Benha University

Faculty Of Engineering (Shoubra)

Communication and Computer Department

Recursion using Stack by LinkedList

Supervised By:

Dr. Lamiaa El Refaei

Eng. Shaimaa Yousry

Computer Organization CCE307

Benha University

Faculty of Engineering (Shoubra)

Communication and Computer Department

Course Project cover page

S#	Student Name	Edu Email	Student ID	Marks			
				Report & Slides (30)	Implementation (50)	Presentation (20)	Total (100)
1	Abdelrahaman Afify Hussien	abdulrahman402565@feng.bu.edu.eg	**221903086**				
2	Mohannad Mohamed Talaat	mohannad402754@feng.bu.edu.eg	**221903171**				
3	Abdullah Mohamed Galal	abdalah.mohamed21@feng.bu.edu.eg	**221903118**				
4	Malak Mounir Abdellatif	malak402682@feng.bu.edu.eg	**231903643**				
5	Nourhan Farag Mohamed	nourhan403033@feng.bu.edu.eg	**231903707**				

Date handed in: 5 / 5 / 2024.

Abstract

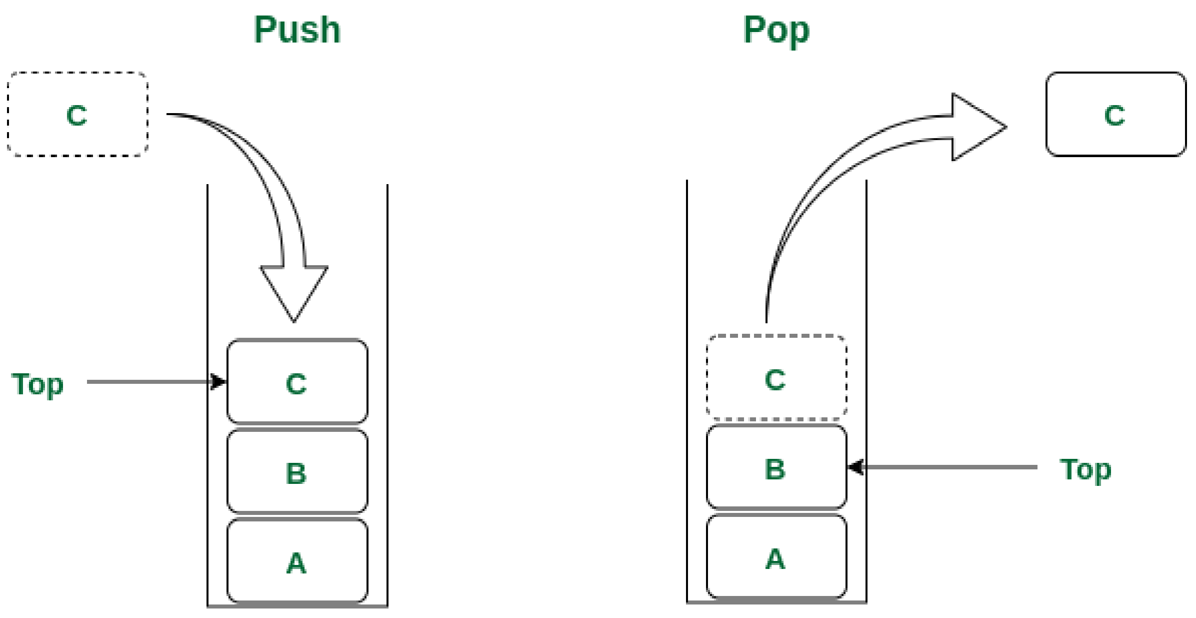
Recursion is a fundamental concept in computer science, wherein a function calls itself until a base condition is met. In this report, we explore the implementation of recursion using a stack based on a linked list, both in the C programming language and in assembly language. The stack, implemented using a linked list, facilitates managing function calls and their respective local variables. This report provides insights into the mechanics of recursion, stack-based implementation, Operations like push and pop in stacks are efficiently implemented through linked list manipulation, and the differences between C and assembly languages in handling recursion. Understanding these principles is essential for designing algorithms that leverage the stack effectively in various computational tasks.

Introduction

Recursion involves breaking down a problem into smaller, similar sub-problems until reaching a base case. This process often relies on function calls, which create a call stack to manage local variables and return addresses. Implementing recursion through a stack based on a linked list offers dynamic memory allocation and flexibility, fundamental aspects of computer organization.

The stack's Last-In-First-Out (LIFO) behavior aligns well with recursive operations. By calling a function within its own definition, a stack frame is created for each invocation, facilitating nested function calls.

This report explores the implementation of recursion using a stack-based approach on a linked list structure. It delves into the theoretical foundations, design considerations, and practical applications of this methodology. With these concepts, developers can enhance the functionality and usability of software applications, ultimately improving user satisfaction.



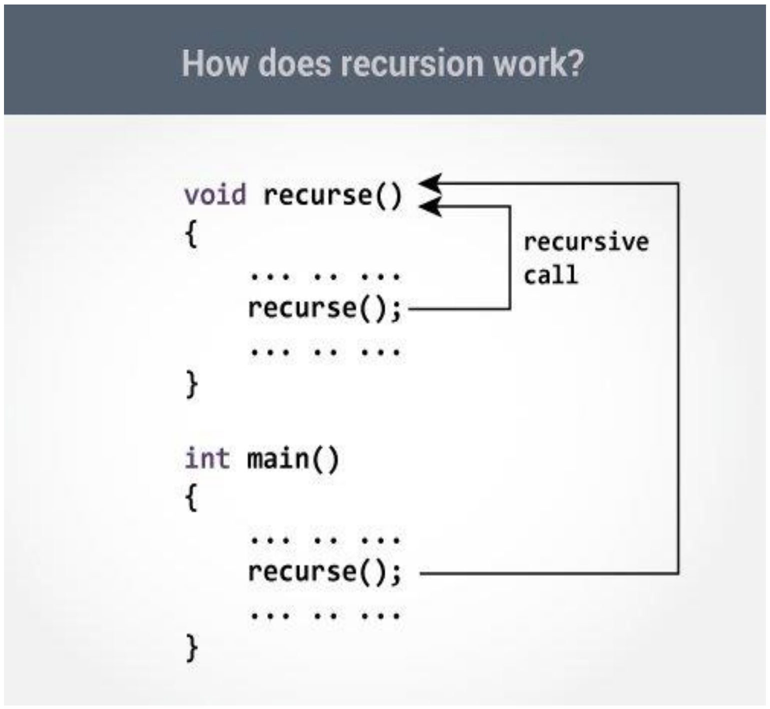
Stack Data Structure

Theory of Recursion:

At its core, recursion comprises two essential components: a base case and a recursive case. The base case represents the terminating condition that halts the recursive process, while the recursive case involves breaking down the problem into smaller instances of the same problem until reaching the base case. This iterative process forms a call stack, wherein each function call adds a new frame until the base case is encountered, triggering the unwinding phase.

Mechanics of Recursion:

Recursion relies on the call stack to manage function calls and their respective local variables. Each recursive call consumes memory space, necessitating careful consideration of memory management and potential stack overflow issues. Understanding stack frames, memory allocation, and the order of function execution is crucial for efficient recursion implementation.



```
void recurse()
{
    ... .. ...
    recurse();
    ... .. ...
}

int main()
{
    ... .. ...
    recurse();
    ... .. ...
}
```

Applications of Recursion:

Recursion finds widespread applications across various domains, including mathematics, computer science, and engineering. In mathematics, recursive sequences and functions play a pivotal role in solving problems related to sequences, series, and fractals. In computer science, recursion is used in algorithms such as tree traversal, sorting (e.g., quicksort), and graph traversal. Recursive techniques also feature prominently in data structures like linked lists, trees, and graphs.

