

Comparing Sorting Algorithms

Generated by Doxygen 1.9.0

1 Data Structure Index	1
1.1 Data Structures	1
2 File Index	3
2.1 File List	3
3 Data Structure Documentation	5
3.1 sorts Struct Reference	5
3.1.1 Detailed Description	5
3.1.2 Field Documentation	5
3.1.2.1 name	5
3.1.2.2 sortProc	5
4 File Documentation	7
4.1 sort-comparisons.c File Reference	7
4.1.1 Detailed Description	8
4.1.2 Macro Definition Documentation	8
4.1.2.1 numAlgs	8
4.1.3 Typedef Documentation	8
4.1.3.1 sorts	8
4.1.4 Function Documentation	9
4.1.4.1 checkAscending()	9
4.1.4.2 checkAscValues()	9
4.1.4.3 heapSort()	9
4.1.4.4 hybridQuicksort()	10
4.1.4.5 hybridQuicksortHelper()	11
4.1.4.6 impPartition()	12
4.1.4.7 insertionSort()	12
4.1.4.8 main()	13
4.1.4.9 merge()	13
4.1.4.10 mergeSort()	14
4.1.4.11 percDown()	14
4.1.4.12 selectionSort()	15
Index	17

Chapter 1

Data Structure Index

1.1 Data Structures

Here are the data structures with brief descriptions:

sorts	5
---------------------------------	---

Chapter 2

File Index

2.1 File List

Here is a list of all files with brief descriptions:

sort-comparisons.c	7
--	---

Chapter 3

Data Structure Documentation

3.1 sorts Struct Reference

Data Fields

- char * [name](#)
- void(* [sortProc](#))(int[], int)

3.1.1 Detailed Description

structure to identify both the name of a sorting algorithm and * a pointer to the function that performs the sort * the main function utilizes this struct to define an array of * the sorting algorithms to be timed by this program. *

3.1.2 Field Documentation

3.1.2.1 name

char* name
the name of a sorting algorithm as text

3.1.2.2 sortProc

void(* sortProc) (int[], int)
the procedure name of a sorting function
The documentation for this struct was generated from the following file:

- [sort-comparisons.c](#)

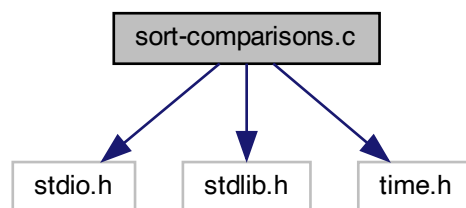
Chapter 4

File Documentation

4.1 sort-comparisons.c File Reference

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
```

Include dependency graph for sort-comparisons.c:



Data Structures

- struct [sorts](#)

Macros

- #define [numAlgs](#) 5

Typedefs

- typedef struct [sorts](#) [sorts](#)

Functions

- void [selectionSort](#) (int a[], int n)
- void [insertionSort](#) (int a[], int n)
- int [impPartition](#) (int a[], int size, int left, int right)
- void [hybridQuicksortHelper](#) (int a[], int size, int left, int right)
- void [hybridQuicksort](#) (int a[], int n)
- void [merge](#) (int alnit[], int aRes[], int alnitLength, int start1, int start2, int end2)
- void [mergeSort](#) (int initArr[], int n)

- void `percDown` (int array[], int hole, int size)
- void `heapSort` (int a[], int n)
- char * `checkAscValues` (int a[], int n)
- char * `checkAscending` (int a[], int n)
- int `main` ()

4.1.1 Detailed Description

Remarks

program times several sorting algorithms on data sets of various sizes *

•

this version includes code for straight selection insertion sorts * stubbs are provided for other sorting algoritms, including * hybrid quicksort, merge sort and heap sort *

•

Author

Henry M. Walker *

Remarks

Assignment Comparison of Sorting Algorithms *

•

Date

August 15, 2022 *

•

Remarks

References *

Dynamic Programming: Anany Levitin, "The Design and * and Analysis of Algorithms", Second Edition, * Sections 3.1 (Selectino Sort), 4.1 (Insertion Sort), * 5.1 (Mergesort), 5.2 (Quicksort), 6.4 (Heapsort) *

•

People participating with Problem/Progra Discussions: * Marcia Watts *

•

4.1.2 Macro Definition Documentation

4.1.2.1 numAlgs

