Markus Pfeiffer
Email: markus.pfeiffer@st-andrews.ac.uk
Homepage: http://www.morphism.de/~markusp
Address: School of Computer Science
        University of St Andrews
        Jack Cole Building, North Haugh
        St Andrews, Fife, KY16 9SX
        United Kingdom

Max Horn
Email: max.horn@math.uni-giessen.de
Homepage: http://www.quendi.de/math
Address: AG Algebra
        Mathematisches Institut
        Justus-Liebig-Universität Gießen
        Arndtstraße 2
        35392 Gießen
        Germany

Christopher Jefferson
Email: caj21@st-andrews.ac.uk
Homepage: http://caj.host.cs.st-andrews.ac.uk/
Address: School of Computer Science
        University of St Andrews
        Jack Cole Building, North Haugh
        St Andrews, Fife, KY16 9SX
        United Kingdom

Steve Linton
Email: steve.linton@st-andrews.ac.uk
Homepage: http://s14.host.cs.st-andrews.ac.uk/
Address: School of Computer Science
        University of St Andrews
        Jack Cole Building, North Haugh
        St Andrews, Fife, KY16 9SX
        United Kingdom
Copyright

© 2015-18 by Chris Jefferson, Steve Linton, Markus Pfeiffer, Max Horn, Reimer Behrends and others

datastructures package is free software; you can redistribute it and/or modify it under the terms of the
GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or
(at your option) any later version.

Acknowledgements

We appreciate very much all past and future comments, suggestions and contributions to this package and its
documentation provided by GAP users and developers.
# Contents

1 Introduction  
   1.1 Purpose and goals of this package  
   1.2 Overview over this manual  
   1.3 Feedback  

2 Installation  
   2.1 Building the Kernel Module  
   2.2 Building the Documentation  

3 Heaps  
   3.1 Introduction  
   3.2 API  
   3.3 Binary Heaps  
   3.4 Pairing Heaps  
   3.5 Declarations  
   3.6 Implementation  

4 Queues and Deques  
   4.1 API  
   4.2 Deques implemented using plain lists  

5 Union-Find  
   5.1 Introduction  
   5.2 API  

6 Hash Functions  
   6.1 Introduction  
   6.2 Hash Functions for Basic Types  
   6.3 Hash Functions for Permutation Groups  

7 Hashmaps  
   7.1 API  

8 Hashsets  
   8.1 API  

9 Strings  
   9.1 APIs  
   9.2 Implementation  

10 Arrays  
   10.1 APIs  
   10.2 Implementation  

11 Pairs  
   11.1 APIs  
   11.2 Implementation  

12 Records  
   12.1 APIs  
   12.2 Implementation  

13 Modules  
   13.1 APIs  
   13.2 Implementation  

14 Streams  
   14.1 APIs  
   14.2 Implementation  

15 Custom Constructs  
   15.1 APIs  
   15.2 Implementation  

16 Database Interfaces  
   16.1 APIs  
   16.2 Implementation  

17 Exceptions  
   17.1 APIs  
   17.2 Implementation  

18 Errors  
   18.1 APIs  
   18.2 Implementation  

19 Testing  
   19.1 APIs  
   19.2 Implementation  

20 Internals  
   20.1 APIs  
   20.2 Implementation  

21 Preferences  
   21.1 APIs  
   21.2 Implementation  

22 Configuration  
   22.1 APIs  
   22.2 Implementation  

23 Source Code  
   23.1 APIs  
   23.2 Implementation  

24 Authors  
   24.1 APIs  
   24.2 Implementation  

25 License  
   25.1 APIs  
   25.2 Implementation  

26 Index  

9 Memoisation .......................................................... 23
  9.1 Memoisation with HashMap .................................. 23

10 Ordered Set Datastructures ...................................... 24
  10.1 Usage .................................................................. 24
  10.2 API ................................................................... 25
  10.3 Default methods ................................................... 27

11 Stacks ................................................................. 29
  11.1 API ................................................................... 29

References ................................................................. 31

Index ....................................................................... 32
Chapter 1

Introduction

1.1 Purpose and goals of this package

The datastructures package for GAP has two main goals:

- Provide abstract interfaces for commonly used datastructures
- Provide good low-level implementations for these datastructures

datastructures requires building of a kernel module for GAP to function, please refer to Chapter 2 for details; the package is not automatically loaded by GAP after it has been installed. You must load the package with `LoadPackage("datastructures");` before its functions become available.

1.2 Overview over this manual

Chapter 2 describes the installation of this package. The remaining chapters describe the available datastructures in this package with a definition of the supported API and details about provided implementations.

