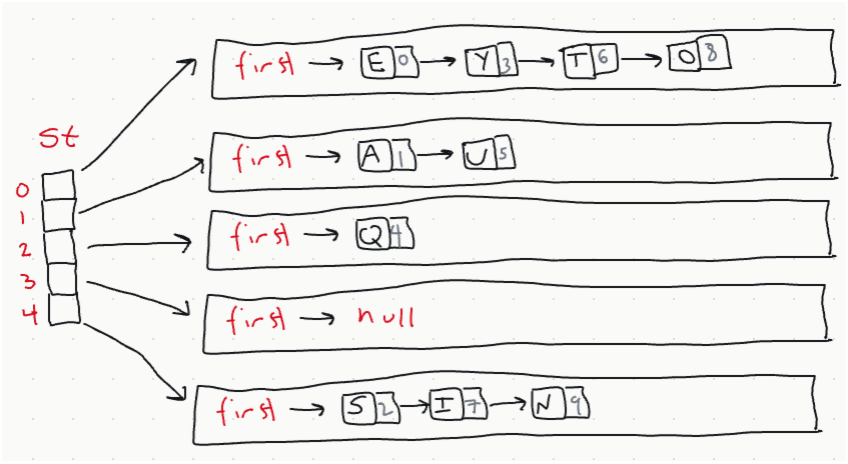

Figure 1: Separate Chaining Hash Table from keys `E A S Y Q U T I O N`.

Sergio E. Garcia Tapia
Algorithms by Sedgewick and Wayne (4th edition) [SW11]
January 03, 2025

3.4: Hash Tables

Exercise 1. Insert the keys `E A S Y Q U T I O N` in that order into an initially empty table of $m = 5$ lists, using separate chaining. Use the hash function `11 * k % m` to transform the kth letter of the alphabet into a table index.

Solution.

- E is the 5th letter, so $11 \cdot 5 \% 5 = 0$.
- A is the 1st letter, so $11 \cdot 1 \% 5 = 1$.
- S is the 19th letter, so $11 \cdot 19 \% 5 = 4$.
- Y is the 25th letter, so $11 \cdot 25 \% 5 = 0$.
- Q is the 17th letter, so $11 \cdot 17 \% e 5 = 2$.
- U is the 21st letter, so $11 \cdot 21 \% 5 = 1$.
- T is the 20th letter, so $11 \cdot 20 \% 5 = 0$.
- I is the 9th letter, so $11 \cdot 9 \% 5 = 4$.
- O is the 15th letter, so $11 \cdot 15 \% 5 = 0$.
- N is the 14th letter, so $11 \cdot 14 \% 5 = 4$.

See Figure 1.

Exercise 2. Develop an alternate implementation of `SeparateChainingHashST` that directly uses the linked-list code from `SequentialSearchST`.

Solution. See `com.segarciat.algs4.ch3.sec4.ex02.SeparateChainingHashST`.

Exercise 3. Modify your implementation of the previous exercises to include an integer field for each key-value pair that is set to the number of entries in the table at the time that pair is inserted. Then implement a method that deletes all keys (and associated values) for which the field is greater than a given integer `k`. *Note*: This extra functionality is useful in implementing the symbol table for a compiler.

Solution. See `com.segarciat.algs4.ch3.sec4.ex03.SeparateChainingHashST`.

Exercise 4. Write a program to find values of `a` and `m`, with `m` as small as possible, such that the hash function `(a * k) % m` for transforming the kth letter of the alphabet into a table index produces distinct values (no collisions) for the keys `S E A R C H X M P L`. The result is known as a *perfect hash function*.

Solution. See `com.segarciat.algs4.ch3.sec4.ex04.PerfectHashFunction`.

Exercise 5. Is the following implementation of `hashCode()` legal?

```
public int hashCode()
{ return 17; }
```

If so, describe the effect of using it. If not, explain why.

Solution. Yes, it is legal. However, because its return value is a constant expression, all values of this type will hash to the same index. In the case of a `SepareteChainingHashST`, this means that all key-value pairs will belong to the same list. This will result in poor performance.

