Name: Chey Naryvety

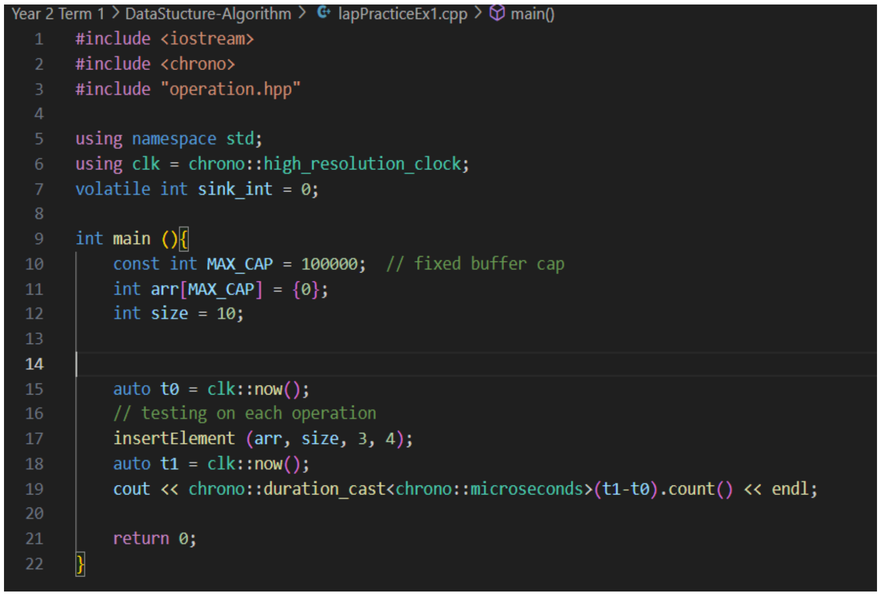
Group: 4

Week 1 – Complexity

Challenge 1 - Insert 1 element in array

1A. Insert into empty array

Test	Runtime (microseconds)
1	0
2	0
3	0
4	0
5	0
AVG	0


Year 2 Term 1 > DataStucture-Algorithm > lapPracticeEx1.cpp > main()

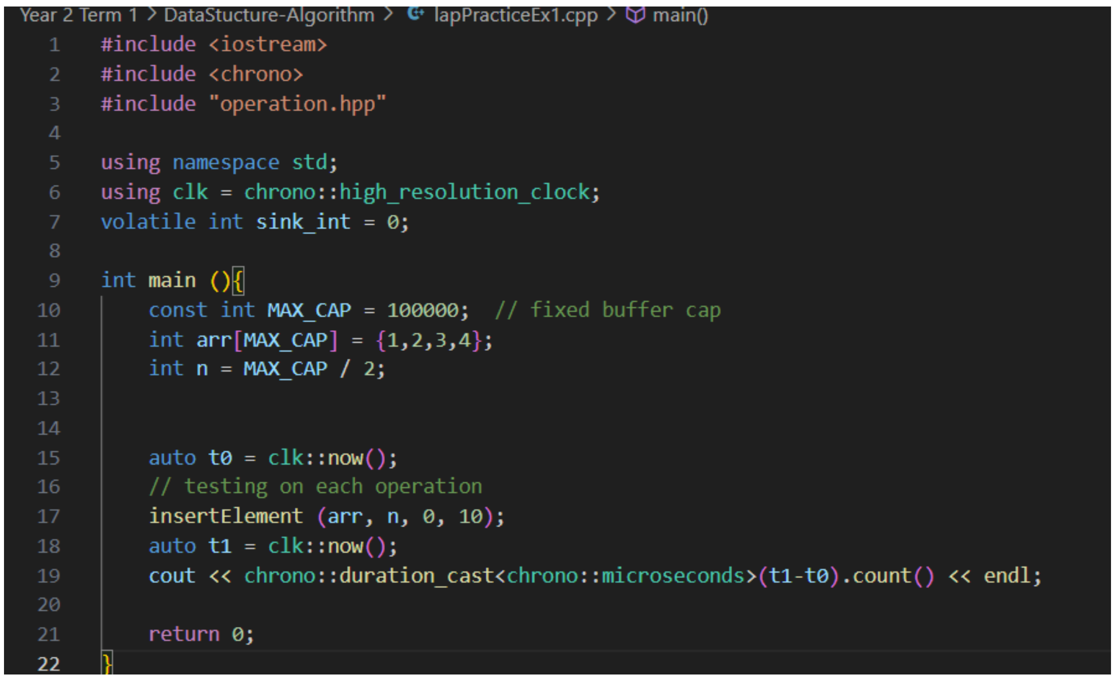
```cpp
#include <iostream>
#include <chrono>
#include "operation.hpp"

using namespace std;
using clk = chrono::high_resolution_clock;
volatile int sink_int = 0;

int main (){
    const int MAX_CAP = 100000;  // fixed buffer cap
    int arr[MAX_CAP] = {0};
    int size = 10;


    auto t0 = clk::now();
    // testing on each operation
    insertElement (arr, size, 3, 4);
    auto t1 = clk::now();
    cout << chrono::duration_cast<chrono::microseconds>(t1-t0).count() << endl;

    return 0;
}
```

Explanation: Inserting into an empty array means there is no need to shift any elements, making the operation runtime is O(1).

