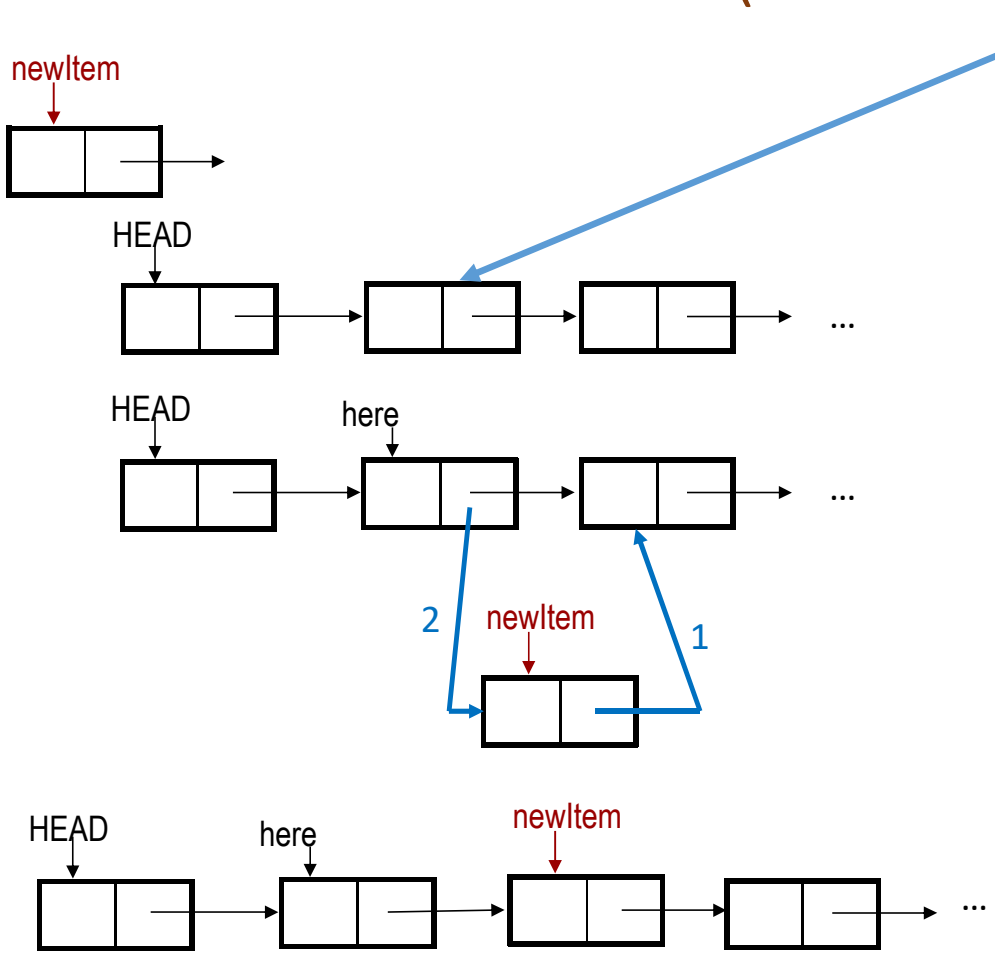
CSE 105:
Data Structures and Algorithms-I
(Part 2)

Instructor

Dr Md Monirul Islam

Insert at Middle (after a desired node)



Review

```
newItem = //create it as before
if (newItem ==NULL)  //error handling
newItem ->data= //assign it ;
newItem ->next=here ->next;
here ->next= newItem;
```

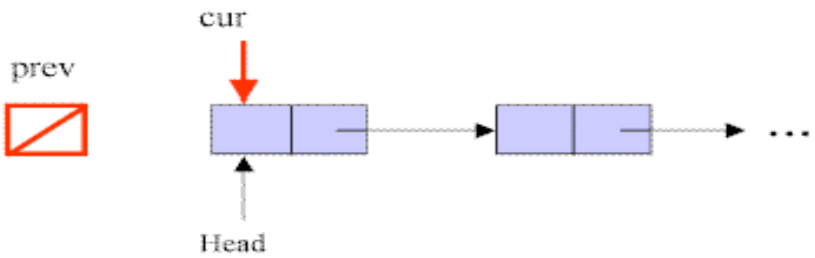
Complexity?

Insert a node after a given value

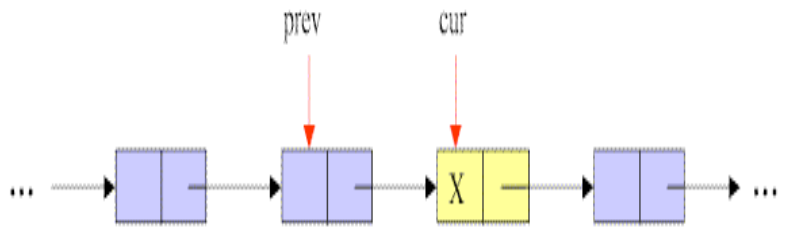
Review

```
//head is the start of list
//value is the given value
//newItem is the node to be inserted


   for (struct node *here = head; here != null; here = here->next {
      if (here->data==value) {
            newItem ->next=here ->next;
            here ->next= newItem;
            exit loop;  //done
      } // if
   } // for
   // Couldn't insert--do something reasonable!
}
```

