Hashing Continued

Instructors: Sam McCauley and Dan Barowy
April 29, 2022
Talk today: gerrymandering and how it relates to computer science (2:30 in Wege)

Any questions?
Linear Probing
Linear Probing

- General idea: store each key-value pair in the first open slot on or after its canonical slot

- Insertion: if a collision occurs at the bin, just scan forward (linearly) until an empty slot is available; store the item there
  - We “wrap around” at the end of the array
  - Let’s call a contiguous region of full bins a run

- Lookup: to find a key-value pair, calculate the bin. Then, scan linearly until the item is found or you reach the end of the run.
Tricky Part: Deletes

- Let’s look at NaiveProbing.java
Tricky Part: Deletes

- Let’s look at NaiveProbing.java
- Simple (not very good) hash function: index of first letter of word
Let’s look at NaiveProbing.java

Simple (not very good) hash function: index of first letter of word

Initial array size = 8
Tricky Part: Deletes

- Let’s look at NaiveProbing.java
- Simple (not very good) hash function: index of first letter of word
- Initial array size = 8
- Add “atlanta” to the hash table, then “detroit,” then “queens”
  - q is the 16th letter of the alphabet (0-indexed)
Let’s look at NaiveProbing.java

Simple (not very good) hash function: index of first letter of word

Initial array size = 8

Add “atlanta” to the hash table, then “detroit,” then “queens”
  • q is the 16th letter of the alphabet (0-indexed)

What happens if we remove “atlanta” and then look up “queens?”
Tricky Part: Deletes

- Let’s look at NaiveProbing.java
- Simple (not very good) hash function: index of first letter of word
- Initial array size = 8
- Add “atlanta” to the hash table, then “detroit,” then “queens”
  - q is the 16th letter of the alphabet (0-indexed)
- What happens if we remove “atlanta” and then look up “queens?”
  - Our run was broken up!
Tricky Part: Deletes

- Let's look at NaiveProbing.java

- Simple (not very good) hash function: index of first letter of word

- Initial array size = 8

- Add "atlanta" to the hash table, then "detroit," then "queens"
  - q is the 16th letter of the alphabet (0-indexed)

- What happens if we remove "atlanta" and then look up "queens?"
  - Our run was broken up!
  - Now get() won’t work correctly
Linear Probing Deletes

- When we delete an element from a run, we create a “hole”
Linear Probing Deletes

• When we delete an element from a run, we create a “hole”
  • Challenge: how do we tell if the run has ended, or if the hole was created with a deletion?
  • Solution: insert a placeholder
    • If we see the placeholder during a lookup, we treat it as a collision, and keep scanning until we find a true hole
    • If we see the placeholder during an insertion, we treat it as an open slot
    • Must still scan the whole run to make sure the key isn’t present later on
Linear Probing Deletes

- When we delete an element from a run, we create a “hole”
  - Challenge: how do we tell if the run has ended, or if the hole was created with a deletion?
- Solution: insert a placeholder

If we see the placeholder during a lookup, we treat it as a collision, and keep scanning until we find a true hole. If we see the placeholder during an insertion, we treat it as an open slot. Must still scan the whole run to make sure the key isn't present later on.
Linear Probing Deletes

- When we delete an element from a run, we create a “hole”
  - Challenge: how do we tell if the run has ended, or if the hole was created with a deletion?
- Solution: insert a placeholder
  - If we see the placeholder during a lookup, we treat it as a collision, and keep scanning until we find a true hole
  - If we see the placeholder during an insertion, we treat it as an open slot
- Must still scan the whole run to make sure the key isn't present later on
Linear Probing Deletes

- When we delete an element from a run, we create a “hole”
  - Challenge: how do we tell if the run has ended, or if the hole was created with a deletion?
- Solution: insert a placeholder
  - If we see the placeholder during a lookup, we treat it as a collision, and keep scanning until we find a true hole
  - If we see the placeholder during an insertion, we treat it as an open slot
Linear Probing Deletes

• When we delete an element from a run, we create a “hole”
  • Challenge: how do we tell if the run has ended, or if the hole was created with a deletion?

