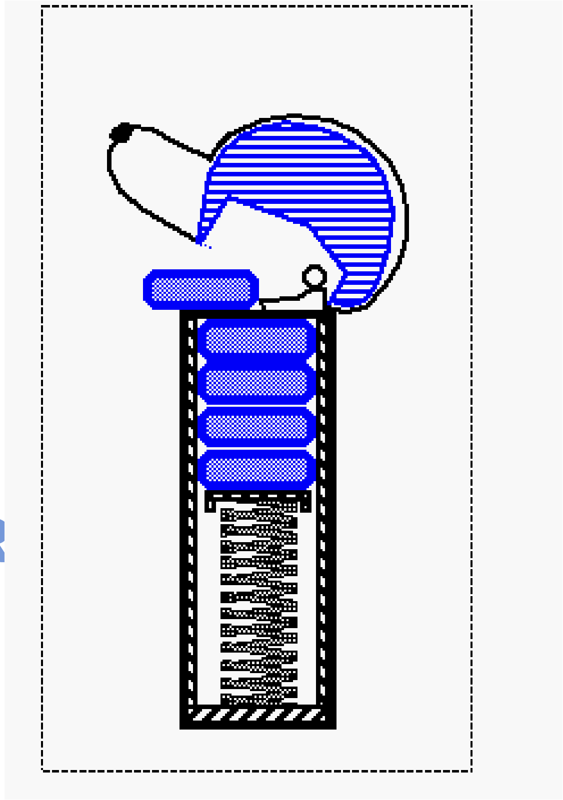
Stacks, Queues, & Lists

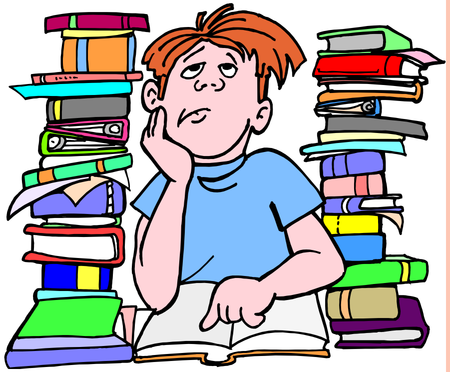
Amortized analysis

Trees

Elementary Data Str



The Stack ADT (§2.1.1)



- The Stack ADT (Abstract Data Type) stores arbitrary objects
- Insertions and deletions follow the last-in first-out scheme
- Think of a spring-loaded plate dispenser
- Main stack operations:
  - push(object): inserts an element
  - object pop(): removes and returns the last inserted element
- Auxiliary stack operations:
  - object top(): returns the last inserted element without removing it
  - integer size(): returns the number of elements stored
  - boolean isEmpty(): indicates whether no elements are stored

Applications of Stacks

- Direct applications
  - Page-visited history in a Web browser
  - Undo sequence in a text editor
  - Chain of method calls in the Java Virtual Machine or C++ runtime environment
- Indirect applications
  - Auxiliary data structure for algorithms
  - Component of other data structures

Method Stack in the JVM

- The Java Virtual Machine (JVM) keeps track of the chain of active methods with a stack
- When a method is called, the JVM pushes on the stack a frame containing
  - Local variables and return value
  - Program counter, keeping track of the statement being executed
- When a method ends, its frame is popped from the stack and control is passed to the method on top of the stack

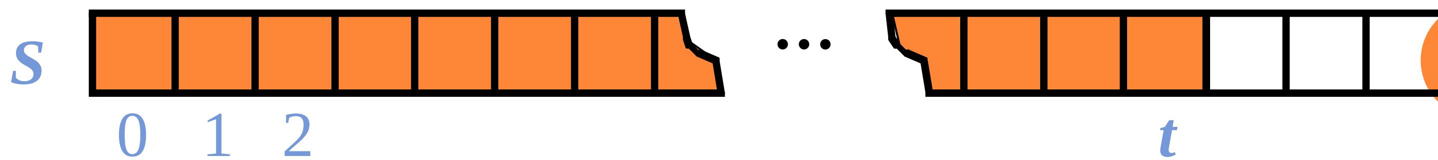
```
main() {
    int i = 5;
    foo(i);
}

foo(int j) {
    int k;
    k = j+1;
    bar(k);
}

bar(int m) {
    …
}
```

ARRAY-BASED STACK (§2.1.1)

- A simple way of implementing the Stack ADT uses an array
- We add elements from left to right
- A variable t keeps track of the index of the top element (size is t+1)

Algorithm *pop*():
  if *isEmpty*() **then**
    throw ***EmptyStackException***
  else
    $t \leftarrow t - 1$
    return $S[t + 1]$

Algorithm *push*(*o*)
  if $t = S.length - 1$ **then**
    throw ***FullStackException***
  else
    $t \leftarrow t + 1$
    $S[t] \leftarrow o$



Performance and Limitations

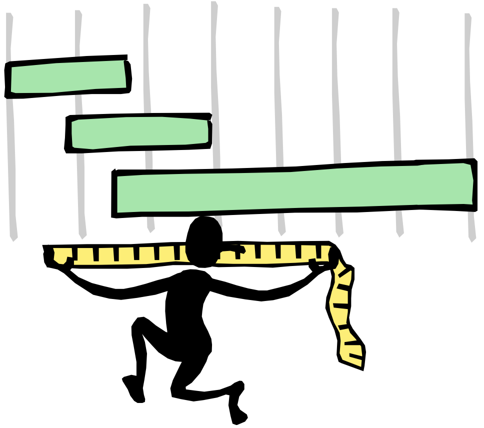
- Performance
  - Let n be the number of elements in the stack
  - The space used is $O(n)$
  - Each operation runs in time $O(1)$
- Limitations
  - The maximum size of the stack must be defined a priori and cannot be changed
  - Trying to push a new element into a full stack causes an implementation-specific exception

Growable Array-based Stack (§1.5)

- In a push operation, when the array is full, instead of throwing an exception, we can replace the array with a larger one
- How large should the new array be?
  - incremental strategy: increase the size by a constant c
  - doubling strategy: double the size

Algorithm *push*(*o*)
  if $t = S.length - 1$ **then**
    $A \leftarrow$ new array of size …
    for $i \leftarrow 0$ **to** t **do**
      $A[i] \leftarrow S[i]$
      $S \leftarrow A$
  $t \leftarrow t + 1$
  $S[t] \leftarrow o$

Comparison of the Strategies



- We compare the incremental strategy and the doubling strategy by analyzing the total time $T(n)$ needed to perform a series of n push operations
- We assume that we start with an empty stack represented by an array of size 1
- We call **amortized time** of a push operation the average time taken by a push over the series of operations, i.e., $T(n)/n$

Analysis of the Incremental Strategy

- We replace the array $k = n/c$ times
- The total time $T(n)$ of a series of n push operations is proportional to

$$n + c + 2c + 3c + 4c + \ldots + kc =$$
$$n + c(1 + 2 + 3 + \ldots + k) =$$
$$n + ck(k + 1)/2$$

- Since c is a constant, $T(n)$ is $O(n + k^2)$, i.e., $O(n^2)$
- The amortized time of a push operation is $O(n)$