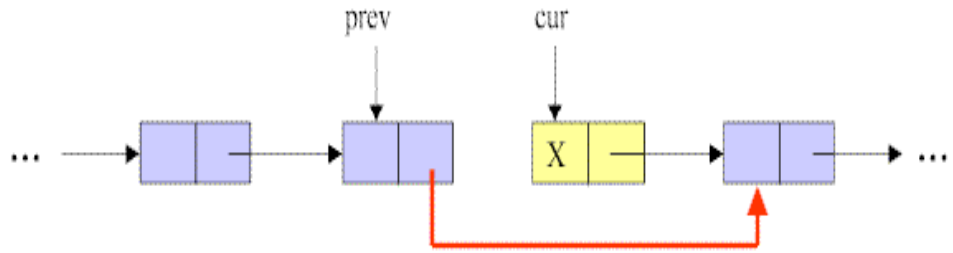
Delete Any



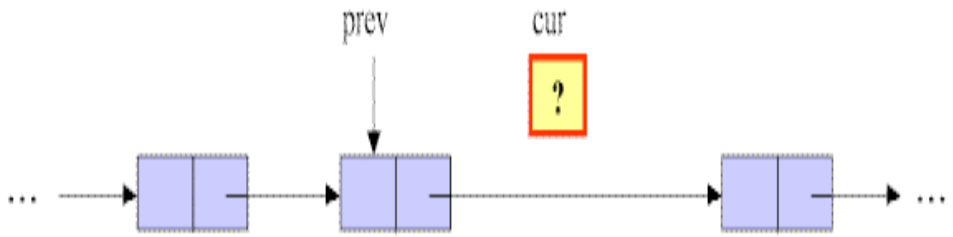
Step 1



Step 2



Step 3



Step 4

Review

```
If head==NULL
      //error handling and return

cur =head;
prev=NULL;
while (cur)
      if curr->value ==x
          break;
      prev =cur;
      cur =cur->next;

if (cur==NULL) //NOT Found, Return

If (prev) prev ->next=curr->next;
else head= head->next;

free (cur);
```

Comparison of Implementations

Array-Based Lists:

- Insertion and deletion are $\Theta(n)$.
- Prev, next and direct access are $\Theta(1)$.
- Array must be allocated in advance.
- No overhead if all array positions are full.

Linked Lists:

- Insertion and deletion are $\Theta(1)$.
- Prev and direct access are $\Theta(n)$.
- Space grows dynamically.
- Every element requires overhead.

Space Comparison

When the array based implementation is better

in space?

Space Comparison

When the array based implementation is better in space?

n: The number of existing elements
D: Maximum number of elements for array.
E: Space for data value.
P: Space for pointer.
Array Based space: DE
LL based space: (E+P)*n

Space Comparison

When the array based implementation is better

in space?

$n(P + E) > DE \Rightarrow n > DE/(P+E)$

n: The number of existing elements
D: Maximum number of elements for array.
E: Space for data value.
P: Space for pointer.
Array Based space: DE
LL based space: (E+P)*n

Space Comparison

When the array based implementation is better

in space?

$n(P + E) > DE \Rightarrow n > DE/(P+E)$

If P=E, the point is at D/2.

n: The number of existing elements
D: Maximum number of elements for array.
E: Space for data value.
P: Space for pointer.
Array Based space: DE
LL based space: (E+P)*n

Space Comparison

When the array based implementation is better

in space?

$n(P + E) > DE \Rightarrow n > DE/(P+E)$

If P=E, the point is at D/2.

=> array-based implementation is more efficient if

- the link field size = the element field size, and
- the array is more than half full.

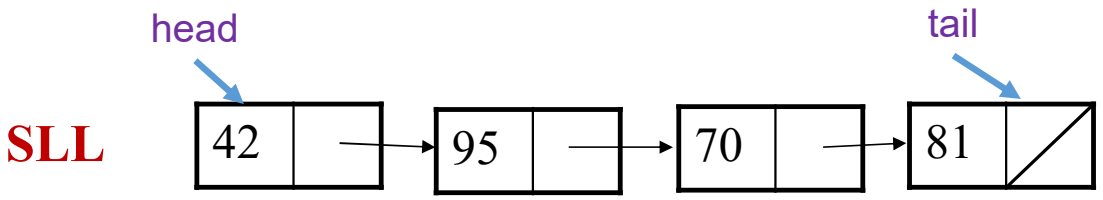
n: The number of existing elements
D: Maximum number of elements for array.
E: Space for data value.
P: Space for pointer.
Array Based space: DE
LL based space: (E+P)*n

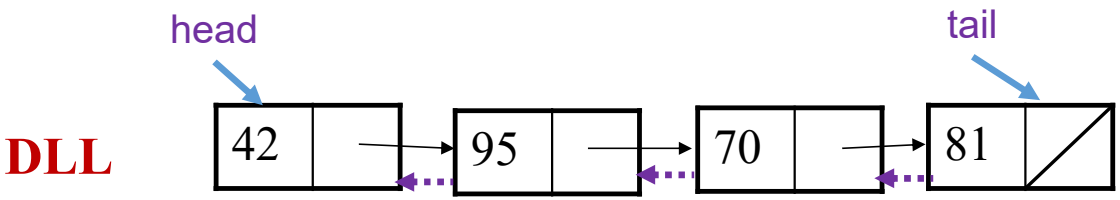
Rule of Thumb

- Linked list is more space efficient when the number of elements varies widely or is unknown.
- Array-based list is generally more space efficient when we know approximately how large the list will become.

