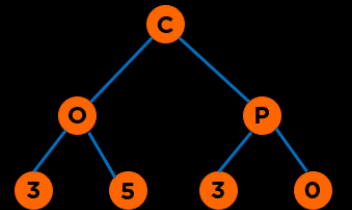


Sets, Maps and Hash Tables



Categories of Data Structures

Linear Ordered

Lists

Stacks

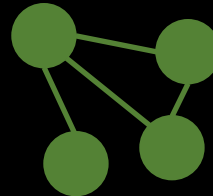
Queues



Non-linear Ordered

Trees

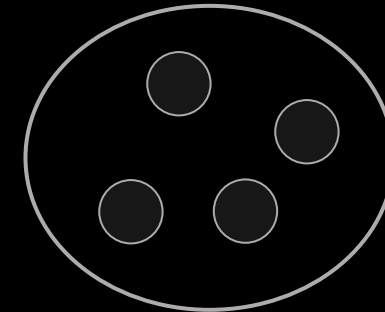
Graphs



Not Ordered

Sets

Tables/Maps



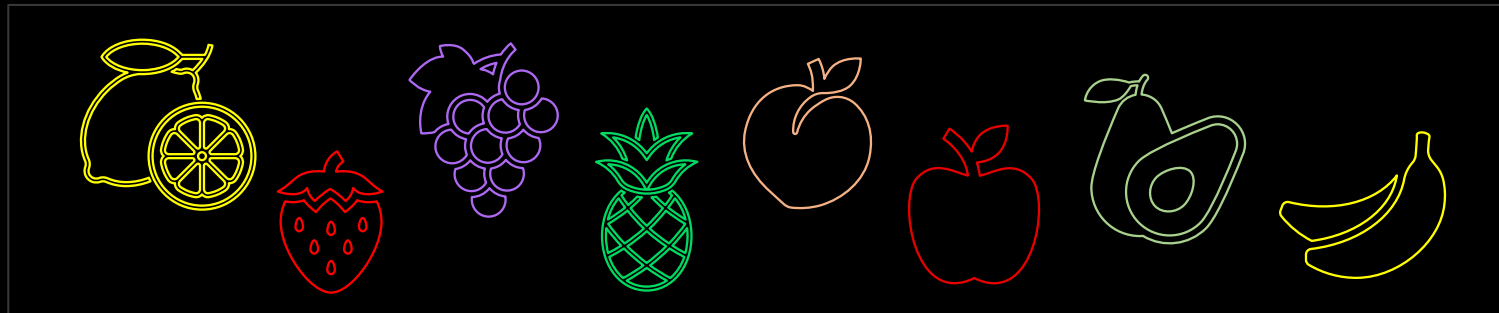
Sets

Sets

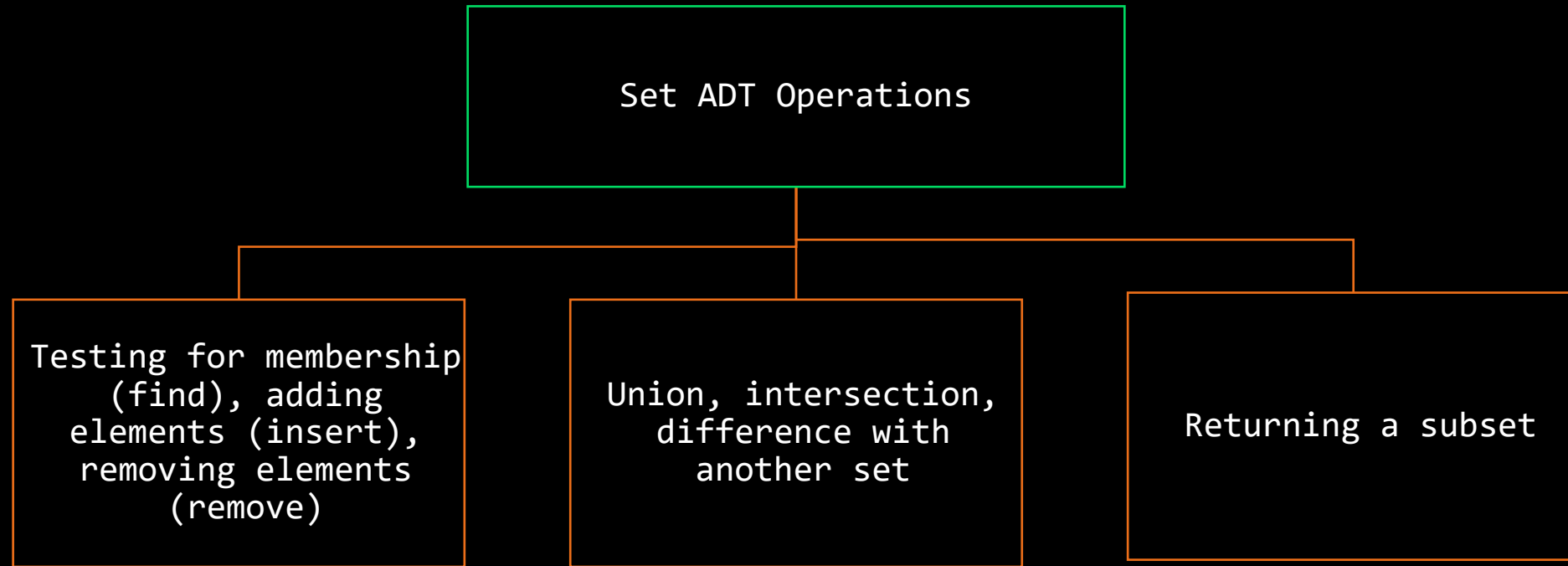
A set is a collection that contains no duplicate elements

Set objects

- are not indexed
- do not reveal the order of insertion of items
- do enable efficient search and retrieval of information
- do allow removal of elements without moving other elements around

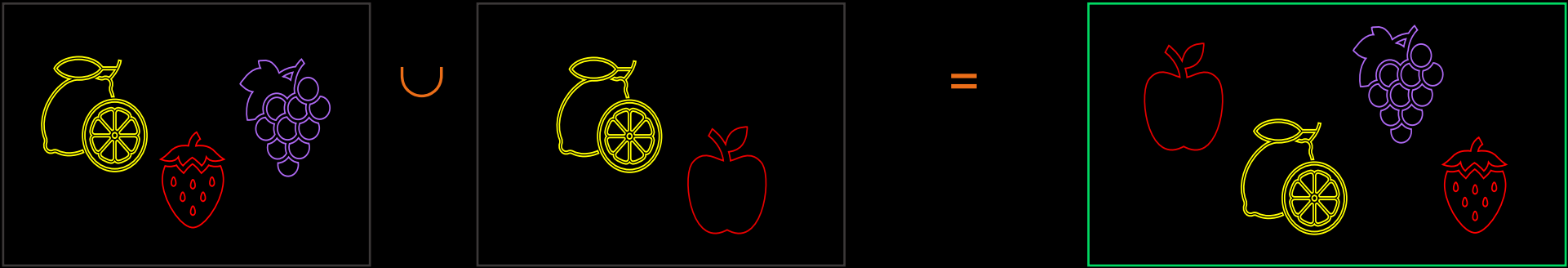


Sets

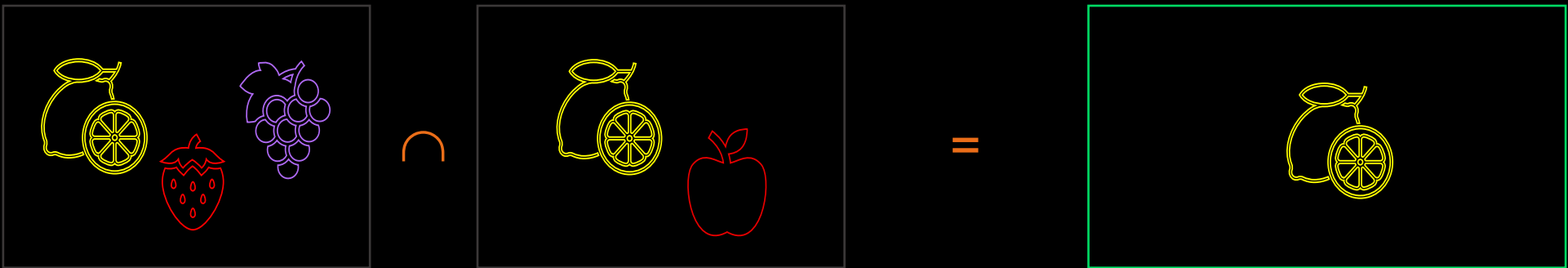


Sets

Union of two sets, $A \cup B$ is a set whose elements belong either to A or B or to both A and B.

