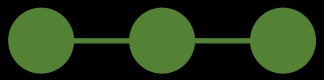
Sets, Maps and Hash Tables

Data Structures & Algorithm

Categories of Data Structures

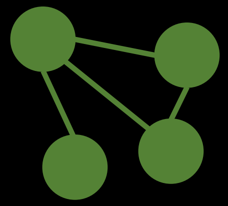
Linear Ordered

- Lists
- Stacks
- Queues



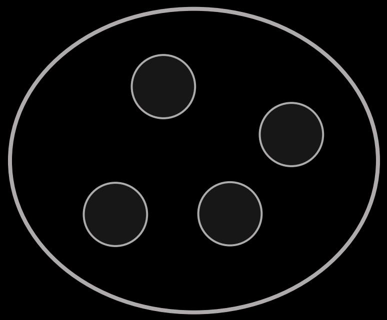
Non-linear Ordered

- Trees
- Graphs



Not Ordered

- Sets
- Tables/Maps



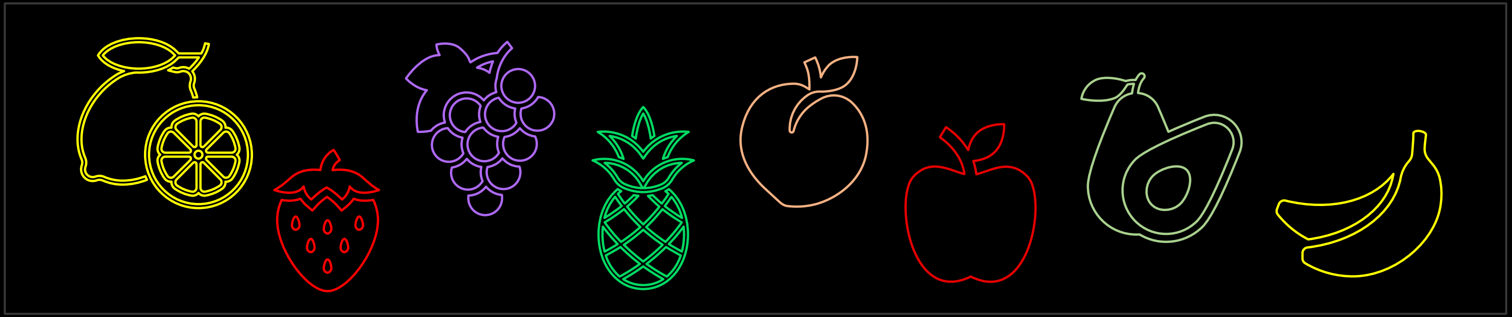
Sets

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



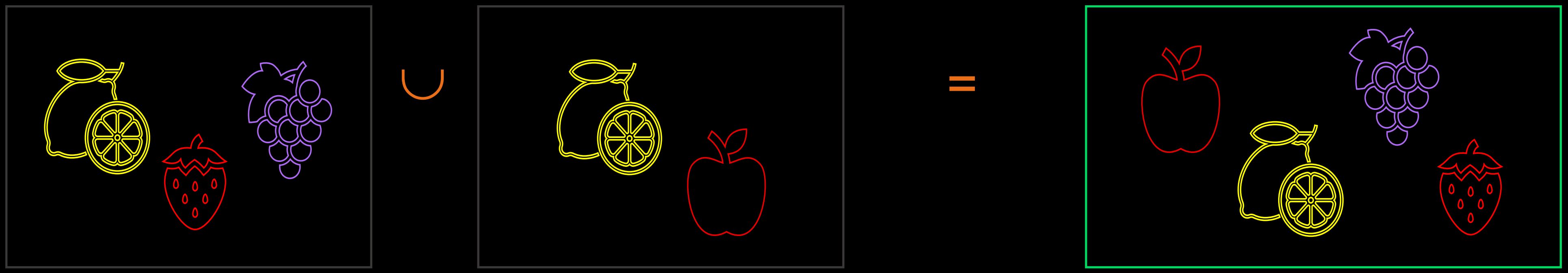
Sets

Set ADT Operations

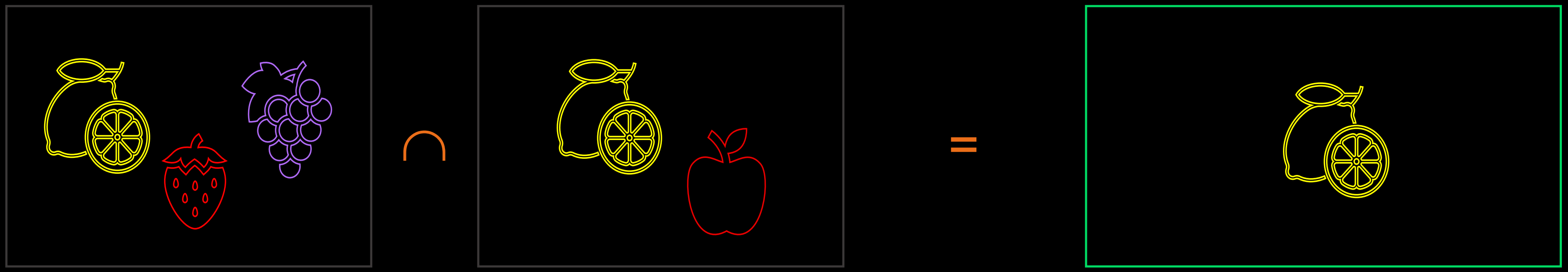
- Testing for membership (find), adding elements (insert), removing elements (remove)
- Union, intersection, difference with another set
- Returning a subset

Sets

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

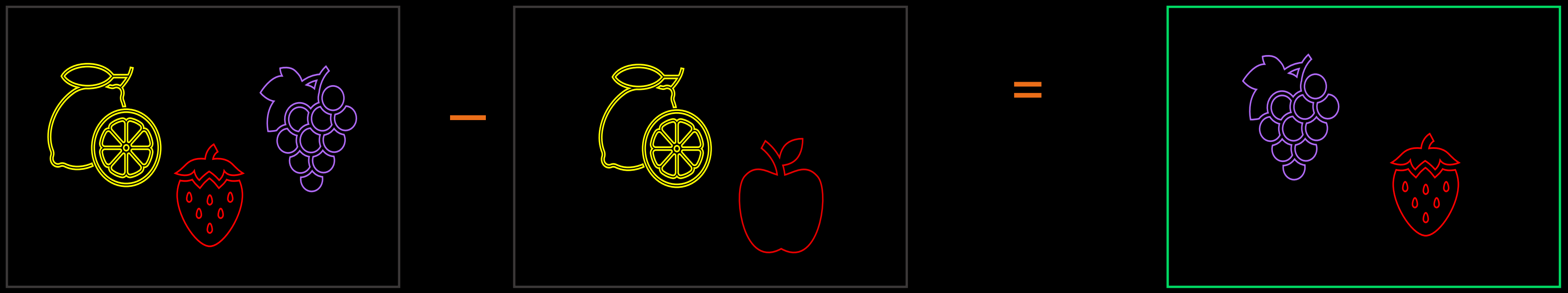


Intersection of sets A ∩ B is the set whose elements belong to both A and B.

