Worksheet 37 - Group 1

Worksheet Group 1 Members

Marc Clinedinst: clinedim@onid.oregonstate.edu
Kelby Faessler: faesslek@onid.oregonstate.edu
James Fitzwater: fitzwatj@onid.oregonstate.edu
Tom Gariepy: gariepyt@onid.oregonstate.edu
Sean Reilly: reillys@onid.oregonstate.edu
Joseph Struth: struthj@onid.oregonstate.edu

Collaborators

Marc, Kelby, James, Tom, Sean, Joseph

Worksheet 37: Hash Tables (Open Address Hashing)

In this worksheet, we provide implementations for hash table functions which are implemented on top of the dynamic array data structure. More specifically, we implement the following hash functions which are based on open address hashing: `addOpenHashTable`, `containsOpenHashTable`, and `_resizeOpenHashTable`. The implementation of these functions are below.

Since this data structure is based on open address hashing, we will first need to define a helper `_hash` function. This function appears below.

```
/*
     This function performs the hash conversion for a particular word.  It first checks to
     make sure that the pointer to the c string is not null, that the table size is greater
     than zero, and that the word is at least three characters long.  If this is the case, it
     takes the third letter of the word and converts it to an integer value.  This value is
     then subtracted by 97 (the ascii value of the letter 'a'--we are assuming that all words
```

```
     are lowercase), and then performs the modulus operation with the size of the table. This
     value, the hash value, is returned.
*/
int _hash(char **word, int tableSize) {
     assert(word != 0);
     assert(tableSize > 0);
     assert(strlen(*word) >= 3);

     int ascii = (*word)[2];

     return (ascii - 97) % tableSize;
}

/*
     This function adds a value to the hash table.  It first checks to make sure that
     the hash table and value are not null.  It then computes the load factor and resizes
     the hash table if the load factor is greater than 0.75. The hash value is computed,
     and the function searches for the first empty index starting with the hash value.  Once
     this index is located, the value is placed there.  The count is incremented.
*/
void addOpenHashTable(struct OpenHashTable *oht, TYPE *value) {
     assert(oht != 0);
     assert(value != 0);

     double loadFactor = (double) oht->count / sizeOpenHashTable(oht);

     if (loadFactor > 0.75) {
          _resizeOpenHashTable(oht);
     }

     int index = _hash(value, sizeOpenHashTable(oht));
```

```
	while (oht->table[index] != 0) {
		index++;
		if (index >= oht->tableSize) {
			index = 0;
		}
	}

	oht->table[index] = value;
	oht->count++;
}

/*
	This function checks to see if a particular value appears within a hash table.  It
	first checks to make sure that the hash table and value are not null.  It then calculates
	the hash value for that value.  It begins probing for that value and returns 1 if the
	value is located.  If the linear probe encounters an empty index, then it returns the
	value 0.
*/
int containsOpenHashTable(struct OpenHashTable *oht, TYPE *value) {
	assert(oht != 0);
	assert(value != 0);

	int index = _hash(value, sizeOpenHashTable(oht));

	while (oht->table[index] != 0) {
		if (strcmp(*(oht->table[index]), *value) == 0) {
			return 1;
		} else {
			index++;
			if (index >= sizeOpenHashTable(oht)) {
```

```
                    index = 0;
                }
            }
        }

        return 0;
}

/*
        This function doubles the size of a hash table.  It begins by making sure that the
        hash table is not null.  It then creates a new hash table that has twice the space
        for storing values as the existing table.  It copies all of the values from the existing
        hash table to the new table.  The current hash table's table is replaced with the new
        hash table and the old hash tabled is freed.
*/
void _resizeOpenHashTable(struct OpenHashTable *oht) {
        assert(oht != 0);

        int index = 0,
            oldTableSize = oht->tableSize;
        oht->tableSize = 2 * oht->tableSize;
        TYPE **oldTable = oht->table;
        oht->table = (TYPE **) malloc(oht->tableSize * TYPE_SIZE);
        assert(oht->table != 0);

        for (index = 0; index < oldTableSize; index++) {
            if (oldTable[index] != 0) {
                addOpenHashTable(oht, oldTable[index]);
                oht->count--;
            }
        }
```

```
      free (oldTable);
}
```

Piazza Discussion Post

https://piazza.com/class/ib2kus4hsie528?cid=255