1.3 Feedback

For bug reports, feature requests and suggestions, please use our issue tracker.
Chapter 2

Installation

datastructures does not work without compiling its kernel module, and is not loaded by GAP by default. To load the package run LoadPackage("datastructures"); at the GAP prompt.

2.1 Building the Kernel Module

To build the kernel module, you will need

- a C compiler, e.g. GCC or Clang
- GNU Make

To install a released version of this package, extract the package’s archive file into GAP’s pkg folder.

To install the current development version of this package, obtain the most recent code from GitHub

    git clone https://github.com/gap-packages/datastructures

To build the kernel module then run the following commands in the package’s directory.

    ./configure
    make

2.2 Building the Documentation

To build the package documentation, run the following command in the package’s directory

    gap makedoc.g
Chapter 3

Heaps

3.1 Introduction

A heap is a tree datastructure such that for any child $C$ of a node $N$ it holds that $C \leq N$, according to some ordering relation $\leq$.

The fundamental operations for heaps are Construction, Pushing data onto the heap, Peeking at the topmost item, and Popping the topmost item off of the heap.

For a good heap implementation these basic operations should not exceed $O(\log n)$ in runtime where $n$ is the number of items on the heap.

We currently provide two types of heaps: Binary Heaps 3.3 and Pairing Heaps 3.4.

The following code shows how to use a binary heap.

```
gap> h := BinaryHeap();
<binary heap with 0 entries>
gap> Push(h, 5);
gap> Push(h, -10);
gap> Peek(h);  
5  
gap> Pop(h);
5  
gap> Peek(h);  
-10
```

The following code shows how to use a pairing heap.

```
gap> h := PairingHeap( {x,y} -> x.rank > y.rank );
<pairing heap with 0 entries>
gap> Push(h, rec( rank := 5 ));
gap> Push(h, rec( rank := 7 ));
gap> Push(h, rec( rank := -15 ));
gap> h;
<pairing heap with 3 entries>
gap> Peek(h);
rec( rank := -15 )
gap> Pop(h);
rec( rank := -15 )
```
3.2 API

For the purposes of the datastructures, we provide a category `IsHeap` (??). Every implementation of a heap in the category `IsHeap` (??) must follow the API described in this section.

3.2.1 `IsHeap` (for `IsObject`)

▶ `IsHeap(arg)` (filter)

   **Returns:** true or false

   The category of heaps. Every object in this category promises to support the API described in this section.

3.2.2 `Heap`

▶ `Heap(arg)` (function)

   Wrapper function around constructors

3.2.3 `NewHeap` (for `IsHeap`, `IsObject`, `IsObject`)

▶ `NewHeap([filter, func, data])` (operation)

   **Returns:** a heap

   Construct a new heap

3.2.4 `Push` (for `IsHeap`, `IsObject`)

▶ `Push(heap, object)` (operation)

   Puts the object `object` a new object onto `heap`.

3.2.5 `Peek` (for `IsHeap`)

▶ `Peek(heap)` (operation)

   Inspect the item at the top of `heap`.

3.2.6 `Pop` (for `IsHeap`)

▶ `Pop(heap)` (operation)

   **Returns:** an object

   Remove the top item from `heap` and return it.

3.2.7 `Merge` (for `IsHeap`, `IsHeap`)

▶ `Merge(heap1, heap2)` (operation)

   Merge two heaps (of the same type)

   Heaps also support `IsEmpty` (Reference: `IsEmpty`) and `Size` (Reference: `Size`)
3.3 Binary Heaps

A binary heap employs a binary tree as its underlying tree datastructure. The implementation of binary heaps in datastructures stores this tree in a flat array which makes it a very good and fast default choice for general purpose use. In particular, even though other heap implementations have better theoretical runtime bounds, well-tuned binary heaps outperform them in many applications.

For some reference see http://stackoverflow.com/questions/6531543

3.3.1 BinaryHeap

\[
\text{BinaryHeap}([\text{isLess}, \text{data}])
\]

Returns: A binary heap

Constructor for binary heaps. The optional argument \text{isLess} must be a binary function that performs comparison between two elements on the heap, and returns true if the first argument is less than the second, and false otherwise. Using the optional argument \text{data} the user can give a collection of initial values that are pushed on the stack after construction.

3.4 Pairing Heaps

A pairing heap is a heap datastructure with a very simple implementation in terms of GAP lists. Push and Peek have \(O(1)\) complexity, and Pop has an amortized amortised \(O(\log n)\), where \(n\) is the number of items on the heap.

For a reference see [FSST86].

