Chapter 20 Lists, Stacks

CS165
Colorado State University

Original slides by Daniel Liang
Modified slides by
Wim Bohm, Sudipto Ghosh

What is a Data Structure?

- A collection of data *elements*
- Stored in a structured fashion
- With operations that access & manipulate elements

Java Collections Framework

- *Collection* is a java interface
  - java.util.Collection
- Defines abstract methods for objects that contain other objects (*elements*)
  - add(E e)
  - remove(E e)
  - contains(E e)
  - toArray(E e)

These are examples, not an exhaustive list

Three Types of Collections

- Lists – Store elements in sequential order
  - Ordered Collection
- Sets – lists allow duplicates, sets do not
  - Unordered Collection
- Maps – data structure based on <key, value> pairs
  - Holds two objects per entry
  - May contain duplicate values
  - Keys are always unique

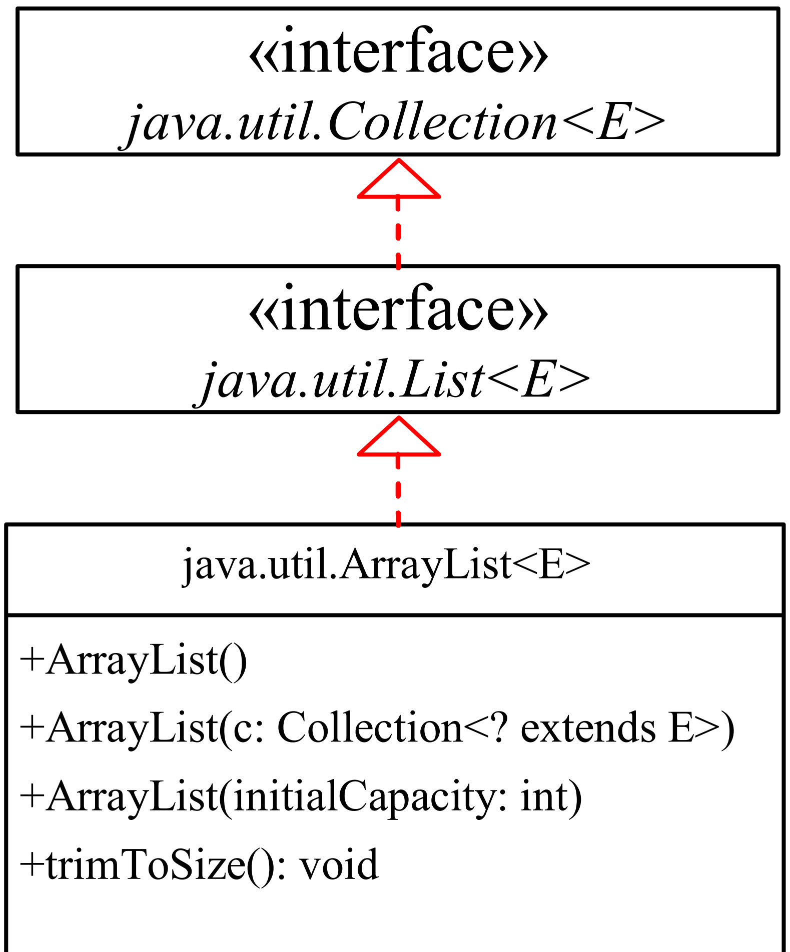
The List Interface

- Elements stored in sequential order
- Programs can specify where an element is stored.
- Programs can access elements by index.

Array vs ArrayList vs LinkedList

- Array
  - Allows element update, but does not support insertion or deletion of elements
  - But the most efficient if insert/delete not needed
- ArrayList class and the LinkedList class
  - Concrete implementations of the List interface.
  - Usage depends on your specific needs (later)
    - ArrayList – Fast random access through indices
    - LinkedList – Fast insertion and deletion of elements at specific locations

java.util.ArrayList



interface List<E>
extends Collection<E>

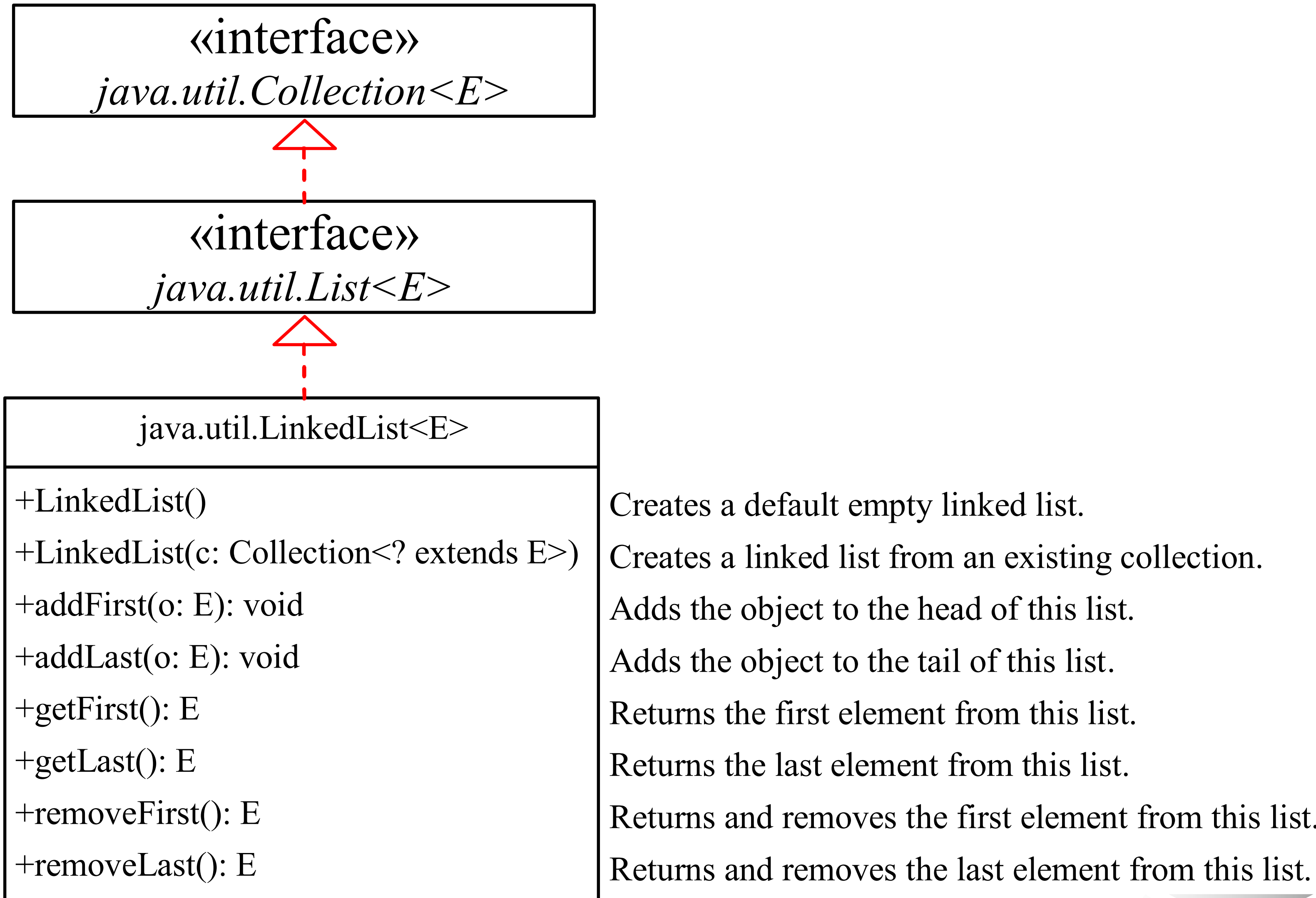
Creates an empty list with the default initial capacity.