```
#define numAlgs 5
```

4.1.3 Typedef Documentation

4.1.3.1 sorts

```
typedef struct sorts sorts
```

structure to identify both the name of a sorting algorithm and * a pointer to the function that performs the sort * the main function utilizes this struct to define an array of * the sorting algorithms to be timed by this program. *

4.1.4 Function Documentation

4.1.4.1 checkAscending()

```
char* checkAscending (  
    int a[],  
    int n )
```

check all array elements are in non-descending order *

Parameters

<i>a</i>	the array to be sorted *
<i>n</i>	the size of the array * returns "ok" if array elements in non-descending order; "NO" otherwise *

Here is the caller graph for this function:



4.1.4.2 checkAscValues()

```
char* checkAscValues (  
    int a[],  
    int n )
```

check all array elements have values 0, 2, 4, . . ., 2(n-1) *

Parameters

<i>a</i>	the array to be sorted *
<i>n</i>	the size of the array * returns "ok" if array contains required elements; "NO" if not *

Here is the caller graph for this function:



4.1.4.3 heapSort()

```
void heapSort (  

```

```

    int a[],
    int n )
heap sort, main function *
```

Parameters

<i>a</i>	the array to be sorted *
<i>n</i>	the size of the array *

Postcondition

the first *n* elements of *a* are sorted in non-descending order *

Here is the call graph for this function:



Here is the caller graph for this function:

**4.1.4.4 hybridQuicksort()**

```

void hybridQuicksort (
    int a[],
    int n )
```

hybrid quicksort, main function * algorithmic elements * random pivot used in partition function * insertion used for small array segments *

Parameters

<i>a</i>	the array to be sorted *
<i>n</i>	the size of the array *

Postcondition

the first n elements of a are sorted in non-descending order *

Here is the call graph for this function:



Here is the caller graph for this function:

**4.1.4.5 hybridQuicksortHelper()**

```

void hybridQuicksortHelper (
    int a[],
    int size,
    int left,
    int right )
  
```

Quicksort helper function * algorithmic elements * quicksort used when array segments $>$ variable breakQuicksortToInsertion * insertion sort used for small array segments *

Parameters

<i>a</i>	the array to be processed *
<i>size</i>	the size of the array *
<i>left</i>	the lower index for items to be processed *
<i>right</i>	the upper index for items to be processed *

Postcondition

sorts elements of *a* between *left* and *right* *

Here is the caller graph for this function:

**4.1.4.6 impPartition()**

```

int impPartition (
    int a[],
    int size,
    int left,
    int right )
  
```

Improved Partition function * uses *a[left]* as pivot value in processing * algorithmic elements * random pivot utilized
 * swaps only when required by finding misplaced large and small elements *

Parameters

<i>a</i>	the array to be processed *
<i>size</i>	the size of the array *
<i>left</i>	the lower index for items to be processed *
<i>right</i>	the upper index for items to be processed *

Postcondition

elements of *a* are rearranged, so that * items between *left* and index *mid* are $\leq a[\text{mid}]$ * items between dex
mid and *right* are $\geq a[\text{mid}]$ *

Returns

mid *

4.1.4.7 insertionSort()

```

void insertionSort (
    int a[],
    int n )
  
```

insertion sort *

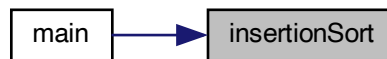
Parameters

<i>a</i>	the array to be sorted *
<i>n</i>	the size of the array *

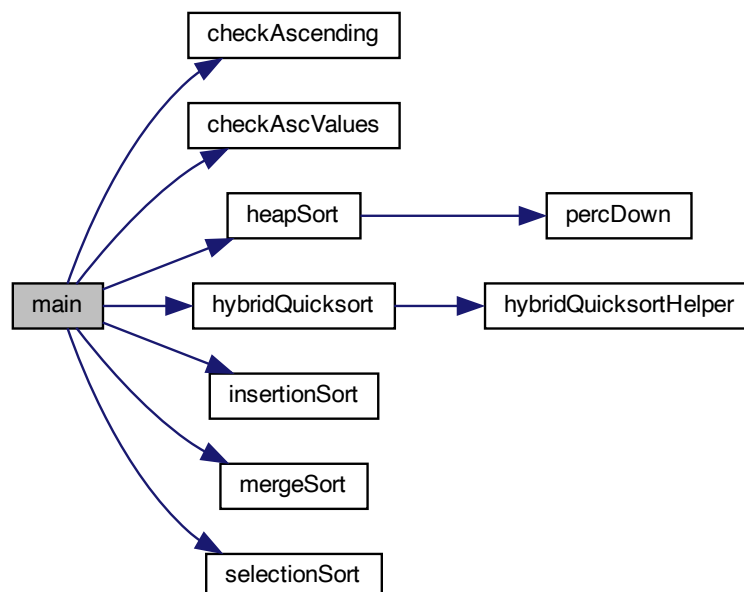
Postcondition

the first *n* elements of *a* are sorted in non-descending order *

Here is the caller graph for this function:

**4.1.4.8 main()**

`int main ()`
driver program for testing and timing sorting algorithms * Here is the call graph for this function:

**4.1.4.9 merge()**

