naryTree XFCN:
A Free HyperCard Utility

By Ari Halberstadt

ABSTRACT

An external function implementation of a general purpose binary tree data structure for HyperCard (on the Macintosh). Nodes in a tree are referred to using a key and each node can contain a datum field. Operations implemented include insert, delete, find, and tree traversal as well as range searching. Several rules for inserting keys are implemented, including exact, ignorecase, and numeric. Source code in C is included. The program is free; for distribution terms see the appropriate sections in the file “Common Manual”.

This manual is intended for people who write scripts for HyperCard and who have some understanding of binary trees.

Contents

Sections

Ap

Figures

Tables

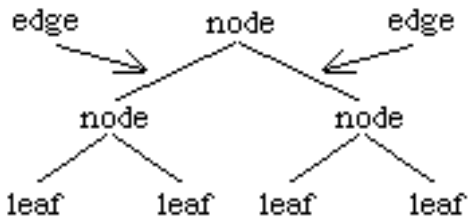
Scripts

Introduction

Binary trees

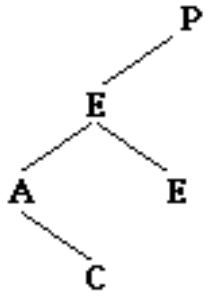
The BinaryTree external function implements ordered binary trees, along with the basic operations needed to manipulate them. The simplest way to explain a binary tree is with a diagram:

Figure 1. **Elements of a tree**



A binary tree contains nodes, and each node—which can be thought of as a parent of other nodes—has two edges leaving it: one goes to its left child, and the other goes to its right child. A node may have only one child or no children at all. A node that has no children is called a leaf. Each node may be assigned a value, which is then used to locate the node and to position it in the tree. An ordered binary tree has the property that the value of the left child is smaller than the value of the node, and the value of the right child is greater than (or equal to) the value of the node. (The BinaryTree external function places items with the same value into the right child.) The following figure shows an ordered binary tree, with letters indicating the values of the nodes (the tree was created by inserting the letters "PEACE" into an initially empty tree):

Figure 2. **An ordered binary tree**



Uses

Binary trees are excellent for maintaining data that change frequently. It is possible to insert a new node and to delete an existing node in an average time of O(lgN). It is also possible to traverse a tree, ie, visit all the nodes in a tree, in several different ways. For instance, recursively visiting the left child, then the node itself, and then the right child, is called in-order traversal (we stop when we reach a leaf). If a tree is traversed in in-order, and the values of the nodes printed as they are visited, then the output will be in sorted order.

There are many other things that can be done with binary trees. For more information, you can read almost any introductory data structures and algorithms text; I recommend the text by Robert Sedgewick, “Algorithms”, 2nd ed., Addison-Wesley (1988).

Using BinaryTree

Naming

Trees are always referred to by a name, which may be any valid HyperTalk string cosisting of letters, digits, and underscores. Capitalization does not matter in a name; thus, "TREE" is considered the same as "tree". Nodes in a tree are referred to by the node's key, which may be any HyperCard string. Each node may also contain a datum field which contains any HyperCard string; the datum field is empty by default.

Creating

Before a tree is used it must be created using the New function. The tree will then continue to exist until it is explicitly disposed of using the Dispose function. BinaryTree maintains a single internal index to all trees created by it, which means that all trees are globally accessible throughout your scripts.

Syntax

All functions implemented on a tree are executed from a single external function [XFCN]. When calling BinaryTree, the basic syntax is

```
btree(function[, parameters])
```

The first parameter always selects the function to perform on the tree, and subsequent parameters vary with each function.

Result

HyperCard expects an XFCN to return a value, and the calling script is required to put this return value somewhere. Since all of the operations in this program are part of a single XFCN, the script must always do something with the result returned, even if nothing useful is returned. It's simplest to use HyperTalk's get keyword, which will place the result into the temporary variable it. For instance,

```
get btree(new, tree) -- create a new tree
if (it ≠ empty) then return it -- return error code
-- continue with script
```

Efficiency

The delay between the time an XFCN is called and the time it starts to execute can be quite noticeable, especially when many calls are made to the XFCN. BinaryTree provides several means by which the number of individual calls may be reduced. Specifically, it allows for a condensed form of passing or getting many parameters from a single function. For instance, to retrieve the datum field from a single node of a tree, the following statement could be used:

```
get btree(get, tree, key) -- get datum for node with given key
```

whereas to retrieve the data fields of several nodes the following statement is more efficient:

```
get btree(get, tree, "key1,key2,key3")
```

The result in the latter statement will be separated by commas, though a different separator can be requested.

