Package: R6DS (via r-universe)

September 7, 2024

Type Package

Title R6 Reference Class Based Data Structures

Version 1.2.0

Description Provides reference classes implementing some useful data structures. The package implements these data structures by using the reference class R6. Therefore, the classes of the data structures are also reference classes which means that their instances are passed by reference. The implemented data structures include stack, queue, double-ended queue, doubly linked list, set, dictionary and binary search tree. See for example <https://en.wikipedia.org/wiki/Data_structure> for more information about the data structures.

Depends R (>= 3.0.0)

License GPL-3

Encoding UTF-8

LazyData true

RoxygenNote 6.1.1

URL https://github.com/yukai-yang/R6DS

BugReports https://github.com/yukai-yang/R6DS/issues

Imports R6

Suggests knitr, rmarkdown

VignetteBuilder knitr

Repository https://yukai-yang.r-universe.dev

RemoteUrl https://github.com/yukai-yang/r6ds

RemoteRef HEAD

RemoteSha 760a9587d83b3790ebe861991e7a1be7f6035811

Contents

`R6DS` *R6DS: provides reference classes implementing some useful data stuctures.*

Description

R6DS stands for **R6** class based Data Structures. The package provides reference classes implementing some useful data stuctures.

Details

Some data structures are quite useful in solving some programming problems, as they offer great convenience and are the keys to implement some algorithms.

The package implements these data structures by using the reference class **R6**. Each class defined in the package represents a certain data structure, and it inherits the R6 reference class which means that it is also a reference class.

In order to create an instance of the R6 type reference class, you will have to use its `new` method as follows:

```
instance <- RStack$new()
```

where `RStack` is an R6 type reference class.

The reference class has the feature that each time when you pass (or assign) an instance of the reference class to somewhere, it is not pass-by-value but pass-by-reference. For example, suppose there is an instance of the reference class `x` who has an attribute `x$y`. If you pass it to a function `func` by using `func(x)`, the function will not copy `x` but pass it as a reference. Inside the function `func`, if there is a sentence like

```
x$y <- 0
```

then the attribute `y` of the global `x` outside the function will be changed to zero.

Of course you can copy the instance of a reference class, but you have to use its `clone` method:

```
new_instance <- instance$clone()
```

Notice that all the classes in the package use instances of some other R6 type reference class as their members. This implies that, according to the rule of the R6 family, you have to add `deep = TRUE` when you `clone` their instances:

```
new_instance <- instance$clone(deep=TRUE)
```

and then you can successfully copy them.

The classes in the package are designed in the way that you cannot get the access to their members directly, as they are declared to be private. Instead, you have to use their methods (member functions) to get them. In the following, a complete list of these classes and their methods in common are presented. Each class has its own methods, and for details of these class-specific methods please refer to their help documents.

Some methods are declared to be "active", or active method, which means that, when you call them, you do not need to use parenthesis. For example, the `size` method is a common active method of all classes in the package. When you call it, you do

```
instance$size
```

So it looks pretty like a member attribute, but actually not.

How to Use the Package

All classes in the package are declared to be non-portable and non-class (R6 standards), which means that the user of the package cannot inherit them.

The user can create and use instances of these classes, and the instances can contain any R objects (vector, matrix, factor, data.frame, list and etc.) as their values.

The author suggest that the user of the package puts the instances of these classes inside other classes to be their members.

But it is still possible to inherit the classes in the package. To this end, the user can copy and paste the source code of the package.

Author and Maintainer

Yukai Yang

Department of Statistics, Uppsala University

<yukai.yang@statistik.uu.se>

References

For the details about the data structures, see Data Structure at Wikipedia.

Classes Available in the Package

`RStack` The RStack reference class implements the data structure stack.

`RQueue` The RQueue reference class implements the data structure queue.

`RDeque` The RDeque reference class implements the data structure double-ended queue.

`RDLL` The RDLL reference class implements the data structure doubly linked list.

