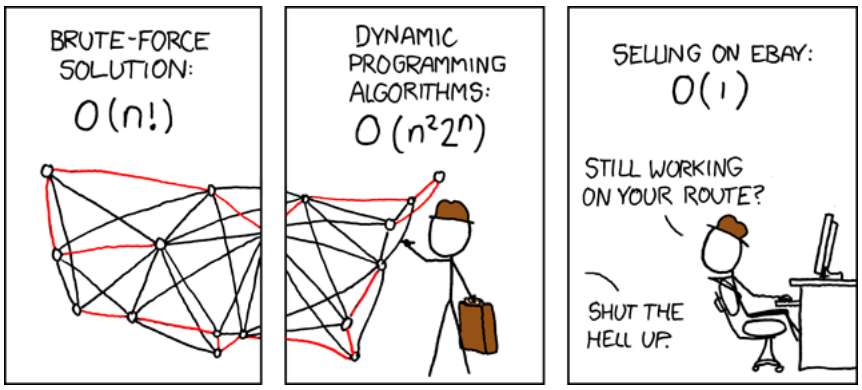
Assignment 9: Data structures

ETH Zurich

Hand-out: 18. November 2011
Due: 29. November 2011



Travelling Salesman Problem © Randall Munroe (http://xkcd.com/399/)

Goals

- Figure out when to use which data structure.
- Practice using data structures.
- Practice implementing your own data structures.

1 Choosing data structures

In the lecture you have learnt that different data structures can represent different kinds of information and support efficient implementation of different operations.

To do

For each of the use cases below pick a data structure (from the ones you have seen in the lecture) that is best suited for that use case, and explain your choice. If you think there is more than one suitable data structure, briefly discuss the trade-offs.

Example. You want to store the names of weekdays and access them by the number of a day within the week.

Answer. An array, because the number of weekdays is fixed, and access by index has to be efficient.

1. You want to store the stations of a public transportation line. New stations can be added to both ends of the line, but not between existing stations. You should be able to traverse the line in both directions.

2. You want to store a phone book, which supports looking up a phone number by name, as well as adding and removing entries.

3. You are looking for a way out of a maze (like in the previous assignment), but you are not allowed to use recursion. You have to store the path you are currently exploring, and be able to go back one step whenever you find yourself in a dead-end and explore a new possibility from there.

4. You want to store a sorted list of strings and support the operation of merging two sorted lists into one, in place (without creating a copy of the lists).

5. You are writing software for a call center. When a client calls, his call should be stored until there is a free operator to pick it up. Calls should be processed in the same order they are received.

To hand in

Hand in your answers for questions 1–5.

2 Short trips: take two

In the previous assignment you had to implement a feature that highlights all the stations that are reachable from a given station in two minutes or less. If your implementation is similar to the master solution, then it doesn't keep track of the stations it already visited and keeps visiting the same stations again and again. As a result, highlighting the short-trip range for stations with a lot of connections (for example, Bürkliplatz) takes a long time.

Note that it is incorrect to simply stop the search when the next station is already highlighted, because the route to that station you found the first time around could be longer than the one you are building this time, and thus you can miss some reachable stations.

In this task you have to find a way to make *highlight_short_distance* more efficient by keeping track of the state of traversal using an appropriate data structure.

To do

1. You can start either from your own solution to "Short trips" from the previous assignment or from the master solution: `http://se.inf.ethz.ch/courses/2011b_fall/eprog/assignments/09/traffic.zip`.

2. What kind of information do you have to keep track of, so that the traversal does not miss reachable stations, but also does not visit the same station unnecessary many times? Choose an appropriate data structure class from the EiffelBase2 library (see Table 1) to store this kind of information, and introduce an attribute of the corresponding type into *SHORT_TRIPS*. Modify your implementation of *highlight_reachable* to make use of the introduced data structure.

3. Compare the performance of *highlight_short_distance* before and after the modification (the difference is especially clear if you change the initial time interval from two to three minutes).

Data structure	Concrete implementations	Description
V_ARRAY	*V_ARRAY*	Arrays: elements are indexed with integers from a continuous interval
V_LIST	*V_ARRAYED_LIST*, *V_LINKED_LISTS*	Lists: elements can be inserted and removed at any position
V_TABLE	*V_HASH_TABLE*, *V_SORTED_TABLE*	Tables: values are indexed by keys; key-value pairs can be updated, added and removed
V_SET	*V_HASH_SET*, *V_SORTED_SET*	Sets: elements are unique and lookup is efficient
V_STACK	*V_LINKED_STACK*	Stacks: last in first out policy
V_QUEUE	*V_LINKED_QUEUE*	Queues: first in first out policy

Table 1: Data structure classes in EiffelBase2

To hand in

Hand in the code of *SHORT_TRIPS*.

