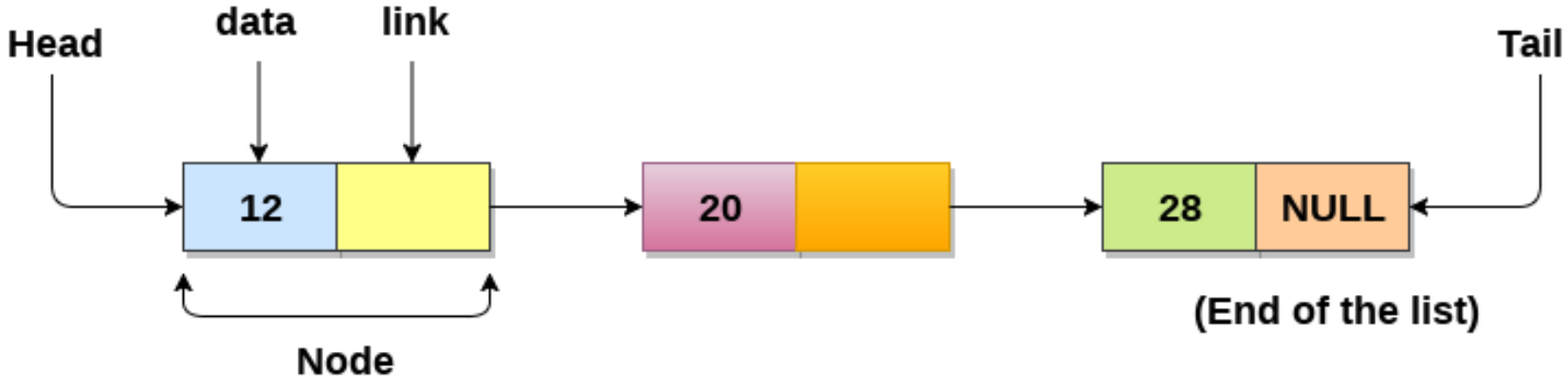
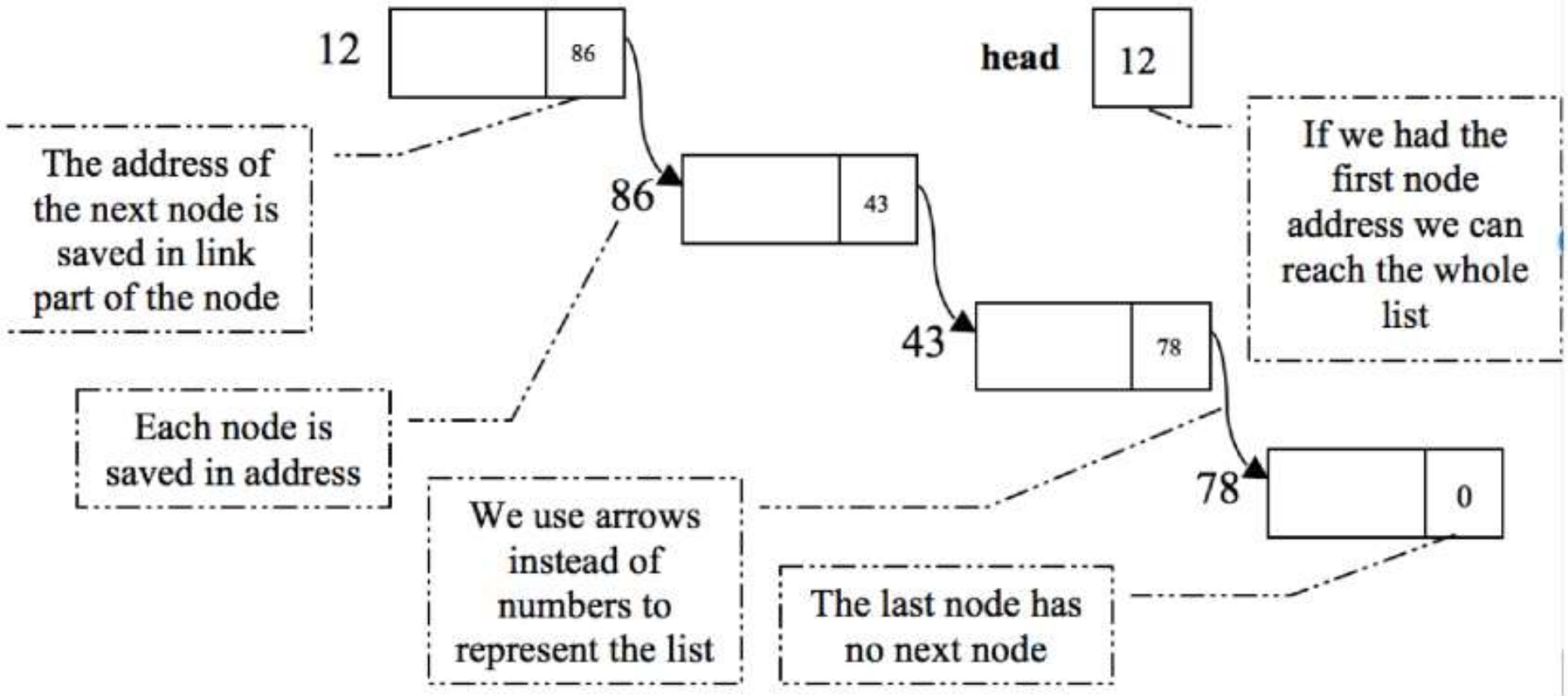
Data Structure

Linked List

Definition

- Linked List can be defined as collection of objects called **nodes** that are randomly stored in the memory
- A node contains two fields i.e., data stored at that particular address and the pointer which contains the address of the next node in the memory.
- The last node of the list contains pointer to the null.

