Linked Lists,
Trees and Tree Algorithms

Week 4

Midterm

8/12/2025

- @5.30pm in Canvas (questions via DM to me in Zoom)
- Material covered: everything covered in Weeks 1,2,3 and 4, on lectures (slides) and tutorials (colab notebooks); all 4 weeks
- Sample questions: https://www.w3schools.com/python/default.asp
- Canvas: Practice quiz with 5 questions (15 minutes)
- @7pm we continue with the lectures
- If the time does not work for you please let me know

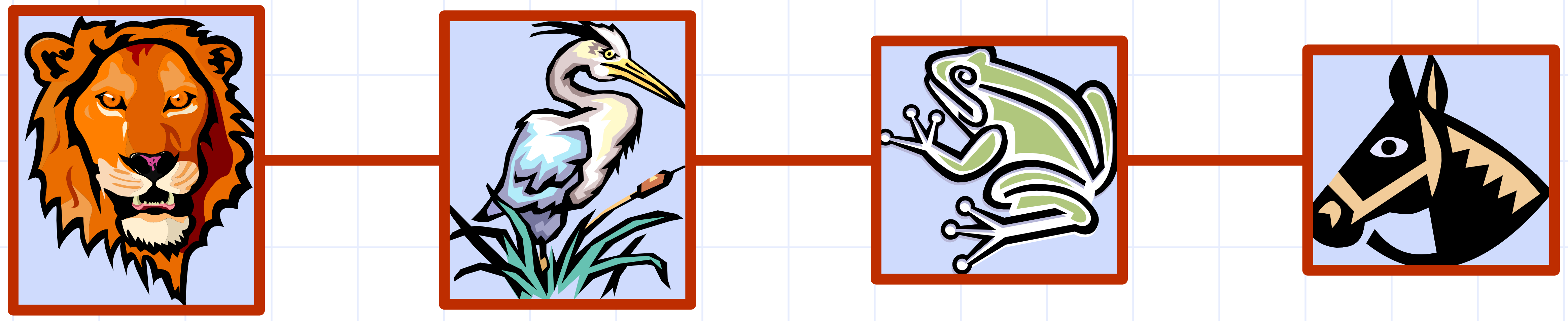
Week 4 Outline

- Linked lists
  - Single
  - Double
- Trees
  - Binary trees
  - Tree algorithms

Week 5 Outline

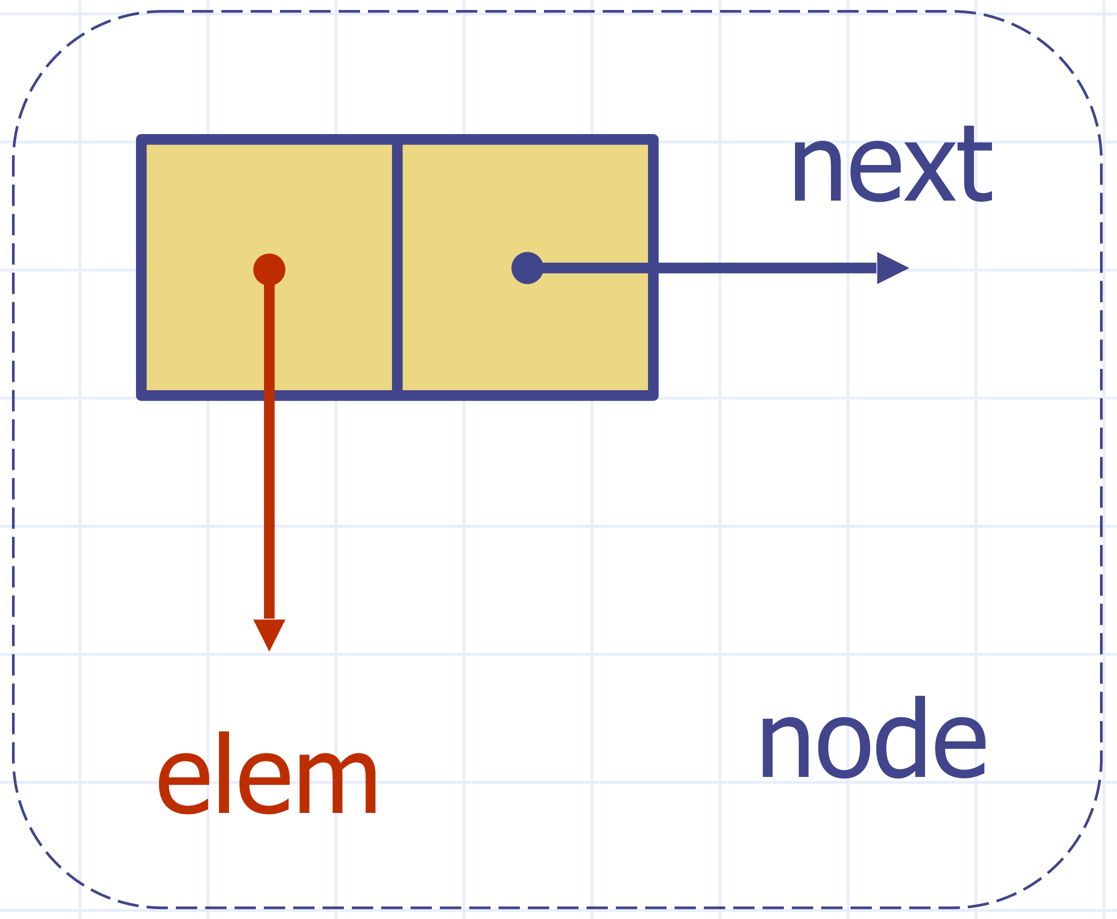
- At 5.30pm Midterm (90 minutes)
- At 7pm: Lecture
- Trees: Priority queues, Heaps, Binary Search Trees
- Project introduction
- Scientific Python packages
  - Numpy
  - Pandas
  - SciPy

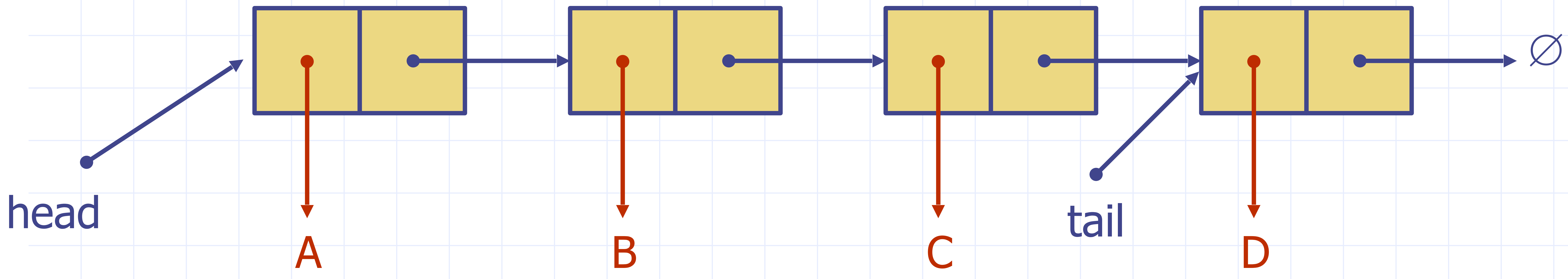
Linked Lists



Singly Linked List

- A singly linked list is a concrete data structure consisting of a sequence of nodes, starting from a head pointer
- Each node stores
  - element
  - link to the next node



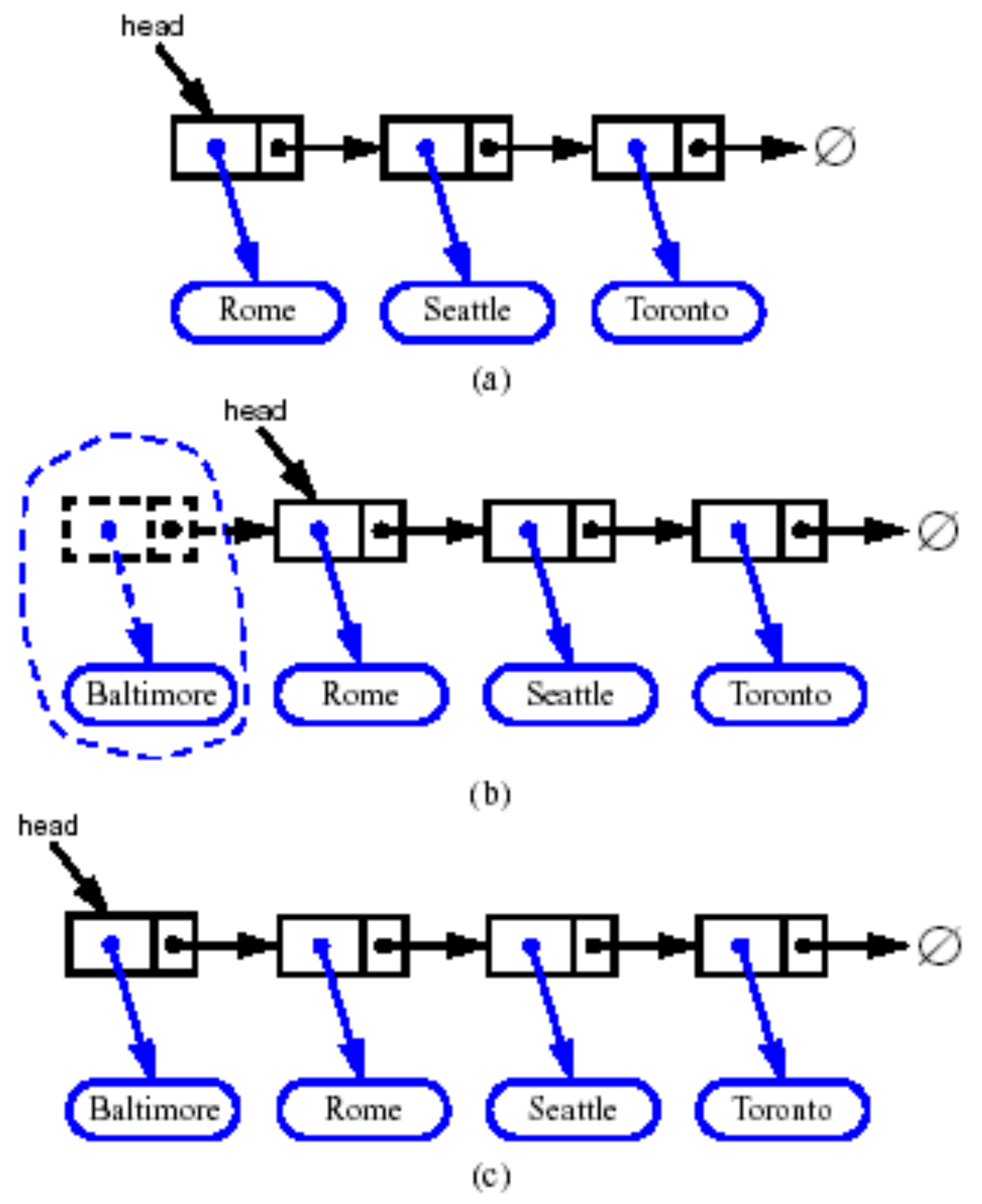


The Unordered List Abstract Data Type

- `List()` creates a new list that is empty. It needs no parameters and returns an empty list.
- `add(item)` adds a new item to the list. It needs the item and returns nothing. Assume the item is not already in the list.
- `remove(item)` removes the item from the list. It needs the item and modifies the list. Raise an error if the item is not present in the list.
- `search(item)` searches for the item in the list. It needs the item and returns a Boolean value.
- `is_empty()` tests to see whether the list is empty. It needs no parameters and returns a Boolean value.
- `size()` returns the number of items in the list. It needs no parameters and returns an integer.
- `append(item)` adds a new item to the end of the list making it the last item in the collection. It needs the item and returns nothing. Assume the item is not already in the list.
- `index(item)` returns the position of item in the list. It needs the item and returns the index. Assume the item is in the list.
- `insert(pos, item)` adds a new item to the list at position `pos`. It needs the item and returns nothing. Assume the item is not already in the list and there are enough existing items to have position `pos`.
- `pop()` removes and returns the last item in the list. It needs nothing and returns an item. Assume the list has at least one item.
- `pop(pos)` removes and returns the item at position `pos`. It needs the position and returns the item. Assume the item is in the list.

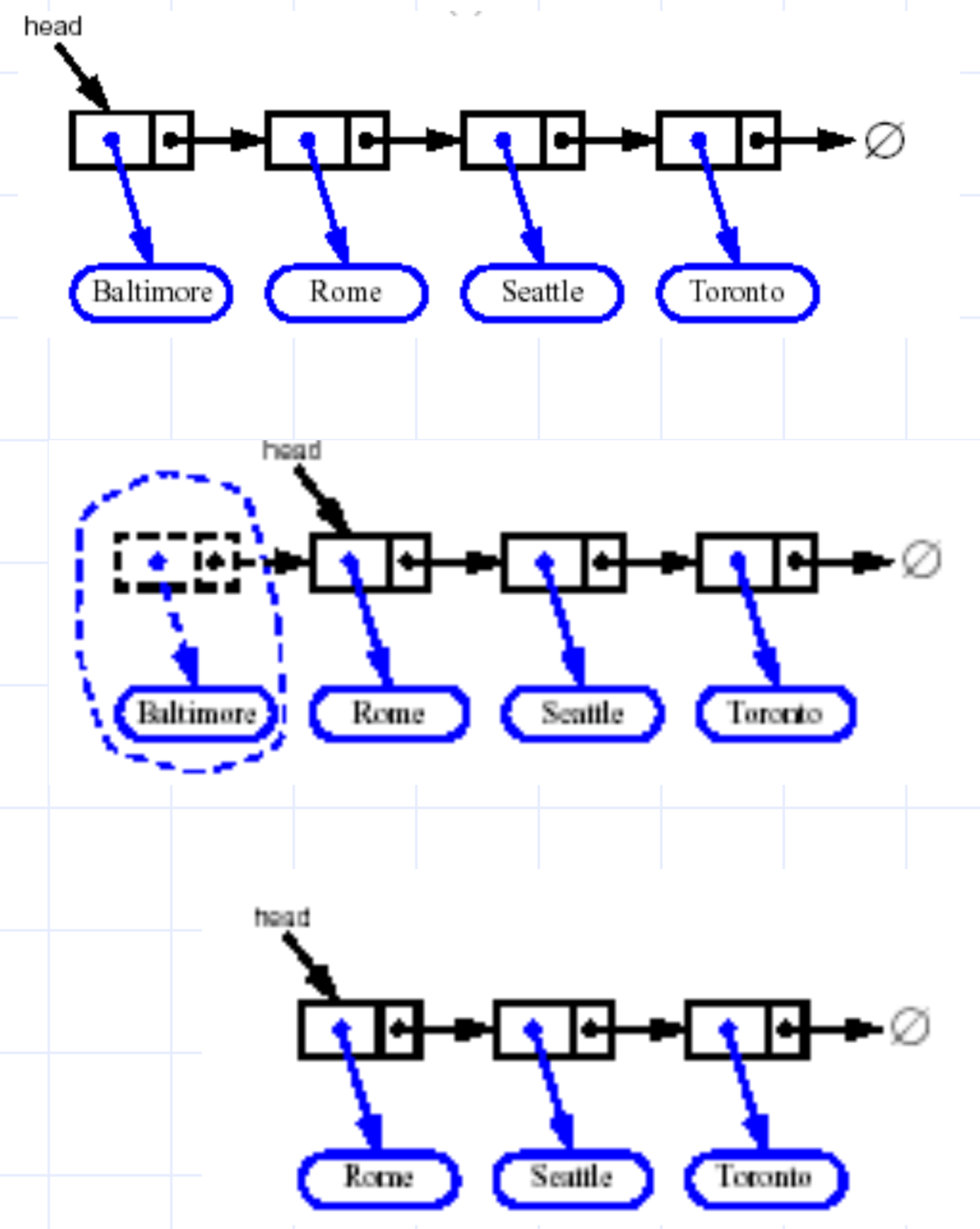
Inserting at the Head

1. Allocate a new node
2. Insert new element
3. Have new node point to old head
4. Update head to point to new node



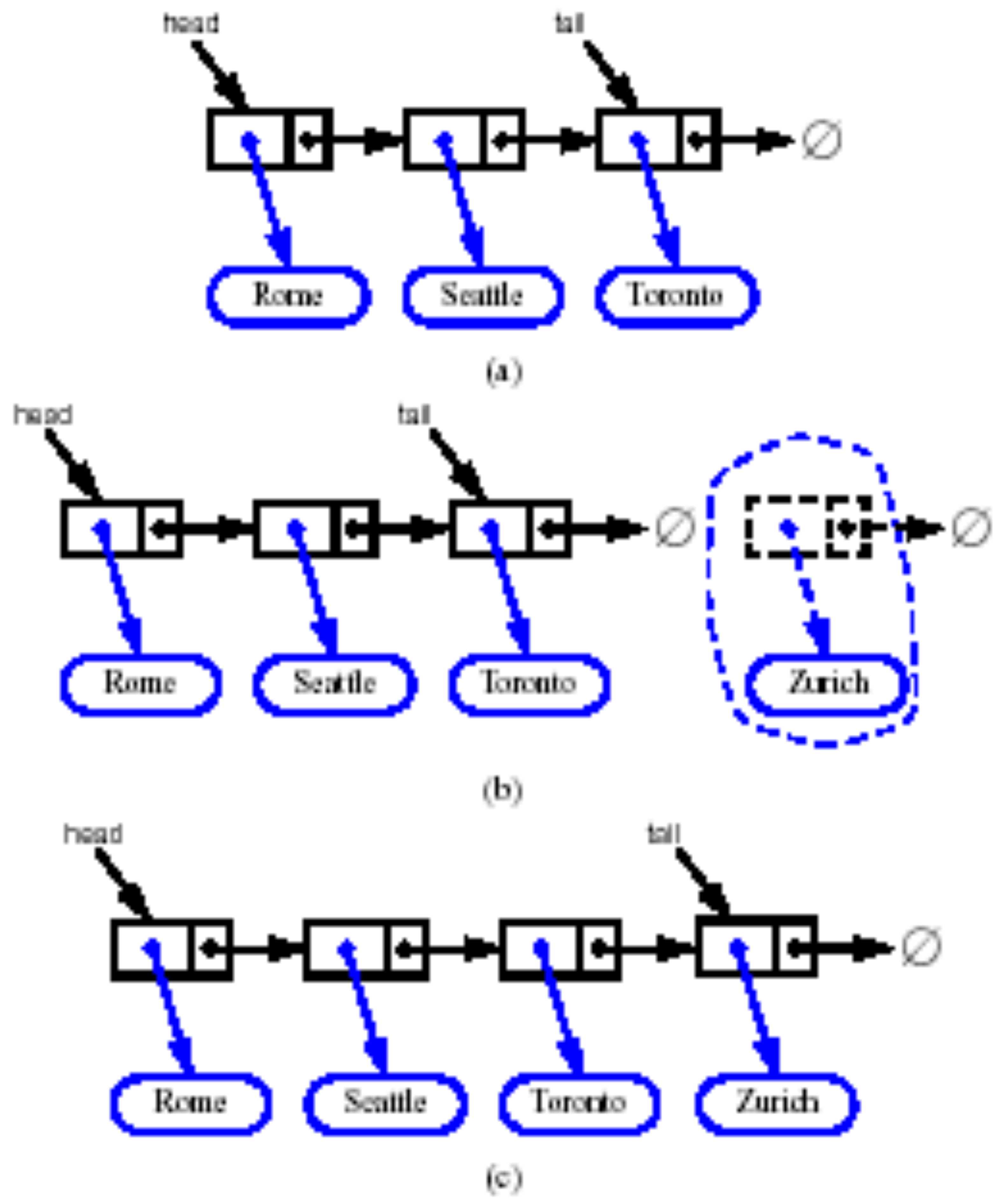
Removing at the Head

1. Update head to point to next node in the list
2. Allow garbage collector to reclaim the former first node



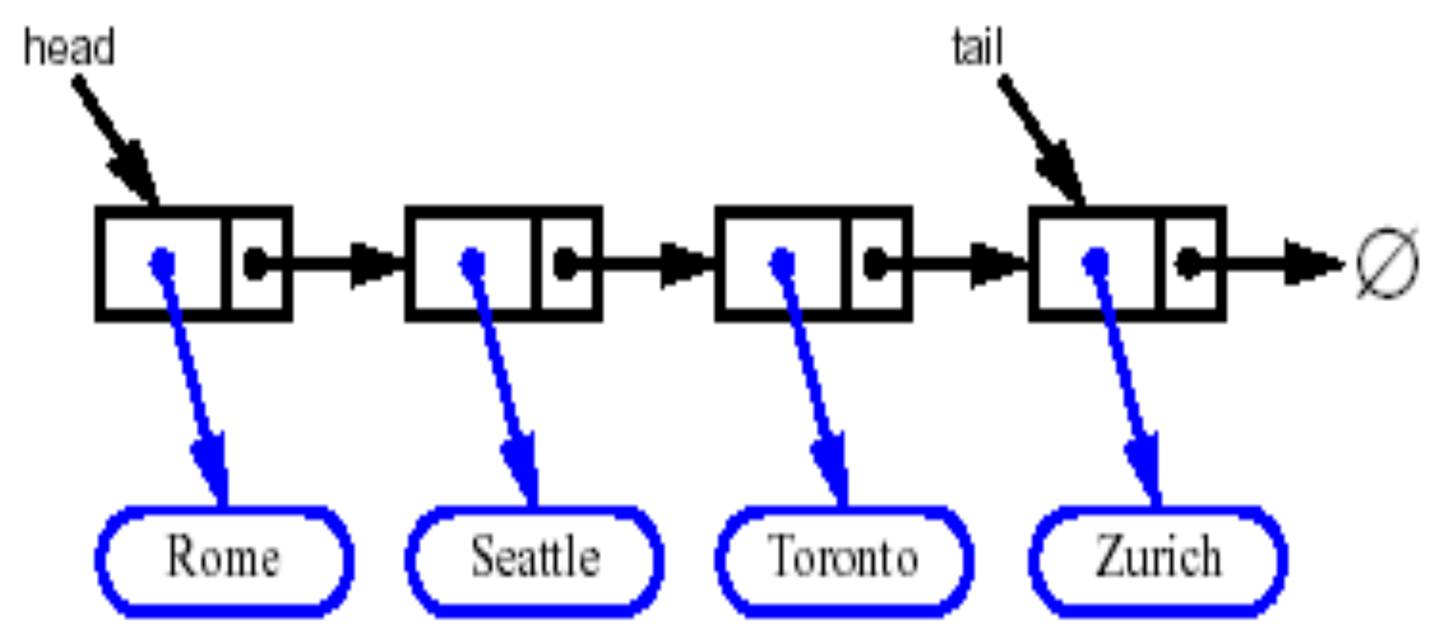
Inserting at the Tail

1. Allocate a new node
2. Insert new element
3. Have new node point to null
4. Have old last node point to new node
5. Update tail to point to new node



