Dynamic Abstract Data Types

Reusable data structures for C programs
Manual version $Revision: 2.63 $
$Date: 1996/08/20 00:11:30 $

Bob Glickstein

This manual describes Dynamic Abstract Data Types, a reusable software library providing dynamically-resizing data structures.

Introduction

This manual describes Dynamic Abstract Data Types, a reusable software library providing dynamically-resizing data structures for use in C system and application programming.

1 Glist

A Glist is a *grow list*. Like a stack, data can be added to and removed from the end. Like an array, elements in the middle can be accessed and changed. A Glist expands automatically to accommodate data as it is added.

A single Glist holds elements of a single type. It actually holds *copies* of those elements. When you hand a Glist a pointer to some data element, it copies that element according to the `sizeof` the data type into its private space. The accessor functions `glist_Nth` and `glist_Last` return pointers to the private copies of the data.

It is important to note that the pointers returned by `glist_Nth` and `glist_Last` are volatile; their contents may relocate if a subsequent `glist_Add` or `glist_Insert` causes a `realloc` of the Glist's private memory, rendering the returned pointers useless. Therefore, pointers into Glists should be used with care. If in doubt, don't save the pointer value, save the index of the element you care about, and use repeated calls to `glist_Nth`.

`void glist_Init` (*struct glist* *`gl`, *int* `eltsize`, *int* `growsize`) [Function]

> Initialize the `struct glist` pointed to by *gl*. The size of an element in the new Glist will be *eltsize*, and when the Glist is full, an attempt to add another element will cause the Glist to be expanded by *growsize* uninitialized entries.

`int glist_EmptyP` (*struct glist* *`gl`) [Macro]

> Yields zero if *gl* is a non-empty Glist, non-zero otherwise.

`int glist_Length` (*struct glist* *`gl`) [Macro]

> Yields the number of elements in *gl*.

`int glist_Add` (*struct glist* *`gl`, *const void* *`elt`) [Function]

> Adds a new value to the end of *gl*. The value is pointed to by *elt*, whose referent is copied into the Glist. The number of bytes to copy was given as *eltsize* in `glist_Init`. Returns the position of the newly-added element. If *elt* is 0, an empty element is added and is initialized with `bzero`.
>
> This function may raise the `strerror(ENOMEM)` exception.

`void glist_Insert` (*struct glist* *`gl`, *const void* *`elt`, *int* `n`) [Function]

> Insert an element into *gl*. The element to insert is pointed to by *elt*. The newly-added element will be located at position *n*. The old elements occupying locations *n* and higher are shifted to the right, resulting in the Glist growing by one. This is an O(n) operation. Legal values for *n* are in the range 0 to `glist_Length(gl)` inclusive. Note that
>
> ```
> glist_Insert(gl, elt, glist_Length(gl));
> ```
>
> is exactly equivalent to
>
> ```
> glist_Add(gl, elt);
> ```
>
> Glists are not optimized for mid-list insertion. If you find yourself using `glist_Insert` frequently, consider using a Dlist instead (see Chapter 2 [Dlist], page 5).
>
> If *elt* is 0, an empty element is inserted and initialized with `bzero`.
>
> This function may raise the `strerror(ENOMEM)` exception.

`void * glist_Last` (*struct glist* *`gl`) [Function]

Returns a pointer to the last element of *gl*.

`void * glist_Nth` (*struct glist* *`gl`, *int* `n`) [Function]

Returns a pointer to the element of *gl* whose position is *n*, where *n* lies between 0 and `glist_Length(gl) - 1`.

`int glist_Pop` (*struct glist* *`gl`) [Macro]

Shortens *gl* by one, discarding the last value in it. Yields the new number of values in the Glist. Do *not* call this macro on an empty Glist!

`void glist_Truncate` (*struct glist* *`gl`, *int* `num`) [Macro]

Truncates *gl* to contain *num* entries (by discarding all but entries 0 through *num* - 1). It is an error for *num* to be greater than `glist_Length(gl)`.

`void glist_Remove` (*struct glist* *`gl`, *int* `n`) [Function]

Removes the element from *gl* whose position is *n*. Elements in *gl* whose positions are higher than *n* are shifted to the left, shortening the Glist by one. This is an O(n) operation.

Glists are not optimized for mid-list removal. If you find yourself using `glist_Remove` frequently, consider using a Dlist instead (see Chapter 2 [Dlist], page 5).

`void glist_Set` (*struct glist* *`gl`, *int* `n`, *const void* *`elt`) [Function]

Destructively replace the element in *gl* at position *n* with the element pointed to by *elt*. If *n* is larger than the largest index in *gl*, then *gl* will automatically be extended to be exactly `n + 1` entries long; however, any intervening entries added by extending *gl* in this way remain completely uninitialized and will initially contain random garbage. If *elt* is 0, an empty element replaces the affected element and is initialized with `bzero`.

This function may raise the `strerror(ENOMEM)` exception.

`void glist_Sort` (*struct glist* *`gl`, *int* (*`compare`)(*const void* *, *const void* *)) [Function]

Sort the elements of *gl* in place according to the sorting predicate *compare* which, as in `qsort`, is a function taking pointers to two elements and returns a value less than, equal to, or greater than zero depending on whether its first argument is to be considered less than, equal to, or greater than its second argument.

`int glist_Bsearch` (*struct glist* *`gl`, *const void* *`probe`, *int* (*`compare`)(*const void* *, *const void* *)) [Function]

Perform binary search for an element of *gl* matching *probe*, using the comparison predicate *compare*. Finds a matching element in `O(log n)` time. The elements of *gl* must be sorted in ascending order with respect to the ordering implied by *compare*, which is a function taking pointers to two elements and returning a value less than, equal to, or greater than zero depending on whether its first argument is to be considered less than, equal to, or greater than its second argument.

If a matching element is found, this function returns its index within *gl*; otherwise -1 is returned.

`void glist_Swap` (*struct glist* *`gl`, *int* `m`, *int* `n`) [Function]

Swaps the two elements in *gl* whose positions are *m* and *n*.

`glist_FOREACH` (*struct glist* **gl*, *t*, *v*, *i*) [Macro]

Replaces the `for (...)` at the beginning of a loop that traverses the elements of a Glist. *gl* is a pointer to a Glist; *t* is the type of an element of the Glist; *v* is the name of a variable of type *t* `*`; and *i* is the name of an `int` variable. Within the body of the loop, *i* iterates from 0 through `glist_Length(gl) - 1`, and in each iteration, *v* points to the *i*th element of *gl*. Example:

```
int i;
struct foo *f;

glist_FOREACH(gl, struct foo, f, i) {
    printf("The bar field of element %d is %s\n", i, f->bar);
}
```

`void glist_Map` (*struct glist* **gl*, *void* (**fn*)(*void* *, *void* *), *void* **data*) [Function]

For each element of *gl*, invoke *fn*, passing a pointer to the element and *data* as arguments. The function *fn* should not alter *gl* (with respect to which elements are in it and in what order), but may alter the contents of individual elements.

Elements of *gl* are traversed in order from 0 through `glist_Length(gl) - 1`.

`void glist_Destroy` (*struct glist* **gl*) [Function]

Releases the memory associated with *gl*. If the elements of *gl* have privately-allocated resources (such as memory or file descriptors), those resources should be released (presumably in a `glist_FOREACH` loop) prior to calling this function, or `glist_CleanDestroy` should be used.

`void glist_CleanDestroy` (*struct glist* **gl*, *void* (**final*)(*void* *)) [Function]

Like `glist_Destroy(gl)`, but first calls *final* on a pointer to each element of *gl* (presumably as a destructor).

`void * glist_GiveUpList` (*struct glist* **gl*) [Function]

Like `glist_Destroy`, but returns *gl*'s private copy of the array of elements without destroying it. No further Glist operations are possible on *gl*. If *n* is the number of elements in *gl* and *s* is the size of an element (as given to `glist_Init`), then the returned array resides in a `malloc` block whose size is at least *n* `*` *s*, and possibly larger. This function may also return 0, indicating that *gl* is and always has been empty (no memory was ever allocated for it).

2 Dlist

A Dlist is a doubly-linked list. Like a Glist, a single Dlist holds elements of a single type. In fact, Dlists are implemented as Glists with next and previous links added. As such, many of the same rules and caveats apply: data elements are *copied* into the Dlist's private space, and pointers into Dlists can become invalid if a Dlist-growing operation (`dlist_Append`, `dlist_Prepend`, `dlist_InsertBefore`, or `dlist_InsertAfter`) causes a `realloc` of the Dlist's private memory. See Chapter 1 [Glist], page 2.

Note that Dlist elements are referred to by an integer index, but the value of the index does not reflect the element's position within doubly-linked list ordering.