Theoretical of Stack:

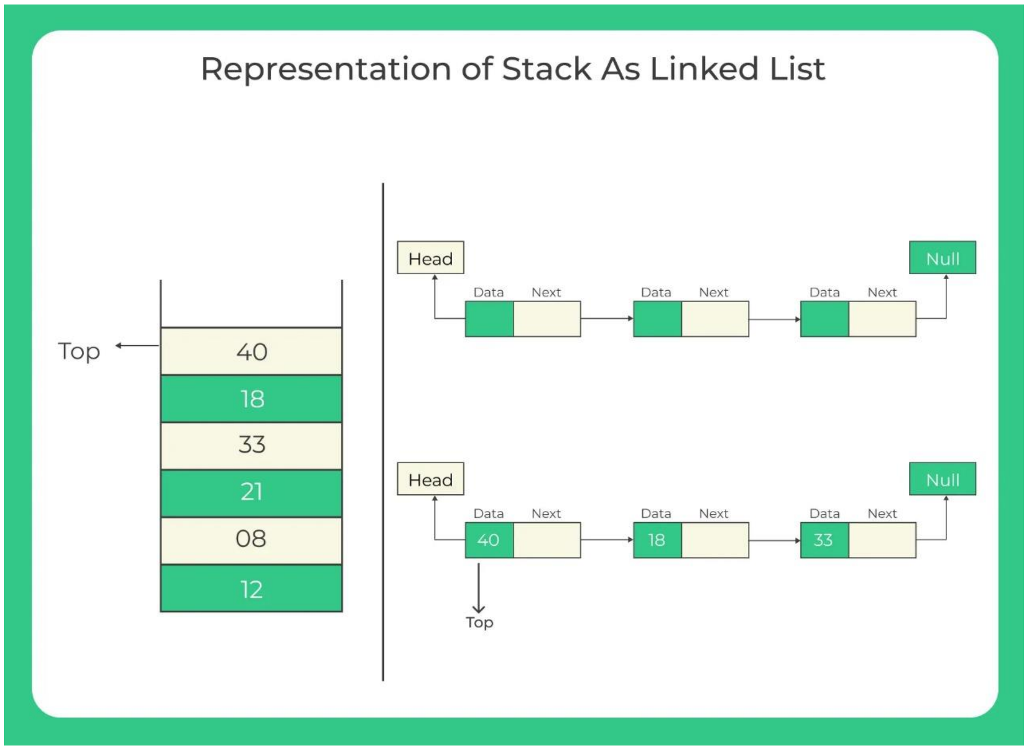
At its core, the stack comprises a contiguous block of memory organized into frames, each representing a function call or data item. Key operations on the stack include push (to add an item), pop (to remove the top item), and peek (to view the top item without removal). The stack pointer, a register or memory location, keeps track of the stack's topmost address.

Implementation Mechanisms:

The stack can be implemented using various data structures, with arrays and linked lists being the most common choices. In array-based implementation, a fixed-size array allocates memory for stack elements, while linked list-based implementation offers dynamic memory allocation and flexibility. Additionally, hardware support for the stack, such as the stack pointer register and stack frames, facilitates efficient stack management at the processor level.

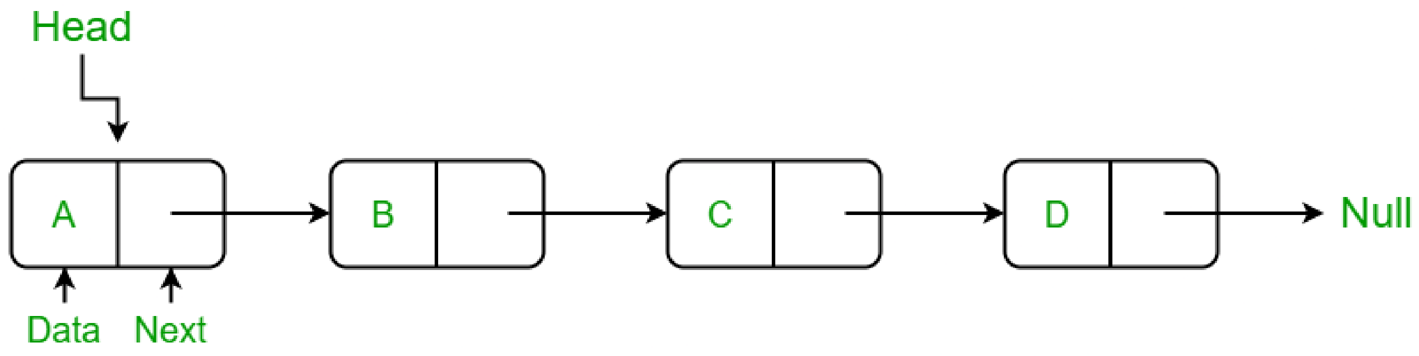
Recursion and Stack:

In recursion, each function call consumes memory space, necessitating a systematic approach to manage these calls. The call stack, a linear data structure, provides a mechanism for storing information about active function calls. However, in certain scenarios, such as when the depth of recursion is unknown or exceeds the system's predefined stack size, a dynamic data structure like a linked list becomes advantageous.



Stack Based on Linked List:

A stack implemented using a linked list offers dynamic memory allocation, allowing for the creation of a flexible stack. Each element of the linked list represents a stack frame, containing information such as function parameters, local variables, and the return address. Pointers facilitate the traversal and manipulation of the stack, ensuring efficient push and pop operations.



Advantages of using LinkedList instead of Array:

1. **Dynamic Memory Management:** Linked lists allow for dynamic memory allocation, meaning that the memory for each node is allocated as needed. This allows the stack to grow or shrink dynamically without the need to predefine a fixed size, as required in arrays.
2. **Efficient Memory Usage:** Linked lists use memory more efficiently than arrays for stacks that change in size frequently. With arrays, you may need to allocate a larger block of memory than necessary to accommodate potential growth, leading to potential memory wastage. Linked lists, on the other hand, only use memory for the actual number of elements in the stack.
3. **Ease of Insertion and Deletion:** Adding or removing elements from the top of the stack (push and pop operations) in a linked list has a time complexity of O(1) if you maintain a reference to the top of the stack. In contrast, performing the same operations in an array may require shifting elements, resulting in a time complexity of O(n), where n is the number of elements in the stack.
4. **No Maximum Capacity Limitation:** Linked lists do not have a maximum capacity limitation like arrays. This means that you can push elements onto the stack until you exhaust the available memory, making linked lists suitable for applications where the size of the stack is unpredictable or can vary significantly.
5. **Easy Implementation of Other Operations**: Linked lists provide easy implementation of other stack operations such as peek (viewing the top element without removing it) and isEmpty (checking if the stack is empty), as these operations can be performed directly on the top node without requiring traversal of the entire list.

Overall, using a linked list to implement a stack provides flexibility, efficient memory usage, and better performance for dynamic data structures compared to using an array.

What is MIPS and why are we using it?

MIPS (Microprocessor without Interlocked Pipeline Stages) is a reduced instruction set computer (RISC) architecture that was developed in the 1980s by MIPS Computer Systems Inc., now MIPS Technologies Inc. It gained widespread popularity due to its simplicity, efficiency, and performance in various computing applications.

Here's why MIPS is used:

1. **Simplicity:** MIPS architecture is designed with a reduced instruction set, focusing on a small set of simple and efficient instructions. This simplicity makes it easier to design, implement, and optimize hardware and software for MIPS-based systems.
2. **Efficiency:** RISC architectures like MIPS prioritize simplicity and efficiency in instruction execution. Instructions in MIPS are typically fixed length, allowing for faster decoding and execution. Additionally, MIPS architectures often feature pipelining, which enables the concurrent execution of multiple instructions, further enhancing performance.
3. **Performance:** MIPS processors are known for their high performance in various computing tasks, including general-purpose computing, embedded systems, digital signal processing (DSP), and graphics processing. The efficient instruction set, and pipelined execution contribute to the overall performance of MIPS-based systems.
4. **Versatility:** MIPS architecture is versatile and scalable, supporting a wide range of applications from low-power embedded systems to high-performance computing platforms. It has been used in consumer electronics, networking equipment, automotive systems, and more.
5. **Industry Adoption:** MIPS architecture has been widely adopted by industry leaders in various sectors, including Silicon Graphics, Cisco Systems, Sony, and Nintendo. Its popularity and widespread support have contributed to its continued use in diverse computing environments.
6. **Educational Purposes:** MIPS architecture is often used in educational settings to teach computer architecture, assembly language programming, and system design principles. Its simplicity and well-defined instruction set make it an excellent choice for learning the fundamentals of computer organization and architecture.

