BLAH Reference Manual

0.95

Generated by Doxygen 1.3.8

Wed Oct 20 17:42:38 2004

Contents

Chapter 1

The BLAH Reference Manual.

Author:
Ingo Schröder
Kilian A. Foth
Michael Daum

1.1 Introduction

This is the reference manual for the BLAH library for the C programming language. BLAH stands for bitstrings, lists, arrays and hashes — admittedly a poorly chosen giving the set of container data types most usually used. Actually there are more data types than these but who cares if the name is funky enuf.

The maintainers of this software can be contacted at the following email address:

```
blah@nats.informatik.uni-hamburg.de
```

1.2 Usage

The BLAH library is installed as both shared and static versions of the library by default. In order to use it your C code must include the header file

```
blah.h
```

For instance, if you installed the header file in a system directory, a minimal C program might look as follows:

```
#include <stdio.h>
#include <blah.h>

int main(int argc, char **argv)
{
  List squares=listNew();
  int i;

  for (i=0; i<100; i++)
```

```
    {
      squares=listPrependElement(squares, (Pointer)(i*i));
    }
  printf("42th square is %d\n", (int)listNthElement(squares, 42));
  listDelete(squares);
}
```

You have to link against the BLAH library as well as the math library to create a binary:

```
$ gcc -o blah-example -lblah -lm blah-example.c
$ ./blah-example
42th square is 3364
$
```

1.3 Overview

The BLAH library defines the following container data types:

- Arrays: An array stores information that is accessed using a number of indices. Arrays usually have two or more dimensions. The number of dimensions as well as the size of each dimension must be provided at creation time of the array and stay fixed. The required time to access a specific information given the corresponding indices is constant; the memory is proportional to the product of all dimension sizes.
- BitStrings: A bitstring stores a sequence of Booleans. The design emphasis is on memory efficiency.
- ByteVectors: A bitvector also stores a sequence of Booleans. However, one byte is used to store a single Boolean. This data type is more time efficient than a bitstring.
- Hashtables: A hashtable stores key-value pairs. The key is used to access the actual information in the value. Access time is (almost) constant for arbitrary sizes of the hashtable and arbitrary keys.
- Lists: A list stores a sequence of objects. Access to the head of the list is efficiently possible while in general access to an arbitrary object requires linear time.
- ListAgenda: An agenda is created by sorting the items according to priority using a simple linked list as its storage medium.
- Memory: Some basic Operations with the memory are done in this module.
- Primes: Prime is a module that makes use of Rabin's Probablistic Primetest-Algorithm for generating the prime numbers equal to the Hash table entries.
- Ringbuffers: It is similar to the vector that stores a series of objects, though the head and the tail are attached inorder to enhance cyclic operations.
- Strings: Strings in C are represented by arrays of characters. The end of the string is marked with a special character, the null character, which is simply the character with the value 0. Whenever we write a string, enclosed in double quotes, C automatically creates an array of characters for us, containing that string, terminated by the *NULL* character.
- TreeAgenda: An agenda is created by sorting the items according to the order of the priority using an unbalanced binary tree.
- Vectors: A vector stores objects at a given index. The size of the vector is linear in the largest used index. Time to access an object given the index is constant.

Chapter 2

BLAH Module Index

2.1 BLAH Modules

Here is a list of all modules:

Chapter 3

BLAH Data Structure Index

3.1 BLAH Data Structures

Here are the data structures with brief descriptions:

Chapter 4

BLAH Page Index

4.1 BLAH Related Pages

Here is a list of all related documentation pages:

Chapter 5

BLAH Module Documentation

5.1 Arrays

5.1.1 Detailed Description

Implementation of an array container.

An array is a matrix-like data structure with an arbitrary (but fixed) dimension and arbitrary (but fixed) size. Items are accessed by a tuple of indices in constant time. Some of the functions of this module use a variable number of arguments like

- arrayNew()
- arraySetElement() and
- arrayElement().

The defined dimension constructing and array with arrayNew() must match the number of indices you provide to arraySetElement() and arrayElement().

Data Structures

- struct ArrayStruct

  internal structure of an array.

Functions

- Array arrayNew (int i,...)

  creates and returns a new vector.

- Array arrayClone (Array a)

  copy an array into a new one.

- void arrayDelete (Array a)

  deletes an array.

- Pointer arraySetElement (Array a, Pointer new,...)
  sets an array element to a new value.
- void arraySetAllElements (Array a, Pointer new)
  sets all array element to a new value
- Pointer arrayElement (Array a,...)
  retrieves an array element.
- int arrayDimension (Array a, int dim)
  returns value for dimension dim.

5.1.2 Function Documentation

5.1.2.1 Array arrayClone (Array *a*)

copy an array into a new one.

Parameters:
a is the source for the cloning

Returns:
a new cloned array

Definition at line 107 of file array.c.

References vectorClone().

5.1.2.2 void arrayDelete (Array *a*)

deletes an array.

Any future access to the array is illegal. Note that this function does not free the memory from the items contained in the array.

Parameters:
a the array to be deleted.

Definition at line 130 of file array.c.

References vectorDelete().

5.1.2.3 int arrayDimension (Array *a*, int *dim*)

returns value for dimension dim.

Parameters:
a the array whose dimension has to be retrieved.
dim the dimension in which the size of the array has to be retrieved.

Returns:
the size of the array in the specified dimension.

Definition at line 230 of file array.c.

References vectorElement(), and vectorSize().

5.1.2.4 Pointer arrayElement (Array *a*, ...)

retrieves an array element.

Parameters:
a the array from which an element has to be retrieved.

... the indices that define which element has to be retrieved.

Returns:
the element identified by the indices.

Definition at line 198 of file array.c.

References vectorElement(), and vectorSize().

5.1.2.5 Array arrayNew (int *i*, ...)

creates and returns a new vector.

This function constructs a new array with an arbitrary number of dimensions.

Parameters:
i the first array dimension

... optional more dimensions

Returns:
a new Array

Definition at line 69 of file array.c.

References vectorAddElement(), and vectorNew().

5.1.2.6 void arraySetAllElements (Array *a*, Pointer *new*)

sets all array element to a new value

Parameters:
a the array whose values should be set to a new value

new the new value to which all the elements of the array have to be set to.

Definition at line 183 of file array.c.

5.1.2.7 Pointer arraySetElement (Array *a*, Pointer *new*, ...)

sets an array element to a new value.

Parameters:

a the array whose element should be set to a new value.

new the new value to which the element must be set to.

... the index of the new element in the array.

Returns:

the old value of the element

Definition at line 150 of file array.c.

References vectorElement(), and vectorSize().

5.2 BitStrings

5.2.1 Detailed Description

Implementation of a string of bits.

A bitstring is a memory efficient implementation of a sequence of Boolean information with arbitrary but fixed length.

Data Structures

- struct BitStringStruct

  internal structure of a string of bits.

Defines

- #define BITS_PER_BYTE 8

  defines our version of a BYTE definition

- #define BYTES_PER_LONG sizeof(unsigned long)

  defines the size of an unsigned long

- #define BITS_PER_LONG (BITS_PER_BYTE∗BYTES_PER_LONG)

  defines the number of bits in an unsigned long integer

- #define E_SIZEMISMATCH -1

  return code for size error messages

- #define check_magic(bs)

  this is defined to a null operation when debugging is switched off.

Functions

- BitString bitNew (int size)

  creates a new bitstring with an initial size of size *bits.*

- BitString bitClone (BitString bs)

  creates a new bitstring which is a complete clone of bs.

- BitString bitCopy (BitString dst, BitString src)

  copies information from src to dst.

- void bitDelete (BitString bs)

  deletes bitstring bs *and frees memory.*

- int bitSize (BitString bs)

  returns the size of bitstring bs.

- void bitPrint (BitString bs)
  prints a bitstring to stdout.
- void resize (BitString bs, int size)
  resize bs to size size*; sets length, mask and size.*
- void bitSet (BitString bs, int no)
  sets a bit.
- void bitClear (BitString bs, int no)
  clears a bit.
- void bitSetAll (BitString bs)
  sets all the bits to one.
- void bitClearAll (BitString bs)
  sets all bits to zero.
- BitString bitAnd (BitString a, BitString b)
  computes a logical AND *between two sets of bits*
- BitString bitOr (BitString a, BitString b)
  computes a logical OR *between two sets of bits*
- Boolean bitCheck (BitString a, BitString b)
  checks whether two sets have bits switched on in common
- Boolean bitGet (BitString bs, int no)
  test for a bit
- Boolean bitIsAllCleared (BitString bs)
  checks whether all bits are cleared
- Boolean bitIsAllSet (BitString bs)
  checks whether all bits are set

5.2.2 Define Documentation

5.2.2.1 #define BITS_PER_BYTE 8

defines our version of a BYTE definition

Definition at line 48 of file bitstring.c.

5.2.2.2 #define BITS_PER_LONG (BITS_PER_BYTE*BYTES_PER_LONG)

defines the number of bits in an unsigned long integer

Definition at line 58 of file bitstring.c.

Referenced by bitClear(), bitGet(), bitNew(), bitSet(), and resize().

5.2.2.3 #define BYTES_PER_LONG sizeof(unsigned long)

defines the size of an unsigned long

Definition at line 53 of file bitstring.c.

Referenced by bitClearAll(), bitClone(), bitCopy(), bitNew(), bitSetAll(), and resize().

5.2.2.4 #define check_magic(bs)

this is defined to a null operation when debugging is switched off.

Switch it on by compiling with -DBITSTRINGS_DEBUG Definition at line 121 of file bitstring.c.

Referenced by bitAnd(), bitCheck(), bitClear(), bitClearAll(), bitClone(), bitCopy(), bitDelete(), bitGet(), bitIsAllCleared(), bitIsAllSet(), bitOr(), bitPrint(), bitSet(), bitSetAll(), bitSize(), bvAddElement(), bvAndElement(), bvCapacity(), bvClone(), bvDelete(), bvElement(), bvInsertElement(), bvIsEmpty(), bvNotElement(), bvOrElement(), bvRemoveElement(), bvSetElement(), bvSetElements(), bvSize(), and resize().

5.2.2.5 #define E_SIZEMISMATCH -1

return code for size error messages

Definition at line 63 of file bitstring.c.

5.2.3 Function Documentation

5.2.3.1 BitString bitAnd (BitString *a*, BitString *b*)

computes a logical *AND* between two sets of bits

sets all the bits in bitstring a that are set to 0 in bitstring b to 0. bitstring b is never modified. the bitstrings must be of the same size.

Parameters:

a the first set of bits

b the second set of bits

Returns:

a on success and NULL on failure.

Definition at line 368 of file bitstring.c.

References check_magic.

5.2.3.2 Boolean bitCheck (BitString *a*, BitString *b*)

checks whether two sets have bits switched on in common

Parameters:

a the first set

b the second set of bits

Returns:
TRUE on the first common bit (i.e. two bits at the same position which are switched on); otherwise false is returned, that there are no common bits switched on. a->size == b->size

Definition at line 424 of file bitstring.c.

References check_magic.

5.2.3.3 void bitClear (BitString *bs*, int *no*)

clears a bit.

if no is larger than current size the bitstring is enlarged. intermediate bits are cleared. Bits are numbered from zero upward

Parameters:
bs the bitstring from which a bit has to be cleared.

no the bit number that has to be cleared.

Definition at line 321 of file bitstring.c.

References BITS_PER_LONG, check_magic, and resize().

5.2.3.4 void bitClearAll (BitString *bs*)

sets all bits to zero.

Parameters:
bs the bitstring whose bits are to be cleared.

Returns:
the new bitstring after setting all its bits to zero.

Definition at line 350 of file bitstring.c.

References BYTES_PER_LONG, and check_magic.

5.2.3.5 BitString bitClone (BitString *bs*)

creates a new bitstring which is a complete clone of bs.

Parameters:
bs the bit string that has to be cloned

Returns:
a copy of the bitstring after cloning.

Definition at line 175 of file bitstring.c.

References bitNew(), BYTES_PER_LONG, and check_magic.

5.2.3.6 BitString bitCopy (BitString *dst*, BitString *src*)

copies information from src to dst.

dst and src must be bitstrings of the same size

Parameters:
dst destination BitString
src source Bitstring

Returns:
1 on success, 0 on failure (e. g. , due to size mismatch)

Definition at line 197 of file bitstring.c.

References BYTES_PER_LONG, and check_magic.

5.2.3.7 void bitDelete (BitString *bs*)

deletes bitstring *bs* and frees memory.

Parameters:
bs the bitstring that has to be deleted.

Definition at line 214 of file bitstring.c.

References check_magic.

5.2.3.8 Boolean bitGet (BitString *bs*, int *no*)

test for a bit

This function checks whether a bitstring has bit at a certain position switched on.

Parameters:
bs the bitstring
no the position where to look at

Returns:
the state of the bit at the given position

Definition at line 454 of file bitstring.c.

References BITS_PER_LONG, and check_magic.

Referenced by bitPrint().

5.2.3.9 Boolean bitIsAllCleared (BitString *bs*)

checks whether all bits are cleared

Parameters:
bs the bitstring we are talking about

Returns:
FALSE on the first bit being switched on, or TRUE if all are switched off

Definition at line 475 of file bitstring.c.

References check_magic.

5.2.3.10 Boolean bitIsAllSet (BitString *bs*)

checks whether all bits are set

See also:
bitIsAllCleared

Parameters:
bs the current bitstring

Returns:
FALSE on the first bit being swithced off, or TRUE of all bits are switched on

Definition at line 498 of file bitstring.c.

References check_magic.

5.2.3.11 BitString bitNew (int *size*)

creates a new bitstring with an initial size of *size* bits.

Parameters:
size specifies the size of the new bit vector to be created.

Returns:
a new bit vector of the specified size.

Definition at line 132 of file bitstring.c.

References BITS_PER_LONG, and BYTES_PER_LONG.

Referenced by bitClone().

5.2.3.12 BitString bitOr (BitString *a*, BitString *b*)

computes a logical *OR* between two sets of bits

sets all the bits in bitstring a that are set to 1 in bitstring b to 1. bitstring b is never modified. the bitstrings must be of the same size.

Parameters:
a the first set of bits
b the second set of bits

Returns:
a on success and NULL on failure.

Definition at line 397 of file bitstring.c.

References check_magic.

5.2.3.13 void bitPrint (BitString *bs*)

prints a bitstring to stdout.

Parameters:
bs the BitString to be printed

Definition at line 240 of file bitstring.c.

References bitGet(), and check_magic.