1B. Insert when not full, no index given

- Inserting at the beginning (index = 0)

Test	Runtime(microseconds)
1	63
2	57
3	51
4	69
5	67
AVG	61

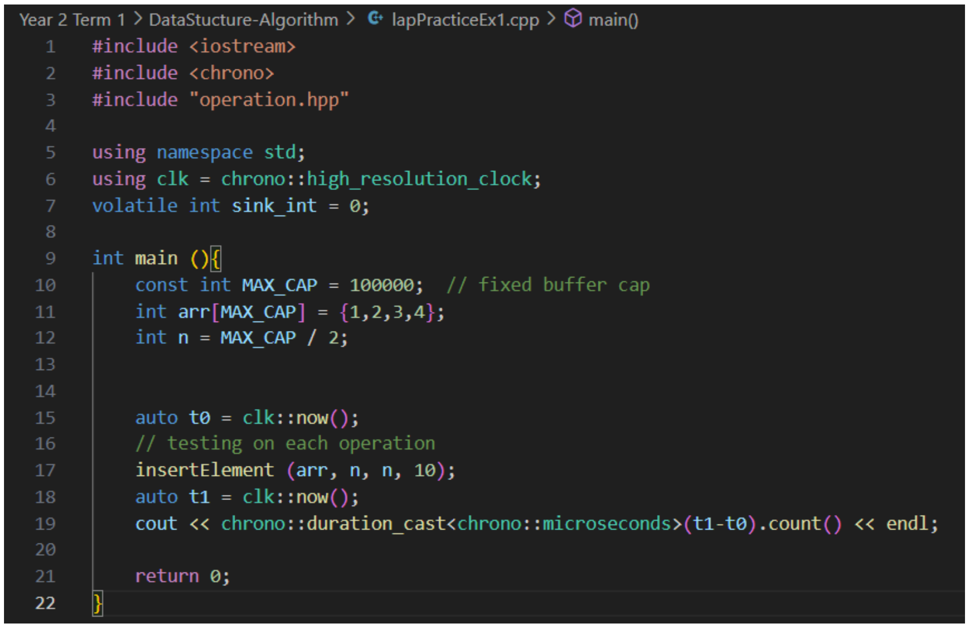

Year 2 Term 1 > DataStucture-Algorithm > lapPracticeEx1.cpp > main()

```cpp
#include <iostream>
#include <chrono>
#include "operation.hpp"

using namespace std;
using clk = chrono::high_resolution_clock;
volatile int sink_int = 0;

int main (){
    const int MAX_CAP = 100000;  // fixed buffer cap
    int arr[MAX_CAP] = {1,2,3,4};
    int n = MAX_CAP / 2;


    auto t0 = clk::now();
    // testing on each operation
    insertElement (arr, n, 0, 10);
    auto t1 = clk::now();
    cout << chrono::duration_cast<chrono::microseconds>(t1-t0).count() << endl;

    return 0;
}
```

Explanation: Inserting at the beginning of an array requires the elements to shift to make space for the new value so the runtime is O(n).

- Inserting at the end (index = n)

Test	Runtime(microseconds)
1	0
2	0
3	0
4	0
5	0
AVG	0


Year 2 Term 1 > DataStucture-Algorithm > lapPracticeEx1.cpp > main()

```cpp
#include <iostream>
#include <chrono>
#include "operation.hpp"

using namespace std;
using clk = chrono::high_resolution_clock;
volatile int sink_int = 0;

int main (){
    const int MAX_CAP = 100000;  // fixed buffer cap
    int arr[MAX_CAP] = {1,2,3,4};
    int n = MAX_CAP / 2;


    auto t0 = clk::now();
    // testing on each operation
    insertElement (arr, n, n, 10);
    auto t1 = clk::now();
    cout << chrono::duration_cast<chrono::microseconds>(t1-t0).count() << endl;

    return 0;
}
```

