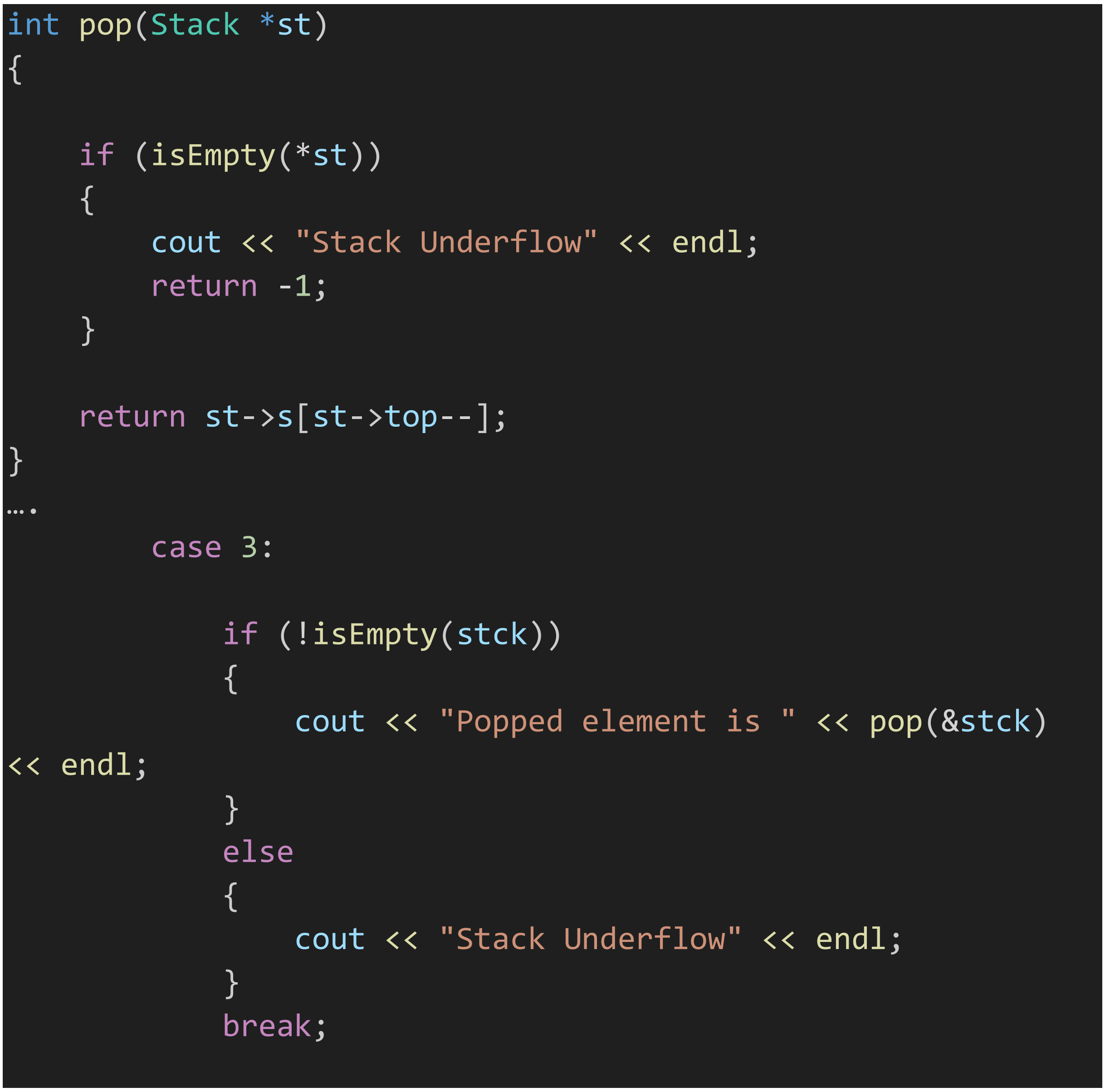
Stack Mechanism Discussion with Time Complexity

5. *Pop Operation*



```
int pop(Stack *st)
{

    if (isEmpty(*st))
    {
        cout << "Stack Underflow" << endl;
        return -1;
    }

    return st->s[st->top--];
}
….
        case 3:

            if (!isEmpty(stck))
            {
                cout << "Popped element is " << pop(&stck)
<< endl;
            }
            else
            {
                cout << "Stack Underflow" << endl;
            }
            break;
```

if stack is Empty then it will print : ``Stack Underflow``.

