```
#include <assert.h>
#include <ctype.h>
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <upo/hashtable.h>
#include <upo/bst.h>

//HASH
/*
Implementare un algoritmo che, data una tabella hash H, con gestione delle collisioni
basata su indirizzamento aperto (open addressing)
e scansioe lineare (linear probing) con uso di tombstone, e una lista di chiavi, calcoli
il massimo
numero di collisioni delle chiavi k contente in l in H.
 *Se una chiave k in k non è contenuto in H, non dev'essere considerata nel calcolo
della media del numero di collisioni
 *Se H è vuota, o se l è vuota, o se nessuna chiave di l è contenuto in H, l'algoritmo
deve restituire il valore -1.
 */
int upo_ht_linprob_max_collisions(const upo_ht_linprob_t ht, const upo_ht_key_list_t
key_list)
{
  if(upo_ht_is_empty(ht) || key_list == NULL) return NULL;

  int max_collision = -1;

  upo_ht_key_list_node_t* current = key_list;

  while(current != NULL)
  {
    size_t hash = ht->key_hash(current->key, ht->capacity);
    int collision = 0;

    for(int i = 0; i < ht->capacity; i++)
    {
      size_t tmp_hash = (hash + i) % ht->capacity;
      if(ht->key_cmp(current->key, ht->slots[tmp_hash].key) == 0 && ht->slots[tmp_hash].keys
!= NULL)
      {
        if(collision > max_collision)
          max_collision = collision + 1;
        break;
      }
      else if(ht->slots[tmp_hash].tombstone && ht->slots[tmp_hash].key != NULL)
        collision++;\
      else
```

```
      break;
    }
    current = current->next;
  }

  return max_collision;
}

double upo_ht_linprob_avg_collisions(const upo_ht_linprob_t ht, const upo_ht_key_list_t
key_list)
{
  if(upo_bst_linprob_is_empty(ht) || key_list == NULL) return -1;

  int total_collisions = 0;
  int num_keys = 0;

  upo_ht_key_list_node_t* current = key_list;

  while(current != NULL)
  {
    size_t hash = ht->key_hash(current->key, ht->capacity);

    int collisions = 0;
    int found = 0;

    for(int i = 0; i < ht->capacity && !found; i++)
    {
      size_t tmp_hash = (hash + i) % ht->capacity;
      if(ht->key_cmp(ht->slots[tmp_hash].keys, current->key) && ht->slots[tmp_hash].keys !=
NULL)
      {
        total_collisions += collisions;
        num_keys++;
        found = 1;
      }
      else if(ht->slots[tmp_hash].tombstone && ht->slots[tmp_hash].keys != NULL)
        collisions++;
    }
    current = current->next;
  }

  if(num_keys == 0) return -1;

  return (double) total_collisions / num_keys;
}
/*
Restituisce la lista delle chiavi salvate nella hash table
*/
```

```
upo_ht_key_list_t upo_ht_sepchain_keys(const upo_ht_sepchain_t ht)
{
  if(upo_ht_is_empty(ht)) return NULL;

  upo_ht_key_list_t list = NULL;

  for(int i = 0; i < ht->capacity; i++)
  {
    if(ht->slots[i].head != NULL){
    for(upo_ht_sepchain_node_t* node = ht->slots[i].head; node != NULL; node = node->next)
    {
      upo_ht_key_list_node_t* list_node = malloc(sizeof(upo_ht_key_list_node_t));
      list_node = node->key;
      list_node->next = list;
      list = list_node;
    }
    }
  }
  return list;
}

upo_ht_key_list_t upo_ht_linprob_keys(const upo_ht_linprob_t ht)
{
  if(upo_ht_linprob_is_empty(ht)) return NULL;

  upo_ht_key_list_t list = NULL;

  for(int i = 0; i < ht->capacity; i++)
  {
    if(ht->slots[i].keys != NULL)
    {
      upo_ht_key_list_node_t* node = malloc(sizeof(upo_ht_key_list_node_t));
      node->key = ht.>slots[i].keys;
      node->next = list;
      list = node;
    }
  }
  return list;
}
```

//BST

/*

Implementare un algoritmo che ritorni il predecessore di una chiave in un albero binario di ricerca (BST).

