Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

# Extending the Eiffel Library for Data Structures and Algorithms: EiffelBase

**Master Thesis**

Olivier Jeger
jeger@computerscience.ch

**Supervising Professor:** Prof. Dr. Bertrand Meyer
**Supervising Assistant:** Dr. Karine Arnout

Chair of Software Engineering
Department of Computer Science
ETH Zurich

October 2004

# Abstract

EiffelBase is intended to be a general, high-quality library covering the basic needs of everyday programming. Many different data structures are provided, as well as algorithms operating on those data structures. The library design dates back to 1985 in its first form.

In this project, the EiffelBase library is extended in a number of areas not yet covered. Library classes for graphs, B-trees, topological sort and union-find are added. The main focus is on the design of the graph classes.

The challenge is to stay at the same level of quality as the previous parts of EiffelBase and to fit the new classes well into the existing class hierarchy.

# Zusammenfassung

EiffelBase ist eine generische Bibliothek mit dem Ziel, qualitativ hochstehende Komponenten für den Programmieralltag bereitzustellen. Viele verschiedene Datenstrukturen stehen zur Verfügung, sowie Algorithmen, welche auf diesen Datenstrukturen operieren. Das erste Design der Bibliothek reicht zurück ins Jahr 1985.

In dieser Arbeit wird die EiffelBase Bibliothek in mehreren Bereichen erweitert. Es werden Klassen für Graphen, B-Bäume, topologisches Sortieren sowie Union-Find hinzugefügt. Das Augenmerk wird dabei hauptsächlich auf das Design der Graph-Klassen gerichtet.

Die Herausforderung besteht darin, auf dem gleichen Qualitätsniveau wie die bisherigen Teile von EiffelBase zu bleiben und die neuen Klassen gut in die bestehende Klassenhierarchie einzubinden.

# Table of Contents

# Introduction

The advantages of software libraries are evident: Reusing predefined components saves valuable project time. Components that have been proven to be correct or that have been intensively tested ensure an error-free execution, which is a gain for the client both in software quality and in time. A convenience of libraries is that they can be nested: large components can be composed of smaller parts.

Time is a valuable resource for every software project. It must not be wasted with repeated development of the same components over and over again. With a library of powerful, generic and high-quality components, the programmer can focus the "real" problem instead of the surrounding algorithms and data structures.

The requirements on a library are demanding. The library classes must be generic enough to be reused in many applications. On the other hand, they must be specific enough to also cover special needs of the client. The supplied algorithms and data structures have to be powerful, but still easy to use. Every single component is required to be of the highest quality. A deficiency would have impacts on many projects; finding the error would consume valuable development time. The library designer should keep the "Open-Closed principle" in mind. After the library is delivered, the class interfaces should not be changed anymore. However, the library should be extendible to allow future enhancements. Last but not least, a certain efficiency must be achieved to make a library usable.

A big problem for the library designer is that he does not know the client and its needs. For certain problems, a design choice is good, for other problems it is bad. Many design decisions made by the library provider are not clearly "right" or "wrong", it depends on the point of view.

The original design of the EiffelBase library dates back to 1985, but it is still used in many new applications. It has stood the test of time, which is evidence of its solidity. EiffelBase has been designed as a general, high-quality library to be used for everyday programming. Together with the algorithms and data structures, the client gets a large amount of contracts. They can help him use the different components correctly, which is another step towards high-quality software.

In this project, we make four different extensions to the EiffelBase library:

1. A graph powerful graph library suited for many applications
2. B-trees as an implementations of balanced search trees
3. Topological sort
4. Union-find

The challenge of this project was to keep up the same quality level as the previous parts of EiffelBase. A lot of time was invested in the design phase to make the result as good as possible. No satisfying result can be achieved when starting with an inadequate design.

The main focus was on the design of the graph library. Existing solutions were analyzed and used as input for the final design. Besides the functionality of the classes, the ease of use was also considered.

# Chapter 1

# Graph library

## 1.1 Motivation for a graph library

There are many applications in computer science that can be modeled with a graph. The requirements for every problem are slightly different. Often, a specific graph implementation is made for each problem separately. Precious project time is consumed to build up a correct graph implementation from scratch. It is needless to say that this is not good practice. The programmer should pay full attention to the real problem, not to the surrounding data structure. Our goal is to build a reliable, well-designed graph library that is flexible enough to satisfy the needs for many different projects.

There are a lot of applications that can benefit from a graph library, both "real life" problems and problems from the theory of computation:

- Modelling the public transport system of a city
- Finding critical task dependencies in project management
- Finding cycles in module dependencies
- Implementing finite automata with directed graphs
- Computing shortest path and minimum spanning trees
- Modelling the travelling salesman problem

## 1.2 Graph theory

The graphs we are considering here are graphs from discrete mathematics. A graph consists of *nodes* (*vertices*) and *edges* which are connections between the nodes. An edge that connects a node to itself is a *loop*. The number of edges attached to a node is called *degree*. In directed graphs, there is a distinction between *in-degree* and *out-degree* which denote the number of incoming and outgoing edges respectively.

Generally speaking, the term *graph* is used to denote an undirected, unweighted simple graph. Those terms are explained briefly in the following sections. We consider only "classic" graphs, not hypergraphs. In hypergraphs, an edge can connect more than two nodes simultaneously.

### 1.2.1 Simple graphs and multigraphs

The word *graph* is often used as a synonym for *simple graphs*. In a simple graph, two vertices can be directly connected by at most one edge. Multigraphs do not have this restriction, they can have multiple edges between two nodes.

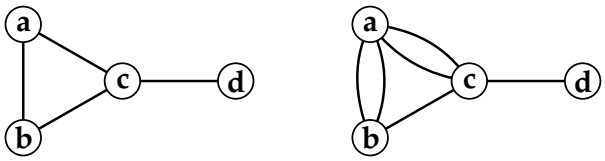



***Figure 1.1:*** *Simple graph and multigraph*

### 1.2.2 Undirected, directed and symmetric graphs

The terms *undirected* and *directed* refer to the edges. Undirected edges can be traversed in both directions, whereas directed edges are only allowed to be passed in the indicated direction. A special case of directed graphs are *symmetric* graphs. In such graphs, every edge that connects two disjoint nodes has a counterpart which points in the opposite direction. Graph loops do not have a counterpart as this could break the simple graph condition.

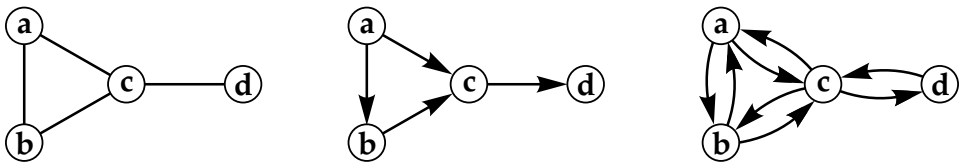



***Figure 1.2:*** *Undirected graph, directed graph and symmetric graph*

### 1.2.3 Weighted graphs

The edges of a *weighted* graph have an additional numeric attribute. The weight usually refers to the "length" of an edge or the "cost". A possible application of weighted graphs is to find the shortest path from one node to another (see also section 1.2.4). For most applications, the edge weights are considered to be positive.

### 1.2.4 Paths and cycles

A *path* is a sequence of connected edges: the end node of an edge is equal to the start node of the subsequent edge in the path. All nodes of a path are mutually distinct. A "path" where the first node and the last node are identical is called a *cycle*. Graph cycles are also referred to as *circuits*.

A common problem for weighted graphs is to find the shortest path between two nodes, where the length of a path is the sum of all its edge weights.

### 1.2.5 Connectedness, components and reachability

A node is *reachable* from another one if there exists a path between the two nodes. If all nodes can be reached from any other node, a graph is called *connected*. Otherwise, it

consists of several *components*. Each component is itself connected, but cannot be reached from within another component.

Directed graph edges can only be traversed in one direction. This leads to an additional distinction between *weak* and *strong* connectedness. A strongly connected directed graph is defined in the same way as a connected undirected graph: every node in a component must be reachable from all other nodes in the same component. The weak connectedness takes the edge direction into account. A graph is called weakly connected if there exists a component $y$ which can be reached from component $x$, but there is no connection back from $y$ to $x$. The graph on the left in figure 1.3 shows a weakly connected graph: starting at node $d$, we can reach every other node, but $d$ cannot be reached from $a$, $b$ or $c$. The addition of an edge from $c$ to $d$ leads to a strongly connected graph.

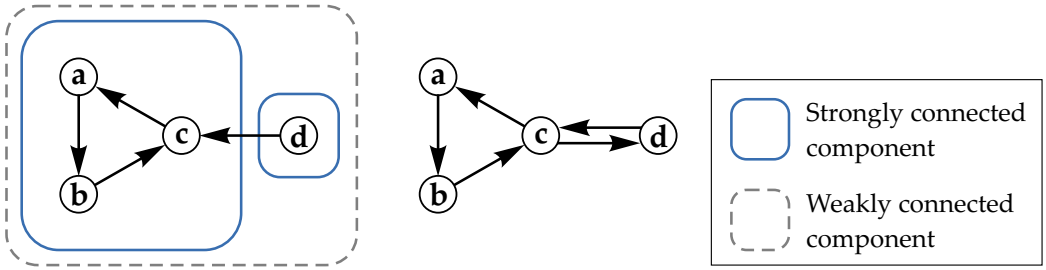



***Figure 1.3:*** *Weakly and strongly connected graph*

### 1.2.6 (Minimum) spanning trees

A connected undirected graph containing no cycle is called a *tree*. There exist several other, equivalent tree definitions:

- *Path uniqueness*: For every two vertices $x$ and $y$, there exists exactly one path from $x$ to $y$
- *Minimal connected graph*: Removing a single edge breaks the tree into two components
- *Maximal graph without cycles*: Adding an edge between any two tree nodes $x$ and $y$ leads to a cycle
- *Euler's formula*: A tree with $n$ nodes has $n - 1$ edges

For every connected simple graph, there exists at least one *spanning tree*. It contains the same node set as the original graph, but only a subset of its edges. A spanning tree is the graph with the minimum number of edges, such that all nodes of the original graph are connected.

In general, there are multiple spanning trees for a given graph. If we are dealing with weighted graphs, a common problem is to find the *minimum spanning tree*, which is the spanning tree with the minimal sum of all edge weights.

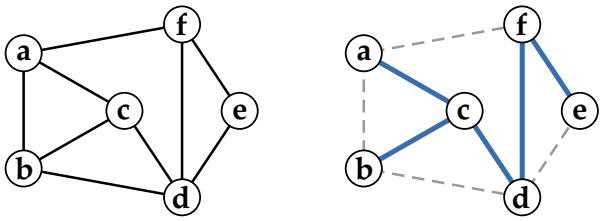



***Figure 1.4:*** *Undirected simple graph and one of its spanning trees*

## 1.3 Bernd Schoeller's solution

### 1.3.1 Overview

In 2003, Bernd Schoeller developed a graph library for Eiffel [1]. The goals were ambitious: the library was intended to be generic, powerful, efficient, easy to use and in the spirit of EiffelBase. It turned out to be hard to satisfy all those requirements at the same time.

A difficult issue when building a graph library is to reconcile the different graph properties which appear to be orthogonal. The most important among them are simple graph versus multigraph, undirected graph versus directed graph and unweighted graph versus weighted graph. The approach chosen by Bernd Schoeller was to encapsulate each concept in a class. Thus, combinations can be achieved by using multiple inheritance. The drawback of this method is the exponential growth of the number of classes. For every new concept, the number of classes is doubled.

Like many other data structures in EiffelBase, the class hierarchy is divided into a structural part and an implementation part. There are two implementations: the first one is based on an adjacency matrix, the second one operates on a dynamically linked structure of nodes and edges. There is no implementation of undirected graphs and the adjacency matrix implementation is not complete. Figure 1.5 shows the class diagram, divided into the clusters *structures* and *implementations*.

### 1.3.2 Representation of nodes and edges

Graph classes take one generic argument which is the node type. The library is designed such that all operations are performed directly on the graph. There are objects for the internal representation of the nodes, but the user has no access to them. In a graph with nodes of type *STRING*, the query *item* returns the current node which is a *STRING* value, not a *NODE* object containing the value. Hence the user must always query the graph to get informations like the degree of a node.

Edges do not have any attributes, neither labels nor weights. They are just links between two nodes. The library is designed to support both undirected and directed graphs. But so far, only directed graphs have been implemented.

### 1.3.3 Cursors and traversal

Like many container data structures in EiffelBase, graphs can be traversed and have a cursor. The cursor concept is a bit special in this case since there are two kinds of objects in a graph: nodes and edges. We want a possibility to visit the nodes and also to walk along their incident edges to explore the graph. A graph cursor is the combination of a

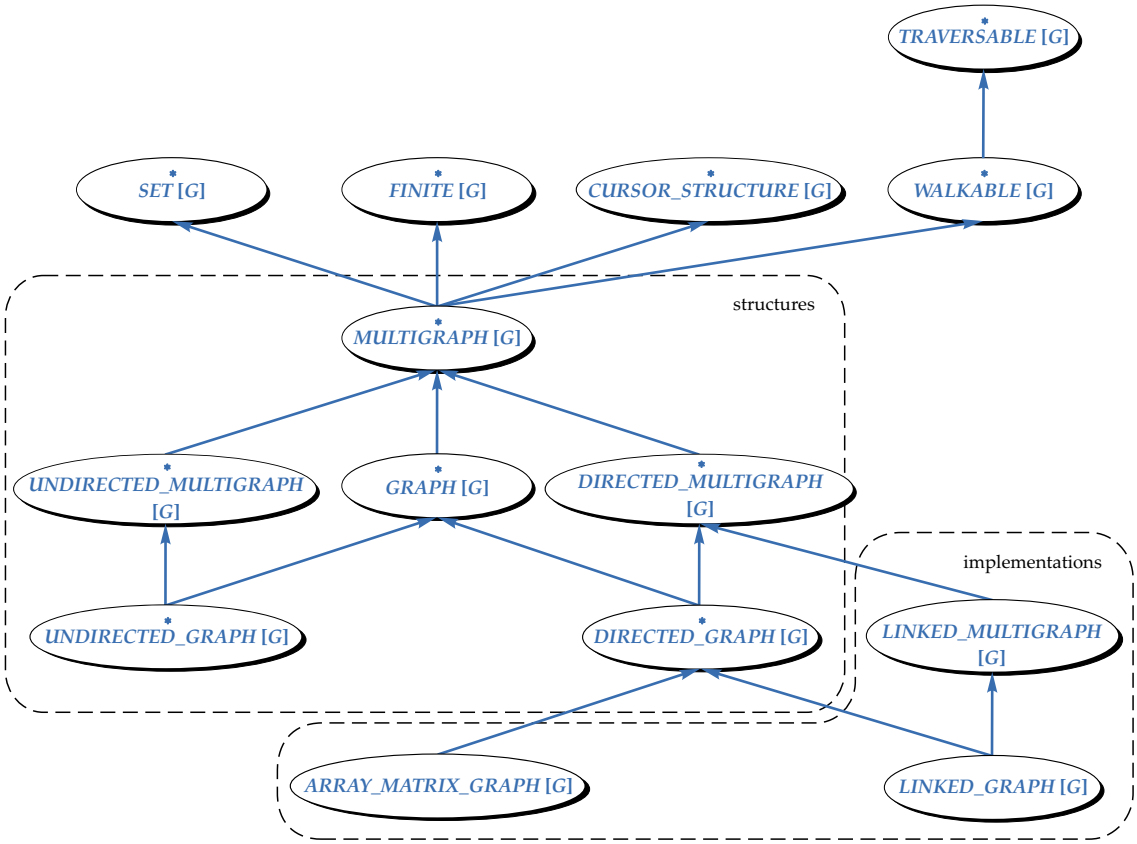



***Figure 1.5:*** *Bernd Schoeller's class hierarchy*

node and one of its incident edges (if there is any). This allows us to iterate over both nodes and edges. Figure 1.6 shows an example:

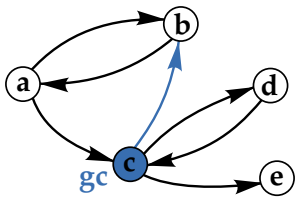



***Figure 1.6:*** *Graph cursor* gc *positioned at node* c*, pointing to target* b

When the cursor is positioned at a node, it can be "turned" with the commands *left* and *right* to focus another outgoing edge. The flag *exhausted* indicates that the cursor went around and has reached the first edge again. Invoking command *forth* moves the cursor along the currently focused edge to the target node (figure 1.7):

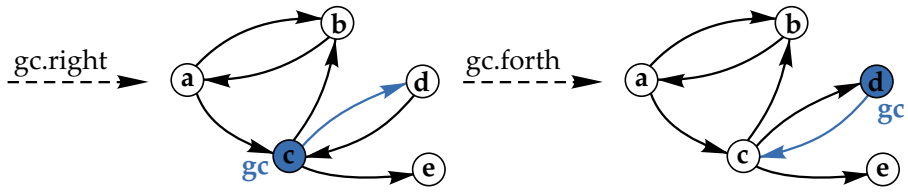



***Figure 1.7:*** *Cursor modifications: turning and walking*

All these features are inherited from the class *WALKABLE*. The current cursor position is accessed through the query *cursor* and can be stored in a variable. To restore a saved cursor position, the *go_to* command is used. It is not mandatory for the *go_to* command that the given cursor position is reachable from the current node.

