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 – why?)
• smoothed analysis (complicated)
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

<table>
<thead>
<tr>
<th>postfix: operator after the operands</th>
</tr>
</thead>
<tbody>
<tr>
<td>• (2+3)*7 becomes 2 3 + 7 *</td>
</tr>
<tr>
<td>• 2+(3*7) is 2 3 7 * +</td>
</tr>
<tr>
<td>• no need for parens</td>
</tr>
</tbody>
</table>

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
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
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
  • $\text{insert}(x)$
  • $\text{max}$ (or $\text{returnMax}$)
  • $\text{extractMax}$ (removes it)
  • $\text{increaseKey}(x,k)$ (set key of $x$ to a new larger value)
  • -OR- $\text{insert}$, $\text{min}$, $\text{extractMin}$, $\text{decreaseKey}$
can sort with priority queue (assuming the descending order)

```plaintext
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 $\lceil \log n \rceil$ (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

```
insert(x):
    heapsize++
    A[heapsize]=x
    i = heapsize
    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

1. Insert 16
2. Insert 14

Viewed as a binary tree:

- Left:
  - 16
  - 11 (2)
  - 8
- Right:
  - 16
  - 12 (3)
  - 7

Viewed as an array:

- Left:
  - 1
  - 2
  - 3
  - 4
  - 5
  - 6
  - 7
  - 16
  - 11
  - 12
  - 8
- Right:
  - 1
  - 2
  - 3
  - 4
  - 5
  - 6
  - 7
  - 16
  - 11
  - 14
  - 8
  - 10
  - 9
  - 12
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(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  
2  
3  if $l \leq heap-size[A]$ and $A[l] > A[i]$ 
4    largest ← l
5  else largest ← i
6  if $r \leq heap-size[A]$ and $A[r] < A[largest]$ 
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 $x$, 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, 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)
```

correctness
- idea sort of clear, build heaps bottom up
- text uses loop invariant!!

time analysis
if tree has height H=\(\lg n\)
- 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 \(2n\)
grinding through the time bound

\[ \sum_{k=0}^{H} 2^k (H - k) = 2^H \sum_{k=0}^{H} \left( \frac{2^k}{2^H} \right) (H - k) \]

\[ = n \cdot \sum_{k=0}^{H} \frac{1}{2^{H-k}} (H - k) \]

\[ = n \cdot \sum_{i=0}^{\infty} \frac{i}{2^i} \leq n \cdot \sum_{i=0}^{\infty} \frac{i}{2^i} = 2 \cdot n \]

why just 2?
- mentioned but not proved in appendix
- “fun” to derive
- can also take derivative of \( \sum_{k=0}^{\infty} x^k = \frac{1}{1-x} \)
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
3. 
   exchange $A[1]$ and $A[i]$
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>$\Theta(\lg n)$</td>
</tr>
<tr>
<td>MINIMUM</td>
<td>$\Theta(1)$</td>
</tr>
<tr>
<td>EXTRACT-MIN</td>
<td>$\Theta(\lg n)$</td>
</tr>
<tr>
<td>UNION</td>
<td>$\Theta(n)$</td>
</tr>
<tr>
<td>DECREASE-KEY</td>
<td>$\Theta(\lg n)$</td>
</tr>
<tr>
<td>DELETE</td>
<td>$\Theta(\lg n)$</td>
</tr>
</tbody>
</table>