HashTable

Problem: Employee Management System

Suppose, we have a company with 20 Employees. Each employee is assigned a 5 digit Employee ID, which is used to search the employee from the company's employee file.
You need to store the information effectively in the system so that we can easily apply the CRUD operations on the data.

Problem: Employee Management System

1	Employee_Name	EmpID	Salary	Position	DOB	Gender	MaritalStatus	DateofHire	EmploymentStatus	Department	ManagerName	RecruitmentSource	Engagement	EmpSatisfaction	Absences
2	Adinolfi, Wilson	10026	62506	Productio	####	M	Single	7/5/2011	Active	Production	Michael Albert	LinkedIn	4.6	5	1
3	Ait Sidi, Karthikey	10084	1E+05	Sr. DBA	####	M	Married	3/30/2015	Voluntarily Termina	IT/IS	Simon Roup	Indeed	4.96	3	17
4	Akinkuolie, Sarah	10196	64955	Productio	####	F	Married	7/5/2011	Voluntarily Termina	Production	Kissy Sullivan	LinkedIn	3.02	3	3
5	Alagbe,Trina	10088	64991	Productio	####	F	Married	1/7/2008	Active	Production	Elijiah Gray	Indeed	4.84	5	15
6	Anderson, Carol	10069	50825	Productio	####	F	Divorced	7/11/2011	Voluntarily Termina	Production	Webster Butler	Google Search	5	4	2
7	Anderson, Linda	10002	57568	Productio	####	F	Single	1/9/2012	Active	Production	Amy Dunn	LinkedIn	5	5	15
8	Andreola, Colby	10194	95660	Software E	####	F	Single	#########	Active	Software Eng	Alex Sweetwater	LinkedIn	3.04	3	19
9	Athwal, Sam	10062	59365	Productio	####	M	Widowed	9/30/2013	Active	Production	Ketsia Liebig	Employee Referral	5	4	19
10	Bachiochi, Linda	10114	47837	Productio	####	F	Single	7/6/2009	Active	Production	Brannon Miller	Diversity Job Fair	4.46	3	4
11	Bacong, Alejandr	10250	50178	IT Support	####	M	Divorced	1/5/2015	Active	IT/IS	Peter Monroe	Indeed	5	5	16
12	Baczenski, Racha	10252	54670	Productio	####	F	Married	1/10/2011	Voluntarily Termina	Production	David Stanley	Diversity Job Fair	4.2	4	12
13	Barbara, Thomas	10242	47211	Productio	####	M	Married	4/2/2012	Voluntarily Termina	Production	Kissy Sullivan	Diversity Job Fair	4.2	3	15
14	Barbossa, Hector	10012	92328	Data Anal	####	M	Divorced	#########	Active	IT/IS	Simon Roup	Diversity Job Fair	4.28	4	9
15	Barone, Francesc	10265	58709	Productio	####	M	Single	2/20/2012	Active	Production	Kelley Spirea	Google Search	4.6	4	7
16	Barton, Nader	10066	52505	Productio	####	M	Divorced	9/24/2012	Voluntarily Termina	Production	Michael Albert	On-line Web applica	5	5	1
17	Bates, Norman	10061	57834	Productio	####	M	Single	2/21/2011	Terminated for Cau	Production	Kelley Spirea	Google Search	5	4	20
18	Beak, Kimberly	10023	70131	Productio	####	F	Married	7/21/2016	Active	Production	Kelley Spirea	Employee Referral	4.4	3	16
19	Beatrice, Courtne	10055	59026	Productio	####	F	Single	4/4/2011	Active	Production	Elijiah Gray	Google Search	5	5	12
20	Becker, Renee	10245	1E+05	Database	####	F	Single	7/7/2014	Terminated for Cau	IT/IS	Simon Roup	Google Search	4.5	4	8
21	Becker, Scott	10277	53250	Productio	####	M	Single	7/8/2013	Active	Production	Webster Butler	LinkedIn	4.2	4	13

Problem: Employee Management System

Which Data Structure should we use?

Solution 1: Employee Management System

We can use AVL Trees to store the information of the Employees.

Solution 1: Employee Management System

Employee ID will be the key using which we will create the Nodes.

Solution 1: Employee Management System

What will be the time Complexity of Insert, Retrieve and Delete Operations?

Solution 1: Employee Management System

What will be the time Complexity of Insert, Retrieve and Delete Operations?

	Insert	Retrieve	Delete
AVL Tree	$O(\log_2(n))$	$O(\log_2(n))$	$O(\log_2(n))$

Employee Management System

Is there any way in which we can perform all these CRUD operations in even faster time.

Employee Management System

Is there any way in which we can perform all these CRUD operations in even faster time.

In order to do this, we will need to know more about where the items might be when we go to look for them in the collection. If every item is where it should be, then the search can use a single comparison to discover the presence of an item.

Employee Management System

We can know the location of each element in the simple array data structure where keys (integers) can be used directly as an index to store values.

Employee Management System

We can know the location of each element in the simple array data structure where keys (integers) can be used directly as an index to store values.

This could work if the Employee IDs were assigned from 0 to 19 index. And if we had to search the information for the 15th employee then we just access the 15th index of the array and complete record of the employee would be accessed

Solution 2: Employee Management System

However, in cases where the keys are large and cannot be used directly as an index, we should use Hashing.

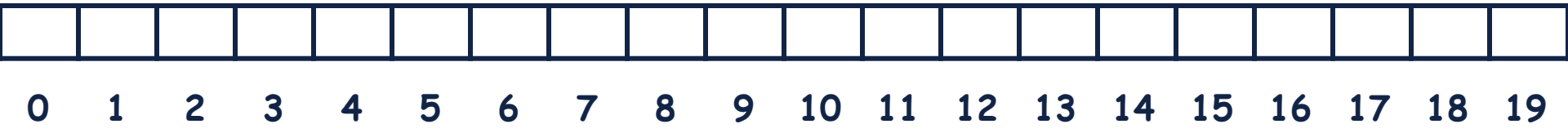
Solution 2: Employee Management System

However, in cases where the keys are large and cannot be used directly as an index, we should use Hashing.

In hashing, large keys are converted into small keys by using hash functions. The values are then stored in an Array data structure called hash table.

Hashing

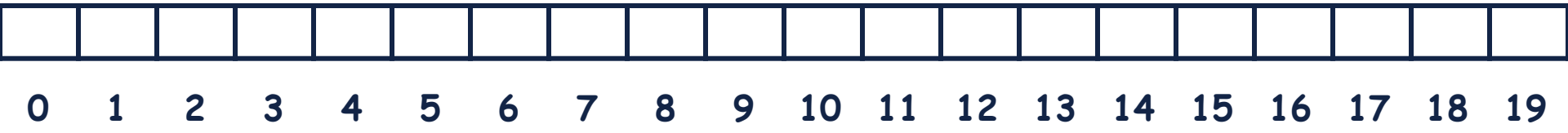
Let's declare an Array of Size 20.



Hashing

Let, the Hash Function we use is

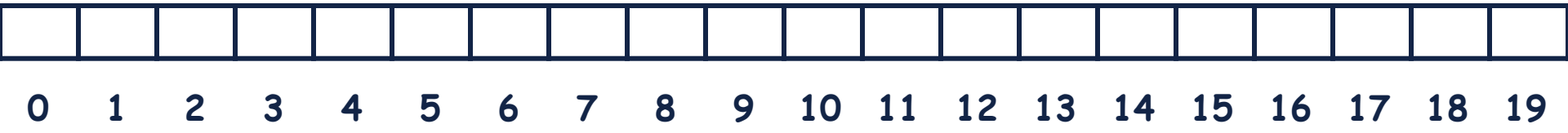
HashFunction = EmployeeID % 20



Hashing

Lets insert the data into the array according to the hashFunction.

