Developing Technical Software

Week 5:
Revisit Dynamic data structures: Linked Lists
Dynamic data structures: Stacks

Assignment 1 Reminder

- Assignment 1 contains 4 questions (Q1 is for 8 marks, Q2 is for 6 marks, Q3 is for 5.5 marks, and Q4 is for 5.5 marks)
- Pseudo code is mandatory
- Submit only one document either in .doc or .docx
- See marking criteria on Canvas
- Due date is 14th November 2018 23:59hrs

What is a Pseudo Code?

- It is an informal way of programming description that does not require any strict programming language syntax or underpinned technology considerations
- It is a simple way of writing programming code in English and uses short phrases to write code (before you actually code it in C or any program language)

Examples of a Pseudo Code

-
  ```
  Start Program1
  Enter three numbers, A, B, C
  Add the numbers together
  Print Sum
  End Program
  ```

-
  ```
  Start Program2
  Enter length l, width w and
  Calculate Perimeter = 2*l + 2* w
  Print Perimeter
  End Program
  ```

Revisit: Dynamic Memory Allocation

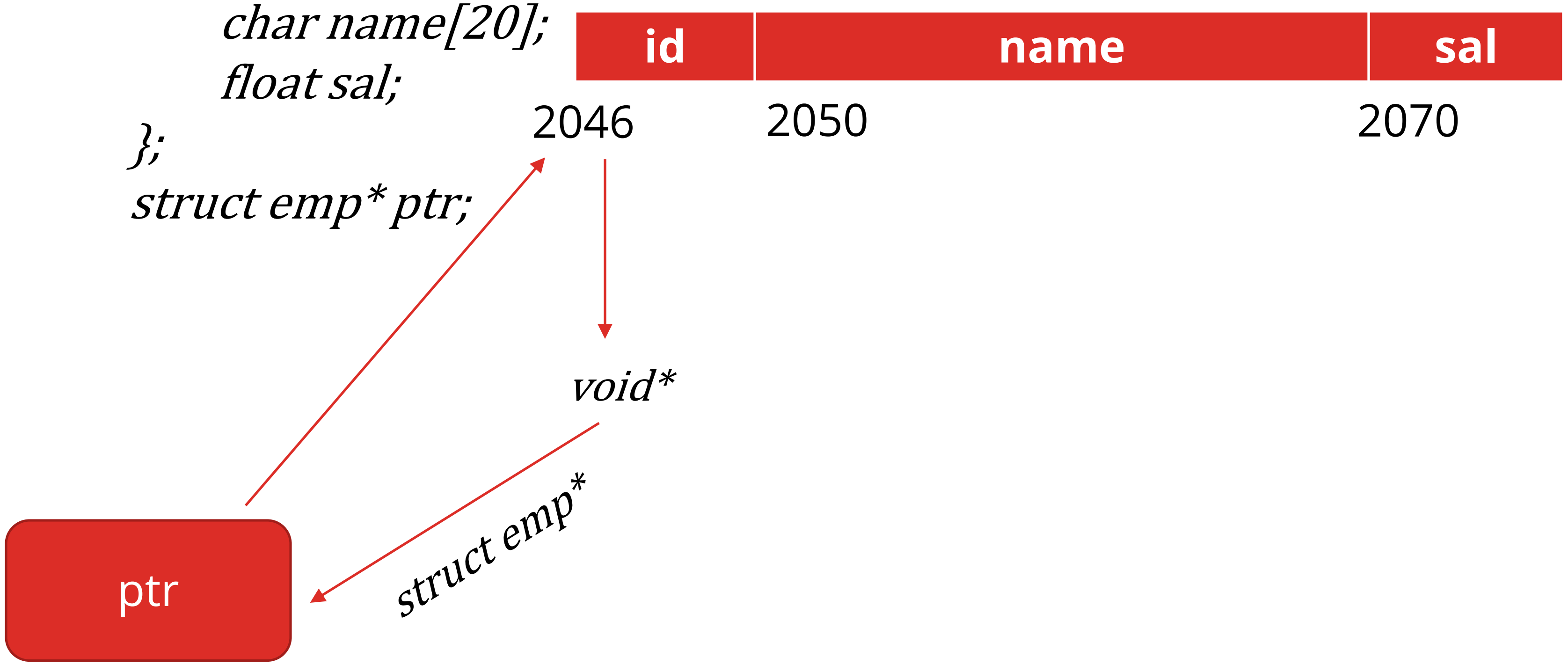
- Static memory allocation means fixed size
- Dynamic memory allocation means the size can be increased or decreased based on our requirement
- To allocate memory dynamically, we need to take the help of predefined functions available in <stdlib.h> library

malloc() – allocate memory to structures

- void* malloc(size_t size);
- Void is the return type and a generic pointer which means it can point to any datatype such as int, char etc
- Size_t is an unsigned positive integer
- Size is the size of the memory block needed
- On successful allocation, it returns a base address of memory block
- On failure, it returns a null pointer

malloc() – Example

```
struct emp{
        int id;
        char name[20];
        float sal;
};
struct emp* ptr;
```



```
ptr = (struct emp*) malloc (sizeof(struct emp));
```

Data Structures

- A way of organizing and storing data in order to efficiently manage
- Its not a programming language
- A data structure is a set of algorithms that can be used to structure the information
- To structure data, many number of algorithms are proposed and can be referred to as abstract data types (set of instructions)

