Today’s topics

- Hash table
- Sorting

References

- Chapter-11 from the CLRS book
- Chapter-5 from Mark Allen Weiss
- Chapter-2.5 from Goodrich and Tamassia

Open addressing

Open addressing

- In open addressing, all elements are stored in hash table slots only
    - No additional linked list is created

- Linear probing and quadratic probing are some of the techniques that are used to handle hash collisions

Linear probing

Linear probing

- In the linear probing scheme, if slot i returned by the hash function is already occupied
  - the element is inserted into the first available slot between i+1 to m-1, where m is the number of slots in the hash table
  - If none of the slots is available between i+1 to m-1, the element is inserted at the first available slot between 0 to i-1

Insert

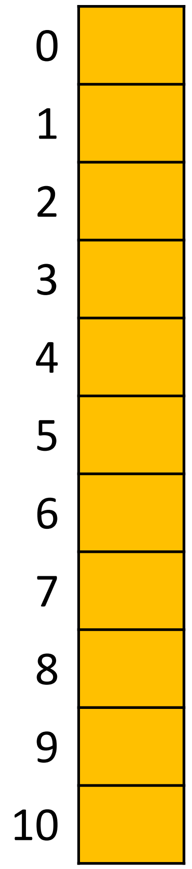
Linear probing

0	
1	
2	13
3	
4	26
5	5
6	37
7	16
8	15
9	
10	21

Insert keys 13, 21, 26, 5, 37, 16, 15

```
int HashFun(int x) {
    return x % 11;
}
```

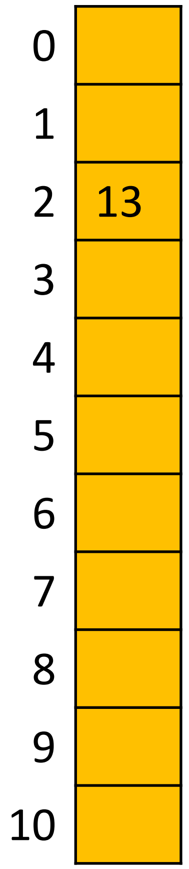
Linear probing



```
Insert keys 13, 21, 26, 5, 37, 16, 15
```

```
int HashFun(int x) {
    return x % 11;
}
```

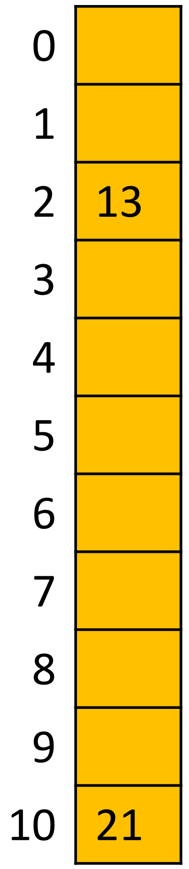
Linear probing



Insert keys 13, 21, 26, 5, 37, 16, 15

```
int HashFun(int x) {
    return x % 11;
}
```

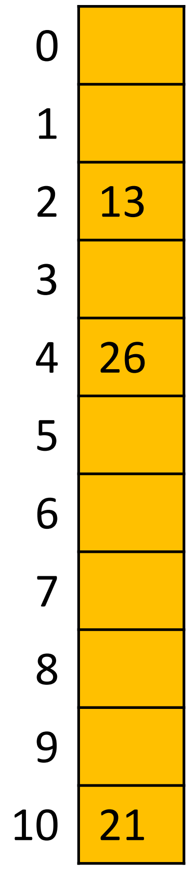
Linear probing



Insert keys 13, 21, 26, 5, 37, 16, 15

```
int HashFun(int x) {
    return x % 11;
}
```

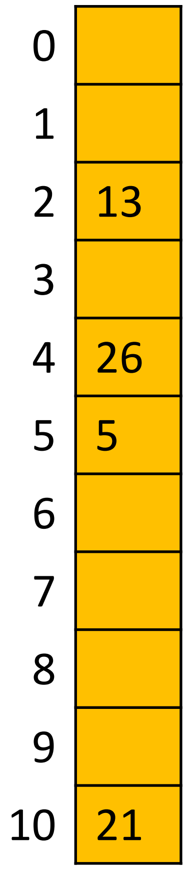
Linear probing



Insert keys 13, 21, 26, 5, 37, 16, 15

```
int HashFun(int x) {
    return x % 11;
}
```

Linear probing



```
Insert keys 13, 21, 26, 5, 37, 16, 15
```

```
int HashFun(int x) {
    return x % 11;
}
```

Linear probing

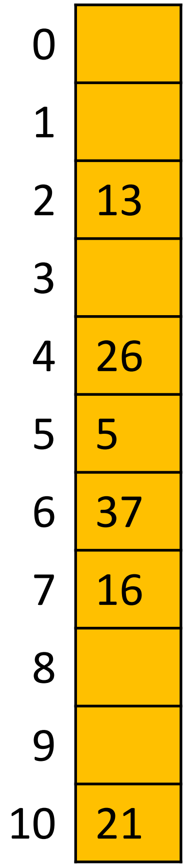
0	
1	
2	13
3	
4	26
5	5
6	37
7	
8	
9	
10	21

`Insert keys 13, 21, 26, 5, 37, 16, 15`

Inserting 37.
37 % 11 = 4
Index 4 is already occupied.
check index 5.
Index 5 is already occupied.
check index 6.
Insert at index 6.

```
int HashFun(int x) {
    return x % 11;
}
```

