dsinf: Inferencja stuktur danych oparta na kodzie źródłowym

Autor: Aleksander Balicki

Promotor: Prof. Witold Charatonik

Wrzesień 2014
Streszczenie

Większość popularnych dzisiaj języków programowania posiada już zaimplementowane biblioteki z wieloma dostępnymi strukturami danych do przechowywania danych. Programiści po prostu muszą nauczyć się, kiedy należy używać danej struktury. W niektórych przypadkach ten wybór jest na tyle prosty, że program mógłby automatycznie dobrać najlepszą strukturę danych. Praca ta opisuje dsinf — framework do dobierania najlepszej struktury danych pasującej do zadania na podstawie kodu źródłowego programu w języku C. dsinf analizuje przypadki użycia struktur danych w całym programie i proponuje najszybszą. Do zapewnienia jakości tej sugestii, używa danych profilera.
Author: Aleksander Balicki

Supervisor: Prof. Witold Charatonik

September 2014
Abstract

When you need to store data in a data structure, most languages popular today already have libraries with convenience data structures implemented. Programmers just have to be taught when to use a particular data structure. Some of the cases of choosing the right data structure look sufficiently easy, so a program could do it automatically. This work describes the dsinf project, an analysis framework for inferring the best data structure matching the task, based on the program's source code in C language. dsinf analyzes the uses of data structures throughout the program and suggests the fastest one. To further enhance the quality of the suggestion, it uses profiler data.
# Contents

1 Introduction 4
   1.1 What is dsinf? .................. 4
   1.2 Comparing to a database ........ 5
   1.3 Comparing to class clusters .... 5
   1.4 C API .......................... 5

2 Data structure inference 7
   2.1 Comparison of the complexities  7
   2.2 Choosing the best data structure 8
   2.3 Collecting the program data .... 9

3 Extensions of the idea 16
   3.1 Second extremal element ........ 16
   3.2 Importance of operations ......... 16
      3.2.1 Code pragmas ................ 18
      3.2.2 Choosing the best data structure with weights 19
      3.2.3 Profile-guided optimization .. 22
      3.2.4 Transforming data structures on-line 22
   3.3 Different element types .......... 24
      3.3.1 Linked data structures ...... 24
   3.4 Upper bound on the element count 29
   3.5 Minimal element count threshold 30
   3.6 Generic data structure modifications 31
      3.6.1 Extremal element cache ..... 32
      3.6.2 Linked elements .......... 32

4 Future work 34

A Implementation 35
   A.1 Progress ....................... 35
   A.2 Compilation .................... 35
1 Introduction

In computer science, a data structure is a particular way of storing and organizing data in a computer so that it can be used efficiently[1].

The interface of a data structure is a set of operations that you can perform on a data structure and some semantics of those operations.

The implementation of a data structure is a realization of some data structure interface in some programming language, that is consistent with the semantics.

There are a lot of different known types of data structures:

- Map/associative array/dictionary is a data structure for storing \((key, value)\) pairs, it allows insertion, deletion, modification and lookup,
- multimap is the same as map, but with the possibility of storing multiple values for a one key,
- list is a data structure for storing ordered sets of values,
- set is a data structure for storing unordered sets of values,
- multiset is the same as set, but with the possibility of storing multiple objects that are equal under some equivalence relation,
- queue is a data structure implementing operations for queuing and dequeuing elements,
- deque is a queue, where you can add and remove elements from both ends as opposed to one,
- stack is a data structure implementing operations for putting something on the top of the stack or removing the top element from it,
- priority queue is like a regular queue, but each element has a priority associated with it, which is used for ordered dequeuing of the elements,
- string is a data structure for storing text,
- tree is a data structure, to hold some tree-like hierarchical structure,
- graph is a data structure, to store elements and a relation on those elements,
- others, which usually are a modification or a mix of previous ones.

A different implementation of those data structures makes it possible, that specific operations run time is lower than the others. So there is no data structure perfect for all the tasks.

1.1 What is dsinf?

dsinf is a source code analyzer. It works by analyzing specially prepared C code. That C code, for every instance of a data structure, uses a type defined in the dsinf library. This type is called a data structure wildcard. The type represents a data structure with the most general interface, i.e. a set union of interfaces of all data structures available in the dsinf framework. It connects the data
structures and operations performed on them and then reasons about which of the available data structure implementations would be best for the instances of data structure wildcards in your code, finally (if possible) choosing the best matches and printing them out or linking them to your code to create a working binary.

The above analysis now works after compiler C code preprocessing (using CPP or a preprocessor included in a C compiler), before the actual compilation. In the future it can be pushed to the phase between compilation and linking as described in section 4.

The analysis process uses asymptotic complexity for data structure comparison. The process can also be modified by providing source code annotations. One can also use test data to tailor the data structure as described in section 3.2.

The idea of the framework is to provide a data structure, that is always "good enough". Due to unsolvability of the general case of the problem, the effect always will be worse than a hand tailored data structure to your task, but can save up some time (both programming time and execution time) if time is an important factor. To reiterate, using dsinf for the most critical bottleneck of your application probably is not a good idea.

1.2 Comparing to a database

There is already a mature solution for storing data and querying it in an efficient way — databases. The main difference between dsinf and a database is that a database has to implement all the operations for all the possible SQL queries, including some crosschecks, counting, summations, extremal elements and with dsinf your data structure implementation does not have to be ready for all the possible SQL queries that can be constructed, but only the ones you used in your source code, which enables it to potentially gain some performance over a database, both in space and time complexity.

1.3 Comparing to class clusters

A class cluster is an architecture that groups a number of private, concrete subclasses under a public, abstract superclass. The grouping of classes in this way provides a simplified interface to the user, who sees only the publicly visible architecture. Behind the scenes, though, the abstract class is calling up the private subclass most suited for performing a particular task[9]. One could have an object representing e.g. an array, and have a lot of different private implementations for it, like large, sparse arrays, or arrays with sizes known at compile time and optimize operations for those cases. The downside of such method is that you can alternate between the data structure representations that need a lot of time to transform into each other. dsinf knows exactly what operations can occur and tailors the data structure to the operations that will happen.

1.4 C API

The framework detects specific C functions in the source code and bases the inference on them. Now we define what those functions are. The first API version is in Figure 1.

There is a problem with the Figure 1 API. If there is an operation on a data structure that is
struct ds;
typedef struct ds *ds;

ds init_d();
void insert_d(ds, dtype);
void update_d(ds, dtype, dtype);
void delete_d(ds, dtype);
void delmax_d(ds);
dtype search_d(ds, dtype);
dtype max_d(ds);
dtype min_d(ds);

Figure 1: The simplified dsinf C API

asymptotically faster than the speed of a lookup of an element, it is impossible to obtain in the desired time with this API. For example, a balanced search tree that guarantees $O(\log n)$ lookups, with a linked list of all the elements (like in section 3.6.2) to enhance the speed of the successor and predecessor operations to $O(1)$. Defining a successor function using this API convention, it would look like this:

dtype successor_d(ds, dtype);

The problem with this function is that it does not take a pointer to the element in a data structure, so it has to actually find the value given in the actual parameter of the function, and then compute the successor in $O(1)$, but the search takes $O(\log n)$, so the whole operation takes $O(\log n)$, which is bad.

