Maps, and Hash Maps

Saikrishna Arcot
M. Hudachek-Buswell

July 18, 2020

Maps

- A map is a searchable, unsorted, unordered collection of key-value entries.

Maps

- A map is a searchable, unsorted, unordered collection of key-value entries.
- In a key-value pair, the key is unique, and can be any data type (later we look at the key being converted to an integer by some function). Every key has a single value associated with it.

Maps

- A map is a searchable, unsorted, unordered collection of key-value entries.
- In a key-value pair, the key is unique, and can be any data type (later we look at the key being converted to an integer by some function). Every key has a single value associated with it.
- The map stores the key-value pair based on the key. The value can be retreived from the map by using the key. In other words, it is a mapping from key to value.

Maps

- A map is a searchable, unsorted, unordered collection of key-value entries.
- In a key-value pair, the key is unique, and can be any data type (later we look at the key being converted to an integer by some function). Every key has a single value associated with it.
- The map stores the key-value pair based on the key. The value can be retreived from the map by using the key. In other words, it is a mapping from key to value.
- The value associated with the key can be any data type or object.

Maps

- A map is a searchable, unsorted, unordered collection of key-value entries.
- In a key-value pair, the key is unique, and can be any data type (later we look at the key being converted to an integer by some function). Every key has a single value associated with it.
- The map stores the key-value pair based on the key. The value can be retreived from the map by using the key. In other words, it is a mapping from key to value.
- The value associated with the key can be any data type or object.
- In some implementations, there may be duplicate values. For this course, multiple entries with the same key will NOT be allowed.

Maps

- The main operations of a map are searching, inserting, and deleting items.

Maps

- The main operations of a map are searching, inserting, and deleting items.
- Maps can be implemented using lists, arrays, or linked lists.

Maps

- The main operations of a map are searching, inserting, and deleting items.
- Maps can be implemented using lists, arrays, or linked lists.
- Implementations using either arrays or linked list are easy, and both result in $O(n)$ search performance.

Maps

- The main operations of a map are searching, inserting, and deleting items.
- Maps can be implemented using lists, arrays, or linked lists.
- Implementations using either arrays or linked list are easy, and both result in $O(n)$ search performance.
- Depending on the key, performance can be reduced to $O(\log n))$.

Maps

- The main operations of a map are searching, inserting, and deleting items.
- Maps can be implemented using lists, arrays, or linked lists.
- Implementations using either arrays or linked list are easy, and both result in $O(n)$ search performance.
- Depending on the key, performance can be reduced to $O(\log n)$).
- (If keys were like an index in an array, then access can be achieved in $O(1)$.)

Map ADT

- The Map Interface enforces the generic type <K,V>.
- The Map ADT has the following methods:
  - get(k): map M has an entry with key k, return its associated value; else, return null
  - put(k, v): insert entry (k, v) into map M; if key k is not already in M, then return null; else, return old value associated with k
  - remove(k): map M has an entry with key k, remove it from M and return its associated value; else, return null
  - size(), isEmpty()
  - entrySet(): return an iterable collection of the entries in M
  - keySet(): return an iterable collection of the keys in M
  - values(): return an iterator of the values in M

Hash Maps

- Maps support the abstraction of using *integer* keys as indices. The capacity of the map is N, and indices range 0 to $N-1$.

Hash Maps

- Maps support the abstraction of using *integer* keys as indices. The capacity of the map is N, and indices range 0 to $N-1$.
- A hash map is similar to a map, but it uses a hash function of the key to compute the index for storing the pair $<$K, V$>$ in an array.

Hash Maps

- Maps support the abstraction of using *integer* keys as indices. The capacity of the map is N, and indices range 0 to $N - 1$.
- A hash map is similar to a map, but it uses a hash function of the key to compute the index for storing the pair <K, V> in an array.
- In order to use the hash code, keys must be converted to integers, if not already integers.

Hash Maps

- Maps support the abstraction of using *integer* keys as indices. The capacity of the map is N, and indices range 0 to $N - 1$.
- A hash map is similar to a map, but it uses a hash function of the key to compute the index for storing the pair <K, V> in an array.
- In order to use the hash code, keys must be converted to integers, if not already integers.
- If the integer keys are not in the given range (0 to $N - 1$), then a HASH function is used to map the keys to corresponding indices in a table.