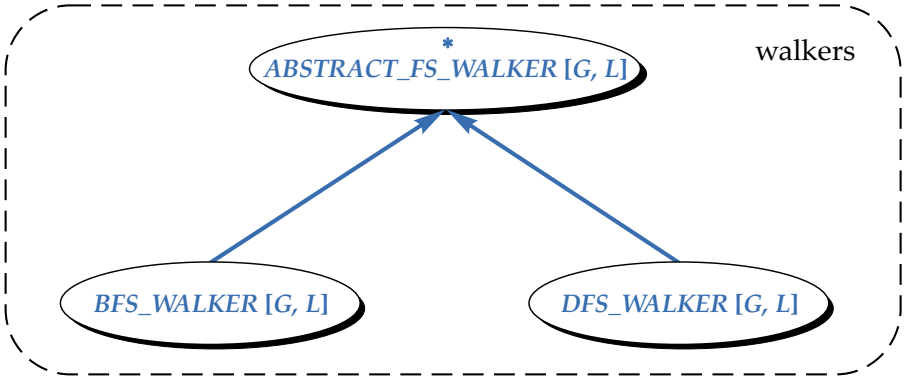



***Figure 1.8:*** *Cluster* walkers *with BFS_WALKER and DFS_WALKER*

Besides directly using the cursors provided by the *GRAPH* class, there is another possibility to traverse a graph. The classes *BFS_WALKER* and *DFS_WALKER* explore the graph in a breadth-first-search and depth-first-search strategy respectively. Internally, the classes make use of graph cursors, whereas the user has only access to the command *forth*. Depending on the strategy, the walker places the graph cursor automatically at the appropriate position.

Both classes have *ABSTRACT_FS_WALKER* as common ancestor. The traversal technique is quite simple: starting from a node, all neighbors that have not been visited so far are added to a container. The cursor is moved to the first node of that collection which is then marked as visited. This procedure is repeated until all nodes have been processed. There is only one small difference in the implementation of the breadth-first-search and depth-first-search approaches: the type of the node container. *BFS_WALKER* uses a *queue* to store the nodes that must be visited, *DFS_WALKER* uses a *stack* instead.

### 1.3.4 Implementation 1: *ARRAY_MATRIX_GRAPH*

The class *ARRAY_MATRIX_GRAPH* implements directed simple graphs, based on an adjacency matrix. Contrary to linked graphs, this implementation has not been finished and thus lacks some functionality. The only supported node type is *INTEGER*. The adjacency matrix is a boolean two-dimensional array of equal height and width. An edge between the nodes $a$ and $b$ implies that the value of *adj_matrix* [$a$, $b$] is *true*, otherwise

*false*. With this simple approach, it is not possible to support multigraphs. The integer nodes are directly used as matrix indices, which can cause problems if the values are negative or non-contiguous. For instance, the adjacency matrix for a graph with only one single node with value 100 has already 10.000 entries. Probably, the approach is too much simplified for real life usage.

A big advantage of the adjacency matrix approach is the efficiency of the data access. Most operations are very fast, especially queries to find out whether two nodes are connected or not. Unfortunately, working with an adjacency matrix does not scale for large, and especially not for sparse graphs. The matrix grows to the square of the number of nodes. If there are only a few edges, only a small part of the matrix contains useful information, the rest is wasted.

As stated before, this implementation is not very elaborate and should mainly demonstrate the feasibility of an adjacency matrix implementation.

### 1.3.5 Implementation 2: *LINKED_GRAPH*

The linked graph implementation is more powerful than the adjacency matrix graph. The nodes can be of an arbitrary type, not just *INTEGER*. The idea of this approach is to maintain a list of incident edges for each node. Internally, the node values are wrapped in a *LINKED_GRAPH_NODE* object. Each *LINKED_GRAPH_NODE* object holds a doubly linked list of all its incident edges, which are of type *LINKED_GRAPH_EDGE*. The *LINKED_GRAPH_NODES* in turn are arranged as a linear list.

Figure 1.9 on the following page shows an example graph with four nodes and the corresponding internal representation. In the left grey box, we can see the the list of *LINKED_GRAPH_NODE* objects, the right side shows the incident edges for each node.

This linked structure grows only in a linear way with the number of nodes and edges. It is much better suited for graphs with only few edges compared to the amount of nodes. However, this implementation is not perfect either. Many operations on a linked graph are obviously quite inefficient: The linked structure needs always to be traversed to access the concerned objects. Because no arrayed data structures or hash tables are used, only linear search is applicable. For large graphs, this drawback is clearly noticeable.

## 1.4 Final design of the graph library

### 1.4.1 Overview

Studying the approach of Bernd Schoeller gave a nice overview over the difficulties when designing a graph library. Although many concepts have been modeled nicely, the whole project was either not powerful enough or became cumbersome for the user. It turned out that the encapsulation of every concept in a new class was blowing up the class hierarchy too much. There are already a lot of deferred classes; we must keep in mind that all classes show up in each implementation again.

The approach for our implementation was rather to have a mix of inheritance and boolean properties which are set at object creation time. The resulting design has four deferred graph classes: *GRAPH*, *UNDIRECTED_GRAPH*, *WEIGHTED_GRAPH* and *UNDIRECTED_WEIGHTED_GRAPH*. In addition, the notion of *edge* shall no longer be

hidden from the client and leads to the classes *EDGE* and *WEIGHTED_EDGE* (figure 1.10).

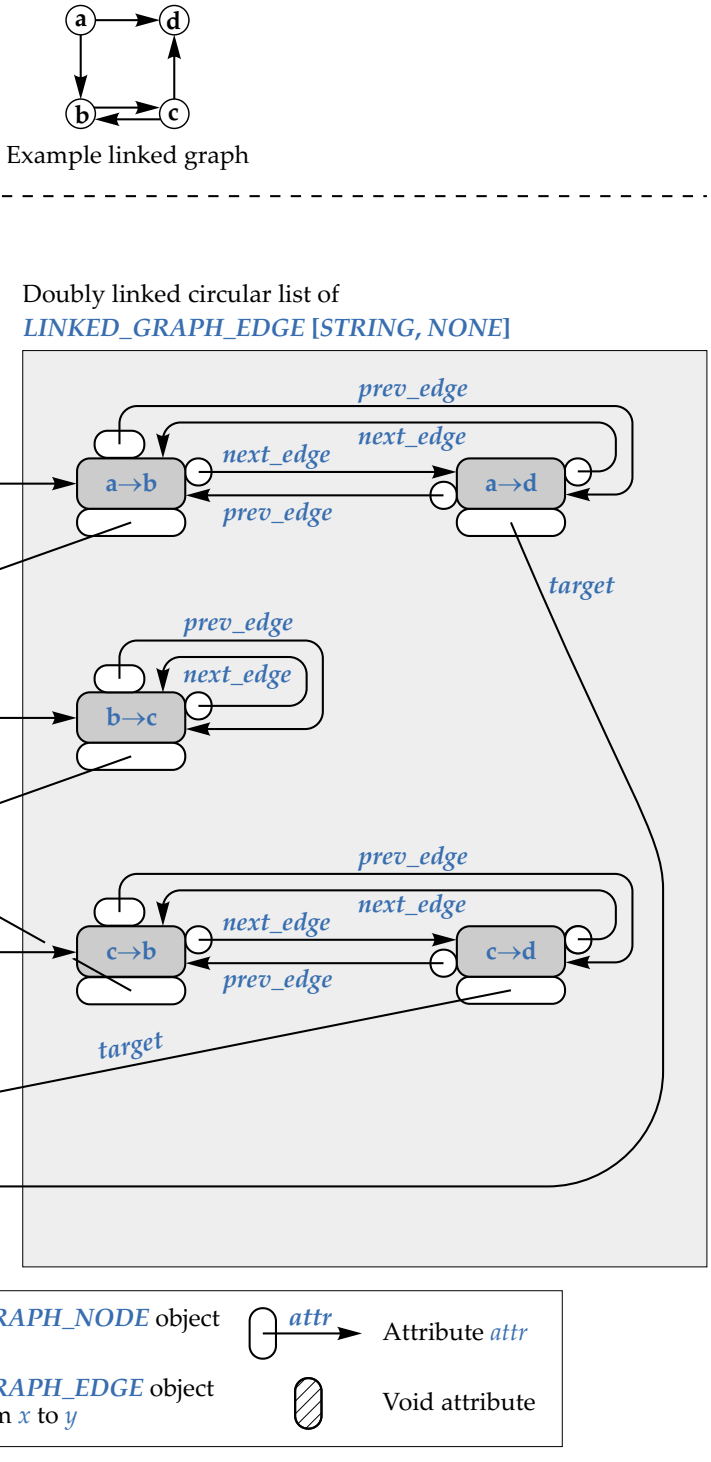



***Figure 1.9:*** *Example directed graph and its internal representation*

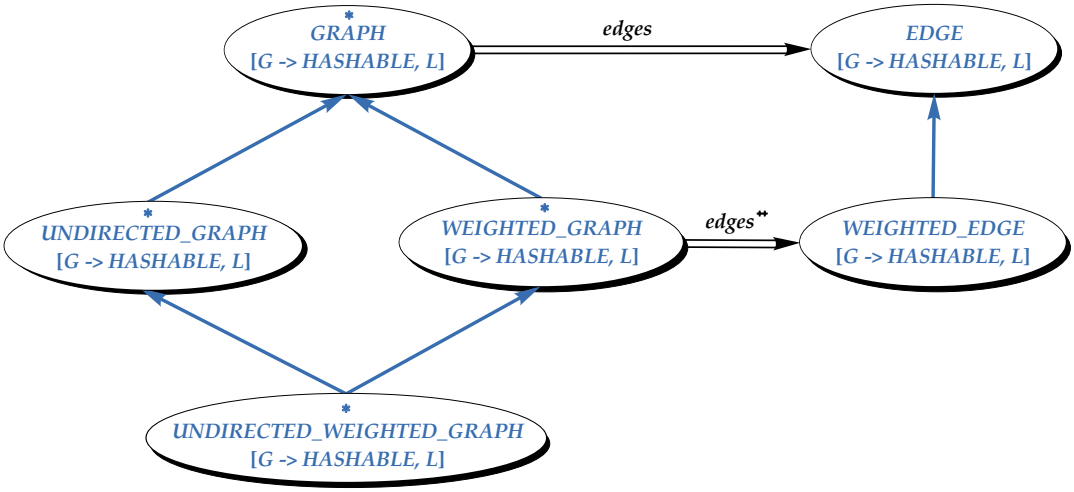



***Figure 1.10:*** *Structure of the graph library*

All graph classes have two generic parameters. The first one denotes the node type, the second one is the type of the edge labels. Those labels can be of an arbitrary type and are used to describe the edges. They must not be confused with edge weights, which are an additional numeric attribute on weighted edges (see section 1.4.3).

The decision whether to have a simple graph or a multigraph is made at creation time. For instance, the class *LINKED_DIRECTED_GRAPH* offers the following creation routines:

| | |
|---|---|
| *make_simple_graph*: | Create a simple graph with at most one edge between two nodes |
| *make_symmetric_graph*: | Create a simple graph where each edge has a counterpart in the opposite direction |
| *make_multi_graph*: | Create a graph with an arbitrary amount of edges between two nodes |
| *make_symmetric_multi_graph*: | Create a graph with an arbitrary amount of edges, all present in both directions |

This new class design leads to a couple of advantages:

- Only four classes have to be written for a new implementation.
- It is easier for the user to decide which class to use because there are not so many classes.
- Algorithms which require to operate on edges are more intuitive to implement due to the availability of *EDGE* objects.

### 1.4.2 Graph nodes

From the user point of view, graph nodes are not encapsulated in an additional *NODE* object. A graph is intended to be a collection of items of type *G* and connections (edges)

between them. It is similar to a linked list, where the user is only interested in the list items, not in the *LINKABLE* objects.

Because several operations take a node item as argument, we need a way to uniquely identify them. Thus, it is not possible to have multiple nodes with the same value in the graph. This property is reflected by the fact that the class *GRAPH* [*G*, *L*] inherits from *SET* [*G*]. The nodes are identified by their value, not by their reference. The flag *changeable_object_comparison* is immutably set to *false* and *object_comparison* is *true*.

Graph nodes cannot be replaced after creation. Otherwise, the client could break the *SET* property by assigning the same value to multiple nodes. To maintain consistency, the objects of type *G* should not be changed after insertion in a graph either.

For consistency reasons, the queries *nodes* and *edges* return only a *copy* of the node and edge set respectively. Otherwise, the user could add, remove or replace any element in either of those collections without any notice to the graph, and the internal data structure would get corrupted.

### 1.4.3 Edges

In discrete mathematics, a graph is defined as a set of nodes and a set of edges. Following that principle, the current library provides also access to objects of type *EDGE*. An edge comprises a start node, an end node and a label. Both nodes are immutable, so it is impossible to redirect an edge.

The label can be of any type and is used to describe the edge. Most commonly, strings are used as edge labels, but it is also possible to use more complex data types. It is even possible to derive the edge weight from the label object (see also section 1.4.4).

Contrary to the graph classes, there are no distinct classes for directed and undirected edges. Obviously, a directed graph contains only directed edges and vice versa. A flag *is_directed* is used to indicate the current status. For instance, this flag is used internally when two edges are compared. Directed edges must point in the same direction to be equal, undirected edges not necessarily.

Considering an undirected edge, it is not clear which node is the start node and which is the end node. Therefore, a further query was introduced for undirected edges only: *opposite_node* takes a node as argument and returns the item at the opposite end of the edge.

### 1.4.4 Weighted edges

The edges of weighted graphs have an additional attribute: a numeric weight. In our graph library, there is a special edge type called *WEIGHTED_EDGE* which defines the attribute *weight* of type *REAL*. The edge weight is a constant value that is assigned when the edge is inserted into a graph by a *put_edge* command. At any time, the weight can be changed by invoking *set_weight* on the corresponding *EDGE* object.

There is a special feature in class *WEIGHTED_EDGE*: it is possible to use another measurement for the edge weight than the attribute *weight*. The user can define a function which computes the edge weight from the current value of the edge label. An arbitrary attribute or function which returns a *REAL* value can be used for that purpose. This can be useful when there are multiple measurements that can be applied to an edge. Imagine a traffic environment where you could store the distance between two locations and the

travelling time in the edge label. You may want to know the shortest distance between two locations or how to reach a location in minimum time. Instead of calling *set_weight* on each edge individually, the edge measurement can be exchanged for the whole graph with one single command. In section 1.7.5, a detailed example with code extracts is given.

### 1.4.5 Graph cursors and traversal

The cursor concept in the graph library is similar to the one used in Bernd Schoeller's implementation (see section 1.3.3). A graph cursor consists of a node and one of its incident edges, if there is any. You can set the cursor position to any node in the graph. Using the commands *left* and *right*, the cursor can be "turned" towards the different incident edges. The command *forth* moves the cursor along the currently focused edge.

One addition compared to Bernd Schoeller's solution: our implementation keeps track of the traversal history and adds a *back* command. It retracts the last step made by a *forth* command. Because any previously visited node or edge may have been removed in the meantime, the query *has_previous* provides information whether the *back* command can be invoked or not.

The graph cursor can be placed at any position in the graph by using the commands *search* and *go_to*. The first one takes a node, the latter a cursor as argument. For both commands, it is not necessary that the given position is reachable from the current node.

The graph walkers are also adopted from Bernd Schoeller's implementation. You have the ability to traverse the graphs using the depth-first-search or breadth-first-search strategy.

### 1.4.6 Implemented graph algorithms

#### Component count

The query *components* returns the number of (weakly connected) graph components. For the implementation, we can benefit from our new union-find data structure which is described in chapter 4. The algorithm to count the components works as follows: after putting all nodes into unary sets, we iterate over all graph edges. If the end nodes of an edge are in different sets, those are merged because we are in the same component. The number of sets at the end of the process will be the number of graph components.

```
Initialize uf as empty union-find structure.
for all graph nodes x do
    uf.put (x)
end
-- All nodes are now stored as unary sets in 'uf'.
for all graph edges e do
    set1 := uf.find (e.start_node)
    set2 := uf.find (e.end_node)
    if set1 /= set1 then
        -- Components are connected: Merge them.
        uf.union (set1, set2)
    end
end
Result := uf.set_count
```

### Node reachability

To find out whether a node $y$ is reachable from node $x$, we could use a backtracking algorithm. In our graph library, we use an approach that makes use of matrix multiplication. Consider the matrix *adj* to be the integer equivalent of an adjacency matrix: when two nodes are connected, the corresponding matrix entry is 1, otherwise 0. This initial matrix can be interpreted as representation of "all paths of length 1".

