Under consideration for publication in J. Functional Programming

More on Balanced Diets

OLIVER FRIEDMANN
Dept. of Computer Science
University of Munich, Germany
(*e-mail:* `Oliver.Friedmann (at) gmail.com`)

MARTIN LANGE
Dept. of Electrical Engineering and Computer Science
University of Kassel, Germany
(*e-mail:* `Martin.Lange (at) uni-kassel.de`)

Abstract

Discrete Interval Encoding Trees (Diets) are data structures for the representation of fat, i.e. densely populated sets over a discrete linear order. In this paper we introduce algorithms for set-theoretic operations like intersection, union, etc. on sets represented as balanced diets. We empirically analyse their performance and show that these algorithms can outperform previously known algorithms on sets, such asthe ones implemented in OCaml's standard library.

1 Introduction

Many algorithms operate on sets of elements of a certain type. It is therefore desirable to have efficient data structures that represent sets in a programming language. There is no natural representation since – mathematically – a set is nothing more than a collection of elements with no further structure on them. Objects that represent such elements and that reside in a standard computer memory are naturally ordered though. That means that any representation of sets in a standard programming language has to introduce and use some additional structure on these elements.

The simplest examples of such representations are lists, introducing an arbitrary ordering that is not even a partial order. The price to pay is possibly multiple occurrences of elements and therefore suboptimal space consumption. Also, lookup operations have bad running times. Very quick lookup / insert / delete operations can be performed on boolean arrays as set representations. This way, the elements are totally ordered by the indices in the array. The disadvantage of this representation is the fact that best-case space consumption is as bad as the worst case. Hence, such representations are only useful for small sets, resp. sets over a small domain.

Larger sets or just sets over larger domains are usually stored as binary search trees (Cormen *et al.*, 1992; Adams, 1993). This also requires a total ordering on their elements, but this ordering is then used in a clever way to perform lookup / insert / delete operations avoiding the traversal of the entire data structure in the average case whilst keeping space consumption low as well. The low running times – logarithmic in the size of the set – can even be guaranteed in the worst-case when the search trees remain balanced. This can be achieved with some minor enhancements on the insert and delete operations and at the expense of a very minor increase in space consumption: the nodes on the trees have to carry some additional information about how balanced their subtrees are. There are various types of balanced search trees for the representation of sets around, the most prominent ones being AVL trees (Adelson-Velskii & Landis, 1962) and red-black trees (Bayer, 1972; Guibas & Sedgewick, 1978).

For certain types of sets, this does not yield a space-optimal representation. Examples include *fat sets* – the opposite of a sparse set – in which elements tend to occur in chunks, i.e. in non-trivial intervals of the underlying total order. Erwig suggested a modified data structure for the presentation of such sets: *discrete interval encoding trees*, or *diets* for short (Erwig, 1998).

Diets are binary search trees in which every node carries two elements of the underlying total order. These two elements define an interval, being the least and the greatest element of that interval. All intervals in a diet are maximal, i.e. no two intervals in it are overlapping and not even touching each other. For instance, the set $\{1,2,3,6,7,9,11,12,13\}$ can be represented by the set of maximal intervals $[1,3]$, $[6,7]$, $[9,9]$, $[11,13]$. These intervals can be stored in a binary search tree since the total odering on the set's elements extends naturally to non-overlapping intervals over this domain.

It should be clear that such a representation can be very succinct for fat sets. The double occurrence of the 9 in this representation indicates though, that this is potentially wasteful for sparse sets. The space consumed by such a representation is not predominantly determined by the size of the set but by the number of closed intervals the set can be decomposed into. This is usually much less for fat sets. On the other hand, lookup / insert / delete operations on standard binary search trees have to be modified in order to work on diets. This, however, does not impede their efficiency under reasonable assumptions about the running times of comparing operations on the underlying domain, and works as one would expect.

- Lookup operations do not compare their argument with the content of a node but check for inclusion in the represented interval by performing one or two comparisons with the interval's bounds.
- Insert operations are modified like the lookups but, additionally, have to keep the invariant about maximality of intervals intact. Hence, if an inserted element extends an existing interval on either side, then this could lead to a merging of that interval with the next, resp. preceding one.
- Delete operations are modified similarly; removing a single element, for instance, may split up an interval into two parts which may require a reordering of the tree's nodes.

Again, the performance of such operations depends on the trees being balanced. What is desirable here is a running time logarithmic in the size of the tree rather than the set. Remember that the size of the tree is the number of maximal intervals that the represented set can be decomposed into. In order to achieve this, trees need to remain balanced.

Erwig, in his introductory paper (Erwig, 1998), has not taken balancing into account. This has been taken up by Ohnishi, Tasaka, and Tamura who showed how to enhance diets by using AVL trees rather than simple binary search trees for the structuring of the intervals (Ohnishi *et al.*, 2003). Note that the insert operation on diets is slightly different from that on simple binary search trees: rebalancing may be required not only due to the insertion of a new interval but also due to the merging of two intervals, which then also entails a deletion step.

This is where the algorithmic handling of diets stops in the literature. In particular, there is no description of efficient set-theoretic (union, intersection, difference, etc.) let alone functional operations (iteration through all elements, partitioning of a set according to an arbitrary predicate, etc.) on balanced diets. Ohnishi et al. describe how to partition a set represented as a diet according to a predicate of the form “less or equal a given element”, but it is easy to see that this is very similar to a deletion operation and does not generalise to arbitrary predicates.

We remark that there are several ways to carry out such operations, not all of them are optimal. For example, there is a balanced diet implementation of sets of integers as part of the CAMOMILE library (Yoriyuki, 2003). It covers the extensive interface of the set implementation in the OCaml standard library[1], including partitioning, iteration and the like on top of the set-theoretic operations. It does not feature optimal algorithms though. There are of course other data structures which serve similar purposes whilst storing data in a different way, for example Patricia tries (Morrison, 1968; Gwehenberger, 1968) which can also be used to represent sets of data.

In this paper we describe better algorithms on balanced diets for set-theoretic and functional operations. An OCAML implementation is publicly available (Friedmann & Lange, 2010) - the code for handling the AVL trees is borrowed from the Objective Caml Standard Library Set module (Leroy, 2010).

The paper is organised as follows. Sect. 2 introduces balanced diets formally. Sect. 3 describes three binary functions on trees, namely the *union*, *intersection* and *difference* of sets and analyses their worst-case running time behaviour. In Sect. 4 we show that these algorithms presented here do indeed improve over existing and alternative ones in practice.

2 Balanced Diets

A *linear order* is a pair $(M,\leq)$ consisting of a set M and a binary relation $\leq \subseteq M \times M$ s.t. for all $x,y,z \in M$ we have

- if $x \leq y$ and $y \leq z$, then $x \leq z$, and
- if $x \leq y$ and $y \leq x$, then $x = y$, and
- $x \leq y$ or $y \leq x$.

As usual, we write $<$ to denote the strict part of $\leq$, i.e. $x < y$ iff $x \leq y$ and $x \neq y$. Also, we write $\top$ for the maximal element of M if it exists, and $\bot$ for the minimal element likewise.

A *discrete linear order* is a triple $(M,\leq,succ)$ s.t. $(M,\leq)$ is a linear order and $succ$: $M \setminus \{\top\} \to M$ is a unary function s.t. for all $x \in M \setminus \{\top\}$:

[1] `http://caml.inria.fr/pub/docs/manual-ocaml/libref/Set.html`

- $x < succ(x)$, and
- there is no y s.t. $x < y$ and $y < succ(x)$.

It is easy to see that each discrete linear order induces another function *pred* which is defined on $M \setminus \{\bot\}$ and behaves like the inverse of *succ*.

In the following we fix a discrete linear order $(M, \leq, succ)$ and introduce the entire theory w.r.t. this fixed one. We will also sometimes speak of M as a discrete linear order when in fact we mean $(M, \leq, succ)$.

An *interval* of M is a non-empty $N \subseteq M$ s.t. for all $x, y, z \in M$:

- if $x \in N$, $y \in N$, $x < z$, and $z < y$, then $z \in N$.

