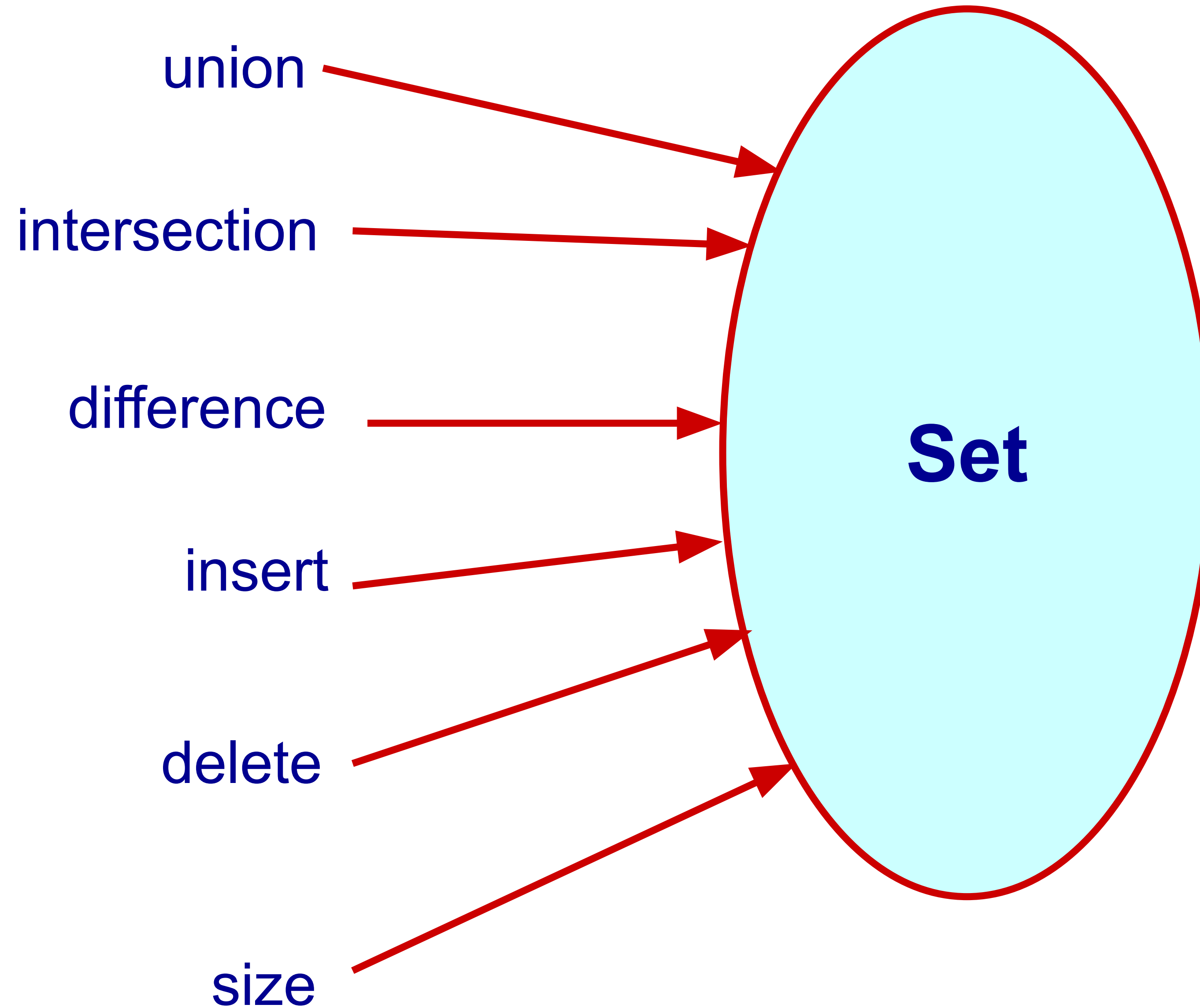
Example 1: List (a sequence of data items of the same type)



We shall later look into a concrete way of implementing a list

A list is ordered: There are a first element, a second element, a third element, and so on.

Example 2 :: Set



Note: A set is an unordered collection.

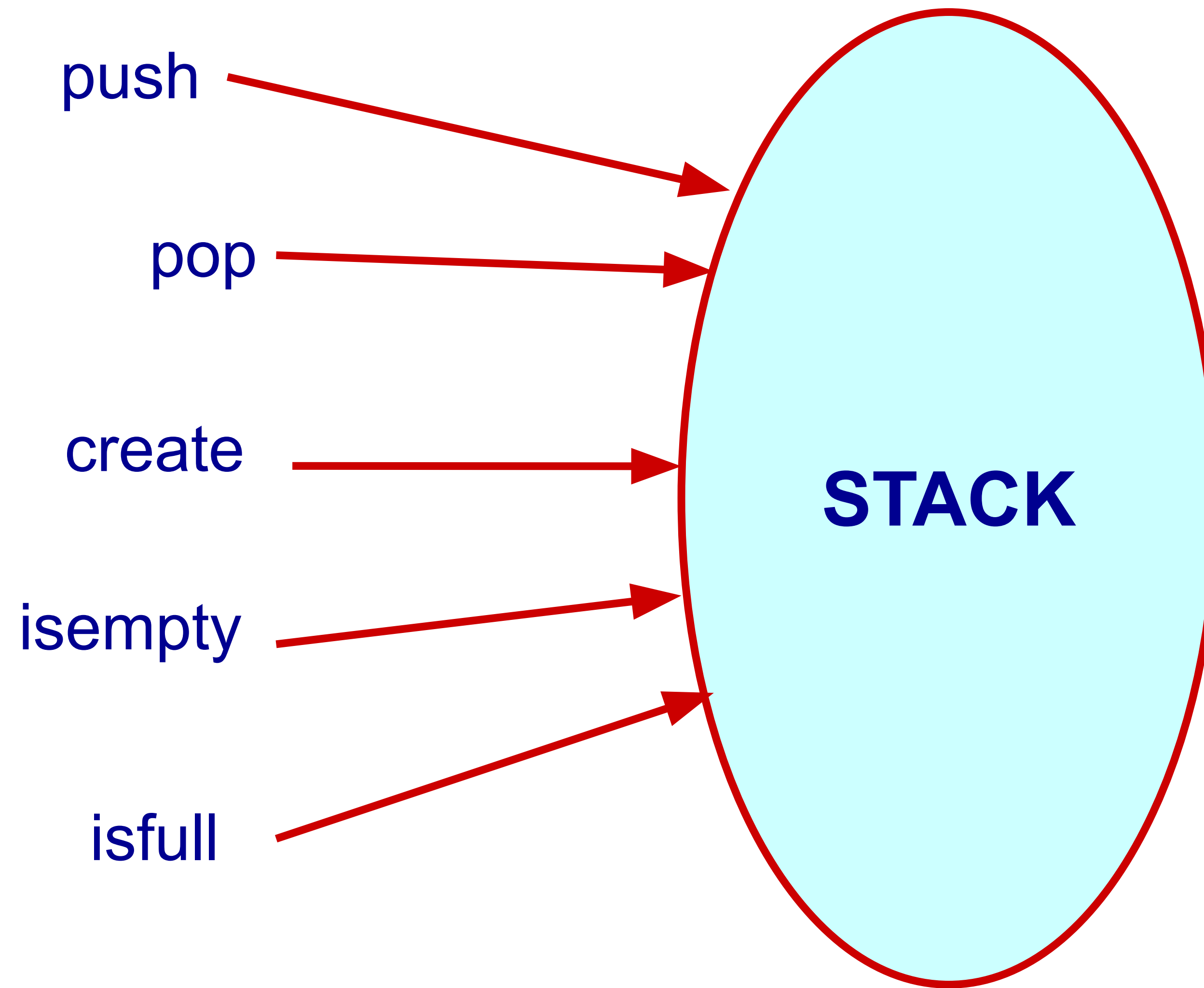
{1,2,3}, {1,3,2}, {2,1,3}, {2,3,1}, {3,1,2}, {3,2,1} are all the same set.

