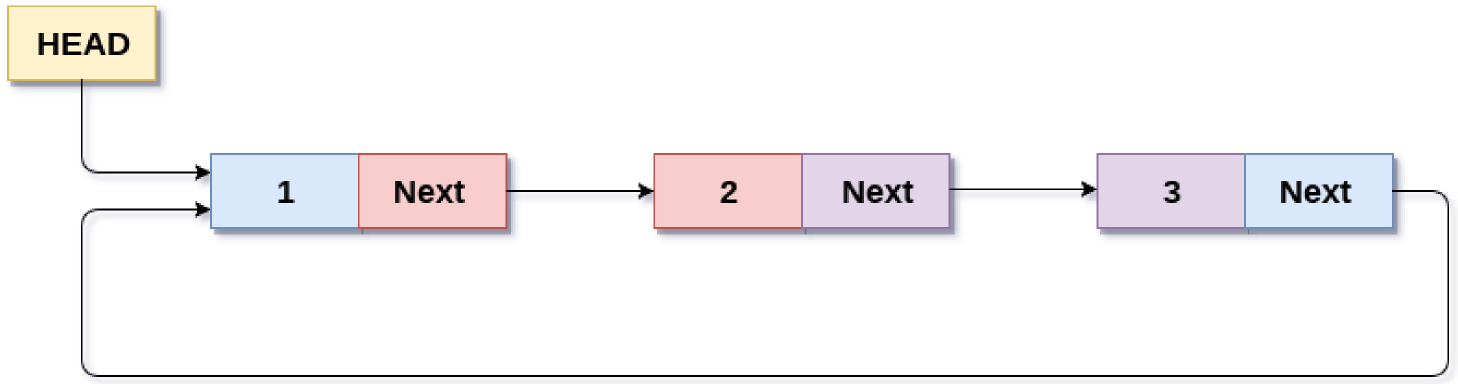
FIRST SEMESTER 2020

Data Structure
Linked List – Part 2

Circular Linked Lists

- the last node of the list contains a pointer to the first node of the list.
- We traverse a circular linked list until we reach the same node where we started.
- The circular linked list has no beginning and no ending.
- There is no null value present in the next part of any of the nodes.
- Circular linked list are mostly used in task maintenance in operating systems
- No longer explicitly maintain the head reference. So long as we maintain a reference to the tail, we can locate the head as the next of tail.



Circular Linked Lists -Example

- There are many examples where circular linked list are being used in computer science including browser surfing where a record of pages visited in the past by the user, is maintained in the form of circular linked lists and can be accessed again on clicking the previous button.

Circular Linked Lists advantages

- Any node can be a starting point. We can traverse the whole list by starting from any point. We just need to stop when the first visited node is visited again.
- We don’t need to maintain two pointers for head and tail if we use circular linked list. We can maintain a pointer to the last inserted node and front can always be obtained as next of last.
- Maintaining only the tail reference not only saves a bit on memory usage, it makes the code simpler and more efficient, as it removes the need to perform additional operations to keep a head reference current.

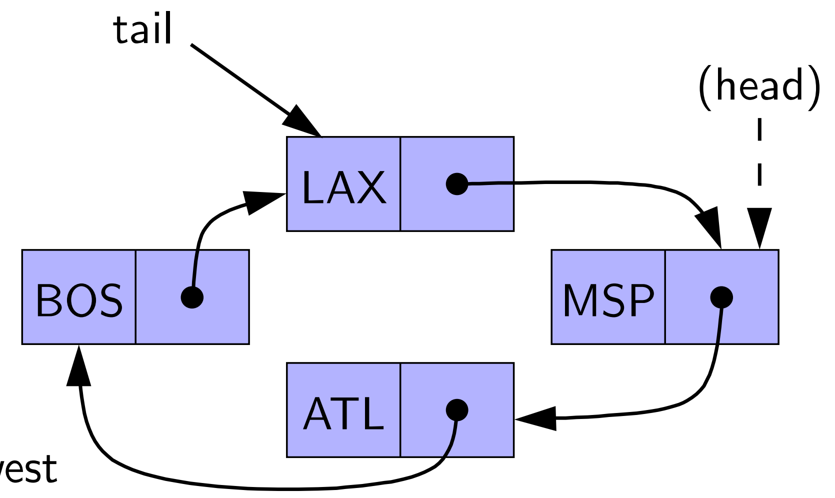
Circular Linked Lists Operations - Traverse