A *finite* interval is such an (non-empty) N that has finitely many elements only. The minimum of a finite interval N is an $x \in N$ s.t. $x \leq y$ for all $y \in N$. It is denoted $\min N$. The maximum is defined accordingly. They are unique because M is linearly ordered and always exist. Furthermore, N is uniquely determined by the pair $[\min N, \max N]$. Hence, such pairs are therefore legal representations of finite intervals. We define

$$[\![[x,y]]\!] \quad := \quad \{z \mid x \leq z \text{ and } z \leq y\}$$

Let $Ivl(M)$ denote the set of all finite intervals over M. In the following we will always assume intervals to be finite without mentioning this explicitly.[2]

Two intervals $[x_0, y_0]$ and $[x_1, y_1]$ are called *independent* if $succ(y_0) < x_1$ or $succ(y_1) < x_0$. Hence, independent intervals do not overlap, they are not even adjacent in the sense that their union is not an interval. Independent intervals are again ordered by an order $\prec$ defined as $[x_0, y_0] \prec [x_1, y_1]$ iff $y_0 < x_1$. It is not hard to see that $\preceq$, its reflexive closure, is again a linear order if restricted to a subset of pairwise independent intervals. It can be used to store independent intervals in a binary search tree.

In the following we will deal with binary trees whose nodes are labeled with intervals of M. The class of all such trees is the smallest class $\mathcal{T}_M$ s.t.

a) $\bot \in \mathcal{T}_M$ (the empty tree), and
b) if $l, r \in \mathcal{T}_M$ and $[x,y] \in Ivl(M)$ then $([x,y], l, r) \in \mathcal{T}_M$.

Note that we do not pose any restrictions on the intervals in a tree here.

Given a tree t, we call all $\bot$-labeled nodes *leaf nodes* and all other nodes *inner nodes*. The top-most node is called *root* of the tree.

For a tree t, we write $root(t)$ to denote the interval at its root, i.e. $root(t) = [x,y]$ if $t = ([x,y], l, r)$ for some $l, r \in \mathcal{T}_M$. Also, we write $nodes(t)$ for the set of all intervals occurring in t, i.e.

$$nodes(t) \quad := \quad \begin{cases} \emptyset & \text{if } t = \bot \\ \{[x,y]\} \cup nodes(l) \cup nodes(r) & \text{if } t = ([x,y], l, r) \end{cases}$$

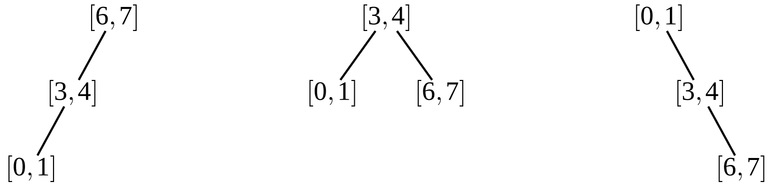
A *discrete interval encoding tree* (diet) is a binary tree that is inductively defined as follows.

[2] It is not difficult to extend everything to infinite intervals of linear order, for example by introducing $\top$ and/or $\bot$ as additional symbols and letting $[x, \top]$ denote $\{y \mid x \leq y\}$.

- $\perp$ is a diet.
- If l and r are diets and $[x,y] \in M$ s.t. $y' < pred(x)$ for all $[\_,y'] \in nodes(l)$ and $succ(y) < x'$ for all $[x',\_] \in nodes(r)$, then $([x,y],l,r)$ is also a diet.

Hence, the intervals occurring in a diet are all independent, and a node that is left of another one carries an interval that is smaller w.r.t. $\prec$.

A diet t represents a finite subset of M in a straight-forward way: $[\![t]\!] := \bigcup\{[\![[x,y]]\!] \mid [x,y] \in nodes(t)\}$. Note that, conversely, each finite subset of M has a unique decomposition into independent intervals, but not necessarily a unique diet representation since, in general, there are many ways to build a tree-structure from a set of pairwise independent intervals. For instance, the set $\{0,1,3,4,6,7\}$ can be represented by three different trees.



The *height* of a tree t is the maximal length of a path from the root to a leaf:

$$height(t) \quad := \quad \begin{cases} 0, & \text{if } t = \perp \\ 1 + \max\{height(l), height(r)\}, & \text{if } t = ([x,y],l,r) \end{cases}$$

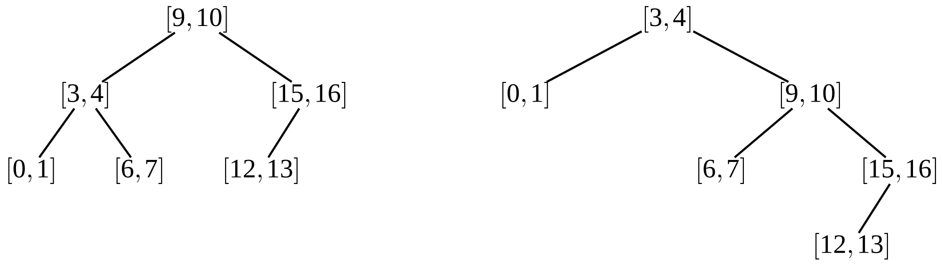
We now introduce the class of balanced diets by an inductive definition. Every leaf is a balanced diet. Furthermore, a tree $t = ([x,y],l,r)$ is *balanced* iff both l and r are balanced and $|height(l) - height(r)| \leq 1$. These kinds of height-balanced trees are also well-known as *AVL* trees (Adelson-Velskii & Landis, 1962). The height of a balanced tree with n nodes is at most $\lceil \log_\Phi n \rceil$ where $\Phi = \frac{1+\sqrt{5}}{2}$.

We say that a pair (l,r) of two balanced diets l and r is *left-right-separate* iff $succ(y) < x$ for every $[\_,y] \in nodes(l)$ and every $[x,\_] \in nodes(r)$. Given an interval $[a,b]$ and a pair (l,r) of two balanced diets l and r, we say that $[a,b]$ is a *separator of* (l,r) iff $succ(y) < a$ for every $[\_,y] \in nodes(l)$ and $succ(b) < x$ for every $[x,\_] \in nodes(r)$.

For rebalancing intermediate trees, we will apply two routines that are generally known as the *reroot of balanced trees* and the *join of balanced trees*. The *reroot operation* is a binary transformation $l \bowtie r$ defined on left-right-separate diets (l,r) and returns a new balanced diet $t = l \bowtie r$ s.t. $[\![t]\!] = [\![l]\!] \cup [\![r]\!]$. The *join operation* is a ternary transformation $l\,{}^{a}\!\bowtie^{b} r$ defined on a pair of diets (l,r) and a separator $[a,b]$, and returns a new balanced diet $t = l\,{}^{a}\!\bowtie^{b} r$ s.t. $[\![t]\!] = [\![l]\!] \cup [a,b] \cup [\![r]\!]$. It is well-known that both rebalancing operations require logarithmic time in the worst-case (Adelson-Velskii & Landis, 1962).

We will use balanced trees as a synonym for AVL trees throughout the paper. However, our approach does not rely on AVL tree balancing, any other approach for maintaining balanced trees could be applied as well.

We also consider a certain subclass of diets that we call *streamed trees*. Every leaf is a *streamed* tree. Furthermore, a tree $t = ([x,y],l,r)$ is *streamed* if l is balanced and r is streamed. Note that every balanced tree is necessarily a streamed tree. Consider the following example: the left tree is balanced (and streamed) while the right tree is only streamed.



3 Operations on Balanced Diets

First, we consider the *diet decomposition* of balanced diets that essentially allows us to access a diet iteratively as a stream in an efficient manner. Second, we briefly describe the basic reading and writing operations on balanced diets that have already been described in Erwig's paper (Erwig, 1998). Third, we consider binary methods, namely the *union*, the *intersection* and the *difference* of two sets. We claim that our methods based on diet decomposition are much more efficient in practice than the standard implementation, e.g. (Yoriyuki, 2003). Finally, we consider some other notable set routines.

3.1 Diet Decomposition

Most operations combining two balanced trees simultaneously handle related data in the trees, in the sense that processing a certain node in the first tree comes along with processing a node in the other tree containing data that are closely related by the total ordering relation. As related data are not necessarily at related positions in the tree, it is impossible to process both trees by simultaneous recursion. However, if the operation on both trees results in a new tree, it turns out to be beneficial in the average case to process one tree by recursion, since in this case the balanced structure of one of the input trees can be transferred to some extent.

