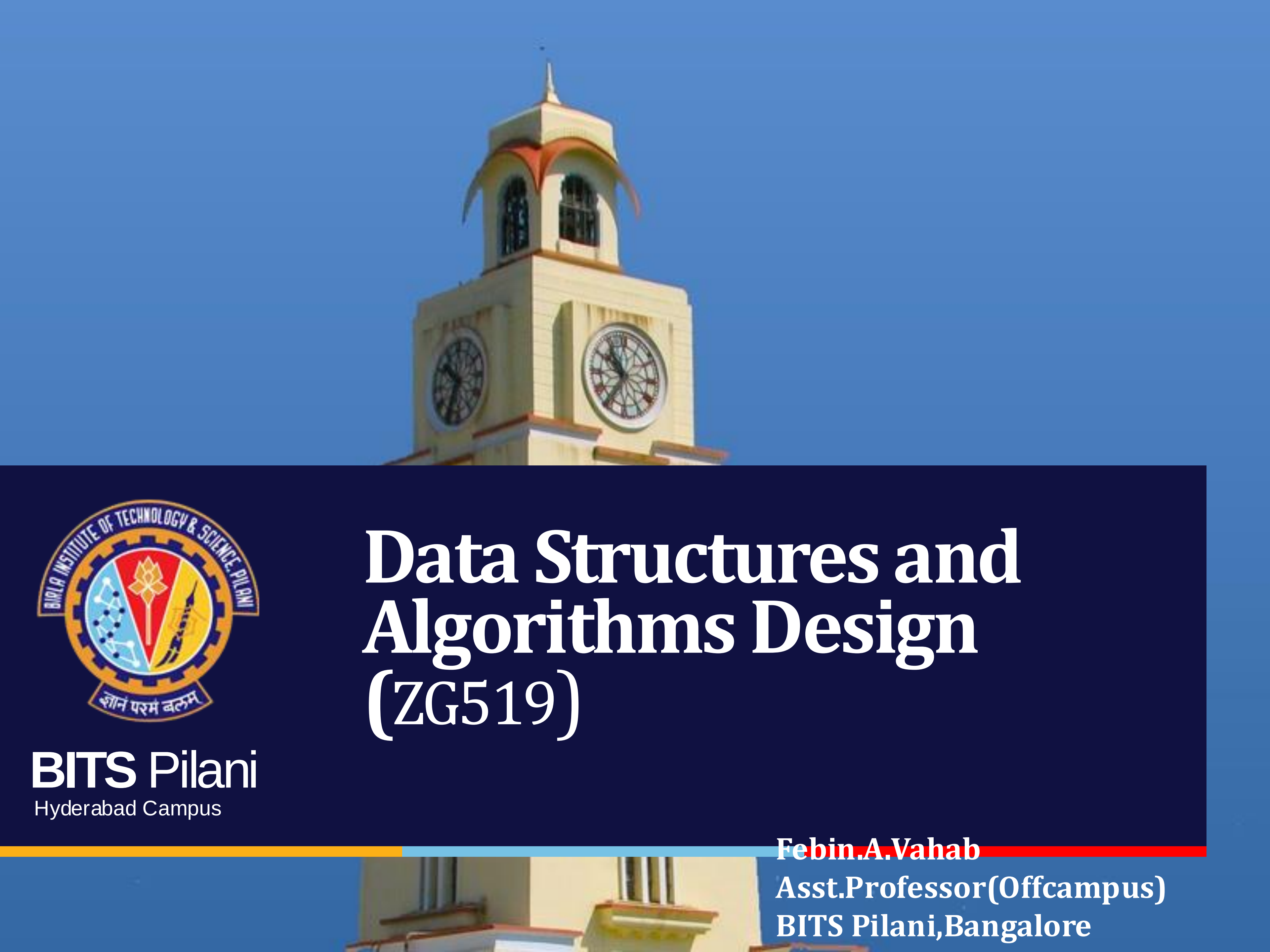


BITS Pilani
Hyderabad Campus

Data Structures and Algorithms Design (ZG519)

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

A + B * C

A B C * +

+ A * B C

(

)

O(n)

A + B * C

	Stack	Postfix
A		A
+	+	A
B		A B
*	+ *	A B
C		A B C
End		A B C *
		A B C * +

$\rightarrow$ (A+B)*(C−D)/E ^ F

Symbol	Action	Stack	Postfix
(	Push	(	
A	Append to the o/p		A
+	Push	(,+	A
B	Append		AB
)	Pop until (		AB+
*	Push	*	AB+
(	Push	* (	AB+
C	Append		AB+C
−	Push	* (−	AB+C

(A + B) * (C - D) / E ^ F

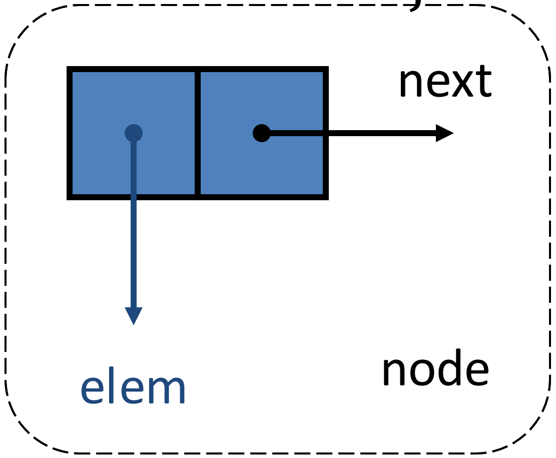
Symbol	Action	Stack	Postfix
-	push	* (-	AB+C
D	Append		AB+CD
)	Pop until (	*	AB+CD-
/	push	* /	AB+CD-
E	Append	* /	AB+CD-E
^	Push	* / ^	AB+CD-E
F	Append	* / ^	AB+CD-EF
End	Pop remaining operators		AB+CD-EF^
			AB+CD-EF^/
			$\Rightarrow$ AB+CD-EF^/*

SESSION 4 -PLAN

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 Trees: Terms and Definition, Tree ADT, Applications Binary Trees : Terms and Definition, Properties, Properties, Representations (Vector Based and Linked), Binary Tree traversal (In Order, Pre Order, Post Order), Applications	T1: 2.2, 4.2

SINGLY LINKED LIST

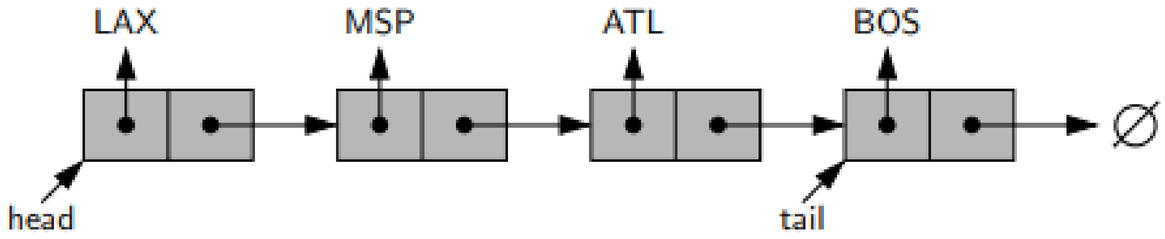
- A singly linked list is a concrete data structure consisting of a sequence of nodes
- Each node stores reference to an object that is a reference to an
  - element
  - link to the next node



- 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

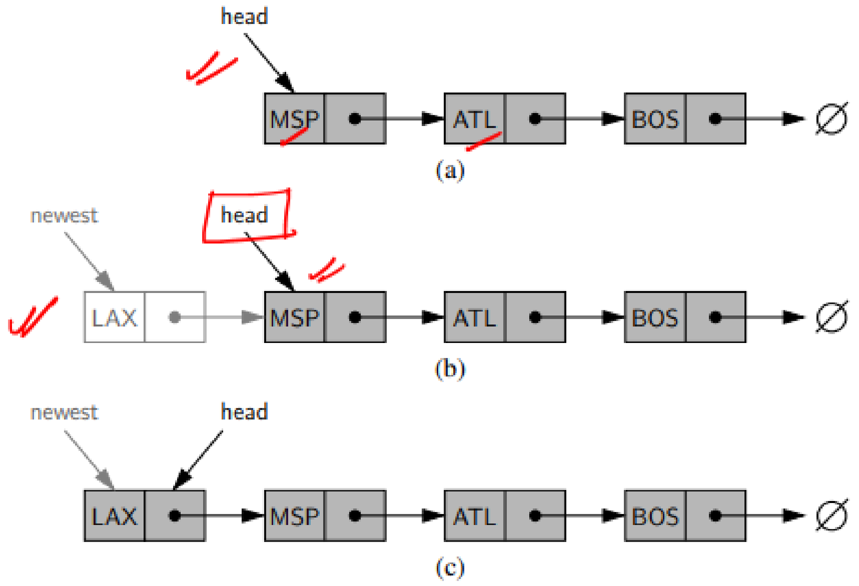


Example of a singly linked list whose elements are strings indicating airport codes

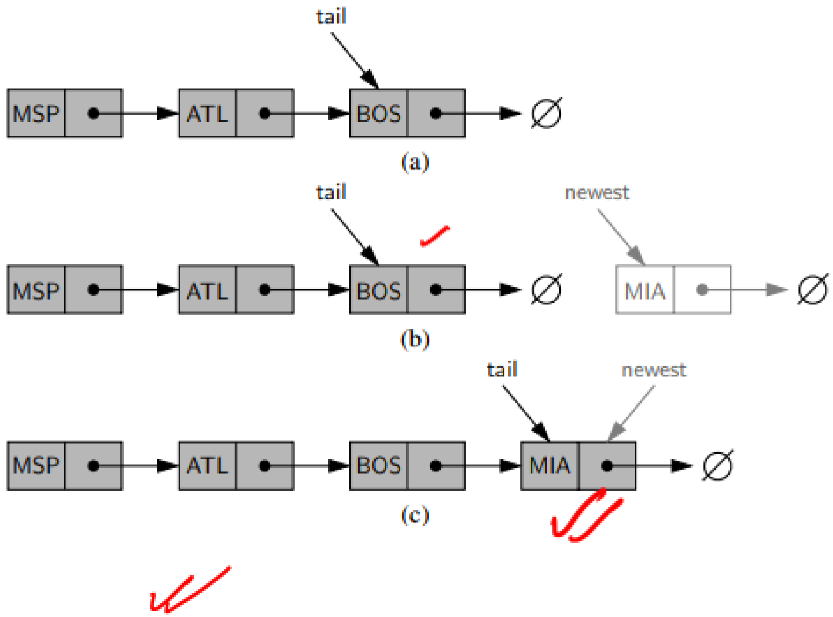
Singly Linked List Implementation

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

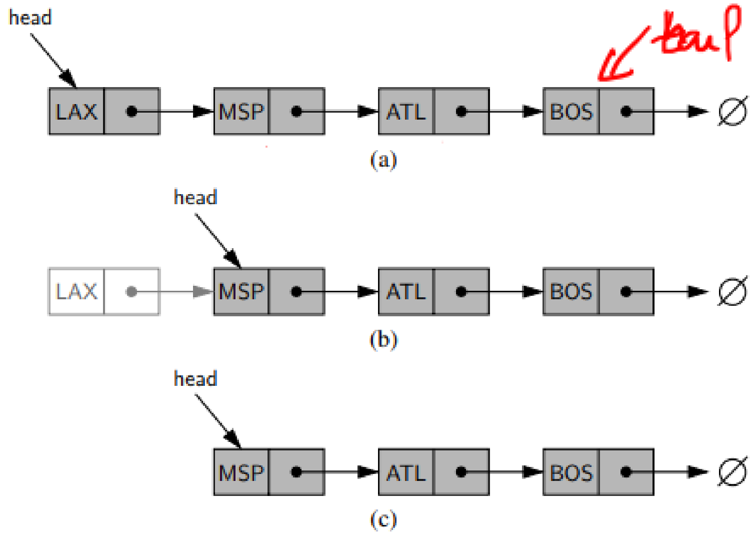
```
class _Node:
  """Lightweight, nonpublic class for storing a singly linked node."""
  __slots__ = '_element', '_next'            # streamline memory usage

  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

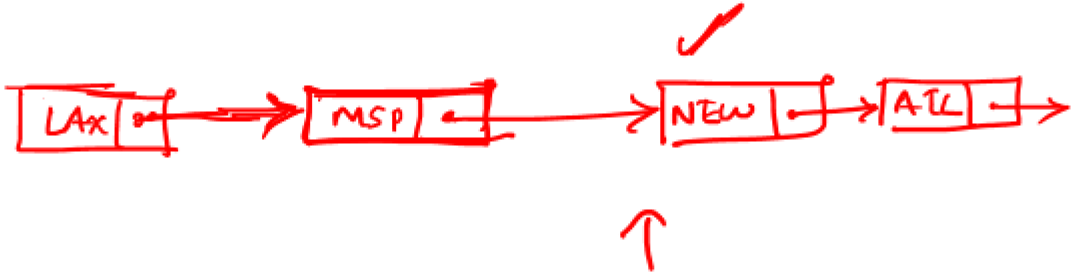
Singly Linked List Operations- Node Based

- *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

ATL , NEW