• Solution: insert a placeholder
  • If we see the placeholder during a lookup, we treat it as a collision, and keep scanning until we find a true hole
  • If we see the placeholder during an insertion, we treat it as an open slot
    • Must still scan the whole run to make sure the key isn’t present later on
Implementation

- Let’s look at HashAssociation.java
Implementation

- Let’s look at HashAssociation.java
- Finally, HasTable.java
Linear Probing Observations

- Code is more complicated than in external chaining, but still manageable
Linear Probing Observations

- Code is more complicated than in external chaining, but still manageable
- The length of a run dictates the performance
Linear Probing Observations

- Code is more complicated than in external chaining, but still manageable
- The length of a run dictates the performance
- Removing elements does not shrink the run—it defers the work to other operations
Linear Probing Observations

- Code is more complicated than in external chaining, but still manageable
- The length of a run dictates the performance
- Removing elements does not shrink the run—it defers the work to other operations
  - Keeping runs small is important, so we may want to reconsider some design decisions if we expect a lot of deletions
Linear Probing Observations

- Downsides of linear probing?
Linear Probing Observations

- Downsides of linear probing?
- What if the array is almost full?
Linear Probing Observations

- Downsides of linear probing?
- What if the array is almost full?
  - *Very* long runs

Does external chaining avoid this problem?
- Short answer: yes
- Only scan through collisions, not the entire run
- Never scans more items than linear probing!
- But: worse cache behavior (locality)
Linear Probing Observations

- Downsides of linear probing?
- What if the array is almost full?
  - Very long runs
- Does external chaining avoid this problem?
Linear Probing Observations

- Downsides of linear probing?
- What if the array is almost full?
  - *Very* long runs
- Does external chaining avoid this problem?
  - Short answer: yes
Linear Probing Observations

- Downsides of linear probing?
- What if the array is almost full?
  - *Very* long runs
- Does external chaining avoid this problem?
  - Short answer: yes
  - Only scan through collisions, not the entire run
Linear Probing Observations

- Downsides of linear probing?

- What if the array is almost full?
  - Very long runs

- Does external chaining avoid this problem?
  - Short answer: yes
  - Only scan through collisions, not the entire run
  - Never scans more items than linear probing!
Linear Probing Observations

- Downsides of linear probing?
- What if the array is almost full?
  - Very long runs
- Does external chaining avoid this problem?
  - Short answer: yes
  - Only scan through collisions, not the entire run
  - Never scans more items than linear probing!
  - But: worse cache behavior (locality)
Performance: Linear Probing vs Chaining

- What is the performance of put(K, V)?

- Linear probing: $O(1 + \text{run length})$

- External Chaining: $O(1 + \text{chain length})$

- Same for get(K), remove(K)

- So: how do we control the length of a run/length of a chain?

- Related: how do we actually choose a hash function?
Performance: Linear Probing vs Chaining

- What is the performance of $\text{put}(K, V)$?
  - Linear probing: $O(1 + \text{run length})$

- Same for $\text{get}(K)$, $\text{remove}(K)$

- Related: how do we actually choose a hash function?
Performance: Linear Probing vs Chaining

- What is the performance of $\text{put}(K, V)$?
  - Linear probing: $O(1 + \text{run length})$
  - External Chaining: $O(1 + \text{chain length})$

- So: how do we control the length of a run/length of a chain?

- Related: how do we actually choose a hash function?
Performance: Linear Probing vs Chaining

• What is the performance of \( \text{put}(K, V) \)?
  - Linear probing: \( O(1 + \text{run length}) \)
  - External Chaining: \( O(1 + \text{chain length}) \)

• Same for \( \text{get}(K) \), \( \text{remove}(K) \)
Performance: Linear Probing vs Chaining

- What is the performance of $\text{put}(K, V)$?
  - Linear probing: $O(1 + \text{run length})$
  - External Chaining: $O(1 + \text{chain length})$

- Same for $\text{get}(K), \text{remove}(K)$

- So: how do we control the length of a run/length of a chain?
Performance: Linear Probing vs Chaining

• What is the performance of \( \text{put}(K, V) \)?
  • Linear probing: \( O(1 + \text{run length}) \)
  • External Chaining: \( O(1 + \text{chain length}) \)

• Same for \( \text{get}(K), \text{remove}(K) \)

• So: how do we control the length of a run/length of a chain?

• Related: how do we actually choose a hash function?
Hashtable Size
Maintaining Hashtable Size

- Like vectors: we need to grow when we run out of space
Maintaining Hashtable Size

- Like vectors: we need to grow when we run out of space
- What do we mean by running out of space?
MaintainingHashtable Size

- Like vectors: we need to grow when we run out of space.
- What do we mean by running out of space?
- We need to make a trade-off between space and performance.
Maintaining Hashtable Size

- Like vectors: we need to grow when we run out of space

- What do we mean by running out of space?

