Worksheet 26 - 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

Collaborators

Marc, Kelby, James, Tom, Sean, Joseph

Worksheet 26: Ordered Bag Using a Sorted Array

In this assignment, we implement several different functions which control the behavior of an ordered bag using a sorted dynamic array. More specifically, we implement the `dynArrAddAt`, `orderedArrayContains`, and `orderedArrayRemove` functions. The `dynArrAddAt` function is by far the most complex of these functions. In any case, these functions are each defined below and are accompanied by comments where needed.

```
/*
     This function performs adds a value to an array at a particular index.  If the value is
     inserted at the very beginning of the array or in the middle of the array, it
     adds the array at the indicated index and shifts the values to the right of that index
     one index to the right.  This function begins by checking that the index is in the
     correct range--between 0 and the size of the array inclusive.  The function then allocates
     a new chunk of memory for the current or resized capacity of the array.  The function
     then copies the values before the given index into the new array, inserts the new value at
     the indicated index, and fills in the remaining values. The function frees the memory for
```

```
	the existing data, and copies over the new data to the dynamic array. Finally, it updates
	the values for size and capacity.
*/
void dynArrAddAt(struct DynArr *v, int index, TYPE newElement) {
	assert(index >= 0);
	assert(index <= sizeDynArr(v));

	TYPE *newData;
	int currentIndex,
		newCap;

	if (sizeDynArr(v) >= v->capacity) {
		newCap = v->capacity * 2;
	} else {
		newCap = v->capacity;
	}

	newData = malloc(sizeof(TYPE) * newCap);

	for (currentIndex = 0; currentIndex < index; currentIndex++) {
		newData[currentIndex] = v->data[currentIndex];
	}

	newData[index] = newElement;

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

	free(v->data);
	v->data = newData;
	v->capacity = newCap;
	v->size++;
}

/*
	This function returns an integer value representing whether or not a particular target
	value appears in the dynamic array.  It utilizes the binary search algorithm, which
	terminates in O(log n) time; this is significantly faster than the linear search
	algorithm, which terminates in O(n) time.  The binary search algorithm returns the
	index where the target value occurs or where the target value should be inserted to
	keep it sorted.  The function compares the value at the index to the targe value; if
	it’s a match, the function returns 1 to indicate that the value was found.  Otherwise,
	the value returns 0 to indicate that the target value was not found.
*/
int orderedArrayContains(struct DynArr *v, TYPE target) {
	int index = _binarySearch(v->data, sizeDynArr(v), target);
	return v->data[index] == target;
}

/*
	This function removes an integer value from the dynamic array if it appears. The function
	utilizes the binary search algorithm to locate the index where the value might occur. If
	the target value appears at that index, then the value is removed and the function
	terminates. Otherwise, the dynamic array is left unchanged.
*/
void orderedArrayRemove(struct DynArr *v, TYPE target) {
	int index = _binarySearch(v->data, sizeDynArr(v), target);
	if (v->data[index] == target) {
		removeAtDynArr(v, index);
		return;
```

```
        }
}
```

In addition to defining the functions above, we also had to provide the complexities for the various implementations of the dynamic array bag functions. A table listing the execution times is below.

	Dynamic Array Bag	Linked List Bag	Ordered Array Bag
Add	O(1)+	O(1)	O(n)
Contains	O(n)	O(n)	O(log n)
Remove	O(n)	O(n)	O(n)

Finally, we had to respond to the following questions.

Short Answers
1. What is the algorithmic complexity of the binary search algorithm?

The algorithmic complexity of the binary search algorithm is O(log n).

2. Using your answer to the previous question, what is the algorithmic complexity of the method contains for an OrderedArrayBag?

Since the contains method uses the binary search function to locate the potential index for the target value, the contains function also executes in O(log n) time.

3. What is the algorithmic complexity of the method addAt?

This function has to loop through every value in an array, so it has O(n) time.

4. Using your answer to the previous question, what is the algorithmic complexity of the method add for an OrderedArrayBag?

Although this function does use the binary search algorithm, which has O(log n) complexity, it also uses the addAt function described above, which has O(n) complexity. O(n) dominates O(log n), so the function has O(n) complexity.

5. What is the complexity of the method removeAt? of the method remove?

The remove function uses the binary search algorithm to locate the index where a value should be removed; this has O(log n) complexity. The remove function potentially creates a hole in the array, which must be filled--an O(n) operation. Since O(n) dominates O(log n), this function has O(n) complexity.

Piazza Discussion

https://piazza.com/class/ib2kus4hsie528?cid=173