Removing at the Tail

- Removing at the tail of a singly linked list is not efficient!
- There is no constant-time way to update the tail to point to the previous node



Array v.s Linked List

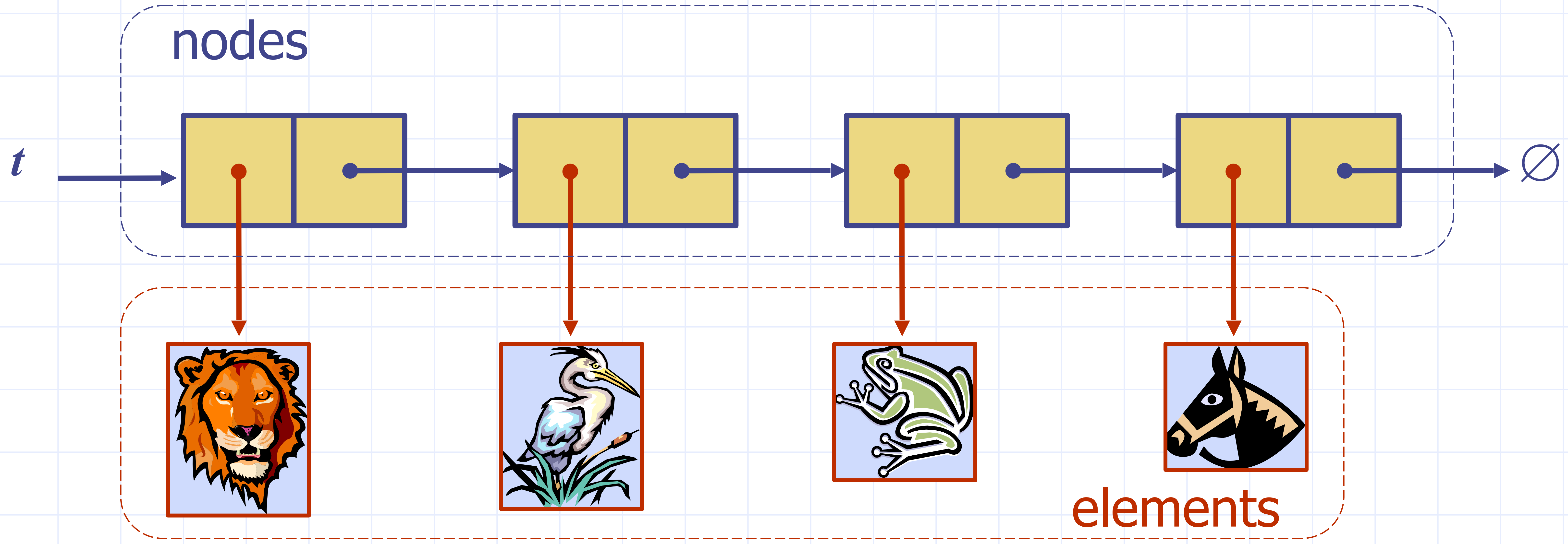
- Array disadvantages:
  - The length of a dynamic array might be longer than the actual number of elements that it stores.
  - Amortized bounds for operations may be unacceptable in real-time systems
  - Insertions and deletions at interior positions of an array are expensive ~O(n)

- Linked list:
  - Each node maintains a reference to its element and one or more references to neighboring nodes in order to collectively represent the linear order of the sequence
  - Elements of a linked list cannot be efficiently accessed by a numeric index k
  - Insertions and deletions at interior positions of a linked list are inexpensive ~O(1)

Stack as a Linked List

- We can implement a stack with a singly linked list
- The top element is stored at the first node of the list
- The space used is $O(n)$ and each operation of the Stack ADT takes $O(1)$ time

ADT = Abstract Data Type

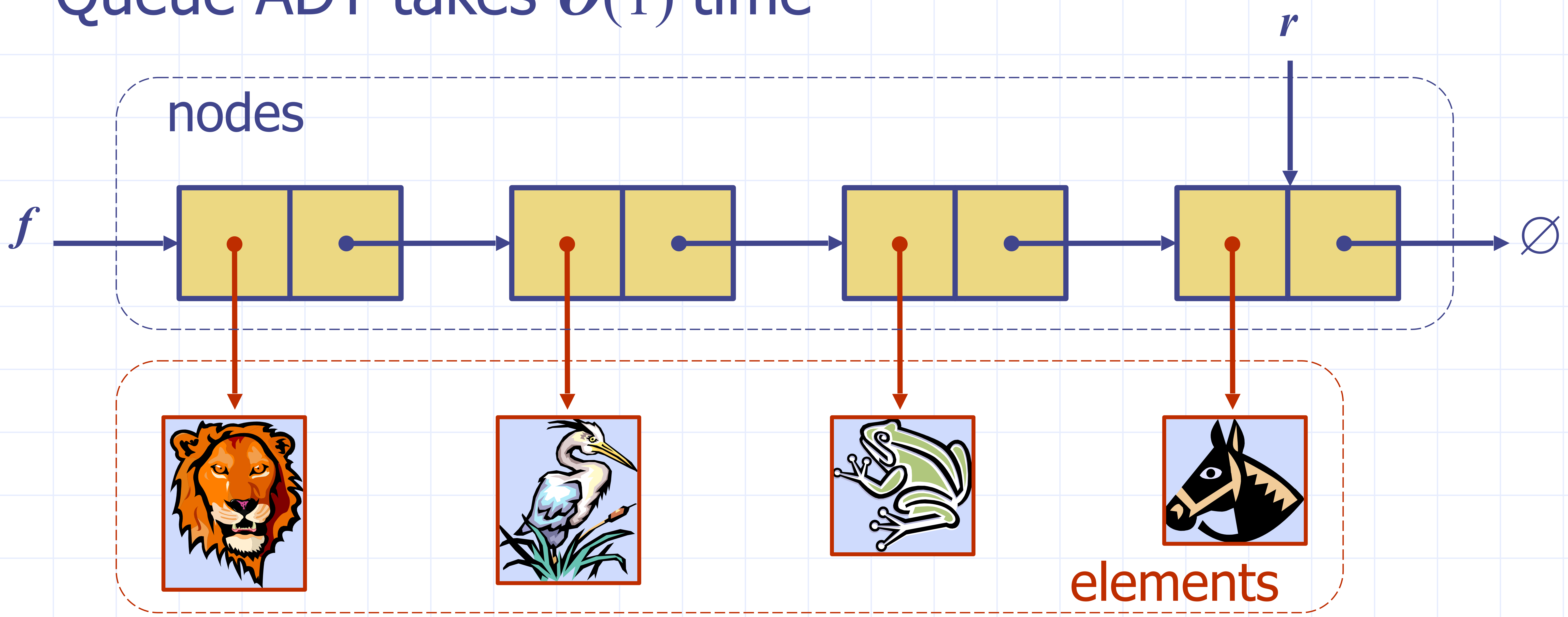


Linked-List Stack in Python

```python
class LinkedStack:
  """LIFO Stack implementation using a singly linked list for storage."""

  #-------------------------- nested _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

  #------------------------------- stack methods -------------------------------
  def __init__(self):
    """Create an empty stack."""
    self._head = None                       # reference to the head node
    self._size = 0                          # number of stack elements

  def __len__(self):
    """Return the number of elements in the stack."""
    return self._size

  def is_empty(self):
    """Return True if the stack is empty."""
    return self._size == 0

  def push(self, e):
    """Add element e to the top of the stack."""
    self._head = self._Node(e, self._head)      # create and link a new node
    self._size += 1

  def top(self):
    """Return (but do not remove) the element at the top of the stack.

    Raise Empty exception if the stack is empty.
    """
    if self.is_empty():
      raise Empty('Stack is empty')
    return self._head._element                  # top of stack is at head of list

  def pop(self):
    """Remove and return the element from the top of the stack (i.e., LIFO).

    Raise Empty exception if the stack is empty.
    """
    if self.is_empty():
      raise Empty('Stack is empty')
    answer = self._head._element
    self._head = self._head._next              # bypass the former top node
    self._size -= 1
    return answer
```

Queue as a Linked List

- We can implement a queue with a singly linked list
  - The front element is stored at the first node
  - The rear element is stored at the last node
- The space used is $O(n)$ and each operation of the Queue ADT takes $O(1)$ time

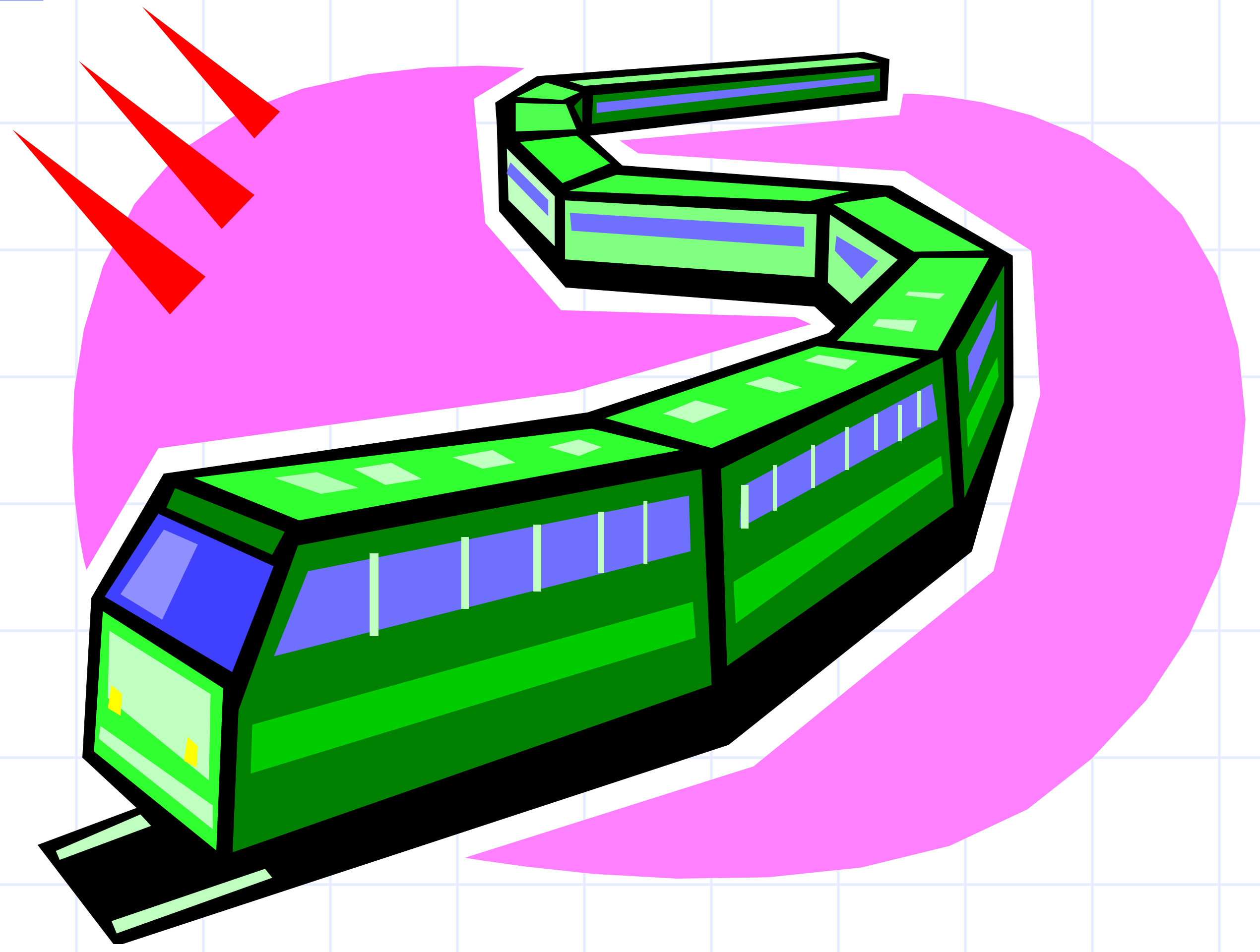


Linked-List Queue in Python

```
class LinkedQueue:
  """FIFO queue implementation using a singly linked list for storage."""

  class _Node:
    """Lightweight, nonpublic class for storing a singly linked node."""
    (omitted here; identical to that of LinkedStack._Node)

  def __init__(self):
    """Create an empty queue."""
    self._head = None
    self._tail = None
    self._size = 0                          # number of queue elements

  def __len__(self):
    """Return the number of elements in the queue."""
    return self._size

  def is_empty(self):
    """Return True if the queue is empty."""
    return self._size == 0

  def first(self):
    """Return (but do not remove) the element at the front of the queue."""
    if self.is_empty():
      raise Empty('Queue is empty')
    return self._head._element              # front aligned with head of list
```

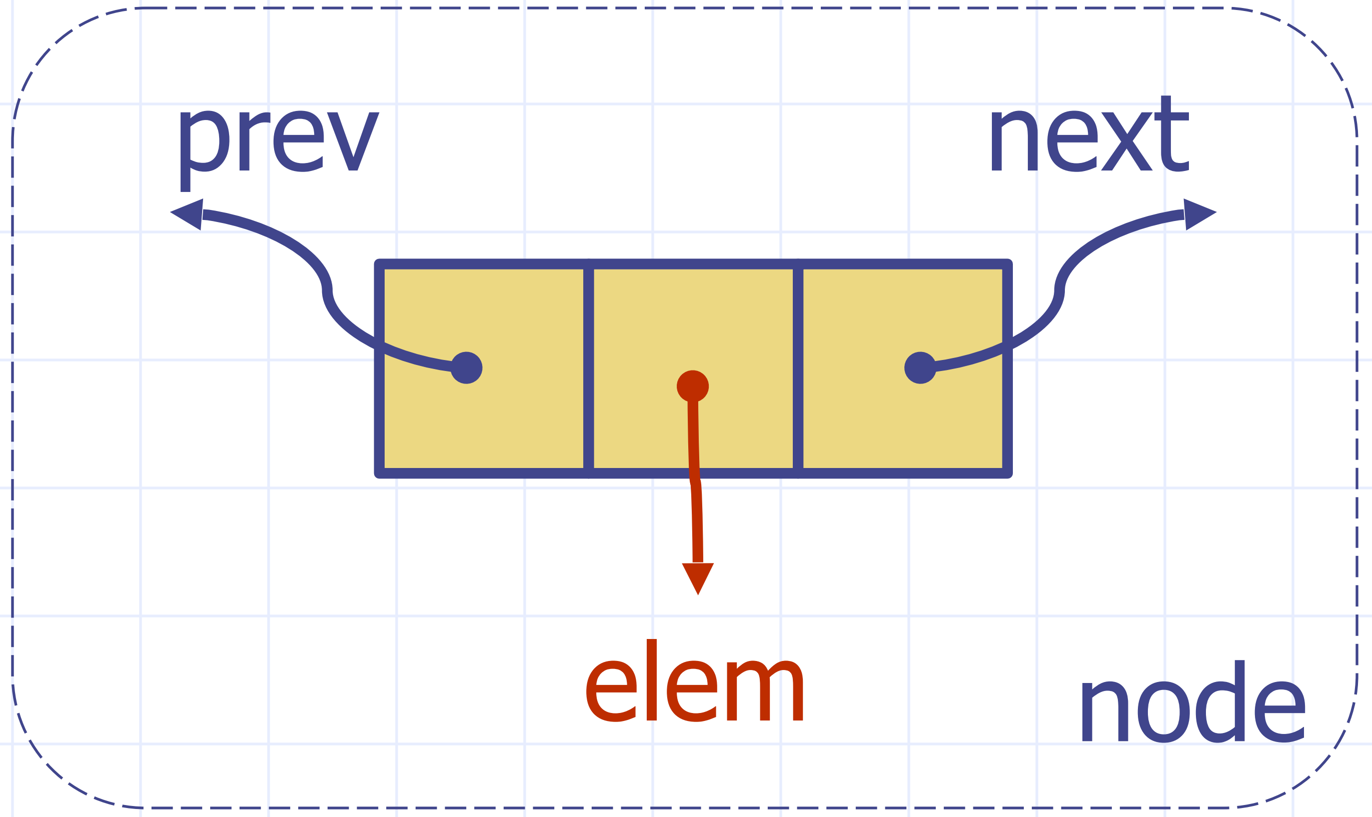
```
  def dequeue(self):
    """Remove and return the first element of the queue (i.e., FIFO).

    Raise Empty exception if the queue is empty.
    """
    if self.is_empty():
      raise Empty('Queue is empty')
    answer = self._head._element
    self._head = self._head._next
    self._size -= 1
    if self.is_empty():                     # special case as queue is empty
      self._tail = None                     # removed head had been the tail
    return answer

  def enqueue(self, e):
    """Add an element to the back of queue."""
    newest = self._Node(e, None)            # node will be new tail node
    if self.is_empty():
      self._head = newest                   # special case: previously empty
    else:
      self._tail._next = newest
    self._tail = newest                     # update reference to tail node
    self._size += 1
```