`void dlist_Init` (*struct dlist* *`dl`, *int* `eltsize`, *int* `growsize`) [Function]

> Initialize the `struct dlist` pointed to by *dl*. The size of an element in the new Dlist will be *eltsize*, and when the Dlist is full, an attempt to add another element will cause the Dlist to be expanded by *growsize* uninitialized entries.

`int dlist_EmptyP` (*struct dlist* *`dl`) [Macro]

> Yields zero if *dl* is non-empty, non-zero otherwise.

`int dlist_Head` (*struct dlist* *`dl`) [Macro]

> Yields the index of the head element of *dl*, or -1 if the list is empty.

`int dlist_Tail` (*struct dlist* *`dl`) [Macro]

> Yields the index of the tail element of *dl*, or -1 if the list is empty.

`void * dlist_HeadElt` (*struct dlist* *`dl`) [Function]

> Yields the head element of *dl*. This is exactly equivalent to
>
> ```
> dlist_Nth(dl, dlist_Head(dl))
> ```
>
> The Dlist *dl* must not be empty.

`void * dlist_TailElt` (*struct dlist* *`dl`) [Function]

> Yields the tail element of *dl*. This is exactly equivalent to
>
> ```
> dlist_Nth(dl, dlist_Head(dl))
> ```
>
> The Dlist *dl* must not be empty.

`int dlist_Length` (*struct dlist* *`dl`) [Macro]

> Yields the number of elements in *dl*.

`void * dlist_Nth` (*struct dlist* *`dl`, *int* `i`) [Macro]

> Yields a pointer to the element of *dl* whose index is *i*.

`int dlist_Next` (*struct dlist* *`dl`, *int* `i`) [Macro]

> Yields the index of the element of *dl* that is next after element number *i*, or -1 if there is no next element.

`int dlist_Prev` (*struct dlist* *`dl`, *int* `i`) [Macro]

> Yields the index of the elements of *dl* that is previous to element number *i*, or -1 if there is no previous element.

`int dlist_Prepend` (*struct dlist* **`dl`**, *const void* **`elt`**) [Function]

Prepend to *dl* the element pointed to by *elt*. Returns the index of the newly-added element. The new element becomes the new head of the Dlist. It has no Prev element, and its Next element is the former head of the Dlist (if any). If *elt* is 0, an empty element initialized with `bzero` is prepended.

This function may raise the `strerror(ENOMEM)` exception.

`int dlist_Append` (*struct dlist* **`dl`**, *const void* **`elt`**) [Function]

Append to *dl* the element pointed to by *elt*. Returns the index of the newly-added element. The new element becomes the new tail of the Dlist. It has no Next element, and its Prev element is the former tail of the Dlist (if any). If *elt* is 0, an empty element initialized with `bzero` is appended.

This function may raise the `strerror(ENOMEM)` exception.

`int dlist_InsertAfter` (*struct dlist* **`dl`**, *int* `after`, *const void* **`elt`**) [Function]

Insert into *dl*, after the element whose index is *after*, an element pointed to by *elt*. Returns the index of the newly-added element. The new element becomes the Next of the element at *after*, and becomes the Prev of *after*'s old Next. If *elt* is 0, an empty element is inserted and initialized with `bzero`.

This function may raise the `strerror(ENOMEM)` exception.

`int dlist_InsertBefore` (*struct dlist* **`dl`**, *int* `before`, *const void* **`elt`**) [Function]

Insert into *dl*, before the element whose index is *before*, an element pointed to by *elt*. Returns the index of the newly-added element. The new element becomes the Prev of the element at *before*, and becomes the Next of *before*'s old Prev. If *elt* is 0, an empty element is inserted and initialized with `bzero`.

This function may raise the `strerror(ENOMEM)` exception.

`void dlist_Remove` (*struct dlist* **`dl`**, *int* `i`) [Function]

Remove from *dl* the element whose index is *i*. The deleted element's old Prev and Next elements are made to point at each other.

`void dlist_Replace` (*struct dlist* **`dl`**, *int* `n`, *const void* **`elt`**) [Function]

Replaces the element of *dl* whose index is *n* with the element pointed to by *elt*. If *elt* is 0, an empty element replaces the affected element, and is initialized with `bzero`.

`dlist_FOREACH` (*struct dlist* **`dl`**, `t`, `v`, `i`) [Macro]

Replaces the `for (...)` at the top of a loop that iterates over the elements of *dl* in order, from head to tail; *t* is the type of an element of the Dlist; *v* is the name of a variable of type `t *`, and *i* is the name of an integer variable which, each iteration through the loop, holds the value of the current index. Within the body of the loop, *v* points to the current (*i*th) element of *dl*. Example:

```
int i;
struct foo *f;

dlist_FOREACH(dl, struct foo, f, i) {
    printf("bar field of next element is %s\n", f->bar);
}
```

The expansion of this macro computes the next element in the Dlist from the current element at the end of each loop iteration. This means that the loop body must not remove the current element from the Dlist; otherwise, the "next" computation will yield garbage.

`dlist_FOREACH2` (*struct dlist* **`dl`*, *`t`*, *`v`*, *`i`*, *`j`*) [Macro]

Like `dlist_FOREACH`, but allows the loop body to remove the current Dlist element. The caller passes the name of an additional integer variable, *j*. The "next" computation takes place at the *top* of each loop and *j* holds the index of the next element.

`void dlist_Map` (*struct dlist* **`dl`*, *void* (**`fn`*)(*void* *, *void* *), *void* **`data`*) [Function]

For each element of *dl*, invoke *fn*, passing a pointer to the element and *data* as arguments. The function *fn* should not alter *dl* (with respect to which elements are in it and in what order), but may alter the contents of individual elements.

Elements of *dl* are traversed in order from head to tail.

`void dlist_Destroy` (*struct dlist* **`dl`*) [Function]

Releases the memory associated with *dl*. If the elements if *dl* have privately-allocated resources (such as memory or file descriptors), those resources should be released (presumably in a `dlist_FOREACH` loop) prior to calling this function, or `dlist_CleanDestroy` should be used.

`void dlist_CleanDestroy` (*struct dlist* **`dl`*, *void* (**`final`*)(*void* *)) [Function]

Like `dlist_Destroy(dl)`, but first calls *final* on a pointer to each element of *dl* (presumably as a destructor).

3 Dynstr

A Dynstr is a dynamically-allocated string which grows and shrinks as necessary to accommodate the characters in it. Dynstr strings are `NUL`-terminated like conventional C strings; in fact, the macro `dynstr_Str` yields the conventional string contained within a Dynstr. However, that string should not be manipulated in certain ways without its containing Dynstr's knowledge. In particular, its terminating `NUL` should not move, and the string should not be passed to `realloc` or `free` or the like.

Since the string contained in a Dynstr may relocate (as the result of a `realloc`), the pointer returned by `dynstr_Str` may become invalid. It is therefore preferable to use repeated calls to `dynstr_Str` than to squirrel away a copy of the pointer.

`void dynstr_Init` (*struct dynstr *d*) [Function]

Initializes the `struct dynstr` pointed to by *d*.

`void dynstr_InitFrom` (*struct dynstr *d, char *str*) [Function]

Initializes *d* (which should not have been previously initialized) with the already-`malloc`'d string *str*. This is equivalent to

```
dynstr_Init(d);
dynstr_Set(d, str);
```

but doesn't create a new copy of the string. Use this function when you have a string in a `malloc`-allocated piece of memory which you wish to place under Dynstr control.

The amount of space `malloc`'d for *str* is presumed to be equal to `1 + strlen(str)`. After this function call, *str* becomes the "property" of *d*, in a sense; the caller should not attempt to `free` or `realloc` it, should `dynstr_Destroy` *d* normally when finished with it, and in general should obey the rules for `dynstr_Str` outlined above.

`int dynstr_EmptyP` (*const struct dynstr *d*) [Macro]

Yields zero if *d* is non-empty (`dynstr_Length` is greater than 0), non-zero otherwise.

`int dynstr_Length` (*const struct dynstr *d*) [Function]

Returns the number of characters in *d*.

`char * dynstr_Str` (*const struct dynstr *d*) [Macro]

Yields the character string contained in *d*. The value returned is never 0 (this is a change from earlier versions of this software).

`void dynstr_Set` (*struct dynstr *d, const char *str*) [Function]

Sets the contents of *d* to be a copy of the string *str*, erasing any old contents of *d*. If *str* is 0, then any allocated memory associated with *d* is freed and the empty string is assigned to *d*.

This function may raise the `strerror(ENOMEM)` exception.

`void dynstr_Append` (*struct dynstr *d, const char *str*) [Function]

Appends to *d* a copy of the character string *str*. Like `strcat(dynstr_Str(d), str)` but without the memory-allocation hassle.

This function may raise the `strerror(ENOMEM)` exception.

`void dynstr_AppendN` (*struct dynstr *`d`, const char *`str`, int `n`*) [Function]

Appends to *d* a copy of the character string *str*, stopping after *n* characters are copied or at the end of *str*, whichever comes first. This function is to `dynstr_Append` as `strncat` is to `strcat`.

