CPE - 593: Homework 7 - SEQ - Huffman Trees and Hashing

Jude M. Eschete

March 18, 2025

1. Tree Applications

1.1 Pseudocode for Building a Huffman Tree (3 points)

Goal: Construct a Huffman tree for a given string based on the frequency of each character.

Pseudocode:

1. **Compute frequencies:** For each unique character in the input, count its frequency. Store this in a map or dictionary $\{\,\text{char} : \text{freq}\}$.
2. **Initialize a min-priority queue (min-heap):**
   - For each character c with frequency f_c, create a leaf node.
   - Insert each node into the min-heap, with priority = frequency.
3. **Build the tree:** While there is more than one node in the min-heap:
   (a) Extract the two nodes with the smallest frequencies (call them n_1 and n_2).
   (b) Create a new internal node n_{new} with:
   $$\text{freq}(n_{\text{new}}) = \text{freq}(n_1) + \text{freq}(n_2)$$
   (c) Set n_1 as the left child and n_2 as the right child of n_{new}.
   (d) Insert n_{new} back into the min-heap.
4. **Final node:** When only one node remains in the min-heap, that node is the root of the Huffman tree.

1.2 Huffman Codes for AAAABBCDDEEEEEE (3 points)

We first compute the frequency of each character. For the string AAAABBCDDEEEEEE:

$$A : 4,$$
$$B : 2,$$
$$C : 1,$$
$$D : 2,$$
$$E : 6.$$

We then build the Huffman tree by combining the lowest-frequency nodes first. A possible final Huffman tree (one of equally valid variations) assigns the codes:

$$A \to 10,$$
$$B \to 1111,$$
$$C \to 1110,$$
$$D \to 110,$$
$$E \to 0.$$

1.3 Encoding of the Original String (2 points)

Using the codes above, we convert each character of AAAABBCDDEEEEEE into its Huffman bit string:

$$\texttt{A} = 10, \quad \texttt{B} = 1111, \quad \texttt{C} = 1110, \quad \texttt{D} = 110, \quad \texttt{E} = 0.$$

The original string is

$$\underbrace{\texttt{AAAA}}_{4\times A}\ \underbrace{\texttt{BB}}_{2\times B}\ \texttt{C}\ \underbrace{\texttt{DD}}_{2\times D}\ \underbrace{\texttt{EEEEEE}}_{6\times E}.$$

Substituting codes in order:

$$\begin{aligned}
\texttt{AAAA} &\to 10\,10\,10\,10 = 10101010,\\
\texttt{BB} &\to 1111\,1111 = 11111111,\\
\texttt{C} &\to 1110,\\
\texttt{DD} &\to 110\,110 = 110110,\\
\texttt{EEEEEE} &\to 0\,0\,0\,0\,0\,0 = 000000.
\end{aligned}$$

Concatenating these segments, the final encoded bit string is:

$$\boxed{10101010111111111110110110000000}.$$

2. True or False

2.1 (1 point)

"A hash function may give the same hash value for distinct input data."

This statement is $\boxed{\text{True}}$, since different keys can naturally collide to the same hash value.

2.2 (1 point)

"When the load factor exceeds a threshold, rehashing is performed. A key stored in the original hash table is copied to the bin of the same index in the new hash table."

This statement is $\boxed{\text{False}}$. Upon rehashing (especially if the table size changes), each key's hash value is recomputed. Typically, items do not remain in the same index after resizing; they are redistributed based on the new table size.

3. Hashing Algorithms (8 points)

Below, we use the hash function $\text{hash}(x) = x \bmod 16$ and insert the keys

$$\{36,\ 52,\ 30,\ 22,\ 38,\ 54\}.$$

We illustrate four collision-resolution strategies: *(a) linear probing*, *(b) quadratic probing*, *(c) double hashing*, and *(d) separate chaining*. In each case, we track the final position (or chain) of each key.

3.1 m=16, Linear Probing

Formula Setup:

$$\text{hash}(x) = x \bmod 16, \quad \text{Probe sequence: } h(x) + i \quad (i = 0, 1, 2, \dots).$$

Insertion order: 36, 52, 30, 22, 38, 54.

1. $\text{hash}(36) = 4$. Slot 4 is empty, place 36 at index 4.
2. $\text{hash}(52) = 4$. Collision at index 4; probe index 5 (empty), place 52 at index 5.
3. $\text{hash}(30) = 14$. Slot 14 is empty, place 30 at index 14.
4. $\text{hash}(22) = 6$. Slot 6 is empty, place 22 at index 6.
5. $\text{hash}(38) = 6$. Collision at index 6; probe index 7 (empty), place 38 at index 7.
6. $\text{hash}(54) = 6$. Collision at index 6, then 7; probe index 8 (empty), place 54 at index 8.

Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Value	–	–	–	–	36	52	22	38	54	–	–	–	–	–	30	–

3.2 m=16, Quadratic "Chaining" (Quadratic Probing)

Formula Setup:

$$\text{hash}(x) = x \bmod 16, \quad \text{Probe sequence: } h(x) + i^2 \quad (i = 0, 1, 2, \dots).$$

Insertion order: 36, 52, 30, 22, 38, 54.

