

Lecture 5: Abstract Data Types: Lists, Stacks and Queues

Reading materials Dale, Joyce, Weems: chapters 2, 4, 5, 6, (parts of 3 that talk about recursive processing of lists) OpenDSA (https://opendsa-server.cs.vt.edu/ODSA/Books/Everything/html/): chapter 9 (Liang 10th: 10.2 (abstraction), 10.6 (stacks), Comprehensive: 22)

Contents

1	Abs	tract Data Types	3
	1.1	Collection	3
	1.2	List ADT	3
		1.2.1 Example: StringList ADT - Informal Specification	3
		1.2.2 Example: StringList ADT - Formal Specification	4
	1.3	Stacks ADT	4
		1.3.1 Example: CharStack ADT - Informal Specification	4
		1.3.2 Example: CharStack ADT - Formal Specification	5
	1.4	Queue ADT	5
		1.4.1 Example: ProcessQueue ADT - Informal Specification	5
		1.4.2 Example: PrintJobQueueADT - Formal Specification	5
2	Arra	ay Based Implementation of List, Stack and Queue	5
	2.1	Implementing a List ADT Using an Array	5
	2.2	Implementing a Stack ADT Using an Array	6
	2.3	Implementing a Queue ADT Using an Array	6
3	Linl	ked Structures	7
	3.1	Review of Java Reference Variables	7
	3.2	Linked Structures	7
		3.2.1 Linking Nodes Together	8
4	Linł	ked-Based Implementation of List, Stack and Queue	11
	4.1	Linked Lists	11
		4.1.1 insert method	11
		4.1.2 remove method	15
		4.1.3 clear method	16
		4.1.4 contains method	17
		4.1.5 indexOf method	17
		4.1.6 get method	17
		4.1.7 size method	18
	4.2	Stacks	18
		4.2.1 Wrapper Implementation	18
		4.2.2 Another Inefficient Implementation	19
		4.2.3 An Efficient Stack Implementation	19
	4.3	Queues	20

5	Performance Analysis 5.1 Singly Linked List ⇔ Array	20 20
6	Prefix and Postfix Expressions	21
	6.1 Prefix Notation (a.k.a. Polish Notation)	21
	6.2 Postfix Notation (a.k.a. Reverse Polish Notation)	21
	6.3 Today's Uses of Postfix Notation	22
7	Other Versions of Lists, Stacks and Queues	23
	7.1 Circular Singly Linked List	23
	7.2 Doubly Linked List	23
	7.3 Deque	23



1 Abstract Data Types

What is the abstraction? Abstraction is a model of a system that includes only the details needed by the viewer/user of such system. The complexity and details of actual implementation should be hidden from the user - that is called **information hiding**. Why? Because the complex design details are not relevant to the user and may make it harder to understand the system. Imagine what would happen if you needed to know and understand every detail of how cars work, before you could use them.

An **abstract data type** (**ADT**) provides a collection of data and a set of operations that act on the data. An ADT's operations can be used without knowing their implementations or how the data is stored, as long as the interface to the ADT is precisely specified. An ADT is implementation independent and, in fact, can be implemented in many different ways (and many different programming languages).

In this course we will use two different ways of specifying an ADT:

- informal specification an English description that provides list of all available operations on the data with their inputs and outputs;
- formal specification a Java interface definition that later can be implemented by concrete classes.

We will be using ADTs from three different perspectives:

- application level (or user, or client): using a class for which you only know the formal ADT specification;
- **logical level**: designing the ADT itself given a set of requirements specified by the non-programmer user (this may involve asking questions);
- implementation level: writing code for a class that implements a given ADT.

1.1 Collection

A collection object is an object that holds other objects. Typical operations provided by collection classes are:

- insert,
- remove,
- iterate through the collection.

A formal ADT for a generic collection is provided by Java's Collection<E> interface: http://docs.oracle.com/javase/8/docs/api/java/util/Collection.html. An ArrayList<E> class implements this ADT.

1.2 List ADT

There are many different possible List abstract data types that require different sets of operations to be defined on them. The ADT for the list that we define in this lecture is a very general one. We will use it (after slight revisions) in several future lectures and provide different ways of implementing the list interface.

