Unit 15: Hash Tables

Anthony Estey

CSC 115: Fundamentals of Programming II

University of Victoria

Unit 15 Overview

- Learning Objectives: (You should be able to…)
  - describe the benefits of hashing
  - evaluation collision resolution policies
  - compare and contrast open-addressing and chaining
  - describe and apply insert, delete, and find operations using various open-addressing and chaining schemes

Recap: Map ADT

- Store **values** associated with **keys**
  - Key and Values can be any type
  - Keys are typically a *comparable* type

- Generally, we want to do be able to efficiently *insert*, *find*, and *remove* elements from our collection

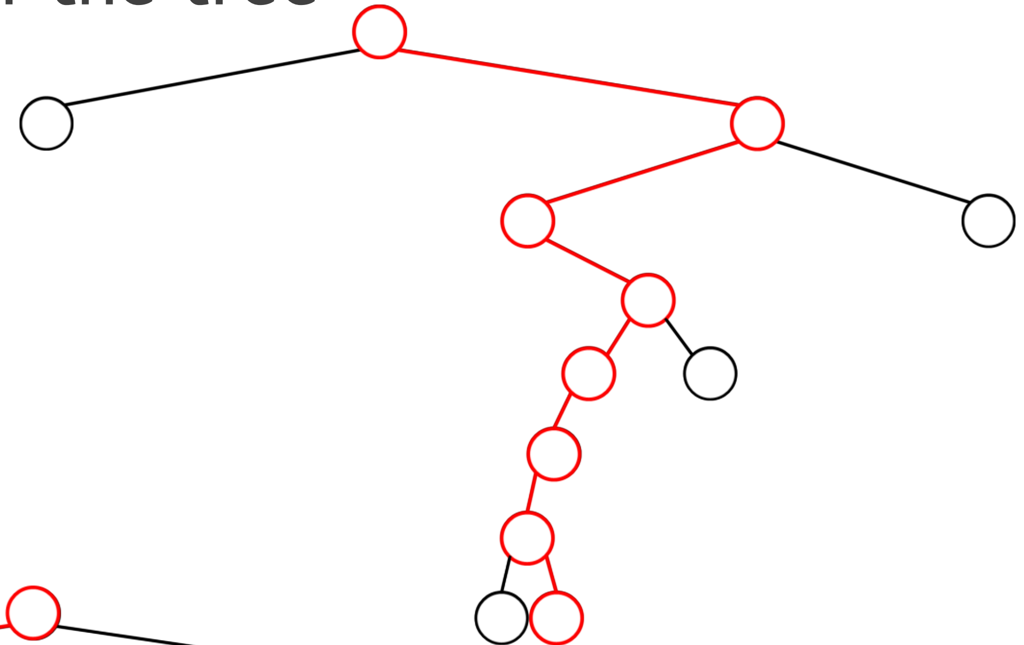
Running Times for Common Operations

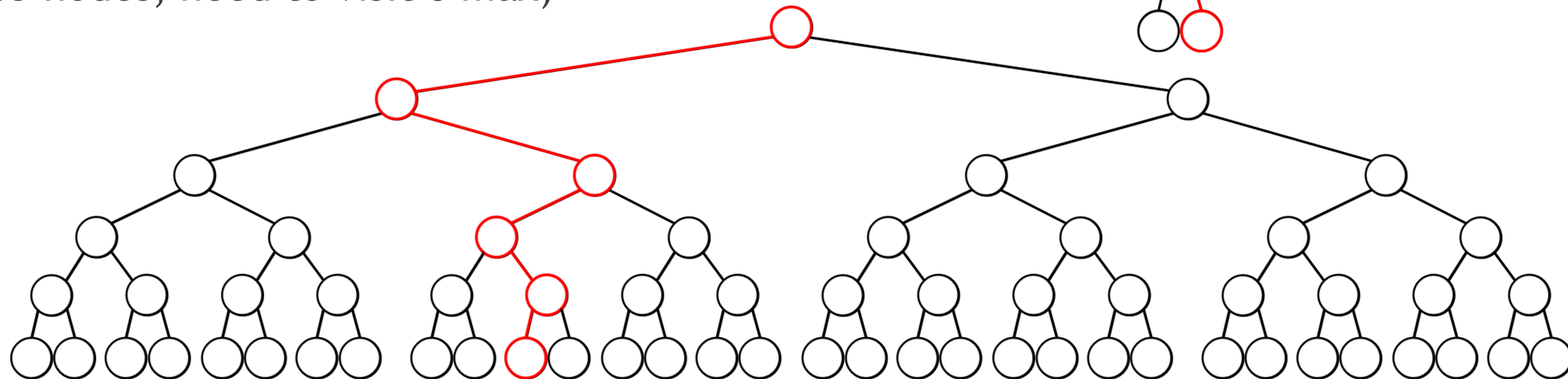
- Worst-case time complexity:

	Insert	Remove	Find
Unordered array	$O(1)$	$O(n)$	$O(n)$
Ordered array	$O(n)$	$O(n)$	$O(\log n)$
Unordered list	$O(1)$	$O(n)$	$O(n)$
Ordered list	$O(n)$	$O(n)$	$O(n)$
BST	$O(\text{height})$	$O(\text{height})$	$O(\text{height})$

BST operation run time

- BST operations are bound by the height of the tree
  - Worst-case:
    - Tree is not balanced
    - $O(n)$
  - Best-case:
    - Tree is balanced
    - $O(\log n)$ (63 nodes, need to visit 6 max)





Running Times for Common Operations

- Worst-case time complexity:

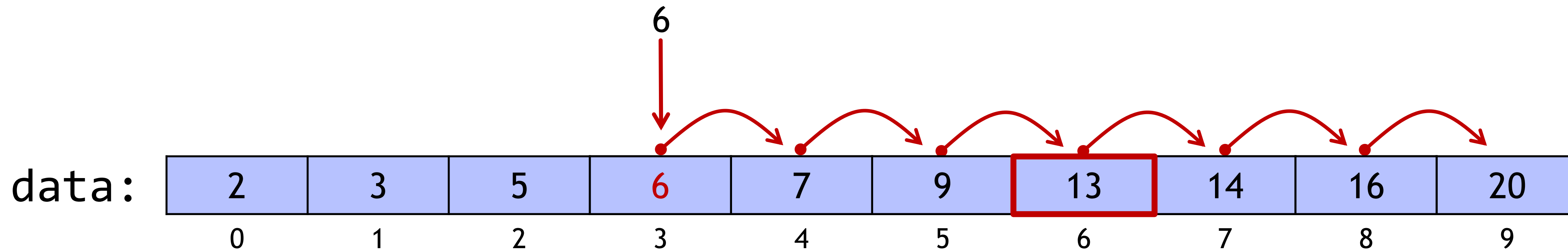
	Insert	Remove	Find
Unordered array	$O(1)$	$O(n)$	$O(n)$
Ordered array	$O(n)$	$O(n)$	$O(\log n)$
Unordered list	$O(1)$	$O(n)$	$O(n)$
Ordered list	$O(n)$	$O(n)$	$O(n)$
BST	$O(n)$	$O(n)$	$O(n)$
BST (balanced)	$O(\log n)$	$O(\log n)$	$O(\log n)$

If we can keep our BSTs balanced, they have very good runtime across all operations!

But can we do better?

Array operatons

- Arrays have efficient $O(1)$ access
  - For example, we can access the element at index 6 right away (data[6])
- We can also assign a value to an array at a specific index in $O(1)$ time
  - The problem is that to maintain order, we need to shuffle all other elements



Hash Tables

- Hash tables consist of an array to store data
  - Can* complete *insert*, *find*, and *remove* operations in $O(1)$ time
- Data often consists of complex types, or pointers to such objects
  - One attribute of the object is designated as the table's **key**
- A *hash function* maps the key to an array index
  - The **key** is converted to an integer
  - The integer mapped to an array index using some function (often the modulo function)

Hash Functions

- Using knowledge of the kind and number of keys to be stored, we should choose/design a hash function so that it is:
  - fast to compute
  - causes few collisions

- A collision occurs when there is already an element in the array at the index we wish to insert into
  - There are a number of ways to handle collisions efficiently that we will explore throughout this unit

Hash table – first example

- For data with numeric keys, we might use $hash(x) = x \bmod m$, where m is the size of the table, (for this example we will assume *m* is larger than the number of keys we expect to store.)

