Dictionaries

IEC 61131 Library for ACSELERATOR RTAC® Projects

SEL Automation Controllers
Table of Contents

Section 1: Dictionaries

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Introduction</td>
<td>3</td>
</tr>
<tr>
<td>Supported Firmware Versions</td>
<td>4</td>
</tr>
<tr>
<td>Global Parameters</td>
<td>4</td>
</tr>
<tr>
<td>Aliases</td>
<td>4</td>
</tr>
<tr>
<td>Structure Definitions</td>
<td>4</td>
</tr>
<tr>
<td>Classes</td>
<td>5</td>
</tr>
<tr>
<td>Benchmarks</td>
<td>9</td>
</tr>
<tr>
<td>Examples</td>
<td>11</td>
</tr>
<tr>
<td>Release Notes</td>
<td>15</td>
</tr>
</tbody>
</table>
Introduction

This library implements a collection of data structures for storing key value pairs. This allows for storing of information indexed by a unique key string.

Determine which data structure to use by looking at the characteristics of the available structures, and choose the one best suited to the job and environment at hand.

This library supplies a single implementation. It is a self-balancing binary search tree as described in class_BinaryTreeDictionary on page 5.

The iterators in this document all refer to being locked out. This refers to the state of the object being such that a non NULL(0) value cannot be retrieved from Next() without a new call to Begin().

Special Considerations

▷ Classes in this library have memory allocated inside them. As such, they should only be created in environments of permanent scope (e.g., Programs, Global Variable Lists, or VAR_STAT sections).

▷ Copying classes from this library causes unwanted behavior. This means the following:

1. The assignment operator “:=” must not be used on any class from this library; consider assigning pointers to the objects instead.

```plaintext
// This is bad and in most cases will provide a compiler error such as:
// "C0328: Assignment not allowed for type class_Object"
myObject := otherObject;

// This is fine
someVariable := myObject.value;
// As is this
pt_myObject := ADR(myObject);
```
2. Classes from this library must never be VAR_INPUT or VAR_OUTPUT members in function blocks, functions, or methods. Place them in the VAR_IN_OUT section or use pointers instead.

## Supported Firmware Versions

You can use this library on any device configured using ACSELERATOR RTAC® SEL-5033 Software with firmware version R143 or higher.

Versions 3.5.0.1 and older can be used on RTAC firmware version R132 and higher.

## Global Parameters

The library applies the following values as maximums; they can be modified when the library is included in a project.

<table>
<thead>
<tr>
<th>Name</th>
<th>IEC 61131 Type</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>g_p_KeyStringLength</td>
<td>UINT</td>
<td>80</td>
<td>The maximum string length for a key.</td>
</tr>
</tbody>
</table>

## Aliases

This section lists aliases defined by this library.

### DATA_VAL

<table>
<thead>
<tr>
<th>ALIAS</th>
<th>IEC 61131 Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>DATA_VAL</td>
<td>__XWORD</td>
</tr>
</tbody>
</table>

## Structure Definitions

This section lists structures defined by this library.

### struct_KeyValuePair

This structure is a simple storage object for holding key-value pairs.

<table>
<thead>
<tr>
<th>Name</th>
<th>IEC 61131 Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key</td>
<td>STRING(g_p_KeyStringLength)</td>
<td>A key associated with a value.</td>
</tr>
<tr>
<td>Data</td>
<td>DATA_VAL</td>
<td>Data storage.</td>
</tr>
</tbody>
</table>
Classes

This section contains the basic definitions, descriptions, and public methods for the public classes that can be instantiated by the user.

class_BinaryTreeDictionary

This class provides a self-balancing binary search tree that stores key-value pairs. To allow this class to accommodate various data types, the value stored is a DATA_VAL, which can store a single 32-bit value or a pointer to a user-defined data structure.

A binary search tree ensures arrangement of all nodes in order by key such that, given a node, all keys in the left subtree are less than the key of the given node and all keys in the right subtree are greater than the key of the given node (Figure 1).

![Figure 1 A Binary Search Tree Holding Integer Values](image)

Binary search trees provide insert, search, and deletion times that are related to the number of items in the tree \(N\) by \(\log(N)\) on average. Under some circumstances, the organization of the simple tree yields much worse performance. Consider a tree created by inserting the keys C, K, and then L (as shown in Figure 2).

