Algorithms and Data Structures - Assignment 5

Hash tables, trees, sorting

Task A) Theory questions

1) Which of the following strategies is not a collision strategy for hashing?

a. Double hashing
b. Separate chaining
c. Binary search
d. Quadratic probing

2) Which of these hash functions would be the best choice for a hash table with 12 values?

a. $i \mod 17$
b. $(3 * i + 1) \mod 12$
c. $(i * 2) \mod 5$
d. $(i + 1) \mod 12$

3) Which runtime cost has, in the worst possible case, finding an element in a hash table (with separate chaining) with n stored keys?

a. $O(\log n)$
b. $O(n)$
c. $O(\log 1/n)$
d. $O(n \log n)$

Assuming the linked-list uses a push_back-styled saving method.

4) What average runtime cost has the insertion into an AVL - Tree?

a. $O(\log n)$
b. $O(n)$
c. $O(n^2)$
d. $O(n \log n)$

5) What happens when a binary search tree is traversed in inorder?

a. The nodes are visited in descending order according to their keys.
b. The nodes are visited in ascending order according to their keys.
c. Just the internal nodes are visited.
d. All nodes are traversed.

6) In the quicksort algorithm, which are bad choices for the pivot element?

a. The middle element.
b. The element of minimal value.
c. A random element.
d. The element of maximal value.

Task B) AVL tree

The aim of this task is to get an overview of the ADT AVL tree. The corresponding files can be found in the folder *Task B*. Several nodes are inserted in the file *main.cpp*, which should result in the following tree:

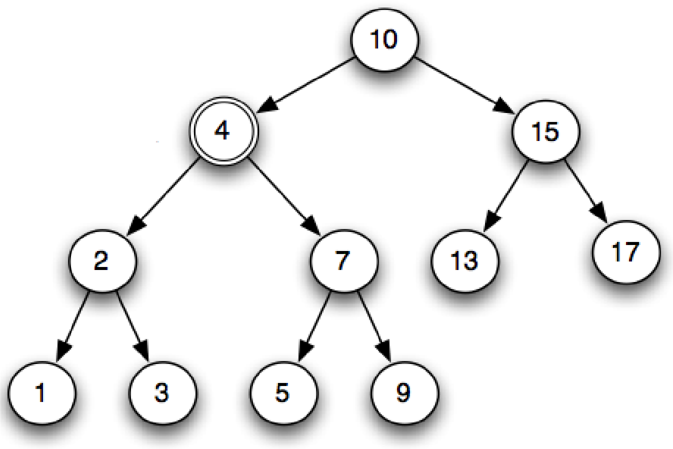

Figure 1: Example AVL tree

Hint: more info is available at http://en.wikipedia.org/wiki/AVL_tree.

1) Insert a node

In the file *avl_tree_node.cpp* you have the skeleton of a method named **void insert(int value)** which should create a new tree node and insert it into the existing tree. Remember to balance the tree afterwards (you can call the function **void restructure()**, which balances the tree under a node and is already implemented for you).

2) get_height()

In the same file we provide you the function skeleton **int get_height()** which should return the height under the actual node. Complete the method so that it returns the correct height (4 in the example tree).

3) Check whether a tree is balanced

We provide you also the function skeleton **bool is_balanced()** which should return if the actual tree is balanced.

Task C) Hash tables

In this exercise you will program your own hash table implementation. If you need theoretical help, go through the slide set *08_HashTables.pdf* in OLAT. The corresponding files for this task can be found in the folder *Task C*.

Our hash table is meant to store items having a key and a name (string). In main.cpp you see that there are two modes of usage: 'l' for linear probing and 'q' for quadratic probing. That means our hash table can put and get items in these two different ways. Complete the necessary methods of the file *HashTable.cpp* (the hash function, the constructor and a content-printing function are already implemented for you).

1) Linear probing

Your task is to program the method **void put(Item* item)** which inserts an item into the hash table. If a collision occurs on insert, you have to solve this situation using linear probing.

Additionally, you have to program the method **Item* get(int key_)** which returns the element identified by the given key. Do this also with linear probing.

2) Quadratic probing

Your task is to program the method **void putQuadratic(Item* item)** which, as before, inserts an item into the hash table. If there is a collision at position i, you should solve it using quadratic probing (i.e. try the indices $i + j^2$, for $j = 1, ..., N$).

Additionally, you have to program the method **Item* getQuadratic(int key_)** which returns an element identified by the given key. Do this also with quadratic probing.

Task D) More sorting

In this exercise, we practice more sorting algorithms. The corresponding files for this task are in the *Task D* folder.

1) Selection Sort

Implement a selection sort algorithm. Complete the function **void selection_sort (std::vector<int>& numbers_)**, in file *selection_sort.cpp*.

2) Mergesort

Figure 2 gives you an overview on how the merge sort algorithm works.

You have to complete the methods in *merge_sort.cpp*. The algorithm consists of two parts: splitting up the vector (do this in **void merge_sort(std::vector<int>& numbers)**) and then, when these two parts are sorted, merge them back together (do this in **void merge(std::vector<int>& numbers1, std::vector<int>& numbers2, std::vector<int>& numbers)**). Remember that the first part of recursive method is always the exit condition. So try to figure out the exit condition first before you complete the rest of the method.

3) Bucket Sort

Implement a bucket sort by filling the skeleton **void bucket_sort(std::vector<int>& numbers)**, in the file *bucket_sort.cpp*. The input numbers in *main.cpp* are all in the range $[0, 999]$.

4) Radix Sort

Implement a radix sort: use buckets to sort the numbers in the list. Complete the function **void radix_sort(std::vector<int>& input_list)** (file *RadixSort.cpp*).

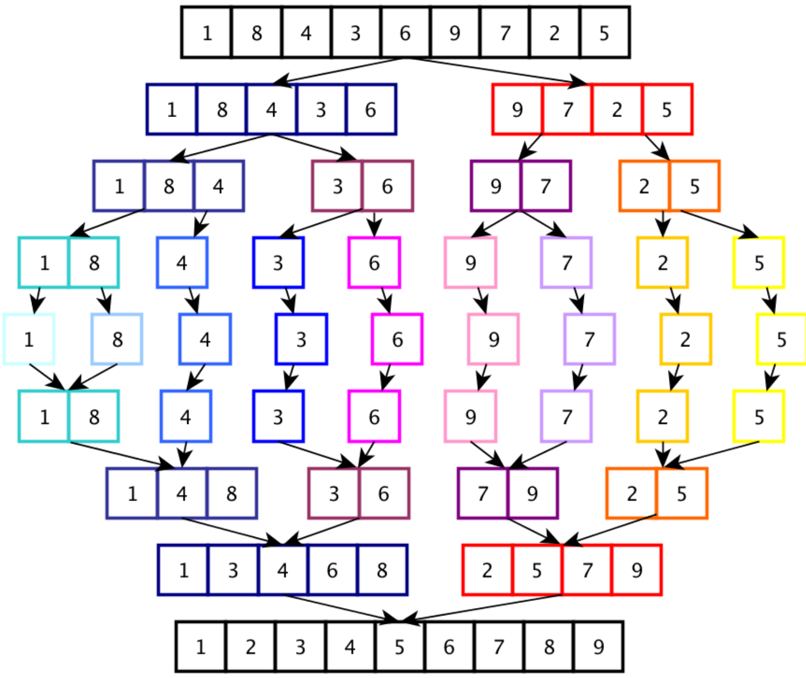

Figure 2: Mergesort workflow