Lecture 7

DVA104 Data Structures, Algorithms, and Program Design

Abu Naser Masud

Masud.abunaser@mdh.se

Topics

Searching:

- Searching in unsorted array/list
- Searching in sorted array
- Hashtable

Searching

The problem of searching is to determine if an item exists among a *collection of items* and return the address of the searched item if found.

Items are to be collected into different containers such as:

- arrays
- Linked lists
- Binary search tree

Searching an Unsorted Data Structure

- There is only one way to search for an item in an unsorted list or array:

Algorithm:

- Look each element of the list /array from the beginning to the end
  - Return YES if data exists
  - Return NO if we reach the end of the list/array without finding the data

Linear Search

Algorithm:

- Look each element of the list /array from the beginning to the end
  - Return YES if data exists
  - Return NO if we reach the end of the list/array without finding the data
- This algorithm is called *Linear Search* (or sequential search)

Linear Search

Time Complexity:

- Reading (and comparing) one element is O (1)
- To get to the next element is O (1)
- We will visit no more than **n** elements, where **n** is the size of the array/list

Step forward and read the elements

Number of visit

(O(1)+O(1)) * O(n) = O(1) * O(n) = O(1*n) = **O(n)**

Conclusion: Linear search has linear time complexity with respect to the size of the data structure

Searching a Sorted Data Structure

Suppose we have a sorted array or list of elements. The sorting can be in ascending or descending order.

Can we perform a more efficient searching by considering the fact that the array is sorted?

Yes!!!

Searching a Sorted Data Structure

Can we perform a more efficient search when we have a sorted data structure?

Let's say we are looking for 30.

18	24	25	26	30	58	59	63	79	88	98	99

Searching: Sorted Data Structure

Can we perform a more efficient search when we have a sorted data structure?

Let's say we are looking for 30.

18	24	25	26	30	58	59	63	79	88	98	99

- We start looking into the middle element. All the elements to the left of the center is less than 58, and all elements to the right are larger.

Searching: Sorted Data Structure

Can we perform a more efficient search when we have a sorted data structure?

Let's say we are looking for 30.

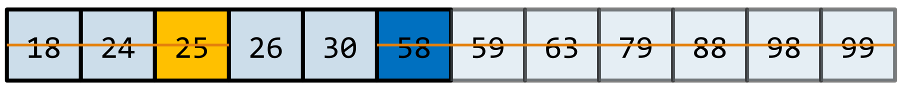
18	24	25	26	30	~~58~~	~~59~~	~~63~~	~~79~~	~~88~~	~~98~~	~~99~~

- We start looking into the center element. All the elements to the left of the center is less than 58, and all elements to the right are larger.

- Next, We reduce the list to be either the first half, or the second half which depends on the center element and the element we are looking for. We thus reduce the search space by half.

Searching: Sorted Data Structure

Let's say we are looking for 30.



- We start looking into the center element. All the elements to the left of the center is less than 58, and all elements to the right are larger.
- We reduce the list to be either the first half, or the second half which depends on the center element and the element we are looking for.
- We continue with the same strategy: look into the centre element and discard looking into the half of the elements of the list

Searching: Sorted Data Structure

Let's say we are looking for 30.

~~18~~	~~24~~	~~25~~	26	30	~~58~~	~~59~~	~~63~~	~~79~~	~~88~~	~~98~~	~~99~~

- We start looking into the center element. All the elements to the left of the center is less than 58, and all elements to the right are larger.
- We reduce the list to be either the first half, or the second half which depends on the center element and the element we are looking for.
- We continue with the same strategy: look into the centre element and discard looking into the half of the elements of the list
- Continue until (i) either the element is found, or (2) the reduced list is empty

Searching: Sorted Data Structure

~~18~~	~~24~~	~~25~~	~~26~~	30	~~58~~	~~59~~	~~63~~	~~79~~	~~88~~	~~98~~	~~99~~

We have found 30. We looked only 4 elements out of 12.

What is the worst-case computational complexity?

$O(\log_2 n)$ where n is the length of the list. Why?

Binary Search

Algorithm (in C like language):
Input: A, n, X
L=0, H=n-1.

```
While (L =< H)      // when L >H, search space contains no elements
        m= L + (H-L)/2
        if (A[m] = X)
                    print(item found at mid)
                    return m
         else if (A[m] > X)
                    H=m-1
         else
                    L = m+1
End while
Print (item not in the list)
Return NOT_Found
```

Walk Through the Algorithm

Algorithm:

Input: A, n,
L=0, H=n-1
While (L =< H)
    m= L + (H-L)/2
    if (A[m] = X)   print(item found at mid)
                    return m
        else if (A[m] > X)
            H=m-1
        else
            L = m+1
End while
Print (item not in the list)
Return NOT_Found

18	24	25	26	30	58	59	63	79	88	98	99
0	1	2	3	4	5	6	7	8	9	10	11

X=63

L	H	m	A[m]= x	A[m]> X	A[m]< X
0	11	5	no	no	yes
6	11	8	no	yes	no
6	7	6	Yes		

Walk Through the Algorithm

Algorithm:

```
Input: A, n,
L=0, H=n-1
While (L =< H)
       m= L + (H-L)/2
       if (A[m] = X)   print(item found at mid)
                          return m
                 else if (A[m] > X)
                        H=m-1
                 else
                       L = m+1
  End while
  Print (item not in the list)
  Return NOT_Found
```

