Worksheet 36: Dynamic Array Dictionary
**Group 11**

In Preparation: Read Chapter 12 to learn more about the Dictionary data structure. If you have not done so already, complete worksheets 14 and 16 to learn more about the dynamic array.

In this lesson you will once again use a dynamic array as an underlying data container, only this time implementing a dictionary. A dictionary is an indexed collection class. This means that elements are always provided in a pair consisting of a key and a value. Just as we used a symbolic constant to define the TYPE for a vector, we will use two symbolic types to define both the KEYTYPE and the VALUETYPE for our vector. The interface file for this abstraction is shown on the next page. By default, we will use a character pointer for the key, and a double for the value. Instead of the EQ and LE macros used with the vector, we will use a compare() function. The dictionary API is shown near the bottom of the page. When a value is inserted both the key and the value are provided. To search the collection the user provides a key, and the corresponding value is returned. A dictionary that associates words and definitions is a good mental model of a Map.

The idea behind the **DynamicArrayDictionary** is that internally elements in the dynamic array are stored as instances of struct **Association**. The internal struct **Association** stores a key and a value.

```
struct association {
    KEYTYPE key;
    VALUETYPE value;
}:
```
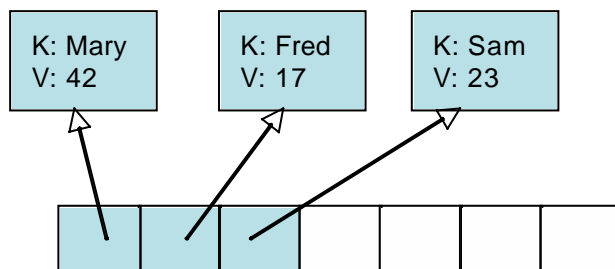

Each element in the Dynamic Array is a pointer to an Association:

When searching to see if there is an entry with a given key, for example, each element of the dynamic array is examined in turn. The key is tested against the search key, and if it matches the element has been found.

A similar approach is used to delete a value. A loop is used to find the association with the key that matches the argument. Once the index of this association is found, the dynamic array **remove** operation is used to delete the value.

Elements in a dictionary must have unique keys. Within the method **put** one easy way to assure this is to first call **containsKey**, and if there is already an entry with the given key call remove to delete it. Then the new association is simply added to the end.

```
void dyArrayDictionaryPut (struct dyArray *vec, KEYTYPE key, VALUETYPE val) {
    struct association * ap;
    if (dyArrayDictionaryContainsKey(vec, key))
        dyArrayDictionaryRemoveKey (vec, key);
    ap = (struct association *) malloc(sizeof(struct association));
    assert(ap != 0);
    ap->key = key;
    ap->value = val;
    dyArrayAdd(vec, ap);
}
```

Based on your implementation, fill in the following table with the algorithmic execution time for each operation:

int containsKey (KEYTYPE key)	O(n)
VALUETYPE get (KEYTYPE key)	O(n)
put (KEYTYPE key, VALUETYPE value)	O(n)
void remove (KEYTYPE key)	O(n)

```
# ifndef DYARRAYDICTH
# define DYARRAYDICTH

/*
   dynamic array dictionary interface file
*/

# ifndef KEYTYPE
# define KEYTYPE char *
# endif

# ifndef VALUETYPE
# define VALUETYPE double
# endif

struct association {
  KEYTYPE key;
  VALUETYPE value;
};

# define TYPE struct association *

# include "dyArray.h"

/* dictionary */
VALUETYPE dyArrayDictionaryGet (struct dyArray * da, KEYTYPE key);
```

```
void dyArrayDictionaryPut (struct dyArray * da, KEYTYPE key, VALUETYPE val);
int dyArrayDictionaryContainsKey (struct dyArray * da, KEYTYPE key);
void dyArrayDictionaryRemoveKey (struct dyArray * da, KEYTYPE key);

# endif
```

```
# include "dyArrayDictionary.h"
# include "dyArrayDictionary.c"

/*finds and places the value associated with key in valptr */
VALUETYPE  dyArrayDictionaryGet (struct dynArr *da, KEYTYPE key) {


Struct association * structure;
Assert (dyArrayDictionaryContainsKey(da, key));
For (int I = 0; I < da->size, i++({
        if(((EQ(struct association *)da->data[i])->key, key){
                structure = (struct assosication *) getDynArr(da, I):
        }
}
Return 0;

}


Where getDynArr is the function from the previous worksheets/assignments:

TYPE getDynArr(DynArr *v, int pos)
{
        /* FIXME: You will write this function */
        assert(v!=0);
   assert(!isEmptyDynArr(v));
   assert(pos < v->size && pos>=0);
        return v->data[pos];

        /* FIXME: you must change this return value */
        //return 1;
}
void dyArrayDictionaryPut (struct dynArr *da, KEYTYPE key, VALUETYPE val) {
     struct association * ap;
     if (dyArrayDictionaryContainsKey(da, key))
          dyArrayDictionaryRemove(da, key);
     ap = (struct association *) malloc(sizeof(struct association));
     assert(ap != 0);
     ap->key = key;
     ap->value = val;
     dyArrayAdd(da, ap);
}

int dyArrayDictionaryContainsKey (struct dynArr *da, KEYTYPE key) {
     int i = 0;
     for(i = 0; i < da->size; i++){
         if(((EQ(struct association *)da->data[i])->key, key)
             return 1;
         }
```

```
        return 0;
}




void dyArrayDictionaryRemove (struct  dynArr *da, KEYTYPE key) {

struct association * remove;
if(((EQ(struct association *)da->data[i])->key, key) // this is an indication that we found what we
were looking for
{
        Remove = (struct association *) (da->vada[i]);
        removeAtDynArr(da,i);
        free(remove);




}

Where remove at function is from the previous worksheets/assignments
void removeAtDynArr(DynArr *v, int idx)
{
        /* FIXME: You will write this function */
    int i;
    assert(v!=NULL);
    assert(!isEmptyDynArr(v));
    for(i = idx; i < v->size; i++){
       v->data[i] = v->data[i + 1];
    }
    v->size--;
}
```