Nodes in a tree

This section discusses the various fields associated with a node. Each node has a key field and a datum field: the key field is used for locating and inserting nodes in the tree, while the datum field may store any additional information. Also, each node has a unique index assigned to it.

Sample tree

The illustrations in this section show several versions of the same tree, which was created using the following script, with text from the poem “Father William” by Lewis Carroll:

Script 1. **MakeExampleTree**

```
on makeExampleTree tree
      -- setup keys
      put "You are old Father William the young man said And your
hair has become very white And yet you incessantly stand on your
head Do you think at your age it is right" into keys
      -- setup data
      put "In my youth Father William replied to his son I feared it
might injure the brain But now that I'm perfectly sure I have none
Why I do it again and again" into data
      -- create a new tree
      get btree(new, tree)
      -- insert keys and data; each key and datum is separated
      -- with a space.
      get btree(insert, tree, keys, data, space)
end makeExampleTree
```

The separator character was set to space so that the input would be broken into words. All punctuation was removed so that it wouldn't get inserted into the tree.

Key field

The key field is used for inserting new nodes and determines the structure of the tree. The key field is also the only way to refer to a specific node. The key field is set when a new node is inserted into the tree, and it may not be changed thereafter. The only way to get rid of a key is by deleting its node using the Delete function.

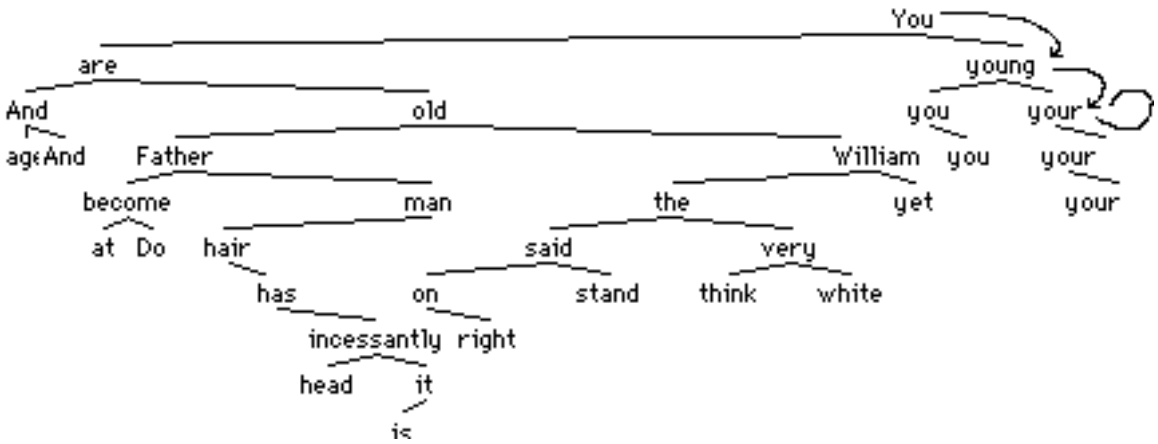
Specifying a key

When a function calls for a key parameter, you must provide the complete text of the key, though there may be a range of strings which can match the key, depending on the value of the compare attribute. For instance, if the value of the tree's compare attribute is exact, then to refer to the key "How are you!", you must enter the entire string "How are you!", exactly as shown. If the compare attribute was instead Ignorecase, then you could enter "HOW are YOU!", "how ARE you!', etc. (You must still include the punctuation.)

Keys illustrated

The following figure contains the keys from the example tree. Notice that upper and lower case letters have been inserted into the tree correctly, since the tree was created using the default value of the compare attribute (Ignorecase). Also, remember the basic property of ordered binary trees: all left children are smaller than the root, and all right children are greater than or equal to the root.

Figure 3. **Father William Keys**



Duplicates

Observe how duplicate keys were inserted, such as "You" and "you". When searching for a node (using any of the commands that refer to a node by its key), only the first occurrence encountered of a node matching that key is ever seen. Thus, if you used a command such as:

```
get btree(get, tree, "you")
```

you would always get the datum at the root of the tree, which is "In". To get all occurrences of the word "you", use the Range search command:

```
get btree(range, tree, "you", "you")
```

which will return "You,you,you".

