Programming with Data Structures

CMPSCI 187
Spring 2016

- **Please find a seat**
  - **Try to sit close to the center (the room will be pretty full!)**
- **Turn off or silence your mobile phone**
- **Turn off your other internet-enabled devices**

Reminder

- Project 3 (sets)
- Tuesday Feb 16: follow Monday schedule
- Work on **practice exam** (download from Piazza) and make sure to go to the discussion section for review.
- **Wednesday Feb 17: mid-term, 7-9pm ILC N151**
- Please **bring a pencil** as you will need to fill opscan sheets.

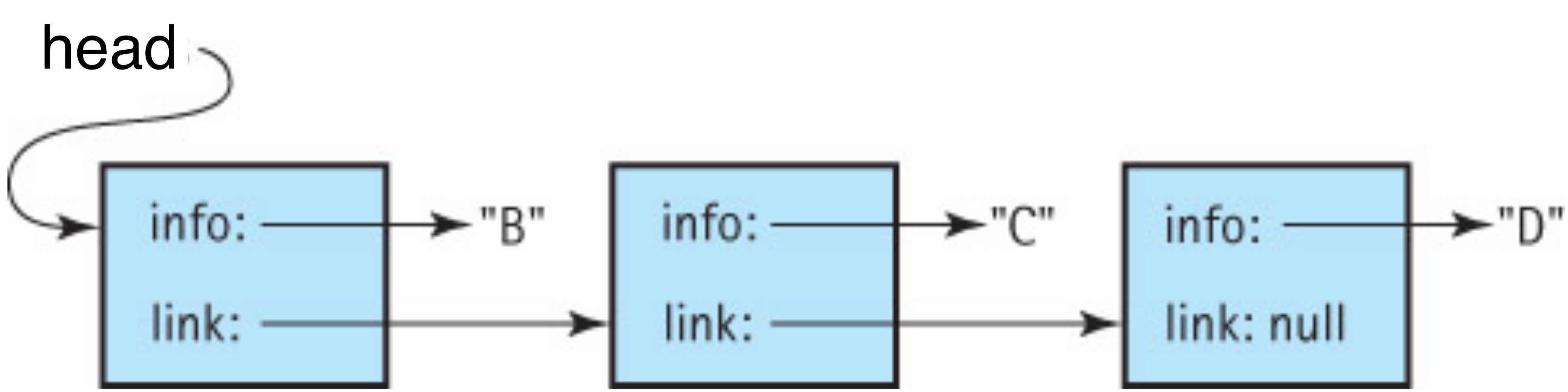
LL Stacks, Analysis, Postfix

- Linked List Stack
- Analysis of Stack Implementations
- Evaluating Postfix Expressions with a Stack

Recall Linked Lists

- For our `LinkedStringLog` class, we first defined a class `LLStringNode` which describes a node on the linked list.

- Each `LLStringNode` object had an `info` variable pointing to a `String`, and a `link` variable pointing to another `LLStringNode` (i.e. the next node on the linked list).



Generic Linked Lists

- To make a generic linked list (which can store not only string, but other type of data), we will define a generic class **`LLNode<T>`**. Its **`info`** variable points to an object of generic type **`T`**, and its **`link`** variable points to another **`LLNode<T>`** object.

- Note there is no need to do the explicit type casting business (as in the **`ArrayStack<T>`** case), because here we are not creating an **array** of generic type.

```java
public class LLNode<T> {
  private LLNode<T> link;
  private T info;

  public LLNode(T info) { this.info = info; }
```

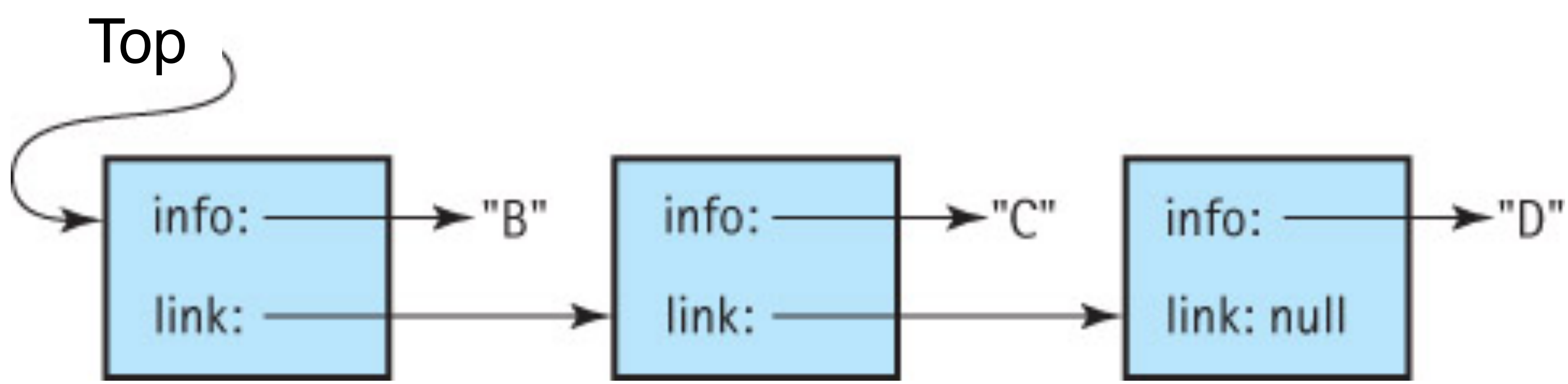
```java
public class LLNode<T> {
  private LLNode<T> link;
  private T info;

  public LLNode(T info) { this.info = info; }

  public void setInfo(T info)
  { this.info = info; }
  public void setLink(LLNode<T> link)
  { this.link = link; }
```

```java
public class LLNode<T> {
  private LLNode<T> link;
  private T info;

  public LLNode(T info) { this.info = info; }

  public void setInfo(T info)
  { this.info = info; }
  public void setLink(LLNode<T> link)
  { this.link = link; }

  public T getInfo( ) { return info; }
  public LLNode<T> getLink( )
  { return link; }
}
```