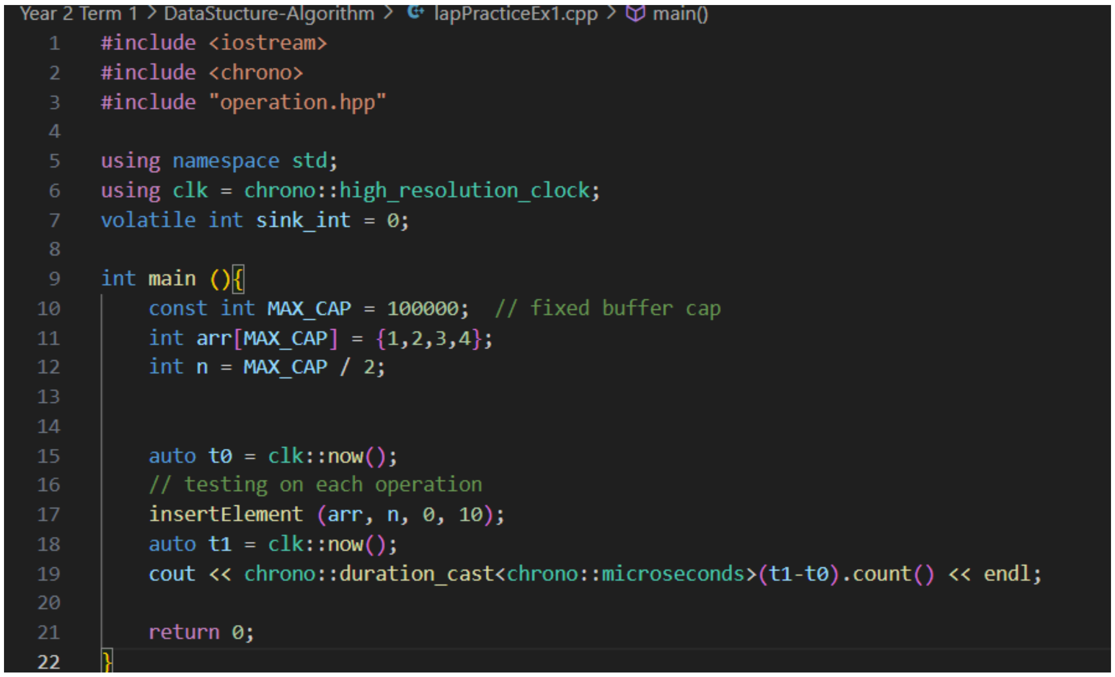
Explanation: Inserting at the end of an array that is not full (half full) require no shifting of the elements so the runtime is O(1) which make it the fastest behavior.

1C. Insert at given index (capacity exists)

Prediction: Fastest to Slowest runtime -> end(n) > middle(n/2) > beginning(0)

- Beginning (index = 0)

Test	Runtime(microseconds)
1	63
2	57
3	51
4	69
5	67
AVG	61



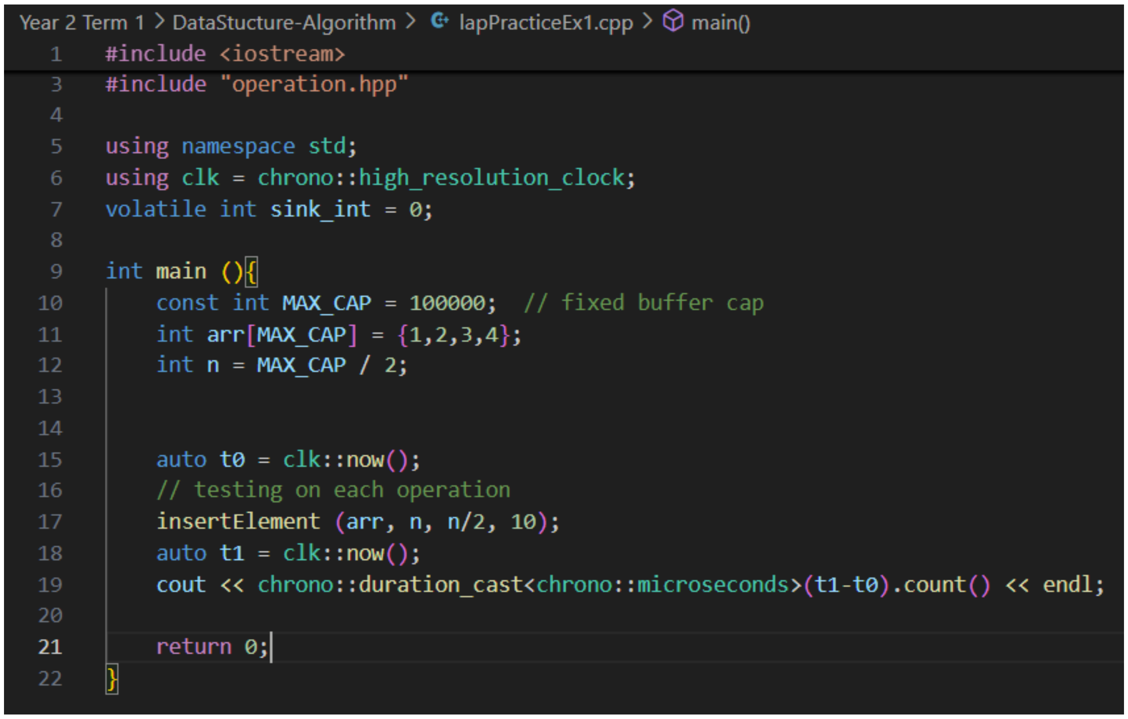
Year 2 Term 1 > DataStucture-Algorithm > lapPracticeEx1.cpp > main()

```cpp
#include <iostream>
#include <chrono>
#include "operation.hpp"

using namespace std;
using clk = chrono::high_resolution_clock;
volatile int sink_int = 0;

int main (){
    const int MAX_CAP = 100000;  // fixed buffer cap
    int arr[MAX_CAP] = {1,2,3,4};
    int n = MAX_CAP / 2;


    auto t0 = clk::now();
    // testing on each operation
    insertElement (arr, n, 0, 10);
    auto t1 = clk::now();
    cout << chrono::duration_cast<chrono::microseconds>(t1-t0).count() << endl;

    return 0;
}
```

Explanation: Inserting at the beginning of an array requires the elements to shift to make space for the new value so the runtime is O(n).

