C++
Hash Tables

CO658 Data Structures & Algorithms

Topics

- Associate Arrays
- Hash Table
- Hashing
- Collisions

Introduction

- A method of using a table to store data which requires frequent access
- The data itself indicates its position in the table
- A keyed array data structure (associative array).
- Overcomes C/C++ lack of support for keyed arrays.

Traditional C approach

```
gameObject[10];
```

Associate array

```
gameObject[player];
```

Enum Types

- The enum type can provide a readable solution to accessing arrays.

```
class GameObject{
    public:string name;
};

enum TGameObject { player = 0, ai = 1, background=2 };


int main(){
    GameObject objects[10];
    objects[TGameObject::player].name = "Master Chief";
    return 0;
}
```

- The TGameObject enum types are implicity assigned an integer value player=0, ai = 1, backGround = 2. Which can be used as the index.

Hash Table

- The enum approach is however reliant on the keys being known at compile time.
- A more flexible approach would be to create the keys dynamically and map them onto an index within the array holding the value (Hashing).
- Transforming the key into an index is performed by a **Hash Function**.

Index	Value
0	
1	
2	
3	
4	
5	
6	

Keys

- Traditional approach is to use an integer as the key but this has no meaning. What does objects[2] hold? Far better to write objects[player].
- To use a string as a key we must be able to map it to an index value (integer). As when using enum player = 0.
- We could use the ASCII code for each character to determine the array index.
- In C we can cast a character to return its ASCII value.

```
(int)'a';
```

- Casting the character 'a' gives the ASCII value 97.
- Alternatively we could cast an element of a string

```
string key = "player";
cout << (int)key[0] << endl;
```

- Displays 112.

Keys

- The string keys are composed of multiple characters, so how do we combine them?
- One approach would be to add the ASCII values for each character in the string.

```
string key = "player";
int index = (int)key[0] + (int)key[1] + (int)key[2] + (int)key[3] + (int)key[4] + (int)key[5];
```

- Giving and index of 653.
- Assuming we limit the number of characters in the key string to 6, it gives us 1536 (255 * 6) elements in the array.
- Problem is that there will be duplicates. The key “ai” and “ia” have the same index!

Hashing

- We need a means of ensuring the index generated by the key is unique.
- One approach would be to multiply each character's ASCII value by the power of its position to the base of 255.
- The analogy being decimal numbers

```
8421
Is equivalent to
8 * 10^3 = 8,000
4 * 10^2 =    400
2 * 10^1 =     20
1 * 10^0 =      1
```

- Apply this to our key

```
“ai";
int(‘a’) * 255^1 = 97 * 255
int(‘i’) * 255^0 =  105
                 =  24840
```

Hashing

- This approach would give us a huge range of index values

```
255^1 + 255^2 + 255^3 + 255^4 + 255^5 + 255^6
```

- In fact it will cause an overflow.
- As we only require the letters ‘a’ .. ‘z’ not the full set of ASCII characters we could assign a unique id to each letter.

```
a = 1; b = 2;c = 3;
```

- This would give use the index range 402,321,276

```
27^1 + 27^2 + 27^3 + 27^4 + 27^5 + 27^6
```

- It is unlikely that we would want an array this size and indeed would not have enough memory!

Hashing

- If we estimated that we would only have 500 items in our array we need to reduce the range of values from 0 to 402,321,276 down to 0 to 499.
- We could achieve this by apply the modulus operator

```
HashKey % Number Elements
```

- This returns a number in the range 0..(Elements -1).
- In practice Hash Tables should be defined with twice the maximum number of values to be stored within them.

