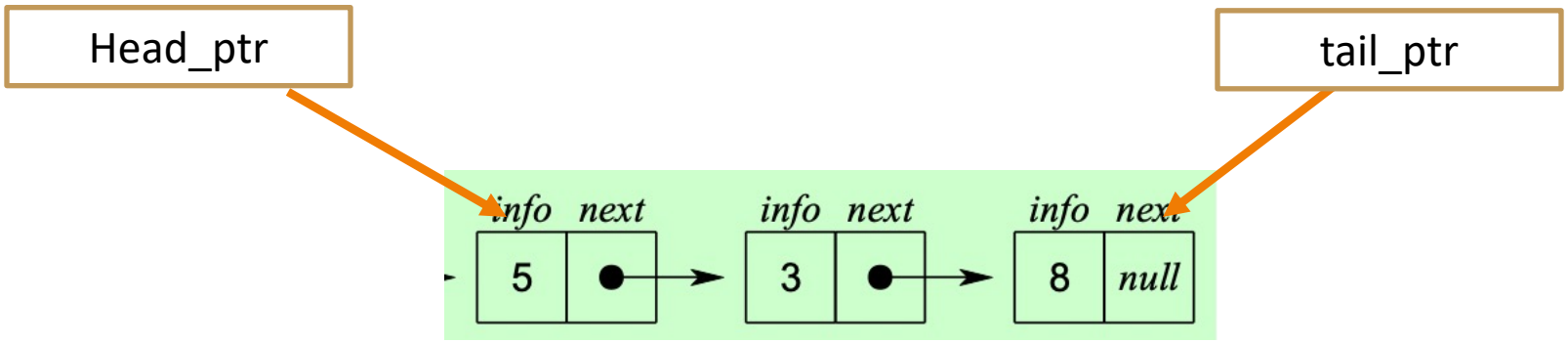
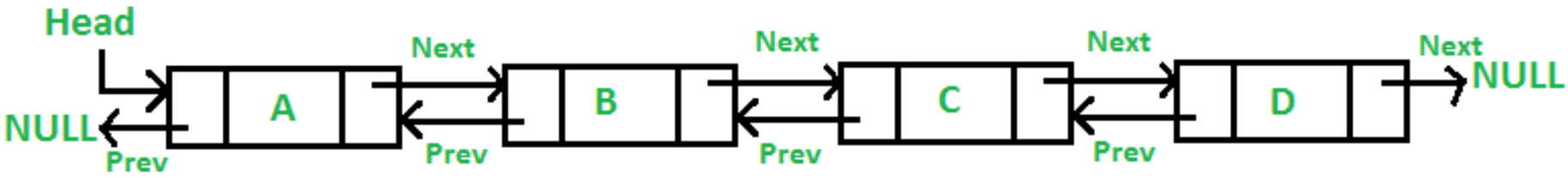
Linked List



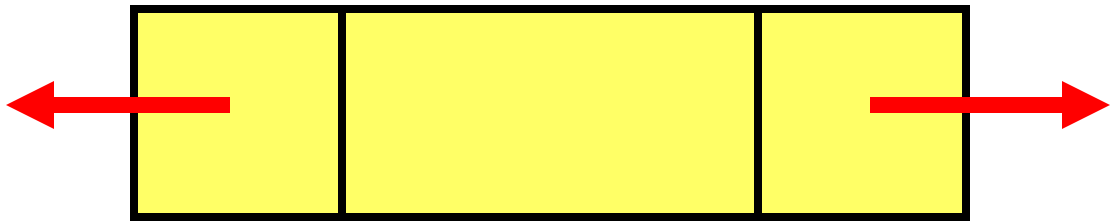

```
Struct Node
{
    typedef double Item;
    Item data;
    Node *link;
};

Node *head_ptr;
Node *tail_ptr;
```

