

# Hashing

# Hash Table

---

- As we saw with binary search, certain data structures such as a binary search tree can help improve the efficiency of searches.
- From linear search to binary search, we improved our search efficiency from  $O(n)$  to  $O(\log n)$  .
- We now present a new data structure, called a hash table, that will increase our efficiency to  $O(1)$  , or constant time.

# Hash Table

---

- A hash table is made up of two parts: an array (the actual table where the data to be searched is stored) and a mapping function, known as a hash function.
- The hash function is a mapping from the input space to the integer space that defines the indices of the array.
- In other words, the hash function provides a way for assigning numbers to the input data such that the data can then be stored at the array index corresponding to the assigned number.

# Hash Table

- Let's take a simple example. First, we start with a hash table array of strings (we'll use strings as the data being stored and searched in this example).
- Let's say the hash table size is 12:
- Next we need a hash function.

Hash Table(strings)	
0	(null)
1	(null)
2	(null)
3	(null)
4	(null)
5	(null)
6	(null)
7	(null)
8	(null)
9	(null)
10	(null)
11	(null)

Figure : The empty hash table of strings

# Hash Function

---

- There are many possible ways to construct a hash function.
- For now, let's assume a simple hash function that takes a string as input.
- The returned hash value will be the sum of the ASCII characters that make up the string mod the size of the table.

```
int hash(char *str, int table_size)
{
    int sum;
    if (str==NULL) return -1;
    for( ; *str!=null; str++) sum += *str;
    return (sum % table_size); }
```

# Hash Function

---

- As mentioned briefly in the previous section, there are multiple ways for constructing a hash function.
- Remember that hash function takes the data as input (often a string), and returns an integer in the range of possible indices into the hash table.
- Every hash function must do that, including the bad ones. So what makes for a good hash function?

# Characteristics of a Good Hash Function

---

- The hash value is fully determined by the data being hashed.
- The hash function uses all the input data.
- The hash function "uniformly" distributes the data across the entire set of possible hash values.
- The hash function generates very different hash values for similar strings.

# Characteristics of a Good Hash Function

---

- Let's examine why each of these is important:
- Rule 1: If something else besides the input data is used to determine the hash, then the hash value is not as dependent upon the input data, thus allowing for a worse distribution of the hash values.
- Rule 2: If the hash function doesn't use all the input data, then slight variations to the input data would cause an inappropriate number of similar hash values resulting in too many collisions.



# Characteristics of a Good Hash Function

---

- Rule 3: If the hash function does not uniformly distribute the data across the entire set of possible hash values, a large number of collisions will result, cutting down on the efficiency of the hash table.
- Rule 4: In real world applications, many data sets contain very similar data elements. We would like these data elements to still be distributable over a hash table.

- Now that we have a framework in place, let's try using it.
- First, let's store a string into the table: "Steve".
- We run "Steve" through the hash function, and find that  $\text{hash}(\text{"Steve"}, 12)$  yields 3:

Hash Table(strings)	
0	(null)
1	(null)
2	(null)
3	"Steve"
4	(null)
5	(null)
6	(null)
7	(null)
8	(null)
9	(null)
10	(null)
11	(null)

Figure : The hash table after inserting "Steve"

- Let's try another string: "Spark". We run the string through the hash function and find that  $\text{hash}(\text{"Spark"}, 12)$  yields 6. Fine. We insert it into the hash table:

Hash Table(strings)	
0	(null)
1	(null)
2	(null)
3	"Steve"
4	(null)
5	(null)
6	"Spark"
7	(null)
8	(null)
9	(null)
10	(null)
11	(null)

Figure : The hash table after inserting "Spark"

- Let's try another: "Notes". We run "Notes" through the hash function and find that  $\text{hash}(\text{"Notes"}, 12)$  is 3. Ok. We insert it into the hash table:
- What happened? :: Collision

Hash Table(strings)	
0	(null)
1	(null)
2	(null)
3	"Steve" "Notes"
4	(null)
5	(null)
6	"Spark"
7	(null)
8	(null)
9	(null)
10	(null)
11	(null)

Figure : A hash table collision

# Collision

---

- A hash function doesn't guarantee that every input will map to a different output.
- There is always the chance that two inputs will hash to the same output.
- This indicates that both elements should be inserted at the same place in the array, and this is impossible. This phenomenon is known as a collision.
- There are many algorithms for dealing with collisions.

# Collision Resolution

---

- We now return to the problem of collisions. When two items hash to the same slot, we must have a systematic method for placing the second item in the hash table. This process is called collision resolution. As we stated earlier, if the hash function is perfect, collisions will never occur. However, since this is often not possible, collision resolution becomes a very important part of hashing.

# Collision Resolution Techniques

---

- Open Addressing (Linear Probing).
- Chaining

# Linear Probing

---

- One method for resolving collisions looks into the hash table and tries to find another open slot to hold the item that caused the collision. A simple way to do this is to start at the original hash value position and then move in a sequential manner through the slots until we encounter the first slot that is empty.
- Note that we may need to go back to the first slot (circularly) to cover the entire hash table.
- This collision resolution process is referred to as **open addressing** or **linear probing**.



# Linear Probing

- Figure shows an extended set of integer items under the simple remainder method hash function (54,26,93,17,77,31,44,55,20).

0	1	2	3	4	5	6	7	8	9	10
77	44	55	20	26	93	17	None	None	31	54

- When we attempt to place 44 into slot 0, a collision occurs. Under linear probing, we look sequentially, slot by slot, until we find an open position. In this case, we find slot 1.

# Linear Probing

- A disadvantage to linear probing is the tendency for clustering, items become clustered in the table.
- This means that if many collisions occur at the same hash value, a number of surrounding slots will be filled by the linear probing resolution.

