Shweta Jones
CSE 13S - Spring 2021
Prof Darrell Long

Assignment 7 Design

Description:

This program uses several different ADTs including nodes, linked lists, hash tables, bit vectors and bloom filters. The program keeps track of badspeak and oldspeak words to decide whether or not the user gets accused of a thoughtcrime. When a basspeak doesn't have a translation, the user is accused of thoughtcrime and sent to joycamp, and otherwise they are sent to counsel. There are a few command line options: h, t, f, m, s. The '-h' option points out the help information which provides program usage information. The '-t' option specifies the size of the hash table, if this is not specified, the default size is set to 10000. If the user writes '-f', this specifies the size of the bloom filter (the default is 2^20). The '-m' option enables mtf, or move-to-front rule. If this is not selected, this rule is not applied. This '-s' option allows for the user to see the stats of the program which are: total number of seeks, average seek length, hash table load, and the bloom filter load. This program is run on a terminal and requires linux/unix. THe program also requires several c files: node.c, ll.c, ht.c, bf.c, bv.c, parser.c, and speck.c, along with their corresponding h files. The program also uses badspeak.txt and newspeak.txt. And finally the main file we run is banhammer.c.

File Breakdown:

Main:

banhammer.c

Other files:

ht.c → ll.c → node.c

bf.c → bv.c

Files:

Banhammer.c:

```
void helpInfo(void)
  Prints help information/usage information

int main(int argc, char **argv)
   int opt = 0
   uint32_t bfSize = BF_SIZE
   uint32_t htSize = HT_SIZE
   bool mtf = false
   bool stats = false
   while ((opt = getopt(argc, argv, OPTIONS)) != -1)
      switch(opt)
          case 'h'
                helpInfo()
                return 0
                break
```

```
        case 't'
                htSize = atoi(optarg)
                Check size
                break
        case 'f'
                bfSize = atoi(optarg)
                Check size
                break
        case 'm'
                mtf = true
                break
        case 's'
                stats = true
                break;
        default
                fprintf(stderr, "Usage %s -[ht:f:ms]\n", argv[0])
HashTable *ht = ht_create(htSize, mtf)

BloomFilter *bf = bf_create(bfSize)

char badList[10000]
FILE *badFile = fopen("badspeak.txt", "r")
while (fscanf(badFile, "%s\n", badList) != EOF)
   bf_insert(bf, badList)
      ht_insert(ht, badList, NULL)
fclose(badFile);

char oldList[4000]
char newList[4000]
FILE *newFile = fopen("newspeak.txt", "r")
while (fscanf(newFile, "%s %s\n", oldList, newList) != EOF)
   bf_insert(bf, oldList)
      ht_insert(ht, oldList, newList)
fclose(newFile)

char *word = NULL
regex_t re

LinkedList *forbidden = ll_create(mtf)
LinkedList *partial = ll_create(mtf)
while((word = next_word(stdin, &re)) !=NULL)
   for (uint32_t i = 0; i < strlen(word); i++)
        word[i] = tolower(word[i])
      if (bf_probe(bf, word) == true && ht_lookup(ht, word) != NULL)
```

```
            if (ht_lookup(ht, word)->newspeak != NULL)
                ll_insert(partial, ht_lookup(ht, word)->oldspeak, ht_lookup(ht,
word)->newspeak)
            else if(ht_lookup(ht, word)->newspeak == NULL)
                ll_insert(forbidden, ht_lookup(ht, word)->oldspeak, NULL)
    if (stats != true)
        if (ll_length(forbidden) != 0 && ll_length(partial) != 0)
            prints the needed information
        else if(ll_length(forbidden) != 0 && ll_length(partial) == 0)
            prints the needed information
        else if(ll_length(forbidden) == 0 && ll_length(partial) != 0)
            prints the needed information
    if (stats == true)
        prints stats information
    clear_words();
    regfree(&re);
    ll_delete(&forbidden);
    ll_delete(&partial);
    bf_delete(&bf);
    ht_delete(&ht);
    return 0;
```

ll.c:

```
LinkedList *ll_create(bool mtf):
    LinkedList *l = (LinkedList *) malloc(sizeof(LinkedList));
    l → length = 0
    l → mtf = mtf
    l → head = node_create(NULL, NULL)
    l → tail = node_create(NULL, NULL)
    l → head → next = l → tail
    tail → prey = l → head
    return 1
void ll_delete(LinkedList **ll):
    while (*ll) →  head = null
        node *temp = (*ll) → head → next
        node_delete(&(*ll) → head)
        (ll) → head = temp
    free(*ll)
    *ll = NULL

uint32_t ll_length(LinkedList *ll):
    return l->length

Node *ll_lookup(LinkedList *ll, char *oldspeak):
```

```
            for (Node *n = l → head → next; n != l → tail; n = n→ next)
                    if compare n→ oldspeak & oldspeak
                            if(l → mtf)
                                    x->prev->next = x->next
                                    x->next->prev = x->prev
                                    x->next = l->head->next
                                    x->prev = l->head
                                    l->head->next->prev = x
                                    l->head->next = x
                            return n
                    return NULL
    void ll_insert(LinkedList *ll, char *oldspeak, char *newspeak):
            Node *x = node_create(oldspeak, newspeak)
            x → next = l → head → next
            x → prey = l → head
            l → head → next → prey = x
            l → head → next = x
            l → length += 1

    void ll_print(LinkedList *ll):
            for (Node *n = l → head → next; n != l → tail; n = n→ next)
                    print(n)
```