Doubly Linked List (DLL)

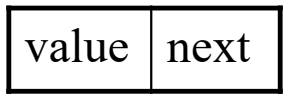
- In SLL, we can only traverse in one direction.
- Often, we need to traverse an LL list in both directions



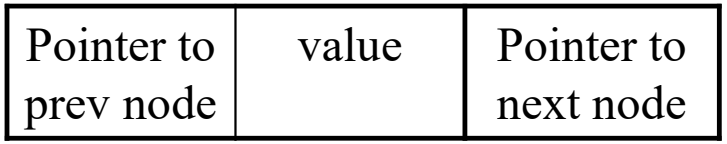


Doubly linked list

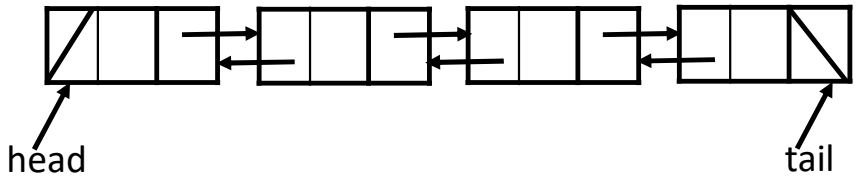
- Each node contains a value, a link to its successor (if any), and a link to its predecessor (if any)
- The header points to the first node in the list



An SLL node

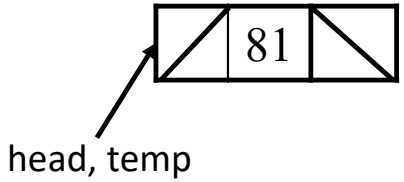


A DLL node



DLL Creation

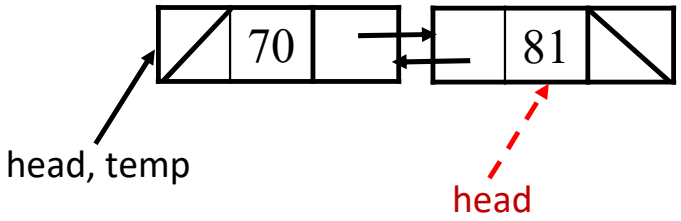
```
// linked list node in C
struct DLLnode {
    int data;
    struct node *next, *prev;
}
```




```
struct DLLnode *head, *temp;
temp = (struct DLLnode *) malloc (sizeof (struct DLLnode));
if (temp==NULL)  //error handling code
temp->data=81;
temp->next=temp->prev=NULL;
head=temp;
```

DLL Creation

```
// linked list node in C
struct DLLnode {
    int data;
    struct node *next, *prev;
}
```




```
struct DLLnode *head, *temp;
temp = (struct DLLnode *) malloc (sizeof (struct DLLnode));
if (temp==NULL)  //error handling code
temp->data=81;
temp->next=temp->prev=NULL;
head=temp;
```

```
temp = (struct DLLnode*) malloc (sizeof (struct DLLnode));
if (temp==NULL)  //error handling code
temp->data=70;
temp->next=temp->prev=NULL;
temp->next=head;
head->prev=temp;
head=temp;
```

DLL pros and cons

- 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)
  - Greater chance of having bugs (because more links must be manipulated)

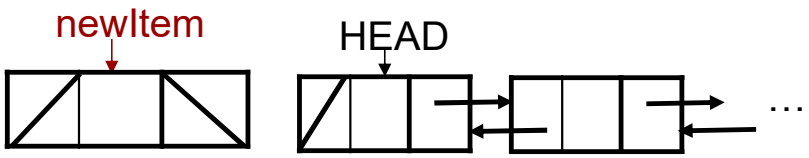
Basic Operations of DLL

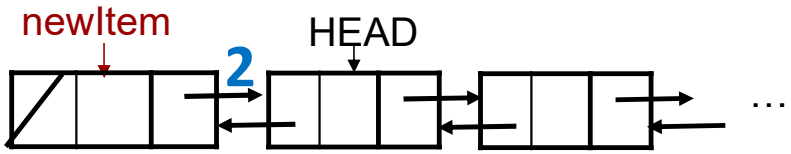
- Insert: Add a new node in the first, last or interior of the list.
- Delete: Delete a node from the first, last or interior of the list.
- Search: Search a node containing particular value in the linked list.

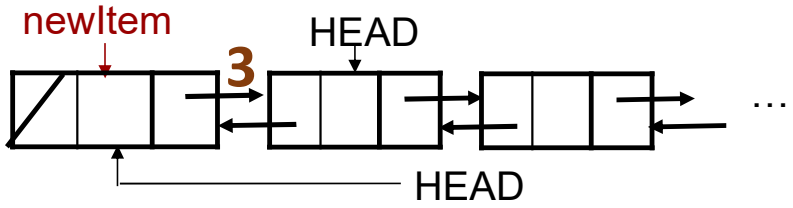
Insert at the beginning of a DLL

Very similar to the SLL

- Step 1. Create a new node, *newItem*.
- Step 2. *newItem is* linked to the first node of the DLL
- Step 3. Set the pointer *head* to *newItem*.



