SE274 Data Structure

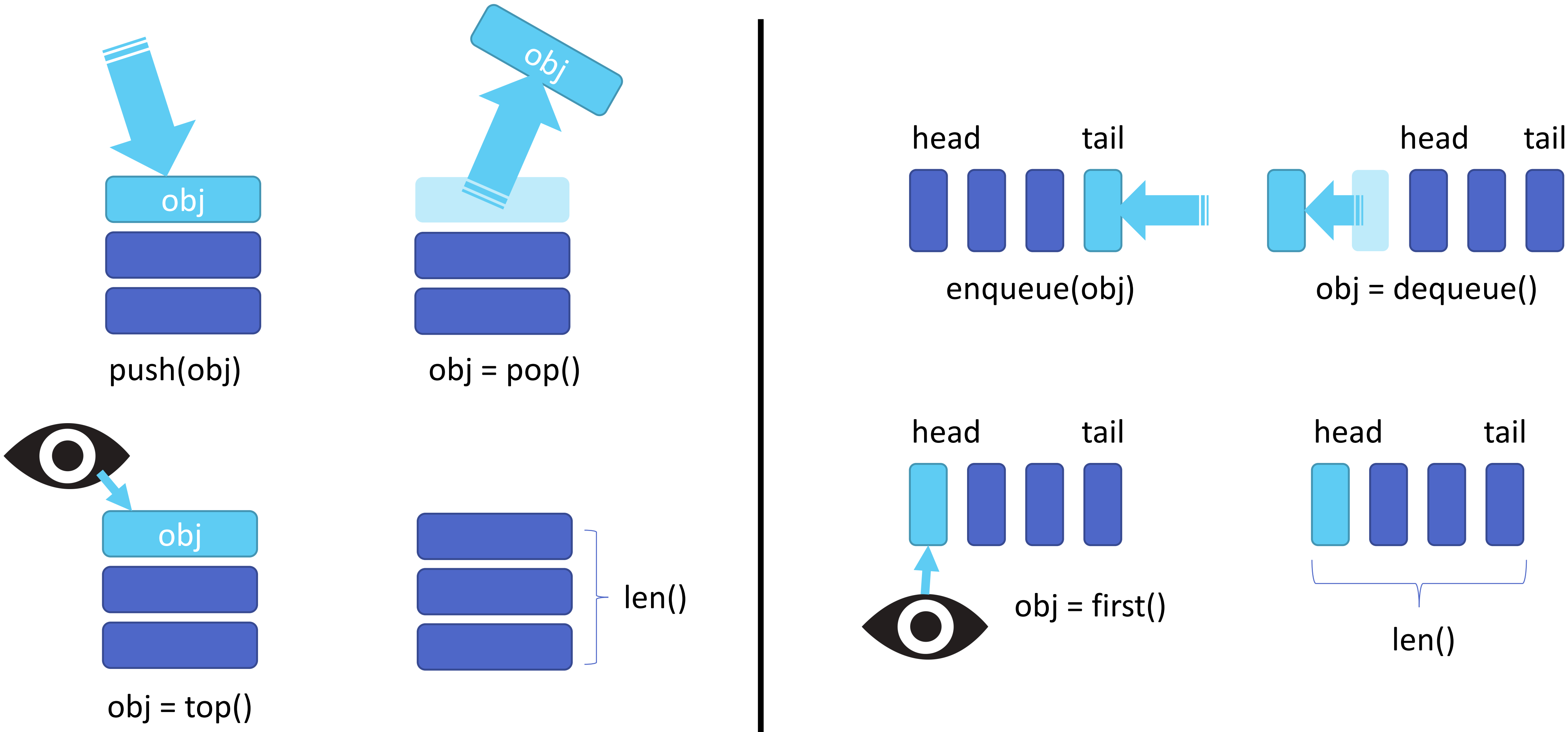
Lecture 4: Linked List, Tree

Mar 23, 2020

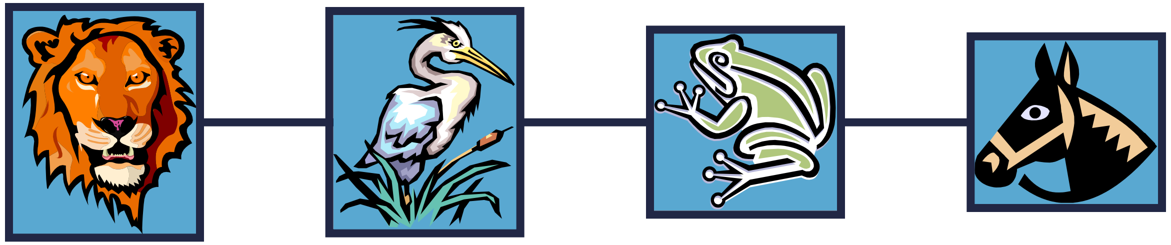
Instructor: Sunjun Kim

Information&Communication Engineering, DGIST

Recap

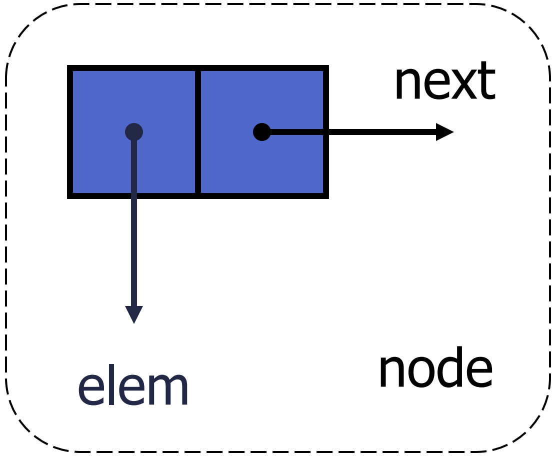

obj
push(obj)
obj
obj = pop()
obj
obj = top()
len()
head
tail
enqueue(obj)
head
tail
obj = dequeue()
head
tail
obj = first()
head
tail
len()