![Figure 2 An Unbalanced Binary Search Tree](image)

Note that the nodes are arranged linearly, rather than as one parent with two children. This causes the behavior of all operations to tend toward a linear performance curve, as opposed to the \(\log(N)\) described previously. To prevent the performance degradation of an unbalanced tree, the binary tree supplied implements a self-balancing algorithm. If inserting or deleting a node leaves the tree unbalanced, the self-balancing tree performs rotations and moves of the nodes in the tree to maintain balance. (Figure 3).
Properties

<table>
<thead>
<tr>
<th>Name</th>
<th>IEC 61131 Type</th>
<th>Access</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size</td>
<td>UDINT</td>
<td>R</td>
<td>The number of key-value pairs stored in this tree.</td>
</tr>
</tbody>
</table>

Properties are internal values made visible through Get and Set accessors. Access is defined as R (read), W (write), or R/W (read/write).

GetData (Method)

This method provides the data associated with the provided key.

Inputs

<table>
<thead>
<tr>
<th>Name</th>
<th>IEC 61131 Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>key</td>
<td>STRING(g_p_KeyStringLength)</td>
<td>The key for the desired value.</td>
</tr>
</tbody>
</table>

Outputs

<table>
<thead>
<tr>
<th>Name</th>
<th>IEC 61131 Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>data</td>
<td>DATA_VAL</td>
<td>The value stored at this key. This value is only valid if the return value is TRUE.</td>
</tr>
</tbody>
</table>

Return Value

<table>
<thead>
<tr>
<th>IEC 61131 Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BOOL</td>
<td>TRUE if the key provided is found in the binary tree, FALSE otherwise.</td>
</tr>
</tbody>
</table>

Processing

Returns the data associated with the provided key.

Insert (Method)

This method inserts a new value into the binary tree.
Inputs

<table>
<thead>
<tr>
<th>Name</th>
<th>IEC 61131 Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>key</td>
<td>STRING(g_p_KeyStringLength)</td>
<td>The key for the desired value.</td>
</tr>
<tr>
<td>data</td>
<td>DATA_VAL</td>
<td>Data to store in the binary tree.</td>
</tr>
</tbody>
</table>

Return Value

<table>
<thead>
<tr>
<th>IEC 61131 Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BOOL</td>
<td>TRUE if the key-value pair was successfully added to the tree. FALSE otherwise.</td>
</tr>
</tbody>
</table>

Processing

If key already exists in the tree, data replaces the data stored in key. If key does not already exist in the tree, a new node that stores both key and data is inserted into the tree. Depending on the state of the tree, the insertion may cause the tree to rebalance.

Delete (Method)

This method removes the key-value pair from the binary tree.

Inputs

<table>
<thead>
<tr>
<th>Name</th>
<th>IEC 61131 Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>key</td>
<td>STRING(g_p_KeyStringLength)</td>
<td>The key for the desired value.</td>
</tr>
</tbody>
</table>

Return Value

<table>
<thead>
<tr>
<th>IEC 61131 Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BOOL</td>
<td>TRUE if the key-value pair was found and deleted.</td>
</tr>
</tbody>
</table>

Processing

This method deletes a key-value pair from the binary search tree. Depending on the state of the tree after deletion, the tree may be rebalanced to maintain lookup performance.

Clear (Method)

This method empties the binary tree.
Processing

This method completely empties the binary tree. It frees any memory allocated to the binary tree. Upon completion of this method, the binary tree object is of size zero and cannot be iterated over.

Begin (Method)

Use this method in conjunction with Next(), NextValue(), and NextKey(). This method places the internal iterator on the first key-value object.

Processing

After this method completes, the following are true:

- The iterator is not locked out.
- A subsequent Next() outputs the first key-value object.
- For an empty tree, Next() returns FALSE and leaves the iterator locked out.

Next (Method)

Use this method in conjunction with Begin(). Next() returns the key-value pair at the present internal iterator position and then increments the iterator.

Outputs

<table>
<thead>
<tr>
<th>Name</th>
<th>IEC 61131 Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>entry</td>
<td>struct_KeyValuePair</td>
<td>The key-value pair at the present iterator position. If the end of the iterator has been reached, key is an empty string and data is zero.</td>
</tr>
</tbody>
</table>

Return Value

<table>
<thead>
<tr>
<th>IEC 61131 Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BOOL</td>
<td>TRUE if a key-value pair was found. FALSE otherwise.</td>
</tr>
</tbody>
</table>

NextKey (Method)

Use this method in conjunction with Begin(). NextKey() returns the key at the present internal iterator position and then increments the iterator.