Linear probing

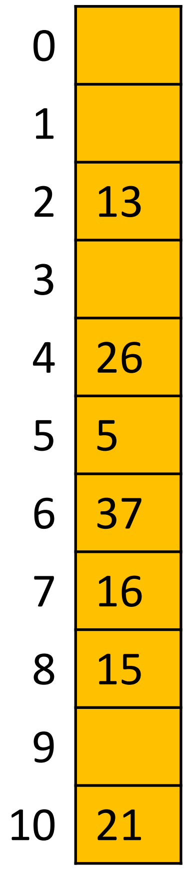


`Insert keys 13, 21, 26, 5, 37, 16, 15`

Inserting 16.
16 % 11 = 5
5 is already occupied
check index 6
6 is already occupied
check index 7
insert at index 7

```
int HashFun(int x) {
    return x % 11;
}
```

Linear probing



`Insert keys 13, 21, 26, 5, 37, 16, 15`

Inserting 15.
15 % 11 = 4
4 is already occupied
check index 5.
5 is already occupied
check index 6.
6 is already occupied
check index 7.
7 is already occupied.
check index 8.
Insert at index 8.

```
int HashFun(int x) {
    return x % 11;
}
```

Insert

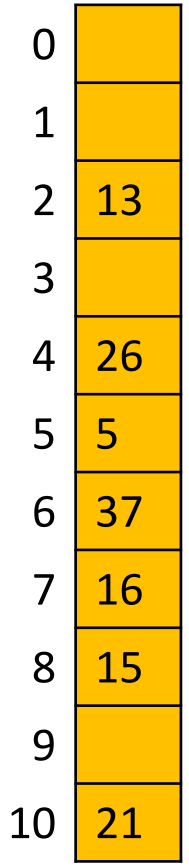
```
HASH_INSERT(h, e, m)
// h is the hash table
// m is the total number of slots in the hash table
// e is the element that has two fields: key and value
// Output: Insert e in the hash table h and return the
index in the hash table

idx = hash_function(e.key, m)
i = 0
while i < m
   if h[idx] == NIL
      h[idx] = e
      return idx
   idx = (idx + 1) % m
   i = i + 1

error "hash table overflow"
```

Search

Linear probing



Search 12

12 % 11 = 1

1 is vacant

```
int HashFun(int x) {
    return x % 11;
}
```

Linear probing

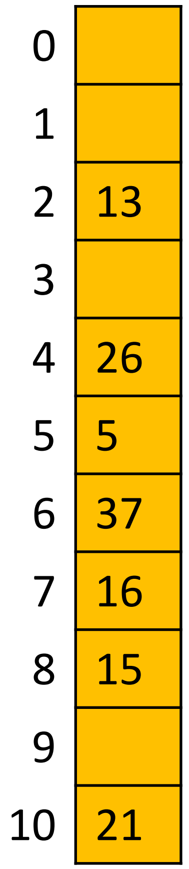
0	
1	
2	13
3	
4	26
5	5
6	37
7	16
8	15
9	
10	21

Search 12

12 % 11 = 1
hash[1] == NIL
12 is not present in the hash table

```
int HashFun(int x) {
    return x % 11;
}
```

Linear probing



Search 15

```
int HashFun(int x) {
    return x % 11;
}
```

Linear probing

0	
1	
2	13
3	
4	26
5	5
6	37
7	16
8	15
9	
10	21

Search 15

15 % 11 = 4
hash[4] != 15 and hash[4] != NIL
check 5
hash[5] != 15 and hash[4] != NIL
check 6
hash[6] != 15 and hash[5] != NIL
check 7
hash[7] != 15 and hash[7] != NIL
check 8
hash[8] == 15 and hash[8] != NIL
return the corresponding entry

```
int HashFun(int x) {
    return x % 11;
}
```

Search

```
HASH_SEARCH(h, k, m)
// h is the hash table
// m is the total number of slots in the hash table
// k is the key being searched
// Output: Return NIL if the element is not stored in the
table; otherwise, return the stored element

idx = hash_function(k, m)
i = 0
while i < m and h[idx] != NIL
    if h[idx].key == k
        return h[idx]
    idx = (idx + 1) % m
    i = i + 1
return NIL
```

Delete

Linear probing

0		
1		
2	13	
3		
4	26	
5	5	
6	37	NIL
7	16	
8	15	
9		
10	21	

Delete 37

```
int HashFun(int x) {
    return x % 11;
}
```

Linear probing

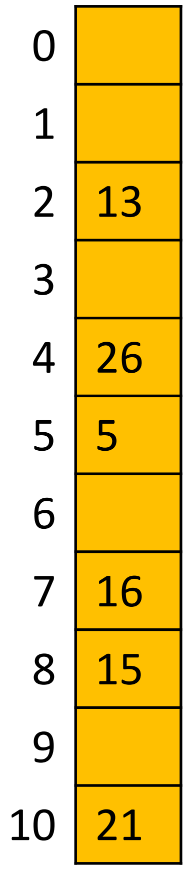
0	
1	
2	13
3	
4	26
5	5
6	37
7	16
8	15
9	
10	21

Delete 37

37 % 11 = 4
hash[4] != 37 and hash[4] != NIL
check 5
hash[5] != 37 and hash[5] != NIL
check 6
hash[6] == 37 and hash[6] != NIL

```
int HashFun(int x) {
    return x % 11;
}
```

Linear probing

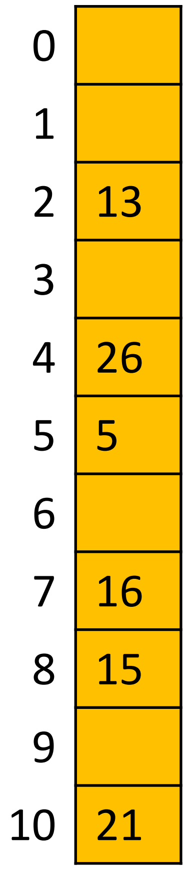


Delete 37

37 % 11 = 4
hash[4] != 37 and hash[4] != NIL
check 5
hash[5] != 37 and hash[5] != NIL
check 6
hash[6] == 37 and hash[6] != NIL
hash[6] = NIL

```
int HashFun(int x) {
    return x % 11;
}
```