DIRECT ANALYSIS OF THE DOUBLING STRATEGY

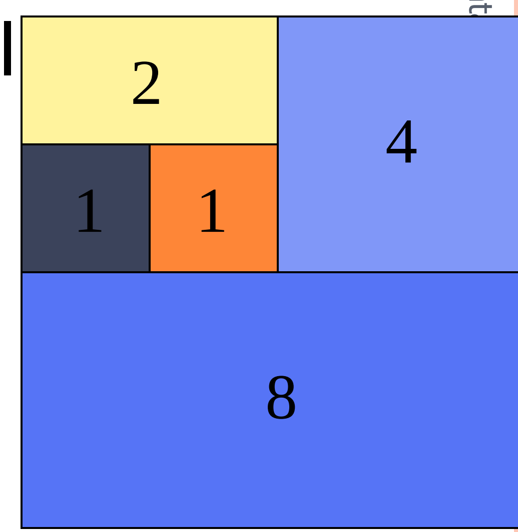
- We replace the array $k = \log_2 n$ times
- The total time $T(n)$ of a series of n push operations is proportional to

$$n + 1 + 2 + 4 + 8 + \ldots + 2^k =$$

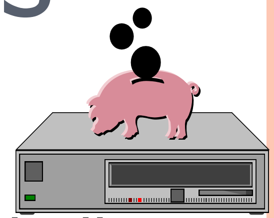
$$n + 2^{k+1} - 1 = 2n - 1$$

- $T(n)$ is $O(n)$
- The amortized time of a push operation is $O(1)$

geometric series



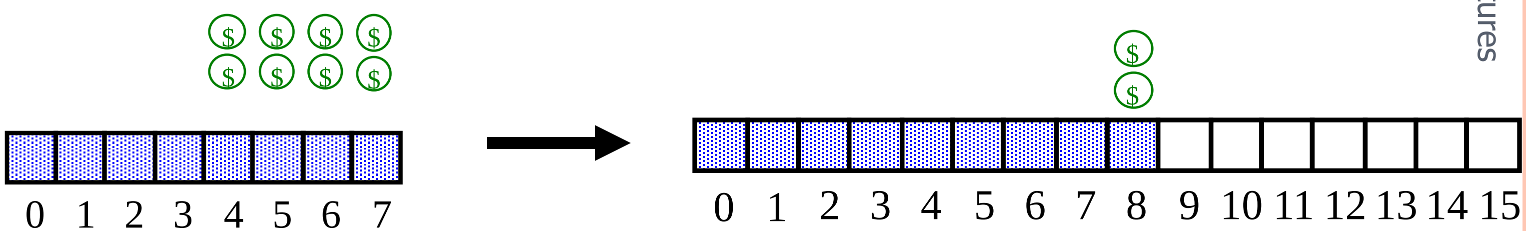
Accounting Method Analysis of the Doubling Strategy



- **Amortization:** to pay of gradually by making periodic payments
- Rather than focusing on each operation separately, it consider the running time of a series of these operations.
- We view a computer as a coin-operated device requiring 1 cyber-dollar for a constant amount of computing.
  - We set up a scheme for charging operations. This is known as an **amortization scheme**.
  - The scheme must give us always enough money to pay for the actual cost of the operation.
  - The total cost of the series of operations is no more than the total amount charged.
- (amortized time) ≤ (total $ charged) / (# operations)

Amortization Scheme for the Doubling Strategy

- Consider again the *k* phases, where each phase consisting of twice as many pushes as the one before.
  - It costs one cyber-dollar for to push one element, excluding the growth of the array.
  - Growing the array from k to 2k costs k cyber-dollars for copying elements.
- At the end of a phase we must have saved enough to pay for the array-growing push of the next phase.
- At the end of phase *i* we want to have saved *i* cyber-dollars, to pay for the array growth for the beginning of the next phase.



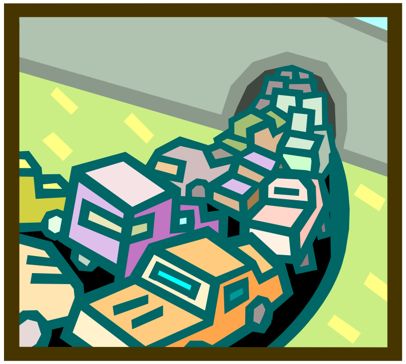
- We charge $3 for a push. The $2 saved for a regular push are "stored" in the second half of the array. Thus, we will have $2(i/2)=i$ cyber-dollars saved at then end of phase *i*.
- Therefore, each push runs in $O(1)$ amortized time; *n* pushes run in $O(n)$ time.

The Queue ADT (§2.1.2)

- The Queue ADT stores arbitrary objects
- Insertions and deletions follow the first-in first-out scheme
- Insertions are at the rear of the queue and removals are at the front of the queue
- Main queue operations:
  - enqueue(object): inserts an element at the end of the queue
  - object dequeue(): removes and returns the element at the front of the queue

- Auxiliary queue operations:
  - object front(): returns the element at the front without removing it
  - integer size(): returns the number of elements stored
  - boolean isEmpty(): indicates whether no elements are stored
- Exceptions
  - Attempting the execution of dequeue or front on an empty queue throws an EmptyQueueException

Applications of Queues

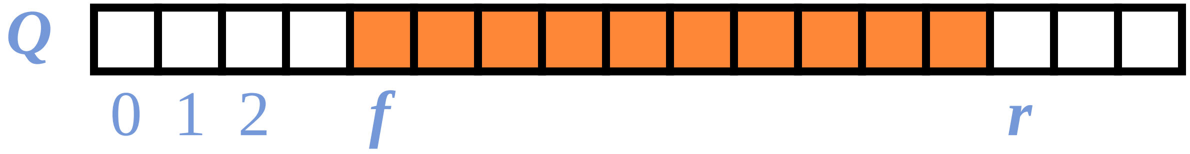


- Direct applications
  - Waiting lines
  - Access to shared resources (e.g., printer)
  - Multiprogramming
- Indirect applications
  - Auxiliary data structure for algorithms
  - Component of other data structures

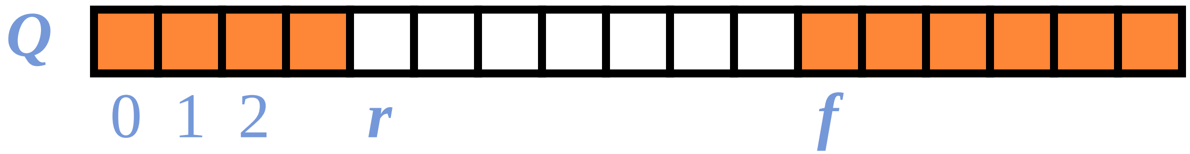
ARRAY-BASED QUEUE

- Use an array of size N in a circular fashion
- Two variables keep track of the front and rear
  - f index of the front element
  - r index immediately past the rear element
- Array location r is kept empty

normal configuration



wrapped-around configuration

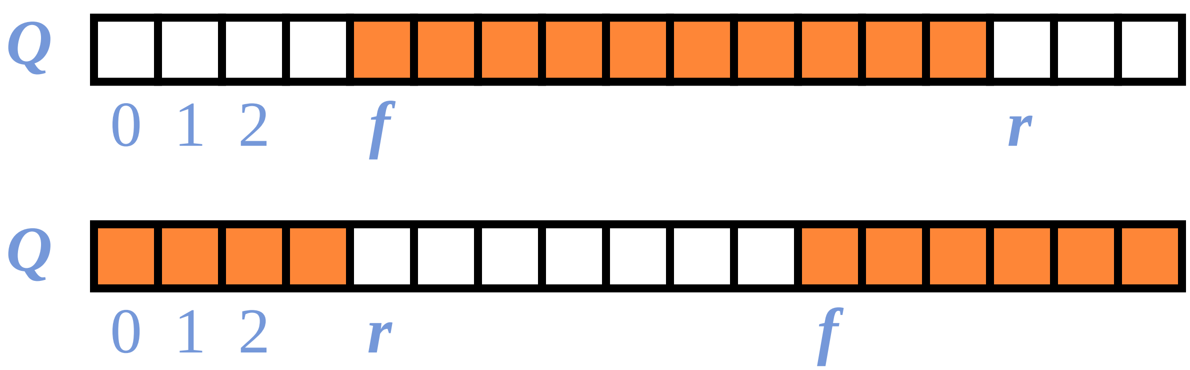


Queue Operations

- We use the modulo operator (remainder of division)

Algorithm *size*()
  return $(N - f + r) \bmod N$

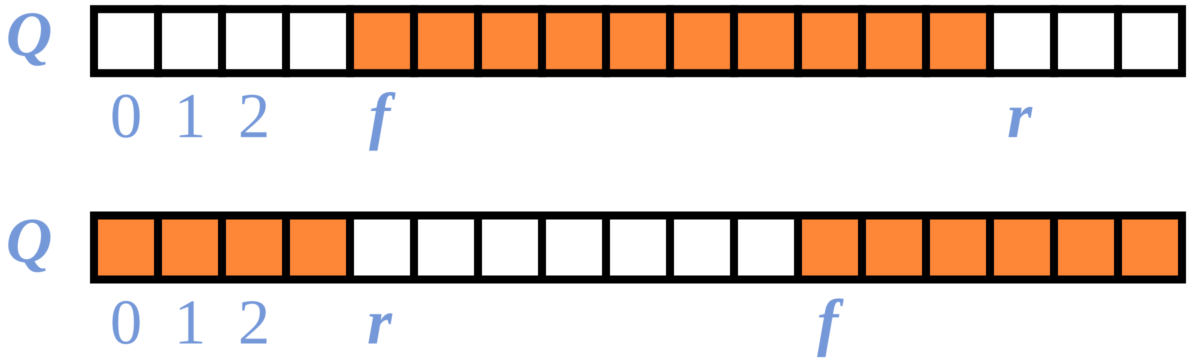
Algorithm *isEmpty*()
  return $(f = r)$



Queue Operations (cont.)

