Linked Representation of Linear List

What is a pointer ?

- A **pointer** is a variable whose value is the address of another variable, i.e., direct address of the memory location.

  - A pointer can contain the memory address of any variable type
    - A primitive (int, char, float)
    - An array
    - A struct or union
    - Dynamically allocated memory
    - Another pointer
    - A function

Why Pointers?

- They allow you to refer to large data structures in a compact way
- They facilitate sharing between different parts of programs
- They make it possible to get new memory dynamically as your program is running
- They make it easy to represent relationships among data items.

Pointer Operations in C

- Creation

  `int *ptr=&`*variable* Returns variable's memory address

- Dereference

  **ptr* Returns contents stored at address

- Indirect assignment

  **ptr*=*val* Stores value at address

- Assignment

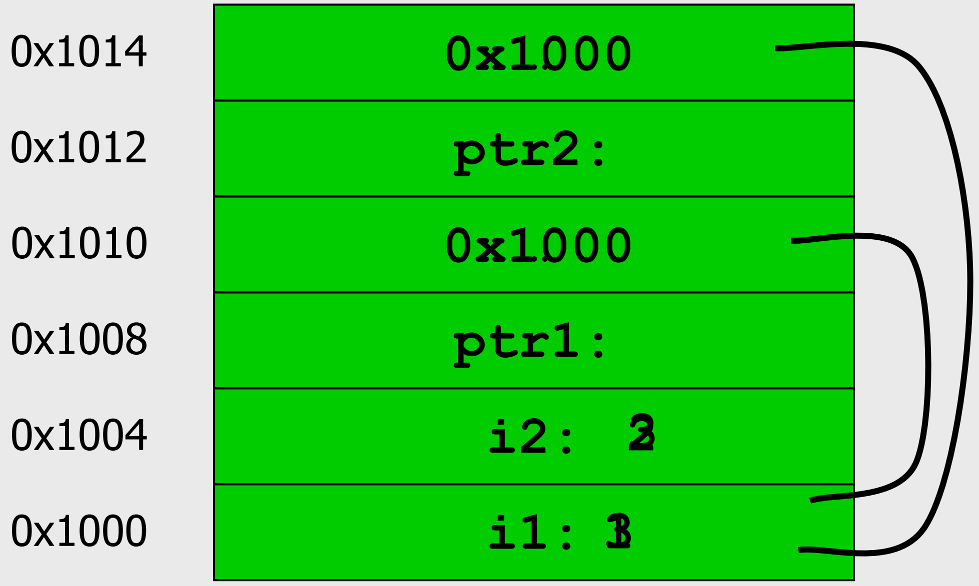
  ptr=*ptr1* Stores pointer in another variable

Using Pointers

```
int  i1;
int  i2;
int *ptr1;
int *ptr2;

i1 = 1;
i2 = 2;
ptr1 = &i1;
ptr2 = ptr1;

*ptr1 = 3;
i2 = *ptr2;
```



Pointers and Arrays in C

- An array name by itself is an address, or pointer in C.
- An array name is a particular fixed address that can be thought of as a fixed or constant pointer.
- When an array is declared, the compiler allocates sufficient space beginning with some base address to accommodate every element in the array.
- The base address of the array is the address of the first element in the array (index position 0).

Pointers and Arrays in C (cont.)

- Suppose we define the following array and a pointer:

  ```
  int   a[100],  *ptr;
  ```

  Assume that the system allocates memory bytes 400, 404, 408, ..., 796 to the array. Assume that integers are allocated 32 bits = 4 bytes.

  - The two statements: ptr = a; and ptr = &a[0]; are equivalent and would assign the value of 400 to ptr.

- Pointer arithmetic provides an alternative to array indexing in C.

  - The two statements: ptr = a + 1; and ptr = &a[1]; are equivalent and would assign the value of 404 to ptr.

Pointers and Arrays in C (cont.)

- Assuming the elements of the array have been assigned values, the following code would sum the elements of the array:

```
sum = 0;
for (ptr = a; ptr < &a[100]; ++ptr)
    sum += *ptr;
```

- In general, if *i* is of type int, then *ptr + i* is the *i*th offset from the address of *ptr.*
- Using this, here is another way to sum the array:

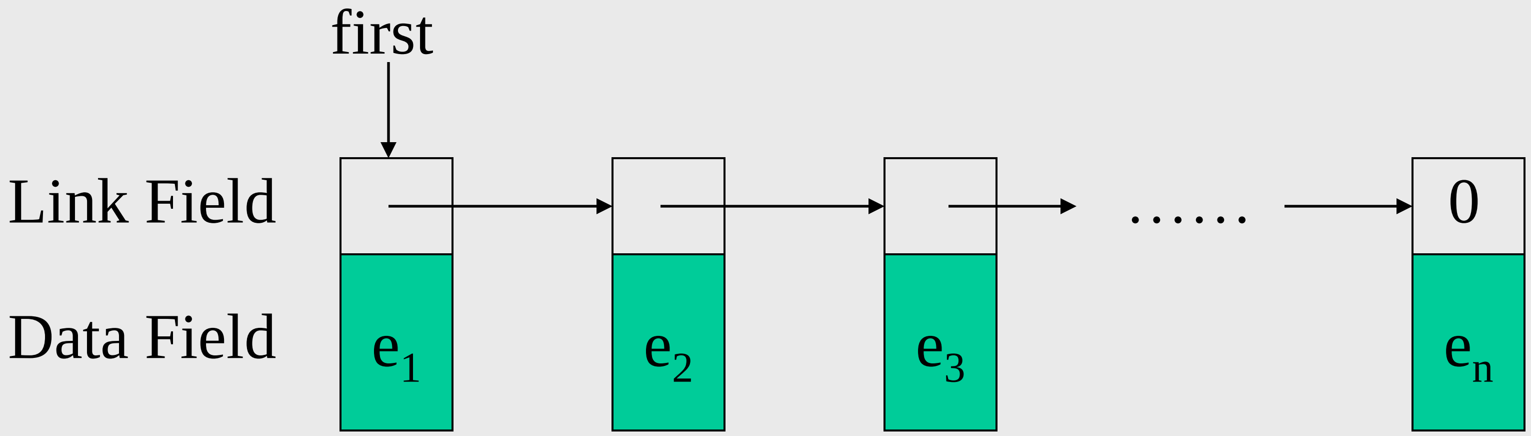
```
sum = 0;
for (i = 0; i < 100; ++i)
    sum += *(a + i);
```

Linked Representation of Linear List

- A **list** or **sequence** is an abstract data type that represents a countable number of ordered values, where the same value may occur more than once.
- A **data structure** is said to be ***linear*** if its elements form a sequence or a **linear** list.
- Each element is represented in a cell or node.
- Each node keeps explicit information about the location of other relevant nodes.
- This explicit information about the location of another node is called a link or pointer.

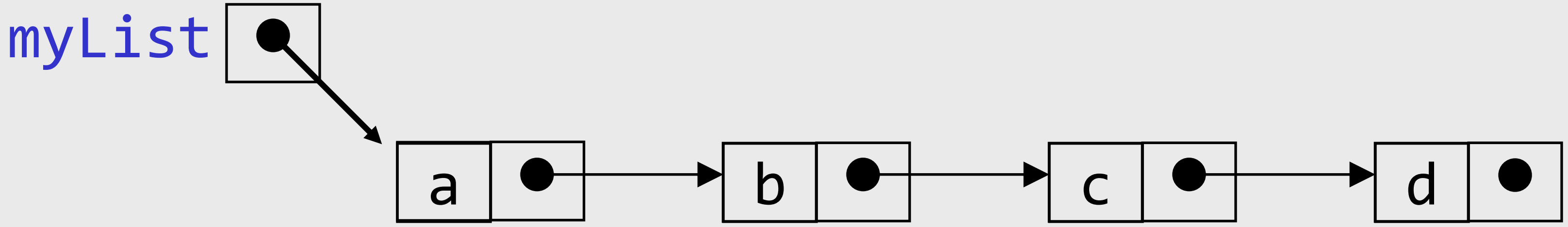
Singly Linked List

- Let $L = (e_1, e_2, \ldots, e_n)$
  - Each element e_i is represented in a separate node
  - Each node has exactly one link field that is used to locate the next element in the linear list
  - The last node, e_n, has no node to link to and so its link field is NULL.
- This structure is also called a chain.



Structure of a linked list

- A linked list consists of:
  - A sequence of nodes



- ➢ Each node contains one/more value/s and a link (pointer or reference) to some other node
- ➢ The last node contains a null link
- ➢ The list may (or may not) have a header

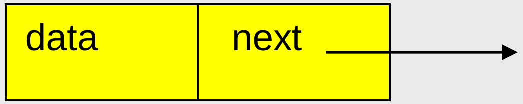
More terminologies

- Successor of a node is the next node in the sequence
  - The last node has no successor
- Predecessor of a node is the previous node in the sequence
  - The first node has no predecessor
- Length of a list is the number of elements in it
  - A list may be empty (contains no elements)

Linked List Implementation/Coding Issues in C

- We can define structures with pointer fields that refer to the structure type containing them

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



- The pointer variable next is called a *link.*
- Each node is linked to a succeeding node through field next.
- The pointer variable next contains an address of either the location in memory of the successor struct list element or the special value NULL.

