BITS Pilani
Hyderabad Campus

Data Structures and Algorithms Design (DSECLZG519)

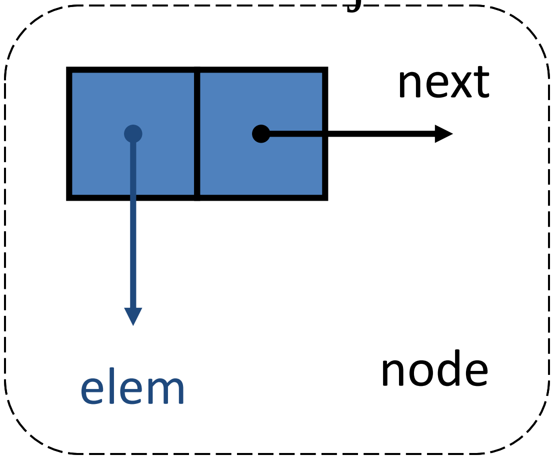
Febin.A.Vahab
Asst.Professor(Offcampus)
BITS Pilani,Bangalore

CONTACT SESSION 4 -PLAN

Contact Sessions(#)	List of Topic Title	Text/Ref Book/external resource
4	Lists- Notion of position in lists, List ADT and Implementation. Sets- Set ADT and Implementation	T1: 2.2, 4.2

SINGLY LINKED LIST

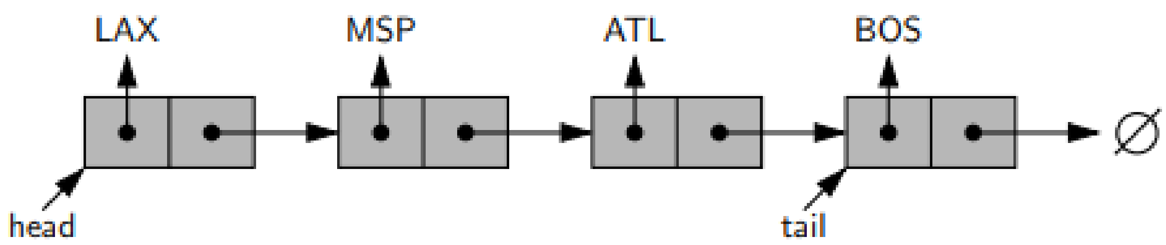
- A singly linked list is a concrete data structure consisting of a sequence of nodes
- Each node stores reference to an object that is a reference to an
  - element
  - link to the next node



- The first and last node of a linked list usually are called the ***head*** and ***tail*** of the list, respectively

Singly Linked List

- Moving from one node to another by following a next reference is known as ***link hopping*** or ***pointer hopping- Traversing the list***
- The order of elements is determined by the chain of ***next*** links going from each node to its successor in the list
- We do not keep track of any index numbers for the nodes in a linked list. So we cannot tell just by examining a node if it is the second, fifth, or twentieth node in the list

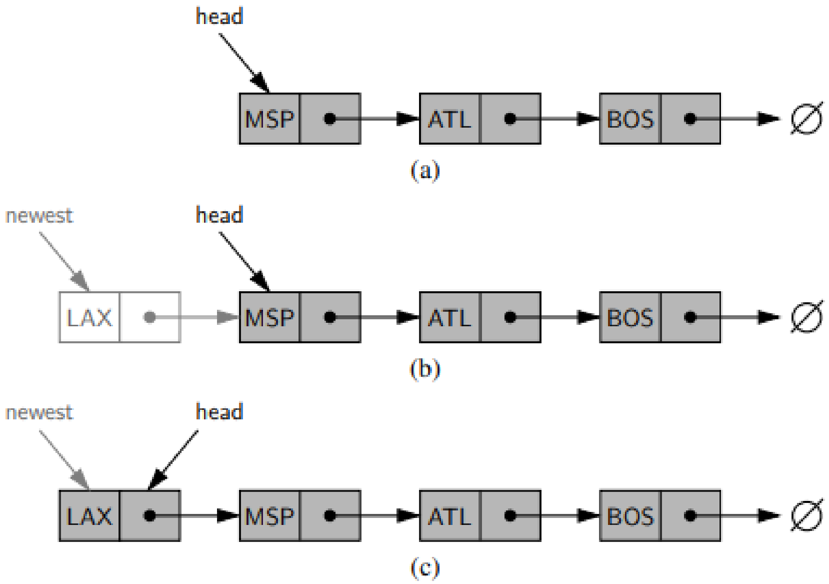


Example of a singly linked list whose elements are strings indicating airport codes