3.4.1 PairingHeap

\[
\text{PairingHeap}([\text{isLess}, \text{data}])
\]

Returns: A pairing heap

Constructor for pairing heaps. The optional argument \text{isLess} must be a binary function that performs comparison between two elements on the heap, and returns true if the first argument is less than the second, and false otherwise. Using the optional argument \text{data} the user can give a collection of initial values that are pushed on the stack after construction.

3.5 Declarations

3.5.1 IsBinaryHeapFlatRep (for IsHeap and IsPositionalObjectRep)

\[
\text{IsBinaryHeapFlatRep}(\text{arg})
\]

Returns: true or false

3.6 Implementation

3.6.1 IsPairingHeapFlatRep (for IsHeap and IsPositionalObjectRep)

\[
\text{IsPairingHeapFlatRep}(\text{arg})
\]

Returns: true or false
Chapter 4

Queues and Deques

4.1 API

4.1.1 IsQueue (for IsObject)

\[\text{IsQueue} \text{ (arg)}\]

\textbf{Returns: true or false}

The category of queues.

4.1.2 IsDeque (for IsObject)

\[\text{IsDeque} \text{ (arg)}\]

\textbf{Returns: true or false}

The category of deques.

4.1.3 PushBack (for IsDeque, IsObject)

\[\text{PushBack} \text{ (deque, object)}\]

Add \textit{object} to the back of \textit{deque}.

4.1.4 PushFront (for IsDeque, IsObject)

\[\text{PushFront} \text{ (deque, object)}\]

Add \textit{object} to the front of \textit{deque}.

4.1.5 PopBack (for IsDeque)

\[\text{PopBack} \text{ (deque)}\]

\textbf{Returns: object}

Remove an element from the back of \textit{deque} and return it.
4.1.6 PopFront (for IsDeque)

\[ \text{PopFront}(\text{deque}) \]

\textbf{Returns:} object

Remove an element from the front of \textit{deque} and return it.
For queues, this is just an alias for PushBack

4.1.7 Enqueue (for IsQueue, IsObject)

\[ \text{Enqueue}(\text{queue, object}) \]

Add \textit{object} to \textit{queue}.

4.1.8 Dequeue (for IsQueue, IsObject)

\[ \text{Dequeue}(\text{queue}) \]

\textbf{Returns:} object

Remove an object from the front of \textit{queue} and return it.

4.1.9 Capacity (for IsQueue)

\[ \text{Capacity}(\text{arg}) \]

Allocated storage capacity of \textit{queue}.

4.1.10 Capacity (for IsDeque)

\[ \text{Capacity}(\text{arg}) \]

Allocated storage capacity of \textit{deque}.

4.1.11 Length (for IsQueue)

\[ \text{Length}(\text{arg}) \]

Number of elements in \textit{queue}.

4.1.12 Length (for IsDeque)

\[ \text{Length}(\text{arg}) \]

Number of elements in \textit{deque}.

4.2 Deques implemented using plain lists

datastructures implements deques using a circular buffer stored in a GAP a plain list, wrapped in a positional object (IsPositionalObjectRep (IsPositionalObjectRep??)).

The five positions in such a deque \( Q \) have the following purpose
- Q! [3] - capacity, the allocated capacity in the deque
- Q! [4] - factor by which storage is increased if capacity is exceeded
- Q! [5] - GAP plain list with storage for capacity many entries

Global constants QHEAD, QTAIL, QCAPACITY, QFACTOR, and QDATA are bound to reflect the above. When a push fills the deque, its capacity is resized by a factor of QFACTOR using PlistDequeExpand. A new empty plist is allocated and all current entries of the deque are copied into the new plist with the head entry at index 1.

The deque is empty if and only if head = tail and the entry that head and tail point to in the storage list is unbound.

4.2.1 PlistDeque

\[ PlistDeque([capacity[, factor]]) \] (function)

Returns: a deque

Constructor for plist based deques. The optional argument capacity must be a positive integer and is the capacity of the created deque, and the optional argument factor must be a rational number greater than one which is the factor by which the storage of the deque is increased if it runs out of capacity when an object is put on the queue.

4.2.2 PlistDequePushFront

\[ PlistDequePushFront(deque, object) \] (function)

Push object to the front of deque.

4.2.3 PlistDequePushBack

\[ PlistDequePushBack(deque, object) \] (function)

Push object to the back of deque.

4.2.4 PlistDequePopFront

\[ PlistDequePopFront(deque) \] (function)

Returns: object or fail

Pop object from the front of deque and return it. If deque is empty, returns fail.

4.2.5 PlistDequePopBack

\[ PlistDequePopBack(deque) \] (function)

Returns: object or fail

Pop object from the back of deque and return it. If deque is empty, returns fail.
4.2.6 PlistDequePeekFront

➤ PlistDequePeekFront(deque)  (function)

Returns: object or fail
Returns the object at the front deque without removing it. If deque is empty, returns fail.

