# The DQUE C Library

## A Double Ended Queue Library in C

**Dale H Anderson**
**9/4/2019**

A flexible, easy to use library of function written in C that can be used to implement FIFOs (queue), LIFOs (stack), and priority queues.

# Introduction

What is the DQUE library?

- DQUE is a set of functions that implement a double-ended queue using a circular linked-list
- Can be used for LIFO, FIFO, and simple priority queues
- Can push and pop nodes onto/from the front or back of the queue with runtime of O(1)
- Priority queue search/insert/remove functions have a runtime of (n)

Why use the DQUE library?

- Simple and easy to use
    - Yes, yes, you've heard that before, but this really is easy (yes, you've heard that too)
    - All structures hidden from user, no need to learn how DQUE works under the covers
    - No need to integrate or adapt to user 's code, all functions used without modifications
    - Uses forward/rearward iterators instead of pointer manipulations
    - Most function names similar to C++ standard library deque/list names modified for C
    - Short function names with most functions only requiring two arguments
    - All functions return error code similar to C-11 functions with additional error checking
- Extremely flexible to use
    - DQUE routines only store a pointer to user's supplied data, so data can be anything
    - A single pointer to data can be inserted on multiple queues
    - A single queue can store multiple types of data (user responsible for tracking types)
    - All priority queue keys/comparison functions are user supplied, so keys can be anything
    - Comparison function not fixed, allows multiple key/function pairs on single queue
- Small, efficient code
    - Extensive code reuse, e.g. only one routine each to insert/remove node onto a queue
    - Efficient dynamic memory allocations that reduce possible memory leaks
    - Most functions are written in separate files so linkers only link code actually used

# DQUE API

## Member functions

### dque_create() – Create an empty queue

```
dque_create(
dque_qhead    **queue,     pointer to pointer to queue head
unsigned       siz,        size of the static memory buffer
char           *buf  )     pointer to static memory to use for qhead and qnode allocations
```

Create and populate a qhead structure for a circular doubly-linked list. This function can be used to create the queue head structure in three different ways. The first, and most common, method is to invoke the function with a buffer size argument of 0 (zero) which creates an empty queue using dynamically allocated memory for the queue head structure. The second method is to invoke the function with a buffer size equal to the size of qhead structure. Again, this creates an empty queue using the specified buffer for the qhead structure, but also prevents the whole library from using any dynamically allocated memory. The user may create qnodes for the queue by using the dque_options() function with the DQUEOPT_NOALLOC option to specify additional memory blocks used to create the qnodes. The third, and final, method is provided as a convenience and is a slight variation of the second method in that the specified buffer is much larger than the qhead structure. The extra buffer space is then used to create as many qnodes as possible and placed on the free list. The user can use the dque_max_size() function to query how many nodes were created.

The queue head structure contains pointers to three lists. The first pointer, called ‘head’, is the actual user queue. The second pointer, called ‘free’, is the free list holding the nodes that are available for the user, but that are not yet used. The third pointer, called ‘blks’, is the list used to track the dynamic memory allocations. When using dynamic memory allocations, the qnodes are created in groups as a single memory allocation. The pointer to the memory allocation is also a pointer to the first node, so the first node is inserted onto the third list, ‘blks’, to be used later to free the memory allocation. The remaining qnodes are inserted on the ‘free’ list.

The queue head contains counts for each of the three lists. The number of qnodes on the use’s queue can be obtained with the dque_size() function. The dque_max_size() function returns the number of qnodes on both the user’s queue and the free list. The qhead structure also contains a variable of bit flags that are used internally to control the behavior of the library functions. The last variable in the queue head is an integer to store the version number of the library.

Returns non-zero error code for failure, zero for success and a pointer to the head of the queue

### dque_destroy() – Destroy a queue, free any dynamic memory allocations

```
dque_destroy(
qhead          **queue )      pointer to pointer to queue head
```

Destroy a queue. If the queue used dynamically allocated memory, then free the allocated memory. The pointer to each node on the queue head structure 'blks' list is the same pointer returned by the memory allocation function. This function frees the memory in the opposite order of their allocation. The function frees the memory for the qhead structure and then sets the user's qhead pointer to NULL.

Returns non-zero error code on failure, zero for success.

### dque_error() – Return an error message to the specified error code

```
dque_error(
dque_err       err,           error code
char           *buf,          buffer for equivalent error message
unsigned int   siz )          size of buffer
```

This function returns an error message equivalent to the specified error code. The error message is placed in the specified buffer up to siz-1 characters. If the error is negative, this function automatically negates the error code. If the error code is out of range of the known error numbers, this function will return the unknown error code and fill the buffer with the unknown error code message.