For now, we will specify List ADTs that expect objects of a specific type. Later, we will revise the List ADT to be general enough to work with any chosen type of objects, i.e., we will define a generic List ADT.

1.2.1 Example: StringList ADT - Informal Specification

The StringList contains a (possibly empty) collection of objects of type String. The list supports the following operations:

insert This operation adds a String object, given as a parameter, to the list of strings.

remove This operation removes a String object, given as a parameter, from the list of strings. If given String object is not on the list, the list content does not change.

clear This operation removes all objects from the list. The list becomes empty.

contains This operation determines if a String object, given as a parameter, is stored in the list or not. It returns true or false, accordingly.

- **indexOf** This operation determines the index (or location) of a String object, given as a parameter. It returns -1 if the given String object is not stored in the list. (The indeces do not imply any ordering of the objects in the list and may have different value depending on the implementation of this ADT).
- get This operation returns an object stored at the index/location given as a parameter.
- size This operation determines the number of objects stored in the list.

toString This operation produces a meaningful String representation of the list.

1.2.2 Example: StringList ADT - Formal Specification

The formal specification of the ADT is given by Java's interface. This makes it language specific, but it is still general enough to be adaptable by implementations in other programming languages.

```
1
2 public interface StringList {
      void insert ( String item );
3
      void remove ( String item );
4
5
      void clear ( );
      boolean contains ( String item );
6
      int indexOf ( String item );
7
      String get ( int index );
8
9
      int size ( );
      String toString ();
10
11 }
```

See the StringList.java file in the lecture05 source code for a documented version of the above interface.

See Java's generic List<E> interface at docs.oracle.com/javase/8/docs/api/java/util/List.html for an example of similar interface.

1.3 Stacks ADT

Stacks are structures in which elements are always added and removed from the same end (depending on how you visualize the stack, you may wish to think of that end as the top of the stack). Stacks are **last in first out** (or **LIFO**) structures.

1.3.1 Example: CharStack ADT - Informal Specification

The CharStack contains a (possibly empty) collection of objects of type Character. The stack supports the following operations:

insert / push This operation adds a Character object, given as a parameter, to the top of the stack of characters.

remove / pop This operation removes and returns a Character object from the top of the stack of characters.

peek This operation returns a Character object from the top of the stack of characters.

toString This operation produces a meaningful String representation of the stack.

1.3.2 Example: CharStack ADT - Formal Specification

```
1
2 public interface CharStack {
3     public void push ( Character item ) ;
4     public Character pop () ;
5     public Character peek () ;
6     public String toString () ;
7 }
```

See the CharStack. java file in lecture source code for a documented version of the above interface.

See http://docs.oracle.com/javase/8/docs/api/java/util/Stack.html for Java's generic Stack<E> class.

1.4 Queue ADT

Queues are structures in which elements are added to one end (rear/back of a queue) and removed from the other end (front of a queue). Queues are **first in first out** structures (**FIFO**).

1.4.1 Example: ProcessQueue ADT - Informal Specification

The **ProcessQueue** contains a (possibly empty) collection of objects of type Process. Process is a user defined class representing a process waiting to be scheduled to run on the core of a CPU of a computer. The queue supports the following operations:

insert / enqueue This operation adds a Process object, given as a parameter, to the end of the queue of processes.

remove / dequeue This operation removes and returns a Process object from the front of the queue of processes.

toString This operation produces a meaningful String representation of the stack.

1.4.2 Example: PrintJobQueueADT - Formal Specification

```
1
2 public interface PrintJobQueue {
3     public void enqueue ( PrintJob item ) ;
4     public PrintJob dequeue () ;
5     public String toString () ;
6 }
```

See the PrintJobQueue.java file in lecture source code for a documented version of the above interface.

See http://docs.oracle.com/javase/8/docs/api/java/util/Queue.html for Java's generic Queue; E; interface. Notice how different methods are handled there when the queue is empty or full.

2 Array Based Implementation of List, Stack and Queue

We first look at the implementation of the three ADTs using storage based on arrays.

2.1 Implementing a List ADT Using an Array

The ArrayList<E> class implements Java's List<E> interface: http://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html.

You could easily (well, you should easily) create a class that contains an ordinary array as its data field and provides methods that implement all the operations required by our ADT discussed in Sec. 1.2.

