STACKS

cs2420 | Introduction to Algorithms and Data Structures | Spring 2016
administrivia...
assignment 6 on linked lists due on Thursday
Assignment 4: Anagrams

Score distribution:

- 0-10: 34
- 11-20: 1
- 21-30: 3
- 31-40: 10
- 41-50: 22
- 51-60: 45
- 61-70: 83

Number of students
last time...
linked lists
linked list vs array

-cost of accessing a random item at location \( i \)?

-cost of \text{removeFirst()}?\ 

-cost of \text{addFirst()}?\ 

\begin{align*}
A) & \quad c \\
B) & \quad \log N \\
C) & \quad N \\
D) & \quad N \log N \\
E) & \quad N^2 \\
F) & \quad N^3 \\
\end{align*}
inserting into an array:

8

inserting into a linked list:
inserting into an array:

5 9 12 17 25

inserting into a linked list:

5 → 9 → 12 → 17 → 25

8
deletion from a linked list:

9 is now stranded — garbage collector will clean it up
doubly-linked lists
- Nodes have a link to next and previous node.
- Allows for traversal in either forward or reverse order.
- Maintains a tail node as well as a head node.
doubly-linked list insertion:
doubly-linked list insertion:

```java
newNode = new Node<Character>();
newNode.data = 'n';
```
doubly-linked list insertion:

```java
newNode = new Node<Character>();
newNode.data = 'n';
newNode.prev = current;
```

![Diagram of a doubly-linked list with nodes a, c, k, o, and y, and a new node n inserted between current and o.]
doubly-linked list insertion:

newNode = new Node<Character>();
newNode.data = 'n';

newNode.prev = current;
newNode.next = current.next;
doubly-linked list insertion:

```java
newNode = new Node<Character>();
newNode.data = 'n';
newNode.prev = current;
newNode.next = current.next;
newNode.prev.next = newNode;
```
doubly-linked list insertion:

newNode = new Node<Character>();
newNode.data = 'n';

newNode.prev = current;
newNode.next = current.next;
newNode.prev.next = newNode;
newNode.next.prev = newNode;
doubly-linked list deletion:

current.prev.next = current.next;
current.next.prev = current.prev;
doubly-linked list deletion:

\[
\text{current.prev.next} = \text{current.next}; \\
\text{current.next.prev} = \text{current.prev};
\]
doubly-linked list deletion:

current.prev.next = current.next;
current.next.prev = current.prev;
doubly-linked list deletion:

current.prev.next = current.next;
current.next.prev = current.prev;
**LinkedList** vs **ArrayList**

- **insertion & deletion:**
  - (assuming position is known)
  - $O(c)$ vs $O(N)$

- **accessing a random item:**
  - $O(N)$ vs $O(c)$

- choose the structure based on the expected use
- what is the common case?
today...
stacks
- a **stack** is a data structure in which insertion and removal is restricted to the **top** (or end) of the list

- also called FIRST-IN, LAST-OUT (FILO)
  - insertion always adds an item to the end
  - deletion always removes an item from the end
important methods

- **push**
  - inserts an item on to the top of the stack

- **pop**
  - removes and returns the item on the top of the stack

- **peek**
  - returns but does not remove the top of the stack

- consecutive calls to **pop** will return items in the reverse order that they were **pushed**
IT IS USEFUL TO THINK OF STACKS AS STANDING UPRIGHT (LIKE A STACK OF DISHES)
it is useful to think of stacks as standing upright (like a stack of dishes)
pop();
push(5);

It is useful to think of stacks as standing upright (like a stack of dishes)
performance

- push, pop, and peek must all be $O(1)$

- we need a very efficient data structure if we expect to only access the last element

HOW CAN WE IMPLEMENT A STACK SO THAT ALL 3 OPERATIONS ARE GUARANTEEED TO BE $O(1)$?
as an array...

- **NOTE**: keep track of a top index

- to push, increment top, then add the item at that index

- to pop, return the item at index top, and decrement top

```
push (a)       push (b)       pop ()

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>a</td>
<td></td>
</tr>
<tr>
<td>a</td>
<td></td>
<td>b</td>
<td></td>
</tr>
<tr>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
</tr>
</tbody>
</table>

top=-1  top=0  top=1  top=0
```
performance

-if we try to push when the underlying array is full, the array must be grown

-any push that requires resizing the array takes $O(N)$ time

-all other operations are constant, $O(1)$

-since pushes that resize the array are rare, the average case for push is still $O(1)$
as a linked list...

- treat the head as the top of the stack
- to push, add to the beginning of the linked list
- to pop, return the top and remove the first item
performance

- linked lists never incur the penalty of resizing
  - adds to a linked list are always $O(1)$

- no wasted extra array space

- all stack operations are $O(1)$

- a stack can be easily implemented on top of an existing linked list with very little extra code!
EXAMPLE: call stack (again!)
-every time a method is invoked a unique frame is created

-when that method returns, execution resumes in the calling frame

-methods return in reverse order in which they were called
  -FILO!
  -what method is the first in and last out?
call stack

main
call stack

main

findAnagrams

sort
main
findAnagrams
sort
compare
call stack
The diagram represents a call stack with the following functions:

- main
- findAnagrams
- sort
- swap

The call stack structure is as follows:

- main
- findAnagrams
- sort
- swap
call stack

sort
findAnagrams
main
call stack

main

findAnagrams

sort

compare
call stack

main

findAnagrams
call stack

main
call stack

main

println
call stack

main
EXAMPLE: symbol matcher
-part of the compilation process for Java’s compiler (and many others) is **symbol matching**

- every `{` must be matched with a corresponding `}`
  - same for `()` and `[]`

-how can we use a stack to determine if all brace symbols are matched?

```java
for(i=0; i<N; i++)
{
    arr[i] = i;
}
```
for (i=0; i<N; i++)
{
    arr[i] = i;
}
for (i = 0; i < N; i++)
{
    arr[i] = i;
}

for (i = 0; i < N; i++)
{
    arr[i] = i;
}
for (i = 0; i < N; i++)
{
    arr[i] = i;
}

for (i = 0; i < N; i++)
{
    arr[i] = i;
}

push

for(i=0; i<N; i++)
{
    arr[i] = i;
}

push

{ push
  pop
}
for (i = 0; i < N; i++)
{
    arr[i] = i;
}
for(i=0; i<N; i++)
{
    arr[i] = i;
}

IF END OF INPUT IS REACHED AND THE STACK IS EMPTY...
ALL THE SYMBOLS ARE BALANCED!
next time...
-reading
  - chapter 16: stacks & queues
  - chapter 2: array-based lists
    - http://opendatastructures.org/ods-java/

-homework
  - assignment 6 due Thursday