- Example: *hash(x) = x mod 7*
  - insert(4)
  - insert(17)
  - find(12)
  - insert(9)
  - delete(17)

0	
1	
2	9
3	17
4	4
5	12
6	

$m = 7$

Collision Handling

- A collision occurs when two keys are mapped to the same index
  - Collisions may occur even when the hash function is good
  - Inevitable if enough elements are added to the hash table

- Collisions start happening more frequency with higher **load factors**
  - The **load factor**, α, is defined as follows:

$$\alpha = (\text{number of elements})/(\text{size of array})$$

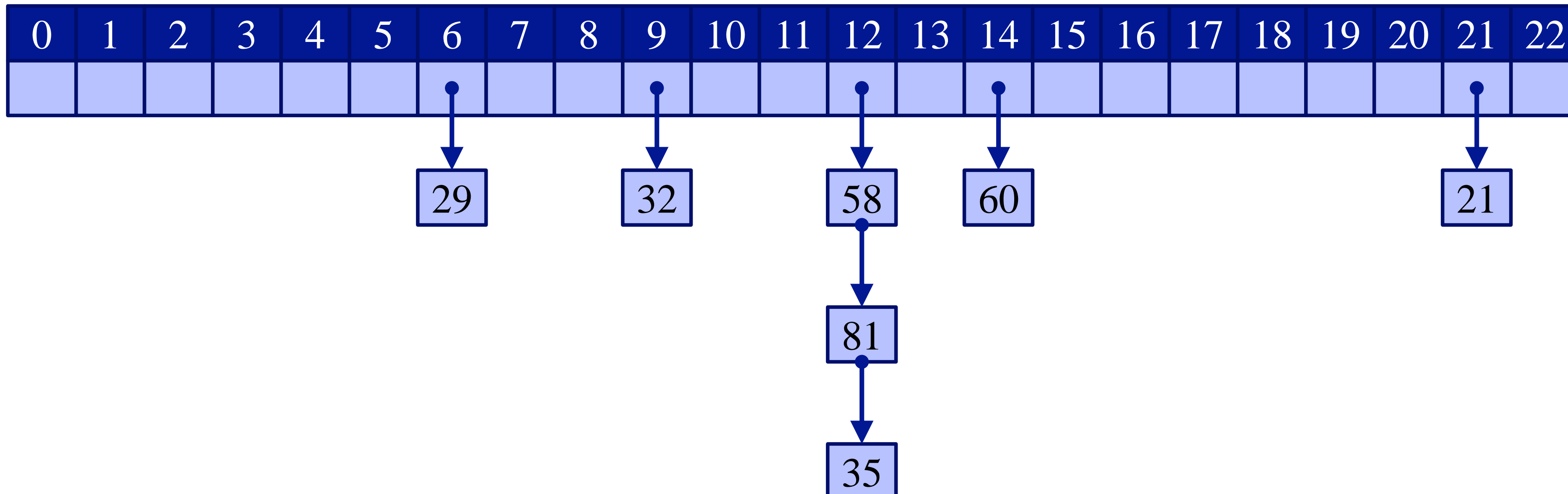
- There are two main ways of dealing with collisions
  - Open addressing
  - Separate chaining

Separate Chaining

- Each entry in the hash table is a pointer to a linked list (or other dictionary-compatible data structure)
  - If a collision occurs the new item is added to the end of the list at the appropriate location
- Performance degrades less rapidly using separate chaining
  - with uniform random distribution, separate chaining maintains good performance even at high load factors $\alpha > 1$

Separate Chaining Example

- $h(x) = x \bmod 23$
- Insert 81, $h(x) = 12$, add to back of the linked list
- Insert 35, $h(x) = 12$
- Insert 60, $h(x) = 14$