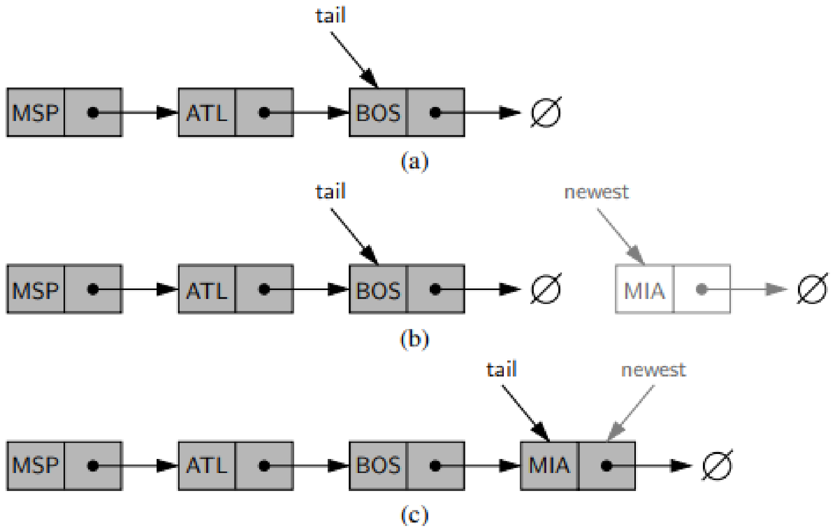
Singly Linked List Implementation

- To implement a singly linked list, we define a Node class

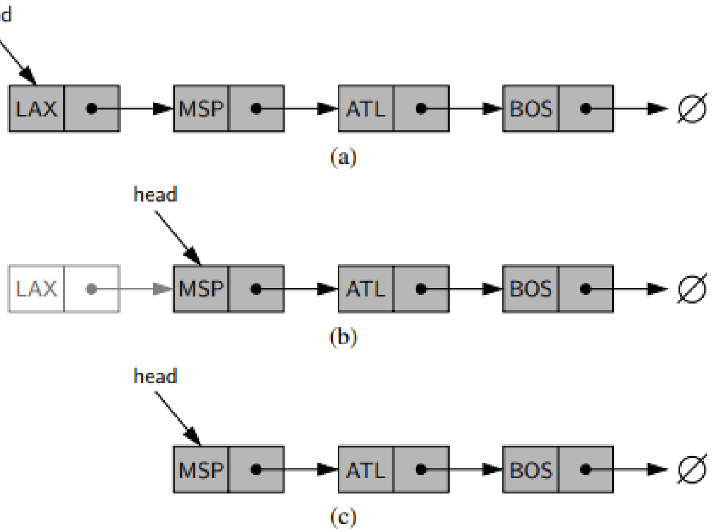
```
class _Node:
  """Lightweight, nonpublic class for storing a singly linked node."""
  __slots__ = '_element', '_next'          # streamline memory usage

  def __init__(self, element, next):       # initialize node's fields
    self._element = element                # reference to user's element
    self._next = next                      # reference to next node
```



Insertion of an element at the head of a singly linked list



Insertion at the tail of a singly linked list



Removal of an element at the head of a singly linked list

Singly Linked List Operations-Node Based

- *Not easy to delete the tail node /Insertbefore operations*
- *Start from the head of the list and search all the way through the list.*
- *Such link hopping operations could take a long time.*

SLL-Node Based-insertBefore

```
insertBefore(n, o):
    if n == first then
        insertFirst(o)
    else
        m = first
        while m.next != n do
            m = m.next
        done
        insertAfter(m, o)
```

SLL-Node Based- Performance

Operation	Worst case Complexity
size, isEmpty	$O(1)$ [1]
first, last, after	$O(1)$ [2]
before	$O(n)$
replaceElement, swapElements	$O(1)$
insertFirst, insertLast	$O(1)$ [2]
insertAfter	$O(1)$
insertBefore	$O(n)$
remove	$O(n)$ [3]