3 English to Swiss-German Dictionary

The ETH administration is unhappy that many professors and teaching assistants do not speak Swiss-German. They asked you to design an English to Swiss-German dictionary application to help the teaching staff learn the language.

You decided to store the dictionary in a data structure called *trie* (or *prefix tree*). A trie is a special kind of a tree that is used to store maps of *keys* to *values* where the keys are usually strings. In your case English words are the keys and Swiss-German words are the values.

In a trie *edges* are labeled with characters in such a way that every path from the root to a node corresponds to a prefix of some key. If such path constitutes a whole key the node it leads to stores the corresponding value; otherwise the node is empty. See figure 1 for an example.

To make lookup in the dictionary even more efficient you decided that the children of each node should be sorted according to the character labels on the edges. For example, in figure 1 the children of the root node are stored in order 'h', 'i', 't'.

To do

1. Download `http://se.inf.ethz.ch/courses/2011b_fall/eprog/assignments/09/dictionary.zip` unzip it and open `dictionary.ecf`.

2. Class *TRIE* contains a skeleton of the trie node data structure. A trie node stores a linked list of edges, where each edge is associated with a label, the child node it is connected to and the next edge in the list. For example, the root node of the trie in figure 1 will store a reference to its first edge (labeled with 'h'), which in turn stores the reference to the 'i' edge, and that one — to the 't' edge. Remember that the edges in the list have to be sorted according to their labels.

   The feature *item*(*k*: *STRING*): *STRING* that looks up a key in the trie is already implemented. You can use this implementation as an example of traversing the trie.

   Your task is to implement two features:

   - *insert* (*k*, *v*: *STRING*) should insert a key-value pair (k, v) into the trie; if the key k already existed the corresponding value should be replaced with v.

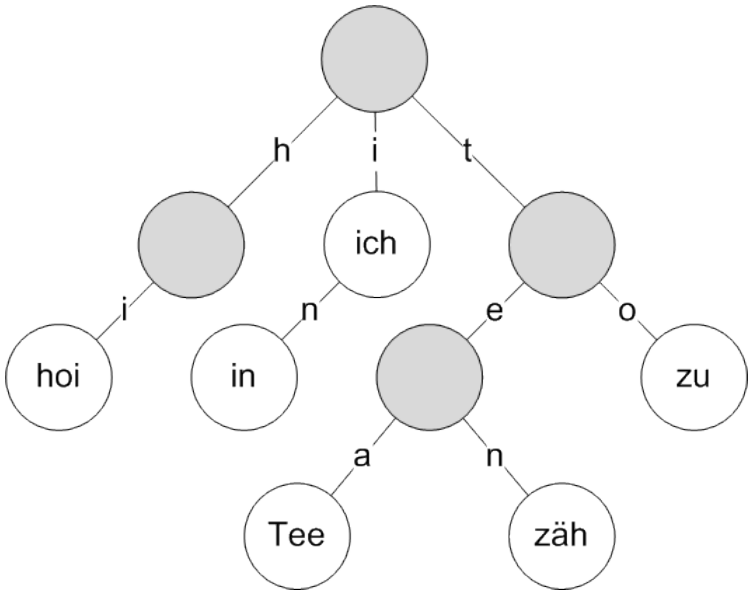


Figure 1: A trie storing translations of English words “hi”, “i”, “in”, “tea”, “ten”, “to”.

- *to_string*: *STRING* should return a printable representation of the trie with keys sorted *lexicographically* (see figure 2 for an example.) **Hint:** For the recursive implementation of *to_string* it is convenient to introduce an auxiliary function *to_string_with_prefix* (*key_prefix*: *STRING*): *STRING*, whose argument denotes the prefix that has to be prepended to every key.

3. Feature *execute* of class *APPLICATION* tests the implementation of *TRIE*. After you are finished with the three features from the previous task, the application should run without contract violations and produce the same output as in figure 2[1].

hi - hoi
i - ich
in - in
tea - Tee
ten - zäh
to - zu

Figure 2: String representation of the trie in figure 1.

To hand in

Hand in the code of *TRIE*.

[1]With the only difference that the symbol ‘ä’ is printed as ‘ae’.