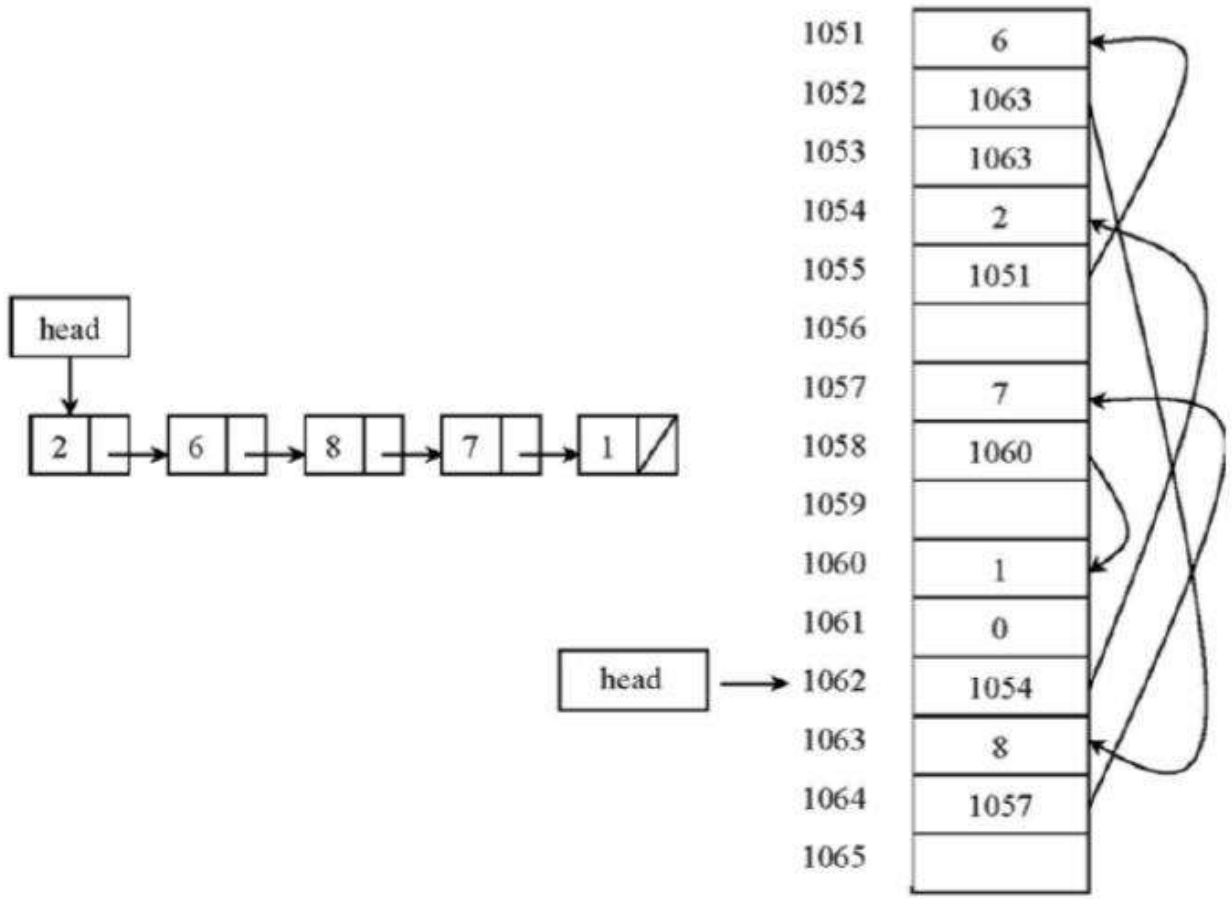



12
86
head
12
The address of the next node is saved in link part of the node
86
43
If we had the first node address we can reach the whole list
43
78
Each node is saved in address
78
0
We use arrows instead of numbers to represent the list
The last node has no next node

Uses of Linked List

- The list is not required to be contiguously present in the memory. The node can reside any where in the memory and linked together to make a list. This achieves optimized utilization of space.
- list size is limited to the memory size and doesn't need to be declared in advance.
- Empty node can not be present in the linked list.
- We can store values of primitive types or objects in the singly linked list

Representation of Linked Lists in Memory



Why use linked list over array?

- It allocates the memory dynamically. All the nodes of linked list are non- contiguously stored in the memory and linked together with the help of pointers.
- Sizing is no longer a problem since we do not need to define its size at the time of declaration. List grows as per the program's demand and limited to the available memory space.

Singly linked list

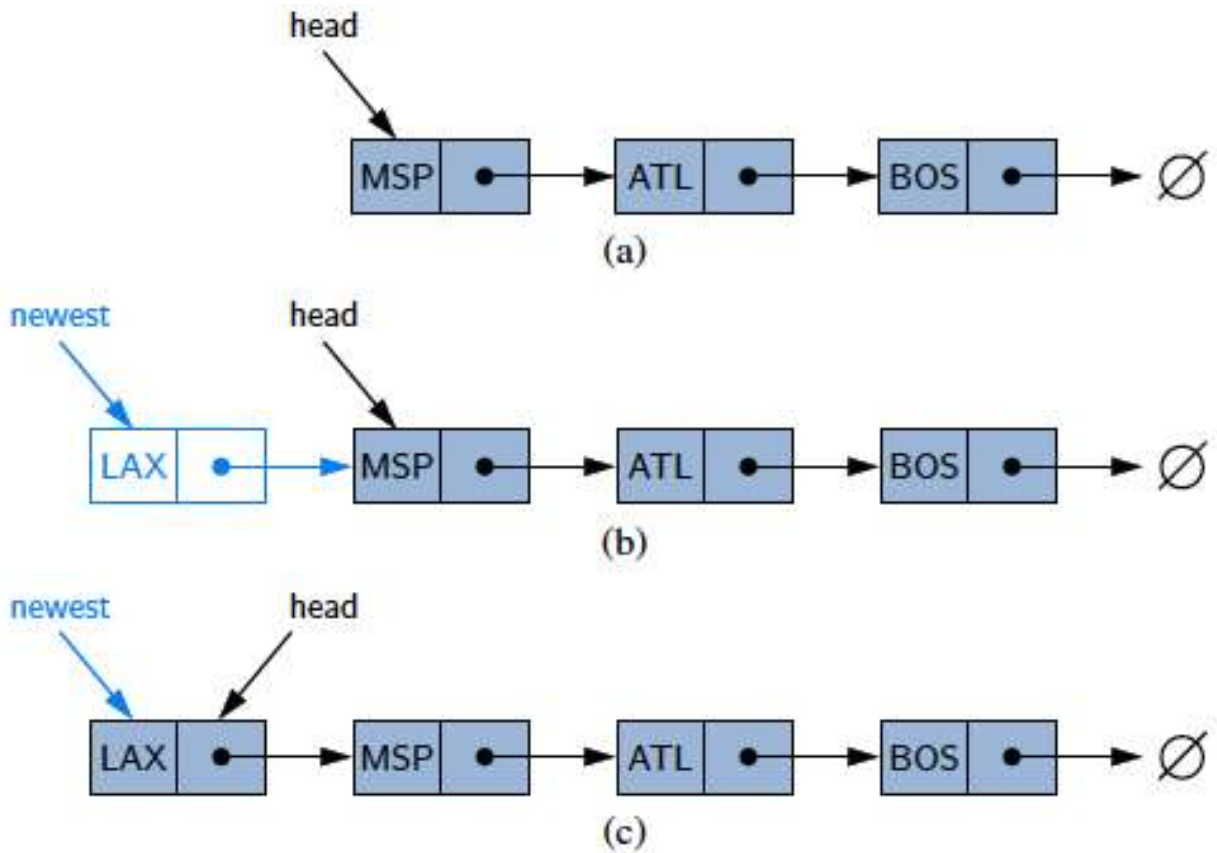
- Singly linked list can be defined as the collection of ordered set of elements. The number of elements may vary according to need of the program. A node in the singly linked list consist of two parts: data part and link part. Data part of the node stores actual information that is to be represented by the node while the link part of the node stores the address of its immediate successor
- One way chain or singly linked list can be traversed only in one direction. In other words, we can say that each node contains only next pointer, therefore we can not traverse the list in the reverse direction.