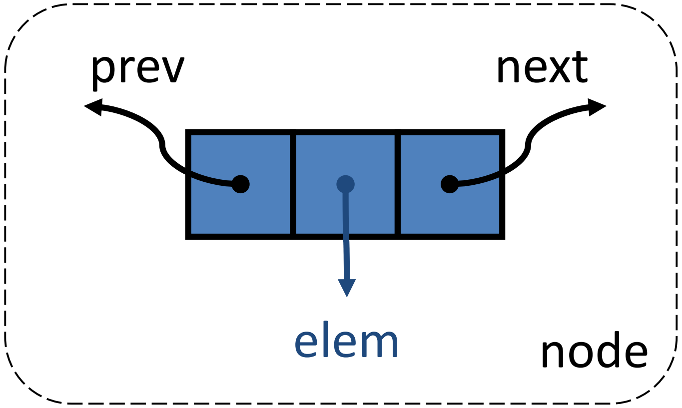
[1] size needs $O(n)$ if we do not store the size on a variable.

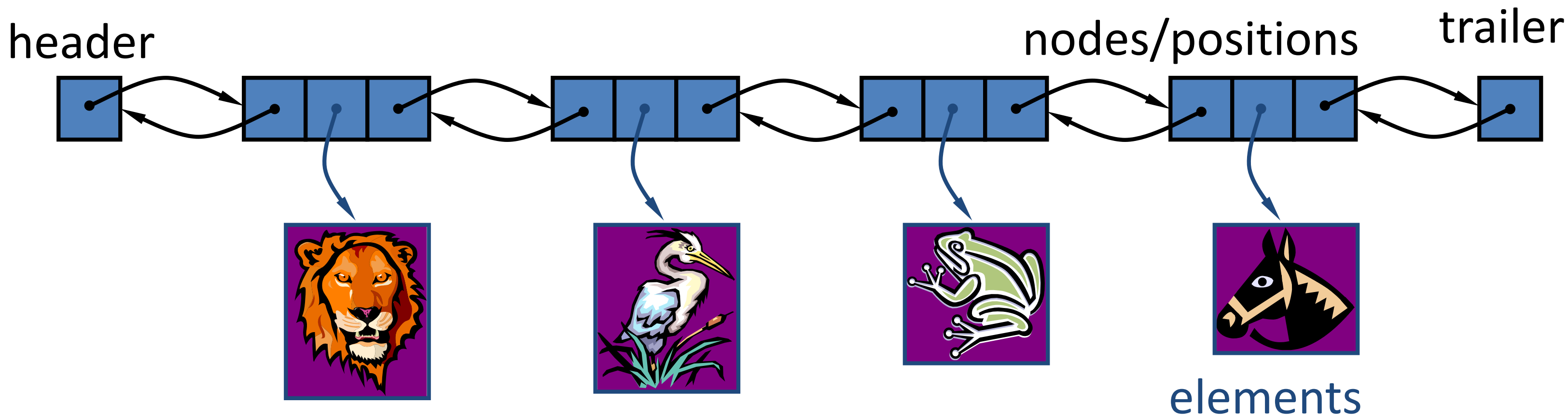
[2] last and insertLast need $O(n)$ if we have no variable *last*.

[3] remove(n) runs in best case in $O(1)$ if $n == first$.

DOUBLY LINKED LIST

- Nodes store:
  - element
  - link to the previous node -*prev*
  - link to the next node-*next*
- Special trailer and header nodes



