CS 171: Introduction to Computer Science II

Stacks

Li Xiong
Announcements/Reminders

• Hw1
  – due Thursday 1/31
  – due Sunday 2/3 with 1 late credit
Roadmap

• Java basics
• OO and inheritance
• Arrays and ArrayList
• Abstract data types
  – Stacks
  – Queues
Abstract Data Types

• Data type
  – A data type is a set of values and a set of operations on those values

• Abstract data type (ADT)
  – An abstract data type is a data type whose internal representation is hidden from the client
Implementing and using ADTs

• ADTs are implemented in Java as classes
  – Instance variables are private (hidden from the client)
  – Instance methods may be public (specified in the API) or private (organize the computation and hidden from the client)

• Application programming interface (API)
  – specifies the behavior of an ADT including constructors and instance methods

• Using ADTs
  – Create objects of implemented ADTs
  – Invoke instance methods to operate on data-type values
Examples of Java ADTs

- **Standard system ADTs**
  - Integer, Double, String, StringBuilder ...

- **Data oriented ADTs** – facilitate organizing and processing data
  - Point2D, Interval1D, Date, ...

- **Collection ADTs** – facilitate organizing and manipulating collections of data
  - Stack, Queue, Priority Queue, ArrayList, ...
Stacks and Queues

• Definition and API - intended behavior
• Implementation – how to implement
• Applications - how to use
Stacks and queues

Fundamental data types.
- Value: collection of objects.
- Operations: insert, remove, iterate, test if empty.
- Intent is clear when we insert.
- Which item do we remove?

Stack. Examine the item most recently added.  LIFO = "last in first out"
Queue. Examine the item least recently added.  FIFO = "first in first out"
Stacks

• A stack stores a set of elements but with only two main operations:
  - **Push**: add an element to the top of the stack
  - **Pop**: remove the top element of the stack.

• Pop always removes the last element that’s added to the stack. This is called **LIFO** (Last-In-First-Out).
Stacks – A Familiar Example

• A can of tennis balls
  – Imagine the entire can represents an array, and each ball is an element.
  – It only allows access to one element at a time: the last element.
  – If you remove the last element, you can then access the next-to-last element.
  – There is no way to directly access the element at the bottom.
Stacks – Another Example

• A dynamic list of tasks you perform everyday:
  – Imagine you start your day by working on task A.
  – At any time you may be interrupted by a co-worker asking you for temporary help on task B.
  – While you work on B, someone may interrupt you again for help on task C.
  – When you are done with C, you will resume working on B.
  – Then you go back to work on A.
  – Think about the sequence of tasks you perform.
Stack Example
Stack Example
Stack Example
Using Stacks

• Stacks are typically used as a programmer’s tool, not for storage
  – used for handling program calls and returns
  – used for data manipulations and problem solving (e.g. reverse)

• Arrays (and others) are typically used as data storage structures (e.g. personal records, inventories ...
Client, implementation, interface

Separate interface and implementation.
Ex: stack, queue, bag, priority queue, symbol table, union-find, ....

Benefits.
- Client can't know details of implementation ⇒
  client has many implementation from which to choose.
- Implementation can't know details of client needs ⇒
  many clients can re-use the same implementation.
- Design: creates modular, reusable libraries.
- Performance: use optimized implementation where it matters.

**Client**: program using operations defined in interface.
**Implementation**: actual code implementing operations.
**Interface**: description of data type, basic operations.
Stack API

Warmup API. Stack of strings data type.

```java
public class StackOfStrings {
    StackOfStrings() {
        // create an empty stack
    }
    void push(String s) {
        // insert a new item onto stack
    }
    String pop() {
        // remove and return the item most recently added
    }
    boolean isEmpty() {
        // is the stack empty?
    }
    int size() {
        // number of items on the stack
    }
}
```

Warmup client. Reverse sequence of strings from standard input.
Stack test client