Dati in input un BST e una chiave k, il predecessore di k nel BST è la più grande chiave k′ contenuta nel BST tale che k′ <k.

Se il predecessore di k non esiste o se il BST è vuoto, l'algoritmo deve ritornare il valore NULL
***/**

```
const void *upo_bst_predecessor(const upo_bst_t bst, const void *key)
{
   if (bst == NULL || key == NULL) return NULL;

   return upo_bst_predecessor_impl(bst->root, key, bst->key_cmp);

}

void *upo_bst_predecessor_impl(upo_bst_node_t *node, const void *key,
upo_bst_comparator_t key_cmp)
{
   if (node == NULL) return NULL;

   if (key_cmp(node->key, key) >= 0)
      return upo_bst_predecessor_impl(node->left, key, key_cmp);

   void *pred = upo_bst_predecessor_impl(node->right, key, key_cmp);
   if (pred == NULL)
      return node->key;
   else
      return pred;
}
```

/*
Implementare un algoritmo che ritorni il rango di una data chiave k in un albero binario di ricerca (BST).
Dato un BST e una chiave k (non necessariamente contenuta nel BST), il rango di k nel BST equivale al numero di chiavi
presenti nel BST che sono minori di k.
***/**

```
size_t upo_bst_rank(const upo_bst_t bst, const void *key)
{
  if(bst == NULL || key == NULL) return 0;

  size_t result = 0;
  upo_bst_rank_impl(bst->root, key, &result, bst->key_cmp);
  return result;
}

void upo_bst_rank_imp(upo_bst_node_t* node, const void *key, size_t* result,
upo_bst_comparator_t key_cmp)
{
  if(node == NULL) return 0;
```

```
  upo_bst_rank_impl(node->left, key, result, key_cmp);

  if (key_cmp(node->key, key) < 0)
    (*result)++;

  upo_bst_rank_impl(node->right, key, result, key_cmp);
}
```

