Getting Started with Shore[1]

The Shore Project Group
Computer Sciences Department
UW-Madison
Madison, WI
Version 1.1.1

October 27, 1997

[1]This research is sponsored by the Advanced Research Project Agency, ARPA order number 018 (formerly 8230), monitored by the U.S. Army Research Laboratory under contract DAAB07-92-C-Q508.

Contents

1 Introduction

This tutorial explains, through the use of detailed examples, how to write application programs in C++ that store and manipulate complex data structures in Shore.

1.1 What this Tutorial Is

This tutorial illustrates many aspects of Shore, including

- how to use the SDL data-definition language to define data types of persistent data,
- how to use the SDL compiler to translate these definitions into C++ class definitions,
- how to implement the member functions of these classes in C++,
- how to write a driver program that creates, looks up, and destroys objects,
- how to compile, build, and run the application program,
- how to perform an "inventory" of persistent objects that have been created, and
- how to combine Shore application programs with "legacy" Unix programs that access Shore objects as if they were Unix files.

1.2 What this Tutorial Is Not

This tutorial does not try to do everything. In particular:

- It is not a general introduction to Shore, its goals, structure or status. You should read An Overview of Shore before reading any further.
- It is not a reference manual. Reference manuals exist or are being written for all aspects of Shore. See The Shore Release for an index to the rest of the documentation.
- It does not even attempt to demonstrate all the features of Shore, just a subset sufficient to get you started.
- Most importantly, *it is not a tutorial on how to write high-quality, efficient applications.* It uses a rather contrived example designed to show off several features, but it does not claim the algorithms presented are a good way to manipulate persistent data.

1.3 What the Example Does

The example program `stree` uses an unbalanced binary search tree as an inverted index to a set of documents. The tree contains one node for each distinct word appearing in any of the documents. Associated with each word is a set of citations, each of which indicates a document and the offset within the document of the start of a line containing the word. Each tree node, citation, and document is a separate object stored in the Shore database. Although the documents are actually stored in the Shore database, they are stored in such a way that existing Unix programs can manipulate them as if they were ordinary files.

The program has options to add and remove documents from the database and to list the lines containing a given word. It also has a debugging option that dumps all the objects in the database. This option illustrates how to write a maintenance program that iterates through the objects in the database in a "raw" form.

The tutorial then shows how to modify the example to use the *index* facility of Shore to accomplish the same task in a different way.

1.4 What the Examples Demonstrate

The example programs illustrate how to define objects with methods ("member functions" in C++) linked together with pointers (called "references" in Shore). They show how to use *relationships*, which generalize pointers, adding the ability to represent "1-to-N" and "M-to-N" associations and automatically maintain inverse "pointers". They also illustrate the Unix-compatibility features of Shore.

1.5 What the Examples are *Not*

First and foremost, the binary search tree program does not illustrate the best—or even a good—way to build inverted indices. The second example, which uses Shore's built-in index feature, is closer to the way a real application would accomplish this task. A binary search tree is not a very good data structure for disk-based data structures, since fetching each node requires a disk access. In fact, even this program was designed to run in main memory, a hash table would be better! The example was chosen to illustrate how a program that manipulates linked data structures can easily be adapted to make those structures persistent, but converting a main-memory program to run *efficiently* with persistent data generally requires a careful re-design of data structures and algorithms.

The examples do not illustrate *all* the features of Shore or the SDL data-definition language. They do not use all the available pre-defined data structures (such sequences or bags), the "module" facility for managing large complex designs, or inheritance (SDL supports multiple inheritance). See The SDL Reference Manual) for more details.

1.6 Reading the Example Program

This tutorial walks through the sources of the search-tree program in detail. These sources, as well as associated test programs and data, may be found in the `src/examples/stree` sub-directory of the distribution. They are also included in (Section 7). The next few sections walk through this example in detail. It will be useful to keep a copy of the program sources close to hand. Throughout this tutorial, we will assume (as does the Shore Software Installation Manual), that the environment variable `$SHROOT` contains the absolute path name of the root directory of the installed Shore software.

2 Defining Data Types

The first step in building an application to run under Shore is to define the data types of all persistent data that will be used by the application. These declarations are written in a

type-definition language called SDL (for Shore Data-definition Language). The file `stree.sdl` (Section 7.1) contains the type definitions for the binary search tree example. This file contains one module, called `stree`, which defines four types: `SearchTree`, `Word`, `Document`, and `Cite`. These types are defined by *interface* definitions, which are quite similar to class definitions in C++.

`SearchTree` is the top-level object. The example program will create exactly one object of this type. Its public interface consists of three operations: `initialize`, `insert`, and `find`. It also defines one private operation, an alternate (overloaded) version of `insert`, and one attribute, a `ref` (persistent pointer) to the `Word` object that is the root of the tree. The comments in `stree.sdl` (Section 7.1) explain the semantics of the operations and attributes; we will remark here only on aspects of the definition that illustrate features of SDL.

The `initialize` operation of a `SearchTree` should be invoked once immediately after it is created. SDL does not yet have the counterpart of constructors and destructors in C++. (This deficiency may be rectified in a future version.) The `insert` operation is intended to be invoked with one argument, which is a (transient) C++ character string. We use type type `lref<char>`, which translates to `char *` in the C++ binding. This is a bit of a kludge. Future versions of SDL will probably have a more general mechanism for declaring operations whose parameters have "opaque" (i.e., non-SDL) types.

`Word` represents a node in the search tree. The private part of the interface is similar to the way one might define a binary search tree node in C++ or C. A `Word` has a string value and pointers to its left and right subtrees. The type `string` is a pre-defined type in SDL representing an arbitrary-length character string. The `relationship` declaration indicates that `Word`s participate in an N–M (many-to-many) relationship with `Cite`s. That is, for each instance `w` of `Word`, `w.cited_by` is a set of zero or more references to `Cite` objects. Moreover, `Word::cited_by` and `Cite::cites` are to be kept *consistent*: `w.cited_by` should contain a reference to an instance `c` of `Cite` if and only if `c.cites` contains a reference to `w`. The operations `find_or_add` and `find` are intended only to be called from the operations of `SearchTree`. They would be in the private part of the interface if SDL had the equivalent of the *friend* declaration of C++.

A `Cite` object represents a citation (an occurrence of a word in a line of a document). The `offset` attribute indicates the offset of the cited line from the beginning of the document, in bytes. If SDL supported first class relationships in the sense of the entity-relationship model—that is, if relationships could have attributes—`Cite` would have been declared as a relationship between `Word` and `Document` with attribute `offset`. Instead, a *Cite* is represented by a separate object, with a many-to-many relationship to `Word` (each word may occur on many lines and each line may cite several words) and a many-to-one relationship to `Documment` (a document may have many lines, but each line cites a unique document).

Finally, `Document` represents an actual document stored in the repository. The type `text` of the attribute `body` is the same as `string`, but has the additional function of declaring that when a `Document` is accessed through the Unix compatibility interface, it will appear to be a file whose contents are the contents of this field.

3 Implementing the Operations

The second step in building an application is to write the code that implements the operations of its interfaces. Currently, this code must be written in C++. The Shore project intends to support other implementation languages in future releases. The implementation code for the search tree example is contained in four files, one for each interface: `tree.C` (Section 7.3), `word.C` (Section 7.4), `cite.C` (Section 7.5), and `document.C` (Section 7.6).

