Geometric Data Structures
Data Structure

**Definition:** A *data structure* is a particular way of organizing and storing data in a computer for efficient search and retrieval, including associated algorithms to perform search, update, and related operations on the data structure.
Elmar Langetepe, Gabriel Zachmann
*Geometric Data Structures for Computer Graphics*

Mark de Berg, Otfried Cheong, Marc van Kreveld, Mark Overmars
*Computational Geometry: Algorithms and Applications*
Hanan Samet
*Foundations of Multidimensional and Metric Data Structures*
Morgan Kaufmann, 2006. 1024 pages.

Dinesh P. Mehta, Sartaj Sahni
*Handbook of Data Structures and Applications*
Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein
Introduction to Algorithms 3rd ed.
MIT Press, 2009

Robert Sedgewick
Addison-Wesley, 1998.
Kurt Mehlhorn
*Data Structures and Algorithms 1: Sorting and Searching*

Kurt Mehlhorn
*Data Structures and Algorithms 3: Multi-Dimensional Searching and Computational Geometry*

Rolf Klein
*Algorithmische Geometrie* 2nd ed.

Franco Preparata, Michael Shamos
*Computational Geometry. An Introduction*
Michael Goodrich, Roberto Tamassia
*Algorithm Design.*
*Foundations, Analysis and Internet Examples*

Thomas Ottmann, Peter Widmayer
*Algorithmen und Datenstrukturen* 4th ed.
Spektrum, 2002.
Definition: An abstract data type is a set of data values and associated operations that are precisely specified independent of any particular implementation.
Abstract Data Type

**Definition:** An *abstract data type* is a set of data values and associated operations that are precisely specified independent of any particular implementation.

A data structure is an actual implementation of an abstract data type.
Dictionary

Abstract data type *dictionary* maintains a subset $S$ of a universe $U$ under the operations

- $\text{INSERT}(x)$
- $\text{DELETE}(x)$
- $\text{SEARCH}(x)$

alternative names: $\text{LOOKUP}(x)$, $\text{RETRIEVE}(x)$, $\text{FIND}(x)$

This data type is also called *set*. 

Ordered Dictionary

Abstract data type *ordered dictionary* maintains a subset $S$ of an ordered universe $U$ under the operations

- $\text{INSERT}(x)$
- $\text{DELETE}(x)$
- $\text{SEARCH}(x)$
- $\text{PREDECESSOR}(x)$
- $\text{SUCCESSOR}(x)$

This data type is also called *ordered set*. 
Order-Statistics Dictionary

Abstract data type *order-statistics dictionary* maintains a subset $S$ of an ordered universe $U$ under the operations

- **Insert**($x$)
- **Delete**($x$)
- **Search**($x$)
- **Rank**($x$)
- **Select**($k$)

**Rank**($x$) computes the rank of $x$ in $S$, i.e., $|\{s \in S : s \leq x\}|$.

**Select**($k$) computes the $k$-th smallest element in $S$.

This data type is also called *dynamic order-statistics set*. 
Monotone Priority Queue

Abstract data type *priority queue* maintains a subset $S$ of an ordered universe $U$ under the operations

- **Insert**($x$)
- **Delete-Min**()
Priority Queue

Abstract data type *priority queue* maintains a set of pairs of items and keys from an ordered universe $U$ under the operations

- **Insert**$(x, key)$
- **Delete-Min**$(\cdot)$
- **Decrease-Key**$(x, key)$
Application:

Dijkstra’s algorithm for single source shortest paths:

\textsc{Dijkstra}(G(V, E), s)

1 \hspace{1em} \textbf{for each} \ v \in V
2 \hspace{0.5em} \textbf{do} \ d[v] \leftarrow \infty
3 \hspace{1em} Q.\textsc{Insert}(v, d[v])
4 \hspace{1em} Q.\textsc{Decrease-key}(s, 0)
5 \hspace{0.5em} \textbf{while} \ Q \text{ is not empty}
6 \hspace{1em} \textbf{do} \ u \leftarrow Q.\textsc{Delete-min}()
7 \hspace{1em} \textbf{for each} \ \text{neighbor} \ w \ \text{of} \ u
8 \hspace{1em} \textbf{do if} \ d[u] + \text{dist}(u, w) < d[w]
9 \hspace{1em} \textbf{then} \ d[w] \leftarrow d[u] + \text{dist}(u, w)
10 \hspace{0.5em} Q.\textsc{Decrease-key}(w, d[w])
Disjoint Set