To fix this we declare a new type for a data structure element. The type encompasses a pointer to the place in the data structure. The search can be started from this pointer, so there is no need to waste time on finding the element.

The second version of the API is in Figure 2. It is updated to use the dselem struct. We use the pointer value instead of the simple value like before.

The following code examples will sometimes use Figure 1 API convention for simplicity.
// use #defines the dstype in its code before including this

```c
struct dselem;
struct ds;

typedef struct ds *ds;
typedef struct dselem *dselem;

ds init_d();
dselem insert_d(ds, dstype);
dselem update_d(ds, dselem, dstype);
void delete_elem_d(ds, dselem);
void delete_val_d(ds, dstype);
void delmax_d(ds);
void delmin_d(ds);
dselem search_d(ds, dstype);
dselem max_d(ds);
dselem min_d(ds);
dselem successor_d(ds, dselem);
dselem predecessor_d(ds, dselem);
```

Figure 2: The dsinf C API

2 Data structure inference

2.1 Comparison of the complexities

If we want to find the best possible data structure for a task, we have to define some kind of ordering on the data structures, so we can judge which structures are better than others. Here we want to compare the asymptotic complexities of operations on data structures. When trying to define such ordering, we encounter a problem. We can not do it for the general case, because the complexity of an operation can be described by arbitrary functions and a comparison of asymptotic complexities is complicated.

Asymptotic complexity of an operation is stored as a pair of type:

\[ \text{AsymptoticalComplexity} = \text{Int} \times \text{Int}, \]

where

\[ (k, l) \text{ means } O(n^k \log^l n). \]

The reason to choose such a type is that it is easier to compare than the general case (we can do a lexicographical comparison of the two numbers) and it distinguishes most of the data structure operation complexities.
Sometimes we have to use some qualified complexities:

\[ \text{ComplexityType} = \{ \text{Normal}, \text{Amortized}, \text{Amortized Expected}, \text{Expected} \} \]  

(3)

The overall complexity can be seen as a type:

\[ \text{Complexity} = \text{AsymptoticalComplexity} \times \text{ComplexityType} \]  

(4)

Here we can also use a lexicographical comparison, but we have to say that

\[
\begin{align*}
\text{Amortized} &> \text{Normal}, \\
\text{Expected} &> \text{Amortized}, \\
\text{Amortized Expected} &> \text{Expected}
\end{align*}
\]

(5) (6) (7)

and that > is transitive.

The ordering is done in such a way, because normal complexity of \( O(f(n)) \) guarantees that no operation will exceed that time multiplied by a constant. Amortized is worse, because here, we can only guarantee that \( n \) consecutive operations will have the average time \( O(f(n)) \), but single operations can spike heavily. Expected is worse than amortized, because here we can only have some probability of achieving the mentioned complexity for an operation and there is no bound on the number of spikes, which were under control in an amortized complexity.

We also always choose the smallest asymptotic-complexity-wise complexity. For example, we have a search operation on a splay tree. It is \( O(n) \), but \( O(\log n) \) amortized, so it is represented as \(((0, 1), \text{Amortized})\).

### 2.2 Choosing the best data structure

We define a set \( \text{DataStructureOperation} \). We can further extend this set, but for now assume that

\[ \text{DataStructureOperation} = \{ \text{Insert, Update, Delete, FindMax, DeleteMax, \ldots} \}. \]  

(8)

Each of the \( \text{DataStructureOperation} \) elements symbolizes an operation you can accomplish on a data structure.

The type

\[ \text{DataStructure} = \text{DataStructureOperation} \rightarrow \text{Complexity} \]  

(9)

represents a data structure \( ds \) and all of the operations implemented for it, with their complexities, as a partial function \( f \) from \( \text{DataStructureOperation} \) to \( \text{Complexities} \). It represents some implementation of a data structure, where existing arguments of \( f \) are the operations, which are possible
to accomplish on ds, and the value of the function represents asymptotic speed of the operation. For example Table 1 presents such representation for Red-Black Trees.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insert</td>
<td>(O(\log n))</td>
</tr>
<tr>
<td>Update</td>
<td>(O(\log n))</td>
</tr>
<tr>
<td>Delete</td>
<td>(O(\log n))</td>
</tr>
<tr>
<td>Find Max</td>
<td>(O(\log n) / O(1))</td>
</tr>
<tr>
<td>Delete Max</td>
<td>(O(\log n))</td>
</tr>
<tr>
<td>Search</td>
<td>(O(\log n))</td>
</tr>
<tr>
<td>Successor</td>
<td>(O(\log n) / O(1))</td>
</tr>
</tbody>
</table>

Table 1: A table showing the representation of Red-Black trees; complexity varies on implementation, speed-up technique described in section 3.6

When trying to find the best suited data structure for a given program \(p\), we look for data structure uses in \(p\). Let

\[
DSU(p) ::= [P(DataStructureOperation)]
\]  

(10)

be a set of groups of data structure operations. One group represents operations on one persistent data structure identity in the source code of \(p\).

For every \(ds \in DSU(p)\), we define a parametrized comparison operator for data structures \(\prec_{ds}\) defined as:

\[
d_1 \prec_{ds} d_2
\]

(11)

\[
|\{(o,c_1) \in d_1 | o \in ds \land o \text{ used in } p \land (o,c_2) \in d_2 \land c_1 < c_2\}| < 0.5 * |\{o \in ds | o \text{ used in } p\}|
\]

(12)

Equation 12 defines an order we use for the comparison of data structures. Intuitively a data structure \(d_1\) is better than a data structure \(d_2\) if it implements "faster" at least half of types of operations used in \(p\).

For a fixed \(p\), we have a preorder on data structures induced by \(\prec_{ds}\) and we can sort data structures available to the framework using this order. The maximum element is the best data structure implementation for the persistent data structure identity.

2.3 Collecting the program data

In Equation 10 the type of the \(DSU()\) operation was mentioned. This section shows, how \(DSU()\) is defined. We are not considering the problem of 2 different variables having the same name.
and/or shadowing each other, to evade this we can just rename each variable to an UUID, and then remember the mapping for nicer output.

