Hash Tables

CS 2860: Algorithms and Complexity

Magnus Wahlström and Gregory Gutin

February 13, 2019

Implementation: Basic idea

- A hash table has both a size (number of contained elements) and a capacity ("space" reserved for further elements)
- A hash table with capacity m contains an array (the table) with m slots, called hash buckets
- The data is distributed across the table, so that:
  1. For every item x, there is one specific slot where it "should" be placed (quickly computable), depending on hash(x)
  2. Almost all of the n different items are placed in different slots
  3. For the case where two items land in the same bucket, need collision handling (for example, external list for each bucket)
- If done perfectly, this would imply constant-time insert, delete, lookup operations

Implementation: Basic idea

- A hash table has both a size (number of contained elements) and a capacity ("space" reserved for further elements)
- A hash table with capacity m contains an array (the table) with m slots, called hash buckets
- The data is distributed across the table, so that:
  1. For every item x, there is one specific slot where it "should" be placed (quickly computable), depending on hash(x)
  2. Almost all of the n different items are placed in different slots
  3. For the case where two items land in the same bucket, need collision handling (for example, external list for each bucket)
- If done perfectly, this would imply constant-time insert, delete, lookup operations

Implementation: Basic idea

- A hash table has both a size (number of contained elements) and a capacity ("space" reserved for further elements)
- A hash table with capacity m contains an array (the table) with m slots, called hash buckets
- The data is distributed across the table, so that:
  1. For every item x, there is one specific slot where it "should" be placed (quickly computable), depending on hash(x)
  2. Almost all of the n different items are placed in different slots
  3. For the case where two items land in the same bucket, need collision handling (for example, external list for each bucket)
- If done perfectly, this would imply constant-time insert, delete, lookup operations

Implementation: Basic idea

- A hash table has both a size (number of contained elements) and a capacity ("space" reserved for further elements)
- A hash table with capacity m contains an array (the table) with m slots, called hash buckets
- The data is distributed across the table, so that:
  1. For every item x, there is one specific slot where it "should" be placed (quickly computable), depending on hash(x)
  2. Almost all of the n different items are placed in different slots
  3. For the case where two items land in the same bucket, need collision handling (for example, external list for each bucket)
- If done perfectly, this would imply constant-time insert, delete, lookup operations

First example: Integer keys

- Have: Array with m slots (say $m = 7$)
- Want: Map any integer (arbitrary 32-bit number) into $\{0, 1, 2, \ldots, 6\}$
- Use modulo operation (Java: %):
  - $0 \,\%\, 7 = 0$
  - $1 \,\%\, 7 = 1$
  - ...
  - $6 \,\%\, 7 = 6$
  - $7 \,\%\, 7 = 0$
  - $8 \,\%\, 7 = 1$
  - ...
- (Like the hands of a clock; minutes are counted mod 60)

First example: Integer keys

- Have: Array with m slots (say $m = 7$)
- Want: Map any integer (arbitrary 32-bit number) into $\{0, 1, 2, \ldots, 6\}$
- Use modulo operation (Java: %):
  - $0 \,\%\, 7 = 0$
  - $1 \,\%\, 7 = 1$
  - $\ldots$
  - $6 \,\%\, 7 = 6$
  - $7 \,\%\, 7 = 0$
  - $8 \,\%\, 7 = 1$
  - $\ldots$
- (Like the hands of a clock; minutes are counted mod 60)

Modulo: Consequences

Quick observations:

- For consecutive data (1000, 1001, 1002...), always get different slots
- For random data, usually get different slots
- Remains: Weird or especially crafted data (more later)
- Also: index (e.g, $1001 \% 7$) is cheap to compute (single CPU instruction)

Modulo: Consequences

Quick observations:

- For consecutive data (1000, 1001, 1002...), always get different slots
- For random data, usually get different slots
- Remains: Weird or especially crafted data (more later)
- Also: index (e.g, 1001 % 7) is cheap to compute (single CPU instruction)

Modulo: Consequences

Quick observations:

- For consecutive data (1000, 1001, 1002...), always get different slots
- For random data, usually get different slots
- Remains: Weird or especially crafted data (more later)
- Also: index (e.g, 1001 % 7) is cheap to compute (single CPU instruction)

Modulo: Consequences

Quick observations:

- For consecutive data (1000, 1001, 1002...), always get different slots
- For random data, usually get different slots
- Remains: Weird or especially crafted data (more later)
- Also: index (e.g, $1001 \% 7$) is cheap to compute (single CPU instruction)

Why should m be a prime number?