- Operation enqueue throws an exception if the array is full
- This exception is implementation-dependent

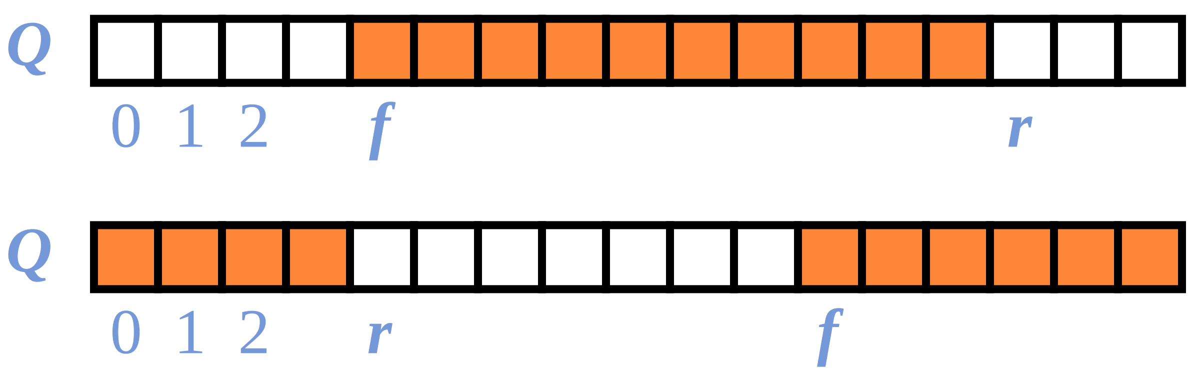
Algorithm *enqueue*(*o*)
  if *size*() = $N - 1$ **then**
    throw *FullQueueException*
  else
    $Q[r] \leftarrow o$
    $r \leftarrow (r + 1) \bmod N$



Queue Operations (cont.)

- Operation dequeue throws an exception if the queue is empty
- This exception is specified in the queue ADT

```
Algorithm dequeue()
  if isEmpty() then
    throw EmptyQueueException
  else
    o ← Q[f]
    f ← (f + 1) mod N
    return o
```

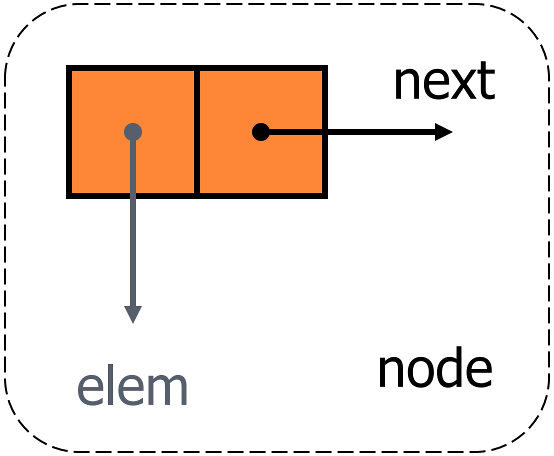


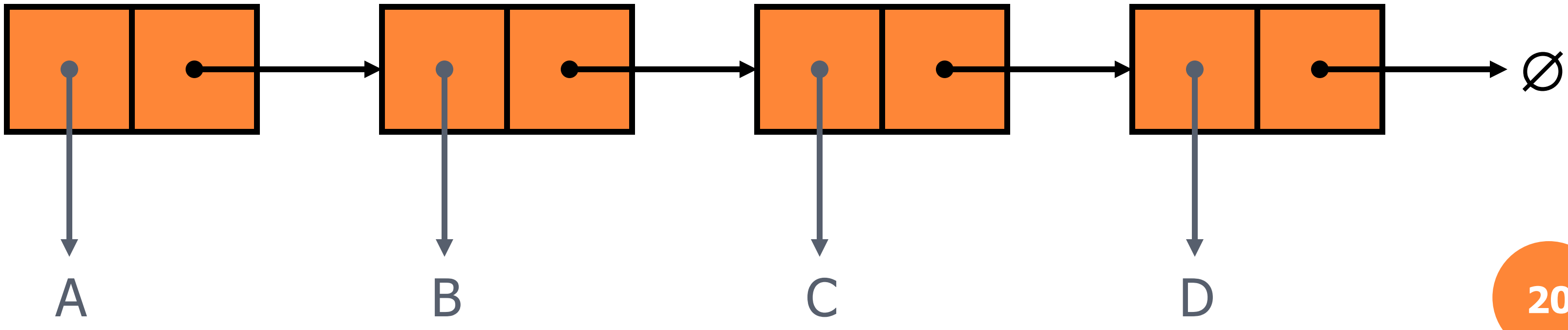
Growable Array-based Queue

- In an enqueue operation, when the array is full, instead of throwing an exception, we can replace the array with a larger one
- Similar to what we did for an array-based stack
- The enqueue operation has amortized running time
  - $O(n)$ with the incremental strategy
  - $O(1)$ with the doubling strategy