Exercise 6. Suppose that keys are binary integers. For a modular hash function with prime $m > 2$, prove that any two binary integers that differ in exactly one bit have different hash values.

Solution.

Proof. Suppose x and y are binary numbers that differ in exactly one bit. Without loss of generality, say that x is larger than y, meaning that in the bit they differ, x has a 1 and y has a 0. Then $x - y$ has a 1 in the bit position where x and y differ, and 0s otherwise. Thus, $x - y$ is a power of 2. Since $m > 2$ and m is prime, it follows that m is not an even number, so $(x - y) \mod m$ is nonzero. This implies $x \mod m \neq y \mod m$, and hence, they both hash to different values. □

Exercise 7. Consider the idea of implementing modular hashing for integer keys with the code `(a * k) % m`, where `a` is an arbitrary fixed prime. Does this change mix up the bits sufficiently well that you can use nonprime `m`?

Solution. No. The mapping $k \mapsto a \cdot k$ is one-to-one so this variation simply re-arranges the inputs to the modular hashing function, but otherwise, it is equivalent to the modular hashing function `n % m`, where this time $n = a \cdot k$. As we have discussed, this requires a prime m for a reasonable spread.

Exercise 8. How many empty lists do you expect to see when you insert n keys into a hash table with `SeparateChainingHashST`, for $n = 10, 10^2, 10^3, 10^4, 10^5$, and 10^6? *Hint*: See **Exercise 2.5.31**.

Solution. Under the uniform hashing assumption, we expect that every integer will uniformly and independently distribute the keys among the integers between 0 and $m-1$. Since each key hashes to an integer in this interval, we can think of the n keys as n integers. Given n integers belonging to the interval $[0, m)$, we want the expected amount of distinct values, which in turn tells us the number of empty lists. **Exercise 2.5.31** says that we should expect this to be about $m \cdot (1 - e^{-\alpha})$, where $\alpha = \frac{n}{m}$. To see where this comes from, consider my probabilistic analysis below.

For each integer k satisfying $0 \leq k < m$, define X_k to be 1 if k occurs at least once in the collection of n integers, and 0 otherwise. Then X_k is a Bernoulli random variable with parameter p, where p is the probability that k belongs to the collection. The probability that k is chosen from a uniform distribution of m integers is $\frac{1}{m}$, and the probability that it is not chosen is $1 - \frac{1}{m}$. If we now perform n independent trials (justified by the uniform hashing assumption), the probability that m is never chosen among the n integers is $q = 1 - p$, and it equals

$$q = 1 - p = \left(1 - \frac{1}{m}\right)^n$$

Since X_k is a Bernoulli random variable, its expectation is given by

$$E(X_k) = p = 1 - \left(1 - \frac{1}{m}\right)^n$$

Let $X = \sum_{k=0}^{m-1} X_k$. Then X is a random variable representing the number of distinct values in the collection. By the linearity of expectation, we have

$$E(X) = E\left(\sum_{k=0}^{m-1} X_k\right) = \sum_{k=0}^{m-1} EX_k = \sum_{k=0}^{m-1} p = mp$$

Hence, there are about mp lists that are occupied, and equivalently, about $m - mp = mq$ lists that are empty. To see that this is consistent with **Exercise 2.5.31**, let $\alpha = \frac{n}{m}$. Then $\frac{\alpha}{n} = \frac{1}{m}$, so

$$\left(1 - \frac{1}{m}\right)^n = \left(1 - \frac{\alpha}{n}\right)^n$$

and thus

$$\lim_{n\to\infty} \left(1 - \frac{1}{m}\right)^n = \lim_{n\to\infty} \left(1 - \frac{\alpha}{n}\right)^n = e^{-\alpha}$$

so that

$$E(X) = mp = m \cdot \left(1 - \left(1 - \frac{1}{m}\right)^n\right) \approx m \cdot \left(1 - e^{-\alpha}\right)$$

For this exercise, we care about $mq \approx e^{-\alpha}$. Now, for the different n values, noting that `SeparateChainingHashST` uses $m = 997$ by default, we find can make the following table:

n	10	10^2	10^3	10^4	10^5	10^6
Empty lists	987	902	366	0	0	0

Exercise 9. Implement an eager `delete()` method for `SeparateChainingHashST`.

