Introduction to Algorithms and Data Structures

Lecture 11: Stacking and Standing
On Line - An Introduction to Stacks
and Queues

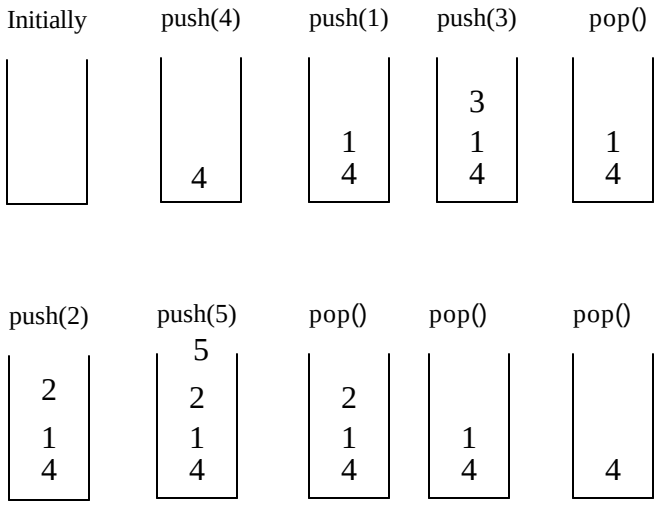
What is a Stack?

- A ***stack*** is a collection of data items where all data is inserted and removed from the same end - the ***top***.
- A stack is considered a last-in, first-out data structure because the newest data is removed first and the oldest data is removed last.

Stack Operations

- There are three basic operations on a stack:
  - ***empty*** ***true*** if the stack is empty; ***false*** if the stack is non-empty.
  - ***push*** - inserts a data item on the top of the stack.
  - ***pop*** - removes a data item from the top of the stack.

A Stack In Action



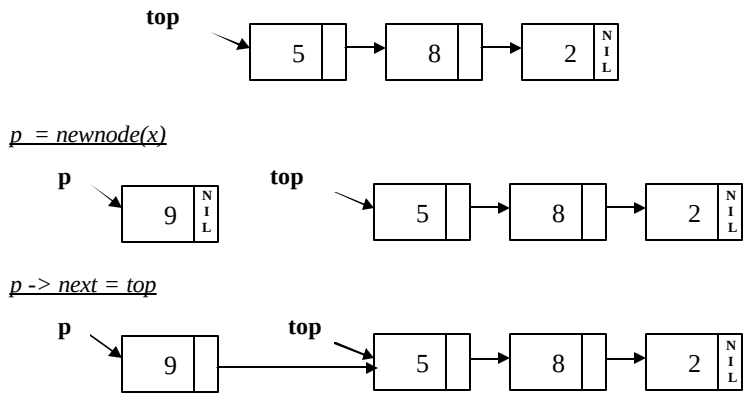
The **`Stack`** class

```
public class Stack  {
  private Node top;

  // newNode() - Creates a new node with the
  //             parameter x
  private Node newNode(int x)  {
    Node p = new Node();
    p.setData(x);
    p.setNext(null);
    return p;
  }

  // Stack() - A default constructor for
  //           stacks
  public Stack()  {
    top = null;
  }
```

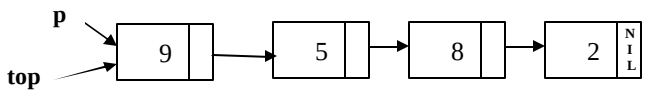
```
  // Stack() - A for stacks where x is the
  //    first item pushed on the stack
  public Stack(int x)  {
    top = newNode(x);
  }

  // empty() - True is the stack is empty
  //           (null pointer)
  //           False is the stack is non-empty
  //           (non-null pointer)
  public boolean empty()  {
    if (top == null)
      return true;
    else
      return false;
  }
```

```
// push() - Insert an item on the top of the
//          stack
public void push(int x)  {
  Node p = newNode(x);

  // Link it to the current top of the stack
  p.setNext(top);

  // Readjust the stack to point HERE
  top = p;
}
```

Tracing **`push()`**



top = p

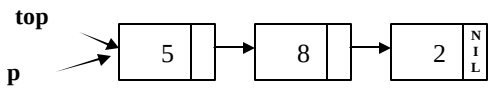



```
// pop() - Remove an item from the top of the
//   stack
public int pop()  {
  Node p;
  int x;

  if (empty())  {
    System.err.println
        ("Popping an empty stack");
    System.exit(0);
  }
```

