FreePascal generic container library

(manual)

http://code.google.com/p/stlpascal

version 1.0

Contents

TVector

Implements selfresizing array. Indexing is 0-based.

Usage example:

```
uses gvector;

type TVectorlli = specialize TVector<longint>;

var Buffer:TVectorlli; i:longint;

begin
  Buffer := TVectorlli.Create;
  {Push 5 elements at the end of array}
  for i:=1 to 5 do
    Buffer.PushBack(i);
  {change 3rd element to 47}
  Buffer[2] := 47;
  {pop last element}
  Buffer.PopBack;
  {print all elements}
  for i:=0 to Buffer.Size-1 do
    writeln(Buffer[i]);

  Buffer.Destroy;
end.
```

Memory complexity: Uses at most 3times bigger memory than maximal array size (this is only case during reallocation). Normal consumption is at most twice as maximal array size.

Members list:

Method	Complexity guarantees
Description	
`Create`	O(1)
Constructor. Creates empty array.	
`function Size(): SizeUInt`	O(1)
Returns size of array.	
`procedure PushBack(value: T)`	Amortized O(1), some operations might take O(N) time, when array needs to be reallocated, but sequence of N operations takes O(N) time
Inserts at the end of array (increases size by 1)	

`procedure PopBack()`	O(1)
Removes element from the end of array (decreases size by 1). When array is empty, does nothing.	

`function IsEmpty(): boolean`	O(1)
Returns true when array is empty	

`procedure Insert(position: SizeUInt; value: T)`	O(N)
Inserts value at position. When position is greater than size, puts value at the end of array.	

`procedure Erase(positine: SizeUInt; value: T)`	O(N)
Erases element from position. When position is outside of array does nothing.	

`procedure Clear`	O(1)
Clears array (set size to zero). But doesn't free memory used by array.	

`function Front: T`	O(1)
Returns first element from array.	

`function Back: T`	O(1)
Returns last element from array.	

`procedure Resize(num: SizeUInt)`	O(N)
Changes size of array to num. Doesn't guarantte anything about value of newly alocated elements.	

`procedure Reserve(num: SizeUInt)`	O(N)
Alocates at least num elements for array. Usefull when you want to pushback big number of elements and want to avoid frequent reallocation.	

`property item[i: SizeUInt]: T; default;`	O(1)
Property for accessing i-th element in array. Can be used just by square brackets (its default property).	

`property mutable[i: SizeUInt]: T;`	O(1)
Returns pointer to i-th element in array. Usefull when you store records.	

TStack

Implements stack.
Usage example:

```
uses gstack;

type stacklli = specialize TStack<longint>;

var data:stacklli; i:longint;

begin
  data:=stacklli.Create;
  for i:=1 to 10 do
    data.Push(10*i);
  while not data.IsEmpty do begin
    writeln(data.Top);
    data.Pop;
  end;

  data.Destroy;
end.
```

Memory complexity: Since underlaying structure is TVector, memory complexity is same.
Members list:

Method	Complexity guarantees
Description	
`Create`	O(1)
Constructor. Creates empty stack.	
`function Size(): SizeUInt`	O(1)
Returns number of elements in stack.	
`procedure Push(value: T)`	Amortized O(1), some operations might take O(N) time, when array needs to be reallocated, but sequence of N operations takes O(N) time
Inserts element on the top of stack.	
`procedure Pop()`	O(1)
Removes element from the top of stack. If stack is empty does nothing.	
`function IsEmpty(): boolean`	O(1)
Returns true when stack is empty	

`function Top: T`	O(1)
Returns top element from stack.	

TDeque

Implements selfresizing array. Indexing is 0-based. Also implement constant time insertion from front.

Usage example:

```
uses gdeque;

type TDequelli = specialize TDeque<longint>;

var Buffer:TDequelli; i:longint;

begin
  Buffer := TDequelli.Create;
  {Push 5 elements at the end of array}
  for i:=1 to 5 do
    Buffer.PushBack(i);
  {change 3rd element to 47}
  Buffer[2] := 47;
  {pop last element}
  Buffer.PopBack;
  {push 3 element to front}
  for i:=1 to 3 do
    Buffer.PushFront(i*10);
  {print all elements}
  for i:=0 to Buffer.Size-1 do
    writeln(Buffer[i]);

  Buffer.Destroy;
end.
```

Memory complexity: Uses at most 3times bigger memory than maximal array size (this is only case during reallocation). Normal consumption is at most twice as maximal array size.

