NAME

Cdt − container data types

SYNOPSIS

```
#include <graphviz/cdt.h>
```

DICTIONARY TYPES

```
Void_t*;
Dt_t;
Dtdisc_t;
Dtmethod_t;
Dtlink_t;
Dtstat_t;
```

DICTIONARY CONTROL

```
Dt_t*       dtopen(Dtdisc_t* disc, Dtmethod_t* meth);
int         dtclose(Dt_t* dt);
void        dtclear(dt);
Dtmethod_t* dtmethod(Dt_t* dt, Dtmethod_t* meth);
Dtdisc_t*   dtdisc(Dt_t* dt, Dtdisc_t* disc, int type);
Dt_t*       dtview(Dt_t* dt, Dt_t* view);
```

STORAGE METHODS

```
Dtmethod_t* Dtset;
Dtmethod_t* Dtbag;
Dtmethod_t* Dtoset;
Dtmethod_t* Dtobag;
Dtmethod_t* Dtlist;
Dtmethod_t* Dtstack;
Dtmethod_t* Dtqueue;
```

DISCIPLINE

```
typedef Void_t*      (*Dtmake_f)(Dt_t*, Void_t*, Dtdisc_t*);
typedef void         (*Dtfree_f)(Dt_t*, Void_t*, Dtdisc_t*);
typedef int          (*Dtcompar_f)(Dt_t*, Void_t*, Void_t*, Dtdisc_t*);
typedef unsigned int (*Dthash_f)(Dt_t*, Void_t*, Dtdisc_t*);
typedef Void_t*      (*Dtmemory_f)(Dt_t*, Void_t*, size_t, Dtdisc_t*);
typedef int          (*Dtevent_f)(Dt_t*, int, Void_t*, Dtdisc_t*);
```

OBJECT OPERATIONS

```
Void_t*   dtinsert(Dt_t* dt, Void_t* obj);
Void_t*   dtdelete(Dt_t* dt, Void_t* obj);
Void_t*   dtsearch(Dt_t* dt, Void_t* obj);
Void_t*   dtmatch(Dt_t* dt, Void_t* key);
Void_t*   dtfirst(Dt_t* dt);
Void_t*   dtnext(Dt_t* dt, Void_t* obj);
Void_t*   dtlast(Dt_t* dt);
Void_t*   dtprev(Dt_t* dt, Void_t* obj);
Void_t*   dtfinger(Dt_t* dt);
Void_t*   dtrenew(Dt_t* dt, Void_t* obj);
int       dtwalk(Dt_t* dt, int (*userf)(Dt_t*, Void_t*, Void_t*), Void_t*);
Dtlink_t* dtflatten(Dt_t* dt);
Dtlink_t* dtlink(Dt_t*, Dtlink_t* link);
Void_t*   dtobj(Dt_t* dt, Dtlink_t* link);
Dtlink_t* dtextract(Dt_t* dt);
int       dtrestore(Dt_t* dt, Dtlink_t* link);
```

DICTIONARY STATUS

```
Dt_t*     dtvnext(Dt_t* dt);
int       dtvcount(Dt_t* dt);
Dt_t*     dtvhere(Dt_t* dt);
int       dtsize(Dt_t* dt);
int       dtstat(Dt_t* dt, Dtstat_t*, int all);
```

HASH FUNCTIONS

```
unsigned int dtstrhash(unsigned int h, char* str, int n);
unsigned int dtcharhash(unsigned int h, unsigned char c);
```

DESCRIPTION

Cdt manages run-time dictionaries using standard container data types: unordered set/multiset, ordered set/multiset, list, stack, and queue.

DICTIONARY TYPES

Void_t*

This type is used to pass objects between *Cdt* and application code. Void_t is defined as void for ANSI-C and C++ and char for other compilation environments.

Dt_t

This is the type of a dictionary handle.

Dtdisc_t

This defines the type of a discipline structure which describes object lay-out and manipulation functions.

Dtmethod_t

This defines the type of a container method.

Dtlink_t

This is the type of a dictionary object holder (see dtdisc().)

Dtstat_t

This is the type of a structure to return dictionary statistics (see dtstat().)

DICTIONARY CONTROL

Dt_t* dtopen(Dtdisc_t* disc, Dtmethod_t* meth)

This creates a new dictionary. disc is a discipline structure to describe object format. meth specifies a manipulation method. dtopen() returns the new dictionary or NULL on error.

int dtclose(Dt_t* dt)

This deletes dt and its objects. Note that dtclose() fails if dt is being viewed by some other dictionaries (see dtview()). dtclose() returns 0 on success and -1 on error.

void dtclear(Dt_t* dt)

This deletes all objects in dt without closing dt.