This function may raise the `strerror(ENOMEM)` exception.

`int dynstr_AppendChar` (*struct dynstr *`d`, int `c`*) [Function]

Appends to *d* the single character *c* and returns the added character.

This function may raise the `strerror(ENOMEM)` exception.

`int dynstr_Chop` (*struct dynstr *`d`*) [Function]

Shortens *d* by one character by chopping it off the end. Returns the chopped character. Inspired by Perl.

`int dynstr_ChopN` (*struct dynstr *`d`, unsigned `n`*) [Function]

Shortens *d* by chopping *n* characters off the end.

No range check is performed; *n* must not be greater than `dynstr_Length(d)`.

`int dynstr_KeepN` (*struct dynstr *`d`, unsigned `n`*) [Function]

Shortens *d* by keeping only the leading *n* characters. *n* is a count, not a positional index; if *n* is zero, the dynstr will become empty.

No range check is performed; *n* must not be greater than `dynstr_Length(d)`.

`void dynstr_Replace` (*struct dynstr *`d`, int `start`, int `len`, const char *`str`*) [Function]

Replace a substring of *d* with a new string. Characters beginning at position *start* (zero-based) and continuing for *len* bytes are replaced by the `NUL`-terminated string *str*, shortening or lengthening *d* as necessary.

No range checks are performed; *start* and `(start + len)` must both lie between 0 and `dynstr_Length(d)`, inclusive.

This function may raise the `strerror(ENOMEM)` exception.

`void dynstr_ReplaceN` (*struct dynstr *`d`, int `start`, int `len`, const char *`str`, int `n`*) [Function]

Like `dynstr_Replace`, but replaces the substring of *d* with at most *n* characters from *str* (stopping earlier if *str* is shorter than *n* characters).

This function may raise the `strerror(ENOMEM)` exception.

`void dynstr_ReplaceChar` (*struct dynstr *`d`, int `start`, int `len`, int `c`*) [Function]

Like `dynstr_Replace`, but replaces the substring of *d* with the single character *c*.

This function may raise the `strerror(ENOMEM)` exception.

`void dynstr_Insert` (*struct dynstr *`d`, int `pos`, const char *`str`*) [Macro]

Insert into *d* at position *pos* the `NUL`-terminated string *str*. This macro simply calls `dynstr_Replace(d, pos, 0, str)`.

`void dynstr_InsertN` (*struct dynstr *`d`, int `pos`, const char *`str`, int `n`*) [Macro]

Insert into *d* at position *pos* the string *str*, stopping after *n* bytes have been copied or upon reaching the end of *str*, whichever comes first. This macro simply calls `dynstr_ReplaceN(d, pos, 0, str, n)`.

`void dynstr_InsertChar` (*struct dynstr *`d`, int `pos`, int `c`*) [Macro]

Insert into *d* at position *pos* the single character *c*. This macro simply calls `dynstr_ReplaceChar(d, pos, 0, c)`.

`void dynstr_Delete` (*struct dynstr *`d`, int `pos`, int `len`*) [Macro]

Delete from *d* a substring of characters beginning at position *pos* and continuing for *len* bytes. This macro simply calls `dynstr_Replace(d, pos, len, "")`.

`void dynstr_Destroy` (*struct dynstr *`d`*) [Function]

Releases the memory associated with *d*.

`char * dynstr_GiveUpStr` (*struct dynstr *`d`*) [Function]

Like `dynstr_Destroy`, but returns *d*'s private copy of the string (as in `dynstr_Str(d)`) without destroying it. No further Dynstr operations are possible on *d*. The resulting string *str* always resides in a `malloc` block whose size is at least `strlen(str) + 1`, and possibly larger.

This function may raise the `strerror(ENOMEM)` exception.

4 Intset

An Intset is a set of integers, implemented as a sorted Dlist of *parts*. Each part is a bit vector of varying size which can represent a portion of the space of all integers. When a value is added to the set, if no part exists to hold the value, then either an existing part is expanded to include the value, or a new part is linked into the Dlist, whichever is more space-efficient.

`void intset_Init` (*struct intset* *`*iset`*) [Function]

Initializes the Intset pointed to by *iset*.

`int intset_EmptyP` (*struct intset* *`*iset`*) [Macro]

Yields zero if *iset* is non-empty, non-zero otherwise.

`int intset_Count` (*struct intset* *`*iset`*) [Macro]

Yields the number of elements in *iset*.

`void intset_Add` (*struct intset* *`*iset`*, *int* *`i`*) [Function]

Adds to the set *iset* the value *i*.

This function may raise the `strerror(ENOMEM)` exception.

`void intset_AddRange` (*struct intset* *`*iset`*, *int* *`start`*, *int* *`end`*) [Function]

Adds to the set *iset* every value in the range *start* through *end*, inclusive (*start* must not be greater than *end*). This function is equivalent to, but much more efficient than

```
int i;

for (i = start; i <= end; ++i)
    intset_Add(iset, i);
```

This function may raise the `strerror(ENOMEM)` exception.

`void intset_Remove` (*struct intset* *`*iset`*, *int* *`i`*) [Function]

Removes from *iset* the value *i*. If *i* was not a member of *iset*, this function does nothing. If this function produces an empty "part" (described above), then the part is deallocated.

`void intset_Clear` (*struct intset* *`*iset`*) [Function]

Removes all values from the set *iset*.

`int intset_Contains` (*struct intset* *`*iset`*, *int* *`i`*) [Function]

Tests whether *iset* contains the value *i*, returning zero if not, non-zero if so.

`int intset_Min` (*struct intset* *`*iset`*) [Function]

Returns the smallest member of *iset*.

If *iset* is empty, this function will raise the `strerror(EINVAL)` exception.

`int intset_Max` (*struct intset* *`*iset`*) [Function]

Returns the largest member of *iset*.

If *iset* is empty, this function will raise the `strerror(EINVAL)` exception.

`int intset_Equal` (*struct intset* *`is1`, *struct intset* *`is2`) [Function]

Tests whether *is1* contains exactly the same members as *is2*. Returns non-zero if they're equal, zero if non-equal.

`void intset_InitIterator` (*struct intset_iterator* *`isi`) [Function]

Initialize *isi* for subsequence use in calls to `intset_Iterate`.

`int * intset_Iterate` (*struct intset* *`iset`, *struct intset_iterator* *`isi`) [Function]

Iterates over the elements of *iset* using the iterator *isi*, which has been initialized with `intset_InitIterator`. Each call to this function returns a pointer to a static area containing the "next" element of *isi* in numerical order (from `intset_Min(iset)` to `intset_Max(iset)`). The iterator *isi* is used to record the current position in the traversal so that the next call to `intset_Iterate` will pick up from the correct spot. Multiple concurrent traversals, each using a separate iterator, are possible.

The contents of *isi* must not change during the traversal. When no elements remain in a traversal, this function returns 0. There is no requirement to complete a particular traversal, and an iterator may be re-initialized at any time to begin a new traversal. There is no need to finalize an iterator.

Example:

```
struct intset_iterator isi;
int *intptr;

intset_InitIterator(&isi);
while (intptr = intset_Iterate(iset, &isi)) {
    code that uses the integer in *intptr
}
```

`void intset_Destroy` (*struct intset* *`iset`) [Function]

Releases the memory associated with *iset*.

5 Hashtab

A Hashtab is a hash table, which is a fast-lookup array. Hashtab implements an *open*-style hash table atop Glists and Dlists. In an open hash table, a fixed number of *buckets* holds a variable number of table elements. Finding the correct bucket is an O(1) operation; finding the element within the bucket is a simple linear search. For this reason, it is desirable to choose a Hashtab configuration that will spread elements among buckets evenly, with no bucket getting too "deep".

Given a hash table element, the correct bucket is found by taking the *hash value* of the element modulo the number of buckets in the Hashtab. The hash value is determined by the table's *hash function*, which should attempt to map elements to as nearly unique integers as is feasible. It is also customary practice to choose a prime number for the number of buckets in the hash table (so the hash value modulo the number of buckets is least likely to exhibit periodicity), though it is by no means necessary.

Since Glists and Dlists are used, the same rules and caveats apply: one data type per Hashtab; data elements are *copied* into the Hashtab's private space; and pointers into Hashtabs can become invalid if a `hashtab_Add` causes a `realloc` of the Hashtab's private memory. See Chapter 1 [Glist], page 2.

`void hashtab_Init` (*struct hashtab* *`ht`, *unsigned int* (*`hashfn`)(*const void* *), *int* (*`compare`)(*const void* *, *const void* *), *int* `eltsize`, *int* `nbuckets`) [Function]

Initializes hash table *ht*. The hash function to use on elements of the new table is *hashfn*, which takes a pointer to an element and returns an `unsigned int` hash value. A predicate to compare pairs of elements, called the *comparison predicate*, is given as *compare*, which should take two pointers to elements and return zero if they are equal, non-zero otherwise. The size of a data element is *eltsize*. The number of buckets to create in the new table is *nbuckets*.

