Lecture 1: Overview

- Read Chapters 1-3, 10
- Stacks
- Queues
- Vectors
- Lists
- Trees
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)

3
push(2)
Stacks

pop()

2

3
Stacks

pop()
returns 2
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

Enqueue(9)

1 3 5 2
Queues

Enqueue(9)

1  3  5  2  9
Queues

Dequeue()
Queues

Dequeue()
returns 1

| 3 | 5 | 2 | 9 |
Errors

- Executing dequeue or 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
Binary Search Trees

- Store data in a tree
- Each node can have up to two child nodes
  - one left child
  - one right child
Binary Search Trees

- All left descendants have smaller values
- All right descendants have larger values
- Nodes w/o descendants are called leaf node
Binary Search Trees

1. To find a value in a tree start at the root
2. If the value is less than the current node look at its left child
3. If the value is greater, look at its right child
Binary Search Trees

1. To find a value in a tree start at the root
2. If the value is less than the current node look at its left child
3. If the value is greater, look at its right child
4. Repeat 2 until either we find the node or we reach the bottom of the tree
5. If we reach the bottom of the tree, it doesn’t contain the value
Searching Tree

- Search for 5
- Start at root

```
4
/   \
1     6
|     |
5     7
```
Searching Tree

- Since 5 > 4 look at right child
• Since 5<6 look at left child
Searching Tree

- Found value!!!
Inserting a Node into a Tree

- Find where to place node:
  Use same algorithm as search algorithm
- Two possible outcomes:
  - Find value – then simply do a replacement
  - End up at node where we can’t search any further:
    - If less than leaf node add to left side
    - If greater than leaf node add to right side
Time Complexity

- Depends on the depth of the tree
- If the tree is balanced, depth $\propto \log(N)$
- If tree isn’t balanced, depth can grow as $N$