Report - assignment 3

Camilla Kristiansen Birkelund IDATT2101 Sept. 11, 2022

Assignment description

The assignment was to compare single pivot and double pivot quicksort by implementing both algorithms in a single test program, then measure the times to see which solution gives the best results for a table of random numbers, a table with several duplicates and a table that is already sorted.

Implementation

Most of the source code was provided by Hafting & Ljosand's book (2014) and the website "GeeksForGeeks" (Elhaiani, 2022), where some changes were made to the source code retrieved from the website according to the assignments suggestions. The libraries chrono, ctime, iostream and stdlib were used to implement a way to measure the time used for each quicksort algorithm, and to efficiently create lists of integers.

When programming in c++, the program can handle a large quantity of elements in the tables, as long as pointers to the heap are used. This is implemented in the main method, as this is not something that is done automatically.

```
int* list1 = new int[SIZE];
int* list2 = new int[SIZE];
```

Figure 1: Pointers to the heap.

A for-loop was used to fill the tables (referred to as list1 and list2 in the source code) with random integers by using rand() from the standard library in c++.

```
for (int i = 0; i < SIZE; i++) {
    list1[i] = rand();
    list2[i] = rand();
}
```

Figure 2: Filling arrays with random integers.

To fill the tables with random integers with several duplicates, another similar for-loop was implemented, but the rand()-function was set to only use values between 50 and 1000. This will ensure many duplicates with the large quantity of elements in the tables.

```
//Initialize two different lists of random integers with several duplicates among the elements.
int* listWithDuplicates1 = new int[SIZE];
int* listWithDuplicates2 = new int[SIZE];

for (int i = 0; i < SIZE; i++) {
    listWithDuplicates1[i] = 50 + (rand() % 950);
    listWithDuplicates2[i] = 50 + (rand() % 950);
}
```

Figure 3: Creating arrays with several duplicates.

To create tables where the integers are already sorted, the std::sort()-method was used. The integers are decided by the SIZE constant.

```
//Initialize two different lists of already sorted integers.
int* sortedList1 = new int[SIZE];
int* sortedList2 = new int[SIZE];

for (int i = 0; i < SIZE; i++) {
    sortedList1[i] = rand();
    sortedList2[i] = rand();
}
```

Figure 4: Creating arrays where the elements are already sorted.

To make sure that the quicksort-methods were working as expected, two methods were implemented; one to check the sum of an array, and one to check if the order of the elements in the array was correct. The method to check the sum of an array was used to check the sum before and after running an algorithm to compare. If the values happened to be different, it would imply that something was wrong in the code.

```c
//Method to find the sum of the elements in an array.
int checkSums(int *list, int n) {
    int sum = 0;

    for (int i = 0; i < n; i++) {
        sum += list[i];
    }

    return sum;
}
```

Figure 5: Implementation of the checkSums()-method.

The method to check the order of the arrays was implemented to return a boolean value, 0 if the array is not in the correct order (ascending from the lowest value to the highest), and 1 if it is in the correct order.

```c
//Method to see if the elements in a list are in the correct order after being sorted.
bool checkOrder(int *list, int n) {
    for (int i = 0; i < n - 1; i++) {
        if (list[i + 1] < list [i]) {
            return false;
        }
    }

    return true;
}
```

Figure 6: Implementation of the checkOrder()-method.

Testing

Checksum and check order

To ensure that the sorting algorithms were reliable, tests to check the sum of the arrays, as well as the order of the elements in the arrays were run. Both the single and dual pivot quicksort method was tested, and confirmed that both algorithms had the same sum for the list before and after sorting, as well as having changed the lists elements to the correct order.

```
Test list for single pivot quicksort:
8, 6, 3, 7, 4, 2, 6, 2, 8, 2,

Size: 10
Sum before sorting the list using single pivot quicksort: 48
Sum after: 48
Was the list in the correct order before using single pivot quicksort? 0
Is the list in the correct order after using single pivot quicksort? 1

List after single pivot quicksort:
2, 2, 2, 3, 4, 6, 6, 7, 8, 8,
```

Figure 7: The result of the single pivot tests.

```
Test list for dual pivot quicksort:
1, 1, 2, 4, 4, 6, 7, 7, 9, 9,
Sum before sorting the list using dual pivot quicksort: 50
Sum after: 50
Was the list in the correct order before using dual pivot quicksort? 0
Is the list in the correct order after using dual pivot quicksort? 1

List after dual pivot quicksort:
1, 1, 2, 4, 4, 6, 7, 7, 9, 9,
```

Figure 8: The result of the dual pivot tests.

Testing a table of random integers

Number of elements in table:	100 000	1 000 000	10 000 000	100 000 000
Time used by the single pivot quicksort method (in µs):	12456	149074	1690600	18875182
Time used by the dual pivot quicksort method (in µs):	11889	144908	1617432	18401594

Table 1: The measured times for the different quicksort methods used on an array of random integers.

Part of the task was to measure the times used by the two different types of quicksort methods when sorting through an array of random integers. These are the resulting times using differently sized arrays for both algorithms.

Testing a table of random integers with several duplicates

Number of elements in table:	100 000	1 000 000	10 000 000	100 000 000
Time used by the single pivot quicksort method (in µs):	9327	105723	1084811	12027447
Time used by the dual pivot quicksort method (in µs):	9390	104739	1081507	11970966

Table 2: The measured times for the different quicksort methods used on an array of random integers with several duplicates.

These are the resulting times of running both quicksort methods for an array of a given amount of elements with several duplicates.

Testing a table that is already sorted

Number of elements in table:	100 000	1 000 000	10 000 000	100 000 000
Time used by the single pivot quicksort method (in µs):	3197	36205	450358	5046959
Time used by the dual pivot quicksort method (in µs):	3398	37657	451265	5048721

Table 3: The measured times for the different quicksort methods used on an array that has elements that are already sorted.

The table shows the results of the times measured for the algorithms when sorting an array that is already sorted.

Results

When the tables are filled with random integers, the dual pivot quicksort method is always slightly faster than the single pivot. The dual pivot is also faster when sorting through an array of random integers with several duplicates (except for the first measurement). Which method is faster can vary depending on several factors, such as implementations and programming language, but dual pivot quicksort should generally be slightly faster (Elhaiani, 2022). While dual pivot quicksort offers the possibility of a faster run time, the worst possible run time is still the same as when using a single pivot quicksort, which is $O(n^2)$ (Elhaiani, 2022).

When the quicksort methods are applied to lists that contain integers that already have been sorted, the runtime is considerably faster than the list of random integers, and the list with several duplicates. This is to be expected, because the algorithm does not have to “swap” the elements after comparing them, but can rather move on to the next comparison.

Sources

Elhaiani, S. (2022, June 19th). *Dual pivot Quicksort*. GeeksForGeeks. https://www.geeksforgeeks.org/dual-pivot-quicksort/

Hafting, H. & Ljosand, M. (2014) *Algoritmer og Datastrukturer med eksempler i C og Java*. Kopinor Pensum AS, Oslo.