```
insertBefore($n, o$):
    if $n == first$ then
        insertFirst($o$)
    else
        $m = first$
        while $m.next \;!= n$ do
            $\boxed{m} = m.next$
        done
        insertAfter($m, o$)
```



$O(n)$

SLL-Node Based- Performance

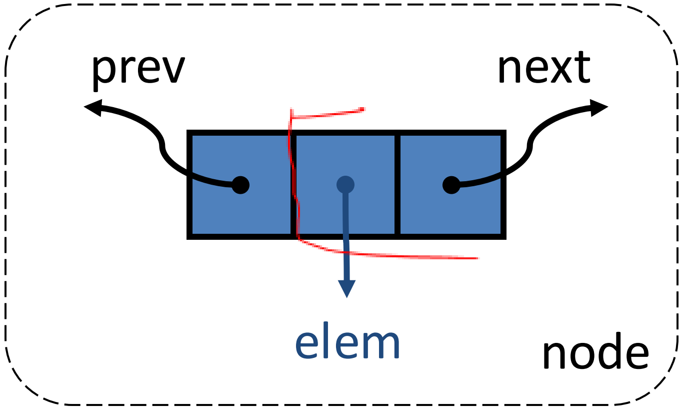
Operation	Worst case Complexity
size, isEmpty	$O(1)$ [1]
first, last, after	$O(1)$ [2]
before	$O(n)$
replaceElement, swapElements	$O(1)$
insertFirst, insertLast	$O(1)$ [2]
insertAfter	$O(1)$
insertBefore	$O(n)$
remove	$O(n)$ [3]

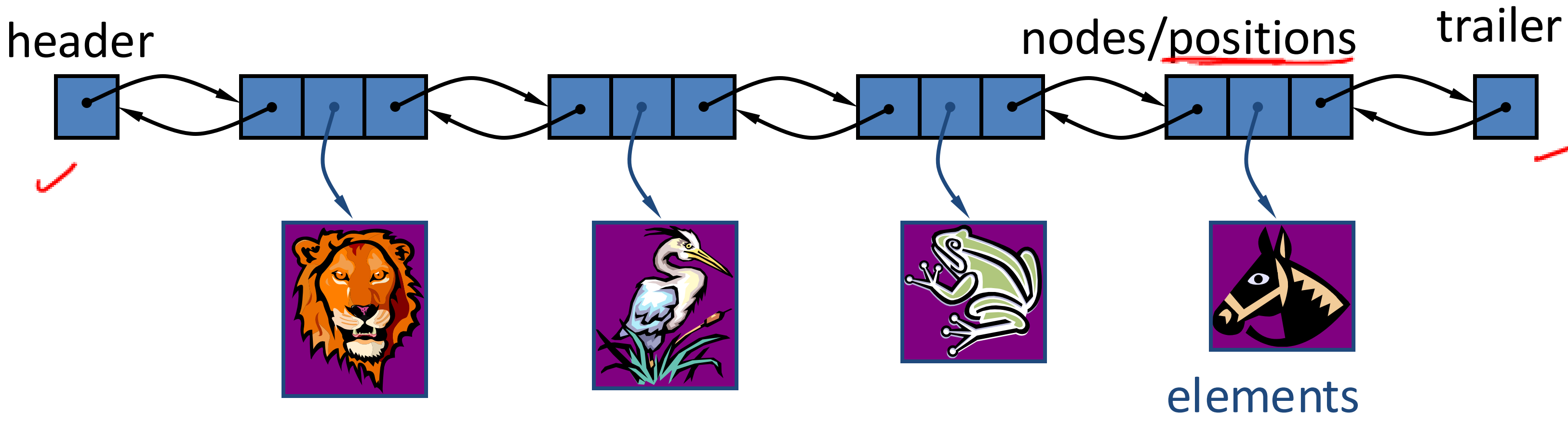
after( )

[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

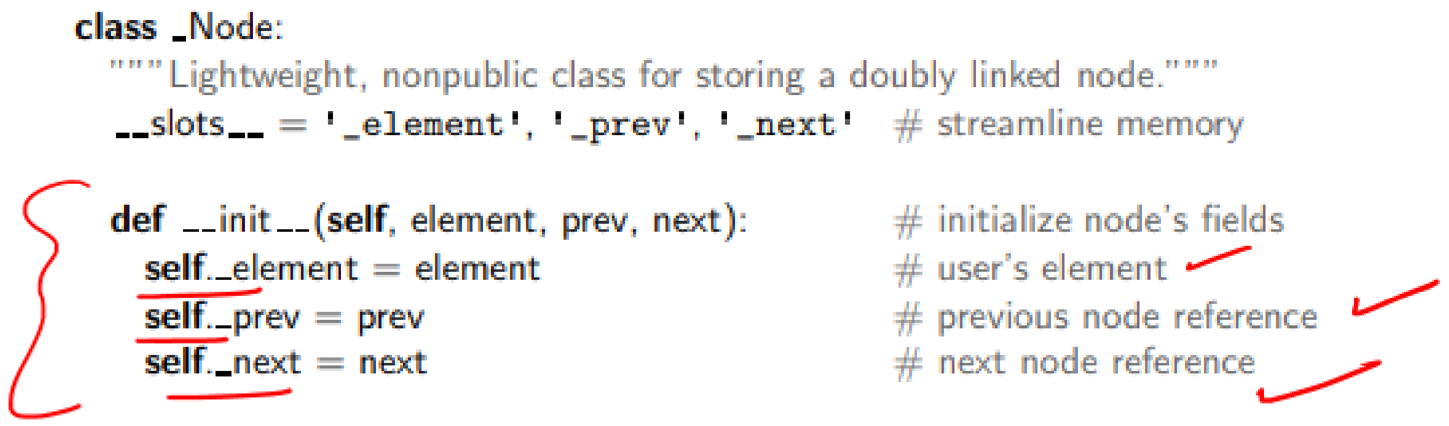
- Nodes store:
  - element
  - link to the previous node -*prev*
  - link to the next node-*next*
- Special trailer and header nodes





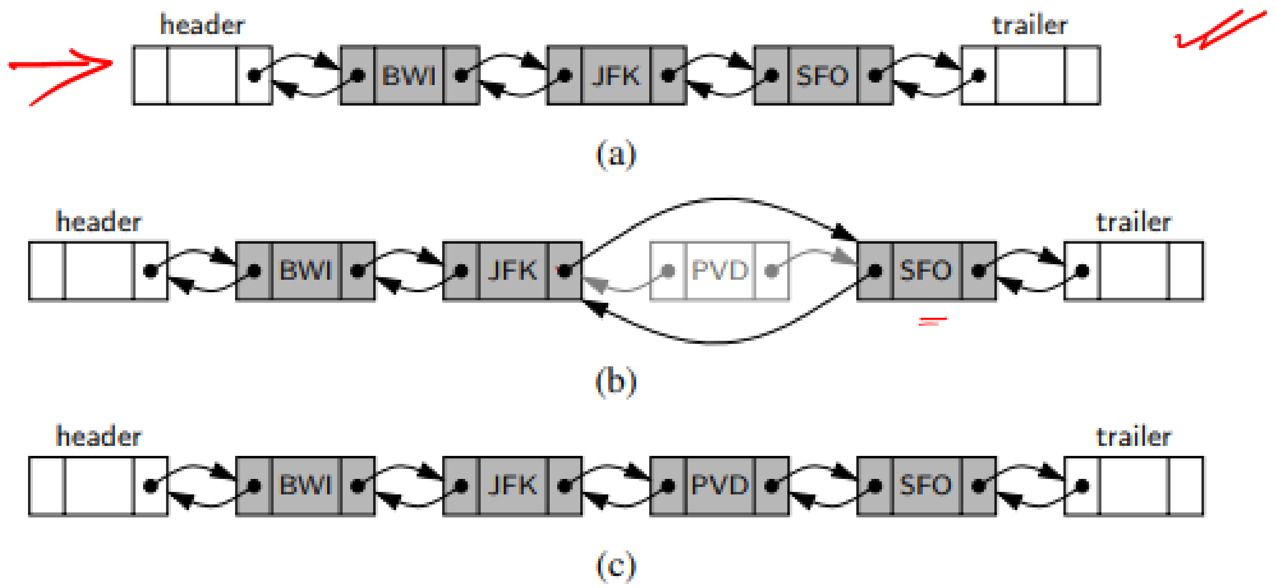
Doubly Linked List Implementation

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



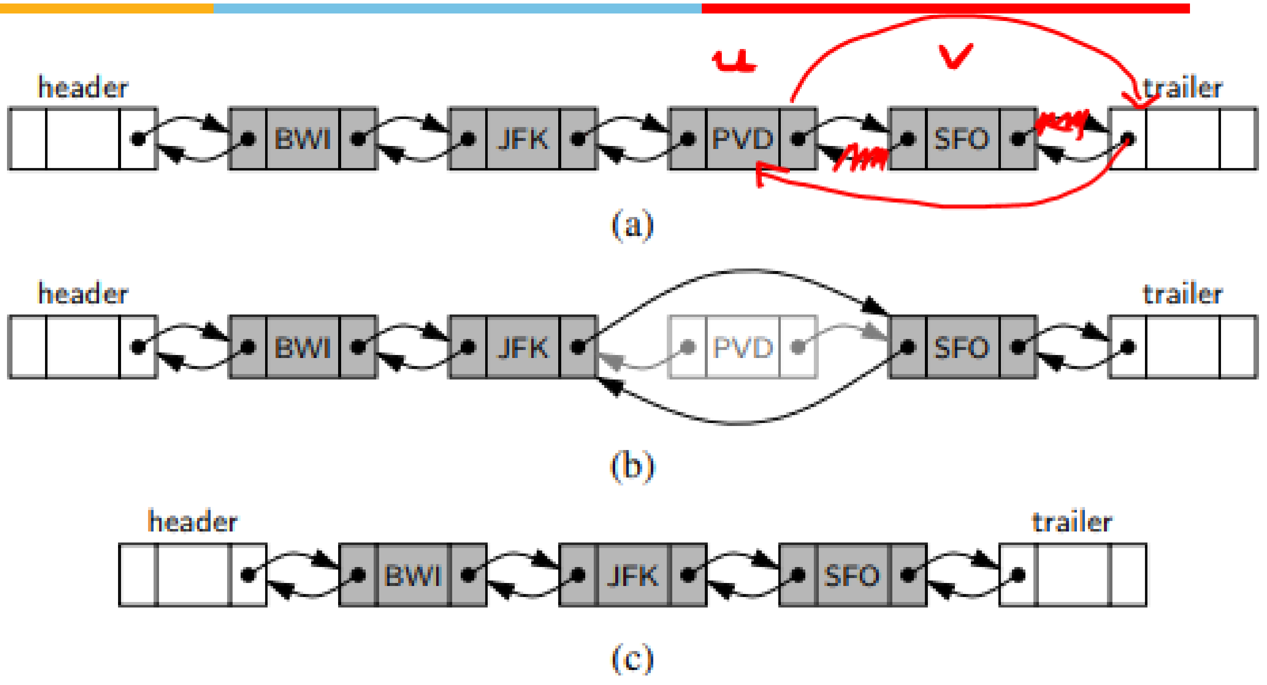
```
class _Node:
  """Lightweight, nonpublic class for storing a doubly linked node."""
  __slots__ = '_element', '_prev', '_next'    # streamline memory

  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)

