HASHING

ALGORITHMS AND DATA STRUCTURES

HASHING

Hashing content

- Basics of hashing
- Hashtable
- Collision resolution techniques
    - Separate Changing
    - Linear Probing
    - Quadratic probing
- Double hashing
- Apps

HASHING

Hashing idea

- Hashing is a technique that is used to uniquely identify a specific object from a group of similar objects. Some examples of how hashing is used in our lives include:
  - In universities, each student is assigned a unique roll number that can be used to retrieve information about them.
  - In libraries, each book is assigned a unique number that can be used to determine information about the book, such as its exact position in the library or the users it has been issued to etc.

HASHING IDEA

- In hashing, large keys are converted into small keys by using **hash functions**. The values are then stored in a data structure called **hash table**. The idea of hashing is to distribute entries (key/value pairs) uniformly across an array. Each element is assigned a key (converted key). By using that key you can access the element in **O(1)** time. Using the key, the algorithm (hash function) computes an index that suggests where an entry can be found or inserted

HASHING IDEA

- An element is converted into an integer by using a hash function. This element can be used as an index to store the original element, which falls into the hash table.
- The element is stored in the hash table where it can be quickly retrieved using hashed key.

```
hash = hashfunc(key)
index = hash % array_size
```

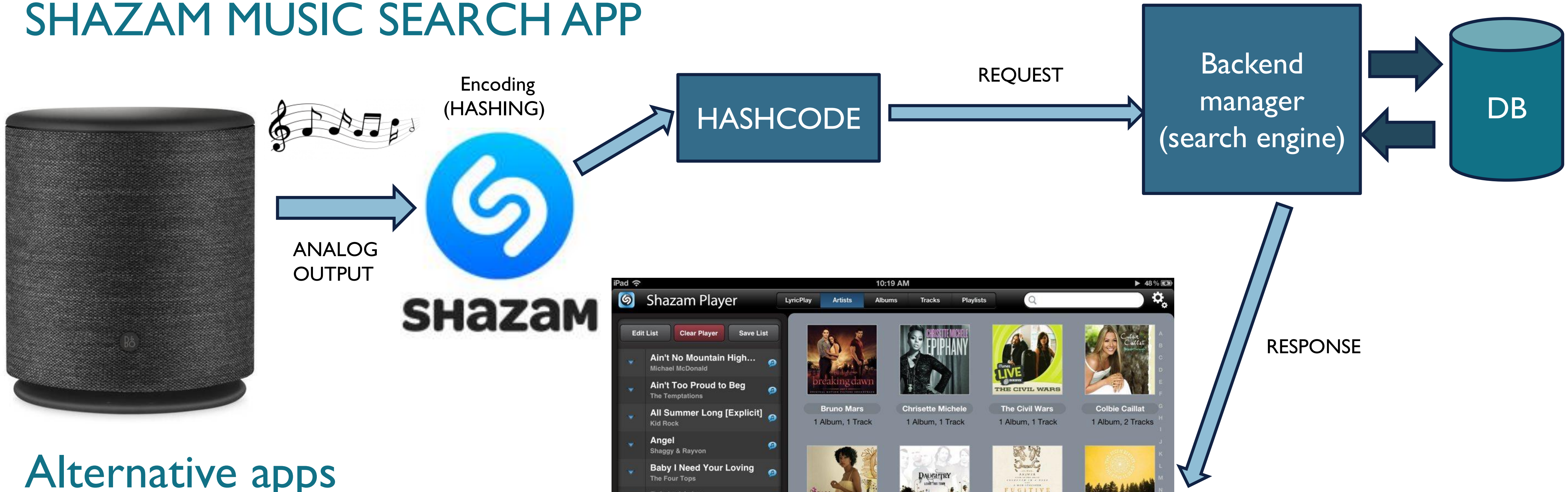
(a number between 0 and array_size – 1)