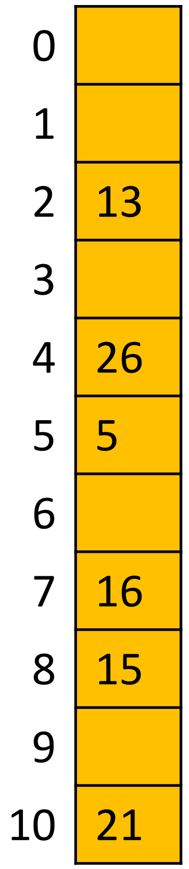
Linear probing



Search 15

```
int HashFun(int x) {
    return x % 11;
}
```

Linear probing



`Search 15`

15 % 11 = 4
hash[4] != 15 and hash[4] != NIL
check 5
hash[5] != 15 and hash[5] != NIL
check 6
hash[6] == NIL
stop.
15 is not present in the hash table, which is not true.

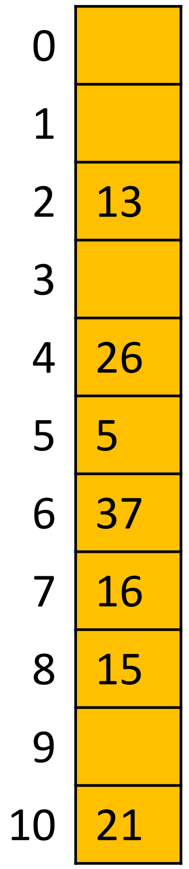
```
int HashFun(int x) {
    return x % 11;
}
```

Linear probing

- During delete, simply setting the corresponding hash slot to NIL may create issues because the subsequent slots may contain entries corresponding to the current and previous slots

- A special marker is used for deleted slots to skip the deleted slot and continue probing during search and delete operations

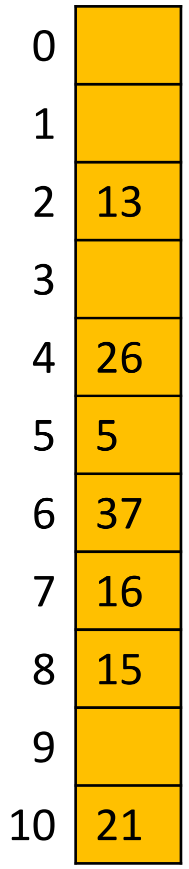
Linear probing



Delete 37

```
int HashFun(int x) {
    return x % 11;
}
```

Linear probing

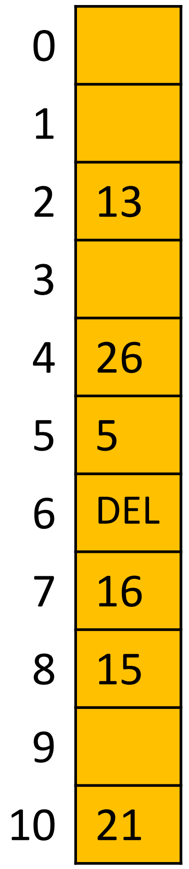


Delete 37

37 % 11 = 4
hash[4] != 37 and hash[4] != NIL
check 5
hash[5] != 37 and hash[5] != NIL
check 6
hash[6] == 37 and hash[6] != NIL

```
int HashFun(int x) {
    return x % 11;
}
```

Linear probing



Delete 37

37 % 11 = 4
hash[4] != 37 and hash[4] != NIL
check 5
hash[5] != 37 and hash[5] != NIL
check 6
hash[6] == 37 and hash[6] != NIL
hash[6] = DEL // special marker

```
int HashFun(int x) {
    return x % 11;
}
```

Linear probing

0	
1	
2	13
3	
4	26
5	5
6	DEL
7	16
8	15
9	
10	21

Search 15

```
int HashFun(int x) {
    return x % 11;
}
```

Linear probing

0	
1	
2	13
3	
4	26
5	5
6	DEL
7	16
8	15
9	
10	21

Search 15

15 % 11 = 4
hash[4] != 15 and hash[4] != NIL
check 5
hash[5] != 15 and hash[5] != NIL
check 6
hash[6] != 15 and hash[6] != NIL
check 7
hash[7] != 15 and hash[7] != NIL
check 8
hash[8] == 15
The key is present.
Return corresponding entry.

```
int HashFun(int x) {
    return x % 11;
}
```

Search

```
HASH_SEARCH(h, k, m)
// h is the hash table
// m is the total number of slots in the hash table
// k is the key being searched
// Output: Return NIL if the element is not stored in the
table; otherwise, return the stored element

idx = hash_function(k, m)
i = 0
while i < m and h[idx] != NIL
    if h[idx] != DEL and h[idx].key == k
        return h[idx]
    idx = (idx + 1) % m
    i = i + 1
return NIL
```

Delete

```
HASH_DELETE(h, k, m)
// h is the hash table
// m is the total number of slots in the hash table
// k is the key that needs to be deleted
// Output: Delete an entry that contains k

idx = hash_function(k, m)
i = 0
while i < m and h[idx] != NIL
    if h[idx] != DEL and h[idx].key == k
        h[idx] = DEL
        return
    idx = (idx + 1) % m
    i = i + 1
return NIL
```

Insert

```
HASH_INSERT(h, e, m)
// h is the hash table
// m is the total number of slots in the hash table
// e is the element that has two fields: key and value
// Output: Insert e in the hash table h and return the
index in the hash table

idx = hash_function(e.key, m)
i = 0
while i < m
   if h[idx] == NIL or h[idx] == DEL
      h[idx] = e
      return idx
   idx = (idx + 1) % m
   i = i + 1
error “hash table overflow”
```

Linear probing

- Does the performance of a search operation in the linear probing mechanism is the same as chaining?