Linked List Operations
Traverse

- Description: This algorithm will traverse the linked from the first node until finishing it. It will print the data inside it on the screen.

```
ALGORITHM Traverse(head)
        // Initialize a traversing pointer to the head
        c = head
        // While we didn't reach the null. The list is not finished yet
        WHILE c != NULL:
                PRINT c -> data
                // Update the traversing pointer to the next node
                // its address is in link part of the current node
                c = c -> link
```

Linked List Operations
Insert at the beginning



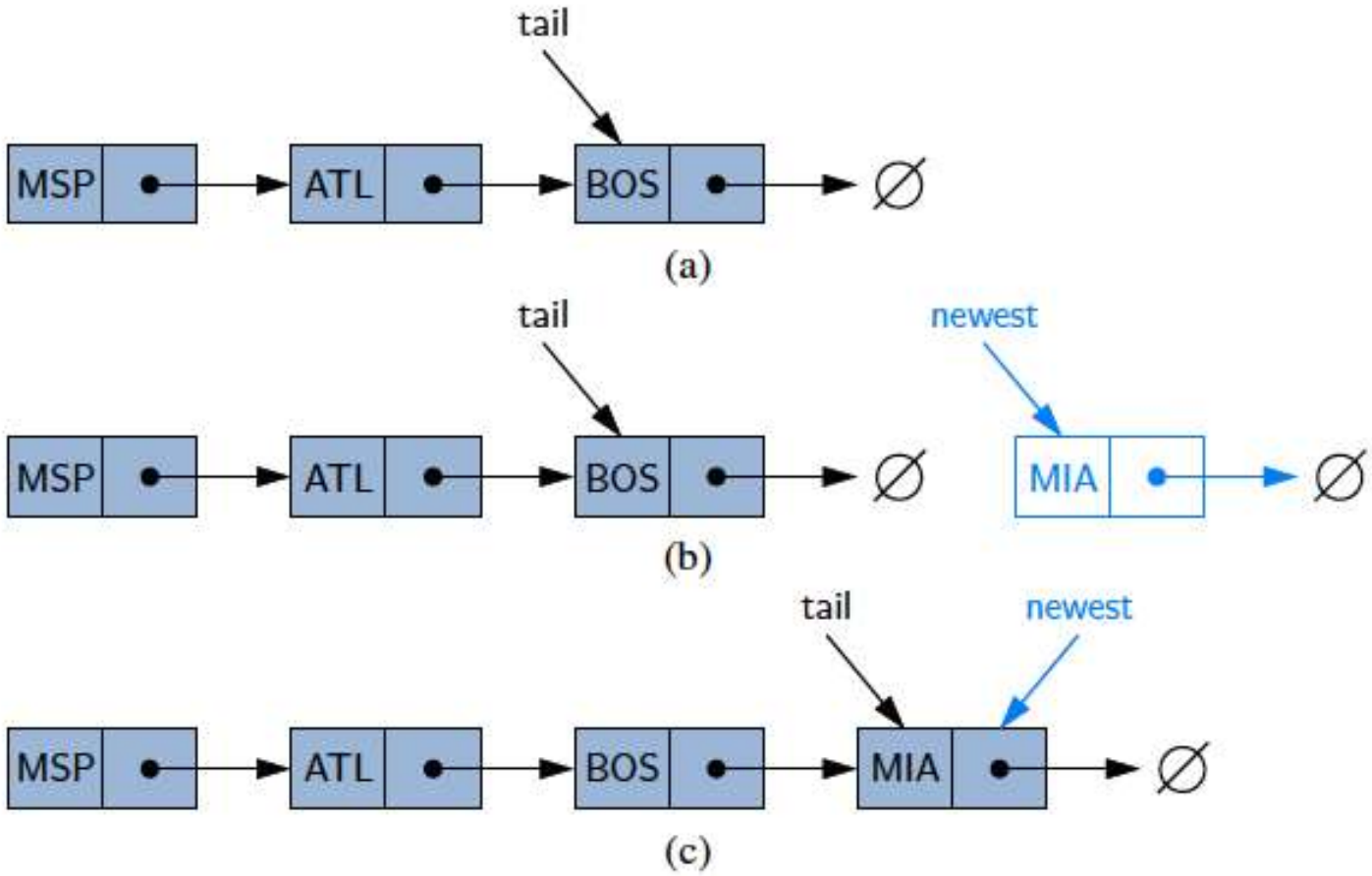
Linked List Operations
Insert at the beginning

- Description: This algorithm will insert one node at the beginning of the linked list. This may change the content of the given head.

ALGORITHM addFirst(e):

```
// create new node instance and store reference to element e
newest = Node(e)
newest.next = head // set new node's next to reference the old head
node head = newest // set variable head to reference the new node
size = size +1 // increment the node count
```

Linked List Operations
Insert at the End



Linked List Operations
Insert at the End

- Description: This algorithm will insert one node at the end of the linked list. This may change the content of the given head.

ALGORITHM addLast(e):

