Stacks

Gaddis 18.1, 18.3

Molly A. O'Neil
CS 2308 :: Spring 2016
The Stack ADT

- A stack is an abstract data type that stores a collection of elements of the same type
  - The elements of a stack are accessed in LIFO (last in, first out) order
  - No random access to non-first elements
    - You can only access or remove the most-recently-added element

- Analogy: A stack of plates or trays in a cafeteria. Clean plates/trays are put on top of the stack, and the next person takes the plate/try on top of the stack.
Stack Operations

• The stack ADT supports the following operations:
  • **push**: add a value onto the top of the stack
    • If it's a size-limited stack, first make sure the stack isn't full
  • **pop**: remove the value at the top of the stack
    • Depending on the implementation, sometimes pop returns the value as well
  • **isEmpty**: returns true if the stack contains no elements
  • **isFull**: if the stack is size-limited, returns true if the stack has no more space to hold additional elements
Stack Illustrated

```javascript
stack.push(3)
stack.push(2)
stack.push(5)
stack.push(1)
x = stack.pop()
```
Stack Illustrated

```python
stack.push(3)
stack.push(2)
stack.push(5)
stack.push(1)
x = stack.pop()
```
Stack Illustrated

```python
stack.push(3)
stack.push(2)
stack.push(5)
stack.push(1)
x = stack.pop()
y = stack.pop()
```
Stack Illustrated

```
stack.push(3)
stack.push(2)
stack.push(5)
stack.push(1)
x = stack.pop()
y = stack.pop()
```
Stack Illustrated

```python
stack.push(3)
stack.push(2)
stack.push(5)
stack.push(1)
x = stack.pop()
y = stack.pop()
stack.push(6)
stack.push(4)
z = stack.pop()
```
```python
stack.push(3)
stack.push(2)
stack.push(5)
stack.push(1)
x = stack.pop()
y = stack.pop()
stack.push(6)
stack.push(4)
z = stack.pop()
```
Stack Illustrated

```python
stack.push(3)
stack.push(2)
stack.push(5)
stack.push(1)
x = stack.pop()
stack.push(6)
y = stack.pop()
stack.push(4)
z = stack.pop()
```

![Stack Diagram]

3 2 6 4

3 2 6
Stack Applications

- Syntax parsing (e.g., bracket matching, cf. Project 7)
  - Compilers have to do this

- Evaluating arithmetic expressions in post-fix notation
  - $4 \ 5 \ + \ 7 \ 2 \ - \ *$ is equivalent to $(4 + 5) \ * \ (7 - 2)$

- Algorithm:
  - Parse input from L to R
  - As you encounter operands, push them onto the stack
  - If you encounter an operator, pop two operands from the stack and apply the operation, then push the result onto the stack
  - Keep going until no more input
Stack Applications

• Syntax parsing (e.g., bracket matching, cf. Project 7)
  • Compilers have to do this

• Evaluating arithmetic expressions in post-fix notation
  • \( 4 \ 5 \ + \ 7 \ 2 \ - \ * \) is equivalent to \( (4 + 5) \ * \ (7 - 2) \)
  • Algorithm:
    • Parse input from L to R
    • As you encounter operands, push them onto the stack
    • If you encounter an operator, pop two operands from the stack and apply the operation, then push the result onto the stack
    • Keep going until no more input
Stack Applications

- Syntax parsing (e.g., bracket matching, cf. Project 7)
  - Compilers have to do this

- Evaluating arithmetic expressions in post-fix notation
  - \(4 5 + 7 2 - *\) is equivalent to \((4 + 5) * (7 - 2)\)

- Algorithm:
  - Parse input from L to R
  - As you encounter operands, push them onto the stack
  - If you encounter an operator, pop two operands from the stack and apply the operation, then push the result onto the stack
  - Keep going until no more input
Stack Applications

• Syntax parsing (e.g., bracket matching, cf. Project 7)
  • Compilers have to do this

• Evaluating arithmetic expressions in post-fix notation
  • \[45 + 72 - *\] is equivalent to \[(4 + 5) * (7 - 2)\]
  • Algorithm:
    • Parse input from L to R
    • As you encounter operands, push them onto the stack
    • If you encounter an operator, pop two operands from the stack and apply the operation, then push the result onto the stack
    • Keep going until no more input

\[7 - 2 = 5\]
Stack Applications

• Syntax parsing (e.g., bracket matching, cf. Project 7)
  • Compilers have to do this

• Evaluating arithmetic expressions in post-fix notation
  • \(4 5 + 7 2 - *\) is equivalent to \((4 + 5) \times (7 - 2)\)

• Algorithm:
  • Parse input from L to R
  • As you encounter operands, push them onto the stack
  • If you encounter an operator, pop two operands from the stack and apply the operation, then push the result onto the stack
  • Keep going until no more input
Stack Applications

- Syntax parsing (e.g., bracket matching, cf. Project 7)
  - Compilers have to do this
- Evaluating arithmetic expressions in post-fix notation
  - \[ 4 \ 5 \ + \ 7 \ 2 \ - \ * \] is equivalent to \((4 + 5) \times (7 - 2)\)
- Algorithm:
  - Parse input from L to R
  - As you encounter operands, push them onto the stack
  - If you encounter an operator, pop two operands from the stack and apply the operation, then push the result onto the stack
  - Keep going until no more input

\[ 9 \ \ast \ 5 = 45 \]
Stack Applications

• Syntax parsing (e.g., bracket matching, cf. Project 7)
  • Compilers have to do this

• Evaluating arithmetic expressions in post-fix notation
  • $4 \ 5 \ + \ 7 \ 2 \ - \ *$ is equivalent to $(4 + 5) \ * \ (7 - 2)$

• Algorithm:
  • Parse input from L to R
  • As you encounter operands, push them onto the stack
  • If you encounter an operator, pop two operands from the stack and apply the operation, then push the result onto the stack
  • Keep going until no more input

Precedence rules and parentheses are never needed!
Stack Applications

• "Undo" feature in text editors
  • Recent text changes are kept in a stack

• Backtracking
  • Imagine you must write code to guide a robot through a maze or labyrinth
  • Every time you reach a decision point, choose a direction you haven't explored yet. If you hit a dead end, you must backtrack...
  • ...but backtrack to where? To the previous decision point!