Return non-zero error code on failure, zero for success.

### dque_error_len() – Return the length of the error message for the specified error code

```
dque_error_len(
dque_err       err,           specified error code
```

unsigned int *len ) returned length of specified error

This function returns the length of error message for the specified error code.

Return non-zero error code on failure, zero for success and length of error message.

### dque_options() – Control queue behavior with options and arguments

dque_options(

dque_qhead *queue, queue to control

int opt, get/set option

__VA_ARGS__) various arguments based on option opt

This function returns complete status and pointer to data based on option. Options:

DQUEOPT_NOOPT dque_options( queue, DQUEOPT_NOOPT )

No option option, hehe.

DQUEOPT_VERSION dque_options( queue, DQUEOPT_VERSION, int *major, int *minor )

Returns major and minor version numbers of the DQUE library.

DQUEOPT_NODECNT dque_options( queue, DQUEOPT_NODECNT, unsigned int siz )

Sets the number of nodes to allocate with each memory allocation. The allocation is treated as an array of qnodes of size siz. The first qnode is placed on the ‘blks’ list to be used later to free the memory allocation. The remaining siz-1 qnodes are place on the ‘free’ list. The default size is 25, and the minimum size is 2.

DQUEOPT_HEADSIZ dque_options( queue, DQUEOPT_HEADSIZ, unsigned int *siz )

Returns the memory size of the queue head structure in siz. This option is provided for the user that wants to allocate their own memory for the queue.

DQUEOPT_NODESIZ dque_options( queue, DQUEOPT_NODESIZ, unsigned int *siz )

Returns the memory size of the queue node structure in siz. This option is provided for the user that wants to allocate their own memory for the queue.

DQUEOPT_NOALLOC dque_options( queue, DQUEOPT_NOALLOC, unsigned int siz, char *buf )

If the queue was created with a siz argument of zero, the queue will use dynamically allocated memory and this option will return an error. If the queue is not using dynamic memory allocation, this option uses the specified buffer, of size siz, to create qnodes. Since this is not using dynamically allocated memory, the first qnode is not placed onto the 'blks' list. All qnodes are placed on the 'free' list.

DQUEOPT_NODUPE dque_options( queue, DQUEOPT_NODUPE, unsigned int siz )

A siz value of non-zero means no duplicates in priority queue, a zero means allow duplicates. If the user attempts to insert a qnode onto a priority queue (a queue built using dque_key_insert) and the key in the new qnode matches an existing qnode in the queue, the insertion will fail and the function will return an error code. The function dque_key_insert() is the only function that can make this check. The user should refrain from using all other functions that add a qnode to the queue since none of these functions can make this duplicate check. (i.e. dque_insert(), dque_push_back(), and dque_push_front())

DQUEOPT_NOSCAN dque_options( queue, DQUEOPT_NODUPE, unsigned int siz )

A siz value of non-zero means do not scan the queue on dque_insert() and dque_remove(), and a zero means to scan the queue to verify that the specified iterator is in the queue. The functions dque_insert() and dque_erase() are given pointer to a queue and a pointer to an iterator. It is assumed that the specified iterator belongs to the specified queue. The internal scan function checks each qnode in the queue looking for the qnode that is also pointed to by the iterator. This scan accounts for the runtime of O(n). The user can use this option to stop the scan, which will result in a runtime of O(1), but the user is then responsible to insure that the specified iterator belongs to the specified queue.

Return non-zero error code on failure, zero for success, and if a query, returned pointers to data.

## Capacity Functions

### dque_empty() – Return non-zero value if queue is empty, otherwise zero

dque_empty(

dque_qhead *queue, queue to check

unsigned int *data ) pointer to returned data

This function returns a non-zero value in the variable data if the queue is empty. The queue is empty if the head pointer of the queue head is NULL.

Return non-zero error code on failure, or zero for success.

### dque_size() – Return the current number of nodes in the user's queue

dque_size(

dque_qhead *queue, queue to check

unsigned int *data ) pointer to returned data

This function returns the number of nodes currently in the user's queue. This function has a runtime of O(1) since all node counts are updated with every insertion and removal.

Return non-zero error code on failure, zero for success and the number of nodes currently in the queue.

### dque_max_size() – Return the current number of nodes in the user's queue and on free list

dque_max_size(

dque_qhead *queue, queue to check

unsigned int *data ) pointer to returned data

Return number of nodes available to use in the queue. This includes nodes currently in the user's queue and the nodes on the free list. However, the allocated blocks list (blks) are not included in the count since they cannot be used by the user. This function has a runtime of O(1) since all node counts are updated with every insertion and removal.