Doubly Linked List



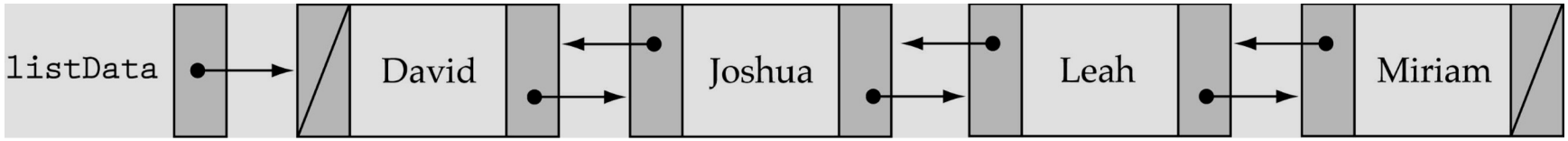
Node data

- info: the user's data
- next, back pointer: the address of the next and previous node in the list



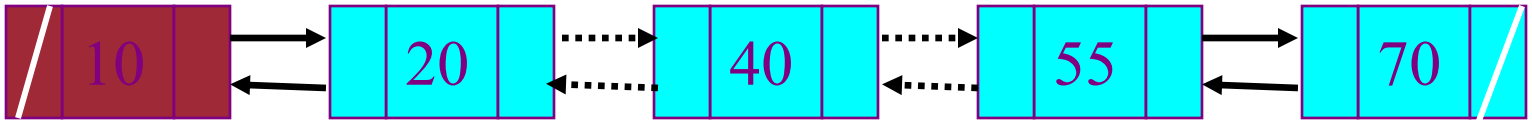
Node data (cont.)

```
struct Node {
    string name;
    Node *prev, *next;
};
Node* NodePtr;
```



Node data (cont.)

```
struct Node {
    int item;
    Node *prev, *next;
};
typedef Node* NodePtr;
```



Doubly Linked List Operations

* `insertNode(NodePtr Head, int item)`
  `//add new node to ordered doubly linked list`

* `deleteNode(NodePtr Head, int item)`
  `//remove a node from doubly linked list`

