Doubly Linked List

Header Linked List

Motivation

- Doubly linked lists are useful for playing video and sound files with “rewind” and instant “replay”
- They are also useful for other linked data where “require” a “fast forward” of the data as needed

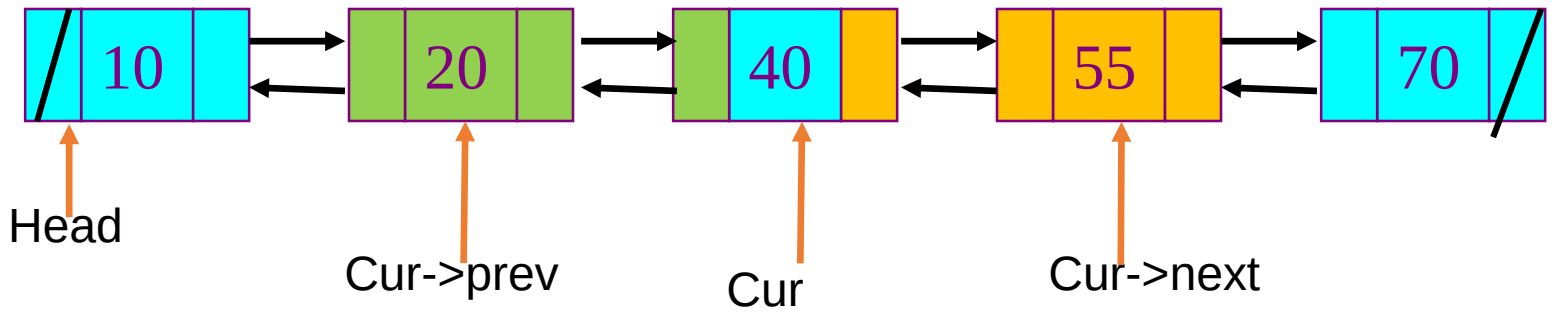
- **list using an array**:
  - Knowledge of list size
  - Access is easy (get the ith element)
  - Insertion/Deletion is harder
- **list using ‘singly’ linked lists**:
  - Insertion/Deletion is easy
  - Access is harder

  - **But, can not ‘go back’!**

Doubly Linked Lists

In a Doubly Linked-List each item points to both its predecessor and successor

- prev points to the predecessor
- next points to the successor



Doubly Linked List Definition

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

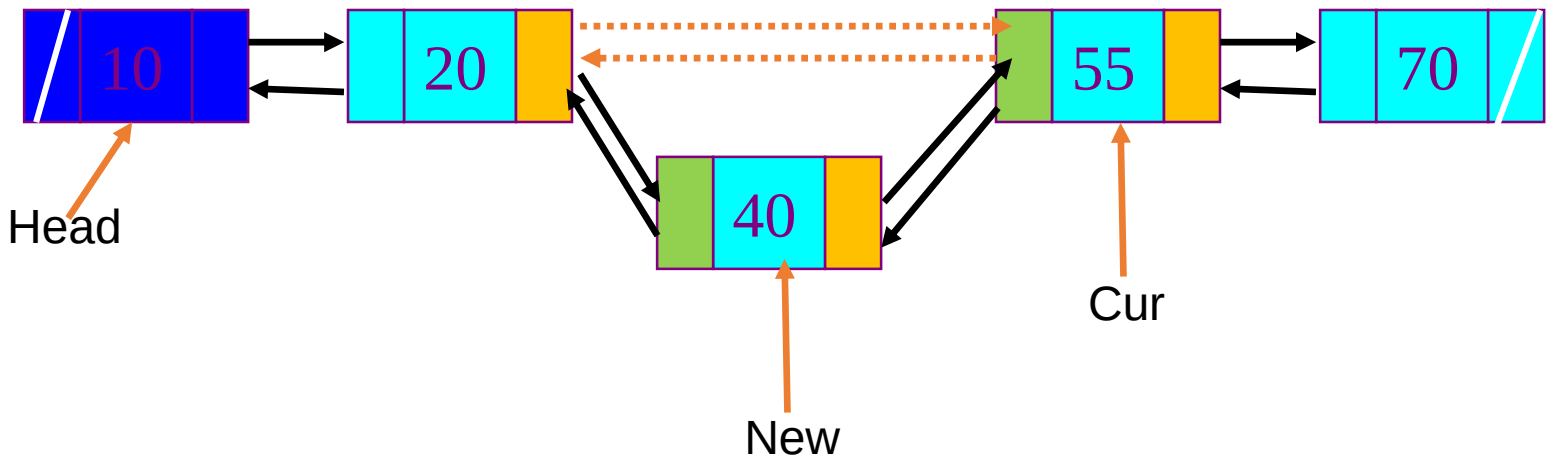
Doubly Linked List Operations

- `insertNode(int item)`
  `//add new node to ordered doubly linked list`
- `deleteNode(int item)`
  `//remove a node from doubly linked list`
- `SearchNode(int item)`
- `Print(int item)`

Inserting a Node

- Insert a node `New` before `Cur` (not at front or rear)

```
New->next = Cur;
New->prev = Cur->prev;
Cur->prev = New;
(New->prev)->next = New;
```



Many special cases to consider.

```
void insertNode( int item) {
        NodePtr *cur,*New;
        cur = searchNode(item);

        if (head==NULL) { …
        }
        else if (cur->prev == NULL) { …
                }
                else if (cur->next==NULL) { …
                }
                        else {
                                Insertion Code …
                }
}
```