- Middle (index = n/2)

Test	Runtime(microseconds)
1	29
2	30
3	31
4	25
5	29
AVG	29



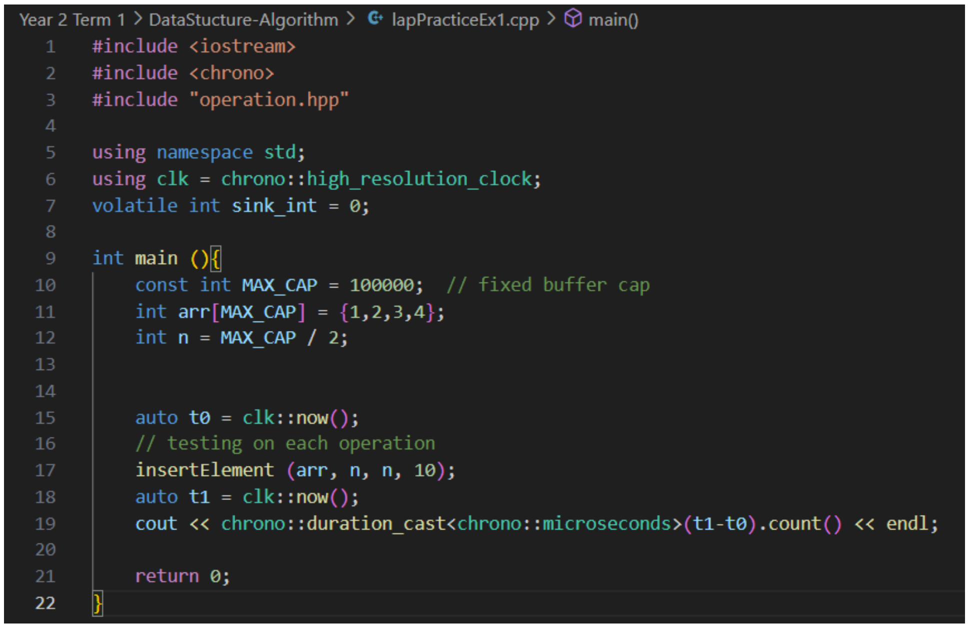
Year 2 Term 1 > DataStucture-Algorithm > lapPracticeEx1.cpp > main()

```cpp
#include <iostream>
#include "operation.hpp"

using namespace std;
using clk = chrono::high_resolution_clock;
volatile int sink_int = 0;

int main (){
    const int MAX_CAP = 100000;  // fixed buffer cap
    int arr[MAX_CAP] = {1,2,3,4};
    int n = MAX_CAP / 2;


    auto t0 = clk::now();
    // testing on each operation
    insertElement (arr, n, n/2, 10);
    auto t1 = clk::now();
    cout << chrono::duration_cast<chrono::microseconds>(t1-t0).count() << endl;

    return 0;
}
```

Explanation: Inserting in the middle of an array requires shifting of half of the elements in the array but not all of the elements (in index = 0 case) making the runtime a bit faster than the first case yet still remains O(n).

- Inserting at the end (index = n)

Test	Runtime(microseconds)
1	0
2	0
3	0
4	0
5	0
AVG	0



Year 2 Term 1 > DataStucture-Algorithm > lapPracticeEx1.cpp > main()

```cpp
#include <iostream>
#include <chrono>
#include "operation.hpp"

using namespace std;
using clk = chrono::high_resolution_clock;
volatile int sink_int = 0;

int main (){
    const int MAX_CAP = 100000;  // fixed buffer cap
    int arr[MAX_CAP] = {1,2,3,4};
    int n = MAX_CAP / 2;


    auto t0 = clk::now();
    // testing on each operation
    insertElement (arr, n, n, 10);
    auto t1 = clk::now();
    cout << chrono::duration_cast<chrono::microseconds>(t1-t0).count() << endl;

    return 0;
}
```

Explanation: Inserting at the end of an array that is require no shifting of the elements so the runtime is O(1).

Tests on each case side-by-side

Test	Case 1 Runtime (microseconds)	Case 2 Runtime (microseconds)	Case 3 Runtime (microseconds)
1	63	29	0
2	57	30	0
3	51	31	0
4	69	25	0
5	67	29	0
AVG	61	29	0

Conclusion: our prediction was right. Inserting at the end (index = n) is the fastest in term of time complexity due to not having to shift any elements in the array while inserting at the middle is faster than inserting an element at the beginning since we don’t have to shift the entirety of the array.