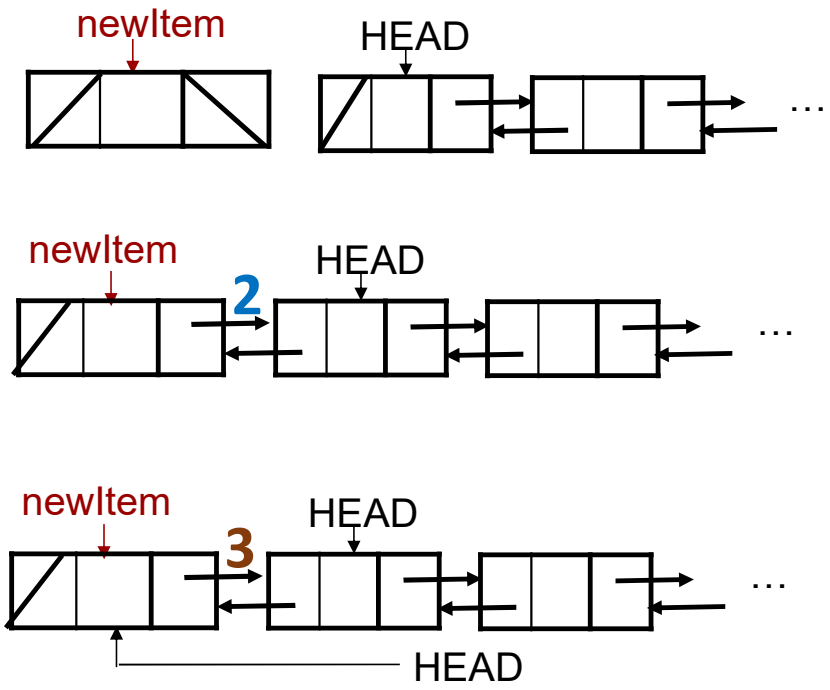




Insert at the beginning of a DLL

Very similar to the SLL

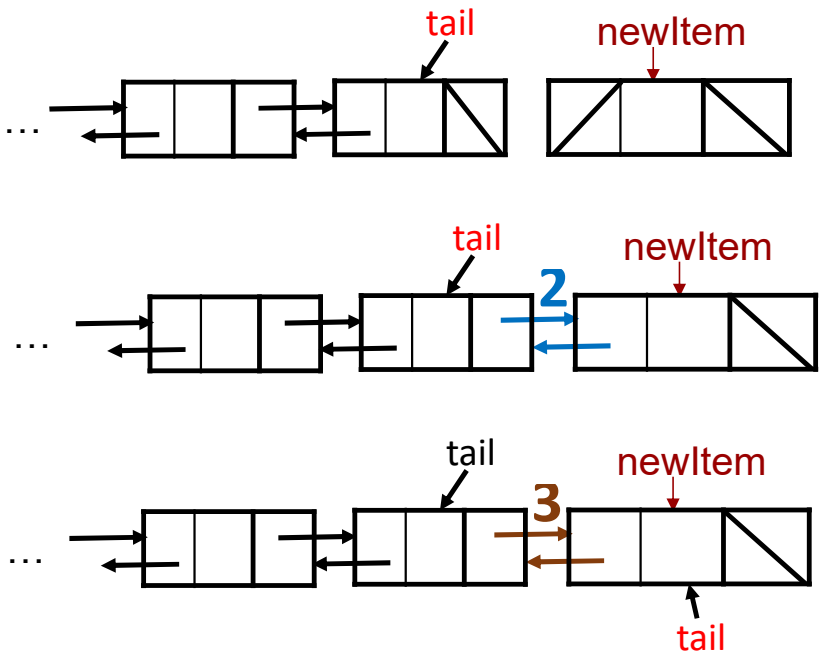
- Step 1. Create a new node, *newItem*.
- Step 2. *newItem is* linked to the first node of the DLL
- Step 3. Set the pointer *head* to *newItem*.




```
struct DLLnode *head, *temp;
newItem = (struct DLLnode*) malloc (sizeof (struct DLLnode));
if (newItem ==NULL)  //error handling code
newItem ->next=temp->prev=NULL;
newItem ->next=head;
 if (head !=NULL) head ->prev= newItem;

head= newItem;
```

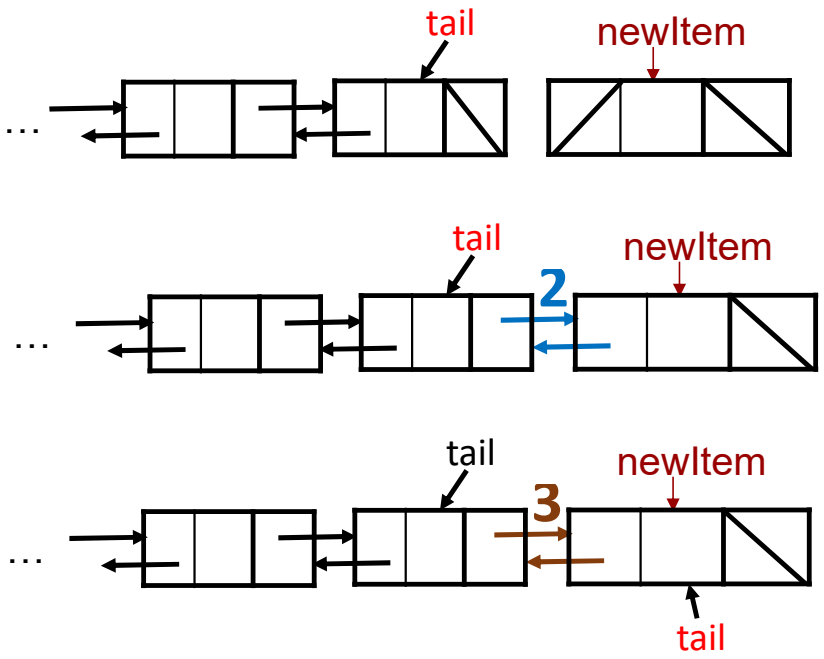
Insert at the end of a DLL

- Step 1. Create a new node, *newItem*.
- Step 2. *newItem is* linked to the tail of the DLL
- Step 3. Set the pointer tail to *newItem*.



Insert at the end of a DLL

- Step 1. Create a new node, *newItem*.
- Step 2. *newItem is* linked to the tail of the DLL
- Step 3. Set the pointer tail to *newItem*.