Hash Maps

- Maps support the abstraction of using *integer* keys as indices. The capacity of the map is N, and indices range 0 to $N - 1$.
- A hash map is similar to a map, but it uses a hash function of the key to compute the index for storing the pair <K, V> in an array.
- In order to use the hash code, keys must be converted to integers, if not already integers.
- If the integer keys are not in the given range (0 to $N - 1$), then a HASH function is used to map the keys to corresponding indices in a table.
- Hash maps are generally implemented using an array (so that you have $O(1)$ access to any index).

Hash Maps vs. Hash Tables

- There are some distinct differences with Hash Maps and Hash Tables:

Hash Maps vs. Hash Tables

- There are some distinct differences with Hash Maps and Hash Tables:
  - Hash Maps allow *null* keys and values. Hash Tables do not allow for the use of null

Hash Maps vs. Hash Tables

- There are some distinct differences with Hash Maps and Hash Tables:
  - Hash Maps allow *null* keys and values. Hash Tables do not allow for the use of null
  - Hash Tables are thread safe because they are synchronized. Hash Maps do not synchronize.

Hash Maps vs. Hash Tables

- There are some distinct differences with Hash Maps and Hash Tables:
  - Hash Maps allow *null* keys and values. Hash Tables do not allow for the use of null
  - Hash Tables are thread safe because they are synchronized. Hash Maps do not synchronize.
  - Hash Maps use *iterator* to iterate through its object values. Hash Tables use enumerator.

Hash Maps vs. Hash Tables

- There are some distinct differences with Hash Maps and Hash Tables:
  - Hash Maps allow *null* keys and values. Hash Tables do not allow for the use of null
  - Hash Tables are thread safe because they are synchronized. Hash Maps do not synchronize.
  - Hash Maps use *iterator* to iterate through its object values. Hash Tables use enumerator.
  - Hash Maps are much faster and use less memory than Hash Tables.

Index Calculation

- The hash code of the key could be used to determine what index the key-value pair should go into, but the range of values that a hash code might take on may have no bounds, or might be outside of the bounds of the array.

Index Calculation

- The hash code of the key could be used to determine what index the key-value pair should go into, but the range of values that a hash code might take on may have no bounds, or might be outside of the bounds of the array.
  - For example, if the hash code of an object is 138129, you do not want to allocate an array of length 138130 just so you can add this item.

Index Calculation

- The hash code of the key could be used to determine what index the key-value pair should go into, but the range of values that a hash code might take on may have no bounds, or might be outside of the bounds of the array.
  - For example, if the hash code of an object is 138129, you do not want to allocate an array of length 138130 just so you can add this item.
  - Similarly, if the hash code of an object is -1829123, in some programming languages, this would be an invalid index regardless of the length of the array.

Index Calculation

- As a result, the resulting hash code is reduced to the length of the backing array by using the modulo operator (%). (The modulus of $a\%N$, where a and N are positive, is guaranteed to return a number between 0 and $N - 1$).

Index Calculation

- As a result, the resulting hash code is reduced to the length of the backing array by using the modulo operator (%). (The modulus of $a\%N$, where a and N are positive, is guaranteed to return a number between 0 and $N-1$).
- The hash code is modded by the length of the array, so that you have a number that is a valid index.

Index Calculation

- As a result, the resulting hash code is reduced to the length of the backing array by using the modulo operator (%). (The modulus of $a\%N$, where a and N are positive, is guaranteed to return a number between 0 and $N - 1$).
- The hash code is modded by the length of the array, so that you have a number that is a valid index.
- Essentially you have compressed the integer key to an index in the array.

Load Factor

- Hash maps have a property called the load factor that determines when the hash map should be resized.

Load Factor

- Hash maps have a property called the load factor that determines when the hash map should be resized.
- The load factor is calculated by dividing the number of key-value pairs in the hash map by the length of the array.

Load Factor

- Hash maps have a property called the load factor that determines when the hash map should be resized.
- The load factor is calculated by dividing the number of key-value pairs in the hash map by the length of the array.
- For example, if there are 8 items stored in a hash map, and the length of the array used to store these items is 11, then the load factor is $\frac{8}{11} \approx 0.73$ (regardless of collision resolution method).