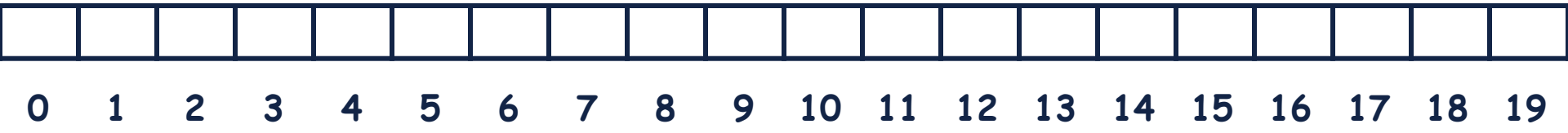
HashFunction = EmployeeID % 20



Hashing

Input: 10026

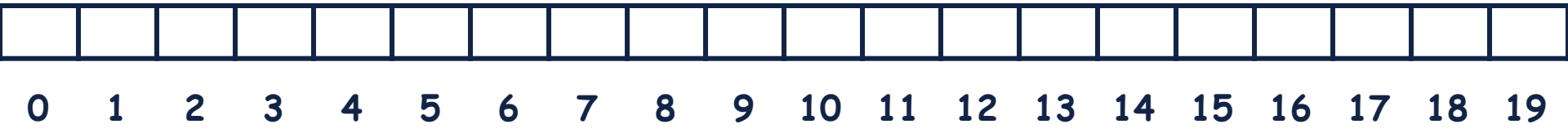
HashFunction = EmployeeID % 20



Hashing

Input: 10026

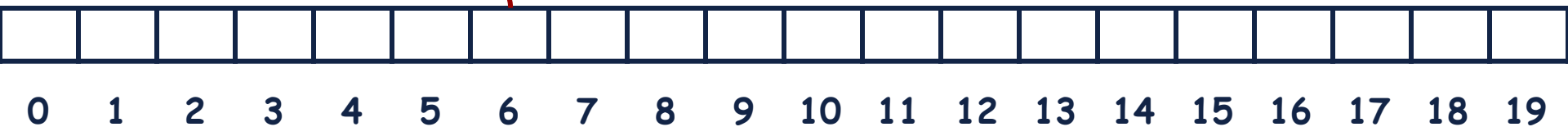
HashFunction = EmployeeID % 20



Hashing

Input: 10026

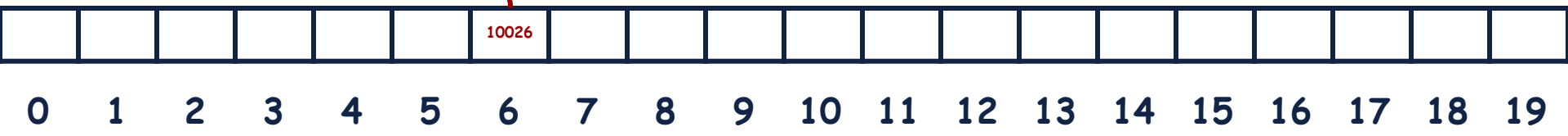
HashFunction = EmployeeID % 20



Hashing

Input: 10026

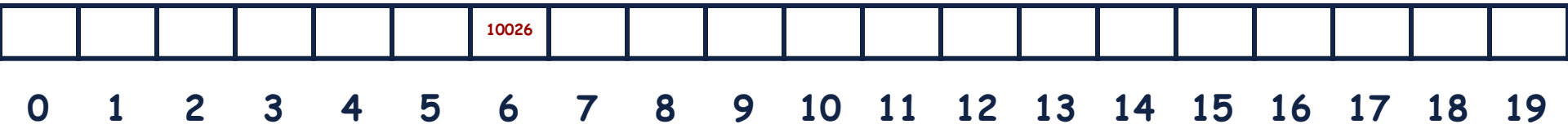
HashFunction = EmployeeID % 20



Hashing

Input: 10084

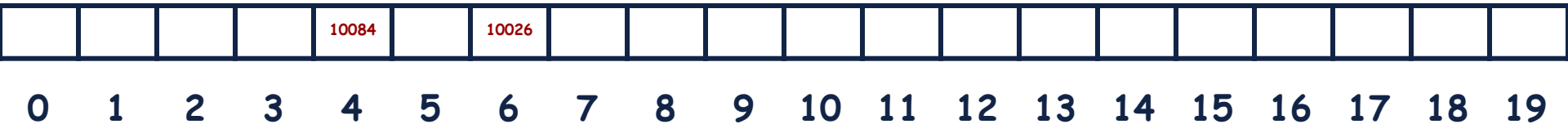
HashFunction = EmployeeID % 20



Hashing

Input: 10084

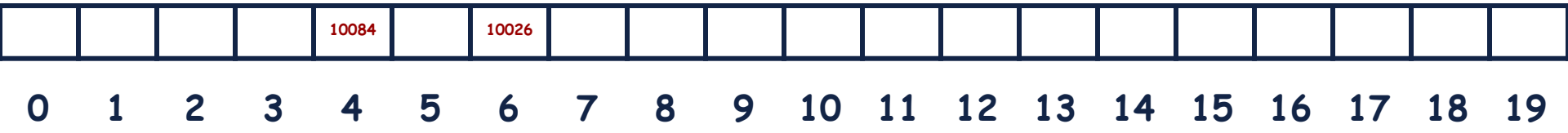
HashFunction = EmployeeID % 20



Hashing

Input: 10196

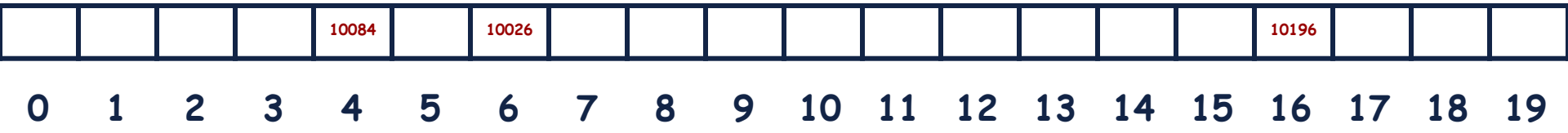
HashFunction = EmployeeID % 20



Hashing

Input: 10196

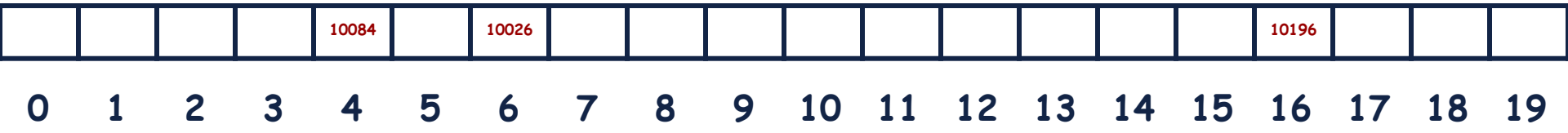
HashFunction = EmployeeID % 20



Hashing

Input: 10088

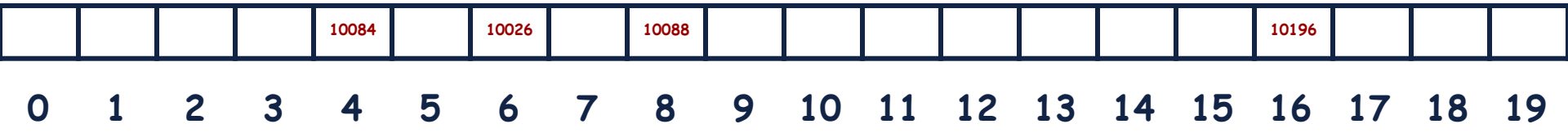
HashFunction = EmployeeID % 20



Hashing

Input: 10088

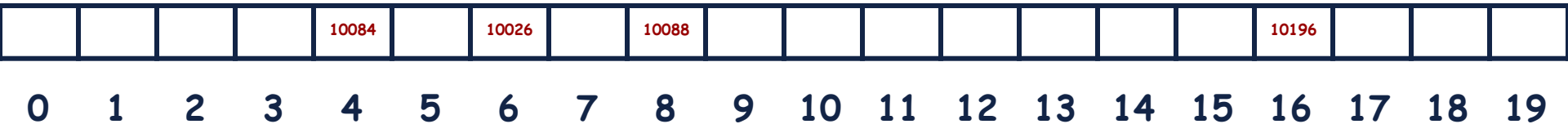
HashFunction = EmployeeID % 20



Hashing

Input: 10069

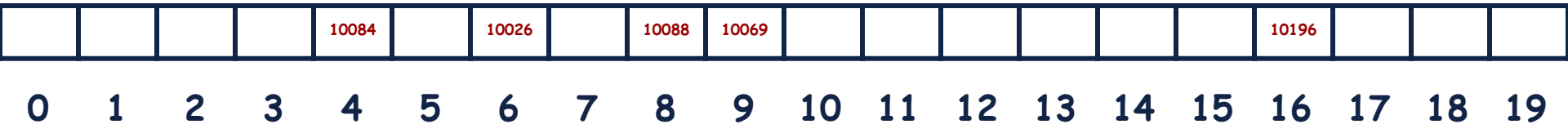
HashFunction = EmployeeID % 20



Hashing

Input: 10069

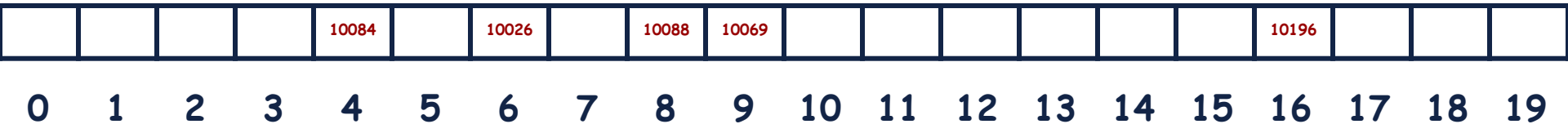
HashFunction = EmployeeID % 20



Hashing

Input: 10002

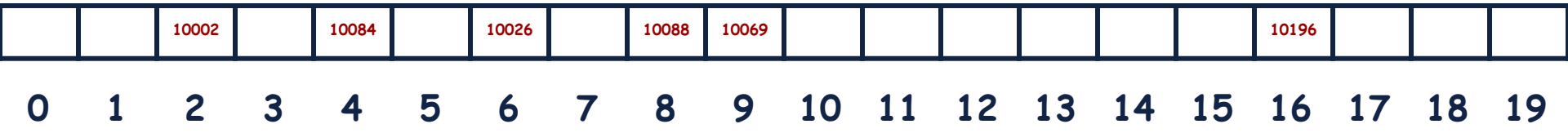