Let $DSOPS(p) :: [(\text{VariableName, DataStructureOperation})]$ be the union of:

1. $\{(g, o) \mid g \text{ is a global data structure variable in } p \land o \text{ is an operation performed on the value of } g, \text{ somewhere in } p\}$

2. $\{(d, o) \mid d \text{ is a data structure declared somewhere in a body of a function } f \text{ in } p \land o \text{ is an operation performed on the value of } d, \text{ somewhere in } f\}$

3. $\{(p', o) \mid p' \text{ is a formal parameter of data structure type, of a function } f \text{ in } p \land o \text{ is an operation performed on the value of } p', \text{ somewhere in } f\}$

An example of the first rule is in Figure 3. In the example there are two global data structure variables declared ($global\_ds\_1, global\_ds\_2$) and in the bodies of functions we perform some operations on them. The rule returns list of pairs of data structures and operations as they were called.

An example of the second rule is in Figure 4. This rule is similar to the first rule, but it works on variables with an auto storage specifier (local variables in C code) like the $declared\_ds$ variable in the example.

An example of the third rule is in Figure 5. This rule works similarly to the two above, only gathering results on variables declared as a function parameter of any function in the program, like the $parameter\_ds$ variable in the example.
```c
typedef int ds_type;
#include "../dsimp/ds.h"

ds global_ds;
ds global_ds2;

int main()
{
    insert_d(global_ds, 5);
    delete_max_d(global_ds);
    printf("%d\n", max_d(global_ds));
    insert_d(global_ds2, 7);
    delete_d(global_ds2, 5);
    delete_d(global_ds2, 5);
    f();
}

void f()
{
    update_d(global_ds2, 7, 2);
}

//yielding:
//    [(global_ds, insert_d), (global_ds, delete_max_d),
//     (global_ds, max_d),
//     (global_ds2, insert_d), (global_ds2, delete_d),
//     (global_ds2, delete_d), (global_ds2, update_d)]

Figure 3: Global variables collection rule

typedef int ds_type;
#include "../dsimp/ds.h"

int main()
{
    ds declared_ds;
    insert_d(declared_ds);
    delete_max_d(declared_ds);
    printf("%d\n", max_d(declared_ds));
}

//yielding:
//    [(declared_ds, insert_d), (declared_ds, delete_max_d),
//     (declared_ds, max_d)]

Figure 4: Declared variables collection rule
```
typedef int dtype;
#include "../dsimp/ds.h"

void f(ds parameter_ds)
{
    insert_d(parameter_ds, 2);
    delete_max_d(parameter_ds);
    printf("%d\n", max_d(parameter_ds));
}

int main()
{
    ds declared_ds;
    update_d(declared_ds, 5, 7)
    f(declared_ds);
}

//yielding:
// [(parameter_ds, insert_d), (parameter_ds, delete_max_d),
// (parameter_ds, max_d)]

Figure 5: Function parameters collection rule
We want to group the elements in $DSOPS(p)$ to detect the persistent identities [6] of data structures, meaning that in one group, there will be data structure operations conducted on one data structure from its allocation, to deallocation or the end of the program, counting in passing the structure as a parameter to another function or copying the pointer to the structure.

$DSU(p)$ is the list of groups created in this step. After this operation, each group has operations performed on the same persistent identity of a data structure. For every such group we find the best matching data structure as shown in section 2.2.

To group the operations:

- **Persistent Identities** - for every two pairs $(d_1, o_1)$ and $(d_2, o_2)$ created using item 1 or item 2, we put them in the same group if $d_1 = d_2$. This rule for the example Figure 3 would group all the operations on $global\_ds\_1$ into one group, and all the operations on $global\_ds\_2$ into another.

- **Function calls** - for every pair $(p, o_1)$ that was created by using item 3 on function $f$, which was called with the actual data structure parameter $d$ as the formal parameter $p$, we put $o_1$ into the group of operations on $d$. This rule for the example Figure 5 would group all the operations on $declared\_ds$ and $parameter\_ds$ into one group. A more complex example is described in Figure 6.

- **Copy propagation** - for every $(do, o_1)$ and $(dc, o_2)$, where the $dc$ variable was obtained by copying the value of the variable $do$, we put $o_1$ and $o_2$ in the same group. In Figure 7 this rule would group $declared\_ds$ and $copied\_ds$ into one group.

- **Uniqueness** - for every group, we delete repeating elements. We notice that in Figure 3 the method yields a list with $(global\_ds\_2, delete\_d)$ two times. This rule takes care of that and leaves only one.
typedef int dstype;
#include "../dsimp/ds.h"

void f(ds parameter_ds)
{
    delete_max_d(parameter_ds);
    printf("%d\n", min_d(parameter_ds));
}

int main()
{
    ds ds1, ds2, ds3;
    insert_d(d1, 5);
    insert_d(d2, 7);

    f(ds1);
    f(ds2);
    printf("%d\n", max_d(ds1));

    ds3 = ds2;
    update_d(ds3, 5, 7);
}

// parameter rule yields:
//
//    [( parameter_ds, delete_max_d ), ( parameter_ds, min_d )]
//
// declared rule yields:
//
//    [( ds1, insert_d ), ( ds1, max_d ),
//     ( ds2, insert_d ),
//     ( ds3, update_d )]
//
// grouping yields following groups:
//
//    [( ds1, insert_d ), ( ds1, max_d ), ( parameter_ds,
//     delete_max_d ), ( parameter_ds, min_d )],
//    [( ds2, insert_d ), ( parameter_ds, delete_max_d ),
//     ( parameter_ds, min_d ), ( ds3, update_d )]

Figure 6: Function call grouping rule
typedef int dstype;
#include ".../dsimp/ds.h"

int main()
{
    ds declared_ds;
    insert_d(declared_ds);

    ds copied_ds = declared_ds;
    printf("%d\n", max_d(copied_ds));
}

Figure 7: Copy propagation grouping rule
3 Extensions of the idea

3.1 Second extremal element

If we want to find the maximal element in a heap, we just look it up in $O(1)$, that is what heaps are for. If we want to find the minimal element we can modify the heap, for it to use an order, which would allow us to lookup the minimal element in $O(1)$. What happens if we want to find the max and the min element in the duration of one program? How to modify our framework to handle this kind of situations?

$$\text{DataStructureOperation} = \{ \ldots, \text{FindFirstExtremalElement}, \text{DeleteFirstExtremalElement}, \text{FindSecondExtremalElement}, \text{DeleteSecondExtremalElement} \}.$$  

Now we can add two complexity costs to the data structure definition. We always try to map the first extremal element to the kind of element that is used more. We can always reverse the order, so the cheaper one can be used primarily, and the more expensive one in situations when we need both types of extremal elements.

3.2 Importance of operations

Detecting the importance of a single data structure operation is an important problem, because a lot of programs will fall into a class where compile-time data structure selection is undecidable. The program in Figure 8 shows that the problem of compile-time deciding on the best data structure is impossible to solve.