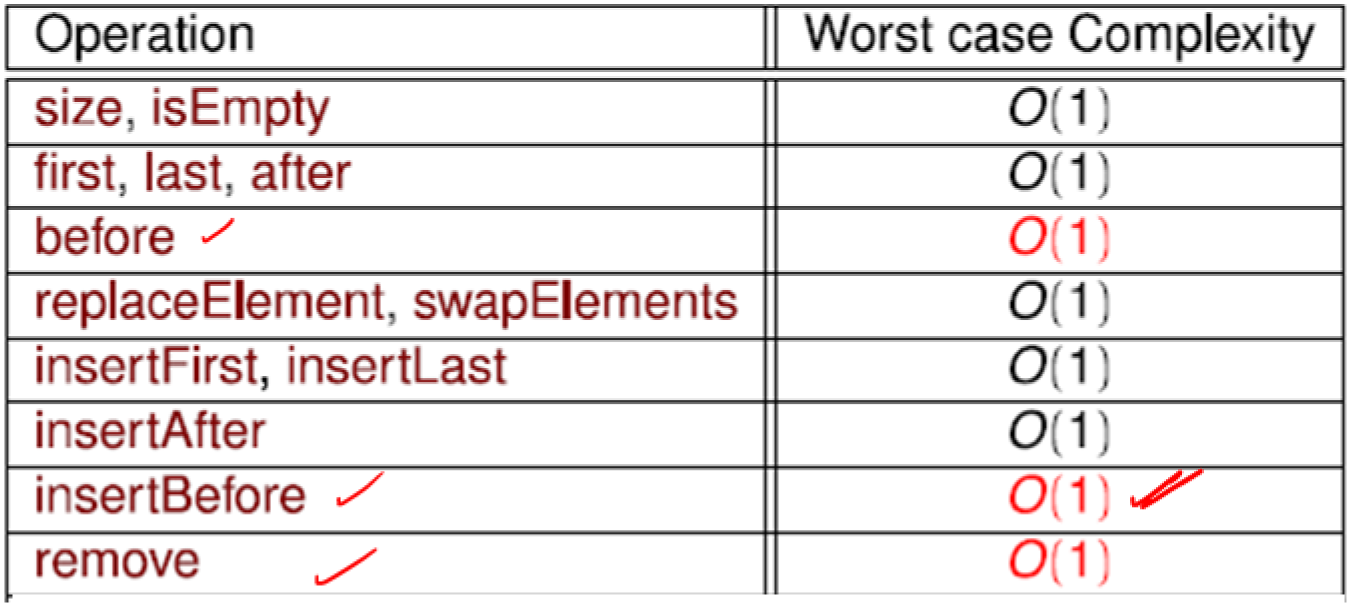
DLL –Node based-Operations

```
Algorithm removeLast():
    if size = 0 then
        Indicate an error: the list is empty
    v ← trailer.getPrev()          {last node}
    u ← v.getPrev()                {node before the last node}
    trailer.setPrev(u)
    u.setNext(trailer)
    v.setPrev(null)
    v.setNext(null)
    size = size − 1
```

```
Algorithm addFirst(v):
    w ← header.getNext()           {first node}
    v.setNext(w)
    v.setPrev(header)
    w.setPrev(v)
    header.setNext(v)
    size = size + 1
```

```
Algorithm addAfter(v,z):
    w ← 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 ← size + 1
```

DLL –Node Based-Performance



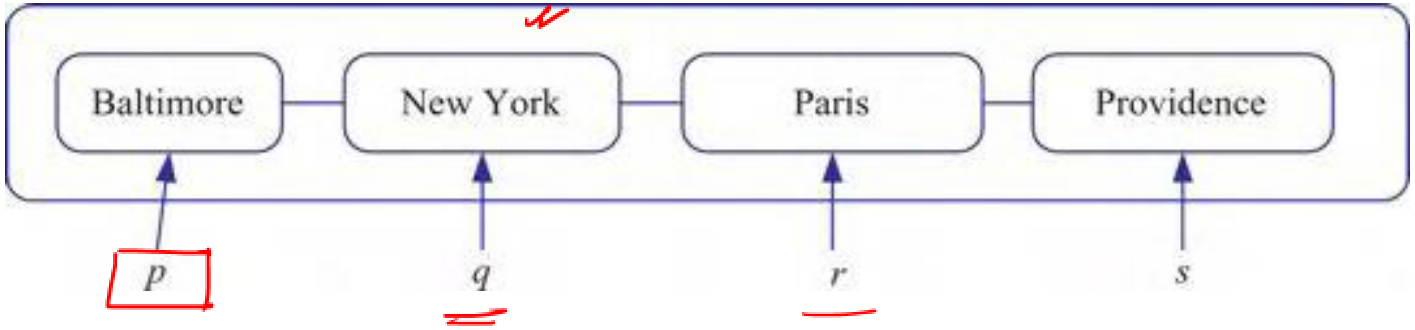
Operation	Worst case Complexity
size, isEmpty	$O(1)$
first, last, after	$O(1)$
before ✓	$O(1)$
replaceElement, swapElements	$O(1)$
insertFirst, insertLast	$O(1)$
insertAfter	$O(1)$
insertBefore ✓	$O(1)$ ✓
remove ✓	$O(1)$

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*.

p. element( ) {q. element ( )}

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*

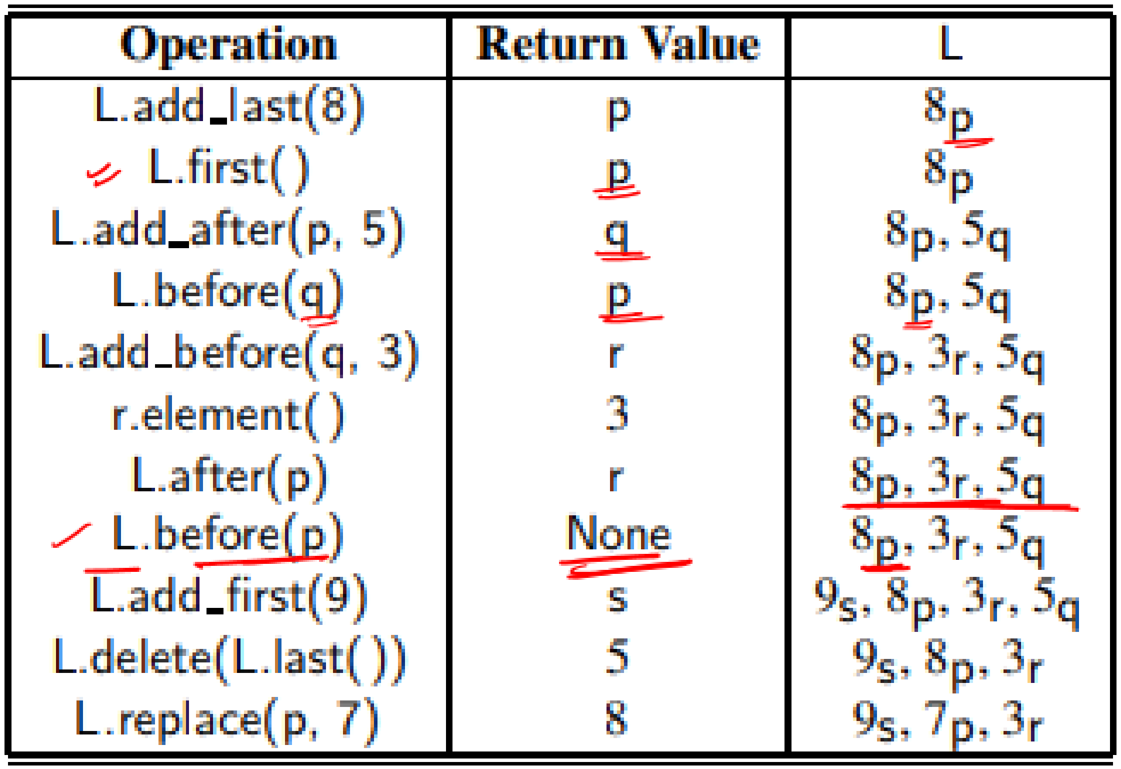
LIST - ADT

- 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

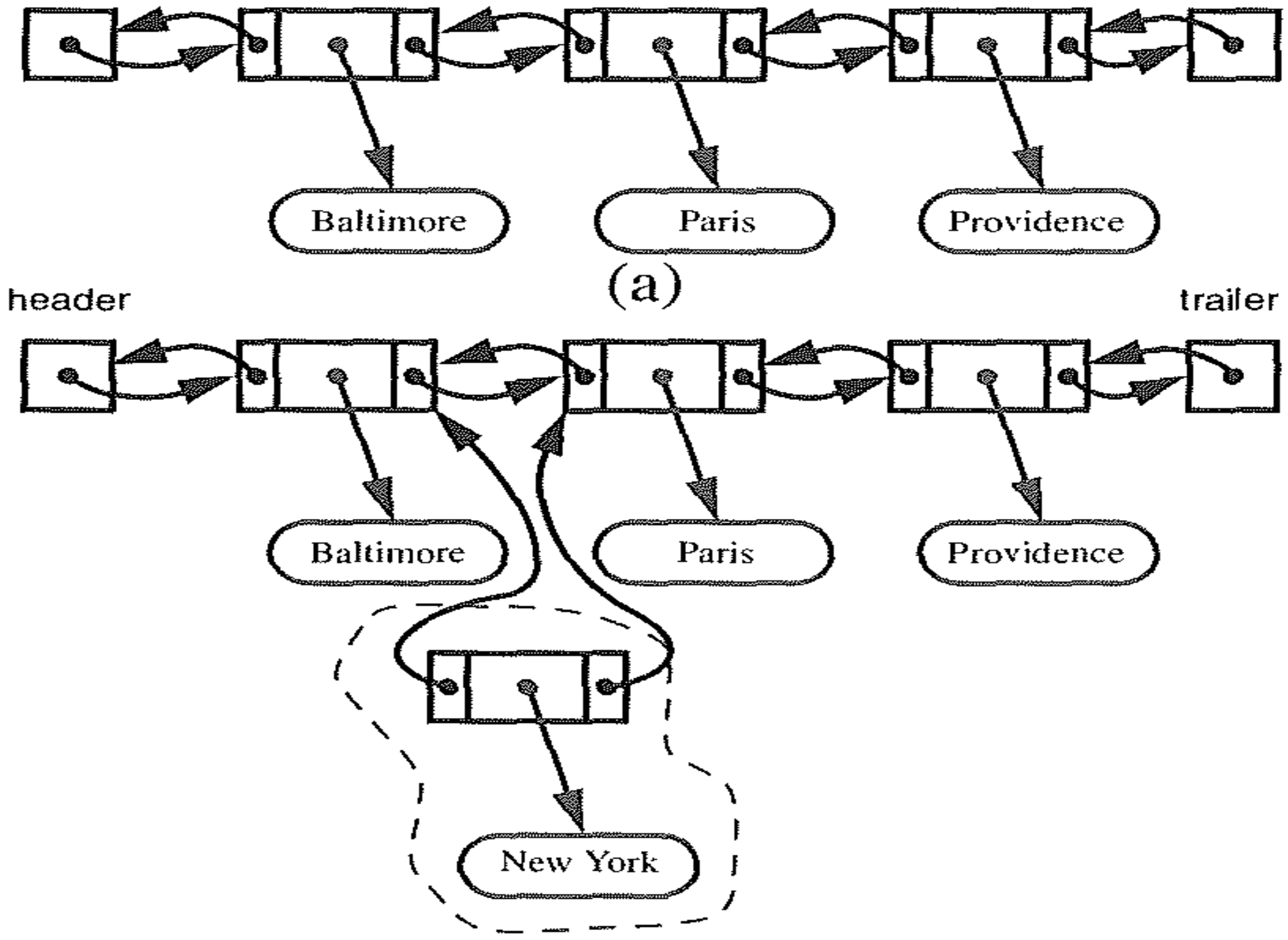
LIST - ADT-Doubly linked list Implementation

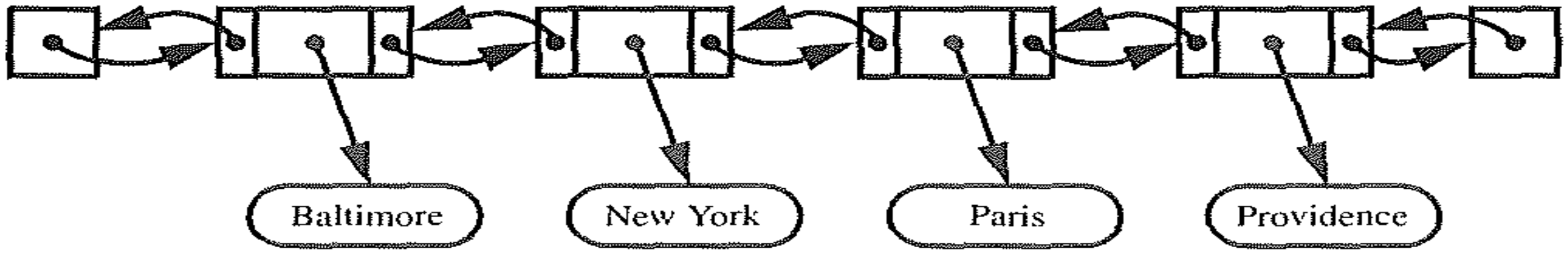
- The nodes of the linked list implement the position ADT, defining a method element ( ) , which returns the element stored at the node.
- The nodes themselves act as positions.