Outputs

<table>
<thead>
<tr>
<th>Name</th>
<th>IEC 61131 Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>key</td>
<td>STRING(g_p_KeyStringLength)</td>
<td>The key at the present iterator position. If the end of the iterator has been reached, key is an empty string.</td>
</tr>
</tbody>
</table>
Return Value

<table>
<thead>
<tr>
<th>IEC 61131 Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BOOL</td>
<td>TRUE if a key-value pair was found. FALSE otherwise.</td>
</tr>
</tbody>
</table>

**NextValue (Method)**

Use this method in conjunction with `Begin()`. `NextValue()` returns the value at the present internal iterator position and then increments the iterator.

**Outputs**

<table>
<thead>
<tr>
<th>Name</th>
<th>IEC 61131 Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>DATA.VAL</td>
<td>The value at the present iterator position. If the end of the iterator has been reached, value is zero.</td>
</tr>
</tbody>
</table>

Return Value

<table>
<thead>
<tr>
<th>IEC 61131 Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BOOL</td>
<td>TRUE if a key-value pair was found. FALSE otherwise.</td>
</tr>
</tbody>
</table>

**Size (Property)**

This method provides the number of nodes within the tree.

Return Value

<table>
<thead>
<tr>
<th>IEC 61131 Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UDINT</td>
<td>The number of nodes within the tree.</td>
</tr>
</tbody>
</table>

**Benchmarks**

**Benchmark Platforms**

The benchmarking tests recorded for this library are performed on the following platforms.

- SEL-3530
  - R134 firmware
- SEL-3354
  - Intel Pentium 1.4 GHz
  - 1 GB DDR ECC SDRAM
SEL-3532 RTAC Conversion Kit
R132 firmware

SEL-3555
- Dual-core Intel i7-3555LE processor
- 4 GB ECC RAM
- R134-V0 firmware

SEL-3555
- Dual-core Intel i7-3555LE processor
- 4 GB ECC RAM
- R134-V0 firmware

Benchmark Test Descriptions

Each of these tests is run on a tree of 1024 entries. The test attempts to make an unbalanced tree by inserting values in order, forcing the tree to continually rebalance itself. Each of the following tests is repeated 100 times, and the total average of all samples is recorded.

For example, the test in Insert records the average of the 1024 • 100 insertions.

Insert
This records the average time taken to insert 1024 sorted key-value pairs into the tree. The test is repeated 100 times and the average time taken for a single execution of Insert() is recorded.

GetData
This test calls GetData() on each of 1024 entries in the tree. The test is repeated 100 times and the average time taken for a single execution of GetData() is recorded.

Delete
This test calls Delete() 1024 times on a populated tree. The test is repeated 100 times and the average time taken for a single execution of Delete() is recorded.

Clear
This test records the average time required to clear the tree populated with 1024 nodes. The test is repeated 100 times and the average time taken for a single execution of Clear() is recorded.
**Begin**

This test records the time required to reset the iterator. `Begin()` is called 1024 times on a populated tree. The test is repeated 100 times and the average time taken for a single execution of `Begin()` is recorded.

**Next**

This test iterates across a full tree of 1024 nodes. The test is repeated 100 times and the average time taken for a single execution of `Next()` is recorded.

**NextKey**

This test iterates across a full tree of 1024 nodes. The test is repeated 100 times and the average time taken for a single execution of `NextKey()` is recorded.

**NextValue**

This test iterates across a full tree of 1024 nodes. The test is repeated 100 times and the average time taken for a single execution of `NextValue()` is recorded.

**Benchmark Results**

Values less than one microsecond have been rounded up.

<table>
<thead>
<tr>
<th>Operation Tested</th>
<th>Platform (time in µs)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>SEL-3530</td>
</tr>
<tr>
<td>GetData</td>
<td>14</td>
</tr>
<tr>
<td>Insert</td>
<td>796</td>
</tr>
<tr>
<td>Delete</td>
<td>799</td>
</tr>
<tr>
<td>Clear</td>
<td>779128</td>
</tr>
<tr>
<td>Begin</td>
<td>3</td>
</tr>
<tr>
<td>Next</td>
<td>2</td>
</tr>
<tr>
<td>NextKey</td>
<td>4</td>
</tr>
<tr>
<td>NextValue</td>
<td>1</td>
</tr>
</tbody>
</table>

**Examples**

_These examples demonstrate the capabilities of this library. Do not mistake them as suggestions or recommendations from SEL._

_Implement the best practices of your organization when using these libraries. As the user of this library, you are responsible for ensuring correct implementation and verifying that the project using these libraries performs as expected._
Ordered Data Retrieval

A user has data that she needs to present in an ordered fashion. She can define the order she needs through the keys, update the values as needed, and then present the data, all while maintaining the same order.