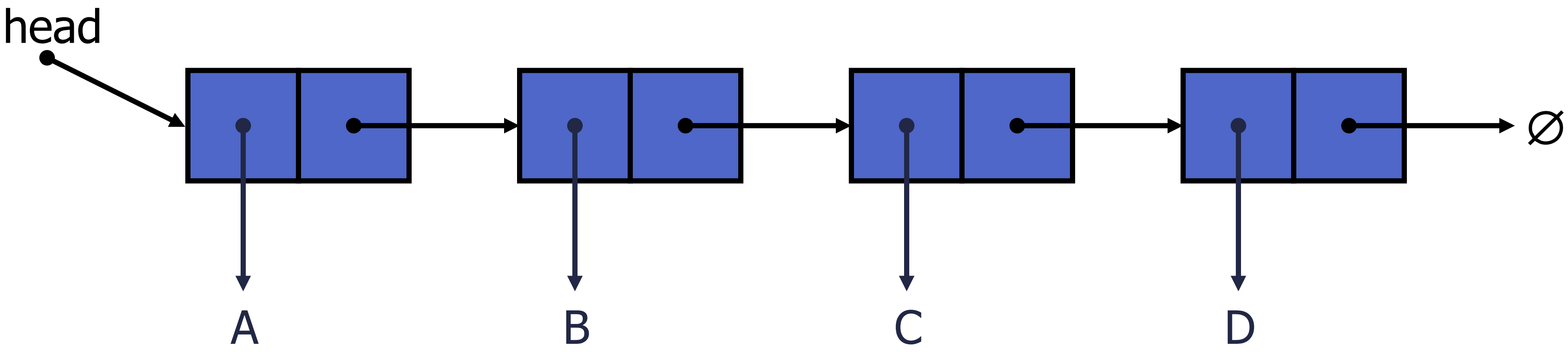
Linked Lists



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





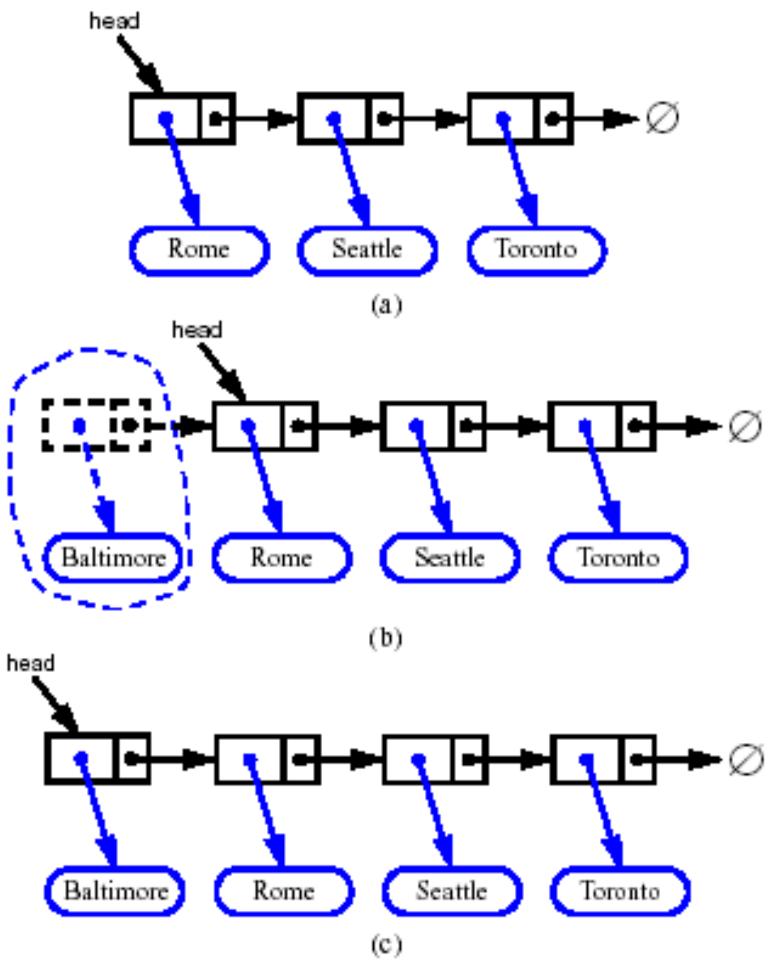
The Node Class for List Nodes

```
class Node:
        __slots__ = '_element', '_next’

        def __init__(self):
            self._element = None
            self._next = None

        def __init__(self, element, nxt):
            self._element = element
            self._next = nxt
```

```
# Optional: assign memory
space for the member
variables, faster!
```

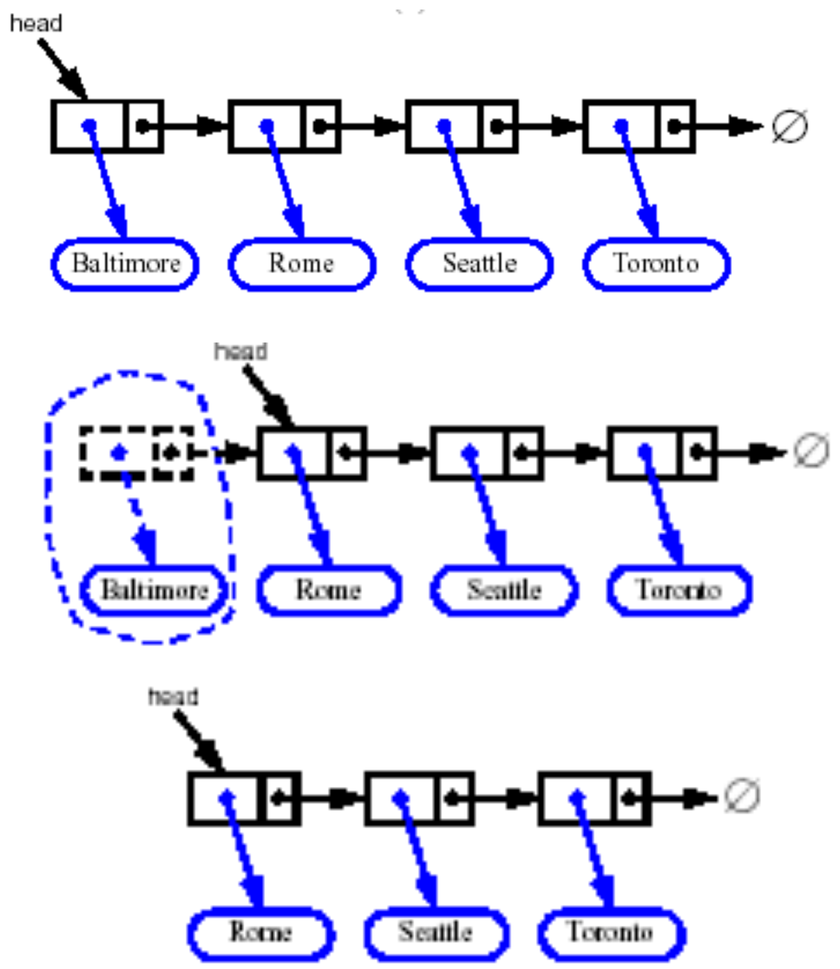
Inserting at the Head

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



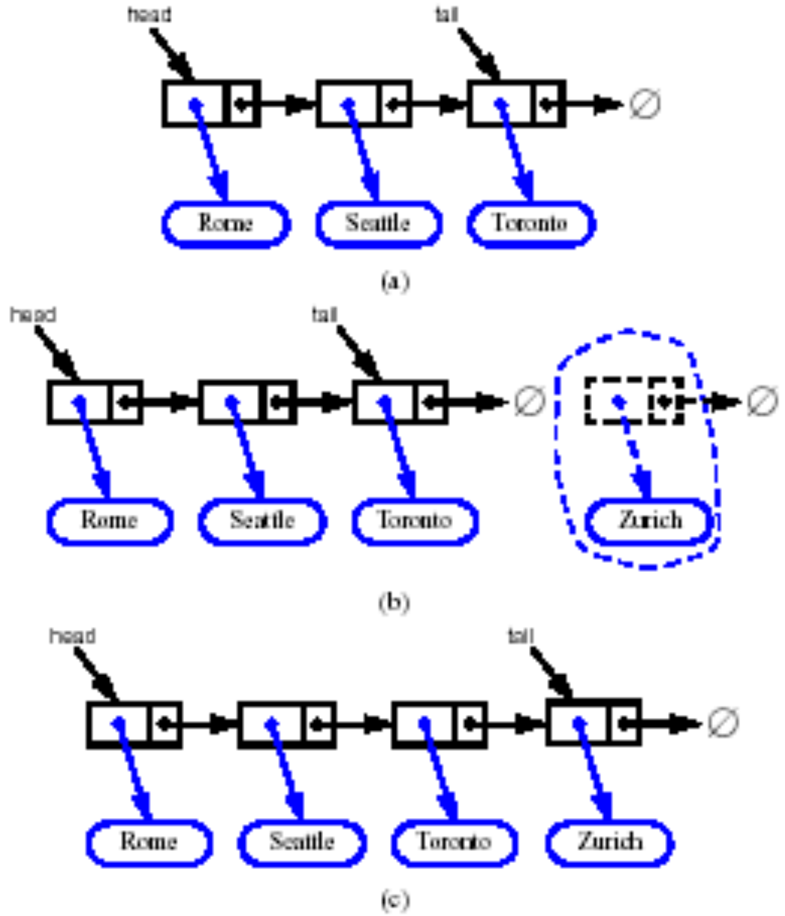
Removing at the Head

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



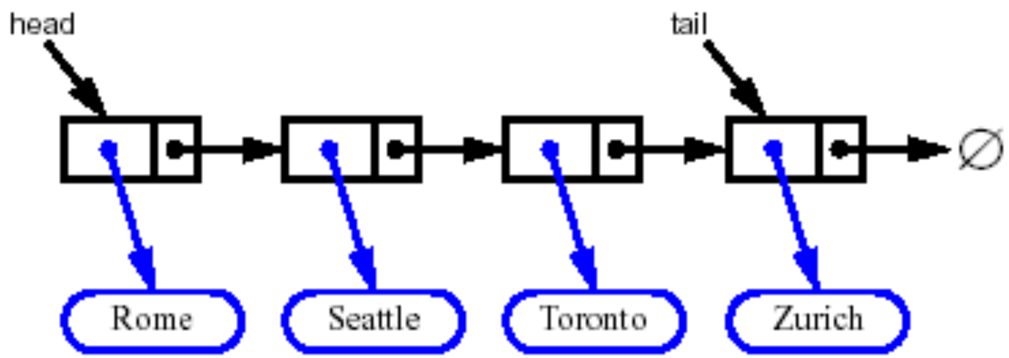
Inserting at the Tail

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



Exercise: Insert/Remove at the middle.

