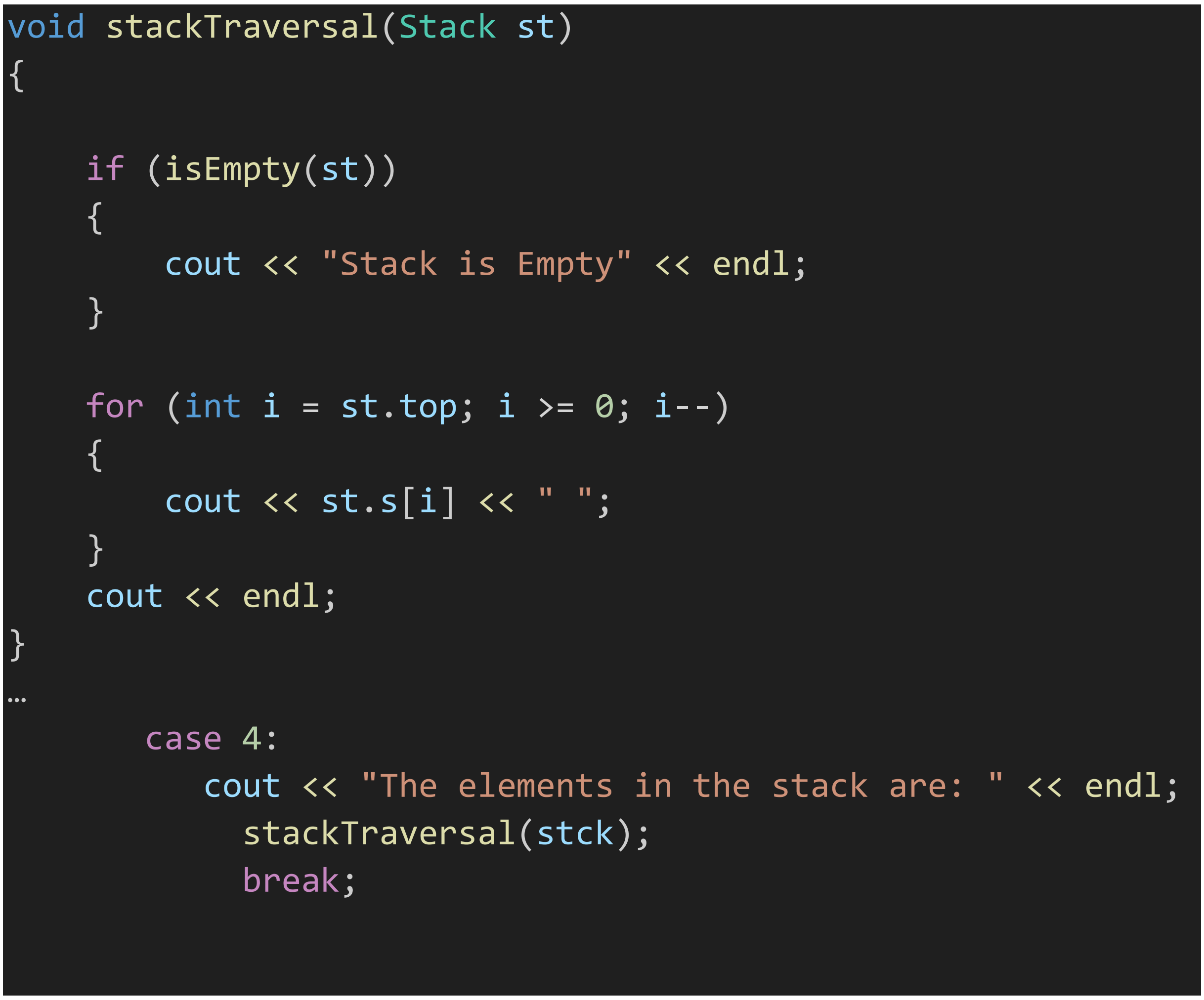
7. Stack Traversal Operation

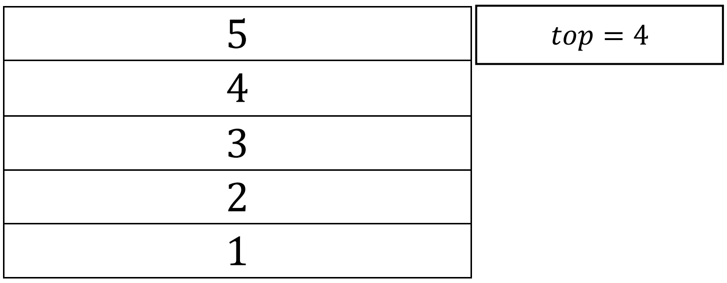


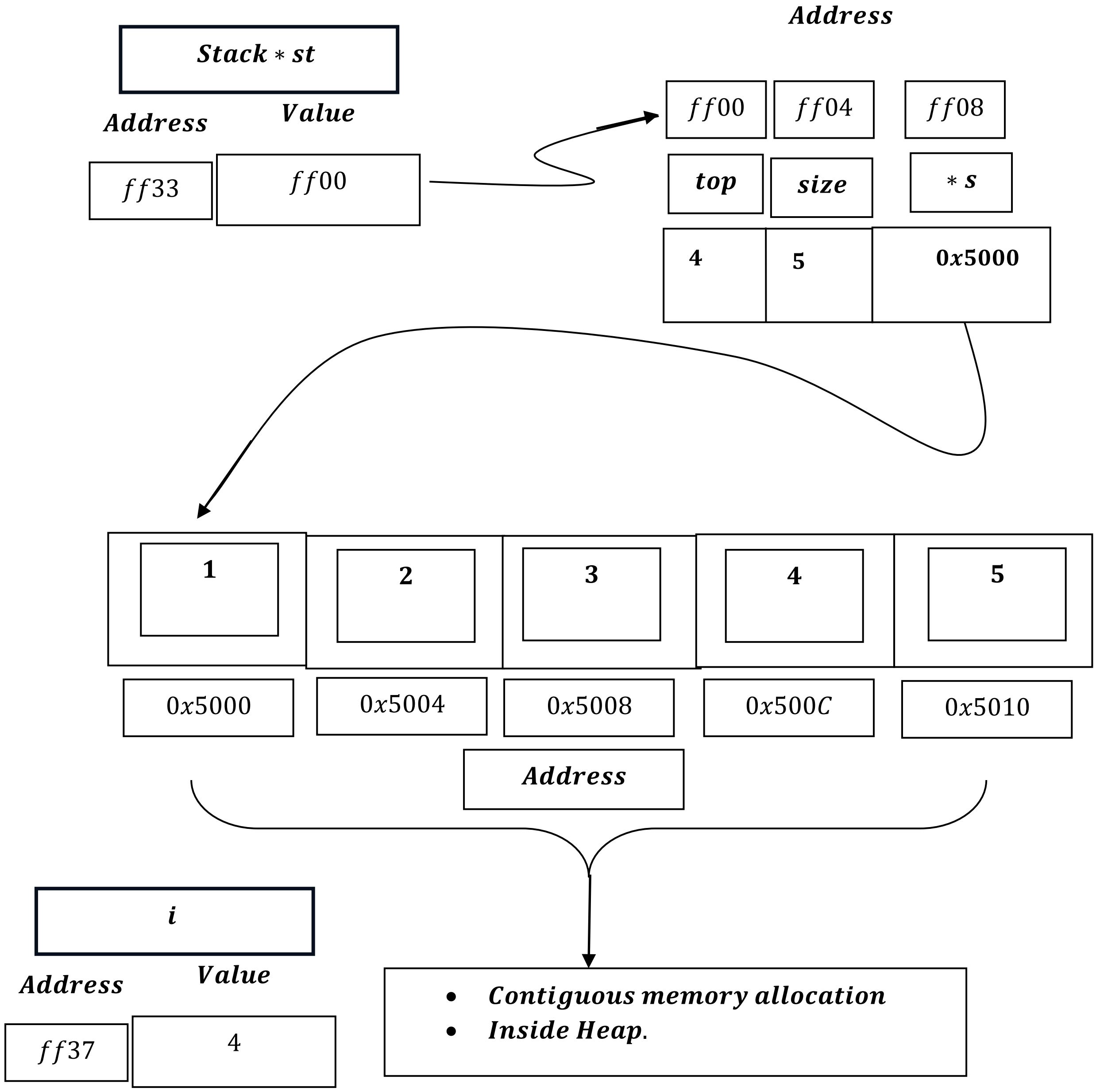
```
void stackTraversal(Stack st)
{

    if (isEmpty(st))
    {
        cout << "Stack is Empty" << endl;
    }

    for (int i = st.top; i >= 0; i--)
    {
        cout << st.s[i] << " ";
    }
    cout << endl;
}
…
       case 4:
          cout << "The elements in the stack are: " << endl;
            stackTraversal(stck);
            break;
```

isEmpty() function return 1 or 0 , when return 0

is false , when return 1 its true , if true then it print ``Stack is Empty`` else if return `0` , its false, then :

it traverse the stack and print the elements.