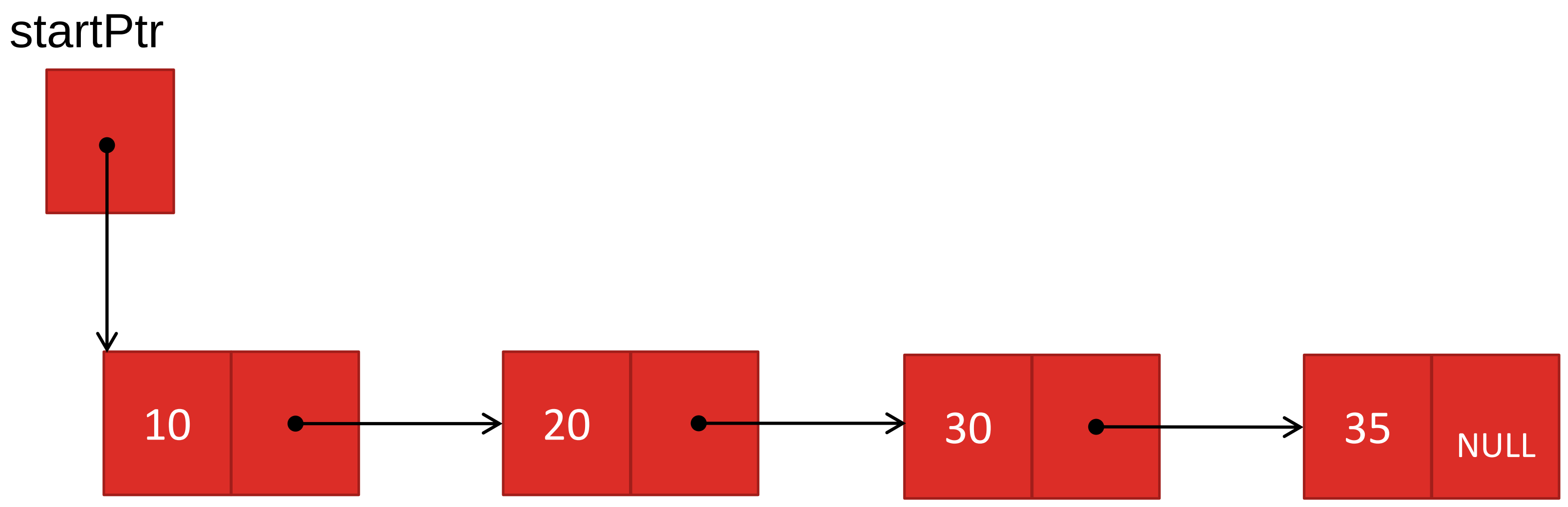
What is a Linked list?

- a linear collection of self-referential structures, called nodes(memory blocks), connected by pointer links
- accessed via a pointer to the 1st node of the list
- Subsequent nodes are accessed via the link pointer member stored in each node.
- the link pointer in the last node of a list is set to NULL (as the end of the list)

More…

- a linear **dynamic** collection
- It is impossible to create a static linked list
- Not possible to fix a limit for linked list size
- In linked lists, insertions and deletions are much faster than arrays
- No direct access to elements of the linked list but only traversing or travelling through all nodes

A linked list with several nodes



To begin with …

```
#include<stdio.h>
#include<stdlib.h>
/* This is the node structure*/
struct node {
  int data;
  struct node *next;
} *start = NULL;

void create(void);
void display(void);

int main()
{
        create();
        display();
        return 0;
}
```

To create node

```
void create()
{
        char ch;
        do{
                struct node *new_node, *current;
                new_node = (struct node *) malloc(sizeof(struct node));
                printf(" \nEnter the data : ");
                scanf("%d", &new_node -> data);
                new_node -> next = NULL;
                if (start == NULL) {
                        start = new_node;
                        current = new_node;
                }
                else {
                        current -> next = new_node;
                        current = new_node;
                }
                printf(“\nAnother node? (y/n): ");
                ch = getche();
        } while (ch != 'n');
}
```

To display nodes

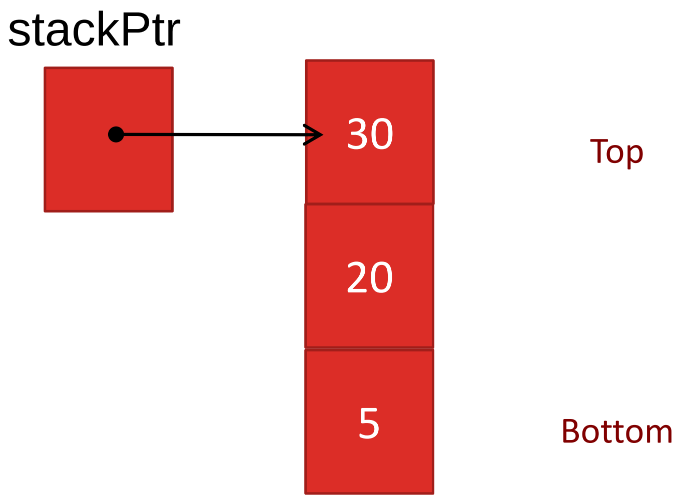
```
void display()
{
     struct node *new_node;
     printf(" \nThe Linked List : ");
     new_node = start;
     while (new_node != NULL){
          printf("%d -> ", new_node-
>data);
          new_node = new_node -> next;
     }
     printf("NULL");
}
```

Output

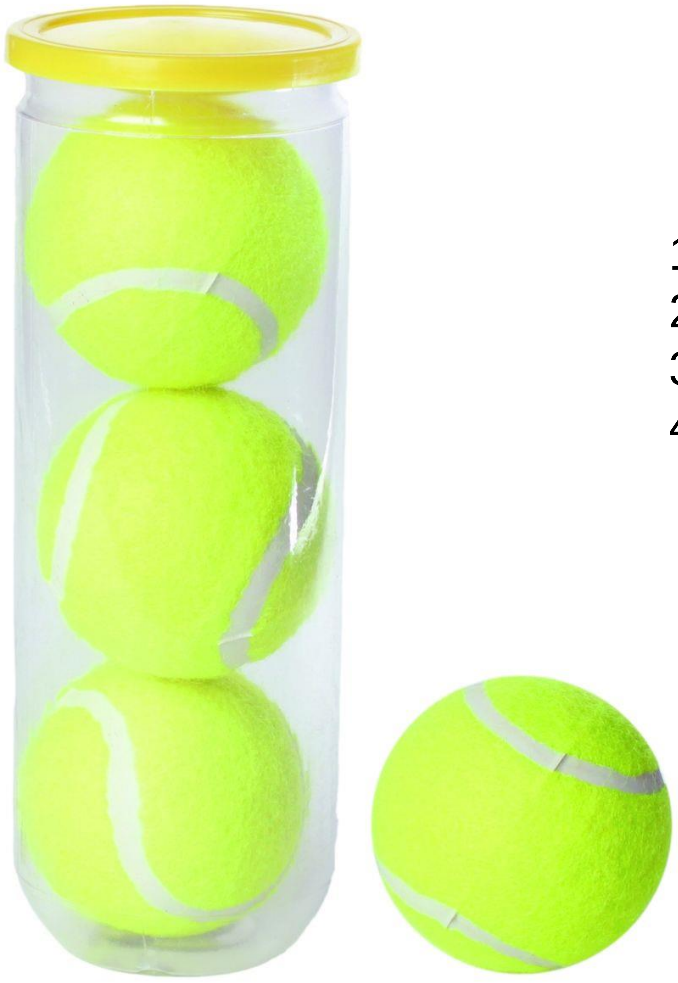
```
•Enter the data : 10
•Another node? (y/n):  y
•Enter the data : 20
•Another node? (y/n): y
•Enter the data : 30
•Another node? (y/n): n
•The Linked List :
•10 -> 20 -> 30 -> NULL
```

What is a stack?

