IqLib

Introduction to SAP Java Data Structures and Algorithms Library

SAP J2EE Engine 6.20

Contents

Overview

IqLib is a data structures and algorithms class library designed to provide fast and professional Java development. Along with the basic data structures, IqLib also provides advanced solutions, as well as efficient algorithms for processing them.

Features

IqLib offers an alternative of Sun's JDK, providing improved implementation of data structures that JDK includes and enhancing its functions significantly, with some advanced concepts:

- Data structures -
  IqLib also provides a set of data structures that includes both basic structures such as arrays and lists, and complex data structures, such as binary search trees, AVL Search trees, randomized search trees (treaps), hashtables, sets, and so on.
- Algorithms -
  IqLib also provides effective algorithms, which can be applied to each data structure using iterators
- Concurrency and synchronization primitives -
  SAP Java class library introduces solutions to problems with concurrency and synchronization of data processing. IqLib provides monitors, semaphores, barriers, lockers, and task executors that enable concurrent and parallel programming.

In addition, IqLib provides adapters to JDK so that both libraries can be used together.

Benefits of Using IqLib

IqLib is designed to offer solutions to three major problems that developers may encounter:

- Insufficient knowledge of complex data structures and algorithms developers are often familiar only with implementing simple data structures and algorithms that usually do not provide optimal performance. With IqLib even programmers that are inexperienced in Java can use more advanced and efficient solutions.

- Long development time -
  implementing complex data structures and algorithms takes time and slows down the development process. IqLib speeds up development, because SAP Java class library provides ready-to-use classes.
- Bugs -
  testing and debugging code is a difficult and time-consuming procedure. IqLib helps reduce the number of bugs by providing tested and reliable solutions.

Goals

IqLib provides Java developers with a useful set of efficient solutions. It achieves the following design goals:

- Easy usage -
  IqLib provides ease of use even for developers who are not experienced in Java
- Reusability -
  IqLib solutions can be reused in multiple programs and for a number of purposes. It saves the effort of implementing similar classes several times.
- Efficiency and high performance -
  IqLib solutions provide maximum efficiency and are optimized for high performance
- Good structure and design -
  IqLib structure and design has been enhanced to make the library consistent, efficient and easy to use
- Extensibility -
  IqLib functions can be extended by adding data structures and algorithms
- Flexibility -
  a single algorithm can be applied to various data structures
- Reliability and stability -
  these solutions are tested thoroughly to prove they are stable and reliable. In addition, each class is documented in detail.

Introduction

IqLib is a *Java generic data structure, algorithm, and multithreading library* used in SAP J2EE Engine. The problem domain on which IqLib is focused, is defined by Nicklaus Wirth's remarkable formula about structure programming:

```
Programming = Algorithms + Data Structures,
```

When applied to OOP, this is modified to the following:

```
Programming = Data Structures + Algorithms,
```

In OOP, with multithreading this becomes:

```
Programming = Data Structures + Algorithms + Multithreading
```

This triad of problems forms the skeleton of object-oriented programming in a multithreaded environment. Since multithreading implies synchronization between threads, IqLib provides a number of synchronization mechanisms to enable concurrent programming.

Packages

IqLib consists of the following packages:

- Data Structures
  `com.inqmy.lib.util.*`
- Algorithms
  `com.inqmy.lib.util.algorithms.*`
- Multithreading & Synchronization
  `com.inqmy.lib.util.synch.*`

Note: This structure presents the core part of IqLib, which is included in v.1.0 and provides basic functions. Future versions will provide advanced solutions.

Key Concepts

IqLib solutions are built on the basis of the following key concepts:

- Element -
  in IqLib, this corresponds to `java.lang.Object` in JDK
- Item -
  this is a wrapper of the element that enables you to store it in data structures. Item is the object that is used to create references between elements. It also provides for their cloning, both deep and shallow.
- Enumeration -
  provides a "snapshot" or static view of elements. It can be used to obtain successive elements from a series of elements.
- Iterator -
  a dynamic view of elements. Iterators are the interface between data structures and algorithms, because algorithms are applied over iterators but affect data structures.
- Comparator -
  presents the binary relation between elements. It can be used to sort elements in particular data structures that require total ordering.
- Data Structure -
  collections of elements. Each data structure imposes a specific organization of elements that is used for the efficient application of algorithms.
- Algorithms -
  sets of actions performed using iterators over data structures. They can either modify a data structure (for example, algorithms for sorting, adding, and removing elements) or simply find a particular element without changing the actual data structure (search algorithms). The use of iterators provides applicability of algorithms over various data structures.

