INTRO TO ALGORITHM, CSE 241

PROFESSOR ZHANG

Homework 4

Peiyun(Seed) Zeng

EN:437801

Due date: November 13, 2014

PROBLEM 1

Problem 4-1. Skip list

We are given a sequence of five integers of 4, 6, 2, 3, 9, in this order, and want to store them in a skip list. In building the skip list, we use a fair coin, and promote an integer to the next level of the skip list if and only if we get a heads (or "1") by flipping the coin. Assume the sequence of coin flips is 10011001011001 $\cdots$.

(a) Show the skip list after inserting 4, 2 and 9.

(b) In class, I discussed an algorithm for building a skip list of n elements in $O(n^2)$ time in the worst case. Using the randomized method we discussed in class, design a randomized skip list building algorithm. Analyze the time complexity of your algorithm.

Solution:

(b):

Abstract

To build a skip list of n elements, we are essentially insert each element to the "skip-list so far". So the efficiency of the algorithm depends all on the insertion method. essentially, Insertion is gonna first search (find) the correct position. Then we put the element there, then we flip our coin to choose how many times we want to promote the element. That is the basic element.

High-Level Design

Two Complementary Method–Search and Insert

1. Find Method. Find(S,e). S as the skip list. e as the element
   Find(S,e) will find the correct position to put the element at the bottom level
   (i) start from the highest level, leftest element.
   (ii) move right until it hit the first element that is bigger or equal at position j+1. record the position of element j and step down one level.
   (iii) do (ii) to (iii) (p will be updated at each step-down) at each level until it hits the ground level
   (iv) final value of p is the correct position
2. Insert Method, Insert(S,e). S as the skip list, e as the element
   (i) Search for the correct position at the bottom level of the skip list, i
   (ii) put the element at i
   (iii) Randomly decide how many levels the element should be promoted (flip the fair coin, as what we did previously. we are gonna use a random number generator instead of flipping coin physically)

The main function

The main function is fairly simple. It will just be a FOR loop that calls insert method n time

```
For i = 1 to n
    Insert(s,e_i)
```

Proof Of Correctness

The correctness of **Search method** is straightforward since it is essentially exhausting all possible positions for putting element e. Through starting from the leftest element at highest level, Find is able to skip some of the elements because it is sorted at each level and each level is linked (features of a skip list)

Now we know Find method is working properly, **Insert** will invoke FIND to find the correct position for element at bottom level of the skip list. Then put it at the position. After all that, It will promote the element by coin flipping mechanism

Then through insert each element one by one via a FOR LOOP, a skip list can be constructed. The for loop clearly terminates.
Thus **Total correctness is proved**

Time Complexity

This is the fun part. What is the time complexity for FIND method? Remember we have discussed in class, in a randomized skip list, the **Search method**, where it will search for a certain element cost $O(\log(n))$ with high probability (WHP in short). FIND method is an identical operation, except it will always find a position while SEARCH will return NONE if element is not int the skip list. So we can use the result we discussed in class. Find will cost $O(\log(n))$ WHP.

Now the real mystery lies in the insert method. After we put the element at at the bottom level, how many times it will get promoted. Assume there is 1/2 chance to promote

$$P[\text{e is at level 1 (bottom level)}] = 1/2 \tag{1}$$

$$P[\text{e is at level 2}] = 1/4 \tag{2}$$

$$P[\text{e is at level k}] = 1/2^k \tag{3}$$

$$P[\text{e is at clog(n) level}] = 1/2^(c * \log(n)) = 1/n^c \tag{4}$$

$$\tag{5}$$

According to the Definition [given in class] of W.H.P, $1/n^c$ is a rather small probability. So it is WHP, element e will promoted no more than O(log(n)) level. Thus the total running time for insert $T_1(n) = O(log(n))$[from search]$+O(log(n)) = O(log(n))$

The main program utilizes a for loop to invoke the insert method n times so Total complexity $\underline{T(n) = n * T_1(n) = O(n * log(n))}$

INTRO TO ALGORITHM, CSE 241

PROFESSOR ZHANG

Homework 4

Peiyun(Seed) Zeng

EN:437801

Due date: November 13, 2014

PROBLEM 2

Harry Potter, the child wizard of Hogwarts fame, has once again run into trouble. Professor Snape has sent Harry to detention and assigned him the task of sorting all the old homework assignments from the last 200 years. Being a wizard, Harry waves his wand and says, ordinatus sortitus, and the papers rapidly pile themselves in order.

Harry is not a particularly competent spell caster, and his spell doesn't work completely. Instead of sorting the papers correctly, each paper was within k slots of its proper position. We now devise a faster algorithm to sort Harry's almost sorted list in $\Theta(n \log k)$ time. For the solution to the problem, we are going to use heaps. Recall that a heap supports the following operations:

(1): MAKE-HEAP() returns a new empty heap.

(2): INSERT(H, key, value) inserts the key/value pair into the heap.

(3): EXTRACT-MIN (H) removes the key/value pair with the smallest key from the heap, returning the value.

Each of these operations run in log k time where k is the number of elements in the heap. (In

class, we learned max-heaps. Min-heaps work in the same manner.)
(a) First, consider the problem of merging t sorted lists. Assume we have lists A1, . . . , At, each of which is sorted. Give an efficient algorithm to merge these lists. The algorithm should run in $n * \log(t)$ time, where n is the total number of elements in all the lists combined. Note that all the lists need not be of the same size. Hint: Use the heaps.

