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*

```
note
  description: "Short trips."

class
  SHORT_TRIPS

inherit
  ZURICH_OBJECTS

feature -- Explore Zurich

  highlight_short_distance (s: STATION)
      -- Highight 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

end
```

3 Bags

Listing 2: Class *LINKED_BAG*

```
class
  LINKED_BAG [G]

feature -- Access

  occurrences (v: G): INTEGER
```

```
      -- Number of occurrences of 'v'.
    local
      c: BAG_CELL [G]
    do
      from
        c := first
      until
        c = Void or else c.value = v
      loop
        c := c.next
      end
      if c /= Void then
        Result := c.count
      end
    ensure
      non_negative_result: Result >= 0
    end

feature -- Element change

  add (v: G; n: INTEGER)
      -- Add 'n' copies of 'v'.
    require
      n_positive: n > 0
    local
      c: BAG_CELL [G]
    do
      from
        c := first
      until
        c = Void or else c.value = v
      loop
        c := c.next
      end
      if c /= Void then
        c.set_count (c.count + n)
      else
        create c.make (v)
        c.set_count (n)
        c.set_next (first)
        first := c
      end
    ensure
      n_more: occurrences (v) = old occurrences (v) + n
    end

  remove (v: G; n: INTEGER)
      -- Remove as many copies of 'v' as possible, up to 'n'.
    require
      n_positive: n > 0
    local
      c, prev: BAG_CELL [G]
```

```
    do
      from
        c := first
      until
        c = Void or else c.value = v
      loop
        prev := c
        c := c.next
      end
      if c /= Void then
        if c.count > n then
          c.set_count (c.count - n)
        elseif c = first then
          first := first.next
        else
          prev.set_next (c.next)
        end
      end
    ensure
      n_less: occurrences (v) = (old occurrences (v) - n).max (0)
    end

  subtract (other: LINKED_BAG [G])
      -- Remove all elements of 'other'.
    require
      other_exists: other /= Void
    local
      c: BAG_CELL [G]
    do
      from
        c := other.first
      until
        c = Void
      loop
        remove (c.value, c.count)
        c := c.next
      end
    end

feature {LINKED_BAG} -- Implementation

  first: BAG_CELL [G]
      -- First cell.

end
```