CSCI 136:
Data Structures
and
Advanced Programming

Lecture 16

Linear structures

Instructor: Dan Barowy

Williams

Topics

- Linear ADTs
- Stack ADT

Your to-dos

1. Lab 6 (partner lab), **due Tuesday 11/1 by 10pm**. (two weeks!)
2. Read **before Fri**: Bailey, Ch 8-8.3.

Announcements

- Colloquium: **What I Did Last Summer (Industry)**, 2:35pm in Wege Auditorium with **cookies**.
- Midterm: **in lab** two weeks from now:
  Wed, October 26 and
  Thu, October 27 and
- Midterm review: **Mon, October 24 in class**.
- No class: **Fri, October 28**.

Radix sort

(a sort where stability matters)

Recall: Abstract Data Type

An **abstract data type** is a mathematical formulation of a data type. ADTs abstract away **accidental** properties of data structures (e.g., implementation details, programming language). Instead, ADTs contain only **essential** properties and are **concisely defined by their logical behavior** over a **set of values** and a **set of operations**.

In an ADT, precisely how data is **represented** on a computer **does not matter**.

By contrast: data structure

A **data structure** is the physical form of a data type, i.e., it is an implementation of an ADT. Generally, data structures are designed to efficiently support the logical operations described by the ADT.

For data structures, precisely how data is **represented** on a computer **matters a lot**. Simple data structures are often composed of simple representations, like primitives, while more complex data structures are composed of other data structures.

ADT example: List

A **list** is a sequential collection of data elements, whose order is not necessarily given by their placement in memory. Elements may store **any type of value**. A list supports **inserting**, **searching** for, and **deleting** any value in a list, **at any location**, although not necessarily efficiently.

Linear ADT

A **linear ADT** is one that presents elements **in a sequence**, even if the elements are **not actually stored that way**.

In a linear ADT, **adding** and **removing** elements is **constrained**, meaning that the structure can only be inspected and modified according to certain rules.

We will talk about two this week: **stack** and **queue**.

Stack ADT

A **stack** is an **abstract data type** that stores a collection of **any type of element**. A stack **restricts which elements are accessible**: elements may only be added and removed from the "**top**" of the collection. The "**push**" operation places an element onto the top of the stack while a "**pop**" operation removes an element from the top.

Stack ADT

Stack ADT

Also sometimes referred to as a **LIFO**: **"last in, first out."**

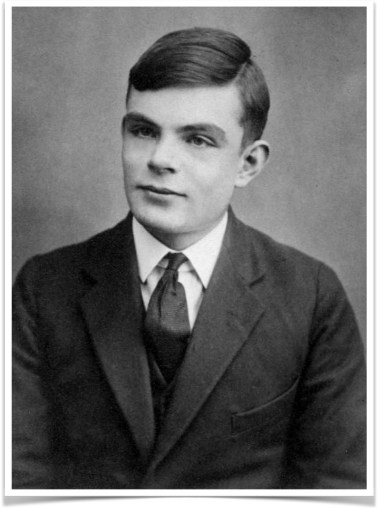
We also sometimes include a "**peek**" operation that lets us look at an element on the top of a stack without removing it, and "**size**" and "**isEmpty**" operations that let us check how many elements are stored and whether a stack stores zero elements, respectively.

Stack ADT

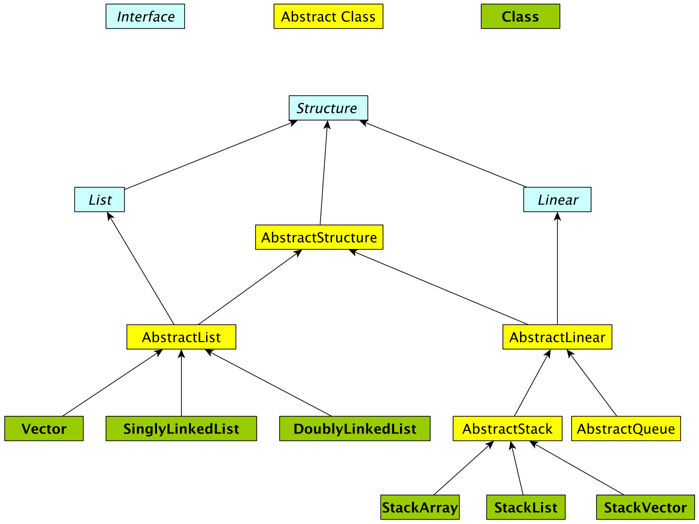
Interesting history: first appeared in print in a paper by Alan Turing (1946).

Unclear if he actually invented it.

push = **bury**,
pop = **unbury**.



structure5 Stack implementations



Interface
Abstract Class
Class
Structure
List
AbstractStructure
Linear
AbstractList
AbstractLinear
Vector
SinglyLinkedList
DoublyLinkedList
AbstractStack
AbstractQueue
StackArray
StackList
StackVector

Application: Arithmetic

A computer can perform arithmetic using a stack.

E.g., 1 + 2 * 3 = 7

Small problem: order of operations in infix arithmetic depends on the operations themselves.

In postfix arithmetic, order is always the same: left to right

E.g., 1 2 3 * + (note: fixed the confusing class example)

Once in this form, processing is easy. (Example)

Activity: Arithmetic

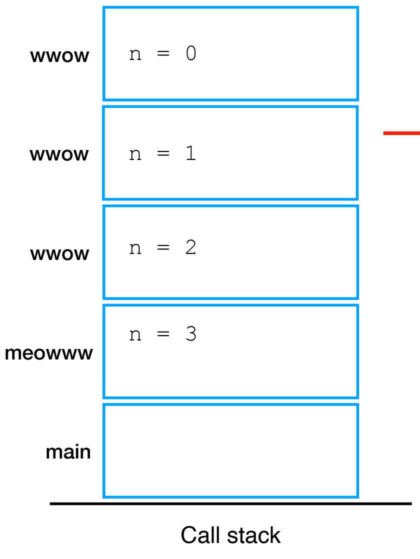
Convert infix to postfix: `x*y+z*w`

1. Add parens to preserve order of operations:
   `((x * y) + (z * w))`
2. Move all operators to the end of each parenthesized expression:
   `((x y *)(z w *) +)`
3. Remove parens:
   `x y * z w * +`

Evaluate these using a stack:

1. **`4 + 1 * 8`**
2. **`5 * (6 + 2) - 12 / 4`**

Cool application: function evaluation


wwow n = 0

wwow n = 1

wwow n = 2

meowww n = 3

main

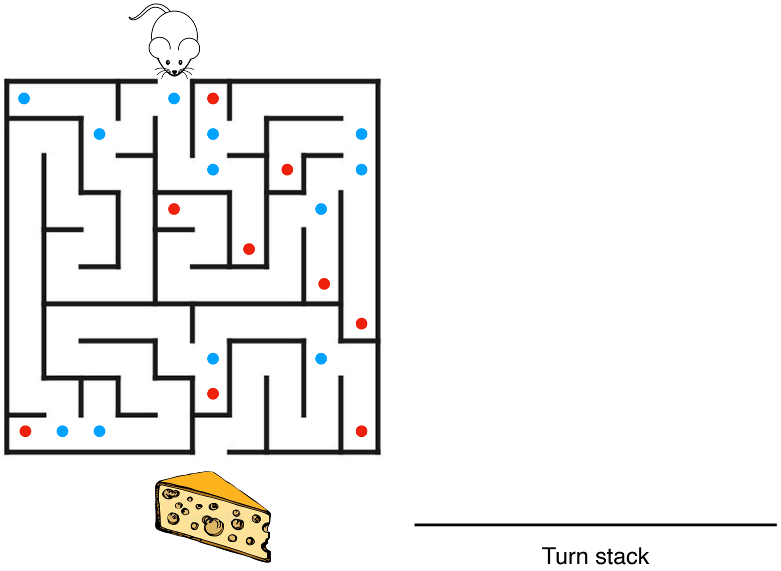
Call stack

```
class Meowww {
  public static String wwow(int n) {
    if (n == 0) {
      return "wow";
    }
    if (n == 1) {
      return "w";
    }
    return wwow(n-1) + wwow(n-2);
  }

  public static String meowww(int n) {
    return "meo" + wwow(n);
  }

  public static void main(String[] args) {
    int n = Integer.valueOf(args[0]);
    System.out.println(meowww(n));
  }
}
```