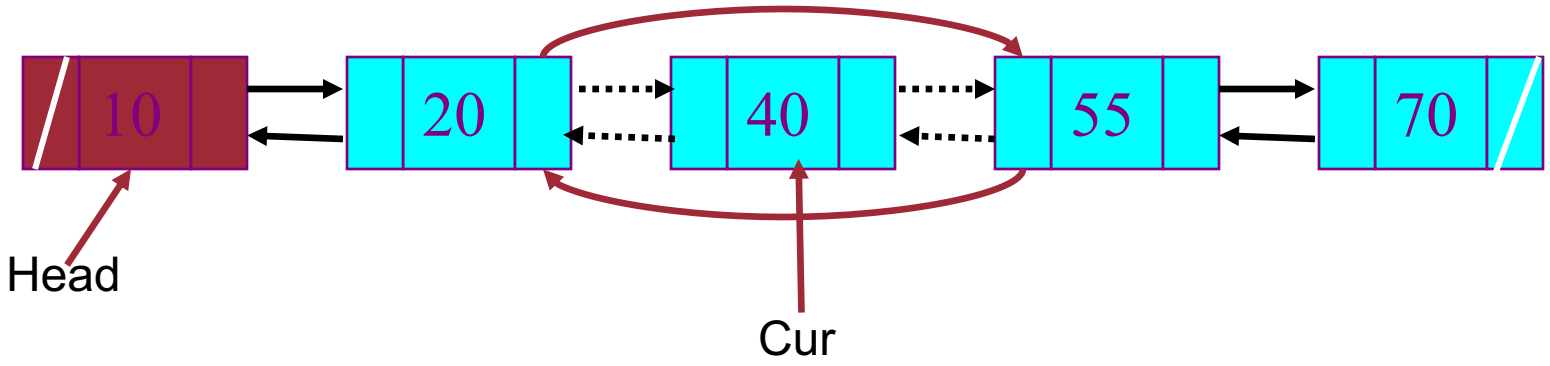
* `searchNode(NodePtr Head, int item)`

* `print(NodePtr Head, int item)`

Deleting a Node

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

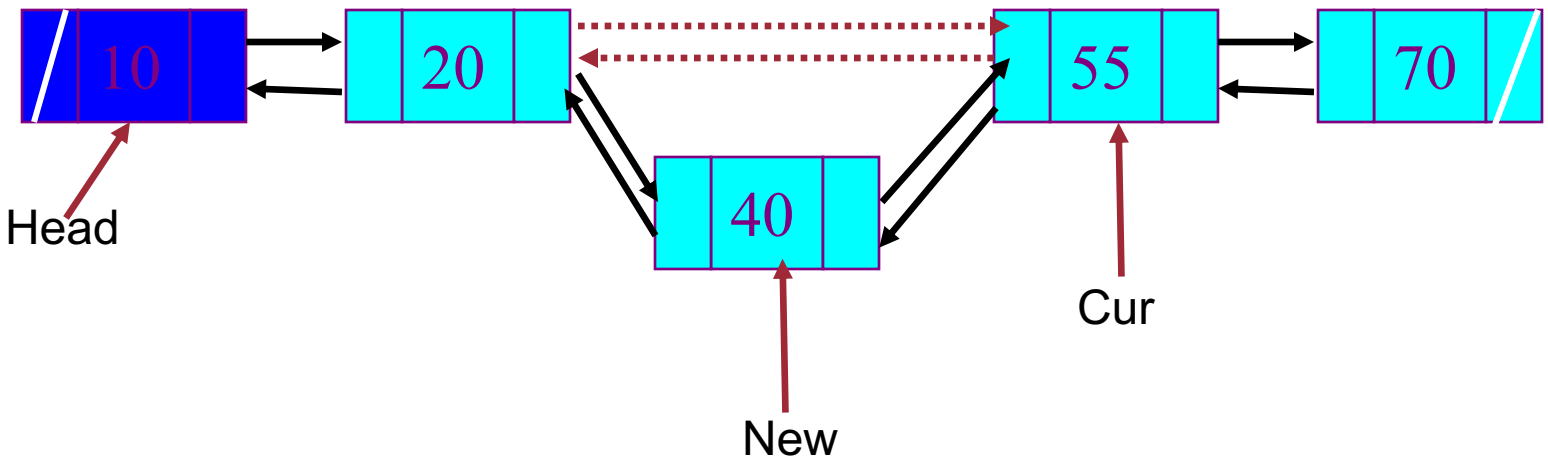
```
(Cur->prev)->next = Cur->next;
(Cur->next)->prev = Cur->prev;
delete Cur;
```