- We need to make a trade-off between space and performance:
  - We want our table size to be large to minimize collisions (and run/chain lengths): leads to good performance, bad space
Maintaining Hashtable Size

- Like vectors: we need to grow when we run out of space

- What do we mean by running out of space?

- We need to make a trade-off between space and performance:
  - We want our table size to be large to minimize collisions (and run/chain lengths): leads to good performance, bad space
  - We want our table size to be small to minimize wasted space (empty slots): leads to good space, bad performance
Maintaining Hashtable Size

- Like vectors: we need to grow when we run out of space

- What do we mean by running out of space?

- We need to make a trade-off between space and performance:
  - We want our table size to be large to minimize collisions (and run/chain lengths): leads to good performance, bad space
  - We want our table size to be small to minimize wasted space (empty slots): leads to good space, bad performance

- Some flexibility (like with Vectors): we don’t know the size up front
Load Factor

- Suppose a hash table with $m$ slots stores $n$ elements
Load Factor

- Suppose a hash table with $m$ slots stores $n$ elements.

- *Load factor* is a measure of how full the hash table is:

  \[
  \text{load factor} = \frac{\text{# elements}}{\text{# slots}} = \frac{n}{m}
  \]
Load Factor

- Suppose a hash table with \( m \) slots stores \( n \) elements

- *Load factor* is a measure of how full the hash table is

  \[
  \text{load factor} = \frac{\# \text{ elements}}{\# \text{ slots}} = \frac{n}{m}
  \]

- A smaller load factor means the hashtable is less full, which likely gives better performance
Using the Load Factor

- We can keep a running count of the table’s elements so that we always know the load factor
Using the Load Factor

- We can keep a running count of the table’s elements so that we always know the load factor
- Given a hashtable’s load factor, what should we do?
Using the Load Factor

- We can keep a running count of the table’s elements so that we always know the load factor.
- Given a hashtable’s load factor, what should we do?
  - If the load factor is high (say > .5), we grow our table.
Using the Load Factor

- We can keep a running count of the table’s elements so that we always know the load factor
- Given a hashtable’s load factor, what should we do?
  - If the load factor is high (say \( > 0.5 \)), we grow our table
- How to grow?
Using the Load Factor

- We can keep a running count of the table’s elements so that we always know the load factor.

- Given a hashtable’s load factor, what should we do?
  - If the load factor is high (say > .5), we grow our table.

- How to grow?

- Vectors: `ensureCapacity()` allocates a new `Object` array, then copies elements over.
Using the Load Factor

- We can keep a running count of the table’s elements so that we always know the load factor.

- Given a hashtable’s load factor, what should we do?
  - If the load factor is high (say \( > 0.5 \)), we grow our table.

- How to grow?
  - Vectors: `ensureCapacity()` allocates a new `Object` array, then copies elements over.

- Does this work for hashtables?
Making Hashtables Larger

- Cannot just copy values! (why?)

  - The canonical slot might change
  - Example: suppose key.hashCode() == 11
  - Then 11 % 8 == 3 but 11 % 16 == 11
  - How can we handle this?
  - To grow our hashtable, we must recompute the canonical slot for each item, then reinsert the item into the new array
Making Hashtables Larger

- Cannot just copy values! (why?)

- The canonical slot might change

Example: suppose key.hashCode() == 11

Then 11 % 8 == 3 but 11 % 16 == 11

To grow our hashtable, we must recompute the canonical slot for each item, then reinsert the item into the new array.
Making Hashtables Larger

- Cannot just copy values! (why?)
- The canonical slot might change

Example: suppose key.hashCode() == 11
Making Hashtables Larger

- Cannot just copy values! (why?)
- The canonical slot might change
- Example: suppose `key.hashCode() == 11`
  - Then `11 % 8 == 3` but `11 % 16 == 11`
Making Hashtables Larger

- Cannot just copy values! (why?)