If *compare* is 0, then `bcmp` (or `memcmp`, whichever is available) is used (along with *eltsize*).

This function may raise the `strerror(ENOMEM)` exception.

`void hashtab_Add` (*struct hashtab* *`ht`, *const void* *`elt`) [Function]

Add to *ht* a copy of the element pointed to by *elt*. The pointer is first passed to *ht*'s hash function to obtain a hash value.

This function may raise the `strerror(ENOMEM)` exception.

`void * hashtab_Find` (*struct hashtab* *`ht`, *const void* *`probe`) [Function]

Find the element of *ht* which matches *probe* and return a pointer to it. Usually, *probe* is a pointer to a partially-filled-in version of the data structure being sought (for instance, if the elements of *ht* are key-value pairs, *probe* would be a pointer to a pair containing a valid key field but nothing in the value field). The element is found using *ht*'s hash function and comparison predicate; this means that *probe* must hash the same as the target element would, and that the comparison predicate must consider *probe* and the target element equal. If no matching element is found, 0 is returned. If multiple matches exist, only the first one found is returned. (There is no practical way to determine which matching element among several would be found first.)

`void hashtab_Remove` (*struct hashtab* *`*ht`*, *const void* *`*probe`*) [Function]

The element of *ht* matching *probe* is found (as described above in `hashtab_Find`) and removed. If no matching element is found, this function silently does nothing. If multiple matches exist, only the first one found is removed. (There is no practical way to determine which matching element among several would be found first.)

If *probe* is 0, then the last element of *ht* returned by a call to `hashtab_Find` or `hashtab_Iterate` is removed without performing a new search. This special case performs no error-checking, so be sure that there *has* been a successful `hashtab_Find` or `hashtab_Iterate` and that the resulting element has not already been removed before calling `hashtab_Remove(ht, 0)`.

`void hashtab_InitIterator` (*struct hashtab_iterator* *`*hti`*) [Function]

Initialize *hti* for subsequent use in calls to `hashtab_Iterate`.

`void * hashtab_Iterate` (*struct hashtab* *`*ht`*, *struct hashtab_iterator* *`*hti`*) [Function]

Iterates over the elements of *ht* using the iterator *hti*, which has been initialized with `hashtab_InitIterator`. Each call to this function returns a pointer to the "next" element of *ht* (according to an apparently-random traversal of *ht*). The iterator *hti* is used to record the current position in the traversal so that the next call to `hashtab_Iterate` will pick up from the correct spot. Multiple concurrent traversals, each using a separate iterator, are possible.

Each traversal is guaranteed to reach each element of *ht* exactly once, but the contents of *ht* must not change during the traversal. (If an iterator is used without reinitializing *after* the Hashtab is modified, the result isn't even guaranteed to be *in* the Hashtab!) When no elements remain in a traversal, this function returns 0. There is no requirement to complete a particular traversal, and an iterator may be re-initialized at any time to begin a new traversal. There is no need to finalize an iterator.

Example:

```
struct hashtab_iterator hti;
struct foo *fooptr;

hashtab_InitIterator(&hti);
while (fooptr = (struct foo *) hashtab_Iterate(ht, &hti)) {
    code that uses fooptr
}
```

`unsigned int hashtab_StringHash` (*const char* *`*str`*) [Function]

Computes and returns a hash value for the `NUL`-terminated string *str*. This function is provided as a convenience; it is quite a good hash function for short and medium-length strings, especially identifier names in programming languages.

`int hashtab_EmptyP` (*struct hashtab* *`*ht`*) [Macro]

Yields zero if *ht* is a non-empty Hashtab, non-zero otherwise.

`int hashtab_Length` (*struct hashtab* *`*ht`*) [Macro]

Yields the number of elements in *ht*.

`int hashtab_NumBuckets` (*struct hashtab* *`*ht`*) [Macro]

Yields the number of buckets in *ht*, which was given by the latest call to `hashtab_Init` or `hashtab_Rehash`.

`void hashtab_Stats` (*const struct hashtab* *`*ht`*, *double* *`*mean`*, *double* *`*variance`*) [Function]

Compute statistics about the distribution of elements among the buckets of *ht*. The mean length of a bucket is placed in the `double` pointed to by *mean*, and the variance is placed in the `double` pointed to by *variance*. (The square root of the variance is the *standard deviation*.) Either *mean* or *variance* may be 0 if that value is not desired, but if the variance is desired but not the mean, be aware that the mean is computed internally anyway.

`void hashtab_Rehash` (*struct hashtab* *`*ht`*, *int* *`newbuckets`*, *unsigned int* (*`*newhash`*)(*const void* *)) [Function]

Redistribute the elements of *ht* among a new number of buckets *newbuckets* and/or using a new hash function *newhash*. To leave the number of buckets alone, *newbuckets* should be 0; to leave the hash function alone, *newhash* should be 0. This function is typically used when `hashtab_Stats` reveals that the hash table has grown large or imperfectly (resulting in buckets that are too large, or in very uneven growth among the buckets), or when deletions have left the hash table sparse and fewer buckets are called for.

This function may raise the `strerror(ENOMEM)` exception.

`void hashtab_Map` (*struct hashtab* *`*ht`*, *void* (*`*fn`*)(*void* *, *void* *), *void* *`*data`*) [Function]

For each element of *ht*, invoke *fn*, passing a pointer to the element and *data* as arguments. The function *fn* should not alter *ht* (with respect to which elements are in it), but may alter the contents of individual elements.

Elements of *ht* are traversed in an apparently-random order.

`void hashtab_Destroy` (*struct hashtab* *`*ht`*) [Function]

Releases the memory associated with *ht*. If the elements of *ht* have privately-allocated resources (such as memory or file descriptors), those resources should be released (presumably in a `hashtab_Iterate` traversal) prior to calling this function, or `hashtab_CleanDestroy` should be used.

`void hashtab_CleanDestroy` (*struct hashtab* *`*ht`*, *void* (*`*final`*)(*void* *)) [Function]

Like `hashtab_Destroy(ht)`, but first calls *final* on a pointer to each element of *ht* (presumably as a destructor).

6 Sklist

A skip list is a special kind of linked list for keeping data in sorted order. Its searching efficiency is comparable to that of a balanced binary tree, but the implementation is vastly simpler and the overhead is lower.

In a skip list, each element is stored in a node of a randomly-chosen *level*. A node of level n contains `n + 1` forward pointers, where the kth pointer points to the next node of level k or higher.

When a new node is inserted in the skip list, its level is chosen based on P, the probability of choosing a higher level for the node. The probability that a node's level is higher than L is `P^(L + 1)`. A skip list's value of P is specified as a ratio (numerator, denominator) in a call to `sklist_Init`. An empirically good value for P is 1/4.

Because the random-number generator is used in creating skip list nodes, it is a good idea for the application to initialize the random-number generator some time before skip list insertions begin happening. Some C libraries provide the function `srand` for this purpose; others provide `srandom`. The macro `Srandom(seed)` invokes the correct variant for your platform.[1]

Skip lists were described in the June 1990 issue of *Communications of the ACM* (volume 33, number 6), in "Skip Lists: A Probabilistic Alternative to Balanced Trees" by William Pugh.

`void sklist_Init` (*struct sklist* *`skl`, *int* `eltsize`, *int* (*`cmp`)(*const void* *, *const void* *), *int* `numerator`, *int* `denominator`) [Function]

Initialize the skip list pointed to by *skl*. The size of an element in the new skip list is *eltsize*; *cmp* is a pointer to a *comparison function*, an `int`-returning function which takes as arguments two pointers to skip list elements and compares them in the style of `strcmp`; and *numerator* and *denominator* form the ratio P described above (the recommended values are 1 and 4).

If *cmp* is 0, then `bcmp` (or `memcmp`, whichever is available) is used (along with *eltsize*).

`int sklist_Length` (*struct sklist* *`skl`) [Macro]

Yields the number of elements in *skl*.

`int sklist_Empty` (*struct sklist* *`skl`) [Macro]

Yields zero if *skl* is non-empty, non-zero otherwise.

`void * sklist_Insert` (*struct sklist* *`skl`, *const void* *`elt`) [Function]

Insert into *skl* the element pointed to by *elt*. The referent of *elt* is copied into the skip list. The number of bytes to copy was given as *eltsize* when the skip list was initialized with `sklist_Init`. Returns a pointer to the new node in the skip list.

This function may raise the `strerror(ENOMEM)` exception.

---

[1] You should always use `Srandom` rather than calling `srand` or `srandom` directly. If your platform supplies both `srand` and `srandom`, there is no good way to tell which random-number generator will be used by Sklist, but `Srandom` will always initialize the right one.