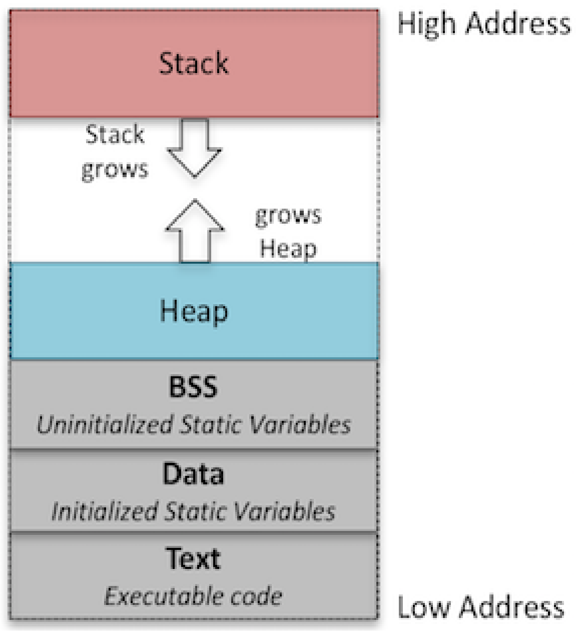
Dynamic Memory Allocation:

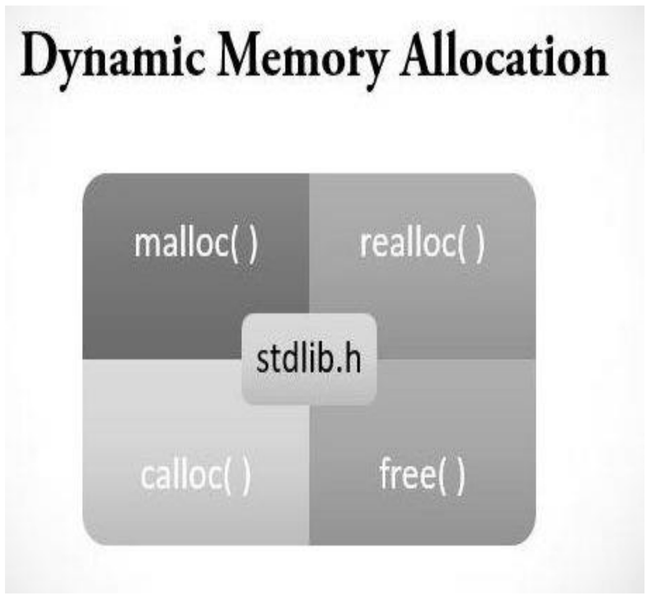
Dynamic memory allocation refers to the process of allocating memory for variables and data structures at runtime, rather than at compile time. Unlike static memory allocation, where memory is allocated and deallocated at compile time, dynamic memory allocation allows programs to allocate memory as needed during program execution.

Dynamic memory allocation is typically performed using functions provided by programming languages or libraries. In languages like C and C++, the standard library provides functions such as *malloc(), calloc(), realloc(),* and *free()* for dynamic memory management. In languages like Python and Java, dynamic memory allocation is handled implicitly by the language runtime.

Advantages of dynamic memory allocation include:

1. **Flexibility:** Dynamic memory allocation allows programs to allocate memory based on runtime conditions and requirements. This flexibility enables programs to adapt to varying data sizes and structures.
2. **Efficiency:** Dynamic memory allocation can be more memory-efficient than static memory allocation, particularly when dealing with data structures whose size or lifetime is not known at compile time. Memory can be allocated and deallocated as needed, reducing memory waste.
3. **Dynamic Data Structures:** Dynamic memory allocation is essential for implementing dynamic data structures such as linked lists, trees, and hash tables. These data structures can grow or shrink in size dynamically, requiring memory allocation and deallocation at runtime.





Memory Segment:

In computer architecture and operating systems, memory segmentation refers to the division of a computer's primary memory (RAM) into distinct segments or regions, each serving a specific purpose. Memory segmentation provides a structured organization of memory, allowing the operating system and applications to manage memory resources effectively.

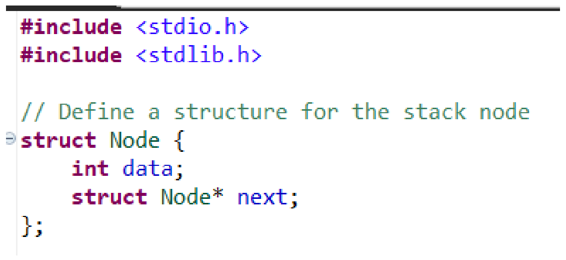
There are several common memory segments in a typical computer system:

1. **Text Segment:** This segment stores the executable code of a program. It includes instructions and constants required for program execution. The code segment is typically read-only to prevent accidental modification of program instructions.
2. **Data Segment:** The data segment contains static and global variables used by a program. It includes initialized data (e.g., global variables with predefined values) and uninitialized data (e.g., variables declared but not initialized). The data segment is typically writable, allowing programs to modify variable values during execution.
3. **Stack Segment:** The stack segment is used for storing function call information, local variables, and function parameters. It follows a Last-In-First-Out (LIFO) structure, where the most recently pushed item is the first to be popped. The stack segment is essential for managing function calls, recursion, and local variable storage.
4. **Heap Segment:** The heap segment is used for dynamic memory allocation. It stores dynamically allocated memory blocks requested by programs during runtime. Unlike the stack, which follows a strict LIFO structure, memory allocation and deallocation in the heap can occur in a non-linear fashion.

Memory segmentation facilitates memory management by providing isolation between different types of data and code, preventing unintended access and modification. It also allows the operating system to enforce memory protection mechanisms, such as read-only access to code segments and stack overflow protection.

Implementation in C:

In C programming language, implementing recursion by stack based on a linked list involves defining structures for nodes and functions for stack operations. The 'push' operation adds a new node to the top of the stack, while 'pop' removes the top node. Recursion functions utilize these stack operations to manage function calls and local variables effectively.

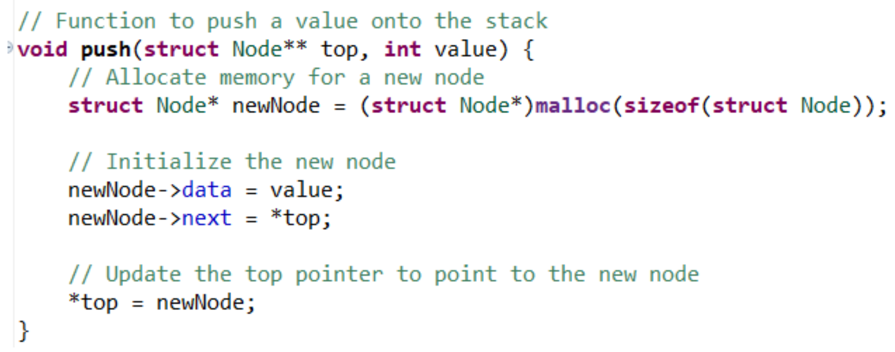


```c
#include <stdio.h>
#include <stdlib.h>

// Define a structure for the stack node
struct Node {
    int data;
    struct Node* next;
};
```

1. ***#include <stdio.h>* and #include <stdlib.h> :**
   - <***stdio.h***> is necessary for input and output operations, like ***printf*** and ***scanf***.
   - <***stdlib.h***> is necessary for dynamic memory allocation functions, like ***malloc*** and ***free***.
2. ***struct Node* Definition:**
   - A structure is a user-defined data type that groups related data items of different data types under a single name.
   - ***data***: An integer field to store the data.
   - ***next***: A pointer to another Node struct, indicating the next node in the linked list.

So, this structure is essentially a basic unit of a singly linked list, which is commonly used to implement a stack. Each node holds some data and a reference to the next node, forming a chain-like structure.

➢ Push function:

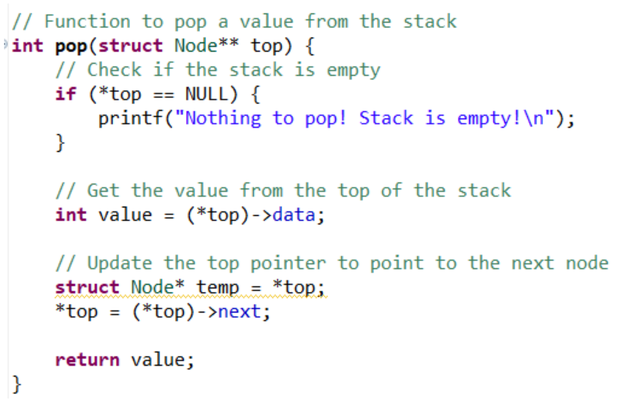


```c
// Function to push a value onto the stack
void push(struct Node** top, int value) {
    // Allocate memory for a new node
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));

    // Initialize the new node
    newNode->data = value;
    newNode->next = *top;

    // Update the top pointer to point to the new node
    *top = newNode;
}
```

1. **Push Parameters:**
   - ***`struct Node** top`:*** A pointer to a pointer to the top of the stack. It's a pointer to a pointer because the function needs to modify the value of the top pointer (updating it to point to the newly created node).
   - ***`int value`:*** The value to be pushed onto the stack.
2. **Allocate Memory for a New Node:**
   - ***`struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));`***
     - This line allocates memory for a new node using the ***`malloc`*** function.
     - ***`sizeof(struct Node)`*** ensures that enough memory is allocated to hold a ***`Node`*** structure.
     - The return value of ***`malloc`*** is cast to ***`struct Node*`*** to match the type of ***`newNode`***.
3. **Initialize the New Node:**
   - ***`newNode->data = value;`:*** This line assigns the input ***`value`*** to the ***`data`*** field of the newly created node.
   - ***`newNode->next = *top;`:*** This line sets the `next` pointer of the new node to point to the current top of the stack. This way, the new node becomes the top of the stack, and its ***`next`*** pointer points to the previous top of the stack.
4. **Update the Top Pointer:**
   - ***`*top = newNode;`:*** This line updates the top pointer to point to the newly created node. Now, the newly created node becomes the top of the stack.