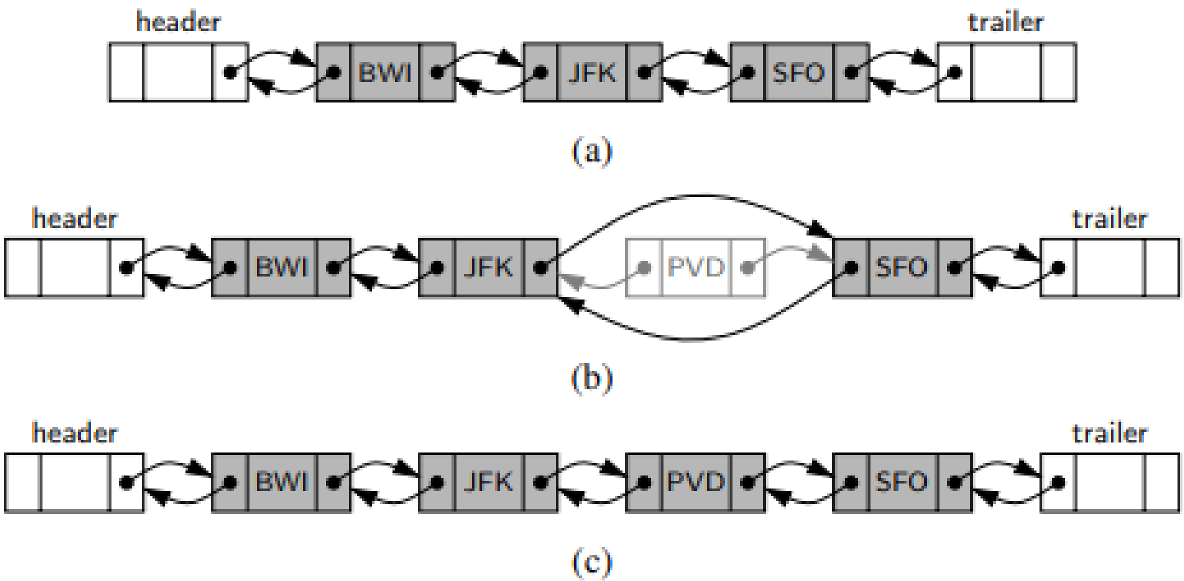


Doubly Linked List Implementation

- To implement a Doubly linked list, we define a Node class

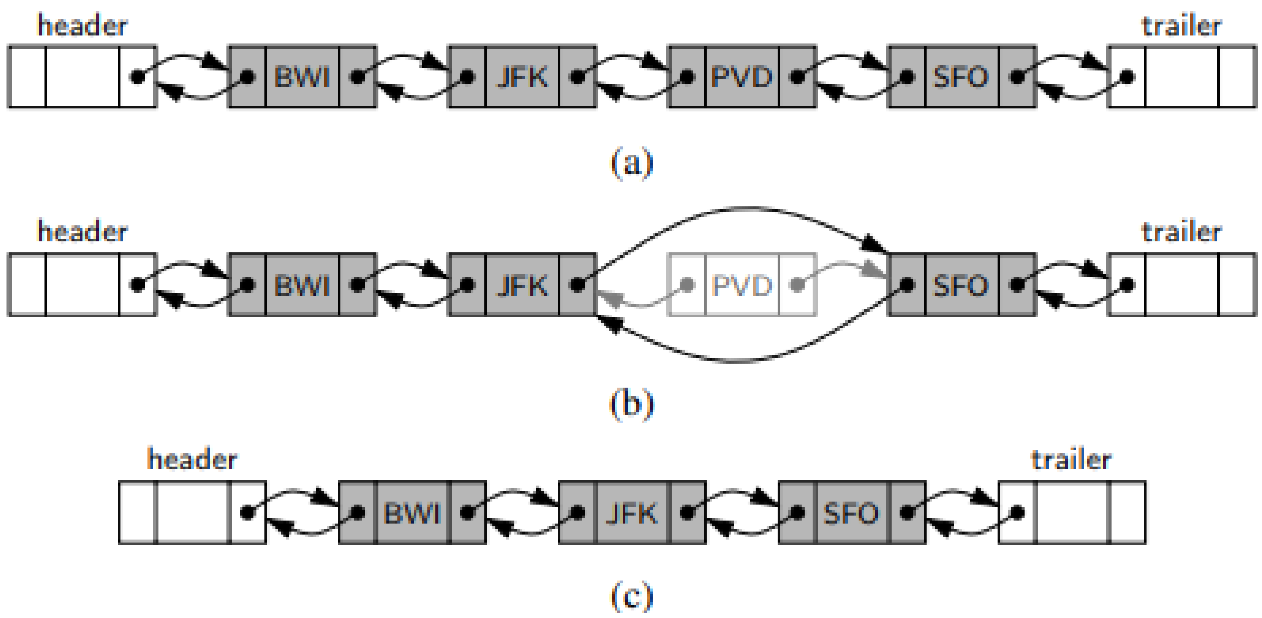
```
class _Node:
  """Lightweight, nonpublic class for storing a doubly linked node."""
  __slots__ = '_element', '_prev', '_next'    # streamline memory

  def __init__(self, element, prev, next):    # initialize node's fields
    self._element = element                   # user's element
    self._prev = prev                         # previous node reference
    self._next = next                         # next node reference
```

