Aalto University
School of Science
Degree Programme in Computer Science and Engineering

Janne Huttunen

Design, Implementation and Evaluation of Efficient Data Storage Algorithms in Segmented Memory Model

Master's Thesis

Espoo, July 4, 2014

Supervisor: Professor Eljas Soisalon-Soininen

Aalto University
School of Science

Aalto University, P.O. BOX 11000, 00076 AALTO
www.aalto.fi

Abstract of master's thesis

Author:	Janne Huttunen
Title of thesis:	Design, Implementation and Evaluation of Efficient Data Storage Algorithms in Segmented Memory Model
Degree programme:	Degree Programme in Computer Science and Engineering
Thesis supervisor:	Prof. Eljas Soisalon-Soininen
Code of professorship:	T-106
Department:	Department of Computer Science and Engineering
Date:	July 4, 2014 **Language:** English
Number of pages:	65 + 11

The data structures and algorithms used for storing and manipulating the ever growing data sets are the most crucial factor that affects the performance of a computer based system. During the past decade or so, most systems have begun to provide a standard set of high performance data structures either as system libraries or via direct support in the programming languages. Some legacy systems however have not quite kept up with this development and still require additional effort from the programmer to achieve good performance scalability.

The objective of this work is to design, implement and evaluate a suitable set of selected data structures on top of one such legacy system, the DX 200 telecommunications platform. This task is complicated by the somewhat eccentric nature of the platform including the memory model it uses. While the modern IA-32 CPU hardware still fully supports the segmented memory model, the world has almost completely moved on and left it behind. In practice this means that it is almost impossible to get any modern high quality tools like e.g. compilers for it and the performance of segment operations hasn't been nearly as aggressively optimized as other CPU functions. All these things always need to be taken into consideration when trying to implement high performance software for the platform and it is almost impossible to implement anything without making at least some trade-offs due to them.

The end result of this work is a tested standard library for the DX 200 platform that implements two types of lists and three types of binary trees and can be used as a basis for future expansion and further evaluation of implementation alternatives. The evaluation done on the implementation clearly shows that even in this kind of system, the balanced trees scale very nicely while the array that is the only data structure provided natively by the used programming languages doesn't scale all that well. This result is totally in line with all the theoretical background and so for its part acts as further proof the validity of the produced implementation.

Keywords: DX 200, segment, memory model, algorithm, binary search tree, red-black tree, splay tree

Aalto-yliopisto
Perustieteiden
korkeakoulu

Aalto-yliopisto, PL 11000, 00076 AALTO
www.aalto.fi
Diplomityön tiivistelmä

Tekijä:	Janne Huttunen
Työn nimi:	Design, Implementation and Evaluation of Efficient Data Storage Algorithms in Segmented Memory Model
Koulutusohjelma:	Tietotekniikan koulutusohjelma
Valvoja:	Prof. Eljas Soisalon-Soininen
Professuurikoodi:	T-106
Laitos:	Tietotekniikan laitos
Päivämäärä:	4. heinäkuuta, 2014 **Kieli:** englanti
Sivumäärä:	65 + 11

Alati kasvavien datajoukkojen tallennukseen ja käsittelyyn käytetyt tietorakenteet ja algoritmit ovat kaikista kriittisimmät tekijät, jotka vaikuttavat tietokonepohjaisen järjestelmän suorituskykyyn. Noin viimeisimmän kuluneen vuosikymmenen aikana useimmat järjestelmät ovat alkaneet tarjota yleistä joukkoa korkean suorituskyvyn tietorakenteita joko kirjastoina tai ohjelmointikielen tarjoaman suoran tuen kautta. Jotkin vanhat järjestelmät eivät kuitenkaan ole aivan pysyneet mukana tässä kehityksessä ja yhä vaativat ohjelmoijalta ylimääräistä vaivaa hyvän suorituskyvyn saavuttamiseksi.

Tämän työn tavoite on suunnitella, toteuttaa ja evaluoida sopiva kokoelma tietorakenteita erään tällaisen järjestelmän – DX 200 -tietoliikennealustan – päällä. Tätä vaikeuttaa alustan jokseenkin erikoinen luonne mukaan lukien sen käyttämä muistimalli. Vaikka nykyaikainen IA-32 -suoritin yhä tukee segmentointia, maailma on lähes täysin siirtynyt eteenpäin ja jättänyt sen jälkeensä. Käytännössä on lähes mahdotonta saada sille mitään nykyaikaisia työkaluja, kuten kääntäjiä, eikä segmenttioperaatioita ole optimoitu läheskään yhtä agressiivisesti kuin muita suorittimen toimintoja. Kaikki tämä tulee aina ottaa huomioon, kun yritetään toteuttaa alustalle suorituskykyistä ohjelmistoa ja on lähes mahdotonta toteuttaa mitään ilman vähintään joitain kompromisseja.

Tämän työn lopputuloksena on testattu kirjasto DX 200 -alustalle, joka toteuttaa kaksi erityyppistä listaa ja kolme binääripuuta. Sitä voidaan myöhemmin tarvittaessa laajentaa tai käyttää erilaisten toteutusvaihtoehtojen tutkimiseen. Toteutukselle tehty testaus osoittaa, että myös tällaisessa järjestelmässä tasapainotetut puut skaalautuvat kauniisti toisin kuin taulukot, jotka ovat ainoa tietorakenne, jonka käytetyt ohjelmointikielet tarjoavat suoraan. Tämä tulos on täysin linjassa teoreettisten odotusten kanssa ja näin omalta osaltaan myös todistaa tehdyn toteutuksen oikeellisuuden.

Avainsanat: DX 200, segmentti, muistimalli, algoritmi, binäärinen hakupuu, punamusta puu, splay-puu

Acknowledgements

This Master's thesis is the culmination of a very long personal journey that started for me already almost three decades ago. Some parts of it have been much harder and taken way more time than I could have ever anticipated, but now this venture is finally complete. Despite of some hardships on the way, I have thoroughly enjoyed most of it and don't have any regrets for embarking on this path. I truly hope that it will continue to lead into interesting places with intriguing new challenges and opportunities also in the future.

I wish to take this opportunity to thank my long time employer Nokia Solutions and Networks Oy (NSN) for providing the required tools and other equipment used during this work. Over the years I have had the privileged opportunity to work with so many great colleagues on lots of interesting things and I sincerely hope that the best things are yet to come.

I would also like to thank my thesis supervisor Professor Eljas Soisalon-Soininen and Dr. Riku Saikkonen for their comments and other insights on this work. They were very valuable for shaping the final structure of the work and the result just would not have been the same without them.

Finally, I would like to thank my parents, especially my mother, who made all this possible in the first place.

Espoo, July 4, 2014

Janne Huttunen

Contents

Abbreviations

ABI	Application Binary Interface
API	Application Programming Interface
BST	Binary Search Tree
COFF	Common Object File Format
CPU	Central Processing Unit
ELF	Executable and Linkable Format
FIFO	First In, First Out
GDT	Global Descriptor Table
GPRS	General Packet Radio Service
GSM	Global System for Mobile Communications
HPC	High-Performance Computing
IA-32	Intel Architecture, 32-bit
IPC	Inter-Process Communication
ISO	International Organization for Standardization
ITU	International Telecommunication Union
LDT	Local Descriptor Table
LIFO	Last In, First Out
MMI	Man-Machine Interface
MMU	Memory Management Unit
NSN	Nokia Siemens Networks
NUMA	Non-Uniform Memory Access

OS	Operating System
PC	Personal Computer
PRNG	Pseudorandom Number Generator
R&D	Research and Development
RAM	Random Access Memory
TETRA	Terrestrial Trunked Radio
TLB	Translation Lookaside Buffer
TNSDL	TeleNokia Specification and Description Language
TSC	Time-Stamp Counter
WCDMA	Wideband Code Division Multiple Access

List of Figures

List of Tables

Chapter 1

Introduction

While the recent decades have seen massive increases in CPU processing capability, mostly attributed to the Moore's law [Moore, 1965], it has at the same time been massively outpaced by the simultaneous increase in the storage capacities of both RAM and hard drives caused by the very same law together with related Kryder's law [Walter, 2005]. Together these increases have led into ever expanding sizes of the data sets that are routinely processed by today's computer systems. Therefore efficient data structures and algorithms are arguably even more essential than ever before in order to efficiently store and access the constantly growing data sets the modern applications have to cope with.

In order to improve the quality of used algorithms and to avoid unnecessary duplication of effort the system software should preferably provide a thoroughly tested and proven set of common data structures and algorithms as a platform component. This would relieve individual programmers from having to implement and debug their own versions over and over again which should increase productivity while simultaneously also improving quality and conceivably even increasing the performance of the system. While most of the modern systems fulfill that criterion either via use of modern programming languages that have rich sets of algorithms and data structures built-in [Naftalin & Wadler, 2006; Hellmann, 2011; Josuttis, 2012; Miller, 2012] or via providing a standard set of system libraries implementing the algorithms [Copeland, 2005; Schäling, 2011], some legacy systems like the DX 200 telecommunications platform have remained with their original programming languages and -models and thus currently don't enjoy all the luxuries of the more modern systems.

1.1 Scope of the Thesis

The objective of this work is to select a set of data structures and related algorithms, design and implement efficient implementations of them on top of the DX 200 platform and finally evaluate the performance of the produced algorithms under various conditions. The result is a re-usable library which offers a simple and easy to use yet flexible API. The library implements the selected algorithms and can act as a basis for future expansion to include

other fundamental data structures and their respective algorithms.

1.2 Structure of the Thesis

Chapter 2 introduces the background and basic theory of the data structures and algorithms that were examined in this work. Chapter 3 describes the primary properties of the target system and how those affect the design decisions made during this work. The specific requirements for the design imposed by the target system and how to fulfill them is presented in chapter 4. The actual implementations of the selected data structures and algorithms themselves are presented in chapter 5. Chapter 6 describes the test setup that was used to evaluate the implementation and presents the measured results and their analysis. Finally the conclusions and points for further study are presented in chapter 7.

Chapter 2

Trees

Trees are common abstract data structures that are used for storing and indexing data in computer systems. They are generally used for e.g. representing hierarchical data, optimizing searches and sorting sets of data. For example the associative arrays or mappings found in many modern programming languages and indices that databases use to optimize searches are often internally implemented as some type of tree structures. Therefore they can be characterized as "the most important nonlinear structures that arise in computer algorithms" [Knuth, 1997a]. Due to their versatility and generally good performance characteristics three specific variants of trees were also selected as the data structures that were implemented and evaluated in this work.

This chapter gives a general overview of the tree data structure and some operations that are common to all tree types. First the required basic terminology is defined in section 2.1 and a formal definition of a tree is presented in section 2.2. Next the generic tree traversal algorithms are presented in section 2.3. How different types of tree structures can be divided into different classes, how they differ from each other and what their common features are is discussed in section 2.4. Finally sections 2.5 and 2.6 present more details of the specific tree types and related algorithms that were examined during this work. The actual details of the selected tree types i.e. the way their algorithms differ from each other is discussed later in chapter 5.

2.1 Terminology

While the graph theory in mathematics defines tree just as a connected acyclic graph [Diestel, 2012], the trees in computer science are usually required to be also both directed and ordered. Therefore in the context of this work a "tree" is a connected hierarchical data structure that forms an acyclic, ordered and directed graph. It consists of "nodes" that may contain the "keys" or "values" and of "links" or "edges" that connect the nodes into each other. Each normal node of a tree has zero or more "child" nodes and exactly one "parent" node. There is a single (topmost) node in a tree that has no parent and it is called the "root" of the tree.

Nodes that have the same parent are called "siblings".

Nodes that don't have any children are called "leaf" or "terminal" nodes. All the other nodes (i.e. the ones that do have at least one child) are called "internal", "inner", "non-terminal" or "branch" nodes.

A group of nodes consisting of one node and all its descendants is called a "subtree". If the root of a subtree is some other node than the root of the whole tree, the subtree is a "proper subtree". Every node of a tree is a root of its own subtree.

The amount of steps needed to traverse from the root node to a certain node is the "depth" of the node. The maximum amount of steps needed to traverse from the root node through a leaf node to the bottom of the tree is called the "height" of the tree. The relative difference of the heights of the subtrees of the root is called the "balance factor". Manipulating the tree in order to improve the balance factor is called "balancing" the tree.

2.2 Definition

Using the terminology presented in section 2.1 a tree can be formally defined as a finite set T of one or more nodes such that

1. there is one specifically distinguished node called the "root" of the tree.
2. the other nodes (excluding the root) are partitioned into zero or more disjoint sets $T_1, T_2, \ldots, T_k$ each of which is in turn a tree. These trees $T_1, T_2, \ldots, T_k$ are called the "subtrees" of the root.

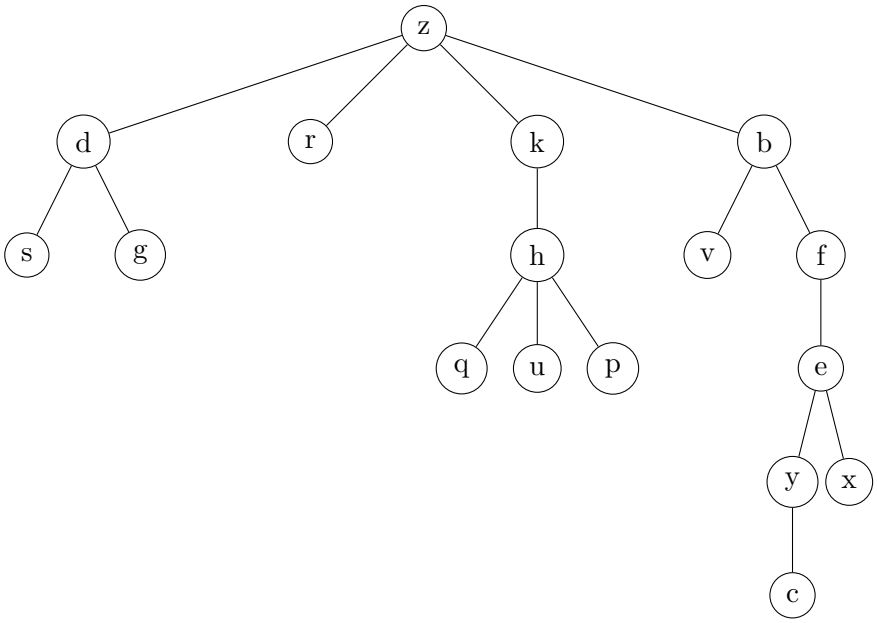
This definition [Knuth, 1997a; Weiss, 1993] is recursive i.e. it defines tree as a single node and a set of trees connected into it by directed links.

Trees can be also defined in non-recursive manner, but the recursive definition is actually very natural and appropriate for trees since recursion is a very fundamental feature of the data structure and the related algorithms. The biological trees growing in nature, after which the data structure is named, similarly can be thought as a set of subtrees (branches) that recursively have their own subtrees until leaves are reached and the recursion terminates. However unlike biological trees, the tree data structures are by convention usually drawn so that the root of the tree is at the top of the diagram, not at the bottom. This distinction is important when using the established terminology describing various tree algorithms i.e. "top-down" algorithms actually start from the root of the tree and traverse towards the leaf nodes while "bottom-up" algorithms symmetrically start from the leaves and work towards the root [Knuth, 1997a].

2.2.1 Forests

The definition of the treelike structures can also be generalized to cover disjoint sets of trees that are commonly called "forests". A forest can be created e.g. simply by deleting the root node of the tree at which point the remaining nodes form a forest. Conversely if a new

Figure 2.1: Example of a Tree



node is added to a forest and all existing trees are considered to be subtrees of the new root node, a single tree is formed. Therefore the terms “tree” and “forest” are often used almost interchangeably [Knuth, 1997a].

However forests and their specific algorithms as such are not in the scope of this work and therefore any more formal definition of them in here is omitted.

2.3 Tree Traversal

One of the very basic operation of any data structure is performing some specific action on each of the elements of said data structure. That is also true on trees and in that case the operation is called “tree traversal” which basically means iterating over all nodes of the tree. This section gives a brief overview of the algorithms that are usually used to implement this.

Performing an action on a node of a tree is commonly called “visiting” the node. Using that definition tree traversal is then defined as the combined act of visiting every node in a tree in an ordered and systematic manner so that each node is visited exactly once [Knuth, 1997a]. Unlike lists or arrays, trees don’t have any obviously unambiguous order in which the nodes should be visited i.e. there are multiple different – but equally valid – orders in which the nodes may be visited in a particular tree. The selection of the used traversal algorithm is based entirely on the needs of the application in question and varies from case to case.

The tree traversal algorithms can be divided in two major classes that are described in more detail in the following sections. Generally the traversal algorithms don’t depend on

the internal structure of the tree i.e. the same two major classes apply equally to all types of trees and e.g. the commonly used binary tree traversals apply to all types of binary trees. Furthermore the algorithms themselves are very simple and really cannot be implemented in any particularly interesting way i.e. they don't need any meaningful designing. Therefore the algorithms that were actually implemented for tree traversal in this work are considered to be obvious and are presented here instead of chapter 5.

2.3.1 Depth First

Depth first tree traversal means that the nodes are traversed so that the tree is always examined as far from the root as possible (i.e. until a leaf node is reached) before backtracking and examining a different alternative. It was known already long before computers, at least in the nineteenth century where it was used as a technique for solving mazes [Even, 2011]. It can be easily implemented using an explicit stack or recursion and requires linear time (every node is visited once) and in worst case linear space (for the stack that is used for backtracking) to the size of the tree. The basic idea of the algorithm is that upon traversing a subtree, the children of the root are always pushed to the stack so that they can be returned to when backtracking. After the push is done the top of the stack is popped and traversed next. In practice the LIFO nature of the stack means that the depth first algorithm will always traverse the children of each node before its siblings and hence the name "depth first".

In the following sections three common variants of depth first binary tree traversal algorithms are presented. The only difference between them is when the root of each subtree is visited with regards to traversing its children; before, between or after. All of the variants presented were also actually implemented as a function in the library for each of the binary trees that were implemented in this work. The function will visit all the nodes of a tree in the order that is given as a argument and call the given function for each node. The true operation of the library function is the same regardless of the specific tree type and is actually implemented with recursion, which places some restrictions on its usage (see chapter 4.1.5). If that proves itself to be an actual problem, in the future an alternative method needs to be designed and implemented i.e. one that still requires linear space but has smaller constant factor than the simple recursion has.

Preorder

Preorder traversal of a tree means that the root node of every subtree is visited before (pre) of any of its children is traversed [Weiss, 1993]. This is useful for situations where the operations on the subtrees depend on the operation on the root like e.g. if the operation is to label each node with its depth . The actual algorithm for this is presented in Algorithm 1.

Algorithm 1 Preorder traversal in a binary tree

1: visit the root node
2: traverse the left subtree
3: traverse the right subtree

Inorder

The second presented alternative is inorder traversal which means that the root node of every subtree is visited in-between (in) traversal of its children [Weiss, 1993]. The actual algorithm for it is shown in Algorithm 2.

Algorithm 2 Inorder traversal in a binary tree

1: traverse the left subtree
2: visit the root node
3: traverse the right subtree

Postorder

Finally the third alternative, postorder traversal, means that the root node of every subtree is visited after (post) both of its children have been traversed [Weiss, 1993]. This is useful for situations where the operations on the root depend on the operations on the subtrees like e.g. calculating the height of a node. The height is always one more than the maximum height of the subtrees, so the heights of the subtrees need to be calculated first. The actual algorithm for postorder traversal is defined in Algorithm 3.

Algorithm 3 Postorder traversal in a binary tree

1: traverse the left subtree
2: traverse the right subtree
3: visit the root node

2.3.2 Breadth First

Breadth first traversal of a tree is very similar to the depth first traversal presented in chapter 2.3.1 with the only difference that instead of a stack the potential nodes to traverse are put into a FIFO queue. This again requires linear time (every node is visited once) and linear space (for the queue) to the size of the tree. In practice the FIFO nature of the queue means that the siblings of each node are always traversed before its children and hence the name “breadth first”.