- A stack is a **last-in**, **first-out** (**LIFO**) data structure.
- New nodes can be added to a stack and removed from a stack *only* at the *top*.
- Following figure illustrates a stack with several nodes.



An example of stack?



1. Which ball went in first?
2. Which ball went in last?
3. Which ball can be taken out first?
4. How can second ball be taken out?

Image source: http://clipartbarn.com/tennis-ball-clip-

Another example of stack?

1. Which CD went in first?
2. Which CD went in last?
3. Which CD can be taken out first?
4. How can second CD be taken out?

Image source:
https://rocketdock.com/addon/icons/30195

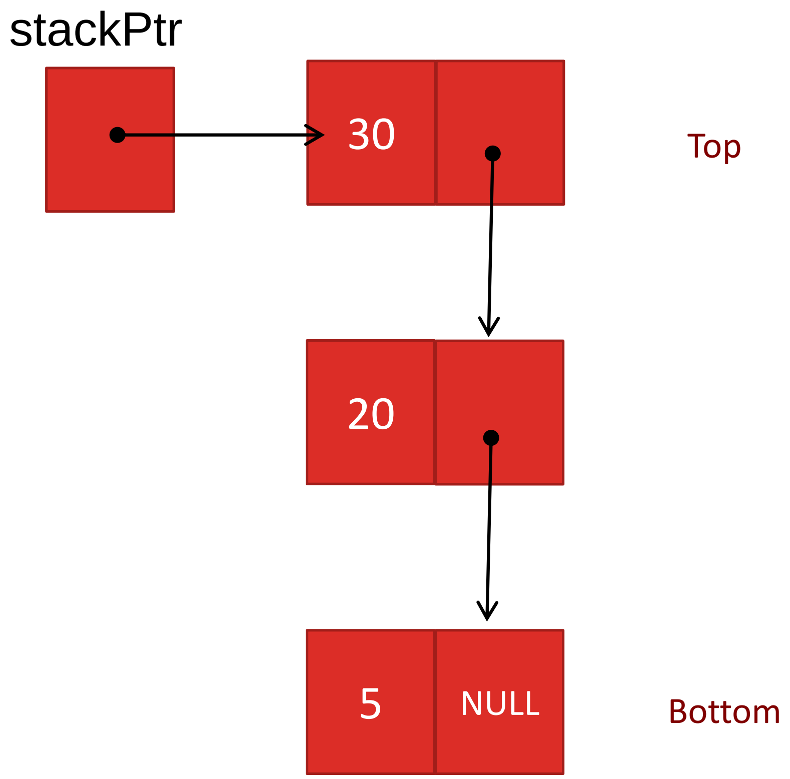
Implementing a stack

- A stack can be implemented as a constrained version of a linked list.
- A stack is referenced via a pointer to the top element of the stack.
- The link member in the last node of the stack is set to NULL to indicate the bottom of the stack.
- Not doing so can lead to runtime errors.

Stack vs. linked-list

- Stacks and linked lists are represented identically. The difference between stacks and linked lists is that insertions and deletions may occur *anywhere* in a linked list, but only at the *top* of a stack.
- Following figure illustrates a stack with several nodes, and stackPtr points to the stack's top element.

Stack



Stack functions

- The primary functions used to manipulate a stack are push and pop.
- Function push creates a new node and places it on *top* of the stack.
- Function pop *removes* a node from the *top* of the stack, *frees* the memory that was allocated to the popped node and *returns the popped value*.

C Program for stack functions

stack.c implements a simple stack of integers.
The program provides three options:

1) push a value onto the stack (function push)

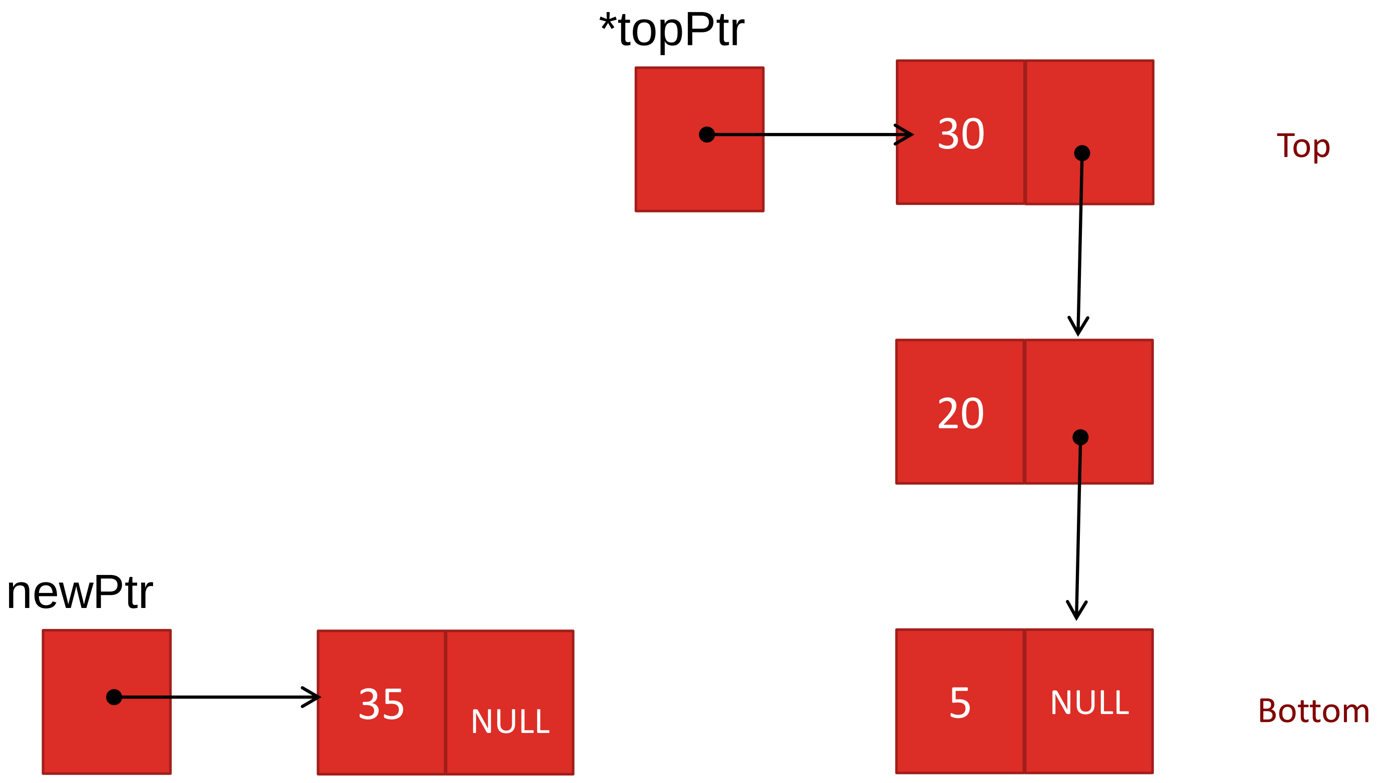
2) pop a value off the stack (function pop)