The important things to notice is that our ADT for the list does not mention a possibility of a full list.

- So, how can we ensure that we always have enough room to add another item? After all the arrays in Java are fixed size, aren't they?
- How does **ArrayList<E**> class do it? Does it really use an array?

How big should the array be with an empty list? The quick answer is that it depends on the application. The array should start with some non-zero size. The initial size can be provided by the user as an option to the constructor.

Resizing an array. In order to "resize" an array that is full, we need to

- allocate a larger array,
- copy all the data from the full array to the new larger array,
- "abandon" the smaller array and use the larger array from now on.

This has to happen without user intervention. Any method that adds elements to the List needs to verify if there is enough room. If there isn't, it needs to resize the array before performing the addition.

How much extra space should we add to the array?

One extra element. This is a very bad choice! Copying of all the items from one array to another takes time. It should be avoided if possible. Increasing the array size by 1 means that we have to do it often.

Fixed number of elements. This generally works well. The actual number of elements should depend on the application.

Doubling current size. This is often implemented and produces good results assuring that the copying of the data is done infrequently. But for very large arrays, this may lead to a very large increase in memory usage.

2.2 Implementing a Stack ADT Using an Array

When implementing a stack ADT using an array, we need to resolve the same size issue as we encountered with the implementation of the list. The solution is the same as the one discussed in the previous section.

Where should the top be? There is another issue that we need to resolve though: at which end of the array should we have the top of the stack? This is the end at which we need to perform all push (addition), pop (removal) and peek operations. We have two options:

- index zero (or front) of the array this is a very bad choice that leads to slow operations (do not ever use this!), why?
- index of the last element of the array this leads to efficient implementation of the operations even though the <u>top</u> of the stack ends up moving as the stack grows (kind of like in a stack of real things)

2.3 Implementing a Queue ADT Using an Array

This one turns out to be the most complicated one from the three. We need to add to one end of the array and remove from the other. To optimize the performance of all the operations we want to make sure that we do not shift elements in the array during these operations. This leads to the implementation in which both ends move and we need to keep track of indexes that are the front and back of the queue.

3 Linked Structures



3.1 Review of Java Reference Variables

In Java, when you allocate a variable of a primitive data type (see http://docs.oracle.com/javase/tutorial/java/ nutsandbolts/datatypes.html for listing of types and descriptions) the memory location associated with that variable contains the value assigned to that variable.

Example:

The following lines of code

int v.	Х
$\mathbf{x} = 5$	5
$\Delta = J_{I}$	5

result in a 32-bit block of memory being associated with the variable x and then the bits are used to represent the value of 5 (for simplicity, I will just write the number 5 instead of its binary equivalent).

With reference type variables the memory allocation is different. When you create a reference variable it does not have any room (bits/bytes in memory) for storing data. It only contains enough memory to store a reference or an address of memory location at which the actual data is stored.

The following line of code creates a reference variable, unless it is followed by creation of an actual object, there is no room to store data.

Dorson	n•	р
1613011	Pr	room for a
		memory address

In fact, you should make a habit of always assigning the null value to a reference variable that does not actually reference any valid data:

Person	p = null;	p	1
		null	

The new operator is used to create the actual object and to assign its memory address to a reference variable.

<pre>p = new Person();</pre>	<u>p</u>	17A6F20B
	17A6F20B	Person object stored

And since we do not really care about the hexadecimal value of the memory address (nor do we really have a way of knowing it in Java), you'll see arrows used to show that a reference variable refers to or "points to" a memory location that contains the actual object.



3.2 Linked Structures

The idea of creating linked structures is based on the reference variables.

When you create an array, a consecutive block of memory locations is assigned to it. This allows to easily access each element using subscript operators.

Linked structures are composed of individual objects that are <u>linked</u> to one another is some way. For example, the following six objects are connected, but they do not necessarily exist in consecutive memory locations.



reference to the first object



Notice that each object has an extra "box" with an arrow coming out of it. The linked structures are composed of special kind of objects, usually referred to as **nodes**. Each node contains the actual data, but also a reference (possibly more than one) to another node. Linked structures get connected through these references to the next node.