5.2.3.14 void bitSet (BitString *bs*, int *no*)

sets a bit.

if *no* is larger than current size the bitstring is enlarged. intermediate bits are cleared. Bits are numbered from zero upwards.

Parameters:
bs the bitstring in which a bit has to be set.
no the number of the bit that has to be set

Definition at line 302 of file bitstring.c.

References BITS_PER_LONG, check_magic, and resize().

5.2.3.15 void bitSetAll (BitString *bs*)

sets all the bits to one.

Parameters:
bs the bitstring whose bits are to be set.

Returns:
the new bitstring after setting all its bits to 1.

Definition at line 336 of file bitstring.c.

References BYTES_PER_LONG, and check_magic.

5.2.3.16 int bitSize (BitString *bs*)

returns the size of bitstring *bs*.

Parameters:
bs the bitstring whose size has to be determined.

Returns:
the size of the bitstring bs

Definition at line 228 of file bitstring.c.

References check_magic.

5.2.3.17 void resize (BitString *bs*, int *size*) [static]

resize bs to size *size*; sets length, mask and size.

Parameters:

bs the bitstring that has to be resized.

size the size to which the bitstring has to be resized.

Returns:

the new resized bitstring.

Definition at line 259 of file bitstring.c.

References BITS_PER_LONG, and BYTES_PER_LONG.

5.3 ByteVectors

5.3.1 Detailed Description

Implementation of a vector of bits.

A bitvector is a special case of a vector which is more memory efficient than a generic vector and more time efficient than a bitstring.

Data Structures

- struct ByteVectorStruct

  internal representation of a bit vector

- struct ByteVectorStruct

  internal representation of a bit vector

Defines

- #define **RESIZEFACTOR** 2
- #define **check_magic**(v)
- #define **RESIZEFACTOR** 2
- #define **check_magic**(v)

Functions

- void resize (ByteVector v)

  doubles capacity of ByteVector.

- ByteVector bvNew (int capacity)

  creates a new empty bitvector with an initial capacity.

- void bvDelete (ByteVector v)

  deletes a bitvector and frees the associated memory.

- int bvAddElement (ByteVector v, char element)

  adds a new element to the end of the ByteVector.

- char bvElement (ByteVector v, int index)

  sets the element at the specific index to a new element.

- char bvRemoveElement (ByteVector v, int index)

  removes element at the specified index.

- char bvInsertElement (ByteVector v, char element, int index)

  inserts a new element at the given index.

- char bvSetElement (ByteVector v, char element, int index)

sets element at the given index.

- char bvAndElement (ByteVector v, char element, int index)
  computes AND to the element at the given index.
- char bvOrElement (ByteVector v, char element, int index)
  computes OR to the element at the given index.
- char bvNotElement (ByteVector v, int index)
  computes NOT to the element at the given index.
- void bvSetElements (ByteVector v, char element, int from, int to)
  sets all the elements between index from *and* to *(excluding to) to a new value.*
- void bvSetAllElements (ByteVector v, char element)
  sets all the elements to a new value.
- int bvCapacity (ByteVector v)
  finds the current capacity of bitvector.
- int bvSize (ByteVector v)
  finds the number of entries(size) in the bitvector.
- char bvIsEmpty (ByteVector v)
  finds if the bit vector is empty or not.
- ByteVector bvClone (ByteVector v)
  creates an exact copy of the specified bitvector.
- ByteVector bvCopy (ByteVector dst, ByteVector src)
  copies entries of one bitvector to another.

5.3.2 Function Documentation

5.3.2.1 int bvAddElement (ByteVector *v*, char *element*)

adds a new element to the end of the ByteVector.

automatically increases the capacity of the ByteVector if necessary

Parameters:

v the vector to which the new element has to be added.

element the element that has to be added to the end of the bitvector v

Returns:

index of the vector

Definition at line 150 of file bitvector-old.c.

References check_magic, and resize().

5.3.2.2 char bvAndElement (ByteVector *v*, char *element*, int *index*)

computes AND to the element at the given index.

if necessary the vector automatically increases its capacity.

Parameters:

v the vector in which the element has to be set at the apecified index.

element the element that has to be set in the bitvector.

index the index at which the element to be set is present.

Returns:

The new value at the index.

Definition at line 310 of file bitvector-old.c.

References check_magic, and resize().

5.3.2.3 int bvCapacity (ByteVector *v*)

finds the current capacity of bitvector.

Parameters:

v the bitvector whose capacity has to be retrieved.

Returns:

the capacity of the bitvector v.

Definition at line 436 of file bitvector-old.c.

References check_magic.

Referenced by bvClone().

5.3.2.4 ByteVector bvClone (ByteVector *v*)

creates an exact copy of the specified bitvector.

Parameters:

v the bitvector for which cloning has to be performed.

Returns:

a complete independent *CLONE* of the bit vector v.

Definition at line 475 of file bitvector-old.c.

References bvCapacity(), bvCopy(), bvNew(), and check_magic.

5.3.2.5 ByteVector bvCopy (ByteVector *dst*, ByteVector *src*)

copies entries of one bitvector to another.

copying is done from bitVector *src* to the bitvector *dst*. *dst* vector automatically increases its capacity.

Parameters:

src source index or the index from which copying has to be initiated.

dst destination index or the index until which the copying has to be done.

Returns:

the new *dst* vector with the elements copied into it.

Definition at line 496 of file bitvector-old.c.

Referenced by bvClone().

5.3.2.6 void bvDelete (ByteVector *v*)

deletes a bitvector and frees the associated memory.

note that nothing user-defined is stored in a bitvector. therefore, the user does not need to free anything himself.

Parameters:

v the bitvector that has to be deleted.

Definition at line 133 of file bitvector-old.c.

References check_magic.

5.3.2.7 char bvElement (ByteVector *v*, int *index*)

sets the element at the specific index to a new element.

if necessary the bitVector automatically increases its capacity.

Parameters:

v the bitvector in which the specified element has to be set to a new element.

index the index of the element that has to be set.

Returns:

the old element or *False* at the specified index.

Definition at line 170 of file bitvector-old.c.

References check_magic.

5.3.2.8 char bvInsertElement (ByteVector *v*, char *element*, int *index*)

inserts a new element at the given index.

if necessary the bitvector automatically increases its capacity. inefficient method,not recommended.

Parameters:

v the bitvector into which the new element has to be inserted.

element the new element that has to be inserted into the bitvector v.

index the index at which the new element has to be inserted in the bitvector v.

Returns:
the old element at the index.

Definition at line 227 of file bitvector-old.c.

References check_magic, and resize().

5.3.2.9 Boolean bvIsEmpty (ByteVector *v*)

finds if the bit vector is empty or not.

Parameters:
v the bitvector for which emptiness is checked.

Returns:
TRUE if bitvector is empty and *FALSE* otherwise.

Definition at line 462 of file bitvector-old.c.

References check_magic.

5.3.2.10 ByteVector bvNew (int *capacity*)

creates a new empty bitvector with an initial capacity.

Specifying a correct or nearly correct capacity slightly improves the efficiency. The bitvector roughly needs capacity bytes.

Parameters:
capacity the capacity of the new vector to be created.

Returns:
a new and empty bit vector of the specified capacity.

Definition at line 107 of file bitvector-old.c.

Referenced by bvClone().

5.3.2.11 char bvNotElement (ByteVector *v*, int *index*)

computes NOT to the element at the given index.

if necessary the vector automatically increases its capacity.

Parameters:
v the vector in which the element has to be set at the apecified index.
index the index at which the element to be set is present.

Returns:
The new value at the index.

Definition at line 363 of file bitvector-old.c.

References check_magic, and resize().

5.3.2.12 char bvOrElement (ByteVector *v*, char *element*, int *index*)

computes OR to the element at the given index.

if necessary the vector automatically increases its capacity.

Parameters:

v the vector in which the element has to be set at the apecified index.

element the element that has to be set in the bitvector.

index the index at which the element to be set is present.

Returns:

The new value at the index.

Definition at line 337 of file bitvector-old.c.

References check_magic, and resize().

5.3.2.13 char bvRemoveElement (ByteVector *v*, int *index*)

removes element at the specified index.

size decreases and all later elements move one position to the front. inefficient method, not recommended.

Parameters:

v the bitvector from which the element has to be removed.

index the index at which the element to be removed is present.

Returns:

old element at the specifed index.

Definition at line 190 of file bitvector-old.c.

References check_magic.

5.3.2.14 void bvSetAllElements (ByteVector *v*, char *element*)

sets all the elements to a new value.

the size increases to the current capacity of the bit vector.

Parameters:

v the bitvector in which all the elements have to be set.

element the elements at the specified indices.

Returns:

the bitvector after setting all the elements.

Definition at line 424 of file bitvector-old.c.

5.3.2.15 char bvSetElement (ByteVector *v*, char *element*, int *index*)

sets element at the given index.

if necessary the vector automatically increases its capacity.

Parameters:

v the vector in which the element has to be set at the apecified index.

element the element that has to be set in the bitvector.

index the index at which the element to be set is present.

Returns:

The old element at the index.

Definition at line 277 of file bitvector-old.c.

References check_magic, and resize().

5.3.2.16 void bvSetElements (ByteVector *v*, char *element*, int *from*, int *to*)

sets all the elements between index *from* and *to* (excluding to) to a new value.

if necesaary the bit vector automatically increases in capacity.

Parameters:

v the bitvector in which the elements have to be set.

element the value of the bits at the corresponding indices.

from the source index from which all the elements in the bitvector have to be set.

to the destination index until which all the elements in the bitvector have to be set.

Returns:

the bitvector v after setting all the elements at the specified indices.

Definition at line 392 of file bitvector-old.c.

References check_magic, and resize().

5.3.2.17 int bvSize (ByteVector *v*)

finds the number of entries(size) in the bitvector.

Parameters:

v the bitvector whose size has to be retrieved.

Returns:

the size of the bitvector v.

Definition at line 449 of file bitvector-old.c.

References check_magic.

5.3.2.18 void resize (ByteVector *v*) [static]

doubles capacity of ByteVector.

Parameters:
v the bitvector that has to be resized.

Returns:
the bitvector after increasing its capacity.

Definition at line 89 of file bitvector-old.c.

References check_magic.

Referenced by bitClear(), bitSet(), bvAddElement(), bvAndElement(), bvInsertElement(), bvNot-Element(), bvOrElement(), bvSetElement(), and bvSetElements().

5.4 Hashtables

5.4.1 Detailed Description

Implementation of hashtables.

A hashtable stores an arbitrary number of objects that are accessed using an arbitrary key. The key is converted to an integer value, so so-called hash value, by the hash function. A hash function should be fast and should map the keys to integers in a highly irregular but consistent way, i.∼e., the keys should be distributed evenly over the whole set of integers. Ideally, objects can then be accessed in constant time based on their key.

Data Structures

- struct HashtableEntryStruct

  internal representation of the hash table entry.

- struct HashtableStruct

  internal representation of the hash table.

- struct HashIteratorStruct

  internal representation of the hash iterator.

Defines

- #define **PRIME_TESTS** 10

Functions

- void rehashHashtable (Hashtable ht)

  rehashes the hashtable.

- Hashtable hashNew (int capacity, double loadFactor, IntFunction ∗hashFunction, IntFunction ∗keyEqualFunction)

  creates a new hashtable with an initial capacity of c.

- Pointer hashSet (Hashtable ht, Pointer key, Pointer value)

  adds the object value with the key key *in the hashtable.*

- Pointer hashGet (Hashtable ht, Pointer key)

  retrieves value associated with the key.

- Pointer ∗ hashGetPointerToValue (Hashtable ht, Pointer key)

  retrieves value associated with the key

- Pointer hashRemove (Hashtable ht, Pointer key)

  removes key/value pair from hashtable

- int hashSize (Hashtable ht)

  retrieves the size of the hashtable

- Boolean hashIsEmpty (Hashtable ht)

  checks if the hashtable is empty.

- Boolean hashContainsKey (Hashtable ht, Pointer key)

  checks if the hashtable contains the specified key.

- Boolean hashContainsValue (Hashtable ht, Pointer value)

  checks if the hashtable contains a specific value.

- void hashDelete (Hashtable ht)

  deletes the hashtable, but can't free the memory for the content.

- void hashForEach (Hashtable ht, VoidFunction *f)

  calls the function 'f(key,value)' for each item in the hashtable.

- void hashForEachWithData (Hashtable ht, VoidFunction *f, Pointer clientData)

  calls the function 'f(element,data)' for each object in the hashtable.

- void hashForEachFree (Hashtable ht, VoidFunction *f)

  calls the function 'f(key, value)' for each item in the Hashtable.

- void hashForEachFreeValue (Hashtable ht, VoidFunction *f)

  List hashForEachFree(), but frees only the embedded value.

- List hashListOfKeys (Hashtable ht)

  retrieves a list of keys of all objects in the hashtable.

- HashIterator hashIteratorNew (Hashtable ht)

  returns a new hash iterator object.

- Pointer hashIteratorNextKey (HashIterator hi)

  returns the next key of a hash-iterator.

- Pointer hashIteratorNextValue (HashIterator hi)

  returns the next value of a hash-iterator.

- void hashIteratorDelete (HashIterator hi)

  deletes and frees hash iterator object

- int hashStringHashFunction (char *s)

  is an example has function for C strings that can be used in hashNew.

- int hashStringEqualFunction (char *s, char *t)

  is an example equality function for C strings that can be used in hashNew.

5.4.2 Function Documentation

5.4.2.1 Boolean hashContainsKey (Hashtable *ht*, Pointer *key*)

checks if the hashtable contains the specified key.

Parameters:

ht the hashtable in which the specified key has to be checked.

key the key whose existance in the hashtable is to be checked.

Returns:

TRUE if the an object under the key exists and *FALSE* otherwise.

Definition at line 346 of file hashtable.c.

5.4.2.2 Boolean hashContainsValue (Hashtable *ht*, Pointer *value*)

checks if the hashtable contains a specific value.

Objects are compared by the standard C operator == and this method is Expensive!!!

Parameters:

ht the hashtable in which the specific value has to be checked.

value the value whose existance in the hashtable has to be checked.

Returns:

TRUE if the object is contained in the hashtable and *FALSE* otherwise.

Definition at line 373 of file hashtable.c.

5.4.2.3 void hashDelete (Hashtable *ht*)

deletes the hashtable, but can't free the memory for the content.

Parameters:

ht the hashtable that has to be deleted.

Definition at line 396 of file hashtable.c.

5.4.2.4 void hashForEach (Hashtable *ht*, VoidFunction * *f*)

calls the function 'f(key,value)' for each item in the hashtable.