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

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

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

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

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

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

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

$\Rightarrow return\ 5$

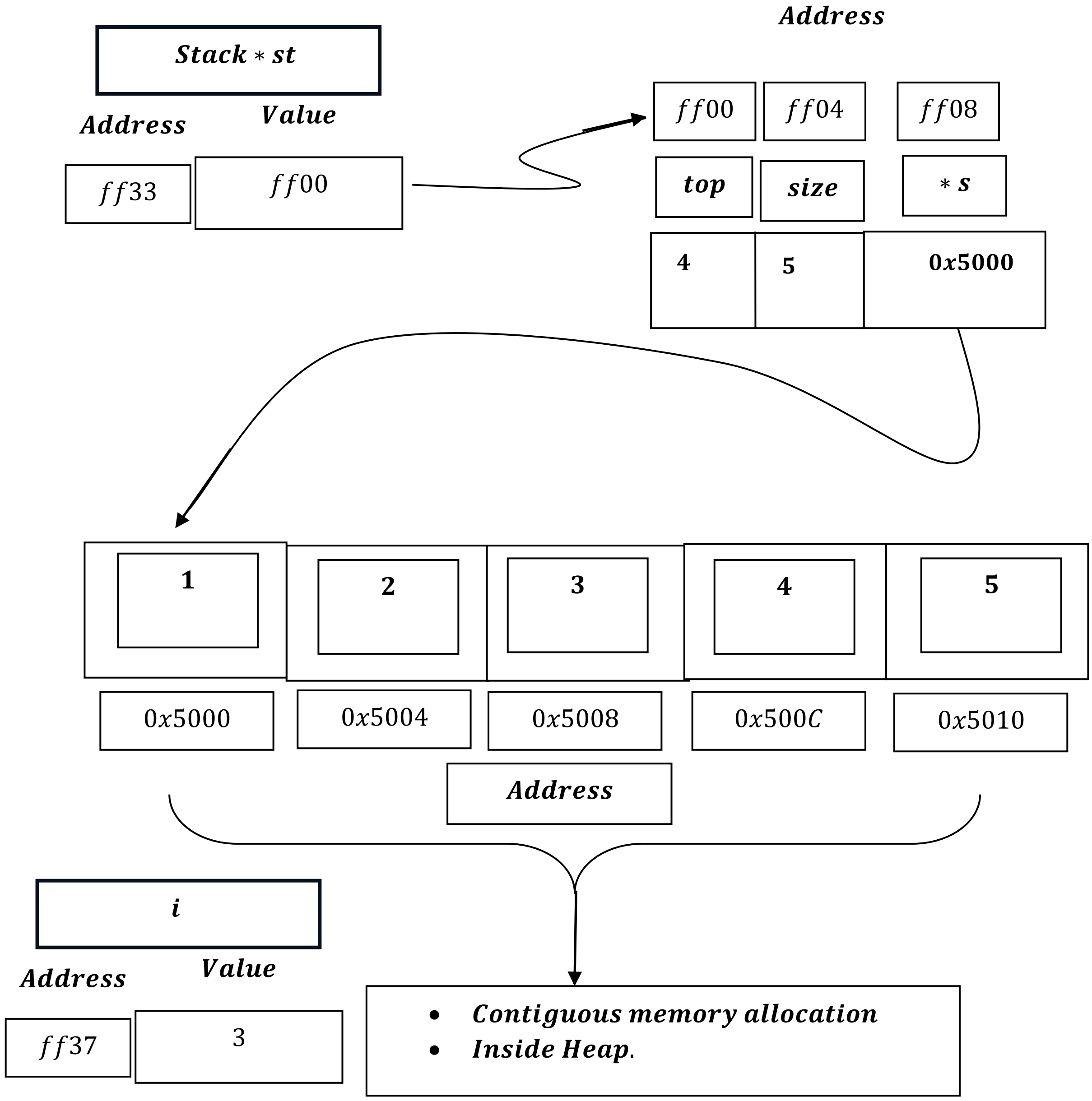
$Now\ \ i\ \ becomes : 3\ , from : i - - .$