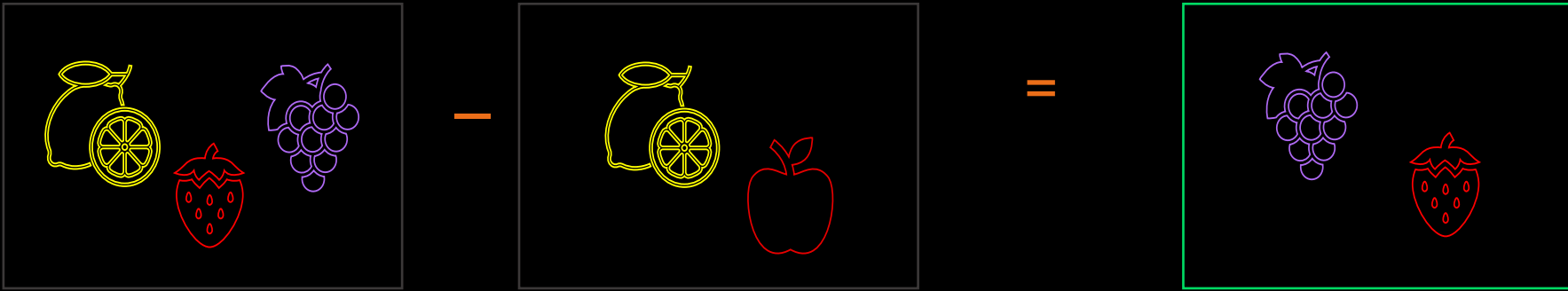


Intersection of sets $A \cap B$ is the set whose elements belong to both A and B.

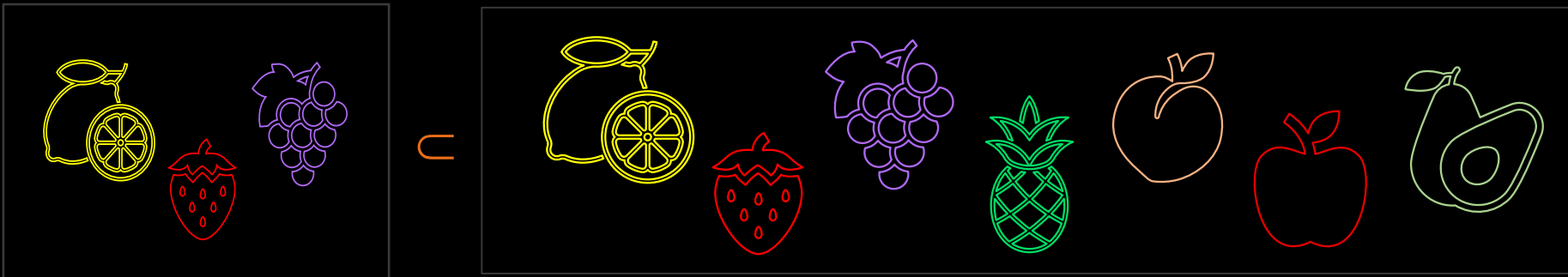


Sets

Difference of sets $A - B$ is the set whose elements belong to A but not to B .

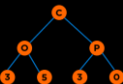


Set A is a subset of set B , $A \subset B$ if every element of set A is also an element of set B .



Lists vs Sets

	Lists	Sets
Order and Access through Element Index	Yes	No
Duplicates	Yes	No
Implementations	Array Based, Linked Lists	Array Based, Tree Based



Sets in C++

	<code>std::set</code>	<code>std::unordered_set</code>
Order in Elements	Yes	No
Initialization	<code>std::set<type> s;</code>	<code>std::unordered_set<type> s;</code>
Common Methods	<code>insert, erase, find, count, size, empty</code>	<code>insert, erase, find, count, size, empty, bucket_size, load_factor</code>
Implementations	Binary Search Tree (TreeSet)	Hash Table (Hash Set)
Time Complexity of Common Operations	$O(\log n)$ for a Self-Balancing BST, e.g. Red Black Tree	$O(1) + O(k)$ for hash

<http://www.cplusplus.com/reference/set/set/>
http://www.cplusplus.com/reference/unordered_set/unordered_set/

Sets in C++ Example

```
01 // Ordered tree-based set
02 set<int> s1;
03
04 // insert elements in random order
05 s1.insert(5);
06 s1.insert(2);
07 s1.insert(4);
08 s1.insert(11);
09 s1.insert(2); // only one 2 will be added to the set
10
11 // printing set
12 set<int> :: iterator itr1;
13 cout << "The set s1 is : ";
14 for (itr1 = s1.begin(); itr1 != s1.end(); ++itr1)
15     cout << " " << *itr1;
```

The set s1 is : 2 4 5 11

```
01 //Unordered Set - Hash-based
02 unordered_set<int> s2;
03
04 // insert elements in random order
05 s2.insert(5);
06 s2.insert(2);
07 s2.insert(4);
08 s2.insert(11);
09 s2.insert(2); // only one 2 will be added to the set
10
11 // printing set
12 unordered_set<int> :: iterator itr2;
13 cout << "The set s2 is:";
14 for (itr2 = s2.begin(); itr2 != s2.end(); ++itr2)
15     cout << " " << *itr2;
16 cout << endl;
17 cout << "Bucket count: " << s2.bucket_count();
18 cout << "\nLoad Factor: " << s2.load_factor();
19 cout << "\nMax Load Factor:" << s2.max_load_factor();
```

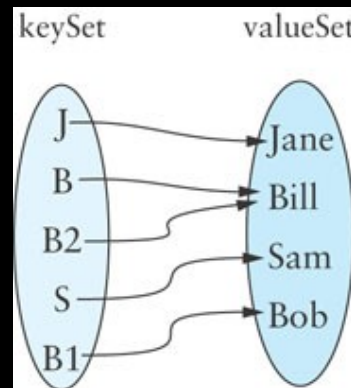
The set s2 is: 11 4 5 2
Bucket count: 7
Load Factor: 0.571429
Max Load Factor: 1

Maps

Maps

A map is a collection of key-value pairs that do not contain duplicate keys.

- Maps are sort of an abstraction over Sets
- The Keys in a map are a Set.
- Values can be non-unique [Many-to-One Relationship, Onto Mapping]
- If you store values along with keys in a Set data structure, you get a Map



`{(J, Jane), (B, Bill),
(S, Sam), (B1, Bob),
(B2, Bill)}`

Maps

Type of item	Key	Value
University student	Student ID number	Student name, address, major, grade point average
Online store customer	E-mail address	Customer name, address, credit card information, shopping cart
Inventory item	Part ID	Description, quantity, manufacturer, cost, price

Maps in C++

	<code>std::map</code>	<code>std::unordered_map</code>
Order in Elements	Yes	No
Initialization (Internally stored as pairs)	<code>std::map<type, type> m;</code>	<code>std::unordered_map <type, type> m;</code>
Common Methods	<code>insert, [], erase, find, count, size, empty</code>	<code>insert, [], erase, find, count, size, empty, bucket_size, load_factor</code>
Implementations	Binary Search Tree (TreeMap)	Hash Table (Hash Map)

<http://www.cplusplus.com/reference/map/map/>
http://www.cplusplus.com/reference/unordered_map/unordered_map/

Maps in C++ Example

```
01 // Ordered tree-based map
02 map<char,int> table;
03
04 // insert elements in random order
05 table['b']=30;
06 table['a']=10;
07 table['c']=50;
08 table['a']=40;
09
10 // printing map
11 for(auto member: table)
12     cout << member.first << " " << member.second << "\n";
```

```
a 40
b 30
c 50
```

```
01 //Unordered Map - Hash-based
02 unordered_map<char,int> table_unordered;
03
04 // insert elements in random order
05 table_unordered['b']=30;
06 table_unordered['a']=10;
07 table_unordered['c']=50;
08 table_unordered['a']=40;
09
10 // printing set
11 for(auto member: table_unordered)
12     cout << member.first << " " << member.second << "\n";
13
14 cout << "Load Factor: " << table_unordered.load_factor();
```

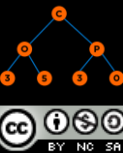
```
c 50
b 30
a 40
Load Factor: 0.428571
```

Questions

Hash Tables

Problems with Tree Based Maps and Sets

- If the datatypes are comparable such as integers or characters, tree-based maps and sets makes sense. **What if the data itself is incomparable?**
- Common operations such as `insert()` or `search()` are $O(\log n)$. **Can we do better than this?**



What if we use an Array: Exploiting constant access?

- Let's say we want to insert 11, 2 and 5 into a set
- Initially all values are false

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

What if we use an Array: Exploiting constant access?

- Let's say we want to insert 11, 2 and 5 into a set
- Initially all values are false
- When we insert an item, we set the value at index to true

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

What if we use an Array: Exploiting constant access?

```
01 class ArraySet
02 {
03     private:
04         bool set[100] = {0};
05     public:
06         void insert(int value);
07         bool search(int value);
08 };
09
10 void ArraySet::insert(int value)
11 {
12     set[value] = 1;
13 }
14
15 bool ArraySet::search(int value)
16 {
17     return set[value];
18 }
```

```
19 int main()
20 {
21     ArraySet testSet;
22     testSet.insert(5);
23     std::cout << std::boolalpha << testSet.search(15) << "\n";
24     std::cout << std::boolalpha << testSet.search(5);
25     return 0;
26 }
```

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

What if we use an Array: Exploiting constant access?

- Let's say we want to insert 11, 2 and 5 into a set
- Initially all values are false
- When we insert an item, we set the value at index to true
- **Common operations**
 - Insert:
 - Find:

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

What if we use an Array: Exploiting constant access?

- Let's say we want to insert 11, 2 and 5 into a set
- Initially all values are false
- When we insert an item, we set the value at index to true
- **Common operations**
 - Insert: $O(1)$
 - Find: $O(1)$

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

What if we use an Array: Exploiting constant access?

- Let's say we want to insert 11, 2 and 5 into a set
- Initially all values are false
- When we insert an item, we set the value at index to true
- Common operations
 - Insert: $O(1)$
 - Find: $O(1)$
- Problems with this: wastes memory and other datatypes?

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

How to deal with Strings?

- **Problems with this:** wastes memory and **other datatypes?**
- **What if we want to store:** “cat” or “dog”?