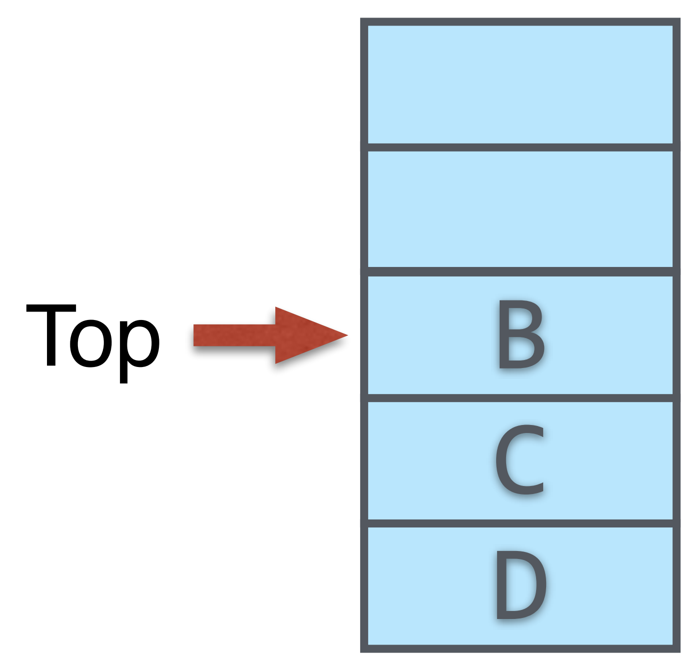
`LinkedStack<T>`

- Now let's implement a generic Stack using linked list as the underlying storage structure.
- The only variable we need for **`LinkedStack<T>`** is a pointer to the **top** of the stack. Though it just points to one node, it implicitly includes the entire chain of nodes as we can traverse the chain.
- Note: constructors for generic classes do not include the `<T>` in their name, but just the name of the class.

LinkedStack<T>



- Conceptually this is what the stack looks like:



Data and Constructors

```
public class LinkedStack<T> implements
    UnboundedStackInterface<T>
{
    protected LLNode<T> top;
    public LinkedStack( ) { top = null; }
}
```

- We define the top variable as protected so that any class that extends `LinkedStack<T>` can access this variable too.

Observers

```
public boolean isEmpty( ) {
                   ?
}

public T top( ) {



                   ?



}
```

- Think for a moment how you would implement these two methods

Observers

```
public boolean isEmpty( ) {
    return (top == null);
}

public T top( ) {
    if (!isEmpty( ))
        return top.getInfo( );
    else throw new StackUnderflowException
        (“top of empty stack”);
}
```

- The object `top` is a node, and we need to return its content rather than just itself.

Transformers

- Recall the `push()` and `pop()` methods.
- The `push()` method accepts a type `T` object, creates a new node, and adds that to the linked list.
  - The node will be inserted at the beginning of the linked list. Why?
  - Here `push()` will never throw an exception, why?
  - Why does the `push()` method accepts a type `T` object as parameter, and not an `LLNode<T>` object?

Transformers

```
public void push (T element) {
    LLNode<T> newNode = new LLNode<T>(element);
    newNode.setLink(top);
    top = newNode;
}

public void pop( ) {
    if (!isEmpty( ))
        top = top.getLink( );
    else throw new
        StackUnderflowException(“pop from empty”);
}
```

- The **pop()** method just discards the top element.

Comparing Running Time

- Will any of our methods take longer for a stack with many elements than for a stack with few elements?

- In other words, if N is the number of elements in the stack, what is the asymptotic running time of each of our methods as a function of N?

- Does it make a difference which implementation we use: `ArrayStack` or `LinkedStack`?

Comparing Running Time

- In fact all of **`push`**, **`pop`**, **`top`**, and **`isEmpty`** methods take O(1) time in each case.
- In **`ArrayStack`**, pushing and popping each involve moving one element of the array and changing topIndex, which is **`O(1)`**.
- In **`LinkedStack`**, we only need a **`O(1)`** pointer operations around the top node.

