Data Structures & Programming

Stacks

Golnar Sheikhshab

Stack Abstract Data Type (ADT)

Last in First out (LIFO) container of objects

Applications:

- back button in browser
- undo functionality
- keeping the variables in recursive calls

Stack ADT

Functions:

size():	Return the number of elements in the stack.
empty():	Return true if the stack is empty and false otherwise.
push(e):	Push e onto the top of the stack.
pop():	Pop the element at the top of the stack.
top():	Return a reference to the element at the top of the stack.

Example 5.3: *The following table shows a series of stack operations and their effects on an initially empty stack of integers.*

Operation	*Output*	*Stack Contents*
push(5)	–	(5)
push(3)	–	(5,3)
pop()	–	(5)
push(7)	–	(5,7)
pop()	–	(5)
top()	*5*	(5)
pop()	–	()
pop()	*"error"*	()
top()	*"error"*	()
empty()	*true*	()
push(9)	–	(9)
push(7)	–	(9,7)
push(3)	–	(9,7,3)
push(5)	–	(9,7,3,5)
size()	*4*	(9,7,3,5)
pop()	–	(9,7,3)
push(8)	–	(9,7,3,8)
pop()	–	(9,7,3)
top()	*3*	(9,7,3)

STL stack

```
#include <stack>
using std::stack;                    // make stack accessible
stack<int> myStack;                  // a stack of integers
```

here the **base type** is int

STL containers including stack, don't throw exceptions

STL stack may cause the program to crash (abort)

for example if one calls pop() on an empty stack

Stack informal interface

```
template <typename E>
class Stack {
public:
    int size() const;
    bool empty() const;
    const E& top() const throw(StackEmpty);
    void push(const E& e);
    void pop() throw(StackEmpty);
};
```

How can we make it a formal interface?

By making it an abstract class. How?

By making one of the functions pure virtual.

Stack formal interface

```
template <typename E>
class Stack {
public:
    virtual int size() const = 0;
    bool empty() const;
    const E& top() const throw(StackEmpty);
    void push(const E& e);
    void pop() throw(StackEmpty);
};
```

Stack Exceptions

```
// Exception thrown on performing top or pop of an empty stack.
class StackEmpty : public RuntimeException {
public:
  StackEmpty(const string& err) : RuntimeException(err) {}
};

// Exception thrown on performing top or pop of an empty stack.
class StackFull : public RuntimeException {
public:
  StackFull(const string& err) : RuntimeException(err) {}
};
```

Class RuntimeException

```
class RuntimeException { // generic run-time exception
private:
  string errorMsg;
public:
  RuntimeException(const string& err) { errorMsg = err; }
  string getMessage() const { return errorMsg; }
};
```

A simple array based stack implementation

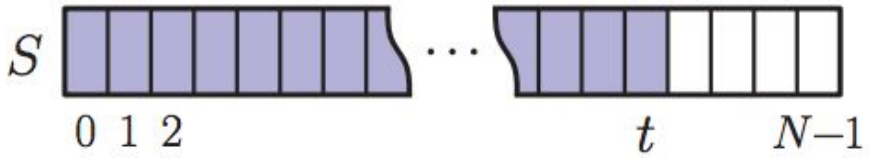


Figure 5.2: Realization of a stack by means of an array S. The top element in the stack is stored in the cell $S[t]$.

Array based implementation of stack

```
Algorithm size():
    return t + 1
Algorithm empty():
    return (t < 0)
Algorithm top():
    if empty() then
        throw StackEmpty exception
    return S[t]
```

```
Algorithm push(e):
    if size() = N then
        throw StackFull exception
    t ← t + 1
    S[t] ← e
Algorithm pop():
    if empty() then
        throw StackEmpty exception
    t ← t − 1
```

Code Fragment 5.3: Implementation of a stack by means of an array.