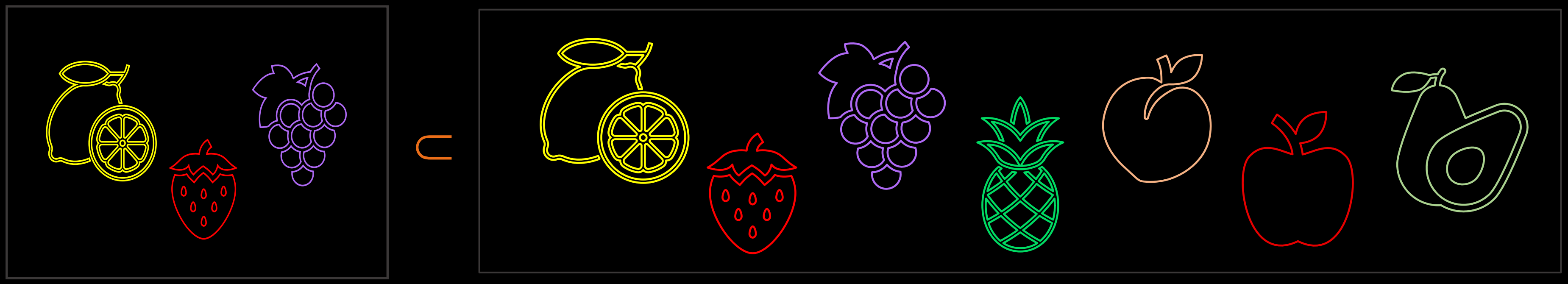


Sets

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 ⊂ 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++

	std::set	std::unordered_set
Order in Elements	Yes	No
Initialization	std::set<type> s;	std::unordered_set<type> s;
Common Methods	insert, erase, find, count, size, empty	insert, erase, find, count, size, empty, bucket_size, load_factor
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

```cpp
// Ordered tree-based set
set <int> s1;

// insert elements in random order
s1.insert(5);
s1.insert(2);
s1.insert(4);
s1.insert(11);
s1.insert(2); // only one 2 will be added to the set

// printing set
set <int> :: iterator itr1;
cout << "The set s1 is : ";
for (itr1 = s1.begin(); itr1 != s1.end(); ++itr1)
           cout << "  " << *itr1;
```

```
The set s1 is :   2  4  5  11
```

```cpp
//Unordered Set - Hash-based
unordered_set <int> s2;

// insert elements in random order
s2.insert(5);
s2.insert(2);
s2.insert(4);
s2.insert(11);
s2.insert(2); // only one 2 will be added to the set

// printing set
unordered_set <int> :: iterator itr2;
cout << "The set s2 is:";
for (itr2 = s2.begin(); itr2 != s2.end(); ++itr2)
           cout << "  " << *itr2;
cout << endl;
cout << "Bucket count: " << s2.bucket_count();
cout << "\nLoad Factor: " << s2.load_factor();
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
```

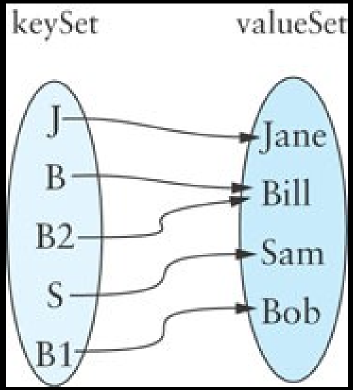
https://onlinegdb.com/SyMCuHOlD

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++

	std::map	std::unordered_map
Order in Elements	Yes	No
Initialization (Internally stored as pairs)	std::map<type, type> m;	std::unordered_map <type, type> m;
Common Methods	insert, [], erase, find, count, size, empty	insert, [], erase, find, count, size, empty, bucket_size, load_factor
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

```cpp
// Ordered tree-based map
map<char,int> table;

// insert elements in random order
table['b']=30;
table['a']=10;
table['c']=50;
table['a']=40;

// printing map
for(auto member: table)
      cout << member.first << " " << member.second <<"\n";
```

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

```cpp
//Unordered Map – Hash-based
unordered_map<char,int> table_unordered;

// insert elements in random order
table_unordered['b']=30;
table_unordered['a']=10;
table_unordered['c']=50;
table_unordered['a']=40;

// printing set
for(auto member: table_unordered)
      cout << member.first << " " << member.second <<"\n";

cout << "Load Factor: " << table_unordered.load_factor();
```

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

https://onlinegdb.com/SkHykUnlP

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?

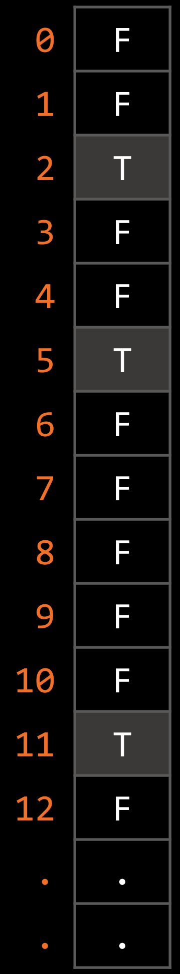
What if we use an Array: Exploiting constant access?

- 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



What if we use an Array: Exploiting constant access?

```cpp
class ArraySet
{
    private:
        bool set[100] = {0};
    public:
        void insert(int value);
        bool search(int value);
};

void ArraySet::insert(int value)
{
    set[value] = 1;
}

bool ArraySet::search(int value)
{
    return set[value];
}
```

```cpp
int main()
{
    ArraySet testSet;
    testSet.insert(5);
    std::cout << std::boolalpha << testSet.search(15) <<“\n”;
    std::cout << std::boolalpha << testSet.search(5);
    return 0;
}
```

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
.	.
.	.

https://onlinegdb.com/TFfwCnEEl

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	.

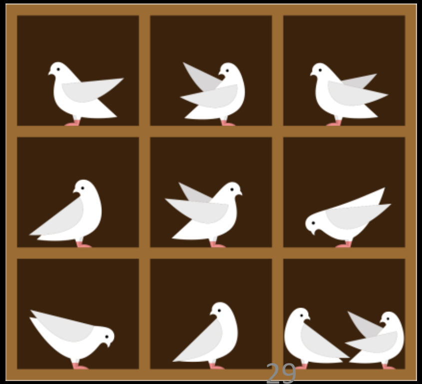
How to deal with Strings?

- 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
.	.