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;
```

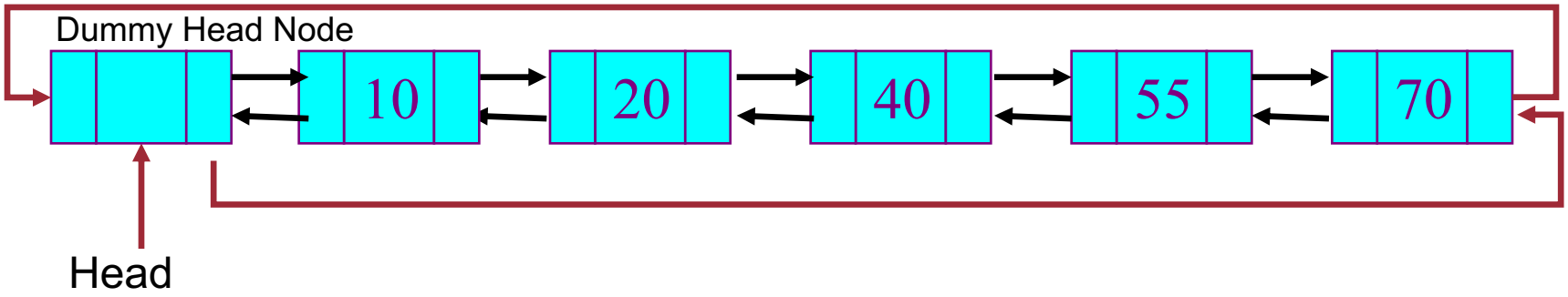


Doubly Linked Lists with Dummy Head Node

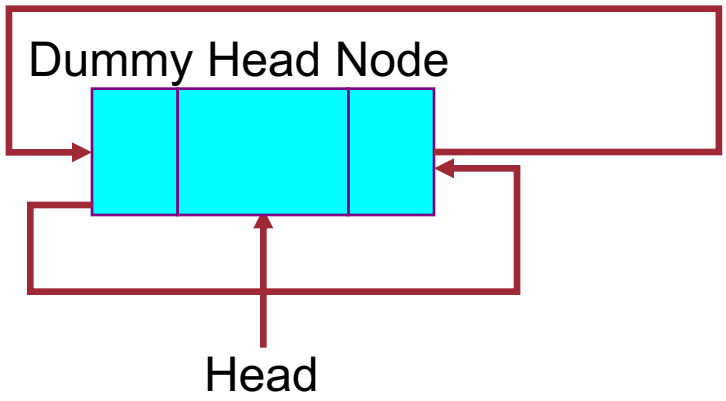
- To simplify insertion and deletion by avoiding special cases of deletion and insertion at front and rear, a dummy head node is added at the head of the list
- The last node also points to the dummy head node as its successor

Doubly Linked Lists with Dummy Head

– Non-Empty List



– Empty List

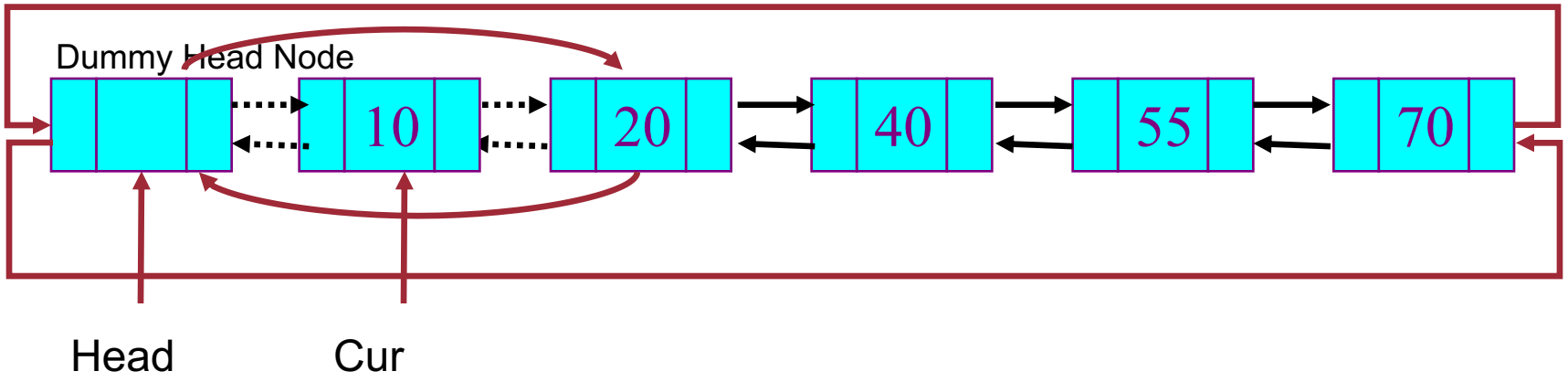