Time complexity of array based implementation

Operation	*Time*
size	$O(1)$
empty	$O(1)$
top	$O(1)$
push	$O(1)$
pop	$O(1)$

C++ implementation of ArrayStack

```
template <typename E>
class ArrayStack :Stack<E> {
public:
    ArrayStack(int capacity=1000):a_(new E[capacity]), capacity_(capacity), top_(-1){}
    int size() const {return top_+1;}
    bool empty() const {return top_<0;}
    const E& top() const throw(StackEmpty);  // implementation in next slide
    void push(const E& e); // implementation in next slide
    void pop() throw(StackEmpty); // implementation in next slide
private:
    E* a_;
    int top_;
    int capacity_;
};
```

C++ implementation of ArrayStack (2)

```
const E& top() const throw(StackEmpty){
        if (empty())
                throw StackEmpty("calling top() on an empty stack!");
        return a_[top_];}

void push(const E& e){
        if (top_>=capacity_)
                throw StackEmpty("can't push into a full stack!");
        a_[++top_] = e;}

void pop() throw(StackEmpty){
        if (empty())
                throw StackEmpty("calling pop() on an empty stack!");
        top_--;}
```

Using ArrayStack

```
#include <iostream>
#include "ArrayStack.h"
using namespace std;
int main(){
	ArrayStack<int> A;          // A = [ ], size = 0
	A.push(7);                  // A = [7*], size = 1
	A.push(13);                 // A = [7, 13*], size = 2
	cout << A.top() << endl;    // A = [7, 13*], outputs: 13
	A.pop();                    // A = [7*], size = 1
	A.push(9);                  // A = [7, 9*], size = 2
	cout << A.top() << endl;    // A = [7, 9*], outputs: 9
	A.pop();                    // A = [7]
}
```

C++ Implementation of LinkedStack (non-generic)

```
typedef string Elem;                                  // stack element type
class LinkedStack {                                   // stack as a linked list
public:
  LinkedStack();                                      // constructor
  int size() const;                                   // number of items in the stack
  bool empty() const;                                 // is the stack empty?
  const Elem& top() const throw(StackEmpty);          // the top element
  void push(const Elem& e);                           // push element onto stack
  void pop() throw(StackEmpty);                       // pop the stack
private:                                              // member data
  SLinkedList<Elem> S;                                // linked list of elements
  int n;                                              // number of elements
};
```

C++ Implementation of LinkedStack (non-generic)

```
LinkedStack::LinkedStack(): S(), n(0) {} // constructor

int LinkedStack::size() const {return n;} // number of items in the stack

bool LinkedStack::empty() const {return n == 0;} // is the stack empty?

const Elem& LinkedStack::top() const throw(StackEmpty) {
  if (empty())
     throw StackEmpty("Top of empty stack");
  return S.front();
}
```

C++ Implementation of LinkedStack (non-generic)

```
void LinkedStack::push(const Elem& e) { // push element onto stack
  ++n;
  S.addFront(e);
}

// pop the stack
void LinkedStack::pop() throw(StackEmpty) {
  if (empty())
     throw StackEmpty("Pop from empty stack");
  --n;
  S.removeFront();
}
```

.h and .cpp files

What goes into the .h file?

What goes into the .cpp file?

How is a generic class different from a non-generic class in terms of .h and .cpp files?

Reversing a vector using stack data structure

```
void reverse(vector<string>& V) {    // reverse a vector
  ArrayStack<string> S(V.size());
  for (int i = 0; i < V.size(); i++) // push elements onto stack
    S.push(V[i]);
  for (int i = 0; i < V.size(); i++) { // pop them in reverse order
    V[i] = S.top();
    S.pop();
  }
}
```

Using stacks for parentheses checking

There are many pairs of parenthesis like characters:

- Parentheses: “(” and “)”
- Braces: “{” and “}”
- Brackets: “[” and “]”
- Floor function symbols: “⌊” and “⌋”
- Ceiling function symbols: “⌈” and “⌉,”

