Cdt: A General and Efficient Container Data Type Library

Kiem-Phong Vo
AT&T Labs

Cdt: A General and Efficient Container Data Type Library

Kiem-Phong Vo

AT&T Labs
600 Mountain Ave
Murray Hill, NJ 07974
kpv@research.att.com

Abstract

Cdt is a container data type library that provides a uniform set of operations to manage dictionaries based on the common storage methods: *list*, *stack*, *queue*, *set* and *ordered set*. It is implemented on top of linked lists, hash tables and splay trees. Applications can dynamically change both object description and storage methods so that abstract operations can be exactly matched with run-time requirements to optimize performance. This paper briefly overviews *Cdt* and presents a performance study comparing it to other popular container data type packages.

1 Introduction

A data structure to store objects is a container data type and an instance of it is a container or a *dictionary*. Common examples of container data types are: balanced and splay trees[1, 9, 10], skip lists[8], hash tables, stacks, queues and lists[2]. Each data type has unique operational and performance properties. For example, inserting an object into a stack is restricted to the stack top and takes constant time while inserting an object into a set of ordered objects represented by a balanced tree takes logarithmic time.

Container data structures are pervasive in programs and there are many library packages to deal with them. Most Unix/C environments include the functions `tsearch`, `hsearch`, `lsearch`, and `bsearch` to manipulate respectively objects stored in binary trees, hash tables, arrays and sorted arrays. C++ provides classes such as `Map` [3] and `Set` [5] to deal with ordered maps and unordered sets. Recently, a new set of C++ templates for ordered and unordered maps and sets called the Standard Template Library[6] has become increasingly popular.

Two common problems with existing container data type packages are interface confusion and performance deficiency. At the interface level, other than similar names, the Unix/C search functions have little else in common in how they manage objects. This is disconcerting because all container data types support the same basic set of abstract operations: *insert*, *delete*, *search*, and *iterate*. When such operations are realized via distinct interfaces and conflicting conventions, programmers have a hard time learning and programming them. The STL templates alleviate this problem by establishing interface guidelines for similar operations in different container data types. However, it is difficult with the STL templates to manage objects in multiple contexts because container and object types must be statically bound together to form dictionaries. Aside from interface issues, we shall see later that not only the older Unix/C packages but also the more modern C++ packages do not always perform well in both time and space usage.

This paper introduces *Cdt*, a C library for managing dictionaries based on common container data types: *list*, *stack*, *queue*, *ordered set*, *ordered multiset*, *unordered set* and *unordered multiset*. *Cdt* is unique among container packages in possessing the following characteristics:

- All dictionaries are manipulated via a uniform set of operations regardless of storage methods (i.e., container data types);
- Storage methods can be dynamically changed, for example, to turn an unordered dictionary to an ordered one;
- Object attributes are described in discipline structures that support both *set-like dictionaries*, i.e., dictionaries that identify objects by matching, and *map-like dictionaries*, i.e., dictionaries that identify objects by keys;
- Discipline structures can be dynamically

changed, for example, to change object comparators;

- Objects can be in multiple dictionaries, including dictionaries in shared memory; and
- Iterations are done directly over objects, i.e., no separate iterator types [6] needed.

Cdt uses splay trees for ordered sets and multisets, hash tables with move-to-front collision chains for unordered sets and multisets, and doubly linked lists for stacks, queues, and lists. The overall interface follows a *method* and *discipline* architecture [4, 11, 12]. Briefly, a library in this architecture provides handles and operations to hold and manipulate resource, methods to parameterize the semantics and performance characteristics of operations, and a discipline structure type that applications can use to define resource attributes and acquisition. Applying this architecture to *Cdt*, a handle is a dictionary and operations include handle creation, object insert, search, delete, etc. A *Cdt* method maps to a container data type while a discipline lets applications define information and operations that pertain directly to objects such as key type and object allocation.

2 The *Cdt* library

Objects are managed via three data types: dictionary, discipline, and method.

- *Dictionary:* A dictionary stores objects.
- *Method:* A dictionary has a method to define how objects are stored within it. Available methods are: `Dtset` for unordered sets, `Dtbag` for unordered multisets, `Dtoset` for ordered sets, `Dtobag` for ordered multisets, `Dtlist` for doubly linked lists, `Dtstack` for stacks and `Dtqueue` for queues.
- *Discipline:* Each dictionary has an application-defined discipline structure that specifies object comparison, hashing, allocation, and event announcement.

