- Country — towns count. Each town has some post office — packages stored & transferred

- post office → some limits (storage, min_weight, max_weight)

- packages stored in some order in the office queue! — processed using order when sending & Receiving.

- Sometimes — transaction b/w post offices — (different towns)
  - 1st one sends all packages to next one.
  - 2nd one accepts — packages — satisfy weight limits of it, rejects all others!
  - Rejected packages — returned! — stored in same order (before being sent).
  - Accepted packages (2nd one) → moves to tail of the queue (same order)

- Process — several queries!

structures: package, post_office and town

1)

Print_all_packages: — given town t — print all packages in this town.

queue: First in first out

```
Town_name:
    0:
        id_0
        id_1
        ...
    1:
        id_2
        id_3
        ...
    ...
```

- 0, 1, ... → post office numbers
- id0, id1 → packages from 0th post office. (queue) — order!
- id2, id3 → from 1st one.

\t b/w (before) id_0

2) send_all_acceptable_packages — source, target (town), postoffice indices → source_office_index, target_office_index

manage: transaction described: b/w source & target post office in different towns.

3) town_with_most_packages: given towns, (one with — most packages in all post offices!)

Several: Return first one from the collection towns!

4) find_town : String name → find town with <u>name</u> name i/p

2 ——→ No. of towns

A ——→ Town name

2 ——→ office count

2 1 3 ——→ Package count, min_weight, max_weight → (describe package)!

a 2 ——→ id weight } 2 packages

b 3 ——→ id weight

1 2 4 ——→ package count, min_weight, max_weigh (office 1)

c 2 ——→ package ~~count~~, weight.

B ——→ Town name

1 ——→ 1 post office.

4 1 4 ——→ 4 pack, min, max

d 1
e 2
f 3
h 4

} 4 packs & 5 weights

5 ——— Numbers of queries.

3 ——— 1- Town-name, name of the town need to be printed. Ref. below.

2 B 0 A 1 ———

3 ———

1 A ———

1 B ———

1 → townName → there is a town → print all packages in that town.

2 → Source city, Source office index, Dest city, target office index (i/p)

↳ Transaction – process!

3 → Town with most packages – Found!

<u>O/P</u>

Type: 1: All packages – Format

Query: 3 – most packages!

① print_all_packages (town t) → void → loop all offices → take count → print

Format:

```
\t    A:
\t    \t   0:
\t    \t   \t   a
No!              b
            1:
                 c
                 e
                 f
                 h
```

```
void print_all_packages (town t)
{
    int i, j;
    printf ("%s:\n", t.name);
    for (i=0; i < t.offices_count; i++)
    {
      printf ("\t%d:\n", i);
      for (j=0; j < t.offices[i].packages_count; j++)
      {
          printf ("\t\t%s\n", t.offices[i].packages[j].id);
      }
} }
```

```
struct town
{
  char *name;
  post_office *offices;
  int offices_count;
};
```

```
struct post_office
{
  int min_weight;
  int max_weight;
  package * packages;
  int package_count;
}
```

```
struct package
{
  char * id;
  int weight;
}
```

② find town – return town * (struct type town) – pointer pointing to struct type town.

"→" dereference then access member.

```
town * find_town (town *towns, int towns_count, char *name)
{
  int i;
  for (i = 0; i < towns_count; i++)
  {
    if (strcmp (name, towns [i].name) = = 0)
                 break;
  }
  return (& towns [i]);
```

∴ towns [i] → struct town

& towns [i] → struct: town *

③ towns_with_most_package (towns * towns, int towns_count)

① loop through all towns

② count – loop all packages in a city!

```
int getTotalPackages (town t)
{
    int i, sum = 0;
    for (i=0; i < t.offices_count; i++)
    {
        sum + = t.offices [i]. packages_count ;
        }
        return sum;
    }
```

```
town   town_with_most_packages (town *towns, int towns_count) {
{
  int i, max=0, pack, big;
  for (i=0; i < towns_count; i++)
  {
    pack = getTotalPackage (towns [i]);
    if (pack > max)
     {
       max = pack;
       big = i;          ———→  to return towns[i]
     }
  }
  return (towns [big]);
```

town * towns

pointer to town

```
void send_all_packages_acceptable (town * source, int source_office_index,
                                   town *target, int target_office_index)
```

Idea:

Source

a	b	c	d	e	f

target

g	h	i			

if eligible – append to target

else – collect separately list (order preserved)

Say: b, d, f → Not eligible

source

a	b	c	d	e	f

target

g	h	i	a	c	e

↙ Not removed! Just copying (so data - will be there)!

list - temp

b	d	f	

Note: Copy list (address) to source → after freeing dynamically allocated space of source!

* To access package weigh - through a package - for short writing getaddress

```
void send_all_acceptable_packages (town * source, int source_office_index,
        town. *target, int target_office_index) {

        int i, K=1, wgt;
        post_office *src = & (source → offices [source_office_index]);
        post_office *tar = & (target → offices [target_office_index]);
        package *new_packages = (package *) malloc (sizeof (package));
        for(i=0; i < src→ packages_count; i++){
            wgt = src → packages [i]. weight;
            if (wgt >= tar → min_weight && wgt <= tar→max_
                                                    weight) {

                (tar→ packages_count) ++
                tar → packages = realloc (tar→ packages, sizeof
                                          (package) *
                                          tar→packages_count)
                                                              ;

                tar → packages [tar → packages_count - 1] =
                                          src → packages [i];
        }

        else
        {
          new_package [K-1] = src → packages [i];
          K++;
          new_packages = realloc (new_packages, ~~*~~sizeof (package)
                                                  *K);
    }
    free (src→packages);
    src→ packages = new_packages;
    src → packages_count = K-1;
}
```

type

(Package *) ←