```
void createHead(NodePtr& Head){
  Head = new Node;
  Head->next = Head;
  Head->prev = Head;
}
```

Deleting a Node

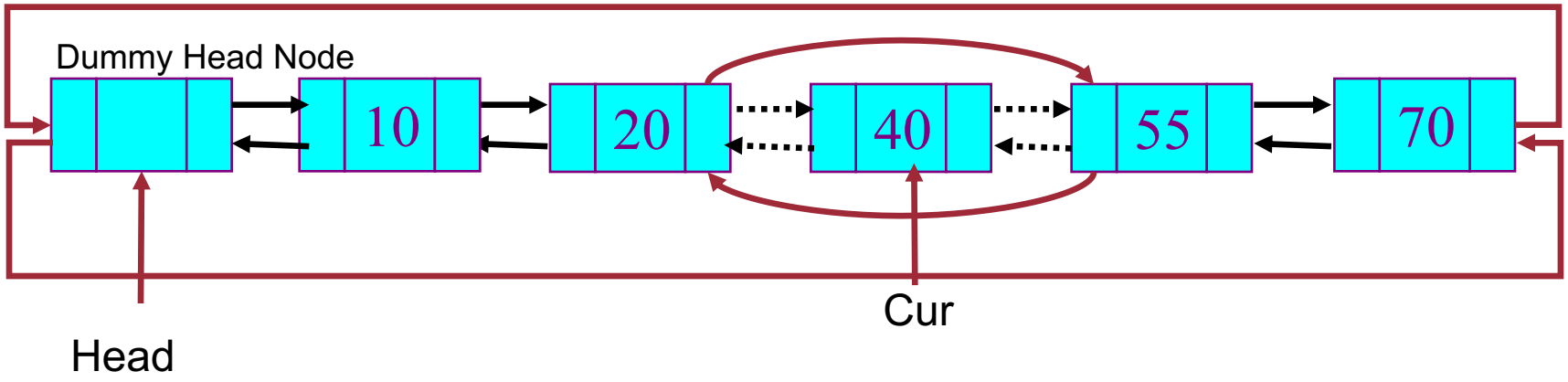
- Delete a node `Cur` at front

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



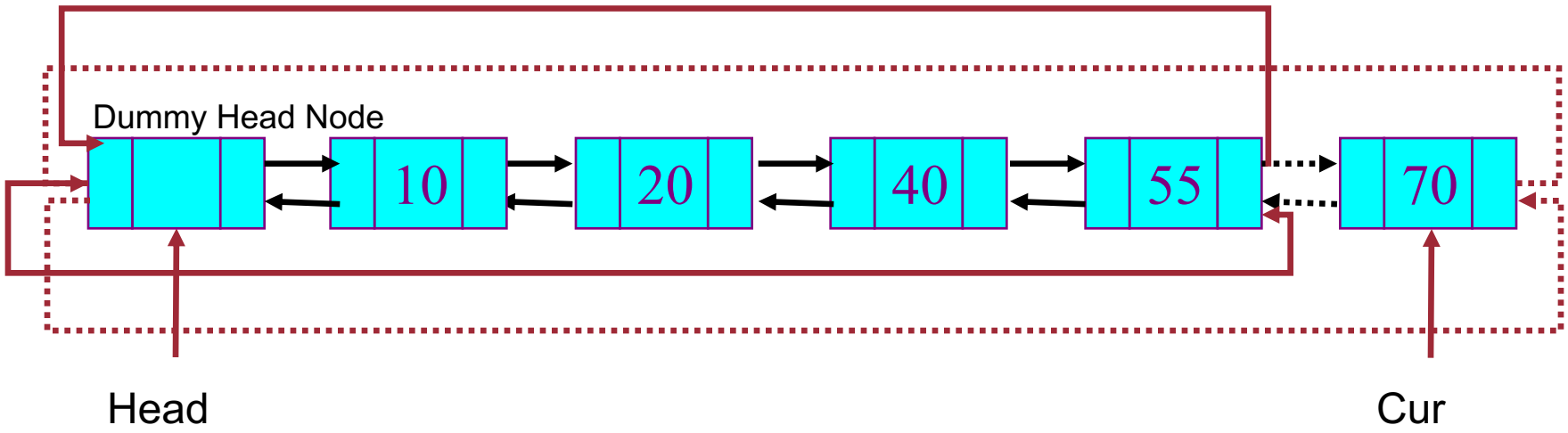
- Delete a node `Cur` in the middle

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



- Delete a node `Cur` at rear

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

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

delete Cur; // same as delete front and middle!
```