How to deal with Strings?

- **Problems with this:** wastes memory and **other datatypes?**
- **What if we want to store: “cat” or “dog”?**
 - **Idea:** Convert “cat” or “dog” into a number
 - **Approach:** Use the first letter – ‘c’ = 3, ‘d’ = 4

1	F
2	F
3	T
4	T
5	F
6	F
7	F
8	F
9	F
10	F
11	F
12	F
.	.
.	.
26	.

How to deal with Strings?

- **Problems with this:** wastes memory and **other datatypes?**
- **What if we want to store: “cat” or “dog”?**
 - **Idea:** Convert “cat” or “dog” into a number
 - **Approach:** Use the first letter – ‘c’ = 3, ‘d’ = 4
 - **Problem:** What happens with “cap”?

1	F
2	F
3	T
4	T
5	F
6	F
7	F
8	F
9	F
10	F
11	F
12	F
.	.
.	.
26	.

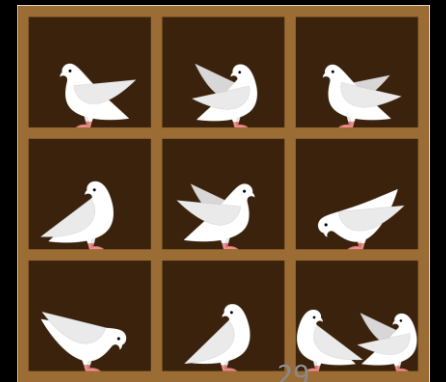
How to deal with Strings?

- **What if we want to store: “cat” or “dog”?**
 - **Idea:** Convert “cat” or “dog” into a number
 - **Approach:** Use the first letter – ‘c’ = 3, ‘d’ = 4
 - **Problem:** What happens with “cap”? – **“Collision”**
- **To fix this use all digits by multiplying each by a power of 27**
 - **Index of “cat” is $(3 \times 27^2) + (1 \times 27^1) + (20 \times 27^0) = 2234$.**

1	F
2	F
3	F
4	F
5	F
6	F
7	F
8	F
9	F
10	F
11	F
12	F
.	.
2234	T
.	.

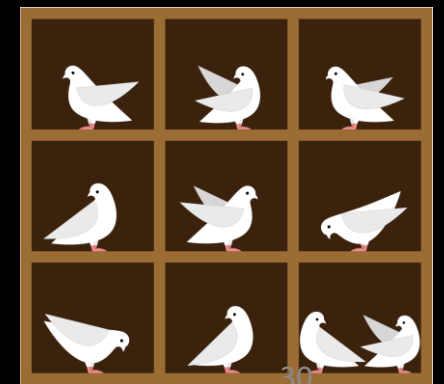
How to deal with Strings?

- To fix this use all digits by multiplying each by a power of 27
 - Index of “cat” is $(3 \times 27^2) + (1 \times 27^1) + (20 \times 27^0) = 2234$.
- As long as base ≥ 26 , we will get a unique number and no collisions. If it is less than 26, we are guaranteed for collisions due to pigeonhole principle



How to deal with Strings?

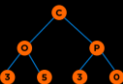
- To fix this use all digits by multiplying each by a power of 27
 - Index of “cat” is $(3 \times 27^2) + (1 \times 27^1) + (20 \times 27^0) = 2234$.
- As long as base ≥ 26 , we will get a unique number and no collisions. If it is less than 26, we are guaranteed for collisions due to pigeonhole principle
 - If base = 2, index of “ac” is $(1 \times 2^1) + (3 \times 2^0) = 5$
 - If base = 2, index of “e” is $(5 \times 2^0) = 5$
 - If base = 27, index of “ac” is $(1 \times 27^1) + (3 \times 27^0) = 30$
 - If base = 27, index of “e” is $(5 \times 27^0) = 5$



How to deal with Strings – ASCII and Unicode?

- **Increase the base for other characters as 26 characters is too restrictive**
 - **ASCII: 128 characters**
 - **Unicode: 143,859 characters**

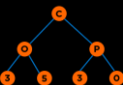
1	F
2	F
3	T
4	T
5	F
6	F
7	F
8	F
9	F
10	F
11	F
12	F
.	.
.	.
26	.



How to deal with Strings – ASCII and Unicode?

- Increase the base for other characters as 26 characters is too restrictive
 - **ASCII: 128 characters**
 - **Unicode: 143,859 characters**
- Fixed the problem of storing other datatypes
- **Problem: ?**

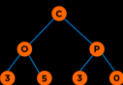
1	F
2	F
3	T
4	T
5	F
6	F
7	F
8	F
9	F
10	F
11	F
12	F
.	.
.	.
26	.



How to deal with Strings – ASCII and Unicode?

- Increase the base for other characters as 26 characters is too restrictive
 - **ASCII: 128 characters**
 - **Unicode: 143,859 characters**
- Fixed the problem of storing other datatypes
- **Problem: How do we store large values? Overflows, lead to collisions again. And we are now wasting even more space.**

1	F
2	F
3	T
4	T
5	F
6	F
7	F
8	F
9	F
10	F
11	F
12	F
.	.
.	.
26	.



Crux of the Problem

Approach

- Data -> Hash Function -> Hash Code

Hash code values for different data map to same index in array even after increasing a lot of space in table:

- “cat” -> transform2(“cat”) -> 34

1. poor hash functions

- “cat” -> transform127(“cat”) -> 48534

2. limitations of language

- “cat” -> transform143859(“cat”) -> 62,086,379,522
-> 1956837378

Collisions are Inevitable due to overflows!

Crux of the Problem

- **Problem**

- Wastes memory if we have hash tables that are large
- Has collisions – based on language limitations or poor hash functions

- **Solution**

- allow collisions
- use collision resolution strategies
- use small table sizes initially and increase it as per need when performance is affected

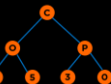
Hash Tables

Approach

- Data -> Hash Function -> Hash Code -> Reduce -> Index
- Insert the data (**D**) at the index in the table and if there is some other data at the index which is not **D**, then there is a collision and use a collision resolution mechanism

Hash Function

- A function that converts a data object to a hash code.
- Properties
 - **Input:** Object x
 - **Output:** An integer representation of x
 - If x is equal to y , $H(x) = H(y)$
 - If x is not equal to y , it would be great if $H(x)$ is not equal to $H(y)$



Hash Function Examples

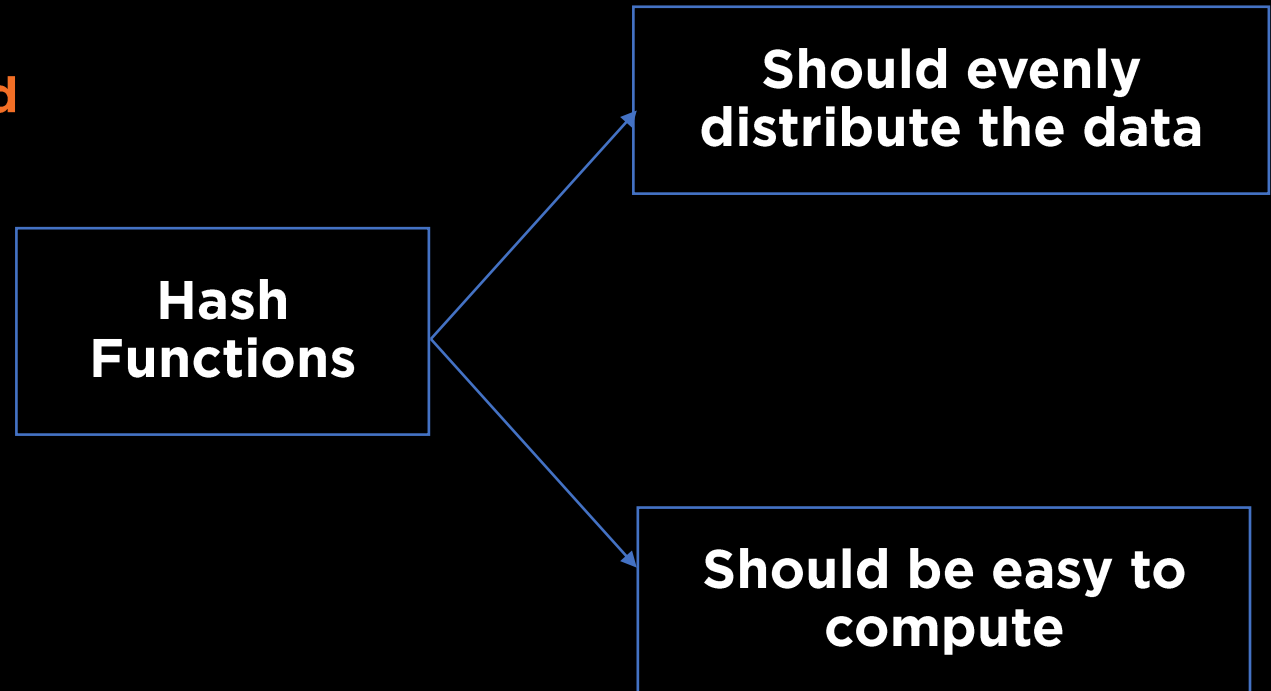
- A function, $H()$ that converts a data object, x to a hash code.
 - $H(x)$: { return 0; }
 - $H(x)$: { return Sum of all ASCII values; }
 - $H(x)$: { return Powers of 31 with ASCII; }
 - $H(x)$: { return Random Number; }
 - $H(x)$: { return Current Time; }