2.1 Dictionary operations

This section briefly overviews the main functions in the *Cdt* library.

`Dt_t* dtopen(Dtdisc_t* disc, Dtmethod_t* meth)` creates a dictionary of type `Dt_t` with the given discipline `disc` and method `meth`. A dictionary `dt` is closed or cleared with `dtclose(Dt_t* dt)` and `dtclear(Dt_t* dt)`.

Functions `Void_t* dtsearch(Dt_t* dt, Void_t* obj)` and `Void_t* dtmatch(Dt_t* dt, char* key)` search the dictionary `dt` for an object matching respectively `obj` or `key`. Such a matched object becomes a *current object* with special semantics in certain operations discussed below. `Void_t` is defined as `void` for ANSI-C or C++ and `char` for older C variants so it is suitable for exchanging addresses between the library and applications.

`Void_t* dtinsert(Dt_t* dt, Void_t* obj)` inserts an object `obj` into the dictionary `dt`. Methods `Dtset` and `Dtoset` allow `obj` to be inserted only if there is no matching object already in `dt`. Other methods always insert a new object because they allow insertion of equal objects. Method `Dtstack` inserts objects at stack top. Method `Dtqueue` inserts objects at queue tail. Method `Dtlist` inserts an object before the current object of `dt` if there is one, or at list head otherwise. An inserted or found object becomes the new current object.

`Void_t* dtdelete(Dt_t* dt,Void_t* obj)` is used to delete from `dt` an object matching `obj` if one exists. `dtdelete(dt,NULL)` works with `Dtstack` and `Dtqueue` and removes respectively the top or head object.

Object iteration depends on a particular object ordering defined by the storage method in use. For `Dtoset` and `Dtobag`, objects are ordered by object comparisons. For `Dtstack`, objects are ordered in the reverse order of insertion. For `Dtqueue`, objects are ordered in the order of insertion. For `Dtlist`, objects are ordered by their list positions. For `Dtset` and `Dtbag`, the object order is defined at the point of use and may change on any search or insert operation.

There are many ways to iterate over objects in a dictionary. The below loop iterates forward over all objects in a dictionary `dt`:

```
for(o = dtfirst(dt); o; o = dtnext(dt,o) )
```

Alternatively, the below loop can be used to iterate backward over objects:

```
for(o = dtlast(dt);  o; o = dtprev(dt,o) )
```

2.2 Storage methods

A storage method is of type `Dtmethod_t` and defines how objects are manipulated. *Cdt* provides the following storage methods:

- `Dtset` and `Dtbag`: These methods are based on hash tables with move-to-front collision chains. `Dtset` stores unique objects while `Dtbag` allows repeatable objects (i.e., objects that compare equal). Repeatable objects are collected together so that any iteration always passes over sections of them. Object accesses take expected O(1) time given a good hash function.

- `Dtoset` and `Dtobag`: These methods store ordered objects in top-down splay trees. `Dtoset` stores unique objects while `Dtobag` allows repeatable objects. Object accesses take amortized O(logn) time. Splay trees adapt well to biased access patterns because frequently accessed objects migrate closer to tree roots.

- `Dtlist`: This method stores repeatable objects in a doubly-linked list. An object is always inserted in front of the current object which is either the list head or established by a search, insert, or iteration. Object insertion and deletion are done in O(1) time.

- `Dtstack` and `Dtqueue`: These methods store repeatable objects in stack and queue order. In a stack order, objects are kept in reverse order of their insertion. In a queue order, objects are kept in order of their insertion. Object insertion and deletion are done in O(1) time.

2.3 Disciplines

A discipline structure is of type `Dtdisc_t`. Applications use disciplines to define object attributes such as comparison, hashing, and allocation.