HashFunction = EmployeeID % 20



Hashing

Input: 10002

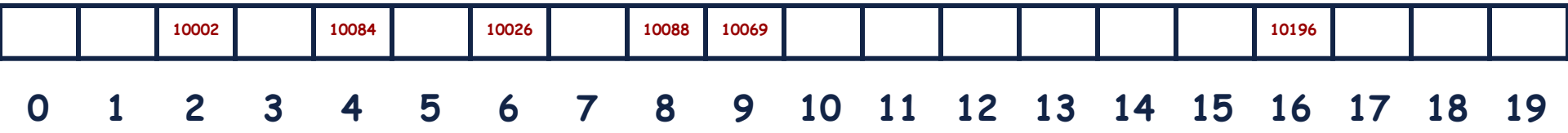
HashFunction = EmployeeID % 20



Hashing

Input: 10194

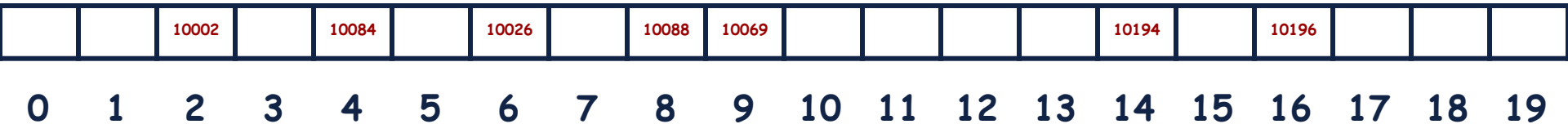
HashFunction = EmployeeID % 20



Hashing

Input: 10194

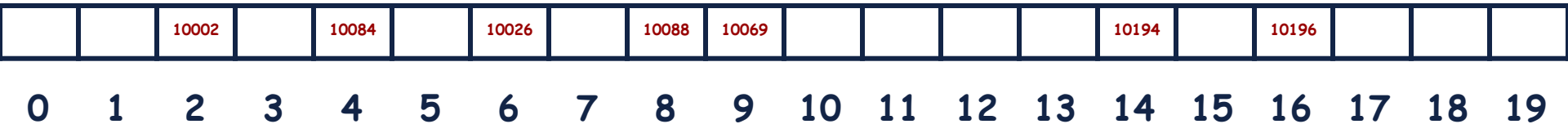
HashFunction = EmployeeID % 20



Hashing

Input: 10062

HashFunction = EmployeeID % 20



Hashing

Input: 10062

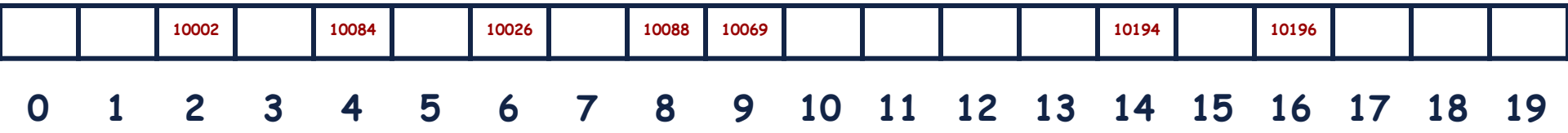
HashFunction = EmployeeID % 20

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
		10002		10084		10026		10088	10069					10194		10196			

Hashing

Input: 10062

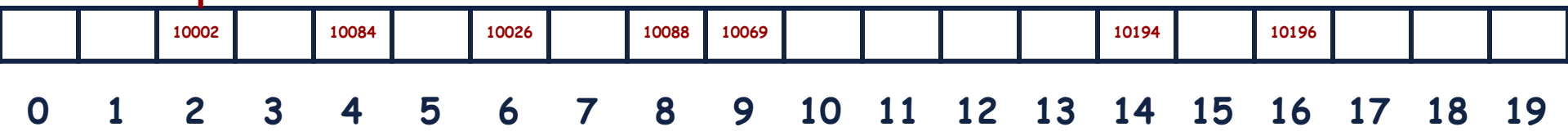
2 HashFunction = EmployeeID % 20



Hashing

Input: 10062

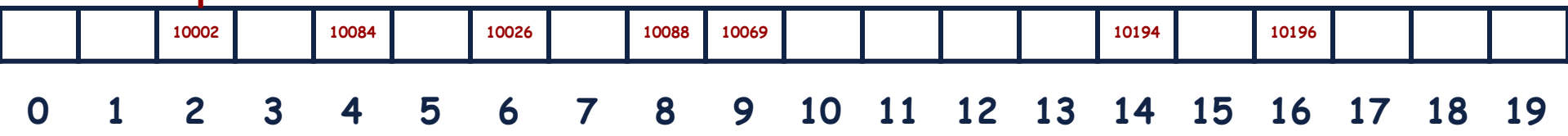
2 HashFunction = EmployeeID % 20



Hashing: Collision Occurred

Input: 10062

2 HashFunction = EmployeeID % 20

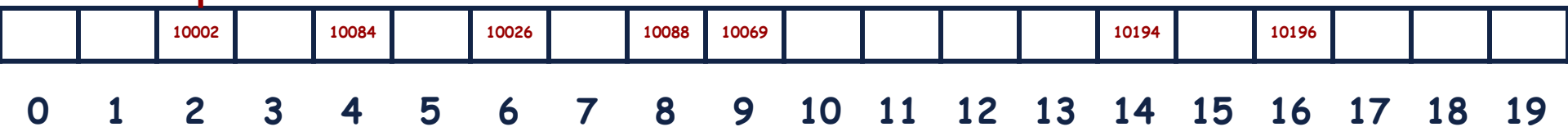


Hashing: Collision Occurred

Input: 10062

2 HashFunction = EmployeeID % 20

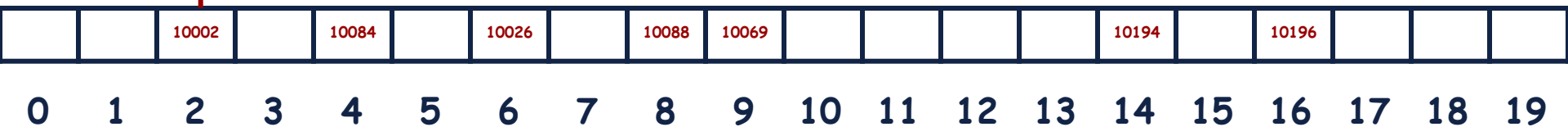
There are multiple Techniques for Collision Resolution



Collision Resolution: Linear Probing

Input: 10062

2 HashFunction = EmployeeID % 20

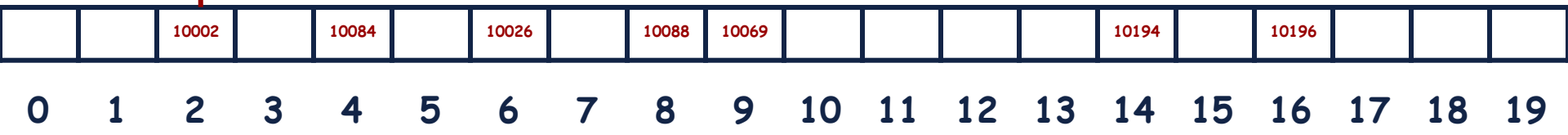


Collision Resolution: Linear Probing

Input: 10062

Find the next empty slot and insert the element.

