

This document describes¹ the Application Programming Interface (API) for the data structures and algorithms discussed in the book *Algorithms* [↗](#) by Robert Sedgewick and Kevin Wayne. The corresponding libraries and data types are part of a package called `dsa`.

Object-oriented Programming

☰ Counter implements Comparable<Counter>	
<code>Counter(String id)</code>	constructs a counter given its <code>id</code>
<code>void increment()</code>	increments this counter by 1
<code>int tally()</code>	returns the current value of this counter
<code>void reset()</code>	resets this counter to zero
<code>boolean equals(Object other)</code>	returns <code>true</code> if this counter and <code>other</code> have the same tally, and <code>false</code> otherwise
<code>String toString()</code>	returns a string representation of this counter
<code>int compareTo(Counter other)</code>	returns a comparison of this counter with <code>other</code> by their tally

☰ Date implements Comparable<Date>	
<code>Date(int month, int day, int year)</code>	constructs a date from <code>month</code> , <code>day</code> , and <code>year</code>
<code>Date(String s)</code>	constructs a date from a string <code>s</code> of the form “MM/DD/YYYY”
<code>int month()</code>	returns the month (an integer between 1 and 12)
<code>int day()</code>	returns the day (an integer between 1 and 31)
<code>int year()</code>	returns the year
<code>Date next()</code>	returns the next date in the calendar
<code>boolean isBefore(Date other)</code>	returns <code>true</code> if this date is before <code>other</code> , and <code>false</code> otherwise
<code>boolean isAfter(Date other)</code>	returns <code>true</code> if this date is after <code>other</code> , and <code>false</code> otherwise
<code>boolean equals(Object other)</code>	returns <code>true</code> if this date is the same as <code>other</code> , and <code>false</code> otherwise
<code>int hashCode()</code>	returns a hash code for this date
<code>String toString()</code>	returns a string representation of this date
<code>int compareTo(Date other)</code>	returns a chronological comparison of this date with <code>other</code>

☰ Transaction implements Comparable<Transaction>	
<code>Transaction(String name, Date date, double amount)</code>	constructs a transaction from a <code>name</code> , <code>date</code> , and <code>amount</code>
<code>Transaction(String s)</code>	constructs a transaction from a string <code>s</code> of the form “name date amount”
<code>String name()</code>	returns the name of the person involved in this transaction
<code>Date date()</code>	returns the date of this transaction
<code>double amount()</code>	returns the amount of this transaction
<code>int hashCode()</code>	returns a hash code for this transaction
<code>String toString()</code>	returns a string representation of this transaction
<code>int compareTo(Transaction other)</code>	returns a comparison of this transaction with <code>other</code> by amount
<code>static Comparator<Transaction> nameOrder()</code>	returns a comparator for comparing two transactions by name
<code>static Comparator<Transaction> dateOrder()</code>	returns a comparator for comparing two transactions by date

¹A data type name in italics denotes an interface.

Algorithms and Data Structures

LinearSearch	
<code>static int indexOf(Object[] a, Object key)</code>	returns the index of <code>key</code> in the array <code>a</code> , or <code>-1</code>
<code>static <T> int indexOf(T[] a, T key, Comparator<T> c)</code>	returns the index of <code>key</code> in the array <code>a</code> , or <code>-1</code> (comparisons are made using the comparator <code>c</code>)
<code>static int indexOf(int[] a, int key)</code>	returns the index of <code>key</code> in the array <code>a</code> , or <code>-1</code>
<code>static int indexOf(double[] a, double key)</code>	returns the index of <code>key</code> in the array <code>a</code> , or <code>-1</code>

BinarySearch	
<code>static <T extends Comparable<T>> int indexOf(T[] a, T key)</code>	returns the index of <code>key</code> in the sorted array <code>a</code> , or <code>-1</code>
<code>static int indexOf(int[] a, int key)</code>	returns the index of <code>key</code> in the sorted array <code>a</code> , or <code>-1</code>
<code>static int indexOf(double[] a, double key)</code>	returns the index of <code>key</code> in the sorted array <code>a</code> , or <code>-1</code>

Bag<Item> extends Iterable<T>	
<code>boolean isEmpty()</code>	returns <code>true</code> if this bag is empty, and <code>false</code> otherwise
<code>int size()</code>	returns the number of items in this bag
<code>void add(T item)</code>	adds <code>item</code> to this bag
<code>String toString()</code>	returns a string representation of this bag
<code>Iterator<T> iterator()</code>	returns an iterator to iterate over the items in this bag

LinkedBag<T> implements Bag<T>	
<code>LinkedBag()</code>	constructs an empty bag
<code>boolean isEmpty()</code>	returns <code>true</code> if this bag is empty, and <code>false</code> otherwise
<code>int size()</code>	returns the number of items in this bag
<code>void add(T item)</code>	adds <code>item</code> to this bag
<code>String toString()</code>	returns a string representation of this bag
<code>Iterator<T> iterator()</code>	returns an iterator to iterate over the items in this bag

ResizingArrayBag<T> implements Bag<T>	
<code>ResizingArrayBag()</code>	constructs an empty bag
<code>boolean isEmpty()</code>	returns <code>true</code> if this bag is empty, and <code>false</code> otherwise
<code>int size()</code>	returns the number of items in this bag
<code>void add(T item)</code>	adds <code>item</code> to this bag
<code>String toString()</code>	returns a string representation of this bag
<code>Iterator<T> iterator()</code>	returns an iterator to iterate over the items in this bag