Load Factor

- When the load factor exceeds a certain threshold, the backing array is resized (the new length is implementation defined, but one formula that might be used is $2n + 1$, where n is the length of the old backing array).

Load Factor

- When the load factor exceeds a certain threshold, the backing array is resized (the new length is implementation defined, but one formula that might be used is $2n + 1$, where n is the length of the old backing array).
- When resizing, the ideal index of each key-value pair needs to be recalculated, because that index is based on the length of the table. (Just copying over the items to the same indices will not work).

Load Factor

- When the load factor exceeds a certain threshold, the backing array is resized (the new length is implementation defined, but one formula that might be used is $2n + 1$, where n is the length of the old backing array).
- When resizing, the ideal index of each key-value pair needs to be recalculated, because that index is based on the length of the table. (Just copying over the items to the same indices will not work).
- The load factor threshold varies depending on the size of the backing array and the rate of collision occurence. Most implementations try to keep the load factor below 0.75.

Properties of Hash Maps

- Because hash maps use the hash code of the key to find the associated value, searches tend to be $O(1)$ (or close to $O(1)$, as we'll see later).

Properties of Hash Maps

- Because hash maps use the hash code of the key to find the associated value, searches tend to be $O(1)$ (or close to $O(1)$, as we'll see later).
- As with maps, hash maps generally have unique keys, but there may be duplicate values.

Properties of Hash Maps

- Because hash maps use the hash code of the key to find the associated value, searches tend to be $O(1)$ (or close to $O(1)$, as we'll see later).
- As with maps, hash maps generally have unique keys, but there may be duplicate values.
- The length of the backing array tends to be a prime number, so that when the hash map is resized, if two items collided in the smaller hash map, they (hopefully) won't collide in the larger hash map.

Collision Resolution Methods

- Because we are modding the hash code by the length of the array to calculate the “ideal” index, it is possible that two keys would need to go into the same index. (For example, both 9%5 and 14%5 are 4.)

Collision Resolution Methods

- Because we are modding the hash code by the length of the array to calculate the "ideal" index, it is possible that two keys would need to go into the same index. (For example, both 9%5 and 14%5 are 4.)
- In addition, two hash codes that are the same might be different objects (particularly in languages where there is a fixed range of numbers that the hash code can take on), which would also mean that two different object would need to go into the same index.

Collision Resolution Methods

- Because we are modding the hash code by the length of the array to calculate the "ideal" index, it is possible that two keys would need to go into the same index. (For example, both 9%5 and 14%5 are 4.)
- In addition, two hash codes that are the same might be different objects (particularly in languages where there is a fixed range of numbers that the hash code can take on), which would also mean that two different object would need to go into the same index.
- There needs to be a way to resolve these collisions. As a result, there are several collision resolution methods that can be used.

Examples

Note that the examples in these slides show only the key, and not the value. The value is not used when adding a new key-value pair. (You can also have a version of a hash map where only one thing is stored; this is called a hash set.)

External Chaining

- The first method is called external chaining. With external chaining, there is essentially a linked list in each index of the array.

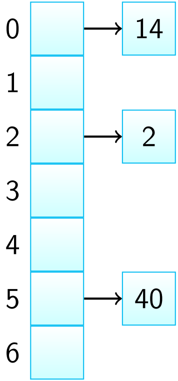
External Chaining

- The first method is called external chaining. With external chaining, there is essentially a linked list in each index of the array.
- When a collision occurs, the new key-value pair is added into the linked list.

Adding

For example, when adding 25 (assume the hash code is the number itself):

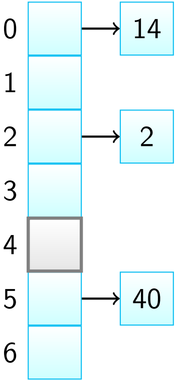
25 would go into index 25 % 7=4



Adding

For example, when adding 25 (assume the hash code is the number itself):

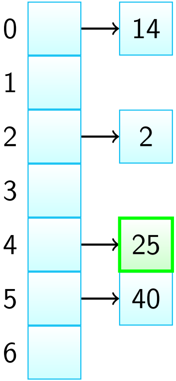
25 would go into index 25 % 7=4



Adding

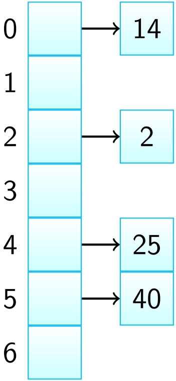
For example, when adding 25 (assume the hash code is the number itself):