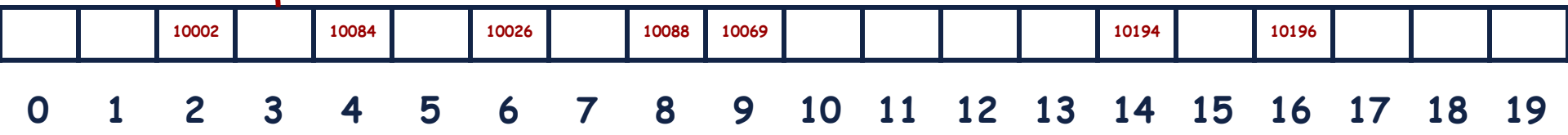
2 HashFunction = EmployeeID % 20



Hashing

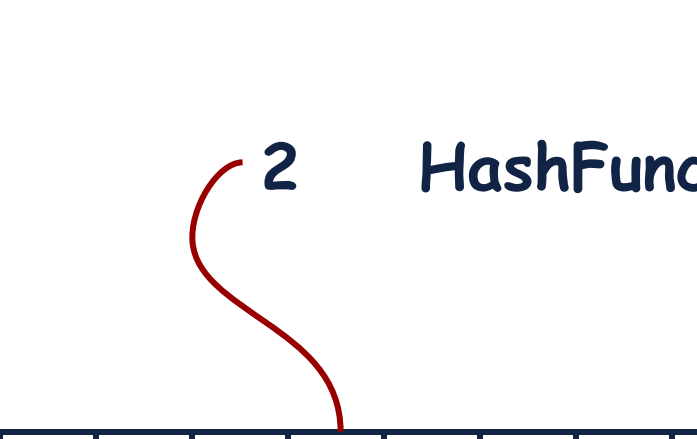
Input: 10062

2 HashFunction = EmployeeID % 20

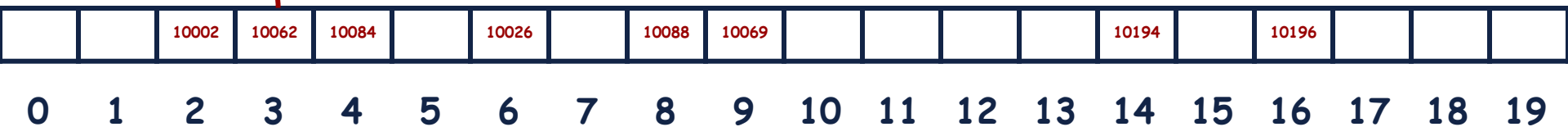


Hashing

Input: 10062



2 HashFunction = EmployeeID % 20

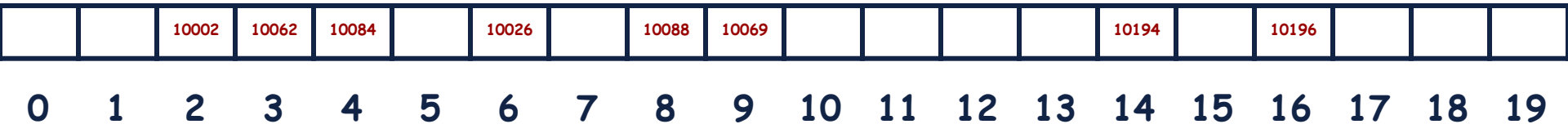


0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19

Hashing

Input: 10114

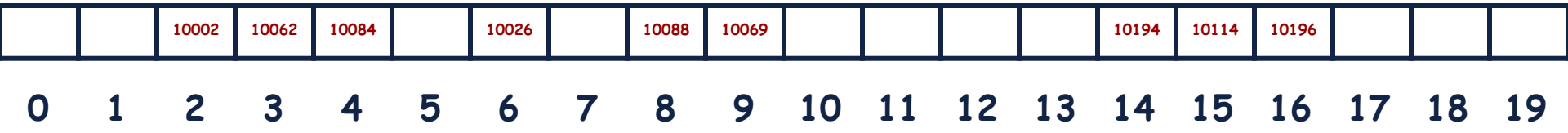
HashFunction = EmployeeID % 20



Hashing

Input: 10114

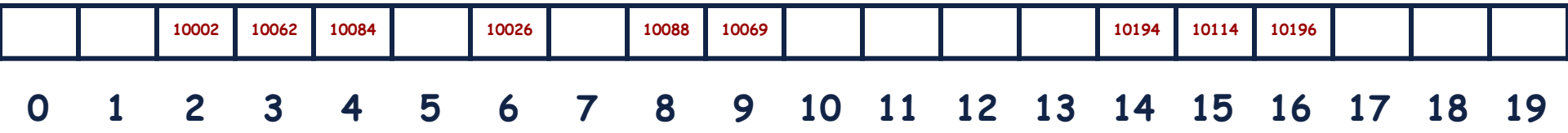
HashFunction = EmployeeID % 20



Hashing

Input: 10250

HashFunction = EmployeeID % 20



Hashing

Input: 10250

HashFunction = EmployeeID % 20

Hashing

Input: 10252

HashFunction = EmployeeID % 20

Hashing

Input: 10252

HashFunction = EmployeeID % 20

Hashing

Input: 10242

HashFunction = EmployeeID % 20

Hashing

Input: 10242

HashFunction = EmployeeID % 20

Hashing

Input: 10012

HashFunction = EmployeeID % 20

Hashing

Input: 10012

HashFunction = EmployeeID % 20

Hashing

Input: 10265

HashFunction = EmployeeID % 20

Hashing

Input: 10265

HashFunction = EmployeeID % 20

Hashing

Input: 10066

HashFunction = EmployeeID % 20

Hashing

Input: 10066

HashFunction = EmployeeID % 20

Hashing

Input: 10061

HashFunction = EmployeeID % 20

Hashing

Input: 10061

HashFunction = EmployeeID % 20

Hashing

Input: 10023

HashFunction = EmployeeID % 20

Hashing

Input: 10023

HashFunction = EmployeeID % 20

Hashing

Input: 10055

HashFunction = EmployeeID % 20

Hashing

Input: 10055

HashFunction = EmployeeID % 20

Hashing

Input: 10245

HashFunction = EmployeeID % 20

Hashing

Input: 10245

HashFunction = EmployeeID % 20

	10061	10002	10062	10084	10242	10026	10265	10088	10069	10250	10066	10252	10012	10194	10114	10196	10023	10055	10245
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19

Hashing

Input: 10277

HashFunction = EmployeeID % 20

	10061	10002	10062	10084	10242	10026	10265	10088	10069	10250	10066	10252	10012	10194	10114	10196	10023	10055	10245
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19

Hashing

Input: 10277

HashFunction = EmployeeID % 20

10277	10061	10002	10062	10084	10242	10026	10265	10088	10069	10250	10066	10252	10012	10194	10114	10196	10023	10055	10245
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19

Solution 2: Employee Management System

The process of hashing revolves around making retrieval of information faster.

Solution 2: Employee Management System

The process of hashing revolves around making retrieval of information faster.

Given the EmployeeID we just have to compute the hash value that corresponds to the EmployeeID and access that index in the Array.

Solution 2: Employee Management System

The process of hashing revolves around making retrieval of information faster.

Given the EmployeeID we just have to compute the hash value that corresponds to the EmployeeID and access that index in the Array.

In case of Collision we have to search the next records in the Array until the value is found or we find an empty space.

Hashing with Linear Probing

Let's code the solution.

Hashing: Implementation

```
struct Employee
{
    int empID;
    string empName;
    int salary;
    string position;
    string DOB;
    char gender;
    string maritalStatus;
    string dateOfHire;
    string employmentStatus;
    string department;
    string managerName;
    string recruitmentSource;
    float engagement;
    int empSatisfaction;
    int absences;
};
```

```
class hashTable
{
    Employee * arr[20] = {NULL};
    int size = 20;
public:

    int hashFunction(int value)
    {
        return value % size;
    }
```

Hashing: Implementation

```
void insert(Employee* e)
{
    int index = hashFunction(e->empID);
    if(arr[index] == NULL)
    {
        arr[index] = e;
    }
    else
    {
        while(arr[index] != NULL)
        {
            index = (index + 1) % size;
        }
        arr[index] = e;
    }
}
```

```
Employee *search(int id)
{
    int index = hashFunction(id);
    for (int x = 0; x < 20; x++)
    {
        if (arr[index] != NULL)
        {
            if (arr[index]->empID == id)
                return arr[index];
            else
                index = (index + 1) % size;
        }
        else
            break;
    }
    return NULL;
}
```

Solution 2: Employee Management System

What is the Time Complexity of this Solution implemented using Hashing with Collision resolution Technique of Linear Probing?

Solution 2: Employee Management System

What is the Time Complexity of this Solution implemented using Hashing with Collision resolution Technique of Linear Probing?

Hashing with Linear Probing	Best	Average	Worst
Insert	O(1)	O(1)	O(n)
Retrieve	O(1)	O(1)	O(n)
Delete	O(1)	O(1)	O(n)

Hashing with Linear Probing: Disadvantages

- Hashing with Linear Probing needs to know beforehand how many elements are required to be stored.
- Due to the collision resolution technique with linear probing, next empty space becomes filled therefore, there is more chances of collisions.

Hashing with Chaining

There is another technique for collision resolution in Hashing.
Chaining technique involves the use of chaining of entries in the hash table using linked lists if a collision occurs.

