This improved solution is implemented below:

```
std::vector<std::string> minimum_index_sum(const std::vector<std::string>& vec1,
                                           const std::vector<std::string>& vec2) {
  std::vector<std::string> solution;
  size_t best_idx_sum = std::numeric_limits<size_t>::max();

  // insert all the strings in one vector into a hash table that maps string -> index
  std::unordered_map<std::string, size_t> idx_map;
  for (size_t idx1 = 0; idx1 < vec1.size(); ++idx1) {
    idx_map.emplace(vec1[idx1], idx1);
  } // for idx1

  // iterate over the other vector, querying the hash table to identify index in first vector
  for (size_t idx2 = 0; idx2 < vec2.size(); ++idx2) {
    std::string& curr = vec2[idx2];
    auto it = idx_map.find(curr);
    if (it != idx_map.end()) {
      size_t idx_sum = it->second + idx2;
      if (idx_sum < best_idx_sum) {
        solution.clear();
        solution.push_back(curr);
        best_idx_sum = idx_sum;
      } // if
      else if (idx_sum == best_idx_sum) {
        solution.push_back(curr);
      } // else if
    } // if
  } // for idx2

  return solution;
} // minimum_index_sum()
```

Since our loops are no longer nested and are instead performed sequentially, the time complexity now becomes $\Theta(m+n)$, where m and n are the lengths of the two vectors. This is because the bodies of both loops can be completed in average-case constant time (since insertion and lookup in a hash table can both be done in $\Theta(1)$ time on average). Notice that an additional optimization we can make is to add the *smaller* of the two vectors to the unordered map — this allows us to reduce our memory usage, as we only need to store the contents of one of the vectors.

Chapter 17 Practice Exercises

Disclaimer: These practice questions are not official and may not have been vetted for course material. You may use these for practice, but you should prioritize official resources from current staff for a more accurate depiction of what you need to know for assignments and exams.

1. Given a hash table (`std::unordered_map<>`) that maps all the items in a grocery store to their prices, which of the following operations can be performed in $\Theta(1)$ time on average?
   - **I.** Finding the price of a given item
   - **II.** Finding the cheapest item in the store
   - **III.** Updating the price of a given item

   A) I only
   B) III only
   C) I and II only
   D) I and III only
   E) I, II, and III

2. Which of the following statements is/are **TRUE**?
   - **I.** Using `operator[]` on a non-existent key in a `std::unordered_map<>` will insert the key into the container.
   - **II.** The worst-case time complexity of finding an element in a `std::unordered_map<>` of size n is $\Theta(1)$.
   - **III.** The `rand()` function, which generates random numbers, is great for hashing since it greatly reduces the chances of collision.

   A) I only
   B) I and II only
   C) I and III only
   D) II and III only
   E) I, II, and III

3. Which of the following is **NOT** a characteristic of a good hash function?
   A) The capability to distribute keys evenly in a hash table
   B) The capability to keep similar keys close together in a hash table
   C) The capability to compute a hash for every possible key
   D) The capability to compute the same hash for the same key
   E) All of the above are characteristics of a good hash function

4. The scores for the Winter 2018 EECS 281 midterm exam fall into the range [17, 99). If we were to input these scores into a hash table of size 41 using the following hash function:
$$h(key) = \lfloor \frac{key-s}{t-s} \times M \rfloor$$
where the relevant range is $[s,t)$ and the hash table size is M, a score of 83 would end up at which index of the hash table?
   A) 31
   B) 32
   C) 33
   D) 34
   E) 35

5. An easy way to compress an integer key into a hash table of size M is to take its modulus with M (i.e., `key` mod M). For which of the following collection of keys is this compression method the most ideal?
   A) A list of all scores on a 20-question multiple choice exam, where each question is worth 5 points
   B) The average number of hours of sleep each students gets on the weekend, rounded up
   C) The number of credits each full time student at the university is taking this semester
   D) The collection of student IDs of all students currently enrolled in EECS 281
   E) All of the above are equally ideal use cases

For questions 6-8, consider the following code:

```
1   int main() {
2      std::unordered_map<std::string, std::string> my_map;
3      my_map.insert(std::make_pair("Paoletti","Darden"));
4      my_map.insert(std::make_pair("Angstadt","Darden"));
5      my_map.insert(std::make_pair("Paoletti","Angstadt"));
6      my_map["Angstadt"] = "Paoletti";
7      my_map.insert(std::make_pair("Paoletti", "Garcia"));
8      std::cout << my_map["Paoletti"] << std::endl;
9      std::cout << my_map["Darden"] << std::endl;
10     my_map.erase("Paoletti");
11     std::cout << my_map["Angstadt"] << std::endl;
12     std::cout << my_map.size() << std::endl;
13  } // main()
```

6. What does line 8 print?
   A) `Paoletti`
   B) `Darden`
   C) `Angstadt`
   D) `Garcia`
   E) Any empty string

7. What does line 11 print?
   A) `Paoletti`
   B) `Darden`
   C) `Angstadt`
   D) `Garcia`
   E) Any empty string

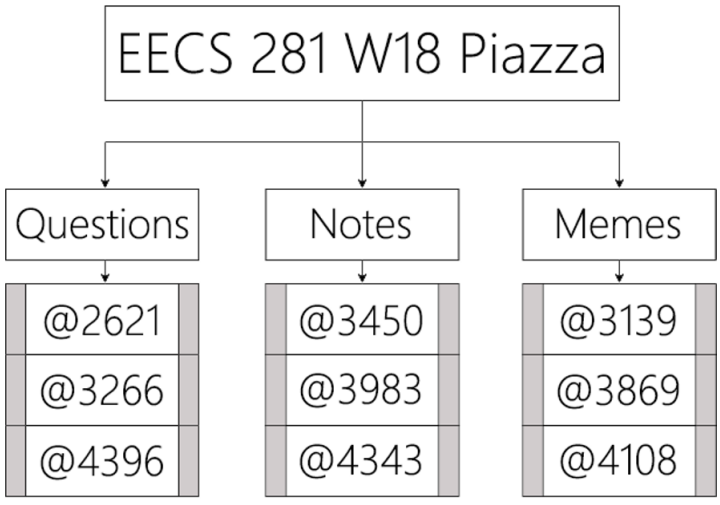
8. What does line 12 print?
   A) `1`
   B) `2`
   C) `3`
   D) `4`
   E) `7`

9. Consider the following snippet of code:

```
1   int main() {
2      std::unordered_map<int32_t, double> umap;
3      m[280] = 281;
4      m[203] = 376;
5      for (auto x : umap) {
6        // do stuff
7      } // for
8   } // main()
```

   What is the type of `x` on line 5?
   A) `int32_t`
   B) `double`
   C) `std::pair<int32_t, double>`
   D) `std::unordered_map<int32_t, double>`
   E) None of the above

10. On the Winter 2018 EECS 281 Piazza page, posts can be categorized into three unique groups: questions, notes, and memes. A representation of this is shown below:



Suppose you implemented the following unordered map that maps the type of each post to the post IDs that correspond to each of these types (for instance, the key `notes` maps to `[3450,3983,4343]`):

```
std::unordered_map<std::string, std::vector<int>> posts;
```

While searching through Piazza, you discover that post @4753 is a meme. Which of the following successfully appends 4753 to the vector associated with the key `memes`?

A) `posts["memes"] = 4753;`
B) `posts["memes"] = memes.push_back(4753);`
C) `posts["memes"] = posts.push_back(4753);`
D) `posts["memes"].push_back(4753);`
E) `posts["memes"].second.push_back(4753);`

For questions 11-12, consider the following code:

```
1   int main() {
2      std::unordered_map<std::string, std::pair<std::string, int32_t>> foods;
3      foods["cabbage"] = std::make_pair("vegetable", 1);
4      foods["banana"] = std::make_pair("fruit", 2);
5      foods["donut"] = std::make_pair("dessert", 3);
6      foods["apple"] = std::make_pair("fruit", 4);
7      foods["eggplant"] = std::make_pair("vegetable", 5);
8      std::vector<std::string> vec;
9      for (const auto& x : foods) {
10        vec.push_back(x.first);
11     } // for
12     std::cout << vec.front() << std::endl;
13     return 0;
14  } // main()
```

11. What does line 12 print?
A) `apple`
B) `fruit`
C) `cabbage`
D) `vegetable`
E) Impossible to determine

12. Which of the following lines of code prints out `1`?
A) `std::cout << foods[0] << std::endl;`
B) `std::cout << foods["cabbage"] << std::endl;`
C) `std::cout << foods["cabbage"].second << std::endl;`
D) `std::cout << foods["cabbage"].second.second << std::endl`
E) More than one of the above

13. Given two unsorted arrays of distinct numbers of size n, you want to find all pairs of numbers, one from array 1 and one from array 2, that sum to a given value. For instance, if you are given

```
arr1[] = {1, 2, 3, 4, 5, 7, 11}
arr2[] = {2, 3, 4, 5, 6, 8, 12}
```

you would return `(1, 8)`, `(3, 6)`, `(4, 5)`, `(5, 4)`, and `(7, 2)`. What is the average-case time complexity of doing this if you use the most efficient algorithm?

A) $\Theta(1)$
B) $\Theta(\log(n))$
C) $\Theta(n)$
D) $\Theta(n\log(n))$
E) $\Theta(n^2)$

14. Which of the following collision resolution methods is **NOT** a form of open addressing?

A) Separate chaining
B) Linear probing
C) Quadratic probing
D) Double hashing
E) All of the above use open addressing

15. Which of the following statements is **FALSE**?

A) The load factor α of a hash table can exceed 1
B) As α increases, the performance of separate chaining does not deteriorate as quickly as the performances of open addressing methods
C) The time complexities of searching and removing are both $\Theta(\alpha)$ on average for a hash table that uses separate chaining to resolve collisions
D) If α is less than 0.5, linear probing is better than double hashing at preventing keys in a hash table from clustering together
E) More than one of the above

16. Which of the following is **NOT** a disadvantage of using quadratic probing to resolve collisions?

A) Two elements that hash to the same position will still have the same probe sequence, regardless of how far they land from their hashed location
B) Quadratic probing is susceptible to primary clustering, where keys typically end up next to each other rather than distributed throughout the entire hash table
C) Depending on the size of the hash table involved, it is possible for quadratic probing to never consider specific indices while searching for an open position
D) The performance of quadratic probing can deteriorate dramatically as the load factor increases
E) None of the above

17. A hash table of size 100 has 40 empty positions and 25 deleted positions. What is its load factor?

A) 0.25
B) 0.35
C) 0.60
D) 0.65
E) 0.75

18. You are given a hash table that immediately doubles its size whenever the load factor becomes ≥ 0.6. You notice that, immediately after inserting an element into this hash table, its size doubled from 75 to 150. How many elements must be in the hash table after your insertion (assuming that no other elements are added or removed)?

A) 30
B) 45
C) 60
D) 75
E) 90

19. Suppose you have a hash table of size $M = 10$ that uses the hash function $H(n) = 3n+7$ and the compression function $C(n) = n \bmod M$. **Linear probing** is used to resolve collisions. You enter the following nine elements into this hash table in the following order: `{10, 8, 18, 17, 4, 20, 6, 3, 16}`. No resizing is done. After all collisions are resolved, which index of the hash table remains empty?