Data Structures

JDK-Like Data Structures

IqLib is designed to be JDK-like to provide Java developers with a library in a style familiar to them, thereby making IqLib usage easier and less time consuming. However, IqLib enhances JDK functions significantly.

While JDK data structures (HashMap, Set, Vector, and so on) contain Objects only, IqLib contains similar data structures for native types:

- HashMapObjectObject, HashMapIntObject, HashMapIntInt, and so on
- Set. SetInt, SetLong, and so on
- ArrayObject, ArrayInt, ArrayLong, and so on
- Stack, Deque, Queue, as well as corresponding wait structures (*WaitQueue*, *WaitStack*, *WaitDeque*). The implementation of these structures blocks a thread that waits to add or delete an element as long as the data structure is full.

IqLib data structure implementations reduce memory usage drastically, thereby providing better performance.

In addition, algorithms can be applied over JDK structures using the adapters provided for them.

Note: At present, algorithms cannot be applied over structures for data types other than Object. This will be provided in a future release of IqLib.

IqLib offers a number of enhancements to JDK functions. For example, IqLib items use ItemAdapters to speed up performance. Limits could be defined for various data structures, iterators and enumerations to apply algorithms over the structures, and the algorithms that are provided present effective solutions and save time. One of the important features of IqLib is that it provides for "deep clone." Deep clone copies the entire structure of he object being cloned, while "shallow clone" shares the instance variables between the clone and the original.

Additional Data Structures

In addition to JDK-like data structures, IqLib provides implementations of advances data structures – lists and trees.

Lists

List is a dynamic ordered data structure that can consist of *n* elements. Dynamic implies that the value of *n* might change. The nodes of a list have linear order based on their position in the data structure. The first element is called *head* and the last one is called *tail*. IqLib includes the following implementations of lists:

Linked List

Linked list is a data structure that connects its elements using pointers. Each node of a linked list contains two fields: *data* (which holds the list element), and *next* (which stores the pointer to the next element). The *next* field of the last node of the list is `NULL`.

Linked List

IqLib provides implementation of *stack* and *queue* as linked lists. Stack is a "Last In First Out" (LIFO) dynamic set, while queue is a "First In First Out" (FIFO) dynamic set.

Doubly Linked List

The nodes of the *doubly linked list* contain three fields: *data*, *previous*, and *next.* The *previous* field stores a pointer to the preceding element. The *next* field holds a pointer to the following element. For this data structure, the previous element reference of the *head* node and the next element reference of the *tail* node are `NULL`.

Doubly Linked List

IqLib presents an implementation of doubly ended queue (*deque*). *Deque* is a data structure that combines the properties of a stack and a queue. Elements can be added to, or deleted from, both the head and the tail of the deque.

Trees

Trees are defined as connected acyclic graphs - the *root node* of the tree is the only one that has no *parent nodes*. The nodes that do not have *children*

are called *leafs* or *external* nodes; all nodes that have *child* nodes (that is, the *parent* nodes) are also called *internal*. Trees are *ordered* or *unordered*. In an *ordered tree*, the children of a node have a specific linear ordering, and can be referred to as first node, second node, and so on. This cannot be done for an unordered tree.

A specific type of ordered tree is *binary tree*. These are ordered trees in which each node has zero, one, or two children. The first and the second child of a node with two children are referred to as *left* and *right* child respectively. A binary tree is *full* when each of its nodes is either a leaf, or has two children. A *perfect* binary tree is a full binary tree, in which all leaves are of the same depth. To obtain a *complete* binary tree, the right-most leaf of a perfect binary tree is removed.

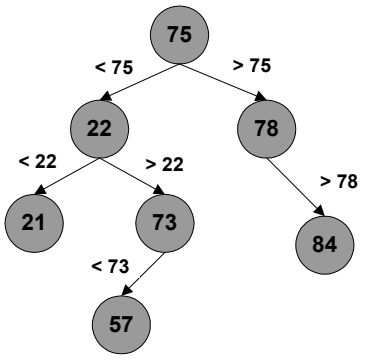
IqLib provides implementations of the following types of tree data structures:

- Binary Search Tree
- AVL Tree
- Treap

For these data structures, the running time of search, insert, and delete procedures is proportional to the height of the tree considered. Asymptotic analyses of these data structures show that their running time for insert, delete or search algorithms is approximately $O(\log_2 N)$ – that is, their height is $O(\log_2 N)$ where N is the number of nodes.