Parameters:

ht the hashtable in which the function 'f' has to be called.

f the function that has to be called for every object in the hashtable ht.

Definition at line 417 of file hashtable.c.

Referenced by strFinalize().

5.4.2.5 void hashForEachFree (Hashtable *ht,* VoidFunction * *f*)

calls the function 'f(key, value)' for each item in the Hashtable.

deletes the hashtable and hashtable becomes inaccessible

Parameters:

ht the hashtable in which the function 'f' has to be called.

f the function that has to be called for every object in the hashtable ht.

Definition at line 455 of file hashtable.c.

Referenced by strFinalize().

5.4.2.6 void hashForEachFreeValue (Hashtable *ht,* VoidFunction * *f*)

List hashForEachFree(), but frees only the embedded value.

F is applied to the value only and must be a unariy void function. Definition at line 475 of file hashtable.c.

5.4.2.7 void hashForEachWithData (Hashtable *ht,* VoidFunction * *f,* Pointer *clientData*)

calls the function 'f(element,data)' for each object in the hashtable.

Parameters:

ht the hashtable in which the function 'f' has to be called.

f the function that has to be called in the hashtable ht.

clientData the data in the function 'f(element,data)'that is called in the hashtable ht.

Definition at line 435 of file hashtable.c.

5.4.2.8 Pointer hashGet (Hashtable *ht,* Pointer *key*)

retrieves value associated with the key.

Parameters:

ht the hashtable from which the value at the specified key has to be retrieved.

key the key at which the value of the hashtable has to be retrieved.

Returns:

the object that(or *NULL*) that is stored under the key in the hashtable.

Definition at line 222 of file hashtable.c.

Referenced by _strLookup().

5.4.2.9 Pointer* hashGetPointerToValue (Hashtable *ht,* Pointer *key*)

retrieves value associated with the key

Parameters:

ht the hashtable from which the value at the specified key has to be retrieved.

key the key at which the value of the hashtable has to be retrieved.

Returns:

the object that(or *NULL*) that is stored under the key in the hashtable.

Definition at line 250 of file hashtable.c.

5.4.2.10 Boolean hashIsEmpty (Hashtable *ht*)

checks if the hashtable is empty.

Parameters:

ht the hash table whose emptiness is checked.

Returns:

TRUE if the hashtable is empty and *False* otherwise.

Definition at line 329 of file hashtable.c.

5.4.2.11 void hashIteratorDelete (HashIterator *hi*)

deletes and frees hash iterator object

Parameters:

hi the hash iterator object that has to be deleted.

Definition at line 603 of file hashtable.c.

5.4.2.12 HashIterator hashIteratorNew (Hashtable *ht*)

returns a new hash iterator object.

Iterators allow to loop through all the elements of a container. However the behaviour is undefined if the container changes while the iterator is still looping.

```
hi = hashIteratorNew(ht);
 while (NULL != (key = hashIteratorNextKey(hi))) {
   do something with key;
 }
 hashIteratorDelete(hi);
```

Parameters:

ht the hashtable for which the new iterator object has to be returned.

Returns:

the new hash iterator object for the hashtable ht.

Definition at line 528 of file hashtable.c.

5.4.2.13 Pointer hashIteratorNextKey (HashIterator *hi*)

returns the next key of a hash-iterator.

Hash iterator points to the following entry afterwards

Parameters:
hi the hash iterator whose next key has to be returned.

Returns:
the next key of the hash iterator hi

Definition at line 552 of file hashtable.c.

5.4.2.14 Pointer hashIteratorNextValue (HashIterator *hi*)

returns the next value of a hash-iterator.

Hash iterator points to the following entry afterwards.

Parameters:
hi the hash iterator whose next value has to be returned.

Returns:
the next value of a hash-iterator hi.

Definition at line 577 of file hashtable.c.

5.4.2.15 List hashListOfKeys (Hashtable *ht*)

retrieves a list of keys of all objects in the hashtable.

Parameters:
ht the hashtable from which the list of keys have to be retrieved.

Returns:
the list of the keys of all objects in the hashtable.

Definition at line 498 of file hashtable.c.

References listPrependElement().

Referenced by strFinalize().

5.4.2.16 Hashtable hashNew (int *capacity*, double *loadFactor*, IntFunction * *hashFunction*, IntFunction * *keyEqualFunction*)

creates a new hashtable with an initial capacity of c.

whenever the number of stored objects exceeds *loadFactor* times the current capacity, the hashtable is automatically resized.

Parameters:
capacity specifies the capacity of the new hashtable to be created.

loadFactor it is a number that should be chosen between 0.5 and 0.9.

hashFunction it is that function which is called with a key as the only parameter and returns the hash value of $*$ that key.

keyEqualFunction It is that function which is called with two keys as parameters.It returns 1 if the keys are to be considered equal and 0 otherwise.

Returns:
the new hashtable with a capacity of c.

Definition at line 136 of file hashtable.c.

References primeNext().

Referenced by strInitialize().

5.4.2.17 Pointer hashRemove (Hashtable *ht*, Pointer *key*)

removes key/value pair from hashtable

Parameters:
ht the hashtable from which the key/value pair is to be removed.

key this shows the value that has to be deleted in the hashtable ht.

Returns:
the object (or *NULL*) that is stored under the key in the hashtable ht.

Definition at line 277 of file hashtable.c.

Referenced by strDelete().

5.4.2.18 Pointer hashSet (Hashtable *ht*, Pointer *key*, Pointer *value*)

adds the object value with the key *key* in the hashtable.

rehashes the hashtable if necessary.

Parameters:
ht the hashtable in which the object value has to be added with the key.

key the key value that has to be added with the *value*

value the object whose value has to be added to the key.

Returns:
the old object that was stored at that key or *NULL*

Definition at line 172 of file hashtable.c.

References rehashHashtable().

Referenced by strRegister().

5.4.2.19 int hashSize (Hashtable *ht*)

retrieves the size of the hashtable

Parameters:
ht the hashtable whose size has to be retrieved.

Returns:
the number of objects that are currently stored in the hashtable.

Definition at line 313 of file hashtable.c.

Referenced by strFinalize(), and strStoreSize().

5.4.2.20 int hashStringEqualFunction (char * *s*, char * *t*)

is an example equality function for C strings that can be used in hashNew.

Parameters:
s the first string that is used in the string comparison function.

t the second string that is used in the string comparison function.

Returns:
0 if they are equal and 1 if not.

Definition at line 649 of file hashtable.c.

Referenced by strInitialize().

5.4.2.21 int hashStringHashFunction (char * *s*)

is an example has function for C strings that can be used in hashNew.

A bit rotating function by Knuth is used here. TODO:

- try different hash functions
- strlen should go out - supply length of key.

Parameters:
s the string on which the hashStringHashFunction is performed.

Returns:
the integer representation of the string.

Definition at line 620 of file hashtable.c.

Referenced by strInitialize().

5.4.2.22 void rehashHashtable (Hashtable *ht*)

rehashes the hashtable.

doubles the capacity and this enlarges the space.

Parameters:

ht the hash table that is to be resized

Returns:

the new resized hashtable.

Definition at line 86 of file hashtable.c.

References primeNext().

Referenced by hashSet().

5.5 Lists

5.5.1 Detailed Description

A list container.

A list is a sequence of data objects where the data items are usually access from the beginning of list. The head of a list is the first data item and the tail is a list containing the remaining objects.

Data Structures

- struct ListStruct

  a list node.

Defines

- #define newListCell (List)memMalloc(sizeof(ListStruct))

  this is used to debug the cell allocation for Lists.

- #define freeListCell memFree

  this is used to debug the cell deallocation for Lists.

Functions

- List listNew ()

  get a new list.

- Pointer listElement (List l)

  returns first item in list.

- List listNext (List l)

  returns the tail of the list.

- Pointer listSetElement (List l, Pointer value)

  set the item of the current list cell.

- Pointer listSetNext (List l, List m)

  set the next of the current list cell.

- List listClone (List l)

  clones a list.

- List listDeepClone (List l, PointerFunction *p)

  Clones a list, performing a deep copy of all items via P.

- List listCopy (List dst, List src)

  set list-items of dst to those of src by reusing old buckets.

- List listAppendList (List front, List rear)
  appends a list to another list.
- List listAppendElement (List l, Pointer item)
  appends an item to the end of a list.
- List listPrependElement (List l, Pointer item)
  prepends an item to a list.
- List listAppendElements (List oldList,...)
  appends a couple of items to the end of a list.
- List listPrependElements (List oldList,...)
  prepends a couple of items to the end of a list.
- List listInsertSorted (List list, Pointer item, BooleanFunction *f)
  inserts an item keeping an order defined by f.
- List listInsertSortedWithData (List list, Pointer item, BooleanFunction *f, Pointer clientData)
  inserts an item keeping an order defined by f and some extra data.
- List listAddUniqueElement (List l, Pointer item)
  adds an item to the list if and only if it is not already present.
- int listSize (List l)
  retrieves number of items in the list.
- Pointer listNthElement (List l, int n)
  returns nth item in list.
- Pointer listLastElement (List l)
  returns last item in list.
- Boolean listContains (List l, Pointer p)
  checks if the list contains a particular item.
- void listForEach (List l, VoidFunction *f)
  calls function 'f' for each list element.
- void listForEachDelete (List l, VoidFunction *f)
  like listForEach, but frees list, list becomes inaccessible.
- List listFilter (List l, BooleanFunction *f)
  filters list, returns new (sub-)list.
- void listDelete (List l)
  frees list, does NOT free elements.
- List listDeleteElement (List l, Pointer p)

deletes all occurences of item from list.

- List listDeleteLastElement (List l)
  deletes the last item from the list.
- Vector listToVector (List l)
  converts a list into a vector.
- int listIndex (List l, Pointer item)
  find a particular item in a list .
- Boolean listIsEqual (List list1, List list2)
  compare two lists item per item.
- List listSort (List l, BooleanFunction *f)
  sorts a list, using a user-specified compare function.
- List listSortWithData (List l, BooleanFunction *f, void *data)
  sorts a list, using a user-specified compare function and some data.
- List listReverse (List l)
  return a new reverse list.

5.5.2 Define Documentation

5.5.2.1 #define freeListCell memFree

this is used to debug the cell deallocation for Lists.

It expands to _freeCell when BLAH is compiled with -DLIST_DEBUG or to the normal memFree function.

See also:
newCell

Definition at line 86 of file list.c.

Referenced by listDelete(), listDeleteElement(), listDeleteLastElement(), and listForEachDelete().

5.5.2.2 #define newListCell (List)memMalloc(sizeof(ListStruct))

this is used to debug the cell allocation for Lists.

It expands to _newCell when BLAH is compiled with -DLIST_DEBUG or to the normal memMalloc function.

See also:
_newCell, freeListCell

Definition at line 66 of file list.c.

Referenced by listAppendElement(), listClone(), listDeepClone(), listInsertSorted(), listInsertSortedWithData(), and listPrependElement().

5.5.3 Function Documentation

5.5.3.1 List listAddUniqueElement (List *l*, Pointer *item*)

adds an item to the list if and only if it is not already present.

Parameters:

l the list to which the specified item has to be added.

item the item that has to be added to the list l.

Returns:

the new list after the addition of the item if its unique.

Definition at line 420 of file list.c.

References listAppendElement(), and listContains().

5.5.3.2 List listAppendElement (List *l*, Pointer *item*)

appends an item to the end of a list.

Parameters:

l the list to which an item has to be appended.

item the item that has to be appended to the end of the list l.

Returns:

the new list after appending.

Definition at line 269 of file list.c.

References newListCell.

Referenced by laInsert(), listAddUniqueElement(), listAppendElements(), listFilter(), and strAppend().

5.5.3.3 List listAppendElements (List *oldList*, ...)

appends a couple of items to the end of a list.

Parameters:

oldList the list to which a couple of items are to be appended.

... the list of items are to be appended to the end of the list.

Definition at line 314 of file list.c.

References listAppendElement().

5.5.3.4 List listAppendList (List *front*, List *rear*)

appends a list to another list.

Parameters:

front a list to whose end another list rear is appended.

rear a list which is appended to the tail of the front list

Returns:
the new list after appending.

Definition at line 247 of file list.c.

5.5.3.5 List listClone (List *l*)

clones a list.

This doesn't clone the items.

Parameters:
l the list that has to be cloned.

Returns:
a newly allocated list after cloning.

Definition at line 158 of file list.c.

References newListCell.

Referenced by listCopy().

5.5.3.6 Boolean listContains (List *l*, Pointer *p*)

checks if the list contains a particular item.

Parameters:
l the list in which the existance of the specified item has to be checked.

p specifies the item that has to be checked for in the list l.

Returns:
TRUE if the list contains the item or *FALSE* otherwise.

Definition at line 490 of file list.c.

Referenced by listAddUniqueElement().

5.5.3.7 List listCopy (List *dst*, List *src*)

set list-items of dst to those of src by reusing old buckets.

It works as follows:

- if dst has more buckets than src then these are freed
- if dst has less buckets than src then new once are allocated
- if dst is NULL then listCopy is equivalent to listClone
- if src is NULL then listCopy is equivalent to listDelete

Parameters:

dst the resulting list

src the originating list

Returns:

the dst List

Definition at line 204 of file list.c.

References listClone(), and listDelete().

5.5.3.8 List listDeepClone (List *l*, PointerFunction * *p*)

Clones a list, performing a deep copy of all items via P.

Definition at line 176 of file list.c.

References newListCell.

5.5.3.9 void listDelete (List *l*)

frees list, does NOT free elements.

Parameters:

l the list that needs to be freed.

Returns:

nothing.

Definition at line 558 of file list.c.

References freeListCell.

Referenced by laInsert(), listCopy(), and strAppend().

5.5.3.10 List listDeleteElement (List *l*, Pointer *p*)

deletes all occurences of item from list.

Parameters:

l the list from which all the occurances of a specified item has to be deleted.

p the particular item whose occurances in the list l has to be deleted.

Returns:

the new list after deleting the item from the list.

Definition at line 576 of file list.c.

References freeListCell.

5.5.3.11 List listDeleteLastElement (List *l*)

deletes the last item from the list.

Parameters:
l the list from which the last item has to be deleted.

Returns:
the new list after deleting the last item form the list l.

Definition at line 612 of file list.c.

References freeListCell.

5.5.3.12 Pointer listElement (List *l*)

returns first item in list.

Parameters:
l the list whose first item has to be returned.

Returns:
the first item in the list l.

Definition at line 109 of file list.c.

Referenced by laBest(), laDelete(), laInsert(), laIteratorNew(), laIteratorNextElement(), laRemoveBest(), listReverse(), strFinalize(), and strFromList().