$= i = i - 1.$

$= i = 4 - 1.$

$= i = 3.$

<u>i is it in post decrement.</u>



$\boldsymbol{st \to s[i]: \Rightarrow s[3].}$

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

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

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

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

$\boldsymbol{i.e. return\ value\ stored\ in\ 0x500C}$

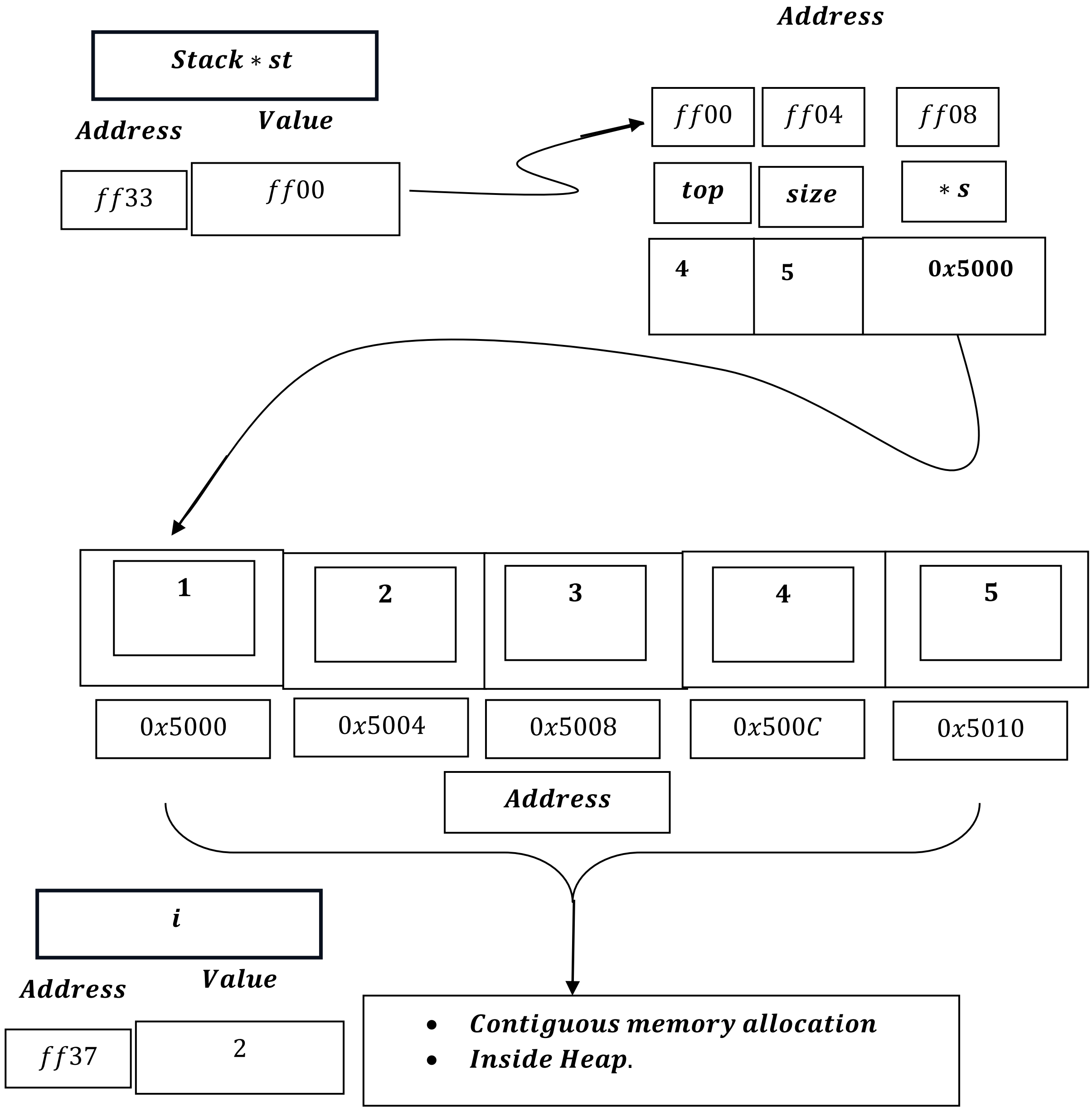
$\boldsymbol{\Rightarrow return\ 4}$