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 $\leftarrow$ e
    v.prev $\leftarrow$ p          {link v to its predecessor} ✓
    v.next $\leftarrow$ p.next     {link v to its successor} ✓
    (p.next).prev $\leftarrow$ v   {link p's old successor to v} ✓
    p.next $\leftarrow$ 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 $\boldsymbol{O}(1)$ time
  - Operation element() of the Position ADT runs in $\boldsymbol{O}(1)$ time

VECTOR-ADT

A Vector stores a list of elements:

- Access via rank/index.

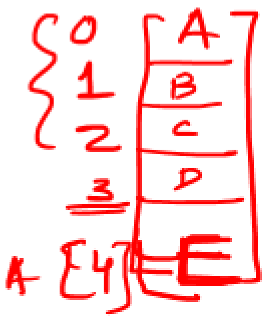
Accessor methods:
  elementAtRank(r),
Update methods:
  replaceAtRank(r, o),
  insertAtRank(r, o),
  removeAtRank(r)
Generic methods:
  size(), isEmtpy()

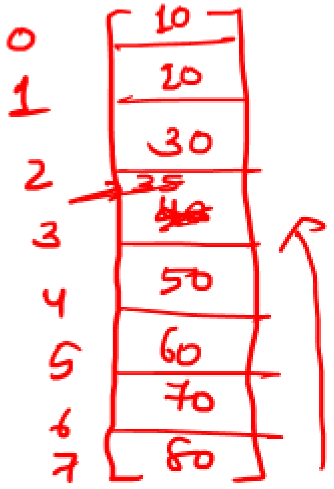
Here r is of type integer, n, m are nodes, o is an object (data).

VECTOR-ADT

InsertAtRank(3, 35)

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

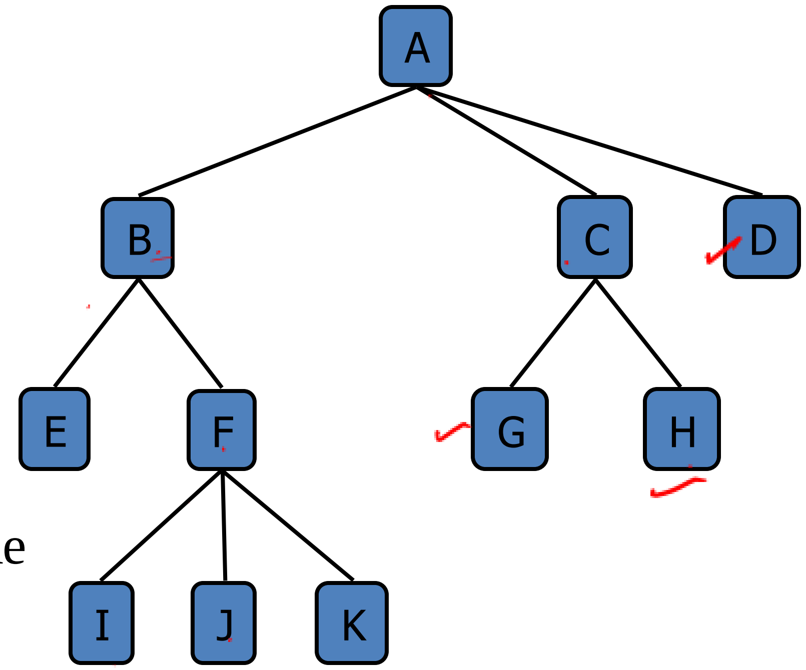
- Iterator allows to traverse the elements in a list or set.
- Iterator ADT provides the following methods:
  - object(): returns the current object
  - hasNext(): indicates whether there are more elements
  - nextObject(): goes to the next object and returns it

TREES

- In computer science, a tree is an abstract model of a hierarchical structure
- A tree consists of nodes with a parent-child relation
- **Applications:**
  - Organization charts
  - File systems
  - Compiler Design/Text processing (syntax analysis & to display the structure of a sentence in a language)
  - Searching Algorithms
  - Evaluating a mathematical expression.

TREE Terminologies

- ***Root:*** node without parent (A)
- ***Internal node***: node with at least one child (A, B, C, F)
- ***External node*** (a.k.a. leaf): node without children (E, I, J, K, G, H, D)
- ***Ancestors of a node***: parent, grandparent, grand-grandparent, etc.
- ***Depth of a node***: number of ancestors
- ***Height of a tree***: maximum depth of any node (3)
- ***Descendant of a node***: child, grandchild, grand-grandchild, etc.
- ***Degree of a node***: the number of its children



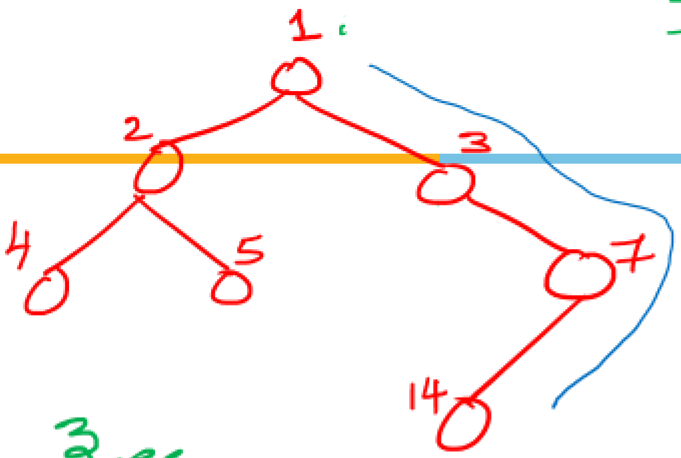
TREE ADT-Depth and Height

- Let v be a node of a tree T.
- The depth of v is the number of ancestors of v, excluding v itself
- If v is the root, then the depth of v is 0.
- Otherwise, the depth of v is one plus the depth of the parent of v.

```
Algorithm depth(T, v):
if T. isRoot(v) then
return 0
else
return 1 + depth (T, T. parent(v))
```

- The running time of algorithm depth (T, v) is ***O(1 + d_v),*** where d_v denotes the depth of the node v in the tree T.
- In the worst case, the depth algorithm runs in O (n) time, where n is the total number of nodes in the tree T

$2i \quad 2i+1$



```
Algorithm depth(T, v):
    if T. isRoot(v) then
        return 0
    else
        return 1 + depth (T, T. parent(v))
```

depth (T, 14)

↑ 3

return (1+ depth (T, 7))

return(1+2)

↑

return (1+1)

↑

return (1+0)

return (1+ depth (T, 3))

return (1+ depth (T, 1))

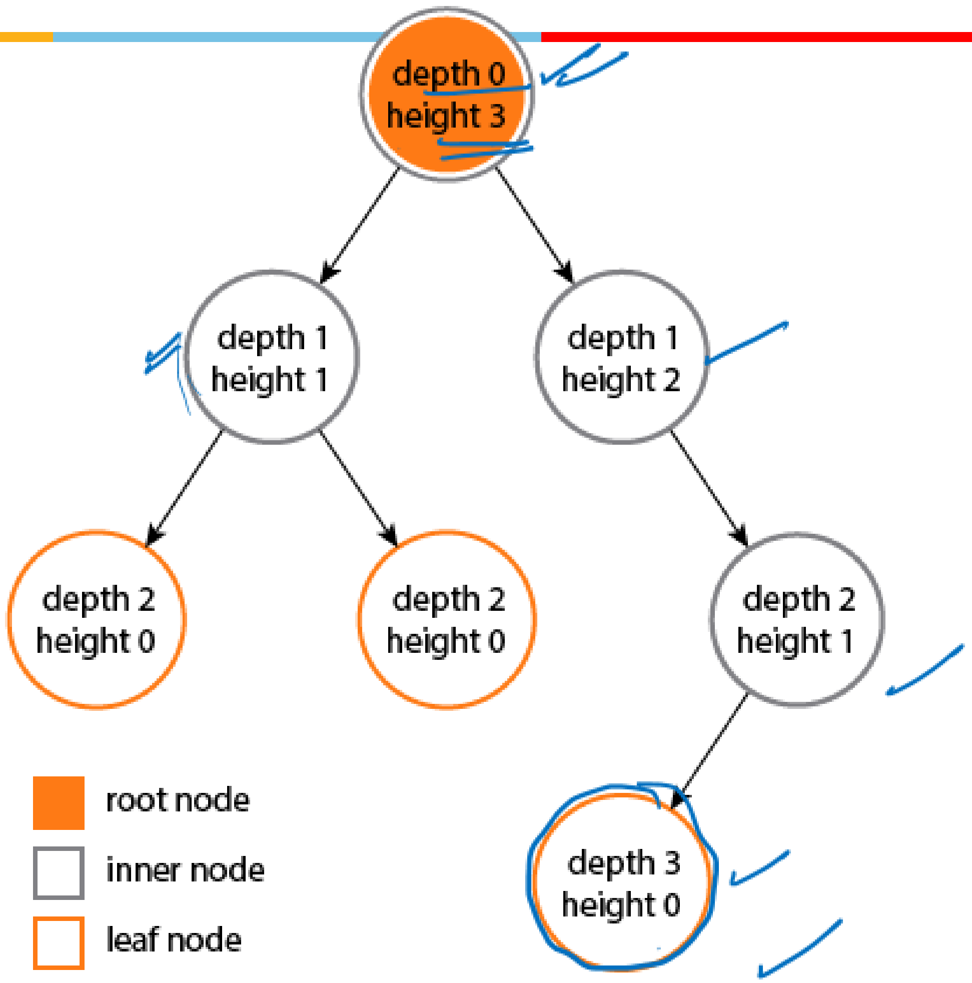
Skewed tree

TREE ADT-Depth and Height

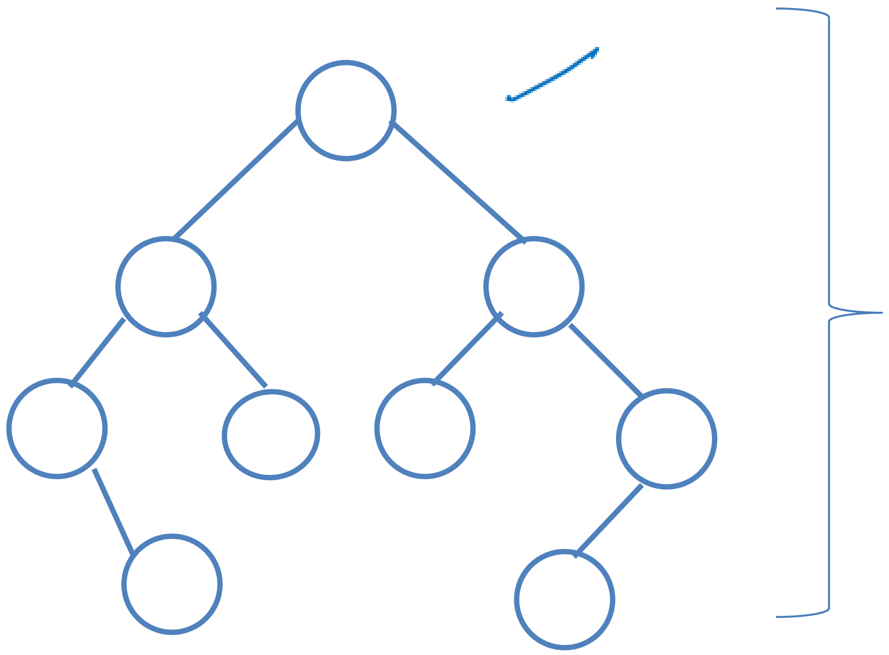
- The ***height of a tree*** T is equal to the maximum depth of an external node of T.
- If v is an external node, then the height o f v is 0.
- Otherwise, the height of v is one plus the maximum height of a child of v.

```
Algorithm height(T, v) :
if T. isExternal (v) then
    return 0
else
    h = 0
    for each w ∈ T.children (v) do
        h = max(h, height(T, w))
    return 1 + h
```

