Algorithms and Analysis

Lesson 11: *Make a hash of it*

Hash tables, separate chaining, open addressing, linear/quadratic probing, double hashing

Outline

1. **Why Hash?**

2. Separate Chaining

3. Open Addressing

   • Quadratic Probing
   • Double Hashing

4. Hash Set and Map

Content Addressable Memory

- Suppose we have a list of objects which we want to look up according to its contents

- This is often referred to as **associative memory** structures

- A classical example would be a telephone directory

  ⋆ We look up a name
  ⋆ We want to know the number

- What data structure should we use?

Content Addressable Memory

- Suppose we have a list of objects which we want to look up according to its contents

- This is often referred to as **associative memory** structures

- A classical example would be a telephone directory

  - ⋆ We look up a name
  - ⋆ We want to know the number

- What data structure should we use?

Content Addressable Memory

- Suppose we have a list of objects which we want to look up according to its contents

- This is often referred to as **associative memory** structures

- A classical example would be a telephone directory

  ⋆ We look up a name
  ⋆ We want to know the number

- What data structure should we use?

Content Addressable Memory

- Suppose we have a list of objects which we want to look up according to its contents

- This is often referred to as **associative memory** structures

- A classical example would be a telephone directory

  - ⋆ We look up a name
  - ⋆ We want to know the number

- What data structure should we use?

Content Addressable Memory

- Suppose we have a list of objects which we want to look up according to its contents

- This is often referred to as **associative memory** structures

- A classical example would be a telephone directory

  ★ We look up a name
  ★ We want to know the number

- What data structure should we use?

Lists and Trees

- To find an entry in a normal list takes $\Theta(n)$ operations

- If we had a sorted list we could use "binary search" to reduce this to $\Theta(\log(n))$

  ⋆ We will study binary search later
  ⋆ Maintaining an ordered list is costly ($\Theta(n)$ insertions)

- We could use a binary search tree

  ⋆ Search is $\Theta(\log(n))$
  ⋆ Insertion/deletion is $\Theta(\log(n))$

Lists and Trees

- To find an entry in a normal list takes $\Theta(n)$ operations

- If we had a sorted list we could use "binary search" to reduce this to $\Theta(\log(n))$

  ⋆ We will study binary search later
  ⋆ Maintaining an ordered list is costly ($\Theta(n)$ insertions)

- We could use a binary search tree

  ⋆ Search is $\Theta(\log(n))$
  ⋆ Insertion/deletion is $\Theta(\log(n))$

Lists and Trees

- To find an entry in a normal list takes $\Theta(n)$ operations

- If we had a sorted list we could use "binary search" to reduce this to $\Theta(\log(n))$

  ⋆ We will study binary search later
  ⋆ Maintaining an ordered list is costly ($\Theta(n)$ insertions)

- We could use a binary search tree

  ⋆ Search is $\Theta(\log(n))$
  ⋆ Insertion/deletion is $\Theta(\log(n))$

Lists and Trees

- To find an entry in a normal list takes $\Theta(n)$ operations

- If we had a sorted list we could use "binary search" to reduce this to $\Theta(\log(n))$

  ⋆ We will study binary search later
  ⋆ Maintaining an ordered list is costly ($\Theta(n)$ insertions)

- We could use a binary search tree

  ⋆ Search is $\Theta(\log(n))$
  ⋆ Insertion/deletion is $\Theta(\log(n))$

Lists and Trees

- To find an entry in a normal list takes $\Theta(n)$ operations

- If we had a sorted list we could use "binary search" to reduce this to $\Theta(\log(n))$

  - ⋆ We will study binary search later
  - ⋆ Maintaining an ordered list is costly ($\Theta(n)$ insertions)

- We could use a binary search tree

  - ⋆ Search is $\Theta(\log(n))$
  - ⋆ Insertion/deletion is $\Theta(\log(n))$

Lists and Trees

- To find an entry in a normal list takes $\Theta(n)$ operations

- If we had a sorted list we could use "binary search" to reduce this to $\Theta(\log(n))$

  ⋆ We will study binary search later
  ⋆ Maintaining an ordered list is costly ($\Theta(n)$ insertions)

- We could use a binary search tree

  ⋆ Search is $\Theta(\log(n))$
  ⋆ Insertion/deletion is $\Theta(\log(n))$

Lists and Trees

- To find an entry in a normal list takes $\Theta(n)$ operations

- If we had a sorted list we could use "binary search" to reduce this to $\Theta(\log(n))$

  - ⋆ We will study binary search later
  - ⋆ Maintaining an ordered list is costly ($\Theta(n)$ insertions)

- We could use a binary search tree

  - ⋆ Search is $\Theta(\log(n))$
  - ⋆ Insertion/deletion is $\Theta(\log(n))$

Thinking Outside the Box

- As with many data structures thinking about the problem differently can lead to much better solutions

- Let us consider the content we want to search on as a **key**

- For telephone numbers the key would be the name of the person we want to phone

- We could get $O(1)$ search, insertion and deletion if we used the key as an index into a big array

- That is the key is a string of, say, 100 characters so can be represented by an 800 digit binary number

- We could look up the key in a table of 2^{800} items

Thinking Outside the Box

- As with many data structures thinking about the problem differently can lead to much better solutions

- Let us consider the content we want to search on as a **key**

- For telephone numbers the key would be the name of the person we want to phone

- We could get $O(1)$ search, insertion and deletion if we used the key as an index into a big array

- That is the key is a string of, say, 100 characters so can be represented by an 800 digit binary number

- We could look up the key in a table of 2^{800} items

Thinking Outside the Box

- As with many data structures thinking about the problem differently can lead to much better solutions

- Let us consider the content we want to search on as a **key**

- For telephone numbers the key would be the name of the person we want to phone

- We could get $O(1)$ search, insertion and deletion if we used the key as an index into a big array

- That is the key is a string of, say, 100 characters so can be represented by an 800 digit binary number

- We could look up the key in a table of 2^{800} items

Thinking Outside the Box

- As with many data structures thinking about the problem differently can lead to much better solutions

- Let us consider the content we want to search on as a **key**

- For telephone numbers the key would be the name of the person we want to phone

- We could get $O(1)$ search, insertion and deletion if we used the key as an index into a big array

- That is the key is a string of, say, 100 characters so can be represented by an 800 digit binary number

- We could look up the key in a table of 2^{800} items
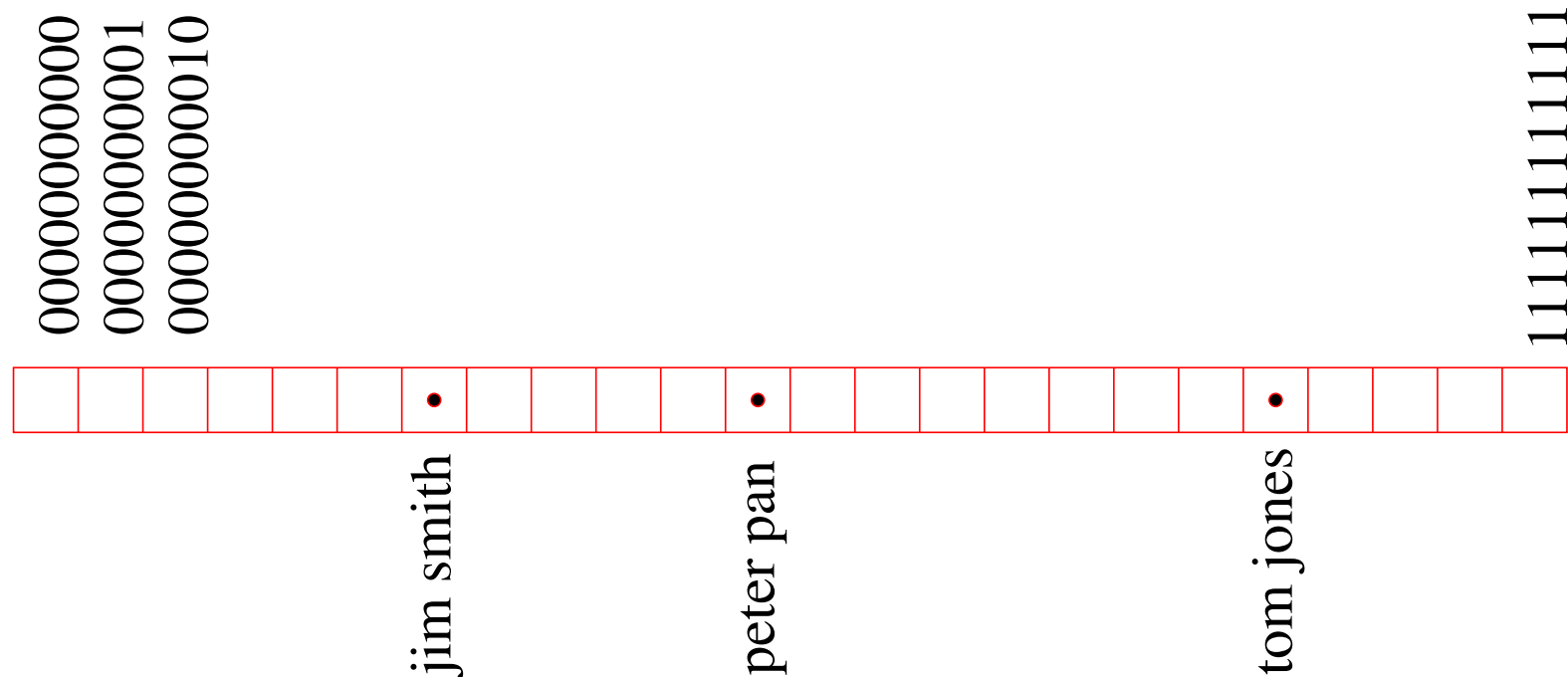

Thinking Outside the Box

- As with many data structures thinking about the problem differently can lead to much better solutions

- Let us consider the content we want to search on as a **key**

- For telephone numbers the key would be the name of the person we want to phone

- We could get $O(1)$ search, insertion and deletion if we used the key as an index into a big array

- That is the key is a string of, say, 100 characters so can be represented by an 800 digit binary number

- We could look up the key in a table of 2^{800} items

Algorithms and Analysis

Thinking Outside the Box

- As with many data structures thinking about the problem differently can lead to much better solutions

- Let us consider the content we want to search on as a **key**

- For telephone numbers the key would be the name of the person we want to phone

- We could get $O(1)$ search, insertion and deletion if we used the key as an index into a big array

- That is the key is a string of, say, 100 characters so can be represented by an 800 digit binary number