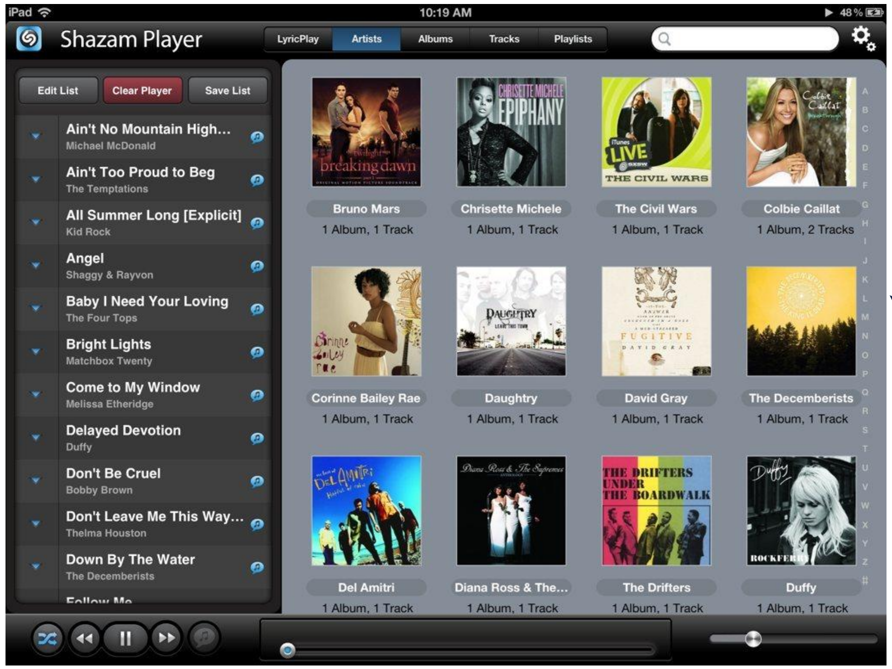
THE GOOD HASHING

- Easy to compute: It should be easy to compute and must not become an algorithm in itself.
- Uniform distribution: It should provide a uniform distribution across the hash table and should not result in clustering.
- Less collisions: Collisions occur when pairs of elements are mapped to the same hash value. These should be avoided.

HASHING APPLICATIONS

SHAZAM MUSIC SEARCH APP





Ain't No Mountain High…
Michael McDonald
Ain't Too Proud to Beg
The Temptations
All Summer Long [Explicit]
Kid Rock
Angel
Shaggy & Rayvon
Baby I Need Your Loving
The Four Tops
Bright Lights
Matchbox Twenty
Come to My Window
Melissa Etheridge
Delayed Devotion
Duffy
Don't Be Cruel
Bobby Brown
Don't Leave Me This Way…
Thelma Houston
Down By The Water
The Decemberists

Bruno Mars 1 Album, 1 Track	Chrisette Michele 1 Album, 1 Track	The Civil Wars 1 Album, 1 Track	Colbie Caillat 1 Album, 2 Tracks
Corinne Bailey Rae 1 Album, 1 Track	Daughtry 1 Album, 1 Track	David Gray 1 Album, 1 Track	The Decemberists 1 Album, 1 Track
Del Amitri 1 Album, 1 Track	Diana Ross & The… 1 Album, 1 Track	The Drifters 1 Album, 1 Track	Duffy 1 Album, 1 Track

Alternative apps

SOUNDCLOUD

Google

HASHING EXAMPLE

HASHING

EXAMPLE 1

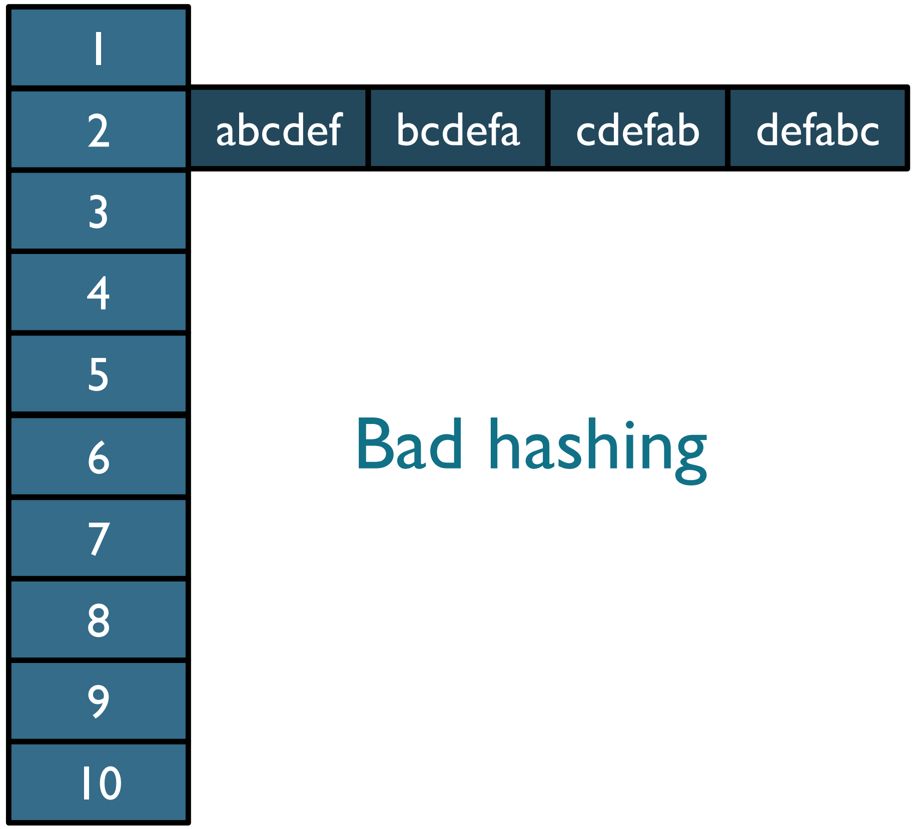
- {“abcdef”,“bcdefa”,“cdefab” ,“defabc” }. Has to be stored in hash table
- The index for a specific string will be equal to the sum of the ASCII values of the characters modulo 599.

