Worksheet 21 - 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 21: Building a Bag Using a Dynamic Array

In this assignment, we implement two different functions which control the behavior of a bag data structure. More specifically, we implement the `removeDynArr` and `containsDynArr` functions. These are described in detail below, with comments where necessary.

```
/*
      This function removes from a dynamic array the first occurrence
      of a value specified by the user.  It does so by performing a
      linear search. If the value is found, we leverage the existing
      removeAtDynArr function to remove the value at the particular
      index and shift elements to their new index.  If the value is
      not found, the dynamic array is not modified.
*/
void removeDynArr(struct DynArr *da, TYPE e) {
      int index;

      for (index = 0; index < sizeDynArr(da); index++) {
            if (EQ(getDynArr(da, index), e)) {
                  removeAtDynArr(da, index);
                  return
            }
      }
}

/*
      This function determines whether or not a value provided by
      the user appears in the dynamic array.  It performs a linear
      search; if the value is found, then the function returns 1
      which represents that the value was found; otherwise, the
      function returns 0 indicating that the value was not found.
*/
```

```
int containsDynArr (struct DynArr *da, TYPE e) {
	int index;

	for (index = 0; index < sizeDynArr(da); index++) {
		if (EQ(getDynArr(da, index), e)) {
			return 1;
		}
	}

	return 0;
}
```

In addition to the function definitions above, we also have to answer four questions for this worksheet. These questions are listed below with our answers.

1. What should the removeAt method do if the index given as argument is not in range?

We have included the assert.h header file in our implementation file. We use this in our function definition for removeAt to ensure that the given index is greater than or equal to zero and less than the size of the array. This ensures that the given index is in range. If the index is not in range, then the program terminates with an assertion error.

2. What is the algorithmic complexity of the method removeAtDynArr?

We have included our code for the removeAtDynArr function below:

```
void removeAtDynArr(struct DynArr *v, int index) {
	assert((index >= 0) && (index < sizeDynArr(v)));

	for (; index < (sizeDynArr(v) - 1); index++) {
		v->data[index] = v->data[index + 1];
	}

	v->size--;
}
```

This function has O(n) complexity.

3. Given your answer to the previous question, what is the worst-case complexity of the method remove?

The worst case is that the first element will be removed from the array. This means that every element after the first element will need to be shifted to the left by one place. This is O(n) complexity.

4. What are the algorithmic complexities of the operations add and contains?

The add operation has O(1) complexity if the element is added at the end of the array; it has O(n) complexity the element is added elsewhere..

The contains operation has O(n) complexity.

Group Minutes

Our Minutes are available on Piazza at the following URL:

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