Activity

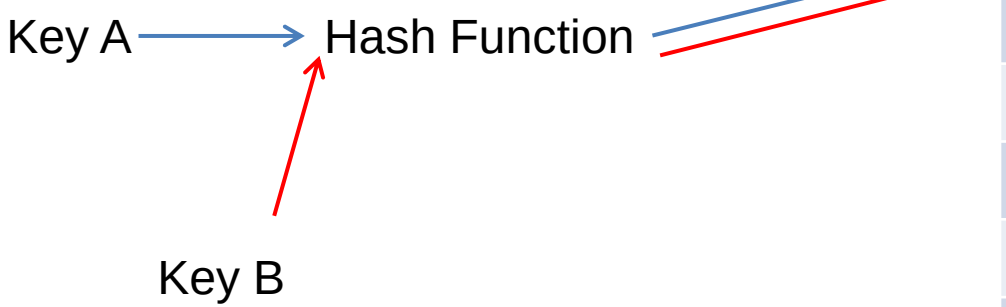
- Attempt Exercise 1

Open / Closed

- While the goal of a hash function is to minimize collisions, some collisions are unavoidable in practice.
- Thus, hashing implementations must include some form of collision resolution policy.
- Collision resolution techniques can be broken into two classes: **open hashing** and **closed hashing**.
- The difference between the two has to do with whether collisions are stored outside the table (open hashing), or whether collisions result in storing one of the records at another slot in the table (closed hashing).

Collisions

- Two keys could map to the same index (**Collide**).
- As when inserting we may find the element occupied!
- One solution (**Open Addressing**) would be to find an alternative location within the array based on some algorithm.
- Or an alternative solution (**Separate Chaining**), is for each element to maintain a Linked List and insert the data into the Linked List.