As 599 is a prime number, it will reduce the possibility of indexing different strings (collisions). It is recommended that you use prime numbers in case of modulo. The ASCII values of a, b, c, d, e, and f are 97, 98, 99, 100, 101, and 102 respectively. Since all the strings contain the same characters with different permutations, the sum will 599.

EXAMPLE I

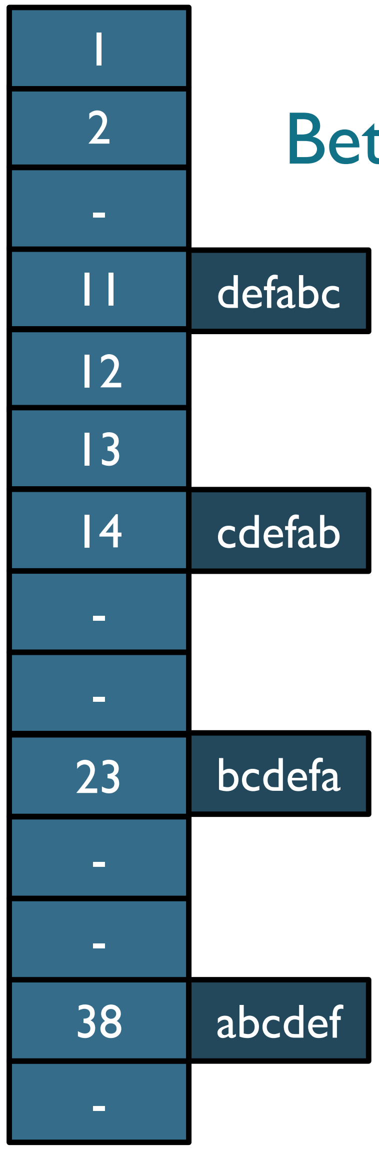
The hash function will compute the same index for all the strings and the strings will be stored in the hash table in the following format. As the index of all the strings is the same, you can create a list on that index and insert all the strings in that list

String	Hash function		Index
abcdef	(97 + *98* + 99 + *100* + 101 + *102*)% 599	=	2
bcdefa	(98 + *99* + 100 + *101* + 102 + *97*)% 599	=	2
cdefab	(99 + *100* + 101 + *102* + 97 + *98*)% 599	=	2
defabc	(100 + *101* + 102 + *97* + 98 + *99*)% 599	=	2



Bad hashing

EXAMPLE 2



Better hashing

Modified hash function

The index for a specific string will be equal to sum of ASCII values of characters multiplied by their respective order in the string after which it is modulo with 2069 (prime number).

String	Hash function	Index
abcdef	(97*1 + 98*2 + 99*3 + 100*4 + 101*5 + 102*6)% 2069	38
bcdefa	(98*1 + 99*2 + 100*3 + 101*4 + 102*5 + 97*6)% 2069	23
cdefab	(99*1 + 100*2 + 101*3 + 102*4 + 97*5 + 98*6)% 2069	14
defabc	(100*1 + 101*2 + 102*3 + 97*4 + 98*5 + 99*6)% 2069	11

