Design and Analysis of Algorithms - IV

Adnan Jakati - Sai Nishanth Mettu

October 2023

1 Solution

Consider the following sequence of items' keys:

$$79, 35, 30, 42, 31, 39, 21, 54, 14, 71, 98, 97, 18, 41, 80, 29$$

Insert the items in the given order into an initially empty hash table with chaining (for collision resolution) using the hash function $h(key) = key \mod 10$, where the hash table $T[0:9]$ is initially empty. Note: Use Chained-Hash-Insert, i.e., insert at the head of the list. We are interested in the state of the hash table at the end after all the items have been hashed, that is, the order (starting from the head) of the items identified by keys.

To insert the items into a hash table with chaining using the given hash function $h(key) = key \mod 10$, we will perform the following steps:

1. Create an initially empty hash table $T[0:9]$, where each slot is an empty linked list.
2. For each item's key in the given order, calculate its hash value using the hash function and insert it at the head of the corresponding linked list in the hash table.

Here's the sequence of inserts:

1. Insert 79 at index 9: $T[9] = 79 \rightarrow$ NULL
2. Insert 35 at index 5: $T[5] = 35 \rightarrow$ NULL
3. Insert 30 at index 0: $T[0] = 30 \rightarrow$ NULL
4. Insert 42 at index 2: $T[2] = 42 \rightarrow$ NULL
5. Insert 31 at index 1: $T[1] = 31 \rightarrow$ NULL
6. Insert 39 at index 9: $T[9] = 39 \rightarrow 79 \rightarrow$ NULL
7. Insert 21 at index 1: $T[1] = 21 \rightarrow 31 \rightarrow$ NULL
8. Insert 54 at index 4: $T[4] = 54 \rightarrow$ NULL
9. Insert 14 at index 4: $T[4] = 14 \rightarrow 54 \rightarrow$ NULL
10. Insert 71 at index 1: $T[1] = 71 \rightarrow 21 \rightarrow 31 \rightarrow$ NULL
11. Insert 98 at index 8: $T[8] = 98 \rightarrow$ NULL
12. Insert 97 at index 7: $T[7] = 97 \rightarrow$ NULL
13. Insert 18 at index 8: $T[8] = 18 \rightarrow 98 \rightarrow$ NULL
14. Insert 41 at index 1: $T[1] = 41 \rightarrow 71 \rightarrow 21 \rightarrow 31 \rightarrow$ NULL
15. Insert 80 at index 0: $T[0] = 80 \rightarrow 30 \rightarrow$ NULL
16. Insert 29 at index 9: $T[9] = 29 \rightarrow 39 \rightarrow 79 \rightarrow$ NULL

The final state of the hash table is as follows:

$$
\begin{aligned}
T[0] &: 80 \rightarrow 30 \rightarrow \text{NULL}\\
T[1] &: 41 \rightarrow 71 \rightarrow 21 \rightarrow 31 \rightarrow \text{NULL}\\
T[2] &: 42 \rightarrow \text{NULL}\\
T[3] &: \text{NULL}\\
T[4] &: 14 \rightarrow 54 \rightarrow \text{NULL}\\
T[5] &: 35 \rightarrow \text{NULL}\\
T[6] &: \text{NULL}\\
T[7] &: 97 \rightarrow \text{NULL}\\
T[8] &: 18 \rightarrow 98 \rightarrow \text{NULL}\\
T[9] &: 29 \rightarrow 39 \rightarrow 79 \rightarrow \text{NULL}
\end{aligned}
$$

This is the result after inserting all the items into the hash table with chaining using the given hash function.

2 Solution

Given a schedule of musicians, each represented as an interval (s_i, e_i), where s_i is the start time and e_i is the end time, determine the minimum number of rooms needed to accommodate all musicians.

To solve the problem, we'll use a sorting-based approach. The idea is to sort the start times and end times and then traverse them to determine the number of rooms needed at any given time.

Algorithm 1 Minimum Rooms Needed

```
procedure MinRoomsNeeded(schedule[])
    Create empty arrays: start_times[], end_times[]
    for musician in schedule do
        Append musician.start to start_times[]
        Append musician.end to end_times[]
    end for
    Sort start_times[] in ascending order
    Sort end_times[] in ascending order
    Initialize $i$ and $j$ to 0
    Initialize rooms_needed to 0
    Initialize current_rooms to 0
    while $i <$ length(schedule) do
        if start_times[$i$] $\leq$ end_times[$j$] then
            Increment $i$
            Increment current_rooms
            Update rooms_needed if current_rooms is greater
        else
            Increment $j$
            Decrement current_rooms
        end if
    end while
    return rooms_needed
end procedure
```

The algorithm proceeds as follows:

1) It initializes two arrays, start_times and end_times, to store the start and end times of musicians.

