Sets and Maps / Hash tables

CS 2860: Algorithms and Complexity

Magnus Wahlström, McCrea 113

Magnus.Wahlstrom@rhul.ac.uk

February 11, 2019

The Set and Map ADTs

The Set ADT

- Recall sets (from CS1860). Quote (Wikipedia):

  > *In mathematics, a set is a collection of distinct objects, considered as an object in its own right.*

- Operations:
  - Set.insert(item)
  - Set.delete(item)
  - Set.contains(item) – does the set contain a copy of item?
- Behaviour (contract):
  - A set contains only distinct objects, hence it will keep only one copy of every item you inserted.
  - Sequence (Set.insert(2), Set.insert(2), Set.delete(2)) means that Set does not contain the number 2
- May / may not support "order-based" type operations (e.g., min)
- Supports some iteration, size, etc. (see later)

The Set ADT

- Recall sets (from CS1860). Quote (Wikipedia):

  In mathematics, a set is a collection of distinct objects, considered as an object in its own right.

- Operations:
  - Set.insert(item)
  - Set.delete(item)
  - Set.contains(item) – does the set contain a copy of item?
- Behaviour (contract):
  - A set contains only distinct objects, hence it will keep only one copy of every item you inserted.
  - Sequence (Set.insert(2), Set.insert(2), Set.delete(2)) means that Set does not contain the number 2
- May / may not support "order-based" type operations (e.g., min)
- Supports some iteration, size, etc. (see later)

The Set ADT

- Recall sets (from CS1860). Quote (Wikipedia):

  In mathematics, a set is a collection of distinct objects, considered as an object in its own right.

- Operations:
  - Set.insert(item)
  - Set.delete(item)
  - Set.contains(item) – does the set contain a copy of item?
- Behaviour (contract):
  - A set contains only distinct objects, hence it will keep only one copy of every item you inserted.
  - Sequence (Set.insert(2), Set.insert(2), Set.delete(2)) means that Set does not contain the number 2
- May / may not support "order-based" type operations (e.g., min)
- Supports some iteration, size, etc. (see later)

The Set ADT

- Recall sets (from CS1860). Quote (Wikipedia):

  > *In mathematics, a set is a collection of distinct objects, considered as an object in its own right.*

- Operations:
  - Set.insert(item)
  - Set.delete(item)
  - Set.contains(item) – does the set contain a copy of item?
- Behaviour (contract):
  - A set contains only distinct objects, hence it will keep only one copy of every item you inserted.
  - Sequence (Set.insert(2), Set.insert(2), Set.delete(2)) means that Set does not contain the number 2
- May / may not support "order-based" type operations (e.g., min)
- Supports some iteration, size, etc. (see later)

The Set ADT

- Recall sets (from CS1860). Quote (Wikipedia):

  > *In mathematics, a set is a collection of distinct objects, considered as an object in its own right.*

- Operations:
  - Set.insert(item)
  - Set.delete(item)
  - Set.contains(item) – does the set contain a copy of item?
- Behaviour (contract):
  - A set contains only distinct objects, hence it will keep only one copy of every item you inserted.
  - Sequence (Set.insert(2), Set.insert(2), Set.delete(2)) means that Set does not contain the number 2
- May / may not support "order-based" type operations (e.g., min)
- Supports some iteration, size, etc. (see later)

Set usage example

Action	Status / returns
(initial `set` empty)	`set=`$\{\}$
`set.add(1);`	`set=`$\{$`1`$\}$
`set.add(3);`	`set=`$\{$`1,3`$\}$
`set.contains(1)`	returns `true`

Set usage example

Action	Status / returns
(initial set empty)	`set={}`
`set.add(1);`	`set={1}`
`set.add(3);`	`set={1,3}`
`set.contains(1)`	returns `true`
`set.add(1);`	`set={1,3}`
`set.add(4);`	`set={1,3,4}`
`set.add(1);`	`set={1,3,4}`
`set.add(1);`	`set={1,3,4}`

Set usage example

Action	Status / returns
(initial set empty)	set={}
`set.add(1);`	set={1}
`set.add(3);`	set={1,3}
`set.contains(1)`	returns `true`
`set.add(1);`	set={1,3}
`set.add(4);`	set={1,3,4}
`set.add(1);`	set={1,3,4}
`set.add(1);`	set={1,3,4}
`set.delete(1)`	set={3,4}
`set.contains(1)`	returns `false`

Uses

- Collecting items, without duplicates
  - Given a long list of names, some repeating, create a collection containing each name only once
- The CS1860 set operations (union, intersection, difference)
  1. Given several lists of names, collect every name occurring in at least one list
  2. "Find all names from list 1 that are not in list 2"
- Operations insert, contains, delete useful in many programs