Empty case

At-the-beginning case

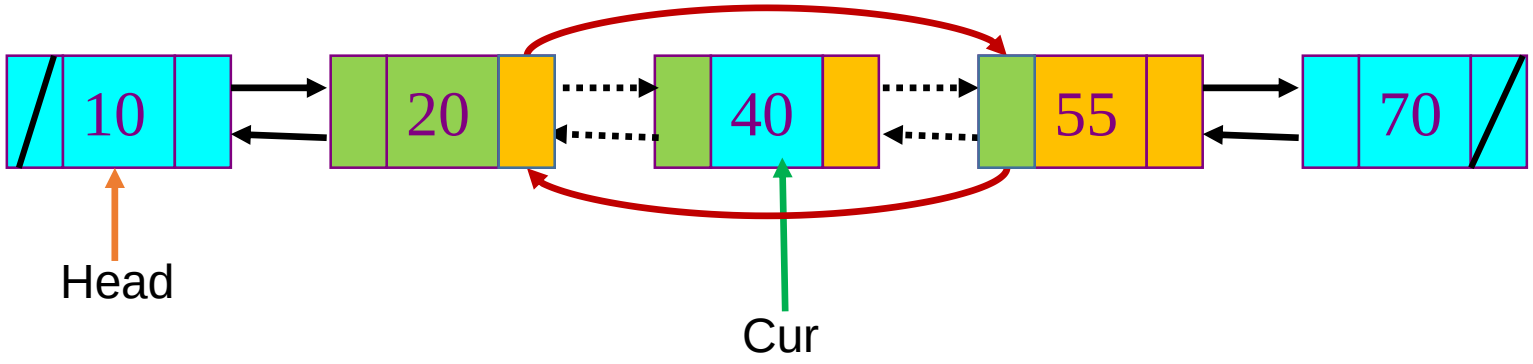
At-the-end case

General case

Deleting a Node

- Delete a node `Cur` (not at front or rear)

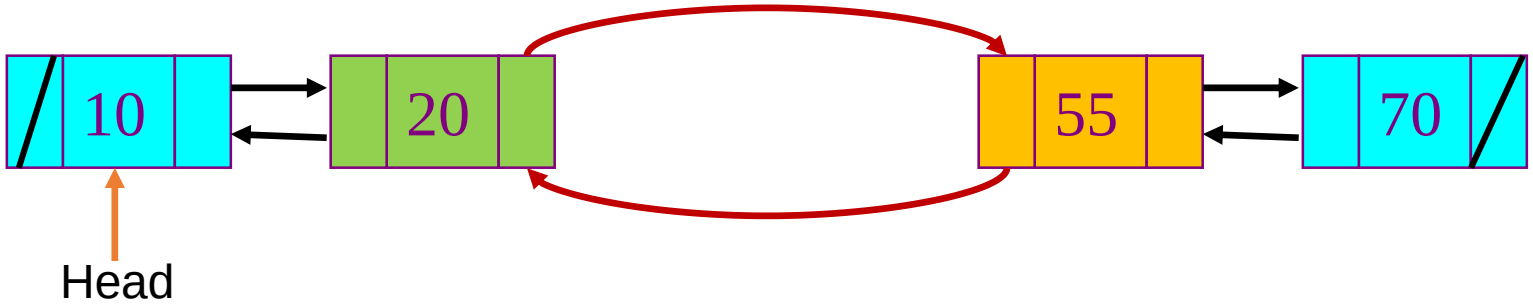
```
(Cur->prev)->next = Cur->next;
(Cur->next)->prev = Cur->prev;
free(Cur);
```



Deleting a Node

- Delete a node `Cur` (not at front or rear)

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



```
void deleteNode(int item) {
        sruct Node *cur;
        cur = searchNode(item);

        if (head==NULL) { …
        }
        else if (cur->prev == NULL) { …
                }
                else if (cur->next==NULL) { …
                }
                        else {

                                (cur->prev)->next = cur->next;
                                (cur->next)->prev = cur->prev;
                                free(cur);
                        }
}
```

Empty case

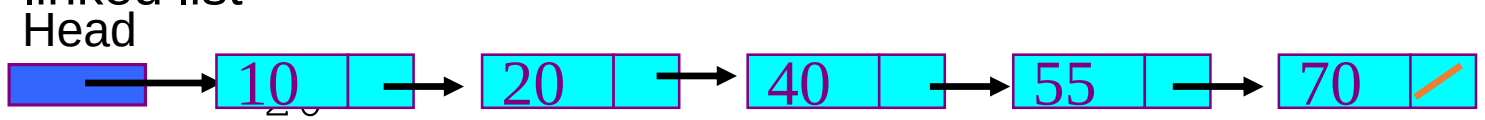
At-the-beginning case

At-the-end case

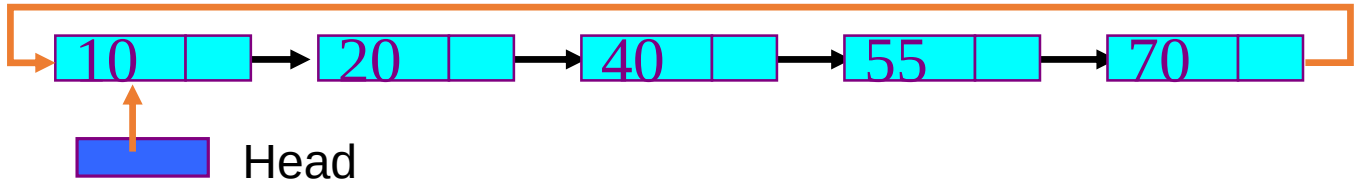
General case

A systematic way is to start from all these cases, then try to simply the codes, …