Members list:

Method	Complexity guarantees
Description	
`Create`	O(1)
Constructor. Creates empty array.	
`function Size(): SizeUInt`	O(1)
Returns size of array.	

`procedure PushBack(value: T)`	Amortized O(1), some operations might take O(N) time, when array needs to be reallocated, but sequence of N operations takes O(N) time.
Inserts at the end of array (increases size by 1)	

`procedure PopBack()`	O(1)
Removes element from the end of array (decreases size by 1). When array is empty, does nothing.	

`procedure PushFront(value: T)`	Same as PushBack.
Inserts at the beginning of array (increases size by 1)	

`procedure PopFront()`	O(1)
Removes element from the beginning of array (decreases size by 1). When array is empty, does nothing.	

`function IsEmpty(): boolean`	O(1)
Returns true when array is empty	

`procedure Insert(position: SizeUInt; value: T)`	O(N)
Inserts value at position. When position is greater than size, puts value at the end of array.	

`procedure Erase(positine: SizeUInt; value: T)`	O(N)
Erases element from position. When position is outside of array does nothing.	

`procedure Clear`	O(1)
Clears array (set size to zero). But doesn't free memory used by array.	

`function Front: T`	O(1)
Returns first element from array.	

`function Back: T`	O(1)
Returns last element from array.	

`procedure Resize(num: SizeUInt)`	O(N)
Changes size of array to num. Doesn't guarantte anything about value of newly alocated elements.	

`procedure Reserve(num: SizeUInt)`	O(N)
Alocates at least num elements for array. Usefull when you want to pushback big number of elements and want to avoid frequent reallocation.	

`property item[i: SizeUInt]: T; default;`	O(1)
Property for accessing i-th element in array. Can be used just by square brackets (its default property).	

`property mutable[i: SizeUInt]: T;`	O(1)
Returns pointer to i-th element in array. Usefull when you store records.	

TQueue

Implements queue.
Usage example:

```
uses gqueue;

type queuelli = specialize TQueue<longint>;

var data:queuelli; i:longint;

begin
  data:=queuelli.Create;
  for i:=1 to 10 do
    data.Push(10*i);
  while not data.IsEmpty do begin
    writeln(data.Front);
    data.Pop;
  end;

  data.Destroy;
end.
```

Memory complexity: Since underlaying structure is TDeque, memory complexity is same.
Members list:

Method	Complexity guarantees
Description	
`Create`	O(1)
Constructor. Creates empty queue.	
`function Size(): SizeUInt`	O(1)
Returns number of elements in queue.	
`procedure Push(value: T)`	Amortized O(1), some operations might take O(N) time, when array needs to be reallocated, but sequence of N operations takes O(N) time
Inserts element at the back of queue.	
`procedure Pop()`	O(1)
Removes element from the beginning of queue. If queue is empty does nothing.	
`function IsEmpty(): boolean`	O(1)
Returns true when queue is empty.	

`function Front: T`	O(1)
Returns the first element from queue.	

TLess, TGreater

Comparators classes. Can be used in PriorityQueue and Sorting as comparator functions. TLess is used for ordering from smallest element to largest, TGreater is used for oposite ordering.

TPriorityQueue

Implements priority queue. It's container which allow insertions of elements and then retrieval of the biggest one.

For specialization it needs two arguements. First is the type T of stored element. Second one is type comparator class, which should have class function `c(a,b: T):boolean` which return true, when a is stricly less then b (or in other words, a should be poped out after b).

Usage example:

```
{$mode objfpc}

uses gpriorityqueue;

type
  lesslli = class
    public
    class function c(a,b: longint):boolean;inline;
  end;

class function lesslli.c(a,b: longint):boolean;inline;
begin
  c:=a<b;
end;

type priorityqueuelli = specialize TPriorityQueue<longint, lesslli>;

var data:priorityqueuelli; i:longint;

begin
  data:=priorityqueuelli.Create;
  for i:=1 to 10 do
    data.Push(random(1000));
  while not data.IsEmpty do begin
    writeln(data.Top);
    data.Pop;
  end;

  data.Destroy;
end.
```

Memory complexity: Since underlaying structure is TVector, memory complexity is same.