Return non-zero error code on failure, zero for success and the maximum node count.

## Access Functions

### dque_back() – Return pointer to user's data from the last element at the back of the queue

dque_back(

| | | |
|---|---|---|
| dque_qhead | *queue, | queue with element data |
| dque_qiter | **data ) | pointer to pointer to returned data |

This function returns a pointer to the user's data from the qnode at the back of the queue. The macro gethead(queue) returns a pointer to the first node in the queue. Since the queue is implemented as a circular doubly-linked list, the previous pointer from the first node pointer points to the last node in the list. So the macro getprev(gethead(queue)) returns a pointer to the last node and the macro getdata(node) returns a pointer to the user's data that is stored in this qnode element. The last node is not removed from the list.

Return non-zero error code on failure, zero for success and pointer to the user's data.

### dque_front() - Return pointer to user's data from the first element at the front of the queue

dque_front(

| | | |
|---|---|---|
| dque_qhead | *queue, | queue with element data |
| dque_qiter | **data ) | pointer to pointer to returned data |

This function returns a pointer to the user's data from the qnode at the front of the queue. The macro gethead(queue) returns a pointer to the first node in the queue, and the macro getdata(node) returns a pointer to the user's data that is stored in this qnode element. The first node is not removed from the list.

Return non-zero error code on failure, zero for success and a pointer to the user's data.

### dque_key_find() – Return pointer to an iterator from the element specified by the key

dque_key_find(

| | | |
|---|---|---|
| dque_qhead | *queue, | queue to search for node w/key |
| void | *key, | key to search the queue with |
| COMPFUNC | comp, | user defined comparison function |
| dque_qiter | **iter ) | pointer to iterator w/key node |

Search the queue for a qnode with the specified key, and if found, returns an iterator pointing to the found qnode. The comparison function will return zero if the current node in queue matches the key. The comparison function takes two arguments in which the first argument MUST be the specified key, and the second argument is user's data pointer stored in each qnode. The user supplied comparison function should return an integer follows:

A value less than zero if key < qnode->data->key

A value equal to zero if key = qnode->data->key

A value greater than zero if key > qnode->data->key

The user is required to specify the comparison function with each call to allow flexibility with the key/comparison function pair. See the file test.c for an example of student records placed into a priority queue. The first field in the record is an integer representing the student ID and the queue is filled using a simple integer comparison function to sort the students in the queue by ID. After that, a more complicated key/comparison function pair is used to find a particular student by name.

Since any queue is ordered, the search of the queue is aborted when the comparison function returns a value that is less than zero. This can be a problem if searching a queue that is not ordered, or searching on a different key other than the one used to order the queue. An example of this second type of search was presented in the previous paragraph with the student records sorted by student ID, and then searched for a specific student name. In either case, the comparison function should return zero when it finds a key match, otherwise the function should always return a value greater than zero to force a search of the entire queue.

All of the descriptions here describe the queue as ordered in ascending order, but a comparison function that negates the return value will create a descending ordered queue with no other changes. This means the comparison function returns a value greater than zero when the specified key is less than the key in the current qnode, and a value less than zero when the specified key is greater than the key in the current qnode.

Since it is possible to search the entire queue, (e.g. the specified key is not in the queue) the runtime is O(n).

This function returns non-zero error code on failure, zero for success and iterator to found qnode.

### dque-get-data() - Return pointer to user's data from the element specified by the iterator

```
dque_get_data(
dque_qiter      *iter,          pointer to iterator
void            **data )        pointer to pointer to user's data
```

Return a pointer to the user's data from the qnode specified by the iterator.

Return non-zero error code on failure, zero for success and a pointer to the user's data.

### dque-set-data() – Set the user's data in the element specified by the iterator

```
dque_set_data(
dque_qiter      *iter,          pointer to iterator
void            *data )         pointer to user's data
```

Set the data in the qnode specified by the iterator. Note that if this is a priority queue, changing the data may violate the order of the queue. To avoid this, it is recommended that the user remove the qnode from the queue, change the data, and then re-insert the qnode to insure that the qnode is in the proper order.

The functions dque_get_data() and dque_set_data() are provided so that a user may swap values (user data) as a part of a sort routine. Both swap and sort functions are planned as a future expansion of this library.

Return non-zero error code on failure, zero for success.