- If they set K of keys is uniformly distributed (i.e., every key in K is equally likely to occur), then the choice of m is not important.
- But, what happens if K is not uniformly distributed?
- If the keys occur in the multiples of 4 then all of the slots that are not multiples of 4 will be empty (which is really bad in terms of hash table performance).
- This can happen.

Why should m be a prime number?

- If they set K of keys is uniformly distributed (i.e., every key in K is equally likely to occur), then the choice of m is not important.
- But, what happens if K is not uniformly distributed?
- If the keys occur in the multiples of 4 then all of the slots that are not multiples of 4 will be empty (which is really bad in terms of hash table performance).
- This can happen.

Why should m be a prime number?

- If they set K of keys is uniformly distributed (i.e., every key in K is equally likely to occur), then the choice of m is not important.
- But, what happens if K is not uniformly distributed?
- If the keys occur in the multiples of 4 then all of the slots that are not multiples of 4 will be empty (which is really bad in terms of hash table performance).
- This can happen.

Why should m be a prime number?

- If they set K of keys is uniformly distributed (i.e., every key in K is equally likely to occur), then the choice of m is not important.
- But, what happens if K is not uniformly distributed?
- If the keys occur in the multiples of 4 then all of the slots that are not multiples of 4 will be empty (which is really bad in terms of hash table performance).
- This can happen.

General object data

- For other data (strings, lists, ...), need to "convert" object into integer via hash function
- Process: e.g., for insert(x):
  1. Compute h=hash(x)
  2. Find hash bucket h%m with modulo operation
  3. If it's empty, good; if not, need collision handling
- Lookup:
  1. Compute h=hash(x)
  2. Find hash bucket h%m
  3. If it contains x or is empty, we're happy (otherwise, see later)
- Since hash(x) behaves "as if random", keys will spread out into different slots
- Most operations will be efficient

General object data

- For other data (strings, lists, ...), need to "convert" object into integer via hash function
- Process: e.g., for insert(x):
  1. Compute h=hash(x)
  2. Find hash bucket h%m with modulo operation
  3. If it's empty, good; if not, need collision handling
- Lookup:
  1. Compute h=hash(x)
  2. Find hash bucket h%m
  3. If it contains x or is empty, we're happy
     (otherwise, see later)
- Since hash(x) behaves "as if random", keys will spread out into different slots
- Most operations will be efficient

General object data

- For other data (strings, lists, ...), need to "convert" object into integer via hash function
- Process: e.g., for insert(x):
  1. Compute h=hash(x)
  2. Find hash bucket h%m with modulo operation
  3. If it's empty, good; if not, need collision handling
- Lookup:
  1. Compute h=hash(x)
  2. Find hash bucket h%m
  3. If it contains x or is empty, we're happy (otherwise, see later)
- Since hash(x) behaves "as if random", keys will spread out into different slots
- Most operations will be efficient

Handling collisions

- There are two different principles for handling collisions:
  1. Chaining (closed addressing): Place colliding keys outside of main table
  2. Open addressing: Place colliding keys inside the table, in a new slot
- Chaining (used in Java): Each table slot is the start of a linked list of table entries
- If several items should go into slot (say) 3, they are put into such a list
- Insert/lookup/delete still fast if lists are short and rare

Handling collisions

- There are two different principles for handling collisions:
  1. Chaining (closed addressing): Place colliding keys outside of main table
  2. Open addressing: Place colliding keys inside the table, in a new slot
- Chaining (used in Java): Each table slot is the start of a linked list of table entries
- If several items should go into slot (say) 3, they are put into such a list
- Insert/lookup/delete still fast if lists are short and rare