Open Addressing

- When an insertion results in a collision, find an empty spot in the array
  - Start at the index to which the hash function mapped the inserted item
  - Look for a free space in the array following a particular search pattern, known as *probing*
- There are three major open addressing schemes
  - Linear probing
  - Quadratic probing
  - Double hashing

Linear Probing

- The hash table is searched sequentially
  - Starting with the original hash location
  - For each time the table is probed (for a free location) add one to the index
    - If the sequence of probes reaches the last element of the array, wrap around to *arr*[0]

Linear Probing Example

- Hash table length is 23 $(m = 23)$
- The hash function, $h(k) = k \ mod \ 23$, where k is the search key value
- The search key values are shown in the table

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
						29			32			58									21	

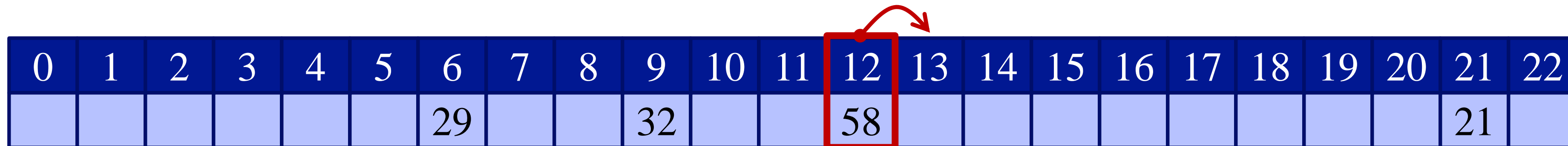
Linear Probing Example

- Insert 49:
- $h(49) = 49\ mod\ 23 = 3$
- So 49 is inserted into index 3

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
						29			32			58									21	

Linear Probing Example

- Insert 80: $h(81) = 81\, mod\; 23 = 12$
- Which collides with 58 so use linear probing to find a free space
- First look at 12 + 1, which is free so insert the item at index 13



Linear Probing Example

- Insert 35: $h(35) = 35\ mod\ 23 = 12$
- Which collides with 58 so use linear probing to find a free space
- First look at 12 + 1, which is occupied so look at 12 + 2 and insert the item at index 14

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
						29			32			58	81								21	

Linear Probing Example

- Insert 12: $h(12) = 12 \, mod \, 23 = 12$
- The item will be inserted at index 16
- **Primary clustering** is beginning to develop, making insertions of keys for which the hash function maps to index 12 less efficient

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
						29			32			58	81	35	60						21	

Linear Probing - problems

- Linear probing leads to ***primary clustering***
  - The table contains groups of consecutively occupied locations
  - These clusters tend to get larger as time goes on
    - Reducing the efficiency of the hash table

Linear Probing Search Code

```
Algorithm Find(k):
    Input: the target key to search the hash table for
    Output: the entry with the given key; null if key not found

    p ← h(k) % m
    for i ← 1 to m do
        e ← data[p]
        if e is null then
            return null
        end if
        if e's key is equal to k then
            return e
        end if
        p ← (p + 1) % m
    end for
    return null
end
```

Quadratic Probing

- Quadratic probing is a refinement of linear probing that prevents primary clustering
  - For each probe p, add p^2 to the original integer index after each collision
- For example:
  - After first collision, look into table at index $h(k) + 1^2$
  - After second collision, look into table at index $h(k) + 2^2$
  - After third collision, look into table at index $h(k) + 3^2$
  - etc
- Main idea:
  - linear probing – check index $h(x)$, then $h(x) + 1$, then $h(x) + 2$, etc.
  - quadratic probing – check index $h(x)$, $h(x) + 1$, $h(x) + 4$, $h(x) + 9$, etc.

