NAME

quickselect − multiple selection of order statistics and sorting

SYNOPSIS

#include <quickselect.h>

unsigned int quickselect_options(void);

void quickselect(void *base, size_t nmemb, size_t size, int (*compar)(const void *, const void *), size_t *pk, size_t nk, unsigned int options);

errno_t quickselect_s(void *base, rsize_t nmemb, rsize_t size, int (*compar)(const void *, const void *, void *), void *context, size_t *pk, size_t nk, unsigned int options);

void QSORT_FUNCTION_NAME(void *base, size_t nmemb, size_t size, int (*compar)(const void , const void *));

errno_t QSORT_S_FUNCTION_NAME(void *base, rsize_t nmemb, rsize_t size, int (*compar)(const void *, const void *, void *), void *context);

DESCRIPTION

The **quickselect** function implements multiple selection of order statistics. Given an array **pk** of **nk** *size_t* elements representing 0−based order statistic ranks, **quickselect** partially orders the array **base** (having **nmemb** elements of size **size**) such that the specified order statistics are in−place in the array. If **pk** is *NULL* or **nk** is 0UL, a full sort of the array is performed.

Function **compar** is provided by the caller, and should return a value less than zero, equal to zero, or greater than zero when the array element pointed to by the first argument is less than, equal to, or greater than the array element pointed to by the second argument.

The **options** argument affects operation by bitwise or'ing any of the following components:

QUICKSELECT_STABLE
: causes sorting or selection to preserve partial order present in the input. There is a substantial performance penalty; the default operation does not guarantee preservation of partial order.

QUICKSELECT_OPTIMIZE_COMPARISONS
: uses minimum-comparison methods and is suitable if the comparison function **compar** is known to be relatively expensive. The default operation attempts to minimize run−time for simple comparisons.

QUICKSELECT_INDIRECT
: allocates an array of pointers and initializes it to point to elements in the **base** array. Sorting or selection then proceeds by dereferencing the pointers for comparisons (using the user-supplied **compar** function, and caching dereferenced pointers where practical), finally rearranging **base** array elements and freeing the allocated pointers. Direct sorting or selection is used if memeory cannot be allocated for the pointers. Caching of dereferenced pointers provides a tiny performance gain compared to caller indirection (the comparison function in that case would do the dereferencing, and caching (e.g. the pivot element used for partitioning) would not be possible). Rearranging **base** array elements after sorting the pointers saves considerable data movement (moving only pointers during sorting or selection, and moving each **base** array element one time to its correct position is considerably less overall data movement than moving **base** array elements during sorting or selection if **size** is appreciably larger than a pointer). However, the final data movement has poor locality of access for random inputs, which may cause poor performance when the data array (the product of element **size** and the number of elements **nmemb**) is large relative to cache size. The primary advantage of internal indirection is caller convenience: the caller provides a normal **compar** function rather than one which dereferences pointers; allocating, initializing, and freeing the pointers is handled transparently to the caller, fallback to direct sorting is also automatic, and the final *O(N)* **base** array reordering is efficient (though not cache-friendly).

bits in QUICKSELECT_NETWORK_MASK

specify the ability to use sorting networks for arrays of size 3 through 12 elements, corresponding to bit 0x01 << size. A sorting network is always used to sort sub−arrays of size 2. Sorting networks for arrays of size 7 or larger are not applicable when **QUICKSELECT_STABLE** is set, and are silently ignored. When **QUICKSELECT_OPTIMIZE_COMPARISONS** is set, only the size 2 sorting network is used; all others are silently ignored. Sorting networks are fast due to low overhead, but are unable to take advantage of pre−existing order in the input (e.g. already-sorted input).

Options may have been limited at library compile time to more restrictive values than those found in *quickselect.h*. Function **quickselect_options** may be called to determine the options available at run-time: bits not present in the returned value provided from **quickselect_options** should not be included in the *options* argument to **quickselect**; they will cause **quickselect** to fail and set *errno* to **EINVAL**.

RETURN VALUES

none for quickselect and QSORT_FUNCTION_NAME. If **__STDC_WANT_LIB_EXT1__** is defined with non-zero value when *quickselect.h* is included, quickselect_s and QSORT_S_FUNCTION_NAME are provided, which return zero on normal exection and non-zero if there is an argument error. Arguments **nmemb** and **size** are compared to **RSIZE_MAX**, and **compar** is compared to **NULL**. In addition, the comparison function **compar** is expected to take a third argument, which is provided by the **contect** argument.