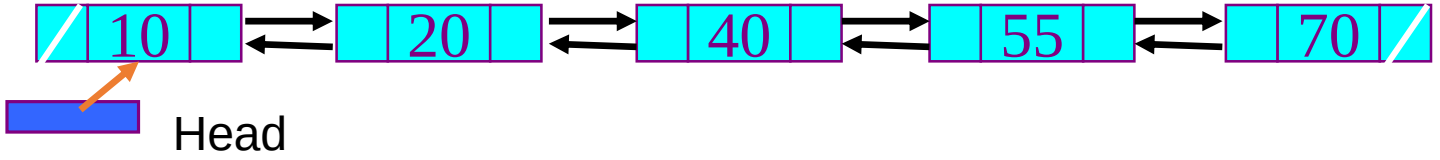
singly linked list



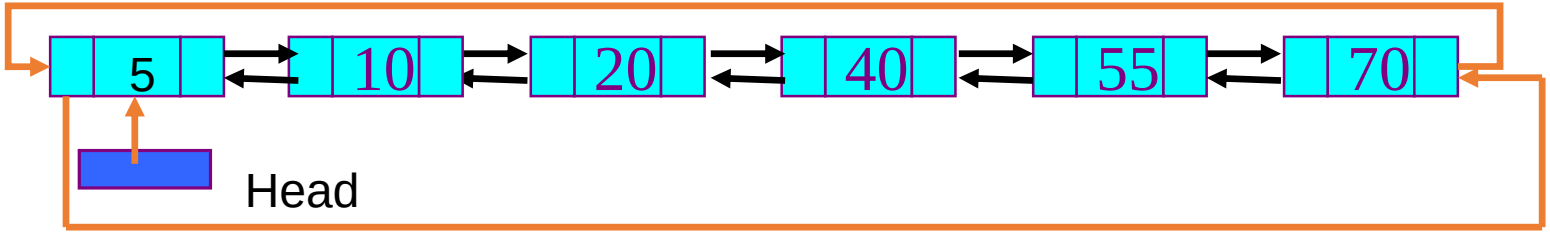
(singly) circular linked list



(regular) doubly linked list



Circular doubly linked list



Header Linked List

- A *header node* is a special node that is found at the *beginning* of the list.
- A list that contains this type of node, is called the header-linked list.
- This type of list is useful when information other than that found in each node is needed.
- *For example*, suppose there is an application in which **the number of items in a list is often calculated.** Usually, a list is always traversed to find the length of the list. However, if the current length is maintained in an additional header node that information can be easily obtained.

Many different linked lists …

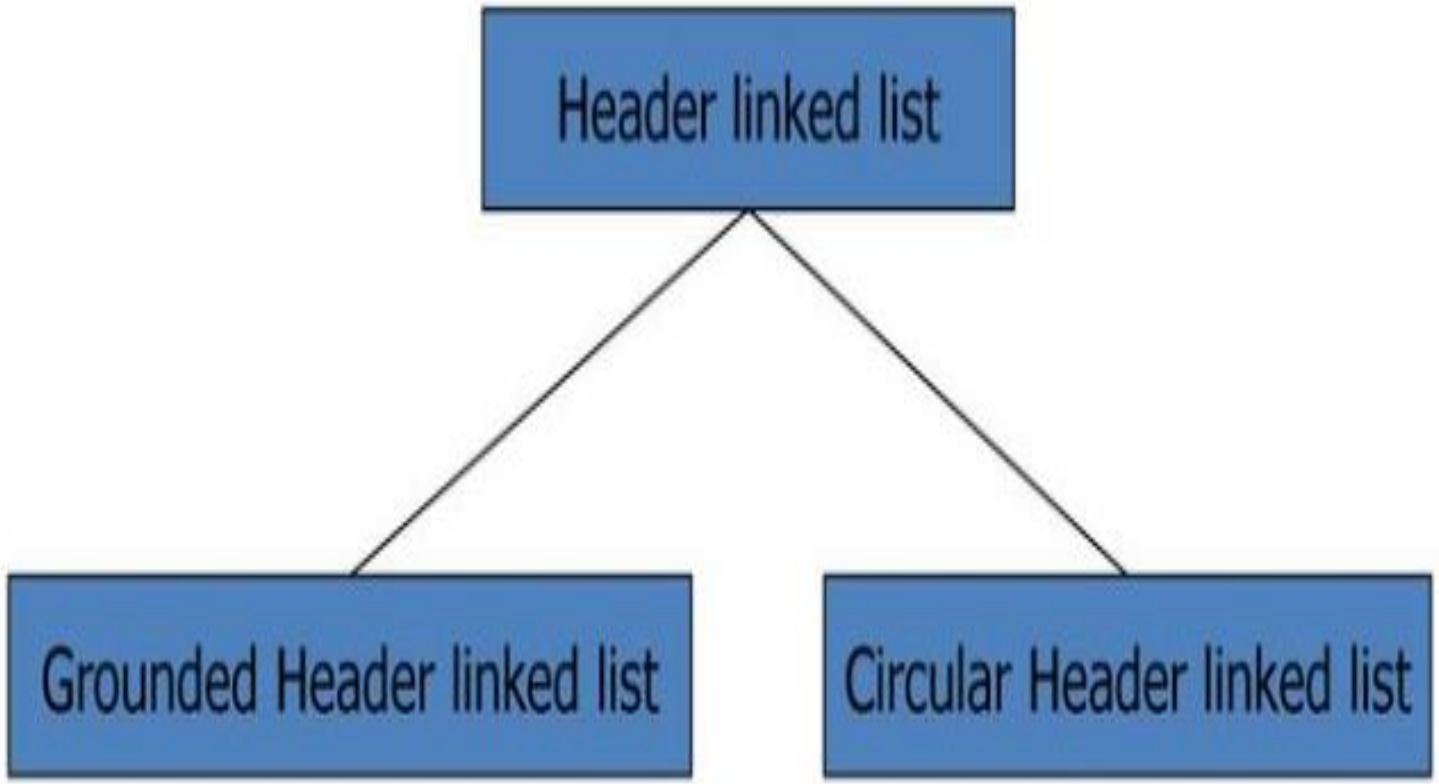
- singly linked lists
  - circular
  - Without ‘Header Node’
  - With Header Node

