

Sets

Lecturer: Dr Anwarul Patwary

Outline

We discuss:

- ▶ What sets are and why we need them
- ▶ Alternative representations for sets
- ▶ Java API interfaces and implementations of sets

Sets: What they are and Why we need them

- ▶ You will be familiar with idea of a set from elementary school: a set is a collection of things.
- ▶ A Set is also an interface in Java:
 - ▶ you can add and remove items from a set
 - ▶ you can check whether the set is empty.
- ▶ Sets differ from queues, stacks and lists, in that sets are **not ordered** and sets have **no duplicates**.

Representations for Sets – BitSet

- ▶ A set can be represented by an array in which each index of the array represents a possible element of the set.
- ▶ The set of all possible values is called the **universe**.
- ▶ If an element represented by i **is a member** of the set, then $a[i] = 1$ (or true).
- ▶ If the element represented by i is not a member of the set, then $a[i] = 0$ (or false).
- ▶ This representation is sometimes called the **bitset** representation of the set, and it is implemented in `java.util.BitSet`.

BitSet operations

Using a bit set we can translate set operations into efficient Java bit operations:

- ▶ insert — OR the appropriate bit with 1
- ▶ delete — AND the appropriate bit with 0
- ▶ isMember — the (boolean) value of the appropriate bit
- ▶ complement — complement of a bit vector
- ▶ union — OR two bit vectors
- ▶ intersection — AND two bit vectors
- ▶ difference — complement and intersection

BitSet performance

- ▶ The operations insert, delete and isMember are $O(1)$, while union and intersection are $O(m)$ where m is the size of the set **universe**.
- ▶ The main disadvantage of the BitSet representation occurs when the size of the universe is large.
 - ▶ For example, to represent a set of (16 bit) numbers, we would need an array of size 65,536.
- ▶ If a typical set only contains a few numbers, then the representation is not space-efficient.
- ▶ Another disadvantage is that universe needs to have a bounded size which is known in advance.

Representations for Sets – List

- ▶ An alternative representation of a set is to use a list.
- ▶ A Linked List, for example, can be used to add, delete and isMember elements of a set.
- ▶ The performance of add, delete or isMember is $O(p)$ where p is the number of elements in the set
 - ▶ This is because in the worst case we have to traverse the whole list to find the element we are looking for.
- ▶ When adding an element, we need to traverse the whole list to check that we are not adding a duplicate.
- ▶ Generating the union or intersection of 2 sets is $O(pq)$ where p and q are the sizes of the two sets.

- ▶ If the elements of the list have a natural ordering, then a list or tree can be used to store list elements in order, which enables more efficient operations.

Java Collection API for Sets

The Java Collections API provides the Set interface and three implementations:

- ▶ HashSet
- ▶ TreeSet
- ▶ SortedSet

We now examine these.

Java Set

- ▶ A collection that contains no duplicate elements.
- ▶ More formally, sets contain no pair of elements $e1$ and $e2$ such that $e1.equals(e2)$, and at most one null element.
- ▶ As implied by its name, this interface models the mathematical *set* abstraction.

Java HashSet

- ▶ This class implements the Set interface backed by a hash table.
- ▶ This class offers constant time performance for the basic operations (add, remove, contains and size) (assuming the hash function disperses the elements properly among the buckets)
- ▶ Iterating over this set requires time proportional to the sum of the HashSet instance's size (the number of elements) plus the “capacity” of the backing HashMap instance (the number of buckets).
- ▶ Thus, it's very important not to set the initial capacity too high (or the load factor too low) if iteration performance is important.

Java TreeSet

- ▶ This implementation provides guaranteed $\log(n)$ time cost for the basic operations (add, remove and contains).

Java SortedSet

- ▶ A Set that further provides a *total ordering* on its elements.
- ▶ The elements are ordered using their natural ordering or by a Comparator.
- ▶ The set's iterator will traverse the set in ascending element order.
- ▶ Several additional operations are provided to take advantage of the ordering.