Solution. See `com.segarciat.algs4.ch3.sec4.ex09.SeparateChainingHashST`.

Exercise 10. Insert the keys `E A S Y Q U T I O N` in that order into an initially empty table of size $m = 16$ using linear probing. Use the hash function `11 * % m` to transform the kth letter of the alphabet into a table index. Redo this exercise for $m = 10$.

Solution. First we can compute the hashes. This time I will assume that kth letter implies k starts at 1, so with $m = 16$, the hash values become:

```
E: 7
A: 11
S: 1
Y: 3
Q: 11
U: 7
T: 12
I: 3
O: 5
N: 10
```

The table below shows the resulting hash table in the style of the text, where a red color indicates the newly inserted key, black indicates probes, and gray keys are not examined. I have shown only keys and omitted values:

key	hash	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
E	7								E								
A	11								E				A				
S	1		S						E				A				
Y	3		S		Y				E				A				
Q	11		S		Y				E				A	Q			
U	7		S		Y				E	U			A	Q			
T	12		S		Y				E	U			A	Q	T		
I	3		S		Y	I			E	U			A	Q	T		
O	5		S		Y	I	O		E	U			A	Q	T		
N	10		S		Y	I	O		E	U		N	A	Q	T		
			S		Y	I	O		E	U		N	A	Q	T		

Now for $m = 10$, the hash values are the following:

```
E: 5
A: 1
S: 9
Y: 5
Q: 7
U: 1
T: 0
```

```
I: 9
O: 5
N: 4
```

The corresponding table is shown below:

key	hash	0	1	2	3	4	5	6	7	8	9
E	5						E				
A	1		A				E				
S	9		A				E				S
Y	5		A				E	Y			S
Q	7		A				E	Y	Q		S
U	1		A	E			E	Y	Q		S
T	0	T	A	E			E	Y	Q		S
I	9	T	A	E	I		E	Y	Q		S
O	5	T	A	E	I		E	Y	Q	O	S
N	4	T	A	E	I	N	E	Y	Q	O	S
		T	A	E	I	N	E	Y	Q	O	S

Exercise 11. Give the contents of a linear-probing hash table that results when you insert the keys `E A S Y Q U T I O N` in that order into an initially empty table of initial size $m = 4$ that is expanded with doubling whenever half full. Use the hash function `11 * k % m` to transform the kth letter of the alphabet into a table index.

Solution. Since the table is initially empty, the first two keys `E` and `A` are hashed using $m = 4$:

```
E: 3
A: 3
```

Hence the table looks as follows:

key	hash	0	1	2	3
E	3				E
A	3	A			E

When we go on to insert `S`, the test `n >= m / 2` passes because `n` is now 2 and `m / 2` is 2, so the table size is doubled to 8. In the process, the previous keys `E` and `A` are rehashed, and then the size stays under $m/2$ in size until `Y` is inserted. The hash values are now as follows:

```
E: 7
A: 3
S: 1
Y: 3
```

The table now looks as such:

key	hash	0	1	2	3	4	5	6	7
E	7								E
A	3				A				E
S	1		S		A				E
Y	3		S		A	Y			E

Now the table size doubles to $m = 16$ as we are about to insert `Q`. Thus we rehash all current values, and insert at least 4 more keys (to reach a size of 8) before having to resize again:

```
E: 7
A: 11
S: 1
Y: 3
Q: 11
U: 7
T: 12
I: 3
```

The table now looks as follows:

key	hash	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
E	7								E								
A	11								E				A				
S	1		S						E				A				
Y	3		S		Y				E				A				
Q	11		S		Y				E				A	Q			
U	7		S		Y				E	U			A	Q			
T	12		S		Y				E	U			A	Q	T		
I	3		S		Y	I			E	U			A	Q	T		

Currently we are at $n = 8$ and $m = 16$, with the keys `O` and `N` remaining. We must resize so that $m = 32$, so rehashing and hashing the new keys yields:

```
E: 23
A: 11
S: 17
Y: 19
Q: 27
U: 7
T: 28
I: 3
O: 5
N: 26
```

Now for this last resize, there are no collisions, so the location on the table for each key is precisely the value of the hash after modding by m.