`void * sklist_Find` (*struct sklist *skl, const void *probe, int record*) [Function]

Find in *skl* a data element which matches *probe* (according to the comparison function given in the call to `sklist_Init`). Return that element if found, or 0 if not. The *record* parameter controls whether the skip list records the path to the found item. If non-zero, the path will be recorded internally and can be used to speed a subsequent `sklist_Remove` or `sklist_CleanRemove` operation (*q.v.*). If *record* is zero, the path will not be recorded and this function will run slightly faster.

This function may raise the `strerror(ENOMEM)` exception (but only if *record* is non-zero).

`void sklist_Remove` (*struct sklist *skl, const void *probe*) [Function]

Remove from *skl* the element which matches *probe* (according to the comparison function given in the call to `sklist_Init`). If *skl* contains multiple matches for *probe* only one is removed. (There is no practical way to determine which matching element among several would be found first.) If no matching element is found, this function silently does nothing. The storage containing the affected element is freed, making the removed element inaccessible after this call; if it contains private data that needs finalizing, use `sklist_CleanRemove` instead.

If *probe* is 0, then the last element found by a call to `sklist_Find` is removed, but only if the *record* parameter was non-zero in that call, and only if no other operation involving a skip list search has intervened. (That includes `sklist_Insert`, `sklist_Remove`, `sklist_CleanRemove`, and `sklist_Find`.) Note that a call to such a function on *any* skip list invalidates the "recorded" information for *all* skip lists. Note that this common idiom is fine:

```
if (sklist_Find(skl, probe, 1)) {
    sklist_Remove(skl, 0);
} else {
    ...not found...
}
```

`void sklist_CleanRemove` (*struct sklist *skl, void *probe, void (*final)(void *)*) [Function]

Like `sklist_Remove(skl, probe)`, but first *final* is called on a pointer to the element to be removed, if one is found. Presumably *final* is a destructor.

`void * sklist_LastMiss` (*struct sklist *skl*) [Function]

Returns the *last miss* element from the previous search operation in *skl*. The last miss is the element immediately "to the left" of an element sought in the previous search, regardless of whether that search succeeded. If there is no such element in *skl*, this function returns 0.

As when using `sklist_Remove(skl, 0)`, this function may only be called after a call to `sklist_Find` in which the *record* was non-zero, and there must have been no intervening skip list search operations (`sklist_Insert`, `sklist_Remove`, `sklist_CleanRemove`, and `sklist_Find`). Note that a call to such a function on *any* skip list invalidates the "recorded" information for *all* skip lists.

`void * sklist_First` (*struct sklist *`skl`*) [Function]

Returns the first element of *skl*, or 0 if *skl* is empty.

`void * sklist_Next` (*struct sklist *`skl`, const void *`elt`*) [Function]

Returns the next element in *skl* after *elt* (which must be a non-zero pointer obtained earlier from *skl*).

`sklist_FOREACH` (*struct sklist *`skl`, `t`, `v`*) [Macro]

Replaces the `for (...)` at the top of a loop that iterates over the elements of *skl* in order; *t* is the type of an element in the Sklist; and *v* is the name of a variable of type `t *`. Example:

```
struct foo *f;

sklist_FOREACH(skl, struct foo, f) {
    printf("bar field of next element is %s\n", f->bar);
}
```

The expansion of this macro computes the next element in the Sklist from the current element at the end of each loop iteration. This means that the loop body must not remove the current element from the Sklist; otherwise, the "next" computation will yield garbage.

`sklist_FOREACH2` (*struct sklist *`skl`, `t`, `v`, `w`*) [Macro]

Like `sklist_FOREACH`, but allows the loop body to remove the current Sklist element. The caller passes the name of an additional variable of type `t *`, *w*. The "next" computation takes place at the *top* of each loop and *w* holds a pointer to the next element.

`void sklist_Map` (*struct sklist *`skl`, void (*`fn`)(void *, void *), void *`data`*) [Function]

For each element of *skl*, invoke *fn*, passing a pointer to the element and *data* as arguments. The function *fn* should not alter *skl* (with respect to which elements are in it), but may alter the contents of individual elements.

Elements of *skl* are traversed in order from first to last.

`void sklist_Destroy` (*struct sklist *`skl`*) [Function]

Releases the memory associated with *skl*. If the elements of *skl* have privately-allocated resources, those resources should be released prior to calling this function (presumably in a `sklist_FOREACH` loop), or `sklist_CleanDestroy` should be used.

`void sklist_CleanDestroy` (*struct sklist *`skl`, void (*`final`)(void *)*) [Function]

Like `sklist_Destroy(skl)`, but first calls *final* on a pointer to each element of *skl* (presumably as a destructor).

7 Prqueue

A Prqueue is a priority queue, implemented as a *heap*-style binary tree atop a Glist. As such, many of the same rules and caveats as in Glists and Dlists apply: one data type per Prqueue; data elements are *copied* into the Prqueue's private space; and pointers into Prqueues can become invalid if a `prqueue_Add` causes a `realloc` of the Prqueue's private memory. See Chapter 1 [Glist], page 2.

When a Prqueue is set up, it is given a pointer to an int-returning function that orders the Prqueue's elements according to some criterion. The element that is "lightest" in this ordering "bubbles up" to be the *head* of the Prqueue. Only the head of a Prqueue can be accessed, but it can always be found in constant time. Adding and removing elements from a Prqueue are O(log *n*) operations, where *n* is the number of elements in in Prqueue.

`void` **`prqueue_Init`** (*struct prqueue *`p`*, *int* (***`compare`)(*void **, *void **), *int* `eltsize`, *int* `growsize`) [Function]

Initializes the `struct prqueue` pointed to by *p*. The comparison function *compare* will be used to compare pairs of elements when adding or removing, and should return an integer less than, equal to, or greater than zero depending on whether its first argument is to be considered heavier than, equal to, or lighter than its second argument. (The element that compares *lightest* according to this function will bubble up to the head of the Prqueue.) The size of an element of the Prqueue is *eltsize*, and when the Prqueue is full, an attempt to add another element will cause the Prqueue to be expanded by *growsize* uninitialized entries.

`int` **`prqueue_EmptyP`** (*struct prqueue *`p`*) [Macro]

Yields zero if *p* is non-empty, non-zero otherwise.

`void *` **`prqueue_Head`** (*struct prqueue *`p`*) [Macro]

Yields a pointer to the top element of *p*.

`void` **`prqueue_Add`** (*struct prqueue *`p`*, *void **`elt`) [Function]

Adds to *p* the element pointed to by *elt*.

This function may raise the `strerror(ENOMEM)` exception.

`void` **`prqueue_Remove`** (*struct prqueue *`p`*) [Function]

Removes the head element from *p*. The next lightest element (according to the comparison *compare* provided to `prqueue_Init`) becomes the new head of the Prqueue.

`void` **`prqueue_Destroy`** (*struct prqueue *`p`*) [Function]

Releases the memory associated with *p*. If the elements of *p* have privately-allocated resources (such as memory or file descriptors), those resources should be released (presumably by repeatedly calling `prqueue_Head` and `prqueue_Remove` to get at each element until the Prqueue is empty) prior to calling this function, or `prqueue_CleanDestroy` should be used.

`void` **`prqueue_CleanDestroy`** (*struct prqueue *`p`*, *void* (***`final`)(*void **)) [Function]

Like `prqueue_Destroy(p)`, but first calls *final* on a pointer to each element of *p* (presumably as a destructor).

8 Dpipe

A Dpipe is a buffered data pipe (a FIFO) at one "end" of which bytes are written and from the other "end" of which the bytes can be read. Unlike a Unix interprocess pipe, reading and writing cannot occur in parallel, so every write must be buffered before the written data can be read back out. The internal buffer of a Dpipe grows dynamically, so no write on a Dpipe will ever block.

A Dpipe can have associated with it a *reader*, a *writer*, both, or neither. The reader is a function which will read bytes from a Dpipe upon demand, emptying its internal buffer. The writer is a function that will supply bytes to a Dpipe upon demand. The reader is invoked by the function `dpipe_Flush` and also by `dpipe_Write`, `dpipe_Put`, and `dpipe_Putchar` if *autoflushing* is enabled (to prevent the internal buffer from growing too large). The writer is invoked by `dpipe_Read` and also by `dpipe_Get` and `dpipe_Getchar` when too few bytes are available in the buffer to satisfy the pending read request.

The association of a reader or a writer with a Dpipe does not prevent other callers from writing to or reading from the Dpipe.

Dpipes can be used to abstract stream-based communication, including between processes. However, it is not possible for two processes to manipulate a Dpipe directly, due to the lack of a locking mechanism to prevent simultaneous accesses from clobbering the data structure. Instead, isolate the interprocess communication at one end of the Dpipe, for example by having the Dpipe's reader or writer communicate through an ordinary Unix pipe with a subprocess.