A) 3
B) 4
C) 5
D) 6
E) None of the above

20. Suppose you have a hash table of size $M = 10$ that uses the hash function $H(n) = 5n + 7$ and the compression function $C(n) = n \bmod M$. **Linear probing** is used to resolve collisions. You enter the following nine elements into this hash table in the following order: `{1, 2, 3, 4, 5, 6, 7, 8, 9}`. No resizing is done. After all collisions are resolved, which index of the hash table remains empty?
    A) 1
    B) 2
    C) 3
    D) 9
    E) None of the above

21. Suppose you have a hash table of size $M = 7$ that uses the hash function $H(n) = n$ and the compression function $C(n) = n \bmod M$. **Quadratic probing** is used to resolve collisions. You enter the following six elements into this hash table in the following order: `{24, 11, 17, 21, 10, 4}`. No resizing is done. After all collisions are resolved, which index of the hash table remains empty?
    A) 1
    B) 2
    C) 5
    D) 6
    E) None of the above

22. Suppose you have a hash table of size $M = 13$ that uses the hash function $H(n) = 2n + 3$ and the compression function $C(n) = n \bmod M$. **Quadratic probing** is used to resolve collisions. You enter the following twelve elements into this hash table in the following order: `{9, 22, 7, 10, 3, 1, 20, 13, 18, 5, 0, 11}`. No resizing is done. After all collisions are resolved, which index of the hash table remains empty? Note: the following are the first five multiples of 13, in case you find them useful: 13, 26, 39, 52, 65.
    A) 2
    B) 6
    C) 11
    D) 12
    E) None of the above

23. A hash table of size 10 uses open addressing with a hash function $H(k) = k$, compression function $C(k) = k \bmod 10$, and linear probing. After entering six values into an empty hash table, its state is as shown below. Which of the following insertion orders is possible?

0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
		62	43	24	82	76	53		

    A) 76, 62, 24, 82, 43, 53
    B) 24, 62, 43, 82, 53, 76
    C) 76, 24, 62, 43, 82, 53
    D) 62, 76, 53, 43, 24, 82
    E) None of the above

24. Suppose you are using a hash function where each string is hashed to an integer representing its first letter. The integer each letter hashes to is shown in the table below:

a	**b**	**c**	**d**	**e**	**f**	**g**	**h**	**i**	**j**	**k**	**l**	**m**	**n**
0	1	2	3	4	5	6	7	8	9	10	11	12	13

o	**p**	**q**	**r**	**s**	**t**	**u**	**v**	**w**	**x**	**y**	**z**
14	15	16	17	18	19	20	21	22	23	24	25

    The compression function $C(x) = x \bmod M$ is used, where M represents the size of the hash table. You are using a hash table of size M. If the strings "paoletti" and "darden" collide in this hash table, which of the following is not a possible value of M?
    A) 3
    B) 6
    C) 9
    D) 12
    E) None of the above

25. Suppose you have a hash table of size $M = 7$ that uses the hash function $H(n) = 2n + 1$ and the compression function $C(n) = n \bmod M$. **Double hashing** is used to resolve collisions, with the following double hashing formula:

$$H(n) + j(5 - (H(n) \bmod 5))$$

    where $H(n)$ represents the integer value the key normally hashes to without compression, $(5 - (H(n) \bmod 5))$ represents the secondary hash function, and j is the collision number. You enter the following six elements into this hash table in the following order: `{6, 10, 4, 13, 11, 12}`. No resizing is done. After all collisions are resolved, which index of the hash table remains empty?
    A) 1
    B) 3
    C) 4
    D) 5
    E) None of the above

26. Suppose you are using a hash function where each string is hashed to an integer representing its first letter. The integer each letter hashes to is shown in the table below:

a	b	c	d	e	f	g	h	i	j	k	l	m	n
0	1	2	3	4	5	6	7	8	9	10	11	12	13

o	p	q	r	s	t	u	v	w	x	y	z
14	15	16	17	18	19	20	21	22	23	24	25

You have a hash table of size 10, and you insert the planets of the solar system (with the sun) into this hash table in the following order:

1. "sun"
2. "mercury"
3. "venus"
4. "earth"
5. "mars"
6. "jupiter"
7. "saturn"
8. "uranus"
9. "neptune"

Collisions are resolved using the **double hashing** formula

$$t(key) + j \times (q - (t(key) \bmod q))$$

where $t(key)$ is the integer that a key hashes to, j is the number of collisions that have occurred so far with that key, and q is 7. Compression is done using the formula $C(n) = n \bmod M$, where M is 10 (the size of the hash table). No resizing is done. After all collisions are resolved, which index of the hash table remains empty?

A) 2
B) 5
C) 6
D) 7
E) None of the above

27. Suppose you have a hash table of size $M = 13$ that uses the same hash function as mentioned previously. You enter the following foods as keys into this hash table in the following order:

1. "apples"
2. "almonds"
3. "avocados"
4. "asparagus"

Collisions are resolved using the **double hashing** formula:

$$t(key) + j \times (q - (t(key) \bmod q))$$

After all collisions are resolved, you notice that the key "asparagus" ended up at index 7. Knowing this, what is a possible value of q?

A) 3
B) 5
C) 7
D) 9
E) 11

28. Suppose you attempted to insert the same four keys into a different hash table, in the same order. Collisions are still resolved using the **double hashing** formula:

$$t(key) + j \times (q - (t(key) \bmod q))$$

You do not know the size M of this hash table. However, you do know that the value of q is 7 and that the key "asparagus" ended up at index 4. With this information, what is a possible value of M?

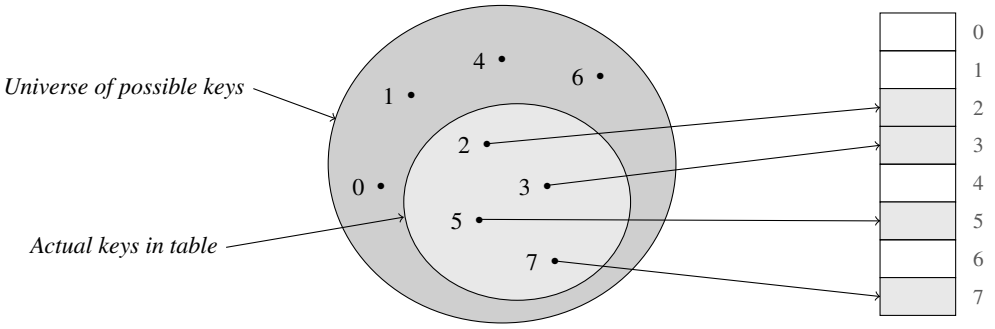
A) 11
B) 13
C) 15
D) 17
E) 19

29. Which of the following statements is **TRUE**?

I. As the load factor of a hash table increases, the average-case performance of finding a key deteriorates.
II. A hash table that uses open addressing can still perform reasonably well with a load factor of $\alpha = 1.1$.
III. A hash table that uses separate chaining can still perform reasonably well with a load factor of $\alpha = 1.1$.

A) I only
B) I and II only
C) I and III only
D) II and III only
E) I, II, and III

30. If the universe of possible keys we want to store in a hash table can be drawn from the set $\{0, 1, \ldots, m-1\}$, where m is small, we can use the value of each key to directly access the underlying table, as shown in the figure below.



This addressing method is known as

A) Direct addressing
B) Hashed addressing
C) Dynamic addressing
D) Dynamic hashing
E) Collision resolution

31. Consider the following snippet of code:

```
1   struct Thing {
2     int32_t x;
3     int32_t y;
4
5     Thing(int32_t x_in, int32_t y_in)
6       : x{x_in}, y{y_in} {}
7   };
8
9   int main() {
10    std::unordered_map<int32_t, Thing> thing_map;
11    Thing my_thing{280, 281};
12
13  } // main()
```

Which of the following statements, when added on line 12, would run without any compilation issues?

I. `thing_map[183] = my_thing;`
II. `thing_map.insert(183, my_thing);`
III. `thing_map.emplace(183, my_thing);`

A) I only
B) III only
C) I and III only
D) II and III only
E) I, II, and III

32. For which of the following applications would a hash table be least efficient?

A) Filtering duplicates from a list of elements
B) Identifying all unique values from a list of elements
C) Printing out values from a list of elements in sorted order
D) Counting the frequencies of values in a list of elements
E) All of the above can be efficiently solved with a hash table

33. Which of the following collision resolution techniques does **NOT** need to keep track of a special marker for deleted items?

A) Separate chaining
B) Linear probing
C) Quadratic probing
D) Double hashing
E) None of the above

34. Which one of the following four hash functions is most ideal for a `std::string`?

Hash 1

```
int32_t hash1(const std::string& s) {
  int32_t hash = 7, n = s.size();
  for (int32_t i = 0; i < n; ++i)
    hash += rand();
  return hash;
} // hash1()
```

Hash 2

```
int32_t hash2(const std::string& s) {
  int32_t hash = 7, n = s.size();
  for (int32_t i = 0; i < n; ++i)
    hash += s[i];
  return hash;
} // hash2()
```

Hash 3

```
int32_t hash3(const std::string& s) {
  int32_t hash = 7, n = s.size();
  for (int32_t i = 0; i < n; ++i)
    hash += s[i] * i;
  return hash;
} // hash3()
```

Hash 4

```
int32_t hash4(const std::string& s) {
  int32_t hash = 7, n = s.size();
  for (int32_t i = 0; i < n; ++i)
    hash += s[i] * rand();
  return hash;
} // hash4()
```

A) Hash 1
B) Hash 2
C) Hash 3
D) Hash 4
E) All of the above work equally well

35. Suppose you had a hash function $H(k)$ that you want to use to hash items into a hash table of size 10. The state of the hash table is shown below, where `X` represents an occupied position and `DELETED` represents a position that previously stored an element. Suppose you want to search for an element K that does not exist in the table, and $H(K) = 5$. If the hash table uses quadratic probing as its collision resolution method, which positions of the hash table must be probed before you can safely conclude that K is not in the table?

			X	X	X	DELETED		X	DELETED
0	1	2	3	4	5	6	7	8	9

A) 5, 6
B) 5, 6, 0
C) 5, 6, 7
D) 5, 6, 9, 3, 0
E) 5, 6, 9, 4, 1

36. Which of the following is the most appropriate situation to use a hash table?
A) Keeping track of the top three values from a stream of integers
B) Finding the highest priority thread to execute in a program, where each thread is assigned a priority value
C) Keeping track of the number of students who have birthdays on each of the 31 days in January
D) Finding information on a vehicle before a used car purchase, when given a 17-character vehicle identification number (VIN)
E) Printing out a list of graduates in descending order of cumulative GPA

37. You want to write a hash function that will distribute keys over 10 buckets, number 0 to 9, for integer keys i ranging from 0 to 2019. The hash table is implemented using separate chaining. Of the following hash and compression functions, which one would be best at distributing the keys uniformly over all 10 buckets?
A) $H(i) = 3i^2 \bmod 10$
B) $H(i) = 6i^2 \bmod 10$
C) $H(i) = 9i^3 \bmod 10$
D) $H(i) = 12i^3 \bmod 10$
E) $H(i) = 15i^4 \bmod 10$

38. Suppose you have a hash table of size $M = 7$ that uses the hash function $H(n) = n^2$ and the compression function $C(n) = n \bmod M$. **Double hashing** is used to resolve collisions, with a secondary hash function of $H'(n) = n^3$. The full double hashing equation is shown below:

$$H(n) + j \times H'(n)$$

where $H(n)$ represents the integer value the key normally hashes to without compression, $H'(n)$ represents the secondary hash function, and j is the collision number. You enter the following five elements into this hash table in the following order: `{7, 6, 5, 4, 3}`. No resizing is done. After all collisions are resolved, which index of the hash table does `3` end up at? Note: the following are the first twenty multiples of 7, in case you find them useful: 7, 14, 21, 28, 35, 42, 49, 56, 63, 70, 77, 84, 91, 98, 105, 112, 119, 126, 133, 140.

A) 0
B) 1
C) 2
D) 6
E) None of the above

39. You are given the following hash table of initial size $M = 7$, which doubles in size and rehashes all its elements when its load factor becomes ≥ 0.5. Duplicates are not allowed. The current state of the hash table is shown below.

