TREES

cs2420 | Introduction to Algorithms

1
administrivia...
- assignment 7 due Wednesday at midnight
- next assignment requires pair-programming
- pick up exams in my office hours
number of students

score

midterm 1

average: 70
median: 73
>=95: 10% of class
100: 4 students
last time...
stacks & queues
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

- all methods should be O(c)
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

<table>
<thead>
<tr>
<th>push (a)</th>
<th>push (b)</th>
<th>pop ()</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>b</td>
<td>a</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>top=--1</th>
<th>top=0</th>
<th>top=1</th>
<th>top=0</th>
</tr>
</thead>
</table>
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, $O(c)$

-since pushes that resize the array are rare, the average case for push is still $O(c)$
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

```
null

push (a)

top

push (b)

top

pop ()

top

```
**performance**

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

- no wasted extra array space

- all stack operations are $O(c)$

- a stack can be easily implemented on top of an existing linked list with very little extra code!
-a queue is a FIRST-IN, FIRST-OUT data structure
  -FIFO

-insert on the back, remove from the front

-operations:
  -enqueue… adds an item to the back of the queue
  -dequeue… removes and returns the item at the front

-terminology avoids confusion with a stack!

-like a stack, all operations are $O(c)$
as an array...

- keep track of front and back indices

- front and back advance through the array
  - enqueueing advances back
  - dequeueing advance front

- what happens when back reaches the end of the array?
performance

- using wrap-around, all operations are $O(c)$ on average

- but, $O(N)$ array growing is still a problem in the worst case!

- how do we hand array growth if there is wrap-around in the queue?
  - this is non-trivial…
as a linked list...

- remember, inserting and deleting to the head and tail of a linked list is automatically $O(c)$

- front is analogous to head
- back is analogous to tail

- no messy wrap-around, or growth issues

- which linked list operations are analogous to enqueue and dequeue?
summary

- linked lists and wrap-around arrays are both $O(c)$ for queue implementations

- BUT, arrays are much more complicated to code

- both queues and stacks require very little code on top of a good linked list implementation
today...
-trees
-terminology
-binary trees
-traversing a tree
-EXAMPLE: expression trees
-DOT format
trees
-trees are a linked data structure with a hierarchical formation

-recall that a linked list has a reference to a next (and sometimes previous) node

-trees can have multiple links, called branches

there are multiple directions you can take at any given node
-trees have a hierarchical structure
  -meaning, any node is a subtree of some larger tree
    -except the very top node!
  -in CS, trees are usually represented with the root at the top

-trees are recursive in nature
  -any given node is itself a tree
  -a tree consists of:
    a data element…
    …and more subtrees
-there is a strict parent-to-child relationship among nodes
  -links only go from parent to child
    -not from child to parent
    -not from sibling to sibling

-every node has exactly one parent, except for the root, which has none

-there is exactly one path from the root to any other node
terminology
- **root node**: the single node in a tree that has no parents

- **parent**: a node’s parent has a direct reference to it
  - nodes have AT MOST one parent

- **child**: a node $B$ is a child of node $A$ if $A$ has a direct reference to $B$

- **sibling**: two nodes are siblings if they have the same parent
-leaf node: a node with no children

-inner node: a node with at least one child

-depth: the number of ancestors a node has
  -ie. how many steps to the root
  -children are exactly one level deeper than their parents
  -a root node has depth 0

-height: the depth of a tree’s deepest leaf node
subtree rooted at node \( a \)

subtree rooted at node \( d \)

subtree rooted at node \( c \)
(leaf nodes are trees too!)
example
The root is ___.

```
v9
  
v8
  
v7
  
v6
  
v5
  
v3
  
v2
v4

v1
```
The root is \( ___ \).
The height is \( ___ \).
The root is ____.
The height is ____.
The parent of $v_3$ is ____.
The root is ___.
The height is ___.
The parent of v3 is ___.
The depth of v3 is ___.
The root is ___.
The height is ___.
The parent of \( v_3 \) is ___.
The depth of \( v_3 \) is ___.
The children of \( v_6 \) are ___.
The root is ____.
The height is ____.
The parent of \textbf{v3} is ____.
The depth of \textbf{v3} is ____.
The children of \textbf{v6} are ____.
The ancestors of \textbf{v1} are ____.
The root is ___.
The height is ___.
The parent of \textbf{v3} is ___.
The depth of \textbf{v3} is ___.
The children of \textbf{v6} are ____.
The ancestors of \textbf{v1} are ___.
The descendants of \textbf{v6} are ___.
The root is \underline{v2}.
The height is \underline{3}.
The parent of \underline{v3} is \underline{v2}.
The depth of \underline{v3} is \underline{2}.
The children of \underline{v6} are \underline{v5, v7, v8}.
The ancestors of \underline{v1} are \underline{v2}.
The descendants of \underline{v6} are \underline{v7, v8, v9}.
The leaves are \underline{v7, v8, v9}. 