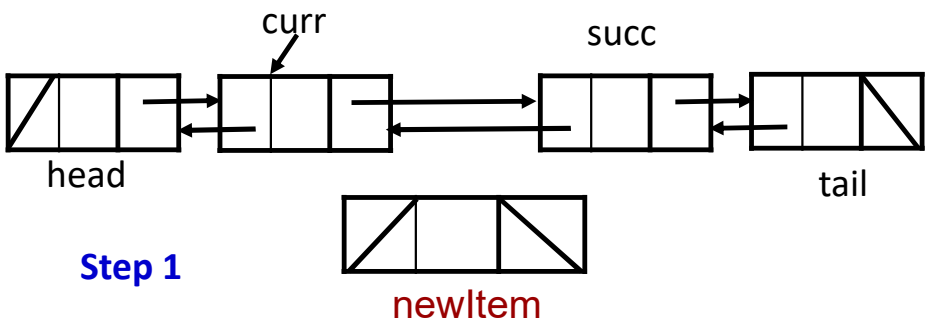


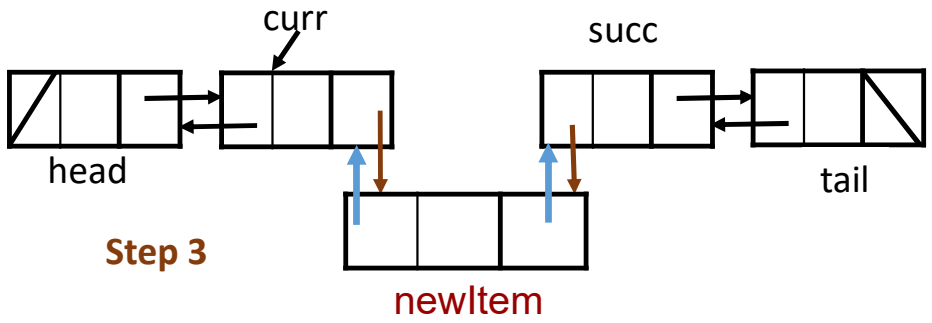

```
struct DLLnode *head, *temp;
newItem = (struct DLLnode*) malloc (sizeof (struct DLLnode));
if (newItem ==NULL)  //error handling code
newItem ->next=temp->prev=NULL;
newItem ->prev=tail;
 if (tail !=NULL) tail->next= newItem;

tail= newItem;
```

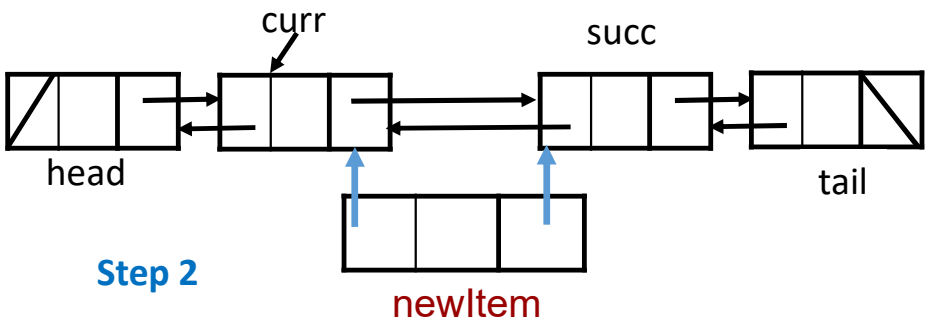
Insert interior of a DLL

Assume we will insert after **curr** AND **succ** is the next node of curr.

- Step 1. Create a new node, *newItem*.
- Step 2. *newItem->prev* is linked curr and *newItem->nex* is linked to succ
- Step 3. Set curr->next = newItem and succ->prev= newItem



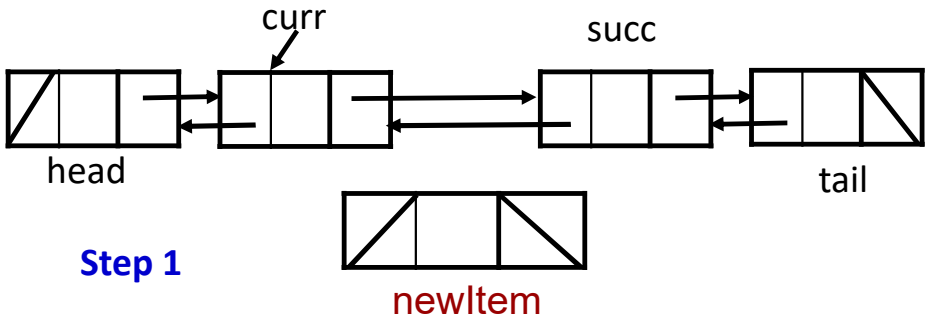


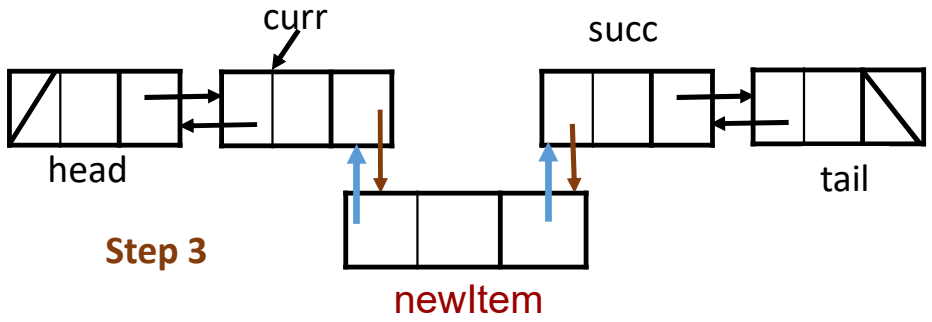


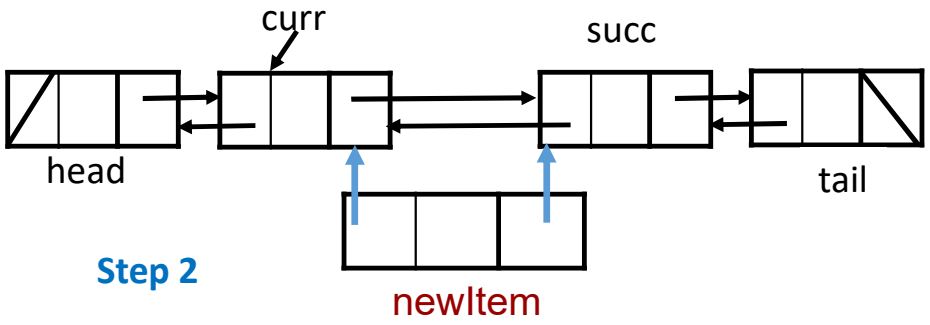
Insert interior of a DLL

Assume we will insert after **curr** AND **succ** is the next node of curr.