Singly Linked List

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

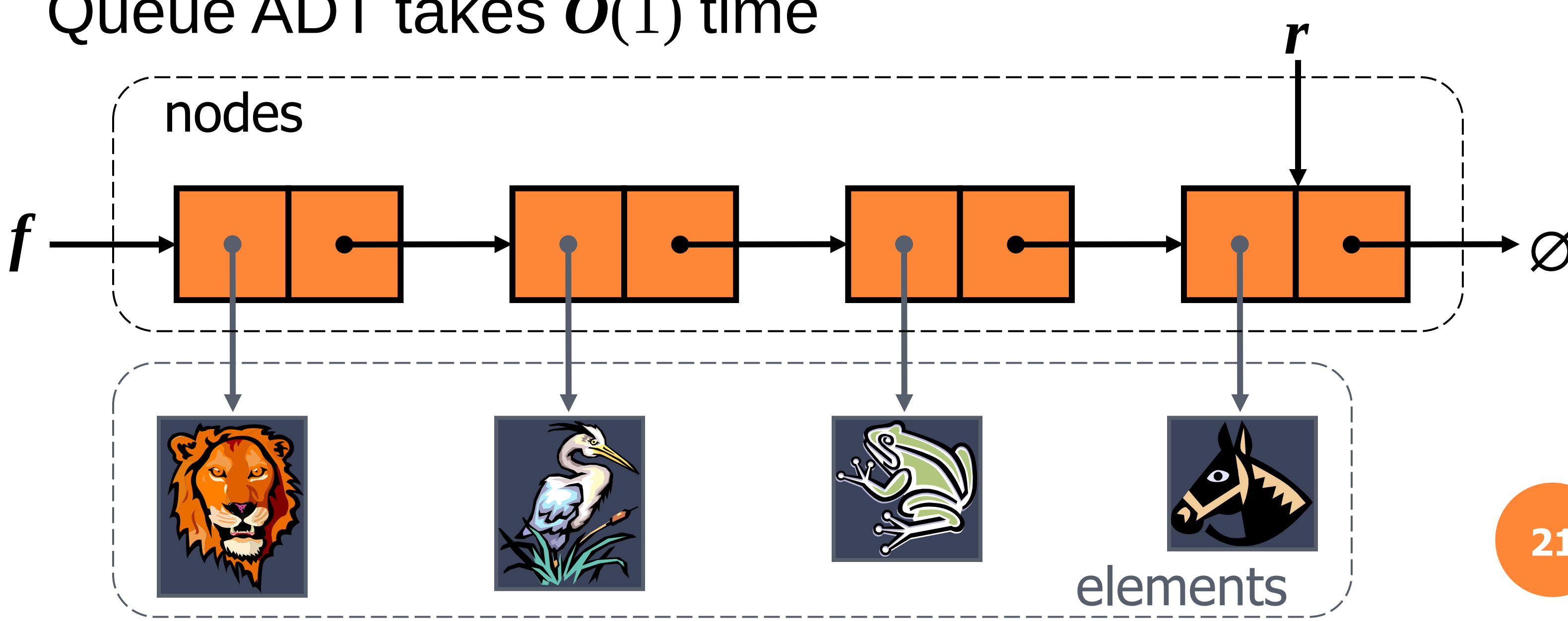




Elementary Data Structures

QUEUE WITH A SINGLY 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



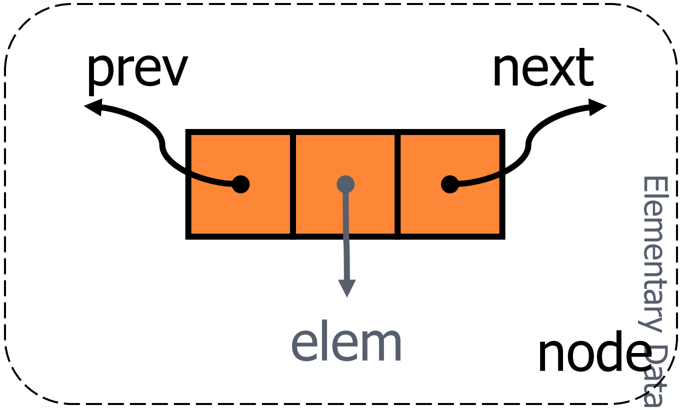
List ADT (§2.2.2)

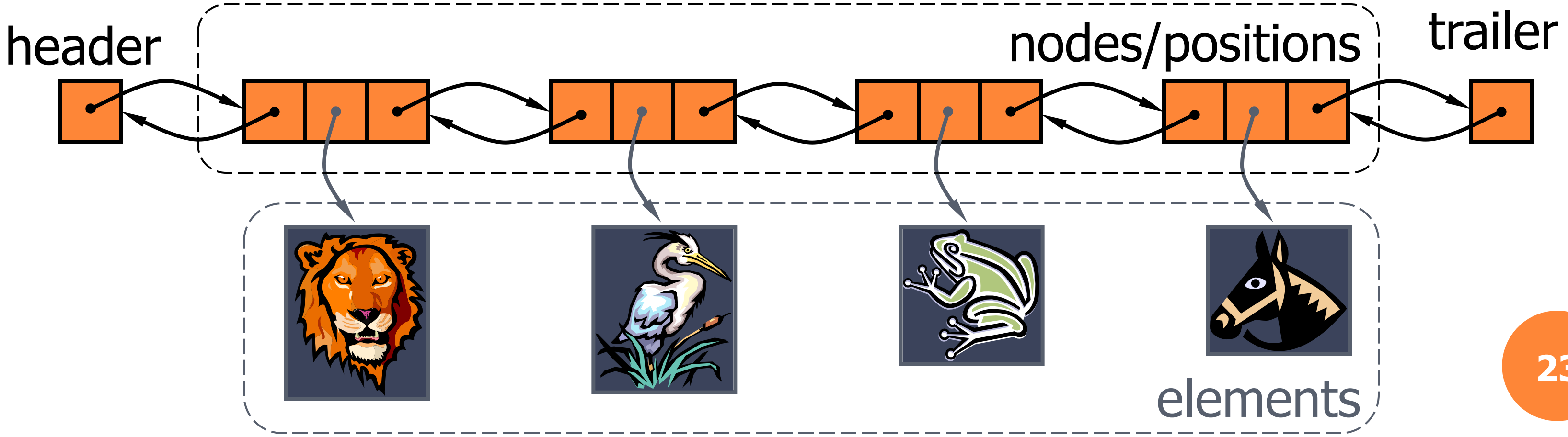
- The List ADT models a sequence of **positions** storing arbitrary objects
- It allows for insertion and removal in the “middle”
- Query methods:
  - isFirst(p), isLast(p)

Accessor methods:
  - first(), last()
  - before(p), after(p)
- Update methods:
  - replaceElement(p, o), swapElements(p, q)
  - insertBefore(p, o), insertAfter(p, o),
  - insertFirst(o), insertLast(o)
  - remove(p)

Doubly Linked List

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



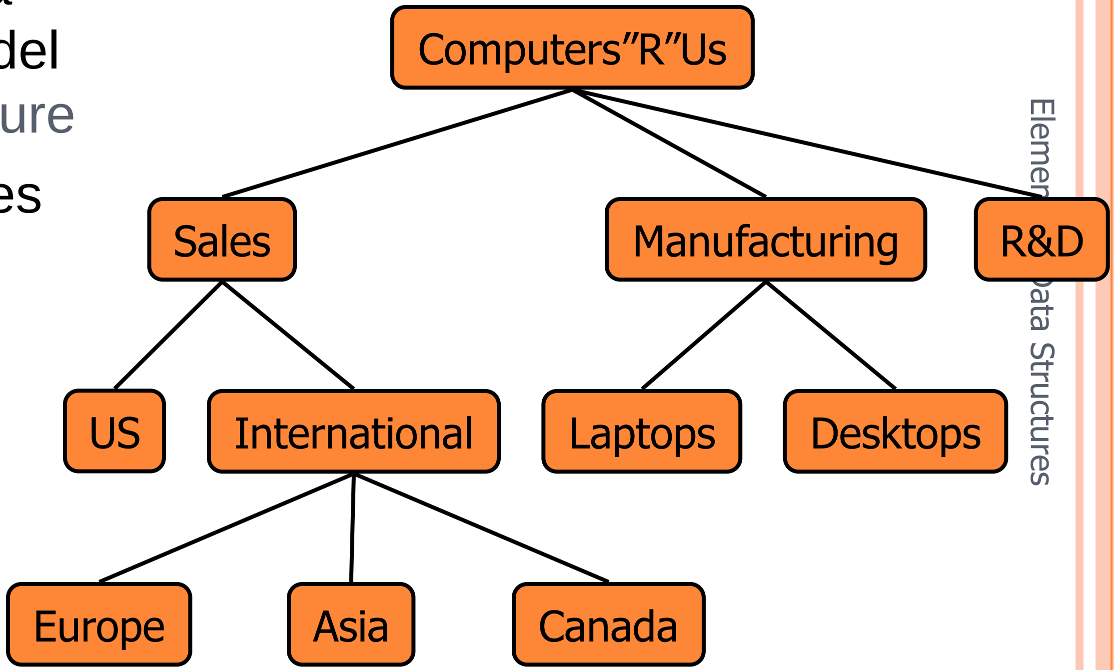


List ADT

- How about array-based List?

TREES (§2.3)

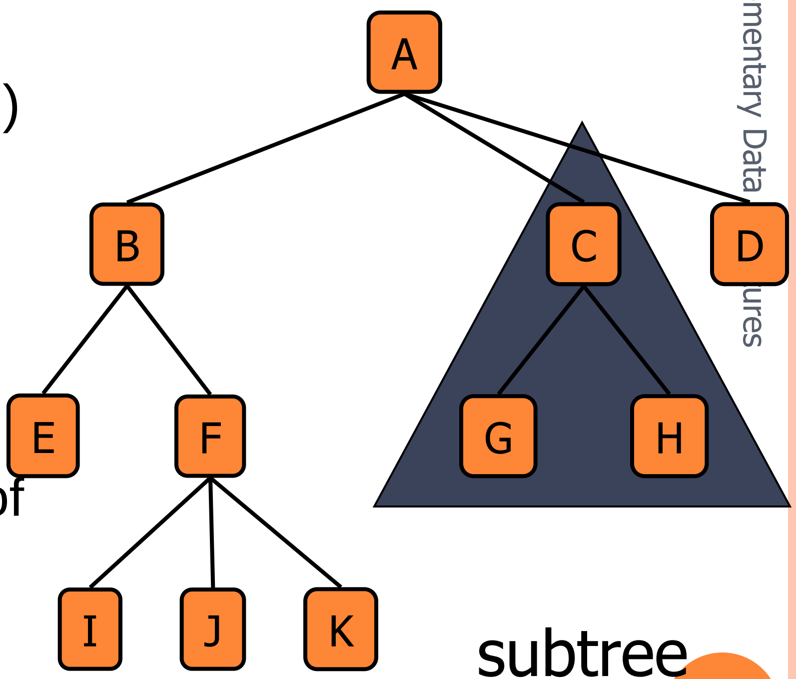
- In computer science, a tree is an abstract model of a hierarchical structure
- A tree consists of nodes with a parent-child relation
- Applications:
  - Organization charts
  - File systems
  - Programming environments