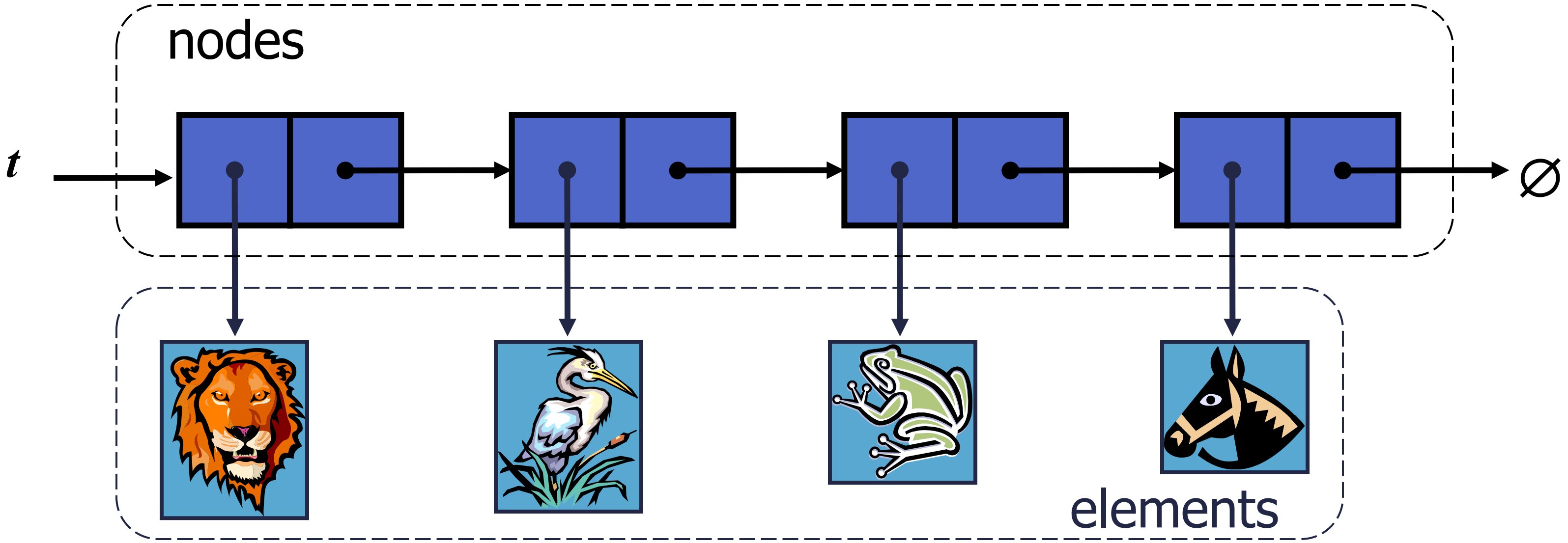
Why use linked list?

	Linked list	Array	Dynamic array
Indexing	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$
Insert/delete at beginning	$\Theta(1)$	N/A	$\Theta(n)$
Insert/delete at end	$\Theta(1)$ when last element is known; $\Theta(n)$ when last element is unknown	N/A	$\Theta(1)$ amortized
Insert/delete in middle	search time + $\Theta(1)$[5][6]	N/A	$\Theta(n)$
Wasted space (average)	$\Theta(n)$	0	$\Theta(n)$[7]

* credit: https://en.wikipedia.org/wiki/Linked_list

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

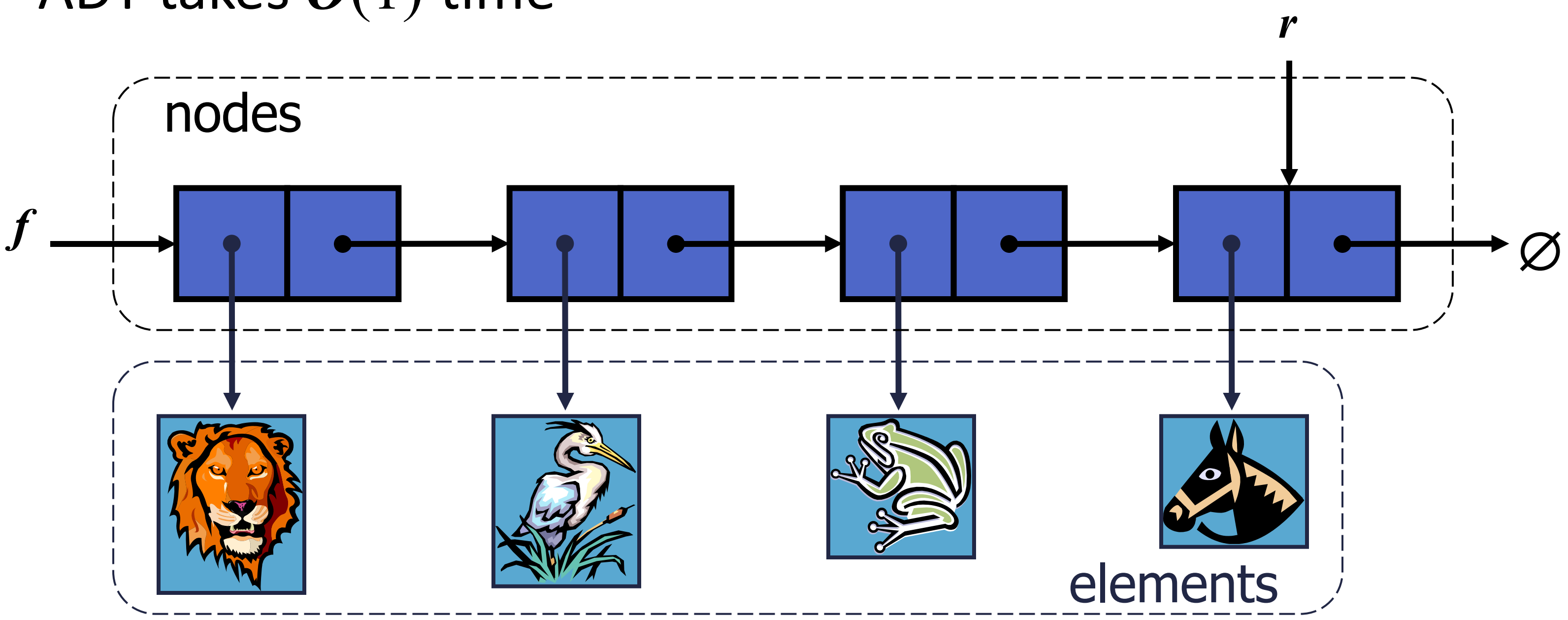


Linked-List Stack in Python

```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

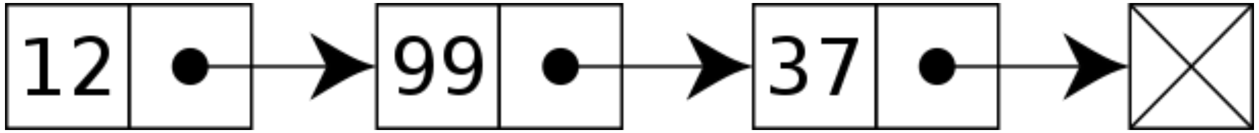


Linked-List Queue in Python

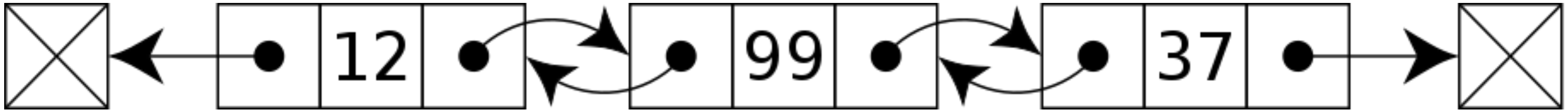
```python
class LinkedQueue:
  """FIFO queue implementation using a singly linked list for storage."""