Queue<Item> extends Iterable<Item>	
<code>boolean isEmpty()</code>	returns <code>true</code> if this queue is empty, and <code>false</code> otherwise
<code>int size()</code>	returns the number of items in this queue
<code>void enqueue(Item item)</code>	adds <code>item</code> to the end of this queue
<code>Item peek()</code>	returns the item at the front of this queue
<code>Item dequeue()</code>	removes and returns the item at the front of this queue
<code>String toString()</code>	returns a string representation of this queue
<code>Iterator<Item> iterator()</code>	returns an iterator to iterate over the items in this queue in FIFO order

☰ `LinkedListQueue<Item>` implements `Queue<Item>`

<code>LinkedListQueue()</code>	constructs an empty queue
<code>boolean isEmpty()</code>	returns <code>true</code> if this queue is empty, and <code>false</code> otherwise
<code>int size()</code>	returns the number of items in this queue
<code>void enqueue(Item item)</code>	adds <code>item</code> to the end of this queue
<code>Item peek()</code>	returns the item at the front of this queue
<code>Item dequeue()</code>	removes and returns the item at the front of this queue
<code>String toString()</code>	returns a string representation of this queue
<code>Iterator<Item> iterator()</code>	returns an iterator to iterate over the items in this queue in FIFO order

☰ `ResizingArrayQueue<Item>` implements `Queue<Item>`

<code>ResizingArrayQueue()</code>	constructs an empty queue
<code>boolean isEmpty()</code>	returns <code>true</code> if this queue is empty, and <code>false</code> otherwise
<code>int size()</code>	returns the number of items in this queue
<code>void enqueue(Item item)</code>	adds <code>item</code> to the end of this queue
<code>Item peek()</code>	returns the item at the front of this queue
<code>Item dequeue()</code>	removes and returns the item at the front of this queue
<code>String toString()</code>	returns a string representation of this queue
<code>Iterator<Item> iterator()</code>	returns an iterator to iterate over the items in this queue in FIFO order

☰ `Stack<Item>` extends `Iterable<Item>`

<code>boolean isEmpty()</code>	returns <code>true</code> if this stack is empty, and <code>false</code> otherwise
<code>int size()</code>	returns the number of items in this stack
<code>void push(Item item)</code>	adds <code>item</code> to the top of this stack
<code>Item peek()</code>	returns the item at the top of this stack
<code>Item pop()</code>	removes and returns the item at the top of this stack
<code>String toString()</code>	returns a string representation of this stack
<code>Iterator<Item> iterator()</code>	returns an iterator to iterate over the items in this stack in LIFO order

☰ `LinkedListStack<Item>` implements `Stack<Item>`

<code>LinkedListStack()</code>	constructs an empty stack
<code>boolean isEmpty()</code>	returns <code>true</code> if this stack is empty, and <code>false</code> otherwise
<code>int size()</code>	returns the number of items in this stack
<code>void push(Item item)</code>	adds <code>item</code> to the top of this stack
<code>Item peek()</code>	returns the item at the top of this stack
<code>Item pop()</code>	removes and returns the item at the top of this stack
<code>String toString()</code>	returns a string representation of this stack
<code>Iterator<Item> iterator()</code>	returns an iterator to iterate over the items in this stack in LIFO order

☰ `ResizingArrayStack<Item>` implements `Stack<Item>`

<code>ResizingArrayStack()</code>	constructs an empty stack
<code>boolean isEmpty()</code>	returns <code>true</code> if this stack is empty, and <code>false</code> otherwise
<code>int size()</code>	returns the number of items in this stack
<code>void push(Item item)</code>	adds <code>item</code> to the top of this stack
<code>Item peek()</code>	returns the item at the top of this stack
<code>Item pop()</code>	removes and returns the item at the top of this stack
<code>String toString()</code>	returns a string representation of this stack
<code>Iterator<Item> iterator()</code>	returns an iterator to iterate over the items in this stack in LIFO order

☰ UF	
<code>int find(int p)</code>	returns the canonical site of the component containing site <code>p</code>
<code>int count()</code>	returns the number of components
<code>boolean connected(int p, int q)</code>	returns <code>true</code> if sites <code>p</code> and <code>q</code> belong to the same component, and <code>false</code> otherwise
<code>void union(int p, int q)</code>	connects sites <code>p</code> and <code>q</code>

☰ QuickFindUF implements UF	
<code>QuickFindUF(int n)</code>	constructs an empty union-find data structure with <code>n</code> sites
<code>int find(int p)</code>	returns the canonical site of the component containing site <code>p</code>
<code>int count()</code>	returns the number of components
<code>boolean connected(int p, int q)</code>	returns <code>true</code> if sites <code>p</code> and <code>q</code> belong to the same component, and <code>false</code> otherwise
<code>void union(int p, int q)</code>	connects sites <code>p</code> and <code>q</code>