3) terminate the program.

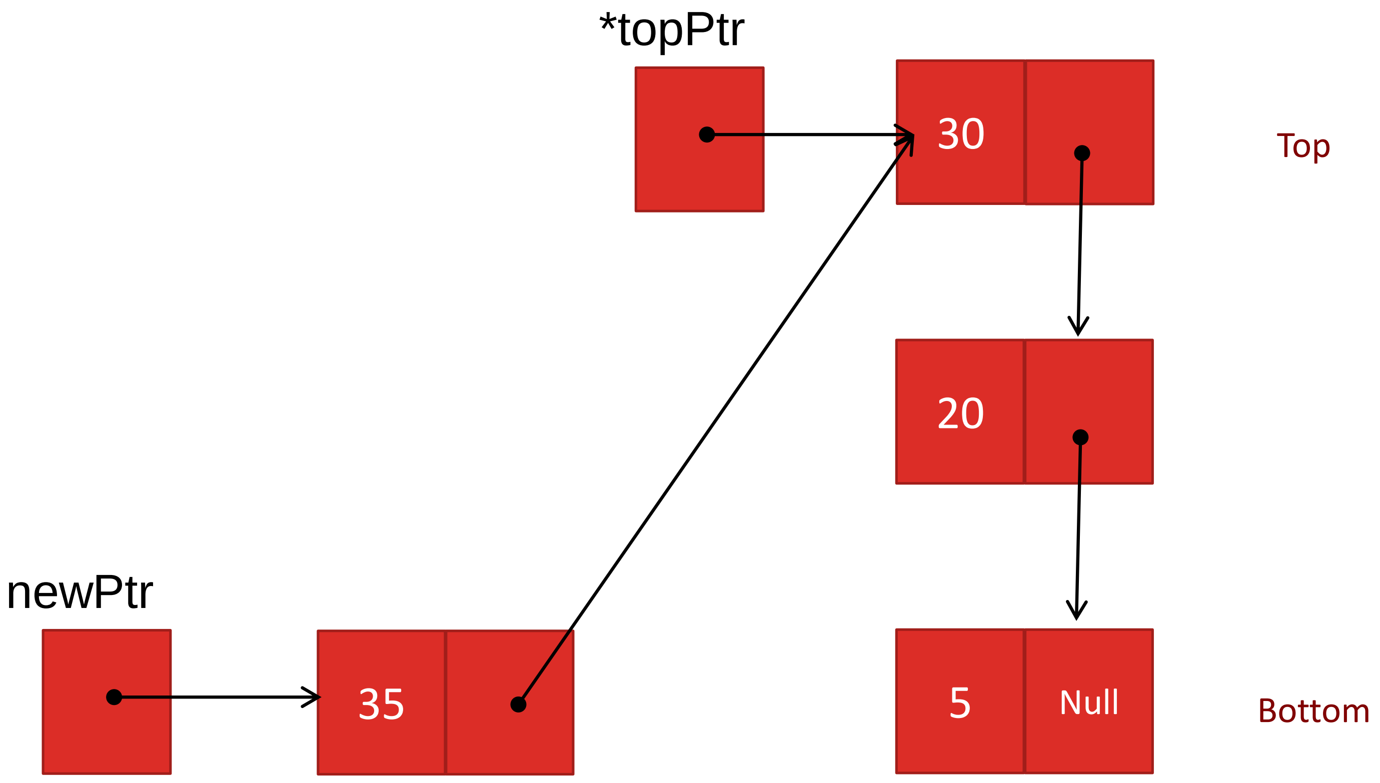
Function push

- Function push places a new node at the top of the stack.
- The function consists of three steps:
    1. Create a *new node* by calling malloc and assign the location of the allocated memory to newPtr.
    2. Assign to newPtr->data the value to be placed on the stack and assign *topPtr (the *stack top pointer*) to newPtr->nextPtr – the link member of newPtr now points to the *previous* top node.
    3. Assign newPtr to *topPtr – *topPtr now points to the *new* stack top.
- Manipulations involving *topPtr change the value of stackPtr in main.