- We could look up the key in a table of 2^{800} items

Hashing

- This approach is slightly wasteful of memory

- Almost all memory locations would be empty

- We can save on memory by folding up the table up onto itself

Algorithms and Analysis

Hashing

- This approach is slightly wasteful of memory

- Almost all memory locations would be empty

- We can save on memory by folding up the table up onto itself

Hashing

- This approach is slightly wasteful of memory

- Almost all memory locations would be empty

- We can save on memory by folding up the table up onto itself
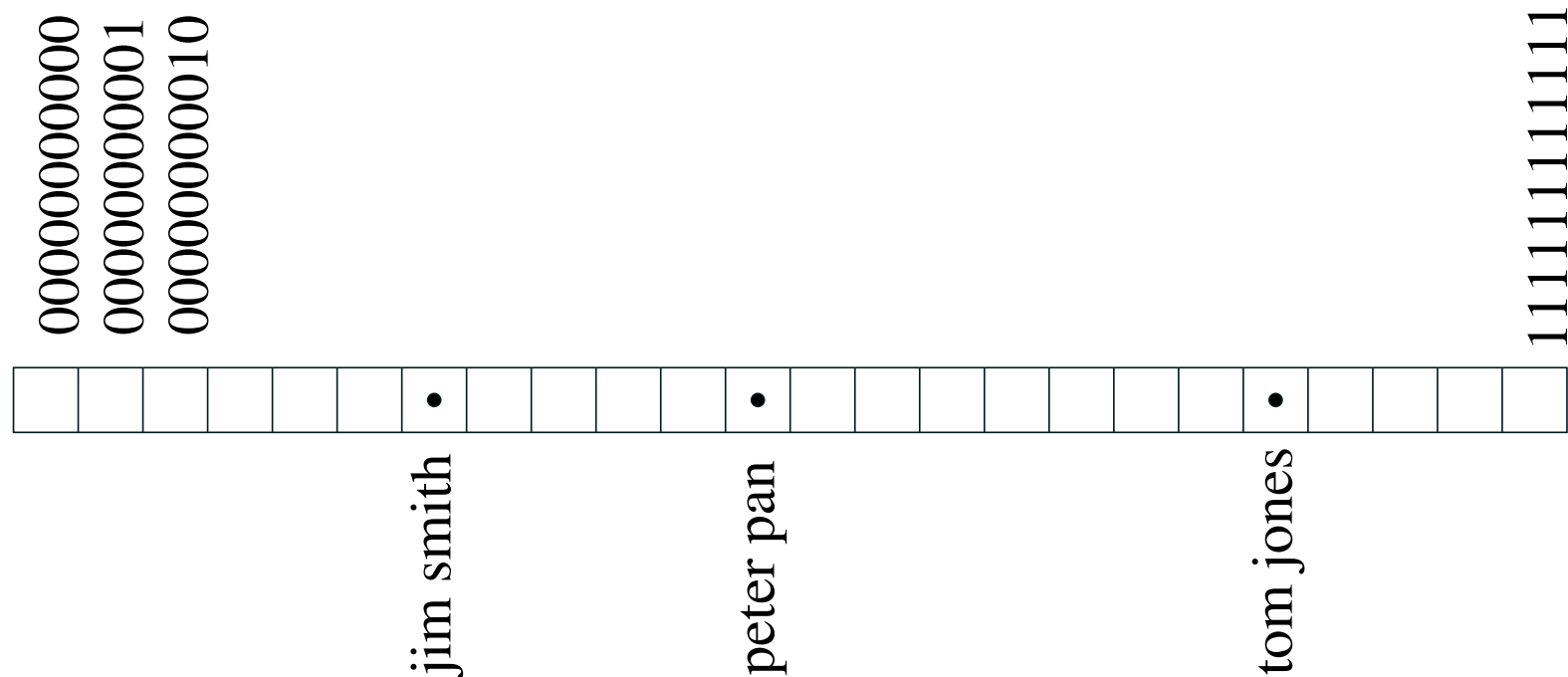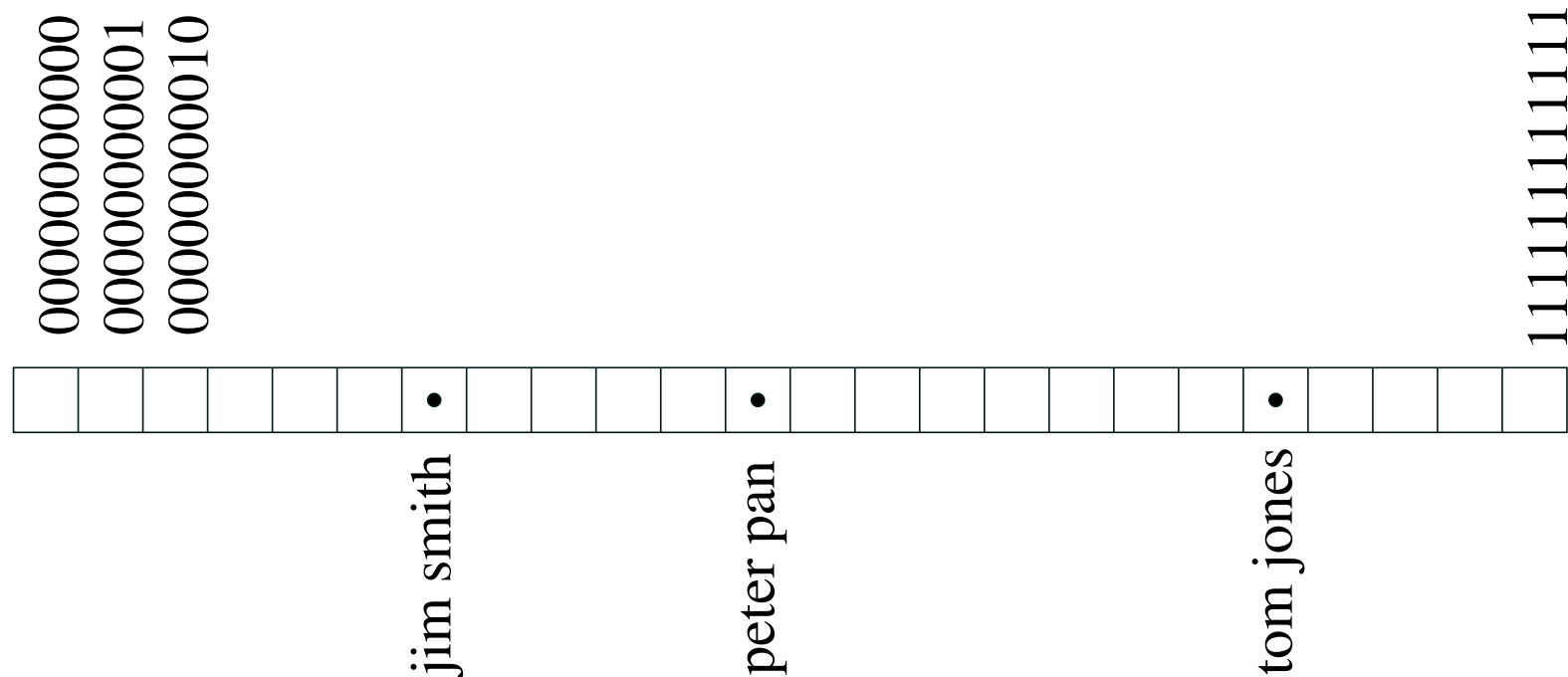
Hashing

- This approach is slightly wasteful of memory

- Almost all memory locations would be empty

- We can save on memory by folding up the table up onto itself

Hashing Codes

- A **hashing function** `hashCode(x)` takes an object, `x`, and returns a positive integer, the **hash code**

- To turn the hash code into an address take the modulus of the table size
  ```
  int index = abs(hashCode(x) % tableSize);
  ```

- If `tableSize` $= 2^n$ we can compute this more efficiently using a mask
  ```
  int index = abs(hashCode(x) & (tableSize -1));
  ```

Hashing Codes

- A **hashing function** `hashCode(x)` takes an object, `x`, and returns a positive integer, the **hash code**

- To turn the hash code into an address take the modulus of the table size

```
int index = abs(hashCode(x) % tableSize);
```

- If $\texttt{tableSize} = 2^n$ we can compute this more efficiently using a mask

```
int index = abs(hashCode(x) & (tableSize -1));
```

Hashing Codes

- A **hashing function** `hashCode(x)` takes an object, `x`, and returns a positive integer, the **hash code**

- To turn the hash code into an address take the modulus of the table size

  ```
  int index = abs(hashCode(x) % tableSize);
  ```

- If `tableSize` $= 2^n$ we can compute this more efficiently using a mask

  ```
  int index = abs(hashCode(x) & (tableSize -1));
  ```

Hashing Functions

- Hashing functions take an object and return an integer

- Hashing functions aren't magic

  ⋆ They tend to add up integers representing the parts of the object

- We want the integers to be close to random so that similar objects are mapped to different integers

- Sometimes two objects will be mapped to the same address—this is known as a **collision**

- Collision resolution is an important part of hashing

Hashing Functions

- Hashing functions take an object and return an integer

- Hashing functions aren't magic

  ⋆ They tend to add up integers representing the parts of the object

- We want the integers to be close to random so that similar objects are mapped to different integers

- Sometimes two objects will be mapped to the same address—this is known as a **collision**

- Collision resolution is an important part of hashing

Hashing Functions

- Hashing functions take an object and return an integer

- Hashing functions aren't magic

  ⋆ They tend to add up integers representing the parts of the object

- We want the integers to be close to random so that similar objects are mapped to different integers

- Sometimes two objects will be mapped to the same address—this is known as a **collision**

- Collision resolution is an important part of hashing

Hashing Functions

- Hashing functions take an object and return an integer

- Hashing functions aren't magic

  ★ They tend to add up integers representing the parts of the object

- We want the integers to be close to random so that similar objects are mapped to different integers

- Sometimes two objects will be mapped to the same address—this is known as a **collision**

- Collision resolution is an important part of hashing

Hashing Functions

- Hashing functions take an object and return an integer

- Hashing functions aren't magic

  ★ They tend to add up integers representing the parts of the object

- We want the integers to be close to random so that similar objects are mapped to different integers

- Sometimes two objects will be mapped to the same address—this is known as a **collision**

- Collision resolution is an important part of hashing

Hashing Functions

- Hashing functions take an object and return an integer

- Hashing functions aren't magic

  ⋆ They tend to add up integers representing the parts of the object

- We want the integers to be close to random so that similar objects are mapped to different integers

- Sometimes two objects will be mapped to the same address—this is known as a **collision**

- Collision resolution is an important part of hashing

Hashing Functions

- Hashing functions take an object and return an integer

- Hashing functions aren't magic

  ⋆ They tend to add up integers representing the parts of the object

- We want the integers to be close to random so that similar objects are mapped to different integers

- Sometimes two objects will be mapped to the same address—this is known as a **collision**

- Collision resolution is an important part of hashing

Hashing Strings

- A strings might be hashed using a function

```cpp
unsigned long long hash(string const& s) {
  unsigned long long results = 12345;

  for (auto ch = s.begin(); ch != s.end(); ++ch) {
    results = 127*results + static_cast<unsigned char>(*ch);
  }
  return results;
}
```