Accessing structure fields

- Given
  - a struct s containing a field f to access f, we write s.f

Example:

declares x, y to be variables of type "struct abc"

```
struct abc {
    int count, bar[10];
} x, y;

x.count = y.bar[3];
```

- Given
  - a pointer **p** to a struct **s** containing a field **f**

  to access f we write

  p->**f** // equiv. to: (*__p__).**f**

Example:

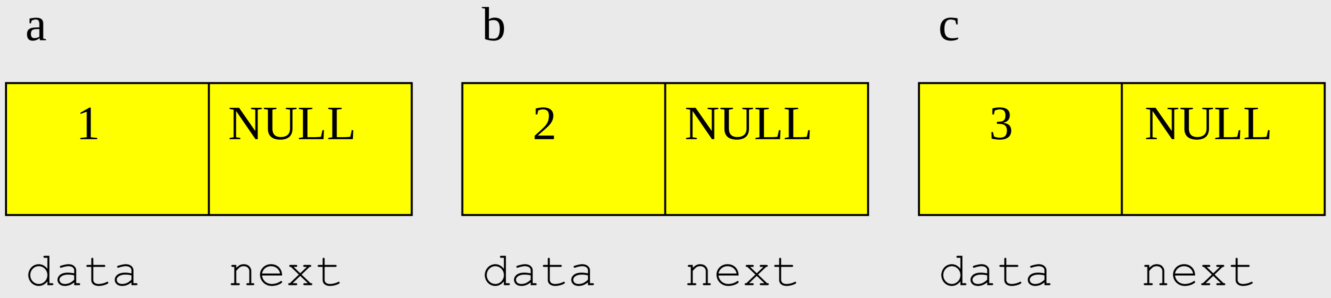
```
struct abc{
    int count, bar[10];
} *p, *q;

p->count = q->bar[3];
```

Example

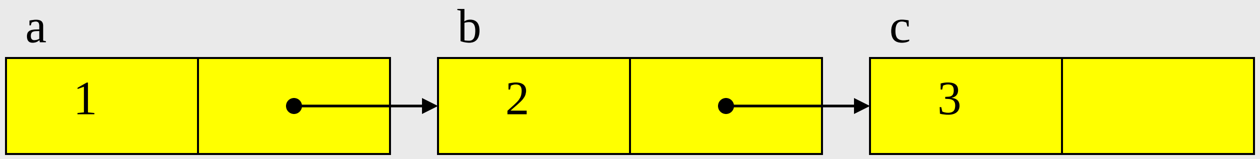
```
struct list {
        int data;
        struct list *next;
}
```

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



Example continues

- a.next = &b;
- b.next = &c;
- a.next -> data has value 2
- a.next -> next -> data has value 3
- b.next -> next -> data error !!



Dynamic Memory Allocation

- Creating and maintaining dynamic data structures requires dynamic memory allocation – the ability for a program to obtain more memory space at execution time to hold new values, and to release space no longer needed.

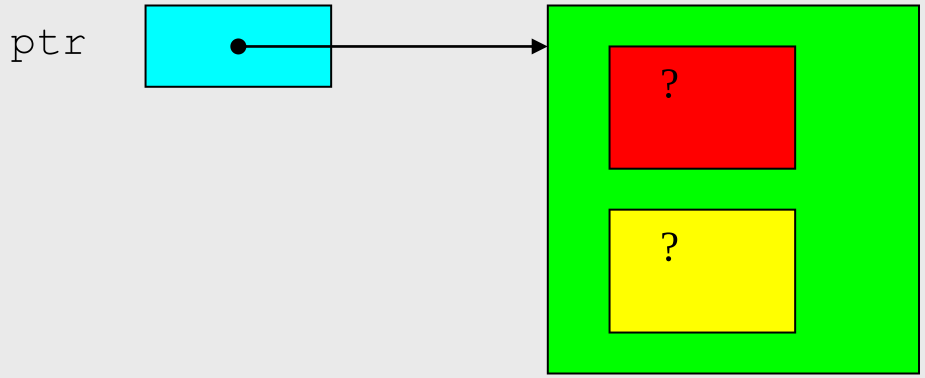
- In C, functions *malloc* and *free*, and operator *sizeof* are essential to dynamic memory allocation.

Dynamic Memory Operators *sizeof* and *malloc*

- Unary operator *sizeof* is used to determine the size in bytes of any data type.
  - sizeof(double)   sizeof(int)
  - sizeof a (double a)   sizeof b (int b)
- Function *malloc* takes as an argument the number of bytes to be allocated and returns a pointer of type void * to the allocated memory.
  - A void * pointer may be assigned to a variable of any pointer type.
  - It is normally used with the *sizeof* operator.

Dynamic Memory Operators in C Example

```
struct node{
   int data;
   struct node *next;
};
 struct node *ptr;
 ptr = (struct node *)                  /*type casting */
      malloc(sizeof(struct node));
```



The *Free* Operator in C

- Function *free* deallocates memory.
  - i.e., the memory is returned to the system so that it can be reallocated in the future.

```
ptr = (struct node *)malloc(sizeof(struct node));

        free(ptr);
```

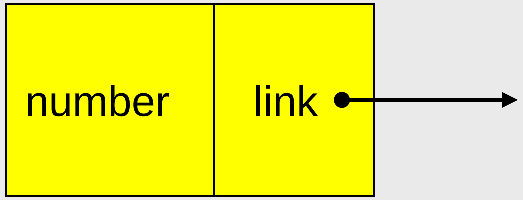
?

ptr

Examples of the Nodes of a Linked List

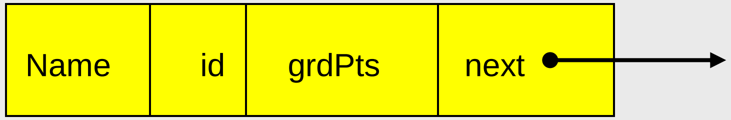
- A node in a linked list is a structure that has at least two fields.
  - data fields.
  - pointer that contains the address of the next node in the sequence.
- A node with one data field:

```
struct node{
  int number;
  struct node *link;
};
```



The Nodes of a Linked List – Examples

- A node with three data fields:



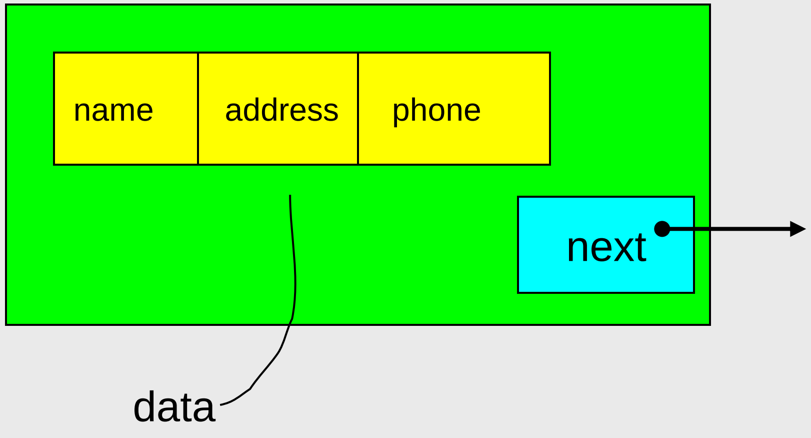

```
struct student{
   char name[20];
   int id;
   double grdPts;
   struct student  *next;
};
```

The Nodes of a Linked List – Examples

- A structure in a node (nested):

```
struct person{
   char name[20];
   char address[30];
   char phone[10];
};

struct person_node{
   struct person data;
   struct person_node   *next;
};
```



Basic Operations on a Linked List

1. Add a node
2. Delete a node
3. Search for a node
4. Traverse (walk) the list
    - Useful for counting operations or aggregate operations

Adding a node into a SLL

- There are many ways a new node can be inserted into a list:
  - As the new first element
  - As the new last element
  - Before a given node (specified by a *reference*)
  - After a given node
  - Before a given value
  - After a given value
- All these are possible, but differ in difficulty

Adding Nodes to a Linked List

There are four steps to add a node to a linked list:

- Allocate memory for the new node.
- Determine the insertion point
  - You need to know only the new node's predecessor (Pre)
- Point the new node to its successor.
- Point the predecessor to the new node.

Pointer to the predecessor (Pre) can be in one of two states:

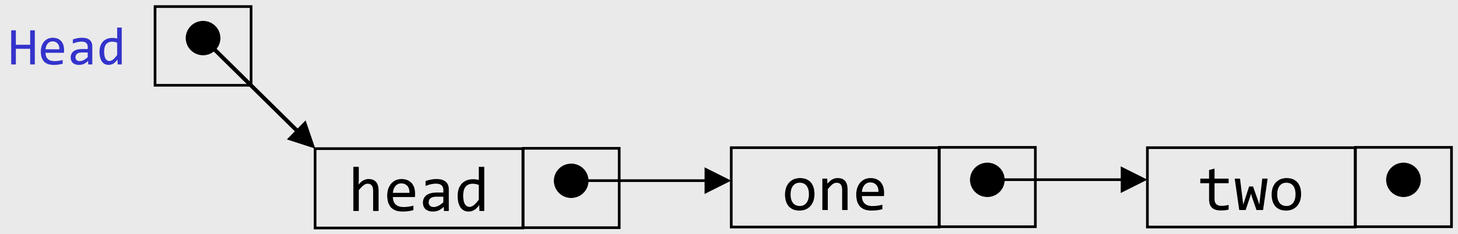
- It can contain the address of a node
  - i.e., you are adding somewhere after the first node – in the middle or at the end
- It can be NULL
  - i.e., you are adding either to an empty list or at the beginning of the list

Header Nodes

- A header linked list is a linked list which always contains a special node called the *header node* at the beginning of the list.
- It is an extra node kept at the front of a list. Such a node does not represent an item in the list. The information portion might be unused.
- Can keep global information about the entire list:
  - number of nodes, pointer to the last node

Using a header node

- The purpose is to keep the list from being null, and to point at the first element
- There are two types of header list
  - *Grounded header list* : is a header list where the last node contain the null pointer.
  - *Circular header list* : is a header list where the last node points back to the header node