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

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

% java StackOfStrings < tobe.txt
```
public static void main(String[] args)
{
    StackOfStrings stack = new StackOfStrings();
    while (!StdIn.isEmpty())
    {
        String item = StdIn.readString();
        if (item.equals("-")) StdOut.print(stack.pop());
        else stack.push(item);
    }
}

% more tobe.txt
 to be or not to - be -- that -- - is

% java StackOfStrings < tobe.txt
to be not that or be
Stacks

• Definition and API - intended behavior
• Implementation – how to implement
• Applications - how to use
Array implementation of a stack.

- Use array $s[]$ to store $N$ items on stack.
- push(): add new item at $s[N]$.
- pop(): remove item from $s[N-1]$. 

<table>
<thead>
<tr>
<th>$s[]$</th>
<th>to</th>
<th>be</th>
<th>or</th>
<th>not</th>
<th>to</th>
<th>be</th>
<th>null</th>
<th>null</th>
<th>null</th>
<th>null</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
</tbody>
</table>

$N$

capacity = 10
Stack: array implementation

```java
public class FixedCapacityStackOfStrings {
    private String[] s;
    private int N = 0;

    public FixedCapacityStackOfStrings(int capacity) {
        s = new String[capacity];
    }

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

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

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

Stack: Array implementation

• Underflow: what happens if pop from an empty stack?
  – Throw exception

• Overflow: what happens if push to a full stack?
  – Use resizing array
Problem. Requiring client to provide capacity does not implement API!
Q. How to grow and shrink array?

First try.
• `push()`: increase size of array `s[]` by 1.
• `pop()`: decrease size of array `s[]` by 1.
Stack: resizing-array implementation

Problem. Requiring client to provide capacity does not implement API!

Q. How to grow and shrink array?

First try.
- `push()`: increase size of array $s[]$ by 1.
- `pop()`: decrease size of array $s[]$ by 1.

Too expensive.
- Need to copy all item to a new array.
- Inserting first $N$ items takes time proportional to $1 + 2 + \ldots + N \sim \frac{N^2}{2}$.
  
  infeasible for large $N$

Challenge. Ensure that array resizing happens infrequently.
Stack: resizing-array implementation

Q. How to grow array?
A. If array is full, create a new array of **twice** the size, and copy items.

```java
public ResizingArrayStackOfStrings()
{
    s = new String[1];
}

public void push(String item)
{
    if (N == s.length) resize(2 * s.length);
    s[N++] = item;
}

private void resize(int capacity)
{
    String[] copy = new String[capacity];
    for (int i = 0; i < N; i++)
        copy[i] = s[i];
    s = copy;
}
```

**Consequence.** Inserting first $N$ items takes time proportional to $N$ (not $N^2$).
Q. How to shrink array?

First try.

- `push()`: double size of array $s[]$ when array is full.
- `pop()`: halve size of array $s[]$ when array is one-half full.
Q. How to shrink array?

First try.

- \texttt{push()}: double size of array \( s[] \) when array is full.
- \texttt{pop()}: halve size of array \( s[] \) when array is one-half full.

Too expensive in worst case.

- Consider push-pop-push-pop-... sequence when array is full.
- Each operation takes time proportional to \( N \).
Q. How to shrink array?

Efficient solution.

- `push()`: double size of array $s[]$ when array is full.
- `pop()`: halve size of array $s[]$ when array is one-quarter full.

```java
public String pop()
{
    String item = s[--N];
    s[N] = null;
    if (N > 0 && N == s.length/4) resize(s.length/2);
    return item;
}
```

Invariant. Array is between 25% and 100% full.
Stack: Array Implementation

• What’s the cost of pushing/adding to a stack of size N?
  – Case 1: array resizing not required
  – Case 2: array resizing required
Stack: amortized cost of adding to a stack

Cost of inserting first $N$ items.  \[ N + (2 + 4 + 8 + \ldots + N) \sim 3N. \]

1 array accesses per push
k array accesses to double to size $k$
(ignoring cost to create new array)

---

Graph:
- x-axis: number of `push()` operations
- y-axis: cost (array accesses)
- One gray dot for each operation
- Red dots give cumulative average
- At $N=128$, the average cost is $3$
Stack resizing-array implementation: performance

**Amortized analysis.** Average running time per operation over a worst-case sequence of operations.

**Proposition.** Starting from an empty stack, any sequence of $M$ push and pop operations takes time proportional to $M$.