HASH TABLE

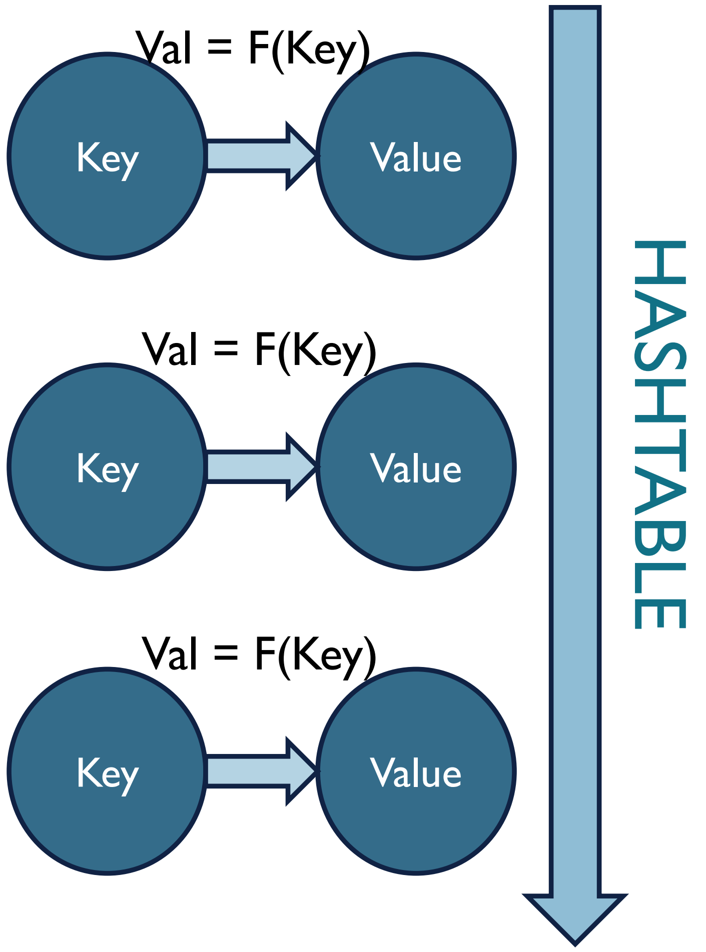
HASHING

HASH TABLE

The Hash Table

A hash table is a data structure that is used to store keys/value pairs. It uses a hash function to compute an index into an array in which an element will be inserted or searched. By using a good hash function, hashing can work well. Under reasonable assumptions, the average time required to search for an element in a hash table is **O(1)**.

K → V Pairs



EXAMPLE

Let us consider string S. You are required to count the frequency of all the characters in this string.

```
string S = “ababcd”;
```

```
void FrecuencyCounting(string S)
   {
       for(char c = ‘a’;c <= ‘z’;++c)
       {
           int frequency = 0;
           for(int i = 0;i < S.length();++i)
               if(S[i] == c)
                   frequency++;
           cout << c << ‘ ‘ << frequency << endl;
       }
   }
```

output

```
a 2
b 2
c 1
d 1
e 0
f 0
...
z 0
```

OPTIMIZED CODE

Let us apply hashing to this problem. Take an array frequency of size 26 and hash the 26 characters with indices of the array by using the hash function. Then, iterate over the string and increase the value in the frequency at the corresponding index for each character. The complexity of this approach is **O(N)** where **N** is the size of the string.

```
int Frequency[26];

    int hashFunc(char c)
    {
        return (c - 'a');
    }

    void countFre(string S)
    {
        for(int i = 0;i < S.length();++i)
        {
            int index = hashFunc(S[i]);
            Frequency[index]++;
        }
        for(int i = 0;i < 26;++i)
            cout<<(char)(i+'a')<< ' '<<Frequency[i]<< endl;
    }
```

