CIS 313

week of Oct 1

second week of the term
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 – why?)*
- smoothed analysis *(complicated)*
linear data structures

Our basic structures: quick review

• arrays
• linked lists
• stacks
• queues
• hash tables
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:

1. Use operand stack S

2. 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)

3. 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
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)

```python
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
example with tree (cont’d)

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
example with tree (cont’d)

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

stack S -> queue Q
pop -> dequeue
push -> enqueue
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
hash tables

• chapter 11
• store n items in a table T of size m
• hash function h determines where to put an item
• issues
  • what to do when two items hash to same location (collision)
  • how to choose good hash function h (minimize collisions)
  • how to choose table size m
  • dynamically increase table size
    • important in databases but not addressed here
collision resolution

• what to do with two items x and y that hash to same location?

h(x.key) = h(y.key)

• open addressing
  • look at other locations in the table
  • table might overflow
  • more complicated

• closed addressing
  • all items that hash to location t stay there in some structure
  • bucket, linked list, ...
chaining

- first: simple version of chaining
- table $T$ with $m$ slots, each containing a linked list
- hash function $h$ maps keys to $\{0, 1, \ldots, m-1\}$
- $\text{insert}(T, x)$: put $x$ in a node at the head of $T(h(x.\text{key}))$
- load factor: $\alpha = n/m$, where $T$ contains $n$ items
- goal: access time should be $1 + \Theta(\alpha)$
- also called *closed addressing* (since item stored at that location)
choosing a hash function

• let k be the key and T a table of size m
• want h(k) to distribute keys uniformly across locations \{0,1,...,m-1\}
• division method: \( h(k) = k \mod m \)
  • choice of table size m important
  • if \( m=2^p \), then only low order bits of k matter
  • if k not distributed well, then h(k) prone to be biased
  • best if m a prime
multiplication method

• pick constant $A$ with $0 < A < 1$

• $h(k) = \lfloor m \cdot ((k \cdot A) \mod 1) \rfloor$ (here “mod 1” means fractional part of real number)

• Knuth suggests $A = \frac{\sqrt{5} - 1}{2} \approx 0.6180339 ...$

• nice example on p 264 of text
universal hashing

- family of hash functions $\mathcal{H}$, maps key universe $U$ onto $\{0, 1, \ldots, m-1\}$
- want for any $k, l \in U$ that the number of $h \in \mathcal{H}$ such that $h(k) = h(l)$ is at most $|\mathcal{H}|/m$
- idea is to pick an $h \in \mathcal{H}$ randomly if possible
- intuitively if keys $k \in U$ not distributed well a random $h \in \mathcal{H}$ will still distribute the locations well and excess avoid collision
- example family: $\mathcal{H}$ will depend on fixed $p, m$
  - $m$ is table size, $p > m$ is a prime so that all keys $k < p$
  - choose $a, b$ with $0 < a < p$, $b < p$ (randomly)
  - $h(k) = ((ak+b) \mod p) \mod m$
  - many details in text, depends on basic number theory
back to collision resolution: open addressing

• for key k=x.key, if location T[h(k)] is full (via collision), need to put x in a different location

• look in a sequence of locations, this is called the probe sequence

• sometimes write h<k,i> for location of probe i on key k

• look in locations h<k,0>, h<k,1>, h<k,2>, ... until find empty slot in which to place x

• simplest (and worst): linear probing
  • h<k,i> = (h(k)+i) mod m
  • that is, if h(k) is full, look in locations h(k)+1, h(k)+2, h(k)+3, ...
better probe sequences

• quadratic probing
  • pick constants c, d
  • $h_{<k,i>} = (h(k) + c*i + d*i^2) \mod m$
  • c, d, m need to be chosen carefully so that $h_{<k,i>}$ can probe entire table

• double hashing
  • use two hash functions $h_1$, $h_2$
  • $h_{<k,i>} = (h_1(k) + i*h_2(k)) \mod m$
  • need m and $h_2(k)$ to be relatively prime
other uses of hash functions

• database indexing
  • need extendible hash tables as many insertions happen
  • not good for range queries (“find all values between a and b”)
  • B-tree indexes more popular

• cryptographically secure hashing
  • password files
  • multi-party communication
  • hash functions very different looking

• Bloom filters, count-min sketch
count-min sketch

• problem: count events in a data-stream, many possible events, want number of occurrences of each event
• conventional data structure too large
• count-min sketch is probabilistic structure, uses sub-linear space
• idea: table size of w columns and d rows
• each row j associates with hash function $h_j$ mapping to $\{0,1,...,w-1\}$
• when event e occurs, increment location $[j, h_j(e)]$
• estimate of number of occurrences of a is the min of all locations $[j, h_j(e)]$
count-min sketch (cont’d)

Algorithm 1 Count-Min Sketch

insert(x):
for $i = 1$ to $d$ do
    $M[i, h_i(x)] \leftarrow M[i, h_i(x)] + 1$
end for

query(x):
$c = \min \{M[i, h_i(x)] \text{ for all } 1 \leq i \leq d\}$
return $c$

![Count-Min Sketch Diagram]