Linear probing

- Does the performance of a search operation in the linear probing mechanism is the same as chaining?
  - No, because in the case of hash collisions, all the elements corresponding to a hash slot may not be consecutive. There could be entries corresponding to the other hash table slots in between.

Linear probing

- The linear probing may create clustering (also called primary clustering) because an empty slot preceded by n occupied slots gets filled next with the probability of $(n+1)/m$. Therefore, the average search time increases.

Quadratic probing

Quadratic probing

- In quadratic probing, if the i^{th} slot returned by the hash function is already occupied, slots at indices $(i + j^2) \bmod m$ are tried for j = 1, 2,3, …; where m is the total number of slots in the hash table

- Quadratic probing doesn’t quarantine that all slots will be visited; however, if m is a prime number, then it definitely finds a slot in the hash table if more than half slots are available

Quadratic probing

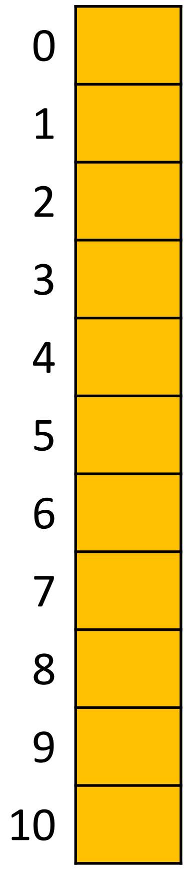
0	
1	
2	13
3	
4	26
5	5
6	16
7	
8	37
9	15
10	21

Insert keys 13, 21, 26, 5, 37, 16, 15

Probing strategy:
i = HashFun(key)
Try (i + j^2) mod 11 for j = 1, 2,3, …

```
int HashFun(int x) {
    return x % 11;
}
```

Quadratic probing

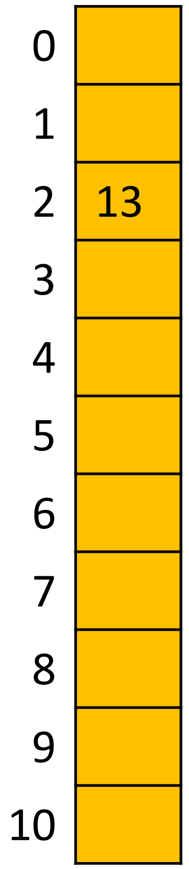


`Insert keys 13, 21, 26, 5, 37, 16, 15`

Probing strategy:
i = HashFun(key)
Try $(i + j^2) \bmod 11$ for j = 1, 2,3, …

```
int HashFun(int x) {
    return x % 11;
}
```

Quadratic probing



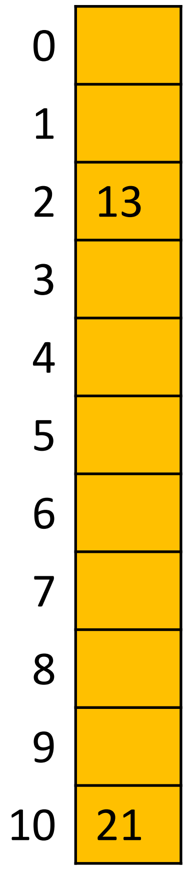
```
Insert keys 13, 21, 26, 5, 37, 16, 15
```

Inserting 13

Probing strategy:
i = HashFun(key)
Try $(i + j^2)$ mod 11 for j = 1, 2,3, …

```
int HashFun(int x) {
    return x % 11;
}
```

Quadratic probing



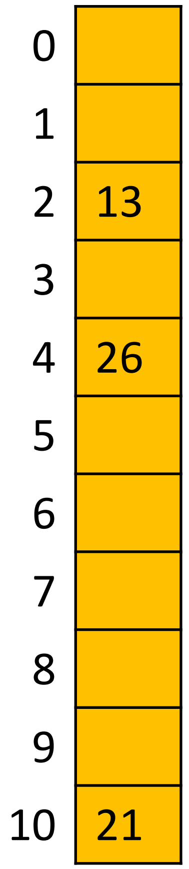
`Insert keys 13, 21, 26, 5, 37, 16, 15`

Inserting 21

Probing strategy:
i = HashFun(key)
Try $(i + j^2)$ mod 11 for j = 1, 2,3, …

```
int HashFun(int x) {
    return x % 11;
}
```

Quadratic probing



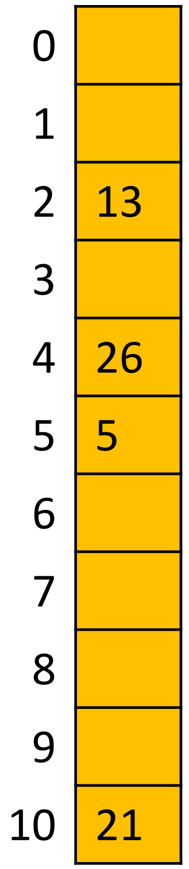
`Insert keys 13, 21, 26, 5, 37, 16, 15`

Inserting 26

Probing strategy:
i = HashFun(key)
Try $(i + j^2)$ mod 11 for j = 1, 2,3, …

```
int HashFun(int x) {
    return x % 11;
}
```