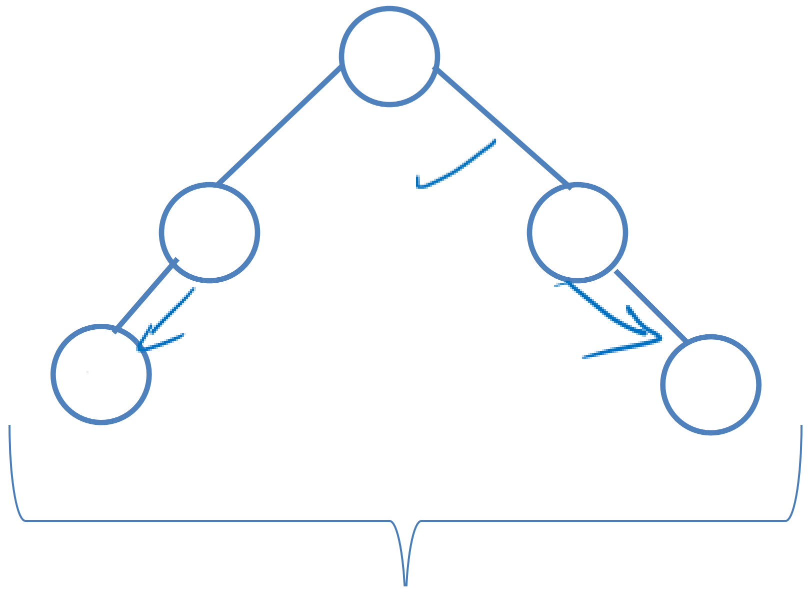
TREE ADT-Depth and Height



Binary Tree



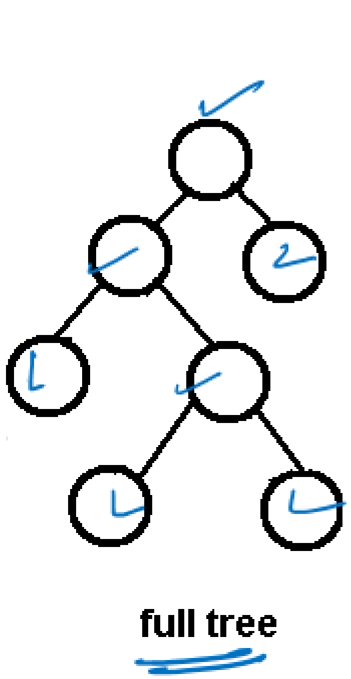
Each node can have at most 2 – child

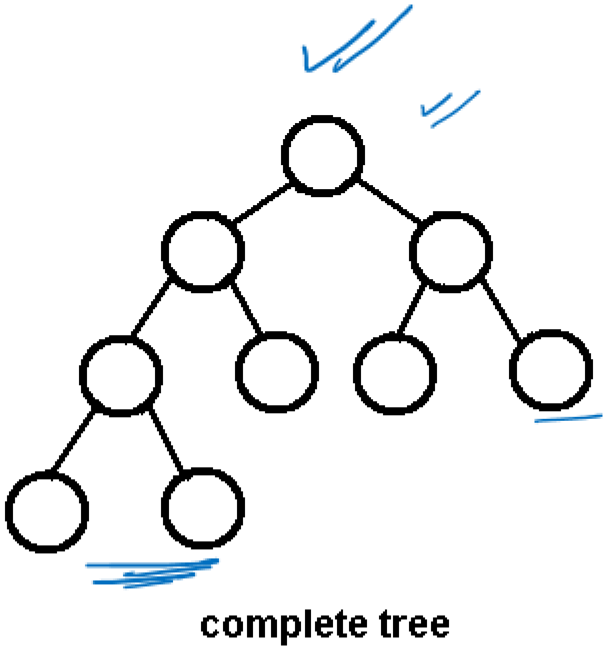


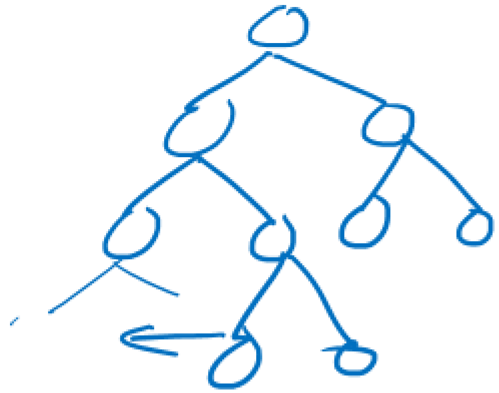
This is also binary tree

What about this ?

Complete and Full(Proper) Binary tree







A **full binary tree** (sometimes proper **binary tree** or 2-**tree**) is a **tree** in which every node other than the leaves has two children

In Complete binary tree All levels except possibly the last are completely filled and all nodes are as left as possible.

TREE ADT

- We use **Positions** to abstract nodes
- Generic methods: Back
  - ***integer size():*** Return the number of nodes in the tree.
  - ***boolean isEmpty()*** :
  - ***ObjectIterator elements():*** Return an iterator of all the elements stored at nodes of the tree.
  - ***positionIterator positions():*** Return an iterator of all the nodes of the tree
- Accessor methods:
  - ***position root():*** Return the root of the tree
  - ***position parent(p):*** Return the parent of node ~~v~~ p; error if ~~v~~ p is root.
  - ***positionIterator children(p):*** Return an iterator of the children of node ~~v~~ p.

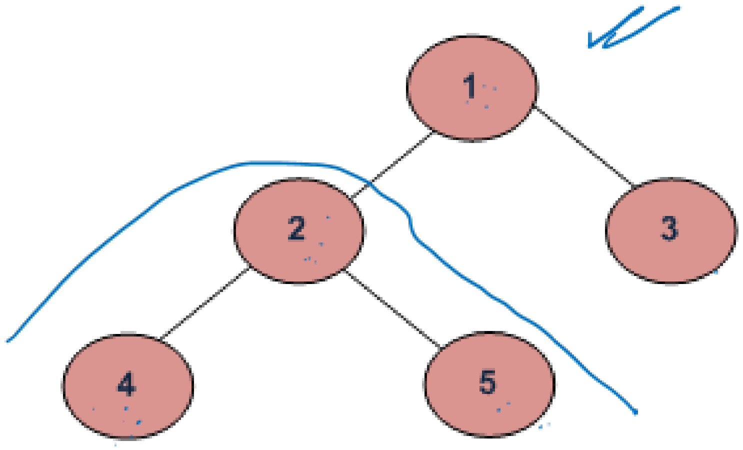
TREE ADT

- Query methods: ✓
  - ***boolean isInternal(p):***Test whether node v is internal. ✓
  - ***boolean isExternal(p):***Test whether node v is external
  - ***boolean isRoot(p):***Test whether node v is the root.
- Update methods:
  - ***swapElements(v, w):***Swap the elements stored at the nodes v and w.
  - ***object replaceElement(v, e):***Replace with e and return the element stored at node v

Assumptions

- The accessor methods root() and parent(v) take $O(1)$ time.
- The query methods isInternal(v), isExternal(v), and isRoot(v) take $O(1)$ time, as well.
- The accessor method children(v) takes $O(c_v)$ time, where c_v is the number of children of v.
- The generic methods swapElements(v, w) and replaceElement(v, e) take $O(1)$ time.
- The generic methods elements() and positions(), which return iterators, take $O(n)$ time, where n is the number of nodes in the tree.
- For the iterators returned by methods elements(), positions(), and children(v), the methods hasNext(), nextObject() or nextPosition() take $O(1)$ time each.

TREE Traversals

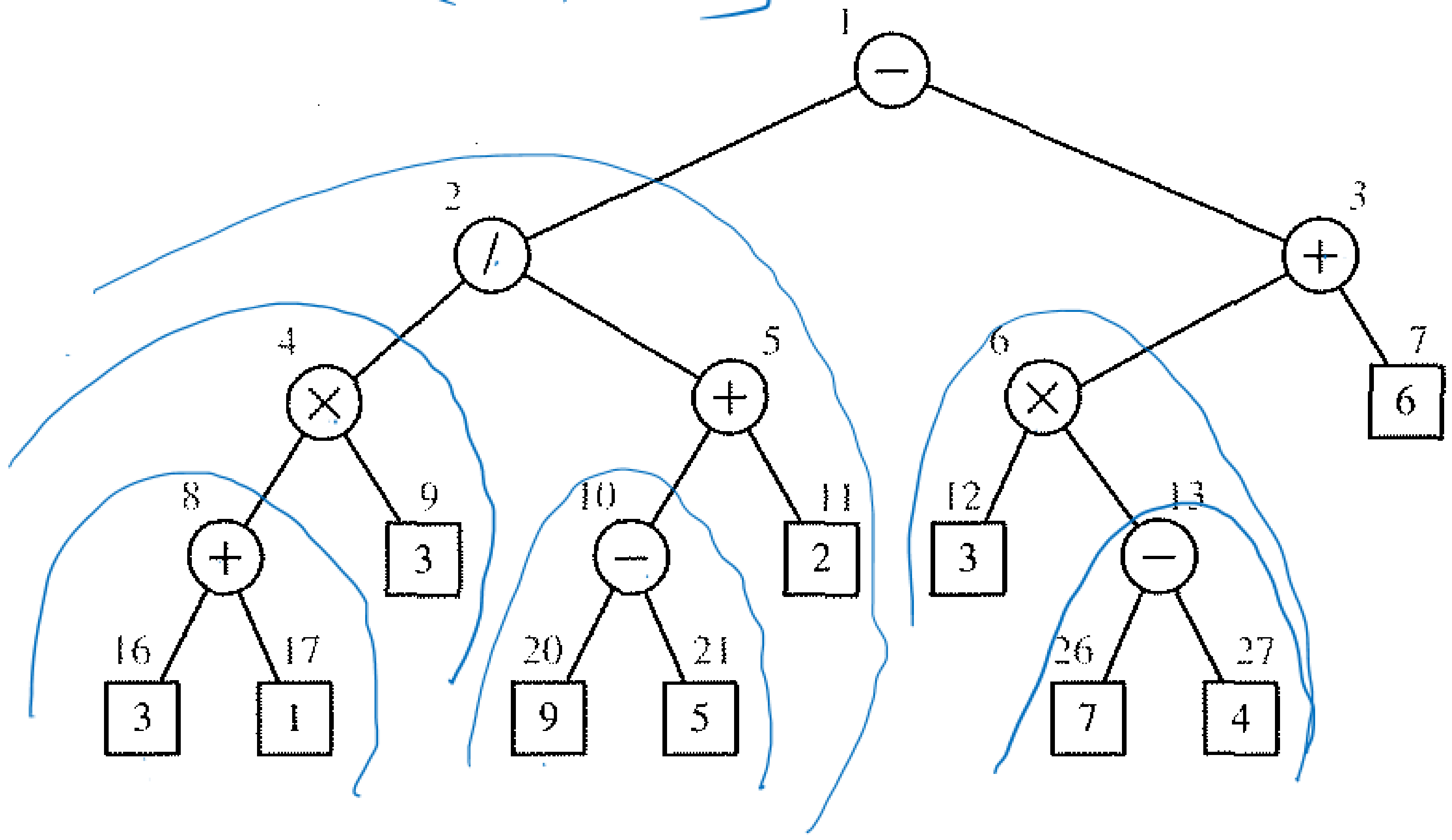


(a) Inorder (Left, Root, Right) : 4 2 5 1 3
(b) Preorder (Root, Left, Right) : 1 2 4 5 3
(c) Postorder (Left, Right, Root) : 4 5 2 3 1

TREE Traversals-Qn1-HW

Left Root Right

$[(3+1)\times 3/((9-5)+2)] - [3\times(7-4)+6]$

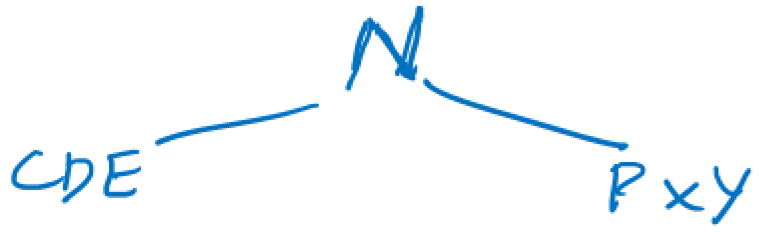


QN-2-Class Discussion

- Find the preorder traversal for the binary tree using
- Inorder traversal : C D E N P X Y
- Postorder traversal: D C E P Y X N
- Also construct the tree.