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



TREE ADT (§2.3.1)

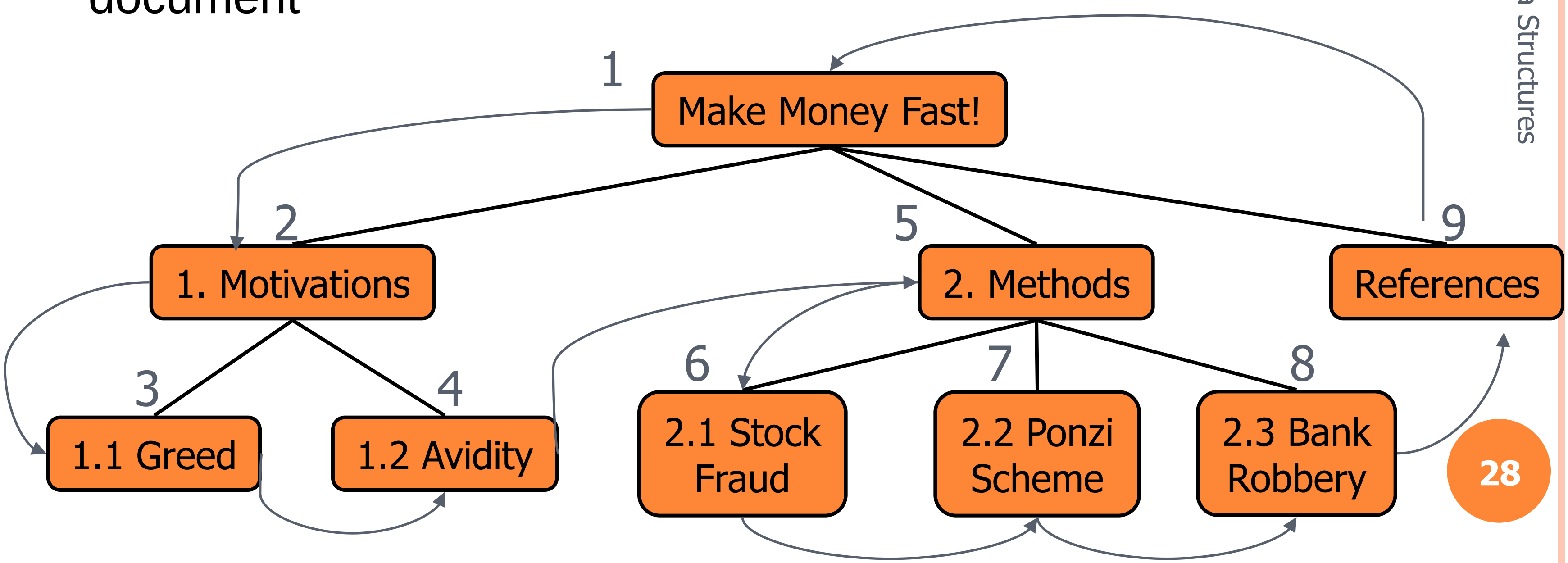
- Generic methods:
  - integer size()
  - boolean isEmpty()
  - objectIterator elements()
  - positionIterator positions()
- Accessor methods:
  - position root()
  - position parent(p)
  - positionIterator children(p)

- Query methods:
  - boolean isInternal(p)
  - boolean isExternal(p)
  - boolean isRoot(p)
- Update methods:
  - swapElements(p, q)
  - object replaceElement(p, o)
- Additional update methods may be defined by data structures implementing the Tree ADT

Preorder Traversal (§2.3.2)

- 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

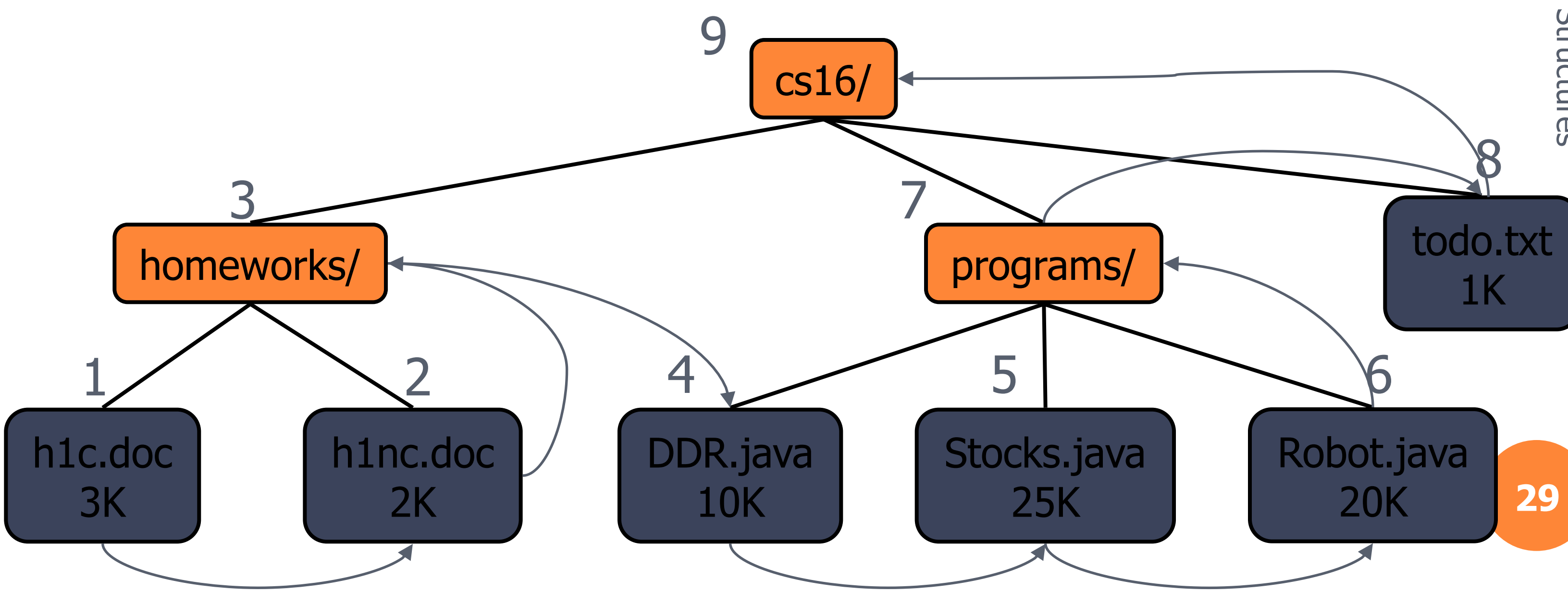
Algorithm *preOrder*(*v*)
    visit(*v*)
    for each child *w* of *v*
        preorder (*w*)



Postorder Traversal (§2.3.2)