Creates an array list from an existing collection.

Creates an empty list with the specified initial capacity.

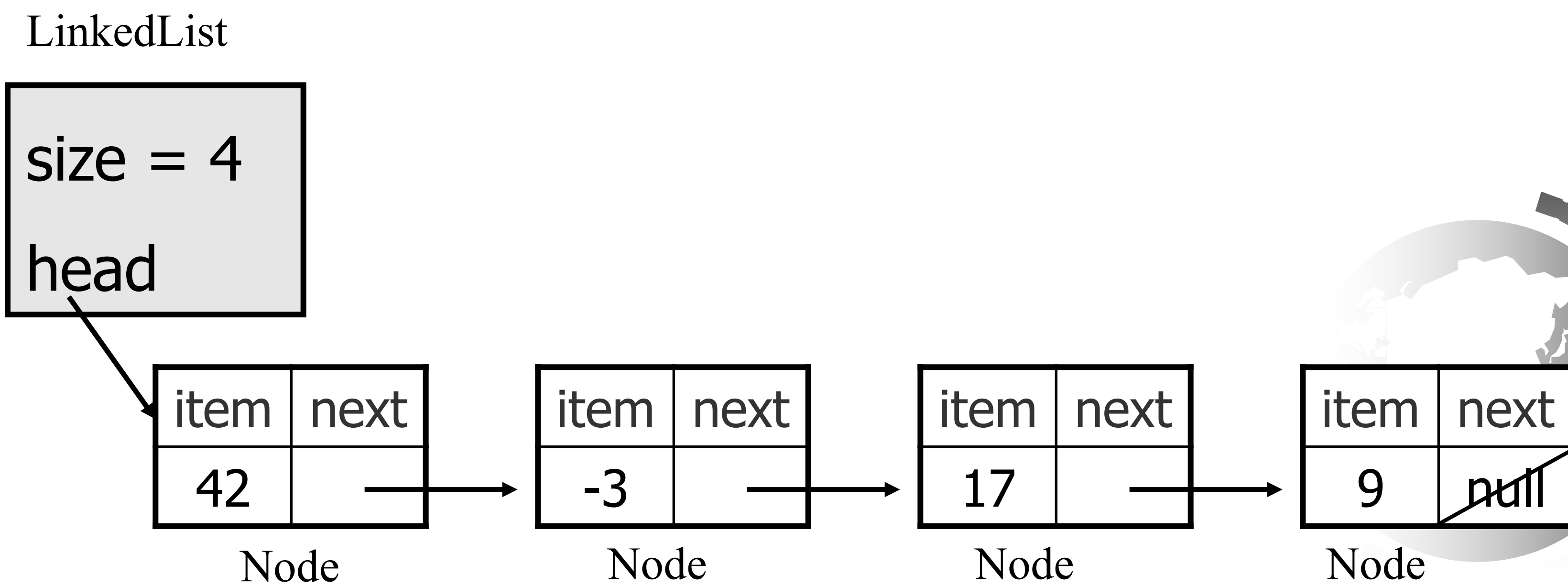
Trims the capacity of this ArrayList instance to be the list's current size.

java.util.LinkedList



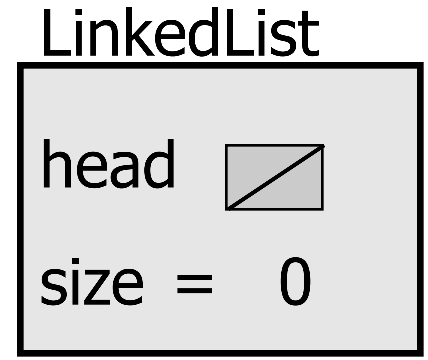
Linked List

- A structure containing (at least) the **size** of the list (# nodes in it) and a **head:** a reference to the first node. (LinkedList object)
- A sequence of nodes, first referring to second referring to third etc. (Node objects)



