Worksheet 36 - 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 36: Dynamic Array Dictionary

In this worksheet, we analyze the complexity and provide implementations for several dictionary functions which are implemented on top of the dynamic array data structure. More specifically, we focus on the following dictionary operations: `containsDictionaryDynArr`, `getDictionaryDynArr`, `putDictionaryDynArr`, and `removeDictionaryDynArr`. The analysis and implementation of these functions are below.

We begin by analyzing the four functions above based on our implementations of these functions. This analysis appears in the table below:

Function	Complexity
`containsDictionaryDynArr`	O(n)
`getDictionaryDynArr`	O(n)
`putDictionaryDynArr`	O(n)
`removeDictionaryDynArr`	O(n)

With these functions now analyzed, we now provide implementations of the functions. Please note that some of these functions are based on implementations of functions that we defined in previous worksheets where we developed the functionality of the DynArr; these functions will not be reproduced in this worksheet. We have also created a couple of helper functions, which will be defined first. Each function will be accompanied by comments where necessary.

We will start by defining our helper functions. These are below.

```
/*
     This function compares two keys to determine if they are equal.
     It first checks that the left and right values are not null; if
     these checks fail, then an assertion error will occur.  Since
     the keys are string values, the function simply returns the
     value of calling the function strcmp on the values of left and
     right.  The function will return a negative value if left is
     less than right. It will return zero if the values are equal. It
     will return a positive value if left is greater than right.
*/
int compare(KEYTYPE left, KEYTYPE right) {
     assert(left != 0);
     assert(right != 0);
     return strcmp(left, right);
}
```

```
/*
     This function creates a new association using a key and value passed  by the user. It
     does so by allocating memory for a new association and checking that the allocation was
     successful.  It then allocates memory for the key and makes sure that this allocation
     was successful.  Finally, it assigns the key and value to the association, and returns
     a pointer to the association.
*/
struct Association *_createAssociation(KEYTYPE key, VALUETYPE value) {
     struct Association *a = (struct Association *) malloc(sizeof(struct Association));
     assert(a != 0);
     a->key = (char *) malloc(sizeof(char *));
     assert(a->key != 0);
     strcpy(a->key, key);
     a->value = value;
     return a;
}
```

With these helper functions defined, we can now move on to defining the functions which define the behavior of the dictionary that is implemented on top of the dynamic array.

```
/*
     This function returns an integer value representing whether a particular key appears in
     a dictionary.  Specifically, the function returns 1 if the key does appear in the
     dictionary; it returns 0 if they key does not appear.  The function first checks to make
     sure that the dynamic array is not null.  It then performs a linear search in the
     dictionary for the key and returns the appropriate integer value.
*/
int containsDictionaryDynArr(struct DynArr *da, KEYTYPE key) {
     assert(da != 0);
     int index = 0;
```

```
	for (index; index < sizeDynArr(da); index++) {
		if (compare(da->data[index]->key, key) == 0) {
			return 1;
		}
	}

	return 0;
}

/*
	This function gets the value that is associated with a particular key in a dictionary.
	It first checks to make sure that the dictionary is not null.  Next, it checks to make
	sure that the dictionary contains the key.  If either of these checks fails, an assertion
	error occurs.  Next, the function performs a linear search for the key and returns the
	value associated with it once it is found.
*/
VALUETYPE getDictionaryDynArr(struct DynArr *da, KEYTYPE key) {
	assert(da != 0);
	assert(containsDictionaryDynArr(da, key));
	int index = 0;

	for (index; index < sizeDynArr(da); index++) {
		if (compare(da->data[index]->key, key) == 0) {
			return da->data[index]->value;
		}
	}
}


/*
```

```
	This function adds or updates a key-value pair in the dictionary.  The function first
	makes sure that the passed dictionary is not null.  It then checks to see if the key is
	already in the dictionary; if so, the function removes the existing key-value par from
	the dictionary.  Finally, the function adds the new key-value pair to the dictionary.
*/
void putDictionaryDynArr(struct DynArr *da, KEYTYPE key, VALUETYPE value) {
	assert(da != 0);
	if (containsDictionaryDynArr(da, key)) {
		removeDictionaryDynArr(da, key);
	}
	addDynArr(da, _createAssociation(key, value));
}

/*
	This function removes a key-value pair from a dictionary.  It first checks that the
	dictionary is not null, not empty, and that the dictionary contains the passed key.
	If any of these checks fails, then an assertion error occurs.  The function then performs
	a linear search through the dictionary for the specified key and removes the key-value
	pair from the dictionary.
*/
void removeDictionaryDynArr(struct DynArr *da, KEYTYPE key) {
	assert(da != 0);
	assert(sizeDynArr(da) > 0);
	assert(containsDictionaryDynArr(da, key));
	int index = 0;

	for (index; index < sizeDynArr(da); index++) {
		if (compare(da->data[index]->key, key) == 0) {
			removeAtDynArr(da, index);
			return;
		}
```

```
        }
}
```

In addition to the responses above, there was a second version of the worksheet worksheet which asked that we provide an alternate version of the getDeictionaryDynArr function. This version is below:

```
void getDictionaryDynArr(struct DynArr *da, KEYTYPE key, VALUETYPE *valueptr) {
        assert(da != 0);
        assert(containsDictionaryDynArr(da, key));
        int index = 0;

        for (index; index < sizeDynArr(da); index++) {
                if (compare(da->data[index]->key, key) == 0) {
                        valueptr = da->data[index]->value;
                        return;
                }
        }
}
```

Although the implementation is slightly different, the complexity of this function remains O(n).

Piazza Discussion Post

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