Cool application: backtracking search

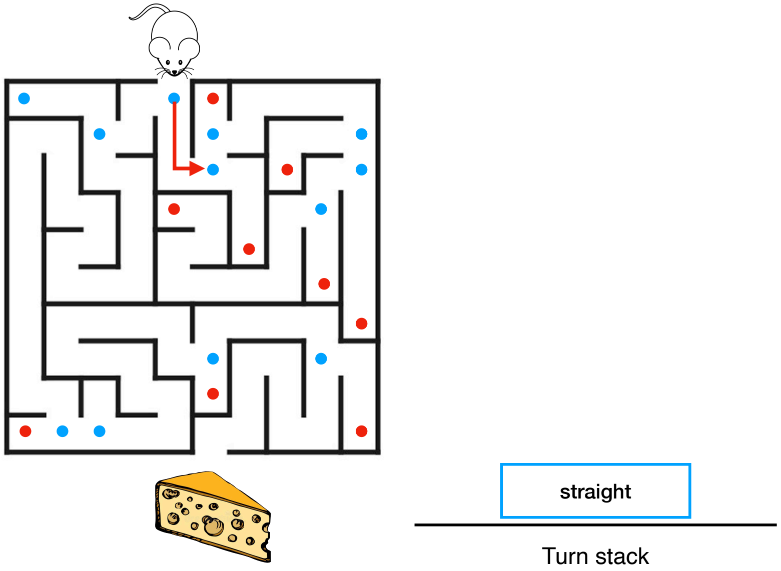
Search strategy: straight, left, right

Decision point: ● Dead end: ●


Turn stack

Search strategy: straight, left, right

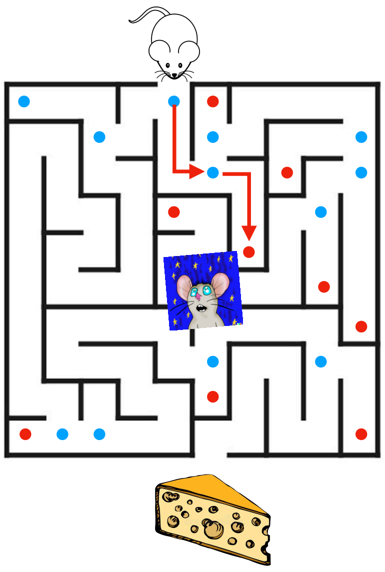
Decision point: ● Dead end: ●


straight

Turn stack

Search strategy: straight, left, right

Decision point: ● Dead end: ●



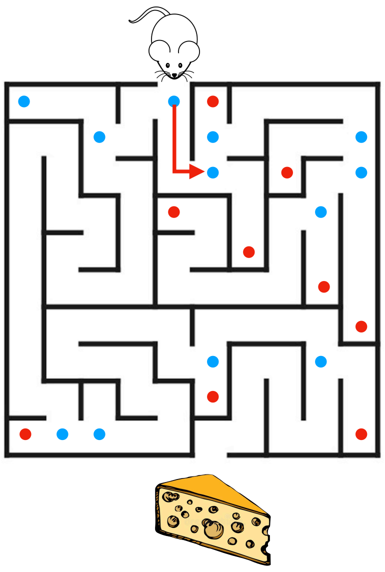
straight

straight

Turn stack

Search strategy: straight, left, right

Decision point: ● Dead end: ●

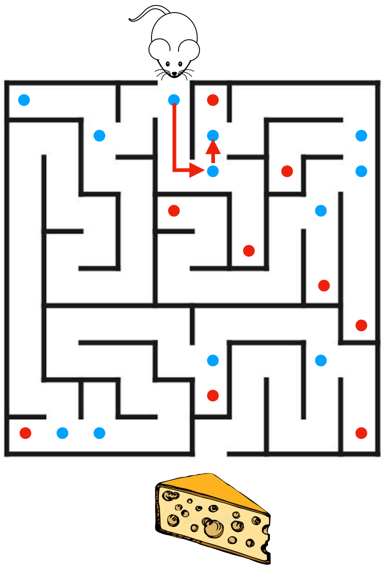


straight

Turn stack

Search strategy: straight, left, right

Decision point: ● Dead end: ●



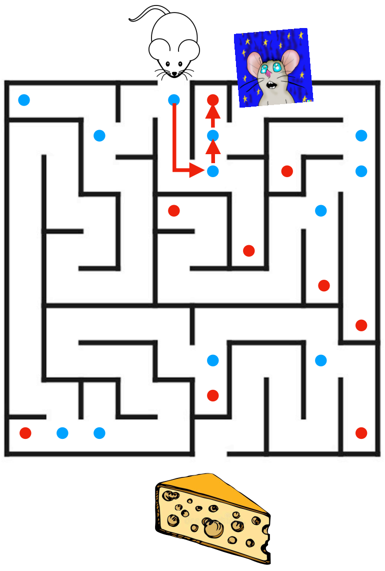
left

straight

Turn stack

Search strategy: straight, left, right

Decision point: ● Dead end: ●



straight

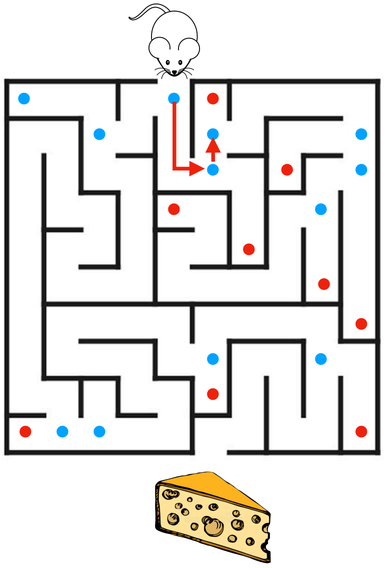
left

straight

Turn stack

Search strategy: straight, left, right

Decision point: ● Dead end: ●



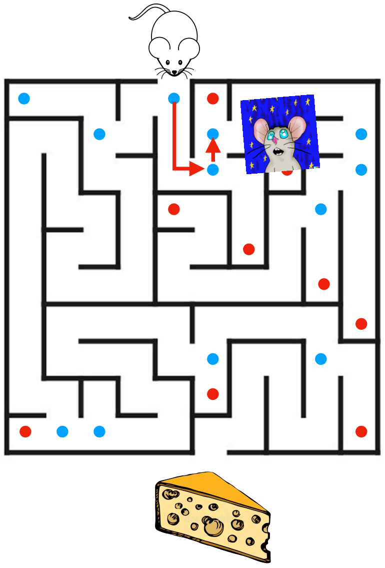
left

straight

Turn stack

Search strategy: straight, left, right

Decision point: ● Dead end: ●



left

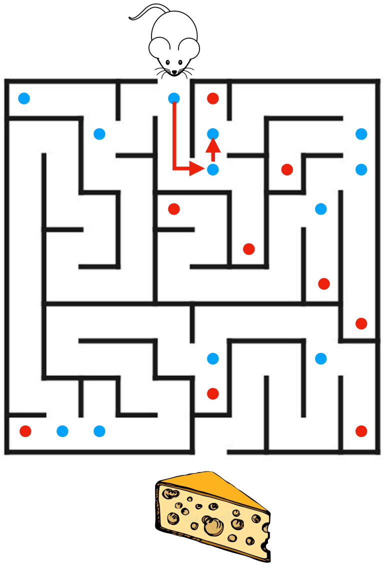
left

straight

Turn stack

Search strategy: straight, left, right

Decision point: ● Dead end: ●



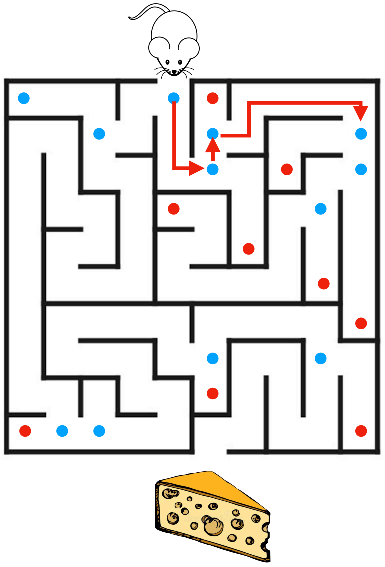
left

straight

Turn stack

Search strategy: straight, left, right

Decision point: ● Dead end: ●



right

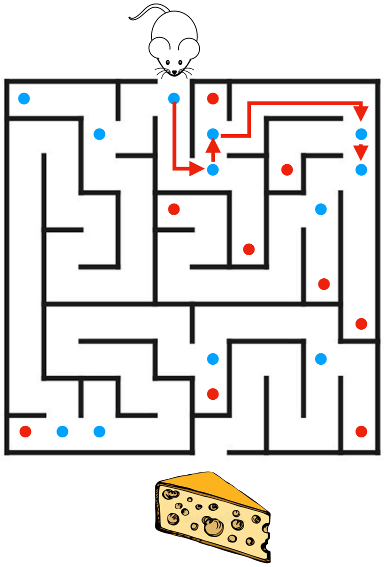
left

straight

Turn stack