newest = Node(e) // create new node instance and store reference to element e

newest.next = null // set new node's next to reference the null object

tail.next = newest // Make old tail node point to new node

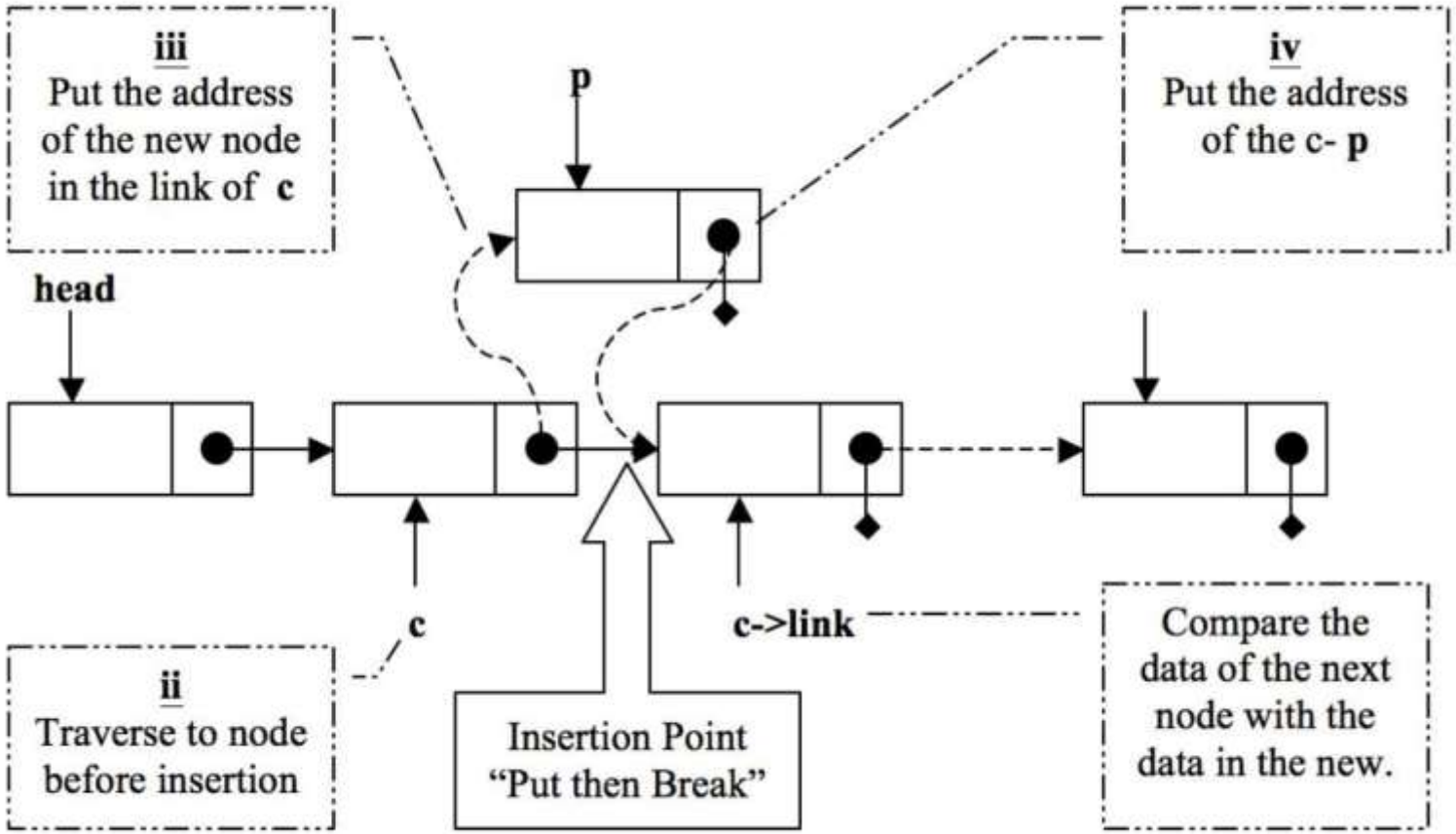
tail = newest // set variable tail to reference the new node

size = size +1 // increment the node count

Linked List Operations
Insert in the Middle

- This algorithm will insert one node in the middle of the linked list.
- The insertion point depends on the relation of the data in the new node with the other data in the list.
- It may be used in sorted insert.
- This algorithm will include the previous addFirst and Traverse.

Linked List Operations
Insert in the Middle



Linked List Operations
Insert in the Middle

```
InsertMiddle(head, newest):
        newest=Node( e)
        IF head = null then // Check if the list is empty:
                head=newest // If true, put the address of the new node in the head.
        ELSE // If not,
        IF newest->data < head->data // check if the new node must be inserted before the head node.
        addFirst()
        ELSE // If not, Traverse until the last node, or until find the node before the wanted one
                c=head
                WHILE ( c!= NULL)
                        IF (newest->data > c->next->data)
                                c=c->next
                // insertion point. [c->next->data > new Node->data].
                Newest->next=c->next // Put the address of the next node in the next of the new node.
                C->next=newest // Put the address of new node in the link of the node before the insertion point.
```

Linked List Operations
Delete from the Beginning

- Description: This algorithm will delete one node from the beginning of the linked list. This may change the content of the given head.

ALGORITHM removeFirst():

```
// Check if the list is empty or not,

IF head = NULL then
        the list is empty

head = head.next //make head point to next node (or null)

size = size – 1 //decrement the node count
```

Linked List Operations
Delete from the End

- Description: This algorithm will delete one node from the end of the linked list. This may change the content of the given head.

Algorithm removeLast( ):

```
//check if the list is empty
IF head = NULL
        The list is empty
IF head->next = NULL // check if there is only one node in the linked list
        head = NULL
ELSE
        C = head
        // looking 2 nodes ahead to find the end of the list
        While (c -> next -> next != NULL)
                c = c -> next
        //move to last node and set it NULL
        c = c-> next
        c -> next = NULL
```

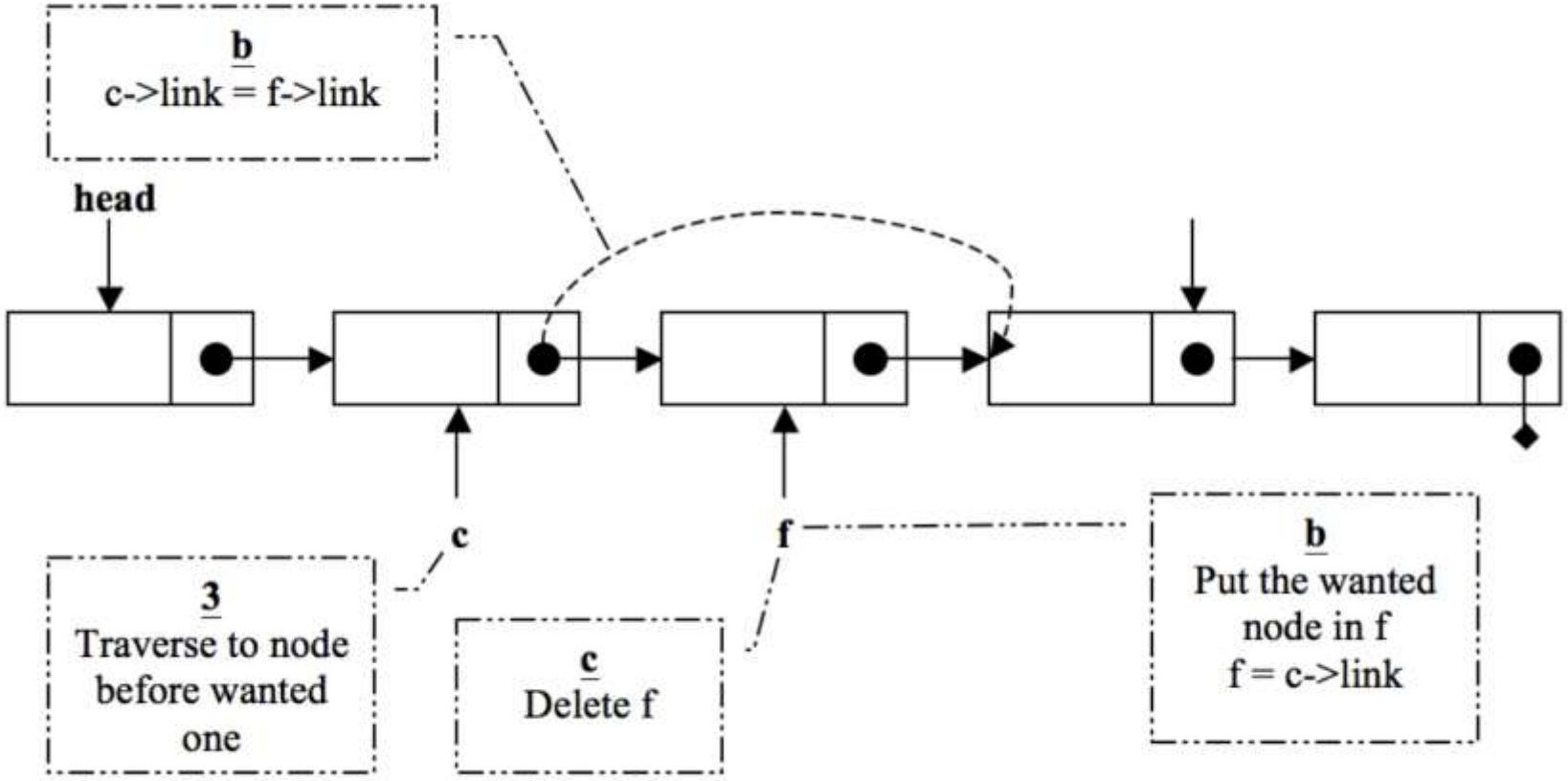
Linked List Operations
Delete from the Middle

