CIS 313: Intermediate Data Structure

week of Oct 14

third week of the term
linear data structures

Our basic structures: quick review

• arrays
• linked lists
• stacks
• queues
• priority queue
• binary heap
stacks

• LIFO: last-in first-out
• can implement stack with array, linked list, ...

• uses of stack
  • implement recursion
  • expression evaluation
  • depth-first search

• stack operations
  • push
  • pop
  • top (or peek)
  • init, isEmpty, isFull
example use of stack: evaluate postfix

postfix: operator after the operands
• (2+3)*7 becomes 2 3 + 7 *
• 2+(3*7) is 2 3 7 * +
• no need for parens

- to evaluate a postfix expression E:
  - use operand stack S
  - for each token x in E, scanning L to R
    - if x is operand (value)
      - S.push(x)
    - else x is operator (+, *, -, ...)
      - v=S.pop
      - w=S.pop
      - z = result of applying operator x to (w,v)
      - S.push(z)

  - return S.pop

  - note: if try to pop on empty stack, then underflow error
    and if stack not empty after last pop then overflow error
queues

• FIFO: first-in, first-out
• useful in job scheduling, models “standing in line”
• implementation: linked list, array (wraparound)
• use to compute breadth-first search of tree, graph
• operations
  • enqueue
  • dequeue
  • front, isEmpty, isFull
example with tree: stack vs queue

Consider a tree T consisting of simple nodes p: fields p.left, p.right, and p.value

We have a simple recursive preorder traversal whose initial call is preorderTrav(T.root)

preorderTrav(node p)

  print p.value
  if p.left != null
    preorderTrav(p.left)
  if p.right != null
    preorderTrav(p.right)
example with tree (cont’d)

preorder traversal:
A B D I J F C G K H

note
inorder: I D J B F A G K C H
postorder: I J D F B K G H C A
implement that traversal with a stack:

stack S of node

S.push(T.root)

while (not S.isEmpty)
  p = S.pop
  print p.value
  if p.right!=null
    S.push(p.right)
  if p.left!=null
    S.push(p.left)

note: need to push the right side first so left side gets visited before it

step through traversal with tree on previous slide
implement that traversal with a queue:

queue Q of node
Q.enqueue(T.root)

while (not Q.isEmpty)
  p = Q.dequeue
  print p.value
  if p.right!=null
    Q.enqueue(p.right)
  if p.left!=null
    Q.enqueue(p.left)

what order do we get with this method?
try example
example with tree (conclusion)

previous queue algorithm gives a (reverse) level-order:

A C B H G F D K J I

nice, somewhat unrelated question,
Reconstruct a binary tree from two of the traversal sequences

data example: you are given only

A B D I J F C G K H (preorder)
I D J B F A G K C H (inorder)

now build the tree
priority queues

• chapter 6
• abstract operations (implementation independent)
• maintains a set S of elements
• operations
  • insert(x)
  • max (or returnMax)
  • extractMax (removes it)
  • increaseKey(x,k) (set key of x to a new larger value)
• -OR- insert, min, extractMin, decreaseKey
can sort with priority queue (assuming the descending order)

PQSort(array A)  //array A has n elements
create PQ Q
for i=1 to n
    Q.insert(A[i])
for i = n down to 1
    A[i] = Q.extractMax

cannot analyze time without implementation

How could we implement a PQ as an unordered list?  
As an ordered list? What’s the complexity of each operation?
unordered list implementation of PQ

• simple
• insert(x) is $O(1)$
• extractMax is $O(n)$
• What does PQSort look like?
  • selection sort
  • time $O(n^2)$, work done in second loop
ordered list implementation of PQ

• also simple
• insert(x) is $O(n)$
• extractMax is $O(1)$
• What does PQSort look like?
  • insertion sort
  • time $O(n^2)$, work done in first loop
binary heap implementation of PQ

• most common implementation
• operations are $O(\log n)$
• uses a binary tree structure
• except that the tree is stored in an array with no pointers
• it is an *implicit* tree, children and parents inferred from location in array

• PQSort becomes *heapsort*
binary heap

• stored in array
• item located in position $i$
  • parent in location $[i/2]$
  • left child in position $2i$
  • right child in position $2i + 1$

• tree is complete
  • all nodes have two children, except maybe parent of “last” one

• tree maintains heap property
  • value stored at location $i$ is greater than or equal to values stored in both its children

• fact: a binary heap with $n$ elements has the height of $\lceil \log n \rceil$ (why?)
binary heap insertion

- put new value $x$ at end of array, extending its size by 1
- value $x$ is now viewed as being at the bottom of the tree
- if $x$ violates heap property (if larger than parent), swap with parent
- repeat until no violation
- time is proportional to height of tree, which is $O(lg n)$

- text handles this differently, they insert $-\infty$ and then use heap-increase-key to the new value
pseudo-code for insert

```plaintext
insert(x):

heapsize++
A[heapsize-1] = x

i = heapsize-1
while i>1 and A[i]>A[parent(i)]
    swap A[i] and A[parent(i)]
    i = parent(i)
```

sometimes called “sift-up” or “bubble-up”
Binary Heap: Insert Operation

viewed as a binary tree

viewed as an array

viewed as a binary tree

viewed as an array
heap extract-max (deletion)