8.1 Basic Dpipe functions

`void` `dpipe_Init` (*struct dpipe **`dp`*,* [Function]
void (*`rd`)(*struct dpipe *, void **)*, void **`rddata`*,*
void (*`wr`)(*struct dpipe *, void **)*, void **`wrdata`*, int* `autoflush`)

Initializes the `struct dpipe` pointed to by *dp*. The function *rd* is the Dpipe's reader, and *rddata* is its callback data. The function *wr* is the Dpipe's writer, and *wrdata* is its callback data. Finally, *autoflush* controls whether autoflushing is enabled for this Dpipe.

When `dpipe_Read` or `dpipe_Getchar` is called on the Dpipe and the Dpipe's buffer is empty, then if *wr* is non-zero, it is invoked with *dp* as its first argument and *wrdata* as its second. It is expected to write some bytes into *dp* (using `dpipe_Write`, `dpipe_Putchar`, or `dpipe_Put`) or to close it (using `dpipe_Close`). Unlike prior versions of the Dpipe library, writer functions are no longer required to perform one of these actions on every call. Instead, the Dpipe code will automatically repeat calls to *wr* until one of the conditions is satisfied.

When `dpipe_Read` or `dpipe_Getchar` is called on the Dpipe and the Dpipe's buffer is empty, and if *wr* is 0, then the exception `dpipe_err_NoWriter` is raised.

When `dpipe_Flush` is called on the Dpipe and *rd* is non-zero, it is invoked repeatedly with *dp* as its first argument and *rddata* as its second, until no bytes remain in the

Dpipe's buffer. Each call to *rd* is expected to read one or more bytes out of the Dpipe.[1] See `dpipe_Flush` for an important recommendation concerning reader functions.

The equivalent of a `dpipe_Flush` also occurs when writing to a Dpipe if *autoflush* is non-zero; in that case, the flush occurs each time the Dpipe's internal buffer reaches a certain fixed limit. (If *rd* is 0, *autoflush* is ignored.)

The convenience type `dpipe_Callback_t`, defined as

```
typedef void (*dpipe_Callback_t)(struct dpipe *,
                                 void *);
```

corresponds to the type of the callbacks *wr* and *rd*.

`int dpipe_Read` (*struct dpipe* **dp*, *char* **buf*, *int n*) [Function]

Read from *dp* into *buf* at most *n* bytes. Returns the actual number of bytes read, which will only be less than *n* if end-of-file is reached. End-of-file is reached when all bytes written to the Dpipe have been read, and the Dpipe has been closed with `dpipe_Close`.

If *n* is greater than the number of bytes in the Dpipe's buffer, necessitating a call to the Dpipe's writer, and the Dpipe has no writer, then no bytes are read and the exception `dpipe_err_NoWriter` is raised. Note, however, that in this case, portions of *buf* may still have been overwritten. This function may also raise the `strerror(ENOMEM)` exception (in the case that an attempt to read causes a writer function to be called).

`void dpipe_Write` (*struct dpipe* **dp*, *const char* **buf*, *int n*) [Function]

Write to *dp* from *buf* exactly *n* bytes. If autoflushing is enabled for the Dpipe, then the Dpipe's reader may be called every so often to keep the internal buffer below a certain fixed limit. Otherwise, all *n* bytes are buffered. Dpipe buffering is dynamic, so *n* is limited only by your platform's memory capacity, or your good sense, whichever comes first. Attempting to write data to a Dpipe on which `dpipe_Close` has been called raises the `dpipe_err_Closed` exception.

This function may raise the `strerror(ENOMEM)` exception.

`int dpipe_Getchar` (*struct dpipe* **dp*) [Function]

Read and return the next character from *dp*. If *dp* is at end-of-file, return the constant `dpipe_EOF`.

This function may raise the `dpipe_err_NoWriter` exception. This function may also raise the `strerror(ENOMEM)` exception (in the case that an attempt to read causes a writer function to be called).

`void dpipe_Putchar` (*struct dpipe* **dp*, *int ch*) [Function]

Write to *dp* the single character *ch*. It is an error to write data to a Dpipe on which `dpipe_Close` has been called.

This function may raise the `strerror(ENOMEM)` exception.

`void dpipe_Unread` (*struct dpipe* **dp*, *const char* **buf*, *int n*) [Function]

Pushes back onto *dp* from *buf* *n* bytes, as if they hadn't been read in the first place. It is possible to unread data on a closed Dpipe or one that is at end-of-file.

---

[1] When `dpipe_Flush` is called on the Dpipe and *rd is* 0, then the exception `dpipe_err_NoReader` is raised.

Note that

```
dpipe_Unread(dp, "abcd", 4);
```

is the same as

```
dpipe_Unread(dp, "cd", 2);
dpipe_Unread(dp, "ab", 2);
```

and that the next four bytes read from the Dpipe will be `"abcd"`.

This function may raise the `strerror(ENOMEM)` exception.

`void dpipe_Ungetchar` (*struct dpipe* *`dp`, *int* `ch`) [Function]

Pushes back onto *dp* the single character *ch*. It is possible to unread data on a closed Dpipe or one that is at end-of-file.

This function may raise the `strerror(ENOMEM)` exception.

`int dpipe_Peekchar` (*struct dpipe* *`dp`) [Function]

Read and return the next character from *dp* without removing it from the stream. If *dp* is at end-of-file, return the constant `dpipe_EOF`.

This function may raise the `dpipe_err_NoWriter` exception. This function may also raise the `strerror(ENOMEM)` exception (in the case that an attempt to peek ahead causes a writer function to be called).

`int dpipe_Ready` (*const struct dpipe* *`dp`) [Macro]

Returns the number of bytes already buffered for reading in *dp*. Put another way, this is the number of bytes that can be read without necessitating a call to the Dpipe's writer. This is a lower bound on the number of bytes ready for reading; the actual number of readable bytes remaining in the Dpipe may be (much) higher, since a call to the Dpipe's writer (if it has one) may produce more data.

`int dpipe_Eof` (*struct dpipe* *`dp`) [Function]

Returns non-zero when *dp* is at end-of-file, zero otherwise.

Warning. This function may yield a "false negative" in some cases; that is, it may return 0 when *dp* really is at end-of-file. The ambiguous case is described under `dpipe_StrictEof` (*q.v.*), which is like `dpipe_Eof` but takes a different approach to addressing the ambiguity. In short, the ambiguity arises when *dp* is empty and not yet closed. If a writer function exists, it can be called to see whether *dp* is about to close, and that's what `dpipe_Eof` does (using `dpipe_Peekchar`).

Since this function may force a call to the writer, this function may raise the `strerror(ENOMEM)` exception.

`int dpipe_StrictEof` (*struct dpipe* *`dp`) [Function]

Returns non-zero when *dp* is at end-of-file, zero otherwise.

Warning. This function may raise the `dpipe_err_NoWriter` exception. Here's why: A Dpipe is definitely at end-of-file when no bytes remain in its buffer and when its internal "closed" flag is set (by a prior call to `dpipe_Close`). Similarly, a Dpipe is definitely *not* at end-of-file when any bytes remain in its buffer. But suppose its buffer is empty and the "closed" flag isn't set; is the Dpipe at end-of-file or not? If the Dpipe has a writer function, the next call to it might close the Dpipe without adding any more bytes; in that case, the answer is "yes," the Dpipe is at end-of-file. But the writer might also supply some bytes, in which case the answer is "no," and there's no way to tell until a call to the writer has been forced. Now suppose that the buffer is empty *and* the "closed" flag isn't set *and* the Dpipe has no writer function: is the Dpipe at end-of-file? (In this case, `dpipe_Eof` simply says "no" and risks a false negative.) There is literally no way for `dpipe_StrictEof` to know; so by raising an exception, `dpipe_StrictEof` effectively says, "I don't know, do you?" Note that this exception can never be raised when *dp* has a writer function.

Since this function forces a call to the writer (in the case described above) using `dpipe_Peekchar`, this function may also raise the `strerror(ENOMEM)` exception.

`void dpipe_Flush` (*struct dpipe *dp*) [Function]

Flush *dp*. Repeatedly calls the Dpipe's reader until no bytes remain in the Dpipe's internal buffer. If the Dpipe has no reader, raises the exception `dpipe_err_NoReader`. It is possible to flush a Dpipe that has been closed.

When a Dpipe has both a reader and a writer, calling `dpipe_Flush` will call the reader which might wind up invoking the writer as well (via `dpipe_Read`, `dpipe_Get`, or `dpipe_Getchar`). Since the goal of `dpipe_Flush` is to empty the Dpipe's buffer, it is undesirable for the reader to attempt to read so many bytes that the writer is invoked, which will simply place more bytes in the Dpipe's buffer. For that reason, the reader might want to limit itself to reading no more than `dpipe_Ready(dp)` bytes; but it must read at least one byte (unless *dp* is at end of file [which sometimes can't be known, see `dpipe_Eof`]), and `dpipe_Ready` could return 0. The consequence of reading more than `dpipe_Ready` bytes (forcing a call to the writer) is simply that the size of the internal buffer will fluctuate before emptying.