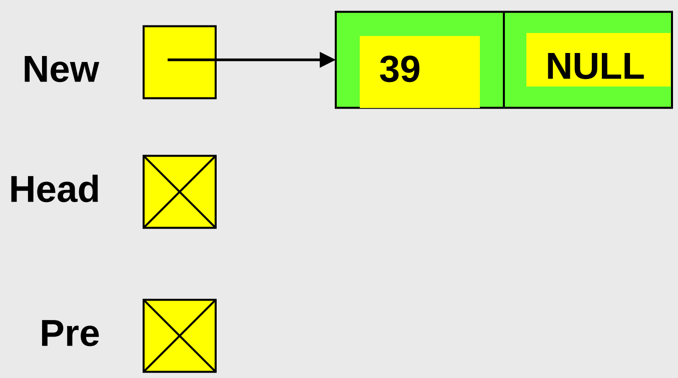


Adding Nodes to an Empty Linked List

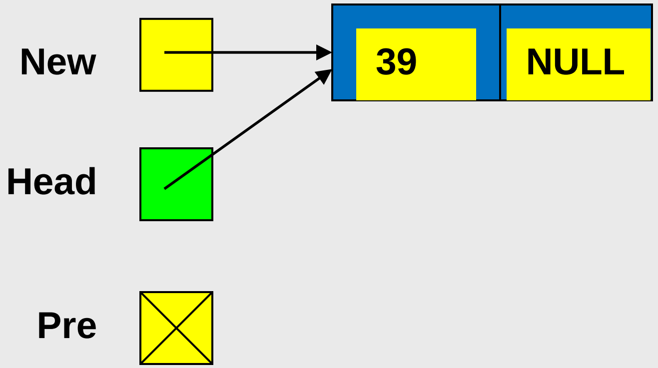
```
struct node{
    int data;
    struct node *next;
};
struct node *New, *Head, *Pre;
```

```
New = (struct node *) malloc(sizeof(struct node));
New->data=39; New -> next = NULL;
Head = New; // point list to first node
```

Before:

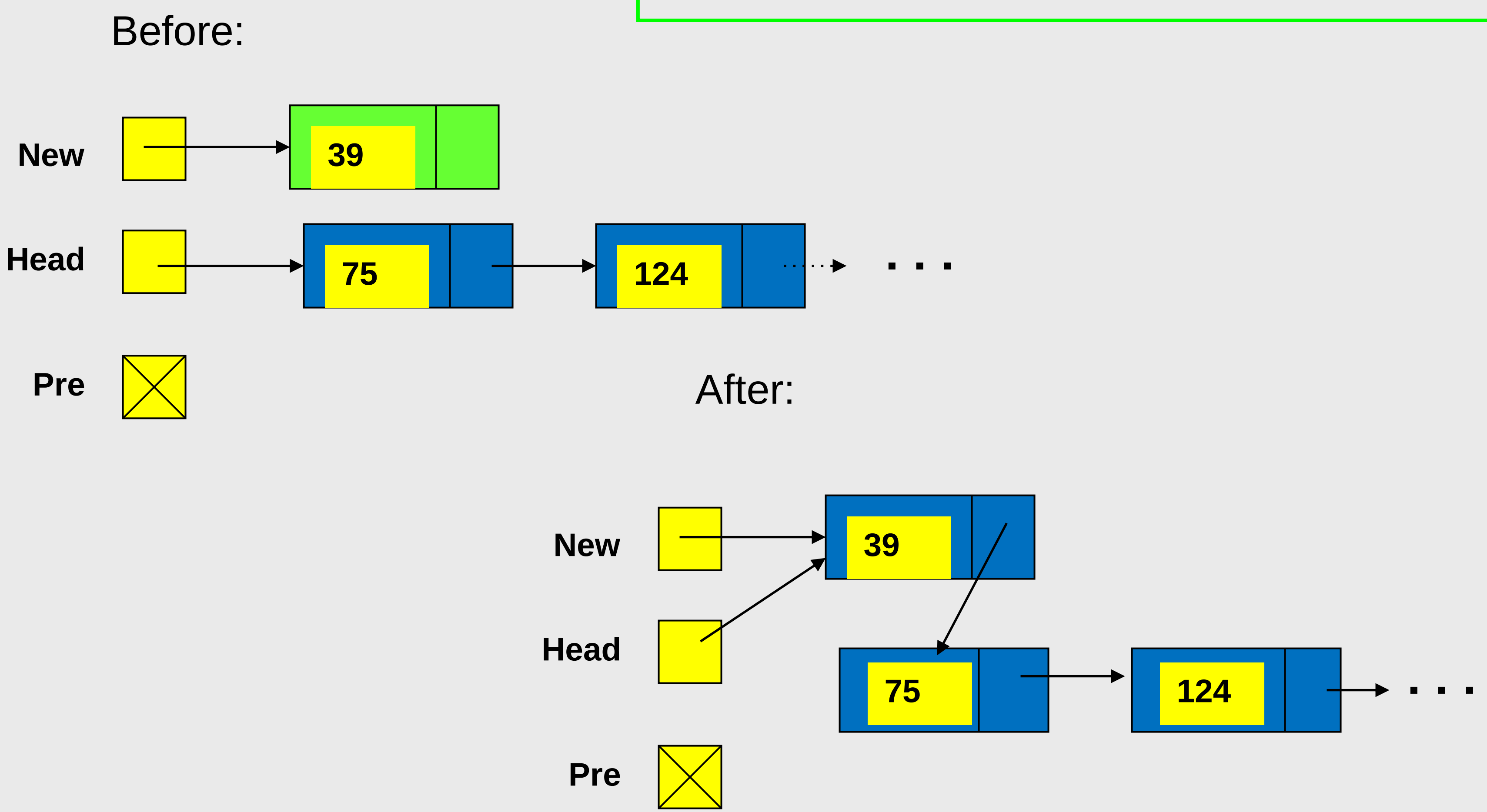


After:

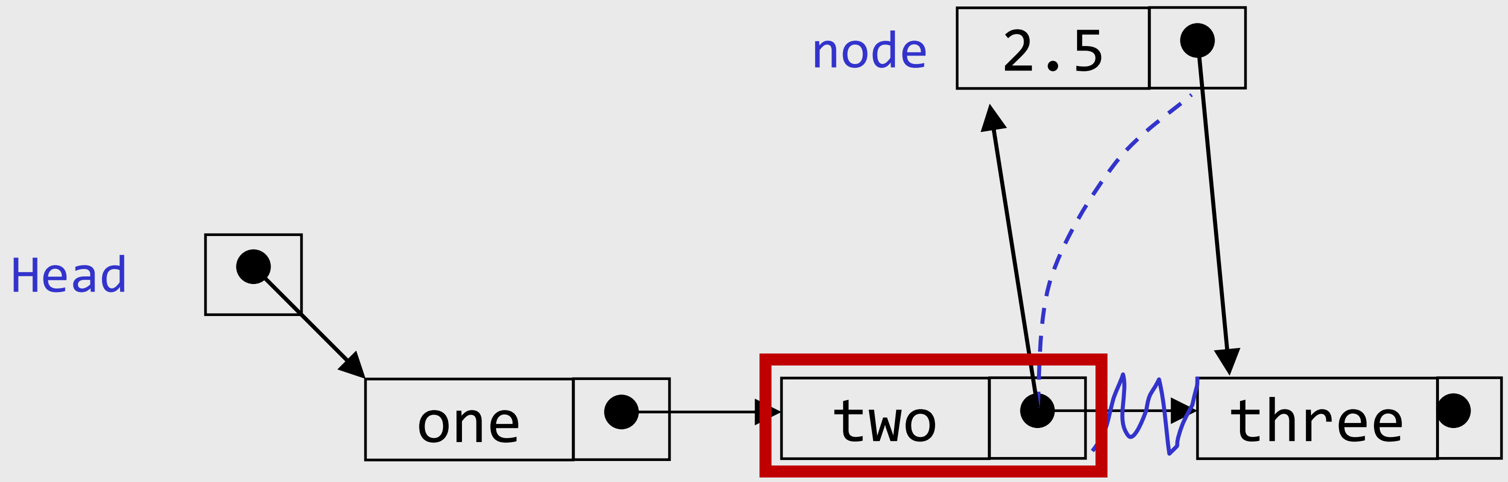


Adding a Node to the Beginning of a Linked List

```
New -> next = Head;  //set link to first node
Head = New;  //Head points to first node
```



Adding/Inserting after



Find the node you want to insert after

First, copy the link from the node that's already in the list

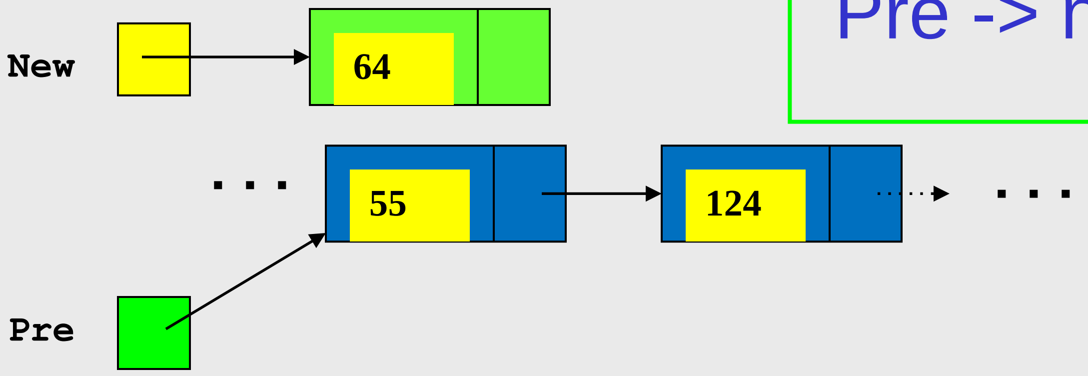
Then, change the link in the node that's already in the list

