Data Structures and Algorithms

This document describes 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.

Fundamentals

```java
Point2D implements Comparable<Point2D>

Point2D(double x, double y) constructs a point (x, y)
double x() returns the x-coordinate of this point
double y() returns the y-coordinate of this point
double r() returns the polar radius of this point
double theta() returns the polar angle (−π, π) of this point
double distanceTo(Point2D other) returns the Euclidean distance between this point and other
double distanceSquaredTo(Point2D other) returns the squared Euclidean distance between this point and other
boolean equals(Object other) returns true if this point and other have the same x- and y-coordinates, and false otherwise
int hashCode() returns a hash code for this point
String toString() returns a string representation of this point
void draw() draws this point using standard draw
void drawTo(Point2D other) draws a line between this point and other using standard draw
int compareTo(Point2D other) returns a comparison of this point with other by their x- and y-coordinates
Comparator<Point2D> atan2Order() returns a comparator for comparing two points by the atan2 angle (−π, π) they make with this point
Comparator<Point2D> polarOrder() returns a comparator for comparing two points by the polar angle (0, 2π) they make with this point
Comparator<Point2D> distanceOrder() returns a comparator for comparing two points by their distance to this point
static Comparator<Point2D> xOrder() returns a comparator for comparing two points by their x-coordinate
static Comparator<Point2D> yOrder() returns a comparator for comparing two points by their y-coordinate
static Comparator<Point2D> rOrder() returns a comparator for comparing two points by their polar radius
```

```java
RectHV

RectHV(double xmin, double ymin, double xmax, double ymax) constructs a rectangle [xmin, ymin] x [xmax, ymax]
double xMin() returns the minimum x-coordinate of any point in this rectangle
double yMin() returns the minimum y-coordinate of any point in this rectangle
double xMax() returns the maximum x-coordinate of any point in this rectangle
double yMax() returns the maximum y-coordinate of any point in this rectangle
double width() returns the width of this rectangle
double height() returns the height of this rectangle
boolean intersects(RectHV other) returns true if this rectangle intersects other, and false otherwise
boolean contains(Point2D p) returns true if this rectangle contains the point p, and false otherwise
double distanceTo(Point2D p) returns the Euclidean distance between the point p and the closest point on this rectangle, and 0 if the point is within
double distanceSquaredTo(Point2D p) returns the squared Euclidean distance between the point p and the closest point on this rectangle, and 0 if the point is within
boolean equals(Object other) returns true if this rectangle and other have the same x- and y-bounds, and false otherwise
int hashCode() returns a hash code for this rectangle
String toString() returns a string representation of this rectangle
void draw() draws this rectangle using standard draw
```

1 A data type name in italics denotes an interface.
### Vector

- **Vector(double[] coords)** constructs a vector given its components
- **double get(int i)** returns the i\(^{th}\) component of this vector
- **Vector add(Vector other)** returns the sum of this vector and other
- **Vector subtract(Vector other)** returns the difference of this vector and other
- **double dot(Vector other)** returns the dot product of this vector and other
- **Vector scale(double alpha)** returns a scaled (by factor alpha) copy of this vector
- **Vector direction()** returns a unit vector in the direction of this vector
- **double magnitude()** returns the magnitude of this vector
- **int dimension()** returns the dimension of this vector
- **String toString()** returns a string representation of this vector

### Counter implements Comparable<Counter>

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

### Date implements Comparable<Date>

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

### Transaction implements Comparable<Transaction>

- **Transaction(String name, Date date, double amount)** constructs a transaction from a name, date, and amount
- **Transaction(String s)** constructs a transaction from a string s of the form “name date amount”
- **String name()** returns the name of the person involved in this transaction
- **Date date()** returns the date of this transaction
- **double amount()** returns the amount of this transaction
- **int hashCode()** returns a hash code for this transaction
- **String toString()** returns a string representation of this transaction
- **int compareTo(Transaction other)** returns a comparison of this transaction with other by amount
- **static Comparator<Transaction> nameOrder()** returns a comparator for comparing two transactions by name
- **static Comparator<Transaction> dateOrder()** returns a comparator for comparing two transactions by date
- **static Comparator<Transaction> amountOrder()** returns a comparator for comparing two transactions by amount
LinearSearch
static int indexOf(Object[] a, Object key) returns the index of key in the array a, or -1