Example 2 :: Set

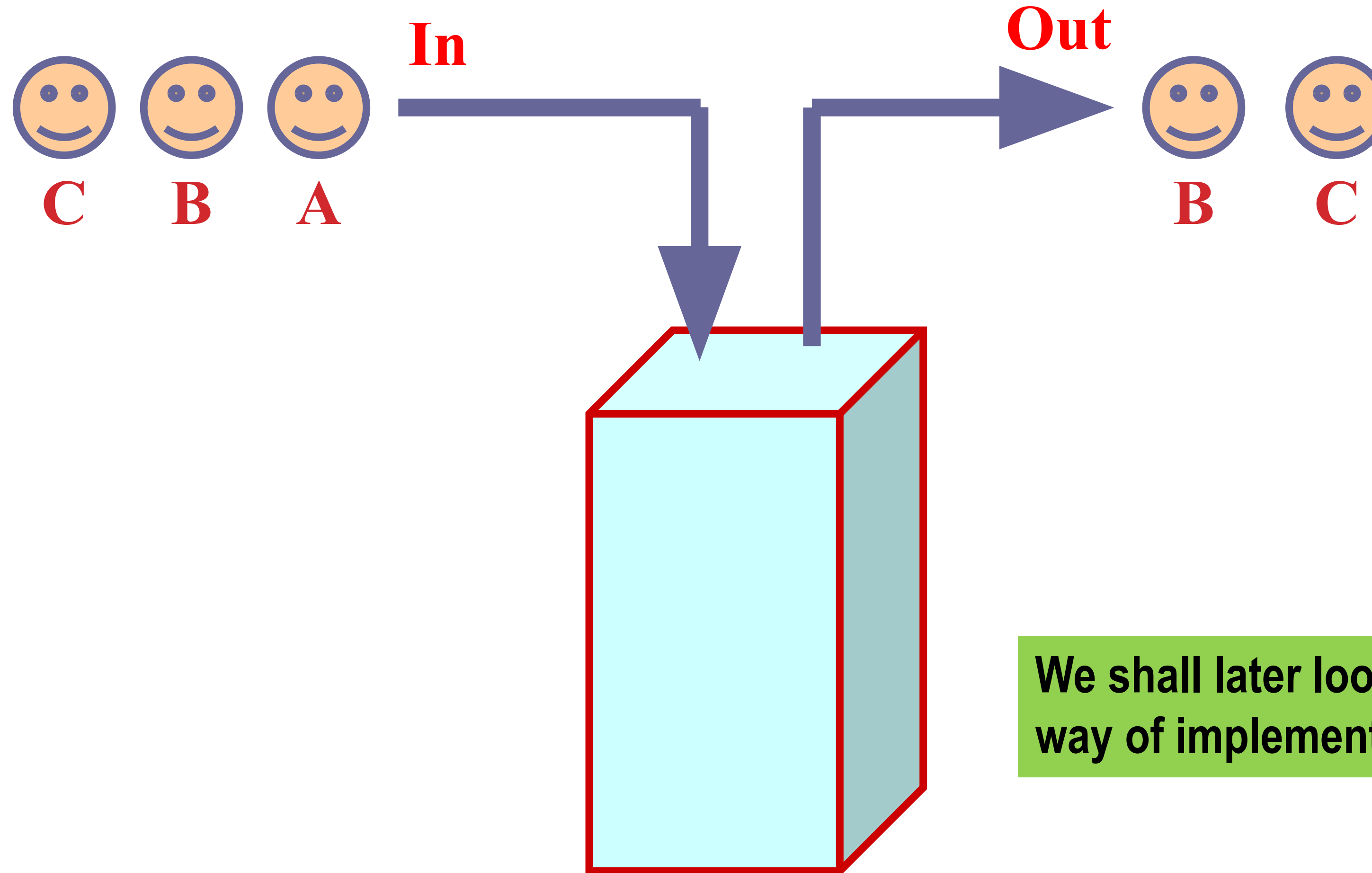
```
void insert (set a, int x);
void delete (set a, int x);
int size (set a);
set union (set a, set b);
set intersection (set a, set b);
set difference (set a, set b);
```