18	24	25	26	30	58	59	63	79	88	98	99
0 1	2	3	4	5	6	7	8	9	10	11	

X=90

L	H	m	A[m]= x	A[m]> X	A[m]< X
0	11	5	no	no	yes
6	11	8	no	no	yes
9	11	10	no	yes	no
9	9	9	no	no	yes
10 L>H	9	-	-	-	-

Worst-Case Complexity of the Algorithm

Algorithm:

```
Input: A, n,
L=0, H=n-1
While (L =< H)
       m= L + (H-L)/2
       if (A[m] = X)   print(item found at mid)
                        return m
                 else if (A[m] > X)
                       H=m-1
                 else
                      L = m+1
  End while
  Print (item not in the list)
  Return NOT_Found
```

Constant Cost: O(1)

Variable Cost: Depends on the number of times the loop executes

Let's say the cost is T(n). Let's break down T(n)

Constant Cost: O(1)

Worst-Case Complexity of the Algorithm

Algorithm:

```
Input: A, n,
L=0, H=n-1
While (L =< H)
       m= L + (H-L)/2
       if (A[m] = X)   print(item found at mid)
                        return m
                 else if (A[m] > X)
                       H=m-1
                 else
                      L = m+1
  End while
  Print (item not in the list)
  Return NOT_Found
```

Let's break down T(n).

- Our search space consists of elements of the search list between the index L and H
- In the first iteration, our search space consists of n elements, and in the second iteration, we have n/2 elements.

- In each iteration, we have constant cost O(1)
- Thus we can write T(n) = T(n/2) + O(1) for iteration 1
- For the sucessive iterations, we can rewrite as follows:
  $T(n) = T(n/2) + O(1)$
  $= T(n/4) + O(1) + O(1)$
  $= T(n/8) + O(1) + O(1) + O(1)$ (3rd iteration)
- So, after third iteration we get
  $T(n) = T(n/2^3) + 3*O(1)$
- After k iteration we get
  $T(n) = T(n/2^k) + k*O(1)$
- In the worst case, item is not in the list, and we search until the search space contains only one element:
  that is, $n/2^k = 1$ and we have $k=\log_2(n)$

So, $T(n) = T(1) + \log_2(n) *O(1) = O(1) + O(\log_2(n)*1) = O(\log_2(n))$

Binary Search

Observation 1:
This is the same as searching a balanced binary search tree! (but more difficult to insert new elements).

Observation 2:
We used a very similar algorithm to build a balanced tree from an array.

Binary search is very effective, but requires our data to be sorted to work.

Search Algorithms

- Linear search: Worst-case complexity O(n)
  - Positive: does not require data to be sorted
  - Negative: O (n) is not effective if you need to search often in large quantities

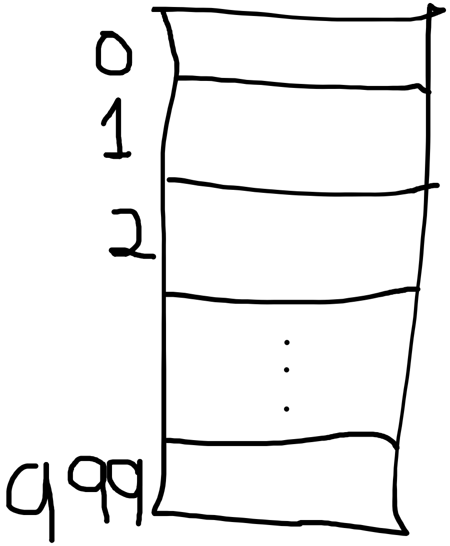
- Binary search: Worst-case complexity $O(\log_2(n))$
  - Positive: computational complexity is attractive!!!
  - Negative: sorting the elements of the list may take time
  - Negative: if we often need to insert or delete data, we need to keep it sorted. (e.g. keeping a binary tree balanced!)

- We can build a data structure that can be searched in O(1) time.

Searching with Constant Complexity

Suppose we have a collection of integer numbers in the range 0 to 999

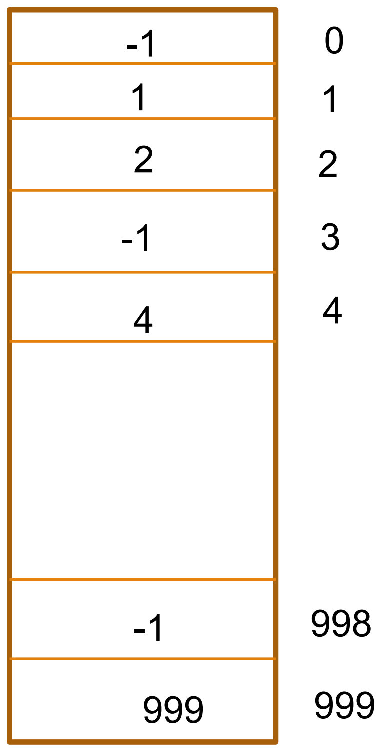
We can store the numbers in the array of size 1000. Suppose A is the array.