Adding a Node to the Middle of a Linked List

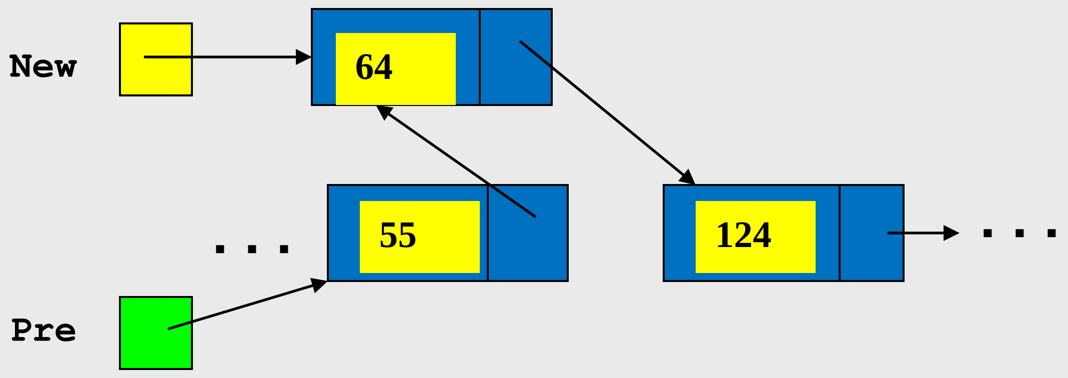
Before:

New -> next = Pre -> next;

Pre -> next = New;

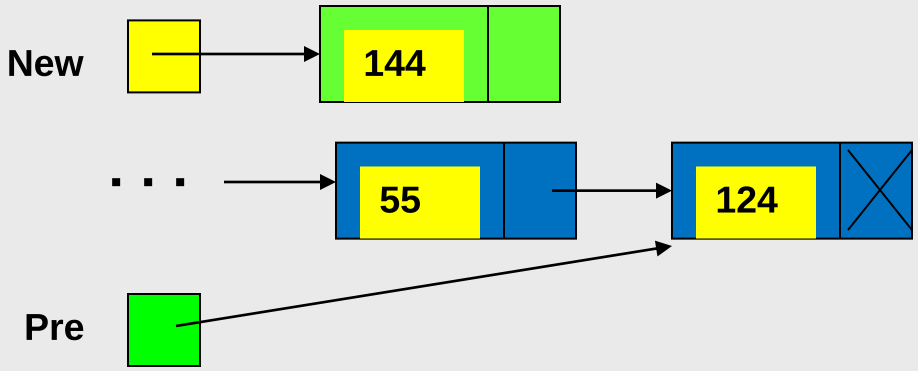


After:



Adding a Node to the End of a Linked List

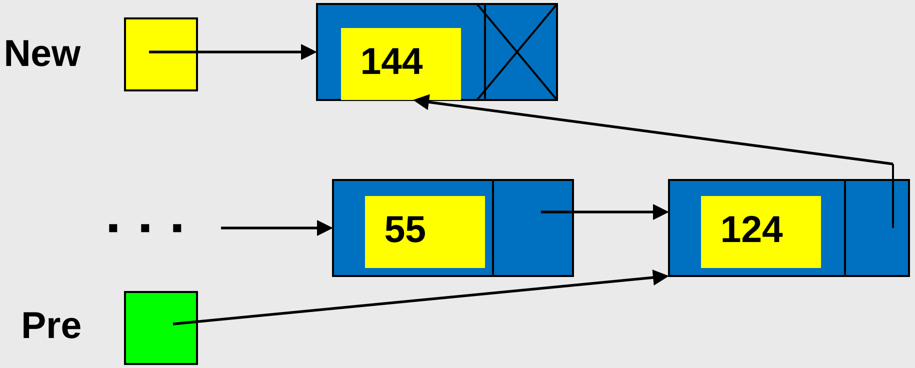
Before:



Code

```
New -> next = NULL;
Pre -> next = New;
```

After:

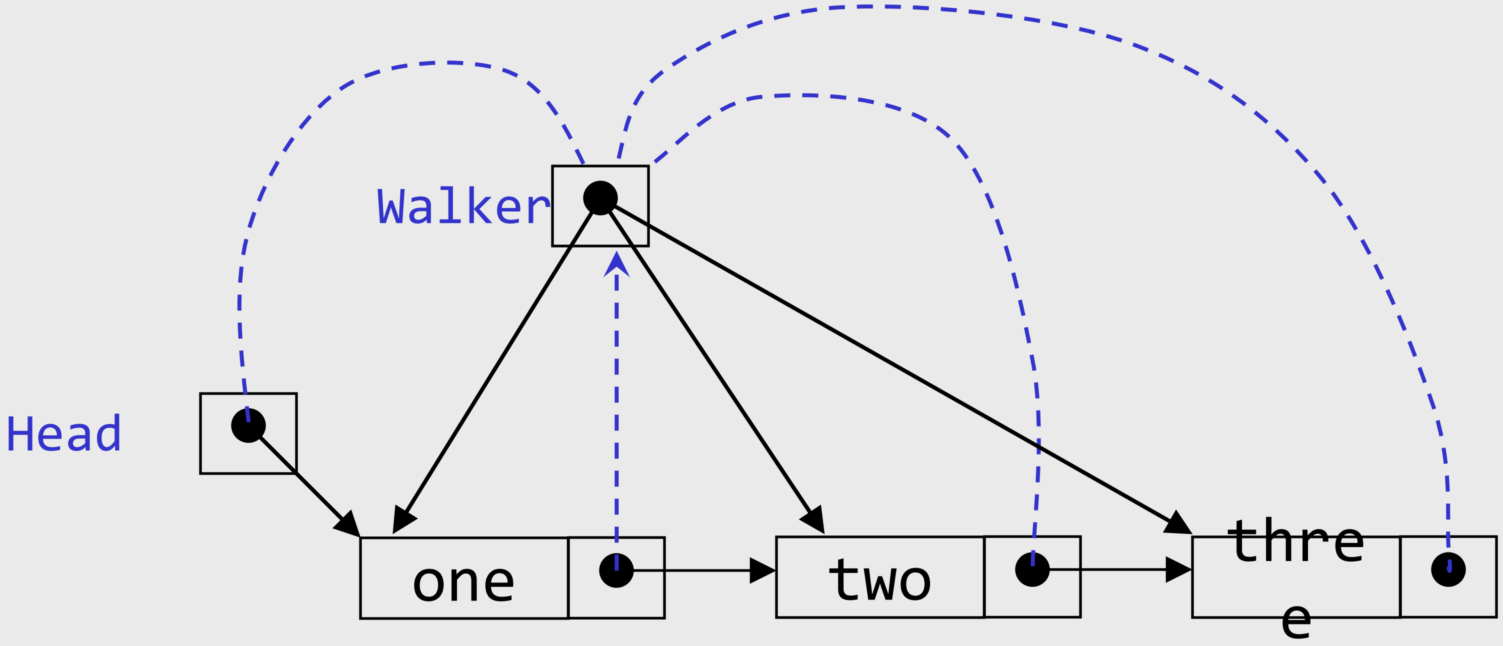


Inserting a Node Into a Linked List

- Given the head pointer (Head), the predecessor (Pre) and the data to be inserted (item). Memory must be allocated for the new node (New) and the links properly set.

```
//insert a node into a linked list
  struct node *New;
  New = (struct node *)  malloc(sizeof(struct node));
  New -> data = item;
  if (Pre == NULL){
      //add before first logical node or to an empty list
      New -> next = Head;
      Head = New;
  }
  else {
      //add in the middle or at the end
      New -> next = Pre -> next;
      Pre -> next = New;
  }
```

Traversing a SLL (animation)



Traversing a Linked List

- List traversal requires that all of the data in the list be processed. Thus, each node must be visited, and the data value examined.

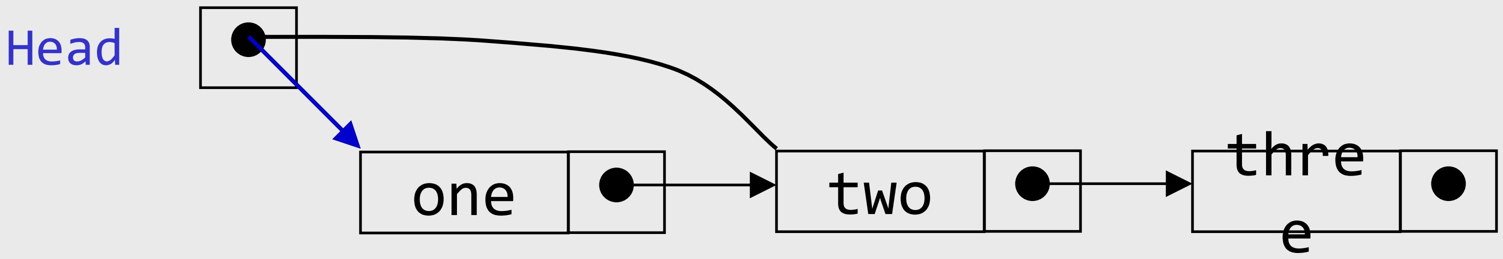
```
//traversing a linked list
struct node *Walker;
Walker = Head;
printf("List contains:\n");
while (Walker != NULL){
    printf("%d ", Walker -> data);
    Walker = Walker -> next;
}
```

Deleting a Node from a Linked List

- Deleting a node requires that we logically remove the node from the list by changing various links and then physically deleting the node from the list (i.e., return it to the heap).
- Any node in the list can be deleted.
  - Note that if the only node in the list is to be deleted, an empty list will result. In this case the head pointer will be set to NULL.
- To logically delete a node:
  - First locate the node itself (Cur) and its logical predecessor (Pre).
  - Change the predecessor's link field to point to the deleted node's successor (located at Cur -> next).
  - Recycle the node using the free( ) function.