1D. Insert at a given index (capacity exists)

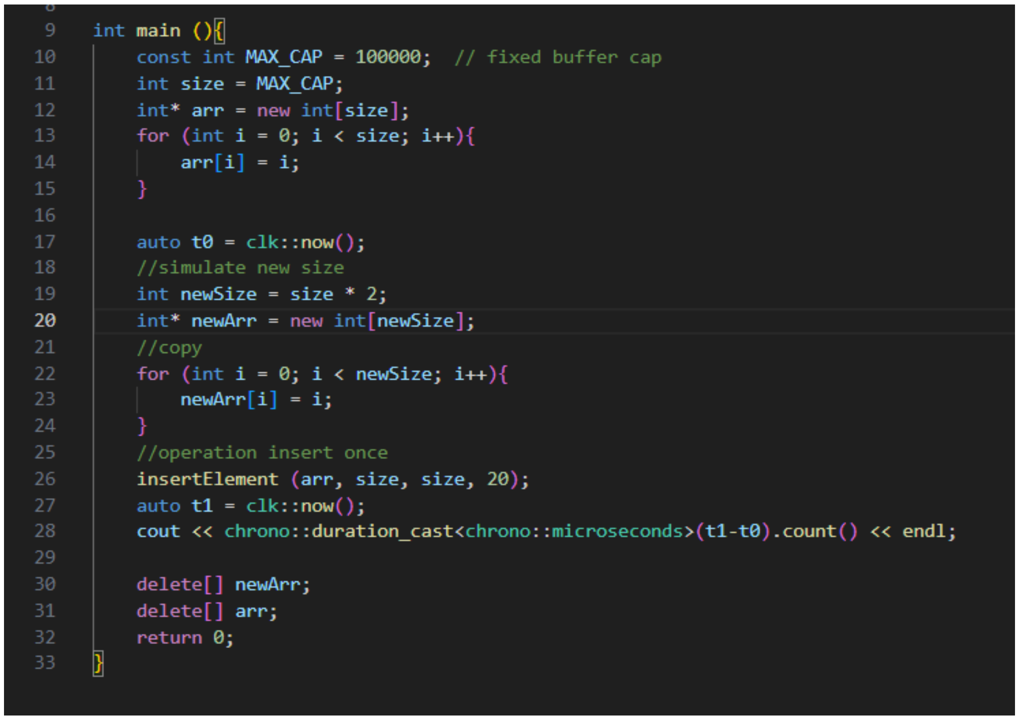
- D1. Insert without overwrite (fixed buffer)

Q: Can you insert without overwrite?

A: No we cant bcus a fixed-capacity array (int arr[max]) cant hold more than “max” elements. Inserting another value would overwrite or go out of bounds (undefined behavior). Resizing is needed.

- D2. Simulate dynamic array

Test	D2 Runtime (microseconds)	1B/1C-end Runtime(microseconds)
1	340	0
2	229	0
3	308	0
4	225	0
5	286	0
AVG	278	0



```
int main (){
    const int MAX_CAP = 100000;  // fixed buffer cap
    int size = MAX_CAP;
    int* arr = new int[size];
    for (int i = 0; i < size; i++){
        arr[i] = i;
    }

    auto t0 = clk::now();
    //simulate new size
    int newSize = size * 2;
    int* newArr = new int[newSize];
    //copy
    for (int i = 0; i < newSize; i++){
        newArr[i] = i;
    }
    //operation insert once
    insertElement (arr, size, size, 20);
    auto t1 = clk::now();
    cout << chrono::duration_cast<chrono::microseconds>(t1-t0).count() << endl;

    delete[] newArr;
    delete[] arr;
    return 0;
}
```

Explanation:

D1: if the fixed array is full, insertion cannot occur without resizing

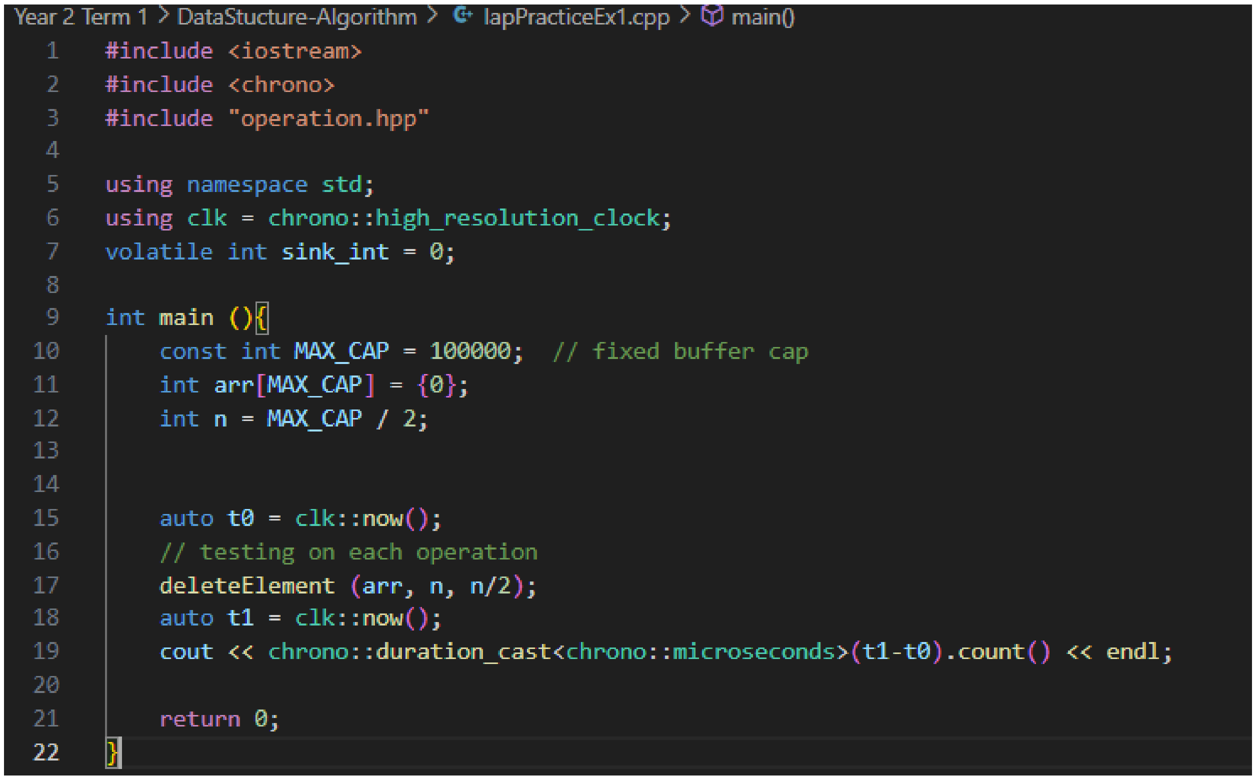
D2: simulate dynamic array by allocating a new buffer of size 2n, copying all elements, and inserting one new element.

So there is an O(n) copy. Compared to 1B/1C-end this is much slower due to memory allocation and element copying.

Challenge 2 - Remove 1 element

2A. Remove from the middle

Test	Runtime (microseconds)
1	32
2	28
3	27
4	30
5	25
AVG	28



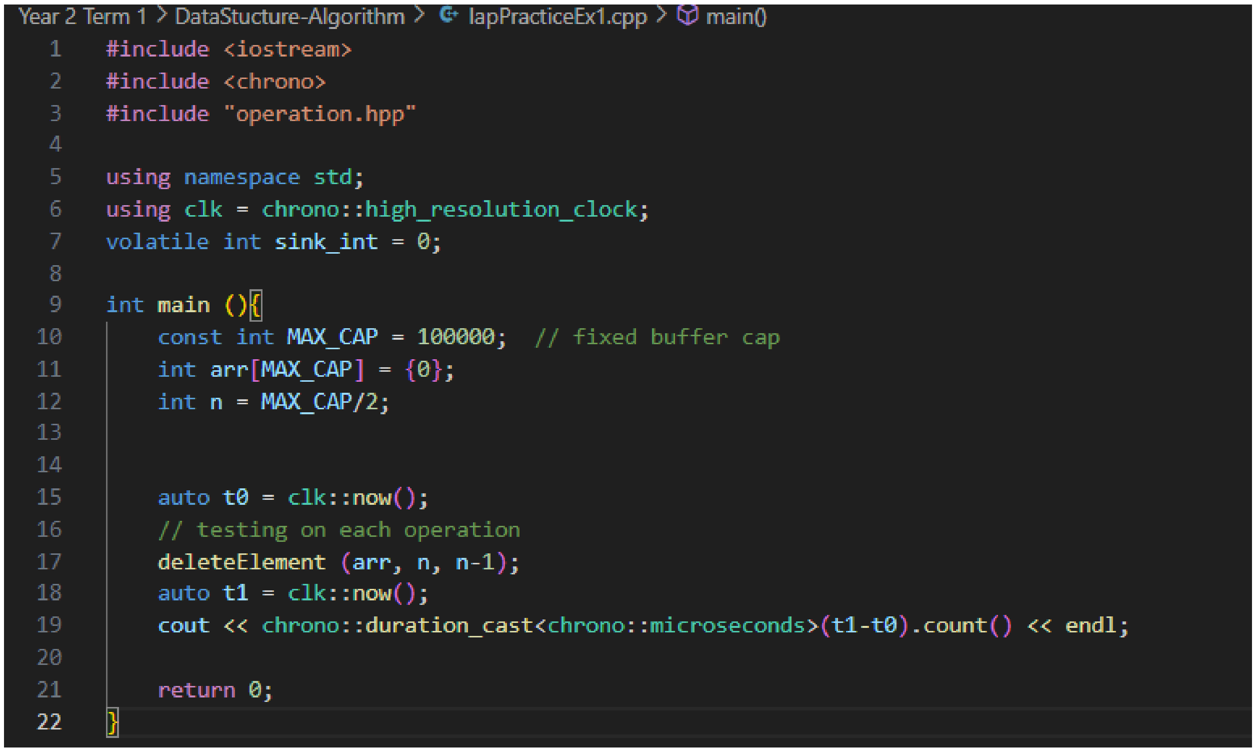
Year 2 Term 1 > DataStucture-Algorithm > lapPracticeEx1.cpp > main()

```cpp
#include <iostream>
#include <chrono>
#include "operation.hpp"

using namespace std;
using clk = chrono::high_resolution_clock;
volatile int sink_int = 0;

int main (){
    const int MAX_CAP = 100000;  // fixed buffer cap
    int arr[MAX_CAP] = {0};
    int n = MAX_CAP / 2;


    auto t0 = clk::now();
    // testing on each operation
    deleteElement (arr, n, n/2);
    auto t1 = clk::now();
    cout << chrono::duration_cast<chrono::microseconds>(t1-t0).count() << endl;

    return 0;
}
```

