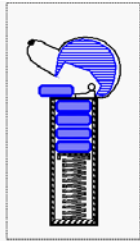


Elementary Data Structures

Stacks, Queues, Vectors, Lists & Sequences
Trees



The Stack ADT (§2.1.1)



- ◆ 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

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
    
```



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

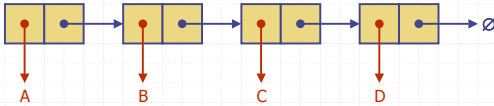
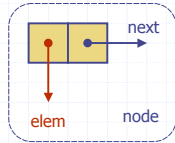
Applications of Queues



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

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

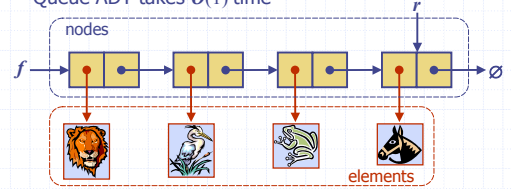


Elementary Data Structures v1.2

7

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 v1.2

8

The Vector ADT

- ◆ The **Vector** ADT extends the notion of array by storing a sequence of arbitrary objects
- ◆ An element can be accessed, inserted or removed by specifying its rank (number of elements preceding it)
- ◆ An exception is thrown if an incorrect rank is specified (e.g., a negative rank)
- ◆ Main vector operations:
 - object **elemAtRank**(integer r): returns the element at rank r without removing it
 - object **replaceAtRank**(integer r , object o): replace the element at rank with o and return the old element
 - **insertAtRank**(integer r , object o): insert a new element o to have rank r
 - object **removeAtRank**(integer r): removes and returns the element at rank r
- ◆ Additional operations **size()** and **isEmpty()**

Elementary Data Structures v1.2

9

Applications of Vectors

- ◆ Direct applications
 - Sorted collection of objects (elementary database)
- ◆ Indirect applications
 - Auxiliary data structure for algorithms
 - Component of other data structures

Elementary Data Structures v1.2

10

Array-based Vector

- ◆ Use an array V of size N
- ◆ A variable n keeps track of the size of the vector (number of elements stored)
- ◆ Operation **elemAtRank**(r) is implemented in $O(1)$ time by returning $V[r]$

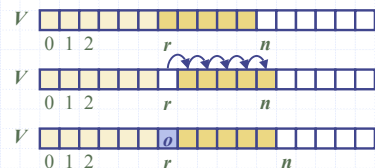


Elementary Data Structures v1.2

11

Insertion

- ◆ In operation **insertAtRank**(r, o), we need to make room for the new element by shifting forward the $n - r$ elements $V[r], \dots, V[n - 1]$
- ◆ In the worst case ($r = 0$), this takes $O(n)$ time

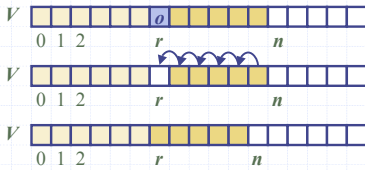


Elementary Data Structures v1.2

12

Deletion

- ◆ In operation **removeAtRank**(r), we need to fill the hole left by the removed element by shifting backward the $n - r - 1$ elements $V[r + 1], \dots, V[n - 1]$
- ◆ In the worst case ($r = 0$), this takes $O(n)$ time



Elementary Data Structures v1.2

13

Performance

- ◆ In the array based implementation of a Vector
 - The space used by the data structure is $O(N)$
 - **size**, **isEmpty**, **elemAtRank** and **replaceAtRank** run in $O(1)$ time
 - **insertAtRank** and **removeAtRank** run in $O(n)$ time
- ◆ If we use the array in a circular fashion, **insertAtRank**(0) and **removeAtRank**(0) run in $O(1)$ time
- ◆ In an **insertAtRank** operation, when the array is full, instead of throwing an exception, we can replace the array with a larger one

Elementary Data Structures v1.2

14

Position ADT

- ◆ The **Position** ADT models the notion of place within a data structure where a single object is stored
- ◆ It gives a unified view of diverse ways of storing data, such as
 - a cell of an array
 - a node of a linked list
- ◆ Just one method:
 - object **element**(): returns the element stored at the position

Elementary Data Structures v1.2

15

List ADT (§2.2.2)

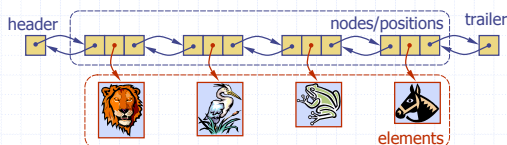
- ◆ The **List** ADT models a sequence of positions storing arbitrary objects
- ◆ It establishes a before/after relation between positions
- ◆ Generic methods:
 - **size**(), **isEmpty**()
- ◆ Query methods:
 - **isFirst**(p), **isLast**(p)
- ◆ Accessor methods:
 - **first**(), **last**()
 - **before**(p), **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 v1.2

16

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

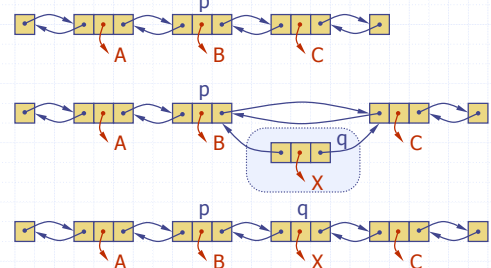


Elementary Data Structures v1.2

17

Insertion

- ◆ We visualize operation **insertAfter**(p, X), which returns position q

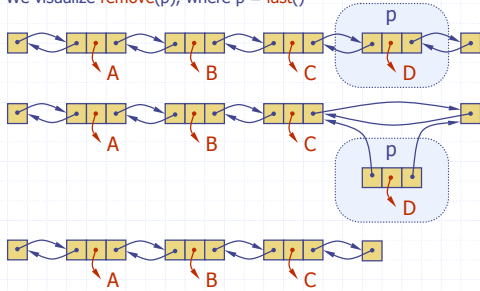


Elementary Data Structures v1.2

18

Deletion

- ◆ We visualize `remove(p)`, where $p = \text{last}()$



Performance