The surprising property comes up when *adj* is multiplied with itself. The non-zero entries in $adj^2$ represent the paths of length 2. Similarly, the non-zero entries of $adj^n$ correspond to the paths of length $n$.

To find out whether $y$ is reachable from $x$, we have to examine the corresponding entry in *adj*. As soon as the value is positive in $adj^i$ for some $i \geq 1$, a path exists. For a graph with $m$ edges, we need to compute $adj^m$ in the worst case. Matrix multiplications can be done in polynomial time, a backtracking algorithm would have exponential complexity.

### Cycle detection

The query *has_cycles* indicates whether a graph contains cycles or not. The different kinds of graphs allow different methods to check for cycles.

For undirected graphs, we can use Euler's formula: an undirected simple graph contains one or several cycles, if $|edges| > |nodes| - components$.

For directed graphs, we use the algorithm for the node reachability and check if there is a path of length greater than zero connecting one of the nodes to itself.

Currently, it is not possible to check if an undirected multigraph contains cycles. The routine *has_cycles* returns *false* in such a case and a message is printed to the screen.

### Minimum spanning tree

Finding the minimum spanning tree for an undirected weighted graph is another elegant application of the union-find data structure. J.B. Kruskal has proposed the following algorithm [2]:

```
Initialize union-find structure uf.
Make empty mst object in which the result is stored.
for all graph nodes x do
    uf.put (x)
end
--All nodes are now stored as unary sets in 'uf'.
Store all graph edges in edges, sorted by increasing weight.
for all e in edges do
    set1 := uf.find (e.start_node)
    set2 := uf.find (e.end_node)
    if set1 /= set1 then
        --Edge connects two disjoint parts of the graph.
        Add e to mst.
        uf.union (set1, set2)
    end
end
Result := mst
```

## 1.5 Limitations

### Cycle detection

Currently, the graph library cannot answer the queries *has_cycles* for undirected multigraphs. For simple graphs, Euler's formula can be applied which states that an undirected graph has cycles if $|edges| > |nodes| - components$. For multigraphs, this formula cannot be applied anymore since multi-edges are not considered to be graph cycles. To cover all possible multigraph scenarios, it would be necessary to check each graph component for cycles individually. This is not possible with the current implementation.

### Adjacency matrix graph has no multigraph support

The implementation that is based on an adjacency matrix does not support multigraphs. Usually, adjacency matrices contain boolean values which refer to whether two nodes are connected or not. In our implementation, the item is not a boolean value, but an *EDGE* object which represents the connection. In case there is no connection between two nodes, the item is *void*. With that approach, only simple graphs can be supported.

### Class hierarchy

A rather subtle issue is that *UNDIRECTED_GRAPH* is a subtype of *GRAPH*. An algorithm operating on a directed graph could produce unexpected results if the actual argument is an undirected graph, containing edges that can be traversed in both directions. For instance, an algorithm that detects graph cycles in a directed graph would find a cycle in an undirected graph with just one single edge, because there is a connection from the first node to the second and vice versa. In a directed graph, this is a cycle, in an undirected graph, it is not.

### Edge traversal using graph walkers

Currently, the graph walkers keep only track of the visited nodes, but not of the visited edges. With that approach, you can explore all *nodes* according to the selected strategy, but the edges are not taken into account. Algorithms that operate on *edges* and need a specific traversal strategy cannot work with the provided graph walkers. When an edge traversal is supplied additionally, the *walker* interface needs to be reconsidered.

### Unstable implementation of used library class

The *LINKED_GRAPH* implementation may raise contract violations at runtime. It is highly probable that they are not directly related to the graph library, but to the class *TWO_WAY_CIRCULAR*. Internally, the incident edges of a node are stored in such a doubly linked circular list. Either some contracts of *TWO_WAY_CIRCULAR* are not reliable, or the implementation contains errors. For our test cases, it was sufficient to turn the postcondition checking off for the corresponding cluster. However, this class should be revised and corrected.

## 1.6 Problems related to Eiffel and EiffelStudio

### 1.6.1 *WEIGHTED_EDGE* cannot inherit from *COMPARABLE*

Some algorithms like finding the shortest path require the edge set to be sorted by ascending weights. What you would do normally is to put all edges in a predefined container such as *SORTED_LIST*. Therefore, the class *WEIGHTED_EDGE* was designed as a descendant of *COMPARABLE* and the $<$ operator was defined to compare the edge weights. Surprisingly, the routine *is_equal* of the class *WEIGHTED_EDGE* raised a contract violation with that setup: according to *is_equal* in *COMPARABLE*, two objects are equal when neither of them is greater than the other. In other words, the contract states that two edges have to be equal when their weights are equal. Of course this is not true because two edges can have the same weight but different labels or start and end nodes. That specific contract made it impossible to use any predefined sorted container. The only way this problem could be solved was to use an unsorted list together with a hand-made implementation of insertion sort.

### 1.6.2 Non-deterministic precursor

Undirected graphs are a subtype of directed graphs. Since its edges are undirected, it makes no sense to have both features *in_degree* and *out_degree* anymore. They are merged into the feature *degree*. To compute the result, the **Precursor** is called since it is not necessary to redefine the implementation. However, it is not clear whether *in_degree* or *out_degree* is called, because there is no possibility to choose a precursor feature. Resolving this indeterministic behavior is left to the compiler writer and is thus a design deficiency of the Eiffel language.

### 1.6.3 Technical problems in EiffelStudio 5.4

The compiler of EiffelStudio 5.4 [3] crashes when using an expanded type as label type. EiffelStudio quits with an error message and it is not possible to complete the compilation process in such a case. The only solution is to use the appropriate reference type, for example *INTEGER_REF* instead of *INTEGER*.

## 1.7 User guide

### 1.7.1 Introduction

The following sections contain examples how the graph library is used. For most steps, the corresponding Eiffel code is listed as well. We start by giving an overview over the different graph classes and explain the basic operations. The complexity of the examples increases step by step.

### 1.7.2 Choice of the graph class

The first step you have to do when using the graph library is to choose among the different graph classes. There are two implementations, one is based on an adjacency matrix, the other one uses a linked data structure. Currently, the *ADJACENCY_MATRIX_GRAPH* implementation supports only simple graphs. Most of its operations are quite fast since the access to both nodes and edges is efficient. Be aware that the adjacency matrix grows to the square of the node amount. When only few edges are present in a graph with many nodes, a lot of memory remains unused.

The *LINKED_GRAPH* implementation supports any type of graphs. When dealing with multigraphs, you should use this version. Some operations may not be as fast as on adjacency matrix graphs, since the edges are arranged as an incidence list. Traversing those lists may take some time.

For both implementations, four graph classes are available:

- *XXX_GRAPH*: directed simple or multigraphs
- *XXX_UNDIRECTED_GRAPH*: undirected simple or multigraphs
- *XXX_WEIGHTED_GRAPH*: directed weighted simple or multigraphs
- *XXX_UNDIRECTED_WEIGHTED_GRAPH*: undirected weighted simple or multigraphs

where the prefix *XXX* stands either for "*ADJACENCY_MATRIX*" or for "*LINKED*". All graph classes have two generic parameters. The first one denotes the node type, the second one is the label type. Those labels are used to describe and to identify the edges.

The second choice concerns the creation routine. The *LINKED_GRAPH* implementation offers four alternatives:

| | |
|---|---|
| *make_simple_graph*: | create a simple graph with at most one edge between two nodes |
| *make_symmetric_graph*: | create a simple graph where each edge has a counterpart in the opposite direction |
| *make_multi_graph*: | create a graph with an arbitrary amount of edges between two nodes |
| *make_symmetric_multi_graph*: | create a graph with an arbitrary amount of edges, all present in both directions |

Since the class *ADJACENCY_MATRIX_GRAPH* does not support multigraphs, you can only choose between *make_simple_graph* and *make_symmetric_graph*.

### 1.7.3 Basic operations

A new graph node is added to a graph with the command *put_node*. Like in a *SET*, it is not possible to have the multiple nodes with the same value in the graph. Repeated calls to *put_node* with the same argument are ignored.

Edges are added using the *put_edge* command. For the edges, the situation is different. If you work with a multigraph, you can put as many edges between two nodes as you want. For simple graphs, an appropriate precondition of *put_edge* command ensures to have at most one connection between two nodes.

Note that the argument list of *put_edge* is different for weighted graphs and unweighted graphs. The *put_edge* command in *WEIGHTED_GRAPH* takes an additional argument of type *REAL*, which denotes the edge weight.

A node is removed from the graph by calling *prune_node*. The library takes care that no dangling edges will arise. All incident edges of the node are automatically be removed as well.

To remove an edge, you should call *prune_edge* with an *EDGE* object as argument. You can get a reference to an *EDGE* object if you know all edge attributes. In such a case, you can use the query *edge_from_values*. It takes the two end nodes and the edge label as argument and returns the corresponding edge. If you are working on a weighted graph, you must also supply the edge weight.

Let's start with a small example: an undirected simple graph as shown in figure 1.11.

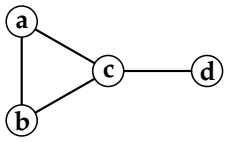



***Figure 1.11:*** *Simple undirected graph*

We choose the class *ADJACENCY_MATRIX_UNDIRECTED_GRAPH* to demonstrate the mapping to Eiffel:

```
build_graph is
        -- Build an example graph.
    local
        graph: ADJACENCY_MATRIX_UNDIRECTED_GRAPH [STRING, NONE]
    do
        -- Create the graph.
        create graph.make_simple_graph

        -- Put the nodes into the graph.
        graph.put_node ("a")
        graph.put_node ("b")
        graph.put_node ("c")
        graph.put_node ("d")
```

```
        -- Connect the nodes with edges.
        graph.put_unlabeled_edge ("a", "b")
        graph.put_unlabeled_edge ("a", "c")
        graph.put_unlabeled_edge ("b", "c")
        graph.put_unlabeled_edge ("c", "d")
    end
```

If you want to have labeled edges, you can use the following code:

```
build_labeled_graph is
        -- Build an example graph with labels.
    local
        graph: ADJACENCY_MATRIX_UNDIRECTED_GRAPH [STRING, STRING]
    do
        -- Create the graph and put the nodes into it.
        -- (same as above)

        -- Connect the nodes with labeled edges.
        graph.put_edge ("a", "b", "a-b")
        graph.put_edge ("a", "c", "a-c")
        graph.put_edge ("b", "c", "b-c")
        graph.put_edge ("c", "d", "c-d")
    end
```

The following graph will be the result:

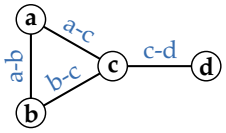



***Figure 1.12:*** *Undirected graph with labeled edges*

### 1.7.4 Directed and symmetric graphs

The graph in the previous example was undirected. The edges of a *directed graph* can only be traversed in one direction. We can only get back if there is another edge pointing in the opposite direction. A special case of directed graphs are *symmetric graphs*. All edges have a counterpart that points in the opposite direction. Hence all nodes of a component are strongly connected. Our graph library provides support for symmetric graphs. For any edge you put into the graph, its symmetric counterpart is also inserted automatically (an exception are graph *loops* which are not duplicated). The label and optionally the weight are copied to the symmetric edge.

Figure 1.13 shows a directed graph and its symmetric equivalent. Below, the corresponding Eiffel code is listed. To get the *directed* graph, the argument to the routine *fill_graph* is a graph created with *make_simple_graph*. To get the *symmetric* graph, it must be created using *make_symmetric_graph*. You can see that the code is exactly the same as for the undirected graph. It is only the graph class and the creation routine which define the different graph shapes.

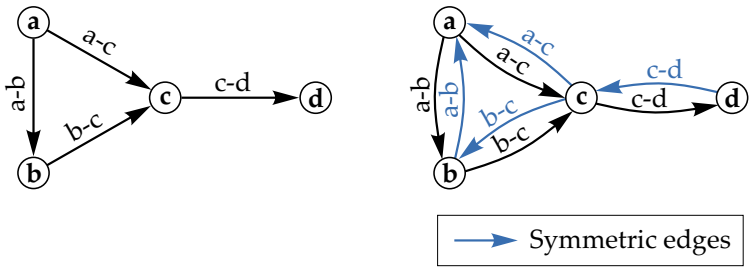



***Figure 1.13:*** *Directed graph and symmetric graph*

```
fill_graph (a_graph: ADJACENCY_MATRIX_GRAPH [STRING, STRING]) is
        -- Put some nodes and edges into 'a_graph'.
    require
        graph_not_void: a_graph /= Void
    do
        -- Put the nodes into the graph.
        graph.put_node ("a")
        graph.put_node ("b")
        graph.put_node ("c")
        graph.put_node ("d")

        -- Connect the nodes with edges.
        graph.put_edge ("a", "b", "a-b")
        graph.put_edge ("a", "c", "a-c")
        graph.put_edge ("b", "c", "b-c")
        graph.put_edge ("c", "d", "c-d")
    end
```

### 1.7.5 Weighted graphs

The edges of weighted graphs have an additional numeric attribute, which is the weight. Accordingly, weighted graphs have a different *put_edge* command. It takes the weight as additional argument of type *REAL*. Similarly, *put_unlabeled_edge* takes now three arguments. Figure 1.14 shows our example graph, now with weighted edges. Below is the corresponding Eiffel code:

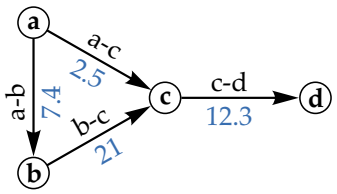



***Figure 1.14:*** *Weighted directed graph*

```
build_weighted_graph is
        -- Build an example graph with labels.
    local
        graph: ADJACENCY_MATRIX_WEIGHTED_GRAPH [STRING, STRING]
    do
        -- Create the graph and put the nodes into it.
        -- (same as in previous examples)

        -- Connect the nodes with weighted edges.
        graph.put_edge ("a", "b", "a-b", 7.4)
        graph.put_edge ("a", "c", "a-c", 2.5)
        graph.put_edge ("b", "c", "b-c", 21)
        graph.put_edge ("c", "d", "c-d", 12.3)
    end
```

The weight of an edge can be exchanged by calling *set_weight* on the corresponding *EDGE* object. If we do not yet have a reference to that edge, we can get it from the graph. The code below shows how the weight of the edge *c*-*d* is replaced by half of its original value:

```
exchange_edge_weight (graph: ADJACENCY_MATRIX_WEIGHTED_GRAPH) is
            -- Example how an edge weight is replaced.
            -- Assume 'graph' is the same as in previous example.
    local
        edge: WEIGHTED_EDGE [STRING, STRING]
        new_weight: REAL
    do
        -- Get appropriate EDGE object.
        edge := graph.edge_from_values ("b", "c", "b-c", 21)
        new_weight := edge.weight / 2
        edge.set_weight (new_weight)
    end
```

### 1.7.6 Advanced use of weighted graphs

The edge weights we have seen so far were exchangeable constant values. Another possibility is to calculate the edge weight directly from the label value. Imagine a complex label type that contains multiple measurements values for an edge. You may want to find to apply the shortest path algorithm for each of those measurement criteria.

As an example, we take a graph that represents a city map. The nodes are locations and the edges are streets that connect the locations. For each edge, the street name, the length and the time to walk along that street is stored. The class *STREET* that is used for the edge labels might look as follows:

```
class
    STREET

feature -- Access

    distance: REAL
            -- Length of the street

    travel_time: REAL
            -- Time to walk from one end to the other

    street_name: STRING
            -- Name of the street

end
```

To compute the edge weight from the label, a weight function must be defined to establish the connection between the label and the weight. The function takes a *WEIGHTED_EDGE* object as argument and returns a *REAL* value. The *WEIGHTED_EDGE* argument is the edge to which the label belongs. The weight function can be defined in any class. For our example, we define two different routines *weight_from_distance* and *weight_from_time*. The only missing step is to tell the graph not to use the stored weight, but a user-defined function to compute the edge weight. This is done using the command *enable_user_defined_weight_function*:

```
class
    MY_CLASS

feature

    replace_edge_weights (wg: WEIGHTED_GRAPH [LOCATION, STREET]) is
            -- Replace the default weights in 'wg' by values
            -- computed from the edge labels.
            -- All edge labels are assumpted to be non-Void.
        do
            -- Use the street length as edge weight
            wg.enable_user_defined_weight_function (agent weight_from_distance)
            -- Do some computations, e.g. find shortest path.
            -- ...

            -- Compute edge weights from travel time
            wg.enable_user_defined_weight_function (agent weight_from_time)
            -- Perform more computations, e.g. compute minimum spanning tree.
            -- ...
        end
```

```
        restore_weights (wg: WEIGHTED_GRAPH [LOCATION, STREET]) is
                -- Restore initial (stored) edge weights
            do
                wg.restore_default_weights
                -- Perform even more operations
                -- ...
            end

    feature {NONE} -- Weight functions

        weight_from_distance (a_edge: WEIGHTED_EDGE): REAL is
                -- Compute weight of 'a_edge' based on the length of the street.
            do
                Result := a_edge.label.distance
            end

        weight_from_time (a_edge: WEIGHTED_EDGE): REAL is
                -- Compute weight of 'a_edge' based on the travel time.
            do
                Result := a_edge.label.travel_time
            end

    end
```

As shown in the sample code, the user-defined weight functions can be deactivated by calling *restore_default_weights*. After that, the default edge weights (stored in the attribute *weight*) are used again.