2) The start and end times are extracted from the input schedule and appended to the respective arrays.

3) Both arrays are sorted in ascending order. Two pointers, i and j, are initialized to 0, along with the variables rooms_needed and current_rooms.

4)The algorithm iterates through the sorted arrays, comparing start times with end times to determine when rooms are needed or released.

5)current_rooms keeps track of the number of rooms in use at a given time, and rooms_needed is updated whenever a new maximum is encountered.

6)The algorithm returns rooms_needed as the minimum number of rooms required.

The algorithm is correct because it identifies the minimum number of rooms needed to accommodate all musicians without conflicts. It does so by tracking overlapping intervals and releasing rooms when musicians finish their performances.

The time complexity of the algorithm is $O(n \log n)$ due to the sorting step, where n is the number of musicians in the schedule.

The algorithm's time complexity of $O(n \log n)$ can be explained as follows:

Sorting Step
The most time-consuming part of this algorithm is the sorting of the 'start_times' and 'end_times' arrays.

1. Sorting 'start_times' takes $O(n \log n)$ time because it involves comparing and rearranging n elements.
2. Similarly, sorting 'end_times' also takes $O(n \log n)$ time for the same reason.
3. The total time for sorting is $2 \cdot O(n \log n)$, which simplifies to $O(n \log n)$.

Iteration
After sorting, the algorithm iterates through the arrays, comparing start times with end times. This iteration has a linear time complexity, $O(n)$, as it processes each musician's start and end times exactly once.
The sorting step dominates the time complexity, so the overall time complexity of the algorithm is $O(n \log n)$ due to the sorting of both 'start_times' and 'end_times'.

In summary, the time complexity of $O(n \log n)$ is achieved because of the sorting operations, while the iteration through the sorted arrays contributes a linear factor to the time complexity. This algorithm efficiently determines the minimum number of rooms needed for the given schedule of musicians.

3 Solution

To prove that in the worst case you have to buy $(n/2) + 1$ ice creams to be guaranteed to eat a chocolate one, consider the worst-case scenario: always choosing vanilla ice creams until you're left with the last one, which must be chocolate.

(a) Statement: In the worst case, you have to buy at least $(n/2)+1$ ice creams to be guaranteed to eat a chocolate one.

Proof by Contradiction:

1. Assumption: Suppose there exists a strategy where you can buy fewer than $(n/2) + 1$ ice creams and still be guaranteed to eat a chocolate one.

2. Define a Variable: Let k be the number of ice creams you buy in the strategy, where $k < (n/2) + 1$.

3. Equal Distribution: Since you have a total of n ice creams, half of them are chocolate and half are vanilla. Therefore, there are $n/2$ chocolate ice creams.

4. Worst Case Scenario: In the worst-case scenario, you are always picking vanilla ice creams. This means you can choose vanilla ice creams $n/2$ times before you are left with the final ice cream.

5. Contradiction: The assumption that you can buy fewer than $(n/2)+1$ ice creams and still be guaranteed to eat a chocolate one contradicts the worst-case scenario. If you buy fewer than $(n/2) + 1$ ice creams, you should be able to choose vanilla ice creams $n/2$ times and still have more ice creams left. But, you can't do this because you only have a total of n ice creams.

6. Conclusion: Since the assumption leads to a contradiction, the original statement is proven to be true. In the worst case, you have to buy at least $(n/2) + 1$ ice creams to be guaranteed to eat a chocolate one.

(b) Randomized Algorithm
To find a chocolate ice cream with probability at least $1 - (1/n)$, i am using the following randomized algorithm:

Algorithm 2 Randomized Ice Cream Selection

```
$k \leftarrow \lceil \log_2(n) \rceil$                      ▷ Number of ice creams to purchase
for $i = 1$ to $k$ do
    Purchase a random ice cream without opening it
    if It's chocolate then
        return Chocolate ice cream found
    end if
end for
Consume the $k$-th ice cream                    ▷ It doesn't matter if it's vanilla or chocolate
return Chocolate ice cream not found
```

The algorithm purchases a random ice cream k times, checking each time if it's chocolate. If chocolate is found during this process, the algorithm terminates. If not, it consumes the last ice cream, whether it's vanilla or chocolate, as you cannot return it. The probability of not finding a chocolate ice cream in k tries is $(1/2)^k$. To guarantee a probability of at least $1 - (1/n)$, k is set to $\lceil \log_2(n) \rceil$.

The probability of getting a chocolate ice cream on the first try is 1/2. If you don't get a chocolate ice cream on the first try, you continue with a probability of 1/2 for each subsequent ice cream.