Uses

- Collecting items, without duplicates
  - Given a long list of names, some repeating, create a collection containing each name only once
- The CS1860 set operations (union, intersection, difference)
  1. Given several lists of names, collect every name occurring in at least one list
  2. "Find all names from list 1 that are not in list 2"
- Operations insert, contains, delete useful in many programs

Set classes in Java

- Two implementations of Set in Java collections library
- TreeSet: Uses balanced binary search tree (red-black tree)
  - A tree containing n objects supports all our operations (including "order-based") in $O(\log n)$ time per operation
  - Supports iteration in sorted order
- HashSet: Uses hash table (seen later)
  - Faster for insert/contains/delete, slow for ordering
  - Supports iteration, but in arbitrary order
- Declaration: `Set<DataType>`, e.g., `HashSet<Integer>`, `TreeSet<String>`

Set classes in Java

- Two implementations of Set in Java collections library
- TreeSet: Uses balanced binary search tree (red-black tree)
  - A tree containing n objects supports all our operations (including "order-based") in $O(\log n)$ time per operation
  - Supports iteration in sorted order
- HashSet: Uses hash table (seen later)
  - Faster for insert/contains/delete, slow for ordering
  - Supports iteration, but in arbitrary order
- Declaration: `Set<DataType>`, e.g., `HashSet<Integer>`, `TreeSet<String>`

Examples: Algorithms using Set

1. Have a list of Strings (with repeats), want to print every string only exactly once
2. Have a list of Strings, want to count how many distinct strings there are
3. Have two lists of names, want people who occur in both lists
4. Have several lists of objects, want items which occur in at least one list (but want only one copy of each)

Examples: Algorithms using Set

1. Have a list of Strings (with repeats), want to print every string only exactly once
2. Have a list of Strings, want to count how many distinct strings there are
3. Have two lists of names, want people who occur in both lists
4. Have several lists of objects, want items which occur in at least one list (but want only one copy of each)

Examples: Algorithms using Set

1. Have a list of Strings (with repeats), want to print every string only exactly once
2. Have a list of Strings, want to count how many distinct strings there are
3. Have two lists of names, want people who occur in both lists
4. Have several lists of objects, want items which occur in at least one list (but want only one copy of each)

Examples: Algorithms using Set

1. Have a list of Strings (with repeats), want to print every string only exactly once
2. Have a list of Strings, want to count how many distinct strings there are
3. Have two lists of names, want people who occur in both lists
4. Have several lists of objects, want items which occur in at least one list (but want only one copy of each)

Map ADT

Specification of `Map` ADT:

- Collection of mappings from keys to values
  (`"janet"` $\mapsto$ `1234`)
- Contains at most one entry per key
- Basic ops: `get(key)`, `put(key,value)`, `remove(key)`,
  the test `containsKey(key)`
- Supports some kind of iteration on keys and values
  (order unspecified)

Map: Phone book example

Action	Result/contents
(initial map empty)	`map={}`
`map.put("john",1234);`	`map={john=1234}`
`map.put("jane",1235);`	`map={john=1234,jane=1235}`
`map.containsKey("john");`	returns `true`

Map: Phone book example

Action	Result/contents
(initial map empty)	`map={}`
`map.put("john",1234);`	`map={john=1234}`
`map.put("jane",1235);`	`map={john=1234,jane=1235}`
`map.containsKey("john");`	returns `true`
`map.put("john",1236);)`	`map={john=1236,jane=1235}`

Map: Phone book example

Action	Result/contents
(initial map empty)	`map={}`
`map.put("john",1234);`	`map={john=1234}`
`map.put("jane",1235);`	`map={john=1234,jane=1235}`
`map.containsKey("john");`	returns `true`
`map.put("john",1236);)`	`map={john=1236,jane=1235}`
`map.remove("john");`	`map={jane=1235}`
`map.containsKey("john");`	returns `false`

Maps via BSTs

- Give TreeNode object two new attributes:
  - node.key: Attribute to sort by (previously called "node.data"); e.g., name in phone book
  - node.value: Property of the key; e.g., phone number in phone book
- Operations (efficient):
  - containsKey(key): same as BST contains (return true/false)
  - get(key): same as BST contains (return node.value or null)
  - put(key,value): same as BST insert
    1. Locate node by key (or create new node)
    2. Modify node.value to new value
  - remove(key): same as BST delete (by key)
- Efficient or not?
  - minKey():
  - minValue():
  - containsValue():
  - Iteration over keys:
  - Iteration over values:

Maps via BSTs

- Give TreeNode object two new attributes:
  - node.key: Attribute to sort by (previously called "node.data"); e.g., name in phone book
  - node.value: Property of the key; e.g., phone number in phone book
- Operations (efficient):
  - containsKey(key): same as BST contains (return true/false)
  - get(key): same as BST contains (return node.value or null)
  - put(key,value): same as BST insert
    1. Locate node by key (or create new node)
    2. Modify node.value to new value
  - remove(key): same as BST delete (by key)
- Efficient or not?
  - minKey():
  - minValue():
  - containsValue():
  - Iteration over keys:
  - Iteration over values:

Maps via BSTs

- Give TreeNode object two new attributes:
  - node.key: Attribute to sort by (previously called "node.data"); e.g., name in phone book
  - node.value: Property of the key; e.g., phone number in phone book
- Operations (efficient):
  - containsKey(key): same as BST contains (return true/false)
  - get(key): same as BST contains (return node.value or null)
  - put(key,value): same as BST insert
    1. Locate node by key (or create new node)
    2. Modify node.value to new value
  - remove(key): same as BST delete (by key)
- Efficient or not?
  - minKey():
  - minValue():
  - containsValue():
  - Iteration over keys:
  - Iteration over values:

Maps via BSTs

- Give TreeNode object two new attributes:
  - node.key: Attribute to sort by (previously called "node.data"); e.g., name in phone book
  - node.value: Property of the key; e.g., phone number in phone book
- Operations (efficient):
  - containsKey(key): same as BST contains (return true/false)
  - get(key): same as BST contains (return node.value or null)
  - put(key,value): same as BST insert
    1. Locate node by key (or create new node)
    2. Modify node.value to new value
  - remove(key): same as BST delete (by key)
- Efficient or not?
  - minKey():
  - minValue():
  - containsValue():
  - Iteration over keys:
  - Iteration over values:

Maps via BSTs

- Give TreeNode object two new attributes:
  - node.key: Attribute to sort by (previously called "node.data"); e.g., name in phone book
  - node.value: Property of the key; e.g., phone number in phone book
- Operations (efficient):
  - containsKey(key): same as BST contains (return true/false)
  - get(key): same as BST contains (return node.value or null)
  - put(key,value): same as BST insert
    1. Locate node by key (or create new node)
    2. Modify node.value to new value
  - remove(key): same as BST delete (by key)
- Efficient or not?
  - minKey():
  - minValue():
  - containsValue():
  - Iteration over keys:
  - Iteration over values:

Maps via BSTs

- Give TreeNode object two new attributes:
  - node.key: Attribute to sort by (previously called "node.data"); e.g., name in phone book
  - node.value: Property of the key; e.g., phone number in phone book
- Operations (efficient):
  - containsKey(key): same as BST contains (return true/false)
  - get(key): same as BST contains (return node.value or null)
  - put(key,value): same as BST insert
    1. Locate node by key (or create new node)
    2. Modify node.value to new value
  - remove(key): same as BST delete (by key)
- Efficient or not?
  - minKey():
  - minValue():
  - containsValue():
  - Iteration over keys:
  - Iteration over values:

Maps via BSTs

- Give TreeNode object two new attributes:
  - node.key: Attribute to sort by (previously called “node.data”); e.g., name in phone book
  - node.value: Property of the key; e.g., phone number in phone book
- Operations (efficient):
  - containsKey(key): same as BST contains (return true/false)
  - get(key): same as BST contains (return node.value or null)
  - put(key,value): same as BST insert
    1. Locate node by key (or create new node)
    2. Modify node.value to new value
  - remove(key): same as BST delete (by key)
- Efficient or not?
  - minKey():
  - minValue():
  - containsValue():
  - Iteration over keys:
  - Iteration over values:

Maps via BSTs

- Give TreeNode object two new attributes:
  - node.key: Attribute to sort by (previously called “node.data”); e.g., name in phone book
  - node.value: Property of the key; e.g., phone number in phone book
- Operations (efficient):
  - containsKey(key): same as BST contains (return true/false)
  - get(key): same as BST contains (return node.value or null)
  - put(key,value): same as BST insert
    1. Locate node by key (or create new node)
    2. Modify node.value to new value
  - remove(key): same as BST delete (by key)
- Efficient or not?
  - minKey():
  - minValue():
  - containsValue():
  - Iteration over keys:
  - Iteration over values:

Maps via BSTs

- Give TreeNode object two new attributes:
  - node.key: Attribute to sort by (previously called "node.data"); e.g., name in phone book
  - node.value: Property of the key; e.g., phone number in phone book
- Operations (efficient):
  - containsKey(key): same as BST contains (return true/false)
  - get(key): same as BST contains (return node.value or null)
  - put(key,value): same as BST insert
    1. Locate node by key (or create new node)
    2. Modify node.value to new value
  - remove(key): same as BST delete (by key)
- Efficient or not?
  - minKey():
  - minValue():
  - containsValue():
  - Iteration over keys:
  - Iteration over values:

Maps via BSTs

- Give TreeNode object two new attributes:
  - node.key: Attribute to sort by (previously called "node.data"); e.g., name in phone book
  - node.value: Property of the key; e.g., phone number in phone book
- Operations (efficient):
  - containsKey(key): same as BST contains (return true/false)
  - get(key): same as BST contains (return node.value or null)
  - put(key,value): same as BST insert
    1. Locate node by key (or create new node)
    2. Modify node.value to new value
  - remove(key): same as BST delete (by key)
- Efficient or not?
  - minKey():
  - minValue():
  - containsValue():
  - Iteration over keys:
  - Iteration over values:

Maps via BSTs

- Give TreeNode object two new attributes:
  - node.key: Attribute to sort by (previously called "node.data"); e.g., name in phone book
  - node.value: Property of the key; e.g., phone number in phone book
- Operations (efficient):
  - containsKey(key): same as BST contains (return true/false)
  - get(key): same as BST contains (return node.value or null)
  - put(key,value): same as BST insert
    1. Locate node by key (or create new node)
    2. Modify node.value to new value
  - remove(key): same as BST delete (by key)
- Efficient or not?
  - minKey(): Efficient – as BST min operation
  - minValue():
  - containsValue():
  - Iteration over keys:
  - Iteration over values:

Maps via BSTs

- Give TreeNode object two new attributes:
  - node.key: Attribute to sort by (previously called "node.data"); e.g., name in phone book
  - node.value: Property of the key; e.g., phone number in phone book
- Operations (efficient):
  - containsKey(key): same as BST contains (return true/false)
  - get(key): same as BST contains (return node.value or null)
  - put(key,value): same as BST insert
    1. Locate node by key (or create new node)
    2. Modify node.value to new value
  - remove(key): same as BST delete (by key)
- Efficient or not?
  - minKey(): Efficient – as BST min operation
  - minValue(): Inefficient – no order between values
  - containsValue():
  - Iteration over keys:
  - Iteration over values:

Maps via BSTs

- Give TreeNode object two new attributes:
  - node.key: Attribute to sort by (previously called "node.data"); e.g., name in phone book
  - node.value: Property of the key; e.g., phone number in phone book
- Operations (efficient):
  - containsKey(key): same as BST contains (return true/false)
  - get(key): same as BST contains (return node.value or null)
  - put(key,value): same as BST insert
    1. Locate node by key (or create new node)
    2. Modify node.value to new value
  - remove(key): same as BST delete (by key)
- Efficient or not?
  - minKey(): Efficient – as BST min operation
  - minValue(): Inefficient – no order between values
  - containsValue(): Inefficient – no order, must check whole tree
  - Iteration over keys:
  - Iteration over values:

Maps via BSTs

- Give TreeNode object two new attributes:
  - node.key: Attribute to sort by (previously called "node.data"); e.g., name in phone book
  - node.value: Property of the key; e.g., phone number in phone book
- Operations (efficient):
  - containsKey(key): same as BST contains (return true/false)
  - get(key): same as BST contains (return node.value or null)
  - put(key,value): same as BST insert
    1. Locate node by key (or create new node)
    2. Modify node.value to new value
  - remove(key): same as BST delete (by key)
- Efficient or not?
  - minKey(): Efficient – as BST min operation
  - minValue(): Inefficient – no order between values
  - containsValue(): Inefficient – no order, must check whole tree
  - Iteration over keys: Yes, ordered iteration
  - Iteration over values:

Maps via BSTs

- Give TreeNode object two new attributes:
  - node.key: Attribute to sort by (previously called "node.data"); e.g., name in phone book
  - node.value: Property of the key; e.g., phone number in phone book
- Operations (efficient):
  - containsKey(key): same as BST contains (return true/false)
  - get(key): same as BST contains (return node.value or null)
  - put(key,value): same as BST insert
    1. Locate node by key (or create new node)
    2. Modify node.value to new value
  - remove(key): same as BST delete (by key)
- Efficient or not?
  - minKey(): Efficient – as BST min operation
  - minValue(): Inefficient – no order between values
  - containsValue(): Inefficient – no order, must check whole tree
  - Iteration over keys: Yes, ordered iteration
  - Iteration over values: Yes, but not in order

HashMap / TreeMap

- Java implementation of map using hash table (in HashMap), resp. red-black tree (in TreeMap)
- Hash table: Fast ($O(1)$ average time) on basic ops, does not support efficient ordering
- Red-black tree: Slightly slower ($O(\log n)$ time basic ops), supports more efficient methods (ordering: min/max, successor/predecessor, ...)
- Declaration: `Map<KeyType,ValueType>`, e.g.:
  - `HashMap<String,Integer>` to map from strings to numbers, no order on strings
  - `TreeMap<String,String>` to map from strings to strings, can efficiently search for "smallest" key, etc.