- Step 1. Create a new node, *newItem*.
- Step 2. *newItem->prev* is linked curr and *newItem->nex* is linked to succ
- Step 3. Set curr->next = newItem and succ->prev= newItem








```
newItem ->next=succ; //curr->next
newItem ->prev=curr;
if (succ) succ->prev=newItem;
if (curr) curr->next= newItem;
```

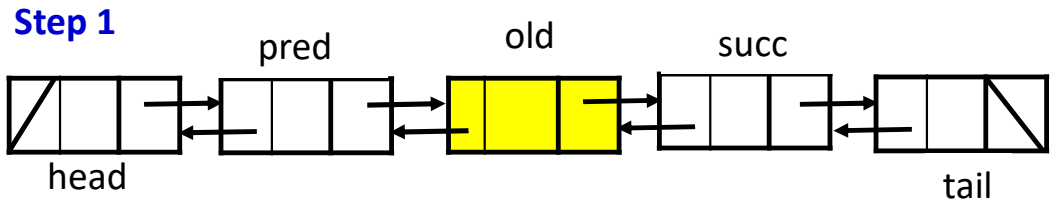
Deletion of a node in DLL

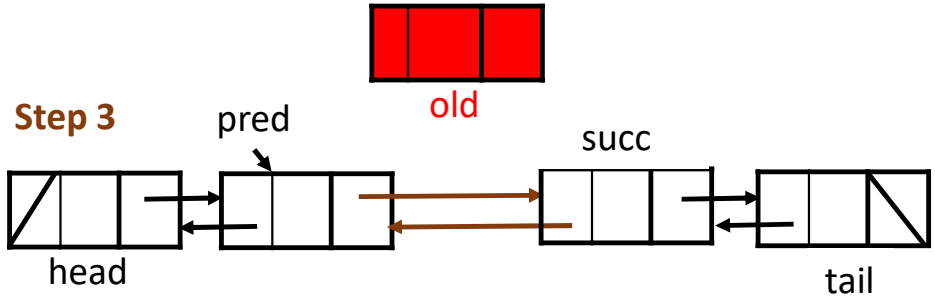
- Deletion in DLL is easier than deletion in SLL
- In SLL, we have to find the predecessor of the discarded node in $O(n)$
- In DLL, the predecessor and successor of a node are immediately known.

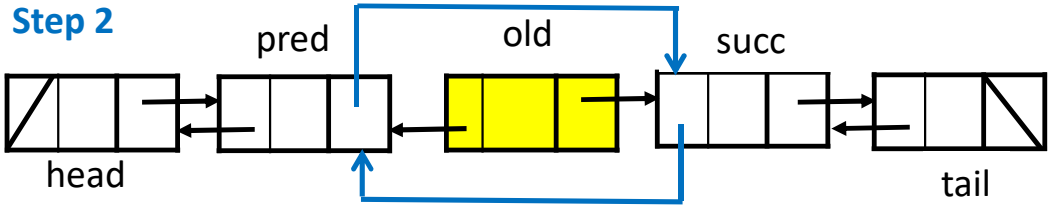
Deletion of a node in DLL

Assume old is to be deleted. pred and succ are its previous and next nodes

- Step1. Set pointer pred =old->prev and succ= old->next
- Step2. Set pred->next=succ and succ->prev= pred
- Step3. Discard the node pointed by *old*.





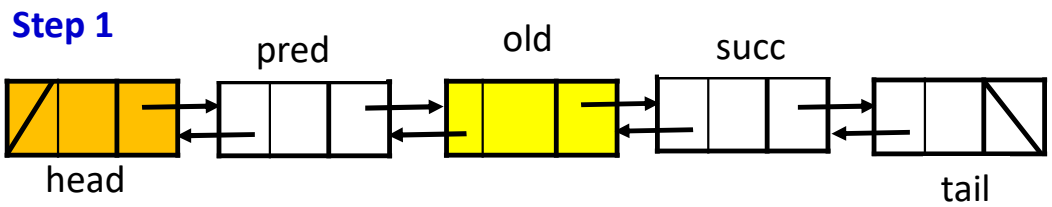


Deletion of a node in DLL (head being deleted)

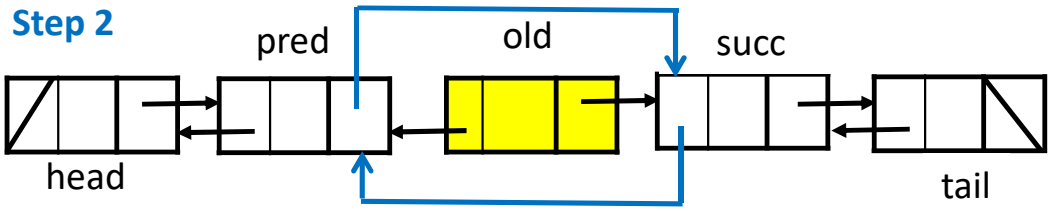
Assume old is to be deleted. pred and succ are its previous and next nodes

- Step1. Set pointer pred =old->prev and succ= old->next
- Step2. Set pred->next=succ and succ->prev= pred
- Step3. Discard the node pointed by *old*.