- This algorithm will delete one node from the middle of the linked list.

  The choosing of the deleted node depends on a specific condition.

- This algorithm will include the previous removFirst and Traverse.

Linked List Operations
Delete from the Middle



Linked List Operations
Delete from the Middle

```
deleteMiddle(head, key):
        IF head != null then // Check if the list is empty or not. If not, proceed.

                IF head->data==key //Check if the wanted node is not the first one: f true,
                        temp=head //Put the head in temp pointer
                        head=head->next // Change the head to its next node address.

                        temp=null // Make head = NULL.

        ELSE // If not, Traverse until the last node, or until find the node before the wanted one

                 c=head

                WHILE (c->next->data!=key)
                        c=c->next // If not move to the next node. [c=c->next]

                IF c->next!=null // If found,
                        temp=c->next // Put the wanted node in a temp pointer

                        //Put the link of the node before temp = the address of node after 'found'.

                        c->next=temp->next
```

Linked List Operations

- Useful video for all LL operations in Java

  https://www.youtube.com/watch?v=SMIq13-FZSE

Advantages and Disadvantages of Linked Lists

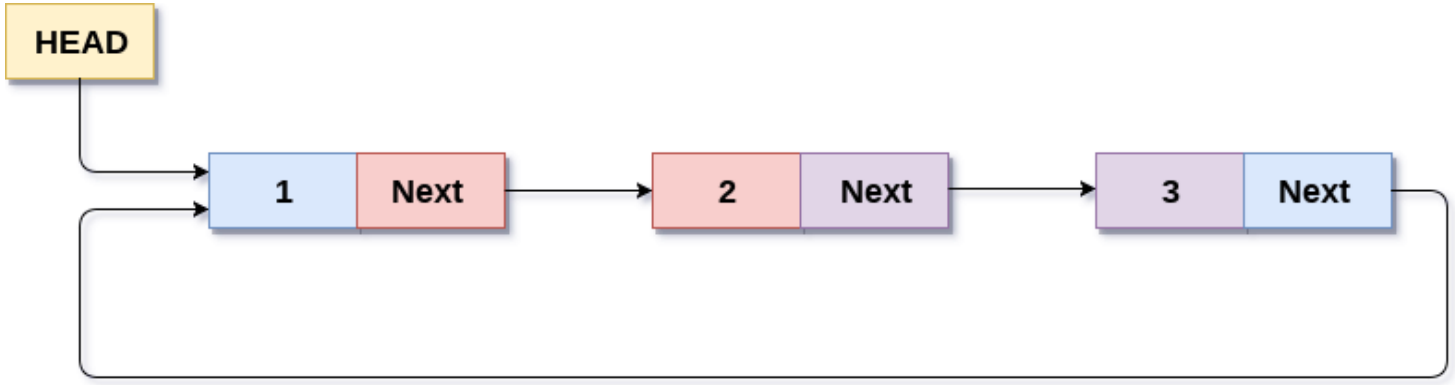
- Advantages of using linked lists:
  - Very flexible.
  - We don’t need to worry about allocating space in advance, can use any free space in memory. We only run out of space when the whole memory is actually full.
  - When doing insertion and deletion no shifting is required. – More efficient for moving large records (leave data in same place in memory, just change some pointers).
- Disadvantages of using linked lists:
  - Wasted space: we store both pointers and data.
  - To access the ith item, we must start at the beginning and follow pointers until we get there. In the worst case, if there are n items in a list and we want the last one, we have to traverse n elements.

Arrays vs. Linked Lists Commands

	In Array	In Linked List
Used counter	`int i`	`Node* c`
Initial address value	`0`	`head`
When to stop looping	`i = = size`	`c = = NULL`
Updating counter	`i = i+1`	`c = c->link`
Content of current element	`A[i]`	`c->data`
First element condition	`i = = 0`	`c = = head`
Last element condition	`i = = size-1`	`c->link = = NULL`
Element before the last	`i = = size-2`	`c->link->link = = NULL`
When its empty	`count = = 0`	`c = = NULL`
When its full	`count = = size`	`Never`
Next element	`A[i+1]`	`c->link`
Next of the next element	`A[i+2]`	`c->link->link`

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.