- doubly linked lists
  - circular
  - Without ‘Header Node’
  - With Header Node

Using ‘Header Node’ is a matter of personal preference!

+ simplify codes (not that much ☺

Types of Header Linked List



Idea of ‘Header Node’

‘Header Node’ is also called a ‘sentinel’, it allows the simplification of special cases

Head/ Start Pointer Instead of pointing to NULL, points to the ‘Header Node’!!!

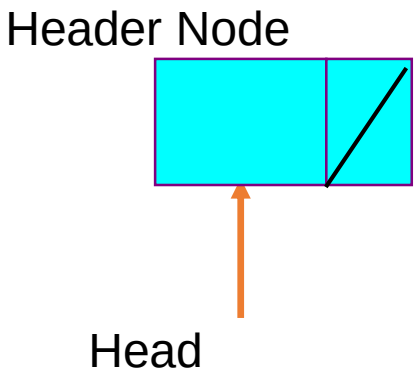
1.Grounded Header Linked List

It is a list whose *last node* contains the *NULL* pointer. In the header linked list the **head** pointer always points to the **header node**.

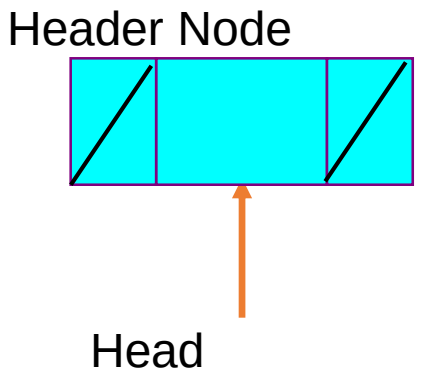
head-> next = NULL indicates that the grounded header linked list is *empty*.

The operations that are possible on this type of linked list are *Insertion, Deletion, and Traversing*.

Empty Grounded singly list:



Empty Grounded doubly list:



Head->next = NULL; compared with head=NULL;

Grounded Header Linked List

- Singly Grounded Header Linked List

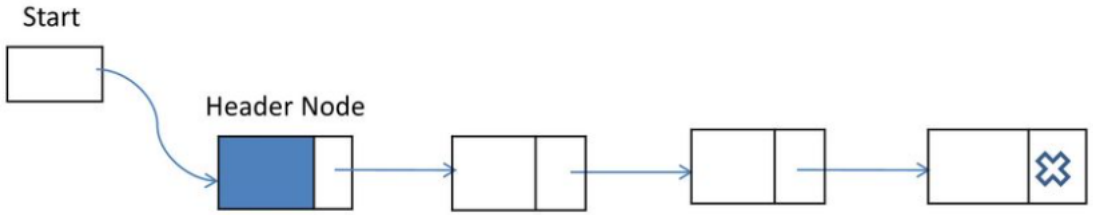
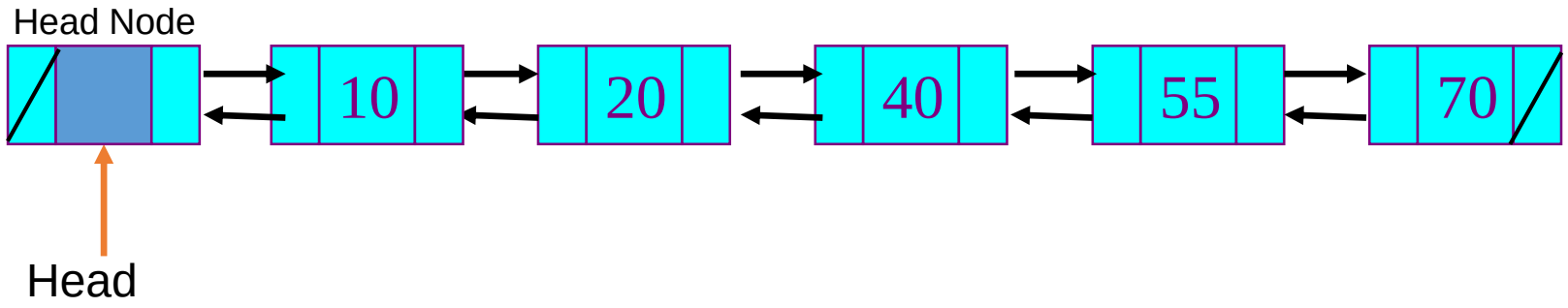


Figure: Grounded Header Link List

- Doubly Grounded Header Linked List



2. Circular Header Linked List

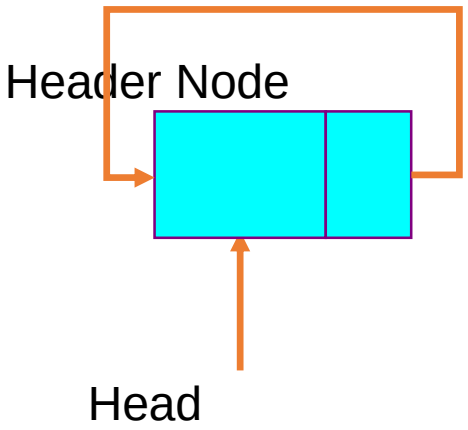
A list in which *last node* points back to the ***header node*** is called circular linked list.