To guarantee a probability of at least $1 - (1/n)$, calculate the probability of failing to get a chocolate ice cream in the first k tries. The probability of not getting a chocolate ice cream in k tries is $(1/2)^k$. You want this probability to be less than $1/n$:

$$(1/2)^k <= 1/n$$

(here we are taking "=" for atleast condition)
Solving for k:

$$k >= \log_2(n)$$

So, the smallest number of ice creams you have to buy to guarantee a probability of at least $1 - (1/n)$ of getting a chocolate ice cream is $k = \lceil \log_2(n) \rceil$.

4 Solution

Explanation The algorithm 'FindMostPopularAnimal' finds the most popular animal in an array of animals $A[1 : n]$ by dividing the problem into smaller subproblems and merging the results. Here's how it works:
1. If the array has only one animal, return that animal as the most popular (base case).
2. Divide the array into two equal halves.
3. Recursively find the most popular animal in the left half ($leftMostPopular`)andtherighthalf$(rightMostPopular').
4. Compare the most popular animals in the left and right halves. If they are the same, return one of them as the most popular animal.
5. If they are different, count the occurrences of both animals in the entire array using the 'CountAnimals' function.
6. Return the animal with the highest count as the most popular overall.

The algorithm is designed to ensure that the most popular animal in the original array is correctly identified. The time complexity is O(n log n) due to the divide-and-conquer approach and the linear time required for counting in the merge step.

Algorithm 3 FindMostPopularAnimal

```
Require: Array of animals $A[1:n]$
Ensure: The most popular animal
  function FINDMOSTPOPULARANIMAL($A[1:n]$)
      if $n == 1$ then
          return $A[1]$                                              ▷ Base case: Only one animal
      end if
      $mid \leftarrow \lfloor n/2 \rfloor$                           ▷ Divide the array into two equal halves
      $leftMostPopular \leftarrow$ FINDMOSTPOPULARANIMAL($A[1:mid]$)          ▷ Find in left half
      $rightMostPopular \leftarrow$ FINDMOSTPOPULARANIMAL($A[mid+1:n]$)       ▷ Find in right half
      if $leftMostPopular == rightMostPopular$ then
          return $leftMostPopular$               ▷ Most popular animal is the same in both halves
      else
          $leftCount \leftarrow$ COUNTANIMALS($A, leftMostPopular$)            ▷ Count left half
          $rightCount \leftarrow$ COUNTANIMALS($A, rightMostPopular$)          ▷ Count right half
          return max($leftMostPopular, rightMostPopular$, with highest count)     ▷ Return most popular
  overall
      end if
  end function
```

To analyze the time complexity of the `FindMostPopularAnimal` algorithm, we use the Master Theorem. The algorithm can be described as follows:

- For an input array of size n, it performs some constant-time work (e.g., comparisons and assignments).
- It recursively calls itself on two subarrays, each of size $n/2$.
- It performs additional constant-time work to compare and count animals based on the results from the subproblems.

The recurrence relation for the time complexity $T(n)$ can be expressed as:

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n)$$

This is because the algorithm divides the problem into two equal subproblems of size $n/2$, and the work required in merging and comparing the results from the subproblems is $O(n)$.

Now, we use the Master Theorem to determine the time complexity:

Master Theorem (General Form):

$$T(n) = a \cdot T\left(\frac{n}{b}\right) + f(n)$$

In our case:

$$\begin{aligned} a &= 2 && \text{(the number of subproblems)} \\ b &= 2 && \text{(the factor by which the problem size is reduced)} \\ f(n) &= O(n) && \text{(the work done outside of recursive calls)} \end{aligned}$$

Let's calculate $\log_b(a)$:

$$\log_2(2) = 1$$

Now, we compare $\log_b(a)$ to the exponent in $f(n)$:
- If $\log_b(a) < 1$ (in this case, it is 1), then $T(n) = O(n^1)$, which means the work outside of the recursive calls dominates the time complexity.

In this case, it's $O(n)$. - If $\log_b(a) = 1$ (in this case, it is 1), then $T(n) = O(n \log n)$.

Therefore, by the Master Theorem, the time complexity of the `FindMostPopularAnimal` algorithm is $O(n \log n)$.

5 Solution

Problem Statement

Let's consider a hash table $T[0 : m-1]$. In order to resolve collisions, we assign each key k a sequence of hash values $h(k,0), h(k,1), \dots, h(k,m-1)$ rather than a single hash value $h(k)$. The hash function is defined as $h : U \times \{0,\dots,m-1\} \to \{0,\dots,m-1\}$, and we insert the item at the position $h(k,i)$ minimizing i that is empty.

We say that a hash function h for open addressing is unimaginably good if it behaves completely randomly. A uniformly random key k is mapped to a sequence of hash values (probes): $(h(k,0), h(k,1), \dots, h(k,m-1))$ which is equal to any permutation of the numbers $0,\dots,m-1$ with equal probability, and the permutations obtained for two different keys are independent random outcomes.