Dtmethod_t dtmethod(Dt_t* dt, Dtmethod_t* meth)

If meth is NULL, dtmethod() returns the current method. Otherwise, it changes the storage method of dt to meth. Object order remains the same during a method switch among Dtlist, Dtstack and Dtqueue. Switching to and from Dtset/Dtbag and Dtoset/Dtobag may cause objects to be rehashed, reordered, or removed as the case requires. dtmethod() returns the previous method or NULL on error.

Dtdisc_t* dtdisc(Dt_t* dt, Dtdisc_t* disc, int type)

If disc is NULL, dtdisc() returns the current discipline. Otherwise, it changes the discipline of dt to disc. Objects may be rehashed, reordered, or removed as appropriate. type can be any bit combination of DT_SAMECMP and DT_SAMEHASH. DT_SAMECMP means that objects will compare exactly the same as before thus obviating the need for reordering or removing new duplicates. DT_SAMEHASH

means that hash values of objects remain the same thus obviating the need to rehash. dtdisc() returns the previous discipline on success and NULL on error.

Dt_t* dtview(Dt_t* dt, Dt_t* view)

A viewpath allows a search or walk starting from a dictionary to continue to another. dtview() first terminates any current view from dt to another dictionary. Then, if view is NULL, dtview returns the terminated view dictionary. If view is not NULL, a viewpath from dt to view is established. dtview() returns dt on success and NULL on error.

If two dictionaries on the same viewpath have the same values for the discipline fields Dtdisc_t.link, Dtdisc_t.key, Dtdisc_t.size, and Dtdisc_t.hashf, it is expected that key hashing will be the same. If not, undefined behaviors may result during a search or a walk.

STORAGE METHODS

Storage methods are of type Dtmethod_t*. *Cdt* supports the following methods:

Dtoset
Dtobag

Objects are ordered by comparisons. Dtoset keeps unique objects. Dtobag allows repeatable objects.

Dtset
Dtbag

Objects are unordered. Dtset keeps unique objects. Dtbag allows repeatable objects and always keeps them together (note the effect on dictionary walking.)

Dtlist

Objects are kept in a list. New objects are inserted either in front of *current object* (see dtfinger()) if this is defined or at list front if there is no current object.

Dtstack

Objects are kept in a stack, i.e., in reverse order of insertion. Thus, the last object inserted is at stack top and will be the first to be deleted.

Dtqueue

Objects are kept in a queue, i.e., in order of insertion. Thus, the first object inserted is at queue head and will be the first to be deleted.

DISCIPLINE

Object format and associated management functions are defined in the type Dtdisc_t:

```
typedef struct
{ int       key, size;
  int       link;
  Dtmake_f  makef;
  Dtfree_f  freef;
  Dtcompar_f comparf;
  Dthash_f  hashf;
  Dtmemory_f memoryf;
  Dtevent_f  eventf;
} Dtdisc_t;
```

int key, size

Each object obj is identified by a key used for object comparison or hashing. key should be non-negative and defines an offset into obj. If size is negative, the key is a null-terminated string with starting address *(Void_t**)((char*)obj+key). If size is zero, the key is a null-terminated string with starting address (Void_t*)((char*)obj+key). Finally, if size is positive, the key is a byte array of length size starting at (Void_t*)((char*)obj+key).

int link
Let obj be an object to be inserted into dt as discussed below. If link is negative, an internally allocated object holder is used to hold obj. Otherwise, obj should have a Dtlink_t structure embedded link bytes into it, i.e., at address (Dtlink_t*)((char*)obj+link).

Void_t* (*makef)(Dt_t* dt, Void_t* obj, Dtdisc_t* disc)
If makef is not NULL, dtinsert(dt,obj) will call it to make a copy of obj suitable for insertion into dt. If makef is NULL, obj itself will be inserted into dt.

void (*freef)(Dt_t* dt, Void_t* obj, Dtdisc_t* disc)
If not NULL, freef is used to destroy data associated with obj.

int (*comparf)(Dt_t* dt, Void_t* key1, Void_t* key2, Dtdisc_t* disc)
If not NULL, comparf is used to compare two keys. Its return value should be <0, =0, or >0 to indicate whether key1 is smaller, equal to, or larger than key2. All three values are significant for method Dtoset and Dtobag. For other methods, a zero value indicates equality and a non-zero value indicates inequality. If (*comparf)() is NULL, an internal function is used to compare the keys as defined by the Dtdisc_t.size field.

unsigned int (*hashf)(Dt_t* dt, Void_t* key, Dtdisc_t* disc)
If not NULL, hashf is used to compute the hash value of key. It is required that keys compared equal will also have same hash values. If hashf is NULL, an internal function is used to hash the key as defined by the Dtdisc_t.size field.