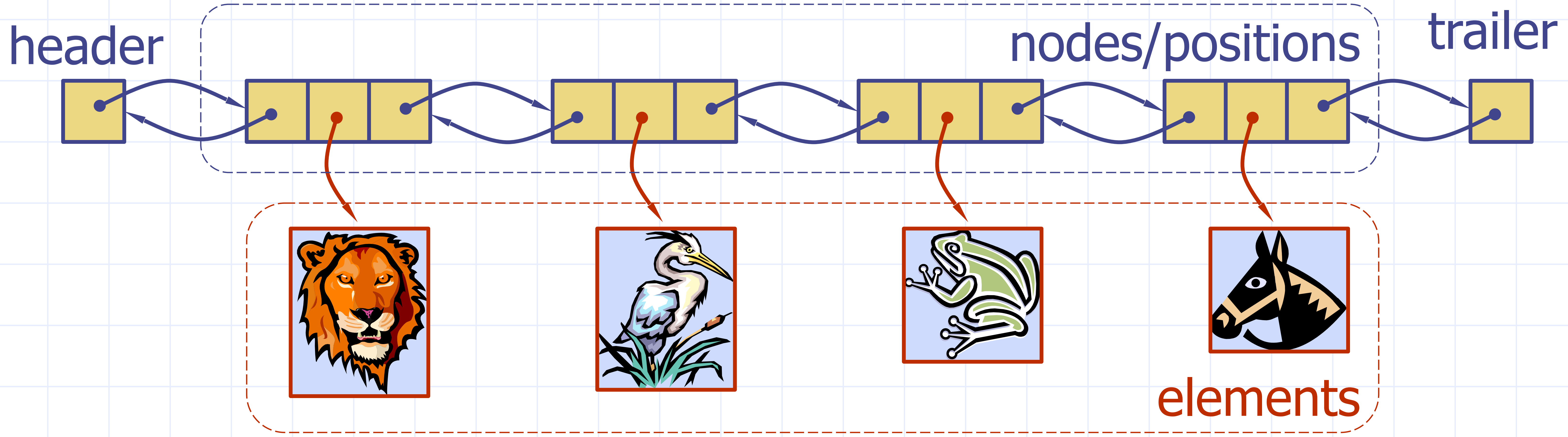
Doubly-Linked Lists



Doubly Linked List

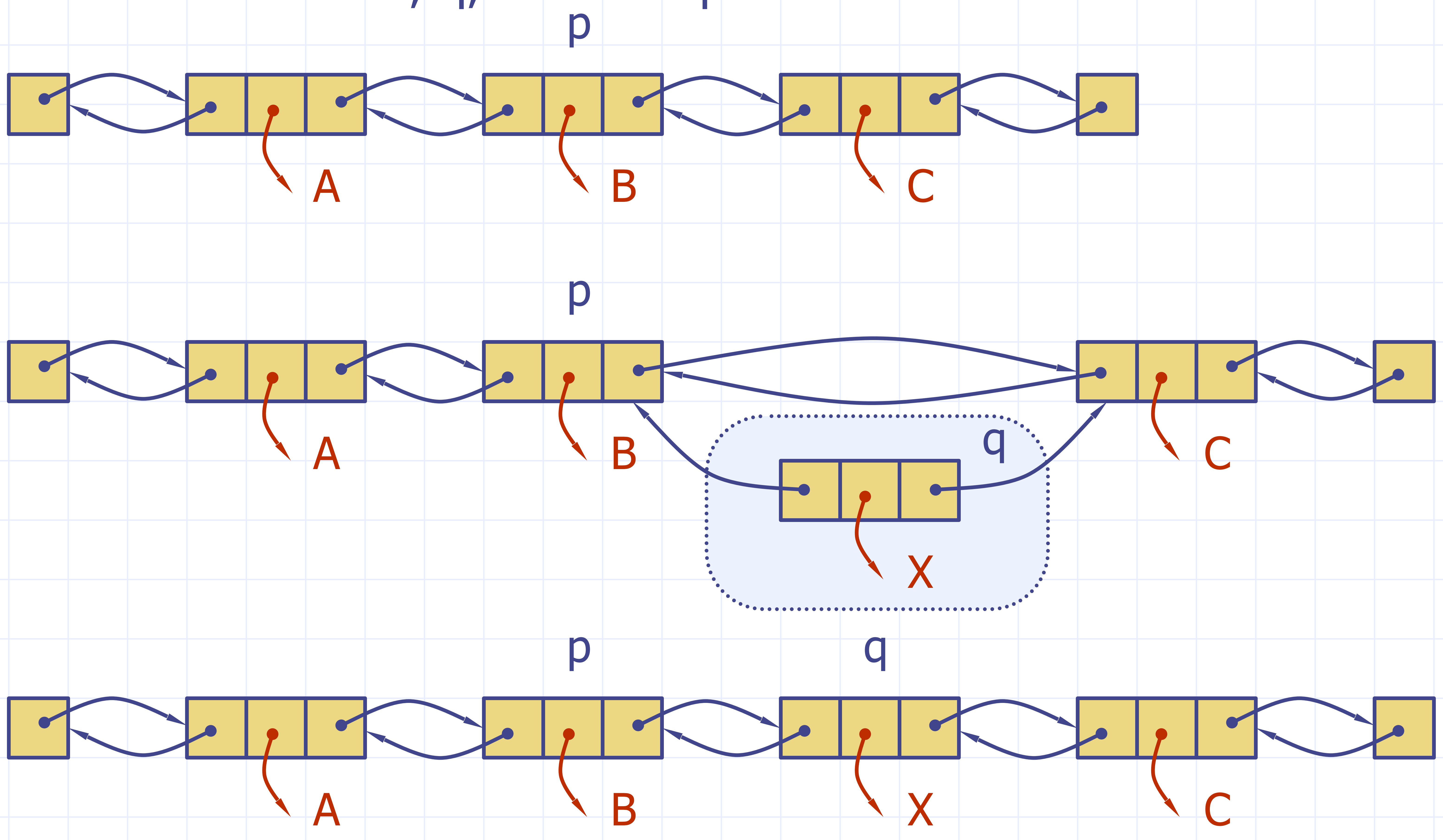
- A doubly linked list provides a natural implementation of the Node List ADT
- Nodes implement Position and store:
  - element
  - link to the previous node
  - link to the next node
- Special trailer and header nodes





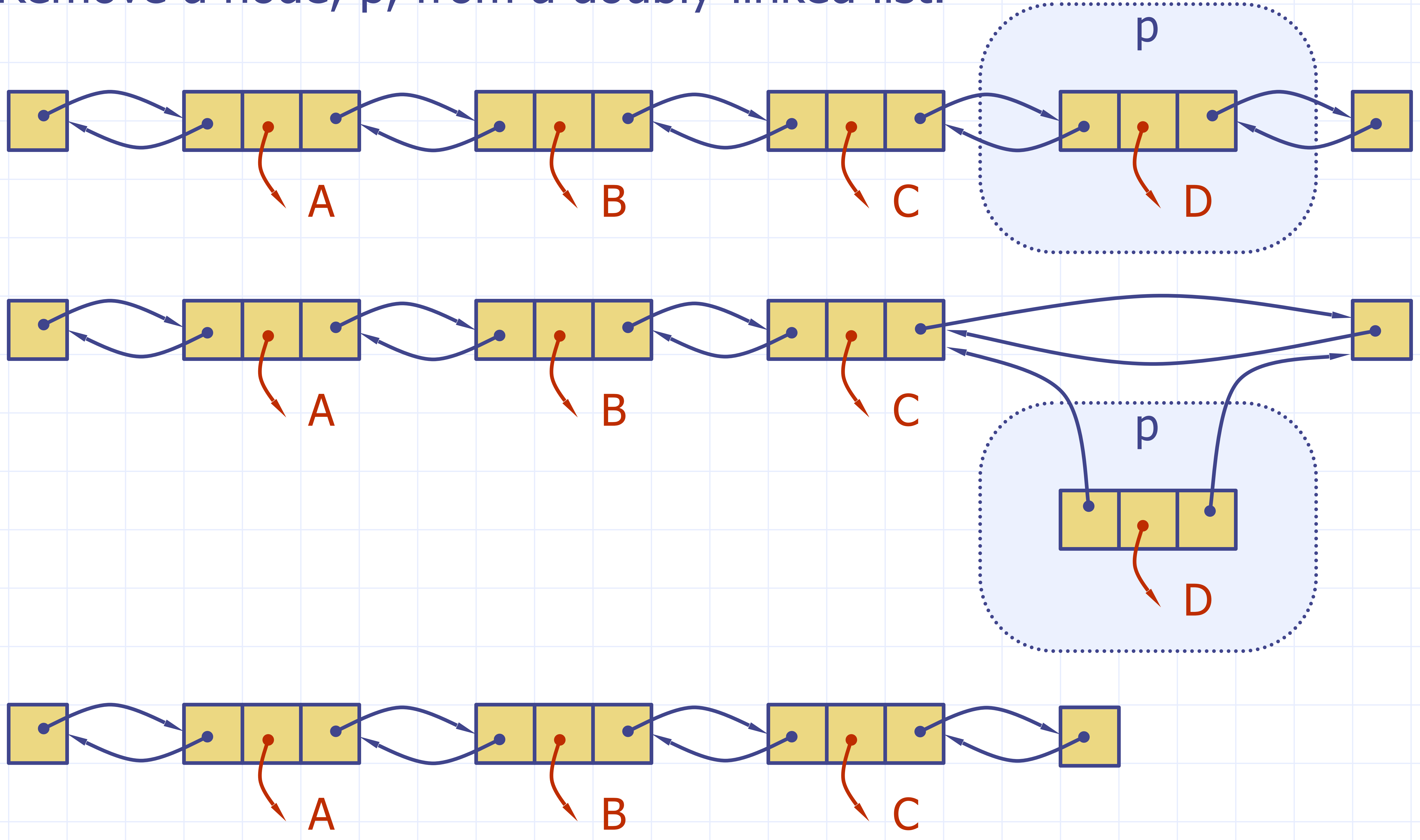
Insertion

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



Deletion

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

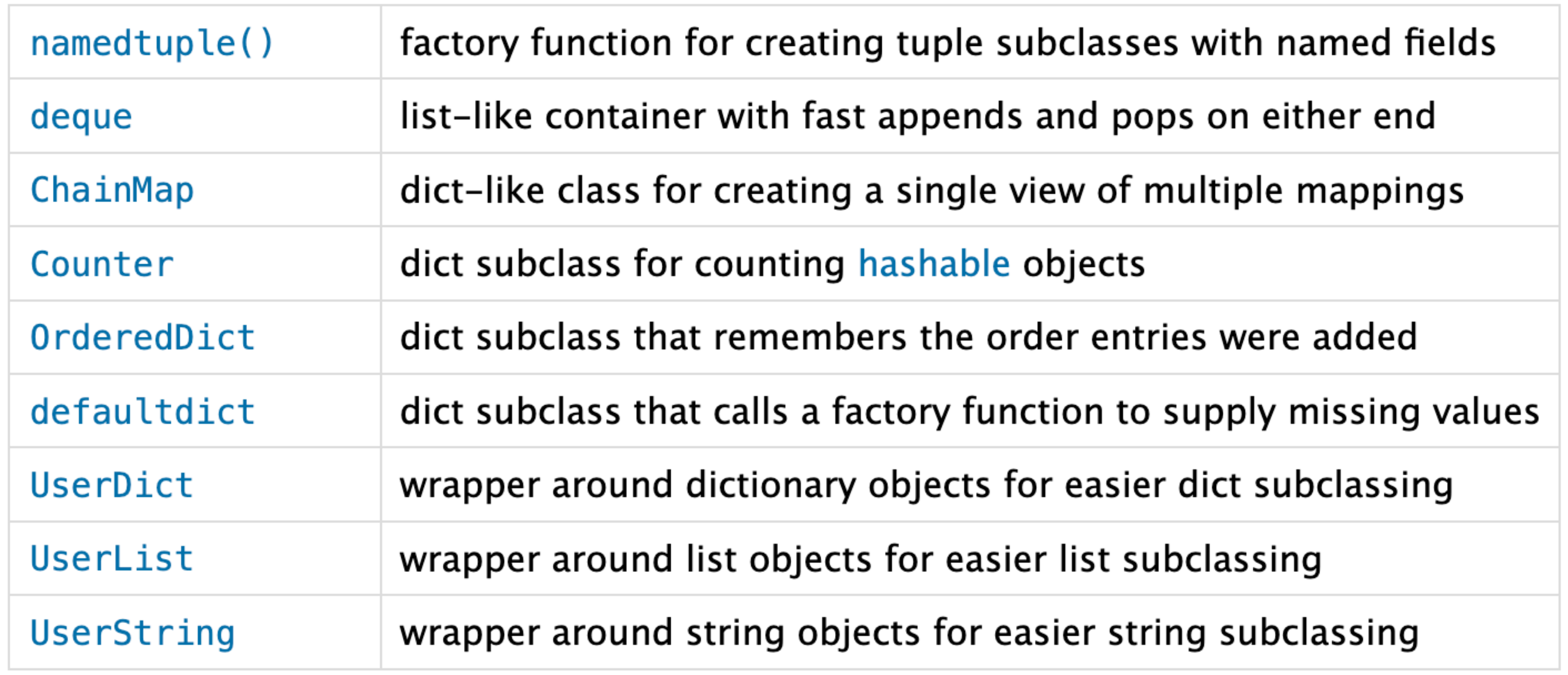


Performance

- In a doubly linked list
  - The space used by a list with n elements is $O(n)$
  - The space used by each position of the list is $O(1)$
  - All the standard operations of a list run in $O(1)$ time

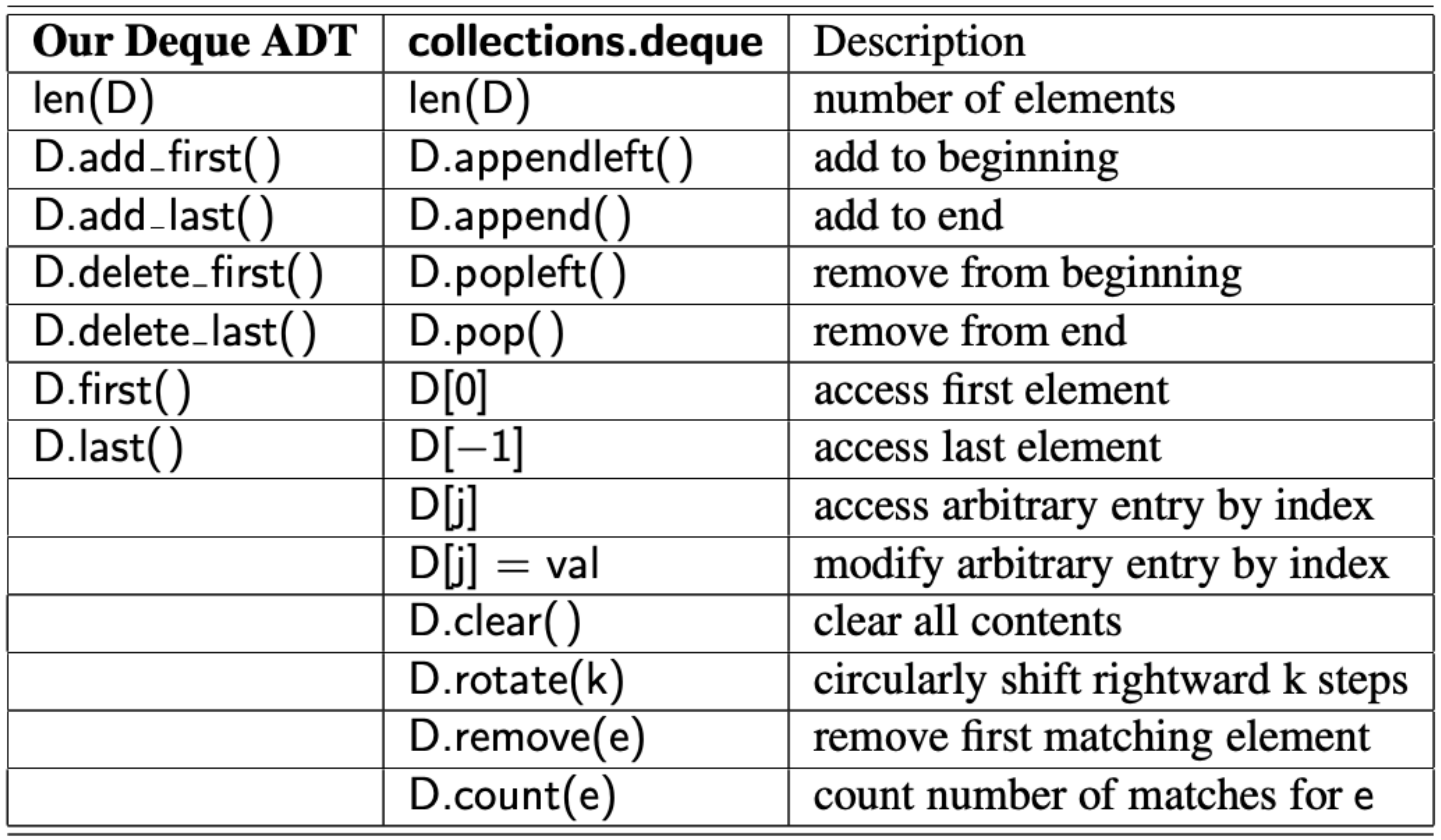
collections Module

- collections — Container datatypes
- implements specialized container datatypes providing alternatives to Python’s general purpose built-in containers, dict, list, set, and tuple



`namedtuple()`	factory function for creating tuple subclasses with named fields
`deque`	list-like container with fast appends and pops on either end
`ChainMap`	dict-like class for creating a single view of multiple mappings
`Counter`	dict subclass for counting hashable objects
`OrderedDict`	dict subclass that remembers the order entries were added
`defaultdict`	dict subclass that calls a factory function to supply missing values
`UserDict`	wrapper around dictionary objects for easier dict subclassing
`UserList`	wrapper around list objects for easier list subclassing
`UserString`	wrapper around string objects for easier string subclassing

Comparison of the deque ADT and the collections.deque class



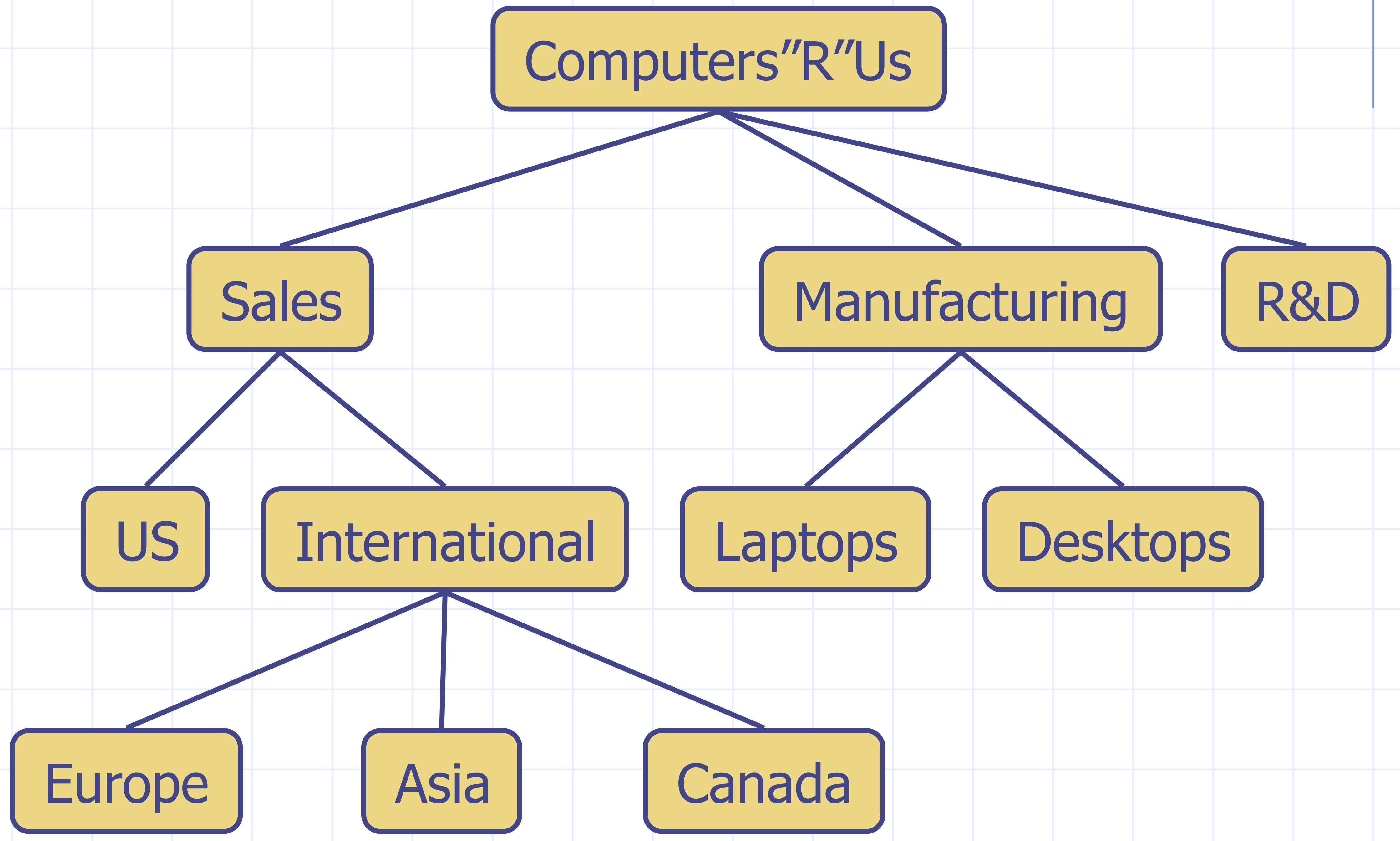
Our Deque ADT	**collections.deque**	Description
len(D)	len(D)	number of elements
D.add_first()	D.appendleft()	add to beginning
D.add_last()	D.append()	add to end
D.delete_first()	D.popleft()	remove from beginning
D.delete_last()	D.pop()	remove from end
D.first()	D[0]	access first element
D.last()	D[−1]	access last element
	D[j]	access arbitrary entry by index
	D[j] = val	modify arbitrary entry by index
	D.clear()	clear all contents
	D.rotate(k)	circularly shift rightward k steps
	D.remove(e)	remove first matching element
	D.count(e)	count number of matches for e