0	1	2	3	4	5	6
`aardvark`	`hamster`					`giraffe`

The hash function used is $H(k) = k[0] - \text{`}a\text{'}$, or the distance of the first character of the string from 'a' (words that start with 'a' are hashed to 0, words that start with 'b' are hashed to 1, etc.). The integer each letter hashes to is shown in the table below for convenience. Compression is done using the formula $C(n) = n \bmod M$, where M is the size of the table. Linear probing is used as the collision resolution technique.

a	**b**	**c**	**d**	**e**	**f**	**g**	**h**	**i**	**j**	**k**	**l**	**m**	**n**
0	1	2	3	4	5	6	7	8	9	10	11	12	13

o	**p**	**q**	**r**	**s**	**t**	**u**	**v**	**w**	**x**	**y**	**z**
14	15	16	17	18	19	20	21	22	23	24	25

You perform the following operations on the given hash table:

1. Delete `"giraffe"`
2. Insert `"fish"`
3. Insert `"lion"`
4. Insert `"frog"`
5. Insert `"ferret"`
6. Delete `"hamster"`
7. Insert `"fox"`

How many positions of the hash table must be probed before you can insert `"fox"` into the table?

A) 2
B) 3
C) 4
D) 5
E) 6

40. What index of the final hash table does `"fox"` end up in after the insertion?

A) 1
B) 3
C) 5
D) 7
E) 9

41. Suppose you have an empty hash table of size M, where each string is hashed to an integer representing its first letter ($a = 0$, $b = 1$, $c = 2$, …, see table from question 39 for the full alphabet). You enter the following strings as keys into this hash table in the following order:

1. "bear"
2. "badger"
3. "bat"
4. "buffalo"

Collisions are resolved using the **double hashing** formula:

$$t(key) + j \times (q - (t(key) \bmod q))$$

The value of q is 7, and compression is performed using the formula $C(n) = n \bmod M$, where M is the size of the table. No resizing is done. After all four strings are inserted, you iterate through the underlying vector from beginning to end and print out the keys in the order in which they are encountered. If `"buffalo"` is the first of the four keys to be printed out, what is a potential value of M?

A) 13
B) 17
C) 19
D) 25
E) There exists no such M that satisfies this condition

42. You are tasked to build a hash table, but you are only allowed to use hash functions that take *linear* time on the size of the input. If you know that your input will be substantially large, which of the following collision resolution methods is the least appropriate?
    A) Separate chaining
    B) Linear probing
    C) Quadratic probing
    D) Double hashing
    E) None of the above

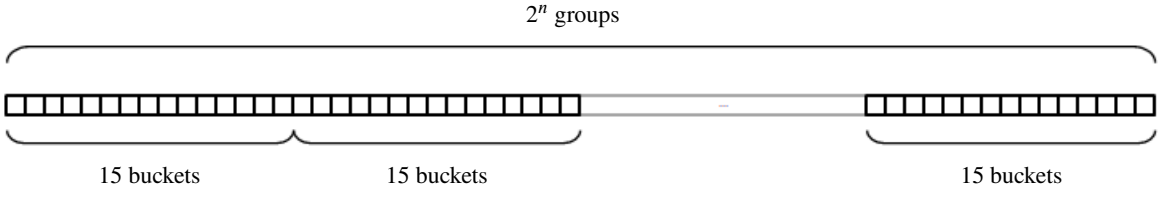
43. Which of the following properties is required for a valid hash function $h(k)$?
    I. A hash function must produce unique hash values for unique inputs (i.e., $h(k_1) \neq h(k_2)$ if $k_1 \neq k_2$ for all k).
    II. A hash function must produce the same hash value for the same inputs (i.e., $h(k_1) = h(k_2)$ if $k_1 = k_2$ for all k).
    III. A hash function must produce similar hash values for similar (but distinct) inputs.

    A) I only
    B) II only
    C) I and II only
    D) II and III only
    E) I, II, and III

44. You are given an open addressing hash table with capacity 31 that doubles whenever its load factor becomes ≥ 0.5. There are currently 12 positions in the table marked as deleted. After inserting an element into this hash table, you notice that the capacity of the hash table immediately doubled from 31 to 62. After this insertion and reallocation, how many elements in this hash table are marked as empty, deleted, and occupied (assuming that no other elements are added or removed)?
    A) Empty: 35, Deleted: 12, Occupied: 15
    B) Empty: 23, Deleted: 24, Occupied: 15
    C) Empty: 34, Deleted: 12, Occupied: 16
    D) Empty: 22, Deleted: 24, Occupied: 16
    E) None of the above

45. Which of the following statements is **FALSE**?
    A) Deleting items from a hash table that uses open addressing will always improve insertion speed
    B) If no resizing is done and no elements are deleted, an open addressing hash table's lookup performance will deteriorate as the number of elements inserted increases
    C) The performance of a hash table that uses separate chaining will be less sensitive to the load factor than the performance of a hash table that uses open addressing
    D) A poor hash function can directly worsen the performance of hash table operations, such as lookup, insertion, and removal
    E) None of the above (i.e., all of the above statements are true)

46. In 2022, the Boost library introduced the `boost::unordered_flat_map<>`, a fast alternative to `std::unordered_map<>` that uses open addressing rather than separate chaining. In the `boost::unordered_flat_map<>`, an underlying array is split into m groups of size 15, where m is always a power of two. A hash function determines which group a key falls into, and each key is inserted into the first available position of that group — if the group to insert into becomes full, future insertions that hash to that group will use quadratic probing to identify another unused group to support the insertion. A rough depiction of this layout is shown below:



    Which of the following could be potential explanations as to why the `boost::unordered_flat_map<>` is so performant?
    A) Because open addressing is used instead of separate chaining with linked lists, there is no indirection required to go from the bucket that an element hashes to to the actual memory location of that element (i.e., less dereferencing overhead)
    B) Because groups of elements are stored contiguously in memory, the Boost hash map is able to take advantage of faster data retrieval that occurs when elements are stored close together in memory (a process known as caching)
    C) By utilizing groups of buckets of size 15, the Boost hash map is able to support up to 15 elements that hash to the same group before a collision resolution technique needs to be used to identify a different group to insert into
    D) The Boost hash map is more lightweight and requires fewer memory allocations compared to a separate chaining approach that uses linked lists, as it does not need to store additional pointers to link the elements that hash to the same group
    E) All of the above

47. Consider the following snippet of code:

```
1  double get_student_gpa(const std::unordered_map<int32_t, double>& gpa_map,
2                         int32_t student_id, double gpa) {
3    auto gpa_it = gpa_map.find(student_id);
4    if (gpa_it != gpa_map.end()) {
5      return gpa_map[student_id];
6    } // if
7
8    return 0.0;
9  } // add_student_gpa()
10
11 int main() {
12   std::unordered_map<int32_t, double> gpa_map;
13
14   int32_t student_id;
15   double gpa;
16   while (std::cin >> student_id >> gpa) {
17     gpa_map[student_id] = gpa;
18   } // while
19
20   for (const auto& entry : gpa_map) {
21     std::cout << entry.first << " "
22               << get_student_gpa(gpa_map, entry.first, entry.second) << std::endl;
23   } // for
24 } // main()
```

Does this code compile? If not, which line causes an issue?

A) This code compiles without issue
B) This code does not compile due to an issue on line 3
C) This code does not compile due to an issue on line 5
D) This code does not compile due to an issue on line 17
E) This code does not compile due to an issue on line 22

48. Which of the following is supported by an `std::unordered_map<>`, but **NOT** an `std::unordered_set<>`?

A) `operator[]`
B) `operator==`
C) `operator<`
D) `.insert()`
E) `.find()`

49. A trusted colleague of yours has developed a method for hash collision resolution that they claim runs in **only Θ(1) time**. This method uses *separate chaining* and is almost identical to the method described in this chapter. The only difference is that, when a collision occurs, the given element is inserted as the **head** node of the linked list at a particular index in Θ(1) time, instead of as the tail node in Θ(m) time, where m is the size of the list at that index. Did your colleague actually develop a valid Θ(1) collision resolution technique that is guaranteed to work in all cases? Why or why not?

50. Consider the following empty hash table of size 10, which stores `std::string` keys.

The hash function used translates the first letter of each string to its distance from the character 'a'. The integer each letter hashes to is shown in the table below. Compression is done using the formula $C(n) = n \bmod M$, where M is the size of the table. No resizing is done.

a	**b**	**c**	**d**	**e**	**f**	**g**	**h**	**i**	**j**	**k**	**l**	**m**	**n**
0	1	2	3	4	5	6	7	8	9	10	11	12	13

o	**p**	**q**	**r**	**s**	**t**	**u**	**v**	**w**	**x**	**y**	**z**
14	15	16	17	18	19	20	21	22	23	24	25

For this question, you may choose whether you want this table to handle collisions using *linear probing* or *quadratic probing*. After deciding the collision resolution method, provide a sequence of *lowercase* strings that you can insert into this table so that a subsequent insertion of the string `"slime"` would cause it to end up at index 7 of the hash table. This is an open ended question, so there may be multiple answers!

51. You are given the following definition of the `DoubleHashTable` class:

```
enum class Status : uint8_t { Empty, Occupied, Deleted };

struct Bucket {
  Status status = Status::Empty;
  int32_t key;
};

class DoubleHashTable {
  std::vector<Bucket> hash_table;
  size_t num_elements = 0;
  static constexpr int32_t table_size = 13;
  static constexpr int32_t prime = 7;

public:
  DoubleHashTable() {
    hash_table.resize(table_size);
  } // DoubleHashTable()

  bool is_full() {
    return num_elements == table_size;
  } // is_full()

  size_t hash1(int32_t key) {
    return key;
  } // hash1()

  size_t hash2(int32_t key) {
    return prime - (key % prime);
  } // hash2()

  bool insert_key(int32_t key) {
    // TODO: Implement this function
  } // insert_key()
};
```

(a) This hash table uses **double hashing** to resolve collisions, where `hash1()` is the primary hash function and `hash2()` is the secondary hash function. Implement the `insert_key()` function (as shown by the TODO above), which inserts the provided key into the hash table and returns whether the key was successfully inserted. If the hash table is full, do not insert anything and return `false`. You may **NOT** use anything from the STL.

```
bool DoubleHashTable::insert_key(int32_t key);
```

(b) Now, suppose you ran the following snippet of code (with a working implementation of `insert_key()`):

```
int main() {
  int32_t arr[] = {11, 24, 25, 41, 45, 48};
  DoubleHashTable table;
  for (int32_t val : arr) {
    table.insert_key(val);
  } // for val
} // main()
```

What does the internal state of the hash table look like? Fill out the empty table below with the values in their correct positions.

0	1	2	3	4	5	6	7	8	9	10	11	12

52. You are currently waiting for the Northwood bus at Pierpont, which you know will arrive after a certain amount of time thanks to the M-Bus app. Since you do not want to wait in silence, you decide to take out a mobile device and listen to some music. You have many songs to choose from on your device, and you know that you have enough time to finish two tracks before the next bus arrives. However, since you want to maximize your listening time without having to stop a song before it finishes, you want to select two songs such that their combined runtime equals the exact wait time required before the bus's arrival.

Implement a function that takes in an integer wait time (in seconds) and a vector of music tracks (in seconds) and determines if there exist two tracks whose combined length equals the wait time. You may assume that the wait time is guaranteed to be exact to the second. One notable detail is that you do *not* want to listen to the same song twice, even if the track length is exactly half of the total wait time, so such tracks should not count as a solution for this problem.

Example 1: Given `tracks = [300, 240, 180, 200]` and a wait time of `540`, you would return `true` since `300+240=540`.

Example 2: Given `tracks = [300, 240, 180, 200]` and a wait time of `600`, you would return `false` since there are no two tracks that sum to this value. Note that the track of length `300` cannot be listened to twice.

```
bool identify_tracks(const std::vector<int32_t>& tracks, int32_t wait_time);
```

Your solution should take linear time on the length of the tracks vector in the average case.