## Iterator Functions

Currently, an iterator for the queue is just a pointer to any node in the queue. Other libraries, and maybe this library in the future, use a separate real data structure for the iterator that contains other information useful for iterator manipulation. The extra information could be iterator direction, a flag to indicate a valid iterator, or a pointer to qhead structure to verify that the iterator belongs on that queue (see dque_insert()/dque_earase()). However, there are some disadvantages to having a separate iterator structure including allocating/deleting the iterator structures, and the required code to manipulate the structures. Currently, the advantages do not outweigh the disadvantages. This does have some implications for the current implementation of iterators.

Since an iterator is simply a pointer to any qnode in the queue, there is no difference between a forward and reverse iterator. This means the functions dque_next() and dque_rnext() may be used with any iterator. Because the queue is implemented as a circular double-linked list, there is no actual end to the queue. Therefore, both dque_end() and dque_rend() return a NULL pointer to an iterator to terminate the iteration through the queue. This means the detection of the end of iteration has to be determined by dque_next() and dque_rnext() so that they can return a NULL pointer to match the NULL pointers

returned by dque_end() and dque_rend(). See dque_next() and dque_rnext() to see how that determination is made. This also means that any iterator will terminate correctly with either dque_next() or dque_rnext() in the appropriate direction.

One downside of implementing iterators in way they are described here is determining when an iterator becomes 'invalid'. If a qnode is inserted or deleted from a queue in "front" of the iterator, the iterator will still "see" the insertion/deletion and so is still valid. If the insertion/deletion is "behind" the iterator, then the iterator will not "see" the insertion/deletion and it is up to the user to decide if the iteration is still valid. However, if the iterator is pointing to a qnode that has been removed from the queue, a call to either dque_next() or dque_rnext() with that iterator will return an error and a NULL iterator pointer. This happens because when a qnode is removed from the queue, the user's data pointer is set to NULL. Since it is illegal, by definition, to have a valid qnode in the queue with a NULL pointer to the user's data, this is a reasonable method to mark an invalid qnode. Both dque_next() and dque_rnext() check for a NULL pointer to the user's data in the qnode specified by the iterator, and if NULL, return an error code and a NULL iterator pointer to indicate an invalid iterator.

It should be noted that because of data types needed by certain polymorphic macros, the iterator pointer declaration is hidden in dque.h by a definition of a fake iterator structure. In the file mydque.h, the real iterator, dque_qiter, is defined to be the same as dque_qnode.

### dque-begin() – Return a forward iterator to the first qnode at the front of the queue

dque_begin(

| | | |
|---|---|---|
| dque_qhead | *queue, | queue with element data |
| dque_qiter | **iter ) | pointer to a pointer to an iterator |

Return iterator to the front of the queue. An iterator for the queue is just a pointer to any node in the queue. The macro gethead(queue) returns a pointer to the first node in the queue. Since this is what is needed, it is returned as the iterator.

Return non-zero error code on failure, zero for success and an iterator to first qnode of the queue.

### dque_rbegin() – Return a reverse iterator to the last qnode at the back of the queue

dque_rbegin(

| | | |
|---|---|---|
| dque_qhead | *queue, | queue with element data |

dque_qiter **iter ) pointer to a pointer to an iterator

Return a "reverse" iterator to the back of the queue. An iterator for the queue is just a pointer to any node in the queue. The macro gethead(queue) returns a pointer to the first node in the queue. The macro getprev(gethead(queue)) returns a pointer to the last node in the queue. Since this is what is needed, it is returned as the iterator.

Return non-zero error code on failure, zero for success and an iterator to first qnode of the queue.

### dque_next() – Return next forward iterator towards the back of the queue

dque_next(

dque_qhead *queue, queue with element data

dque_qiter **iter ) pointer to a pointer to an iterator

Return an iterator to the next qnode forward in the queue. An iterator for the queue is just a pointer to any qnode in the queue. The macro getnext(iter) returns a pointer to the next qnode forward in the queue and the macro gethead(queue) returns a pointer to the first qnode in the queue. If the next qnode forward from the iterator's current position is equal to the first node, then the iteration is complete and this function will return a NULL pointer to stop the iteration.

Return non-zero error code on failure, zero for success and an iterator to the next node forward in the queue.

### dque_rnext() – Return the next reverse iterator towards the front of the queue

dque_rnext(

dque_qhead *queue, queue with element data

dque_qiter **iter ) pointer to a pointer to an iterator

Return the iterator to the next qnode rearward in the queue. An iterator for the queue is just a pointer to any qnode in the queue. The macro getprev(iter) returns a pointer to the previous qnode in the queue and the macro gethead(queue) returns a pointer to the first node in the queue. If the current

iterator is equal to the first node, then we have completed the iteration and this function will return a NULL pointer to stop the iteration.