On the whole, this code is similar to the code one would write to implement the C++ classes generated from the interface definitions. However, there are several points to note:

- Each of these files `#includes` the header file `stree.h`, which is generated from the interface definitions. This file, in turn, `#includes` header files from the Shore library that define a variety of types and macros mentioned below.
- Types defined as `ref<T>` in SDL are written as `Ref<T>` in C++. Similar remarks apply to `set`, `bag`, etc.
- Operations declared `const` in SDL become `const` member functions in the C++ binding.
- Values of type `Ref<T>` can be used as if they were of type `const T*`. For example, consider the body of the function `SearchTree::find`. The attribute `root` of interface `SearchTree`, which is declared to have type `ref<Word>`, is translated to a data member of type `Ref<Word>` of class `SearchTree`. Thus the test "`if (root)`" checks whether the `root` pointer is null, and the call "`root->find(str)`" invokes the const member function `Word::find`.
- The special member function `Ref<T>::update` converts a value of type `Ref<T>` into a pointer of type `T*`. In other words, `update` explicitly performs the conversion from `Ref` to pointer described in the preceding paragraph, but without the `const` qualification. For example, consider the body of the function `SearchTree::insert`. Since the automatic conversion described in the preceding paragraph can only convert a `Ref<Word>` to a `const Word *`, the compiler would reject `root->find_or_add(s)` because `Word::find_or_add` is a non-const member function of `Word`. Thus we write `root.update()->find_or_add(s)` to explicitly convert `root` from `Ref<Word>` to `Word *`. The `update` function has the runtime effect of locking the referenced object to prevent interference from other users and marking it as (potentially) modified, so that it will be written back to persistent storage at the end of a transaction.

4 The Main Program

The main program of our sample application is in `main.C` (Section 7.2). Most of the code should be clear to any experienced C++ programmer. We will only concentrate on those statements that exercise Shore features.

4.1 Initialization

Any program that interacts with Shore must call the static member function `Shore::init` exactly once before doing any Shore operations, to initialize the client-side machinery. It

searches the command line (supplied by the first two arguments, which are usually the same as the first two arguments to `main`) for options specifically meaningful Shore and removes them from `argc` and `argv`. The forth argument to `Shore::init` is the name of an options file, which can supply parameters, such as the size of the object cache. It is a good idea to get the name of this file from the environment, as indicated here, rather than wiring into the program. If there is no value for `STREE_RC` specified in the environment, the standard Unix library function `getenv` will return 0, and a null fourth argument tells `Shore::init` to use reasonable defaults. The third argument to `Shore::init` is the application name, which is used to look up options in the option file. A null argument (as shown here) tells `Shore::init` to use `argv[0]`. For more details, consult the init(oc) manual page.

Like many Shore interface functions, `Shore::init` returns a value of type `shrc` ("rc" stands for "return code"). The macro `SH_DO` is handy for calling functions that are not expected to fail. It evaluates its argument and verifies that the result is `RCOK`. If not, it prints (on `cerr`) an error message and aborts the program. For more details about errors, consult the errors(oc) manual page. `SH_DO` is described on the transaction(oc) manual page.

4.2 Transactions

Every Shore operation except `Shore::init` must be executed inside a transaction. A transaction groups a set of interactions with the database into a single atomic unit. Shore ensures that transactions running concurrently by multiple programs have a net effect that is equivalent to running them one at a time. (This property is called "serializability"). Moreover, if a transaction should fail, Shore guarantees that all changes to the database performed by the transaction are undone. A program starts a transaction by invoking the macro `SH_BEGIN_TRANSACTION`. Its argument is a variable of type `shrc`. When the program has successfully completed all the actions in a transaction, it invokes the parameterless macro `SH_COMMIT_TRANSACTION` to make all of its changes to the database permanent and to unlock any database objects that Shore may have locked to ensure serializability. In exceptional circumstances, Shore may reject the attempt to commit the transaction. Therefore, `SH_COMMIT_TRANSACTION` returns an `shrc` value. Since we do not want to try any fancy recovery actions if `SH_COMMIT_TRANSACTION` fails in our application, we invoke it with `SH_DO`.

On occasion, an application program may discover that a transaction cannot be completed for application-specific reasons. In such occasions, the program can explicitly *abort* the transaction by calling the macro `SH_ABORT_TRANSACTION(rc)`, where `rc` is a value of type `shrc`. In addition to requesting Shore to undo all changes to persistent data and release all locks, this macro performs a longjmp, returning control to the statement following the most recently executed `SH_BEGIN_TRANSACTION` and assigning the `shrc` value supplied to `SH_ABORT_TRANSACTION` to the result parameter of `SH_BEGIN_TRANSACTION`. Since any transaction may be aborted in this manner, each call to `SH_BEGIN_TRANSACTION` should be followed by code that tests the resulting `shrc` and takes corrective action if it is not `RCOK`. The member function `shrc::fatal` prints an appropriate message and aborts the program. The macro `SH_DO` previously described behaves somewhat differently if a transaction is active and an error is detected: Instead of terminating the program, it invokes `SH_ABORT_TRANSACTION`. For more details about errors, consult the transaction(oc) manual page.

4.3 Registered Objects

After beginning a transaction, our example program calls `Shore::chdir`. to go to directory `stree`. `Shore::chdir` is similar to the `chdir` system call of Unix: It alters the current Shore working directory. Note that the program has two "current working directories": one that is applies to Unix system calls and one that applies to all path names used in calls to Shore.

Like the Unix system call, `Shore::chdir` will fail if the the directory does not exist. In the case of our example program, `Shore::chdir` fails for this reason the first time the program is run. It recovers by creating the directory (using `Shore::mkdir`, which is similar to the Unix function of that name), and reissues the `chdir` request). A failure of the first `chdir` operation for any other reason is a catastrophic error. Thus the program checks that the return code is either `RCOK` or `SH_NotFound` and aborts the transaction otherwise.

When run for the first time, our example program also creates two "registered" objects: an instance of `SearchTree` registered under the path name `stree/repository` and a pool named `stree/pool`. The pool is used later to allocate "anonymous" instances of `Word` and `Cite`. See An Overview of Shore for more information about registered and anonymous objects and pools. The `SearchTree` object is created by a form of the C++ *new* operator applied to the class name `SearchTree` using C++ "placement syntax" to supply the path name and permission bits for the new object. If the operation should fail for any reason (such as permission denied), it will cause an abort of the current transaction, returning control to the statement following the most recent `SH_BEGIN_TRANSACTION`. Otherwise, a reference to the new object is assigned to the global variable `repository`.

The creation of the pool `nodes`. illustrates an alternative way of creating a registered object. The variable `nodes` is declared to have type `Ref<Pool>`, where `Pool` is a pre-defined Shore type. The class `Ref<T>`, for any `T`, has several static member functions, such as `create_registered`, `create_anonymous`, and `create_pool`. Each one has parameters to supply a path name and protection mode, as well as a result parameter to receive a reference to the created object. In this case we call `nodes.create_pool` to create a new `Pool` object. (We could have written equivalently `Ref<Pool>::create_pool`). Each of these functions returns an `shrc` result to indicate success or failure.

The differing failure modes of these two ways of creating registered objects illustrate a general design principle of Shore. Many Shore operations are invoked implicitly. Another example of an implicit operation is dereferencing a `Ref<T>` value. When any of them fails (for example, if the reference is null or dangling), Shore responds by aborting the current transaction. If the program needs more precise control—in particular, if it wants a chance to recover from the error—it must use an alternative interface by explicitly calling a Shore function that yields a return code.

If the directory `stree` already exists, the program expects to find existing registered objects `stree/repository` and `stree/pool`. Each reference class `Ref<T>` has a static member function `lookup`, with a path-name input parameter and an output parameter of type `Ref<T>`. This function looks for a registered object with the given name, and if one is found, checks that its type (as indicated by data stored in the database) matches `T`. If both checks succeed, a reference to the object is returned in the result parameter. The initializations of `repository` and `nodes` illustrate two ways of invoking this function.

Finally, the main program performs one of four operations depending on a command-line switch. It either adds one or more documents to the repository, looks up a word, removes a

document from the repository, or dumps all anonymous objects.