53. Consider the following `Message` object, which represents a message that is sent in a chat server:

```
struct Message {
  std::string sender;
  std::string text;
};
```

The `sender` member stores the username of the person who sent a message as a lowercase string, and the `text` member stores the contents of the actual message itself. Given a vector of `Message` objects, implement a function that returns the user with the highest word count among all their messages. If there is a tie, return the user whose username is *largest* lexicographically. You may assume that every word in each message will be separated by a single space. *Hint: the `std::count()` method could be helpful here.*

For example, given a vector of messages that correspond to the following conversation in the EECS 281 Discord server:

8/19/2022 5:22:56 PM **slime:** did anyone new get 281
8/19/2022 5:55:16 PM **slime:** @everyone
8/19/2022 5:55:29 PM **iteemhe:** emmm why
8/19/2022 5:55:34 PM **doubledelete:** i didn't expect you to do it
8/19/2022 5:55:35 PM **iteemhe:** why allow everyone
8/19/2022 5:55:40 PM **toafu:** Oh he actually did it
8/19/2022 5:55:50 PM **denalz:** slime moment
8/19/2022 5:56:06 PM **denalz:** that was funny tho
8/19/2022 5:56:12 PM **denalz:** im not bothered
8/19/2022 5:56:28 PM **deebs:** unacceptable behavior
8/19/2022 5:58:16 PM **amadeus:** We have never had this many concurrent users online
8/19/2022 6:00:25 PM **slime:** server's active today
8/19/2022 6:01:46 PM **kryptof:** I wonder why

there are three users with a tie for the highest word count (9): "amadeus", "denalz", and "slime". In this case, "slime" is returned because it is the largest username lexicographically (since 's' comes after 'a' and 'd').

```
std::string user_with_highest_word_count(const std::vector<Message>& messages);
```

You solution should take $O(nd)$ time in the average case, where n is the number of messages and d is the maximum length of a message.

54. Two pairs (a,b) and (c,d) are symmetric if $b = c$ and $a = d$. Suppose you are given a vector of pairs, and you want to find all symmetric pairs on the vector. The first element of all pairs is distinct. For instance, if you are given the following vector:

```
vec = [(14, 23), (11, 2), (52, 83), (49, 38), (38, 49), (2, 11)]
```

you would return `[(11, 2), (2, 11)]` and `[(49, 38), (38, 49)]` (in any order).

```
std::vector<std::vector<std::pair<int32_t, int32_t>>> find_symmetric_pairs(
  const std::vector<std::pair<int32_t, int32_t>>& vec);
```

Your solution should take linear time on the length of the vector in the average case.

55. Given a vector of n elements where elements may be repeated, implement a function that returns the minimum number of elements that need to be deleted from the vector so that all elements in the vector are equal. For instance, if you are given the following vector:

```
vec = [3, 6, 8, 6, 2, 7, 6, 3, 1, 3, 6]
```

you would return `7`, as this is the minimum number of deletions required to obtain a vector where all the elements are the same (in this case, you would get a vector with all 6's).

```
int32_t min_deletions_for_all_equal(const std::vector<int32_t>& vec);
```

Your solution should take linear time on the length of the vector in the average case.

56. You are given an array with repeated elements. Implement a function that returns the maximum distance between any two occurrences of a repeated element. For example, given the following input:

```
vec = [1, 2, 3, 2, 2, 1, 3, 3, 2]
```

you would return `7`, since the maximum distance between any two repeated elements in the distance between the 2 at index 1 and the 2 at index 8, or 8 - 1 = 7.

```
int32_t max_repeated_distance(const std::vector<int32_t>& vec);
```

Your solution should take linear time on the length of the vector in the average case.

57. You are given an array and a target value k. Implement a function that returns the number of subarrays in the input array that have a sum of k. A subarray is a contiguous non-empty sequence of values in an array. For example, given the following input and a target of $k = -1$:

```
vec = [2, -3, 4, -2, 1, -1]
```

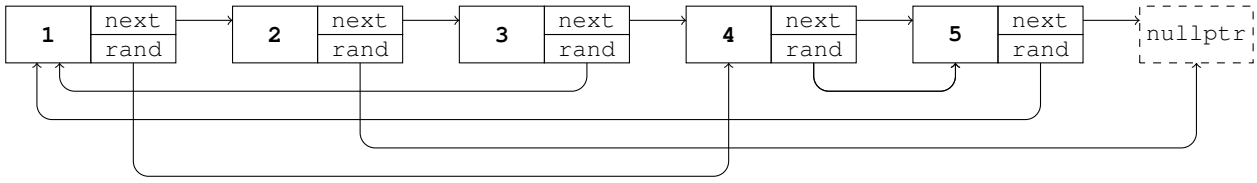
you would return `5`, since there are five subarrays that sum up to -1:

```
[2, -3, 4, -2, 1, -1]   [2, -3, 4, -2, 1, -1]   [2, -3, 4, -2, 1, -1]   [2, -3, 4, -2, 1, -1]   [2, -3, 4, -2, 1, -1]
```

```
int32_t num_subarrays_with_sum_k(const std::vector<int32_t>& vec, int32_t k);
```

Your solution should take linear time on the length of the vector in the average case.

58. You are given a singly-linked list where each node contains an additional random pointer that could point to any node in the list (or `nullptr`). An example of one possible list is shown below:



Each node is represented as follows:

```
struct Node {
  int32_t val;
  Node* next;
  Node* random;
  Node(int32_t val_in) : val{val_in}, next{nullptr}, random{nullptr} {}
};
```

Implement a function that returns a deep copy of the list.

```
Node* copy_random_list(Node* head);
```

Your solution should take linear time on the length of the list in the average case.

59. You are in charge of a transit system for a large city, and you are given the following `TransitManager` class, which handles data on ridership. Implement the following methods for the `TransitManager` class:
    - `void check_in(int64_t rider_id, const std::string& station_name, int64_t timestamp);`
      - This method is called when the rider with ID `rider_id` checks into the station `station_name` at timestamp `timestamp`.
      - Once a rider checks into a station, they cannot check in again until after they check out. You may assume that this error condition will never happen for this problem.
    - `void check_out(int64_t rider_id, const std::string& station_name, int64_t timestamp);`
      - This method is called when the rider with ID `rider_id` checks out from station `station_name` at timestamp `timestamp`.
    - `double get_avg_time(const std::string& start_station, const std::string& end_station);`
      - This method returns the average time it takes to travel from `start_station` to `end_station`. This value is calculated from all previous riders who directly traveled between the two stations (where start is the check in station and end is the check out station). For example, if rider 1 traveled from start to end in 5 minutes, and rider 2 traveled from start to end in 7 minutes, the average travel time between start and end would be 6 minutes (assuming no other travelers have the same start and end stations).
    - `int64_t get_num_rides(int64_t rider_id);`
      - This method returns the number of rides initiated with the rider with ID `rider_id`. The number of rides is increased when a rider checks in (so if a rider checks into their 5th ride but has not checked out yet, this function would still return 5).
    - `int64_t get_num_route(const std::string& start_station, const std::string& end_station);`
      - This method returns the number of riders who have traveled the route between `start_station` and `end_station` (i.e., the count is only increased for the route $A \to B$ if a rider checks in at the start station A and checks out at the end station B).

    You may assume that calls to `check_in()` and `check_out()` are consistent, and that timestamps are provided in chronological order.

```
1  class TransitManager {
2  private:
3    // TODO: Add any data structures here!
4  public:
5    void check_in(int64_t rider_id, const std::string& station_name, int64_t timestamp) {
6      // TODO: Implement this
7    } // check_in()
8
9    void check_out(int64_t rider_id, const std::string& station_name, int64_t timestamp) {
10     // TODO: Implement this
11   } // check_out()
12
13   double get_avg_time(const std::string& start_station, const std::string& end_station) {
14     // TODO: Implement this
15   } // get_avg_time()
16
17   int64_t get_num_rides(int64_t rider_id) {
18     // TODO: Implement this
19   } // get_num_rides()
20
21   int64_t get_num_route(const std::string& start_station, const std::string& end_station) {
22     // TODO: Implement this
23   } // get_num_route()
24 };
```

60. Your local bank is designing a `TransactionManager` class that can be used to automate all transactions. This class will need to support four operations, which you are tasked to implement:
    - **bool** `open(const std::string& account_name);`
      - Opens an account with the account name `account_name` and return whether this was successful. If an account already exists with the requested name, return `false`.
    - **bool** `deposit(const std::string& account_name, double amount);`
      - Deposits the amount `amount` into the account `account_name`, and returns a Boolean specifying whether the transaction was successful. If `account_name` does not exist, a deposit should not happen and this method should return `false`.
    - **bool** `withdraw(const std::string& account_name, double amount);`
      - Withdraws the amount `amount` from the account `account_name`, and returns a Boolean specifying whether the transaction was successful. If `account_name` does not exist, or if the account is trying to withdraw more money than they have, a withdrawal should not happen and this method should return `false`.
    - **bool** `transfer(const std::string& src_account, const std::string& dest_account, double amount);`
      - Transfers the amount `amount` from the account of `src_account` into the account of `dest_account`, and returns whether the transaction was successful. If either account does not exist, or if there is not enough money in `src_account`, a transfer should not happen and this method should return `false`.

    An outline of this class is provided below:

```
class TransactionManager {
private:
  // TODO: Add any data structures here!
public:
  bool open(const std::string& account_name) {
    // TODO: Implement this
  } // open()

  bool deposit(const std::string& account_name, double amount) {
    // TODO: Implement this
  } // deposit()

  bool withdraw(const std::string& account_name, double amount) {
    // TODO: Implement this
  } // withdraw()

  bool transfer(const std::string& src_account, const std::string& dest_account, double amount) {
    // TODO: Implement this
  } // transfer()
};
```

61. In this problem, you will be designing a `Database` class that represents a simplified database. You are given n tables that you need to create in this database. Each table is given a name and a list of columns, which is provided to you in the following `TableInfo` object:

```
struct TableInfo {
  std::string table_name;
  std::vector<std::string> column_names;
};
```

    The list of table and column names are guaranteed to be unique. For example, given a `TableInfo` where `table_name` is `"Students"` and `column_names` is `["Name", "Year", "Major", "GPA"]`, you would create a table named `"Students"` in your database with the columns of `"Name"`, `"Year"`, `"Major"`, and `"GPA"`. For simplicity, you may assume that all values are stored in the database as strings, even if they are numerical values (such as GPA in the example above).

    Your `Database` class will need to support the following operations, which you are tasked to implement:
    - **bool** `create(const TableInfo& table_info);`
      - Creates a table with the settings described in the table info, and return whether this was successful. If the table name exists already, return `false`.
    - **int64_t** `insert(const std::string& table_name, const std::vector<std::string>& row_data);`
      - Inserts the contents of `row_data` into the table with name `table_name`, and assigns this row with an ID that is returned. You are guaranteed that the table `table_name` will exist, and the contents of `row_data` will correspond to the columns that the table was initialized with. For example, with our `"Students"` table, you could be given the following `row_data` input:

        `row_data = ["Alice", "3", "Computer Science", "3.9"]`

        In this case, you would insert a row into the table with these values, which correspond to `"Name"`, `"Year"`, `"Major"`, and `"GPA"`, respectively.
      - Each row is assigned an auto-incremented ID value that starts from 1. For instance, the first row inserted is assigned an ID of 1, the second row inserted is assigned an ID of 2, and so on. Deleted rows do not impact the next assignable ID — if the row with ID 2 is deleted, and another row is inserted, that new row would get assigned an ID of 3 instead of 2.
    - **bool** `remove(const std::string& table_name, int64_t row_id);`
      - Deletes the row in the table `table_name` with the ID `row_id`, if it exists, and returns whether a deletion was performed. You are guaranteed that the table `table_name` will exist.

