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

© 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.
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)

\[
\texttt{IsHeap(arg)}
\]

\textbf{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

\[
\texttt{Heap(arg)}
\]

Wrapper function around constructors

3.2.3 NewHeap (for IsHeap, IsObject, IsObject)

\[
\texttt{NewHeap([filter, func, data])}
\]

\textbf{Returns:} a heap

Construct a new heap

3.2.4 Push (for IsHeap, IsObject)

\[
\texttt{Push(heap, object)}
\]

Puts the object object a new object onto heap.

3.2.5 Peek (for IsHeap)

\[
\texttt{Peek(heap)}
\]

Inspect the item at the top of heap.

3.2.6 Pop (for IsHeap)

\[
\texttt{Pop(heap)}
\]

\textbf{Returns:} an object

Remove the top item from heap and return it.

3.2.7 Merge (for IsHeap, IsHeap)

\[
\texttt{Merge(heap1, heap2)}
\]

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](http://stackoverflow.com/questions/6531543)

3.3.1 BinaryHeap

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

Returns: A binary heap

Constructor for binary heaps. The optional argument \textit{isLess} must be a binary function that performs comparison between two elements on the heap, and returns \texttt{true} if the first argument is less than the second, and \texttt{false} otherwise. Using the optional argument \texttt{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 \texttt{GAP} lists. \texttt{Push} and \texttt{Peek} have $O(1)$ complexity, and \texttt{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

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

Returns: A pairing heap

Constructor for pairing heaps. The optional argument \textit{isLess} must be a binary function that performs comparison between two elements on the heap, and returns \texttt{true} if the first argument is less than the second, and \texttt{false} otherwise. Using the optional argument \texttt{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)

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

Returns: \texttt{true} or \texttt{false}

3.6 Implementation

3.6.1 IsPairingHeapFlatRep (for IsHeap and IsPositionalObjectRep)

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

Returns: \texttt{true} or \texttt{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)

**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)

Add object to queue.

4.1.8 Dequeue (for IsQueue, IsObject)

▷ Dequeue(queue)

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

4.1.9 Capacity (for IsQueue)

▷ Capacity(arg)

Allocated storage capacity of queue.

4.1.10 Capacity (for IsDeque)

▷ Capacity(arg)

Allocated storage capacity of deque.

4.1.11 Length (for IsQueue)

▷ Length(arg)

Number of elements in queue.

4.1.12 Length (for IsDeque)

▷ Length(arg)

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
• \texttt{Q![1]} - head, the index in \texttt{Q![5]} of the first element in the deque
• \texttt{Q![2]} - tail, the index in \texttt{Q![5]} of the last element in the deque
• \texttt{Q![3]} - capacity, the allocated capacity in the deque
• \texttt{Q![4]} - factor by which storage is increased if capacity is exceeded
• \texttt{Q![5]} - GAP plain list with storage for capacity many entries

Global constants \texttt{QHEAD}, \texttt{QTAIL}, \texttt{QCAPACITY}, \texttt{QFACTOR}, and \texttt{QDATA} are bound to reflect the above. When a push fills the deque, its capacity is resized by a factor of \texttt{QFACTOR} using \texttt{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

\begin{verbatim}
\begin{verbatim}
\texttt{PlistDeque(capacity[, factor])} (function)

\textbf{Returns: a deque}

Constructor for plist based deques. The optional argument \texttt{capacity} must be a positive integer and is the capacity of the created deque, and the optional argument \texttt{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.
\end{verbatim}
\end{verbatim}

4.2.2 PlistDequePushFront

\begin{verbatim}
\texttt{PlistDequePushFront(deque, object)} (function)

Push \texttt{object} to the front of \texttt{deque}.
\end{verbatim}

4.2.3 PlistDequePushBack

\begin{verbatim}
\texttt{PlistDequePushBack(deque, object)} (function)

Push \texttt{object} to the back of \texttt{deque}.
\end{verbatim}

4.2.4 PlistDequePopFront

\begin{verbatim}
\texttt{PlistDequePopFront(deque)} (function)

\textbf{Returns: object or fail}

Pop object from the front of \texttt{deque} and return it. If \texttt{deque} is empty, returns \texttt{fail}.
\end{verbatim}

4.2.5 PlistDequePopBack

\begin{verbatim}
\texttt{PlistDequePopBack(deque)} (function)

\textbf{Returns: object or fail}

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

 chóng 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

 chóng 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

 chóng 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)

\(\text{IsPartitionDS}(\text{arg})\) (filter)

Returns: true or false

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

5.2.2 PartitionDS (for IsPartitionDS, IsPosInt)

\(\text{PartitionDS}(\text{filter}, n)\) (constructor)

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

5.2.3 PartitionDS (for IsPartitionDS, IsCyclotomicCollColl)

\(\text{PartitionDS}(\text{filter}, \text{partition})\) (constructor)

Returns the union find structure of partition.