Exercise 13. Which of the following scenarios leads to expected *linear* running time for a random search hit in a linear-probing hash table?

(a) All keys hash to the same index.

(b) All keys hash to different indices.

(c) All keys hash to an even-numbered index.

(d) All keys hash to different even-numbered indices.

Solution.

(a) If all keys hash to the same index, then a cluster of size n is created. Searching for a random value in that cluster takes linear time.

(b) If all keys hash to different indices, then we expect a search hit on the first probe.

(c) I expect this to be the same as part (a).

(d) I expect this to be the same as part (b).

Exercise 14. Answer the previous question for a search *miss*, assuming that the search key is equally likely to hash to each table position.

Solution. I expect the answers to the exactly the same.

Exercise 15. How many compares could it take, in the worst case, to insert n keys into an initially empty table, using linear probing with array resizing?

Solution. In the worst case, all keys hash to the same value. With resizing, all keys that are already in the table are rehashed when the table size doubles. No compares are necessary for the first key. When there are $k \geq 1$ keys already in the table, we need $k-1$ compares because the fact all keys hash to the same value implies a run of length k. Thus, the number of compares (assuming no resizing) would be:

$$1 + 2 + \cdots + n - 1 = \frac{n(n-1)}{2}$$

Next, we account for resizing. When the table size is $k \geq 1$, so that k keys need to be rehashed, the number of compares is $\frac{k(k-1)}{2}$. The number of times the array is resized depends on the original table size and the value of n. For simplicity, assume that the initial table size is 2, and that $n = 2^b$ for some integer b. Then the table is resized when the table has 1 item, then again when it has 2 items, and so on, until it has 2^b items. The table size the ends at size 2^{b+1}, with $n = 2^b$ items in the table. Effectively, the table is resized b times. When it resizes from 2^j to $2^j + 1$, where $1 \leq j \leq b$, the table has 2^{j-1} items to rehash, and the necessary number of compares is:

$$1 + 2 + \cdots + 2^{j-1} - 1 = \frac{2^{j-1} \cdot (2^{j-1} - 1)}{2} = 2^{j-2} \cdot (2^{j-1} - 1) = 2^{2j-3} - 2^{j-2}$$

Now we sum from $j = 1$ to $j = b$:

$$\sum_{j=1}^{b} \left(2^{2j-3} - 2^{j-2}\right) = \frac{1}{8} \sum_{j=1}^{b} 4^j - \frac{1}{4} \sum_{j=1}^{b} 2^j$$

The resulting expression is roughly on the same order of magnitude as $n(n-1)/2$, where $n = 2^b$.

Exercise 16. Suppose that a linear-probing table of size 10^6 is half full, with occupied positions chosen at random. Estimate the probability that all positions with indices divisible by 100 are occupied.

Solution. Suppose a random trial consists of choosing an integer position at random from 0 to $10^6 - 1$. Let $X_k = 1$ if the kth multiple of 100 less than 10^6 is chosen at least once after n trials, and 0 otherwise. Here, $n = \frac{1}{2}10^6$ because the table is half-full. Since all positions are equally likely, the probability k is chosen on a given trial is $\frac{1}{10^6}$. The probability that it is not chosen among the n values is $q = \left(1 - \frac{1}{10^6}\right)^n$. Thus, $P(X_k = 0) = q$, and $P(X_k = 1) = p = 1 - q$.

