Documentation for queue.h and queue.c

Steven Andrews, © 2003-2021

Header

```
#ifndef __queue_h
#define __queue_h

#include <limits.h>
#if !defined(LLONG_MAX) || defined(WIN32)
  typedef long int Q_LONGLONG;
  #define Q_LLONG_MAX LONG_MAX
  #define Q_LLI "%li"
#else
  typedef long long int Q_LONGLONG;
  #define Q_LLONG_MAX LLONG_MAX
  #define Q_LLI "%lli"
#endif


enum Q_types {Qusort,Qvoid,Qint,Qdouble,Qlong};

typedef struct qstruct{
  enum Q_types type;
  void **kv;
  int *ki;
  double *kd;
  Q_LONGLONG *kl;
  int (*keycmp)(void *,void *);
  void **x;
  int n;
  int f;
  int b; } *queue;

#define q_frontkeyV(q) ((q)->b==(q)->f?NULL:(q)->kv[(q)->f])
#define q_frontkeyI(q) ((q)->b==(q)->f?0:(q)->ki[(q)->f])
#define q_frontkeyD(q) ((q)->b==(q)->f?0:(q)->kd[(q)->f])
#define q_frontkeyL(q) ((q)->b==(q)->f?0:(q)->kl[(q)->f])

queue q_alloc(int n,enum Q_types type,int (*keycmp)(void *,void *));
int q_expand(queue q,int addspace);
void q_free(queue q,int freek,int freex);
void q_null(queue q);
int q_enqueue(void *kv,int ki,double kd,Q_LONGLONG kl,void *x,queue q);
int q_push(void *kv,int ki,double kd,Q_LONGLONG kl,void *x,queue q);
int q_insert(void *kv,int ki,double kd,Q_LONGLONG kl,void *x,queue q);
void q_front(queue q,void **kvptr,int *kiptr,double *kdptr,Q_LONGLONG
      *klptr,void **xptr);
int q_pop(queue q,void **kvptr,int *kiptr,double *kdptr,Q_LONGLONG *klptr,void
      **xptr);
int q_length(queue q);
int q_maxlength(queue q);
int q_next(int i,void **kvptr,int *kiptr,double *kdptr,Q_LONGLONG *klptr,void
       **xptr,queue q);

#endif
```

Requires: `stdlib.h, queue.h`

Example program: `LibTest.c`, `Smoldyn`

History:

4/16/95	Written
4/20/95	Modified
1/15/02	Mostly rewritten using CodeWarrior C. Some testing.
1/16/02	Ported to Linux.
3/14/02	Moved `FltCmp` to VoidComp.c.
3/7/03	Added `qmaxlength` and `qnext`.
6/21/04	Added `qexpand` but haven't tested it yet.
11/15/08	Major changes to change from only void* keys to also integer and double keys.
1/14/09	Added long long keys.
1/18/09	Fixed bugs with them.
7/17/10	Changed long long keys again so that they work on Windows.
11/21/22	Made integers into secondary keys.

This implements a queue or stack of user definable length for arbitrary pointers using a circular array. It is also possible to maintain a sorted queue. While speed is slightly sacrificed, these routines are reasonably friendly about full and empty queues. When a queue is full, a new item overwrites the oldest item and the queue stays full. Reading from an empty queue results in a `NULL` returned value and the queue stays empty. The only significant thing to watch out for is that overwritten information is not freed, meaning that the data should not need freeing, the queue should not be the master list of data, or overfilling a queue should be avoided.

A queue often contains two lists, one of keys (`k`) and one of items (`x`). A queue also has a type, which is enumerated in the list `Q_types`. These are:

type	keys	key	items	meaning
`Qusort`	none	none	`void*`	unsorted queue
`Qvoid`	`void*`	`kv`	`void*`	`void*` keys
`Qint`	`int`	`ki`	`void*`	`int` keys
`Qdouble`	`double`	`kd`	`void*`	`double` keys
`Qlong`	`Q_LONGLONG`	`kl`	`void*`	`Q_LONGLONG` keys

The type of the queue is specified when it is allocated and may not be changed thereafter. Note that void* is a general pointer type, which can be used for strings, pointers to structures, or set to `NULL`s (for a sorted queue with a single list, set each item to `NULL`). The appropriate key variable that is listed above should be used for a given type of queue; for queues that don't use integer keys, the integer is used as a secondary key to allow subsorting among identical primary keys. The type `Q_LONGLONG` is defined in the header file to be a `long long int` if such a thing is defined locally, or to just be a `long int` if not.

If the queue is of type `Qvoid`, then a function called `keycmp` is required for comparing keys. Key comparison routines in the VoidComp.c library include

`FltCmp`, `IntCmp`, and `StringCmp`, and although the first two of these are not advised because they cause compiler warnings when used on 64-bit machines (they are also unnecessary due to the `Qint` and `Qdouble` queue types).