25 would go into index 25 % 7=4



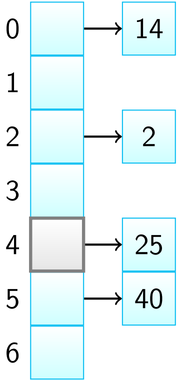
Adding

For example, when adding 18 (assume the hash code is the number itself):
18 would go into index 18 % 7=4



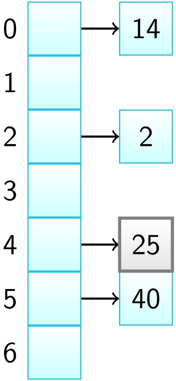
Adding

For example, when adding 18 (assume the hash code is the number itself):
18 would go into index 18 % 7=4



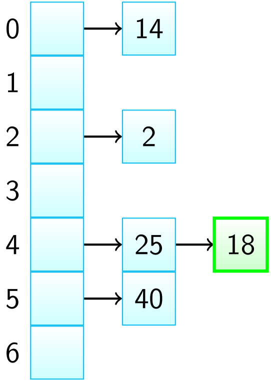
Adding

For example, when adding 18 (assume the hash code is the number itself):
18 would go into index 18 % 7=4



Adding

For example, when adding 18 (assume the hash code is the number itself):
18 would go into index 18 % 7=4



Searching and Removing

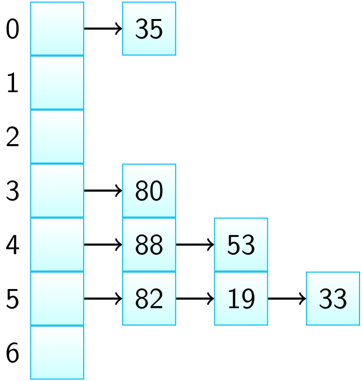
- When searching, first determine the index the key would be in, and search the linked list for the key.

Searching and Removing

- When searching, first determine the index the key would be in, and search the linked list for the key.
- To remove a key-value pair, search for the key and remove that key-value pair from the linked list.

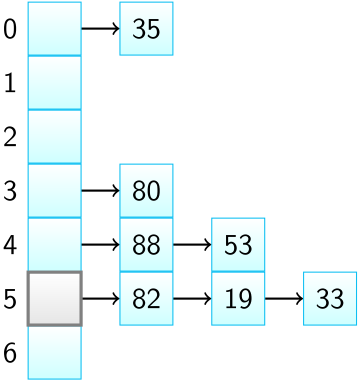
Searching

For example, when searching for 33 (assume the hash code is the number itself):
33 would go into index 33 % 7=5



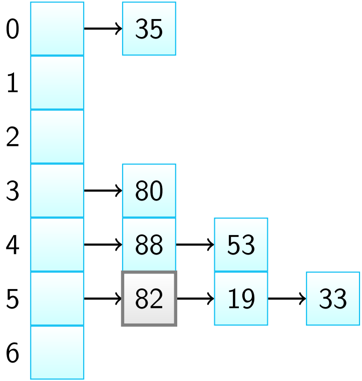
Searching

For example, when searching for 33 (assume the hash code is the number itself):
33 would go into index 33 % 7=5



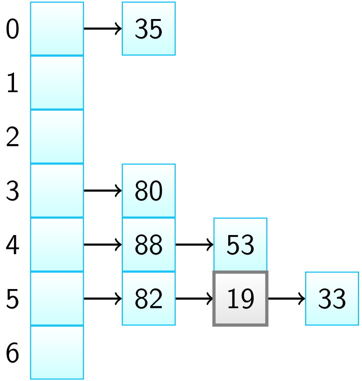
Searching

For example, when searching for 33 (assume the hash code is the number itself):
33 would go into index 33 % 7=5



Searching

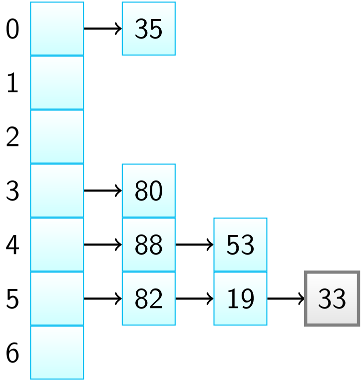
For example, when searching for 33 (assume the hash code is the number itself):
33 would go into index 33 % 7=5