5.5.3.13 List listFilter (List *l*, BooleanFunction * *f*)

filters list, returns new (sub-)list.

Parameters:
l the list on which the function is evaluated.
f the function that is evaluated in the list l to generate a (sub-) list

Returns:
an independent list consisting of all the elements of the given list l that make function 'f' evaluate to *TRUE*

Definition at line 539 of file list.c.

References listAppendElement().

5.5.3.14 void listForEach (List *l*, VoidFunction * *f*)

calls function 'f' for each list element.

Parameters:
l the list in which the function 'f' is called.
f the function which is called for each element in the list l.

Definition at line 504 of file list.c.

5.5.3.15 void listForEachDelete (List *l*, VoidFunction * *f*)

like listForEach, but frees list, list becomes inaccessible.

Parameters:

l the list in which the function 'f' has to be called.

f the function that is called for each element in the list l.

Definition at line 518 of file list.c.

References freeListCell.

5.5.3.16 int listIndex (List *l*, Pointer *item*)

find a particular item in a list .

Parameters:

l the list from which the specified item has to be found.

item the item that has to be found from the list l.

Returns:

the index of the item if found else zero.

Definition at line 666 of file list.c.

5.5.3.17 List listInsertSorted (List *list*, Pointer *item*, BooleanFunction * *f*)

inserts an item keeping an order defined by f.

Parameters:

list the list into which a specified item has to be inserted.

item the item that has to be inserted into the list l.

f the function that defines the ordering of the during insertion.

Returns:

the new list after insertion operation.

Definition at line 359 of file list.c.

References newListCell.

5.5.3.18 List listInsertSortedWithData (List *list*, Pointer *item*, BooleanFunction * *f*, Pointer *clientData*)

inserts an item keeping an order defined by f and some extra data.

Parameters:

list the list into which a specified item has to be inserted.

item the item that has to be inserted into the list l.

'f' the function that defines the ordering of the during insertion.

clientData the data that determines the ordering along with the function 'f'

Returns:
the new list after insertion operation.

Definition at line 390 of file list.c.

References newListCell.

5.5.3.19 Boolean listIsEqual (List *list1*, List *list2*)

compare two lists item per item.

Parameters:
list1 the first list whose items are to be compared.

list2 the second list whose items are compared with the list 1.

Returns:
TRUE if the items are all equal and *FALSE* otherwise.

Definition at line 689 of file list.c.

References listSize().

5.5.3.20 Pointer listLastElement (List *l*)

returns last item in list.

Parameters:
l the list from which the last item has to be retrieved.

Returns:
the last item of the list l.

Definition at line 471 of file list.c.

5.5.3.21 List listNew ()

get a new list.

A new List is represented by NULL.

Returns:
NULL

Definition at line 98 of file list.c.

5.5.3.22 List listNext (List *l*)

returns the tail of the list.

Parameters:
l the list whose tail has to be returned.

Returns:
the last item of the list.

Definition at line 120 of file list.c.

Referenced by laDelete(), laInsert(), laIteratorNextElement(), laRemoveBest(), listReverse(), strFinalize(), and strFromList().

5.5.3.23 Pointer listNthElement (List *l*, int *n*)

returns nth item in list.

Parameters:
l the list whose 'n'th item has to be retrieved.

n the number that idicates the item that has to be retrieved.

Returns:
the item in the 'n'th position of the list.

Definition at line 453 of file list.c.

5.5.3.24 List listPrependElement (List *l*, Pointer *item*)

prepends an item to a list.

Parameters:
l the list to which an item has to be prepended.

item the item that has to be prepended to the list l.

Returns:
the new list after prepending.

Definition at line 296 of file list.c.

References newListCell.

Referenced by hashListOfKeys(), laInsert(), laIteratorNew(), listPrependElements(), listReverse(), and vectorToList().

5.5.3.25 List listPrependElements (List *oldList*, ...)

prepends a couple of items to the end of a list.

Parameters:
oldList the list to which the items are prepended.

... the list of items that are to be prepended to the list l.

Returns:
the new list after prepending.

Definition at line 336 of file list.c.

References listPrependElement().

5.5.3.26 List listReverse (List *l*)

return a new reverse list.

Parameters:
l the list whose data items are to be reversed.

Returns:
a new list after reversing the list l.

Definition at line 753 of file list.c.

References listElement(), listNext(), and listPrependElement().

5.5.3.27 Pointer listSetElement (List *l*, Pointer *value*)

set the item of the current list cell.

Parameters:
l the list from which the specified item has to be set.

value the value of the item in the list that has to be set.

Returns:
the list after setting the specified item.

Definition at line 132 of file list.c.

Referenced by laIteratorNextElement().

5.5.3.28 Pointer listSetNext (List *l*, List *m*)

set the next of the current list cell.

Parameters:
l the list from which the specified item has to be set.

m next of the current list cell

Definition at line 144 of file list.c.

Referenced by laInsert(), and laIteratorNextElement().

5.5.3.29 int listSize (List *l*)

retrieves number of items in the list.

Parameters:
l the list whose size has to be retrieved.

Returns:
the total number of items in the list l.

Definition at line 435 of file list.c.

Referenced by listIsEqual(), listToVector(), strFinalize(), and strFromList().

5.5.3.30 List listSort (List *l*, BooleanFunction * *f*)

sorts a list, using a user-specified compare function.

Parameters:
l the list that has to be sorted using the function 'f'
f the compare function that is used for the sorting of the list l.

Returns:
the new list after sorting using the function 'f'

Definition at line 713 of file list.c.

References listToVector(), vectorDelete(), vectorSort(), and vectorToList().

5.5.3.31 List listSortWithData (List *l*, BooleanFunction * *f*, void * *data*)

sorts a list, using a user-specified compare function and some data.

Parameters:
l the list that has to be sorted using the function 'f'
f the compare function that is used for the sorting of the list l.
data the data that defines the sorting along with the function 'f'

Returns:
the new list after sorting using the function 'f' and the data.

Definition at line 734 of file list.c.

References listToVector(), vectorDelete(), vectorSortWithData(), and vectorToList().

5.5.3.32 Vector listToVector (List *l*)

converts a list into a vector.

Parameters:
l the list that has to be converted into a vector v.

Returns:
the vector v corresponding to the list l.

Definition at line 642 of file list.c.

References listSize(), vectorNew(), and vectorSetElement().

Referenced by listSort(), and listSortWithData().

5.6 ListAgenda

5.6.1 Detailed Description

This module is an implementation of the agenda interface using a plain list as its the storage medium.

This is used by the module netsearch, but has been superseded by the more efficienct module treeagenda.

Data Structures

- struct ListAgendaEntryStruct
  this type represents an entry of an agenda.
- struct ListAgendaStruct
  quick, should be binary tree.

Functions

- ListAgenda laNew (int maxsize, VoidFunction f)
  creates a new ListAgenda.
- Boolean laSetVerbosity (ListAgenda a, Boolean b)
  sets verbosity flag.
- Boolean laVerbosity (ListAgenda a)
  gets verbosity flag.
- int laSize (ListAgenda a)
  retrieves size of the agenda.
- int laMaxSize (ListAgenda a)
  retrieves the size limit of the agenda.
- int laMaxSizeSoFar (ListAgenda a)
  retrieves the largest attained size of the agenda.
- Boolean laIsEmpty (ListAgenda a)
  checks if the specified ListAgenda is empty.
- Boolean laIsTruncated (ListAgenda a)
  checks for the truncation of the ListAgenda a.
- Boolean laResetTruncated (ListAgenda a)
  reset the agenda truncation warning.
- Boolean laInsert (ListAgenda a, double score, Pointer state)
  inserts a new entry into the agenda.

- Pointer laBest (ListAgenda a)

  returns the best entry (= first) from the agenda.

- Pointer laRemoveBest (ListAgenda a)

  removes and returns best entry (= first) from the agenda.

- void laDelete (ListAgenda a)

  deletes the specified agenda.

- ListAgendaIterator laIteratorNew (ListAgenda a)

  creates a new iterator object.

- Pointer laIteratorNextElement (ListAgendaIterator ai)

  returns the next item in iterator object.

- void laIteratorDelete (ListAgendaIterator ai)

  deletes the iterator object.

5.6.2 Function Documentation

5.6.2.1 Pointer laBest (ListAgenda *a*)

returns the best entry (= first) from the agenda.

Parameters:

a the ListAgenda from which the best item has to be retrieved.

Returns:

the best item in *a* (or NULL if a is empty).The item remains in the agenda.

Definition at line 313 of file listagenda.c.

References listElement().

5.6.2.2 void laDelete (ListAgenda *a*)

deletes the specified agenda.

The function deallocates all the items in *a* using ∗freeState(). Then it deallocates all the entries and the agenda itself.

Parameters:

a the ListAgenda that has to be deleted.

Definition at line 357 of file listagenda.c.

References listElement(), and listNext().

5.6.2.3 Boolean laInsert (ListAgenda *a*, double *score*, Pointer *state*)

inserts a new entry into the agenda.

Parameters:

a the ListAgenda into which a new element has to be inserted.

score the *state* is sorted into the list according to the *score*.

state the item that is inserted into the ListAgenda *a*.

Returns:

FALSE if the agenda has been truncated and *TRUE* if we were able to insert the item without any unpleasant side effects. This is going to be reported only once.

If the insertion leads to an overflow, then the worst item from the list is removed. Note that the item to be removed may be the *state* itself. Definition at line 233 of file listagenda.c.

References listAppendElement(), listDelete(), listElement(), listNext(), listPrependElement(), and listSetNext().

5.6.2.4 Boolean laIsEmpty (ListAgenda *a*)

checks if the specified ListAgenda is empty.

Parameters:

a the ListAgenda for which emptiness has to be checked.

Returns:

TRUE if agenda *a* is empty and *FALSE* otherwise.

Definition at line 191 of file listagenda.c.

5.6.2.5 Boolean laIsTruncated (ListAgenda *a*)

checks for the truncation of the ListAgenda *a*.

Parameters:

a the ListAgenda on which the function *laIsTruncated* is performed.

Returns:

TRUE if the agenda is already truncated and *FALSE* otherwise.

Definition at line 202 of file listagenda.c.

5.6.2.6 void laIteratorDelete (ListAgendaIterator *ai*)

deletes the iterator object.

The function deallocates the list cell that was used by the iterator.

Parameters:

ai the iterator object that has to be deleted.

Definition at line 422 of file listagenda.c.

5.6.2.7 ListAgendaIterator laIteratorNew (ListAgenda *a*)

creates a new iterator object.

This is actually just a single new list cell that points to the first entry.

Parameters:
a the ListAgenda for which a new iterator has to be created.

Returns:
a new ListAgendaIterator that will return all items of *a* sorted by score.

Definition at line 380 of file listagenda.c.

References listElement(), and listPrependElement().

5.6.2.8 Pointer laIteratorNextElement (ListAgendaIterator *ai*)

returns the next item in iterator object.

Parameters:
ai the iterator of the ListAgenda *a*.

Returns:
the best item in the underlying agenda that was not already returned by the iterator.

Definition at line 399 of file listagenda.c.

References listElement(), listNext(), listSetElement(), and listSetNext().

5.6.2.9 int laMaxSize (ListAgenda *a*)

retrieves the size limit of the agenda.

Parameters:
a the ListAgenda for which the maximum size has to be determined.

Returns:
the max number of entries that can be held by the ListAgenda a.

Definition at line 169 of file listagenda.c.

5.6.2.10 int laMaxSizeSoFar (ListAgenda *a*)

retrieves the largest attained size of the agenda.

Parameters:
a the ListAgenda for which the largest size has to be retrieved.

Returns:
the maximum size occupied by the agenda *a* so far.

Definition at line 180 of file listagenda.c.

5.6.2.11 ListAgenda laNew (int *maxsize*, VoidFunction *f*)

creates a new ListAgenda.

Parameters:

maxsize the maximum capacity of the new ListAgenda that has to be created.

f the function that used for deallocating an element.

Returns:

a pointer to a new ListAgenda that can hold upto *maxsize* entries.

Definition at line 114 of file listagenda.c.

5.6.2.12 Pointer laRemoveBest (ListAgenda *a*)

removes and returns best entry (= first) from the agenda.

Parameters:

a the ListAgenda from which the best item has to be retrieved.

Returns:

the best item in *a.The* corresponding entry is removed and deallocated.
It must not be called on an empty agenda.

Definition at line 331 of file listagenda.c.

References listElement(), and listNext().

5.6.2.13 Boolean laResetTruncated (ListAgenda *a*)

reset the agenda truncation warning.

Parameters:

a the ListAgenda whose truncationWarning will be set to FALSE.

Returns:

the old value of the truncationWarning

Definition at line 213 of file listagenda.c.

5.6.2.14 Boolean laSetVerbosity (ListAgenda *a*, Boolean *b*)

sets verbosity flag.

Parameters:

a the ListAgenda whose verbosity property is set

b a Boolean which we set the verbosity to

Returns:

the old verbosity state

Definition at line 134 of file listagenda.c.

5.6.2.15 int laSize (ListAgenda *a*)

retrieves size of the agenda.

Parameters:
a the ListAgenda for which the size has to be retrieved.

Returns:
the current number of entries in the ListAgenda *a*.

Definition at line 158 of file listagenda.c.

5.6.2.16 Boolean laVerbosity (ListAgenda *a*)

gets verbosity flag.

Parameters:
a the ListAgenda whose verbosity property is set

Returns:
the verbosity state

Definition at line 147 of file listagenda.c.

5.7 Memory

5.7.1 Detailed Description

Some basic Operations with the memory are done in this module.

Functions

- void * memMallocCheck (void *ptr, char *file, int line)
  used for checking the memory allocation for pointers.
- void memFreeFunction (void *pointer)
  sometimes memFree must be a function

5.7.2 Function Documentation

5.7.2.1 void memFreeFunction (void * *pointer*)

sometimes memFree must be a function

Parameters:
pointer the pointer that determines the memory free function.

Definition at line 62 of file memory.c.

5.7.2.2 void* memMallocCheck (void * *ptr*, char * *file*, int *line*)

used for checking the memory allocation for pointers.

a function that checks whether pointer is NULL, used by a version of memMalloc (c. f. memMalloc.h)

Parameters:
ptr the ptr whose value has to be checked for NULL.
file the file that contains the pointer.
line the line in which the pointer exists.

Returns:
error message if pointer equals null else returns the pointer.

Definition at line 47 of file memory.c.

5.8 Primes

5.8.1 Detailed Description

Primes is a module that makes use of Rabin's Probablistic Primetest-Algorithm for generating the prime numbers equal to the Hash table entries.