Members list:

Method	Complexity guarantees

Description	
`Create`	O(1)
Constructor. Creates empty priority queue.	
`function Size(): SizeUInt`	O(1)
Returns number of elements in priority queue.	
`procedure Push(value: T)`	Amortized O(lg N), some operations might take O(N) time, when underlaying array needs to be reallocated, but sequence of N operations takes O(N lg N) time.
Inserts element at the back of queue.	
`procedure Pop()`	O(lg N)
Removes the biggest element from queue. If queue is empty does nothing.	
`function IsEmpty(): boolean`	O(1)
Returns true when queue is empty.	
`function Top: T`	O(1)
Returns the biggest element from queue.	

TArrayUtils

Set of utilities for manipulating arrays data.

Takes 3 arguements for specialization. First one is type of array (can be anything, which is accesible by [] operator, e. g. ordinary array, vector, ...), second one is type of array element.

Members list:

Method	Complexity guarantees
Description	
`procedure RandomShuffle(arr: TArr, size:SizeUint)`	O(N)
Shuffles elements in array in random way	

TOrderingArrayUtils

Set of utilities for manipulating arrays data.

Takes 3 arguements for specialization. First one is type of array (can be anything, which is accesible by [] operator, e. g. ordinary array, vector, ...), second one is type of array element, third one is comparator class (see TPriorityQueue for definition of comparator class).

Usage example for sorting:

```
uses garrayutils, gutil, gvector;

type vectorlli = specialize TVector<longint>;
     lesslli = specialize TLess<longint>;
     sortlli = specialize TOrderingArrayUtils<vectorlli, longint, lesslli>;

var data:vectorlli; n,i:longint;

begin
  randomize;
  data:=vectorlli.Create;
  read(n);
  for i:=1 to n do
    data.pushback(random(1000000000));
  sortlli.sort(data, data.size());
  for i:=1 to n do
    writeln(data[i-1]);

  data.Destroy;
end.
```

Members list:

Method	Complexity guarantees
Description	
`procedure Sort(arr: TArr, size:SizeUint)`	O(N log N) average and worst case. Uses QuickSort, backed up by HeapSort, when QuickSort ends up in using too much recursion.
Sort array arr, with specified size. Array indexing should be 0 based.	

TSet

Implements container for storing ordered set of unique elements. Takes 2 arguments for specialization, first one is type of elements, second one is comparator class. Usage example:

```
uses gset, gutil;

type lesslli=specialize TLess<longint>;
     setlli=specialize TSet<longint, lesslli>;

var data:setlli; i:longint; iterator:setlli.PNode;

begin
  data:=setlli.Create;

  for i:=0 to 10 do
    data.insert(i);

  {Iteration through elements}
  iterator:=data.Min;
  while iterator<>nil do begin
    writeln(iterator^.Data);
    iterator:=data.next(iterator);
  end;

  writeln(data.FindLess(7)^.Data);

  data.Destroy;
end.
```

Some methods return type of PNode. It has field Data, which can be used for retrieving data from that node. This node can be also used for navigation between elements by methods of set class. (But don't do anything else with it, you can lose data integrity.)

Memory complexity: Size of stored base + constant overhead for each stored element (3 pointers + one boolean).

Members list:

Method	Complexity guarantees
Description	
`Create`	O(1)
Constructor. Creates empty set.	
`function Size(): SizeUInt`	O(1)
Returns number of elements in set.	

`procedure Insert(value: T)`	O(lg N), N is number of elements in set
Inserts element into set.	

`procedure Delete(value: T)`	O(lg N), N is number of elements in set
Deletes value from set. If element is not in set, nothing happens.	

`function Find(value: T):PNode`	O(lg N)
Searches for value in set. If value is not there returns nil. Otherwise returns pointer to tree node (type PNode), which can be used for retrieving data from set.	

`function FindLess(value: T):PNode`	O(lg N)
Searches for greatest element less than value in set. If such element is not there returns nil. Otherwise returns pointer to tree node (type PNode), which can be used for retrieving data from set.	

`function FindLessEqual(value: T):PNode`	O(lg N)
Searches for greatest element less or equal than value in set. If such element is not there returns nil. Otherwise returns pointer to tree node (type PNode), which can be used for retrieving data from set.	

`function FindGreater(value: T):PNode`	O(lg N)
Searches for smallest element greater than value in set. If such element is not there returns nil. Otherwise returns pointer to tree node (type PNode), which can be used for retrieving data from set.	

`function FindGreaterEqual(value: T):PNode`	O(lg N)
Searches for smallest element greater or equal than value in set. If such element is not there returns nil. Otherwise returns pointer to tree node (type PNode), which can be used for retrieving data from set.	