- The numbers 12345 and 127 is to try to prevent clashes

Hashing Strings

- A strings might be hashed using a function

```cpp
unsigned long long hash(string const& s) {
  unsigned long long results = 12345;

  for (auto ch = s.begin(); ch != s.end(); ++ch) {
    results = 127*results + static_cast<unsigned char>(*ch);
  }
  return results;
}
```

- The numbers 12345 and 127 is to try to prevent clashes

Hashing Strings

- A strings might be hashed using a function

```cpp
unsigned long long hash(string const& s) {
  unsigned long long results = 12345;

  for (auto ch = s.begin(); ch != s.end(); ++ch) {
    results = 127*results + static_cast<unsigned char>(*ch);
  }
  return results;
}
```

- The numbers 12345 and 127 is to try to prevent clashes

Hashing Strings

- A strings might be hashed using a function

```cpp
unsigned long long hash(string const& s) {
  unsigned long long results = 12345;

  for (auto ch = s.begin(); ch != s.end(); ++ch) {
    results = 127*results + static_cast<unsigned char>(*ch);
  }
  return results;
}
```

- The numbers 12345 and 127 is to try to prevent clashes—there are lots of alternatives

Hashing Strings

- A strings might be hashed using a function

```cpp
unsigned long long hash(string const& s) {
  unsigned long long results = 12345;

  for (auto ch = s.begin(); ch != s.end(); ++ch) {
    results = 127*results + static_cast<unsigned char>(*ch);
  }
  return results;
}
```

- The numbers 12345 and 127 is to try to prevent clashes—there are lots of alternatives

- What we want is that strings that might be similar receive very different hash codes

---

DIY

- The `unordered_set<T, Hash<T> >` allows you to define your own hash function

- By default this is set to `std::hash<T>(T)`

- Not all classes have hash function defined so you will need to do this

- Care is needed to make you hash function produce near random hash codes

DIY

- The `unordered_set<T, Hash<T> >` allows you to define your own hash function

- By default this is set to `std::hash<T>(T)`

- Not all classes have hash function defined so you will need to do this

- Care is needed to make you hash function produce near random hash codes

DIY

- The `unordered_set<T, Hash<T> >` allows you to define your own hash function

- By default this is set to `std::hash<T>(T)`

- Not all classes have hash function defined so you will need to do this

- Care is needed to make you hash function produce near random hash codes

DIY

- The `unordered_set<T, Hash<T> >` allows you to define your own hash function

- By default this is set to `std::hash<T>(T)`

- Not all classes have hash function defined so you will need to do this

- Care is needed to make you hash function produce near random hash codes

Outline

1. Why Hash?

2. **Separate Chaining**

3. Open Addressing

    • Quadratic Probing
    • Double Hashing

4. Hash Set and Map

Collision Resolution

- Collisions are inevitable and must be dealt with

- There are two commonly used strategies

  - ★ Separate chaining—make a hash table of lists
  - ★ Open addressing—find a new position in the hash table

- Collisions add computational cost

- They occur when the hash table becomes full

- If the hash table becomes too full then it may need to be resized

---

Collision Resolution

- Collisions are inevitable and must be dealt with

- There are two commonly used strategies

  ⋆ Separate chaining—make a hash table of lists
  ⋆ Open addressing—find a new position in the hash table

- Collisions add computational cost

- They occur when the hash table becomes full

- If the hash table becomes too full then it may need to be resized

Collision Resolution

- Collisions are inevitable and must be dealt with

- There are two commonly used strategies

  - ⋆ Separate chaining—make a hash table of lists
  - ⋆ Open addressing—find a new position in the hash table

- Collisions add computational cost

- They occur when the hash table becomes full

- If the hash table becomes too full then it may need to be resized

Collision Resolution

- Collisions are inevitable and must be dealt with

- There are two commonly used strategies

  ⋆ Separate chaining—make a hash table of lists
  ⋆ Open addressing—find a new position in the hash table

- Collisions add computational cost

- They occur when the hash table becomes full

- If the hash table becomes too full then it may need to be resized

---

Collision Resolution

- Collisions are inevitable and must be dealt with

- There are two commonly used strategies

  - ⋆ Separate chaining—make a hash table of lists
  - ⋆ Open addressing—find a new position in the hash table

- Collisions add computational cost

- They occur when the hash table becomes full

- If the hash table becomes too full then it may need to be resized

---

Collision Resolution

- Collisions are inevitable and must be dealt with

- There are two commonly used strategies

  ⋆ Separate chaining—make a hash table of lists
  ⋆ Open addressing—find a new position in the hash table

- Collisions add computational cost

- They occur when the hash table becomes full

- If the hash table becomes too full then it may need to be resized

Collision Resolution

- Collisions are inevitable and must be dealt with

- There are two commonly used strategies

  ⋆ Separate chaining—make a hash table of lists
  ⋆ Open addressing—find a new position in the hash table

- Collisions add computational cost

- They occur when the hash table becomes full

- If the hash table becomes too full then it may need to be resized

---

Resizing a Hash Table

- **Resizing a hash table is easy**

  - ★ Create a new hash table of, say, twice the size
  - ★ Iterate through the old hash table adding each element to the new hash table

- Note that you have to recompute all the hash codes

- Resizing a hash table has a modest amortised cost, but can give you a very hiccupy performance

- The size of a hash table is a classic example of a memory-space versus execution time trade off—using bigger (sparser) hash tables speeds up performance

---

Resizing a Hash Table

- Resizing a hash table is easy

    ⋆ Create a new hash table of, say, twice the size
    ⋆ Iterate through the old hash table adding each element to the new hash table

- Note that you have to recompute all the hash codes

- Resizing a hash table has a modest amortised cost, but can give you a very hiccupy performance

- The size of a hash table is a classic example of a memory-space versus execution time trade off—using bigger (sparser) hash tables speeds up performance

Resizing a Hash Table

- Resizing a hash table is easy

  ⋆ Create a new hash table of, say, twice the size
  ⋆ Iterate through the old hash table adding each element to the new hash table

- Note that you have to recompute all the hash codes

- Resizing a hash table has a modest amortised cost, but can give you a very hiccupy performance

- The size of a hash table is a classic example of a memory-space versus execution time trade off—using bigger (sparser) hash tables speeds up performance

Resizing a Hash Table

- Resizing a hash table is easy

  - ⋆ Create a new hash table of, say, twice the size
  - ⋆ Iterate through the old hash table adding each element to the new hash table

- Note that you have to recompute all the hash codes

- Resizing a hash table has a modest amortised cost, but can give you a very hiccupy performance

- The size of a hash table is a classic example of a memory-space versus execution time trade off—using bigger (sparser) hash tables speeds up performance

Resizing a Hash Table

- Resizing a hash table is easy

  - ★ Create a new hash table of, say, twice the size
  - ★ Iterate through the old hash table adding each element to the new hash table

- Note that you have to recompute all the hash codes

- Resizing a hash table has a modest amortised cost, but can give you a very hiccupy performance

- The size of a hash table is a classic example of a memory-space versus execution time trade off—using bigger (sparser) hash tables speeds up performance

Resizing a Hash Table

- Resizing a hash table is easy

  ⋆ Create a new hash table of, say, twice the size
  ⋆ Iterate through the old hash table adding each element to the new hash table

- Note that you have to recompute all the hash codes

- Resizing a hash table has a modest amortised cost, but can give you a very hiccupy performance

- The size of a hash table is a classic example of a memory-space versus execution time trade off—using bigger (sparser) hash tables speeds up performance

Resizing a Hash Table

- Resizing a hash table is easy

  ⋆ Create a new hash table of, say, twice the size
  ⋆ Iterate through the old hash table adding each element to the new hash table

- Note that you have to recompute all the hash codes

- Resizing a hash table has a modest amortised cost, but can give you a very hiccupy performance

- The size of a hash table is a classic example of a memory-space versus execution time trade off—using bigger (sparser) hash tables speeds up performance

Separate Chaining

- In separate chaining we build a singly-linked list at each table entry

Separate Chaining

- In separate chaining we build a singly-linked list at each table entry

$$\left(d_2 d_1\right) \rightarrow \boxed{d_2 + 3d_1} \rightarrow \left(h\right) \rightarrow \boxed{h\%10} \rightarrow \left(i\right)$$

loading factor, $\lambda = \frac{0}{10}$

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

Separate Chaining

- In separate chaining we build a singly-linked list at each table entry

$\text{add}(34)$

$34 \rightarrow \boxed{d_2 + 3d_1} \rightarrow 15 \rightarrow \boxed{h\%10} \rightarrow 5$

loading factor, $\lambda = \dfrac{1}{10}$

0	null
1	null
2	null
3	null
4	null
5	•
6	null
7	null
8	null
9	null

$5 \rightarrow \boxed{34 \mid \text{null}}$

Separate Chaining

- In separate chaining we build a singly-linked list at each table
  entry

$$\text{add}(12)$$

$$12 \rightarrow \boxed{d_2 + 3d_1} \rightarrow 7 \rightarrow \boxed{h\%10} \rightarrow 7$$

$$\text{loading factor, } \lambda = \frac{2}{10}$$

0	null
1	null
2	null
3	null
4	null
5	•	→	34	null
6	null
7	•	→	12	null
8	null
9	null

Separate Chaining

- In separate chaining we build a singly-linked list at each table entry

$\text{add}(55)$

$55 \rightarrow \boxed{d_2 + 3d_1} \rightarrow 20 \rightarrow \boxed{h\%10} \rightarrow 0$

loading factor, $\lambda = \dfrac{3}{10}$

0	• →	55	null
1	null
2	null
3	null
4	null
5	• →	34	null
6	null
7	• →	12	null
8	null
9	null

Separate Chaining

- In separate chaining we build a singly-linked list at each table entry

$$\text{add}(17)$$

$$17 \rightarrow \boxed{d_2 + 3d_1} \rightarrow 22 \rightarrow \boxed{h\%10} \rightarrow 2$$

$$\text{loading factor, } \lambda = \frac{4}{10}$$

0	• → $\boxed{55 \mid \text{null}}$
1	null
2	• → $\boxed{17 \mid \text{null}}$
3	null
4	null
5	• → $\boxed{34 \mid \text{null}}$
6	null
7	• → $\boxed{12 \mid \text{null}}$
8	null
9	null

Separate Chaining

- In separate chaining we build a singly-linked list at each table entry

$\text{add}(63)$

loading factor, $\lambda = \frac{5}{10}$

$63 \rightarrow \boxed{d_2 + 3d_1} \rightarrow 15 \rightarrow \boxed{h\%10} \rightarrow 5$

0	•	→	55	null
1	null
2	•	→	17	null
3	null
4	null
5	•	→	34	•	→	63	null
6	null
7	•	→	12	null
8	null
9	null

Separate Chaining

- In separate chaining we build a singly-linked list at each table entry

$\text{add}(14)$

$$14 \rightarrow \boxed{d_2 + 3d_1} \rightarrow 13 \rightarrow \boxed{h\%10} \rightarrow 3$$

loading factor, $\lambda = \dfrac{6}{10}$

0	•	$\rightarrow$ 55 | null
1	null	
2	•	$\rightarrow$ 17 | null
3	•	$\rightarrow$ 14 | null
4	null	
5	•	$\rightarrow$ 34 | • $\rightarrow$ 63 | null
6	null	
7	•	$\rightarrow$ 12 | null
8	null	
9	null	

