Cdt: A Container Data Type Library

KIEM-PHONG VO

2B-112, AT&T Laboratories, 600 Mountain Avenue, Murray Hill, NJ 07974, U.S.A.

(kpv@research.att.com)

SUMMARY

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*, *ordered set/multiset*, **and** *unordered set/multiset*. **Both object description and storage method in a dictionary can be dynamically changed so that abstract operations can be exactly matched with run-time requirements for operational flexibility and performance. A study comparing Cdt and other popular container packages shows that Cdt performs best in both computing time and space usage.**

KEYWORDS: set, map, stack, queue, list, dictionary, method, discipline, binary tree, hash table

1 Introduction

A container data type is an interface to manage collections of objects. An instance of a container data type is a container or a *dictionary*. Common data structures to implement container data types are: balanced trees[1, 2] , skip lists[3] , hash tables, stacks, queues and lists[4] . Each data structure has unique operational and performance properties. Stack insertion is restricted to stack top and takes constant time while balanced tree insertion takes logarithmic time. Container data types are pervasive in programs and there are many library packages to deal with them. Unix/C environments provide functions `tsearch`, `hsearch` and `lsearch` to manipulate respectively objects stored in binary trees, hash tables, and lists. C++ provides classes such as `Map` [5] and `Set` [6] to deal with ordered maps and unordered sets. The C++ community is quickly converging on the Standard Template Libraries[7] , STL, a set of templates that include ordered and unordered maps and sets.

Many container data type packages suffer from interface confusion and rigidity due to insufficient distinction between abstract container data types and their implementation data structures. The worst example of this is the family of Unix/C search functions which, from function to function, present completely different object management interfaces. All container data types support the same abstract set of operations: *insert*, *delete*, *search*, and *iterate*. When such basic operations employ distinct and conflicting interfaces, programmers have a hard time learning and programming them. The newer STL templates alleviate this problem by establishing guidelines so that similar operations in different container data types use the same interfaces. However, container and object types must be statically bound. This tends to reduce flexibility in object interpretation and matching of dictionary types to usage contexts.

Cdt is a library for managing dictionaries based on the common container data types: *list*, *stack*, *queue*, *ordered set/multiset*, and *unordered set/multiset*. Ordered sets and multisets are implemented with splay trees[8] , unordered sets and multisets are based on hash tables with move-to-front collision chains, and stacks, queues, and lists use doubly-linked lists. These implementation data structures were chosen for their simplicity and good performance. In all cases, object storage requires a minimum overhead of two or three pointers per object.

The Cdt programming interface possesses the following characteristics:

- Dictionary operations use a uniform abstract interface independent of container types;
- Container types are dynamically changeable, for example, to change a dictionary from being unordered to being ordered;
- Object attributes are described by a discipline structure that supports 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 are dynamically changeable, for example, to change how objects are compared and identified;
- Iterations are done directly over objects, i.e., no separate iterator types [7] are needed; and
- Sharing objects between dictionaries and sharing dictionaries between processes are supported.

The remainder of the paper is organized as follows. Section 2 discusses dictionary types and operations and gives examples of their usage. Section 3 presents a performance study showing that Cdt outperforms other common container packages and gives examples of performance tuning using the flexible Cdt interface. Section 4 summarizes and discusses the results.

2 The Cdt library

The Cdt interface follows a *method and discipline architecture* [9, 10, 11] . Briefly, a library in this architecture provides:

- A handle type to hold resources;
- A set of abstract operations on handles and resources;
- A set of *method*s, each of which encapsulates the semantics and performance characteristics of a particular implementation of operations; and
- A *discipline* type to define resource attributes.

Dictionary handle	**Abstract operations**
Objects	Dictionary opening and closing
Dictionary states	Object insertion, deletion, search, iteration
	Method and discipline changes

Discipline type	**Storage methods**
Key, comparator, hasher	Set, multiset
Constructor, destructor	Ordered set, ordered multiset
Event handler	List, stack and queue

Figure 1: A method and discipline architecture for Cdt

Figure 1 shows the method and discipline architecture as applied to Cdt. The left part shows that dictionary handles hold states and objects (i.e., dictionary resources) whose attributes are parameterized by a discipline type. The right part shows that dictionaries, objects, and their types are manipulated by abstract operations which are parameterized by storage methods. Thus, the top part of Figure 1 is completely abstract, i.e., independent from concrete object types and storage methods. Further, though not true in general of a method and discipline architecture, Cdt objects are independent from storage methods and, to a large extent, from disciplines. This means that both methods and disciplines are dynamically changeable. Later examples show that this feature of the Cdt interface is a key for both flexible application construction and efficient performance tuning.

The below subsections discuss Cdt data types, operations, methods, and disciplines in detail. In many cases, this is done via examples based on code fragments from the program shown in Figure 2. This example application is similar to ones from the *Map* associative array paper [5] and the Unix/C manual page for the function `tsearch()`. The program reads tokens from the standard input stream, calculates the frequency count for each distinct token, and outputs tokens with their frequencies. Note that omitted from Figure 2 are a few inessential grammatical statements and the function `readtoken()` to parse the input stream into tokens.

2.1 Dictionary data types and operations

Lines `1-16` of Figure 2 show how the application imports and defines data types and functions. The header file `sfio.h` [9] is included for I/O functions. The Cdt header file `cdt.h` declares dictionary data types, values and functions. For example, it defines `Dt_t`, the type of dictionary handles, `Dtdisc_t`, the type for disciplines (Section 2.3), and `Dtmethod_t`, the type for methods (Section 2.2). `cdt.h` also defines a type `Void_t*` suitable for exchanging addresses between the library and applications. `Void_t` is defined to be `void` for ANSI-C and C++ and `char` for older C flavors.

The `Token_t` data structure defined on lines `3-7` associates a token `name` and its frequency count `freq`. Embedded in `Token_t` is an object holder structure `link` of type `Dtlink_t` (Section 2.3). This is an option provided by Cdt to allow an application to allocate both an object and its holder together, thus saving time and reducing memory fragmentation.

