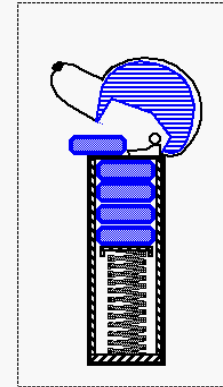


Elementary Data Structures

Stacks, Queues, & Lists
Amortized analysis
Trees



The Stack ADT

- ◆ The **Stack** ADT 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

Elementary Data Structures

3

Array-based Stack

- ◆ 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 $S[t + 1]$

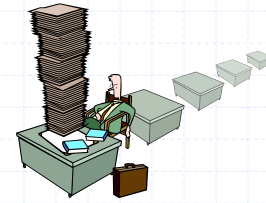
Algorithm *push(o)*:
if $t = S.length - 1$ **then**
 throw *FullStackException*
else
 $t \leftarrow t + 1$
 $S[t] \leftarrow o$



Elementary Data Structures

4

Growable Array-based Stack



- ◆ 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
 $A[i] \leftarrow S[i]$
 $S \leftarrow A$
 $t \leftarrow t + 1$
 $S[t] \leftarrow o$

Elementary Data Structures

5

The Queue ADT



- ◆ 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**

Elementary Data Structures

6

Applications of Queues



◆ Direct applications

- Waiting lines
- Access to shared resources (e.g., printer)
- Multiprogramming

◆ Indirect applications

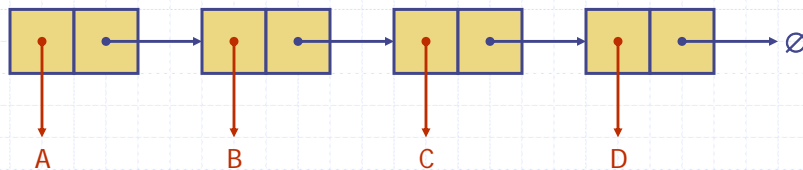
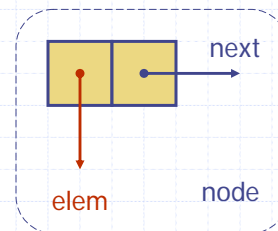
- Auxiliary data structure for algorithms
- Component of other data structures

Singly Linked List

◆ A singly linked list is a concrete data structure consisting of a sequence of nodes

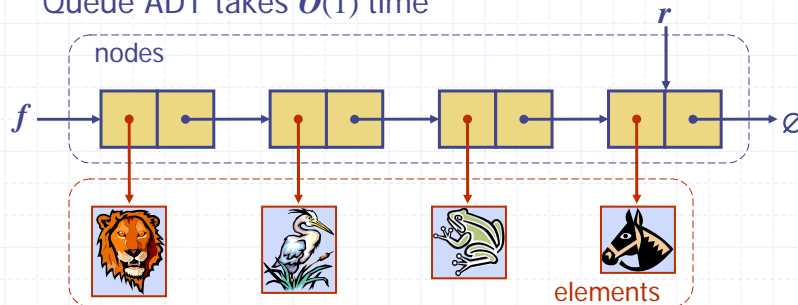
◆ Each node stores

- element
- link to the next node



Queue with a Singly Linked List

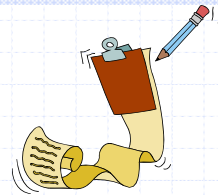
- ◆ We can implement a queue with a singly linked list
 - The front element is stored at the first node
 - The rear element is stored at the last node
- ◆ The space used is $O(n)$ and each operation of the Queue ADT takes $O(1)$ time



Elementary Data Structures

9

List ADT



- ◆ 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:

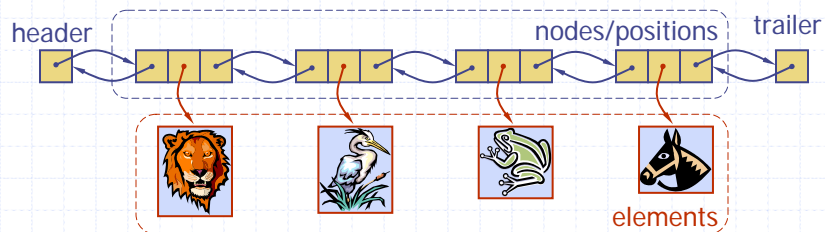
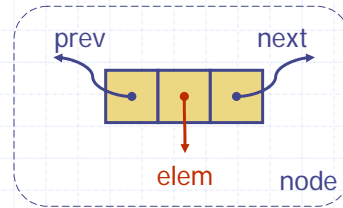
- **replaceElement**(p, o), **swapElements**(p, q)
- **insertBefore**(p, o), **insertAfter**(p, o),
- **insertFirst**(o), **insertLast**(o)
- **remove**(p)