1. $36 \bmod 16 = 4$. Place 36 at index 4.
2. $52 \bmod 16 = 4$. Collision at 4; try $4 + 1^2 = 5$ (empty), place 52 at 5.
3. $30 \bmod 16 = 14$. Place 30 at index 14.
4. $22 \bmod 16 = 6$. Place 22 at index 6.
5. $38 \bmod 16 = 6$. Collision at 6; try $6 + 1^2 = 7$ (empty), place 38 at 7.
6. $54 \bmod 16 = 6$. Collision at 6 and 7; try $6 + 1^2 = 7$ (taken), then $6 + 2^2 = 10$ (empty), place 54 at 10.

Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Value	–	–	–	–	36	52	22	38	–	–	54	–	–	–	30	–

3.3 m=16, Double Hashing

Formula Setup:

$$\text{hash}_1(x) = x \bmod 16, \quad \text{hash}_2(x) = 7 - (x \bmod 7),$$
$$\text{Probe sequence: } h_1(x) + i \cdot h_2(x) \quad (i = 0, 1, 2, \dots).$$

Insertion order: 36, 52, 30, 22, 38, 54.

1. 36: $\text{hash}_1(36) = 4$. Place 36 at index 4.
2. 52: $\text{hash}_1(52) = 4$ (collision), $\text{hash}_2(52) = 7 - (52 \bmod 7) = 7 - 3 = 4$. Next index $= (4 + 1 \times 4) \bmod 16 = 8$. Slot 8 empty, place 52.
3. 30: $\text{hash}_1(30) = 14$. Place 30 at index 14.
4. 22: $\text{hash}_1(22) = 6$. Slot 6 empty, place 22.
5. 38: $\text{hash}_1(38) = 6$ (collision), $\text{hash}_2(38) = 7 - (38 \bmod 7) = 7 - 3 = 4$. Next index $= (6 + 1 \times 4) \bmod 16 = 10$. Slot 10 empty, place 38.
6. 54: $\text{hash}_1(54) = 6$ (collision), $\text{hash}_2(54) = 7 - (54 \bmod 7) = 7 - 5 = 2$. We check:

$$(6 + 1 \times 2) \bmod 16 = 8 \quad \text{(occupied by 52)},$$
$$(6 + 2 \times 2) \bmod 16 = 10 \quad \text{(occupied by 38)},$$
$$(6 + 3 \times 2) \bmod 16 = 12 \quad \text{(empty)}.$$

Place 54 at index 12.

Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Value	–	–	–	–	36	–	22	–	52	–	38	–	54	–	30	–

3.4 m=16, Separate Linear Chaining

Formula Setup:

$\text{hash}(x) = x \bmod 16$; Collisions in the same bin using a linear (linked) chain.

Insertion order: 36, 52, 30, 22, 38, 54.

1. hash(36) = 4. Bucket 4: [36].
2. hash(52) = 4. Bucket 4: [36, 52].
3. hash(30) = 14. Bucket 14: [30].
4. hash(22) = 6. Bucket 6: [22].
5. hash(38) = 6. Bucket 6: [22, 38].
6. hash(54) = 6. Bucket 6: [22, 38, 54].

Index	Chain
0	[]
1	[]
2	[]
3	[]
4	[36, 52]
5	[]
6	[22, 38, 54]
7	[]
8	[]
9	[]
10	[]
11	[]
12	[]
13	[]
14	[30]
15	[]

4. Self-Study on Java 17 `Hashtable` Implementation (2 points)

In Java 17, the `Hashtable` class is a legacy collection introduced in the earliest versions of Java and remains for historical and backward-compatibility reasons. Unlike the more commonly used `HashMap`, `Hashtable` is synchronized by default, making it thread-safe in simple concurrent scenarios. This synchronization may introduce performance overhead, which is why many modern applications favor using `HashMap` (often in combination with a `Collections.synchronizedMap` wrapper or a `ConcurrentHashMap` for robust concurrency).

Internally, `Hashtable` uses **separate chaining** to resolve collisions. Specifically, it maintains an array of buckets, each of which stores a singly linked list (a chain) of `Entry` objects. An `Entry` contains the key, the associated value, the key's hash code, and a pointer to the next `Entry` in the chain. When new elements are inserted, hash(key) mod capacity determines the bucket index. If the bucket is already occupied, the element is linked after existing nodes. This design ensures that multiple keys mapping to the same index do not overwrite each other but simply join the existing chain.

If the number of stored entries grows large relative to the table size, the load factor triggers a rehash. During rehashing, `Hashtable` doubles its capacity (unless specified otherwise) and recomputes the bucket for each key. This process distributes the keys more evenly across the enlarged table, preventing excessively long chains from forming. Despite this automatic resizing, high contention or extremely large data sets can still degrade performance due to the linear traversal of chains.

Overall, Java 17's `Hashtable` adheres to a traditional collision-handling approach by linking entries in a chain. While it may suffice for smaller-scale, concurrent usage, developers often choose `HashMap` for non-synchronized contexts. For heavily multi-threaded environments, `ConcurrentHashMap` is often a more efficient choice, since it avoids locking the entire map during updates. Nevertheless, the under-the-hood mechanism for dealing with collisions—separate chaining—has remained consistent within `Hashtable` since its inception.