30
The root is ___.
The height is ___.
The parent of \(v3\) is ___.
The depth of \(v3\) is ___.
The children of \(v6\) are ___.
The ancestors of \(v1\) are ___.
The descendants of \(v6\) are ___.
The leaves are ___.
Every node other than ___ is the root of a subtree.
binary trees
- Binary trees are a special case of a tree in which a node can have AT MOST two children.

- These nodes are designated *left* and *right*.

- In this class we will mostly concentrate on binary trees.

What should the implementation of a binary tree look like? What about a binary tree node?
- Each node has two reference variables
  - One for each of the two children
- If there is no child, the reference is set to null
class BinaryNode<E> {
    E data;
    BinaryNode left;
    BinaryNode right;
}

-what are the values of left and right for a leaf node?

-this is the just the Node class!
    -the BinaryTree class would contain what?
traversing a tree
-traversing a *linked list* is simple
  -there is only one way to go!

-how do we traverse a binary tree if we want to visit every node?
  -eg. we want to print out the data at every node

-how do we decide which direction to take at each node?
depth-first traversal

- to visit every node, go both directions at each node

- trees are recursive in nature

- start at root, recursively traverse the left subtree, then the right subtree

- if the subtree is null, stop (return)
public static void DFT(BinaryNode N) {
    if (N == null)
        return;

    System.out.println(N.data);

    DFT(N.left);
    DFT(N.right);
}

what does this print out?
traversal orders

- pre-order: use the node before traversing its children

- in-order: traverse left child, use node, traverse right child

- post-order: use node after traversing both children
-pre-order:
  use N   // eg. print N
  DFT(N.left);
  DFT(N.right);

-in-order:
  DFT(N.left);
  use N   // eg. print N
  DFT(N.right);

-post-order:
  DFT(N.left);
  DFT(N.right);
  use N   // eg. print N

note: nodes are still traversed in the same order, but “used” (printed) in a different order
EXAMPLE: expression trees
how can we traverse this tree to evaluate the expression?

$$(3 - (15/11)) + (7*2^4)$$
public static double evaluate(Node n) {
    if (n.isLeaf())
        return n.value;

    double leftVal = evaluate(n.left);
    double rightVal = evaluate(n.right);

    switch (n.operator) {
    case '+':
        return leftVal + rightVal;
    case '-':
        return leftVal - rightVal;
    ...
    }
}
public static double evaluate(Node n) {
    if (n.isLeaf())
        return n.value;

double leftVal = evaluate(n.left);
double rightVal = evaluate(n.right);

switch (n.operator) {
    case '+':
        return leftVal + rightVal;
    case '-':
        return leftVal - rightVal;
    ...
}
}

Node class has these fields and method!
DOT format
- DOT is a tool for tree (and graph) visualization
  - it is part of the GraphViz software
  - [http://www.graphviz.org](http://www.graphviz.org)
  - installed on the CADE machines

- DOT is also a file format for trees (and graphs)
  - we can (and will!) write Java code to read them as input to construct a tree, as well as output them from an existing tree for debugging purposes
(simplified) DOT format

-the DOT language as *many* features for specifying the layout of a tree (and graph)

-the simplest format looks like this:

```plaintext
graph myGraph{
    "a" -- "b"
    "a" -- "c"
    "c" -- "g"
    "c" -- "j"
}
```

```
\begin{tikzpicture}
    \node (a) at (0,0) {a};
    \node (b) at (-1,-1) {b};
    \node (c) at (1,-1) {c};
    \node (g) at (0,-2) {g};
    \node (j) at (0,-2) {j};
    \draw (a) -- (b);
    \draw (a) -- (c);
    \draw (c) -- (g);
    \draw (c) -- (j);
\end{tikzpicture}
```
DOT tool

-the CADE Linux machines have the command-line DOT tool installed

dot -Tgif input.dot -o output.gif

-“-Tgif” means create a .gif file as the result
-“-o” means specify the name of the output file
next time…
- **reading**
  - chapters 18 and 19 in book: trees & binary search trees
  - chapter 6
    - [http://opendatastructures.org/ods-java/](http://opendatastructures.org/ods-java/)

- **homework**
  - assignment 7 due Wednesday