$\boldsymbol{Now\ \ i\ \ becomes : 2\ , from : i - - .}$

$\boldsymbol{= i = i - 1.}$

$\boldsymbol{= i = 3 - 1.}$

$\boldsymbol{= i = 2.}$



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

$st \rightarrow s[i]: \Longrightarrow s[2].$

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

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

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

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

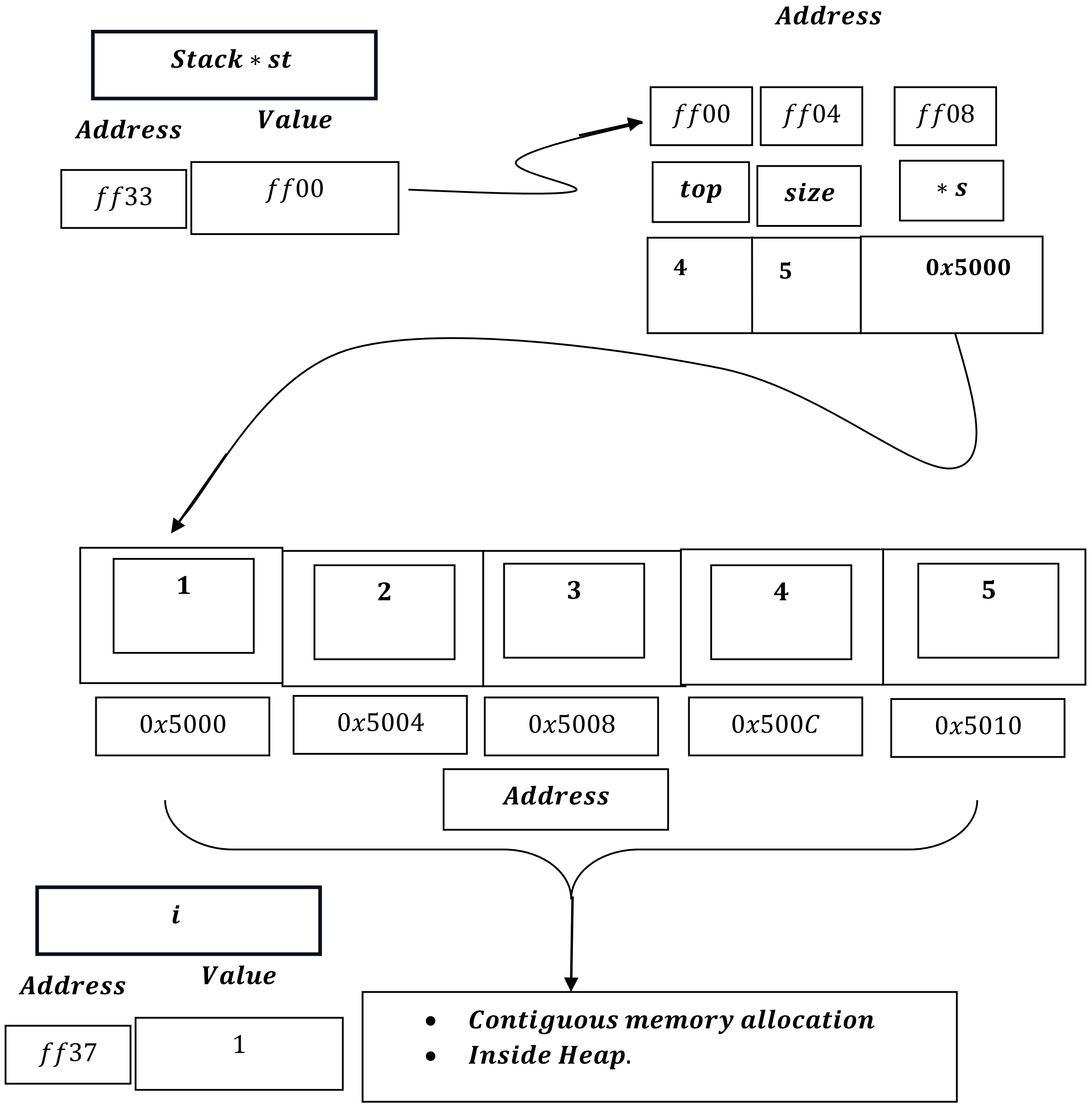
$\Longrightarrow return\ 3$