3.2.1 Linking Nodes Together

We want to connect several nodes together in order to create a linked structure. Assume that our node will store a String object as the data. This means that each node contains data part (String object), and the reference to another node:

```
class StringNode {
   private String data;
   private StringNode next;
   public String getData () {
     return data;
   }
   public String getNext () {
     return next;
   }
   public void setData (String data ) {
     this.data = data;
   }
   public void setNext (StringNode next )
     this.next = next;
   }
```



}

The table below shows the steps needed to create several nodes and link them together into a linked structure.



Joanna Klukowska joannakl@cs.nyu.edu





4 Linked-Based Implementation of List, Stack and Queue

4.1 Linked Lists

A linked list can be simply thought of as a reference to the first node of a linked structure: from there we can follow the "links" to get to every other node. Our first linked list class contains a reference to the first node - we call it head, and several methods that allow us to work with a list. In fact, it implements the StringList interface that we discussed before:

```
1
2 public interface StringList {
3
      void insert ( String item );
4
      void remove ( String item );
      void clear ( );
5
      boolean contains ( String item );
6
      int indexOf ( String item );
7
      String get ( int index );
8
9
      int size ( );
10
      String toString ();
11 }
```

The implementation details and algorithms assume that we have a collection of connected nodes (as shown in previous section) with a reference to the first node called head.

4.1.1 insert method

We want to insert a new node into an existing list of nodes. The steps differ slightly depending on where the new node needs to be inserted: front, back or middle of the list.

Inserting an element at the beginning Assume that we have a linked list with four nodes:







WARNING: If you change where the head reference points to first, then the rest of the list will be lost!



Inserting an element at the end Again, assume that we start with a linked list with four nodes:





We want to add a new node at the end of this list, i.e., the current last node should point to our new node.



Since, we only have a reference to the first node, we need to find the last node (easy to find since it is the only one pointing to null). So we have a new reference, called current that now points to the last node.



Joanna Klukowska joannakl@cs.nyu.edu





In this case, the order of the last two steps does not matter. (In fact, if the StringNode constructor automatically sets the next reference to null that step can be omitted.)

These steps in code:

```
//create new node
StringNode newNode = new StringNode ( );
//create the current reference and point it to the last node
StringNode current = head;
while (current.next != null )
    current = current.getNext();
//make the new node point to null
newNode.setNext(null);
//make the last node point to the new last node
current.setNext (newNode);
```

Inserting an element in the middle The previous two scenarios covered special cases. Now, we have the same initial linked list, but we want to add a node in the middle. At this point it does not matter how the actual location is selected, so let's add it after the second node.

head



At this point it does not matter how the actual location is selected, so let's add it after the second node. We need the current reference to point to the node that should be before the new node after the insertion (the exact details of how to advance the current reference from head to that particular position depend on how the position is selected).







As with inserting at the front, we need to make sure not to disconnect the rest of the list during the insertion. The new node has to point to its successor





These steps in code:

```
//create new node
StringNode newNode = new StringNode ();
//create the current reference and advance it to
//the node after which we want to insert
StringNode current = head;
... //depends on specifics of the problem
//make the new node point to its successor
newNode.setNext(current.getNext());
```

```
newNode.setNext(current.getNext() );
//make the current node point to the new node
current.setNext(newNode);
```



4.1.2 remove method

As with the insertion, the removal of a node is slightly different depending on where the node is located in the list.

Removing an element at the beginning We start with our four-node linked list



Removing an element at the end We start with our four-node linked list

head



and we want to remove the very last node (the one that points to null). This means that we first need to find the node right before it ...



```
//before the last node
StringNode current;
current = head;
//assume that there are at least two nodes in the list
//(otherwise we are deleting the first node)
while ( current.getNext().getNext() != null )
    current = current.getNext();
//disconnect the last node by pointing the one
//before it to null
current.setNext(null);
```

Removing an element in the middle We start with our four-node linked list



and we want to remove a node in the middle (not the first one or the last one). We need the current reference to point to the node before the one that needs to be removed (how this is done depends on the specifics of how the node to be deleted is selected).



The node pointed to be current needs to point to the node "one away" in the list (the one that the node to be deleted is pointing to).



