HO CHI MINH UNIVERSITY OF TECHNOLOGY
Faculty of Computer Science and Engineering

Assignment

Computer Architecture

Instructor: Pham Quoc Cuong
Student: Dao Hoanh Dat - 1752157

HO CHI MINH, 2019

Contents

1 Introduction of Quick Sort **2**

2 Introduction of Binary Search **2**

3 Implementation **2**
3.1 Question 1 2
3.1.1 Question 1.1 2
3.2 Question 2 3
3.2.1 Question 2.1 3
3.2.2 Question 2.2 3

1 Introduction of Quick Sort

QuickSort is a Divide and Conquer algorithm. It picks an element as pivot and partitions the given array around the picked pivot. There are many different versions of quickSort that pick pivot in different ways.

- Always pick first element as pivot.
- Always pick last element as pivot.
- Pick a random element as pivot.
- Pick median as pivot.

The key process in quickSort is partition(). Target of partitions is, given an array and an element x of array as pivot, put x at its correct position in sorted array and put all smaller elements (smaller than x) before x, and put all greater elements (greater than x) after x. All this should be done in linear time.

2 Introduction of Binary Search

Binary Search: Search a sorted array by repeatedly dividing the search interval in half. Begin with an interval covering the whole array. If the value of the search key is less than the item in the middle of the interval, narrow the interval to the lower half. Otherwise narrow it to the upper half. Repeatedly check until the value is found or the interval is empty. The idea of binary search is to use the information that the array is sorted and reduce the time complexity to O(Log n).

3 Implementation

3.1 Question 1

3.1.1 Question 1.1

In the fisrt question, I use 10 label in order to implement the quick sort and also display the result, which is: main, quickSort, end-condition, partition, loop, condition-partition, increase-i, swap, end-loop, printArray

- Main: The code will be started from here.
- quickSort: this is the main label of the quick sort method, it will first store the address, low index, high index of the array. It also navigates the code to other labels.
- end-condition: this label is the end of the quick sort function. It will load all the value after sorting which is prepared for displaying.
- partition: this label is the core of the quick sort function. In this assignment, I takes last element (high index) as pivot, compares each of elements in the array with the pivot, and places them left (greater than pivot), otherwise place them right (smaller than pivot).
- loop: this is a small label in the partition. It will check the number of loops with the index of last element to continue loop or end loop.

- condition-partition: this label is using to compare the value of each element with the pivot. If the element greater than the pivot, the index i which is initialized with the value (low - 1) will be increased by 1, and the value of array[i] and the value of the compared element will be swapped, otherwise, the code just increases the number of loops.
- increase-i: increasing the value of i by 1
- swap: change the position of 2 elements
- end-loop: this label is going to tell you that the loop end and the function will swap arr[i+1] with pivot. The complier will turn back to the quickSort (main label) to continue to check the left and the right of the pivot.
- printArray: display the result on the screen after sorting by quick sort method.

3.2 Question 2

3.2.1 Question 2.1

The idea of this question is also the same with the question 1.1 but sorting in ascending order. I also use the same method (quick sort) to solve this problem.

3.2.2 Question 2.2

In the next question, I use 9 label in order to implement the binary search and also display the result, which is: input, binarySearch, loop-search, half-left, half-right, finish, not-exist, NA, output

- input: this label will take the input from the user
- binarySearch: this is the main label of the binary search method, it will first initialize the low index, high index of the array. It also navigates the code to other labels.
- loop-search: this label will take the middle element of the array and compare it to the input from user. If the element is equivalent to the input, the compiler will return the index of that input, otherwise, the compiler continue to check the half left with the same method if the input smaller than the middle element, or continue with the right one if the input greater than the middle.
- half-left: keep the low index, change the high index by the middle index - 1
- half-right: keep the high index, change the low index by the middle index + 1
- finish: return the result
- not-exist: return -1 doesn't have the input in the array
- NA, output: display the result on the screen depend on what value they receive.