**Worksheet 10: Using Big-Oh to Estimate Wall Clock Time**

**In preparation**: Read Chapter 4 to learn more about the concept of big-oh notation.

As you learned in Chapter 4, a big-Oh description of an algorithm is a characterization of the change in execution time as the input size changes. If you have actual execution timings (“wall clock time”) for an algorithm with one input size, you can use the big-Oh to estimate the execution time for a different input size. The fundamental equation says that the ratio of the big-Oh’s is equal to the ratio of the execution times. If an algorithm is O(f(n)), and you know that on input n1 it takes time t1, and you want to find the time t2 it will take to process an input of size n2, you create the equation

f(n1) / f(n2) = t1 / t2

To illustrate, suppose you want to actually perform the mind experiment from Lesson 7. You ask a friend to search for the phone number for “Chris Smith” in 8 pages of a phone book. Your friend does this in 10 seconds. From this, you can estimate how long it would take to search a 256 page phone book. Remembering that binary search is O(log n), you set up the following equation:

log(8)/log(256), which is 3 / 8 = 10 / X

Solving for X gives you the answer that your friend should be able to find the number in about 24 seconds. Now you time your friend perform a search for the name attached to a given number in the same 8 pages. This time your friend takes 2 minutes. Recalling that a linear search is O(n), this tells you that to search a 256 page phone could would require:

8/256 = 2 / X

Solving for X tells you that your friend would need about 64 minutes, or about an hour. So a binary search is really faster than a linear search.

Recall from the last lesson the table of execution times for common algorithms. Using these, estimate the actual execution times for various tasks in the following problems.

|  |  |
| --- | --- |
| Linear search | O(n) |
| Binary search | O(log n) |
| countOccurrences | O(n) |
| isPrime | O(Sqrt(n)) |
| printPrimes | O(nSqrt(n)) |
| matMult | O(n^3) |
| SelectionSort | O(n^2) |

**Worksheet 14: Introduction to the Dynamic Array**

**In Preparation**: Read Chapter 5 to learn more about abstraction and the basic abstract data types, and the dynamic array as a n implementation technique.

Suppose you want to write a program that will read a list of numbers from a user, place them into an array and then compute their average. How large should you make the array? It must be large enough to hold all the values, but how many numbers will the user enter? A common solution to this difficulty is to create an array that is larger than necessary, and then only use the initial portion. This is termed a *partially filled array*.

When you use a partially filled array there are two integer values of interest. First is the array length. When discussing partially filled array this is sometimes termed the *capacity* of the array. Second is the current *size*, that is, the amount of the array that is currently being used. The size is generally maintained by a separate integer variable. If we place the values into a structure it will be easier to keep them together.

struct partFillArray {

double data[50];

int size;

int capacity;

};

Array2

It is important to not confuse the size and the capacity. For example, in computing the sum and average you do not want to use the capacity:

double average (struct partFillArray \* pdata) {

double sum = 0.0;

int i;

for (i = 0; i < 50; i++) /\* Error–loop using length \*/

sum = sum + pdata->data[i]; /\* Error-uninitialized value \*/

return sum / 50; /\* Error–average is under estimated \*/

}

Instead, you want to use the size:

double average (struct partFillArray \* pdata) {

double sum = 0.0;

int i;

for (i = 0; i < pdata->size; i++)

sum = sum + pdata->data[i];

return sum / pdata->size;

}

The technique of partially filled arrays works fine until the first time that the user enters more numbers than were originally anticipated. When this happens, the size can exceed the capacity, and unless some remedial action is taken an array indexing error will occur. Worse yet, since the validity of index values is not checked in C, this error will not be reported and may not be noticed.

A common solution to this problem is to use a pointer to a ***dynamically allocated array***, rather than a fixed length array as shown above in the partFillArray. Of course, this means that the array must be allocated before it can be used. We can write an initialization routine for this purpose, and a matching routine to free the data. Next, we likewise encapsulate the action of adding a new element into a function. This function can check that the size does not exceed the capacity, and if it does increase the length of the array (generally by doubling) and copying all the elements into the new area. Now the user can enter any number of values and the data array will be automatically expanded as needed.