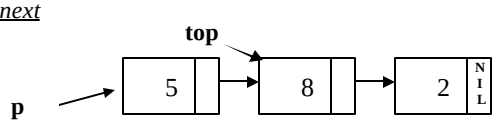
```
    // Set p pointing to the stack
    p = top;

    // Take the data from the node
    x = p.getData();

    // Have top point one node further down
    top = p.getNext();
    return x;
  }
}
```

Tracing **pop()**

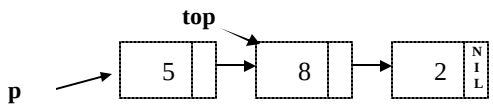
p = top



top = p ->next

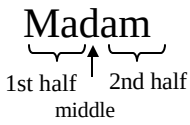


delete p



Example - A Program to Detect Palindromes

- An palindrome is a word that is spelled the way forwards and backwards.
- If we place the first half of the word on the stack, it should match the letters in the second half of the stack.
- Example:



RunPalindrome.java

```
import java.util.Scanner;
public class RunPalindrome  {
  // main() - A driver for the Palindrome class
  public static void main(String[] args) {
    Palindrome  pal = new Palindrome();
    String newWord = new String();
    Scanner keyb = new Scanner(System.in);

    // Get the word
    System.out.println(" Enter a word\t?");
    newWord = keyb.nextLine();
```

```
    // Print the result
    if (pal.isPalindrome(newWord))
      System.out.println(newWord
                         + " is an palindrome.");
    else
      System.out.println(newWord
                       + " is not an palindrome.");
  }
}
```

Palindrome.java

```
public class Palindrome  {
  // ispalindrome() - Test to see if a word is an
  //           palindrome by pushing the
  //           first half of the word on a stack
  public boolean isPalindrome(String x)  {
    boolean itIsPalindrome = true;
    char    oldChar, pushChar;
    String  t = new String();
    int     i, len, mid;
    MyStack   s = new MyStack();
    len = x.length();
    mid = len/2;
    // Push the first half on the stack
    for  (i = 0;  i < mid;  i++)  {
      pushChar = nextChar(x,i);
      s.push(pushChar);
    }
```

```
    // There are an odd number of letters
    // The middle letter matches itself
    if (len%2 == 1)
      i++;
    // Compare the second half to the first in
    // reverse order
    while (i < len) {
      oldChar = s.pop();
      if (oldChar != nextChar(x, i))
        return(false);
      i++;
    }
    return(true);
  }
```

```
  // nextChar()  - Returns the ith character in
  //               the string x in lower case
  private char nextChar(String x, int i) {
    String t = new String();

    // t is a one-character substring of x
    t = x.substring(i, i+1);
    // t is converted to lower case
    t = t.toLowerCase();
    // We want to return a char not a String
    return t.charAt(0);
  }
}
```

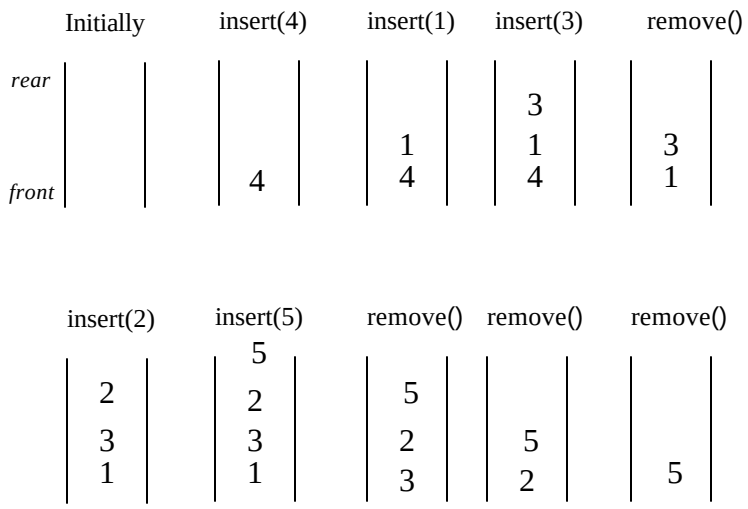
What is a Queue?