Hashing with Chaining

Input: 10062

HashFunction = EmployeeID % 20

Hashing with Chaining

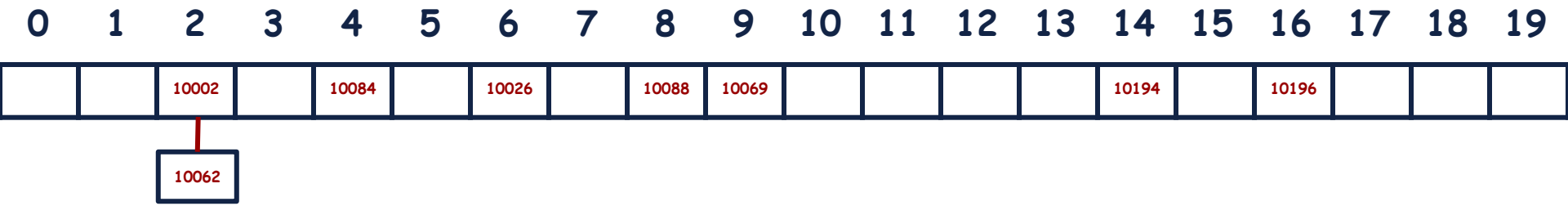
Input: 10062

2 HashFunction = EmployeeID % 20

Hashing with Chaining

Input: 10062

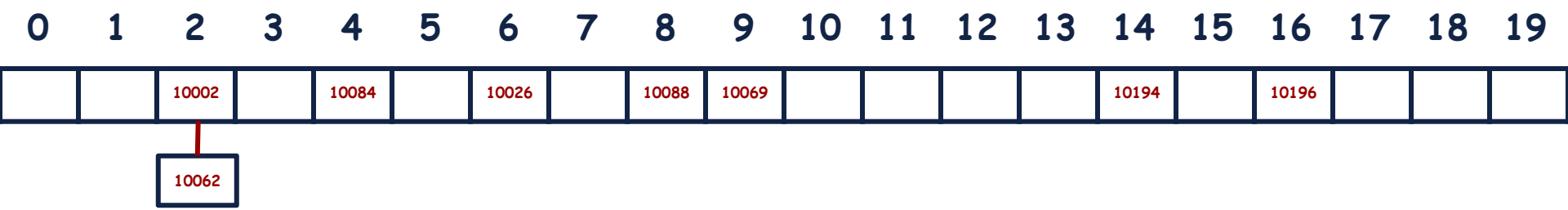
2 HashFunction = EmployeeID % 20



Hashing with Chaining

Input: 10114

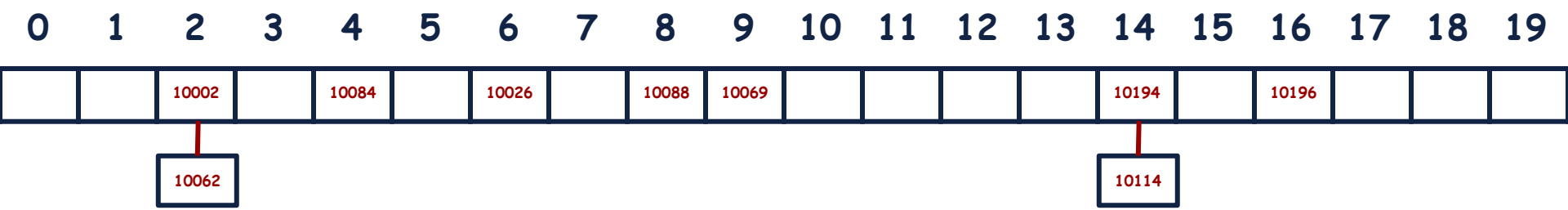
HashFunction = EmployeeID % 20



Hashing with Chaining

Input: 10114

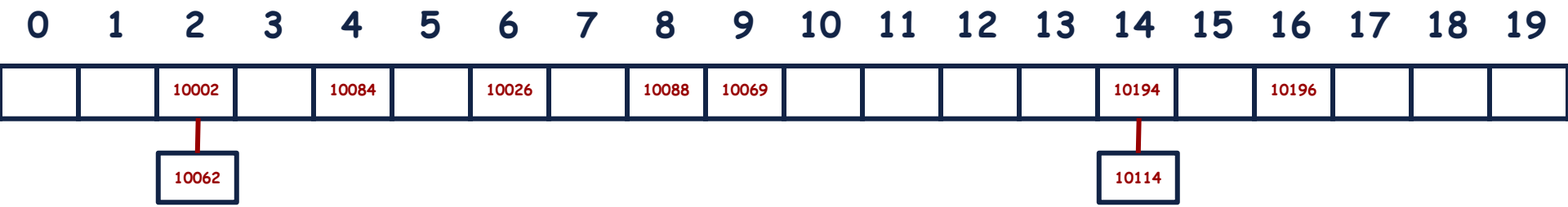
HashFunction = EmployeeID % 20



Hashing with Chaining

Input: 10250

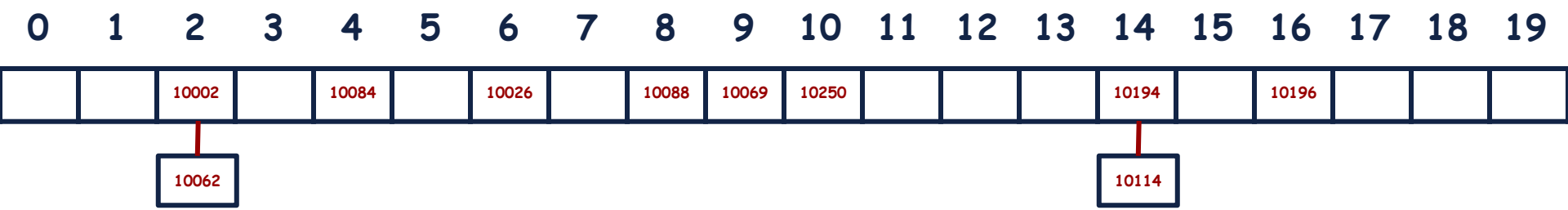
HashFunction = EmployeeID % 20



Hashing with Chaining

Input: 10250

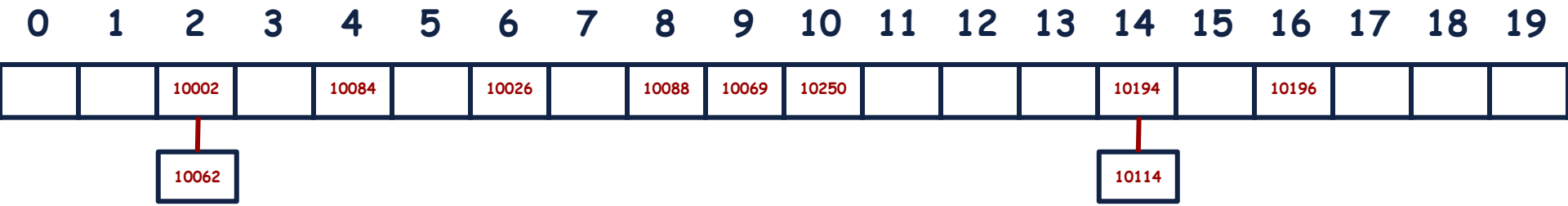
HashFunction = EmployeeID % 20



Hashing with Chaining

Input: 10252

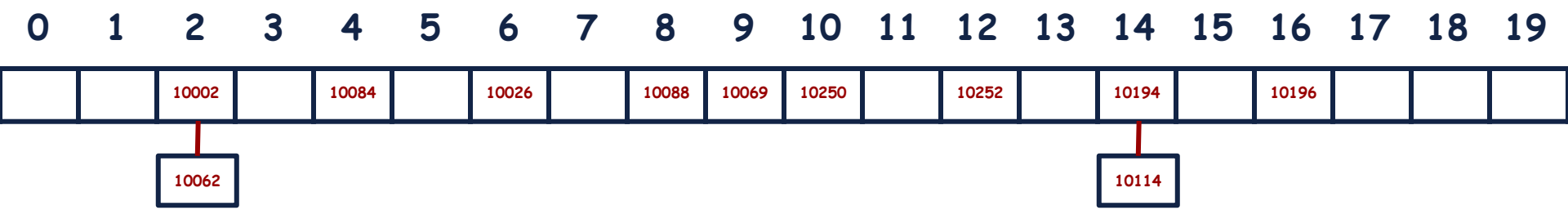
HashFunction = EmployeeID % 20



Hashing with Chaining

Input: 10252

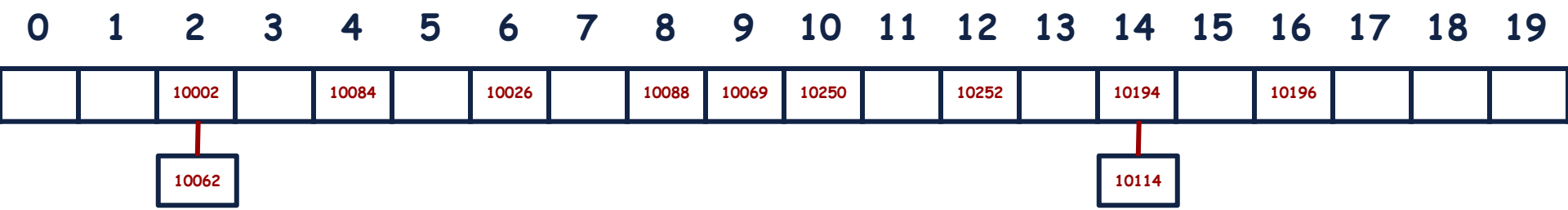
HashFunction = EmployeeID % 20



Hashing with Chaining

Input: 10242

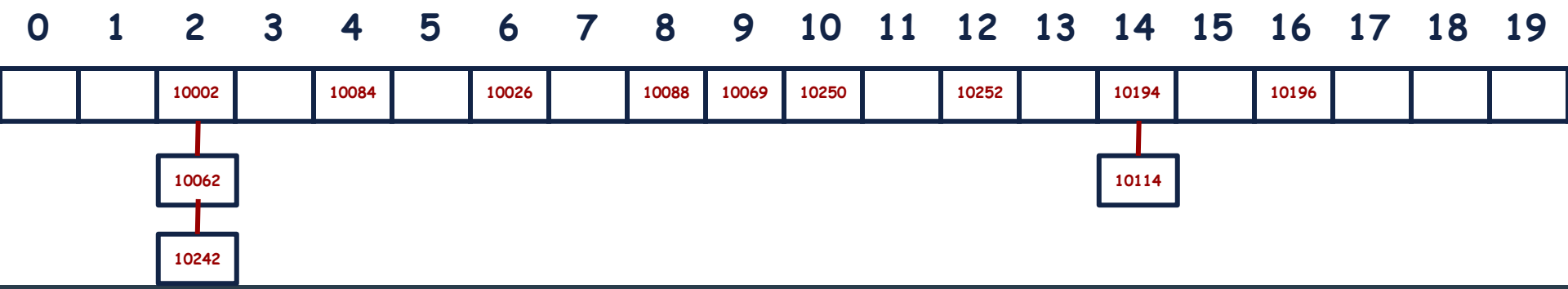
HashFunction = EmployeeID % 20



Hashing with Chaining

Input: 10242

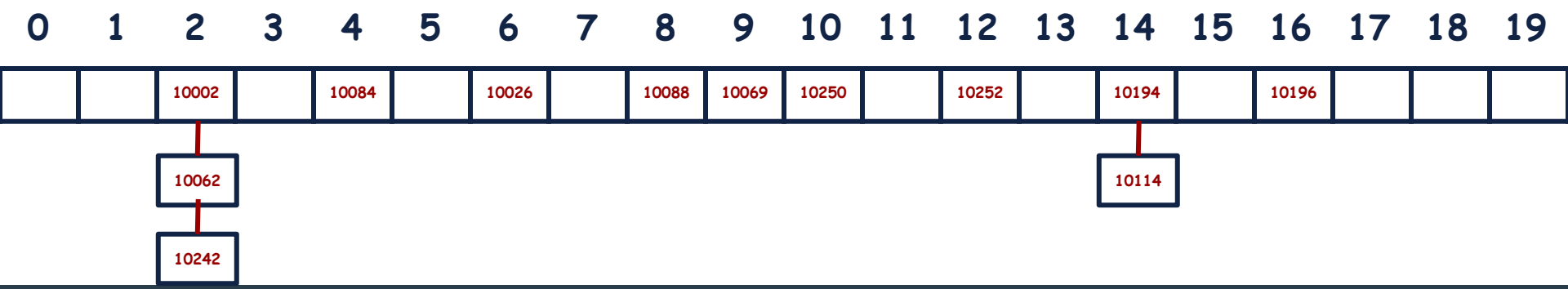
HashFunction = EmployeeID % 20



Hashing with Chaining

Input: 10012

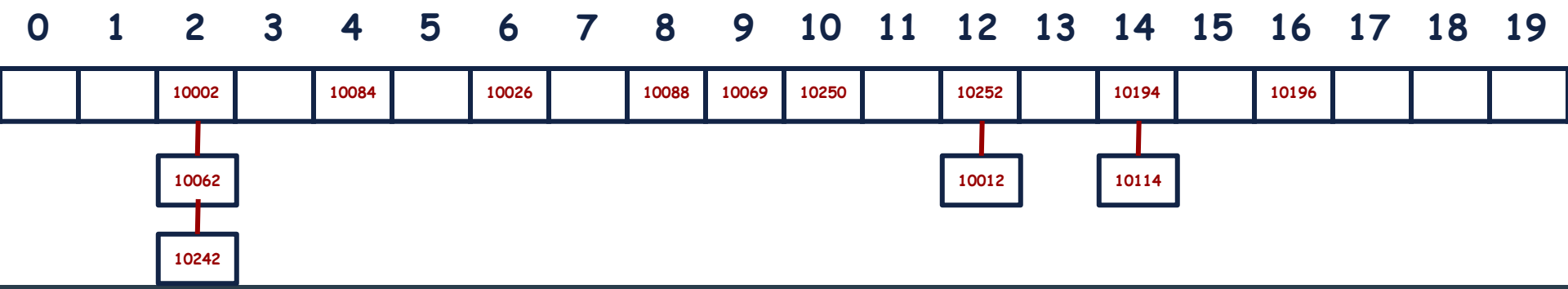
HashFunction = EmployeeID % 20



Hashing with Chaining

Input: 10012

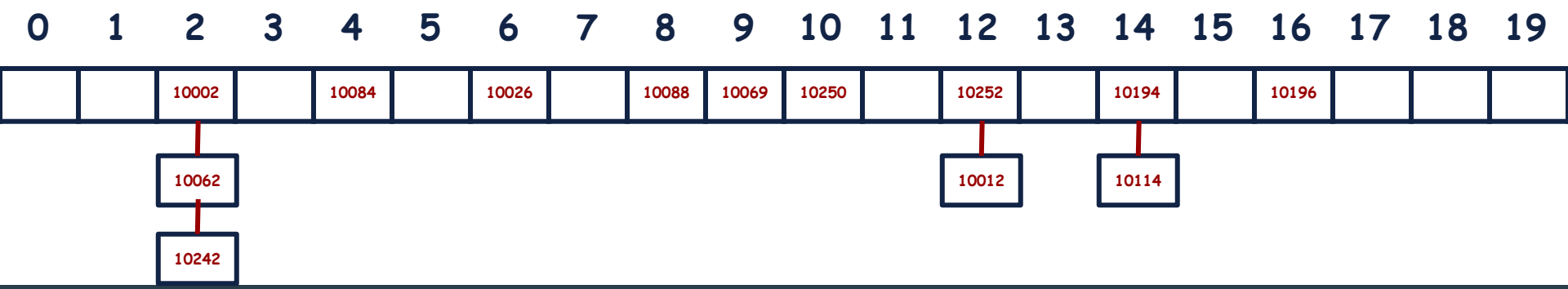
HashFunction = EmployeeID % 20



Hashing with Chaining

Input: 10265

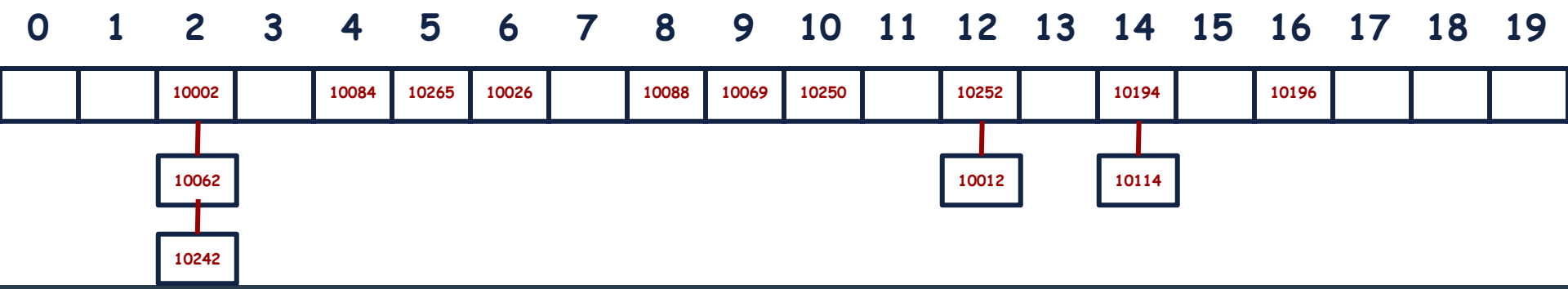
HashFunction = EmployeeID % 20



Hashing with Chaining

Input: 10265

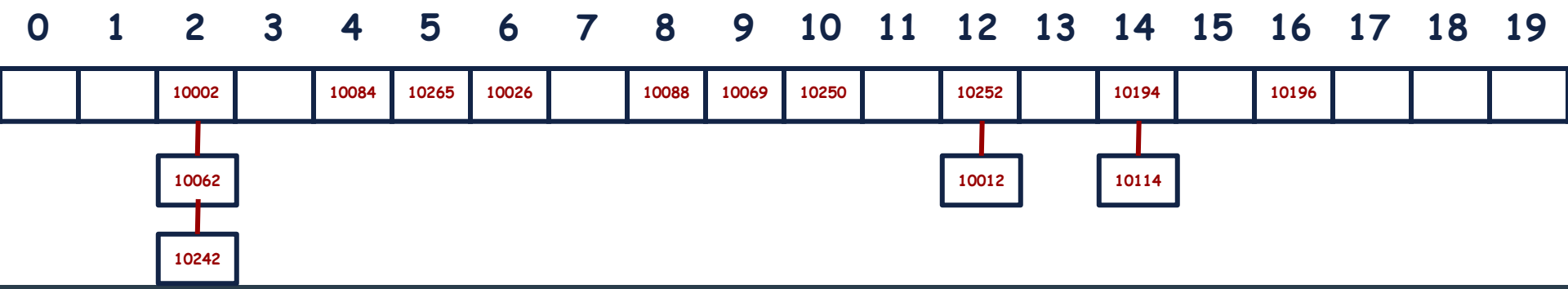
