**2019-2020 Algos Exam**

\*Some proposed solutions may have errors, please comment / correct if so

**Question 1**

1. **i.** Best case b.key = k

Worst case: k is not in B

For any input, Ω (1) - in best case, O(N) - has to check every key, omega DNE

**ii.** Auxiliary space best case = Ω (1)

Worst case = O(N)

T(N) = omega(1) if N=1

T(N) = T(N-1) + d if N>1

T(N-1) = T(N-2) + 2d if N>2

T(N) = T(N-i) + id if N>1, let i=N-1

T(N) = T(1) + (N-1)d

T(N) = O(N)

1. i) Best Case Ω (N)

Worst Case O(N+sum(N+N-1+N-2....etc)) = O(N^2)

ii) Best Case Ω (1)

Worst Case O(N + N) = O(2N)

Space used for copying trees + recursive calls

**Question 2**

Procedure ranked(A,L,R,k)

P = partition(A, L, R)

If k==p

return A[k]

Else if k>p

return ranked(A,P+1,R,k-P)

Else if k<p

return ranked(A,L,P,k)

End if

End procedure

// alternatively could order all L to R-1 and return A[k]

1. i) Best case the partition hits k on first go, T(N) = Ω(M)

ii) Worst case is O(N\*(N-1)) = O(N^2)

iii) Worst case in even partition divide: O(log2(N)\*(N-1))=O(NlogN)

**Question 3**

a)

i. T(N,m=fixed table size) = theta(N/m), (7.10 in notes)

ii. T(N) = amortised O(1) for a resizeable table (7.12) (This should be O(1), not amortized O(1) (asking for expected time to search) -> check lecture. The expected time to search in a resizable table is O(a) = O(1), having N/M below some constant a (accounting method). The time for inserting a new key in that resizable table indeed will be amortized O(1))

b) i. the load factor of the table is always less than or equal to 1 because with probing the elements are stored directly in the table positions; N/m therefore cannot be greater than 1.

N = number of elements

M = number of table positions

Therefore N can never be greater than M

ii. Other functions include search. The search function is dependent on the probe sequence; if there is a NIL changed in an existing space which has positions after it, it you won’t be able to find the keys in the table after that newly deleted space.

c)

i. Worst case in dequeue when out is empty theta(N) because you pop N times and push N times theta(2N) = theta(N)

**Amortised Cost Method:**

|  |  |
| --- | --- |
| In stack | Out Stack |
| 4 |  |
| 3 |  |
| 2 |  |
| 1 |  |

Enqueue 4 items, Current Cost = 4

|  |  |
| --- | --- |
| In stack | Out Stack |
|  |  |
|  | 2 |
|  | 3 |
|  | 4 |

Dequeue item 1, pop 4 items from In + push 4 items to Out + dequeue item 1

Current Cost = 4 + 4 + 1

Dequeue each other item (will not pop and push)

Current Cost = 3

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **EX1: Total amortised cost** | **Steps** | | **Cost** | | **Total Cost** |
| a | EQ 1 | | 1 | | 1 |
| 2a | EQ 2 | | 1 | | 2 |
| 3a | EQ 3 | | 1 | | 3 |
| 4a | EQ 4 | | 1 | | 4 |
| 5a | DQ 1 | | 4+4+1=9 | | 13 |
| 6a | DQ 2 | | 1 | | 14 |
| 7a | DQ 3 | | 1 | | 15 |
| 8a | DQ 4 | | 1 | | 16 |
| **EX2: Steps** | | **Cost** | | **Total Cost** | |
| EQ 1 | | 1 | | 1 | |
| EQ 2 | | 1 | | 2 | |
| EQ 3 | | 1 | | 3 | |
| DQ 1 | | 3+3+1=7 | | 10 | |
| DQ 2 | | 1 | | 11 | |
| DQ 3 | | 1 | | 12 | |

Choose a=3 s.t. it satisfies the values in the tables above (able to pay back all total costs in any period t)

Constant as op increases

**Accounting Method:**

We are paying 1 to push x to the in stack and then prepaying 2 for it to be popped from in and pushing into the out stack. The dequeue operation will pay the remaining 1 for it to be popped out of the out stack for the final time. Thus, a=3 is sufficient.