4.2.7 PlistDequePeekBack

➤ PlistDequePeekBack(deque)  (function)

Returns: object or fail
Returns the object at the back deque without removing it. If deque is empty, returns fail.

4.2.8 PlistDequeExpand

➤ PlistDequeExpand(deque)  (function)

Helper function to expand the capacity of deque by the configured factor.

Queues are linear data structure that allow adding elements at the end of the queue, and removing elements from the front. A deque is a double-ended queue; a linear data structure that allows access to objects at both ends.

The API that objects that lie in IsQueue (??) and IsDeque (??) must implement the API set out below.

datastructures provides
Chapter 5

Union-Find

5.1 Introduction

datastructures defines the interface for mutable data structures representing partitions of \([1..n]\), commonly known as union-find data structures. Key operations are Unite (5.2.5) which fuses two parts of a partition and Representative (5.2.4) which returns a canonical representative of the part containing a given point.

5.2 API

5.2.1 IsPartitionDS (for IsObject)

▷ IsPartitionDS(arg) 

Returns: true or false

Category of datastructures representing partitions. Equality is identity and family is ignored.

5.2.2 PartitionDS (for IsPartitionDS, IsPosInt)

▷ PartitionDS(filter, n) 

Family containing all partition data structures Returns the trivial partition of the set \([1..n]\).

5.2.3 PartitionDS (for IsPartitionDS, IsCyclotomicCollColl)

▷ PartitionDS(filter, partition) 

Returns the union find structure of partition.

5.2.4 Representative (for IsPartitionDS, IsPosInt)

▷ Representative(unionfind, k) 

Returns: a positive integer

Returns a canonical representative of the part of the partition that \(k\) is contained in.
5.2.5  Unite (for IsPartitionDS and IsMutable, IsPosInt, IsPosInt)

\[ \text{Unite}(\text{unionfind}, k1, k2) \]  
(operation)

Fuses the parts of the partition unionfind containing \( k1 \) and \( k2 \).

5.2.6  RootsIteratorOfPartitionDS (for IsPartitionDS)

\[ \text{RootsIteratorOfPartitionDS}(\text{unionfind}) \]  
(operation)

Returns: an iterator

Returns an iterator that runs through canonical representatives of parts of the partition unionfind.

5.2.7  NumberParts (for IsPartitionDS)

\[ \text{NumberParts}(\text{unionfind}) \]  
(attribute)

Returns: a positive integer

Returns the number of parts of the partition unionfind.

5.2.8  SizeUnderlyingSetDS (for IsPartitionDS)

\[ \text{SizeUnderlyingSetDS}(\text{unionfind}) \]  
(attribute)

Returns: a positive integer

Returns the size of the underlying set of the partition unionfind.

5.2.9  PartsOfPartitionDS (for IsPartitionDS)

\[ \text{PartsOfPartitionDS}(\text{unionfind}) \]  
(attribute)

Returns: a list of lists

Returns the partition unionfind as a list of lists.
Chapter 6

Hash Functions

6.1 Introduction

A hash function in datastructures is a function $H$ which maps a value $X$ to a small integer (where a small integer is an integer in the range $[-2^{28} \ldots 2^{28}-1]$ on a 32-bit system, and $[-2^{60} \ldots 2^{60}-1]$ on a 64-bit system), under the requirement that if $X = Y$, then $H(X) = H(Y)$.

A variety of hash functions is provided by datastructures, with different behaviours. A bad choice of hash function can lead to serious performance problems.

datastructures does not guarantee consistency of hash values across release or GAP sessions.

6.2 Hash Functions for Basic Types

6.2.1 HashBasic

$\text{\texttt{HashBasic(obj...)}}$

Returns: a small integer

Hashes any values built inductively from

- built-in types, namely integers, booleans, permutations, transformations, partial permutations, and

- constructors for lists and records.

This function is variadic, treating more than one argument as equivalent to a list containing the arguments, that is $\text{HashBasic(x,y,z)} = \text{HashBasic([x,y,z])}$.

6.3 Hash Functions for Permutation Groups

datastructures provides two hash functions for permutation groups; $\text{Hash\_PermGroup\_Fast (6.3.1)}$ is the faster one, with higher likelihood of collisions and $\text{Hash\_PermGroup\_Complete (6.3.2)}$ is slower but provides a lower likelihood of collisions.

6.3.1 Hash\_PermGroup\_Fast

$\text{\texttt{Hash\_PermGroup\_Fast(group)}}$

Returns: a small integer
Hash_PermGroup_Fast (6.3.1) is faster than Hash_PermGroup_Complete (6.3.2), but will return the same value for groups with the same size, orbits and degree of transitivity.

