ChonkSort

Generated by Doxygen 1.8.17

Chapter 1

BMSRS

Chapter 2

Namespace Index

2.1 Namespace List

Here is a list of all documented namespaces with brief descriptions:

Chapter 3

Namespace Documentation

3.1 Arrays::anonymous_namespace{sort.cpp} Namespace Reference

Functions

- template<typename E >
  void parallelPrefixFill (E &x)
- template<> void **parallelPrefixFill**< **uint64_t** > (uint64_t &x)
- template<> void **parallelPrefixFill**< **uint32_t** > (uint32_t &x)
- template<> void **parallelPrefixFill**< **uint16_t** > (uint16_t &x)
- template<> void **parallelPrefixFill**< **uint8_t** > (uint8_t &x)
- template<typename E >
  constexpr int bitScanRev (E l)
- template<typename E >
  constexpr void **swap** (E *const i, E *const j)
- template<typename E , bool Leftmost>
  void iSort (E *const low, E *const high)
- template<typename E >
  void siftDown (E *const a, const int i, const int size)
- template<typename E >
  void hSort (E *const low, E *const high)
- template<typename E , bool Leftmost = true>
  void qSort (E *const low, E *const high, const int height)

Variables

- constexpr uint32_t **DPQSInsertionThreshold** = 32
- constexpr uint64_t DeBruijn64
- constexpr uint8_t **DeBruijnTableF** [ ]

3.1.1 Function Documentation

3.1.1.1 bitScanRev()

```
template<typename E >
constexpr int Arrays::anonymous_namespace{sort.cpp}::bitScanRev (
            E l )  [constexpr]
```

bitScanReverse

Authors

Kim Walisch

Mark Dickinson

Parameters

bb	bitboard to scan @precondition bb != 0

Returns

index (0..63) of most significant one bit

3.1.1.2 hSort()

```
template<typename E >
void Arrays::anonymous_namespace{sort.cpp}::hSort (
            E *const low,
            E *const high )  [inline]
```

Heap Sort

Classical heap sort that sorts the given range in ascending order, building a max heap and continuously sifting/swapping the max element to the previous rightmost index.

Author

Ellie Moore

Template Parameters

E	the element type

Parameters

low	a pointer to the leftmost index
high	a pointer to the rightmost index

3.1.1.3 iSort()

```
template<typename E , bool Leftmost>
void Arrays::anonymous_namespace{sort.cpp}::iSort (
            E *const low,
            E *const high )  [inline]
```

Insertion Sort

Classical ascending insertion sort packaged with a "pairing" optimization to be used in the context of Quick Sort. This optimization is used whenever the portion of the array to be sorted is padded on the left by a portion with lesser elements. The fact that all of the elements on the left are automatically less than the elements in the current portion allows us to skip the costly lower boundary check in the nested loops.

Authors

Josh Blosh

John Bently

Ellie Moore

Template Parameters

E	the element type

Parameters

low	a pointer to the leftmost index
high	a pointer to the rightmost index
isLeftmost	whether or not this subarray is a left-most partition.

3.1.1.4 parallelPrefixFill()

```
template<typename E >
void Arrays::anonymous_namespace{sort.cpp}::parallelPrefixFill (
            E & x )
```

Fill trailing bits using prefix fill.

```
Example:
      10000000 » 1
    = 01000000 | 10000000
    = 11000000 » 2
    = 00110000 | 11000000
    = 11110000 » 4
    = 00001111 | 11110000
    = 11111111
```

Template Parameters

E	The type

Parameters

x	The integer

3.1.1.5 qSort()

```
template<typename E , bool Leftmost = true>
void Arrays::anonymous_namespace{sort.cpp}::qSort (
             E *const low,
             E *const high,
             const int height )
```

Introspective Multi-Pivot Quick Sort

This sort uses a recursive pattern to re-arrange data in ascending order. Whenever the size of the current interval falls below the threshold of 32 elements, insertion sort is used. On return, the data in the current interval will be in sorted order.