Separate Chaining

- In separate chaining we build a singly-linked list at each table entry

add(88)

$$88 \rightarrow \boxed{d_2 + 3d_1} \rightarrow 32 \rightarrow \boxed{h\%10} \rightarrow 2$$

loading factor, $\lambda = \frac{7}{10}$

0	• → $\boxed{55 \mid \text{null}}$
1	null
2	• → $\boxed{17 \mid •}$ → $\boxed{88 \mid \text{null}}$
3	• → $\boxed{14 \mid \text{null}}$
4	null
5	• → $\boxed{34 \mid •}$ → $\boxed{63 \mid \text{null}}$
6	null
7	• → $\boxed{12 \mid \text{null}}$
8	null
9	null

Search

- To find an entry in a hash table we again use the hash function on a key to find the table entry and then we search the list

- The time complexity depends on where objects are hashed

- If the objects are evenly dispersed in the table, search (and insertion) is $\Omega(1)$

- If the objects are hashed to the same entry in the hash table then search is $O(n)$

- Provided you have a good hashing function and the hash table isn't too full you can expect $\Theta(1)$ average case performance

Search

- To find an entry in a hash table we again use the hash function on a key to find the table entry and then we search the list

- The time complexity depends on where objects are hashed

- If the objects are evenly dispersed in the table, search (and insertion) is $\Omega(1)$

- If the objects are hashed to the same entry in the hash table then search is $O(n)$

- Provided you have a good hashing function and the hash table isn't too full you can expect $\Theta(1)$ average case performance

Search

- To find an entry in a hash table we again use the hash function on a key to find the table entry and then we search the list

- The time complexity depends on where objects are hashed

- If the objects are evenly dispersed in the table, search (and insertion) is $\Omega(1)$

- If the objects are hashed to the same entry in the hash table then search is $O(n)$

- Provided you have a good hashing function and the hash table isn't too full you can expect $\Theta(1)$ average case performance
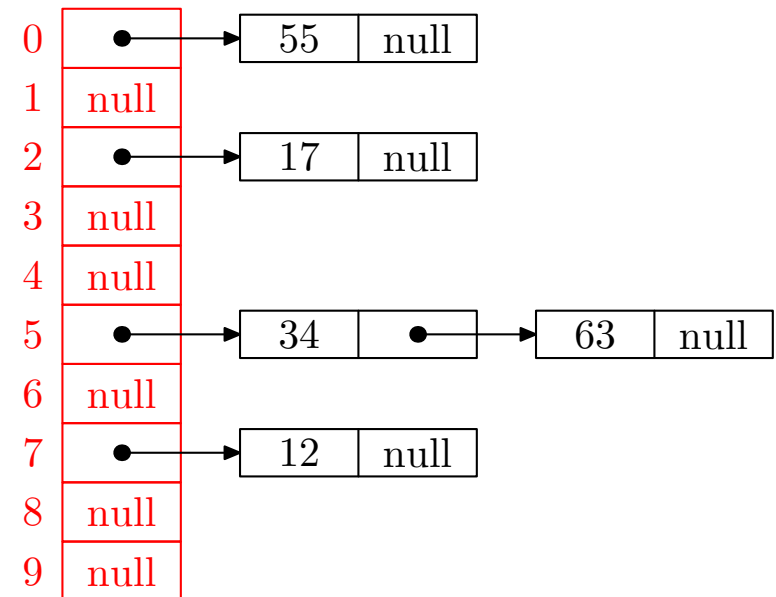
Search

- To find an entry in a hash table we again use the hash function on a key to find the table entry and then we search the list

- The time complexity depends on where objects are hashed

- If the objects are evenly dispersed in the table, search (and insertion) is $\Omega(1)$

- If the objects are hashed to the same entry in the hash table then search is $O(n)$

- Provided you have a good hashing function and the hash table isn't too full you can expect $\Theta(1)$ average case performance

Search

- To find an entry in a hash table we again use the hash function on a key to find the table entry and then we search the list

- The time complexity depends on where objects are hashed

- If the objects are evenly dispersed in the table, search (and insertion) is $\Omega(1)$

- If the objects are hashed to the same entry in the hash table then search is $O(n)$

- Provided you have a good hashing function and the hash table isn't too full you can expect $\Theta(1)$ average case performance
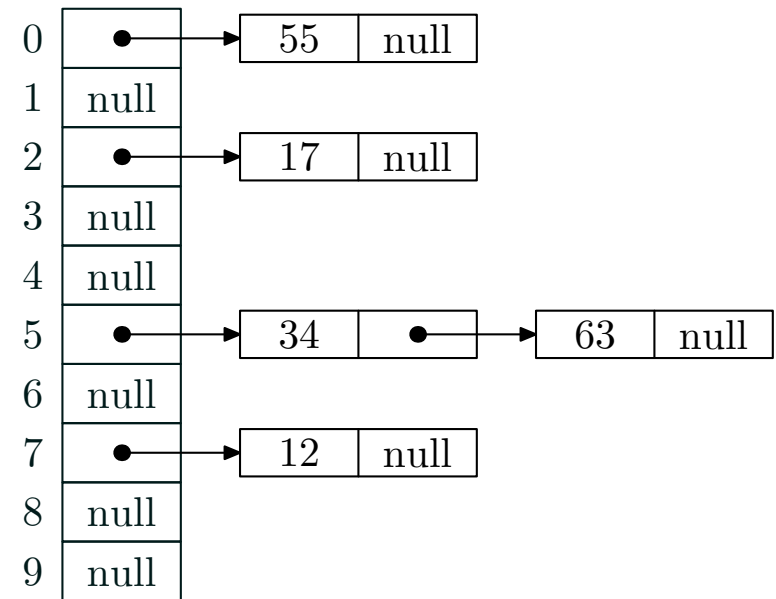
Iterating Over a Hash Table

- To iterate over a hash table we

  ★ Iterate through the array
  ★ At each element we iterate through the linked list

- The order of the elements appears random

- This becomes more efficient as the table becomes fuller

55, 17, 34, 63, 12

Iterating Over a Hash Table

- To iterate over a hash table we

  ⋆ Iterate through the array
  ⋆ At each element we iterate through the linked list

- The order of the elements appears random

- This becomes more efficient as the table becomes fuller

55, 17, 34, 63, 12
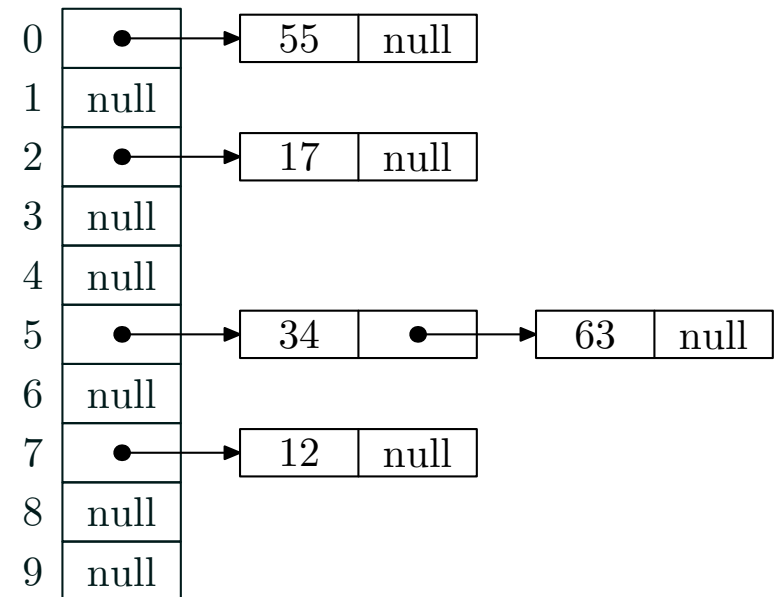
Iterating Over a Hash Table

- To iterate over a hash table we

  ⋆ Iterate through the array
  ⋆ At each element we iterate through the linked list

- The order of the elements appears random

- This becomes more efficient as the table becomes fuller

```
0  [ • ] ────▶ [ 55 | null ]
1  [ null ]
2  [ • ] ────▶ [ 17 | null ]
3  [ null ]
4  [ null ]
5  [ • ] ────▶ [ 34 | • ] ────▶ [ 63 | null ]
6  [ null ]
7  [ • ] ────▶ [ 12 | null ]
8  [ null ]
9  [ null ]
```
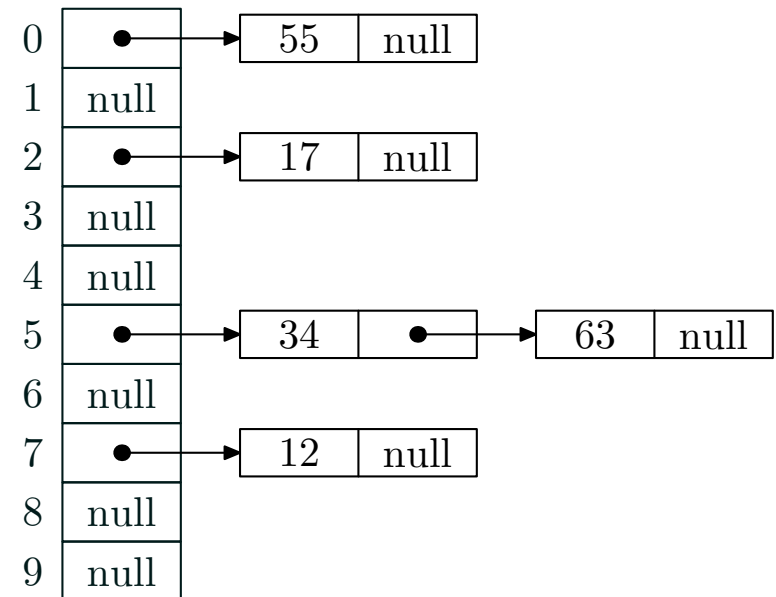
55, 17, 34, 63, 12

Iterating Over a Hash Table

- To iterate over a hash table we

  ★ Iterate through the array
  ★ At each element we iterate through the linked list

- The order of the elements appears random

- This becomes more efficient as the table becomes fuller

55, 17, 34, 63, 12

Outline

1. Why Hash?

2. Separate Chaining

3. **Open Addressing**

   - Quadratic Probing
   - Double Hashing

4. Hash Set and Map

Open Addressing

- In open addressing we have a single table of objects (without a linked-list)

- In the case of a collision a new location in the table is found

- The simplest mechanism is known as **linear probing** where we move the entry to the next available location

Open Addressing

- In open addressing we have a single table of objects (without a linked-list)