HashFunction = EmployeeID % 20



Hashing with Chaining

Input: 10066

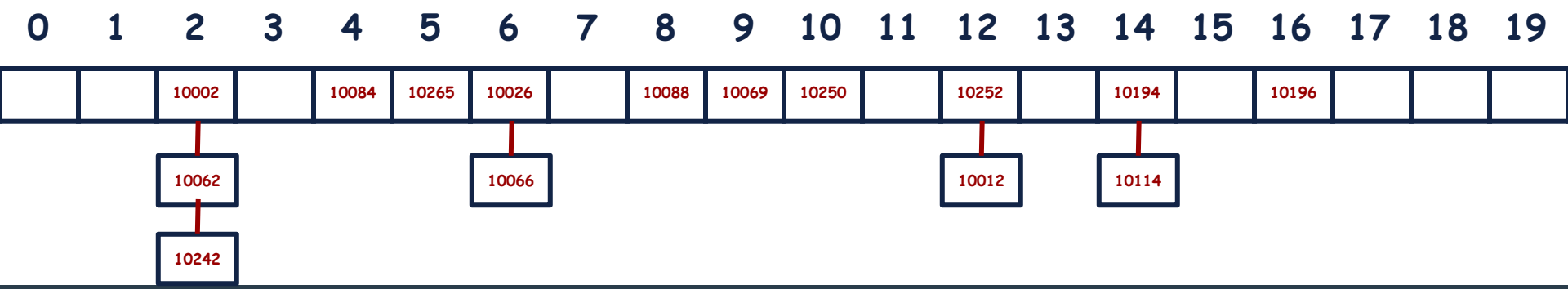
HashFunction = EmployeeID % 20



Hashing with Chaining

Input: 10066

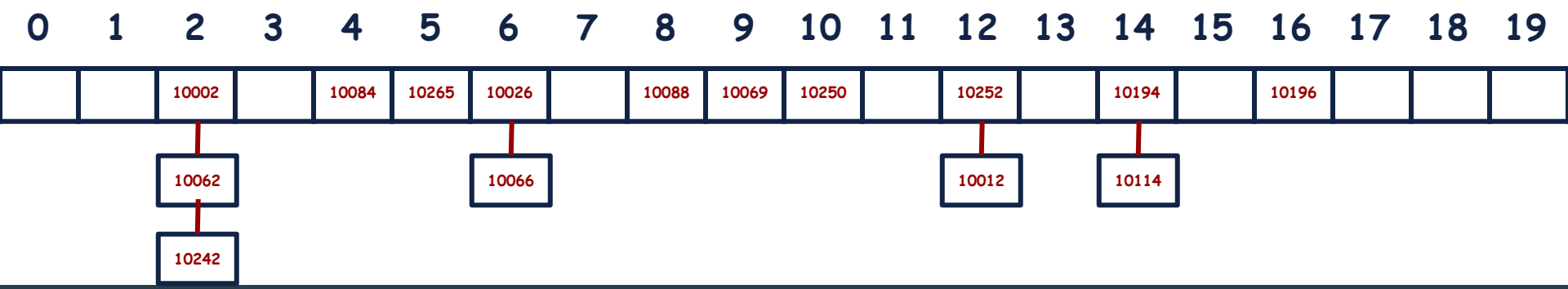
HashFunction = EmployeeID % 20



Hashing with Chaining

Input: 10061

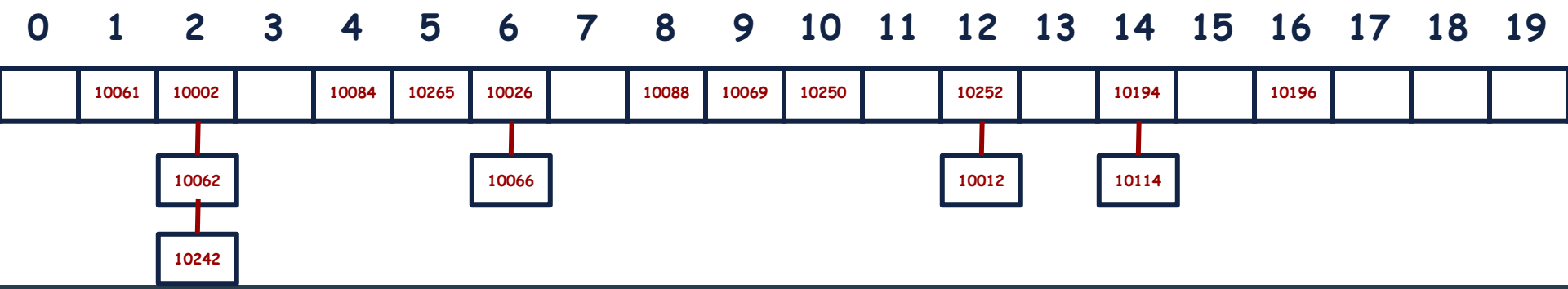
HashFunction = EmployeeID % 20



Hashing with Chaining

Input: 10061

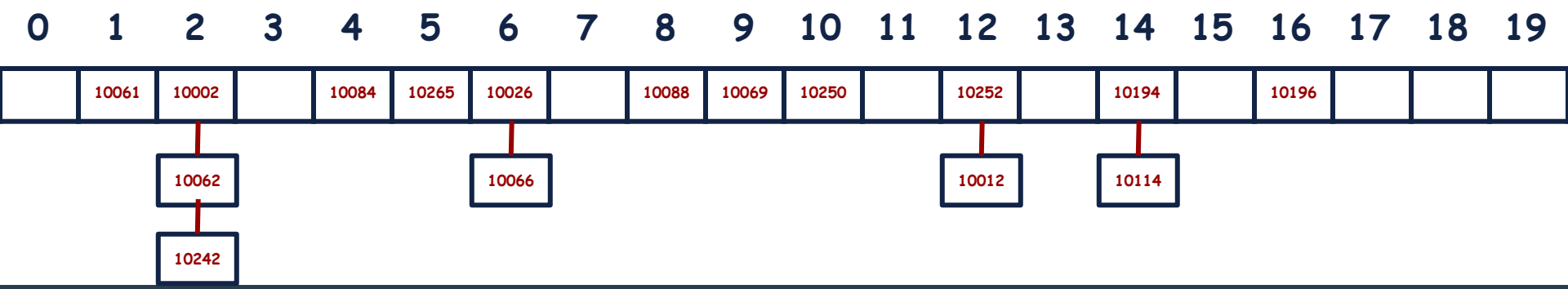
HashFunction = EmployeeID % 20



Hashing with Chaining

Input: 10023

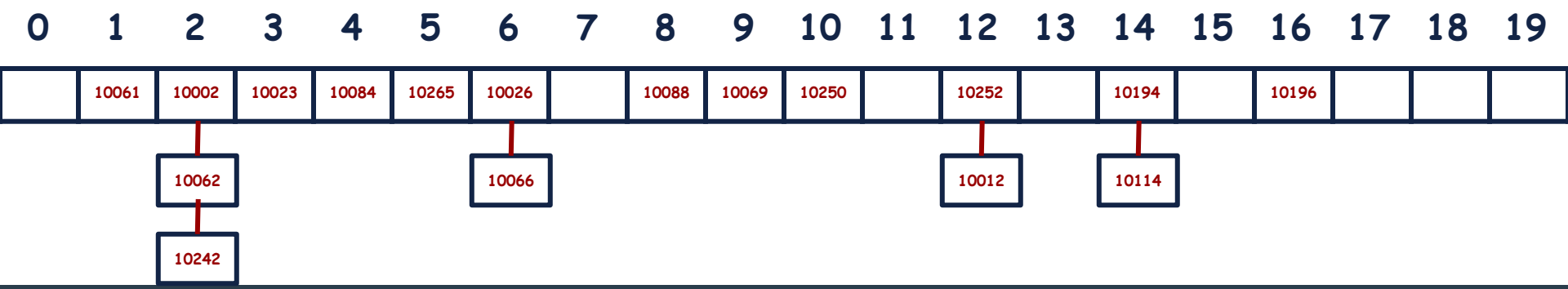
HashFunction = EmployeeID % 20



Hashing with Chaining

Input: 10023

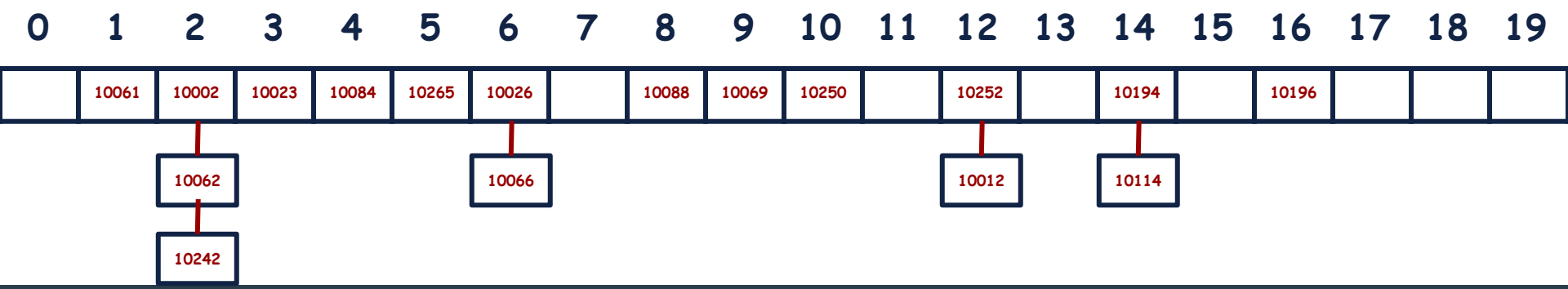
HashFunction = EmployeeID % 20



Hashing with Chaining

Input: 10055

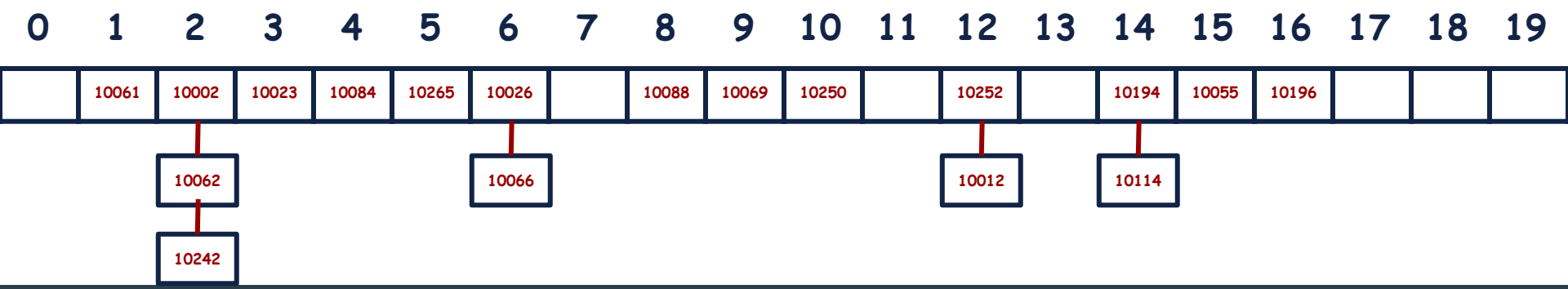
HashFunction = EmployeeID % 20



Hashing with Chaining

Input: 10055

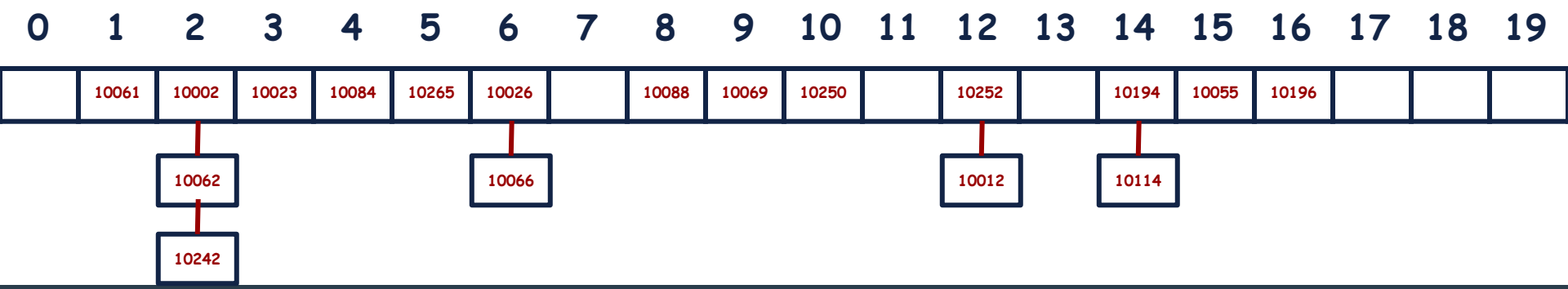
HashFunction = EmployeeID % 20



Hashing with Chaining

Input: 10245

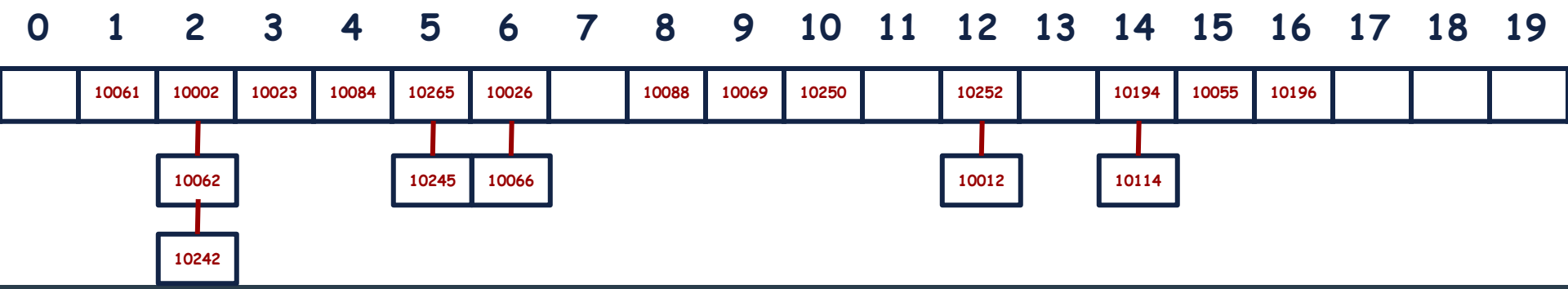
HashFunction = EmployeeID % 20



Hashing with Chaining

Input: 10245

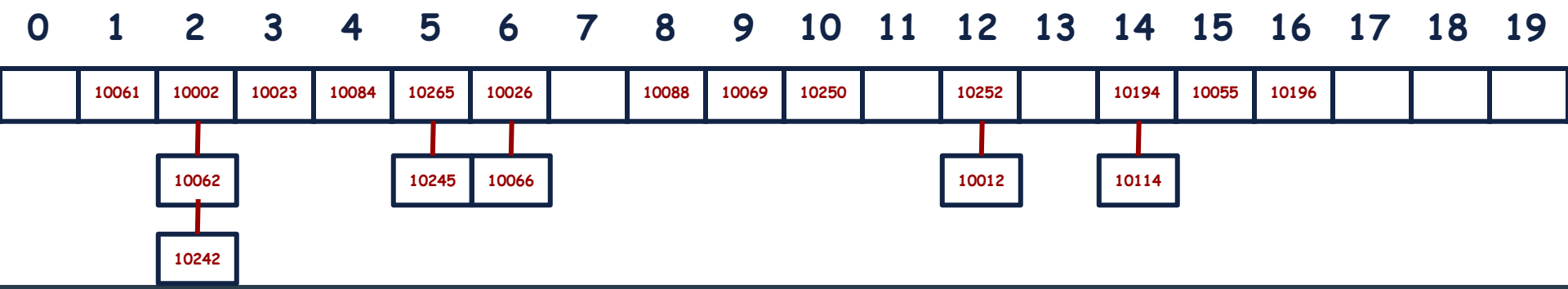
HashFunction = EmployeeID % 20



Hashing with Chaining

Input: 10277

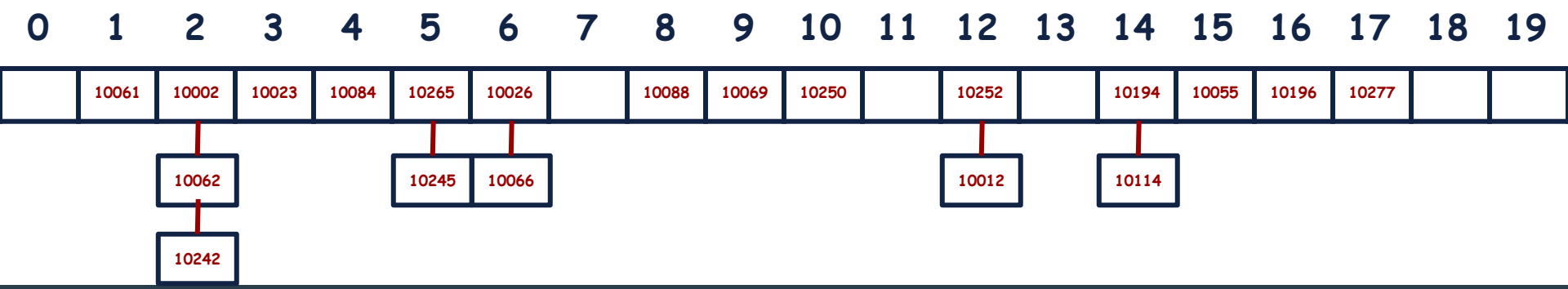
HashFunction = EmployeeID % 20



Hashing with Chaining

Input: 10277

HashFunction = EmployeeID % 20



Hashing with Chaining

Let's code the solution.

Hashing: Implementation

```
struct Employee{
    int empID;
    string empName;
    int salary;
    string position;
    string DOB;
    char gender;
    string maritalStatus;
    string dateOfHire;