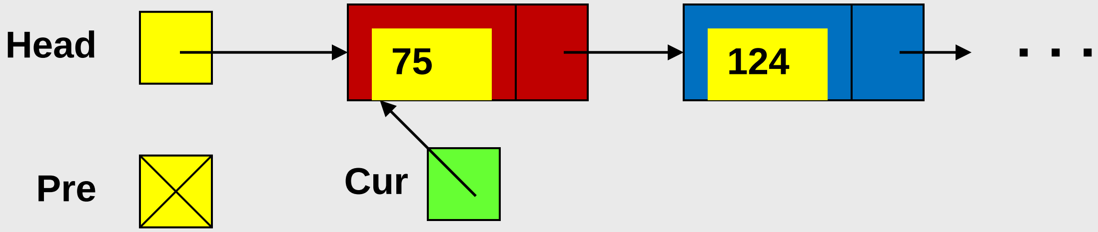
Deleting an element from a SLL

- To delete the first element, change the link in the header



Deleting the First Node from a SLL

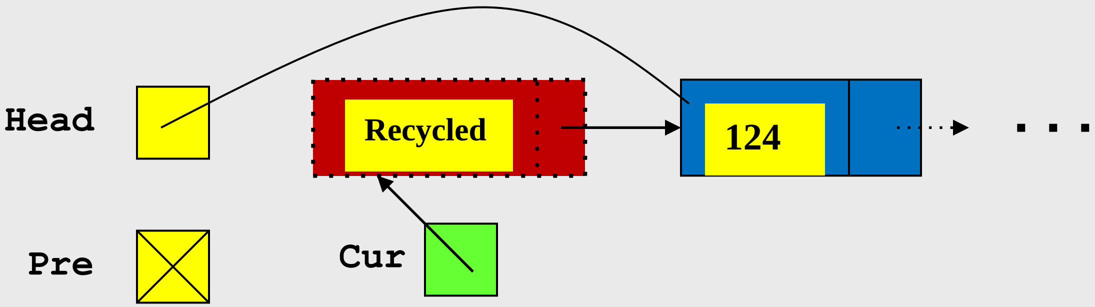
Before:



Code:

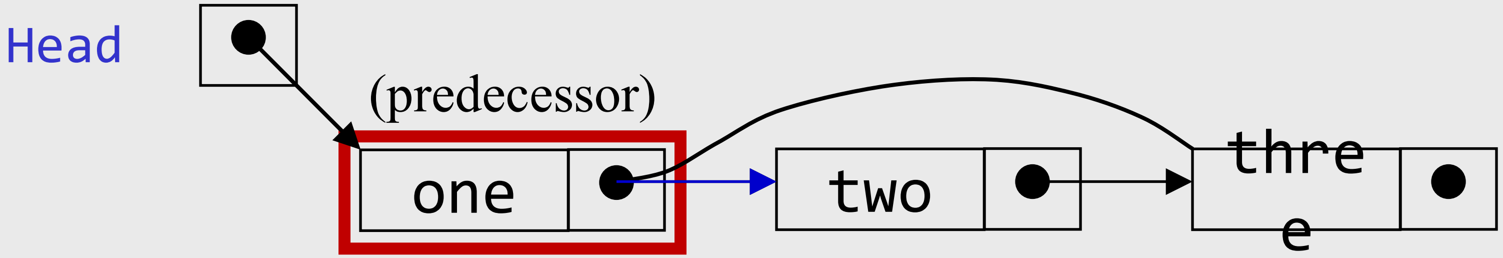
```
Head = Cur -> next;
free(Cur);
```

After:



Deleting an element from a SLL

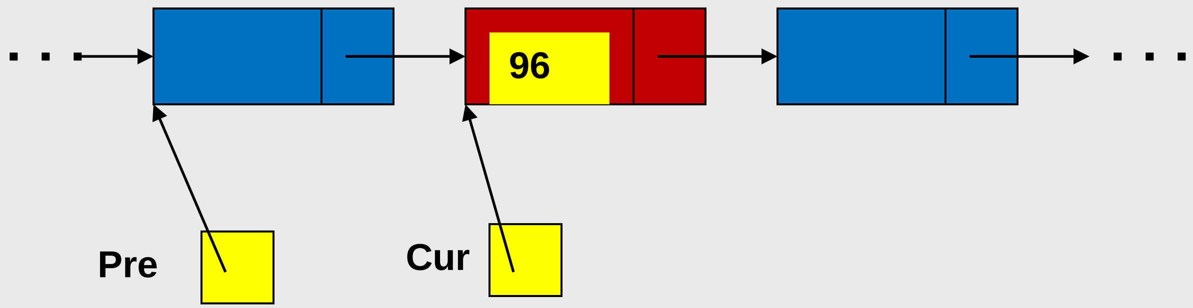
- To delete some other element, change the link in its predecessor



- Deleted nodes will eventually be garbage collected

Deleting a Node from a Linked List – General Case

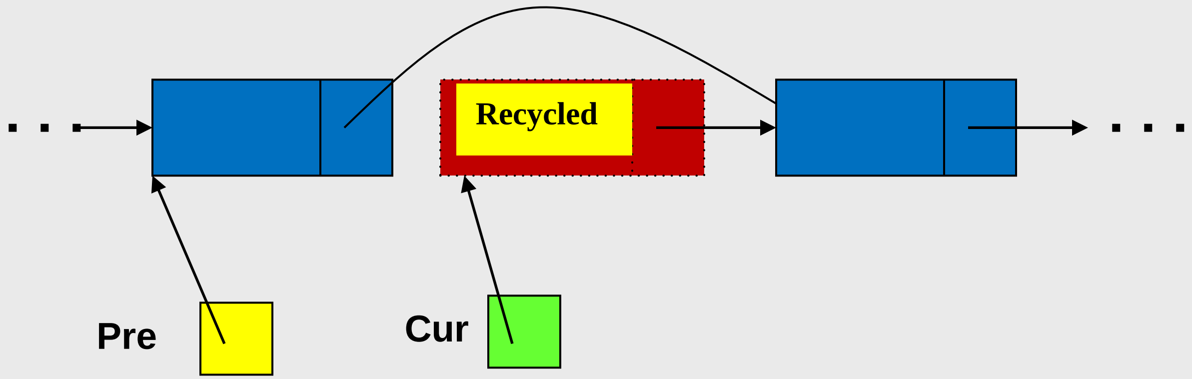
Before:



Code:

```
Pre -> next = Cur -> next;
free(Cur);
```

After:



Deleting a Node From a Linked List

- Given the head pointer (Head), the node to be deleted (Cur), and its predecessor (Pre), delete Cur and free the memory allocated to it.

Initialize: Pre = NULL;   Cur = Head;

Traverse the list to the current node

(Pre=Cur; Cur=Cur->next;)

```
//Delete a node from a linked list
if (Pre == NULL)
        //Deletion is on the first node of the list
        Head = Cur -> next;
else
        //Deleting a node other than the first node of the list
        Pre -> next = Cur -> next;
free(Cur).
```

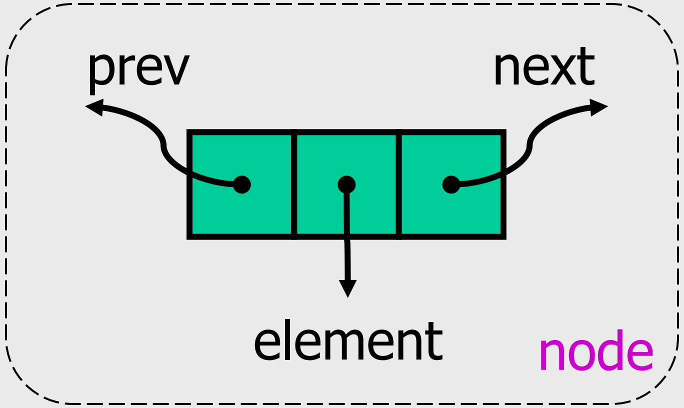
Searching a Linked List

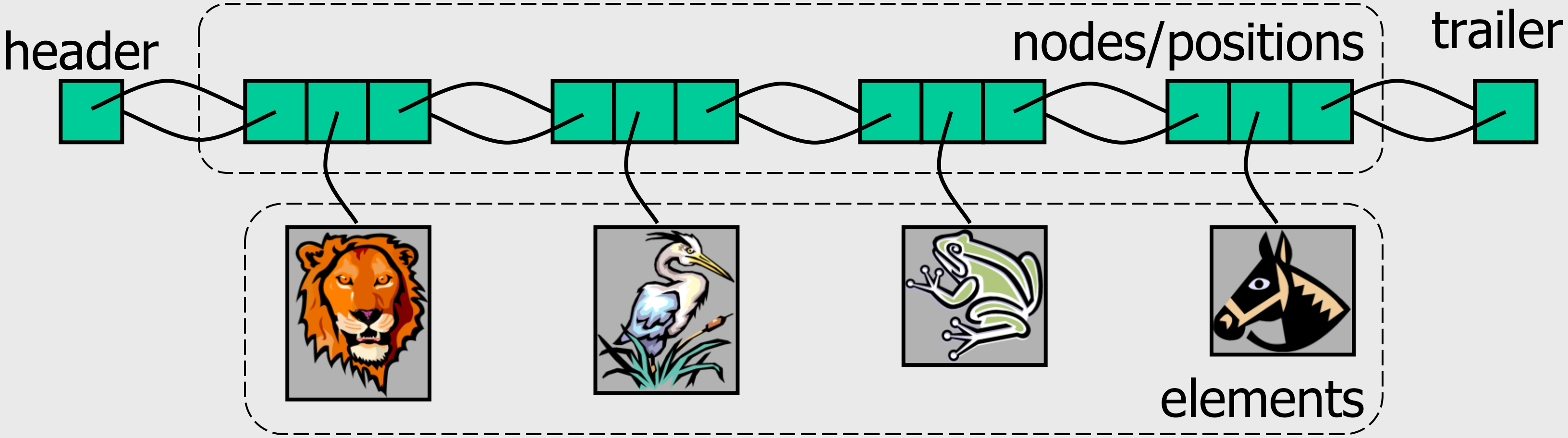
- Notice that both the insert and delete operations on a linked list must search the list for either the proper insertion point or to locate the node corresponding to the logical data value that is to be deleted.

```
//Search the nodes in a linked list
Pre = NULL;        Cur = Head;
//Search until the target value is found or the end of the list is reached
while (Cur != NULL && Cur -> data != target) {
    Pre = Cur;
    Cur = Cur -> next;
}
//Determine if the target is found or ran off the end of the list
if (Cur != NULL)
   found = 1;
else
   found = 0;
```

Doubly Linked List

- A doubly linked list provides a natural implementation of the List ADT
- Nodes implement Position and store:
  - **element**
  - **link to the previous node**
  - **link to the next node**
- Special trailer and header nodes





Doubly-Linked Lists

- It is a way of going *both* directions in a linked list, *forward* and *reverse*.
- Many applications require a quick access to the *predecessor* node of some node in list.

```
typedef struct Node{
        int data;
        struct Node *left;
        struct Node *right;
}node;
```

Basic Operations on a Doubly-Linked List

1. Add a node.
2. Delete a node.
3. Search for a node.
4. Traverse (walk) the list. Useful for counting operations or aggregate operations.