In the queue structure, `n`, `f`, and `b` are queue indices: `n` is the total number of spaces in the list, including usable spaces and one additional space so full lists can be distinguished from empty lists, `f` is the address of the front of the list, and `b` is one more than the address of the last element (a new element is added at position `b`). While it shouldn't generally be necessary, here is a useful code fragment for looking at each item of a queue, from front to back:

```
for(i=q->f;i!=q->b;i=(i+1)%q->n) x=q->x[i];
```

Macros

```
#define q_frontkeyV(q) ((q)->b==(q)->f?NULL:(q)->kv[(q)->f])
#define q_frontkeyI(q) ((q)->b==(q)->f?0:(q)->ki[(q)->f])
#define q_frontkeyD(q) ((q)->b==(q)->f?0:(q)->kd[(q)->f])
#define q_frontkeyL(q) ((q)->b==(q)->f?0:(q)->kl[(q)->f])
```

These macros return `NULL` or 0 if the queue is empty and otherwise return the key of the item at the front. The queue is not affected. The macro which is used needs to corresponds to the queue type.

Functions

```
queue q_alloc(int n,enum Q_types qtype,int (*keycmp)(void *,void *));
```

`q_alloc` creates a new empty queue, of type `qtype`, with `n` usable spaces. It allocates the necessary memory and initializes all spaces to 0s or `NULL`s, as appropriate. If the queue is of type `Qvoid`, and it is to be a sorted queue, then `keycmp` needs to be sent in as a function that can compare the contents of two void*s. Otherwise, send in `keycmp` as `NULL`. The return value is a pointer to a queue unless memory allocation failed, in which case `NULL` is returned. `n` is allowed to be any value larger than or equal to zero. While a queue with 0 usable spaces is not likely to be very useful, all routines here are designed to work properly with this input.

```
int q_expand(queue q,int addspace);
```

`q_expand` makes a queue longer by replacing the current arrays with ones that are longer than the current ones by `addspace`. Items and keys are copied and indices are maintained. `addspace` is allowed to be negative to shrink a queue, although an error of 2 is returned if an attempt is made to shrink it smaller than 0 usable spaces. If it is shrunk to be smaller than the current queue length, the oldest (farthest from the front) items are dropped. An error of 1 is returned if memory could not be allocated; otherwise a return of 0 indicates no errors. Note that `qexpand` cannot tell a full queue from an empty queue, so it should be called before the queue is full; for example, expand the queue when `qinsert` or `qpush` return 1.

```
void q_free(queue q,int freek,int freex);
```

Frees a queue. If `freex` or `freek` are 1, items and/or keys are freed as well.

```
void q_null(queue q);
```

Sets the indicies to indicate an empty queue. It does not free any memory or erase the contents of cells.

`int q_enqueue(void *kv,int ki,double kd,Q_LONGLONG kl,void *x,queue q);`

Adds an item (`x`) and a key (`kv`, `ki`, `kd`, or `kl`) to the back of the queue. The key with the type that matches the queue type is looked at. If a secondary key is desired, use the integer entry for that one; the others should be set to 0 or `NULL`, as appropriate. It returns the total number of usable spaces remaining in the queue, or -1 if the routine just overwrote an old element.

`int q_push(void *kv,int ki,double kd,Q_LONGLONG kl,void *x,queue q);`

Adds an item and a key to the front of the queue. The key with the type that matches the queue type is looked at. If a secondary key is desired, use the integer entry for that one; the others should be set to 0 or `NULL`, as appropriate. As with `enqueue`, it returns the number of remaining spaces or -1 if the routine just overwrote an old element.

`int q_insert(void *kv,int ki,double kd,Q_LONGLONG kl,void *x,queue q);`

Inserts an item in the correct place of a sorted queue, where the keys are in ascending order from front to back. If an element is inserted to an already full queue, the item at the back of the list is dropped from the queue. The key with the type that matches the queue type is looked at. If a secondary key is desired, use the integer entry for that one; the others should be set to 0 or `NULL`, as appropriate. If an existing item has the same primary key, then the secondary key is used for further sorting. It that's equal too, then the new item is added farther toward the back of the queue. As with `q_enqueue`, it returns the number of remaining spaces or -1 if the routine just overwrote an old element.

`void q_front(queue q,void **kvptr,int *kiptr,double *kdptr,Q_LONGLONG *klptr,void **xptr);`

Returns the item and key at the front, but doesn’t remove them from the list. Only the appropriate k*ptr is filled in (plus integer secondary key if appropriate). Either `k*ptr` or `xptr` may be `NULL` if that result is not wanted. If the queue is empty, both returned values are `NULL`.

`int q_pop(queue q,void **kvptr,int *kiptr,double *kdptr,Q_LONGLONG *klptr,void **xptr);`

`qpop` is just like `qfront`, but it also removes the frontmost item and key. Also, it returns the number of items in the queue after the frontmost one is popped. Returns -1 if the queue was already empty when the function was called.

`int q_length(queue q);`

Returns the number of items stored in the queue. If the queue is full, `qlength` returns 0, since it can't distinguish a full queue from an empty one.

`int q_maxlength(queue q);`

Returns the maximum number of items that can be stored in this queue, which is just `q->n-1`.

`int q_next(int i,void **kvptr,int *kiptr,double *kdptr,Q_LONGLONG *klptr,void **xptr,queue q);`

`q_next` is a useful command for looking at each item in the queue without changing the queue. `i` is the index of the previous item in the queue, or is `-1` to indicate that the next item should be the first one. The return value is the index of the next item, along with pointers to the key and item stored there. Here is a typical use for this command:

```
i=-1;
while((i=qnext(i,NULL,&ki,NULL,NULL,&x,q))>=0) {
        print out ki and x }
```

If the key and/or item are not wanted, then pass a `NULL` pointer into `k*ptr` or `xptr`.