In summary, this function allocates memory for a new node, initializes it with the provided value, updates the top pointer to point to the new node, and pushes it onto the stack. If memory allocation fails, it prints an error message and exits the program.

➢ Pop function:



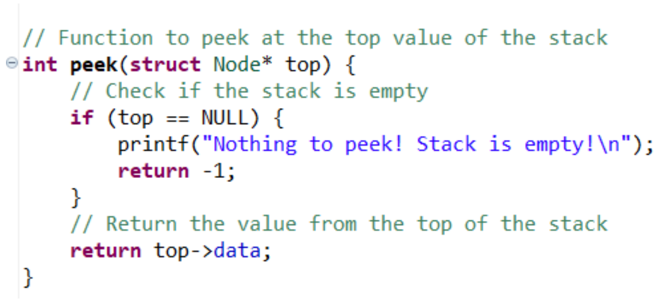
```
// Function to pop a value from the stack
int pop(struct Node** top) {
    // Check if the stack is empty
    if (*top == NULL) {
        printf("Nothing to pop! Stack is empty!\n");
    }

    // Get the value from the top of the stack
    int value = (*top)->data;

    // Update the top pointer to point to the next node
    struct Node* temp = *top;
    *top = (*top)->next;

    return value;
}
```

1. ***`int pop(struct Node** top)`:***
   - A pointer to a pointer to the top of the stack. It's a pointer to a pointer because the function needs to modify the value of the top pointer (updating it to point to the next node after popping).

2. **Check if the Stack is Empty:**
   - ***`if (*top == NULL) { ... }`*** : This ***`if`*** statement checks if the stack is empty by verifying if the ***`top`*** pointer is ***`NULL`***. If the stack is empty, it prints a message indicating that there's nothing to pop, and the function returns without popping anything.
3. **Get the Value from the Top of the Stack:**
   - ***`int value = (*top)->data;`*** :This line retrieves the data from the top of the stack and stores it in the variable ***`value`***. It accesses the ***`data`*** field of the node pointed to by ***`*top`***.
5. **Update the Top Pointer:**
   - ***`struct Node* temp = *top;`:*** This line temporarily stores the address of the node to be popped in a variable named ***`temp`***. This is done to avoid memory leaks and ensure proper deallocation if needed.
   - ***`*top = (*top)->next;`:*** This line updates the ***`top`*** pointer to point to the next node in the stack. It effectively removes the top node from the stack by moving the ***`top`*** pointer to the next node.
6. **Return the Popped Value:**
   - ***`return value;`***: This line returns the value that was popped from the stack. The calling function can then use this returned value as needed.

This function is responsible for popping a value from the stack. It ensures that the stack is not empty before attempting to pop, retrieves the value from the top of the stack, updates the ***`top`*** pointer to point to the next node, and returns the popped value. If the stack is empty, it prints a message indicating so and returns without popping anything.

➢ Peek Function:



```c
// Function to peek at the top value of the stack
int peek(struct Node* top) {
    // Check if the stack is empty
    if (top == NULL) {
        printf("Nothing to peek! Stack is empty!\n");
        return -1;
    }
    // Return the value from the top of the stack
    return top->data;
}
```

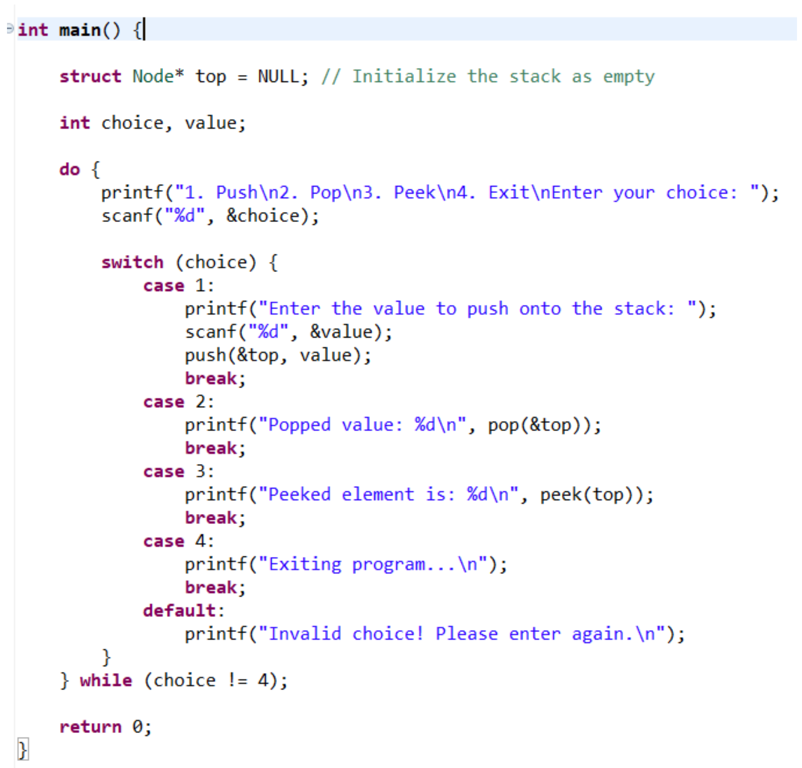
1. ***`int peek(struct Node* top)`:*** This function takes a pointer to the top of the stack (***`top`***) as its argument and returns an integer, which is the value of the top element of the stack.
2. **Check for Empty Stack:**
   - ***`if (top == NULL)`*** {`: This ***`if`*** statement checks if the stack is empty by verifying if the pointer to the top of the stack (`top`***)*** is ***`NULL`***.

- **\`*printf("Nothing to peek! Stack is empty!\n");*\`:** If the stack is empty, this message is printed to inform the user that there's nothing to peek.

3. **Return Value from the Top of the Stack:**
   1. **\`*return top->data;*\`:** If the stack is not empty, this line returns the data value of the top node of the stack (\`***top->data***\`). It accesses the data field of the top node using the arrow operator (\`->\`).

In summary, the \`***peek***\` function checks if the stack is empty. If not, it returns the value of the top element of the stack without modifying the stack itself.

➢ Main Function:



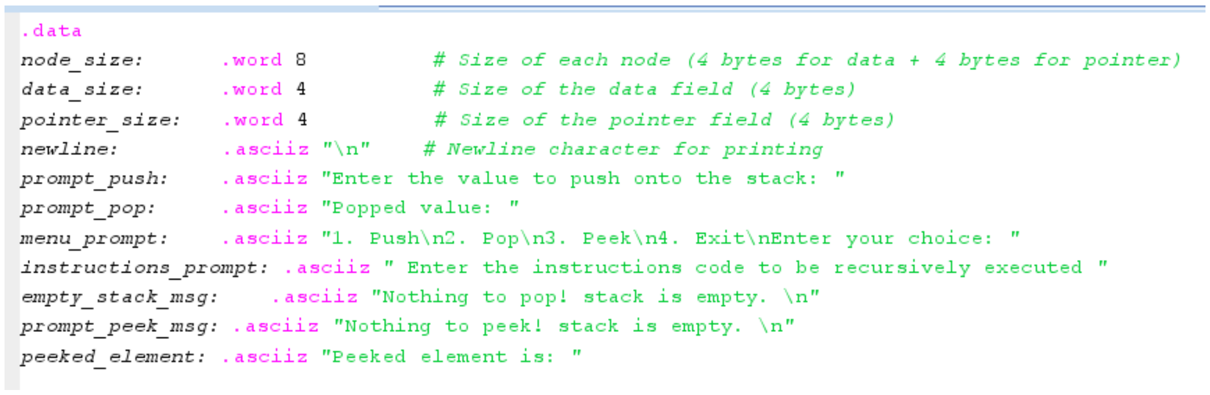
```
int main() {

    struct Node* top = NULL; // Initialize the stack as empty

    int choice, value;

    do {
        printf("1. Push\n2. Pop\n3. Peek\n4. Exit\nEnter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                printf("Enter the value to push onto the stack: ");
                scanf("%d", &value);
                push(&top, value);
                break;
            case 2:
                printf("Popped value: %d\n", pop(&top));
                break;
            case 3:
                printf("Peeked element is: %d\n", peek(top));
                break;
            case 4:
                printf("Exiting program...\n");
                break;
            default:
                printf("Invalid choice! Please enter again.\n");
        }
    } while (choice != 4);

    return 0;
}
```

1. **\`*int main() {*\`:** This is the main function, which serves as the entry point of the program. It returns an integer (\`***int***\`), indicating the status of the program's execution to the operating system.
2. **Initialization:**
   - **\`*struct Node* top = NULL*;**\`: This line initializes the stack as empty by declaring a pointer to the top of the stack (\`***top***\`) and setting it to \`***NULL***\`. This pointer will be used to keep track of the top element of the stack.

