Solution of Third Homework

Han Wu, Yu Chen

1 Definition of BST (10 points)

(10 pts) Let T be a binary search tree whose keys are distinct, let x be a leaf node, and let y be its parent. Show that y.key is either the smallest key in T larger than x.key or the largest key in T smaller than x.key.

[We are expecting: a detailed proof, using the definition of a binary search tree.]
There are two cases:

1) x is the left child of y. In this case, y is larger than x. So we will prove that y is the smallest key in T larger than x. Suppose that there is another node z who is larger than x and is smaller than y. Because z is larger than x, it must be in x's right subtree, or x must be in z's left subtree. However, x is a leaf node, so z cannot be in x's right subtree, it can only be an ancestor of x, and x is in its left subtree. Because z is smaller than y, it must be in y's left subtree, or y is in z's right subtree. However, y has only one left child, namely x, so z can only be an ancestor of y, and y is in its right subtree, contradicting to the condition that x is in z's left subtree(x is y's left child, and y is in z's right subtree, so x must be in z's right subtree as well). Thus, there is no such z exists. So we can conclude that y is the smallest key in T larger than x.
2) x is the right child of y. The proof is similar than the above one, so we omit it here.

2 Insertion of BST (10 points)

(10 pts) We can sort a given set of n numbers by first building a binary search tree containing these numbers (using Insert(key) in page 18 of lecture 7 repeatedly to insert the numbers one by one) and then printing the numbers by an inorder tree walk. What are the worst-case and best-case running times for this sorting algorithm?

[We are expecting: a clear description about the running times of the worst-case and best-case along with one small paragraph for each case. You may make examples to explain your answers.]
Because the in order traversal of a tree only need $O(n)$, the time complexity of above algorithm is dominated by the building time of the tree.

- Worst-case: if we use a sorted list to build the tree, the result tree will be like a linked list with no left/right subtrees. The running time is $O(n^2)$.
- Best-case: if we use a list which can build a relatively balanced binary search tree, then the overall running time could be $O(nlogn)$.

3 Number of Collisions (10 points)

(10 pts) Suppose we use a hash function h to hash n distinct keys into an array T of length m. Assuming simple uniform hashing, what is the expected number of collisions? More precisely, what is the expected cardinality of $\{\{k, l\}: k \neq l \text{ and } h(k) = h(l)\}$? **[We are expecting: a detailed proof.]**
$\frac{n^2-n}{2m}$. Details can refer to CLRS Solution 11.2-1.

4 Hash Tables (5 points)

In this problem, we will investigate a way to improve the worst-case performance of the chained hash tables that we saw in class. Suppose that $\mathcal{U}$ is a universe of size M, let n be an integer and that $\mathcal{H}$ is a universal family of hash functions that map $\mathcal{U}$ into buckets labeled $1, \ldots, n$. Suppose that $M \geq n^2$.

1. (2 pts) What is the worst-case running time for `search` in a chained hash table? That is, suppose that after h is chosen, an adversary chooses $u_1, \ldots, u_n \in \mathcal{U}$ to insert into the table, and then runs a `search` query on an item of their choosing; how long will this last `search` query take in the worst case?

   [We are expecting: One or two sentences with your answer, and a description of how this might happen.]
   The worst-case running time for `search` in a chained hash table is $\Omega(n)$. In the worst-case, $h(u_1) = h(u_2) = \cdots = h(u_n)$. Then these n elements are all divided into the same bucket. Therefore, we need to search and compare these n elements in the worst-case.

2. (3 pts) Find a way to modify of the chained hash table that we saw in class so that, if at most n items $u_1, \ldots, u_n \in \mathcal{U}$ are ever inserted into your modified data structure:

   - The expected time[1] to perform `search,insert,delete` is still $O(1)$.
   - The worst-case running time (in the sense of part (a)) to perform `search, insert, delete` operations are $O(\log(n))$.

   Your modified data structure should still involve choosing a hash function $h \in \mathcal{H}$ uniformly at random, and should still use only $O(n \log(M))$ space.

   [We are expecting: a description of the data structure and a description of how to perform `search`, `insert`, and `delete` in this data structure. We are also expecting an informal argument (a paragraph or two) about the expected and worse case running time. You do not need to write a formal proof, and you may use results and arguments from class or the textbook.]
   1. Convert the linked list structure in the bucket of hash table into balanced binary tree (e.g. Red-Black Tree). In the updated data structure, for the `search`, `insert`, `delete` operation, we first hash the elements by hash function h. Then we search the target element in the balanced binary tree of the corresponding bucket.
   2. Due to the randomness of hash function, we can complete L operations in $O(L)$ times. So that the expected time to perform above operations is $O(1)$.
   3. In the worst-case, we need to search in a n elements bucket (Maintenance with balanced binary tree). So we need $O(\log n)$ time to perform operations.
   4. In addition, the storage cost is the same as the linked list structure. So it should still use only $O(n \log M)$ space.

5 Bloom filter (5 points)

Suppose that you are making a lightweight web browser and you have a large, fixed blacklist of w malicious websites. Let M denote the number of websites on the whole internet (this is a huge number!) and suppose for this problem that this number does not change.

Your goal is to use randomness to design a data structure that can be shipped with the browser, that is as small as possible. The data structure can be queried with a function called `isMalicious`, and has the following three properties:

- (**Small space.**) The data structure uses at most b bits, where b is an integer that is a design parameter you'd like to minimize.