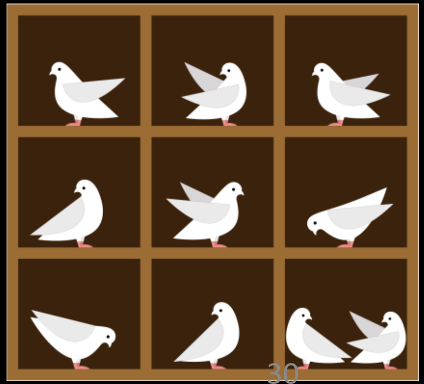
How to deal with Strings?

- 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 -> $\boxed{\text{Hash Function}}$ -> $\boxed{\text{Hash Code}}$

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
- "cat" -> transform127("cat") -> 48534
- "cat" -> transform143859("cat") -> 62,086,379,522 -> 1956837378

1. poor hash functions
2. limitations of language

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): { return 0; }
  - Ok - H(x): { return Sum of all ASCII values; }
  - Good - H(x): { return Powers of 31 with ASCII; }
  - Invalid - ~~H(x): { return Random Number; }~~
  - Invalid - ~~H(x): { 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

Hash Functions

→ Should evenly distribute the data

→ Should be easy to compute

Collision Resolution

- Buckets and Load Factor

- Separate Chaining
  - Open Hashing
  - Fixed
  - Resizable

- Open Addressing
  - Closed Hashing
  - Linear Probing
  - Quadratic Probing

Collision Resolution: Terms

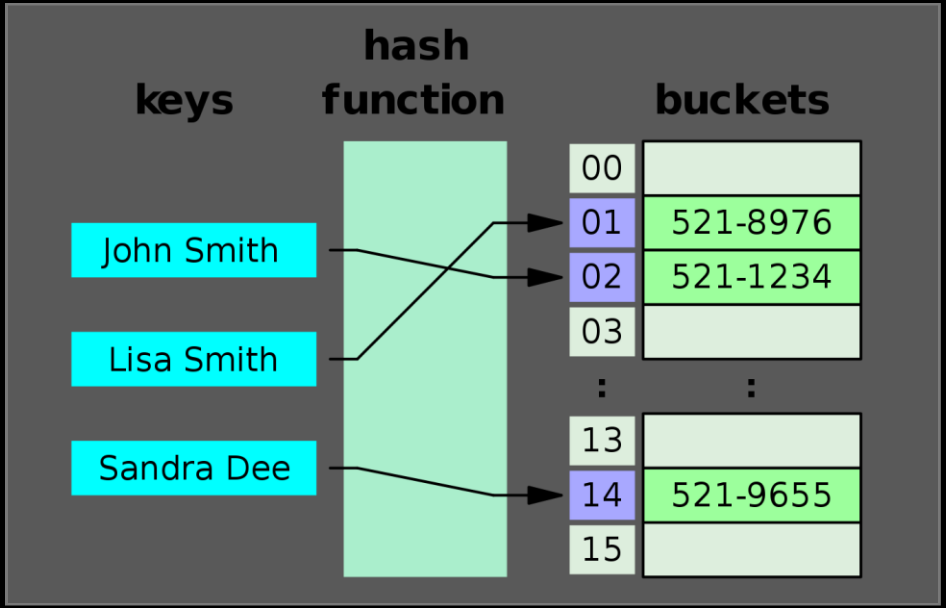
- **Buckets**

  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

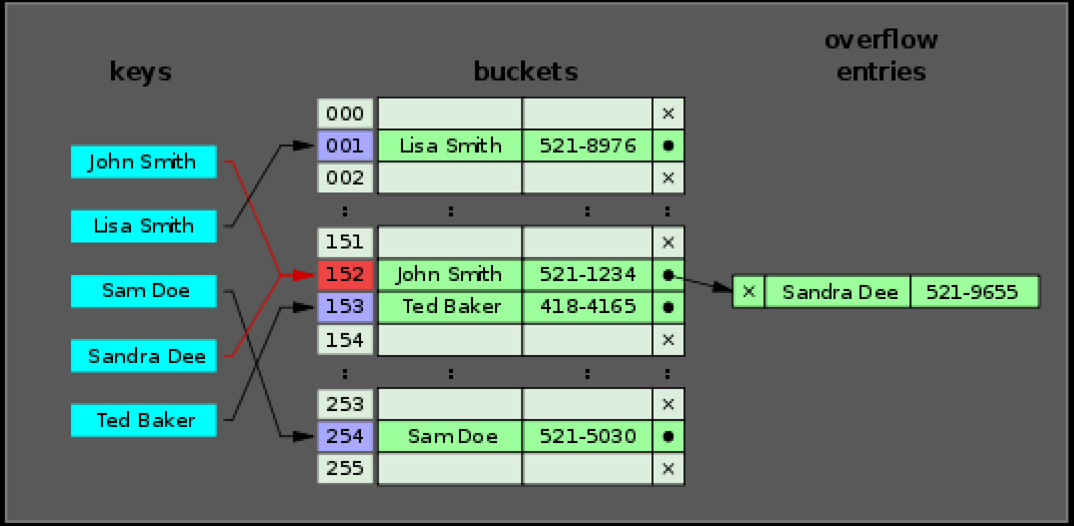


https://en.wikipedia.org/wiki/Hash_table

Collision Resolution: Separate Chaining

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

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

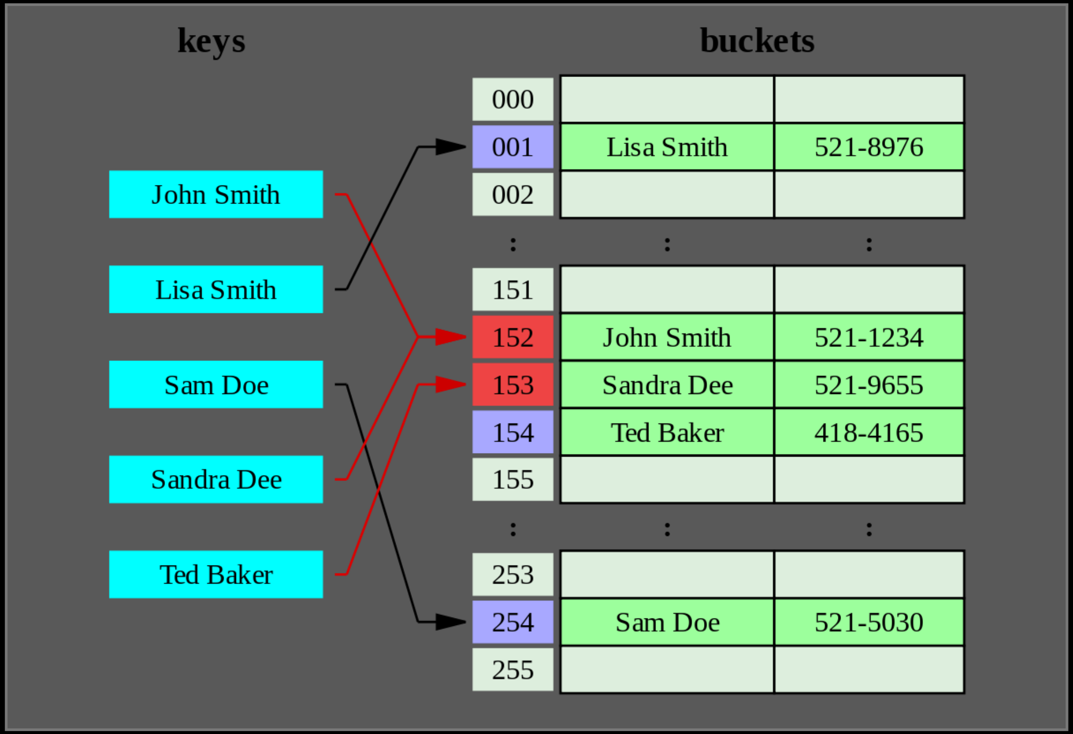


https://en.wikipedia.org/wiki/Hash_table

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



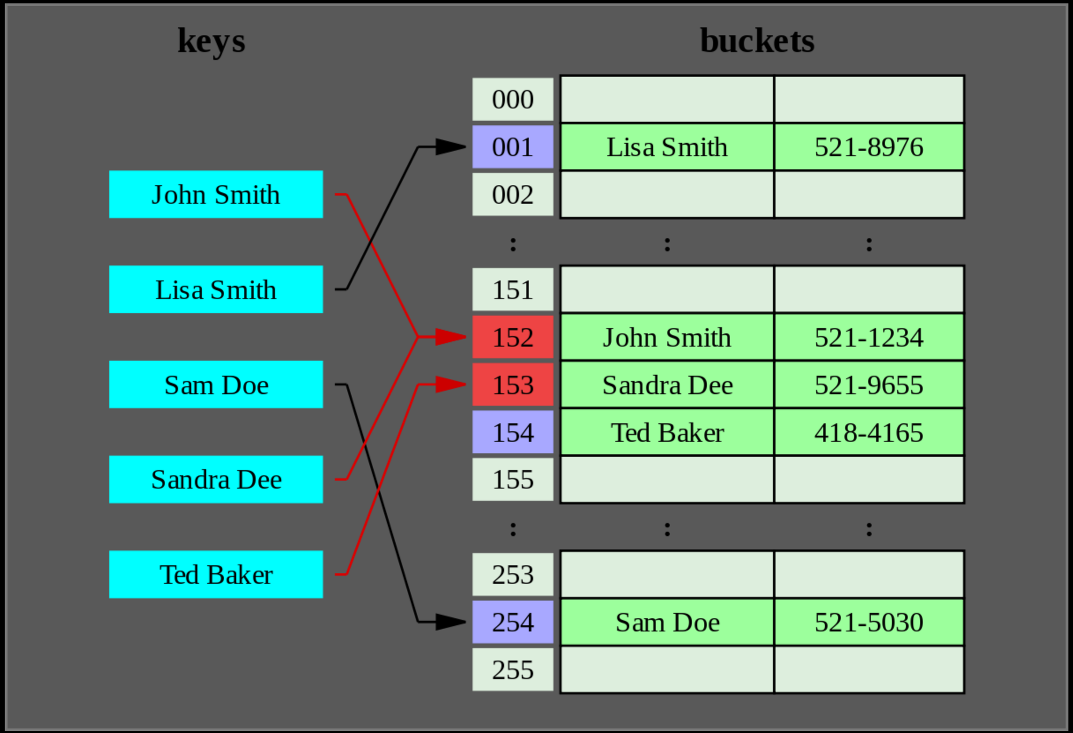
https://en.wikipedia.org/wiki/Hash_table

Move the probe by 1 unit

