Matrix — Content Manual

June 21, 2004

Contents

1 Introduction

This manual contains detailed information about different contents of Matrix. These include, for example, different data structures that are located in $MATRIX/code/matrix/structures/FDT/probe and in $MATRIX/code/matrix/structures/CDT/probe/.

Although this information is detailed description of the different contents and is therefore mainly meant for developers, a normal user can also find some useful information from here, such as instructions for the basic use of data structures.

2 Data structures

This section handles the data structures that are delivered with Matrix. The implementation and functioning of each structure is explained. Also the basic use of structures is explained. In addition, for every structure its default representation and all possible representations are mentioned.

Structure	**Default representation**	**Possible representations**
Fundamental data types		
Array	array	array
Linked list	list	list
Binary tree	layered tree	layered tree, leaf tree, array
Common tree	layered tree	layered tree, leaf tree, layered graph vertex
Undirected graph	Fruchterman-Reingold graph	layered graph, Kamada-Kawai graph, Fruchterman-Reingold graph, dummy graph, array
Directed graph	Fruchterman-Reingold graph	layered graph, Kamada-Kawai graph, Fruchterman-Reingold graph, dummy graph, array
Conceptual data types		
Queue	list	list
Stack	array, list[1]	array, list
Binary heap	layered tree	array, layered tree, leaf tree
Binary search tree	layered tree	array, layered tree, leaf tree, layered graph vertex
AVL tree	layered tree	array, layered tree, leaf tree, layered graph vertex
Red-black tree	layered tree	array, layered tree, leaf tree, layered graph vertex
B-tree	layered tree	layered tree, leaf tree
Digital search tree	layered tree	layered tree, leaf tree
Radix search trie	layered tree	array, layered tree, leaf tree, layered graph vertex
Splay tree	layered tree	array, layered tree, leaf tree, layered graph vertex

2.1 Fundamental data types

This section contains explanations for each FDTs. These data structures are in $MATRIX/code/structures/FDT/probe.

Fundamental data types, FDTs, include the basic structures like binary trees, arrays, linked lists and graphs. These are presented in the following subsections.

2.1.1 Table

Default representation: array
Possible representations: array
Source code: Table.java

Inserting keys in a table can be done by dropping them either on the key (initially empty or on the index.

A table of Objects that can be visualised as an Array in Matrix. It is possible to create a "random table", which will randomize its keys each time it is refreshed from the GUI. The random table uses three letter keys.

There is also Matrix.java which is two dimensional array.

2.1.2 Linked List

Default representation: list
Possible representations: list
Source code: LinkedListImpl.java

Inserting keys in the list can be done by dropping them onto the structure. This always inserts the keys as the first element of the list. To insert a key in the middle of the list, drop the new key onto the node after which you want the new key to be inserted.

Implementation of a linked list. A linked list is a collection of nodes that contain some information, and a pointer to the next node in the list.

2.1.3 Tree

1. **Binary Tree**
   Source code: BinaryTree.java
   Binary Tree is an interface for all kinds of binary trees. There are two different FDT implementations for binary tree.

   BinTrei
   Default representation: layered tree
   Possible representations: layered tree, leaf tree, array
   Source code: BinTrei.java
   Static binary tree. Implemented as an array. Implicit.

   BinTree
   Default representation: layered tree

Possible representations: layered tree, leaf tree, array, layered graph vertex
Source code: BinTree.java
Dynamic binary tree. Implemented truly as a binary tree. Explicit.

2. **CommonTree**
   Default representation: layered tree
   Possible representations: layered tree, leaf tree, layered graph vertex
   Source code: SimulationTree2Impl.java

   Inserting new nodes can be done by dropping keys onto an existing node. The new key is added as a child node of the node it was inserted in.

   A common tree implementation in which every node has implemented as a VirtualArray. The first position (index = DATA = 0) stores the element and the rest of the array stores the children.

2.1.4 Graph

Default representation: Fruchterman-Reingold graph
Possible representations: layered graph, Kamada-Kawai graph, Fruchterman-Reingold graph, dummy graph, array

1. **Directed graph**
   Source code: DirectedGraphImpl.java

   Nodes can be inserted by dropping them onto the graph. Inserting edges can be done in two ways:

   - by selecting **Insert edge** from the source node's pop-up menu and then clicking on the target node
   - by clicking the source node with Shift key held down and then clicking on the target node.