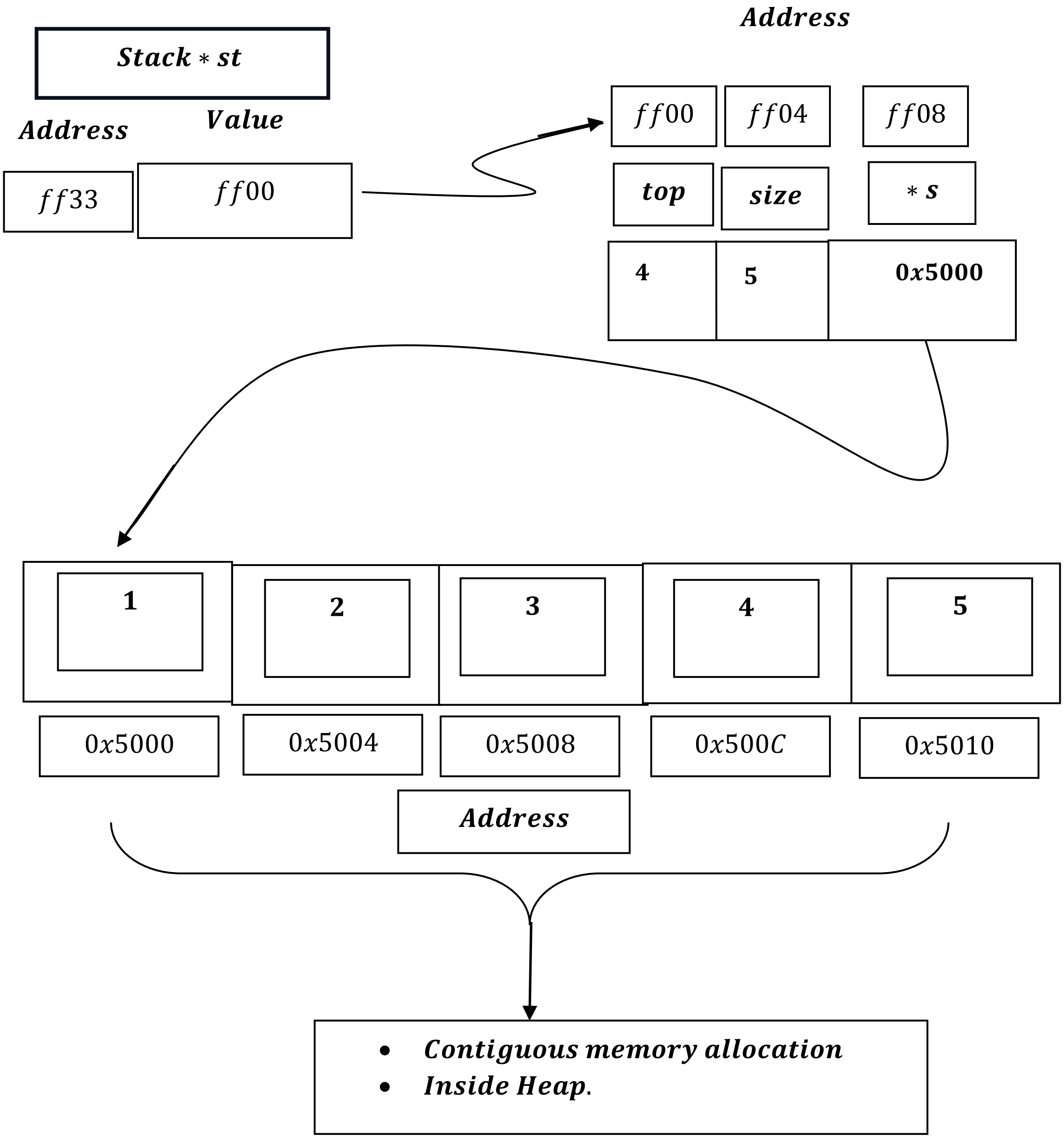
else Top = Top − 1 and element `x` [user input] is pushed to top of the stack.

We have to remember we are doing post decrement .