- ◆ In the implementation of the List ADT by means of 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 operations of the List ADT run in $O(1)$ time
 - Operation `element()` of the Position ADT runs in $O(1)$ time

Sequence ADT

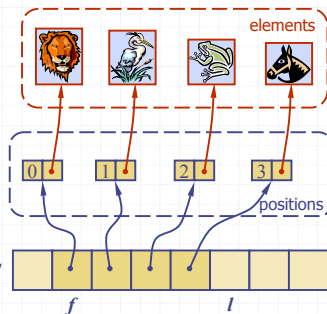
- ◆ The **Sequence** ADT is the union of the Vector and List ADTs
- ◆ Elements accessed by
 - Rank, or
 - Position
- ◆ Generic methods:
 - `size()`, `isEmpty()`
- ◆ Vector-based methods:
 - `elemAtRank(r)`, `replaceAtRank(r, o)`, `insertAtRank(r, o)`, `removeAtRank(r)`
- ◆ List-based methods:
 - `first()`, `last()`, `before(p)`, `after(p)`, `replaceElement(p, o)`, `swapElements(p, q)`, `insertBefore(p, o)`, `insertAfter(p, o)`, `insertFirst(o)`, `insertLast(o)`, `remove(p)`
- ◆ Bridge methods:
 - `atRank(r)`, `rankOf(p)`

Applications of Sequences

- ◆ The Sequence ADT is a basic, general-purpose, data structure for storing an ordered collection of elements
- ◆ Direct applications:
 - Generic replacement for stack, queue, vector, or list
 - small database (e.g., address book)
- ◆ Indirect applications:
 - Building block of more complex data structures

Array-based Implementation

- ◆ We use a circular array storing positions
- ◆ A position object stores:
 - Element
 - Rank
- ◆ Indices f and l keep track of first and last positions



Sequence Implementations

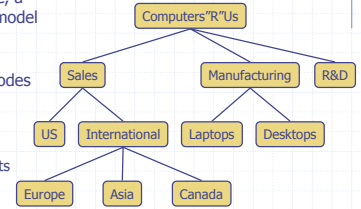
| Operation | Array | List |
|---|-------|------|
| <code>size</code> , <code>isEmpty</code> | 1 | 1 |
| <code>atRank</code> , <code>rankOf</code> , <code>elemAtRank</code> | 1 | n |
| <code>first</code> , <code>last</code> , <code>before</code> , <code>after</code> | 1 | 1 |
| <code>replaceElement</code> , <code>swapElements</code> | 1 | 1 |
| <code>replaceAtRank</code> | 1 | n |
| <code>insertAtRank</code> , <code>removeAtRank</code> | n | n |
| <code>insertFirst</code> , <code>insertLast</code> | 1 | 1 |
| <code>insertAfter</code> , <code>insertBefore</code> | n | 1 |
| <code>remove</code> | n | 1 |

Iterators

- An iterator abstracts the process of scanning through a collection of elements
- Methods of the ObjectIterator ADT:
 - object `object()`
 - boolean `hasNext()`
 - object `nextObject()`
 - reset()
- Extends the concept of Position by adding a traversal capability
- Implementation with an array or singly linked list
- An iterator is typically associated with another data structure
- We can augment the Stack, Queue, Vector, List and Sequence ADTs with method:
 - ObjectIterator `elements()`
- Two notions of iterator:
 - snapshot: freezes the contents of the data structure at a given time
 - dynamic: follows changes to the data structure

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 ADT (§2.3.1)

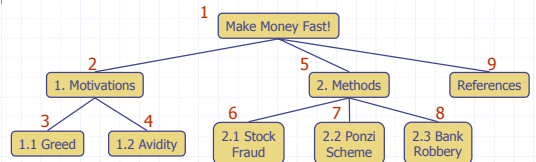
- We use positions to abstract nodes
- Generic methods:
 - integer `size()`
 - boolean `isEmpty()`
 - objectIterator `elements()`
 - positionIterator `positions()`
- Accessor methods:
 - position `root()`
 - position `parent(p)`
 - positionIterator `children(p)`
- Query methods:
 - boolean `isInternal(p)`
 - 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



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

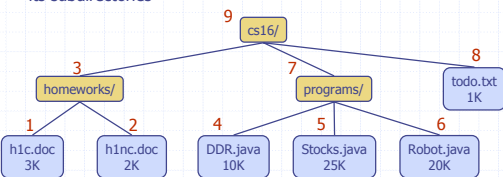
Algorithm *preOrder*(v)
visit(v)
for each child w of v
preorder (w)



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

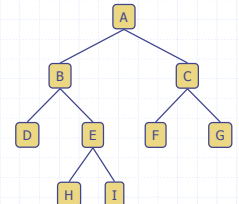
Algorithm *postOrder*(v)
for each child w of v
postOrder (w)
visit(v)



Binary Trees (§2.3.3)

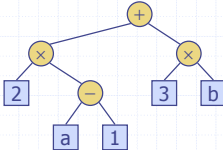
- A binary tree is a tree where:
 - Each internal node has at most two children
- A **proper** binary tree is a binary tree where:
 - each internal node has exactly two children
 - The children are an ordered pair, denoted left child and right child.
- Alternative recursive definition: a (proper) binary 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 (proper) 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 $(2 \times (a - 1) + (3 \times b))$

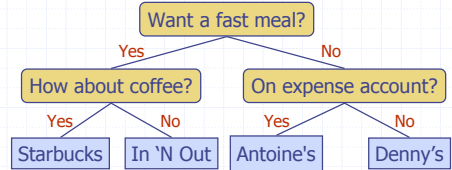


Elementary Data Structures v1.2

31

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 v1.2

32

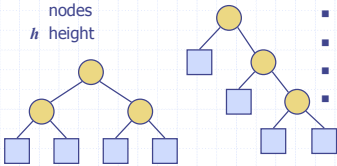
Properties of (Proper) Binary Trees

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 v1.2

33

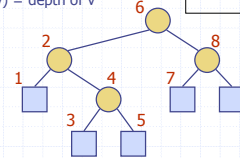
Inorder Traversal

- 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))
    
```