`RSet` The RSet reference class implements the data structure set.

`RDict` The RDict reference class implements the data structure dictionary.

`RBST` The RBST reference class implements the data structure binary search tree.

Common Methods of the Classes in the Package

`new(..., collapse=NULL)` a method belonging to the class which create an instance of the class. The method creates a new instance of some class in the package containing the values in `...` and `collapse` as its elements.

`toList` an active immutable method of an instance which returns a list containing its elements (a copy).

Note that `RBST` has multiple versions of the `toList` methods.

`is_empty()` a method which returns a boolean showing if the instance is empty.

`size` an active immutable method of an instance to return its size (like the `length` of an R vector).

`release()` a method of an instance which does the garbage collection and releases the redundant memory occupation.

Note that `RDict` `RBST` do not have this methhod.

RBST *The RBST reference class*

Description

The RBST reference class implements the data structure binary search tree (BST).

Usage

```
RBST
```

Format

An object of class `R6ClassGenerator` of length 24.

Details

A BST is a particular type of container storing elements in nodes by following a binary tree structure. So the element is the value of the corresponding node in the tree.

The BST has one root on top, which is the first node of the tree, and each node in the BST has at most two sub-nodes (left sub-node and right sub-node) which can be the roots of their sub-trees.

The BST should be equipped with the "<" and "=" operations such that any two nodes in the tree can be compared. Note that, by the definitions of the "<" and "=" operations, the operation ">" is also defined.

The BST structure follows strictly the rules that, for a certain node in the tree, any nodes in its left sub-tree must be strictly smaller ("<") than it, any nodes in its right sub-tree must be strictly larger (">") than it, and any two nodes in the tree must not be equal (no "=").

Therefore, the BST is a special set or dictionary equipped with "<", ">" operations.

When you create a new RBST instance, you have to input two functions which defines the bodies of the two private methods `lessthan` and `equal`. The RBST instance then will use them to make comparison and decide where to put new nodes (build the BST).

Each time a new node is inserted, the BST algorithm finds its location on the tree. Then you can imagine, the BST is efficient in maintaining (inserting and deleting), searching and traversing the tree. An average O(log n) time complexity can be achieved by applying the BST algorithm.

A very important fact is that, the RBST only compares the nodes by using the function `equal`. So it will regard any two nodes identical if `equal` returns `TRUE`, even though they are different.

We see that the BST can also be regarded as a dictionary, as the key of the dictionary is actually the value input into `insert`, `delete` and `search_for`.

The traversals of the BST (in-order, pre-order, and post-order) are implemented as well. A `callback` function can be input into the `traverse` function to specify how to treat the traversed nodes. By default (if you do not input anything here) the `traverse` function prints the traversed nodes. But of course you can, for example, store them by changing the `callback` function, see the examples below.

The elements in the BST are not necessarily to be of the same type, and they can even contain functions.

References

For the details about the BST data structure, see BST at Wikipedia.

Class Method

The class method belongs to the class.

`new(lessthan, equal, ..., collapse=NULL)` The `new` method creates a new instance of the RBST class containing the values in `...` and `collapse` as its nodes.

The argument `lessthan` takes a function defining the "<" operation, and the argument `equal` takes a function defining the "=" operation. Both of the functions takes two values of the nodes in the tree and return a boolean.

`lessthan` can take, for example, the form

```
lessthan <- function(x, y) return(x$num < y$num)
```

where `x` and `y` are values of two nodes in the tree with the attribute `num`.

`equal` can take, for example, the form

```
equal <- function(x, y) return(x$num == y$num)
```

where `x` and `y` are values of two nodes in the tree with the attribute `num`.

Immutable Methods

The immutable methods do not change the nodes of the instance.

`toList`**,** `toList_pre`**, and** `toList_post` The active method `toList` returns a list containing its elements (a copy).