BinarySearch
static int indexOf(Comparable[] a, Comparable key) returns the index of key in the sorted array a, or -1

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

LinkedBag<Item> implements Bag<Item>
LinkedBag() constructs an empty bag

ResizingArrayBag<Item> implements Bag<Item>
ResizingArrayBag() constructs an empty bag

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

LinkedQueue<Item> implements Queue<Item>
LinkedQueue() constructs an empty queue

ResizingArrayQueue<Item> implements Queue<Item>
ResizingArrayQueue() constructs an empty queue

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

LinkedStack<Item> implements Stack<Item>
LinkedStack() constructs an empty stack
ResizingArrayStack<Item> implements Stack<Item>

ResizingArrayStack() constructs an empty stack

UF

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

QuickFindUF implements UF

QuickFindUF(int n) constructs an empty union-find data structure with n sites

QuickUnionUF implements UF

QuickUnionUF(int n) constructs an empty union-find data structure with n sites

WeightedQuickUnionUF implements UF

WeightedQuickUnionUF(int n) constructs an empty union-find data structure with n sites

Sorting

Bubble, Selection, Insertion, Shell, Merge, Quick, Quick3way, Heap

static void sort(Comparable[] a) sorts the array a according to the natural order of its objects
static void sort(Object[] a, Comparator c) sorts the array a according to the order induced by the comparator c

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 true if this minPQ is empty, and false otherwise
int size() returns the number of keys in this minPQ
void insert(Key key) adds key to this minPQ
Key min() returns the smallest key in this minPQ
Key delMin() removes and returns the smallest key in 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 `true` if this maxPQ is empty, and `false` otherwise
- `int size()` returns the number of keys in this maxPQ
- `void insert(Key key)` adds `key` to this maxPQ
- `Key max()` returns the largest key in this maxPQ
- `Key delMax()` removes and returns the largest key in 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, maxN)`
- `boolean isEmpty()` returns `true` if this indexMinPQ is empty, and `false` otherwise
- `int size()` returns the number of keys in this indexMinPQ
- `void insert(int i, Key key)` associates `key` with index `i` in this indexMinPQ
- `void change(int i, Key key)` changes the key associated with index `i` to `key` in this indexMinPQ
- `boolean contains(int i)` returns `true` if `i` is an index in this indexMinPQ, and `false` 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 `i` in this indexMinPQ
- `int delMin()` removes the smallest key from this indexMinPQ and returns its associated index
- `void delete(int i)` removes the key associated with index `i` in 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 delMax()` 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
- `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`
## Searching

### BasicST<Key, Value>

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean isEmpty()</td>
<td>returns true if this symbol table is empty, and false otherwise</td>
</tr>
<tr>
<td>int size()</td>
<td>returns the number of key-value pairs in this symbol table</td>
</tr>
<tr>
<td>void put(Key key, Value value)</td>
<td>inserts the key and value pair into this symbol table</td>
</tr>
<tr>
<td>Value get(Key key)</td>
<td>returns the value associated with key in this symbol table, or null</td>
</tr>
<tr>
<td>boolean contains(Key key)</td>
<td>returns true if this symbol table contains key, and false otherwise</td>
</tr>
<tr>
<td>void delete(Key key)</td>
<td>deletes key and the associated value from this symbol table</td>
</tr>
<tr>
<td>Iterable&lt;Key&gt; keys()</td>
<td>returns all the keys in this symbol table</td>
</tr>
</tbody>
</table>