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

typedef struct
{   Dtlink_t link;
    int      freq;
    char*    name;
} Token_t;

Dtdisc_t     Tkdisc = { offsetof(Token_t,name), -1, 0 };

Token_t* newtoken(const char* s)
{   Token_t* tk;
    tk = malloc(sizeof(Token_t));
    tk->name = malloc(strlen(s)+1);
    strcpy(tk->name,s);
    tk->freq = 1;
    return tk;
}

main()
{   char*    s;
    Token_t* tk;
    Dt_t*    dt = dtopen(&Tkdisc,Dtset);

    while((s = readtoken(sfstdin)) )
    {   if((tk = dtmatch(dt,s)) )
            tk->freq += 1;
        else dtinsert(dt,newtoken(s));
    }

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

Figure 2: A program to count token frequencies

The discipline `Tkdisc` on line `8` describes attributes of `Token_t` to Cdt. Disciplines will be discussed in Section 2.3. The function `newtoken()` on lines `9-16` creates `Token_t` structures from strings given in the argument `s`. Note that omitted are the error checks for the `malloc` calls.

2.1.1 Object manipulation

Line `20` of Figure 2 shows that a dictionary `dt` is opened via the call `dtopen(&Tkdisc,Dtset)`. This call requires a discipline to describe objects and a method to manage them. In this case, the method `Dtset` (Section 2.2) is selected to keep unordered tokens in a hash table. A dictionary `dt` can be closed with `dtclose(dt)` or cleared (i.e., deleting all objects without closing) with `dtclear(dt)`.

Line `22` shows the call `dtmatch(dt,s)` to search for a token matching a key `s`. Thus, `dt` is being used as a map-like dictionary that maps token strings to their frequencies. Line `23` updates the frequency of a token if the search succeeds. An object matching a given object `obj` can also be found via the call `dtsearch(dt,obj)`.

Line `24` shows the call `dtinsert(dt,newtoken(s))` to create a new token via `newtoken(s)` and insert it into `dt`. The semantics of `dtinsert()` depends on the method in use. Here, `Dtset` is a set method and will not allow repeated objects so an insertion will fail if there is already a matching object in the dictionary. The `dtmatch()` test on line `22` guarantees that this `dtinsert()` call is safe.

The call `dtdelete(dt,obj)` is used to delete from `dt` an object matching a given object `obj`. Note that `obj` itself does not have to be in `dt`.

A successful search, insert, or iterate operation defines a *current object*. This object can be obtained via the call `dtfinger(dt)`. Normally, an object `obj` should not be changed while it is in a dictionary as that may violate certain internal data structure. However, a frequent operation is to search for an object and update it. If such an update causes a change in how the object is identified (e.g., changing its key), the call `dtrenew(dt,obj)` can be used to update the internal dictionary structure. `dtrenew()` works only if `obj` is the current object of `dt`. Section 2.4 gives an example of using `dtrenew()`.

2.1.2 Iteration

Object iteration depends on a particular object ordering defined by the storage method in use. Cdt provides methods: `Dtoset`, `Dtobag`, `Dtstack`, `Dtqueue`, `Dtlist`, `Dtset` and `Dtbag` (Section 2.2). `Dtoset` and `Dtobag` order objects by comparison. `Dtstack` orders objects in the reverse order of their insertion. `Dtqueue` orders objects in the order of their insertion. `Dtlist` orders objects by their list positions. `Dtset` and `Dtbag` fix the object ordering only at the point of use and may change this ordering on any search or insert operation.

Line `26` of Figure 2 shows that a way to iterate over all objects in dictionary `dt` is:

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

The call `dtfirst(dt)` initializes `tk` to the first object in the dictionary `dt`. Then, `dtnext(dt,tk)` is continually called to iterate over all following objects. A noteworthy point here is that the above iteration loop does not require a separate iterator type; iteration is done directly over objects. It is also possible to loop over objects in `dt` in reverse order via the below loop:

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

When objects are ordered (`Dtoset` and `Dtobag`), `dtnext(dt,tk)` and `dtprev(dt,tk)` can be meaningful even if `tk` is not in the dictionary. For example, suppose that `dt` currently contains two tokens `1` and `3`. Suppose that `tk` is a `Token_t` representing `2`, then even though `dtsearch(dt,tk)` fails, `dtnext(dt,tk)` and `dtprev(dt,tk)` will return `3` and `1` respectively.

When many iterations over the same set of objects are required, the function call overhead for `dtnext()` and `dtprev()` can be significant. A more efficient iteration style uses the call `dtflatten(dt)` to get a linked list of objects then traverses and accesses objects via the macro calls `dtlink(dt,link)` and `dtobj(dt,link)`. For example, suppose that `link` is of type `Dtlink_t*`, lines `26-27` of Figure 2 could be rewritten somewhat more efficiently as:

```
for(link = dtflatten(dt); link; dtlink(dt,link))
{   tk = dtobj(dt,link);
    sfprintf(sfstdout, "%s:\t%d\n", tk->name, tk->freq);
}
```

The call `dtwalk(dt,userf,data)` provides yet another way to iterate over objects and execute actions. Function `userf` is called as `(*userf)(dt,obj,data)` for each object `obj` in dictionary `dt`. A walk can be terminated early by having `userf` return a negative value.

2.1.3 Viewpath of dictionaries

Many applications keep multiple related dictionaries. For example, a language parser may maintain a dictionary of tokens per scope. In that case, it is useful to connect dictionaries in nested scopes so that a token search can start from an inner scope and continue to outer scopes. To do this, the call `dtview(dt1,dt2)` can be used to establish a view from a dictionary `dt1` to another dictionary `dt2`. Such connected views form a *viewpath*. `dtview()` disallows cycles in viewpaths to avoid infinite loops. Note also that an application is responsible for ensuring that dictionaries on a viewpath are compatible with respect to object identification (Section 2.3).

