datastructures

Collection of standard data structures for GAP

0.2.3

18 December 2018

Markus Pfeiffer

Max Horn

Christopher Jefferson

Steve Linton

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://sl4.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

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

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, `Push`ing data onto the heap, `Peek`ing at the topmost item, and `Pop`ping 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.

Example
```
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.

Example
```
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]`*`)` (constructor)

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 implemenataion 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

▷ `BinaryHeap([isLess[, data]])` (function)

Returns: A binary heap

Constructor for binary heaps. The optinal argument *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 *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

▷ `PairingHeap([isLess[, data]])` (function)

Returns: A pairing heap

Constructor for pairing heaps. The optional argument *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 *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)

▷ `IsBinaryHeapFlatRep(arg)` (filter)

Returns: `true` or `false`

3.6 Implementation

3.6.1 IsPairingHeapFlatRep (for IsHeap and IsPositionalObjectRep)

▷ `IsPairingHeapFlatRep(arg)` (filter)

Returns: `true` or `false`

Chapter 4

Queues and Deques

4.1 API

4.1.1 IsQueue (for IsObject)

▷ `IsQueue(`*arg*`)` (filter)

Returns: `true` or `false`

The category of queues.

4.1.2 IsDeque (for IsObject)

▷ `IsDeque(`*arg*`)` (filter)

Returns: `true` or `false`

The category of deques.

4.1.3 PushBack (for IsDeque, IsObject)

▷ `PushBack(`*deque, object*`)` (operation)

Add *object* to the back of *deque*.

4.1.4 PushFront (for IsDeque, IsObject)

▷ `PushFront(`*deque, object*`)` (operation)

Add *object* to the front of *deque*.

4.1.5 PopBack (for IsDeque)

▷ `PopBack(`*deque*`)` (operation)

Returns: object

Remove an element from the back of *deque* and return it.

4.1.6 PopFront (for IsDeque)

▷ `PopFront(deque)` (operation)

Returns: object

Remove an element from the front of *deque* and return it.

For queues, this is just an alias for PushBack

4.1.7 Enqueue (for IsQueue, IsObject)

▷ `Enqueue(queue, object)` (operation)

Add *object* to *queue*.

4.1.8 Dequeue (for IsQueue, IsObject)

▷ `Dequeue(queue)` (operation)

Returns: object

Remove an object from the front of *queue* and return it.

4.1.9 Capacity (for IsQueue)

▷ `Capacity(arg)` (attribute)

Allocated storage capacity of *queue*.

4.1.10 Capacity (for IsDeque)

▷ `Capacity(arg)` (attribute)

Allocated storage capacity of *deque*.

4.1.11 Length (for IsQueue)

▷ `Length(arg)` (attribute)

Number of elements in *queue*.

4.1.12 Length (for IsDeque)

▷ `Length(arg)` (attribute)

Number of elements in *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![1]` - head, the index in `Q![5]` of the first element in the deque
- `Q![2]` - tail, the index in `Q![5]` of the last element in the deque
- `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 optinal 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*`)` (filter)

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*`)` (constructor)

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*`)` (constructor)

Returns the union find structure of *partition*.

5.2.4 Representative (for IsPartitionDS, IsPosInt)

▷ `Representative(`*unionfind*`,` *k*`)` (operation)

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)

▷ `Unite(`*`unionfind, k1, k2`*`)` (operation)

Fuses the parts of the paritition *`unionfind`* containing *`k1`* and *`k2`*.

5.2.6 RootsIteratorOfPartitionDS (for IsPartitionDS)

▷ `RootsIteratorOfPartitionDS(`*`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)

▷ `NumberParts(`*`unionfind`*`)` (attribute)

Returns: a positive integer

Returns the number of parts of the partition *`unionfind`*.

5.2.8 SizeUnderlyingSetDS (for IsPartitionDS)

▷ `SizeUnderlyingSetDS(`*`unionfind`*`)` (attribute)

Returns: a positive integer

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

5.2.9 PartsOfPartitionDS (for IsPartitionDS)

▷ `PartsOfPartitionDS(`*`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..2^28-1]` on a 32-bit system, and `[-2^60..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

▷ `HashBasic(`*obj...*`)` (function)

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 `HashBasic(x,y,z) = HashBasic([x,y,z])`.

6.3 Hash Functions for Permutation Groups

datastructures provides two hash functions for permutation groups; `Hash_PermGroup_Fast` (6.3.1) is the faster one, with higher likelihood of collisions and `Hash_PermGroup_Complete` (6.3.2) is slower but provides a lower likelihood of collisions.

6.3.1 Hash_PermGroup_Fast

▷ `Hash_PermGroup_Fast(`*group*`)` (function)

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

▷ Hash_PermGroup_Complete(*group*) (function)

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)

▷ `IsHashMap(arg)` (filter)

Returns: `true` or `false`

Category of hash maps

7.1.2 HashMap

▷ `HashMap([hashfunc[, eqfunc, ]][capacity])` (function)

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)

▷ `Keys(h)` (operation)

Returns: a list

Returns the list of keys of the hashmap *h*.