# ifndef TYPE

# define TYPE int

# endif

struct DynArr

{

TYPE \*data; /\* pointer to the data array \*/

int size; /\* Number of elements in the array \*/

int capacity; /\* capacity of the array \*/

};

void initDynArr(struct DynArr \*v, int capacity)

{

v->data = malloc(sizeof(TYPE) \* capacity);

assert(v->data != 0);

v->size = 0;

v->capacity = capacity;

}

void freeDynArr(struct DynArr \*v)

{

if(v->data != 0)

{

free(v->data); /\* free the space on the heap \*/

v->data = 0; /\* make it point to null \*/

}

v->size = 0;

v->capacity = 0;

}

int sizeDynArr( struct DynArr \*v)

{

return v->size;

}

void addDynArr(struct DynArr \*v, TYPE val)

{

/\* Check to see if a resize is necessary \*/

if(v->size >= v->capacity)

\_setCapacityDynArr(v, 2 \* v->capacity);

v->data[v->size] = val;

v->size++;

}

The only thing missing now is the \_setCapacityDynArr function. Complete the implementation of \_setCapacityDynArr. Pay careful attention to the order of operations. Remember that since you’re creating a new array, you’ll want to eventually get rid of the old one to avoid a ‘memory leak’.

void \_setCapacityDynArr(struct DynArr \*v, int newCap)

{

//create new array

struct DynArr newArr;

initDynArr (&newArr, newCap);

//set size

newArr->size = v->size;

//copy data

for (int i = 0; i < v->capacity; i++) {

newArr[i] = v[i];

}

//free memory

struct DynArray \*temp = v;

v = newArr;

freeDynArr(&temp);

}

The code shown above introduces a number of features of the C language that you may not have seen previously. Let us describe some of these here.

TYPE. We want to create a library of general-purpose functions for managing collections of various types of elements. In order to make our code completely independent from the type of value being stored, we have defined the element type using a symbolic name, TYPE. The use of ifdef surrounding the definition is a common C idiom. If the user has already provided an alternative definition we will use that, otherwise the symbolic name is given a default value of int.

initDynArr and freeDynArr. The C language does not provide a way to automatically initialize a structure, such as a constructor does for you in Java or C++. Instead, programmers must typically write a special initialization function. Programmers must then remember to initialize a structure, and to free memory when they are finished with the structure.

struct DynArr myData; /\* create a new dynamic array \*/

…

initDynArr (&myData, 50); /\* initial capacity is 50 elements \*/

…

freeDynArr (&myData);

Notice that the structure is declared as a simple variable. However, because the functions require a pointer, the *address-of* operator (the ampersand) is used to produce a pointer to the structure.

da->size. Whenever you use a pointer you must make it clear when you are referring to the pointer itself and when you are referring to the value it points to. Normally a pointer value must first be dereferenced, using the \* operator, to indicate that you mean the value it points to, not the pointer itself. Accessing a field in a structure referred to by a pointer could be written using the dereference operator, as in (\*da).size. However, this combination of pointer dereferencing and field access occurs so frequently that the designers of C provided a convenient shorthand.

malloc. The function malloc is used to perform *memory allocation* in C. The argument is an integer indicating the number of bytes requested. In order to determine how many bytes are required for each element the function sizeof is invoked. Multiply the number of elements you need by the size of each element, and you have a block of memory that can be used as an array.

assert. The malloc function will return zero if there is insufficient memory to satisfy a request. The assert macro will halt execution with an error message if its argument expression is not true. Assertions can be used any time a condition must be satisfied in order to continue.

free. The function free is the opposite of malloc. It is used to return a block of memory to the free store. Such memory might later be reused to satisfy a subsequent malloc request. You should never use malloc without knowing where and when the memory will subsequently be freed.

*Defensive programming*. When the memory is released in the function freeDynArr the size and the capacity are both set to zero. This ensures that if a subsequent attempt is made to insert a value into the container, there will not be an attempt to index into an already deleted array.

sizeDynArr. Since the size field is stored as part of the dynamic array structure there really is no need for this function, since the user can always access the field directly. However, this function helps preserve encapsulation. The end user for our container need not understand the structure definition, only the functions needed to manipulate the collection. *How would your code have to change in order to completely hide the structure away from the user?*

\_setCapacityDynArr. An underscore is treated as a legal letter in the C language definition for the purposes of forming identifiers. There is a common convention in the C programming community that function names beginning with an underscore are used “internally”, and should never be directly invoked by the end user. We will follow that convention in our code. The function \_setCapacityDynArr can be called by dynamic array functions, but not elsewhere.

Note carefully the order of operations in the function \_setCapacityDynArr. First, the new array is created. Next, the old values are copied into the new array. The free statement then released the old memory. Finally, the pointer is changed to reference the new array.

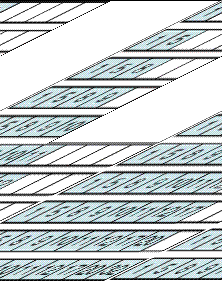
In order to allow a dynamically allocated array to be used in the same fashion as a normal array, we need functions that will get and set values at a given position. We can also make our function more robust than the regular C array by checking index positions. Complete the implementation of the following functions. *Use assert to check that index positions are legal*.