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

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



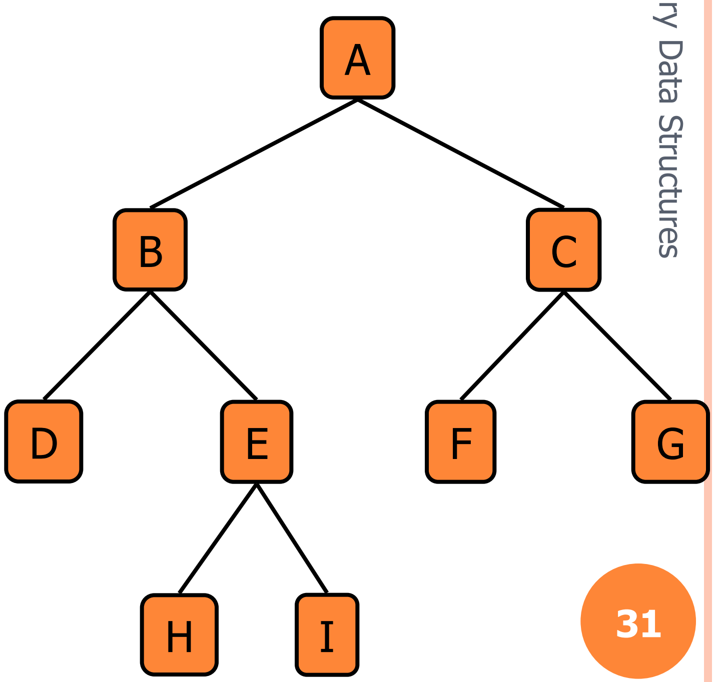
Amortized Analysis of Tree Traversal

- Time taken in preorder or postorder traversal of an *n*-node tree is proportional to the sum, taken over each node v in the tree, of the time needed for the recursive call for v.
  - The call for v costs \$($c_v + 1$), where c_v is the number of children of v
  - For the call for v, charge one cyber-dollar to v and charge one cyber-dollar to each child of v.
  - Each node (except the root) gets charged twice: once for its own call and once for its parent's call.
  - Therefore, traversal time is **O(n).**

BINARY TREES (§2.3.3)

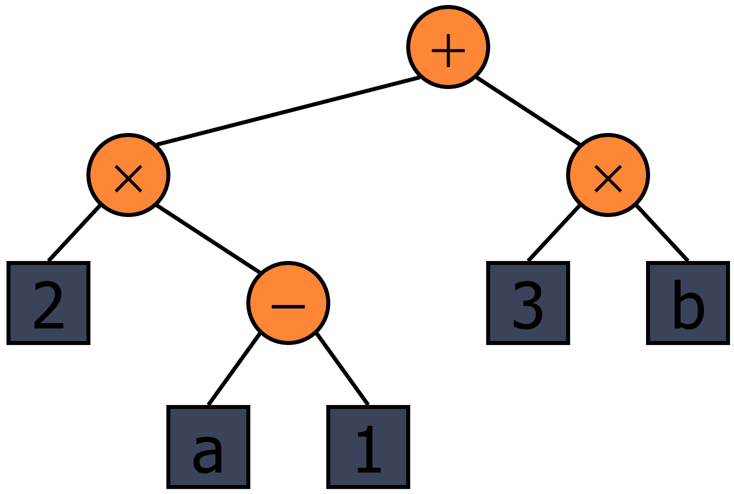
- A binary tree is a tree with the following properties:
  - Each internal node has at most two children (proper)
  - The children of a node are an ordered pair (left child comes before right child)
- We call the children of an internal node left child and right child
- Alternative recursive definition: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



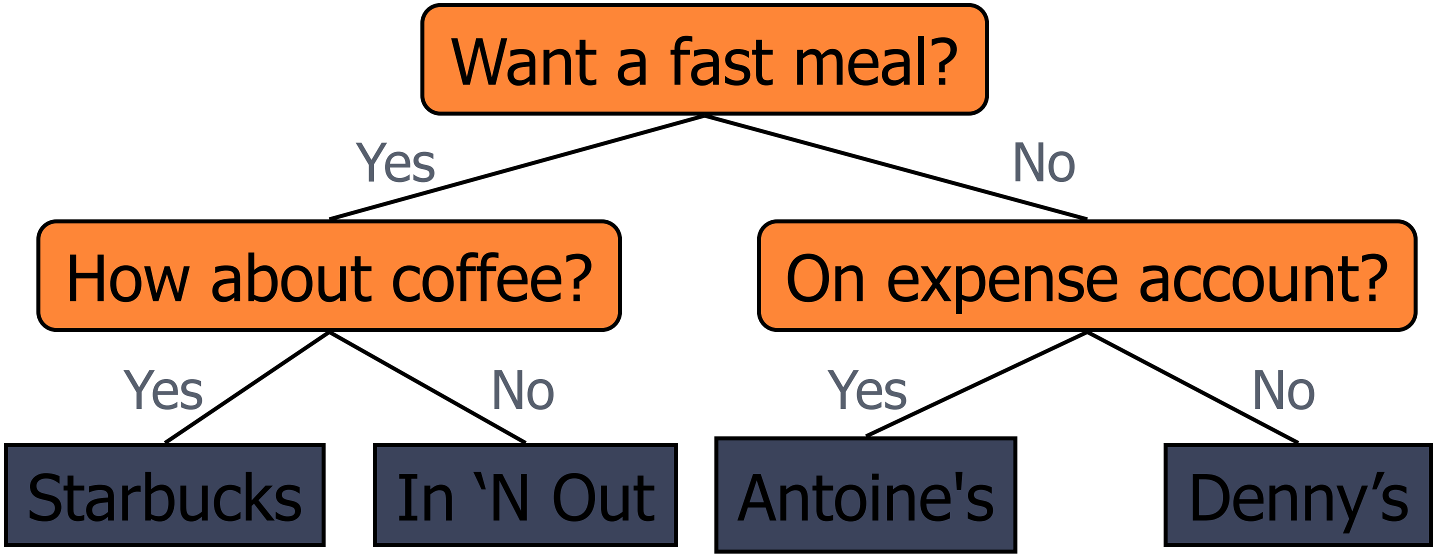
ARITHMETIC EXPRESSION TREE

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



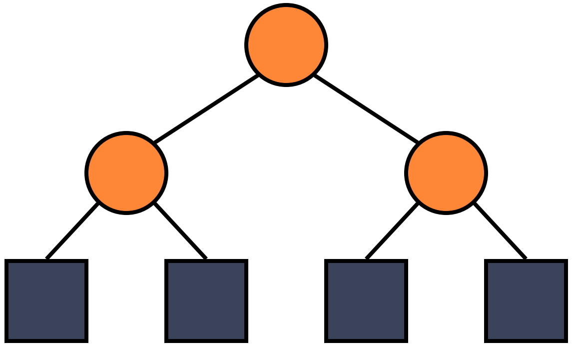
DECISION TREE

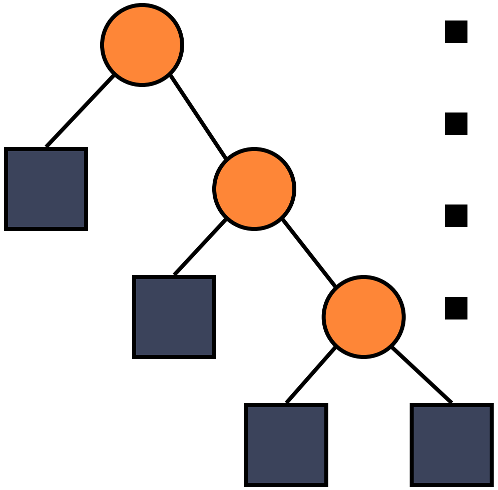
- Binary tree associated with a decision process
  - internal nodes: questions with yes/no answer
  - external nodes: decisions
- Example: dining decision



Properties of Binary Trees

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



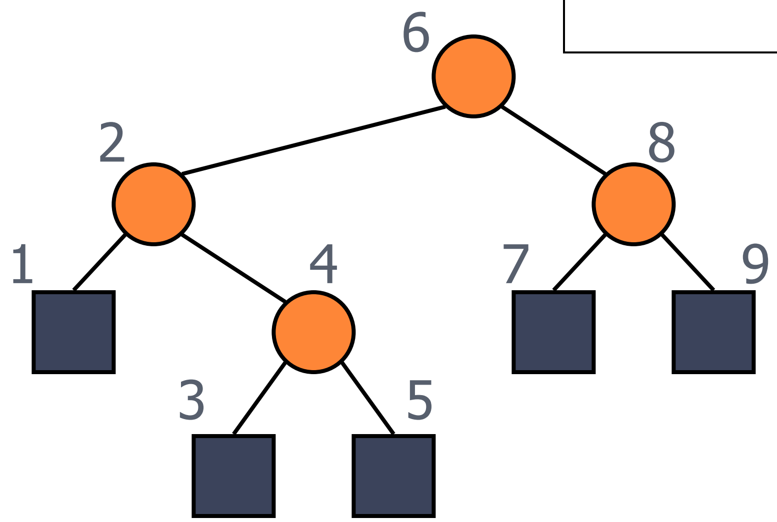


- Properties:
  - $e = i + 1$
  - $n = 2e - 1$
  - $h \leq i$
  - $h \leq (n - 1)/2$
  - $e \leq 2^h$
  - $h \geq \log_2 e$
  - $h \geq \log_2 (n + 1) - 1$