Traversing a tree

Since the search normally terminates at the first occurrence of a key, it is not possible to traverse a tree containing duplicate keys using the Left and Right commands. For instance, consider what would happen to the following recursive script to traverse our “Father William” tree:

```
on traverse tree, node
```

```
      if (node ≠ empty) then
            traverse tree, btree(right, tree, node)
            traverse tree, btree(left, tree, node)
      end if
end traverse
```

the traverse handler is called with the command

```
traverse tree, btree(root, tree)
```

First, traverse will call itself with node equal to "young", then it will call itself with node equal to the first "your", and then again it will call itself with node equal to the **first** "your". This function will thus encounter an infinite loop and will be abnormally terminated by HyperCard. The first few steps taken by this function are shown with the arrows in the figure.

Solution

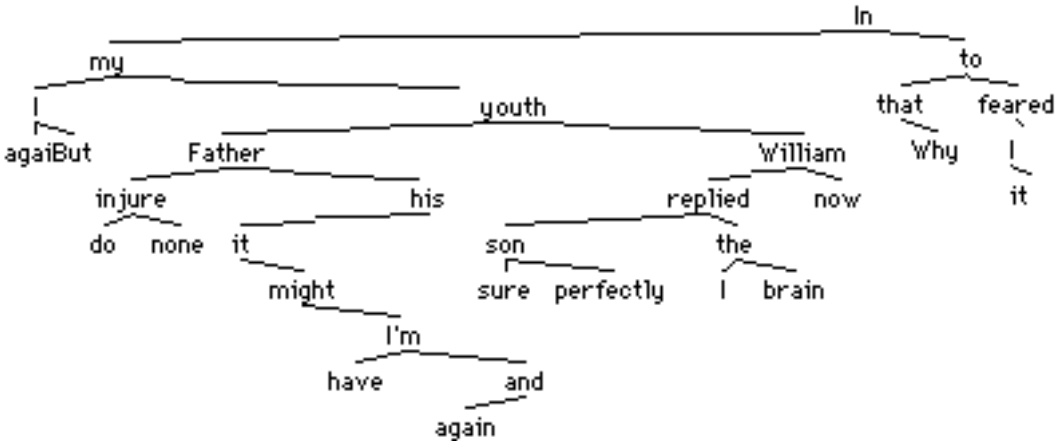
The above example raises an important problem: how is it possible to traverse a tree? The solution is to avoid recursion in HyperTalk by having BinaryTree traverse the tree for us. Since a tree may be represented as an array, we use level order traversal (or whatever order is appropriate) to get the indexes into this array. We then do another traversal using the same order to get the keys or data fields of the tree. A repeat loop can then scan through the array and execute the desired operations. An example of several scripts implementing this method may be found in the DrawTree card of the demonstration stack.

Datum field

The datum field may be filled with any string the user wants to put into it, and is empty by default. Each datum field is permanently associated with the node to which it belongs. You may change the contents of a datum field at any time using the Set function, and get its value using the Get function.

The next figure shows the datum fields for our example tree. Notice that the data do not follow the rules that the keys in the tree follow: the data are positioned in the tree only according to the order in which the Insert command encountered them, so that each datum corresponds to a specific key. Also notice that the trees in this figure and the previous figure have the same shape, since they are both depicitons of different information taken from the same tree.

Figure 4. **Father William Data**



Indexes

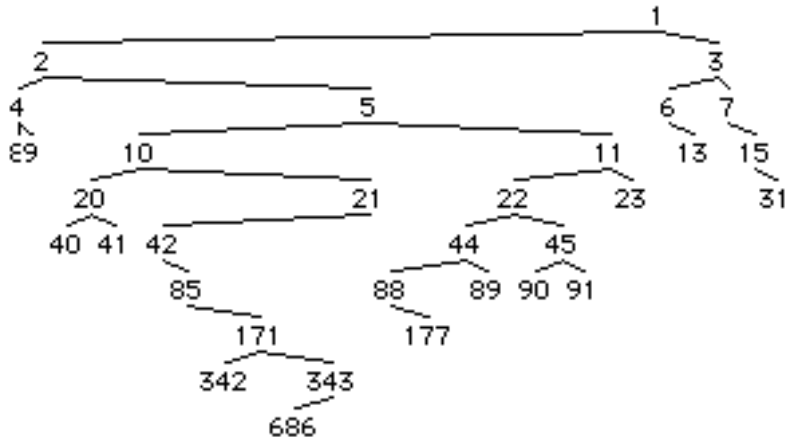
A node's index is an integer which is implicitly associated with the node according to its position in the tree. Therefore, any operation that changes the structure of the tree, such as deletion of a node, may change the index associated with any specific key.