- `std::optional<std::string> select(const std::string& table_name, int64_t row_id, const std::string& column_name);`
  - Returns the value of the cell in the table `table_name` associated with the row with ID `row_id` and column name `column_name`. If no such value exists, return `std::nullopt`. For example, if `select()` is called with a table name of `"Students"`, a row ID of `1`, and a column name of `"GPA"`, you would return `"3.9"`.

An outline of this class is provided below:

```
1  class Database {
2  private:
3    // TODO: Add any data structures here!
4  public:
5    bool create(const TableInfo& table_info) {
6      // TODO: Implement this
7    } // create()
8
9    int64_t insert(const std::string& table_name, const std::vector<std::string>& row_data) {
10     // TODO: Implement this
11   } // insert()
12
13   bool remove(const std::string& table_name, int64_t row_id) {
14     // TODO: Implement this
15   } // remove()
16
17   std::optional<std::string> select(const std::string& table_name, int64_t row_id,
18                                     const std::string& column_name) {
19     // TODO: Implement this
20   } // select()
21 };
```

Chapter 17 Exercise Solutions

1. **The correct answer is (D).** Hash tables provide constant time access to a value given its key, so both I and III can be performed in $\Theta(1)$ time. However, they are not sorted, so finding the cheapest item in the store would not be doable in $\Theta(1)$ time.

2. **The correct answer is (A).** Only statement I is true: `operator[]` inserts a key if it does not exist. Statement II is false because the worst-case is $\Theta(n)$. Statement III is false because a requirement of hash functions is that they must always hash the same key to the same hash value, which is not guaranteed if you use a random number generator to determine your hash value.

3. **The correct answer is (B).** A good hash function evenly distributes keys in a hash table; there is no requirement to group similar elements together. In fact, you would likely want the keys in a hash table to be evenly spread out instead of clustered together, which may be a hindrance to performance for open addressing.

4. **The correct answer is (C).** Using $= 17$, $t = 99$, $key = 83$, and $M = 41$, we get the following:
$$h(key) = \lfloor \tfrac{83-17}{99-17} \times 41 \rfloor = \lfloor \tfrac{66}{82} \times 41 \rfloor = \lfloor 66 \times \tfrac{41}{82} \rfloor = \lfloor 66 \times \tfrac{1}{2} \rfloor = 33$$

5. **The correct answer is (D).** The compression method provided works best when the keys are randomly distributed. The test scores are not evenly distributed; since each question is worth 5 points, scores can only take on the values 0, 5, 10, 15, …, 90, 95, 100. The number of hours of sleep and number of credits taken by each student are also not evenly distributed and take on a rather restricted range of values. Only the collection of student IDS at the university is random as the ID of any particular student in EECS 281 could be any 8-digit number.

6. **The correct answer is (B).** The `insert()` method does not do anything if the key already exists in the unordered map, as unordered maps cannot have duplicate keys. On line 3, the key `"Paoletti"` is initialized with the value `"Darden"`. Lines 5 and 7 do not do anything since the key `"Paoletti"` already exists in the table. Thus, the value of `"Darden"` remains unchanged and is printed out on line 8.

7. **The correct answer is (A).** Even though the key `"Angstadt"` is initially created with the value `"Darden"` on line 4, it is updated on line 6 to a value of `"Paoletti"`. Thus, `"Paoletti"` is printed out on line 11.

8. **The correct answer is (B).** Prior to line 10, there are three keys that exist in the hash table: `"Paoletti"` (created on line 3), `"Angstadt"` (created on line 4), and `"Darden"` (created on line 9 – recall that `operator[]` automatically inserts a key if it does not already exist). After `"Paoletti"` is removed from the hash table on line 10, there are only two keys remaining, so line 12 prints out `2`.

9. **The correct answer is (C).** The type of an element in an unordered map is a key-value pair.

10. **The correct answer is (D).** The expression `posts["memes"]` is a reference for the value associated with the key `"memes"`, or the vector of post IDs associated with this key. As a result, you can treat `posts["memes"]` as the type of the value (a vector) and simply push back the new ID.

11. **The correct answer is (E).** An unordered map is, by definition, unordered. Thus, you cannot determine which key was encountered first while iterating over the map.

12. **The correct answer is (C).** `foods["cabbage"]` is the value type of the unordered map, which is a `std::pair<std::string, int32_t>`. Thus, to get the integer value of this pair (which is `1` for the key `"cabbage"`), you would need to reference the `.second` member of the pair.

13. **The correct answer is (C).** To solve this problem, first iterate over the first array and store all of its elements in an `std::unordered_set<>`. Then, iterate over the second array and check if the difference between each element in this array and the target sum is in the unordered set. If it is, you have found a pair that sums to the target value. Since you are simply iterating over the arrays of length n, and inserting into an unordered set takes constant time on average, the average-case time complexity of the overall solution is $\Theta(n)$.

14. **The correct answer is (A).** Open addressing is a class of collision resolution methods in which open or empty hash table positions are used to resolve collisions. This is not the case for separate chaining, which uses closed addressing to resolve collisions (e.g., if a key hashes to position x, you would know that this key can be found at position x, even if it might not be alone at that location).

15. **The correct answer is (D).** Option (A) is true because load factor can be greater than 1 if separate chaining is used. Similarly, option (B) is true because open addressing methods need to search for open positions in the case of a collision, which are harder to find as the table fills up. Option (C) is true because the cost of search and erase in the average case is $\Theta(\alpha)$ for a hash table that uses separate chaining, as the average linked list in the table would have a length of $\alpha = N/M$ if the N keys are evenly distributed across all M indices. Option (D) is false because in linear probing, collisions often result in keys being placed close to their hashed location (since we always incrementing the index by one when checking for the next available position), while in double hashing, keys that result in collisions are more evenly spread throughout the hash table (since we use a separate hash function to determine our increment).

16. **The correct answer is (B).** Options (A) and (C) are related to the concept of quadratic residuals, which is a disadvantage of quadratic probing: depending on the table size, certain indices of the table may never be visited during the collision resolution process (see section 17.4). Option (D) is also a disadvantage this is characteristic of open addressing techniques. Option (B) is not a disadvantage, since open positions are not searched for consecutively (instead, quadratic probing uses increments of 1, 4, 9, …, which is less susceptible to primary clustering). Primary clustering is mainly a concern of linear probing, where there is a tendency for many keys to end up next to each other.

17. **The correct answer is (B).** If a hash table of size 100 has 40 empty positions and 25 deleted positions, then there must be 100 - 40 - 25 = 35 positions that are occupied by existing values. Thus, the load factor is 35/100 = 0.35.

18. **The correct answer is (B).** Your insertion must have caused the hash table to reach a load factor of 0.6, as it doubled and rehashed after the insertion. Since the original size of the hash table was 75, your insertion must have added the 0.6 * 75 = 45th value into the hash table.

19. **The correct answer is (B).** Here is how the elements are placed into this hash table:
    - 10 is placed at index (3(10) + 7) mod 10 = 37 mod 10 = 7
    - 8 is placed at index (3(8) + 7) mod 10 = 31 mod 10 = 1
    - 18 is placed at index (3(18) + 7) mod 10 = 61 mod 10 = 1 (collision!)
      - Linear probing sends 18 to the next open index after index 1, or index 2
    - 17 is placed at index (3(17) + 7) mod 10 = 58 mod 10 = 8
    - 4 is placed at index (3(4) + 7) mod 10 = 19 mod 10 = 9
    - 20 is placed at index (3(20) + 7) mod 10 = 67 mod 10 = 7 (collision!)
      - Linear probing sends 20 to the next open index after index 7
      - Indices 8 and 9 are occupied, so the search is looped to the beginning, and 20 is placed at index 0, which is open
    - 6 is placed at (3(6) + 7) mod 10 = 25 mod 10 = 5
    - 3 is placed at (3(3) + 7) mod 10 = 16 mod 10 = 6
    - 16 is placed at index (3(16) + 7) mod 10 = 55 mod 10 = 5 (collision!)
      - Linear probing sends 16 to the next open index after index 5
      - This forces 16 to index 3 (since the search loops around to the beginning)

    This is what the final table looks like:

0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
20	8	18	16		6	3	10	17	4

20. **The correct answer is (A).** In this case, the odd numbers hash to index 2 and the even numbers hash to index 7. Thus, 1 hashes to 2, 2 hashes to 7, 3 hashes to 3, 4 hashes to 8, 5 hashes to 4, 6 hashes to 9, 7 hashes to 5, 8 hashes to 0, and 9 hashes to 6.

    This is what the final table looks like:

0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
8		1	3	5	7	9	2	4	6

21. **The correct answer is (B).** Here is how the elements are placed into this hash table:
    - 24 is placed at index 24 mod 7 = 3
    - 11 is placed at index 11 mod 7 = 4
    - 17 is placed at index 17 mod 7 = 3 (collision!)
      - Quadratic probing sends 17 to index $(3 + 1^2)$ mod 7 = 4 (collision!)
      - Quadratic probing sends 17 to index $(3 + 2^2)$ mod 7 = 7 mod 7 = 0
    - 21 is placed at index 21 mod 7 = 0 (collision!)
      - Quadratic probing sends 21 to index $(0 + 1^2)$ mod 7 = 1
    - 10 is placed at index 10 mod 7 = 3 (collision!)
      - Quadratic probing sends 10 to index $(3 + 1^2)$ mod 7 = 4 (collision!)
      - Quadratic probing sends 10 to index $(3 + 2^2)$ mod 7 = 7 mod 7 = 0 (collision!)
      - Quadratic probing sends 10 to index $(3 + 3^2)$ mod 7 = 12 mod 7 = 5

- 4 is placed at index 4 mod 7 = 4 (collision!)
  - Quadratic probing sends 4 to index $(4 + 1^2)$ mod 7 = 5 (collision!)
  - Quadratic probing sends 4 to index $(4 + 2^2)$ mod 7 = 8 mod 7 = 1 (collision!)
  - Quadratic probing sends 4 to index $(4 + 3^2)$ mod 7 = 13 mod 7 = 6

This is what the final table looks like:

0	**1**	**2**	**3**	**4**	**5**	**6**
17	21		24	11	10	4

22. **The correct answer is (C).** Here is how the elements are placed into this hash table:
    - 9 is placed at index (2(9) + 3) mod 13 = 21 mod 13 = 8
    - 22 is placed at index (2(22) + 3) mod 13 = 47 mod 13 = 8 (collision!)
      - Quadratic probing sends 22 to index $(8 + 1^2)$ mod 13 = 9
    - 7 is placed at index (2(7) + 3) mod 13 = 17 mod 13 = 4
    - 10 is placed at index (2(10) + 3) mod 13 = 10 mod 13 = 10
    - 3 is placed at index (2(3) + 3) mod 13 = 9 mod 13 = 9 (collision!)
      - Quadratic probing sends 3 to index $(9 + 1^2)$ mod 13 = 10 (collision!)
      - Quadratic probing sends 3 to index $(9 + 2^2)$ mod 13 = 13 mod 13 = 0
    - 1 is placed at index (2(1) + 3) mod 13 = 6 mod 13 = 13
    - 20 is placed at index (2(20) + 3) mod 13 = 43 mod 13 = 4 (collision!)
      - Quadratic probing sends 20 to index $(4 + 1^2)$ mod 13 = 5
    - 13 is placed at index (2(13) + 3) mod 13 = 29 mod 13 = 3
    - 18 is placed at index (2(18) + 3) mod 13 = 39 mod 13 = 0 (collision!)
      - Quadratic probing sends 18 to index $(0 + 1^2)$ mod 13 = 1
    - 5 is placed at index (2(5) + 3) mod 13 = 13 mod 13 = 0 (collision!)
      - Quadratic probing sends 5 to index $(0 + 1^2)$ mod 13 = 1 (collision!)
      - Quadratic probing sends 5 to index $(0 + 2^2)$ mod 13 = 4 (collision!)
      - Quadratic probing sends 5 to index $(0 + 3^2)$ mod 13 = 9 (collision!)
      - Quadratic probing sends 5 to index $(0 + 4^2)$ mod 13 = 3 (collision!)
      - Quadratic probing sends 5 to index $(0 + 5^2)$ mod 13 = 12
    - 0 is placed at index (2(0) + 3) mod 13 = 3 mod 13 = 3 (collision!)
      - Quadratic probing sends 0 to index $(3 + 1^2)$ mod 13 = 4 (collision!)
      - Quadratic probing sends 0 to index $(3 + 2^2)$ mod 13 = 7
    - 11 is placed at index (2(11) + 3) mod 13 = 25 mod 13 = 12 (collision!)
      - Quadratic probing sends 11 to index $(12 + 1^2)$ mod 13 = 0 (collision!)
      - Quadratic probing sends 11 to index $(12 + 2^2)$ mod 13 = 3 (collision!)
      - Quadratic probing sends 11 to index $(12 + 3^2)$ mod 13 = 8 (collision!)
      - Quadratic probing sends 11 to index $(12 + 4^2)$ mod 13 = 2