Doubly Linked List Implementation



Adding an element to a doubly linked list with header and trailer sentinels: (a) before the operation; (b) after creating the new node; (c) after linking the neighbors to the new node

Doubly Linked List Implementation



Removing the element PVD from a doubly linked list: (a) before the removal; (b) after linking out the old node; (c) after the removal (and garbage collection)

DLL –Node based-Operations

innovate achieve lead

Algorithm removeLast():
- **if** size $= 0$ **then**
  - Indicate an error: the list is empty
- $v \leftarrow$ trailer.getPrev() {last node}
- $u \leftarrow v$.getPrev() {node before the last node}
- trailer.setPrev(u)
- u.setNext(trailer)
- v.setPrev(**null**)
- v.setNext(**null**)
- size $=$ size $- 1$

Algorithm addAfter(v,z):
- $w \leftarrow v$.getNext() {node after v}
- z.setPrev(v) {link z to its predecessor, v}
- z.setNext(w) {link z to its successor, w}
- w.setPrev(z) {link w to its new predecessor, z}
- v.setNext(z) {link v to its new successor, z}
- size $\leftarrow$ size $+ 1$

Algorithm addFirst(v):
- $w \leftarrow$ header.getNext() {first node}
- v.setNext(w)
- v.setPrev(header)
- w.setPrev(v)
- header.setNext(v)
- size $=$ size $+ 1$

DLL –Node Based-Performance

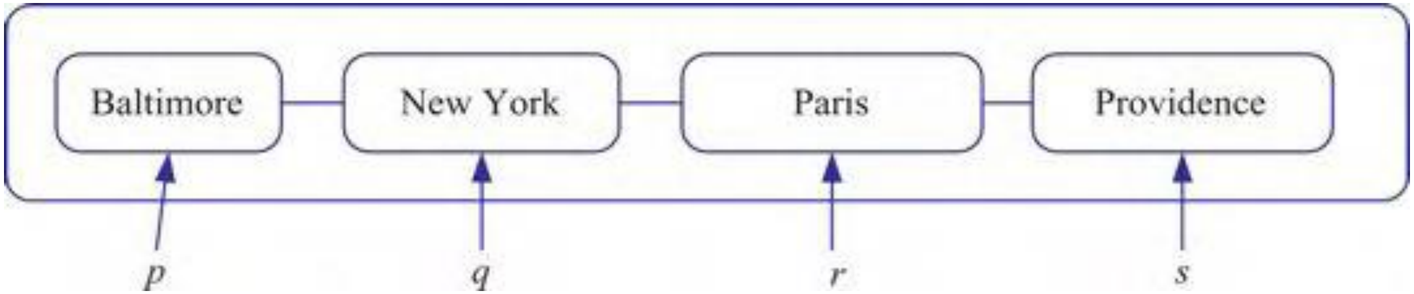
Operation	Worst case Complexity
size, isEmpty	$O(1)$
first, last, after	$O(1)$
before	$O(1)$
replaceElement, swapElements	$O(1)$
insertFirst, insertLast	$O(1)$
insertAfter	$O(1)$
insertBefore	$O(1)$
remove	$O(1)$

LIST -ADT

- One of the great benefits of a linked list structure is that it is possible to perform O(1)-time insertions and deletions at arbitrary positions of the list, as long as we are given a reference to a relevant node of the list.
- Such direct use of nodes would violate the object-oriented design principles of abstraction and encapsulation
- It will be simpler for users of our data structure if they are not bothered with unnecessary details of our implementation
- We ensure that users cannot invalidate the consistency of a list by mismanaging the linking of nodes
- For these reasons, instead of relying directly on nodes, we introduce an independent ***position abstraction*** to denote the location of an element within a list, and then a complete positional list ADT that can encapsulate a doubly linked list

POSITION -ADT

- "place" of an element relative to others in the list.
- In this framework, we view a list as a collection of elements that stores each element at a position and that keeps these positions arranged in a linear order



A node list. The positions in the current order are *p*, *q*, *r*, and *s*.