The problem is to check if the parenthesizing is correct

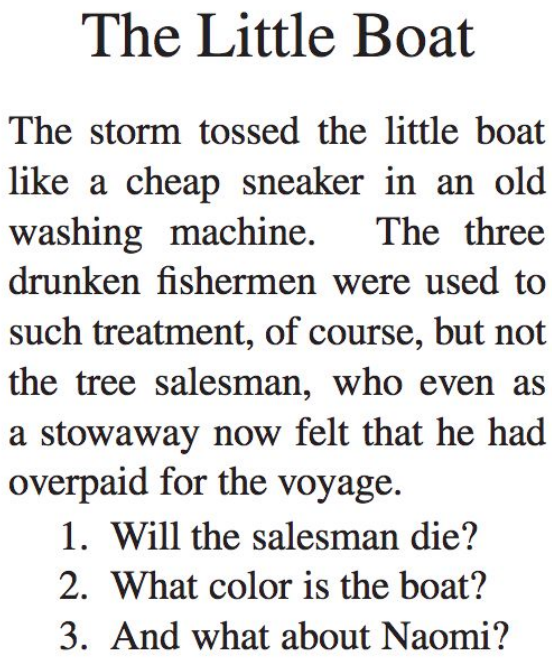
- Correct: ( )(( )){([( )])}
- Correct: ((( )(( )){([( )])}))
- Incorrect:)(( )){([( )])}
- Incorrect: ({[])}
- Incorrect: (

```
Algorithm ParenMatch(X, n):
    Input: An array X of n tokens, each of which is either a grouping symbol, a
      variable, an arithmetic operator, or a number
    Output: true if and only if all the grouping symbols in X match
    Let S be an empty stack
    for i ← 0 to n − 1 do
      if X[i] is an opening grouping symbol then
        S.push(X[i])
      else if X[i] is a closing grouping symbol then
        if S.empty() then
          return false          {nothing to match with}
        if S.top() does not match the type of X[i] then
          return false          {wrong type}
        S.pop()
    if S.empty() then
      return true               {every symbol matched}
    else
      return false              {some symbols were never matched}
```

If we have only regular parentheses "(" and ")", do we need stacks to check if the parenthesizing is correct?

Matching tags in an HTML document

```
<body>
<center>
<h1> The Little Boat </h1>
</center>
<p> The storm tossed the little
boat like a cheap sneaker in an
old washing machine. The three
drunken fishermen were used to
such treatment, of course, but
not the tree salesman, who even
as a stowaway now felt that he
had overpaid for the voyage. </p>
<ol>
<li> Will the salesman die? </li>
<li> What color is the boat? </li>
<li> And what about Naomi? </li>
</ol>
</body>
```



The Little Boat

The storm tossed the little boat like a cheap sneaker in an old washing machine. The three drunken fishermen were used to such treatment, of course, but not the tree salesman, who even as a stowaway now felt that he had overpaid for the voyage.

1. Will the salesman die?
2. What color is the boat?
3. And what about Naomi?

```
vector<string> getHtmlTags() {                  // store tags in a vector
  vector<string> tags;                          // vector of html tags
  while (cin) {                                 // read until end of file
    string line;
    getline(cin, line);                         // input a full line of text
    int pos = 0;                                // current scan position
    int ts = line.find("<", pos);               // possible tag start
    while (ts != string::npos) {                // repeat until end of string
      int te = line.find(">", ts+1);            // scan for tag end
      tags.push_back(line.substr(ts, te-ts+1)); // append tag to the vector
      pos = te + 1;                             // advance our position
      ts = line.find("<", pos);
    }
  }
  return tags;                                  // return vector of tags
}
```

```
                                                    // check for matching tags
bool isHtmlMatched(const vector<string>& tags) {
  LinkedStack S;                                    // stack for opening tags
  typedef vector<string>::const_iterator Iter;      // iterator type
                                                    // iterate through vector
  for (Iter p = tags.begin(); p != tags.end(); ++p) {
    if (p->at(1) != '/')                            // opening tag?
      S.push(*p);                                   // push it on the stack
    else {                                          // else must be closing tag
      if (S.empty()) return false;                  // nothing to match - failure
      string open = S.top().substr(1);              // opening tag excluding '<'
      string close = p->substr(2);                  // closing tag excluding '</'
      if (open.compare(close) != 0) return false; // fail to match
      else S.pop();                                 // pop matched element
    }
  }
  if (S.empty()) return true;                       // everything matched - good
  else return false;                                // some unmatched - bad
}
```

Reading material

Section 5.1 of your textbook