    This is what the final table looks like:

0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**
3	18	11	13	7	20	1	0	9	22	10		5

23. **The correct answer is (C).** Note that all the elements are in their proper positions except for 82 and 53. For 82 to end up at index 5, all indices from 2 to 4 must have been filled when 82 was inserted. Thus, 82 must have been inserted after 62, 43, and 24, eliminating choice (A). For 53 to end up at index 7, all indices from 3 to 6 must have been filled when 53 was inserted. Thus, 53 must have been inserted after 43, 24, 82, and 76. This eliminates choices (B) and (D). Only choice (C) remains, and it does in fact work.

24. **The correct answer is (C).** For the strings "paoletti" and "darden" to collide, 15 mod M must be the same as 3 mod M. This is true for all of the above except for $M = 9$ (15 mod 9 = 6 while 3 mod 9 = 3).

25. **The correct answer is (A).** Here is how the elements are placed into this hash table:
    - 6 is placed at index (2(6) + 1) mod 7 = 13 mod 7 = 6
    - 10 is placed at index (2(10) + 1) mod 7 = 21 mod 7 = 0
    - 4 is placed at index (2(4) + 1) mod 7 = 9 mod 7 = 2
    - 13 is placed at index (2(13) + 1) mod 7 = 27 mod 7 = 6 (collision!)
      - The secondary hash function produces a value of (5 - (27 mod 5)) = 3, so we will probe open positions a distance of 3 apart
      - Double hashing sends 13 to index (6 + 1(3)) mod 7 = 9 mod 7 = 2 (collision!)
      - Double hashing sends 13 to index (6 + 2(3)) mod 7 = 12 mod 7 = 5
    - 11 is placed at index (2(11) + 1) mod 7 = 23 mod 7 = 2 (collision!)
      - The secondary hash function produces a value of (5 - (23 mod 5)) = 2, so we will probe open positions a distance of 2 apart
      - Double hashing sends 11 to index (2 + 1(2)) mod 7 = 4 mod 7 = 4 (collision!)
    - 12 is placed at index (2(12) + 1) mod 7 = 25 mod 7 = 4 (collision!)
      - The secondary hash function produces a value of (5 - (25 mod 5)) = 5, so we will probe open positions a distance of 5 apart
      - Double hashing sends 12 to index (4 + 1(5)) mod 7 = 9 mod 7 = 2 (collision!)
      - Double hashing sends 12 to index (4 + 2(5)) mod 7 = 14 mod 7 = 0 (collision!)
      - Double hashing sends 12 to index (4 + 3(5)) mod 7 = 19 mod 7 = 5 (collision!)
      - Double hashing sends 12 to index (4 + 4(5)) mod 7 = 24 mod 7 = 3

This is what the final table looks like:

0	1	2	3	4	5	6
10		4	12	11	13	6

26. **The correct answer is (B).** Here is how the elements are placed into this hash table:
    - "sun" is placed at index 18 mod 10 = 8
    - "mercury" is placed at index 12 mod 10 = 2
    - "venus" is placed at index 21 mod 10 = 1
    - "earth" is placed at index 4 mod 10 = 4
    - "mars" is placed at index 12 mod 10 = 2 (collision!)
      - Double hashing sends "mars" to index 2 + 1(7 - 12 mod 7) mod 10 = 2 + 2 = 4 (collision!)
      - Double hashing sends "mars" to index 2 + 2(7 - 12 mod 7) mod 10 = 2 + 4 = 6
    - "jupiter" is placed at index 9 mod 10 = 9
    - "saturn" is placed at index 18 mod 10 = 8 (collision!)
      - Double hashing sends "saturn" to index 8 + 1(7 - 18 mod 7) mod 10 = 11 mod 10 = 1 (collision!)
      - Double hashing sends "saturn" to index 8 + 2(7 - 18 mod 7) mod 10 = 14 mod 10 = 4 (collision!)
      - Double hashing sends "saturn" to index 8 + 3(7 - 18 mod 7) mod 10 = 17 mod 10 = 7
    - "uranus" is placed at index 20 mod 10 = 0
    - "neptune" is placed at index 13 mod 10 = 3

    This is what the final table looks like:

0	1	2	3	4	5	6	7	8	9
"uranus"	"venus"	"mercury"	"neptune"	"earth"		"mars"	"saturn"	"sun"	"jupiter"

27. **The correct answer is (E).** All four strings hash to the same location in a hash table of size $M = 13$. Thus, we know the following:
    - "apples" is hashed to index 0
    - "almonds" is hashed to index 0 + 1(q - 0 mod q) mod 13 = 1q mod 13
    - "avocados" is hashed to index 0 + 2(q - 0 mod q) mod 13 = 2q mod 13
    - "asparagus" is hashed to index 0 + 3(q - 0 mod q) mod 13 = 3q mod 13

    Since "asparagus" ended up at index 7, we know that 3q has a remainder of 7 when divided by 13 (e.g., 3q mod 13 = 7). This is only true in the case of $q = 11$, since 33 mod 13 = 7.

28. **The correct answer is (D).** We can solve this problem by using the same process as in the previous question, but solving for M instead of q:
    - "apples" is hashed to index 0
    - "almonds" is hashed to index 0 + 1(7 - 0 mod 7) mod M = 7 mod M
    - "avocados" is hashed to index 0 + 2(7 - 0 mod 7) mod M = 14 mod M
    - "asparagus" is hashed to index 0 + 3(7 - 0 mod 7) mod M = 21 mod M

    Since "asparagus" ended up at index 4, we know that 21 mod M is equal to 4. This works for $M = 17$.

29. **The correct answer is (C).** Only statements I and III are true. Statement II is false because, unlike separate chaining, you cannot have more than one element at each index of the hash table if open addressing is used.

30. **The correct answer is (A).** Direct addressing is the name for this addressing method, where the key is directly mapped to a position in the hash table without any hashing.

31. **The correct answer is (B).** Only statement III compiles. Statement I does not work because `operator[]` requires the value type to be default constructible (since it needs to create an object of the value type), and `Thing` does not have a default constructor. Statement II does not work because a pair needs to be inserted into an unordered map as a single entity, and not just its constructor arguments (e.g., `thing_map.insert({183, my_thing})` or `thing_map.insert(std::make_pair(183, my_thing))` would work).

32. **The correct answer is (C).** To support efficient operations, hash tables do not store their elements in sorted order, so a different data structure will be preferable if you want to print a given collection of elements in sorted order.

33. **The correct answer is (A).** With separate chaining, each element will always end up at the bucket located at the position it hashes to, so no deleted marked is necessary. The special deleted marker is only needed for open addressing since the position of elements may be different than their hashed positions due to the collisions that are encountered, and we need a way to find these elements even if the values they collided with upon insertion have been removed.

34. **The correct answer is (C).** Hashes 1 and 4 are non-viable since they use a random number generator, which prevents the same string from being hashed to the same hash value every time (a requirement for hash functions). Between hashes 2 and 3, hash 3 is better since the hash value also depends on a character's positioning, which reduces the likelihood of collisions (e.g., for hash 2, "act" and "cat" would have the same hash value, but would not for hash 3).

35. **The correct answer is (E).** The indices probed are 5, (5 + 1) mod 10 = 6, (5 + 4) mod 10 = 9, (5 + 9) mod 10 = 4, and (5 + 16) mod 10 = 1 before we can conclude that K is not in the table. Note that index 1 needs to be probed because we need to confirm that it is empty before we can make our conclusion.

36. **The correct answer is (D).** Options (A) and (E) require knowledge of the ordering of elements, which is not well-suited for the intended use case of a hash table. Option (B) needs to keep track of a priority value for each value, which is not ideal either (a priority queue would be better for this). Hash tables are best designed for data that require fast lookups, of which options (C) and (D) fit the bill. However, option (C) involves keys that fall into a small universe of keys from 1 to 31, so a hash table is not necessary (we could just use an array of size 31 and index into it based on the day of the month). Option (D) is the best use case for a hash table, since it involves key-value lookups involving a string that cannot easily take advantage of direct addressing.

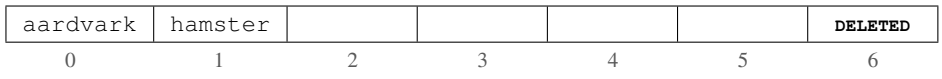
37. **The correct answer is (C).** Options (B) and (D) would only place elements at even number indices, leaving half the hash table with no elements. Option (E) would only place elements at indices 0 and 5, leaving eight other positions empty. This leaves us with (A) and (C). However, if you write out the indices for the first few keys using both of these hash functions, you would notice that the value of a perfect square can only end in 0, 1, 4, 5, 6, and 9 (e.g., **1**, **4**, **9**, 1**6**, 2**5**, 3**6**, 4**9**, 6**4**, 8**1**, 10**0**, 12**1**, 14**4**, 16**9**, 19**6**, 22**5**, 25**6**, 28**9**, 32**4**, 36**1**, 40**0**, 44**1**, etc.). Because of this, the value of $3i^2$ can only end with the digit 0, 3, 2, 5, 8, or 7, which would always leave indices 1, 4, 6, and 9 empty. On the other hand, a cube number can end with any digit, a pattern that repeats based on the final digit of the number that is being cubed (e.g., **1**, **8**, 2**7**, 6**4**, 12**5**, 21**6**, 34**3**, 51**2**, 72**9**, 100**0**, etc.). This also applies to multiples of 9, which can also end with any digit: **9**, 1**8**, 2**7**, 3**6**, 4**5**, 5**4**, 6**3**, 7**2**, 8**1**, 9**0**, etc.). Because of this, $9i^3$ would evenly distribute the numbers from 0 to 2019 among all 10 buckets of the hash table.

