CIS 313: Intermediate Data Structure

week of Jan 14

second week of the term
algorithm time bounds

Let $\mathcal{A}$ be some algorithm operating on an input $x$

- worst case
  - $\mathcal{A}$ has worst case time $O(t(n))$ if there are constants $c$ and $N$ such that for all $n>N$ and all inputs $x$ of length $n$, $\mathcal{A}$ completes its computation on input $x$ using at most $c \cdot t(n)$ steps
  - $\mathcal{A}$ has worst case time $\Omega(t(n))$ if there are constants $c$ and $N$ such that for all $n>N$, there exists an input $x$ of length $n$ such that $\mathcal{A}$ uses at least $c \cdot t(n)$ steps to finish its computation on $x$

- average case
- expected case *(a measure that makes sense if algorithm is randomized)*
- best case *(not very useful)*
- smoothed analysis *(complicated)*
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 breath-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 $\text{preorderTrav}(T.root)$

```
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

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
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 PA

- 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 \( \lfloor \lg n \rfloor \) (why?)