Function prototypes

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



Visualization of a Stack



We shall later look into a concrete way of implementing a stack

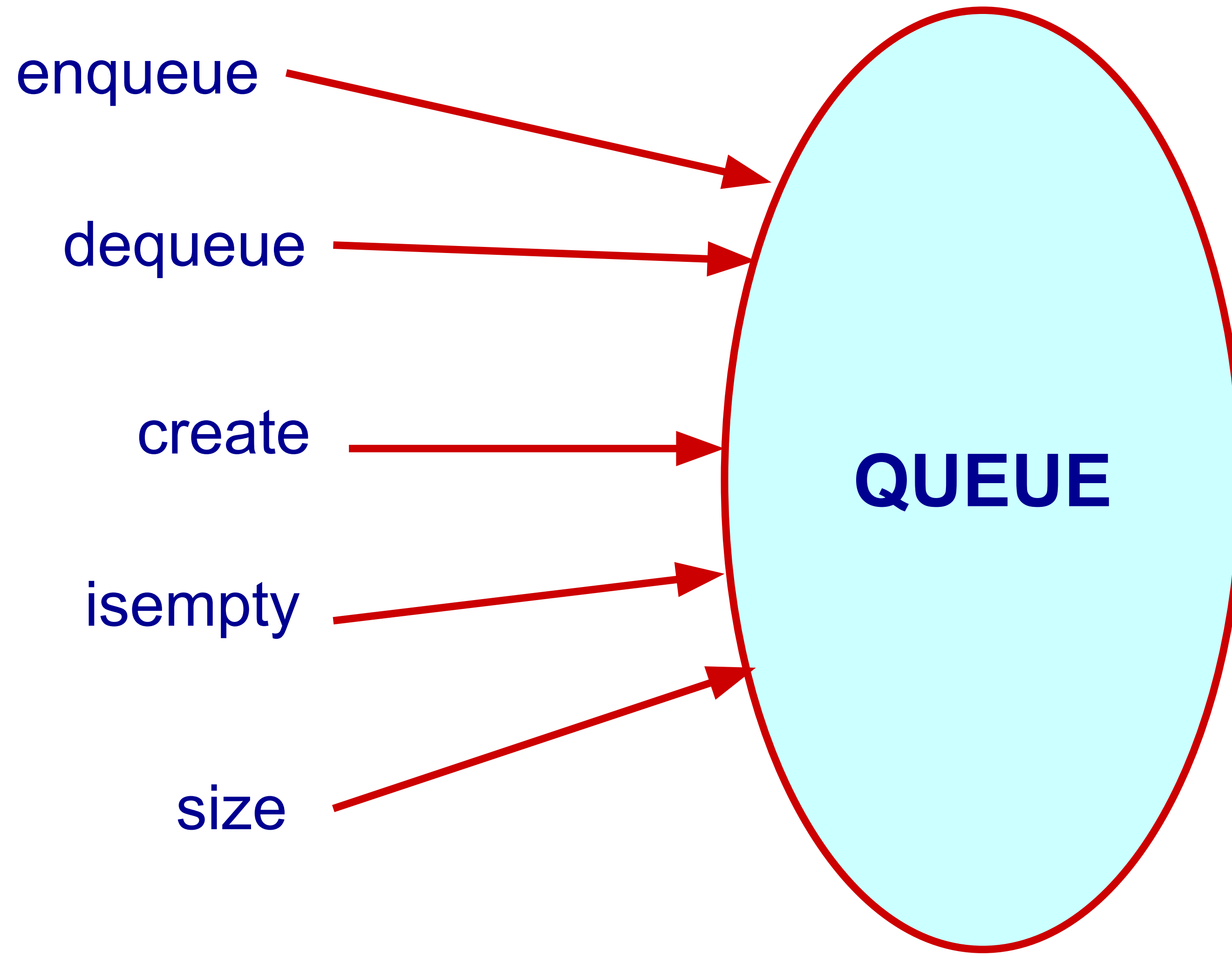
Example 3 :: Last-In-First-Out STACK

Assume:: stack contains integer elements

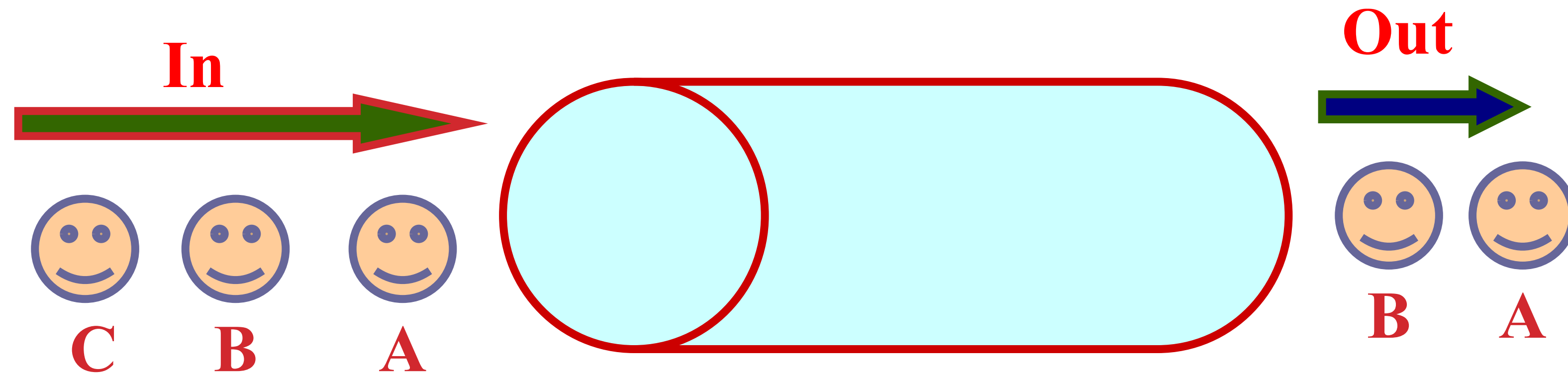
```
void push (stack s, int element);
                    /* Insert an element in the stack */
int pop (stack s);
                    /* Remove and return the top element */
void create (stack  s);
                    /* Create a new stack */
int isempty (stack s);
                    /* Check if stack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

We shall later look into a concrete way of implementing a stack

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



Visualization of a Queue



We shall later look into a concrete way of implementing a queue

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

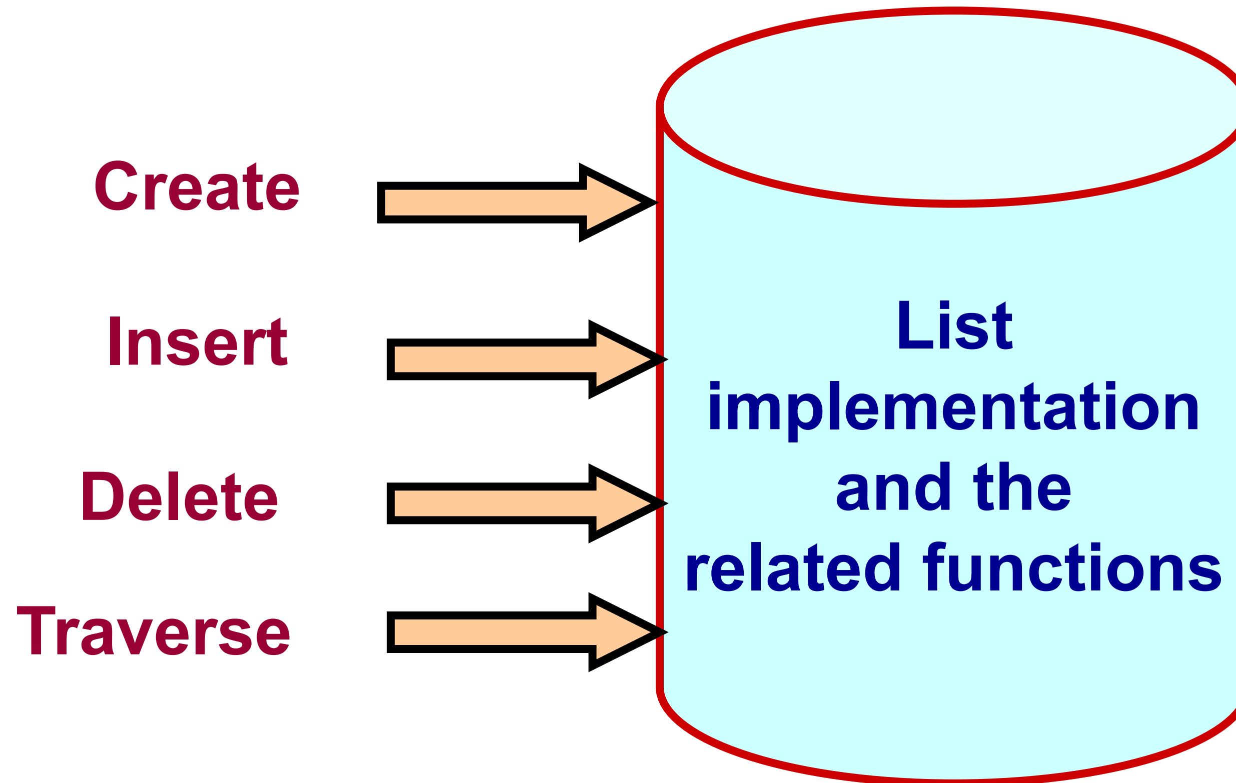
Assume:: queue contains integer elements

```
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 */
```

We shall later look into a concrete way of implementing a queue

Lists

List (an ordered sequence of data items of the same type)



Lists

- List is a sequence of data items (usually of the same type).
- Array – one way to represent a list.
- Advantages of arrays:
  - Compact: no wastage of space
  - Easy and constant time access given index of an element

- Problems with arrays
  - Size of an array should be specified beforehand (at least while dynamically allocating memory).
  - Deleting/Inserting an element requires shifting of elements.

Can we have some other implementation of Lists, to address these problems?

Self-Referential Structures

A structure referencing itself – how?

We use a pointer inside a structure that points to a structure of the same type.

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

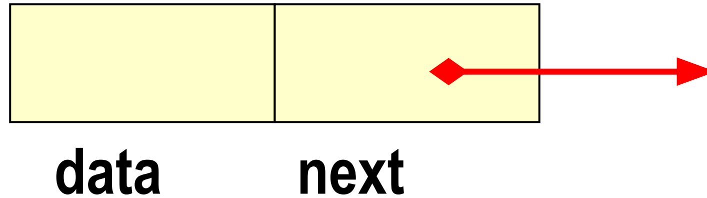
Self-Referential Structures

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

The pointer variable next is called a link.
Each structure is linked to a succeeding structure by the next pointer.

Pictorial representation

A structure of type **struct list**

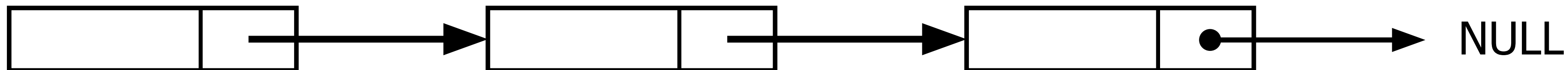