Note that that there are $10^6/100 = 10^4$ multiples of 100 less than 10^6 (including 0). Let $X = \sum_{k=1}^{10^4} X_k$. Since all X_k are Bernoulli with the same parameter p, and since they are independent, X is binomial. Therefore, the probability that all 10^4 multiples are chosen is $P(X = 10^4)$, which is given by

$$P(X = 10^4) = \binom{10^4}{10^4} p^{10^4}(1-p)^{10^4 - 10^4} = p^{10^4} = \left(1 - \left(1 - \frac{1}{10^6}\right)^n\right)^{10^4}$$

Let m be the table size, n be the number of keys occupied in the table, and $\alpha = \frac{n}{m}$. Then $m = 10^6$, $n = \frac{1}{2}m$, and $\alpha = \frac{1}{2}$. We can re-write and approximate the expression above as

$$P(X = 10^4) = \left(1 - \left(1 - \frac{\alpha}{n}\right)^n\right)^{10^4} \approx (1 - e^{-\alpha})^{10^4}$$

Here, $1 - e^{-\alpha} \approx 0.39$, and raising this to the 10^4 power is exceedingly small.

Exercise 17. Show the result of using the `delete()` method on page 471 to delete `C` from the table resulting from using `LinearProbingHashST` with our standard indexing client (shown on page 469).

Solution. The table on 469 looks as follows:

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
key	P	M			A	C	S	H	L		E				R	X

Here, `C` belongs to the cluster made up by `A C S H L`. When `delete("C")` is issued, the keys that come after it in the cluster, namely `S H L`, are rehashed. From the example, we know that `S` hashes to `6`, `H` hashes to `4`, and `L` hashes to `6`. This results in `S` being at the same position, then `H` ends at position 5 after a collision at index 4 with `A`, and `L` ends at position 7 after a collision at index 6 with `S`:

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
key	P	M			A	H	S	L			E				R	X

Exercise 18. Add a constructor to `SeparateChainingHashST` that gives the client the ability to specify the average number of probes to be tolerated for searches. Use array resizing to keep the average list size less than the specified value, and use the technique described on page 478 to ensure that the modulus for `hash()` is prime.

Solution. See `com.segarciat.algs4.ch3.sec4.ex18.SeparateChainingHashST`.

Exercise 19. Implement `keys()` for `SeparateChainingHashST` and `LinearProbingHashST`.

Solution. See `com.segarciat.algs4.ch3.sec4.ex19`.

Exercise 20. Add a method to `LinearProbingHashST` that computes the average cost of a search hit in the table, assuming that each key in the table is equally likely to be sought.

Solution. See `com.segarciat.algs4.ch3.sec4.ex20.LinearProbingHashST`.

Exercise 21. Add a method to `LinearProbingHashST` that computes the average cost of a search *miss* in the table, assuming a random hash function. *Note*: You do not have to compute any hash functions to solve this problem.

Solution. See `com.segarciat.algs4.ch3.sec4.ex21.LinearProbingHashST`.

Exercise 22. Implement `hashCode()` for various types: `Point2D`, `Interval`, `Interval2D`, and `Date`.

Solution. See `com.segarciat.algs4.ch3.sec4.ex22`.

Exercise 23. Consider modular hashing for string keys with `R = 256` and `m = 255`. Show that this is a bad choice because any permutation of letters within a string hashes to the same value.

Solution. Modular hashing would treat each characters as a base-256 integer. If c is such a character, then a single character would be hashed as follows:

$$(R \cdot hash + c) \mod m = (256 \cdot hash + c) \mod 255 = (hash + c) \mod 255$$

where the last equality follows from the fact that $256 = 1 \mod 255$. In a multi-character string, each character would be weighted by a power of R that corresponds to its position. However, because $R \mod m$ is 1, all characters are weighed the same. Thus, the position of characters becomes irrelevant, implying that order no longer places a role and hence any permutation leads to the same hash.

Exercise 24. Analyze the space usage of separate chaining, linear probing, and BSTs for `double` keys. Present your results in a table like the one on page 476.

Solution.