Defines

- #define **drand48**() ((double) rand() / (double) RAND_MAX)
- #define **srand48**(x) (srand((x)))

Functions

- unsigned long addMod (unsigned long x, unsigned long y, unsigned long m)

  addition in a modulo.

- unsigned long multMod (unsigned long x, unsigned long y, unsigned long m)

  multiplication in a modulo.

- unsigned long powMod (unsigned long x, unsigned long y, unsigned long m)

  exponent in a modulo.

- int primeRabin (unsigned long number, unsigned long times)

  Rabin's probablistic primetest-algorithm.

- unsigned long primeNext (unsigned long number, unsigned long times)

  returns the next prime after a given number.

5.8.2 Function Documentation

5.8.2.1 unsigned long addMod (unsigned long *x*, unsigned long *y*, unsigned long *m*) [static]

addition in a modulo.

Parameters:

x the first element used in addition.

y the second element used in addition

m the modulud function

Returns:

the added result.

Definition at line 51 of file primes.c.

Referenced by multMod().

5.8.2.2 unsigned long multMod (unsigned long *x*, unsigned long *y*, unsigned long *m*) [static]

multiplication in a modulo.

Parameters:

x the first element used in multiplication.

y the second element used in multiplication.

m the modulo function.

Returns:

the multiplied result.

Definition at line 64 of file primes.c.

References addMod().

Referenced by powMod().

5.8.2.3 unsigned long powMod (unsigned long *x*, unsigned long *y*, unsigned long *m*) [static]

exponent in a modulo.

Parameters:

x the base function.

y the exponent function.

m the modulo function.

Returns:

the result - x raised to the exponential y.

Definition at line 87 of file primes.c.

References multMod().

Referenced by primeRabin().

5.8.2.4 unsigned long primeNext (unsigned long *number*, unsigned long *times*)

returns the next prime after a given number.

Definition at line 123 of file primes.c.

References primeRabin().

Referenced by hashNew(), and rehashHashtable().

5.8.2.5 int primeRabin (unsigned long *number*, unsigned long *times*)

Rabin's probablistic primetest-algorithm.

Parameters:

number ...

times ...

Returns:
true if number is a prime by testing it a few times

Definition at line 106 of file primes.c.

References powMod().

Referenced by primeNext().

5.9 Ringbuffers

5.9.1 Detailed Description

Implementation of ring buffers.

It is similar to the vector that stores a series of objects,though the head and the tail are attached inorder to enhance cyclic operations.

Data Structures

- struct RingBufferStruct

  internal representation of the ring buffer.

Functions

- RingBuffer rbNew (int capacity)

  creates a new ringbuffer.

- void rbDelete (RingBuffer rb)

  deletes a ringbuffer.

- Boolean rbIsEmpty (RingBuffer rb)

  checks if the given ringbuffer is empty.

- Boolean rbIsFull (RingBuffer rb)

  checks if a given ring buffer is full.

- int rbAddTopElement (RingBuffer rb, Pointer element)

  adds a new element to the top of a ring buffer.

- int rbAddBottomElement (RingBuffer rb, Pointer element)

  adds a new element to the bottom of a ring buffer.

- Pointer rbTopPeek (RingBuffer rb)

  returns the last inserted element without removing it.

- Pointer rbBottomPeek (RingBuffer rb)

  returns the first inserted element without removing it.

- int rbSize (RingBuffer rb)

  finds the current size of the buffer.

- int rbCapacity (RingBuffer rb)

  finds the maximum capacity of the specified buffer.

- Pointer rbRemoveTopElement (RingBuffer rb)

  removes the element from the top of the buffer.

- Pointer rbRemoveBottomElement (RingBuffer rb)
  removes the element from the bottom of the buffer.
- Boolean rbContains (RingBuffer rb, Pointer element)
  checks if the ringbuffer contains a specified element.
- RingBuffer rbCopy (RingBuffer dst, RingBuffer src)
  makes a copy of the container from src *to* dst.
- RingBuffer rbClone (RingBuffer src)
  makes a new copy of the ring buffer.
- void rbForEachWithData (RingBuffer rb, VoidFunction *f, Pointer data)
  iterate over all items in the ring buffer and apply a function
- void rbClear (RingBuffer rb)
  empties the buffer.

5.9.2 Function Documentation

5.9.2.1 int rbAddBottomElement (RingBuffer *rb*, Pointer *element*)

adds a new element to the bottom of a ring buffer.

Parameters:

rb ringbuffer to which addition has to be done.

element element that has to be added to the ring buffer.

Returns:

the number of elements in the buffer.

Definition at line 138 of file ringbuffer.c.

References rbIsFull().

5.9.2.2 int rbAddTopElement (RingBuffer *rb*, Pointer *element*)

adds a new element to the top of a ring buffer.

Parameters:

rb ringbuffer to which addition has to be done.

element element that has to be added to the ring buffer.

Returns:

the number of elements in the buffer

Definition at line 115 of file ringbuffer.c.

References rbIsFull().

5.9.2.3 Pointer rbBottomPeek (RingBuffer *rb*)

returns the first inserted element without removing it.

Parameters:
rb ringbuffer from which the bottom element is returned.

Returns:
the first inserted element from the ringbuffer rb.

Definition at line 180 of file ringbuffer.c.

References rbIsEmpty().

5.9.2.4 int rbCapacity (RingBuffer *rb*)

finds the maximum capacity of the specified buffer.

Parameters:
rb ringbuffer whose maximum capacity is returned.

Returns:
the maximum number of elements the buffer rb can hold.

Definition at line 207 of file ringbuffer.c.

5.9.2.5 void rbClear (RingBuffer *rb*)

empties the buffer.

Parameters:
rb refers to the buffer that is to be emptied.

Definition at line 338 of file ringbuffer.c.

5.9.2.6 RingBuffer rbClone (RingBuffer *src*)

makes a new copy of the ring buffer.

Parameters:
src source of the ringbuffer to be cloned.

Returns:
a new copy of a ring buffer after cloning.

Definition at line 306 of file ringbuffer.c.

References rbCopy(), and rbNew().

5.9.2.7 Boolean rbContains (RingBuffer *rb*, Pointer *element*)

checks if the ringbuffer contains a specified element.

Parameters:

rb ringbuffer in which the search has to be made.

element the element to be seached in the ringbuffer rb.

Returns:

TRUE if the buffer contains element, *FALSE* otherwise.

Definition at line 261 of file ringbuffer.c.

5.9.2.8 RingBuffer rbCopy (RingBuffer *dst*, RingBuffer *src*)

makes a copy of the container from *src* to *dst*.

Parameters:

src this refers to the source ring buffer that has to be copied to the destination buffer.

dst this refers to the destination buffer into which the copying has been done.

Returns:

the dst ringbuffer after copying.

Definition at line 283 of file ringbuffer.c.

Referenced by rbClone().

5.9.2.9 void rbDelete (RingBuffer *rb*)

deletes a ringbuffer.

Parameters:

rb the ringbuffer that is to be deleted.

Definition at line 80 of file ringbuffer.c.

5.9.2.10 void rbForEachWithData (RingBuffer *rb*, VoidFunction * *f*, Pointer *data*)

iterate over all items in the ring buffer and apply a function

Parameters:

rb this is the ring buffer in which the iteration is done.

f this is the function that is used for the iteration in the ring buffer rb.

data determines the data in the ringbuffer.

Definition at line 321 of file ringbuffer.c.

5.9.2.11 Boolean rbIsEmpty (RingBuffer *rb*)

checks if the given ringbuffer is empty.

Parameters:
rb ringbuffer that is to be checked.

Returns:
TRUE if the buffer is empty and *FALSE* otherwise.

Definition at line 92 of file ringbuffer.c.

Referenced by rbBottomPeek(), rbRemoveBottomElement(), rbRemoveTopElement(), and rbTopPeek().

5.9.2.12 Boolean rbIsFull (RingBuffer *rb*)

checks if a given ring buffer is full.

Parameters:
rb ringbuffer that is to be checked

Returns:
TRUE if buffer is full and *FALSE* otherwise.

Definition at line 103 of file ringbuffer.c.

Referenced by rbAddBottomElement(), and rbAddTopElement().

5.9.2.13 RingBuffer rbNew (int *capacity*)

creates a new ringbuffer.

Parameters:
capacity specifies the capacity with which the new ringbuffer has to be created.

Returns:
a new ringBuffer with the size *capacity*.

Definition at line 57 of file ringbuffer.c.

Referenced by rbClone().

5.9.2.14 Pointer rbRemoveBottomElement (RingBuffer *rb*)

removes the element from the bottom of the buffer.

Parameters:
rb ringbuffer from which the removal is done.

Returns:
the first inserted element in the buffer rb.

Definition at line 240 of file ringbuffer.c.

References rbIsEmpty().

5.9.2.15 Pointer rbRemoveTopElement (RingBuffer *rb*)

removes the element from the top of the buffer.

Parameters:
rb ringbuffer from which the removal is done.

Returns:
the last inserted element in the buffer rb.

Definition at line 218 of file ringbuffer.c.

References rbIsEmpty().

5.9.2.16 int rbSize (RingBuffer *rb*)

finds the current size of the buffer.

Parameters:
rb ringbuffer whose size is returned.

Returns:
the current number of elements in the buffer.

Definition at line 196 of file ringbuffer.c.

5.9.2.17 Pointer rbTopPeek (RingBuffer *rb*)

returns the last inserted element without removing it.

Parameters:
rb ringbuffer from which top element is returned.

Returns:
the last inserted element from the ring buffer rb.

Definition at line 161 of file ringbuffer.c.

References rbIsEmpty().

5.10 Strings

5.10.1 Detailed Description

implementation of strings.

Strings in C are represented by arrays of characters.The end of the string is marked with a special character, the null character, which is simply the character with the value 0. Whenever we write a string, enclosed in double quotes, C automatically creates an array of characters for us, containing that string, terminated by the *NULL* character.

FIXME: this comment is plain copy/pasted from cdg.c

The symbol table cdgSymbolTable)is used to share strings registered (cdgRegisterString()) to it. Sharing strings speeds up string comparison alot as we dont need strcmp for this any more. A pointer comarison suffices. So we store strings in a hash. But be warned: changing a registered string directly will break things seriously. If you need to change a registered string, make a copy of it (strCopy), manipulate it for your needs and then register it once again.

Data Structures

- struct SharedStringStruct

  strings with reference counters

Typedefs

- typedef SharedStringStruct * **SharedString**

Functions

- SharedString _strNewSharedString (const String str)

  allocate a new SharedString.

- void _strDeleteSharedString (SharedString)

  deallocated a SharedString.

- void _strDeleteStoreEntry (String, SharedString)

  deallocate a key value pair.

- SharedString _strLookup (String)

  lookup a string in the string store.

- String _strTryRegister (String)

  try to register a new string.

- String strCopy (const String s)

  this performs string copying function.

- String strVPrintf (const String fmt, va_list ap)

  returns a formated string.

- String strPrintf (const String fmt,...)
  returns a formated string.
- String strCat (const String a, const String b)
  concatenates two strings.
- String strFromList (List list)
  concatenates a list of strings.
- String strAppend (const String head,...)
  concatenates many strings together.
- void strDelete (String str)
  unregister a string This function tries deallocate the `str` *string when its reference counter licenses it.*
- String strRegister (const String str)
  register a string in symbol table.
- int strStoreSize (void)
  return the number of shared strings.
- void strFinalize (void)
  module finalization routine.
- String strDecode (String word)
  Translate a string from unicode UTF-8 to ISO-8859-1.
- void strInitialize (void)
  module initialization routine.

Variables

- Hashtable _strStore = NULL
  container for shared strings.
- iconv_t **_strConversionDescriptor** = (iconv_t)-1

5.10.2 Function Documentation

5.10.2.1 void _strDeleteSharedString (SharedString *sstr*) `[static]`

deallocated a SharedString.

This function deallocates a SharedString and its workload. Definition at line 335 of file string.c.

References SharedStringStruct::data.

Referenced by _strDeleteStoreEntry(), and strDelete().

5.10.2.2 void _strDeleteStoreEntry (String *key*, SharedString *value*) [static]

deallocate a key value pair.

This is used to deallocate the key and the value of the _strStore. Definition at line 326 of file string.c.

References _strDeleteSharedString().

Referenced by strFinalize().

5.10.2.3 SharedString _strLookup (String *str*) [static]

lookup a string in the string store.

This function returns a pointer to a SharedString if the given string argument is already shared, or NULL if this string isn't shared yet. Definition at line 169 of file string.c.

References _strStore, and hashGet().

Referenced by _strTryRegister(), strDelete(), and strRegister().

5.10.2.4 SharedString _strNewSharedString (const String *str*) [static]

allocate a new SharedString.

This function constructs a new SharedString. It contains no worload data yet. Definition at line 297 of file string.c.

References SharedStringStruct::counter, and SharedStringStruct::data.

Referenced by strRegister().

5.10.2.5 String _strTryRegister (String *str*) [static]

try to register a new string.

This function only registers new strings. It will not increase the reference counter of an already registered string. In any case it will return a known string. Definition at line 183 of file string.c.

References _strLookup(), and strRegister().

Referenced by strCat().

5.10.2.6 String strAppend (const String *head*, ...)

concatenates many strings together.

This function allocates the memory for the result string. The argument strings are not modified by the function. The last string in the argument list must be NULL.

Parameters:

str the head of the string to be produced

... represents the strings to be appended to the head.

Returns:

the new appended string.

Definition at line 272 of file string.c.

References listAppendElement(), listDelete(), and strFromList().

5.10.2.7 String strCat (const String *a*, const String *b*)

concatenates two strings.

This is our version of the standard unix strcat() with the differences that both arguments are `const` strings. A concatenated shared string of `a` and `b` is returned. Both arguments might be NULL.

Returns:
the target concatenated with the source.

Definition at line 202 of file string.c.

References _strTryRegister(), strPrintf(), and strRegister().

5.10.2.8 String strCopy (const String *s*)

this performs string copying function.

This function constructs a copy of the given source string. The returned string is not shared any more as its source might have been. So in order to manipulate a shared string, first strCopy() it, then alter it and strRegister() it finaly. While copying the string new memory is allocated for you. Take care of it.

Parameters:
s the string that has to be copied.

Returns:
the new copied string.

Definition at line 101 of file string.c.

Referenced by strFromList(), and strRegister().

5.10.2.9 String strDecode (String *word*)

Translate a string from unicode UTF-8 to ISO-8859-1.

The String will be left untouched if there is any problem while decoding. Definition at line 450 of file string.c.