The chains do not indicate first or last nodes.

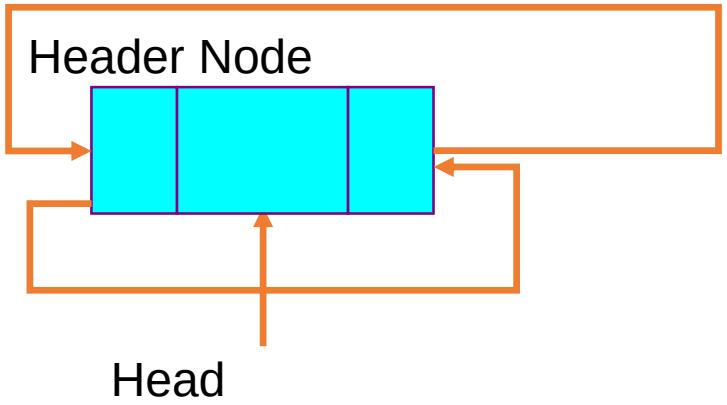
In this case, external pointers provide a frame of reference because last node of a circular linked list does **not contain** the **NULL** pointer.

The possible operations on this type of linked list are *Insertion, Deletion and Traversing*.

Empty Circular singly list:



Empty Circular doubly list:



Head->next = head; compared with head=NULL;

Idea of ‘Header Node’

- Singly Circular Header Linked List

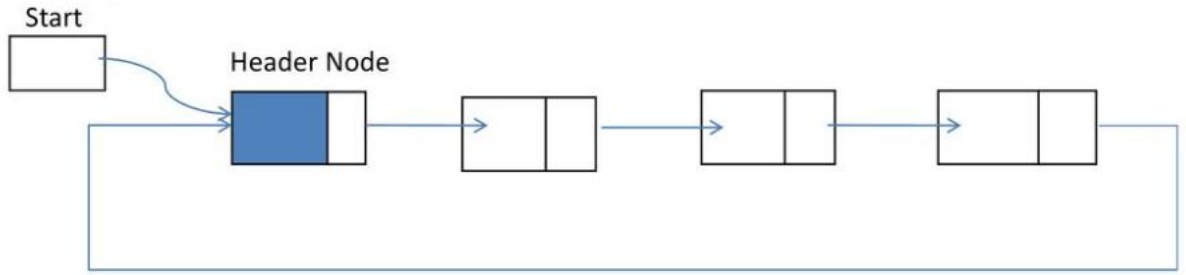
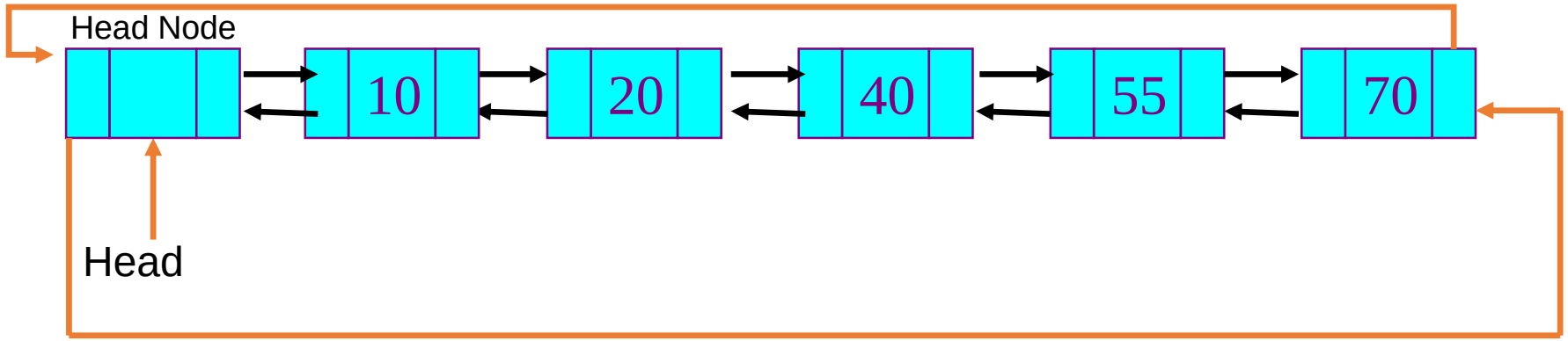


Figure: Circular Linked List with header node

- Doubly Circular Header Linked List



Print the grounded singly/doubly header linked list:

```
void print(){
   struct Node *cur=head->next;
   while(cur != NULL){
     printf("%d ->",cur->data);
     cur = cur->next;
    }
}
```

Print the circular singly/doubly header linked list:

```
void print(){
   struct Node *cur=head->next;
   while(cur != head){
      printf("%d ->",cur->data);
      cur = cur->next;
         }
}
```

Searching a node in Grounded singly/doubly Header Linked List

(returning NULL if not found the element):

```
struct Node * searchNode(int item){
  struct Node *cur = head->next;
while((cur != NULL) && (item != cur->data))
      { cur=cur->next;}

  return cur;
}
```