<table>
<thead>
<tr>
<th></th>
<th>best</th>
<th>worst</th>
<th>amortized</th>
</tr>
</thead>
<tbody>
<tr>
<td>construct</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>push</td>
<td>1</td>
<td>N</td>
<td>1</td>
</tr>
<tr>
<td>pop</td>
<td>1</td>
<td>N</td>
<td>1</td>
</tr>
<tr>
<td>size</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

doubling and halving operations

order of growth of running time for resizing stack with N items
Stacks

• Definition and API - intended behavior
• Implementation – how to implement
  – Implementation using fixed size array
    • FixedCapacityStackOfStrings.java
  – Implementation using resizable array
    • ResizingArrayStack.java (using generics)
  – Implementation using generics
• Applications - how to use
Parameterized stack

We implemented: StackOfStrings.
We also want: StackOfURLs, StackOfInts, StackOfVans, ...
Parameterized stack

We implemented: `StackOfStrings`.

We also want: `StackOfURLs, StackOfInts, StackOfVans, ....`

Attempt 1. 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.

@#$*! most reasonable approach until Java 1.5.
Parameterized stack

We implemented: StackOfStrings.
We also want: StackOfURLs, StackOfInts, StackOfVans, ....

Attempt 2. Implement a stack with items of type object.
- Casting is required in client.
- Casting is error-prone: run-time error if types mismatch.

```java
StackOfObjects s = new StackOfObjects();
Apple a = new Apple();
Orange b = new Orange();
s.push(a);
s.push(b);
a = (Apple) (s.pop());
```

run-time error
Parameterized stack

We implemented: StackOfStrings.
We also want: StackOfURLs, StackOfInts, StackOfVans, ....

Attempt 3. Java generics.
• Avoid casting in client.
• Discover type mismatch errors at compile-time instead of run-time.

Guiding principles. Welcome compile-time errors; avoid run-time errors.
Generic stack: array implementation

```java
public class FixedCapacityStackOfStrings {
    private String[] s;
    private int N = 0;

    public FixedCapacityStackOfStrings(int capacity) {
        s = new String[capacity];
    }

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

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

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

```java
public class FixedCapacityStack<Item> {
    private Item[] s;
    private int N = 0;

    public FixedCapacityStack(int capacity) {
        s = new Item[capacity];
    }

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

    public void push(Item item) {
        s[N++] = item;
    }

    public Item pop() {
        return s[--N];
    }
}
```

@#$^! generic array creation not allowed in Java
Generic stack: array implementation

```
public class FixedCapacityStackOfStrings {
    private String[] s;
    private int N = 0;

    public StackOfStrings(int capacity) {
        s = new String[capacity];
    }

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

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

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

```
public class FixedCapacityStack<Item> {
    private Item[] s;
    private int N = 0;

    public FixedCapacityStack(int capacity) {
        s = (Item[]) new Object[capacity];
    }

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

    public void push(Item item) {
        s[N++] = item;
    }

    public Item pop() {
        return s[--N];
    }
}
```
Generic data types: autoboxing

Q. What to do about primitive types?

Wrapper type.
• Each primitive type has a wrapper object type.
• Ex: `Integer` is wrapper type for `int`.

Autoboxing. Automatic cast between a primitive type and its wrapper.