4.4 Anonymous Objects

In a typical Shore application, the vast majority of objects created will not have path names. Unlike registered objects, which can be accessed either by path name or by references from other objects, these "anonymous" objects can only be accessed by following references. To assist in clustering, and to allow the application (and system administrators) to keep track of all allocated space, Shore requires each anonymous object to be allocated from a *pool*, which is a registered object. Our example program uses just one pool for all anonymous objects. A more sophisticated program might use a separate pool for each type extent, or for each major component of a complex data structure.

The function `SearchTree::insert(char *fname)` in `tree.C` (Section 7.3) shows how to create anonymous objects. The expression "`new (nodes) Cite`" allocates a new instance of interface `Cite` from the pool referenced by `nodes`. The function `Document::finalize(char *fname)` in `document.C` (Section 7.6) shows how to destroy an anonymous object: If `p` is a reference (an instance of `Ref<T>`, for some type `T`), `p.destroy()` destroys the object referenced by `p`. Registered objects cannot be explicitly destroyed; like Unix files, they are deleted by the system when they have no path names designating them. An example of code to delete a registered object may be found in the function `delete_file` in `main.C` (Section 7.2) The call `Shore::unlink(fname)` removes the name `fname` from a registered object. Since this object will not have any aliases ("hard" links, which are created by `Shore::link`), unlinking it will cause it to be destroyed.

4.5 Relationships

The definitions in `stree.sdl` include two bidirectional relationships. One links words to their citations and the other links citations to the documents they cite. A bidirectional relationship has two names, one for each direction. For example, the relationship between citations and documents is called "`doc`" in the `Cite`-to-`Document` direction and "`cited_by`" in the reverse direction. This relationship is declared by the declaration

```
relationship ref<Document> doc inverse cited_by;
```

in interface `Cite`, and by the declaration

```
relationship set<Cite> cited_by inverse doc;
```

in interface `Document`. The SDL compiler checks that the two declarations are consistent. The use of "`ref`" rather than "`set`" in the first of these declarations indicates a functional dependency from `Cite` to `Document`: Each `Cite` is related to at most one `Document`.

In the C++ binding, these declarations give rise to data members `Cite::doc`, of type `Ref<Document>` and `Document::cited_by`, of type `Set<Cite>`. Similarly, the relationship between words and citations is represented by `Word::cited_by` and `Cite::cites`.

The type `Set<T>` represents a set of zero or more references to distinct `T` objects. It has member functions to add and delete values of type `Ref<Cite>` and to iterate through its contents. The details of the interface, which are documented in the set(cxxlb) manual page,

are likely to change in future releases of Shore. An example of the use of the current interface may be seen in `word.C` (Section 7.4). In `Word::occurs_on`, a citation of word `w` is recorded by adding the reference `cite` to `w.cited_by`. The runtime support automatically adds a reference to `w` to `cite->cites`. The function `Word::occurrence` uses the member function `Set<Cite>::get_elt` to retrieve (a reference to) one of the citations of a word, while `Word::count` uses `Set<Cite>::get_size` to determine how many citations there are. A reference can be deleted from a set with `Set<T>::del`. `Document::finalize` uses an alternative interface: The function `Set<T>::delete_one` returns a one of the references, deleting it from the set. (The implementation chooses an arbitrary reference to return; it retuns `NULL` if the set is empty).

The function `Document::finalize` is called just before destroying a `Document` object. Although the runtime system automatically updates one end of an bidirectional relationship when the other end is updated by assignment, it does not (yet) update inverse relationships properly when an object is destroyed. (This is a bug; it will be fixed in a future release). However, even if it did so, there might be application-specific cleanup operations required. In our example program, we would like to "garbage collect" the `Cite` objects associated with the document being removed. `Document::finalize` iterates through the citations of the document, invoking `Cite::finalize` on each one and then destroying it. `Cite::finalize` simply removes all references from the citation to `Word` objects, thereby removing the citation from the `cited_by` set of each word. The example program does not remove words from the binary search tree when their citation counts drop to zero. Adding code to do so would not be hard, but it would not illustrate any additional features of Shore.

4.6 Strings and Text

The pre-defined type `string` is implemented as a `char *` pointer and a length (so strings can contain null bytes). When a persistent object containing strings is written to disk, the actual string data is appended to the object and the pointers are converted to a form appropriate for storage on disk. When it is brought back into memory, the pointers are restored ("swizzled") to memory addresses. When an ordinary C++ (null-terminated) string is assigned to a Shore string, the bytes (up to and including the terminating null byte) are copied to dynamically allocated space. See for example, `Word::initialize`. When an object containing strings is removed from the object cache, its string space is freed. Thus Shore strings have value semantics.

Standard library string functions such as `strcmp`, `strncmp`, `strlen`, etc., as well as `memcpy` and `bcopy` are overloaded to work with Shore strings. In addition to `strlen`, strings support an operation `blen` which returns the total length (including null bytes). It is also possible to assign a character or string to an arbitrary offset in a Shore string. The target string is expanded if necessary to accommodate the data. For example, `Document::append` extends the `body` field of a document by invoking the `sdl_string::set` function (`Document::body` is actually of type `text`, but `text` and `string` are the same for the purposes of this discussion). See the string(cxxlb) manual page for more details.

4.7 Scanning Pools

The example program supports an option (–p) for dumping all the anonymous objects in the `pool` created by the program. This last option is useful for verifying that the object deletion code is working correctly, and illustrates how one might write administrative programs for maintaining a complex database. `pool_list` in `main.C` (Section 7.2) creates a `PoolScan` object to scan the contents of the pool, and tests whether the creation was successful. (If for example, the named pool did not exist or permission was denied, the `scan` object would be created in an "invalid" state, and would test as *false* when converted to Boolean.) The function `PoolScan::next` returns a reference to the "next" object in the pool (according to some arbitrary ordering) in its result parameter. It returns some `shrc` value other than `RCOK` when no more objects remain. The result parameter must be of type `Ref<any>`, the persistent analogue of `void *` (a reference to an object of unknown type). The actual type of object can be tested dynamically with the function `TYPE(T)::isa(Ref<any> &ref)`. Each interface `T` defined in an SDL definition gives rise to a *type object* (or *meta-type*), which is available as a global variable named `TYPE_OBJECT(T)` (of type `TYPE(T)`). One of the member functions of this object is `isa`, which accepts a parameter of type `Ref<any>`, tests whether it is a reference to an object of type `T`, and if so returns a reference of type `Ref<T>` to it. Otherwise, `isa` returns a null reference. It should be noted that this interface for dynamic type checking is provisional; it may be replaced with a facility more nearly resembling the `dynamic_cast` syntax for run-time type identification (RTTI) recently added to the proposed C++ standard.

After checking that the type of returned object conforms to one of the expected types (the program only creates anonymous objects of type `Word` and `Cite`), `pool_list` uses the reference (as converted by `isa`) to call the appropriate print function (`Word::print` or `Cite::print`).

5 Building and Running the Example Program

5.1 Starting the Shore Server

To build the example program, you must have a copy of the Shore server running. The document Shore Software Installation Manual, particularly the section *Testing Your Installation*, gives simple instructions on how to start a server. You probably want to do this in a separate window. The server will accept interactive commands from the keyboard. The only one you will need for this demonstration is "bye", which causes the server to shut down cleanly and exit. The server may also occasionally produce debugging output.