Index	Value
0	
1	
2	
3	
4	
5	
6	

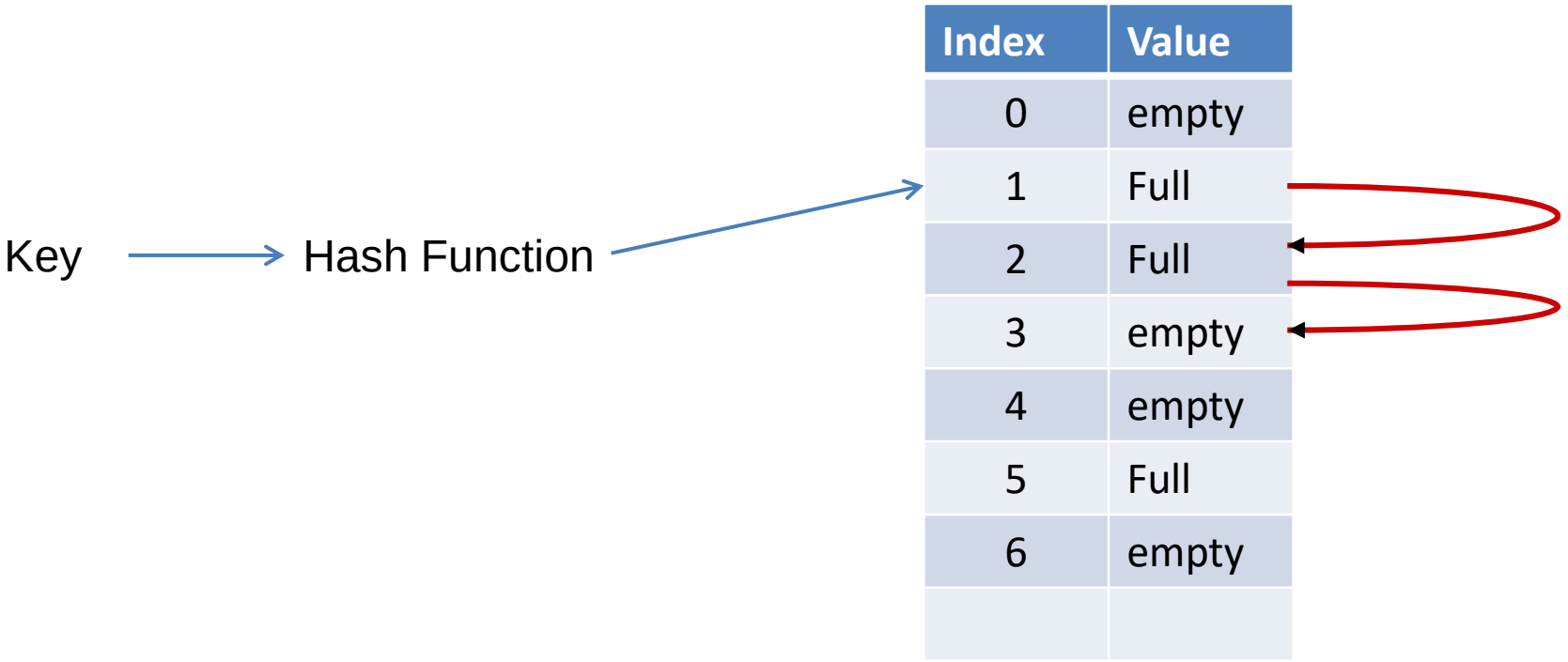
Open Addressing

- Three solutions to collisions using Open Addressing
  1. **Linear Probing** : Insertion by incrementing index (Probing) until vacant element found. e.g. If x is index Probed, indexes would be x+1, x+2, x+3 etc. But this can lead to **primary clustering**. Once established clusters grow fast as indexes are used up and leading to degrading performance