Search strategy: straight, left, right

Decision point: ● Dead end: ●



straight

right

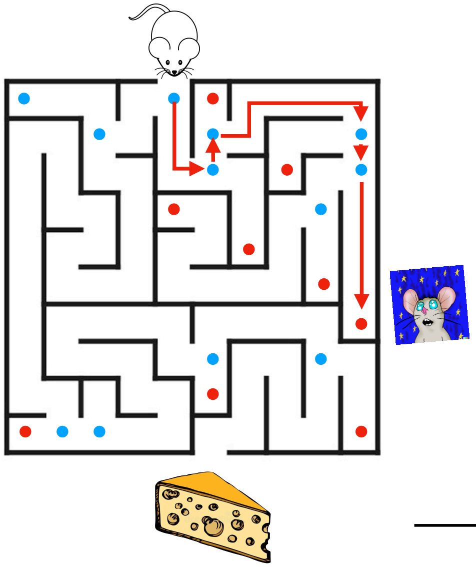
left

straight

Turn stack

Search strategy: straight, left, right

Decision point: ● Dead end: ●



straight

straight

right

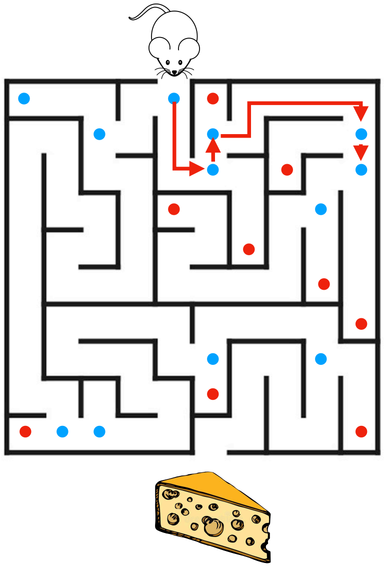
left

straight

Turn stack

Search strategy: straight, left, right

Decision point: ● Dead end: ●



straight

right

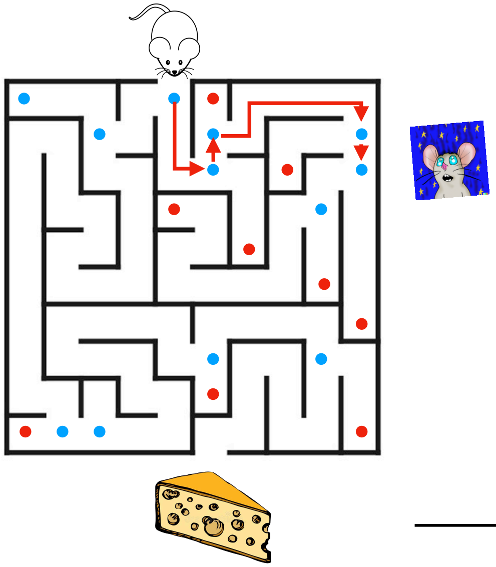
left

straight

Turn stack

Search strategy: straight, left, right

Decision point: ● Dead end: ●



left

straight

right

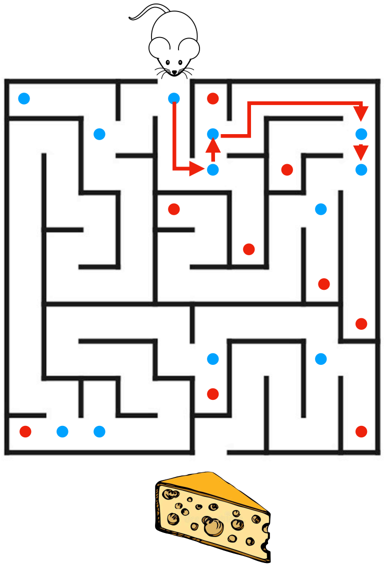
left

straight

Turn stack

Search strategy: straight, left, right

Decision point: ● Dead end: ●



straight

right

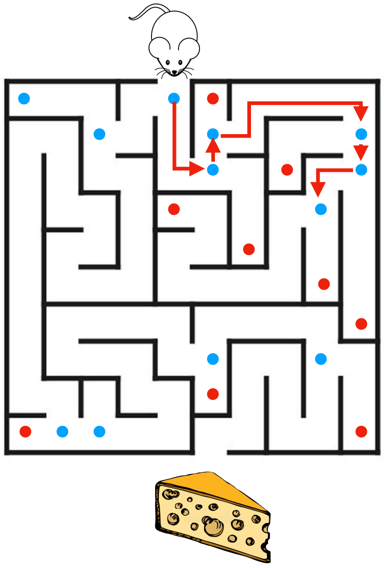
left

straight

Turn stack

Search strategy: straight, left, right

Decision point: ● Dead end: ●



right

straight

right

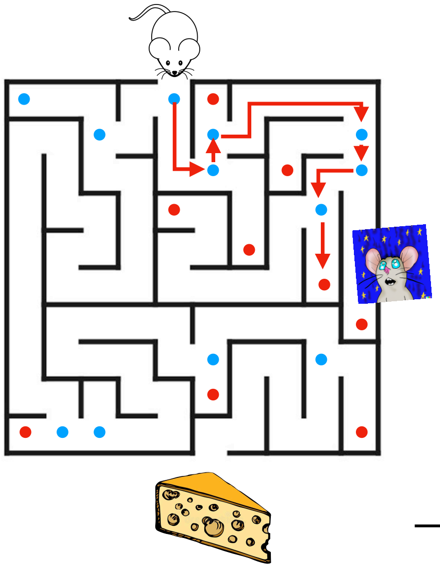
left

straight

Turn stack

Search strategy: straight, left, right

Decision point: ● Dead end: ●



straight

right

straight

right

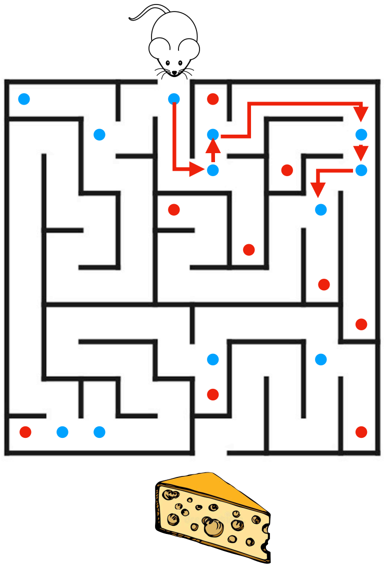
left

straight

Turn stack

Search strategy: straight, left, right

Decision point: ● Dead end: ●



right

straight

right

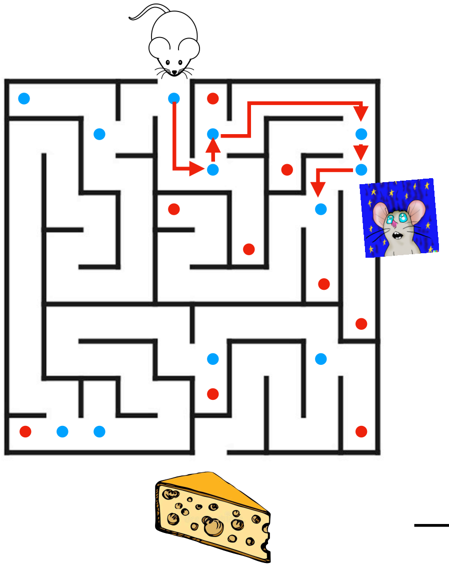
left

straight

Turn stack

Search strategy: straight, left, right

Decision point: 🔵 Dead end: 🔴



left

right

straight

right

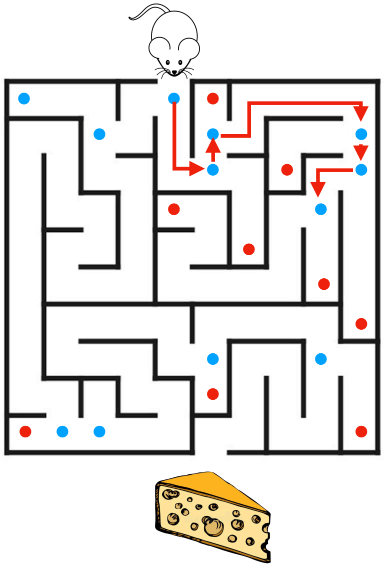
left

straight

Turn stack