Figure 1 shows `Dtdisc_t`. `Dtdisc_t.key` and `Dtdisc_t.size` identify a key of type `Void_t*` used for object comparison or hashing. `Dtdisc_t.key` defines the offset in an object where the key resides. `Dtdisc_t.size` defines the key type. A positive value means that the key is a byte array of given length, a zero value means that the key is a null-terminated string, and a negative value means that the key is a null-terminated string whose address is stored at the key offset.

```
typedef struct
{ int         key;     /* key offset    */
  int         size;    /* key size/type */
  int         link;    /* object holder */
  Dtmake_f    makef;   /* object makef  */
  Dtfree_f    freef;   /* object freef  */
  Dtcompar_f  comparf; /* comparator    */
  Dthash_f    hashf;   /* hash function */
  Dtmemory_f  memoryf; /* allocator     */
  Dtevent_f   eventf;  /* event handler */
} Dtdisc_t;
```

Figure 1: A discipline structure

Objects are held in a dictionary via holders of type `Dtlink_t`. If `Dtdisc_t.link` is negative, the library will allocate object holders. Otherwise, the library assumes that object holders are embedded inside objects and `Dtdisc_t.link` defines the offset in an object where the holder resides.

`Dtdisc_t.makef` and `Dtdisc_t.freef`, if defined, are called to make and free objects when they are inserted or deleted. If `Dtdisc_t.makef` is not defined, then in the call `dtinsert(dt,obj) obj` itself will be inserted.

If `Dtdisc_t.comparf` or `Dtdisc_t.hashf` are not defined, some internal functions are used. By allowing both key definition and compare function in a discipline, both set-like and map-like dictionaries are supported.

`Dtdisc_t.memoryf`, if defined, is used to allocate space. `Dtdisc_t.eventf`, if defined, announces various events such as dictionary opening and closing and method or discipline changes.

2.4 An example *Cdt* application

A common container data type example is given in the *Map* associative array paper [3] and the Unix/C manual page for the function `tsearch()`. This application reads a text file, partitions it into tokens (strings separated by space, tab and new line characters), keeps frequency count for each token, and finally writes out the tokens and their frequencies.

Figure 2 shows an implementation of the token counting example. Omitted are a few minor grammatical statements and the function `readtoken()` to parse an input stream into tokens. The below comments are based on line numbers in the figure:

```
 1. #include     <sfio.h>
 2. #include     <cdt.h>

 3. typedef struct
 4. {   Dtlink_t link;
 5.     char*    token;
 6.     int      freq;
 7. } Token_t;

 8. Dtdisc_t Tkdisc =
 9.   { offsetof(Token_t,token), -1, 0 };

10. Token_t* newtoken(char* s)
11. {   Token_t* tk;
12.     tk = malloc(sizeof(Token_t));
13.     tk->token = malloc(strlen(s)+1);
14.     strcpy(tk->token,s);
15.     tk->freq = 1;
16.     return tk;
17. }

18. main()
19. {   char*    s;
20.     Token_t* tk;
21.     Dt_t*    dt = dtopen(&Tkdisc,Dtset);

22.     while((s = readtoken(sfstdin)) )
23.     {   if((tk = dtmatch(dt,s)) )
24.                 tk->freq += 1;
25.         else dtinsert(dt,newtoken(s));
26.     }

27.     for(tk = dtfirst(dt); tk;
28.         tk = dtnext(dt,tk) )
29.         sfprintf(sfstdout,"%s:\t%d\n",
30.                  tk->str, tk->freq);
31. }
```

Figure 2: Program to count tokens

`1-2:` The header file `sfio.h` [4] declares I/O functions. `cdt.h` is the *Cdt* public header file and declares necessary types, values and functions.

`3-7:` `Token_t` is a structure to hold a string `token` and a frequency count `freq`. It also embeds the container holder structure in the `link` field.

`8-9:` The discipline `Tkdisc` describes attributes of `Token_t` objects. The ANSI-C macro `offsetof()` defines the offset of `Token_t.token` in `Token_t`. Since `Token_t.token` points to a null-terminated string, `Tkdisc.size` is set to `-1`. `Tkdisc.link` is set to `0`, the offset to `Token_t.link` in `Token_t`.

`10-17:` `newtoken()` is a function to create a new `Token_t` structure from a given string `s`. To simplify the exposition, error checks for the `malloc` calls were omitted.

`21:` A new dictionary `dt` is created based on the discipline `Tkdisc` and the method `Dtset`. Here it is assumed that tokens need not be sorted. Otherwise, `Dtoset` could be used (see also Section 4).

`22-26:` Tokens are read and inserted into `dt`. Line 22 uses `dtmatch()` to find out if a token matching the current read token already exists in `dt`. In that case, only its frequency count is updated. Otherwise, Line 24 creates and inserts a new token structure into `dt`.

`27-30:` These lines loop over all tokens and output both tokens and their frequency counts. Note that this is done directly over objects without the aid of any iterator type [6].