- We initialize A[i] =-1 for all i between 0 and 999
- We can store number n in the nth index of the array for any n>=0 and n=< 999 (i.e. A[n] = n).

Searching with Constant Complexity

```
int search(int arr[], int value)
{
    if(value <0 || value>999)
        return 0;
    else if(arr[value] == -1)
        return 0;
    return 1;
}
```



Searching with Constant Complexity

Complexity: No loop: **O(1)**! Even **insert** and **delete** operation have constant complexity.

- However, if our collection contains the set of elements {1,67,888}, we consume 1000 memory locations even though we need only 3 locations.
- We are wasting memory!!!
- More Problems:
  - What happens if we want to save negative items?
  - What happens if we want to save something other than integers?

So we get a constant time complexity by getting a place in the data structure to store and retrieve data in constant time.

We keep this logic and can improve the method to reduce above problems.

Associative Array/Map

Associative array is an abstract data type specifically designed for

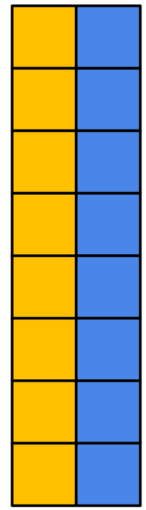
- collecting data of the form (key, value)
- all possible keys must be *unique*
- We associate a key with a data. We use the key to search for a certain value.
- allows effective operation of the following type:
  - Find value (if any) of a given key
  - Insert new (key, value) to the collection
  - Update the value that is bind to a key
  - Delete a particular (key, value) pair

Associative Array/Map

"bucket"

key value

An array of "Buckets/slots"



Exampel:

```
KeyType: personnummer (int)
ValueType: struct person

KeyType: articlenr (int)
ValueType: struct product

KeyType: name (char*)
ValueType: struct person

keyType: ord (char*)
valueType: beskrivning (char*)
```

```
struct Bucket
{
   Keytype key;
   Valuetype value;
};
```

Remember: Key must be unique!!!

Hash Table

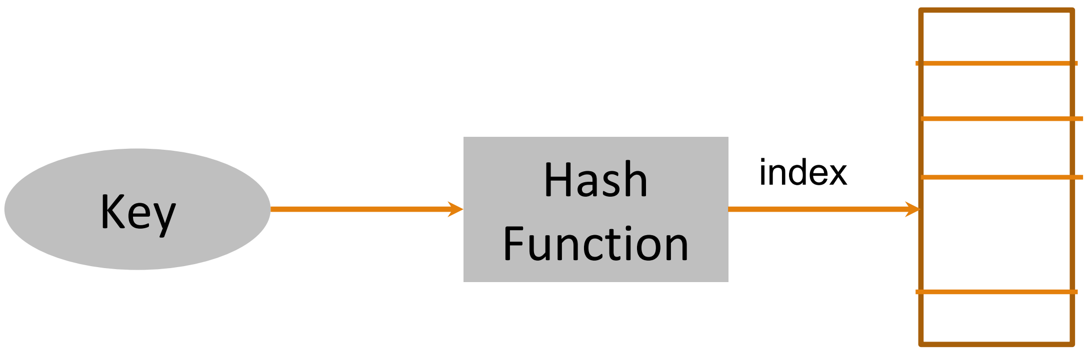
- A Hash table is a common implementation of an associative array
- The idea is based on the fact that each key can only exist in one place in the array

- Given a hash table T, and data item (k,v), we need to provide the following:
  - Insert(T,k,v)
  - Delete(T,k,v)
  - lookup(T,k)
  - Update(T,k,v)

- We want all the above operations to be in O(1) time *without wasting memory* and *support variety of key types*!!!

Hash Table

We use a hash function to find the mapping between key and the array index.



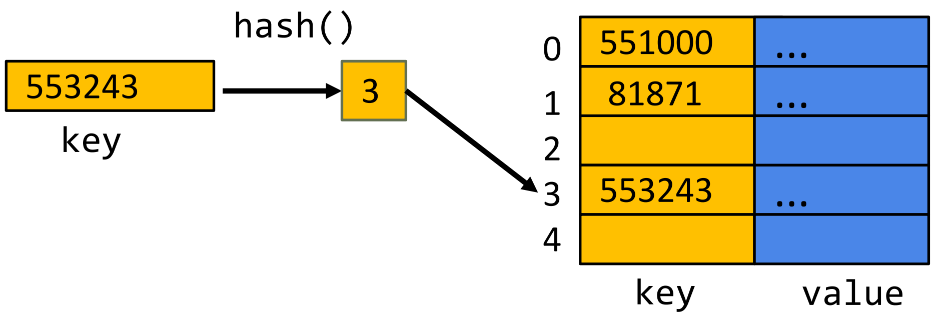

```
/*Returns the index of the hash table associated with the key */
  int hash(Key key);
```

Key can be anything unique that you use to identify a particular record/data

The size of the array should be at least the maximum possible number of record that you want to store