6.3.2 Hash_PermGroup_Complete

- **Returns:** a small integer

Hash_PermGroup_Complete (6.3.2) is slower than Hash_PermGroup_Fast (6.3.1), but is extremely unlikely to return the same hash for two different groups.
Chapter 7

Hashmaps

A hash map stores key-value pairs and allows efficient lookup of keys by using a hash function. datastructures currently provides a reference implementation of hashmaps using a hashtable stored in a plain GAP list.

7.1 API

7.1.1 IsHashMap (for IsObject and IsFinite)

\[ \text{IsHashMap} (\text{arg}) \]

Returns: true or false

Category of hash maps

7.1.2 HashMap

\[ \text{HashMap}(\text{[hashfunc, eqfunc, ]}[\text{capacity}]) \]

Create a new hash map. The optional argument hashfunc must be a hash- function, eqfunc must be a binary equality testing function that returns true if the two arguments are considered equal, and false if they are not. Refer to Chapter 6 about the requirements for hashfunctions and equality testers. The optional argument capacity determines the initial size of the hashmap.

7.1.3 Keys (for IsHashMap)

\[ \text{Keys}(\text{h}) \]

Returns: a list

Returns the list of keys of the hashmap h.

7.1.4 Values (for IsHashMap)

\[ \text{Values}(\text{h}) \]

Returns: a list

Returns the set of values stored in the hashmap h.
7.1.5 **KeyIterator (for IsHashMap)**

- **KeyIterator(h)**
  - **Returns:** an iterator
    - Returns an iterator for the keys stored in the hashmap $h$.

7.1.6 **ValueIterator (for IsHashMap)**

- **ValueIterator(h)**
  - **Returns:** an iterator
    - Returns an iterator for the values stored in the hashmap $h$.

7.1.7 **KeyValueIterator (for IsHashMap)**

- **KeyValueIterator(h)**
  - **Returns:** an iterator
    - Returns an iterator for key-value-pairs stored in the hashmap $h$.

7.1.8 **\[\] (for List-style access for hashmaps)**

- **List-style access for hashmaps**.

7.1.9 **\[\]:\= (for List-style access for hashmaps)**

- **List-style access for hashmaps**.

7.1.10 **\in (for IsObject, IsHashMapRep)**

- **\in(object, hashmap)**
  - **Test whether a key is stored in the hashmap.**

7.1.11 **IsBound\[\] (for)**

- **Test whether a key is stored in the hashmap.**

7.1.12 **Unbind\[\] (for)**

- **Delete a key from a hashmap.**

7.1.13 **Size (for IsHashMapRep)**

- **Size(hashmap)**
  - **Determine the number of keys stored in a hashmap.**
7.1.14  **IsEmpty (for IsHashMapRep)**

> **IsEmpty** *(object, hashmap)*  

Test whether a hashmap is empty.
Chapter 8

Hashsets

A hash set stores objects and allows efficient lookup whether an object is already a member of the set.

`datastructures` currently provides a reference implementation of hashsets using a hashtable stored in a plain GAP list.

8.1 API

8.1.1 IsHashSet (for IsObject and IsFinite)

```plaintext
▷ IsHashSet(arg)

Returns: true or false

Category of hashsets
```

8.1.2 HashSet

```plaintext
▷ HashSet([hashfunc[, eqfunc, ]]\[capacity])

Create a new hashset. The optional argument `hashfunc` must be a hash-function, `eqfunc` must be a binary equality testing function that returns `true` if the two arguments are considered equal, and `false` if they are not. Refer to Chapter 6 about the requirements for hashfunctions and equality testers. The optional argument `capacity` determines the initial size of the hashmap.
```