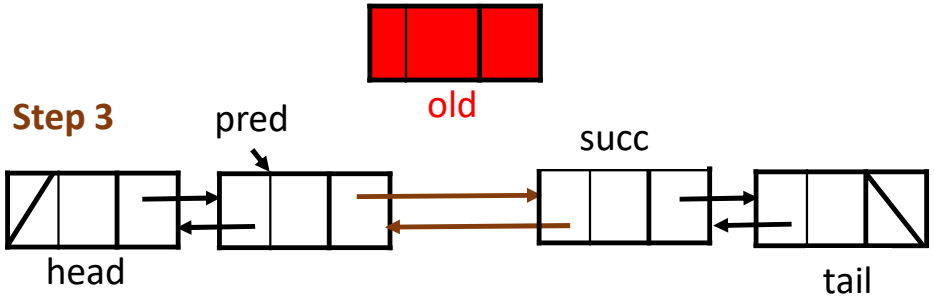
Step 1



Step 2



Step 3



Will it work?

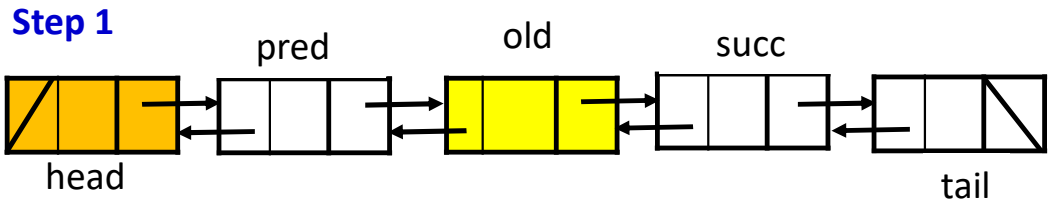
```
pred=old->prev;      succ=old->next;
pred->next=succ;
succ->prev=pred;
free(old)
```

Deletion of a node in DLL (head being deleted)

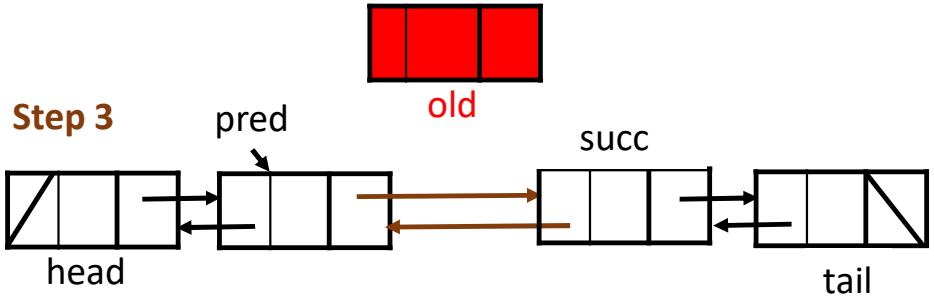
Assume old is to be deleted. pred and succ are its previous and next nodes

- Step1. Set pointer pred =old->prev and succ= old->next
- Step2. Set pred->next=succ and succ->prev= pred
- Step3. Discard the node pointed by *old*.

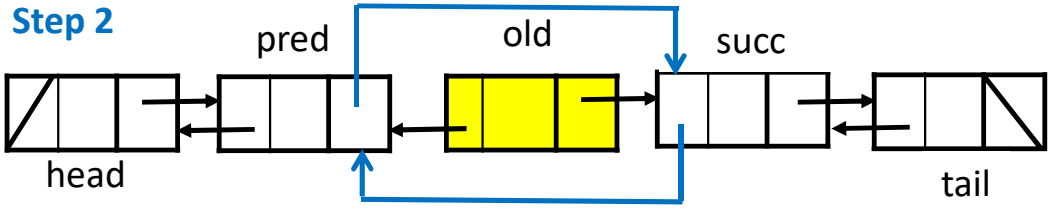
Step 1



Step 3



Step 2



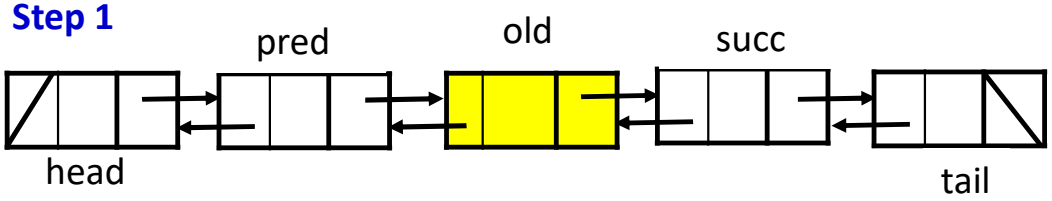

```
pred=old->prev;      succ=old->next;
if (pred) pred->next=succ;
else head=succ;
succ->prev=pred;
free(old)
```

Deletion of a node in DLL (**tail** being deleted)

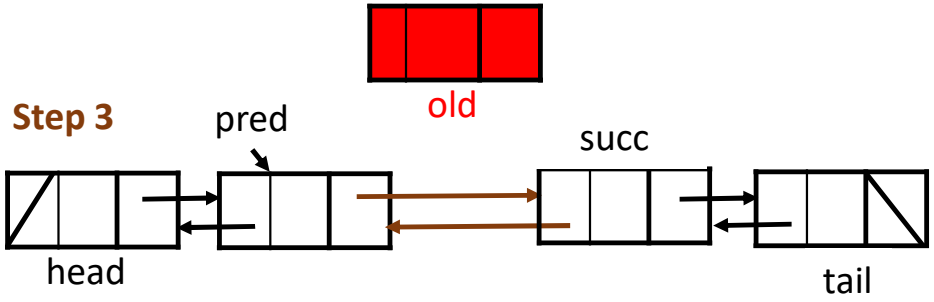
Assume old is to be deleted. pred and succ are its previous and next nodes

- Step1. Set pointer pred =old->prev and succ= old->next
- Step2. Set pred->next=succ and succ->prev= pred
- Step3. Discard the node pointed by *old*.