**/*
Dati in input un albero binario di ricerca (BST), una chiave k (non necessariamente contenuta nel BST) e un valore
intero d>=0, implementare un algoritmo che ritorni il numero di foglie del sotto-alber radicato in k e che si trovano a
una profondità d del BST; si noti che la radice dell'albero ha profondità 0 e che il conteggio finale può includere il nodo
con la chiave k se questo è una foglia e si trova a profondità d. Se la chiave k non è contenuta nel BST, l'algoritmo deve
ritornare il valore 0.
*/**

```
size_t upo_bst_subtree_count_leaves_depth(const upo_bst_t bst, const void *key, size_t d)
{
  if(bst == NULL || key == NULL || d < 0) return 0;

  upo_bst_node_t* node = upo_bst_find_node(bst->root, key, bst->key_cmp);
  if(node == NULL) return 0;

  return upo_bst_subtree_count_leaves_depth_impl(node, d);
}

size_t upo_bst_subtree_count_leaves_depth_impl(upo_bst_node_t* node, size_t depth)
{
  if(node == NULL) return 0;
  int count = 0;
  if(depth == 0) count = 1;

  return count + upo_bst_subtree_count_leaves_depth_impl(node->left, depth - 1) +
upo_bst_subtree_count_leaves_depth_impl(node->right, depth - 1);
}

upo_bst_node_t* upo_bst_find_node(upo_bst_node_t* node, const void* key,
upo_bst_comparator_t key_cmp)
{
  if(node == NULL) return NULL;
  size_t cmp = key_cmp(node->key, key);

  if(cmp == 0) return node;
  else if(cmp > 0)
```

```
    return upo_bst_find_node(node->left, key, key_cmp);
  else
    return upo_bst_find_node(node->right, key, key_cmp);
}

/*
Implementare un algoritmo che ritorni la lista delle chiavi in un albero binario di ricerca (BST) che sono minori o
uguali a una chiave k. Dato un BST e una chiave k (non necessariamente contenuta nel BST), il numero di chiavi
nel BST minori o uguali a k si ottiene contando tutte le chiavi contenute nel BST che sono minori della o uguali alla chiave
k.
*/

upo_bst_key_list_t upo_bst_keys_le(const upo_bst_t bst, const void *key)
{
  if(bst == NULL || key == NULL) return NULL;
  upo_bst_key_list_t list = NULL;

  upo_bst_keys_le_impl(bst->root, key, bst->key_cmp, &list);
  return list;
}

void upo_bst_keys_le_impl(upo_bst_node_t* node, const void *key, upo_bst_comparator_t key_cmp, upo_bst_key_list_t *list)
{
  if(node != NULL)
  {
    upo_bst_keys_le_impl(node->left, key, key_cmp, list);

    if(key_cmp(node->key, key) >= 0)
    {
      upo_bst_key_list_node_t* list_node = malloc(sizeof(upo_bst_key_list_node_t));
      list_node->key = node->key;
      list_node->next = *list;
      *list = list_node;
    }

    upo_bst_keys_le_impl(node->right, key, key_cmp, list);
  }
}

/*
Dato un albero BST ed un intervallo di chiavi ad estremi inclusi, restituire la somma dei valori la cui chiave è
compresa nell'intervallo
*/
```

```
void *upo_bst_sum_in_range(const upo_bst_t tree, const void *low, const void *high)
{
  if(tree == NULL || low == NULL || high == NULL) return 0;

  return upo_bst_sum_in_range_impl(tree->root, low, high, tree->key_cmp);
}

void* upo_bst_sum_in_range_impl(upo_bst_node_t* node, const void* low, const void *high,
upo_bst_comparator_t key_cmp)
{
  if(node == NULL) return 0;
  int sum = 0;

  size_t cmp_low = key_cmp(node->key, low);
  size_t cmp_high = key_cmp(node->key, high);

  if(cmp_low >= 0 && cmp_high <= 0)
    sum += *(int*)node->key;

  if(cmp_low > 0)
     sum += (int)upo_bst_sum_in_range_impl(node->left, low, high, key_cmp);
  if(cmp_high < 0)
     sum += (int)upo_bst_sum_in_range_impl(node->right, low, high, key_cmp);

  return sum;
}
```

**/*
Implementare un algoritmo che ritorni il numero di nodi di un sotto-albero in un albero binario di ricerca (BST) che si
trovano a una profondità pari.
Dato un BST e una chiave k, il numero di nodi del sotto-albero (del BST) radicato in k e situati a una profondità pari si ottiene
contando tutte i nodi che si trovano a profondità pari e che sono contenuti nel sotto-albero il cui nodo radice ha come chiave il
valore k. Si noti che la radice dell'intero BST ha profondità 0, che è un numero pari. Il conteggio dei nodi include anche la radice
del sotto-albero se si trova a una profondità pari. Se la chiave k non è presente nel BST o se il BST è vuoto o se il sotto-albero non
contiene nodi a profondità pari, l'algoritmo deve ritornare il valore 0.
*/**

```
size_t upo_bst_subtree_count_even(const upo_bst_t bst, const void *key)
{
  if(bst == NULL || key == NULL) return 0;

  size_t depth = 0;
  upo_bst_node_t* node = upo_bst_find_node_impl(bst->root, key, bst->key_cmp, depth);
```

```
  return upo_bst_subtree_count_even_impl(node, depth);
}

size_t upo_bst_subtree_count_even_impl(const upo_bst_node_t *node, size_t depth)
{
    if (node == NULL) return 0;

    size_t count = 0;
    if (depth % 2 == 0)
        count = 1;

    return count + upo_bst_subtree_count_even_impl(node->left, depth + 1) +
upo_bst_subtree_count_even_impl(node->right, depth + 1);
}


upo_bst_node_t* upo_bst_find_node(upo_bst_node_t* node, const void *key,
upo_bst_comparator_t key_cmp, size_t *depth)
{
  if(node == NULL) return NULL;

  size_t cmp = key_cmp(node->key, key);

  if(cmp == 0) return node;
  else if(cmp > 0)
  {
    (*depth)++;
    return upo_bst_find_node_impl(node->left, key, key_cmp, depth);
  }
  else
  {
    (*depth)++;
    return upo_bst_find_node_impl(node->right, key, key_cmp, depth);
  }
}
```