Note that Dpipes contain a preventive mechanism without which `dpipe_Flush` could invoke the reader, which could invoke the writer, which could (if autoflushing is enabled) invoke `dpipe_Flush` and the reader recursively. The mechanism simply makes `dpipe_Flush` a no-op any time a read is in progress.

`void dpipe_Pump` (*struct dpipe *dp*) [Function]

Pump all the data through *dp*. This function is only for Dpipes that have both a reader and a writer function. It works by repeatedly calling `dpipe_Get` (to get data from the writer function), then `dpipe_Unget` (to put the newly-read data back into the Dpipe), then `dpipe_Flush` (to send the data to the reader function). This cycle repeats until end-of-file is reached on the Dpipe.

This function may raise the `dpipe_err_NoWriter` or `dpipe_err_NoReader` exceptions if called on a Dpipe without both a reader and a writer function. This function may also raise the `strerror(ENOMEM)` exceptionor.

`void dpipe_Close` (*struct dpipe *dp*) [Function]

Close *dp*. Called to indicate that the writer has no more data to write to the Dpipe. When the reader has exhausted the reading buffer after a Dpipe has been closed, the writer is not called to add data to the Dpipe; instead, the Dpipe is said to be at end-of-file.

If autoflush is enabled for *dp* and *dp* has a reader function, then `dpipe_Close` flushes *dp* via a call to `dpipe_Flush`.

`void * dpipe_wrdata` (*struct dpipe* `*dp`) [Macro]

Yields the callback data for *dp*'s writer function (e.g., to finalize it prior to a `dpipe_Destroy`), which was specified in `dpipe_Init`.

`void * dpipe_rddata` (*struct dpipe* `*dp`) [Macro]

Yields the callback data for *dp*'s reader function (e.g., to finalize it prior to a `dpipe_Destroy`), which was specified in `dpipe_Init`.

`void dpipe_Destroy` (*struct dpipe* `*dp`) [Function]

Releases the memory associated with *dp*. It is not necessary to close *dp* before destroying it.

8.2 Dpipe block operations

When writing normally to a Dpipe, data is copied from the caller's buffer into the Dpipe's internal storage. Internally, the Dpipe places the copied data into a block of memory allocated by `malloc`. It can be much more efficient if the caller is able to *donate* such a block containing the data to be written. Donating a block means the caller relinquishes control over the buffer containing the data to be written; that buffer must have been created with `malloc`. The Dpipe then uses that block directly, with the data already in it, as its internal storage rather than create more storage of its own, then have to copy the data into it.

When reading from a Dpipe, a similar optimization is available. If the caller is able to accept a pointer to a Dpipe-internal `malloc` block (whose size cannot be known in advance), there is no need for copying data out of the Dpipe, then destroying the block with `free`.

`void dpipe_Put` (*struct dpipe* `*dp`, *char* `*buf`, *int* `n`) [Function]

Like `dpipe_Write`, but avoids copying by linking *buf* directly into the internal storage of *dp*. The buffer *buf* must be a `malloc`-allocated block of at least *n* bytes, and *n* must be at least 1. The caller relinquishes ownership of *buf* and must perform no further operations with it (including `free`—the Dpipe will take care of disposing of the space at the proper time).

Like `dpipe_Write`, if autoflushing is enabled for *dp*, then the Dpipe's reader may be called via `dpipe_Flush`; also, this function can raise the `dpipe_err_Closed` and `strerror(ENOMEM)` exceptions.

`int dpipe_Get` (*struct dpipe* `*dp`, *char* `**bufp`) [Function]

Like `dpipe_Read`, but assigns to `*bufp` the address of a `malloc`-allocated block containing one or more bytes from *dp*'s internal storage. Return value is the number of bytes contained in `*bufp`. If *dp* is at end-of-file, 0 is returned and `*bufp` will also be 0. If *dp*'s internal buffer is empty but it has not yet been closed with `dpipe_Close`, then the writer is called to supply one or more bytes, but if *dp* has no writer, then the exception `dpipe_err_NoWriter` is raised.

The Dpipe relinquishes its ownership of the `malloc` block it returns, meaning, among other things, that the caller must be prepared to dispose of it at some point using `free` (or donate it to a future `dpipe_Put`, or possibly `dynstr_InitFrom` [see Chapter 3 [Dynstr], page 8]).

This function may raise the `strerror(ENOMEM)` exception (in the case that an attempt to read causes a writer function to be called).

`void dpipe_Unget (struct dpipe *dp, char *buf, int n)` [Function]

Like `dpipe_Unread`, but *buf* must be a `malloc` block of at least *n* bytes which the caller relinquishes.

This function may raise the `strerror(ENOMEM)` exception.

8.3 Dpipelines

This section discusses *Dpipelines*, a convenience facility using Dpipes. A Dpipeline is a sequence of Dpipes, each feeding its output into the next one's input, via a *filter* function. The filter functions in a Dpipeline are called on demand. Each time one is invoked, it is required to read at least one byte from its source Dpipe *or* write at least one byte to its destination Dpipe *or* close its destination Dpipe.

Every non-empty Dpipeline has an ordinary Dpipe at its source, or *writing end*, and an ordinary Dpipe at its destination, or *reading end*. Callers are free to obtain handles to these Dpipes (via `dpipeline_wrEnd` and `dpipeline_rdEnd`) and use them in an ordinary way.[2]

`void dpipeline_Init (struct dpipeline *dpl, void (*rd)(struct dpipe *, void *), void *rddata, void (*wr)(struct dpipe *, void *), void *wrdata)` [Function]

Initializes the Dpipeline pointed to by *dpl*. The reader function for the rightmost Dpipe is *rd*, and its callback data is *rddata*. The writer function for the leftmost Dpipe is *wr*, and its callback data is *wrdata*. Each of *rd* and *wr* may be 0. If *rd* is non-zero, autoflush is enabled for all Dpipes in the Dpipeline.

A newly-initialized Dpipeline is empty and contains no filters, no reading end, and no writing end.

`void dpipeline_Prepend (struct dpipeline *dpl, void (*filter)(struct dpipe *, struct dpipe *, void *), void *filterdata, void (*finalize)(dpipeline_Filter_t, void *))` [Function]

Prepend a filter to *dpl*. The new filter function is *filter* which, when invoked, is passed a source Dpipe, a destination Dpipe, and *filterdata*. The function *finalize*, if non-zero, is invoked during `dpipeline_Destroy` and is passed *filter* and *filterdata* (for cleanup, if necessary). There must not be a "foreign" Dpipe prepended to *dpl* (see `dpipeline_PrependDpipe`, below); if there is, the exception `dpipe_err_Pipeline` is raised.

The source Dpipe is newly created and becomes the writing end of the Dpipeline; its writer is *wr* as passed to `dpipeline_Init` (with *wrdata* as its callback data). The destination Dpipe is the former writing end of the Dpipeline, or is newly created (and becomes the reading end of the Dpipeline, with *rd* and *rddata*, as passed to `dpipeline_Init`, as its reader and callback data) if the Dpipeline was empty.

The convenience type `dpipeline_Filter_t`, defined as

```
typedef void (*dpipeline_Filter_t)(struct dpipe *,
                                   struct dpipe *,
                                   void *);
```

[2] Care is only required in calling `dpipe_Close`. A filter function should close its destination Dpipe when it is out of data to write. Apart from that, `dpipe_Close` should only be called on the writing end.

corresponds to the type of the filter function *filter*.

The convenience type `dpipeline_Finalize_t`, defined as

```
typedef void (*dpipeline_Finalize_t)(dpipeline_Filter_t,
                                     void *);
```

corresponds to the type of the finalizing function *finalize*.

`void dpipeline_Append` (*struct dpipeline* *`dpl`, *void* (*`filter`)(*struct dpipe *, struct dpipe *, void **), *void* *`filterdata`, *void* (*`finalize`)(*dpipeline_Filter_t, void **)) [Function]

Append a filter to *dpl*. The new filter function is *filter* which, when invoked, is passed a source Dpipe, a destination Dpipe, and *filterdata*. The function *finalize*, if non-zero, is invoked during `dpipeline_Destroy` and is passed *filter* and *filterdata* (for cleanup, if necessary). There must not be a "foreign" Dpipe appended to *dpl* (see `dpipeline_AppendDpipe`, below); if there is, the exception `dpipe_err_Pipeline` is raised.

The destination Dpipe is newly created and becomes the reading end of the Dpipeline; its reader is *rd* as passed to `dpipeline_Init` (with *rddata* as its callback data). The source Dpipe is the former reading end of the Dpipeline, or is newly created (and becomes the writing end of the Dpipeline, with *wr* and *wrdata*, as passed to `dpipeline_Init`, as its writer and callback data) if the Dpipeline was empty.