Quadratic Probing Example

- Hash table length is 23 $(m = 23)$
- The hash function, $h(k) = k \bmod 23$, where k is the search key value
- The search key values are shown in the table

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
						29			32			58									21	

Quadratic Probing Example

- Insert 81: $h(81) = 81 \bmod 23 = 12$
- Which collides with 58 so use quadratic probing to find a free space
- First look at $12 + 1^2 = 13$, which is free, so insert the item at index 13

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
						29			32			58									21	

Quadratic Probing Example

- Insert 35: $h(35) = 35 \bmod 23 = 12$
- Which collides with 58 so use quadratic probing to find a free space
- First look at $12 + 1^2 = 13$, which is occupied
- Next, look at $12 + 2^2 = 16$, and insert into index 16 since it is free

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
						29			32			58	81								21	

Quadratic Probing Example

- Insert 12: $h(12) = 12 \bmod 23 = 12$
- Index 14 collides with 58. First look at $12 + 1^2 = 13$, then look at $12 + 2^2 = 16$, then look at $12 + 3^2 = 21$, which is also occupied!
- The next probe index is $12 + 4^2 = 28$. Similar to linear probing, if the index goes past the end of the table it wraps around to index 0

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
						29			32			58	81			35					21	

We will need to make sure to always keep the index values within the bounds of the hash table

$$(12 + 4^2) \bmod m = 28 \bmod 23 = 5$$

Quadratic Probing Example

- Insert 51: $h(31) = 31 \bmod 23 = 8.$

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
					12	29			32			58	81			35					21	

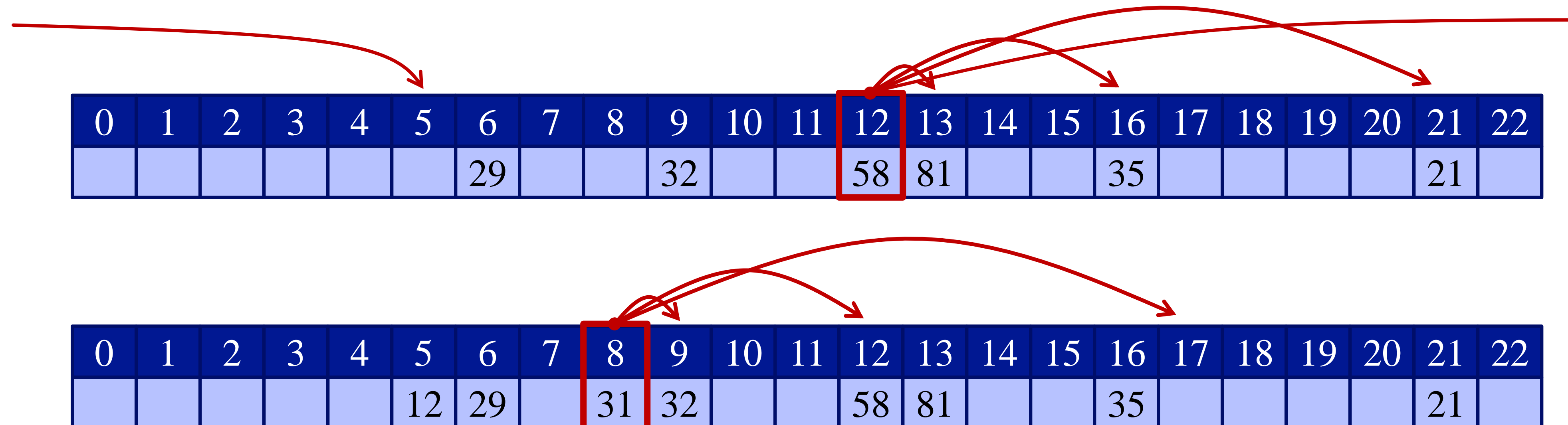
Quadratic Probing Example

- Insert 77: $h(77) = 77 \bmod 23 = 8$.
- Index 8 collides with 31.
- First look at $8 + 1^2 = 9$, then at $8 + 2^2 = 12$, which is also occupied
- Finally, we look at $8 + 3^2 = 17$, which is open

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
					12	29		31	32			58	81			35					21	

Quadratic Probing Example

- We have now seen two different insertion examples where there was a collision at index 12
  - but each probe sequence visited a different index next



Main problem is when keys map to the same *initial* index

Quadratic Probing Problems

- Quadratic probing can result in something called *secondary* clustering
  - keys that hash to the same index all go up in the same sequence
  - this delays the collision resolution for those keys, and sometimes quadratic probing will never find a valid index even when the table is not full!