Left Root Right

[Left Right Root]

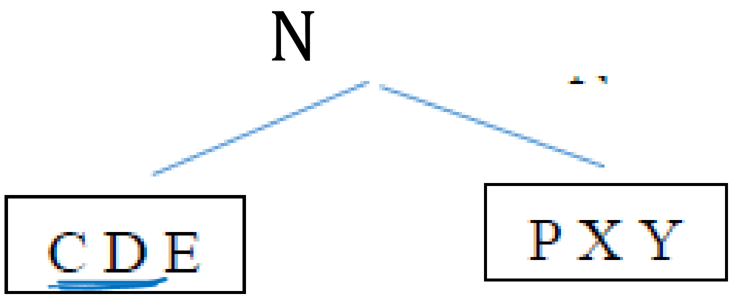


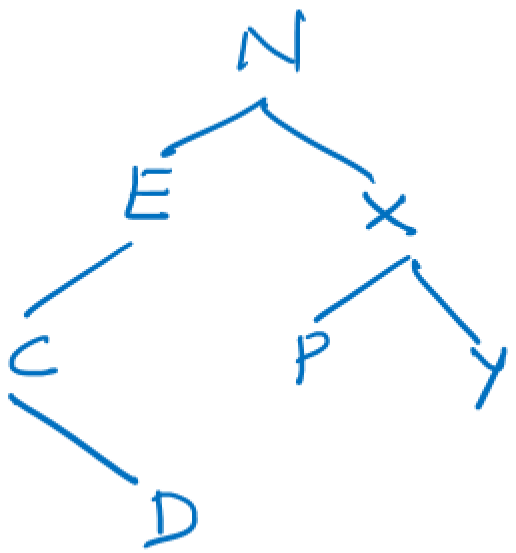
Qn2:Traversal-ANSWER

- Inorder traversal -C D E N P X Y
- Post order traversal - D C E P Y X N
- We first find the last node in postorder. The last node is "N", we know this value is root as root always appear in the end of postorder traversal.
- We search "N" in inorder to find left and right subtrees of root. Everything on left of "N" in inorder is in left subtree and everything on right is in right subtree.

Qn2:Traversal-ANSWER

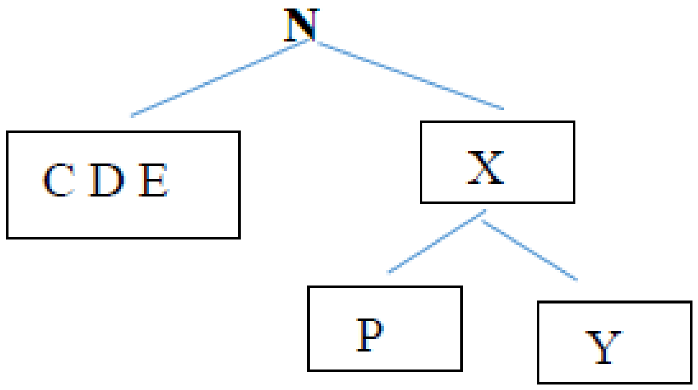
- Inorder traversal -C D $\boxed{E}$ N P $\boxed{X}$ Y
- Post order traversal - D C E P Y X N
-





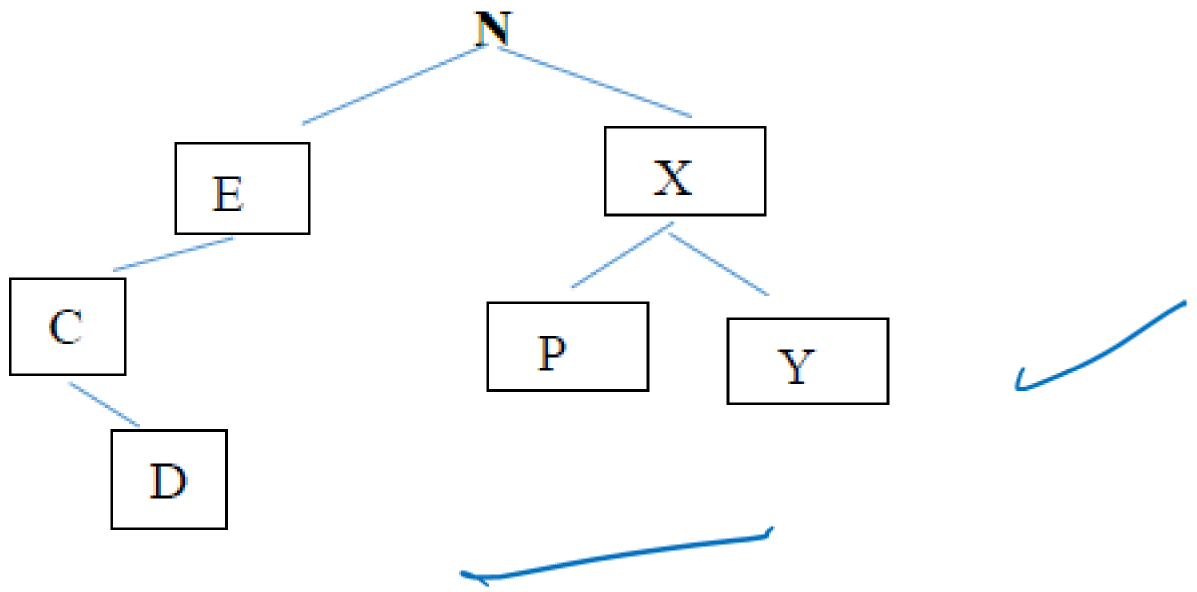
Qn2:Traversal-ANSWER

- Inorder traversal -C D E N P X Y
- Post order traversal - D C E P Y X N
- We recur the above process for following
- Recur for in = {P,X,Y} and post = {P,Y,X}
- Make the created tree as right child of root



Qn2:Traversal-ANSWER

- Inorder traversal -C D E N P X Y
- Post order traversal - D C E P Y X N
- Recur for in = {C,D,E} and post = {D,C,E}



Qn2:Traversal-ANSWER

- Inorder traversal -C D E N P X Y
- Post order traversal - D C E P Y X N
-
- **ANSWER:** **N E C D X P Y** ✓

QN-3-HW

- Suppose that you are using a doubly linked list, maintaining a reference to the first and last node in the list, along with its size, to implement the operations below
- **addFirst(item):** prepend the item to the beginning of the list
- **get(i)** :return the item at position i in the list
- **set(i, item)** :replace position i in the list with the item
- **removeLast()** delete and return the item at the end of the list

- What is the worst-case running time of each of the operation above?

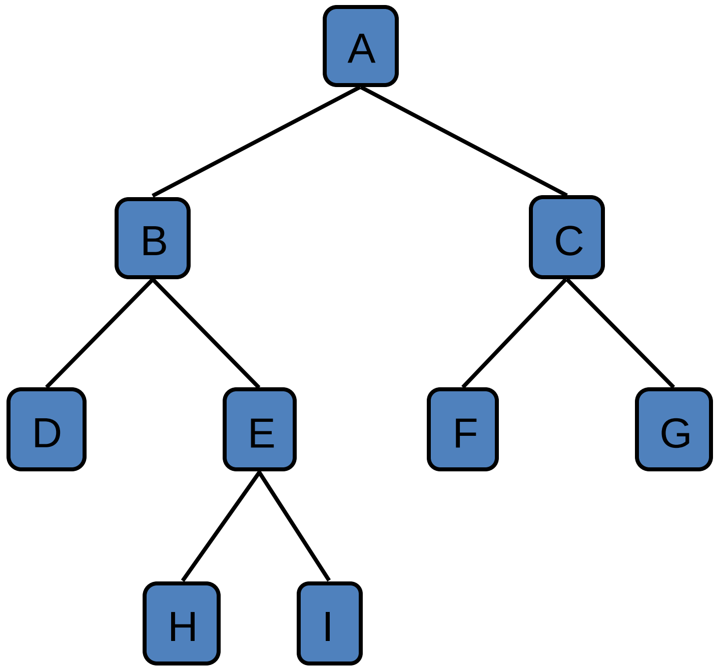
QN-4-HW

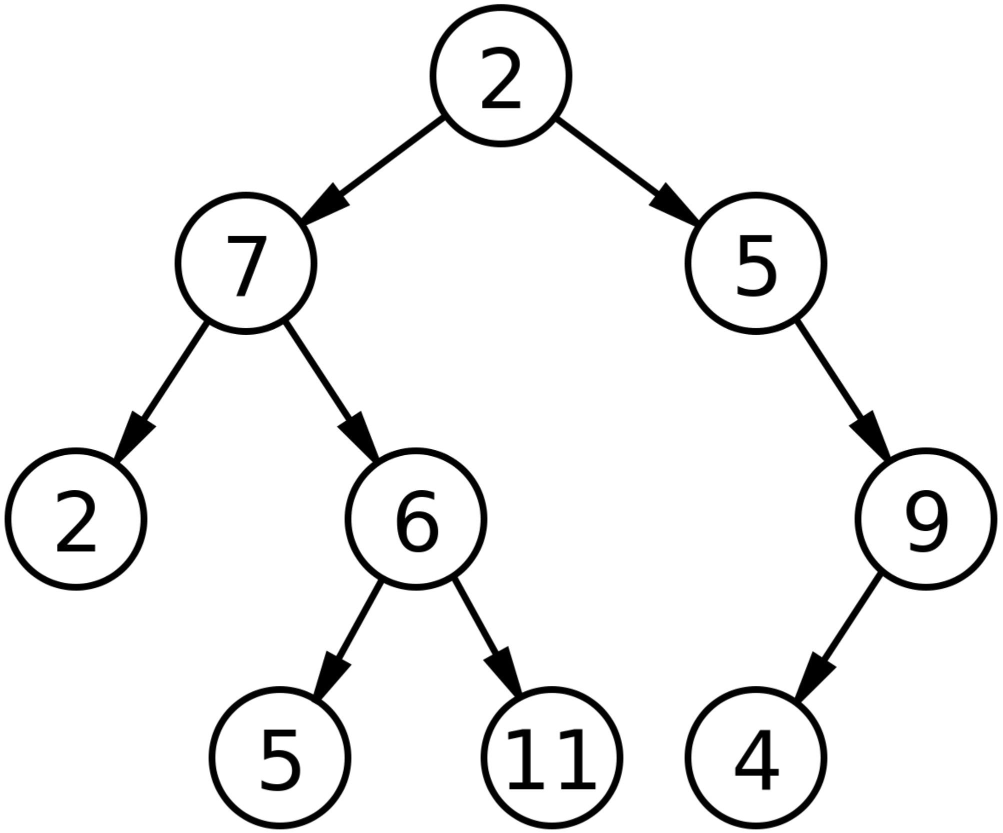
- We can construct a full binary tree from pre order and post order traversals. Discuss with an example.