### OrderedST<Key extends Comparable<Key>, Value>

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean isEmpty()</td>
<td>returns true if this symbol table is empty, and false otherwise</td>
</tr>
<tr>
<td>int size()</td>
<td>returns the number of key-value pairs in this symbol table</td>
</tr>
<tr>
<td>void put(Key key, Value value)</td>
<td>inserts the key and value pair into this symbol table</td>
</tr>
<tr>
<td>Value get(Key key)</td>
<td>returns the value associated with key in this symbol table, or null</td>
</tr>
<tr>
<td>boolean contains(Key key)</td>
<td>returns true if this symbol table contains key, and false otherwise</td>
</tr>
<tr>
<td>void delete(Key key)</td>
<td>deletes key and the associated value from this symbol table</td>
</tr>
<tr>
<td>Iterable&lt;Key&gt; keys()</td>
<td>returns all the keys in this symbol table in sorted order</td>
</tr>
<tr>
<td>Key min()</td>
<td>returns the smallest key in this symbol table</td>
</tr>
<tr>
<td>Key max()</td>
<td>returns the largest key in this symbol table</td>
</tr>
<tr>
<td>void deleteMin()</td>
<td>deletes the smallest key and the associated value from this symbol table</td>
</tr>
<tr>
<td>void deleteMax()</td>
<td>deletes the largest key and the associated value from this symbol table</td>
</tr>
<tr>
<td>Key floor(Key key)</td>
<td>returns the largest key in this symbol table that is smaller than or equal to key</td>
</tr>
<tr>
<td>Key ceiling(Key key)</td>
<td>returns the smallest key in this symbol table that is greater than or equal to key</td>
</tr>
<tr>
<td>int rank(Key key)</td>
<td>returns the number of keys in this symbol table that are strictly smaller than key</td>
</tr>
<tr>
<td>Key select(int k)</td>
<td>returns the key in this symbol table with the rank k</td>
</tr>
<tr>
<td>int size(Key lo, Key hi)</td>
<td>returns the number of keys in this symbol table that are in the interval [lo, hi]</td>
</tr>
<tr>
<td>Iterable&lt;Key&gt; keys(Key lo, Key hi)</td>
<td>returns the keys in this symbol table that are in the interval [lo, hi] in sorted order</td>
</tr>
</tbody>
</table>

### LinearSearchST<Key, Value> implements BasicST<Key, Value>

LinearSearchST() constructs an empty symbol table

### BinarySearchST<Key extends Comparable<Key>, Value> implements OrderedST<Key, Value>

BinarySearchST() constructs an empty symbol table

### BinarySearchTreeST<Key extends Comparable<Key>, Value> implements OrderedST<Key, Value>

BinarySearchTreeST() constructs an empty symbol table

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Iterable&lt;Key&gt; preOrder()</td>
<td>returns all the keys from this symbol table in pre-order</td>
</tr>
<tr>
<td>Iterable&lt;Key&gt; inOrder()</td>
<td>returns all the keys from this symbol table in in-order</td>
</tr>
<tr>
<td>Iterable&lt;Key&gt; postOrder()</td>
<td>returns all the keys from this symbol table in post-order</td>
</tr>
</tbody>
</table>

### RedBlackBinarySearchTreeST<Key extends Comparable<Key>, Value> implements OrderedST<Key, Value>

RedBlackBinarySearchTreeST() constructs an empty symbol table
### SeparateChainingHashST<Key, Value> implements BasicST<Key, Value>

- **SeparateChainingHashST()**: constructs an empty symbol table

### Set<Key extends Comparable<Key>> implements Iterable<Key>

- **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(Key key)**: adds key to this set, if it is not already present
- **boolean contains(Key key)**: returns true if this set contains key, and false otherwise
- **void delete(Key key)**: deletes key from this set
- **Iterator<Key> iterator()**: returns an iterator to iterate over the keys in this set in sorted order

### 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 i'th component of this vector to value
- **double get(int i)**: returns the i'th 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 a directed 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
### Data Structures and Algorithms

#### DFSPaths

- **DFSPaths(Graph G, int s)**: Computes paths between source vertex `s` and every other vertex in the graph `G`.
- **boolean hasPathTo(int v)**: Returns true if there is a path between the source and vertex `v`, and false otherwise.
- **Iterable<Integer> pathTo(int v)**: Returns a path between the source and vertex `v`, or null.

