Lecture Notes

1. Introduction
   - Collection ADTs (lists, sets) were *unstructured*: just held “stuff”
   - Sorting *imposed* structure (ordering) as part of the collection’s *state*
   - Other ADTs: structure is part of the collection’s *behavior*
   - Difference is subtle but vital

2. Stacks
   a. Definition: A *stack* is an abstract data type that holds elements in a LIFO manner
      - LIFO: last-in, first-out
      - Restricted ADT (access limited only through core functionality)
   b. Core functionality
      - Push – Places elements on *top* of the stack
      - Pop – Removes elements from the *top* of the stack (and returns them)
   c. Secondary functionality
      - Peek
      - isEmpty
      - size
      - find (but not remove, otherwise LIFO would be broken)
      - contains
      - clear
      - Other considerations:
        - Initialization of the stack
        - Behavior of Push when the stack is “full”
        - Behavior of Pop when the stack is empty
   d. Implementation
      - Use List ADT
        - Array-based: appropriate for bounded (capacity constrained) stacks
        - Linked-list: best for unbounded stacks (constant push/pop operations)
      - Support via composition: the Stack owns a List
      - Push: insert at head/insert at index 0
      - Pop: remove from head/remove from index 0
      - Secondary functionality in terms of List ADT interface
      - Even if underlying list supports arbitrary size, possible to impose a size limitation in the Stack ADT
      - Demonstration: unl.cse.stacks.MyStack
e. Applications
   • Hardware (system) stack
   • Program stacks (method calls)
   • Unfolding recursion
   • Compilers: evaluating expressions, Shunting Yard Algorithm

f. In Practice: Java Collections
   i. java.util.Stack<E>
      • implements java.util.List interface (actually a subclass of Vector, an array-based list); as a consequence: LIFO can be broken
      • Allows null elements
      • E peek() – returns but does not remove the top element, may throw an EmptyStackException
      • E pop() – removes and returns the top object, may throw an EmptyStackException
      • E push(E item) – pushes the given item onto the top of the stack and returns it
      • int search(Object o) – returns distance of first instance of the given object (top element is distance 1), -1 if does not exist

3. Queues
   a. Definition: A queue is an abstract data type which holds elements in a FIFO manner
      • First-In First-Out
      • A line: no line jumping!
   b. Core Functionality
      • Enqueue (aka push, offer)
      • Dequeue (aka pop, remove, poll)
   c. Secondary Functionality
      • Peek
      • isEmpty
      • size
      • find
      • contains
      • clear
      • remove (tired of waiting, leave the line)
      • Other Considerations
   d. Implementation
      • Use of a List ADT
      • Composition: Queue ADT owns a List
      • Enqueue: insert at tail/insert at index [size]
      • Dequeue: remove at head/remove at index 0
   e. Special Queues & Applications
      i. Deque (“deck”): double-ended queue
Can insert at both ends but remove from one
Can remove from both ends but insert at one
Applications
  o  Stacks/Queues with a maximum capacity
  o  Job scheduling (high/low priority at each end)
  o  Application’s list of “undo” operations
Java (6+): `java.util.Deque<E>` (interface)
  o  `ArrayDeque`
  o  `LinkedList`
  o  `Queue (addLast(e), offerLast(e), removeFirst, pollFirst, getFirst, peekFirst)` or
  o  `Stack (addFirst(e), removeFirst(), peekFirst())`

ii. Priority Queue
  •  Queue that is not FIFO but based on some sort of priority
  •  Enqueue inserts an element in the queue according to its priority (not necessarily at the end of the line)
  •  Dequeue returns the highest priority element
  •  Line-jumping or “VIP” service
  •  Implementation: ordered list!
  •  Can be used to sort!
  •  Demo: `MyPriorityQueue<T>`
    o  Priority provided upon call to enqueue
    o  Priority maintained through a Map data structure
    o  Observe: implementation leads to a Java memory leak!

iii. Buffers

iv. Consumer-Producer patterns
  •  Resources may be limited; operations expensive, but not required to operate synchronously (asynchronous operation)
  •  Enqueueing a request and processing it later means that control flow can continue
  •  Example: basic operations to persist to a database, email a message
  •  Producer/Consumers: requests/responses, connections pooling, etc.

f. In Practice: Java Collections
  i. Queue<E>
    •  Interface (implementation: `LinkedList` among others)
    •  `boolean offer(E o)` – enqueues the given element (true if successful, false otherwise)
    •  `E poll()` – dequesues and returns the next element, returns `null` for an empty queue
    •  `E remove()` – dequesues and returns the next element, throws a `NoSuchElementException` when empty
• E peek() – retrieves but does not remove the next element, returns null if empty
• E element() – Retrieves but does not remove the head of the queue, throws a NoSuchElementException when empty

ii. BlockingQueue<E>
• Interface (implementations: LinkedBlockingQueue, PriorityBlockingQueue)
• Thread safe
• Producer consumer (blocks if poll is run on an empty queue)
• Applicable if resources are scarce (say network or database connection pools)

iii. PriorityQueue<E>
• A class
• Heap implementation (more on heaps later)
• Makes use of a comparator for priority