Linked List: constructor

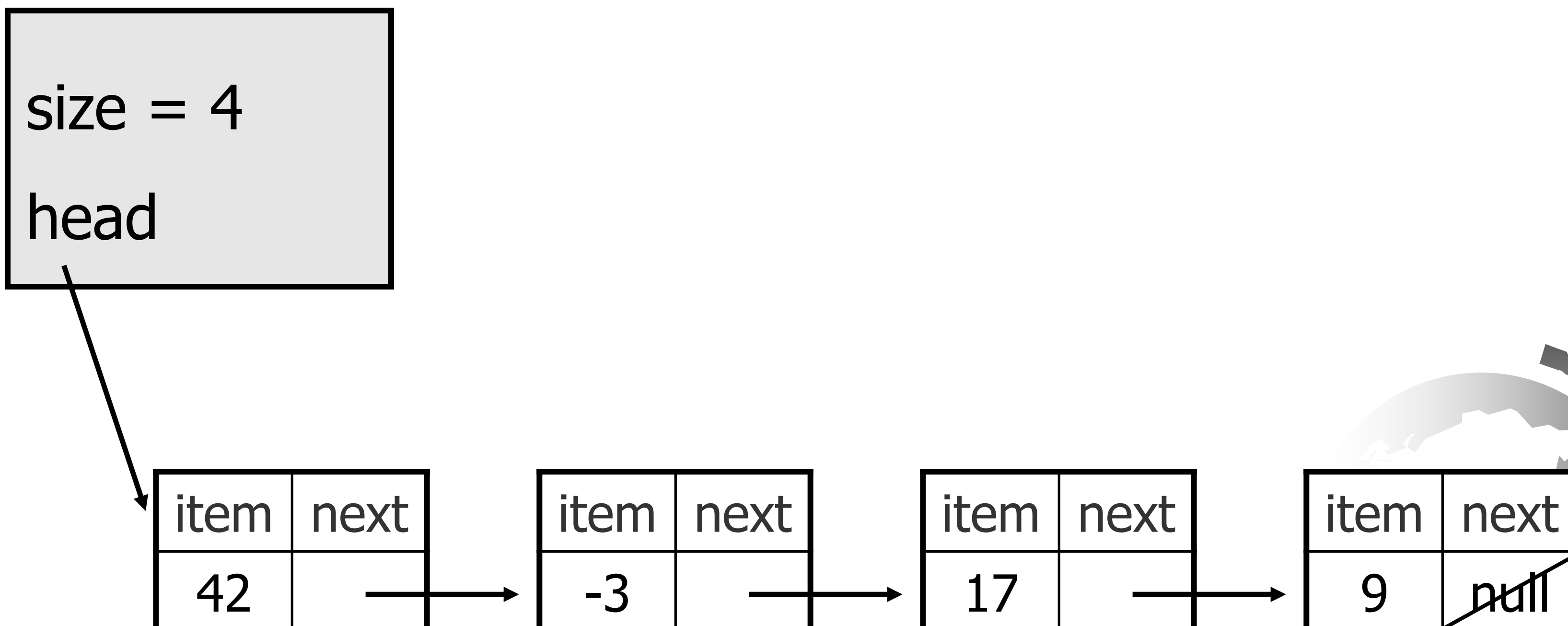
```
public class LinkedList {
    private Node head;
    private int size;

    public LinkedList() {
        head = null;
        size = 0;
    }

// Code for add, remove, find, clear . .

}
```



```java
public class Node {
    private Object item;
    private Node next;
    public Node(Object item) {
        this.item = item;
        this.next = null;
    }
    public Node(Object item, Node next) {
        this.item = item;
        this.next = next;
    }
    public void setNext(Node nextNode) {
        next = nextNode;
    }
    public Node getNext() {
        return next;
    }
    public Object getItem() {
        return item;
    }
    public void setItem(Object item){
        this.item = item;
    }
}
```

Implementing `add`

- How do we add to a linked list at a given index?



Implementing add

- How do we add a node to a linked list at a given index?

Consider all the possible cases!

1. Index out of bounds
2. Insert at head
3. Insert in middle
4. Insert at end

The add method

```
public void add(int index, Object item){
   if (index<0 || index>size)
        throw new IndexOutOfBoundsException(”out of bounds”);
   if (index == 0) {
        head = new Node(item, head);
   }
   else { // find predecessor of node
        Node curr = head;
        for (int i=0; i<index-1; i++){
           curr = curr.getNext();
        }
        curr.setNext(new Node(item, curr.getNext()));
   }
   size++;
}
```

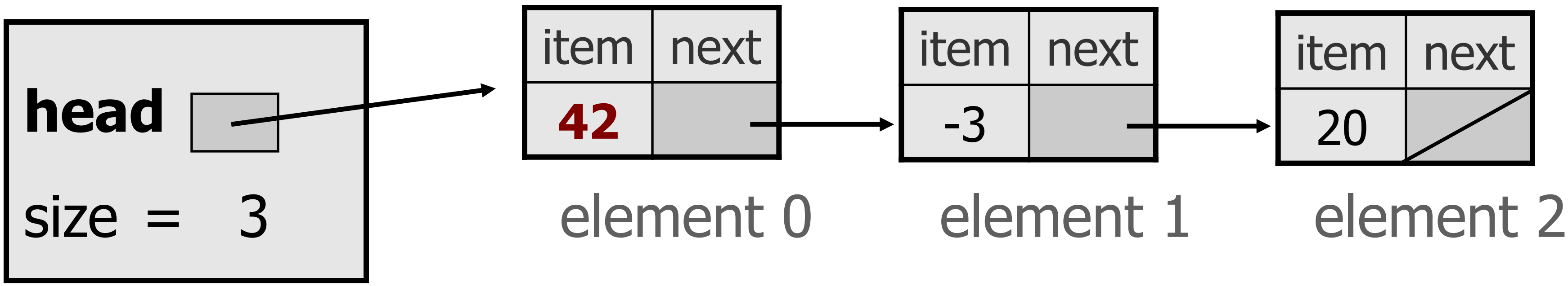
Implementing `remove`

- How do we remove a node?

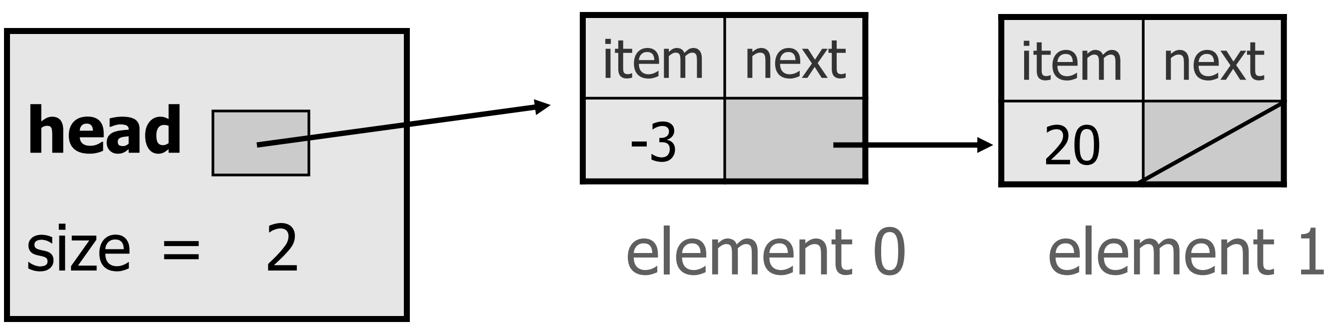
- Cases:
  - Index out of range
  - At the head
  - In the middle
  - At the end

Removing the first node

- Before removing element at index 0:



- After:

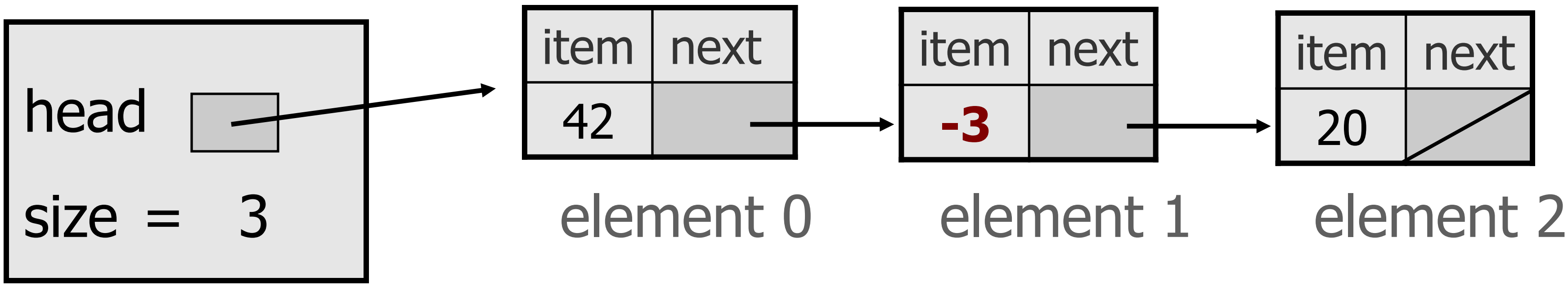


The `remove` method

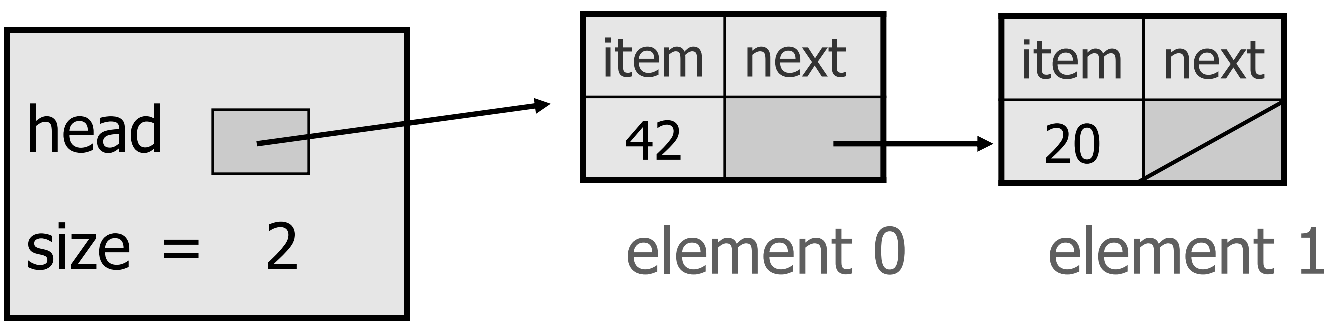
```
public void remove(int index) {
   if (index<0 || index >= size)
         throw new IndexOutOfBoundsException
                ("List index out of bounds");
    if (index == 0) {
        // special case: removing first element
        head = head.getNext();
    } else {
        // removing from elsewhere in the list
        Node current = head;
        for (int i = 0; i < index - 1; i++) {
            current = current.getNext();
        }
        current.setNext(current.getNext().getNext());
    }
    size--;
}
```

Removing node from the middle

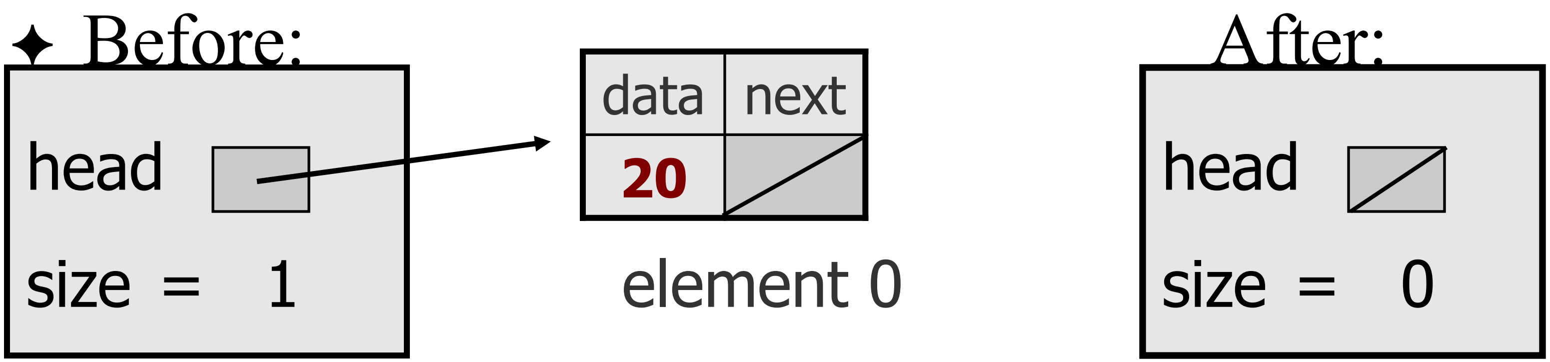
- Before removing element at index 1:



- After:



List with a single element



- We must change head to `null`.
- Do we need a special case to handle this?

The clear method

- How do you implement a method for removing all the elements from a linked list?

The clear method

```
public void clear() {
     head = null;
     size = 0;
}
```

- Where did all the memory go?
- Java’s garbage collection mechanism takes care of it!
- An object is eligible for garbage collection when no references exist that refer to it

Linear **time**-ordered structures Stacks and Queues

- Two data structures that reflect a ***temporal*** relationship
  - order of removal based on order of insertion
- We will consider:
  - "last come, first serve: take from the top of the pile"
    - last in first out - LIFO (stack)
  - "first come, first serve"
    - first in first out - FIFO (queue)

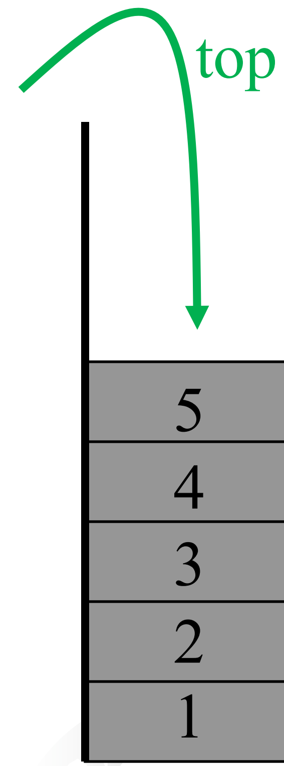
Stacks or queues?