In Figure 8, the best data structure is depending on the user input, which is not known at compile time. If we analyze it as is, it will decide based on the information, that every operation is used, so the final data structure will not be specific to the task, but just average at everything.

Code that is not totally dependent on the user input can also cause problems to analyze.

In Figure 9 we see a very costly instruction used only once (delete max d), and a few instructions (insert, search) run in a loop for a few million times. To anyone that knows program complexities it is obvious that this one heavy instruction does not affect the execution time of the whole program, yet the framework at the current state treats those instructions equally and will probably choose something like Red-Black Trees, to minimize the time of the delete max d, instead of ignoring its cost and using a very fast Hashable.

We can formally show, that the problem of choosing the fastest data structure is undecidable, by showing that it is not easier than the halting problem. In Figure 10 we run the delete max d only if a turing machine that we defined accepts an input, so we do not know if we should choose a data structure that implements delete max d fast, or just ignore it, because it will not ever be used because of the machine not stopping the computation.
```c
#include <ds.h>

int main(int argc, char **argv)
{
    char c;
    ds d = init_d();
    while(1)
    {
        scanf("%c", &c);
        switch(c){
            case 'i':
                insert_d(d, 5);
                break;
            case 'u':
                update_d(d, 5, 7);
                break;
            case 'd':
                delete_d(d, 5);
                break;
            case 'm':
                max_d(d);
                break;
            case 'x':
                exit(0);
                break;
            default:
                //every other possible operation
                break;
        }
    }
}
```

Figure 8: An example of the problem of choosing the fastest data structure for a program being undecidable, user interaction

```c
#include <ds.h>

int main(int argc, char **argv)
{
    ds d = init_d();

    for(int i = 0; i < 42042323; i++)
        insert_d(d, i);

    printf("DEBUG: %d", delete_max_d(d));

    for(int i = 0; i < 42042323; i++)
        search_d(d, i);
}
```

Figure 9: An example of the problem of choosing the fastest data structure for a program being hard to analyze, no user interaction

#include <ds.h>
#include "turing.h"

int main(int argc, char **argv)
{
    ds d = init_d();
    insert_d(d, 6);

    turing_machine t;
    /*
     * [...] definition of the turing machine here
     */

    if (turing_machine_accepts(t, 2342)) {
        printf("accepted: \%d", delete_max_d(d));
    }
}

Figure 10: An example formally showing the undecidability of the problem, no user interaction

3.2.1 Code pragmas

A possible solution to this problem is to let the programmer add code pragmas to his source code, so he decides how important an instruction is in relation to other instructions and then the framework makes use of those values in choosing the data structure.

#include <ds.h>

int main(int argc, char **argv)
{
    ds d = init_d();

    for(int i = 0; i < 42042323; i++)
        insert_d(d, i, DSNF_IMPORTANT);

    printf("DEBUG: \%d", delete_max_d(d, DSNF_IGNORE));

    for(int i = 0; i < 42042323; i++)
        search_d(d, i, DSNF_IMPORTANT);
}

Figure 11: Source code with pragmas that state the importance of the operation

In the example in Figure 11 the programmer adds weights to operations, assigning very low values for statements that are used rarely or for debug purposes, and very high values for crucial parts of
the program.

We would have to extend the API like in Figure 12.

```c
struct ds;
typedef struct ds *ds;
typedef weight int;

const DSINF_IMPORTANT = 100;
const DSINF_IGNORE = 0;

ds init_d();
void insert_dw(ds, dstype, weight);
void update_dw(ds, dstype, dstype, weight);
void delete_dw(ds, dstype, weight);
void delmax_dw(ds, weight);
void delmin_dw(ds, weight);
dstype search_dw(ds, dstype, weight);
dstype max_dw(ds, weight);
dstype min_dw(ds, weight);
```

Figure 12: dsinf API, using additional arguments for operation importance

This is not a perfect solution, because we still need the programmer to judge which operations should have high weights, but it is nice when a programmer wants to use it for debug purposes or otherwise tinker with it.

### 3.2.2 Choosing the best data structure with weights

We need to change the algorithm for choosing the best data structure, for it to handle weights to the operations in the source code. We change the type of the $DSU()$ operation from Equation 10 to:

$$DSU_w(p) :: [(VariableName, DataStructureOperation, Int)]$$  \hspace{1cm} (13)

This change introduces weights for data structure operations in program $p$. We can use the additional $Int$ field to store the specified weight of the operation. For operations that are used multiple times, we sum the weights and the resulting sum is the value for that operation.

The $DSU()$ definition needs a change from section 2.3:

- Every rule in $DSOPS(p)$ now adds a triple, where the additional argument is weight obtained by using a method from section 3.2.1 or section 3.2.3.
- The last step of grouping, which removed repeated elements, now sums up the weights of the same operation elements and substitutes it with a new element with the sum as its weight.
The change the definition of Equation 12 is needed. We could still simply compare data structure operations on their complexity, making that one point, and multiply that point by the weight of the operation for that data structure identity. That is not a very precise solution, let us consider an example in Figure 13.

```c
#include <ds.h>

const int n = 4096;
const int x = 10;

int main(int argc, char **argv)
{
    ds d = init_d();

    for(int i = 0; i < n; i++)
        insert_d(d, i, 0);

    delete_max_d(d, 1);
    delete_min_d(d, 1);

    for(int i = 0; i < x; i++)
        search_d(d, i, x);
}
```

Figure 13: Example for weighted data structure choice algorithm, using pragmas API; the weight is counted per call site, not per function call, so the weight will not be $x^2$, but $x$ for `search_d`.

Let us consider the class of programs, like in Figure 13, but with different $n$ and $x$ values. Assume for simplicity that we ignore the insertions and the complexity functions of `search_d`, `delete_max_d` and `delete_min_d` are $\log_2 n$, $\log_2 n$, $\log_2 n$ for Red Black Trees and 1, $n$ and $n$ for Hash tables respectively — we discard big-O constants. Then, the answer to the question which data structure is better for this case can be stated as the inequality:

$$x + 2n > (x + 2)\log_2(n)$$

The left side is the cost of $x$ searches ($x * 1$) and deleting the maximal ($n$) and the minimal ($n$) element on Hashtables. The right side is the cost of $x$ searches ($x * \log_2 n$) and deleting the maximal ($\log_2 n$) and the minimal ($\log_2 n$) element on Red Black Trees.

This plot in Figure 14 shows, where the better solution would be to use Red Black Trees (blue area), and where to use Hashtables (white area). The blue cone, although smaller, still covers a lot of cases. The earlier algorithm yields hashtables for virtually all examples here ($x > 2$), which means that the data structures chosen in the blue cone on the plot are not the best ones for the case.

