Assignment 7 Design

Michael Coe

November 2021

bv.c

Goal: To implement a bitvector ADT

- `struct BitVector`

  This ADT has 2 variables:

  ```
  uint32_t length;
  uint8_t *vector;
  ```

- `BitVector *bv_create(uint32_t length)`
  This function creates a new bit vector and allocate `vector` to be the length of `(length / 8)+1`.
- `void bv_delete(BitVector **bv)`

  This function frees `(*bv)->vector`. if `*bv != NULL`, free `*bf`, sets `*bv` to `NULL`.
- `uint32_t bv_length(BitVector *bv)`

  This function returns `bv->length`.
- `bool bv_set_bit(BitVector *bv, uint32_t i)`

  This function returns false if `i` is less than `bv->length`, otherwise, create two variables: `uint32_t index = i / 8; uint8_t bit = i % 8`, then use bitwise and with `bv->vector[index]` and `1 << bit` then return true.
- `bool bv_clr_bit(BitVector *bv, uint32_t i)`

  This function returns false if `i` is less than `bv->length`, otherwise, create to variables `uint32_t index = i / 8;uint8_t bit = i % 8`, then use bitwise and with `c->vector[index]` and `0 << bit` then return true.

- `bool bv_get_bit(BitVector *bv, uint32_t i))`

  This function returns false if `i` is less than `bv->length`, otherwise, create to variables `uint32_t index = i / 8;uint8_t bit = i % 8`, then if `bv->vector[index] >> bit` and `1` then return true, otherwise return false.

- `void bv_print(BitVector *bv)`

  This function iterates from 0 to `bv->length` and calls `bv_get_bit()` and prints a 1 if it returns true and prints a 0 otherwise.

bf.c

Goal: This program creates a bloomfilter to check what is likely to be a member of set

- `struct BloomFilter`
  This Structure has 4 variables:

  ```
  uint64_t primary [2]; // Primary hash function salt.
  uint64_t secondary [2]; // Secondary hash function salt.
  uint64_t tertiary [2]; // Tertiary hash function salt.
  BitVector *filter;
  ```

- `BloomFilter *bf_create(uint32_t size)`
  This function creates a bloomfilter ADT by setting the hashes to the salts given in header files and `filter` to `*bv_create(uint32_t length)`.

- `void bf_delete(BloomFilter **bf)`
  If `*bf != NULL`, this function calls `bv_delete(&((*bf)->filter))`, frees `*bf`, then sets `*bf` to `NULL`.

- `uint32_t bf_size(BloomFilter *bf)`
  This function returns `bv_length(bf->filter)`.

- `void bf_insert(BloomFilter *bf, char *oldspeak)`
  This function hashes `oldspeak` using `hash( salt[], oldspeak)` 3 times with the `salt[]` variable being `bf->primary`, `bf->secondary`, and `bf->tertiary` for each respective hash. Then call `bv_set_bit()` for `bf->filter` with the return value of each hash.

- `bool bf_probe(BloomFilter *bf, char *oldspeak)`
  This function hashes `oldspeak` three times the same way as the previous function, but this time return false if any of the 3 inserts into the bit vector return false. Otherwise return true.

- `uint32_t bf_count(BloomFilter *bf)`
  This function creates an integer `count = 0` and creates a for loop that iterates from 0 to `bf_size(bf)`. Inside the loop it calls `bv_get_bit()` and increments count if the return value is true. After the loop ends return count.

- `void bf_print(BloomFilter *bf)`
  This function calls `bv_print`.

node.c

Goal: To create Node ADT.

- `Struct Node`

  This ADT has 4 variables:

  ```
  char *oldspeak;
  char *newspeak;
  Node *left;
  Node *right;
  ```

- `*node_create(char *oldspeak, char *newspeak)`

  This function creates a new Node pointer and copies the inputs to the char variables using `strdup()`, then return the new pointer.

- `void node_delete(Node **n)`

  If `*n != NULL`, this function frees the two char variables and `*n` and sets `*n = NULL`.

- `void node_print(Node *n)`

  This function prints `("oldspeak: %s, newspeak: %s", n->oldspeak, n-> newspeak)`. `NULL`.

bst.c

Goal: To build a binary search tree ADT.

- `Node *bst_create(void)` This function creates and returns a null node pointer
- `void bst_delete(Node **root)` This function recursively deletes every node in a binary tree. if root does not point to NULL, then call `delete_tree()` for both the left and right children, then call `node_delete(root)`.
- `uint32_t bst_height(Node *root)` This function recursively finds the height of a given node in a bst. If the root has both children, call `bst_height()` for both children and return 1 plus the largest return value. If the root only has 1 child, return 1 plus the return value `bst_height()` for that child. If the root has no children, then return 1.

- `uint32_t bst_size(Node *root)` This function returns the size of a tree bases at the root recursively. If the root has both children, call `bst_size()` for both children and return 1 plus the sum of both return value. If the root only has 1 child, return 1 plus the return value `bst_size()` for that child. If the root has no children, then return 1.
- `Node *bst_find(Node *root, char *oldspeak)` This function recursively searches for the node containing the correct oldspeak in the binary search tree. If the `root->oldspeak == oldspeak` (using `strcmp()`), then return the root. Call `st_find()` on each of the roots children, and if the return value of either of them are not `NULL` return the non-`NULL` value. Otherwise return `NULL`.
- `void bst_print(Node *root)` This function first recursively calls itself on the left node if it exits, then calls `node_print(root)`, and finally calls itself on the right node if it exists.

ht.c

Goal: To build a functioning hashtable

- `HashTable *ht`$_c reate(uint32\_tsize)$
- `void ht_delete(HashTable **ht)`
- `uint32_t ht_size(HashTable *ht)`
- `Node *ht_lookup(HashTable *ht, char *oldspeak)`
- `void ht_insert(HashTable *ht, char *oldspeak, char *newspeak)`
- `double ht_avg_bst_size(HashTable *ht)`
- `double ht_avg_bst_height(HashTable *ht)`
- `void ht_insert(HashTable *ht, char *oldspeak, char *newspeak)`
- `void ht_print(HashTable *ht)`

banhammer.c