☰ QuickUnionUF implements UF	
<code>QuickUnionUF(int n)</code>	constructs an empty union-find data structure with <code>n</code> sites
<code>int find(int p)</code>	returns the canonical site of the component containing site <code>p</code>
<code>int count()</code>	returns the number of components
<code>boolean connected(int p, int q)</code>	returns <code>true</code> if sites <code>p</code> and <code>q</code> belong to the same component, and <code>false</code> otherwise
<code>void union(int p, int q)</code>	connects sites <code>p</code> and <code>q</code>

☰ WeightedQuickUnionUF implements UF	
<code>WeightedQuickUnionUF(int n)</code>	constructs an empty union-find data structure with <code>n</code> sites
<code>int find(int p)</code>	returns the canonical site of the component containing site <code>p</code>
<code>int count()</code>	returns the number of components
<code>boolean connected(int p, int q)</code>	returns <code>true</code> if sites <code>p</code> and <code>q</code> belong to the same component, and <code>false</code> otherwise
<code>void union(int p, int q)</code>	connects sites <code>p</code> and <code>q</code>

Sorting

☰ Bubble, Selection, Insertion, Shell, Merge, Quick, Quick3way, Heap	
<code>static <T extends Comparable<T>> void sort(T[] a)</code>	sorts the array <code>a</code> according to the natural order of its objects
<code>static <T> void sort(T[] a, Comparator<T> c)</code>	sorts the array <code>a</code> according to the order induced by the comparator <code>c</code>
<code>static void sort(int[] a)</code>	sorts the array <code>a</code>
<code>static void sort(double[] a)</code>	sorts the array <code>a</code>

☰ MinPQ<Key> implements Iterable<Key>	
MinPQ()	constructs an empty minPQ
MinPQ(Comparator<Key> c)	constructs an empty minPQ with the given comparator
MinPQ(int capacity)	constructs an empty minPQ with the given capacity
MinPQ(int capacity, Comparator<Key> c)	constructs an empty minPQ with the given capacity and comparator
boolean isEmpty()	returns <code>true</code> if this minPQ is empty, and <code>false</code> otherwise
int size()	returns the number of keys in this minPQ
void insert(Key key)	adds <code>key</code> to this minPQ
Key min()	returns the smallest key in this minPQ
Key deleteMin()	removes and returns the smallest key in this minPQ
String toString()	returns a string representation of this minPQ
Iterator<Key> iterator()	returns an iterator to iterate over the keys in this minPQ in ascending order

☰ MaxPQ<Key> implements Iterable<Key>	
MaxPQ()	constructs an empty maxPQ
MaxPQ(Comparator<Key> c)	constructs an empty maxPQ with the given comparator
MaxPQ(int capacity)	constructs an empty maxPQ with the given capacity
MaxPQ(int capacity, Comparator<Key> c)	constructs an empty maxPQ with the given capacity and comparator
boolean isEmpty()	returns <code>true</code> if this maxPQ is empty, and <code>false</code> otherwise
int size()	returns the number of keys in this maxPQ
void insert(Key key)	adds <code>key</code> to this maxPQ
Key max()	returns the largest key in this maxPQ
Key deleteMax()	removes and returns the largest key in this maxPQ
String toString()	returns a string representation of this maxPQ
Iterator<Key> iterator()	returns an iterator to iterate over the keys in this maxPQ in descending order

☰ IndexMinPQ<Key extends Comparable<Key>> implements Iterable<Key>	
IndexMinPQ(int maxN)	constructs an empty indexMinPQ with indices from the interval $[0, \text{maxN})$
boolean isEmpty()	returns <code>true</code> if this indexMinPQ is empty, and <code>false</code> otherwise
int size()	returns the number of keys in this indexMinPQ
void insert(int i, Key key)	associates <code>key</code> with index <code>i</code> in this indexMinPQ
void change(int i, Key key)	changes the key associated with index <code>i</code> to <code>key</code> in this indexMinPQ
boolean contains(int i)	returns <code>true</code> if <code>i</code> is an index in this indexMinPQ, and <code>false</code> otherwise
int minIndex()	returns the index associated with the smallest key in this indexMinPQ
Key minKey()	returns the smallest key in this indexMinPQ
Key keyOf(int i)	returns the key associated with index <code>i</code> in this indexMinPQ
int deleteMin()	removes the smallest key from this indexMinPQ and returns its associated index
void delete(int i)	removes the key associated with index <code>i</code> in this indexMinPQ
String toString()	returns a string representation of this indexMinPQ
Iterator<Integer> iterator()	returns an iterator to iterate over the indices in this indexMinPQ in ascending order of the associated keys

IndexMaxPQ<Key> extends Comparable<Key>> implements Iterable<Key>	
IndexMaxPQ(int maxN)	constructs an empty indexMaxPQ with indices from the interval [0, maxN)
boolean isEmpty()	returns true if this indexMaxPQ is empty, and false otherwise
int size()	returns the number of keys in this indexMaxPQ
void insert(int i, Key key)	associates key with index i in this indexMaxPQ
void change(int i, Key key)	changes the key associated with index i to key in this indexMaxPQ
boolean contains(int i)	returns true if i is an index in this indexMaxPQ, and false otherwise
int maxIndex()	returns the index associated with the largest key in this indexMaxPQ
Key maxKey()	returns the largest key in this indexMaxPQ
Key keyOf(int i)	returns the key associated with index i in this indexMaxPQ
int deleteMax()	removes the largest key from this indexMaxPQ and returns its associated index
void delete(int i)	removes the key associated with index i in this indexMaxPQ
String toString()	returns a string representation of this indexMaxPQ
Iterator<Integer> iterator()	returns an iterator to iterate over the indices in this indexMaxPQ in descending order of the associated keys

