

Sets

Dr. Anirban Ghosh

School of Computing
University of North Florida



What is a set?

Definition

A **set** is an unordered collection of elements, **without duplicates**. One can perceive sets as maps where the records do not have any value parts. The keys themselves are the information stored in a set.

Examples

- Maintaining a set of UNF student records
- Maintaining a set of words from a text
- Maintaining a set of towns in Florida

Common set operations

Let S and T be two sets. Then,

- **(UNION)** $S \cup T$ contains the elements from both the sets S and T
- **(INTERSECTION)** $S \cap T$ contains the elements common to both the sets S and T
- **(DIFFERENCE)** $S - T$ contains the elements which are not in T but are in S

Example

Let $S = \{22, 10, -5, 99\}$, $T = \{99, 22, -6, 82, 65\}$

- **(UNION)** $S \cup T = \{22, 10, -5, 99, -6, 82, 65\}$
- **(INTERSECTION)** $S \cap T = \{22, 99\}$
- **(DIFFERENCE)** $S - T = \{10, -5\}$

How to implement sets in Java?

- Sets are like maps where records do not have values (only keys)
- The best choices in our case are **red-black trees** and **hash-tables**
- Implementation is not too hard in our case: just ignore the value fields everywhere in our **red-black tree** and **hash-table** implementations

Set operations in coding

- **S.contains(e)**. verifies if the element e is a member of S
- **S.addAll(T)**. performs the set union operation $S \cup T$ and updates S to also include all elements of T that are not present in S ; S is replaced by: $S \cup T$
- **S.retainAll(T)**. performs the set intersection operation $S \cap T$ and updates S so that it only keeps those elements that are also elements of T ; S is replaced by: $S \cap T$
- **S.removeAll(T)**. performs the set difference operation $S - T$ and updates S by removing any of its elements that also occur in T ; S is replaced by: $S - T$

Algorithms

S.addAll(T); performing $S = S \cup T$

for (every element e in T)

add the element e to S ; // S will be duplicate-free since it is a set

S.retainAll(T); performing $S = S \cap T$

let tempS be a temporary empty set;

for (every element e in S)

if T contains e , add e to tempS;

Replace S with tempS;

S.removeAll(T); performing $S = S - T$

let tempS be a temporary empty set;

for (every element e in S)

if T does not contain e , add e to tempS;

Replace S with tempS;

See the classes `TreeSetRBTree` and `HashSetSeparateChaining`

Sets in Java

- **java.util.HashSet.** faster than TreeSet in practice; a sorted sequence of the items cannot be obtained in $O(n)$ time; in fact, while using HashSets, we do not care about the sortedness property of the set

<https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/HashSet.html>

- **java.util.TreeSet.** very fast in practice but a bit slower than HashSet in practice; a sorted sequence of the items can be easily obtained in $O(n)$ time just by iterating the set; TreeSets are **sorted sets**

<https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/TreeSet.html>