Hash Function Examples

- A function, $H()$ that converts a data object, x to a hash code.
 - Poor - $H(x): \{ \text{return } 0; \}$
 - Ok - $H(x): \{ \text{return Sum of all ASCII values; } \}$
 - Good - $H(x): \{ \text{return Powers of 31 with ASCII; } \}$
 - Invalid - ~~$H(x): \{ \text{return Random Number; } \}$~~
 - Invalid - ~~$H(x): \{ \text{return Current Time; } \}$~~

Hash Function Examples

- A function, $H()$ that converts a data object, x to a hash code.
 - $H(x)$: { return Powers of 31 with ASCII; }
 - Primes are usually used over composites
 - Smaller primes are preferred for faster calculations



Collision Resolution

- **Buckets and Load Factor**
- **Separate Chaining**
 - Open Hashing
 - Fixed
 - Resizable
- **Open Addressing**
 - Closed Hashing
 - Linear Probing
 - Quadratic Probing

Collision Resolution: Terms

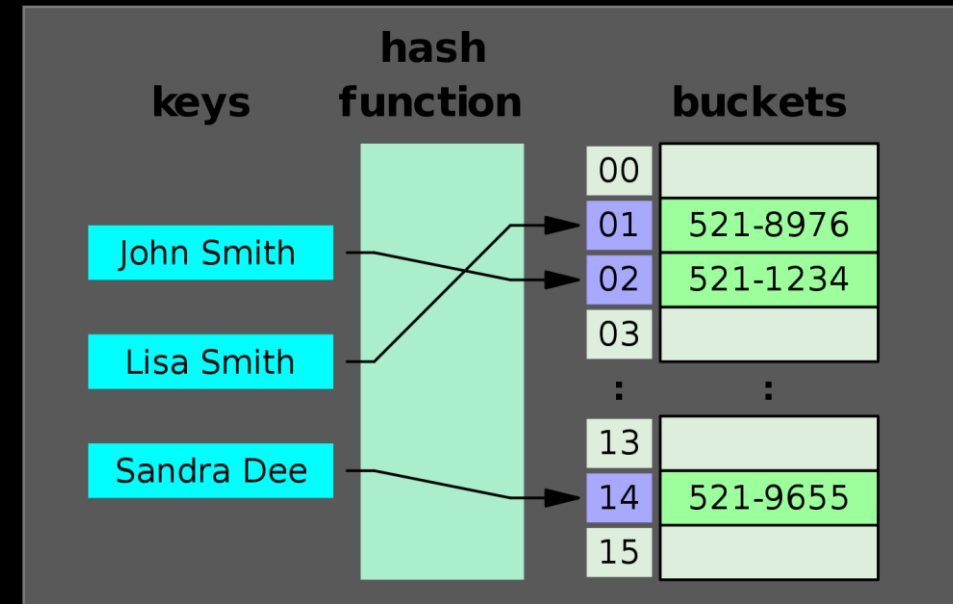
- **Buckets**

Total slots in the Hash Table structure

- **Load Factor**

$$\text{Load Factor}(\alpha) = \frac{\text{Total number of entries in the Hash Table}}{\text{Number of buckets}}$$

If load factor increases a certain threshold, then move to a larger table using rehashed values

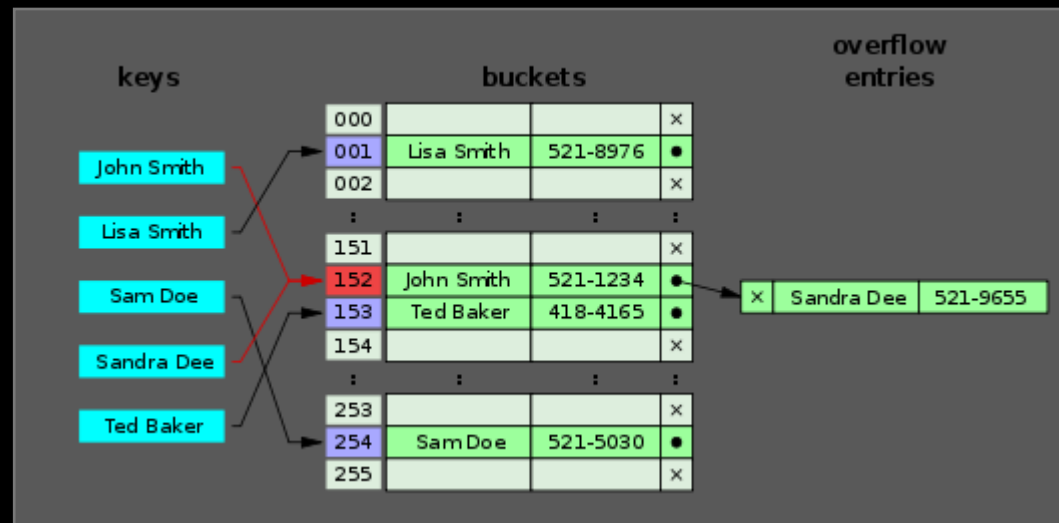


https://en.wikipedia.org/wiki/Hash_table

Collision Resolution: Separate Chaining

Data -> Hash Function -> Hash Code -> Reduce -> Index

Key Idea: buckets store a linked list; collisions are appended to the list

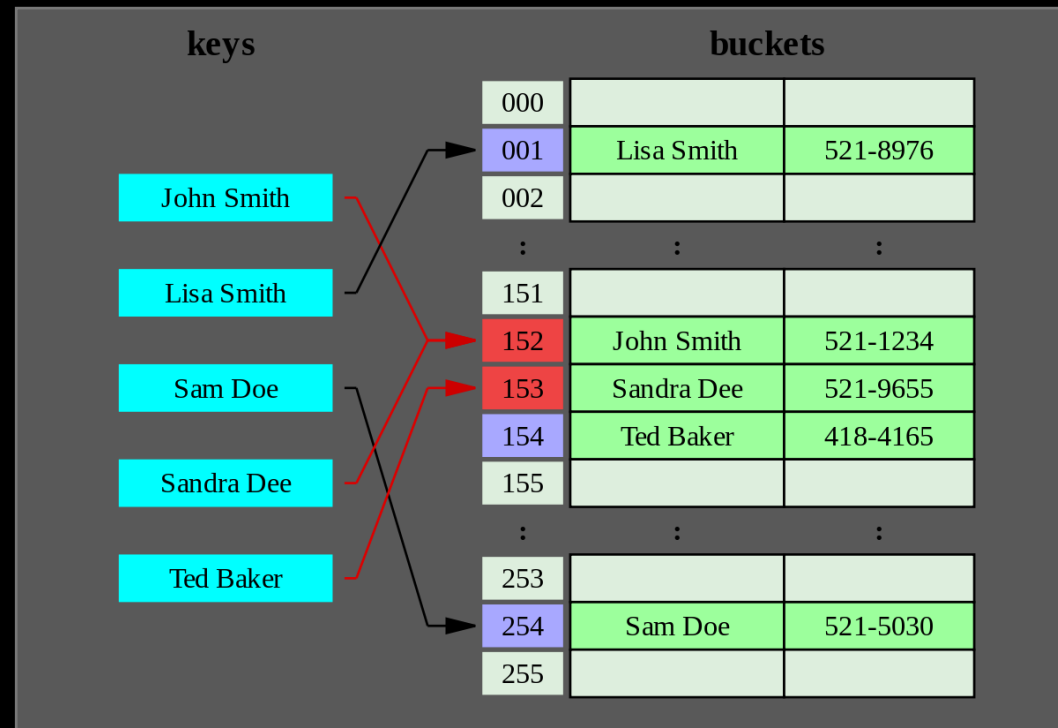


https://en.wikipedia.org/wiki/Hash_table

Collision Resolution: Open Addressing (Linear Probing)

Data -> Hash Function -> Hash Code -> Reduce -> Index

Key Idea: all entries in a bucket; collisions are added to empty spots

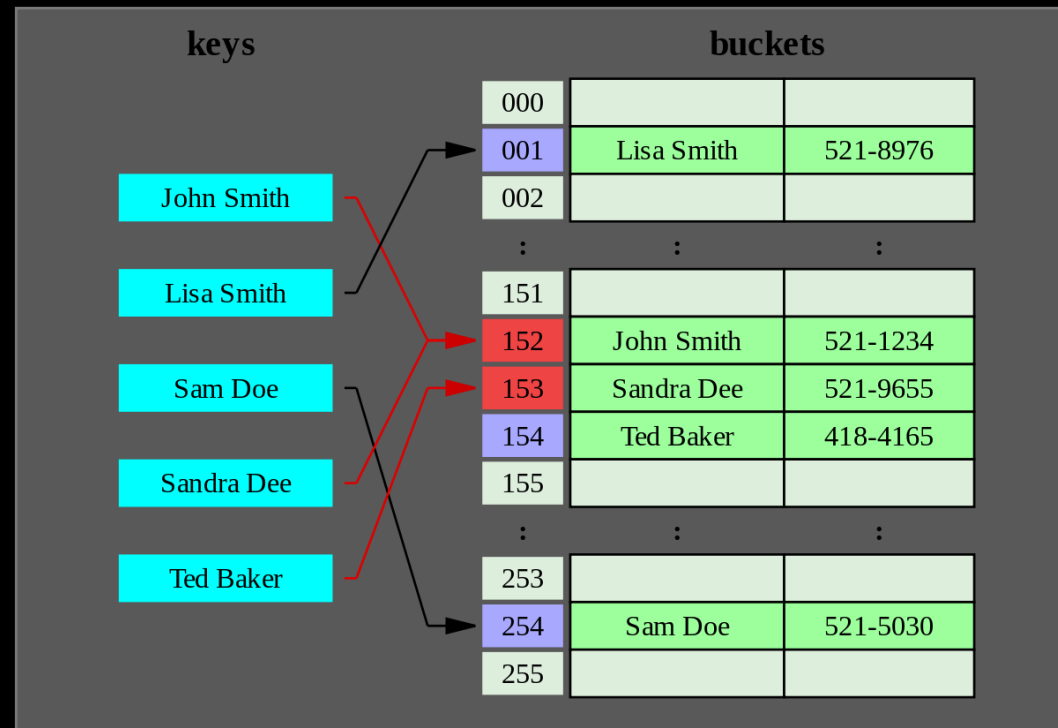


Move the probe by 1 unit

Collision Resolution: Open Addressing (Quadratic Probing)