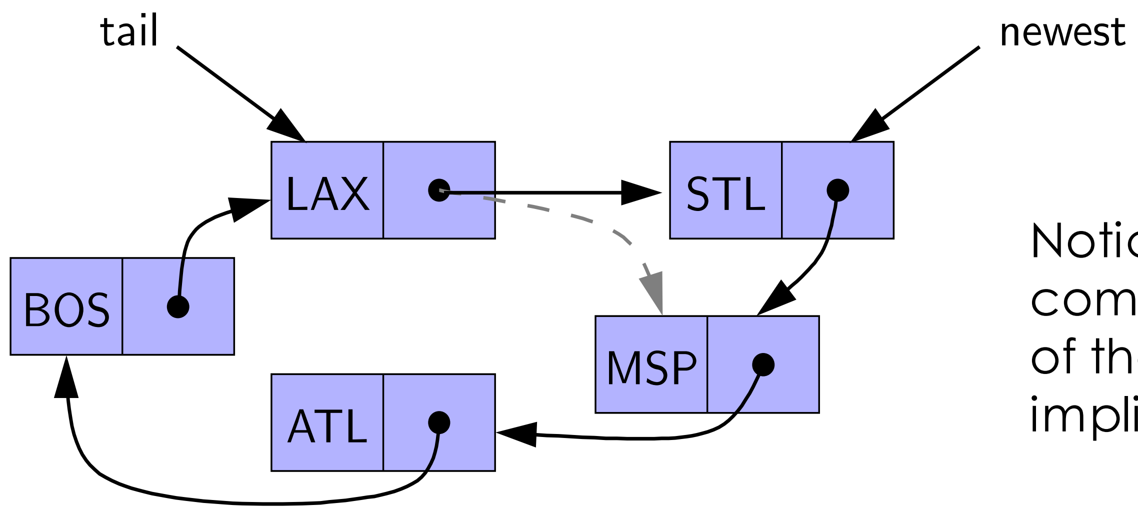
In a conventional linked list, we traverse the list from the head node and stop the traversal when we reach NULL. In a circular linked list, we stop traversal when we reach the first node again.

Traversal(tail):

```
    c= tail->next //Set the current node to head
    IF tail != null // List is not empty
        DO // Keep printing nodes till we reach the first node again
            Print c->data
            c=c->next
        WHILE c!=tail->next
```

Circular Linked Lists Operations – Insert at the beginning





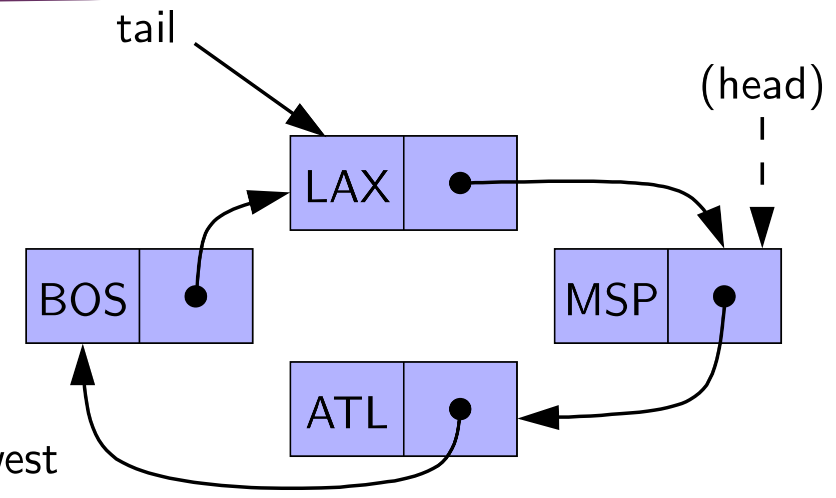
Notice that when the operation is complete, STL is the first element of the list, as it is stored within the implicit head, tail->Next