$\underline{1.\textit{When Stack is Full}:}$

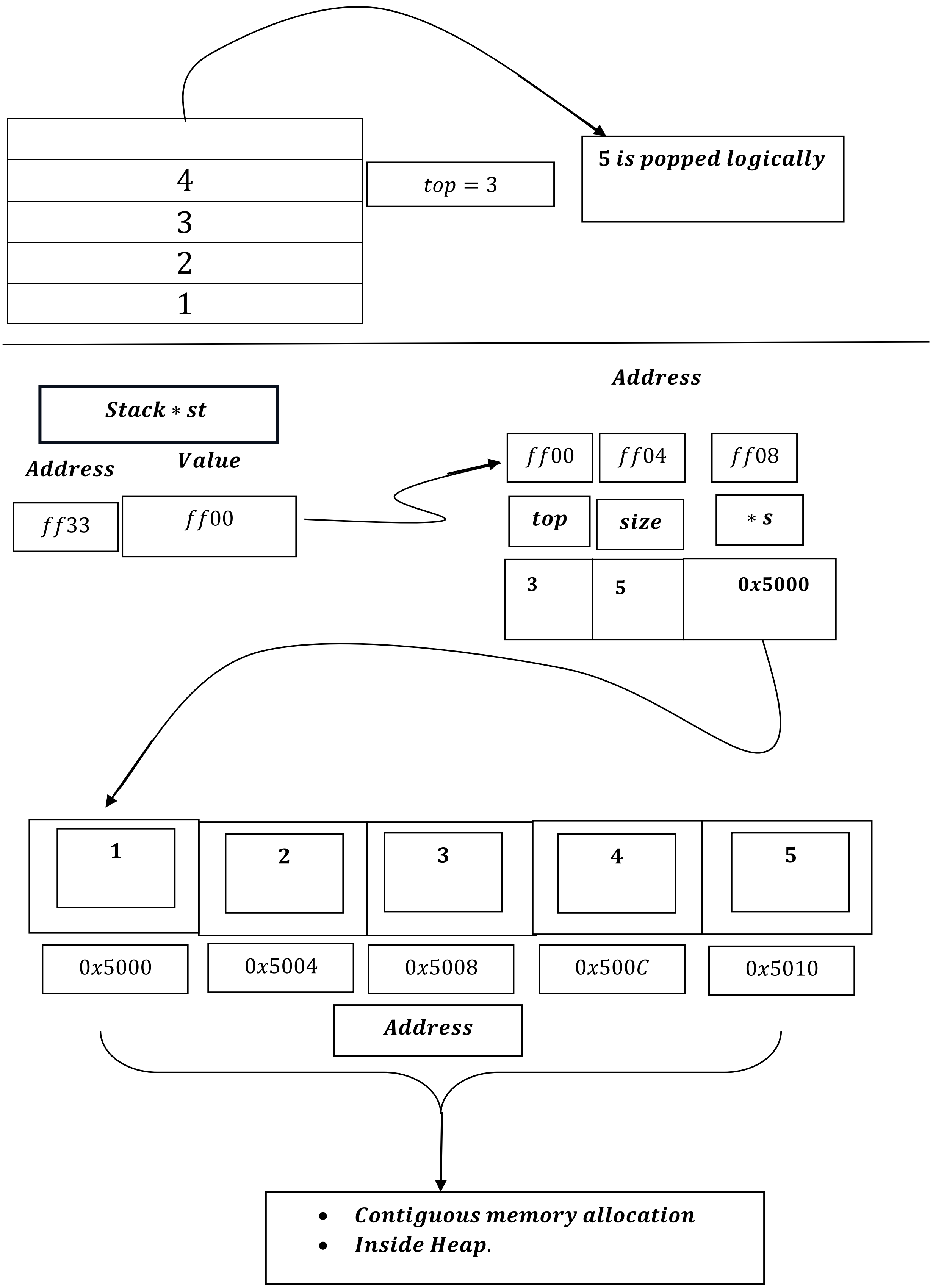
5
4
3
2
1

$top = 4$



- *Contiguous memory allocation*
- *Inside Heap.*

$1st\ Pop\ (5)$



- *Contiguous memory allocation*
- *Inside Heap.*

Note: its post decrement , hence :

$st \rightarrow s[st \rightarrow top] \Rightarrow s[4]$

$return\ value\ stored\ in\ BaseAddress + (index\ \times size\ of\ int)$

$i.e. return\ value\ stored\ in\ 0x5000 + (4 \times 4\ bytes)$

$i.e. return\ value\ stored\ in\ 0x5000 + 16$

$i.e. return\ value\ stored\ in\ 0x5000 + 10$

$i.e. return\ value\ stored\ in\ 0x5010$

$\Rightarrow return\ 5$

$Now\ \ Top\ \ becomes : 3\ , from : st \rightarrow top - - .$

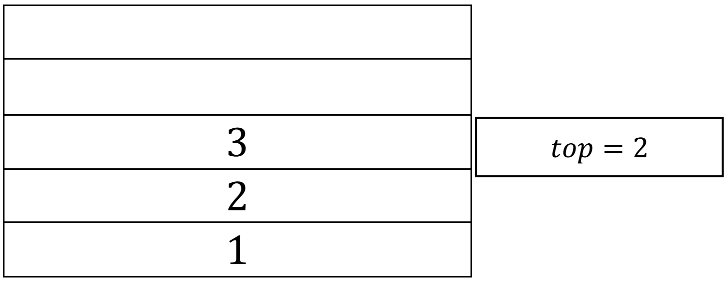
$= st \rightarrow top = st \rightarrow top - 1.$