Inversions	
static long count(Comparable[] a)	returns the number of inversions in the array a according to the natural order of its objects
static long count(Object[] a, Comparator c)	returns the number of inversions in the array a according to the order induced by the comparator c
static long count(int a[])	returns the number of inversions in the array a
static long count(double a[])	returns the number of inversions in the array a

Searching

BasicST<K, V>	
boolean isEmpty()	returns true if this symbol table is empty, and false otherwise
int size()	returns the number of key-value pairs in this symbol table
void put(K key, V value)	inserts the key and value pair into this symbol table
V get(K key)	returns the value associated with key in this symbol table, or null
boolean contains(K key)	returns true if this symbol table contains key, and false otherwise
void delete(K key)	deletes key and the associated value from this symbol table
Iterable<K> keys()	returns all the keys in this symbol table
String toString()	returns a string representation of this symbol table

☰ *OrderedST*<K extends Comparable<K>, V>

<code>boolean isEmpty()</code>	returns <code>true</code> if this symbol table is empty, and <code>false</code> otherwise
<code>int size()</code>	returns the number of key-value pairs in this symbol table
<code>void put(K key, V value)</code>	inserts the <code>key</code> and <code>value</code> pair into this symbol table
<code>V get(K key)</code>	returns the value associated with <code>key</code> in this symbol table, or <code>null</code>
<code>boolean contains(K key)</code>	returns <code>true</code> if this symbol table contains <code>key</code> , and <code>false</code> otherwise
<code>void delete(K key)</code>	deletes <code>key</code> and the associated value from this symbol table
<code>Iterable<K> keys()</code>	returns all the keys in this symbol table in sorted order
<code>K min()</code>	returns the smallest key in this symbol table
<code>K max()</code>	returns the largest key in this symbol table
<code>void deleteMin()</code>	deletes the smallest key and the associated value from this symbol table
<code>void deleteMax()</code>	deletes the largest key and the associated value from this symbol table
<code>K floor(K key)</code>	returns the largest key in this symbol table that is smaller than or equal to <code>key</code>
<code>K ceiling(K key)</code>	returns the smallest key in this symbol table that is greater than or equal to <code>key</code>
<code>int rank(K key)</code>	returns the number of keys in this symbol table that are strictly smaller than <code>key</code>
<code>K select(int k)</code>	returns the key in this symbol table with the rank <code>k</code>
<code>int size(K lo, K hi)</code>	returns the number of keys in this symbol table that are in the interval <code>[lo, hi]</code>
<code>Iterable<K> keys(K lo, K hi)</code>	returns the keys in this symbol table that are in the interval <code>[lo, hi]</code> in sorted order
<code>String toString()</code>	returns a string representation of this symbol table

☰ *LinearSearchST*<K, V> implements *BasicST*<K, V>

<code>LinearSearchST()</code>	constructs an empty symbol table
<code>boolean isEmpty()</code>	returns <code>true</code> if this symbol table is empty, and <code>false</code> otherwise
<code>int size()</code>	returns the number of key-value pairs in this symbol table
<code>void put(K key, V value)</code>	inserts the <code>key</code> and <code>value</code> pair into this symbol table
<code>V get(K key)</code>	returns the value associated with <code>key</code> in this symbol table, or <code>null</code>
<code>boolean contains(K key)</code>	returns <code>true</code> if this symbol table contains <code>key</code> , and <code>false</code> otherwise
<code>void delete(K key)</code>	deletes <code>key</code> and the associated value from this symbol table
<code>Iterable<K> keys()</code>	returns all the keys in this symbol table
<code>String toString()</code>	returns a string representation of this symbol table

☰ BinarySearchST<K extends Comparable<K>, V> implements OrderedST<K, V>

BinarySearchST()	constructs an empty symbol table
boolean isEmpty()	returns <code>true</code> if this symbol table is empty, and <code>false</code> otherwise
int size()	returns the number of key-value pairs in this symbol table
void put(K key, V value)	inserts the <code>key</code> and <code>value</code> pair into this symbol table
V get(K key)	returns the value associated with <code>key</code> in this symbol table, or <code>null</code>
boolean contains(K key)	returns <code>true</code> if this symbol table contains <code>key</code> , and <code>false</code> otherwise
void delete(K key)	deletes <code>key</code> and the associated value from this symbol table
Iterable<K> keys()	returns all the keys in this symbol table in sorted order
K min()	returns the smallest key in this symbol table
K max()	returns the largest key in this symbol table
void deleteMin()	deletes the smallest key and the associated value from this symbol table
void deleteMax()	deletes the largest key and the associated value from this symbol table
K floor(K key)	returns the largest key in this symbol table that is smaller than or equal to <code>key</code>
K ceiling(K key)	returns the smallest key in this symbol table that is greater than or equal to <code>key</code>
int rank(K key)	returns the number of keys in this symbol table that are strictly smaller than <code>key</code>
K select(int k)	returns the key in this symbol table with the rank <code>k</code>
int size(K lo, K hi)	returns the number of keys in this symbol table that are in the interval <code>[lo, hi]</code>
Iterable<K> keys(K lo, K hi)	returns the keys in this symbol table that are in the interval <code>[lo, hi]</code> in sorted order
String toString()	returns a string representation of this symbol table