Breadth first traversal is not implemented in the scope of this work because it was not deemed necessary at this time. It can be added later if some real use-case for it arises.

2.4 Types of Trees

Every tree regardless of its type has the same basic structure and elements i.e. nodes linked into each other in a hierarchical order. What separates them from each other is constraints put on the (external) structure of the tree and the (internal) management algorithms used for placing the keys and maintaining the tree.

2.4.1 External Differences

Externally the different types of trees are separated by how many keys a single node can store and how many children a single node may or must have. The most common tree types are binary trees (section 2.5) that usually store *one* key and have $\{0 \dots 2\}$ children per node and various b-tree variants (section 2.6) that generally store $\{1 \dots n\}$ keys and have *zero* or $\{2 \dots n + 1\}$ children per node. I.e. the external type defines how the tree looks, but not exactly how it is constructed or maintained on insertions and deletions.

2.4.2 Internal Differences

Internally these tree types are further split into multiple subtypes based on the algorithms used to maintain the tree. These algorithms may e.g. determine where in the tree each key is placed and how the balance of the tree is maintained. Different tree maintenance algorithms then give the tree its fundamental properties which most of the time mean things like e.g. logarithmic insertion, deletion and search times.

The following sections give a brief overview of some of the tree types commonly used for storing and accessing data in computer programs. The in-depth implementation details with the exact algorithms are presented later in chapter 5. There exists dozens of different tree types and related algorithms, but only a select few of them are actually included in this overview. They were chosen so that they are relevant in the scope of this work.

2.5 Binary Trees

A binary tree, as the name suggests, is type of a tree data structure where each node has maximum of two children that by convention are commonly called the “left” and the “right” child. The nodes in a binary tree can be ordered according to selected criteria to implement different data structures, like e.g. binary heaps and binary search trees. In this work three different types of binary trees – basic unbalanced binary search tree, red-black tree and splay tree – will be implemented and their performance characteristics evaluated on real target hardware. The implementation details with the used algorithms are presented in chapter 5 and the evaluation setup and results in chapter 6.

2.5.1 Binary Search Tree

A binary search tree (BST) is a type of binary tree, where the key of every node of the left subtree is less or equal to the key of the root node and the key of every node of the right subtree is greater or equal to the key of the root node. This rule is the so called *binary search tree property* and the same rule also recursively applies to every subtree of the BST [Cormen *et al.* , 2001].

In the scope of this work only a variant of the BST where there are no duplicate keys is considered i.e. the keys in any two distinct nodes of the tree are always truly less or greater, never equal. From that stricter definition and its recursive application it naturally follows that every node of the tree also must have a unique key, the smallest key is always in the leftmost and the greatest key in the rightmost node of the tree. It also follows that accessing the nodes in *inorder* i.e. recursively accessing first the left subtree, then the root node and finally the right subtree (see section 2.3.1) will access the nodes in strictly ascending order as sorted by the keys of the nodes. These fundamental properties apply as-is also to the balanced binary trees presented later in this chapter i.e. they can be considered to be just extensions of a basic BST.

In a perfectly balanced BST, the computational complexity of searching, insertion and deletion is relative to $\log N$, where N is the number of nodes in the tree. This is because in a binary tree on each depth level d of the tree there can be maximum of 2^d nodes ($d = \{0 \dots n\}$) and thus the maximum amount of nodes in a binary tree with height h is

$$maxnodes = \sum_{d=0}^{h-1} 2^d = \frac{1-2^h}{1-2} = 2^h - 1 \tag{2.1}$$

From equation 2.1 it conversely follows that a perfectly balanced binary tree of N nodes has a height h of

$$h = \log_2(N+1) \tag{2.2}$$

I.e. the height of the tree is proportional to the logarithm of the number of nodes in the tree. Since the computational complexity of searching, insertion and deletion is linear to the height of the tree, equation 2.2 means that at the same time they are logarithmic to the number of nodes in the tree.

However the basic BST algorithms don't do anything to maintain any kind of balance in the tree. This means that in the worst case scenario every node of the BST only has one child and the tree becomes effectively analogous with a linked list as can be trivially demonstrated by inserting the nodes to the tree with their keys sorted in strictly ascending or descending order. In that kind of degenerate tree all the operations have linear execution time, just like normal list operations inherently have.

Despite this limitation a pure BST can be useful in some situations where the access patterns of the data are known to be suitable for it i.e. random enough insertion and deletion patterns are guaranteed [Pfaff, 2004]. Its main advantage is very simple implementation with

relatively fast (as long as the order of the keys truly is random) insertion and deletion that don't require any rotations or other additional manipulations of the tree. In the scope of this work, the BST also acts as a very good reference into which to compare the performance of the more advanced (balanced) data structures presented in sections 2.5.2 through 2.5.4. Therefore a plain BST is is one of the data structures actually implemented in this work and the details of the algorithms are presented in chapter 5.1.

2.5.2 AVL tree

An AVL tree is a self-balancing binary search tree named after its inventors Georgii Adelson-Velski and Evgenii Landis [Adelson-Velski & Landis, 1962]. It was first published in 1962 and is considered to be the first such data structure invented [Sedgewick, 1983].

The basic idea is of an AVL tree is to maintain a maximum height difference of one between the left and right subtree of any node of the tree. Is is accomplished by performing necessary rotations in the tree whenever the tree changes i.e. a node is inserted or deleted. The binary tree rotation is shown in Figure 5.3 that shows the rotation in red-black tree, but the same principle applies also in AVL tree, just without the colors. Searching in the AVL tree doesn't do anything special and is implemented like in a plain BST (see chapter 5.1).

In principle an AVL tree maintains a very rigid balance in the tree to achieve $O(\log N)$ insertion, deletion and searching performance. This causes it to have relatively high insertion and deletion costs due to the high amount of required re-balancing rotations, but for the same reason it also has highly efficient searches. An AVL tree has been shown to perform especially well for data where insertion is done in sorted order and searches in random order [Bell & Gupta, 1993; Pfaff, 2004]. It also is a very good choice for any data structure where the number of searches is sufficiently higher than the number of insertions and deletions.

While the AVL tree in itself is a fine data structure and relatively simple to implement, in the context of this work an AVL tree will not be actually implemented and is presented here just as a considered – but ultimately discarded – alternative. This is mostly because it's performance is expected to be too similar to the red-black tree to warrant implementing both at this point [Štrbac Savić & Tomašević, 2012]. Furthermore unlike with red-black tree, it is not straightforward to implement AVL tree using only single top-down pass like suggested by the general requirements imposed upon the design in chapter 4.1. Together these facts make the red-black tree a much better choice for this work but an AVL tree implementation can be added to the library later if a real use-case for it arises.

2.5.3 Red-Black Tree

A red-black tree is another variant of a binary search tree that like an AVL tree also guarantees logarithmic running time for searching, insertion and deletion. Like in AVL tree, this is accomplished by rebalancing the tree every time nodes are inserted or deleted, although the exact algorithm how it is done is different. It was first invented by Rudolf Bayer, who called them "symmetric binary B-trees" [Bayer, 1972]. The current commonly used name "red-

black tree" was presented in a paper by Leonidas J. Guibas and Robert Sedgewick [Guibas & Sedgewick, 1978]. An alternative variant, a left leaning red-black tree was introduced by Sedgewick in his paper "Left-leaning Red-Black Trees" [Sedgewick, 2008]. The intention of this variant is to reduce the number of symmetric cases present in regular red-black trees by placing stricter requirements for the structure of the tree. This was done with the intention that the implementation could be simplified by getting rid of the separate implementation for the symmetrical left and right leaning cases by allowing only the left leaning case as a valid state of the tree. This is the variant that is implemented in this work and is called simply "red-black tree" or "RB tree" in this document.

Red-black trees are a good compromise when the ratio of insertions and deletions to searches is higher and may contain sequential keys [Pfaff, 2004]. Compared to AVL trees the RB tree has a bit less strict balance which may cause it to have a slightly higher height for the same set of keys, but at the same time makes the balancing simpler (less rotations required to maintain the balance) and thus insertion and deletion are generally faster. There also exists efficient algorithms for doing both insertions and deletions in RB tree using only a single top-down pass, which makes it a good match for the design requirements presented in chapter 4.

The RB tree is one of the data structures selected for implementation and evaluation in this work. The details of the implementation and the operation of the algorithms are presented in chapter 5.2.

2.5.4 Splay tree

A splay tree is yet another type of a self-adjusting binary search tree. It was originally invented by Daniel Sleator and Robert Tarjan [Sleator & Tarjan, 1985]. All the basic operations on a splay tree run in $\log N$ *amortized* time, which means that while any single operation may take $O(N)$ time, a sequence of m operations runs in $m \log N$ time. The worst case of a splay tree is the same as the worst case of a BST i.e. in a degenerate tree every node has only one child and the tree is effectively a linked list. This situation happens when e.g. all the keys are inserted in strictly ascending (or descending) order to the tree. However, as is shown in chapter 6.4.3, the performance of the splay tree compared to a plain BST is quite different even in this situation and unlike BST, the splay tree will gradually balance itself as soon as any non-sequential accesses are done.

The splay tree also has the additional interesting property that the recently accessed nodes are always at or near the root. This makes the performance very good when the access pattern has reasonable amount of "locality" i.e. the same subset of nodes is accessed multiple times before the access moves to another subset of the tree [Pfaff, 2004]. This is very common in e.g. caches and other similar structures. Under that kind of conditions splay tree also has very good locality of access when considering the RAM of the computer. This can have significant performance benefits on modern computers where main memory has major access latency compared to the internal cache of the CPU [Lee & Martel, 2007].

The in addition to the theoretical linear worst case performance the main disadvantage of

a basic splay tree is the relatively high cost of constant rotations. Some solutions do exist to address also this problem. Generally the amount of splaying can be reduced by introducing probabilistic behavior without impacting the fundamental operation of the data structure [Fürer, 1999; Albers & Karpinski, 2002]. Also methods for reducing splaying based on the structure of the tree with regard to the current access pattern do exist [Lee & Martel, 2007; Aho *et al.* , 2008]. However these methods and their operation are not in the scope of this work.

The splay tree is one of the data structures selected to be implemented in this work. The details of the implementation and related algorithms are described in chapter 5.3.

2.6 B-Tree

A B-tree is a data structure invented by Rudolf Bayer and Edward McCreight [Bayer & McCreight, 1972]. It can be thought as a generalization of a binary tree. Unlike a binary tree node, a B-tree node may include multiple keys and have corresponding number of child nodes. This reduces the amount of re-balancing operations needed when inserting or deleting keys. This remains true even while, unlike a binary tree, a B-tree always maintains a perfect balance of nodes. Note that since different nodes can include different amount of keys, having a perfect balance of nodes doesn't necessarily mean perfect balance of keys. While all the subtrees of each node always have the same height, they still may have different number of keys in them.

Generally the B-tree is defined to have the following properties:

- Each path from the root to any leaf has the same length.
- Each node except the root and leaves has at least $k + 1$ children. The root is either a leaf node or has at least two children.
- Each node has at most $2k + 1$ children.

The actual keys of a B-tree are stored in the internal nodes. Each (non-root) node can thus store between k and $2k$ keys. The value k is called the "order" of the tree. The keys themselves are ordered in the tree so that for each node its first subtree contains all the keys smaller than the smallest key a_1 of the node, the second subtree all the keys between a_1 and a_2 etc.

The reason to set the upper and lower limits of children like this is to make it always possible to split a node into two separate nodes or to merge two nodes into to one node if needed. These operations are performed to maintain the balance of the tree when nodes are inserted to or deleted from the tree.

A B-tree supports searching, insertion and deletion in logarithmic amortized time. This is accomplished because the tree always maintains a perfect balance i.e. every leaf node is at equal depth from the root.

In theory, the order of the tree may be chosen arbitrarily to suit the specific application. In practice, high order B-trees are usually used when the "unit size" of data is naturally

large and the corresponding access time is long. They are very well suited for e.g. disk based data storage where the smallest accessible unit is the size of one or more disk block(s) (commonly 512 bytes or more on modern hardware) and the access latency between blocks is measured in milliseconds (vs. single byte unit size and nanosecond latencies in RAM). Many file systems, like e.g. BTRFS [Rodeh *et al.* , 2013], NTFS [Russinovich *et al.* , 2012] and HFS+ [HFS+, 2004], and on-disk databases utilize some variant of B-trees for different indexes and directories. Due to its widespread usage, the B-tree and its variants can be considered to be the de facto standard way of organizing indexes in databases [Comer, 1979].

In theory, high order B-trees are also better for in-memory uses than low order trees. They have lower depth and therefore operations on the tree are supposed to be faster. However this theory completely overlooks the fact that in the real world situations the required intra-node operations also take time and the actual tree operations are very fast when the whole data structure is always resident in RAM (which implies also no paging to disk). Depending on the actual internal structure of the node and the corresponding algorithms the intra-node operations may be equally or even more expensive than the inter-node operations (like e.g. node to node traversal or modification of the links between nodes) [Hansen, 1981]. Also, another downside for in-memory uses of high order B-trees is that they tend to cause too much storage space overhead due to partially filled nodes.

In practice for in-memory uses it is common to limit the order of the tree. It has been empirically shown that if operating on main memory speeds the optimal choice for the order is 3 or 4 [Weiss, 1993]. In such a use case higher order trees are not an advantage. Due to these features and other design requirements presented in chapter 4 a generic B-tree is not considered to be in the scope of this work and thus is not implemented by the library.

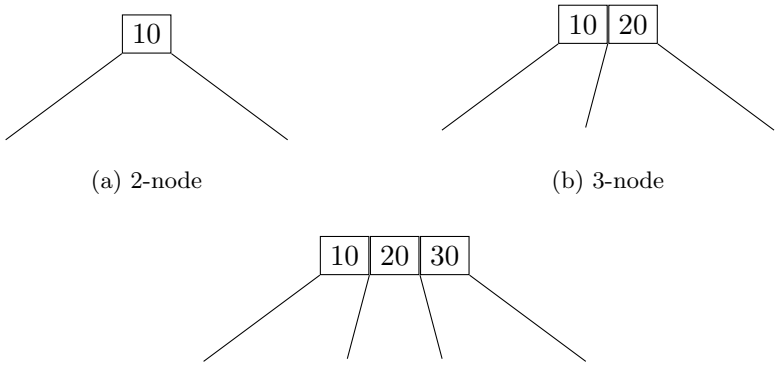
2.6.1 2-3-4-Tree

A 2-3-4-tree is a limited variant of B-tree where every non-leaf node has 2, 3 or 4 children. The different nodes types are correspondingly called “2-node”, “3-node” and “4-node” (see Figure 2.2).

2-3-4-trees are a special case of the generic B-tree and thus share all the important properties, like the perfect balance of nodes and therefore logarithmic running time of basic operations. Due to the limited order of the tree, 2-3-4-trees are better suited for pure in-memory data storage than higher order B-trees. However the most interesting property of them regarding this work is that the red-black trees (see section 2.5.3) are an isometry of 2-3-4-trees and thus share the basic algorithms with each other [Hinze, 1999].

The following sections represent an overview of insertion and deletion of keys in a 2-3-4-tree. Actually implementing a generic B-tree or a 2-3-4-tree is not in the scope of this work, but the basic theory of 2-3-4-tree operations is presented here because due to the isometry between it and red-black trees these algorithms form the basis from which the corresponding red-black tree operations presented in section 5.2 are derived.

Figure 2.2: 2-3-4-Tree Node Types



(a) 2-node

(b) 3-node

(c) 4-node

Insertion

Insertion into a 2-3-4-tree is done by first searching the leaf node where the new key belongs. If the leaf node has room for the new key (it is a 2-node or 3-node), the key is inserted there. If the key doesn't fit to the leaf node (it is a 4-node), the node is split into two 2-nodes and the middle key, the new key is added to either the left or right node as appropriate and the middle key is inserted into the parent node. Inserting to the parent may need a similar split again and so on, until the root of the tree is reached. If there still isn't room for the key, a new root is created with the middle key. This is the only way the height of the tree grows.

Deletion

Deletion from a 2-3-4-tree is a somewhat more complex operation. At first the key to be deleted is searched from the tree. If it is found in an internal node, it is first swapped with a leaf key that precedes it. Now the key is in a leaf node and it is deleted. If the node becomes empty, but either sibling node has more than one key, a key is pulled from the parent into the empty node and from the sibling into the now vacant place in the parent. If both siblings are 2-nodes and thus cannot spare a key, they are fused together to form a 4-node with a key pulled from the parent. If the parent was a 2-node it may now be empty and the same operations need to be performed on it and so on all the way up to the root node. If the root is also a 2-node, its children are fused with it into one node and the height of the tree decreases by one.

Chapter 3

Target System

The target of this work is the DX 200 telecommunications platform, which is a modular, fault-tolerant computer platform designed for high availability applications. While the DX 200 network element has a lot in common with a normal general purpose computer running a general purpose operating system, it also has its own peculiar features that affect the way software is designed and implemented on top of the platform.

This chapter gives a brief overview of the DX 200 platform and its major features to give background and insight into things that needed to be considered for the design requirements and implementation decisions that are presented in chapters 4 and 5. The background, history and current status of the platform is first presented in section 3.1. The basic structure of the target hardware is described in section 3.2 and the runtime execution environment including the operating system in section 3.3. Then perhaps the most distinctive feature of the whole platform (by today's standards) – the used memory model – with its pros and cons is presented in its own section 3.4. Finally the programming environment including supported programming languages and provided methods of distributing shared code and definitions is described in section 3.5.

3.1 Background

The DX 200 was originally developed for a digital telephone exchange by Televa Oy during the early 1970s [Mäkinen, 1995]. From the late 1970s the DX 200 development was continued by Telefenno Oy, a joint venture between Televa and Nokia Oy. In 1981 Nokia acquired the majority of the company shares and its name was changed into Telenokia Oy (later Nokia Telecommunications Oy and Nokia Networks). Since 2007 the DX 200 has been maintained and developed by Nokia Siemens Networks (later Nokia Solutions and Networks). It is currently used as the platform in multiple types of network elements for GSM, GPRS, WCDMA and TETRA networks.

Despite of the multiple decades of history the DX 200 platform has, it still doesn't provide any directly re-usable implementations of data structures and related algorithms (like e.g.

lists, queues, trees etc.). This causes every programmer either to roll their own (with possible bugs or suboptimal implementations) or worse yet to completely ignore the whole issue and just use the array construct provided by the programming language. There are no real statistics available, but empirically (by examining quite a large number of different programs) especially the latter solution appears to be a quite commonly the selected choice. Unfortunately it is relatively common to find a DX 200 program using no more sophisticated data structure than a plain array. In some cases this might be justifiable if e.g. the amount of data is small, but in other cases it has also caused quite serious performance issues. Since arrays in general have linear ($O(N)$) performance at best, they really don't scale well to large amounts of data.

3.2 Hardware

A DX 200 based network element normally consists of multiple loosely coupled computer units that perform different roles in the system. The computers are connected into a dedicated message bus that they can use to communicate with each other. A typical network element consists of dozens of these computer units. The configuration is scalable and can be adjusted based on targeted capacity. A large system can contain up to a few hundred units, if needed.

Since the software implemented in this work doesn't include any distributed components (i.e. it doesn't need to communicate with any external entities), the distributed nature of the overall system can be completely ignored. In the scope of this work the hardware consists of only one computer running the software.

The computer unit itself is a proprietary design and form factor, but it is based on the standard Intel IA-32 CPU architecture that is also used in common desktop PCs. The rest of the hardware design in modern DX 200 computer unit is also heavily influenced by the PC. Industry standard components and designs are used throughout the board and only augmented with proprietary solutions where necessary.

3.3 Execution Environment

The DX 200 runs the NSN proprietary DMX real-time operating system. The OS is based on a microkernel with most of the system services provided by libraries and program blocks external to the kernel. The operating system kernel itself is only responsible for providing the most basic memory management, process creation/destruction and inter-process communication.