POSITION ADT

- The positions are arranged in a linear order
- A position is itself an abstract data type that supports the following simple method
  - **element(): Return the element stored at this position**
- A position is always defined relatively
- A position p will always be "after" some position q and "before" some position s (unless p is the first or last position).
- The position *p* does not change even if we replace or swap the element *e* stored at *p* with another element
- They are viewed internally by the linked list as nodes, but from the outside, they are viewed only as positions
- We can give each node *v* instance variables prev and next that respectively refer to the predecessor and successor nodes of *v*

LIST - ADT

- Container of elements that stores each element at a position
- Using the concept of position to encapsulate the idea of "node" in a list, the following methods can be defines for a list
- ***L.first ( ):***Return the position of the first element of S; an error occurs if S is empty.
- ***L.last()*** :Return the position of the last element of S; an error occurs if S is empty.
- ***L.isFirst(p)*** :Return a Boolean value indicating whether the given position is the first one in the list.
- ***L.islast (p)***
- ***L.before(p)*** :Return the position of the element of S preceding the one at position p; an error occurs if p is the first position.
- ***L.after (p) ,L.size(), L.isEmpty()***

LIST – ADT-Update methods

- ***L.replace(p, e)*** : Replace the element at position p with e, returning the element formerly at position p.
- ***L.swap (p , q)*** : Swap the elements stored at positions p and q, so that the element that is at position p moves to position q and the element that is at position q moves to position p.
- ***L.add_First(e)*** : Insert a new element e into S as the first element.
- ***L.add_last(e)*** : Insert a new element e into S as the last element.
- ***L.add_before (p, e)*** : Insert a new element e into S before position p in S; an error occurs if p is the first position.
- ***L.add_after(p, e)*** : Insert a new element e into S after position p in S; an error occurs if p is the last position.
- ***L.delete(p)*** **:** Remove from S the element at position p.

LIST ADT Operation-Position Based

.

Operation	Return Value	L
L.add_last(8)	p	8_p
L.first( )	p	8_p
L.add_after(p, 5)	q	$8_p, 5_q$
L.before(q)	p	$8_p, 5_q$
L.add_before(q, 3)	r	$8_p, 3_r, 5_q$
r.element( )	3	$8_p, 3_r, 5_q$
L.after(p)	r	$8_p, 3_r, 5_q$
L.before(p)	None	$8_p, 3_r, 5_q$
L.add_first(9)	s	$9_s, 8_p, 3_r, 5_q$
L.delete(L.last( ))	5	$9_s, 8_p, 3_r$
L.replace(p, 7)	8	$9_s, 7_p, 3_r$

LIST – ADT: Linked List Implementation

LIST - ADT-Doubly linked list Implementation

- The nodes of the linked list implement the position ADT, defining a method element ( ) , which returns the element stored at the node.
- The nodes themselves act as positions.

LIST - ADT-Doubly linked list Implementation

- **INSERTION**

  Algorithm *add_after(p,e):*

//Inserting an element e after a position p in a linked list.

