Assignment 7 - Document

Jack Campbell

banhammer.c - This file should act at the main file and should include main().

ht.c - The file will contain the 'HashTable' structure. Hash tables are similar to a the dictionary structure found in languages like Python. Every value has a corresponding key that it is paired with. The functions needed to use a hash table are: ht_create(), ht_delete(), ht_size(), ht_lookup(), ht_insert(), count(), and ht_print().

ll.c - a This file will contain the 'LinkedList' data structure. A Linked List is set of node containing pointers to those in front and behind them; this allows for ease of access and ordering. The functions required are: ll_create(), ll_delete(), ll_lookup(), ll_insert(), and ll_print().

node.c - The file will contain the 'node' ADT. Huffman trees are made of nodes and contain a pointer to its left child, a pointer to its right child, a symbol, and the frequency of that symbols occurrence. The frequency is only required in the encoder because it tracks the occurrence of symbols from the input file.

bf.c - This file will contains the structure 'Bloom Filter' and everything related to using it. Function include: bf_create(), bf_delete, bf_size, bf_insert(), bf_probe(), bf_count(), and bf_print().

bv.c - This file will contains the structure 'BitVector' and everything related to using it. Function include: bv_create(), bv_delete, bv_length, bv_set_bit(), bv_clr_bit(), bv_get_bit(), and bf_print().

PURPOSE

The function of banhammer.c is to read in the .txt files badspeak.txt, newspeak.txt, as well as words from stdin. The words from stdin should then be compared to those found in the other two files and added to their corresponding linked list — either rightspeak or thoughtcrime. After reading in all of the strings from stdin, a message should be displayed depending on the size of each linked list. Statistics also may be printed, although not concurrently with the message.

structure banhammar

- #include all that is required
- #define arguement options and regex word
- initialize main function
  - initialize boolean values cooresponding arguement presence to False
  - initialize while loop - iterate through getopt() values -
    - switch statement for each argument
      - set boolean values to True if cooresponding arguement is found
      - if no or incorrect arguments are found, print help message and end program
  - open badspeak and newspeak with fopen()
  - build and populate hash table and bloomfilter with fscanf() [values come from newspeak.txt and badspeak.txt]
  - build two empty linked lists: rightspeak & thoughtcrime
  - use regex to read strings from stdin and populate rightspeak & thoughtcrime
  - if 's' argument is found, print statistics. else, print messages
  - close files and clear memory

```
while (word=next_word(reg_out) != NULL)
    if (bf_probe(word) = true)
        n = ht_lookup(word)
    if (n_newspeak = NULL)
        ll_insert(rightspeak, word)
    if (n_newspeak != NULL)
        ll_insert(thoughtcrime, word)
```

Nodes

```
node_create ( oldspeak , newspeak )
    < allocate  memory >
    (node) oldspeak  =  strdup ( oldspeak )
    (node) newspeak  =  strdup ( newspeak )
    (node) next = NULL
    (node) prev = NULL
    return N
```

node_create should build a new node structure and populate it with incoming values: setting its anterior and posterior to NULL.

```
node_delete (node **N )
    if  n  is  non-NULL
        free  n
        n = NULL
```

node_delete() should free all memory related to the input node.

```
node_print (node *N )
    printf ( N -> oldspeak , N -> newspeak , N -> prev , N -> next )
```

```
strdup (s)
    < allocate  memory >
    strcpy ( new_string ,  s )
    return  new_string
```

strdup() should emulate the built in function and return a copy of the input string.

Bit Vector

```
struct BitVector
    uint32 length
    uint32 *vector

bv_create ( length )
    < allocate memory >
    (bv) length = length
    (bv) vector [length]
    return bv

bv_delete ( v )
    free (*v -> vector)
    free (*v)
    v = NULL

bv_length ( v )
    return length

bv_set_bit ( v, i )
    v2 = 0x1
    byte = i/8 — of what byte
    index = i%8 — at what bit
    v2 = v2 shift left by index
    v[byte] = v[byte] [OR] v2

bv_clear_bit ( v, i )
    v2 = 0x1
    byte = i/8 — of what byte
    index = i%8 — at what bit
    v2 = v2 shift left by index
    v[byte] = v[byte] [AND] [not](v2)

bv_get_bit ( v, i )
    v2 = 0x1
    byte = i/8 — of what byte
    index = i%8 — at what bit
    v2 = v2 shift left by index
    v2 = v[byte] & v2
    v2 = v2 shift right by index
    return v2

bv_print ( v )
    for ( i (to) v->length )
        printf ( v->vector[i]);
    printf ( newline );
```

bv_create should create a new bitvector structure, which includes an array of bits and the total length.

bv_set_bit() should set a bit at a given index as 1. (use bitvector manipulation.)

bv_clr_bit() should set a bit at a given index as 0. (use bitvector manipulation.)

bv_get_bit() should return a bit at a given index. (use bitvector manipulation.)

