Worksheet 38 - 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 38: Hash Tables using Buckets

In this worksheet, we provide implementations for hash table functions which are implemented on top of an array which stores a linked list at each index. More specifically, we implement the following hash functions which are based on a hash table using buckets: `initHashTable`, `addHashTable`, `containsHashTable`, `removeHashTable`, and `_resizeHashTable`. The implementation of these functions are below.

Since this data structure is based on hashing, we will first need to define a `_getHashIndex` function. This function is defined below, with an explanatory comment.

```
/*
     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 _getHashIndex(char *word, int tableSize) {
     assert(word != 0);
     assert(tableSize > 0);
     assert(strlen(word) >= 3);

     int ascii = word[2];

     return (ascii - 97) % tableSize;
}
```

With this hash function defined, we can now move onto defining the other functions which control the behavior of the hash table. These functions are defined below with explanatory comments for each function.

```
/*
     This function adds a value to the hash table.  It first makes sure that the passed hash
     table is not null.  The function then calculates the hash value.  Next, it creates a new
     link and inserts it at the beginning of the linked list; the hash table’s count is also
     incremented.  Next, the function calculates the load factor and resizes the table if
     necessary.
*/
void addHashTable(struct HashTable *ht, TYPE value) {
     assert(ht != 0);

     int hashIndex = _getHashIndex(value, sizeHashTable(ht));

     struct Link *newLink = (struct Link *) malloc(sizeof(struct Link));
     assert(newLink != 0);
     newLink->value = value;
```

```
	newLink->next = ht->table[hashIndex];
	ht->table[hashIndex] = newLink;
	ht->count++;

	float loadFactor = (double) ht->count / sizeHashTable(ht);

	if (loadFactor > 8.0) {
		_resizeTable(ht);
	}
}

/*
	This function checks to see if a function contains a particular value. It first checks
	to make sure that the hash table is not null and the value is not null.  It then
	calculates the hash index and searches through the appropriate index to find the value.
	If the value is found, the function returns 1.  If the value is not found, the function
	returns 0.
*/
int containsHashTable(struct HashTable *ht, TYPE value) {
	assert(ht != 0);
	assert(value != 0);

	int hashIndex = _getHashIndex(value, sizeHashTable(ht));
	struct Link *current = ht->table[hashIndex];

	while (current) {
		if (strcmp(current->value, value) == 0) {
			return 1;
		}
		current = current->next;
	}
	return 0;
}

/*
	This function initializes a hash table.  It first checks to make sure that the passed
	hash table is not null.  It then allocates memory to store the hash table and checks that
	the allocation was successful.  The function then sets the count and tableSize fields for
	the hash table.  The function then loops through the table and sets each index to null.
*/
void initHashTable(struct HashTable *ht, int tableSize) {
	assert(ht != 0);
	ht->table = (struct Link **) malloc(sizeof(struct Link **) * tableSize);
	assert(ht->table != 0);
	ht->count = 0;
	ht->tableSize = tableSize;
	int index = 0;

	for (index; index < tableSize; index++) {
		ht->table[index] = 0;
	}
}

/*
	This function removes a value from the hash table.  It first checks to make sure that the
	hash table is not null, that the value is not null, and that the size of the hash table is
	greater than 0.  The function then checks to see if the value is in the hash table.  If
	not, the function terminates.  If the value is in the hash table, the function begins
	searching for the value from the hash index.  Once it is located, the function removes the
	value and decrements the count.
*/
```

```
void removeHashTable(struct HashTable *ht, TYPE value) {
	assert(ht != 0);
	assert(value != 0);
	assert(sizeHashTable(ht) > 0);

	if (containsHashTable(ht, value)) {
		int hashIndex = _getHashIndex(value, sizeHashTable(ht));
		struct Link *current = ht->table[hashIndex];
		struct Link *previous = ht->table[hashIndex];
		while (current) {
			if (strcmp(current->value, value) == 0) {
				if (current == ht->table[hashIndex]) {
					ht->table[hashIndex] = current->next;
				} else {
					previous->next = current->next;
				}
				free(current);
				ht->count--;
				return;
			} else {
				previous = current;
				current = current->next;
			}
		}
	}
}
```

Piazza Discussion Post

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