$= st \rightarrow top = 4 - 1.$

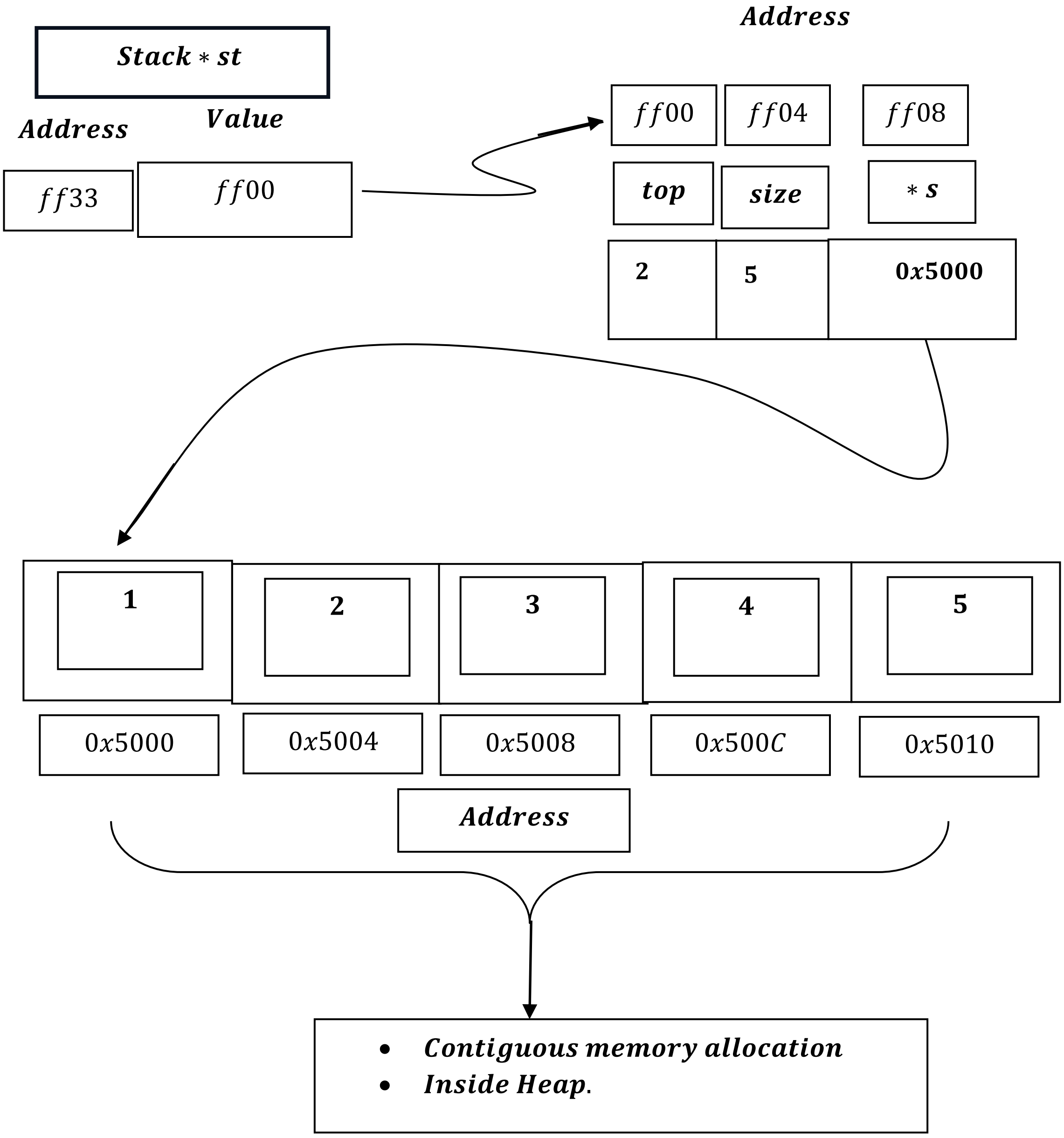
$= st \rightarrow top = 3.$

$Though\ top\ value\ changed\ , return\ value\ `5`\ but\ `5`\ is\ still\ physically\ present\ until\ it\ is\ freed\ manually\ .$

$2nd\ Pop\ (4)$



$4\ is\ popped\ \ logically.$



$$st \to s[st \to top] : \Rightarrow s[3].$$

$$return\ value\ stored\ in\ BaseAddress + (index\ \times size\ of\ int)$$

$$i.e. return\ value\ stored\ in\ 0x5000 + (3 \times 4\ bytes)$$

$$i.e. return\ value\ stored\ in\ 0x5000 + 12$$

$$i.e. return\ value\ stored\ in\ 0x5000 + C$$

$$i.e. return\ value\ stored\ in\ 0x500C$$

$$\Rightarrow return\ 4$$

$\boldsymbol{Now\ \ Top\ \ becomes : 2\ , from : st \rightarrow top - - .}$

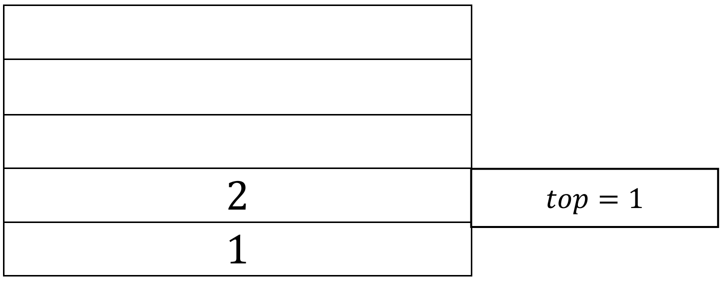
$\boldsymbol{= st \rightarrow top = st \rightarrow top - 1.}$