Binary Search Tree

Binary search trees are organized as binary trees, for which the associated key value of the left child element is less than the key value of the parent node, and the key value of the right child element is greater than that of the parent. This type of data structure combines binary search possibilities with good performance for insert and delete algorithms. Although this type of data structure provides a fairly simple way to find, insert, or delete nodes, the optimal $O(\log_2 N)$ height is not guaranteed because binary search trees are not height balanced.



Binary Search Tree

AVL Tree

AVL tree is a binary search tree that is strictly balanced – that is, the balance factor of each node is –1, 0, or 1. The balance factor of a node is computed by subtracting the height of its left subtree from the height of its right subtree. AVL trees provide for $O(\log_2 N)$ search time, although the running time of insert or delete algorithms is increased.

Treap

Randomised binary trees (treaps) are randomly balanced binary trees that prove to be even more efficient than AVL trees. The optimal algorithm running time ($O(\log_2 N)$) is achieved for both search and change (insert or delete) procedures.

Threadsafe Versus Nonthreadsafe Data Structures

Since multithreading is a vital part of object-oriented programming, IqLib provides dual implementations of data structures: both threadsafe and nonthreadsafe.

Threadsafe (synchronized) implementation guarantees correctness but it is slower than non-threadsafe implementation. You should use nonthreadsafe implementation for better performance when access involves reading only.

Note: Future enhancements of IqLib will include differentiation between read and write operations for threadsafe data structures.

An example of threadsafe versus nonthreadsafe implementation is `Set` (nonthreadsafe) and `ConcurrentSet` (threadsafe).

Iterators

Iterators are the interface between data structures and algorithms. Through appropriate iterators, algorithms can be made reusable over various data structures.

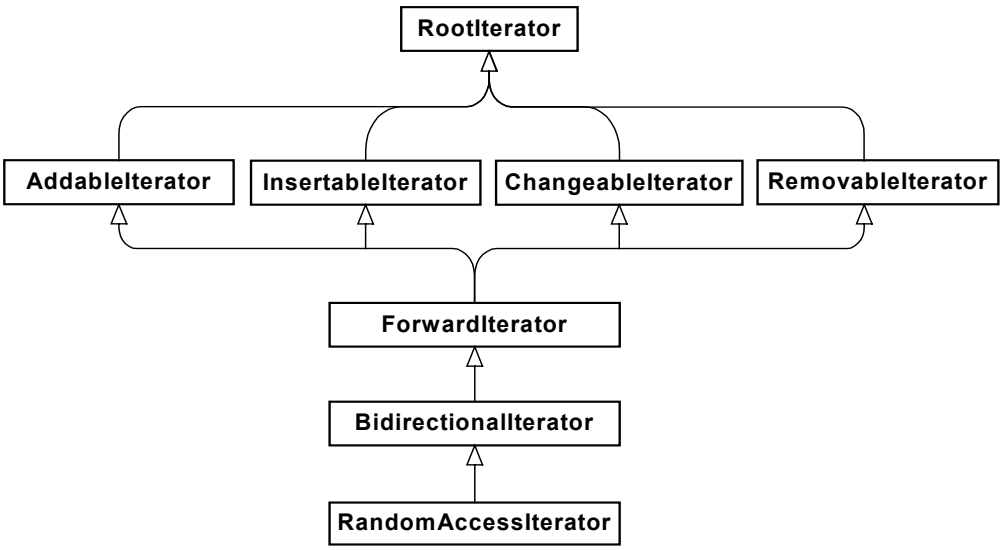
IqLib provides four interfaces that have methods for changing the underlying data structures. These are:

- AddableIterator -
  the interface for iterators that are used to add elements to data structures
- InsertableIterator -
  the interface for iterators used to insert elements in data structures at the position to which the iterator points
- ChangableIterator -
  the interface for iterators that are used to change the element to which the iterator points
- RemovableIterator
  the interface for iterators that are used to remove the element to which the iterator points

All of the interfaces listed above extend the `RootIterator` interface, which provides iterators' basic functions.