$Now\ \ i\ \ becomes : 1\ , from : i - - .$

$= i = i - 1.$

$= i = 2 - 1.$

$= i = 1.$



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

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

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

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

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

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

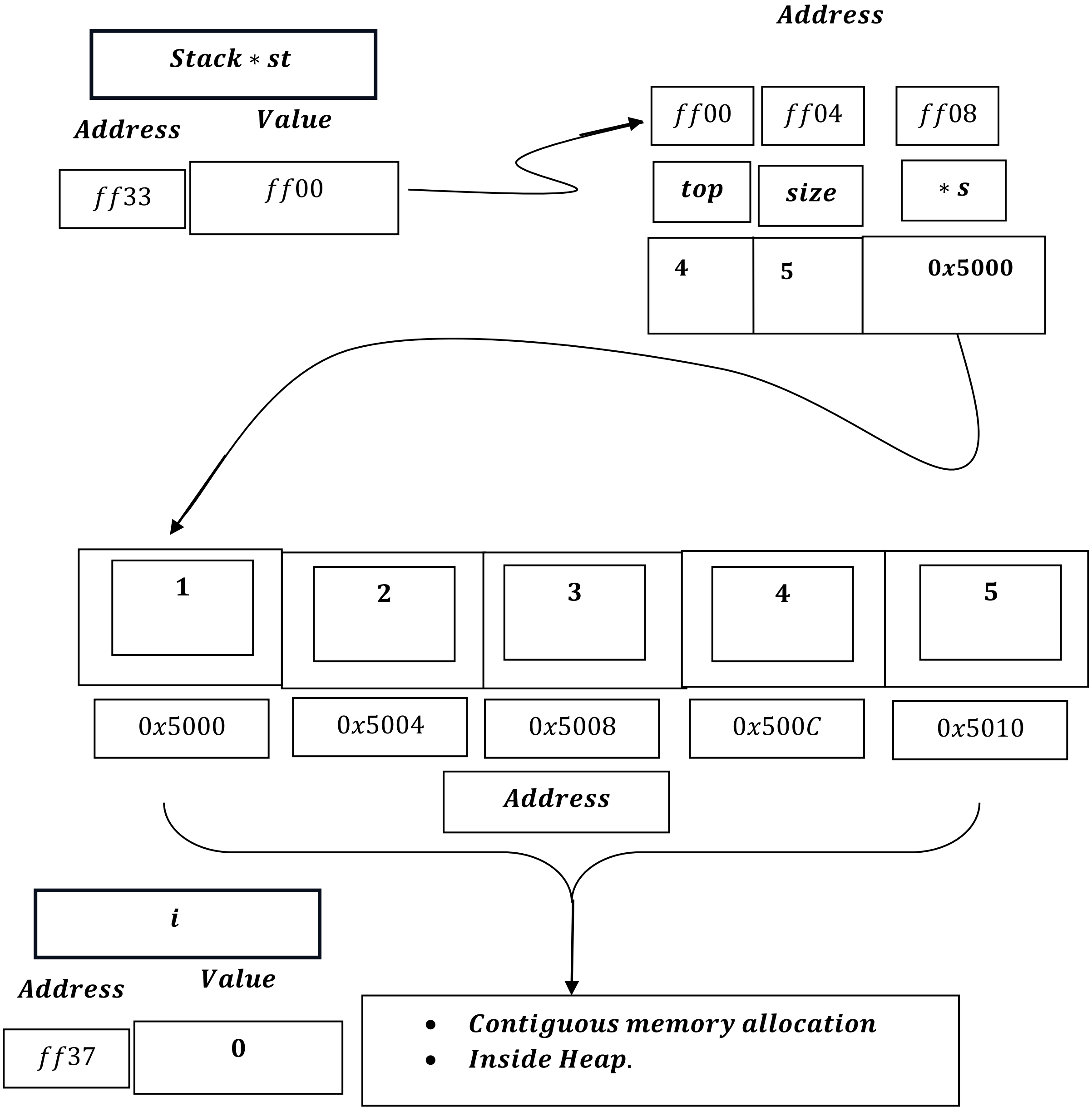
$\Rightarrow return\ 2$