Quadratic probing



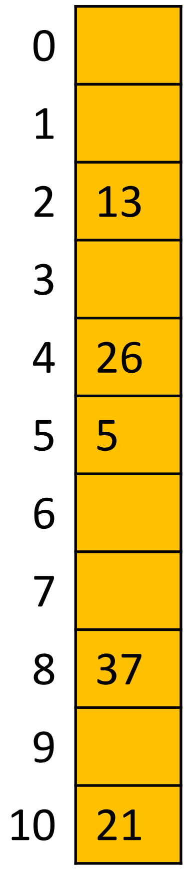
`Insert keys 13, 21, 26, 5, 37, 16, 15`

Inserting 5

Probing strategy:
i = HashFun(key)
Try $(i + j^2)$ mod 11 for j = 1, 2,3, …

```
int HashFun(int x) {
    return x % 11;
}
```

Quadratic probing



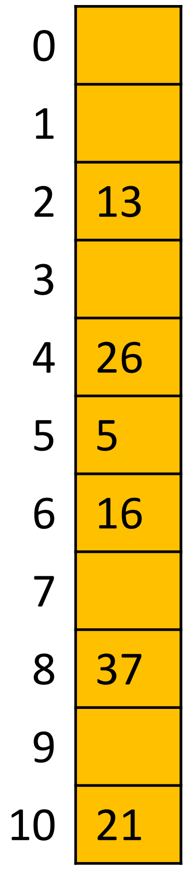
`Insert keys 13, 21, 26, 5, 37, 16, 15`

Inserting 37
37 % 11 = 4 is occupied
(4 + 1) % 11 = 5 is occupied
(4 + 4) % 11 = 8 is available

Probing strategy:
i = HashFun(key)
Try $(i + j^2)$ mod 11 for j = 1, 2,3, …

```
int HashFun(int x) {
    return x % 11;
}
```

Quadratic probing



`Insert keys 13, 21, 26, 5, 37, 16, 15`

Inserting 16
16 % 11 = 5 is occupied
(5 + 1) % 11 = 6 is available

Probing strategy:
i = HashFun(key)
Try (i + j^2) mod 11 for j = 1, 2,3, …

```
int HashFun(int x) {
    return x % 11;
}
```

Quadratic probing

0	
1	
2	13
3	
4	26
5	5
6	16
7	
8	37
9	15
10	21

Insert keys 13, 21, 26, 5, 37, 16, 15

Inserting 15
15 % 11 = 4 is occupied
(4 + 1) % 11 = 5 is occupied
(4 + 4) % 11 = 8 is occupied
(4 + 9) % 11 = 2 is occupied
(4 + 16) % 11 = 9 is available

Probing strategy:
i = HashFun(key)
Try (i + j^2) mod 11 for j = 1, 2,3, …

```
int HashFun(int x) {
    return x % 11;
}
```

$(a+b) \bmod m = (a \bmod m + b \bmod m) \bmod m$

$(a_1 m + c_1 + b_1 m + c_2)$

Quadratic probing

- If quadratic probing is used and m is prime, then first $\left\lfloor \frac{m}{2} \right\rfloor$ alternative locations given by the quadratic probing algorithm are distinct

$1 \leq j_1 < j_2 \leq \lfloor \frac{m}{2} \rfloor$

$(i + j_1^2) \bmod m = (i + j_2^2) \bmod m$

$(j_2^2 - j_1^2) \bmod m = 0$

$((j_2 + j_1)(j_2 - j_1)) \bmod m = 0$

$0 < j_2 + j_1 < m$

$0 < j_2 - j_1 < m$

Quadratic probing

- If quadratic probing is used and m is prime, then first $\left\lfloor \frac{m}{2} \right\rfloor$ alternative locations given by the quadratic probing algorithm are distinct

Let's say this is not true.
It means there exists, j_1, j_2 such that $1 \le j_1 < j_2 \le \lfloor m/2 \rfloor$
and
$(i + j_1^2) \ mod \ m = (i + j_2^2) \ mod \ m$
$j_1^2 \ mod \ m = j_2^2 \ mod \ m$
$(j_2^2 - j_1^2) \ mod \ m = 0$
$((j_2 + j_1)(j_2 - j_1)) \ mod \ m = 0$

The above condition only holds if $(j_2 + j_1)\ (j_2 - j_1)$ is either zero or a multiple of m. However, none of these can be true because $1 \le j_2 - j_1 < j_2 + j_1 < m$ and m is prime. Therefore, for first $\left\lfloor \frac{m}{2} \right\rfloor$ probes always give distinct values.

Quadratic probing

- Although quadratic probing eliminates primary clustering, it creates its own kind of clustering called secondary clustering. This is because elements that hash to the same positions will probe at similar slots.

Double hashing

- Double hashing scheme uses two hash functions, hash1 and hash2
    - hash2 never returns zero
- i = hash1(key) is already occupied, then an offset o = hash2(key) is computed
    - probing is done at index (i+o)%m, (i+(2*o))%m, (i+(3*o))%m, and so on until an empty slot is encountered, where m is the size of the hash table
    - if o < m and m is not divisible by o, then we can always find an empty slot during insertion if a slot is available

Double hashing

0	15
1	
2	13
3	16
4	26
5	5
6	
7	
8	
9	37
10	21

$i = 4$

$o = 6$

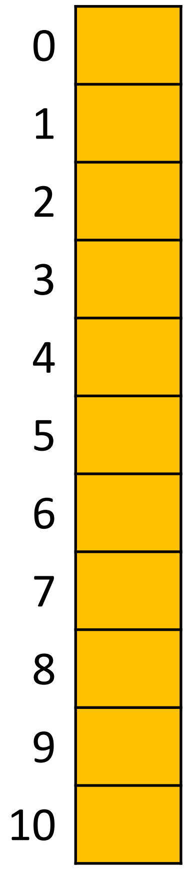
```
Insert keys 13, 21, 26, 5, 37, 16, 15
```

Probing strategy:
i = hash1(key)
o = hash2(key)
Try (i + (j*o)) mod 11 for j = 1, 2,3, …

```
int hash1(int x) {
    return x % 11;
}

int hash2(int x) {
    return 7 - (x % 7);
}
```