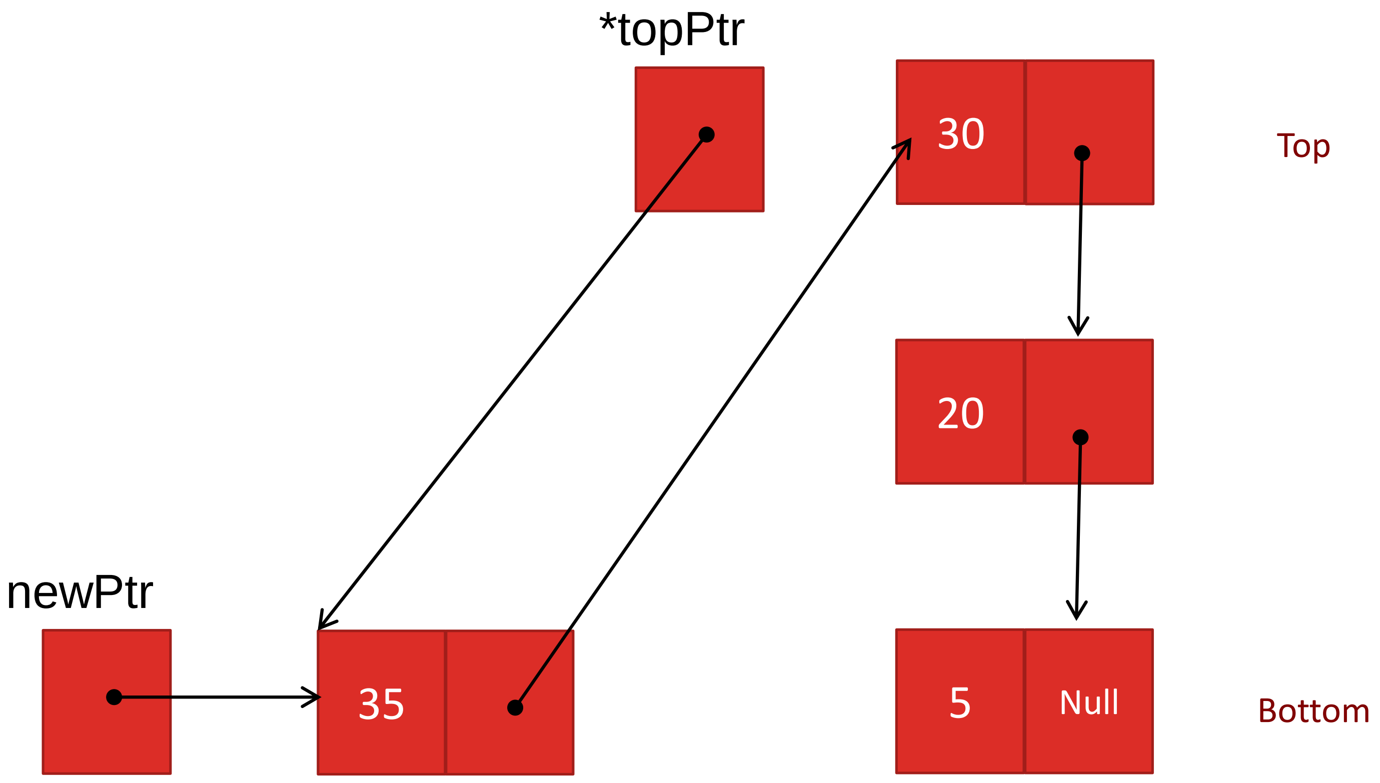
Push (Step 1)



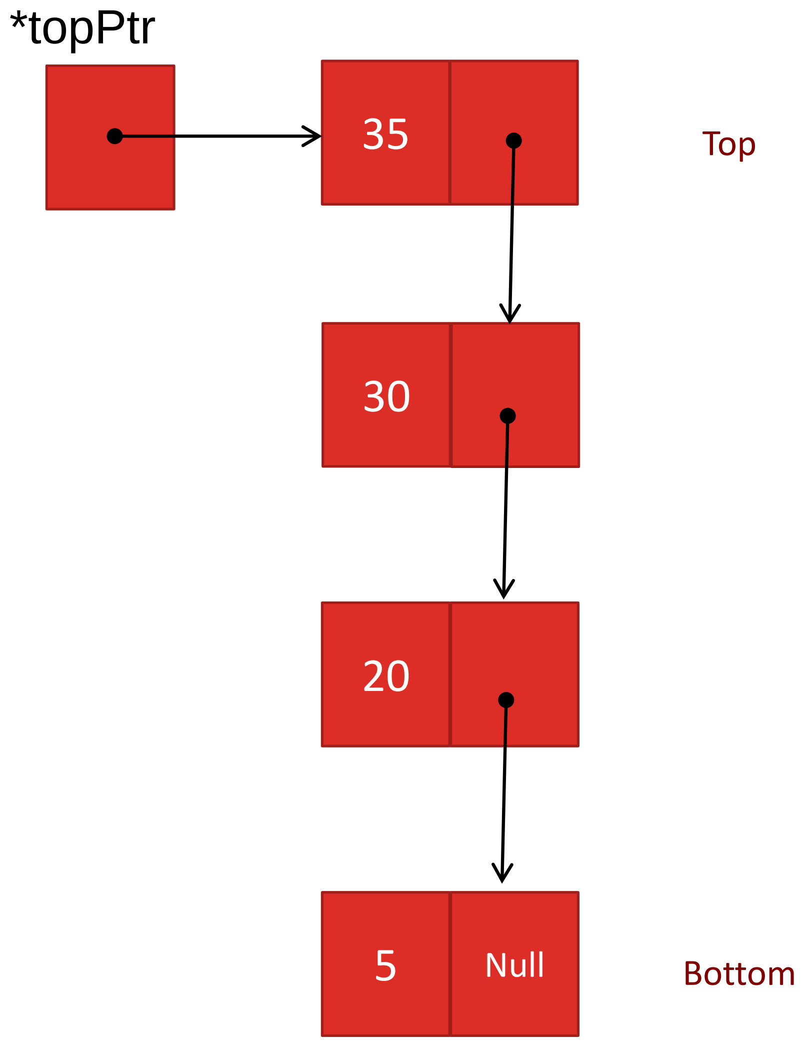
Push (Step 2)



Push (Step 3)