Data -> Hash Function -> Hash Code -> Reduce -> Index

Key Idea: all entries in a bucket; collisions are added to empty spots



Move the probe by 1, 4, 9, 16 ... units

Hash Table Example: Separate Chaining

Data -> Hash Function -> Hash Code -> Reduce -> Index

$H(\text{key}) = \text{Sum of key's ASCII characters}$

$R(\text{Hashcode}) = \text{Hashcode} \% \text{TABLE_SIZE}$

0	
1	
2	
3	
4	

Hash Table Example: Separate Chaining

Data -> Hash Function -> Hash Code -> Reduce -> Index

$H(\text{key}) = \text{Sum of key's ASCII characters}$

$R(\text{Hashcode}) = \text{Hashcode} \% \text{TABLE_SIZE}$

Insert a.

0	
1	
2	
3	
4	

Hash Table Example: Separate Chaining

Data -> Hash Function -> Hash Code -> Reduce -> Index

$H(\text{key}) = \text{Sum of key's ASCII characters}$

$R(\text{Hashcode}) = \text{Hashcode} \% \text{TABLE_SIZE}$

Insert a.

$H(a) = 97$

$\text{Index} = H(a) \% 5 = 2$

0	
1	
2	
3	
4	

Hash Table Example: Separate Chaining

Data -> Hash Function -> Hash Code -> Reduce -> Index

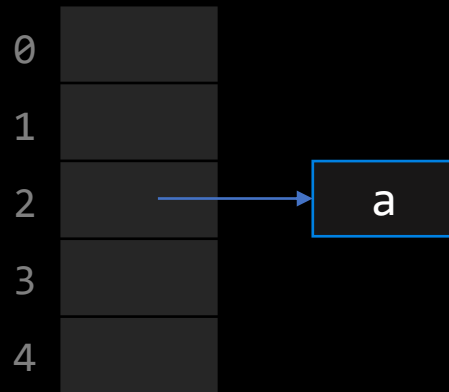
$H(\text{key}) = \text{Sum of key's ASCII characters}$

$R(\text{Hashcode}) = \text{Hashcode} \% \text{TABLE_SIZE}$

Insert a.

$H(a) = 97$

$\text{Index} = H(a) \% 5 = 2$



Load Factor = 0.2

Maximum Load Factor = 0.8

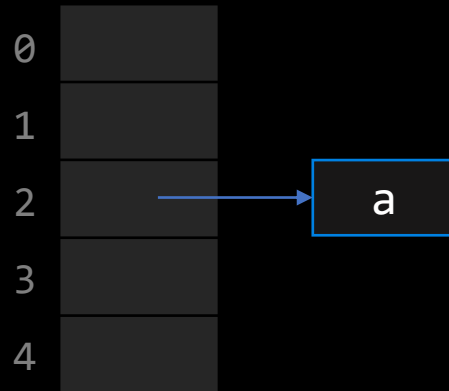
Hash Table Example: Separate Chaining

Data -> Hash Function -> Hash Code -> Reduce -> Index

$H(\text{key}) = \text{Sum of key's ASCII characters}$

$R(\text{Hashcode}) = \text{Hashcode} \% \text{TABLE_SIZE}$

Insert ac.



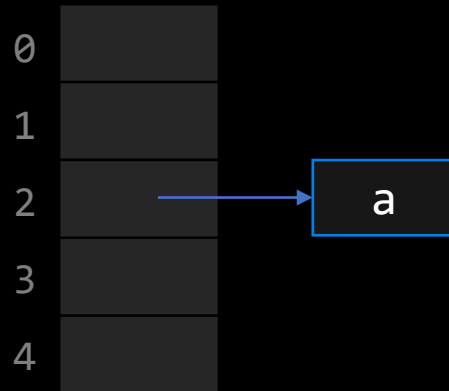
Hash Table Example: Separate Chaining

Data -> Hash Function -> Hash Code -> Reduce -> Index

$H(\text{key}) = \text{Sum of key's ASCII characters}$

$R(\text{Hashcode}) = \text{Hashcode} \% \text{TABLE_SIZE}$

Insert ac. $H(\text{ac}) = 97 + 99 = 196$
 $\text{Index} = H(\text{ac}) \% 5 = 1$



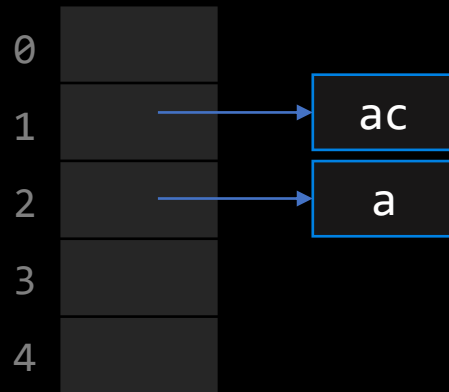
Hash Table Example: Separate Chaining

Data -> Hash Function -> Hash Code -> Reduce -> Index

$H(\text{key}) = \text{Sum of key's ASCII characters}$

$R(\text{Hashcode}) = \text{Hashcode} \% \text{TABLE_SIZE}$

Insert ac. $H(\text{ac}) = 97 + 99 = 196$
 $\text{Index} = H(\text{ac}) \% 5 = 1$



Load Factor = 0.4

Maximum Load Factor = 0.8

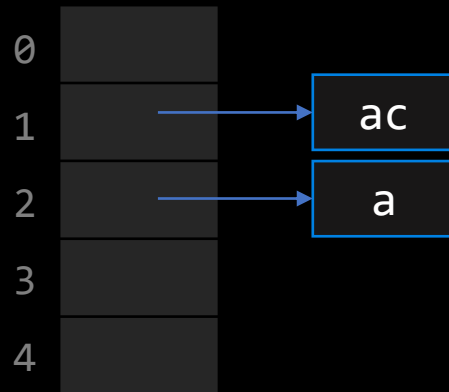
Hash Table Example: Separate Chaining

Data -> Hash Function -> Hash Code -> Reduce -> Index

$H(\text{key}) = \text{Sum of key's ASCII characters}$

$R(\text{Hashcode}) = \text{Hashcode} \% \text{TABLE_SIZE}$

Insert f.



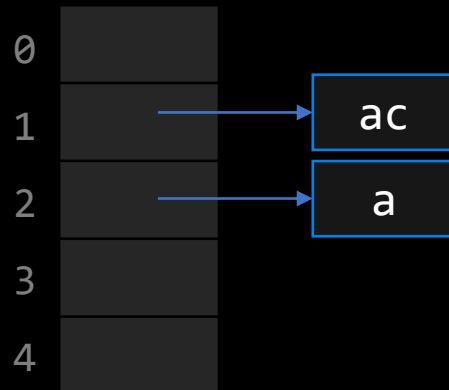
Hash Table Example: Separate Chaining

Data -> Hash Function -> Hash Code -> Reduce -> Index

$H(\text{key}) = \text{Sum of key's ASCII characters}$

$R(\text{Hashcode}) = \text{Hashcode} \% \text{TABLE_SIZE}$

Insert f. $H(f) = 101$
 $\text{Index} = H(f) \% 5 = 1$



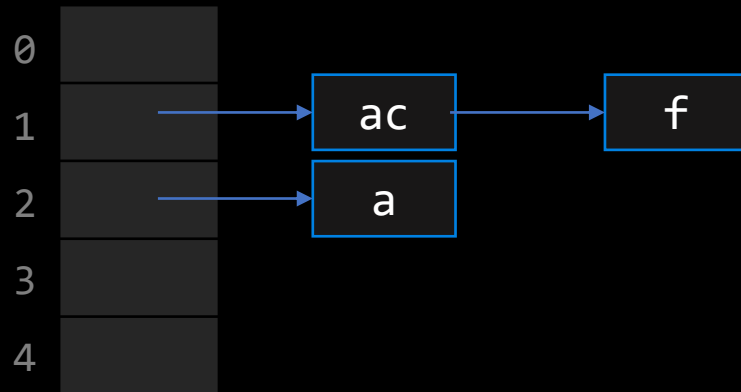
Hash Table Example: Separate Chaining

Data -> Hash Function -> Hash Code -> Reduce -> Index

$H(\text{key}) = \text{Sum of key's ASCII characters}$

$R(\text{Hashcode}) = \text{Hashcode} \% \text{TABLE_SIZE}$

Insert f. $H(f) = 101$
 $\text{Index} = H(f) \% 5 = 1$



Load Factor = 0.6

Maximum Load Factor = 0.8

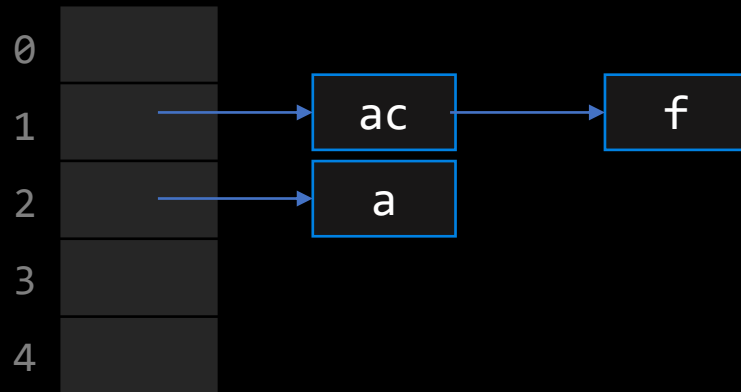
Hash Table Example: Separate Chaining

Data -> Hash Function -> Hash Code -> Reduce -> Index

$H(\text{key}) = \text{Sum of key's ASCII characters}$

$R(\text{Hashcode}) = \text{Hashcode} \% \text{TABLE_SIZE}$

Search f.



