Data Structures

Giri Narasimhan
Office: ECS 254A
Phone: x-3748
giri@cs.fiu.edu

Linear Data Structures

Java's List interface

- Get(idx)
- Set(idx, value)
- Add(idx, value)
- Remove(idx)
- listIterator(pos)

(My)ArrayList Implementation

- See Figures 3.15 and 3.16 from Weiss text
- Maintains the following data
  - list of items in an array called theItems[]
  - Array capacity (length of above array)
  - Current size called theSize
- Allows the following operations
  - Change in capacity (capacity doubled if array fills up)
    - No change upon removal
  - Implementation of get(idx) and set(idx,x)
  - Implementation of size(), isEmpty(), clear()
  - Implementation of Iterator interface
    - Index called current
    - next(), hasNext(), remove()

add and remove in ArrayList

- Both involve moving items
- Operation add(idx,x) involves moving all items from position idx onward to move in order to make space for x
- Operation remove(idx) involves moving all items from position idx+1 to close the gap created by the removal
- Study carefully how ArrayIterator is implemented

Time Complexity of List Operations

	ArrayList	LinkedList
get(idx)	O(1)	O(N)
set(idx, x)	O(1)	O(N)
add(idx,x)	O(N)	O(N)
remove(idx)	O(N)	O(N)
addBefore(p,x)		O(1)
Remove(p)		O(1)

Stack ADT

```
public interface Stack { // Stack class ADT
  // Reinitialize the stack.
  public void clear();

  // Push "it" onto the top of the stack
  public push(Object it);

  // Remove and return the element at the top of the stack
  public Object pop();

  // Return a copy of the top element
  public Object peek();

  // Return the number of elements in the stack
  public int length();
}
```

Stack Class using Arrays

```
class AStack implements Stack {

  private Object stackArray[];    // Array holding stack
  private static final int defaultSize = 10;
  private int maxSize;            // Maximum size of stack
  private int top;                // First free position at top

  // Implementation of all the operations

}
```

ArrayStack Implementation

```
public boolean push(Object it) {
    if (top >= maxSize) return false;
    stackArray[top++] = it;
    return true;
  }

public Object pop() {
    if (top == 0) return null;
    return stackArray[--top];
  }

public Object peek() {
    if (top == 0) return null;
    return stackArray[top-1];
  }
```

Stack Class using Linked Lists

```
class LinkedStack implements Stack {

  private Node top;              // Pointer to first element
  private int size;              // Number of elements

  // Implementation of all the operations

}
```

LinkedStack Implementation

```
public boolean push(Object it) {
    top = new Node(it, top);
    size++;
    return true;
  }

public Object pop() {
    if (top == null) return null;
    Object it = top.data;
    top = top.next;
    size--;
    return it;
  }

  public Object peek() {
    if (top == null) return null;
    return top.data;
  }
```

	ArrayStack	LinkedStack
push(x)	O(1)	O(1)
pop()	O(1)	O(1)
peek()	O(1)	O(1)

Queue ADT

```
public interface Queue { // Queue class ADT
  // Reinitialize queue
  public void clear();

  // Put element on rear
  public boolean enqueue(Object it);

  // Remove and return element from front
  public Object dequeue();

  // Return front element
  public Object frontValue();

  // Return queue size
  public int length();
}
```

Queues using Arrays

```
class AQueue implements Queue {

  private Object queueArray[]; // Array holding elements
  private static final int defaultSize = 10;
  private int maxSize;        // Maximum size of queue
  private int front;          // Index of front element
  private int rear;           // Index of rear element

  // Implementation of all the operations

}
```

Why front and rear?

ArrayQueue Implementation

```
public boolean enqueue(Object it) {
    if (((rear+2) % maxSize) == front) return false;  // Full
    rear = (rear+1) % maxSize; // Circular increment
    queueArray[rear] = it;
    return true;
  }
public Object dequeue() {
    if(length() == 0) return null;
    Object it = queueArray[front];
    front = (front+1) % maxSize; // Circular increment
    return it;
  }
public int length() { return ((rear+maxSize) - front + 1) % maxSize; }
```

Need to distinguish between full and empty.
Hence array is declared to be of size N+1 while using only N spots.
Thus, full is checked by:
(rear+2) % maxSize) == front
Empty is checked by: length() == 0

Queue using Linked Lists

```
class LQueue implements Queue {

  private Link front; // Pointer to front queue node
  private Link rear;  // Pointer to rear queuenode
  private int size;   // Number of elements in queue

  // Implementations of all operations

}
```

LinkedQueue Implementation

```
public boolean enqueue(Object it) {
    rear.next = new Link(it, null);
    rear = rear.next;
    size++;
    return True;
}
public Object dequeue() {
    if (size == 0) return null;
    Object it = front.next.data; // retrieve the value
    front.next = front.next.next; // Advance front
    if (front.next() == null) rear = front; // Last element
    size--;
    return it; // Return element
  }
public Object frontValue() {
    if (size == 0) return null;
    return front.next.data;
  }
```

	ArrayQueue	LinkedQueue
enqueue(x)	O(1)	O(1)
dequeue()	O(1)	O(1)
frontValue()	O(1)	O(1)

Applications of Stacks

- Balancing Parentheses
  - Is this balanced?
    - (()((()()())()))
  - Push when you see an open parenthesis.
  - Pop when you see a closed one.
  - If stack empties at end of string, it is balanced.
  - Else, it is not.
- Postfix Expressions
  - How to evaluate this:
    - 4 2 * 5 + 7 + 1.5 *
  - Push when you see a number
  - Pop two when you see an operator, operate and push result

How to deal with (unary) negative signs such as 4 * (-5)?

Applications of Stacks … 2

- Infix to Postfix Conversion
  - How to convert this:
    - **Infix**: ((4 * 2) + 5 + 7) * 1.5 **to Postfix**: 4 2 * 5 + 7 + 1.5 *
  - **Infix**: 4 * 2
  - Idea: Push operators on a stack and print operands
  - **Infix**: 4 + 2 + 3
  - Idea: Before pushing operators, pop previous operator
  - **Infix**: 4 + 2 * 5 + 3
  - Idea: Before pushing operators, pop all operators of higher or equal precedence
  - **Infix**: 4 * (2 + 3) + 5
  - Idea: Push open parenthesis on stack. When closed parenthesis is encountered, pop stack until open parenthesis is popped

Applications of Queues

- Servers
  - Disk server, File Server, Print Server, …