☰ SeparateChainingHashST<K, V> implements BasicST<K, V>

SeparateChainingHashST()	constructs an empty symbol table
boolean isEmpty()	returns <code>true</code> if this symbol table is empty, and <code>false</code> otherwise
int size()	returns the number of key-value pairs in this symbol table
void put(K key, V value)	inserts the <code>key</code> and <code>value</code> pair into this symbol table
V get(K key)	returns the value associated with <code>key</code> in this symbol table, or <code>null</code>
boolean contains(K key)	returns <code>true</code> if this symbol table contains <code>key</code> , and <code>false</code> otherwise
void delete(K key)	deletes <code>key</code> and the associated value from this symbol table
Iterable<K> keys()	returns all the keys in this symbol table
String toString()	returns a string representation of this symbol table

BinarySearchTreeST<K extends Comparable<K>, V> implements OrderedST<K, V>	
BinarySearchTreeST()	constructs an empty symbol table
boolean isEmpty()	returns true if this symbol table is empty, and false otherwise
int size()	returns the number of key-value pairs in this symbol table
void put(K key, V value)	inserts the key and value pair into this symbol table
V get(K key)	returns the value associated with key in this symbol table, or null
boolean contains(K key)	returns true if this symbol table contains key, and false otherwise
void delete(K key)	deletes key and the associated value from this symbol table
Iterable<K> keys()	returns all the keys in this symbol table in sorted order
K min()	returns the smallest key in this symbol table
K max()	returns the largest key in this symbol table
void deleteMin()	deletes the smallest key and the associated value from this symbol table
void deleteMax()	deletes the largest key and the associated value from this symbol table
K floor(K key)	returns the largest key in this symbol table that is smaller than or equal to key
K ceiling(K key)	returns the smallest key in this symbol table that is greater than or equal to key
int rank(K key)	returns the number of keys in this symbol table that are strictly smaller than key
K select(int k)	returns the key in this symbol table with the rank k
int size(K lo, K hi)	returns the number of keys in this symbol table that are in the interval [lo, hi]
Iterable<K> keys(K lo, K hi)	returns the keys in this symbol table that are in the interval [lo, hi] in sorted order
String toString()	returns a string representation of this symbol table
Iterable<K> preOrder()	returns all the keys from this symbol table in “pre” order
Iterable<K> inOrder()	returns all the keys from this symbol table in “in” order
Iterable<K> postOrder()	returns all the keys from this symbol table in “post” order

RedBlackBinarySearchTreeST<K extends Comparable<K>, V> implements OrderedST<K, V>	
RedBlackBinarySearchTreeST()	constructs an empty symbol table
boolean isEmpty()	returns true if this symbol table is empty, and false otherwise
int size()	returns the number of key-value pairs in this symbol table
void put(K key, V value)	inserts the key and value pair into this symbol table
V get(K key)	returns the value associated with key in this symbol table, or null
boolean contains(K key)	returns true if this symbol table contains key, and false otherwise
void delete(K key)	deletes key and the associated value from this symbol table
Iterable<K> keys()	returns all the keys in this symbol table in sorted order
K min()	returns the smallest key in this symbol table
K max()	returns the largest key in this symbol table
void deleteMin()	deletes the smallest key and the associated value from this symbol table
void deleteMax()	deletes the largest key and the associated value from this symbol table
K floor(K key)	returns the largest key in this symbol table that is smaller than or equal to key
K ceiling(K key)	returns the smallest key in this symbol table that is greater than or equal to key
int rank(K key)	returns the number of keys in this symbol table that are strictly smaller than key
K select(int k)	returns the key in this symbol table with the rank k
int size(K lo, K hi)	returns the number of keys in this symbol table that are in the interval [lo, hi]
Iterable<K> keys(K lo, K hi)	returns the keys in this symbol table that are in the interval [lo, hi] in sorted order
String toString()	returns a string representation of this symbol table
Iterable<K> preOrder()	returns all the keys from this symbol table in “pre” order
Iterable<K> inOrder()	returns all the keys from this symbol table in “in” order
Iterable<K> postOrder()	returns all the keys from this symbol table in “post” order

☰ Set<K extends Comparable<K>> implements Iterable<K>	
Set()	constructs an empty set
boolean isEmpty()	returns true if this set is empty, and false otherwise
int size()	returns the number of keys in this set
void add(K key)	adds key to this set, if it is not already present
boolean contains(K key)	returns true if this set contains key, and false otherwise
void delete(K key)	deletes key from this set
Iterator<K> iterator()	returns an iterator to iterate over the keys in this set in sorted order
String toString()	returns a string representation of this set

☰ SparseVector	
SparseVector(int n)	constructs an n-dimensional zero vector
int dimension()	returns the dimension of this vector
int size()	returns the number of nonzero entries in this vector
void put(int i, double value)	sets the ith component of this vector to value
double get(int i)	returns the ith component of this vector
SparseVector plus(SparseVector other)	returns the sum of this vector and other
SparseVector scale(double alpha)	returns the scalar-vector product of this vector and alpha
double dot(SparseVector other)	returns the dot product of this vector and other
double magnitude()	returns the magnitude of this vector
String toString()	returns a string representation of this vector