The order of the list can be "traverse-in-order" by using `toList`, "traverse-pre-order" by using `toList_pre`, or "traverse-post-order" by using `toList_post`

`traverse(mode, callback=function(item){print(item)}, ...)` The `traverse` method takes at least two arguments which are `mode` and `callback`.

The `mode` takes a value in one of the three strings `"in"`, `"pre"`, and `"post"` which indicate *traverse-in-order*, *traverse-pre-order*, and *traverse-post-order*, respectively.

The `callback` takes a function specifying how to handle the value of each node in the tree. By default, `callback` prints the nodes by using the `print` function.

Note that the first argument of the `callback` function must be the value of the node but not the node itself!

`callback` can have two or more arguments. The method also takes `...` as the additional arguments for the `callback` function if any.

`search_for(val)` The method `search_for` uses the `equal` function to compare `val` with the nodes in BST. It returns the value of the node if the node is `equal` to the given value, and `NULL` otherwise.

As the tree has been structured strictly by following the rules introduced above, there is no need to search the whole tree in most cases, and the maintaining and searching are efficient.

`min` The active method `min` returns the smallest node in the tree, and `NULL` if the tree is empty.

`max` The active method `min` returns the largest node in the tree, and `NULL` if the tree is empty.

Mutable Methods

The mutable methods changes the nodes of the instance.

`insert(..., collapse=NULL)` The method `insert` inserts new nodes into the tree. If some nodes are `equal` to the nodes in the tree, they will not be inserted.

`delete(val)` The method `delete` removes the node which is `equal` to `val`. If the node is found, then it will be removed and the function returns a `TRUE`, and if the node is not found, then it will do nothing and returns a `FALSE`,

Author(s)

Yukai Yang, <yukai.yang@statistik.uu.se>

See Also

R6DS for the introduction of the reference class and some common methods

Examples

```
### create a new instance

# you have to define two functions for "<" and "="
lessthan <- function(x, y) return(x$key < y$key)
equal <- function(x, y) return(x$key == y$key)
# remember that the nodes in the BST have the "key" variable
# and it is numeric

# to create a new instance of the class
bst <- RBST$new(lessthan=lessthan, equal=equal)
```

```
# of course you can start to push elements when creating the instance
bst <- RBST$new(lessthan=lessthan, equal=equal,
    list(key=5, val="5"), collapse=list(list(key=3,val="3"), list(key=9,val="9")))
# the following sentence is equivalent to the above
bst <- RBST$new(lessthan=lessthan, equal=equal,
    list(key=5, val="5"), list(key=3,val="3"), list(key=9,val="9"))
# where the three lists are inserted into the BST

### maintaining

bst$insert(list(key=5, val="6"))
bst$insert(list(key=6, val="5"))

bst$delete(list(key=7, val="7"))
# FALSE
bst$delete(list(key=6, val="7"))
# TRUE and delete list(key=6, val="5")
# though val are different

### searching

bst$search_for(list(key=0, val="0"))
# NULL
bst$search_for(list(key=5, val="0"))
# the BST has a node whose key is 5

### min and max

# min and max are two active functions
# so the parenthesis is not needed
bst$min
bst$max

### toList

bst$toList
bst$toList_pre
bst$toList_post

### traversing

# by default, the callback function prints the nodes
# but you can re-define the callback function
queue <- RQueue$new()
callback <- function(item)queue$enqueue(item)
# remember that RQueue is a reference class
# so the new callback will store the traversed nodes

bst$traverse(mode = "in", callback=callback)
tmp = queue$dequeue(); print(tmp)
while(!is.null(tmp)) {tmp = queue$dequeue(); print(tmp)}
bst$traverse(mode = "in", callback=callback)
```

```
tmp = queue$dequeue(); print(tmp)
while(!is.null(tmp)) {tmp = queue$dequeue(); print(tmp)}

# pre-order traversing
bst$traverse(mode = "pre", callback=callback)
tmp = queue$dequeue(); print(tmp)
while(!is.null(tmp)) {tmp = queue$dequeue(); print(tmp)}

# post-order traversing
bst$traverse(mode = "post", callback=callback)
tmp = queue$dequeue(); print(tmp)
while(!is.null(tmp)) {tmp = queue$dequeue(); print(tmp)}
```

`RDeque` *The RDeque reference class*

Description

The RDeque reference class implements the data structure double-ended queue (deque).