```
struct list {
    int data ;
    struct list *next ;
} ;
```

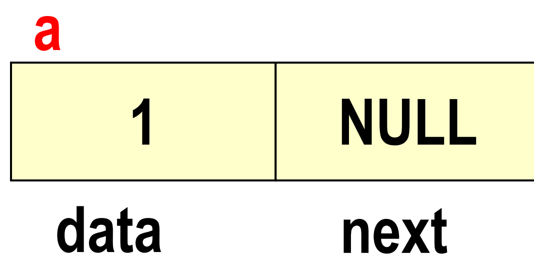
The pointer variable next contains either

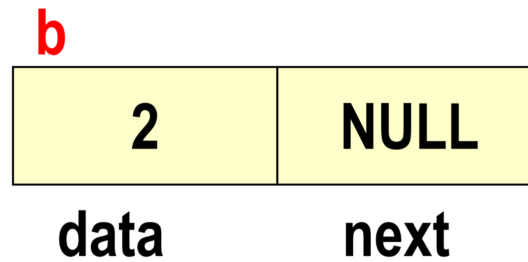
- address of the location in memory of the successor list element
- or the special value **NULL** defined as 0.

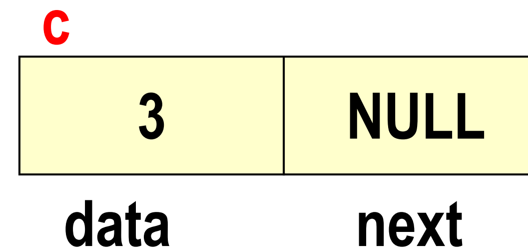
NULL is used to denote the end of the list.



```c
struct list a, b, c;

a.data = 1; b.data = 2; c.data = 3;
a.next = b.next = c.next = NULL;
```

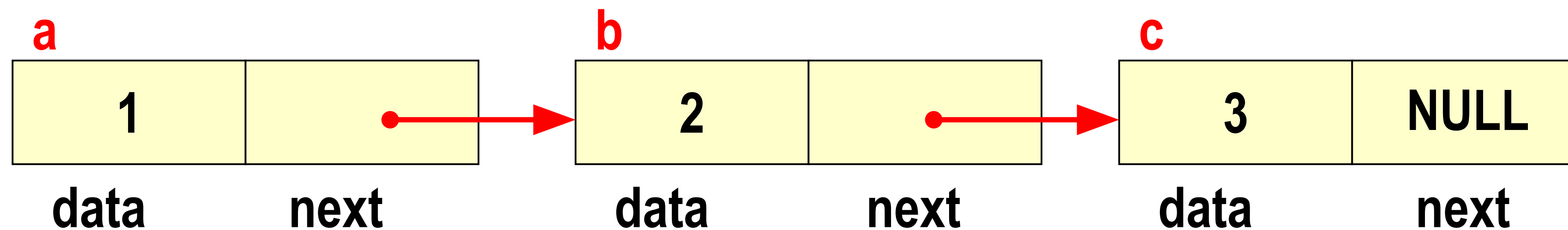






Chaining these together

a.next = &b;
b.next = &c;



What are the values of :

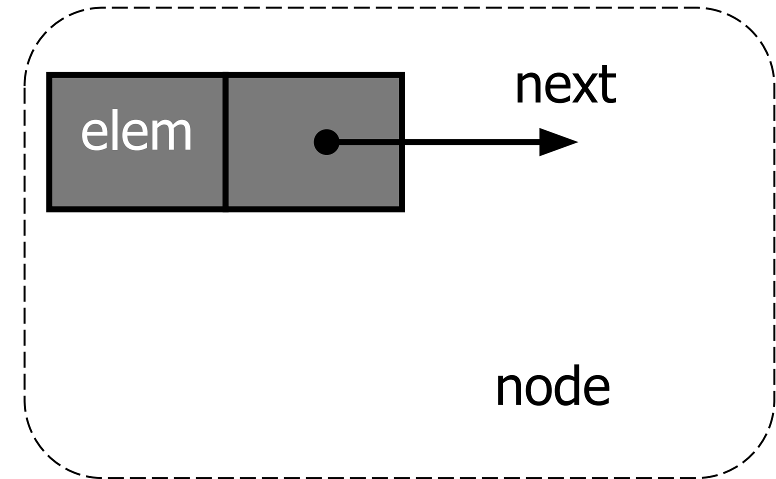
- a.next->data — 2
- a.next->next->data — 3

Linked Lists

A singly linked list is a concrete data structure consisting of a sequence of nodes

Each node stores

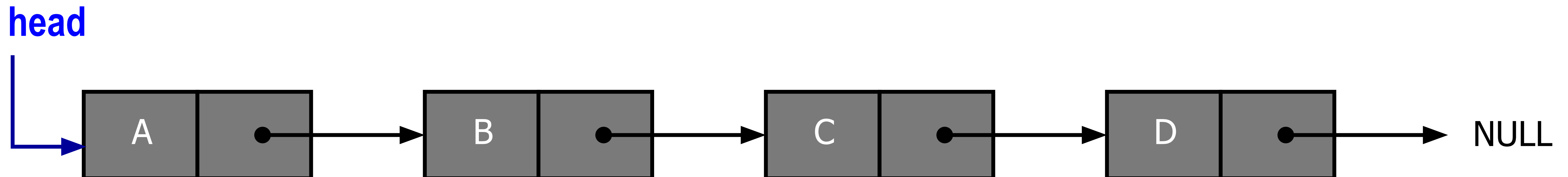
- an element
- a link to the next node



A head pointer addresses the first element of the list.

Each element points at a successor element.

The last element has a link value NULL.



Header file : list.h

```
typedef char DATA;
// In this example, we store a char in each node; can be int, float, ...
struct list {
   DATA d;
   struct list * next;
};
typedef struct list ELEMENT;
typedef ELEMENT* LINK;
```

Our own header file !!

We can place this .h file in the same directory as the .c source file, and include the header file as #include “list.h”

Dynamic memory allocation: Review

```
typedef struct {
        int hiTemp;
        int loTemp;
        double precip;
} WeatherData;
```

```
int main ( )
{
    int numdays;
    WeatherData  *days;
    scanf (“%d”, &numdays) ;
    days=(WeatherData *)malloc(sizeof(WeatherData)*numdays);
    if (days == NULL) printf (“Insufficient memory\n”);
    ...
   free (days) ;
}
```

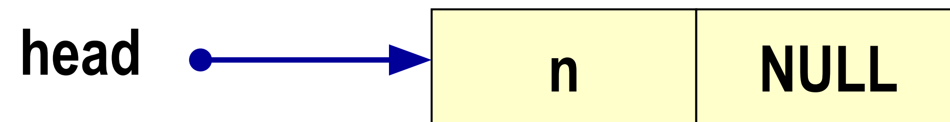
Storage allocation

```
LINK head ;
head = (LINK) malloc (sizeof(ELEMENT));
head->d = 'n';
head->next = NULL;
```

Recall:

typedef struct list ELEMENT;
typedef ELEMENT* LINK;

creates a single element list.



Storage allocation

```
head->next = (LINK) malloc (sizeof(ELEMENT));
head->next->d = 'e';
head->next->next = NULL;
```

Recall:

typedef struct list ELEMENT;
typedef ELEMENT* LINK;

A second element is added.

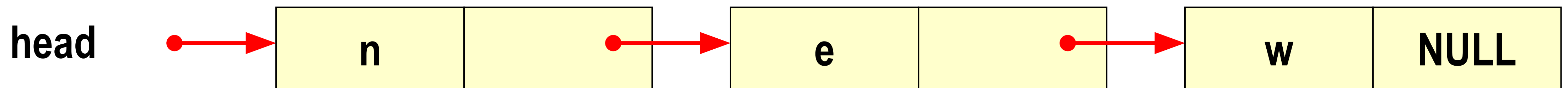
Storage allocation