A *disjoint-set data structure* maintains a partition under the operations

- \textbf{FIND}(x)
- \textbf{UNION}(x, y)

This data type is also called *union-find data structure.*
Planar Point Location

A planar subdivision is a partitioning of the plane into vertices, edges, and faces. In the planar point location problem we are given a planar subdivision $S$ and a query point $q$ and the task is to find the vertex or edge or face containing $q = (q_x, q_y)$. 
Klee’s Measure Problem

Given a collection of intervals, what is the length of their union?
Klee’s Measure Problem

Given a collection of intervals, what is the length of their union?

Given a collection of axis-aligned rectangles, what is the area of their union?
Range Queries

Given a set $S$ of $n$ points in $\mathbb{R}^d$ and a family $\mathcal{R}$ of subspaces of $\mathbb{R}^d$. Elements of $\mathcal{R}$ are called ranges. Report or count all elements in $S$ intersecting a given query range. $\mathcal{R}$ is called range space.
orthogonal range searching

simplex range searching

orthogonal range searching

halfspace range searching
Stabbing Queries

Given a set $S$ of subsets of $\mathbb{R}^d$. Report or count all elements in $S$ intersected by a given query object, e.g. a point or a line (segment).
Interval Overlap

Maintain a set of closed intervals

\[ I = \{ [x_1, x'_1], [x_2, x'_2], \ldots, [x_n, x'_n] \} \]

such that an emptiness stabbing query with a closed interval

\[ [q, q'] \]

can be answered quickly.
Interval Overlap

Maintain a set of closed intervals

\[ I = \{ [x_1, x'_1], [x_2, x'_2], \ldots, [x_n, x'_n] \} \]

such that an emptiness stabbing query with a closed interval

\[ [q, q'] \]

can be answered quickly. Remember, we want to check whether any of the intervals in \( I \) overlaps \([q, q']\).
Six cases:
Application:

Intersection detection of axis-parallel rectangles using plane-sweep

“Sweep” a horizontal line from top to bottom. Maintain the intersection intervals of the rectangles and the sweep line in a data structure appropriate for interval overlap queries. Check for overlap at the upper sides of the rectangles.
Sweep line reaches an upper side:

\[ \text{OVERLAP}(I) \]

\[ \text{INSERT}(I) \]
Sweep line reaches a lower side:

\[
\text{DELETE}(I)
\]
Binary Tree

**Definition:** A *binary tree* contains exactly one external node or is comprised of three disjoint sets of nodes: An internal node \( v \) called *root* node, a binary tree called its *left subtree* and a binary tree called its *right subtree*. 

If \( w \) is a (left or right) child of \( u \), the node \( u \) is called the *parent* of \( w \).
Binary Tree

Definition: A *binary tree* contains exactly one external node or is comprised of three disjoint sets of nodes: An internal node $v$ called *root* node, a binary tree called its *left subtree* and a binary tree called its *right subtree*. A binary tree without internal nodes is called *empty*. 
Binary Tree

**Definition:** A *binary tree* contains exactly one external node or is comprised of three disjoint sets of nodes: An internal node $v$ called *root* node, a binary tree called its *left subtree* and a binary tree called its *right subtree*. A binary tree without internal nodes is called *empty*. If the left subtree is nonempty, its root is called the *left child* of $v$. Likewise, if the right subtree is nonempty, its root is called the *right child* of $v$. 
Binary Tree

Definition: A binary tree contains exactly one external node or is comprised of three disjoint sets of nodes: An internal node $v$ called root node, a binary tree called its left subtree and a binary tree called its right subtree. A binary tree without internal nodes is called empty. If the left subtree is nonempty, its root is called the left child of $v$. Likewise, if the right subtree is nonempty, its root is called the right child of $v$. An internal node without children is a leaf.
Binary Tree

Definition: A binary tree contains exactly one external node or is comprised of three disjoint sets of nodes: An internal node \( v \) called root node, a binary tree called its left subtree and a binary tree called its right subtree. A binary tree without internal nodes is called empty. If the left subtree is nonempty, its root is called the left child of \( v \). Likewise, if the right subtree is nonempty, its root is called the right child of \( v \). An internal node without children is a leaf. If \( w \) is a (left or right) child of \( u \), the node \( u \) is called the parent of \( w \).
Binary Tree

