Entry no:2022EE31907

For Stack_A:

```
void Stack_A ::push(int data){
    if(  size <= 1024){
    //  counter=counter+1;
      //  size=size+1;
      stk[size]=data;
       size=size+1;


    }
    else{
    throw std::runtime_error("Stack Full");

    }

    }
```

In This, we are comparing the size with 1024, and then adding elements to the stack with the help of a fixed array takes O(1) time complexity, as this code uses only comparison of size and adding data to its place, which can be done in O[1]

Time complexity.

```
int Stack_A ::pop(){
    if (size<1){
     throw std::runtime_error("Empty Stack");

    //  return -1;
    }
    else{
      // int t = stk[size-1];
      // counter--;
      size=size-1;
      return stk[size];
    }

    }
```

In this also only if-else statements are used which can be performed in O[1] time complexity.

For Stack_B:

```
void Stack_B ::push(int data){

    if(size>=capacity){
      int capacity2=capacity*2;
      int*temp = new(std::nothrow) int[capacity2];
     if(temp){
```

```
        for(int i=0;i<size;i++){
         temp[i]=stk[i];
         }
        delete[] stk;
       stk=temp;
       capacity=capacity2;

   }
   else{
     throw std::runtime_error("Out Of Memory");
   }
   }
       stk[size]=data;
        size=size+1;

}
```

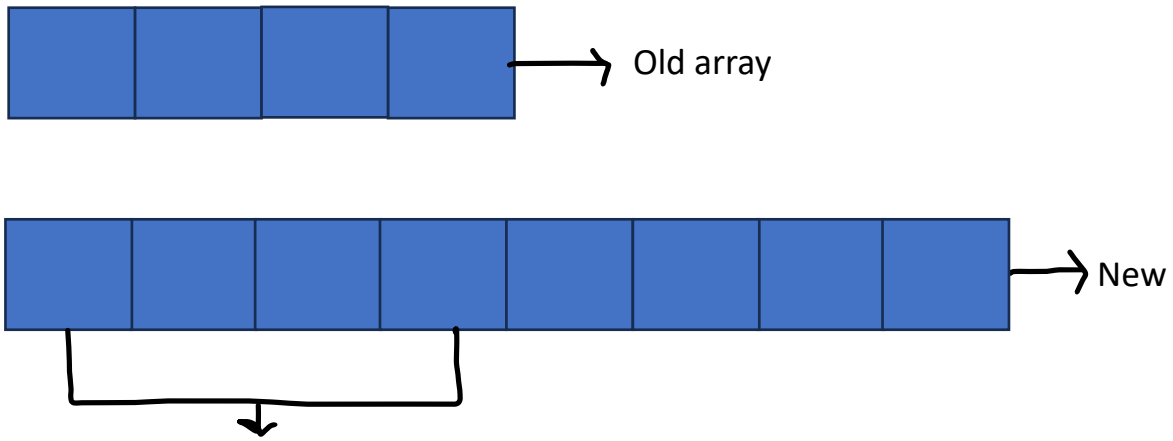
In this we are comparing the capacity of array with the size of array, if size is greater than array capacity, which is initially 1024, then we will double the capacity of array and copy the elements present in the array into new array and then allocate new array to replace original array after deleting it.

Case 1:

In this if (capacity of array > size of array) then there is no need of doubling capacity, hence in that case we are pushing elements in array with the help of if-else statements, hence for this case time complexity is O(1).

Case 2:

In this if (capacity of array < size of the array) then there is a need of doubling capacity, for this situation we will first create new dynamic array and copy the old dynamic array into the newly created array by doubling its capacity.



Copying elements from the old array to the new array takes O(n) time, as we are using for loop from i=0 to the size of an array. After that, we are deleting the old array and the capacity of the array is set to the new capacity.

But case 2 is the worst case and this happens infrequently, hence in the average case the time complexity of the array is O[1] only.

For average case=

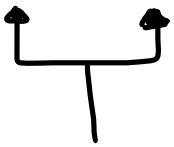
$$\frac{[1+1+1+\ldots\ldots\ldots\ldots\ldots+n]}{n} = (2n-1)/n$$ hence $O[(2n-1)/n] = O[2] = O[1]$

hence amortised time complexity = O[1]

```
int Stack_B ::pop(){
     if (size == 0){
     throw std::runtime_error("Empty Stack");
     }
    int result=stk[size-1];
        size=size-1;
      if((capacity/4)>1024){
        if(size<(capacity/4)){
      int capacity2=(capacity/2);
      int *temp= new int[capacity2];
      for(int i=0;i<size;i++){
         temp[i]=stk[i];

      }
       delete[] stk;
         stk=temp;
         capacity=capacity2;
    }
        }

      return result;


    }
```

As said in the push, in-pop operation we will first check whether elements present in array(array size) are less $1/4^{th}$ of its capacity or not, if they are then it will create a new array which is ½ of its capacity and copy elements from old array to new one, which is done by using for loop hence time complexity will be O[n].

old



elements present in array upto this.

Then we assign new array to old one after deleting old one and assign capacity to new capacity.

On other hand if size of array > capacity/4 then there only size is reduced and poping takes place hence for that case time complexity will be O(1).

The case where size of array <capacity/4 is not that much frequent, hence on an average the time complexity for pop operation will be O[1] only.

For average case=

$$\frac{[1+1+1+\ldots\ldots\ldots\ldots\ldots+n]}{n} = (2n-1)/n$$ hence $O[(2n-1)/n] = O[2] = O[1]$

hence amortised time complexity = O[1]

For Stack_C:

```
void Stack_C ::push(int data){

    return stk->insert(data);
}
```

```
void List::insert(int v) {
    // try {
        Node* n = new (nothrow) Node(v, nullptr, nullptr);
        if (!n) {
            throw runtime_error("Out Of Memory");
        }

        n->prev = sentinel_tail->prev;
        n->next = sentinel_tail;
        sentinel_tail->prev->next = n;
        sentinel_tail->prev = n;
        size = size + 1;
    // } catch (const bad_alloc& e) {
    //     throw runtime_error("Memory allocation failed");
    // }
}
Node* List:: get_head(){
```

```
  Node* pointer= sentinel_head;
  return sentinel_head;

   }
```

For doing push operation we are using insert from the list class, that takes O[1] time as only if condition used to check runtime error and further statement allocation will take O[1] time complexity only.

```
int Stack_C ::pop(){
    if(stk->get_size()==0){
        throw std::runtime_error("Empty Stack");
        // return -1;
    }
    return stk->delete_tail();


}
```

```
int List:: delete_tail(){

  Node* temp= sentinel_tail->prev;
  int data= temp->get_value();
  sentinel_tail->prev=temp->prev;
  temp->prev->next=sentinel_tail;
  delete temp;


 size=size-1;
 return data;
  }
```

For doing pop operation we are using delete_tail from the list class, which takes O[1] time as the only assignment is used to store node links and then the tail node is deleted, which is done with O[1] time complexity.