A better approach would be to approximate the current element count $N$ and actually evaluate the complexity function on the element count, multiplied by the weight, then the sum of those would be our metric of profitability of a data structure implementation. That would more accurately describe.
DSCOST describes the cost to use the data structure \( d \) for \( dsu \) usages.

\[
DSCOST(d, dsu) = \sum_{u \in dsu} OPCOST(u, d)
\]  

(16)

\( OPCOST \) describes the cost of a single operation \( o \), with weight \( w \) and complexity function \( c \) for a single data structure usage \( u \).

\[
OPCOST(u, d) = c(N) \times w \text{ where } (o, c) \in d, u = (v, o, w)
\]  

(17)

This generalization invalidates the point of storing data structure complexities in a easily comparable
way. Now we can just use any functions, that are reasonable to compute in small time, or even more complex ones if not using any runtime solutions like in section 3.2.4. One could even try to approximate the time better than just in the big-O notation, so the actual constants would matter.

For user guided approaches, like in section 3.2.1, user would have to manually specify a global count approximation for the cost optimization.

### 3.2.3 Profile-guided optimization

Profile-guided optimization (PGO) is a compiler optimization technique in computer programming to improve program runtime performance. In contrast to traditional optimization techniques that solely use the source code, PGO uses the results of test runs of the instrumented program to optimize the final generated code [2].

Usually the technique is used for optimizing hot loops and if statements. The binary saves logs of itself working and which source lines are hit more, then a system-wide daemon can recompile the parts of the binary to make it faster for the common case.

If the user has test data he can run against his program, we can take advantage of that. First we choose the best data structure using an unmodified method and link some library to the executable. Of course this does not have to be the best data structure possible. Then the user can run the test suite with code coverage option, like `gcov` in GCC, turned on in the compiler. This generates a file like the one shown in Figure 15.

Then the user can pass the `gcov` generated files and the source code to the framework again. The framework can extract line hits from the `gcov` files on the data structure operations and set weights on the operations according to the extracted data and then use the choosing algorithm for operations with weights from section 3.2.2.

In this method we can approximate the element count $N$ based on our coverage data, counting all insert and delete operations.

It is a better solution than letting the user set the weights himself, but we have to keep in mind that the data the inference is based upon comes from tests and there is no guarantee the real world data will match the test data.

### 3.2.4 Transforming data structures on-line

Another technique known in compilers we might use is called Just-In-Time Compilation (JIT). JIT, also known as dynamic translation, is a method to improve the runtime performance of computer programs based on byte code (virtual machine code). Since byte code is interpreted it executes more slowly than compiled machine code, unless it is actually compiled to machine code, which could be performed before the execution — making the program loading slow — or during the execution. In this latter case, which is the basis for JIT compilation, the program is stored in memory as byte code, but the code segment currently running is preparatively compiled to physical machine code in order to run faster [3].

This technique is, as PGO (section 3.2.3), used mostly for peephole optimizations, which means we always look at a small part of the code, like a hot `if` statement or a loop. With PGO we gathered
typedef int dstype;
#include "ds.h"

int main(int argc, const char *argv[]) {

ds d = init_d();

for(int i = 0; i < 402341; i++)
    insert_d(d, i);

search_d(d, 4);
}

---
0: Source: gcov.c
0: Graph: gcov.gcno
0: Data: gcov.gceda
0: Runs: 1
0: Programs: 1
1: typedef int dstype;
2: #include "ds.h"
3:
4: int main(int argc, const char *argv[]) {
5: {
6:
7:     ds d = init_d();
8:
9:     for(int i = 0; i < 402341; i++)
10:         insert_d(d, i);
11:
12:     search_d(d, 4);
13: }

Figure 15: Example of a source code file and the .gcov file, generated by running the compiled program

23
information over time and changed the binary, JIT does basically the same thing, only works at runtime. Usually we implement some proxy abstraction, that decides if, at the current time, it is profitable to compile the executed part to assembly instead of interpreting it.

Instead of compiling a bytecode to assembly, we can use a proxy to count the specific operations on our data structure, using the counts of usage as the new weights in the data structure inference algorithm in section 3.2.2. We substitute the data structure with a faster one, if the program would obviously benefit time-wise from the substitution. The new implementation could of course be compiled from a bytecode (e.g. LLVM IR) to assembly just like the standard JIT approach does it.

The element count \(N\) can be computed precisely, because we proxy all the data structure operations, so we just keep a counter.

The heuristics for detection of a good moment for the substitution should be stronger and more careful than in the standard JIT way. Compiling a part of code to assembly, in most cases, is not as costly as rebuilding a data structure, because unused code will not be run and a wrong data structure can slow down the whole program quite a bit. Building a sensible set of heuristics is very hard even for the standard JIT, e.g. the PyPy project tried a lot of different JIT approaches, before finding the one that is working well [8]. Depending on the heuristics here, this may be the most beneficial option for a program, or a big performance hit.

### 3.3 Different element types

Currently the framework works only for integer elements. We could extend it to every primitive type in C, but it would require some changes. Some data structures require the type to be comparable, which would not be a big problem, because there is a comparison semantics defined on those types. There is also a hash function needed, because some data structures, like a hashtable, need to compute hash values to work. We can just use the binary representation of other types, and use it as an integer for the hash function. This whole modification does not need any user interaction.

There is a bigger problem with composite types. Comparing pointers is not obvious. You may want to compare addresses or the contents under that address, depending if you want object or structural equality. The same problem appears in hash function arguments. There is also a problem with possible memory leaks. You can pass a pointer to a string to the data structure and lose the pointer in the program, then when the structure deletes the pointer and the string stays in the memory to the program’s end. Adding this to the framework would require passing the comparison function, hash function and some kind of destructor function to the data structure, or possibly use some reference counting system. It would be very technical and would be beyond the scope of this thesis. With an array or a big struct passed to the framework, there is a problem if we want to share the data structure and just copy the pointer, copying the whole thing may be a bad idea, because it can have quite a lot of data inside.

#### 3.3.1 Linked data structures

When we want to store structs like Figure 16 in a data structure, the framework, at the moment, could generate some data structure for this kind of structs. The operation on the data structure would look like in Figure 17.
struct person {
    int height;
    int weight;
    int age
    char name[128];
};

int cmp_height(struct person p1, struct person p2) {
    return p1.height - p2.height;
}

int cmp_weight(struct person p1, struct person p2) {
    return p1.weight - p2.weight;
}

Figure 16: An example record that we want to store in a data structure

#include <ds.h>
#include "person.h"

int main(int argc, char **argv) {

    struct person p;
    p.height = 143;
    p.weight = 213;
    p.age = 23;
    p.name = "James_Pearseed"

    ds d = ds_init(cmp_height);
    insert_d(d, p);
}

Figure 17: Use of the dsinf calls, to perform some data structure operations with the records; not very comfortable to use

In Figure 17 a comparison function is used; it compares the height. We can use any function that compares a coordinate or a combination of coordinates, but only one order of the elements is available per data structure. This can also be achieved without using structs at all, like in Figure 18.

