- intro to stacks
- function call and return example
- reversing digits example
- stack exercise

Next week and homework

- next week: dynamic implementation of stack
- homework by next week:
  - download, run and understand this week's example programs
  - read Horstmann, section 16.3 on stacks; Chapter 18 Generic Classes

Lab

- homework assigned this week. See Canvas for due date

Review last week

- introduced the reference or pointer, so that we can build dynamic data structures
- began reading our excellent course textbook:
  - Horstmann, Cay. Big Java Early Objects, 7th edition, Wiley, 2018, ISBN 978-1-119-49909-1. (Paper or eTextbook formats are available.)
  - (note that this is the same textbook used for the CSCI 114 prereq class)
  - you were told to read section 15.5 on stacks before this week, to prepare for the new content
- reviewed BlueJ, required for writing all our Java programs this semester
  - homework on this was assigned, due by end of last week
- reviewed how to write Javadoc comments
  - an essential part of program clarity, required in all programming assignments to earn full credit

Introduction to this week

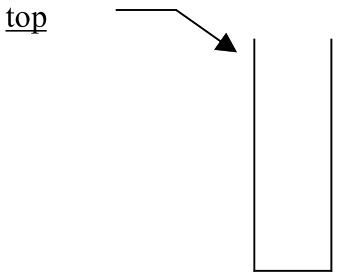
- this week we begin our tour of the standard set of data structures
- next several weeks with stacks
- start this week with stacks in abstract
- will introduce the characteristics and primitive operators for a stack
- review some example applications
- will do an array implementation of stack for homework

Intro to stacks

Objective: introduce abstract idea of a stack and its primitive operations. Introduce implementation using an array

Items may be pushed and popped only at the top

- "ordered collection of items such that items may be added (pushed) and removed (popped) only at the top" e.g.



1. add A, B, C, D at the top

2. remove 2 items from top

3. add E, F at the top

Figure 1 push and pop a stack of characters

  - Q: what will be the contents of the stack, top to bottom?

  - A: F, E, B, A

- is Last In First Out (LIFO) data structure

  - very different to First In First Out – i.e. queue at the bank

Abstract idea vs. implementation

- the abstract idea of a stack is clearly as a dynamic data structure

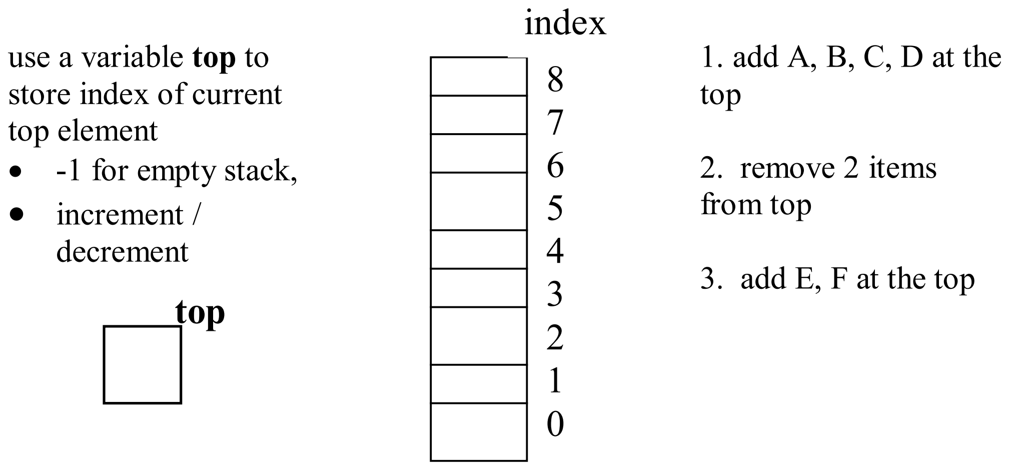
  - grows and shrinks over time

- implementation can be static or dynamic

  - will consider the simpler static implementation first, because it's more familiar to us

  - i.e. stack implemented as an array

  - (NOTE: assume that every stack item is of the same data type)



use a variable **top** to store index of current top element

- -1 for empty stack,
- increment / decrement

1. add A, B, C, D at the top

2. remove 2 items from top

3. add E, F at the top

Figure 2 static implementation of the stack as an array

- (BTW, note that while the abstraction of stack can be accessed at the top only
  - the array implementation can actually be accessed at any element)

Begin the array implementation

- would package these together as the instance variables of a new `Stack` class e.g.

```
public class Stack {
    public static final int MAX = 9;

    private int element[];
    private int top;
    …
}
```

  - notice here that the data type stored inside this stack is actually `int`

- then the constructor to initialize a new empty stack:

```
public Stack()
{
    element = new int[MAX];
    top = -1;  //stack starts empty
}
```

Stack primitives – "the set of operations that act on a data structure"

- set of operations to act on the data structure. Here, for a Stack object s:

- s.push(i) – adds item i to top of stack s
- i = s.pop() – removes top item from stack s

Some problems to watch for:

- pop() an empty stack
  - is called underflow
  - implementation must handle this in some appropriate way
  - all we will do is output an error message to standard error
- push() to a full stack
  - called overflow
  - (is an implementation issue here, because the array has a fixed, limited size `MAX`)