Adding Nodes to a Doubly-Linked List

There are four steps to add a node to a doubly-linked list:

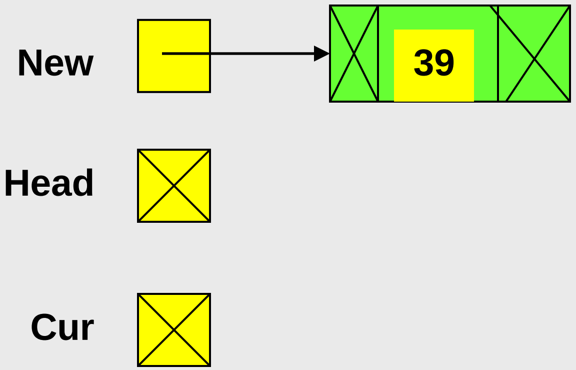
- Allocate memory for the new node.
- Determine the insertion point to be after (Cur).
- Point the new node to its successor and predecessor.
- Point the predecessor and successor to the new node.

Current node pointer (Cur) can be in one of two states:

- it can contain the address of a node (i.e. you are adding somewhere after the first node – in the middle or at the end)
- it can be NULL (i.e. you are adding either to an empty list or at the beginning of the list)

Adding Nodes to an Empty Doubly-Linked List

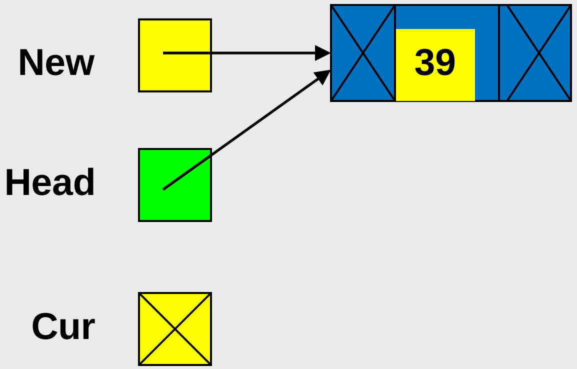
Initial:



Code:

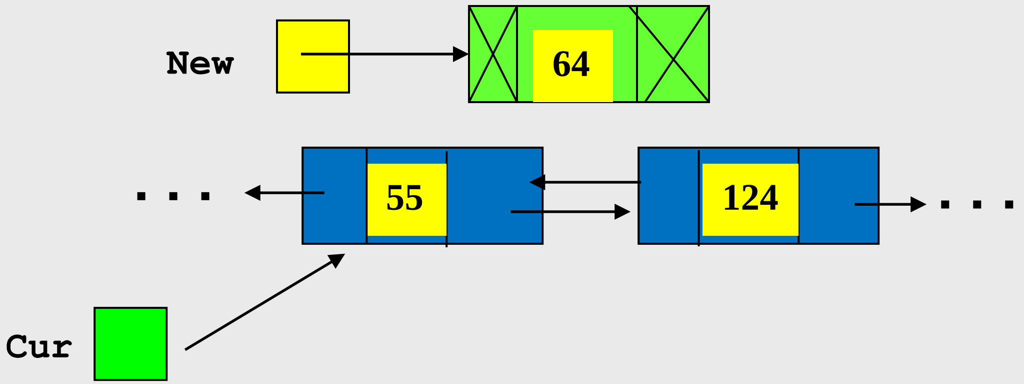
```
New = (struct node *)     /*create node*/
          malloc(sizeof(struct Node));
New -> data = 39;
New -> right = Head;
New -> left = Head;
Head = New;
```

After:

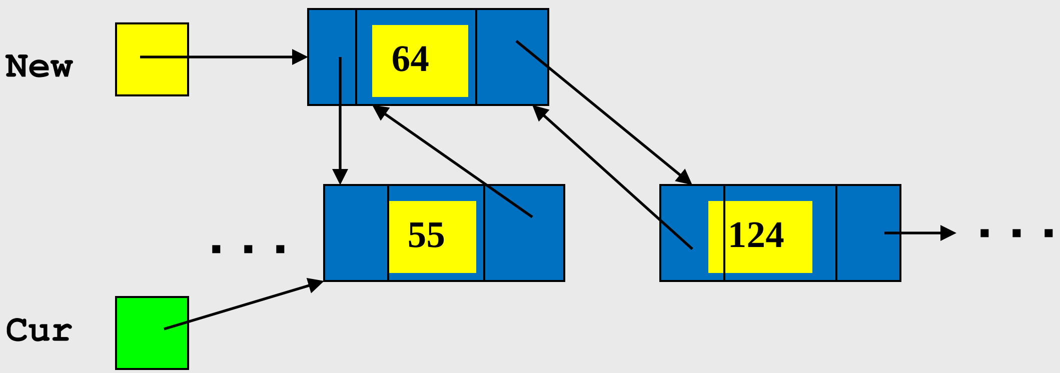


Adding a Node to the Middle of a Doubly-Linked List

Before:



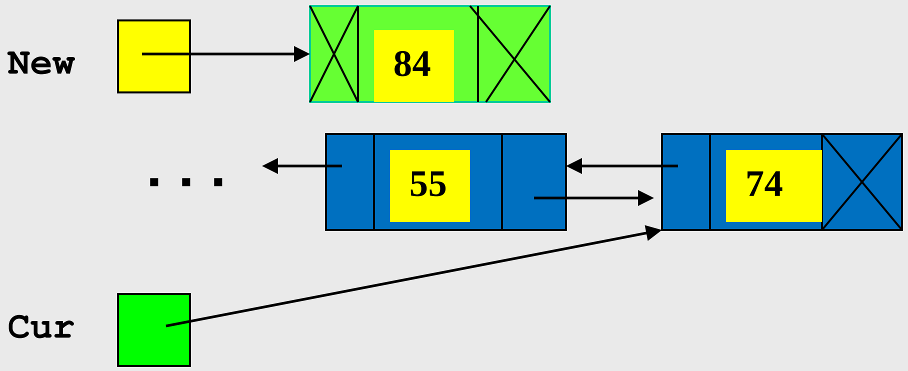
After:



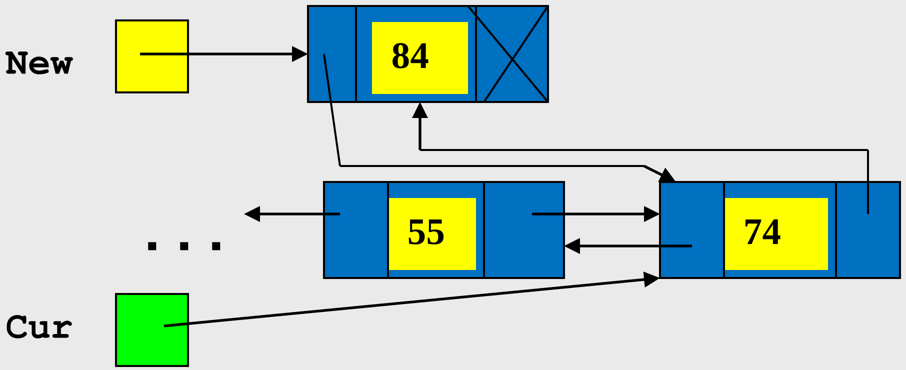

```
New = (struct node *)
malloc(sizeof(struct Node));
New -> data = 64;
New -> left = Cur;
New -> right = Cur -> right;
Cur -> right -> left = New;
Cur -> right = New;
```

Adding a Node to the End of a Doubly-Linked List

Before:



After:




```
New = (struct Node *)
malloc(sizeof(struct Node));
New -> data = 84;
New -> left = Cur;
New -> right = Cur -> right;
Cur -> right  = New;
```

Inserting a Node Into a Doubly-Linked List

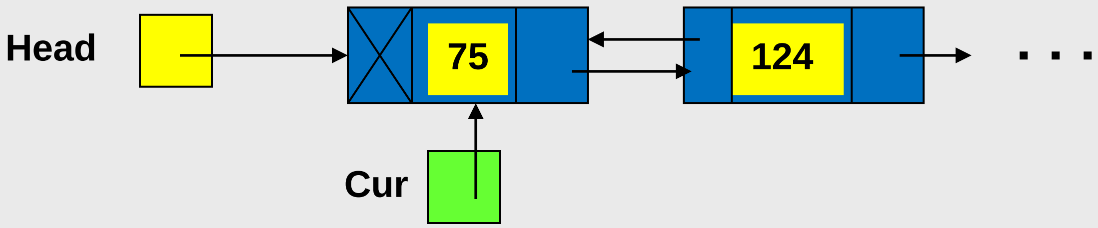
```
//insert a node into a linked list
   struct node *New;
   New = (struct node *)malloc(sizeof(struct node));
   New -> data = item;
   if (Cur == NULL){     //add before first logical node or to an empty list
      New -> left = Head;
       New -> right = Head;
      Head = New;
   }
   else {
         if (Cur -> right == NULL) {           //add at the end
          New -> left = Cur;
            New -> right = Cur -> right;
            Cur -> right  = New;
            }
         else {                                          //add in the middle
                New -> left = Cur;
        New -> right = Cur -> right;
               Cur -> right -> left = New;
               Cur -> right = New;
        }
      }
```

Deleting a Node from a Doubly-Linked List

- Deleting a node requires that we logically remove the node from the list by changing various links and then physically deleting the node from the list (i.e., return it to the heap).
- To logically delete a node:
  - First locate the node itself (Cur).
  - Change the predecessor’s and successor's link fields to point each other.
  - Recycle the node using the free( ) function.