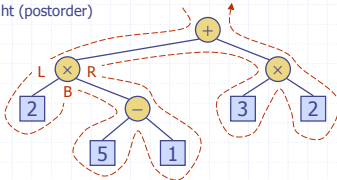


Elementary Data Structures v1.2

34

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 v1.2

35

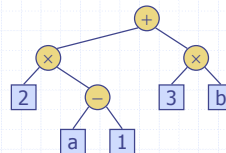
Printing Arithmetic Expressions

- 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(")")
    
```



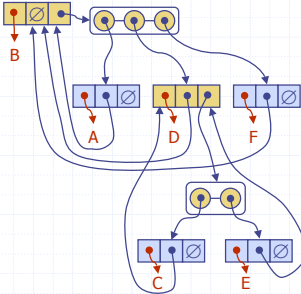
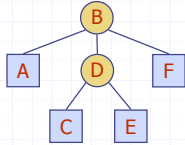
$((2 \times (a - 1)) + (3 \times b))$

Elementary Data Structures v1.2

36

Linked Data Structure for Representing Trees (§2.3.4)

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

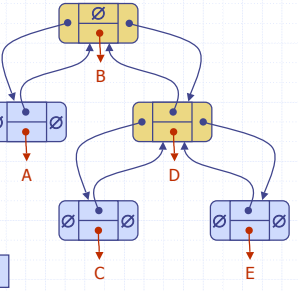
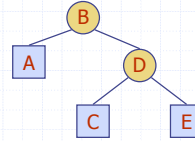


Elementary Data Structures v1.2

37

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 v1.2

38

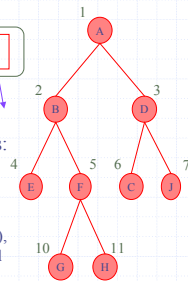
Array-Based Representation of Binary Trees

- ◆ 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 v1.2

39