    string employmentStatus;
    string department;
    string managerName;
    string recruitmentSource;
    float engagement;
    int empSatisfaction;
    int absences;
    Employee *next;
};
```

```
class hashTable
{
    Employee * arr[20] = {NULL};
    int size = 20;
public:

    int hashFunction(int value)
    {
        return value % size;
    }
```

Hashing: Implementation

```
void insert(Employee *e)
{
    int index = hashFunction(e->empID);
    if (arr[index] == NULL)
    {
        arr[index] = e;
    }
    else
    {
        Employee *temp = arr[index];
        while (temp->next != NULL)
        {
            temp = temp->next;
        }
        temp->next = e;
    }
}
```

```
Employee * search(int id)
{
    int index = hashFunction(id);
    if (arr[index]->empID == id)
        return arr[index];
    else
    {
        Employee *temp = arr[index]->next;
        while (temp != NULL)
        {
            if (temp->empID == id)
                return temp;
            temp = temp->next;
        }
    }
    return NULL;
}
```

Hashing with Chaining

What is the Time Complexity of this Solution implemented using Hashing with Collision resolution Technique of Chaining?

Hashing with Chaining

What is the Time Complexity of this Solution implemented using Hashing with Collision resolution Technique of Chaining?

Hashing with Chaining	Best	Average	Worst
Insert	O(1)	O(1)	O(n)
Retrieve	O(1)	O(1)	O(n)
Delete	O(1)	O(1)	O(n)

Chaining VS Linear Probing

Chaining	Linear Probing
Chaining is requires less computation to insert an element.	Linear Probing requires more computation to insert an element due to Clustering.