Solution:

Preparation

Before everything, for the convenience of analyzing the complexity. let's specify what is the time complexity of each operations should be

Note: n here is the number of element in the heap

- MAKE-HEAP()
  For the MAKE-HEAP(), as we discussed in class, it will cost us O(n) time
- INSERT(H,key,value)
  in the INSERT, we are gonna put the element at the end of the array. A violation might be created, we can push the violation upward. At the worst case, this element need all the way to the top level, which will need $\log(n)$ time.
- EXTRACT-MIN(H)
  This will cost $\log(n)$ time as we discussed in class.

Introduction

So we are gonna construct a heap H first. The element will have a **key that is the smallest(first essentially) element of a list** (ex. A_i). The value of the element in the heap will be **the index of the list** (ex. i for list A_i). To merge all the sorted list to get the final result, we just have to extract from the heap, then insert the next element(current minimum) in a certain sorted list back to the heap. We do this process n times, we can get the final sorted list.

High-Level Design

Note: The heap H in this problem is gonna be a MIN-HEAP

1. Build heap H using MAKE-HEAP
2. Initially, insert the first (smallest) element of each sorted list (A_i) to the heap by using INSERT(H, $A_i[1]$,i)
3. now we construct a FOR loop that has n iterations, each iteration will have following operations

a) Extract the first (smallest) element e_1 from the heap using EXTRACT-MIN(H). e_1 has a value i that is the index of a certain sorted list

b) put the key of e_1 to a new array A

c) e_1 has a value i, that is the index for an array A_i

d) after e_1 has been extracted from heap, we insert the next element e_2 in the same sorted array A_i back to the heap. Thus the heap will hold the smallest element of all arrays $[A_1...A_t]$ again at the end of each iteration.

4. return Array A

Proof of correctness

The **loop invariant (LI)** in this case is a fact that **heap will hold the smallest element of all arrays** $[A_1...A_t]$**.**

At the last step inside our FOR loop, we will insert back the next element e_2 of the sorted array A_i back to the heap. Thus, LI is maintained at the end of each iteration.

The FOR loop has n iterations, since LI is maintained, heap will always hold the smallest element of all sorted lists. At each iteration, step (a) inside the for loop will extract the smallest element and (b) will put it to a new array.Since every time we extract the next smallest element from the heap, the new array A will always be sorted.

After n iterations, at termination, a sorted New array A with all n elements will be the total sorted list. Also, the for loop will obviously terminate. The **total correctness** is then proved.

Time Complexity

The time complexity is rather straightforward.

- Assume MAKE-HEAP costs O(n) time[Discussed in Class].
- Other steps outside the for loop only costs constant time.
- the FOR loop has n iterations, at each iteration
  - (a) the EXTRACT-MIN will cost $O(\log(t))$,since there are only t elements in the heap. [Discussed in class].
  - (d) Insertion will also cost $O(\log(t))$ time.[discussed above]
- Thus $T(n) = O(n) + O(n * \log(t)) + O(1) = \underline{O(n * \log(t))}$

(b) Apply the above algorithm to sort the Harry's almost sorted array in n lg k time. Hint: Try to convert the Harry's almost sorted array into 2k sorted lists and then use the above algorithm to merge them.

Solution:

Introduction Silly Harry has accomplished to almost-sort the array. That is to say, any element(or paper, in Harry's case) **e** in the current array is in the right position within k slots. Therefore, for any A[i] in the array, $i \leq n-2k-1$, we have property $A[i] \leq A[i+2k+1]$. Using this property, we can partition the original array to 2k number of sorted arrays.

$$A_1 = \{A[1], A[2k+1], A[4k+1], ...A[n-2k-1]\}$$
$$A_2 = \{A[2], A[2k+2], A[4k+2], ...A[n-2k-2]\}$$
$$A_3 = \{A[3], A[2k+3], A[4k+3], ...A[n-2k-3]\}$$
$$.$$
$$.$$
$$.$$
$$A_{2k} = \{A[2k], A[4k], A[6k], ..., A[n]\}$$

Therefore, this problem is reduced to the exact problem as the previous one. Now we have 2k number of sorted arrays, we just need to merge them(like we did in previous question.)

High-Level Design

This is gonna be embarrassingly similar to the algorithm of the last problem, since they are essentially the same algorithm.

1. Build heap H using MAKE-HEAP
2. Initially, insert the first (smallest) element of each sorted list (**in this case, it is element A[i] to A[2k]**) to the heap by using INSERT(H,$A[i]$,i)
3. now we construct a FOR loop that has n iterations, each iteration will have following operations
   a) Extract the first (smallest) element e_1 from the heap using EXTRACT-MIN(H). e_1 has a value i that is the index of a certain sorted list
   b) put the key of e_1 to a new array A

c) if not the end of list,that is if $i+2k \leq n$,we insert A[i+2k] back to the heap. A[i+2k], as we shown above, is the next element in the same sorted array as e_1

4. return array A

Pseudo Code

```
Sort-Almost-Sorted(B,n,k)
1.    H = MAKE-HEAP()
2.    A = []  #Created new Array A
3.    for i = 1 to 2k
4.        INSERT(H,A[i],i)
5.    for j = 1 to n
6.        e = EXTRACT-Min(H)
7.        A[i] = B[j]
8.        if j+2k <= n
9.            INSERT(H,A[j+2k],j)
10. return A
```

Proof of Correctness

It is an identical argument as the previous algorithm. So we start from LI and proves it is maintained. Then LI combined with the termination condition, our post-condition will be satisfied. The only thing changed is the number of sorted arrays changed from t to 2k.

Time Complexity

In the analysis for previous problem, the final complexity is $T(n) = O(n * \log(t))$, in this case t is changed to 2k. thus the time complexity here should be $T(n) = O(n * \log(2k)) = O(n * \log(k))$