Search strategy: straight, left, right

Decision point: 🔵 Dead end: 🔴



right

straight

right

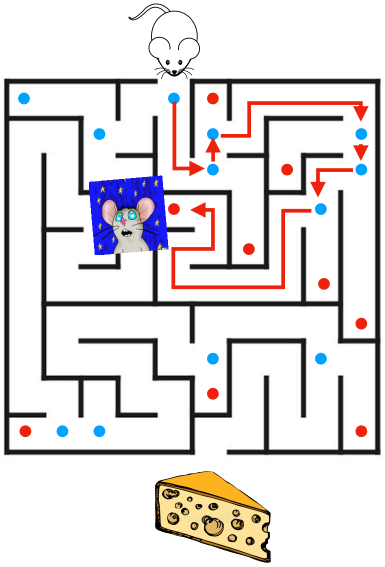
left

straight

Turn stack

Search strategy: straight, left, right

Decision point: 🔵 Dead end: 🔴



right

right

straight

right

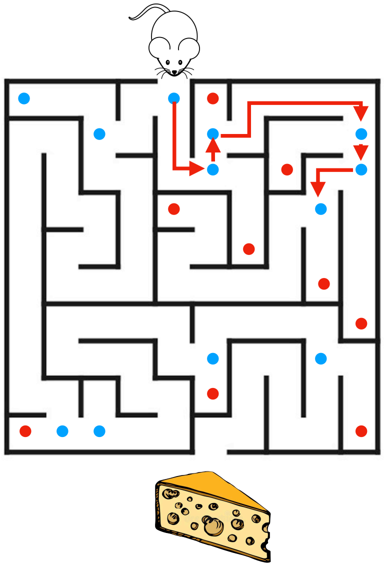
left

straight

Turn stack

Search strategy: straight, left, right

Decision point: 🔵 Dead end: 🔴



right

straight

right

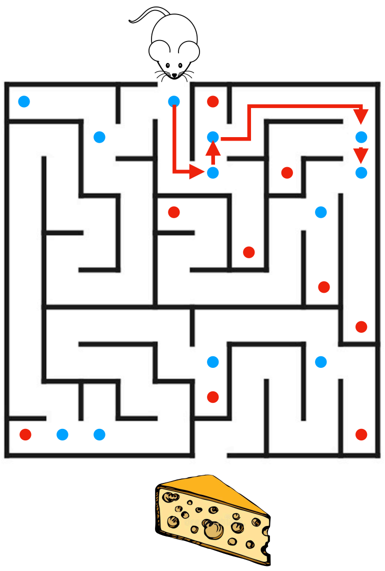
left

straight

Turn stack

Search strategy: straight, left, right

Decision point: ● Dead end: ●



straight

right

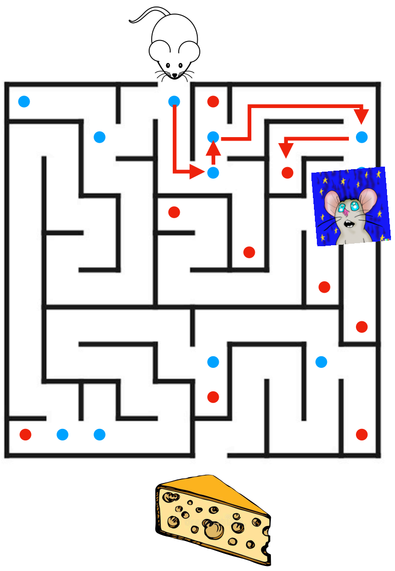
left

straight

Turn stack

Search strategy: straight, left, right

Decision point: ● Dead end: ●



right

right

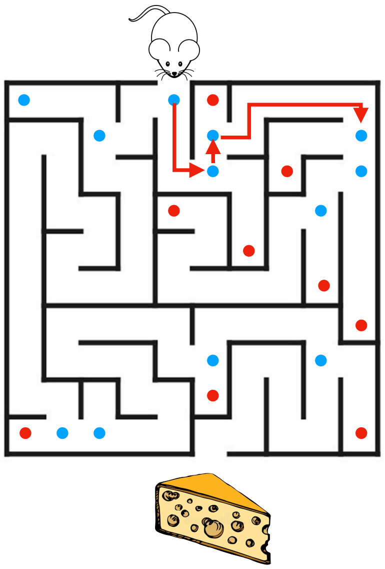
left

straight

Turn stack

Search strategy: straight, left, right

Decision point: ● Dead end: ●



right

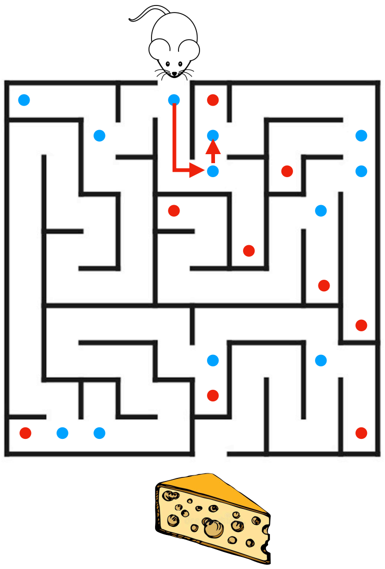
left

straight

Turn stack

Search strategy: straight, left, right

Decision point: ● Dead end: ●



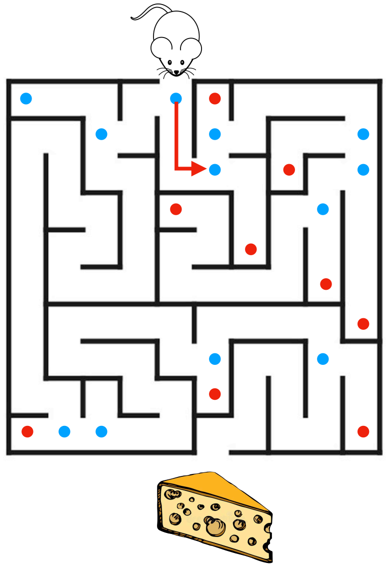
left

straight

Turn stack

Search strategy: straight, left, right

Decision point: ● Dead end: ●

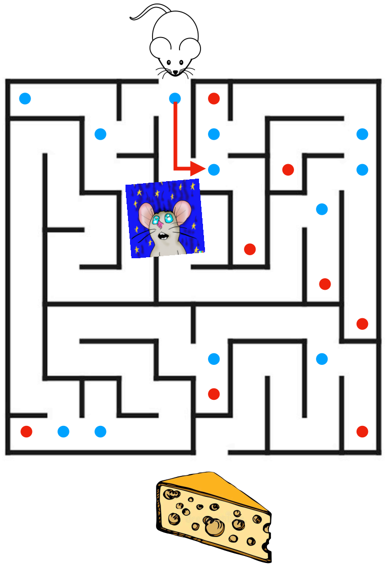


straight

Turn stack

Search strategy: straight, left, right

Decision point: ● Dead end: ●



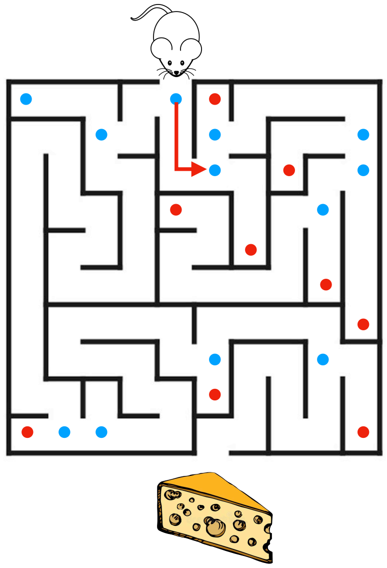
right

straight

Turn stack

Search strategy: straight, left, right

Decision point: ● Dead end: ●