- In the case of a collision a new location in the table is found

- The simplest mechanism is known as **linear probing** where we move the entry to the next available location
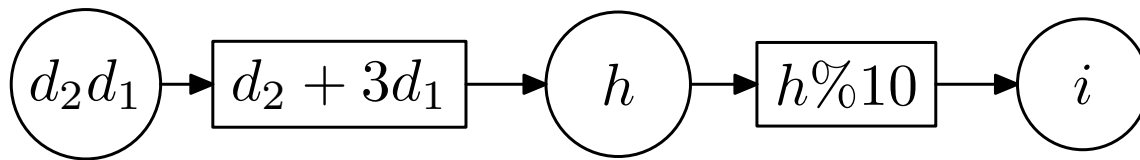
Open Addressing

- In open addressing we have a single table of objects (without a linked-list)

- In the case of a collision a new location in the table is found

- The simplest mechanism is known as **linear probing** where we move the entry to the next available location

Linear Probing

$$d_2 d_1 \rightarrow \boxed{d_2 + 3d_1} \rightarrow h \rightarrow \boxed{h\%10} \rightarrow i$$

$$\text{loading factor, } \lambda = \frac{0}{10}$$

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

Linear Probing

add(34)

loading factor, $\lambda = \dfrac{1}{10}$

Linear Probing

add(12)

$12 \rightarrow \boxed{d_2 + 3d_1} \rightarrow 7 \rightarrow \boxed{h\%10} \rightarrow 7$

loading factor, $\lambda = \dfrac{2}{10}$

0	null
1	null
2	null
3	null
4	null
5	$\bullet \rightarrow$ 34
6	null
7	$\bullet \rightarrow$ 12
8	null
9	null

Linear Probing

$\text{add}(55)$

$55 \rightarrow \boxed{d_2 + 3d_1} \rightarrow 20 \rightarrow \boxed{h\%10} \rightarrow 0$

loading factor, $\lambda = \dfrac{3}{10}$

0	• $\rightarrow$ 55
1	null
2	null
3	null
4	null
5	• $\rightarrow$ 34
6	null
7	• $\rightarrow$ 12
8	null
9	null

Linear Probing

add(17)

$$d_2 + 3d_1$$

$$h\%10$$

17 → $d_2 + 3d_1$ → 22 → $h\%10$ → 2

loading factor, $\lambda = \dfrac{4}{10}$

0	● → 55
1	null
2	● → 17
3	null
4	null
5	● → 34
6	null
7	● → 12
8	null
9	null

Linear Probing

add(63)

$$63 \rightarrow \boxed{d_2 + 3d_1} \rightarrow 15 \rightarrow \boxed{h\%10} \rightarrow 5$$

loading factor, $\lambda = \dfrac{5}{10}$

0	●	→ 55
1	null	
2	●	→ 17
3	null	
4	null	
5	●	→ 34
6	●	→ 63
7	●	→ 12
8	null	
9	null	

Linear Probing

$$\text{add}(73)$$

$73 \rightarrow \boxed{d_2 + 3d_1} \rightarrow 16 \rightarrow \boxed{h\%10} \rightarrow 6$

$$\text{loading factor, } \lambda = \frac{6}{10}$$

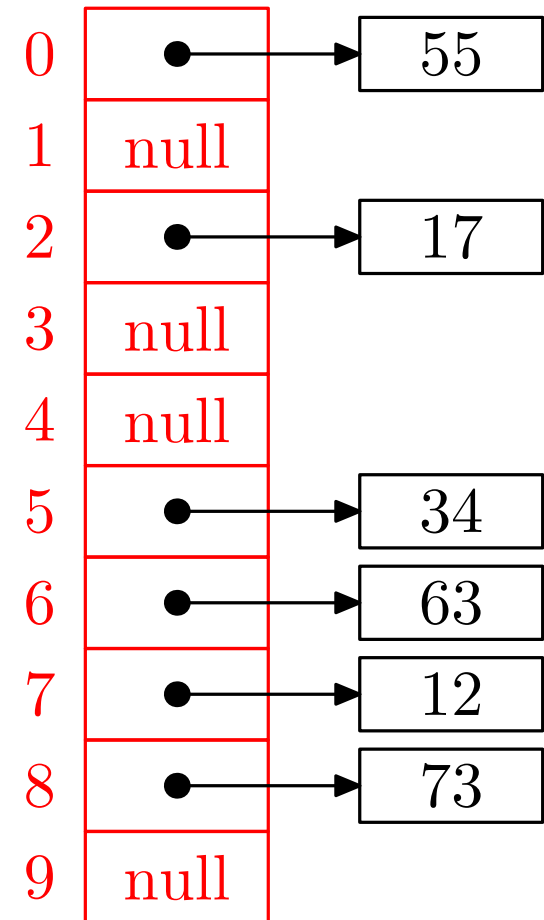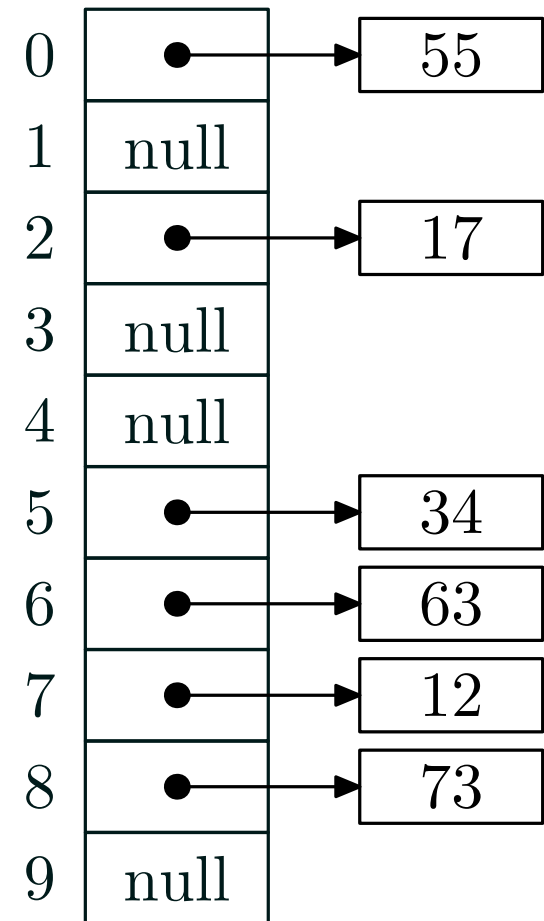|	
0	$\bullet \rightarrow 55$
1	null
2	$\bullet \rightarrow 17$
3	null
4	null
5	$\bullet \rightarrow 34$
6	$\bullet \rightarrow 63$
7	$\bullet \rightarrow 12$
8	$\bullet \rightarrow 73$
9	null

Linear Probing Pile Up

- The entries will tend to pile up or cluster—this is sometimes referred to as **primary clustering**

- Clusters become worse as the number of entries grow

- Clusters will increase the number of probes needed to find an insert location

- The proportion of full entries in the table is known as the **loading factor**

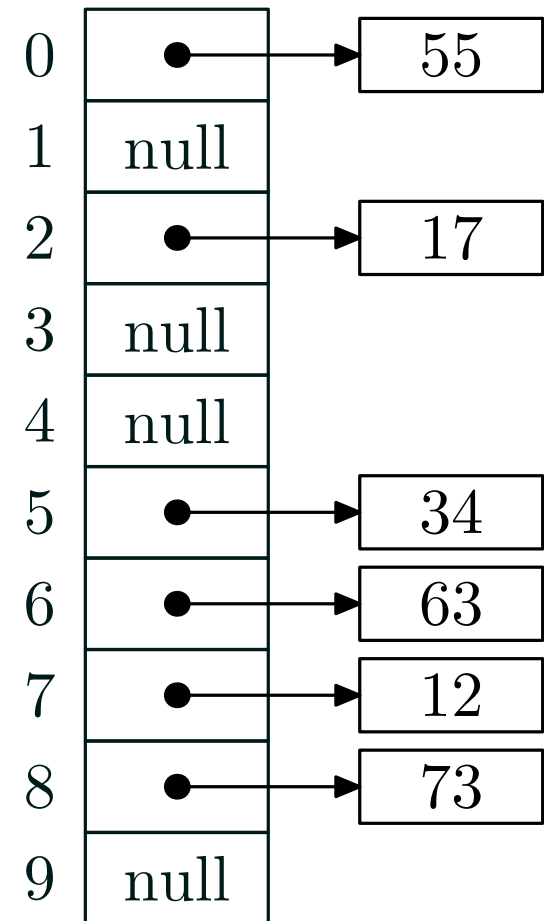|		
0	● →	55
1	null	
2	● →	17
3	null	
4	null	
5	● →	34
6	● →	63
7	● →	12
8	● →	73
9	null	

Linear Probing Pile Up

- The entries will tend to pile up or cluster—this is sometimes referred to as **primary clustering**

- Clusters become worse as the number of entries grow

- Clusters will increase the number of probes needed to find an insert location

- The proportion of full entries in the table is known as the **loading factor**

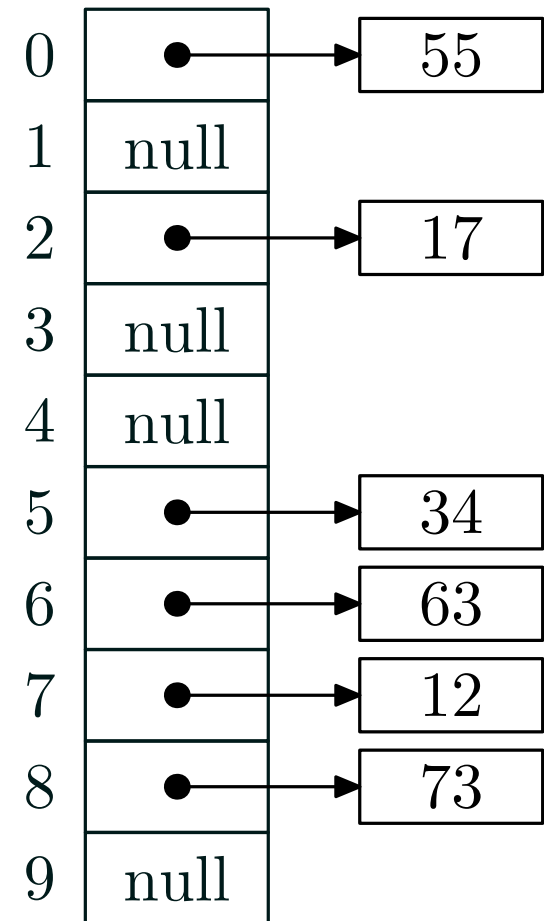|	
0	● → 55
1	null
2	● → 17
3	null
4	null
5	● → 34
6	● → 63
7	● → 12
8	● → 73
9	null

Linear Probing Pile Up

- The entries will tend to pile up or cluster—this is sometimes referred to as **primary clustering**