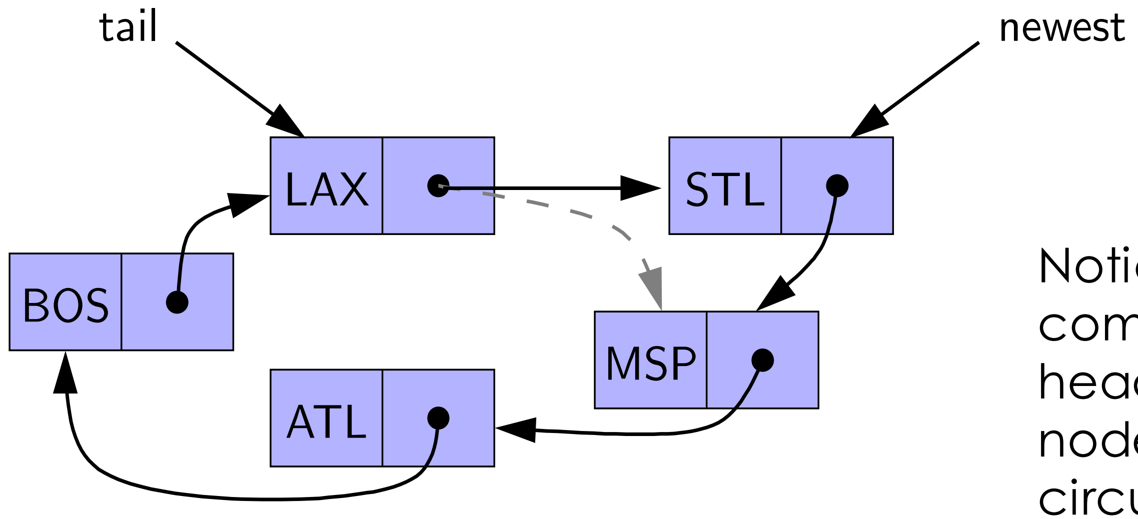
Circular Linked Lists Operations – Insert at the beginning

We can add a new element at the front of the list by creating a new node and linking it just after the tail of the list.

```
addFirst(e):
    newest= Node( e) // create new node
    newest->next=tail->next // set the new node to point at head
    tail->next=newest //set the new node as head
```

Circular Linked Lists Operations – Insert at the End





Notice that when the operation is complete, MSP become the head and STL become the last node because the list is a circular.

Circular Linked Lists Operations – Insert at the End

we can rely on the use of a call to addFirst and then immediately advance the tail reference so that the newest node becomes the last.

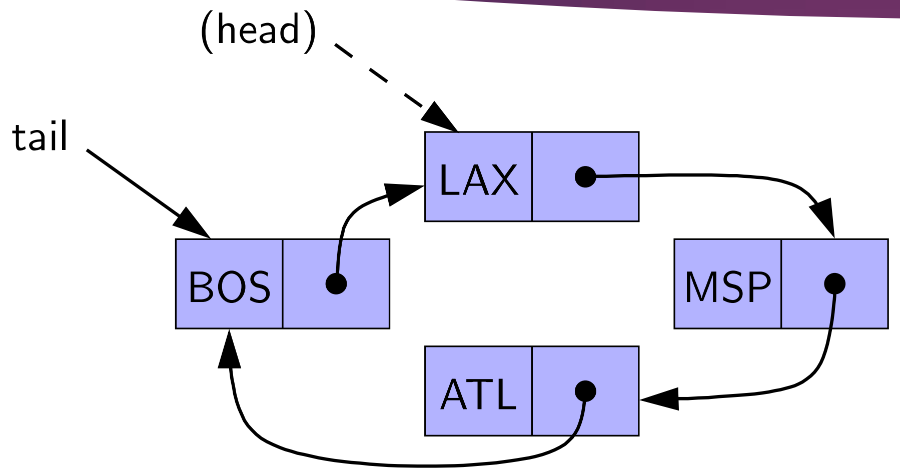
```
addLast(e):
    addFirst( e)// add new node to the beginning of the list
    tail=tail->next //advance the tail to the next node
```

