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
      times_exists: times /= 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
```

4 MOOC: Genericity, Data Structures

Genericity

- Assume you have a class *SORTED_LIST* [*G* −> *COMPARABLE*] with, among others, routine

  ```
  sort
          -- Sort the elements of current.
      do
          ...
      end
  ```

  Assume to have, in another class, the variable definition *slp*: *SORTED_LIST* [*PERSON*]. The following statement is true: the definition would compile if class *PERSON* does inherit from *COMPARABLE*.

- Assume you have just created an object of type *LIST* [*PERSON*]. What happens if you try to add an object of type *CAR* to the list? Assume *CAR* does not inherit from *PERSON*. The answers that apply are: "It will not work. I will get a compile time error" and "It will not work. The only objects allowed into the list are those of type *PERSON* and its descendants".

- Assume you have just created an object of type *LIST* [*PERSON*]. What happens if you try to add an object of type *STUDENT* to the list? Assume *STUDENT* does inherit from *PERSON*. The answers that apply are: "It will work: I can add a STUDENT to a *LIST* [*PERSON*] if *STUDENT* inherits from *PERSON*" and "It will work: not only I can add a *STUDENT* to a *LIST* [*PERSON*] if *STUDENT* inherits from *PERSON*, but I can always add to the list an object of class *PERSON* and of any class inheriting from *PERSON*."

- Assume you have created an object of type *LIST* [*PERSON*], and filled it in with objects of types *STUDENT* and *TEACHER*. What happens if you try to retrieve an object from the list? Assume *STUDENT* and *TEACHER* do inherit from *PERSON*. The answers that apply are: "It will work: I can retrieve a *STUDENT* from a *LIST* [*PERSON*], and the same for a *TEACHER*, given that I know that *STUDENT* and *TEACHER* both inherit from *PERSON*" and "It will work: it's just that every time I retrieve an object, I don't know if it will be of type *STUDENT* or *TEACHER*".

- The true statements about classes and types are: "Any non-generic class is a type", "For a generic class to be a type, we need to provide an actual type for the generic parameter", and "Any type is a class.".

- Declaring a class as *ARRAY* [*ARRAY* [*STRING*]] is legal for the Eiffel compilers you are using in this course: True.

Data Structures

- Which basic data structure stores items in contiguous memory locations, each identified by an integer index? An array, an arrayed list (a list implemented using an array)

- Which basic data structure does not provide access to all stored items, but only to the one which was added first? An arrayed queue (a queue implemented using an array), a linked queue (a queue implemented using a linked list).

- The following statements about hash tables are true: Hash tables are a particular kind of associative arrays; A hash table allows to access items via integer keys; Which hash function we use can influence the efficiency of all operations in a hash table.
- Assume you need a data structure in which you can insert elements in the middle efficiently. Which data structure and which implementation would you choose? A linked list.
- Assume you have to write a program that has to find the exit of a labyrinth. You have to store the path you are currently exploring, be able to go back one step whenever you find yourself in a dead-end, and explore a new possibility from there. Assuming you don't want to use recursion, which data structure would you choose? A stack.
- Assume you have to write a program supporting the operation of merging two sorted lists into one "in place" (without creating a copy of the lists). Which kind of data structure would be more efficient to use? A linked list.