- A ***queue*** is a collection of data items where data is inserted on one end (called the ***rear***) and removed from the other end - call the ***front***.
- A queue is considered a first-in, first-out data structure because the oldest data is inserted first and removed last, preserving the order in which it is entered.

Queue Operations

- There are three basic operations on a stack:
  - ***empty*** ***true*** if the queue is empty; ***false*** if the queue is non-empty.
  - ***insert*** - inserts a data item on the rear of the queue.
  - ***remove*** - removes a data item from the front of the queue.

A Queue In Action



MyQueue.java

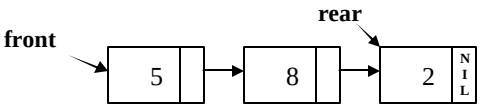
```
public class MyQueue  {
  private MyNode front, rear;

  // Queue() - A default constructor for
  //           queues
  public MyQueue()  {
    front = null;
    rear = null;
  }

  // Queue() - A constructor for queues where
  //           x is the first item inserted on
  //           the queue
  public MyQueue(int x)  {
    front = newNode(x);
    rear = front;
  }
```

```
  // newNode() - Creates a new node with the
  //             parameter x
  private MyNode newNode(int x)  {
    MyNode p = new MyNode();
    p.setData(x);
    p.setNext(null);
    return p;
  }

  private void error(String message)  {
    System.out.println(message);
    System.exit(1);
  }
}
```

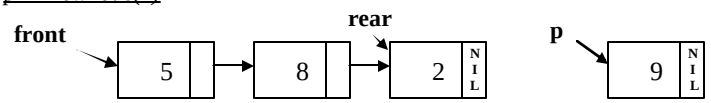
```
  // empty() - True is the stack is empty
  //           (null object)
  //    False is the stack is non-empty
  //           (non-null pointer)
  public boolean empty()  {
    if (front == null)
      return true;
    else
      return false;
  }
```

```
// insert() - Insert an item on the rear of
//            the queue
public void insert(int x)  {
  MyNode  p = newNode(x);

  if (rear == null)
    // This is the whole queue so front
    // and rear must bother point here
    front = p;
  else
    // Have the current last node
    // point to the new last node
    rear.setNext(p);

  // Readjust the queue to point to the new
  // rear node
  rear = p;
}
```

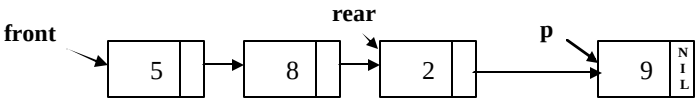
Tracing **`insert()`**

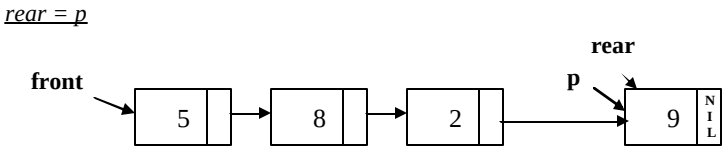


p = newnode(x)



rear -> next = p

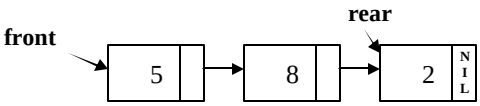




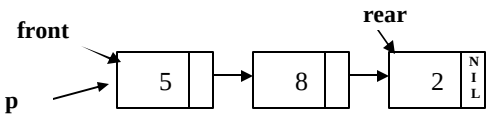

```
// remove() - Remove an item from the front
//            of the queue
public int remove()  {
  MyNode p;
  int  x = 0;

  if (front == null)
    // The queue is empty - there's nothing
    // to remove
    error("Queue underflow");
  else {
    // Have p point to the node we're removing
    // Have front pointing to the new front
    // node
    p = front;
    // Take the data so it can be returned
    x = p.getData();
```

```
    // Readjust front
    front = p.getNext();
    if (front == null)
      // rear must also be readjusted
      rear = null;
  }
  return x;
}
```

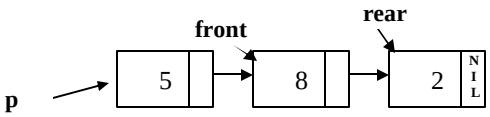
Tracing **`remove()`**



p = front



front = p -> next



delete p