In Figure 18 we dereference structs as arrays of chars. To get to any field, we just take a pointer to a field in the array and cast it to the right type. So it does not really give us any more expressiveness. It would be nice if our program enabled things like comparing structs by more than one condition in one program. We would want to be able to use operations like in Figure 19.
int main(int argc, char **argv) {
    char p[sizeof(struct person)];
    ...
    ds d = dsinit(cmp_height);
    insert_d(d, p);
}

Figure 18: Encoding of the problem of having one order data structure on records in C, into code using pointers

enum person_orders {
    height_order = 0,
    weight_order,
    person_orders_count
};

int main(int argc, const char *argv[]) {
    ds d = init_d(person_orders_count, cmp_height, cmp_weight);
    ...
    search_d(d, 4, height_order);
    delete_max_d(d, weight_order);
}

Figure 19: Examples of the new dsinf API that takes different orders on structs into account

We change the current API of dsinf (Figure 20) to enable passing an arbitrary number of comparison functions defining orders on the data to the constructor init_d. We use the va_args mechanism in C, which is why we need the order count as the first argument, because the va_arg macro needs the first argument of a known type, to start iterating over the rest of the arguments. Next arguments are function pointers of type int(void*, void*).

ds init_d(int order_count, ...);

Figure 20: Change in the API, enabling us to define multiple orders on data

To achieve this, we need to change the way we choose structures for record types. We will check all the data structure tuples, of length of the order count, in the cross-product of all data structures. The result will be a linked data structure, i.e. a number of data structures, in which elements have pointers to elements in other data structures, representing the same element value — like on Figure 21. For example, if we consider the code from the earlier example in Figure 19, the result may be a (Red-Black-Tree, Hashable) pair, where the RBT is for the weight order, and the hashtable
Figure 21: Visualization of a linked data structure: elements from a balanced tree, representing one order of data, connected with the elements of a linked list, representing the second order of data.

In the choice algorithm we make distinctions between operations that are order specific and those that are order independent operations. An example of an order specific operation is $\text{max}_d$, which finds an element that is maximal in a given order; an example of an order independent operation is $\text{insert}_d$, which does not take the order as an argument, as it has to perform an operation on all the linked data structures.

Sometimes order specific operations, aside from performing an operation on the data structure related to their order, have to update the state in other data structures, e.g. $\text{delete}_d$, after firing, has to run a $\text{delete}_d$ operation on the element on the other data structures responsible for different orders. It does not have to be the same operation. For example, $\text{delete\_max}_d$ does not have to call $\text{delete\_max}_d$, because it already has the data structure element pointer, and can just call $\text{delete}_d$ — we call the operation that will have the desired effect with the smallest complexity cost. Calling $\text{delete\_max}_d$ would not be correct for this example, because it would delete the maximal element from another order.

We define a function:

$$\text{RELATED} :: \text{DataStructureOperation} \rightarrow \text{DataStructureOperation}$$

(18)
For read-only general operations the function is undefined, as we have an edge case for that in Equation 35:

\[
\text{RELATED}(\text{Size}) = \text{undefined} \\
\text{RELATED}(\text{AnyElement}) = \text{undefined}
\]  

For read-write general operations the function is defined as:

\[
\text{RELATED}(\text{Insert}) = \text{Insert} \\
\text{RELATED}(\text{Delete}) = \text{Delete}
\]

For read-only order specific operations we use \textit{Nop}, an operation that does nothing:

\[
\text{RELATED}(\text{FindMax}) = \textit{Nop} \\
\text{RELATED}(\text{FindMin}) = \textit{Nop} \\
\text{RELATED}(\text{Find}) = \textit{Nop} \\
\text{RELATED}(\text{Successor}) = \textit{Nop} \\
\text{RELATED}(\text{Predecessor}) = \textit{Nop}
\]

For read-write order specific operations the function is defined as:

\[
\text{RELATED}(\text{DeleteMax}) = \text{Delete} \\
\text{RELATED}(\text{DeleteMin}) = \text{Delete}
\]

For every supported operation one has to decide if it is read-only or read-write, is it order specific or independent and define what is the related operation for it.

To create an ordering of linked data structures we modify Equation 15 as follows:

\[
ld_1 <_{dsu} ld_2 \iff \text{DSCOST}(ld_1, dsu) \leq \text{DSCOST}(ld_2, dsu)
\]  

Variables \(ld_1\) and \(ld_2\) are linked data structures consisting of \(m\) linked simple structures.

\[
ld_1 = d_{1,1}, d_{1,2}, ..., d_{1,m} \\
ld_2 = d_{2,1}, d_{2,2}, ..., d_{2,m}
\]

\textit{DSCOST} describes the cost of the linked data structure \(ld\) for usages \(dsu\).
\[ DSCOST(ld, dsu) = \sum_{u \in dsu} OPCOST(u, ld) \]  

\( OPCOST \) describes the cost of a single usage \( u \) of the linked data structure \( ld \). This has two cases because for read-only order-independent operations we can use the data structure that performs the operation the fastest. The other case is the sum of the main operation on the order specific data structure plus all cost of all the related operations on all the other data structures.

\[
OPCOST(u, ld) = \begin{cases} 
\min_{d \in ld} SOPCOST(u, d) & \text{if } ROOI(u) \\
\sum_{d \in ld} SOPCOST(u, d) + RELCOST(u, d, ld) & \text{otherwise}
\end{cases}
\]  

\( ROOI \) is a predicate that is true only for read-only order independent operations:

\[ ROOI(u) = True \text{ iff } u = (v, o, w) \land o \text{ read-only order independent operation} \]  

\( SOPCOST \) describes the cost of a single usage \( u \) on one of the substructures \( d \) related to the ordering the operation uses.

\[ SOPCOST(u, d) = c(N) \times w \text{ where } (o, c) \in d, u = (v, o, w) \]  

\( RELCOST \) describes the cost of a single usage \( u \) on all the substructures other than \( d \). It uses the \( ROPCOST \) cost for the operations, as we perform the related operation of the operation \( o \) on all the other structures.

\[ RELCOST(u, d, ld) = \sum_{d' \in ld, d \neq d'} ROPCOST(u, d') \]  

\[ ROPCOST(u, d) = c(N) \times w \text{ where } (RELATED(o), c) \in d, u = (v, o, w) \]  

### 3.4 Upper bound on the element count

When analyzing the input program, we can try to gather whether the maximum element count in the data structure is bounded by some constant. If we can obtain this information (in general it is of course undecidable), we can use it to enhance the data structure generated for the program.