- The canonical slot might change

- Example: suppose `key.hashCode() == 11`

- Then `11 % 8 == 3` but `11 % 16 == 11`

- How can we handle this?
Making Hashtables Larger

- Cannot just copy values! (why?)
- The canonical slot might change
  - **Example:** suppose `key.hashCode() == 11`
  - Then `11 % 8 == 3` but `11 % 16 == 11`
- How can we handle this?
- To grow our hashtable, we must recompute the canonical slot for each item, then reinsert the item into the new array
When to grow?

- Choose some load factor
When to grow?

- Choose some load factor

- .50 and .66 are very popular; depends a bit on the use case
When to grow?

- Choose some load factor

- .50 and .66 are very popular; depends a bit on the use case

- Tradeoff between size and performance
When to grow?

- Choose some load factor

- .50 and .66 are very popular; depends a bit on the use case

- Tradeoff between size and performance

- structure5 Hashtable uses .6
Array Sizes

- Some people like using hash tables whose size is a prime

Reason: remember that we use \( \% \) array.length to calculate the canonical slot. A prime size can help "spread out" the items. Downside: need to find a prime size when doubling. We won't worry about this in this class; just a heads up. You'll often see a hash table of size 997 or something—this is why.
Array Sizes

- Some people like using hash tables whose size is a prime
- Reason: remember that we use `% array.length` to calculate the canonical slot
Array Sizes

- Some people like using hash tables whose size is a prime
- Reason: remember that we use `array.length %` to calculate the canonical slot
- A prime size can help “spread out” the items
Array Sizes

- Some people like using hash tables whose size is a prime

- Reason: remember that we use `% array.length` to calculate the canonical slot

- A prime size can help “spread out” the items

- Downside: need to find a prime size when doubling
Some people like using hash tables whose size is a prime.

Reason: remember that we use `arr.length` to calculate the canonical slot.

A prime size can help “spread out” the items.

Downside: need to find a prime size when doubling.

We won’t worry about this in this class; just a heads up. You’ll often see a hash table of size 997 or something—this is why.
Choosing Hash Functions
Good Hash Functions

- Good hash functions:
  - Are fast to compute
  - Uniformly distribute keys across the range
  - Rules of thumb to make good hash functions? Not really. We almost always have to test "goodness" empirically
Good Hash Functions

- Good hash functions:
  - Are fast to compute
Good Hash Functions

- Good hash functions:
  - Are fast to compute
  - Uniformly distribute keys across the range
Good Hash Functions

- Good hash functions:
  - Are fast to compute
  - Uniformly distribute keys across the range

- Rules of thumb to make good hash functions?
Good Hash Functions

- Good hash functions:
  - Are fast to compute
  - Uniformly distribute keys across the range

- Rules of thumb to make good hash functions?
  - Not really. We almost always have to test “goodness” empirically
Hashing Strings

- What are some reasonable hash functions for Strings
  - One idea: use the first character's Unicode value? (Every character is stores as a number in Java).
    - Problems with this?
      - Can only return 0–255
      - Not uniform (some letters far more common)
    - Sum of the Unicode values of all characters?
      - Still not uniform! (We'll see in a second)
      - Doesn't work well for large hashtables
      - Not good at avoiding collisions: smile, limes, miles, and slime are all the same
Hashing Strings

- What are some reasonable hash functions for Strings?
- One idea: use the first character’s Unicode value? (Every character is stored as a number in Java). Problems with this?
Hashing Strings

- What are some reasonable hash functions for Strings

- One idea: use the first character’s Unicode value? (Every character is stores as a number in Java). Problems with this?
  - Can only return 0–255
Hashing Strings

- What are some reasonable hash functions for Strings

- One idea: use the first character’s Unicode value? (Every character is stores as a number in Java). Problems with this?
  - Can only return 0–255
  - Not uniform (some letters far more common)
Hashing Strings

• What are some reasonable hash functions for Strings

• One idea: use the first character’s Unicode value? (Every character is stores as a number in Java). Problems with this?
  • Can only return 0–255
  • Not uniform (some letters far more common)

• Sum of the Unicode values of all characters?
Hashing Strings

- What are some reasonable hash functions for Strings?
- One idea: use the first character’s Unicode value? (Every character is stores as a number in Java). Problems with this?
  - Can only return 0–255
  - Not uniform (some letters far more common)
- Sum of the Unicode values of all characters?
  - Still not uniform! (We’ll see in a second)
Hashing Strings

• What are some reasonable hash functions for Strings

• One idea: use the first character’s Unicode value? (Every character is stores as a number in Java). Problems with this?
  • Can only return 0–255
  • Not uniform (some letters far more common)

• Sum of the Unicode values of all characters?
  • Still not uniform! (We’ll see in a second)
  • Doesn’t work well for large hashtables
Hashing Strings

- What are some reasonable hash functions for Strings

- One idea: use the first character’s Unicode value? (Every character is stores as a number in Java). Problems with this?
  