5.2 Building the Application

The Shore documentation release contains the source code for the examples in this tutorial. Assuming you have fetched and unpacked the documentation release as described in the *Shore Software Installation Manual*, you will have a directory `$SHROOT/examples`, where `$SHROOT` is the root directory of the documentation relase.

```
mkdir stree
cp -R $SHROOT/examples/stree/* stree
cd stree
```

```
make stree
```

You should see something like this.

```
rm -f stree.h
/usr/local/shore/bin/sdl -f -s stree.sdl -B -L -o stree.h
g++ -g -I/usr/local/shore/include -c main.C
g++ -g -I/usr/local/shore/include -c tree.C
g++ -g -I/usr/local/shore/include -c word.C
g++ -g -I/usr/local/shore/include -c cite.C
g++ -g -I/usr/local/shore/include -c document.C
g++ -g -I/usr/local/shore/include -c stree_defs.C
g++ -g -I/usr/local/shore/include -o stree main.o stree_defs.o
    tree.o word.o cite.o document.o
    /usr/local/shore/lib/libshore.a -lnsl
```

The second line invokes the SDL compiler. The command-line options ask it to perform several functions. The option `-s stree.sdl` asks it to parse the specification and install the resulting compiled versions of the module `stree` in the Shore database (you must have a Shore server running when you do this). The module is a registered object named `/type/stree`. Other options can be used to put it elsewhere in the Shore database. See the sdl(sdl) manual page for more details. The `-L` option asks SDL to link all the modules together (in this case, there is only one module), the `-B` option tells it to generate a C++ language binding from the module just generated, and the `-o stree.h` option directs it to place the results into the file file `stree.h` in the current directory. This file is included by all of our source files. We then compile all of the source files and link them together, along with the Shore runtime support library. Any C++ compiler should be usable, but the current release is only tested to work with the GNU compiler (g++) listed in the *Requirements* section of the *Shore Release 1.1.1* manual.

We have already explained the source files `main.C` (Section 7.2), `tree.C` (Section 7.3), `word.C` (Section 7.4), `cite.C` (Section 7.5), and `document.C` (Section 7.6). The first of these is the main program, while the rest define the member functions for each of the classes corresponding to interfaces defined in `stree.sdl`. The file `stree_defs.C` (Section 7.7) is a small file containing just two lines:

```
#define MODULE_CODE
#include "stree.h"
```

The generated file `stree.h` contains some function definitions and initializations of global variables. Compiling it with `MODULE_CODE` defined generates these functions and initializations for linking with the rest of the program.

5.3 Running Some Examples

5.3.1 A Small Example

First use `stree` to add the files `test1`, `test2`, and `test3` to the repository.

```
% stree -aV test?
```

The output should look like this

```
Indexing file test1
Indexing file test2
Indexing file test3
about to commit
committed
```

Next, use the `-lV` (list verbose) option to look up some words.

```
% stree -lV six
========== six
test2: two six
test2: three six
test2: six two
test2: six three
test2: six six
test2: six seven
test2: seven six
test3: four six
test3: five six
test3: six four
test3: six five
test3: six six
test3: six seven
test3: seven six
**** 14 citations

% stree -lV eight
========== eight
**** Not found
```

Use the `-d` option to remove some of the documents.

```
% stree -d test2

% stree -lV six
========== six
test3: four six
test3: five six
test3: six four
test3: six five
test3: six six
test3: six seven
test3: seven six
**** 7 citations

% stree -d test1
```

```
% stree -lV seven
========== seven
test3: four seven
test3: five seven
test3: six seven
test3: seven four
test3: seven five
test3: seven six
test3: seven seven
**** 7 citations
```

Use the `-p` option to see what anonymous objects remain in the pool.

```
% stree -p
Word 'one' occurs on 0 lines
Word 'three' occurs on 0 lines
Word 'five' occurs on 7 lines
Word 'seven' occurs on 7 lines
Word 'two' occurs on 0 lines
Word 'six' occurs on 7 lines
Cite, offset 0 in file test3 cites four
Word 'four' occurs on 7 lines
Cite, offset 10 in file test3 cites four five
Cite, offset 20 in file test3 cites four six
Cite, offset 29 in file test3 cites four seven
Cite, offset 40 in file test3 cites five four
Cite, offset 50 in file test3 cites five
Cite, offset 60 in file test3 cites five six
Cite, offset 69 in file test3 cites five seven
Cite, offset 80 in file test3 cites six four
Cite, offset 89 in file test3 cites six five
Cite, offset 98 in file test3 cites six
Cite, offset 106 in file test3 cites six seven
Cite, offset 116 in file test3 cites seven four
Cite, offset 127 in file test3 cites seven five
Cite, offset 138 in file test3 cites seven six
Cite, offset 148 in file test3 cites seven
```

Remove the remaining document from the repository and verify that the pool contains only `Word` objects.

```
% stree -d test3
```

```
% stree -p
Word 'one' occurs on 0 lines
Word 'three' occurs on 0 lines
```

```
Word 'five' occurs on 0 lines
Word 'seven' occurs on 0 lines
Word 'two' occurs on 0 lines
Word 'six' occurs on 0 lines
Word 'four' occurs on 0 lines
```

5.3.2 A Larger Example

The `stree` directory has a sub-directory called `sonnets` which contains all 154 of Shakespeare's sonnets, one per file. For this test, add sonnets 10 through 19 to the repository.

```
% stree -aV sonnets/sonnet01?
Indexing file sonnets/sonnet010
Indexing file sonnets/sonnet011
Indexing file sonnets/sonnet012
Indexing file sonnets/sonnet013
Indexing file sonnets/sonnet014
Indexing file sonnets/sonnet015
Indexing file sonnets/sonnet016
Indexing file sonnets/sonnet017
Indexing file sonnets/sonnet018
Indexing file sonnets/sonnet019
about to commit
committed

% stree -lV summers
========== summers
sonnet012:   And summer's green all girded up in sheaves
sonnet018:   Shall I compare thee to a summer's day?
sonnet018:   And summer's lease hath all too short a date:

% stree -l summers
sonnet012
sonnet018
sonnet018
```

Note that sonnet 18 is listed twice, since "summers" appears on two different lines in that sonnet.

To illustrate access to the Shore database from existing Unix utilities, mount the Shore database as a Unix file system, as explained in the Shore Software Installation Manual, in the section *NFS-Mounting the Shore File System. If you follow the instructions there, you will have the Shore database mounted as /shoremnt.*

```
% ls -l /shoremnt
total 2
drwxr-xr-x   1 solomon  solomon        12 Aug  6 16:33 stree/
prwxr-xr-x   1 solomon  solomon         0 Aug  6 15:50 testpool|
```

```
drwxr-xr-x   1 solomon  solomon        12 Aug  6 16:23 types/
% ls -l /shoremnt/stree
total 6
prw-r--r--   1 solomon  solomon         0 Aug  6 16:33 pool|
-rw-r--r--   1 solomon  solomon         0 Aug  6 16:33 repository
-rw-r--r--   1 solomon  solomon       650 Aug  6 16:33 sonnet010
-rw-r--r--   1 solomon  solomon       709 Aug  6 16:33 sonnet011
-rw-r--r--   1 solomon  solomon       657 Aug  6 16:33 sonnet012
-rw-r--r--   1 solomon  solomon       637 Aug  6 16:33 sonnet013
-rw-r--r--   1 solomon  solomon       623 Aug  6 16:33 sonnet014
-rw-r--r--   1 solomon  solomon       647 Aug  6 16:33 sonnet015
-rw-r--r--   1 solomon  solomon       630 Aug  6 16:33 sonnet016
-rw-r--r--   1 solomon  solomon       677 Aug  6 16:33 sonnet017
-rw-r--r--   1 solomon  solomon       656 Aug  6 16:33 sonnet018
-rw-r--r--   1 solomon  solomon       662 Aug  6 16:33 sonnet019
```