  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
```

Variants of linked lists

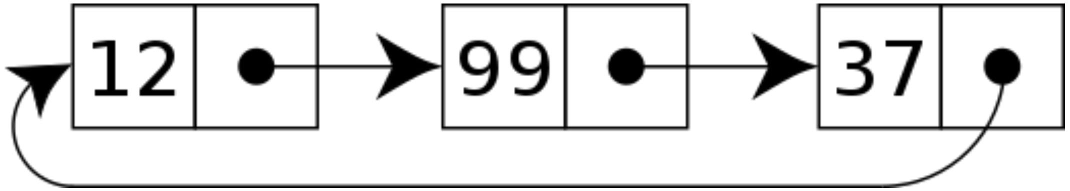
- Singly linked list



- Doubly linked list

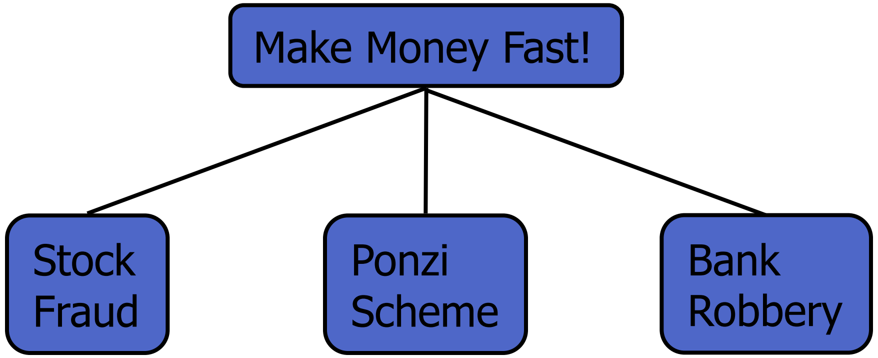


- Circular linked list



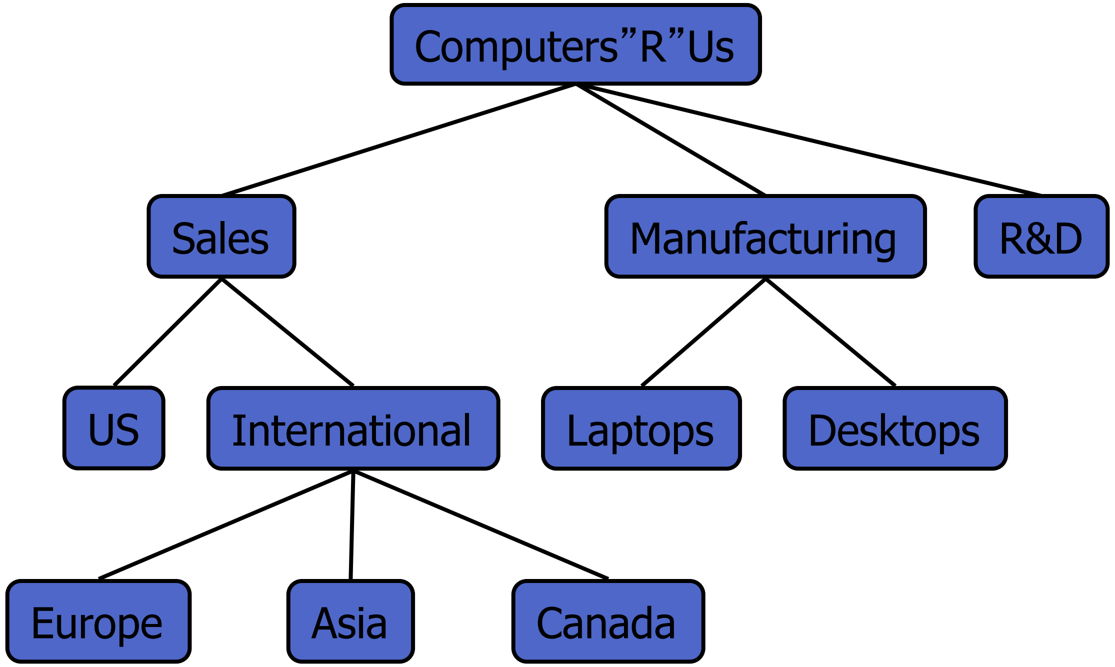
* Image credit: https://en.wikipedia.org/wiki/Linked_list

Trees



What is a Tree

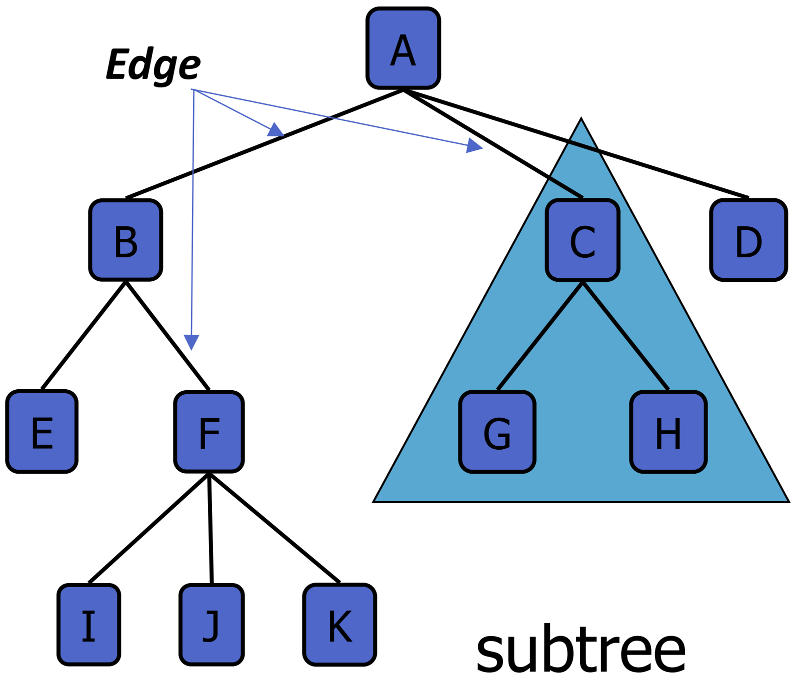
- In computer science, a tree is an abstract model of a hierarchical structure
- A tree consists of nodes with a parent-child relation
- Applications:
  - Organization charts
  - File systems
  - Programming environments



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 (3)
- Descendant of a node: child, grandchild, grand-grandchild, etc.

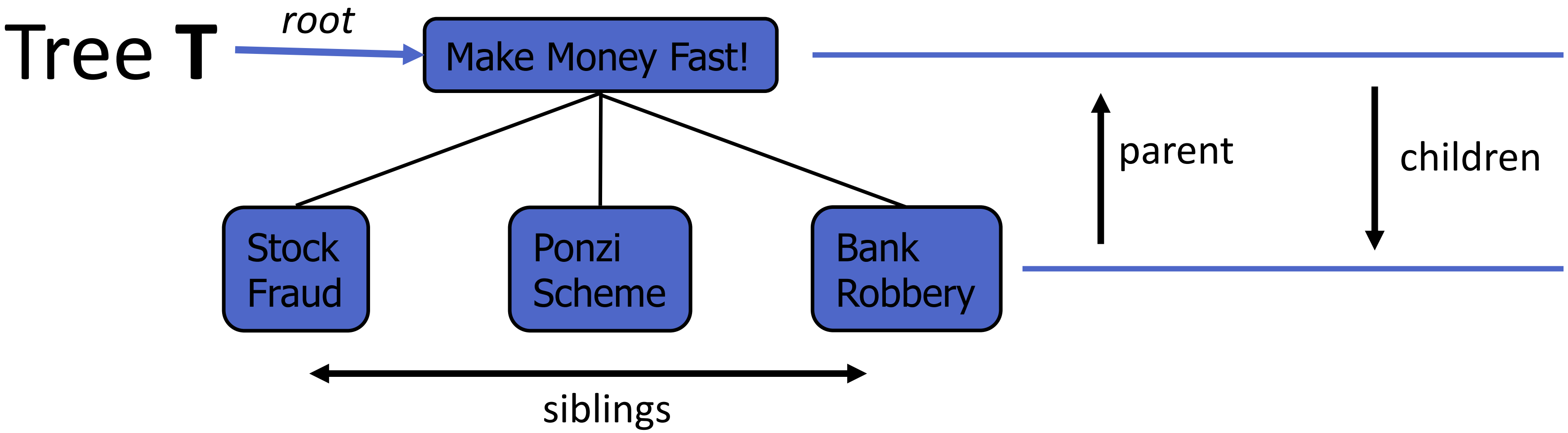
- Subtree: tree consisting of a node and its descendants



Path of J: A/B/F/J

Formal Definition

- We define a *tree* **T** as a set of ***nodes*** storing elements such that ***Nodes*** have **a *parent-child*** relationship, that satisfies:
  - If **T** is nonempty, it has a special node, called the ***root*** of **T**.
  - Each node ***v*** of **T** different from the root has a unique *parent node* ***w***; every node with parent ***w*** is a child of ***w,*** nodes that share the same parent are called *siblings.*



Tree ADT

- We use positions to abstract nodes
- Generic methods:
  - Integer len()
  - Boolean is_empty()
  - Iterator positions() → get positions
  - Iterator iter() → get elements
- 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

* Iterator 보충자료) https://dojang.io/mod/page/view.php?id=2405

Abstract Tree Class in Python

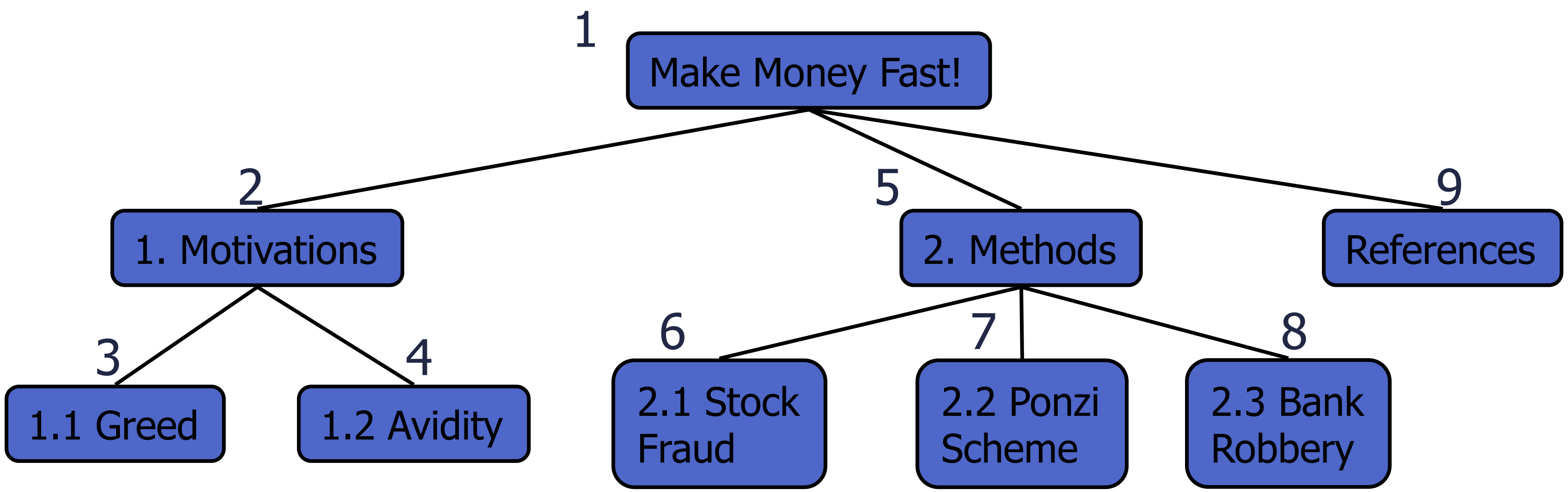
```python
class Tree:
  """Abstract base class representing a tree structure."""