8.1.3 AddSet (for IsHashSetRep, IsObject)

```plaintext
▷ AddSet(hashset, obj)

Add `obj` to `hashset`.
```

8.1.4 \in (for IsObject, IsHashSetRep)

```plaintext
▷ \in(obj, hashset)

Test membership of `obj` in `hashset`
8.1.5 RemoveSet (for IsHashSetRep, IsObject)

\[ \text{RemoveSet(}hashset, \text{ obj)} \]  
Remove \text{ obj} from \text{ hashset}.

8.1.6 Size (for IsHashSetRep)

\[ \text{Size(}hashset) \]  
Return the size of a hashset Returns an integer

8.1.7 IsEmpty (for IsHashSetRep)

\[ \text{IsEmpty(}hashset) \]  
Returns: a boolean
Test a hashset for emptiness.

8.1.8 Set (for IsHashSetRep)

\[ \text{Set(}hashset) \]  
Returns: a set
Convert a hashset into a GAP set

8.1.9 AsSet (for IsHashSetRep)

\[ \text{AsSet(}hashset) \]  
Returns: an immutable set
Convert a hashset into a GAP set

8.1.10 Iterator (for IsHashSetRep)

\[ \text{Iterator(}set) \]  
Returns: an iterator
Create an iterator for the values contained in a hashset. Note that elements added to the hashset after the creation of an iterator are not guaranteed to be returned by that iterator.
Chapter 9

Memoisation

datastructures provides simple ways to cache return values of pure functions.

9.1 Memoisation with HashMap

9.1.1 MemoizeFunction

\[ \text{MemoizeFunction}\(\text{function}, \text{options}\) \]

Returns: A function

MemoizeFunction returns a function which behaves the same as function, except that it caches the return value of function. The cache can be flushed by calling FlushCaches (Reference: FlushCaches).

This function does not promise to never call function more than once for any input – values may be removed if the cache gets too large, or GAP chooses to flush all caches, or if multiple threads try to calculate the same value simultaneously.

The optional second argument is a record which provides a number of configuration options. The following options are supported.

flush (default true)

If this is true, the cache is emptied whenever FlushCaches (Reference: FlushCaches) is called.

contract (defaults to ReturnTrue (Reference: ReturnTrue))

A function that is called on the arguments given to function. If this function returns false, then errorHandler is called.

errorHandler (defaults to none)

A function to be called when an input that does not fulfil contract is passed to the cache.
Chapter 10

Ordered Set Datastructures

In this chapter we deal with datastructures designed to represent sets of objects which have an intrinsic ordering. Such datastructures should support fast (possibly amortised) $O(\log n)$ addition, deletion and membership test operations and allow efficient iteration through all the objects in the datastructure in the order determined by the given comparison function. Since they represent a set, adding an object equal to one already present has no effect.

We refer to these as ordered set datastructure because they differ from the GAP notion of a set in a number of ways:

- They all lie in a common family `OrderedSetDSFamily` and pay no attention to the families of the Objects stored in them.
- Equality of these structures is by identity, not equality of the represented set.
- The ordering of the objects in the set does not have to be default GAP ordering "less than", but is determined by the Attribute `LessFunction`.

Three implementations of ordered set data structures are currently included: skiplists, binary search trees and (as a specialisation of binary search trees) AVL trees. AVL trees seem to be the fastest in general, and memory usage is similar. More details to come.

10.1 Usage

```
gap> s := OrderedSetDS(IsSkipListRep, \{x,y\} -> String(x) < String(y));
<skiplist 0 entries>
gap> AddSet(s, 1);
gap> AddSet(s, 2);
gap> AddSet(s, 10);
gap> AddSet(s, \{1,2,3\});
gap> RemoveSet(s, \{1,2,3\});
1
gap> AsListSorted(s);
[ 1, 10, 2 ]
```