Note that there are 12 registered objects in the directory `stree`*: 10 sonnets (objects of class* `Document`*), the object* `repository` *(of class* `SearchTree`*), and the* `pool` *object. The* `pool` *and* `repository` *show up under Unix as having zero size, since neither has a* `text` *member, but each of the sonnets shows up as a file whose contents are the same as its* `text` *member* `body`*.*

```
% cat /shoremnt/stree/sonnet018
  Shall I compare thee to a summer's day?
  Thou art more lovely and more temperate:
  Rough winds do shake the darling buds of May,
  And summer's lease hath all too short a date:
  Sometime too hot the eye of heaven shines,
  And often is his gold complexion dimmed,
  And every fair from fair sometime declines,
  By chance, or nature's changing course untrimmed:
  But thy eternal summer shall not fade,
  Nor lose possession of that fair thou ow'st,
  Nor shall death brag thou wand'rest in his shade,
  When in eternal lines to time thou grow'st,
    So long as men can breathe or eyes can see,
    So long lives this, and this gives life to thee.
```

The shell script `swc` *(Section 7.8) illustrates how Shore applications can be combined with "legacy" Unix programs.*

```
% swc summers
     14     118     657 sonnet012
     14     114     656 sonnet018
     28     232    1313 total
```

This script uses the output of `stree -l` *(piped through* `sort -u` *to remove duplicates) as the list of arguments to a standard Unix utility (in this case* `wc`*) which accesses the objects as if*

they were ordinary Unix files. Note that there is no need to use a special version of `wc` *or even to re-link* `wc` *with a special library.*

A slightly more sophisticated example is afforded by `sedit` *(Section 7.9), which invokes an editor on the set of sonnets containing a given word. If your editor (as indicated by the* `EDITOR` *environment variable) is emacs, you will see this:*

```
% sedit summers
MR Buffer         Size   Mode           File
-- ------         ----   ----           ----
.  sonnet018      656    Text           /shoremnt/stree/sonnet018
   sonnet012      657    Text           /shoremnt/stree/sonnet012
   *scratch*      0      Lisp Interaction
*  *Buffer List*  274    Text
```

```
--%%-Emacs: *Buffer List*   6:54am 0.23   (Buffer Menu)--All--------------------
  Shall I compare thee to a summer's day?
  Thou art more lovely and more temperate:
  Rough winds do shake the darling buds of May,
  And summer's lease hath all too short a date:
  Sometime too hot the eye of heaven shines,
  And often is his gold complexion dimmed,
  And every fair from fair sometime declines,
  By chance, or nature's changing course untrimmed:
  But thy eternal summer shall not fade,
  Nor lose possession of that fair thou ow'st,
  Nor shall death brag thou wand'rest in his shade,
-----Emacs: sonnet018      6:54am 0.23   (Text Fill)--Top-----------------------
Commands: d, s, x, u; f, o, 1, 2, m, v; ~, %; q to quit; ? for help.
```

To clean up after this test, you can remove all the documents from the repository and then remove the repository itself with the `-c` *option*

```
% stree -d sonnet010 sonnet011 sonnet012 sonnet013 sonnet014
% stree -d sonnet015 sonnet016 sonnet017 sonnet018 sonnet019
% stree -c
```

or you can simply remove the repository through the Unix compatibility interface

```
% rm -rf /shoremnt/stree
```

6 Using Indexes

As mentioned above, a binary search tree is not really an appropriate data structure for persistent data. Shore has a built-in index feature designed just for tasks such as this. The

`src/examples/stree` *sub-directory of the distribution contains a version of the search-trees example that uses a Shore index instead of a search tree. Instead of* `stree.sdl` *we use* `doc_index.sdl`*. The source file* `stree_defs.C` *is replaced by* `doc_index_defs.C`*,* `tree.C` *is replaced by* `docIndex.C`*, and* `main.C`*,* `word.C`*,* `cite.C`*, and* `document.C` *are replaced by* `ix_main.C`*,* `ix_word.C`*,* `ix_cite.C`*, and* `ix_document.C`*, respectively. Since most of the program is unchanged, we shall only consider the differences here.*

First consider `doc_index.sdl`*. We have removed from interface* `Word` *the attributes* `left` *and* `right` *and operations* `find_or_add` *and* `find` *and added a "destructor" named* `finalize`*. The interface* `SearchTree` *has been renamed* `DocIndex`*. Its attribute*

```
attribute ref<Word> root;
```

has been replaced by

```
attribute index<string,Word> ind;
```

which declares an index *mapping strings to* `Word` *references.*[1] *We have also added the operation* `delete_word`*, which we would have included in* `SearchTree` *if we weren't so lazy.*

Each of the source files `main.C`*,* `word.C`*,* `cite.C`*, and* `document.C` *has been edited to replace*

```
#include "stree.h"
```

with

```
#include "doc_index.h",
```

replace occurrences of type `Ref<SearchTree>` *with* `Ref<DocIndex>`*, etc. The code in* `main.C` *for the –p option has been extended to dump the contents of the index.* `Word.C` *has been edited to remove all mention of the deleted attributes* `left` *and* `right` *and operations* `find_or_add` *and* `find`*, and a new* `finalize` *operation has been added, which simply calls* `DocIndex::delete_word`*.*

In `cite.C`*, the* `finalize` *operation checks each* `Word` *cited by the* `Cite` *being deleted to see if its reference count has dropped to zero. If so, it calls the* `Word`*'s* `finalize` *method and destroys it. Note that the* `Word` *is first removed from* `Cite::cites` *relationship, which also removes the current* `Cite` *from the inverse* `Word::cited_by` *relationship, thus decrementing the* `Word`*'s reference count.*

The only change from `document.C` *to* `ix_document.C` *the name of the* `#include` *file.*

Finally, consider the new source file `docIndex.C` *(Section 7.10). The* `index` *attribute* `ind` *of interface* `DocIndex` *compiles into a data member whose type is derived from the predefined Shore type* `sdl_index_base`*. Each such index object must be initialized before it is used. The "constructor"* `DocIndex::initialize` *calls* `sdl_index_base::init`*, which takes a single argument of type* `IndexKind`*—an enumeration type that includes the values* `BTree` *and* `UniqueBtree`*. (Other values of this enumeration correspond to index types that are not yet supported.) In our case, we choose* `UniqueBtree`*, meaning that we want a B-tree that has at most one value for each key. The member function DocIndex::insert uses the member function* `sdl_index_base::find` *to determine whether a* `Word` *with the given key already exists in the*

[1] (If the second type argument to the `index` template is an interface type, SDL automatically inserts a level of indirection. Had the declaration been `index<string,long>`, we would get an index from strings to integers, not references to integers.

index. If not, one is created and a reference to it is inserted into the index with the member function `sdl_index_base::insert`. *Finally, the* `Word` *object that was found or created is updated to add the given* `Cite` *to its set of occurrences.*

`DocIndex::insert_file` *is identical to* `SearchTree::insert_file`, *and* `DocIndex::find` *simply calls* `sdl_index_base::find`. `DocIndex::delete_word` *is called to remove reference to a* `Word` *from the index when it is no longer referenced. The member function* `sdl_index_base::remove` *has an input parameter of the key-type of the index, and an integer output parameter* `count`. *It removes all index entries whose keys match the first argument and returns a count of the number of entries removed. In our case, this count should always be one.*

`DocIndex::print` *illustrates how to iterate through the entries in an index using the Shore template class* `IndexScanIter`, *which has two type parameter corresponding to the types of the keys and values of the index. The types given must match the types of the language binding, which might differ syntactically from the SDL type declarations. (For example,* `Ref<Word>` *must be used in place of* `Word` *here, similarly SDL* `string` *translates to* `sdl_string` *in a C++ language binding.)*

The iterator's constructor takes an index of the appropriate type (in this case `DocIndex::ind`*) as a parameter. For ordered indices (such as B-trees), it also has two optional arguments to indicate lower and upper bounds. When these values are supplied, the iterator only returns keys in the indicated range. When they are omitted (as in this example) the iterator returns all entries in the index.*