As an example of using viewpath, consider three dictionaries `dt1`, `dt2`, and `dt3` on a viewpath from `dt1` to `dt2` to `dt3`. Suppose that `dt1` contains tokens `1`, `2`, `3`; `dt2` contains tokens `2`, `3`, `4`; and `dt3` contains tokens `1`, `4`, and `5`. A search for `5` starting from `dt1` will return the corresponding token in `dt3`. An iteration starting from `dt1` will see the sequence `1`, `2`, `3`, `4` and `5` while an iteration starting from `dt2` will see the sequence `2`, `3`, `4`, `1` and `5`. Note that objects in a dictionary already seen in an earlier dictionary are masked. In addition, objects in a dictionary are always seen together.

2.2 Methods

A storage method is of type `Dtmethod_t` and defines how objects are managed. A dictionary can change its method at run time to suit the computational style needed.

2.2.1 Storage methods

The Cdt storage methods are:

- `Dtset` and `Dtbag`: These methods are based on hash tables with move-to-front collision chains. The move-to-front heuristic enables frequently accessed objects to migrate closer to the chain fronts. This works well in general and is particularly useful when an application selects a bad hash function. `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 where n is the number of objects. Amortization means that the time bound is not per operation, but it is a bound over the average time per operation in a long enough sequence of operations. This is fine for Cdt because typical dictionaries are built from scratch so any sequence of operations is long enough. A splay tree also adapts well to biased access patterns because frequently accessed objects are migrated closer to the tree root.
- `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 while a search may take O(n) time where n is the current number of objects in the list.
- `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. In addition to the usual use of `dtdelete()`, the call `dtdelete(dt,NULL)` is valid only for a dictionary `dt` using `Dtstack` or `Dtqueue` and deletes respectively the stack top or the queue tail. Object insertion and deletion take O(1) time.

2.2.2 Method changing

Suppose that the token counting application requires that tokens are output in a lexicographic order. This can be accomplished in Figure 2 by simply switching method from `Dtset` to `Dtoset` before the output loop on lines `26-27`:

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

Note that `dt` could have been opened with `Dtoset` to start with. However, doing so might be sub-optimal on certain types of input. During the construction phase only token existence is of interest, not their order and a search tree may consume more time than a hash table. Benchmark results in Section 3.4 show that mixing methods strategically can have significant impact on performance.

2.3 Disciplines

Object attributes such as key, comparator, hasher, constructor and destructor are defined in a discipline structure of type `Dtdisc_t`. A discipline structure must be specified when a dictionary is opened and can be dynamically changed.

2.3.1 Discipline structure

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

Figure 3: A discipline structure

Figure 3 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: (1) a positive value for a byte array of given length, (2) a zero value for a null-terminated string, and (3) a negative value for a null-terminated string whose address is stored at the key offset.

Objects are held in a dictionary via holders of type `Dtlink_t`. When `Dtdisc_t.link` is non-negative, holders are assumed to be embedded inside objects at the offset defined by `Dtdisc_t.link`. Otherwise, object holders will be allocated separately. For example, line 8 of Figure 2 defines `Tkdisc.link` to be `0` indicating the position of the field `Token_t.link` in `Token_t`. If `Tkdisc.link` was negative, Cdt would allocate holders separately. In that case, the field `Token_t.link` would become redundant and can be removed.

`Dtdisc_t.makef` and `Dtdisc_t.freef`, if not `NULL`, specify functions to make and free objects when they are inserted or deleted. For example, a call `dtinsert(dt,obj)` will insert `obj` itself if `makef` is `NULL`. Otherwise, `(*makef)(dt,obj,disc)` is issued to construct a new object based on the prototype object `obj`. `disc` is the discipline in use.

`Dtdisc_t.comparf`, if not `NULL`, is called as `(*comparf)(dt,e1,e2,disc)` to compare two elements `e1` and `e2`. A negative, zero, or positive return value indicates respectively that `e1` is less than, equal, or larger than `e2`. If `Dtdisc_t.comparf` is `NULL`, keys are compared by one of the ANSI-C functions `memcmp()` or `strcmp()`, with a particular choice depending on `Dtdisc_t.size`. For example, in Figure 2, `strcmp()` will be used since `Tkdisc.comparf` is `NULL` and `Tkdisc.size` is negative.

`Dtset` and `Dtbag` are based on hash tables and require key hashing, i.e., a mapping of keys to unsigned integer values. Though this mapping does not have to be one-to-one, *Cdt requires that keys compared equal must have equal hash values.* If `Dtdisc_t.hashf` is not `NULL`, it specifies a function called as `(*hashf)(dt,key,disc)` to compute from `key` a hash value of type `unsigned int`. If `Dtdisc_t.hashf` is `NULL`, an internal string hashing function is used.

By default, object holders and other internal structures are allocated and freed via the ANSI-C functions `malloc()` and `free()`. When `memoryf` is defined, it is used to allocate and free memory in calls of the form `(*memoryf)(dt,area,size,disc)`. The arguments `area` and `size` are treated similarly to the corresponding arguments of the ANSI-C function `realloc()`. By defining appropriate `Dtdisc_t.memoryf` functions, dictionaries can be built in any type of memory. Accesses to a dictionary always start from its *managing area* which shall be the first area to be allocated via a `memoryf` call. This fact could be used to build dictionaries in shared or persistent memory (Section 2.4.2)

During dictionary operations, certain events may be generated. If `Dtdisc_t.eventf` is not `NULL`, it will be called as `(*eventf)(dt,e,v,disc)` to announce an event `e` of type `int` and a corresponding value `v` of type `Void_t*`. A negative return value from `eventf` causes the on-going operation to terminate. The available events are:

- `DT_OPEN`: `dt` is being opened. A positive return value from `eventf` indicates that `dt` shares memory with another dictionary, say `odt`. In this case, `eventf` should also return in `*(Void_t**)v` the dictionary management area of `odt` (see the `memoryf` discussion above) so that `dt` can be initialized with it.
- `DT_CLOSE`: `dt` is being closed.
- `DT_DISC`: The current discipline is being changed to a new discipline `(Dtdisc_t*)v`. If a dictionary does not want change disciplines, it can prevent this by having an event function returning a negative value on this event.
- `DT_METH`: The current method is being changed to a new method `(Dtmethod_t*)v`. As with disciplines, if a dictionary does not wish its method to be change, it can install an event function that returns a negative value on this event.