  #------------------------------- nested Position class -------------------------------
  class Position:
    """An abstraction representing the location of a single element."""

    def element(self):
      """Return the element stored at this Position."""
      raise NotImplementedError('must be implemented by subclass')

    def __eq__(self, other):
      """Return True if other Position represents the same location."""
      raise NotImplementedError('must be implemented by subclass')

    def __ne__(self, other):
      """Return True if other does not represent the same location."""
      return not (self == other)          # opposite of __eq__

  # ---------- abstract methods that concrete subclass must support ----------
  def root(self):
    """Return Position representing the tree's root (or None if empty)."""
    raise NotImplementedError('must be implemented by subclass')

  def parent(self, p):
    """Return Position representing p's parent (or None if p is root)."""
    raise NotImplementedError('must be implemented by subclass')

  def num_children(self, p):
    """Return the number of children that Position p has."""
    raise NotImplementedError('must be implemented by subclass')

  def children(self, p):
    """Generate an iteration of Positions representing p's children."""
    raise NotImplementedError('must be implemented by subclass')

  def __len__(self):
    """Return the total number of elements in the tree."""
    raise NotImplementedError('must be implemented by subclass')

  # ---------- concrete methods implemented in this class ----------
  def is_root(self, p):
    """Return True if Position p represents the root of the tree."""
    return self.root( ) == p

  def is_leaf(self, p):
    """Return True if Position p does not have any children."""
    return self.num_children(p) == 0