- We will see an example during lecture!

Quadratic Probing

- Quadratic probing solves the problem of *primary* clustering, but introduces a new problem, *secondary* clustering

- Secondary clustering is not a problem if:
  - the data (keys) are not significantly skewed
  - the hash table is large enough (ie. has enough free space)
  - the hash function scatters the keys evenly across the table

- Main problem: quadratic probing may fail at $\alpha > 0.5$

Double Hashing

- In both linear and quadratic probing, the probe *sequence* is independent of the key
  - for linear probing the probe sequence increases by 1 index each time
  - for quadratic probing, the sequence jumps 1, 4, 9, etc from the original index

- Double hashing produces *key dependent* probe sequence
  - A second hash function, h_2, determine the probe sequence

- The second hash function, h_2, should follow these guidelines:
  - $h_2(k) \neq 0$
  - $h_2 \neq h_1$
  - A typical h_2 function is $h_2(k) = p - (k \bmod p)$ where p is a prime number $< m$

Double Hashing Example

- Hash table length is 23 $(m = 23)$
- The hash function, $h(k) = k \bmod 23$, where k is the search key value
- The second hash function, $h_2(k) = 5 - (k \bmod 5)$

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
						29			32			58									21	

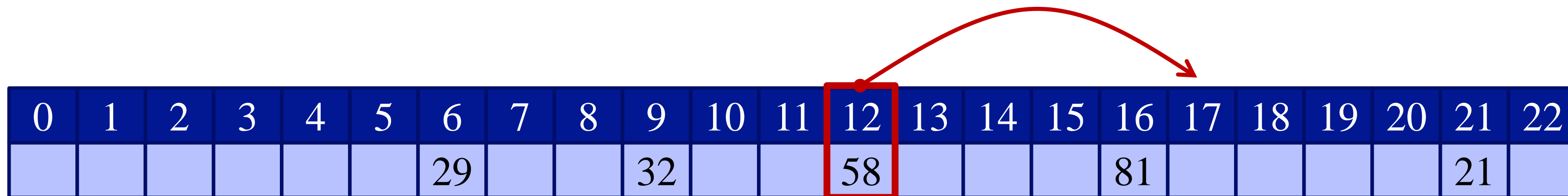
Double Hashing Example

- Insert 81: $h(81) = 81 \bmod 23 = 12$
- Which collides with 58 so use h_2 to find the probe sequence value
- $h_2(81) = 5 - (81 \bmod 5) = 4$
- So the probe sequence for 81 is 4: insert at index $12 + 4 = 16$

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
						29			32			58									21	

Double Hashing Example

- Insert 35: $h(35) = 35 \bmod 23 = 12$
- Which collides with 58 so use h_2 to find the probe sequence value
- $h_2(35) = 5 - (35 \bmod 5) = 5$
- So the probe sequence for 35 is 5: insert at index $12 + 5 = 17$



Double Hashing Example

- Insert 60: $h(60) = 60 \bmod 23 = 14$

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
						29			32			58				81	35				21	

Double Hashing Example

- Insert 83: $h(83) = 83 \bmod 23 = 14$
- Which collides with 60 so use h_2 to find the probe sequence value
- $h_2(83) = 5 - (83 \bmod 5) = 2$
- So the probe sequence for 83 is 3: insert at index $14 + 2 = 16$
- Since index 16 is occupied we will search 2 indexes further to 18

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
						29			32			58		60		81	35				21	

The second hash function determines the probe sequence (the number of spots to jump each time)