2.3.2 Discipline changing

Treatment of objects can be dynamically changed by changing the disciplines that describe them. For example, suppose that the token counting application of Figure 2 requires tokens to be output in order of their frequencies. Figure 4 shows the necessary code. Lines 1-6 define a function `fcompare()`

```
 1. int fcompare(Dt_t* dt, Void_t* t1, Void_t* t2, Dtdisc_t* disc)
 2. { int  d;
 3.   if((d = ((Token_t*)t1)->freq - ((Token_t*)t2)->freq) != 0)
 4.        return d;
 5.   else return strcmp(((Token_t*)t1)->name,((Token_t*)t2)->name);
 6. }

 7. Tkdisc.key = Tkdisc.size = 0;
 8. Tkdisc.comparf = fcompare;
 9. dtdisc(dt,&Tkdisc,DT_SAMECMP|DT_SAMEHASH);
10. dtmethod(dt,Dtoset);
```

Figure 4: Order tokens by frequency

to compare tokens first by frequencies, then by names. Note that `fcompare()` requires entire token structures, not just the `Token_t.name` fields. Lines 7-10 define code to be inserted before the output loop on lines `26-27`. `Tkdisc.key` and `Tkdisc.size` are set to 0 so that `Token_t` objects will be passed to `fcompare()` per its requirement. The `dtdisc()` call on line `9` announces the discipline change. The flag `DT_SAMECMP` indicates that objects remain distinct with the new comparator. Thus, no check for new duplicates will be performed. Similarly, the flag `DT_SAMEHASH` indicates that hash values for objects remain the same so objects will not be rehashed. These two flags save computation that would have been wasted anyway in the subsequent call `dtmethod(dt,Dtoset)` on line `10` which changes the storage method to `Dtoset` and reorders all objects.

2.4 Object and dictionary sharing

With proper use of disciplines, objects can be shared between dictionaries. Further, such dictionaries may even be in different processes.

2.4.1 Sharing objects among dictionaries

As an example of object sharing, consider an event processing system in which events arrive randomly but are processed in a priority order. Priorities of events are allowed to be updated dynamically. Figure 5 shows how to do this with two dictionaries, `pdt`, for processing events by priorities and `edt`, for processing events by values.

Lines `1-6` define the event type. Since the same event must appear in two different dictionaries, two object holder fields `elink` and `plink` are defined to be used respectively in the two dictionaries, `pdt` and `edt`. Though the link fields could have been left out so that the dictionaries will allocate holders separately from the event objects, doing it this way helps save memory allocation overhead.

Lines `7-21` define disciplines and discipline functions. Discipline `Pdisc` of `pdt` uses the comparator `pcompare()` to order events first by reverse priority values then by event values. In this way, events

```
typedef struct
{ int      e;      /* event type    */
  int      p;      /* priority      */
  Dtlink_t elink;  /* holder in edt */
  Dtlink_t plink;  /* holder in pdt */
} Event_t;

int pcompare(Dt_t* dt, Void_t* e1, Void_t* e2, Dtdisc_t* disc)
{ if(((Event_t*)e2)->p != ((Event_t*)e1)->p)
        return ((Event_t*)e2)->p - ((Event_t*)e1)->p;
  else  return ((Event_t*)e1)->e - ((Event_t*)e2)->e;
}

Void_t* emake(Dt_t* dt, Void_t* e, Dtdisc_t* disc)
{ Event_t* newe = malloc(sizeof(Event_t));
  newe->p = ((Event_t*)e)->p;
  newe->e = ((Event_t*)e)->e;
  return (Void_t*)newe;
}

void efree(Dt_t* dt, Void_t* e, Dtdisc_t* disc)
{ free(e); }

Dtdisc_t Pdisc = { 0, 0, offsetof(Event_t,plink), 0, 0, pcompare };
Dtdisc_t Edisc = { 0, sizeof(int), offsetof(Event_t,elink), emake, efree };
...

Dt_t*  edt = dtopen(&Edisc,Dtset);
Dt_t*  pdt = dtopen(&Pdisc,Dtoset);
for(;;)
{ Event_t  *es, *e = readevent();
  if((es = dtsearch(edt,e)) )
  {    es = dtsearch(pdt,es);
       es->p = e->p;
       dtrenew(pdt,es);
  }
  else dtinsert(pdt,dtinsert(edt,e));
  if((e = dtfirst(pdt)) )
  { ...process event e...
       dtdelete(pdt,e);
       dtdelete(edt,e);
  }
}
```

Figure 5: Sharing event objects between two dictionaries

with higher priorities will be processed first. Discipline `Edisc` of `edt` specifies the value of an event as its key. Objects are allocated and freed only in `edt` via the functions `emake()` and `efree()`.

Lines `22-23` show dictionary creation. Dictionary `pdt` uses method `Dtoset` because it needs to prioritize events. Dictionary `edt` uses method `Dtset` for fast search.

Lines `24-37` show the event processing loop. Line `25` reads an event `e`. Lines `26-30` check to see there is an event `es` that matches `e`. If so, the priority of `es` is updated and `dtrenew(pdt,es)` is called to update its position in `pdt`. Otherwise, line `31` adds a new event. Note that event insertion must be done first in `edt` because the new event structure is allocated there.

Lines `32-36` process the highest priority event and deletes it from the event set.

2.4.2 Sharing dictionaries across processes

In the above example, `edt` and `pdt` can be shared from different processes via shared memory. Doing this requires a discipline function `memoryf` that allocates from shared memory regions with the same base addresses across processes (because objects are accessed via pointers), and an event handler `eventf` that takes special action on the event `DT_OPEN`.