### 1.7.7 Graph algorithms

To give a demonstration of the graph algorithms provided by the graph library, we use a slightly more complex graph than before:

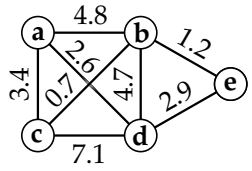



***Figure 1.15:*** *Example weighted graph*

#### Eulerian graphs

The graph shown in figure 1.15 might look familiar to you because it can be drawn in a single closed line without lifting the pencil (without the labels, of course). Such a graph is called *Eulerian* in graph theory. The query *is_eularian* is available for both undirected and directed graphs, although the implementation is different.

### Shortest path

Consider a route planning system as an example to find the shortest connection between two locations. In our example graph, we want to find the shortest path between *c* and *d*.

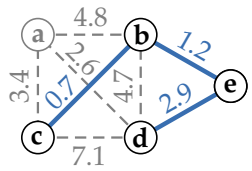



***Figure 1.16:*** *Shortest path from* c *to* d

The routine *shortest_path* takes two node items as arguments and returns the shortest path between these nodes. Because a path is a linear structure, it does not make sense to return a whole graph. The result is just a list of all edges contained in the path.

### Minimum spanning tree

In contrast to the shortest path, the minimum spanning tree is still a graph. It contains the same nodes as the initial graph, but only a subset of its edges. The minimum spanning tree is accessible on undirected weighted graphs via the routine *minimum_spanning_tree*. Figure 1.17 shows the minimum spanning tree of our example graph.

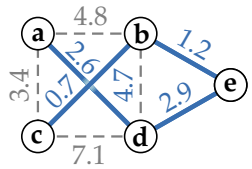



***Figure 1.17:*** *Minimum spanning tree*

## 1.7.8 Visualizing the graph

Currently, there is no Eiffel library which is capable to visualize graphs. There is a freely available third-party tool *dot* from the *graphviz* library [4] which can be used for that purpose. The *out* routine of all graph classes returns a string of compatible format which can then be stored in a text file. The tools *dot* and *dotty* are able to generate images of various formats from that text file.

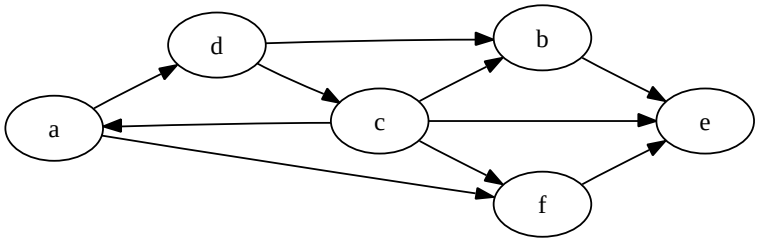



***Figure 1.18:*** *Example graph visualization, generated by the* dot *tool*

# Chapter 2

# B-trees

## 2.1 Introduction

B-trees belong to the family of *balanced search trees*. Search trees are used to implement dictionary functions, such as *put*, *has* and *remove* in an efficient way. The items (*keys*) of a search tree are sorted, i.e. all items in the left subtree are smaller or equal than the current item and the items in the right subtree are greater or equal.

The problem when using search trees is that they can degrade to linear lists and the efficiency gets lost. This is highly dependant on the insertion order of the items. Balanced trees are designed to prevent such degrading. If possible, the items are arranged such that all subtrees get the same height. If some tree operation breaks that property, the items are rearranged. We will see that in B-trees, the dictionary functions have all logarithmic time complexity, even in the worst case.

## 2.2 Theoretical view

### 2.2.1 Motivation

B-trees have been designed to hold large quantity of data, even amounts that do not fit into main memory. In such cases, it is necessary to store parts of the tree in secondary memory like hard disk drives. Compared to main memory, disk access is very slow. So the goal is to minimize the amount of disk accesses during the tree operations.

Let's take the *search* operation as an example: The proceeding to find an item $x$ in a search tree is as follows (a more detailed description is given in section 2.2.3): beginning at the root, a node is examined to see whether it contains $x$ or not. If the item has not been found in the current node, the lookup has to be carried on in a child node. Fortunately, we can benefit from the search tree property to determine the subtree where the search must continue. The corresponding node must now be loaded into main memory. In the worst case, when $x$ is not part of the tree items, the lookup ends only when a leaf node is reached. In such a case, $h + 1$ tree nodes are loaded from disk, where $h$ is the height of the tree.

The seek operation of the disk drive takes several milliseconds to complete, hence avoiding any single disk access is valuable. In binary trees, the best case tree height is $\left\lceil \log_2 \left( \frac{N+1}{2} \right) \right\rceil$, where $N$ is the number of tree items. The fundamental idea of B-trees is to optimize the yield of a disk access by ruling out as many uninteresting subtrees as possible. Thus, the path to find an item becomes shorter and less disk access is needed. B-tree nodes contain many items and references to subtrees. The tree becomes very broad,

but the height remains particularly low. The optimal node size is equal to the disk's block size. This setup leads to the minimum amount of disk accesses.

Consider the following example: a B-tree whose nodes have 199 subtree references is capable to store up to 1999999 items without having greater height than 4. The optimal height of a binary tree containing the same items would be 20.

### 2.2.2 General properties

In contrast to most other trees, B-tree nodes have multiple items and multiple children. The *order* of a B-tree defines the maximum and minimum amount of items and children. In literature, this notion is not used consistently. We use the terminology proposed by D. Knuth [5] which defines the order as the maximum number of children of a node. Non-empty B-trees of order $m$ have the following properties:

1. The root has at least 2 children.
2. Each non-leaf node except the root has at least $\lceil m/2 \rceil$ children.
3. Every node has at most $m$ children.
4. Every node with $i$ children has $i$-1 items and these items partition the keys in the children in the fashion of a search tree.
5. The leaf nodes are empty and are all on the same level.

B-trees of order 3 are also called *2-3 trees*. In general, B-trees of order $m$ are called $\lceil m/2 \rceil$-$m$ trees, because each inner node except the root has at least $\lceil m/2 \rceil$ and at most $m$ children.

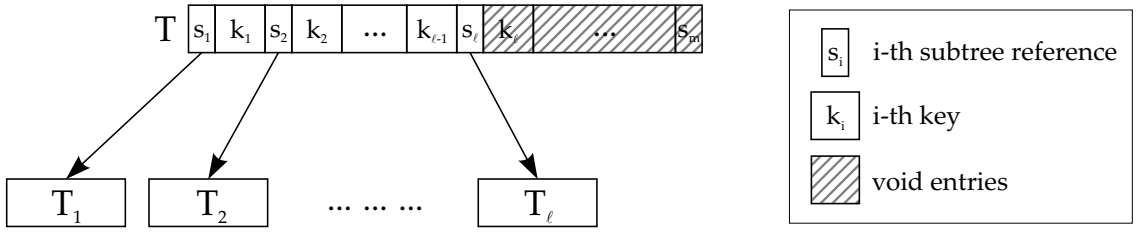



***Figure 2.1:*** *B-tree of order* m *with l children*

Figure 2.1 shows a B-tree of order $m$. Each non-leaf node except the root has items (or *keys*) $k_1$ to $k_{l-1}$ and subtree references $s_1$ to $s_l$, where $\lceil \frac{m}{2} \rceil \leq l \leq m$. All items are unique; the items stored in subtree $s_i$ are smaller than $k_i$ and similarly, the items in subtree $s_{i+1}$ are greater than $k_i$.

Let's have a look at the height $h$ of a B-tree compared to the number of items $N$: The number of leaf nodes is maximized if every node has exactly $m$ children. Hence the maximum number of children $N_{max} = m^h$. A tree with $N$ keys has $N + 1$ leaf nodes. The height of a B-tree with $N$ items is between the following bounds:

$$h \leq 1 + \log_{\lceil m/2 \rceil} \left( \tfrac{N+1}{2} \right) \text{ and } h \geq \log_m (N + 1).$$

There are some special kinds of B-trees: in $B^+$-*trees*, every item is mapped to an integer key. The inner nodes contain only the keys, all effective data is stored in the leaf nodes. We cannot use this approach because in our implementation, we use the item values as keys directly. Another kind are $B^*$-*trees:* the nodes are kept at least 2/3 full by redistributing the items in a special way. Since the complexity remains logarithmic, we have decided to implement standard B-trees.

### 2.2.3 Basic operations

#### Searching

Searching for a key $x$ in a B-tree is a generalization of the searching technique in binary trees. We start at the root node and decide whether $x$ is part of the items $k_1$ to $k_l$, $1 \leq l < m$. To accelerate that decision, binary search may be used. If $x$ is not present in the current node, the search continues in subtree $s_i$, where $i$ is smallest index of an item greater than $x$. If no such item exists, the search continues in $s_{l+1}$ which is the last non-void subtree. This procedure is repeated until $x$ is found or until the search ends without success in a leaf node.

It is obvious that at most $h + 1$ nodes have to be examined to get the result. As we have seen in section 2.2.2, the height is logarithmic compared to the number of elements and therefore the *has* query completes in $\mathbf{O}(\log_m(N))$.

#### Insertion

The first step to insert a new key $x$ into a B-tree is to search for $x$. This is necessary because duplicate items are not allowed (see also section 2.3.3). New keys are not yet part of the B-tree, so the search ends in an (empty) leaf node. Let $p$ be the parent of that leaf and $s_i$ the subtree pointer to the leaf node where the search has ended. Due to the nature of the search algorithm, $x$ will be inserted into node $p$. There are two possible situations:

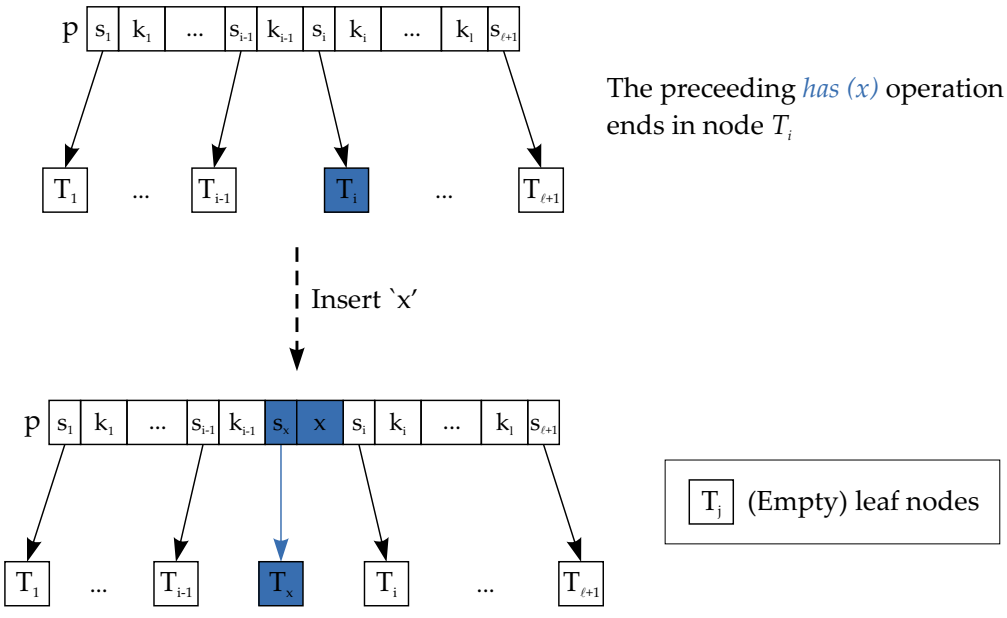



***Figure 2.2:*** *Insertion of* x *into non-full B-tree node*

1. $p$ has less than $m-1$ items (figure 2.2):
   In this case, $x$ is placed in $p$ between $k_{i-1}$ and $k_i$ and a new empty leaf is inserted just to the left of $x$.

2. $p$ has already $m-1$ items:
   In this case we put $x$ into $p$ as described in case 1. Afterwards, $p$ is split into two parts: one node will contain the items $k_1$ to $k_{\lceil m/2 \rceil - 1}$ and the other one will contain $k_{\lceil m/2 \rceil + 1}$ to $k_m$. The middle item $k_{\lceil m/2 \rceil}$ is inserted into $\varphi p$, the parent of $p$. This procedure is repeated recursively up to the root or until a non-full node is reached. If the root node needs to be split, a new root is created which will get the two tree nodes from the last split operation as child nodes and the middle item as single key. Figure 2.3 illustrates the splitting process of an inner node:

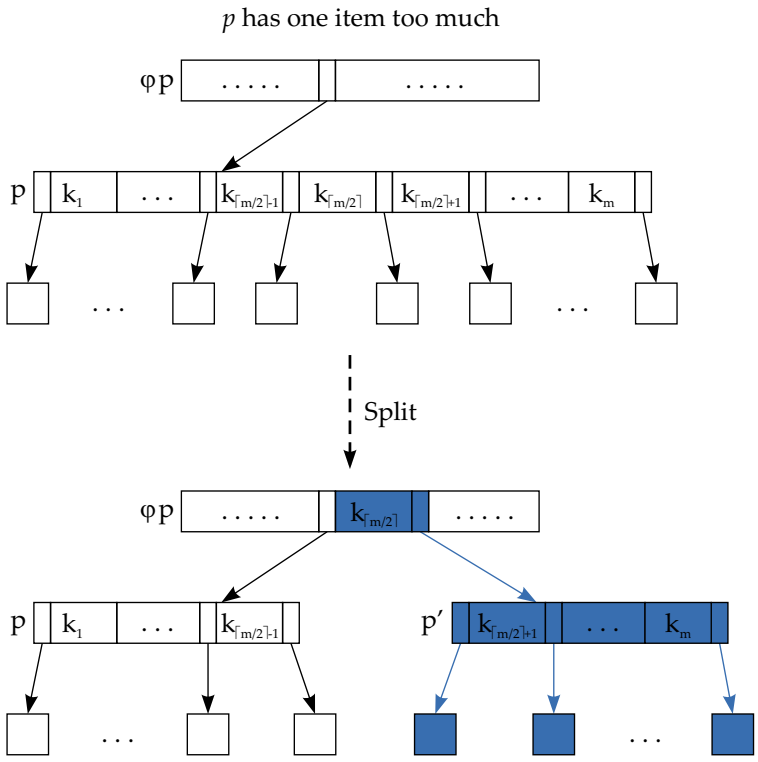



***Figure 2.3:*** *Splitting of node* p *after insertion of a new key*

The insertion operation in a B-tree with order $m$ and $N$ items (and therefore $N+1$ leafs) terminates at the latest after $\log_{\lceil m/2 \rceil}(N+1)$ steps, which gives us the complexity of $\mathbf{O}(\log(N))$.

### Removal

It is difficult to remove an item $x$ from an inner node of a B-tree, because we must hold up the B-tree property number 4: "every node with $i$ children has $i$-1 items (...)". Either we put a different item at the position of $x$, or we rearrange the tree such that the corresponding node has one child less than before. Rearranging the tree is much too

expensive. Instead, the following algorithm can be applied to remove an item: $x$ is pushed down the tree to the lowermost level. It is deleted and one of the (empty) leaf nodes is removed from the tree. If the number of remaining keys in that node is less than $\lceil m/2 \rceil - 1$, the tree must be balanced.

Pushing down $x$ from an inner node towards the leafs is easy: it is simply exchanged with its *symmetric predecessor,* which is the largest item smaller than $x$. Because we are in an inner node, there is always such an element. The symmetric predecessor is in the lowermost level of the tree, hence we have achieved our first goal. It is obvious that the tree is not sorted at the moment, but the only item which is at the wrong position is $x$. Since it will be removed in the next step anyway, the tree remains sorted at the end.

The difficult part is to remove $x$ from the lowermost level in the tree. We must keep up the condition that all nodes must have at least $\lceil m/2 \rceil - 1$ items. Let $T$ be the tree node where $x$ is located. The trivial part is when $T$ has at least $\lceil m/2 \rceil$ items. Then, $x$ and one of the empty leaf nodes are removed; no further balancing is necessary. Otherwise, $x$ is removed and four cases are to be considered:

1. $T$ is the root. If no keys remain, $T$ becomes the empty tree. Otherwise, no balancing is needed because the root is permitted to have as few as two subtrees and one single key. For the remaining cases, $T$ is not the root.

2. $T$ has $\lceil m/2 \rceil - 2$ items and it also has a sibling immediately to the left with at least $\lceil m/2 \rceil$ keys. The tree is balanced by doing an LL-rotation as shown in figure 2.4 on the next page. Notice that after the rotation, both siblings have at least $\lceil m/2 \rceil - 1$ keys. Furthermore, the heights of the siblings remain unchanged. Therefore, the resulting tree is a valid B-tree.