Usage

```
RDeque
```

Format

An object of class `R6ClassGenerator` of length 24.

Details

A deque is an ordered list of elements generalizing the queue data structure. One can append and pop (return and remove) elements from both sides (left and right, front and rear) of the deque.

The elements in the deque are not necessarily to be of the same type, and they can be any R objects.

References

For the details about the deque data structure, see Deque at Wikipedia.

Immutable Methods

The immutable methods do not change the instance.

`peekleft()` This method returns the leftmost (front) element of the deque. It returns `NULL` if the deque is empty.

`peek()` This method returns the rightmost (rear) element of the deque. It returns `NULL` if the deque is empty.

Mutable Methods

The mutable methods change the instance.

`appendleft(..., collapse=NULL)` The `appendleft` method appends the elements in `...` and `collapse` into the deque to the left (front).

Note that if you append elements in this order:

`instance$appendleft(elem1, elem2, elem3)`

The order of them inside the deque will be

`elem3, elem2, elem1, ...`

and `elem3` will be the new front of the deque.

`append(..., collapse=NULL)` The `append` method appends the elements in `...` and `collapse` into the deque to the right (rear).

`popleft()` The `popleft` method returns and removes the leftmost (front) element in the deque. It returns `NULL` if the deque is empty.

`pop()` The `pop` method returns and removes the rightmost (rear) element in the deque. It returns `NULL` if the deque is empty.

Author(s)

Yukai Yang, <yukai.yang@statistik.uu.se>

See Also

RStack, RQueue, and R6DS for the introduction of the reference class and some common methods

Examples

```
### create a new instance

# to create a new instance of the class
deque <- RDeque$new()

# the previous RDeque instance will be removed if you run
deque <- RDeque$new(0, 1, 2, collapse=list(3, 4))
# the following sentence is equivalent to the above
deque <- RDeque$new(0, 1, 2, 3, 4)
# where the numbers 0, 1, 2, 3, 4 are enqueued into the deque

### append and appendleft

# it can be one single element
deque$append(5)
# it can be several elements separated by commas
# note the whole list will be one element of the deque
# because it is not passed through the collapse argument
deque$append(list(a=10,b=20), "Hello world!")
# the collapse argument takes a list whose elements will be collapsed
# but the elements' names will not be saved
deque$append(collapse = list(x=100,y=200))
```

```
# they can be used together
deque$append("hurrah", collapse = list("RDeque",300))

# this string will be the new head
deque$appendleft("a string")
# we can update the head by
deque$appendleft("string3","string2","string1")
# "string1" will be the leftmost

### peekleft and peek
deque$peekleft()
# "string1"
deque$peek()
# 300

### popleft and pop

val <- deque$popleft()
# "string1"
val <- deque$pop()
# 300

# then we keep dequeuing!
while(!is.null(val)) val <- deque$pop()
```

`RDict` *The RDict reference class*

Description

The RDict reference class implements the data structure dictionary.

Usage

```
RDict
```

Format

An object of class `R6ClassGenerator` of length 24.

Details

A dictionary is a collection of (key, value) pairs as its elements such that each possible key appears at most once in the collection. The dictionary data structure does not care the order of the elements.

The keys of the elements in the dictionary are stored as strings. The values in the dictionary are not necessarily to be of the same type, and they can be any R objects.

References

For the details about the dictionary data structure, see Dictionary at Wikipedia.

Immutable Methods

The immutable methods do not change the instance.

`has(key)` The method `has` returns a boolean indicating if the dictionary contains the element with the key `"key"`.