5.2.4 Representative (for IsPartitionDS, IsPosInt)

\(\text{Representative}(\text{unionfind}, k)\) (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\(\text{unionfind}, \ k1, \ k2\)  
  (operation)

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

5.2.6  **RootsIteratorOfPartitionDS (for IsPartitionDS)**

▷ RootsIteratorOfPartitionDS\(\text{unionfind}\)  
  (operation)

  Returns: an iterator

  Returns an iterator that runs through canonical representatives of parts of the partition \text{unionfind}.

5.2.7  **NumberParts (for IsPartitionDS)**

▷ NumberParts\(\text{unionfind}\)  
  (attribute)

  Returns: a positive integer

  Returns the number of parts of the partition \text{unionfind}.

5.2.8  **SizeUnderlyingSetDS (for IsPartitionDS)**

▷ SizeUnderlyingSetDS\(\text{unionfind}\)  
  (attribute)

  Returns: a positive integer

  Returns the size of the underlying set of the partition \text{unionfind}.

5.2.9  **PartsOfPartitionDS (for IsPartitionDS)**

▷ PartsOfPartitionDS\(\text{unionfind}\)  
  (attribute)

  Returns: a list of lists

  Returns the partition \text{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

$\triangleright$ 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; 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

$\triangleright$ 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

- **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

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

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 IsHashMapRep, IsObject)

\[\] (hashmap, object)

List-style access for hashmaps.

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

\[\]::=(hashmap, object, object)

List-style assignment 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 IsHashMapRep, IsObject)

IsBound\[\](object, hashmap)

Test whether a key is stored in the hashmap.

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

Unbind\[\](object, hashmap)

Delete a key from a hashmap.
7.1.13 Size (for IsHashMapRep)

\[ \text{Size}(\text{hashtable}) \] (function)

Determine the number of keys stored in a hashmap.

7.1.14 IsEmpty (for IsHashMapRep)

\[ \text{IsEmpty}(\text{object}, \text{hashtable}) \] (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)

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

Returns: true or false

Category of hashsets

8.1.2 HashSet

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

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)

\[\text{AddSet}\ (\text{hashset}, \text{obj})\]

Add \text{obj} to \text{hashset}.

8.1.4 \text{\in} (for IsObject, IsHashSetRep)

\[\text{\in}\ (\text{obj}, \text{hashset})\]

Test membership of \text{obj} in \text{hashset}
8.1.5 **RemoveSet (for IsHashSetRep, IsObject)**

Remove set `(hashset, obj)`

Remove `obj` from `hashset`.

8.1.6 **Size (for IsHashSetRep)**

Size `(hashset)`

Return the size of a hashset Returns an integer

8.1.7 **IsEmpty (for IsHashSetRep)**

IsEmpty `(hashset)`

Returns: a boolean
Test a hashset for emptiness.

8.1.8 **Set (for IsHashSetRep)**

Set `(hashset)`

Returns: a set
Convert a hashset into a GAP set

8.1.9 **AsSet (for IsHashSetRep)**

AsSet `(hashset)`

Returns: an immutable set
Convert a hashset into a GAP set

8.1.10 **Iterator (for IsHashSetRep)**

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

▷ MemoizeFunction(function[, 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 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

```gap
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;
```
10.2 API

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

10.2.1 IsOrderedSetDS (for IsObject)

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

\hspace{1cm}\textbf{Returns:} true or false

\hspace{1cm}Category of ordered set.

10.2.2 IsStandardOrderedSetDS (for IsOrderedSetDS)

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

\hspace{1cm}\textbf{Returns:} true or false

\hspace{1cm}Subcategory of ordered sets where the ordering is GAP’s default UNKNOWNEntity(leq)

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

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

\hspace{1cm}\textbf{Returns:} an ordered set datastructure

\hspace{1cm}The family that contains all ordered set datastructures. Constructors for ordered sets

\hspace{1cm}The argument \text{filter} is a filter that the resulting ordered set object will have.

\hspace{1cm}The optional argument \text{lessThan} must be a binary function that returns \text{true} if its first argument

\hspace{1cm}is less than its second argument, and \text{false} otherwise. The default \text{lessThan} is GAP’s built in

\hspace{1cm}UNKNOWNEntity(leq).

\hspace{1cm}The optional argument \text{initialEntries} gives a collection of elements that the ordered set is

\hspace{1cm}initialised with, and defaults to the empty set.

\hspace{1cm}The optional argument \text{randomSource} is useful in a number of possible implementations that

\hspace{1cm}use randomised methods to achieve good amortised complexity with high probability and simple data

\hspace{1cm}structures. It defaults to the global Mersenne twister.

10.2.4 OrderedSetDS (for IsOrderedSetDS, IsFunction, IsRandomSource)

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

10.2.5 OrderedSetDS (for IsOrderedSetDS, IsListOrCollection, IsRandomSource)

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

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

10.2.7 OrderedSetDS (for IsOrderedSetDS, IsFunction)

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

10.2.8 OrderedSetDS (for IsOrderedSetDS, IsListOrCollection)

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

10.2.9 OrderedSetDS (for IsOrderedSetDS)

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

10.2.10 AddSet (for IsOrderedSetDS and IsMutable, IsObject)

\[\text{AddSet}(set, object)\] (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 \textit{object} to \textit{set}. Does nothing if \textit{object} in \textit{set}.