Circular Linked Lists Operations – Delete from the Beginning

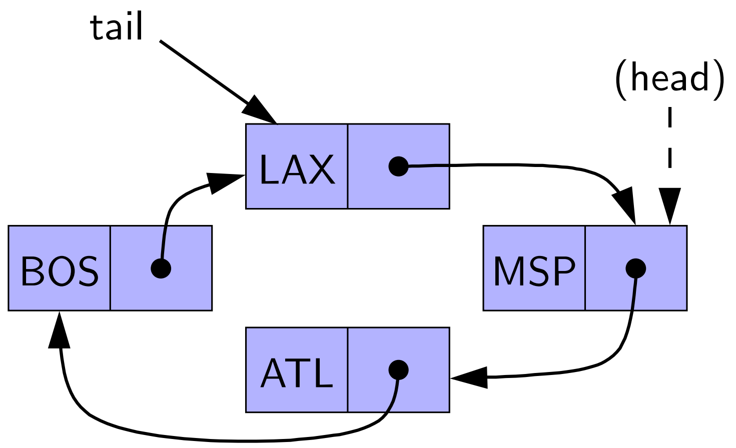
removing the first node from a circularly linked list can be accomplished by simply updating the next field of the tail node to bypass the implicit head.

```
removeFirst():
    head=tail->next// store the head
    IF head==tail // only one node lift
        tail=null //remove last node fom the list
    ELSE
        tail->next=head->next //set the tail next to the next of head
```

Circular Linked Lists Operations – Rotate



(a) before the rotation, representing sequence. { LAX, MSP, ATL, BOS }



b) after the rotation, rep- resenting sequence { MSP, ATL, BOS, LAX }.

Note: the head pointer is only for demonstration the head reference could be obtained as tail next

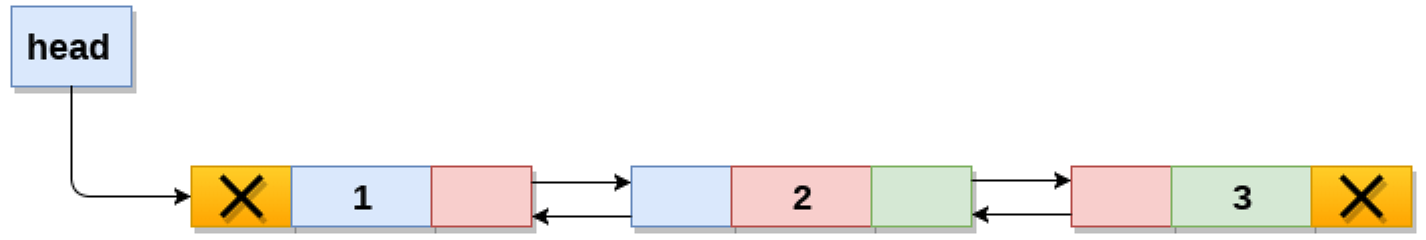
Circular Linked Lists Operations – Rotate

We do not move any nodes or elements, we simply advance the tail reference to point to the node that follows it (the implicit head of the list).

```
rotate(): // rotate the first element to the back of the list
    IF tail != null
        tail=tail->next // the old head becomes the new tail
```