Elementary Data Structures

10

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

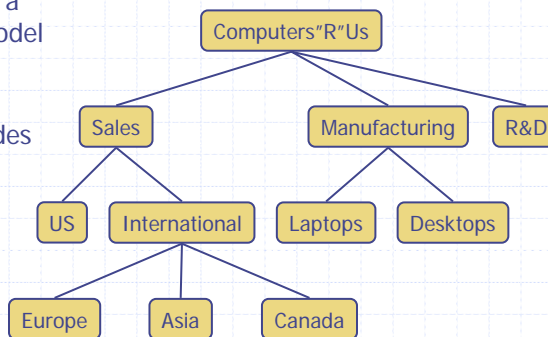


Elementary Data Structures

11

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
 - Programming environments



Elementary Data Structures

12

Tree ADT



- ◆ We use positions to abstract nodes
- ◆ 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

Elementary Data Structures

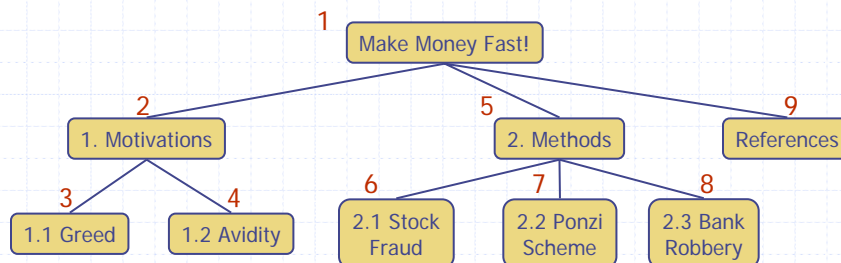
13

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)



Elementary Data Structures

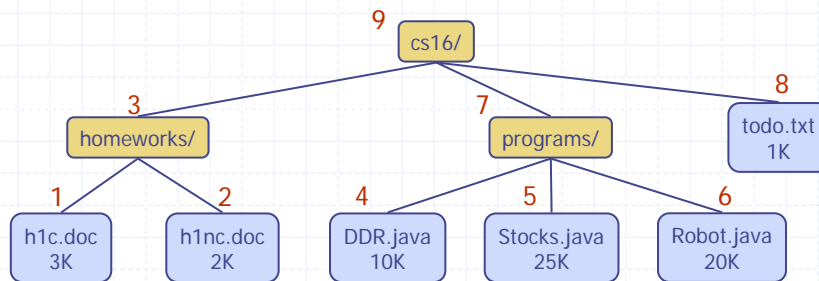
14

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)



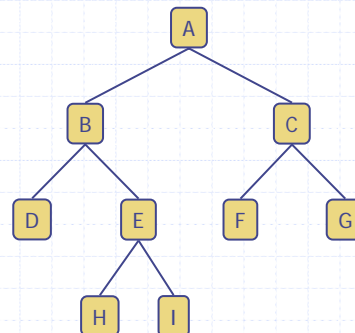
Elementary Data Structures

15

Binary Trees

- ◆ A binary tree is a tree with the following properties:
 - Each internal node has 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

- ◆ Applications:
 - arithmetic expressions
 - decision processes
 - searching

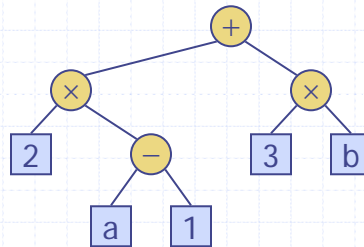


Elementary Data Structures

16

Arithmetic Expression Tree

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

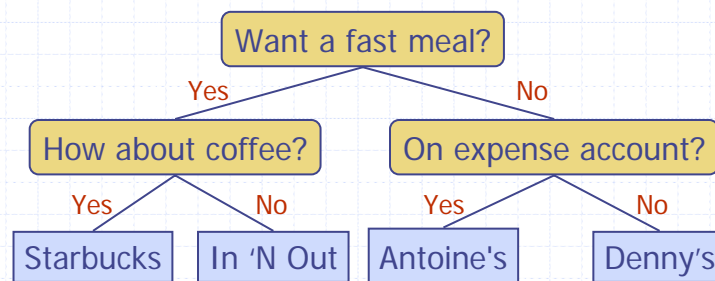


Elementary Data Structures

17

Decision Tree

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



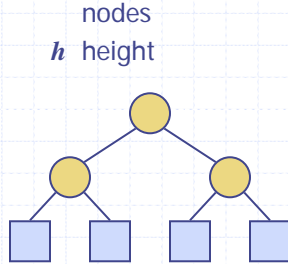
Elementary Data Structures

18

Properties of Binary Trees

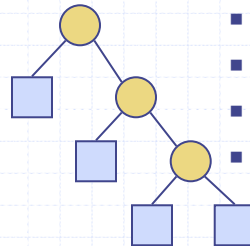
Notation

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



Properties:

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



Elementary Data Structures

19

Inorder Traversal

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

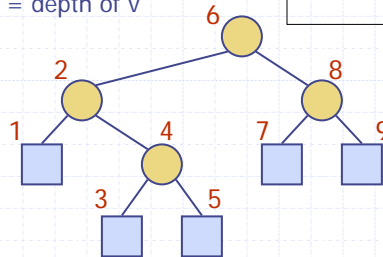
- ◆ Application: draw a binary tree

- $x(v)$ = inorder rank of v
- $y(v)$ = depth of v

Algorithm *inOrder*(v)

```

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

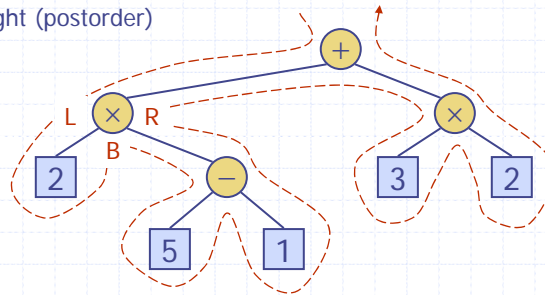


Elementary Data Structures

20

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)

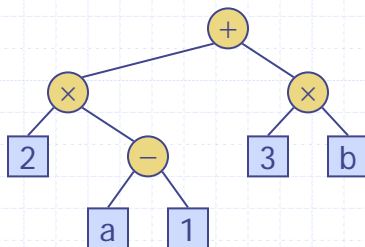


Elementary Data Structures

21

Printing 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 isInternal (v)
    print "("
    inOrder (leftChild (v))
    print (v.element ())
if isInternal (v)
    inOrder (rightChild (v))
    print ")"
    
```

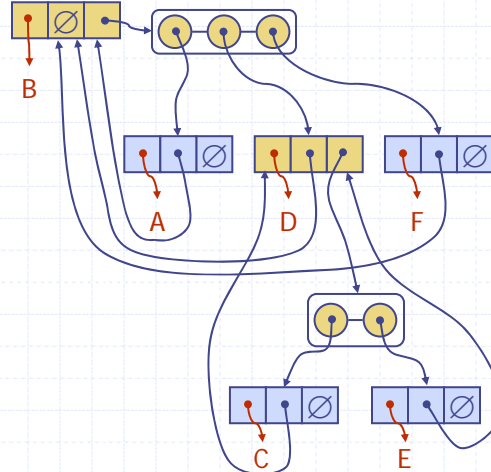
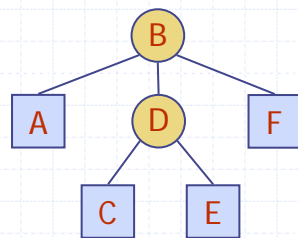
$((2 \times (a - 1)) + (3 \times b))$

Elementary Data Structures

22

Linked Data Structure for Representing Trees

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

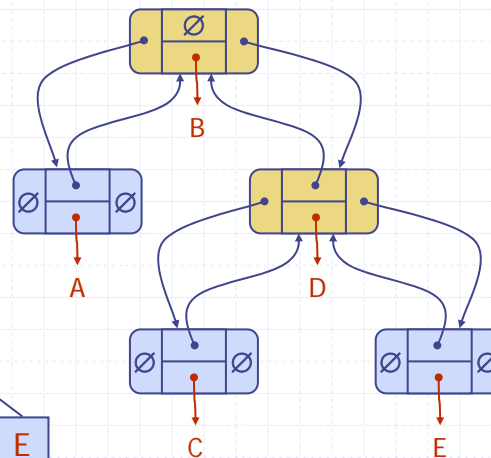
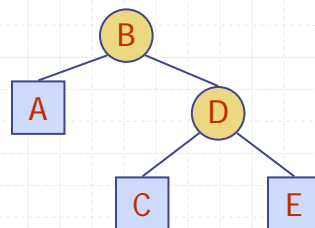


Elementary Data Structures

23

Linked Data 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

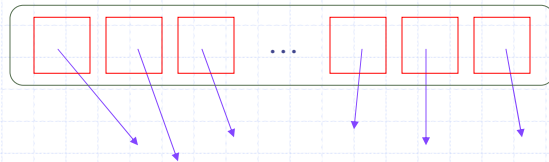


Elementary Data Structures

24

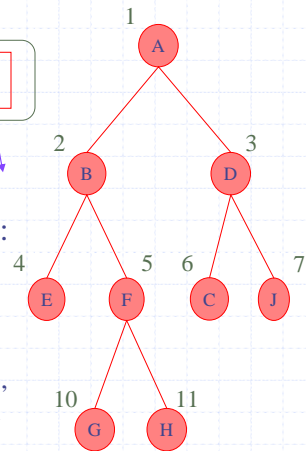
Array-Based Representation of Binary Trees

◆ nodes are stored in an array



■ let $\text{rank}(\text{node})$ be defined as follows:

- $\text{rank}(\text{root}) = 1$
- if node is the left child of $\text{parent}(\text{node})$,
 $\text{rank}(\text{node}) = 2 * \text{rank}(\text{parent}(\text{node}))$
- if node is the right child of $\text{parent}(\text{node})$,
 $\text{rank}(\text{node}) = 2 * \text{rank}(\text{parent}(\text{node})) + 1$



Elementary Data Structures

25