We utilise a way to extract the elements represented by the tree according to their ordering without touching nodes of the tree that are not on a path to a node that is being extracted. This guarantees that – when this operation is embedded into a loop for instance – unnecessary operations on the diet are being avoided.

The idea is a well-known trick, linearising the tree in a lazy-evaluation manner: in order to extract the least element in the tree, we simply do right-rotations until finally a node with a leaf on the left-hand side comes up. Then, we return the node and its right subtree. In this manner, we only traverse the path in the tree to the least node and simultaneously rotate in such a way that extracting the next element can be either performed at the top of the tree or takes place in a region of the tree yet unvisited.

The following function *extr* takes a non-empty stream and extracts the smallest interval from it, i.e. it returns a pair consisting of this interval and a stream representing what is left-over after removal of that interval.

$$\begin{array}{rcl} \mathit{extr}(\alpha,\perp,r) & := & (\alpha,r) \\ \mathit{extr}(\alpha,(\beta,l',r'),r) & := & \mathit{extr}(\beta,l',(\alpha,r',r)) \end{array}$$

It is then possible to extract a list of the k smallest intervals in a stream t by using *extr* iteratively.

$$\left.\begin{array}{rcl} \mathit{extract}_1(t) & := & [\alpha] \\ \mathit{extract}_{k+1}(t) & := & \alpha :: \mathit{extract}_k(t') \end{array}\right\} \quad \text{if } \mathit{extr}(t) = (\alpha,t')$$

The following lemma states that this extraction is more efficient then simply transforming t into a list of intervals and returning the first k elements of this list.

Lemma 1 *Let t be a stream with n nodes and $k \leq n$. The result of $\mathit{extract}_k(t)$ can be computed in time $\mathscr{O}(\max(k,\ \log n))$.*

This is quite obvious since each path that is traversed by any call of *extr* has height $\mathscr{O}(\log n)$ and contains at most one node already visited, namely the root of the current tree. The complexity is obviously optimal since extracting one element clearly takes time $\mathscr{O}(\log n)$ in the worst-case and extracting the first k elements takes at least time $\mathscr{O}(k)$.

3.2 Basic Operations

The basic reading operations that can be performed on sets essentially comprise the *emptyness check*, the *membership check*, the *iteration* over the elements, the *folding* over the elements and the computation of the *cardinality* of the set. All these routines are straightforward and well covered in the literature on data structures.

The basic writing operations comprise the *insertion* of an interval into a balanced diet, *adding* a single element to a balanced diet – which is based on the insertion of a singleton interval – and the *removal* of a single element from a balanced diet.

More concretely, given a diet t and an interval $[a,b]$, the operation $\mathit{insert}([a,b],t)$ returns a new diet t' s.t. $[\![t']\!] = [\![[a,b]]\!] \cup [\![t]\!]$. Similarly, given a single element a instead of an interval $[a,b]$, the operation $\mathit{add}(a,t)$ returns a new diet t' s.t. $[\![t']\!] = \{a\} \cup [\![t]\!]$, and the operation $\mathit{remove}(a,t)$ returns a new diet t' s.t. $[\![t']\!] = [\![t]\!] \setminus \{a\}$. These operations are described and analysed in Erwig's introductory work (Erwig, 1998). They are presented as operations on – not necessarily balanced – diets, and it is straightforward to add rebalancing instructions into the algorithms in order to turn them into operations being performed on balanced diets (Ohnishi *et al.*, 2003). The runtime complexities of these operations are logarithmic in the worst-case.

3.3 Binary Operations

The binary operations on sets – *intersection*, *union* and *difference* of sets – allow many approaches to realize them. The intrinsic problem is that an independent recursive descent on both trees is desired but not easily possible. This is where the diet decomposition becomes useful.

3.3.1 Intersection

The intersection *inter* of two diets t and s proceeds by traversing one of the two trees, say t, from left to right entering deeper levels only if necessary while performing the intersection of the current interval of t with all appropriate intervals from the other tree.

Being based on a traversal of t, the structure of the intersection diet of t and s maintains the already balanced structure of t whenever it is possible.

On the other hand, the balanced structure of s is of no interest. The algorithm treats s as a stream of ordered intervals with restricted look-ahead knowledge, meaning that the algorithm is only interested in the currently remaining minimal interval. Therefore, the algorithm will access s only through the *extr* function.

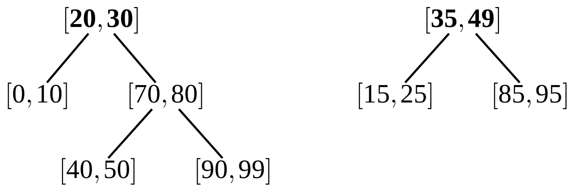
Since we will want to call *inter* recursively to compute the intersection of a tree t and what is left of s and then proceed with the result of the intersection, we are also interested in what is left of s after intersecting it with t. More precisely, $\mathit{inter}(t,s)$ will return a tuple (a,b) with $[\![a]\!] = [\![t]\!] \cap [\![s]\!]$ and $[\![b]\!] = \{i \in [\![s]\!] \mid i > j \text{ for all } j \in [\![t]\!]\}$.

```
fun $\mathit{inter}(t,s)$ =
    if $t = \bot$ or $s = \bot$ then $(\bot,\ s)$
    else let $([x,y],l,r) = t$ and $([x',y'],\_) = \mathit{extr}(s)$ in
        if $x' \geq x$ then $\mathit{interhelp}(\bot,[x,y],r,s)$
        else let $(l',s') = \mathit{inter}(l,s)$ in
            $\mathit{interhelp}(l',[x,y],r,s')$
```

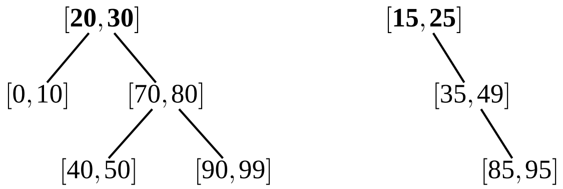
The helper function *interhelp* takes four parameters l, $[x,y]$, r and s, and computes the union of l with the intersection of $([x,y],\bot,r)$ and s, and returns the remains of s in addition. In other words, *interhelp* assumes that l is a diet left of $[x,y]$ that already has been computed as the intersection of the original trees and hence simply attaches it to the intersection of the rest that is to be computed.

```
fun $\mathit{interhelp}(l,[x,y],r,s)$ =
    if $s = \bot$ then $(l,\ \bot)$
    else let $([x',y'],u) = \mathit{extr}(s)$ in
        if $y' < x$ then
            $\mathit{interhelp}(l,[x,y],r,u)$
        else if $y < x'$ then
            let $(r',s') = \mathit{inter}(r,s)$ in
            $(l \bowtie r', s')$
        else if $y' \geq \mathit{pred}(y)$ then
            let $(r',s') = \mathit{inter}(r,s)$ in
            let $i = \max(x,x')$ and $j = \min(y,y')$ in
            $(l\ {}^{i}\!\bowtie^{j} r', s')$
        else
            let $l' = \mathit{insert}([\max(x,x'),y'],l)$ in
            $\mathit{interhelp}(l',[\mathit{succ}(y'),y],r,u)$
```

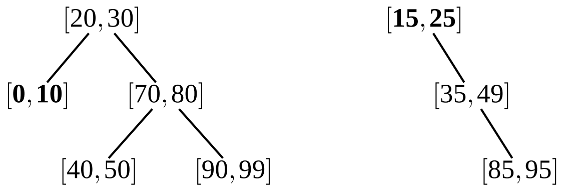
Consider the following two trees for instance. We will follow the intersection algorithm on them in an implicit way: the right tree will be used as an ordered interval stream and the left tree will be used both as input and result tree. This way, it becomes obvious how the overall structure of the left tree is more or less maintained in the construction of the result tree. From now on, we will call the right tree "stream" and refer to the left tree simply as the "tree".