Doubly linked list

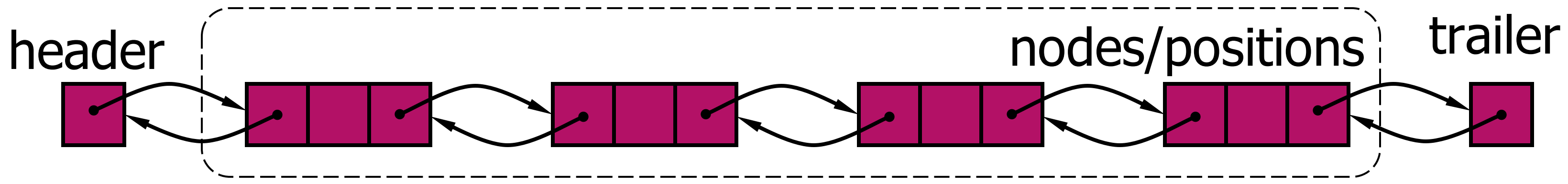
- A doubly linked list can be traversed forward and backward
- Doubly linked list is a complex type of linked list in which a node contains a pointer to the previous as well as the next node in the sequence
- a doubly linked list, a node consists of three parts: node data, pointer to the next node in sequence (next pointer) , pointer to the previous node (previous pointer)



Doubly Linked List

Doubly linked list

- In order to avoid some special cases when operating near the boundaries of a doubly linked list
- It helps to add special nodes at both ends of the list: a header node at the beginning of the list, and a trailer node at the end of the list.
- These "dummy" nodes are known as sentinels (or guards), and they do not store elements of the primary sequence.
- When using sentinel nodes, an empty list is initialized so that the next field of the header points to the trailer, and the prev field of the trailer points to the header;

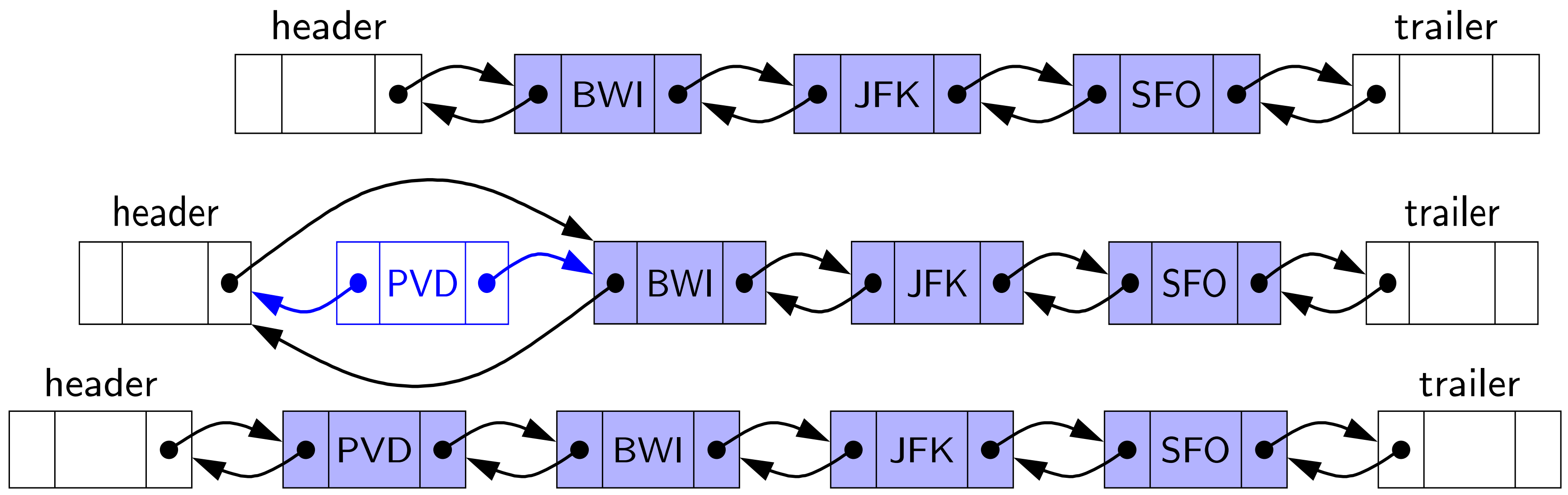


Doubly linked list

- We could implement a doubly linked list without sentinel nodes.
- But the slight extra memory devoted to the sentinels greatly simplifies the logic of our operations.
- The header and trailer nodes never change—only the nodes between them change.
- We can treat all insertions in a unified manner, because a new node will always be placed between a pair of existing nodes.
- Every element that is to be deleted is guaranteed to be stored in a node that has neighbors on each side.