Double hashing



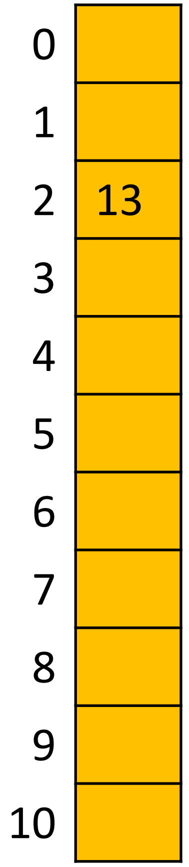
```
Insert keys 13, 21, 26, 5, 37, 16, 15
```

Probing strategy:
i = hash1(key)
o = hash2(key)
Try (i + (j*o)) mod 11 for j = 1, 2,3, …

```
int hash1(int x) {
    return x % 11;
}

int hash2(int x) {
    return 7 - (x % 7);
}
```

Double hashing



`Insert keys 13, 21, 26, 5, 37, 16, 15`

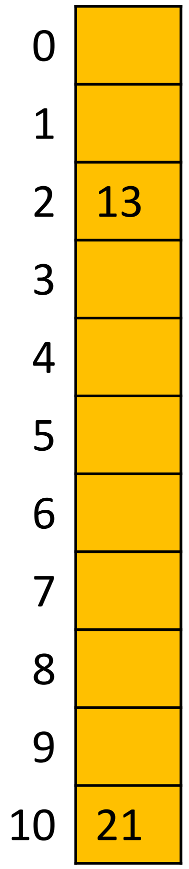
Inserting 13 at 13 % 11 = 2

Probing strategy:
i = hash1(key)
o = hash2(key)
Try (i + (j*o)) mod 11 for j = 1, 2,3, …

```
int hash1(int x) {
    return x % 11;
}

int hash2(int x) {
    return 7 – (x % 7);
}
```

Double hashing



Insert keys 13, 21, 26, 5, 37, 16, 15

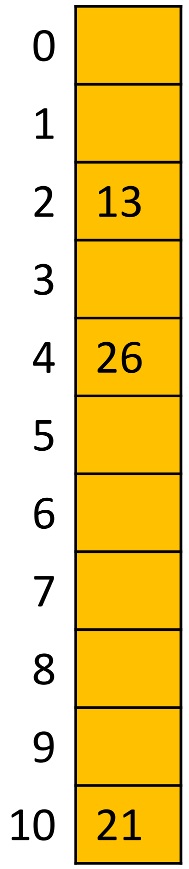
Inserting 21 at 21 % 11 = 10

Probing strategy:
i = hash1(key)
o = hash2(key)
Try (i + (j*o)) mod 11 for j = 1, 2,3, …

```
int hash1(int x) {
    return x % 11;
}

int hash2(int x) {
    return 7 – (x % 7);
}
```

Double hashing



Insert keys 13, 21, 26, 5, 37, 16, 15

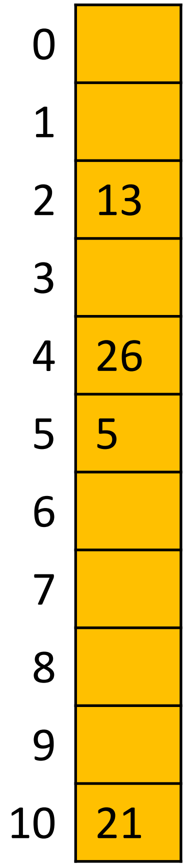
Inserting 26 at 26 % 11 = 4

Probing strategy:
i = hash1(key)
o = hash2(key)
Try (i + (j*o)) mod 11 for j = 1, 2,3, …

```
int hash1(int x) {
    return x % 11;
}

int hash2(int x) {
    return 7 - (x % 7);
}
```

Double hashing



Insert keys 13, 21, 26, 5, 37, 16, 15

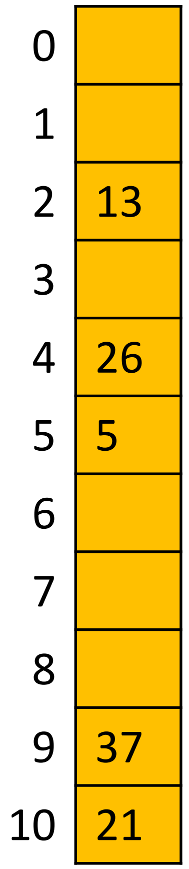
Inserting 5 at 5 % 11 = 5

Probing strategy:
i = hash1(key)
o = hash2(key)
Try (i + (j*o)) mod 11 for j = 1, 2,3, …

```
int hash1(int x) {
    return x % 11;
}

int hash2(int x) {
    return 7 – (x % 7);
}
```

Double hashing



```
Insert keys 13, 21, 26, 5, 37, 16, 15
```

Inserting 37 at 37 % 11 = 4, which is occupied
off = 7 – (37 % 7) = 5
Inserting 37 at (4 + 5) % 11 = 9

Probing strategy:
i = hash1(key)
o = hash2(key)
Try (i + (j*o)) mod 11 for j = 1, 2,3, …

```
int hash1(int x) {
    return x % 11;
}

int hash2(int x) {
    return 7 - (x % 7);
}
```

Double hashing

Probing strategy:
i = hash1(key)
o = hash2(key)
Try (i + (j*o)) mod 11 for j = 1, 2,3, …