3 Performance

Among the various container data types, hash tables and binary trees are most common and also have large variation in implementation quality. This section presents results from a performance study that compared various set and map container structure packages based on hash tables and binary trees.

3.1 Methodology

The token counting application in Section 2.4 was used as a benchmark. To minimize implementation variation, a single program based on Figure 2 was written. Compile time options allowed switching usages of the *Cdt* methods `Dtoset` and `Dtset`, the Unix/C package `tsearch`, the C++ classes `Set` and `Map`, and the STL templates `map` and `hashmap`. All implementations used the same string comparison function, a variant of `strcmp()` that also keeps invocation count. In addition, all hash table implementations used the same hash function supplied by *Cdt* so that hash value computation would be uniform. This was necessary because the default hash functions in some of the packages were not very good. For example, the comparison counts in Section 3.2 for the `Set` package would have been much higher if its default hash function was used.

A variety of input files were used:

- *ps*: PostScript source of a technical paper,
- *src*: an archive of C source code,
- *kjv*: a King James version of the bible,
- *mbox*: a personal mail archive,
- *host*: a database mapping IP addresses to machine hosts, and
- *city*: a database mapping cities to area codes.

File	Size	Tokens	Distinct	Length
ps	1,989K	335,997	11,912	38.00
src	1,169K	149,886	27,964	16.40
kjv	4,441K	822,587	33,916	8.01
mbox	2,701K	419,197	49,903	9.83
city	1,349K	81,206	69,610	18.17
host	2,722K	449,554	102,566	16.71

Table 1: Summary of benchmark input files

Table 1 summarizes input file statistics: file size in K-bytes, total number of tokens, number of distinct tokens, and average length of a token. These input files represent a wide variety of data ranging from *ps* which has relatively few distinct tokens to *city* which has about 85% distinct tokens. Tokens in *host* and *city* are also highly ordered.

Program	Size
`Set`	146,632
`hashmap`	179,988
`map`	145,892
`Map`	83,795
`tsearch`	66,744
`Dtset+Dtoset`	73,712

Table 2: Sizes of benchmark programs

The experiment was performed on a SPARC-20 running SUN OS5.0. Table 2 shows the sizes of the benchmark programs. Both `Dtset` and `Dtoset` were combined in the same benchmark program with invocation options for method selection. Except for `Map`, other C++ packages caused the test code to be about twice as large as the C versions, a sign of code bloating due to the use of templates. Comparing the results of compiling `main(){}` with C and C++ showed that only about 5K can be attributed to language difference.

Program execution was done at night on a quiescent machine. Each time measurement was obtained by running the same test 9 times, computing total cpu and system times for each run, discarding the top two and bottom two scores to reduce variance, then averaging the remaining five scores. Space measurements were done by calling `sbrk(0)` before any dictionary was opened and after all output was done and computing differences in the return values.

3.2 Hash table packages

Below are brief descriptions of the container packages that use hash tables to implement unordered sets and maps. The Unix/C `hsearch` package was omitted because it was too slow to measure.

- `Set`: The C++ `Set` class that comes standard with our compiler. This uses a hash table with chaining to resolve collisions.
- `hashmap`: The C++ STL `hashmap` template. This uses a hash table with chaining to resolve collisions.
- `Dtset` Method `Dtset` of *Cdt*. This uses a hash table with chaining. The collision chains use a move-to-front heuristic to improve search time.

Dataset	`Set`	`hashmap`	`Dtset`
ps	328	682	324
src	131	566	122
kjv	799	4,038	789
mbox	386	1,839	370
city	35	292	12
host	384	2,279	347

Table 3: Hash: comparison counts in thousands

Table 3 shows comparison counts for the hash table packages in units of thousands. `hashmap` performed worst, with comparison counts many times higher than that of `Set` and `Dtset`. `Dtset` asserted that tokens compared equal must have the same hash values. This fact was used effectively to reduce many comparisons because the hash function distinguished objects well. The low comparison counts for `Set` suggested that it might have used the same strategy as `Dtset`. `Dtset` retains a slight edge perhaps due to its move-to-front strategy on collision chains.

