Data Structures

Data Structures and Data Types

Data types
- Set of values.
- Set of operations on those values.
- Some are built in to Java: `int, double, char, . . .`
- Most are not: `Complex, Picture, Charge, Stack, Queue, Graph, . . .`

↑ this lecture (Stack)

Data structures.
- Represent data.
- Represent relationships among data.
- Some are built in to Java: arrays, `String`, . . .
- Most are not: linked list, circular list, tree, sparse array, graph, . . .

↑ this lecture (linked list) ↑ TSP (circular list) ↑ next lecture (tree)

Design challenge for every data type: What data structure to use?
- Requirement 1: Space usage.
- Requirement 2: Time usage for data-type methods

Collections

Fundamental data types.
- Set of operations (add, remove, test if empty) on generic data.
- Intent is clear when we insert.
- Which item do we remove?

Stack. (this lecture)
- Remove the item most recently added.
- Ex: cafeteria trays, Web surfing.

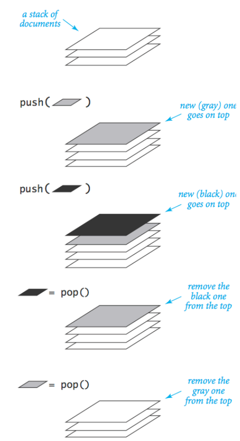
LIFO = "last in first out"

Queue. (see text)
- Remove the item least recently added.
- Ex: Registrar's line.

FIFO = "first in first out"

Symbol Table. (next lecture)
- Remove item with a given key.
- Ex: Phone book

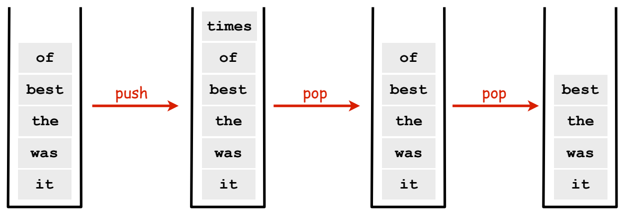
Stacks



Operations on a pushdown stack

Stack API

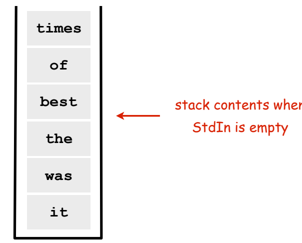
public class *StackOfStrings		
	*StackOfStrings()	*create an empty stack*
boolean	isEmpty()	*is the stack empty?*
void	push(String item)	*push a string onto the stack*
String	pop()	*pop the stack*



Stack Client Example 1: Reverse

```
public class Reverse
{
   public static void main(String[] args)
   {
      StackOfStrings stack = new StackOfStrings();
      while (!StdIn.isEmpty())
         stack.push(StdIn.readString());
      while (!stack.isEmpty())
         StdOut.print(stack.pop());
      StdOut.println();
   }
}
```

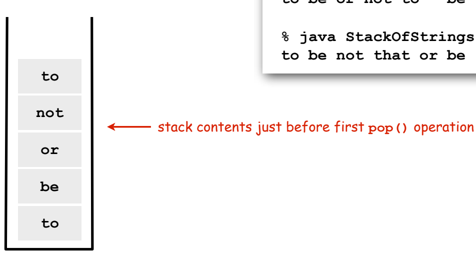
```
% more tiny.txt
it was the best of times

% java Reverse tiny.txt
times of best the was it
```



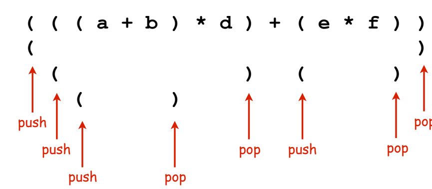
Stack Client Example 2: Test Client

```
public static void main(String[] args)
{
   StackOfStrings stack = new StackOfStrings();
   while (!StdIn.isEmpty())
   {
      String item = StdIn.readString();
      if (item.compareTo("-") != 0)
         stack.push(item);
      else
         System.out.print(stack.pop());
   }
   System.out.println();
}
```

```
% more test.txt
to be or not to - be - - that - - - is

% java StackOfStrings  < test.txt
to be not that or be
```