Hash Table Example: Separate Chaining

Data -> Hash Function -> Hash Code -> Reduce -> Index

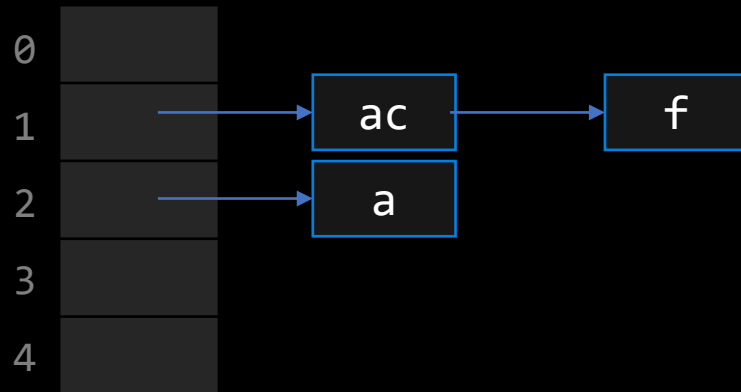
$H(\text{key}) = \text{Sum of key's ASCII characters}$

$R(\text{Hashcode}) = \text{Hashcode} \% \text{TABLE_SIZE}$

Search f.

$H(f) = 101$

$\text{Index} = H(f) \% 5 = 1$



Hash Table Example: Separate Chaining

Data -> Hash Function -> Hash Code -> Reduce -> Index

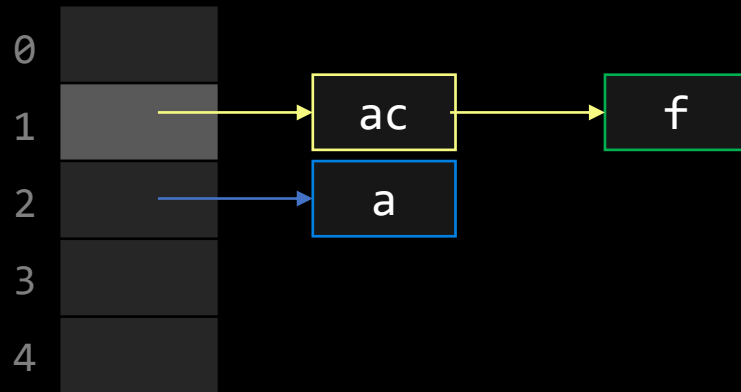
$H(\text{key}) = \text{Sum of key's ASCII characters}$

$R(\text{Hashcode}) = \text{Hashcode} \% \text{TABLE_SIZE}$

Search f.

$H(f) = 101$

$\text{Index} = H(f) \% 5 = 1$



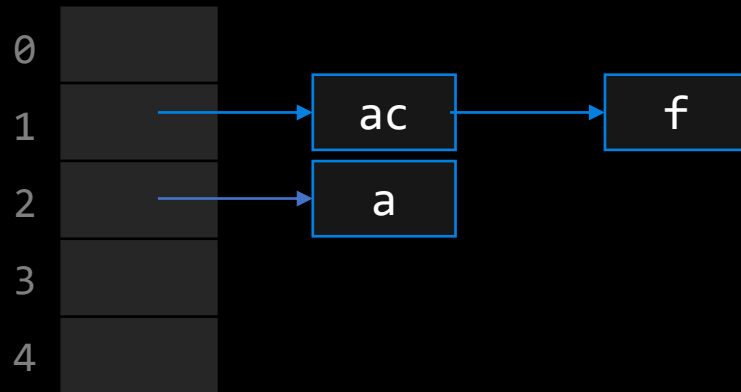
Hash Table Example: Separate Chaining

Data -> Hash Function -> Hash Code -> Reduce -> Index

$H(\text{key}) = \text{Sum of key's ASCII characters}$

$R(\text{Hashcode}) = \text{Hashcode} \% \text{TABLE_SIZE}$

Insert cat.



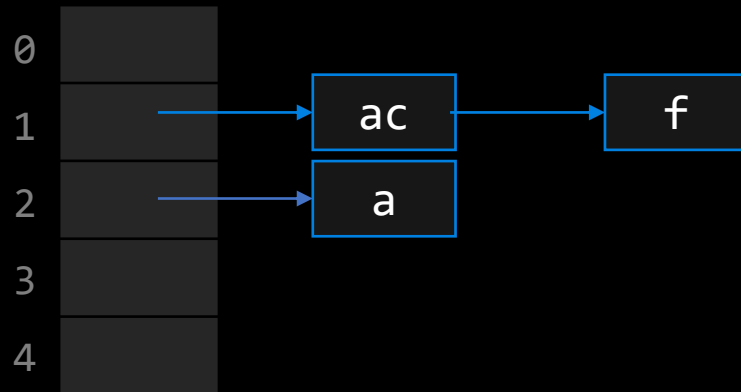
Hash Table Example: Separate Chaining

Data -> Hash Function -> Hash Code -> Reduce -> Index

$H(\text{key}) = \text{Sum of key's ASCII characters}$

$R(\text{Hashcode}) = \text{Hashcode} \% \text{TABLE_SIZE}$

Insert cat. $H(\text{cat}) = 99 + 97 + 116 = 312$
 $\text{Index} = H(\text{cat}) \% 5 = 2$



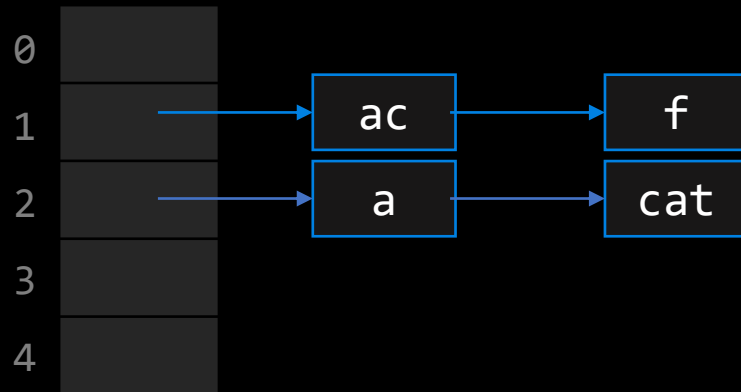
Hash Table Example: Separate Chaining

Data -> Hash Function -> Hash Code -> Reduce -> Index

$H(\text{key}) = \text{Sum of key's ASCII characters}$

$R(\text{Hashcode}) = \text{Hashcode} \% \text{TABLE_SIZE}$

Insert cat. $H(\text{cat}) = 99 + 97 + 116 = 312$
 $\text{Index} = H(\text{cat}) \% 5 = 2$