Collision Resolution: Open Addressing (Quadratic Probing)

Data -> $\boxed{\text{Hash Function}}$ -> Hash Code -> $\boxed{\text{Reduce}}$ -> Index

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



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

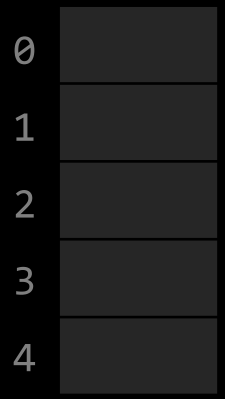
Move the probe by 1, 4, 9, 16 … units

Hash Table Example: Separate Chaining

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

`H(key) = Sum of key's ASCII characters`

`R(Hashcode) = Hashcode % TABLE_SIZE`



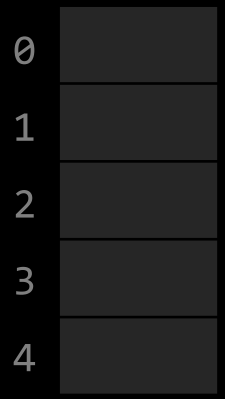
Hash Table Example: Separate Chaining

Data -> $\boxed{\text{Hash Function}}$ -> Hash Code -> $\boxed{\text{Reduce}}$ -> Index

`H(key) = Sum of key's ASCII characters`

`R(Hashcode) = Hashcode % TABLE_SIZE`

Insert a.



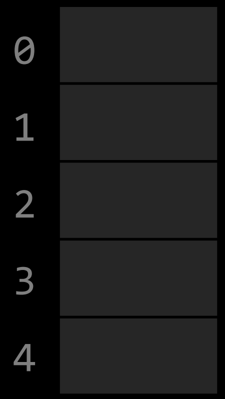
Hash Table Example: Separate Chaining

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

`H(key) = Sum of key's ASCII characters`

`R(Hashcode) = Hashcode % TABLE_SIZE`

```
Insert a.      H(a) = 97
               Index = H(a) % 5 = 2
```

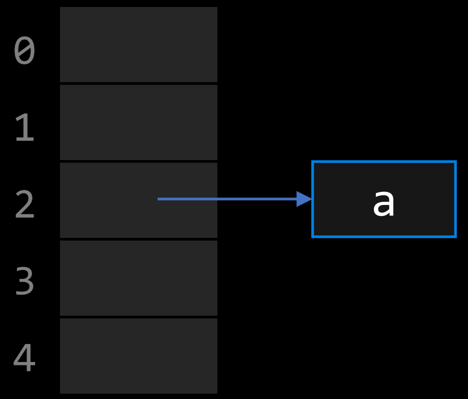


Hash Table Example: Separate Chaining

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

H(key) = Sum of key's ASCII characters | R(Hashcode) = Hashcode % TABLE_SIZE

Insert a.   H(a) = 97
Index = H(a) % 5 = 2



Load Factor = 0.2
Maximum Load Factor = 0.8

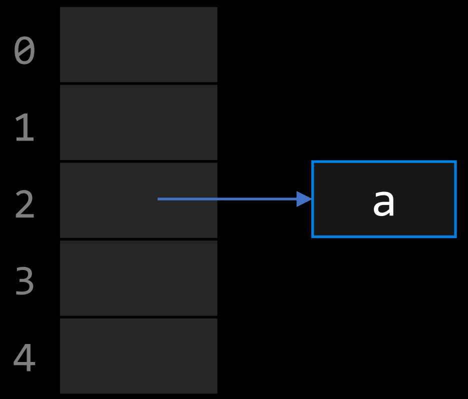
Hash Table Example: Separate Chaining

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

`H(key) = Sum of key's ASCII characters`

`R(Hashcode) = Hashcode % TABLE_SIZE`

Insert ac.