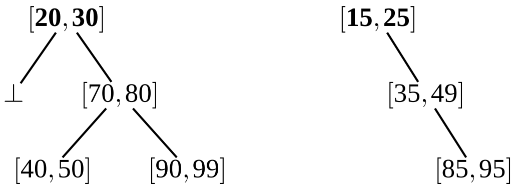
First, we need to perform a right-rotation on the stream to bring the smallest interval to the top of it.



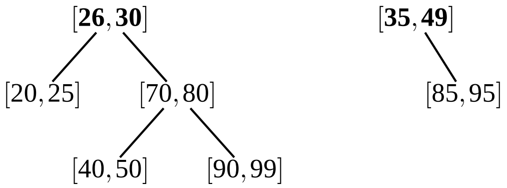
Comparing the top intervals, it could be the case that the left subtree of the tree contains an intersection with the stream, hence the algorithm descends the tree to the left.



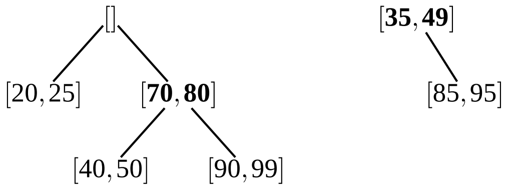
Since the current interval of the tree lies below the minimal interval of the stream, we can drop it and return to the higher level of the tree again.



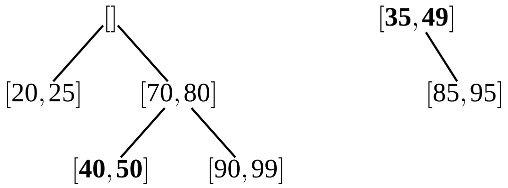
Next, the top intervals intersect and the upper bound of the stream interval is below the upper bound of the tree interval. Hence, the algorithm computes the intersection of both intervals, inserts the result into the left subtree, keeps the remainder of the tree's top interval and pops the top interval of the stream.



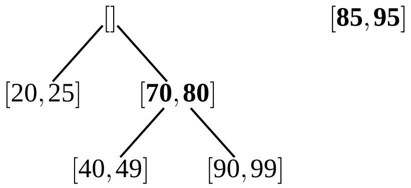
As the lower bound of the stream is above the upper bound of the top interval of the tree, it is safe to remove it and descend to the right subtree. Note that we have an empty root now which is to be fixed afterwards.



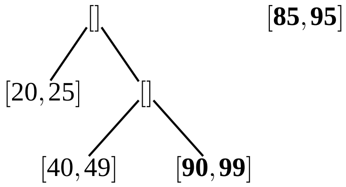
Since the upper bound of the stream's smallest interval is below the lower bound of the current interval of the tree, the algorithm descends to the left subtree.



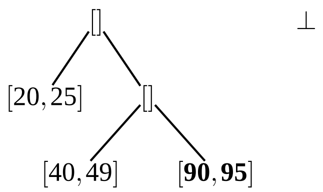
Again, the algorithm encounters an intersection of the two intervals that are currently focussed. Although the upper bound of the tree's interval is above the upper bound of the stream, we do not have to keep the remainder of the interval in this case, because it is only above the stream's bound by one and since all intervals in the original trees have to be independent, it cannot be the case that we miss an intersection by dropping the remainder. We replace the old interval in the tree by the intersection of the two intervals, given by the maximum of the lower bounds and the minimum of the upper bounds.



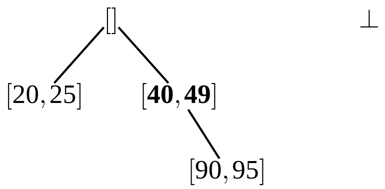
Since the lower bound of the stream is above the upper bound of the current interval, we are safe to remove it and descend to the right subtree.



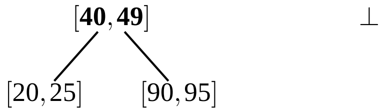
We compute the intersection of the current interval and the last interval of the stream.



Finally, we have to join all subtrees with no root. Technically, this process just happens whenever the respective recursive calls return. Hence, we first combine the two subtrees on the right.



As the very last step, we restore the root of the full tree and return it as result of the intersection of the two original trees.



3.3.2 Difference

The difference *diff* of two diets t and s is computed in a similar fashion. It proceeds by traversing the first tree t, from left to right; the other tree s, is treated as a stream of ordered intervals that will be only accessed via the *extr* function.

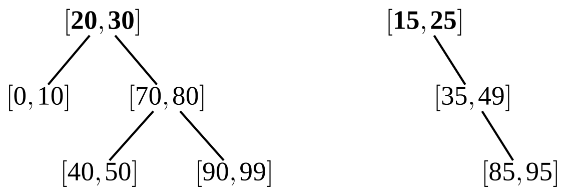
Again, we need to keep track of all parts of the stream that have not been processed in recursive calls. Therefore, *diff* will return a pair, containing the computation of the difference so far and what remains of the stream. More formally, $\mathit{diff}(t,s)$ returns a tuple (a,b) with $[\![a]\!] = [\![t]\!] \setminus [\![s]\!]$ and $[\![b]\!] = \{i \in [\![s]\!] \mid i \succ j \text{ for all } j \in [\![t]\!]\}$.

```
fun $\mathit{diff}(t,s)$ =
    if $t = \perp$ or $s = \perp$ then $(t,\ s)$
    else let $([x,y],l,r) = t$ and $([x',y'],\_) = \mathit{extr}(s)$ in
        if $x' \geq x$ then $\mathit{diffhelp}(l,[x,y],r,s)$
        else let $(l',s') = \mathit{diff}(l,s)$ in
            $\mathit{diffhelp}(l',[x,y],r,s')$
```

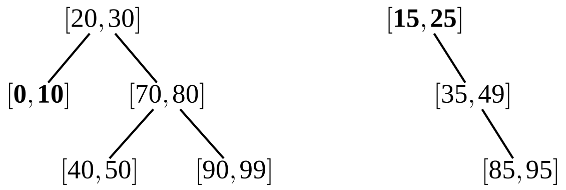
The helper function *diffhelp* takes four parameters l, $[x,y]$, r and s, and computes the union of l with the difference of $([x,y],\perp,r)$ and s, and returns the remains of s in addition. In other words, *diffhelp* assumes that l is a diet left of $[x,y]$ that already has been computed as the difference of the original trees and hence simply attaches it to the difference of the rest that is to be computed.

```
fun $diffhelp(l,[x,y],r,s)$ =
    if $s = \bot$ then $(l\,{}^{x}\!\bowtie^{y} r,\ \bot)$
    else let $([x',y'],u) = extr(s)$ in
        if $y' < x$ then
            $diffhelp(l,[x,y],r,u)$
        else if $y < x'$ then
            let $(r',s') = diff(r,s)$ in
            $(l\,{}^{x}\!\bowtie^{y} r',s')$
        else if $x < x'$ then
            let $l' = insert([x,pred(x')],l)$ in
            $diffhelp(l',[x',y],r,s)$
        else if $y' < y$ then
            $diffhelp(l,[succ(y'),y],r,u)$
        else let $(r',s') = diff(r,s)$ in
            $(l \bowtie r',s')$
```

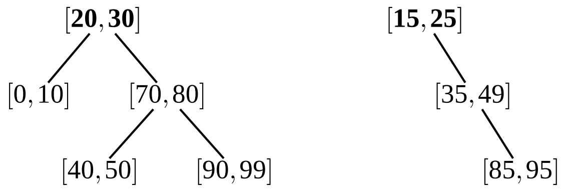
Consider the two diets used to explain the mechanism of the intersection algorithm above. We will follow the difference algorithm on them, too. The right tree will be used as an ordered interval stream and the left tree will serve both as input and result tree. First, we need to perform a right-rotation on the stream again in order to bring the smallest interval to the top of it.



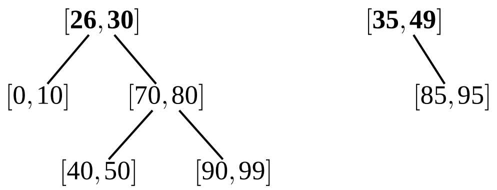
Comparing the top intervals, it could be the case that the left subtree of the tree contains an intersection with the stream, hence the algorithm descends the tree to the left.