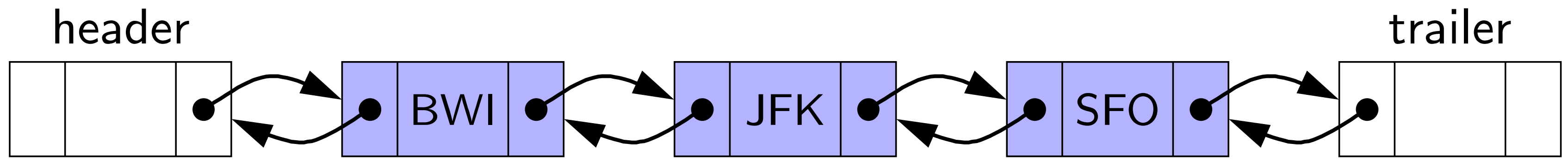
Doubly Linked List operations - Insert

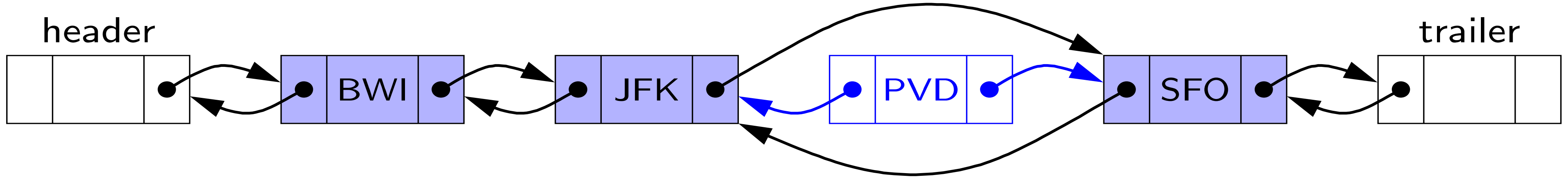
- Every insertion into our doubly linked list representation will take place between a pair of existing nodes.
- when a new element is inserted at the front of the sequence, we will simply add the new node between the header and the node that is currently after the header.

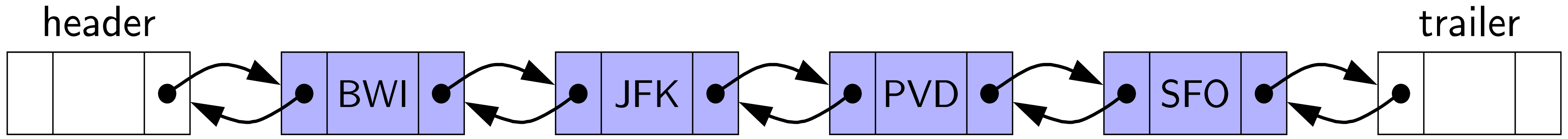


Doubly Linked List operations - Insert

- Insert a new node, q, between p and its successor.







Doubly Linked List operations - Insert

- Insert a new node, q, between p and its successor.

```
addAfter(p,e):
    newest=Node (e)
    newest->prev=p //link new node to its predecessor
    newest->next=p->next //link new node to its successor
    successor=p->next
    successor->prev=newest//link p's old successor to new node
    p->next=newest//link p to its new successor
```

Doubly Linked List operations - Insert

- Insert a new node, q, between p and its predecessor.

```
addBefore(p,e):
    newest=Node(e)
    newest->next=p //link new node to its successor
    predecessor=p->pev // hold p pervious node
    p->prev=newest  // link p to new node
    newest->prev =predecessor  // link new node to its predecessor
    predecessor->next= newest //link the  predecessor to the new
   node
```