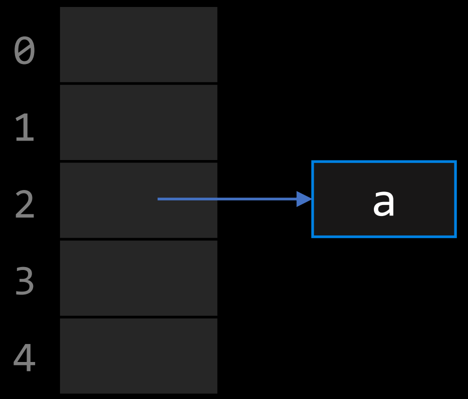
Hash Table Example: Separate Chaining

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

`H(key) = Sum of key's ASCII characters`

`R(Hashcode) = Hashcode % TABLE_SIZE`

```
Insert ac.     H(ac) = 97 + 99 = 196
               Index = H(ac) % 5 = 1
```



Hash Table Example: Separate Chaining

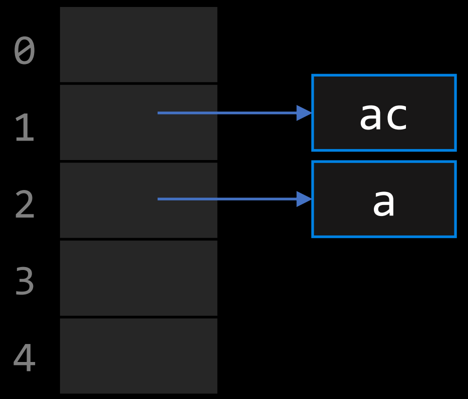
Data -> Hash Function -> Hash Code -> Reduce -> Index

`H(key) = Sum of key’s ASCII characters`

`R(Hashcode) = Hashcode % TABLE_SIZE`

Insert ac.

H(ac) = 97 + 99 = 196
Index = H(ac) % 5 = 1



Load Factor = 0.4
Maximum Load Factor = 0.8

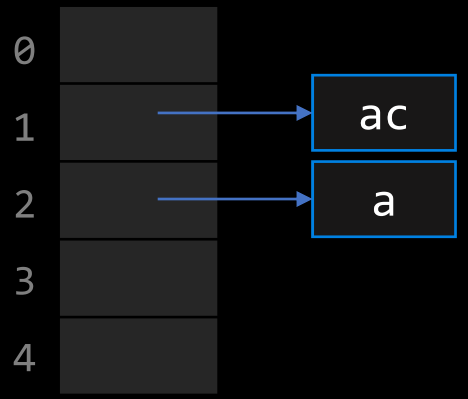
Hash Table Example: Separate Chaining

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

`H(key) = Sum of key's ASCII characters`

`R(Hashcode) = Hashcode % TABLE_SIZE`

`Insert f.`



Hash Table Example: Separate Chaining

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

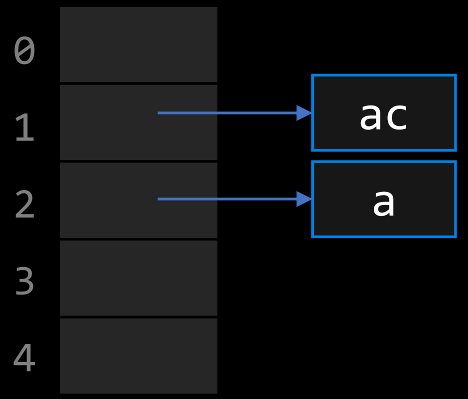
H(key) = Sum of key’s ASCII characters

R(Hashcode) = Hashcode % TABLE_SIZE

Insert f.

H(e) = 101

Index = H(e) % 5 = 1



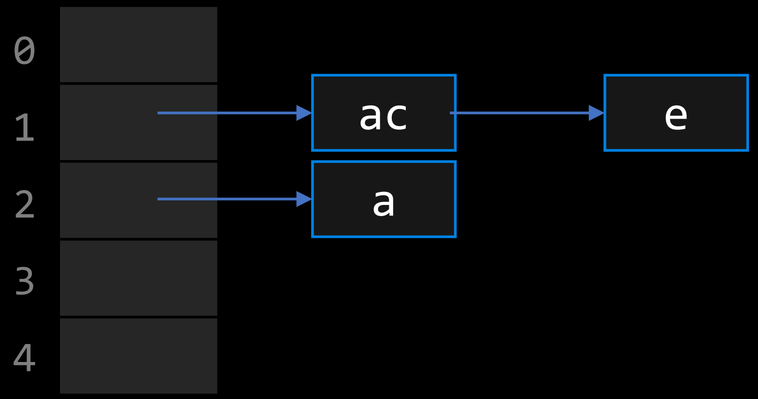
Hash Table Example: Separate Chaining

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

`H(key) = Sum of key’s ASCII characters`

`R(Hashcode) = Hashcode % TABLE_SIZE`

```
Insert f.      H(e) = 101
               Index = H(e) % 5 = 1
```



Load Factor = 0.6
Maximum Load Factor = 0.8

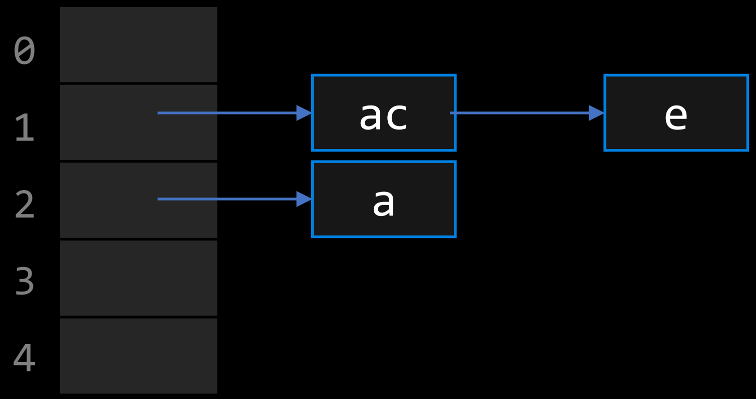
Hash Table Example: Separate Chaining

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

`H(key) = Sum of key's ASCII characters`

`R(Hashcode) = Hashcode % TABLE_SIZE`

Search e.