HashMap / TreeMap

- Java implementation of map using hash table (in HashMap), resp. red-black tree (in TreeMap)
- Hash table: Fast ($O(1)$ average time) on basic ops, does not support efficient ordering
- Red-black tree: Slightly slower ($O(\log n)$ time basic ops), supports more efficient methods (ordering: min/max, successor/predecessor, ...)
- Declaration: `Map<KeyType,ValueType>`, e.g.:
  - `HashMap<String,Integer>` to map from strings to numbers, no order on strings
  - `TreeMap<String,String>` to map from strings to strings, can efficiently search for "smallest" key, etc.

Examples

How can the following be implemented via maps?

1. A set (for example, set of String)?
2. A counting set (a.k.a. multiset) – keeps track of the number of copies of each item?
3. An array (say, `Integer[]`)?
4. Assume you receive a long list of Book objects, with Author information (e.g., String Book.author). You want to find the author who has written the most books in the list.
5. As above, but the list contains repetitions, and you only want to count distinct books.

Hash tables

Hash tables

- One-sentence explanation: A hash table is like an "advanced array", whose indices can be arbitrary objects (strings, lists, class objects...), not just integers
  - Ex: If A is an array, `A[0]="dog"` can be read as "associate the key 0 with the value 'dog'"
  - For a hash table, you could also associate the key "dog" with the value 0
- Good news: Operations set, get, check membership in constant average time $O(1)$ (just like ordinary arrays, *on average*).
- Restriction: No ordering supported. Cannot perform `min`, `successor`, ... (without going over all data inserted, e.g., $O(n)$ running time)

Hash tables

- One-sentence explanation: A hash table is like an "advanced array", whose indices can be arbitrary objects (strings, lists, class objects. . .), not just integers
  - Ex: If A is an array, `A[0]="dog"` can be read as "associate the key 0 with the value 'dog'"
  - For a hash table, you could also associate the key "dog" with the value 0
- Good news: Operations set, get, check membership in constant average time $O(1)$ (just like ordinary arrays, *on average*).
- Restriction: No ordering supported. Cannot perform `min`, `successor`, . . . (without going over all data inserted, e.g., $O(n)$ running time)

Hash tables

- One-sentence explanation: A hash table is like an "advanced array", whose indices can be arbitrary objects (strings, lists, class objects...), not just integers
  - Ex: If A is an array, `A[0]="dog"` can be read as "associate the key 0 with the value 'dog'"
  - For a hash table, you could also associate the key "dog" with the value 0
- Good news: Operations set, get, check membership in constant average time $O(1)$ (just like ordinary arrays, *on average*).
- Restriction: No ordering supported. Cannot perform `min`, `successor`, ... (without going over all data inserted, e.g., $O(n)$ running time)

Example

Python code (notation cleaner than Java):

```
>>> A=dict()
>>> A["cat"]=2
>>> A["dog"]=3
>>> A["cat"]
2
>>> s="dog"
>>> A[s]
3
>>> A["squirrel"]
  ---> Error message: key "squirrel" not found
```

Hash tables, usage

- Perfect implementation of unordered Sets and Maps
- Hash-based Set$\langle T \rangle$:
  - Set: Collection of objects without repetition
  - Operations:
    - insert(item)
    - contains(item)
    - delete(item)
  - All in average[1] time $O(1)$ – best possible
  - Min value, successor, etc. (order-based operations) either not available or very slow at $\Theta(n)$ time
- Use cases: Set operations – remove/detect duplicates, collect objects, search for any one out of several objects

[1] Actually, almost always

Hash tables, usage

- Perfect implementation of unordered Sets and Maps
- Hash-based Set$\langle T \rangle$:
  - Set: Collection of objects without repetition
  - Operations:
    - insert(item)
    - contains(item)
    - delete(item)
  - All in average[1] time $O(1)$ – best possible
  - Min value, successor, etc. (order-based operations) either not available or very slow at $\Theta(n)$ time
- Use cases: Set operations – remove/detect duplicates, collect objects, search for any one out of several objects

[1] Actually, almost always

Hash tables, usage, cont.