  def is_empty(self):
    """Return True if the tree is empty."""
    return len(self) == 0
```

* Recursion 보충자료: https://dojang.io/course/view.php?id=7&section=39

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

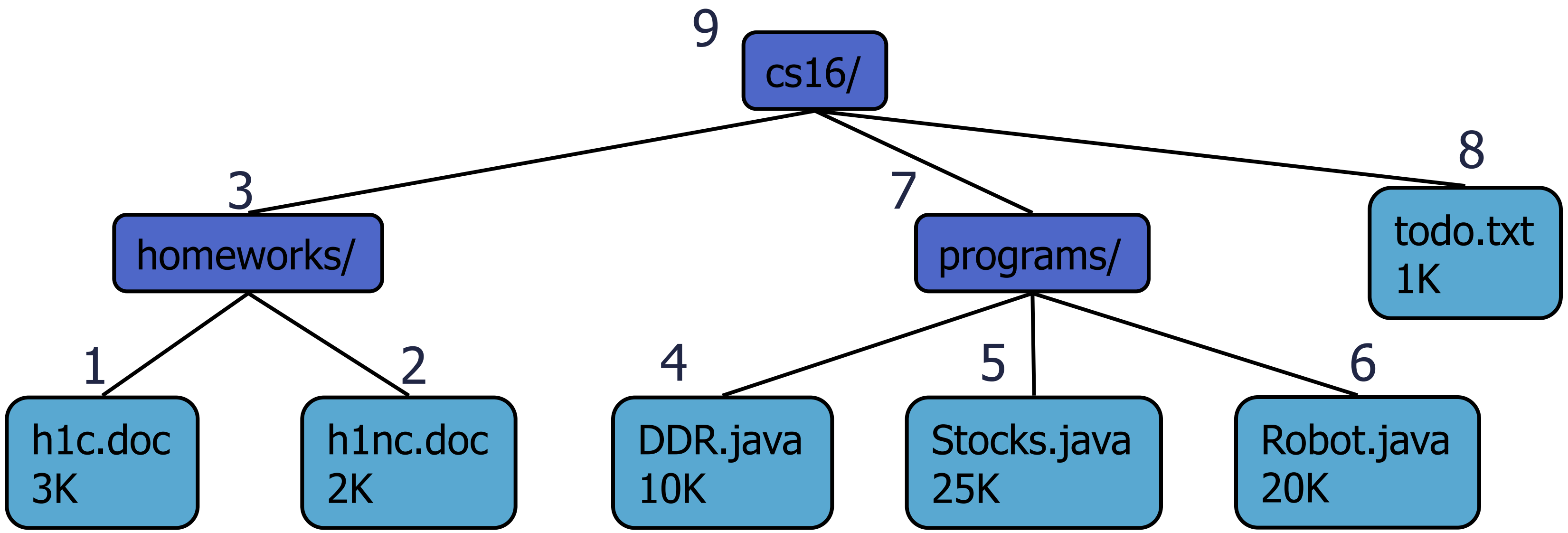
Algorithm ***preOrder***(v)
    visit(v)
    for each child w of v
        preorder (w)



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

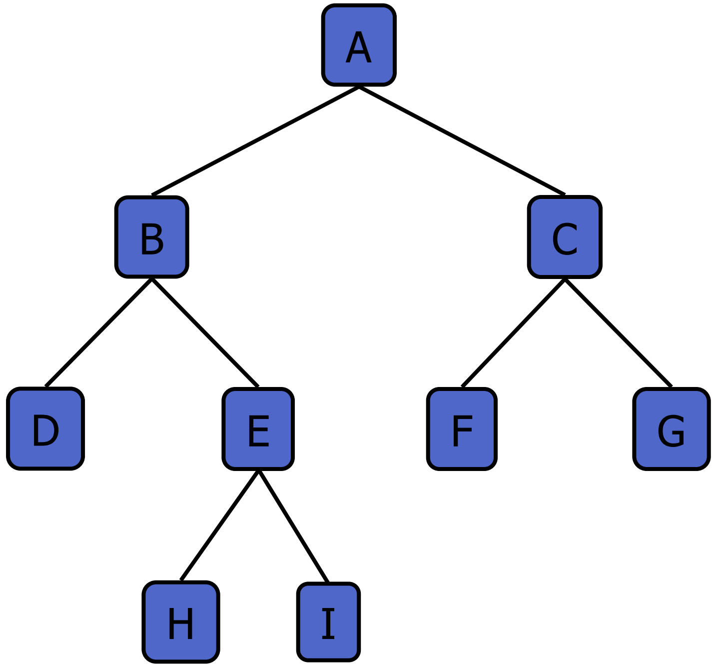
Algorithm *postOrder*(*v*)
    for each child *w* of *v*
        postOrder (*w*)
    visit(*v*)



Binary Trees

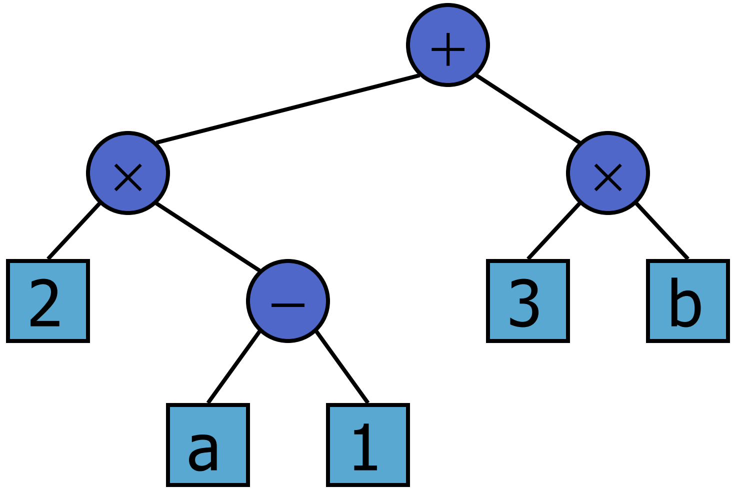
- A binary tree is a tree with the following properties:
  - Each internal node has at most two children (exactly two for proper binary trees)
  - The children of a node are an ordered pair
- We call the children of an internal node left child and right child
- Alternative recursive definition: a binary 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



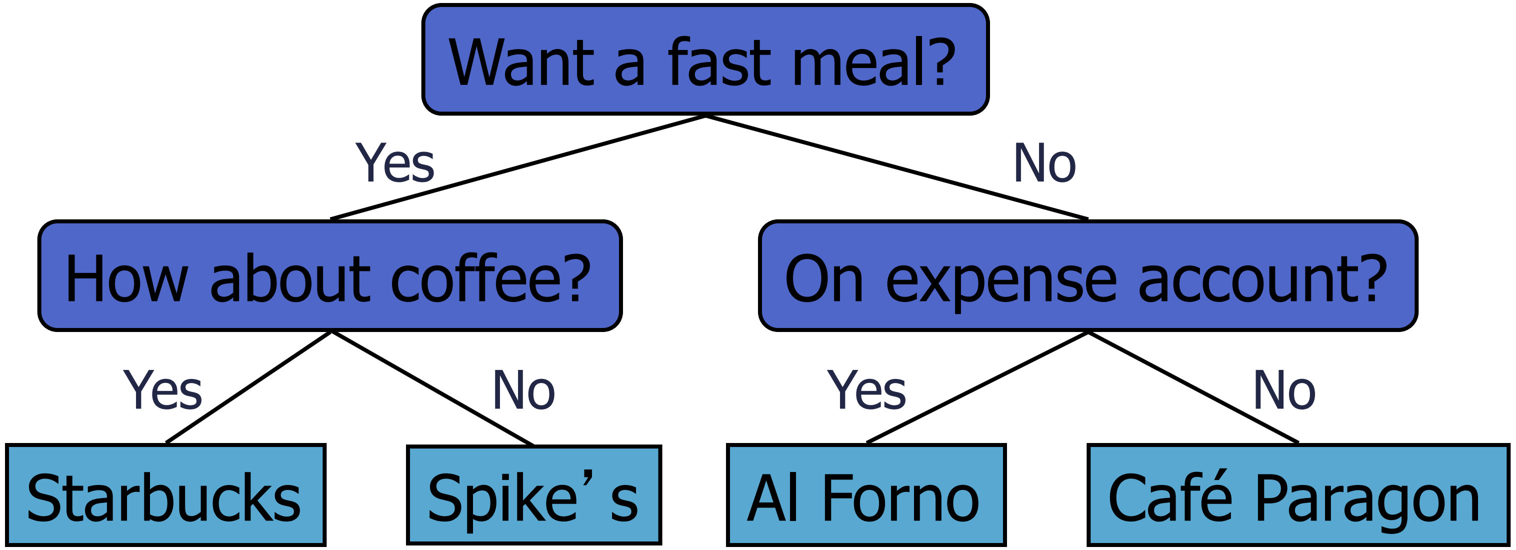
Arithmetic Expression Tree

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



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

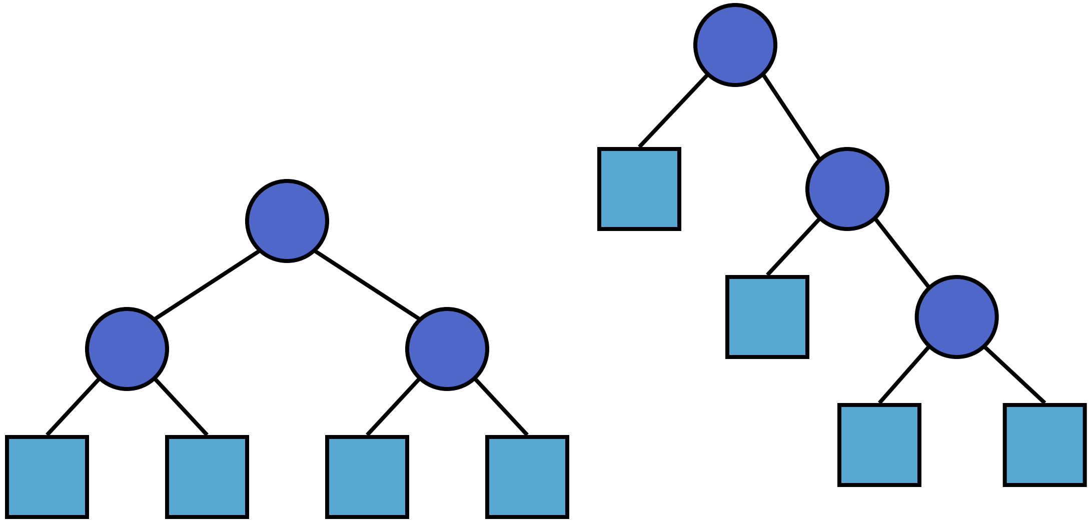
- Notation

  $\boldsymbol{n}$ number of nodes

  $\boldsymbol{e}$ number of external nodes