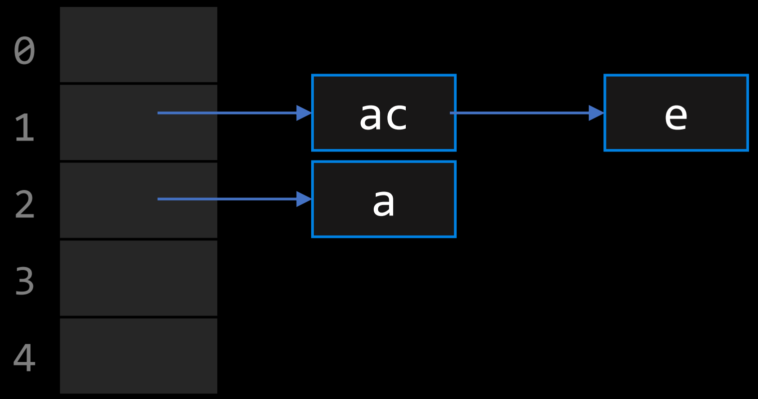


Hash Table Example: Separate Chaining

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

`H(key) = Sum of key's ASCII characters` `R(Hashcode) = Hashcode % TABLE_SIZE`

```
Search e.     H(e) = 101
              Index = H(e) % 5 = 1
```



Hash Table Example: Separate Chaining

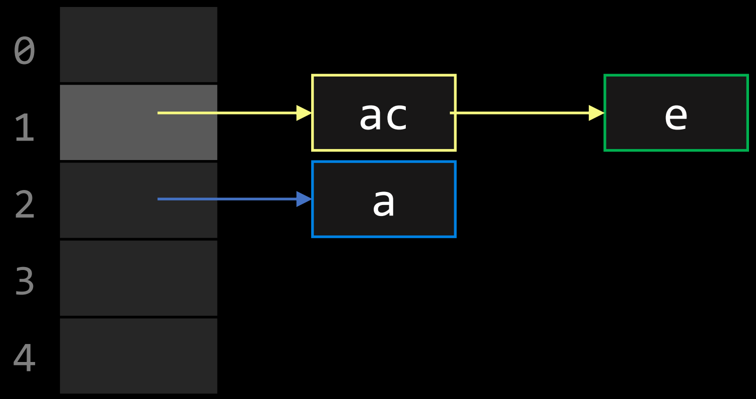
Data -> Hash Function -> Hash Code -> Reduce -> Index

`H(key) = Sum of key's ASCII characters`

`R(Hashcode) = Hashcode % TABLE_SIZE`

Search e.

H(e) = 101
Index = H(e) % 5 = 1



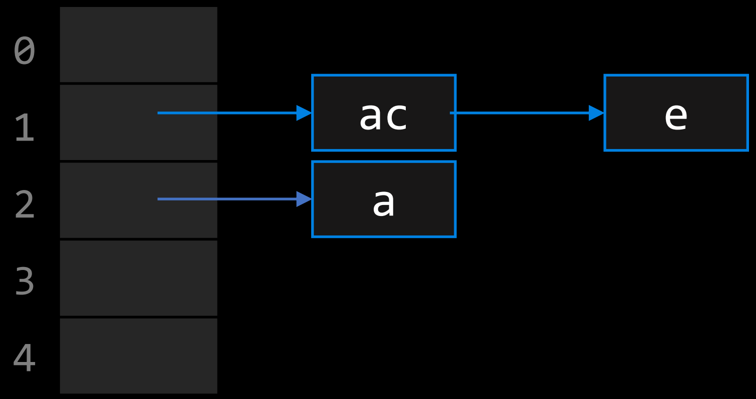
Hash Table Example: Separate Chaining

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

`H(key) = Sum of key's ASCII characters`

`R(Hashcode) = Hashcode % TABLE_SIZE`

`Insert cat.`



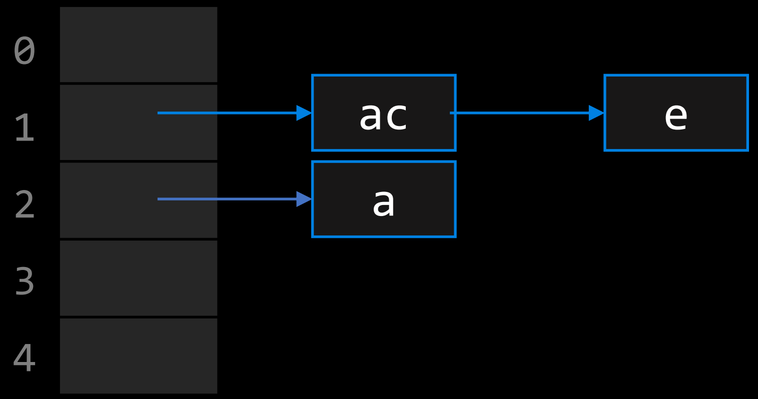
Hash Table Example: Separate Chaining

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

`H(key) = Sum of key's ASCII characters`

`R(Hashcode) = Hashcode % TABLE_SIZE`

```
Insert cat.    H(cat) = 99 + 97 + 116 = 312
               Index = H(cat) % 5 = 2
```



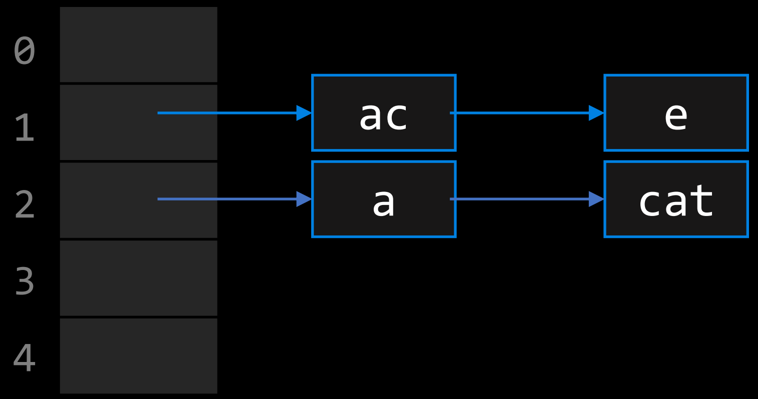
Hash Table Example: Separate Chaining

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

`H(key) = Sum of key's ASCII characters`

`R(Hashcode) = Hashcode % TABLE_SIZE`

```
Insert cat.    H(cat) = 99 + 97 + 116 = 312
               Index = H(cat) % 5 = 2
```




```
Load Factor = 0.8
Maximum Load Factor = 0.8
```