Create a new node *v*

v.element ← *e*

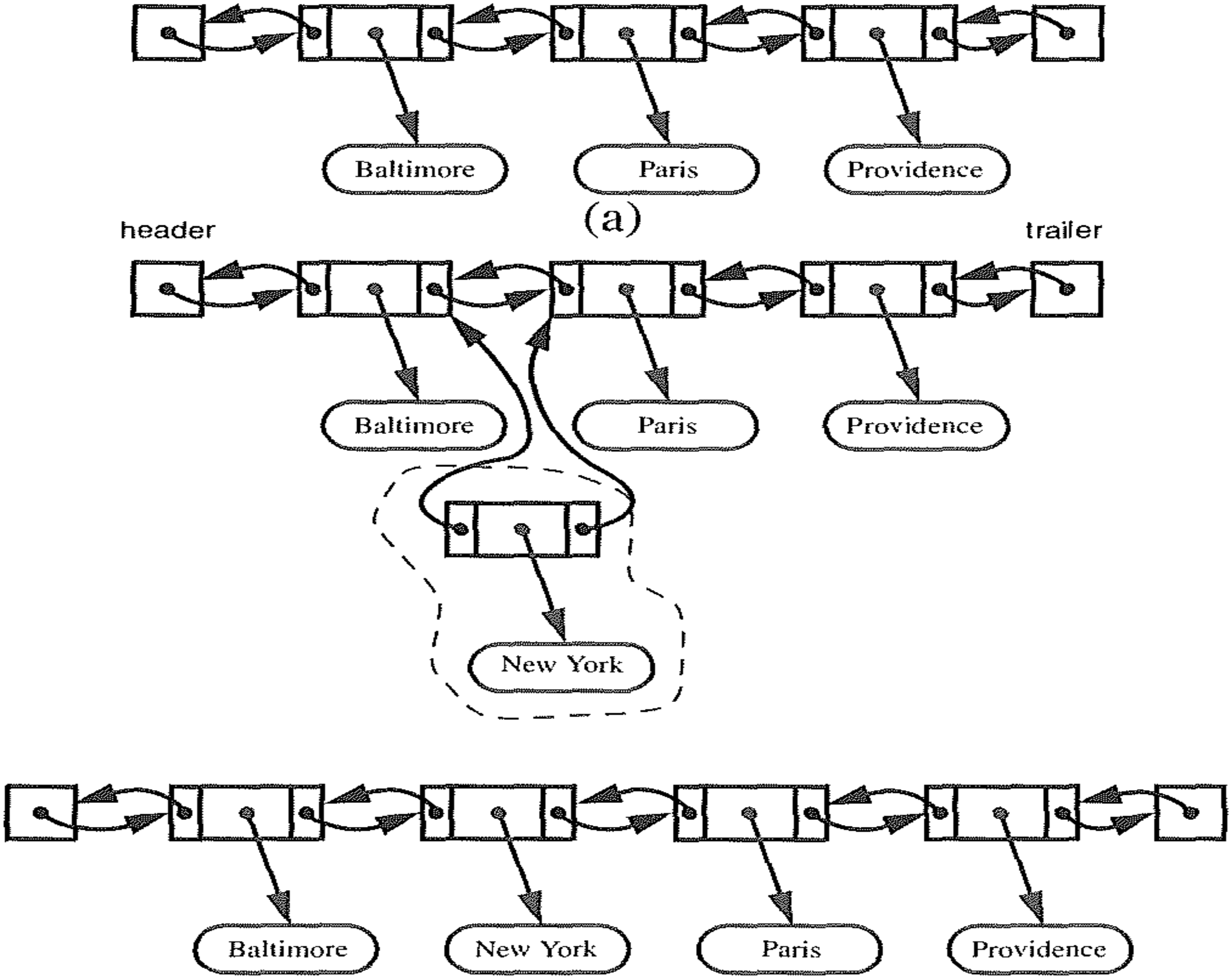
v.prev← *p* {link *v* to its predecessor}

v.next ← *p.next* {link *v* to its successor}

(p.next).prev ←v {link *p*'s old successor to *v*}

p.next ←*v* {link *p* to its new successor, *v*}

return *v* {the position for the element *e*}


Baltimore
Paris
Providence
(a)
header
trailer
Baltimore
Paris
Providence
New York
Baltimore
New York
Paris
Providence

LIST - ADT-Doubly linked list Implementation

- **DELETION**

  Algorithm delete(*p*):
  t ← *p*.element {a temporary variable to hold the return value}
  (p.prev).next ←p.next {linking out *p*}
  (p.next).prev ←p.prev
  p.prev←**nul** {invalidating the position *p*}
  p.next←**null**
  return *t*

- link the two neighbours of p to refer to one another as new neighbours-linking out p.

LIST - ADT linked list Implementation

- In the implementation of the List ADT by means of a linked list
  - All the operations of the List ADT run in $O(1)$ time
  - Operation element() of the Position ADT runs in $O(1)$ time

VECTOR-ADT

A Vector stores a list of elements:

- Access via rank/index.

Accessor methods:
  elementAtRank(r),
Update methods:
  replaceAtRank(r, o),
  insertAtRank(r, o),
  removeAtRank(r)
Generic methods:
  size(), isEmtpy()

Here r is of type integer, n, m are nodes, o is an object (data).

VECTOR-ADT

Method	Time
size()	$O(1)$
isEmpty()	$O(1)$
elemAtRank(r)	$O(1)$
replaceAtRank(r, e)	$O(1)$
insertAtRank(r, e)	$O(n)$
removeAtRank(r)	$O(n)$

Iterators ADT

- Iterator allows to traverse the elements in a list or set.
- Iterator ADT provides the following methods:
  - object(): returns the current object
  - hasNext(): indicates whether there are more elements
  - nextObject(): goes to the next object and returns it

THANK YOU!

BITS Pilani
Hyderabad Campus