Syntactic sugar. Behind-the-scenes casting.

```java
Stack<Integer> s = new Stack<Integer>();
s.push(17);    // s.push(new Integer(17));
int a = s.pop(); // int a = s.pop().intValue();
```

Bottom line. Client code can use generic stack for any type of data.
Stack: Implementations

- Stack of strings using fixed-capacity array: [FixedCapacityStackOfStrings.java](#)
- Generic stack using fixed-capacity array: [FixedCapacityStack.java](#)
- Generic stack using a resizing array: [ResizingArrayStack.java](#)
- Generic stack using a linked list (later): [Stack.java](#)
Stacks

• Definition and API - intended behavior
• Implementation – how to implement
• Applications - how to use
Stack: Applications

- Application 1: Reverse a list of integers
- Application 2: Delimiter matching
- Application 3: Expression evaluation
- Hw2: N-Queens Problem
- Other applications
  - Undo/redo history
  - Browsing history (back button in browser)
  - Call stack
Application 1: Reverse a list of integers

- Reads a sequence of integers, prints them in reverse order
Application 1: Reverse a list of integers

- Reads a sequence of integers, prints them in reverse order
  - Push the integers to a stack one by one
  - pop and print them one by one

Reverse.java
Application 2 – Delimiter Matching

• Check if the parentheses in an mathematical expression is balanced:
  \[(w * (x + y) / z - (p / (r - q))\)

• It may have several different types of delimiters: braces\{}, brackets\[], parentheses().
  – Each opening on the left delimiter must be matched by a closing (right) delimiter.
  – Left delimiters that occur later should be closed before those occurring earlier.
Application 2 – Delimiter Matching

• Examples:
  
  c[d]
  a{b[c]d}e
  a{b[c]d}e
  a[b{c}d]e}
  a{b(c)
Application 2 – Delimiter Matching

• Examples:

```plaintext
c[d]       // correct
a{b[c]d}e  // correct
a{b(c)d}e  // not correct; ] doesn't match ( 
 a[b{c}d]e} // not correct; nothing matches final }
a{b(c)}    // not correct; nothing matches opening {
```
Application 2 – Delimiter Matching

• Use a stack:
  – Read characters from the string.
  – Whenever you see a left (opening) delimiter, push it to the stack.
  – Whenever you see a right (closing) delimiter, pop the opening delimiter from the stack and match.
  – What are the error conditions?
Application 2 – Delimiter Matching

• Use a stack:
  – Read characters from the string.
  – Whenever you see a left (opening) delimiter, push it to the stack.
  – Whenever you see a right (closing) delimiter, pops the opening delimiter from the stack and match.
  – If they don’t match, matching error.
  – If stack is empty when you try to match a closing delimiter, missing left delimiter error
  – If the stack is non-empty after all characters are processed, missing right delimiter error
Application 2 – Delimiter Matching

• Program: ~cs171000/share/code/Brackets
• Why does this work?
  – Delimiters that are opened last must be closed first.
  – This conforms exactly with the LIFO property of the stack.
public void check()
{
    int stackSize = input.length(); // get max stack size
    StackX theStack = new StackX(stackSize); // make stack

    for(int j=0; j<input.length(); j++) // get chars in turn
    {
        char ch = input.charAt(j); // get char
        switch (ch) // get char
        {
            case '{': // opening symbols
                case '[':
                case '(':
                    theStack.push(ch); // push them
                    break;
            case '}': // closing symbols
                case ']':
                case ')':
                    if( !theStack.isEmpty() ) // if stack not empty,
                        {
                            char chx = theStack.pop(); // pop and check
                            if( (ch=='}' && chx!='{') ||
                                (ch==']' && chx!='[') ||
                                (ch==')' && chx!='(') )
                                System.out.println("Error: "+ch+" at "+j);
                        }
                    else // prematurely empty
                        System.out.println("Error: "+ch+" at "+j);
                    break;
            default:    // no action on other characters
                        break;
        } // end switch
    } // end for
    // at this point, all characters have been processed
    if( !theStack.isEmpty() )
        System.out.println("Error: missing right delimiter");
} // end check()
```java
public void check()
{
    int stackSize = input.length(); // get max stack size
    StackX theStack = new StackX(stackSize); // make stack

    for(int j=0; j< input.length(); j++)  // get chars in turn
    {
        char ch = input.charAt(j); // get char
        switch(ch) // get char
            {
            case '{':                      // opening symbols
                case '[':
                case '(':                      // push them
                    theStack.push(ch);          // push them
                    break;
                case '}':                      // closing symbols
                case ']' :
                case ')':
                if(!theStack.isEmpty())   // if stack not empty,
                {
                    char chx = theStack.pop();  // pop and check
                    if( (ch=='}') && chx!='{') ||
                        (ch==']' && chx!='[') ||
                        (ch==')' && chx!='(') )
                        System.out.println("Error: "+ch+" at "+j);
                }
                else                        // prematurely empty
                    System.out.println("Error: "+ch+" at "+j);
                    break;
                default:    // no action on other characters
                    break;
            }  // end switch
        }  // end for
    // at this point, all characters have been processed
    if( !theStack.isEmpty() )
        System.out.println("Error: missing right delimiter");
    }  // end check()
```