Figure 6 shows the definition of an event handler `evhandler()`. On line `3`, `evhandler()` uses `shared_managing_area(disc)` (some application-specific function) on the event `DT_OPEN` to see if there is already a shared managing area (Section 2.3.1). In that case, `evhandler()` sets `*v` to point to this area then returns with a value `1`. On receiving a positive return value from the event handler, `dtopen()` uses the given data to reinitialize the new dictionary instance. If no shared management area exists, `evhandler()` returns `0`, causing `dtopen()` to call `disc->memoryf` to create a new management area.

Also shown in Figure 6 are the new definitions of the disciplines `Pdisc` and `Edisc` and discipline functions `emake()` and `efree()` based on `memoryf`. A subtlety here is that the shared managing area of the collective dictionary `edt` must be separate from that of the collective dictionary `pdt` because `edt` and `pdt` are separate dictionaries. This means that `Edisc` and `Pdisc` must be extended to keep private data about their managing areas. Below is a common technique to extend `Dtdisc_t` to an extended type `Evdisc_t` suitable for this example. C casting rules allow pointers to `Dtdisc_t` and `Evdisc_t` to be interchangeable.

```
typedef struct
{ Dtdisc_t disc;
  ... data for managing area ...
} Evdisc_t;
```

The above examples show how Cdt supports sharing objects among dictionaries and sharing dictionaries among processes. Left undiscussed are the important issues of how to allocate shared memory keeping the same virtual addresses and how to manage concurrency. Both of these issues are beyond the scope of Cdt but there are relevant tools such as Vmalloc [11] for generalized memory allocation, and standard operating system facilities (e.g., semaphores) for negotiating safe concurrent accesses to shared memory.

```
int evhandler(Dt_t* dt, int ev, Void_t* v, Dtdisc_t* disc)
{ Void_t*   sma;
  if(ev == DT_OPEN && (sma = shared_managing_area(disc)) )
  {    *((Void_t**)v) = sma;
       return 1;
  }
  else return 0;
}

Void_t* emake(Dt_t* dt, Void_t* e, Dtdisc_t* disc)
{ Event_t* newe;
  newe = (*disc->memoryf)(dt,(Void_t*)0,sizeof(Event_t),disc);
  newe->p = ((Event_t*)e)->p;
  newe->e = ((Event_t*)e)->e;
  return (Void_t*)newe;
}

void efree(Dt_t* dt, Void_t* e, Dtdisc_t* disc)
{ (*disc->memoryf)(dt,e,0,disc);
}

Evdisc_t Pdisc = { { 0, 0, offsetof(Event_t,plink),
                     (Dtmake_f)0, (Dtfree_f)0, pcompare,
                     (Dthash_f)0, allocator, evhandler
                   }
                   ...Pdisc shared memory data...
                 };
Evdisc_t Edisc = { { 0, sizeof(int), offsetof(Event_t,elink),
                     emake, efree, (Dtcompar_f)0, (Dthash_f)0,
                     allocator, evhandler
                   }
                   ...Edisc shared memory data...
                 };
```

Figure 6: Disciplines for sharing event dictionaries

3 Performance

Among the various container data types, ordered and unordered sets are most common and also have most variation in implementation quality. This section presents results from a performance study that compared various set and map container data type packages.

3.1 Methodology

The token counting application in Figure 2 was used as a benchmark. To minimize implementation variation, a single program was written with compile time options to switch implementations based on the Cdt methods `Dtoset` and `Dtset`, the Unix/C package `tsearch`, the C++ classes `Set` and `Map`, and the STL templates `map` and `hashmap` (`ftp://butler.hpl.hp.com/stl`, release 10/31/1995). For uniformity, all implementations used the same string comparison function and, for hash table methods, the same hash function.

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 most input files appear more or less in random order. However, tokens in *host* and *city* are highly ordered. Thus, this set of input data provides a realistic testbed for checking the performance of the various packages.

Program	Size
`hashmap`	179,988
`Set`	146,632
`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 OS5.4. The Cdt version of the test program combined `Dtset` and `Dtoset` with invocation switches for method selection. Table 2 shows executable code sizes of various versions. The `hashmap`, `map` and `Set` versions were about twice as large as the others. Compiling `main(){}` with C and C++ showed that C++ compilation only added about 5K to the resulting executable code. Thus, the `hashmap`, `map` and `Set` versions were large because of the implementation of these packages and not the language.

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 Unordered set packages

The container packages for unordered sets and maps considered here were:

- `hashmap`: The STL `hashmap` template. This uses hash tables with collision chains.
- `Set`: The `Set` class. This uses hash tables with chaining to resolve collisions.
- `Dtset`: Method `Dtset` of Cdt. This uses hash tables with move-to-front collision chains.

Package	*ps*	*src*	*kjv*	*mbox*	*city*	*host*
`hashmap`	681,802	565,586	4,037,962	1,839,234	291,828	2,279,318
`Set`	328,199	130,870	799,337	385,711	35,410	384,460
`Dtset`	324,085	122,085	788,681	369,597	11,602	346,992

Table 3: Comparison counts of unordered set packages

Table 3 shows comparison counts. `hashmap` was worst, with comparison counts many times higher than that of `Set` and `Dtset`. `Dtset` exploited the assertion that equal tokens must have the same hash values to minimize comparisons. Judging from its low comparison counts, `Set` might have used the same strategy as `Dtset`. However, `Dtset` still did better, thanks to its move-to-front collision chains.