$\boldsymbol{= st \rightarrow top = 3 - 1.}$

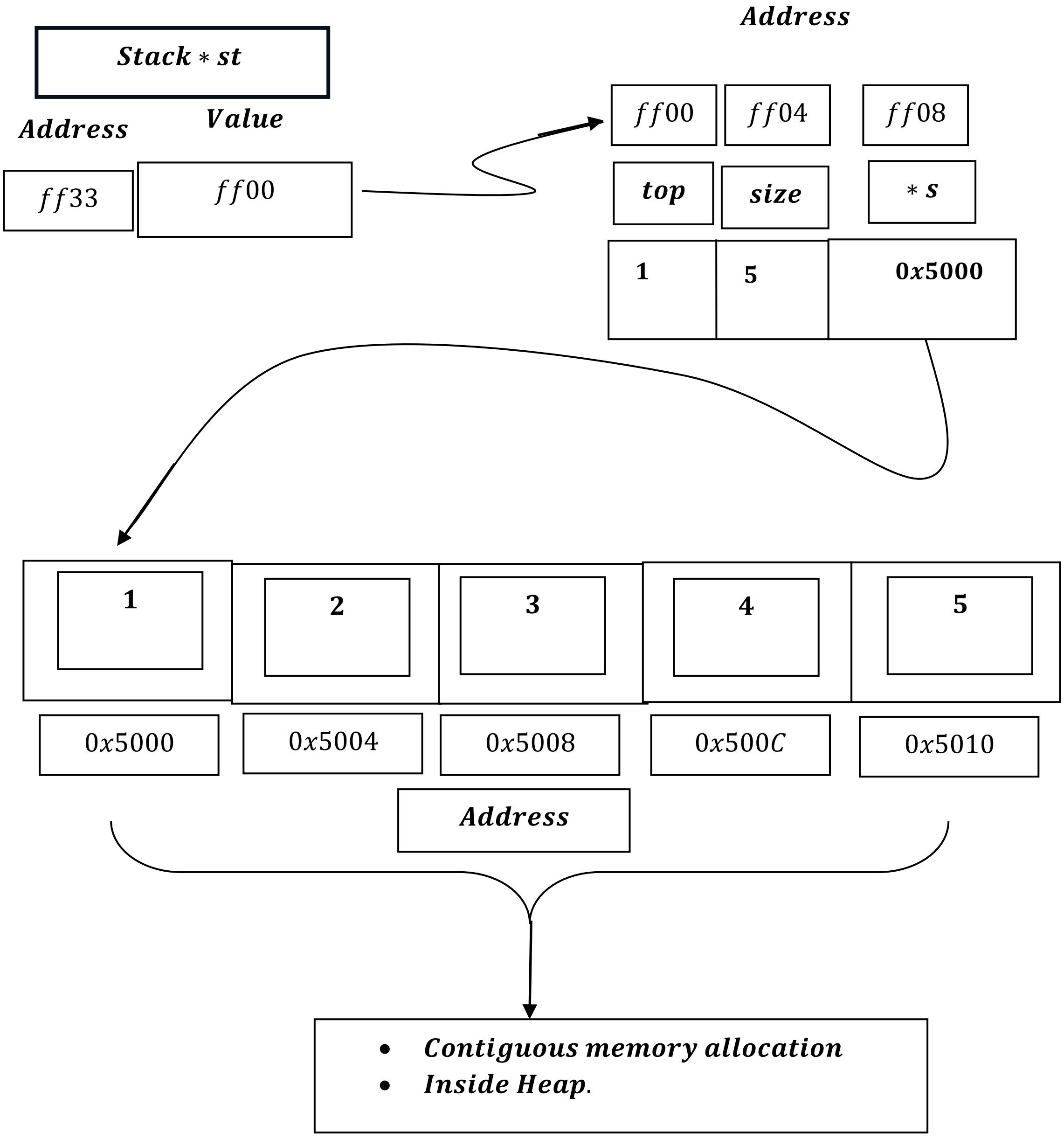
$\boldsymbol{= st \rightarrow top = 2.}$

Though top value changed ,return value`4` but `4` is still physically present until it is freed manually .

$\boldsymbol{3rd\ Pop\ (3)}$



3 is popped logically.



$st \rightarrow s[st \rightarrow top]: \Rightarrow s[2].$

$return\ value\ stored\ in\ BaseAddress + (index\ \times size\ of\ int)$

$i.e. return\ value\ stored\ in\ 0x5000 + (2 \times 4\ bytes)$

$i.e. return\ value\ stored\ in\ 0x5000 + 8$

$i.e. return\ value\ stored\ in\ 0x5008$

$\Rightarrow return\ 3$

$\boldsymbol{Now\ \ Top\ \ becomes : 1\ , from : st \rightarrow top - - \ .}$

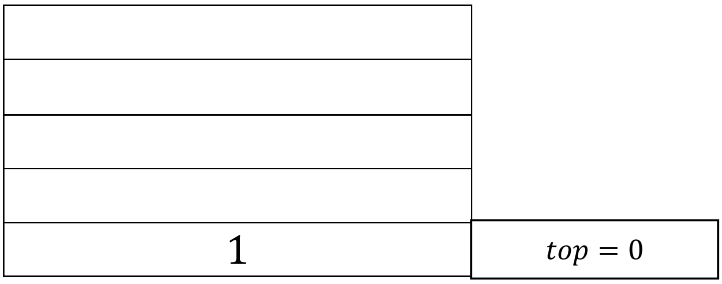
$\boldsymbol{= st \rightarrow top = st \rightarrow top - 1.}$