☰ SparseMatrix	
SparseMatrix(int m, int n)	constructs an m x n dimensional zero matrix
int nRows()	returns the number of rows in this matrix
int nCols()	returns the number of columns in this matrix
int size()	returns the number of nonzero entries in this matrix
void put(int i, int j, double value)	sets the entry at row i and column j in this matrix to value
double get(int i, int j)	returns the entry in this matrix at row i and column j
SparseMatrix plus(SparseMatrix other)	returns the sum of this matrix and other
SparseVector times(SparseVector x)	returns the product of this matrix and the vector x
String toString()	returns a string representation of this matrix

Graphs

☰ Graph	
Graph(int V)	constructs an empty graph with v vertices and 0 edges
Graph(In in)	constructs a graph from the input stream in
int V()	returns the number of vertices in this graph
int E()	returns the number of edges in this graph
void addEdge(int v, int w)	adds an undirected edge between vertices v and w in this graph
Iterable<Integer> adj(int v)	returns the vertices adjacent to vertex v in this graph
int degree(int v)	returns the degree of vertex v in this graph
String toString()	returns a string representation of this graph

Paths	
<code>boolean hasPathTo(int v)</code>	returns <code>true</code> if there is a path between a designated source vertex and vertex <code>v</code> , and <code>false</code> otherwise
<code>Iterable<Integer> pathTo(int v)</code>	returns a path between a designated source vertex and vertex <code>v</code> , OR <code>null</code>
<code>double distTo(int v)</code>	returns the shortest distance between a designated source vertex and vertex <code>v</code> , or ∞

DFSPaths implements Paths	
<code>DFSPaths(Graph G, int s)</code>	computes paths between source vertex <code>s</code> and every other vertex in the graph <code>G</code>
<code>boolean hasPathTo(int v)</code>	returns <code>true</code> if there is a path between a designated source vertex and vertex <code>v</code> , and <code>false</code> otherwise
<code>Iterable<Integer> pathTo(int v)</code>	returns a path between a designated source vertex and vertex <code>v</code> , OR <code>null</code>

BFSPaths implements Paths	
<code>BFSPaths(Graph G, int s)</code>	computes shortest paths between source vertex <code>s</code> and every other vertex in the graph <code>G</code>
<code>boolean hasPathTo(int v)</code>	returns <code>true</code> if there is a path between a designated source vertex and vertex <code>v</code> , and <code>false</code> otherwise
<code>Iterable<Integer> pathTo(int v)</code>	returns a path between a designated source vertex and vertex <code>v</code> , OR <code>null</code>
<code>double distTo(int v)</code>	returns the shortest distance between a designated source vertex and vertex <code>v</code> , or ∞

SymbolGraph	
<code>SymbolGraph(In in, String delim)</code>	constructs a symbol graph from the input stream <code>in</code> and using <code>delim</code> as the delimiter
<code>boolean contains(String s)</code>	returns <code>true</code> if this symbol graph contains vertex <code>s</code> , and <code>false</code> otherwise
<code>int indexOf(String s)</code>	returns the integer associated with the vertex <code>s</code> in this symbol graph
<code>String nameOf(int v)</code>	returns the name of the vertex associated with the integer <code>v</code> in this symbol graph
<code>Graph graph()</code>	returns the graph associated with this symbol graph
<code>String toString()</code>	returns a string representation of this symbol graph

DiGraph	
<code>DiGraph(int v)</code>	constructs an empty digraph with <code>v</code> vertices and 0 edges
<code>DiGraph(In in)</code>	constructs a digraph from the input stream <code>in</code>
<code>int V()</code>	returns the number of vertices in this digraph
<code>int E()</code>	returns the number of edges in this digraph
<code>void addEdge(int v, int w)</code>	adds the directed edge <code>v->w</code> to this digraph
<code>Iterable<Integer> adj(int v)</code>	returns the vertices adjacent from vertex <code>v</code> in this digraph
<code>int outDegree(int v)</code>	returns the out-degree of vertex <code>v</code> in this digraph
<code>int inDegree(int v)</code>	returns the in-degree of vertex <code>v</code> in this digraph
<code>String toString()</code>	returns a string representation of this digraph

DFSPaths implements Paths	
<code>DFSPaths(DiGraph G, int s)</code>	computes paths from source vertex <code>s</code> to every other vertex in the digraph <code>G</code>
<code>boolean hasPathTo(int v)</code>	returns <code>true</code> if there is a path between a designated source vertex and vertex <code>v</code> , and <code>false</code> otherwise
<code>Iterable<Integer> pathTo(int v)</code>	returns a path between a designated source vertex and vertex <code>v</code> , OR <code>null</code>

BFSDiPaths implements Paths

<code>BFSDiPaths(DiGraph G, int s)</code>	computes shortest paths from source vertex <code>s</code> to every other vertex in the digraph <code>G</code>
<code>boolean hasPathTo(int v)</code>	returns <code>true</code> if there is a path between a designated source vertex and vertex <code>v</code> , and <code>false</code> otherwise
<code>Iterable<Integer> pathTo(int v)</code>	returns a path between a designated source vertex and vertex <code>v</code> , or <code>null</code>
<code>double distTo(int v)</code>	returns the shortest distance between a designated source vertex and vertex <code>v</code> , or ∞