What can we do with coin dispenser?

- "**push**" a coin into the dispenser.
- "**pop**" a coin from the dispenser.
- "**peek**" at the coin on top, but don't pop it.
- "**isEmpty**" check whether this dispenser is empty or not.

Stacks

- Last In First Out (LIFO) structure
  - A stack of dishes in a café
  - A stack of quarters in a coin dispenser
- Add/Remove done from **same end**: the top



Possible Stack Operations

- **`isEmpty()`**: determine whether stack is empty
- **`push()`**: add a new item to the stack
- **`pop()`**: remove the item added most recently
- **`peek()`**: retrieve, but don't remove, the item added most recently
- What would we call a collection of these ops?
  - An Interface

Checking for balanced braces

- How can we use a stack to determine whether the braces in a string are **balanced**?

abc{defg{ijk}{l{mn}}op}qr

abc{def}}{ghij{kl}m

Can you define balanced?

Pseudocode

```
while ( not at the end of the string){
      if (the next character is a “{“){
            aStack.push(“{“)
      }
      else if (the character is a “}”) {
            if(aStack.isEmpty()) ERROR!!!
            else aStack.pop()
      }
}
if(!aStack.isEmpty()) ERROR!!!
```

question

- Could you use a single int to do the same job?

- How?

Try it on

abc{defg{ijk}{l{mn}}op}qr {st{uvw}xyz}

abc{def}}{ghij{kl}m

Expressions

- Types of Algebraic Expressions
  - Prefix
  - Postfix
  - Infix
- Prefix and postfix are easier to parse. No ambiguity. Infix requires extra rules: **precedence** and **associativity**. **What are these?**
- Postfix: operator applies to the operands that immediately precede it.
- Examples:
  1. - 5 * 4 3
  2. 5 - 4 * 3
  3. 5 4 3 * -

Infix Expressions

- **Infix** notation places each operator between two operands for binary operators:

```
A * x * x + B * x + C; // quadratic equation
```

- This is the customary way we write math formulas in programming languages.
- However, we need to specify an order of evaluation in order to get the correct answer.

Evaluation Order

- The evaluation order you may have learned in math class is named PEMDAS:

parentheses → exponents → multiplication, division → addition, subtraction

Associativity

Operators with same precedence:

* /

and

\+ -

are evaluated left to right: 2-3-4 = (2-3)-4

Infix Example

- How a Java infix expression is evaluated, parentheses added to show association.

```
z = (y * (6 / x) + (w * 4 / v)) – 2;
z = (y * (6 / x) + (w * 4 / v)) – 2; // parentheses
z = (y * (6 / x)) + (w * 4 / v) – 2; // multiplication (L-R)
z = (y * (6 / x)) + ((w * 4) / v) – 2; // multiplication (L-R)
z = (y * (6 / x)) + ((w * 4) / v) – 2; // division (L-R)
z = ((y * (6 / x)) + ((w * 4) / v))) – 2; // addition (L-R)
z = ((y * (6 / x)) + ((w * 4) / v))) – 2; // subtraction (L-R)
z = ((y * (6 / x)) + ((w * 4) / v))) – 2; // assignment
```

Postfix Expressions

- **Postfix** notation places the operator after two operands for binary operators:

A * x * x + B * x + C // infix version

A x * x * B x * + C + // postfix version

- Also called reverse polish notation, just like a vintage Hewlett-Packard calculator!
- No need for parentheses, because the evaluation order is unambiguous.

Postfix Example

- Evaluating the same expression as postfix, must search left to right for operators:

(y * (6 / x) + (w * 4 / v)) – 2 // original infix

y 6 x / * w 4 * v / + 2 - // postfix translation

(y (6 x /) *) w 4 * v / + 2 -

((y (6 x /) *) w 4 * v / + 2 -

(y (6 x /) *) (w 4 *) v / + 2 -

(y (6 x /) *) ((w 4 *) v /) + 2 –

((y (6 x /) *) ((w 4 *) v /) +) 2 -

(((y (6 x /) *) ((w 4 *) v /) +) 2 -)

Prefix Expressions

- **Prefix** notation places the operator before two operands for binary operators:

```
A * x * x + B * x + C        // infix version
```

```
+ +  * * A x  x * B x C      // prefix version
```

- Also called polish notation, because first documented by polish mathematician.
- No need for parentheses, because the evaluation order is unambiguous.

Infix, Postfix, Prefix Conversion

Infix	Postfix	Prefix	Notes
A * B + C / D	A B * C D / +	+ * A B / C D	multiply A and B, divide C by D, add the results
A * (B + C) / D	A B C + * D /	/ * A + B C D	add B and C, multiply by A, divide by D
A * (B + C / D)	A B C D / + *	* A + B / C D	divide C by D, add B, multiply by A

What type of expression is 5 4 3 – *

A. Prefix

B. Infix

C. Postfix

D. None of the above (i.e., illegal)

What is the infix form of 5 4 3 – *

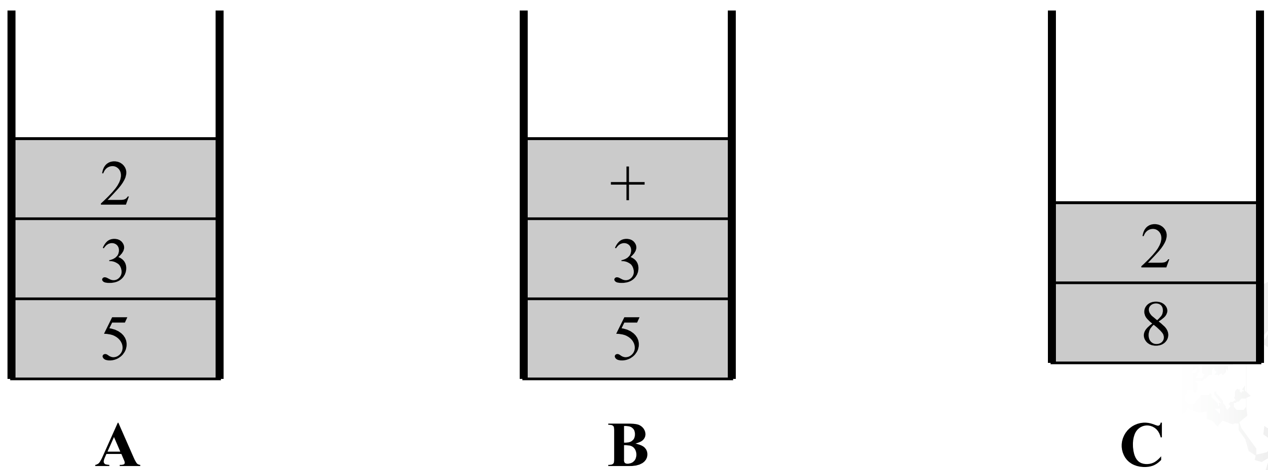
Evaluating a Postfix Expression

```
while there are input tokens left
    read the next token
    if the token is a value
        push it onto the stack.
    else
        // the token is an operator taking n arguments
        pop the top n values from the stack and perform the operation
        push the result on the stack
if there is only one value in the stack return it as the result
else
    throw an exception
```

Draw Stacks to evaluate 5 3 + 2 *

Quick check

- If the input string is “5 3 + 2 *”, which of the following could be what the stack looks like when trying to evaluate it?