References strRegister().

5.10.2.10 void strDelete (String *str*)

unregister a string This function tries deallocate the `str` string when its reference counter licenses it.

Note, that the pointer `str` might get invalid or not depending on the reference counter. Definition at line 353 of file string.c.

References _strDeleteSharedString(), _strLookup(), _strStore, SharedStringStruct::counter, and hashRemove().

5.10.2.11 void strFinalize (void)

module finalization routine.

This function is only called by blahInitialize() and should not be used from outside It basically deallocates the `_strStore`. It also closes the conversionHandler. Definition at line 418 of file string.c.

References _strDeleteStoreEntry(), _strStore, hashForEach(), hashForEachFree(), hashListOfKeys(), hashSize(), listElement(), listNext(), and listSize().

Referenced by blahFinalize().

5.10.2.12 String strFromList (List *list*)

concatenates a list of strings.

This function takes a list of strings and concatenates them together in a newly allocated string. Be sure that all list elements are realy of type string. We can't grant that here. If the list is NULL or empty NULL is returned to you. The return value is a registered string.

Parameters:
list of strings

Returns:
the new appended string.

Definition at line 230 of file string.c.

References listElement(), listNext(), listSize(), strCopy(), and strRegister().

Referenced by strAppend().

5.10.2.13 void strInitialize (void)

module initialization routine.

This function is only called by blahInitialize() and should not be used from outside It basically allocates the `_strStore`. It also sets the conversion handler for String decoding Definition at line 500 of file string.c.

References _strStore, hashNew(), hashStringEqualFunction(), and hashStringHashFunction().

Referenced by blahInitialize().

5.10.2.14 String strPrintf (const String *fmt*, ...)

returns a formated string.

This function basically has been taken from the sprintf() manual page. The differences between sprintf() and strPrintf() are that you don't have to bother about memory allocation. We allocate enuf memory to hold the formated result string. Further more then this string is strRegister()ed for you, so you might get an already shared string returned to you. Use strDelete() to indicate your lake of interest on the result string. Definition at line 152 of file string.c.

References strVPrintf().

Referenced by strCat().

5.10.2.15 String strRegister (const String *str*)

register a string in symbol table.

This function registers a string to be shared. This is done by copying it into the `_strStore` (leaving the argument string pointer untouched). If the string already exists in symbol table the _stored_ string is returned. If the string doesn't exist then the string is _copied_ and entered in the symbol table; the new string is returned. In no case the memory of string s is referenced by the symbol table. But the returned string is owned by the symbol table and will be shared by other references later on. So be careful and never change a registered string in place. Use strCopy() first to check out a copy of a shared string. If `str` is NULL then a registered empty string is returned, that is strRegister(NULL) == strRegister(""). Definition at line 380 of file string.c.

References _strLookup(), _strNewSharedString(), _strStore, SharedStringStruct::counter, SharedStringStruct::data, hashSet(), and strCopy().

Referenced by _strTryRegister(), strCat(), strDecode(), strFromList(), and strVPrintf().

5.10.2.16 int strStoreSize (void)

return the number of shared strings.

Definition at line 406 of file string.c.

References _strStore, and hashSize().

5.10.2.17 String strVPrintf (const String *fmt*, va_list *ap*)

returns a formated string.

This function is our version of vsprintf(). See strPrintf() for more information. Definition at line 117 of file string.c.

References strRegister().

Referenced by strPrintf().

5.10.3 Variable Documentation

5.10.3.1 Hashtable _strStore = NULL `[static]`

container for shared strings.

This global variable stores all registered strings Definition at line 61 of file string.c.

Referenced by _strLookup(), strDelete(), strFinalize(), strInitialize(), strRegister(), and strStoreSize().

5.11 TreeAgenda

5.11.1 Detailed Description

implemention of an agenda interface using an unbalanced binary search tree.

This module exports an agenda as used in the CDG netsearch module. The agenda manages a set of elements annoted with the ratings. Elements can be inserted according to their rating, and the first element can be retrieved.The agenda is not responsible for determining the rating of an element;it can only deal with pairs(subsequently called items) of elements and scores.

Todo

Actually the b-tree used here can be implemented in a more general way to be more usefull. An agenda is just one way to use a b-treeish storage organization.

Data Structures

- struct TANodeStruct

  this type represents an entry of an agenda.

- struct TreeAgendaStruct

  quick, should be binary tree.

- struct TreeAgendaIteratorStruct

  this structure instantiates the generic agenda iterator.

Defines

- #define **TA_DEBUG** 0

Functions

- void taPrintNode (TreeAgenda a, int index)

  prints a node.

- int taCheckNode (TreeAgenda a, int index, Boolean recursive, Boolean better)

  performs internal consistency checks on a TreeAgenda.

- int taDeleteNode (TreeAgenda a, int index, Boolean delete)

  deletes a node.

- TreeAgenda taNew (int maxsize, VoidFunction *f)

  creates a new agenda.

- Boolean taVerbosity (TreeAgenda a)

  gets verbosity flag.

- Boolean taSetVerbosity (TreeAgenda a, Boolean b)

sets verbosity flag.

- int taSize (TreeAgenda a)
  retrieves size of the agenda.
- int taMaxSize (TreeAgenda a)
  retrieves the size limit of the agenda.
- int taMaxSizeSoFar (TreeAgenda a)
  retrieves the largest attained size of the agenda.
- Boolean taIsEmpty (TreeAgenda a)
  checks if the specified TreeAgenda is empty.
- Boolean taIsTruncated (TreeAgenda a)
  checks for the truncation of the TreeAgenda a.
- Boolean taResetTruncated (TreeAgenda a)
  reset the agenda truncation warning.
- Boolean taInsert (TreeAgenda a, double score, Pointer state)
  inserts a new entry into the agenda.
- Pointer taBest (TreeAgenda a)
  returns the best entry (= first) from the agenda.
- Pointer taRemoveBest (TreeAgenda a)
  removes and returns best entry (= first) from the agenda.
- void taDelete (TreeAgenda a)
  deletes the specified agenda.
- TreeAgendaIterator taIteratorNew (TreeAgenda a)
  creates a new iterator object.
- Pointer taIteratorNextElement (TreeAgendaIterator ai)
  returns the next item in iterator object.
- void taIteratorDelete (TreeAgendaIterator ai)
  deletes the iterator object.

5.11.2 Function Documentation

5.11.2.1 Pointer taBest (TreeAgenda *a*)

returns the best entry (= first) from the agenda.

Parameters:

a the TreeAgenda from which the best item has to be retrieved.

Returns:
the best item in *a* (or NULL if a is empty).The item remains in the agenda.

Definition at line 533 of file treeagenda.c.

5.11.2.2 int taCheckNode (TreeAgenda *a*, int *index*, Boolean *recursive*, Boolean *better*)

performs internal consistency checks on a TreeAgenda.

This is only for debugging purposes.

Parameters:
a the TreeAgenda that has to be checked.
index the index of the node in the treeagenda.
recursive is *TRUE* if repetition occurs and *FALSE* otherwise.
better is *TRUE* if better consistency and *FALSE* otherwise.

Returns:
the number of nodes that are checked.

Definition at line 145 of file treeagenda.c.

References taPrintNode().

Referenced by taInsert().

5.11.2.3 void taDelete (TreeAgenda *a*)

deletes the specified agenda.

The function deallocates all the items in *a* using ∗freeState(). Then it deallocates all the entries and the agenda itself.

Parameters:
a the ListAgenda that has to be deleted.

Definition at line 625 of file treeagenda.c.

References taDeleteNode().

5.11.2.4 int taDeleteNode (TreeAgenda *a*, int *index*, Boolean *delete*)

deletes a node.

Parameters:
a the TreeAgenda in which a node has to be deleted.
index the index of the node that has to be deleted in TreeAgenda.
delete *TRUE* if the item itself is also deallocated and *FALSE* otherwise.

Definition at line 199 of file treeagenda.c.

Referenced by taDelete().

5.11.2.5 Boolean taInsert (TreeAgenda *a*, double *score*, Pointer *state*)

inserts a new entry into the agenda.

Parameters:

a the TreeAgenda into which a new element has to be inserted.

state the item that is inserted into the TreeAgenda *a*.

score the *state* is sorted into the list according to the *score*.

Returns:

FALSE if the agenda has been truncated and *TRUE* if we were able to insert the item without any unpleasant side effects. This is going to be reported only once.

If the insertion leads to an overflow,then one of the elements from the tree is deallocated.Note that the element to be removed may be the *state* itself. Definition at line 379 of file treeagenda.c.

References taCheckNode(), and taPrintNode().

5.11.2.6 Boolean taIsEmpty (TreeAgenda *a*)

checks if the specified TreeAgenda is empty.

Parameters:

a the TreeAgenda for which emptiness has to be checked.

Returns:

TRUE if agenda *a* is empty and *FALSE* otherwise.

Definition at line 337 of file treeagenda.c.

5.11.2.7 Boolean taIsTruncated (TreeAgenda *a*)

checks for the truncation of the TreeAgenda *a*.

Parameters:

a the TreeAgenda on which the function *taIsTruncated* is performed.

Returns:

TRUE if the agenda is already truncated and *FALSE* otherwise.

Definition at line 348 of file treeagenda.c.

5.11.2.8 void taIteratorDelete (TreeAgendaIterator *ai*)

deletes the iterator object.

The function deallocates the tree element that was used by the iterator.

Parameters:

ai the iterator object that has to be deleted.

Definition at line 702 of file treeagenda.c.

5.11.2.9 TreeAgendaIterator taIteratorNew (TreeAgenda *a*)

creates a new iterator object.

This function creates a new iterator for TreeAgenda *a*.

Parameters:
a the TreeAgenda for which a new iterator has to be created.

Returns:
a new TreeAgendaIterator that will return all items of *a* sorted by score.

Definition at line 649 of file treeagenda.c.

5.11.2.10 Pointer taIteratorNextElement (TreeAgendaIterator *ai*)

returns the next item in iterator object.

Parameters:
ai the iterator of the TreeAgenda *a*.

Returns:
the best item in the underlying agenda that was not already returned by the iterator.

Definition at line 666 of file treeagenda.c.

5.11.2.11 int taMaxSize (TreeAgenda *a*)

retrieves the size limit of the agenda.

Parameters:
a the TreeAgenda for which the maximum size has to be determined.

Returns:
the max number of entries that can be held by the TreeAgenda a.

Definition at line 315 of file treeagenda.c.

5.11.2.12 int taMaxSizeSoFar (TreeAgenda *a*)

retrieves the largest attained size of the agenda.

Parameters:
a the TreeAgenda for which the largest size has to be retrieved.

Returns:
the maximum size occupied by the agenda *a* so far.

Definition at line 326 of file treeagenda.c.

5.11.2.13 TreeAgenda taNew (int *maxsize*, VoidFunction * *f*)

creates a new agenda.

Parameters:

maxsize determines the desired maximal size of an agenda.

f the function for deallocating the element.

Returns:

a pointer to the new structure that can hold upto *maxsize* entries.

The agenda cannot determine whether it is safe to call this function on an element or not.Passing the wrong function to a taNew() results undefined behaviour. Definition at line 239 of file treeagenda.c.

5.11.2.14 void taPrintNode (TreeAgenda *a*, int *index*)

prints a node.

It is used by the taCheckNode().

Parameters:

a the TreeAgenda whose node has to be printed.

index the index of the node that has to be printed.

Definition at line 104 of file treeagenda.c.

Referenced by taCheckNode(), taInsert(), and taRemoveBest().

5.11.2.15 Pointer taRemoveBest (TreeAgenda *a*)

removes and returns best entry (= first) from the agenda.

Parameters:

a the TreeAgenda from which the best item has to be retrieved.

Returns:

the best item in *a.The* corresponding entry is removed and deallocated.
It must not be called on an empty agenda.

Definition at line 549 of file treeagenda.c.

References taPrintNode().

5.11.2.16 Boolean taResetTruncated (TreeAgenda *a*)

reset the agenda truncation warning.

Parameters:

a the TreeAgenda whose truncationWarning will be set to FALSE.

Returns:

the old value of the truncationWarning

Definition at line 359 of file treeagenda.c.

5.11.2.17 Boolean taSetVerbosity (TreeAgenda *a*, Boolean *b*)

sets verbosity flag.

Parameters:
a the TreeAgenda whose verbosity property is set
b a Boolean which we set the verbosity to

Returns:
the old verbosity state

Definition at line 289 of file treeagenda.c.

5.11.2.18 int taSize (TreeAgenda *a*)

retrieves size of the agenda.

Parameters:
a the TreeAgenda for which the size has to be retrieved.

Returns:
the current number of entries in the TreeAgenda *a*.

Definition at line 304 of file treeagenda.c.

5.11.2.19 Boolean taVerbosity (TreeAgenda *a*)

gets verbosity flag.

Parameters:
a the TreeAgenda whose verbosity property is set

Returns:
the verbosity state

Definition at line 278 of file treeagenda.c.

5.12 Vectors

5.12.1 Detailed Description

implementation of vectors.

A Vector is a one-dimensional array but allows for automatic resizing.i.e,the size need not be known in advance.

Data Structures

- struct VectorStruct

  internal representation of a vector

Defines

- #define **RESIZEFACTOR** 2
- #define **checkVector**(v, s)

Functions

- void resizeVector (Vector v)

  resize vector, double capacity.

- Vector vectorNew (int capacity)

  creates a new vector with an initial capacity.

- void vectorDelete (Vector v)

  deletes vector, but can't free memory for the content.

- int vectorAddElement (Vector v, void ∗element)

  adds a new element to the end of the vector.

- Pointer vectorElement (Vector v, int index)

  sets the element at a specific index to a new element.

- Pointer vectorRemoveElementAt (Vector v, int index)

  removes element at the index.

- int vectorRemoveElement (Vector v, void ∗element)

  removes the specified element from the vector.

- Pointer vectorInsertElement (Vector v, void ∗element, int index)

  inserts a new element at the specified index.

- Pointer vectorSetElement (Vector v, void ∗element, int index)

  sets the element at the given index.