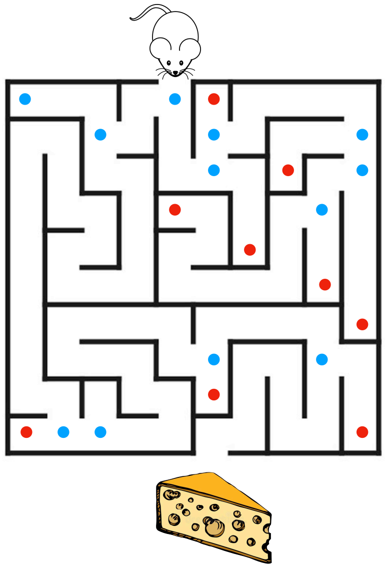


straight

Turn stack

Search strategy: straight, left, right

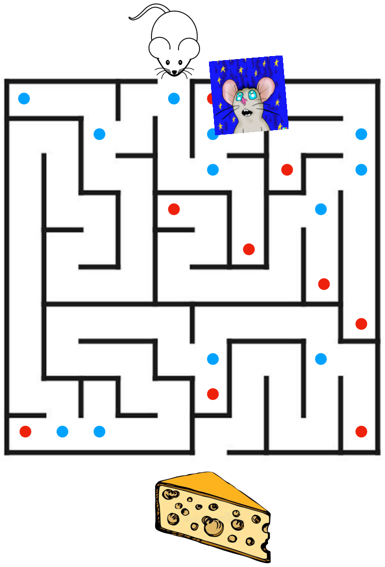
Decision point: ● Dead end: ●



Turn stack

Search strategy: straight, left, right

Decision point: ● Dead end: ●

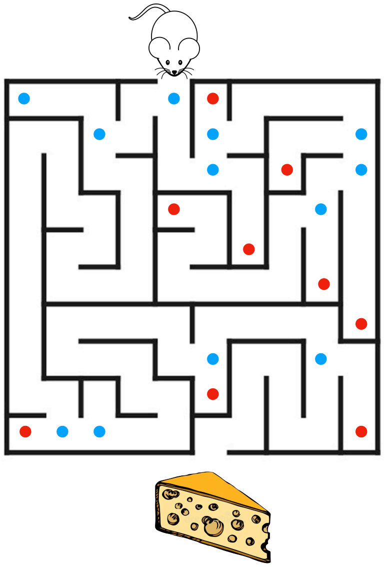


left

Turn stack

Search strategy: straight, left, right

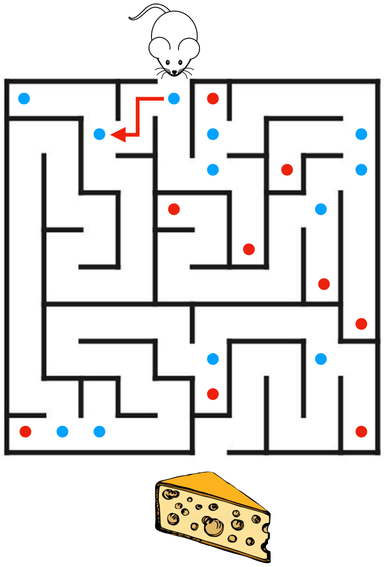
Decision point: ● Dead end: ●



Turn stack

Search strategy: straight, left, right

Decision point: ● Dead end: ●

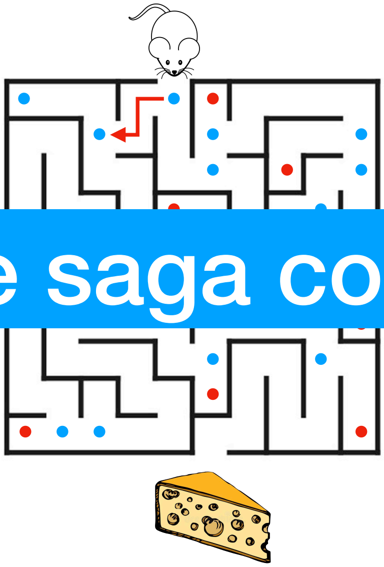


right

Turn stack

Search strategy: straight, left, right

Decision point: ● Dead end: ●



The saga continues…

right

Turn stack

Stack implementation

Stack data structures

StackArray

A **StackArray** is a stack implemented using an **array** for element storage.

Pros: **push** and **pop** are **O(1)** operations.

Cons: data structure has a maximum **capacity**.

Stack data structures

StackVector

A **StackVector** is a stack implemented using a **Vector** for element storage.

Pros: **push** and **pop** are amortized **O(1)** operations. There is no maximum capacity.

Cons: Most of the time, ops take **O(1)** time, but occasionally (when the underlying array needs to grow) an **O(n)** cost is incurred. This may be fine for most applications, but if the application cannot tolerate wide variation in time, this is a bad choice.

Also, unless the underlying array is completely full, Vectors **waste some space**.