Calculating indexes

The index of the root of a tree is always **1**, the left child of a node with index **i** has index **2*i** and the right child has index **2*i+1**. The parent of the node has index **i div 2**, while the depth of the node is **trunc(log2(i))**. If you specify a different root node to any of the tree traversal functions or to the range searching function then that node is assigned index number 1, instead of the actual root of the tree.

The following figure shows the indexes into the example tree. The root of the tree has index 1, its left child index 2 and its right child index 3. The indexes continue sequentially until we skip index 12, which would have been the left child of the node with index 6 (since 2*6=12).

Figure 5. **Father William Indexes**



Attributes

Every tree has some attributes used to modify the way the tree behaves, add extra functionality to a tree, and make some operations more efficient. The functions SetAttribute and GetAttribute are used to set and get the attributes' values. Attributes may have different types, such as Boolean or integer. Every attribute also has a default value that is set when a tree is created. Following is a table giving the names, types, and default values for all of the attributes. Following the table are descriptions of each of the attributes.

Table 1. **Attributes**

Name	Value	Default
Sorted	True	False
	False	
Compare	Exact	Ignorecase
	Ignorecase	
	International	
	Numeric	

Unique

Only unique instances of each key are allowed. When inserting a key into a

tree, a check is made to see if the key already exists. If it does, then the key is not inserted and an error code is returned. Duplicate keys in a tree can interfere with some operations; see the discussion of keys for more details.

Compare

The compare attribute controls the rules for comparing items when searching for a key and when inserting keys into a tree. The value of this attribute is set when a tree is created using the New function, and can not be changed thereafter. Since a tree can be output in sorted order when traversed in in-order, these key comparison rules may be viewed as modifying the sorting rules for a tree. Descriptions of each of the possible values are given below.

Exact

Keys are compared exactly as they are, and strict ASCII ordering is used when inserting items. For instance, the following items are in sorted order: "Aardvark, Hello, ^[\, me, them, you". These ordering rules are obviously not suitable for a tree containing text.

Ignorecase

Upper and lower case letters are correctly compared and inserted. When searching for a key, distinctions between upper and lower case letters are ignored, so that "UPPER" is considered the same as "upper". When inserting, upper case letters are considered smaller than lower case letters. When the tree is output in sorted order, letters will all be grouped together, but upper case letters will come before lower case letters. For instance, the following items are in sorted order: "aardvark, UPPER, Upper, upper". Character case is not ignored for letters with diacritics, so that "å" is not the same as "Å", even though "a" would be the same as "A".

International

Correctly compares and inserts non-English text containing diacritical marks and special characters, depending on the international resources in your System file. This is similar to the international style of HyperCard's sort command.

Numeric

Compares and inserts keys numerically. Any white spaces preceding the keys are ignored (eg, spaces, tabs, returns). The comparison is first done on the numeric component of the keys, and then, if the keys are equal, a sub-comparison is done on any non-numeric characters following the keys. For instance, the key “4a” is smaller than the key “4b”. The capitalization of any extra characters is ignored, so that the string “5B” would match the string “5b”.

Traversing a tree

Traversing a tree means visiting every node in the tree in some specific order. As each node is visited, we can perform some action, such as printing its contents. For instance, one method of traversing a tree is to recursively visit the left subtree, then the right subtree, and then the root node. This method is called in-order traversal.

Algorithms

BinaryTree provides all the basic algorithms for tree traversal: in-order, pre-order, post-order, and level-order. These functions, by default, start at the root of the tree and return a comma separated list of the keys of the nodes encountered. It is also possible to request the data or the indexes of the nodes encountered, to specify a different output separator, and to start from a node other than the root node. These functions should be sufficient for nearly all tree traversal problems. If you must traverse the tree using a different algorithm, you can use the Left, Right, Root, and Parent functions to access nodes.