$\boldsymbol{Now\ \ i\ \ becomes : 0\ , from : i--.}$

$\boldsymbol{= i = i - 1}.$

$\boldsymbol{= i = 1 - 1}.$

$\boldsymbol{= i = 0}.$



$\boldsymbol{st \rightarrow s[i]: \Longrightarrow s[0].}$

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

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

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

$\boldsymbol{i.e. return\ value\ stored\ in\ 0x5000}$

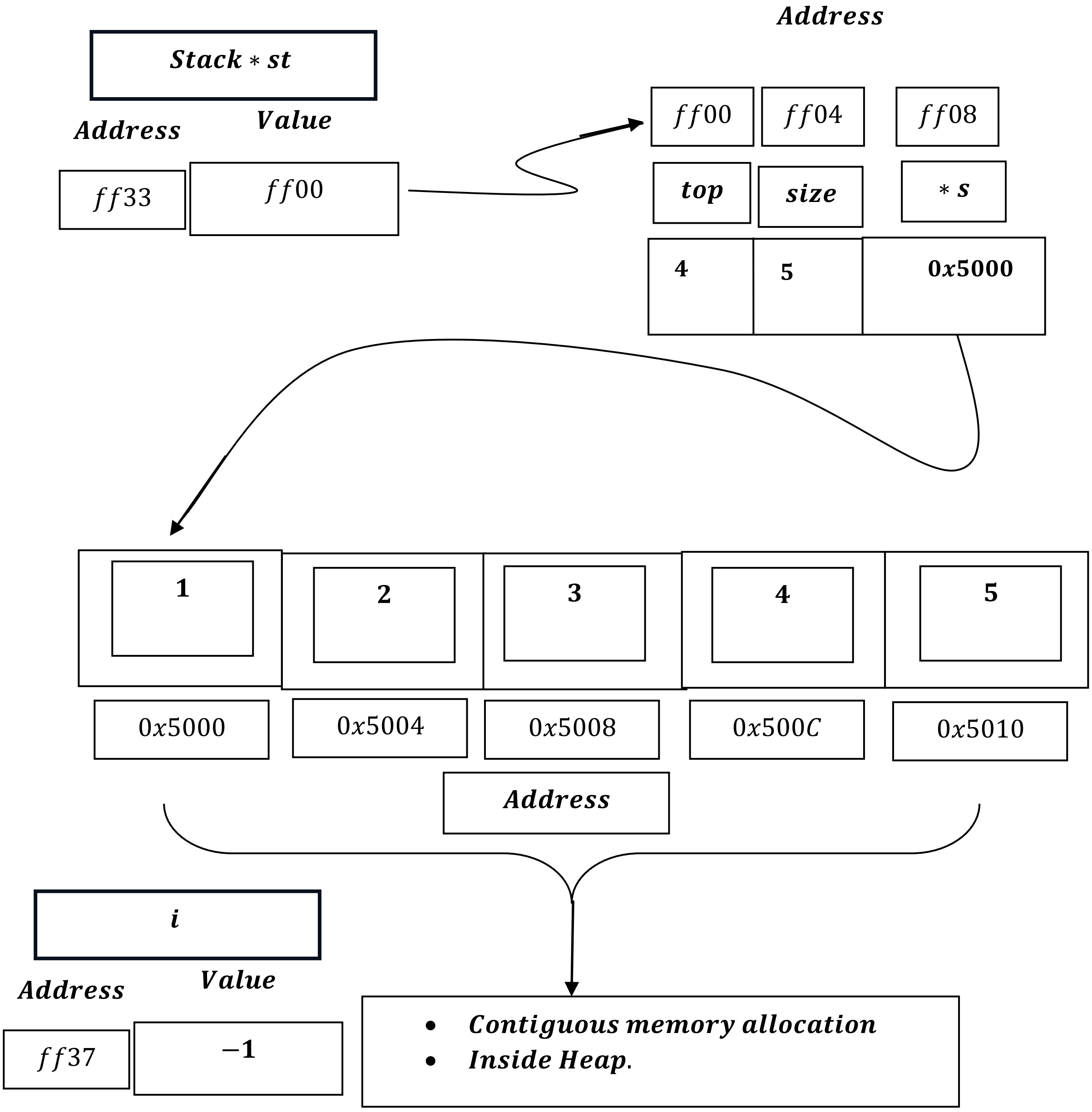
$\boldsymbol{\Longrightarrow return\ 1}$