Stack data structures

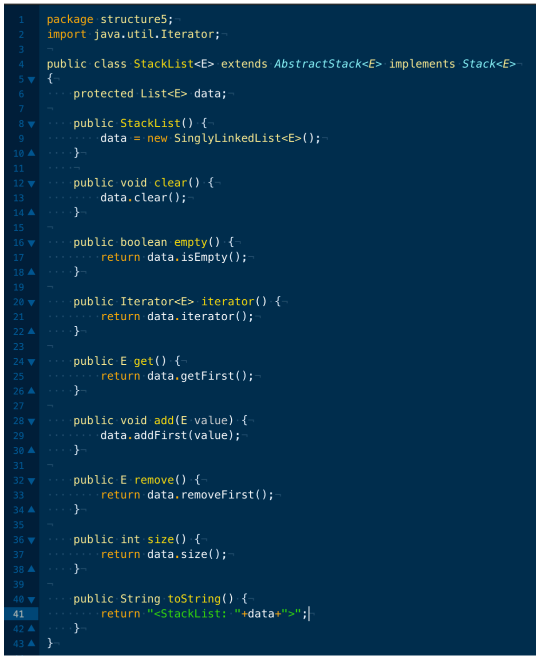
StackList

A **StackList** is a stack implemented using a **List** (usu. **SLL**) for element storage.

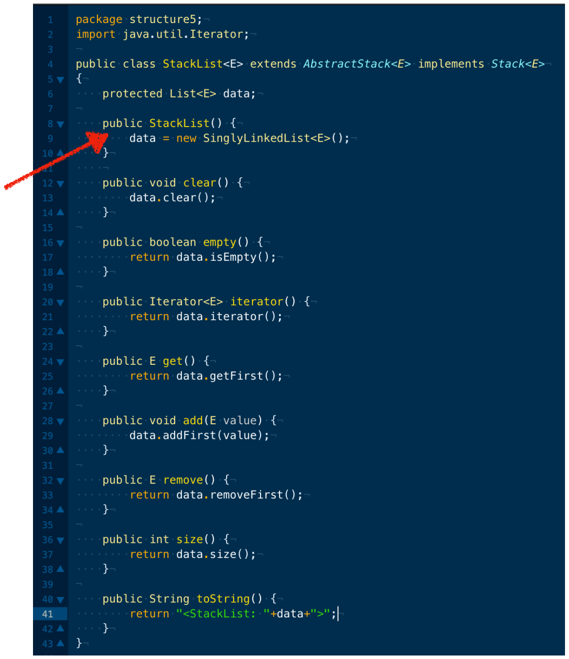
Pros: **push** and **pop** are **O(1)** operations. There is no maximum capacity, and no wasted space. **push** and **pop** costs are predictable (always the same), unlike **StackVector**.

Cons: because of the way computer hardware is implemented, a **StackList**'s constant-time cost is likely to be much higher than a **StackVector**'s. So a **StackList**'s performance may be **more predictable** than a **StackVector**, but it will likely be **slower on average**.

Let’s look at **StackList**

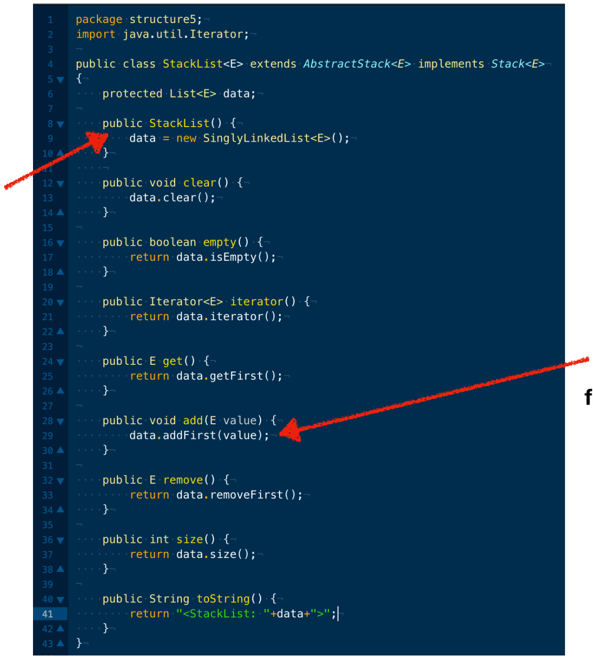


```
package structure5;
import java.util.Iterator;

public class StackList<E> extends AbstractStack<E> implements Stack<E>
{
    protected List<E> data;

    public StackList() {
        data = new SinglyLinkedList<E>();
    }

    public void clear() {
        data.clear();
    }

    public boolean empty() {
        return data.isEmpty();
    }

    public Iterator<E> iterator() {
        return data.iterator();
    }

    public E get() {
        return data.getFirst();
    }

    public void add(E value) {
        data.addFirst(value);
    }

    public E remove() {
        return data.removeFirst();
    }

    public int size() {
        return data.size();
    }

    public String toString() {
        return "<StackList: "+data+">";
    }
}
```



```
package structure5;
import java.util.Iterator;

public class StackList<E> extends AbstractStack<E> implements Stack<E>
{
    protected List<E> data;

    public StackList() {
        data = new SinglyLinkedList<E>();
    }

    public void clear() {
        data.clear();
    }

    public boolean empty() {
        return data.isEmpty();
    }

    public Iterator<E> iterator() {
        return data.iterator();
    }

    public E get() {
        return data.getFirst();
    }

    public void add(E value) {
        data.addFirst(value);
    }

    public E remove() {
        return data.removeFirst();
    }

    public int size() {
        return data.size();
    }

    public String toString() {
        return "<StackList: "+data+">";
    }
}
```

Uses an SLL for storage.