7.1.4 Values (for IsHashMap)

▷ `Values(h)` (operation)

Returns: a list

Returns the set of values stored in the hashmap *h*.

7.1.5 KeyIterator (for IsHashMap)

▷ `KeyIterator(h)` (operation)

Returns: an iterator

Returns an interator for the keys stored in the hashmap *h*.

7.1.6 ValueIterator (for IsHashMap)

▷ `ValueIterator(h)` (operation)

Returns: an iterator

Returns an interator for the values stored in the hashmap *h*.

7.1.7 KeyValueIterator (for IsHashMap)

▷ `KeyValueIterator(h)` (operation)

Returns: an iterator

Returns an interator for key-value-pairs stored in the hashmap *h*.

7.1.8 \[\] (for IsHashMapRep, IsObject)

▷ `\[\](hashmap, object)` (function)

List-style access for hashmaps.

7.1.9 \[\]\:\= (for IsHashMapRep, IsObject, IsObject)

▷ `\[\]\:\=(hashmap, object, object)` (function)

List-style assignment for hashmaps.

7.1.10 \in (for IsObject, IsHashMapRep)

▷ `\in(object, hashmap)` (function)

Test whether a key is stored in the hashmap.

7.1.11 IsBound\[\] (for IsHashMapRep, IsObject)

▷ `IsBound\[\](object, hashmap)` (function)

Test whether a key is stored in the hashmap.

7.1.12 Unbind\[\] (for IsHashMapRep, IsObject)

▷ `Unbind\[\](object, hashmap)` (function)

Delete a key from a hashmap.

7.1.13 Size (for IsHashMapRep)

▷ `Size(hashmap)` (function)

Determine the number of keys stored in a hashmap.

7.1.14 IsEmpty (for IsHashMapRep)

▷ `IsEmpty(object, hashmap)` (function)

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)

▷ `IsHashSet(arg)` (filter)

Returns: `true` or `false`

Category of hashsets

8.1.2 HashSet

▷ `HashSet([hashfunc[, eqfunc, ]][capacity])` (function)

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)

▷ `AddSet(hashset, obj)` (function)

Add *obj* to *hashset*.

8.1.4 \in (for IsObject, IsHashSetRep)

▷ `\in(obj, hashset)` (function)

Test membership of *obj* in *hashset*

8.1.5 RemoveSet (for IsHashSetRep, IsObject)

▷ `RemoveSet(hashset, obj)` (function)

Remove *obj* from *hashset*.

8.1.6 Size (for IsHashSetRep)

▷ `Size(hashset)` (function)

Return the size of a hashset Returns an integer

8.1.7 IsEmpty (for IsHashSetRep)

▷ `IsEmpty(hashset)` (function)

Returns: a boolean

Test a hashset for emptiness.

8.1.8 Set (for IsHashSetRep)

▷ `Set(hashset)` (function)

Returns: a set

Convert a hashset into a GAP set

8.1.9 AsSet (for IsHashSetRep)

▷ `AsSet(hashset)` (function)

Returns: an immutable set

Convert a hashset into a GAP set

8.1.10 Iterator (for IsHashSetRep)

▷ `Iterator(set)` (function)

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

▷ `MemoizeFunction(`*`function[, options]`*`)` (function)

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 the 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

Example

```
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;
```

```
false
gap> 97 in b;
true
```

10.2 API

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

10.2.1 IsOrderedSetDS (for IsObject)

▷ `IsOrderedSetDS(arg)` (filter)

Returns: `true` or `false`

Category of ordered set.

10.2.2 IsStandardOrderedSetDS (for IsOrderedSetDS)

▷ `IsStandardOrderedSetDS(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)

▷ `OrderedSetDS(filter[, lessThan[, initialEntries[, randomSource]]])` (constructor)

Returns: an ordered set datastructure

The family that contains all ordered set datastructures. Constructors for ordered sets

The argument *filter* is a filter that the resulting ordered set object will have.

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

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

The optional argument *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)

▷ `OrderedSetDS(arg1, arg2, arg3)` (constructor)

10.2.5 OrderedSetDS (for IsOrderedSetDS, IsListOrCollection, IsRandomSource)

▷ `OrderedSetDS(arg1, arg2, arg3)` (constructor)

10.2.6 OrderedSetDS (for IsOrderedSetDS, IsFunction, IsListOrCollection)

▷ `OrderedSetDS(arg1, arg2, arg3)` (constructor)

10.2.7 OrderedSetDS (for IsOrderedSetDS, IsFunction)

▷ `OrderedSetDS(arg1, arg2)` (constructor)

10.2.8 OrderedSetDS (for IsOrderedSetDS, IsListOrCollection)

▷ `OrderedSetDS(arg1, arg2)` (constructor)

10.2.9 OrderedSetDS (for IsOrderedSetDS)

▷ `OrderedSetDS(arg)` (constructor)

10.2.10 AddSet (for IsOrderedSetDS and IsMutable, IsObject)

▷ `AddSet(set, object)` (operation)

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* set.