An iterator has a function member `next` *to advance to the next entry, a data member* `eof` *to indicate if any more entries exist, and data members* `cur_key` *and* `cur_val` *representing the key and value of the current entry. As this code illustrates, the* `next` *function has to be called once at the start to "prime the pump", and* `eof` *flag is false after advancing past the last entry. This interface may change in future releases. See the index(cxxlb) manual page for more details.*

The script `run_tests.sh` *included in the source distribution runs all of the tests listed above using both the* `stree` *and* `doc_index` *versions of the program.*

7 Appendix: Program Sources

7.1 stree.sdl

```
module stree {
    // interfaces defined here
    interface SearchTree;   // the top-level construct
    interface Word;         // a binary search tree node -- represents one word
    interface Document;     // a document stored in the repository
    interface Cite;         // a reference to a line in a document



    // A binary search tree of Word objects
    interface SearchTree {
      private:
        attribute ref<Word> root;   // the root of the tree
```

```
    // Update the entry matching WORD to add a citation.
    // Add a new Word if necessary.
    void insert(in lref<char> word, in ref<Cite> cite);

  public:
    // Constructor:  make an empty tree
    void initialize();

    // Insert a new Document into the repository.  The argument is a
    // pathname to be interpreted in the Unix name space as the name of a
    // Unix file containing the raw data.  A new Document object with
    // the same base name is created in the current Shore directory,
    // filled with a copy of the file's context, and indexed by all of
    // its words.
    void insert_file(in lref<char> src);

    // Retrieve the Word object matching the argument.
    // Return NULL if not found.
    ref<Word> find(in lref<char> word) const;
};



// There is one Word object for each distinct word appearing in any
// document in the repository.
interface Word {
private:
    attribute string value;
    attribute ref<Word> left, right;
public:
    relationship set<Cite> cited_by inverse cites;
    // Constructor: empty occurrences list
    void initialize(in lref<char> word);

    // How many occurrences?
    long count() const;

    // Get ith occurrence (returns NULL if not that many)
    ref<Cite> occurrence(in long i) const;

    // The following methods are meant to be used only by SearchTree.

    // Find decendant matching WORD creating one if necessary
    ref<Word> find_or_add(in lref<char> word);
```

```
    // Find only, return NULL on not found
    ref<Word> find(in lref<char> word) const;

    // Debugging dump
    void print(in long verbose) const;

    // Add an occurence
    void occurs_on(in ref<Cite> cite);
};


// A Cite object represents a citation.  There is one Cite object for each
// line of each document in the repository.  There is thus a many-many
// relationship from Cite to Word and a many-one relationship from Cite
// to Document.
interface Cite {
private:
    attribute long offset;
public:
    relationship ref<Document> doc inverse cited_by;
    relationship set<Word> cites inverse cited_by;
    // Constructor
    void initialize(in ref<Document> d, in long o);

    // Print the referenced line
    void print(in long vflag) const;

    // Destructor
    void finalize();
};


// A Document is a chunk of text that looks like a Unix file.
// We also record the file name under which it was created.
// (The need to record the name may go away when Shore adds a way to find
// the pathname of a registered object given a Ref to it.)
interface Document {
private:
    attribute text body;
    attribute string name;
    attribute long cur_len;
public:
    relationship set<Cite> cited_by inverse doc;
    // Constructor:  The body is empty.
    void initialize(in lref<char> base_name, in long len);
```

```
        // Add some text to the end of the body.
        void append(in lref<char> str);

        // Read-only access to the file name.
        lref<char> get_name() const;

        // Current length of text
        long size() const;

        // Print a line starting at OFFSET
        void print_line(in long offset) const;

        // Destructor
        void finalize();
    };

}
```

7.2 main.C

```
/*
 * ShoreConfig.h is needed only by applications
 * that distinguish platforms.  (Stree does not,
 * but we include this for documentation purposes.)
 */
#include <ShoreConfig.h>

#include <iostream.h>
#include <fstream.h>
#include <std.h>
#include "stree.h"

Ref<SearchTree> repository;
Ref<Pool> nodes;        // Place to create new anonymous objects

const char *DEMO_DIR = "stree";

char *argv0;
int verbose;
extern "C" int optind;

enum OPERATION {
    OP_NONE, OP_ADD, OP_LIST, OP_DEL, OP_POOL_LIST, OP_CLEAR
} operation = OP_NONE;
```

```
static void add_files(int argc, char *const*argc);
static void list_files(char *str);
static void delete_files(int argc, char *const*argc);
static void pool_list();
static void clear_all();

void usage() {
    cerr << "usage:" << endl;
    cerr << "\t" << argv0 << " -a[V] fname [fname ...]" << endl;
    cerr << "\t" << argv0 << " -l[V] word" << endl;
    cerr << "\t" << argv0 << " -d fname [fname ...]" << endl;
    cerr << "\t" << argv0 << " -p" << endl;
    cerr << "\t" << argv0 << " -c" << endl;
    cerr << "\t" << "the -V option turns on verbose mode" << endl;
    exit(1);
}

int main(int argc, char *argv[])
{
    argv0 = argv[0];
    shrc rc;



    // initialize connection to server
    SH_DO(Shore::init(argc, argv, 0, getenv("STREE_RC")));

    // get command-line options
    int c;
    while ((c = getopt(argc,argv,"aldpcV")) != EOF) switch(c) {
        case 'a': operation = OP_ADD; break;
        case 'l': operation = OP_LIST; break;
        case 'd': operation = OP_DEL; break;
        case 'p': operation = OP_POOL_LIST; break;
        case 'c': operation = OP_CLEAR; break;
        case 'V': verbose++; break;
        default: usage();
    }

    if (operation == OP_NONE)
        usage();



    // Start a transaction for initialization
    SH_BEGIN_TRANSACTION(rc);
    if (rc)
```

```
    rc.fatal(); // this terminates the program with extreme prejudice

// Check that our demo directory exists
rc = Shore::chdir(DEMO_DIR);
if (rc != RCOK) {
    if (rc != RC(SH_NotFound))
        SH_ABORT_TRANSACTION(rc);

    // Not found.  Must be the first time through.
    // Create the directory
    SH_DO(Shore::mkdir(DEMO_DIR, 0755));
    SH_DO(Shore::chdir(DEMO_DIR));



    // Make a new SearchTree object ...
    repository = new("repository", 0644) SearchTree;
    repository.update()->initialize();

    // ... and a pool for allocating Nodes.
    SH_DO(nodes.create_pool("pool", 0644, nodes));
} else { // not first time

    // Get the repository root from the database ...
    SH_DO(Ref<SearchTree>::lookup("repository",repository));

    // ... and the pool for creating nodes
    SH_DO(nodes.lookup("pool", nodes));
}

SH_DO(SH_COMMIT_TRANSACTION);

switch (operation) {
    case OP_ADD:
        add_files(argc-optind, argv+optind);
        break;
    case OP_LIST:
        if (optind != argc-1)
            usage();
        list_files(argv[optind]);
        break;
    case OP_DEL:
        delete_files(argc-optind, argv+optind);
        break;
    case OP_POOL_LIST:
        pool_list();
```

```
            break;
        case OP_CLEAR:
            clear_all();
            break;
        default: break;
    }

    return 0;
} // main

// Add all the named files to the repository
static void add_files(int argc, char *const*argv) {
    shrc rc;

    SH_BEGIN_TRANSACTION(rc);
    if (rc)
        rc.fatal();
    for (int i=0; i<argc; i++)
        repository.update()->insert_file(argv[i]);
    if (verbose)
        cout << "about to commit" << endl;
    SH_DO(SH_COMMIT_TRANSACTION);
    if (verbose)
        cout << "committed" << endl;
} // add_files

// List all uses of a word
static void list_files(char *str) {
    shrc rc;
    int occurrences=0;

    SH_BEGIN_TRANSACTION(rc);
    if (rc)
        rc.fatal();
    Ref<Word> w = repository->find(str);
    if (verbose)
        cout << "========== " << str << endl;
    if (w && w->count() > 0) {
        Ref<Cite> c;
        for (int i=0; c = w->occurrence(i); i++) {
            occurrences++;
            c->print(verbose);
        }
    } else if (verbose) {
        cout << "**** Not found" << endl;
        occurrences = -1;
```

```
    }
    if(occurrences >= 0 && verbose) {
        cout << "**** " << occurrences << " citation"
            << (char *)(occurrences==1?"":"s") << endl;
    }
    SH_DO(SH_COMMIT_TRANSACTION);
} // list_files


// Removed the named files from the repository
static void delete_files(int argc, char *const*argv) {
    shrc rc;

    SH_BEGIN_TRANSACTION(rc);
    if (rc)
        rc.fatal();

    for (int i=0; i<argc; i++) {
        Ref<Document> d;
        SH_DO(d.lookup(argv[i],d));
        d.update()->finalize();
        SH_DO(Shore::unlink(argv[i]));
    }
    if (verbose)
        cout << "about to commit" << endl;
    SH_DO(SH_COMMIT_TRANSACTION);
    if (verbose)
        cout << "committed" << endl;
} // delete_files


static void pool_list() {
    shrc rc;

    SH_BEGIN_TRANSACTION(rc);
    if (rc)
        rc.fatal();

    Ref<any> ref;
    Ref<Word> w;
    Ref<Cite> c;
    {
        PoolScan scan("pool");
        if (scan != RCOK)
            SH_ABORT_TRANSACTION(scan.rc());
```

```
        while (scan.next(ref, true) == RCOK) {
            if (w = TYPE_OBJECT(Word).isa(ref)) {
                w->print(1);
            }
            else if (c = TYPE_OBJECT(Cite).isa(ref)) {
                c->print(2);
            }
            else cout << " Unknown type of object" << endl;
        }
    }
    SH_DO(SH_COMMIT_TRANSACTION);
} // pool_list



static void clear_all() {
    shrc rc;

    SH_BEGIN_TRANSACTION(rc);
    if (rc)
        rc.fatal();

    rc = Shore::unlink("repository");
    if (rc)
        cout << rc << endl;

    SH_DO(nodes.lookup("pool", nodes));
    SH_DO(nodes.destroy_contents());
    rc = Shore::unlink("pool");
    if (rc)
        cout << rc << endl;

    rc = Shore::chdir("..");
    if (rc)
        cout << rc << endl;

    SH_DO(Shore::rmdir(DEMO_DIR));

    SH_DO(SH_COMMIT_TRANSACTION);
} // clear_all
```