   VertexImpl
   Source code: VertexImpl.java

2. **Undirected graph**
   Source code: UndirectedGraphImpl.java

   Nodes and edges can be inserted in the same way as for directed graphs.

   UndirVertex
   Source code: UndirVertex.java

   An implementation of Vertex for undirected graphs. All edges added to a graph consisting of UndirVertices are made two-way.

2.1.5 Others

Other files in $MATRIX/code/structures/FDT/probe: ArrayList.java, CommonGraphImpl.java, CommonLabelTreeImpl.java, LabeledBinTree.java, StaticLinkedListImpl.java, VanillaTable.java

2.2 Conceptual Data Types

This section contains explanations for each CDTs. These data structures are in $MATRIX/code/structures/CDT/probe.

Conceptual data types, CDTs, are more complex structures that have a pre-defined set of operations. The implementation of an operation depends on the CDT. Inserting keys should be always done by dropping the keys on the title of the CDT. This way, the structure can decide which position the new structure should be added to. Keys can be deleted by selecting either delete from the pop-up menu or toolbar or by holding the Shift key while dropping them outside the structure.

The currently available CDTs are presented in the following sections.

2.2.1 Dictionaries

Dictionaries have three methods: insert, delete, search. At the moment search is not implemented.

1. **Binary Search Tree**
   Default representation: layered tree
   Possible representations: array, layered tree, leaf tree, layered graph vertex
   Source code: BinSearchTree.java

   A Binary search tree. A binary search tree is a dictionary in the form of a binary tree where the left subtree of each node contains smaller keys than the node's key, and the right subtree contains larger keys. Duplicate keys goes to the left branch of the node.

2. **AVL Tree**
   Default representation: layered tree
   Possible representations: array, layered tree, leaf tree, layered graph vertex
   Source code: AVLTree.java

   Labeled AVL-tree with node's height in node's label.

3. **Red-black Tree**
   Default representation: layered tree
   Possible representations: array, layered tree, leaf tree, layered graph vertex
   Source code: RBTree.java

   Implementation of the RedBlackTree. This is implemented by encapsulating RedBlackNodes inside this RBTree. RebBlackTree is balanced binary search tree with following properties:

   (a) Root is black.

   (b) Every route from root to leaf contains same number of black nodes.

   (c) Red node can't have red childrens.

Every node has parent information so when using methods inherited from BinSearchTree parent information may also need some updating.

RedBlackNode
Source code: RedBlackNode.java
Implementation of the RedBlackNode in RedBlackTree.

4. **Digital Search Tree**
   Default representation: layered tree
   Possible representations: layered tree, leaf tree
   Source code: DigitalSearchTree.java

   A digital search tree. In a digital search tree keys are inserted into the tree according to their bit representation.

5. **Radix Search Trie**
   Default representation: layered tree
   Possible representations: array, layered tree, leaf tree, layered graph vertex
   Source code: RadixSearchTrie.java

   A radix search trie. A radix search trie is a dictionary where keys are stored in a binary tree according to their bit values. All keys are stored in leaf nodes and internal nodes are empty.

6. **Splay Tree**
   Default representation: layered tree
   Possible representations: array, layered tree, leaf tree, layered graph vertex
   Source code: SplayTree.java

   Splay tree is a data structure where the operations adjust the data structure to ensure O(log(n)) amortized running time. Adjustment is done by splaying operations.

7. **B-Tree**
   Default representation: layered tree
   Possible representations: layered tree, leaf tree
   Source code: BTree.java

   A representation of a B-Tree. A B-Tree is balanced, k-ary search tree, where the branching factor k depends on a set minimum degree m. Each node of a B-Tree (except root) has from m-1 to 2m-1 keys and from m to 2m children. Root is empty if the tree is empty. Otherwise the root will have a minimum of one key and, if the tree contains more than 2m keys, two children.

8. **Faulty Binary Search Tree ?**
   Source code: FaultyBinSearchTree.java

   A Faulty Binary search tree that can be used for demonstration purposes. Dublicates are inserted to left whereas in deletion replacing key is taken from the right branch. Therefore the tree is not working correctly.

2.2.2 Basic Data Structures

1. **Queue**
   Default representation: list

Possible representations: list
Source code: QueueImpl.java

Implementation of a queue. This queue is implemented as a (doubly) linked list. Thus, this queue is a collection of nodes that contains elements of the queue. Every node contains also pointers to the previous node and the next node. The node which pointer to previous node points to null is a first node of the queue.