SymbolDiGraph

<code>SymbolDiGraph(In in, String delim)</code>	constructs a symbol digraph from the input stream <code>in</code> and using <code>delim</code> as the delimiter
<code>boolean contains(String s)</code>	returns <code>true</code> if this symbol digraph contains vertex <code>s</code> , and <code>false</code> otherwise
<code>int indexOf(String s)</code>	returns the integer associated with the vertex <code>s</code> in this symbol digraph
<code>String nameOf(int v)</code>	returns the name of the vertex associated with the integer <code>v</code> in this symbol digraph
<code>DiGraph diGraph()</code>	returns the digraph associated with this symbol digraph
<code>String toString()</code>	returns a string representation of this symbol digraph

DiCycle

<code>DiCycle(DiGraph G)</code>	determines whether the digraph <code>G</code> has a directed cycle and, if so, finds such a cycle
<code>boolean hasCycle()</code>	returns <code>true</code> if a directed cycle was detected, and <code>false</code> otherwise
<code>Iterable<Integer> cycle()</code>	returns a directed cycle, or <code>null</code>

DFSOrders

<code>DFSOrders(DiGraph G)</code>	determines depth-first orders (pre, post, and reverse post) for the digraph <code>G</code>
<code>int pre(int v)</code>	returns the pre-order number of vertex <code>v</code>
<code>int post(int v)</code>	returns the post-order number of vertex <code>v</code>
<code>Iterable<Integer> pre()</code>	returns the vertices in pre-order
<code>Iterable<Integer> post()</code>	returns the vertices in post-order
<code>Iterable<Integer> reversePost()</code>	returns the vertices in reverse post-order

Topological

<code>Topological(DiGraph G)</code>	determines whether the digraph <code>G</code> has a topological order and, if so, finds such an order
<code>boolean hasOrder()</code>	returns <code>true</code> if there exists a topological order, and <code>false</code> otherwise
<code>Iterable<Integer> order()</code>	returns a topological order, or <code>null</code>

Edge implements Comparable<Edge>

<code>Edge(int v, int w, double weight)</code>	constructs an edge between vertices <code>v</code> and <code>w</code> of the given <code>weight</code>
<code>int either()</code>	returns one endpoint of this edge
<code>int other(int v)</code>	returns the endpoint of this edge that is different from vertex <code>v</code>
<code>double weight()</code>	returns the weight of this edge
<code>String toString()</code>	returns a string representation of this edge
<code>int compareTo(Edge other)</code>	returns a comparison of this edge with <code>other</code> by their weights

EdgeWeightedGraph	
EdgeWeightedGraph(int V)	constructs an empty edge-weighted graph with v vertices and 0 edges
EdgeWeightedGraph(In in)	constructs an edge-weighted graph from the input stream in
int V()	returns the number of vertices in this edge-weighted graph
int E()	returns the number of edges in this edge-weighted graph
void addEdge(Edge e)	adds an edge e to this edge-weighted graph
Iterable<Integer> adj(int v)	returns the edges incident on vertex v in this edge-weighted graph
int degree(int v)	returns the degree of vertex v in this edge-weighted graph
Iterable<Edge> edges()	returns all the edges in this edge-weighted graph
String toString()	returns a string representation of this edge-weighted graph

Kruskal	
Kruskal(EdgeWeightedGraph G)	determines the minimum spanning tree (MST) of the edge-weighted graph G
Iterable<Edge> edges()	returns the edges in the MST
double weight()	returns the sum of the edge weights in the MST

DiEdge	
DiEdge(int v, int w, double weight)	constructs a directed edge from vertex v to vertex w of the given $weight$
int from()	returns the tail vertex of this directed edge
int to()	returns the head vertex of this directed edge
double weight()	returns the weight of this directed edge
String toString()	returns a string representation of this directed edge

EdgeWeightedDiGraph	
EdgeWeightedDiGraph(int V)	constructs an empty edge-weighted digraph with v vertices and 0 edges
EdgeWeightedDiGraph(In in)	constructs an edge-weighted digraph from the input stream in
int V()	returns the number of vertices in this edge-weighted digraph
int E()	returns the number of edges in this edge-weighted digraph
void addEdge(DiEdge e)	adds a directed edge e to this edge-weighted digraph
Iterable<Integer> adj(int v)	returns the directed edges incident from vertex v in this edge-weighted digraph
int outDegree(int v)	returns the out-degree of vertex v in this edge-weighted digraph
int inDegree(int v)	returns the in-degree of vertex v in this edge-weighted digraph
Iterable<DiEdge> edges()	returns all the directed edges in this edge-weighted digraph
String toString()	returns a string representation of this edge-weighted digraph

Dijkstra implements Paths	
Dijkstra(DiGraph G, int s)	determines the shortest paths from the source vertex s to every other vertex in the edge-weighted digraph G
boolean hasPathTo(int v)	returns <code>true</code> if there is a path between a designated source vertex and vertex v , and <code>false</code> otherwise
Iterable<Integer> pathTo(int v)	returns a path between a designated source vertex and vertex v , or <code>null</code>
double distTo(int v)	returns the shortest distance between a designated source vertex and vertex v , or ∞