- `SeparateChainingHashST`: There are 16 bytes of object overhead, 4 bytes for the reference to its `m` field, and 8 bytes for the reference variable `st`, and 24 bytes for the array object itself`st` (16 bytes overhead, 4 bytes for its length field, and 4 bytes padding). With 4 bytes of padding, we arrive at 56 bytes. Now, there are m `SequentialSearchST` objects, each requiring 32 bytes when empty (see **Exercise 3.1.21**), and an 8 byte reference to them, for a total of $8m$ bytes worth of references. Overall, this means $40m$ bytes. If there are n `double` in the table (regardless of which `SequentialSearchST` they are in), then the extra cost is $56n$, because each is store in a `Node` that takes up about 48 bytes, plus the actual cost for the `double` themselves, which are autoboxed to `Double`. Overall, the overhead is about $56 + 40m + 56n$, plus the cost for the `double` themselves.

- `LinearProbingHashST`: There are 16 bytes of object overhead, 4 bytes for the `n` field, 4 bytes for the `m` field, 8 bytes for the reference to the `keys` array and 24 bytes for the array itself thus totaling 32 due to `keys`, and similarly for the `vals` array, for a total of 88 bytes. Now, for each key-value pair, there are 8 bytes for a reference to a key and 8 bytes for a reference to a value. When the table is half-full, meaning $n = m/2$, we have $2 \cdot 8n$ for the keys and $2 \cdot 8n$ for the values (times 2 because of the reference to unoccupied entries). It could also be one-eight full, in which case it's $8 \cdot 8n$ in both cases. Altogether, the cost is around $88 + 32n$ to $88 + 128n$. This does not account for the `double` values themselves, and the fact that they would be subject to autoboxing to `Double`.

Exercise 25. *Hash cache.* Modify `Transaction` on page 462 to maintain an instance variable `hash`, so that `hashCode()` can save the hash value the first time it is called for each object, and does not have to recompute it on subsequent calls. *Note*: This idea works only for immutable types.

Solution. See `com.segarciat.algs4.ch3.sec4.ex25.Transaction`.

Exercise 26. *Lazy delete for linear probing.* Add to `LinearProbingHashST` a `delete()` method that deletes a key-value pair by setting the value to `null` (but not removing the key) and later removing the pair from the table in `resize()`. Your primary challenge is to decide when to call `resize()`. *Note*: You should overwrite the `null` value if a subsequent `put()` operations associates a new value with the key. Make sure that your program takes into account the number of such *tombstone* items, as well as the number of empty positions, in making the decision whether to expand or contract the table.

Solution. See `com.segarciat.algs4.ch3.sec4.ex26.LinearProbingHashST`.

Exercise 30. *Chi-square statistic.* Add a method to `SeparateChainingHashST` to compute the χ^2 statistic for the hash table. With n keys and table size m, this number is defined by the equation:

$$\chi^2 = (m/n) \cdot ((f_0 - n/m)^2 + (f_1 - n/m)^2 + \cdots + (f_{m-1} - n/m)^2)$$

where f_i is the number of keys with has value i. This statistic is one way of checking our assumption that the hash function produces random values. If so, this statistic obeys a χ^2 distribution with $m - 1$ degrees of freedom, so its mean is $m - 1$, and its variance is $2 \cdot (m - 1)$.

Exercise 34. *Hash cost.* Determine empirically the ratio of the time required for `hash()` to the time required for `compareTo()`, for as many commonly-used types of keys for which you can get meaningful results.

Solution. See `com.segarciat.algs4.ch3.sec4.ex34`.

Exercise 36. *List length range.* Write a program that inserts n random `int` keys into a hash table of size $n/100$ using separate chaining, then finds the length of the shortest and longest lists, for $n = 10^3, 10^4, 10^5, 10^6$.

Solution. See `com.segarciat.algs4.ch3.sec4.ex36`.

Exercise 38. *Separate-chaining distribution.* Write a program that inserts 10^5 random non-negative integers less than 10^6 into a hash table of size 10^5 using separate chaining, and that plots the total cost for each 10^3 consecutive insertions. Discuss the extent to which your results validate **Proposition K**.

Solution. See `com.segarciat.algs4.ch3.sec4.ex38`.

References

[SW11] Robert Sedgewick and Kevin Wayne. *Algorithms*. 4th ed. Addison-Wesley, 2011. ISBN: 9780321573513.