Stack Interface

push(StackItemType newItem)

- adds a new item to the top of the stack

StackItemType pop() throws StackException

- deletes the item at the top of the stack and returns it
- Exception when deletion fails

StackItemType peek() throws StackException

- returns the top item from the stack, but does not remove it
- Exception when retrieval fails

boolean isEmpty()

- returns true if stack empty, false otherwise

Comparison of Implementations

- Options for Implementation:
  - **Array** based implementation
  - **Array List** based implementation
  - **Linked List** based implementation
- What are the advantages and disadvantages of each implementation?
- Let's look at a Linked List based implementation

Stacks and Recursion

- Most implementations of recursion maintain a stack of activation records, called

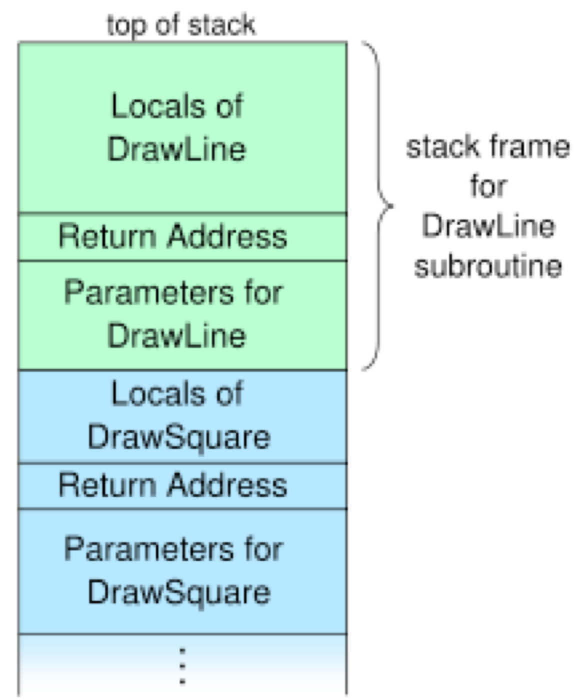
  the Run Time Stack

- Activation records, or Stack Frames, contain parameters, local variables and return information of the method called

- The most recently executed activation record is stored at the top of the stack. So a call pushes a new activation record on top of the RT stack

Applications - the run-time stack

- Nested method calls tracked on call stack (aka run-time stack)
  - First method that returns is the last one invoked
- Element of call stack - activation record or stack frame
  - parameters
  - local variables
  - return address: pointer to next instruction to be executed in calling method



http://en.wikipedia.org/wiki/Image:Call_stack_layout.svg

Factorial example

```
int factorial(n){
  // pre n>=0
  // post return n!
  if(n==0) { r=1; return r;}
  else {r=n*factorial(n-1); return r;}
}
```

RTS factorial(3): wind phase

if(n==0) { r=1; return r;}
*else {r=n*factorial(n-1); return r;}*

only active frame: top of the run time stack

			n=0, r=1
		n=1, r=?	n=1, r=?
	n=2, r=?	n=2, r=?	n=2, r=?
n=3, r=?	n=3, r=?	n=3, r=?	n=3, r=?