These steps in code:

```
//create a current reference and advance it to the one
//before the node to be deleted
StringNode current;
current = head;
... //depends on specifics of the problem
//connect the node pointed to by current with the node
//pointed to be the node being deleted
current.setNext(current.getNext().getNext() );
```

4.1.3 clear method

The clear() method of the list ADT empties the list. This can be accomplished by simply pointing the head reference of our list to null.

4.1.4 contains method

The contains() method determines if a particular data item is in the list or not. With linked lists the data items are stored inside the nodes. When implementing contains method we need to make sure that we are "looking at" the actual data items, not at the nodes.

Here is an algorithm for searching in the linked list:

```
make current reference equal to head
while current != null
   check if current.data is what we are after
   if yes
      return true
   advance current to the next node (current = current.next )
   if we reach this point we did not find what
   we were after so return false
```

4.1.5 indexOf method

This method may be considered inappropriate for the linked list - after all we do not use indexes to access the elements. But since the get method expects such index, we will provide it anyway. The index returned by this method does not provide any information regarding ordering between the elements. In fact, at this point we are dealing with unsorted lists.

The implementation of this method is very similar to the implementation of contains method: we need to locate the element and return its position rather than returning Boolean value indicating it was found or not.

Here is an algorithm for searching in the linked list and returning the position information:

```
make current reference equal to head
set index to zero
while current != null
check if current.data is what we are after
if yes
    return the index
    advance current to the next node (current = current.next )
    increment the index by one
if we reach this point we did not find what
we were after so return null (or -1 if return type is int)
```

4.1.6 get method

The get method returns the data element of the list at a particular location, call it index. Since we only have a reference to the first node, we need to jump over the first index-1 nodes and then return the data element in the indexth node.

Here is an algorithm for finding and returning $index^{th}$ node in a linked list:

```
make current reference equal to head
set counter to zero
while current != null and counter != index
increment the counter by one
if counter == index
return current.data
otherwise (we reached the end of the list)
return null
```



4.1.7 size method

The most efficient implementation of this method would make use of an auxiliary data field that keeps track of the number of elements in the list: its value is adjusted on every deletion and insertion. The method then simply returns a value of this variable.

But as an exercise we will develop this method to compute a size of a list in two different ways: iteratively and recursively.

Iterative algorithm The details of an iterative algorithm were discussed in previous parts of this section. We need to move through all the nodes incrementing a counter until we find a node that is pointing to null.

Here is an iterative algorithm for determining the size of a linked list:

```
make current reference equal to head
set counter to zero
while current != null
increment the counter by one
return the value of counter
```

Recursive algorithm We need to decide on the base case and the recursive step of the recursive algorithm. The smallest possible linked list is an empty list. In that case the head points to null and the size is zero. This is the base case of the algorithm. If the head reference does not point to null than we know that the size is at least 1 (since head points to an actual node) plus the size of the list that the first node points to. This is the recursive step.

Here is a recursive algorithm for determining the size of a linked list:

```
size ( StringNode current )
if current == null
return 0;
else return 1 + size(current.next);
```

4.2 Stacks

1

In implementing linked structure based stacks we are going to reuse the code developed in the previous section. The stack class implementation needs to implement the interface that we have been using previously (but this time it is written for String objects):

```
public interface StringStack {
    public void push ( String item ) ;
    public String pop () ;
    public String peek () ;
    public String toString () ;
    6}
```

4.2.1 Wrapper Implementation

The quick and dirty solution is to define a stack class that has our LinkedStringList as a data field and calls its respective methods. This approach reuses methods of the LinkedStingList class and **wraps** them in its own methods appropriate for a stack.

```
2 public class LinkedStringStack implements StringStack{
3
4 private LinkedStringList stack;
5
6 /* constructor needs to create a LinkedStringList object */
7 public LinkedStringStac ( ) {
8 stack = new LinkedStringList ();
9 }
```

10



```
11
      /* uses the insertBack() method of the list */
      public void push ( String item ) {
12
13
        if (item != null )
          stack.insertBack( item ) ;
14
15
      }
16
17
      /* uses the removeBack() method of the list,
          but needs to store the removed item to return it \star/
18
19
      public String pop () {
        String item = null;
20
        if ( ! stack.isEmpty () ) {
21
          item = stack.get( stack.size() - 1);
22
23
          stack.removeBack() ;
24
        }
25
        return item;
26
      }
27
28
      /* this uses the get() method of the list */
      public String peek () {
29
30
        String item = null;
        if ( ! stack.isEmpty () )
31
          item = stack.get( stack.size() - 1);
32
33
        return item;
34
      }
35
36
      public String toString () {
37
        return stack.toString();
38
39
      }
40 }
```

4.2.2 Another Inefficient Implementation

We can also implement the stack class with its own head reference, numOfElements variable and the methods

- push() that is identical to insertBack(),
- pop() that is identical to removeBack(),
- peek() that uses get() method to obtain the last element,
- toString().