Trees and Tree Traversal Algorithms

General Trees

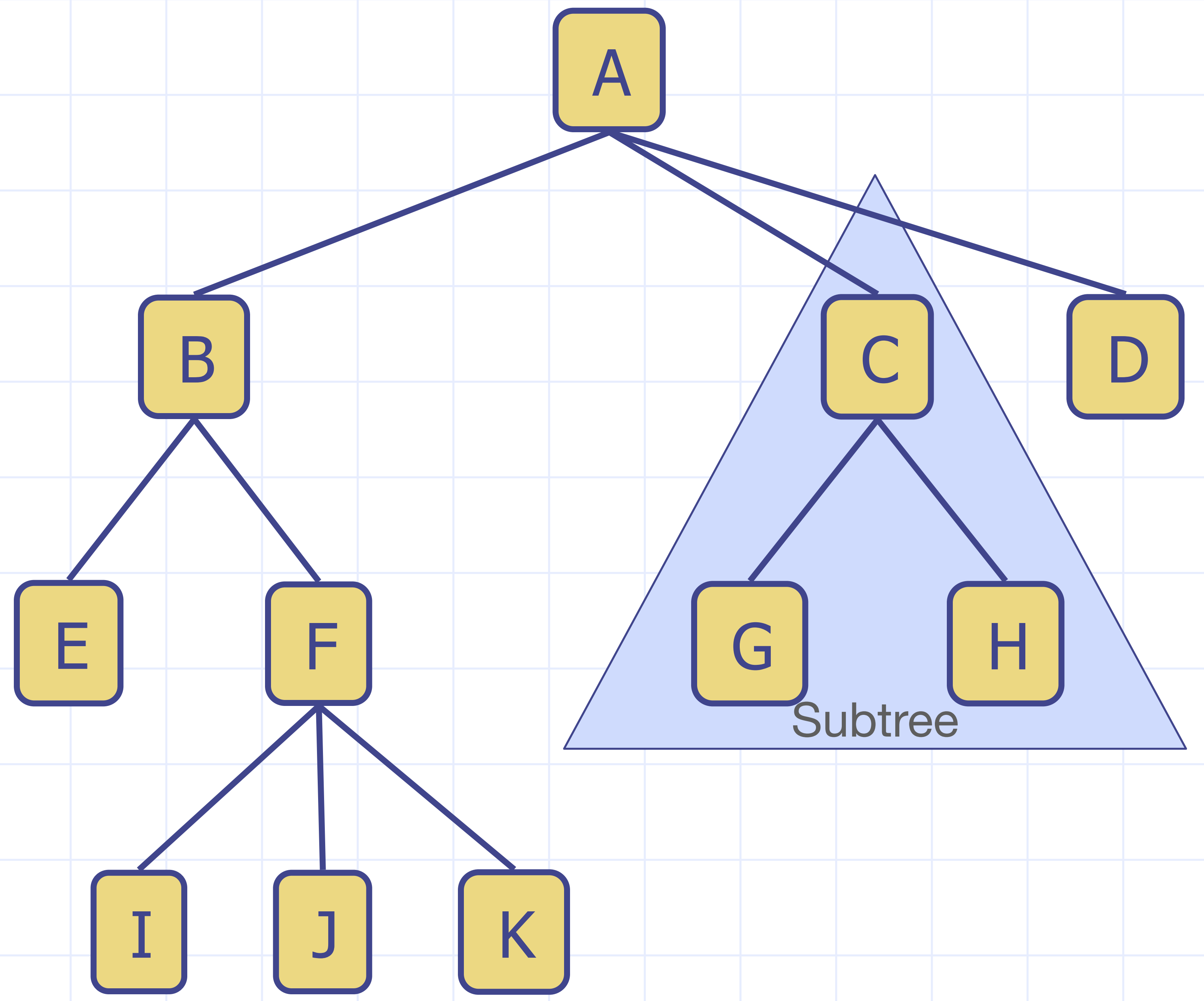
What is a Tree

- In computer science, a tree is an abstract model of a hierarchical structure
- A tree T consists of set of **nodes** with a **parent-child relation**:
  - The **root** - special node without a parent
  - Each other node has a unique parent node; every node with a parent is a child of that parent node
- Applications:
  - Organization charts
  - File systems
  - Programming environments



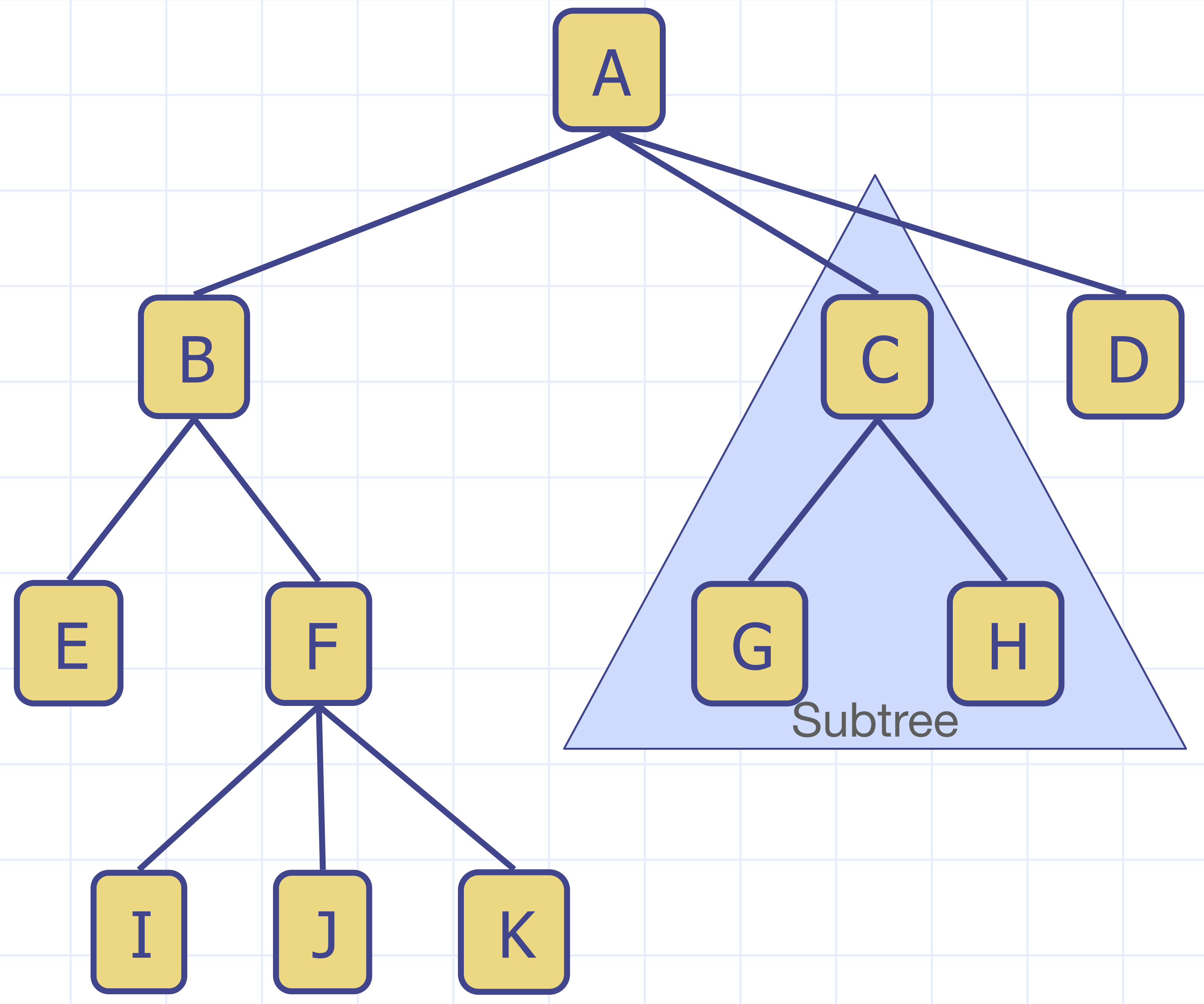
Tree Terminology

- **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 (?)
- **Descendant of a node**: child, grandchild, grand-grandchild, etc.
- **Subtree**: tree consisting of a node and its descendants



Tree Terminology

- **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.
- **Subtree**: tree consisting of a node and its descendants



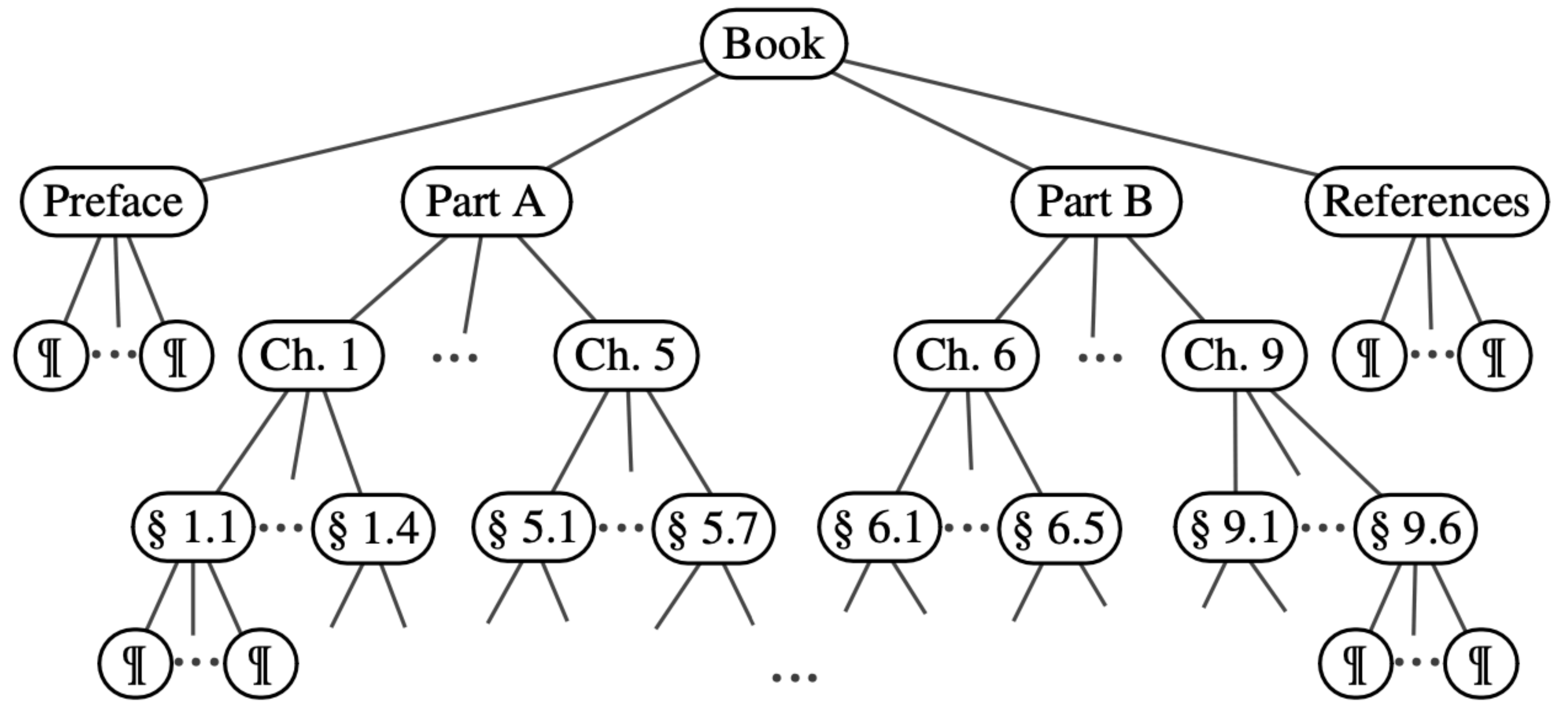
Tree ADT

- We use positions to abstract nodes
- Generic methods:
  - Integer len()
  - Boolean is_empty()
  - Iterator positions()
  - Iterator iter()
- Accessor methods:
  - position root()
  - position parent(p)
  - Iterator children(p)
  - Integer num_children(p)
- Query methods:
  - Boolean is_leaf(p)
  - Boolean is_root(p)
- Update method:
  - element replace (p, o)
- Additional update methods may be defined by data structures implementing the Tree ADT

Preorder Traversal

- A traversal visits the nodes of a tree in a systematic manner
- In a preorder traversal, a node is visited before its descendants
- Application: print a structured document

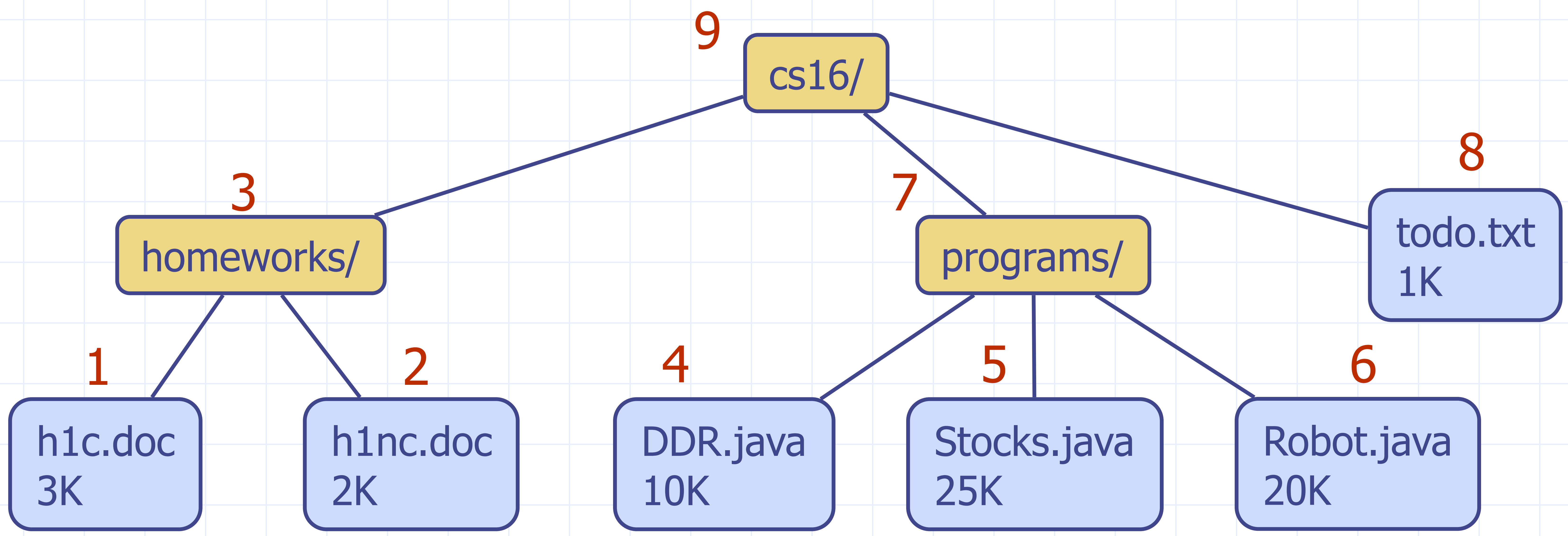
```
Algorithm preOrder(v)
    visit(v)
    for each child w of v
        preorder (w)
```



Postorder Traversal

- In a postorder traversal, a node is visited after its descendants
- Application: compute space used by files in a directory and its subdirectories

```
Algorithm postOrder(v)
   for each child w of v
         postOrder (w)
   visit(v)
```

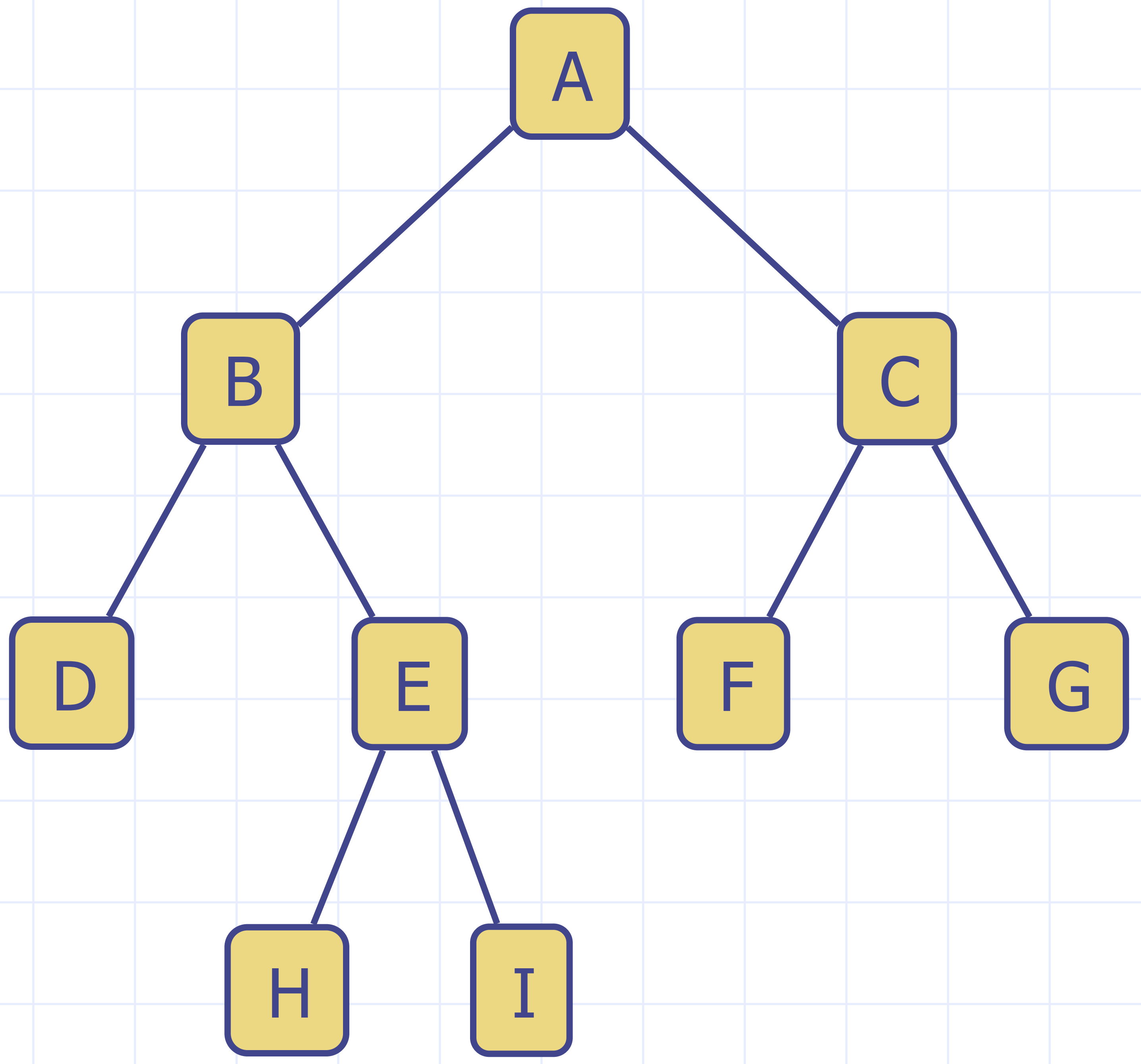


Binary Trees

Binary Trees

- **A binary tree** is a tree with the following properties:
  - Each internal node has at most two children:
    - Exactly two for **proper binary trees**
    - **Improper**
  - 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