In chaining, Hash table never fills up, we can always add more elements to chain.	In Linear Probing, table may become full.
Chaining is mostly used when it is unknown how many keys may be inserted or deleted.	Linear Probing is used when the number of keys is known.
Cache performance of chaining is not good as keys are stored using linked list.	Linear Probing provides better cache performance as everything is stored in the same table.
Some slots of hash table in chaining are never used.	In Linear Probing, a slot can be used even if an input doesn't map to it.
Chaining uses extra space for links.	No links in Linear Probing.

Learning Objective

Students should be able to use **Hashing** for efficient CRUD operations.

Self Assessment 01

A hash table of length 10 with hash function h(k)=k mod 10, and linear probing. After inserting 6 values into an empty hash table

0	
1	
2	42
3	23
4	34
5	52
6	46
7	33
8	
9	

Which one of the following choices gives a possible order in which the key values could have been inserted in the table?

- 46, 42, 34, 52, 23, 33
- 34, 42, 23, 52, 33, 46
- 46, 34, 42, 23, 52, 33
- 42, 46, 33, 23, 34, 52

Self Assessment 02

The keys 12, 18, 13, 2, 3, 23, 5 and 15 are inserted into an initially empty hash table of length 10 using open addressing with hash function h(k) = k mod 10 and linear probing. What is the resultant hash table?

0	
1	
2	2
3	23
4	
5	15
6	
7	
8	18
9	

0	
1	
2	12
3	13
4	
5	5
6	
7	
8	18
9	

0	
1	
2	12
3	13
4	2
5	3
6	23
7	5
8	18
9	15

0	
1	
2	12,2
3	13,3,23
4	
5	5,15
6	
7	
8	18
9	