Stack Client Example 3: Balanced Parentheses



Stack Client Example 3: Balanced Parentheses

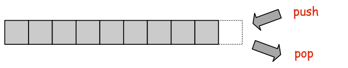
```
public class Balanced
{
   public static void main(String[] args)
   {
      StackOfStrings stack = new StackOfStrings();
      while (!StdIn.isEmpty())
      {
         String item = StdIn.readString();
         if (item.compareTo("(") == 0)
            stack.push(item);
         if (item.compareTo(")") == 0)
         {
            if (stack.isEmpty())
            {  StdOut.println("Not balanced"); return; }
            stack.pop();
         }
      }
      if (!stack.isEmpty()) StdOut.println("Not balanced");
      else                  StdOut.println("Balanced");
   }
}
```

```
% java Balanced
( ( ( a + b ) * d ) + ( e * f ) )
Balanced

% java Balanced
( ( a + b ) * d ) + ( e * f ) )
Not balanced
```


Stack: Array Implementation

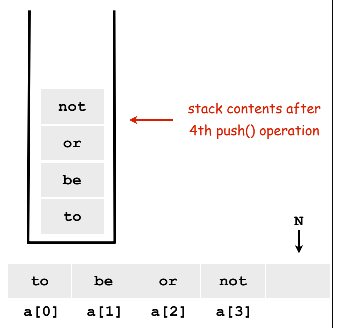
Array implementation of a stack.

- Use array `a[]` to store `N` items on stack. ← PROBLEM: How big to make array? (Stay tuned.)
- `push()` add new item at `a[N]`.
- `pop()` remove item from `a[N-1]`.



Temporary solution: Make client provide capacity.

```
public class ArrayStackOfStrings
{
   private String[] a;
   private int N = 0;

   public ArrayStackOfStrings(int max)
   {  a = new String[max];  }

   public boolean isEmpty()
   {  return (N == 0);  }

   public void push(String item)
   {  a[N++] = item;  }

   public String pop()
   {  return a[--N];  }
}
```



Array Stack: Trace

	StdIn	StdOut	N	a[] 0	1	2	3	4
			0					
push	to		1	to				
	be		2	to	be			
	or		3	to	be	or		
	not		4	to	be	or	not	
	to		5	to	be	or	not	to
pop	-	to	4	to	be	or	not	to
	be		5	to	be	or	not	be
	-	be	4	to	be	or	not	be
	-	not	3	to	be	or	not	be
	that		4	to	be	or	that	be
	-	that	3	to	be	or	that	be
	-	or	2	to	be	or	that	be
	-	be	1	to	be	or	that	be
	is		2	to	is	or	not	to

Stack Challenge: Stack Sort?

Q. Can we always insert pop commands (-) to make strings come out sorted?

Ex 1: 6 5 4 3 2 1 - - - - - -

Ex 2: 1 - 2 - 3 - 4 - 5 - 6 -

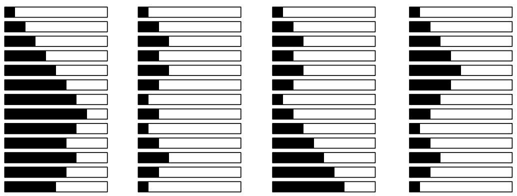
Ex 3: 4 1 - 3 2 - - - 6 5 - -

Array Stack: Performance

Running time. Push and pop take constant time. ✓

Memory. Proportional to client-supplied capacity, not number of items. ✗

What's the problem?
- API does not call for capacity (never good to change API)
- Client might have multiple stacks (stay tuned for an example)
- Client might not know what capacity to use (depends on its client)



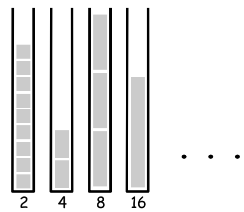
Challenge. Stack implementation where space use is not fixed ahead of time.

Example: potential stack client

Possible implementation of Java memory management system (sketch)

Maintain N stacks
- stack i: blocks of contiguous 2^i byte chunks of memory
- **new**: pop from stack t, where 2^t is smallest block that will hold new object
- stack t empty? pop from t+1, split in half, push 2 blocks on stack t
- garbage collector: periodically finds unused memory blocks ← How? See COS 226. and pushes onto appropriate stack.