$\boldsymbol{= st \rightarrow top = 2 - 1.}$

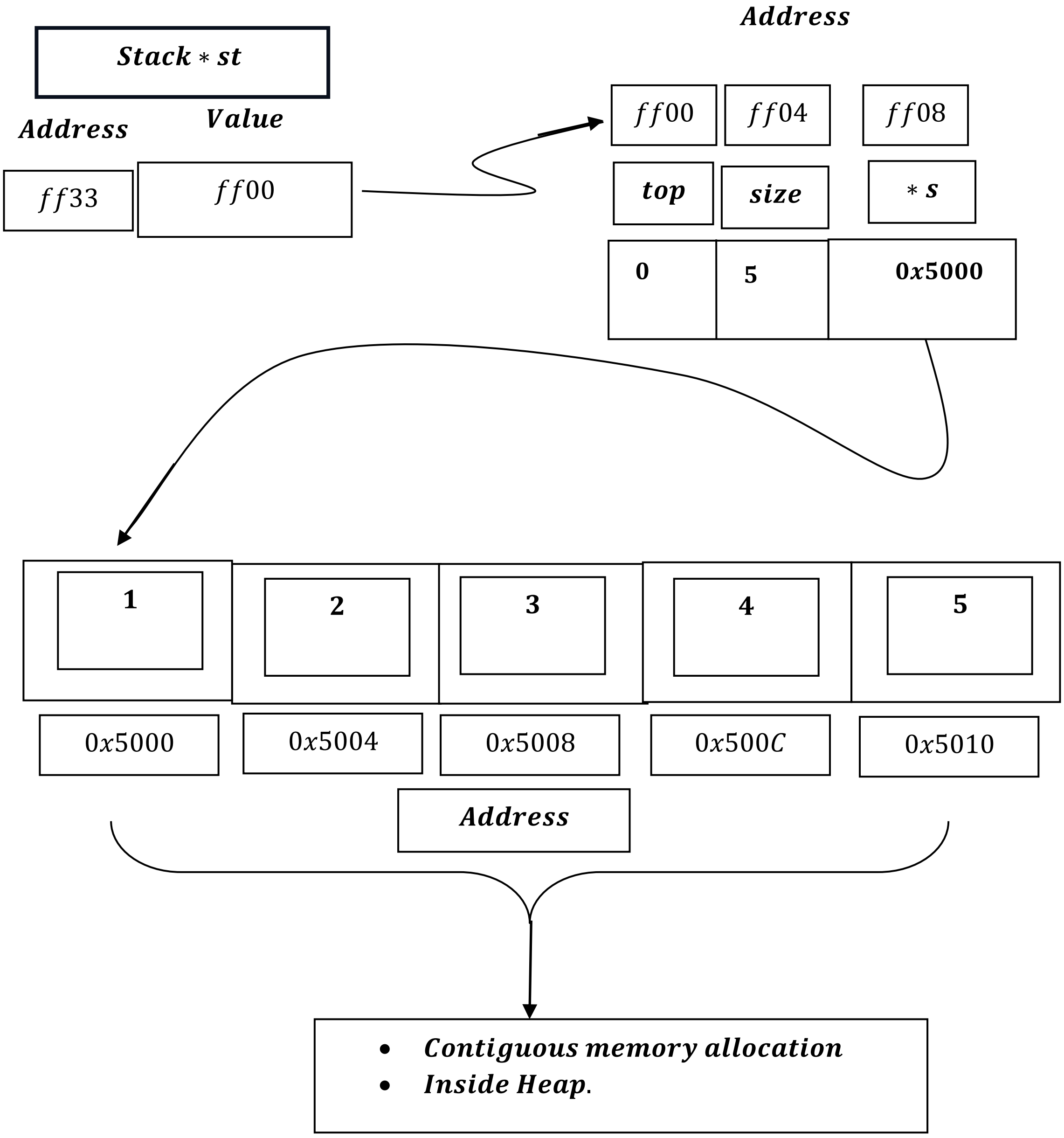
$\boldsymbol{= st \rightarrow top = 1.}$

$\boldsymbol{Though\ top\ value\ changed\ , return\ value\ `3`\ but\ `3`\ is\ still\ physically\ present\ until\ it\ is\ freed\ manually\ .}$

$\boldsymbol{4rth\ Pop\ (2)}$



$\boldsymbol{2\ is\ popped\ logically.}$



- *Contiguous memory allocation*
- *Inside Heap.*

$st \rightarrow s[st \rightarrow top] : \Rightarrow s[1].$

$return\ value\ stored\ in\ BaseAddress + (index\ \times size\ of\ int)$

$i.e. return\ value\ stored\ in\ 0x5000 + (1 \times 4\ bytes)$

$i.e. return\ value\ stored\ in\ 0x5000 + 4$

$i.e. return\ value\ stored\ in\ 0x5004$

$\Rightarrow return\ 2$

Now Top becomes : 0 , from : $st \rightarrow top --$.