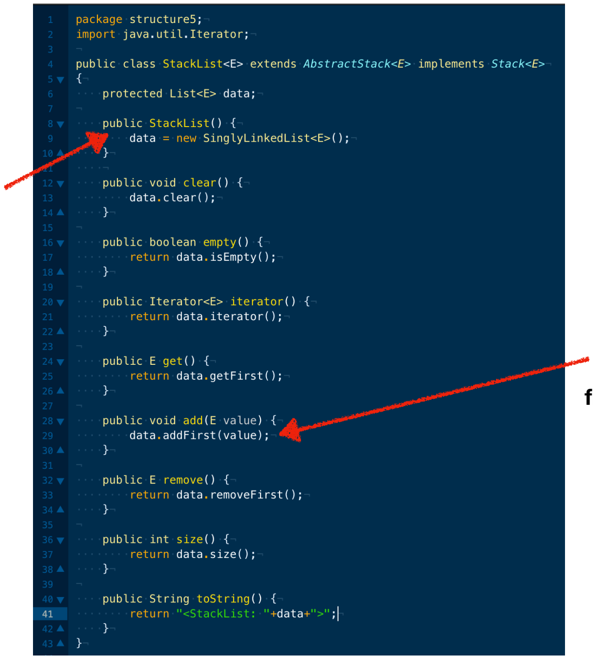


```
package structure5;
import java.util.Iterator;

public class StackList<E> extends AbstractStack<E> implements Stack<E>
{
    protected List<E> data;

    public StackList() {
        data = new SinglyLinkedList<E>();
    }

    public void clear() {
        data.clear();
    }

    public boolean empty() {
        return data.isEmpty();
    }

    public Iterator<E> iterator() {
        return data.iterator();
    }

    public E get() {
        return data.getFirst();
    }

    public void add(E value) {
        data.addFirst(value);
    }

    public E remove() {
        return data.removeFirst();
    }

    public int size() {
        return data.size();
    }

    public String toString() {
        return "<StackList: "+data+">";
    }
}
```

Uses an SLL for storage.

Adding an element puts it at the front of the list.

Uses an SLL for storage.

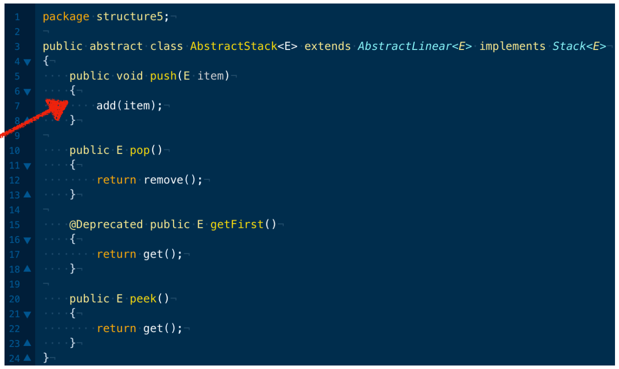


```
package structure5;
import java.util.Iterator;

public class StackList<E> extends AbstractStack<E> implements Stack<E>
{
    protected List<E> data;

    public StackList() {
        data = new SinglyLinkedList<E>();
    }

    public void clear() {
        data.clear();
    }

    public boolean empty() {
        return data.isEmpty();
    }

    public Iterator<E> iterator() {
        return data.iterator();
    }

    public E get() {
        return data.getFirst();
    }

    public void add(E value) {
        data.addFirst(value);
    }

    public E remove() {
        return data.removeFirst();
    }

    public int size() {
        return data.size();
    }

    public String toString() {
        return "<StackList: "+data+">";
    }
}
```

Adding an element puts it at the front of the list.

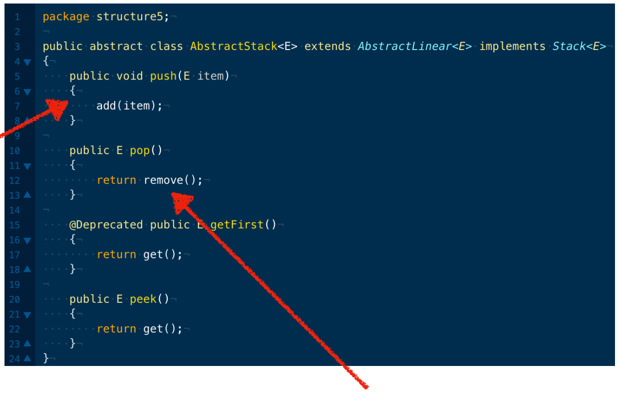
Wait! What about push?

push just calls add.



```
package structure5;

public abstract class AbstractStack<E> extends AbstractLinear<E> implements Stack<E>
{
    public void push(E item)
    {
        add(item);
    }

    public E pop()
    {
        return remove();
    }

    @Deprecated public E getFirst()
    {
        return get();
    }

    public E peek()
    {
        return get();
    }
}
```

push just calls add.

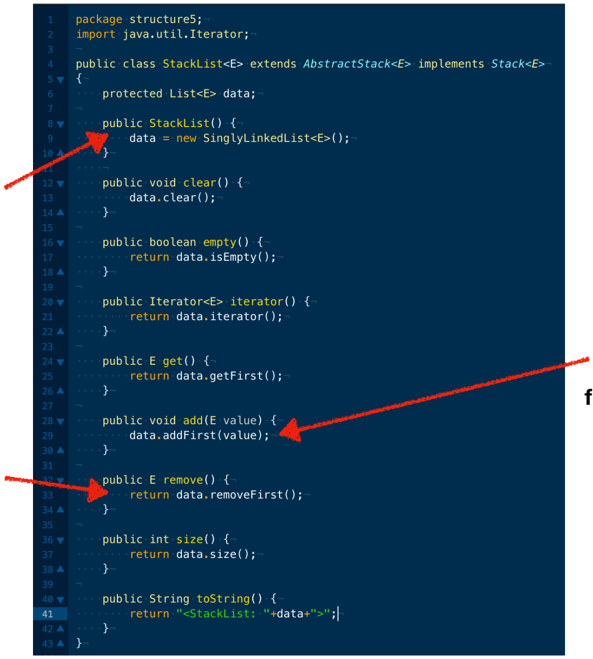


```
package structure5;

public abstract class AbstractStack<E> extends AbstractLinear<E> implements Stack<E>
{
    public void push(E item)
    {
        add(item);
    }

    public E pop()
    {
        return remove();
    }

    @Deprecated public E getFirst()
    {
        return get();
    }

    public E peek()
    {
        return get();
    }
}
```

pop just calls remove.

Uses an SLL for storage.

Removing an element removes the first element in the list.



```
package structure5;
import java.util.Iterator;

public class StackList<E> extends AbstractStack<E> implements Stack<E>
{
    protected List<E> data;

    public StackList() {
        data = new SinglyLinkedList<E>();
    }

    public void clear() {
        data.clear();
    }

    public boolean empty() {
        return data.isEmpty();
    }

    public Iterator<E> iterator() {
        return data.iterator();
    }

    public E get() {
        return data.getFirst();
    }

    public void add(E value) {
        data.addFirst(value);
    }

    public E remove() {
        return data.removeFirst();
    }

    public int size() {
        return data.size();
    }

    public String toString() {
        return "<StackList: "+data+">";
    }
}
```

Adding an element puts it at the front of the list.

Recap & Next Class

Today:

Linear ADTs

Stack

Next class:

Queue, etc.