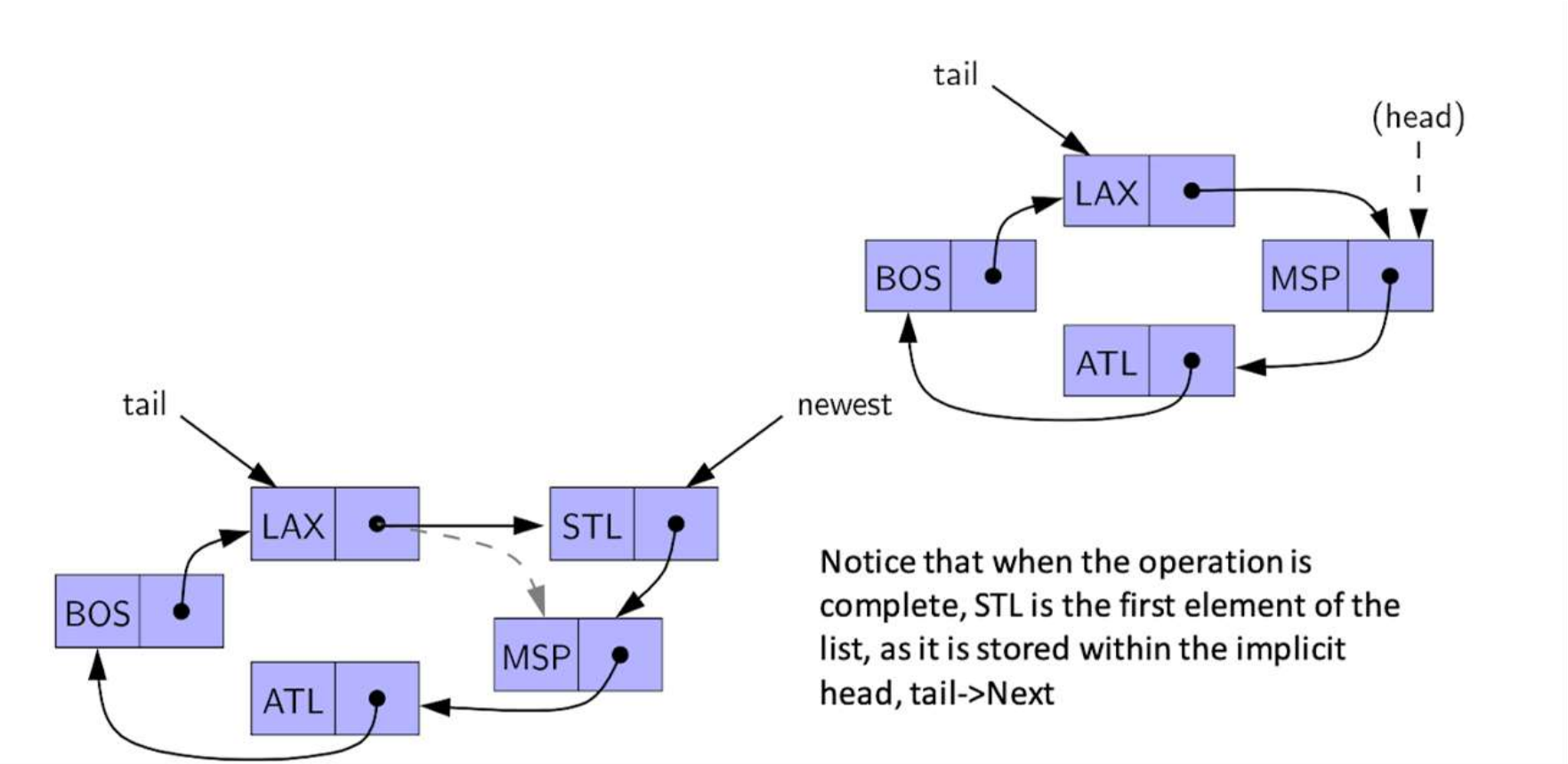
Circular Linked Lists Operations Traverse

- 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



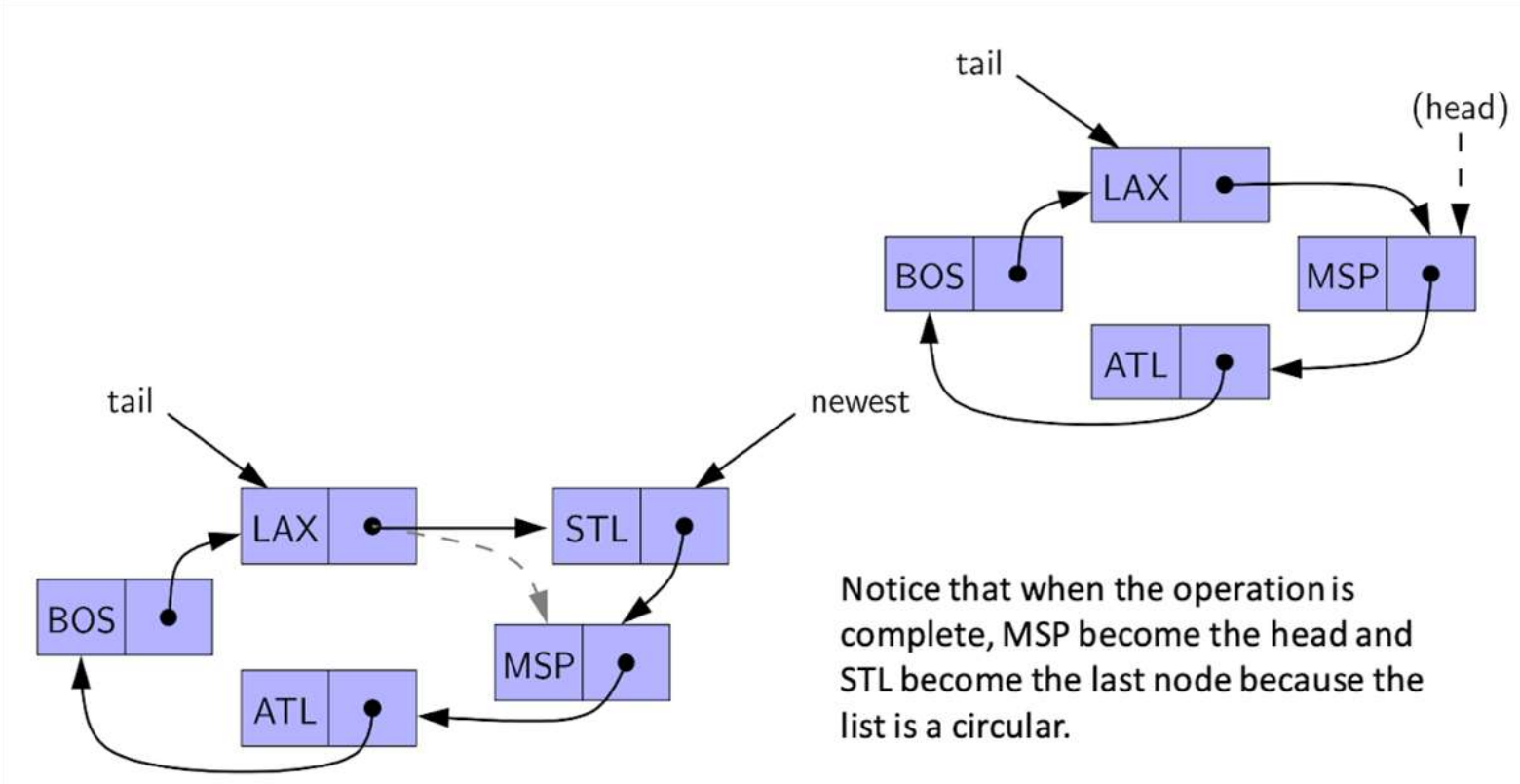
Circular Linked Lists Operations
Insert at the beginning