Suppose we are using an open-addressed hash table of size m to store n random items, where $n \le \frac{m}{2}$, with uniformly distributed keys. Assume an unimaginably good random hash function. For any i, let X_i denote the random variable giving us the number of probes required for the ith insertion into the table, and let $X = \max_i X_i$ denote the length of the longest probe sequence.

Solutions

(a)

To prove that $P(X_i > k) \le \frac{1}{2^k}$, we can use the fact that the hash function is unimaginably good, meaning that for each insertion, the probe sequence is a uniformly random permutation of the numbers 0 to $m-1$, and these permutations for different keys are independent.

For a single insertion, the probability that $X_i > k$ is the probability that the item being inserted experiences more than k probes. Since the probes are determined by a random permutation, the probability of any specific sequence of probes is $\frac{1}{\binom{m}{k}}$, where $\binom{m}{k}$ represents the number of ways to choose k distinct values from a set of m. This is because there are $\frac{m!}{(m-k)!k!}$ possible permutations of the first k numbers, and there are $m!$ total permutations.

Now, we want to find the probability that $X_i > k$, which is the sum of the probabilities of all permutations that require more than k probes:

$$P(X_i > k) = \sum \frac{1}{\binom{m}{k}} \text{ for all values where } X_i > k.$$

Since there are $\frac{m!}{(m-k)!k!}$ possible permutations, and we are summing over permutations requiring more than k probes, we get:

$$P(X_i > k) = \sum \frac{(m-k)!\frac{m!}{k!}}{m!} = \sum \frac{1}{\binom{m}{(m-k)}} = \sum \frac{1}{\binom{m}{k}} \text{ for all values where } X_i > k.$$

Now, using the fact that $\binom{m}{k}$ is at most m^k, we have:

$$P(X_i > k) = \sum \frac{1}{\binom{m}{k}} \text{ for all values where } X_i > k \le \sum \frac{1}{m^k} = \frac{1}{m^k} \sum 1 = \frac{1}{m^k}.$$

Since this holds for any specific insertion, we can conclude that $P(X_i > k) \le \frac{1}{2^k}$ for all i and k.

(b)

To prove that $P(X_i > 2\log_2(n)) \le \frac{1}{n^2}$, we'll use a similar approach. We want to find the probability that the ith insertion requires more than $2\log_2(n)$ probes.

Using the same reasoning as in part (a), we have:

$$P(X_i > 2\log_2(n)) = \sum \frac{1}{\binom{m}{2\log_2(n)}} \text{ for all values where } X_i > 2\log_2(n).$$

Since $\binom{m}{k}$ is at most m^k, we can write:

$$P(X_i > 2\log_2(n)) \le \sum \frac{1}{m^{2\log_2(n)}} = \sum \frac{1}{n^2} = \frac{1}{n^2}.$$

This shows that $P(X_i > 2\log_2(n)) \le \frac{1}{n^2}$ for all i.

(c)

To prove that $P(X > 2\log_2(n)) \leq \frac{1}{n}$, we can use a union bound. We have n insertions, and we want to find the probability that the length of the longest probe sequence X is greater than $2\log_2(n)$.

Using the union bound:

$$P(X > 2\log_2(n)) = P(X_1 > 2\log_2(n) \text{ or } X_2 > 2\log_2(n) \text{ or } \ldots \text{ or } X_n > 2\log_2(n)) \leq P(X_1 > 2\log_2(n)) + P(X_2 > 2\log_2(n))$$

From part (b), we know that $P(X_i > 2\log_2(n)) \leq \frac{1}{n^2}$ for all i. Therefore, by summing these probabilities for all i, we get:

$$P(X > 2\log_2(n)) \leq \frac{1}{n^2} + \frac{1}{n^2} + \ldots + \frac{1}{n^2}(n \text{ times}) = n\left(\frac{1}{n^2}\right) = \frac{1}{n}.$$

So, we've shown that $P(X > 2\log_2(n)) \leq \frac{1}{n}$.

(d)

To prove that $E(X) = O(\log_2(n))$, we can use the hint provided.

The hint states that for our particular random variable X, it holds that:

$$E(X) \leq P(X \geq k)n + k, \text{ for } 0 < k < n.$$

Let's use this inequality with $k = 2\log_2(n)$:

$$E(X) \leq P(X \geq 2\log_2(n))n + 2\log_2(n).$$

From part (c), we know that $P(X \geq 2\log_2(n)) \leq \frac{1}{n}$, so we can substitute this in:

$$E(X) \leq \left(\frac{1}{n}\right)n + 2\log_2(n) = 1 + 2\log_2(n).$$

Since we're interested in the upper bound, we can ignore the constant 1. Therefore, we have:

$$E(X) = O(\log_2(n)),$$

as the linear term $(2\log_2(n))$ dominates the constant.