Bloom Filter

```
bf_create()
    < allocate memory >
    (bf) primary [0] = given num
    (bf) primary [1] = given num
    (bf) secondary [0] = given num
    (bf) secondary [1] = given num
    (bf) tertiary [0] = given num
    (bf) tertiary [1] = given num
    (bf) filter = create bv
    return bf

bf_delete ( bf )
    bv_delete (filter)
    free (bf)
    bf = NULL

bf_size ( bf )
    return bv_length (bf -> filter)

bf_insert ( bf , oldspeak )
    index = hash((bf) primary , oldspeak) % bf.size ((bf) filter)
    bv_set ( (bf) filter, index)
    index = hash((bf) secondary, oldspeak) % bf.size ((bf) filter)
    bv_set ( (bf) filter, index)
    index = hash( (bf) tertiary , oldspeak) % bf.size ((bf) filter)
    bv_set ( (bf) filter, index)

bf_probe ( bf , oldspeak )
    index = hash( (bf) primary , oldspeak) % bf.size ( (bf) filter)
    bit1 = bv_get ( (bf) filter, index)
    index = hash((bf) secondary, oldspeak) % bf.size ( (bf) filter)
    bit2 = bv_get ( (bf) filter, index)
    index = hash( (bf) tertiary , oldspeak) % bf.size ( (bf) filter)
    bit3 = bv_get ( (bf) filter, index)
    if ((bit1 ^ bit2 ^ bit3) == 1)
        return true
    return false
```

bf_create() should create a new bloomfilter structure. The structure should include three, two element arrays of large numbers provided by the professor. A bv should also be added as “bf->filter”.

should get the hash value of a string and set a bit at that index in the bloom filter.

should get the bits at the index of each hash value and return whether all bits are 1 or not.

```
bf_count( bf )
    count = 0
    for (each bit in bf->filter)
        count += bv_get_bit
    return count

bf_print ( bf )
    bv_print (bf->filter)
```

bf_count() should return the number of set bits in the bloom filter.

Hash Table

```
ht_create ( mtf )
    < allocate  memory >
    (ht) salt [0] = given num
    (ht) salt [1] = given num
    (ht) size = size
    (ht) mtf = mtf
    (ht) lists = < allocate  memory >
    return ht
```

ht_create() should create a new hash table structure. The structure should include a two element arrays of large hex numbers provided by the professor, size, and a mtf boolean. An array should also be added as "ht->lists".

```
ht_delete (ht)
    for i in ht.size,
        ll.delete each lists[i]
    free lists[i]
    lists[] = NULL

ht_size ( ht )
    return ( ht -> size )

ht_lookup ( ht , oldspeak )
    index = hash ( bf-> primary , oldspeak ) % ht_size ( ht_lists )
    if ( !ht -> lists [index] )
        return NULL
    return ll_lookup ( ht -> lists [index] , oldspeak , newspeak )

ht_insert ( ht , oldspeak , newspeak )
    index = hash ( bf-> primary , oldspeak ) % ht_size ( ht_lists )
    if ( !ht -> lists [index] )
        return NULL
    ll_insert ( ht -> lists [index] , oldspeak , newspeak )

ht_count ( ht )
    count = 0
    for ( each bit in ht -> lists )
        count += bv_get_bit
    return count
```

should get the hash index of a string and return the function ll_lookup() at that index.

should get the hash index of a string and call ll_insert() at that index.

ht_count() should return the number of non-NULL linked lists in the hash table.

Linked list

```
ll_create (mtf)
    (ll) length   = 0
    (ll) mtf      = mtf
    (ll) head     = node_create ('head', NULL)
    (ll) tail     = node_create ('tail', NULL)
    (ll) head_next  = (ll) head
    (ll) tail_prev  = (ll) tail

ll_delete ( ll )
    while (*head != NULL)
        node *next = NULL
        next = *head -> next
        node_delete (head)
        *head = next
    return

ll_length ( ll )
    return (ll) length

ll_lookup ( ll , oldspeak)
    current =  head_next
    links += 1
    while (oldspeak  match  isn't  found)
            current =  current_next
            seeks += 1
    if ( current = tail)
        return NULL
    if ( not at head  and  mtf = true)
        < mtf logic >   check lab doc
    return  current

ll_insert ( ll , oldspeak , newspeak )
    insert_node = node_create ()

    insert_node_prev = head
    insert_node_next = head_next
    head_next_prev = insert_node
    head_next = insert_node
    length += 1
    return

ll_print ( ll )
    for ( head ->  tail / when NULL is reached)
        print ( curr->key )
```

ll_create() should create a new linkedlist structure that includes length, mtf, and a head and tail node.

ll_lookup() should iterate though all elements of a linked list searching for a given string. If that string is not found NULL should be returned. If the string is not already at the head of the linkedlist and mtf = true, the element should be moved to the front of the linked list. If the element is found it will be returned at the end.

ll_insert() should create a new node with key oldspeak and value newspeak. This node should be inserted in the front of the linked list and total length should be incremented by one.

ll_insert() Picture