This approach avoids making duplicate method calls, but is still very inefficient: we need to traverse the entire list for every insertion, deletion and get operation.

4.2.3 An Efficient Stack Implementation

Since a stack data structure only requires accesses (push, pop, peek) on one end of the linked list, the top of a stack should be at the very beginning of the list. This way every access takes only one step and there is never a reason to traverse the entire list.

An efficient linked list based stack implementation should have its own head reference, (optional) numOfElements variable and the methods

• push() that is identical to insertFront(),

- pop() that is identical to removeFront(),
- peek () that is identical to get () called with last index,
- toString().

Queues 4.3

As with the stack, in implementing linked structure based queues we are going to reuse the code developed for the linked list. The queue class implementation needs to implement the interface that we have been using previously (but this time it is written for String objects):

```
1
2 public interface StringQueue {
     public void enqueue ( String item ) ;
3
     public String dequeue () ;
4
5
     public String toString () ;
6 }
```

A queue requires accesses to both ends of the list, so on one of its operations the entire list has to be traversed. Assuming that the queue is somewhat balanced (i.e. number of insertions is approximately the same as number of removals), it does not really matter which end of the list is considered the front of the queue and which is considered the back of the queue.

A possible linked list based queue implementation would have its own head reference, (optional) numOfElements variable and the methods

- enqueue() that is identical to insertBack(),
- dequeue() that is identical to removeFront(),
- toString().

We will return to a linked list based implementation of a queue when we discuss doubly linked lists. This will provide a more efficient solution than the one above that uses singly linked lists.

5 **Performance Analysis**

Singly Linked List \Leftrightarrow Array 5.1

Assume the problem size is N and that we are working with an unsorted list.

operation	array implementation	linked list implementation
access an element at location i (i is variable)	O(1)	O(N)
find an element	O(N)	O(N)
insert an element at the beginning	O(N)	O(1)
insert an element at the end (assume we know the current number of ele- ments)	O(1), or O(N), if need to resize	O(N)
insert an element in the middle based on index (somewhere inside, not first or last)	O(N)	O(N)
remove an element in the front	O(N)	O(1)
remove an element in the end (assume we know the current number of elements)	O(1)	O(N)
remove an element in the middle based on index (somewhere inside, not first or last)	O(N)	O(N)



6 Prefix and Postfix Expressions

How would you write a program to evaluate a mathematical, arithmetic expression? For example something like this

$$((15/(7 - (1 + 1))) \times 3) - (2 + (1 + 1)).$$

The code that can evaluate such expressions has to:

- find and evaluate all subexpressions that are surrounded by parenthesis;
- for each operator figure out what its operands are; this involves determining precedence of operators (i.e. knowing the order of operations).

It turns out that it is much simpler to write code for evaluation of arithmetic expressions when they are written in a format that lets us ignore parenthesis and that does not depend on operator precedence. These two ways are prefix and postfix notations.

6.1 Prefix Notation (a.k.a. Polish Notation)

For more details and the history see the Wikipedia page at http://en.wikipedia.org/wiki/Polish_notation.

Prefix notation is a notation for writing arithmetic expressions in which the operator comes before its operands.

infix		tfix													
2 + 5	+	2	5												
$(2+4) \times 5$	×	+	2	4	5										
$2+4 \times 5$	+	2	×	4	5										
$((15/(7 - (1 + 1))) \times 3) - (2 + (1 + 1))$	-	×	/	15	_	7	+	1	1	3	+	2	+	1	1

```
Scan the given prefix expression from right to left
for each token in the input prefix expression
if the token is an operand then
    push onto stack
else if the token is an operator then
    operand1 = pop stack
    operand2 = pop stack
    compute operand1 operator operand2
    push result onto stack
return top of stack as result
```

Exercise: Apply this algorithm to the following expression:

- * / 15 - 7 + 1 1 3 + 2 + 1 1

6.2 Postfix Notation (a.k.a. Reverse Polish Notation)

For more details and the history see the Wikipedia page at http://en.wikipedia.org/wiki/Reverse_Polish_notation.

