Discussion 3

Table of Contents

- Homework
- Dynamic Array

Homework

- Deadline
  - OCT 22, 2023 11:59 PM
- Grade
  - 1. test_submitted_test_cases
  - 2. test_build_for_grading
    - test_linked_list (60/60)
    - test_stack (15/15)
    - test_queue (15/15)
    - test_memory_safety (10/10)

Dynamic Array

Recap of Fixed-sized Array & Linked List

Operation	**Fixed-sized Array (e.g. `std::array`)**	**Linked List (e.g. `std::list`)**
Random Access (`[]`)		
Append		
Insert		
Remove		
Memory		

Recap of Fixed-sized Array & Linked List

Operation	**Fixed-sized Array (e.g. `std::array`)**	**Linked List (e.g. `std::list`)**
Random Access (`[]`)	$O(1)$	$O(n)$
Append	Not supported	$O(1)$
Insert	Not supported	$O(1)$
Remove	Not supported	$O(1)$
Memory	Continuous	Not Continuous

Recap of Fixed-sized Array & Linked List

	Array	**LinkedList**
Memory	Contiguous location & Less memory needed	Non-contiguous location & more memory needed (value and pointer)
Size	Fixed size	Dynamic size
Element access	$O(1)$ constant time	$O(n)$ linear time
Insert/Deletion element	Slow	Fast

What's a dynamic array?

A dynamic array can automatically grow or shrink in size as needed.

Why dynamic array?

- We may not know the size of the array at the time of array declaration
- Elements can be added or removed at anytime
- Then, why not linked list?
    - We need efficient random access

Features of dynamic array

1. **Dynamic Sizing**: Dynamic arrays can grow or shrink in size during runtime. When you need to add more elements than the array can currently hold, it can allocate additional memory, copying the existing elements into the new space. This allows for flexible memory management.
2. **Random Access**: Like traditional arrays, dynamic arrays provide constant-time $O(1)$ access to elements by index, which makes them efficient for random access operations.
3. **Efficient Appends**: Appending an element to the end of a dynamic array typically has an amortized constant-time $O(1)$ complexity, as long as you have sufficient capacity. This makes dynamic arrays suitable for scenarios where you frequently add elements to the end.

Limitations

1. **Memory Overhead**:
    1. Reserve more memory than their current contents require to accommodate potential future growth.
    2. Memory wastage if the array size significantly exceeds the actual data size.
2. **Resizing Overhead**: When a dynamic array needs to grow beyond its current capacity, it may need to allocate a new block of memory, copy the existing elements, and deallocate the old memory. This operation can be time-consuming and may lead to a sudden spike in time complexity.
3. **Limited Insertion/Deletion Efficiency of elements in the middle** : While dynamic arrays offer fast random access and efficient appending to the end, inserting or deleting elements in the middle can be less efficient. Such operations may require shifting elements, leading to linear time $O(n)$ complexity.

Array vs List

What's the difference between array and list?

⚠️

There is no universal definition of array and list.

Different languages have different definitions & implementations.

Array & List in C++

In C++,

- `std::list` is a double-linked list
- `std::vector` is a dynamic array
- `std::array` is a fixed-size array
- C-style array is also a fixed-size array

In general context and some other languages

- Array often refers fixed-size array
- List usually refers to dynamic array

How to implement a dynamic array?

Naive Approach

When we append to the end of the array, we create a new array with **size + 1**, copy all the elements from the old array to the new array, and then append the new element to the new array.

- Con: $O(n)$ time complexity for append

Improved Naive Approach

When we append to the end of the array, we check if the array is full. If it is full, we create a new array with **size + 4**, copy all the elements from the old array to the new array, and then append the new element to the new array.