Both of the following two sentences are equivalent:

```
instance$has("keyname")
instance$has(keyname)
```

`get(key)` The method `get` returns the value of the element whose key is `"key"`. It returns `NULL` if no element is found.

`keys` The method `keys` returns a vector of the keys in the dictionary.

`values` The method `values` returns a list of the values in the dictionary (unnamed list).

Mutable Methods

The mutable methods change the instance.

`add(key, val)` The method `add` adds a new element (the pair key and val) into the dictionary. It will not add element with the key which exists already in the dictionary. It returns a boolean showing if the adding is successful.

Note that any element with the key `""` (empty string) will not be added.

`add_multiple(..., collapse=NULL)` The method `add_multiple` adds new elements into the dictionary. It will not add element with the key which exists already in the dictionary.

The argument `...` stands for any input with the form

`keyname1 = value2, keyname2 = value2, ...`

Therefor, the input can take the form

```
instance$add(key1=1, key2="hello", key3=list(1))
```

and the keys of the elements will be strings like `"key1"`, `"key2"`, and `"key3"`, respectively.

If the `keyname` is missing, the `value` will not be added.

`delete(key)` The method `delete` removes the element with the key `key` in the dictionary.

Suppose that the key name of the element that you want to remove is "keyname". Both of the following two sentences are valid:

```
instance$delete("keyname")
instance$delete(keyname)
```

It returns a boolean showing if the element is found and deleted.

Author(s)

Yukai Yang, <yukai.yang@statistik.uu.se>

See Also

R6DS for the introduction of the reference class and some common methods

Examples

```
### create a new instance

# to create a new instance of the class
dict <- RDict$new()

# of course you can start to add elements when creating the instance
dict <- RDict$new(id0001=1, id0002=2, collapse=list(id0003=3, id0004=4))
# the following sentence is equivalent to the above
dict <- RDict$new(id0001=1, id0002=2, id0003=3, id0004=4)
# where the three lists are inserted into the dictionary

### immutable methods

dict$keys
dict$values

dict$has(id0001)
dict$has("id0005")
# TRUE as it has the key attribute

dict$get(id0006)
dict$get("id0002")

### mutable methods

dict$add(id0005, 5)

dict$add(key="id0006", val=6)

dict$delete(id0001)
```

RDLL *The RDLL reference class*

Description

The RDLL reference class implements the data structure doubly linked list (DLL).

Usage

```
RDLL
```

Format

An object of class `R6ClassGenerator` of length 24.

Details

A doubly linked list is an ordered list of elements with multiple operations. The DLL is a powerful sequantial data structure in the sense that it can be regarded as the generalized version of the data structures stack, queue, deque.

The class `RDLL` inherits the `RDeque` class, and therefor it has all the methods that `RDeque` has.

The DLL is much more friendly and flexible as it offers more useful methods to help the user get access to its elements than `RStack`, `RQueue` and `RDeque`. See below its immutable methods and mutable methods.

It is worth noting that the classes `RSet` inherits the RDLL class, and therefor it has all the methods that the RDLL has.

The elements in the DLL are not necessarily to be of the same type, and they can be any R objects.

References

For the details about the DLL data structure, see DLL at Wikipedia.

Immutable Methods

The immutable methods do not change the instance.

`show(callback=function(val){print(val)}, ...)` The `show` method takes a funtion input (argument `callback`) specifying how to handle the elements in the DLL. It also takes `...` as the additional arguments for the `callback` function if any.

By default, the `show` method prints the elements by using the `print` function.