Properties
- many stacks
- stack size unpredictable



Stack implementation without capacity restriction (as in API) is a requirement

Linked Lists

Sequential vs. Linked Data Structures

Sequential data structure. Put object one next to another.
- TOY: consecutive memory cells.
- Java: array of objects.

Linked data structure. Include in each object a link to the another one.
- TOY: link is memory address of next object.
- Java: link is reference to next object.

Key distinctions. (get i^{th} element)
- Array: arbitrary access, fixed size.
- Linked list: sequential access, variable size. (get next element)

Linked structures.
- Not intuitive, overlooked by naive programmers
- Flexible, widely used method for organizing data

addr	value
C0	"Alice"
C1	"Bob"
C2	"Carol"
C3	-
C4	-
C5	-
C6	-
C7	-
C8	-
C9	-
CA	-
CB	-

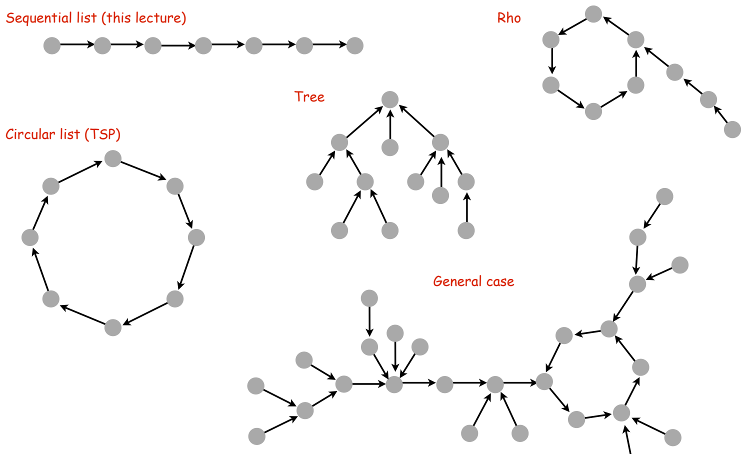
array

addr	value
C0	"Carol"
C1	null
C2	-
C3	-
C4	"Alice"
C5	CA
C6	-
C7	-
C8	-
C9	-
CA	"Bob"
CB	C0

linked list

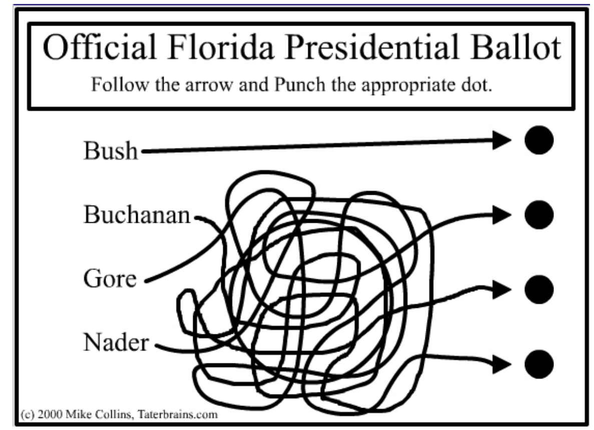
Singly-linked data structures

From the point of view of a particular object, all of these structures look the same:



Multiply linked structures: many more possibilities!

Linked structures can become intricate



Linked Lists

Linked list.

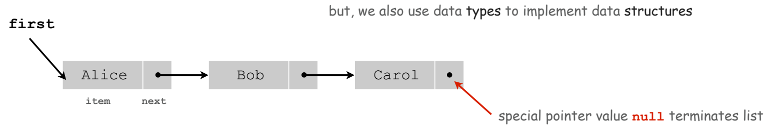
- Simplest linked structure.
- A recursive data structure.
- A item plus a pointer to another linked list (or empty list).
- Unwind recursion: linked list is a sequence of items.

Node data type.