Handling collisions

- There are two different principles for handling collisions:
  1. Chaining (closed addressing): Place colliding keys outside of main table
  2. Open addressing: Place colliding keys inside the table, in a new slot
- Chaining (used in Java): Each table slot is the start of a linked list of table entries
- If several items should go into slot (say) 3, they are put into such a list
- Insert/lookup/delete still fast if lists are short and rare

Chaining lookup procedure (Java)

- Will use two ubiquitous Java functions:
  - `object.hashCode()`: Computes a 32-bit hash of object
  - `object1.equals(object2)`: Checks whether object1 and object2 have "equivalent" contents
  - (Note: x.equals(y) and x == y are very different: x==y tests pointer equality
- Lookup procedure: Consider call `lookup(item)`
  1. Let `hash1 = item.hashCode()`
  2. Compute hash bucket `bucket = (hash1 % m)` (table capacity m)
  3. For each element `item2` stored in slot `bucket`:
     1. If `item2.hashCode() != hash1`: Continue to next object
     2. If `item2.hashCode() == hash1`, test `item2.equals(item)`
     3. If positive, return YES (found a copy of item in the hash table)
  4. If we have looked at all items in the bucket, the answer is NO

Chaining lookup procedure (Java)

- Will use two ubiquitous Java functions:
  - `object.hashCode()`: Computes a 32-bit hash of object
  - `object1.equals(object2)`: Checks whether object1 and object2 have "equivalent" contents
  - (Note: x.equals(y) and x == y are very different: x==y tests pointer equality
- Lookup procedure: Consider call `lookup(item)`
  1. Let `hash1 = item.hashCode()`
  2. Compute hash bucket `bucket = (hash1 % m)` (table capacity m)
  3. For each element `item2` stored in slot `bucket`:
     3.1 If `item2.hashCode() != hash1`: Continue to next object
     3.2 If `item2.hashCode() == hash1`, test `item2.equals(item)`
     3.3 If positive, return YES (found a copy of item in the hash table)
  4. If we have looked at all items in the bucket, the answer is NO

Insert, remove, get, put with chaining

- Insert(item) on chaining hash tables:
  1. Compute hash1=item.hashCode()
  2. Compute bucket=hash1 % m
  3. For every element item2 in slot bucket:
     3.1 If `item2.hashCode() != hash1`: Continue to next object
     3.2 If `item2.hashCode() == hash1`, test `item2.equals(item)`
     3.3 If positive, return (item already contained)
  4. If loop terminates, add item into bucket (e.g., to the front)
- Remove(item): very similarly
- For get, put (Map rather than Set):
  1. Every object x in hash table has x.key and x.value types
  2. E.g., put(key,value):
     2.1 Find or create slot x for object key exactly as above
     2.2 Let x.value==value

Insert, remove, get, put with chaining

- Insert(item) on chaining hash tables:
  1. Compute hash1=item.hashCode()
  2. Compute bucket=hash1 % m
  3. For every element item2 in slot bucket:
     3.1 If `item2.hashCode() != hash1`: Continue to next object
     3.2 If `item2.hashCode() == hash1`, test `item2.equals(item)`
     3.3 If positive, return (item already contained)
  4. If loop terminates, add item into bucket (e.g., to the front)
- Remove(item): very similarly
- For get, put (Map rather than Set):
  1. Every object x in hash table has x.key and x.value types
  2. E.g., put(key,value):
     2.1 Find or create slot x for object key exactly as above
     2.2 Let x.value==value

Insert, remove, get, put with chaining

- Insert(item) on chaining hash tables:
  1. Compute hash1=item.hashCode()
  2. Compute bucket=hash1 % m
  3. For every element item2 in slot bucket:
     3.1 If `item2.hashCode() != hash1`: Continue to next object
     3.2 If `item2.hashCode() == hash1`, test `item2.equals(item)`
     3.3 If positive, return (item already contained)
  4. If loop terminates, add item into bucket (e.g., to the front)
- Remove(item): very similarly
- For get, put (Map rather than Set):
  1. Every object x in hash table has x.key and x.value types
  2. E.g., put(key,value):
     2.1 Find or create slot x for object key exactly as above
     2.2 Let x.value==value