Return non-zero error code on failure, zero for success and an iterator to the previous node rearward in the queue.

### dque_end() – Return a forward iterator to stop the iteration of the queue

```
dque_end(
dque_qhead    *queue,        queue to have iterator
dque_qiter    **iter )       pointer to a  pointer to an iterator
```

Return a forward iterator to stop iteration. Since a circular doubly-linked list does not use a sentinel node, there is technically no end iterator. However, we use a null pointer to stop iteration. The functions dque_next() and dque_rnext() determine if the current iterator is at the end or beginning of the queue respectively and return a NULL pointer. Also, most pointer errors will return a NULL pointer.

Return non-zero error code on failure, zero for success and an iterator to stop iteration.

### dque_rend() – Return a reverse iterator to stop the iteration of the queue

```
dque_rend(
dque_qhead    *queue,        queue to have iterator
dque_qiter    **iter )       pointer to a pointer to an iterator
```

Return a reverse iterator to stop iteration. Since a circular doubly-linked list does not use a sentinel qnode, there is technically no end iterator. However, we use a null pointer to stop iteration. The functions dque_next() and dque_rnext() determine if the current iterator is at the end or beginning of the queue respectively and return a NULL pointer. Also, most pointer errors will return a NULL pointer.

Return non-zero error code on failure, zero for success and an iterator to stop iteration.

## Modifier Functions

### dque_delete() – Alias for dque_erase() in C only

### dque_erase() – Remove qnode specified by the iterator, return pointer to user's data

```
dque_erase(
dque_qhead    *queue,        queue to have node deleted
void          **data,        Return user's data from deleted node
dque_qiter    *iter )        iterator to the specified node
```

Remove qnode from the specified queue at the specified iterator. A 'queue' is simply a pointer to a qnode in the circular doubly-linked list with the pointed at node considered the head of the queue. If the place to remove the node is not at the front of the queue, then use the iterator as a temporary queue head to remove the node at the front of the temporary 'queue'. Also, if the iterator is not pointing to the first node in the queue, and if the DQUE_NOSCAN flag is not set, then the queue is searched to insure that the iterator is pointing to a node in the specified queue. See the file dque_works.pdf for a more detailed description on how this works.

If the iterator points to the first node in the queue, then no scan is required and the first node is removed from the queue. If the iterator does not point to the first node and the option DQUE_NOSCAN is not set, then the queue is scanned to insure that the iterator points to a qnode in this queue. If the iterator is found, then that qnode indicated by the iterator is deleted. If the iterator is not found in the queue, then the returned data is set to NULL and no error code is returned. This is by definition since it is defined that removing a non-existent node is not an error. If the qnode indicated by the iterator is successfully removed from the queue, then the pointer to the user's data in the qnode is set to the returned data variable and the qnode data pointer is then set to NULL to indicate that the qnode is invalid. The qnode is then inserted onto the 'free' list.

Return non-zero error code on failure, zero for success and pointer to data from removed qnode.

### dque_insert() – Insert a qnode with user's data just before the qnode specified by iterator

```
dque_insert(
dque_qhead    *queue,        queue to insert node into
void          *data,         data to insert into queue
```

| | | |
|---|---|---|
| dque_qiter | *iter ) | iterator indicating insertion point |

Insert a new node with the specified data into the queue just before the node specified by the iterator. A 'queue' is simply a pointer to a node in the circular doubly-linked list with the node pointed at by the pointer considered the head of the queue. If the iterator ‘iter’ is a NULL pointer, then it is assumed that the new node should be appended to the end of the queue. If the place to insert the node is not at the front or back of the queue, then use the iterator is used as a ‘temporary’ queue to insert the node at the 'back' of the temporary 'queue' which is also just before the where the iterator is pointing. Also, if the iterator is not pointing to the first node in the queue, and if the DQUE_NOSCAN flag is not set, then the queue is searched to insure that the iterator is pointing to a node in the specified queue.

The ‘free’ list is checked, and if empty, a call is made to an internal routine to allocate more qnodes. Assuming that allocation succeeded, a node is removed from the ‘free’ list and the user’s data is set in the new qnode. If the iterator is NULL, then the new qnode is appended to the end of the queue. If the iterator points to the first qnode in the queue, the new qnode is inserted into the front of the queue. If neither situation applies, then it points to a qnode in the interior of the queue. If the option DQUE_NOSCAN is not set, then the queue is scanned to insure that the iterator points to a qnode in this queue. If the iterator belongs to this queue, then this function inserts the new node into the queue using the iterator as a ‘temporary’ queue head.