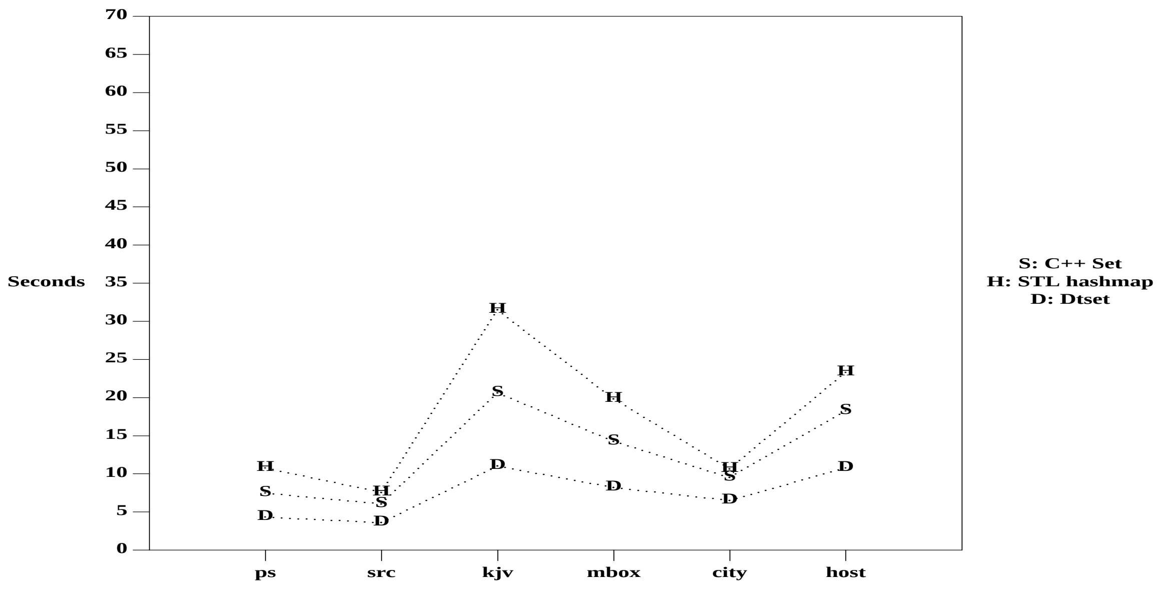

Figure 7: Time performance of unordered set packages

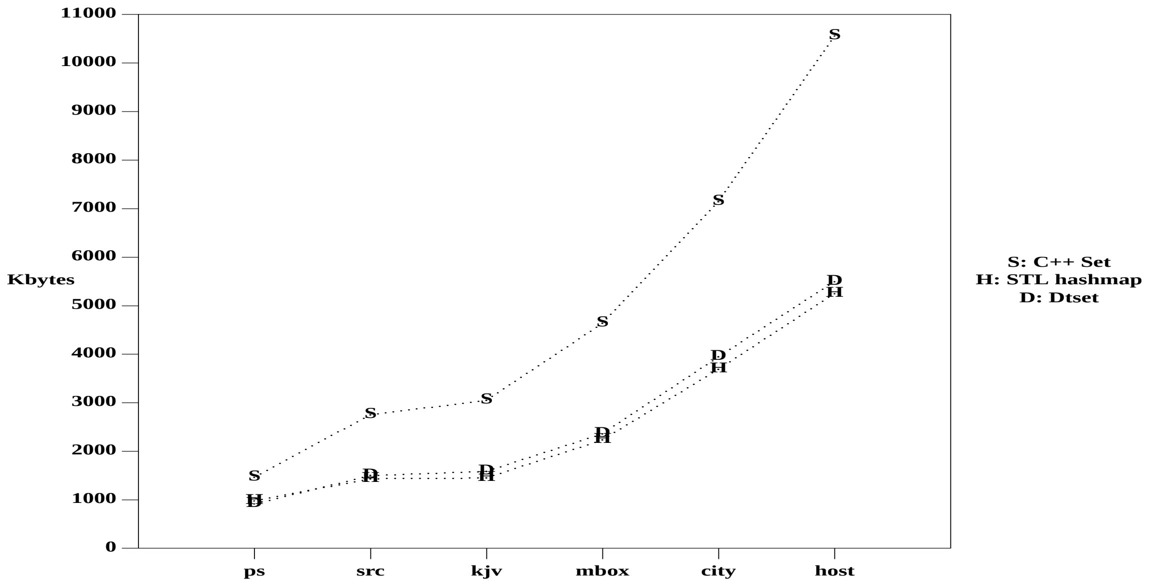

Figure 8: Space usage of unordered set packages

Figure 7 shows time performance for the unordered set packages. Poor comparison counts directly translated to poor computing time. `hashmap` was worst, sometimes up to a factor of 3 slower than `Dtset`. `Dtset` was fastest among the three packages.

Figure 8 shows space usage. `Dtset` and `hashmap` used about the same amount of space. `Set` often used twice as much space as the other packages.

3.3 Ordered set packages

As discussed in Section 2.2.2, the token counting example may require tokens to be output in order. A natural solution is to use container packages that maintain ordered tokens. The container packages for ordered sets and maps considered here were:

- `Map`: The `Map` class. This uses AVL balanced trees.
- `map`: The STL `map` template. This uses red-black balanced trees.
- `tsearch`: The `tsearch` function in SUN OS 5.4. This uses plain binary trees.
- `Dtoset`: Method `Dtoset` of Cdt. This uses top-down splay trees.

Package	*ps*	*src*	*kjv*	*mbox*	*city*	*host*
`Map`	14,267,464	5,857,168	28,141,722	15,746,680	3,840,718	22,963,206
`map`	10,434,097	4,553,264	26,118,542	13,229,379	3,006,238	15,657,198
`tsearch`	7,313,655	3,532,762	13,525,549	7,396,212	12,189,319	19,749,995
`Dtoset`	1,636,228	1,773,115	8,592,385	5,424,683	1,630,525	2,567,979

Table 4: Comparison counts for ordered set packages

Table 4 shows comparison counts. Except for *city* and *host*, `tsearch` performed well despite its simplistic data structure. This was because tokens appeared randomly in most datasets so the constructed binary trees were naturally balanced. By contrast, tokens were highly ordered in *city* and *host* so the corresponding trees were skewed causing bad performance for `tsearch`. The balanced tree packages `Map` and `map` ignored any ordering property in the data and used about the same number of comparisons with `map` having a slight edge. The splay tree approach in `Dtoset` adapted well to data biases and helped `Dtoset` to win in all cases.