The fundamental architecture of the system is based on autonomous process groups called "families". These processes communicate with each other and with other families by using the common IPC mechanism provided by the OS kernel. The IPC is implemented by sending and receiving asynchronous messages and is really ubiquitous throughout the whole DX 200 system. Every process automatically has a "mailbox" capable of sending and receiving these

messages. Almost all of the system services are accessed by sending a request as an IPC message to a known provider. Many of these services also have a synchronous API, but generally those just hide the actual IPC that still happens behind the scenes.

Especially in traditional DX 200 software any kind of real synchronous services or shared resources (like e.g. shared memory) have been almost extinct and only used by the OS kernel and some very specific subsystems (like e.g. the memory file system). In the recent years the situation has changed somewhat due to e.g. porting code to DX 200 from systems where synchronous services are the norm. Still the preferable design choice is to either implement system services asynchronously or to very least make any synchronous services stateless.

3.4 Memory Model

One of the most distinctive features of DMX, especially when compared to most modern operating systems, is the heavy use of the segmented memory model provided by the Intel IA-32 architecture. Nowadays most general purpose operating systems rely on paging provided by the MMU for managing and protecting the memory [Soma *et al.* , 2014] and the segmentation is rarely used except in some specific small embedded systems [Sjödin & von Platen, 2001]. The closest technology that somewhat resembles the segmented memory is probably NUMA [Lameter, 2013] that is commonly used in high end general purpose computers, but even that isn't a very good match since it is mainly about the hardware while the IA-32 segmentation is purely a software issue. In segmentation the underlying memory hardware itself has uniform access characteristics and the differentiating factor is the actual instructions that are used by the program. While some small part of the issues are similar, NUMA essentially exists in a different layer for solving different problems and actually some IA-32 implementations include also NUMA support.

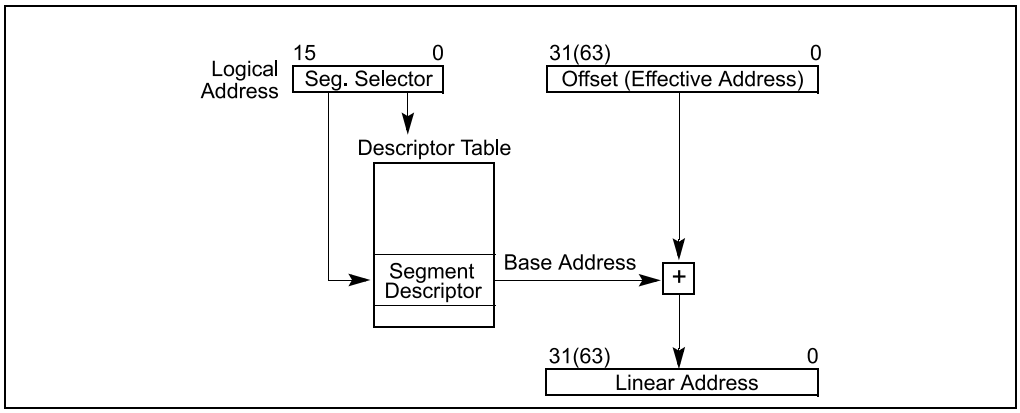
General purpose operating systems (like e.g. Linux or different UNIX variants), even when running on an IA-32 architecture, use the so called "flat" memory model, where logically everything is in one contiguous memory segment. Physically the IA-32 architecture always has the segments present, but the operating systems configure the segment registers so that from the software point of view the different segments do not seem to exist.

DMX instead uses the so called "compact" memory model, where every program visible address consists of a 16-bit "selector" that selects the referenced segment and a 32-bit "offset" that tells the actual byte inside the segment to access. Every accessible segment is described in either GDT or LDT and has

- a base address, which is used for calculating the final linear address
- a limit, which is used for trapping attempts to access beyond the end of the segment
- flags, which tell the type of the segment, allowed operations (read, write, execute) and separate different privilege levels from each other

The selector selects an entry from either GDT and LDT and after all the limit and access checks are done the final address is calculated as

Figure 3.1: Logical Address to Linear Address Translation [Intel, 2009b]



$$address = base + offset \tag{3.1}$$

This linear address is then subject to the normal page table based mapping if MMU is activated. The flat memory model is achieved by setting all segment limits to the maximum value and all segment bases to zero. This means that the limit won't be ever enforced and the linear address will always be the same as the offset.

In a normal DMX process the program code, the data and the stack all are in a different segment. Every shared resource the OS has (library code, data, shared memory files, etc.) generally also has its own segment that resides the GDT. In addition, the segmentation affects also how the private resources of a program are managed. The whole dynamic memory management is based on the concept of allocating and freeing buffers that normally occupy one segment each in the LDT of the process. Similar buffers are also used for internal IPC i.e. every IPC message is its own segment. When sending a message the OS kernel passes the ownership of the segment from one process to another and modifies the LDT of the sending and receiving processes to change the visibility of the message.

3.4.1 Segmentation Advantages

The segmented memory model has multiple benefits, one of which is more strict separation and protection between processes. In generic operating systems, the protection between processes is more commonly based on paging and implemented using the MMU. In that kind of system every process has its own set of page tables and runs in its own virtual address space. Processes don't see each other's memory and thus cannot access it unless special measures are taken to explicitly share memory.

While DMX does also support MMU, it is only used for special purposes, like accessing memory mapped hardware and for on-demand loading of data from hard disk. It isn't used for page based protection. In DMX there exists only one set of page tables i.e. every process in the computer unit and even the DMX microkernel itself run in the same virtual address

space. This makes e.g. context switching very light-weight since, unlike in the multi address space model, there is no need to switch page tables and flush the TLB. At the same time, having only a single address space provides no protection between processes.

Instead, in DMX the private memory of each process is in its own segment that is only visible in its LDT. This limits processes that can see the memory to the ones sharing the same LDT (usually the processes in the same family) and even then accessing the data of another process by accident is quite improbable. Accidental access would require both forming a valid segment selector and consecutively accessing the segment with offset that is below the limit of the segment in question. Since math (addition and subtraction) with pointers doesn't ever change the selector part at all, it is not possible to make the access to overflow from one segment to another by e.g. accidentally adding wrong value to the pointer.

Another very useful benefit of the segments is that they also implement more strict checks for overflows. In IA-32 hardware the segment limits are enforced with byte granularity when the segment size is below one megabyte. In contrast the paging hardware (MMU) can only enforce access restrictions with page size granularity (usually four kilobytes), which means either having huge overhead by having every object placed on its own page or risking access overflowing from one memory object to the adjacent one.

In the light of common stack smashing attacks [One, 1996], another significant benefit of the segmentation is that it automatically gives a strict separation of code and data into different segments and makes it possible to define different access rights for them. In a way it allows partial emulation of Harvard architecture, which is obviously more secure against code injection attacks [Riley *et al.* , 2010]. The data segments (including stack segments) are defined not to be executable while code segments are executable, but not writable. Essentially this allows very similar configuration to the "No-eXecute" page attribute [Intel, 2009b] that was added to the IA-32 architecture recently, but segmentation has existed since the architecture was first introduced. Since the DX 200 platform has significant amount of old deployments that can still remain in use for a decade or more, any new hardware developments cannot easily be taken into use and thus won't help much in the real world.

Additionally, in DX 200 the program code is supposed to be completely immutable which is further enforced by the system periodically verifying the IDs and checksums of all loaded program blocks. This makes it very hard to perform the common attack where malicious code is injected into a running process by deliberately overflowing its stack using improperly formed input. Since in DMX environment the stack is in a completely different segment than the program code, the stack can never spill over any existing code. It still may be possible to inject the malicious code into the stack, but since stack segments are not executable, it cannot ever be run. It should still be noted that like with the "No-eXecute" attribute, this alone won't protect from the more advanced "return-to-libc" and "return-oriented programming" type attacks [Krahmer, 2005; Shacham, 2007; Roemer *et al.* , 2012] that don't require code injection.

3.4.2 Segmentation Drawbacks

While there are clear benefits, unfortunately segmentation also has some really serious downsides. One of them is that the memory model very tightly ties the whole DX 200 system to the IA-32 architecture. While significant portions of both the application and platform software is written in languages that are (at least in theory) portable, it still would be almost impossible to port the system to some other CPU architecture. Even while many other CPU architectures also support some types of segments [Jacob & Mudge, 1998], the actual implementations are so different that the pervasive use of the IA-32 segmentation hardware would be very hard to emulate on top of a CPU that didn't support it natively. Fortunately the very impressive overall performance improvements the IA-32 architecture has seen during the lifetime of the DX 200 platform has made this less of an issue.

The flip side of the coin is that it also isn't always all that simple to port code from outside into DX 200. Even a substantial portion of software that has been specifically designed to be portable, is not really capable to handle the segmented memory model very gracefully. The segmented memory model seems to break a lot of very common assumptions the programmers tend to take for granted. However, with very careful design even some large software projects have been successfully converted to run on top of DMX.

Another significant downside of the segmented memory is that the size of the LDT and GDT is limited by the hardware and either of them can contain only maximum of 8191 entries. This is not all that much when thinking about modern systems with multiple gigabytes of memory equipped into every computer unit. In practice this has led into kind of "hybrid" memory models where e.g. a few large segments are allocated and then subdivided into smaller objects. This gives more flexibility and allows allocating much larger number of objects, but loses a lot of the original benefits of the segmentation.

Other perceived downsides of the compact memory model are that it is hard to find tools that would properly support it (see section 3.5), it may sometimes be harder to understand than the flat memory model and perhaps most importantly, it causes a significant performance hit. Since most modern operating systems don't actively use multiple segments, the segment handling instructions have been seriously left behind when the manufacturers have optimized their CPUs for better performance. What's even worse, the general trend seems to be that the gap is getting bigger on each new CPU generation [Fog, 2014].

While some recent research suggests using IA-32 segmentation for actually increasing the performance [Basu *et al.* , 2013; Soma *et al.* , 2014], this is not applicable to the DMX. Segmentation can increase the performance as long as it both replaces the paging completely (the performance increase comes from avoiding TLB misses) and the programs only access a single segment (they don't need the slow instructions and can use a good modern compiler), neither of which is true in a system running DMX. While it may be possible to gain performance by replacing paging with segmentation in some specific HPC scenarios, DMX still has the paging enabled under the segments and generally every object is a different segment so accessing multiple different segments is very common.

The performance hit of segmentation is actually realized when the access is switched from

one segment into another. Depending on the specific operation, access between segments can currently be up to 6-10 times slower than accessing the local segment. Furthermore the use of multiple segments may also cause stalls in the out-of-order execution that modern IA-32 CPUs do, which means that under some conditions the performance hit may be even greater still [Intel, 2009a]. This performance hit is then further increased because the compiler often cannot prove whether the access is switched from one segment into another and in order to be safe thus needs to generate the code as if the switch was actually done. Partially this is inevitable i.e. it is impossible for the compiler to determine it, but the lack of high quality optimizing compilers for the segmented memory model is definitely also a significant factor in it.

From the programming point of view all this means that there exist two types of both program code and data: "near" and "far". As near (intra-segment) access is significantly faster than far (inter-segment) access, unnecessary far access should be avoided where possible. Naturally the trade-off is that near access cannot reach outside of the current segment.

3.5 Programming Environment

While the DX 200 system runs on hardware platform that is based on the very common Intel IA-32 CPU architecture and, especially in its modern versions, in other ways also very much resembles an ordinary personal computer, it still is an embedded environment [Silander, 1999]. The platform is not self-hosting but a cross-compiler toolchain is used instead. Historically the cross-compilation was done on a VAX/VMS system, but at the present day almost all development work is done on personal workstations that run some version of Microsoft Windows. Like noted earlier, it is relatively hard to find compilers and linkers that support the used memory model. Other than that, in order to have full interoperability between all supported languages and respective tools, the OMF-386 object format is used instead of more common object formats like e.g. ELF or COFF. This is the only object format supported by the old tools, but unfortunately the support for it in modern tools is almost nonexistent. For these reasons a lot of the used tools are proprietary and most of them are quite old and often more or less direct ports from their original VAX/VMS counterparts.

3.5.1 Programming Languages

The DX 200 system supports four main programming languages: PL/M, C, TNSDL and assembly. While some of them are not encouraged for new programs anymore, a lot of legacy code still exists written in all of these and therefore they all still remain very much relevant for the DX 200 software development.

PL/M

PL/M is a programming language for Intel CPUs originally developed by Intel in the 1970s [PL/M, 1987]. It is the language in which most of the oldest DX 200 software was originally written. Since a lot of this software is still used in the present day, the PL/M language itself is also still part of the environment. In the wider usage PL/M is basically obsolete and it has been totally unsupported for years. It is therefore practically impossible to get any modern tools that would support it. Since there really aren't any viable alternatives, the old Intel compiler from 1980s is still used for DX 200 software compilation. It is usable and doesn't have any real showstopper bugs, but since it hasn't been updated since the Intel 386 was released, its ability to produce efficient code for modern processors is obviously somewhat limited.

The use of PL/M in new software is nowadays discouraged. Since the C compilers are configured by default to emit code that is ABI compatible with PL/M, it is even possible to mix PL/M and C code in single program block i.e. write new subroutines for old PL/M programs in C.

C

C is a general-purpose programming language originally developed in 1972 by Dennis Ritchie at the AT&T Bell Labs [Ritchie, 1996]. It was first used for implementing the UNIX operating system on a PDP-11 computer. Since then it has become one of the most popular program languages in the world and its syntax has been mimicked by many newer languages like e.g. Java, JavaScript, C++ and C#. In its modern form the C language is defined by ISO [ISO, 1999; ISO, 2011].

C is very popular as a programming language for implementing system software (e.g. UNIX and Linux kernels are written almost completely in C). This is mostly due to the relatively low level of abstraction (i.e. almost everything that can be done in assembly can also be done using C) while still maintaining a reasonable level of portability. This also gives it high level of efficiency i.e. the resulting program doesn't have any unforeseen space or execution time overheads.

In DX 200 environment C is currently the preferred language for writing complex data manipulation routines. It is possible to write complete programs for DX 200 system in C, but compared to writing the main structure of the process in TNSDL, using only pure C may require much more work and be quite a lot more error prone. In some specific cases, like e.g. extension programs and libraries, using only C is much more feasible.

The DX 200 programming environment currently supports two completely different C compiler toolchains. The old C code is compiled (and usually also linked) using an old C386 toolchain from Intel [C386, 1988]. This compiler also dates back to the Intel 386 era and has practically nothing to do with the modern icc compiler that Intel is currently distributing. Unlike icc, the old compiler supports the required compact memory model, but due to its age and the fact that it hasn't been supported in a long time, it again cannot produce very efficient code for modern processors. Furthermore, since the C language is still evolving

(the latest commonly used standard "C99" was released in 1999, a new upcoming standard "C11" was released in 2011), but the compiler hasn't been upgraded, it is lacking all the modern features of the language.

Another C toolchain provided by the DX 200 programming environment is the so called "CAD-UL" toolchain named after the original vendor [CAD-UL, 1998]. The support for this toolchain is also no longer available from the original vendor, but it has been maintained for NSN by a 3rd party via a support contract [NSN, 2007]. This has made it possible to have e.g. limited C99 support implemented in the compiler. Also the optimizations and code generation have been improved to more modern standards. In some specific cases this has greatly compensated for the performance impact of the compact memory model (i.e. far pointers). It is advisable to use only the CAD-UL toolchain when compiling new C code and old programs are also being gradually converted from Intel tools to CAD-UL. That process is somewhat slowed down by the fact that while both implement the same C language and CAD-UL clearly aims to be very much Intel compatible, especially the old Intel toolchain has a lot of quirks that CAD-UL tools don't reproduce, but old programs still depend on.

TNSDL

TNSDL is a NSN proprietary programming language based on ITU-T SDL-88 language [Lindqvist *et al.* , 1995]. It was originally specifically designed and implemented for programming the DX 200. It has been ported to generate code for also some other platforms, but since the language concepts are quite tightly coupled with the services the DMX operating system provides, a significant effort is needed to adapt each port for the target platform. Therefore it still is mainly used for developing software for DMX based platforms.

The TNSDL isn't actually compiled directly into object modules. It is instead first translated into C, which is then compiled using a C compiler and can then be linked with a normal linker. Despite this, TNSDL still is the "primary" language of the DX 200 programming environment. It is commonly used for implementing the overall framework of processes with possible subroutines written in other languages. The TNSDL has built-in in support for implementing state machines that communicate using asynchronous messages (which is the primary process model for DX 200 programs), which makes it very easy to implement message passing protocols on it. It is not that well suited for complex data manipulation, but since it is naturally fully ABI compatible with C, any complex subroutines can be written in C, where necessary.

The TNSDL is also used for defining all the public interfaces of the DX 200 platform. All shared definitions (like e.g. common data types, IPC messages, API definitions, file structures, alarm definitions, etc.) are first written in TNSDL and then suitable header files for other supported languages are machine-generated from the master file. Since all the supported languages are ABI compatible (with some exceptions that can easily be avoided), the actual language used to implement the service does not affect the user of the interface in any way. This way e.g. almost all the system libraries are available from all of the supported programming languages.

3.5.2 Public Definitions

The public definitions (like e.g. constants, data structures, function prototypes etc.) are stored together in a so called "sack" environment. The general idea is that every program in a single software build is compiled against the same set of common definitions to make their public interfaces compatible with each other. The definitions themselves are versioned and may be changed between builds. If that happens, changes into programs may also be needed to make them compile and/or work again. Incompatible changes into system services are thus possible, but should be avoided as much as possible as it easily causes the need to maintain multiple different branches of the software.

3.5.3 Libraries

The DX 200 system supports two types of libraries: static and shared. The static libraries work the same way as they do in most operating systems. In principle, a static library is just a collection of object modules inside a single archive. These libraries are then stored in the "sack" environment where they can be accessed when some program is built. During compilation the linker collects the required code from the static libraries and places it into the resulting binary. This collecting is done on object module granularity i.e. if one routine or variable of a module is referenced by the linked code, the whole module is included into the resulting program module. This should be kept in mind when implementing services as static libraries. In order to avoid including unnecessary code in programs, every function of a static library should be in its own object module.

In contrast shared libraries in DMX work quite differently from the way they work in most generic operating systems. There is no dynamic linking involved, but instead all references are already resolved during the compilation. In practice this is implemented so that the shared program code (or data) resides in a fixed GDT segment and thus is globally visible in the system memory. The addresses of the shared symbols are exported to the programs that wish to use them via a file that is stored in the "sack". These files are included in the list of linked objects when a program is built, which allows the linker to resolve the required references.

Unlike with static library, using a shared library doesn't increase the size of the program, but an inter-segment call is required to access it. This can have a noticeable performance impact if the service is used a lot. Therefore, as usual, there is a trade-off to be made between space and time efficiency when selecting the type of library to implement.

Chapter 4

Design

The features of the target system presented in chapter 3 give a number of design decisions to consider. These decisions don't affect the fundamental principles of the selected algorithms at all, but they still have a significant impact on the actual implementation details. While using a proper algorithm already gives e.g. logarithmic computational complexity it still is important to consider how the actual single step of the computation is implemented in order to get good performance. This is especially true in an environment (like the target system of this work) where some group of similar, but slightly different operations may have an order of magnitude difference in the actual execution time or some of them may not be feasible to use at all.

This chapter first presents the general requirements derived from the target platform properties in section 4.1 and explains their rationale in sections 4.1.1 through 4.1.5. Finally the concurrency control method (or the lack thereof) used in this work is explained in section 4.2.

4.1 General Requirements

There are a number of general requirements for the design of the data structures and algorithms derived from the features of the target system.

1. The library must be usable from all programming languages supported by the platform.
2. It must be possible for an object to be a member of multiple data structures at the same time.
3. The library must be optimized for performance in the target system.
4. The memory consumption overhead must be minimized.
5. Recursion must be avoided.

The reasoning behind these requirements and the solutions for fulfilling them are presented in detail in the following sections.