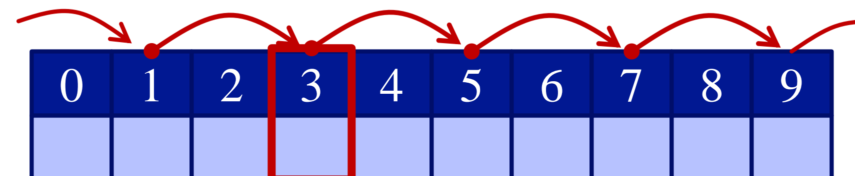
Double Hashing Recap

- linear probing: the probe sequence increases by 1 index each time
- quadratic probing: the probe sequence jumps 1, 4, 9, etc. positions from the original index
- double hashing: the number of positions the sequence jumps each time from the original index is specified by a secondary hash function

- The second hash function, h_2, should follow these guidelines:
  - $h_2(k) \neq 0$
  - $h_2 \neq h_1$
  - A typical h_2 function is $h_2(k) = p - (k \bmod p)$ where p is a prime number $< m$

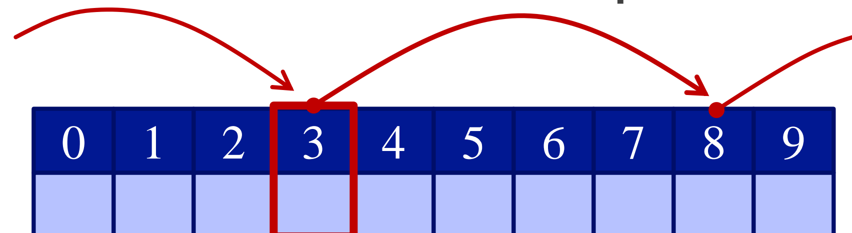
Double Hashing Tips

- It is best for both the table size, m, and the secondary hash function to be prime numbers
- We don't want the secondary hash function to produce a number that is a multiple of the table size
- For example, look at the following table where $m = 10$
  - Look at the indexes visited if the second hash function produced a value of 2:



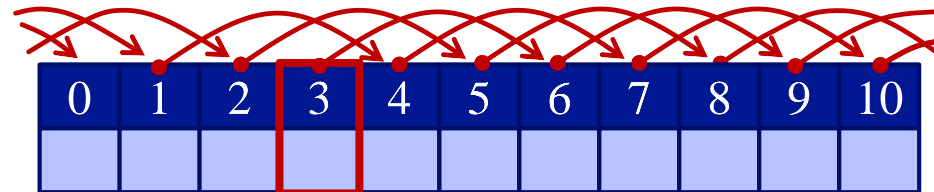
Double Hashing Tips

- It is best for both the table size, m, and the secondary hash function to be prime numbers
- We don't want the secondary hash function to produce a number that is a multiple of the table size
- For example, look at the following table where $m = 10$
  - Look at the indexes visited if the second hash function produced a value of 2:
  - What about with a second hash function sequence of 5?



Double Hashing Tips

- It is best for both the table size, m, and the secondary hash function to be prime numbers
- We don't want the secondary hash function to produce a number that is a multiple of the table size
- For example, look at the following table where $m = 11$
  - Second hash function value of 3:
  - Every index is visited!



Hash Table Removal

- Removals add complexity to hash tables
  - It’s easy to find and remove an element, given the hash function
  - After an element is removed from the table, the space becomes unoccupied
  - The now empty location may make a probe sequence terminate prematurely

Hash Table Removal Example

- Hash table length is 11 $(m = 11)$
- The hash function, $h(k) = k \bmod 11$, where k is the search key value
- Insert 24: $h(24) = 24 \bmod 11 = 2$

0	1	2	3	4	5	6	7	8	9	10
	56		80							

Hash Table Removal Example

- Hash table length is 11 $(m = 11)$
- The hash function, $h(k) = k \bmod 11$, where k is the search key value
- Insert 24: $h(24) = 24 \bmod 11 = 2$
- Insert 12: $h(12) = 12 \bmod 11 = 1$

0	1	2	3	4	5	6	7	8	9	10
	56	24	80							

Hash Table Removal Example

- Hash table length is 11 $(m = 11)$
- The hash function, $h(k) = k \bmod 11$, where k is the search key value
- Insert 24: $h(24) = 24 \bmod 11 = 2$
- Insert 12: $h(12) = 12 \bmod 11 = 1$
- Find 12 ✓