Stack Application: Postfix Expression

- Goal: evaluate arithmetic expressions.
- **Terminology**:
    - Operands: numbers
    - Operators: `+ - / *`
- For simplicity, we stick to binary operators (i.e. each operator takes two numbers as input).
- **Infix notation**: operators are placed between two operands. This is what we are familiar with.
    - **`(2 + 14) * 23`**

Postfix Expression

- **Postfix notation**: operators are placed after operands.
  - `5 3 -` `// 5 - 3`
  - `A B /` `// A / B`
  - `2 14 + 23 *` `// (2+14)*23`
- An operator acts on the two values to its left, where a value may be either an operand in the original expression or result of a previous operator.
- Note the order of the operands (further left -> operator -> closer left). This matters to `-` and `/`.

Postfix Expression

- **Postfix notation**: operators are placed after operands.
  - `5 3 -` `// 5 - 3`
  - `A B /` `// A / B`
  - `2 14 + 23 *` `// (2+14)*23`
- Also known as **RPN (Reverse Polish Notation)**
- Fun to play with RPN calculators.

Additional Examples

Infix	Postfix
A+B–C	AB+C–
A*B/C	AB*C/
A+B*C	ABC*+
A*B+C	
A*(B+C)	
A*B+C*D	
(A+B)*(C–D)	
((A+B)*C)–D	
A+B*(C–D/(E+F))	

Additional Examples

Infix	Postfix
A+B–C	AB+C–
A*B/C	AB*C/
A+B*C	ABC*+
A*B+C	AB*C+
A*(B+C)	ABC+*
A*B+C*D	AB*CD*+
(A+B)*(C–D)	AB+CD–*
((A+B)*C)–D	AB+C*D–
A+B*(C–D/(E+F))	ABCDEF+/–*+

Postfix Expression

- You may wonder how to convert an infix expression to postfix expression. That’s a rather involved topic.
- For now we assume we are given postfix expression to begin with, and we need to write an algorithm to evaluate it.
- Note that whenever you encounter an operator, you apply it to the last two operands (on the left).
- This suggests using a **stack to store the operands**.

Evaluate Postfix Expression

- Scan the expression
- When we see an **operand**, push it onto the stack.
- When we see an **operator**, pop two values off of the stack, and apply the operator to them, then push the result back onto the stack.
- At the end, if we have exactly one value on the stack, that is our answer.

Evaluate Postfix Expression

- Let's simulate the algorithm for the expression:

  `3 4 5 + 2 * 3 4 * - +`

- If we ever encounter an empty stack, or if we finish with more than one value on it, the postfix expression was not valid. Thus our algorithm also tests a candidate postfix expression for validity.

Evaluate Postfix Expression

- Examples of invalid postfix expression:
    - `1 2 3 +`
    - `1 + *`
- What would happen in our algorithm for the above two expressions?

```
while more token exist:
   get an token
   if token is an operand:
      stack.push(token);
   else:
      operand2 = stack.top();
      stack.pop();
      operand1 = stack.top();
      stack.pop();
      result = (operand1) token operand2;
      stack.push(result);
result = stack.top();
stack.pop();
return result
```

```java
public static int evaluate(String expression) {
     LinkedStack<Integer> stack = new LinkedStack<Integer>();
     int operand1, operand2, result = 0;
     String op;
     Scanner tokenizer = new Scanner(expression);
     while (tokenizer.hasNext( )) {
         if (tokenizer.hasNextInt( )) {
             stack.push(tokenizer.nextInt( ));
         } else {
```

```java
public static int evaluate(String expression) {
    LinkedStack<Integer> stack = new LinkedStack<Integer>();
    int operand1, operand2, result = 0;
    String op;
    Scanner tokenizer = new Scanner(expression);
    while (tokenizer.hasNext( )) {
        if (tokenizer.hasNextInt( )) {
            stack.push(tokenizer.nextInt( ));
        } else {
            op = tokenizer.next( );
            if (stack.isEmpty( )) throw new PFE("stack underflow");
            operand2 = stack.top( );
            stack.pop( );
            if (stack.isEmpty( )) throw new PFE("stack underflow");
            operand1 = stack.top( );
            stack.pop( );
            // result = operand1 op operand2
            stack.push(result);
        }
```

```
public static int evaluate(String expression) {
     LinkedStack<Integer> stack = new LinkedStack<Integer>();
     int operand1, operand2, result = 0;
     String op;
     Scanner tokenizer = new Scanner(expression);
     while (tokenizer.hasNext( )) {
         if (tokenizer.hasNextInt( )) {
             stack.push(tokenizer.nextInt( ));
         } else {
             op = tokenizer.next( );
             if (stack.isEmpty( )) throw new PFE("stack underflow");
             operand2 = stack.top( );
             stack.pop( );
             if (stack.isEmpty( )) throw new PFE("stack underflow");
             operand1 = stack.top( );
             stack.pop( );
             // result = operand1 op operand2
             stack.push(result);
         }
         if (stack.isEmpty( )) throw new PFE ("stack underflow");
         result = stack.top( );
         stack.pop( );
         if (!stack.isEmpty( )) thrown new PFE ("too many operands");
         return result;
     }
 }
```