```
head->next->next = (LINK) malloc (sizeof(ELEMENT));
head->next->next->d = 'w';
head->next->next->next = NULL;
```

Recall:

typedef struct list ELEMENT;
typedef ELEMENT* LINK;

We have a 3-element list pointed to by head.
The list ends when next is the NULL pointer.
So linked lists are NULL-terminated lists.



Operations on Lists

(i) How to initialize such a self-referential structure (LIST),

(ii) How to insert such a structure into the LIST,

(iii) How to delete elements from it,

(iv) How to search for an element in it,

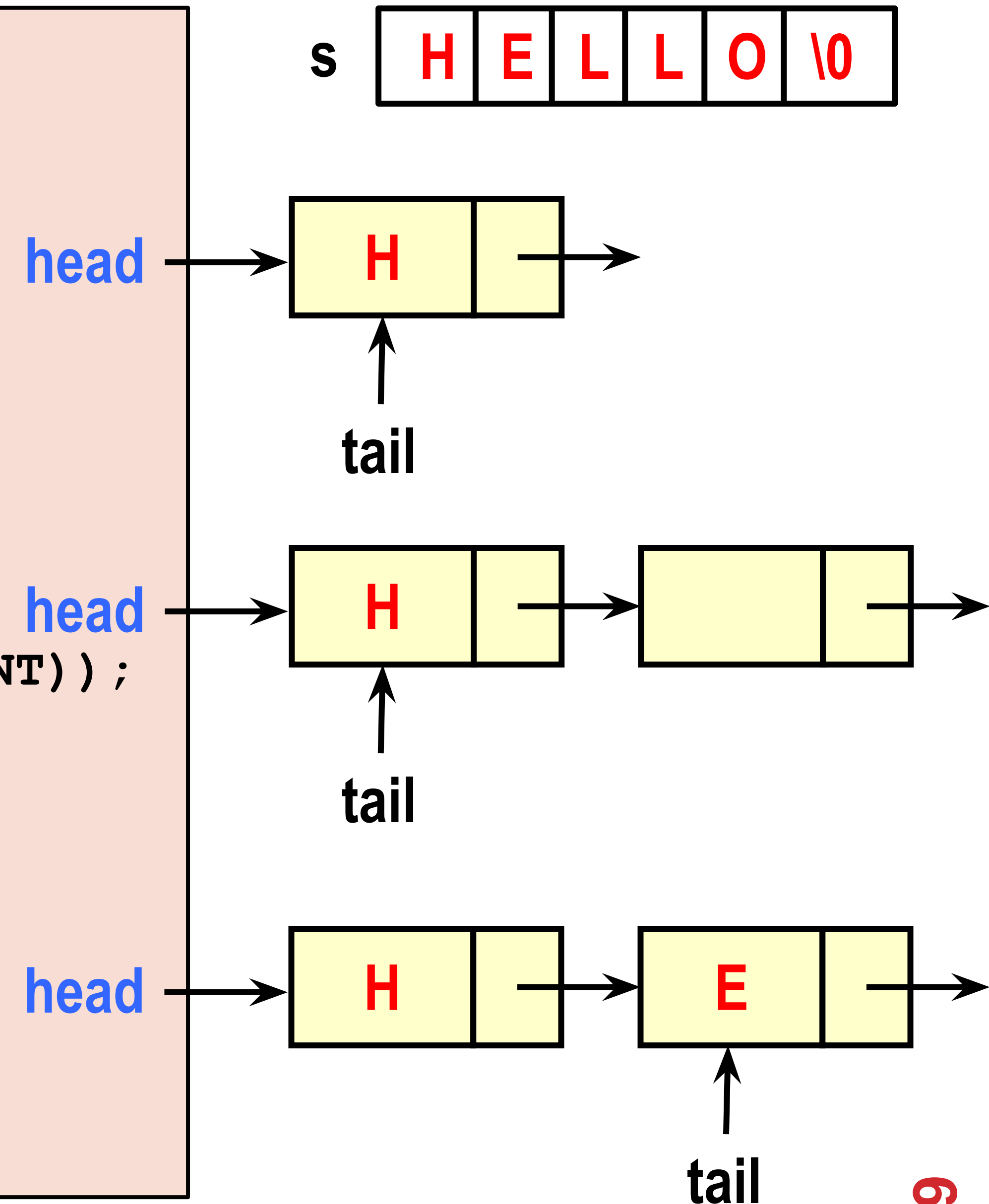
(v) How to print it,

(vi) How to free the space occupied by the LIST?

Creating a list from scratch

Produce a list from a string (each character in a node)

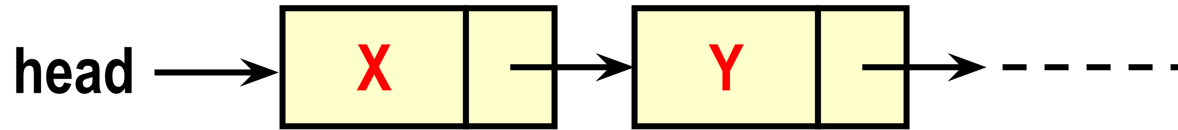
```
LINK StrToList (char s[ ]) {
    LINK head = NULL, tail;
    int  i;

    if (s[0] != '\0')   {
       head = (LINK) malloc (sizeof(ELEMENT));
       head->d = s[0];
       tail = head;
       for (i=1; s[i] != '\0'; i++)  {
          tail->next = (LINK) malloc(sizeof(ELEMENT));
          tail = tail->next;
          tail->d = s[i];
       }
       tail->next = NULL;
    }
    return head;
}
```

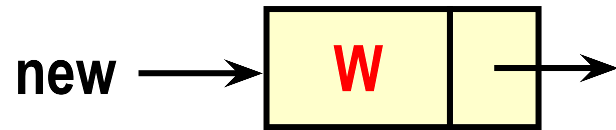


Inserting a new element (node) into an existing linked list

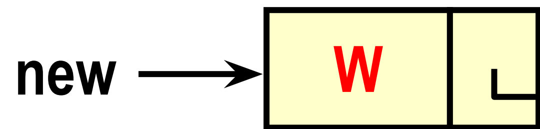
Inserting at the Head

1. Allocate a new node
2. Insert new element
3. Make new node point to old head
4. Update head to point to new node

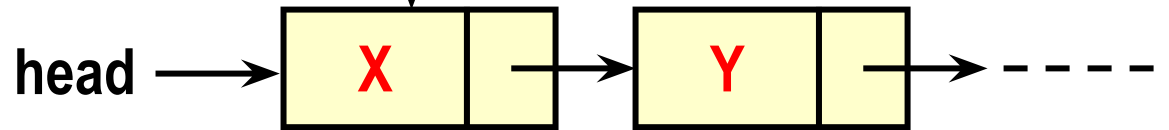


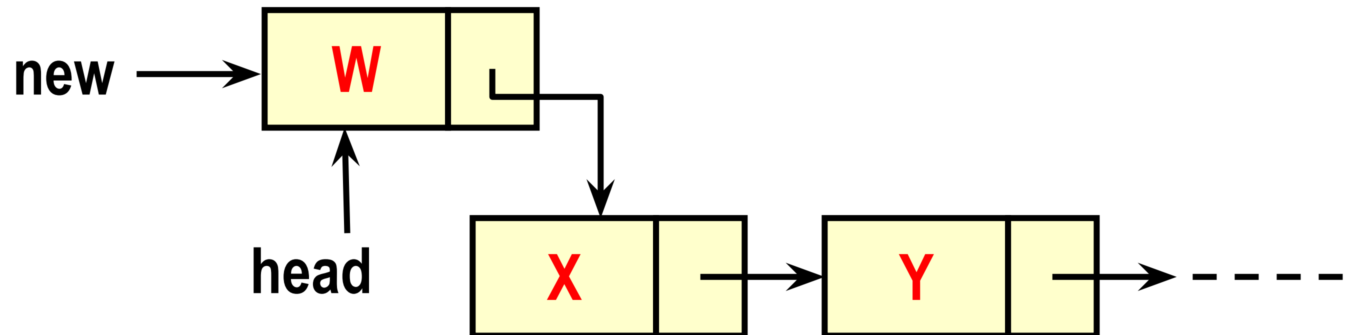