Inorder Traversal

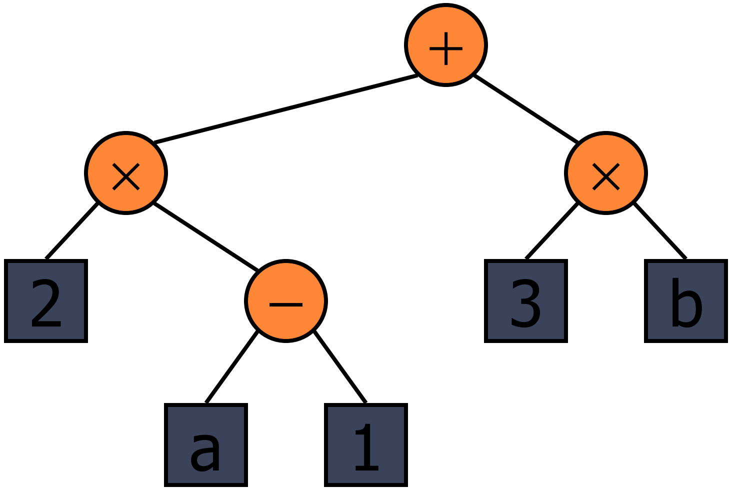
- In an inorder traversal a node is visited after its left subtree and before its right subtree

```
Algorithm inOrder(v)
    if isInternal (v)
        inOrder (leftChild (v))
    visit(v)
    if isInternal (v)
        inOrder (rightChild (v))
```



PRINTING ARITHMETIC EXPRESSIONS

- Specialization of an inorder traversal
  - print “(“ before traversing left subtree
  - print operand or operator when visiting node
  - print “)“ after traversing right subtree



Algorithm *printExpression*(*v*)

    if *isInternal* (*v*)

        print(“(’’)

        inOrder (*leftChild* (*v*))

    print(*v.element* ())

    if *isInternal* (*v*)

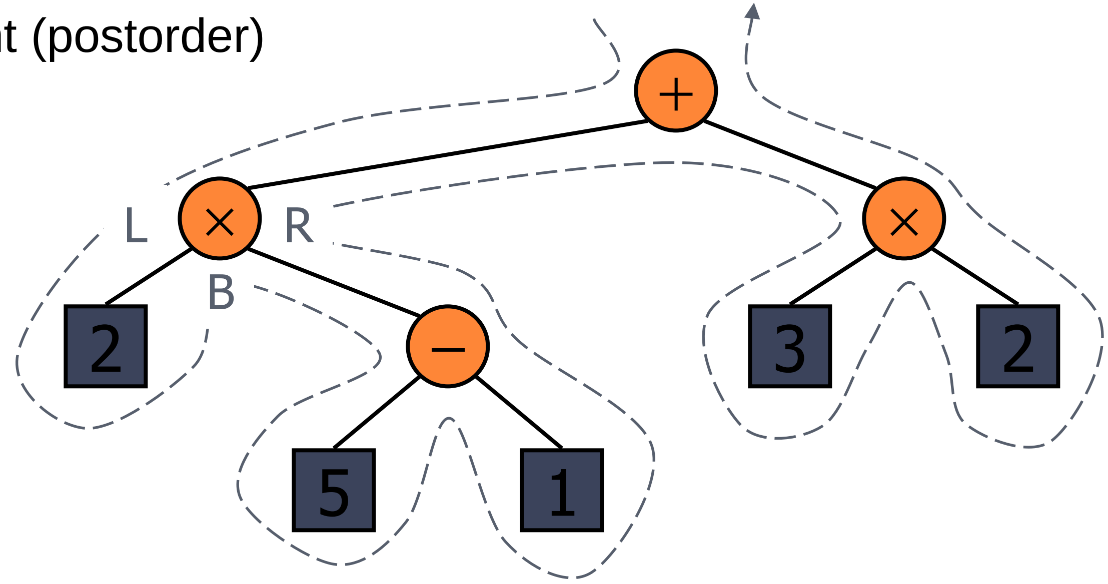
        inOrder (*rightChild* (*v*))

        print (“)’’)

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

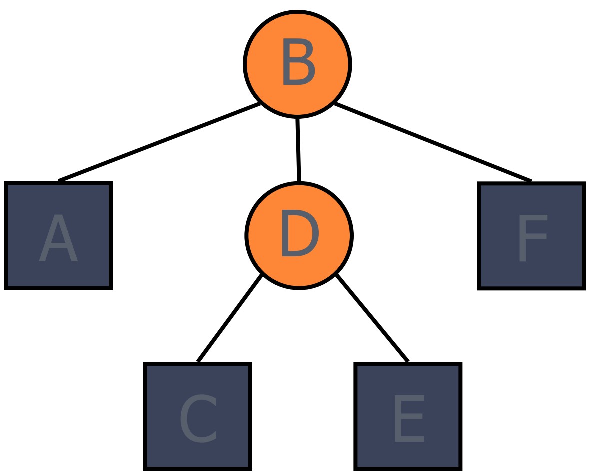
EULER TOUR TRAVERSAL

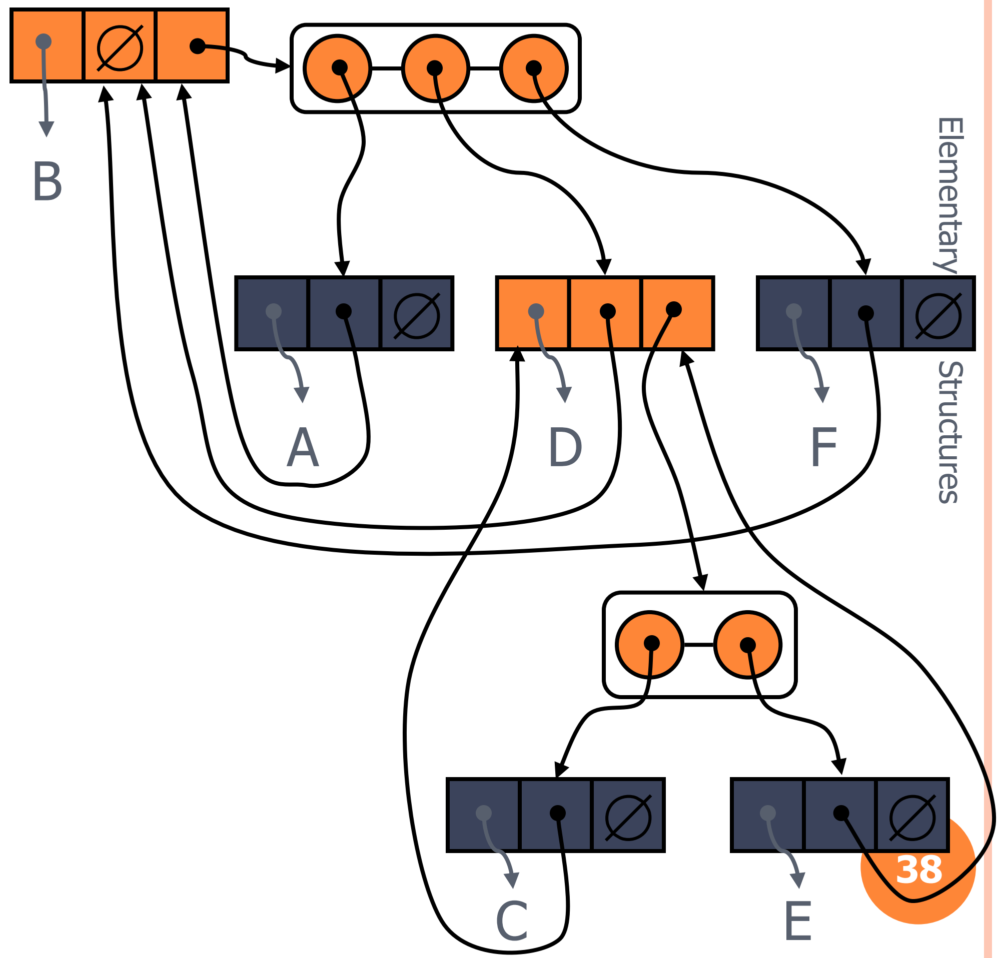
- Generic traversal of a binary tree
- Includes a special cases the preorder, postorder and inorder traversals
- Walk around the tree and visit each node three times:
  - on the left (preorder)
  - from below (inorder)
  - on the right (postorder)