Note: If a tree contains duplicate keys then the Left, Right, and Parent functions can not be used to traverse the tree. This problem is discussed in the section above about keys.

Function descriptions

Functions

This section contains an alphabetical list of all of the functions implemented.

"!"

Syntax

```
string btree("!")
```

Description

Returns a string giving the version of BinaryTree, the full name of the program, the author, a copyright notice, and the date and time of compilation. The string has the basic form “BinaryTree XFCN, Version 0.9, by Ari Halberstadt, Copyright © 1990, date time”.

Examples

```
get btree("!")
```

"?"

Syntax

```
string btree("?")
```

Description

Returns a string giving a brief summary of the functions and call syntax for BinaryTree.

Examples

```
get btree("?")
```

Delete

Syntax

```
error btree(Delete, tree, key[, separator])
```

Description

Deletes the node with the given the *key* from the *tree*. If the *separator* character

is given then each key delimited by the separator character is deleted from the tree.

Notes

Each key must exist in the tree.

Examples

```
get btree(delete, tree, key)
      -- deletes a single key

get btree(delete, tree, "key1,key2,key3", ",")
      -- deletes keys "key1", "key2", and "key3"
```

Dispose

Syntax

`error btree(Dispose, tree)`

Description

Completely disposes of all of the nodes in the *tree* and of the *tree* itself. Call this function where you are completely finished with a tree.

Examples

```
get btree(dispose, tree)
```

Error

Syntax

`error btree(Error)`

Description

Returns the error code set by the last function executed. If the last function was executed successfully then returns empty.

Examples

```
get btree(error)
```

Get

Syntax

`string btree(Get, tree, key[, keys_separator[, data_separator]])`

Description

Returns the datum field for the node with the given *key*. If the *keys_separator* character is given then Get is called for every key delimited by the separator. The output items are separated by *keys_separator*, unless *data_separator* is given, in which case the output items are separated by the data separator.

Examples

```
get btree(get, tree, key)
      -- gets the datum field of the node with the given key.
```

```
get btree(get, tree, "key1,key2,key3", ",")
      -- gets the data fields for the nodes whose keys are
      -- "key1", "key2", and "key3". The data are separated
      -- by commas, for instance: "datum1,datum2,datum3".

get btree(get, tree, "key1,key2,key3", ",", ";")
      -- gets the data fields for the nodes whose keys are
      -- "key1", "key2", and "key3". The data are separated
      -- by semicolons, for instance: "datum1;datum2;datum3".
```

GetAttribute

Syntax

`string btree(GetAttribute,` *tree*, *attribute*`)`

Description

Returns the value of the named *attribute*. See descriptions of attributes for more details.

Notes

The type of the returned value depends on the type of the attribute.

Examples

```
get btree(getattribute, tree, unique)
      -- returns true or false

get btree(getattribute, tree, compare)
      -- might return "exact", "ignorecase", etc.
```

Height

Syntax

`integer btree(Height, tree[, root_key])`

Description

Returns the height of the *tree*. If the *root_key* parameter is given then returns the height of the tree whose root is the node with the given key.

Examples

```
get btree(height, tree)

get btree(height, tree, some_key)
```

Inorder

Syntax

`string btree(Inorder, tree, [type, [separator, [root_key]]])`

Description

Traverses the *tree* in in-order and returns the keys of the nodes encountered. The *type* parameter specifies the type of the items to collect, and may be any one of "Keys", "Data", or "Indexes"; the default is "Keys". Each output item is by default separated with a comma, unless the *separator* character is given, in which case output is separated with the separator. The *root_key* specifies the key of the node from which to start traversing the tree; by default traversal starts at the root of the tree.

Notes

In-order traversal of the tree's keys results in sorted output.