Searching

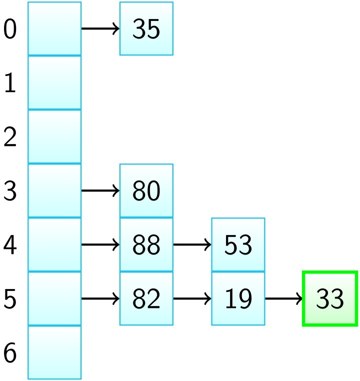
For example, when searching for 33 (assume the hash code is the number itself):

33 would go into index 33 % 7=5



Searching

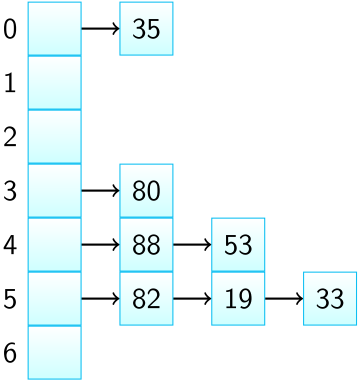
For example, when searching for 33 (assume the hash code is the number itself):
33 would go into index 33 % 7=5



Removing

For example, when removing 19 (assume the hash code is the number itself):

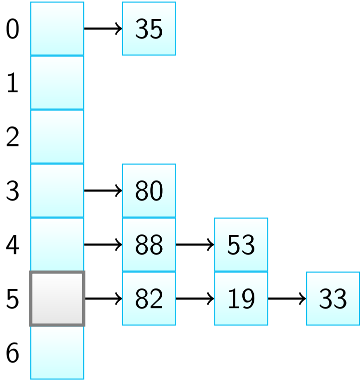
19 would go into index 19 % 7=5



Removing

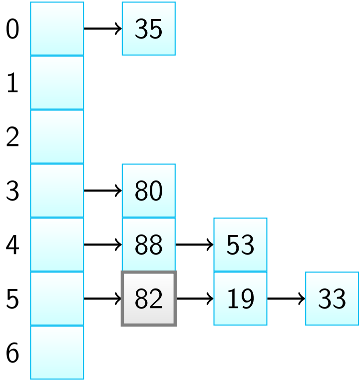
For example, when removing 19 (assume the hash code is the number itself):

19 would go into index 19 % 7=5



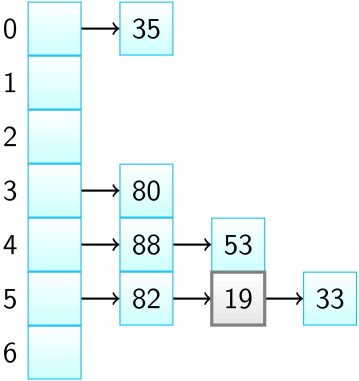
Removing

For example, when removing 19 (assume the hash code is the number itself):
19 would go into index 19 % 7=5



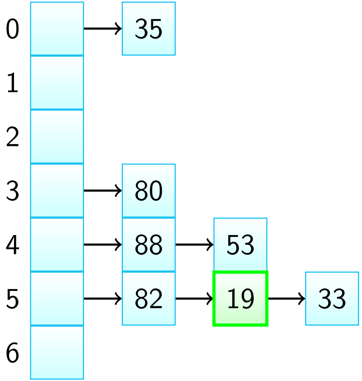
Removing

For example, when removing 19 (assume the hash code is the number itself):
19 would go into index 19 % 7=5



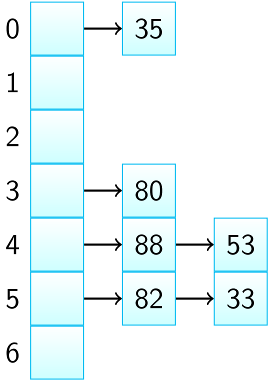
Removing

For example, when removing 19 (assume the hash code is the number itself):
19 would go into index 19 % 7=5



Removing

For example, when removing 19 (assume the hash code is the number itself):
19 would go into index 19 % 7=5



Linear Probing

- The second of these methods is called linear probing. In this method, if a key-value pair is being added into index i, but there is already another key-value pair at index i, and the keys are different, then index $i + 1$ is checked.