Figure 9 shows time performance for the ordered set packages. `map` was sometimes slower than `Map` even though it had better comparison counts, perhaps due to some hidden bookkeeping costs in `map`. `Dtoset` was fastest among the ordered set packages, sometimes by a factor of two or three. In fact, comparing with Figure 7 showed that `Dtoset` was only slower than `Dtset` and was comparable with or faster than the other unordered set packages.

Figure 10 shows memory usage. `tsearch` and `Map` used more memory than other packages. `Map`'s extra space was due to the balancing data. The cause for `tsearch`'s poor memory usage was unclear

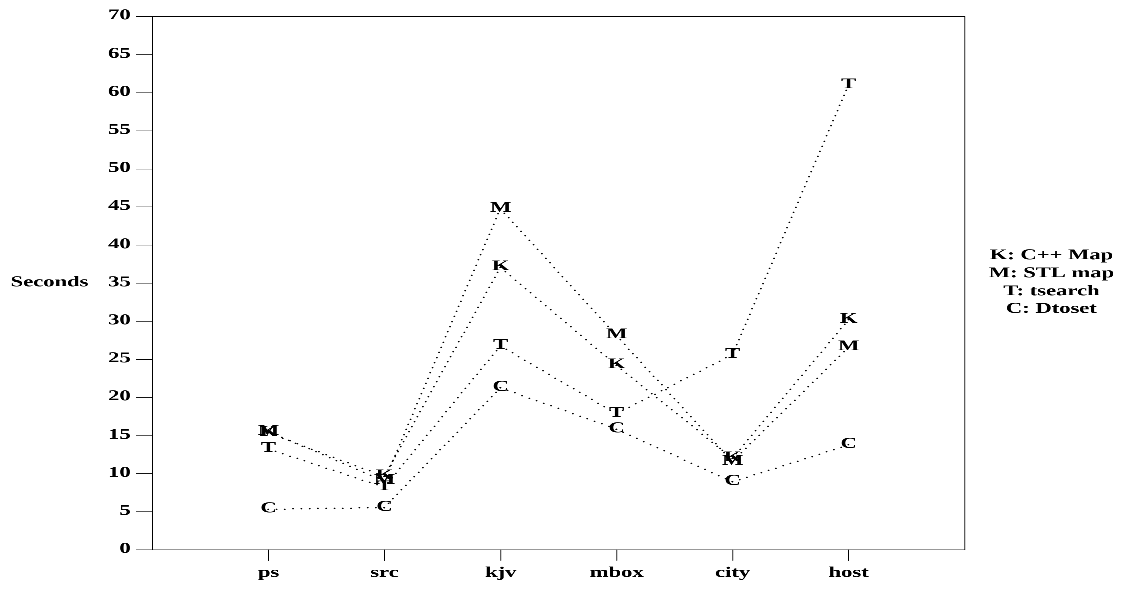

Figure 9: Time performance of ordered set packages

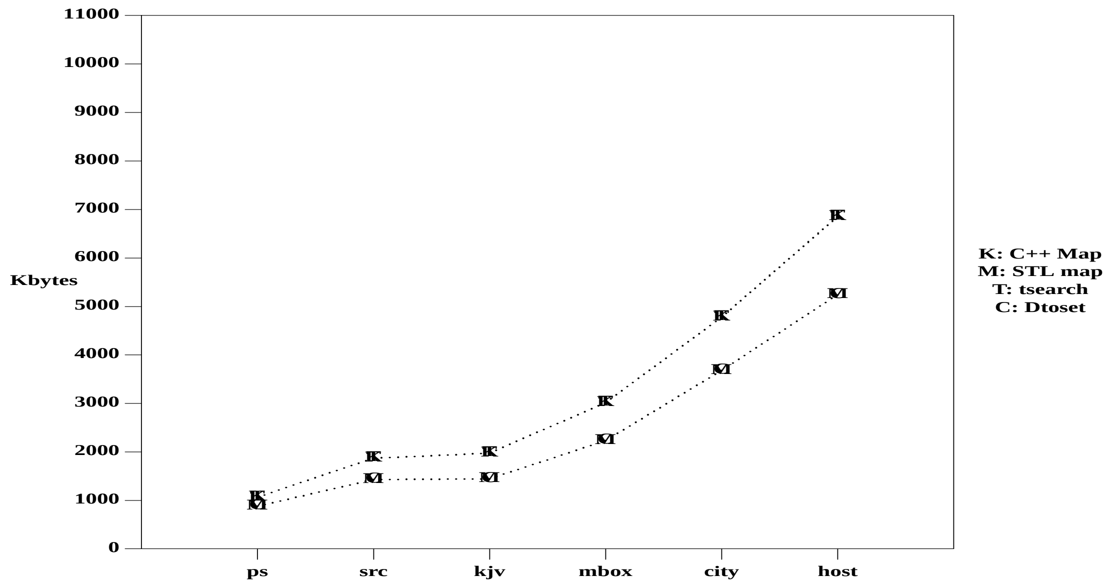

Figure 10: Space usage of ordered set packages

although a memory trace using Vmalloc revealed a mysterious extra allocation after each allocation of a holder. Finally, comparing Figure 8 and Figure 10 revealed that the Cdt hash-based method `Dtset` only used slightly more space than the tree-based methods `Dtoset` and `map`. In fact, it used less space than `tsearch` and `Map`.