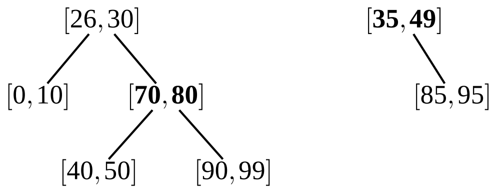
Since the current interval of the tree lies below the minimal interval of the stream, we can keep it and return to the higher level of the tree again.



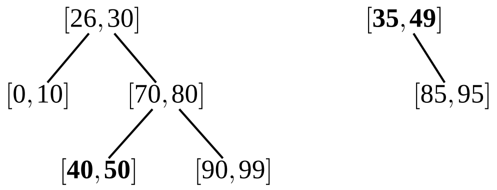
Next, the top intervals intersect and the upper bound of the stream interval is below the upper bound of the tree interval. Hence, the algorithm computes the difference of both intervals, replaces the top interval of the tree with it and pops the top interval of the stream.



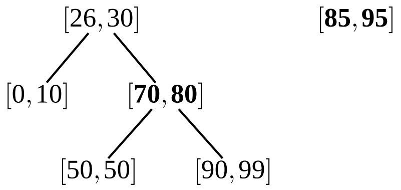
As the lower bound of the stream is above the upper bound of the top interval of the tree, it is safe to keep it and descend to the right subtree.



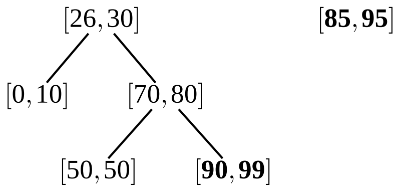
Since the upper bound of the stream's smallest interval is below the lower bound of the current interval of the tree, the algorithm descends to the left subtree.



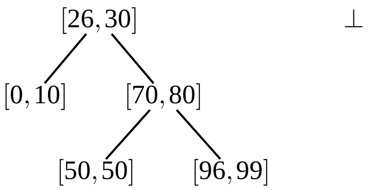
Again, the algorithm encounters an intersection of the two intervals that are currently focussed. Since the upper bound of the stream interval is below the upper bound of the tree interval, the algorithm computes the difference of both intervals, replaces the current interval of the tree with it and pops the top interval of the stream.



Since the lower bound of the stream is above the upper bound of the current interval, we are safe to keep it and descend to the right subtree.



Finally, we compute the intersection of the current interval and the last interval of the stream.



In this case, we are lucky to be able to maintain the overall structure of the original tree.

3.3.3 Union

Building the union of two diets t and s is a bit more complicated than computing their intersection or their difference for the following reason: say that the lower bound of the current interval of the stream s lies below the lower bound of the current interval of the tree t. Hence, we would make a recursive call to compute the union of the left subtree l and the stream s resulting in a new subtree l' and some remains s'. It may happen now that the subtree l' intersects with the current interval of the tree, namely in case that the largest interval in l intersects with an interval in the stream that intersects itself with the current interval of the tree.

In order to circumvent this problem, we add a *limitation parameter* which is just a value that is not to be exceeded by the left subtree. In this case, the limitation parameter would be related to the lower bound of the current interval of the tree. Assuming again that the largest interval of l intersects with a stream interval that intersects with the tree's current interval, we apply a little trick to keep all the data on the one hand and to stay below the limitation parameter on the other hand. Instead of adding the union of the largest interval of l and the related interval of s to l', we simply push it to the stream again. Therefore, we can deal properly with the intersection of this interval and the tree's current interval.

The union of two diets t and s again proceeds by traversing one of the two trees, say t, from left to right; the other tree, say s, is treated as a stream of ordered intervals that will be only accessed via the *extr* function.

As explained before, we need to add a parameter specifying the current limitation. There is no bound as initial limitation, hence we can use the value $\top$ which is either the maximal element of the underlying domain or a natural extension thereof.

```
fun $union(t,s)$ =
    let $(t',s') = unionhelp(t,s,\top)$ in
    $t' \bowtie s'$
```

We will call a helper function *unionhelp* accepting three parameters t, s and the limitation parameter $\max[\![t]\!] < \varepsilon$ that returns a pair (t',s') s.t. $[\![t']\!] \cup [\![s']\!] = [\![t]\!] \cup [\![s]\!]$, $\max[\![t']\!] < \min[\![s']\!]$, $\max[\![t']\!] < \varepsilon$ and $x \in [\![s']\!]$ with $x < \varepsilon$ implies $succ(x) \in [\![s']\!]$ (in other words, if the minimum of s' is below ε, then the lowest interval in s' contains ε) for the reasons explained in the first paragraphs.

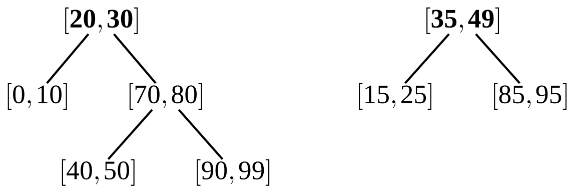
This particularly implies that calling *unionhelp* with the initial t and s with limitation $\top$ yields a pair (t',s') with s' being not necessarily empty. We only know that if s' is not empty then it lies above t'. Therefore, we simply need to rebalance s' (remember, we are using it as a stream) and combine it with t'.

```
fun $unionhelp(t,s,\varepsilon)$ =
    if $t = \bot$ or $s = \bot$ then $(t,\ s)$
    else let $([x,y],l,r) = t$ and $([x',y'],\_) = extr(s)$ in
        if $x' \geq x$ then $unionhelp2(l,[x,y],r,s,\varepsilon)$
        else let $(l',s') = unionhelp(l,s,pred(x))$ in
            $unionhelp2(l',[x,y],r,s',\varepsilon)$
```

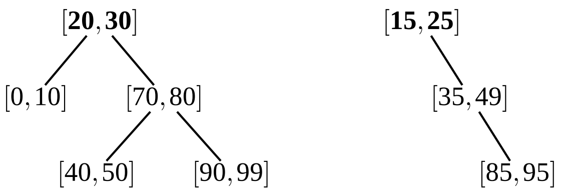
The helper function *unionhelp2* takes five parameters l, $[x,y]$, r, s and ε assuming that $\max[\![l]\!] < x$, $\max[\![l]\!] < \min[\![s]\!]$, $y < \varepsilon$ and $\max[\![r]\!] < \varepsilon$, and returns a pair (t',s') s.t. $[\![t']\!] \cup [\![s']\!] = [\![l\,{}^{x}{\bowtie}^{y}\,r]\!] \cup [\![s]\!]$, $\max[\![t']\!] < \min[\![s']\!]$, $\max[\![t']\!] < \varepsilon$ and $x \in [\![s']\!]$ with $x < \varepsilon$ implies $succ(x) \in [\![s']\!]$. In other words, *unionhelp2* assumes that l is a diet left of $[x,y]$ that already has been computed as the union of the original trees and hence simply attaches it to the union of the rest that is to be computed.

```
fun $unionhelp2(l,[x,y],r,s,\varepsilon)$ =
    if $s = \bot$ then $(l\,{}^{x}{\bowtie}^{y}\,r,\ \bot)$
    else let $([x',y'],u) = extr(s)$ in
        if $y' < pred(x)$ then
            let $l' = insert([x',y'],l)$ in
            $unionhelp2(l,[x,y],r,u,\varepsilon)$
        else if $x' > succ(y)$ then
            let $(r',s') = unionhelp(r,s,\varepsilon)$ in
            $(l\,{}^{x}{\bowtie}^{y}\,r',s')$
        else if $y \geq y'$ then
            let $i = \min(x,x')$ in
            $unionhelp2(l,[i,y],r,u,\varepsilon)$
        else if $y' \geq \varepsilon$ then
            let $i = \min(x,x')$ in
            $(l,([i,y'],u))$
        else let $i = \min(x,x')$ in
            let $(r',s') = unionhelp(r,([i,y'],u),\varepsilon)$ in
            $(l \bowtie r',s')$
```

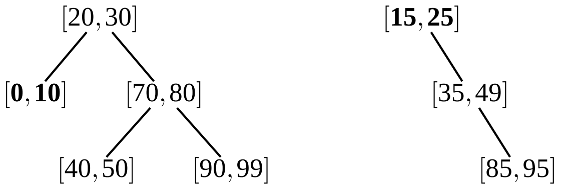
Consider the diets of the two examples from above again. We will follow the union algorithm on them in the same way: the right tree will be used as an ordered interval stream and the left tree will be used both as input and result tree.