  2. **Quadratic Probing** : Avoids clustering by stepping the probe e.g. x + 1, x + 4, x + 9, x + 16. The distance from the initial probe is the square of the step number. X+1^2, x+2^2, x+3^2. Disadvantage is that keys mapped to the same index will apply the same algorithm (**secondary clustering**).
  3. **Double Hashing :** Colliding keys are hashed a second time using a different hash function. The value returned is the step size. A common algorithms is

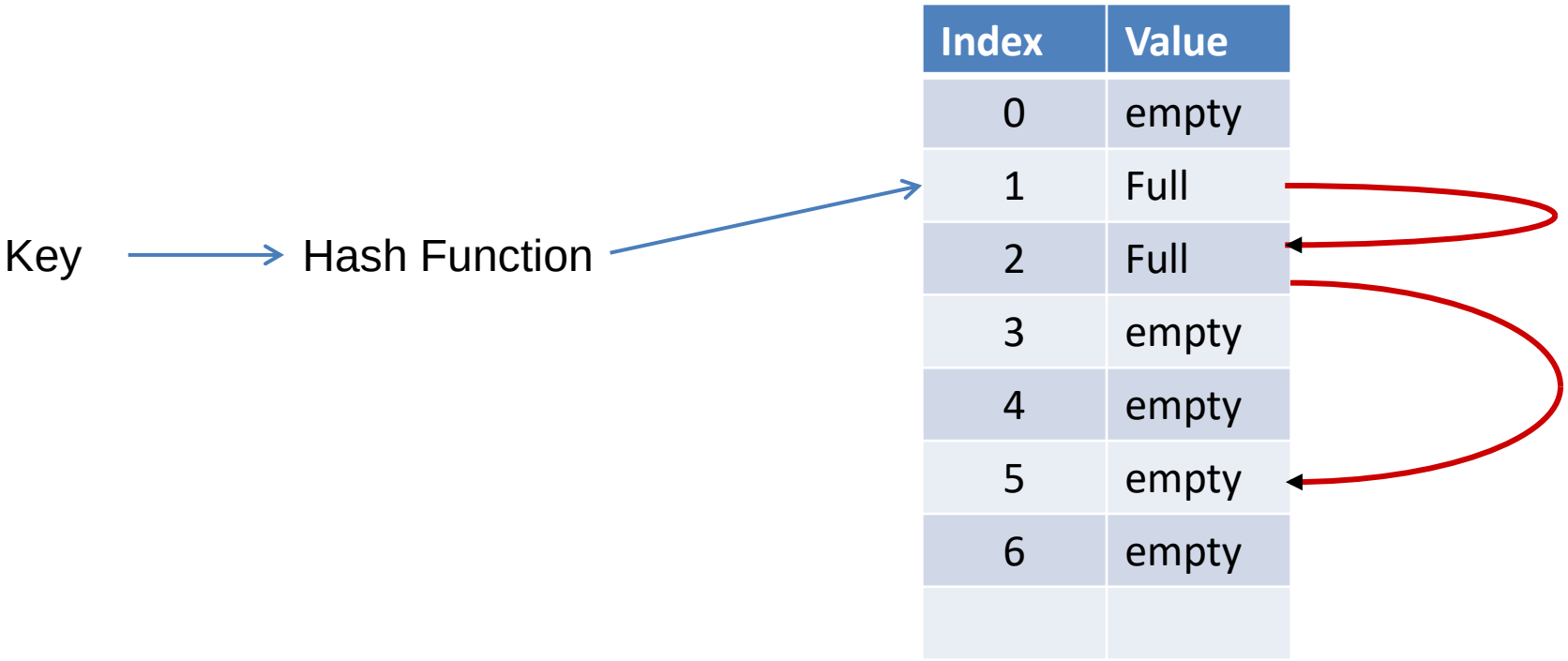
     step = constant – (key % constant)