Alternative collisions handling

- Alternative to chaining: Keep data inside table
- Advantages:
  - Less memory usage
  - Better data locality (linked lists frequently slower)
- Open addressing: If our first slot is full, try another one...
  - Linear probing: After slot i, try slot $i+1$, $i+2$, ...
  - Double hashing: Have second hash function `hash2` which decides element-specific search order ($0 < \mathrm{hash2} < m$)
  - Try slots h1(x), (h1(x)+h2(x)), (h1(x)+2*h2(x)), ... (mod m)
- Linear probing can lead to "pileups", but double hashing is efficient (colliding objects go to different slots)

Alternative collisions handling

- Alternative to chaining: Keep data inside table
- Advantages:
  - Less memory usage
  - Better data locality (linked lists frequently slower)
- Open addressing: If our first slot is full, try another one...
  - Linear probing: After slot i, try slot $i+1$, $i+2$, ...
  - Double hashing: Have second hash function `hash2` which decides element-specific search order ($0 < \text{hash2} < m$)
  - Try slots h1(x), (h1(x)+h2(x)), (h1(x)+2*h2(x)), ... (mod m)
- Linear probing can lead to "pileups", but double hashing is efficient (colliding objects go to different slots)

Alternative collisions handling

- Alternative to chaining: Keep data inside table
- Advantages:
  - Less memory usage
  - Better data locality (linked lists frequently slower)
- Open addressing: If our first slot is full, try another one...
  - Linear probing: After slot i, try slot $i+1$, $i+2$, ...
  - Double hashing: Have second hash function `hash2` which decides element-specific search order ($0 < \text{hash2} < m$)
  - Try slots h1(x), (h1(x)+h2(x)), (h1(x)+2*h2(x)), ... (mod m)
- Linear probing can lead to "pileups", but double hashing is efficient (colliding objects go to different slots)

Alternative collisions handling

- Alternative to chaining: Keep data inside table
- Advantages:
  - Less memory usage
  - Better data locality (linked lists frequently slower)
- Open addressing: If our first slot is full, try another one...
  - Linear probing: After slot i, try slot $i+1$, $i+2$, ...
  - Double hashing: Have second hash function `hash2` which decides element-specific search order ($0 < \mathrm{hash2} < m$)
  - Try slots h1(x), (h1(x)+h2(x)), (h1(x)+2*h2(x)), ... (mod m)
- Linear probing can lead to "pileups", but double hashing is efficient (colliding objects go to different slots)

Hash tables worst/average case

Average case versus worst case

- A running theme in this course:
  - Worst case is easier to estimate (usually)
  - Worst case is a sure guarantee
  - Average case can be better than worst case (quicksort)
  - Average case makes some assumptions about use pattern (ex: "input is in random order"; "all hash keys act purely random")
  - If this assumption is reasonable, then we're happy using average case data
  - But is it reasonable?

Average case versus worst case

- A running theme in this course:
  - Worst case is easier to estimate (usually)
  - Worst case is a sure guarantee
  - Average case can be better than worst case (quicksort)
  - Average case makes some assumptions about use pattern (ex: "input is in random order"; "all hash keys act purely random")
  - If this assumption is reasonable, then we're happy using average case data
  - But is it reasonable?

Average case versus worst case

- A running theme in this course:
  - Worst case is easier to estimate (usually)
  - Worst case is a sure guarantee
  - Average case can be better than worst case (quicksort)
  - Average case makes some assumptions about use pattern (ex: "input is in random order"; "all hash keys act purely random")
  - If this assumption is reasonable, then we're happy using average case data
  - But is it reasonable?

Average case versus worst case

- A running theme in this course:
  - Worst case is easier to estimate (usually)
  - Worst case is a sure guarantee
  - Average case can be better than worst case (quicksort)
  - Average case makes some assumptions about use pattern (ex: "input is in random order"; "all hash keys act purely random")
  - If this assumption is reasonable, then we're happy using average case data
  - But is it reasonable?