- Hash-based Map⟨Key, Value⟩:
  - Map: Collection of “assignments” (key) $\mapsto$ (value)
  - Operations
    - get(key), containsKey(key)
    - put(key,value)
    - remove(key)
  - All in average[2] time $O(1)$ – best possible
  - Order-based operations (min key, successor) and value-related operations (contains value, min value) either not available or very slow at $\Theta(n)$ average-case time
- Contrast AVL tree implementation:
  - Key-related (get, put, ...) operations $O(\log n)$ time
  - Order-based operations (min key, successor) $O(\log n)$ time
  - Value-related ops $\Theta(n)$ worst case (but tweaks exist)

[2] Again, average here means almost always

Hash tables, usage, cont.

- Hash-based Map$\langle$Key, Value$\rangle$:
  - Map: Collection of "assignments" (key) $\mapsto$ (value)
  - Operations
    - get(key), containsKey(key)
    - put(key,value)
    - remove(key)
  - All in average[2] time $O(1)$ – best possible
  - Order-based operations (min key, successor) and value-related operations (contains value, min value) either not available or very slow at $\Theta(n)$ average-case time
- Contrast AVL tree implementation:
  - Key-related (get, put, ...) operations $O(\log n)$ time
  - Order-based operations (min key, successor) $O(\log n)$ time
  - Value-related ops $\Theta(n)$ worst case (but tweaks exist)

[2] Again, average here means almost always

Hash tables, usage, cont.

- Hash-based Map⟨Key, Value⟩:
  - Map: Collection of "assignments" (key) $\mapsto$ (value)
  - Operations
    - get(key), containsKey(key)
    - put(key,value)
    - remove(key)
  - All in average[2] time $O(1)$ – best possible
  - Order-based operations (min key, successor) and value-related operations (contains value, min value) either not available or very slow at $\Theta(n)$ average-case time
- Contrast AVL tree implementation:
  - Key-related (get, put, ...) operations $O(\log n)$ time
  - Order-based operations (min key, successor) $O(\log n)$ time
  - Value-related ops $\Theta(n)$ worst case (but tweaks exist)

[2] Again, average here means almost always

Hash tables, usage, cont.

- Hash-based Map$\langle$Key, Value$\rangle$:
  - Map: Collection of "assignments" (key) $\mapsto$ (value)
  - Operations
    - get(key), containsKey(key)
    - put(key,value)
    - remove(key)
  - All in average[2] time $O(1)$ – best possible
  - Order-based operations (min key, successor) and value-related operations (contains value, min value) either not available or very slow at $\Theta(n)$ average-case time
- Contrast AVL tree implementation:
  - Key-related (get, put, ...) operations $O(\log n)$ time
  - Order-based operations (min key, successor) $O(\log n)$ time
  - Value-related ops $\Theta(n)$ worst case (but tweaks exist)

[2] Again, average here means almost always

Iteration in HashMap

- Most (all?) map implementations support iteration over keys and over values
- However, the order is not specified for HashMap
- Example:
  - Insert keys ("cat"=1, "dog"=2, "elephant"=3)
  - Iterate over keys may produce e.g. the order (dog, cat, elephant)
  - Iteration over values may give, e.g., order (2,1,3)
- Recall: TreeMap supports ordered key iteration, unordered value iteration

Iteration in HashMap

- Most (all?) map implementations support iteration over keys and over values
- However, the order is not specified for HashMap
- Example:
  - Insert keys (“cat”=1, “dog”=2, “elephant”=3)
  - Iterate over keys may produce e.g. the order (dog, cat, elephant)
  - Iteration over values may give, e.g., order (2,1,3)
- Recall: TreeMap supports ordered key iteration, unordered value iteration

Iteration in HashMap

- Most (all?) map implementations support iteration over keys and over values
- However, the order is not specified for HashMap
- Example:
  - Insert keys ("cat"=1, "dog"=2, "elephant"=3)
  - Iterate over keys may produce e.g. the order (dog, cat, elephant)
  - Iteration over values may give, e.g., order (2,1,3)
- Recall: TreeMap supports ordered key iteration, unordered value iteration

Technical notes (Java)

- Names get, put, containsKey, remove as given for basic HashMap/TreeMap operations
- Syntax for iteration:
  - HashMap.keySet(): Returns a Set⟨Keys⟩ object with all keys
  - Also exists: values() for values, entrySet() for (key,value) pairs (see Java online documentation)
- Usage: Since a Set is iterable:

```
for (KeyType key : hashmap.keySet()) {
    System.out.println(key + " maps to " +
          hashmap.get(key));
}
// This will print all (key=value) pairs in the hashmap
```

Technical notes (Java)