```
new = (LINK)malloc(sizeof(ELEMENT));
```




```
new-> next = head;
```

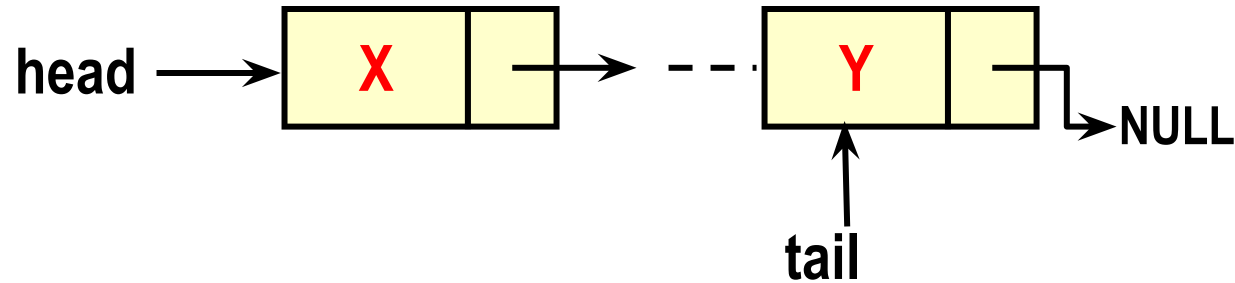




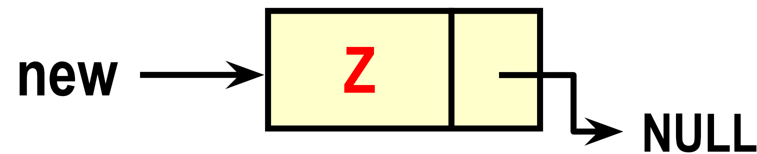

```
head = new;
```

Inserting at the Tail

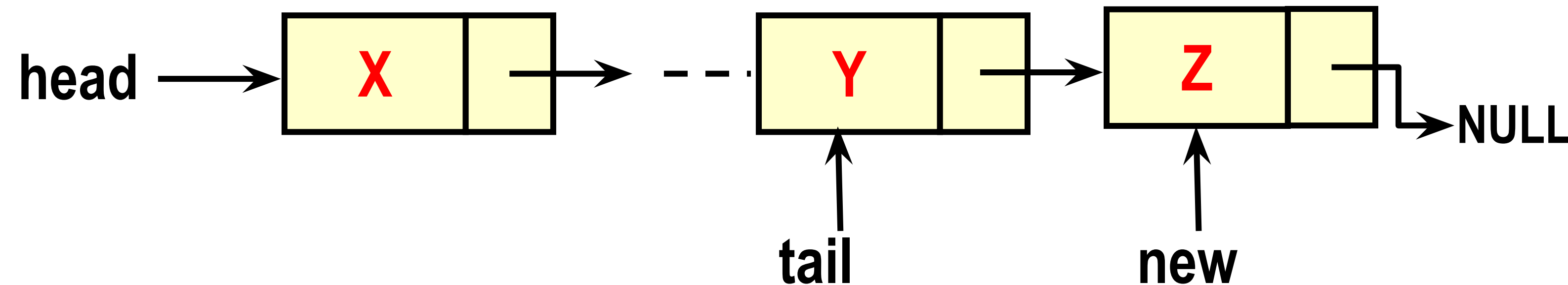
1. Allocate a new node
2. Insert new element
3. Have new node point to null
4. Have old last node point to new node
5. Update tail to point to new node



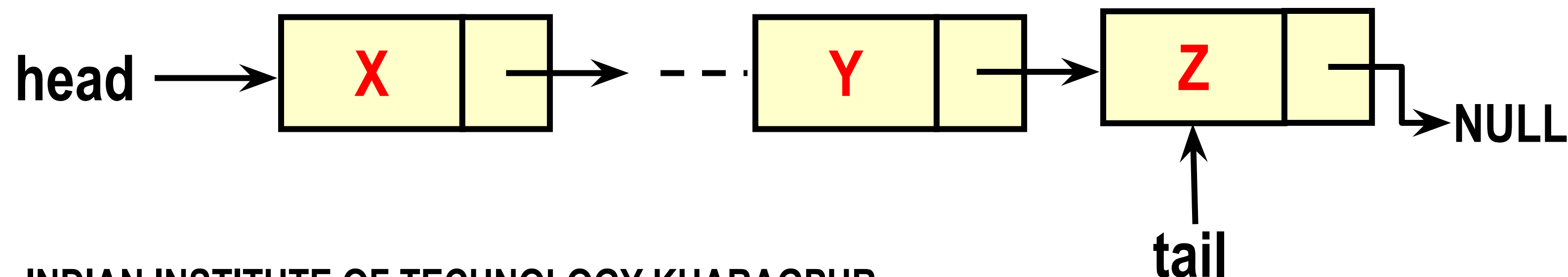

```
new = malloc(sizeof(ELEMENT));
```




```
new->next = NULL;
```




```
tail->next = new;
```




```
tail = new;
```

Insertion into a sorted list

Create a *new* node containing the data

Find the correct place in the list, and

Link the *new* node at this place.

newp → N

head → A → - - - M → P → - - -

prev (points to M), ptr (points to P)

```
newp = malloc(sizeof(ELEMENT));
```

head → A → - - - M → N → P → - - -

prev (points to M), ptr (points to P), newp → N

```
prev -> next  = newp;
newp -> next = ptr;
```

We will ensure that the pointer prev is always one element behind pointer ptr. The new node is to be inserted between prev and ptr

Insertion into a sorted list: Implementation

```
struct list {
   int data;
   struct list *next;
};
typedef struct list ELEMENT;
typedef ELEMENT *LINK;
```

Note: The Tail pointer is not maintained.

```
LINK create_node(int val)
{
   LINK newp;
   newp = (LINK) malloc (sizeof(ELEMENT));
   newp -> data = val;
   return newp;
}
```

Insertion function