Hash Function

```
hash(553243) == 3    /* Article number 553243 will end up at
                        index 3 in the array */
```

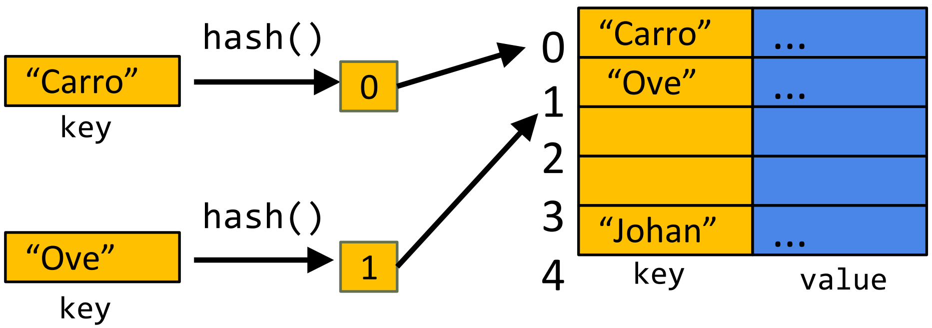



```
typedef int Key;
typedef struct product Value;
```

A given article number can only be found in a specific location in the array.

Hash Function

```
hash("Carro") == 0    /*Carro will end up with index 0*/
```




```
typedef int Key;
typedef struct product Value;
```

A given name can only be found in a place in the array

Hash Function

Example of hash function:

```
int hash(Key articleNumber)
{
   return articleNumber % TABLE_SIZE;
}
```

Will return an index between 0 and the size of the table-1

Hash Table

```
void insert (Hashtable * htable, const key key, const
                    value value);
```

- Insert the {key, value} pair on the index in the hash table as determined by the hash function hash (key).

  If the key is already in the table, the old value is overwritten so that the key is now associated with the value (This is a very particular implementation, not a strict requirement)

Hash Table

```
void insert (Hashtable * htable, const key key, const
value value);
```

- Insert the {key, value} pair on the index in the hash table as determined by the hash function hash (key).

  If the key is already in the table, the old value is overwritten so that the key is now associated with the value.

- Expected complexity O(1)

Hash Table

Example:

```
Hashtable htable = createHashtable(100);
struct product p = createProduct(...);

 /*let's assume that p has articlenr 55210 */

insert(&htable, 55210, p);
```

Hash Table

```
void delete(Hashtable* htable, const Key key);
```

Removes the key-value pair found on the index obtained from hash (key).

```
Value* lookup(const Hashtable* htable, const Key key);
```

Call hash (key) to find out the index. Returns a pointer to the value associated with the key in the table or NULL if no such value exist.

Hash Table

What happens if

hash (5528103) == 3?

But also

hash (4133) == 3

Keys 5528103 and 4133 may refer to various articles, but end up in the same index!

This scenario is usually common. We call it a *collision* and we need a method to have *collision resolution*.

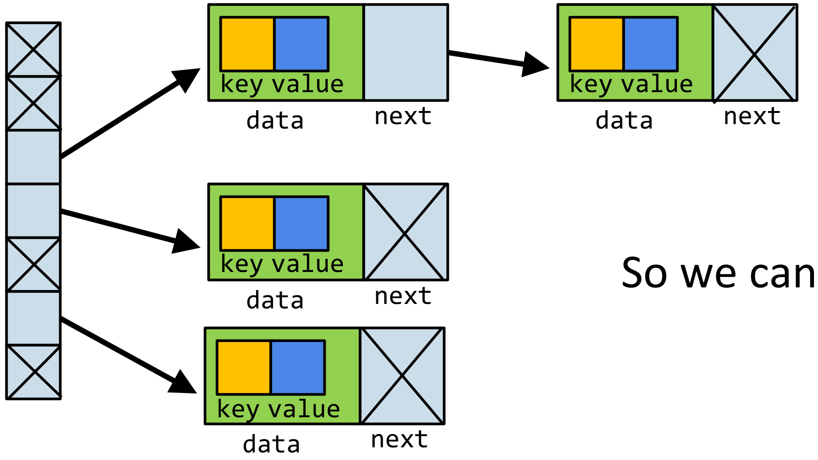
Collision Resolution

- Chaining
- Open Addressing

Collision Resolution: Chaining

Instead of representing the hash table as an array of buckets, we can represent it as an array of linked lists.

The data part of the linked list will be "Bucket".



So we can have more buckets in the same place!