- Applications:
  - arithmetic expressions
  - decision processes
  - searching



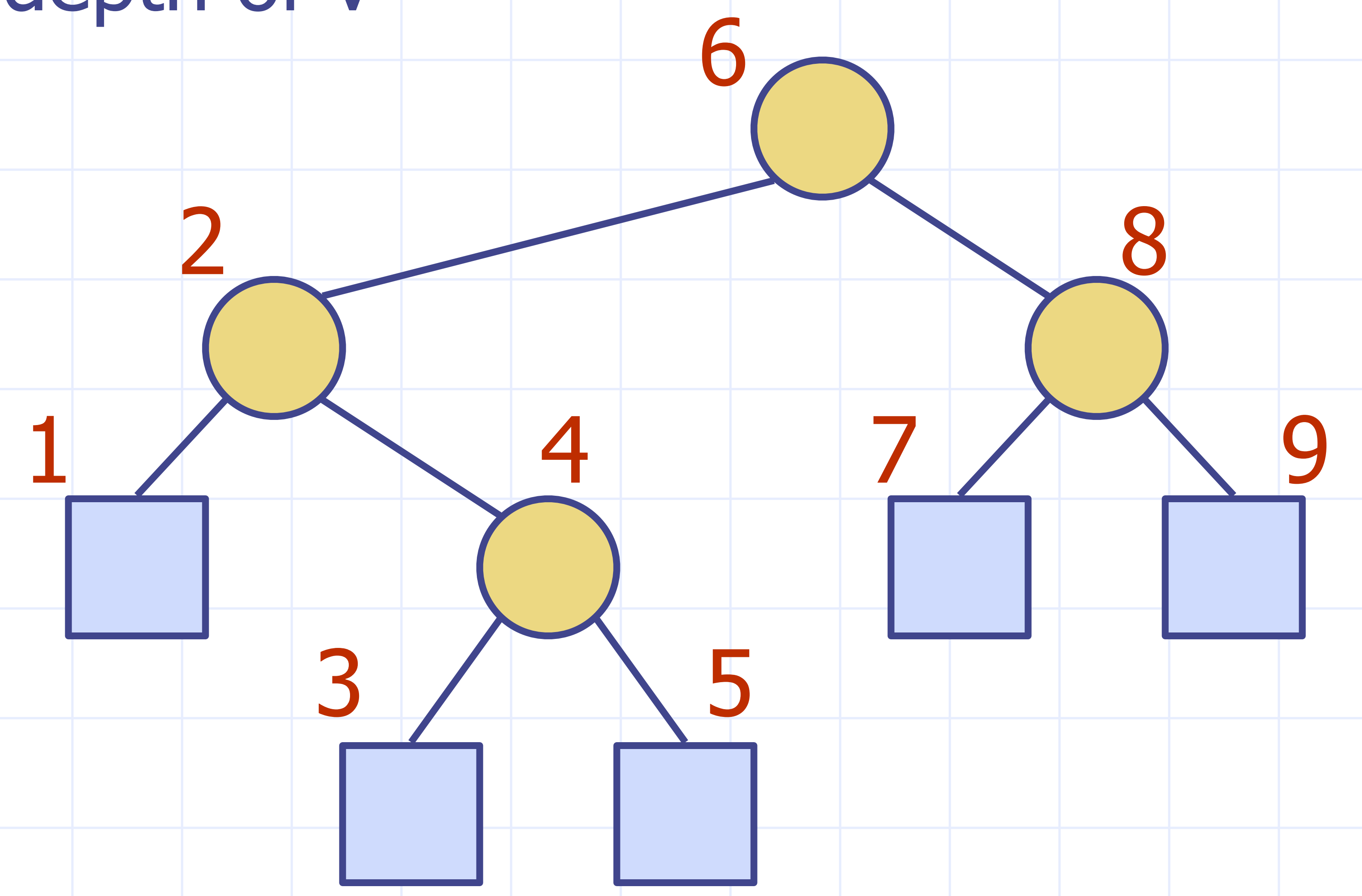
BinaryTree ADT

- The BinaryTree ADT extends the Tree ADT, i.e., it inherits all the methods of the Tree ADT
- Additional methods:
  - position left(p)
  - position right(p)
  - position sibling(p)
- Update methods may be defined by data structures implementing the BinaryTree ADT

Inorder Traversal

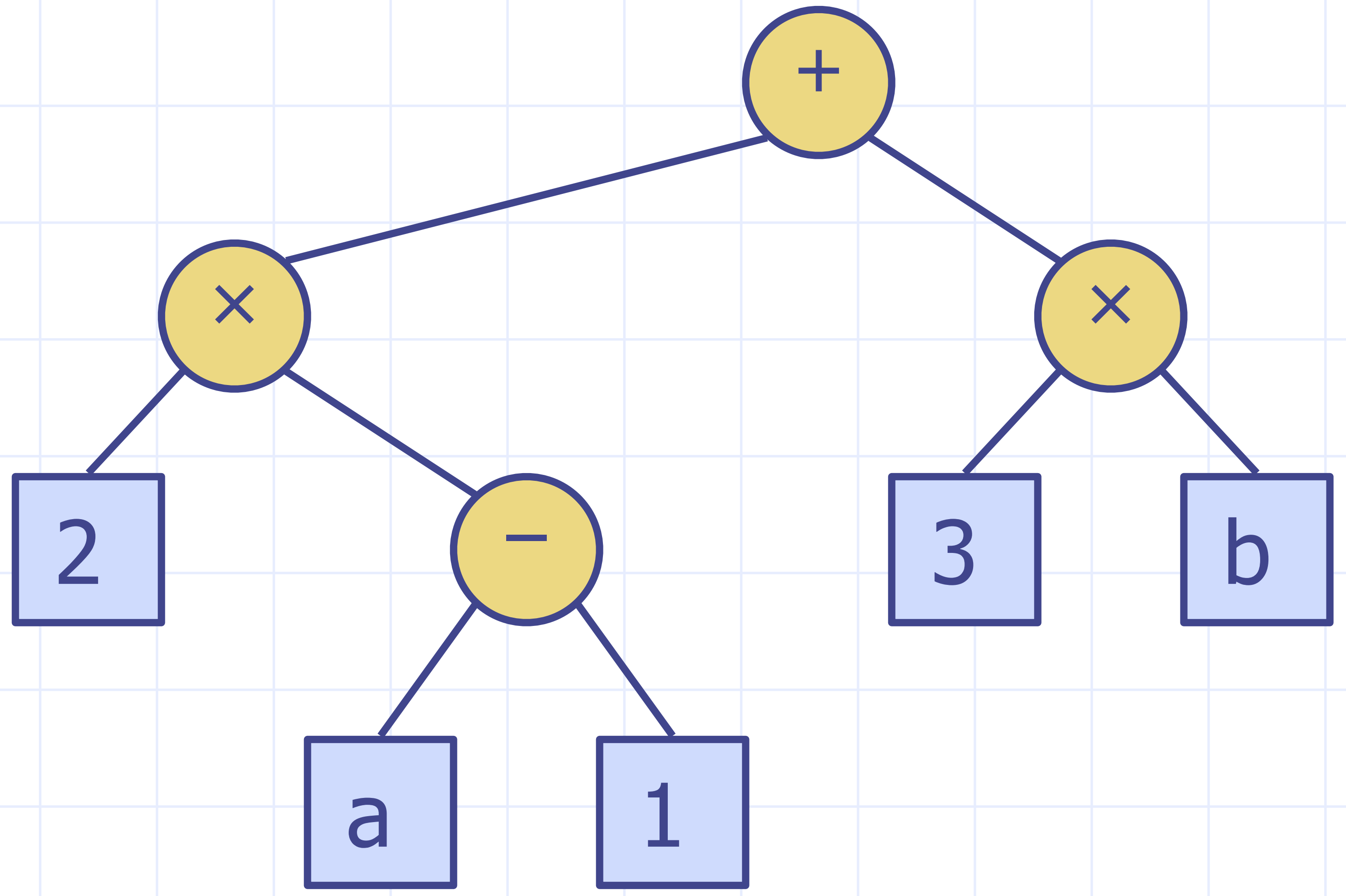
- In an inorder traversal a node is visited after its left subtree and before its right subtree
- Application: draw a binary tree
  - x(v) = inorder rank of v
  - y(v) = depth of v

```
Algorithm inOrder(v)
    if v has a left child
        inOrder (left (v))
    visit(v)
    if v has a right child
        inOrder (right (v))
```

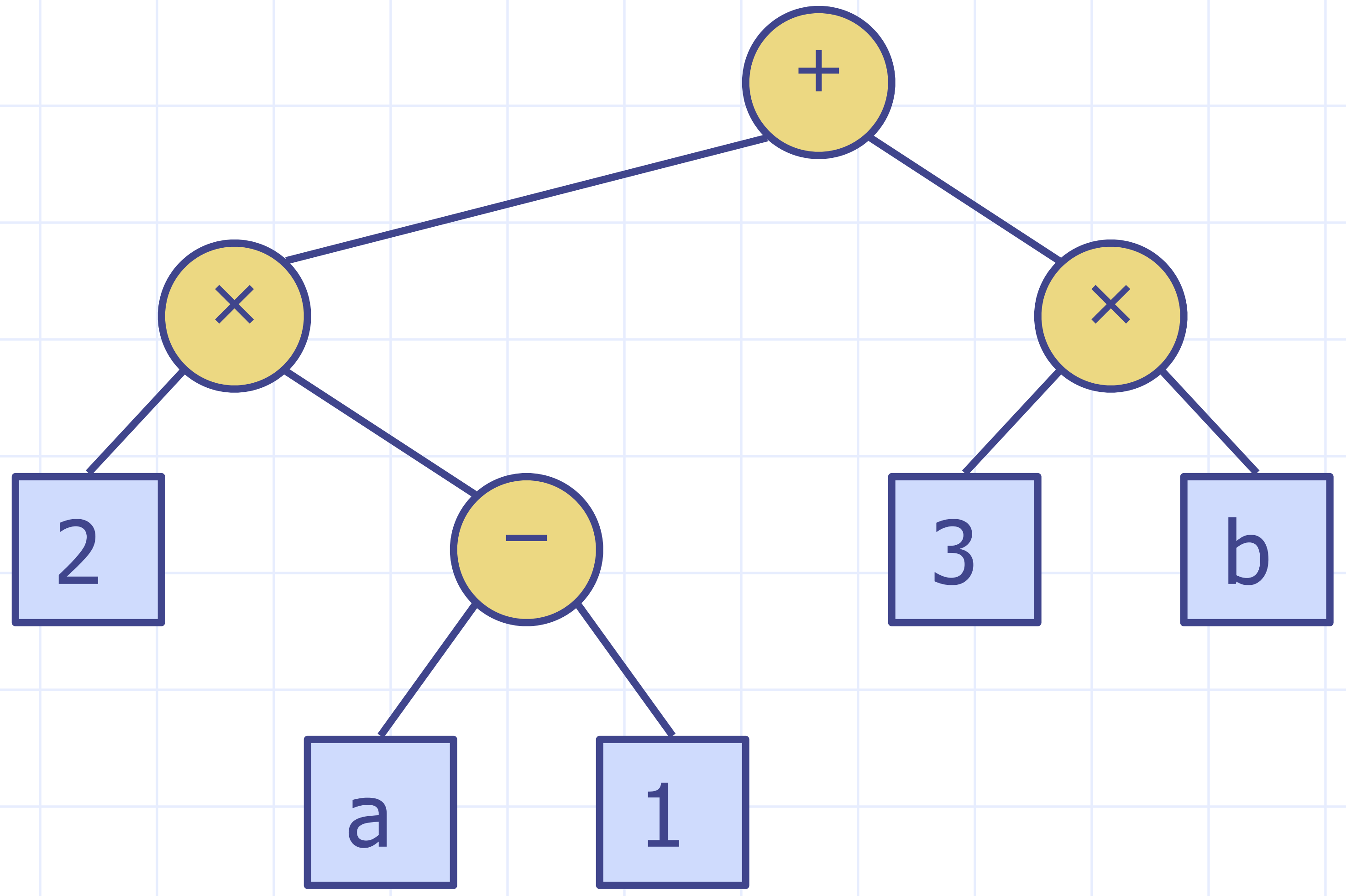


Question #1

- What type of tree traversal is relevant for this case?
  - A: pre-order: parent node evaluate, left, right child
  - B: In-order: evaluate left child-parent-right child
  - C: Post-order: evaluate left and right child, parent
- Example: arithmetic expression tree for expression: $(2 \times (a - 1) + (3 \times b))$
- Write out the resulting expression for each of the traversals

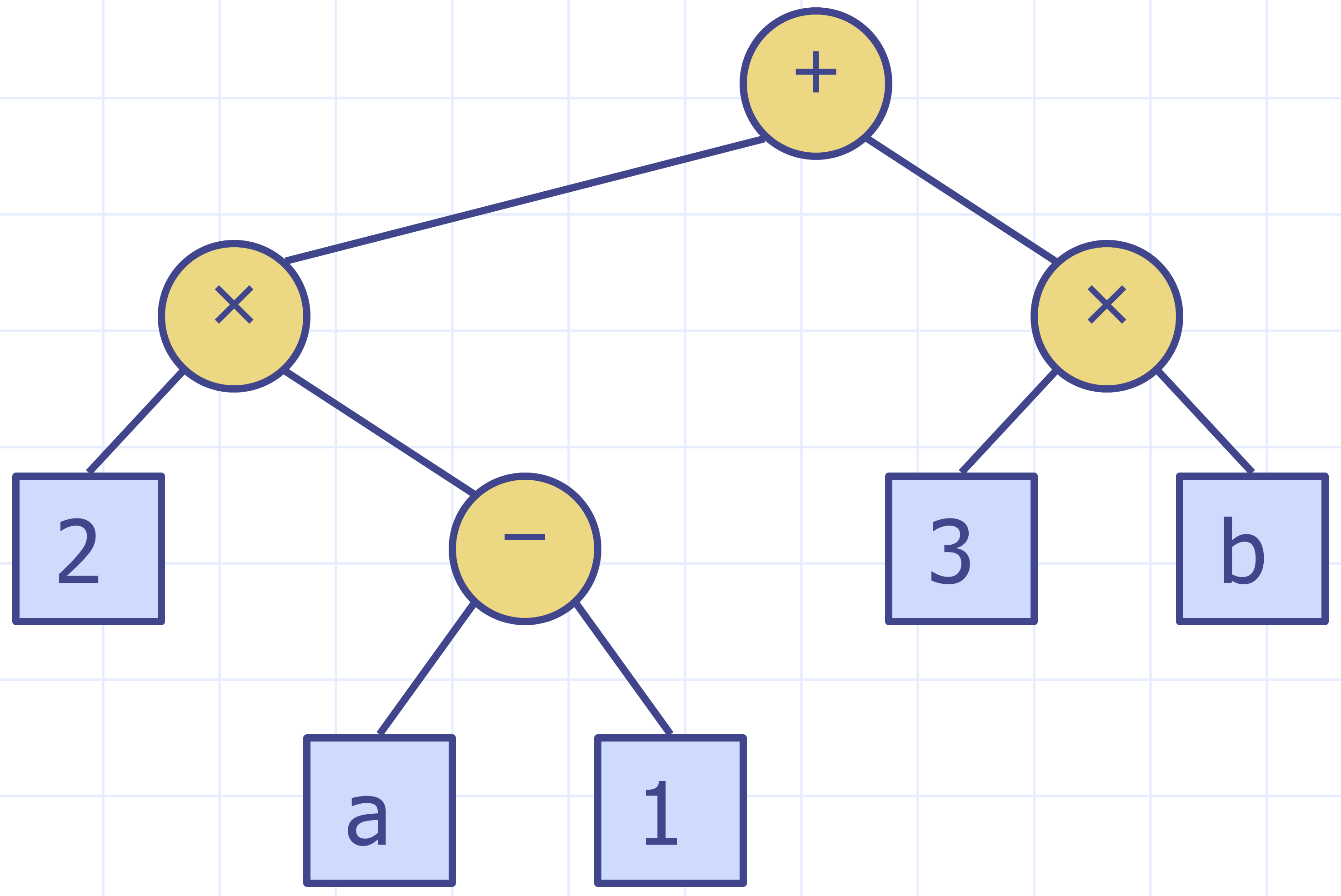


Answer #1



Arithmetic Expression Tree

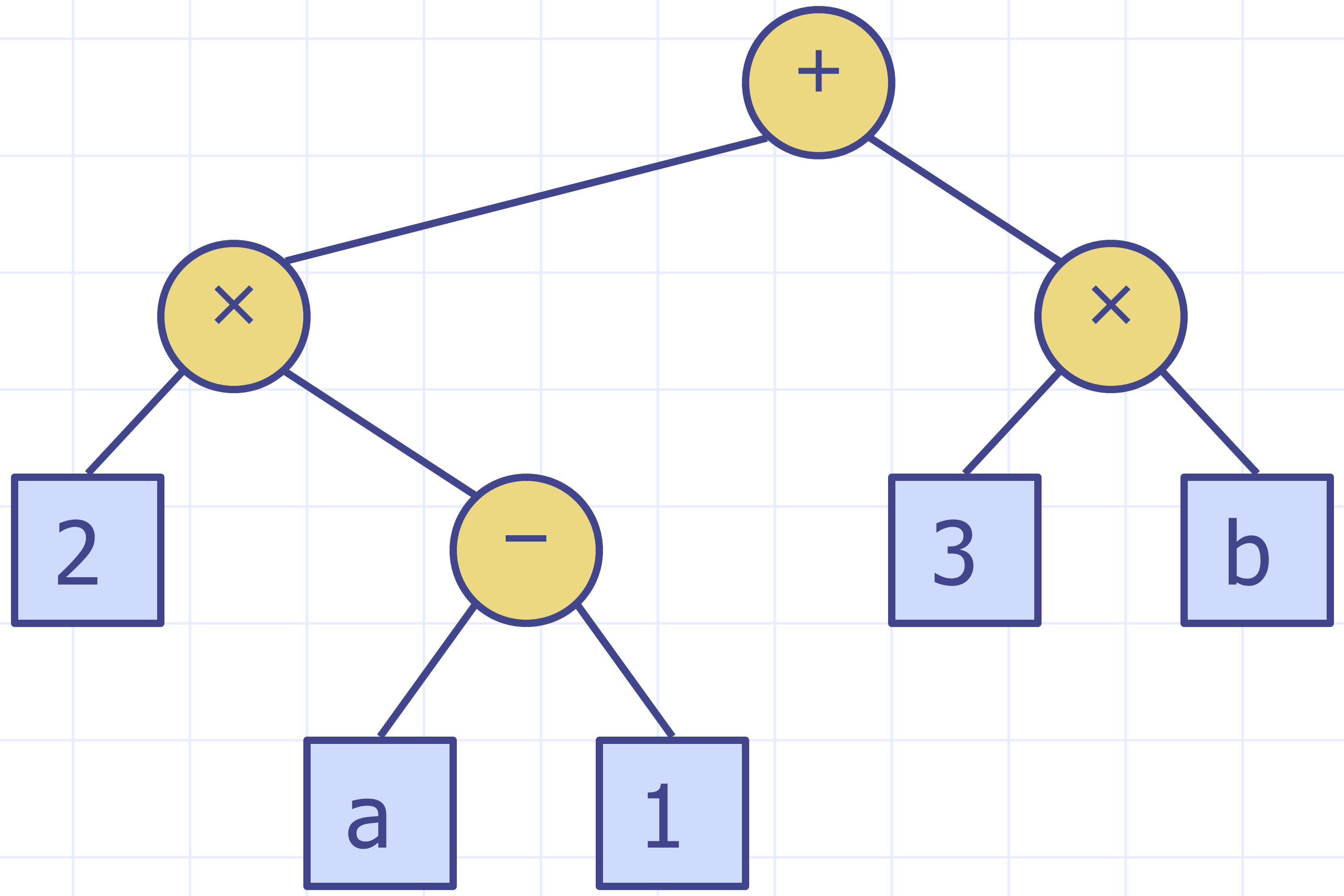
- Binary tree associated with an arithmetic expression
  - internal nodes:
    - operators
    - Node value: defined by applied operations on children
  - external nodes, leaves:
    - operands
    - Leaf value: numbers or variables
- Example: arithmetic expression tree for the expression $(2 \times (a - 1) + (3 \times b))$



Print Arithmetic Expressions

- Specialization of an inorder traversal
  - print operand or operator when visiting node
  - print "(" before traversing left subtree
  - print ")" after traversing right subtree
  - **Tree is represented by its root**

```
Algorithm printExpression(v)
    if v has a left child
            print("(")
        inOrder (left(v))
    print(v.element ())
    if v has a right child
        inOrder (right(v))
            print (")")
```

