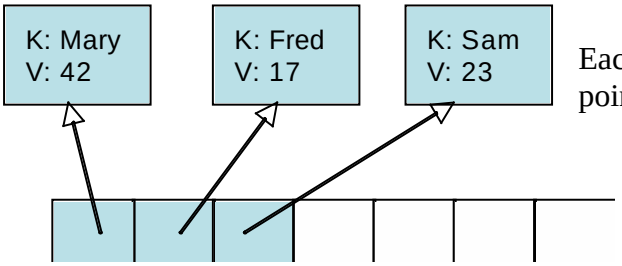
Worksheet 36: Dynamic Array Dictionary

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

To extract a value you use the function get. Rather than return the association element, this function takes as argument a pointer to a memory address where the value will be stored. This technique allows us to ignore an extra error check; if there is no key with the given value the function should do nothing at all.

```
void dyArrayDictionaryGet (struct dynArray *da, KEYTYPE key, VALUETYPE *valptr)
```

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

int containsKey (KEYTYPE key)	O(n)
Void get (KEYTYPE key, VALUETYPE *ptr)	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 "dynamicArray.h"
```

```
/* dictionary */
void dyArrayDictionaryGet (struct dynArray *da, KEYTYPE key, VALUETYPE *valptr)
void dyArrayDictionaryPut (struct dynArray * da, KEYTYPE key, VALUETYPE val);
int dyArrayDictionaryContainsKey (struct dynArray * da, KEYTYPE key);
void dyArrayDictionaryRemoveKey (struct dynArray * da, KEYTYPE key);

# endif
```

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

/*finds and places the value associated with key in valptr */
void dyArrayDictionaryGet (struct dynArray *da, KEYTYPE key, VALUETYPE *valptr){
        assert(da != 0);

        for(int i = 0; i < da->size; i++){
                if((da->data[i])->key == key){          //(da->data[i]) is an association pointer
                        (*valptr)=(da->data[i])->val;   //dereference *valptr to store val
                        return;
                }
}

void dyArrayDictionaryPut (struct dynArray *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 dynArray *da, KEYTYPE key) {
        assert(da != 0);

        for(int i = 0; i < da->size; i++){
                if((da->data[i])->key == key){
                        return 1;                       //key was found
                }
        }
        return 0;                                       //key was not found
}

void dyArrayDictionaryRemoveKey (struct  dynArray *da, KEYTYPE key) {
        assert(da != 0);

        for(int i = 0; i < da->size; i++){
                if((da->data[i])->key == key){
                        free(da->data[i]);              //memory referenced by pointer freed
                        removeAtDynArr(da, i);          //association pointer removed
                }
}
```