4.1.1 Usability

In practice requirement 1 means that the interface definitions and data types of the public API will be written in TNSDL and a tool is used to generate the necessary definitions for the other supported programming languages. The actual algorithms themselves will be implemented using C, which is more suitable than TNSDL for this kind of data manipulation. The code will be compiled using the CAD-UL toolchain, but ultimately the selected programming language and used toolchain are internal implementation details and won't affect the users of the library in any way.

4.1.2 Object Structure

The requirement 2 means that a single object can simultaneously be a part of e.g. multiple different data structures (trees, lists, queues, etc.). In practice this will affect how to allocate the internal data structures the algorithms use for keeping track of the objects. The basic idea used in this work is that the necessary data structure(s) will be embedded inside the tracked object itself. Since it will be possible to embed multiple of these internal structures inside a single object and each of them can be used to insert the object into a different structure, the requirement is fulfilled.

Due to the way the data structures are declared, the full internal structure of these embedded data structures will be visible to the user of the library. However the users still are supposed treat them like they were fully opaque i.e. they must be only accessed by using the provided public API.

4.1.3 Performance

To fulfill requirement 3 i.e. in order to achieve good performance but still maintain generic usability, the program code of the algorithms will be implemented as a static library, but the actual links between objects will be far pointers. In practice this means that when the library is used the program code will be linked into the same segment as the code of the caller. This makes the program code bigger and results the code potentially being in the memory multiple times, but the benefit is that accessing the library won't incur the cost of a very expensive inter-segment call.

At the same time for maximum flexibility the actual accessed objects may reside anywhere in the accessible memory and all the objects in a single data structure don't even have to reside in the same segment. This is a trade-off between execution speed and flexibility. If a need arises, special single segment variants of the algorithms can be implemented later.

Also, due to main memory bandwidth not keeping up with CPU speed increases, recent studies have shown that for maximum performance the used algorithms and data structures need to be also conscious of the cache memory hierarchy [Rao & Ross, 1999; Rao & Ross,

2000; Saikkonen & Soisalon-Soininen, 2008] or carefully designed to be oblivious to it [Frigo *et al.* , 2012; Bender *et al.* , 2005; Yotov *et al.* , 2007]. However due to the generic nature of this work and the object structure specified in section 4.1.2, the library cannot affect the placement of any of the data it processes. Therefore the cache effects are not considered to be in the scope of this work.

4.1.4 Memory Usage

Requirement 4 affects a couple of design decisions. Since in the compact memory model every a pointer takes six bytes of memory (two bytes of selector + four bytes of offset) the memory usage grows quickly when complex data structures with lots of pointers are used. In order to conserve memory, only the absolutely necessary data will be stored. For this reason e.g. the tree nodes implemented in this work won't store a pointer to the parent node. The algorithms will compensate for this by keeping track of the ancestor node(s) where necessary.

Also, since the algorithms will be implemented as a shared library and the linker collects the code from such libraries on object module granularity, every routine (and possible common subroutine) will be implemented in a different source code module. This way the toolchain can put each of them in a different object module, which in turn allows the linker to collect only those routines that are actually used without getting any extraneous code at the same time. The savings on the size of the final program image size are not necessarily very big, but on the other hand it is easy to implement and it has no major downsides.

4.1.5 Recursion

Finally, requirement 5 comes from the fact the in a DX 200 system, a lot of processes have relatively small stacks, at least by today's standards. This seems to be especially true of the programs implemented back when the system memory was a much more limited resource than it is in the current hardware. To complicate the matter even further, the stack sizes of the processes are statically defined during program compilation. This means that the stack space won't expand dynamically on demand unlike it does in a lot of contemporary operating systems.

Therefore, in order to not impose any restrictions on the users of the algorithms and to avoid causing unexpected stack overruns, it is generally recommended that the algorithms are implemented non-recursively where possible. This generally minimizes the stack usage and even more importantly makes the usage constant i.e. the worst case requires no more stack space than the best case does. It is important to know especially the worst case stack usage in order to make it easier to determine how much stack a process needs and allow a conservative amount of stack to be allocated. Otherwise potentially large amounts of stack would need to be allocated just to be sure that the process won't run out of it if the worst case happens to be encountered.

To satisfy this requirement every search, insertion and deletion algorithm implemented

in this work will do only an iterative top-down pass without recursion. The only exception to this is the tree traversal which is implemented with recursion and needs to be used with care.

4.2 Concurrency

The algorithms implemented during this work expect any locking (if required) to be handled outside the library e.g. by acquiring a lock before calling the routines. In practice this means that a single global lock will cover the whole tree. With types of trees where searching doesn't alter the tree, a readers-writer lock can be used to allow concurrent reading, but for making modifications the whole tree needs to be exclusively locked.

The advantage of this kind of design is that it is easy to implement correctly, keeps the library interface simple and won't impose any unnecessary overhead in cases where no locking is needed. One alternative would be to include the locking inside the library functions, which would make the API more inflexible or require multiple variants of the routines for different types of locking. Another alternative would be to pass the locking primitives as arguments to the library via function pointers, but this would make the API more complex to use and might impose restrictions on supported programming languages.

In some use-cases a more fine-grained locking scheme could be desirable. In first stage this could mean e.g. having a separate lock inside each tree node and the routines internally locking and unlocking them as the tree is manipulated. Even though this won't be implemented in the scope of this work, the "single top-down pass" design of the algorithms (other than splay tree) is perfectly suited for adding something like that later. The splay tree is a special case since with it every operation including searching causes the root of the tree to be modified and thus cannot be as simply converted into per-node locking schemes.

In case event better concurrency is required, there has been a lot of research done on achieving it by relaxing the balancing requirements temporarily and fixing the resulting imbalance in the background or during subsequent operations like e.g. [Hanke *et al.* , 1997], [Malmi, 1997], [Larsen, 2000] and [Afek *et al.* , 2012]. However they are not in the scope of this work and are left for future improvements.

4.3 Implemented Operations

There exists a lot of different operations that can be applied to a tree, the obvious ones being *insertion* of a key, *deletion* of a key and *searching* for an exact match for a key. Those three can be considered to be the minimal set of operations that still is useful and together with depth-first tree traversal are also the only ones that will be implemented in this work. If the need arises, other operations – like e.g. finding the smallest or largest key or finding the key that immediately precedes or succeeds the given key – can easily be added later for the applicable data structures.

To put this into perspective the API implemented in this work needs to be compared with

Table 4.1: Index Operations.

	This	Java	Python	C#	GLib	C++	Boost
insert	✓	✓	✓	✓	✓	✓	✓
delete	✓	✓	✓	✓	✓	✓	✓
search	✓	✓	✓	✓	✓	✓	✓
iterate	✓	✓	✓	✓	✓	✓	✓
first		✓		✓		✓	✓
last		✓		✓		✓	✓
lower		✓					
higher		✓				✓	✓
floor		✓					
ceiling		✓				✓	✓
headset	!	✓					
subset	!	✓					
tailset	!	✓					

other similar APIs provided by other systems. In order to visualize the differences Table 4.1 shows some examples of possible operations that are applicable to trees or other index-like structures and describes which ones are supported by various implementations. Most of the systems listed in the table support multiple different index-like container classes that may each support different operations, so the entry in the table tries to aggregate those together and present the commonly available operations. This is further complicated by the fact that while some of the listed systems don't seem to support some of the operations, they often have very powerful iterator and filter facilities that can be combined to implement the missing operations very compactly (although sometimes with non-obvious efficiency) thus alleviating the need for having an explicit API for it. Therefore the table tries to list those operations only that are explicitly available as a part of the core functionality or can be implemented in an obvious and unambiguous way, like e.g. with single added function call.

All in all, it is not completely unambiguous what criteria should be used to determine some of the more borderline cases and consequently the table also cannot be considered to represent the only truth in the issue. Especially it should not be used to blindly compare the systems to each other, but it still can be used to illustrate what the lowest common

denominator of different implementations is. As can be easily seen, while the implementation that will be done in this work is indeed the minimal one, it still is totally usable and there are other implementations available for different systems that don't provide any more functionality. The actual algorithms of each operation that will be implemented in this work are described in the next chapter.

If needed, all the listed operations can be later implemented also in the library implemented in this work, except the subset functions. Since this implementation expects the object itself to provide the required space for the metadata the node requires (section 4.1.2), the object cannot be a member of arbitrary number of trees simultaneously. Therefore it is not possible to provide a generic routine that would construct a new tree that contained a subset of the nodes of the original unless the semantics of the operations are modified so that the nodes are at the same time deleted from the original tree.

Chapter 5

Implementation

Since the scope of this work concerns only in-memory data structures and not e.g. disk storage, the choice or implemented data structures was limited to those most suited for in-memory use. Namely three different types of binary trees were implemented: a binary search tree (BST), a left leaning red-black tree (RB tree) and a splay tree.

The implemented algorithms were carefully either designed or adapted to comply with all the design requirements – especially the avoiding of recursion – imposed upon the used algorithms by the target system as presented in chapter 4.1. As explained, the possibly very limited stack space available to the programs in the target environment makes the usage of iterative algorithm implementations much less restricting than using their recursive counterparts. In practice this meant making significant changes or using substantial resources to get some of the algorithms to comply with the requirements while others required no changes whatsoever and could be adapted from the literature almost as-is.

This chapter presents the detailed descriptions of the selected data structures and algorithms and explains the design changes that were required compared to the original algorithms commonly found in the literature. First the pure unbalanced BST is described in section 5.1. Then the left leaning variant of the red-black tree is described in section 5.2 and finally the splay tree implementation is described in section 5.3.

5.1 Binary Search Tree

The pure BST algorithms don't do anything special to keep the tree balanced. In practice, given a semi-random access pattern, the resulting tree is neither perfectly balanced nor a linked list. In sections 5.2 and 5.3 two different balancing algorithms are presented for improving the balance and even guaranteeing the logarithmic execution time. The effect these algorithms have is measured under different conditions and the results are presented in chapter 6.

The BST algorithms implemented in this work were designed from scratch to comply with the constraints presented in chapter 4.1. Due to the very simple nature of the data structure

and the associated algorithms basically the only significant design guideline was to make sure that every implemented operation can be done without recursion (chapter 4.1.5), which can be easily accomplished in a very straightforward and obvious way. The resulting basic algorithms for manipulating a plain BST are described in the following sections.

5.1.1 Searching

Searching in a BST is shown in Algorithm 4. The idea is to roughly halve the search space with each comparison. If the tree is perfectly balanced, this algorithm will run in $\log N$ time and in the worst case when every inner node of the tree has only one child (the tree is actually a list) it will take linear time. This same search algorithm is used also by the red-black tree (see section 5.2.2).

Algorithm 4 Searching in a BST

```
1: while node ≠ nul do
2:     if key = node.key then
3:         return node
4:     else if key < node.key then
5:         node ← node.left
6:     else
7:         node ← node.right
8:     end if
9: end while
10: return nul
```

5.1.2 Insertion

In order to insert a node into a BST, the tree is first searched for the new key. If the key is found, the new node cannot be inserted into the tree (since every node must have a unique key). Therefore if the key is already found in the tree an error is returned. If the key is not found, the new node is inserted into the leaf node where the search ended. The full algorithm is shown in Algorithm 5.

5.1.3 Deletion

Deletion from a BST is done by first searching for the matching key again. If it is found and it is not a leaf node, it is exchanged with either the greatest key of its left subtree or the smallest key of its right subtree. After the swap the matching node is now a leaf node and it can be deleted. The full algorithm is shown in Algorithm 6.

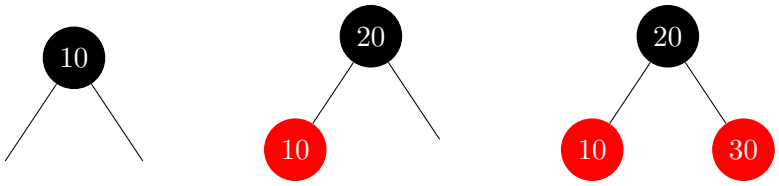
Algorithm 5 Insertion into a BST

```
1: loop
2:     if key = node.key then
3:         return error
4:     else if key < node.key then
5:         if node.left ≠ nul then
6:             node ← node.left
7:         else
8:             node.left ← newnode
9:             return success
10:        end if
11:    else
12:        if node.right ≠ nul then
13:            node ← node.right
14:        else
15:            node.right ← node
16:            return success
17:        end if
18:    end if
19: end loop
```

Algorithm 6 Deletion from a BST

```
1: node ← root
2: while node ≠ nul do
3:     if key = node.key then
4:         if node.right ≠ nul then
5:             exchange node with the smallest node of node.right
6:         else if node.left ≠ nul then
7:             exchange node with the greatest node of node.left
8:         end if
9:         delete node
10:        return success
11:    else if key < node.key then
12:        node ← node.left
13:    else
14:        node ← node.right
15:    end if
16: end while
17: return error
```

Figure 5.1: Red-Black Tree Node Types



(a) 2-node (b) 3-node (c) 4-node

5.2 Red-Black Tree

The red-black tree is a balanced BST variant that guarantees logarithmic execution time for insertion, deletion and searching. The basic idea of a RB tree is to represent a specific type of B-tree, namely a 2-3-4-tree (see section 2.6.1), as a binary search tree with two different color attributes. A lone black node represents a 2-node while red nodes are used to represent 3-nodes and 4-nodes (Figure 5.1). The left leaning red-black tree further requires that the red link on a 3-node always points to the left. This restriction is introduced to reduce the number of symmetric cases.

Since the red-black tree is basically a representation of a B-tree and B-trees have all leaf nodes at equal depth, it can be easily seen that in a red-black tree the maximum height of the left subtree is twice as much as the height of the right subtree. This invariant holds also for every subtree of the tree.

The RB tree algorithms implemented in this work were inspired by Sedgewick's lecture notes [Sedgewick, 2008], but since the algorithms presented by him were both recursive and partially incomplete the final algorithms were in principle developed from scratch. The recursive algorithms were re-written into iterative form in order to comply with the design requirements and the missing parts were developed specifically for this work. Very little apart from some general ideas remains of the original algorithms, so this implementation can fairly be considered to be a new original design. The resulting RB tree algorithms are described in detail in the following sections.

5.2.1 Tree Operations

The RB tree operations are equivalent with 2-3-4-tree operations. The splitting and joining of nodes is done by flipping the red up or down the tree (Figure 5.2) and borrowing from a sibling node is done by rotating the red links as necessary (Figure 5.3).

While in practical implementation the color is usually stored in the node, it can also be thought as the color of the link connecting two nodes together. When a link is rotated, its color doesn't change. However if the implementation stores the color in the node, it has to swap the colors of the nodes.

Figure 5.2: Red-Black Tree Color Flip

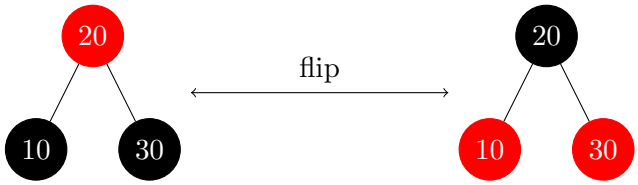
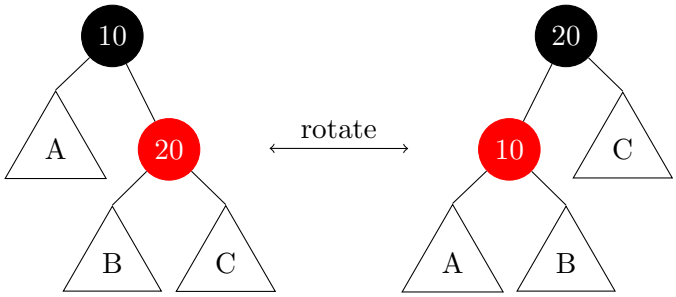


Figure 5.3: Red-Black Tree Rotations



5.2.2 Searching

Searching in a RB tree is done the same way as in a BST (see Algorithm 4 on page 32). Since unlike a BST, a RB tree has a guaranteed balance, the algorithm is also guaranteed to run in $\log N$ execution time.

5.2.3 Insertion

The basic principle of insertion is the same as with BST. The difference is that RB tree will maintain balance even after the insertion is done. The balance is maintained if the node at the bottom of the tree where the insertion is done is not a 4-node. This is accomplished by splitting any 4-nodes on the way down from the root.

In his lecture notes, Sedgewick demonstrates a recursive algorithm for insertion [Sedgewick, 2008]. In this work that algorithm was adapted to work without recursion. This makes the algorithm a bit more complex to implement, but the added complexity is not that big and is still well worth the benefits – especially considering the features of target system. The full algorithm is presented in Algorithm 7.

5.2.4 Deletion

The deletion also works in principle the same way as with BST. In the first step the deleted node is exchanged with a node at the bottom of the tree and then deleted from the tree. In order to maintain the balance, the bottom node must not be a 2-node. This is accomplished by performing one of the two operations on every 2-node on the way down.

Algorithm 7 Non-recursive insertion into a RB tree

```
1: if root.left.color = red and root.right.color = red then
2:     root.left.color ← black
3:     root.right.color ← black
4: end if
5: newnode.color ← red
6: node ← root
7: loop
8:     if key = node.key then
9:         return error
10:    else if key < node.key then
11:        if node.left = nul then
12:            node.left ← newnode {node inserted, rebalance if needed}
13:            if node.color = red then
14:                rotate node right
15:            end if
16:            return success
17:        else
18:            next ← node.left
19:        end if
20:    else
21:        if node.right = nul then
22:            node.right ← newnode {node inserted, rebalance if needed}
23:            if node.left.color = black then
24:                rotate newnode left
25:            end if
26:            if node.color = red then
27:                rotate node right
28:            end if
29:            return success
30:        else
31:            next ← node.right
32:        end if
33:    end if{split any 4-node on the way down}
34:    if next.left.color = red and next.right.color = black then
35:        colorflip next
36:        if node.color = red then
37:            if key < node.key then
38:                rotate node right
39:            else if node.left.color = black then
40:                rotate next left
41:                rotate node right
42:            end if
43:        else if key > node.key and node.left.color = black then
44:            rotate next left
45:        end if
46:    end if
47:    node ← next
48: end loop
```

1. Merge siblings.
2. Borrow from sibling.

In practice merging is done by flipping the red down and borrowing by rotating the red links. In his lecture notes, Sedgewick presents the basic outline of a recursive deletion algorithm [Sedgewick, 2008]. When the algorithm is refactored to get rid of the recursion, the details of the algorithm get quite a bit more complicated, but the operations themselves stay the same. Instead of fixing right leaning reds on the way up, like a recursive algorithm does, they must be fixed already on the way down. The algorithm must check if it is about to leave a right leaning red link behind and rotate it to correct the situation if necessary. The full algorithm is presented in Algorithm 8.

5.3 Splay Tree

A splay tree is a self balancing binary tree that has amortized logarithmic performance. Unlike e.g. RB tree, the splay tree is rebalanced also on every search (as opposed to rebalancing only on insertion and deletion). This means that also searching for all the keys in the tree in sorted order will cause the tree to degenerate into a linked list. There are ways to avoid the worst case by e.g. doing random rebalancing (searching for a random key) when needed. These algorithms are not in the scope of this work i.e. only the basic algorithm is implemented and evaluated.

The heart of the splay tree algorithms is the splay operation that is used by all the common tree operations. The splay algorithm used in this work is a direct adaptation of the top-down splay algorithm originally presented by Sleator and Tarjan [Sleator & Tarjan, 1985]. It is already based on single iterative top-down pass and therefore fulfills the design requirements presented in chapter 4.1 without any modifications. The final splay algorithm together with the tree operations using it are presented in detail in the following sections.

5.3.1 Splaying

Every operation on a splay tree is based on a common rebalancing operation called splaying. Splaying is based on a similar rotation operation as RB tree has. Since in splay tree, the nodes don't have colors, the actual criteria for rotations to perform are different. Splaying the tree with key X performs steps of rotations on the tree so that the node which has key X becomes the root of the tree (or would become if it existed in the tree). The rotations are done in such a fashion that the balance of the tree is also improved while the tree is manipulated.

