Stacks

Section 5.1
Abstract Data Types (ADTs)

- An abstract data type (ADT) is an abstraction of a data structure and operations on it.
- An ADT specifies:
  - Data stored
  - Operations on the data
  - Error conditions associated with operations
The Stack ADT

- The **Stack** ADT stores arbitrary objects
- Insertions and deletions follow the last-in first-out (LIFO) scheme
- Main stack operations:
  - `void push(object)`: inserts an element on the top.
  - `void pop()`: removes the top element. Gives a StackEmpty error if there is no element to remove.
- Auxiliary stack operations:
  - `object top()`: returns the element at the top without removing it. Gives a StackEmpty error if the stack is empty.
  - `integer size()`: returns the number of elements stored.
  - `boolean empty()`: indicates whether no elements are stored.
- void `push(object)`: inserts an element on the top.
- void `pop()`: removes the top element.
- object `top()`: returns the element at the top without removing it.
- integer `size()`: returns the number of elements stored.
- boolean `empty()`: indicates whether no elements are stored.
Stack Interface

- Pseudo-C++ interface corresponding to our Stack ADT
- Uses an exception class `StackEmpty`
- Different from the built-in C++ STL class `stack`

```cpp
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);
};
```
STL stack class

- The Standard Template Library (STL) provides an implementation of a stack.
- To declare a stack of integers:
  ```
  #include <stack>
  std::stack<int> myStack;
  ```
- STL’s stack interface is basically the same as the one we just saw, except that executing `pop` or `top` on an empty stack results in *undefined behavior*. This generally means your program crashes.
Array-based Stack Implementation

- A simple way of implementing the Stack ADT uses an array
- We add elements from left to right
- A variable $t$ keeps track of the index of the top element

Algorithm size()
return $t + 1$

Algorithm pop()
if empty() then
    throw StackEmpty
else
    $t \leftarrow t - 1$
    return $S[t + 1]$

© 2010 Goodrich, Tamassia
Array-based Stack

- The array storing the stack elements may become full
- A push operation will then throw a `StackFull` exception
  - Limitation of the simple array-based implementation
  - Not intrinsic to the Stack ADT

Algorithm `push(o)`

\[
\text{if } t = \text{capacity} - 1 \text{ then throw } \text{StackFull} \\
\text{else} \\
\quad t \leftarrow t + 1 \\
\quad S[t] \leftarrow o
\]
Performance and Limitations

- **Performance**
  - Let $n$ be the number of elements in the stack
  - The space used is at least $n$
  - Each operation runs in time $O(1)$

- **Limitations**
  - The maximum size of the stack must be defined *a priori* and cannot be changed
  - Trying to push a new element into a full stack causes an implementation-specific exception
Array-based Stack in C++

template <typename E>
class ArrayStack {
private:
    E* S; // array holding the stack
    int cap; // capacity
    int t; // index of top element
public:
    // constructor given capacity
    ArrayStack(int c) :
        S(new E[c]), cap(c), t(-1) { }

    void pop() {
        if (empty()) throw StackEmpty
            ("Pop from empty stack");
        t--;
    }
    void push(const E& e) {
        if (size() == cap) throw
            StackFull("Push to full stack");
        S[++t] = e;
    }

Array-based Stack in C++

```cpp
const E& top() {
    if (empty()) throw StackEmpty ("Top from empty stack");
    return S[t];
}
int size() {
    return t+1;
}
bool empty() {
    return t < 0;
}
} // end of class body
```

Book shows this implementation outside of the class body, like in a .cpp file. It also includes proper templating for that case, and the proper throw declarations.
Linked List-based Stack

- We can implement a stack with a singly linked list.
- The top element is stored at the first node of the list.
- The space used is $O(n)$ and each operation of the Stack ADT takes $O(1)$ time.
- No restrictions on the number of elements.
### Example use in C++

ArrayStack<int> A;
A.push(7);
A.push(13);
cout << A.top() << endl; A.pop();
A.push(9);
cout << A.top() << endl; 
cout << A.top() << endl; A.pop();

ArrayStack<string> B(10);
B.push("Bob");
B.push("Alice");
cout << B.top() << endl; B.pop();
B.push("Eve");

* indicates top

// A = [ ], size = 0
// A = [7*], size = 1
// A = [7, 13*], size = 2
// A = [7*], outputs: 13
// A = [7, 9*], size = 2
// A = [7, 9*], outputs: 9
// A = [7*], outputs: 9