Examples

```
get btree(inorder, tree)
      -- Traversal starts from the root of the tree. The
      -- keys encountered are returned, and are separated
      -- with commas. For instance, "key1,key2,key3".

get btree(inorder, tree, keys)
      -- This is the same as the previous example.

get btree(inorder, tree, data, ";")
      -- Traversal starts from the root of the tree. The
      -- data fields of the nodes encountered are returned,
      -- and the data are separated with semicolons. For
      -- instance, "datum1;datum2;datum3".

get btree(inorder, tree, indexes, ";")
      -- Traversal starts from the root of the tree. The
      -- indexes of the nodes encountered are returned,
      -- and they are separated with semicolons. For
      -- instance, "1;2;3".

get btree(inorder, tree, keys, ",", start_key)
      -- Traversal starts from the node with start_key.
      -- The keys of the nodes encountered are returned,
      -- and they are separated with commas. For
      -- instance, "key1,key2,key3".
```

Insert

Syntax

`error btree(Insert, tree,` *key*`[,` *datum*`, [`*keys_separator*`[,` *data_separator*`]])`

Description

Inserts a node with the given *key* and optional *datum* into the tree. If the *keys_separator* parameter is given then Insert is called repeatedly for each key and datum pair delimited by the keys separator. If the *data_separator* is also given then the data are delimited by the data separator, while the keys are still delimited by the keys separator. If there are fewer data then keys then the data fields of the nodes belonging to the extra keys are empty.

Notes

If the unique attribute is true and one of the keys already exists then an error code is returned.

Examples

```
get btree(insert, tree, key)
      -- Inserts a node with the given key into the tree.
      -- The datum field of the node is empty.

get btree(insert, tree, key, datum)
      -- Inserts a node with the given key into the tree.
      -- The datum field of the node is set to the given
      -- datum.

get btree(insert, tree, "key1,key2,key3", "datum1,datum2,datum3",
","")
      -- Inserts nodes with keys "key1", "key2", and "key3" into
      -- the tree. The data fields of these nodes are assigned the
      -- corresponding data: "datum1", "datum2", and "datum3".

get btree(insert, tree, "key1:key2:key3", "datum1;datum2;datum3",
":", ";")
      -- Inserts nodes with keys "key1", "key2", and "key3" into
      -- the tree. The data fields of these nodes are assigned the
      -- corresponding data: "datum1", "datum2", and "datum3".
```

Left

Syntax

`string btree(Left,` *tree*`,` *key*`)`

Description

Returns the key of the left child of the node with the given *key*, or empty if the node is a leaf.

Examples

```
get btree(left, tree, key)
```

Levelorder

Syntax

`string btree(Levelorder,` *tree*`[,` *type*`[,` *separator*`[,` *root_key*`]]])`

Description

Same as the lnorder function except that nodes are visited in level-order.

New

Syntax

`error btree(New,` *tree*`[,` *compare*`])`

Description

Creates a new tree with the name given in the *tree* parameter. This function must be called before a tree can be used. The *compare* parameter specifies the rules used for comparing the keys in the tree; see the description of the compare attribute for

more details. When you have completely finished using a tree, you can use the Dispose function to release the memory used by the tree.

Examples

```
get btree(new, tree)

get btree(new, tree, ignorecase)

get btree(new, tree, numeric)
```

Parent

Syntax

```
string btree(Parent, tree, key)
```

Description

Returns the key of the parent of the node with the given *key*, or empty if the node is the root of the *tree*.

Examples

```
get btree(parent, tree, key)
```

Postorder

Syntax

```
error btree(Postorder, tree, [type, [separator, [root_key]]])
```

Description

Same as the lnorder function except that nodes are visited in post-order.

Preorder

Syntax

```
error btree(Preorder, tree, [type, [separator, [root_key]]])
```

Description

Same as the lnorder function except that nodes are visited in pre-order.

Range

Syntax

```
string btree(Range, tree, low_key, high_key[, type[, separator[, root_key]]])
```

Description

Executes an in-order range search for nodes with keys between the given*low_key* and *high_key*, and returns the keys of the nodes encountered.The *type* parameter specifies the type of the items to collect, and may be any one of "Keys", "Data", or "Indexes"; the default is "Keys". Each output item is by default separated with a

comma, unless the *separator* character is given, in which case output is separated with the separator. The *root_key* specifies the key of the node from which to start traversing the tree; by default traversal starts at the root of the tree.

