Assignment 2

Alex Baker

02/18/2016

CS 362

Problem 1:

A stack can be implemented using two stacks. Let these two stacks be called q\_in and q\_out. A queue is a first in first out (FIFO) data structure, while a stack is a last in first out (LIFO) data structure. This is why two stacks are needed, in order to reverse the order of the stack to pop out the first element first in the stack. To add an element to the queue, that data element will be pushed onto stack q\_in. To remove an element, pop an element from q\_out. If q\_out is empty, then pop all elements on q\_in and push them onto q\_out, and then proceed by popping elements off q\_out.

Pseudo Code:

Stack q\_in = empty;

Stack q\_out = empty;

Function add(elem):

q\_in.push(elem)

return

Function remove():

if q\_out is empty:

while q\_in is not empty:

temp = q\_in.pop()

q\_out.push(temp)

end while

end if

return q\_out.pop()

Working Example:

Lets add elements 1, 2, 3. By adding them in this order, they should be removed in this order as well. Adding the three elements is just three push operations from the pseudo code. So q\_in will look like this:

q\_in = 1, 2, 3

If we wanted to remove two elements, we should get back 1 and 2. Since q\_out is currently empty, the elements in q\_in are moved to q\_out:

q\_out = 3, 2, 1

Then to get the two elements, we pop two elements off of q\_out which are 1 and then 2. The data structure is now:

q\_in = empty

q\_out = 3

If we add elements 4 and 5 to the queue, the stack will become:

q\_in = 4, 5

q\_out = 3

if we want to remove an element, this only requires one pop since q\_out is not empty, so the element will be 3, and the stacks will look like:

q\_in = 4, 5

q\_out = empty

Since q\_out is empty now, if another remove is called, then q\_in will be transferred to q\_out and then elements can be popped off of q\_out.

Time Complexity:

Adding an element to the queue requires one push onto stack q\_in, so this is O(1) or constant time.

Removing an element has two scenarios, if q\_out is empty, and if q\_out has elements in it. If q\_out has elements in it, then the remove is just on pop call, which is constant time. If q\_out is empty, then q\_in will need to be transferred to q\_out. If q\_in has n elements, then it will take n pops from q\_in and n pushes onto q\_out, followed by one last pop from q\_out, so this time will be 2n+1. So removing an element from a queue is O(n).

Problem 2:

Merging two heaps can be done in O(n+m). Let h1 be the first heap with n elements and h2 be the second heap with m elements. Assume both heaps h1 and h2 are implemented efficiently using arrays. Then h1 and h2 can be merged into the heap H by merging the two arrays, and then build the heap property from the merged array. From class, a heap can be build from an array in O(n), where n is the number of elements in the array. In the case of heap H, is has n+m elements from the two arrays of size n and m. So this algorithm runs in O(n+m) time.

Algorithm:

Array h1 = length of n keys

Array h2 = length of m keys

Function merge(array h1, array h2)

Array h3 = length of n+m keys

Copy h1 and h2 keys into h3

Heapify(h3)

Return h3

Example:

h1 = [5, 3, 1]

h2 = [6, 4, 2]

h3 = [5, 3, 1, 6, 4, 2]

heapify(h3) = [6, 5, 2, 3, 4, 1]

Time Complexity:

Merging two arrays takes n+m time if n and m are the sizes of the two arrays. This is because each element of each array needs to be copied once. Once the arrays are merge into a larger n+m array, heapify can run on the larger array which was shown in class to run in O(n+m) time. So the time to merge two heaps stored in an array takes 2n+2m time, or O(n+m).

Problem 3:

Claim:

Yes, the AVL tree is still bounded by O(log n)

Let h be the height of the AVL tree and n be the number of nodes in the tree.

Proof: Induction

A proof using induction was discussed in class for when the height difference is the same and at most one, so this proof will only cover when the height difference is two.

Base:

h = 0, n = 1

Induction Step:

Assume for all AVL trees of height the claim is true. Need to show this is also true for

case: left sub tree has height k and right sub tree has height k-2

Problem 4:

For a data set with most duplicates, counting sort would provide the best time complexity since it can reduce the number of inputs that need to be sorted. Counting sort runs in O(n+k) time where n is the total number of elements and k is the number of distinct elements to be sorted. After counting sort runs, a typical comparison sort can be run on the k distinct elements. Since k << n this will have a faster running time than a tradition comparison sort which runs O(n log n). So O(k log k) << O(n log n) since k << n.

Problem 5:

Yes, Divide and conquer will still work. P and Q can be decomposed into polynomials based on their index.

