CS 008
Lecture notes
3/12/24

1 Outline

- Announcements
- Abstract data types vs Data Structures
- Style guide

2 Announcements

A few things before the lecture begins:

- Fill out Assignment 1 progress update 2 by midnight tonight.
- New lab assignment today, due midnight Thursday.
- There'll be another assignment on Thursday.

For students, please stay on track with the assignments. There will be extensions with assignments if people are behind on the assignments.

3 Abstract data types vs Data Structures

An abstract data type is not exactly a data structure. When an abstract data type is implemented, that implementation of an abstract data type is a data structure.

Example:

Abstract datatype examples:

- Bag
- Sequence

Data structure examples:

- Bag implemented using static-length array
- Bag implemented using dynamic arrays
- Sequence implemented using static-length array
- Sequence implemented using dynamic arrays

3.1 Dynamic arrays

Question:

Why are dynamic arrays useful?

Answer: With dynamic arrays, you can modify the size of the array. Unlike normal arrays that need to be declared with a specific size. However, one big use for dynamic arrays is it can be implemented with unlimited length.

To modify the size of the dynamic array, you can create a temporary pointer and create a new dynamic space, copying over the old array onto this new space and then deleting the old space.

Question:

Why is the size of the dynamic array doubled when expanded?

Answer: Unless the new element in the array is going to be the last one added, creating more space in the array is useful so that the array does not need to be deleted and created again with a new size.

In addition to this doubling the size of the array with a resize is more computationally efficient. While it could be the least memory efficient, it's more computational efficient. It is also very similar to C++'s implementation of the vector class.

4 Style guide

Examples can be found on Canvas.
Try to develop a style that is helpful for others (and including yourself) that would make the code easier to read.

Note:

Do not waste time and create whole test suites for assignments

4.1 Preconditions

Preconditions are stated for a function and are conditions that must be true at the beginning of a function.

4.2 Postconditions

Similarly to a precondition, the postconditions are the conditions that must hold at the end of the function. It describes the result of the function.

5 Big-O Notation and Algorithmic Complexity

Every function that is run will execute a finite number of operation. Even large functions will have a finite amount of operations.

Examples:

Some examples of operations:

- a=b (Assignment operation)
- data (Indexing operation)
- 42; (Multiplication operation)
- used++; (Incrementation operation)
- if (b==c) ; (Equality comparison operator)

The number of operations a function executes is known as runtime. The actual number of operations a function executes can depend on the vvalues of its parameters. We often talk about **best-case**, **average-case**, **worst-case** runtimes. In terms of Big-O notation, we think of big numbers and work with the worst-case runtimes.

Big-O notation is a description of a polynomial function. Any polynomial function can be described through Big-O notation.

Example:

Standard Big-O analysis:

- Arrange the terms of the function like a standard polynomial
- Drop all except the highest power term
- Drop the coefficient from the highest power term
- This becomes the 'order' of the function and can be described as $f(x) = O(x^3)$

To determine the "order" (Big-O class) of a function in our program:

- Determine the number of operations the function executes in general
- Look at the worst-case number of operations
- Express this as a function
- Then perform the standard Big-O analysis from the previous example
  - Big-O notation is customarily expressed with generic variable names like n.

5.1 Operations and constants

Figure:

```
#include <cstddef>
#include <iostream>

int main()
{
    size_t dynamic_array_size = 5;
    size_t data_entries = 0;
    int* dynamic_array_ptr = new int[dynamic_array_size];
    int* temporary_array_ptr;
```

```
    // how many will elements will we try to stuff into
↪   the array?
    //int last_loop_iteration = dynamic_array_size-1;
    //int last_loop_iteration = dynamic_array_size;
    int last_loop_iteration = 2*dynamic_array_size;

    // let's put some elements on the array
    for (int index = 0; index <= last_loop_iteration;
↪   index++)
    {
        std::cout << "Starting for loop iteration" <<
↪   std::endl;
        std::cout << "Size of array: " <<
↪   dynamic_array_size << std::endl;
        std::cout << "elements in array: " <<
↪   data_entries << std::endl;

        // we might need to add more capacity to the
↪   array
        if (data_entries >= dynamic_array_size)
        {
            std::cout << "Looks like we need to add more
↪   capacity to the array." << std::endl;
            dynamic_array_size *= 2;
            temporary_array_ptr = new
↪   int[dynamic_array_size];
            std::copy(dynamic_array_ptr,
↪   dynamic_array_ptr + data_entries,
↪   temporary_array_ptr);
            delete dynamic_array_ptr;
            dynamic_array_ptr = temporary_array_ptr;
            std::cout << "New array capacity: " <<
↪   dynamic_array_size << std::endl;
        }

        // adding element to array
        dynamic_array_ptr[index] = index;
        data_entries++;
        std::cout << "Ending for loop iteration" <<
↪   std::endl;
        std::cout << " " << std::endl;
    }
```

```
    return 0;
}
```

The Figure above represents a chunk of code that will resize a dynamic array if it is full. With the knowledge we have with polynomial functions and some information about Big-O notation, how would we approach the analysis of this?

Examples:

The operations within the for loop.
Case 1: Adding elements to a non-full array

- `dynamic_array_ptr[index] = index;`: 2 operations
- `data_entries++;`: 1 operation
- **Total**: 3 operations

Case 2: Adding elements to a full array

- `temporary_array_ptr = new int[dynamic_array_size];`: 1 operation
- `std::copy(dynamic_array_ptr, dynamic_array_ptr + data_entries, temporary_array_ptr);`: N-operations
- `delete dynamic_array_ptr;`: 1 operation
- `dynamic_array_ptr = temporary_array_ptr;`: 1 operation

Note:

In C++, unused variables should be deleted with the delete keyword. In addition, when working with constants in Big-O notation, it would run in $O(1)$ even if the constant is extremely larger since in coefficient terms, cx^0 (where c represents the constant value) would drop and turn it into 1.

Example: $O(3) \rightarrow O(1)$

To add on to Big-O notation with constants, if we deal with a function $f(n) = n + 3$, the constant is dropped because n is the highest order and

therefere anything else would be dropped resulting in a time complexity of $O(n)$.