- Clusters become worse as the number of entries grow

- Clusters will increase the number of probes needed to find an insert location

- The proportion of full entries in the table is known as the **loading factor**

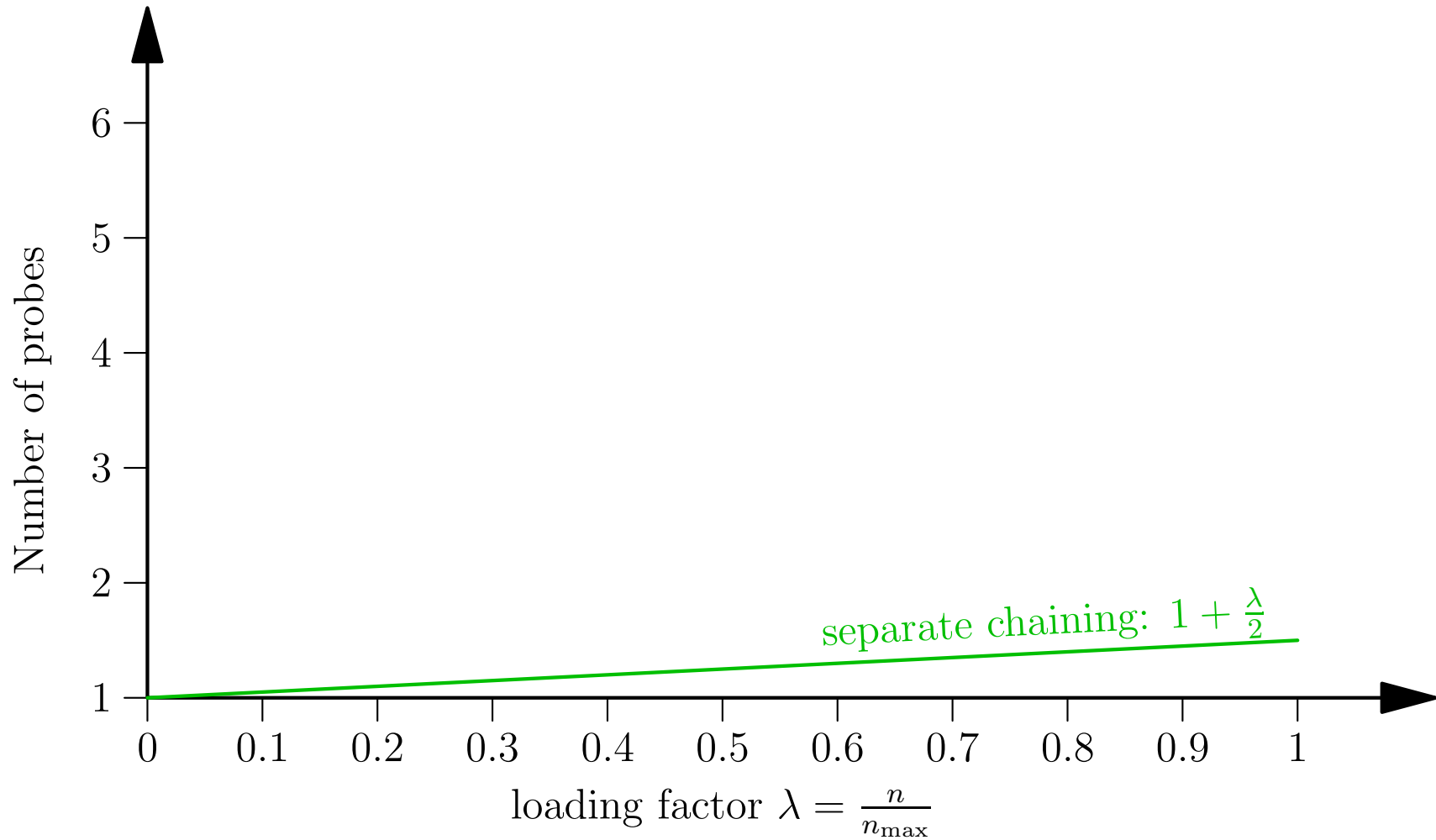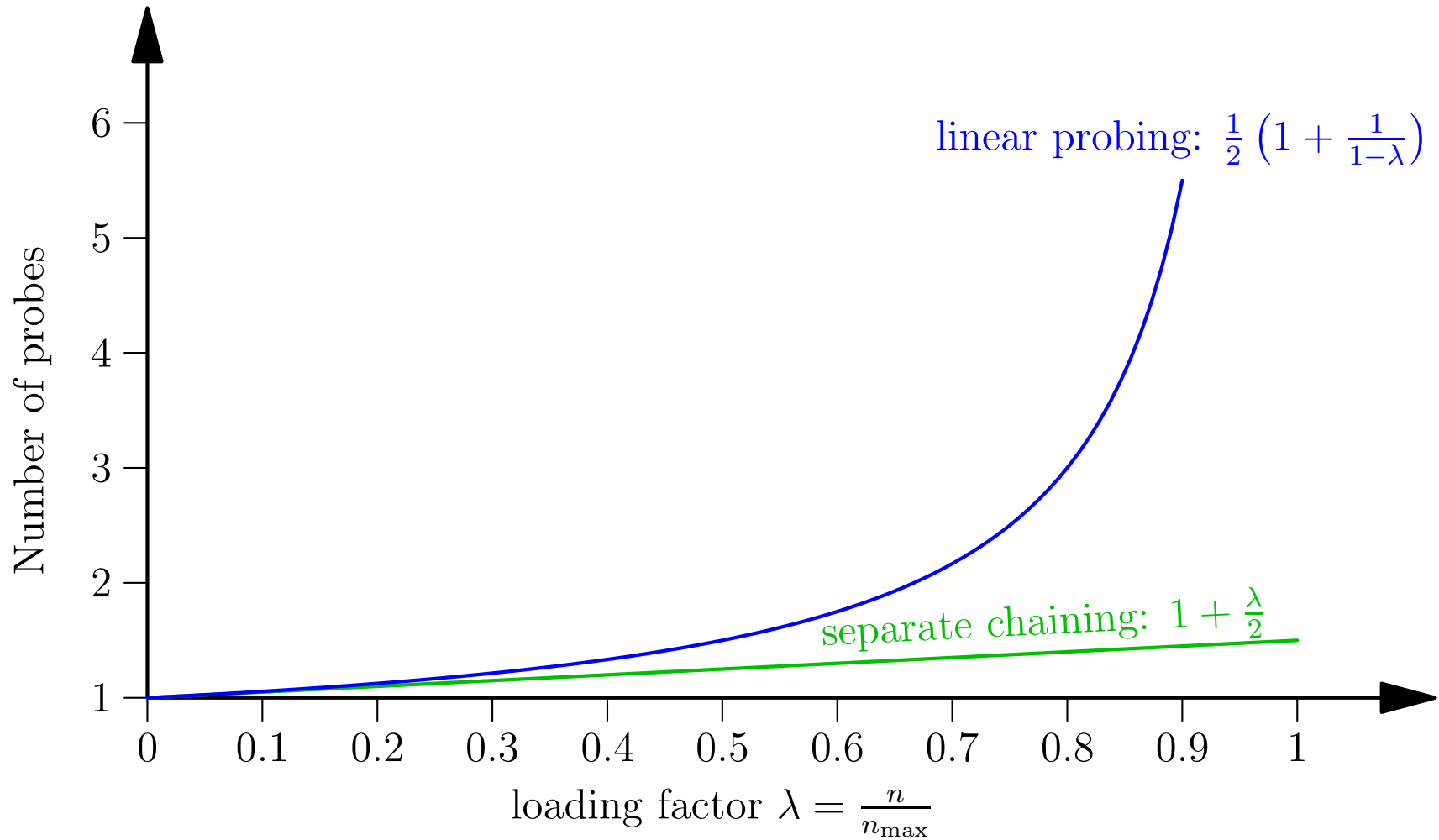|		
0	●	→ 55
1	null	
2	●	→ 17
3	null	
4	null	
5	●	→ 34
6	●	→ 63
7	●	→ 12
8	●	→ 73
9	null	

Linear Probing Pile Up

- The entries will tend to pile up or cluster—this is sometimes referred to as **primary clustering**

- Clusters become worse as the number of entries grow

- Clusters will increase the number of probes needed to find an insert location

- The proportion of full entries in the table is known as the **loading factor**

0	● → 55
1	null
2	● → 17
3	null
4	null
5	● → 34
6	● → 63
7	● → 12
8	● → 73
9	null

