Dynamic Table

A dynamic table is a data structure with the ability to expand or collapse itself depending on the number of elements it contains. It supports random access (a resizeable array), and does not need to have a fixed upper bound (bounded only by hardware or system) at the time of declaration. It also saves space as it can decrease the size of the table if most of the table is unoccupied. We will see that the amortized cost of insertion and deletion into this table remains constant, even though increasing and decreasing the size of the table is an operation that takes O(n) in terms of time.
The downside is that as the size of the array grows, it may be difficult to find contiguous memory locations.

This is carried out by a set of operations , namely :

- **Insert**
- **Delete**
- **Increase** (size of table) by a factor
- **Decrease** (size of table) by a factor

Insert operation:

```
if(table->cur_size >=  table->max_size){
        increase(table);
}
table->t[table->cur_size] = ele;
table->cur_size++;
```

Increase operation:

```
table->max_size = ceil(table->max_size * table->increase_factor);
table->t = (char*) realloc(table->t, table->max_size);
```

This operation inserts an element to the end of the array/table - an operation which takes constant time. If the table is full, it calls the increase function to increase the size by a factor and reallocs a block of memory, copying over all the elements currently in the table to the new block. This is an O(n) time operation.

Delete operation:

```
if(table->cur_size > 0){
                table->cur_size-- ;
                if(table->cur_size < ((table->max_size)*table->decrease_factor)){
                        decrease(table);
                }
```

```
        }
```

Decrease operation:

```
    table->max_size = table->max_size * table->decrease_factor;
    if(table->max_size <= 0){
        table->max_size = 1;
    }
    table->t = (char*) realloc(table->t, table->max_size);
```

The delete operation decrements the size and checks how much of the table is occupied. This is done in constant time. If this is less than a chosen fraction of the table (say, ½ is unoccupied), it reallocates the current array block to a smaller array block, which is an O(n) operation.
These O(n) operations do not occur too often, and hence the amortized cost of each **Insert** and **Delete** operation is still constant, as shown below.

Note: It may sound faster to reduce the size pointer and free the extra memory in the **Decrease** operation, however C does not allow us to free memory from an arbitrary location in an allocated block of memory.

A generate_op() function is used to randomly generate insert and delete operations in the following ratios -

- 1:1
- 3:2
- 4:2

Eg. **in the 3:2 ratio**

```
    for(i=0; i<n; i++){
        select_type =  rand() % 5;
        if((select_type == 0) || (select_type == 1) || (select_type == 2) ){
            arr[s] = insert;
        }
        else {
            arr[s] = delete_fn;
        }
        s++;
    }
```

The amount (factor) the table grows or shrinks by is integral to increasing its space and time efficiency. Based on the values of insertion below (refer to Page 4), we can see that, even though the maximum times of insertion for these factors (increasing and decreasing the table), the average times stayed low and are almost the same across factors (constant).
The values of deletion exhibit a similar trend.

If we look at the times as the number of operations increase (refer to Page 5), (from 20 to 100 to 1000 and so on till a million) we can see that even though the average times may be higher for fewer operations, they come down to a constant value after about 10,000 operations.

Pointers and Dynamic Tables :

1. Pointers to indices in dynamic tables will need to be updated after every realloc otherwise they will be pointing to unallocated memory or memory thats being used for something else.
A solution to this would be to make the pointer point to an offset from the starting pointer, which points to the start of the reallocated block.

2. If the reallocation went wrong or the block was not allocated the properly, the realloc function would return a null pointer and set errno to a value depending on the error. This could tell us if the block allocated was invalid/ not allocated.

__Times for Insertion__

Maximum Time for Insertion:

Imp : The times in red are the maximum times, while the ones in purple are the minimum times.

(1:1)

No. of operations: 1000000

	2.000000	3.000000	1.750000	1.500000	1.250000
0.250000	0.074251	***0.012736***	0.051565	***0.218070***	**0.019863**
0.500000	0.052064	0.068045	0.071943	0.089522	0.070186
0.750000	0.058406	0.149110	0.068767	**0.214518**	0.072691

(3:2)

No. of operations: 1000000

	2.000000	3.000000	1.750000	1.500000	1.250000
0.250000	0.056672	**0.474375**	0.053479	0.473403	0.055705
0.500000	0.086513	**0.474042**	***0.633721***	0.052770	0.147841
0.750000	**0.037560**	***0.028803***	0.038092	0.473668	0.052379

(4:2)

No. of operations: 1000000

	2.000000	3.000000	1.750000	1.500000	1.250000
0.250000	0.059669	**0.035155**	0.049402	0.043142	0.059104
0.500000	0.058343	**0.082389**	0.052616	0.045728	0.071185
0.750000	**0.072391**	**0.027476**	0.040199	0.044398	0.052842

Average time for Insertion:

(all ratios)

No. of operations: 1000000

	2.000000	3.000000	1.750000	1.500000	1.250000
0.250000	0.000025	**0.000025**	0.000024	**0.000026**	**0.000025**
0.500000	0.000026	**0.000025**	**0.000025**	0.000025	0.000026
0.750000	**0.000025**	**0.000025**	0.000025	**0.000027**	0.000026

<u>Times as number of operations increase:</u>
<u>(4:2)</u>
Average times for all operations:

No. of operations: 20

	2.000000	3.000000	1.750000	1.500000	1.250000
0.250000	0.004500	**0.001481**	0.000626	0.000034	**0.001118**
0.500000	0.001236	0.001579	**0.000663**	0.000039	0.000780
0.750000	0.001393	0.000057	0.000376	0.000056	0.000271

No. of operations: 100

	2.000000	3.000000	1.750000	1.500000	1.250000
0.250000	0.000230	**0.000481**	0.000211	0.000101	**0.000036**
0.500000	0.000117	0.000349	**0.000085**	0.000084	0.000077
0.750000	0.000051	0.000144	0.000115	0.000184	0.000038

No. of operations: 1000

	2.000000	3.000000	1.750000	1.500000	1.250000
0.250000	0.000043	**0.000027**	0.000027	0.000039	**0.000048**
0.500000	0.000029	0.000041	**0.000027**	0.000027	0.000027
0.750000	0.000040	0.000037	0.000039	0.000028	0.000099

No. of operations: 10000

	2.000000	3.000000	1.750000	1.500000	1.250000
0.250000	0.000029	**0.000028**	0.000026	0.000030	**0.000027**
0.500000	0.000036	0.000029	**0.000025**	0.000027	0.000036
0.750000	0.000030	0.000027	0.000027	0.000030	0.000028

No. of operations: 100000

	2.000000	3.000000	1.750000	1.500000	1.250000
0.250000	0.000026	**0.000027**	0.000028	0.000027	**0.000026**
0.500000	0.000027	0.000027	**0.000028**	0.000027	0.000028
0.750000	0.000026	0.000026	0.000027	0.000026	0.000028

No. of operations: 1000000

	2.000000	3.000000	1.750000	1.500000	1.250000
0.250000	0.000026	**0.000026**	0.000028	0.000039	**0.000026**
0.500000	0.000029	0.000027	**0.000026**	0.000026	0.000026
0.750000	0.000027	0.000028	0.000027	0.000027	0.000026