ERRORS

If **base** is *NULL*, **nmemb** is 0UL, **size** is 0UL, **compar** is *NULL*, or **options** requests unavailable options, the global variable *errno* is set to **EINVAL**.

EXAMPLES

```
#include "quickselect.h"
#include <errno.h>
size_t karray[2];
unsigned int options;
karray[0] = (nmemb-1UL)/2UL;
karray[1] = nmemb/2UL;
options = quickselect_options();

quickselect(base, nmemb, size, compar, NULL, karray, 2UL, QUICKSELECT_NETWORK_MASK & options);
```

places the median (**nmemb** odd) or medians (**nmemb** even) in the middle element(s) of the array pointed to by **base.** Refer to the BUGS and CAVEATS section regarding duplicated order statistic ranks.

```
quickselect(base, nmemb, size, compar, NULL, NULL, 0UL, QUICKSELECT_NETWORK_MASK & options);
```

sorts the array, and is equivalent to

```
QSORT_FUNCTION_NAME(base, nmemb, size, compar);

errno = 0;
quickselect(base, nmemb, size, compar, NULL, NULL, 0UL, QUICKSELECT_NETWORK_MASK | QUICKSELECT_STABLE);
```

requests a stable sort; if that option is unavailable, the array **base** will be unaltered, and *errno* will be set to **EINVAL**.

APPLICATION USAGE

If the macro **QSORT_FUNCTION_NAME** is defined before *quickselect.h* is included when compiling the *quickselect.c* source, a sorting function with the same semantics as **qsort** is generated, using the specified

name. A library implementation of **qsort** may be generated by defining **QSORT_FUNCTION_NAME** as **qsort**.

RATIONALE

While many libraries include a standard **qsort** function, those **qsort** implementations may tend to quadratic performance on adverse inputs. Many implementations exhibit poor performance for some types of structured input sequences, such as reverse-sorted or rotated sequences. Most **qsort** implementations provide no guarantee of stablilty (in the sense of preservation of partial order), and there is no means to adjust the algorithm to compensate for expensive comparisons. Few libraries provide a function for selection of order statistics. Those libraries that do provide a selection function usually only permit selection of a single order statistic per function call.

BUGS and CAVEATS

Array **pk** may be sorted by **quickselect** and therefore initially unsorted order statistic ranks may be permuted by a call to **quickselect**. It is recommended (but not required) that the order statistics array **pk** be supplied in sorted order.

If array **pk** contains duplicated ranks, those duplicates will be ignored during processing and will be grouped together by sorting after processing. This may be expensive if a large number of order statistics are specified and there is at least one duplicate. It is recommended (but not required) that the order statistics array **pk** contain no duplicates.

quickselect has expected and worst−case linear complexity for finding a single order statistic. Worst−case non-stable selection of multiple order statistics is linearithmic. **quickselect** has expected and worst−case linearithmic complexity for non-stable sorting.

When stable sorting or selection is specified by setting **QUICKSELECT_STABLE**, selection may become linearithmic and sorting may become $O(N \log^2 N)$ due to additional data movement (the complexity of comparisons is unchanged). Moreover, locality of access may be poor, resulting in performance deterioration due to cache size effects. Sorting and selection may remain in−place, or additional size−related temporary memory may be used if available.

Compiled library code might have been built with **QUICKSELECT_STABLE** and/or **QUICKSELECT_NETWORK_MASK** set to values other than those which appear in *quickselect.h*. If the library was built without the option to sort or select while maintaining partial order stability, specifying **QUICKSELECT_STABLE** in *options* will not be effective; quickselect will set *errno* to **EINVAL** and the array will not be sorted. Similarly, support for sorting network sizes may have been limited when the library object files were built. It is strongly recommended to call *quickselect_options* to determine which option bits are available. Compiled options may also be determined by running external programs such as *what* or *ident* on the library archive containing *quickselect*.

FUTURE DIRECTIONS

none

SEE ALSO

qsort, what, ident

CHANGE HISTORY

Function implementation initial version June 2016. Implementation backward−compatible updates through December 2017.

Manual page initial version January 2017. Latest manual page update December 2017.

AUTHOR

Bruce Lilly <bruce.lilly@gmail.com>

LICENSE

The zlib/libpng license is a recognized "free" software license by the Free Software Foundation: https://directory.fsf.org/wiki/License:Zlib

********** Copyright notice (part of the license) **********

********** (end of license) **********

You may send bug reports to bruce.lilly@gmail.com with subject "quickselect".