**Definition:** A binary tree contains exactly one external node or is comprised of three disjoint sets of nodes: An internal node \( v \) called *root* node, a binary tree called its *left subtree* and a binary tree called its *right subtree*. A binary tree without internal nodes is called *empty*. If the left subtree is nonempty, its root is called the *left child* of \( v \). Likewise, if the right subtree is nonempty, its root is called the *right child* of \( v \). An internal node without children is a *leaf*. If \( w \) is a (left or right) child of \( u \), the node \( u \) is called the *parent* of \( w \).

**Examples:**
[Source: Sedgewick; *Algorithmen in Java*. © Pearson Education]
**Definition:** The *height* of a node $v$ in a binary tree is the length of the longest path from $v$ to a leaf.
Definition: The *height* of a node \( v \) in a binary tree is the length of the longest path from \( v \) to a leaf. The height of a binary tree \( T \) is the height of the root of \( T \).
**Definition:** The *height* of a node \( v \) in a binary tree is the length of the longest path from \( v \) to a leaf. The height of a binary tree \( T \) is the height of the root of \( T \).

**Definition:** The *depth* of a node \( v \) in a binary tree \( T \) is the length of the path form the root of \( T \) to \( v \).
**Definition:** The *height* of a node $v$ in a binary tree is the length of the longest path from $v$ to a leaf. The height of a binary tree $T$ is the height of the root of $T$.

**Definition:** The *depth* of a node $v$ in a binary tree $T$ is the length of the path from the root of $T$ to $v$.

**Definition:** A node $v$ in a binary tree is an *ancestor* of a node $w$ if $w$ belongs to the set of nodes of one of $v$’s children.
**Definition:** A *binary search tree* is a binary tree that has a key $K$ associated to each internal node such that the keys of the nodes in the left subtree are smaller than $K$ and the keys of the nodes in the right subtree are larger than $K$.

[Source: Sedgewick; *Algorithmen in Java.* ©Pearson Education]
Binary Search Tree

**Definition:** A *binary search tree* is a binary tree that has a key $K$ associated to each internal node such that the keys of the nodes in the left subtree are smaller than $K$ and the keys of the nodes in the right subtree are larger than $K$.

![Binary Search Tree Diagram]

Binary search trees implement the abstract data type ordered dictionary.
Search

To search for key $K$, start at the root and walk down the tree. Use the key $K_v$ of the current node $v$ to navigate through the tree: If $K = K_v$, we are done. If $K < K_v$, go to the left subtree, if $K > K_v$ go to the right subtree. In the figure on the right we search for key H.
Search

To search for key $K$, start at the root and walk down the tree. Use the key $K_v$ of the current node $v$ to navigate through the tree: If $K = K_v$, we are done. If $K < K_v$, go to the left subtree, if $K > K_v$ go to the right subtree. In the figure on the right we search for key H.

Lemma:
Searching for a key in a binary search tree of height $h$ takes time $O(h)$.

Here and in the sequel we assume that comparing keys takes constant time.
To insert a key $K$, we search for $K$ as described above. If we find $K$, we are done. Otherwise, the search will end in an external node. We replace the external node by a new internal node $v$ and associate $K$ with $v$. In the figure on the right we insert key M.
To insert a key $K$, we search for $K$ as described above. If we find $K$, we are done. Otherwise, the search will end in an external node. We replace the external node by a new internal node $v$ and associate $K$ with $v$. In the figure on the right we insert key $M$.

**Lemma:**
Insertion of a key in a binary search tree of height $h$ takes time $O(h)$. 

[Source: Sedgewick; *Algorithmen in Java*. © Pearson Education]
**DELETE**

To delete a key $K$, we search for $K$ as described above. If we do not find $K$, we are done. Otherwise, let $v_K$ be the node containing $K$. If $v_K$ has at most one child, we remove $v_K$. If there is a child $v_c$, we make $v_c$ a child of $v_K$’s parent. Otherwise, if $v_K$ has two children, we search for $K$’s successor, i.e., for the leftmost internal node $v_l$ in $v_K$’s right subtree. Node $v_l$ has at most one child! We exchange the keys of $v_K$ and $v_l$ and remove $v_l$ (handling the child as described above, if any). In the figure we delete key E. The key of the leftmost internal node in the right subtree is G.