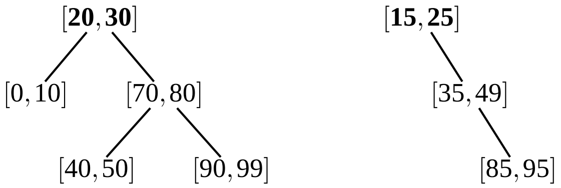
First, we need to perform a right-rotation on the stream to bring the smallest interval to the top of it.



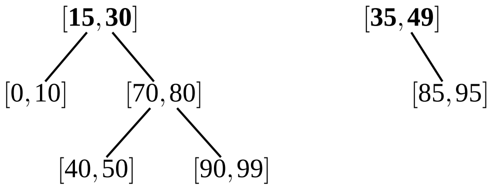
Comparing the top intervals, it could be the case that the left subtree of the tree contains an intersection with the stream, hence the algorithm descends the tree to the left.



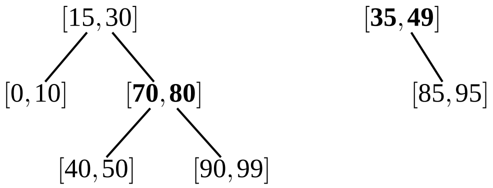
Since the current interval of the tree lies below the minimal interval of the stream, we can keep it and return to the higher level of the tree again.



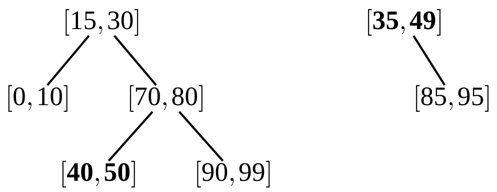
Next, the top intervals intersect and the upper bound of the stream interval is below the upper bound of the tree interval. Hence, the algorithm computes the union of both intervals, replaces the top interval of the tree with it and pops the top interval of the stream.



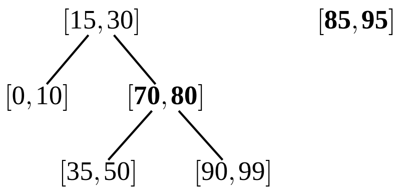
As the lower bound of the stream is above the upper bound of the top interval of the tree, it is safe to keep it and descend to the right subtree.



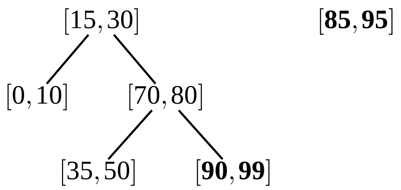
Since the upper bound of the stream's smallest interval is below the lower bound of the current interval of the tree, the algorithm descends to the left subtree.



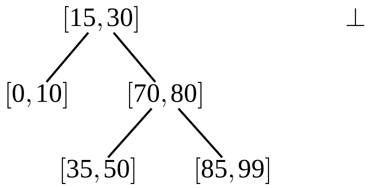
Again, the algorithm encounters an intersection of the two intervals that are currently focussed. Since the upper bound of the stream interval is below the upper bound of the tree interval, the algorithm computes the union of both intervals, replaces the current interval of the tree with it and pops the top interval of the stream.



Since the lower bound of the stream is above the upper bound of the current interval, we are safe to keep it and descend to the right subtree.



Finally, we compute the union of the current interval and the last interval of the stream.



3.4 Worst-Case Complexities

It is not too hard to see that all three routines run in time that is linearithmic in the number of nodes of the input diets. All three binary routines are based on a recursive descent of the first tree and a diet decomposition of the second tree, hence $\mathscr{O}(n)$ is required to walk through all nodes. A recursive call also possibly includes one rebalancing call and hence we get an additional rebalancing factor of $\mathscr{O}(\log n)$.

Lemma 2 *Let r and s be balanced diets with at most n nodes. The worst-case complexity of inter(r,s), union(r,s) and diff(r,s) described in Sect. 3.3 is $\mathscr{O}(n \cdot \log n)$.*

3.5 Linear Binary Operations

There is an alternative to the three algorithms presented above. Instead of flattening only one of the trees into a stream and using the structure of the other tree for recursion, one can flatten both trees into streams, perfom the corresponding operations on them and recreate a balanced tree from the resulting stream.

Is easy to see that flattening a tree into a list can be realized by a depth-first traversal of the tree in time $\mathcal{O}(n)$. Also, given two flattened trees, intersection, union and difference can be computed in time $\mathcal{O}(n)$ by walking through both streams simultaneously, resulting in an ordered list. It is known that ordered lists can be transformed back into balanced trees in time $\mathcal{O}(n)$ (Hinze, 1999).

Lemma 3 *Let r and s be balanced diets with at most n nodes. The worst-case complexity of inter*(r,s)*, union*(r,s) *and diff*(r,s) *described in Sect. 3.5 is* $\mathcal{O}(n)$.

3.6 Other Operations

Other operations that are usually carried out on sets are *filtering* w.r.t. a given predicate, *partitioning* w.r.t. a given predicate and *splitting* w.r.t. a given number. Splitting is a standard operation on balanced trees and essentially runs just the same on balanced diets (Ohnishi *et al.*, 2003). As partitioning is almost the same as filtering, we focus on a description of the latter operation here.

Standard filtering on balanced trees is usually realized by a recursion on the input tree, applying the filter predicate on the subtrees first, and then checking whether the root node matches the predicate or not. Depending on that either a reroot or a join of the filtered subtrees is carried out. With balanced diets, the root node has to be treated a bit differently: applying the predicate to each number of the represented interval of the root node results in a list of potentially separated numbers that have to be reassembled to a list of intervals again. If the length of the list is zero, we apply the reroot operation again, if the length is one, we apply the join again, and otherwise, we join the subtrees with the first interval of the list and insert all the others by applying the *insert* operation.

4 Empirical Evaluation

Our publicly available prototype implementation (Friedmann & Lange, 2010) of the balanced diets is realized in the functional language OCaml. It defines the signature for a so-called `MeasurableType` that explains how to compare, increment or decrement elements of the considered type and how to compute the counting measure distance between two elements[3]. A concrete instantiation of a `MeasurableType` can then be mapped via a functor to a concrete instantiation of the `Set`[4] signature.

We only consider the binary set operations – *union, intersection* and *difference* – as they particularly benefit from our genuine diet decomposition. We compare the following approaches with each other:

[3] Used for efficient computation of the cardinality.

