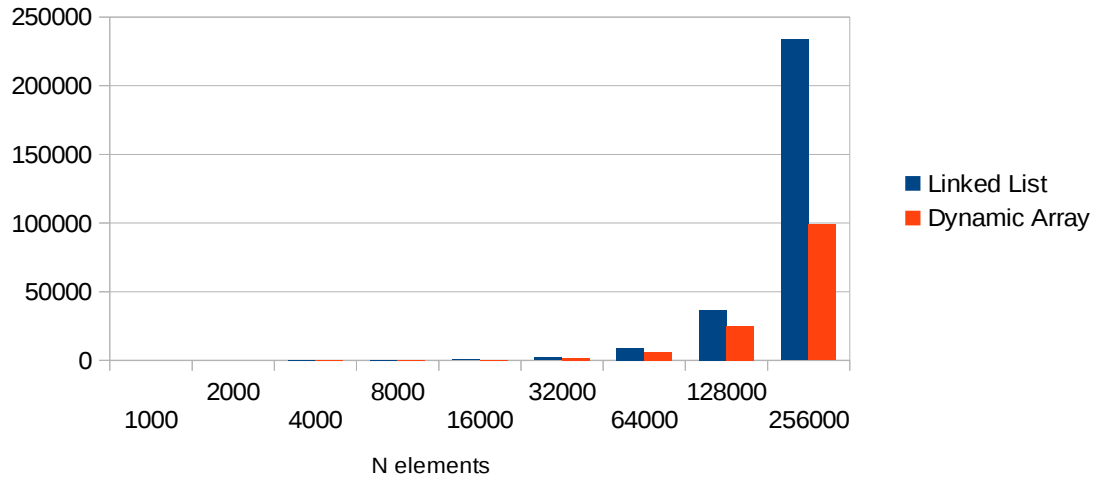
Brandon Swanson
Assignment 3 CS261 10/23/14

Time/Memory Analysis

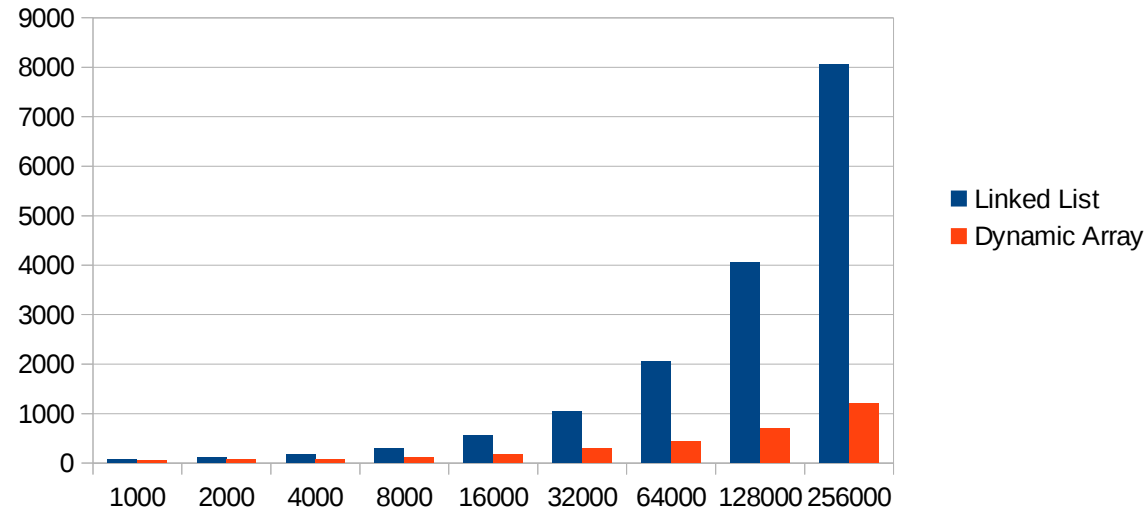
number of elements	Linked List	Dynamic Array
1000	0	0
2000	0	0
4000	30	20
8000	130	90
16000	560	390
32000	2220	1540
64000	8940	6260
128000	36440	24850
256000	233570	98940

Time in MilliSeconds



number of elements	Linked List	Dynamic Array
1000	80	52
2000	112	68
4000	176	84
8000	300	116
16000	552	180
32000	1052	304
64000	2052	444
128000	4052	696
256000	8052	1200

Memory Usage (KB)



Which of the implementations uses more memory? Explain why.
The linked list implementation uses more memory. Even though the dynamic array must reallocate the memory to a larger chuck when it capacity is exceeded there is still much less overhead. Because in the array all the data is stored as one contiguous block of memory and one struct holding the pointer to first element and the size and capacity information (3 ints)
By contrast each link in the link list requires two pointers (32 or 64 bit integer). This would account for a disparity of roughly 3 to 1 but there is more to consider. I researched what actually happens when calling malloc() and free(), for instance when you call free() how does the system know how much memory to free from the supplied pointer. As it turns out each allocated block of memory has an "accounting" section located before the returned pointer in memory containing the size of the allocated block, error checking etc. also there is padding to make the allocated block of memory equal to a power of 2 set by the system. So when allocating space for a link more than just 3 time the the bits needed for INT are allocated.
The dynamic array has only one such accounting and padding section in memory, and each value need not maintain two additional data fields. For this reason we see a dramatic difference in memory usage.

Which of the implementations is the fastest? Explain why.

Linked List

```
int containsList (struct linkedList *lst, TYPE e)
{

    //NULL checking
    assert(lst);
    assert(lst->lastLink);
    assert(lst->firstLink);
    assert(!isEmptyList(lst));

    for(struct DLink *l=lst->firstLink->next;
            l!=lst->lastLink;
            l=l->next){

        //null checking
        assert(l);

        if(l->value == e) return(1); //item found
    }

    return(0); //item not found

}
```

Dynamic Array

```
int containsDynArr(DynArr *v, TYPE val)
{
    assert(v!=NULL);
    assert(v->size>0);

    //SEARCH FOR ELEMENT
    for(int i=0;i<v->size;i++){

                                //ELEMENT FOUND
        if(v->data[i]==val) return 1;

    }
    return 0; //ELEMENT NOT FOUND

}
```

The dynamic array is faster. I put the code for each function side by side because I was a bit surprised at the result. Even though random access of the dynamic array is astronomically faster because neither of these structures are sorted then neither of them involve random access. It is a linear traversal through each container so it must come down to the number of operations required for each location increment and then value access. Comparing these two functions, within the loop each one performs similar access and compare, but the linked list must assert that it has not received a null pointer. Additionally the method for incrementing the position might account for the speed difference, the dynamic array uses arithmetic on the index variable, whereas the linked list is accessing disparate chunks of memory to find the next link and compare vs the sentinal

Would you expect anything to change if the loop performed remove() instead of contains()? If so, what?

Yes, the collection would get continuously smaller after each remove causing successive calls to remove() to be performed faster. In this particular instance where the additions are performed sequentially and uniquely (1,2,3,..n) and the Contains() test loop performs the same order of values.

For the array this would mean that the first element in the array would always be the one being searched for. And for the list this would depend on how the add() operation was implemented (added to front or added to the back)

for the linked list this could mean a very optimum constant time operation. But for the array each operation would require shifting each element in the array.