  - Can only return 0–255
  
  - Not uniform (some letters far more common)

- Sum of the Unicode values of all characters?
  
  - Still not uniform! (We’ll see in a second)
  
  - Doesn’t work well for large hashtables

  - Not good at avoiding collisions: smile, limes, miles, and slime are all the same
Sum of Unicode Values

- This is the hash of all words in the UNIX spellchecking dictionary
  - $x$-axis is bucket; $y$-axis is number of words that hash to the bucket
Sum of Unicode Values

- This is the hash of all words in the UNIX spellchecking dictionary
  - $x$-axis is bucket; $y$-axis is number of words that hash to the bucket
- Uses 997 buckets
Sum of Unicode Values

- This is the hash of all words in the UNIX spellchecking dictionary
  - $x$-axis is bucket; $y$-axis is number of words that hash to the bucket
- Uses 997 buckets
- Hash of a string $s$: $\sum_{i=0}^{s.length} s.charAt(i)$
Sum of Unicode Values

- Hash of a string $s$: $\sum_{i=0}^{\text{length } s} 2^i \cdot s\text{.charAt}(i)$
Sum of Unicode Values

- Hash of a string $s$: $\sum_{i=0}^{s.length} 2^i \cdot s.charAt(i)$
- Better! But still not great.
Sum of Unicode Values

- Hash of a string $s$: $\sum_{i=0}^{s.length} 256^i \cdot s.charAt(i)$
• Hash of a string $s$: $\sum_{i=0}^{s.length} 256^i \cdot s.charAt(i)$

• Really good! But do we need numbers as big as $256^i$?
Sum of Unicode Values

- Hash of a string $s$: $\sum_{i=0}^{s.length} 31^i \cdot s.charAt(i)$
Sum of Unicode Values

- Hash of a string $s$: $\sum_{i=0}^{s.length} 31^i \cdot s.charAt(i)$
- This is (essentially) what Java uses to hash strings!
Other Objects?

- Integers?

In Java: `i.hashCode()` is `i`.

Could be terrible depending on your data. Might want to use another `hashCode()` method in that case. One popular one (has theoretical performance guarantees!):

\[ h(x) = (ax + b) \mod p \]

What about other classes? Write your own (probably similar) `hashCode()` methods. Test empirically to make sure elements are spread out.
Other Objects?

- Integers?
  - In Java: `i.hashCode()` is `i`

- Might want to use another `hashCode()` method in that case

- One popular one (has theoretical performance guarantees!):
  
  \[ h(x) = (ax + b) \mod p \]

- What about other classes?
  - Write your own (probably similar) `hashCode()` methods. Test empirically to make sure elements are spread out
Other Objects?

- Integers?
  - In Java: `i.hashCode()` is `i`
  - Could be terrible depending on your data

- Might want to use another `hashCode()` method in that case

  One popular one (has theoretical performance guarantees!):
  
  \[ h(x) = (ax + b) \mod p \]
Other Objects?

- Integers?
  - In Java: `i.hashCode() is i`
  - Could be terrible depending on your data
  - Might want to use another `hashCode()` method in that case
Other Objects?

- Integers?
  - In Java: `i.hashCode()` is `i`
  - Could be terrible depending on your data
  - Might want to use another `hashCode()` method in that case
  - One popular one (has theoretical performance guarantees!):

\[ h(x) = (ax + b) \mod p \]
Other Objects?

- Integers?
  - In Java: `i.hashCode()` is `i`
  - Could be terrible depending on your data
  - Might want to use another `hashCode()` method in that case
  - One popular one (has theoretical performance guarantees!):
    \[ h(x) = (ax + b) \mod p \]

- What about other classes?
Other Objects?

- Integers?
  - In Java: `i.hashCode() is i`
  - Could be terrible depending on your data
  - Might want to use another `hashCode()` method in that case
  - One popular one (has theoretical performance guarantees!):
    \[ h(x) = (ax + b) \mod p \]

- What about other classes?
  - Write your own (probably similar) `hashCode()` methods. Test empirically to make sure elements are spread out
Hashtable Performance
Hashtable Performance

- Given the hash code of an object $o$, how long does $\text{get}(o)$ take?

