CSCI 112
Thursday, July 22, 2010

Nodes

Node – a node has two parts

1. Data portion
2. Link portion

A list of nodes is a "**Linked List**."

StackTest.java

```
class StackTest{
	public static void main( String[] args ){
		Stack<Integer> s = new Stack<Integer>();

		s.push(10);
		s.push(20);
		s.push(30);
		s.push(40);
		s.push(50);

		while( s.isEmpty == false ){
			System.out.println( s.pop() );
		}
	}
}
```

Stack.java

```
class Stack<T>{
	private class Node{
		public T data;
		public Node next;
	}

	private Node head;
	private int size;

	public Stack(){
		size = 0;
		head = null;
	}
	public boolean isEmpty(){ return size == 0; }
	public int size(){ return size; }
	public void push(T data){
```

```
                Node myNode = new Node();
                myNode.data = data;
                myNode.next = head;
                head = myNode;
                size++;
        }
        public T pop(){
                if( isEmpty() ){
                        System.out.println( "Error: Just like Pringles, you can only pop until they're all
gone." );
                        System.exit(1);
                }
                size--;
                T myData = head.data;
                head = head.next;
                return myData;
        }
        public T peek(){
                if( isEmpty() ){
                        System.out.println( "Error: You're out of stuff to peek at." );
                        System.exit(1);
                }
                return head.data;
        }
}
```

Queues

While Stacks follow the **LIFO** model, Queues follow the **FIFO (First-In-First-Out)** model.

Two Basic Operations of a Queue

enqueue → add to the back of the list
dequeue → remove from the front of the list

Typically you will also have another method known as **front**, which looks at the first element.

Queue Implementation Algorithm (Pseudo-code):

```
class Queue<T>
   int MAX_SIZE = 100
   int front
   int back
   T[] elements

   Queue:
      front = 0
```

```
    back  = 0
    elements = Array[MAX_SIZE]

isEmpty:
    return front == back

isFull:
    return front == (back + 1)%MAX_SIZE

size:
    return (back - front + MAX_SIZE) % MAX_SIZE

front:
    if isEmpty:
        print "Error: Queue is empty."
        exit
    return elements[front]

enqueue(data):
    if isFull:
        print "Error: Queue is full."
        exit
    elements[back] = data
    back = (back + 1) % MAX_SIZE

dequeue:
    if isEmpty:
        print "Error: Queue is empty."
        exit
    temp = elements[front]
    front = (front+1) % MAX_SIZE
    return temp
```

Queue Implementation: Array

```
class Queue<T>{
        private final int MAX_SIZE = 100;
        private int front = 0;
        private int back = 0;
        private T[] element;

        public Queue{
                front = 0;
                back = 0;
                elements = (T[] new Object[MAX_SIZE];
        }
        public boolean isEmpty(){
                return front == back;
        }
```

```
	public boolean isFull(){
		return front == (back+1)%MAX_SIZE;
	}
	public int size(){
		return (back-front+MAX_SIZE)%MAX_SIZE);
	}
	public T front(){
		if(isEmpty()){
			System.out.println( "Error: empty" );
			System.exit(1);
		}
		return elements[front];
	}
	public void enqueue(T data){
		if(isFull()){
			System.out.println( "Error: full" );
			System.exit(1);
		}
		element[back] = data;
		back = (back+1)%MAX_SIZE;
	}
	public T dequeue(){
		if(isEmpty()){
			System.out.println( "Error: empty" );
			System.exit(1);
		}
		T temp = elements[front];
		front = (front+1)%MAX_SIZE;
		return temp;
	}
}
```

Priority Queue

Priority Queue – a cross between a queue and an insertion sort. When enqueue'ing data, you perform an insertion sort to decide where to put it.

Dijkstra's Algorithm – shortest distance solver, an example of a priority queue.

Node Insertion

Two Problems

- insert at beginning
- insert beyond the beginning

Two Pieces of Information to Insert

- The value to be inserted
- the position

Insert at Beginning

1. Make a new node to hold data.
2. Set the new node's next field to head
3. Set the head to the new note.

Insert Beyond the Beginning

1. Make a new node to hold the data.
2. Traverse the list until you reach the node prior to the insertion point.
   - thisNode → node prior to insertion point
   - myNode → node to be inserted
3. thisNode's next field should be copied to myNode's next field
4. thisNode's next field should now point to myNode

Deleting a Node/Node Removal

Two Problems

- delete from beginning
- delete beyond beginning

One Piece of Information

- position to delete

Delete from Beginning

1. Store head.data field in temporary
2. head should point to head's next field

Delete Beyond the Beginning

1. Traverse the list until you reach the node prior to the deletion point.
   - thisNode → node prior to deletion point
2. Save thisNode.next.data in a temporary.
3. ThisNote.next = thisNode.next.next