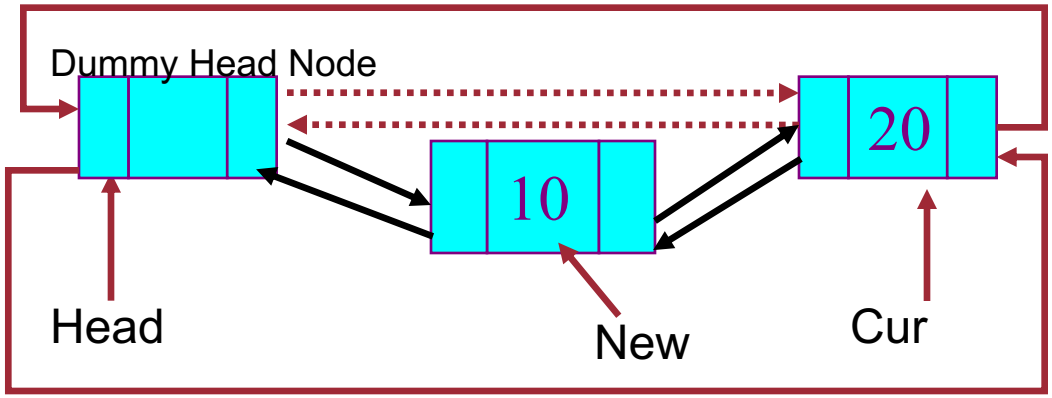


```
void deleteNode(NodePtr Head, int item){
  NodePtr Cur;
  Cur = searchNode(Head, item);
  if(Cur != NULL){
     Cur->prev->next = Cur->next;
     Cur->next->prev = Cur->prev;
     delete Cur;
  }
}
```

Inserting a Node

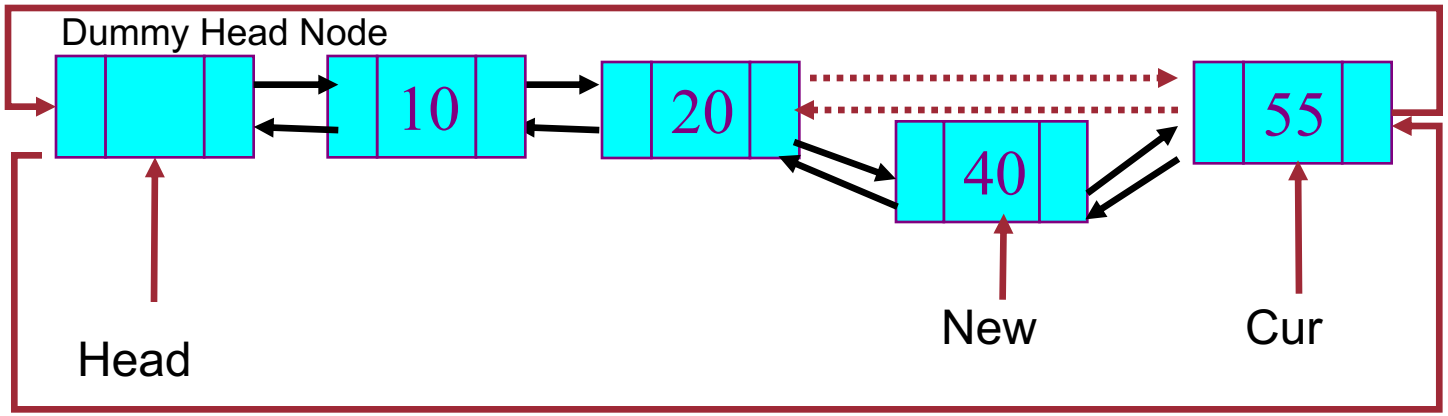
- Insert a Node `New` after dummy node and before `Cur`

```
New->next = Cur;

New->prev = Cur->prev;

Cur->prev = New;

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



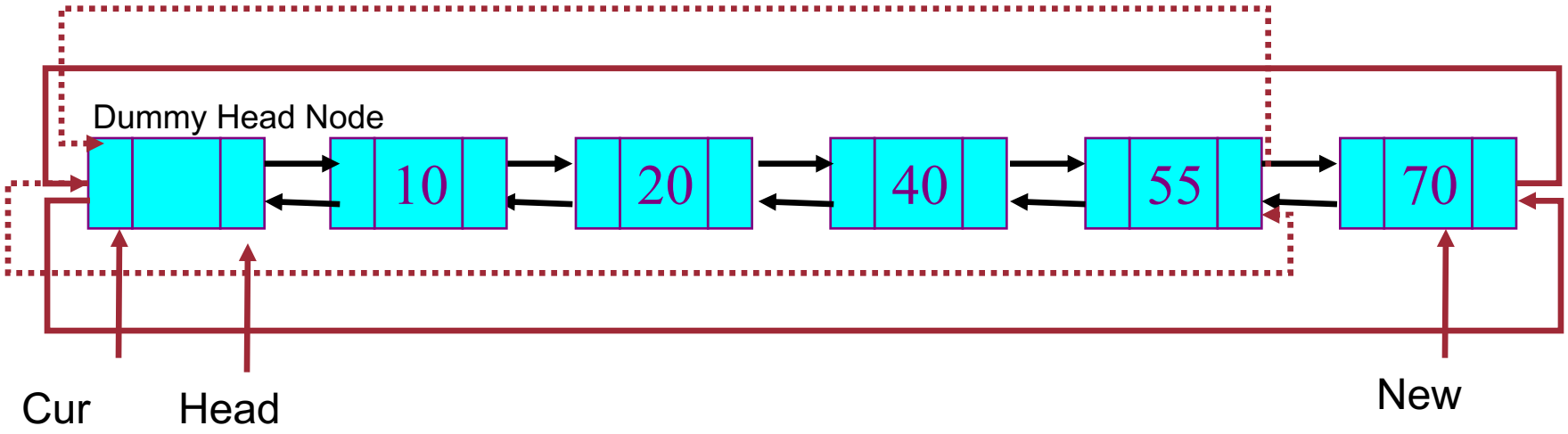
- Insert a Node `New` in the middle and before Cur