Postfix notation is a notation for writing arithmetic expressions in which the operator follows its operands.

Joanna Klukowska joannakl@cs.nyu.edu

infix	postfix
2 + 5	$2 \ 5 \ +$
$(2+4) \times 5$	$2 \ 4 \ + \ 5 \ imes$
$2+4 \times 5$	$2 \ 4 \ 5 \ imes \ +$
$((15/(7-(1+1))) \times 3)-(2+(1+1))$	$15 7 1 1 + - / 3 \times 2 1 1 + + -$

```
Scan the given postfix expression from left to right
for each token in the input postfix expression
  if the token is an operand
    push it (its value) onto a stack
  else if the token is an operator
    operand2 = pop stack
    operand1 = pop stack
    compute operand1 operator operand2
    push result onto stack
return top of the stack as result
```

Exercise: Apply this algorithm to the following expression:

15 7 1 1 + - / 3 * 2 1 1 + + -

```
for each character in the input infix string expression
  if the character is an operand
    append to postfix string expression
 else if the character is a left brace
    push it onto the operator stack
 else if the character is an operator
    if the stack is not empty
      while top element on the stack has higher precedence
        pop the stack and append to postfix string expression
    push it (the current operator) onto the operator stack
 else if the character is a right brace
    while the operator stack is not empty
      if the top of the operator stack is not a matching left brace
        pop the operator stack and append to postfix string expression
      else
        pop the left brace and discard
       break
while the operator stack is not empty
 pop the operator stack and append to postfix string expression
```

6.3 Today's Uses of Postfix Notation

RPN stands for "Reverse Polish Notation" which is another name for postfix notation.

Why use RPN Calculator? http://todllc.com/blog/9-reasons-to-use-an-rpn-calculator/ RPN Calculator App for Android https://play.google.com/store/apps/details?id=com.ath0.rpn&hl=en



RPN Calculator 48 App for iPad/iPhone https://itunes.apple.com/us/app/rpn-calculator-48/id336580727?mt=8

Many of the HP Calculators still use RPN as well as infix notation.

7 Other Versions of Lists, Stacks and Queues

We discussed the basic definitions and implementations of a singly linked list, a stack and a queue. We saw that in some cases it would have been mode efficient or, at least, more convenient to have additional features and capabilities in those implementations. Such changes result in slightly modified structures that we will discuss now.

7.1 Circular Singly Linked List

This structure is still singly linked (only forward references from each node), but the last node, instead of "pointing" to null, points to the first node. This requires that all the methods that need to traverse the entire list have a different loop termination condition or base case for recursion. They cannot look for the null reference. Instead, the end of the list is indicated by a node that references the same node as the head.

WARNING: the implementation that looks for the end node which has the same reference as the head reference has to

- Make sure that it is really the same node that the head is pointing to, NOT that it just contains the same data.
- Make sure that such node is found after traversing the whole list, NOT when the current is initialized to head reference.

What would be an application of a circular linked list?

7.2 Doubly Linked List

The node used in a doubly linked list has references to both its successor and its predecessor.

```
class Node <T> {
  T data;
  Node<T> next;
  Node<T> prev;
  //methods
}
```

The first node is pointed to by a head reference, the last node is pointed to by a tail reference. The first node has prev pointing to null and the last node has next pointing to null.

Java's LinkedList;E¿ class http://docs.oracle.com/javase/8/docs/api/java/util/LinkedList.html provides an implementation of a doubly linked list.

7.3 Deque

This strange name is short for **double ended queue**. It is a hybrid structure that allows insertions and removals at both ends of the queue.

Java has a Deque<E> interface http://docs.oracle.com/javase/8/docs/api/java/util/Deque.html and one of its implementations if provided by Java's LinkedList;E¿ class http://docs.oracle.com/javase/7/docs/api/java/util/LinkedList.html (the same one that implements a doubly linked list).