Table 4 shows time performance. Poor comparison counts directly translated to poor computing time. `hashmap` was worst, sometimes up to a factor of 3

Dataset	`Set`	`hashmap`	`Dtset`
ps	7.47	10.76	4.32
src	6.06	7.57	3.59
kjv	20.61	31.49	11.01
mbox	14.27	19.84	8.20
city	9.49	10.66	6.49
host	18.28	23.31	10.78

Table 4: Hash: times in seconds

Dataset	`Set`	`hashmap`	`Dtset`
ps	1,464	976	912
src	2,752	1,432	1,496
kjv	3,048	1,448	1,584
mbox	4,632	2,232	2,360
city	7,144	3,688	3,944
host	10,552	5,248	5,496

Table 5: Hash: space in K-bytes

slower than `Dtset`. `Dtset` was fastest among the three packages.

Table 5 shows space usage. `Dtset` and `hashmap` used about the same amount of space. `Set` sometimes used twice as much space as the other packages.

3.3 Binary tree packages

A further requirement could be stimulated in the token counting example that tokens must be output in a lexicographic order. In that case, a natural solution is to use container packages that maintain ordered tokens. Below are the studied container packages for ordered sets and maps:

- `tsearch`: The `tsearch` function in SUN OS5.4. This uses plain binary trees.
- `Map`: The C++ `Map` class. This is based on AVL balanced trees.
- `map`: The C++ STL `map` template. This uses red-black balanced trees.
- `Dtoset`: Method `Dtoset` of *Cdt*. This uses top-down splay trees.

Table 6 shows comparison counts for the binary tree methods. Except for *city* and *host*, `tsearch` performed well despite its simplistic data structure. This is because most datasets consist of more or less random tokens and binary trees built from such random data are naturally balanced. `tsearch` did poorly on *city* and *host* whose tokens were highly ordered. The balanced tree packages `Map` and `map` ignored any such ordering property in the data. Both packages used about the same number of comparisons with `map` having a slight edge. The splay tree approach in `Dtoset` took advantage of data ordering to reduce comparisons. As a result, `Dtoset` was the clear winner in all cases.

Dataset	`Map`	`map`	`tsearch`	`Dtoset`
ps	14,267	10,434	7,314	1,636
src	5,857	4,553	3,533	1,773
kjv	28,142	26,119	13,526	8,592
mbox	15,747	13,229	7,396	5,425
city	3,841	3,006	12,189	1,631
host	22,963	15,657	19,750	2,568

Table 6: Tree: comparison counts in thousands

Dataset	`Map`	`map`	`tsearch`	`Dtoset`
ps	15.37	15.51	13.26	5.33
src	9.63	9.05	8.20	5.53
kjv	37.08	44.79	26.77	21.27
mbox	24.22	28.13	17.86	15.85
city	12.01	11.58	25.60	8.92
host	30.19	26.58	61.00	13.76

Table 7: Tree: times in seconds

Table 7 shows time performance. As with the hash table methods, comparison counts mapped directly to time. `Dtoset` was fastest, sometimes by a factor of 3 or more over some of the other methods. Note that `Dtoset` was even faster than the STL `hashmap` package which did not have to order tokens.

Table 8 shows memory usage. `tsearch` and `Map` used more memory than other methods. `Map`'s extra space was due to the balancing data. The cause for `tsearch`'s poor memory usage was unclear although a memory trace using *Vmalloc* [12] revealed a mysterious extra allocation after each holder allocation.

Dataset	`Map`	`map`	`tsearch`	`Dtoset`
ps	1,064	872	1,064	880
src	1,864	1,424	1,872	1,432
kjv	1,968	1,440	1,976	1,440
mbox	3,008	2,224	3,016	2,232
city	4,768	3,672	4,776	3,680
host	6,840	5,240	6,856	5,240

Table 8: Tree: space in K-bytes

4 Flexible programming with *Cdt*

To output tokens in order, a strategy that often works better than just using `Dtoset` is as follows. First, `Dtset` is used to construct the token dictionary. Then, `Dtoset` is used right before outputting to sort tokens into the right order. To implement this strategy, the below line of code can be inserted before Line 26 of Figure 2:

```
dtmethod(dt,Dtoset);
```

Dataset	`Dtset`	`Dtoset`	`Dtset+Dtoset`
ps	324	1,636	498
src	122	1,773	608
kjv	789	8,592	1,307
mbox	370	5,425	1,282
city	12	1,631	1,420
host	347	2,568	2,014

Table 9: Tuning: comparison counts in thousands

Table 9 shows comparison counts for the above strategy. Except for *city* and *hosts*, `Dtset+Dtoset` improves substantially over the exclusive use of `Dtoset`, up to 70% for *kjv*.