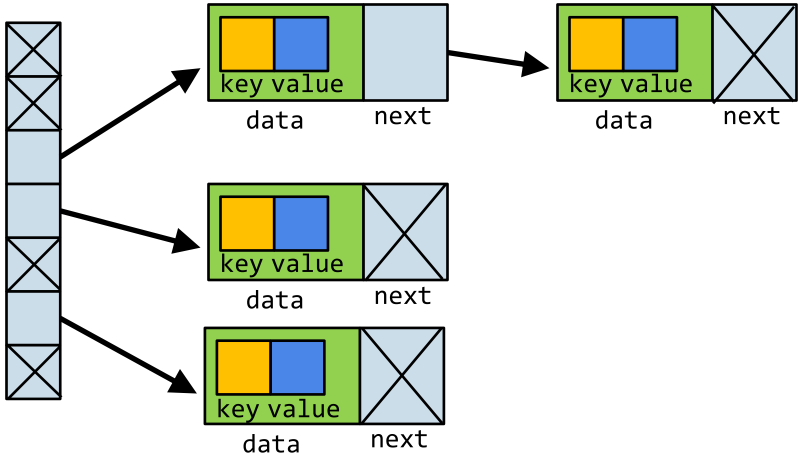
Collision Resolution: Chaining

Insert Algorithm: insert(htable, key, value)

```
1.  index = hash (key).
2.  If (htable[index] == NULL)
          htable[index] = createNode(key,value);
4.  else
5.        temp= htable[index];
6.         while(temp->key!=key && temp->next!=NULL)
                    temp=temp->next
7.  If(temp->key==key ) temp->value = value
    else temp->next=createNode(key,value);
```

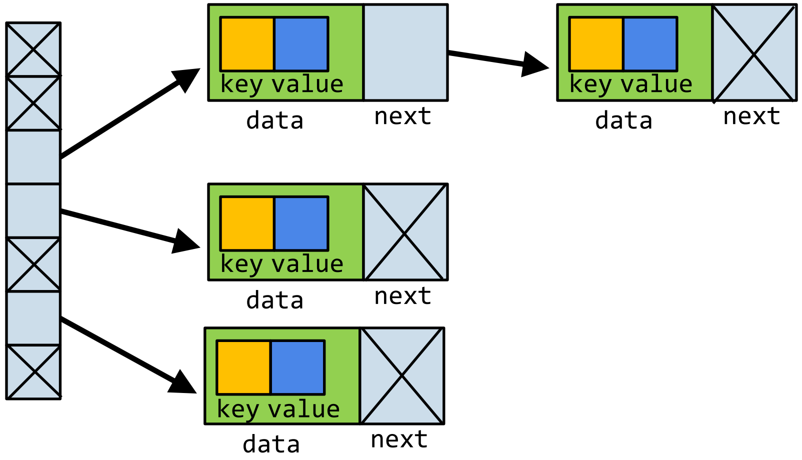
Collision Resolution: Chaining

- How do we delete a record with a given key?



Collision Resolution: Chaining

- How do we lookup a record with a given key?



Collision Resolution: Chaining

Properties of a good hash function:

- **Uniformity:** We need a good hash function that evenly distribute keys among the buckets/slots.
- **Low Cost:** Computation must be quick
- **Effective Conversion:** Noninteger keys should be turned into integer keys and then turn them into index of the hash table.
  Example: if the key is a sequence of characters, we can use the ASCII number of the characters.
- **Determinism:** the same hash value (i.e. index) is always generated for a given key.

Read: Chapter 15, Data structures using C, Reema Thareja

Collision Resolution: Chaining

- Assume that the hash function uniformly distributes the key. That means every buckets or slots are expected to be mapped by equal number of keys.

- Suppose you have n keys and m slots. Then the average no. of keys per slot is called *load factor*. So, load factor $\boldsymbol{\alpha} = n/m$

- Average complexity of unsuccessful search is $O(1+ \boldsymbol{\alpha})$. Why?

- Similarly, average complexity of successful search is $O(1+ \boldsymbol{\alpha}/2) = O(1+ \boldsymbol{\alpha})$. Why?

Collision Resolution: Open Addressing

- All elements occupy the hash table itself. Each table entry contains either an element or NULL/-1.
- To perform insertion, if we get a collision then we try to find another index - *rehash* . We successively examine or *probe* the hash table until we find an empty slot in which to put the element.
- When searching for an element, we systematically examine table slots until either we find the desired element or we have ascertained that the element is not in the table.

Collision Resolution: Open Addressing

The simplest strategy is named *linear probing*

- If the *hash (key)* position is occupied, we use (hash (key) +1)% Tsize
- If even hash (key) +1 is is occupied, we use (hash (key) +2)% Tsize
- If we come to the end of the array, we jump to the beginning again as we do with a circular queue to find a free index.

We are looking forward to a free space - meaning that all key will not be on its hash (key)

Linear Probing

- So, we may use the following hash function to resolve collision

$$h(k,i) = (hash(k) + i)\ \%\ Tsize$$

```
Algorithm: INSERT (T, k)
    i = 0;
    repeat
     j = h (k,i);
     if T[j] == NULL
       T[j]=k;
       return j;
     else i =i +1
   until i == m
   Error "hashT overflow"
```

```
Algorithm: Lookup (T, k)
    i = 0;
    repeat
      j = h (k,i);
      if T[j] == k
          return j;
      i =i +1
until i == m
return NULL
```

Linear Probing

- So, we may use the following hash function to resolve collision
  $h(k,i) = (hash(k) + i)\ \%\ Tsize$

```
Algorithm: INSERT (T, k)
    i = 0;
    repeat
     j = h (k,i);
     if T[j] == NULL
       T[j]=k;
       return j;
     else i =i +1
   until i == m
   Error "hashT overflow"
```

```
Algorithm: Lookup (T, k)
    i = 0;
    repeat
      j = h (k,i);
      if T[j] == k
            return j;
      i =i +1
until i == m
return NULL
```