7.3 tree.C

```
// Member functions of the SearchTree class
#include <iostream.h>
```

```
#include <string.h>
#include <ctype.h>
#include "stree.h"
#include <sys/types.h>
#include <sys/stat.h>

extern Ref<Pool> nodes;
static int getword(const char *&p, char *res, int size);

extern int verbose; // defined in main.C

void SearchTree::initialize() {
    root = NULL;
}



void SearchTree::insert(char *s, Ref<Cite> c) {
    Ref<Word> w;
    if (root) {
        w = root.update()->find_or_add(s);
    }
    else {
        root = new(nodes) Word;
        root.update()->initialize(s);
        w = root;
    }
    w.update()->occurs_on(c);
}

void SearchTree::insert_file(char *fname) {
    shrc rc;

    if (verbose)
        cout << "Indexing file " << fname << endl;

    // Open input file
    ifstream in(fname);
    if (!in) {
        perror(fname);
        SH_ABORT_TRANSACTION(rc);
    }
    // do a unix stat to get the total size of the file.
    struct stat in_stat;
    if (stat(fname,&in_stat)) {
        perror(fname);
```

```
        SH_ABORT_TRANSACTION(rc);
    }

    // Create target document

    // Strip leading path from file name;
    char *base_name = strrchr(fname, '/');
    if (base_name)
        base_name++;
    else
        base_name = fname;

    Ref<Document> doc;
    rc = doc.new_persistent(base_name, 0644, doc);
    if (rc) {
        perror(base_name);
        SH_ABORT_TRANSACTION(rc);
    }
    doc.update()->initialize(base_name,in_stat.st_size);

    // for each line of the document ...
    char linebuf[1024];
    while (in.getline(linebuf, sizeof linebuf -1)) {
        long off = doc->size();

        // copy the line to the body of the document
        doc.update()->append(linebuf);
        doc.update()->append("\n");

        // allocate a new Cite object for this line
        Ref<Cite> cite = new (nodes) Cite;
        cite.update()->initialize(doc, off);

        // for each word on the line ...
        char word[100];
        const char *p = linebuf;
        while (getword(p, word, sizeof word)) {
            // link the citation to the word
            insert(word, cite);
        }
    }
}



Ref<Word> SearchTree::find(char *str) const {
```

```
    if (root)
        return root->find(str);
    return NULL;
}

// Copy a word of at most SIZE characters (including terminating null)
// in to the buffer starting at RES.  Start searching at location P.
// Words are delimited by white space.  The result is translated to lower
// case, with all non-letters eliminated.
// P is updated to point to the first character not copied.
// The result is 1 if a word is found, 0 if '\0' is encountered first.
static int getword(const char *&p, char *res, int size) {
    for (;; ) {
        // skip leading white space
        while (isspace(*p))
            p++;

        // check for eoln
        if (*p == 0)
            return 0;

        // gather non-space characters, translating to lower case and
        // ignoring non-alpha characters
        int len;
        for (len = 0; len < size-1 && *p && !isspace(*p); p++) {
            if (isupper(*p))
                res[len++] = tolower(*p);
            else if (islower(*p))
                res[len++] = *p;
        }
        if (len > 0) {
            res[len] = 0;
            return 1;
        }
        // otherwise, word was all digits and punctuation, so try again.
    }
}
```

7.4 word.C

```
// Member functions of the Word class
#include <iostream.h>
#include <string.h>
#include "stree.h"

extern Ref<Pool> nodes;
```

```
void Word::initialize(char *word) {
    value = word;
    left = NULL;
    right = NULL;
}

long Word::count() const {
    return cited_by.get_size();
}


Ref<Cite> Word::occurrence(long i) const {
    return cited_by.get_elt(i);
}

Ref<Word> Word::find_or_add(char *s) {
    int i = strcmp(s,value);
    if (i == 0)
        return this;
    if (i < 0) {
        if (left) return left.update()->find_or_add(s);
        else {
            left = new(nodes) Word;
            left.update()->initialize(s);
            return left;
        }
    }
    else {
        if (right) return right.update()->find_or_add(s);
        else {
            right = new(nodes) Word;
            right.update()->initialize(s);
            return right;
        }
    }
}

Ref<Word> Word::find(char *s) const {
    int i = strcmp(s,value);
    if (i == 0) return this;
    if (i < 0) return left ? left->find(s) : (Ref<Word>)NULL;
    return right ? right->find(s) : (Ref<Word>)NULL;
}
```

```
void Word::occurs_on(Ref<Cite> cite) {
    cited_by.add(cite);
}

void Word::print(long verbose) const {
    if (verbose) {
        int s = cited_by.get_size();
        cout << "Word '" << (char *)value
            << "' occurs on " << s << " line" << (s==1 ? "" : "s") << endl;
    }
    else cout << (char *)value;
}
```