Void_t* (*memoryf)(Dt_t* dt, Void_t* addr, size_t size, Dtdisc_t* disc)
If not NULL, memoryf is used to allocate and free memory. When addr is NULL, a memory segment of size size is requested. If addr is not NULL and size is zero, addr is to be freed. If addr is not NULL and size is positive, addr is to be resized to the given size. If memoryf is NULL, *malloc(3)* is used. When dictionaries share memory, a record of the first allocated memory segment should be kept so that it can be used to initialize new dictionaries (see below.)

int (*eventf)(Dt_t* dt, int type, Void_t* data, Dtdisc_t* disc)
If not NULL, eventf announces various events. If it returns a negative value, the calling operation will terminate with failure. Unless noted otherwise, a non-negative return value let the calling function proceed normally. Following are the events:

DT_OPEN:
dt is being opened. If eventf returns zero, the opening process proceeds normally. A positive return value indicates that dt uses memory already initialized by a different dictionary. In that case, *(Void_t**)data should be set to the first allocated memory segment as discussed in memoryf. dtopen() may fail if this segment is not returned or if it has not been properly initialized.

DT_CLOSE:
dt is being closed.

DT_DISC:
The discipline of dt is being changed to a new one given in (Dtdisc_t*)data.

DT_METH:
The method of dt is being changed to a new one given in (Dtmethod_t*)data.

OBJECT OPERATIONS

Void_t* dtinsert(Dt_t* dt, Void_t* obj)
This inserts an object prototyped by obj into dt. If there is an existing object in dt matching obj and the storage method is Dtset or Dtoset, dtinsert() will simply return the matching object. Otherwise, a new object is inserted according to the method in use. See Dtdisc_t.makef for object construction. dtinsert() returns the new object, a matching object as noted, or NULL on error.

Void_t* dtdelete(Dt_t* dt, Void_t* obj)

If obj is not NULL, the first object matching it is deleted. If obj is NULL, methods Dtstack and Dtqueue delete respectively stack top or queue head while other methods do nothing. See Dtdisc_t.freef for object destruction. dtdelete() returns the deleted object (even if it was deallocated) or NULL on error.

Void_t* dtsearch(Dt_t* dt, Void_t* obj)
Void_t* dtmatch(Dt_t* dt, Void_t* key)

These functions find an object matching obj or key either from dt or from some dictionary accessible from dt via a viewpath (see dtview().) dtsearch() and dtmatch() return the matching object or NULL on failure.

Void_t* dtfirst(Dt_t* dt)
Void_t* dtnext(Dt_t* dt, Void_t* obj)

dtfirst() returns the first object in dt. dtnext() returns the object following obj. Objects are ordered based on the storage method in use. For Dtoset and Dtobag, objects are ordered by object comparisons. For Dtstack, objects are ordered in reverse order of insertion. For Dtqueue, objects are ordered in order of insertion. For Dtlist, objects are ordered by list position. For Dtset and Dtbag, objects use some internal ordering which may change on any search, insert, or delete operations. Therefore, these operations should not be used during a walk on a dictionary using either Dtset or Dtbag.

Objects in a dictionary or a viewpath can be walked using a for(;;) loop as below. Note that only one loop can be used at a time per dictionary. Concurrent or nested loops may result in unexpected behaviors.

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

Void_t* dtlast(Dt_t* dt)
Void_t* dtprev(Dt_t* dt, Void_t* obj)

dtlast() and dtprev() are like dtfirst() and dtnext() but work in reverse order. Note that dictionaries on a viewpath are still walked in order but objects in each dictionary are walked in reverse order.

Void_t* dtfinger(Dt_t* dt)

This function returns the *current object* of dt, if any. The current object is defined after a successful call to one of dtsearch(), dtmatch(), dtinsert(), dtfirst(), dtnext(), dtlast(), or dtprev(). As a side effect of this implementation of *Cdt*, when a dictionary is based on Dtoset and Dtobag, the current object is always defined and is the root of the tree.