In addition, IqLib provides three more interfaces:

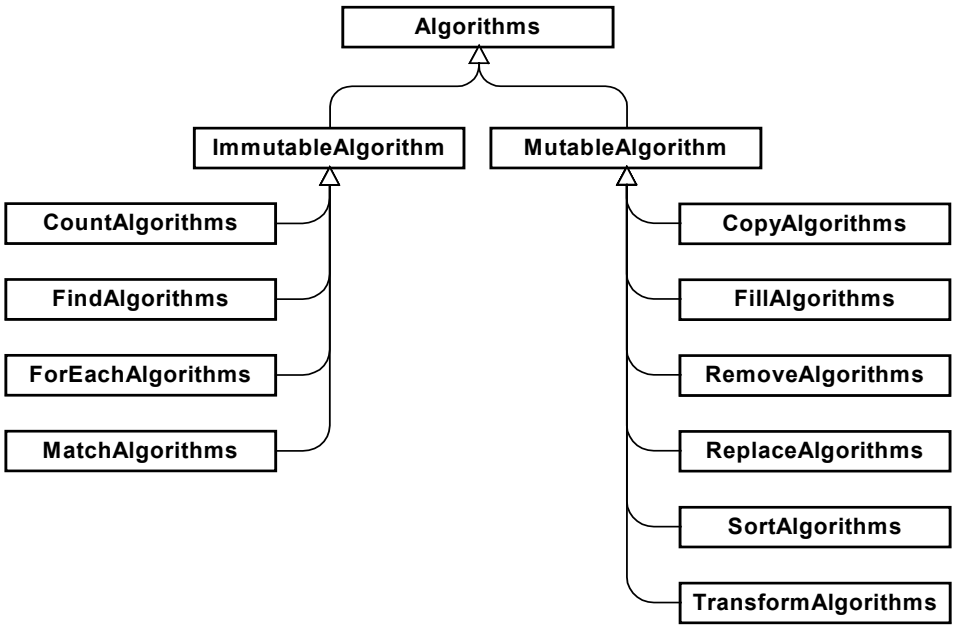
- ForwardIterator -
  this interface extends all of the interfaces mentioned above; `AddableIterator`, `InsertableIterator`, `ChangableIterator`, and `RemovableIterator`. It lets you browse data structures in a forward direction only.
- BidirectionalIterator -
  this interface extends `ForwardIterator`, but lets you browse the underlying data structure both backward and forward.
- RandomAccessIterator -
  this interface extends `BidirectionalIterator` and also provides two additional methods that return the current position of the iterator, and can point it at a randomly chosen position in the data structure


RootIterator
AddableIterator
InsertableIterator
ChangeableIterator
RemovableIterator
ForwardIterator
BidirectionalIterator
RandomAccessIterator

Algorithms

Algorithms are applied over iterators that provide access to the underlying data structure. In this way, data structures appear unified to algorithms, which can be reused many types over various types of data structures – that is, algorithms are *generic*.

IqLib includes two major types of algorithms: `MutableAlgorithm` and `ImmutableAlgorithm`. `MutableAlgorithm` abstract class is the parent class for all algorithms that modify the data structure - for example, `CopyAlgorithms`, `RemoveAlgorithms`, `SortAlgorithms`, and so on. `ImmutableAlgorithm` is the parent class for algorithms that do not involve changes in the data structure, such as `FindAlgorithms`, `MatchAlgorithms`, and so on.



- CopyAlgorithms -
  contains algorithms to copy one iterator to another, insert elements to other iterators, and swap the elements of two iterators
- FillAlgorithms -
  generates elements in a data structure
- RemoveAlgorithms -
  removes elements from iterators

- ReplaceAlgorithms -
  replaces elements of iterators
- SortAlgorithms -
  contains algorithms to sort iterators
- TransformAlgorithms -
  transforms iterators by applying functions over them
- CountAlgorithms -
  counts the number of occurrences of an element in a data structure
- FindAlgorithms -
  contains algorithms to find an element in a data structure
- ForEachAlgorithms -
  applies `UnaryFunction` over elements in data structure
- MatchAlgorithms -
  tests for equality of iterators

Multithreading and Synchronization

Multithreaded environment imposes synchronization of the access to variables because all threads share single memory. IqLib provides a variety of mechanisms for synchronization, as well as a framework for parallel and serial task execution.

Synchronization is achieved using event listeners for all synchronization primitives. IqLib synchronized implementations support different types of thread scheduling:

- JVM -
  the standard Java Virtual Machine thread-scheduling scheme (random)
- First In First Out (FIFO) -
  in accordance with this scheme, threads continue with their work in the order in which they have been put in waiting queue
- Priority -
  threads are "awoken" in the order, defined by a priority that is assigned to them
- NON_INFINITY_READ -
  this scheme guarantees that threads can perform the write operation
- NON_INFINITY_WRITE -
  this scheme guarantees that threads can perform the read operation

Task execution framework enables you to build a graph of tasks to be executed simultaneously or consecutively.

IqLib also connects to thread pooling, thereby improving performance by using reusing treads.