#### BFSPaths

- **BFSPaths(Graph G, int s)**: Computes shortest paths between source vertex `s` and every other vertex in the graph `G`.
- **boolean hasPathTo(int v)**: Returns true if there is a path between the source and vertex `v`, and false otherwise.
- **Iterable<Integer> pathTo(int v)**: Returns a shortest path between the source and vertex `v`, or null.
- **int distTo(int v)**: Returns the shortest-path distance between the source and vertex `v`, or ∞.

#### SymbolGraph

- **SymbolGraph(In in, String delim)**: Constructs a symbol graph from the input stream `in` and using `delim` as the delimiter.
- **boolean contains(String s)**: Returns true if this symbol graph contains vertex `s`, and false otherwise.
- **int indexOf(String s)**: Returns the integer associated with the vertex `s` in this symbol graph.
- **String nameOf(int v)**: Returns the name of the vertex associated with the integer `v` in this symbol graph.
- **Graph graph()**: Returns the graph associated with this symbol graph.

#### DiGraph

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

#### DFSDiPaths

- **DFSDiPaths(DiGraph G, int s)**: Computes paths from source vertex `s` to every other vertex in the digraph `G`.
- **boolean hasPathTo(int v)**: Returns true if there is a path from the source to vertex `v`, and false otherwise.
- **Iterable<Integer> pathTo(int v)**: Returns a path from the source to vertex `v`, or null.

#### BFSDiPaths

- **BFSDiPaths(DiGraph G, int s)**: Computes shortest paths from source vertex `s` to every other vertex in the digraph `G`.
- **boolean hasPathTo(int v)**: Returns true if there is a path from the source to vertex `v`, and false otherwise.
- **Iterable<Integer> pathTo(int v)**: Returns a shortest path from the source to vertex `v`, or null.
- **int distTo(int v)**: Returns the shortest-path distance from the source to vertex `v`, or ∞.

#### SymbolDiGraph

- **SymbolDiGraph(In in, String delim)**: Constructs a symbol digraph from the input stream `in` and using `delim` as the delimiter.
- **boolean contains(String s)**: Returns true if this symbol digraph contains vertex `s`, and false otherwise.
- **int indexOf(String s)**: Returns the integer associated with the vertex `s` in this symbol digraph.
- **String nameOf(int v)**: Returns the name of the vertex associated with the integer `v` in this symbol digraph.
- **DiGraph diGraph()**: Returns the digraph associated with this symbol digraph.
### DiCycle

DiCycle(DiGraph G) determines whether the digraph $G$ has a directed cycle and, if so, finds such a cycle.

- **boolean hasCycle()** returns `true` if a directed cycle was detected, and `false` otherwise.
- **Iterable<Integer> cycle()** returns a directed cycle, or `null`.

### DFSOrders

DFSOrders(DiGraph G) determines depth-first orders (pre, post, and reverse post) for the digraph $G$.

- **int pre(int v)** returns the pre-order number of vertex $v$.
- **int post(int v)** returns the post-order number of vertex $v$.
- **Iterable<Integer> pre()** returns the vertices in pre-order.
- **Iterable<Integer> post()** returns the vertices in post-order.
- **Iterable<Integer> reversePost()** returns the vertices in reverse post-order.

### Topological

Topological(DiGraph G) determines whether the digraph $G$ has a topological order and, if so, finds such an order.

- **boolean hasOrder()** returns `true` if there exists a topological order, and `false` otherwise.
- **Iterable<Integer> order()** returns a topological order, or `null`.

### Edge implements Comparable<Edge>

Edge(int v, int w, double weight) constructs an edge between vertices $v$ and $w$ of the given weight.