<a href = "https://codeblab.com/wp-content/uploads/2009/09/DualPivotQuicksort.pdf"

Here is the original Dual-Pivot Quicksort by Vladimir Yaroslavskiy.

<a href = "https://www.researchgate.net/publication/289974363_Multi-Pivot_Quicksort_Theory_and_Experiments"

Here is the three pivot partition algorithm by Kushagra, Lopez-Ortix, Munro, and Qiao.

An optimized Dual-Pivot Quicksort by Bloch and Bently can be found in the open-source jdk within the java.util library.

This sort is derived from the resources listed above.

Optimizations:

1. Use Smart Pivot Selection

   Pivots are carefully selected from a set of five candidates. These candidates are the leftmost element, the element at the first tercile, the middle element, the element at the second tercile, and the rightmost element. All five of these candidates are sorted in place. If none are equal, three pivots are chosen from the middle. If an outside pair is equal, two pivots are selected from the tercile indices. However, if a middle pair is equal, then a single pivot will be selected from the middle and traditional Quick Sort will be used. This process helps to select pivots that divide the data as evenly as possible, mitigating the possibility of the worst-case $O(N^2)$ time complexity.

2. Use Four-Way or Three-Way Partitioning When Possible

   Multi-way partitioning results in smaller intervals to partition in recursive calls. This makes cache hits more likely. Multi-way partitioning also helps to trim the height of the sort tree.

3. Ignore In-Order and Equal Elements

   Runs of elements will be ignored.

4. Keep the Middle Portion(s) as Small as Possible

   After partitioning with more than one pivot, if the middle portion comprises more than 2/3 of the current interval, we swap all elements equal to the pivots out of the way before recursively sorting.

5. Use Insertion Sort On Little Sub-Arrays

   Insertion Sort is small and relies heavily on swapping. Fewer instructions help to alleviate the overhead that plagues sorting algorithms, and consistent swapping leverages CPU caching to its fullest extent. These features allow insertion sort to beat the runtimes of O(nlogn) sorts on small sets of data.

6. Use Heap Sort to prevent Quicksort's worst-case time complexity.

   This sort is an introspective sort&mdash a sort that switches to a guaranteed O(nlogn) algorithm whenever time complexity is becoming quadratic. Here, Heap Sort is used to avoid the overhead of auxilliary storage.

Authors

Vladimir Yaroslavskiy

Josh Bloch

Jon Bently

Shrinu Kushagra

Alejandro Lopez-Ortiz

J. Ian Munro

Aurick Qiao

Ellie Moore

Template Parameters

E	the element type
Leftmost	whether we are considering a leftmost partition.

Parameters

low	a pointer to the leftmost index
high	a pointer to the rightmost index
Leftmost	whether or not this subarray is a left-most partition.

3.1.1.6 siftDown()

```
template<typename E >
void Arrays::anonymous_namespace{sort.cpp}::siftDown (
            E *const a,
            const int i,
            const int size )  [inline]
```

A generic "sift down" method (AKA max-heapify)

Template Parameters

E	the element type

Parameters

a	the pointer to the base of the current sub-array
i	the starting index
size	the size of the current sub-array

3.1.2 Variable Documentation

3.1.2.1 DeBruijn64

```
constexpr uint64_t Arrays::anonymous_namespace{sort.cpp}::DeBruijn64  [constexpr]
```

Initial value:

```
=
            0x03F79D71B4CB0A89L
```

The DeBruijn constant.

3.1.2.2 DeBruijnTableF

```
constexpr uint8_t Arrays::anonymous_namespace{sort.cpp}::DeBruijnTableF[]  [constexpr]
```

Initial value:

```
= {
            0,  47,  1, 56, 48, 27,  2, 60,
            57, 49, 41, 37, 28, 16,  3, 61,
            54, 58, 35, 52, 50, 42, 21, 44,
            38, 32, 29, 23, 17, 11,  4, 62,
            46, 55, 26, 59, 40, 36, 15, 53,
            34, 51, 20, 43, 31, 22, 10, 45,
            25, 39, 14, 33, 19, 30,  9, 24,
            13, 18,  8, 12,  7,  6,  5, 63
        }
```

Index