Load Factor = 0.8

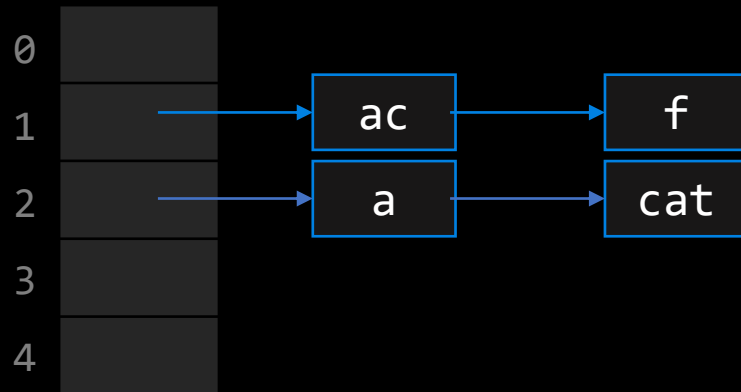
Maximum Load Factor = 0.8

Hash Table Example: Separate Chaining

Data -> Hash Function -> Hash Code -> Reduce -> Index

$H(\text{key}) = \text{Sum of key's ASCII characters}$

$R(\text{Hashcode}) = \text{Hashcode} \% \text{TABLE_SIZE}$



Load Factor = 0.8

Maximum Load Factor = 0.8

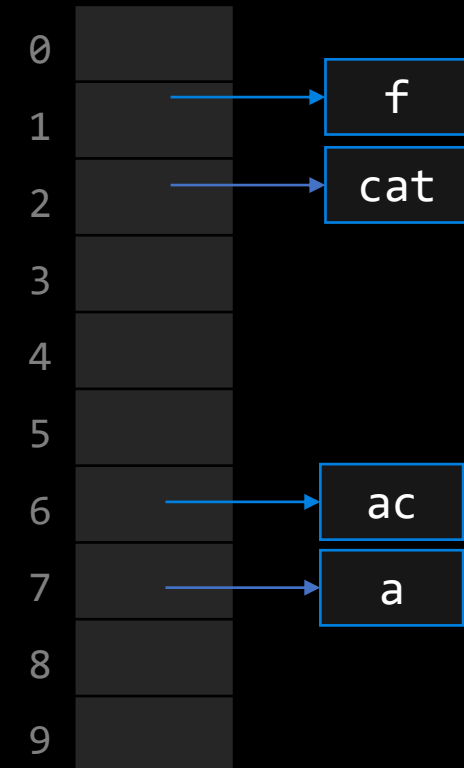
Rehashing

$H(\text{ac}) = 196, \text{Index} = 196 \% 10 = 6$

$H(\text{f}) = 101, \text{Index} = 101 \% 10 = 1$

$H(\text{a}) = 97, \text{Index} = 97 \% 10 = 7$

$H(\text{cat}) = 312, \text{Index} = 312 \% 10 = 2$

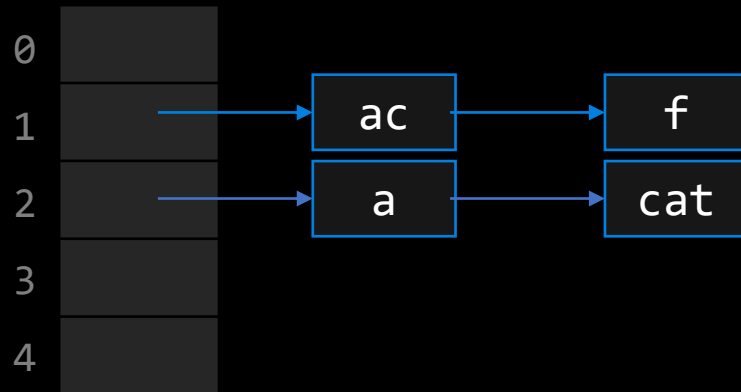


Hash Table Example: Separate Chaining

Data -> Hash Function -> Hash Code -> Reduce -> Index

$H(\text{key}) = \text{Sum of key's ASCII characters}$

$R(\text{Hashcode}) = \text{Hashcode} \% \text{TABLE_SIZE}$



Load Factor = 0.8

Maximum Load Factor = 0.8

Rehashing

$H(\text{ac}) = 196, \text{Index} = 196 \% 10 = 6$

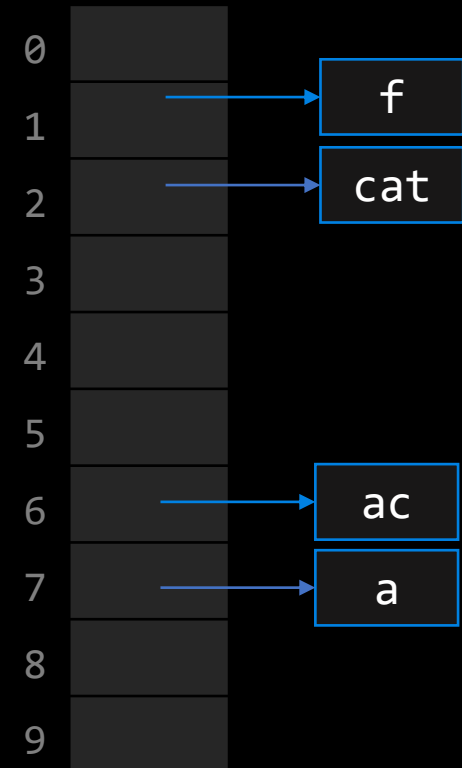
$H(\text{f}) = 101, \text{Index} = 101 \% 10 = 1$

$H(\text{a}) = 97, \text{Index} = 97 \% 10 = 7$

$H(\text{cat}) = 312, \text{Index} = 312 \% 10 = 2$

Load Factor = 0.4

Maximum Load Factor = 0.8



Hash Table Example: Open addressing with Linear Probing

Data -> Hash Function -> Hash Code -> Reduce -> Index

$H(\text{key}) = \text{Sum of key's ASCII characters}$

$R(\text{Hashcode}) = \text{Hashcode} \% \text{TABLE_SIZE}$

Open addressing/Closed Hashing: Index is not determined by hash code, i.e., index is open

0	
1	
2	
3	
4	

Hash Table Example: Open addressing with Linear Probing

Data -> Hash Function -> Hash Code -> Reduce -> Index

$H(\text{key}) = \text{Sum of key's ASCII characters}$

$R(\text{Hashcode}) = \text{Hashcode} \% \text{TABLE_SIZE}$

Insert:

$H(\text{ac}) = 196, \text{Index} = 196 \% 5 = 1$

0	
1	
2	
3	
4	

Hash Table Example: Open addressing with Linear Probing

Data -> Hash Function -> Hash Code -> Reduce -> Index

$H(\text{key}) = \text{Sum of key's ASCII characters}$

$R(\text{Hashcode}) = \text{Hashcode} \% \text{TABLE_SIZE}$

Insert:

$H(\text{ac}) = 196, \text{Index} = 196 \% 5 = 1$

0	
1	ac
2	
3	
4	

Hash Table Example: Open addressing with Linear Probing

Data -> Hash Function -> Hash Code -> Reduce -> Index

$H(\text{key}) = \text{Sum of key's ASCII characters}$

$R(\text{Hashcode}) = \text{Hashcode} \% \text{TABLE_SIZE}$

Insert:

$H(\text{ac}) = 196, \text{Index} = 196 \% 5 = 1$

0	
1	ac
2	
3	
4	

Hash Table Example: Open addressing with Linear Probing

Data -> Hash Function -> Hash Code -> Reduce -> Index

$H(\text{key}) = \text{Sum of key's ASCII characters}$

$R(\text{Hashcode}) = \text{Hashcode} \% \text{TABLE_SIZE}$

Insert:

$H(\text{ac}) = 196, \text{Index} = 196 \% 5 = 1$

$H(\text{f}) = 101, \text{Index} = 101 \% 5 = 1$

0	
1	ac
2	
3	
4	

Hash Table Example: Open addressing with Linear Probing

Data -> Hash Function -> Hash Code -> Reduce -> Index

$H(\text{key}) = \text{Sum of key's ASCII characters}$

$R(\text{Hashcode}) = \text{Hashcode} \% \text{TABLE_SIZE}$

Insert:

$H(\text{ac}) = 196, \text{Index} = 196 \% 5 = 1$

$H(\text{f}) = 101, \text{Index} = 101 \% 5 = 1$

0	
1	ac
2	f
3	
4	

Hash Table Example: Open addressing with Linear Probing

Data -> Hash Function -> Hash Code -> Reduce -> Index

$H(\text{key}) = \text{Sum of key's ASCII characters}$

$R(\text{Hashcode}) = \text{Hashcode} \% \text{TABLE_SIZE}$

Insert:

$H(\text{ac}) = 196, \text{Index} = 196 \% 5 = 1$

$H(\text{f}) = 101, \text{Index} = 101 \% 5 = 1$

$H(\text{a}) = 97, \text{Index} = 97 \% 5 = 2$

0	
1	ac
2	f
3	
4	

Hash Table Example: Open addressing with Linear Probing

Data -> Hash Function -> Hash Code -> Reduce -> Index

$H(\text{key}) = \text{Sum of key's ASCII characters}$

$R(\text{Hashcode}) = \text{Hashcode} \% \text{TABLE_SIZE}$

Insert:

$H(\text{ac}) = 196, \text{Index} = 196 \% 5 = 1$

$H(\text{f}) = 101, \text{Index} = 101 \% 5 = 1$

$H(\text{a}) = 97, \text{Index} = 97 \% 5 = 2$

0	
1	ac
2	f
3	a
4	

Hash Table Example: Open addressing with Linear Probing

Data -> Hash Function -> Hash Code -> Reduce -> Index

$H(\text{key}) = \text{Sum of key's character ASCII}$

$R(\text{Hashcode}) = \text{Hashcode} \% \text{TABLE_SIZE}$

Insert:

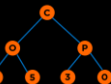
$H(\text{ac}) = 196, \text{Index} = 196 \% 5 = 1$

$H(\text{f}) = 101, \text{Index} = 101 \% 5 = 1$

$H(\text{a}) = 97, \text{Index} = 97 \% 5 = 2$

$H(\text{cat}) = 312, \text{Index} = 312 \% 5 = 2$

0	
1	ac
2	f
3	a
4	



Hash Table Example: Open addressing with Linear Probing

Data -> Hash Function -> Hash Code -> Reduce -> Index

$H(\text{key}) = \text{Sum of key's character ASCII}$

$R(\text{Hashcode}) = \text{Hashcode} \% \text{TABLE_SIZE}$

Insert:

$H(\text{ac}) = 196, \text{Index} = 196 \% 5 = 1$

$H(\text{f}) = 101, \text{Index} = 101 \% 5 = 1$

$H(\text{a}) = 97, \text{Index} = 97 \% 5 = 2$

$H(\text{cat}) = 312, \text{Index} = 312 \% 5 = 2$

0	
1	ac
2	f
3	a
4	cat

Hash Table Example: Open addressing with Linear Probing

Data -> Hash Function -> Hash Code -> Reduce -> Index

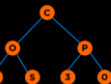
$H(\text{key}) = \text{Sum of key's character ASCII}$