• Algorithm:
  • At every decision point, push all possible choices onto the stack. Pop a choice and go that route. If you hit a dead end, pop the next choice from the stack.
Backtracking

Decision point @ E4:
- choices: North, South
Backtracking

Decision point @ E4:
   - choices: North, South

E4: South

Decision point @ D6:
   - choices: North, South

D6: South
D6: North
E4: North
Backtracking

Decision point @ E4:
  - choices: North, South

E4: South

Decision point @ D6:
  - choices: North, South

D6: South

Dead End! Backtrack:

D6: North
E4: North
Backtracking

Decision point @ E4:
  - choices: North, South

E4: South

Decision point @ D6:
  - choices: North, South

D6: South

Dead End! Backtrack:

D6: North
Backtracking

Decision point @ E4:
  - choices: North, South

E4: South

Decision point @ D6:
  - choices: North, South

D6: South

Dead End! Backtrack:

D6: North

Dead End! Backtrack:

E4: North
Backtracking

Decision point @ E4:
  - choices: North, South

E4: South

Decision point @ D6:
  - choices: North, South

D6: South

Dead End!  Backtrack:

D6: North

Dead End!  Backtrack:

E4: North
Backtracking

Decision point @ E4:
- choices: North, South

E4: South

Decision point @ D6:
- choices: North, South

D6: South

Dead End!  Backtrack:

D6: North

Dead End!  Backtrack:

E4: North

Decision point @ G4:
- choices: East, West
Backtracking

Decision point @ E4:
  - choices: North, South

E4: South

Decision point @ D6:
  - choices: North, South

D6: South

Dead End!  Backtrack:

D6: North

Dead End!  Backtrack:

E4: North

Decision point @ G4:
  - choices: East, West

G4: West
Backtracking

Dead End! Backtrack:
Backtracking

Dead End! Backtrack:

G4: East
Backtracking

Dead End! Backtrack:

G4: East

Success!

This backtracking process is used for many other algorithms as well.
Hardware Stacks

- We've already learned about one hardware application of a stack: where function local variables are stored in memory.
- The stack is also used to store the return address during a function call.
  - This is how nested function calls work (remember the push/pop behavior from the Dynamic Memory lecture?)
  - When the function terminates, its stack frame is removed and the return address (where to resume executing code, i.e. where the function call was) is retrieved.
Stack Terminology

• **Stack overflow**
  • The condition resulting from trying to push an element onto a full stack

```java
if(!stack.isFull())
    stack.push(item);
```

• **Stack underflow**
  • The condition resulting from trying to pop an empty stack

```java
if(!stack.isEmpty())
    stack.pop();
```
Stack Implementations

• A stack is an ADT, which means it's defined only in terms of the properties of the data and the operations that can be performed on that data

• Like with the List ADT, there are several data structures we could use to implement a Stack:
  
    • **Linked list**  (...but really, you've already done this one!)
      • Push == prepend node to list
      • Pop == remove front node from list and return its value

    • **Fixed-size array**
      • Keep track of top index; stack has maximum size and can be full

    • **Dynamic array**
      • Keep track of top index; re-size/re-allocate array when full
A Fixed-Size Array Stack Class

class IntStack {
    private:
        int *stackArray; // the stack array
        int stackSize;   // the array size
        int top;         // which array index is the stack top?

    public:
        IntStack(int);              // constructor
        IntStack(const IntStack &); // copy constructor
        ~IntStack();                // destructor
        void push(int);
        int pop();
        bool isFull() const;
        bool isEmpty() const;
};
Stack: Constructors

```cpp
IntStack::IntStack(int size) {
    if(size <= 0)
        size = 1;  // or error

    stackArray = new int[size];
    stackSize = size;
    top = -1;                // empty
}
```

```cpp
IntStack::IntStack(const IntStack &rhs) {
    stackArray = new int[rhs.stackSize];
    stackSize = rhs.stackSize;

    for(int i = 0; i < stackSize; i++)
        stackArray[i] = rhs.stackArray[i];

    top = rhs.top;
}
```
IntStack::~IntStack() {
    delete [] stackArray;
}
Stack: Push

```cpp
void IntStack::push(int item) {
    assert(!isFull()); // stack overflow!
    stackArray[++top] = item;
}
```

The `assert()` statement will cause the program to terminate with an error if `!isFull()` returns false. The best way to remember this: `assert` means "Make sure..."

Really easy syntax for error checking & communication. Just need to `#include <cassert>`

Runtime error will include the line number of the `assert()`

If you want a helpful error message, too, there's always this hack: `assert(!isFull() && "Attempting to push to full stack!");`
int IntStack::pop() {
    assert(!isEmpty());       // stack underflow!

    return stackArray[top--];
}
Stack: isFull & isEmpty

bool IntStack::isFull() {
    return (top == stackSize - 1); // returns true if there is
    // no empty space on stack
}

bool IntStack::isEmpty() {
    return (top == -1);       // returns true if there are no
    // items on stack
}