     Where constant is prime and less than the array size.

Linear Probing



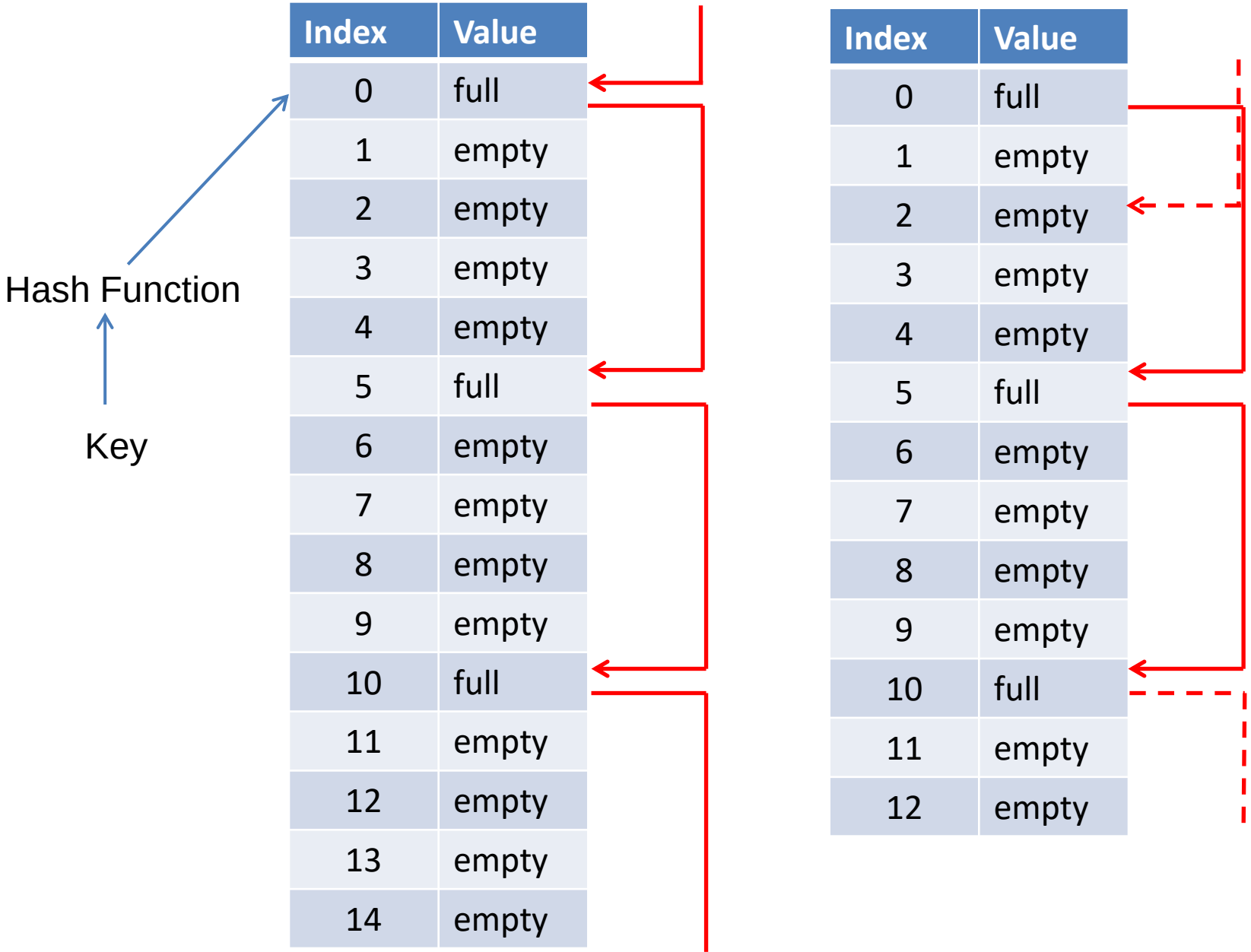
Quadratic Probing



Importance of Prime

- The Hash Table size must be a prime number to ensure a free element is found.
- Example 1 : **non prime size** of 15 (index 0 to 14). The initial index is 0 and step size of 5 and 0, 5 and 10 elements are full
- The Probe sequence is 0, 5, 10, 0, 5, 10 and repeats never finding a free element!
- Example 2 : **prime size is 13** (index 0 to 12). The initial index is 0 and step size of 5 and 0, 5 and 10 elements are full
- The Probe sequence is 0, 5, 10, 2, 7, 12 a free element will eventually be found assuming the table isn't full.

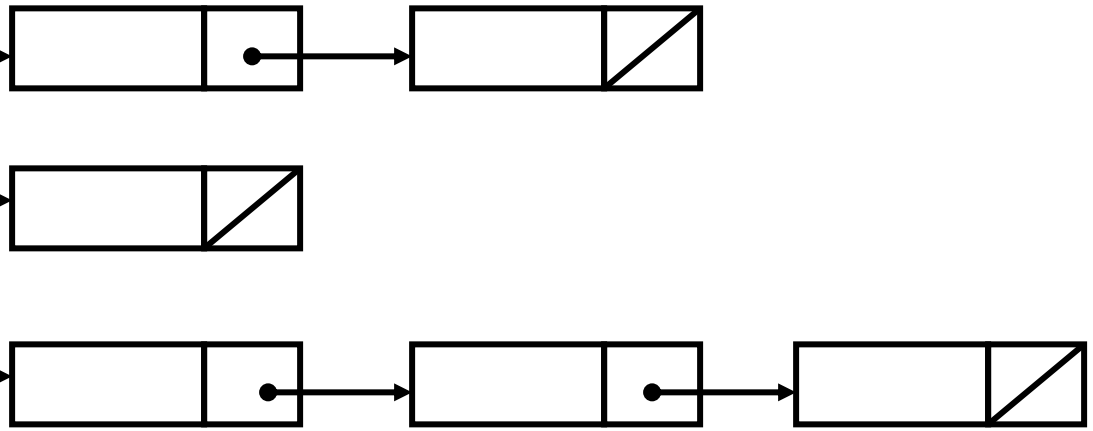
Importance of Prime



Separate Chaining

- Finding the initial index is fast O(1) but searching the list O(N).
- An alternative is to use a 2D array (2nd Dim is known as a **bucket**) but you must know the dimension in advance.

Index	Value
0	
1	
2	
3	
4	
5	
6	



Hash Function Performance

- Hash function should be simple as it will impact on operational speed.
- Use bitwise operators for performance.
- Ideally the Hash Function should distribute the data as evenly as possible and avoid clustering

Conclusion

- A data structure that offers fast insertion and searching O(1).
- Relatively easy to implement.
- Usually based on arrays so difficult to expand.
- Performance can degrade as the array becomes full / Clustered
- No ability to visit the data in order.