3. **Menu Loop:**
   - ***`do { ... } while (choice != 4);`:*** This is a do-while loop that continues to execute until the user chooses to exit the program (selects option 4). The loop allows the user to perform multiple stack operations before choosing to exit.
4. **Menu Prompt:**
   - ***`printf("1. Push\n2. Pop\n3. Peek\n4. Exit\nEnter your choice: ");`:*** This line prints the menu options to the console, prompting the user to enter their choice.
5. **User Input:**
   - **`scanf("%d", &choice);`**: This line reads the user's choice from the console and stores it in the variable `choice`.
6. **Switch Statement:**
   - ***`switch (choice) { ... }`:*** This switch statement evaluates the value of `choice` and executes the corresponding case based on the user's input.
7. **Menu Options:**
   - ***Case 1 (`Push`):*** Prompts the user to enter a value to push onto the stack and calls the ***`push`*** function.
   - ***Case 2 (`Pop`):*** Calls the ***`pop`*** function to remove and print the top element of the stack.
   - ***Case 3 (`Peek`):*** Calls the ***`peek`*** function to print the value of the top element of the stack without removing it.
   - ***Case 4 (`Exit`):*** Prints a message indicating the program is exiting and terminates the loop.
8. **Loop Continuation:**
   - The loop continues to execute until the user selects the exit option (case 4).
9. **Return Statement:**
   - `***return 0***;`: This line indicates the successful termination of the program by returning `0` to the operating system. A return value of `0` conventionally signifies successful execution.

MIPS Assembly Code:

➢ Data section :



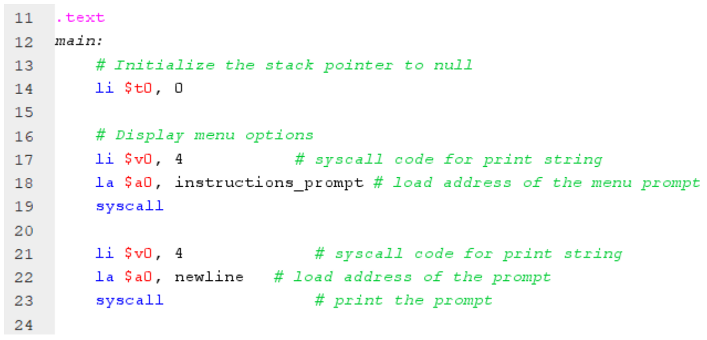
```
.data
node_size:       .word 8          # Size of each node (4 bytes for data + 4 bytes for pointer)
data_size:       .word 4          # Size of the data field (4 bytes)
pointer_size:    .word 4          # Size of the pointer field (4 bytes)
newline:         .asciiz "\n"     # Newline character for printing
prompt_push:     .asciiz "Enter the value to push onto the stack: "
prompt_pop:      .asciiz "Popped value: "
menu_prompt:     .asciiz "1. Push\n2. Pop\n3. Peek\n4. Exit\nEnter your choice: "
instructions_prompt: .asciiz " Enter the instructions code to be recursively executed "
empty_stack_msg:    .asciiz "Nothing to pop! stack is empty. \n"
prompt_peek_msg: .asciiz "Nothing to peek! stack is empty. \n"
peeked_element: .asciiz "Peeked element is: "
```

- ***`node_size`*** stores the size of each node, which is typically the sum of the sizes of the data field and the pointer field.
- ***`data_size`*** stores the size of the data field in each node.
- ***`pointer_size`*** stores the size of the pointer field in each node.
- ***`newline`***: It is used for printing a newline when required.
- ***`prompt_push`*** contains the message prompting the user to enter a value to push onto the stack.
- ***`prompt_pop`*** contains the message indicating the popped value.
- ***`menu_prompt`*** contains the menu options displayed to the user.
- ***`instructions_prompt`*** contains instructions prompt.
- ***`empty_stack_msg`*** contains the message indicating that the stack is empty during a pop operation.
- ***`prompt_peek_msg`*** contains the message indicating that the stack is empty during a peek operation.
- ***`peeked_element`*** contains the message indicating the peeked element.

These strings are used throughout the program to provide instructions to the user, display stack-related messages, and prompt for input. By defining them as strings in the `.data` section, they can be easily referenced and reused throughout the program.

Text Section:

➢ Main Label:

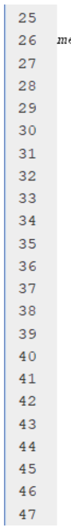


```
.text
main:
    # Initialize the stack pointer to null
    li $t0, 0

    # Display menu options
    li $v0, 4           # syscall code for print string
    la $a0, instructions_prompt # load address of the menu prompt
    syscall

    li $v0, 4             # syscall code for print string
    la $a0, newline   # load address of the prompt
    syscall               # print the prompt

```

- "***.text***": This directive specifies that the following lines contain executable instructions.
- "***main***:": This is a label indicating the start of the `***main***` function.
- "***li $t0, 0***": This instruction loads the immediate value 0 into register ***'$t0'***. In this context, register ***'$t0'*** is being used to represent the stack pointer. Setting it to 0 initializes the stack pointer to null, indicating an empty stack.
- "***li $v0, 4***": This instruction loads the immediate value 4 into register ***'$v0'***. In MIPS assembly, register ***'$v0'*** is typically used to specify a system call service number. Here, the value 4 corresponds to the service number for printing a string.
- "***la $a0, instructions_prompt***": This instruction loads the address of the `instructions_prompt` string into register ***'$a0'***. The '***la'*** (load address) instruction is used to load the address of a memory location. The 'instructions_prompt' string contains the prompt message for entering instructions code to be recursively executed.
- "***syscall***": This instruction triggers a system call, with the service number specified in register ***'$v0'*** and the arguments passed in registers ***'$a0'*** to ***'$a3'***. In this case, it prints the string stored at the address in register ***'$a0'***.
- "***li $v0, 4***": Similar to line 4, this instruction loads the immediate value 4 into register ***'$v0'***, indicating the service number for printing a string.
- "***la $a0, newline***": This instruction loads the address of the '***newline'*** string into register ***'$a0'***. The 'newline' string contains the newline character (`\n`), which is used to print a newline.
- "***syscall***": This instruction triggers a system call to print the newline character, which results in moving the cursor to the beginning of the next line in the output.

In summary, this code snippet initializes the stack pointer to null and then displays menu options by printing the `instructions_prompt` string followed by a newline character. The menu options prompt the user to enter instructions code to be recursively executed.

➢ Menu Label:



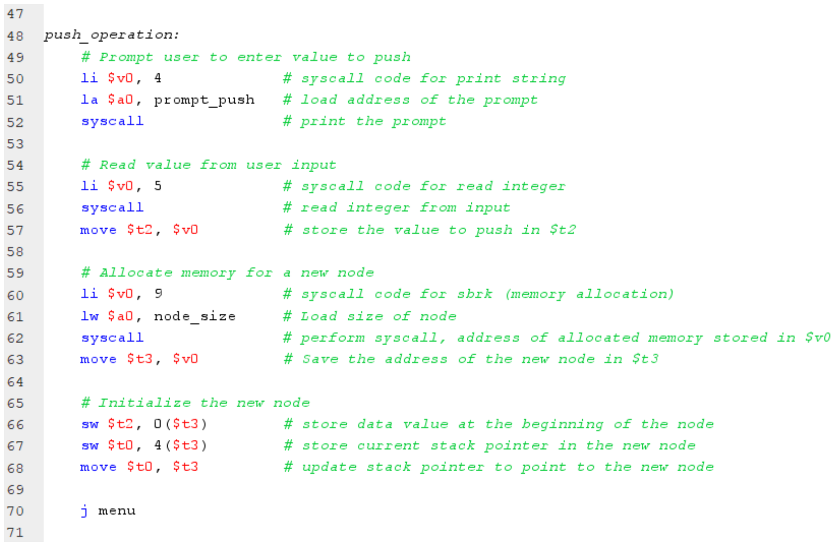
```

menu:
    li $v0, 4           # syscall code for print string
    la $a0, menu_prompt # load address of the menu prompt
    syscall             # print the menu prompt

    # Read user choice
    li $v0, 5           # syscall code for read integer
    syscall             # read integer from input
    move $t1, $v0       # store the user choice in $t1

    # Execute user choice
    # Push operation
    beq $t1, 1, push_operation
    # Pop operation
    beq $t1, 2, pop_operation
    # Print stack
    beq $t1, 3, peek
    # Exit program
    beq $t1, 4, exit_program
    # Invalid choice
    j menu

```

- ***menu:***: This is a label indicating the start of the menu section.
- ***li $v0, 4***: This instruction loads the immediate value 4 into register ***$v0***. In MIPS assembly, ***register $v0*** is typically used to specify a system call service number. Here, the value 4 corresponds to the service number for printing a string.
- ***la $a0, menu_prompt***: This instruction loads the address of the menu_prompt string into register ***$a0***. The la (load address) instruction is used to load the address of a memory location. The ***menu_prompt*** string contains the menu options prompting the user to choose an operation.
- ***syscall***: This instruction triggers a system call, with the service number specified in register ***$v0*** and the arguments passed in registers ***$a0*** to ***$a3***. In this case, it prints the string stored at the address in register $a0.
- ***li $v0, 5***: This instruction loads the immediate value 5 into register ***$v0***. In MIPS assembly, register ***$v0*** is typically used to specify a system call service number. Here, the value 5 corresponds to the service number for reading an integer from input.
- ***syscall***: This instruction triggers a system call to read an integer from the input. The integer entered by the user is stored in register ***$v0***.
- ***move $t1, $v0***: This instruction copies the value in register $v0 (which contains the user's choice) into register ***$t1***. Register ***$t1*** is commonly used to store temporary values in MIPS assembly.
- ***beq $t1, 1, push_operation***: This instruction checks if the value in register ***$t1*** is equal to 1. If it is, the control flow branches to the label ***push_operation***. This represents the case where the user chooses to perform a push operation on the stack.
- ***beq $t1, 2, pop_operation:*** Similarly, this instruction checks if the value in register ***$t1*** is equal to 2. If it is, the control flow branches to the label pop_operation. This represents the case where the user chooses to perform a pop operation on the stack.