- 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



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):

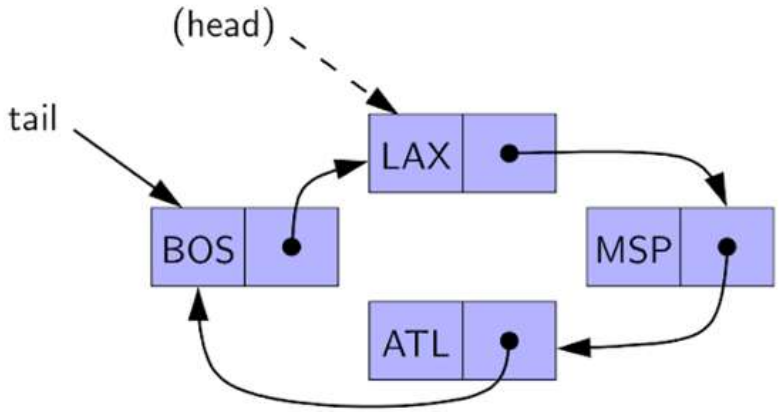
```
addFirst( e)// add new node to the beginning of the list
tail=tail->next //advance the tail to the next node
```

Circular Linked Lists Operations
Delete from the Beginning

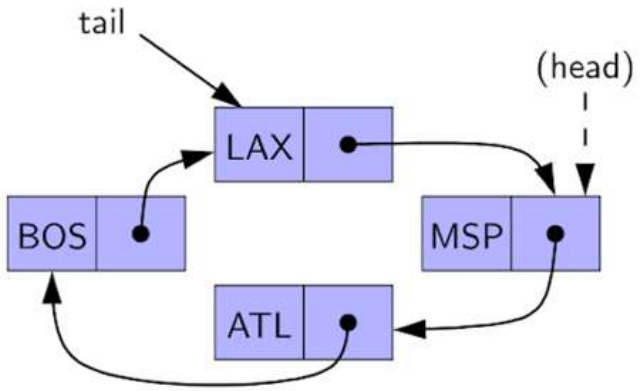
- removing the first node from a circularly linked list can be accomplished by simply updating the next field of the tail node to bypass the implicit head.

```
removeFirst():
        head=tail->next// store the head
        IF head==tail // only one node lift
                tail=null //remove last node from 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

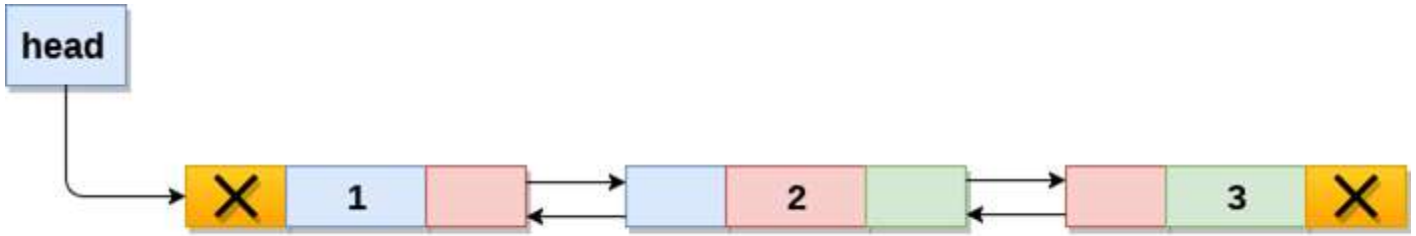
Circular Linked Lists Operations
Rotate

- 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
```

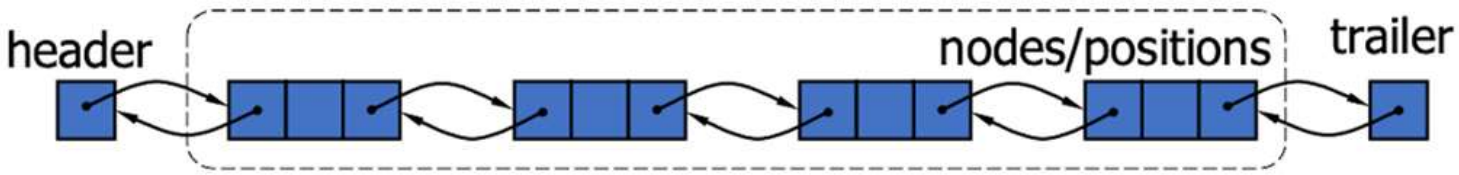
Doubly linked list

- 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

- 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;

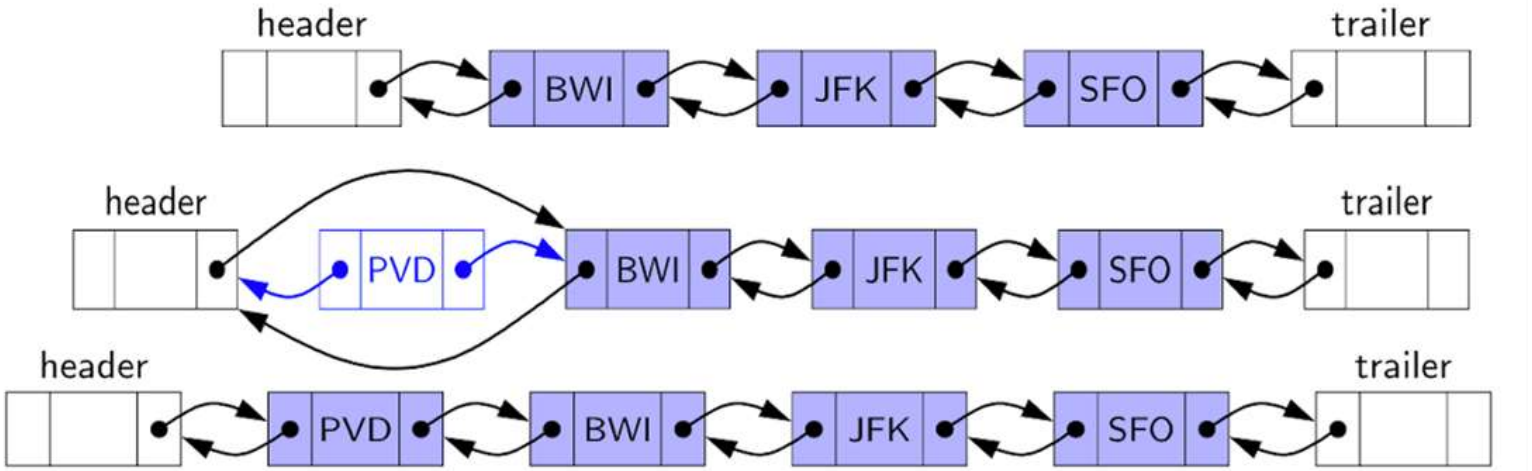


Doubly linked list

- 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.

