Solution 9: Data structures

ETH Zurich

1 Choosing data structures

1. You can use a doubly-linked list. An arrayed list is also suitable if it is implemented as a circular buffer (that is, the list can start from any element in the array), in which case inserting in the beginning of the list is also efficient. A disadvantage of an arrayed list is that adding a station will sometimes take longer (when the array does not have any more free slots and has to be reallocated), an advantage is fast access by index, which is not mentioned in the scenario, but is always good to have.

A disadvantage of a doubly-linked list is high memory overhead: in addition to the reference to a station object each list element stores two other references (to the next and the previous element). Arrayed list also has a memory overhead (free array slots), however for common implementations this overhead will not be as high.

2. A hash table with names (strings) as keys and phone numbers as values, because hash table allows efficient access by key.

3. A stack, because the step that was added last is always the first to roll back.

4. A linked list, because it supports efficient insertion of the elements of the second list into the proper place inside the first list while merging. The insertion is done by re-linking existing cells and does not require creating a copy of either of the lists.

5. A queue, because the first call added to the data structure should be the first one to be processed.

2 Short trips: take two

Listing 1: Class SHORT_TRIPS

```plaintext
note
description: "Short trips."

class SHORT_TRIPS

inherit ZURICH_OBJECTS

feature -- Explore Zurich

highlight_short_distance (s: STATION)
    -- Highlight stations reachable from ‘s’ within 3 minutes.

require
```
station_exists: s /= Void
  do
    create times
    highlight_reachable (s, 3 * 60)
  end

feature {NONE} -- Implementation

times: V_HASH_TABLE [STATION, REAL_64]
  -- Table that maps a station to the maximum time that was left after visiting that station.
  -- Stations that were never visited, are not in the table.

highlight_reachable (s: STATION; t: REAL_64)
  -- Highlight stations reachable from 's' within 't’ seconds.
  require
    station_exists: s /= Void
  local
    line: LINE
    next: STATION
  do
    if t >= 0.0 and (not times.has_key (s) or else times [s] < t) then
      times [s] := t
      Zurich_map.station_view (s).highlight
      across
        s.lines as li
      loop
        line := li.item
        next := line.next_station (s, line.north_terminal)
        if next /= Void then
          highlight_reachable (next, t - s.position.distance (next.position) / line.speed)
        end
        next := line.next_station (s, line.south_terminal)
        if next /= Void then
          highlight_reachable (next, t - s.position.distance (next.position) / line.speed)
        end
      end
    end
  end

3 English to Swiss-German Dictionary

Listing 2: Class TRIE

note
description: "Trie data structure, where child nodes are stored in a linked list, sorted by label.”

class TRIE

2
feature -- Access

item (k: STRING): STRING
    -- Value associated with key 'k'.
    require
        key_not_void: k /= Void
    local
        e: EDGE
    do
        if k.is_empty then
            Result := value
        else
            from
                e := first
            until
                e = Void or else e.label >= k[1]
            loop
                e := e.next
            end
            if e /= Void and then e.label = k[1] then
                Result := e.node.item (k.substring (2, k.count))
            end
        end
    end

to_string: STRING
    -- String representation with keys sorted lexicographically.
    do
        Result := to_string_with_prefix ("")
    ensure
        result_not_void: Result /= Void
    end

feature -- Element change

insert (k, v: STRING)
    -- Insert key-value pair (k, v).
    require
        key_not_void: k /= Void
    local
        e, found: EDGE
    do
        if k.is_empty then
            value := v
        else
            if first = Void or else first.label > k[1] then
                create found.make (k[1], create {TRIE})
                found.set_next (first)
                first := found
            else
                from
$e := \text{first}$

until $e.\text{next} = \text{Void or else } e.\text{next.label} > k \ [1]$ loop
  $e := e.\text{next}$
end

if $e.\text{label} = k \ [1]$ then
  $\text{found} := e$
else
  $\text{create } \text{found.make} \ (k \ [1], \ \text{create} \ \{ \text{TRIE} \})$
  $\text{found.set_next} \ (e, \text{next})$
  $e.\text{set_next} \ (\text{found})$
end
end

$\text{found.node.insert} \ (k.\text{substring} \ (2, \ k.\text{count}), \ v)$

end

ensure $\text{correct_value: item} \ (k) \sim v$
end

feature { \text{TRIE} } -- Implementation

first: \text{EDGE} -- Edge connecting to the first child node.

value: \text{STRING} -- Value stored in the node.
  -- 'Void' if the node doesn't correspond to a whole key.

to_string_with_prefix (key_prefix: \text{STRING}): \text{STRING}
  -- String representation with keys sorted lexicographically
  -- and 'key_prefix' prepended to all keys.

require
  key_prefix_not_void: key_prefix /= \text{Void}

local
  e: \text{EDGE}

do
  if value /= \text{Void} then
    Result := key_prefix + " - " + value + "%N"
  else
    Result := ""
  end
from
  e := first
until $e = \text{Void}$ loop
  Result.append (e.node.to_string_with_prefix (key_prefix + e.label.out))
  $e := e.\text{next}$
end

ensure $\text{result_not_void}: \text{Result} /= \text{Void}$
end

end