# Worksheet 15: Amortized Constant Execution Time

**In preparation**: Read Chapter 5 to learn more about the basic abstract data types, and the introduction to the dynamic array. If you have not done so already you should complete worksheet 14, which introduces the dynamic array.

In the previous worksheet you analyzed the algorithmic execution time for the **add** operation in adynamic array. When the size was less than the capacity, the execution time was constant. But when a reallocation became necessary, execution time slowed to O(n).

This might at first seem like a very negative result, since it means that the worst case execution time for addition to a dynamic array is O(n). But the reality is not nearly so bleak. Look again at the picture that described the internal array as new elements were added to the collection.

Notice that the costly reallocation of a new array occurred only once during the time that ten elements were added to the collection. If we compute the *average* cost, rather than the *worst case* cost, we will see that the dynamic array is still a relatively efficient container.

To compute the average, count 1 “unit” of cost each time a value is added to the dynamic array without requiring a reallocation. When the reallocation occurs, count one “unit” of cost for each assignment performed as part of the reallocation process, plus one more for placing the new element into the newly enlarged array. How many “units” are spent in the entire process of inserting these ten elements? What is the average “unit” cost for an insertion?

When we can bound an “average” cost of an operation in this fashion, but not bound the worst case execution time, we call it *amortized* *constant* execution time. Amortized constant execution time is often written as O(1)+, the plus sign indicating it is not a guaranteed execution time bound.

Do a similar analysis for 25 consecutive add operations, assuming that the internal array begins with 5 elements (as shown). What is the cost when averaged over this range?

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Add Elmnt** | **Array Size** | **Array Cap.** | **Cost Units** | **Big Oh** |
| 1 | 1 | **5** | 1 | O(c) |
| 2 | 2 | 1 | O(c) |
| 3 | 3 | 1 | O(c) |
| 4 | 4 | 1 | O(c) |
| 5 | 5 | 1 | O(c) |
| 6 | 6 | **10** | 5 + 1 = **6** | **O(n)** |
| 7 | 7 | 1 | O(c) |
| 8 | 8 | 1 | O(c) |
| 9 | 9 | 1 | O(c) |
| 10 | 10 | 1 | O(c) |
| 11 | 11 | **20** | 10 + 1 = **11** | **O(n)** |
| 12 | 12 | 1 | O(c) |
| 13 | 13 | 1 | O(c) |
| 14 | 14 | 1 | O(c) |
| 15 | 15 | 1 | O(c) |
| 16 | 16 | 1 | O(c) |
| 17 | 17 | 1 | O(c) |
| 18 | 18 | 1 | O(c) |
| 19 | 19 | 1 | O(c) |
| 20 | 20 | 1 | O(c) |
| 21 | 21 | **40** | 20 + 1 = **21** | **O(n)** |
| 22 | 22 | 1 | O(c) |
| 23 | 23 | 1 | O(c) |
| 24 | 24 | 1 | O(c) |
| 25 | 25 | 1 | O(c) |

# Worksheet 16: Dynamic Array Stack

**In Preparation**: Read Chapter 6 to learn more about the Stack data type. If you have not done so already, you should complete worksheets 14 and 15 to learn about the basic features of the dynamic array.

Conceptual Stack Operations

void push (TYPE newValue)

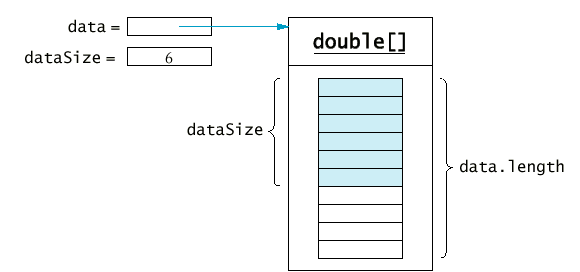
TYPE top ()

void pop ()

Boolean isEmpty ()

In Chapter 6 you read about the Stack data abstraction. A stack maintains values in order based on their time of insertion. When a value is removed from the stack it is the value that has been most recently added to the stack. The abstract definitions of the stack operations are shown at right.

As you learned in Worksheet 14, a positive feature of the array is that it provides random access to values. Elements are accessed using an index, and the time it takes to access any one element is no different from the time it takes to access another. However, a fundamental problem of the simple array is that the size must be specified at the time the array is created. Often the size cannot be easily predicted; for example if the array is being filled with values being read from a file. A solution to this problem is to use a *partially filled array*; an array that is purposely larger than necessary. A separate variable keeps track of the number of elements in the array that have been filled.