`callback=function(val){print(val)}`

You can see that `show` is powerful as it makes it possible to freely manipulate the elements in the DLL. For example, you can define

`func <- function(val, arg1, arg2){ do something here on val with arg1 and arg2 }`

and then

`instance$show(func, arg1, arg2)`

And you can also store the elements by using instances of reference classes. For example,

`func <- function(val, queue){ queue$enqueue(val) }`

where `queue` is an instance of `RQueue`. The code can be

`queue <- RQueue$new()`

`instance$show(func, queue)`

`elem_at(index)` It returns the element (a copy) at position `index` (a positive integer). `index` must be a scalar, and if it is a vector of more than one element, only the first element will be considered. If the value of `index` is out of the bounds of the instance, a `NULL` will be returned.

`peekleft()` See `RDeque`.

`peek()` See `RDeque`.

Mutable Methods

The mutable methods change the instance.

`insert_at(index, val)` This function inserts a new element `val` at position `index`. It returns `TRUE` if the insertion is successful, and `FALSE` if the `index` is out of the bounds. It will push all the elements at and after `index` rightward.

Thus, suppose that `instance` is an instance of the class.

`insert_at(1, val)`

is equivalent to `appendleft` in `RDeque`, and

`insert_at(instance$size+1, val)`

is equivalent to `append` in `RDeque`, `push` in `RStack`, and `enqueue` in `RQueue`.

`remove_at(index)` This function returns and removes the element at position `index`. It returns `NULL` if the `index` is out of the bounds.

Thus, suppose that `instance` is an instance of the class.

`remove_at(1, val)` is equivalent to `popleft` in `RDeque`, and

`remove_at(instance$size, val)` is equivalent to `pop` in `RDeque` and `RStack`, and `dequeue` in `RQueue`.

`appendleft(..., collapse=NULL)` See `RDeque`.

`append(..., collapse=NULL)` See `RDeque`.

`popleft()` See `RDeque`.

`pop()` See `RDeque`.

Author(s)

Yukai Yang, <yukai.yang@statistik.uu.se>

See Also

RDeque, RSet, and R6DS for the introduction of the reference class and some common methods

Examples

```
### create a new instance

# to create a new instance of the class
dll <- RDLL$new()

# the previous RDLL instance will be removed if you run
dll <- RDLL$new(0, 1, 2, collapse=list(3, 4))
# the following sentence is equivalent to the above
dll <- RDLL$new(0, 1, 2, 3, 4)
# where the numbers 0, 1, 2, 3, 4 are appended into the DLL

### immutable methods

# show
dll$show()
```

```
# elem_at
dll$elem_at(1)

# toList
tmp <- dll$toList

### mutable methods

# insert_at
dll$insert_at(1, -1)
dll$insert_at(dll$size+1, "end")

# remove_at
for(iter in 1:dll$size) dll$remove_at(1)
```

RLinkedList *The linked list reference class*

Description

The linked list reference class gives

Usage

```
RLinkedList
```

Format

An object of class `R6ClassGenerator` of length 24.

Author(s)

Yukai Yang, <yukai.yang@statistik.uu.se>

RNode *The node reference class*

Description

The node reference class gives

Usage

```
RNode
```

Format

An object of class `R6ClassGenerator` of length 24.

Author(s)

Yukai Yang, <yukai.yang@statistik.uu.se>

`RQueue` *The RQueue reference class*

Description

The RQueue reference class implements the data structure queue.

Usage

```
RQueue
```

Format

An object of class `R6ClassGenerator` of length 24.

Details

A queue is an ordered list of elements following the First-In-First-Out (FIFO) principle. The `enqueue` method takes elements and add them to the rear terminal position (right) of the queue, while the `dequeue` method returns and removes the element in the queue from the front terminal position (left).

The elements in the queue are not necessarily to be of the same type, and they can be any R objects.

References

For the details about the queue data structure, see Queue at Wikipedia.

Immutable Methods

The immutable method does not change the instance.

`peekleft()` This method returns the leftmost (front) element in the queue. It returns `NULL` if the queue is empty.

Mutable Methods

The mutable methods change the instance.

`enqueue(..., collapse=NULL)` The `enqueue` method enqueues the elements in `...` and `collapse` into the queue (to the right or rear).

Note that you can input multiple elements.