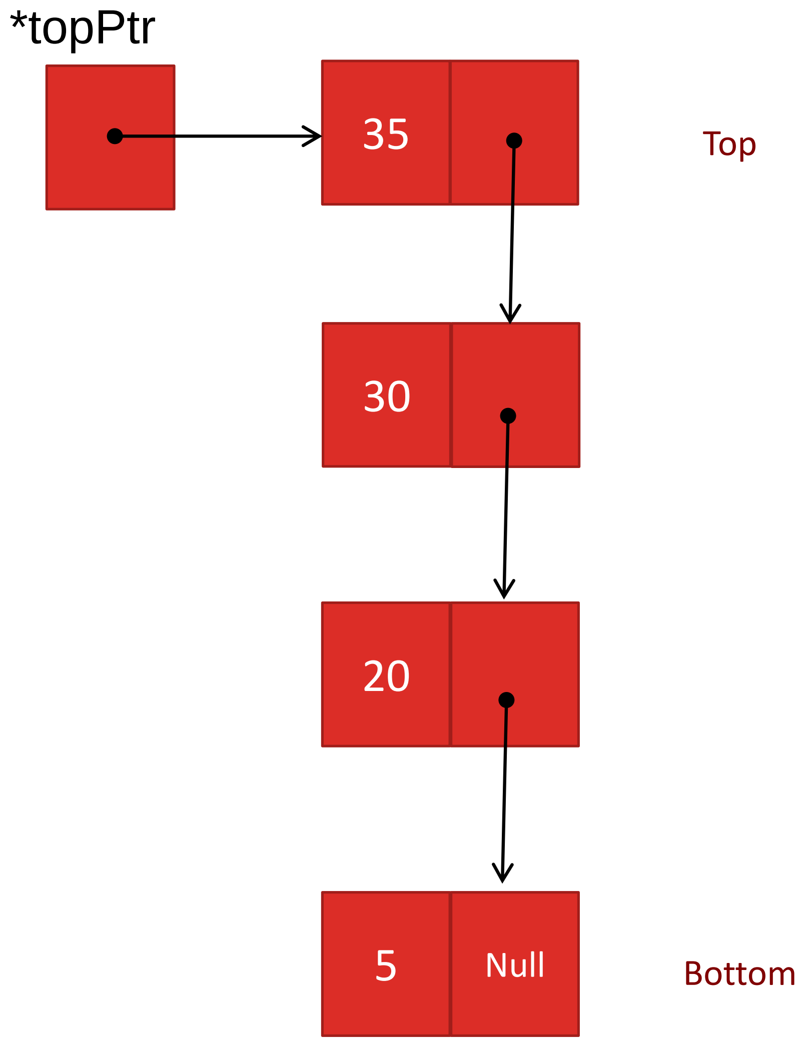
Stack (after Push)



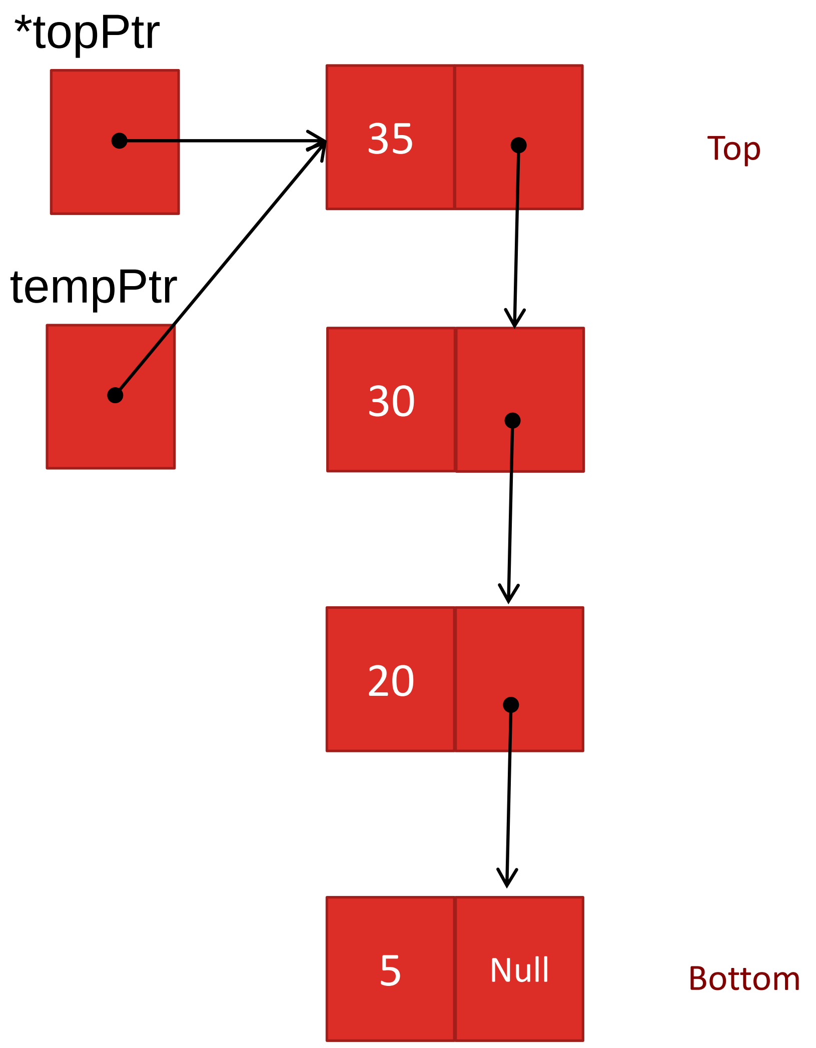
Function pop

- Function pop removes a node from the top of the stack.
- Function main determines whether the stack is empty before calling pop.
- The pop operation consists of five steps:
  1. Assign *topPtr to tempPtr; tempPtr will be used to *free* the unneeded memory.
  2. Assign (*topPtr) ->data to popValue to *save* the value in the top node.
  3. Assign (*topPtr)->nextPtr to *topPtr so *topPtr contains *address of the new top node*.
  4. *Free the memory* pointed to by tempPtr. Function **free** is used to *free the memory* pointed to by tempPtr
  5. *Return popValue* to the caller.