Then:

From here, the divide and conquer algorithm can be applied the same as in the high/low method.

Problem 6:

Problem 7:

The radix sort algorithm can leverage the counting sort algorithm to sort the integers by length, and then using radix sort to sort the different digits of the same length. After radix sort is finished, the results can be concatenated together, since numbers with more digits are larger than numbers with smaller digits, assuming there is no padding. As described in problem 4, counting sort would sort the digits by length in O(n + k) time where n is the total number of digits and k is the number of digits with different lengths. Radix sort will run k times, once for each set of different length digits. Radix sort runs is O(wn) where n is the number of digits in each number. So running counting sort followed by k radix sorts would have a running time of O(k+n) + kO(wn).

Problem 8:

To find the set of all overlapping intervals, first sort the intervals in increasing order by start point. This can be done in O(n log n) time. Next, iterate over the list comparing the end point of the current interval the starting point of the next interval. Since the intervals are sorted, this can be done in one iteration because if the next interval in the list does not overlap with the current interval, then all later intervals cannot overlap either because their start point is later than the end point.

Algorithm:

sort(intervals by start point)

for i = 0 to number of intervals – 1:

if interval[i][1] >= interval[i+1][0]

print interval[i] and interval[i+1]

Example:

intervals = [{1, 4}, {2, 3}, {5, 6}, {3, 4}]

sort(intervals) = [{1, 4}, {2, 3}, {3, 4}, {5, 6}]

Then comparing {1, 4} and {2, 3} will overlap and {2, 3} and {3, 4} will overlap

Time Complexity:

Sorting can be done in O(n log n) time. The loop only iterates through the intervals once, which takes n time. So this runs in O(n log n + n) = O(n log n)

Problem 9:

Let A be the input sequence of n digits. bestSoFar is an array of length n which holds values of the longest increase subsequence ending at the element in A at the same index, and the array startedFrom be the index where the subsequence started from. To calculate the value at index i in bestSoFar, all of the elements where j < i will need to be checked for increasing order. First, dynamically check the index j in bestSoFar to see if ending at the new element will make a longer subsequence than before. It also needs to be checked if A[j] <= A[i] to make sure the subsequence is increasing. The index of the largest value in bestSoFar in the ending index of the longest increasing subsequence, and the value of startedFrom at this index is the starting index in A of the subsequence.

Pseudo Code:

input: Array A of size n

initialize: bestSoFar, startedFrom to be arrays of length n

endIndex = 0 is the index of the ending of the best subsequence

maxLength = 0

bestSoFar[0] = 1

startedFrom[0] = 0

for i = 1 to n-1:

bestSoFar[i] = 1

startedFrom[i] = i

for j = i-1 down to 0:

if bestSoFar[j] +1 > bestSoFar[i] and A[j] <= A[i]:

bestSoFar[i] = bestSoFar[j] + 1

startedFrom[i] = j

endif

endfor

if bestSoFar[i] > maxLength:

endIndex = i

maxLength = bestSoFar[i]

endif

endfor

return A[startedFrom[endIndex]] to A[endIndex]

Example:

A = [2, 1, 4, 3]

The outerloop starts at i = 0, since this is the first index, the second loop doesn’t run, and the end index gets set to 0.

When i = 1, we start the second loop at j = 0

The first comparison checks whether bestSoFar[j]+1 > bestSoFar[i], so 1+1 > 1, which is true, however the second comparison fails since 2 is not less than 1.

i = 2, j = 1

bestSoFar[1] + 1 > bestSoFar[2]; 1+1 > 1, so this passes. 1 is less than 4 so we enter the ifstatement.

bestSoFar[2] = bestSoFar[1]+1 = 2

startedFrom[2] = 1

i = 2, j = 0

The comparison bestSoFar[0]+1 > bestSoFar[2] fails since 2>2 fails.

the inner loop exits

endIndex = 2

maxLength = 2

i = 3, j = 2-0

All of these comparisons will fail the inner loop since 4 is not less than 3

So the endIndex is 2, and the start index is startedFrom[2] = 1. So the largest increasing subsequence in A is from A[1]-A[2] = [1, 4]

Time Complexity:

This algorithm runs in O(n^2) since the nested loops run at most n times each.

Problem 10:

Let there be two knapsacks, . Then introduce a two dimensional indicator indicating which element goes to which knapsack, so

So:

KP(Ki,n) =

max {

KP(K,n),

KP(Ki-Kn, n-1) + Cn,

KP(Kj-Kn, n-1) + Cn

}