3.4 Performance tuning with Cdt

The Cdt program used in the discussed benchmark tests was deliberately left suboptimal so that a fair comparison in terms of memory allocation could be made with the other packages. Cdt discipline could be used to further tune the code to reduce space usage and improve execution time.

```
typedef struct
{ Dtlink_t link;
  int      freq;
  char     name[1];
} Token_t;

Dtdisc_t   Tkdisc = { offsetof(Token_t,name), 0, 0};

Token_t* newtoken(char* s)
{ Token_t* tk;
  tk = malloc(sizeof(Token_t)+strlen(s));
  strcpy(tk->name,s);
  tk->freq = 1;
}
```

Figure 11: Discipline to reduce memory allocation for token counting

Figure 11 shows a definition of `Token_t` and a corresponding `newtoken()` function that requires only a single `malloc()` call to allocate both the `Token_t` structure and the token name. This avoids `malloc()` headers and reduces fragmentation. Note that `Tkdisc.size` is set to `0` to indicate that `Token_t.name` is now a null-terminated array of characters and not a null-terminated string as in Figure 2. Lines `3-16` of Figure 2 should be replaced by Figure 11 for this optimization.

Tables 5.a and 5.b compare the code in Figure 2 (`Dtoset` and `Dtset`) and the new code (`Dtoset+` and `Dtset+`). Though time saving is small, space saving can be up to 20%. This is significant in modern information systems that routinely manipulate gigabytes of data.

Matching methods and computational needs can significantly enhance performance. Consider the problem of printing tokens in order as discussed in Section 2.2.2. For a dataset with many repeated tokens, a combined strategy, `Dtset+Dtoset`, of constructing the token dictionary with `Dtset` and sorting it with `Dtoset` before printing can improve performance over the sole use of `Dtoset` because `Dtset` is faster than `Dtoset`. Table 6.a shows that `Dtset+Dtoset` substantially improves comparison counts over the sole use of `Dtoset`, up to 70% for *kjv*. Except for *city* and *host* which contain mostly distinct tokens, Table 6.b shows that time measurements improved accordingly.

Package	*ps*	*src*	*kjv*	*mbox*	*city*	*host*
`Dtoset`	880	1,432	1,440	2,232	3,680	5,240
`Dtoset+`	744	1,184	1,136	1,768	3,096	4,360
`Dtset`	912	1,496	1,584	2,360	3,944	5,496
`Dtset+`	776	1,248	1,296	1,896	3,352	4,616

a. Space usage (in K-bytes)

Package	*ps*	*src*	*kjv*	*mbox*	*city*	*host*
`Dtset`	4.32	3.59	11.01	8.20	6.49	10.78
`Dtset+`	4.23	3.32	10.54	7.58	6.23	10.24
`Dtoset`	5.33	5.53	21.27	15.85	8.92	13.76
`Dtoset+`	5.14	5.13	19.01	14.05	8.07	12.72

b. Cpu+system times (in seconds)

Table 5: Performance when disciplines are used to tailor object types

Package	*ps*	*src*	*kjv*	*mbox*	*city*	*host*
`Dtoset`	1,636,228	1,773,115	8,592,385	5,424,683	1,630,525	2,567,979
`Dtset+Dtoset`	498,090	608,173	1,307,352	1,282,181	1,420,223	2,013,692

a. Comparison counts

Package	*ps*	*src*	*kjv*	*mbox*	*city*	*host*
`Dtoset`	5.33	5.53	21.27	15.85	8.92	13.76
`Dtset+Dtoset`	4.83	5.12	12.50	11.55	12.73	18.38

b. Cpu+system times (in seconds)

Table 6: Performance when methods are matched to usage contexts

Performance variation due to input data as shown in Table 6.b is typical in practical applications. Input data often come with a variety of special characteristics. Rarely can efficient algorithms be devised that adapt smoothly and efficiently to data diversity. Thus, a good design principle is to implement the best default method of computation but, if possible, also let users have a choice in picking and combining methods to optimize based on specific knowledge of input data. Cdt enables applications to do this in the context of using container data types. In fact, to facilitate data collection, the benchmark program was written to allow strategy selection at invocation time. It is not easy to do the same with the other container packages.

4 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. Based on a method and discipline architecture, Cdt achieves an interface that keeps orthogonal the three main 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 for containers, including the C++ Standard Template Libraries.

Contemporary container libraries tend to tie interfaces closely to implementation methods. 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 templates, the close tie between implementation techniques and abstract interfaces can reduce generality. 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 the minimal object requirements of the respective underlying data structures. 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 convert a container from a `hashmap` to a `map` (and vice versa) in the style discussed in Section 3.4. Cdt avoids such interface limitations by making the dictionary type and accompanying dictionary operations completely abstract and parameterizable by methods and disciplines, which are orthogonal and dynamically mutable dictionary attributes.

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 and provides 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 or hash functions. The loss of static type checking is balanced by the added programming flexibility as shown throughout the paper. In particular, examples showed how disciplines can be used to share objects in different dictionaries based on distinct object semantics. Further, dictionaries can be built in shared or persistent memory, given some suitable memory allocation method. The efficiency loss resulting 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 enables Cdt to be used as a dynamically loadable shared library. In any case, for applications such as the discussed token counting example that 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 `Dtset` and `Dtoset` performed as well as or better than their counterparts in other C and C++ container libraries. Cdt methods consistently used about the same or less space than other packages while they could be faster by a factor of two or three depending on the particular packages. The use of splay trees and hash tables with move-to-front collision chains as implementation data structures enabled Cdt ordered set and unordered set methods to perform well in a wide range of input data. Among all studied container packages, the Cdt unordered set method `Dtset` was fastest, followed by Cdt ordered set method `Dtoset`. Aside from better data structure implementation, the flexible Cdt interface allows performance tuning at a level beyond other packages. Examples were given showing how significant performance gains can be made with selective matching of disciplines and methods to usage contexts.

Cdt is a descendant of Libdict [12] . It is a mature library and has been used in many applications ranging from language processors and graph manipulation systems to large-scale information processing and event handling communication systems. Some of these applications routinely manipulate dictionaries with hundreds of thousands of objects.

Acknowledgement

Cdt evolved over many years and benefited from advice 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.

Code availability

Cdt source code can be obtained from `http://www.research.att.com/sw/tools/cdt/`.

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

[3] T. Papadakis. *Skip Lists and Probabilistic Analysis of Algorithms.* Uni. of Waterloo, 1993.

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

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

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

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

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

[9] 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.

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

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

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