Hash Table Example: Separate Chaining

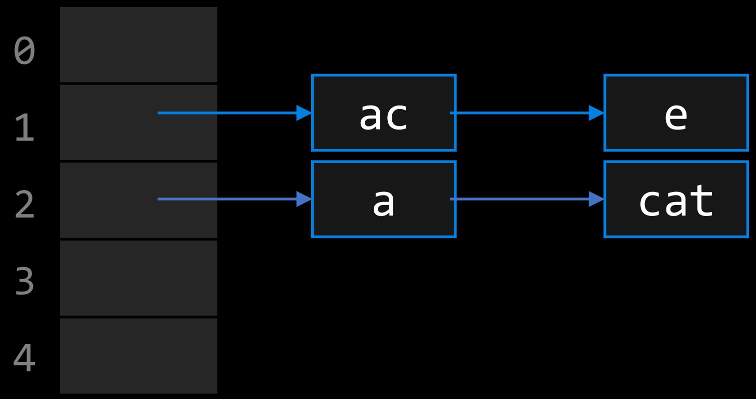
Data -> Hash Function -> Hash Code -> Reduce -> Index

`H(key) = Sum of key's ASCII characters`

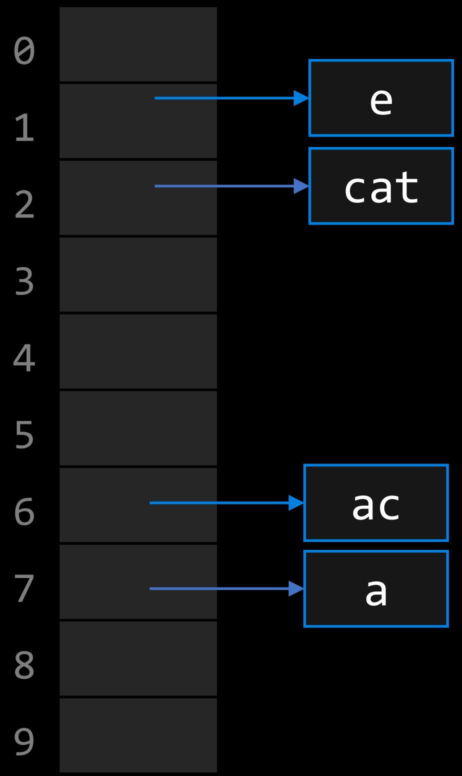
`R(Hashcode) = Hashcode % TABLE_SIZE`

Rehashing

```
H(ac) = 196, Index = 196 % 10 = 6
H(e) = 101, Index = 101 % 10 = 1
H(a) = 97, Index = 97 % 10 = 7
H(cat) = 312, Index = 312 % 10 = 2
```




```
Load Factor = 0.8
Maximum Load Factor = 0.8
```

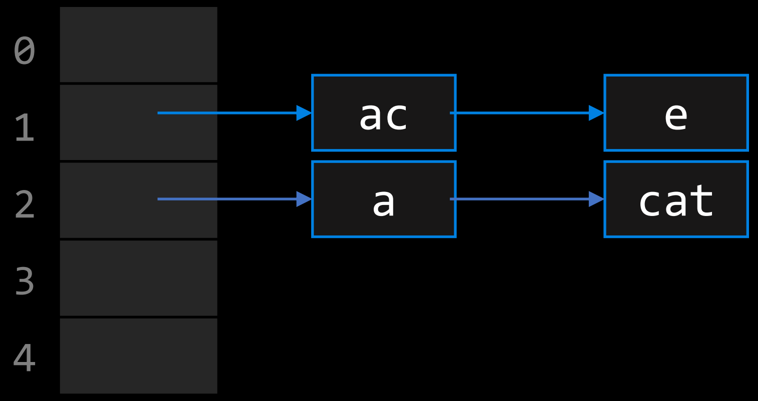


Hash Table Example: Separate Chaining

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

`H(key) = Sum of key’s ASCII characters`

`R(Hashcode) = Hashcode % TABLE_SIZE`

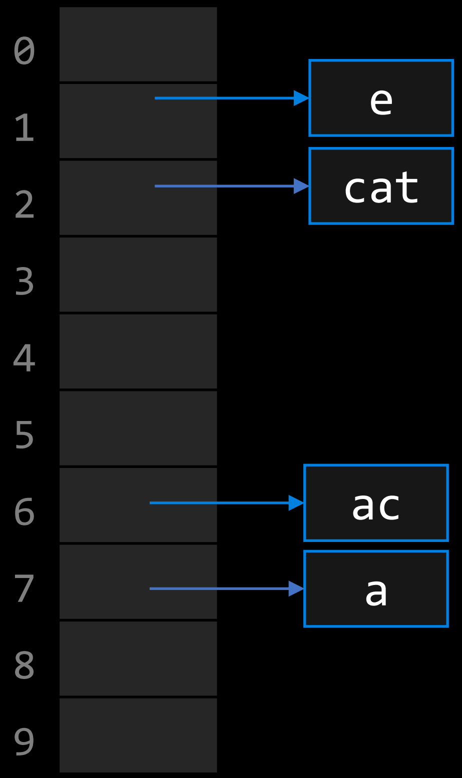


Load Factor = 0.8
Maximum Load Factor = 0.8

Rehashing

H(ac) = 196, Index = 196 % 10 = 6
H(e) = 101, Index = 101 % 10 = 1
H(a) = 97, Index = 97 % 10 = 7
H(cat) = 312, Index = 312 % 10 = 2

Load Factor = 0.4
Maximum Load Factor = 0.8



Hash Table Example: Open addressing with Linear Probing

Data -> $\boxed{\text{Hash Function}}$ -> Hash Code -> $\boxed{\text{Reduce}}$ -> Index

`H(key) = Sum of key’s ASCII characters` | `R(Hashcode) = Hashcode % TABLE_SIZE`

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

0	
1	
2	
3	
4	

https://stackoverflow.com/questions/9124331/meaning-of-open-hashing-and-closed-hashing

Hash Table Example: Open addressing with Linear Probing

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

`H(key) = Sum of key's ASCII characters`

`R(Hashcode) = Hashcode % TABLE_SIZE`

```
Insert:

H(ac) = 196, 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(key) = Sum of key's ASCII characters

R(Hashcode) = Hashcode % TABLE_SIZE

Insert:

H(ac) = 196, Index = 196 % 5 = 1

0	
1	ac
2	
3	
4	

Hash Table Example: Open addressing with Linear Probing

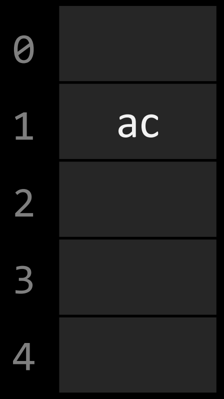
Data -> Hash Function -> Hash Code -> Reduce -> Index

`H(key) = Sum of key's ASCII characters`

`R(Hashcode) = Hashcode % TABLE_SIZE`