Average case versus worst case

- A running theme in this course:
  - Worst case is easier to estimate (usually)
  - Worst case is a sure guarantee
  - Average case can be better than worst case (quicksort)
  - Average case makes some assumptions about use pattern (ex: "input is in random order"; "all hash keys act purely random")
  - If this assumption is reasonable, then we're happy using average case data
  - But is it reasonable?

Average/worst case differences

Consider the following cases! When do the worst cases arise?

1. Quicksort(array,low,high) with pivot array[low]
   - Average case $O(n \log n)$ for random input
   - Worst case $O(n^2)$ arises:
   - The worst case is (unlikely/possible):

Average/worst case differences

Consider the following cases! When do the worst cases arise?

1. Quicksort(array,low,high) with pivot array[low]
   - Average case $O(n \log n)$ for random input
   - Worst case $O(n^2)$ arises: When input is already sorted
   - The worst case is (unlikely/possible):

Average/worst case differences

Consider the following cases! When do the worst cases arise?

1. Quicksort(array,low,high) with pivot array[low]
   - Average case $O(n \log n)$ for random input
   - Worst case $O(n^2)$ arises: When input is already sorted
   - The worst case is (unlikely/possible): Very plausible

Average/worst case differences

Consider the following cases! When do the worst cases arise?

1. Quicksort(array,low,high) with pivot array[low]
   - Average case $O(n \log n)$ for random input
   - Worst case $O(n^2)$ arises: When input is already sorted
   - The worst case is (unlikely/possible): Very plausible
2. Quicksort(array, low, high) with pivot array[(low+high)/2]
   - Average case $O(n \log n)$ for random input
   - Worst case $O(n^2)$ arises:
   - The worst case is (unlikely/possible):

Average/worst case differences

Consider the following cases! When do the worst cases arise?

1. Quicksort(array,low,high) with pivot array[low]
   - Average case $O(n \log n)$ for random input
   - Worst case $O(n^2)$ arises: When input is already sorted
   - The worst case is (unlikely/possible): Very plausible
2. Quicksort(array, low, high) with pivot array[(low+high)/2]
   - Average case $O(n \log n)$ for random input
   - Worst case $O(n^2)$ arises: When middle item is always smallest
   - The worst case is (unlikely/possible):

Average/worst case differences

Consider the following cases! When do the worst cases arise?

1. Quicksort(array,low,high) with pivot array[low]
   - Average case $O(n \log n)$ for random input
   - Worst case $O(n^2)$ arises: When input is already sorted
   - The worst case is (unlikely/possible): Very plausible
2. Quicksort(array, low, high) with pivot array[(low+high)/2]
   - Average case $O(n \log n)$ for random input
   - Worst case $O(n^2)$ arises: When middle item is always smallest
   - The worst case is (unlikely/possible): Probably unlikely?

Hash table worst-case usage

- Worst case for hash table operations is $\Theta(n)$, when all items have the same hash code
  - Item 1 into slot s, time 1
  - Item 2 into slot s: time 2 (to scan existing item)
  - Item 3 into slot s: time 3 (to scan existing items)
  - ...
  - Item n into slot s: time n (to scan existing items)
- Total work to insert n such items is $\Theta(n^2)$
- Likely/unlikely?
- Worst case essentially requires malicious crafting (reverse-engineer hashCode implementation, break it to figure out how to produce collisions)
- Unless your hash function is very bad, this is unlikely

Hash table worst-case usage

- Worst case for hash table operations is $\Theta(n)$, when all items have the same hash code
  - Item 1 into slot s, time 1
  - Item 2 into slot s: time 2 (to scan existing item)
  - Item 3 into slot s: time 3 (to scan existing items)
  - ...
  - Item n into slot s: time n (to scan existing items)
- Total work to insert n such items is $\Theta(n^2)$
- Likely/unlikely?
- Worst case essentially requires malicious crafting (reverse-engineer hashCode implementation, break it to figure out how to produce collisions)
- Unless your hash function is very bad, this is unlikely