Dataset	`Dtset`	`Dtoset`	`Dtset+Dtoset`
ps	4.32	5.33	4.83
src	3.59	5.53	5.12
kjv	11.01	21.27	12.50
mbox	8.20	15.85	11.55
city	6.49	8.92	12.73
host	10.78	13.76	18.38

Table 10: Tuning: times in seconds

Table 10 show time performance. Time usages for `Dtset+Dtoset` markedly improved over the lone use of `Dtoset` on most datasets except for *city* and *hosts*. For these datasets, though comparison counts went down somewhat, time measurements actually went up. This was because these datasets contained many distinct tokens and `Dtoset` ended up repeating `Dtset`'s work.

The above situation is common in practice. Programs must often deal with data that have special characteristics. It is seldom the case that efficient algorithms can be devised to adapt smoothly to the data diversity and operate optimally in each special situation. Therefore, whenever possible, a good design principle is to let users select and combine

```
 1. int freqcmp(Dt_t* dt, Void_t* arg1,
 2.             Void_t* arg2, Dtdisc_t* disc)
 3. {   int      d;
 4.     Token_t* t1 = (Token_t*)arg1;
 5.     Token_t* t2 = (Token_t*)arg2;
 6.     if((d = t1->freq - t2->freq) != 0)
 7.         return d;
 8.     else return strcmp(t1->token,t2->token);
 9. }

10. Tkdisc.comparf = freqcmp;
11. Tkdisc.key = Tkdisc.size = 0;
12. dtdisc(dt,&Tkdisc,DT_SAMEHASH|DT_SAMECMP);
13. dtmethod(dt,Dtoset);
```

Figure 3: Order tokens by frequency

computing methods to optimize processing based on specific knowledge of the data. *Cdt* simplifies doing this in the context of using container data types. In fact, the benchmark program was written to allow strategy selection at invocation time. It is not easy to do the same using the other container packages.

As another example of *Cdt*'s flexibility, suppose that the output requirement is changed to ordering tokens in increasing order of frequency. To do this, Figure 2 should be augmented with Lines 1-9 of Figure 3 before `main()` and Lines 10-13 of the same figure before Line 26. The below comments pertain to line numbers in Figure 3:

- `1-9`: The function `freqcmp()` compares tokens first by frequency, then by token names. So within a group of tokens with the same frequency, tokens will be ordered lexicographically.
- `10`: The comparator is redefined to be `freqcmp()`.
- `11`: `Tkdisc.key` and `Tkdisc.size` are set to `0` to indicate that `Token_t` objects will be compared whole instead of via the key strings `Token_t.token`.
- `12`: `dtdisc()` is called to officially change the discipline. Normally, a discipline change implies rearranging of objects because hash values may have changed or objects that used to compare disctint may have become equal. The flags `DT_SAMEHASH` and `DT_SAMECMP` tell `dtdisc()` that, in this case, both hash values and object comparison remain unchanged. The latter is strictly untrue but it saves computation that would be done anyway on line `13`.

13: `dtmethod(dt,Dtoset)` is called to switch the storage method to `Dtoset` and sort tokens by the new comparator.

Note that in this example it is possible to use method `Dtoset` with the comparator `freqcmp()` from the start of the application. However, doing so would have been prohibitively expensive because objects must be deleted and reinserted each time their frequencies are updated. Thus, for efficiency, it is necessary that `Dtset` is used during dictionary construction and `Dtoset` is used only at the end before outputting.

5 Discussion

This paper introduced *Cdt*, a container data type library. The library provides the common storage methods: *set*, *multiset*, *ordered set*, *ordered multiset*, *list*, *stack* and *queue* which are often seen only in isolated packages. *Cdt* achieves an interface that keeps orthogonal the three design dimensions: dictionary operations, storage methods, and object descriptions. This is a goal attempted but not quite achieved by other recent work on reusable components such as the C++ Standard Template Library.