  $\boldsymbol{i}$ number of internal nodes

  $\boldsymbol{h}$ height



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

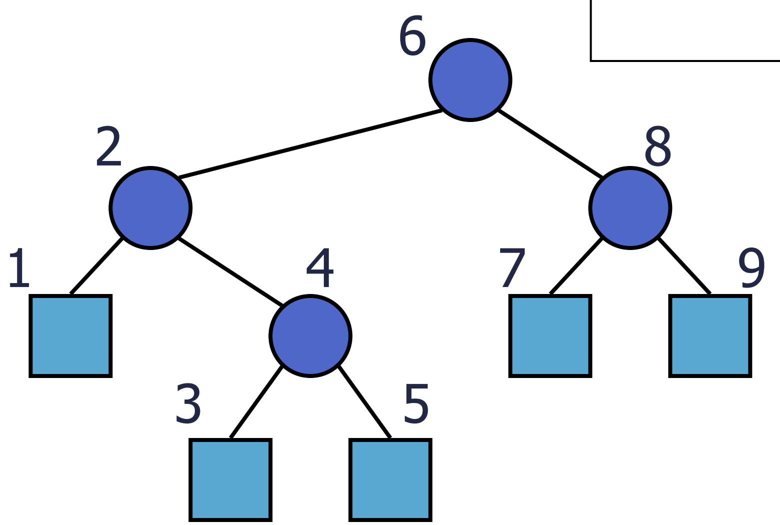
BinaryTree ADT

- 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

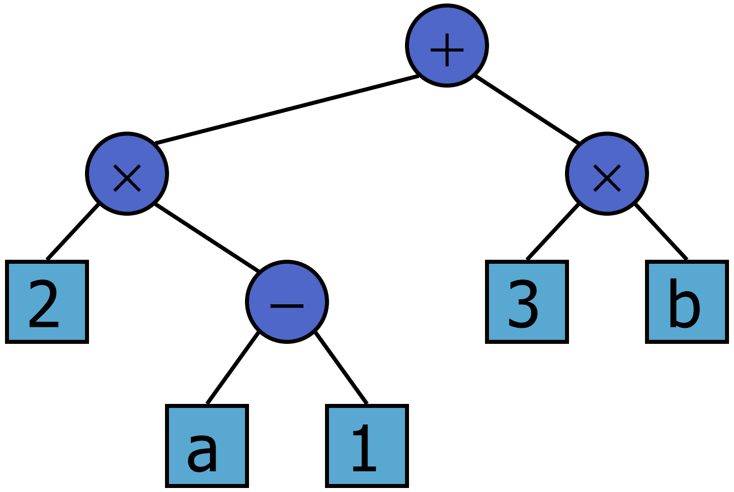
Algorithm ***inOrder***(*v*)
    if *v* **has a left child**
        inOrder (*left* (*v*))
    visit(*v*)
    if *v* **has a right child**
        inOrder (*right* (*v*))



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

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 (“)’ ’)

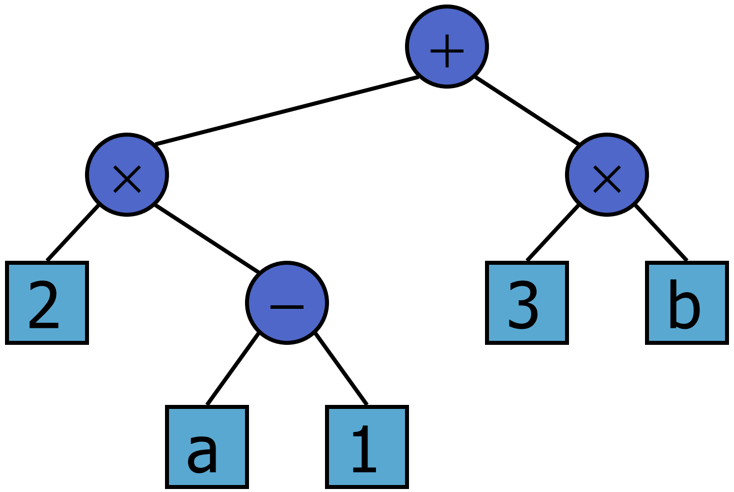


$((2 \times (a - 1)) + (3 \times b))$

Evaluate Arithmetic Expressions

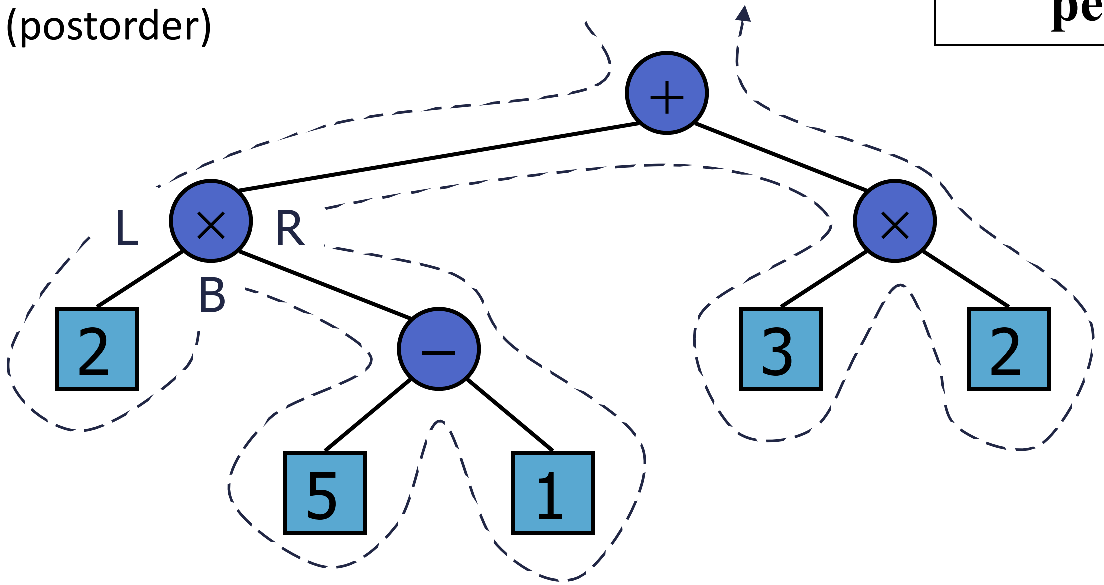
- Specialization of a postorder traversal
  - recursive method returning the value of a subtree
  - when visiting an internal node, combine the values of the subtrees

Algorithm *evalExpr*(v)
    if *is_leaf* (v)
        return *v.element* ()
    else
        $x \leftarrow$ *evalExpr*(*left* (v))
        $y \leftarrow$ *evalExpr*(*right* (v))
        $\Diamond \leftarrow$ operator stored at v
        return $x \Diamond y$



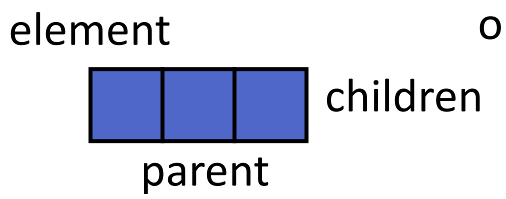
Euler Tour Traversal

- Generic traversal of a binary tree
- Includes a special cases the preorder, postorder and inorder traversals