Return the address

Searching a node in Circular singly/doubly Header Linked List

(returning NULL if not found the element):

```
struct Node * searchNode(int item){
      struct Node *cur = head->next;

while((cur != head) && (item != cur->data))
     { cur=cur->next;}

  if (cur == head) {  cur = NULL; }
     // we didn't find
return cur;
  }
```

Return Address

```
Struct Node *Head= (struct node *)malloc(sizeof(struct node));
Head->data=0;
Head->next = NULL;
void main(){
        struct Node *temp;

        createHead();
        insertNode(3);
        insertNode(5);
        insertNode(2);
        print();
        insertNode(7);
        insertNode(1);
        insertNode(8);
        print();
        deleteNode(7);
        deleteNode(0);
        print();
        temp = searchNode(5);
        if(temp !=NULL)
            printf(" Data is contained in the list“);
        else
            printf(" Data is NOT contained in the list“);
 }
```

Result is

2 3 5

1 2 3 5 7 8

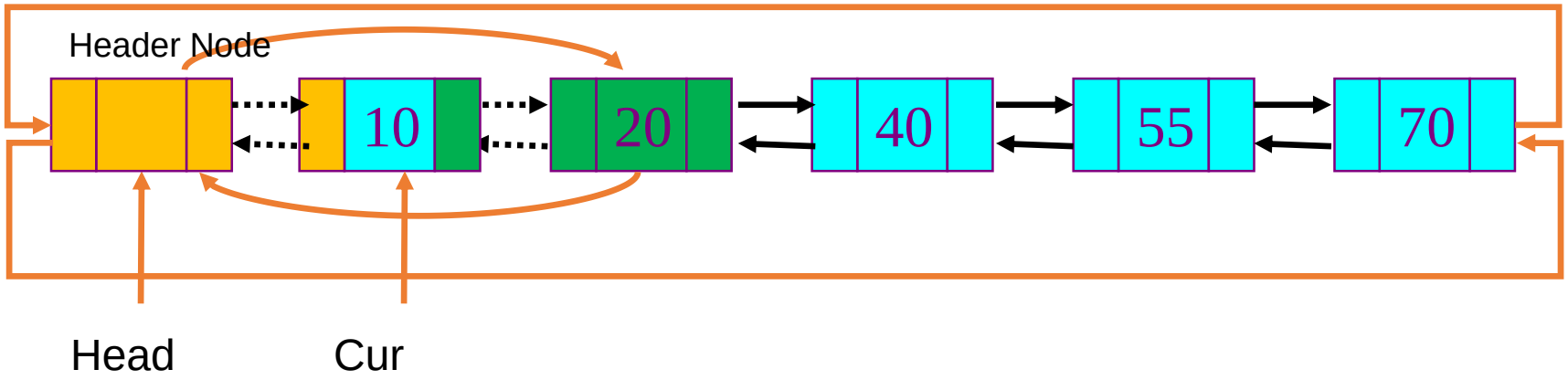
1 2 3 5 8

Data is contained in the list

Deleting a Node in Circular Header Doubly Linked List

- Delete a node `Cur` at front

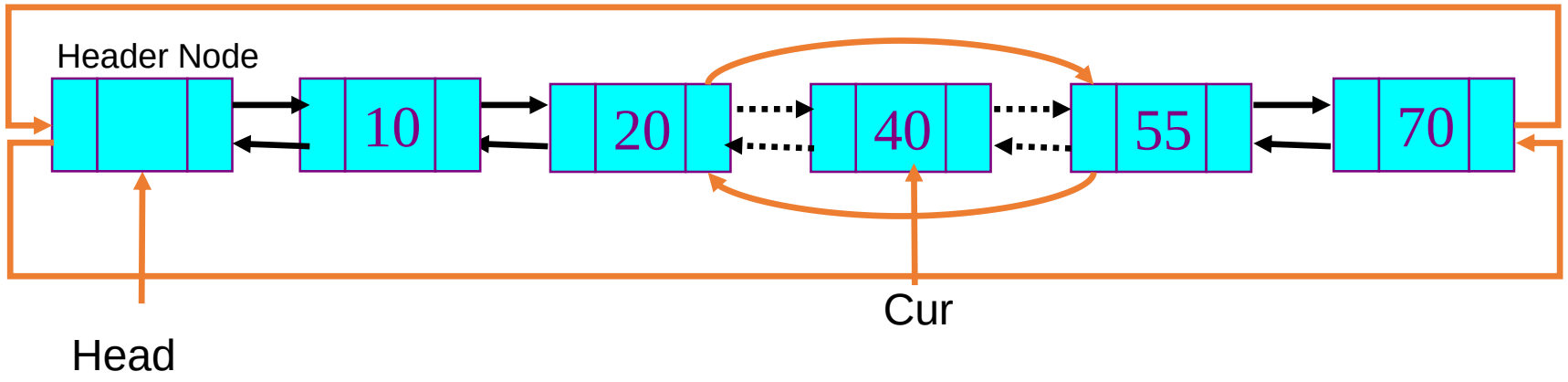
```
(Cur->prev)->next = Cur->next;
(Cur->next)->prev = Cur->prev;
free(Cur);
```