When the element count in the data structure is potentially infinite, we have to create an implementation of the structure that allocates memory lazily, when it needs the space for new elements. We can imagine that a structure can allocate one chunk on every insertion and free a chunk on every deletion, or even allocate chunks twice as big and amortize the number of allocations to \( O(\log n) \).
```c
#include <stdio.h>

int main(int argc, char **argv)
{
    ds d = ds_init();

    for (int i = 0; i < 1024; i++)
        insert_d(d, f(i));

    g();
    delete_d(d, 1337);
    //part of the code not inserting any more elements

    return 0;
}
```

Figure 22: An example showing a situation where beneficial would be to create a statically allocated data structure instead of a dynamically allocated, because of statically known element count.

When we have the information about the element count, we can allocate the whole static buffer for the data structure, and this removes the whole cost of allocation during insertions and deletions. The example in Figure 22 shows such a program.

In Figure 22 after the first loop, there are no more insertions into the data structure, so if we can detect it during our analysis, then we can create a statically allocated version of the data structure.

### 3.5 Minimal element count threshold

It is worth noticing that we compare only the asymptotic complexity of data structures. Some awfully complicated structures can have good asymptotic results, but the constant is quite high. We can avoid this problem by setting a threshold for each structure, what is the smallest number of elements to use this data structure.

Another problem arises – how to know at compile time, how many elements a data structure will have at runtime. We can ask the user to explicitly specify the number during compilation or we can try to detect how big the declared data is, with some kind of constant folding analysis, that checks if the insertion to the data structure is in a loop that is run more times than the threshold. This is a fragile solution, because few programs are easy to analyze this way.

Another solution is runtime tallying the size, and using the method of transforming data structures on-line described in section 3.2.4. When we hit a threshold for a data structure with a bigger constant, but overall better fitting to our task, we transform the old one into this one.

In Figure 23 we see a code example that would yield some kind of a priority queue in our framework,
#include <stdio.h>

int main(int argc, char **argv)
{
    ds d = ds_init();

    for (int i = 0; i < 100; i++)
        insert_d(d, f(i));

    delete_d(d, 1337);

    for (int i = 0; i < 10000; i++)
        insert_d(d, f(i));

    delete_max_d(d);

    return 0;
}

Figure 23: An example showing a situation where beneficial would be to transform a data structure into a more complex one, because the number of elements is sufficiently big, so the whole transformation is profitable although it only has a few elements, and some of the fastest priority queues are pretty heavy-weight and it would be better just to use some simpler data structure, because the setup cost is not amortized by the operations.

3.6 Generic data structure modifications

When searching for the most apt data structure, we need to have some kind of data structure database, where we keep all the structure metadata and implementations for the framework to use. Ideally there would be a lot of different data structures there.

When implementing a data structure, one could easily modify the implementation to maybe match some rare specific task that is not needed in most cases. It would be wasteful to keep a copy of the data structure for each small combination of those modifications if we want to return a data structure tailored perfectly to the task. Of course some of the modifications are very data structure specific, so a generalization is not possible, but for some cases, we can extract a piece of code, like a wrapper for a data structure that modifies its behavior in some specific way (a similar pattern was used in [6]).
3.6.1 Extremal element cache

If we want to be able to lookup extremal elements in the data structure, we do not really have to know the implementation specifics of the data structure. We only need to intercept the calls which insert and delete elements. When we insert an element, we compare it to a cache variable, which keeps the biggest element (if our extremal element is the maximal element) known to be in the data structure.

But this approach also has some drawbacks – if we delete an element, and it occurs that the deleted element is the current maximum, we have to find the new maximal element from the rest of elements, which can be asymptotically more costly than the delete itself.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Complexity</th>
<th>Complexity with cache</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insert</td>
<td>$O(\log n)$</td>
<td>$O(log \ n)$</td>
</tr>
<tr>
<td>Update</td>
<td>$O(\log n)$</td>
<td>$O(log \ n)$</td>
</tr>
<tr>
<td>Delete</td>
<td>$O(\log n)$</td>
<td>$O(log \ n)$</td>
</tr>
<tr>
<td>Find Max</td>
<td>$O(\log n)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>Delete Max</td>
<td>$O(\log n)$</td>
<td>$O(log \ n)$</td>
</tr>
<tr>
<td>Search</td>
<td>$O(\log n)$</td>
<td>$O(log \ n)$</td>
</tr>
</tbody>
</table>

Table 2: A table showing the cost changes of Red-Black trees; complexities without and with the extremal element cache

<table>
<thead>
<tr>
<th>Operation</th>
<th>Complexity</th>
<th>Complexity with cache</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insert</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>Update</td>
<td>$O(1)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>Delete</td>
<td>$O(1)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>Find Max</td>
<td>$O(n)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>Delete Max</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>Search</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
</tbody>
</table>

Table 3: A table showing the cost changes of Hashables; complexities without and with the extremal element cache

Figure 24 shows an example implementation. We should note that a lot of data structures already implement this pattern [4].

3.6.2 Linked elements

When we have a structure that keeps some data, it usually takes $O(n)$ or $O(\log n)$ operations to find a predecessor or a successor. To improve this we can create an overlay with a doubly linked list on the elements, so our predecessor/successor lookups are $O(1)$. The drawback here is that the cost of insert is increased by searching the successor to link it appropriately. The cost of update is also affected. The delete cost stays the same, because we can wire the elements in $O(1)$, just like the removal of an element in a double linked list.
```c
struct ds_new {
    ds old;
    int max_elem;
}

void insert_d(ds_new *d, int elem){
    if (elem > d->max_elem)
        d->max_elem = elem;
    old_insert_d(d->old, elem);
}

void delete_d(ds_new *d, int elem){
    old_delete_d(d->old, elem)
    if (elem == d->max_elem)
        d->max_elem = old_max_d(d->old);
}

void update_d(ds_new *d, int old, int new){
    if (new > d->max_elem) {
        d->max_elem = new;
    }
    old_update_d(d->old, old, new);
}

int max(ds_new *d, int elem){
    return d->max_elem;
}

void delete_max_d(ds_new *d){
    old_delete_max_d(d->old)
    d->max_elem = old_max_d(d->old);
}
```

Figure 24: An example implementation of the extremal element cache data structure wrapper
4 Future work

It may be a good idea to rewrite the framework to analyze the LLVM\cite{5} intermediate representation internally, instead of the C language abstract syntax tree. This could yield better results because of the optimizations that clang\cite{7} can apply to the initial code, like removing unnecessary calls to data structure operations, so that the framework can infer the data structure based on the optimized code. LLVM also enables a link-time optimization, so after linking the chosen data structure to a program, another optimization step is performed. Also, analyzing LLVM IR is easier than analyzing the C language.

The framework can be easily generalized to substitute data structure comparison functions, so one could write a function that, as an example, takes I/O into account, or any combination of I/O and big-O complexity or even something more complicated.
A Implementation

A.1 Progress

The framework currently contains a full C language parser. There are a few caveats, e.g. inline assembly is not parsed, goto is not supported. It uses language-c Haskell library, so it parses common GCC extensions and works on most of the C code available today.