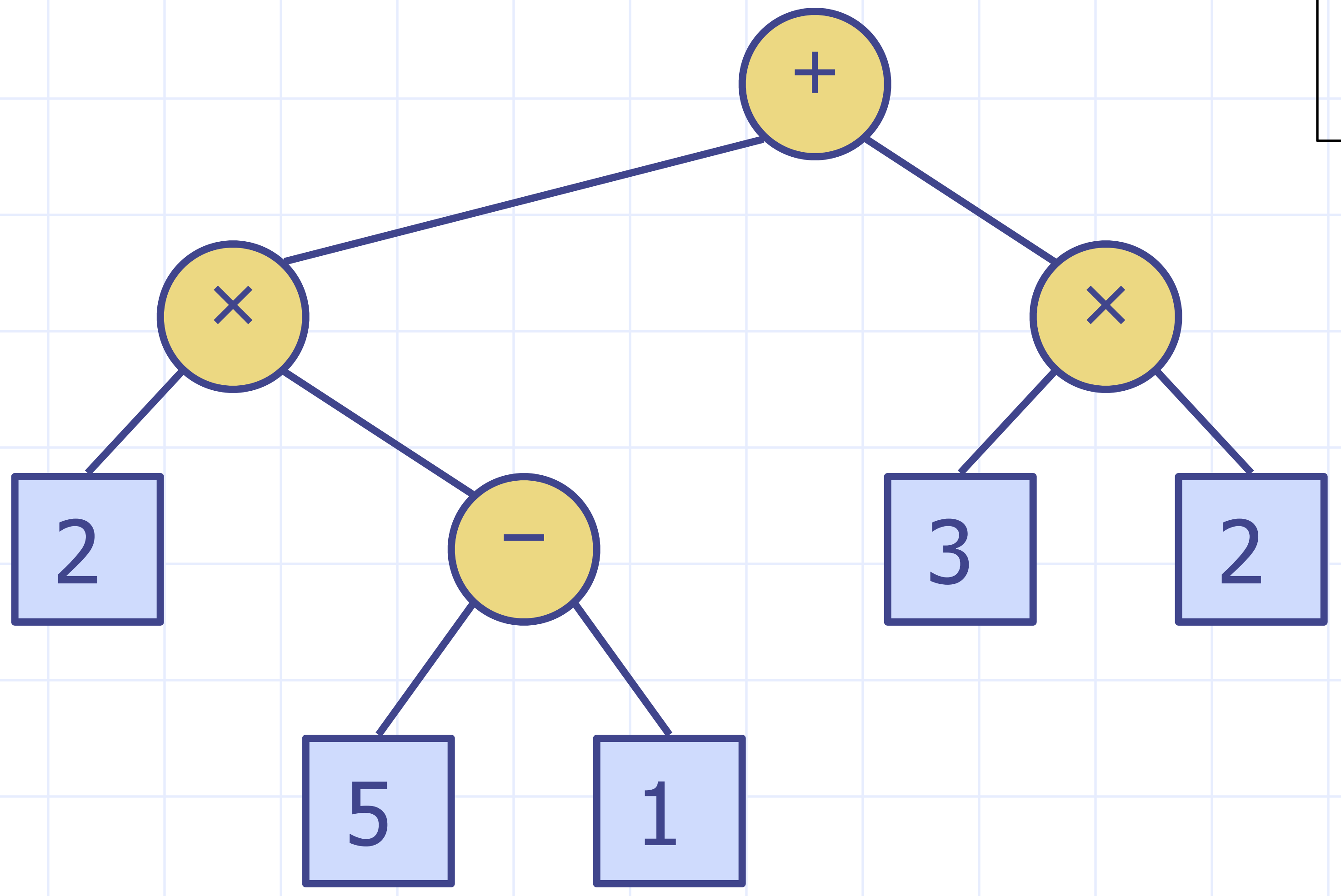


$((2 \times (a - 1)) + (3 \times b))$

Evaluate Arithmetic Expressions

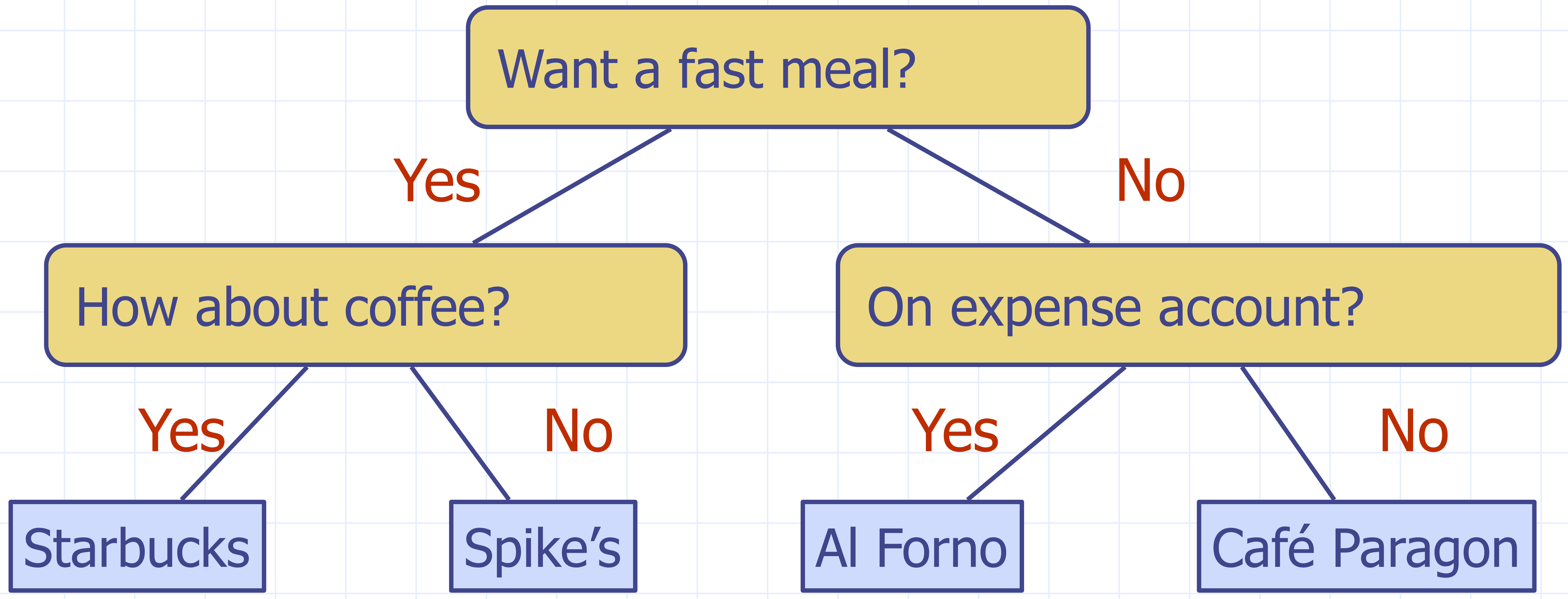
- Specialization of a postorder traversal
  - recursive method returning the value of a subtree
  - when visiting an internal node, combine the values of the subtrees

```
Algorithm evalExpr(v)
    if is_leaf (v)
        return v.element ()
    else
            x ← evalExpr(left (v))
            y ← evalExpr(right (v))
            ◊ ← operator stored at v
        return x ◊ y
```

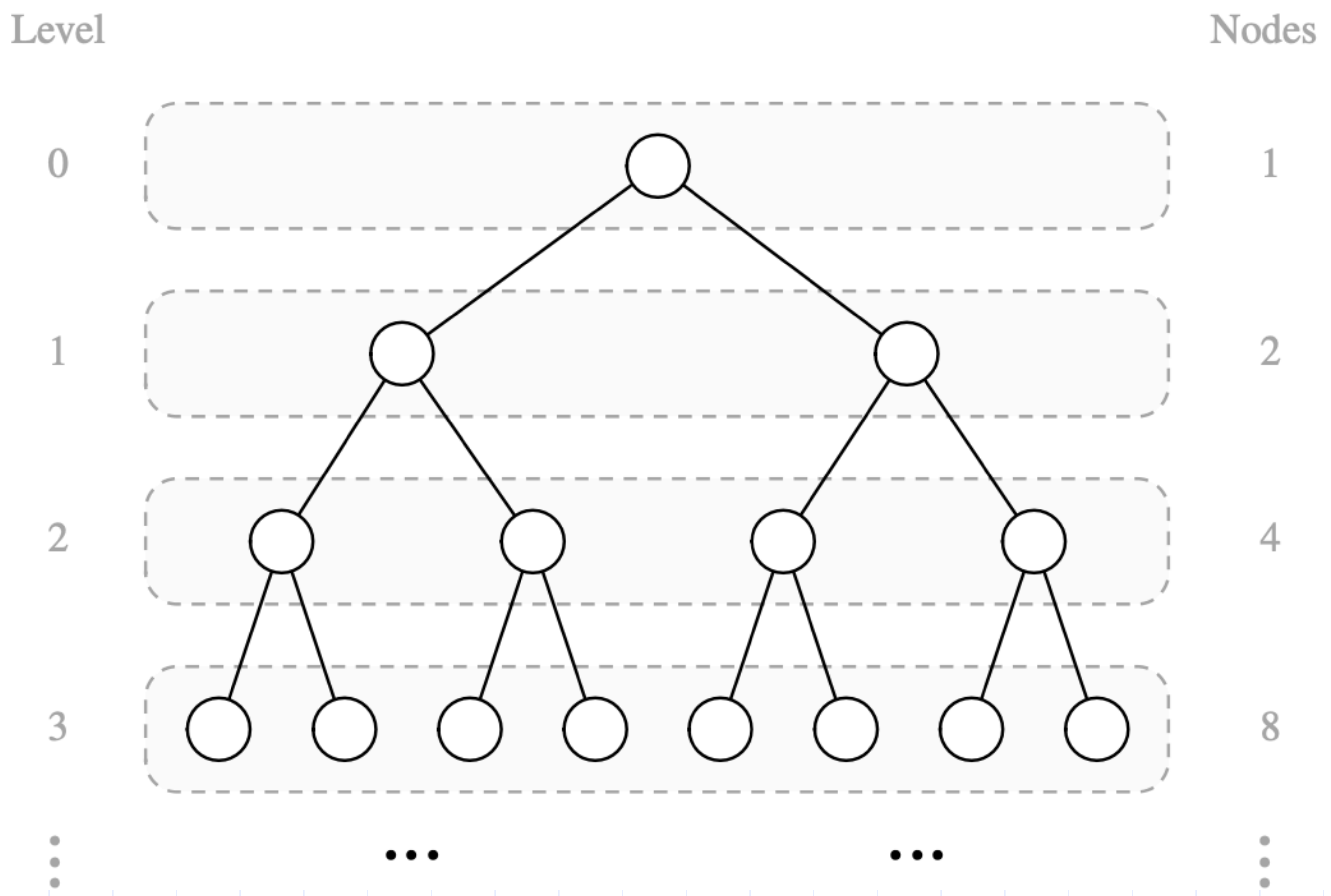


Decision Tree

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



Maximum number of nodes in the levels of a binary tree



H=3 height of a proper tree

No_of_nodes=2^(H+1)-1

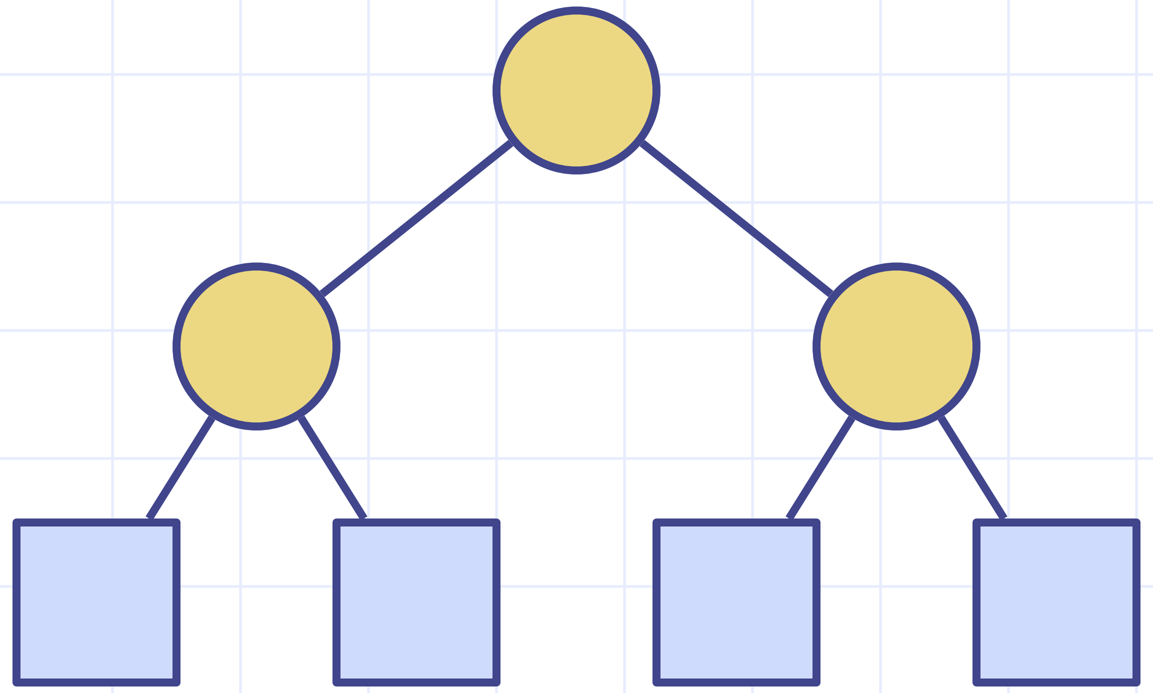
Level: all nodes of a same depth

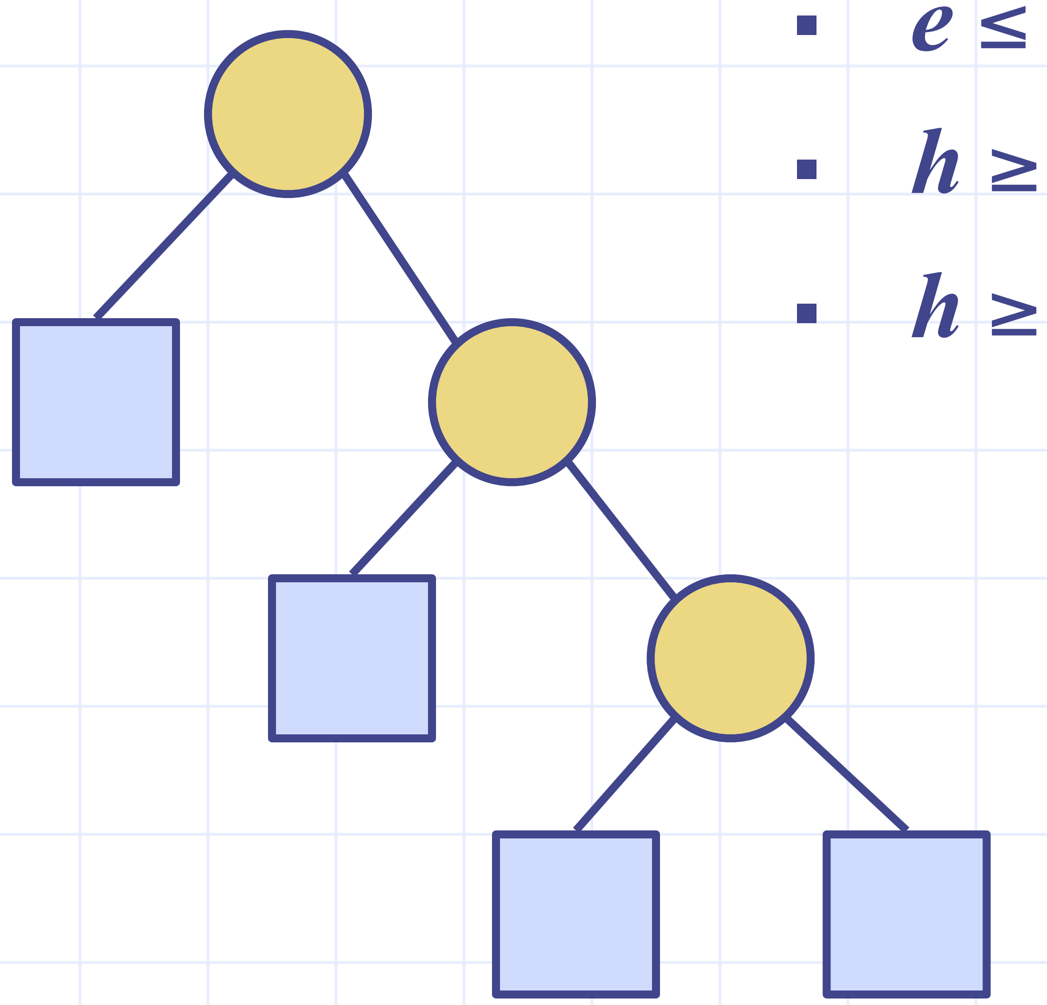
Properties of Proper (Full) Binary Trees

- Notation

n	number of nodes
e	number of external nodes
i	number of internal nodes
h	height

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

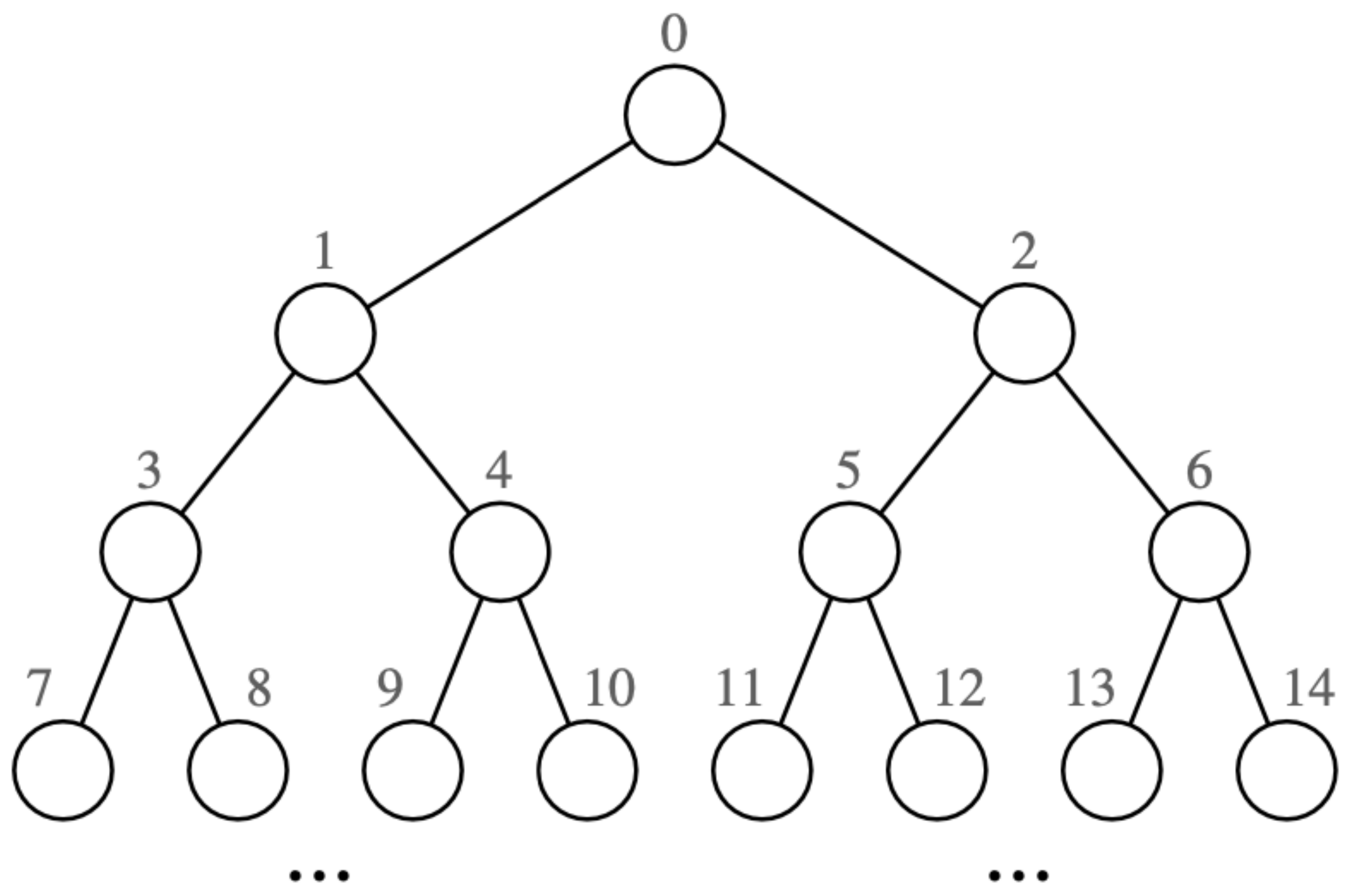




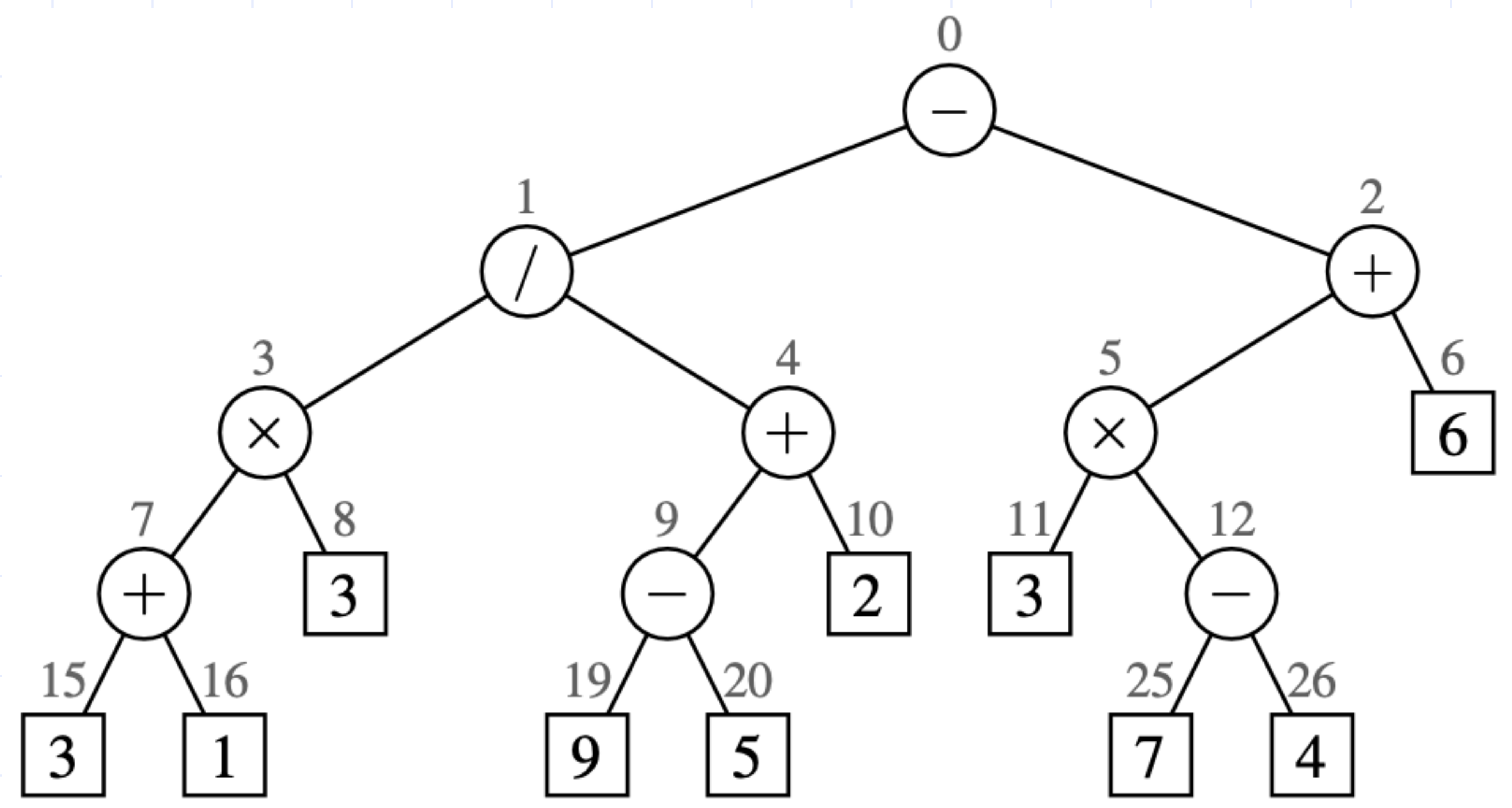
A full binary tree : every node other than leaves has two children

