Group 20

Edward Francis
Danielle Goodman
Justin Kruse
Charlotte Murphy
Garret Sweetwood
Woohyuk Yang

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.
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:

```
struct DynArr {
TYPE * data;
        int size;
        int capacity;
        };

        /* initialize a dynamic array structure with given capacity */
void initDynArr (struct DynArr * da, int initialCapacity);

        /* internal method to double the capacity of a dynamic array */
void _setCapacityDynArr (struct DynArr * da);

        /* release dynamically allocated memory for the dynamic array */
void freeDynArr (struct DynArr * da);

        /* return number of elements stored in dynamic array */
int sizeDynArr (struct DynArr * da);

        /* add a value to the end of a dynamically array */
void addDynArr (struct DynArr * da, TYPE e);

        /* remove the value stored at position in the dynamic array */
void removeAtDynArr (struct DynArr * da, int position);

        /* retrieve element at a given position */
TYPE getDynArray (struct DynArr * da, int position);

        /* store element at a given position */
void putDynArr (struct DynArr * da, int position, TYPE value);


# define TYPE int
```

```
struct DynArr {
	TYPE *data;
	int size;
	int capacity;
};


/* Dynamic Array implementation of the Stack Interface */

void pushDynArray (struct DynArr * da, TYPE e) {

	if(da->size == da->capacity)
	{
		_setCapacityDynArr(da);
	}

	addDynArr(da, e);

}

TYPE topDynArray (struct DynArr * da) {

	int size = sizeDynArr(da);
	assert(size != 0);
	int pos = size - 1;
	return getDynArray(da, pos);

}

void popDynArray (struct DynArr * da) {

	int size = sizeDynArr(da);
	assert(size != 0);
	removeAtDynArr(da, size - 1);

}

int isEmptyDynArray (struct DynArr * da) {

	if(!sizeDynArr(da))	//if there is nothing in the array.
		return 1;	//DynArray is empty
	else
		return 0;

}
```

Questions

1. What is the algorithmic execution time for the operations **pop** and **top**?
Ans. It takes constant time, which is O(1).

2. What is the algorithmic execution time for the operation **push,** assuming there is sufficient capacity for the new elements?
Ans. It takes constant time, which is O(1).

3. What is the algorithmic execution time for the internal method **_setCapacityDynArr**?
Ans. It can be described as it takes O(n). Because _setCapacityDynArr needs to make another array whose capacity is twice of the current dynamic array and copy every single elements(n elements) from the old array to the new array.

4. Using as a basis your answer to 3, what is the algorithmic execution time for the operation **push** assuming that a new array must be created.
Ans. It is O(n), also. As long as new dynamic array is reallocated, inserting one more element takes only constant time, which we do not have to count when we consider algorithmic execution time.