$\boldsymbol{Now\ \ i\ \ becomes : -1\ , from : i - - .}$

$\boldsymbol{= i = i - 1.}$

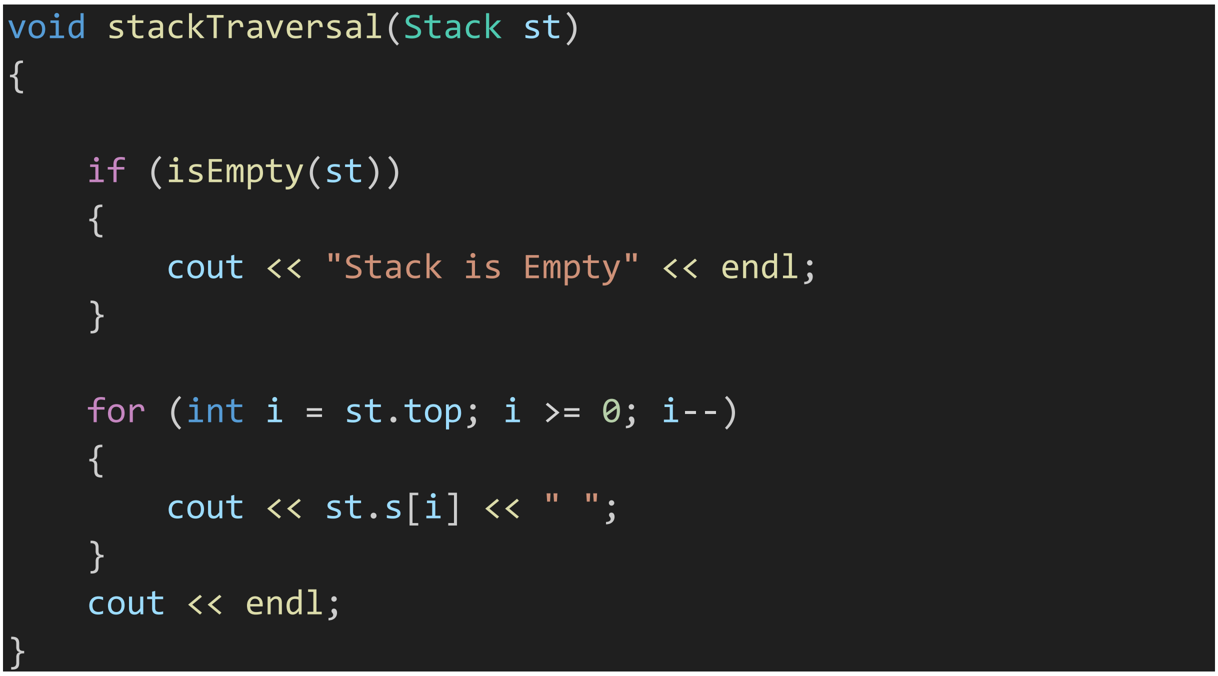
$\boldsymbol{= i = 0 - 1.}$

$\boldsymbol{= i = -1.}$



When $i = -1$, loop condition fails , hence return void and exit from the stack traversal function .

Time Complexity



```
void stackTraversal(Stack st)
{

    if (isEmpty(st))
    {
        cout << "Stack is Empty" << endl;
    }

    for (int i = st.top; i >= 0; i--)
    {
        cout << st.s[i] << " ";
    }
    cout << endl;
}
```

1. *Function overhead due to function call which includes creation of stack frame for the function stackTraversal() takes constant amount of time* : $O(1)$.

2. *isEmpty() function return* 1 *or* 0 , *when return* 0 *is false* , *when return* 1 *its true* ,
if condition its become true i.e. stack is empty.
[*if condition check takes constant time* : $O(1)$.]

Then inner statement runs:

→ *Print ``Stack is Empty``.* → *Takes constant time* $O(1)$.

$\mathbf{3.}\,\boldsymbol{If\ stack\ is\ not\ empty, stack\ traversal\ occurs}$:

$\rightarrow$ $\boldsymbol{loop\ runs\ from\ i = \text{`}n-1\text{`}\ to\ \text{`}0\text{`}\ [where\ \text{`}n\text{`}\ is\ size]}$:

$\boldsymbol{print\ \ value\ stored\ at\ s[i]. \rightarrow Runs}$

$\boldsymbol{1\ unit\ of\ time\ +1\ unit\ of\ time\ +\cdots n\ to\ times}$

$\boldsymbol{Note : \left[0\ to\ n-1\ \approx 1\ to\ n[resequenced]\right]}$

$\boldsymbol{Hence : O(n).}$

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

$\boldsymbol{When\ stackTraversal()\ \ is\ called}$:

1. $\boldsymbol{A\ stack\ frame\ (activation\ record)\ is\ created.}$
2. $\boldsymbol{The\ return\ address\ is\ stored.}$
3. $\boldsymbol{Parameters\ are\ handled.}$
4. $\boldsymbol{Control\ jumps\ to\ the\ function.}$
5. $\boldsymbol{After\ execution, the\ stack\ frame\ is\ removed.}$

$\boldsymbol{These\ steps}$:

- $\boldsymbol{Do\ not\ depend\ on\ input\ size\ (n).}$
- $\boldsymbol{Take\ a\ fixed\ number\ of\ CPU\ instructions.}$

$\boldsymbol{Hence, Function\ Call\ Overhead\ =\ O(1).}$

The function call overhead, including stack frame creation and destruction for stackTraversal(), takes constant time $O(1)$, since it does not depend on the input size.

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.