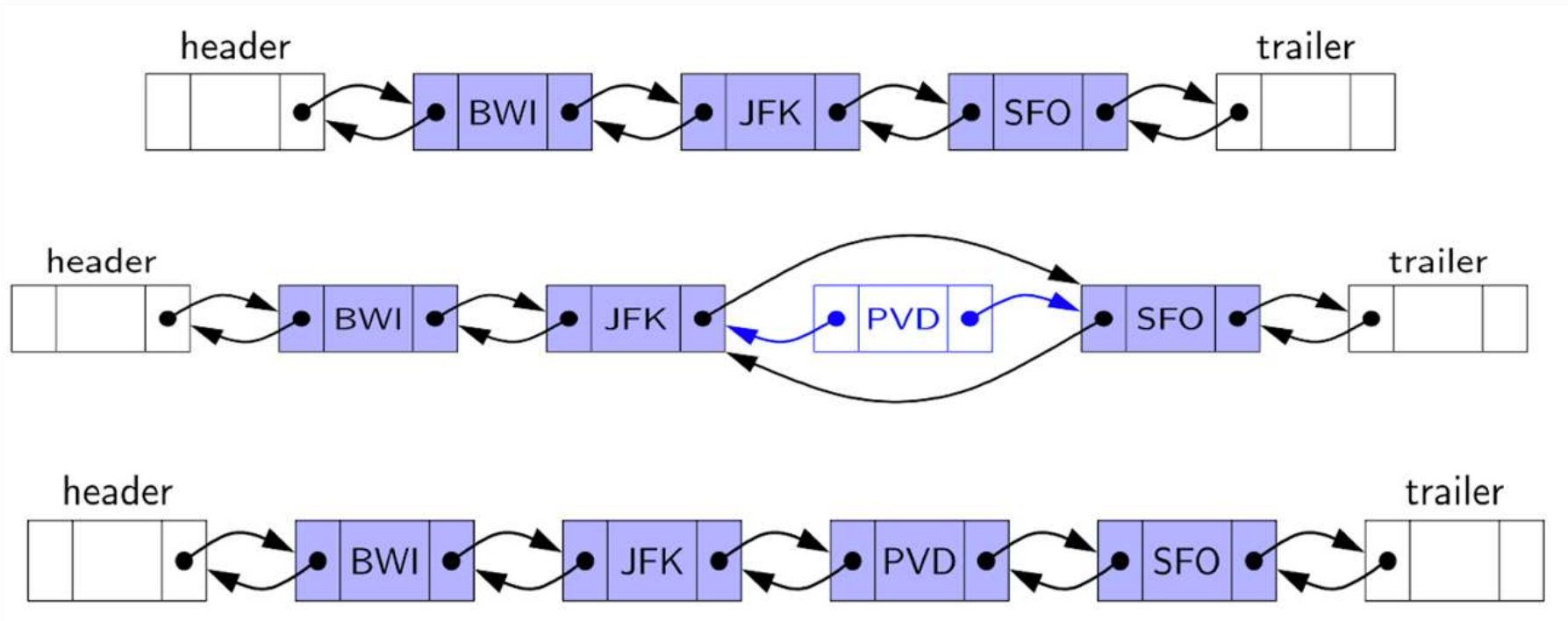
Doubly Linked List operations
Insert

- 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.

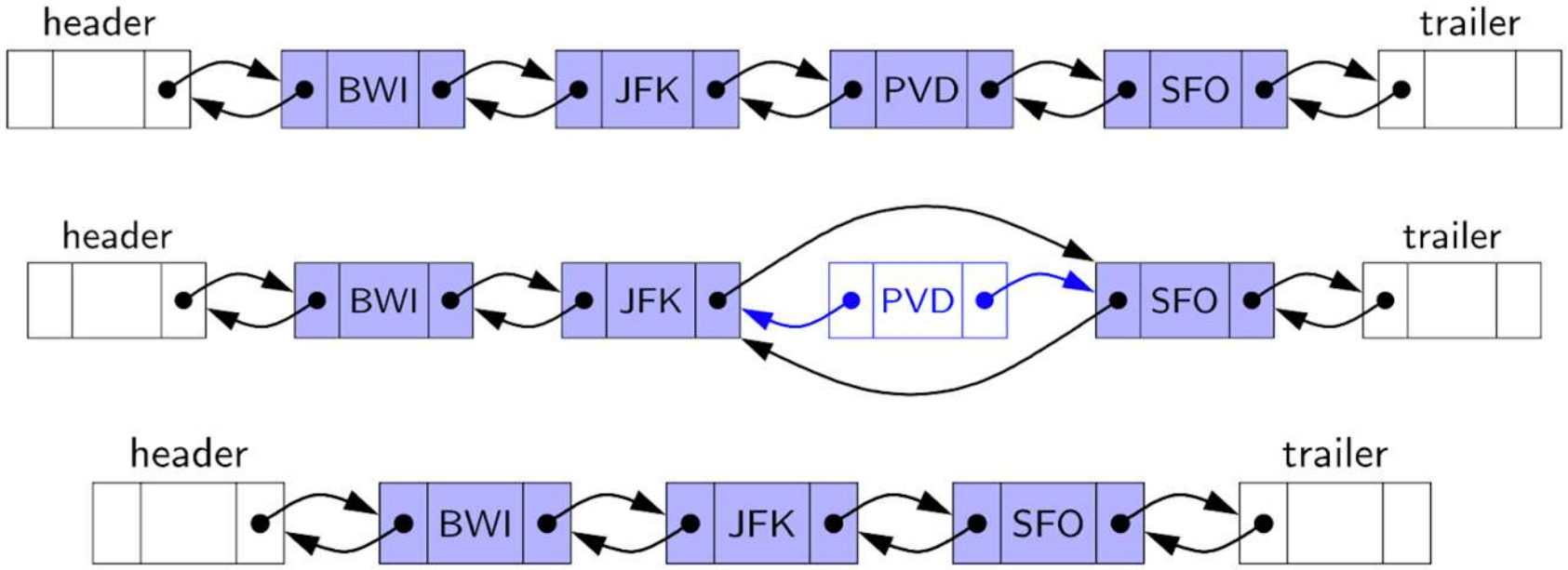
addBefore(p,e):

```
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
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