A proper binary tree : a full binary tree in which all leaves are at the same depth

Binary Tree Node Numbering



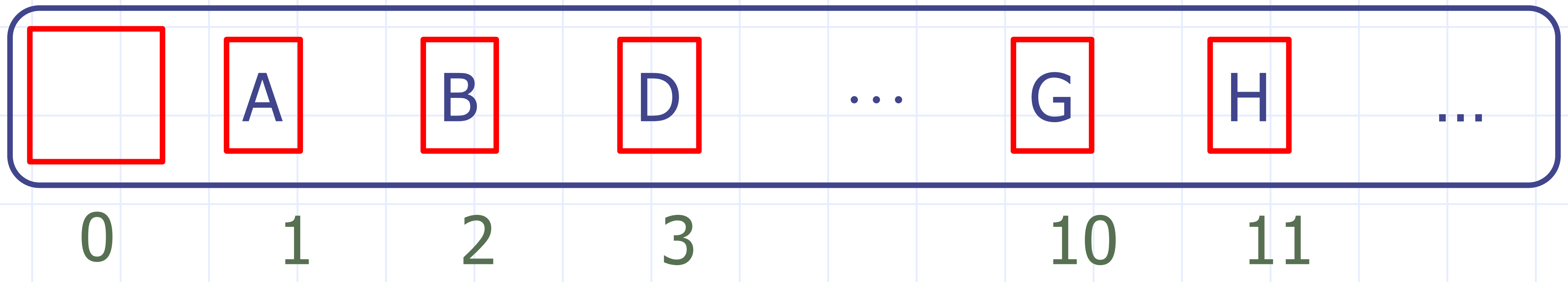
Binary Tree Node Numbering



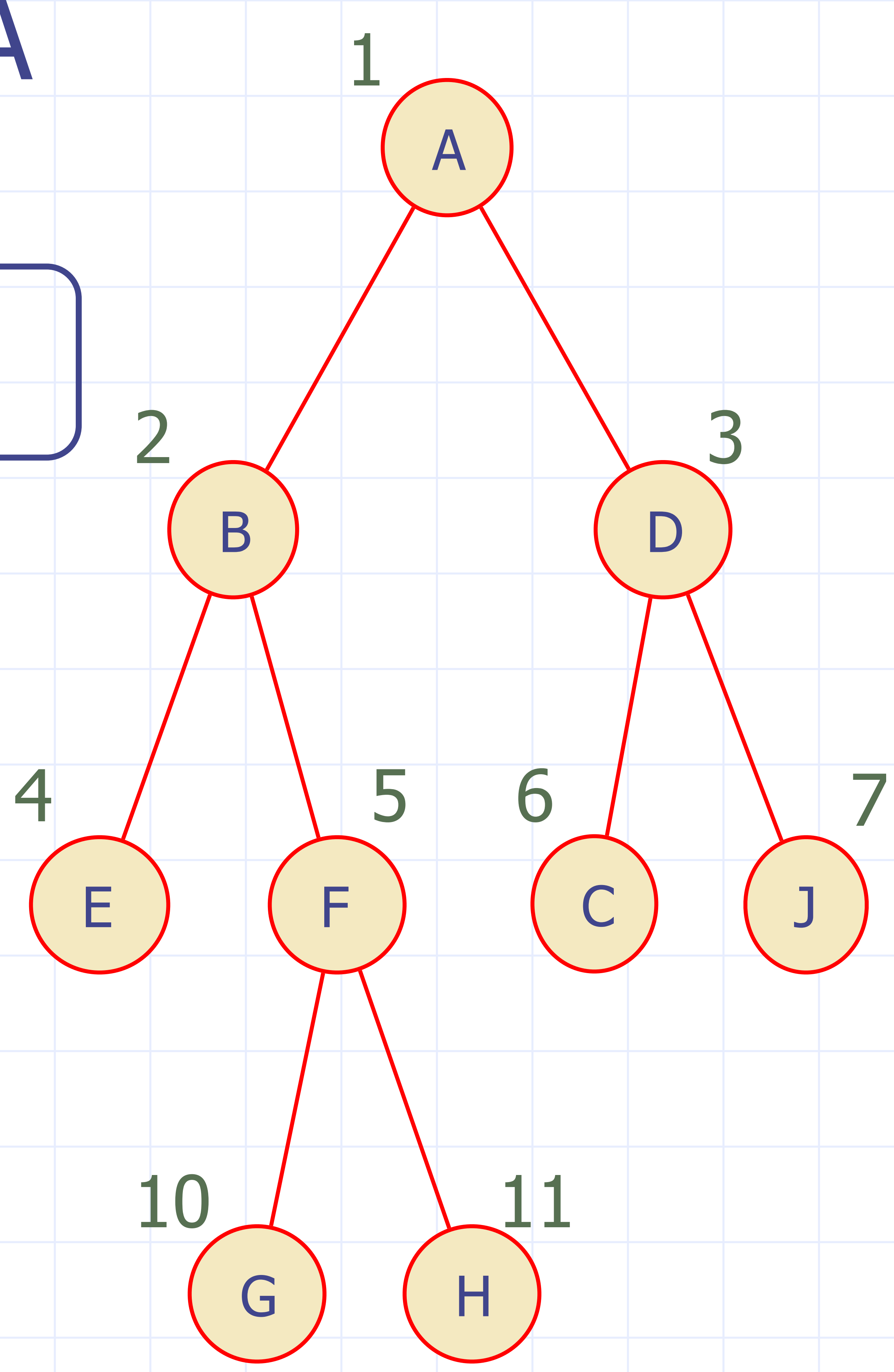
Depth of a node
Height of a tree = max Depth

Array-Based Representation of Binary Trees

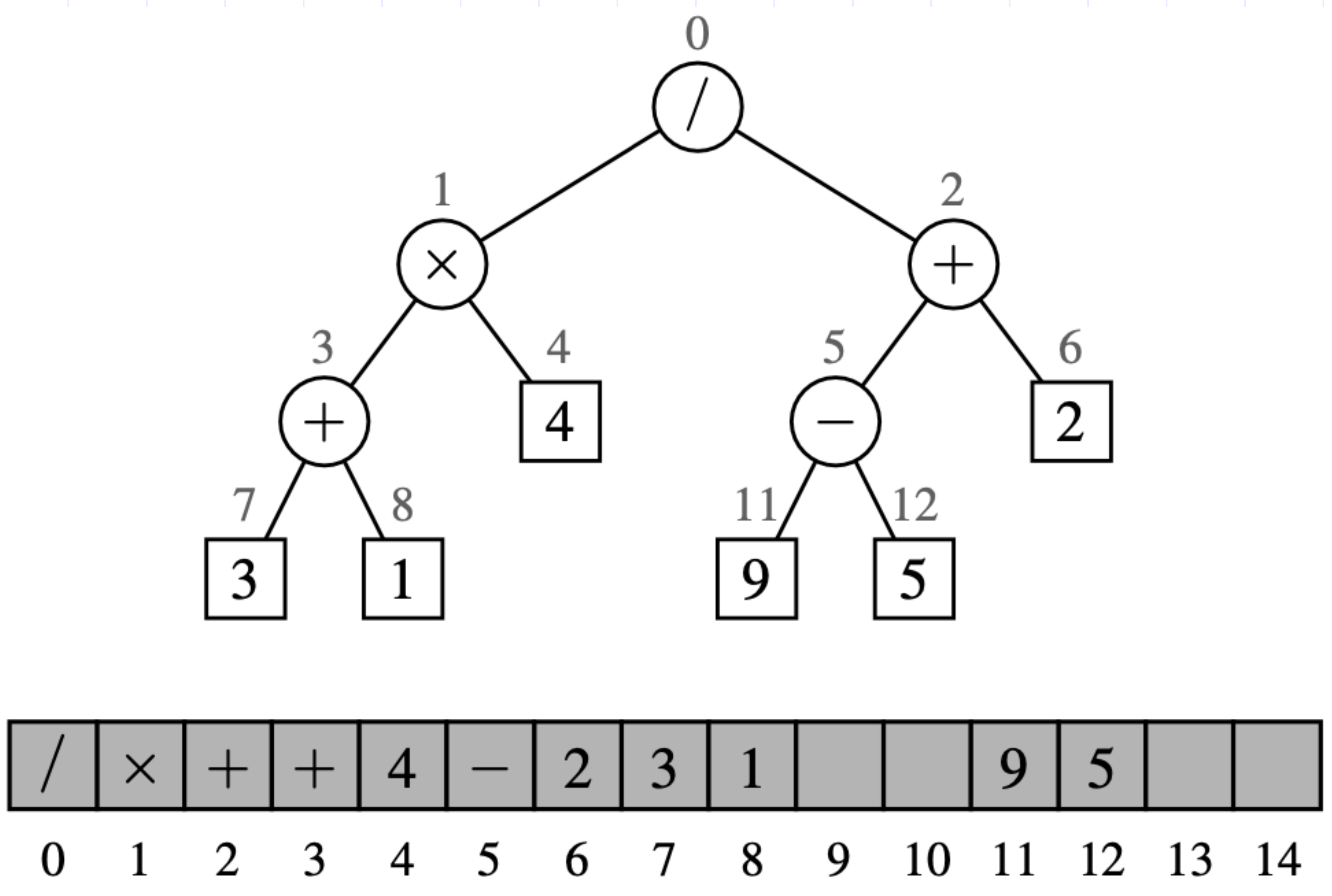
- Nodes are stored in an array A



- Node v is stored at A[rank(v)]
  - rank(root) = 1
  - if node is the left child of parent(node), rank(node) = 2 · rank(parent(node))
  - if node is the right child of parent(node), rank(node) = 2 · rank(parent(node)) + 1



Representation of a binary tree by means of an array



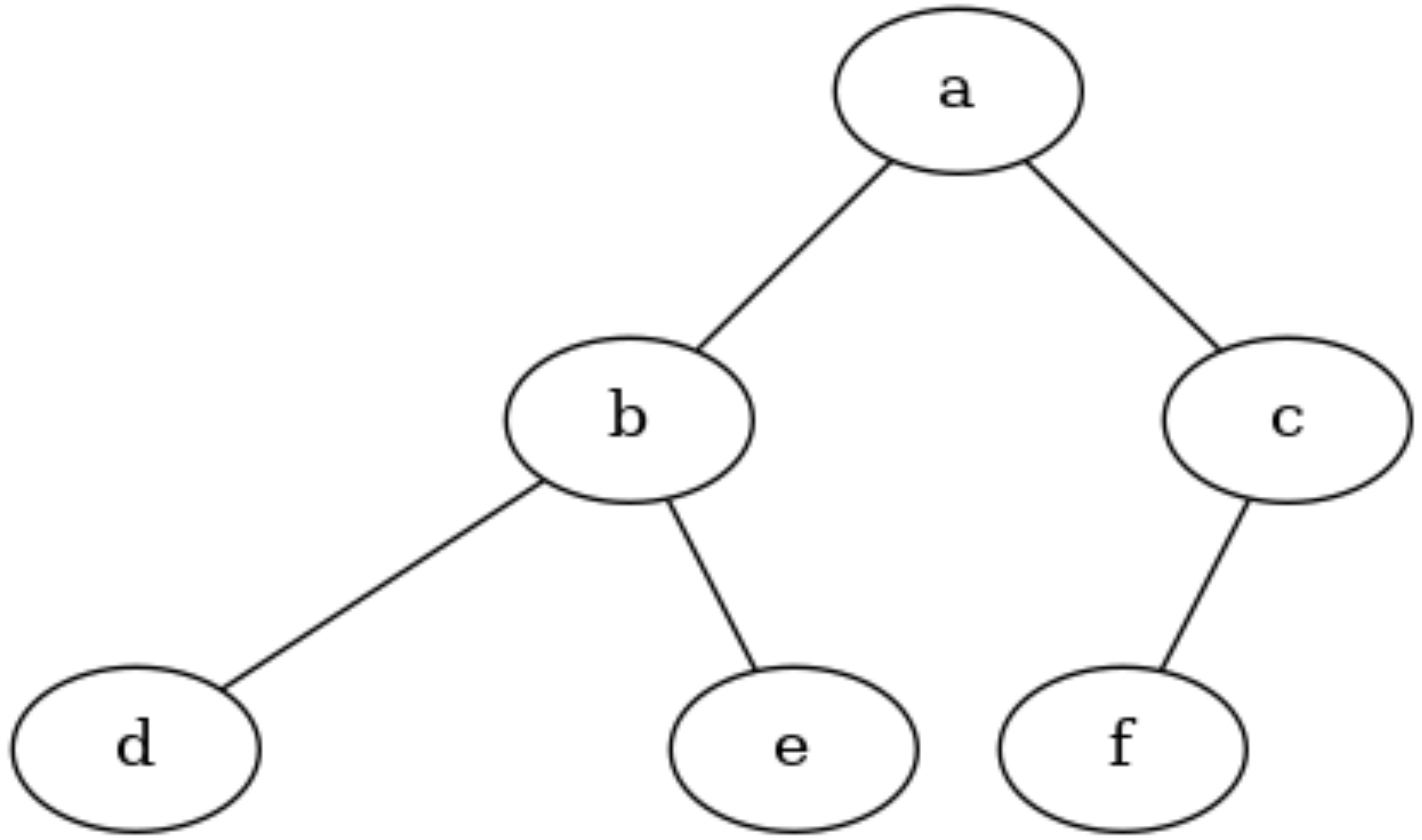
/	×	+	+	4	−	2	3	1			9	5		
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

Implementation

- `BinaryTree()` creates a new instance of a binary tree.
- `get_root_val()` returns the object stored in the current node.
- `set_root_val(val)` stores the object in parameter `val` in the current node.
- `get_left_child()` returns the binary tree corresponding to the left child of the current node.
- `get_right_child()` returns the binary tree corresponding to the right child of the current node.
- `insert_left(val)` creates a new binary tree and installs it as the left child of the current node.
- `insert_right(val)` creates a new binary tree and installs it as the right child of the current node.

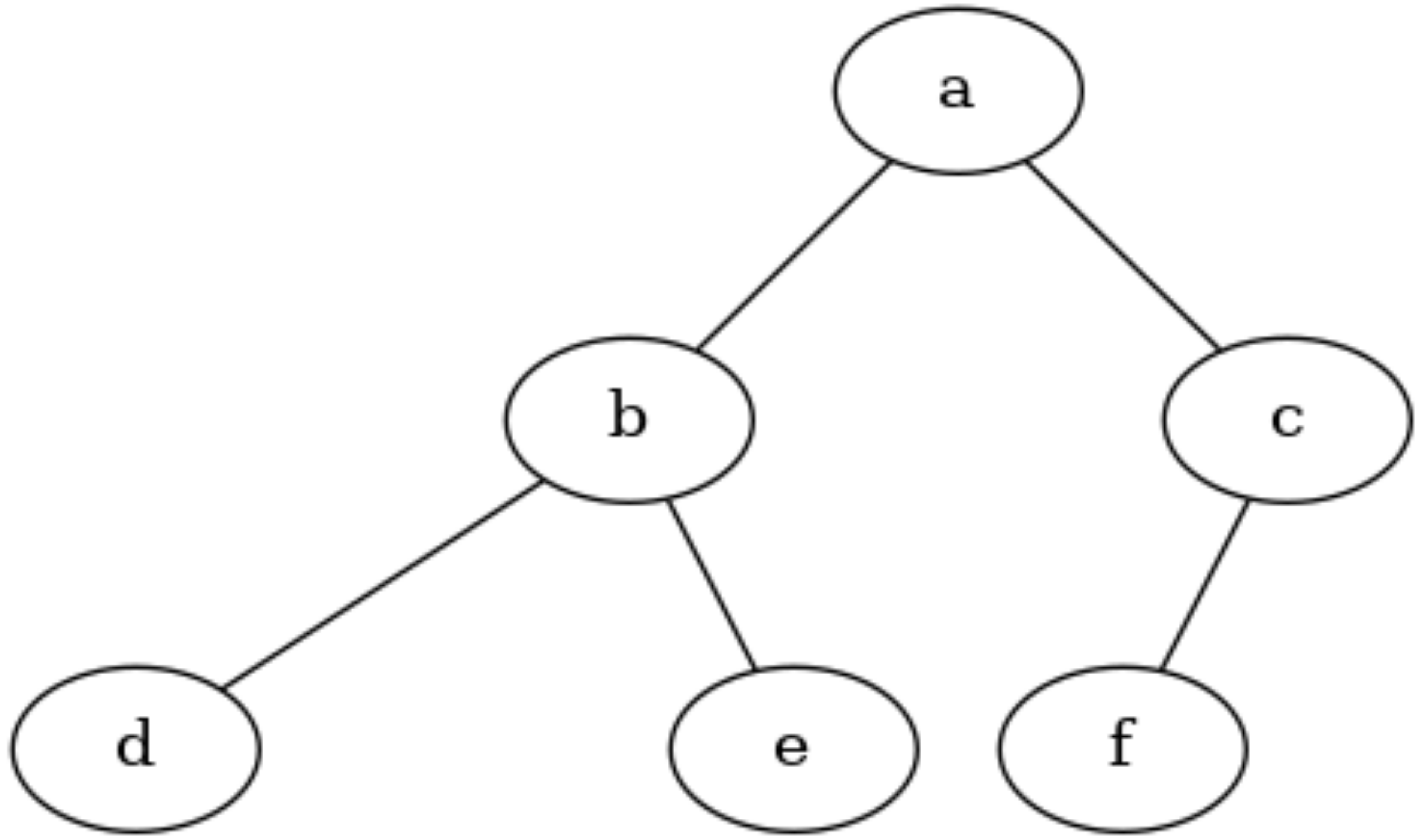
Question #1: A Small Tree

List of Lists Representation



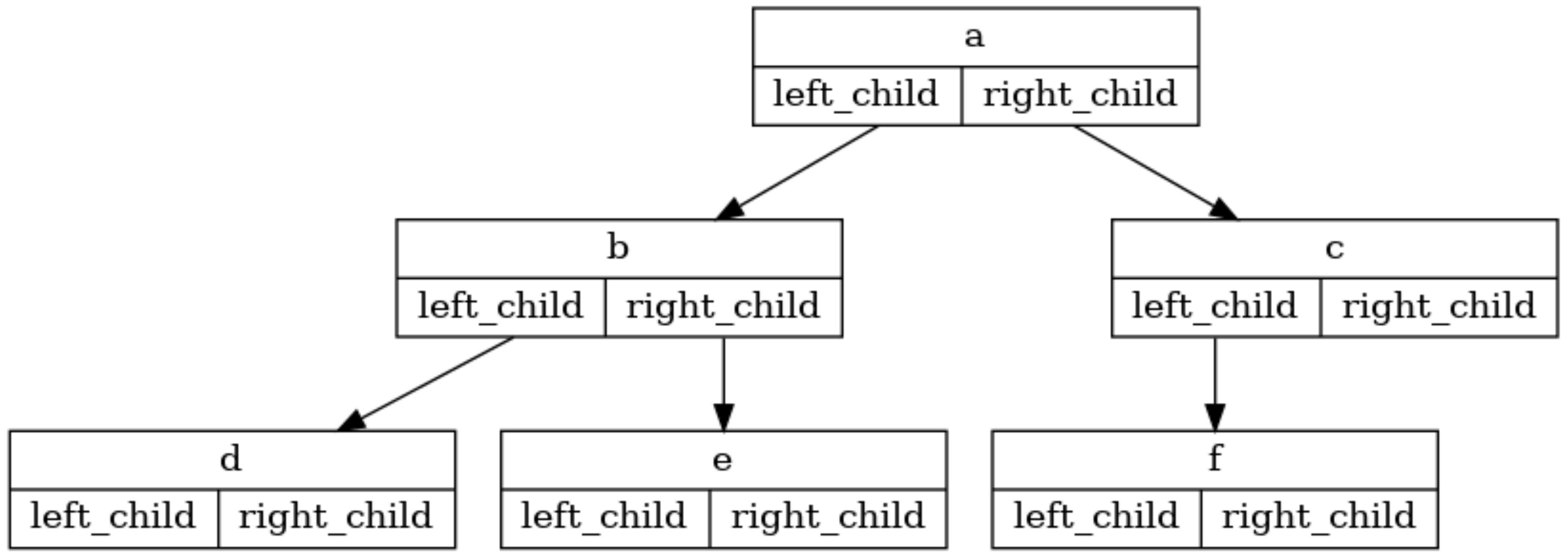
A1: A Small Tree

List of Lists Representation



Nodes and References

```
class TreeNode:
    def __init__(self, value):
        self.value = value
        self.left = None
        self.right = None
```