7.5 cite.C

```
// Member functions of the Cite class
#include <iostream.h>
#include "stree.h"

void Cite::initialize(Ref<Document> d, long o) {
    doc = d;
    offset = o;
}

void Cite::print(long v) const {
    switch (v) {
        default:
        case 0: // just the file name
            cout << doc->get_name() << endl;
            break;
        case 1: // the file name and the corresponding line
            cout << doc->get_name() << ": ";
            doc->print_line(offset);
            break;
        case 2: // debugging version
            cout << "Cite, offset " << offset
                << " in file " << doc->get_name()
                << " cites";
            {
                int count = cites.get_size();
                for (int i = 0; i < count; i++) {
                    Ref<Word> w = cites.get_elt(i);
                    cout << " ";
                    w->print(0);
                }
                cout << endl;
```

```
            }
            break;
    }
}

void Cite::finalize() {
    while (cites.delete_one()) {}
}
```

7.6 document.C

```
// Member functions of the Document class
#include <iostream.h>
#include <string.h>
#include "stree.h"

void Document::initialize(char *base_name, long ilen) {
    body = 0;
    name = base_name;
    // set a char at the end of the body to initialze the
    // string space.
    body.set(ilen-1,0);
    // initialize cur_len.
    cur_len = 0;
}



void Document::append(char *str) {
    // body.set(str, body.length(), ::strlen(str));
    int str_size = ::strlen(str);
    body.set(str, cur_len, str_size);
    cur_len += str_size;
}

char *Document::get_name() const {
    return name;
}

long Document::size() const {
    // return body.strlen();
    return cur_len;
}

void Document::print_line(long offset) const {
    char buf[100];
```

```
    body.get(buf, offset, sizeof buf);
    buf[sizeof buf - 1] = 0;
    char *p = strchr(buf, '\n');
    if (p) *++p = 0;
    cout << buf;
}



void Document::finalize() {
    Ref<Cite> p;
    while (p = cited_by.delete_one()) {
        p.update()->finalize();
        SH_DO(p.destroy());
    }
}
```

7.7 stree_defs.C

```
#define MODULE_CODE
#include "stree.h"
```

7.8 swc

```
#!/bin/sh
# ------------------------------------------------------------------ #
# -- Copyright (c) 1994, 1995 Computer Sciences Department,    -- #
# -- University of Wisconsin-Madison, subject to the terms     -- #
# -- and conditions given in the file COPYRIGHT.  All Rights   -- #
# -- Reserved.                                                 -- #
# ------------------------------------------------------------------ #
# $Header: /p/shore/shore_cvs/src/examples/stree/swc,v 1.3 1995/04/26 11:03:06 solomon Exp $

mountpoint=/shoremnt
program=stree

if test $1x = -ix
then
    program=doc_index
    shift
fi

if test $# -ne 1
then
    echo "usage: $0 [-i] keyword"
```

```
    exit 1
fi

prog_path=`pwd`/$program

cd $mountpoint/$program
wc `$prog_path -l $1 | sort -u`
```

7.9 sedit

```
#!/bin/sh
# ------------------------------------------------------------- #
# -- Copyright (c) 1994, 1995 Computer Sciences Department,    -- #
# -- University of Wisconsin-Madison, subject to the terms     -- #
# -- and conditions given in the file COPYRIGHT.  All Rights   -- #
# -- Reserved.                                                 -- #
# ------------------------------------------------------------- #
# $Header: /p/shore/shore_cvs/src/examples/stree/sedit,v 1.3 1995/04/26 11:03:03 solomon Exp $

mountpoint=/shoremnt
program=stree

if test $1x = -ix
then
    program=doc_index
    shift
fi

if test -t 0
then
    edit=${EDITOR:-emacs}
else
    edit="echo EDIT"
fi

if test $# -ne 1
then
    echo "usage: $0 [-i] keyword"
    exit 1
fi

prog_path=`pwd`/$program

files="`$prog_path -l $1`" || exit 1
if [ -n "$files" ]
then
```

```
    cd $mountpoint/$program
    $edit $files
else
    echo $1 not found
fi
```

7.10 docIndex

```
// Member functions of the DocIndex class
#include <iostream.h>
#include <string.h>
#include <ctype.h>
#include <sys/types.h>
#include <sys/stat.h>
#include "doc_index.h"

extern Ref<Pool> nodes;
static int getword(const char *&p, char *res, int size);

extern int verbose; // defined in main.C

void DocIndex::initialize() {
    SH_DO(ind.init(UniqueBTree));
}



void DocIndex::insert(char *s, Ref<Cite> c) {
    Ref<Word> w;
    bool found;

    SH_DO(ind.find(s,w,found));
    if (!found) {
        w = new(nodes) Word;
        w.update()->initialize(s);
        SH_DO(ind.insert(s,w));
    }
    w.update()->occurs_on(c);
}

void DocIndex::insert_file(char *fname) {
    shrc rc;

    if (verbose)
        cout << "Indexing file " << fname << endl;
```

```
// Open input file
ifstream in(fname);
if (!in) {
    perror(fname);
    SH_ABORT_TRANSACTION(rc);
}
// do a unix stat to get the total size of the file.
struct stat in_stat;
if (stat(fname,&in_stat)) {
    perror(fname);
    SH_ABORT_TRANSACTION(rc);
}

// Create target document

// Strip leading path from file name;
char *base_name = strrchr(fname, '/');
if (base_name)
    base_name++;
else
    base_name = fname;

Ref<Document> doc;
rc = doc.new_persistent(base_name, 0644, doc);
if (rc) {
    perror(base_name);
    SH_ABORT_TRANSACTION(rc);
}
doc.update()->initialize(base_name,in_stat.st_size);

// for each line of the document ...
char linebuf[1024];
while (in.getline(linebuf, sizeof linebuf -1)) {
    long off = doc->size();

    // copy the line to the body of the document
    doc.update()->append(linebuf);
    doc.update()->append("\n");

    // allocate a new Cite object for this line
    Ref<Cite> cite = new (nodes) Cite;
    cite.update()->initialize(doc, off);

    // for each word on the line ...
    char word[100];
    const char *p = linebuf;
```

```
        while (getword(p, word, sizeof word)) {
            // link the citation to the word
            insert(word, cite);
        }
    }
}



Ref<Word> DocIndex::find(char *str) const {
    Ref<Word> w;
    bool found;

    SH_DO(ind.find(str,w,found));
    if (found)
        return w;
    return NULL;
}

// Copy a word of at most SIZE characters (including terminating null)
// in to the buffer starting at RES.  Start searching at location P.
// Words are delimited by white space.  The result is translated to lower
// case, with all non-letters eliminated.
// P is updated to point to the first character not copied.
// The result is 1 if a word is found, 0 if '\0' is encountered first.
static int getword(const char *&p, char *res, int size) {
    for (;; ) {
        // skip leading white space
        while (isspace(*p))
            p++;

        // check for eoln
        if (*p == 0)
            return 0;

        // gather non-space characters, translating to lower case and
        // ignoring non-alpha characters
        int len;
        for (len = 0; len < size-1 && *p && !isspace(*p); p++) {
            if (isupper(*p))
                res[len++] = tolower(*p);
            else if (islower(*p))
                res[len++] = *p;
        }
        if (len > 0) {
            res[len] = 0;
```

```
            return 1;
        }
        // otherwise, word was all digits and punctuation, so try again.
    }
}

void DocIndex::delete_word(sdl_string w) {
    int count;
    SH_DO(ind.remove(w,count));
    if (verbose)
        cout << "deleted " << count << (count==1 ? " copy" : " copies")
            << " of word '" << (char *)w << "' from the index" << endl;
}

void DocIndex::print() const {
    // index_iter<typeof(ind)> iterator(ind);
    IndexScanIter<sdl_string,Ref<Word> > iterator(this->ind);
    SH_DO(iterator.next());
    while (!iterator.eof) {
        cout << "key: '" << iterator.cur_key << "' value: ";
        iterator.cur_val->print(1);
        SH_DO(iterator.next());
    }
}
```