10.2.11 RemoveSet (for IsOrderedSetDS and IsMutable, IsObject)

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

\textbf{Returns:} 0 or 1

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

10.2.12 \texttt{\textbackslash in} (for IsObject, IsOrderedSetDS)

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

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

10.2.13 LessFunction (for IsOrderedSetDS)

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

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

**Returns:** true or false

10.3.8 Length (for IsOrderedSetDS)

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

10.3.9 Position (for IsOrderedSetDS, IsObject, IsInt)

\[ \text{Position}(\text{arg}_1, \text{arg}_2, \text{arg}_3) \]

10.3.10 PositionSortedOp (for IsOrderedSetDS, IsObject)

\[ \text{PositionSortedOp}(\text{arg}_1, \text{arg}_2) \]

10.3.11 PositionSortedOp (for IsOrderedSetDS, IsObject, IsFunction)

\[ \text{PositionSortedOp}(\text{arg}_1, \text{arg}_2, \text{arg}_3) \]
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()

Returns: a slice
Constructor for slices

11.1.2 IsSlice (for IsList)

▷ IsSlice(arg)

Returns: true or false
Category of slices

11.1.3 \[\] (for IsSliceRep, IsPosInt)

▷ \[\](slice, value)

List-style access for slices.

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

▷ \[\]:=(slice, value, object)

List-style assignment for slices.

11.1.5 \in (for IsObject, IsSliceRep)

▷ \in(object, slice)

Test whether a value is stored in the slice.
11.1.6 **IsBound**[](slice, value) (for IsSliceRep, IsPosInt)

Test whether a location is bound in a slice.

11.1.7 **Unbind**[](slice, value) (for IsSliceRep and IsMutable, IsPosInt)

Unbind a value from a slice.

11.1.8 **Length** (for IsSliceRep)

Determine the length of a slice.
Chapter 12

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.

12.1 API

12.1.1 Stack

▷ Stack()  
  Returns: stack  
  Constructor for stacks

12.1.2 IsStack (for IsObject)

▷ IsStack(arg)  
  Returns: true or false  
  Category of heaps

12.1.3 Push (for IsStack, IsObject)

▷ Push(stack, object)  
  Puts object onto stack.

12.1.4 Peek (for IsStack)

▷ Peek(stack)  
  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)  
  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)

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

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
AsSortedSet
  for IsOrderedSetDS, 27
BinaryHeap, 9
  for IsHashMapRep, IsObject, 19
  for IsSliceRep, IsPosInt, 29
  \[
  \forall \exists : \forall
  \]
  for IsHashMapRep, IsObject, IsObject, 19
  for IsSliceRep and IsMutable, IsPosInt, IsObject, 29
  \in
  for IsObject, IsHashMapRep, 19
  for IsObject, IsHashSetRep, 21
  for IsObject, IsOrderedSetDS, 26
  for IsObject, IsSliceRep, 29
Capacity
  for IsDeque, 11
  for IsQueue, 11
Dequeue
  for IsQueue, IsObject, 11
Enqueue
  for IsQueue, IsObject, 11
Enumerator
  for IsOrderedSetDS, 27
  for IsOrderedSetDS, 27
HashBasic, 16
HashMap, 18
HashSet, 21
Hash_PermGroup_Complete, 17
Hash_PermGroup_Fast, 16
Heap, 8
IsBinaryHeapFlatRep
  for IsHeap and IsPositionalObjectRep, 9
IsBound\[\]
  for IsHashMapRep, IsObject, 19
  for IsSliceRep, IsPosInt, 30
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 IsObjectDS, 14
IsQueue
  for IsObject, 10
IsSlice
  for IsList, 29
IsStack
  for IsObject, 31
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
  for IsSliceRep, 30
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, 14
Peek
  for IsHeap, 8
  for IsStack, 31
PlistDeque, 12
PlistDequeExpand, 13
PlistDequePeekBack, 13
PlistDequePeekFront, 13
PlistDequePopBack, 12
PlistDequePopFront, 12
PlistDequePushBack, 12
PlistDequePushFront, 12
Pop
  for IsHeap, 8
  for IsStack, 31
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, 31
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], 32
for IsHashMapRep, 20
for IsHashSetRep, 22
for IsOrderedSetDS, 27
SizeUnderlyingSetDS
    for IsPartitionDS, 15
Slice, 29
Stack, 31

Unbind\\]
    for IsHashMapRep, IsObject, 19
    for IsSliceRep and IsMutable, IsPosInt, 30
Unite
    for IsPartitionDS and IsMutable, IsPosInt, Is-PosInt, 15

ValueIterator
    for IsHashMap, 19
Values
    for IsHashMap, 18