- void vectorSetElements (Vector v, void *element, int from, int to)
  sets all elements between FROM *and* TO(excluding to) *to a new value.*
- void vectorSetAllElements (Vector v, void *element)
  sets all the elements to a new value.
- int vectorIndexOf (Vector v, void *element, int index)
  finds the index of an entry in the vector.
- int vectorCapacity (Vector v)
  returns the current capacity of vector.
- int vectorSize (Vector v)
  returns the current number of entries in the vector.
- Boolean vectorIsEmpty (Vector v)
  checks if the vector is empty or not.
- Boolean vectorContains (Vector v, void *element)
  checks if a vector contains a given element.
- Vector vectorClone (Vector v)
  clones a vector.
- Vector vectorCopy (Vector dst, Vector src)
  copies all the entries of one vector src *to another vector* dst.
- void doSorting (Vector v, BooleanFunction *f, void *data, char use)
  internal helper function for sorting.
- Vector vectorSort (Vector v, BooleanFunction *f)
  sorts a vector using a user-specified compare function.
- Vector vectorSortWithData (Vector v, BooleanFunction *f, void *clientData)
  sorts a vector, using a user-specified compare function and some extra data.
- List vectorToList (Vector v)
  converts a vector into a list.

5.12.2 Function Documentation

5.12.2.1 void doSorting (Vector *v*, BooleanFunction * *f*, void * *data*, char *use*) [static]

internal helper function for sorting.

Parameters:

v the vector in which the sorting operation has to be done.

f the function that is used for the sorting of the vector f.

data some additional data that is handed over to the comparison function.

use determines the usage in the program

Definition at line 500 of file vector.c.

References vectorElement(), vectorSetElement(), and vectorSize().

Referenced by vectorSort(), and vectorSortWithData().

5.12.2.2 void resizeVector (Vector *v*) [static]

resize vector, double capacity.

Parameters:

v the vector that has to be resized.

Returns:

the new vector after resizing.

Definition at line 85 of file vector.c.

Referenced by vectorAddElement(), vectorInsertElement(), vectorSetElement(), and vectorSetElements().

5.12.2.3 int vectorAddElement (Vector *v*, void * *element*)

adds a new element to the end of the vector.

if necessary the vector automatically increases its capacity.

Parameters:

v the vector to which the new element has to be added.

element the element that has to be added to the vector v.

Returns:

the index of the vector after addition.

Definition at line 147 of file vector.c.

References resizeVector().

Referenced by arrayNew().

5.12.2.4 int vectorCapacity (Vector *v*)

returns the current capacity of vector.

Parameters:

v the vector whose capacity has to be retrieved.

Returns:

the capacity of the vector v.

Definition at line 396 of file vector.c.

Referenced by vectorClone().

5.12.2.5 Vector vectorClone (Vector *v*)

clones a vector.

Parameters:
v the vector that has to be cloned.

Returns:
a new vector after cloning is done.

Definition at line 455 of file vector.c.

References vectorCapacity(), vectorCopy(), and vectorNew().

Referenced by arrayClone().

5.12.2.6 Boolean vectorContains (Vector *v*, void * *element*)

checks if a vector contains a given element.

Parameters:
v the vector in which a particular number has to be searched.
element the element which has to be checked in the vector v.

Returns:
TRUE if vector contains the element and *FALSE* otherwise.

Definition at line 436 of file vector.c.

5.12.2.7 Vector vectorCopy (Vector *dst*, Vector *src*)

copies all the entries of one vector *src* to another vector *dst*.

Parameters:
src source vector from which the copying is done.
dst destination vector to which copying is done.

Returns:
the dst vector with the entries copied.

Definition at line 473 of file vector.c.

Referenced by vectorClone().

5.12.2.8 void vectorDelete (Vector *v*)

deletes vector, but can't free memory for the content.

Parameters:
v the vector that has to be deleted.

Definition at line 130 of file vector.c.

Referenced by arrayDelete(), listSort(), and listSortWithData().

5.12.2.9 Pointer vectorElement (Vector *v*, int *index*)

sets the element at a specific index to a new element.

if necessary the vector automatically increases its capacity.

Parameters:

v the vector in which the element has to be set.

index the index at which the element that has to be set is located.

Returns:

the old element (or *NULL*) at that index.

Definition at line 168 of file vector.c.

Referenced by arrayDimension(), arrayElement(), arraySetElement(), doSorting(), and vectorToList().

5.12.2.10 int vectorIndexOf (Vector *v*, void * *element*, int *index*)

finds the index of an entry in the vector.

Parameters:

v the vector in which the index of the element has to be retrieved.

element the element whose index has to be retrieved in the vector v.

index determines the index of all the elements in the vector.

Returns:

index of a given element in the vector or '-1' if the element does not belong to that specified vector.

Definition at line 377 of file vector.c.

Referenced by vectorRemoveElement().

5.12.2.11 Pointer vectorInsertElement (Vector *v*, void * *element*, int *index*)

inserts a new element at the specified index.

Size increases automatically if necessary. Inefficient method,Not recommended.

Parameters:

v the vector into which the new element has to be added.

element the element that has to be added in the vector v.

index the specified index at which the insertion has to be done.

Returns:

the old element at that index (or *NULL*).

Definition at line 249 of file vector.c.

References resizeVector().

5.12.2.12 Boolean vectorIsEmpty (Vector *v*)

checks if the vector is empty or not.

Parameters:
v the vector for which the emptiness is checked.

Returns:
TRUE if the vector is empty and *FALSE* otherwise.

Definition at line 422 of file vector.c.

5.12.2.13 Vector vectorNew (int *capacity*)

creates a new vector with an initial capacity.

specifying a correct or nearly correct capacity may slightly improve the efficiency.

Parameters:
capacity the capacity with which the new vector has to be created.

Returns:
the new vector created with size capacity.

Definition at line 104 of file vector.c.

Referenced by arrayNew(), listToVector(), and vectorClone().

5.12.2.14 int vectorRemoveElement (Vector *v*, void * *element*)

removes the specified element from the vector.

If the element occurs multiple times, the first item will be removed. If the element does not belong to the vector the program is aborted. Not efficient,not recommended.

Parameters:
v the vector from which the element to be removed is present.
element the element that has to be removed from the vector v.

Returns:
the associated index of the vector after removal.

Definition at line 221 of file vector.c.

References vectorIndexOf(), and vectorRemoveElementAt().

5.12.2.15 Pointer vectorRemoveElementAt (Vector *v*, int *index*)

removes element at the index.

all later elements move one position in front and size decreases. inefficient, not recommended.

Parameters:
v the vector from which the element has to be removed.

index the index at which the element that has to be removed is located.

Returns:
the old element at the specified index.

Definition at line 189 of file vector.c.

Referenced by vectorRemoveElement().

5.12.2.16 void vectorSetAllElements (Vector *v*, void * *element*)

sets all the elements to a new value.

Size increases to the current capacity of the vector.

Parameters:
v the vector in which all the elements have to be set to the new value.
element determines the element that is being set in vector v.

Definition at line 361 of file vector.c.

References vectorSetElements().

5.12.2.17 Pointer vectorSetElement (Vector *v*, void * *element*, int *index*)

sets the element at the given index.

size may increase automatically if necessary.

Parameters:
v the vector in which the specified element has to be set.
element the element that has to be set in the vector v.
index the index at which the element has to be set.

Returns:
the old element at that index.

Definition at line 285 of file vector.c.

References resizeVector().

Referenced by doSorting(), and listToVector().

5.12.2.18 void vectorSetElements (Vector *v*, void * *element*, int *from*, int *to*)

sets all elements between *FROM* and *TO(excluding to)* to a new value.

size may increase automatically if necessary.

Parameters:
v the vector in which the elements have to be set.
element determines the element that is being set.
from this is the source index from which the elements are set.

to this is the destination index until which the elements are set.

Returns:
the new vector after setting the elements.

Definition at line 316 of file vector.c.

References resizeVector().

Referenced by vectorSetAllElements().

5.12.2.19 int vectorSize (Vector *v*)

returns the current number of entries in the vector.

Parameters:
v the vector whose current number of entries has to be determined.

Returns:
the total number of entries in the vector v.

Definition at line 409 of file vector.c.

Referenced by arrayDimension(), arrayElement(), arraySetElement(), doSorting(), and vectorToList().

5.12.2.20 Vector vectorSort (Vector *v*, BooleanFunction * *f*)

sorts a vector using a user-specified compare function.

The function f is called as f(a,b)

Parameters:
v the vector that has to be sorted.
f the function that is used for sorting.

Returns:
should return *TRUE* if the element *a* should be before element *b*.

Definition at line 536 of file vector.c.

References doSorting().

Referenced by listSort().

5.12.2.21 Vector vectorSortWithData (Vector *v*, BooleanFunction * *f*, void * *clientData*)

sorts a vector, using a user-specified compare function and some extra data.

The function f is called as f(a,b,)and some data.

Parameters:
v the vector that has to be sorted.
f the function that is used for sorting.
clientData the additional data that is handed over to the comparison function.

Returns:
should return *TRUE* if the element *a* should be before element *b*

Definition at line 554 of file vector.c.

References doSorting().

Referenced by listSortWithData().

5.12.2.22 List vectorToList (Vector *v*)

converts a vector into a list.

Parameters:
v the vector that has to be converted into a list l.

Returns:
the list l corresponding to the vector v.

Definition at line 568 of file vector.c.

References listPrependElement(), vectorElement(), and vectorSize().

Referenced by listSort(), and listSortWithData().

5.13 Main module

5.13.1 Detailed Description

This is the main module for the BLAH library.

By now it provides the initialization and finalization of the library. So call blahInitialize() before using any functions of this library. You might want to call blahFinalize() before exiting your application.

Functions

- void blahInitialize (void)

  initialize the blah library.

- void blahFinalize (void)

  finalize the blah library.

5.13.2 Function Documentation

5.13.2.1 void blahFinalize (void)

finalize the blah library.

This will call the finalization routines of those modules that might need a finalization. Definition at line 53 of file blah.c.

References strFinalize().

5.13.2.2 void blahInitialize (void)

initialize the blah library.

This will call the initlialization routines of those modules that need to be initialized at application start. Definition at line 43 of file blah.c.

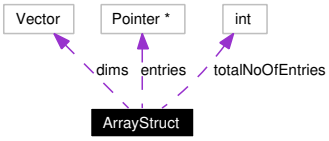
References strInitialize().

Chapter 6

BLAH Data Structure Documentation

6.1 ArrayStruct Struct Reference

Collaboration diagram for ArrayStruct:



6.1.1 Detailed Description

internal structure of an array.

Definition at line 53 of file array.c.

Data Fields

- Vector dims
  vector of dimensions
- int totalNoOfEntries
  product of dimensions
- Pointer * entries
  table of entries

6.1.2 Field Documentation

6.1.2.1 Vector ArrayStruct::dims

vector of dimensions

Definition at line 54 of file array.c.

6.1.2.2 Pointer* ArrayStruct::entries

table of entries

Definition at line 56 of file array.c.

6.1.2.3 int ArrayStruct::totalNoOfEntries

product of dimensions

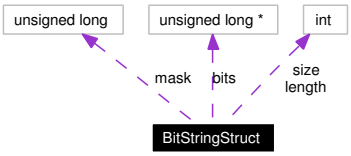
Definition at line 55 of file array.c.

The documentation for this struct was generated from the following file:

- array.c

6.2 BitStringStruct Struct Reference

Collaboration diagram for BitStringStruct:



6.2.1 Detailed Description

internal structure of a string of bits.

Definition at line 71 of file bitstring.c.

Data Fields

- int size
  size of bitstring in bits
- int length
  length of array; length=capacity/BITS_PER_LONG
- unsigned long mask
  mask for the last unsigned long
- unsigned long * bits
  array of longs

6.2.2 Field Documentation

6.2.2.1 unsigned long* BitStringStruct::bits

array of longs

Definition at line 80 of file bitstring.c.

6.2.2.2 int BitStringStruct::length

length of array; length=capacity/BITS_PER_LONG

Definition at line 77 of file bitstring.c.

6.2.2.3 unsigned long BitStringStruct::mask

mask for the last unsigned long

Definition at line 79 of file bitstring.c.

6.2.2.4 int BitStringStruct::size

size of bitstring in bits

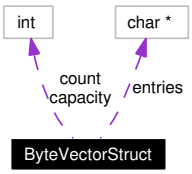
Definition at line 76 of file bitstring.c.

The documentation for this struct was generated from the following file:

- bitstring.c

6.3 ByteVectorStruct Struct Reference

Collaboration diagram for ByteVectorStruct:



6.3.1 Detailed Description

internal representation of a bit vector

Definition at line 47 of file bitvector-old.c.

Data Fields

- int count
  number of entries
- int capacity
  capacity of table
- char * entries
  table of entries
- char * entries
  table of entries

6.3.2 Field Documentation

6.3.2.1 int ByteVectorStruct::capacity

capacity of table

Definition at line 53 of file bytevector.c.

6.3.2.2 int ByteVectorStruct::count

number of entries

Definition at line 52 of file bytevector.c.

6.3.2.3 char* ByteVectorStruct::entries

table of entries

Definition at line 54 of file bytevector.c.

6.3.2.4 char* ByteVectorStruct::entries

table of entries

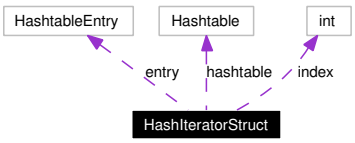
Definition at line 54 of file bitvector-old.c.

The documentation for this struct was generated from the following files:

- bitvector-old.c
- bytevector.c

6.4 HashIteratorStruct Struct Reference

Collaboration diagram for HashIteratorStruct:



6.4.1 Detailed Description

internal representation of the hash iterator.

Definition at line 72 of file hashtable.c.

Data Fields

- Hashtable hashtable
  Where do we belong to?
- int index
  index of entry
- HashtableEntry entry
  determines _next_ HashtableEntry

6.4.2 Field Documentation

6.4.2.1 HashtableEntry HashIteratorStruct::entry

determines _next_ HashtableEntry

Definition at line 75 of file hashtable.c.

6.4.2.2 Hashtable HashIteratorStruct::hashtable

Where do we belong to?

Definition at line 73 of file hashtable.c.

6.4.2.3 int HashIteratorStruct::index

index of entry

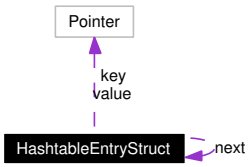
Definition at line 74 of file hashtable.c.

The documentation for this struct was generated from the following file:

- hashtable.c

6.5 HashtableEntryStruct Struct Reference

Collaboration diagram for HashtableEntryStruct:



6.5.1 Detailed Description

internal representation of the hash table entry.

Definition at line 50 of file hashtable.c.

Data Fields

- Pointer key
  hash key
- Pointer value
  value
- HashtableEntryStruct * next
  pointer to next entry

6.5.2 Field Documentation

6.5.2.1 Pointer HashtableEntryStruct::key

hash key

Definition at line 51 of file hashtable.c.