T(N) of any operation (EQ/DQ) = amortised theta(1) because a constant amortised cost a=3 is always sufficient

**Via Potential method:**

We’ll define . This satisfies the desirable properties of , i.e.:

* , as initially the In stack is size 0.
* , as the In stack size is obviously never negative
* Low-cost operations (ENQUEUE) increase , (increases it by 1)
* High-cost operations (DEQUEUE) decrease , (reduce it to 0)

Now we must analyse for all operations. Suppose in all cases that the before-state is an In stack of size N. There are three cases:

* The operation is an ENQUEUE, in which case:
* The operation is a DEQUEUE and Out is empty, in which case:
  + , the actual cost is the cost of a push/pop pair for each element and the final pop.
  + , as In is cleared during this DEQUEUE, and ,
  + Hence,
* The operation is a DEQUEUE and Out is NOT empty, in which case:
  + the actual cost is just the cost of the POP.
  + , since the In stack doesn’t change size, so .,
  + Hence,

So we’ve shown that we can find a , such that it’s bounded by a constant () for all operations, so it’s amortised constant time.

**Question 4**

i. Greedy algo will not check all possible solutions so is faster

(However, this only works if the greedy choice is guaranteed to find an optimal solution, so not always preferred)

ii. Disprove:

If they choose to watch the shortest film f1, they will not be able to watch f2 or f3 and watch 1 film total. However, the best optimisation is watching f2 and f3, so 2 total films max

|  |  |  |  |
| --- | --- | --- | --- |
|  | f1 | f2 | f3 |
| f.start | 3 | 1 | 4 |
| f.end | 5 | 4 | 8 |

iii.

~~O(2~~~~N~~~~) - same as naïve recursion soln in dynamic programming. Need to use a top-down (memoisation – store already calculated values) or bottom up approach (solve smallest subproblems and build upwards).~~

<https://www.cs.princeton.edu/~wayne/kleinberg-tardos/pearson/04GreedyAlgorithms-2x2.pdf,> slides 5-8

Time complexity is O(N^2)

Iterates through all N options and selects earliest finishing film that does not overlap. Then calls again with remaining N-1 options. Then N-2, etc... until all have been checked. Thus, recurrence relation is (same as worst case of insertion sort, p28 of notes):

T(N) = T(N-1)+d --> 1+2+…+(N-1) = N(N-1)/2 which is equal to O(N^2)

This can be improved by sorting by finish time first (which would take O(NlogN) time), and then searching through this list would take theta(N) time, this the total time taken would become O(NlogN) as the algo is dominated by the searching time.

iv.

**Theorem:** Let there be some either empty or partially decided list of films, Ft , where all films up to and including time t have been chosen (e.g. t could be 0). Let fk be the film with the earliest end-time such that it’s start time is on or after t. Then I claim that there is an optimal solution that includes the list of films Ft as described,and the film fk.

**Proof:** Let S be an optimal solution that includes the partial solution Ft. If S includes fk then we’re done. So, assume that it doesn’t.

If it doesn’t then there must be a film in S, fj, such that fj starts on or after t, and ends on or after fk by definition of fk (i.e. since it’s the earliest finisher). Obviously, we can swap out this film fj for our fk and it will still be optimal, as it doesn’t affect any choices made after the end time of fj.

Hence, the greedy choice is correct.

1. i. dynamic programming means making a space-time trade off and storing subproblem solutions so that they can be looked up, rather than resolving them, when they are needed again

ii.

F(N,t1,t2) = 0, N=0 or t2-t1=0

F(N,t1,t2) = 1, N=1

F(N,t1,t2) = 1+max(F(N-1,t1+time(earliest ending movie),t2) , N>1

Not 100%, feels ok but any issues pls comment

, if

, if

, otherwise

Please do not hesitate to comment if there’s any error