There are three different splay steps: zig step (Figure 5.4), zig-zig step (Figure 5.5) and zig-zag step (Figure 5.6). The selected step depends on the place of the node in the tree. The zig step is only used when the parent of the node is the root of the tree, zig-zig when the link from the grandparent to the parent and from the parent to the node point to the

Algorithm 8 Non-recursive deletion from a RB tree

```
1: node ← root
2: loop
3:     if key = node.key then
4:         match ← node
5:         key ← key + ϵ
6:     end if
7:     if key < node.key then
8:         if node.left.color = black and node.left.left.color = black then
9:             colorflip node
10:            if node.right.left.color = red then
11:                rotate node.right.left right
12:                rotate node.right left
13:                colorflip node
14:                if node.right.right.color = red then
15:                    rotate node.right.right left
16:                end if
17:            end if
18:        end if
19:        if leaving right leaning red behind then
20:            rotate it left
21:        end if
22:        next ← node.left
23:    else
24:        if node.left.color = red and node.right.color = black then
25:            rotate node.left right
26:        end if
27:        if node.right = black and node.right.left.color = black then
28:            colorflip node
29:            if node.left.left.color = red then
30:                rotate node.left right
31:                colorflip node
32:            end if
33:        end if
34:        if leaving right leaning red behind then
35:            rotate it left
36:        end if
37:        next ← node.right
38:    end if
39:    if next = nul then
40:        if match ≠ nul then
41:            exchange match with node
42:            delete node
43:            return success
44:        else
45:            return error
46:        end if
47:    end if
48:    node ← next
49: end loop
```

Figure 5.4: Zig Step

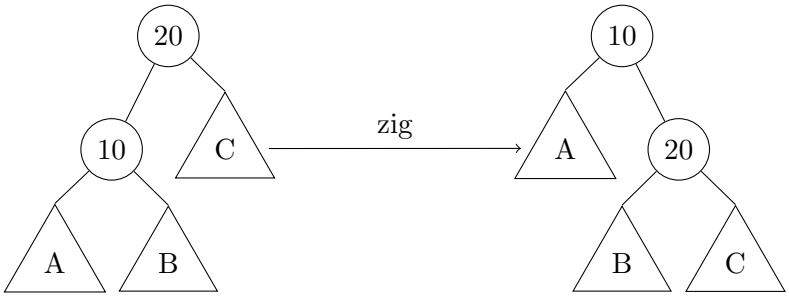
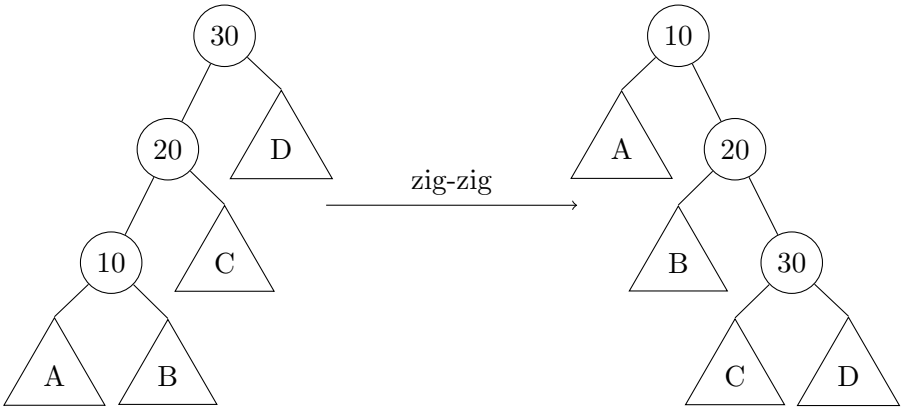


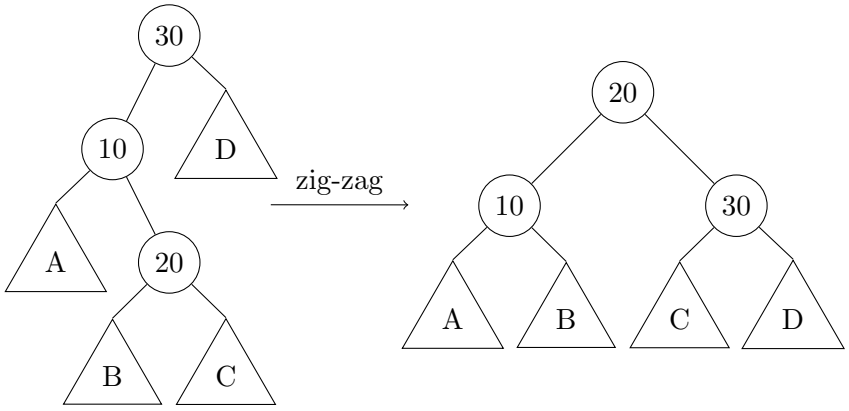
Figure 5.5: Zig-Zig Step



same direction (i.e. both are left children or both are right children) and the zig-zag step in other cases.

The actual implemented algorithm in this work is a top-down splay and is based on the

Figure 5.6: Zig-Zag Step



top-down splaying algorithm presented in the original paper by Sleator and Tarjan [Sleator & Tarjan, 1985]. The full algorithm is shown in Algorithm 9.

Algorithm 9 Top-down splay

```
1:  node ← root
2:  loop
3:      if key = node.key then
4:          break the loop
5:      else if key < node.key then
6:          if key < node.left.key then
7:              rotate node right
8:          end if
9:          if node.left = nul then
10:             break the loop
11:         end if
12:         link node right
13:     else
14:         if key > node.right.key then
15:             rotate node left
16:         end if
17:         if node.right = nul then
18:             break the loop
19:         end if
20:         link node left
21:     end if
22: end loop
23: assemble the tree
```

5.3.2 Searching

As every splay tree is also a fully qualified BST, it would be possible to use the same searching algorithm as BST (see Algorithm 5). However that would mean that it would also be possible to continuously hit the worst case, which wouldn't allow the algorithm to meet the amortized $\log N$ execution time guarantee [Weiss, 1993]. This could be done e.g. by first creating a degenerate tree in sequential order and then constantly searching for the only leaf node over and over. Every one of these searches would then have linear ($O(N)$) execution time and thus M consecutive searches would have MN execution time, which breaks the amortized time guarantee.

To make it impossible for these bad sequences to exist, the searching is instead implemented so that it also rebalances the tree while the key is being searched. This is achieved by splaying the tree with the searched key. After the splaying is done, if the key exists in the tree, it has been moved to the root of the tree and the algorithm returns it. The complete search algorithm is shown in Algorithm 10.

Algorithm 10 Searching in a splay tree

```
1: splay the tree with key
2: if key = root.key then
3:     return root
4: else
5:     return nul
6: end if
```

5.3.3 Insertion

Insertion is implemented using the same splay operation as searching is. First the tree is splayed with the key to be inserted in to the tree. If the key is already in the tree, the splaying will move it to the root of the tree. Although splay tree can be adapted to allow duplicate keys, the version implemented in this work doesn't support it, so an error is returned if the key is already found in the tree. The new key is then inserted at the root and the left and right subtrees of the old tree are attached to the new root as its children as appropriate. The complete algorithm is show in Algorithm 11.

Algorithm 11 Insertion in a splay tree

```
1: splay the tree with key
2: if key was found then
3:     return error
4: else if key < root.key then
5:     node.left ← root.left
6:     root.left ← nul
7:     node.right ← root
8: else
9:     node.right ← root.right
10:    root.right ← nul
11:    node.left ← root
12: end if
13: root ← node
14: return success
```

5.3.4 Deletion

Like all splay tree operations, deletion also begins with splaying the tree with the key to be deleted. This will move the corresponding node into the root of the tree. If the new root of the tree doesn't match the key, it means that the key doesn't exist in the tree and an error is returned. If it matches, the root of the tree is deleted. If the deleted node had no left subtree, the right child becomes the new root. Otherwise the left child becomes the new root, the left subtree is splayed so that it points to the left and finally the original

right subtree is attached as the right subtree of the new root. The full deletion algorithm is shown in Algorithm 12.

Algorithm 12 Deletion from a splay tree

```
1: splay the tree with key
2: if key was not found then
3:     return error
4: else if root.left = nul then
5:     root ← root.right
6: else
7:     root ← root.left
8:     splay the tree with key
9:     root.right ← node.right
10: end if
11: return success
```

Chapter 6

Evaluation

After the design and implementation of the selected algorithms, they were evaluated using a number of test cases. The cases were selected so that it would be reasonable to expect them to demonstrate the relative pros and cons of each of the algorithms. Each of the selected tests was run with every implemented algorithm (if applicable) to be able to evaluate the relative performance of the algorithms. Furthermore, due to the unique nature of the splay tree a specific test was run only for it. This test was designed specifically to test the re-balancing ability of the algorithm.

This chapter first describes the test setup and explains how the performance of each algorithm was measured in section 6.1. Then the methods used for generating the test sequences are depicted in section 6.2. The naive array implementation that was used as a reference is outlined in section 6.3 and finally the actual test cases and their results are reported in section 6.4.

6.1 Test Setup

As the core of the implemented work is a static library, it naturally doesn't have any kind of human interface. The DX 200 system has two main text based human interfaces, the MMI and the service terminal [Silander, 1999]. Both of these interfaces support "extensions" i.e. programs that add additional functionality to the interface. The MMI is normally used by the operator to configure and monitor the system, while the service terminal is more of a low-level debugging interface used mainly by the R&D for software development and problem analysis purposes. The service terminal is not supposed to be normally used by the operator, but it may be used in order to resolve certain fault situations. It allows much more detailed control over the low-level operation of the system, which may sometimes be necessary.

In order to be able to run tests on the library API and to provide human interface for it, a dedicated service terminal extension was implemented. The extension program is linked with the implemented library and uses its public API to execute the operations on the

data structures. It also includes additional code for e.g. verifying the integrity of the data structures and the necessary instrumentation for collecting measurement data during the tests. It allows running the selected test with chosen parameters, collects the measurements and presents the results in a suitable format for further analysis.

6.1.1 Hardware

All the actual tests were run on a CP710-A plug-in unit, which is a NSN proprietary computer unit for DX 200 in M98 mechanics [Kilpeläinen, 2003]. It is based on a Mobile Intel Pentium 3 CPU running at 800MHz. However, since the measurements are not compared against any external references, but only against other measurements made on the same hardware, the exact specification of the hardware or the absolute values measured are not very important. The real point of the measurements is to compare the results of different algorithms against each other to see what their relative performances are.

6.1.2 Measurements

All execution times were measured using the run-time tracking facility of the DMX operating system. The clock DMX uses for its run-time counters is derived from the internal TSC of the CPU [Intel, 2009b]. It is incremented on every clock cycle of the CPU, but in order to maintain compatibility with older hardware (where the TSC did not exist, but an external timer chip was used instead), the result is scaled by the operating system so that from the external observer's point of view the clock appears to tick at 1.25MHz regardless of the actual clock frequency of the CPU. This gives the run-time measurement a resolution of 0.8μs. Theoretically, if the execution was done in very short bursts, this could be too coarse and thus skew the results. However, since the computer unit was otherwise practically idle during the tests and the test itself continuously consumed as much CPU time as the OS was able to give it (almost 100%), the resolution was not an issue.

The other thing measured in selected tests was the height of the tree. It was calculated by simply traversing through the whole tree and recording the height of both left and right subtrees. As the height of a tree is the maximum of the heights of its subtrees plus one, the total height of the tree could be calculated from the results of the tree traversal.

6.2 Test Sequences

Pseudorandom numbers for all of the tests were provided by a very simple linear congruential generator [Lehmer, 1951]. Its randomness is not particularly good, but just for generating test sequences, it was sufficient. The repeatability of results, speed of execution and ease of implementation were much more important factors than perfect randomness. Also, too good randomness may also distort the results. E.g. randomness of the test set can cause a BST to achieve better balance than real-life data would.

Some of the tests also required different permutations of the test data. They were gener-

Table 6.1: The Computational Complexity of Array Operations

Find Free Slot	Insert	Delete	Search
$O(1)$	$O(1)$	$O(N)$	$O(N)$

ated with Fisher-Yates shuffle algorithm [Durstenfeld, 1964; Knuth, 1997b]. The randomness for the shuffle was provided by the aforementioned pseudorandom number generator, so the same quality applies also to it. Ultimately the quality of the shuffle was also more than sufficient for the task.

6.3 Array

In order to get a point of comparison and an idea how much of a difference a good algorithm really makes, a simple array based data storage was also implemented for reference. This kind of storage is often used as the first implementation when some concept is tested with small amount of data, mostly due to the very straight-forward implementation it has (the basic array manipulation is provided by the programming language). Unfortunately, in the absence of generic re-usable data structures, very often the array seems to be kept also as the final data structure. This remains true even when the amount of data eventually grows, which can lead to all kind of performance problems. It is very useful to see how it compares with more advanced algorithms and it nicely demonstrates the point.

In the most simple array based storage implementations, even the free elements of the array are not kept in any kind of efficient structure but are just searched from the array when needed. In this implementation it is assumed that the free elements are kept in e.g. a list (a LIFO stack) and can be found in constant time if available. This is similar to how the tree tests work and thus won't give any of the tested algorithms any unfair advantage or handicap. Since the data structure in question is just an unsorted array, the insertion after the free element has been found just consists of writing the data there, so there is no additional steps or overheads involved.

In the deletion case it was expected that first the node to be deleted needs to be found (i.e. only the key is known in advance, not the actual node). Since the data structure is just an unsorted array, the search is done by a linear scan through the array until a matching key is found or the whole array has been scanned. If the matching key was found and the operation was deletion, the element in question was returned to the list of free elements.

The complexity of all the different operations on the implemented array storage is summarized in Table 6.1.

6.4 Tests

This section describes the actual test cases, their outcomes and analysis of the observed numerical results during the testing. The measurements are presented as graphs for easy visual comparison between algorithms. In some cases multiple measurements tracked each other so closely they are hard to distinguish from each other in the graphs even when using logarithmic scale. In those cases the positions of each measurement will be explained in the text.

6.4.1 Stress Test

After the implementation was finished, each implemented algorithm was subjected to a rigorous testing to make sure there were no implementation bugs left in the code. One major part of this testing was a stress tester that performed a series of insertions and deletions with random keys selected from a random sized key space. After each insertion/deletion the complete tree was validated to make sure that all the invariants the algorithm in question is supposed to hold were still true. After a number of rounds the tree was re-initialized, a new key space was randomly selected and the test restarted.

This test was run without interruption for multiple days for each of the implemented algorithms. All of them passed the stress testing without any detected errors.

6.4.2 Tree Height with Random Keys

In this test the height of the tree was measured after a number of pseudorandom keys were inserted into the tree. The size of the keyspace was increased exponentially after each measurement. Since the test includes a random component, it was repeated multiple times with different pseudorandom sequence of keys each time and the average was also calculated.

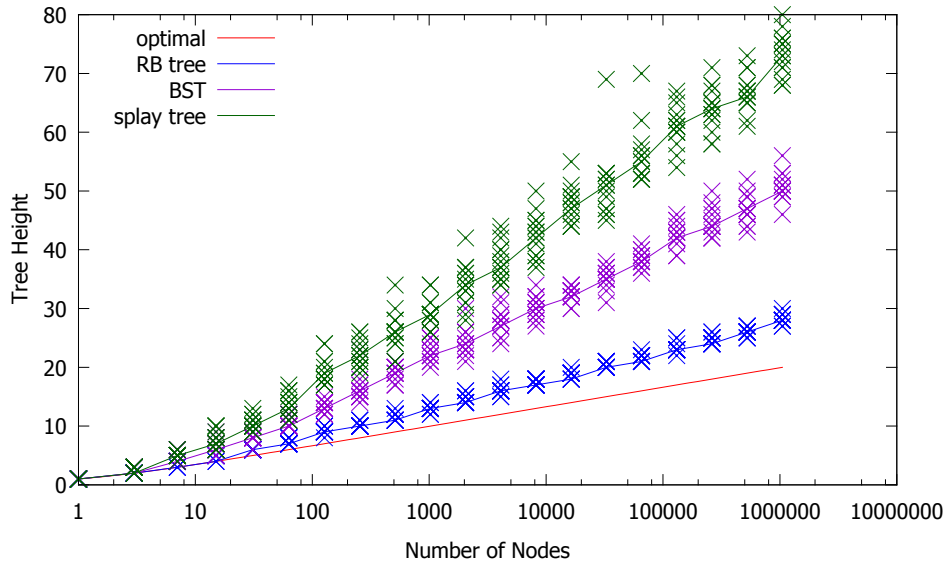
Like established in chapter 2.5.1, the maximum amount of nodes N a binary tree with height h can have is $N = 2^h - 1$. From this follows that the optimum height of a perfectly balanced tree is $h = \lceil log_2(N + 1) \rceil$. This gives a theoretical optimal limit that cannot be exceeded and is also shown in the graph for reference. The results of the test are shown in Figure 6.1.

It can be seen in the results that the height of every evaluated tree grows linearly as the keyspace grows exponentially i.e. the height of the tree grows logarithmically as a function of the tree size. There are only minor differences on the rate of the growth between the algorithms.

As expected, the algorithm with the guaranteed balance (RB tree) results in the lowest tree height and also has the lowest variance. The results are very consistent and the final height of the tree is around half of the next best algorithm (BST). The height is actually even very close to the optimal tree.

Since the insertion was done in a pseudorandom order even the algorithm which doesn't do any balancing (BST) performs very well. Its height and variance are somewhat larger than with RB tree, but at the same time better than with splay tree. From the tree height

Figure 6.1: Tree Height (Random Keys)



point of view even BST is very much usable if the key sequences are known to be random enough.

The height of a splay tree is the worst of the three and it also has the highest variance. However its results are still very decent considering that its true advantages lie elsewhere.

6.4.3 Tree Height with Sequential Keys

In this test the height of the tree was again measured after a number of keys were inserted into the tree but this time the insertion was done in a sequential order. Like in section 6.4.2, the size of the keyspace was again increased exponentially after each measurement. Since the key sequence in the test is fixed, the results are the same on every run. Therefore the test was only run once for each algorithm type. The results are shown in Figure 6.2.

The result looks like there is only one graph. This is caused by both BST and splay tree growing linearly and thus warping the scale of the graph so that it is impossible to distinguish the graphs of the RB tree and the optimal case from the X-axis. In order to be able to see the difference the results are shown again using logarithmic scale in Figure 6.3.

In this figure the RB tree and optimal case can also be seen as their own graph although they still track each other so closely that they both look like the same graph. In order to further see any difference between the two the graphs with linear growth are removed from the Figure 6.4.

The results clearly show how the RB tree still has a logarithmic height and it very closely tracks the optimum height. In fact it tracks so closely that due to the scale of the graph it basically still cannot be distinguished from the optimum result at all, which is an excellent result. The RB tree obviously handles even sequential sequences very efficiently.