- $O(\text{run length})$ for linear probing; $O(\text{chain length})$ for external chaining

- Assumes that $\text{.equals()}$ is $O(1)$ time

- How long does calculating a hash code take? Can be long for, say, a long string.

- $O(1)$ in terms of the number of items in the hash table

- Another example of being careful about how we're stating our running time. Usually: in terms of number of strings in the table. But do we care about the length of our strings?
Hashtable Performance

- Given the hash code of an object $o$, how long does $get(o)$ take?
- $O(\text{run length})$ for linear probing; $O(\text{chain length})$ for external chaining
Hashtable Performance

- Given the hash code of an object `o`, how long does `get(o)` take?
- \(O(\text{run length})\) for linear probing; \(O(\text{chain length})\) for external chaining
- Assumes that `.equals()` is \(O(1)\) time
Hashtable Performance

- Given the hash code of an object \( o \), how long does \( \text{get}(o) \) take?

- \( O(\text{run length}) \) for linear probing; \( O(\text{chain length}) \) for external chaining

- Assumes that .\text{equals}() is \( O(1) \) time

- How long does calculating a hash code take?
Hashtable Performance

- Given the hash code of an object \( o \), how long does \( \text{get}(o) \) take?
- \( O(\text{run length}) \) for linear probing; \( O(\text{chain length}) \) for external chaining
- Assumes that \( .\text{equals}() \) is \( O(1) \) time
- How long does calculating a hash code take?
  - Can be long for, say, a long string.
Hashtable Performance

- Given the hash code of an object \( o \), how long does \( \text{get}(o) \) take?
  - \( O(\text{run length}) \) for linear probing; \( O(\text{chain length}) \) for external chaining
- Assumes that \( \text{.equals()} \) is \( O(1) \) time
- How long does calculating a hash code take?
  - Can be long for, say, a long string.
  - \( O(1) \) in terms of the number of items in the hash table
Hashtable Performance

- Given the hash code of an object \( o \), how long does \( \text{get}(o) \) take?

- \( O(\text{run length}) \) for linear probing; \( O(\text{chain length}) \) for external chaining

- Assumes that \( \text{equals}() \) is \( O(1) \) time

- How long does calculating a hash code take?
  - Can be long for, say, a long string.
  - \( O(1) \) in terms of the number of items in the hash table
  - Another example of being careful about how we’re stating our running time. Usually: in terms of number of strings in the table. But do we care about the length of our strings?
Impact on Performance

• Let’s say we have constant load factor
Impact on Performance

- Let’s say we have constant load factor
- Assume we have a good hash function
Impact on Performance

- Let’s say we have constant load factor
- Assume we have a good hash function
  - Spreads objects out “like random”
Impact on Performance

- Let’s say we have constant load factor
- Assume we have a good hash function
  - Spreads objects out “like random”
- Then an average bucket has constant chain length
Impact on Performance

- Let’s say we have constant load factor
- Assume we have a good hash function
  - Spreads objects out “like random”
- Then an *average* bucket has *constant chain length*
- An *average* bucket is in a run of *constant length*
Impact on Performance

• Let’s say we have constant load factor

• Assume we have a good hash function
  • Spreads objects out “like random”

• Then an *average* bucket has constant chain length

• An *average* bucket is in a run of constant length

• (With overwhelming probability, never gets worse than $O(\log n)$ for any bucket)
Impact on Performance

- Let’s say we have constant load factor
- Assume we have a good hash function
  - Spreads objects out “like random”
- Then an *average* bucket has *constant* chain length
- An *average* bucket is in a run of *constant* length
  (With overwhelming probability, never gets worse than $O(\log n)$ for any bucket)
- Usually we say we have $O(1)$ performance. True on average; the actual worst case might be a bit worse
## Summary of Map Performance

<table>
<thead>
<tr>
<th></th>
<th>put</th>
<th>get</th>
<th>space</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unsorted Vector</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>Unsorted List</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>Sorted Vector</td>
<td>$O(n)$</td>
<td>$O(\log n)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>Balanced BST</td>
<td>$O(\log n)$</td>
<td>$O(\log n)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>Hashtable (average)</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
<td>$O(n)$</td>
</tr>
</tbody>
</table>