Unweighted inference capabilities are implemented and work on C programs including function calls from other functions. Examples of that are mentioned in section A.2.

The inferred data structures are selected from a set generated by using generic data structure modifications described in section 3.6.

The dsinf binary can work as a compile driver. Any developer can add his own data structure implementation, as long as it conforms to the declared API. The program can automatically analyze, compile and link the inferred data structure, so there is no work left to be done.

A.2 Compilation

To install dsinf you need cabal — a haskell build manager. The preferred way of obtaining cabal is by downloading the Haskell Platform (http://www.haskell.org/platform/) — it has cabal bundled inside. You can also use a package manager of your favorite Linux distribution.

After installing cabal, clone the repository and build by running the following commands:

git clone https://github.com/alistra/data-structure-inferrer
cd data-structure-inferrer/
made dsinf

This will clone the dsinf git repository and build the package using a cabal sandbox. This can take a little while, but all the dependencies are installed locally and will not affect your haskell setup in the system. This will install the dsinf binary in your current working directory.

A.3 Usage

The program has a few modes of working:

- Recommendation mode - the framework analyzes the input files and the data structure operations in them, then outputs the recommended data structure on standard output
- Advice mode - the framework checks if any small change in the data structure operations would give the program a speed boost
- Compile mode - works as the Recommendation mode, but instead of printing the data structure to standard output, it compiles the files and links it with the chosen structure implementation.

We can trigger the modes using appropriate flags:
dsinf [-OPT1 [VAL1] [-OPT2 [VAL2] [...]] [-- [CCOPTS]]
-o file --output-file Output file
-r --recommend Give recommendations about the data structure in the supplied code (default)
-a --advice Give advice about the data structure in the supplied code
-c --compile Compile the code with recommended structure linked
-i --inline Inline the code implementing the recommended structure in the supplied code
-v --verbose Enable verbose messages
-h --help Show help

CCOPTS are passed directly to the compiler

There are test files available so that one can check if everything is works alright.

% ./dsinf C/tests/d1_delmax_d2_max.c

C/tests/d1_delmax_d2_max.c:
The recommended structure for:
d1 in main
is:
Red-Black Trees
The recommended structure for:
d2 in main
is:
Hashtable with extreme element caching

A.4 System Architecture

A.4.1 Core - Recommend.hs

The core part of the recommendation engine is actually very simple:

recommendDS :: [OperationName] -> IO Structure
recommendDS opns = do
  let sorted = reverse $ sortBy (\x y-> compareDS x y opns) allStructures
  let bestStructures = head $ groupBy (\x y -> compareDS x y opns == EQ) sorted
  ridx <- randomRIO (0, length bestStructures - 1)
  return $ bestStructures !! ridx

The argument opns :: [OperationName] of this function is a list of data structure operations performed on a persistent data structure identity in the analyzed program. The allStructures contains all the data structure definitions available for the recommendation engine. More about allStructures is explained in section A.4.3. The compareDS function is an implementation of the ordering in Equation 12.

In these functions elements from allStructures are sorted using the compareDS order. Then we take the group of first elements that are equal within this ordering — the structures represent the fastest
structures for the persistent data structure identity. We return a random fastest data structure, so the program can return one data structure.

A.4.2 C Analyzer - C/Analyzer.hs, Analyzer.hs

This part of the code uses the language-c Haskell library to parse C code. It analyzes the functions in the source files and extracts the data and passes it to the Core part in an easily understandable format as DSUs like defined in section 2.3.

To run the whole analysis, we call the `analyzeC` function. It takes the file path as an argument.

```haskell
analyzeC :: FilePath -> IO [DSInfo]
```

The return value is a list of `DSInfos` which are simply lists of `DSUs` with all the names, which the particular persistent data structure identity can be called. This information is sufficient to pass it to the Core described in section A.4.1.

The `analyzeC` function traverses the whole abstract syntax tree of the program, which is obtained from the language-c parser. It recurses through C statements, expressions and other language parts as described in the C99 standard. It gathers the `DSUs` and groups them according to rules described in section 2.3.

A.4.3 Data structure definitions - AllStructures.hs

This file stores all structure definitions available to the recommendation engine. This is how an example definition looks like:

```haskell
-- | Linked list
ll :: Structure
ll = DS "Linked List"
    [ Op BoundByRef (LinLog 1 0, N),
      Op DecreaseValByRef (LinLog 0 0, N),
      Op DeleteByRef (LinLog 0 0, N),
      Op DeleteExtremalVal (LinLog 1 0, N),
      Op Difference (LinLog 2 0, N),
      Op Empty (LinLog 0 0, N),
      Op ExtremalVal (LinLog 1 0, N),
      Op FindByVal (LinLog 1 0, N),
      Op InsertVal (LinLog 0 0, N),
      Op Intersection (LinLog 2 0, N),
      Op Map (LinLog 1 0, N),
      Op Size (LinLog 0 0, N),
      Op SymDifference (LinLog 2 0, N),
      Op Union (LinLog 0 0, N),
      Op UpdateByRef (LinLog 0 0, N) ]
```
The *LinLog* constructor stores the asymptotic complexity as described in Equation 2. The *N* part is a marker, signifying that this is a normal complexity, as opposed to an expected or amortized complexity.

This part also implements the technique described in section 3.6 that provides generic data structure modifications, e.g. extremal element caching:

```haskell
-- | A function that adds an extremal element cache to a data structure
extremalElemCache :: Structure -> Structure
extremalElemCache (DS name ops) = DS (name ++ " with extreme element caching") ops' where
  extVal = fromJust $ find (\dsop -> getOpName dsop == ExtremalVal) ops
  delByRef = fromJust $ find (\dsop -> getOpName dsop == DeleteByRef) ops
  ops' = 
    [Op DeleteByRef (max (getComplexity extVal) (getComplexity delByRef)),
     Op ExtremalVal (LinLog 0 0, N)] ++
    filter (\dsop -> getOpName dsop `notElem` [ExtremalVal, DeleteByRef]) ops
```

The *extremalElemCache* function takes a data structure definition as an input and returns a data structure definition with modified costs according to the generic modification. As adding an extremal element cache will speed up (to constant time) the lookup of an extremal value, the resulting data structure definition has the cost of *ExtremalVal* as *(LinLog 0 0, N)* — normal constant time. The tradeoff in this case is that, the *DeleteByRef* operation has to update the cache, if it removed the extremal value. This adds the cost of the old *ExtremalVal* (the new is constant) to the cost of deletion.
References

[6] Purely Functional Data Structures - Chris Okasaki
[9] Cocoa Core Competencies - Class Clusters -