```
gap> b := OrderedSetDS(IsBinarySearchTreeRep, Primes);
<bst size 168>
gap> 91 in b;
```

Example
10.2 API

Every implementation of an ordered set datastructure must follow the API set out below

10.2.1 IsOrderedSetDS (for IsObject)

\[ \text{IsOrderedSetDS}(\text{arg}) \] (filter)

Returns: true or false
Category of ordered set.

10.2.2 IsStandardOrderedSetDS (for IsOrderedSetDS)

\[ \text{IsStandardOrderedSetDS}(\text{arg}) \] (filter)

Returns: true or false
Subcategory of ordered sets where the ordering is GAP’s default UNKNOWNEntity(leq)

10.2.3 OrderedSetDS (for IsOrderedSetDS, IsFunction, IsListOrCollection, IsRandomSource)

\[ \text{OrderedSetDS}(\text{filter[, } \text{lessThan[, } \text{initialEntries[, } \text{randomSource]}]]) \] (operation)

Returns: an ordered set datastructure
The family that contains all ordered set datastructures. Constructors for ordered sets
The argument \text{filter} is a filter that the resulting ordered set object will have.

The optional argument \text{lessThan} must be a binary function that returns true if its first argument is less than its second argument, and false otherwise. The default \text{lessThan} is GAP’s built in UNKNOWNEntity(leq).

The optional argument \text{initialEntries} gives a collection of elements that the ordered set is initialised with, and defaults to the empty set.

The optional argument \text{randomSource} is useful in a number of possible implementations that use randomised methods to achieve good amortised complexity with high probability and simple data structures. It defaults to the global Mersenne twister.

10.2.4 OrderedSetDS (for IsOrderedSetDS, IsFunction, IsRandomSource)

\[ \text{OrderedSetDS}(\text{arg1, arg2, arg3}) \] (operation)

10.2.5 OrderedSetDS (for IsOrderedSetDS, IsListOrCollection, IsRandomSource)

\[ \text{OrderedSetDS}(\text{arg1, arg2, arg3}) \] (operation)
10.2.6 OrderedSetDS (for IsOrderedSetDS, IsFunction, IsListOrCollection)

\[\text{OrderedSetDS}(\text{arg1, arg2, arg3})\]

10.2.7 OrderedSetDS (for IsOrderedSetDS, IsFunction)

\[\text{OrderedSetDS}(\text{arg1, arg2})\]

10.2.8 OrderedSetDS (for IsOrderedSetDS, IsListOrCollection)

\[\text{OrderedSetDS}(\text{arg1, arg2})\]

10.2.9 OrderedSetDS (for IsOrderedSetDS)

\[\text{OrderedSetDS}(\text{arg})\]

10.2.10 AddSet (for IsOrderedSetDS and IsMutable, IsObject)

\[\text{AddSet}(\text{set, object})\]

Other constructors cover making an ordered set from another ordered set, from an iterator, from a function and an iterator, or from a function, an iterator and a random source.

Adds object to set. Does nothing if object in set.

10.2.11 RemoveSet (for IsOrderedSetDS and IsMutable, IsObject)

\[\text{RemoveSet}(\text{set, object})\]

Returns: 0 or 1

Remove object from set if present, and returns the number of copies of object that were in set, that is 0 or 1. This for consistency with multisets.

10.2.12 \ in (for IsObject, IsOrderedSetDS)

\[\text{\ in}(\text{object, set})\]

All objects in IsOrderedSetDS must implement \ in, which returns true if object is present in set and false otherwise.

10.2.13 LessFunction (for IsOrderedSetDS)

\[\text{LessFunction}(\text{set})\]

The binary function to perform the comparison for elements of the set.
10.2.14 **Size (for IsOrderedSetDS)**

\[ \text{Size(set)} \]

The number of objects in the set

10.2.15 **IteratorSorted (for IsOrderedSetDS)**

\[ \text{IteratorSorted(set)} \]

**Returns:** iterator

Returns an iterator of `set` that can be used to iterate through the elements of `set` in the order imposed by `LessFunction`.

10.3 **Default methods**

Default methods based on `IteratorSorted` are installed for the following operations and attributes, but can be overridden for data structures that support better algorithms.

10.3.1 **Iterator (for IsOrderedSetDS)**

\[ \text{Iterator(arg)} \]

10.3.2 **AsSSortedList (for IsOrderedSetDS)**

\[ \text{AsSSortedList(arg)} \]

10.3.3 **AsSortedList (for IsOrderedSetDS)**

\[ \text{AsSortedList(arg)} \]

10.3.4 **AsList (for IsOrderedSetDS)**

\[ \text{AsList(arg)} \]

10.3.5 **EnumeratorSorted (for IsOrderedSetDS)**

\[ \text{EnumeratorSorted(arg)} \]

10.3.6 **Enumerator (for IsOrderedSetDS)**

\[ \text{Enumerator(arg)} \]
10.3.7 IsEmpty (for IsOrderedSetDS)

\[\text{IsEmpty}(arg)\] (property)

Returns: true or false