`dequeue()` The `dequeue` method dequeues (returns and removes) one element (the leftmost or front) fron the queue. It returns `NULL` if the queue is empty.

Author(s)

Yukai Yang, <yukai.yang@statistik.uu.se>

See Also

R6DS for the introduction of the reference class and some common methods

Examples

```
### create a new instance

# to create a new instance of the class
queue <- RQueue$new()

# the previous RQueue instance will be removed if you run
queue <- RQueue$new(0, 1, 2, collapse=list(3, 4))
# the following sentence is equivalent to the above
queue <- RQueue$new(0, 1, 2, 3, 4)
# where the numbers 0, 1, 2, 3, 4 are enqueued into the queue

### enqueue elements

# it can be one single element
queue$enqueue(5)
# it can be several elements separated by commas
# note the whole list will be one element of the queue
# because it is not passed through the collapse argument
queue$enqueue(list(a=10,b=20), "Hello world!")
# the collapse argument takes a list whose elements will be collapsed
# but the elements' names will not be saved
queue$enqueue(collapse = list(x=100,y=200))
# they can be used together
queue$enqueue("hurrah", collapse = list("RQueue",300))

### dequeue an element

# dequeue only one element at a time
val <- queue$dequeue()
# then we keep dequeuing!
while(!is.null(val)) val <- queue$dequeue()
```

`RSet` *The RSet reference class*

Description

The RSet reference class implements the data structure set.

Usage

```
RSet
```

Format

An object of class `R6ClassGenerator` of length 24.

Details

A set is a collection of items or elements equipped with the "=" operators such that any two elements in the set cannot be equal. The set data structure does not care the order of the elements.

It should be noticed that, in your design, if any two elements in the set can be easily compared, by simply, for example, keys, numbers, and etc., the RSet should not be recommended due to efficiency reason. The RSet is suitable for the cases when you have a relatively complex "=" operation between two elements in the set.

The class `RSet` inherits the `RDLL` class, and therefor it has all the methods that `RDLL` has.

Note that the methods `insert_at`, `appendleft`, `append` in the super class still works without checking if the new element equals any other elements in the set. Normally they should be depreciated in the `RSet` class, but this is not done in the current version of the package. It is strongly recommended that the user should use the `add` method to add a new element when using the `RSet` class.

The elements in the set are not necessarily to be of the same type, and they can be any R objects.

References

For the details about the set data structure, see Set at Wikipedia.

Class Method

The class method belongs to the class.

`new(equal, ..., collapse=NULL)` The `new` method creates a new instance of the RSet class containing the values in `...` and `collapse` as its elements.

The argument `equal` takes a function defining the "=" operation, The function set to `equal` takes two values of the elements in the set and return a boolean. It can be, for example, of the form

```
equal <- function(x, y) return(x$num == y$num)
```

where `x` and `y` are values of two elements in the set with the attribute `num`.

Immutable Methods

The immutable methods do not change the elements of the instance.

`has(val)` The method `has` returns a boolean indicating if the set contains `val`.

`union(rset)` The method `union` merges the elements in `rset`, an instance of some class in the package, with its elements, and returns a new union set of the two.

`intersection(rset)` The method `intersection` returns a new intersection set (RSet) of the current set and `rset`, an instance of some class in the package.

`difference(rset)` The method `difference` returns a new difference set (RSet) of the current set and `rset`, an instance of some class in the package (current instance minus `rset`).

`subset(rset)` The method `subset` returns a boolean indicating if the current set is a subset of `rset`, an instance of some class in the package.

`contains(rset)` The method `contains` returns a boolean indicating if the current set contains `rset`, an instance of some class in the package.

Mutable Methods

The mutable methods change the instance.

`add(val)` The method `add` adds a new element into the set and returns a booleank showing if the insertion is successful.

`add_multiple(..., collapse=NULL)` The method `add_multiple` adds new elements in `...` and `collapse` into the set.