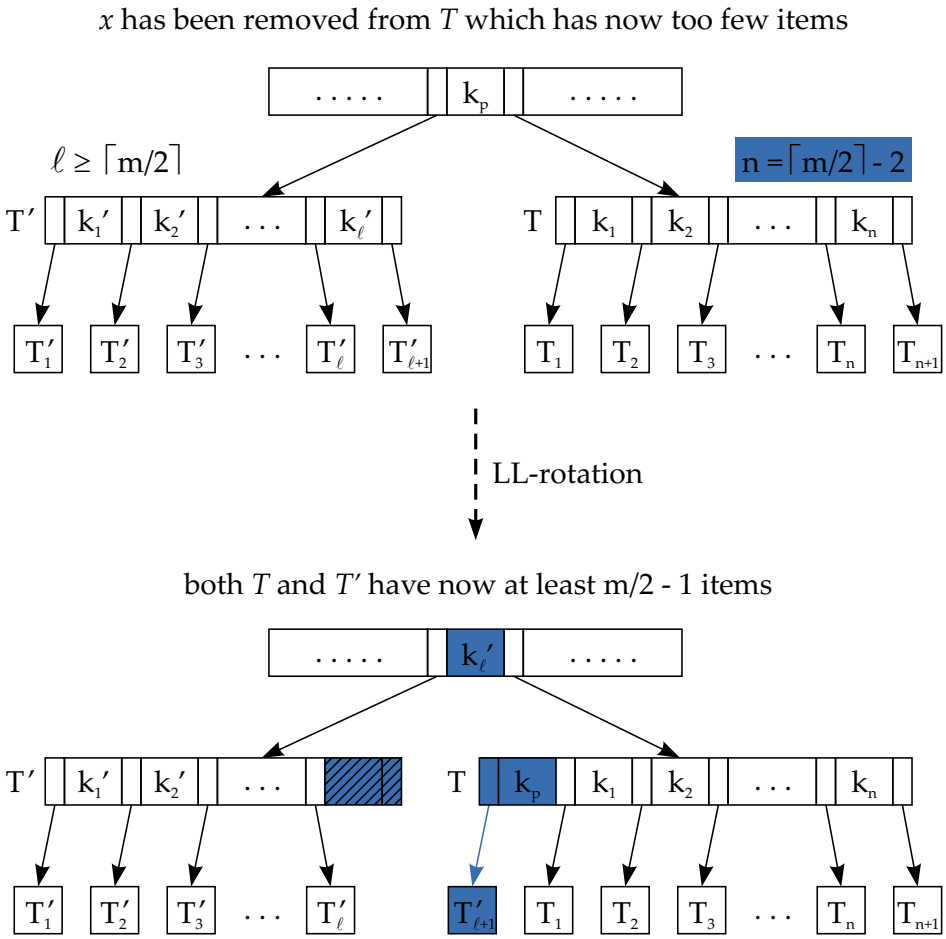



***Figure 2.4:*** *LL-rotation after pruning* x

3. Mirror of the second case: $T$ has $\lceil m/2 \rceil - 2$ items and has a sibling immediately to the right with at least $\lceil m/2 \rceil$ keys. In this case, the tree can be balanced by doing an RR-rotation. The RR-rotation is exactly the same as the LL-rotation, just performed in the opposite direction.

4. The immediate siblings have only $\lceil m/2 \rceil - 1$ keys. In this case, there are too few items to perform a rotation. The solution is to merge $T$ with one of its siblings. The new node will contain all items and children of $T$, all items and children of $T'$ and $k_j$, the appropriate item from the parent node (figure 2.5). It is not possible that the merged node exceeds its capacity. If $m$ is even, it will have $m - 2$ items, otherwise it will have $m - 1$ items which is just the limit.
   Because the item $k_j$ is pushed down into $T$, it must be removed from the parent node. This is done in the same way as any other item is removed. Hence it is possible that the parent node will also be merged with one of its siblings and so on, up to the root.

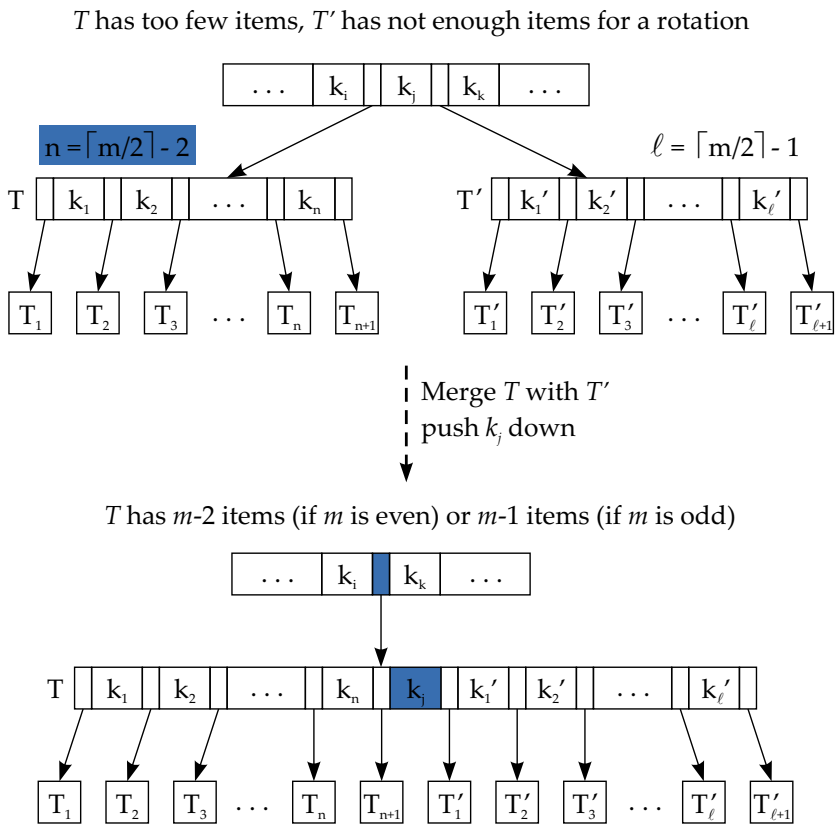



***Figure 2.5:*** *Subtree merging after removing* x

## 2.3 Implementation

### 2.3.1 Fitting B-trees into the tree cluster

In EiffelBase, there is already a cluster *tree* which contains implementations of several kinds of trees. All existing trees can have an arbitrary amount of children, but only one item per node. In contrast, the number of items in a B-tree node depends on the number of children, so we chose not to inherit from an existing class.

B-trees belong to the family of *balanced search trees*. Therefore, an intermediate class has been introduced between *TREE* and *B_TREE*. The class *BALANCED_SEARCH_TREE* may serve as base for future balanced search tree implementations like AVL-trees for example.

Figure 2.6 shows the inheritance diagram including the new classes *BALANCED_TREE* and *B_TREE*:

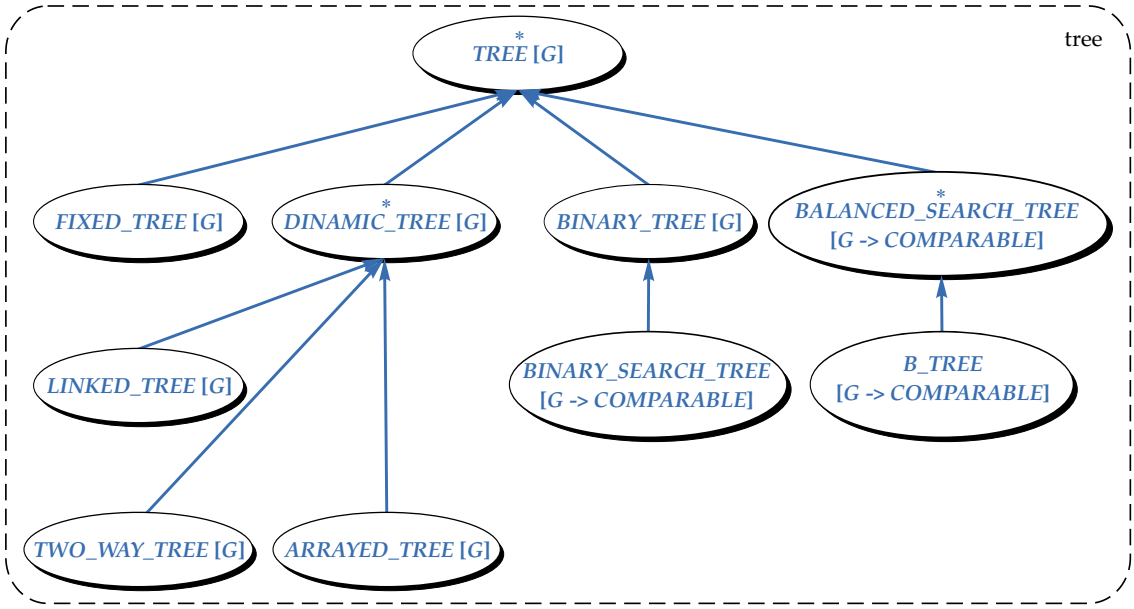



***Figure 2.6:*** *Class hierarchy of cluster* tree

### 2.3.2 *BALANCED_TREE* features

The common properties of balanced search trees are encapsulated in this class. Any operations which modify directly either the tree items or the arrangement of the subtrees are not exported to the client anymore because balanced trees are self-organizing.

The new features of *BALANCED_SEARCH_TREE* are:

- *put, extend* (*v*: *G*)
  Put *v* at appropriate position in the tree (unless *v* already exists).
- *prune* (*v*: *G*)
  Remove *v* from the tree (*v* may be in a subtree).
- *min*: *G*
  Minimum item stored in the tree
- *max*: *G*
  Maximum item stored in the tree
- *is_sorted*: *BOOLEAN*
  Are all tree items still in sorted order? Modifying item values after insertion in the tree may violate the search tree criterion.
- *has_unique_items*: *BOOLEAN*
  Are all items in the tree distinct?
- *is_valid_balanced_search_tree*: *BOOLEAN*
  Are both *is_sorted* and *has_unique_items* satisfied?
- *sort*
  Restore order of all items and remove duplicate items.

The following routines are not available anymore because balanced trees are self-organizing:

- *tree_put* (*v*: *G*)
  Replace item in current node by *v*. (renamed feature *put* from class *TREE*)
- *child_put*, *child_replace* (*v*: *G*)
  Replace item of current child node by *v*.
- *put_child* (*n*: *BALANCED_TREE*)
  Make *n* a child of the current node.
- *replace* (*v*: *G*)
  Replace item in current node by *v*.
- *replace_child* (*n*: *BALANCED_TREE*)
  Replace current child node by *n*.
- *prune_tree* (*n*: *BALANCED_TREE*)
  Remove *n* from the children. (renamed feature *prune* from class *TREE*)
- *forget_left*
  Detach all left siblings from the tree.
- *forget_right*
  Detach all right siblings from the tree.
- *wipe_out*
  Remove all children of current node.
- *sprout*
  Detach current node from tree and make it a new root.

As described in section 2.3.3, the comparison criterion in balanced search trees is unchangeably set to *object_comparison* because there must not be duplicate keys in the tree. However, it is still possible to destroy the order of a balanced tree or to introduce duplicate items by modifying the item values after insertion in the tree. For that reason, all features which rely on the search tree property have the precondition *is_valid_balanced_tree*. The command *sort* can be used to restore that condition.

### 2.3.3 Implementation of class *B_TREE*

#### Storage of items and children

The items and children of a B-tree node are stored in a list. There are several aspects which are to be considered for the right choice of the list class. Table 2.1 shows a comparison of the two candidates:

| *LINKED_LIST* | *FIXED_LIST* |
|---|---|
| Having found the correct position, insertion and removal completes in O(1) | All items to the right of the insertion position must be shifted one position to the right. Similarly, a part of the list must be copied to the left after removal |
| Only linear search is applicable to find an item in the list. | Binary search may be used because the items are increasingly sorted. |
| The list must be traversed to access the $i$-th item. | Fast access to the $i$-th item because an array is used to store the elements. |

***Table 2.1:*** *Comparison of FIXED_LIST and LINKED_LIST for choice of item list.*

It was not a priori clear which class would be more efficient in practice. We expected *LINKED_LIST* to produce the better results: the tree height is very low and hence the search paths are quite short. So the advantage of binary search over linear search might not be that significant. On the other hand, insertion and removal in linked lists does not involve shifting any items which is a big advantage.

We have implemented both alternatives and have run performance tests to compare the efficiency. In our test setup, a large amount of randomly chosen integer numbers were put into a B-tree. The three basic operations *put*, *has* and *prune* have been applied many times and the overall time consumption has been measured. Although the items were chosen randomly, the effective test cases were identical for both implementations.

| *Continuous output after each step* | | | | |
|---|---|---|---|---|
| **B-tree order** | **Items** | ***LINKED_LIST*** | ***FIXED_LIST*** | **Speedup** |
| 3 | 651 | 4,757 ms | 4,364 ms | 8.3 % |
| 3 | 689 | 5,118 ms | 4,596 ms | 10.2 % |
| 9 | 815 | 5,237 ms | 4,647 ms | 11.3 % |
| *Summarized output only at end of test run* | | | | |
| **B-tree order** | **Items** | ***LINKED_LIST*** | ***FIXED_LIST*** | **Speedup** |
| 100 | 99,513 | 13,705 ms | 8,933 ms | 34.8 % |
| 200 | 39,242 | 6,924 ms | 2,927 ms | 57.7 % |
| 200 | 99214 | 27,476 ms | 11,284 ms | 58.9 % |

***Table 2.2:*** *Performance comparison between LINKED_LIST and FIXED_LIST*

Table 2.2 shows the result of the benchmark tests. The listed values are the arithmetic mean of five test runs. To our surprise, the *FIXED_LIST* produced significantly better results than the *LINKED_LIST* implementation. The higher the tree order was, the larger the relative time difference became. Apparently, the effects of binary search and array-based memory access overbalance the performance loss when shifting around parts of the list.

### Leaf node optimization

So far, we have considered the leaf nodes of a B-tree to be empty. This allowed us to state that any B-tree node with $n$ items has $n + 1$ children. From the programmer's point of view, empty nodes are a useless waste of memory. It is not necessary to have a complete

level of tree nodes which do not have any function other than "being leafs". We have decided to omit the lowest tree level.

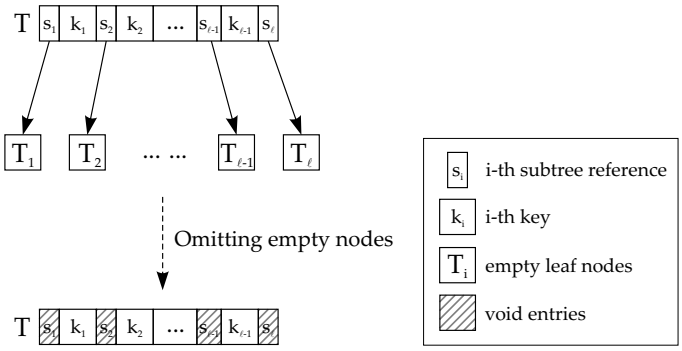



***Figure 2.7:*** *Memory optimization by avoiding empty leaf nodes*

Figure 2.7 shows the same tree once from the theoretical point of view and once how it is implemented. All empty leaf nodes $T_i$ are omitted. Instead, the node $T$ becomes itself a leaf node. The tree height is also affected; it is decreased by one compared to the theoretical view. Obviously, the statements 1., 2. and 5. from section 2.2.2 are not correct anymore in their current form. However, there is no impact at all to the overall functionality of the B-tree. It is just an optimization for memory space.

### Optimization of the *put* operation

A B-tree must not have duplicate item values, therefore an item $x$ is only inserted if the result of *has* $(x)$ is false. The node where $x$ will be put is exactly the same where the *has* query ends. It is useless to search for that specific node again when inserting $x$. We have equipped each B-tree node with an internal attribute *matching_node*. The routine *has* stores the node where the search has ended in the *matching_node* attribute of the root node. The *put* operation can now benefit from the stored value and can directly operate on the correct tree node without performing the same search immediately again.

### Linear representation

To compute the *linear_representation* of all tree items, an in-order traversal is performed and all items are stored in an arrayed list. In case the tree is sorted, this results in a sorted list.

Be aware that B-trees have initially been designed to swap parts of the data to secondary memory. The *linear_representation* puts *all* tree items into a single list in main memory, which can lead to memory shortage.

### Sorting

To invoke any command which relies on the search tree property, a B-tree must be sorted. The query *is_sorted* reports the current status. If necessary, the tree can be sorted by calling *sort*.

The first version of *is_sorted* was implemented as follows: First, the *linear_representation* of all items was generated; then it was traversed to see if all items are stored in ascend-

ing order. It is obvious that the whole tree must be traversed before the result can be computed, even if the first two items are already in wrong order.