- ***beq $t1, 3, peek***: This instruction checks if the value in register ***$t1*** is equal to 3. If it is, the control flow branches to the label peek. This represents the case where the user chooses to peek at the top element of the stack.
- ***beq $t1, 4, exit_program***: Similarly, this instruction checks if the value in register ***$t1*** is equal to 4. If it is, the control flow branches to the label ***exit_program***. This represents the case where the user chooses to exit the program.
- ***j menu***: If none of the previous conditions are met, this instruction unconditionally jumps back to the menu label, restarting the menu loop to prompt the user for another choice.

In summary, this code snippet displays a menu of options, reads the user's choice, and then executes the corresponding operation based on the user's input. If the input is not valid, it loops back to the menu to prompt the user again.

➢ Push Label:



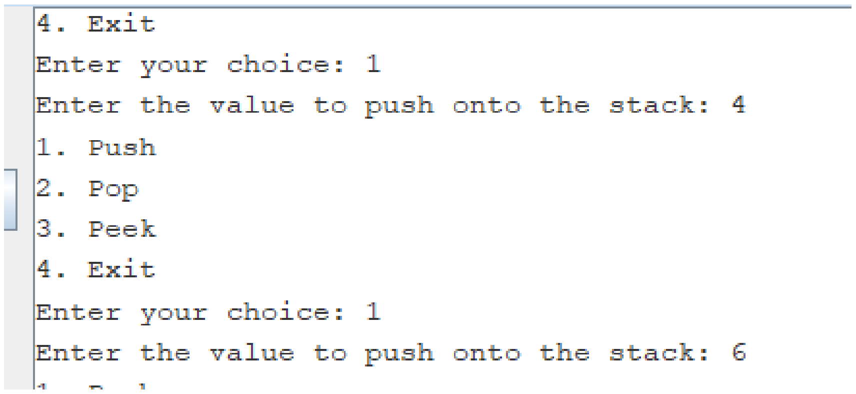
```

push_operation:
    # Prompt user to enter value to push
    li $v0, 4               # syscall code for print string
    la $a0, prompt_push     # load address of the prompt
    syscall                 # print the prompt

    # Read value from user input
    li $v0, 5               # syscall code for read integer
    syscall                 # read integer from input
    move $t2, $v0           # store the value to push in $t2

    # Allocate memory for a new node
    li $v0, 9               # syscall code for sbrk (memory allocation)
    lw $a0, node_size       # Load size of node
    syscall                 # perform syscall, address of allocated memory stored in $v0
    move $t3, $v0           # Save the address of the new node in $t3

    # Initialize the new node
    sw $t2, 0($t3)          # store data value at the beginning of the node
    sw $t0, 4($t3)          # store current stack pointer in the new node
    move $t0, $t3           # update stack pointer to point to the new node

    j menu

```

- ***push_operation:***: This is a label indicating the start of the ***push_operation*** subroutine.
- ***li $v0, 4***: This instruction loads the immediate value 4 into register ***$v0***. In MIPS assembly, register ***$v0*** is typically used to specify a system call service number. Here, the value 4 corresponds to the service number for printing a string.
- ***la $a0, prompt_push***: This instruction loads the address of the ***prompt_push*** string into register ***$a0***. The ***la (load address)*** instruction is used to load the address of a memory location. The prompt_push string contains the prompt message for entering the value to be pushed onto the stack.
- ***syscall***: This instruction triggers a system call, with the service number specified in ***register $v0*** and the arguments passed in ***registers $a0*** to ***$a3***. In this case, it prints the string stored at the address in register ***$a0***, prompting the user to enter the value to push onto the stack.

- ***li $v0, 5***: This instruction loads the immediate value 5 into register ***$v0***. In MIPS assembly, register ***$v0*** is typically used to specify a system call service number. Here, the value 5 corresponds to the service number for reading an integer from input.
- ***syscall***: This instruction triggers a system call to read an integer from the input. The integer entered by the user is stored in register ***$v0***.
- ***move $t2, $v0***: This instruction copies the value in register ***$v0*** (which contains the user's input) into register ***$t2***. Register ***$t2*** is commonly used to store temporary values in MIPS assembly.
- ***li $v0, 9***: This instruction loads the immediate value 9 into register ***$v0***. In MIPS assembly, register ***$v0*** is typically used to specify a system call service number. Here, the value 9 corresponds to the service number for sbrk, which is used for memory allocation.
- ***lw $a0, node_size***: This instruction loads the value stored in memory location node_size into register ***$a0***. The lw (load word) instruction is used to load the value from memory. The node_size variable contains the size of each node in the stack.
- ***syscall***: This instruction triggers a system call to perform memory allocation using sbrk, with the size specified in register ***$a0***. The address of the allocated memory block is returned and stored in register ***$v0***.
- ***move $t3, $v0***: This instruction copies the value in ***register $v0*** (which contains the address of the allocated memory block) into register ***$t3***. Register ***$t3*** is used to store the address of the new node.
- ***sw $t2, 0($t3):*** This instruction stores the value to be pushed (stored in register ***$t2***) at the beginning of the memory block pointed to by register ***$t3***. This initializes the data field of the new node with the value to be pushed.
- ***sw $t0, 4($t3):*** This instruction stores the current stack pointer (stored in register ***$t0***) in the memory block pointed to by register ***$t3***, offset by 4 bytes. This initializes the pointer field of the new node with the address of the previous top of the stack.
- ***move $t0, $t3:*** This instruction updates the stack pointer ***($t0)*** to point to the newly allocated node. After pushing a new value onto the stack, the top of the stack is updated to point to the newly created node.
- ***j menu***: This instruction unconditionally jumps back to the menu label, returning control to the menu loop after successfully pushing a value onto the stack.

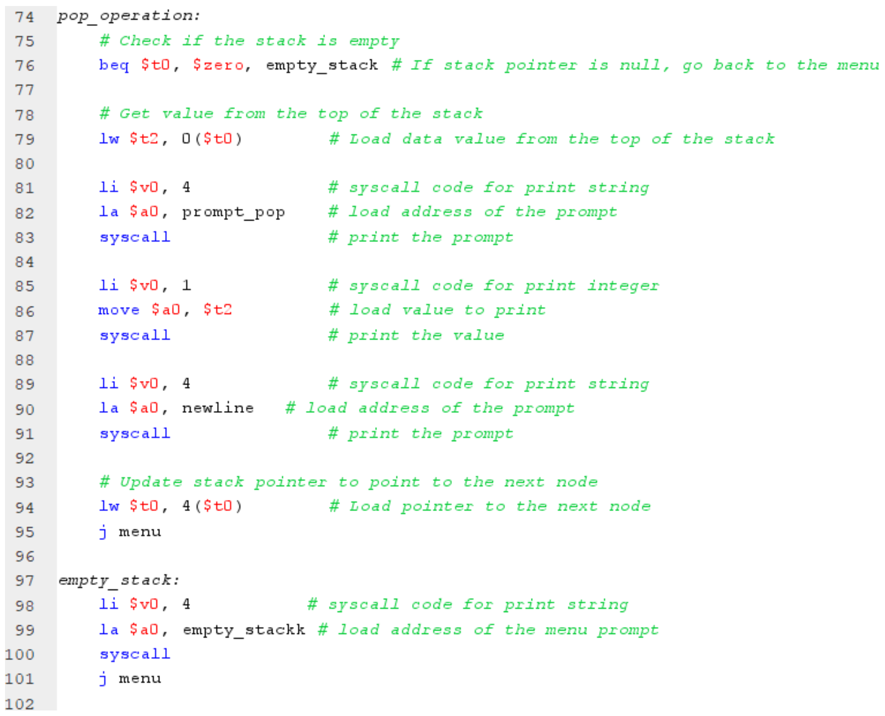
In summary, this subroutine prompts the user to enter a value to be pushed onto the stack, reads the user's input, allocates memory for a new node, initializes the new node with the entered value, updates the stack pointer to point to the new node, and then returns control to the menu loop.