10.3.8 Length (for IsOrderedSetDS)

\[\text{Length}(arg)\] (attribute)

10.3.9 Position (for IsOrderedSetDS, IsObject, IsInt)

\[\text{Position}(arg1, arg2, arg3)\] (operation)

10.3.10 PositionSortedOp (for IsOrderedSetDS, IsObject)

\[\text{PositionSortedOp}(arg1, arg2)\] (operation)

10.3.11 PositionSortedOp (for IsOrderedSetDS, IsObject, IsFunction)

\[\text{PositionSortedOp}(arg1, arg2, arg3)\] (operation)
Chapter 11

Stacks

A stack is a deque where items can be Pushed onto the stack, and the top item can be Popped off the stack.
Stacks are wrapped GAP plain lists.

11.1 API

11.1.1 Stack

Stack() (function)

Returns: stack
Constructor for stacks

11.1.2 IsStack (for IsObject)

IsStack(arg) (filter)

Returns: true or false
Category of heaps

11.1.3 Push (for IsStack, IsObject)

Push(stack, object) (operation)

Puts object onto stack.

11.1.4 Peek (for IsStack)

Peek(stack) (operation)

Returns: object or fail
Return the object at the top of stack. If stack is empty, returns fail

11.1.5 Pop (for IsStack)

Pop(stack) (operation)

Returns: object or fail
Remove the top item from stack and return it. If stack is empty, this function returns fail.
11.1.6 Size (for [IsStack])

\[ \text{Size}(\text{arg}) \]

Number of elements on stack
References

Index

datastructures, 6

AddSet
  for IsHashSetRep, IsObject, 21
  for IsOrderedSetDS and IsMutable, IsObject, 26
AsList
  for IsOrderedSetDS, 27
AsSet
  for IsHashSetRep, 22
AsSortedList
  for IsOrderedSetDS, 27
AsSSortedList
  for IsOrderedSetDS, 27

BinaryHeap, 9

Capacity
  for IsDeque, 11
  for IsQueue, 11

Dequeue
  for IsQueue, IsObject, 11

Enqueue
  for IsQueue, IsObject, 11

Enumerator
  for IsOrderedSetDS, 27

EnumeratorSorted
  for IsOrderedSetDS, 27

HashBasic, 16
HashMap, 18
Hash_PermGroup_Complete, 17
Hash_PermGroup_Fast, 16
HashSet, 21
Heap, 8
\in
  for IsObject, IsHashMapRep, 19
  for IsObject, IsHashSetRep, 21

IsBinaryHeapFlatRep
  for IsHeap and IsPositionalObjectRep, 9

IsDeque
  for IsObject, 10

IsEmpty
  for IsHashMapRep, 20
  for IsHashSetRep, 22
  for IsOrderedSetDS, 28

IsHashMap
  for IsObject and IsFinite, 18

IsHashSet
  for IsObject and IsFinite, 21

IsHeap
  for IsObject, 8

IsOrderedSetDS
  for IsObject, 25

IsPairingHeapFlatRep
  for IsHeap and IsPositionalObjectRep, 9

IsPartitionDS
  for IsObject, 14

IsQueue
  for IsObject, 14

IsStack
  for IsObject, 29

IsStandardOrderedSetDS
  for IsOrderedSetDS, 25

Iterator
  for IsHashSetRep, 22
  for IsOrderedSetDS, 27

IteratorSorted
  for IsOrderedSetDS, 27

KeyIterator
  for IsHashMap, 19

Keys
  for IsHashMap, 18

KeyValueIterator
  for IsHashMap, 19
Length
  for IsDeque, 11
  for IsOrderedSetDS, 28
  for IsQueue, 11
LessFunction
  for IsOrderedSetDS, 26
MemoizeFunction, 23
Merge
  for IsHeap, IsHeap, 8
NewHeap
  for IsHeap, IsObject, IsObject, 8
NumberParts
  for IsPartitionDS, 15
OrderedSetDS
  for IsOrderedSetDS, 26
  for IsOrderedSetDS, IsFunction, 26
  for IsOrderedSetDS, IsFunction, IsListOrCollection, 26
  for IsOrderedSetDS, IsListOrCollection, 26
  for IsOrderedSetDS, IsListOrCollection, IsRandomSource, 25
PairingHeap, 9
PartitionDS
  for IsPartitionDS, IsCyclotomicCollColl, 14
  for IsPartitionDS, IsPosInt, 14
PartsOfPartitionDS
  for IsPartitionDS, 15
Peek
  for IsHeap, 8
  for IsStack, 29
PlistDeque, 12
PlistDequeExpand, 13
PlistDequePeekBack, 13
PlistDequePeekFront, 13
PlistDequePopBack, 12
PlistDequePopFront, 12
PlistDequePushBack, 12
PlistDequePushFront, 12
Pop
  for IsHeap, 8
  for IsStack, 29
PopBack
  for IsDeque, 10
PopFront
  for IsDeque, 11
Position
  for IsOrderedSetDS, IsObject, IsInt, 28
PositionSortedUp
  for IsOrderedSetDS, IsObject, 28
  for IsOrderedSetDS, IsObject, IsFunction, 28
Push
  for IsHeap, IsObject, 8
  for IsStack, IsObject, 29
PushBack
  for IsDeque, IsObject, 10
PushFront
  for IsDeque, IsObject, 10
RemoveSet
  for IsHashSetRep, IsObject, 22
  for IsOrderedSetDS and IsMutable, IsObject, 26
Representative
  for IsPartitionDS, IsPosInt, 14
RootsIteratorOfPartitionDS
  for IsPartitionDS, 15
Set
  for IsHashSetRep, 22
Size
  for [IsStack], 30
  for IsHashMapRep, 19
  for IsHashSetRep, 22
  for IsOrderedSetDS, 27
SizeUnderlyingSetDS
  for IsPartitionDS, 15
Stack, 29
Unite
  for IsPartitionDS and IsMutable, IsPosInt, IsPosInt, 15
ValueIterator
  for IsHashMap, 19
Values
  for IsHashMap, 18