Open Addressing

Complexity of insert and search/lookup:

Best Case: O (1) (all elements get a unique place)

Worst Case: O(Tsize)
We have to go through all indexes to realize that the table is full

Example: Linear Probing

Hash (k) = k % Tsize

Linear Probing:

h(k,i) = (hash(k) + i) % Tsize

```
insert(T,5522);
insert(T,110);
insert(T,555);
insert(T,102);
insert(T,553);
lookup(T,553);
lookup(T,53);
delete(T,102);
lookup(T,553);
```

0	-1
1	-1
2	-1
3	-1
4	-1
5	-1
6	-1
7	-1
8	-1
9	-1

Initialize -1 (or any suitable value) to indicate the Empty location in the table.

Example: Linear Probing

Hash (k) = k % 10

Linear Probing:

h(k,i) = (hash(k) + i) % 10

```
insert(T,5522); h(5522,0) == 2
insert(T,110);
insert(T,555);
insert(T,102);
insert(T,553);
lookup(T,553);
lookup(T,53);
delete(T,102);
lookup(T,553);
```

0	-1
1	-1
2	5522
3	-1
4	-1
5	-1
6	-1
7	-1
8	-1
9	-1

Example: Linear Probing

Hash (k) = k % 10

Linear Probing:

h(k,i) = (hash(k) + i) % 10

```
insert(T,5522); h(5522,0) == 2
insert(T,110); h(110,0) == 0
insert(T,555);
insert(T,102);
insert(T,553);
lookup(T,553);
lookup(T,53);
delete(T,102);
lookup(T,553);
```

0	110
1	-1
2	5522
3	-1
4	-1
5	-1
6	-1
7	-1
8	-1
9	-1

Example: Linear Probing

Hash (k) = k % 10

Linear Probing:

h(k,i) = (hash(k) + i) % 10

```
insert(T,5522); h(5522,0) == 2
insert(T,110); h(110,0) == 0
insert(T,555); h(555,0) == 5
insert(T,102);
insert(T,553);
lookup(T,553);
lookup(T,53);
delete(T,102);
lookup(T,553);
```

0	110
1	-1
2	5522
3	-1
4	-1
5	555
6	-1
7	-1
8	-1
9	-1

Example: Linear Probing

Hash (k) = k % 10

Linear Probing:

h(k,i) = (hash(k) + i) % 10

```
insert(T,5522); h(5522,0) == 2
insert(T,110); h(110,0) == 0
insert(T,555); h(555,0) == 5
insert(T,102); h(102,0) == 2
insert(T,553);
lookup(T,553);
lookup(T,53);
delete(T,102);
lookup(T,553);
```

0	110
1	-1
2	5522
3	-1
4	-1
5	555
6	-1
7	-1
8	-1
9	-1

Collision!

Example: Linear Probing

Hash (k) = k % 10

Linear Probing:

$$h(k,i) = (hash(k) + i)\ \%\ 10$$

```
insert(T,5522); h(5522,0) == 2
insert(T,110); h(110,0) == 0
insert(T,555); h(555,0) == 5
insert(T,102); h(102,1) == 3
insert(T,553);
lookup(T,553);
lookup(T,53);
delete(T,102);
lookup(T,553);
```

0	110	
1	-1	
2	5522	Kollision!
3	-1	Free!
4	-1	
5	555	
6	-1	
7	-1	
8	-1	
9	-1	

Example: Linear Probing

Hash (k) = k % 10

Linear Probing:

h(k,i) = (hash(k) + i) % 10

0	110	
1	-1	
2	5522	Collision!
3	102	Free!
4	-1	
5	555	
6	-1	
7	-1	
8	-1	
9	-1	

```
insert(T,5522); h(5522,0) == 2
insert(T,110); h(110,0) == 0
insert(T,555); h(555,0) == 5
insert(T,102); h(102,1) == 3
insert(T,553);
lookup(T,553);
lookup(T,53);
delete(102);
lookup(T,553);
```

Example: Linear Probing

Hash (k) = k % 10

Linear Probing:

h(k,i) = (hash(k) + i) % 10

```
insert(T,5522); h(5522,0) == 2
insert(T,110); h(110,0) == 0
insert(T,555); h(555,0) == 5
insert(T,102); h(102,1) == 3
insert(T,553); h(553,0) == 3
lookup(T,553);
lookup(T,53);
delete(T,102);
lookup(T,553);
```

0	110
1	-1
2	5522
3	102
4	-1
5	555
6	-1
7	-1
8	-1
9	-1

Collision!