Explanation: Removing a value from the middle of an array requires half of the elements in the array to shift to the left after removal so it’s O(n).

2B. Remove from the end

Test	Runtime (microseconds)
1	0
2	0
3	0
4	0
5	0
AVG	0

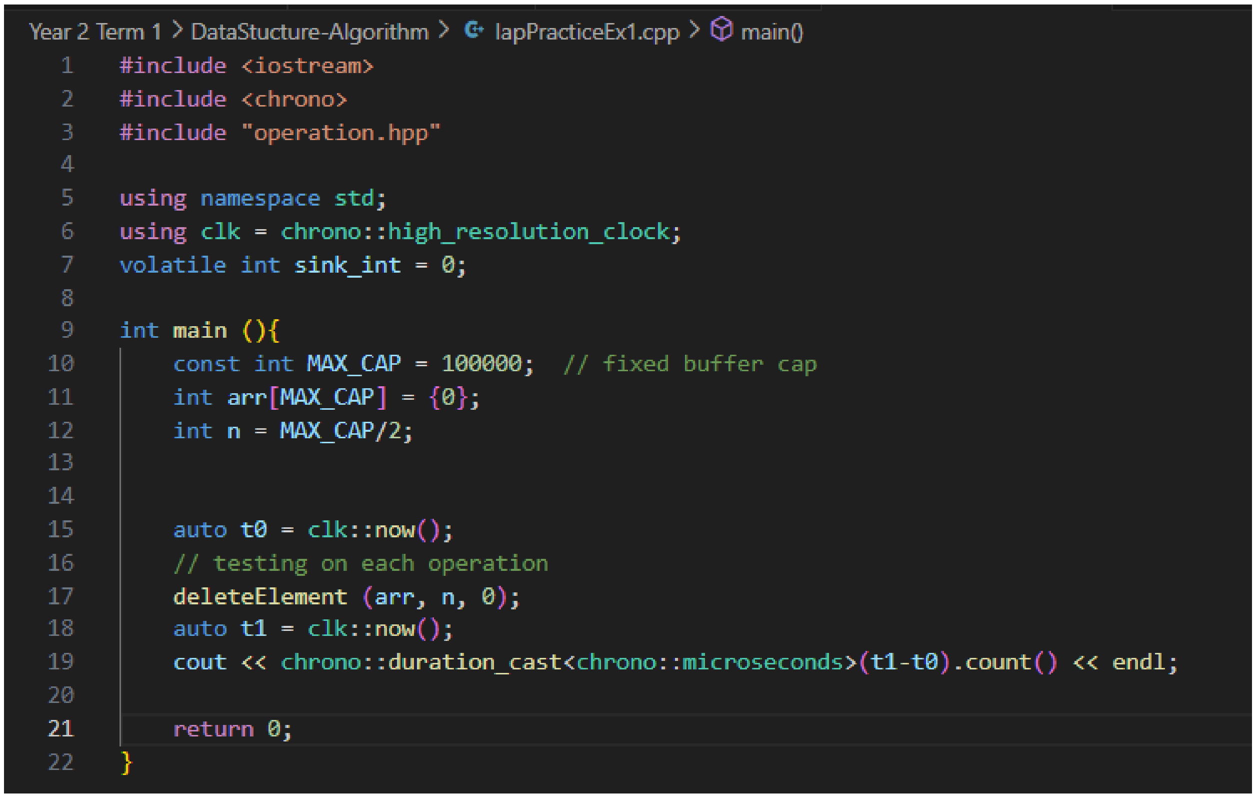

Year 2 Term 1 > DataStucture-Algorithm > lapPracticeEx1.cpp > main()

```cpp
#include <iostream>
#include <chrono>
#include "operation.hpp"

using namespace std;
using clk = chrono::high_resolution_clock;
volatile int sink_int = 0;

int main (){
    const int MAX_CAP = 100000;  // fixed buffer cap
    int arr[MAX_CAP] = {0};
    int n = MAX_CAP/2;


    auto t0 = clk::now();
    // testing on each operation
    deleteElement (arr, n, n-1);
    auto t1 = clk::now();
    cout << chrono::duration_cast<chrono::microseconds>(t1-t0).count() << endl;

    return 0;
}
```

Explanation: Removing a value from the end of an array doesn't require any shifting so the operation time is O(1).