6.5.2.2 struct HashtableEntryStruct* HashtableEntryStruct::next

pointer to next entry

Definition at line 53 of file hashtable.c.

6.5.2.3 Pointer HashtableEntryStruct::value

value

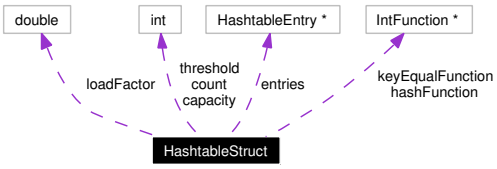
Definition at line 52 of file hashtable.c.

The documentation for this struct was generated from the following file:

- hashtable.c

6.6 HashtableStruct Struct Reference

Collaboration diagram for HashtableStruct:



6.6.1 Detailed Description

internal representation of the hash table.

Definition at line 59 of file hashtable.c.

Data Fields

- int count

  number of entries in hashtable

- int threshold

  limit when table is rehashed

- int capacity

  capacity of table

- double loadFactor

  ratio when to rehash

- IntFunction * keyEqualFunction

  equality function for key

- IntFunction * hashFunction

  hash function

- HashtableEntry * entries

  table of entries

6.6.2 Field Documentation

6.6.2.1 int HashtableStruct::capacity

capacity of table

Definition at line 62 of file hashtable.c.

6.6.2.2 int HashtableStruct::count

number of entries in hashtable

Definition at line 60 of file hashtable.c.

6.6.2.3 HashtableEntry* HashtableStruct::entries

table of entries

Definition at line 66 of file hashtable.c.

6.6.2.4 IntFunction* HashtableStruct::hashFunction

hash function

Definition at line 65 of file hashtable.c.

6.6.2.5 IntFunction* HashtableStruct::keyEqualFunction

equality function for key

Definition at line 64 of file hashtable.c.

6.6.2.6 double HashtableStruct::loadFactor

ratio when to rehash

Definition at line 63 of file hashtable.c.

6.6.2.7 int HashtableStruct::threshold

limit when table is rehashed

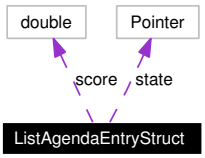
Definition at line 61 of file hashtable.c.

The documentation for this struct was generated from the following file:

- hashtable.c

6.7 ListAgendaEntryStruct Struct Reference

Collaboration diagram for ListAgendaEntryStruct:



6.7.1 Detailed Description

this type represents an entry of an agenda.

Only elements of the same type should be inserted into the same agenda, since there can be only one function (freestate())for deallocating the elements.

Definition at line 55 of file listagenda.c.

Data Fields

- double score

  contains the rating of an agenda element

- Pointer state

  holds a pointer to the actual element

6.7.2 Field Documentation

6.7.2.1 double ListAgendaEntryStruct::score

contains the rating of an agenda element

Large scores are sorted to appear before small scores.Definition at line 56 of file listagenda.c.

6.7.2.2 Pointer ListAgendaEntryStruct::state

holds a pointer to the actual element

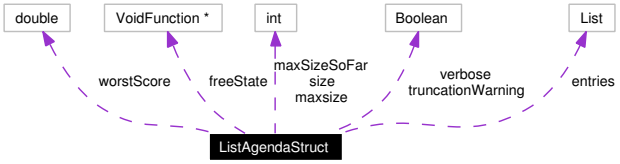
Since its type is unknown, the agenda never uses this value except to return it.Definition at line 60 of file listagenda.c.

The documentation for this struct was generated from the following file:

- listagenda.c

6.8 ListAgendaStruct Struct Reference

Collaboration diagram for ListAgendaStruct:



6.8.1 Detailed Description

quick, should be binary tree.

Definition at line 69 of file listagenda.c.

Data Fields

- int maxsize

  Gives the maximal allowed size of the agenda.

- Boolean verbose

  If set to true then truncation warnings take effect.

- Boolean truncationWarning

  This is set after an overflow messsage is printed.

- int size

  gives the current size of the agenda

- int maxSizeSoFar

  gives the maximal value that the size has reached so far

- double worstScore

  holds the score of the last item of the agenda

- VoidFunction * freeState

  used on an element when when it is lost through overflow or when the entire agenda is freed

- List entries

  holds an item in an agenda

6.8.2 Field Documentation

6.8.2.1 List ListAgendaStruct::entries

holds an item in an agenda

The agenda will allocate and deallocate its own items.The elements,however, will only be deallocated in special cases.Definition at line 90 of file listagenda.c.

6.8.2.2 VoidFunction* ListAgendaStruct::freeState

used on an element when when it is lost through overflow or when the entire agenda is freed

Definition at line 87 of file listagenda.c.

6.8.2.3 int ListAgendaStruct::maxsize

Gives the maximal allowed size of the agenda.

If more elements are inserted to it,the agenda will overflow,and the elements with the worst scores will be lost.Definition at line 70 of file listagenda.c.

6.8.2.4 int ListAgendaStruct::maxSizeSoFar

gives the maximal value that the size has reached so far

Definition at line 80 of file listagenda.c.

6.8.2.5 int ListAgendaStruct::size

gives the current size of the agenda

Definition at line 79 of file listagenda.c.

6.8.2.6 Boolean ListAgendaStruct::truncationWarning

This is set after an overflow messsage is printed.

It inhibits any further warning messages.Definition at line 76 of file listagenda.c.

6.8.2.7 Boolean ListAgendaStruct::verbose

If set to true then truncation warnings take effect.

Definition at line 75 of file listagenda.c.

6.8.2.8 double ListAgendaStruct::worstScore

holds the score of the last item of the agenda

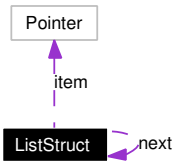
This field helps us to quickly determine whether an insert operation will make the agenda overflow.Definition at line 83 of file listagenda.c.

The documentation for this struct was generated from the following file:

- listagenda.c

6.9 ListStruct Struct Reference

Collaboration diagram for ListStruct:



6.9.1 Detailed Description

a list node.

Definition at line 42 of file list.c.

Data Fields

- Pointer item
  void element link to member of list
- ListStruct * next
  link to the next element in the list

6.9.2 Field Documentation

6.9.2.1 Pointer ListStruct::item

void element link to member of list

Definition at line 43 of file list.c.

6.9.2.2 struct ListStruct* ListStruct::next

link to the next element in the list

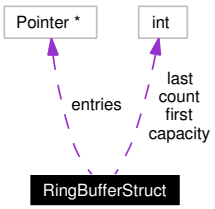
Definition at line 44 of file list.c.

The documentation for this struct was generated from the following file:

- list.c

6.10 RingBufferStruct Struct Reference

Collaboration diagram for RingBufferStruct:



6.10.1 Detailed Description

internal representation of the ring buffer.

Definition at line 43 of file ringbuffer.c.

Data Fields

- int first
  index of first item
- int last
  index of the item after the last item
- int count
  number of entries
- int capacity
  capacity of table
- Pointer * entries
  table of entries

6.10.2 Field Documentation

6.10.2.1 int RingBufferStruct::capacity

capacity of table

Definition at line 47 of file ringbuffer.c.

6.10.2.2 int RingBufferStruct::count

number of entries

Definition at line 46 of file ringbuffer.c.

6.10.2.3 Pointer* RingBufferStruct::entries

table of entries

Definition at line 48 of file ringbuffer.c.

6.10.2.4 int RingBufferStruct::first

index of first item

Definition at line 44 of file ringbuffer.c.

6.10.2.5 int RingBufferStruct::last

index of the item after the last item

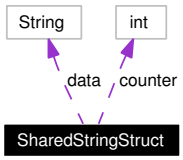
Definition at line 45 of file ringbuffer.c.

The documentation for this struct was generated from the following file:

- ringbuffer.c

6.11 SharedStringStruct Struct Reference

Collaboration diagram for SharedStringStruct:



6.11.1 Detailed Description

strings with reference counters

Definition at line 66 of file string.c.

Data Fields

- String data
  the workload
- int counter
  a reference counter

6.11.2 Field Documentation

6.11.2.1 int SharedStringStruct::counter

a reference counter

Definition at line 68 of file string.c.

Referenced by _strNewSharedString(), strDelete(), and strRegister().

6.11.2.2 String SharedStringStruct::data

the workload

Definition at line 67 of file string.c.

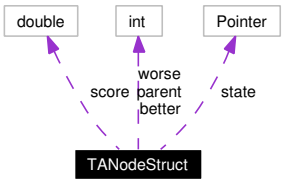
Referenced by _strDeleteSharedString(), _strNewSharedString(), and strRegister().

The documentation for this struct was generated from the following file:

- string.c

6.12 TANodeStruct Struct Reference

Collaboration diagram for TANodeStruct:



6.12.1 Detailed Description

this type represents an entry of an agenda.

Definition at line 59 of file treeagenda.c.

Data Fields

- int parent

  index of parent node or -1 if root

- int better

  index of left child or -1 if non-existent

- int worse

  index of right child or -1 if non-existent

- double score

  score for this state

- Pointer state

  pointer to state

6.12.2 Field Documentation

6.12.2.1 int TANodeStruct::better

index of left child or -1 if non-existent

Definition at line 61 of file treeagenda.c.

6.12.2.2 int TANodeStruct::parent

index of parent node or -1 if root

Definition at line 60 of file treeagenda.c.

6.12.2.3 double TANodeStruct::score

score for this state

Definition at line 63 of file treeagenda.c.

6.12.2.4 Pointer TANodeStruct::state

pointer to state

Definition at line 64 of file treeagenda.c.

6.12.2.5 int TANodeStruct::worse

index of right child or -1 if non-existent

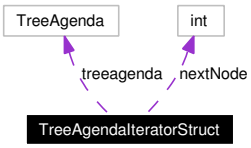
Definition at line 62 of file treeagenda.c.

The documentation for this struct was generated from the following file:

- treeagenda.c

6.13 TreeAgendaIteratorStruct Struct Reference

Collaboration diagram for TreeAgendaIteratorStruct:



6.13.1 Detailed Description

this structure instantiates the generic agenda iterator.

Definition at line 88 of file treeagenda.c.

Data Fields

- TreeAgenda treeagenda

  points to the underlying TreeAgenda.

- int nextNode

  an index to the next node to be returned.

6.13.2 Field Documentation

6.13.2.1 int TreeAgendaIteratorStruct::nextNode

an index to the next node to be returned.

Definition at line 90 of file treeagenda.c.

6.13.2.2 TreeAgenda TreeAgendaIteratorStruct::treeagenda

points to the underlying TreeAgenda.

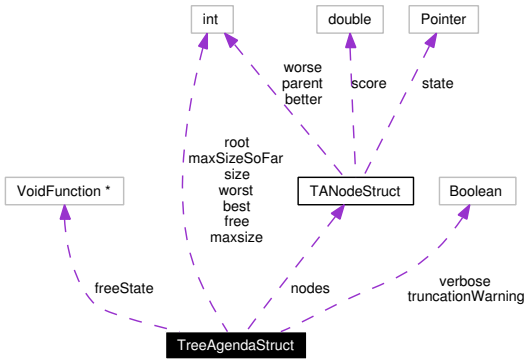
Definition at line 89 of file treeagenda.c.

The documentation for this struct was generated from the following file:

- treeagenda.c

6.14 TreeAgendaStruct Struct Reference

Collaboration diagram for TreeAgendaStruct:



6.14.1 Detailed Description

quick, should be binary tree.

Definition at line 70 of file treeagenda.c.

Data Fields

- int size
  total number of used nodes
- int maxsize
  number of allocated nodes
- int maxSizeSoFar
  maximum number of used nodes so far
- Boolean verbose
  If set to true then truncation warnings take effect.
- Boolean truncationWarning
  flag whether warning has been emitted
- int free
  index of first free node or -1 if non-existent
- int root
  index of root node or -1 if non-existent
- int best
  index of best node or -1 if non-existent
- int worst

index of worst node or -1 if non-existent

- VoidFunction * freeState
  free function
- TANodeStruct * nodes
  array of maxsize nodes

6.14.2 Field Documentation

6.14.2.1 int TreeAgendaStruct::best

index of best node or -1 if non-existent

Definition at line 78 of file treeagenda.c.

6.14.2.2 int TreeAgendaStruct::free

index of first free node or -1 if non-existent

Definition at line 76 of file treeagenda.c.

6.14.2.3 VoidFunction* TreeAgendaStruct::freeState

free function

Definition at line 80 of file treeagenda.c.

6.14.2.4 int TreeAgendaStruct::maxsize

number of allocated nodes

Definition at line 72 of file treeagenda.c.

6.14.2.5 int TreeAgendaStruct::maxSizeSoFar

maximum number of used nodes so far

Definition at line 73 of file treeagenda.c.

6.14.2.6 TANodeStruct* TreeAgendaStruct::nodes

array of maxsize nodes

Definition at line 81 of file treeagenda.c.

6.14.2.7 int TreeAgendaStruct::root

index of root node or -1 if non-existent

Definition at line 77 of file treeagenda.c.

6.14.2.8 int TreeAgendaStruct::size

total number of used nodes

Definition at line 71 of file treeagenda.c.

6.14.2.9 Boolean TreeAgendaStruct::truncationWarning

flag whether warning has been emitted

Definition at line 75 of file treeagenda.c.

6.14.2.10 Boolean TreeAgendaStruct::verbose

If set to true then truncation warnings take effect.

Definition at line 74 of file treeagenda.c.

6.14.2.11 int TreeAgendaStruct::worst

index of worst node or -1 if non-existent

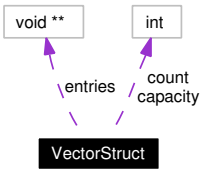
Definition at line 79 of file treeagenda.c.

The documentation for this struct was generated from the following file:

- treeagenda.c

6.15 VectorStruct Struct Reference

Collaboration diagram for VectorStruct:



6.15.1 Detailed Description

internal representation of a vector

Definition at line 49 of file vector.c.

Data Fields

- int count
  number of entries
- int capacity
  capacity of table
- void ** entries
  table of entries

6.15.2 Field Documentation

6.15.2.1 int VectorStruct::capacity

capacity of table

Definition at line 54 of file vector.c.

6.15.2.2 int VectorStruct::count

number of entries

Definition at line 53 of file vector.c.

6.15.2.3 void** VectorStruct::entries

table of entries

Definition at line 55 of file vector.c.

The documentation for this struct was generated from the following file:

- vector.c

Chapter 7

BLAH Page Documentation

7.1 Todo List

Group TreeAgenda Actually the b-tree used here can be implemented in a more general way to be more usefull. An agenda is just one way to use a b-treeish storage organization.

Index