The second and final version stops as soon as it encounters the first item which breaks up the sorted order: the tree is traversed according to the in-order strategy and the largest item which has been found until now is passed to the next node. In that node, the appropriate item is compared to the largest item until now and if it is smaller, the query stops and the result will be false. This method requires almost no additional memory (unlike the linear representation) and is even more efficient because it stops earlier in case of negative result.

Sorting a B-tree by rearranging the items seems to be quite difficult. For simplicity, the following approach was taken in this implementation: A new tree is created and all items of the current tree are inserted into the new tree using the *put* operator. The root of current tree is replaced by the root of the new tree. The implementation of *put* makes sure that the new tree is sorted and does not contain any duplicate items anymore.

#### Object identification and reference comparison

Search trees have two different comparison criteria: the search tree property requires that items in the left subtree of a node are smaller or equal than the item in that particular node and those in the right subtree have to be greater. The second comparison criterion comes from the *CONTAINER* class in EiffelBase. Two objects can be compared for equality either based on their references or based on their object value.

It is uncertain what should happen when the $\leq$ operator is used together with reference comparison. The implementation of both smaller or equal and greater or equal are expressed in terms of the $<$ and $>$ operator only. *Current* <= *other* is implemented as **not** *other* < *Current*. As we can see, the objects are only compared by size, but never tested for equality. Hence it is possible that the result of $\leq$ is true although *other* is neither smaller nor equal.

Another consequence when using reference equality is that the *has* query becomes quite cumbersome in balanced trees. Imagine a balanced search tree that contains several different objects, but all of them have the same value. Because the tree is balanced, it is probable that there will exist a right subtree containing the same item as its parent node. This would violate the search tree property and hence it was decided to forbid reference comparison in balanced trees.

We cannot guarantee that an items value is not changed after insertion into a balanced tree. It is possible to modify a tree to contain twice the same object value. As we have seen before, this can cause problems and therefore the operations *put*, *prune* and *has* have the precondition *has_unique_items*.

## 2.4 Limitations and problems with existing tree classes

### 2.4.1 Class invariant from class *TREE* produces contract violation

The main limitation of the current implementation is that class invariant checking must be turned off for the EiffelBase cluster. Objects of type *TREE* can have an arbitrary number of children. The number of children is called *arity*, the maximum number of children is *child_capacity*. It is possible to iterate over the child nodes from *child_start*

until *child_after*. But unfortunately, *child_after* is defined differently than *after* of *LIST*-like structures:

```
feature -- Status report
    child_after: BOOLEAN is
            -- Is there no valid child position to the right of cursor?
        do
            Result := child_index = child_capacity + 1
        end
```

In our opinion, it is wrong to define *child_after* in terms of *child_capacity*. It should rather be *true* if *child_index* is greater than *arity*. Otherwise it would not make sense to differentiate between *arity* and *child_capacity*. The feature *child_after* should look as follows:

```
feature -- Status report
    child_after: BOOLEAN is
            -- Is there no valid child position to the right of cursor?
        do
            Result := child_index = arity + 1
        end
```

It is useless to redefine *child_after* in class *B_TREE*, since a class invariant in *TREE* enforces the correctness of the initial definition. Currently, the consequence is a contract violation when invariant checking is turned on.

### 2.4.2 Incompleteness of binary search tree operations

In general, search trees should provide at least three basic operations (dictionary functions):

- Insertion: *put* (*v*: *G*)
- Lookup: *has* (*v*: *G*): *BOOLEAN*
- Removal: *prune* (*v*: *G*)

In class *BINARY_SEARCH_TREE*, only *put* and *has* are available. You have the ability to prune subtrees, but not individual items.

### 2.4.3 Vulnerability of binary search trees

A basic property of search trees is the automatic arrangement of items. The *put* operation must not overwrite the item in the current node, but place it at the appropriate position in the tree. Otherwise, the binary search tree would be destroyed.

The *put* and *extend* operations in class *BINARY_SEARCH_TREE* are implemented according to that concept. But nevertheless, it is possible to call *replace* on a tree node and overwrite the current item. Since the feature is called on the *tree* object, one might expect that the binary tree is rearranged such that the result remains a search tree. But this is not the case, the item is just overwritten and the search tree is destroyed.

The solution would be either to rearrange the replaced item or to restrict the export status of *replace* to *BINARY_SEARCH_TREE* only.

### 2.4.4 Wrong implementation of *sorted* in binary search trees

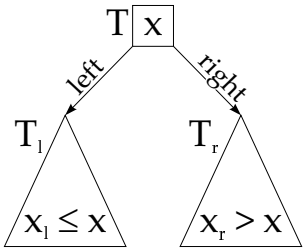



***Figure 2.8:*** *Item arrangement in a binary search tree*

According to figure 2.8, items in $T_l$ must be smaller or equal than $x$ and items in $T_r$ must be greater than $x$. This property must hold for all nodes recursively. Class *BINARY_SEARCH_TREE* provides the *sorted* query which checks if that property is satisfied.

Unfortunately, the implementation compares $x$ only with the immediate child items. Cases where an item in a right subtree of $T_l$ is greater than $x$ are not taken into account. The trees shown in figure 2.9 will all be qualified as *sorted* although the ones with modified values are not:

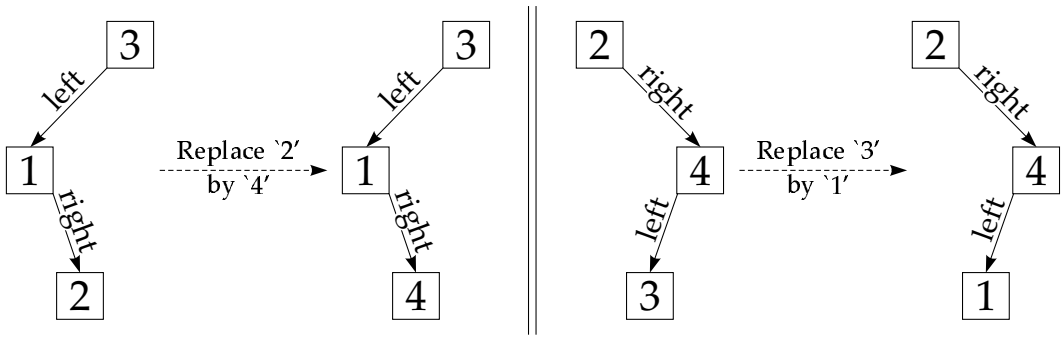



***Figure 2.9:*** *All trees are qualified as* sorted

The source code to build the modified tree in the left part of figure 2.9 looks as follows:

```
feature

    sorted_test is
            -- Test 'sorted' feature of class 'BINARY_SEARCH_TREE'.
        local
            bst: BINARY_SEARCH_TREE [INTEGER_REF]
            i, j, k: INTEGER_REF
            is_sorted: BOOLEAN
        do
            -- Create 'i', 'j' and 'k'.
            create i
            create j
            create k
```

```
        i.set_item (3)
        j.set_item (1)
        k.set_item (2)
        -- Put 'i', 'j' and 'k' into a binary search tree.
        create bst.make (i)
        bst.put (j)
        bst.put (k)
        -- Modify item value of 'k'.
        k.set_item (4)
        -- Check if 'bst' is sorted.
        is_sorted := bst.sorted -- 'is_sorted' will be True.
    end
```

A possible correct implementation of *sorted* would be to perform an in-order traversal and to check if the current item is always greater or equal than the maximum item until now.

### 2.4.5 Unusual *linear_representation* of binary search trees

The last remark is rather a style reproach than an error. Binary search trees are a sorted data structure. Thus, it would seem natural that the *linear_representation* holds the items in ascending order. But the tree is traversed following the *preorder* strategy when the *linear_representation* is produced, which leads to a completely mixed up item arrangement. We would have preferred an *in-order* traversal which takes exactly the same effort but leads to a sorted list of items.

This kind of linear representation could even be used to answer the *sorted* query: The linear representation must just be traversed sequentially and each item has to be greater or equal than its predecessor.

# Chapter 3

# Topological sort

## 3.1 Introduction

Topological sorting is necessary when we want to order a number of elements that do not have an absolute value. The only information we have is a set of binary relations between some of the elements. For each pair involved in such a relation, we can state which element must occur first in the output. The task is to find a linear ordering which conforms to all constraints.

There are many situations where topological sorting can be used. Here are some examples:

- Produce a schedule from a set of tasks. Some tasks can only start after the completion of other ones. A topological sort produces a schedule that conforms to all constraints.
- Windowed operating system: Find an order in which the windows must be drawn to achieve the correct overlapping.
- Programming language with multiple inheritance: Generate a list of all classes involved in a project, such that the ancestors appear before their heirs.

To illustrate the problem, we take the overlapping windows as example. To achieve the layout shown in figure 3.1, the system must paint the windows in the correct order. Windows which are overlapped must be drawn before those which overlap them.

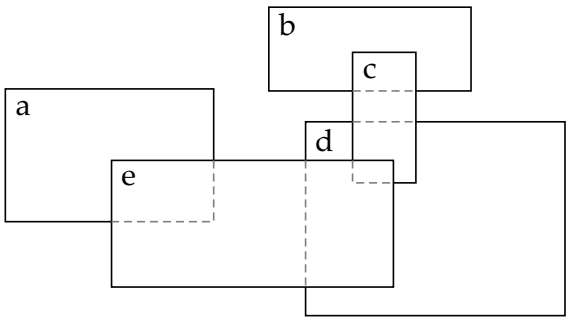



***Figure 3.1:*** *Topological sort example: Overlapping windows*

The windows do not have an absolute depth value (z-coordinate); we only have information of the kind "$x$ is behind $y$". Listing these relations, we get the following set of constraints:

- $a$ is behind $e$

- $b$ is behind $c$
- $c$ is behind $e$
- $d$ is behind $c$
- $d$ is behind $e$

The task is now to find a total order on the elements $a$-$e$ that respects these constraints. For our example, there exist several solutions; two possibilities are:

1. a, b, d, c, e
2. d, b, c, a, e

## 3.2 Mathematical formulation

We can formulate the topological sort problem in mathematical terms with sets, (acyclic) relations and (total) order relations. Below, we provide definitions for these notions.

> *Definition*: **The topological sort problem**
>
> Given an acyclic relation $r$ on a finite set, find a total order relation of which $r$ is a subset.

> *Definition*: **Relation**
>
> A *relation* over a set $A$ (short for *binary* relation) — is a set of pairs of the form $[x, y]$ where both elements of the pair, $x$ and $y$, are members of $A$.

> *Definition*: **Acyclic relation**
>
> A relation is acyclic if it has no cycle.

with:

> *Definition*: **Cycle in a relation**
>
> A *cycle* for a relation $r$ over a set $A$ is a non-empty sequence $x_1, \ldots\ x_m$ of elements of $A$ ($m \geq 1$) such that all successive pairs $[x_i, x_{i+1}]$ for $1 \leq i \leq m$ belong to $r$, and $x_m = x_1$.

To succeed, topological sort requires an acyclic relation. However, the provided algorithm is capable of partially processing cyclic constraints.

> *Definition*: **Predecessor**
>
> A *predecessor* of an element $y$ for a relation $r$ is an element $x$ such that the pair $[x, y]$ belongs to $r$.

**No-predecessor theorem**

For any acyclic relation $p$ over a non-empty finite set $A$, there exists an element $x$ of $A$ with no predecessor for $p$.

The proof of the no-predecessor theorem is given in [6]. The only part that is missing now is the connection between the input relation and the sorted output. The output list is a total order relation.

*Definition*: **Order relation (strict, possibly partial)**

A relation is an *order relation* if it satisfies the following properties for any elements $x$, $y$, $z$ of the underlying set $X$:

*Irreflexive*: the relation has no pair of the form $[x, x]$.
*Transitive*: whenever the relation contains a pair $[x, y]$ and a pair $[y, z]$, it also contains the pair $[x, z]$.

*Definition*: **Total order relation (strict)**

A *total order* is an order relation that additionally is:

*Total*: for any $a$ and $b$, one of the following holds: $[a, b]$ is in the relation; $[b, a]$ is in the relation; $a = b$.

We have seen that there must not be any cycles in the input for the existence of a topological sort. The statement does also hold the other way round:

**Topological sort theorem**

For any acyclic relation $r$ over a finite set $A$, there exists a total order relation $t$ over $A$ such that $r \subseteq t$.

The proof is by induction on the number of elements $n$ in the set $A$. For $n = 1$, the set $A$ consists of a single element $x$. The only acyclic relation $p$ in $A$ is the empty relation, since a relation containing the pair $[x, x]$ would create a cycle. This proves the base step.

For the induction step, we assume that the theorem holds for $n$ elements. We consider an acyclic relation on a set $A$ of $n+1$ elements. According to the no-predecessor theorem, there is at least one element $x$ with no predecessor.
Let $A'$ be $A \setminus \{x\}$ and $r'$ be $p \setminus \{[x, y]\}$ for any $y \in A$. Clearly, $r'$ is an acyclic relation over $A'$ (removing pairs from a relation cannot create cycles). $A'$ has $n$ elements; by the induction hypothesis, there exists a total order $t'$ over $A'$ that is compatible with $r'$. The relation $t$ over $A$ consists of the following pairs:

- all the pairs in $t'$
- all pairs of the form $[x, y]$, where $y$ is an element of $A'$

We can see that $t$ is a total order and that $p \subseteq t$. The result is a total order compatible with $p$.

## 3.3 Algorithm

### 3.3.1 Overall form

The topological sort algorithm is inspired by the proof of the topological sort theorem. The difference is that we reverse the induction step; we go from $n$ to $n-1$ elements.

The general idea of the algorithm is simple: Before an element $x$ from $A$ can be placed in the sorted output, all of its direct and indirect predecessors must be processed. The reverse of this statement is that any element without predecessors is ready to be added to the sorted output. Let us rename our element set $A$ to *elements* and the acyclic relation to *constraints*. The total order on the elements is called *sorted_elements*. The topological sort algorithm looks as follows:

**while** *elements* is not empty **do**
    Let $x$ be an element without predecessor
    Produce $x$ as the next element of *sorted_elements*
    Remove $x$ from *elements*
    Remove all pairs starting with $x$ from *constraints*
**end**

### 3.3.2 Handling cyclic constraints

One way to implement topological sort is to accept only cycle-free constraints. For a small set of constraints, this may be easy to determine, but for large sets it is not a trivial task. The solution in our implementation is more flexible. Depending on the input, we produce the best possible result: If the constraints are cycle-free, the result is a total order on all elements. Otherwise, the output contains all members of *elements* that are not involved in a cycle.

The procedure to find cycles in the constraints and the sorting process have a lot in common. The topological sort algorithm described in section 3.3.1 allows us to answer the question for cycles quite easily. According to the no-predecessor theorem, there exists at least one element without predecessor at each stage, as long as the input does not contain any cycles. Conversely, there is no such item if there is a cycle. Hence we have found a cycle if there is no element $x$, although *elements* is not empty.

Let *candidates* be the items without predecessor, ready to be written into *sorted_elements*. The refined version of our algorithm is now:

Put all members of *elements* without predecessor into *candidates*
**while** *candidates* is not empty **do**
    Pick an element $x$ from *candidates*
    Produce $x$ as the next element of *sorted_elements*
    Remove all pairs starting with $x$ from *constraints*
    Put all non-processed members of *elements* without predecessor into *candidates*
**end**
**if** *elements* is not empty **then**
    Report cycle
**end**

### 3.3.3 Overlapping windows example

To illustrate the workflow of the topological sort algorithm, we reconsider the overlapping windows example from section 3.1. On the right side of figure 3.2, we can see the dependency graph that corresponds to the given windows $a - e$:

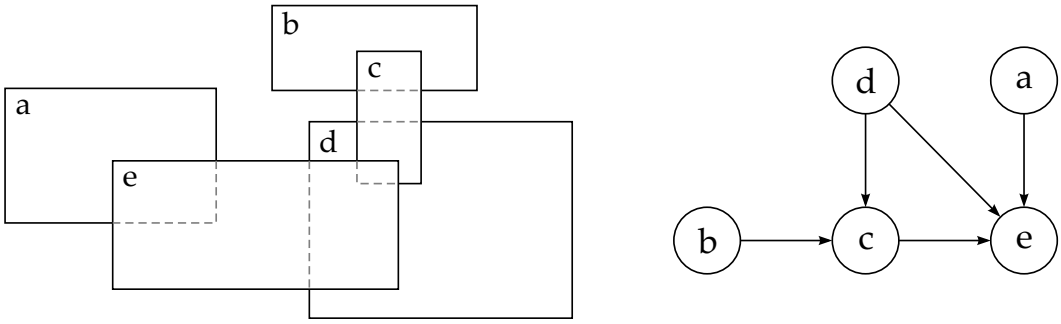



***Figure 3.2:*** *Overlapping windows and corresponding dependency graph*

The dependency graph contains all the information needed for the algorithm: The nodes correspond to the *elements* set, whereas the edges are the *constraints*. An edge from node $x$ to node $y$ signifies that the window $x$ must be drawn before $y$. For the algorithm, we are interested in the elements with no predecessor. It is easy to see that nodes without any incoming edges meet that condition. Hence we can put $a$, $b$ and $d$ into *candidates*. The first step is done, now we go for the main loop.