• similar but element moves down
• idea: remove and return root (in location 1 of the tree)
• move rightmost element into that empty location ...
• ... and reduce the heapsize
• tree shape is maintained but root location may violate heap property
• note: rest of tree still has heap property
• swap node with larger (why) of it’s children
• repeat while heap property violated until leaf hit
• called “sift-down” or “bubble-down”
**Max-Heapify**

```plaintext
Max-Heapify(A, i)

// Input: A: an array where the left and right children of i root heaps (but i may not), i: an array index
// Output: A modified so that i roots a heap
// Running Time: O(log n) where n = heap-size[A] - i

1. l ← LEFT(i)
2. r ← RIGHT(i)
   4. largest ← l
5. else largest ← i
   7. largest ← r
8. if largest ≠ i
   9. exchange A[i] and A[largest]
10. Max-Heapify(A, largest)
```
first attempt at sorting

1. for each element $x$, \textit{insert} $x$ into a heap
   - time per insert $O(lg\ n)$, total $O(n \ lg\ n)$
   - this can be made much faster

2. while the heap is not empty, \textit{extract-max}
   - output is a sorted list (reversed)
   - each extract-max is $O(lg\ n)$, total $O(n \ lg\ n)$
   - cannot be made faster

BUILDHEAP uses deletion idea to get linear overall time
buildheap code

**BUILD-MAX-HEAP(A)**

// Input: A: an (unsorted) array
// Output: A modified to represent a heap.
// Running Time: O(n) where n = length[A]
1. heap-size[A] ← length[A]
2. for i ← [length[A]/2] downto 1
   3. MAX-HEAPIFY(A, i)

**time analysis**
if tree has height H=\(\log n\)
- all nodes at level \(k\) take time \(H-k\) to sift down
- there are \(2^k\) nodes at level \(k\)
- total time is \(\sum_{0}^{H} 2^k (H - k)\)
- can show this is at most \(2n\)

**correctness**
- idea sort of clear, build heaps bottom up
- text uses loop invariant!!
grinding through the time bound

\[
\sum_{k=0}^{H} 2^k (H - k) = 2^H \sum_{k=0}^{H} \left(\frac{2^k}{2^H}\right)(H - k) \\
= n \cdot \sum_{k=0}^{H} \frac{1}{2^{H-k}} (H - k) \\
= n \cdot \sum_{i=0}^{H} \frac{i}{2^i} \leq n \cdot \sum_{i=0}^{\infty} \frac{i}{2^i} = 2 \cdot n
\]

\[2^H \approx 2^{\log_2 n} = n\]

re-index
Now heapsort

**Heap-Sort(A)**

// Input: A: an (unsorted) array
// Output: A modified to be sorted from smallest to largest
// Running Time: $O(n \log n)$ where $n = \text{length}[A]

1. **Build-Max-Heap(A)**
2. **for** $i = \text{length}[A]$ **downto** 2
   4. $\text{heap-size}[A] \leftarrow \text{heap-size}[A] - 1$
   5. **Max-Heapify(A, 1)**

step 1: $\Theta(n)$ time
steps 2-5: $\Theta(n \log n)$ time
other heap operation: increase-key

• an item can be increased in $O(\lg n)$ time
• after the increase, it would need to be sifted up as in the insert method
• the same applies to the decrease-key operation in a min heap
• this operation is a crucial step in Dijkstra’s method (shortest path) and Prim’s method (minimum spanning tree)
• it can be implemented in $O(1)$ amortized time using Fibonacci heaps

• we will not cover Fibonacci heaps, but next we look at a similar and simpler structure: binomial heaps
<table>
<thead>
<tr>
<th>Procedure</th>
<th>Binary heap (worst-case)</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAKE-HEAP</td>
<td>$\Theta(1)$</td>
</tr>
<tr>
<td>INSERT</td>
<td>$\Theta(\log n)$</td>
</tr>
<tr>
<td>MINIMUM</td>
<td>$\Theta(1)$</td>
</tr>
<tr>
<td>EXTRACT-MIN</td>
<td>$\Theta(\log n)$</td>
</tr>
<tr>
<td>UNION</td>
<td>$\Theta(n)$</td>
</tr>
<tr>
<td>DECREASE-KEY</td>
<td>$\Theta(\log n)$</td>
</tr>
<tr>
<td>DELETE</td>
<td>$\Theta(\log n)$</td>
</tr>
</tbody>
</table>
union of binary heaps

<table>
<thead>
<tr>
<th>Procedure</th>
<th>Binary heap (worst-case)</th>
<th>Binomial heap (worst-case)</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAKE-HEAP</td>
<td>$\Theta(1)$</td>
<td>$\Theta(1)$</td>
</tr>
<tr>
<td>INSERT</td>
<td>$\Theta(\lg n)$</td>
<td>$O(\lg n)$</td>
</tr>
<tr>
<td>MINIMUM</td>
<td>$\Theta(1)$</td>
<td>$O(\lg n)$</td>
</tr>
<tr>
<td>EXTRACT-MIN</td>
<td>$\Theta(\lg n)$</td>
<td>$\Theta(\lg n)$</td>
</tr>
<tr>
<td>UNION</td>
<td>$\Theta(n)$</td>
<td>$O(\lg n)$</td>
</tr>
<tr>
<td>DECREASE-KEY</td>
<td>$\Theta(\lg n)$</td>
<td>$\Theta(\lg n)$</td>
</tr>
<tr>
<td>DELETE</td>
<td>$\Theta(\lg n)$</td>
<td>$\Theta(\lg n)$</td>
</tr>
</tbody>
</table>