Deleting a Node in Circular Header Doubly Linked List

- Delete a node `Cur` in the middle

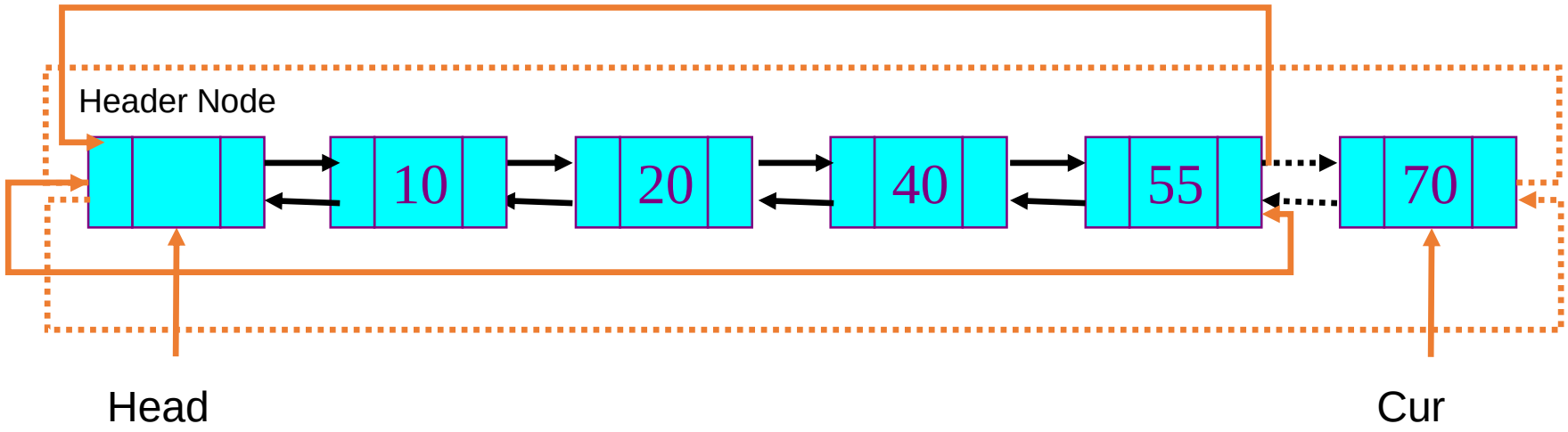
```
(Cur->prev)->next = Cur->next;
(Cur->next)->prev = Cur->prev;
free(Cur);  // same as delete front!
```



Deleting a Node in Circular Header Doubly Linked List

- Delete a node `Cur` at rear

```
(Cur->prev)->next = Cur->next;
(Cur->next)->prev = Cur->prev;
free(Cur);  // same as delete front and middle!
```



Function to Delete a Node in Circular Header Doubly Linked List

```
void deleteNode(int item){
        struct Node *cur;
        cur = searchNode(item);
        if(cur != NULL){
                cur->prev->next = cur->next;
                cur->next->prev = cur->prev;


                free(cur);
        }
}
```

If we found the element, it does not mean any emptyness!

Inserting a Node in Circular Header Doubly Linked List

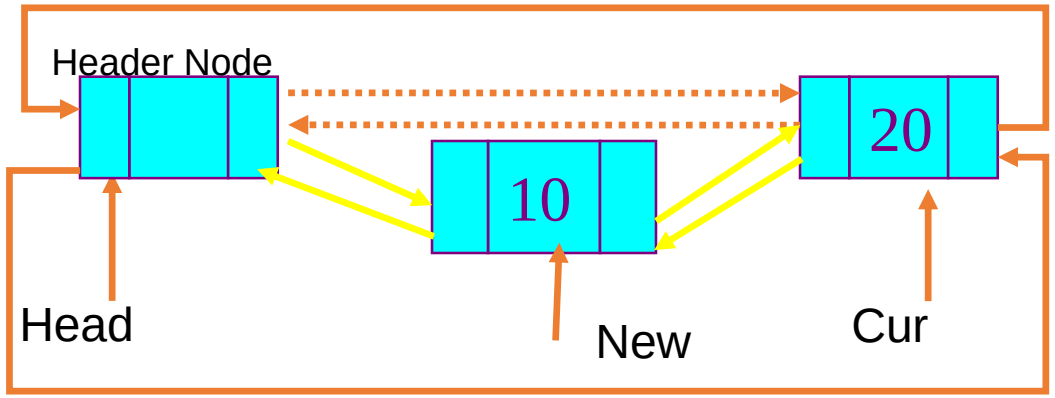
- Insert a Node `New` after header node and before `Cur`

```
New->next = Cur;

New->prev = Cur->prev;

Cur->prev = New;

(New->prev)->next = New;
```



Inserting a Node in Circular Header Doubly Linked List

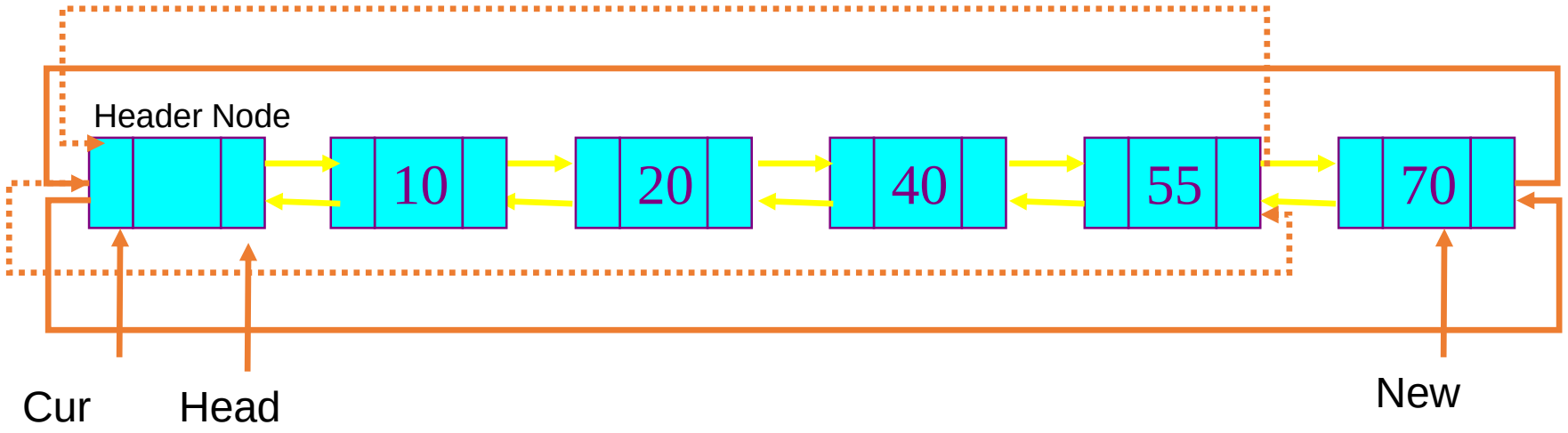
- Insert a Node `New` at Rear (with `Cur` pointing to header node)

```
New->next = Cur;

New->prev = Cur->prev;

Cur->prev = New;

(New->prev)->next = New;
```