Let us assume we pick $d$ as first element from *candidates*. The item $d$ is removed from *elements*, which corresponds to removing the node $d$ from the dependency graph. The equivalence for removing all pairs starting with $d$ from the constraints is to remove all edges starting at $d$ from the graph. In our case, there are two such edges, one to node $c$ and one to node $e$. The resulting dependency graph looks as follows:

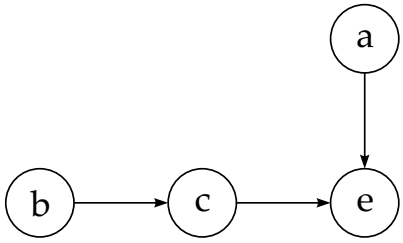



***Figure 3.3:*** *Dependency graph after removal of node* d *and its outgoing edges*

Now we have the choice to pick either $a$ or $b$ from *candidates*. Let us assume we pick $b$ as next element, after that $a$, $c$ and finally $e$. Our final linear order will then be $d$, $b$, $a$, $c$, $e$.

In this example, we can clearly see the reason why the topological sort problem can have multiple solutions: If *candidates* contains multiple elements at some point, we have the free choice which element to take. The absence of further constraints for these elements leads to multiple output possibilities. The different strategies for picking an element are described in section 3.4.6.

## 3.4 Implementation

### 3.4.1 Motivation for an object-oriented approach

One possibility for the implementation would be an algorithmic approach like this:

```
topologically_sorted (elements: ...; constraints: ...): LIST [...] is
        -- Enumeration of the members of 'elements',
        -- in an order compatible with 'constraints'
```

However, it is not clear in which class this routine should be integrated. Putting it into a container class would be a possibility or using a utility class. But this solution does not at all conform to the spirit of Eiffel. The implementation we chose for EiffelBase follows the approach given in the textbook *Touch of Class* [6]. The proposed solution is a much more flexible and elegant object-oriented approach. The merits are as follows:

- abstraction into class instead of single function
- command-query separation
- genericity for the element type
- more flexibility for the result: information about cyclic constraints and partial solution in such cases

### 3.4.2 Class design: *TOPOLOGICAL_SORTER*

The implementation consists of a single class *TOPOLOGICAL_SORTER* [*G*]. Any instance of it represents an instance of the topological sort problem. The class has the following basic features:

- *record_element* (*e*: *G*)
  Include *e* in the set of elements.
- *record_constraint* (*e*, *f*: *G*)
  Include [*e*, *f*] in the constraints.
- *process*
  Perform topological sort and make the result available in *sorted_elements*.
- *sorted_elements*: *LIST* [*G*]
  Sorted list computed by *process*

We may not assume that the constraints are always consistent. Especially with many constraints, cyclic dependencies may occur and the topological sort can not succeed. Instead of refusing cyclic input, our algorithm should sort as many elements as possible. The items involved in a cycle are reported at the end. Two more features are introduced for that matter:

- *cycle_found*: *BOOLEAN*
  Did *process* find any cyclic dependencies in the constraints?
- *cycle_list*: *LIST* [*G*]
  List of elements involved in cyclic constraints (if *cycle_found* is *true*)

### 3.4.3 Storage of elements and constraints

As discussed in [6], the choice of the data structure to store the elements and constraints has a significant impact on the efficiency of the algorithm. Declaring the attributes as follows is not the best decision:

```
elements: LINKED_LIST [G]
        -- All elements to be sorted

constraints: LINKED_LIST [TUPLE [G, G]]
        -- Constraints between the elements.
```

The performance can be significantly improved if we do not store the constraints as they are given to us. Instead of storing the constraints as pairs of elements, we stay closer to the algorithm described in section 3.3.1: For every element, we keep track of its (direct) successors and the number of (direct) predecessors. We have two arrays, one for the predecessors and one for the successors. What we need is an association between the elements and the array entries. This is achieved by enumerating the elements according to the insertion order. Duplicate element insertions are detected, so the element index is unique for each item. For a fast mapping from element to index, we use a hash table. The following features are used to store the elements and constraints internally:

```
feature {NONE} -- Implementation

element_of_index: ARRAY [G]
        -- Elements in insertion order

index_of_element: HASH_TABLE [INTEGER, G]
        -- Index of every element

successors: ARRAY [LINKED_LIST [INTEGER]]
        -- Indexed by element numbers; for each element 'x',
        -- gives the list of its successors (the elements 'y'
        -- such that there is a constraint '[x, y]')

predecessor_count: ARRAY [INTEGER]
        -- Indexed by element numbers; for each, says how many
        -- predecessors the element has

candidates: DISPENSER [INTEGER]
        -- Elements with no predecessor, ready to be released
```

The *candidates* are a collection of elements ready to be written into the result list *sorted*. There are several strategies to select a candidate. They are described in section 3.4.6.

### 3.4.4 Contracts

The class *TOPOLOGICAL_SORTER* is designed according to the command-query separation principle. After all the elements and constraints are recorded, *process* is called which makes the result available in *sorted_elements*. It does not make sense to ask for

*sorted* before *process* has been called. Neither is it reasonable to call *process* multiple times without changing the elements or constraints. The flag *done* has been introduced to express that fact. The contract view of the class looks as follows:

```
class interface
    TOPOLOGICAL_SORTER [G]

create
    make

feature -- Initialization

    record_constraint (e, f: G)
            -- Add the constraint '[e, f]'.
        require
            not_sorted: not done
            not_void: e /= Void and f /= Void

    record_element (e: G)
            -- Add 'e' to the set of elements, unless already present.
        require
            not_sorted: not done

feature -- Access

    count: INTEGER
            -- Number of elements

    cycle_found: BOOLEAN
            -- Did the original constraints imply a cycle?
        require
            sorted: done

    cycle_list: LIST [G]
            -- Elements involved in cycles
        require
            sorted: done

  sorted_elements: LIST [G]
            -- List, in an order respecting the constraints, of all
            -- the elements that can be ordered in that way
        require
            sorted: done

feature -- Status report
    done: BOOLEAN
            -- Has topological sort been performed?
```

```
feature -- Status setting

    reset
            -- Allow further updates of the elements and constraints.
        ensure
            fresh: not done

feature -- Element change

    process
            -- Perform a topological sort over all applicable elements.
            -- Results are accessible through 'sorted_elements', 'cycle_found'
            -- and 'cycle_list'.
        require
            not_sorted: not done
        ensure
            sorted: done

invariant
    elements_not_void: element_of_index /= Void
    hash_table_not_void: index_of_element /= Void
    predecessor_count_not_void: predecessor_count /= Void
    successors_not_void: successors /= Void
    candidates_not_void: candidates /= Void

    element_count: element_of_index.count = count
    predecessor_list_count: predecessor_count.count = count
    successor_list_count: successors.count = count

    cyclists_iff_cycle: done implies (cycle_found = (cycle_list /= Void))
    all_items_sorted: (done and then not cycle_found) implies (count = sorted.count)
    no_item_forgotten: (done and then cycle_found) implies
                    (count = sorted_elements.count + cycle_list.count)

end
```

We can see that *sorted_elements*, *cycle_found* and *cycle_list* have the precondition *done*. The current version of Eiffel does not allow attributes to have a precondition, hence we need three hidden features for the effective storage of the values:

```
feature {NONE} -- Implementation

    has_cycle: like cycle_found
            -- Internal attribute with same value as 'cycle_found'.
    cycle_list_impl: like cycle_list
            -- Internal attribute with same value as 'cycle_list'.
    output: LINKED_LIST [G]
            -- Internal attribute with same value as 'sorted_elements'.
```

The next version of Eiffel, specified in *Eiffel: The Language 3* [7], will support assertions for attributes and will make these hidden attributes obsolete.

### 3.4.5 Performance analysis

The topological sort algorithm takes a set of elements and a set of constraints as input. We assume that there are $n$ elements and $m$ constraints. To inspect all elements and constraints, we need at least $\mathbf{O}(m+n)$ steps. The exciting result is that the implementation can keep up the complexity of $\mathbf{O}(m+n)$, both in space and time.

#### Input

##### *record_element*

There are five steps to do for registering an element $x$:

- Check if $x$ is already registered. Implemented with a hash table: $\mathbf{O}(1)$.
- If it was not already present: add $x$ to *elements*: $\mathbf{O}(1)$.
- Add $x$ to a hash table to associate it with an index: $\mathbf{O}(1)$.
- Set predecessor count for $x$ to 0: $\mathbf{O}(1)$.
- Create an empty successor list for $x$: $\mathbf{O}(1)$.

Applied to all $n$ elements, we get a complexity of $\mathbf{O}(n)$.

##### *record_constraint*

Adding a new constraint $[e, f]$ consists of three operations:

- Call *record_element* for both $e$ and $f$ to make sure both are part of *elements*: $\mathbf{O}(1)$.
- Increment the predecessor count of $f$: $\mathbf{O}(1)$.
- Add $f$ to the successors of $e$: $\mathbf{O}(1)$.

Since there are $m$ constraints, the complexity for *record_constraints* is $\mathbf{O}(m)$. Thus we can say that the input completes in $\mathbf{O}(m+n)$.

#### Processing

Let us reconsider the topological sort algorithm:

```
Put all members of elements without predecessor into candidates
while candidates is not empty do
    Pick an element x from candidates
    Produce x as the next element of sorted_elements
    Remove all pairs starting with x from constraints
    Put all non-processed members of elements without predecessor into candidates
end
```

**if** *elements* is not empty **then**
    Report cycle
**end**

The first line requires the *predecessors* array to be fully traversed, which takes $\mathbf{O}(n)$ time. Repeating the same operation in each loop iteration would lead to a complexity of at least $\mathbf{O}(n^2)$, but the further candidates can be found more efficiently.

The first two statements in the loop are trivial and complete in constant time. The line "Remove all pairs starting with *x* from *constraints*" is more interesting: The successors of $x$ are not needed anymore. We could set the corresponding entry in *successors* to *void*, but it turns out that the algorithm does not consider these entries anymore, so we can leave the array it as it is. What we must do when removing all pairs $[x, y]$ is to decrease the predecessor count of $y$. At first sight, this seems to lead to $\mathbf{O}(m * n)$, but that is not true. For each constraint, this operation is done at most once in the whole processing. So the complexity for this operation remains $\mathbf{O}(m)$.

The last task remaining for the main loop is to find elements with no predecessor, based on the new situation. Fortunately, we can combine this step with the last one. At the same time when decreasing the entry in *predecessor_count*, we can check if the value has become 0. If so, we put the corresponding element into *candidates*. The complexity is not affected by this additional step.

When the main loop terminates, we must check for cyclic constraints. No more than $\mathbf{O}(n)$ steps are required for that. All elements whose number of predecessors is still not zero could not be sorted and are added to *cycle_list*. Additionally, the flag *cycle_found* is set. Summarizing all steps of the processing part, we can keep up the overall complexity of $\mathbf{O}(m + n)$.

### 3.4.6 Parameterizing topological sort

#### Output strategies for the *candidates*

The result of a topological sort is a total order on the elements. In general, there are multiple solutions to the topological sort problem. The output order depends on the set of candidates, which are the elements with no predecessor. If there is always just one such candidate, the output order is unambiguous. As soon as there are multiple candidates, we have free choice of the output order. There are several strategies to pick the next element for output. The most frequently used of them are:

- Smallest element first
- Largest element first
- FIFO: output order is the same order as the input order
- LIFO: output order is the reverse of the input order

The *candidates* are declared as a *DISPENSER*, which is a deferred type. It is the actual type of *candidates* which defines the element that is returned by the query *item*. For instance, when the actual type of *candidates* is *ARRAYED_STACK*, the LIFO strategy is applied. In our implementation, only the FIFO and LIFO strategy could be implemented.

There are two reasons why *smallest element first* and *largest element first* could not be realized:

In section 3.4.3, we saw how the elements are mapped to an index. It is convenient for the implementation if *candidates* also holds indices rather than the elements themselves. When the *largest element first* strategy is chosen, the query *item* yields the largest index stored in *candidates*. However, the largest *index* does not necessarily correspond to the largest *element*, so this is the first obstacle.

The second reason is more severe: Even if *candidates* held the elements directly, we would not be able to get the largest (or smallest) element first. Let us assume that *candidates* is declared of type *DISPENSER* [*G*]. To use the *largest element first* strategy, the actual type of *candidates* must be a *PRIORITY_QUEUE*. The problem is that the generic parameter of *PRIORITY_QUEUE* is constrained by *COMPARABLE*. This means that the generic parameter *G* of our class *TOPOLOGICAL_SORTER* must conform to both *HASHABLE* and *COMPARABLE*. Unfortunately, EiffelStudio 5.4 does not support multiple classes for constrained genericity. The next version of Eiffel, specified in *Eiffel: The Language 3* [7] will support multiple generic constraints. Then it will be possible to provide further output strategies.

### Realization in *TOPOLOGICAL_SORTER*

The default output strategy for unconstrained elements is *FIFO*. It can be changed by calling *use_lifo_output* and *use_fifo_output* respectively. The current output mode is indicated by the flag *fifo_output*. As stated in section 3.4.6, the appropriate output strategy is achieved by choosing an actual type for *candidates* at creation time. A class invariant assures that *candidates* is never *void*.

```
feature -- Status report

    fifo_output: BOOLEAN
            -- Is FIFO strategy used for output of
            -- unconstrained elements?

feature -- Status setting

    use_fifo_output: BOOLEAN
            -- Use FIFO strategy for output of unconstrained elements.
        do
            create {LINKED_QUEUE [INTEGER]} candidates.make
            fifo_output := True
        ensure
            fifo_output: fifo_output
        end
```

```
        use_lifo_output: BOOLEAN
                -- Use LIFO strategy for output of unconstrained elements.
            do
                create {ARRAYED_STACK [INTEGER]} candidates.make (1)
                fifo_output := False
            ensure
                lifo_output: not fifo_output
            end

    feature {NONE} -- Implementation

        candidates: DISPENSER [INTEGER]
                -- Elements with no predecessor, ready to be released

    invariant

        candidates_not_void: candidates /= Void
```

# Chapter 4

# Union-find

## 4.1 Introduction

A union-find data structure is needed in the following situation: we have $n$ disjoint elements and a number of sets between 1 and $n$. Every element is part of exactly one of those sets. This implies that all sets are disjoint. For these sets, we want to have the following operations:

- Fast lookup, to which set a given element belongs
- Efficient merging of two sets

Maybe the most famous application of the union-find data structure are graph algorithms. J.B. Kruskal has proposed an algorithm to find the minimum spanning tree for undirected graphs [2]. The graph library described in chapter 1 can benefit in two ways from a union-find data structure: besides computing the minimum spanning tree, we have also an elegant way to count the number of graph components.

The implementation we made for EiffelBase is generic, powerful and efficient. The operations on the union-find structure obey by nature the command-query separation principle. This allows us to make an elegant mapping into an Eiffel class.

## 4.2 Representation of the sets

We need a way to represent the sets internally to optimize the efficiency of the basic operations *union* and *find*. The basic idea is to represent each set as a tree-like structure. The tree nodes correspond to the set elements, but there exists no specific item order. The particularity is that the tree is "inverted": every node knows its parent, but not its children.

It is inconvenient to pass whole sets as arguments to the union-find routines. We need a way to identify the different sets with some kind of label. Since the sets are finite, there exists exactly one root element for each tree. In addition, the sets are disjoint, no element is part of multiple sets. Thus, we can use the root element as an identifier for the whole set. The root element has no special property; any set element can be the root node. Figure 4.1 shows an example with four sets and 18 elements in total.

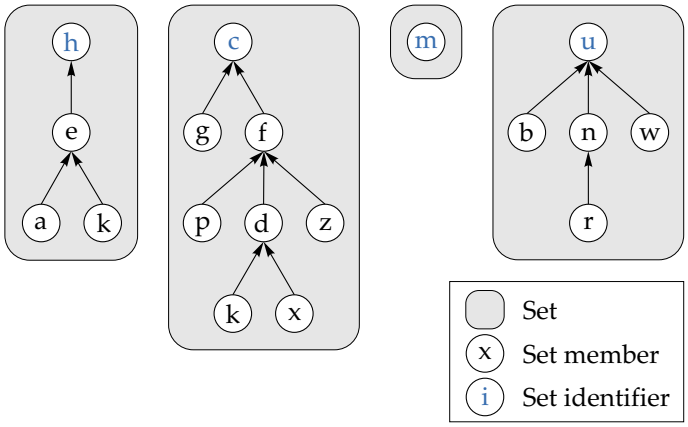



***Figure 4.1:*** *Internal representation of the sets using a tree structure*

## 4.3 Algorithms

### 4.3.1 *find*

To find out to which set a specific item belongs, the tree structure is traversed up to the root. For instance, if we want to find the set containing $x$ in figure 4.1, we traverse the corresponding tree up to the root, which is node $c$. By definition, the root node is set identifier and therefore the result.