output

```
a 2
b 2
c 1
d 1
e 0
f 0
 …
z 0
```

HASH TABLE

Frequency

Char	Index	Value
A	0	0
B	1	0
C	2	0
D	3	0
E	4	0
	-	-
	-	-
Y	24	0
Z	25	0

After Update

Frequency

Char	Index	Value
A	0	2
B	1	2
C	2	1
D	3	1
E	4	0
Y	24	0
Z	25	0

COLLISION RESOLUTION TECHNIQUE – SEPARATE CHAINING

HASHING

COLLISION RESOLUTION: SEPARATE CHAINING

Separate Chaining

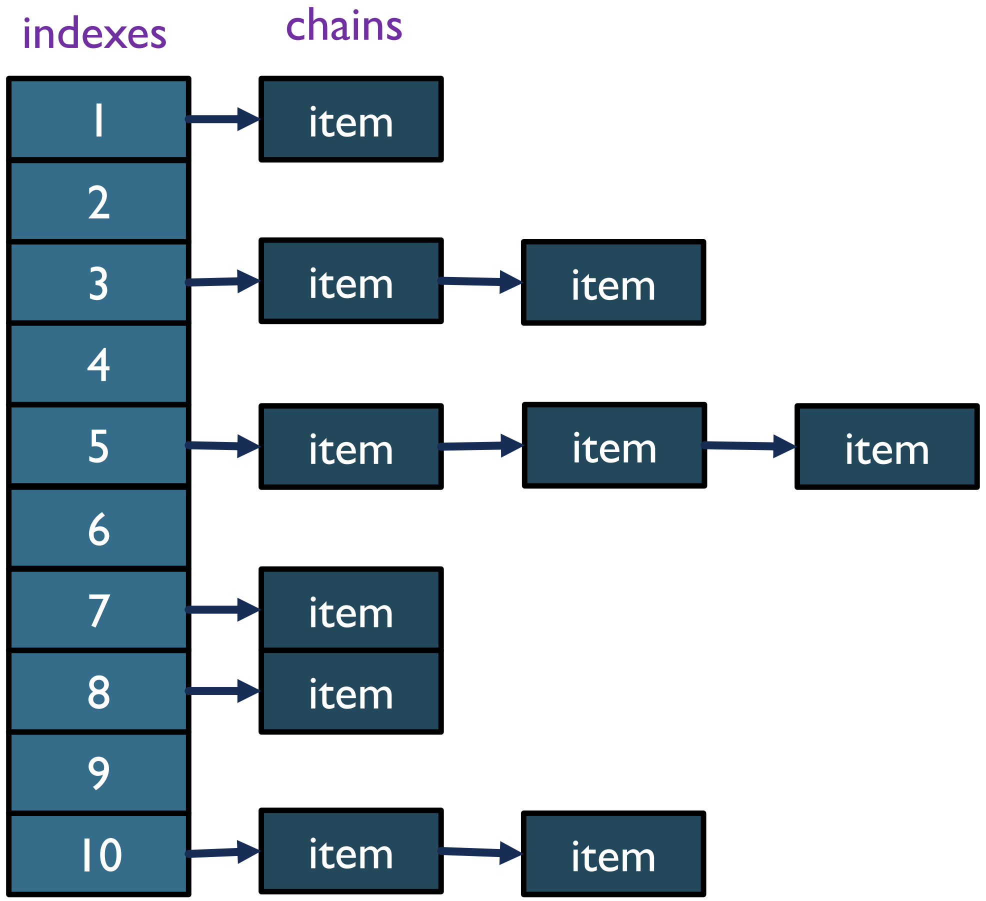
- Separate chaining is one of the most commonly used collision resolution techniques. It is usually implemented using linked lists. In separate chaining, each element of the hash table is a linked list. To store an element in the hash table you must insert it into a specific linked list. If there is any collision (i.e. two different elements have same hash value) then store both the elements in the same linked list.

```
vector<vector<item>> HT;
```

