Other notations

• big-Omega (lower bound)
  • \( f(n) \) is \( \Omega(g(n)) \) if there are constants \( c > 0 \) and \( n_0 \geq 1 \) such that \( f(n) \geq cg(n) \) for \( n \geq n_0 \)

• big-Theta (tight bound)
  • \( f(n) \) is \( \Theta(g(n)) \) if there are constants \( c > 0 \), \( c' > 0 \), and \( n_0 \geq 1 \) such that \( cg(n) \leq f(n) \leq c'g(n) \) for \( n \geq n_0 \)

• little-o (strict upper bound)
  • \( f(n) \) is \( o(g(n)) \) if for any constant \( c > 0 \) there is a constant \( n_0 \geq 0 \) such that \( f(n) \leq cg(n) \) for \( n \geq n_0 \)

• little-omega (strict lower bound)
  • \( f(n) \) is \( \omega(g(n)) \) if for any constant \( c > 0 \) there is a constant \( n_0 \geq 0 \) such that \( f(n) \geq cg(n) \) for \( n \geq n_0 \)
Lecture 2: Overview

- Stacks
- Queues
- Vectors
- Lists
Stacks

- Common data structure
- Works like a stack of trays in a cafeteria
- Two principal operations:
  - `Push(object)` – places the object on the top of the stack
  - `object pop()` – removes the object from the top of the stack and returns it
- Auxiliary operations:
  - `object top()` – returns the object from the top of the stack
  - `integer size()` – returns the size of the stack
  - `boolean isEmpty()` – returns whether the stack is empty
- Implement using dynamically allocated data structures
Stacks

push(3)
Stacks

push(3)
Stacks

push(2)
Stacks

push(2)
Stacks

pop()
Stacks

pop()
returns 3
Errors

- It is illegal to execute
  - Pop on an empty stack
  - Top on an empty stack
- Java implementations should throw an exception in these cases
Possible Implementation Strategies

• Array based
  • Store pointers in an array
  • Has either fixed maximum size or requires
    allocating a new array if the original array
    fills up

• Linked data structure
  • Create an wrapper object for each item in
    the stack
  • This wrapper object points to the stored
    object and to the previous object
Queues

- Common data structure
- Works like the line at the grocery store
- Two operations:
  - Enqueue(object) – places the value or object at the end of the line
  - object Dequeue() – removes the value or object from the beginning of the line
- Auxiliary operations:
  - object front() – returns the object at the front of the line without removing it
  - integer size() – returns the number of objects in the queue
  - boolean isEmpty() – indicates whether the queue is empty
Queues

1 3 5 2
Queues

Enqueue(9)

1 3 5 2
Queues

Enqueue(9)

\[
\begin{array}{cccccc}
1 & 3 & 5 & 2 & 9 \\
\end{array}
\]
Queues

Dequeue()
Queues

Dequeue()
returns 1

3 5 2 9
Errors

• Executing dequeue of front on an empty queue
Possible Implementation Strategies

• Array based
  • Store pointers in an array
  • Use in a circular fashion
  • Has either fixed maximum size or requires allocating a new array if the original array fills up

• Linked data structure
  • Create an wrapper object for each item in the stack
  • This wrapper object points to the stored object and to the previous object
Vector

• Extends the abstraction of an array
• Operations:
  • object elemAtRank(integer r) - returns the element at rank r
  • object replaceAtRank(integer r, object o) - replace element at rank r with o and return the old element
  • insertAtRank(integer r, object o) - insert a new element o to have rank r (moves all object with rank >r up one rank)
  • object removeAtRank(integer r) - removes and returns the element at rank r
  • size() / isEmpty()
Implementation

• Typically use an array
• If the array fills up, allocate a new larger array and copy the contents
Insertion

new value

1 3 5 2 9
List

• Methods:
  • isFirst(p), isLast(p) - returns whether object is first or last
  • first(), last() - returns the object that is last or first
  • before(p), after(p) - returns the object before or after p
  • replaceElement(p, o) - replace p with o
  • swapElements(p, q) - swap p and q
  • insertBefore(p, o), insertAfter(p, o) - insert p before or after o
  • insertFirst(o), insertLast(o) - insert o at the beginning or end
  • remove(p) - remove p
Doubly Linked List

- Standard heap structure to implement the list API
- List consist of a chain of nodes
- Each node points to:
  - the previous node
  - the next node
  - the stored object/value
Doubly Linked List

next pointers

prev pointers