Reducing Number of Probes

separate chaining: $1 + \frac{\lambda}{2}$

Number of probes

loading factor $\lambda = \frac{n}{n_{\max}}$

Reducing Number of Probes

Number of probes vs. loading factor $\lambda = \frac{n}{n_{\max}}$

linear probing: $\frac{1}{2}\left(1 + \frac{1}{1-\lambda}\right)$

separate chaining: $1 + \frac{\lambda}{2}$

Reducing Number of Probes

Number of probes vs *loading factor $\lambda = \frac{n}{n_{\max}}$*

linear probing: $\frac{1}{2}\left(1 + \frac{1}{1-\lambda}\right)$

quadratic probing/double hashing: $-\frac{\ln(1-\alpha)}{\alpha}$

separate chaining: $1 + \frac{\lambda}{2}$
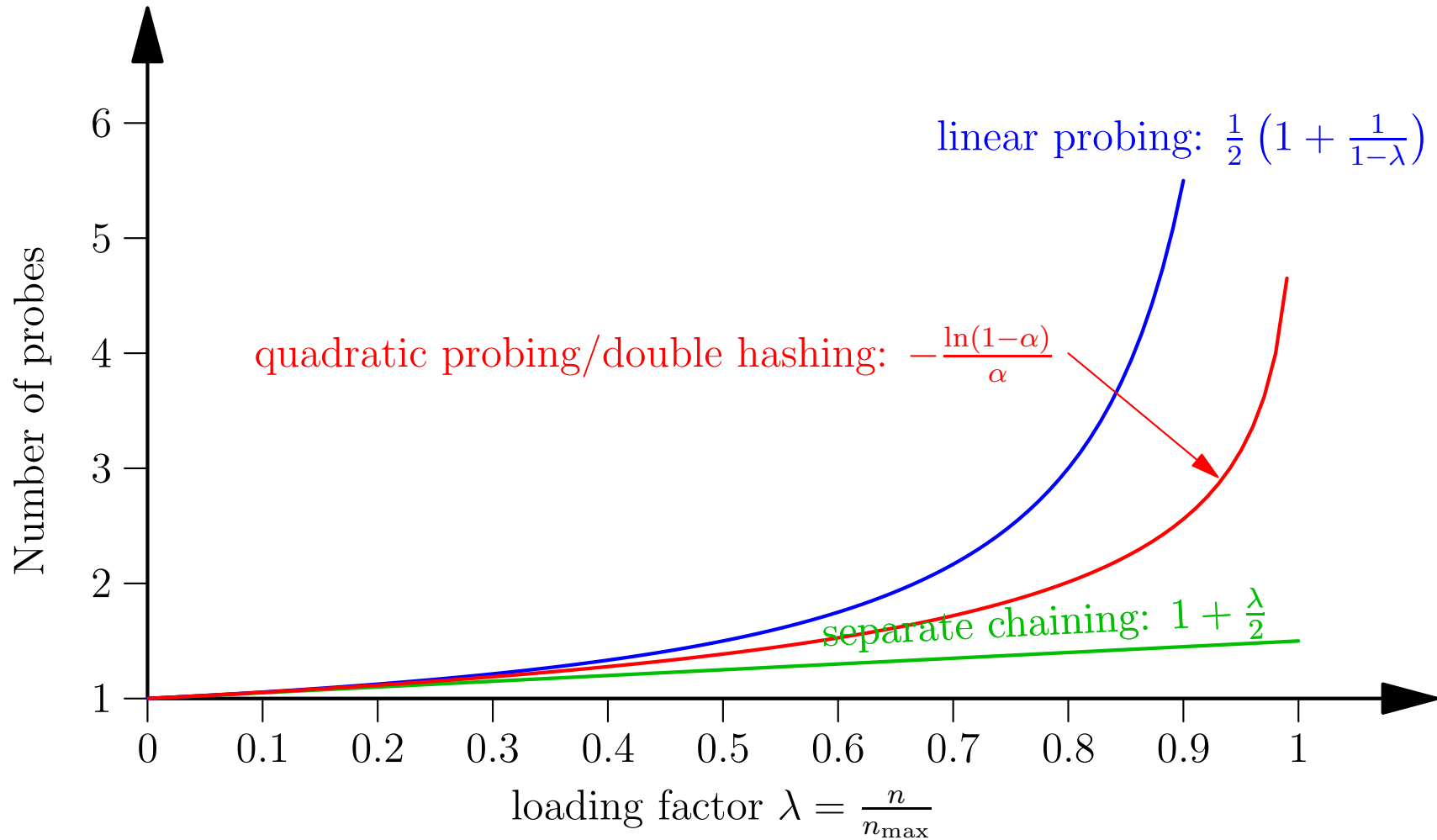
Reducing Number of Probes

- To avoid clustering we can use **quadratic probing**

Reducing Number of Probes

To avoid clustering we can use **quadratic probing** or **double hashing**

Quadratic Probing

- In quadratic probing we try the locations $h(x) + d_i$ where $h(x)$ is the original hash code and $d_i = i^2$

- That is we takes steps 1, 4, 9, 16, . . .

- Quadratic probing prevents primary clustering so dramatically decreases the number of probes needed to find a free location when the table is reasonably full

- One problem is that if we are unlucky we might not be able to add an element to the hash table even if the table isn't full

- However, if the size of the table is prime then quadratic probing will always find a free position provided it is not more than half full

Quadratic Probing

- In quadratic probing we try the locations $h(x) + d_i$ where $h(x)$ is the original hash code and $d_i = i^2$

- That is we takes steps 1, 4, 9, 16, . . .

- Quadratic probing prevents primary clustering so dramatically decreases the number of probes needed to find a free location when the table is reasonably full

- One problem is that if we are unlucky we might not be able to add an element to the hash table even if the table isn't full

- However, if the size of the table is prime then quadratic probing will always find a free position provided it is not more than half full

Quadratic Probing

- In quadratic probing we try the locations $h(x) + d_i$ where $h(x)$ is the original hash code and $d_i = i^2$

- That is we takes steps 1, 4, 9, 16, . . .

- Quadratic probing prevents primary clustering so dramatically decreases the number of probes needed to find a free location when the table is reasonably full

- One problem is that if we are unlucky we might not be able to add an element to the hash table even if the table isn't full

- However, if the size of the table is prime then quadratic probing will always find a free position provided it is not more than half full

Quadratic Probing

- In quadratic probing we try the locations $h(x) + d_i$ where $h(x)$ is the original hash code and $d_i = i^2$

- That is we takes steps 1, 4, 9, 16, . . .

- Quadratic probing prevents primary clustering so dramatically decreases the number of probes needed to find a free location when the table is reasonably full

- One problem is that if we are unlucky we might not be able to add an element to the hash table even if the table isn't full

- However, if the size of the table is prime then quadratic probing will always find a free position provided it is not more than half full

Quadratic Probing

- In quadratic probing we try the locations $h(x) + d_i$ where $h(x)$ is the original hash code and $d_i = i^2$

- That is we takes steps 1, 4, 9, 16, . . .

- Quadratic probing prevents primary clustering so dramatically decreases the number of probes needed to find a free location when the table is reasonably full

- One problem is that if we are unlucky we might not be able to add an element to the hash table even if the table isn't full

- However, if the size of the table is prime then quadratic probing will always find a free position provided it is not more than half full

---

Double Hashing

- An alternative strategy is to known as double hashing where the locations tried are $h(x) + d_i$ where $d_i = i \times h_2(x)$

- $h_2(x)$ is a second hash function that depends on the key

- A good choice is $h_2(x) = R - (x \mod R)$ where R is a prime smaller than the table size

- It is important that $h_2(x)$ is not a divisor of the table size

  - Either make sure the table size is prime or
  - Set the step size to 1 if $h_2(x)$ is a divisor of the table size

Double Hashing

- An alternative strategy is to known as double hashing where the locations tried are $h(x) + d_i$ where $d_i = i \times h_2(x)$

- $h_2(x)$ is a second hash function that depends on the key

- A good choice is $h_2(x) = R - (x \mod R)$ where R is a prime smaller than the table size

- It is important that $h_2(x)$ is not a divisor of the table size

  - ⋆ Either make sure the table size is prime or
  - ⋆ Set the step size to 1 if $h_2(x)$ is a divisor of the table size

Double Hashing

- An alternative strategy is to known as double hashing where the locations tried are $h(x) + d_i$ where $d_i = i \times h_2(x)$

- $h_2(x)$ is a second hash function that depends on the key

- A good choice is $h_2(x) = R - (x \mod R)$ where R is a prime smaller than the table size

- It is important that $h_2(x)$ is not a divisor of the table size

  - ⋆ Either make sure the table size is prime or
  - ⋆ Set the step size to 1 if $h_2(x)$ is a divisor of the table size

Double Hashing

- An alternative strategy is to known as double hashing where the locations tried are $h(x) + d_i$ where $d_i = i \times h_2(x)$

- $h_2(x)$ is a second hash function that depends on the key

- A good choice is $h_2(x) = R - (x \mod R)$ where R is a prime smaller than the table size

- It is important that $h_2(x)$ is not a divisor of the table size

  - ⋆ Either make sure the table size is prime or
  - ⋆ Set the step size to 1 if $h_2(x)$ is a divisor of the table size

Double Hashing

- An alternative strategy is to known as double hashing where the locations tried are $h(x) + d_i$ where $d_i = i \times h_2(x)$

- $h_2(x)$ is a second hash function that depends on the key

- A good choice is $h_2(x) = R - (x \mod R)$ where R is a prime smaller than the table size

- It is important that $h_2(x)$ is not a divisor of the table size

  - ⋆ Either make sure the table size is prime or
  - ⋆ Set the step size to 1 if $h_2(x)$ is a divisor of the table size

Double Hashing

- An alternative strategy is to known as double hashing where the locations tried are $h(x) + d_i$ where $d_i = i \times h_2(x)$

- $h_2(x)$ is a second hash function that depends on the key

- A good choice is $h_2(x) = R - (x \mod R)$ where R is a prime smaller than the table size

- It is important that $h_2(x)$ is not a divisor of the table size

    - Either make sure the table size is prime or
    - Set the step size to 1 if $h_2(x)$ is a divisor of the table size

Problems with Remove

- For all open addressing hash systems removing an entry is a problem

- Remember our strategy to find an input x is

  1. Compute the array index based on the hash code of x
  2. If the array location is empty then the search fails
  3. If the array location contains the key the search succeeds
  4. otherwise find a new location using an open addressing strategy and go to 2

- If we remove an entry then find might reach an empty location which was previously full

- This can prevent us finding a true entry

Problems with Remove

- For all open addressing hash systems removing an entry is a problem

- Remember our strategy to find an input x is

  1. Compute the array index based on the hash code of x
  2. If the array location is empty then the search fails
  3. If the array location contains the key the search succeeds
  4. otherwise find a new location using an open addressing strategy and go to 2

- If we remove an entry then find might reach an empty location which was previously full

- This can prevent us finding a true entry

Problems with Remove

- For all open addressing hash systems removing an entry is a problem

- Remember our strategy to find an input x is

  1. Compute the array index based on the hash code of x
  2. If the array location is empty then the search fails
  3. If the array location contains the key the search succeeds
  4. otherwise find a new location using an open addressing strategy and go to 2

- If we remove an entry then find might reach an empty location which was previously full

- This can prevent us finding a true entry

Problems with Remove

- For all open addressing hash systems removing an entry is a problem

- Remember our strategy to find an input x is

  1. Compute the array index based on the hash code of x
  2. If the array location is empty then the search fails
  3. If the array location contains the key the search succeeds
  4. otherwise find a new location using an open addressing strategy and go to 2

- If we remove an entry then find might reach an empty location which was previously full

- This can prevent us finding a true entry

Problems with Remove

- For all open addressing hash systems removing an entry is a problem

- Remember our strategy to find an input x is

  1. Compute the array index based on the hash code of x
  2. If the array location is empty then the search fails
  3. If the array location contains the key the search succeeds
  4. otherwise find a new location using an open addressing strategy and go to 2

- If we remove an entry then find might reach an empty location which was previously full

- This can prevent us finding a true entry

Problems with Remove

- For all open addressing hash systems removing an entry is a problem

- Remember our strategy to find an input x is

  1. Compute the array index based on the hash code of x
  2. If the array location is empty then the search fails
  3. If the array location contains the key the search succeeds
  4. otherwise find a new location using an open addressing strategy and go to 2

- If we remove an entry then find might reach an empty location which was previously full

- This can prevent us finding a true entry

Problems with Remove

- For all open addressing hash systems removing an entry is a problem

- Remember our strategy to find an input x is

  1. Compute the array index based on the hash code of x
  2. If the array location is empty then the search fails
  3. If the array location contains the key the search succeeds
  4. otherwise find a new location using an open addressing strategy and go to 2

- If we remove an entry then find might reach an empty location which was previously full

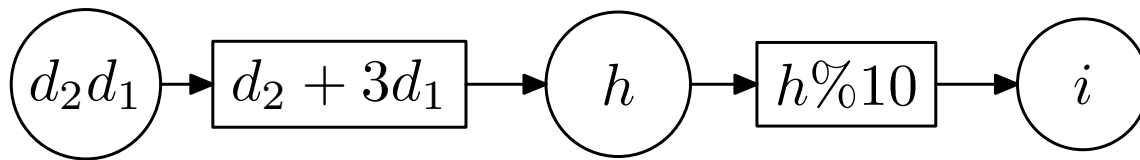- This can prevent us finding a true entry