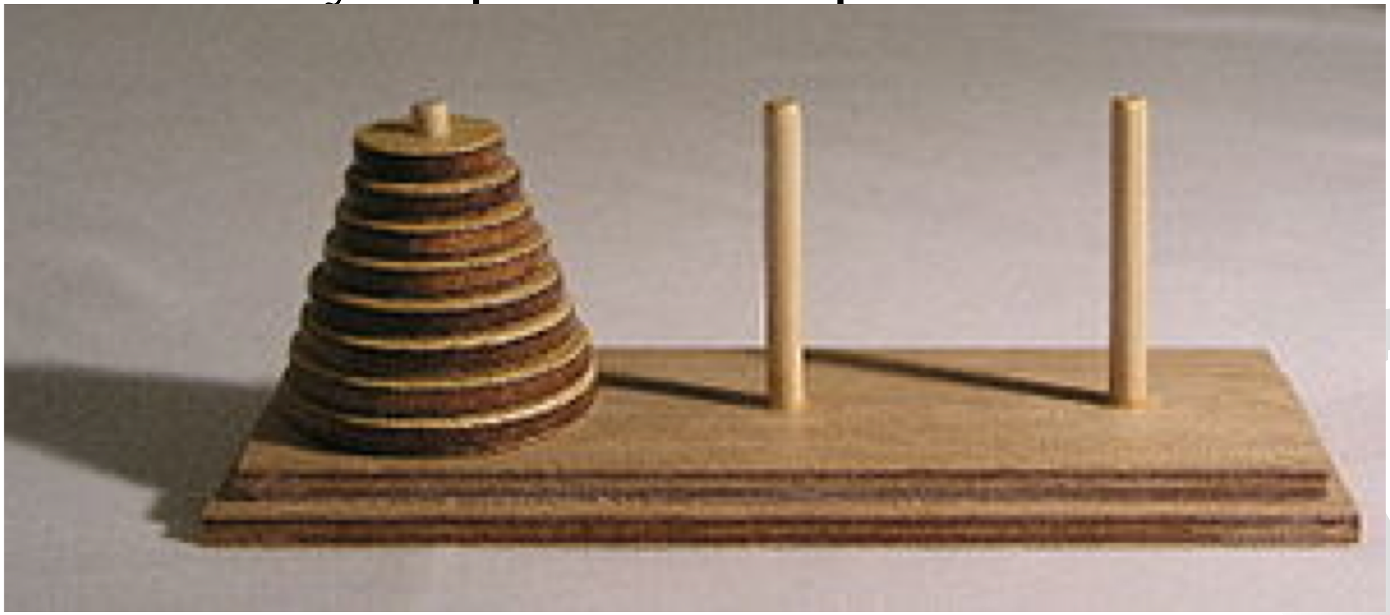
RTS factorial(3): unwind phase

if(n==0) { r=1; return r;}
*else {r=n*factorial(n-1); return r;}*

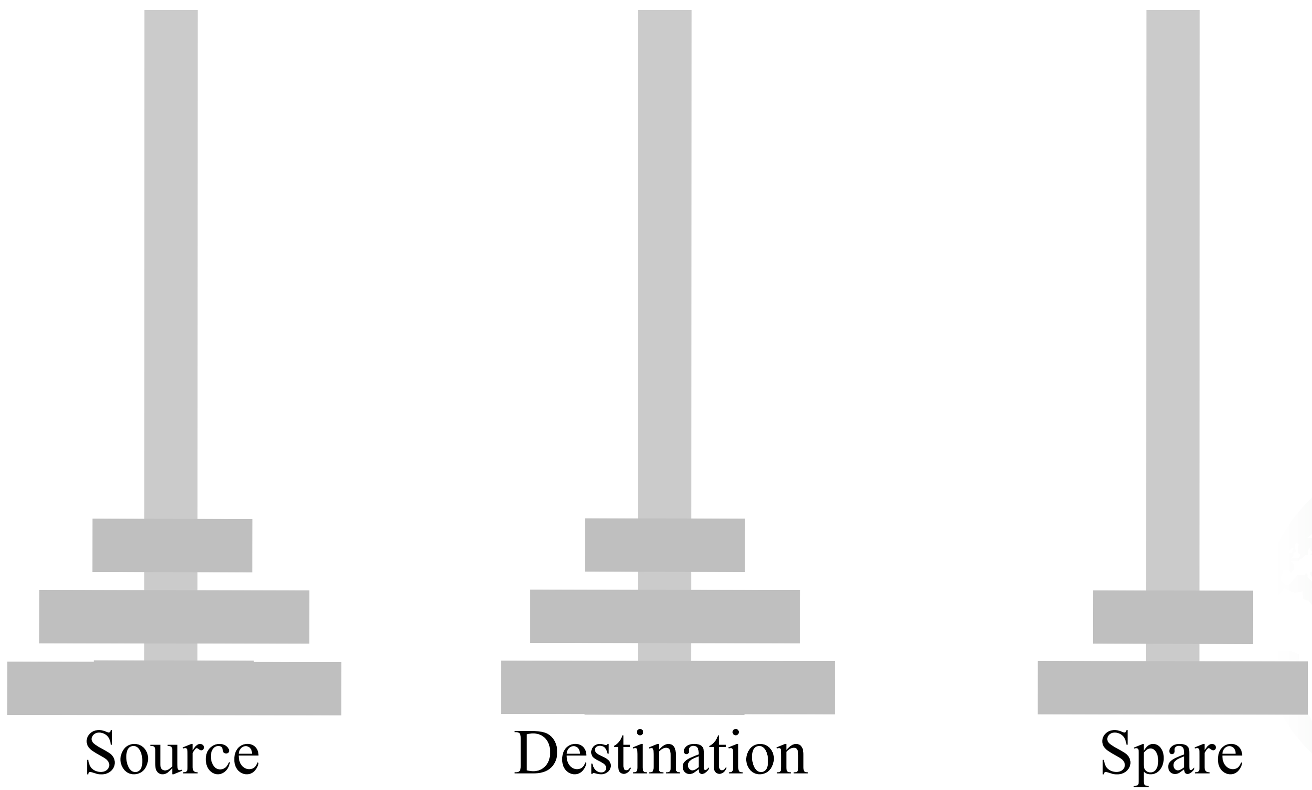
n=1, r=1			
n=2, r=?	n=2, r=2		
n=3, r=?	n=3, r=?	n=3, r=6	return 6

More complex example: The Towers of Hanoi

- **Move pile of disks from source to destination**
- **Only one** disk may be moved at a time.
- No disk may be placed on top of a smaller disk.



Moves in the Towers of Hanoi



Recursive Solution

```
// pegs are numbers, via is computed
// f: from: source peg, t: to: destination peg, v: via: intermediate peg
// state corresponds to return address, v is computed
public void hanoi(int n, int f, int t){
  if (n>0) {
    // state 0
    int v = 6 - f - t;
    hanoi(n-1,f, v);
    // state 1
    System.out.println("move disk " + n + " from " + f + " to " + t);
    hanoi(n-1,v,t);
    //  state 2
  }
}
```

Run time stack for hanoi(3,1,3)

```
if (n>0) {
    // state 0
    int v = 6 - f - t;
    hanoi(n-1,f, v);
    // state 1
    System.out.println("move disk " + n +
                       " from" + f + " to" + t);
    hanoi(n-1,v,t);
    // state 2
  }
```

only active frame:
top of the run time stack

0:n=3,f=1,t=3

0:n=2,f=1,t=2
1:n=3,f=1,t=3

0:n=1,f=1,t=3
1:n=2,f=1,t=2
1:n=3,f=1,t=3

0:n=0,f=1,t=2
1:n=1,f=1,t=3
1:n=2,f=1,t=2
1:n=3,f=1,t=3

Run time stack for hanoi(3,1,3)

```
if (n>0) {
    // state 0
    int v = 6 - f - t;
    hanoi(n-1,f, v);
    // state 1
    System.out.println("move disk " + n +
                    " from" + f + " to" + t);
    hanoi(n-1,v,t);
    // state 2
  }
```

System.out:

"move disk 1 from 1 to 3"

"move disk 2 from 1 to 2"

etcetera

	0:n=0,f=2,t=3		
1:n=1,f=1,t=3	2:n=1,f=1,t=3	2:n=1,f=1,t=3	
1:n=2,f=1,t=2	1:n=2,f=1,t=2	1:n=2,f=1,t=2	1:n=2,f=1,t=2
1:n=3,f=1,t=3	1:n=3,f=1,t=3	1:n=3,f=1,t=3	1:n=3,f=1,t=3

Hanoi with explicit run time stack

- The main loop of the program is:

while(rts not empty){
pop frame
check frame state
perform appropriate actions
}

// state x in code

state 0: initial state
nothing has been done

state 1: back from first "call"

state 2: back from second "call"

While loop:

Initially there is one Frame [state 0,n,from,to] on rts

Keep popping frames until rts is empty

When popping a frame:

if n == 0 do nothing (discard frame)

else if frame in state 0:

// do first call hanoi(n-1,from,via):

push [1,n,from,to] and push [0,n-1,from,via]

else if in state 1:

print disk n move

//do second call hanoi(0,n-1,via,to)

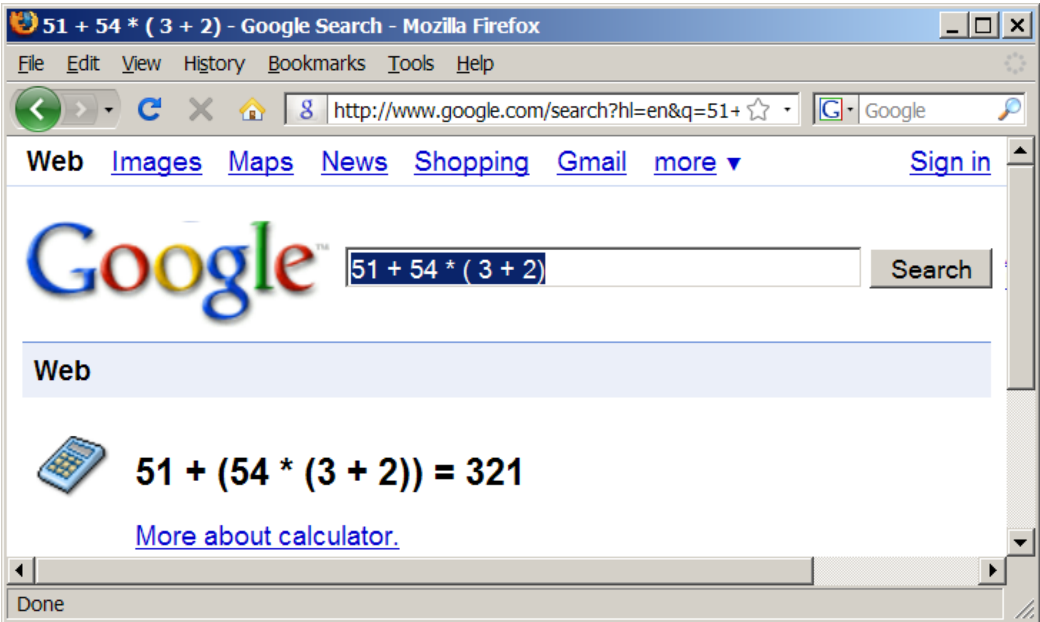
push [2,n,from,to] and push [0,n-1,via,to]

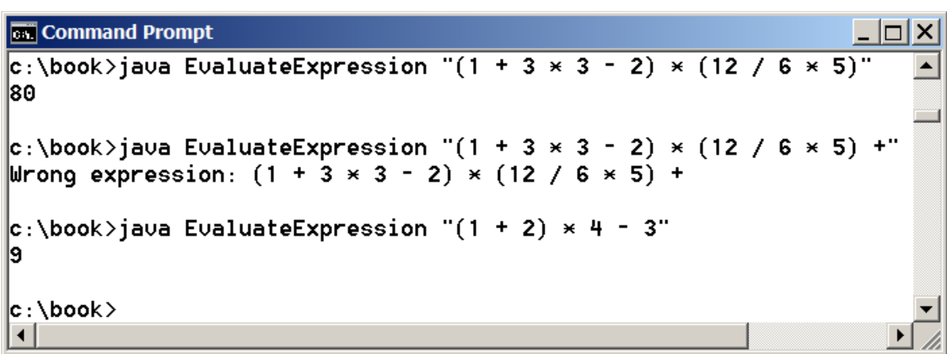
else (in state 2):

do nothing

Case Study: Evaluating Expressions

Stacks can be used to evaluate infix expressions.





Evaluate Expression

Run

Some examples

- 2 + 3

When we see + we haven't seen operand 3 yet. Use an operandStack to push operands, and an operatorStack to push operators:

push (2, operandStack)

push (+, operatorStack)

push (3, operandStack)

End of expression: apply operator to operands

Why wait until we see the end or rest of expression?

2+3*4

- 2 + 3 – 4 is (2+3) – 4, and NOT 2 + (3-4)

  push (2, operandStack)

  push (+, operatorStack)

  push (3, operandStack)

Seeing -: apply operator on stack to operands

            push(-, operatorStack)

  push(4, operandStack)

End: apply operator(s) to operands

- 2+3*4-5

  push (2, operandStack)

  push (+, operatorStack)

  push (3, operandStack)

 *: has precedence over +, so

  push (*, operatorStack)

  push (4, operandStack)

-: apply operators to operands,

  push (-, operatorStack)

5:push (5, operandStack)

End: apply operators to operands

- 2*(3+4)/5

  push (2, operandStack)

  push (*, operatorStack)

(: make a substack at top of operatorStack:

  push (‘(‘, operatorStack)

  push (3, operandStack)

  push (+, operatorStack)

  push (4, operandStack)

): apply operators to operands until ‘(’, pop (‘(’)

  push (/, operatorStack)

  push (5, operandStack)

End: apply operators to operands

Algorithm

Phase 1: Scanning the expression

The program scans the expression from left to right to extract operands, operators, and the parentheses.

1.1. If the extracted item is an operand, push it to **operandStack**.

1.2. If the extracted item is a + or **-** operator, process all the operators at the top of **operatorStack** and push the extracted operator to **operatorStack**.

1.3. If the extracted item is a * or / operator, process the * or / operators at the top of **operatorStack** and push the extracted operator to **operatorStack**.

1.4. If the extracted item is a **(** symbol, push it to **operatorStack**.

1.5. If the extracted item is a **)** symbol, repeatedly process the operators from the top of **operatorStack** until seeing the **(** symbol on the stack.

Phase 2: Clearing the stack

Repeatedly process the operators from the top of **operatorStack** until **operatorStack** is empty.

Example

Expression	*Scan*	*Action*	operandStack	operatorStack
(1 + 2)*4 − 3	(	Phase 1.4		(
(1 + 2)*4 − 3	1	Phase 1.1	1	(
(1 + 2)*4 − 3	+	Phase 1.2	1	+ (
(1 + 2)*4 − 3	2	Phase 1.1	2 1	(
(1 + 2)*4 − 3	)	Phase 1.5	3	
(1 + 2)*4 − 3	*	Phase 1.3	3	*
(1 + 2)*4 − 3	4	Phase 1.1	4 3	*
(1 + 2)*4 − 3	−	Phase 1.2	12	−
(1 + 2)*4 − 3	3	Phase 1.1	3 12	−
(1 + 2)*4 − 3	none	Phase 2	9	