You can use one of
this data structure

```
vector<list<item>> HT;
```

Hash Table

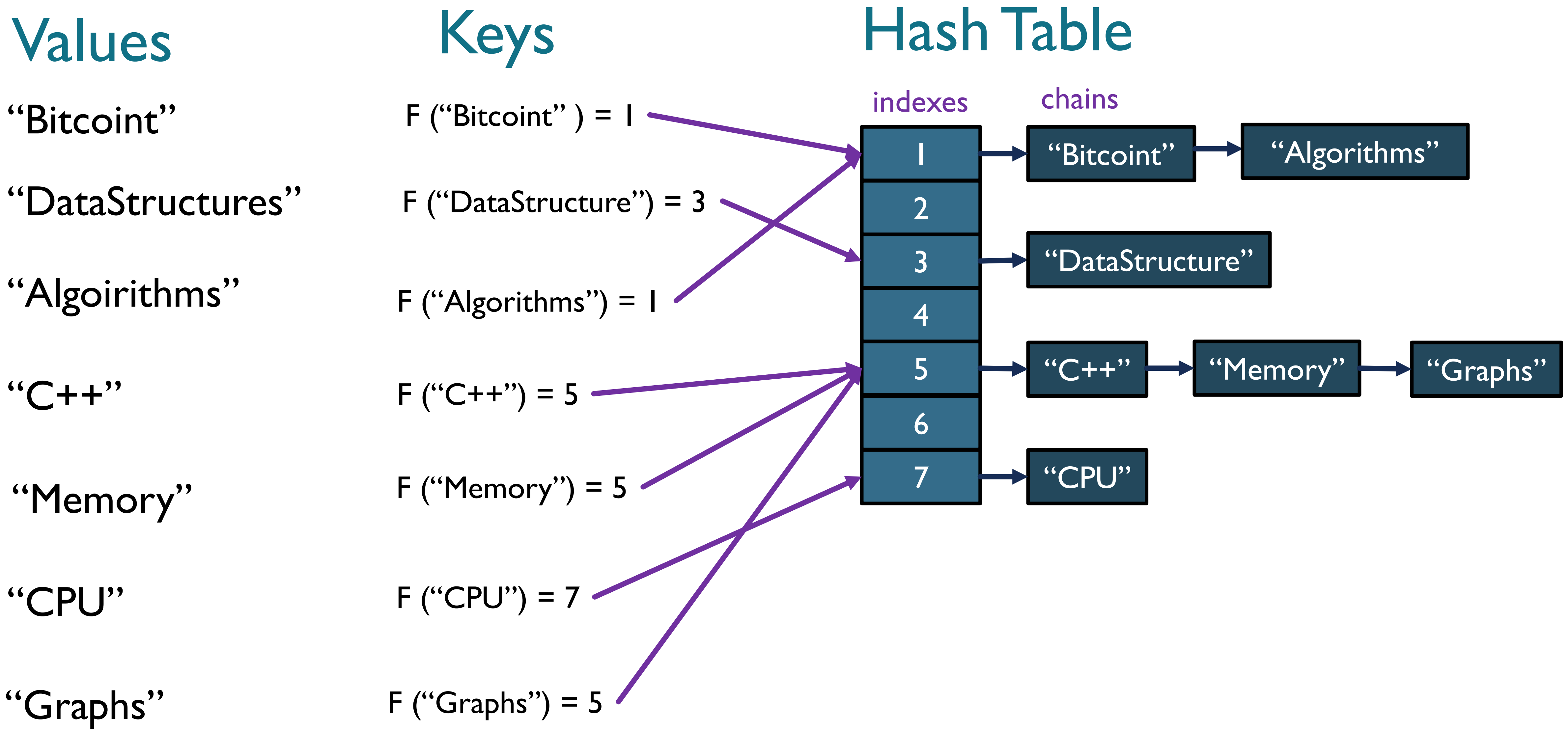


COLLISION RESOLUTION: SEPARATE CHAINING

- The cost of a lookup is that of scanning the entries of the selected linked list for the required key. If the distribution of the keys is sufficiently uniform, then the average cost of a lookup depends only on the average number of keys per linked list. For this reason, chained hash tables remain effective even when the number of table entries (N) is much higher than the number of slots.

- For separate chaining, the worst-case scenario is when all the entries are inserted into the same linked list. The lookup procedure may have to scan all its entries, so the worst-case cost is proportional to the number (N) of entries in the table.

EXAMPLE


Values
Keys
Hash Table
"Bitcoint"
"DataStructures"
"Algoirithms"
"C++"
"Memory"
"CPU"
"Graphs"
F ("Bitcoint") = 1
F ("DataStructure") = 3
F ("Algorithms") = 1
F ("C++") = 5
F ("Memory") = 5
F ("CPU") = 7
F ("Graphs") = 5
indexes
chains
1
2
3
4
5
6
7
"Bitcoint"
"Algorithms"
"DataStructure"
"C++"
"Memory"
"Graphs"
"CPU"

COLLISION RESOLUTION: SEPARATE CHAINING

- The cost of a lookup is that of scanning the entries of the selected linked list for the required key. If the distribution of the keys is sufficiently uniform, then the average cost of a lookup depends only on the average number of keys per linked list. For this reason, chained hash tables remain effective even when the number of table entries (N) is much higher than the number of slots.

- For separate chaining, the worst-case scenario is when all the entries are inserted into the same linked list. The lookup procedure may have to scan all its entries, so the worst-case cost is proportional to the number (N) of entries in the table.

SEPARATE CHAINING: IMPLEMENTATION

```
vector<string>hashTable[20];
int hashTableSize=20;



void insert(string s){

   // Compute the index
   int index = hashFunc(s);
   // Insert the element in the
   //linked list at the articular index
   hashTable[index].push_back(s);

}
```

```
void search(string s){
    //Compute the index by using the hash function
    int index = hashFunc(s);
    //Search the linked list at that specific index
    for(int i = 0;i < hashTable[index].size();i++)
    {
       if(hashTable[index][i] == s)
       {
           cout << s << " is found!" << endl;
           return;
       }
   }
   cout << s << " is not found!" << endl;
}
```