[Source: Sedgewick; *Algorithmen in Java*. ©Pearson Education]
To delete a key $K$, we search for $K$ as described above. If we do not find $K$, we are done. Otherwise, let $v_K$ be the node containing $K$. If $v_K$ has at most one child, we remove $v_K$. If there is a child $v_c$, we make $v_c$ a child of $v_K$’s parent. Otherwise, if $v_K$ has two children, we search for $K$’s successor, i.e., for the leftmost internal node $v_l$ in $v_K$’s right subtree. Node $v_l$ has at most one child! We exchange the keys of $v_K$ and $v_l$ and remove $v_l$ (handling the child as described above, if any). In the figure we delete key E. The key of the leftmost internal node in the right subtree is G.

Lemma:
Deletion of a key in a binary search tree of height $h$ takes time $O(h)$. 

[Source: Sedgewick; *Algorithmen in Java*. ©Pearson Education]
Insertion
That's it.
Lemma:
The worst-case height of a binary search tree for $n$ keys is $\Theta(n)$.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Complexity</th>
<th>Worst-case</th>
</tr>
</thead>
<tbody>
<tr>
<td>Search</td>
<td>$O(n)$</td>
<td></td>
</tr>
<tr>
<td>Insert</td>
<td>$O(n)$</td>
<td></td>
</tr>
<tr>
<td>Delete</td>
<td>$O(n)$</td>
<td></td>
</tr>
</tbody>
</table>

[Source: Sedgewick; *Algorithmen in Java*. © Pearson Education]
Balanced Binary Search Trees

**Definition:** A node \( v \) in an binary search tree \( T \) is \( \alpha \)-weight-balanced, for some \( \frac{1}{2} \leq \alpha < 1 \), if

\[
\text{size}(lc[v]) \leq \alpha \cdot \text{size}(v) \tag{1}
\]
\[
\text{size}(rc[v]) \leq \alpha \cdot \text{size}(v) \tag{2}
\]

Here and later on, \( lc(v) \) and \( rc(v) \) denote the left and right child of a node \( v \), respectively.
Balanced Binary Search Trees

**Definition:** A node $v$ in an binary search tree $T$ is $\alpha$-weight-balanced, for some $\frac{1}{2} \leq \alpha < 1$, if

$$\text{size}(lc[v]) \leq \alpha \cdot \text{size}(v) \quad (1)$$

$$\text{size}(rc[v]) \leq \alpha \cdot \text{size}(v) \quad (2)$$

Here and later on, $lc(v)$ and $rc(v)$ denote the left and right child of a node $v$, respectively.

**Definition:** A binary search tree $T$ is $\alpha$-weight-balanced if all nodes in $T$ are $\alpha$-weight-balanced.
Balanced Binary Search Trees

**Definition:** A node \( v \) in an binary search tree \( T \) is \( \alpha \)-weight-balanced, for some \( \frac{1}{2} \leq \alpha < 1 \), if

\[
\begin{align*}
\text{size}(lc[v]) & \leq \alpha \cdot \text{size}(v) \\
\text{size}(rc[v]) & \leq \alpha \cdot \text{size}(v)
\end{align*}
\]  

(1)

Here and later on, \( lc(v) \) and \( rc(v) \) denote the left and right child of a node \( v \), respectively.

**Definition:** A binary search tree \( T \) is \( \alpha \)-weight-balanced if all nodes in \( T \) are \( \alpha \)-weight-balanced.

**Definition:** A binary search tree \( T \) of size \( n \) is \( \alpha \)-height-balanced if

\[
h(T) \leq \lceil \log_{(1/\alpha)} n \rceil =: h_\alpha(n)
\]  

(3)
[Source: Sedgewick; *Algorithmen in Java*. ©Pearson Education]
Rotation

You can restructure a tree using rotations as illustrated below. Rotations preserve the ordering of the keys associated with the nodes.
Right Rotation
Right Rotation

Left Rotation

[Source: Sedgewick; *Algorithmen in Java*. ©Pearson Education]