The worst-case scenario we can get is one single tree, degenerated to a linear list. In that case, we must perform $\mathbf{O}(n)$ steps to get the set identifier. That performance is not a good enough as final result. We could as well have used a linked list instead of a tree. There is a possibility to optimize the *find* operation significantly: on our way up to the root, we encounter multiple elements. Of course, all of them belong to the same set as $x$. At the end of the *find* operation, we can attach all those elements directly to the root node by modifying their *parent* pointer. In any future call to *find* for one of those items, the root node is reached in only one step. Figure 4.2 shows the reorganization of the tree structure after calling *find* $(x)$:

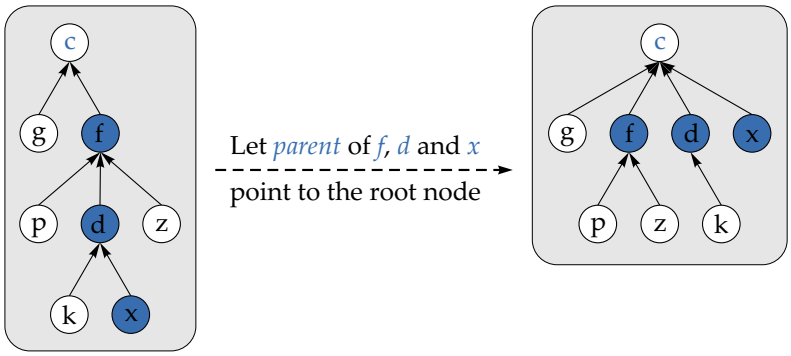



***Figure 4.2:*** *Element rearrangement to optimize future* *find* *queries*

It can be shown that the trees produced by this strategy have an average height $h \leq \log_2(n)$, where $n$ is the number of tree elements. Every invocation of *find* opti-

mizes the tree structure. Because of the average tree height, we can achieve an overall complexity of $\mathbf{O}(\log(n))$. Because of our optimization step, the more often *find* is called, the more efficient the query gets (up to $\mathbf{O}(1)$ in the best case).

### 4.3.2 *union*

When building the union of two sets, we want to avoid moving all members individually from one set into the other one. Additionally, we want to keep up the high efficiency of *find* we have achieved by optimizing the tree structure of both sets.

In our tree structure, all set members are directly or indirectly attached to the root node. To merge two sets, it is sufficient to make the root node of one tree parent of the other root node. The nice effect is that all other set elements will follow automatically. All elements of both sets are now unified in the same tree and are identified by the common root node. Figure 4.3 shows the tree structure after calling *union (h, u)*:

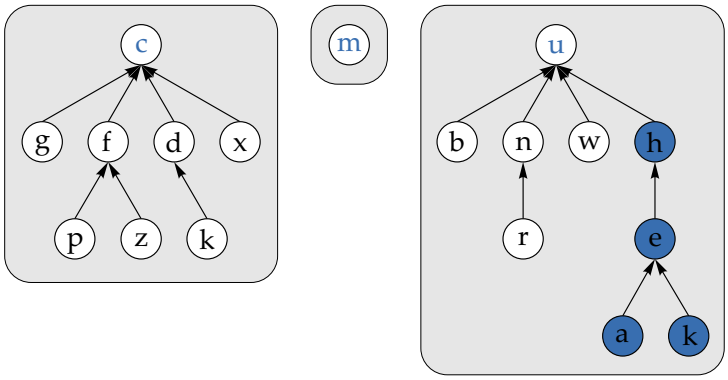



***Figure 4.3:*** *Tree structure after calling union (h, u)*

Only one parent pointer must be updated to unify the two sets. It is not even necessary to search for the root nodes because they are the set identifiers and are therefore passed as arguments to the *union* command. The complexity of the *union* operation is only $\mathbf{O}(1)$.

In average, both trees had at most logarithmic height before the union. The height of the resulting tree will be at most one more than the maximum of both heights. It is easy to see that the tree height remains as low as possible when the taller tree remains unchanged and the other tree is attached to its root node. Because we have no *child* pointers, it is not so easy to compute the height of our trees. It is easier to keep track of the number of tree elements. We have seen that there is a direct correlation between the number of elements and the tree height. Hence we use the number of tree elements to determine which tree is attached to the other.

## 4.4 Implementation in Eiffel

### 4.4.1 Class design

The class *UNION_FIND_STRUCTURE* holds the tree representation of the sets and provides all necessary operations. The set elements can be of an arbitrary type and lead

to the generic parameter *G*. For efficiency reasons, we make use of a hash table. *G* is therefore constrained by *HASHABLE*.

In section 4.3, we have seen that the root element is used as set identifier. To avoid confusion between the set identifiers and the elements, we have decided to use *INTEGER* numbers to identify the sets. This goes well in-line with the implementation, as internally the trees consist of *INTEGER* elements (see also section 4.4.2).

The main features provided in *UNION_FIND_STRUCTURE* are:

- *put* (*e*: *G*)
  Put *e* into a unary set and add it to the union-find structure
- *find* (*e*: *G*): *INTEGER*
  Find the set to which *e* belongs
- *union* (*s*, *t*: *INTEGER*)
  Merge the sets *s* and *t*
- *count*: *INTEGER*
  Number of registered elements
- *set_count*: *INTEGER*
  Number of sets

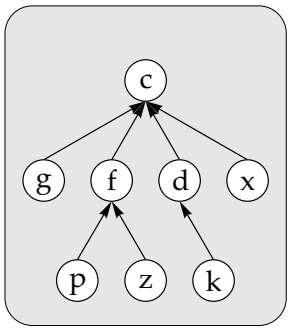
### 4.4.2 Internal representation using arrays

Although the pictures from section 4.3 may suggest to use a tree implementation, we choose another internal representation. All elements have only a pointer up to their parent. We do not need to know the children of a node, as it is the case in the Eiffel tree structures. A very efficient implementation can be made when operating with arrays.

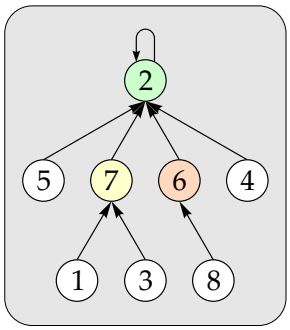
Internally, it is not convenient to work with objects of type *G*. Instead, we map every element to a unique integer value. When a new element is added, it is inserted into the hash table *index_of_element* and gets mapped to the current value of *count*. The same item can only be added once to the whole structure, repeated insertions are ignored.

Now that we have an integer representation of the set elements, it is easy to get an *INTEGER* identifier denoting the whole set. Instead of using the root *element* as set identifier, we take the root *index*. Since all elements have different numbers, there cannot be a name clash.

In figure 4.4 on the facing page, we can see how the tree structure is represented internally. All elements are stored in an array *elements*. The hash table *index_of_element* contains the inverse mapping from an element to its array index. The third array *parent* is used to represent the tree structure. The indices are the same than those of *elements*; the array holds the parent index for each tree node. The inner nodes of the example tree in figure 4.4 are highlighted to give more clearness, how the *parent* array is used. We can eliminate some special cases if every element has an entry in *parent*. That is the reason why the root node has a the self-reference.




Tree structure with elements




Tree structure with integers

*elements*

| p | c | z | x | g | d | f | k |
|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |

*index_of_element*

| | 5 | | | 1 | | 3 | | 4 | 6 | | 2 | 8 | 7 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | g | | | p | | z | | x | d | | c | k | f |

*parent*

| 7 | 2 | 7 | 2 | 2 | 2 | 2 | 6 |
|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |

***Figure 4.4:*** *Internal representation of the tree structure with arrays*

### 4.4.3 *find*

*find* (*x*) returns the identifier of the set containing *x*. The tree structure is traversed up to the root by following the entries of *parent*. If the root node points to itself in *parent*, we do not have to care whether an element has a parent or not. The following code shows how the set identifier is determined:

```
find (e: G): INTEGER is
        -- Identifier of set containing 'e'
    require
        has_element: has (e)
    local
        index: INTEGER
    do
        -- Traverse the "tree" upwards to get the root index.
        from
            index := index_of_element.item (e)
        until
            -- The "root" set number is found when the value is equal to its index.
            parent.item (index) = index
        loop
            -- Get the parent set number until the "root" set number is found.
            index := parent.item (index)
```

```
            end
            Result := index
        end
```

Note that the above code is used only as an illustration how the tree is traversed. The real implementation of the *find* routine has more contracts and contains the optimization mentioned in section 4.3.1.

### 4.4.4 *union*

The *union* operation takes two set identifiers and merges the corresponding sets. The merging step is particularly easy to implement. We only need to attach one of the root nodes to the other one.

There are still some considerations to do for the implementation. First, we must make sure that both arguments are valid set identifiers. The query *valid_identifier* is required to be *true* for both arguments. An identifier is only valid if it denotes a root element. Second, we must decide which set is kept and which is made part of the other. In section 4.3.2, we have seen that the answer is attach the root of the smaller set to the root of the larger set. The feature *items_in_set* takes a set identifier as argument and returns the appropriate set size. It is not computed, but stored in the array *elements_in_set*. Then, the *union* operation consists of two steps:

1. Replacing the larger set size in *elements_in_set* by the sum of both set sizes.
2. Changing the appropriate entry in *parent*, such that the root of the smaller set points to the other root node.

### 4.4.5 Further routines

Since we are writing a library class, we provide some additional features to make the implementation more powerful.

#### *valid_indices*

The straightforward approach to get all set identifiers is to iterate over all possible set identifiers from 1 to *count* and to check whether the corresponding element is a root element. We keep an internal list for the valid set identifiers, which is even more efficient.

#### *i_th_set*

Sometimes, we want to have the inverse operation of *find*. Having a valid set identifier, we want to know which elements belong to that specific set. Obviously, the internal representation using the *parent* array is not suited to answer that question. Of course, we could replace the internal representation by a full-featured tree data structure, but probably a lot of our great efficiency would get lost. Let us keep in mind that this is just an additional feature provided for convenience, not a core component of the union-find data structure.

The basic idea is to have a linked list of all elements belonging to the same set. The element order is absolutely irrelevant. When two sets are merged with the *union* command,

one list is simply appended to the other. We must make sure that these additional steps have the least possible impact on the performance of *union* and *find*. Therefore, we do not use the linked lists provided in EiffelBase, but use an array representation similar to the *parent* array. Our "lists" can be merged in $\mathbf{O}(1)$; there is no noticeable impact on the speed of the other operations.

#### *sets*

From the two features *identifiers*, which returns all set identifiers, and *i_th_set*, we can derive another feature: *sets* returns a collection of all sets that are currently stored in our union-find structure. This operation is used rather rarely and has mainly been provided for completeness.

## 4.5 Difficulties in the design of *UNION_FIND_STRUCTURE*

The most difficult part of the class design was the choice of the feature names. The reason for the problems is the equivalence of the noun *set* and the verb *to set*. For instance, the query *set_count* which returns the number of sets could be misunderstood as command to set the value of *count*. We could not find a better feature name, so the final version of the class contains that name nevertheless.

# Chapter 5

# Assessment and future work

## 5.1 Summary

### 5.1.1 Thesis overview

The EiffelBase library is extended in several areas not yet covered. There is now a powerful graph library, an implementation of balanced trees with B-trees, topological sort and a union-find data structure. The reusability of the components appears even at the library level: The union-find structure is used in the graph library to compute the number of components and the minimum spanning tree. After a conversion to the correct data types, it is even possible to get a topological sort of directed acyclic graphs.

### 5.1.2 Graph library

Taking the graph library proposed by Bernd Schoeller as a starting point, a graph library has been developed. Contrary to the initial solution, not every graph concept is encapsulated in a new class. The final class hierarchy consists of only four deferred graph classes. In addition, there is an edge type accessible to the user.

There are two different implementations. The first one is based on an adjacency matrix and it is very fast, but there is no support for multigraphs. The second one uses a linked structure for the incident edges of each node. That implementation is suited for all needs, but it not exactly as fast as the adjacency matrix variant.

A couple of graph algorithms are implemented. You can compute the number of graph components, determine whether a graph contains cycles and check if it is Eulerian. Additionally, you can compute the shortest path and the minimum spanning tree.

### 5.1.3 B-trees

The *tree* cluster is extended by a modern form of trees: B-trees. They are part of the family of balanced search trees. To avoid the worst case of degenerating to a linear list, the nodes are self-organizing and are balanced after insertion and removal operations if necessary.

B-tree nodes contain multiple elements and multiple children references. The tree becomes very broad, but remains particularly flat. With that design, the smallest amount of tree nodes must be inspected to search for an element. This is a big advantage when parts of the tree are swapped to the hard disk which has very high access times.

The deferred class *BALANCED_SEARCH_TREE* has been integrated in the class hierarchy. It provides a generic interface for balanced search trees and allows future implementations of other balanced trees, like AVL-trees or splay trees.

The implementation phase has uncovered some deficiencies in the existing *tree* cluster. A class invariant in class *TREE* is wrong (see section 2.4.1) and the implementation of *sorted* in binary search trees contains errors (section 2.4.4).

### 5.1.4 Topological sort

With topological sort, a total order can be found for elements with only a partial order. The implementation is very flexible: if no such order exists, as many items as possible are sorted, the other elements are reported to by part of a cycle. The class is well designed according to the command-query separation principle and the implementation is very efficient.

## 5.2 Future work

### 5.2.1 Improving the graph library

#### Data structure

A lot of time has been invested in the design phase of the graph library. The class hierarchy is compact, but very powerful. The two implementations are suited for many applications, but there are still some facilities to be added.

The class *ADJACENCY_MATRIX_GRAPH* currently supports only simple graphs. In a future version, the adjacency matrix could consist of edge lists instead of single edges. Alternatively, a whole new implementation could be made with respect to multigraphs.

It is even arguable whether the use of a whole adjacency matrix is a good design at all. The matrix grows to the square of the node amount; in many cases it contains only few edges compared to its size. Maybe another representation can be found, where only the effective adjacency relations are stored. A possibility could be to make use of hash tables to reduce the waste of memory space.

The initial design of the *LINKED_GRAPH* implementation proposed by Bernd Schoeller used a linked list to store the nodes. For the current version, that list was replaced by a hash table which brought a significant performance boost. There might be other optimizations to be uncovered to make that implementation even more efficient and elegant.

#### Algorithms

The current version of the graph library supports the commonly used graph algorithms: *cycle detection*, *shortest path* and the *minimum spanning tree*. There exist many further graph algorithms which could be added as an extension:

- Find the *clique* of a graph
- Compute the *independent set*
- Determine $n$-connectedness of components
- Test for *isomorphism*

### 5.2.2 Topological sort

The topological sort implementation is very elegant, robust and efficient. Only two output strategies for unconstrained elements could be implemented so far. When the next version of Eiffel [7] is available, the strategies *first-in-first-out* and *last-in-last-out* should be added to provide even more flexibility.

### 5.2.3 Balanced trees

The B-trees we have implemented are part of the category of *balanced trees*. To reflect that fact, the class *BALANCED_SEARCH_TREE* has been introduced as an intermediate class between the generic *TREE* class and the *B_TREE* implementation. The class provides an interface for future implementations of other balanced trees. The tree cluster could be extended by *AVL-trees* or *splay trees*.

# Acknowledgments

I would like to thank my supervisor Dr. Karine Arnout for her helpful feedback and reviews of my report. Further, I want to thank Prof. Dr. Bertrand Meyer for giving me the opportunity to do this master thesis in his group. Special thanks go to Beat Fluri and Rolf Bruderer for their valuable ideas and support in many situations. I am particularly grateful to my parents for always supporting me at the best during my studies.

# References


[1] Bernd Schoeller. *Graph library for Eiffel.*
http://se.inf.ethz.ch/people/schoeller/index.html

[2] J.B. Kruskal. *On the Shortest Spanning Subtree of a Graph and the travelling Salesman Problem.*
Proc. AMS 7, 1956

[3] ISE EiffelStudio 5.4
http://www.eiffel.com

[4] Graphviz - graph drawing tools.
AT&T labs
Available from http://www.graphviz.org

[5] D.E. Knuth. *The Art of Computer Programming, Vol.3: Sorting and Searching.*
Addison-Wesley, Reading, Massachusetts, 1973

[6] Bertrand Meyer. *Touch of Class. Learning to program* well *with Object Technology and Design by Contract* (in preparation).
Prentice Hall PTR.
Available from http://www.inf.ethz.ch/personal/meyer/down/touch.
Accessed August 2004

[7] Bertrand Meyer: Eiffel: *The Language, Third edition* (in preparation).
Prentice Hall PTR.
Available from http://www.inf.ethz.ch/personal/meyer/#Progress.
Accessed September 2004

[8] Bertrand Meyer. *Object-Oriented Software Construction.*
Prentice Hall PTR, 2nd edition, 1997

[9] T. Ottmann, P. Widmayer. *Algorithmen und Datenstrukturen.*
Spektrum Akademischer Verlag, 3rd edition, 1996