Hash table worst-case usage

- Worst case for hash table operations is $\Theta(n)$, when all items have the same hash code
  - Item 1 into slot s, time 1
  - Item 2 into slot s: time 2 (to scan existing item)
  - Item 3 into slot s: time 3 (to scan existing items)
  - ...
  - Item n into slot s: time n (to scan existing items)
- Total work to insert n such items is $\Theta(n^2)$
- Likely/unlikely?
- Worst case essentially requires malicious crafting (reverse-engineer hashCode implementation, break it to figure out how to produce collisions)
- Unless your hash function is very bad, this is unlikely

Hash table worst-case usage

- Worst case for hash table operations is $\Theta(n)$, when all items have the same hash code
  - Item 1 into slot s, time 1
  - Item 2 into slot s: time 2 (to scan existing item)
  - Item 3 into slot s: time 3 (to scan existing items)
  - ...
  - Item n into slot s: time n (to scan existing items)
- Total work to insert n such items is $\Theta(n^2)$
- Likely/unlikely?
- Worst case essentially requires malicious crafting (reverse-engineer hashCode implementation, break it to figure out how to produce collisions)
- Unless your hash function is very bad, this is unlikely

Hash table worst-case usage

- Worst case for hash table operations is $\Theta(n)$, when all items have the same hash code
  - Item 1 into slot s, time 1
  - Item 2 into slot s: time 2 (to scan existing item)
  - Item 3 into slot s: time 3 (to scan existing items)
  - ...
  - Item n into slot s: time n (to scan existing items)
- Total work to insert n such items is $\Theta(n^2)$
- Likely/unlikely?
- Worst case essentially requires malicious crafting (reverse-engineer hashCode implementation, break it to figure out how to produce collisions)
- Unless your hash function is very bad, this is unlikely

Summary

- Hash tables implement Set and Map efficiently on average
- Ingredients:
  1. Hash function x.hashCode() gives digital fingerprint
  2. Have table of p hash buckets to store objects in
  3. Some kind of collision handling (open/closed) to handle over-populated buckets
  4. If the hash function is high quality, and capacity is, e.g., prime number p $\approx$ 2n, then most requests don't collide
- Worst case unavoidably much worse, but also very unlikely

Summary

- Hash tables implement Set and Map efficiently on average
- Ingredients:
  1. Hash function x.hashCode() gives digital fingerprint
  2. Have table of p hash buckets to store objects in
  3. Some kind of collision handling (open/closed) to handle over-populated buckets
  4. If the hash function is high quality, and capacity is, e.g., prime number $p \approx 2n$, then most requests don't collide
- Worst case unavoidably much worse, but also very unlikely

Summary

- Hash tables implement Set and Map efficiently on average
- Ingredients:
  1. Hash function x.hashCode() gives digital fingerprint
  2. Have table of p hash buckets to store objects in
  3. Some kind of collision handling (open/closed) to handle over-populated buckets
  4. If the hash function is high quality, and capacity is, e.g., prime number p $\approx$ 2n, then most requests don't collide
- Worst case unavoidably much worse, but also very unlikely

Summary

- Hash tables implement Set and Map efficiently on average
- Ingredients:
  1. Hash function x.hashCode() gives digital fingerprint
  2. Have table of p hash buckets to store objects in
  3. Some kind of collision handling (open/closed) to handle over-populated buckets
  4. If the hash function is high quality, and capacity is, e.g., prime number $p \approx 2n$, then most requests don't collide
- Worst case unavoidably much worse, but also very unlikely

Summary

- Hash tables implement Set and Map efficiently on average
- Ingredients:
  1. Hash function x.hashCode() gives digital fingerprint
  2. Have table of p hash buckets to store objects in
  3. Some kind of collision handling (open/closed) to handle over-populated buckets
  4. If the hash function is high quality, and capacity is, e.g., prime number p $\approx$ 2n, then most requests don't collide
- Worst case unavoidably much worse, but also very unlikely