Example: Linear Probing

Hash (k) = k % 10

Linear Probing:

h(k,i) = (hash(k) + i) % 10

```
insert(T,5522); h(5522,0) == 2
insert(T,110);  h(110,0) == 0
insert(T,555);  h(555,0) == 5
insert(T,102);  h(102,0) == 2
insert(T,553);  h(553,1) == 4
lookup(T,553);
lookup(T,53);
delete(T,102);
lookup(T,553);
```

0	110	
1	-1	
2	5522	
3	102	Collision!
4	-1	Free!
5	555	
6	-1	
7	-1	
8	-1	
9	-1	

Example: Linear Probing

Hash (k) = k % 10

Linear Probing:

h(k,i) = (hash(k) + i) % 10

```
insert(T,5522); h(5522,0) == 2
insert(T,110);  h(110,0) == 0
insert(T,555);  h(555,0) == 5
insert(T,102);  h(102,0) == 2
insert(T,553);  h(553,1) == 4
lookup(T,553);
lookup(T,53);
delete(T,102);
lookup(T,553);
```

0	110
1	-1
2	5522
3	102
4	553
5	555
6	-1
7	-1
8	-1
9	-1

Example: Linear Probing

Hash (k) = k % 10

Linear Probing:

$$h(k,i) = (hash(k) + i) \% 10$$

```
insert(T,5522); h(5522,0) == 2
insert(T,110);  h(110,0) == 0
insert(T,555);  h(555,0) == 5
insert(T,102);  h(102,0) == 2
insert(T,553);  h(553,1) == 4
lookup(T,553);
lookup(T,53);
delete(T,102);
lookup(T,553);
```

0	110
1	-1
2	5522
3	102
4	553
5	555
6	-1
7	-1
8	-1
9	-1

If we want to Insert 812, on which index should 812 be added?

Example: Linear Probing

Hash (k) = k % 10

Linear Probing:

h(k,i) = (hash(k) + i) % 10

```
insert(T,5522); h(5522,0) == 2
insert(T,110);  h(110,0) == 0
insert(T,555);  h(555,0) == 5
insert(T,102);  h(102,0) == 2
insert(T,553);  h(553,1) == 4
lookup(T,553);  h(553,0) == 3
lookup(T,53);
delete(T,102);
lookup(T,553);
```

0	110
1	-1
2	5522
3	102
4	553
5	555
6	-1
7	-1
8	-1
9	-1

553 != 102. But we must keep looking!

Example: Linear Probing

Hash (k) = k % 10

Linear Probing:

$$h(k,i) = (hash(k) + i) \% 10$$

```
insert(T,5522); h(5522,0) == 2
insert(T,110);  h(110,0) == 0
insert(T,555);  h(555,0) == 5
insert(T,102);  h(102,0) == 2
insert(T,553);  h(553,1) == 4
lookup(T,553);  h(553,1) == 4
lookup(T,53);
delete(T,102);
lookup(T,553);
```

Linear probing

0	110
1	-1
2	5522
3	102
4	553
5	555
6	-1
7	-1
8	-1
9	-1

553! Data found.

Example: Linear Probing

Hash (k) = k % 10

Linear Probing:

h(k,i) = (hash(k) + i) % 10

```
insert(T,5522); h(5522,0)  == 2
insert(T,110);  h(110,0)  == 0
insert(T,555);  h(555,0)  == 5
insert(T,102);  h(102,0)  == 2
insert(T,553);  h(553,1)  == 4
lookup(T,553);  h(553,1)  == 4
lookup(T,53);   h(53,0)  == 3
delete(T,102);
lookup(T,553);
```

0	110
1	-1
2	5522
3	102
4	553
5	555
6	-1
7	-1
8	-1
9	-1

102 != 53,
keep looking

Example: Linear Probing

Hash (k) = k % 10
Linear Probing:
h(k,i) = (hash(k) + i) % 10

0	110
1	-1
2	5522
3	102
4	553
5	555
6	-1
7	-1
8	-1
9	-1

553 != 53,
Keep looking

```
insert(T,5522); h(5522,0) == 2
insert(T,110);  h(110,0) == 0
insert(T,555);  h(555,0) == 5
insert(T,102);  h(102,0) == 2
insert(T,553);  h(553,1) == 4
lookup(T,553);  h(553,1) == 4
lookup(T,53);   h(53,1) == 4
delete(T,102);
lookup(T,553);
```

Example: Linear Probing

Hash (k) = k % 10

Linear Probing:

$$h(k,i) = (hash(k) + i)\ \%\ 10$$

```
insert(T,5522); h(5522,0) == 2
insert(T,110);  h(110,0) == 0
insert(T,555);  h(555,0) == 5
insert(T,102);  h(102,0) == 2
insert(T,553);  h(553,1) == 4
lookup(T,553);  h(553,1) == 4
lookup(T,53);   h(53,2) == 5
delete(T,102);
lookup(T,553);
```

0	110
1	-1
2	5522
3	102
4	553
5	555
6	-1
7	-1
8	-1
9	-1

555 != 53,
Keep looking

Example: Linear Probing

Hash (k) = k % 10

Linear Probing:

h(k,i) = (hash(k) + i) % 10

```
insert(T,5522); h(5522,0) == 2
insert(T,110);  h(110,0) == 0
insert(T,555);  h(555,0) == 5
insert(T,102);  h(102,0) == 2
insert(T,553);  h(553,1) == 4
lookup(T,553);  h(553,1) == 4
lookup(T,53);   h(53,3) == 6
delete(T,102);
lookup(T,553);
```

0	110
1	-1
2	5522
3	102
4	553
5	555
6	-1
7	-1
8	-1
9	-1

Free spot! 53 is not in the table. Return NULL.