0	
1	
2	13
3	16
4	26
5	5
6	
7	
8	
9	37
10	21

Insert keys 13, 21, 26, 5, 37, 16, 15

Inserting 16 at 16 % 11 = 5, which is occupied
off = 7 – (16 % 7) = 5
Inserting 16 at (5 + 5) % 11 = 10, which is occupied
Inserting 16 at (5 + (2*5)) % 11 = 4, which is occupied
Inserting 16 at (5 + (3*5)) % 11 = 9, which is occupied
Inserting 16 at (5 + (4*5)) % 11 = 3

```
int hash1(int x) {
    return x % 11;
}

int hash2(int x) {
    return 7 - (x % 7);
}
```

Double hashing

Probing strategy:
i = hash1(key)
o = hash2(key)
Try (i + (j*o)) mod 11 for j = 1, 2,3, …

0	15
1	
2	13
3	16
4	26
5	5
6	
7	
8	
9	37
10	21

Insert keys 13, 21, 26, 5, 37, 16, 15

Inserting 15 at 15 % 11 = 4, which is occupied
off = 7 – (15 % 7) = 6
Inserting 15 at (4 + 6) % 11 = 10, which is occupied
Inserting 15 at (4 + (2*6)) % 11 = 5, which is occupied
Inserting 15 at (4 + (3*6)) % 11 = 0

```
int hash1(int x) {
    return x % 11;
}

int hash2(int x) {
    return 7 - (x % 7);
}
```

Rehashing

Rehashing

- In the open addressing approach, what can we do when the hash table is full?

Rehashing

- In the open addressing approach, what can we do when the hash table is full?
  - We use an approach similar to the dynamic arrays
  - Create a new hash table with a size equal to a prime number close to twice the size of the previous hash table
  - The hash function for the new hash table is adjusted according to the new size of the hash table
  - Scan the old table and insert all elements one by one in the new hash table
  - Delete the old hash table
  - This approach is also called rehashing
  - Doubling the size reduces the amortized cost of an insert operation

Rehashing

- Time complexity :- **O(n)**

O(n) if key distribution is uniform

Delete

- If there are too many deleted slots in that hash table, the search may unnecessarily take a lot of time

- How can we get rid of the deleted slots to search faster?

Delete

- If there are too many deleted slots in that hash table, the search may unnecessarily take a lot of time

- How can we get rid of the deleted slots to search faster?
    - We can use a **rehash** operation in which the size of the new hash table is the same as the old hash table

Decision tree

References

- Chapter-7.9 from Mark Allen Weiss
- Chapter-4.4 from Goodrich and Tamassia

Decision tree

- The binary search take O(log n) time to search an element from the sorted array
- If the hash table is sufficiently large, the search can be done in O(1)
- What is the key difference between binary search and the hash table search?

Decision tree

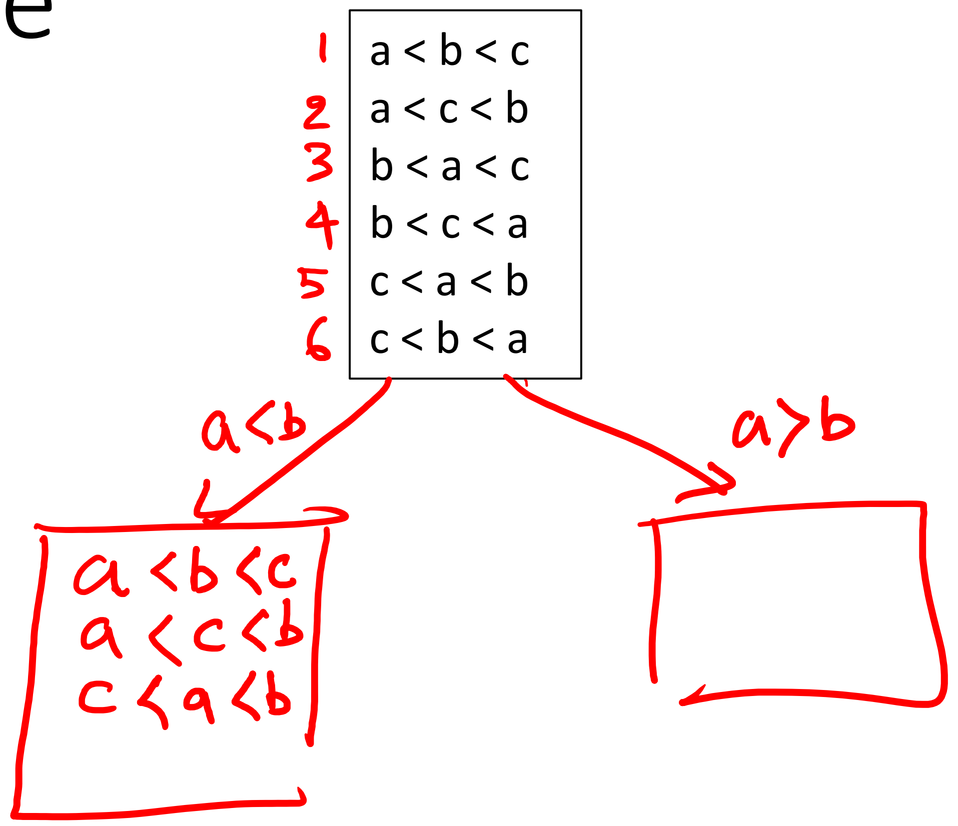
- The binary search doesn’t care about the type of the key
    - It needs a comparison function that acts as a black box
    - The return value of the comparison function could be <, >, <=, >=, ==, !=