node.c:

```
    Node *node_create(char *oldspeak, char *newspeak):
            malloc node
                    if malloc failed: return null
            if oldspeak != null
                    strdup oldspeak to n->oldspeak
            else
                    n->oldspeak == null
            if newspeak != null
                    strdup newspeak to n->newspeak
            else
                    n->newspeak == null
            n->next = null
            n->prev = null

    void node_delete(Node **n):
            free ((*n)->oldspeak)
            free((*n)->newspeak)
            free(*n)
            set *n = null
```

```
void node_print(Node *n):
    if n->oldspeak & n->newspeak != null
        print both
    if n->oldspeak != null & n->newspeak == null
        print n->oldspeak
```

ht.c:

```
void ht_delete(HashTable **ht):
    iterate from i =0 to i<ht_size(*ht)
        if(*ht)->lists[i]
            ll_delete(&(*ht)->lists[i]) free((*ht)->lists)
    free(*ht)
    *ht = null

uint32_t ht_size(HashTable *ht):
    return size

Node *ht_lookup(HashTable *ht, char *oldspeak):
    linkedlist *s = ht->lists(hash(salt, oldspeak)%ht_size)
    if malloc for s failed
        return null
    node *lookup_node = ll_lookup(s, oldspeak)
    return lookup_node

void ht_insert(HashTable *ht, char *oldspeak, char *newspeak):
    hashKey = hash(salt, oldspeak)%ht_size
    if(ht->list[hashKey] == null)
        ht->lists[hashKey] = ll_create(ht->mtf)
    ll_insert(ht->lists[hashKey], oldspeak, newspeak);

uint32_t ht_count(HashTable *ht):

void ht_print(HashTable *ht):
    from i=0 to ht_size
        linkedlist *p = ht->lists[i]
    if p!= null
        ll_print(p)
```

bv.c:

```
BitVector *bv_create(uint32_t length):
   BitVector *bv = (BitVector *) malloc(sizeof(BitVector))
   if (bv==NULL)
      return NULL
   bv->length = length
```

```
        bv->vector = (uint8_t *) calloc ((length/8), sizeof(uint8_t))
        return bv

    void bv_delete(BitVector **bv):
        free((*bv)->vector)
        free(*bv)
        *bv = NULL

    uint32_t bv_length(BitVector *bv):
        return bv->length

    void bv_set_bit(BitVector *bv, uint32_t i):
        uint8_t mask = 1 << (i%8)
        bv->vector[i/8] = bv->vector[i/8]&mask

    void bv_clr_bit(BitVector *bv, uint32_t i):
        uint8_t mask = ~(1 << (i%8))
        bv->vector[i/8] = bv->vector[i/8]&mask

    uint8_t bv_get_bit(BitVector *bv, uint32_t i):
        uint8_t mask = 1 << (i%8)
        return (bv->vector[i/8]&mask) >> (i%8)

    void bv_print(BitVector *bv):
        int count = 0
        for (uint32_t i = 0; i<bv_length(bv); i++)
            printf("%d", bv_get_bit(bv, i))
                if (bv_get_bit(bv, i) == 1)
                    count += 1
        printf("\n")
```

bf.c:

```
    BloomFilter *bf_create(uint32_t size):
        BloomFilter *bf = (BloomFilter*) malloc(sizeof(BloomFilter));
        if(bf)
            bf->primary[0] = 0x5adf08ae86d36f21
                bf->primary[1] = 0xa267bbd3116f3957
                bf->secondary[0] = 0x419d292ea2ffd49e
                bf->secondary[1] = 0x09601433057d5786
                bf->tertiary[0] = 0x50d8bb08de3818df
                bf->tertiary[1] = 0x4deaae187c16ae1d
                bf->filter = bv_create(size)
                if (!bf->filter)
                free(bf)
                bf = NULL
        return bf

    void bf_delete(BloomFilter **bf):
        bv_delete(&(*bf)->filter)
        free(*bf)
        *bf = NULL

    uint32_t bf_size(BloomFilter *bf):
        return bv_length(bf->filter)

    void bf_insert(BloomFilter *bf, char *oldspeak):
        uint32_t firstIndex = hash(bf->primary, oldspeak) % bf_size(bf)
        bv_set_bit(bf->filter, firstIndex)
        Repeat for second and third index

    bool bf_probe(BloomFilter *bf, char *oldspeak):
        uint32_t firstIndex = hash(bf->primary, oldspeak) % bf_size(bf)
        bv_get_bit(bf->filter, firstIndex)
        Repeat for second and third index
        if (firstIndex == 1 && secondIndex == 1 && thirdIndex == 1)
            return true;
        return false;

    uint32_t bf_count(BloomFilter *bf):
        uint32_t count = bf_size(bf)
        return count

    void bf_print(BloomFilter *bf):
        bv_print(bf->filter)
```