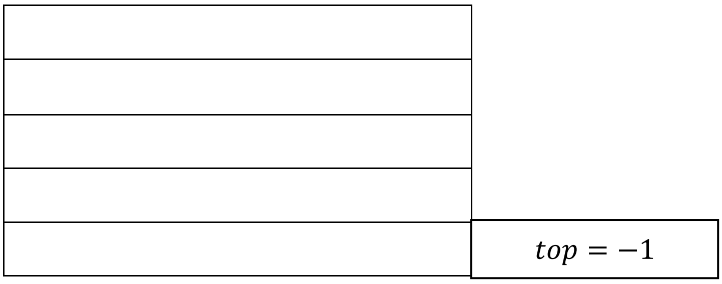
$= st \rightarrow top = st \rightarrow top - 1.$

$= st \rightarrow top = 1 - 1.$

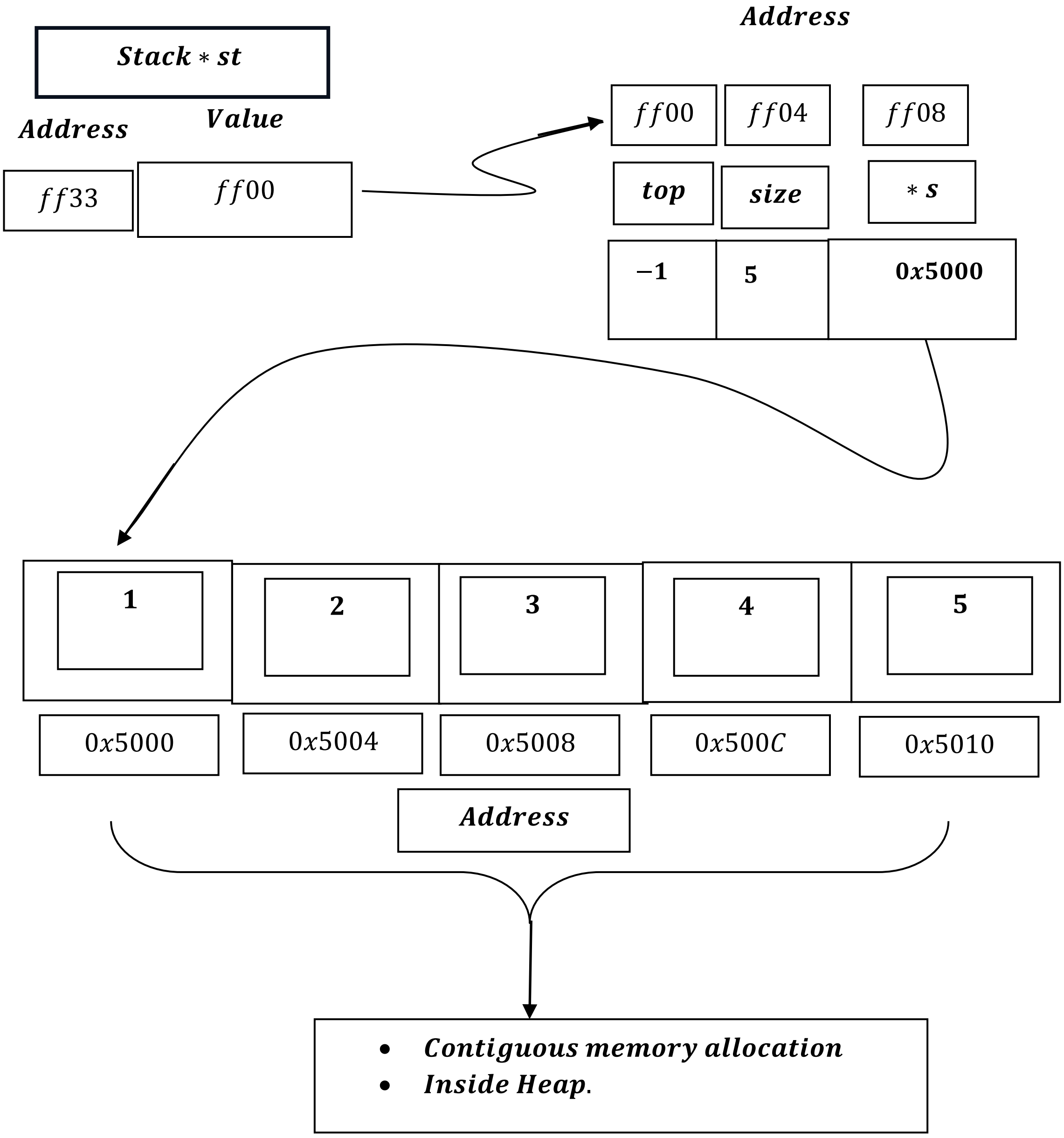
$= st \rightarrow top = 0.$

Though top value changed , return value `2` but `2` is still physically present until it is freed manually .

$5th\ Pop\ (1)$



1 is popped logically.