Strings

Alphabet	
<code>static Alphabet BINARY</code>	the binary alphabet $\{0, 1\}$
<code>static Alphabet OCTAL</code>	the octal alphabet $\{0, 1, 2, 3, 4, 5, 6, 7\}$
<code>static Alphabet DECIMAL</code>	the decimal alphabet $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$
<code>static Alphabet HEXADECIMAL</code>	the hexadecimal alphabet $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F\}$
<code>static Alphabet DNA</code>	the DNA alphabet $\{A, C, G, T\}$
<code>static Alphabet LOWERCASE</code>	the lowercase alphabet $\{a, b, c, \dots, z\}$
<code>static Alphabet UPPERCASE</code>	the uppercase alphabet $\{A, B, C, \dots, Z\}$
<code>static Alphabet PROTEIN</code>	the protein alphabet $\{A, C, D, E, F, G, H, I, K, L, M, N, P, Q, R, S, T, V, W, Y\}$
<code>static Alphabet BASE64</code>	the base-64 alphabet (64 characters)
<code>static Alphabet ASCII</code>	the ASCII alphabet (0 – 127)
<code>static Alphabet EXTENDED_ASCII</code>	the extended ASCII alphabet (0 – 255)
<code>static Alphabet UNICODE16</code>	the Unicode 16 alphabet (0 – 65,535)
<code>Alphabet()</code>	constructs a new alphabet using characters 0 through 255
<code>Alphabet(int radix)</code>	constructs a new alphabet using characters 0 through <code>radix - 1</code>
<code>Alphabet(String s)</code>	constructs a new alphabet from the string of characters <code>s</code>
<code>boolean contains(char c)</code>	returns <code>true</code> if <code>c</code> is a character in this alphabet, and <code>false</code> otherwise
<code>int radix()</code>	returns the radix of this alphabet
<code>int lgRadix()</code>	returns the binary logarithm (rounded up) of this alphabet's radix
<code>int toIndex(char c)</code>	returns the index of <code>c</code>
<code>int[] toIndices(String s)</code>	returns the indices of the characters in <code>s</code>
<code>char toChar(int index)</code>	returns the character with the given index
<code>String toChars(int[] indices)</code>	returns the characters with the given indices

LSD	
<code>static void sort(String[] a)</code>	sorts the array <code>a</code> of fixed-length strings over the extended ASCII alphabet

MSD	
<code>static void sort(String[] a)</code>	sorts the array <code>a</code> of strings over the extended ASCII alphabet

TrieST	
<code>TrieST()</code>	constructs an empty symbol table
<code>boolean isEmpty()</code>	returns <code>true</code> if this symbol table is empty, and <code>false</code> otherwise
<code>int size()</code>	returns the number of key-value pairs in this symbol table
<code>void put(String key, V value)</code>	inserts the <code>key</code> and <code>value</code> pair into this symbol table
<code>V get(String key)</code>	returns the value associated with <code>key</code> in this symbol table, or <code>null</code>
<code>boolean contains(String key)</code>	returns <code>true</code> if this symbol table contains <code>key</code> , and <code>false</code> otherwise
<code>void delete(String key)</code>	deletes <code>key</code> and the associated value from this symbol table
<code>Iterable<String> keys()</code>	returns all the keys in this symbol table
<code>Iterable<String> keysWithPrefix(String prefix)</code>	returns all the keys in this symbol table that start with <code>prefix</code>
<code>Iterable<String> keysThatMatch(String pattern)</code>	returns all the keys in this symbol table that match <code>pattern</code> , where the <code>.</code> symbol is treated as a wildcard character
<code>Iterable<String> longestPrefixOf(String query)</code>	returns the string in this symbol table that is the longest prefix of <code>query</code> , or <code>null</code>
<code>String toString()</code>	returns a string representation of this symbol table

☰ KMP

<code>KMP(String pattern, int radix)</code>	preprocesses the <code>pattern</code> string with alphabet size given by <code>radix</code>
<code>int search(String text)</code>	returns the index of the first occurrence of the pattern string within the <code>text</code> string, or the length of the text string

☰ NFA

<code>NFA(String regexp)</code>	constructs a nondeterministic finite state automaton (NFA) from <code>regexp</code>
<code>boolean recognizes(String text)</code>	returns <code>true</code> this NFA recognizes <code>text</code> , and <code>false</code> otherwise

☰ Genome

<code>static void compress()</code>	reads from standard input a sequence characters over the alphabet $\{A, C, G, T\}$; compresses them using two bits per character; and writes the results to standard output
<code>static void expand()</code>	reads from standard input a sequence of genome-compressed bits; expands each two bits into a character over the alphabet $\{A, C, G, T\}$; and writes the results to standard output

☰ RunLength

<code>static void compress()</code>	reads from standard input a sequence of bits; compresses them using run-length coding with 8-bit run lengths; and writes the results to standard output
<code>static void expand()</code>	reads from standard input a sequence of runlength-compressed bits; expands them; and writes the results to standard output

☰ Huffman

<code>static void compress()</code>	reads from standard input a sequence of bytes; compresses them using Huffman codes with an 8-bit alphabet; and writes the results to standard output
<code>static void expand()</code>	reads from standard input a sequence of Huffman-compressed bits; expands them; and writes the results to standard output