Binary Trees

- A binary tree is a tree with the following properties:
  - Each internal node has two children
  - Each internal node has **ATMOST** two children
  - The children of a node are an ordered pair
- We call the children of an internal node -**left child and right child**
- Alternative recursive definition: a binary tree is either
  - a tree consisting of a single node, or
  - a tree whose root has an ordered pair of children, each of which is a binary tree

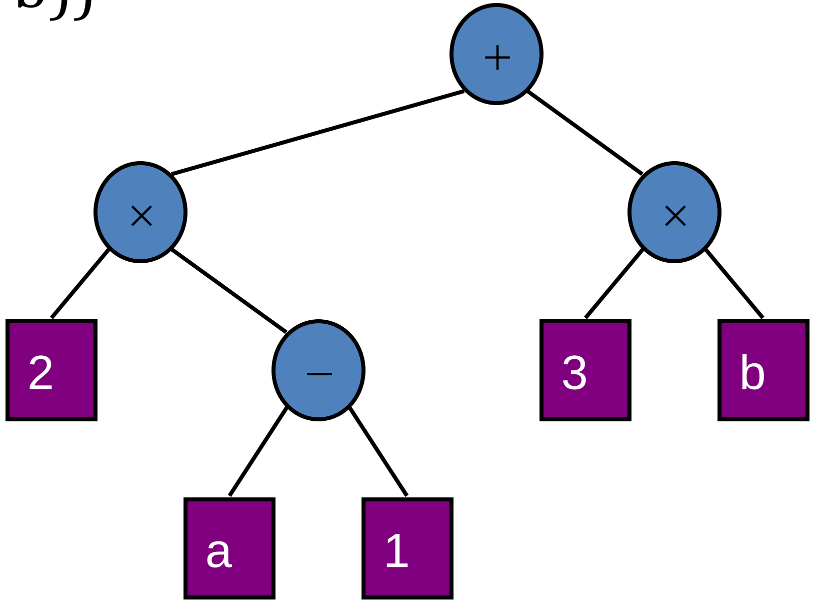
Binary Trees





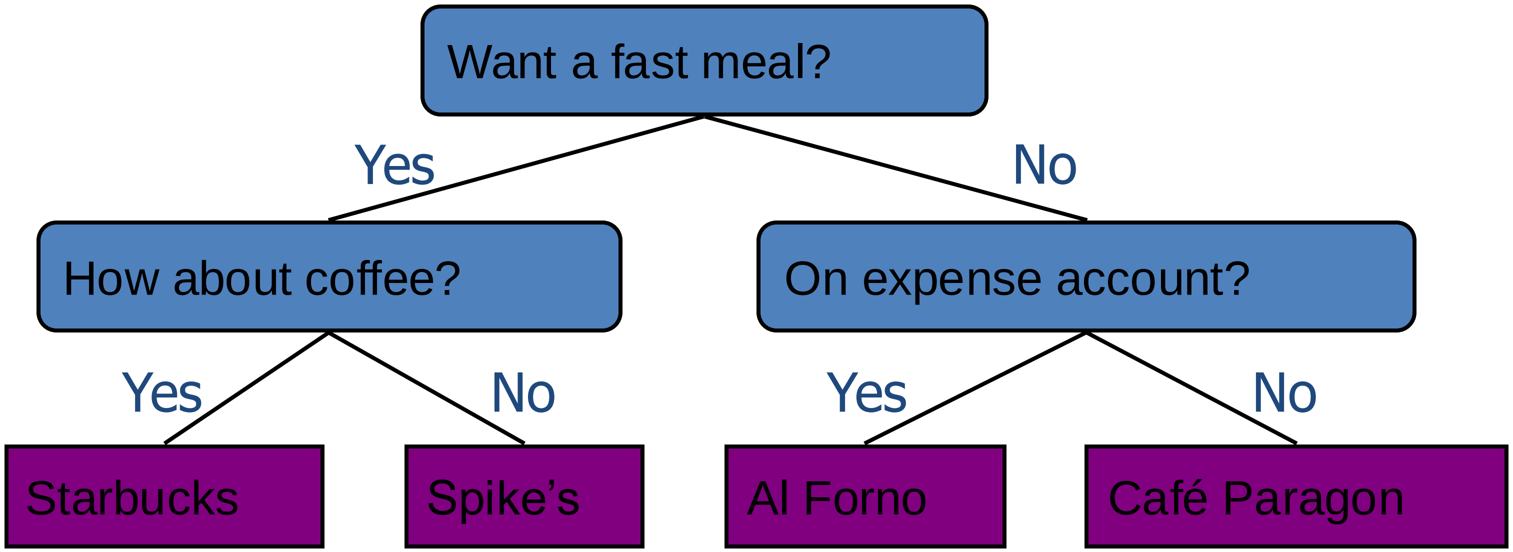
Arithmetic Expression Tree

- Binary tree associated with an arithmetic expression
  - internal nodes: operators
  - external nodes: operands
- Example: arithmetic expression tree for the expression ((2 × (a − 1)) + (3 × b))



Decision Tree

- Binary tree associated with a decision process
  - internal nodes: questions with yes/no answer
  - external nodes: decisions
- Example: dining decision



Properties of (proper)Binary Trees

- Notation

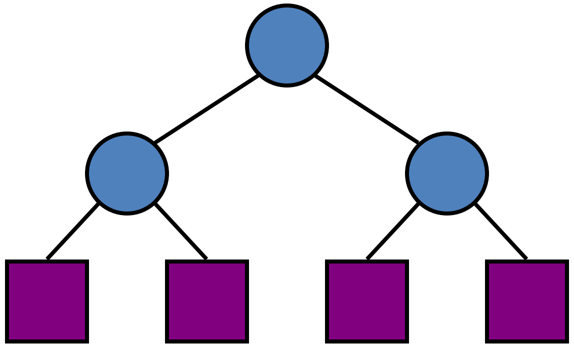
  $\boldsymbol{n}$ number of nodes

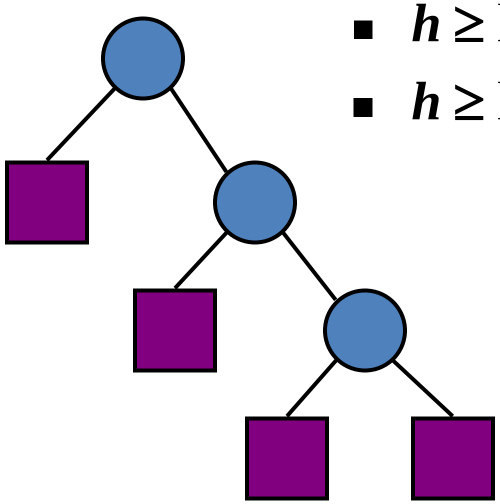
  $\boldsymbol{e}$ number of external nodes

  $\boldsymbol{i}$ number of internal nodes

  $\boldsymbol{h}$ height

- Properties:
  - $e = i + 1$
  - $n = 2e - 1$
  - $h \leq i$
  - $h \leq (n - 1)/2$
  - $e \leq 2^h$
  - $h \geq \log_2 e$
  - $h \geq \log_2 (n + 1) - 1$





Properties of Binary Trees

Let T be a (proper) binary tree with n nodes, and let h denote the height of T. Then T has the following properties:

- The number of external nodes in T is at least h + 1 and at most 2^h .
- The number of internal nodes in T is at least h and at most $2^h - 1$.
- The total number of nodes in T is at least 2h + 1 and at most $2^{h+1} - 1$.
- The height of T is at least log(n + 1) -1 and at most (n - 1) /2, that is, log(n + 1) - 1 <=h <= (n - 1)/2.
- The number of external nodes is 1 more than the number of internal nodes.

BinaryTree ADT-Methods

- The BinaryTree ADT extends the Tree ADT, i.e., it inherits all the methods of the Tree ADT

Additional methods:

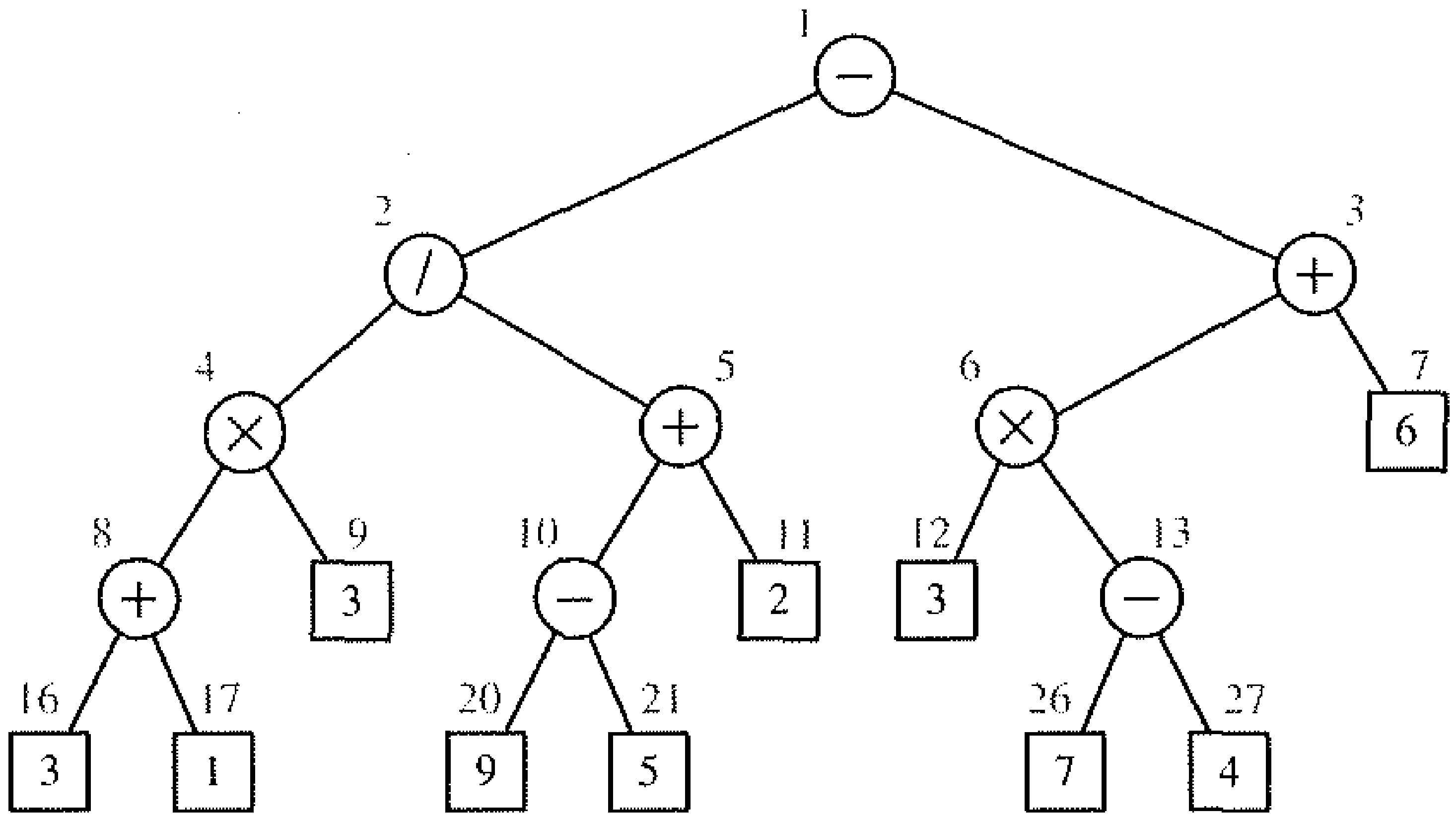
- ***position leftChild(v)*** Return the left child of v; an error condition occurs if v is an external node.
- ***position rightChild(v)***
- ***position sibling(p)***

- Update methods may be defined by data structures implementing the BinaryTree ADT