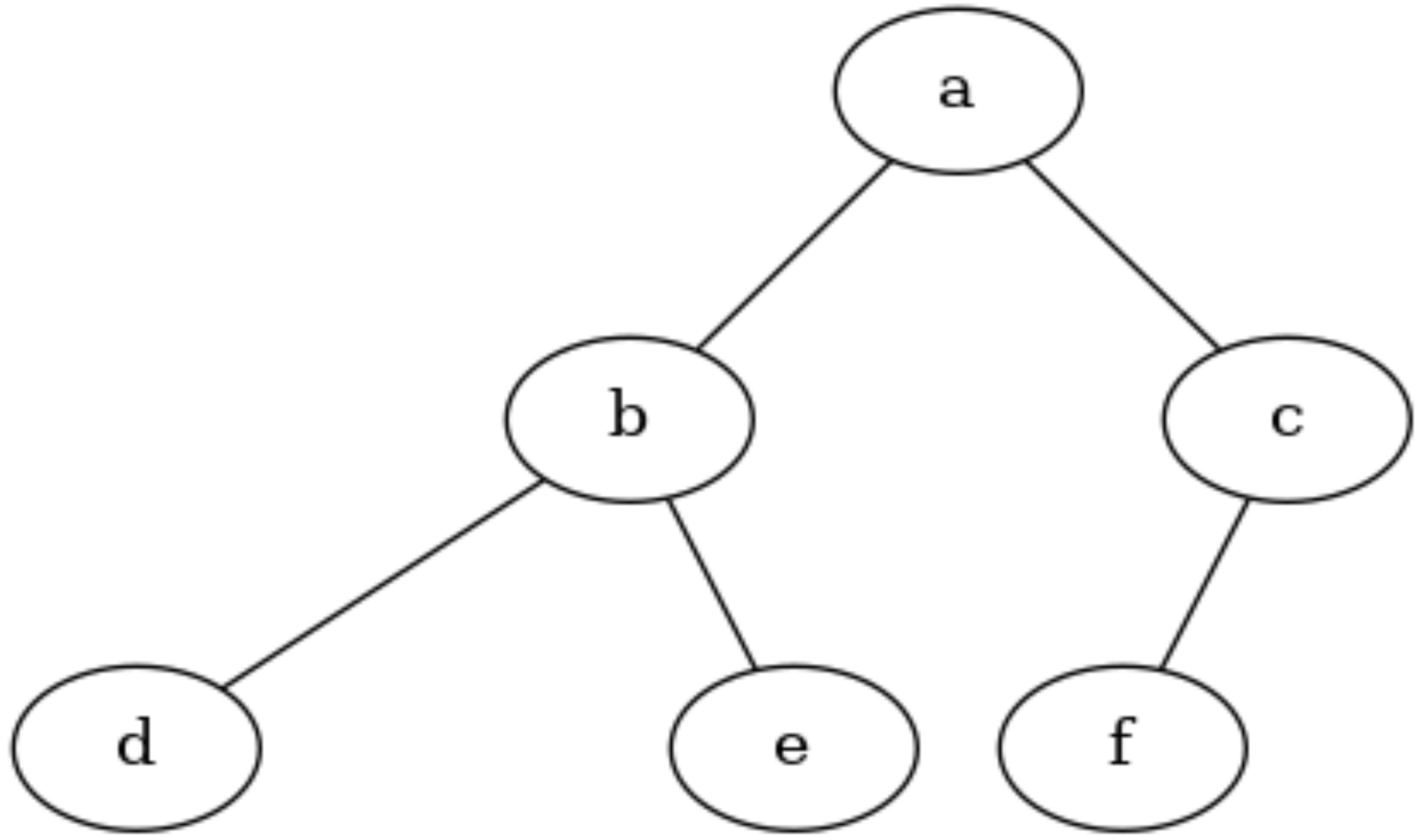
Tree Traversals

- https://runestone.academy/ns/books/published/pythonds3/Trees/TreeTraversals.html
- T tree, n nodes, height h: recursively implemented O(n) time complexity, O(h) space complexity

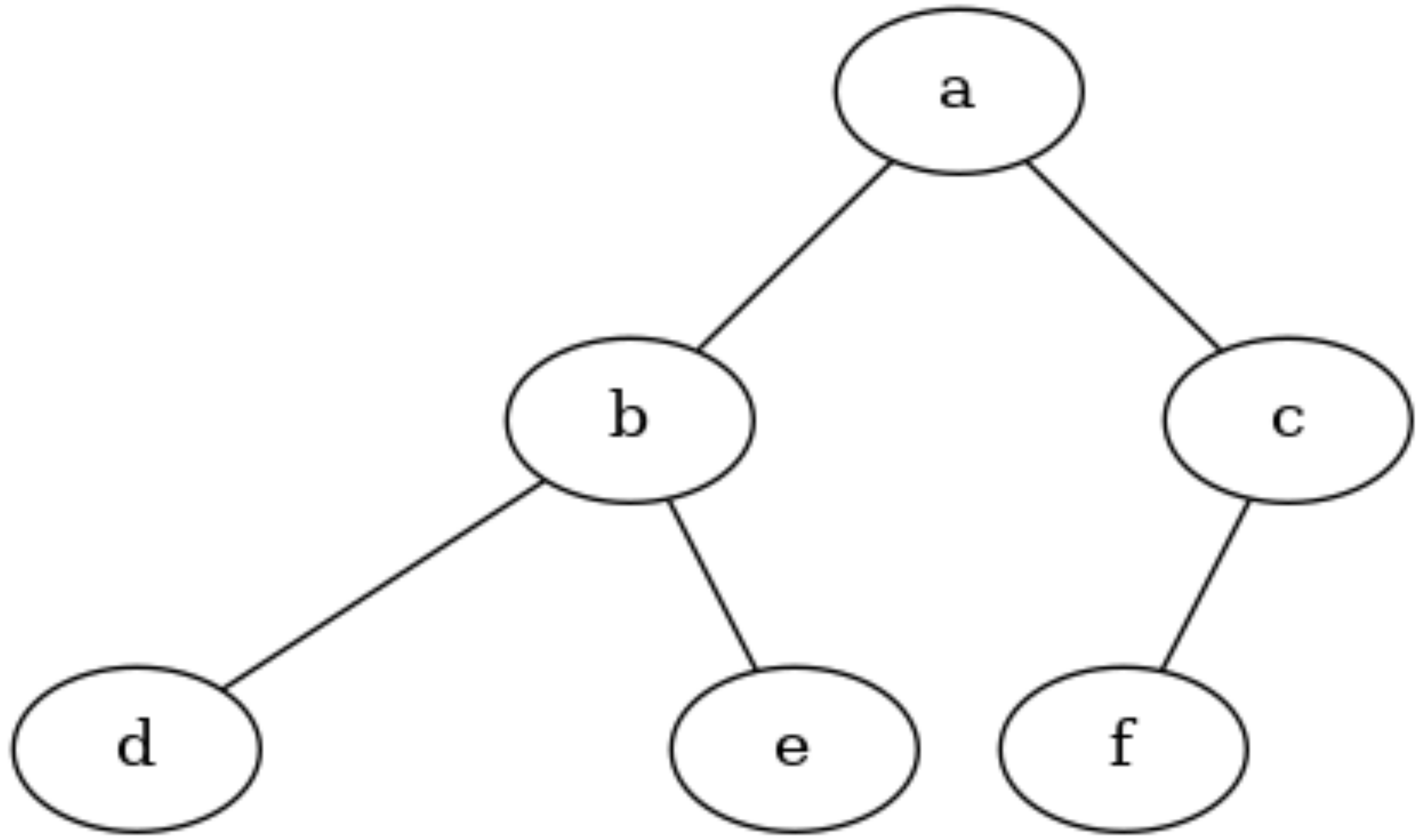
Question #2 In-order traversal

Left node—>parent—>right node:” ?

In what order are nodes visited during a in-order traversal of the tree?



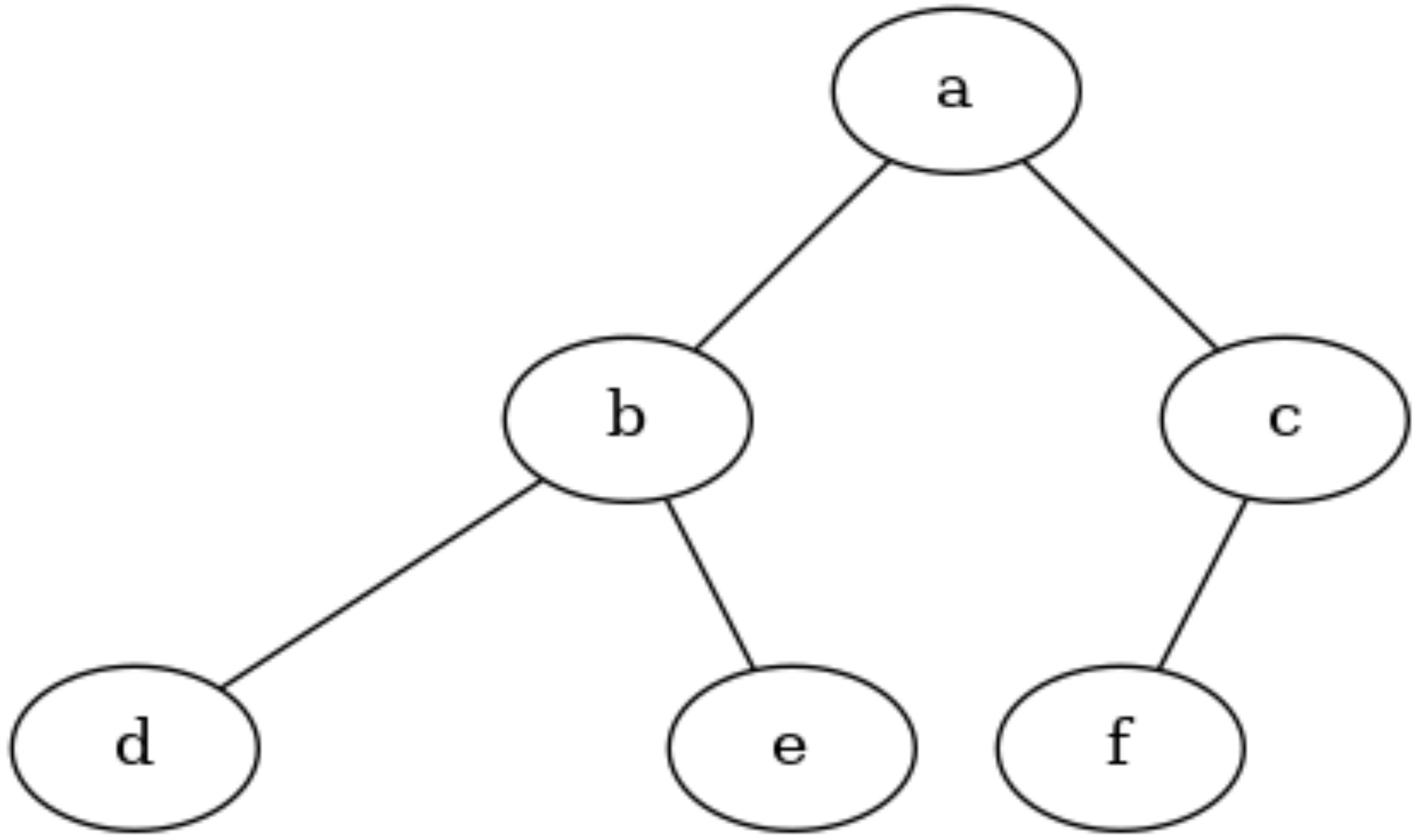
Answer #2 In-order traversal



Question #3: Preorder traversal

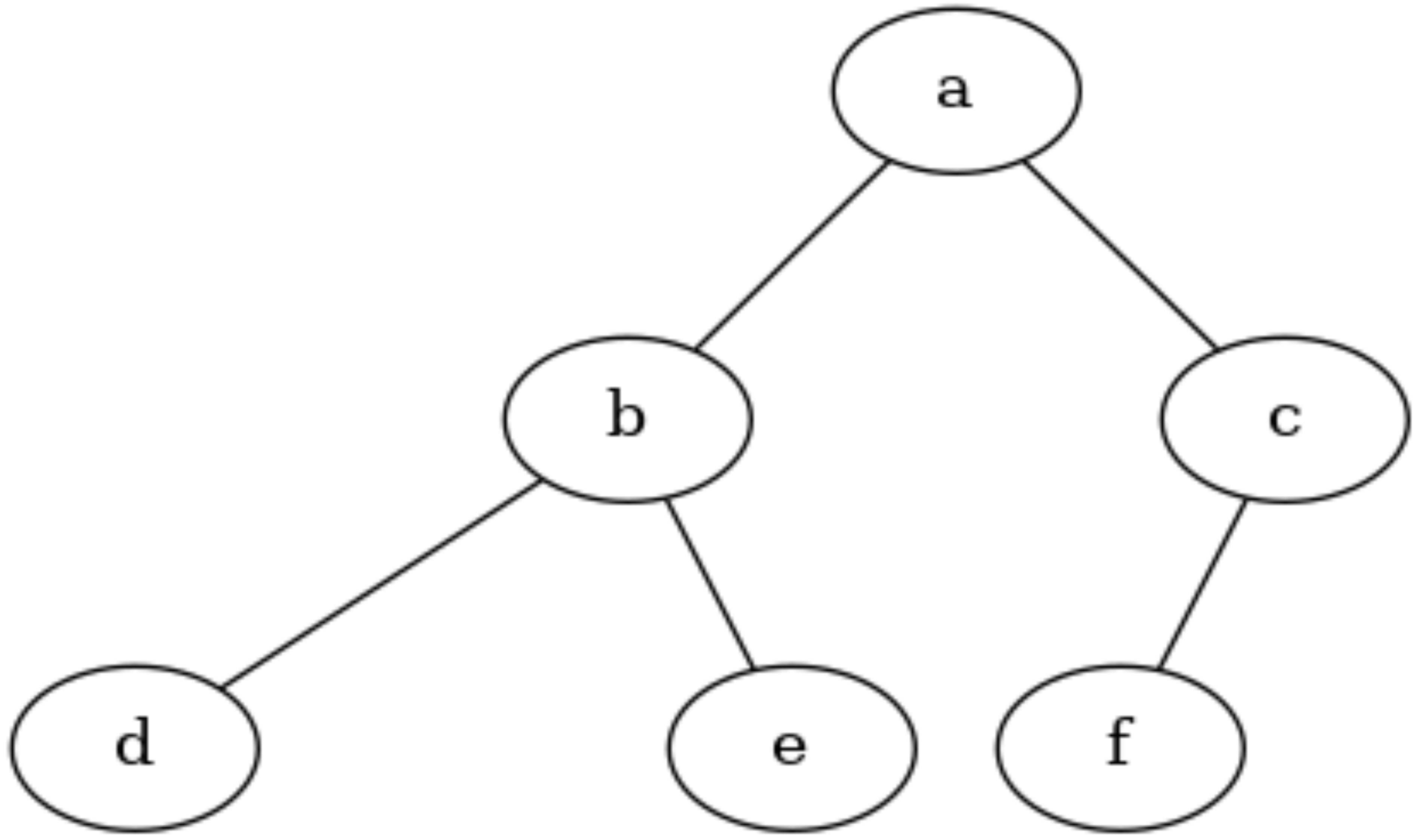
Parent —> left —>right: ?

In what order are nodes visited during a preorder traversal of the tree?



Answer #3: Preorder traversal

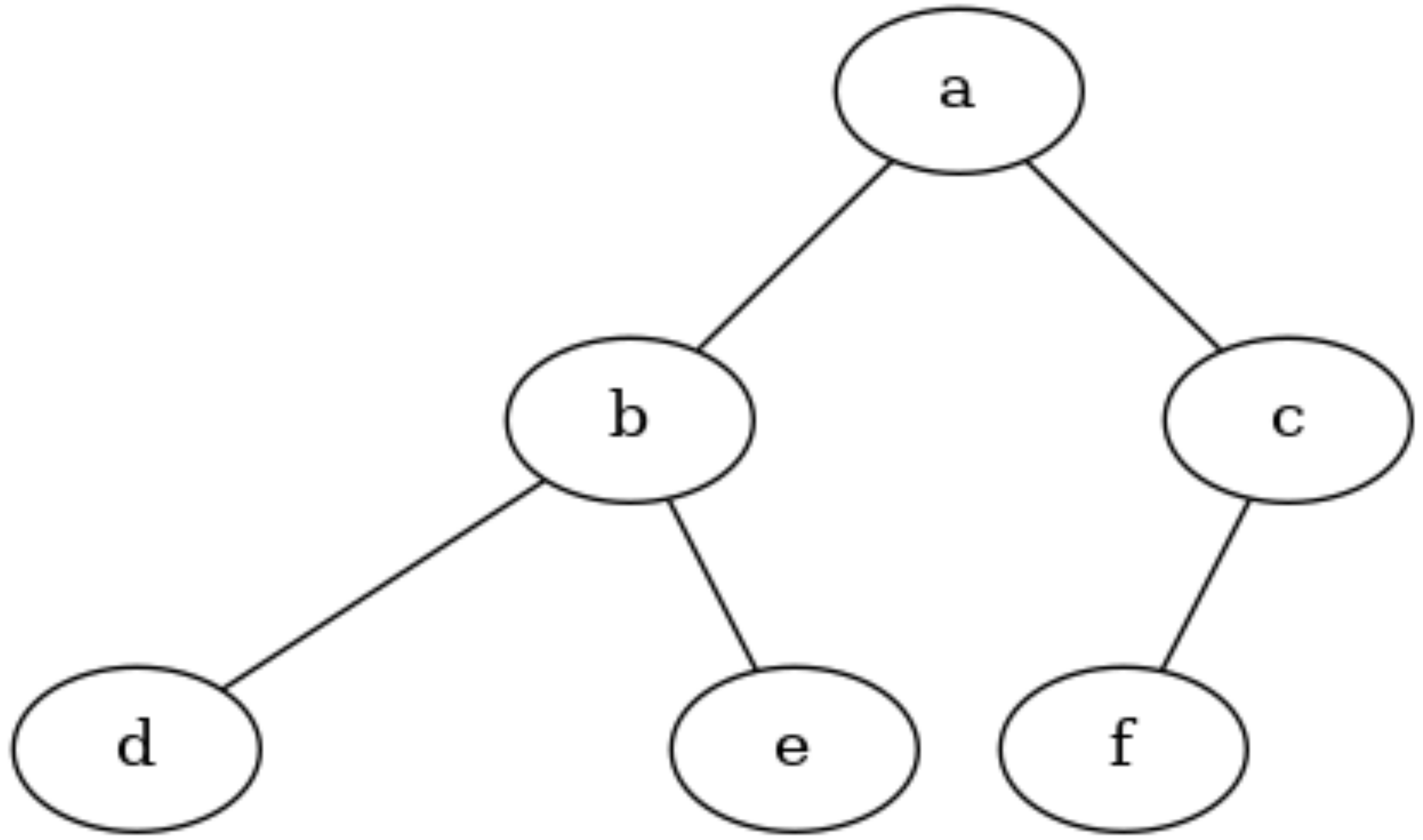
Parent —> left —>right: ?



Question #4: Postorder Traversal

Left node —> right node —> parent node:?

In what order are nodes visited during a postorder traversal of the tree?



Answer #4: Postorder Traversal

Left node —> right node —> parent node:?