Examples

```
get btree(range, tree, goodbye, hello)
      -- Returns a comma separated list of all of the keys
      -- whose values fall between the strings "goodbye"
      -- and "hello".

get btree(range, tree, first, last, data)
      -- Returns a comma separated list of all of the data
      -- fields of all of the keys whose values fall between
      -- the strings "first" and "last".

get btree(range, tree, 1, 10, keys, ";")
      -- Returns a semicolon separated list of all of the keys
      -- whose values fall between the strings "1"
      -- and "10".

get btree(range, tree, aardvark, zyzzyva, indexes, ";")
      -- Returns a semicolon separated list of all of the
      -- indexes of the keys whose values fall between the
      -- strings "1" and "10".

get btree(range, tree, me, you, keys, ",", root_key)
      -- Starting from the node whose key is root_key,
      -- returns a comma separated list of the keys whose
      -- values fall between the strings "me" and "you".
```

Right

Syntax

`string btree(Right,` *tree, key*`)`

Description

Returns the key of the right child of the node with the given *key*, or empty if the node is a leaf.

Examples

```
get btree(right, tree, key)
```

Root

Syntax

`string btree(Root,` *tree*`)`

Description

Returns the key of the root of node of the *tree*, or empty if the *tree* is empty.

Examples

```
get btree(root, tree)
```

Set

Syntax

`error btree(Set,` *tree*`,` *key*`,` *datum*`[,` *keys_separator*`[,` *data_separator*`]])`

Description

Sets the value of the datum field of the node with the given *key* to the given *datum*. If the *keys_separator* parameter is given then Set is called on all sets of keys and data delimited by the separator. If the *data_separator* is given then the data are delimited by the data separator instead of by the keys separator. Every key in the list must already exist in the tree.

Examples

```
get btree(set, tree, key, datum)
      -- Sets the datum field of the node with the given key.

get btree(set, tree, "key1,key2,key3", "datum1,datum2,datum3",",")
      -- Sets the data field of the node with "key1" to "datum1",
      -- sets the data field of the node with "key2" to "datum2",
      -- and sets the data field of the node with "key3" to
      -- "datum3".

get btree(set, tree, "key1,key2,key3", "datum1;datum2;datum3",",",
";")
      -- Sets the data field of the node with "key1" to "datum1",
      -- sets the data field of the node with "key2" to "datum2",
      -- and sets the data field of the node with "key3" to
      -- "datum3".
```

SetAttribute

Syntax

`error btree(SetAttribute,` *tree*, *attribute, value*`)`

Description

Sets the value of the named *attribute*. See descriptions of attributes for more details.

Notes

It is not possible to change the compare attribute once a tree is created. To change this attribute's value the tree must be completely rebuilt. The following script will accomplish this:

Script 2. **SetCompare**

```
function setCompare tree, value, separator
      -- save keys and data
      put btree(preorder, tree, false, separator) into keys
      if (btree(error)) then return btree(error)
      put btree(preorder, tree, true, separator) into data
      if (btree(error)) then return btree(error)
      -- dispose and then recreate tree with
      -- new compare attribute value
      get btree(dispose, tree)
      if (it ≠ empty) then return it
      get btree(new, tree, value)
      if (it ≠ empty) then return it
      -- insert keys and data
      get btree(insert, tree, keys, data, separator)
      return it
end setCompare
```

Examples

```
get btree(setattribute, tree, unique, false)
      -- Sets the value of the unique attribute to false.
```

Limitations and bugs

This section describes any limitations on the size and number of data that the program may manipulate. Also discussed are any known bugs, with suggested ways to work around them.

Limitations

This section lists various minimum and maximum sizes for trees. All limits may be smaller depending on the availability of memory and other computer resources. It is unlikely BinaryTree will actually encounter an error associated with the exhaustion of available memory since HyperCard is more likely to quit first.

In the following table, the value represented by Integer is 32,767 and the value represented by LongInt is 2,147,483,647.

Table 2. **BinaryTree limits**

Item	Limit
Number of trees	LongInt
Nodes in a tree	LongInt
Length of a key	LongInt
Length of a datum	LongInt

Known bugs

This section is included for updates on possible and real bugs, and for the dissemination of temporary solutions. Pseudo-bugs will also be reported here (a pseudo-bug is defined as “weird behavior deriving from the correct definition of the software”).

- In trees with duplicate keys, only the first node encountered when searching for a key is ever returned. In the future I may support referring to a node using its index, in addition to the current method of referring to a node by key. This is only a psuedo-bug.

Version information

This section describes features that have changed from previous versions. Also discussed are plans for the future of this program.

Changes from earlier versions

There have been no earlier versions.

Future plans

Self organizing tree

I intend to implement a version of a self organizing tree, such as a splay tree. The tree could have an automatic mode, in which the tree reorganizes itself if it becomes very unbalanced and the rest of the time behaves like a simple tree. Another option would be to reorganize the tree after every insertion or deletion.