- *Contiguous memory allocation*
- *Inside Heap.*

$$st \to s[st \to top]: \Rightarrow s[0].$$

$$return\ value\ stored\ in\ BaseAddress + (index\ \times size\ of\ int)$$

$$i.e. return\ value\ stored\ in\ 0x5000 + (0 \times 4\ bytes)$$

$$i.e. return\ value\ stored\ in\ 0x5000 + 0$$

$$i.e. return\ value\ stored\ in\ 0x5000$$

$$\Rightarrow return\ 1$$

$Now\ Top\ becomes: -1, from: st \rightarrow top--.$

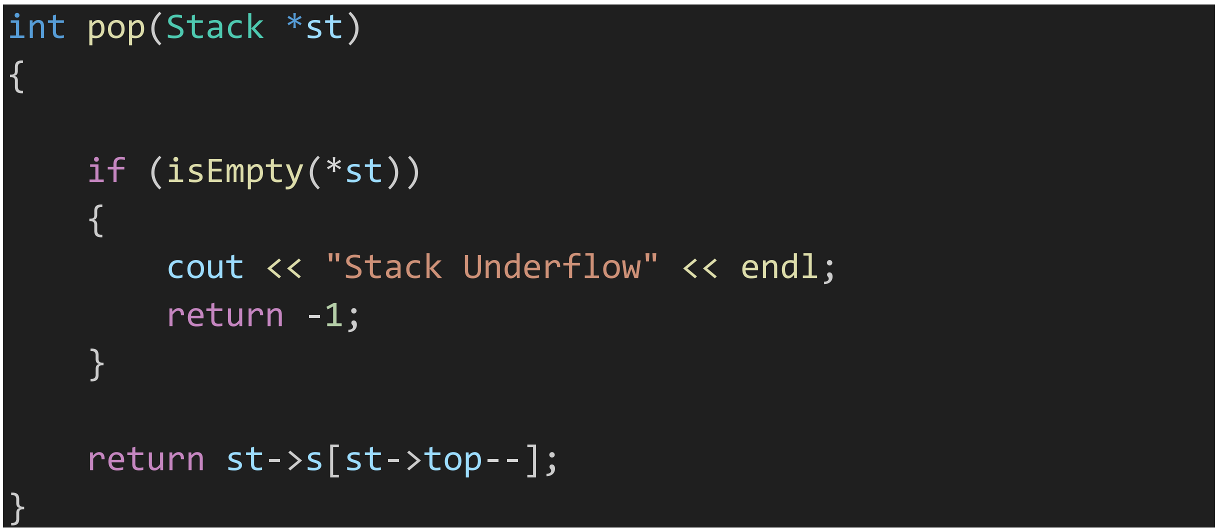
$= st \rightarrow top = st \rightarrow top - 1.$

$= st \rightarrow top = 0 - 1.$

$= st \rightarrow top = -1.$

$Though\ top\ value\ changed, return\ value\ `1`\ but\ `1`\ is\ still\ physically\ present\ until\ it\ is\ freed\ manually.$

$Time\ Complexity$



```
int pop(Stack *st)
{

    if (isEmpty(*st))
    {
        cout << "Stack Underflow" << endl;
        return -1;
    }

    return st->s[st->top--];
}
```

$\rightarrow Function\ Overhead\ takes\ constant\ amount\ of\ time\ complexity.\ i.e.\ O(1).$

$\rightarrow If\ isEmpty()\ function\ returns\ true\ i.e.\ 1, condition\ checking\ takes\ constant\ amount\ of\ time\ complexity\ i.e.\ O(1).$

$\rightarrow Prints\ ``Stack\ Underflow``\ takes\ constant\ time: O(1).$

$$\rightarrow \boldsymbol{return - 1\ \ takes\ constant\ time} : \boldsymbol{O}(1).$$

$\rightarrow \boldsymbol{return\ st \rightarrow s[st \rightarrow top]\ takes\ constant\ amount\ of\ time} : \boldsymbol{O}(1).$

$\rightarrow \boldsymbol{Decrement\ Top\ i.e. Top = Top - 1\ takes\ constant\ amount\ of\ time}: \boldsymbol{O}(1).$ $\boldsymbol{Due\ to\ post\ decrement.}$

$$\boldsymbol{Total\ Time\ Complexity} : \boldsymbol{O}(1) + (\boldsymbol{O}(1) + (\boldsymbol{O}(1) + \boldsymbol{O}(1)) + \boldsymbol{O}(1) + \boldsymbol{O}(1) = \boldsymbol{O}(1).$$

$$\boldsymbol{i.e. O}(1)\ \boldsymbol{ran\ constant\ amount\ of\ time\ not\ `n`\ times}.$$

What is Function Call Overhead?

Function call overhead means the small amount of extra work the system does when a function is called.

```
int pop(Stack *st)
{

    if (isEmpty(*st))
    {
        cout << "Stack Underflow" << endl;
        return -1;
    }

    return st->s[st->top--];
}
```

$\boldsymbol{Function\ call}$:

```
        pop(&stck);
```

Even if our function does almost nothing, the system still has to:

1. Save the current execution state
2. Pass arguments to the function
3. Allocate space for local variables
4. Jump to the function’s memory location
5. After execution, return back to the caller

All of this takes a **constant amount of time**.

Stack Data structure basically shows how call stack frame works i.e. in LIFO method , but in Stack Data Structure we manually do push() and pop() , where in call stack frame, it creates stack frame for entire function containing local variables , parameters , return address and saved registers and adds or push another stack frame if called again and pops whole stack frame automatically thus different from Stack datastructure , yet conceptually similar.
