1 Question 1 (1+1+1 pts)

Write pseudocodes. Use open-addressing.

- **HASH-SEARCH(T,k):** Return value if exists, NIL otherwise.
- **HASH-INSERT(T,k,v):** If k already exists, update its value. If it does not exist, then insert it unless table is full (return NIL if full).
- **HASH-DELETE(T,k).**

### Algorithm 1: HASH-SEARCH(T,k)

1. \(i \leftarrow 0\)
2. #Assuming m shows table size
3. repeat
4.   #Hash function
5.   \(j \leftarrow h(k, i)\)
6.   #Assuming each node is a tuple of 2
7.   #and index 0 is key, 1 is value
8.   if \(T[j][0] = k\) then return \(T[j][1]\);
9.   \(i \leftarrow i + 1\)
10. until \(T[j] = NIL\) or \(i = m\);
11. return NIL

### Algorithm 2: HASH-INSERT(T,k,v)

1. #First, make a search over all the table to see if we have the key
2. \(i \leftarrow 0\)
3. repeat
4.   \(j \leftarrow h(k, i)\)
5.   if \(T[j][0] = k\) then
6.     #Found the key, update the value and exit
7.     \(T[j][1] \leftarrow v\)
8.     return \(v\)
9.     end
10. \(i \leftarrow i + 1\)
11. until \(T[j] = NIL\) or \(i = m\);
12. #If we reach here, it means we couldn’t find the key.
13. #Now we should make another search to find an empty slot.
14. \(i \leftarrow 0\)
15. repeat
16.   \(j \leftarrow h(k, i)\)
17.   if \(T[j] = NIL\) or \(T[j][0] = DELETED\) then
18.     #Found empty slot, insert and exit
19.     \(T[j] \leftarrow (k, v)\)
20.     return \(v\)
21.   end
22. \(i \leftarrow i + 1\)
23. until \(i = m\);
24. #If we reach here, it means table is full (no empty slots).
25. return NIL
Observe \texttt{HASH-SEARCH} returns the value if the key exists in the table, so it won’t be very useful to call it from \texttt{HASH-INSERT} or \texttt{HASH-DELETE}. Also observe when \texttt{HASH-SEARCH} sees DELETED, it treats it as yet another occupied slot, it increments $i$ and keeps searching.

\begin{algorithm}
\caption{HASH-DELETE(T,k)}
\begin{algorithmic}[1]
\State $i \leftarrow 0$
\Repeat
\State $j \leftarrow h(k, i)$
\If {$T[j][0] == k$}
\Comment{Lazy-deletion}
\State $T[j] = \text{(DELETED, DELETED)}$
\State \text{return DELETED}
\EndIf
\State $i \leftarrow i + 1$
\Until {$T[j] == \text{NIL}$ or $i == m$;}
\Comment{If we reach here it means we couldn’t find key in table}
\State \text{return NIL}
\end{algorithmic}
\end{algorithm}
2 Question 2 (1+1 pt)

Everything else is the same, prove/disprove two things (compare two strategies C (chaining) and O (open-addressing)):

- Number of primary collisions: \( C \leq O \).
- Number of slots searched for the case of successful search: \( C \leq O \).

If a collision happens in Chaining, then you don’t use a new slot, but you do in Open-Addressing. So Open-Addressing fills up more quickly (in case where collisions happen), increasing the chances of collision. Hence the first statement \((C \leq O)\) is correct. (As an edge case, observe you fill the table with no collision, and \(0 = 0\).)

Second part is false and can be disproved with a counterexample for a successful search where Open Addressing > Chaining. (For example, assume you are inserting x and y and both have the same hash function. It will take only 1 place to look in chaining (since chaining prepends), but 2 in open addressing).
3 Question 3 (1+1 pt)

- Write pseudocode for a function that takes a BST $T$ (or the root note $x$), and two keys $a < b$, print all such $c$s in $T$ satisfying $a \leq c \leq b$ in sorted order. Find time-complexity.

- Can you do it efficiently (at least as efficient as part 1) with Hash-Table? If yes, give pseudocode. If no, explain.

<table>
<thead>
<tr>
<th>Algorithm 4: PRINT-BETWEEN(x,a,b)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 if $x \neq \text{NIL}$ then</td>
</tr>
<tr>
<td>2 \hspace{1em} PRINT-BETWEEN(x.left,a,b)</td>
</tr>
<tr>
<td>3 \hspace{1em} if $a \leq x.key \leq b$ then print(x.key);</td>
</tr>
<tr>
<td>4 \hspace{1em} PRINT-BETWEEN(x.right,a,b)</td>
</tr>
<tr>
<td>end</td>
</tr>
</tbody>
</table>

This is a modified version of in order tree walk. It has time complexity $O(n)$ because the if check on line 3 is performed on every single node the tree has. In-order traversal ensures printed keys are sorted.

However, we are losing the notion of ordering in hash tables. So the only way to do this is to get all keys and do a sort afterwards, and print all keys which lie in $[a, b]$. Since the procedure would involve a sorting, it would require $O(n \log n)$ time, which is not as efficient as a BST.