The only representation of the linked list is list so there are no other possible representations for QueueImpl.

2. **Stack**

StackImpl
Default representation: list
Possible representations: list
Source code: StackImpl.java

Implementation of a stack. This stack is implemented as a (doubly) linked list. Thus, this stack is a collection of nodes that contains elements of the stack. Every node contains also pointers to the previous node and the next node. The node which pointer to previous node points to null is a top node of the stack.

This stack extends LinkedListImpl. This causes that StackImpl is always represented as a list and no other representations are available for it.

Stack
Default representation: array
Possible representations: array
Source code: Stack.java

This stack is implemented so that it extends Table. This causes that Stack is always represented as an array and no other representations are available for it.

StackTopVisible
Default representation: array
Possible representations: array
Source code: StackTopVisible.java

This stack is also implemented so that it extends Table and the visualization is therefore an array. The difference to the stack mentioned above is that this stack is implemented so that only the topmost element is visible.

2.2.3 Priority Queues

1. **Binary Heap**
Default representation: layered tree
Possible representations: array, layered tree, leaf tree
Source code: BinHeap.java

2.2.4 Others

Other files in $MATRIX/code/structures/CDT/probe: OrderedList.java

2.3 Utils

Utils are structures that are used to make the manipulation of CDTs and FDTs easier.

Trash
Source code: Trash.java

Trash is a utility, which can be used to delete objects. All visual objects that are dropped onto the Trash are deleted.

2.4 Algorithms

2.4.1 Sequential Search

1. **SearchTable**
   Source code: SearchTable.java

   Abstraction that takes an array (P) as it's input and outputs the start index or indeces of all occurences of P in S by using the given method. Example1: P could be an Element and the method could be linear search algorithm Example2: P could be an Array and the method could be Boyer-Moore-Horspool algorithm The subclasses could determine the method(s) by overwriting the searchKey and searchPattern methods, respectively. By default the two example methods are provided.

3 Representations

There are total four different main representations which are used to visualize the data structures. In addition, two of these are divided into more detailed layouts. The possible representations for a data structure can vary and it depends on the implementation of that data structure.

3.1 array

Array -representation contains only itself.

The layout Array (represented in Fig. 1) can currently be used to represent arrays and trees.

Array visualizations contain a hotspot in their upper right corner. An arrow ("->") appears when the mouse cursor is held over this hotspot. The number of positions the array can hold can be changed by dragging this hotspot left or right.

The size of the array can also be changed using the pop-up menu's submenu Filters.

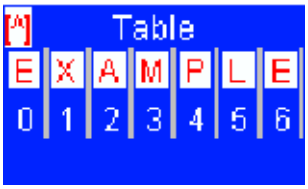

Figure 1: Example of Array layout

3.2 list

List -representation contains only itself.

The layout List is represented in Fig. 2. It can be used to represent linked lists, stacks and queues.

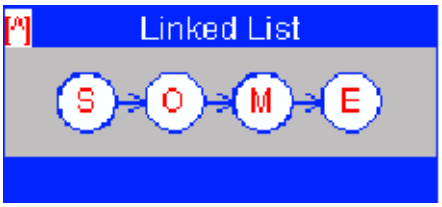

Figure 2: Example of List layout

3.3 tree

3.3.1 layered tree

The Layered Tree layout (represented in Fig. 3) draws a tree using the Layered-Tree-Draw algorithm (Algorithm 3.1 in Battista et. al., **Graph Drawing**, extended to support non-binary trees and variable-size nodes.

3.3.2 leaf tree

A tree-drawing algorithm that keeps the left-right order of the leaves.

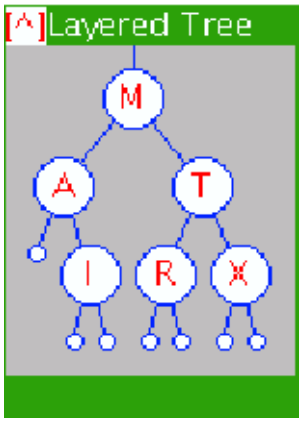

Figure 3: Example of Layered Tree layout

3.4 graph

3.4.1 layered graph vertex

3.4.2 dummy graph

A dummy graph layout algorithm. Lays the graph nodes in a row.

The dummy graph layout (represented in Fig. 4) is the most simple layout. All the nodes are just positioned in a horizontal line.

