Shubham Singh / 2022300118 / Comps B / Batch C

Exp 9: Double hashing

```
typedef struct KeyValue {
        char* key;
        char* value;
        bool is deleted;
} KeyValue;

typedef struct HashTable {
        ~~char~~
        KeyValue ** array;
        int size;
        float load_factor;
        int num_keys;
        int num_occupied indices
        int num_ops
} HashTable;

KeyValue * Create_key_value (char* key, char * value) {
    KeyValue * newkeyvalue = malloc (sizeof (KeyValue));
    if (~~a~~KeyValue ≠ NULL) {
      newkeyvalue -> key = malloc ((strlen (key)+1) * sizeof(char));
      new~~k~~keyvalue -> value = malloc ((strlen (value)+1) * sizeof(char));
      strcpy (~~ht -> array~~ newkeyvalue -> key, key);
      strcpy (newkeyvalue -> value, value);
      newkeyvalue -> is Deleted = False;
    }
  return newkeyvalue;
}
```

```
HashTable *      createHashTable ( ) {
    HashTable *    newTable = (HashTable*) malloc (sizeof(HashTable))
    newTable -> array = (KeyValue**) malloc (TABLE_SIZE *
                                       sizeof (KeyValue*));
    for ( int i=0 ; i< TABLE_SIZE; i++) {
         newtable -> array [i] = NULL;
    }

    newTable  -> size =  TABLE_SIZE;
    newTable  -> load_factor = 0;
    newTable  -> num_keys = 0;
    newTable  -> num_occupied_indices = 0;
    newTable  -> num_ops = 0;
    return  newTable;
}


int  key_to_int ( char * key) {
     int  hash = 0,  ind = 0;
     while ~~(key [ind] != '0') {~~  (key [ind] != '\0') {
          hash += ((int) key [ind] + 128);
          ind++;
     }
    hash % =  TABLE_SIZE;
    return  hash;
 }


int   secondhash  (int  n) {
     return  ( 11 - (n % 11));
 }
```

```
int insert_key_value ( HashTable * ht, char * key, char * value)
    if (ht -> num_occupied_indices == TABLE_SIZE ){
        return -1;
    }
    int h1 = key_to_int (key);
    int retval;
    KeyValue * to_insert = create KeyValue (key, value);
    if (ht -> array [h1] == NULL) {
        ht -> array [h1] = to_insert ;
        ht -> num_ops ++;
        retval = h1;
    }
    else if (ht -> array [h1] ~~=~~-> is_Deleted == True) {
        ht -> array [h1] -> is_Deleted = False;
        strcpy ( ht -> array [h1] -> key, key);
        strcpy ( ht -> array [h1] -> value, value);
        free (to_insert);
        ht -> num_ops ++;
        retval = h1;
    }
    else {
        int h2 = second hash (h1)
        int index = h1 ;
        while (ht -> array [index] != NULL) {
            if (ht -> array [index] -> is Deleted == True) {
                ht -> array [index] -> is Deleted = False;
                strcpy (ht -> array [index] -> key, key);
                strcpy (ht -> array [index] -> value, value);
                free (to_insert);
                retval = index ;
                ht -> num_ops ++;
                goto was deleted;
            }
            index ~~=~~+= h2;
            index % = TABLE_SIZE;
            ht -> num ops ++;
            if (index == h1) {
                return -1;
            }
        }
        ht -> array [index] = to_insert;
        retval = index;
        ht -> num_ops++;
    }
    was deleted:
    ht -> num keys ++;
    ht -> num_occupied_indices ++;
    return retval;
}


char* search_key (HashTable * ht, char* key) {
    int h1 = key_to_int (key);
    int index = h1;
    if (ht -> array [index] == NULL) {
        return NULL;
    }
    else if ( strcmp ( ht -> array [index] -> key, key) == 0 &&
                ht -> array [index] -> isDeleted == false) {
        ht -> num_ops ++;
        return ht -> ~~{~~array [index] -> value;
    }
    else {
                                index
        int h2 = second hash (~~h1~~);
```

```
for (int i=0;  i< TABLE_SIZE; i++){
        index += h2;
        index % = TABLE_SIZE;
        ht -> num_ops ++;
        if (ht -> array [index] == NULL) {
            return NULL;
        }
        else if (~~ht->~~ strcmp (ht ->array(index] -> key, key)==0
                  && ht ->array [index] ->is Deleted == false) {
            return ht -> array [index] -> value;
        }
        else if (index == h1) {
            return NULL;
        }
    }
}
return NULL;
}


int delete_key (HashTable * ht, char * key) {
    int n = ht-> num_ops;
    char * temp = search_key (ht, key);
    ht -> num_ops ~~++~~ = n;
    if (temp == NULL) {
        return -1;
    }
    int index = key_to_int (key);
    int h2 = secondhash (index);
```

```
        while (strcmp (ht -> array [index] -> key, key)!=0) {
            ht -> num_ops ++;
            index += h2;
            index %= TABLE_SIZE;
        }
        ht -> num_ops ++;
        ht -> array [index] -> is Deleted = True;
        ht -> num_keys --;
        ht -> num_occupied_indices --;
        return index;
    }

float get_load_factor (HashTable * ht) {
      float lf = (float) ht -> num_keys / TABLE_SIZE;
      ht -> load_factor = lf;
       return lf;
  }

 float get_average_probes (HashTable * ht ) {
    return ht-> num_ops /(float) ht -> num_occupied
                                         indices ;
    }

void display (HashTable * ht) {
    printf ("displaying Hash table :\n");
    printf ("\n index \t %-35s \t %-35s \n\n", "KEY",
                                          "VALUE");
    for (int i=0; i< TABLE_SIZE; i++) {
        if (ht -> array [i] == NULL {
            printf ("%-5d \t %-35s \t %-35s \n", i,
                                 "NULL", "NULL");
        }
                                                  ->
        else if ( ht->array [i] -> isDeleted == True) {
            printf ( " %-5d,\t %-35s \t %-35s \n", i,
                                                    "deleted", "deleted" );

            }
        else {
            printf (" %-5d,\t %-35s \t %-33s \n ", i,
                        ht ->array [i] -> key, ht->array [i]->value );
        }
    }
    printf ("\n");
}}
```