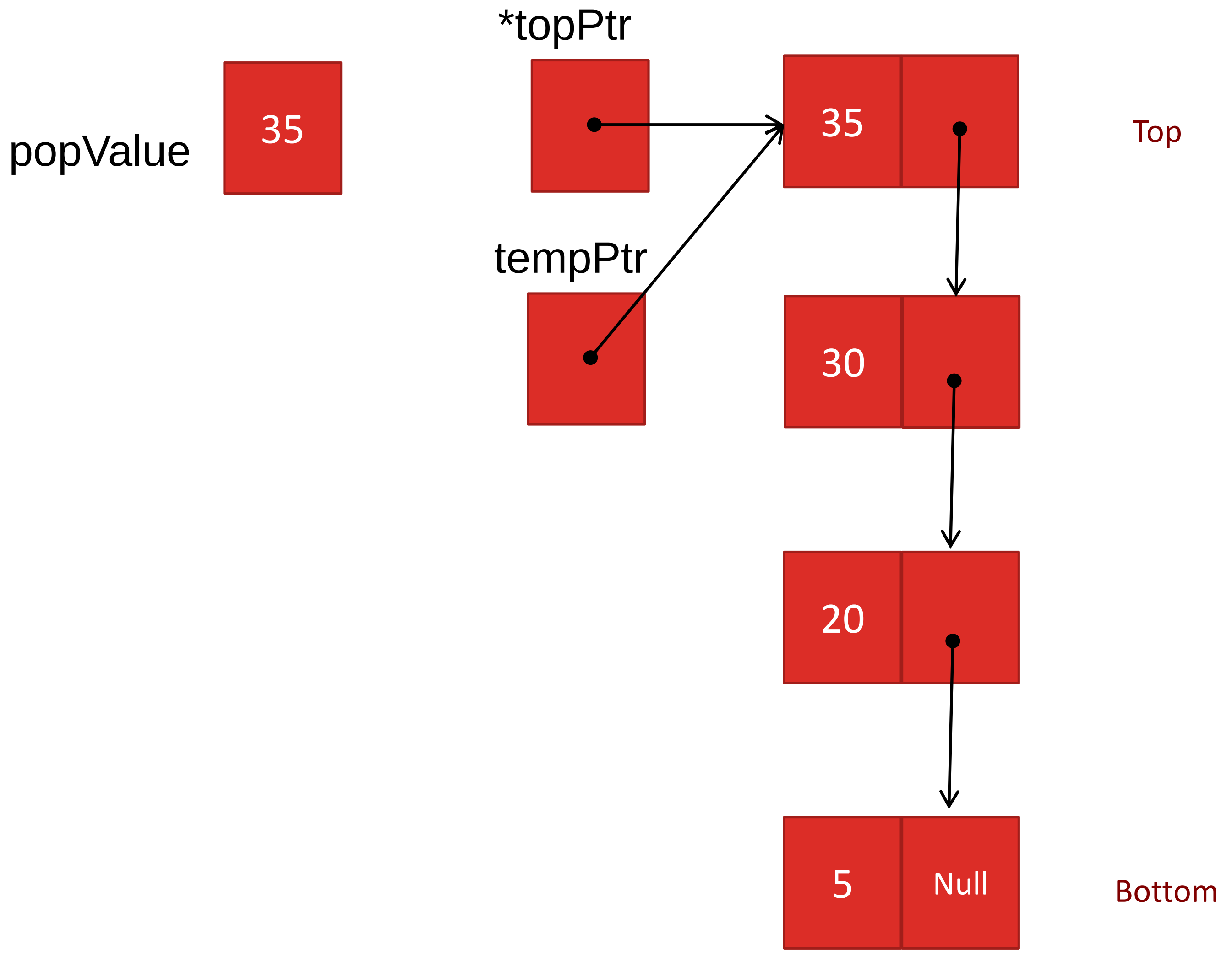
Pop (Step 0)



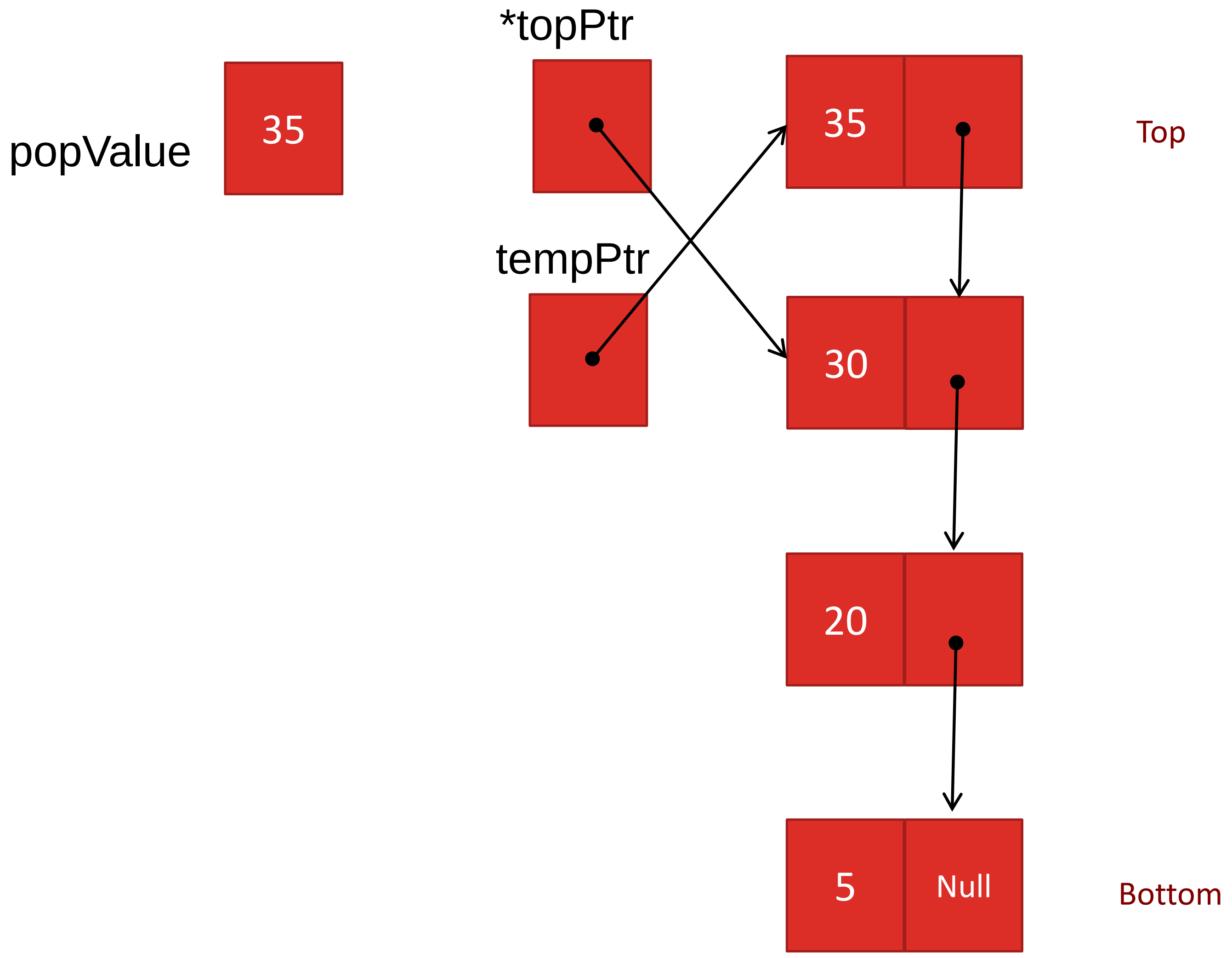
Pop (Step 1)