Return non-zero error code on failure, zero for success.

### dque_key_delete() – Alias for dque_key_erase() in C only

### dque_key_erase() – Remove the first qnode with the specified key, and return the user’s data

dque_key_erase(

| | | |
|---|---|---|
| dque_qhead | *queue, | queue to scan for qnode to delete |
| void | *key, | key to search queue with |
| COMPFUNC | comp, | user defined comparison function |
| void | **data ) | pointer to data from the removed qnode |

Search the queue using specified key for the node to delete. It is technically OK (by definition) if the queue is empty or if the qnode specified by the key is not in the queue. In either case, no error is returned and the pointer to data is set to NULL.

Assuming that a qnode exists in the queue that matches the specified key, it is removed from the queue. The returned data variable is set to the qnode data pointer, and then the qnode data pointer is set to NULL to indicate that the qnode is invalid. The qnode is then inserted onto the 'free' list.

The comparison function takes two arguments and the first one MUST be the specified key argument and the data pointer stored in each node as the second argument. The user supplied comparison function should return an integer follows:

A value less than zero if key < qnode->data->key

A value equal to zero if key = qnode->data->key

A value greater than zero if key > qnode->data->key

See the function dque_key_find() for further explanation on how the key/comparison function pair work together.

Return non-zero error code on failure, zero for success and pointer to the data from the deleted qnode.

### dque_key_insert() – Insert new node with user's data into sorted queue

```
dque_key_insert(
dque_qhead    *queue,      queue to scan for insertion
void          *key,        key to use for insertion point
COMPFUNC      comp,        user defined comparison function
void          *data )      data to insert into queue
```

Search the queue for insertion point using specified key. Insertion point is defined to be just before the node with a key that is larger than specified key. This means new qnodes will be placed towards the back of the queue behind any other qnodes with the same key value. This forms a FIFO queue for qnodes of the same value.

The dque_myscan() function will scan the entire queue and return the correct insertion point. However, it will return a pointer to the first node in the queue if the new node is to be inserted at the front or the back of the queue, that is, the node has the smallest or the largest value in the queue. This would require two checks on return, one to see if the insertion point matches the head of the queue, and one to see if the node should be the first or the last node. This function chooses to make these checks before calling dque_scan() by checking to see if node to insert is to be inserted at the front or the back of the queue. These checks will not save much execution time if the node is to be inserted at the front of the

queue, but does save some time if the node is to inserted at the back of the queue since the scan of the queue is not required. This also saves time because the returned insertion point from dque_myscan() is guaranteed to be within the queue and no further checks are required. The downside of this approach is that if dque_myscan() is called, one, or both, of these checks are repeated. Note that both of these checks must be made using the user specified comparison function.

The comparison function takes two arguments and the first one MUST be the specified key argument and the data pointer stored in each qnode as the second argument. The user supplied comparison function should return an integer follows:

A value less than zero if key < qnode->data->key

A value equal to zero if key = qnode->data->key

A value greater than zero if key > qnode->data->key

See the function dque_key_find() for further explanation on how the key/comparison function pair work together.

Under normal circumstances, the key and the data to insert will be the same, so as a convenience, if the key is a null pointer, the pointer to the data will be used for the key pointer.

Return non-zero for failure, zero for success

### dque_pop-bqck() – Remove qnode from the back of the queue, return pointer to user's data

```
dque_pop_back(
dque_qhead    *queue,        queue with element data
void          **data )       pointer to pointer to returned data
```

If he queue is empty, this function will return no error and set the data pointer to NULL. If the queue is not empty, this function removes the last qnode from the queue, places it onto the free list, and then returns the pointer to the data from the removed qnode. This function also sets the data pointer of the removed node to NULL to indicate the node is now invalid to any iterators.

Return non-zero error code on failure, zero for success and pointer to user's data,

### dque_pop_front() - Remove qnode from the front of the queue, return pointer to user's data

dque_pop_front(

| | | |
|---|---|---|
| dque_qhead | *queue, | queue with element data |
| void | **data ) | pointer to pointer to returned data |

If he queue is empty, this function will return no error and set the data pointer to NULL. If the queue is not empty, this function removes the first qnode from the queue, places it onto the free list, and then returns the pointer to the data from the removed qnode. This function also sets the data pointer of the removed node to NULL to indicate the node is now invalid to any iterators.