- Names get, put, containsKey, remove as given for basic HashMap/TreeMap operations
- Syntax for iteration:
  - HashMap.keySet(): Returns a Set⟨Keys⟩ object with all keys
  - Also exists: values() for values, entrySet() for (key,value) pairs (see Java online documentation)
- Usage: Since a `Set` is iterable:

```
for (KeyType key : hashmap.keySet()) {
    System.out.println(key + " maps to " +
            hashmap.get(key));
}
// This will print all (key=value) pairs in the hashmap
```

Hash Tables – Implementation

Hash tables, plan

1. Motivating prelude: implementing Set of numbers $1, \ldots, n$ when n is small
2. Further ingredients:
   - 2.1 Implementing Set of arbitrary integers (e.g., $1, \ldots, 2^{64}$), but only n at a time: modulo and collisions
   - 2.2 Turning objects into integers: hash functions (briefly)
3. Now: Implementation focus (collision strategies, etc.)
4. Later: Hash functions, more applications

Hashtable prelude

- Question: How can we design a very efficient set, supporting insert, contains, delete in average $\Theta(1)$ time
- ...if it only needs to be able to store numbers 0, 1, ..., n-1?
- Answer: Use an array Boolean myset[n]
- myset[i] encodes whether the set contains i
- Assume fixed known size n; can use varying size with dynamic reallocation trick (when full, allocate of size 1.5n)

Hashtable prelude

- Question: How can we design a very efficient set, supporting insert, contains, delete in average $\Theta(1)$ time
- ... if it only needs to be able to store numbers 0, 1, ..., n-1?
- Answer: Use an array Boolean myset[n]
- myset[i] encodes whether the set contains i
- Assume fixed known size n; can use varying size with dynamic reallocation trick (when full, allocate of size 1.5n)

Hashtable prelude

- Question: How can we design a very efficient set, supporting insert, contains, delete in average $\Theta(1)$ time
- ... if it only needs to be able to store numbers 0, 1, ..., n-1?
- Answer: Use an array Boolean myset[n]
- myset[i] encodes whether the set contains i
- Assume fixed known size n; can use varying size with dynamic reallocation trick (when full, allocate of size 1.5n)

Boolean array, example

	Array						Action
0	1	2	3	4	5	6	
F	F	F	F	F	F	F	Initial value

Boolean array, example

Array							Action
0	1	2	3	4	5	6	
F	F	F	F	F	F	F	Initial value
F	T	F	F	F	F	F	myset.insert(1)

Boolean array, example

Array 0	1	2	3	4	5	6	Action
F	F	F	F	F	F	F	Initial value
F	T	F	F	F	F	F	myset.insert(1)
F	T	F	T	F	F	F	myset.insert(3)

Boolean array, example

Array							Action
0	1	2	3	4	5	6	
F	F	F	F	F	F	F	Initial value
F	T	F	F	F	F	F	myset.insert(1)
F	T	F	T	F	F	F	myset.insert(3) myset.contains(1) $\Rightarrow$ True

Boolean array, example

Array 0	1	2	3	4	5	6	Action
F	F	F	F	F	F	F	Initial value
F	T	F	F	F	F	F	myset.insert(1)
F	T	F	T	F	F	F	myset.insert(3) myset.contains(1) $\Rightarrow$ True
F	T	F	T	F	F	F	myset.insert(1)

Boolean array, example

0	1	2	3	4	5	6	Action
F	F	F	F	F	F	F	Initial value
F	T	F	F	F	F	F	myset.insert(1)
F	T	F	T	F	F	F	myset.insert(3) myset.contains(1) $\Rightarrow$ True
F	T	F	T	F	F	F	myset.insert(1)
F	T	F	T	T	F	F	myset.insert(4)

Boolean array, example

0	1	2	3	4	5	6	Action
F	F	F	F	F	F	F	Initial value
F	T	F	F	F	F	F	myset.insert(1)
F	T	F	T	F	F	F	myset.insert(3) myset.contains(1) $\Rightarrow$ True
F	T	F	T	F	F	F	myset.insert(1)
F	T	F	T	T	F	F	myset.insert(4)
F	F	F	T	T	F	F	myset.remove(1)

Boolean array, example

Array 0	1	2	3	4	5	6	Action
F	F	F	F	F	F	F	Initial value
F	T	F	F	F	F	F	myset.insert(1)
F	T	F	T	F	F	F	myset.insert(3) myset.contains(1) $\Rightarrow$ True
F	T	F	T	F	F	F	myset.insert(1)
F	T	F	T	T	F	F	myset.insert(4)
F	F	F	T	T	F	F	myset.remove(1) myset.contains(1) $\Rightarrow$ False