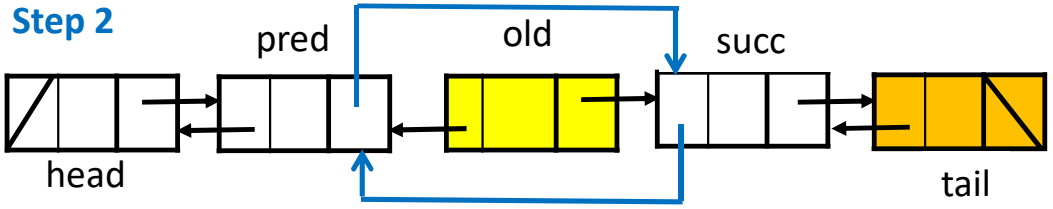
Step 1



Step 3



Step 2



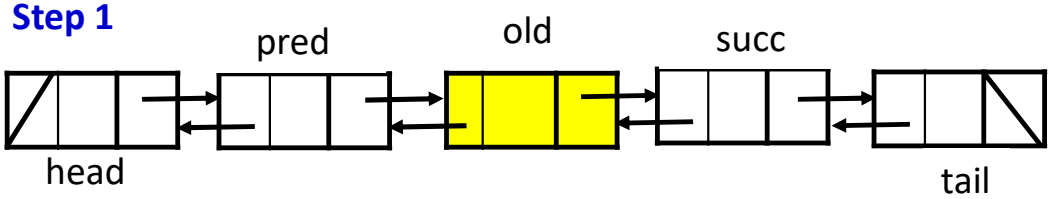

```
pred=old->prev;      succ=old->next;
if (pred) pred->next=succ;
else head=succ;
if (succ) succ->prev=pred;
else tail=pred;
free(old)
```

Deletion of a node in DLL (tail being deleted)

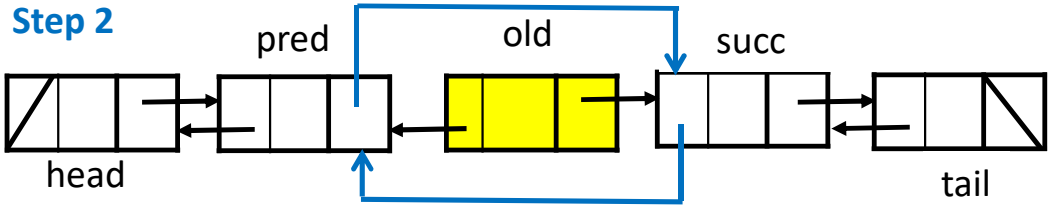
Assume old is to be deleted. pred and succ are its previous and next nodes

- Step1. Set pointer pred =old->prev and succ= old->next
- Step2. Set pred->next=succ and succ->prev= pred
- Step3. Discard the node pointed by *old*.

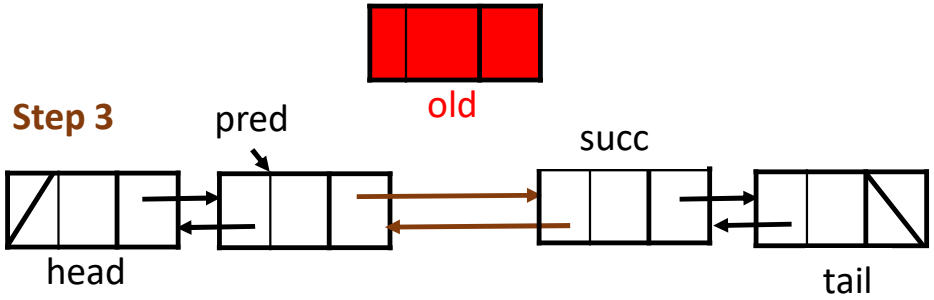
Step 1



Step 2



Step 3



Will it work if old is the only node?

```
pred=old->prev;      succ=old->next;
if (pred) pred->next=succ;
else head=succ;
if (succ) succ->prev=pred;
else tail=pred;
free(old)
```

Singly linked list

	head	*middle*	tail
Find	$\Theta(1)$	$O(n)$	$\Theta(1)$
Insert Before	$\Theta(1)$	$O(n)$	$\Theta(n)$
Insert After	$\Theta(1)$	$\Theta(1)^*$	$\Theta(1)$
Replace	$\Theta(1)$	$\Theta(1)^*$	$\Theta(1)$
Erase	$\Theta(1)$	$O(n)$	$\Theta(n)$
Next	$\Theta(1)$	$\Theta(1)^*$	n/a
Previous	n/a	$O(n)$	$\Theta(n)$

* These assume we have already accessed the middle node—an $O(n)$ operation

Doubly linked lists

	head	*middle*	tail
Find	$\Theta(1)$	$O(n)$	$\Theta(1)$
Insert Before	$\Theta(1)$	$\Theta(1)^*$	$\Theta(1)$
Insert After	$\Theta(1)$	$\Theta(1)^*$	$\Theta(1)$
Replace	$\Theta(1)$	$\Theta(1)^*$	$\Theta(1)$
Erase	$\Theta(1)$	$\Theta(1)^*$	$\Theta(1)$
Next	$\Theta(1)$	$\Theta(1)^*$	n/a
Previous	n/a	$\Theta(1)^*$	$\Theta(1)$

* These assume we have already accessed the middle node—an $O(n)$ operation