- **int either()** returns one endpoint of this edge.
- **int other(int v)** returns the endpoint of this edge that is different from vertex $v$.
- **double weight()** returns the weight of this edge.
- **String toString()** returns a string representation of this edge.
- **int compareTo(Edge other)** returns a comparison of this edge with $other$ 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.
Data Structures and Algorithms

### 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

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 true if there is a path from the source to vertex v, and false otherwise
- Iterable<Integer> pathTo(int v) returns a shortest path from the source to vertex v, or null
- int distTo(int v) returns the shortest-path distance from the source to vertex v, or ∞
Alphabet

- **static Alphabet BINARY** - the binary alphabet \{0, 1\}
- **static Alphabet OCTAL** - the octal alphabet \{0, 1, 2, 3, 4, 5, 6, 7\}
- **static Alphabet DECIMAL** - the decimal alphabet \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}
- **static Alphabet HEXADECIMAL** - the hexadecimal alphabet \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F\}
- **static Alphabet DNA** - the DNA alphabet \{A, C, G, T\}
- **static Alphabet LOWERCASE** - the lowercase alphabet \{a, b, c, ..., z\}
- **static Alphabet UPPERCASE** - the uppercase alphabet \{A, B, C, ..., Z\}
- **static Alphabet BASE64** - the base-64 alphabet (64 characters)
- **static Alphabet ASCII** - the ASCII alphabet (0−127)
- **static Alphabet EXTENDED_ASCII** - the extended ASCII alphabet (0−255)
- **static Alphabet UNICODE16** - the Unicode 16 alphabet (0−65,535)

**Alphabet()** constructs a new alphabet using characters 0 through 255

**Alphabet(int radix)** constructs a new alphabet using characters 0 through radix - 1

**Alphabet(String s)** constructs a new alphabet from the string of characters s

**boolean contains(char c)** returns true if c is a character in this alphabet, and false otherwise

**int radix()** returns the radix of this alphabet

**int lgRadix()** returns the binary logarithm (rounded up) of this alphabet’s radix

**int toIndex(char c)** returns the index of c

**int[] toIndices(String s)** returns the indices of the characters in s

**char toChar(int index)** returns the character with the given index

**String toChars(int[] indices)** returns the characters with the given indices

LSD

**static void sort(String[] a)** sorts the array a of fixed-length strings over the extended ASCII alphabet

MSD

**static void sort(String[] a)** sorts the array a of strings over the extended ASCII alphabet

TrieST

**TrieST()** 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(String key, Value value)** inserts the key and value pair into this symbol table

**Value get(String key)** returns the value associated with key in this symbol table, or null

**boolean contains(String key)** returns true if this symbol table contains key, and false otherwise

**void delete(String key)** deletes key and the associated value from this symbol table

**Iterable<String> keys()** returns all the keys in this symbol table

**Iterable<String> keysWithPrefix(String prefix)** returns all the keys in this symbol table that start with prefix

**Iterable<String> keysThatMatch(String pattern)** returns all the keys in this symbol table that match pattern, where the . symbol is treated as a wildcard character

**Iterable<String> longestPrefixOf(String query)** returns the string in this symbol table that is the longest prefix of query, or null

KMP

**KMP(String pattern, int radix)** preprocesses the pattern string with alphabet size given by radix

**int search(String text)** returns the index of the first occurrence of the pattern string within the text string, or the length of the text string
### NFA

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NFA(String regexp)</td>
<td>constructs a nondeterministic finite state automaton (NFA) from regexp</td>
</tr>
<tr>
<td>boolean recognizes(String text)</td>
<td>returns true if this NFA recognizes text, and false otherwise</td>
</tr>
</tbody>
</table>

### Genome

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>static void compress()</td>
<td>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</td>
</tr>
<tr>
<td>static void expand()</td>
<td>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</td>
</tr>
</tbody>
</table>

### RunLength

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>static void compress()</td>
<td>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</td>
</tr>
<tr>
<td>static void expand()</td>
<td>reads from standard input a sequence of runlength-compressed bits; expands them; and writes the results to standard output</td>
</tr>
</tbody>
</table>

### Huffman

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>static void compress()</td>
<td>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</td>
</tr>
<tr>
<td>static void expand()</td>
<td>reads from standard input a sequence of Huffman-compressed bits; expands them; and writes the results to standard output</td>
</tr>
</tbody>
</table>