Figure 6.2: Tree Height (Sequential Keys)

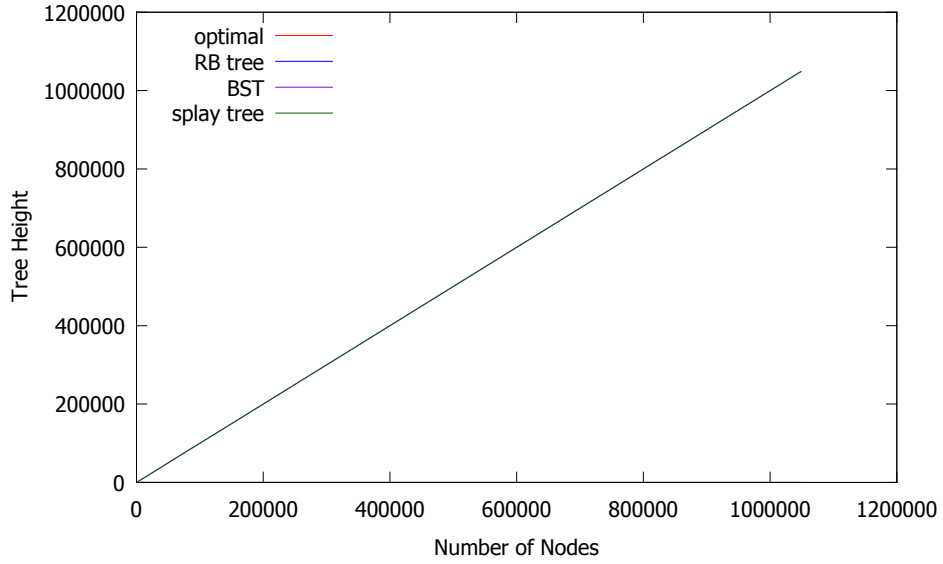


Figure 6.3: Tree Height (Sequential Keys, Logarithmic)

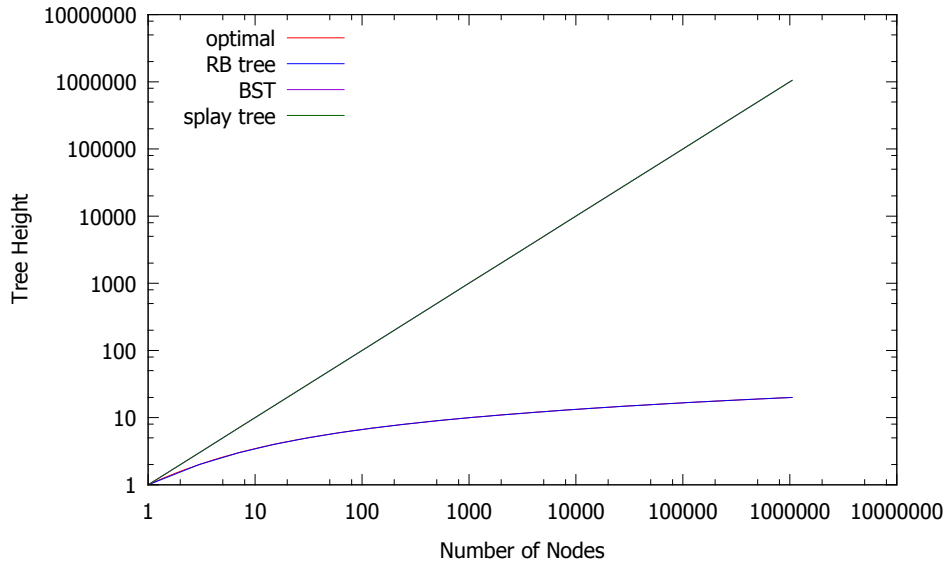
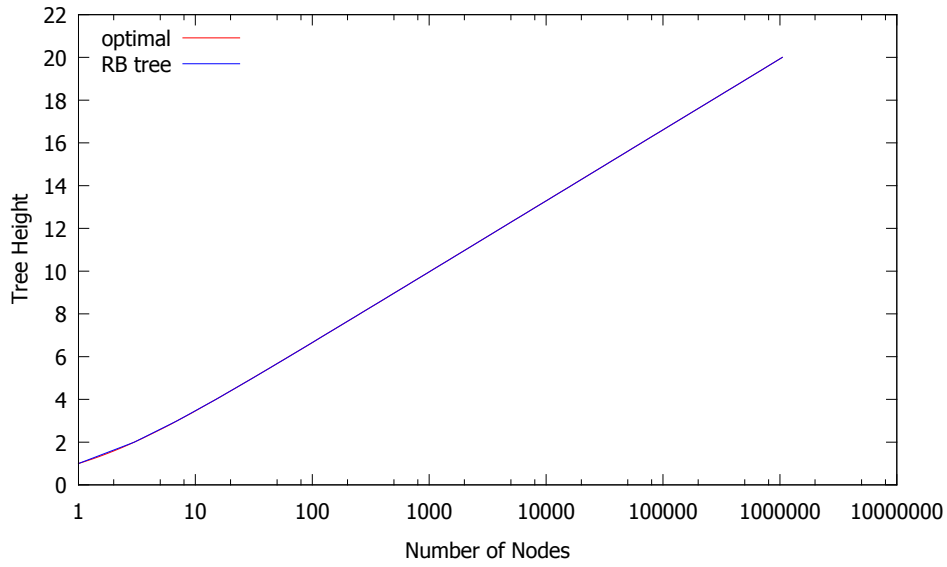


Figure 6.4: Red-Black Tree Height (Sequential Keys)



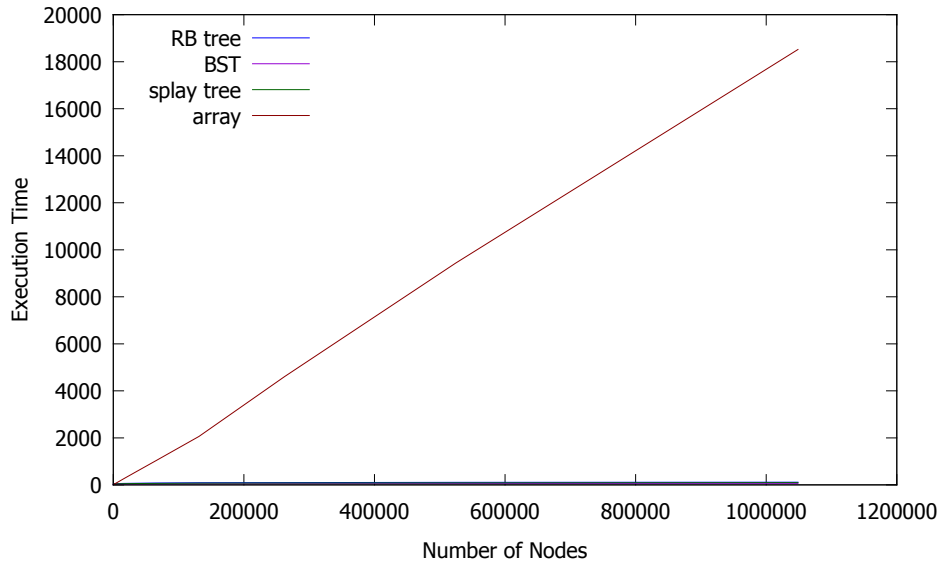
However this is the worst case scenario of both BST and splay tree. They both clearly show linear growth of the tree height as a function of the tree size. This is as expected and happens because in both cases the whole tree degenerates into a list i.e. every node of the "tree" has only one child. While in practice this means that the first search will have the same execution time, the difference is that after the first search the splay tree will have started to balance itself while the BST still remains completely degenerate.

Also, while the results with regards to the resulting tree height are the same with both BST and splay tree, the actual test run is very different indeed. The BST test takes ages to run while the splay test is actually very quick. This happens because in the BST the new node is always inserted as a leaf node while in the splay tree the new node becomes the new root of the tree. Essentially the splay tree is constructed "upside down" compared to the BST. The execution time of inserting a key in sequential order into a splay tree is $O(1)$ while inserting the same key into a BST is $O(N)$. Constructing a tree with N keys requires only $N-1$ comparisons with splay tree, but $\frac{1}{2}(N^2-N)$ comparisons with BST. This combined with the fact that the splay tree begins to balance itself as soon as non-sequential access is done (as is demonstrated in section 6.4.6) makes its overall performance much better than the performance of the BST in this test.

6.4.4 Random Insertion/Deletion

In this test the keys were pseudorandomly inserted to and deleted from the tree. The key selection function was such that the more of the keys were currently in the tree, the more likely it was to delete a key and vice versa. This makes the amount of nodes in the tree mostly fluctuate around the half of the size of the total keyspace.

Figure 6.5: Random Insertion/Deletion Speed



The amount of executed operations was the same regardless of the size of the keyspace and the size of the keyspace was increased after every measurement. Since in this test the actual execution time was measured (instead of tree height), the simple array implementation was also compared to the tree algorithms. The results of the test are shown in Figure 6.5.

The results show very clearly why a simple array is a bad idea when the amount of data is anything more than a handful of elements. Even when insertion is implemented in $O(1)$ time the cost of finding the correct element to remove makes the performance in the test so poor that when shown in the same figure, the graphs of the tree algorithms look like they are tracking the X-axis. The zoomed-in part of the results where the size of the keyspace is still small is shown in Figure 6.6.

The array doesn't need to do inter-segment access and the implementation itself is very simple. This gives it an edge when the size of the keyspace is small, but like the results show, it doesn't help very far. The execution time still grows linearly while the growth with tree algorithms is logarithmic. When the size of the keyspace reaches a couple of thousand the array is already the slowest of the tested algorithms. And keeping in mind that insertion to the array has $O(1)$ execution time but searching has $O(N)$ time and in most real-life situations searches clearly dominate over insertions and deletions, the array is really not suitable for any significant amounts of data.

To get a better idea how the tree algorithms compare to each other, the results of the test are shown again in Figure 6.7 with the array results removed.

The results clearly show the logarithmic growth every tested tree algorithm has. While the height of the tree is smallest with RB tree, in this test it gets the worst result of the three. This happens because in order to achieve the very good height it needs to do much more work per every insertion/deletion to maintain the balance of the tree. In this implementation

Figure 6.6: Random Insertion/Deletion Speed (Small Keyspace)

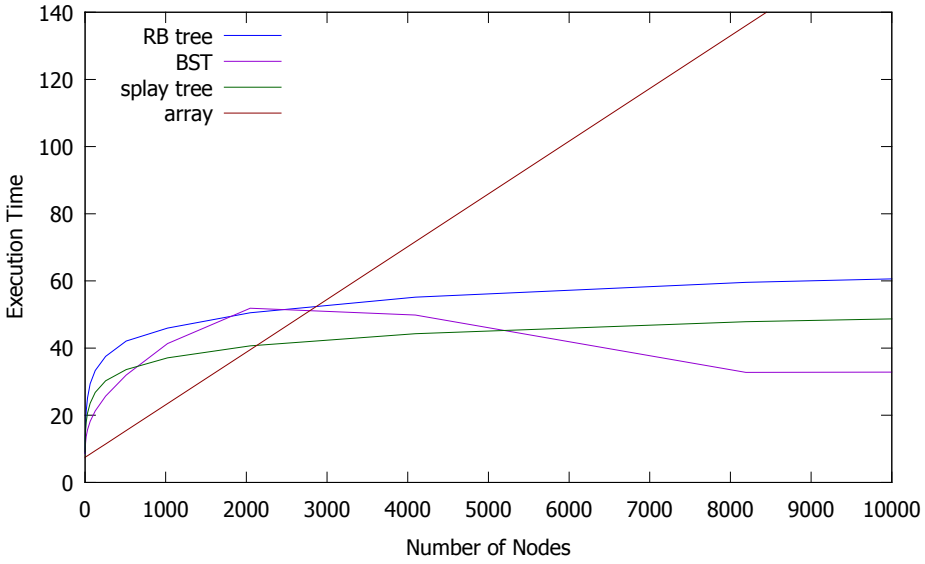


Figure 6.7: Random Insertion/Deletion Speed (Only Trees)

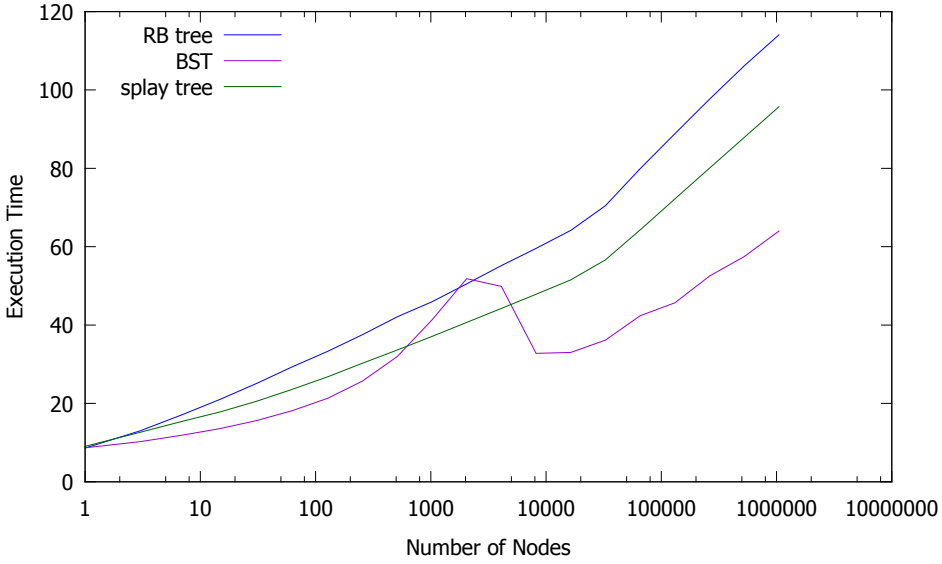
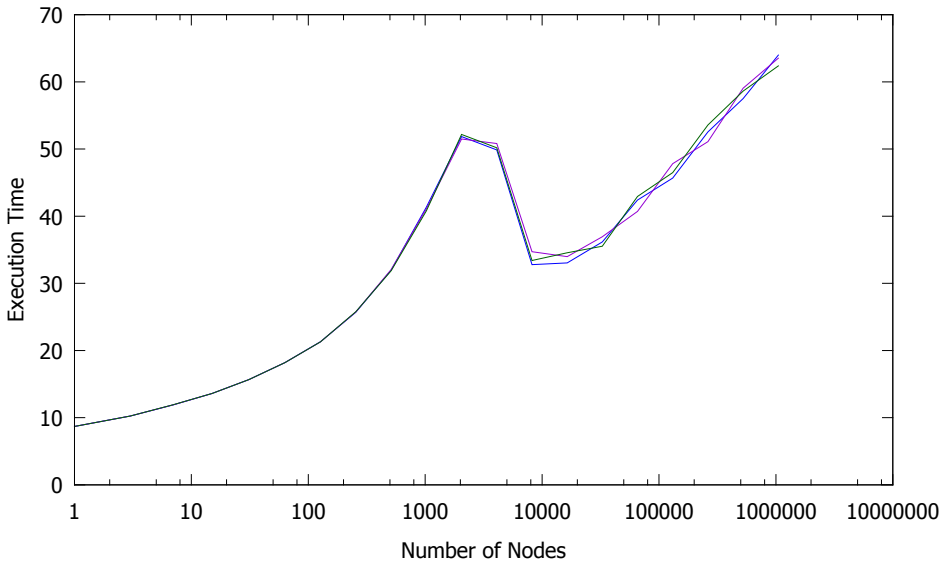


Figure 6.8: Random Insertion/Deletion Speed (BST)



these operations furthermore include inter-segment accesses that are known to be slow. The result still is very decent, but for purely insertion/deletion dominated workloads the RB tree may not be the optimal choice. However in mixed workloads the low height still gives it an edge.

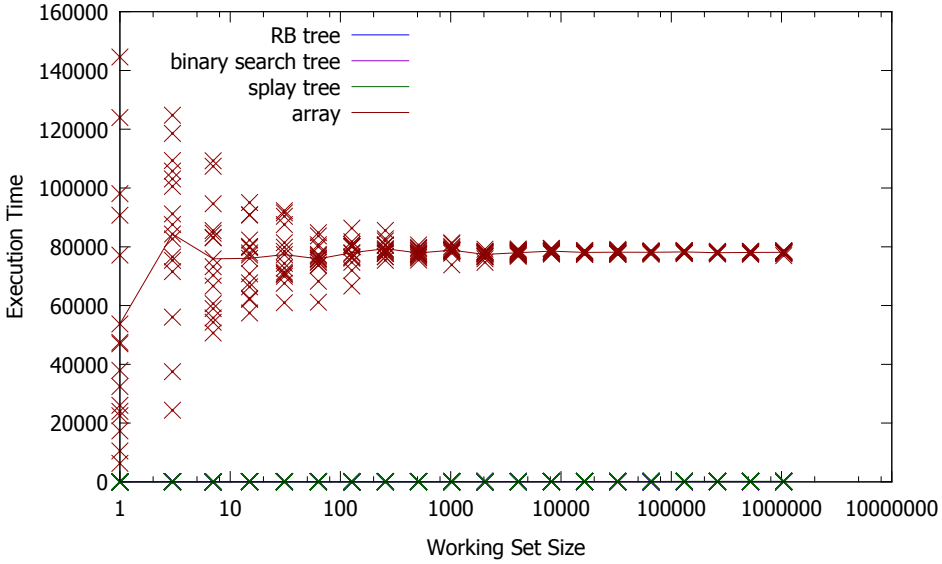
The splay tree gets a little bit better results in this test than the RB tree. The resulting tree is taller, but the simpler balancing operations give faster execution time. The overall shape of the graph is very similar to the RB tree with only the angle of growth a bit lower. All in all, the splay tree is a very good choice for insertion/deletion heavy workloads.

The BST is the fastest of the three, as expected. It doesn't do any balancing, so both insertion and deletion are very fast as long as the tree height doesn't increase too much. Since in this test the order of the key operations is random the tree height of BST remains competitive to the other two. The BST is the fastest choice for insertion/deletion heavy workloads, but only if it is known that the order of operations will be random enough.

The graph of the BST shows a peculiar bump around a couple of thousand elements. While it might be comfortable to suspect it to be a measuring error, it really isn't. After the bump was observed visually, the test was repeated multiple times with different random seeds and the results were very consistently the same. A few of the results are shown in Figure 6.8 and as can be seen all the test runs exhibit the same bump at the same number of nodes.

At certain point the execution time really gets smaller as the keyspace gets larger. The best explanation so far is that somehow the BST code interacts badly with the simple PRNG that is used to generate the test sequences, but it has not been thoroughly investigated yet.

Figure 6.9: Search Time



6.4.5 Searching

In this test a fixed size tree was first constructed using a pseudorandom key sequence. After the tree was constructed, a pseudorandom subset of the keyspace was selected. Those keys were then searched from the tree in a pseudorandom order. The size of the subset was increased after each measurement. This test is supposed to demonstrate the performance of the algorithms with regard to the locality of the data access.

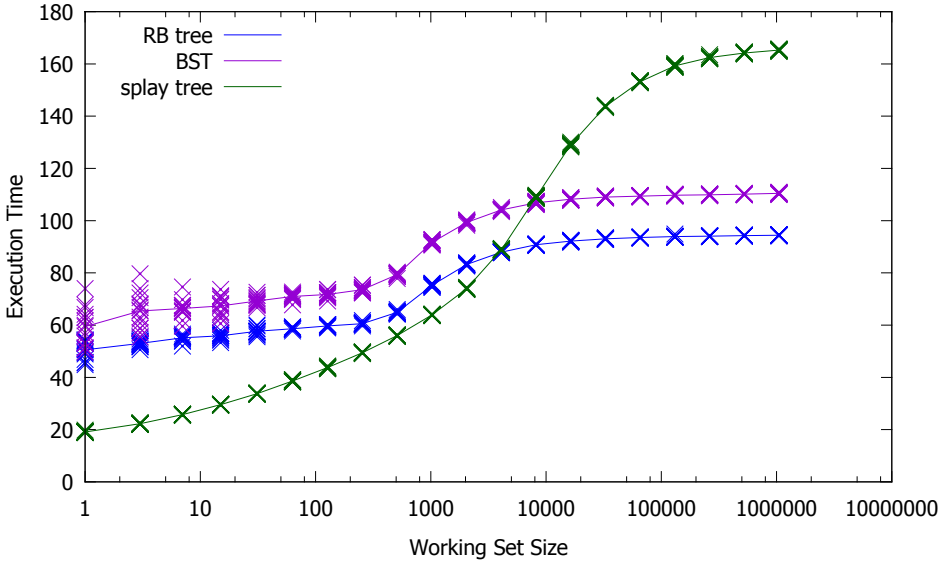
As the test has pseudorandom components, it was repeated multiple times and the average was also calculated. And since the actual execution time is measured in the test, the array implementation was also included for reference. The results of the test are shown in Figure 6.9.