**/*
Implementare un algoritmo che, dato un albero binario di ricerca (BST), una chiave k (non necessariamente contenuta nel BST) e
un intero n, restituisca:
*l'n-esima chiave più piccola del sottoalbero la cui radice ha come chiave k, se esiste e se k è contenuta nel BST;
*NULL, se l'nesima chiave più piccola non esiste, se k non è contenuta nel BST, o se il BST è vuoto.
Si noti che l'nesima chiave più picola è la chiave che si troverebbe nell'nesima posizione se le chiavi fossero disposte in
ordine di grandezza
*/**

```
void *upo_bst_nmin(const upo_bst_t tree, const void *key, const int n)
{
  if(tree == NULL || key == NULL || n < 0) return NULL;

  upo_bst_node_t* node = upo_bst_find_node(tree->root, key, tree->key_cmp);

  int copyN = n;
  return upo_bst_nmin_impl(node, &copyN);
}

void* upo_bst_nmin_impl(upo_bst_node_t* node, int *n)
{
  if(node == NULL) return NULL;

  upo_bst_node_t* left = upo_bst_nmin_impl(node->left, n);
  if(left != NULL) return left;

  (*n)--;
  if(*n == 0) return node;

  return upo_bst_nmin_impl(node->right, n);
}

void* upo_bst_find_node(upo_bst_node_t* node, const void* key, upo_bst_comparator_t
key_cmp)
{
  if(node == NULL) return NULL;

  int cmp = key_cmp(node->key, key);

  if(cmp == 0)
    return node;
  else if(cmp > 0)
    return upo_bst_find_node(node->left, key, key_cmp);
  else
    return upo_bst_find_node(node->right, key, key_cmp);
}
```

**/*
Implementare un algoritmo che ritorni il valore e la profondità di una chiave in un albero binario di ricerca (BST.
Se la chiave non è contenuta nel BST, l'algoritmo deve ritornare null come valore della chiave e -1 come profondità.
Per profondità di una chiave s'intende la profondità del nodo del BST in cui la chiave è memorizzata.
Si ricordi che la radice di un BST si trova a profondità zero.
*/**

```
void *upo_bst_get_value_depth(const upo_bst_t bst, const void *key, long *depth)
```

```
{
  if(bst == NULL || key == NULL || depth < 0)
  {
  (*depth) = -1;
  return NULL;
  }

  (*depth) = 0;
  return upo_bst_get_valure_depth_impl(bst->root, key, depth, bst->key_cmp);
}

void *upo_bst_get_value_depth_impl(upo_bst_node_t* node, const void* key, long *depth,
upo_bst_comparator_t key_cmp)
{
  if(node == NULL)
  {
    *depth = -1;
    return NULL;
  }

  int cmp = key_cmp(node->key, key);

  if(cmp == 0)
    return node;
  else if(cmp > 0)
  {
    (*depth)++;
    return upo_bst_get_value_depth_impl(node->left, key, depth, key_cmp);
  }
  else
  {
    (*depth)++;
    return upo_bst_get_value_depth_impl(node->right, key, depth, key_cmp);
  }
}
```