```
Insert:

H(ac) = 196, Index = 196 % 5 = 1
```



Hash Table Example: Open addressing with Linear Probing

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

`H(key) = Sum of key's ASCII characters`

`R(Hashcode) = Hashcode % TABLE_SIZE`

```
Insert:

H(ac) = 196, Index = 196 % 5 = 1
H(e) = 101, 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(key) = Sum of key's ASCII characters`

`R(Hashcode) = Hashcode % TABLE_SIZE`

```
Insert:

H(ac) = 196, Index = 196 % 5 = 1
H(e) = 101, Index = 101 % 5 = 1
```

0	
1	ac
2	e
3	
4	

Hash Table Example: Open addressing with Linear Probing

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

`H(key) = Sum of key's ASCII characters`

`R(Hashcode) = Hashcode % TABLE_SIZE`

```
Insert:

H(ac) = 196, Index = 196 % 5 = 1
H(e) = 101, Index = 101 % 5 = 1
H(a) = 97, Index = 97 % 5 = 2
```

0	
1	ac
2	e
3	
4	

Hash Table Example: Open addressing with Linear Probing

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

`H(key) = Sum of key's ASCII characters`

`R(Hashcode) = Hashcode % TABLE_SIZE`

```
Insert:

H(ac) = 196, Index = 196 % 5 = 1
H(e) = 101, Index = 101 % 5 = 1
H(a) = 97, Index = 97 % 5 = 2
```

0	
1	ac
2	e
3	a
4	

Hash Table Example: Open addressing with Linear Probing

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

```
H(key) = Sum of key's character ASCII
```

```
R(Hashcode) = Hashcode % TABLE_SIZE
```

```
Insert:

H(ac) = 196, Index = 196 % 5 = 1
H(e) = 101, Index = 101 % 5 = 1
H(a) = 97, Index = 97 % 5 = 2
H(cat) = 312, Index = 312 % 5 = 2
```

0	
1	ac
2	e
3	a
4	

Hash Table Example: Open addressing with Linear Probing

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

`H(key) = Sum of key's character ASCII`

`R(Hashcode) = Hashcode % TABLE_SIZE`

```
Insert:

H(ac) = 196, Index = 196 % 5 = 1
H(e) = 101, Index = 101 % 5 = 1
H(a) = 97, Index = 97 % 5 = 2
H(cat) = 312, Index = 312 % 5 = 2
```

0	
1	ac
2	e
3	a
4	cat

Hash Table Example: Open addressing with Linear Probing

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

`H(key) = Sum of key's character ASCII`

`R(Hashcode) = Hashcode % TABLE_SIZE`

```
Search ab:

H(ab) = 195, Index = 195 % 5 = 0
```

0	
1	ac
2	e
3	a
4	cat

Hash Table Example: Open addressing with Linear Probing

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

`H(key) = Sum of key's character ASCII`

`R(Hashcode) = Hashcode % TABLE_SIZE`

```
Search ab:

H(ab) = 195, Index = 195 % 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.
```

0	
1	ac
2	e
3	a
4	cat

Statuses:
- Never used
- Deleted
- Occupied

Statuses:
- Never used
- Deleted
- Occupied

Hash Table Example: Open addressing with Quadratic Probing

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

`H(key) = Sum of key's character ASCII`

`R(Hashcode) = Hashcode % 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(ac) = 196, Index = 196 % 5 = 1
H(e) = 101, Index = 101 % 5 = 1
```

0	
1	ac
2	e
3	
4	

Hash Table Example: Open addressing with Quadratic Probing

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

`H(key) = Sum of key's character ASCII`

`R(Hashcode) = Hashcode % 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(ac) = 196, Index = 196 % 5 = 1
H(e) = 101, Index = 101 % 5 = 1
H(j) = 106, Index = 106 % 5 = 1
```

0	
1	ac
2	e
3	
4	

Hash Table Example: Open addressing with Quadratic Probing

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

`H(key) = Sum of key's character ASCII`

`R(Hashcode) = Hashcode % 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(ac) = 196, Index = 196 % 5 = 1
H(e) = 101, Index = 101 % 5 = 1
H(j) = 106, Index = 106 % 5 = 1


Move 1 (conflict 2 - e),
Move 4 from hashed value; (1+4)
```

0	j
1	ac
2	e
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

```
class Set
{
    private:
        string arraySet[100];
    public:
        void insert(int value);
        bool search(int value);
};

void ArraySet::insert(int value)
{
    //find the hash of the value
    //reduce the hash to get an index
    //check if value is not at index
            //insert value at index
    //otherwise, use collision resolution strategy
}

bool ArraySet::search(int value)
{
    //find the hash of the value
    //reduce the hash to get an index
    //check if value is not at index
            //return false
    //otherwise, search based on collision resolution strategy
}
```

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

Sets and Maps in C++ Example

```cpp
//Unordered Set - Hash-based
unordered_set <int> s2;

// insert elements in random order
s2.insert(5);
s2.insert(2);
s2.insert(4);
s2.insert(11);
s2.insert(2); // only one 2 will be added to the set

// printing set
unordered_set <int> :: iterator itr2;
cout << "The set s2 is:";
for (itr2 = s2.begin(); itr2 != s2.end(); ++itr2)
           cout << "  " << *itr2;
cout << endl;
cout << "Bucket count: " << s2.bucket_count();
cout << "\nLoad Factor: " << s2.load_factor();
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
```

https://onlinegdb.com/SyMCuHOlD

```cpp
//Unordered Map - Hash-based
unordered_map<char,int> table_unordered;

// insert elements in random order
table_unordered['b']=30;
table_unordered['a']=10;
table_unordered['c']=50;
table_unordered['a']=40;

// printing set
for(auto member: table_unordered)
      cout << member.first << " " << member.second <<"\n";

cout << "Load Factor: " << table_unordered.load_factor();
```

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

https://onlinegdb.com/SkHykUnlP

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}

https://stepik.org/lesson/390638/step/2

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}
```

https://stepik.org/lesson/390638/step/2

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}

```
pair<int, int> two_sum(vector<int>& arr, int target)
{
    unordered_map<int, int> map;
    pair<int, int> result(-1, -1);
    for (int i = 0; i < arr.size(); i++)
    {
        int diff = target - arr[i];
        if(map.count(diff))          //check if complement is present in the set
        {
            result.first = map[diff];
            result.second = i;
            break;
        }
        map[arr[i]] = i;             //add the element to the set otherwise
    }
    return result;
}
```