The results of the array show a huge variance when the subset is small. This is obvious, since if the selected subset happens to be in the end of the array the execution time will be much higher than when the subset happens to be in the beginning of the array. As the subset grows larger the variance becomes smaller and the samples converge on the average result which itself is only determined by the total amount of data and remains almost constant regardless of the subset size.

The results of the tree algorithms in comparison are so much better that in the figure they cannot be distinguished from the X-axis. In order to compare them, the results are shown again in Figure 6.10 with the array results removed.

The results clearly show the advantage that splay tree has when the subset is small (i.e. the locality of the access is high). This is as expected since the splay tree keeps the recently accessed nodes always near the root of the tree. This directly improves the total execution time and furthermore makes it very consistent i.e. lowers the variance. When the subset

Figure 6.10: Search Time (Only Trees)



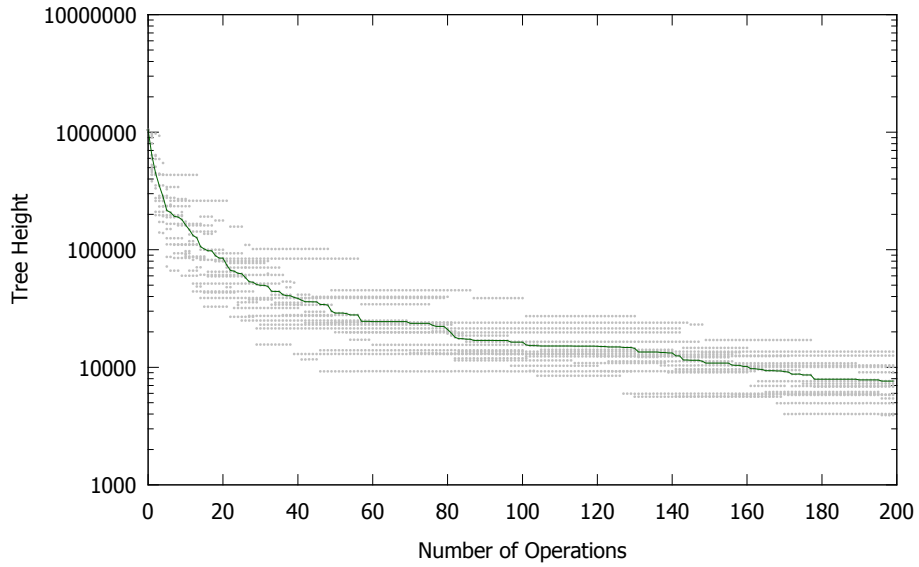
size grows enough, the splay tree loses its advantage and starts to suffer from its worse tree height. The splay tree still is the best choice of the three when the locality of the access is high.

The RB tree again performs very admirably. Its execution time has very low variance regardless of the size of the subset. Furthermore since it has no special preference for any of the keys and searching a key doesn't make any changes to the tree, the actual execution time is almost constant. Since the RB tree has the lowest tree height, its worst case searching performance is also the best among the three.

The performance of the BST very closely tracks that of the RB tree. This happens because the tree was generated using a pseudorandom key sequence and therefore the balance of the BST is quite good even while it doesn't do anything to achieve it. In this situation it only has a slightly higher variance – especially when the subset is small – and due to greater tree height also a bit higher overall execution time. The execution time of the BST also remains almost constant when the subset size increases just like with the RB tree.

While the graphs of both the RB tree and the BST are almost constant, there is an interesting bump in both around the same subset size. This is most likely caused by the size of active data set exceeding the L1 cache of the CPU. This causes the execution time to start rising as part of the data needs to fetched from slower caches. Once the working set grows so large that a significant portion of it doesn't fit into the L1 cache anymore, the graph levels again. Presumably there would be a similar, but higher, bump at the set size where the amount of active data exceeds the L2 cache, but since the caches of modern CPUs are so large, that limit is not reached in this test.

Figure 6.11: Splay Tree Self-balancing



6.4.6 Splay Tree Self-balancing

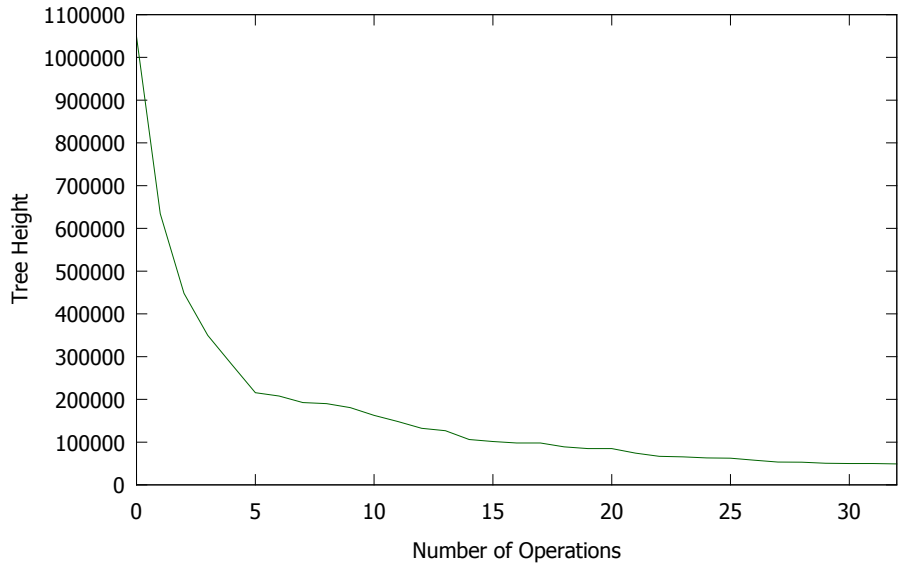
In this test a fixed size splay tree was first constructed with sequential keys. This generated a degenerate tree where every inner node had only one child and thus the height of the tree was the same as the number of nodes in the tree. After the tree was constructed, a sequence of pseudorandom keys was searched in the tree. As in splay tree searches also balance the tree, this caused tree to balance itself which reduced the height of the tree. The height of the tree was measured after every search. The results of the test are shown in Figure 6.11.

The results show that the height of the tree drops very quickly in the beginning, but it also slows down quite fast. Also the variance is quite high i.e. the rebalancing depends quite heavily on the actual key sequence. Since the tree only has $2^{20} - 1$ keys, the optimal height of the tree would be around 20. After the first 200 operations the height still is around 10000, which is nowhere near the optimal height, but the height is still getting smaller.

The Figure 6.12 shows the first few operations in more detail. It can be seen that by the 5th operation the height of the tree has been reduced to about 1/5th of the original height. By the 15th-20th operation the tree height is around 1/10th of the original, but the speed the height reduces has slowed to almost standstill. It is, however, important to remember that these numbers represent the height of the complete tree and the maximum depth is found on a subtree that hasn't been accessed. The algorithm cannot balance any branch of the tree that is not accessed, but as soon as it is, the splay operation will reduce its height too.

So in summary, the splay tree clearly re-balances itself when it is accessed in non-sequential order. This happens very fast at first, but also slows down relatively quickly. The result of this rebalancing depends heavily on the actual sequence of the accessed keys so that the

Figure 6.12: Splay Tree Self-balancing (First Operations)



commonly accessed parts of the tree get rebalanced faster. And finally all this happens while the tree still keeps the recently accessed node near the root i.e. the access that has high locality remains very fast.

Chapter 7

Conclusions

In this work three different types of binary trees were presented (in chapter 2.5). The basic tree algorithms (insertion, deletion, searching) for each of these trees were also described (in chapter 5). The actual implementations of the algorithms were carefully designed to take into account the idiosyncrasies of the target platform (presented in chapter 3), like e.g. small stacks and two different pointer classes.

The designed algorithms were implemented as a static library on a DX 200 computing platform and their performance was evaluated against both each other and a simple array based data storage (in chapter 6). The results of the evaluation were very consistently similar to the expectations and distinctly showed the need to use proper algorithms when dealing with anything more than a handful of data.

7.1 Results

An array was found unsuitable for storing any significant amount of data, which was to be expected. It should therefore only be used when the amount of data is known to be small enough or if the data can (at least in the performance sensitive paths) be directly addressed using the index. If the performance sensitive operations include searching, an array is not the recommended choice.

The BST performed very nicely when the insertion and deletion sequences were suitable for it. The insertion and deletion were very fast and the correctness of the algorithms was also easy to verify due to their simplicity. For insertion/deletion heavy workloads where the key sequences are known to be random enough the plain BST might be a viable choice. The risk is that the worst case performance of BST is very poor. If it is possible that the workload might contain bad sequences, a different tree type should be used.

The RB tree performed very well in every test, making it the proper choice for generic use. It had no significant weaknesses and in a lot of tests its performance was close to the optimal result. The only issue is that the required balancing on insertion and deletion is quite complex and thus slows down the operation. On searches the lowest height of the tree

makes RB tree the best performer in the generic cases.

The splay tree had a good performance in every test except one, where it exhibited linear behavior. Even in this worst case scenario the insertion performance was very good and it was shown that the tree will automatically rebalance itself as soon as some non-sequential accesses are done. In addition, it was demonstrated that splay tree has a measurable advantage in searching speed when only a small subset of the total tree is accessed. This makes splay trees very suitable for implementing e.g. caches.

All the tree implementations suffered some performance loss compared to the array due to the design choice to used far pointers to link the nodes of the trees. This gives them maximum flexibility, but unfortunately also makes them quite a lot slower. For optimum performance in situations where it is feasible, also near pointer versions of the library functions should be available.

7.2 Additional Work Already Done

The implemented library was also augmented with similar implementations of lists and queues with fully symmetrical sets of operations. Lists and queues are not really very interesting from the evaluation point of view since their execution time depends solely on the amount of elements (i.e. not at all on the contents of the elements themselves). Therefore their performance can easily be deduced even without measuring it (see Table 7.1). However they still are the basic building block of more complex algorithms and data structures and there is no reason everybody should always implement their own. It is much better to have one set of fully verified and easy to use implementations available as a library.

7.3 Future Work

There still are a number of things that can be done to improve the implemented library and further research to be performed. The following sections describe some ideas for further work.

7.3.1 Red-Black Tree Balancing Optimization

There probably is still a lot of room for optimizing especially the red-black tree implementation. The current implementation, for the sake of implementation simplicity (avoiding too great lookahead) under some circumstances, may makes rotations in the tree that aren't strictly necessary. A better algorithm could possibly avoid these.

7.3.2 Standard Red-Black Tree

The implementation done in this work is based on the left leaning red-black trees that were intended to simplify the implementation by avoiding symmetrical cases. It certainly does avoid the symmetry, but intuitively it also at the same time may require more rotations to

Table 7.1: The Computational Complexity of List Operations.

operation	first	last	next	prev	insert head	insert tail
slist	O(1)	O(N)	O(1)	O(N)	O(1)	O(N)
dlist	O(1)	O(N)	O(1)	O(1)	O(1)	O(N)
queue	O(1)	O(1)	O(1)	O(N)	O(1)	O(1)
deque	O(1)	O(1)	O(1)	O(1)	O(1)	O(1)
operation	insert before	insert after	remove head	remove tail	remove	concat
slist	O(N)	O(1)	O(1)	O(N)	O(N)	O(N)
dlist	O(1)	O(1)	O(1)	O(N)	O(1)	O(N)
queue	O(N)	O(1)	O(1)	O(1)	O(N)	O(1)
deque	O(1)	O(1)	O(1)	O(1)	O(1)	O(1)

maintain the tree. Also while the recursive two-pass algorithm for the left leaning tree is very simple and clean, the single top-down pass version implemented in this work already is much more complex reducing the advantage. This should be studied further.

At the same time there might be pure programming tricks that can be done to get rid of most of the duplicated code due to the symmetry. This would make the advantages of the left leaning variant smaller and might make the normal red-black tree actually a better data structure.

7.3.3 Near vs. Far Pointers

The effect the inter-segment access has on the performance of the trees should be studied further. If the effect is found to be large enough to warrant further action, a new set of functions should be implemented to operate on data structures where every element is guaranteed to reside in the same segment. In such a case near pointers could be used to link the objects to each other, which would save two bytes of memory per link and would also be expected to run faster. Then it would be up to the individual programmer using the library to choose between flexibility and efficiency as needed.

7.3.4 Splay Tree

While in real life scenarios it seldom is an actual problem, the linear worst case performance of a splay tree could use some further work. The ways to avoid the worst case performance discussed in chapter 2.5.4 should be studied and evaluated. It should be easy to extend the implemented testing framework for measuring the impact of any attempt to improve the

insertion and search algorithms of the splay tree. If a suitable workaround was found, it could be added to the current splay tree implementation.

7.3.5 Additional Data Structures

The implemented library was always intended to act as a central point for implementing all kinds of re-usable data structures and their respective algorithms. Whenever a need for new type of algorithms is identified, the library could be extended to implement it. This way the software developers would always know where to look when needing a suitable data structure or algorithm for their program.

Without even any specific demand e.g. a generic framework for implementing hash tables and optimized hash functions for the common cases (like e.g. ASCII strings) would be a useful addition. Also certain other trees and tree-like structures could be added to the library. E.g. T Trees [Lehman, 1986], tries [Fredkin, 1960] and binomial [Vuillemin, 1978] or Fibonacci heaps [Fredman & Tarjan, 1987] could find their uses if a generic implementation for them was available.

REFERENCES

Adelson-Velski Georgii M., & Landis Evgenii M. 1962. An algorithm for the organization of information. *Soviet Mathematics Doklady*, **146**, 1259–1263.

Afek Yehuda, Kaplan Haim, Korenfeld Boris, Morrison Adam, & Tarjan Robert Endre. 2012. CBTree: A Practical Concurrent Self-Adjusting Search Tree. *Pages 1–15 of:* Aguilera Marcos K. (ed), *DISC*. Lecture Notes in Computer Science, vol. 7611. Springer.

Aho Timo, Elomaa Tapio, & Kujala Jussi. 2008. Reducing splaying by taking advantage of working sets. *Pages 1–12 of: WEA'08: Proceedings of the 7th international conference on Experimental algorithms*. Berlin, Heidelberg: Springer-Verlag.

Albers Susanne, & Karpinski Marek. 2002. Randomized splay trees: theoretical and experimental results. *Information Processing Letters*, **81**(4), 213–221.

Basu Arkaprava, Gandhi Jayneel, Chang Jichuan, Hill Mark D., & Swift Michael M. 2013. Efficient virtual memory for big memory servers. *Pages 237–248 of:* Mendelson Avi (ed), *ISCA*. ACM.

Bayer Rudolf. 1972. Symmetric Binary B-Trees: Data structure and Maintenance Algorithms. *Acta Informatica*, **1**(4), 290–306.

Bayer Rudolf, & McCreight Edward M. 1972. Organization and Maintenance of Large Ordered Indexes. *Acta Inf.*, **1**, 173–189.

Bell Jim, & Gupta Gopal. 1993. An Evaluation of Self-adjusting Binary Search Tree Techniques. *Software Practice and Experience*, **23**, 369–382.

Bender Michael A., Fineman Jeremy T., Gilbert Seth, & Kuszmaul Bradley C. 2005. Concurrent Cache-oblivious B-trees. *Pages 228–237 of: Proceedings of the Seventeenth Annual ACM Symposium on Parallelism in Algorithms and Architectures*. SPAA '05. New York, NY, USA: ACM.

C386. 1988 (Aug.). *C-386 User's Guide*. Intel Corporation.

CAD-UL. 1998 (Feb.). *CXX386 C++ Cross Optimizing Compiler Reference Manual*. A1.01 edn. Computer Aided Design Ulm GmbH.

Comer Douglas. 1979. Ubiquitous B-Tree. *ACM Comput. Surv.*, **11**(2), 121–137.

Copeland Tom. 2005 (June). *Manage C data using the GLib collections.* IBM Corporation.

Cormen Thomas H., Stein Clifford, Rivest Ronald L., & Leiserson Charles E. 2001. *Introduction to Algorithms.* 2nd edn. McGraw-Hill Higher Education.

Diestel Reinhard. 2012. *Graph Theory, 4th Edition.* Graduate texts in mathematics, vol. 173. Springer.

Durstenfeld Richard. 1964. Algorithm 235: Random permutation. *Commun. ACM*, **7**(7), 420.

Even Shimon. 2011. *Graph Algorithms.* 2nd edn. New York, NY, USA: Cambridge University Press.

Fog Agner. 2014 (Feb.). *Instruction tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD and VIA CPUs.* Technical University of Denmark.

Fredkin Edward. 1960. Trie memory. *Commun. ACM*, **3**(9), 490–499.

Fredman Michael L., & Tarjan Robert Endre. 1987. Fibonacci heaps and their uses in improved network optimization algorithms. *J. ACM*, **34**(3), 596–615.

Frigo Matteo, Leiserson Charles E., Prokop Harald, & Ramachandran Sridhar. 2012. Cache-Oblivious Algorithms. *ACM Trans. Algorithms*, **8**(1), 4:1–4:22.

Fürer Martin. 1999. Randomized splay trees. *Pages 903–904 of: SODA '99: Proceedings of the tenth annual ACM-SIAM symposium on Discrete algorithms.* Philadelphia, PA, USA: Society for Industrial and Applied Mathematics.

Guibas Leo J., & Sedgewick Robert. 1978. A dichromatic framework for balanced trees. *Foundations of Computer Science, Annual IEEE Symposium on*, **0**, 8–21.

Hanke Sabine, Ottmann Thomas, & Soisalon-Soininen Eljas. 1997. Relaxed balanced red-black trees. *Pages 193–204 of:* Bongiovanni Giancarlo, Bovet DanielPierre, & Battista Giuseppe (eds), *Algorithms and Complexity.* Lecture Notes in Computer Science, vol. 1203. Springer Berlin Heidelberg.

Hansen Wilfred J. 1981. A Cost Model for the Internal Organization of B+-Tree Nodes. *ACM Trans. Program. Lang. Syst.*, **3**(4), 508–532.

Hellmann Doug. 2011. *The Python Standard Library by Example.* Developer's Library. Pearson Education, Inc.

HFS+. 2004 (Mar.). *HFS Plus Volume Format.* Apple Computer, Inc., Cupertino, CA, USA. Technical Note TN1150.

Hinze Ralf. 1999 (Sept.). Constructing Red-Black Trees. *Pages 89–99 of:* Okasaki Chris (ed), *Proceedings of the Workshop on Algorithmic Aspects of Advanced Programming Languages (WAAAPL' 99).* The proceedings appeared as a technical report of Columbia University, CUCS-023-99.

Intel. 2009a (Sept.). *Intel® 64 and IA-32 Architectures Optimization Reference Manual*. Intel Corporation.

Intel. 2009b (Sept.). *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A: System Programming Guide, Part 1*. Intel Corporation.

ISO. 1999. *ISO C Standard 1999*. Tech. rept. International Organization for Standards, Geneva, Switzerland. ISO/IEC 9899:1999 draft.

ISO. 2011. *ISO/IEC 9899:2011 Information technology — Programming languages — C*. Geneva, Switzerland: International Organization for Standardization.

Jacob Bruce, & Mudge Trevor. 1998. Virtual Memory in Contemporary Microprocessors. *j-IEEE-MICRO*, **18**(4), 60–75.

Josuttis Nicolai M. 2012. *The C++ Standard Library: A Tutorial and Reference*. 2nd edn. Addison-Wesley Professional.

Kilpeläinen Janne. 2003. *Implementation Specification, CP710-A*. 2 edn. Nokia Oyj.

Knuth Donald E. 1997a. *The art of computer programming, volume 1 (3rd ed.): fundemental algorithms*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.

Knuth Donald E. 1997b. *The art of computer programming, volume 2 (3rd ed.): seminumerical algorithms*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.

Krahmer Sebastian. 2005 (Sept.). *x86-64 Buffer Overflow Exploits and The Borrowed Code Chunks Exploitation Technique*.

Lameter Christoph. 2013. NUMA (Non-Uniform Memory Access): An Overview. *ACM Queue*, **11**(7), 40.