Deleting the First Node from a Doubly-Linked List

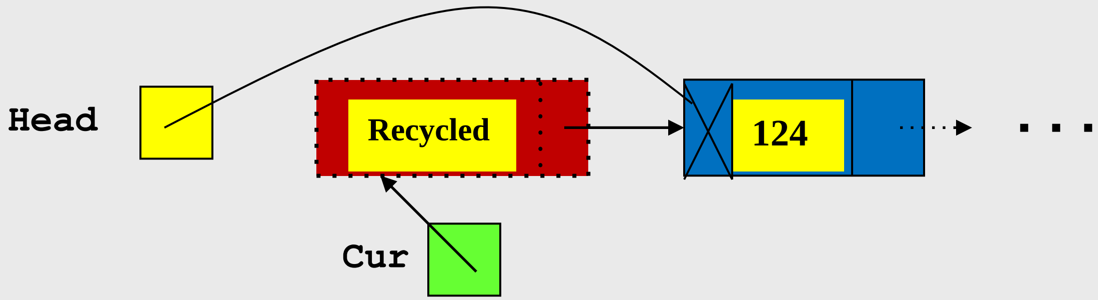
Before:



Code:

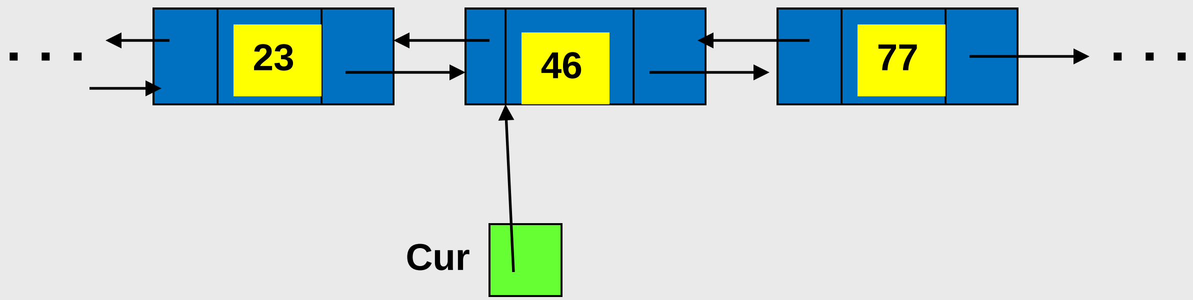
```
Cur=Head
Head = Cur -> right;
Cur ->right -> left = NULL;
free(Cur);
```

After:

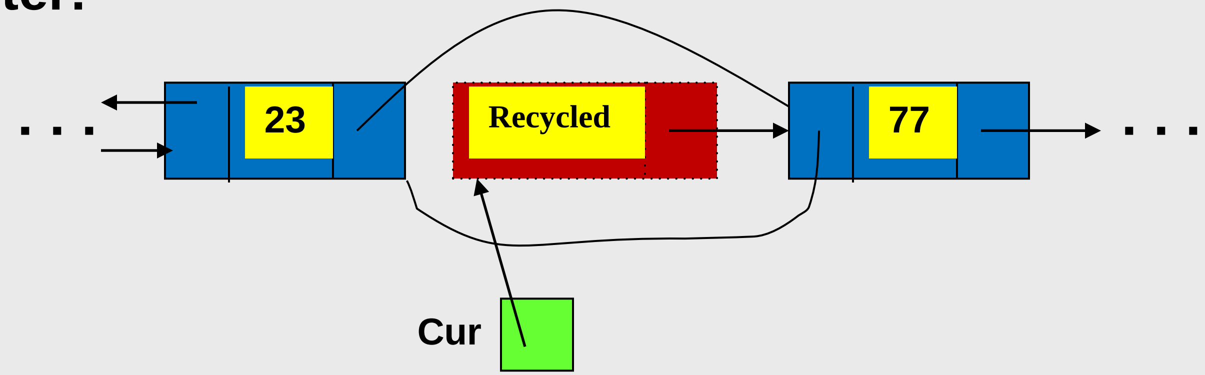


Deleting a Node from a Linked List – General Case

Before:



After:



Deleting a Node From a Doubly-Linked List

```
//delete a node from a linked list
if (Cur -> left == NULL){
        //deletion is on the first node of the list
        Head = Cur -> right;
        Cur -> right -> left = NULL;
{
else {
        //deleting a node other than the first node of the list
        Cur -> left -> right = Cur -> right;
       Cur -> right -> left = Cur -> left;
}
free(Cur).
```

Searching a Doubly-Linked List

- Notice that both the insert and delete operations on a linked list must search the list for either the proper insertion point or to locate the node corresponding to the logical data value that is to be deleted.

```
//search the nodes in a linked list
Cur = Head;
//search until the target value is found or the end of the list is reached
while (Cur != NULL &&  Cur -> data != target) {
      Cur =  Cur -> right;
}
//determine if the target is found or ran off the end of the list
if (Cur != NULL)
   found = 1;
else
   found = 0;
```

Traversing a Doubly-Linked List

- List traversal requires that all of the data in the list be processed. Thus, each node must be visited and the data value examined.

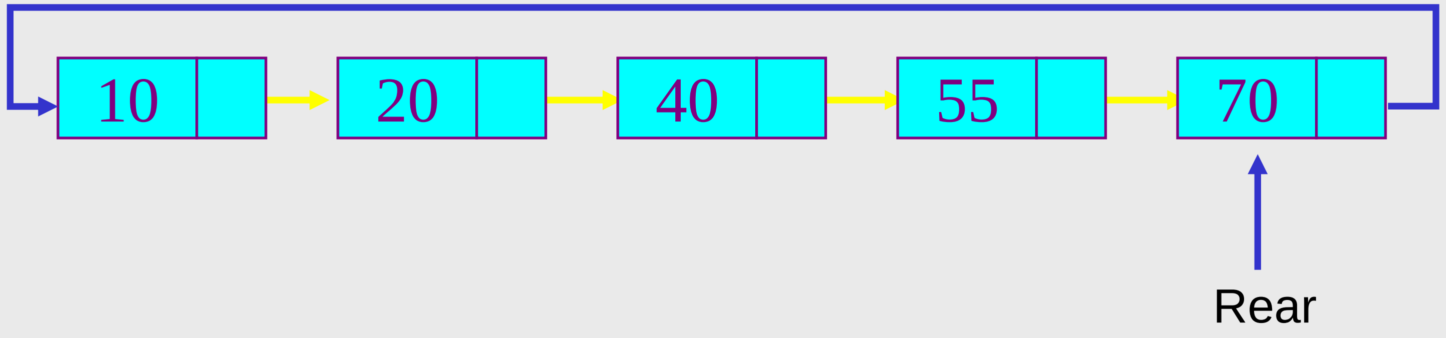
```
//traverse a linked list
struct node *Walker;
Walker = Head;
printf("List contains:\n");
while (Walker != NULL){
    printf("%d ", Walker -> data);
    Walker = Walker -> right;
}
```

DLLs compared to SLLs

- Advantages:
  - Can be traversed in either direction (may be essential for some programs)
  - Some operations, such as deletion and inserting before a node, become easier
- Disadvantages:
  - Requires more space
  - List manipulations are slower (because more links must be changed)

Circular Linked Lists

- A Circular Linked List is a special type of Linked List
- It supports traversing from the end of the list to the beginning by making the last node point back to the head of the list
- A Rear pointer is often used instead of a Head pointer

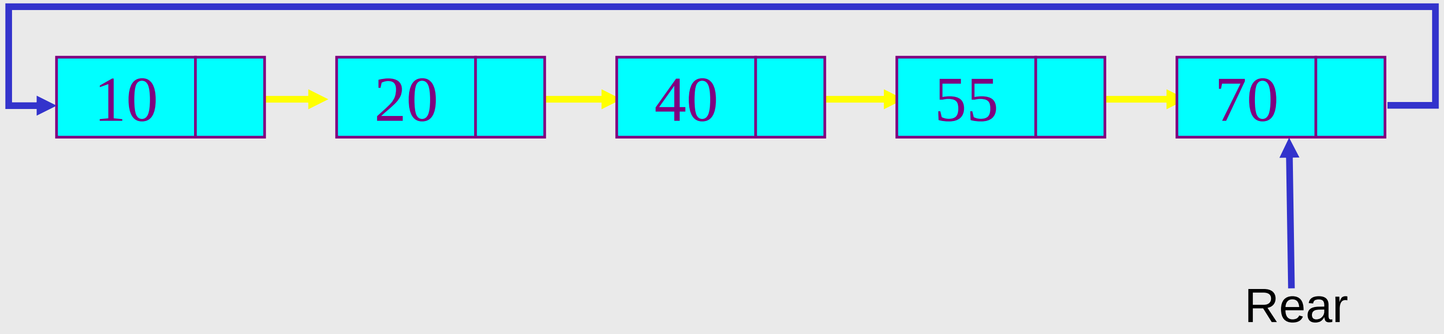


Motivation

- Circular linked lists are usually sorted
- Circular linked lists are useful for playing video and sound files in “looping” mode
- They are also a stepping stone to implementing graphs

Traverse the list

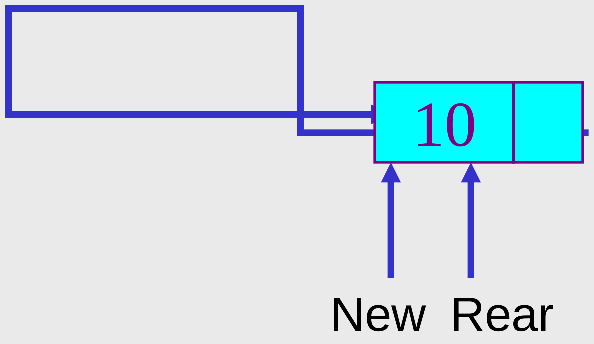
```
void print(Rear){
    node *Cur;
    if(Rear != NULL){
        Cur = Rear->next;
        do{
            Printf("%d",Cur->data);
            Cur = Cur->next;
        }while(Cur != Rear->next);
    }
}
```