- Walk around the tree and visit each node three times:
  - on the left (preorder)
  - from below (inorder)
  - on the right (postorder)

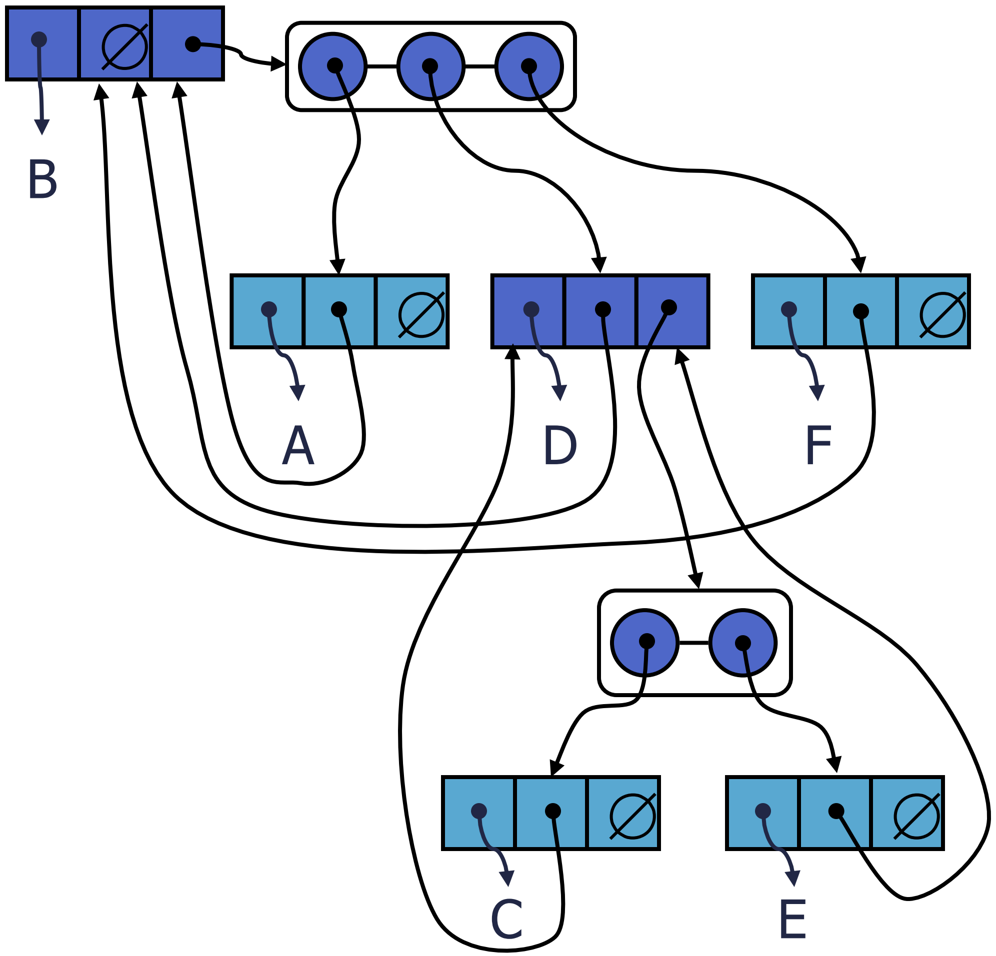


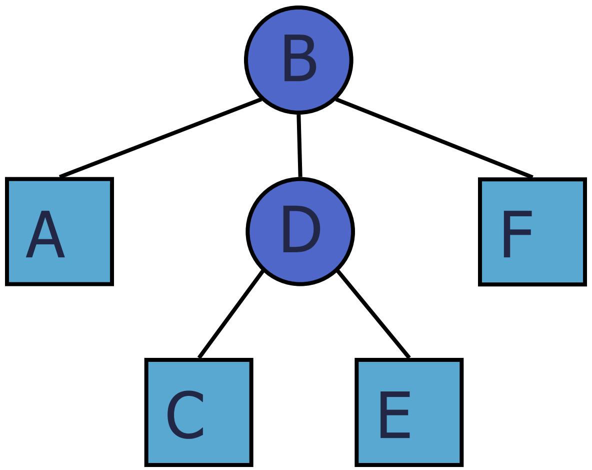
Algorithm ***eulertour***(*T*, *p*)
  perform *pre_visit (p)*
  for each child *c* in *T.children(p)* do
    eulertour(*T*, *c*)
  perform *post_visit (p)*

Linked Structure for General Trees



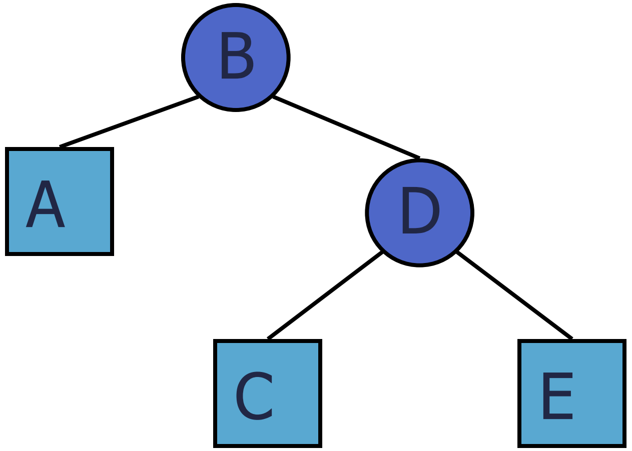
- A node is represented by an object storing
  - Element
  - Parent node
  - Sequence of children nodes
- Node objects implement the Position ADT

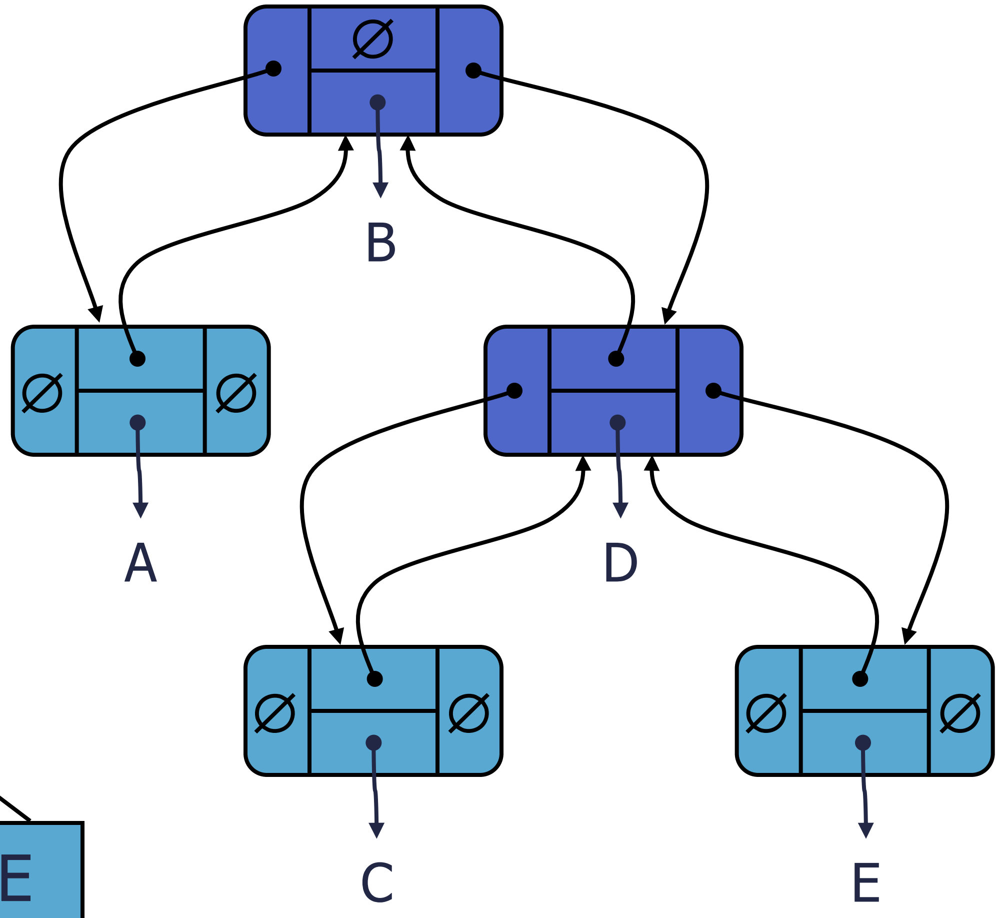




Linked Structure for Binary Trees

- A node is represented by an object storing
  - Element
  - Parent node
  - Left child node
  - Right child node
- Node objects implement the Position ADT





Exercise

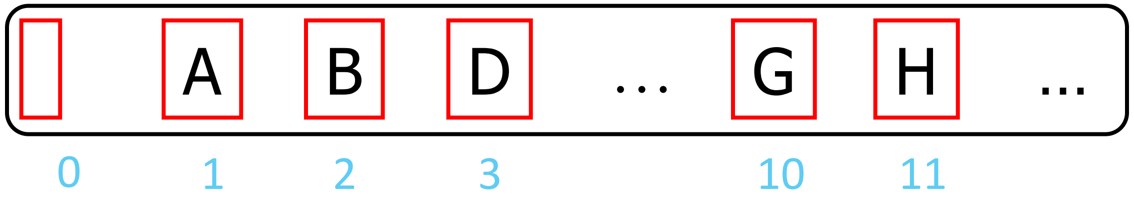
- In notebooks/Exercise folder, the following files are given as a template.
  - tree.py
  - binary_tree.py
  - linked_binary_tree.py

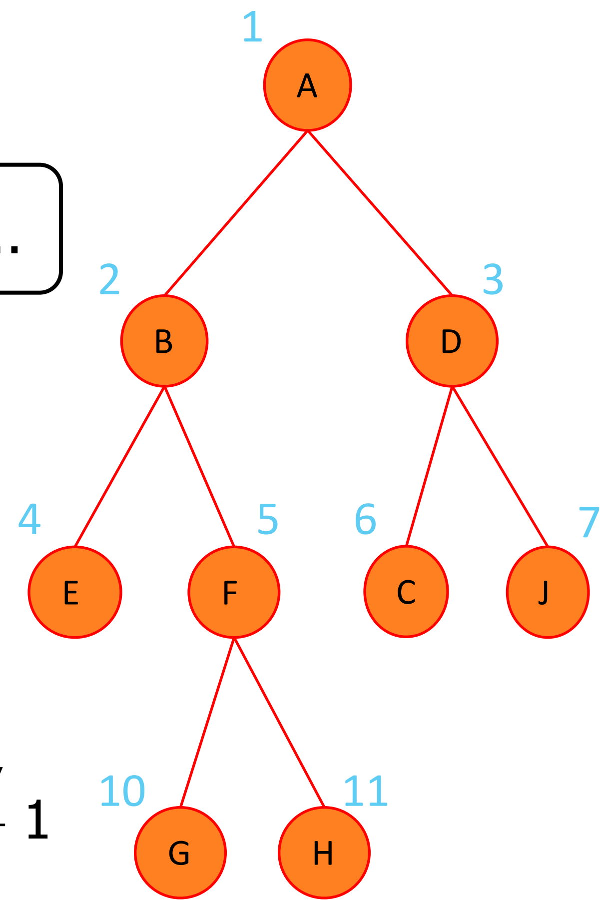
- Try not looking at the solution, fill in the blanks in the codes, in the order of the files presented above

```
raise NotImplementedError('EXERCISE')
```

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