Two missing ingredients

1. We want to store arbitrary objects, not just Integers
   - Solution: Hash functions
   - Digital fingerprint:
     1.1 If `x.equals(y)`, then hash(x)==hash(y) for sure
     1.2 If not, then hash(x)$\neq$hash(y) very probably
     1.3 Should behave "as if random"
   - Many applications – more later on
2. Space usage: We will have 2^{32} possible hash codes, but will only store $n \ll 2^{32}$ objects — wasteful to allocate 2^{32} slots
   - Use table with size $p > n$ (technical note: p is a prime number)
   - Try to place numbers 0, p, 2p, ... in slot 0,
     1, p+1, 2p+1, ... in slot 1, ... (called modulo)
   - If numbers are random, then risk of "collision" is low
     (but not zero – we need collision management)

Two missing ingredients

1. We want to store arbitrary objects, not just Integers
   - Solution: Hash functions
   - Digital fingerprint:
     1.1 If `x.equals(y)`, then hash(x)==hash(y) for sure
     1.2 If not, then hash(x)$\neq$hash(y) very probably
     1.3 Should behave “as if random”
   - Many applications – more later on
2. Space usage: We will have 2^{32} possible hash codes, but will only store $n \ll 2^{32}$ objects — wasteful to allocate 2^{32} slots
   - Use table with size $p > n$ (technical note: p is a prime number)
   - Try to place numbers 0, p, 2p, ... in slot 0, 1, p+1, 2p+1, ... in slot 1, ... (called modulo)
   - If numbers are random, then risk of “collision” is low (but not zero – we need collision management)

Two missing ingredients

1. We want to store arbitrary objects, not just Integers
   - Solution: Hash functions
   - Digital fingerprint:
     1.1 If `x.equals(y)`, then hash(x)==hash(y) for sure
     1.2 If not, then hash(x)$\neq$hash(y) very probably
     1.3 Should behave "as if random"
   - Many applications – more later on
2. Space usage: We will have 2^{32} possible hash codes, but will only store $n \ll 2^{32}$ objects — wasteful to allocate 2^{32} slots
   - Use table with size $p > n$ (technical note: p is a prime number)
   - Try to place numbers 0, p, 2p, . . . in slot 0,
     1, p+1, 2p+1, . . . in slot 1, . . . (called modulo)
   - If numbers are random, then risk of "collision" is low
     (but not zero – we need collision management)

Two missing ingredients

1. We want to store arbitrary objects, not just Integers
   - Solution: Hash functions
   - Digital fingerprint:
     1.1 If `x.equals(y)`, then hash(x)==hash(y) for sure
     1.2 If not, then hash(x)$\neq$hash(y) very probably
     1.3 Should behave "as if random"
   - Many applications – more later on
2. Space usage: We will have 2^{32} possible hash codes, but will only store $n \ll 2^{32}$ objects — wasteful to allocate 2^{32} slots
   - Use table with size $p > n$ (technical note: p is a prime number)
   - Try to place numbers 0, p, 2p, ... in slot 0, 1, p+1, 2p+1, ... in slot 1, ... (called modulo)
   - If numbers are random, then risk of "collision" is low (but not zero – we need collision management)

Two missing ingredients

1. We want to store arbitrary objects, not just Integers
   - Solution: Hash functions
   - Digital fingerprint:
     1.1 If `x.equals(y)`, then hash(x)==hash(y) for sure
     1.2 If not, then hash(x)$\neq$hash(y) very probably
     1.3 Should behave "as if random"
   - Many applications – more later on
2. Space usage: We will have 2^{32} possible hash codes, but will only store $n \ll 2^{32}$ objects — wasteful to allocate 2^{32} slots
   - Use table with size $p > n$ (technical note: p is a prime number)
   - Try to place numbers 0, p, 2p, ... in slot 0,
     1, p+1, 2p+1, ... in slot 1, ... (called modulo)
   - If numbers are random, then risk of "collision" is low
     (but not zero – we need collision management)

Two missing ingredients

1. We want to store arbitrary objects, not just Integers
   - Solution: Hash functions
   - Digital fingerprint:
     1.1 If `x.equals(y)`, then hash(x)==hash(y) for sure
     1.2 If not, then hash(x)≠hash(y) very probably
     1.3 Should behave "as if random"
   - Many applications – more later on
2. Space usage: We will have 2^{32} possible hash codes, but will only store $n \ll 2^{32}$ objects — wasteful to allocate 2^{32} slots
   - Use table with size $p > n$ (technical note: p is a prime number)
   - Try to place numbers 0, p, 2p, . . . in slot 0,
     1, p+1, 2p+1, . . . in slot 1, . . . (called modulo)
   - If numbers are random, then risk of "collision" is low
     (but not zero – we need collision management)