LINKED DATA STRUCTURE FOR REPRESENTING TREES (§2.3.4)

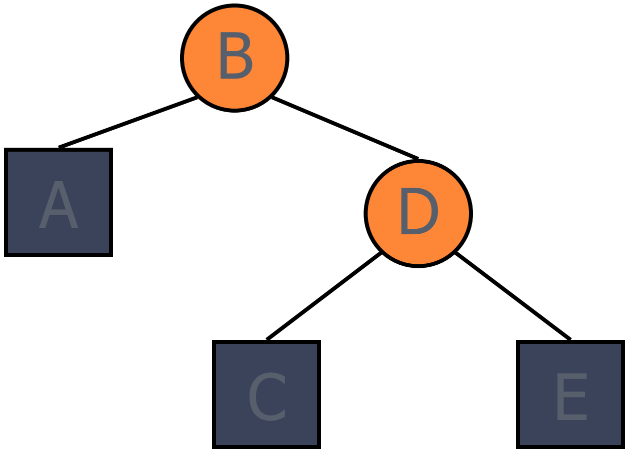
- A node is represented by an object storing
  - Element
  - Parent node
  - Sequence of children nodes

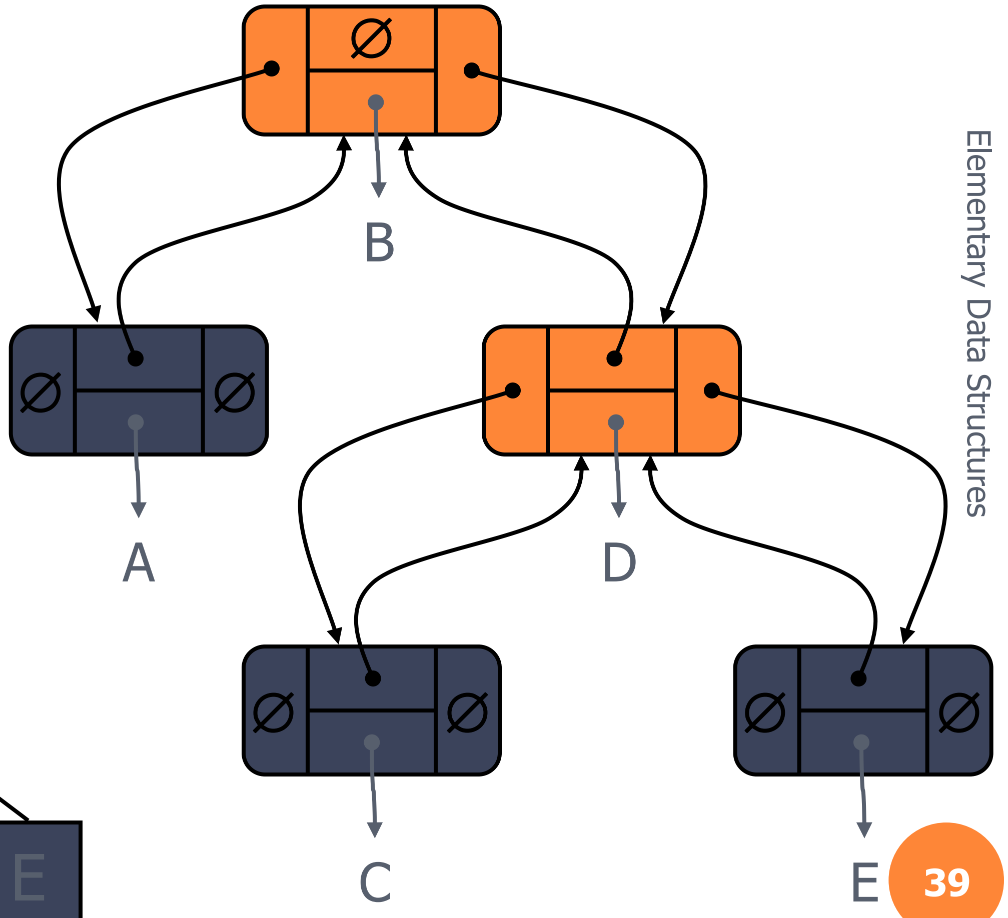




LINKED DATA STRUCTURE FOR BINARY TREES

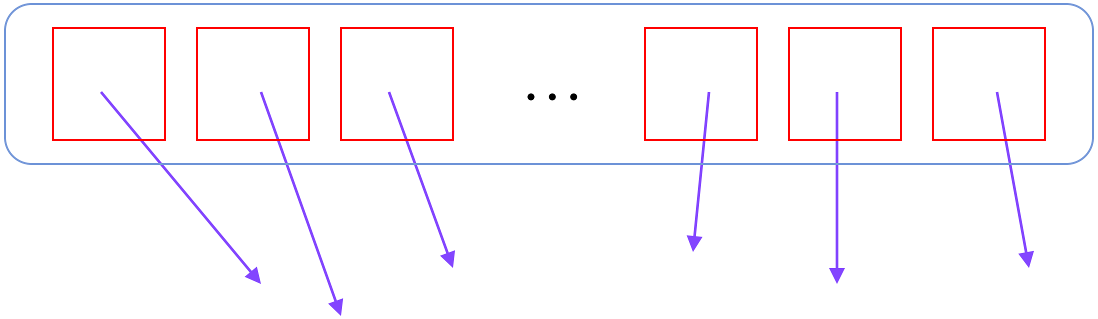
- A node is represented by an object storing
  - Element
  - Parent node
  - Left child node
  - Right child node

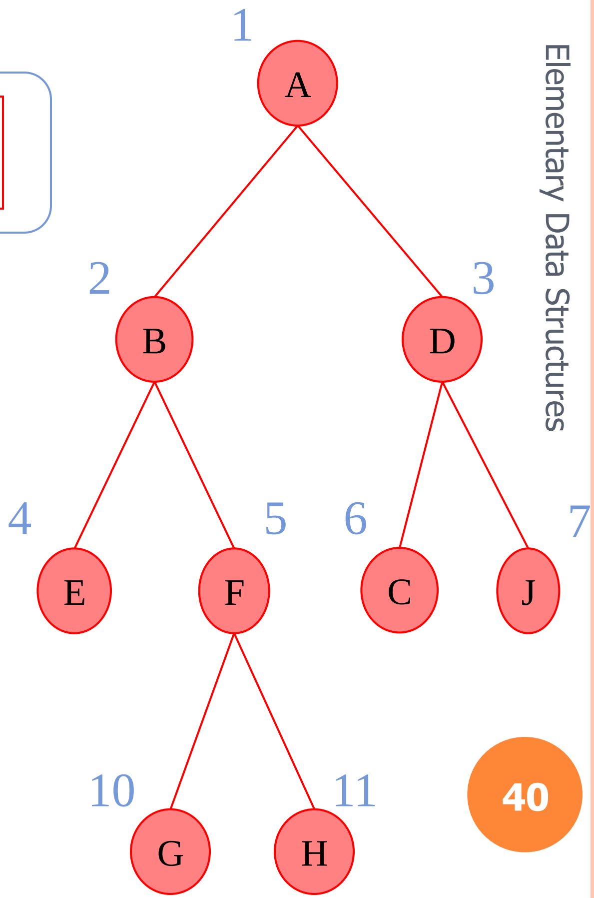




ARRAY-BASED REPRESENTATION OF BINARY TREES

- nodes are stored in an array





  - let rank(node) be defined as follows:
    - 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

Array-Based Representation of Binary Trees

- Space requirement

$$N = 2^{(n+1)/2}$$