Linear Probing

- The second of these methods is called linear probing. In this method, if a key-value pair is being added into index i, but there is already another key-value pair at index i, and the keys are different, then index $i + 1$ is checked.
- If index $i + 1$ already has a key-value pair, and the keys are different, then index $i + 2$ is checked. If that index isn't empty, then $i + 3$ is checked, and so on. (If you reach the end of the array, go to the starting of the array.)

Linear Probing

- The second of these methods is called linear probing. In this method, if a key-value pair is being added into index i, but there is already another key-value pair at index i, and the keys are different, then index $i + 1$ is checked.
- If index $i + 1$ already has a key-value pair, and the keys are different, then index $i + 2$ is checked. If that index isn't empty, then $i + 3$ is checked, and so on. (If you reach the end of the array, go to the starting of the array.)
- The search stops when you find a slot where you can insert the new key-value pair.

Adding

For example, when adding 18 (assume the hash code is the number itself):
18 would go into index 18 % 7=4

0	
1	
2	
3	10
4	11
5	5
6	

Adding

For example, when adding 18 (assume the hash code is the number itself):
18 would go into index 18 % 7=4

0	
1	
2	
3	10
4	11
5	5
6	

Adding

For example, when adding 18 (assume the hash code is the number itself):
18 would go into index 18 % 7=4

0	
1	
2	
3	10
4	11
5	5
6	

Adding

For example, when adding 18 (assume the hash code is the number itself):
18 would go into index 18 % 7=4

0	
1	
2	
3	10
4	11
5	5
6	

Adding

For example, when adding 18 (assume the hash code is the number itself):
18 would go into index 18 % 7=4

0	
1	
2	
3	10
4	11
5	5
6	18

Searching

- When searching for a key, calculate the index that the key would ideally be at.

Searching

- When searching for a key, calculate the index that the key would ideally be at.
- If the key is at that index, then you've found the key, and you can get the value.

Searching

- When searching for a key, calculate the index that the key would ideally be at.
- If the key is at that index, then you've found the key, and you can get the value.
- If the key is not at that index, then go on to the next index (loop back around if at the end of the array).

Searching

- When searching for a key, calculate the index that the key would ideally be at.
- If the key is at that index, then you've found the key, and you can get the value.
- If the key is not at that index, then go on to the next index (loop back around if at the end of the array).
- Continue until you find the key, you reach an index with `NULL` at that index (more on this later), or you go through the entire array.

Searching

For example, when searching for 9 (assume the hash code is the number itself):
9 would ideally be at index 9 % 7=2

0	
1	
2	72
3	9
4	23
5	4
6	18

Searching

For example, when searching for 9 (assume the hash code is the number itself):

9 would ideally be at index 9 % 7=2

0	
1	
2	72
3	9
4	23
5	4
6	18

Searching

For example, when searching for 9 (assume the hash code is the number itself):

9 would ideally be at index 9 % 7=2

0	
1	
2	72
3	9
4	23
5	4
6	18

Searching

For example, when searching for 9 (assume the hash code is the number itself):
9 would ideally be at index 9 % 7=2

0	
1	
2	72
3	9
4	23
5	4
6	18

Searching

For example, when searching for 30 (assume the hash code is the number itself):

30 would ideally be at index 30 % 7=2

0	
1	
2	72
3	9
4	23
5	4
6	18

Searching

For example, when searching for 30 (assume the hash code is the number itself):

30 would ideally be at index 30 % 7=2

0	
1	
2	72
3	9
4	23
5	4
6	18

Searching

For example, when searching for 30 (assume the hash code is the number itself):
30 would ideally be at index 30 % 7=2

0	
1	
2	72
3	9
4	23
5	4
6	18

Searching

For example, when searching for 30 (assume the hash code is the number itself):

30 would ideally be at index 30 % 7=2

0	
1	
2	72
3	9
4	23
5	4
6	18

Searching

For example, when searching for 30 (assume the hash code is the number itself):
30 would ideally be at index 30 % 7=2

0	
1	
2	72
3	9
4	23
5	4
6	18

Searching

For example, when searching for 30 (assume the hash code is the number itself):
30 would ideally be at index 30 % 7=2

0	
1	
2	72
3	9
4	23
5	4
6	18

Searching

For example, when searching for 30 (assume the hash code is the number itself):
30 would ideally be at index 30 % 7=2

0	
1	
2	72
3	9
4	23
5	4
6	18

Searching

For example, when searching for 30 (assume the hash code is the number itself):

30 would ideally be at index 30 % 7=2

0	
1	
2	72
3	9
4	23
5	4
6	18

30 is not in the hash map.