public void check()
{
    int stackSize = input.length();  // get max stack size
    StackX theStack = new StackX(stackSize);  // make stack

    for(int j=0; j<input.length(); j++)  // get chars in turn
    {
        char ch = input.charAt(j);  // get char
        switch(ch)
        {
            case '{':                      // opening symbols
            case '[':
            case '(':  
                theStack.push(ch);          // push them
                break;
            case '}':                      // closing symbols
            case ']':  
            case ')':  
                if( !theStack.isEmpty() )   // if stack not empty,
                {
                    char chx = theStack.pop();  // pop and check
                    if( (ch=='{') & (chx!='{') ||
                        (ch==']') & (chx!='[') ||
                        (ch==')') & (chx!='(') )
                        System.out.println("Error: "+ch+" at "+j);
                }
                else                        // prematurely empty
                    System.out.println("Error: "+ch+" at "+j);
                break;
            default:    // no action on other characters
                break;
        }  // end switch
    }  // end for
    // at this point, all characters have been processed
    if( !theStack.isEmpty() )
        System.out.println("Error: missing right delimiter");
}  // end check()
public void check()
{
    int stackSize = input.length();  // get max stack size
    StackX theStack = new StackX(stackSize);  // make stack

    for(int j=0; j<input.length(); j++)  // get chars in turn
    {
        char ch = input.charAt(j);  // get char
        switch(ch)
        {
            case '{':                      // opening symbols
                case '[':
                case '(':
                    theStack.push(ch);          // push them
                    break;
            case '}':                      // closing symbols
                case ']':
                case ')':
                    if( !theStack.isEmpty() )   // if stack not empty,
                    {
                        char chx = theStack.pop();  // pop and check
                        if( (ch=='}'&&chx!='{') ||
                            (ch==']'&&chx!='[') ||
                            (ch==')'&&chx!='(') )
                            System.out.println("Error: "+ch+" at "+j);
                    }
                    else                        // prematurely empty
                        System.out.println("Error: "+ch+" at "+j);
                    break;
            default:    // no action on other characters
                break;
        }  // end switch
    }  // end for

    // at this point, all characters have been processed
    if( !theStack.isEmpty() )
        System.out.println("Error: missing right delimiter");
}  // end check()
Application: Arithmetic Expression Evaluation

- Infix arithmetic expressions – operators are placed between two operands
  
  \[ 2 + 3 \]
  
  \[ 1 + (2 + 3) \times (4 \times 5) \]

- Fully parenthesized infix arithmetic expressions
  
  \[(2 + 3)\]
  
  \[(1 + (2 + 3) \times (4 \times 5))\]

Arithmetic expression evaluation

Goal. Evaluate infix expressions.