Inserting a Node in Circular Header Doubly Linked List

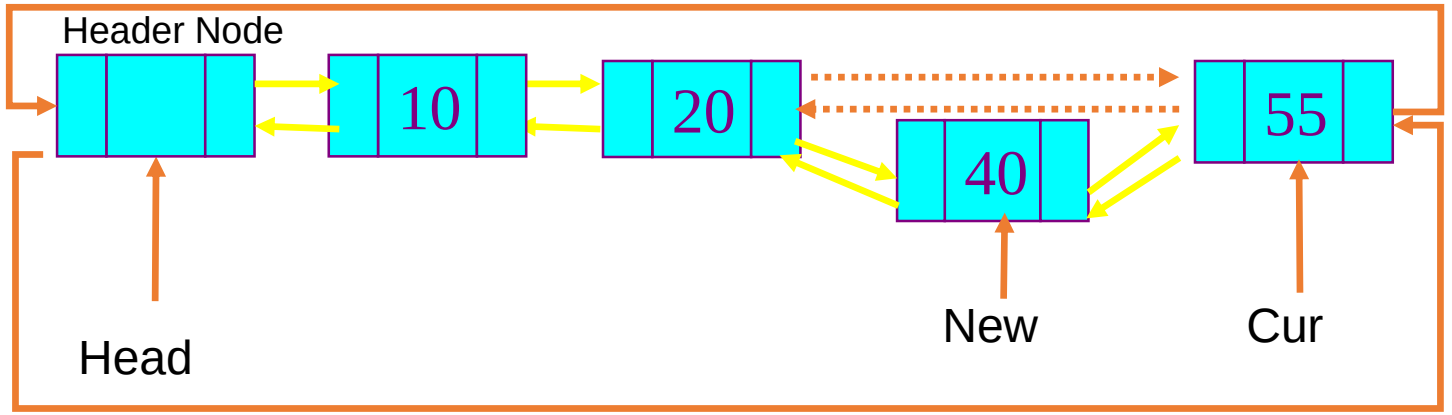
- Insert a Node `New` in the middle and before Cur

```
New->next = Cur;

New->prev = Cur->prev;

Cur->prev = New;

(New->prev)->next = New;
```



Inserting a Node in Circular Header Doubly Linked List

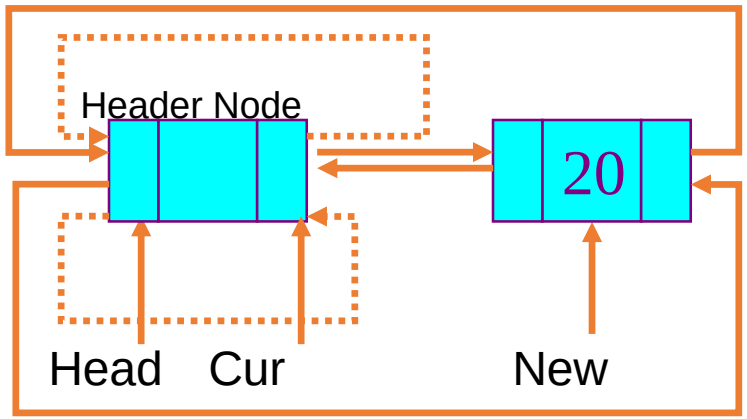
- Insert a Node `New` to Empty List (with `Cur` pointing to header node)

```
New->next = Cur;

New->prev = Cur->prev;

Cur->prev = New;

(New->prev)->next = New;
```

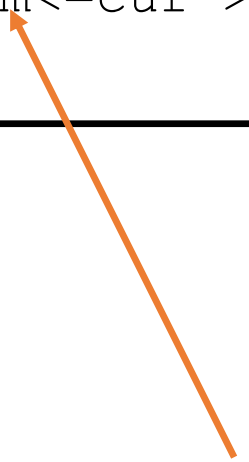


Function to insert a new node in Circular Doubly Header Linked List in sorted order

```
void insertNode(int item){
        struct Node *newp, *cur;
creation    newp = (struct node *)malloc(sizeof(struct node));
        newp->data = item;

location    cur = head->next;
        while ((cur != head)&&(!(item<=cur->data)))
            cur = cur->next;

insertion   newp->next = cur;
        newp->prev = cur->prev;
        cur->prev = newp;
        (newp->prev)->next = newp;
}
```



It is similar to, but different from SearchNode!
(it returns NULL if no element)