[4] `http://caml.inria.fr/pub/docs/manual-ocaml/libref/Set.html`

Intervals	Density	Union					Intersection					Difference				
		OCamlSet	Camomile	CamlDiets	LinearOp	PatriciaSet	OCamlSet	Camomile	CamlDiets	LinearOp	PatriciaSet	OCamlSet	Camomile	CamlDiets	LinearOp	PatriciaSet
20,000	50%	0.16s	0.03s	0.02s	0.03s	0.15s	0.12s	0.03s	0.02s	0.03s	0.12s	0.08s	0.04s	0.02s	0.03s	0.05s
40,000	50%	0.19s	0.05s	0.04s	0.07s	0.17s	0.14s	0.05s	0.03s	0.06s	0.13s	0.10s	0.07s	0.03s	0.06s	0.07s
60,000	50%	0.21s	0.08s	0.06s	0.10s	0.19s	0.14s	0.07s	0.05s	0.09s	0.14s	0.11s	0.11s	0.05s	0.09s	0.08s
80,000	50%	0.23s	0.11s	0.07s	0.13s	0.20s	0.15s	0.09s	0.07s	0.11s	0.14s	0.12s	0.15s	0.06s	0.11s	0.09s
100,000	50%	0.24s	0.13s	0.09s	0.16s	0.22s	0.15s	0.12s	0.08s	0.14s	0.14s	0.14s	0.19s	0.08s	0.15s	0.10s
120,000	50%	0.25s	0.16s	0.10s	0.19s	0.23s	0.15s	0.14s	0.10s	0.17s	0.14s	0.14s	0.22s	0.09s	0.17s	0.10s
140,000	50%	0.26s	0.18s	0.12s	0.22s	0.24s	0.15s	0.16s	0.11s	0.20s	0.14s	0.14s	0.25s	0.10s	0.19s	0.11s
160,000	50%	0.27s	0.20s	0.13s	0.28s	0.24s	0.15s	0.18s	0.12s	0.24s	0.14s	0.15s	0.29s	0.11s	0.23s	0.12s
180,000	50%	0.30s	0.24s	0.16s	0.33s	0.28s	0.17s	0.21s	0.15s	0.31s	0.16s	0.16s	0.34s	0.15s	0.27s	0.13s
200,000	50%	0.31s	0.27s	0.18s	0.38s	0.28s	0.17s	0.23s	0.15s	0.32s	0.16s	0.16s	0.37s	0.15s	0.27s	0.12s
220,000	50%	0.29s	0.27s	0.18s	0.36s	0.26s	0.15s	0.24s	0.16s	0.32s	0.14s	0.15s	0.38s	0.16s	0.28s	0.12s
240,000	50%	0.29s	0.29s	0.19s	0.39s	0.27s	0.15s	0.26s	0.17s	0.33s	0.14s	0.15s	0.42s	0.17s	0.30s	0.12s
260,000	50%	0.32s	0.33s	0.22s	0.46s	0.30s	0.17s	0.29s	0.20s	0.39s	0.15s	0.16s	0.48s	0.20s	0.36s	0.14s
280,000	50%	0.34s	0.36s	0.24s	0.50s	0.31s	0.16s	0.31s	0.20s	0.40s	0.15s	0.16s	0.49s	0.20s	0.36s	0.13s
300,000	50%	0.29s	0.34s	0.22s	0.45s	0.27s	0.15s	0.30s	0.19s	0.35s	0.13s	0.15s	0.49s	0.19s	0.34s	0.12s
320,000	50%	0.31s	0.36s	0.24s	0.49s	0.29s	0.15s	0.32s	0.21s	0.40s	0.13s	0.15s	0.53s	0.20s	0.38s	0.13s
340,000	50%	0.31s	0.38s	0.25s	0.52s	0.29s	0.15s	0.34s	0.22s	0.41s	0.14s	0.16s	0.60s	0.23s	0.43s	0.14s
360,000	50%	0.34s	0.42s	0.28s	0.60s	0.33s	0.16s	0.37s	0.25s	0.49s	0.15s	0.17s	0.63s	0.24s	0.45s	0.14s
380,000	50%	0.34s	0.42s	0.29s	0.60s	0.32s	0.15s	0.38s	0.24s	0.46s	0.14s	0.16s	0.63s	0.24s	0.43s	0.13s
400,000	50%	0.31s	0.41s	0.27s	0.56s	0.29s	0.15s	0.38s	0.24s	0.45s	0.13s	0.15s	0.63s	0.23s	0.41s	0.13s

(a) Interval benchmark

Density	Intervals	Union					Intersection					Difference				
		OCamlSet	Camomile	CamlDiets	LinearOp	PatriciaSet	OCamlSet	Camomile	CamlDiets	LinearOp	PatriciaSet	OCamlSet	Camomile	CamlDiets	LinearOp	PatriciaSet
10%	90,000	0.06s	0.11s	0.10s	0.18s	0.05s	0.02s	0.06s	0.03s	0.06s	0.01s	0.04s	0.16s	0.06s	0.12s	0.03s
15%	127,500	0.10s	0.15s	0.14s	0.25s	0.08s	0.03s	0.09s	0.05s	0.10s	0.02s	0.06s	0.22s	0.09s	0.16s	0.05s
20%	160,000	0.13s	0.19s	0.17s	0.30s	0.11s	0.05s	0.12s	0.07s	0.13s	0.04s	0.07s	0.28s	0.11s	0.21s	0.06s
25%	187,500	0.16s	0.23s	0.19s	0.34s	0.14s	0.06s	0.14s	0.09s	0.17s	0.05s	0.09s	0.32s	0.13s	0.24s	0.07s
30%	210,000	0.18s	0.25s	0.20s	0.37s	0.16s	0.08s	0.17s	0.11s	0.20s	0.06s	0.10s	0.35s	0.14s	0.26s	0.08s
35%	227,500	0.21s	0.27s	0.21s	0.38s	0.19s	0.10s	0.20s	0.13s	0.23s	0.08s	0.12s	0.39s	0.15s	0.29s	0.10s
40%	240,000	0.23s	0.28s	0.20s	0.40s	0.21s	0.12s	0.23s	0.15s	0.30s	0.10s	0.14s	0.44s	0.18s	0.32s	0.12s
45%	247,500	0.29s	0.32s	0.23s	0.46s	0.27s	0.14s	0.26s	0.18s	0.35s	0.13s	0.15s	0.45s	0.18s	0.34s	0.13s
50%	250,000	0.28s	0.29s	0.20s	0.41s	0.27s	0.15s	0.26s	0.18s	0.33s	0.14s	0.15s	0.43s	0.17s	0.31s	0.12s
55%	247,500	0.31s	0.29s	0.18s	0.41s	0.29s	0.17s	0.28s	0.19s	0.34s	0.16s	0.16s	0.43s	0.18s	0.31s	0.13s
60%	240,000	0.37s	0.30s	0.19s	0.44s	0.36s	0.21s	0.30s	0.21s	0.39s	0.20s	0.18s	0.44s	0.19s	0.35s	0.15s
65%	227,500	0.40s	0.29s	0.17s	0.36s	0.40s	0.22s	0.29s	0.21s	0.55s	0.22s	0.17s	0.39s	0.17s	0.30s	0.14s
70%	210,000	0.37s	0.24s	0.13s	0.28s	0.37s	0.23s	0.27s	0.19s	0.51s	0.23s	0.16s	0.36s	0.16s	0.28s	0.14s
75%	187,500	0.39s	0.22s	0.11s	0.25s	0.41s	0.26s	0.26s	0.18s	0.54s	0.27s	0.17s	0.34s	0.15s	0.29s	0.16s
80%	160,000	0.44s	0.21s	0.09s	0.21s	0.48s	0.31s	0.25s	0.18s	0.56s	0.33s	0.17s	0.29s	0.14s	0.24s	0.15s
85%	127,500	0.48s	0.17s	0.07s	0.18s	0.51s	0.32s	0.20s	0.15s	0.50s	0.36s	0.16s	0.23s	0.11s	0.18s	0.14s
90%	90,000	0.45s	0.12s	0.04s	0.11s	0.50s	0.32s	0.14s	0.11s	0.43s	0.35s	0.15s	0.15s	0.08s	0.12s	0.12s

(b) Density benchmark

Fig. 1. Runtime results.

1. *OCamlSet* (Leroy, 2010): the original OCaml Set implementation by Leroy,
2. *Camomile* (Yoriyuki, 2003): the diet implementation by Yoriyuki,
3. *CamlDiets* : the balanced diet implementation described above,
4. *LinearOp* : the alternative method with linear binary operations as described in Sect. 3.5, and
5. *PatriciaSet* (Filliâtre, 2008): the Patricia set implementation by Filliâtre.

The empirical evaluation is based on two different classes of randomized sets within the fixed domain $\mathscr{U} = \{1,\ldots,10^6\}$. This range allows us to generate sufficiently large sets with non-neglegible running times when being fed to the binary operations. Given a single benchmark instance, we generate 100 sets matching the constraints of the instance uniformly at random, and carry out each binary operation of each set implementation on all pairs of generated sets (i.e. $100 \cdot 99$); every operation is repeated 10 times to improve the accuracy of the empirical measurements. Finally, the average running time (i.e. overall time divided by $100 \cdot 99 \cdot 10$) is calculated and included in the tables of Fig. 1.

All tests have been carried out on a 64-bit machine with Opteron$^{\text{TM}}$ CPUs. The implementation does not support parallel computations, hence, each test is run on one core only.

We briefly describe the parameters used to measure the benchmark sets. Let $\mathscr{U}$ be a fixed domain. We consider three kinds of measures for sets $S \subseteq \mathscr{U}$. Here, a *measure* is a function $\mu : 2^{\mathscr{U}} \to \mathbb{R}$.