2C. Remove from the beginning

Test	Runtime(microseconds)
1	62
2	57
3	53
4	66
5	55
AVG	59



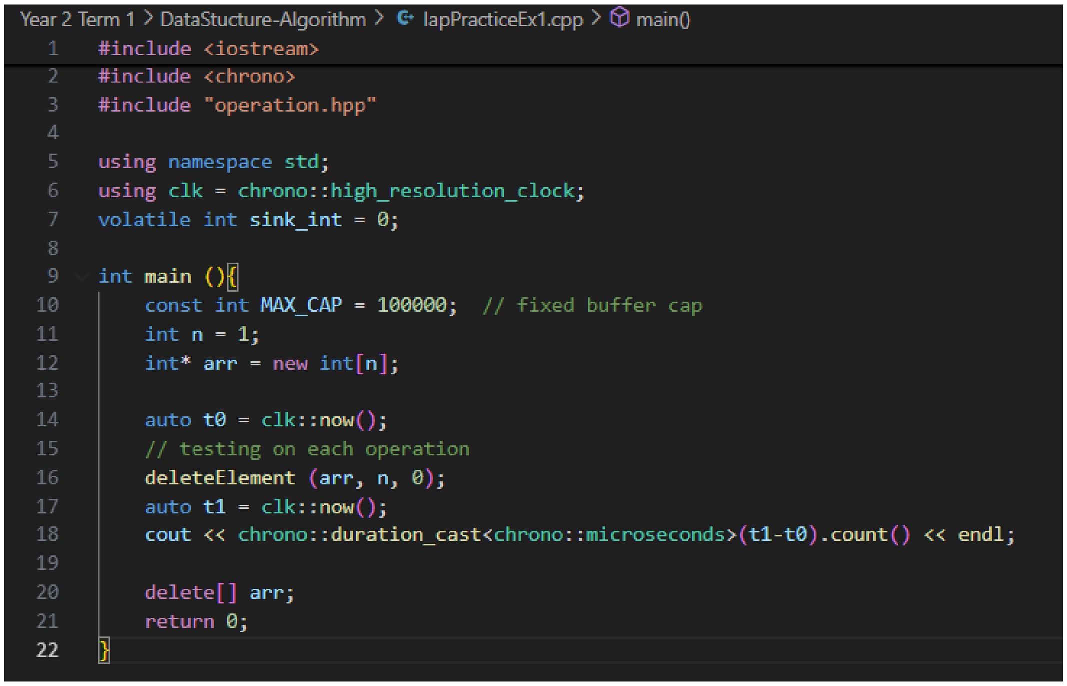
Year 2 Term 1 > DataStucture-Algorithm > lapPracticeEx1.cpp > main()

```cpp
#include <iostream>
#include <chrono>
#include "operation.hpp"

using namespace std;
using clk = chrono::high_resolution_clock;
volatile int sink_int = 0;

int main (){
    const int MAX_CAP = 100000;  // fixed buffer cap
    int arr[MAX_CAP] = {0};
    int n = MAX_CAP/2;


    auto t0 = clk::now();
    // testing on each operation
    deleteElement (arr, n, 0);
    auto t1 = clk::now();
    cout << chrono::duration_cast<chrono::microseconds>(t1-t0).count() << endl;

    return 0;
}
```

Explanation: Removing a value from the beginning of an array requires shifting of the rest of the array back in place so it takes time. It is O(n).

2D. Edge Sanity

Test	Runtime (microseconds)
1	0
2	0
3	0
4	0
5	0
AVG	0



Year 2 Term 1 > DataStucture-Algorithm > lapPracticeEx1.cpp > main()

```cpp
#include <iostream>
#include <chrono>
#include "operation.hpp"

using namespace std;
using clk = chrono::high_resolution_clock;
volatile int sink_int = 0;

int main (){
    const int MAX_CAP = 100000;  // fixed buffer cap
    int n = 1;
    int* arr = new int[n];

    auto t0 = clk::now();
    // testing on each operation
    deleteElement (arr, n, 0);
    auto t1 = clk::now();
    cout << chrono::duration_cast<chrono::microseconds>(t1-t0).count() << endl;

    delete[] arr;
    return 0;
}
```

Explanation: the size is 1 and we only have to removing 1 element thus no shifting so it’s fast.

Comparison: Compared to 2A and 2C which is measured this one is much faster by comparison since there is no need for shifting

Challenge 3 - Single Question

Accessing an element by index like arr[i] requires no scanning and is a constant time operation aka O(1) because it provide the direct index that the compiler can access the memory address. For instance, if i run small timing loop reading arr[i] for many different i values , it would show the constant times regardless of array size. In contrast, linear search would take a lot more time since we have to read thru every element in the array until u find a match/target. Best case scenario is when the target is at index 0 the very first index so the search is done under a constant time O(1) but the worst case scenario is when the target is at the end of the array or isn’t in the array at all which mean we have to scan thru everything n times making the time complexity O(n).