0	1	2	3	4	5	6	7	8	9	10
77	44	55	20	26	93	17	None	None	31	54

# Linear Probing

- One way to deal with clustering is to extend the linear probing technique so that instead of looking sequentially for the next open slot, we skip slots, thereby more evenly distributing the items that have caused collisions.
- Figure shows the items when collision resolution is done with a “plus 3” probe. This means that once a collision occurs, we will look at every third slot until we find one that is empty.

0	1	2	3	4	5	6	7	8	9	10
77	55	None	44	26	93	17	20	None	31	54

# Linear Probing

- The general name for this process of looking for another slot after a collision is rehashing. With simple linear probing, the rehash function is  $newhashvalue = rehash(oldhashvalue)$ .

Where  $rehash(pos) = (pos + 1) \% sizeof table$

$rehash(pos) = (pos + 3) \% sizeof table$

in general,  $rehash(pos) = (pos + skip) \% sizeof table$

- A variation of the linear probing idea is called quadratic probing.
- Instead of using a constant “skip” value, we use a rehash function that increments the hash value by 1, 3, 5, 7, 9, and so on.

0	1	2	3	4	5	6	7	8	9	10
77	44	20	55	26	93	17	None	None	31	54

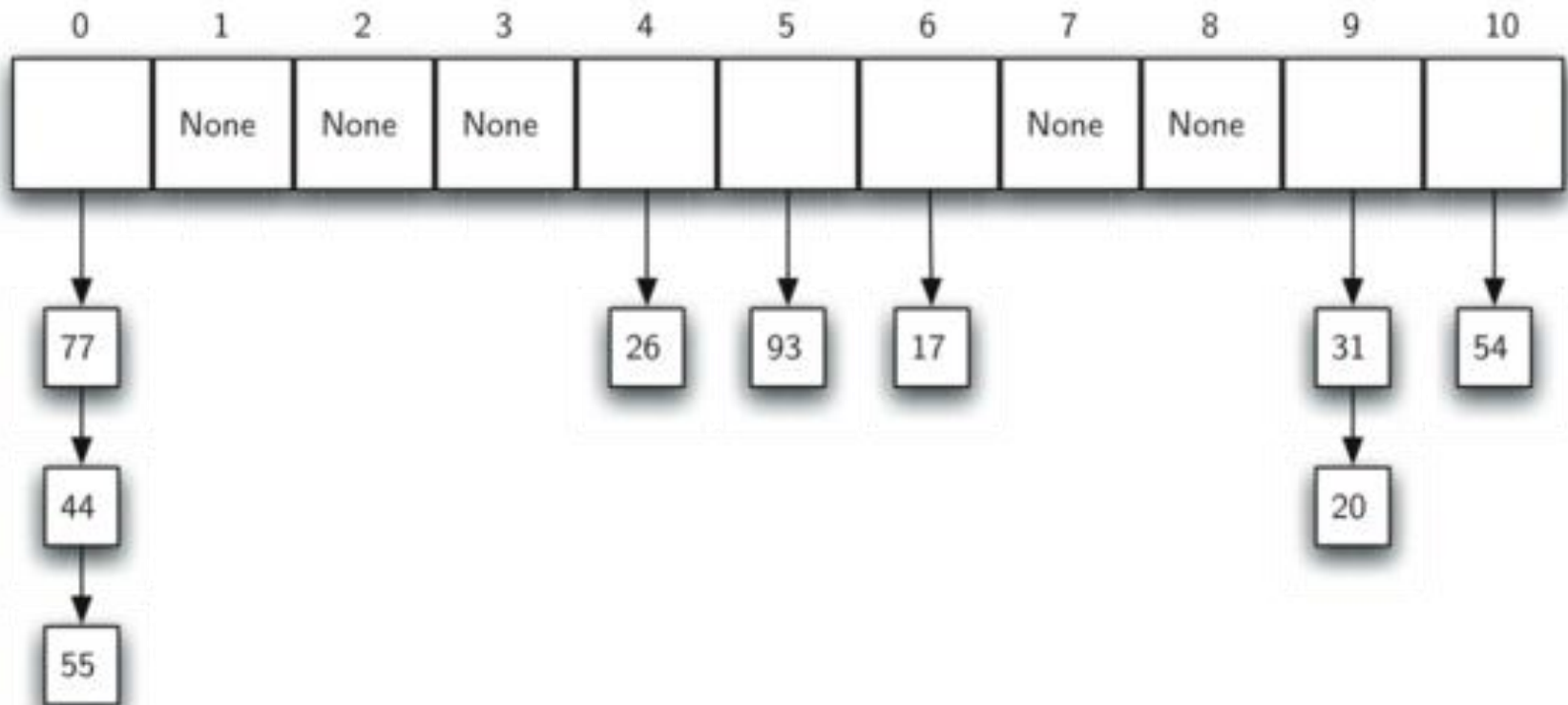
# Chaining

---

- An alternative method for handling the collision problem is to allow each slot to hold a reference to a collection (or chain) of items.
- Chaining allows many items to exist at the same location in the hash table. When collisions happen, the item is still placed in the proper slot of the hash table.
- As more and more items hash to the same location, the difficulty of searching for the item in the collection increases.

# Chaining

- Figure shows the items as they are added to a hash table that uses chaining to resolve collisions.



# Chaining

---

- When we want to search for an item, we use the hash function to generate the slot where it should reside. Since each slot holds a collection, we use a searching technique to decide whether the item is present. The advantage is that on the average there are likely to be many fewer items in each slot, so the search is perhaps more efficient. We will look at the analysis for hashing at the end of this section.

End of Chapter