COLLISION RESOLUTION TECHNIQUE – LINEAR PROBING

HASHING

LINEAR PROBING

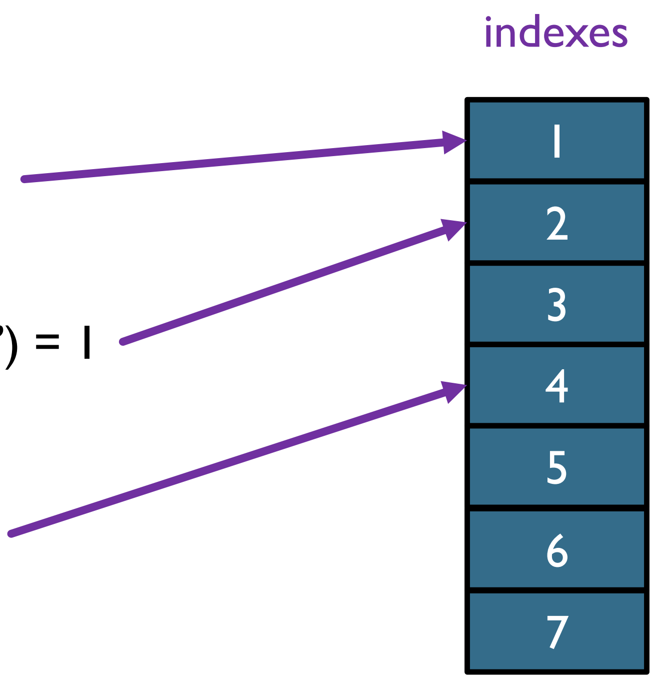
- In open addressing, instead of in linked lists, all entry records are stored in the array itself. When a new entry has to be inserted, the hash index of the hashed value is computed and then the array is examined (starting with the hashed index). If the slot at the hashed index is unoccupied, then the entry record is inserted in slot at the hashed index else it proceeds in some probe sequence until it finds an unoccupied slot.
- When searching for an entry, the array is scanned in the same sequence until either the target element is found or an unused slot is found. This indicates that there is no such key in the table. The name "open addressing" refers to the fact that the location or address of the item is not determined by its hash value

```
index = index % hashTableSize
index = (index + 1) % hashTableSize
index = (index + 2) % hashTableSize
index = (index + 3) % hashTableSize
```

LINEAR PROBING

Values	Keys
“Bitcoint”	F (“Bitcoint”) = 1
“DataStructures”	F (“DataStructure”) = 1
“Algoirithms”	F (“Bitcoint”) = 4

Hash Table



LINEAR PROBING

```
string hashTable[21];
int hashTableSize = 21;

void insert(string s){

 int index = hashFunc(s);
 while(hashTable[index] != "")
    index = (index + 1) % hashTableSize;
 hashTable[index] = s;
}
```

```
void search(string s){
int index = hashFunc(s);
   //Search for an unused slot
   //if the index will exceed the hashTableSize
   // then roll back
 while(hashTable[index] != s
          && hashTable[index] != "")
   index = (index + 1) % hashTableSize;
   //Check if the element is present
    //in the hash table
   if(hashTable[index] == s)
     cout << s << " is found!" << endl;
   else
     cout << s << " is not found!" << endl;
}
```

COLLISION RESOLUTION TECHNIQUE – QUADRATIC PROBING

HASHING

QUADRATIC PROBING

Quadratic probing is similar to linear probing and the only difference is the interval between successive probes or entry slots. Here, when the slot at a hashed index for an entry record is already occupied, you must start traversing until you find an unoccupied slot. The interval between slots is computed by adding the successive value of an arbitrary polynomial in the original hashed index

Let us assume that the hashed index for an entry is index and at index there is an occupied slot. The probe sequence will be as follows:

index = index % hashTableSize

index = (index + 1^2) % hashTableSize
index = (index + 2^2) % hashTableSize
index = (index + 3^2) % hashTableSize

QUADRATIC PROBING

```
string hashTable[21];
int hashTableSize = 21;


void insert(string s){
    //Compute the index using the hash function
   int index = hashFunc(s);
   //Search for an unused slot and if the index will exceed the hashTableSize roll back
   int h = 1;
   while(hashTable[index] != ""){
       index = (index + h*h) % hashTableSize;
       h++;
   }
   hashTable[index] = s;
}
```

QUADRATIC PROBING

```
void search(string s){

    //Compute the index using the Hash Function
    int index = hashFunc(s);
    //Search for an unused slot and if the index will exceed the hashTableSize roll back
    int h = 1;
    while(hashTable[index] != s and hashTable[index] != "")
    {
       index = (index + h*h) % hashTableSize;
       h++;
    }
    //Is the element present in the hash table
    if(hashTable[index] == s)
        cout << s << " is found!" << endl;
     else
       cout << s << " is not found!" << endl;
}
```

DOUBLE HASHING

HASHING

DOUBLE HASHING

Double Hashing

- Double hashing is similar to linear probing and the only difference is the interval between successive probes. Here, the interval between probes is computed by using two hash functions.
- Let us say that the hashed index for an entry record is an index that is computed by one hashing function and the slot at that index is already occupied. You must start traversing in a specific probing sequence to look for an unoccupied slot. The probing sequence will be:

```
index = (index + 1 * indexH) % hashTableSize;
index = (index + 2 * indexH) % hashTableSize;
```

Here, **indexH** is the hash value that is computed by another hash function.
Implementation of hash table with double hashing