Return non-zero error code on failure, zero for success and pointer to user's data

### dque_push_back() – Insert new qnode with the user's data onto the back of the queue

dque_push_back(

| | | |
|---|---|---|
| dque_qhead | *queue, | queue with element data |
| void | *data ) | data to insert for new qnode |

This function checks the free list, and if empty, calls as dque_myalloc() to allocate more qnodes. The function then removes a qnode from the free list, sets the data pointer to the user's data, and then inserts the qnode at the end (back) of the queue.

Return non-zero error code on failure, zero for success.

### dque_push_front() - Insert new qnode with the user's data onto the front of the queue

dque_push_back(

| | | |
|---|---|---|
| dque_qhead | *queue, | queue with element data |
| void | *data ) | data to insert for new qnode |

This function checks the free list, and if empty, calls as dque_myalloc() to allocate more qnodes. The function then removes a qnode from the free list, sets the data pointer to the user's data, and then inserts the qnode at the front of the queue.

Return non-zero error code on failure, zero for success.

## Compiling the DQUE Library

The DQUE library was developed and tested using the gcc compiler (built by MYSYS2 project) 9.2.0. However, the level of coding was deliberately kept to an earlier level of C, with two exceptions, to allow the library to be used in older C environments. Beginning with the C11 specification, a pre-processor keyword called '_Generic' was added to allow pseudo polymorphic types. See the section called "Polymorphism and Short Names" for how the _Generic keyword was used in this library. The other exception was the pre-processor keyword '__VA_ARGS__' that was added with the C99 specification.

Another reason for holding down the level C used in the library is the Microsoft MSVC++ compiler. According to here the MSVC++ has limited support for C, officially supporting only ANSI C89 standard, although much of the C90 standard was added in Visual C++ 2013. However, the support for the keyword __VA_ARGS__ does not conform to the C standard as explained here. Anyone using the MSVC++ compiler with this library should compile the library with the equivalent of the –std=c90 flag described below.

The library and the test program were compiled using different C standards with the gcc –std flag using options c11, c99, c90, and –ansi with the –Wpadantic . Both the library and test program compiled without warnings and the test program ran successfully in each case.

## Polymorphism and Short Names

The C programming language does not support polymorphism, period. That said, the C11 standard introduced a new pre-processor identifier called '_Generic' which allows for pseudo polymorphic types. See the C11 standard for a complete description of this identifier. In this library, the _Generic identifier allows the pre-processor to switch between different functions based on the function's argument type. This paper will describe a hypothetical example that will illustrate some of the implications of using this identifier as future libraries are added to this library.

All of the function names in this library start with 'dque_'to avoid any name resolution problems. However, longer names are cumbersome to use if they are not required. Some pre-processor macros have been included in the library header file that will allow the user to use the short names under most circumstances. Now assume there is another library based on using a heap data structure. Both of these libraries can be used to implement a priority queue, but the heap based queue would be considered the better choice because of performance. However, printing out the elements the queue in order would favor the linked-list implementation. For this example, assume that the user needs to use both types of priority queues in the same file. Looking at just the insert functions, this library has dque_key_insert() and the heap library has heap_key_insert(). Obviously, using the long names will avoid any name resolution problems, but it would be nice to be able to use the short names.

For C environments older than C11, that is without the _Generic identifier, these libraries have a scheme to allow the user to use the short names on one, but only one, library. Each library has an arbitrary number assigned to it for identification. For example, assume the DQUE library is set to one, and the heap library is set to two. There is another #define, called INSERT_DEF, which is set to one of these numbers to select the short names from that library. The library that is selected will expose macros that will allow the short names for that library to be used. The user can set the INSERT_DEF #define explicitly, but if not, then INSERT_DEF will be set to the library of the first included header file. In either case, the user must use the long names for the other library. None of this is required if only one library is being used, and the user can use the short names just by including the one library header file.

For C environments at the C11 level or greater, the library can use the _Generic identifier far more effectively than simple #defines. In order for _Generic identifier to work, it has to be able switch on a unique data type. Since most of the functions take a pointer to a qhead structure, or a pointer to a pointer to a qhead structure, the type definition for the qhead structure was extended to include these two pointer types. However, the functions dque_error() and dque_error_len() do not take either of these pointers, but instead, they take a simple integer error code. To make _Generic work, the individual integers were turned into a uniquely named enumerated type.

