Worksheet 16 - 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

Worksheet 16: Dynamic Stack Array

In this worksheet, we will be implementing a Dynamic Stack Array. In order to make this work easier, we will be leveraging the definitions of functions we have previously written for the Dynamic Stack Array. For this worksheet, we have to define the following functions: `pushDynArray`, `topDynArray`, `popDynArray`, and `isEmptyDynArray`. These functions are defined below, with accompanying comments.

```
/*
     The implementation of this function is quite simple.  The
     process of adding a value to the top of a stack is essentially
     the same as adding a value ot the end of an array, which is an
     operation that the existing addDynArr function already
performs.
     As a result, we simply wrap the addDynArr function in the
     pushDynArray function.  We're essentially just creating a new
     name for this function with this wrapping.
*/
void pushDynArray(struct DynArr *da, TYPE e) {
     addDynArr(da, e);
}

/*
     This function returns the value that appears at the top of the
     stack.  This function also leverages functions that are already
     available in the DynArr implementation.  Specificaly, it
     leverages the sizeDynArr function and the getDynArr function.
It
     first makes sure that the size of the stack is greater than
     zero; if this is the case, then it returns the value at the top
     of the stack.
*/
```

```
TYPE topDynArray (struct DynArr *da) {
     assert(sizeDynArr(da) > 0);
     return getDynArr(da, sizeDynArr(da) - 1);
}

/*
     This function removes the value at the top of the stack.  It
     leverages the existing sizeDynArr function to make sure that
the
     size of the array is greater than 0.  If this is true, then the
     size of the stack is decremented.  This effectively removes the
     value at the top of the stack.
*/
void popDynArray (struct DynArr * da) {
     assert(sizeDynArr(da) > 0);
     da->size--;
}

/*
     This function returns an integer value representing whether or
     not the stack is empty.  It does so by leveraging the existing
     sizeDynArr function.  It calls this function on the dynamic
     stack and compares the returned value to 0.  The result of this
     comparison is returned.
*/
int isEmptyDynArray (struct DynArr *da) {
     return sizeDynArr(da) == 0;
}
```

With these functions defined, our stack behavior is no complete. The worksheet also asked us to address the following questions.

1. What is the algorithmic execution time for the operations pop and top?

The pop and top operations have O(1), or constant execution time.

2. What is the algorithmic execution time for the operation push, assuming there is sufficient capacity for the new elements?

Assuming that there is sufficient capacity for the new elements, the push operation has O(1), or constant execution time.

3. What is the algorithmic execution time for the internal method _setCapacityDynArr?

The internal method _setCapacityDynArr has O(n), or linear execution time. This method has to copy all of the elements in the existing array to a 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.

Assuming that a new array must be created, the push operation has O(n), or linear execution time. The push method will call the _setCapacityDynArr message to resize the array; this function has O(n) complexity. Once this function terminates, the process of adding another element is O(1). The O(n) function will dominate the O(1) function, so push has O(n) complexity when a new array must be created.

Group Minutes

Our Minutes are available on Piazza at the following URL:

https://piazza.com/class/ib2kus4hsie528?cid=72