- Pro: less memory allocation & copying for append than the naive approach
- Con: still $O(n)$ time complexity for append

Improved Naive Approach

Operation	Capacity	Size	Memory Alloc/Dealloc	Elements Copied
Initialize	4	0	1	0
Append	4	1	0	1
Append	4	2	0	1
Append	4	3	0	1
Append	4	4	0	1
Append	8	5	1	4 + 1
Append	8	6	0	1
Append	8	7	0	1
Append	8	8	0	1
Append	12	9	1	8 + 1
Append	12	10	0	1
Append	...	...	...	...

Improved Naive Approach (not a formal proof)

Total time to append n elements into the dynamic array that is empty:

$$
\begin{aligned}
T_n(n) &= n + \sum_{k=1}^{\frac{n}{4}-1} 4k \\
&= n + 4\sum_{k=1}^{\frac{n}{4}-1} k \\
&= n + 4 \cdot \frac{\frac{n}{4} \cdot (\frac{n}{4} - 1)}{2} \\
&= n + \frac{n^2}{8} - \frac{n}{2} \\
&= O(n^2)
\end{aligned}
$$

Amortized time complexity for append:

$$
T_1(n) = \frac{1}{n} O(n^2) = O(n)
$$

Improved Naive Approach (not a formal proof)

For any fixed $m > 0$, total time to append n elements into the dynamic array that is empty:

$$\begin{aligned} T_n(n) &= n + \sum_{k=1}^{\frac{n}{m}-1} mk \\ &= n + m \sum_{k=1}^{\frac{n}{m}-1} k \\ &= n + m \cdot \frac{\frac{n}{m} \cdot (\frac{n}{m} - 1)}{2} \\ &= n + \frac{n^2}{2m} - \frac{n}{2} = O(n^2) \end{aligned}$$

Amortized time complexity for append:

$$T_1(n) = \frac{1}{n} O(n^2) = O(n)$$

Actual Approach

When we append to the end of the array, we check if the array is full. If it is full, we create a new array with **size * 2**, copy all the elements from the old array to the new array, and then append the new element to the new array.

- Pro: even less memory allocation & copying for append
- Con: average memory utilization is lower

Actual Approach

Operation	Capacity	Size	Memory Alloc/Dealloc	Elements Copied
Initialize	1	0	1	0
Append	1	1	0	1
Append	2	2	1	1 + 1
Append	4	3	1	2 + 1
Append	4	4	0	1
Append	8	5	1	4 + 1
Append	8	6	0	1
Append	8	7	0	1
Append	8	8	0	1
Append	16	9	1	8 + 1
Append	16	10	0	1
Append	...	...	...	...

Time Complexity for Actual Approach (not a formal proof)

Time for appending n elements into the dynamic array that is empty:

$$
\begin{aligned}
T_n(n) &= n + \sum_{k=1}^{\log_2 n} 2^{k-1} \\
&= n + \frac{1}{2} \sum_{k=1}^{\log_2 n} 2^k \\
&= n + \frac{1}{2} \cdot \frac{2(2^{\log_2 n} - 1)}{2 - 1} \\
&= n + (2^{\log_2 n} - 1) \\
&= n + (n - 1) = O(n)
\end{aligned}
$$

Amortized time complexity for append:

$$
T_1(n) = \frac{1}{n} O(n) = O(1)
$$

Time Complexity Comparison

Operation	Dynamic Array (e.g. `std::vector`)	Fixed-sized Array (e.g. `std::array`)	Doubly-linked List (e.g. `std::list`)
Random Access (`[]`)	$O(1)$	$O(1)$	$O(n)$
Append	$O(1)$ (amortized)	Not supported	$O(1)$
Insert	$O(n)$	Not supported	$O(1)$
Remove	$O(n)$	Not supported	$O(1)$
Memory	Continuous	Continuous	Not Continuous

Actual Performance

In practice, Dynamic Array is usually faster than Linked List in all operations when the number of elements is less than 20,000.

Why?

Actual Performance

- big O only tells us the asymptotic performance
- continuous memory locations for elements makes dynamic array practically faster than linked list