Doubly Linked List operations - Insert

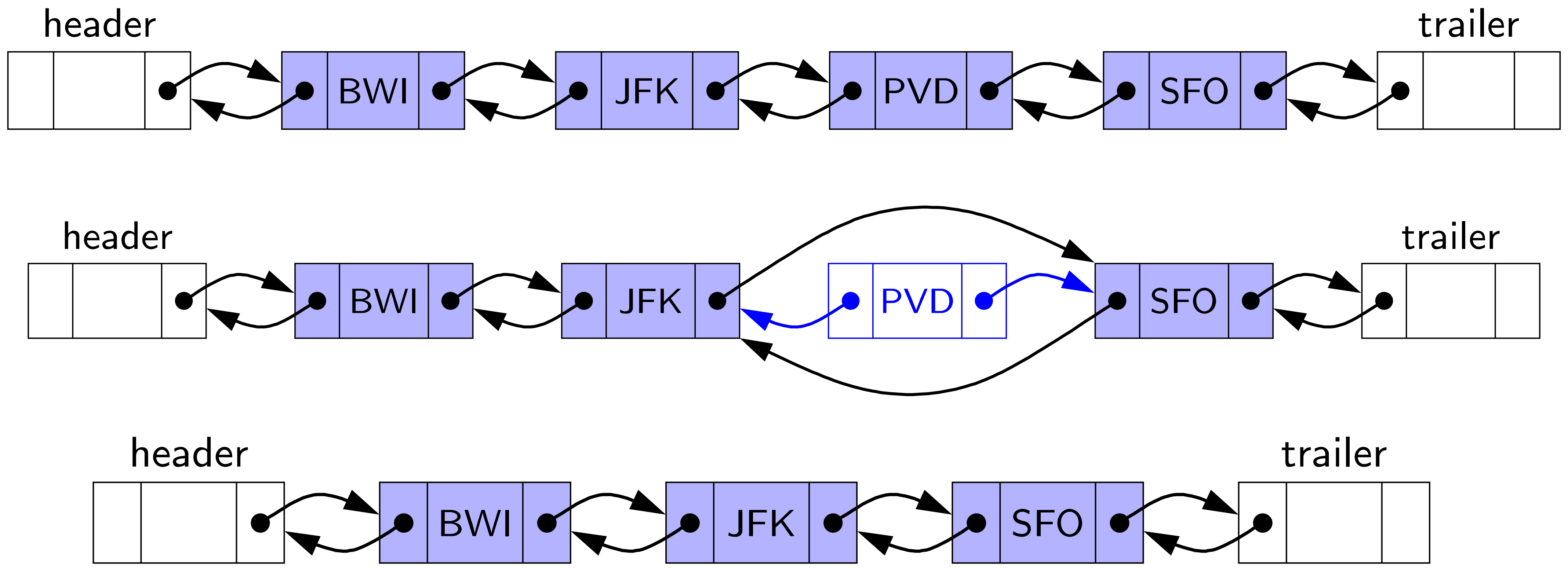
- Insert a new node, q, between its predecessor and its successor.

```
addBetween(predecessor ,successor ,e):
    newest=Node (e)
    newest->prev=predecessor //link new node to its predecessor
    newest->next= successor //link new node to its successor
    successor->prev=newest//link successor to new node
    predecessor->next=newest   //link predecessor to its new successor
```

Doubly Linked List operations - Delete

- To delete a node, the two neighbors of the node to be deleted are linked directly to each other, thereby bypassing the original node.
- As a result, that node will no longer be considered part of the list
- Because of our use of sentinels, the same implementation can be used when deleting the first or the last element of a sequence, because even such an element will be stored at a node that lies between two others.

Doubly Linked List operations - Delete



Doubly Linked List operations - Delete

- Remove a node, p, from a doubly linked list.

```
delete(p):
    predecessor =p->prev //hold pervious node
    successor =p->next //hold next node
     predecessor ->next= successor //link previous node to next node
     successor >prev= predecessor    //link next node to previous node
    p->next=null //remove p pointers
    p->prev=null //remove p pointers
```