If the user is using only one library, the _Generic version of the #defines act much like the older #defines and allow the user to use the short names for that library. The difference is when multiple libraries are used. Because each library uses different qhead structures, or at least a differently named qhead structures, the _Generic #defines can be expanded to include all of the libraries. This means, the user

can use the short names for all of the libraries and the _Generic identifier will keep all of them sorted out correctly. Almost like true polymorphism, but more work.

## The Wrapper Functions

The function in this library emulate, in spirit, some of the *_s() function introduced with C11 in that they all have extensive internal error checking and all of them return an error code. However, functions defined this way can be cumbersome to use. In time honored tradition, wrapper functions can be created to make the functions easier to use depending on the environment. Depending on how the wrapper functions are written, some errors maybe obscured.

The file dque_util.c is not a part of the DQUE library, but does contain some of the more generic wrapper function used in the test program. The first group of wrapper functions are used to make the iterator functions more usable. If there is an error in the wrapped function, the wrapper function will return a NULL iterator. This will stop any iteration, but it is up to the user to detect that there is a problem. If the user does, the error is saved in a global error code variable. If the user does not, the iteration will terminate early without the user taking the error into account. The user could use a counter to make sure that the iteration was complete, but this was not done in the test program.

The second set of wrapper functions are used to provide generic stack, queue, and priority queue routines. Since the appropriate insert function returned an error code, there was no need to create a separate wrapper function and so the _Generic #define could call the insertion function directly. The remaining wrapper function for each group, were created to return something other than the error code of the wrapped function. Again, the test program tends to ignore any problems found in the wrapper functions.

Other implementations of C functions used to build a priority queue will embed the comparison function in the equivalent of the qhead structure. That way, the user does not need to specify the comparison function with every call. This library did not use this structure to increase the flexibility of having different key/comparison function pairs. However, as a proof of concept, the priority queue wrapper functions define a new qhead data structure with the comparison function as a part of the new structure. This makes the pri_push() and pri_pop() functions similar to the stack and queue push and pop routines. However, the new structure required creating new create and destroy functions to create and destroy the new data structure.

These wrapper functions could be improved by the user based on the actual user's environment. For example, the pop functions return a pointer to NULL since that is what is defined as the user's data type stored in the elements, but the user would know what is really being stored in the elements, and cast the return type in the wrapper functions.

## Examples

All of the examples are in the test program in the file test.c. The test program starts by testing some erroneous input, and then builds a queue using static memory to test some static memory errors. However, the test program calls a wrapper function called prtest() which takes a string explaining the test. The expected return code from the called function, and then the return code of the called function. prtest() compares the expected return code with the actual return code, and if equal output a “pass” string. If the return codes are not equal, prtest() outputs the error message after the string explaining the test, and then a “fail” string.

The test program goes on to test queues with data of pointer to integers, pointer to characters (string), and then pointers to a simple data structure of student school record. Because the queue only stores a pointer to the user’s data, that data can be anything. The test program goes on to test every function in the library including the generic stack, queue, and priority queue wrappers. However, since the object of the test program is to test the library, the test of the wrapper functions is not that thorough.

The tests of the priority queue using a student record required a creative key and comparison function.

```
typedef struct {
    int  id;
    char name[20];
    int  year;
    int  classes[7];
} record;

typedef struct {
    int   idx;
    int   val;
    char *str;
} reckey;

int reccomp( void *keyval, void *recval ) {
    reckey *key = (reckey *)keyval;
    record *rec = (record *)recval;
    int     retval = 1;               /* return +val to search whole queue */

    switch (key->idx) {
        case 0: if (key->val == rec->id)              { retval = 0; } break;
        case 1: if (strcmp(key->str, rec->name) == 0) { retval = 0; } break;
        case 2: if (key->val == rec->year)            { retval = 0; } break;
    }

    return (retval);
}
```

The priority queue was built using a ‘for’ loop and the same integer comparison function used with the integer tests. This could be done because the first field of the student record is the integer student ID. The test program goes on to search the queue for a student named Sara West, but since the queue was

ordered based on student ID, the entire queue must be searched. One way is to iterate through the queue and look for Sara West through each iterator pointer. Another way is to use the key/comparison function pair shown above. The structure reckey has variable called 'idx' which is used as an index into which field to use in the comparison function, and other types of variables to use as the 'key'. Since the name is the second field, the index variable is set to 1 (zero indexed) and the variable 'str' is set to "Sara West". However, the real trick to the comparison function is that the function always returns 1 until it finds the correct name of Sara West, and then returns zero to stop the search. By always returning 1, it forces the dque_key_find() function to search the entire queue. This trick can used anytime there is a need to force a search of the entire queue.

## Copyright