```
void merge (
    int aInit[],
    int aRes[],
    int aInitLength,
    int start1,
    int start2,
    int end2 )
```

merge sort helper function *

Parameters

<i>aInit</i>	source array for merging *
<i>aRes</i>	target array for merging *
<i>aInitLength</i>	the size of the array segment to be merged *
<i>start1</i>	the first index of the first array segment to be merged *
<i>start2</i>	the first index of the second array segment to be merged *
<i>end2</i>	the last index of the second array segment to be merged *

Postcondition

elements `aInit[start1]..aInit[start1+mergeSize]` merged with `* aInit[start2]..aInit[end2] *` with the result placed in `aRes` * Note: it may be that `start2 >= aInit.length`, in which case, only the `* valid part of aInit[start1]` is copied *

4.1.4.10 mergeSort()

```
void mergeSort (
    int initArr[],
    int n )
```

merge sort helper function *

Parameters

<i>initArr</i>	the array to be sorted *
<i>n</i>	the size of the array *

Postcondition

the first `n` elements of `a` are sorted in non-descending order *

Here is the caller graph for this function:



4.1.4.11 percDown()

```
void percDown (
    int array[],
    int hole,
    int size )
```

percDown function *

Parameters

<i>array</i>	the array to be made into a heap, starting at hold *
<i>hole</i>	base of subtree for start of processing *
<i>size</i>	the size of the array *

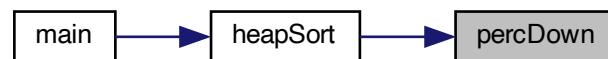
Precondition

all nodes in left and right subtrees of the hole node are heaps *

Postcondition

all nodes in the tree from the hole node downward form a hea *

Here is the caller graph for this function:

**4.1.4.12 selectionSort()**

```

void selectionSort (
    int a[],
    int n )
straight selection sort *
  
```

Parameters

<i>a</i>	the array to be sorted *
<i>n</i>	the size of the array *

Postcondition

the first n elements of a are sorted in non-descending order *

Here is the caller graph for this function:



Index

- checkAscending
 - sort-comparisons.c, [9](#)
- checkAscValues
 - sort-comparisons.c, [9](#)
- heapSort
 - sort-comparisons.c, [9](#)
- hybridQuicksort
 - sort-comparisons.c, [10](#)
- hybridQuicksortHelper
 - sort-comparisons.c, [11](#)
- impPartition
 - sort-comparisons.c, [12](#)
- insertionSort
 - sort-comparisons.c, [12](#)
- main
 - sort-comparisons.c, [13](#)
- merge
 - sort-comparisons.c, [13](#)
- mergeSort
 - sort-comparisons.c, [14](#)
- name
 - sorts, [5](#)
- numAlgs
 - sort-comparisons.c, [8](#)
- percDown
 - sort-comparisons.c, [14](#)
- selectionSort
 - sort-comparisons.c, [15](#)
- sort-comparisons.c, [7](#)
 - checkAscending, [9](#)
 - checkAscValues, [9](#)
 - heapSort, [9](#)
 - hybridQuicksort, [10](#)
 - hybridQuicksortHelper, [11](#)
 - impPartition, [12](#)
 - insertionSort, [12](#)
 - main, [13](#)
 - merge, [13](#)
 - mergeSort, [14](#)
 - numAlgs, [8](#)
 - percDown, [14](#)
 - selectionSort, [15](#)
 - sorts, [8](#)
- sortProc
 - sorts, [5](#)

- sorts, [5](#)
 - name, [5](#)
 - sort-comparisons.c, [8](#)
 - sortProc, [5](#)