Example: Linear Probing

Hash (k) = k % 10

Linear Probing:

h(k,i) = (hash(k) + i) % 10

```
insert(T,5522); h(5522,0)  == 2
insert(T,110);  h(110,0)  == 0
insert(T,555);  h(555,0)  == 5
insert(T,102);  h(102,0)  == 2
insert(T,553);  h(553,1)  == 4
lookup(T,553);  h(553,1)  == 4
lookup(T,53);   h(53,3)   == 6
delete(T,102);  h(102,0)  == 2
lookup(T,553);
```

0	110
1	-1
2	5522
3	102
4	553
5	555
6	-1
7	-1
8	-1
9	-1

5522 != 102, keep looking

Example: Linear Probing

Hash (k) = k % 10

Linear Probing:

h(k,i) = (hash(k) + i) % 10

0	110
1	-1
2	5522
3	102
4	553
5	555
6	-1
7	-1
8	-1
9	-1

Found!!!
Delete

```
insert(T,5522); h(5522,0) == 2
insert(T,110);  h(110,0) == 0
insert(T,555);  h(555,0) == 5
insert(T,102);  h(102,0) == 2
insert(T,553);  h(553,1) == 4
lookup(T,553);  h(553,1) == 4
lookup(T,53);   h(53,3)  == 6
delete(T,102);  h(102,1) == 3
lookup(T,553);
```

Example: Linear Probing

Hash (k) = k % 10

Linear Probing:

$$h(k,i) = (hash(k) + i\ )\ \%\ 10$$

0	110
1	-1
2	5522
3	#Deleted
4	553
5	555
6	-1
7	-1
8	-1
9	-1

Mark as deleted

Why???

```
insert(T,5522); h(5522,0) == 2
insert(T,110);  h(110,0)  == 0
insert(T,555);  h(555,0)  == 5
insert(T,102);  h(102,0)  == 2
insert(T,553);  h(553,1)  == 4
lookup(T,553);  h(553,1)  == 4
lookup(T,53);   h(53,3)   == 6
delete(T,102);  h(102,1)  == 3
lookup(T,553);
```

Example: Linear Probing

Hash (k) = k % 10

Linear Probing:

h(k,i) = (hash(k) + i) % 10

0	110
1	-1
2	5522
3	-1
4	553
5	555
6	-1
7	-1
8	-1
9	-1

Suppose we mark
The location as
empty

```
insert(T,5522); h(5522,0) == 2
insert(T,110);  h(110,0) == 0
insert(T,555);  h(555,0) == 5
insert(T,102);  h(102,0) == 2
insert(T,553);  h(553,1) == 4
lookup(T,553);  h(553,1) == 4
lookup(T,53);   h(53,3)  == 6
delete(T,102);  h(102,1) == 3
lookup(T,553);
```

If we were to look for 553 now, we would look at index 3 and realize that it is empty.

Thus, we reach a wrong conclusion that 553 does not exist.

Example: Linear Probing

Hash (k) = k % 10

Linear Probing:

h(k,i) = (hash(k) + i) % 10

0	110
1	-1
2	5522
3	#Deleted
4	553
5	555
6	-1
7	-1
8	-1
9	-1

```
insert(T,5522); h(5522,0)  == 2
insert(T,110);  h(110,0)  == 0
insert(T,555);  h(555,0)  == 5
insert(T,102);  h(102,0)  == 2
insert(T,553);  h(553,1)  == 4
lookup(T,553);  h(553,1)  == 4
lookup(T,53);   h(53,3)   == 6
delete(T,102);  h(102,1)  == 3
lookup(T,553);
```

We can then modify the insert procedure to consider all ”Deleted” Cell as empty. The lookup procedure will work as usual.

Linear Probing Algorithms

```
Algorithm: INSERT (T, k)
    i = 0;
    repeat
     j = h (k,i);
     if (T[j] == NULL or
           T[j] == #Deleted)
       T[j]=k;
       return j;
     else i =i +1
   until i == m
   Error "hashT overflow"
```

```
Algorithm: DELETE (T, k)
    i = 0;
    repeat
     j = h (k,i);
     if T[j] == k
       T[j]=#Deleted;
       return j;
     else i =i +1
   until i == m
   Error "Not Found"
```

Example: Linear Probing

```
int hash(int key)
{
   return key % 10;
}
```

Suppose now that the hash table looks like this, and we want to perform Insert (T, 538)

Will it insert the (key, data) pair? If so, what index should 538 be added?

0	
1	
2	5522
3	553
4	
5	555
6	
7	
8	118
9	399

How to avoid collisions?

On the hash table size

- The table size should be greater than the number of items you want to store
- The bigger the table, the less risk of collisions
- However, the table should not waste more memory than necessary
- It has been found that about 1.3 * the number of elements usually work
- The size should be a prime? (Because the hash feature often uses modulo.

Good hash function

- Modulo on a prime avoids forming clusters of "similar" keys
- Sometimes we may need further calculations on the key