Synchronization Mechanisms

IqLib offers implementations of four mechanisms for synchronization: semaphores, barriers, monitors, and read-write lockers.

Semaphore

Semaphore is used when the number of resources (items) is less than the number of threads that have to be associated with the items. To synchronize the access to the resources (items), the semaphore is initialized to the number of items available. Threads request an item and are associated to one as long as there are free items available. As each thread occupies an item, the

number of available items is decremented by one. When there are no more resources available, the requesting threads wait until other threads release items.

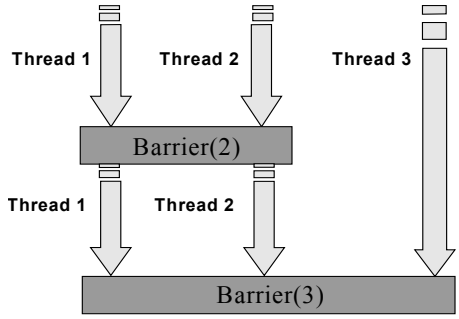
A semaphore that is initialized to a single item is called *mutex* (mutual exclusive lock).

Another type of semaphore is *countdown*. If countdown is used, all threads wait until the value of 0 is reached.

Barrier

Barrier is an object that is initialized to a certain number of threads (T) it controls. Threads arrive at the "barrier" point at a specified phase of their work. When reaching the barrier, threads wait until all T threads finish their work and then continue their computations. In distributed computing, threads can exchange information at the barrier point.

Countdown can also be perceived as a one-time barrier, as it cannot be reset.



Barrier

Monitor

Monitor is a standard JDK synchronization mechanism. It is a high-level synchronization primitive. The concept of monitor involves a single thread executing in the monitor at a time, while all other threads wait. When the thread that executes finishes its work, waiting threads can be signaled or broadcast to continue their work. If signaled (broadcast), a waiting thread is "re-activated" once the monitor becomes free.

Both monitors and semaphores are a proper solution of the "consumer-producer" problem, when one thread needs resources that are generated by

another thread. The first thread does not execute until the second one has finished executing.

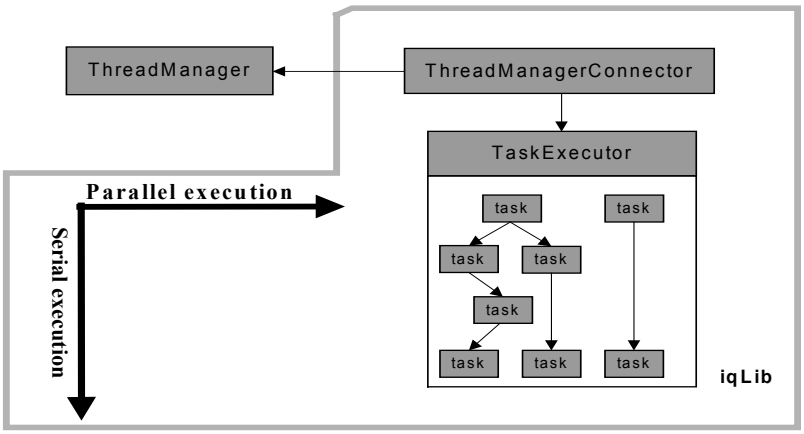
ReadWriteLocker

ReadWriteLocker is a synchronization mechanism, based on imposing read and write locks on resources. A single resource can be read-locked by multiple threads executing simultaneously, but you cannot impose other read or write locks on a resource that has already been locked for writing. Also, you cannot impose write locks on read-locked resources.

Task Framework

The IqLib concept of multithreading includes a framework for serial and parallel execution of tasks along with the various synchronization mechanisms. This framework lets you create a graph of tasks to be executed.

Task is a wrapper of Runnable object. It implies work to be executed. Task Executor is an object that supports a set of tasks and provides methods for their serial or parallel execution. Task Executor can connect to a Thread Manager or to a thread pool using *TheadManagerConnector*. It executes each task in a separate thread that could be assigned priority.



Task Framework

Future Enhancements

The following enhancements are planned in future versions of IqLib:

- The ability to apply algorithms over all types of data structures, not just over data structures for Objects
- Differentiation between read and write operations in threadsafe data structures
- Implementation of graphs and advanced trees
- Implementation of new algorithms
  - Setting intersection, union, difference
  - Permutations, variations, and combinations; s huffling, reversing and rotating
  - Algorithms over graphs (such as DFS, BFS, Djikstra, critical path, Floyd), and so on.