Removing

- When removing a key-value pair, once we find the key in the array, we could just set that index to `NULL`.

Removing

- When removing a key-value pair, once we find the key in the array, we could just set that index to `NULL`.
- However, this will break searching for a key that had to be placed somewhere besides the ideal index.

Removing

- When removing a key-value pair, once we find the key in the array, we could just set that index to `NULL`.
- However, this will break searching for a key that had to be placed somewhere besides the ideal index.
- As a result, when removing a key-value pair, we set a deleted/defunct marker that indicates that something was stored at this index, and to continue searching.

Removing

For example, when removing 72 (assume the hash code is the number itself), and setting the index to `NULL`:
72 would ideally be at index 72 % 7=2

0	
1	
2	72
3	9
4	23
5	4
6	18

Removing

For example, when removing 72 (assume the hash code is the number itself), and setting the index to `NULL`:
72 would ideally be at index 72 % 7=2

0	
1	
2	72
3	9
4	23
5	4
6	18

Removing

For example, when removing 72 (assume the hash code is the number itself), and setting the index to `NULL`:
72 would ideally be at index 72 % 7=2

0	
1	
2	72
3	9
4	23
5	4
6	18

Removing

For example, when removing 72 (assume the hash code is the number itself), and setting the index to `NULL`:
72 would ideally be at index 72 % 7=2

0	
1	
2	
3	9
4	23
5	4
6	18

Now try searching for 9; what happens?

Removing

For example, when removing 72 (assume the hash code is the number itself), and setting a deleted marker:
72 would ideally be at index 72 % 7=2

0	
1	
2	72
3	9
4	23
5	4
6	18

Removing

For example, when removing 72 (assume the hash code is the number itself), and setting a deleted marker:
72 would ideally be at index 72 % 7=2

0	
1	
2	72
3	9
4	23
5	4
6	18

Removing

For example, when removing 72 (assume the hash code is the number itself), and setting a deleted marker:
72 would ideally be at index 72 % 7=2

0	
1	
2	72
3	9
4	23
5	4
6	18

Removing

For example, when removing 72 (assume the hash code is the number itself), and setting a deleted marker:
72 would ideally be at index 72 % 7=2

0	
1	
2	DELETED
3	9
4	23
5	4
6	18

Now try searching for 9; what happens?

Back to Searching

To recap, when searching for a key, and the key is not at the ideal index, continue searching until you find the key, you reach an index with `NULL` at that index (i.e. an index where nothing was ever added to that index), or you go through the entire array.

Back to Adding

- Because of the deleted markers, adding to a linear-probing hash map becomes a little more complicated.

Back to Adding

- Because of the deleted markers, adding to a linear-probing hash map becomes a little more complicated.
- When adding to a linear-probing hash map, when you first come across an index with the deleted marker set, save that index, and continue searching in the next index.

Back to Adding

- Because of the deleted markers, adding to a linear-probing hash map becomes a little more complicated.
- When adding to a linear-probing hash map, when you first come across an index with the deleted marker set, save that index, and continue searching in the next index.
- If you later find that the key is already in the hash map, update the value associated with the key **at that index** (do not use the index you saved earlier).

Back to Adding

- If you don't find the key and instead reach an index with `NULL`, or you go through the entire array, add the new key-value pair at the index you saved earlier.

Back to Adding

- If you don't find the key and instead reach an index with `NULL`, or you go through the entire array, add the new key-value pair at the index you saved earlier.
- This is done because we want the key-value pairs as close to the ideal index as possible, so that the search time is closer to $O(1)$ than $O(n)$.

Quadratic Probing

- The third method is called quadratic probing. This is the same as linear probing except instead of checking i, $i+1$, $i+2$, $i+3$, $\ldots$, you instead check i, $i+1^2$, $i+2^2$, $i+3^2$, $\ldots$.