DOUBLE HASHING

```
string hashTable[21];
int hashTableSize = 21;


void insert(string s){
    //Compute the index using the hash function1
    int index = hashFunc1(s);
    int indexH = hashFunc2(s);
    //Search for an unused slot and if the index exceeds the hashTableSize roll back
    while(hashTable[index] != "")
        index = (index + indexH) % hashTableSize;
    hashTable[index] = s;
}
```

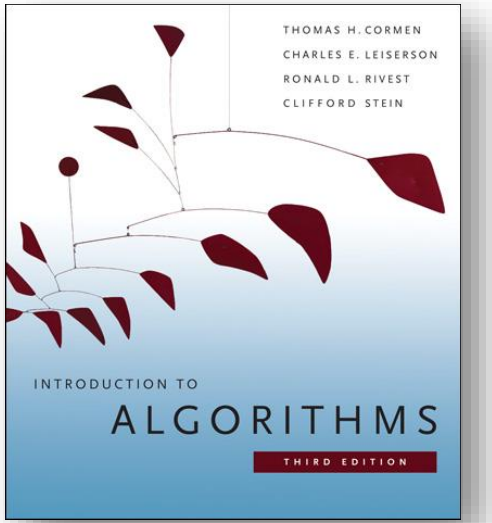
DOUBLE HASHING

```
void search(string s){
      //Compute the index using the hash function
      int index = hashFunc1(s);
      int indexH = hashFunc2(s);
      //Search for an unused slot and if the index exceeds
      // the hashTableSize roll back
      while(hashTable[index] != s and hashTable[index] != "")
          index = (index + indexH) % hashTableSize;
      //Is the element present in the hash table
      if(hashTable[index] == s)
          cout << s << " is found!" << endl;
      else
          cout << s << " is not found!" << endl;
}
```

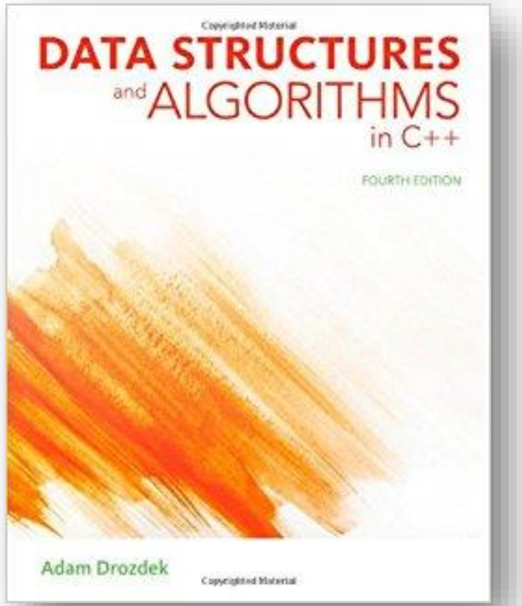
APPLICATIONS

- Associative arrays: Hash tables are commonly used to implement many types of in-memory tables. They are used to implement associative arrays (arrays whose indices are arbitrary strings or other complicated objects).
- *Database indexing*: Hash tables may also be used as disk-based data structures and database indices (such as in dbm).
- *Caches*: Hash tables can be used to implement caches i.e. auxiliary data tables that are used to speed up the access to data, which is primarily stored in slower media.
- *Object representation*: Several dynamic languages, such as Perl, Python, JavaScript, and Ruby use hash tables to implement objects.
- Searching apps

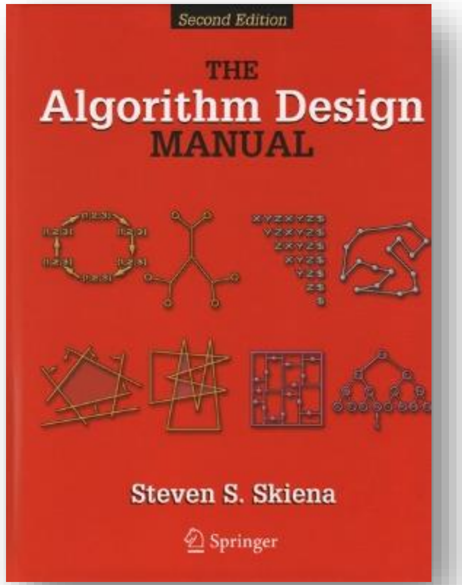
LITERATURE



Thomas H. Cormen
Introduction to Algorithms
Chapter 11 Hash Tables
Page 253.



Adam Drozdek
Data structures and Algorithms in C++
Chapter 10 Hashing
Page 548



Steven S. Skiena
The Algorithms Design Manual
Chapter 3.7 Hashing and string
Page 89