Sample input to push:



```
4. Exit
Enter your choice: 1
Enter the value to push onto the stack: 4
1. Push
2. Pop
3. Peek
4. Exit
Enter your choice: 1
Enter the value to push onto the stack: 6
```

➢ Pop Label :



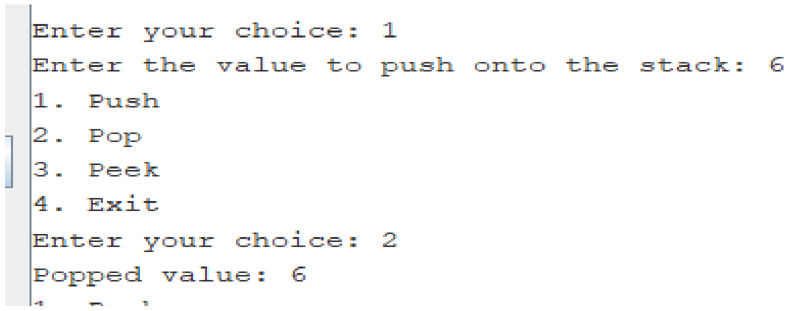
```
pop_operation:
    # Check if the stack is empty
    beq $t0, $zero, empty_stack # If stack pointer is null, go back to the menu

    # Get value from the top of the stack
    lw $t2, 0($t0)         # Load data value from the top of the stack

    li $v0, 4              # syscall code for print string
    la $a0, prompt_pop     # load address of the prompt
    syscall                # print the prompt

    li $v0, 1              # syscall code for print integer
    move $a0, $t2          # load value to print
    syscall                # print the value

    li $v0, 4              # syscall code for print string
    la $a0, newline    # load address of the prompt
    syscall                # print the prompt

    # Update stack pointer to point to the next node
    lw $t0, 4($t0)         # Load pointer to the next node
    j menu

empty_stack:
    li $v0, 4            # syscall code for print string
    la $a0, empty_stackk # load address of the menu prompt
    syscall
    j menu
```

1. **Check if the stack is empty:**
   - ***`beq $t0, $zero, empty_stack`:*** This instruction checks if the stack pointer ***`$t0`*** is null (indicating an empty stack). If the stack is empty, it branches to the label ***`empty_stack`***, which handles the case where there's nothing to pop from the stack.

2. **Get value from the top of the stack:**
   - ***`lw $t2, 0($t0)`:*** This instruction loads the data value from the top of the stack into register ***`$t2`***. It accesses the data field of the node pointed to by the stack pointer ***`$t0`.***
3. **Print prompt for pop:**
   - ***`li $v0, 4`:*** Load syscall code for printing a string.
   - ***`la $a0, prompt_pop`***: Load the address of the prompt for popping from the stack.
   - ***`syscall`***: Print the prompt for popping from the stack.
4. **Print the popped value:**
   - ***`li $v0, 1`:*** Load syscall code for printing an integer.
   - ***`move $a0, $t2`:*** Move the value to print (popped value) to register ***`$a0`***.
   - ***`syscall`***: Print the popped value.
5. **Print newline:**
   - ***`li $v0, 4`***: Load syscall code for printing a string.
   - ***`la $a0, newline`***: Load the address of the newline character.
   - ***`syscall`***: Print the newline character.
6. **Update stack pointer:**
   - ***`lw $t0, 4($t0)`:*** Load the pointer to the next node from the current top of the stack. This updates the stack pointer to point to the next node below the current top.
   - ***`j menu`***: Jump back to the menu to allow further operations on the stack.
7. **Handle empty stack case (*`empty_stack`* label):**
   - ***`li $v0, 4`***: Load syscall code for printing a string.
   - ***`la $a0, empty_stackk`***: Load the address of the message indicating an empty stack.
   - ***`syscall`***: Print the message indicating that there's nothing to pop from the stack.
   - ***`j menu`***: Jump back to the menu to allow the user to choose another operation.

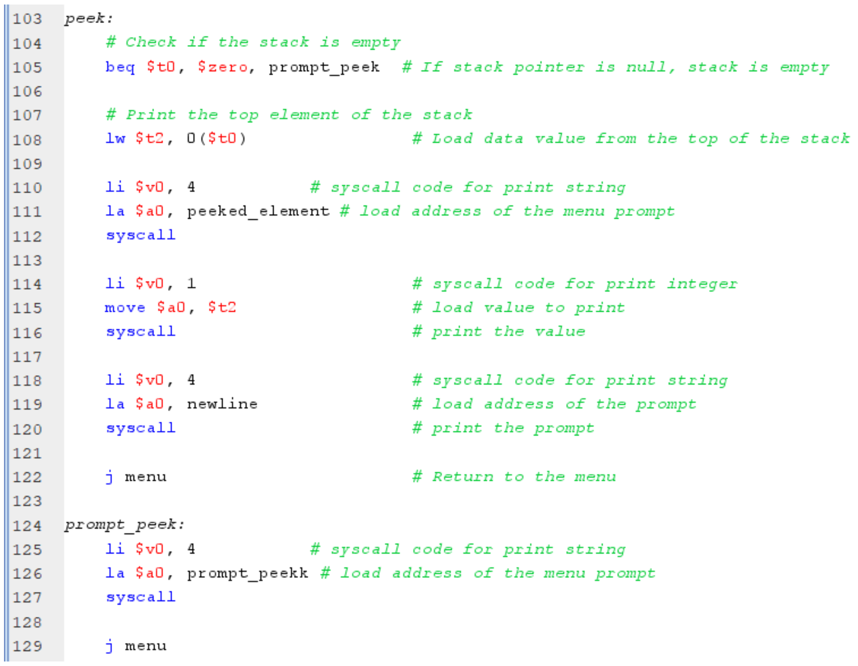
Overall, this code segment handles popping an element from the stack. It first checks if the stack is empty. If not, it pops the top element, prints it, updates the stack pointer, and returns to the menu. If the stack is empty, it prints a message and returns to the menu.

Sample input to pop:



```
Enter your choice: 1
Enter the value to push onto the stack: 6
1. Push
2. Pop
3. Peek
4. Exit
Enter your choice: 2
Popped value: 6
```

➢ Peek Label:



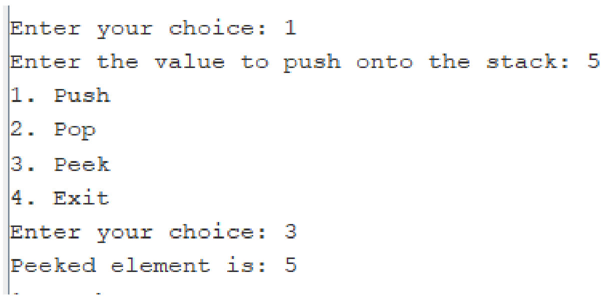
```
peek:
    # Check if the stack is empty
    beq $t0, $zero, prompt_peek  # If stack pointer is null, stack is empty

    # Print the top element of the stack
    lw $t2, 0($t0)               # Load data value from the top of the stack

    li $v0, 4           # syscall code for print string
    la $a0, peeked_element # load address of the menu prompt
    syscall

    li $v0, 1                    # syscall code for print integer
    move $a0, $t2                # load value to print
    syscall                      # print the value

    li $v0, 4                    # syscall code for print string
    la $a0, newline              # load address of the prompt
    syscall                      # print the prompt

    j menu                       # Return to the menu

prompt_peek:
    li $v0, 4           # syscall code for print string
    la $a0, prompt_peekk # load address of the menu prompt
    syscall

    j menu
```

1. **Check if the stack is empty:**
   - `***beq $t0, $zero, prompt_peek***`: This instruction checks if the stack pointer `***$t0***` is null (indicating an empty stack). If the stack is empty, it branches to the label `***prompt_peek***`, which handles the case where there's nothing to peek from the stack.
2. **Print the top element of the stack:**
   - `***lw $t2, 0($t0)***`**:** This instruction loads the data value from the top of the stack into register `***$t2***`**.** It accesses the data field of the node pointed to by the stack pointer `***$t0***`.
3. **Print message indicating peeked element:**
   - `***li $v0, 4***`: Load syscall code for printing a string.
   - `***la $a0, peeked_element***`: Load the address of the message indicating the peeked element.
   - `***syscall***`: Print the message indicating the peeked element.
4. **Print the peeked value:**
   - `***li $v0, 1***`: Load syscall code for printing an integer.
   - `***move $a0, $t2***`: Move the value to print (peeked value) to register `$a0`.
   - `***syscall***`: Print the peeked value.
5. **Print newline:**
   - `***li $v0, 4***`: Load syscall code for printing a string.
   - `***la $a0, newline***`: Load the address of the newline character.
   - `***syscall***`: Print the newline character.
6. **Return to the menu:**
   - `***j menu***`: Jump back to the menu to allow further operations on the stack.