Larsen Kim S. 2000. AVL Trees with Relaxed Balance. *J. Comput. Syst. Sci.*, **61**(3), 508–522.

Lee Eric K., & Martel Charles U. 2007. When to use splay trees. *Softw. Pract. Exper.*, **37**(15), 1559–1575.

Lehman Tobin Jon. 1986. *Design and Performance Evaluation of a Main Memory Relational Database System (T Tree)*. Ph.D. thesis, The University of Wisconsin.

Lehmer D. H. 1951. Mathematical Methods in Large Scale Computing Units. *Annals Comp. Laboratory Harvard University*, **26**, 141–146.

Lindqvist Markus, Ruohtula Erkki, Kettunen Esa, & Tuominen Heikki. 1995. *The TNSDL Book*. 3 edn. Nokia Telecommunications Oy.

Malmi Lauri. 1997. *On Updating and Balancing Relaxed Balanced Search Trees in Main Memory*. Dissertation from Helsinki University of Technology: Teknillinen Korkeakoulu. Univ.

Miller Rick. 2012. *C# Collections: A Detailed Presentation.* Falls Church, VA, USA: Pulp Free Press.

Moore Gordon E. 1965. Cramming more components onto integrated circuits. *Electronics*, **38**(8).

Mäkinen Marco. 1995. *Nokia Saga.* 2 edn. Gummerus.

Naftalin Maurice, & Wadler Philip. 2006. *Java Generics and Collections.* O'Reilly Media, Inc.

NSN. 2007. *CAD-UL C Compiler for x86 Protected Mode: Volume 1, The Compiler Reference Manual.* CAD-UL AG and Nokia Siemens Networks.

One Aleph. 1996. Smashing the Stack for Fun and Profit. *Phrack*, **7**(49).

Pfaff Ben. 2004. Performance analysis of BSTs in system software. *SIGMETRICS Perform. Eval. Rev.*, **32**(1), 410–411.

PL/M. 1987 (Oct.). *PL/M Programmer's Guide.* Intel Corporation.

Rao Jun, & Ross Kenneth A. 1999. Cache Conscious Indexing for Decision-Support in Main Memory. *Pages 78–89 of:* Atkinson Malcolm P., Orlowska Maria E., Valduriez Patrick, Zdonik Stanley B., & Brodie Michael L. (eds), *VLDB'99, Proceedings of 25th International Conference on Very Large Data Bases, September 7-10, 1999, Edinburgh, Scotland, UK.* Morgan Kaufmann.

Rao Jun, & Ross Kenneth A. 2000. Making B+-Trees Cache Conscious in Main Memory. *Pages 475–486 of: In Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data.*

Riley Ryan, Jiang Xuxian, & Xu Dongyan. 2010. An Architectural Approach to Preventing Code Injection Attacks. *IEEE Trans. Dependable Sec. Comput.*, **7**(4), 351–365.

Ritchie Dennis M. 1996. The development of the C programming language. *History of programming languages-II*, 671–698.

Rodeh Ohad, Bacik Josef, & Mason Chris. 2013. BTRFS: The Linux B-Tree Filesystem. *Trans. Storage*, **9**(3), 9:1–9:32.

Roemer Ryan, Buchanan Erik, Shacham Hovav, & Savage Stefan. 2012. Return-Oriented Programming: Systems, Languages, and Applications. *ACM Trans. Inf. Syst. Secur.*, **15**(1), 2:1–2:34.

Russinovich Mark E., Solomon David A., & Ionescu Alex. 2012. *Windows Internals - Parts 1 and 2.* 6th edn. Microsoft Press.

Saikkonen Riku, & Soisalon-Soininen Eljas. 2008. Cache-sensitive Memory Layout for Binary Trees. *Pages 241–255 of:* Ausiello Giorgio, Karhumäki Juhani, Mauri Giancarlo, & Ong Luke (eds), *Fifth Ifip International Conference On Theoretical Computer Science – Tcs 2008.* IFIP International Federation for Information Processing, vol. 273. Springer Boston.

Schäling Boris. 2011. *The Boost C++ Libraries.* XML Press.

Sedgewick Robert. 1983. *Algorithms.* Addison-Wesley. Chap. 15, page 199.

Sedgewick Robert. 2008. *Left-leaning Red-Black Trees.* Tech. rept. Princeton University.

Shacham Hovav. 2007. The Geometry of Innocent Flesh on the Bone: Return-into-libc Without Function Calls (on the x86). *Pages 552–561 of: Proceedings of the 14th ACM Conference on Computer and Communications Security.* CCS '07. New York, NY, USA: ACM.

Silander Simo. 1999. *DX Primer, Introduction to DX 200 Software Engineering.* Nokia Telecommunications Oy.

Sjödin Jan, & von Platen Carl. 2001. Storage allocation for embedded processors. *Pages 15–23 of: CASES.*

Sleator Daniel Dominic, & Tarjan Robert Endre. 1985. Self-adjusting binary search trees. *J. ACM*, **32**(3), 652–686.

Soma Yuki, Gerofi Balazs, & Ishikawa Yutaka. 2014. Revisiting Virtual Memory for High Performance Computing on Manycore Architectures: A Hybrid Segmentation Kernel Approach. *Pages 3:1–3:8 of: Proceedings of the 4th International Workshop on Runtime and Operating Systems for Supercomputers.* ROSS '14. New York, NY, USA: ACM.

Štrbac Savić Svetlana, & Tomašević Milo. 2012. Comparative Performance Evaluation of the AVL and Red-black Trees. *Pages 14–19 of: Proceedings of the Fifth Balkan Conference in Informatics.* BCI '12. New York, NY, USA: ACM.

Vuillemin Jean. 1978. A data structure for manipulating priority queues. *Commun. ACM*, **21**(4), 309–315.

Walter Chip. 2005. Kryder's Law. *Scientific American*, July.

Weiss Mark Allen. 1993. *Data Structures and Algorithm Analysis in C.* The Benjamin/Cummings Publishing Company, Inc.

Yotov Kamen, Roeder Tom, Pingali Keshav, Gunnels John, & Gustavson Fred. 2007. An Experimental Comparison of Cache-oblivious and Cache-conscious Programs. *Pages 93–104 of: Proceedings of the Nineteenth Annual ACM Symposium on Parallel Algorithms and Architectures.* SPAA '07. New York, NY, USA: ACM.

Appendix A

Public API

The public header file for the library (in TNSDL).

```
/* Datatypes */

/* Singly-linked list */

TYPE slist_item_t
    REPRESENTATION STRUCT
        offs area_size_t;
        next POINTER FAR (slist_item_t);
    ENDSTRUCT;
ENDTYPE slist_item_t;

TYPE slist_head_t
    REPRESENTATION STRUCT
        first POINTER FAR (slist_item_t);
    ENDSTRUCT;
ENDTYPE slist_head_t;

/* Doubly-linked list */

TYPE dlist_item_t
    REPRESENTATION STRUCT
        offs area_size_t;
        next POINTER FAR (dlist_item_t);
        prev POINTER FAR (dlist_item_t);
    ENDSTRUCT;
ENDTYPE dlist_item_t;

TYPE dlist_head_t
    REPRESENTATION STRUCT
        first POINTER FAR (dlist_item_t);
    ENDSTRUCT;
ENDTYPE dlist_head_t;
```

```
/* Queue */

TYPE queue_item_t
    REPRESENTATION STRUCT
        offs area_size_t;
        next POINTER FAR (queue_item_t);
    ENDSTRUCT;
ENDTYPE queue_item_t;

TYPE queue_head_t
    REPRESENTATION STRUCT
        first POINTER FAR (queue_item_t);
        last  POINTER FAR (queue_item_t);
    ENDSTRUCT;
ENDTYPE queue_head_t;

/* Deque */

TYPE deque_item_t
    REPRESENTATION STRUCT
        offs area_size_t;
        next POINTER FAR (deque_item_t);
        prev POINTER FAR (deque_item_t);
    ENDSTRUCT;
ENDTYPE deque_item_t;

TYPE deque_head_t
    REPRESENTATION STRUCT
        first POINTER FAR (deque_item_t);
        last  POINTER FAR (deque_item_t);
    ENDSTRUCT;
ENDTYPE deque_head_t;

/* Binary search tree */

TYPE bst_foreach_attr_t
    CONSTANT
        in_c   = 0x0;
        pre_c  = 0x1;
        post_c = 0x2;
        mask_c = 0x3;
        reverse_c = 0x4;
    REPRESENTATION dword;
ENDTYPE bst_foreach_attr_t;

TYPE bst_item_t
    REPRESENTATION STRUCT
        offs  area_size_t;
        left  POINTER FAR (bst_item_t);
```

```
        right POINTER FAR (bst_item_t);
    ENDSTRUCT;
ENDTYPE bst_item_t;

TYPE bst_tree_t
    REPRESENTATION STRUCT
        root POINTER FAR (bst_item_t);
    ENDSTRUCT;
ENDTYPE bst_tree_t;

/* Splay tree */

TYPE splay_item_t
    REPRESENTATION STRUCT
        offs  area_size_t;
        left  POINTER FAR (splay_item_t);
        right POINTER FAR (splay_item_t);
    ENDSTRUCT;
ENDTYPE splay_item_t;

TYPE splay_tree_t
    REPRESENTATION STRUCT
        root POINTER FAR (splay_item_t);
    ENDSTRUCT;
ENDTYPE splay_tree_t;

/* Red-black tree */

TYPE rb_item_t
    REPRESENTATION STRUCT
        offs  area_size_t;
        left  POINTER FAR (rb_item_t);
        right POINTER FAR (rb_item_t);
    ENDSTRUCT;
ENDTYPE rb_item_t;

TYPE rb_tree_t
    REPRESENTATION STRUCT
        root POINTER FAR (rb_item_t);
    ENDSTRUCT;
ENDTYPE rb_tree_t;

/* The library interface */

    SERVICES SYNC
    LIBRARY dtylib;

    /* Singly-linked list */

    slist_init_head(
```

```
    IN/OUT head slist_head_t
) ->, NEAR => ;

slist_init_item(
    IN     ptr  anypointer,
    IN/OUT item slist_item_t
) ->, NEAR => ;

slist_insert_head(
    IN/OUT head slist_head_t,
    IN/OUT item slist_item_t
) ->, NEAR => ;

slist_insert_tail(
    IN/OUT head slist_head_t,
    IN/OUT item slist_item_t
) ->, NEAR => ;

slist_insert_before(
    IN/OUT head slist_head_t,
    IN/OUT old  slist_item_t,
    IN/OUT item slist_item_t
) ->, NEAR => ;

slist_insert_after(
    IN/OUT head slist_head_t,
    IN/OUT old  slist_item_t,
    IN/OUT item slist_item_t
) ->, NEAR => ;

slist_remove_head(
    IN/OUT head slist_head_t
) -> anypointer, NEAR => ;

slist_remove_tail(
    IN/OUT head slist_head_t
) -> anypointer, NEAR => ;

slist_remove(
    IN/OUT head slist_head_t,
    IN/OUT item slist_item_t
) ->, NEAR => ;

slist_first(
    IN/OUT head slist_head_t
) -> anypointer, NEAR => ;

slist_last(
    IN/OUT head slist_head_t
) -> anypointer, NEAR => ;
```

```
slist_next(
    IN/OUT item slist_item_t
) -> anypointer, NEAR => ;

slist_prev(
    IN/OUT head slist_head_t,
    IN/OUT item slist_item_t
) -> anypointer, NEAR => ;

slist_concat(
    IN/OUT head1 slist_head_t,
    IN/OUT head2 slist_head_t
) ->, NEAR => ;

slist_foreach(
    IN/OUT head slist_head_t,
    IN     func offset,
    IN     arg  anypointer
) ->, NEAR => ;

/* Doubly-linked list */

dlist_init_head(
    IN/OUT head dlist_head_t
) ->, NEAR => ;

dlist_init_item(
    IN     ptr  anypointer,
    IN/OUT item dlist_item_t
) ->, NEAR => ;

dlist_insert_head(
    IN/OUT head dlist_head_t,
    IN/OUT item dlist_item_t
) ->, NEAR => ;

dlist_insert_tail(
    IN/OUT head dlist_head_t,
    IN/OUT item dlist_item_t
) ->, NEAR => ;

dlist_insert_before(
    IN/OUT head dlist_head_t,
    IN/OUT old  dlist_item_t,
    IN/OUT item dlist_item_t
) ->, NEAR => ;

dlist_insert_after(
    IN/OUT head dlist_head_t,
```

```
    IN/OUT old  dlist_item_t,
    IN/OUT item dlist_item_t
) ->, NEAR => ;

dlist_remove_head(
    IN/OUT head dlist_head_t
) -> anypointer, NEAR => ;

dlist_remove_tail(
    IN/OUT head dlist_head_t
) -> anypointer, NEAR => ;

dlist_remove(
    IN/OUT head dlist_head_t,
    IN/OUT item dlist_item_t
) ->, NEAR => ;

dlist_first(
    IN/OUT head dlist_head_t
) -> anypointer, NEAR => ;

dlist_last(
    IN/OUT head dlist_head_t
) -> anypointer, NEAR => ;

dlist_next(
    IN/OUT item dlist_item_t
) -> anypointer, NEAR => ;

dlist_prev(
    IN/OUT item dlist_item_t
) -> anypointer, NEAR => ;

dlist_concat(
    IN/OUT head1 dlist_head_t,
    IN/OUT head2 dlist_head_t
) ->, NEAR => ;

dlist_foreach(
    IN/OUT head dlist_head_t,
    IN     func offset,
    IN     arg  anypointer
) ->, NEAR => ;

/* Queue */

queue_init_head(
    IN/OUT head queue_head_t
) ->, NEAR => ;
```

```
queue_init_item(
    IN     ptr  anypointer,
    IN/OUT item queue_item_t
) ->, NEAR => ;

queue_insert_head(
    IN/OUT head queue_head_t,
    IN/OUT item queue_item_t
) ->, NEAR => ;

queue_insert_tail(
    IN/OUT head queue_head_t,
    IN/OUT item queue_item_t
) ->, NEAR => ;

queue_insert_before(
    IN/OUT head queue_head_t,
    IN/OUT old  queue_item_t,
    IN/OUT item queue_item_t
) ->, NEAR => ;

queue_insert_after(
    IN/OUT head queue_head_t,
    IN/OUT old  queue_item_t,
    IN/OUT item queue_item_t
) ->, NEAR => ;

queue_remove_head(
    IN/OUT head queue_head_t
) -> anypointer, NEAR => ;

queue_remove_tail(
    IN/OUT head queue_head_t
) -> anypointer, NEAR => ;

queue_remove(
    IN/OUT head queue_head_t,
    IN/OUT item queue_item_t
) ->, NEAR => ;

queue_first(
    IN/OUT head queue_head_t
) -> anypointer, NEAR => ;

queue_last(
    IN/OUT head queue_head_t
) -> anypointer, NEAR => ;

queue_next(
    IN/OUT item queue_item_t
```

```
) -> anypointer, NEAR => ;

queue_prev(
    IN/OUT head queue_head_t,
    IN/OUT item queue_item_t
) -> anypointer, NEAR => ;

queue_concat(
    IN/OUT head1 queue_head_t,
    IN/OUT head2 queue_head_t
) ->, NEAR => ;

queue_foreach(
    IN/OUT head queue_head_t,
    IN     func offset,
    IN     arg  anypointer
) ->, NEAR => ;

/* Deque */

deque_init_head(
    IN/OUT head deque_head_t
) ->, NEAR => ;

deque_init_item(
    IN     ptr  anypointer,
    IN/OUT item deque_item_t
) ->, NEAR => ;

deque_insert_head(
    IN/OUT head deque_head_t,
    IN/OUT item deque_item_t
) ->, NEAR => ;

deque_insert_tail(
    IN/OUT head deque_head_t,
    IN/OUT item deque_item_t
) ->, NEAR => ;

deque_insert_before(
    IN/OUT head deque_head_t,
    IN/OUT old  deque_item_t,
    IN/OUT item deque_item_t
) ->, NEAR => ;

deque_insert_after(
    IN/OUT head deque_head_t,
    IN/OUT old  deque_item_t,
    IN/OUT item deque_item_t
) ->, NEAR => ;
```

```
deque_remove_head(
    IN/OUT head deque_head_t
) -> anypointer, NEAR => ;

deque_remove_tail(
    IN/OUT head deque_head_t
) -> anypointer, NEAR => ;

deque_remove(
    IN/OUT head deque_head_t,
    IN/OUT item deque_item_t
) ->, NEAR => ;

deque_first(
    IN/OUT head deque_head_t
) -> anypointer, NEAR => ;

deque_last(
    IN/OUT head deque_head_t
) -> anypointer, NEAR => ;

deque_next(
    IN/OUT item deque_item_t
) -> anypointer, NEAR => ;

deque_prev(
    IN/OUT item deque_item_t
) -> anypointer, NEAR => ;

deque_concat(
    IN/OUT head1 deque_head_t,
    IN/OUT head2 deque_head_t
) ->, NEAR => ;

deque_foreach(
    IN/OUT head deque_head_t,
    IN     func offset,
    IN     arg  anypointer
) ->, NEAR => ;

/* Binary search tree */

bst_init_tree(
    IN/OUT tree bst_tree_t
) ->, NEAR => ;

bst_init_item(
    IN     ptr  anypointer,
    IN/OUT item bst_item_t
```

```
) ->, NEAR => ;

bst_insert(
    IN/OUT tree    bst_tree_t,
    IN/OUT item    bst_item_t,
    IN     cmpfunc offset
) -> error_t, NEAR => ;

bst_remove(
    IN/OUT tree    bst_tree_t,
    IN/OUT item    bst_item_t,
    IN     cmpfunc offset
) -> error_t, NEAR => ;

bst_find(
    IN/OUT tree    bst_tree_t,
    IN     arg     anypointer,
    IN     cmpfunc offset
) -> anypointer, NEAR => ;

bst_foreach(
    IN/OUT tree bst_tree_t,
    IN     func offset,
    IN     arg  anypointer,
    IN     attr bst_foreach_attr_t
) ->, NEAR => ;

/* Splay tree */

splay_init_tree(
    IN/OUT tree splay_tree_t
) ->, NEAR => ;

splay_init_item(
    IN     ptr  anypointer,
    IN/OUT item splay_item_t
) ->, NEAR => ;

splay_insert(
    IN/OUT tree    splay_tree_t,
    IN/OUT item    splay_item_t,
    IN     cmpfunc offset
) -> error_t, NEAR => ;

splay_remove(
    IN/OUT tree    splay_tree_t,
    IN/OUT item    splay_item_t,
    IN     cmpfunc offset
) -> error_t, NEAR => ;
```

```
splay_find(
    IN/OUT tree    splay_tree_t,
    IN     arg     anypointer,
    IN     cmpfunc offset
) -> anypointer, NEAR => ;

splay_foreach(
    IN/OUT tree splay_tree_t,
    IN     func offset,
    IN     arg  anypointer,
    IN     attr bst_foreach_attr_t
) ->, NEAR => ;

/* Red-black tree */

rb_init_tree(
    IN/OUT tree rb_tree_t
) ->, NEAR => ;

rb_init_item(
    IN     ptr  anypointer,
    IN/OUT item rb_item_t
) ->, NEAR => ;

rb_insert(
    IN/OUT tree    rb_tree_t,
    IN/OUT item    rb_item_t,
    IN     cmpfunc offset
) -> error_t, NEAR => ;

rb_remove(
    IN/OUT tree    rb_tree_t,
    IN/OUT item    rb_item_t,
    IN     cmpfunc offset
) -> error_t, NEAR => ;

rb_find(
    IN/OUT tree    rb_tree_t,
    IN     arg     anypointer,
    IN     cmpfunc offset
) -> anypointer, NEAR => ;

rb_foreach(
    IN/OUT tree rb_tree_t,
    IN     func offset,
    IN     arg  anypointer,
    IN     attr bst_foreach_attr_t
) ->, NEAR => ;

ENDLIBRARY dtylib;
```