About the program

«SECTION NOT YET AVAILABLE»

Appendix A. Functions (by operation)

This section lists the functions implemented according to the operations they perform.

Getting information

```
string      btree("!")
string      btree("?")
```

Getting errors

```
error       btree(Error)
```

Creating and disposing of trees

```
error       btree(New, tree)
error       btree(Dispose, tree)
```

Inserting and deleting nodes

```
error       btree(Insert, tree, key[, datum[, keys_separator[, data_separator]]])
error       btree(Delete, tree, key[, separator])
```

Setting and getting data

```
error       btree(Set, tree, key, datum[, keys_separator[, data_separator]])
string      btree(Get, tree, key[, keys_separator[, data_separator]])
```

Accessing nodes

```
string      btree(Left, tree, key)
string      btree(Right, tree, key)
string      btree(Parent, tree, key)
string      btree(Root, tree)
```

Range searching

```
string      btree(Range, tree, low, high[, type[, separator[, root_key]]])
```

Traversing

```
string      btree(Inorder, tree[, type[, separator[, root_key]]])
string      btree(Preorder, tree[, type[, separator[, root_key]]])
string      btree(Postorder, tree[, type[, separator[, root_key]]])
string      btree(Levelorder, tree[, type[, separator[, root_key]]])
```

Getting height of trees

```
integer     btree(Height, tree[, root_key])
```

Setting and getting attributes

```
error       btree(SetAttribute, tree, attribute, value)
string      btree(GetAttribute, tree, attribute)
```

Appendix B. Function quick reference

The following table is an alphabetic list of all of the functions implemented, the types of data they return, and their syntax.

Table 3. **Function quick reference**

Returns	Syntax
string	`btree("!")`
string	`btree("?")`
error	`btree(Delete, tree, key[, separator])`
error	`btree(Dispose, tree)`
error	`btree(Error)`
string	`btree(Get, tree, key[, keys_separator[, data_separator]])`
string	`btree(GetAttribute, tree, attribute)`
integer	`btree(Height, tree[, root_key])`
string	`btree(Inorder, tree[, type[, separator[, root_key]]])`
error	`btree(Insert, tree, key[, datum[, keys_separator[, data_separator]]])`
string	`btree(Left, tree, key)`
string	`btree(Levelorder, tree[, type[, separator[, root_key]]])`
error	`btree(New, tree)`
string	`btree(Parent, tree, key)`
string	`btree(Postorder, tree[, type[, separator[, root_key]]])`
string	`btree(Preorder, tree[, type[, separator[, root_key]]])`
string	`btree(Range, tree, low, high[, type[, separator[, root_key]]])`
string	`btree(Right, tree, key)`
string	`btree(Root, tree)`
error	`btree(Set, tree, key, datum[, keys_separator[, data_separator]])`
error	`btree(SetAttribute, tree, attribute, value)`

Appendix C. Resources used

This appendix gives a complete list of resources needed by this program. These resources must be installed in your stack for the program to work (see the section on installation in the common manual).

The following table lists the resources with their default IDs and names, along with a short description of the data contained in each resource and how the resource is used by the program. The resource of type TABL is described in the common manual.

Table 4. **Resources**

Type	Name	Description
XFCN	btree	The XFCN whose purpose it is to load, lock, and call the PROC resource.
PROC	BinaryTree	The resource containing the executable code.
STR#	BinaryTree:ResourceMap	Map of resources used by BinaryTree.
STR#	BinaryTree:Info	Version and usage information.
TABL	BinaryTree:Functions	Names of the functions.
TABL	BinaryTree:Attributes	Names of the attributes.
TABL	BinaryTree:Compares	Names of the comparison attributes.
TABL	BinaryTree:Outputs	Names of output types passed to tree traversal functions (eg, "keys", "indexes", "data").

Appendix D. Revision history

This section is to be used for recording any changes made to this manual. This is necessary since I do not want inconsistencies or mistakes introduced by others to reflect on my reputation, and, if the revisions improve this product, then the person who made the improvements should receive full credit. For consistency, please enter dates as Year-Month-Day.

Table 5. **Revision history**

Date	Name	Comments
90-07-18	Ari Halberstadt	This is an example entry
90-07-11	Ari Halberstadt	Version 0.9