[1] This is expected time in the formal sense discussed in class: for any fixed set of items $u_1, \ldots, u_n \in \mathcal{U}$, and for any sequence of L `search,insert,delete` operations only involving these elements, the expected amount of time, over the randomness used to choose the data structure, to perform all L operations is $O(L)$.

- (**No false negatives.**) If a website x is on the blacklist, then `isMalicious`(x) always returns `True`.
- (**Unlikely false positives.**) Fix a website x that is *not* on the blacklist. Then with probability at least 0.99 over the randomness that you used when choosing the data structure, `isMalicious`(x) returns `False`.

Above, we emphasize that the probability works as follows: the blacklist is first fixed and *will never change.* Fix some website x, either malicious or not, and keep it in mind. Now we use randomness to generate the data structure that will ship with the browser. If x was malicious, then with probability 1, `isMalicious(x) = True`. If x was not malicious, then with probability at least 0.99, `isMalicious(x) = False`. The only randomness in the problem has to do with the choice of the data structure.

1. (1pt) Suppose that we just ship the blacklist directly with the browser (so there is no randomness, and we always correctly classify both malicious and legitimate websites). How many bits does this require? You may assume that a single website (out of all M of them) can be represented using $\log(M)$ bits. **[We are expecting: a single sentence.]** $b = w(\log(M))$ bits

For parts (b) and (c): Let n be an integer. Let $\mathcal{H}$ be a collection of functions $h : \{x \ : \ x \text{ is a website } \} \to \{1, \ldots, n\}$ that map websites to one of n buckets, and suppose that $\mathcal{H}$ is a universal hash family. Notice that the size of the domain of h is M. Suppose that, as with the universal hash family we saw in class, the description of an element $h \in \mathcal{H}$ requires $d = O(\log(M))$ bits.

(b) (2 pts) Below, we describe one way to randomly generate a data structure, using the universal hash family $\mathcal{H}$. The pseudocode to construct the data structure is:

```
Choose h in H at random.
Initialize an array T that stores n bits.
Initially, all entries of T are 0.
for x in the black list:
      T[h(x)] = 1.
end for
```

Then we ship T and a description of h with the browser. In order to query the data set, we define

```
function isMalicious(x):
      if T[h(x)] = 1:
            return True
      else:
            return False
      end if
end function
```

Suppose that $n = 100w$. Show that the three requirements above are met, with $b = d + 100w$.

[We are expecting: a proof that all three properties hold: small space, no false negatives, unlikely false positives.]

Proof.

(1) **Small space:** We use $d = O(\log(M))$ bits for storing hash functions $h \in \mathcal{H}$, and $100w$ bits for buckets. Therefore, the total memory used is $b = d + 100w$.

(2) **No false negatives:** In the pseudo code, we set the bucket bits of each website in the blacklist to 1 ($T(h(x)) = 1$). Therefore, when the user calls the `isMalicious(x)`, it will return `true` if the query website x is a malicious website.

(3) **Unlikely false positives:** By the definition of Universal Hash Function Family, we have

$$\forall u_i, u_j \in M \text{ and } u_i \neq u_j, P[h(x_i) = h(x_j) : h \leftarrow \mathcal{H}] \leq \frac{1}{n}.$$

Therefore, the probability that the hash value of a malicious website is the same as that of a normal website is $P_{fp} \leq w * \frac{1}{100w} = \frac{1}{100}$. So that `isMalicious(x)` returns `False` with probability at least 0.99.

□

(c) (2 pts) Now consider the following variant on the data structure from the previous part. Let $\mathcal{H}$, d and n be as above. Instead of one array T, we will initialize 10 arrays $T_1, \ldots, T_{10}$, as follows:

```
Choose h_1,...,h_10 independently and uniformly random in H.
Initialize 10 arrays T_1,...,T_10 that store n bits each.
Initially, for all i=1,...,10, all entries of T_i are 0.
for x in the black list:
      for i = 1,...,10:
            T_i[h_i(x)] = 1.
end for
```

We ship $T_1, \ldots, T_{10}$, and a description of $h_1, \ldots, h_{10}$, with the browser. Now, in order to ask if x is on the black list, we define:

```
function isMalicious(x):
      if T_i[h_i(x)] = 1 for all i = 1,...,10:
            return True
      else:
            return False
      end if
end function
```

Suppose that $n = 2w$. Show that all three requirements are met, with $b = 10d + 20w$.

Notice that when w is much bigger than d, this is better than part (b)!

[We are expecting: A proof that all three properties hold. You may reference your proof in part (b) for part (c).]

Proof.

(1) **Small space:** We use $10d$ bits for storing hash functions h_i, and $10 * 2w$ bits for 10 buckets. Therefore, the total memory used is $b = 10d + 20w$.

(2) **No false negatives:** In the pseudo code, we set the bucket bits of each website in the black-list of each hash function to 1 ($\forall i \in [1..10], T(h_i(x)) = 1$). Therefore, when the user calls the `isMalicious(x)`, it will return `true` if the query website x is a malicious website.

(3) **Unlikely false positives:** By the definition of Universal Hash Function Family, we have

$$\forall u_i, u_j \in M \text{ and } u_i \neq u_j, P[h(x_i) = h(x_j) : h \leftarrow \mathcal{H}] \leq \frac{1}{n}.$$

Therefore, the probability that the hash value of a malicious website is the same as that of a normal website is $P_{fp} \leq (w * \frac{1}{2w})^1 0 = \frac{1}{1024}$. So that `isMalicious(x)` returns `False` with probability at least $0.999 > 0.99$.

□