Binary Trees-Representations Vector Based

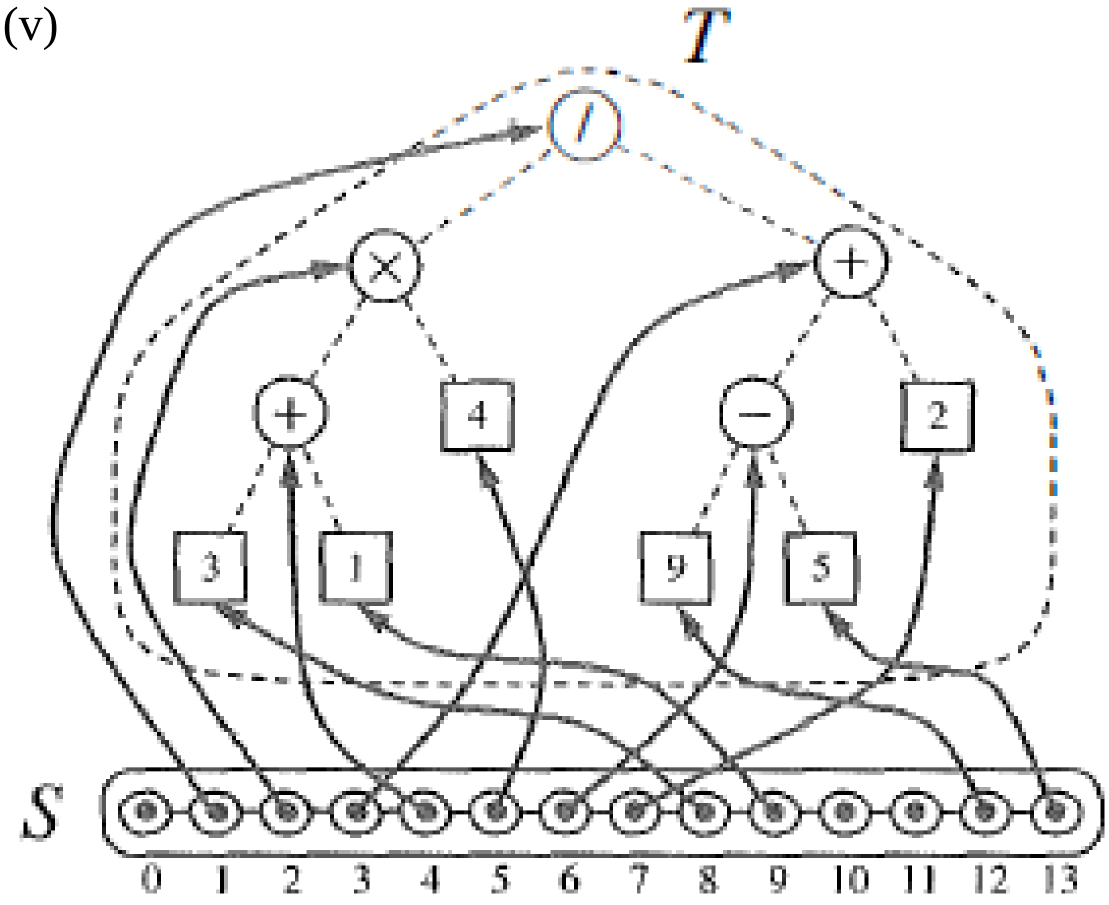
- For every node v of T , let p(v) be the integer defined as follows.
- If v is the root of T, then p(v) = 1 .
- If v is the left child of node u, then p(v) = 2p(u) .
- If v is the right child of node u, then p(v) = 2p (u) + 1 .
- ***p is known as a level numbering***

Binary Trees-Representations Vector Based



Binary Trees-Representations-Vector Based

Node v of T is associated with the element of S at rank p (v)



Binary Trees-Representations Vector Based

- Let n be the number of nodes of T, and let P_M be the maximum value of p (v) over all the nodes of T .
- Vector S has size N = P_M + 1 since the element of S at rank 0 is not associated with any node of T .
- Vector S will have, in general, a number of empty elements that do not refer to existing nodes of T.
- These empty slots could correspond to empty external nodes or even slots where descendants of such nodes would go
- In the worst case, $N = 2^{(n+1)/2}$
- //n is the number of nodes of T,N is the vector size

Binary Trees-Representations Vector Based

- Operation Time

– positions, elements	O(n)
– swap Elements, replaceElement	O(1)
– root, parent, children	O(1)
– leftChild, rightChild, sibling	O (1)
– isInternal , isExternal, isRoot	O(1)

[Methods]

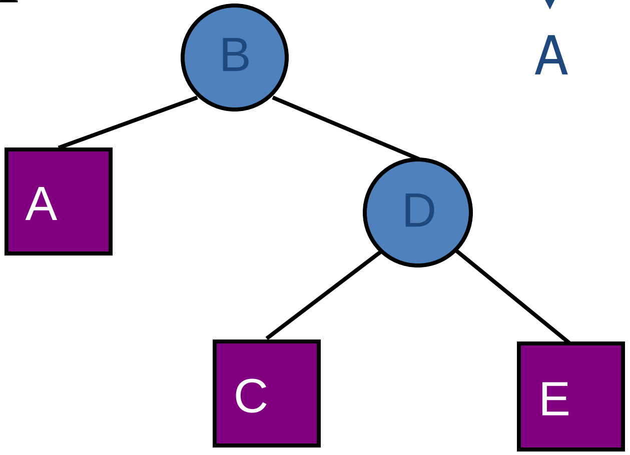
- Fast and Simple
- Space inefficient if the height of the tree is large

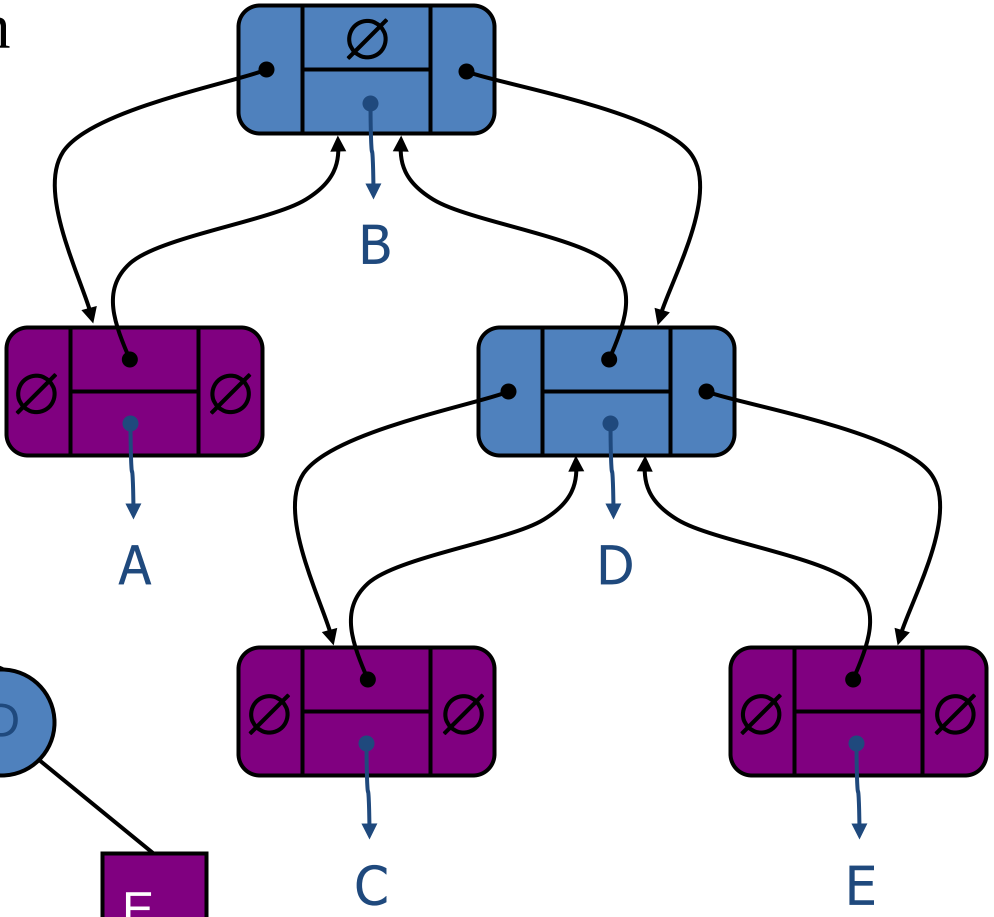
Binary Trees-Representations Linked Structure

A node is represented by an object storing

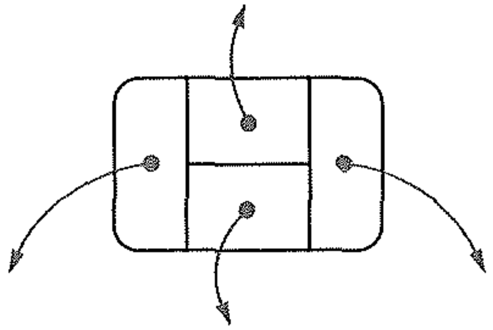
- Element
- Parent node
- Left child node
- Right child node

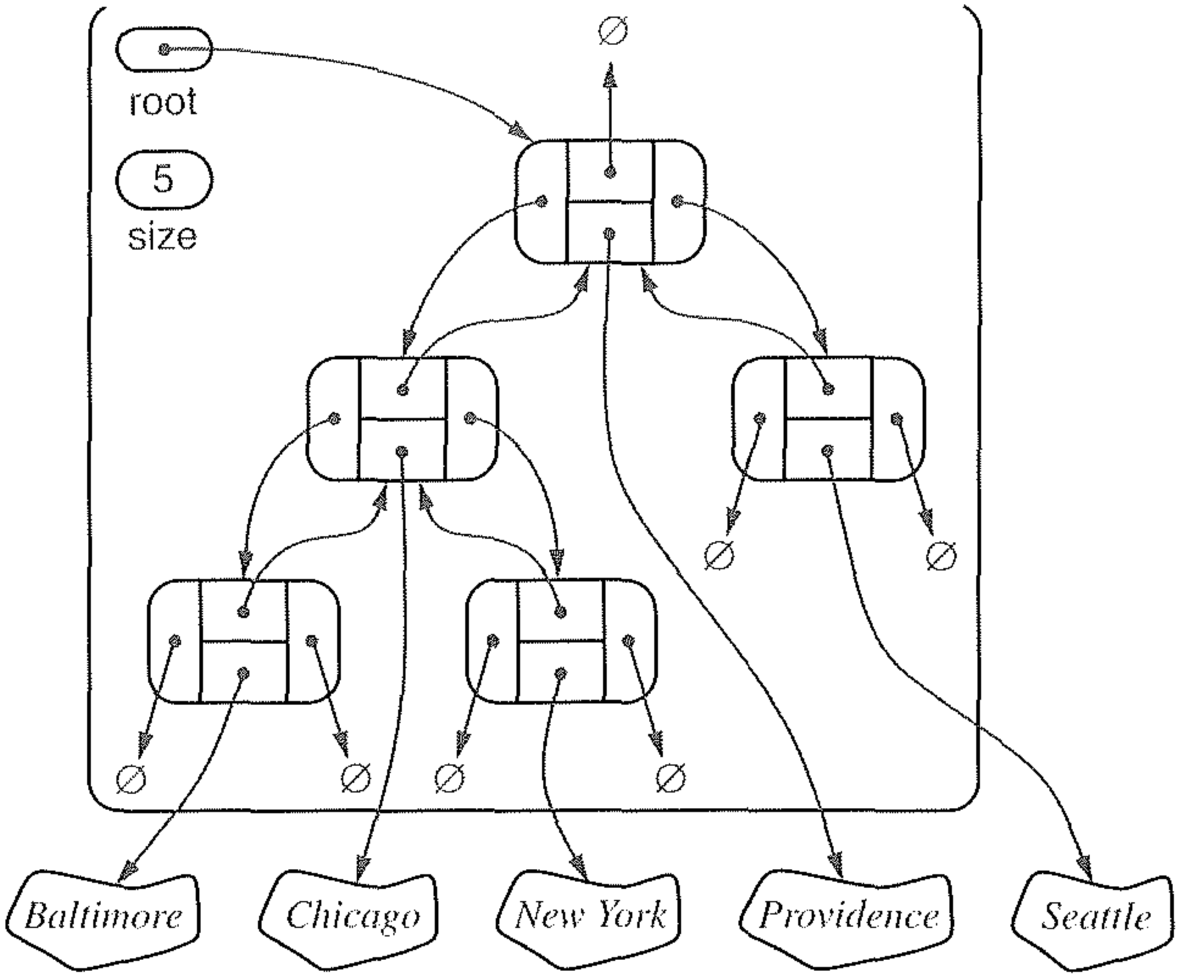
Node objects implement the Position ADT





Binary Trees-Representations Linked Structure





Binary Trees-Representations Linked Structure

- Operation Time
- Size()
- isEmpty()
- swapElements(v,w)
- replaceElement(v,e)

O(1)

- Positions()
- Elements()

O(n)

Binary Trees-Applications

- Data Base indexing
- BSP in Video Games
- Path finding algorithms in AI applications
- Huffman coding
- Heaps
- SET AND MAP IN C++
- Syntax tree

THANK YOU!

BITS Pilani
Hyderabad Campus