The convenience type `dpipeline_Filter_t`, defined as

```
typedef void (*dpipeline_Filter_t)(struct dpipe *,
                                   struct dpipe *,
                                   void *);
```

corresponds to the type of the filter function *filter*.

The convenience type `dpipeline_Finalize_t`, defined as

```
typedef void (*dpipeline_Finalize_t)(dpipeline_Filter_t, void *);
```

corresponds to the type of the finalizing function *finalize*.

`int dpipeline_Length` (*struct dpipeline* *`dpl`) [Macro]

Yields the number of filters in *dpl*.

`void dpipeline_PrependDpipe` (*struct dpipeline* *`dpl`, *struct dpipe* *`dp`) [Function]

Prepend to the Dpipeline *dpl* the Dpipe *dp*, making *dp* the new writing end. There must be at least one filter in *dpl*; if there isn't, the exception `dpipe_err_Pipeline` is raised. If *dp* has a reader and reader data, they are cached and overwritten (to be restored to the Dpipe upon `dpipeline_UnprependDpipe` or `dpipeline_Destroy`). If *dpl* has a writer, it is superseded, and its writer data is not used (but can still be accessed with `dpipeline_wrdata(dpl)`). The autoflush status of *dp* does not change, though it will not affect the autoflush status of the other Dpipes in *dpl*. While *dp* is attached to *dpl*, do not use `dpipe_rddata(dp)`, since it will contain pipeline-private data.

The new writing end is termed a *foreign Dpipe*. When *dpl* is destroyed (with `dpipeline_Destroy`), *dp* will *not* be automatically destroyed. Note that the reader and reader data of *dp*, if any, are inaccessible while *dp* is attached to *dpl*.

It is not possible to call `dpipeline_Prepend` on a Dpipeline to which a foreign Dpipe has been prepended. The prepended Dpipe must remain the writing end of the Dpipeline.

It *is* possible to call `dpipeline_PrependDpipe` more than once on a given Dpipeline. Each call *replaces* the previous writing end with the new Dpipe. The previous writing end has its reader and reader data restored, but it is not returned or saved anywhere, so if necessary, a pointer to it should be obtained (with `dpipeline_wrEnd`) prior to replacing it. (If the writing end of a Dpipeline is not foreign, then the Dpipeline will manage that Dpipe itself. Foreign writing ends, however, are the responsibility of the caller.)

It is possible to connect two Dpipelines like this:

```
dpipeline_PrependDpipe(dpl2, dpipeline_rdEnd(dpl1));
```

which will cause *dpl1* to feed into *dpl2*. Remember, though, that this causes the reader and reader data of `dpipeline_rdEnd(dpl1)` to be hidden, and that `dpipeline_Destroy(dpl2)` will not reclaim `dpipeline_rdEnd(dpl1)` (nor would you want it to).

This function may raise the `strerror(ENOMEM)` exception.

`void` **`dpipeline_AppendDpipe`** (*struct dpipeline* *`dpl`, *struct dpipe* *`dp`) [Function]

Append to the Dpipeline *dpl* the Dpipe *dp*, making *dp* the new reading end. There must be at least one filter in *dpl*; if there isn't, the exception `dpipe_err_Pipeline` is raised. If *dp* has a writer and writer data, they are cached and overwritten (to be restored to the Dpipe upon `dpipeline_UnappendDpipe` or `dpipeline_Destroy`). If *dpl* has a reader, it is superseded, and its reader data is not used (but can still be accessed with `dpipeline_rddata(dpl)`). The autoflush status of *dp* does not change, though it will not affect the autoflush status of the other Dpipes in *dpl*. While *dp* is attached to *dpl*, do not use `dpipe_wrdata(dp)`, since it will contain pipeline-private data.

The new reading end is termed a *foreign Dpipe*. When *dpl* is destroyed (with `dpipeline_Destroy`), *dp* will *not* be automatically destroyed. Note that the writer and writer data of *dp*, if any, are inaccessible while *dp* is attached to *dpl*.

It is not possible to call `dpipeline_Append` on a Dpipeline to which a foreign Dpipe has been appended. The appended Dpipe must remain the reading end of the Dpipeline.

It *is* possible to call `dpipeline_AppendDpipe` more than once on a given Dpipeline. Each call *replaces* the previous reading end with the new Dpipe. The previous reading end has its writer and writer data restored, but it is not returned or saved anywhere, so if necessary, a pointer to it should be obtained (with `dpipeline_rdEnd`) prior to replacing it. (If the reading end of a Dpipeline is not foreign, then the Dpipeline will manage that Dpipe itself. Foreign reading ends, however, are the responsibility of the caller.)

It is possible to connect two Dpipelines like this:

```
dpipeline_AppendDpipe(dpl1, dpipeline_wrEnd(dpl2));
```

which will cause *dpl1* to feed into *dpl2*. Remember, though, that this causes the writer and writer data of `dpipeline_wrEnd(dpl2)` to be forgotten, and that `dpipeline_Destroy(dpl1)` will not reclaim `dpipeline_wrEnd(dpl2)` (nor would you want it to).

This function may raise the `strerror(ENOMEM)` exception.

`struct dpipe * dpipeline_UnprependDpipe` (*struct dpipeline* *`dpl`) [Function]

Removes and returns the foreign Dpipe from *dpl* that was prepended by `dpipeline_PrependDpipe`. If the writing end of *dpl* is not foreign, the exception `dpipe_err_Pipeline` is raised.

The foreign Dpipe has its reader and reader data restored, and the Dpipeline has its writer and writer data restored. The foreign Dpipe is replaced (in the Dpipeline) with a new non-foreign Dpipe.

This function may raise the `strerror(ENOMEM)` exception.

`struct dpipe * dpipeline_UnappendDpipe` (*struct dpipeline* *`dpl`) [Function]

Removes and returns the foreign Dpipe from *dpl* that was appended by `dpipeline_AppendDpipe`. If the reading end of *dpl* is not foreign, the exception `dpipe_err_Pipeline` is raised.

The foreign Dpipe has its writer and writer data restored, and the Dpipeline has its reader and reader data restored. The foreign Dpipe is replaced (in the Dpipeline) with a new non-foreign Dpipe.

This function may raise the `strerror(ENOMEM)` exception.

`struct dpipe * dpipeline_wrEnd` (*struct dpipeline* *`dpl`) [Function]

Returns the Dpipe at the "writing end" of *dpl*.

`struct dpipe * dpipeline_rdEnd` (*struct dpipeline* *`dpl`) [Function]

Returns the Dpipe at the "reading end" of *dpl*.

If a Dpipeline has both a reader function and a writer function, it is possible to "pump" all the data through it by calling `dpipe_Pump(dpipeline_rdEnd(dpl))`.

`void * dpipeline_wrdata` (*struct dpipeline* *`dpl`) [Macro]

Yields the writer callback data specified in `dpipeline_Init`. This is the same as `dpipe_wrdata(dpipeline_wrEnd(dpl))`, unless a foreign Dpipe has been prepended with `dpipeline_PrependDpipe`.

`void * dpipeline_rddata` (*struct dpipeline* *`dpl`) [Macro]

Yields the reader callback data specified in `dpipeline_Init`. This is the same as `dpipe_rddata(dpipeline_rdEnd(dpl))`, unless a foreign Dpipe has been appended with `dpipeline_AppendDpipe`.

`void dpipeline_Destroy` (*struct dpipeline* *`dpl`) [Function]

Releases all the memory associated with *dpl* (including destroying all the Dpipes it contains), except for foreign Dpipes. If the writing end of *dpl* is foreign, its reader and reader data are restored. If the reading end of *dpl* is foreign, its writer and writer data are restored.

9 Miscellaneous

`void safe_bcopy` (*void* *`src`, *void* *`dest`, *int* `n`) [Function]

Copies *n* bytes from *src* to *dest*, correctly handling overlapping blocks of memory. This function should only be called when it is known that the blocks at *src* and *dest* can overlap (if *src* and *dest* point to two separate `malloc` blocks, for example, or if they point inside two seperately-declared structures or arrays, they *cannot* overlap, so `safe_bcopy` should not be used). This function only gets defined if the native `bcopy` cannot handle overlaps. If it can, then `safe_bcopy` is defined as a macro in terms of the native `bcopy`. (The macro will use `memcpy` if `bcopy` is not available.)

10 Compiling

To use DYNADT in your application, link with the library `libdynadt.a`, and also with `libexcept.a`, on which DYNADT depends. Include one or more of the following header files depending on which subsystems of DYNADT you are using: `glist.h`; `dlist.h`; `dynstr.h`; `intset.h`; `hashtab.h`; `sklist.h`; `prqueue.h`; `dpipe.h`.

Index

F

G

H

I

O

P

R

S

V

W

Table of Contents