Void_t* dtrenew(Dt_t* dt, Void_t* obj)

This function repositions and perhaps rehashes an object obj after its key has been changed. dtrenew() only works if obj is the current object (see dtfinger()).

dtwalk(Dt_t* dt, int (*userf)(Dt_t*, Void_t*, Void_t*), Void_t* data)

This function calls (*userf)(walk,obj,data) on each object in dt and other dictionaries viewable from it. walk is the dictionary containing obj. If userf() returns a <0 value, dtwalk() terminates and returns the same value. dtwalk() returns 0 on completion.

Dtlink_t* dtflatten(Dt_t* dt)
Dtlink_t* dtlink(Dt_t* dt, Dtlink_t* link)
Void_t* dtobj(Dt_t* dt, Dtlink_t* link)

Using dtfirst()/dtnext() or dtlast()/dtprev() to walk a single dictionary can incur significant cost due to function calls. For efficient walking of a single directory (i.e., no viewpathing), dtflatten() and dtlink() can be used. Objects in dt are made into a linked list and walked as follows:

```
for(link = dtflatten(dt); link; link = dtlink(dt,link) )
```

Note that dtflatten() returns a list of type Dtlink_t*, not Void_t*. That is, it returns a dictionary holder pointer, not a user object pointer (although both are the same if the discipline field link is non-negative.) The macro function dtlink() returns the dictionary holder object following link. The macro function dtobj(dt,link) returns the user object associated with link, Beware that the flattened object list is unflattened on any dictionary operations other than dtlink().

Dtlink_t* dtextract(Dt_t* dt)
int dtrestore(Dt_t* dt, Dtlink_t* link)

dtextract() extracts all objects from dt and makes it appear empty. dtrestore() repopulates dt with objects previously obtained via dtextract(). dtrestore() will fail if dt is not empty. These functions can be used to share a same dt handle among many sets of objects. They are useful to reduce dictionary overhead in an application that creates concurrently many dictionaries. It is important that the same discipline and method are in use at both extraction and restoration. Otherwise, undefined behaviors may result.

DICTIONARY INFORMATION

Dt_t* dtvnext(Dt_t* dt)

This returns the dictionary that dt is viewing, if any.

int dtvcount(Dt_t* dt)

This returns the number of dictionaries that view dt.

Dt_t* dtvhere(Dt_t* dt)

This returns the dictionary v viewable from dt where an object was found from the most recent search or walk operation.

int dtsize(Dt_t* dt)

This function returns the number of objects stored in dt.

int dtstat(Dt_t *dt, Dtstat_t* st, int all)

This function reports dictionary statistics. If all is non-zero, all fields of st are filled. Otherwise, only the dt_type and dt_size fields are filled. It returns 0 on success and -1 on error.

Dtstat_t contains the below fields:

int dt_type:
: This is one of DT_SET, DT_BAG, DT_OSET, DT_OBAG, DT_LIST, DT_STACK, and DT_QUEUE.

int dt_size:
: This contains the number of objects in the dictionary.

int dt_n:
: For Dtset and Dtbag, this is the number of non-empty chains in the hash table. For Dtoset and Dtobag, this is the deepest level in the tree (counting from zero.) Each level in the tree contains all nodes of equal distance from the root node. dt_n and the below two fields are undefined for other methods.

int dt_max:
: For Dtbag and Dtset, this is the size of a largest chain. For Dtoset and Dtobag, this is the size of a largest level.

int* dt_count:
: For Dtset and Dtbag, this is the list of counts for chains of particular sizes. For example, dt_count[1] is the number of chains of size 1. For Dtoset and Dtobag, this is the list of sizes of the levels. For example, dt_count[1] is the size of level 1.

HASH FUNCTIONS

unsigned int dtcharhash(unsigned int h, char c)
unsigned int dtstrhash(unsigned int h, char* str, int n)

These functions compute hash values from bytes or strings. dtcharhash() computes a new hash value from byte c and seed value h. dtstrhash() computes a new hash value from string str and seed value h. If n is positive, str is a byte array of length n; otherwise, str is a null-terminated string.

IMPLEMENTATION NOTES

Dtset and Dtbag are based on hash tables with move-to-front collision chains. Dtoset and Dtobag are based on top-down splay trees. Dtlist, Dtstack and Dtqueue are based on doubly linked list.

AUTHOR

Kiem-Phong Vo, kpv@research.att.com