Quadratic Probing

- The third method is called quadratic probing. This is the same as linear probing except instead of checking i, $i + 1$, $i + 2$, $i + 3$, $\ldots$, you instead check i, $i + 1^2$, $i + 2^2$, $i + 3^2$, $\ldots$.
- The benefit of this is that the collisions are spaced out in the hash map, and other key-value pairs might not be shifted too much from their ideal index (maybe).

Adding

For example, when adding 27 (assume the hash code is the number itself):
27 would go into index 27 % 7=6

0	7
1	
2	
3	
4	
5	
6	13

Adding

For example, when adding 27 (assume the hash code is the number itself):

27 would go into index 27 % 7=6

0	7
1	
2	
3	
4	
5	
6	$\boxed{13}$

Adding

For example, when adding 27 (assume the hash code is the number itself):

27 would go into index 27 % 7=6

0	7
1	
2	
3	
4	
5	
6	13

Adding

For example, when adding 27 (assume the hash code is the number itself):

27 would go into index 27 % 7=6

0	7
1	
2	
3	
4	
5	
6	13

Adding

For example, when adding 27 (assume the hash code is the number itself):

27 would go into index 27 % 7=6

0	7
1	
2	
3	27
4	
5	
6	13

Forced Resizing

- With quadratic probing, despite there being empty slots in the backing array, it is possible that a key-value pair cannot be added in.

Forced Resizing

- With quadratic probing, despite there being empty slots in the backing array, it is possible that a key-value pair cannot be added in.
- This is because not all of the indices will be checked when doing quadratic probing.

Forced Resizing

- With quadratic probing, despite there being empty slots in the backing array, it is possible that a key-value pair cannot be added in.
- This is because not all of the indices will be checked when doing quadratic probing.
- For example, for a hash map with a backing array of length 7, if a key's ideal index is 6, and quadratic probing needs to be done, the indices that will be checked are 6, 0, 3, 1, 1, 3, 0, 6, etc. (note how only indices 0, 1, 3, and 6 will be checked).

Forced Resizing

- In such cases, the backing array will need to be resized (regardless of the load factor) so that the key-value pair can be added in.

Forced Resizing

- In such cases, the backing array will need to be resized (regardless of the load factor) so that the key-value pair can be added in.
- In general, with a backing array of length n, if you have probed n times, then you must resize the array.

Forced Resizing

For example, when adding 44 (assume the hash code is the number itself):
44 would go into index 44 % 7=2

0	
1	
2	9
3	17
4	32
5	
6	6

Forced Resizing

For example, when adding 44 (assume the hash code is the number itself):

44 would go into index 44 % 7=2

0	
1	
2	9
3	17
4	32
5	
6	6

Forced Resizing

For example, when adding 44 (assume the hash code is the number itself):

44 would go into index 44 % 7=2

0	
1	
2	9
3	17
4	32
5	
6	6

Forced Resizing

For example, when adding 44 (assume the hash code is the number itself):

44 would go into index 44 % 7=2

0	
1	
2	9
3	17
4	32
5	
6	6

Forced Resizing

For example, when adding 44 (assume the hash code is the number itself):
44 would go into index 44 % 7=2

0	
1	
2	9
3	17
4	32
5	
6	6

Forced Resizing

For example, when adding 44 (assume the hash code is the number itself):
44 would go into index 44 % 7=2

0	
1	
2	9
3	17
4	32
5	
6	6

Forced Resizing

For example, when adding 44 (assume the hash code is the number itself):

44 would go into index 44 % 7=2

0	
1	
2	9
3	17
4	32
5	
6	6

And the cycle continues...

Performance

- Given that the keys give a variety of different hash codes for different objects (i.e. it isn't just a number between 1 and 10 when the key can be 1000 unique objects), few collisions *should* occur, and adding, searching, and removing from a hash map is $O(1)$.

Performance

- Given that the keys give a variety of different hash codes for different objects (i.e. it isn't just a number between 1 and 10 when the key can be 1000 unique objects), few collisions *should* occur, and adding, searching, and removing from a hash map is $O(1)$.
- If the keys have a limited range of hash codes, or collisions occur for other reasons, then adding, searching, and removing from a hash map is $O(n)$.