38. **The correct answer is (D).** Here is how the elements are placed into this hash table:
    - 7 is placed at index $7^2 \bmod 7 = 0$
    - 6 is placed at index $6^2 \bmod 7 = 1$
    - 5 is placed at index $5^2 \bmod 7 = 4$
    - 4 is placed at index $4^2 \bmod 7 = 2$
    - 3 is placed at index $3^2 \bmod 7 = 2$ (collision!)
        - The secondary hash function produces a value of $3^3 = 27$, so we will probe open positions a distance of 27 apart
        - Double hashing sends 3 to index $(2 + 1(27)) \bmod 7 = 29 \bmod 7 = 1$ (collision!)
        - Double hashing sends 3 to index $(2 + 2(27)) \bmod 7 = 56 \bmod 7 = 0$ (collision!)
        - Double hashing sends 3 to index $(2 + 3(27)) \bmod 7 = 83 \bmod 7 = 6$

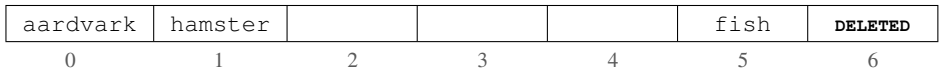
    This is what the final table looks like:

0	**1**	**2**	**3**	**4**	**5**	**6**
7	6	4		5		3

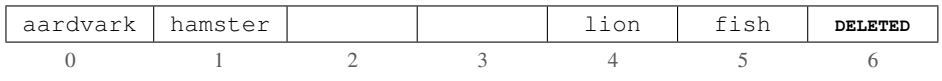
39. **The correct answer is (D).** Following the given operations, this is what happens to the hash table. First, we delete `"giraffe"` and mark it with a deleted flag.



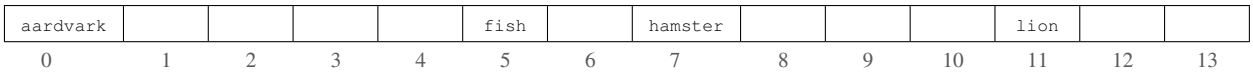
Then, we insert `"fish"`, which hashes to index 5.



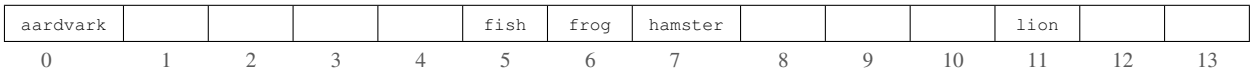
Then, we insert `"lion"`, which hashes to index 11 mod 7 = 4.



The number of elements in our table is now 4, which causes the load factor to exceed 0.5. As a result, the table doubles in size and rehashes all the elements to their correct positions in the new table. Deleted markers are not carried over to the new table.



Then, we insert `"frog"`, which hashes to index 5. There is a collision, so `"frog"` is sent to index 6.



Then, we insert `"ferret"`, which collides at index 5. Using linear probing, `"ferret"` is sent to index 8.

Then, we delete `"hamster"`, so we mark it with a deleted flag.

To insert `"fox"`, we would need to probe indices 5, 6, 7, 8, and 9 before you can safely insert it into the table.

40. **The correct answer is (D).** `"fox"` would be placed at the first index probed that is not occupied, which is index 7. Notice that deleted elements do not actually hold any value, so `"fox"` can still be inserted there.

41. **The correct answer is (C).** The string `"bear"` hashes to index 1, so for `"buffalo"` to be printed out first, it must have been sent to index 0 after collision resolution. All four strings collide on index 1, so the index each string gets sent to is:
    - `"badger"`: 1 + 1(7 - (1 mod 7)) mod M = 1 + 1(6) mod M = 7 mod M
    - `"bat"`: 1 + 2(7 - (1 mod 7)) mod M = 1 + 2(6) mod M = 13 mod M
    - `"buffalo"`: 1 + 3(7 - (1 mod 7)) mod M = 1 + 3(6) mod M = 19 mod M

    For `"buffalo"` to be sent to index 0, M can only be 19 from the provided options.

42. **The correct answer is (D).** Double hashing uses a secondary hash function to determine the collision probing interval, so it would be affected the most if the hash functions you use have to take linear time.

43. **The correct answer is (B).** Only II is a requirement of hash functions. Hash functions do not need to be unique for each input (which is resolved using collision resolution). There is also no need for hash functions to produce similar hash values for similar inputs.

44. **The correct answer is (E).** Deleted markers are not copied over during reallocation, so the number of deleted elements is 0. The number of occupied positions would be 16 (since the 16th element would cause the load factor to exceed 0.5 if the original hash table size was 31), and the number of empty positions would be 62 - 16 = 46.

45. **The correct answer is (A).** Deleting items from a hash table that uses open addressing does not guarantee improved insertion speed. This is because we still have to check if an element we want to insert exists, which involves probing deleted positions in the hash table as if an actual value existed at that position.

46. **The correct answer is (E).** All of the options are true and could be valid explanations.

47. **The correct answer is (C).** `operator[]` cannot be used on a `const` unordered map (since a `const` implementation of this operator is not defined), so this code does not compile. Even though this use case is valid (since we check if the key exists first), the compiler still checks for this to safeguard against a potential runtime error. One potential fix for this error is to return `gpa_it->second` on line 5, since the iterator returned by `.find()` points to the key-value pair containing the element, if found.

48. **The correct answer is (A).** `operator[]` exists in an `std::unordered_map<>`, but not an `std::unordered_set<>`, since unordered sets do not associate a value with each key. `operator==` is defined for both containers, `operator<` is defined for neither, and `.insert()` and `.find()` are defined for both.

49. No, you would still have to iterate over the list to check if an element already exists before you insert it. This would take up to $\Theta(m)$ time.

50. There are multiple ways to solve this problem. If linear probing is chosen, you would want to fill the entire table except index 7, since `"slime"` normally hashes to index 8 and would have to collide and wrap around the entire table before it reaches index 7. If quadratic probing is used (the easier approach), you can insert three strings that hash to index 8, so that inserting `"slime"` would cause it to end up at index $(8 + 3^2)$ mod 10 = 17 mod 10 = 7.

51. (a) One possible solution is as follows. We use the first hash to generate a hash for the given key. If a collision occurs, we use the second hash to generate another key, using the double hashing method until the key is successfully inserted into the table.

```
1   bool DoubleHashTable::insert_key(int32_t key) {
2     if (is_full()) {
3       return false;
4     } // if
5     size_t index = hash1(key) % table_size;
6     if (hash_table[index].status == Status::Empty) {
7       hash_table[index].status = Status::Occupied;
8       hash_table[index].key = key;
9     } // if
10    else {
11      size_t increment = hash2(key);
12      size_t j = 1;
13      while (true) {
14        size_t new_index = (index + j * increment) % table_size;
15        if (hash_table[new_index].status == Status::Empty) {
16          hash_table[new_index].status = Status::Occupied;
17          hash_table[new_index].key = key;
18          break;
19        } // if
20        ++j;
21      } // while
22    } // else
23    ++num_elements;
24    return true;
25  } // insert_key()
```

    (b) Here is how the elements are placed into this hash table:
    - 11 is placed at index 11 mod 13 = 11
    - 24 is placed at index 24 mod 13 = 11 (collision!)
        - 24 is placed at index (11 + 1(7 - (24 mod 7))) mod 13 = 15 mod 13 = 2
    - 25 is hashed to index 25 mod 13 = 12

- 41 is hashed to index 41 mod 13 = 2 (collision!)
  - 24 is placed at index (2 + 1(7 - (41 mod 7))) mod 13 = 3 mod 13 = 3
- 45 is hashed to index 45 mod 13 = 6
- 48 is hashed to index 48 mod 13 = 9

This is what the final table looks like:

		24	41			45			48		11	25
0	1	2	3	4	5	6	7	8	9	10	11	12

52. One possible solution is as follows: we can create an unordered set that stores the lengths of the tracks and check if a track exists in the unordered set such that two tracks are able to sum to the wait time. To prevent listening to the same track twice, we check the unordered set for a value before inserting into it.

```
bool identify_tracks(const std::vector<int32_t>& tracks, int32_t wait_time) {
  std::unordered_set<int32_t> track_lengths;
  for (int32_t curr_length : tracks) {
    if (track_lengths.find(wait_time - curr_length) != track_lengths.end()) {
      return true;
    } // if
    track_lengths.insert(curr_length);
  } // for
  return false;
} // identify_tracks()
```

53. This is a fairly standard hash table problem, where you are tasked to solve a problem that involves key-value lookups (where the key is a user and the value is the number of words they have sent). One such implementation is shown below. Here, we initialize an unordered map that maps from username to word count. We iterate over the list of messages and count the number of spaces (and add one) to get the word count of each message. We then add it to the value in the map associated with the user that sent the message. After all the messages are processed, we iterate over the map and return the user we encounter with the highest word count.

```
std::string user_with_highest_word_count(const std::vector<Message>& messages) {
  std::unordered_map<std::string, int32_t> word_count_by_user;
  for (const Message& message : messages) {
    // can also iterate over string and count spaces instead of std::count()
    word_count_by_user[message.sender] +=
      (std::count(message.text.begin(), message.text.end(), ' ') + 1);
  } // for

  std::string max_word_count_user;
  int32_t max_encountered = 0;
  for (auto [user, word_count] : word_count_by_user) {
    if (word_count == max_encountered) {
      max_word_count_user = std::max(max_word_count_user, user);
    } // if
    else if (word_count > max_encountered) {
      max_word_count_user = user;
      max_encountered = word_count;
    } // else if
  } // for

  return max_word_count_user;
} // user_with_highest_word_count()
```

54. One potential solution to this problem is to traverse over the vector of pairs and store the points into a hash table where the first values (x-coordinates) are keys and the second values (y-coordinates) are values. For each coordinate, we check if its y-coordinate exists as a key in this hash table. If so, compare the other coordinate and add to the result if equal.

```
std::vector<std::vector<std::pair<int32_t, int32_t>>> find_symmetric_pairs(
    const std::vector<std::pair<int32_t, int32_t>>& vec) {
  std::vector<std::vector<std::pair<int32_t, int32_t>>> result;
  std::unordered_map<int32_t, int32_t> xy_map;
  for (auto [x, y] : vec) {
    auto y_it = xy_map.find(y);
    if (y_it != xy_map.end() && y_it->second == x) {
      result.push_back({{x, y}, {y, x}});
    } // if
    else {
      xy_map[x] = y;
    } // else
  } // for

  return result;
} // find_symmetric_pairs()
```

55. One potential solution to this problem is to traverse over the vector and store the frequency of each element in a hash table, where the element is the key and the frequency is the value. Then, we identify the element with the greatest frequency and subtract this frequency from the number of elements in the vector to get the minimum number of deletions required.

```
int32_t min_deletions_for_all_equal(const std::vector<int32_t>& vec) {
  std::unordered_map<int32_t, int32_t> frequency_map;
  for (int32_t val : vec) {
    ++frequency_map[val];
  } // for

  int32_t max_freq_encountered = 0;
  for (auto [val, freq] : frequency_map) {
    if (freq > max_freq_encountered) {
      max_freq_encountered = freq;
    } // if
  } // for

  return vec.size() - max_freq_encountered;
} // min_deletions_for_all_equal()
```

56. One potential solution to this problem is to iterate over the given elements and map each element to the first index at which it occurs in the vector. Thus, whenever you encounter a repeated element, you will be able to easily identify that you have seen its value before as well as the first position it occurred at. This will allow you to keep track of the maximum distance you have seen so far.

```
int32_t max_repeated_distance(const std::vector<int32_t>& vec) {
  std::unordered_map<int32_t, int32_t> first_index_of_value;
  int32_t result = 0;
  for (int32_t i = 0; i < vec.size(); ++i) {
    auto first_index_it = first_index_of_value.find(vec[i]);
    if (first_index_it == first_index_of_value.end()) {
      first_index_of_value[vec[i]] = i;
    } // if
    else {
      result = std::max(result, i - first_index_it->second);
    } // else
  } // for i
  return result;
} // max_repeated_distance()
```

57. If you were to try to brute force this problem and check every possible subarray, you would end up with an $\Theta(n^2)$ solution since there are $\Theta(n^2)$ possible subarrays that can be generated out of the input array of n. Other techniques such as two pointer and sliding window do not work here, since the array gives you no information on how to move the two pointers or adjust your window (largely due to the fact that values can be negative). Instead, the optimal solution is keep track of an unordered map that maps each prefix sum to the number of times that sum is encountered. The algorithm is as follows:
    - Traverse the array and use the unordered map to keep track of the number of times you have encountered each prefix sum.
    - If the current prefix sum ever equals k, add one to the solution.
    - Look in the unordered map to see how many prefixes have a sum equal to the difference between the current sum and k (e.g., sum - k). We add this value to our solution (since we can remove these prefixes from the current sum to get a subarray that sums to k).
    - After traversing the entire array, return the calculated solution.