Many contemporary container libraries are unwieldy because their interfaces are not sufficiently abstract and operations are tied too closely to container data types. At worst, this leads to divergent interfaces for the same basic operations as shown by the Unix/C search functions. Even with better interface design as in the STL case, the close tie between implementation techniques and abstract interfaces can reduce the generality of the library. For example, instead of a general container template that can be parameterized by storage methods, STL provides various container templates such as `hashmap` and `map` that are strongly bound to minimal object requirements according to the respective implementation techniques. As a result, although `hashmap` and `map` provide similar functions they require objects with different type specifications. This means that there is no simple way to dynamically convert a `hashmap` container to a `map` container in the style discussed in Section 4. *Cdt* avoids such interface limitations by making dictionary operations completely abtract and parametrizable by methods and disciplines which are orthogonal and mutable attributes of dictionaries. The method and discipline architecture naturally lifts a library interface to its most general level. Perhaps some future STL work can benefit from such an interface analysis and design.

Cdt disciplines are run-time structures used to define object attributes such as keys, comparison, hashing, and allocation. By allowing both comparators and keys, *Cdt* generalizes set-like and map-like container packages. This leads to a unifying interface to manage such containers. Using run-time structures for type definition means losing certain services common to C++ templates such as static type checking and inlining of comparison functions. The loss of static type checking is balanced out by the added programming flexibility. For example, *Cdt* allows the same objects to be described in multiple ways and both disciplines (i.e., object types) and methods (i.e., container types) can be arbitrarily mixed and changed. The efficiency loss resulted from no inlining of object comparisons is compensated by the advantage of having a single library code image and consequent code size reduction as exemplified in Table 2. A single code image also makes possible using *Cdt* as a dynamically loadable shared library. Further, for applications such as the discussed token counting example which require relatively complex objects, any function call overhead to compare two objects would be negligible relative to the cost of the comparison itself.

A performance study showed that *Cdt* methods `Dtoset` and `Dtset` performed as well or better than their counterparts in other C and C++ container libraries including the modern STL components. The *Cdt* methods consistently used about the same or less space than other packages while they were faster than other packages by up to a factor of two or more. The use of splay trees and hash tables with self-adjusting collision chains enable these methods to perform well in a wide range of input data. Examples were given showing how further performance gains can be made with selective matching of disciplines and methods at run time.

Cdt is a descendant of *Libdict* [7]. It is a mature library and has been used in many applications including large-scale information systems that routinely handle dictionaries with tens to hundreds thousands of objects.

Acknowledgement

Cdt evolved over many years and benefited from advices and demands of many friends and users. In particular, I'd like to thank Glenn Fowler, David

Korn, and Stephen North who patiently survived several generations of interface changes. In addition, I'd like to thank Carl Staelin whose careful reading of this paper helped improving it greatly.

Code availability

Cdt source code is available at:
`http://www.research.att.com/sw/tools/reuse/`.

REFERENCES

[1] G.M. Adelson-Velskii and E.M. Landis. An Algorithm for the Organization of Information. *Soviet Math. Doklady*, 3:1259–1263, 1962.

[2] Donald E. Knuth. *The Art of Computer Programming, Volume 3: Sorting and Searching*. Addison-Wesley, 1973.

[3] Andrew R. Koenig. Associative Arrays in C++. In *Proceedings of Summer 1988 USENIX Conference*, pages 173–186, 1988.

[4] David G. Korn and Kiem-Phong Vo. SFIO: Safe/Fast String/File IO. In *Proc. of the Summer '91 Usenix Conference*, pages 235–256. USENIX, 1991.

[5] Unix System Laboratories. *USL C++ Standard Components Programmer's Reference*. AT&T and Unix System Laboratories, Inc., 1990.

[6] David R. Musser and Atul Saini. *STL Tutorial and Reference Guide*. Addison-Wesley, 1995.

[7] Stephen C. North and Kiem-Phong Vo. Dictionary and Graph Libraries. In *Proc. of the Winter '93 Usenix Conference*, pages 1–11. USENIX, 1993.

[8] T. Papadakis. *Skip Lists and probabilistic Analysis of Algorithms*. University of Waterloo, 1993.

[9] Robert Sedgewick. *Algorithms, 2nd Edition*. Addison-Wesley, 1988.

[10] D. Sleator and R.E. Tarjan. Self-Adjusting Binary Search Trees. *JACM*, 32:652–686, 1985.

[11] Kiem-Phong Vo. Writing Reusable Libraries with Disciplines and Methods. In *Practical Reusable UNIX Software*. John Wiley & Sons, 1994.

[12] Kiem-Phong Vo. Vmalloc: A General and Efficient Memory Allocator. *Software Practice & Experience*, 26:1–18, 1996.