---

Problems with Remove

- For all open addressing hash systems removing an entry is a problem

- Remember our strategy to find an input x is

  1. Compute the array index based on the hash code of x
  2. If the array location is empty then the search fails
  3. If the array location contains the key the search succeeds
  4. otherwise find a new location using an open addressing strategy and go to 2

- If we remove an entry then find might reach an empty location which was previously full

- This can prevent us finding a true entry

Linear Probing Example

Algorithms and Analysis

Linear Probing Example

$$\text{find}(12) \longrightarrow \text{true}$$

$$12 \longrightarrow \boxed{d_2 + 3d_1} \longrightarrow 7 \longrightarrow \boxed{h\%10} \longrightarrow 7$$

0	●	→	55
1	null		
2	●	→	17
3	null		
4	null		
5	●	→	34
6	●	→	63
7	●	→	12
8	●	→	73
9	null		

Linear Probing Example

find(44) $\longrightarrow$ fail

$44 \rightarrow \boxed{d_2 + 3d_1} \rightarrow 16 \rightarrow \boxed{h\%10} \rightarrow 6$

0	$\bullet$	$\rightarrow$	55
1	null		
2	$\bullet$	$\rightarrow$	17
3	null		
4	null		
5	$\bullet$	$\rightarrow$	34
6	$\bullet$	$\rightarrow$	63
7	$\bullet$	$\rightarrow$	12
8	$\bullet$	$\rightarrow$	73
9	null		

Linear Probing Example

$\mathrm{find}(73) \longrightarrow \mathrm{true}$

$73 \longrightarrow \boxed{d_2 + 3d_1} \longrightarrow 16 \longrightarrow \boxed{h\%10} \longrightarrow 6$

0	●	→ 55
1	null	
2	●	→ 17
3	null	
4	null	
5	●	→ 34
6	●	→ 63
7	●	→ 12
8	●	→ 73
9	null	

Linear Probing Example

$\text{delete}(12) \longrightarrow \text{true}$

$12 \longrightarrow \boxed{d_2 + 3d_1} \longrightarrow 7 \longrightarrow \boxed{h\%10} \longrightarrow 7$

0	•	→ 55
1	null	
2	•	→ 17
3	null	
4	null	
5	•	→ 34
6	•	→ 63
7	null	
8	•	→ 73
9	null	

Linear Probing Example

$\text{find}(12) \longrightarrow \text{fail}$

$12 \rightarrow \boxed{d_2 + 3d_1} \rightarrow 7 \rightarrow \boxed{h\%10} \rightarrow 7$

0	• → 55
1	null
2	• → 17
3	null
4	null
5	• → 34
6	• → 63
7	null
8	• → 73
9	null

Linear Probing Example

Linear Probing Example

$$\text{find}(73) \longrightarrow \text{fail}$$

$$73 \rightarrow \boxed{d_2 + 3d_1} \rightarrow 16 \rightarrow \boxed{h\%10} \rightarrow 6$$

0	● →	55
1	null	
2	● →	17
3	null	
4	null	
5	● →	34
6	● →	63
7	null	
8	● →	73
9	null	

Lazy Remove

- One easy fix is to mark the deleted table with a special entry

- A find method would consider this entry as full

- An iterator would ignore this entry

- An insert operator could insert a new entry in these special locations

Lazy Remove

- One easy fix is to mark the deleted table with a special entry

- A find method would consider this entry as full

- An iterator would ignore this entry

- An insert operator could insert a new entry in these special locations

Lazy Remove

- One easy fix is to mark the deleted table with a special entry

- A find method would consider this entry as full

- An iterator would ignore this entry

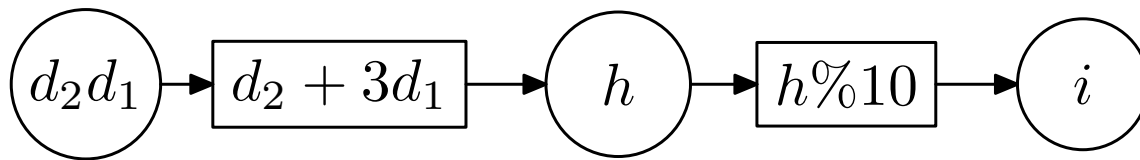- An insert operator could insert a new entry in these special locations

Lazy Remove

- One easy fix is to mark the deleted table with a special entry

- A find method would consider this entry as full

- An iterator would ignore this entry

- An insert operator could insert a new entry in these special locations

Lazy Remove in Action

$$\left(d_2 d_1\right) \rightarrow \boxed{d_2 + 3d_1} \rightarrow \left(h\right) \rightarrow \boxed{h\%10} \rightarrow \left(i\right)$$

0	● →	55
1	null	
2	● →	17
3	null	
4	null	
5	● →	34
6	● →	63
7	● →	12
8	● →	73
9	null	

Lazy Remove in Action

$\text{find}(12) \longrightarrow \text{true}$

$12 \rightarrow \boxed{d_2 + 3d_1} \rightarrow 7 \rightarrow \boxed{h\%10} \rightarrow 7$

0	●	55
1	null	
2	●	17
3	null	
4	null	
5	●	34
6	●	63
7	●	12
8	●	73
9	null	

Lazy Remove in Action

$$\text{find}(73) \longrightarrow \text{true}$$

$$73 \rightarrow \boxed{d_2 + 3d_1} \rightarrow 16 \rightarrow \boxed{h\%10} \rightarrow 6$$

0	• → 55
1	null
2	• → 17
3	null
4	null
5	• → 34
6	• → 63
7	• → 12
8	• → 73
9	null

Lazy Remove in Action

$$\text{delete}(12) \longrightarrow \text{true}$$

$$\boxed{12} \rightarrow \boxed{d_2 + 3d_1} \rightarrow \boxed{7} \rightarrow \boxed{h\%10} \rightarrow \boxed{7}$$

0	•	55
1	null	
2	•	17
3	null	
4	null	
5	•	34
6	•	63
7	•	dummy
8	•	73
9	null	

Lazy Remove in Action

$\text{find}(12) \longrightarrow \text{fail}$

Lazy Remove in Action

$$\text{find}(73) \longrightarrow \text{true}$$

$73 \rightarrow \boxed{d_2 + 3d_1} \rightarrow 16 \rightarrow \boxed{h\%10} \rightarrow 6$

0	• → 55
1	null
2	• → 17
3	null
4	null
5	• → 34
6	• → 63
7	• → dummy
8	• → 73
9	null

Outline

1. Why Hash?

2. Separate Chaining

3. Open Addressing

   - Quadratic Probing
   - Double Hashing

4. **Hash Set and Map**

What Strategy to Use?

- Most libraries including the STL (and the Java Collection class) use separate chaining

- This has the advantage that its performance does not degrade badly as the number of entries increase

- This reduces the need to resize the hash table

- The C++ standard did not include a hash table until C++11 ☹—although very good hash tables existed in C++

What Strategy to Use?

- Most libraries including the STL (and the Java Collection class) use separate chaining

- This has the advantage that its performance does not degrade badly as the number of entries increase

- This reduces the need to resize the hash table

- The C++ standard did not include a hash table until C++11 ☹—although very good hash tables existed in C++

What Strategy to Use?

- Most libraries including the STL (and the Java Collection class) use separate chaining

- This has the advantage that its performance does not degrade badly as the number of entries increase

- This reduces the need to resize the hash table

- The C++ standard did not include a hash table until C++11 ☹—although very good hash tables existed in C++

What Strategy to Use?

- Most libraries including the STL (and the Java Collection class) use separate chaining

- This has the advantage that its performance does not degrade badly as the number of entries increase

- This reduces the need to resize the hash table

- The C++ standard did not include a hash table until C++11 ☹—although very good hash tables existed in C++

What Strategy to Use?

- Most libraries including the STL (and the Java Collection class) use separate chaining

- This has the advantage that its performance does not degrade badly as the number of entries increase

- This reduces the need to resize the hash table

- The C++ standard did not include a hash table until C++11 ☹—although very good hash tables existed in C++

Hash Sets and Maps

- C++ also provides an `unordered_map<Key, V>` class

- It's performance is asymptotically superior to `map`, $O(1)$ rather than $O(\log(n))$

- Hash functions can take time to compute (it is often $O(\log(n))$) so `unordered_set`s might not be faster than `set`s

- One major difference is that the iterator for `set`s return the elements in order, `undordered_set`'s iterator doesn't

Hash Sets and Maps

- C++ also provides an `unordered_map<Key, V>` class

- It's performance is asymptotically superior to `map`, $O(1)$ rather than $O(\log(n))$

- Hash functions can take time to compute (it is often $O(\log(n))$) so `unordered_set`s might not be faster than `set`s

- One major difference is that the iterator for `set`s return the elements in order, `undordered_set`'s iterator doesn't

Hash Sets and Maps

- C++ also provides an `unordered_map<Key, V>` class

- It's performance is asymptotically superior to `map`, $O(1)$ rather than $O(\log(n))$

- Hash functions can take time to compute (it is often $O(\log(n))$) so `unordered_set`s might not be faster than `set`s

- One major difference is that the iterator for `set`s return the elements in order, `unordered_set`'s iterator doesn't

Applications

- Hash tables are used everywhere

- E.g. most databases use hash tables to speed up search

- In many document applications hash tables will be being generated in the background

- Content addressability is ubiquitous to many application where hash tables are used as standard

Applications

- Hash tables are used everywhere

- E.g. most databases use hash tables to speed up search

- In many document applications hash tables will be being generated in the background

- Content addressability is ubiquitous to many application where hash tables are used as standard

Applications

- Hash tables are used everywhere

- E.g. most databases use hash tables to speed up search

- In many document applications hash tables will be being generated in the background

- Content addressability is ubiquitous to many application where hash tables are used as standard

---

Applications

- Hash tables are used everywhere

- E.g. most databases use hash tables to speed up search

- In many document applications hash tables will be being generated in the background

- Content addressability is ubiquitous to many application where hash tables are used as standard

Lessons

- Hash tables are one of the most useful tools you have available

- They aren't particularly difficult to understand, but you need to know about

  - ⋆ hashing functions
  - ⋆ collision strategies
  - ⋆ performance (i.e. when they work)

Lessons

• Hash tables are one of the most useful tools you have available

• They aren't particularly difficult to understand, but you need to know about

  ⋆ hashing functions
  ⋆ collision strategies
  ⋆ performance (i.e. when they work)

Lessons

- Hash tables are one of the most useful tools you have available

- They aren't particularly difficult to understand, but you need to know about

  ⋆ hashing functions
  ⋆ collision strategies
  ⋆ performance (i.e. when they work)

Lessons

- Hash tables are one of the most useful tools you have available

- They aren't particularly difficult to understand, but you need to know about

  - ⋆ hashing functions
  - ⋆ collision strategies
  - ⋆ performance (i.e. when they work)

Lessons

- Hash tables are one of the most useful tools you have available

- They aren't particularly difficult to understand, but you need to know about

  ⋆ hashing functions
  ⋆ collision strategies
  ⋆ performance (i.e. when they work)