Insert Node

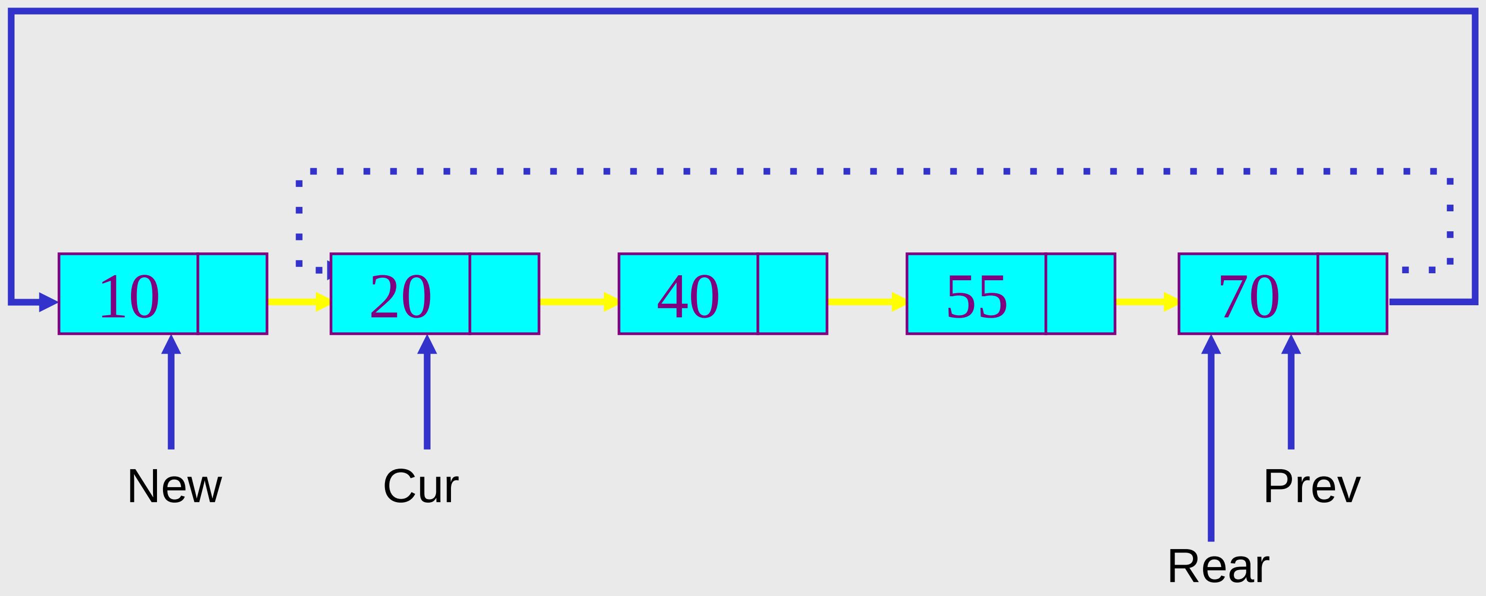
- Insert into an empty list

```
node *New = (node*)malloc(sizeof(node));
New->data = 10;
Rear = New;
Rear->next = Rear;
```



Insert to head of a Circular Linked List

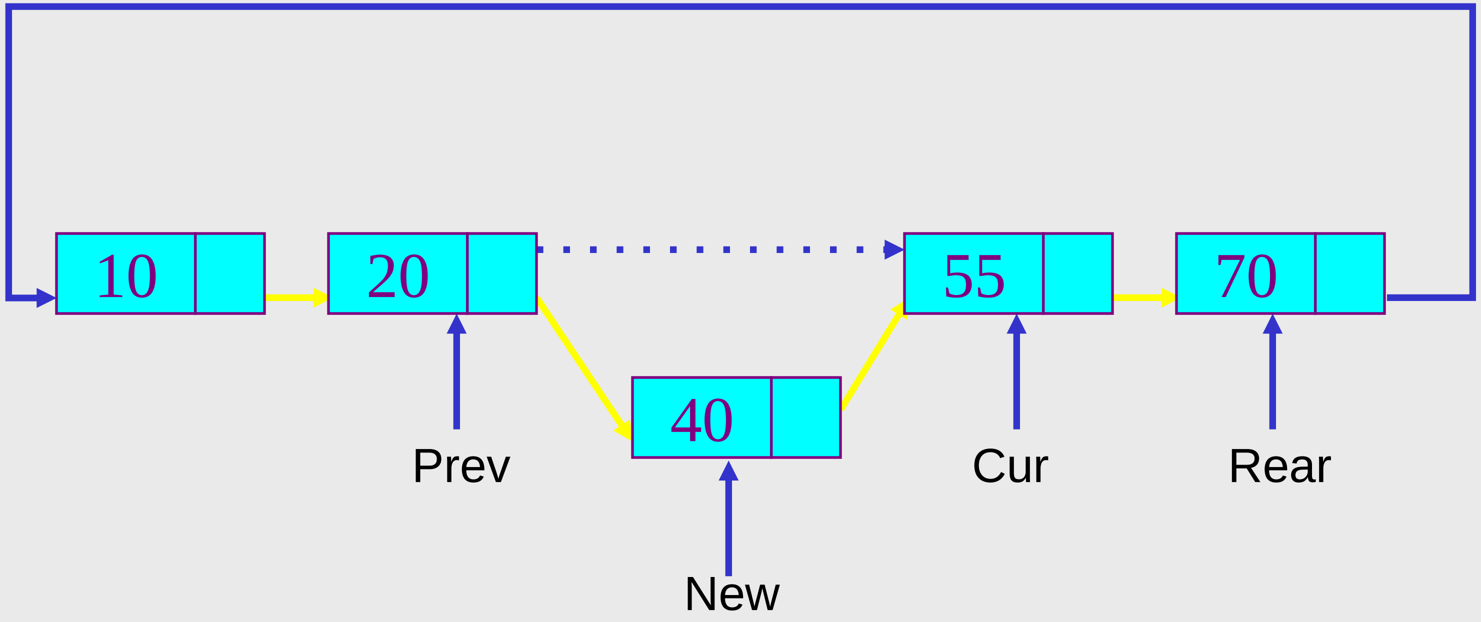
```
New->next = Cur;  // same as: New->next = Rear->next;
Prev->next = New; // same as: Rear->next = New;
```



Insert to middle of a Circular Linked List between `Pre` and `Cur`

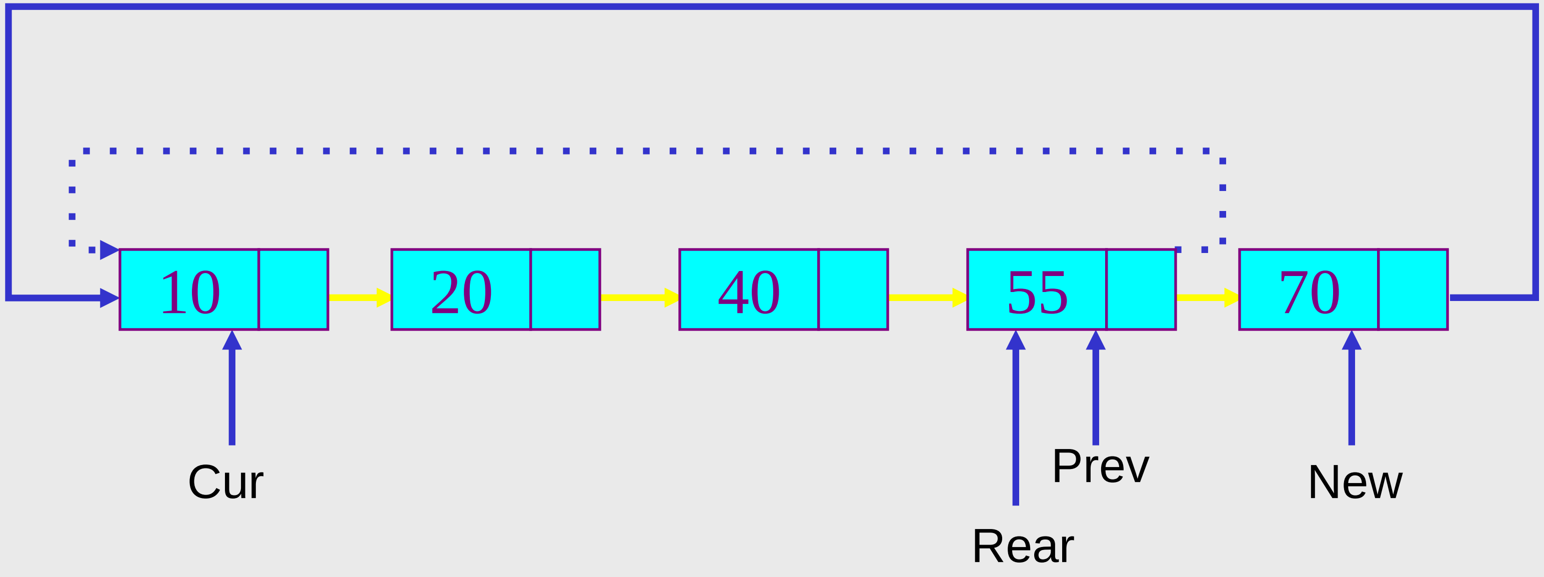
New->next = Cur;

Prev->next = New;



Insert to end of a Circular Linked List

```
New->next = Cur;        // same as: New->next = Rear->next;
Prev->next = New;       // same as: Rear->next = New;
Rear = New;
```



Other operations on linked lists

- Most “algorithms” on linked lists—such as insertion, deletion, and searching—are pretty obvious; you just need to be careful
- Sorting a linked list is just messy, since you can’t directly access the n^{th} element—you have to count your way through a lot of other elements

Slides and figures have been collected from various publicly available Internet sources for preparing the lecture slides of IT2001 course. I acknowledge and thank all the original authors for their contribution to prepare the content.