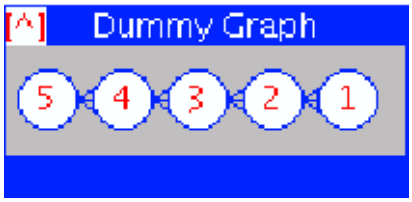

Figure 4: Example of Dummy Graph layout

3.4.3 layered graph

A version of the DAG drawing algorithm from Chapter 9 ("Layered Drawings of Digraphs") in Battista et. al., **Graph Drawing**, supporting arbitrary graphs and variable-size nodes.

The layered graph layout uses a directed acyclic graph algorithm supporting arbitrary graphs and variable-size nodes. It is represented in Fig. 5.

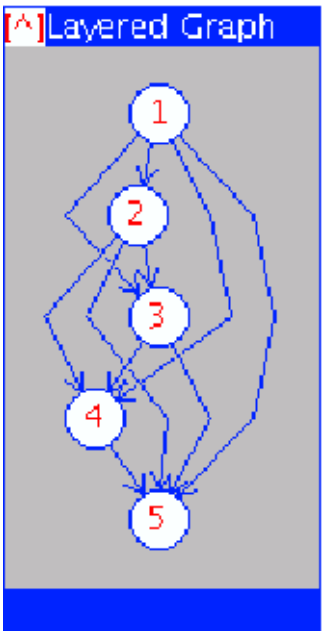

Figure 5: Example of Layered Graph layout

3.4.4 Kamada-Kawai graph

A version of the Kamada-Kawai graph drawing algorithm from paper "An Algorithm for Drawing General Undirected Graphs" in Information Processing Letters 31 (1989) by Kamada and Kawai. The algorithm has been modified for non-uniform vertices as described in paper "Drawing Graphs with Non-Uniform Vertices" in Proceedings of Working Conference on Advanced Visual Interfaces (AVI'02) by Harel and Koren.

The Kamada-Kawai graph layout (represented in Fig. 6) is a layout for graphs that uses an algorithm developed by Kamada and Kawai to layout the nodes.

The layout can be modified by the user by selecting **Change edge length** from the graph's pop-up menu. This value is the optimal length for the edges used in the algorithm. Changing this value can have a tremendous effect on the layout; either positive or negative.

3.4.5 Fruchterman-Reingold graph

A version of the Fruchterman-Reingold graph drawing algorithm from paper "Graph Drawing by Force-directed Placement" in Software - Practice and Experience, Vol. 21(11) (1991) by Thomas Fruchterman and Edward Reingold. The algorithm has been modified for non-uniform vertices as described in paper "Drawing Graphs with Non-Uniform Vertices" in Proceedings of Working Conference on Advanced Visual Interfaces (AVI'02) by Harel and Koren.

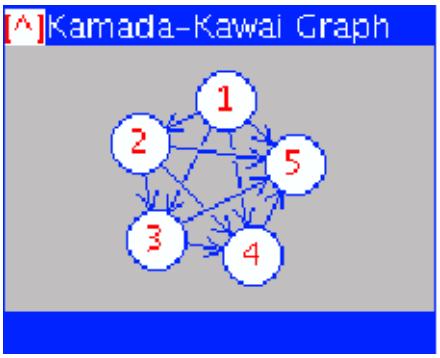

Figure 6: Example of Kamada-Kawai Graph layout

The Fructerman-Reingold graph layout (represented in Fig. 7) is a layout for graphs that uses an algorithm developed by Fructerman and Reingold to layout the nodes.

The layout can be modified like the Kamada-Kawai layout by selecting **Change edge length** from the graph's pop-up menu.

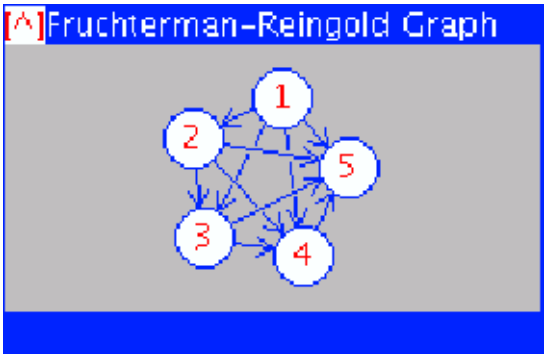

Figure 7: Example of Fruchterman-Reingold Graph layout