First, we consider the standard *counting measure* $\mu_C(S) := |S|$, giving the number of elements in a set S. Second, we consider the *density measure* $\mu_D(S) := \frac{|S|}{|\mathscr{U}|}$ that relates the number of elements to the overall size of the domain. Last, we specify the *interval measure* $\mu_I(S) := |\{[x,y] \subseteq S \mid x \leq y,\, (x-1) \notin S \text{ and } (y+1) \notin S\}|$ that counts the number of independent intervals contained in S.

Given a (finite) family $\mathscr{F} \subseteq 2^{\mathscr{U}}$ of sets and a measure μ, we define the *expected measure* $E_{\mathscr{F}}[\mu]$ as the expected value of the random variable μ with a uniform distribution over the elements of $\mathscr{F}$, i.e.

$$E_{\mathscr{F}}[\mu] = \frac{1}{|\mathscr{F}|} \sum_{S \in \mathscr{F}} \mu(S)$$

All considered set representation approaches are based on binary trees, therefore we are interested in the numbers of nodes and the heights of the trees representing the sets. Let $S \subseteq \mathscr{U}$. The *number of nodes* $\alpha_{\text{impl}}(S)$ that is required to represent S which is $\mu_I(S)$ for impl $=$ *Camomile*, *CamlDiets*, *LinearOp*, and is $\mu_C(S)$ for *OCamlSet*; and the *height of the representation* $\beta_{\text{impl}}(S)$ which is logarithmic in $\alpha_{\text{impl}}(S)$ in these four cases. We ignore empty leaf nodes here. For impl $=$ *PatriciaSet* these measures depend on the actual distribution of the set in the underlying domain and are therefore not easy to estimate. The number of elements is of course an upper bound on the representation size up to a constant factor, and the uniform random distribution of the elements should result in balanced Patricia tries. Thus, their height can be expected to be logarithmic in the size as well.

Finally, we describe the two benchmark settings in which we carry out the three operations on several sets.

Interval Benchmark The *interval benchmark* is based on the class of sets from the domain $\mathscr{U}$ s.t. the number of independent intervals equals a given constant c. We consider therefore the family of classes $\mathscr{I}_c = \{S \subseteq 2^{\mathscr{U}} \mid \mu_I(S) = c\}$. It is easy to see that $E_{\mathscr{I}_c}[\mu_I] = c$, $E_{\mathscr{I}_c}[\mu_S] = 0.5 \cdot |\mathscr{U}| = 500,000$ and hence $E_{\mathscr{I}_c}[\mu_D] = 0.5$.

We perform the interval benchmark for different parameterizations c, ranging from $20,000$ intervals to $400,000$ intervals. Technically, our set generator uses the parameterization c to pick $2 \cdot c$ pairwise different numbers $start_1 < end_1 < \ldots < start_c < end_c$ from the domain $\mathscr{U}$ and uses the ordered sequence of $2 \cdot c$ numbers to derive a set of intervals $[start_i, end_i]$.

The average times needed to build the union, intersection or difference of two sets with the same number of intervals are presented in Fig. 1(a).

The following observations can be made. (1) The running times of all implementations rise with the number of intervals. This is to be expected. (2) The binary routines of the balanced diet approach generally outperform Yoriyuki's Camomile method as well as the alternative diet binary routines. (3) The balanced diet implementation generally yields a better average running time than all other methods. This might be due to the stream decomposition based approach. (4) For sets with a small number of intervals, the diet-based approaches perform much better than Patricia sets and standard sets. (5) For sets with a high number of intervals, diet-based approaches are beaten by the other two. Particularly, Patricia sets always seem to be a bit better than the original set approach, but not by a large amount.

Density Benchmark The *density benchmark* on the other hand is based on the class of sets from the domain $\mathscr{U}$ s.t. the cardinality of the set divided by the cardinality of the domain is very close to a given proportionality degree or *density* $0 \leq p \leq 1$. More formally, we consider the family of classes $\mathscr{D}_p = \{S \subseteq 2^{\mathscr{U}} \mid \mu_D(S) = p\}$. It is not too hard to see that $E_{\mathscr{D}_p}[\mu_I] = p \cdot (1-p) \cdot |\mathscr{U}| = p \cdot (1-p) \cdot 500,000$, $E_{\mathscr{D}_p}[\mu_S] = p \cdot |\mathscr{U}| = p \cdot 500,000$ and hence $E_{\mathscr{D}_p}[\mu_D] = p$.

We perform the density benchmark for different parameterizations p, ranging from 0.1 to 0.9. Technically, our set generator uses the parameterization p to pick every element $e \in \mathscr{U}$ with probability p.

The average time needed to build the union, intersection or difference of two sets with the same density are presented in Fig. 1(b).

The following observations can be made. (1) The running times of the balanced diet approach rise with the number of intervals rather than the density. They are particularly low for sets with high density and therefore few intervals. (2) Again, the binary routines of Yoriyuki's Camomile method as well as the alternative diet method are outperformed by the balanced diet implementation. (3) The standard set approach and the Patricia sets beat the diet-based methods on sets with a small density, since low-density sets consist of a large number of intervals compared to the number of elements.

5 Conclusion

We considered the representation of sets as balanced diets and introduced the concept of the so-called diet decomposition that allows us to realize highly efficient binary routines on sets. We provide empirical justifications, showing that even mildly populated sets can benefit from the representation as balanced diets.

The evaluation section above answers a few preliminary questions about the use of balanced diets. A much more elaborate investigation is of course possible, for instance considering sets that occur in certain scenarios rather than randomly generated ones, variations of other parameters like the domain size, etc. We remark that tests which combine two sets of different sizes / densities have shown similar comparisons of the running times between the methods considered here.

Also, it would be interesting to combine the decomposition approach used for the balanced diets here with a non-diet representation and examine the effect it would have on those data structures.

Acknowledgment. We would like to thank the anonymous referees for their thorough reports containing many comments that helped to improve the presentation of this paper. In particular, they have pointed out the possibility to obtain linear algorithms as described at the end of Sect. 3.4.

References

Adams, S. (1993). Efficient sets - a balancing act. *Journal of functional programming*, **3**(4), 553–561.

Adelson-Velskii, G., & Landis, E. M. (1962). An algorithm for the organization of information. *Pages 263–266 of: Proceedings of the ussr academy of sciences*, vol. 146.

Bayer, R. (1972). Symmetric binary B-trees: Data structure and maintenance algorithms. *Acta informatica*, **1**, 290–306.

Cormen, T. H., Leiserson, C. E., & Rivest, R. L. (1992). *Introduction to algorithms*. 6th edn. MIT Press and McGraw-Hill Book Company.

Erwig, M. (1998). Functional pearls: Diets for fat sets. *Journal of functional programming*, **8**(6), 627–632.

Filliâtre, J.-C. (2008). *Patricia set*. `http://www.lri.fr/~filliatr/software.en.html`.

Friedmann, O., & Lange, M. (2010). *Camldiets*. `http://www2.tcs.ifi.lmu.de/camldiets`.

Guibas, L. J., & Sedgewick, R. (1978). A dichromatic framework for balanced trees. *Pages 8–21 of: 19th ann. symp. on foundations of computer science, FOCS'78*. IEEE.

Gwehenberger, G. (1968). Anwendung einer binären Verweiskettenmethode beim Aufbau von Listen. *Elektronische rechenanlagen*, **10**(5), 223–226.

Hinze, Ralf. (1999). Constructing red-black trees. *Pages 89–99 of: In proceedings of workshop on algorithmic aspects of advanced programming languages*.

Leroy, X. (2010). *Ocaml set*. `http://caml.inria.fr/pub/docs/manual-ocaml/libref`.

Morrison, D. R. (1968). PATRICIA: Practical algorithm to retrieve information coded in alphanumeric. *Journal of the acm*, **15**(4).

Ohnishi, S., Tasaka, H., & Tamura, N. (2003). Efficient representation of discrete sets for constraint programming. *Pages 920–924 of: Proc. 9th int. conf. on principles and practice of constraint programming, CP'03*. LNCS, vol. 2833. Springer.

Yoriyuki, Y. (2003). *Camomile set*. `http://camomile.sourceforge.net`.