0	1	2	3	4	5	6	7	8	9	10
	56	24	80	12						

```
Algorithm Find(k):
    p ← h(k) % m
    for i ← 1 to m do
        e ← data[p]
        if e is null then
            return null
        end if
        if e's key is equal to k then
            return e
        end if
        p ← (p + 1) % m
    end for
    return null
end
```

Hash Table Removal Example

- Hash table length is 11 $(m = 11)$
- The hash function, $h(k) = k \bmod 11$, where k is the search key value
- Insert 24: $h(24) = 24 \bmod 11 = 2$
- Insert 12: $h(12) = 12 \bmod 11 = 1$
- Find 67 ✘

0	1	2	3	4	5	6	7	8	9	10
	56	24	80	12						

```
Algorithm Find(k):
    p ← h(k) % m
    for i ← 1 to m do
        e ← data[p]
        if e is null then
            return null
        end if
        if e's key is equal to k then
            return e
        end if
        p ← (p + 1) % m
    end for
    return null
end
```

Hash Table Removal Example

- Hash table length is 11 $(m = 11)$
- The hash function, $h(k) = k \bmod 11$, where k is the search key value
- Insert 24: $h(24) = 24 \bmod 11 = 2$
- Insert 12: $h(12) = 12 \bmod 11 = 1$
- Remove 24

0	1	2	3	4	5	6	7	8	9	10
	56	24	80	12						

Algorithm Find(k):
  $p \leftarrow h(k) \% m$
  for $i \leftarrow 1$ to m **do**
    $e \leftarrow \text{data}[p]$
    if e is $null$ **then**
      return $null$
    end if
    if e's key is equal to k **then**
      return e
    end if
    $p \leftarrow (p + 1) \% m$
  end for
  return $null$
end

Hash Table Removal Example

- Hash table length is 11 $(m = 11)$
- The hash function, $h(k) = k \bmod 11$, where k is the search key value
- Insert 24: $h(24) = 24 \bmod 11 = 2$
- Insert 12: $h(12) = 12 \bmod 11 = 1$
- Remove 24
- Find 12 ✘

This is a problem!

0	1	2	3	4	5	6	7	8	9	10
	56		80	12						

```
Algorithm Find(k):
    p ← h(k) % m
    for i ← 1 to m do
        e ← data[p]
        if e is null then
            return null
        end if
        if e's key is equal to k then
            return e
        end if
        p ← (p + 1) % m
    end for
    return null
end
```

Hash Table Removal

- Problem: when we remove an entry, it may make a probe sequence terminate prematurely:
  - Meaning that the find algorithm may determine an entry isn't in the hash table when it actually is!

- Potential solutions:
  - Search through the whole hash table?
  - Rehash items each removal?

- We want our operations to be $O(1)$, not $O(n)$!

Idea: Tombstone

- Mark table locations where an item has been removed
  - In textbooks this mark is often referred to as a tombstone
  - The find algorithm does not identify tombstones as null

- The hash function, $h(k) = k \text{ mod}$
- Remove 24

0	1	2	3	4	5	6	7	8	9	10
	56	--	80	12						

Idea: Tombstone

- Mark table locations where an item has been removed
  - In textbooks this mark is often referred to as a tombstone
  - The find algorithm does not identify tombstones as null

- The hash function, $h(k) = k \bmod$
- Remove 24
- Find 12 ✓

0	1	2	3	4	5	6	7	8	9	10
	56	--	80	12						

Idea: Tombstone

- Mark table locations where an item has been removed
  - In textbooks this mark is often referred to as a tombstone
  - The find algorithm does not identify tombstones as null

- The hash function, $h(k) = k \bmod$
- Remove 24
- Find 12 ✓

0	1	2	3	4	5	6	7	8	9	10
--	56	--	80	12	--		--		--	

If the table becomes clogged with tombstones, then we can re-hash the entries (but it's inefficient)

It's okay if the runtime inefficient re-hashing operation is only performed infrequently