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

- Code
  Evaluate.java
- Demo
  http://algs4.cs.princeton.edu/lectures/13DemoDijkstraTwoStack.mov
Postfix (RPN) Notation

• For computers to parse the expressions, it’s more convenient to represent expressions in postfix notation, also known as reverse polish notation (RPN)

• Operators are placed after operands.
  
  \[23 + \]
  
  \[AB/\]

• Postfix notation is parenthesis-free as long all operators have fixed # operands
Evaluating Postfix Expressions

• Example: 345+*612+/-

• This is equivalent to the infix expression:
  3*(4+5) - 6 / (1+2)

How do we evaluate this postfix expression?
Implementation Idea

• Whenever we encounter an operator, we apply it to the last two operands we’ve seen.

<table>
<thead>
<tr>
<th>Item Read from Postfix Expression</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operand</td>
<td>Push it onto the stack.</td>
</tr>
<tr>
<td>Operator</td>
<td>Pop the top two operands from the stack and apply the operator to them. Push the result.</td>
</tr>
</tbody>
</table>

345+*612+/-
Hw2 – N-Queens Problem

• Place N queens on a NxN chess board such that no two queens attack each other.
  – No two queens share the same row, column, or diagonal

• A search problem
Brute-force Search

• Enumerate all possible candidates and check whether it satisfies the constraint

• If we know N ahead of time (say, N=4), we can use an N-nested loop (enforcing one queen in each row)

```cpp
for (int i=0; i<n; i++) {
    for (int j=0; j<n; j++) {
        for (int k=0; k<n; k++) {
            for (int m=0; m<n; m++) {
                // check validity of (i,j,k,m)
            }
        }
    }
}
```
Backtracking

• For each row, place a queen in first valid position, then move to next row
• If there is no valid position, then backtrack to previous row and try next position
• If you successfully place a queen in the last row, then a solution is found, backtrack to find next solution
Using backtracking to solve 4-queens.

The orange squares represent the placement of queens. Based off of a diagram in [5].
Using Stack for Backtracking

• Use Stack to keep track of current column positions in each row
  – If you find a valid position in current row, push to the stack and start on next row
  – If there is no valid position in current row, backtrack to previous row - pop the position of previous row from the stack and search for valid position further down the row

• Example stack for 4-queens

• When is a solution found?

• When should the search stop?
Using Stack for Backtracking

• When is a solution found?
  – Stack has size N

• How to find all solutions?
  – When a solution is found, pretend it is not a solution, backtrack to previous row and find next solution

• When should the search stop?
  – When try to pop from stack but stack is empty (no more valid position in first row and need to backtrack)
Algorithm

Create empty stack and set current position to 0
Repeat {
    loop from current position to the last position until a valid position is found //current row
    if there is a valid position {
        push the position to stack, set current position to 0 // move to next row
    }
    if there is no valid position {
        if stack is empty, break // stop search
        else pop from stack, set current position to next position // backtracking to previous row
    }
    if stack has size N { // a solution is found
        pop from stack, set current position to next position // backtracking to find next solution
    }
}
Check for Valid Position

• A good idea to implement a method for checking if a position is valid given the queen positions in previous rows
• Need to check against all positions in previous rows (all elements in the stack)
Check for Valid Position

- Let $c$ be the position (column index) of the trial queen, $r$ be its row index.
- Let $c(i)$ be the position of an existing queen $i$, and $i$ be its row index.
  - Note that $c(i)$ is stored in the stack.
- Loop over all existing queen $i$
  - If $(c(i) == c)$, there is a column conflict.
  - If $(c - c(i) == r - i )$, there is a major diagonal conflict.
  - If $(c(i) - c == r - i )$, there is a minor diagonal conflict.
- Otherwise, the trial queen is conflict free!
Java Stack class

• Generic Stack class in java.util package
• Useful methods
  – `size()`: Returns number of elements
  – `pop()`: Removes and returns the object at the top
  – `push(E item)`: Pushes an item onto the top
  – `get(int index)`: Returns the element at the specified position
Using Java Stack Class

```java
import java.util.Stack;

Stack<Integer> s = new Stack<Integer>();
s.push(0); // push position 0 to the stack
s.push(2); // push position 3 to the stack
int size = s.size(); // return number of elements
int col = s.get(0); // return value at index 0
col = s.pop(); // pop position from the stack
```