The *dynamic array* data type uses this approach. The array of values is encapsulated within a structure boundary, as is the current *size* of the collection. The size represents the number of elements in the array currently in use. The size is different from the *capacity*, which is the actual size of the array. Because the array is referenced by a pointer, an allocation routine must be called to set the initial size and create the initial memory area. A separate destroy routine frees this memory. You wrote these earlier in Worksheet 14.

The function addDynArray(struct DynArr \* da, TYPE v) adds a new value to end of a dynamic array. Recall from Worksheet 14 that this function could potentially increase the size of the internal buffer if there was insufficient space for the new value. This is shown in the following two pictures. In the first picture there is space for the new value, so no reallocation is needed. In the second picture there is no longer enough space, and so a new buffer is created, the elements are copied from the old buffer to the new, and the value is then inserted into the new buffer. You wrote the function dynArrayAdd in worksheet 14. Do you remember the worst-case algorithmic execution time for this function?

# Your task in this worksheet is to write the code for the Stack functions push, pop, top and isEmpty. These functions should use a dynamic array (passed as an argument) for the storage area. Use an assertion to check that the stack has at least one element when the functions top or pop are called. Your job will be greatly simplified by making use of the following functions, which you developed in previous lessons:

**Worksheet 21: Building a Bag using a Dynamic Array**

**In preparation**: Read Chapter 8 to learn more about the Bag data type. If you have not done so already, complete Worksheet 14 to learn about the basic features of the dynamic array.

The stack, queue and deque data abstractions are all characterized by maintaining values in the order they were inserted. In many situations, however, it is the values themselves, and not their time of insertion, that is of primary importance. The simplest data structure that is simply concerned with the values, and not their time of insertion, is the Bag. A conceptual definition of the Bag operations is shown at right. In subsequent lessons we will encounter several different implementation techniques for this abstraction. In this lesson we will explore how to create a bag using a dynamic array as the underlying storage area.

Conceptual Bag interface

void add (TYPE newValue);

Boolean contains (TYPE testValue);

void remove (TYPE testValue);

Recall that the dynamic array structure maintained three data fields. The first was a reference to an array of objects. The number of positions in this array, held in an integer data field, was termed the *capacity* of the container. The third value was an integer that represented the number of elements held in the container. This was termed the *size* of the collection. The size must always be smaller than or equal to the capacity.

Array2

As new elements are inserted, the size is increased. If the size reaches the capacity, then a new internal array is created with twice the capacity, and the values are copied from the old array into the new. In Worksheet14 you wrote a routine \_setCapacityDynArr, to perform this operation.

To add an element to the dynamic array you can simply insert it at the end. This is exactly the same behavior as the function addDynArray you wrote in Worksheet 14.

The contains function is also relatively simple. It simply uses a loop to cycle over the index values, examining each element in turn. If it finds a value that matches the argument, it returns true. If it reaches the end of the collection without finding any value, it returns false. Because we want the container to be generalized, we define equality using a macro definition. This is similar to the symbolic constant trick we used to define the type TYPE. The macro is defined as follows:

# ifndef EQ

# define EQ(a, b) (a == b)

# endif

The ifndef preprocessor instruction allows the user to provide an alternative definition for the EQ function. If none is provided, the primitive equality operator will be used.

The remove function is the most complicated of the Bag abstraction. To simplify this task we will divide it into two distinct steps. The remove function, like the function contains, will loop over each position, examining the elements in the collection. If it finds one that matches the desired value, it will invoke a separate function, removeAt (from Worksheet 14), that removes the value held at a specific location.

void removeDynArr (struct DynArr \* da, TYPE test) {

int i;

for (i = 0; i < da->size; i++) {

if (EQ(test, da->data[i])) { /\* found it \*/

\_dynArrayRemoveAt(da, i);

return;

}

}

}

Notice two things about the remove function. First, if no matching element is found, the loop will terminate and the function return without making any change to the data. Second, once an element has been found, the function returns. This means that if there were two or more occurrences of the value that matched the test element, only the first would be removed.

The removeAt function takes as argument an index position in the array, and removes the element stored at that location. This is complicated by the fact that when the element is removed, all values stored at locations with higher index values must be “moved down”.

VectorRemove

Once the values are moved down, the count must be decremented to indicate the size of the collection is decreased.

Based on these ideas, complete the following skeleton implementation of the bag functions for the dynamic array. You can use any of the functions you have previously written in earlier lessons.