Class4 - Stacks

Summary

A stack is a dynamic data structure that follows a LIFO order.

Dynamic Data Structures

- linked lists
- `front -> node ... node -> nullptr`
- represent `{l1, l2, l3}`
  To remove the first element, change the front pointer to the next item and delete the old reference
  To add an element to the front, change front pointer and make the new node point to the previous front

List `find` operation
`find(x)` - true if x in on the list, otherwise false
In an array:

```
for i=0 ... n-1 {
  if a[i] = x
    return True
}
return False
```

In a linked list:

```
c <- first
while c is not nullptr {
  n <- node c points to
  if data stored at n is x
    return True
  c <- next pointer in n
}
return False
```

Stack

> ADT that stores a collection of objects in Last-In-First-Out order

Necessary / Fundamental Operations:
`push(x)`: inserts an element at the top
`pop(x)`: removes an element at the top

Convenient:
`empty()`: check for emptiness

- could just get people to use size() == 0 but this method improves our code

`size()`: return number of elements on stack
`top()`: return top, but don't remove it

- Could just `x <- pop(); push(x); // use x` but this makes the code simpler and clearer

Examples of Stack:

- undo button / redo button
- backwards / forwards buttons in browsers

Stack based algorithms for checking grouping symbols

```
// I is an input stream of symbols
S <- new stack
while there are symbols to read from I
  c <- next symbol from I
  if e is a left grouping symbol
    push c on S
  else if c is a right grouping symbol
    if S is empty report error and stop
    d <- pop S
    if c,d dont match report error and stop
end while
if S is not empty report error
report "ok"
```

Stack: Partially-Filled Array Implementation

Variables:
A - Array of stack elements
capacity - size of array
top - index in array of top stack element (-1 if stack is empty)

Stack: Linked list implementation

Which end should be the top?
push: insert at top = front of the list
pop: remove from top = remove from front of list

Complexity of Operations

Linked list: push and pop both take constant time

array: pop takes constant time
push takes constant time - but assuming the stack size remains smaller than the array

In both implementations:

- top requires constant time
- size and empty require constant time if we keep track of the size

Q: what would you do so an array-based implementation would not have a fixed maximum stack size? How would it affect complexity of operations?