`function Min:PNode`	O(lg N)
Returns node containing smallest element of set. If set is empty returns nil.	

`function Max:PNode`	O(lg N)
Returns node containing largest element of set. If set is empty returns nil.	

`function Next(x:PNode):PNode`	O(lg N) worst case, but traversal from smallest element to largest takes O(N) time
Returns successor of x. If x is largest element of set, returns nil.	

`function Prev(x:PNode):PNode`	O(lg N) worst case, but traversal from largest element to smallest takes O(N) time
Returns predecessor of x. If x is smallest element of set, returns nil.	

`function IsEmpty(): boolean`	O(1)
Returns true when set is empty.	

TMap

Implements container for ordered associative array with unique keys. Takes 3 arguments for specialization, first one is type of keys, second one is type of values, third one is comparator class for keys. Usage example:

```
uses gmap, gutil;

type lesslli=specialize TLess<longint>;
     maplli=specialize TMap<longint, longint, lesslli>;

var data:maplli; i:longint; iterator:maplli.TMSet.PNode;

begin
  data:=maplli.Create;

  for i:=0 to 10 do
    data[i]:=10*i;

  {Iteration through elements}
  iterator:=data.Min;
  while iterator<>nil do begin
    writeln(iterator^.Data.Key, ' ', iterator^.Data.Value);
    iterator:=data.next(iterator);
  end;

  writeln(data.FindLess(7)^.Data.Value);

  data.Destroy;
end.
```

Some methods return type TMSet.PNode. Usefull fields are Data.Key, Data.Value, for retrieving actual Key and Value from node. This node can be also used for navigation between elements by methods of set class. You can also change value in node (but not key). (But don't do anything else with it, you can lose data integrity.)

Memory complexity: Size of stored base + constant overhead for each stored element (3 pointers + one boolean).

Members list:

Method	Complexity guarantees
Description	
`Create`	O(1)
Constructor. Creates empty map.	

`function Size(): SizeUInt`	O(1)
Returns number of elements in map.	

`procedure Insert(key: TKey; value: TValue)`	O(lg N), N is number of elements in map
Inserts key value pair into map. If key was already there, it will have new value assigned.	

`procedure Delete(key: TKey)`	O(lg N)
Deletes key (and associated value) from map. If element is not in map, nothing happens.	

`function Find(key: T):TMSet.PNode`	O(lg N)
Searches for key in map. If value is not there returns nil. Otherwise returns pointer to tree node (type TMSet.PNode), which can be used for retrieving data from map.	

`function FindLess(key: T):TMSet.PNode`	O(lg N)
Searches for greatest element less than key in map. If such element is not there returns nil. Otherwise returns pointer to tree node (type TMSet.PNode), which can be used for retrieving data from map.	

`function FindLessEqual(key: T):TMSet.PNode`	O(lg N)
Searches for greatest element less or equal than key in map. If such element is not there returns nil. Otherwise returns pointer to tree node (type TMSet.PNode), which can be used for retrieving data from map.	

`function FindGreater(key: T):TMSet.PNode`	O(lg N)
Searches for smallest element greater than key in map. If such element is not there returns nil. Otherwise returns pointer to tree node (type TMSet.PNode), which can be used for retrieving data from map.	

`function FindGreaterEqual(key: T):TMSet.PNode`	O(lg N)
Searches for smallest element greater or equal than key in map. If such element is not there returns nil. Otherwise returns pointer to tree node (type TMSet.PNode), which can be used for retrieving data from map.	

`function Min:TMSet.PNode`	O(lg N)
Returns node containing smallest key of map. If map is empty returns nil.	

`function Max:TMSet.PNode`	O(lg N)
Returns node containing largest key of map. If map is empty returns nil.	

`function Next(x:TMSet.PNode):TMSet.PNode`	O(lg N) worst case, but traversal from smallest element to largest takes O(N) time
Returns successor of x. If x is largest key of map, returns nil.	

`function Prev(x:TMSet.PNode):TMSet.PNode`	O(lg N) worst case, but traversal from largest element to smallest takes O(N) time
Returns predecessor of x. If x is smallest key of map, returns nil.	

`function IsEmpty(): boolean`	O(1)
Returns true when map is empty.	
`function GetValue(key:TKey):TValue`	O(lg N)
Returns value associated with key. Is key isn't in map crashes.	
`property item[i: Key]: TValue; default;`	O(ln N)
Property for accessing key i in map. Can be used just by square brackets (its default property).	