- The hash table will not work if the key can’t be mapped to an integer

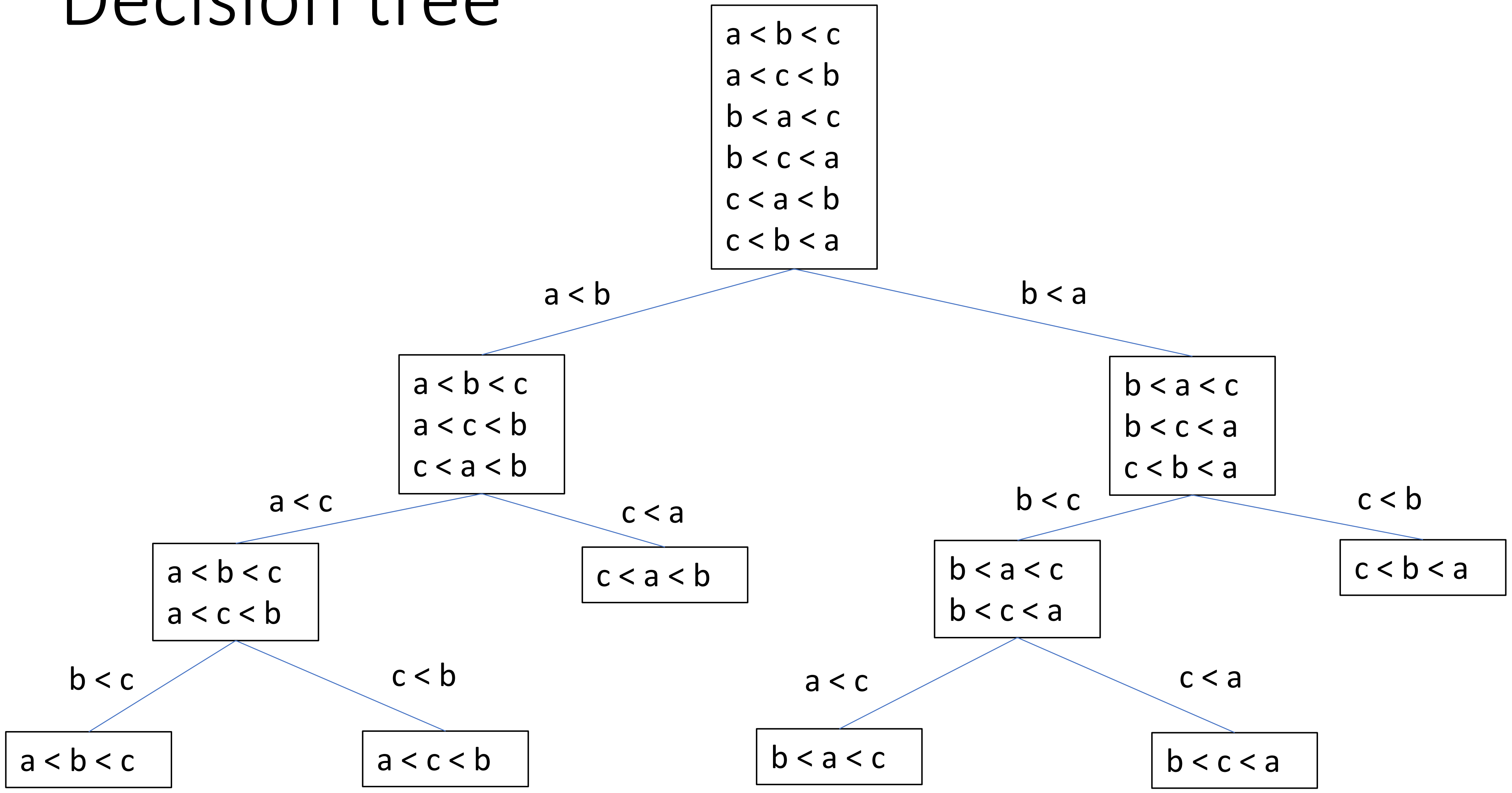
Decision tree

- The decision tree is an abstraction to find a lower bound
- We'll use a binary decision tree to find the lower bound of sorting algorithms
- Each internal node in the decision tree is a comparison operation that may take place during the execution of the algorithm
- The leaf nodes are the output of the algorithm
- The goal is to obtain a lower bound on the number of comparisons to generate the output

Decision tree



Decision tree



Decision tree

- The previous slide shows a decision tree for sorting three distinct elements a, b, c using just the comparison operations
- The leaf nodes correspond to the possible outputs of the program
- Notice that for three elements, there could be six different possible outcomes; the program may take different paths during runtime to generate an outcome

Decision tree

- Notice that, in general, a program may take different paths to derive the same output
- Therefore, the number of leaves is at least the number of all possible outcomes
- The internal nodes of the decision tree are comparison operations
- After every comparison operation, the program may take two different paths, and the only operations we care about are the comparison operations; therefore, the decision tree is a binary tree

Decision tree

- The lower bound on the number of comparisons is the height of the decision tree

- The height is minimum when the decision tree is a nearly complete or complete binary tree, and the leaves contain only the possible outputs

Decision tree

- What is the number of leaf nodes in the decision tree for sorting when we want to sort n numbers?

Decision tree

- What is the number of leaf nodes in the decision tree for sorting when we want to sort n numbers?
    - n!

- What is the height, h, of the decision tree when the number of leaves is n!

Decision tree

- What is the number of leaf nodes in the decision tree for sorting when we want to sort n numbers?
  - n!
- What is the height, h, of the decision tree when the number of leaves is n!
  - $2^h >= n!$
  - h >= log(n!)

The minimum number of comparisons = log(n!)

Decision tree

- What is the number of leaf nodes in the decision tree for sorting when we want to sort n numbers?
  - n!

- What is the height, h, of the decision tree when the number of leaves is n!
  - 2^h >= n!
  - h >= log(n!)

  The minimum number of comparisons = log(n!)

  $$\log(n!) \geq \log\left(\frac{n}{2}^{\frac{n}{2}}\right) \geq \frac{n}{2} * \log\left(\frac{n}{2}\right)$$

  $$\geq \frac{n}{2} * (\log(n) - 1) = \Omega(n * \log(n))$$