- A reference to a `String`.
- A reference to another `Node`.

```
public class Node
{
   private String item;
   private Node next;
}
```

Confusing point:
Purpose of data **structure** is to represent data in a data **type**
but, we also use data **types** to implement data **structures**



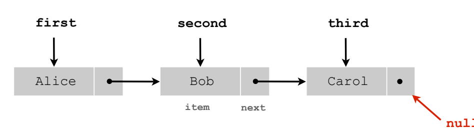
Building a Linked List

```
Node third  = new Node();
third.item  = "Carol";
third.next  = null;

Node second = new Node();
second.item = "Bob";
second.next = third;

Node first  = new Node();
first.item  = "Alice";
first.next  = second;
```

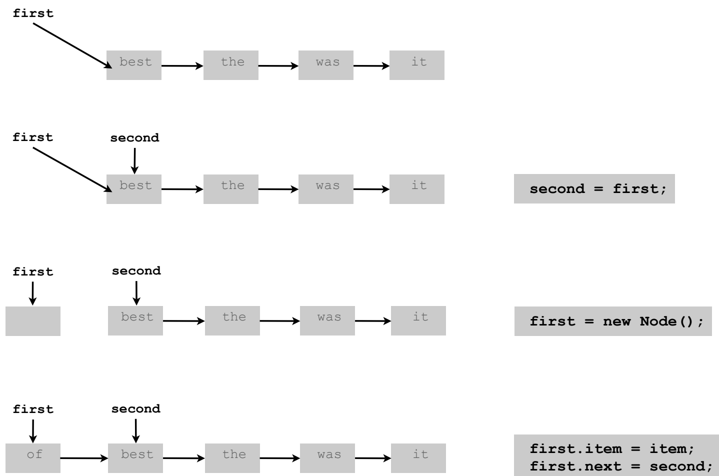
first	C4
second	CA
third	C0

addr	Value
C0	"Carol"
C1	null
C2	-
C3	-
C4	"Alice"
C5	CA
C6	-
C7	-
C8	-
C9	-
CA	"Bob"
CB	C0
CC	-
CD	-
CE	-
CF	-

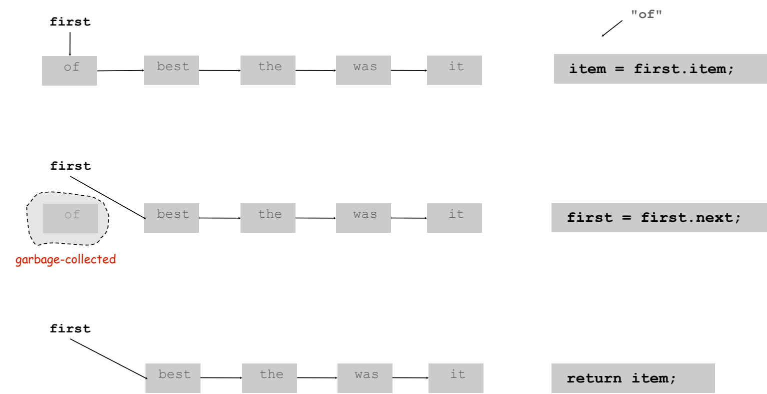
main memory



Stack Push: Linked List Implementation

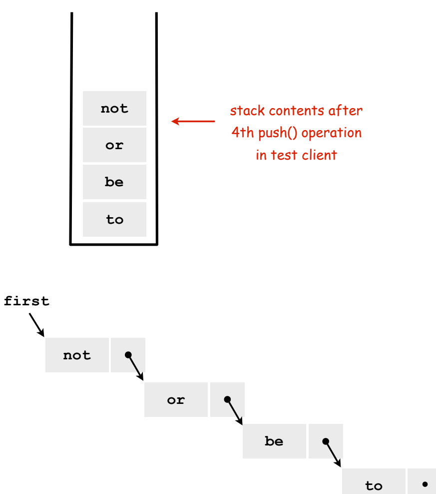


Stack Pop: Linked List Implementation



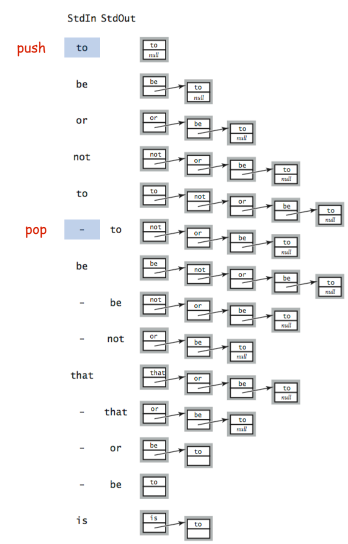
Stack: Linked List Implementation

```
public class LinkedStackOfStrings
{
   private Node first = null;

   private class Node
   {
      private String item;
      private Node next;
   }

   public boolean isEmpty()
   {  return first == null;  }

   public void push(String item)
   {
      Node second = first;
      first = new Node();
      first.item = item;
      first.next = second;
   }

   public String pop()
   {
      String item = first.item;
      first = first.next;
      return item;
   }
}
```