    For example, consider the example input `[2, -3, 4, -2, 1, -1]` and $k = -1$. We start traversing the array and visit the first value of 2, which corresponds to a prefix of `[2]`. Since this prefix is not equal to k, and the difference between it and k (2 - (-1) = 3) do not exist in the unordered map, we do not update our solution. This prefix is added to the map with an initial count of 1.

Key	Value
2	1

    Number of Subarrays: 0

    The next prefix we encounter is `[2, -3]`, which has a sum of `-1`. This prefix is equal to k, so we increment the number of subarrays encountered by 1. The difference between the prefix sum and k, or -1 - (-1) = 0, does not exist in the map, so we do not add anything additional to the solution. Lastly, we add the prefix sum of -1 to the map, with an initial count of 1.

Key	Value
2	1
-1	1

    Number of Subarrays: 1

    The next prefix we encounter is `[2, -3, 4]`, which has a sum of 3. This prefix sum is not equal to k, and the difference between this sum and k, or 3 - (-1) = 4, does not exist in the map, so we do not increment the number of subarrays at this step. We add the prefix sum of 3 to the map, with an initial count of 1.

Key	Value
2	1
-1	1
3	1

    Number of Subarrays: 1

The next prefix we encounter is `[2, -3, 4, -2]`, which has a sum of 1. This prefix sum is not equal to k, so we do not increment our solution from this check. However, the difference between this sum and k, or 1 - (-1) = 2, does exist in the map with a count of 1, so we add 1 to our solution. Why? Our unordered map indicates that there was 1 prefix that we previously encountered that summed to 2, so we can remove that 1 prefix from our current prefix to obtain a contiguous subarray with a sum of -1. That is, our current prefix `[2, -3, 4, -2]` does not sum to -1, but one earlier prefix (in this case, `[2]`) sums to the difference between the current prefix sum and k, so we can simply remove `[2]` from the beginning of `[2, -3, 4, -2]` to get a subarray `[-3, 4, -2]` that sums to $k = -1$. Note that if the prefix count for 2 were 2 instead of 1, we would add 2 to our solution (since there would be 2 prefixes that we would be able to remove from our current prefix to obtain a sum of k). Lastly, we add the original prefix sum of 1 to the map, with an initial count of 1.

Key	Value
2	1
-1	1
3	1
1	1

Number of Subarrays: 2

The next prefix we encounter is `[2, -3, 4, -2, 1]`, which has a sum of 2. This prefix sum is not equal to k, but the difference between this sum and k, or 2 - (-1) = 3, does exist in the map with a count of 1, so we add 1 to our solution. Since we encountered another prefix sum of 2, we increment its count in the unordered map.

Key	Value
2	2
-1	1
3	1
1	1

Number of Subarrays: 3

The last prefix we encounter is `[2, -3, 4, -2, 1, -1]`, which has a sum of 1. This prefix sum is not equal to k, but the difference between this sum and k, or 1 - (-1) = 2, does exist in the map with a count of 2, so we add 2 to our solution. Since we encountered another prefix sum of 1, we increment its count in the unordered map.

Key	Value
2	2
-1	1
3	1
1	2

Number of Subarrays: 5

We have completed our traversal of the array, so our solution is 5. An implementation of this solution is shown below:

```
1  int32_t num_subarrays_with_sum_k(const std::vector<int32_t>& vec, int32_t k) {
2    std::unordered_map<int32_t, int32_t> prefix_count;
3    int32_t num_subarrays_k = 0, prefix_sum = 0;
4    for (int32_t i = 0; i < vec.size(); ++i) {
5      prefix_sum += vec[i];
6      if (prefix_sum == k) {
7        ++num_subarrays_k;
8      } // if
9      auto diff_it = prefix_count.find(prefix_sum - k);
10     if (diff_it != prefix_count.end()) {
11       num_subarrays_k += diff_it->second;
12     } // if
13     ++prefix_count[prefix_sum];
14   } // for i
15   return num_subarrays_k;
16 } // num_subarrays_with_sum_k()
```

58. Performing a deep copy of the list without the random pointer is fairly straightforward (see chapter 8). The tricky part of this problem involves performing a deep copy of the list so that each random pointer points to correct node of the newly copied list. One solution is to use a hash table to map each node in the original list to the corresponding copy in the new list. We first build our deep copy normally without considering the random pointer (i.e., only setting the next value properly), inserting each copy into the hash table. Then, we iterate over each node of the original list and use the hash table to find the deep copy of the node associated with its random pointer, and then set the random pointer of the current node's copy to this deep copy. An implementation is shown below:

```
Node* copy_random_list(Node* head) {
  if (!head) {
    return nullptr;
  } // if
  std::unordered_map<Node*, Node*> orig_to_copy;

  // Deep copy of original list sans random pointer, adding the copy to the unordered map
  Node* head_copy = new Node(head->val);
  orig_to_copy.emplace(head, head_copy);
  Node* orig = head->next;
  Node* copy = head_copy;
  while (orig) {
    Node* orig_copy = new Node(orig->val);
    orig_to_copy.emplace(orig, orig_copy);
    copy->next = orig_copy;
    copy = orig_copy;
    orig = orig->next;
  } // while

  // Use unordered map to identify deep copy of random node, and set in new list
  orig = head;
  copy = head_copy;
  while (orig) {
    copy->random = orig->random ? orig_to_copy[orig->random] : nullptr;
    orig = orig->next;
    copy = copy->next;
  } // while

  return head_copy;
} // copy_random_list()
```

59. This problem can be solved using hash tables, since we can take advantage of fast lookups to retrieve information on each rider or route. There are several ways to tackle this problem, but one strategy is to keep track of unordered maps that:
    - map each rider ID to the number of rides they have taken
    - map each rider ID to information that determines if they are currently checked into a ride (i.e., start station and timestamp)
    - map each route to the total time taken and the number of rides (for calculating both the average time and the route ridership)

    Every time a rider checks in, we add them to a map to keep track of their starting station and timestamp, and every time they check out, we take this information to help compute the average time for each route. One implementation of this solution is provided below:

```
class TransitManager {
private:
  // Used to keep track of where a rider started if they are currently on a route
  struct CurrentRideInfo {
    std::string check_in_station;
    int64_t timestamp{};
  };

  // Used to keep track of information needed to calculate route info
  struct CurrentRouteInfo {
    int64_t total_time{};
    int64_t num_rides_on_route{};
  };

  std::unordered_map<int64_t, int64_t> rider_to_num_rides;
  std::unordered_map<int64_t, CurrentRideInfo> rider_to_current_ride;
  // This stores each route as a string, but there are other ways to do this
  std::unordered_map<std::string, CurrentRouteInfo> route_to_info;

public:
  void check_in(int64_t rider_id, const std::string& station_name, int64_t timestamp) {
    ++rider_to_num_rides[rider_id];
    rider_to_current_ride[rider_id] = CurrentRideInfo{.check_in_station = station_name,
                                                      .timestamp = timestamp};
  } // check_in()

  // ... continued on next page ...
```

```
    void check_out(int64_t rider_id, const std::string& station_name, int64_t timestamp) {
      auto [start_station, start_timestamp] = rider_to_current_ride[rider_id];
      std::string route = start_station + '-' + station_name;

      CurrentRouteInfo& route_info = route_to_info[route];
      route_info.total_time += (timestamp - start_timestamp);
      ++route_info.num_rides_on_route;

      rider_to_current_ride.erase(rider_id);
    } // check_out()

    double get_avg_time(const std::string& start_station, const std::string& end_station) {
      std::string route = start_station + '-' + end_station;
      const CurrentRouteInfo& route_info = route_to_info[route];
      return static_cast<double>(route_info.total_time) / route_info.num_rides_on_route;
    } // get_avg_time()

    int64_t get_num_rides(int64_t rider_id) {
      return rider_to_num_rides[rider_id];
    } // get_num_rides()

    int64_t get_num_route(const std::string& start_station, const std::string& end_station) {
      std::string route = start_station + '-' + end_station;
      return route_to_info[route].num_rides_on_route;
    } // get_num_route()
  };
```

60. This problem can be solved using a hash table that maps each account name to their account balance, with each of the operations looking up each account name and modifying their balance. One possible implementation of the problem is shown below:

```
  class TransactionManager {
  private:
    std::unordered_map<std::string, double> account_to_balance;

  public:
    bool open(const std::string& account_name) {
      auto account_it = account_to_balance.find(account_name);
      if (account_it != account_to_balance.end()) {
        return false;
      } // if
      // NOTE: emplace actually returns whether the key already exists, so you
      // can actually just call emplace and return the bool in the return pair here
      account_to_balance.emplace(account_name, 0);
      return true;
    } // open()

    bool deposit(const std::string& account_name, double amount) {
      auto account_it = account_to_balance.find(account_name);
      if (account_it == account_to_balance.end()) {
        return false;
      } // if
      account_it->second += amount;
      return true;
    } // deposit()

    bool withdraw(const std::string& account_name, double amount) {
      auto account_it = account_to_balance.find(account_name);
      if (account_it == account_to_balance.end() || account_it->second < amount) {
        return false;
      } // if
      account_it->second -= amount;
      return true;
    } // withdraw()

    bool transfer(const std::string& src_account, const std::string& dest_account, double amount) {
      auto src_account_it = account_to_balance.find(src_account);
      auto dest_account_it = account_to_balance.find(dest_account);
      if (src_account_it == account_to_balance.end() ||
          dest_account_it == account_to_balance.end()) {
        return false;
      } // if
      return withdraw(src_account, amount) && deposit(dest_account, amount);
    } // transfer()
  };
```

61. This problem can be solved using a hash table that maps table names to table info, where each method looks up the table and applies any requested changes. One possible implementation of the problem is shown below:

```
class Database {
private:
  struct TableData {
    std::unordered_map<int64_t, std::vector<std::string>> row_id_to_data;
    std::unordered_map<std::string, size_t> column_name_to_index;
    int64_t max_id = 0;
  };

  std::unordered_map<std::string, TableData> table_data;

public:
  bool create(const TableInfo& table_info) {
    auto table_it = table_data.find(table_info.table_name);
    if (table_it != table_data.end()) {
      return false;
    } // if
    auto& column_map = table_data[table_info.table_name].column_name_to_index;
    for (size_t i = 0; i < table_info.column_names.size(); ++i) {
      column_map[table_info.column_names[i]] = i;
    } // for
    return true;
  } // create()

  int64_t insert(const std::string& table_name, const std::vector<std::string>& row_data) {
    auto& table = table_data[table_name];
    table.row_id_to_data.emplace(++table.max_id, row_data);
    return table.max_id;
  } // insert()

  bool remove(const std::string& table_name, int64_t row_id) {
    auto& table = table_data[table_name];
    auto row_it = table.row_id_to_data.find(row_id);
    if (row_it == table.row_id_to_data.end()) {
      return false;
    } // if
    table.row_id_to_data.erase(row_it);
    return true;
  } // remove()

  std::optional<std::string> select(const std::string& table_name, int64_t row_id,
                                    const std::string& column_name) {
    const auto& table = table_data[table_name];
    auto row_it = table.row_id_to_data.find(row_id);
    if (row_it == table.row_id_to_data.end()) {
      return std::nullopt;
    } // if
    auto col_it = table.column_name_to_index.find(column_name);
    if (col_it == table.column_name_to_index.end()) {
      return std::nullopt;
    } // if
    return row_it->second[col_it->second];
  } // select()
};
```