Other useful stack primitives

- could add some other useful stack operations, e.g.
- s.isEmpty() – true if s is empty e.g. to pop everything off the stack:

```
while (!s.empty())
    i = s.pop()
```

- s.isFull() – true if s is full
- i = s.top() – returns a copy of top item on stack without modifying the stack
  - so s.top() could be implemented as:

```
i = s.pop()
s.push(i)
```

- s.clear() – removes all elements from stack
  - could be:

```
while (!s.empty())
    i = s.pop()
```

Summary

- stack is a 'Last In First Out' (LIFO) data structure
- the most important primitive operations are:
  - push()
  - pop()
- can be implemented statically or dynamically
  - will do an array implementation for homework this week
- use a stack when a problem has 'LIFO characteristics'. This is quite common e.g.
  - to reverse something
  - whenever backtracking is required – "to return to a previously encountered state"
  - will look at some stack example applications demonstrating these…

Function call and return example

Objective: show a LIFO kind of problem, where a stack is used to implement call and return

- a stack is used to implement function call and return in procedural programming languages e.g.

```
main()          foo()          bar()          wah()
{               {              {              {
   ~~~             ~~~            ~~~            ~~~
   ~~~             ~~~            ~~~            ~~~
@foo()          #bar()         $wah()            ~~~
   ~~~             ~~~            ~~~            ~~~
   ~~~             ~~~            ~~~         }
}               }              }
```

Figure 3 method calls from main(), with each return address marked

  - here are some method calls from main(), with each return address marked

- when the first method call to foo() is executed, we push the return address of the instruction we come back to onto the return address stack, and so on

  - e.g. here's the stack when we're in wah():

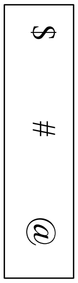


Figure 4 return addresses pushed to the stack

  - then we pop the stack each time a method ends, returning correctly to the method call

  - (BTW, all the other stuff local to each calling method is also pushed to the stack each method call)

- at end of main(), stack is empty…

- means that the program has ended
- neat!

Summary

- a stack application, used in almost every programming language
- (also a good preparation for recursion, coming soon)

Reversing digits example

Objective: introduce next week's stack exercise

- stacks are used when we need to reverse something. For example, reverse the order of digits in an integer
  - – e.g. for the integer 12345
  - – use a stack to reverse the order of digits, to give 54321

<u>Use a stack to reverse digits</u>

- will actually represent the integer as a string of characters, to avoid overflow problems. So we need a stack of datatype `Character` to do the reverse:

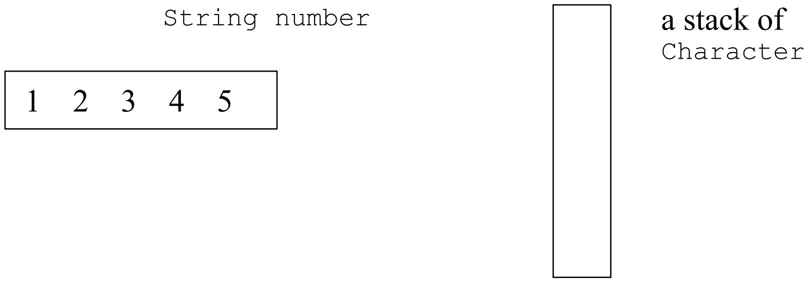


Figure 5 use a string and a stack of characters

- reversing algorithm in pseudocode is something like:

```
loop for the digits in the number
   push digit to stack
while (!stack is empty)
        pop stack
```

- so reversing the digits would look something like:

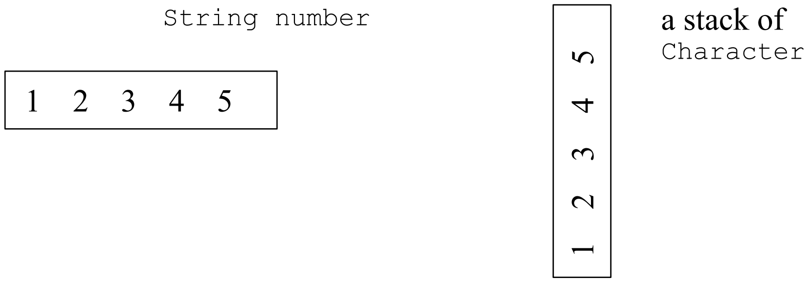


Figure 6 reversing, with all the digits pushed to the stack

- all the digits have been pushed to the stack

- now pop the stack, to get digits in reverse order

Summary

- you will write this reverse using a stack next week

Stack exercise

Objective: a quick exercise to get you working with stacks

- reverse an array of 10 integers using a stack:

a

3	7	4	6	2	0	9	1	5	8

Figure 7 example array to be reversed

  - do this now, pencil and paper, pseudocode, 10 minutes, keep it simple

Review

- should be something like:

```
create empty stack s
for (i = 0; i < 10: ++i)
    s.push(a[i]);
for (i = 0; i < 10: ++i)
    a[i] = s.pop();
```

Summary

- a common use of stacks is to reverse something

Next week and homework

- next week: dynamic implementation of stack
- homework by next week:
  - download, run and understand this week's example programs
  - read Horstmann, section 16.3 on stacks; Chapter 18 Generic Classes

Lab

- homework assigned this week. See Canvas for due date