$R(\text{Hashcode}) = \text{Hashcode} \% \text{TABLE_SIZE}$

Search ab:

$H(ab) = 195, \text{Index} = 196 \% 5 = 0$

0	
1	ac
2	f
3	a
4	cat



Hash Table Example: Open addressing with Linear Probing

Data -> Hash Function -> Hash Code -> Reduce -> Index

$H(\text{key}) = \text{Sum of key's character ASCII}$

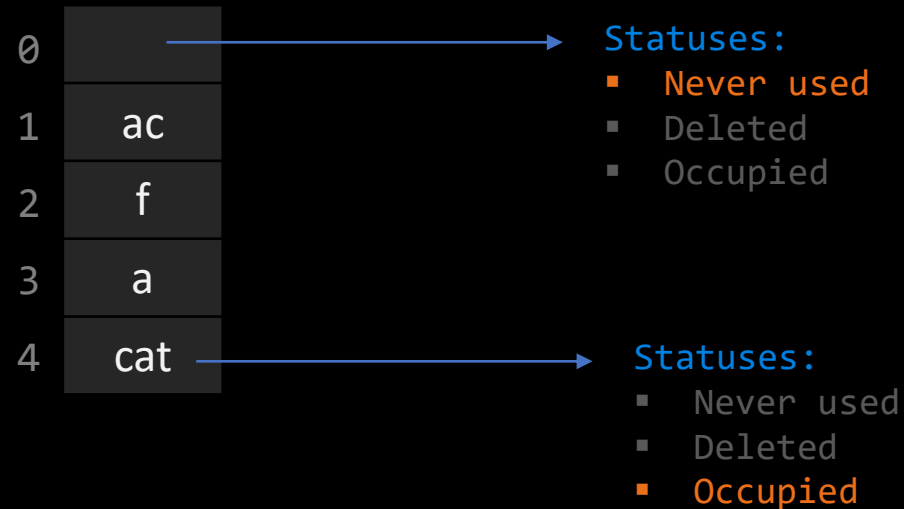
$R(\text{Hashcode}) = \text{Hashcode} \% \text{TABLE_SIZE}$

Search ab:

$H(\text{ab}) = 195$, $\text{Index} = 196 \% 5 = 0$

Look at bucket 0; if never occupied, then stop and return false;

If occupied, repeat till a bucket is available that is never used, or element is found.



Hash Table Example: Open addressing with Quadratic Probing

Data -> Hash Function -> Hash Code -> Reduce -> Index

$H(\text{key}) = \text{Sum of key's character ASCII}$

$R(\text{Hashcode}) = \text{Hashcode} \% \text{TABLE_SIZE}$

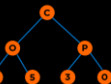
Same as linear probing but indexes are moved quadratically
e.g., 1, 4, 9, 16, 25 ... to avoid clusters in the hash table.

Insert:

$H(\text{ac}) = 196$, Index = $196 \% 5 = 1$

$H(\text{f}) = 101$, Index = $101 \% 5 = 1$

0	
1	ac
2	f
3	
4	



Hash Table Example: Open addressing with Quadratic Probing

Data -> Hash Function -> Hash Code -> Reduce -> Index

$H(\text{key}) = \text{Sum of key's character ASCII}$

$R(\text{Hashcode}) = \text{Hashcode} \% \text{TABLE_SIZE}$

Same as linear probing but indexes are moved quadratically
e.g., 1, 4, 9, 16, 25 ... to avoid clusters in the hash table.

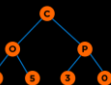
Insert:

$H(\text{ac}) = 196, \text{Index} = 196 \% 5 = 1$

$H(\text{f}) = 101, \text{Index} = 101 \% 5 = 1$

$H(\text{k}) = 106, \text{Index} = 106 \% 5 = 1$

0	
1	ac
2	f
3	
4	



Hash Table Example: Open addressing with Quadratic Probing

Data -> Hash Function -> Hash Code -> Reduce -> Index

$H(\text{key}) = \text{Sum of key's character ASCII}$

$R(\text{Hashcode}) = \text{Hashcode} \% \text{TABLE_SIZE}$

Same as linear probing but indexes are moved quadratically
e.g., 1, 4, 9, 16, 25 ... to avoid clusters in the hash table.

Insert:

$H(\text{ac}) = 196$, Index = $196 \% 5 = 1$

$H(\text{f}) = 101$, Index = $101 \% 5 = 1$

$H(\text{k}) = 106$, Index = $106 \% 5 = 1$

Move 1 (conflict 2 - f),

Move 4 (conflict 1 - ac),

Move 9 (open 0 - k).

0	k
1	ac
2	f
3	
4	

Hash Table Performance

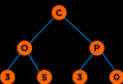
- Time complexities of Search/Insert/Delete are $O(1)$ on average
- Time complexities of Search/Insert/Delete are $O(n)$ in the worst case



Hash Tables and Map vs Set

```
01 class Set
02 {
03     private:
04         string arraySet[100];
05     public:
06         void insert(int value);
07         bool search(int value);
08 };
09
10 void ArraySet::insert(int value)
11 {
12     //find the hash of the value
13     //reduce the hash to get an index
14     //check if value is not at index
15         //insert value at index
16     //otherwise, use collision resolution strategy
17 }
18
19 bool ArraySet::search(int value)
20 {
21     //find the hash of the value
22     //reduce the hash to get an index
23     //check if value is not at index
24         //return false
25     //otherwise, search based on collision resolution strategy
26 }
```

Remember C++ Unordered Maps and Sets are backed by Hash Tables



Sets and Maps in C++ Example

```
01 //Unordered Set - Hash-based
02 unordered_set<int> s2;
03
04 // insert elements in random order
05 s2.insert(5);
06 s2.insert(2);
07 s2.insert(4);
08 s2.insert(11);
09 s2.insert(2); // only one 2 will be added to the set
10
11 // printing set
12 unordered_set<int>::iterator itr2;
13 cout << "The set s2 is:";
14 for (itr2 = s2.begin(); itr2 != s2.end(); ++itr2)
15     cout << " " << *itr2;
16 cout << endl;
17 cout << "Bucket count: " << s2.bucket_count();
18 cout << "\nLoad Factor: " << s2.load_factor();
19 cout << "\nMax Load Factor:" << s2.max_load_factor();
```

```
The set s2 is: 11 4 5 2
Bucket count: 7
Load Factor: 0.571429
Max Load Factor: 1
```

```
01 //Unordered Map - Hash-based
02 unordered_map<char,int> table_unordered;
03
04 // insert elements in random order
05 table_unordered['b']=30;
06 table_unordered['a']=10;
07 table_unordered['c']=50;
08 table_unordered['a']=40;
09
10 // printing set
11 for(auto member: table_unordered)
12     cout << member.first << " " << member.second << "\n";
13
14 cout << "Load Factor: " << table_unordered.load_factor();
```

```
c 50
b 30
a 40
Load Factor: 0.428571
```

<https://onlinegdb.com/SkHykUnlP>

<https://onlinegdb.com/SyMCuH0lD>

Questions

Mentimeter

Menti.com

4481 0013



Mentimeter

Menti.com

2056 3140



10.1.2 Two Sum Problem

N-sum is a common problem where you are given an array and asked to see if there are N numbers that add up to a target. For this stepik module, you'll be asked to complete Two-Sum. This means you'll be given an array of integers and you have to determine if there are 2 values that sum to a desired target. The method signature is `pair<int, int> two_sum(vector<int> arr, int target)`, which returns a pair of the indices whose values sum to the desired target. If no such 2 value exists, return the pair `{-1,-1}`. Make sure that the smaller index is first.

Example:

```
arr = [3, 5, 11, 12, 15]
```

```
target = 17
```

```
Output = {1,3}
```

10.1.2 Two Sum Problem

N-sum is a common problem where you are given an array and asked to see if there are N numbers that add up to a target. For this stepik module, you'll be asked to complete Two-Sum. This means you'll be given an array of integers and you have to determine if there are 2 values that sum to a desired target. The method signature is `pair<int, int> two_sum(vector<int> arr, int target)`, which returns a pair of the indices whose values sum to the desired target. If no such 2 value exists, return the pair `{-1,-1}`. Make sure that the smaller index is first.

Example:

```
arr = [3, 5, 11, 12, 15]
```

```
target = 17
```

```
Output = {1,3}
```

10.1.2 Two Sum Problem

N-sum is a common problem where you are given an array and asked to see if there are N numbers that add up to a target. For this stepik module, you'll be asked to complete Two-Sum. This means you'll be given an array of integers and you have to determine if there are 2 values that sum to a desired target. The method signature is `pair<int, int> two_sum(vector<int> arr, int target)`, which returns a pair of the indices whose values sum to the desired target. If no such 2 value exists, return the pair `{-1,-1}`. Make sure that the smaller index is first.

Example:

`arr = [3, 5, 11, 12, 15]`

`target = 17`

`Output = {1,3}`

```
01 pair<int, int> two_sum(vector<int>& arr, int target)
02 {
03     unordered_map<int, int> map;
04     pair<int, int> result(-1, -1);
05     for (int i = 0; i < arr.size(); i++)
06     {
07         int diff = target - arr[i];
08         if(map.count(diff))           //check if complement is present in the set
09         {
10             result.first = map[diff];
11             result.second = i;
12             break;
13         }
14         map[arr[i]] = i;              //add the element to the set otherwise
15     }
16     return result;
17 }
```