```
LINK insert (int value,  LINK ptr)
{
   LINK newp, prev, first;
   newp = create_node(value);
   if (ptr == NULL || value <= ptr-> data) { // insert as new first node
      newp -> next = ptr;
      return newp;   // return pointer to first node
   } else {       //  insert in the middle (not first element)
      first = ptr;   // remember start
      prev = ptr;
      ptr = ptr-> next;
      while (ptr != NULL  && value > ptr -> data) {
         prev = ptr; ptr  = ptr -> next;
      }
      prev -> next = newp;  // link in
      newp -> next = ptr;   //new node
      return first;
   }
}
```

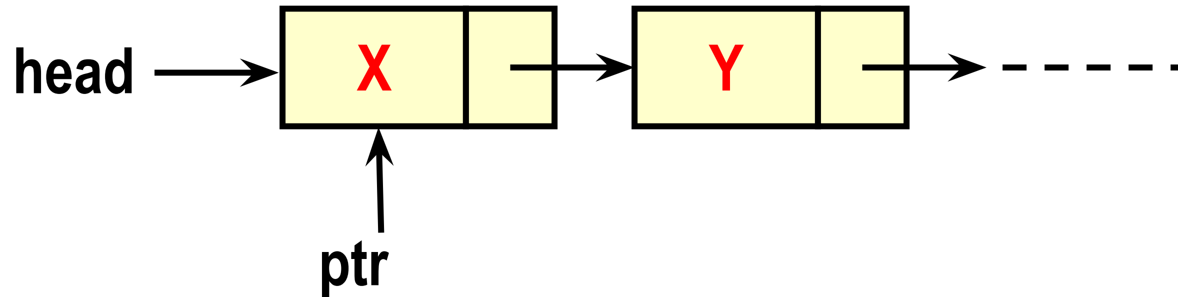
The insert function is called as
head = insert(val, head);
where val is the new value to be inserted

We assume the list is kept sorted in increasing order of *data* in the nodes.

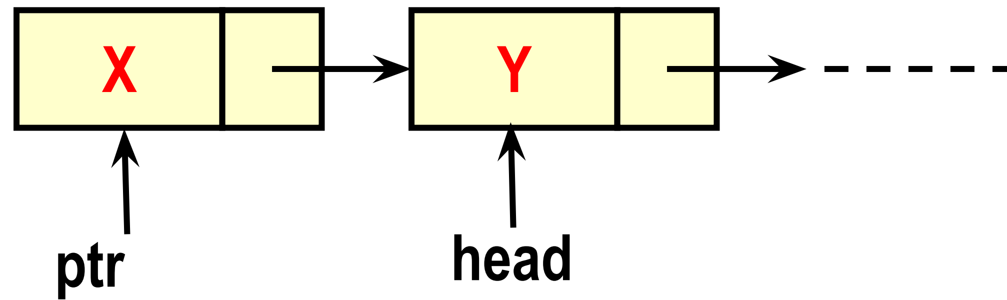
Deleting an element (node) from an existing list

Removing the first node

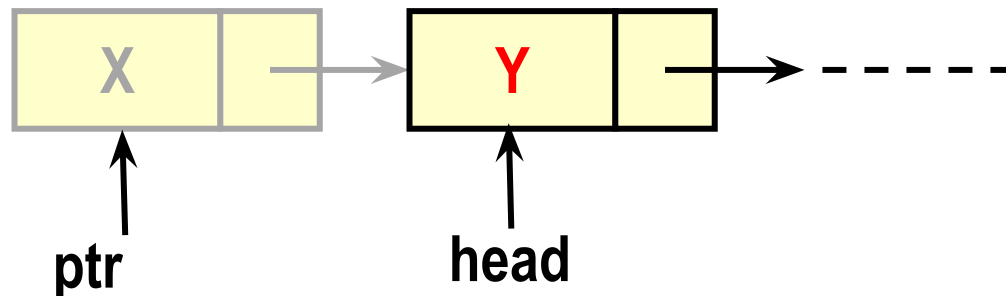
1. Update head to point to next node in the list
2. Free the former first node




```
ptr = head;
```



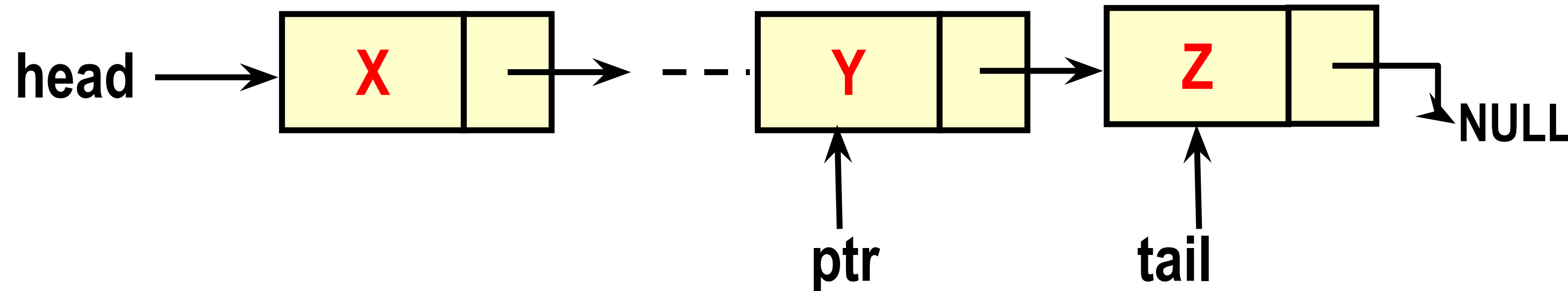

```
head = ptr->next;
```

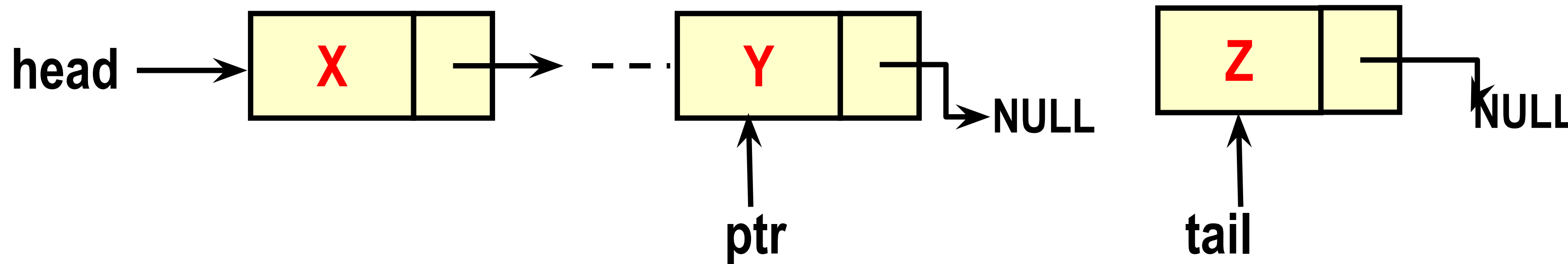



```
free(ptr);
```

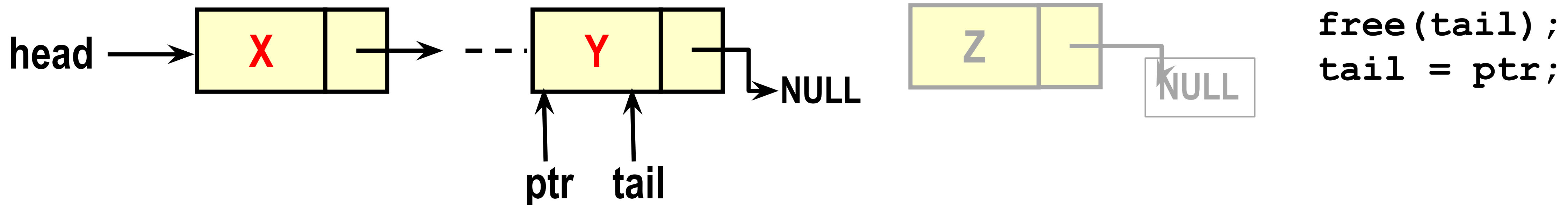
Removing the Tail (last node)

1. Bring ptr to the second last node
2. Make ptr->next equal to NULL
3. Free tail
4. Make ptr the new tail





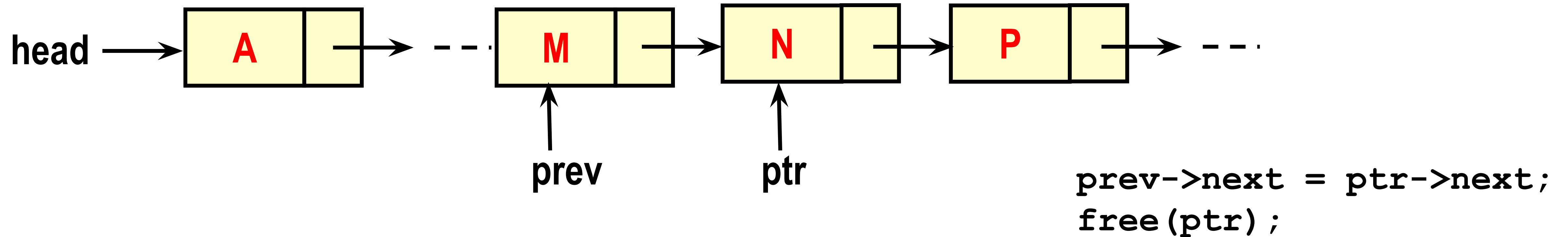