// B = [ ], size = 0
// B = [Bob*], size = 1
// B = [Bob, Alice*], size = 2
// B = [Bob*], outputs: Alice
// B = [Bob, Eve*], size = 2
Applications of Stacks

- Direct applications
  - Page-visited history in a Web browser
  - Undo sequence in a text editor
  - Chain of method calls in the C++ run-time system

- Indirect applications
  - Auxiliary data structure for algorithms
  - Component of other data structures

© 2010 Goodrich, Tamassia
C++ Run-Time Stack

- The C++ run-time system keeps track of the chain of active functions with a stack.
- When a function is called, the system pushes on the stack a frame containing:
  - Local variables and return value
  - Program counter, keeping track of the statement being executed
- When the function ends, its frame is popped from the stack and control is passed to the function on top of the stack.
- Allows for recursion

```c++
main() {
    int i = 5;
    foo(i);
}
foo(int j) {
    int k;
    k = j+1;
    bar(k);
}
bar(int m) {
    ...
}
```

© 2010 Goodrich, Tamassia

Stacks
Parentheses Matching

- Each “(”, “{”, or “[” must be paired with a matching “)”, “}”, or “]”
  - correct: ( )(( )){([([ ]))}
  - correct: ((( )( )){([([ ]))}
  - incorrect: )(( )){([([ ]))}
  - incorrect: ([ ])
  - incorrect: (}
Parentheses Matching Algorithm

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.pop() does not match the type of X[i] then
        return false {wrong type}

if S.empty() then
    return true {every symbol matched}
else return false {some symbols were never matched}
Using a stack as an auxiliary data structure in an algorithm

Given an array $X$, the span $S[i]$ of $X[i]$ is the maximum number of consecutive elements $X[j]$ immediately preceding $X[i]$ and such that $X[j] \leq X[i]$

Spans have applications to financial analysis

- E.g., stock at 52-week high

<table>
<thead>
<tr>
<th>X</th>
<th>6</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>1</td>
</tr>
</tbody>
</table>
Quadratic Algorithm

Algorithm $spans1(X, n)$

Input array $X$ of $n$ integers
Output array $S$ of spans of $X$

$S \leftarrow$ new array of $n$ integers

for $i \leftarrow 0$ to $n - 1$ do

$s \leftarrow 1$

while $s \leq i$ and $X[i - s] \leq X[i]$

$s \leftarrow s + 1$

$S[i] \leftarrow s$

return $S$

Algorithm $spans1$ runs in $O(n^2)$ time
Computing Spans with a Stack

- We keep in a stack the indices of the elements visible when “looking back”
- We scan the array from left to right
  - Let $i$ be the current index
  - We pop indices from the stack until we find index $j$ such that $X[i] < X[j]$
  - We set $S[i] \leftarrow i - j$
  - We push $i$ onto the stack
Linear Algorithm

Each index of the array
- Is pushed into the stack exactly once
- Is popped from the stack at most once

The statements in the while-loop are executed at most $n$ times overall

Algorithm $spans2(X, n)$

```plaintext
S ← new array of $n$ integers
A ← new empty stack
for $i ← 0$ to $n - 1$ do
    while (¬A.empty() ∧ $X[A.top()] ≤ X[i]$) do
        A.pop()
        if A.empty() then
            $S[i] ← i + 1$
        else
            $S[i] ← i - A.top()$
    A.push($i$)
return $S$
```

© 2010 Goodrich, Tamassia
Algorithm \textit{spans2}(X, n)

\begin{itemize}
  \item $S \leftarrow$ new array of $n$ integers $\text{O}(1)$
  \item $A \leftarrow$ new empty stack $\text{O}(1)$
  \item \textbf{for} $i \leftarrow 0$ \textbf{to} $n - 1$ \textbf{do}
  \item \quad while ($\neg A\.empty() \land X[A\.top()] \leq X[i]$) \textbf{do}
  \item \quad \quad $A\.pop()$ $\text{O}(1)$
  \item \quad \quad \textbf{if} $A\.empty()$ \textbf{then}
  \item \quad \quad \quad $S[i] \leftarrow i + 1$ $\text{O}(1)$
  \item \quad \quad \textbf{else}
  \item \quad \quad \quad $S[i] \leftarrow i - A\.top()$ $\text{O}(1)$
  \item \quad \textbf{A\.push}(i) $\text{O}(1)$
  \item \textbf{return} $S$ $\text{O}(1)$
\end{itemize}

© 2019 Shermer