Note difference between `first` and `second`:
`first`: an instance variable that retains state
`second`: a local variable that goes out of scope

Linked List Stack: Trace



Linked-List Stack: Performance

Running time. Push and pop take constant time. ✓

Memory. Always proportional to number of items in stack. ✓

Stack Data Structures: Tradeoffs

Two data structures to implement the Stack data type.

Array.
- Every push/pop operation take constant time.
- But… must fix maximum capacity of stack ahead of time.

Linked list.
- Every push/pop operation takes constant time.
- But… uses extra space and time to deal with references.

Client can evaluate performance tradeoffs to choose among implementations
(implicitly choosing among underlying data structures)

List-Processing Challenge 1

What does the following code do?

```
...
Node list = null;
while (!StdIn.isEmpty())
{
   Node old = list;
   list = new Node();
   list.item = StdIn.readString();
   list.next = old;
}
for (Node t = list; t != null; t = t.next)
   StdOut.println(t.item);
...
```

A. Reverses the strings in StdIn.

Note: Better to use a Stack, represented as a linked list.
In this course, we always do list processing in data-type implementations.

List-Processing Challenge 2

What does the following code do?

```
...
Node list = new Node();
list.item = StdIn.readString();
Node last = list;
while (!StdIn.isEmpty())
{
   last.next = new Node();
   last = last.next;;
   last.item = StdIn.readString();
}
...
```

A. Puts the strings on StdIn in a linked list, in the order they are read
(assuming at least 1 string on StdIn).

Note: Could use a Queue, represented as a linked list (see text).
In this course, we always do list processing in data-type implementations.
This code might be the basis for an initializaion method in a data type.

Parameterized Data Types

Parameterized Data Types

We implemented: `StackOfStrings`.

We also want: `StackOfMemoryBlocks`, `StackOfURLs`, `StackOfInts`, …

Strawman. Implement a separate stack class for each type.
- Rewriting code is tedious and error-prone.
- Maintaining cut-and-pasted code is tedious and error-prone.

Generics

Generics. Parameterize stack by a single type.

"Stack of Apples"

parameterized type

```
Stack<Apple> stack = new Stack<Apple>();
Apple  a = new Apple();
Orange b = new Orange();
stack.push(a);
stack.push(b);    // compile-time error
a = stack.pop();
```

sample client

Can't push an "Orange" onto a "Stack of Apples"

Generic Stack: Linked List Implementation

String stack (for reference)

```
public class LinkedStackOfStrings
{
   private Node first = null;

   private class Node
   {
      private String item;
      private Node next;
   }

   public boolean isEmpty()
   {  return first == null;  }

   public void push(String item)
   {
      Node second = first;
      first = new Node();
      first.item = item;
      first.next = second;
   }

   public String pop()
   {
      String item = first.item;
      first = first.next;
      return item;
   }
}
```

```
public class Stack<Item>
{
   private Node first = null;

   private class Node
   {
      private Item item;
      private Node next;
   }

   public boolean isEmpty()
   {  return first == null;  }

   public void push(Item item)
   {
      Node second = first;
      first = new Node();
      first.item = item;
      first.next = second;
   }

   public Item pop()
   {
      Item item = first.item;
      first = first.next;
      return item;
   }
}
```

parameterized type name chosen by programmer

Autoboxing

Generic stack implementation. Only permits reference types.

Wrapper type.
- Each primitive type has a wrapper reference type.
- Ex: `Integer` is wrapper type for `int`.
- Wrapper type has larger set of operations than primitive type.
- Values of wrapper type are objects.