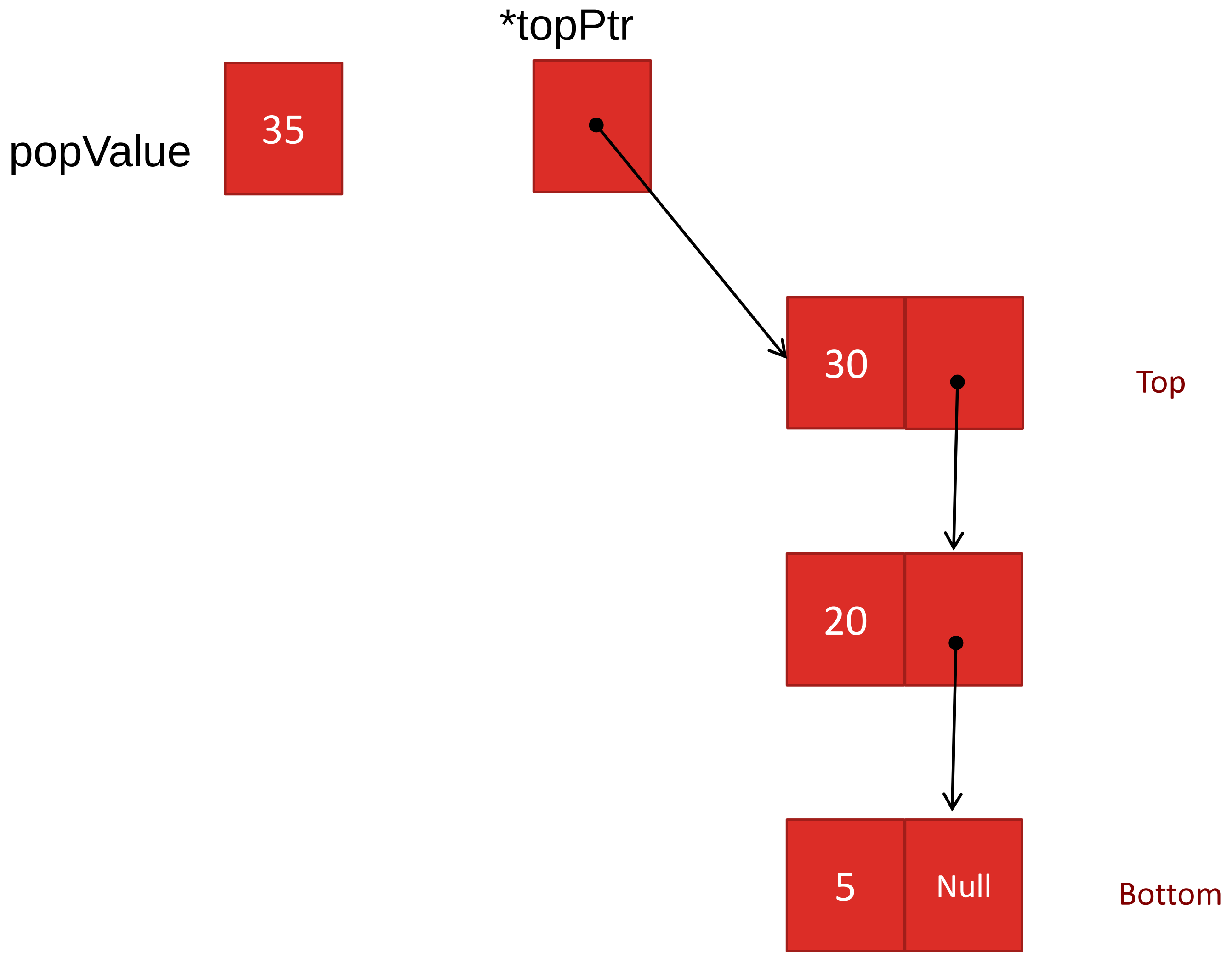
Pop (Step 2)



Pop (Step 3)



Pop (Step 4)



Recursion

- A **recursive function** is a function that *calls itself* either directly or indirectly through another function.
- We may consider solving a problem by recursive function call if
  - The *simplest case* of the problem could be easily solved (**base case**)
  - The problem could be represented by a *similar but simpler* version of the original problem (**recursive call**)
- **Example**: Recursively calculating factorials

```
long fact(int n)
{
     if  (n<=1) return 1;
     else return (n*fact(n-1));
}
```

Applications of Stacks (1)

- Whenever a *function call* is made, the called function must know how to *return* to its caller, so the *return address* is pushed onto a stack.
- If a series of function calls occurs, the successive return values are pushed onto port recursive function calls in the same manner as conventional non-recursive calls.

Applications of Stacks (2)

- Stacks contain the space created for *automatic variables* on each invocation of a function.
- When the function returns to its caller, the space for that function's automatic variables is popped off the stack, and these variables no longer are known to the program.

Applications of Stacks (3)

- Stacks are used by compilers in the process of evaluating expressions and generating machine-language code.
- The exercises explore several applications of stacks.