```
ptr->next = NULL;
```



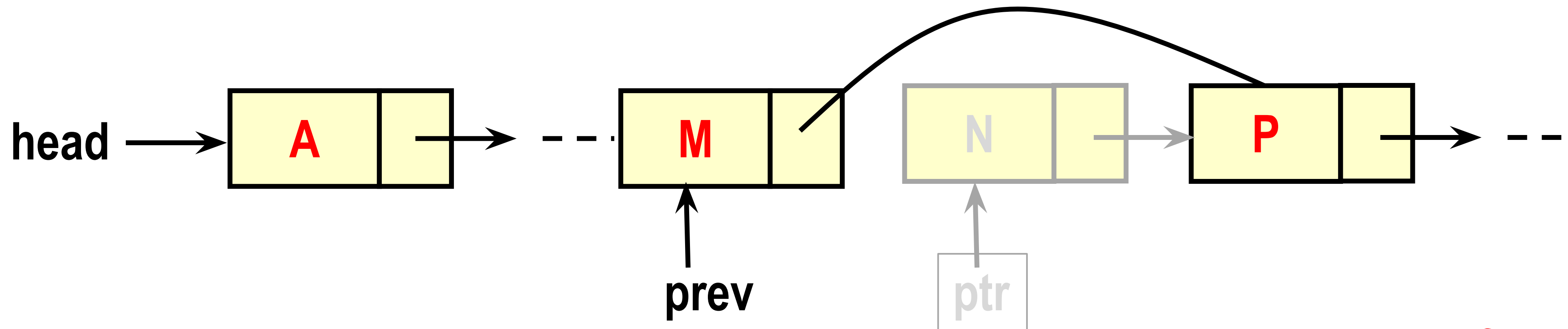

```
free(tail);
tail = ptr;
```

Deletion from the middle of the list
Item to delete: N

Steps:
- Find the item (to be deleted) in the list
- Bring prev to the previous node
- Link out node to be deleted, and
- Free up this node as free space.




```
prev->next = ptr->next;
free(ptr);
```



What will happen if we did the following?

```
free(ptr);
prev->next = ptr->next;
```

Deletion function

```
//  delete the item from a sorted list
LINK delete_item (int val, LINK ptr)
{
   LINK prev, first;

   first = ptr;  // remember start

   if (ptr == NULL) return NULL;

   if (val == ptr -> data) { // first node to be deleted
       ptr = ptr -> next; // second node
       first->next = NULL;
        free(first); // free up node
        return ptr;  // 2nd node is new head
   }
```

Note: The Tail pointer is not maintained.

The function is called as
head = delete_item (val, head);
where val is the value to be deleted.

Deletion function: Continued

```
    // check rest of list
    prev = ptr;
    ptr = ptr -> next;

    // find node to delete
    while ( (ptr != NULL) && (val > ptr->data) ) {
        prev = ptr;
        ptr  = ptr -> next;
    }

    if ( (ptr == NULL) || (val != ptr->data) )  {
        // val not found in ascending list
        return first; // no change in the original list

   // val found, delete ptr node
    prev -> next = ptr -> next;
    ptr->next = NULL;
    free(ptr);        // free node
    return first; // original
}
```

Search, print, and other operations

Searching for a data element in a linked list

```
int Search( LINK head, int element )
{
  LINK temp;

    temp = head;
    while (temp != NULL)   {
      if (temp -> data == element) return 1;
      temp = temp -> next;
    }
    return 0;
}
```

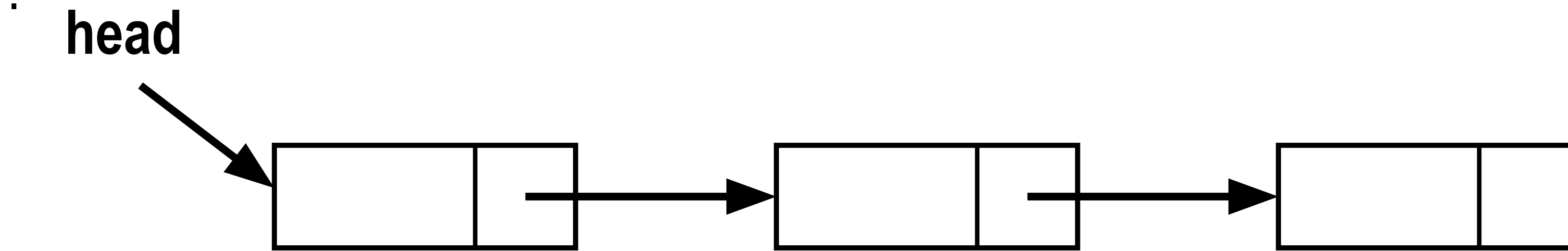
Returns 1 if element is found,
0 otherwise

Printing a linked list

```
void print_list ( LINK head )
{
    LINK temp;
    temp = head;
    while(temp != NULL) {
        if(temp->next == NULL)// for the last element
            printf("%d. END OF LIST \n", temp->data);
        else                  // for other elements
            printf("%d -> ", temp->data);
        temp = temp->next;
    }
}
```

We are dealing with the last element as a special case since we want to print ‘END OF LIST” at the end.

Printing a linked list backwards



How can you print the elements of a list backwards when the links are in forward direction ?

Printing a linked list backwards – *recursively*

```
void PrintList(LINK head)
{
   if (head == NULL) return;    /* Empty list: Nothing to print */
   if (head -> next == NULL) { /* boundary condition to stop recursion */
      printf ("%d", head -> data);
      return;
   }
   PrintList(head -> next);       /* calling function recursively */
   printf(" %d", head -> data);  /* Printing current element */
   return;
}
```

Counting the nodes in a linked list

RECURSIVE APPROACH

```
int count ( LINK head )
{
   if (head == NULL) return 0;
   return 1 + count(head->next);
}
```

ITERATIVE APPROACH

```
int count ( LINK head )
{
   int cnt = 0;
   while (head) {
       ++cnt;
       head = head->next;
   }
   return cnt;
}
```

Freeing all the nodes of a linked list

In each iteration temp1 points to the head of the list and temp2 points at the second node.

```
void FreeAll ( ELEMENT *head )
{
   ELEMENT *temp1, *temp2;
   temp1 = head;
   while (temp1 != NULL)  {
   temp2 = temp1 -> next;
      free(temp1);
   temp1 = temp2;
   }
}
```

Recursive approach

```
void FreeAll ( ELEMENT *head )
{
   if (head == NULL) return;

   /* Recursively free the
      rest of the list */
   FreeAll(head -> next);

   /* Free the first node */
   free(head);
}
```

What will happen if we free the first node of the list without placing a pointer on the second?

What will happen if we traverse a freed linked list?

Practice Problems

1. Concatenate two lists (iteratively)
2. Reverse a list
3. Delete the maximum element from a list
4. Rotate the list by k positions counter-clockwise

For each of the above, first create the linked list by reading in integers from the keyboard and inserting one by one to an empty list

Split a linked list of integers into two sublists as follows. The new lists must use the same nodes of the original list, that is, do not malloc, but adjust the links only.

5. The two sublists are the first and the second halves of the original list.
6. The first sublist consists of the odd integers, and the second sublist the even integers of the original list.