10.2.11 RemoveSet (for IsOrderedSetDS and IsMutable, IsObject)

▷ `RemoveSet(set, object)` (operation)

Returns: `0` or `1`

Removes *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)

▷ `\in(object, set)` (operation)

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)

▷ `LessFunction(set)` (attribute)

The binary function to perform the comparison for elements of the set.

10.2.14 Size (for IsOrderedSetDS)

▷ `Size(set)` (attribute)

The number of objects in the set

10.2.15 IteratorSorted (for IsOrderedSetDS)

▷ `IteratorSorted(set)` (operation)

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)

▷ `Iterator(arg)` (operation)

10.3.2 AsSSortedList (for IsOrderedSetDS)

▷ `AsSSortedList(arg)` (attribute)

10.3.3 AsSortedList (for IsOrderedSetDS)

▷ `AsSortedList(arg)` (attribute)

10.3.4 AsList (for IsOrderedSetDS)

▷ `AsList(arg)` (attribute)

10.3.5 EnumeratorSorted (for IsOrderedSetDS)

▷ `EnumeratorSorted(arg)` (attribute)

10.3.6 Enumerator (for IsOrderedSetDS)

▷ `Enumerator(arg)` (attribute)

10.3.7 IsEmpty (for IsOrderedSetDS)

▷ `IsEmpty(`*arg*`)` (property)

Returns: `true` or `false`

10.3.8 Length (for IsOrderedSetDS)

▷ `Length(`*arg*`)` (attribute)

10.3.9 Position (for IsOrderedSetDS, IsObject, IsInt)

▷ `Position(`*arg1, arg2, arg3*`)` (operation)

10.3.10 PositionSortedOp (for IsOrderedSetDS, IsObject)

▷ `PositionSortedOp(`*arg1, arg2*`)` (operation)

10.3.11 PositionSortedOp (for IsOrderedSetDS, IsObject, IsFunction)

▷ `PositionSortedOp(`*arg1, arg2, arg3*`)` (operation)

Chapter 11

Slices

A slice is a sublist of a list. Creating a slice does not copy the original list, and changes to the list also change a slice of the list.

11.1 API

11.1.1 Slice

▷ `Slice()` (function)

Returns: a slice

Constructor for slices

11.1.2 IsSlice (for IsList)

▷ `IsSlice(arg)` (filter)

Returns: `true` or `false`

Category of slices

11.1.3 \[\] (for IsSliceRep, IsPosInt)

▷ `\[\](slice, value)` (function)

List-style access for slices.

11.1.4 \[\]\:\= (for IsSliceRep and IsMutable, IsPosInt, IsObject)

▷ `\[\]\:\=(slice, value, object)` (function)

List-style assignment for slices.

11.1.5 \in (for IsObject, IsSliceRep)

▷ `\in(object, slice)` (function)

Test whether a value is stored in the slice.

11.1.6 IsBound\[\] (for IsSliceRep, IsPosInt)

▷ `IsBound\[\](`*`slice, value`*`)` (function)

Test whether a location is bound in a slice.

11.1.7 Unbind\[\] (for IsSliceRep and IsMutable, IsPosInt)

▷ `Unbind\[\](`*`slice, value`*`)` (function)

Unbind a value from a slice.

11.1.8 Length (for IsSliceRep)

▷ `Length(`*`slice`*`)` (function)

Determine the length of a slice.

Chapter 12

Stacks

A stack is a deque where items can be `Push`ed onto the stack, and the top item can be `Pop`ped off the stack.

Stacks are wrapped GAP plain lists.

12.1 API

12.1.1 Stack

▷ `Stack()` (function)

Returns: stack

Constructor for stacks

12.1.2 IsStack (for IsObject)

▷ `IsStack(`*`arg`*`)` (filter)

Returns: `true` or `false`

Category of heaps

12.1.3 Push (for IsStack, IsObject)

▷ `Push(`*`stack, object`*`)` (operation)

Puts *`object`* onto *`stack`*.

12.1.4 Peek (for IsStack)

▷ `Peek(`*`stack`*`)` (operation)

Returns: object or `fail`

Return the object at the top of *`stack`*. If *`stack`* is empty, returns `fail`

12.1.5 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.

12.1.6 Size (for [IsStack])

▷ `Size(arg)` (attribute)

Number of elements on *stack*

References

[FSST86] Michael L. Fredman, Robert Sedgewick, Daniel D. Sleator, and Robert E. Tarjan. The pairing heap: A new form of self-adjusting heap. *Algorithmica*, 1(1):111–129, Nov 1986. 9

Index