Autoboxing. Automatic cast from primitive type to wrapper type.
Autounboxing. Automatic cast from wrapper type to primitive type.

```
Stack<Integer> stack = new Stack<Integer>();
stack.push(17);        // Autobox     (int -> Integer)
int a = stack.pop();   // Auto-unbox (Integer -> int)
```


Stack Applications

Real world applications.
- Parsing in a compiler.
- Java virtual machine.
- Undo in a word processor.
- Back button in a Web browser.
- PostScript language for printers.
- Implementing function calls in a compiler.

Stack Client 4: Arithmetic Expression Evaluation

Goal. Evaluate infix expressions.

value stack
operator stack

(1 + ((2 + 3) * (4 * 5)))

operand   operator

Two stack algorithm. [E. W. Dijkstra]
- Value: push onto the value stack.
- Operator: push onto the operator stack.
- Left parens: ignore.
- Right parens: pop operator and two values; push the result of applying that operator to those values onto the operand stack.

Arithmetic Expression Evaluation

```
public class Evaluate
{
   public static void main(String[] args)
   {
      Stack<String> ops  = new Stack<String>();
      Stack<Double> vals = new Stack<Double>();
      while (!StdIn.isEmpty())
      {
         String s = StdIn.readString();
         if      (s.equals("("))              ;
         else if (s.equals("+"))    ops.push(s);
         else if (s.equals("*"))    ops.push(s);
         else if (s.equals(")"))
         {
            String op = ops.pop();
            if      (op.equals("+"))
               vals.push(vals.pop() + vals.pop());
            else if (op.equals("*"))
               vals.push(vals.pop() * vals.pop());
         }
         else vals.push(Double.parseDouble(s));
      }
      StdOut.println(vals.pop());
   }
}
```

```
% java Evaluate
( 1 + ( ( 2 + 3 ) * ( 4 * 5 ) ) )
101.0
```

Correctness

Why correct? When algorithm encounters an operator surrounded by two values within parentheses, it leaves the result on the value stack.

```
( 1 + ( ( 2 + 3 ) * ( 4 * 5 ) ) )
```

So it's as if the original input were:

```
( 1 + ( 5 * ( 4 * 5 ) ) )
```

Repeating the argument:

```
( 1 + ( 5 * 20 ) )
( 1 + 100 )
101
```

Extensions. More ops, precedence order, associativity, whitespace.

```
1 + (2 - 3 - 4) * 5 * sqrt(6*6 + 7*7)
```


Postfix

Observation 1. Remarkably, the 2-stack algorithm computes the same value if the operator occurs after the two values.

```
( 1 ( ( 2 3 + ) ( 4 5 * ) * ) + )
```

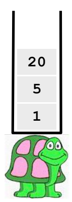
Observation 2. Now all of the parentheses are redundant!

```
1 2 3 + 4 5 * * +
```


Jan Lukasiewicz

Bottom line. Postfix or "reverse Polish" notation.

Real-World Stack Application: PostScript



PostScript (Warnock-Geschke, 1980s). A turtle with a stack.

- postfix program code
- add commands to drive virtual graphics machine
- add loops, conditionals, functions, types

PostScript code

```
100 100 moveto
100 300 lineto
300 300 lineto
300 100 lineto
stroke
```

units are points (72 per inch)

define a path

draw the path

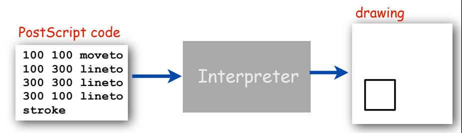
Simple virtual machine, but not a toy.

- Easy to specify published page.
- Easy to implement on various specific printers
- Revolutionized world of publishing.
- Virtually all printed material is PostScript.

Context/Definitions/Summary

Interpreter.

- Takes a program as input
- Does what that program would do.
- Simulates a virtual machine.



Compiler.

- Takes a program as input
- Produces a program as output.
- Produces code for a (real) machine.

TOY is our proxy for a real machine

Data Type and Virtual Machine are the same thing!

- Set of values = machine state.
- Operations on values = machine operations.

Virtual machines you have used

- LFSR
- Stack
- TOY
- PostScript
- Java Virtual Machine (another stack machine)

Data Structure.

- Represent data and relationships among data in a data type.
- array, linked list, compound, multiple links per node