`delete(val)` The method `delete` removes the element which is `equal` to `val` in the set. It returns a boolean showing if the deletion is successful (if the element is not found in the set).

Author(s)

Yukai Yang, <yukai.yang@statistik.uu.se>

See Also

RDLL and R6DS for the introduction of the reference class and some common methods

Examples

```
### create a new instance

# you have to define "="
equal <- function(x, y) return(x$key == y$key)
# remember that the elements in the set must have the "key" attribute

# to create a new instance of the class
set <- RSet$new(equal=equal)

# of course you can start to add elements when creating the instance
set <- RSet$new(equal=equal,
    list(key=5, val="5"), collapse=list(list(key=3,val="3"), list(key=9,val="9")))
# the following sentence is equivalent to the above
set <- RSet$new(equal=equal,
    list(key=5, val="5"), list(key=3,val="3"), list(key=9,val="9"))
# where the three lists are inserted into the set

### immutable methods

set$has(list(key=5, num=10))
# TRUE as it has the key attribute
```

```
### mutable methods

set$add(list(key=5, num=10))
# FALSE

set$add(list(key=10, val="10"))
# TRUE

set$delete(list(key=10))
# TRUE and list(key=10, val="10") is removed

# union
another_set <- RSet$new(equal=equal,
    list(key=5, val="5"), list(key=11,val="11"))
set$union(another_set)$show()

# intersection
set$intersection(another_set)$show()

# difference
set$difference(another_set)$show()

# subset
set$subset(another_set)

# contains
set$contains(another_set)
```

`RStack` *The RStack reference class*

Description

The RStack reference class implements the data structure stack.

Usage

```
RStack
```

Format

An object of class `R6ClassGenerator` of length 24.

Details

A stack is an ordered list of elements following the Last-In-First-Out (LIFO) principle. The `push` method takes elements and add them to the top position (right) of the stack, while the `pop` method returns and removes the last "pushed" (top or rightmost) element in the stack.

The elements in the stack are not necessarily to be of the same type, and they can be any R objects.

References

For the details about the stack data structure, see Stack at Wikipedia.

Immutable Methods

The immutable method does not change the instance.

`peek()` This method returns the last pushed (top or rightmost) element in the stack. It returns `NULL` if the stack is empty.

Mutable Methods

The mutable methods change the instance.

`push(..., collapse=NULL)` The `push` method pushes the elements in `...` and `collapse` into the stack (to the top or right).

Note that you can input multiple elements.

`pop()` The `pop` method pops (returns and removes) the last pushed (rightmost) element in the stack. It returns `NULL` if the stack is empty.

Author(s)

Yukai Yang, <yukai.yang@statistik.uu.se>

See Also

R6DS for the introduction of the reference class and some common methods

Examples

```
### create a new instance

# to create a new instance of the class
stack <- RStack$new()

# the previous RStack instance will be removed if you run
stack <- RStack$new(0, 1, 2, collapse=list(3, 4))
# the following sentence is equivalent to the above
stack <- RStack$new(0, 1, 2, 3, 4)
# where the numbers 0, 1, 2, 3, 4 are pushed into the stack

### push elements

# it can be one single element
stack$push(5)
# it can be several elements separated by commas
# note the whole list will be one element of the stack
# because it is not passed through the collapse argument
stack$push(list(a=10,b=20), "Hello world!")
# the collapse argument takes a list whose elements will be collapsed
# but the elements' names will not be saved
```

```
stack$push(collapse = list(x=100,y=200))
# they can be used together
stack$push("hurrah", collapse = list("RStack",300))

### pop an element

# pop only one element at a time
val <- stack$pop()
# then we keep poping!
while(!is.null(val)) val <- stack$pop()
```

`version` *Show the version number of some information.*

Description

This function shows the version number and some information of the package.

Usage

```
version()
```

Author(s)

Yukai Yang, <yukai.yang@statistik.uu.se>

Index