```
New->next = Cur;

New->prev = Cur->prev;

Cur->prev = New;

(New->prev)->next = New; // same as insert front!
```



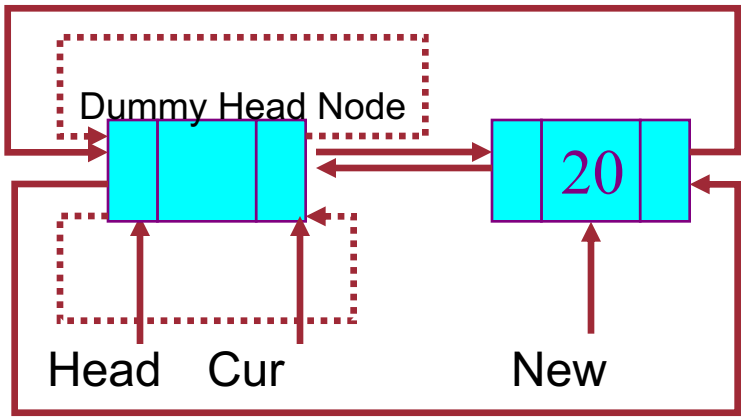
- Insert a Node `New` at Rear (with `Cur` pointing to dummy head)

```
New->next = Cur;

New->prev = Cur->prev;

Cur->prev = New;

(New->prev)->next = New; // same as insert front!
```



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

```
New->next = Cur;

New->prev = Cur->prev;

Cur->prev = New;

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



```
void insertNode(NodePtr Head, int item){
   NodePtr New, Cur;
   New = new Node;
   New->data = item;

   Cur = Head->next;
   while(Cur != Head){   //position Cur for insertion
       if(Cur->data < item)
             Cur = Cur->next;
       else
             break;
   }
   New->next = Cur;
   New->prev = Cur->prev;
   Cur->prev = New;
   (New->prev)->next = New;
}
```

Searching a node:

```
NodePtr searchNode(NodePtr Head, int item){
  NodePtr Cur = Head->next;
  while(Cur != Head){
     if(Cur->data == item)
          return Cur;
     if(Cur->data < item)
          Cur = Cur->next;
     else
          break;
  }
  return NULL;
}
```

Print the whole list:

```
void print(NodePtr Head){
  NodePtr Cur=Head->next;
  while(Cur != Head){
     cout << Cur->data << " ";
     Cur = Cur->next;
  }
  cout << endl;
}
```