Solution

First the user initializes the full tree. Later, she can iterate across the entire structure to receive those data in key alphabetical order.

```
PROGRAM prg_SortedLookup
VAR
  MyBinaryLookupTree : class_BinaryTreeDictionary;
  CurrentData : struct_KeyValuePair;
  Initializing : BOOL := TRUE;
  Check : BOOL := TRUE;
END_VAR

IF Initializing THEN
  //First put the data into the tree as key value pairs
  MyBinaryLookupTree.Insert('Boxes', 1250);
  MyBinaryLookupTree.Insert('TapeRolls', 200);
  MyBinaryLookupTree.Insert('Pallets', 13);
  MyBinaryLookupTree.Insert('BubbleWrap', 75);
  Initializing := FALSE;
ELSE
  MyBinaryLookupTree.Begin();
  WHILE Check DO
    Check := MyBinaryLookupTree.Next(entry => CurrentData);
    IF CurrentData.data <> 0 THEN
      // Do some meaningful work
      END_IF
  END_WHILE
END_IF
```

Creating a Quick Lookup Table

Objective

A user has a collection of data he desires to look up quickly based on unique description strings. He needs to store it now and use parts of it later based on system state.
Assumptions

This example assumes that there is a user-specified IEC 61131 data type that is defined as shown in Code Snippet 2 and a function using that particular structure as shown in Code Snippet 3.

Code Snippet 2  struct_JobDefinition

```
TYPE struct_JobDefinition:
  STRUCT
    JobName : STRING(32);
    Duration : UDINT;
    Input : REAL;
  END_STRUCT
END_TYPE
```

Code Snippet 3  fun_DoWork

```
FUNCTION fun_DoWork : BOOL
  VAR_IN_OUT
    pt_currentCommand : POINTER TO struct_JobDefinition;
  END_VAR

; //Program the work that should be done here
```

Solution

First the user initializes the full tree. Later, based on some request, the required data can be retrieved.
PROGRAM prg_BinaryTree
VAR
  CurrentData : BOOL;
  JobSelector : INT;
  Initializing : BOOL := TRUE;
  Working : BOOL := FALSE;

  MyBinaryLookupTree : class_BinaryTreeDictionary;
  CurrentJob : STRING(g_p_KeyStringLength);
  pt_CurrentData : POINTER TO struct_JobDefinition;

  Job1Data : struct_JobDefinition :=
    (JobName := 'My First Job', Duration := 10, Input := 17.5);
  Job2Data : struct_JobDefinition :=
    (JobName := 'My Second Job', Duration := 5, Input := 31.75);
  Job3Data : struct_JobDefinition :=
    (JobName := 'My Third Job', Duration := 30, Input := 3.25);
  IdleData : struct_JobDefinition :=
    (JobName := 'No Current Job', Duration := 0, Input := 0);
END_VAR

IF Initializing THEN
  //First put the data into the tree as key value pairs
  MyBinaryLookupTree.Insert('Job1', ADR(Job1Data));
  MyBinaryLookupTree.Insert('Job2', ADR(Job2Data));
  MyBinaryLookupTree.Insert('Job3', ADR(Job3Data));
  MyBinaryLookupTree.Insert('Idle', ADR(IdleData));
  Initializing := FALSE;
  Working := TRUE;
END_IF

CASE JobSelector OF
  1: CurrentJob := 'Job1';
  2: CurrentJob := 'Job2';
  3: CurrentJob := 'Job3';
ELSE
  CurrentJob := 'Idle';
END_CASE

IF Working THEN
  CurrentData := MyBinaryLookupTree.GetData(CurrentJob, data =>
    pt_CurrentData);
  fun_DoWork(pt_CurrentData);
END_IF

Code Snippet 4  prg_BinaryTree
## Release Notes

<table>
<thead>
<tr>
<th>Version</th>
<th>Summary of Revisions</th>
<th>Date Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.5.1.0</td>
<td>Allows new versions of ACSELERATOR RTAC to compile projects for previous firmware versions without SEL IEC types “Cannot convert” messages. Must be used with R143 firmware or later.</td>
<td>20180921</td>
</tr>
<tr>
<td>3.5.0.1</td>
<td>Initial release.</td>
<td>20140811</td>
</tr>
</tbody>
</table>