7. **Handle empty stack case (`*prompt_peek*` *label*):**
   - `***li $v0, 4***`: Load syscall code for printing a string.
   - `***la $a0, prompt_peekk***`: Load the address of the message indicating an empty stack during peek operation.
   - `***syscall***`: Print the message indicating that there's nothing to peek from the stack.
   - `***j menu***`: Jump back to the menu to allow the user to choose another operation.

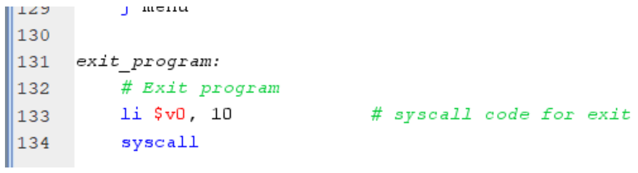
This code segment handles the peek operation. It first checks if the stack is empty. If not, it peeks at the top element, prints it, and returns to the menu. If the stack is empty, it prints a message and returns to the menu.

Sample input to peek:



```
Enter your choice: 1
Enter the value to push onto the stack: 5
1. Push
2. Pop
3. Peek
4. Exit
Enter your choice: 3
Peeked element is: 5
```

➢ Exit program :



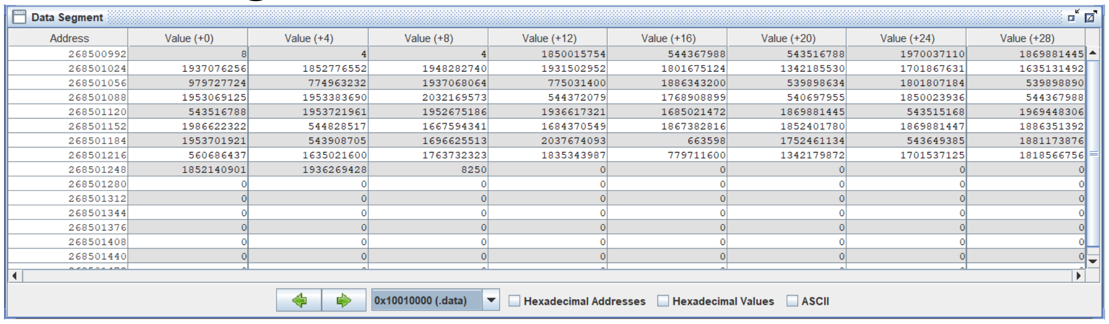
```
    j menu

exit_program:
    # Exit program
    li $v0, 10              # syscall code for exit
    syscall
```

- `***li $v0, 10***`**:** This instruction loads the syscall code `***10***` into register `***$v0***`, which is the code for the `***exit***` syscall. The `***exit***` syscall terminates the program.
- `***syscall***`: This instruction executes the system call specified by the value in register `***$v0***`, which in this case is the `***exit***` syscall. When the `***exit***` syscall is executed, the program terminates immediately without any further execution.

In summary, the `***exit_program***` section of the code is responsible for terminating the program using the `***exit***` syscall. When executed, it immediately ends the program's execution and returns control to the operating system.

Memory Segment:

➢ Data Segment:

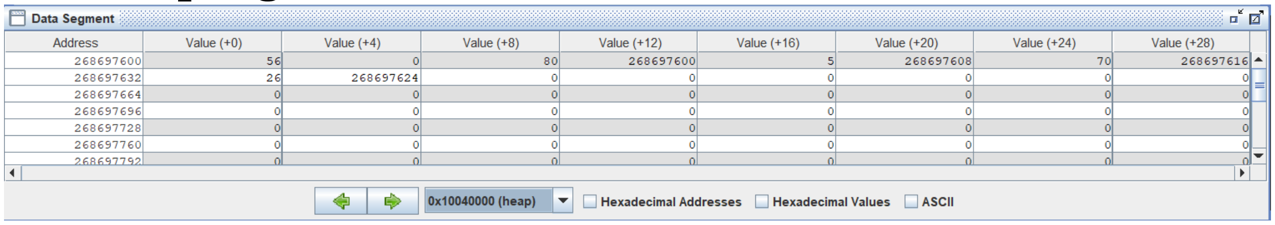


Data Segment

Address	Value (+0)	Value (+4)	Value (+8)	Value (+12)	Value (+16)	Value (+20)	Value (+24)	Value (+28)
268500992	8	4	4	1850015754	544367988	543516788	1970037110	1869881445
268501024	1937076256	1852776552	1948282740	1931502952	1801675124	1342185530	1701867631	1635131492
268501056	979727724	774963232	1937068064	775031400	1886343200	539898634	1801807184	539898890
268501088	1953069125	1953383690	2032169573	544372079	1768908899	540697955	1850023936	544367988
268501120	543516788	1953721961	1952675186	1936617321	1685021472	1869881445	543515168	1969448306
268501152	1986622322	544828517	1667594341	1684370549	1867382816	1852401780	1869881447	1886351392
268501184	1953701921	543908705	1696625513	2037674093	663598	1752461134	543649385	1881173876
268501216	560686437	1635021600	1763732323	1835343987	779711600	1342179872	1701537125	1818566756
268501248	1852140901	1936269428	8250	0	0	0	0	0
268501280	0	0	0	0	0	0	0	0
268501312	0	0	0	0	0	0	0	0
268501344	0	0	0	0	0	0	0	0
268501376	0	0	0	0	0	0	0	0
268501408	0	0	0	0	0	0	0	0
268501440	0	0	0	0	0	0	0	0

0x10010000 (.data) ☐ Hexadecimal Addresses ☐ Hexadecimal Values ☐ ASCII

➢ Heap Segment:

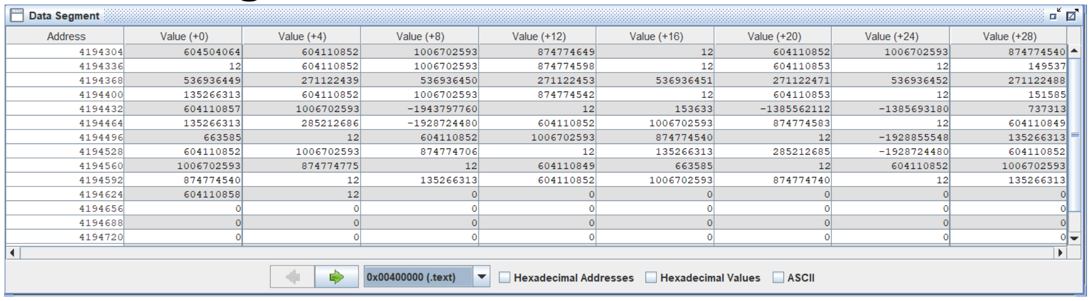


Data Segment

Address	Value (+0)	Value (+4)	Value (+8)	Value (+12)	Value (+16)	Value (+20)	Value (+24)	Value (+28)
268697600	56	0	80	268697600	5	268697608	70	268697616
268697632	26	268697624	0	0	0	0	0	0
268697664	0	0	0	0	0	0	0	0
268697696	0	0	0	0	0	0	0	0
268697728	0	0	0	0	0	0	0	0
268697760	0	0	0	0	0	0	0	0
268697792	0	0	0	0	0	0	0	0

0x10040000 (heap) ☐ Hexadecimal Addresses ☐ Hexadecimal Values ☐ ASCII

➢ Text Segment:

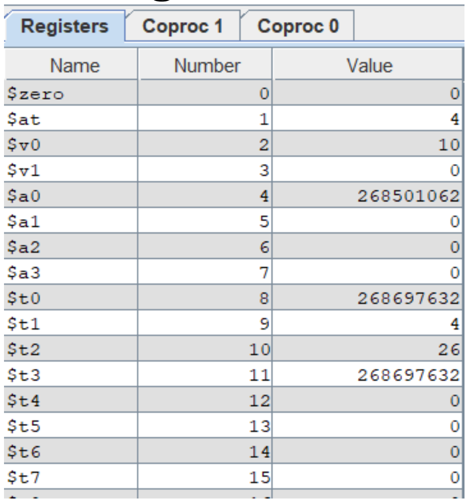


Data Segment

Address	Value (+0)	Value (+4)	Value (+8)	Value (+12)	Value (+16)	Value (+20)	Value (+24)	Value (+28)
4194304	604504064	604110852	1006702593	874774649	12	604110852	1006702593	874774540
4194336	12	604110852	1006702593	874774598	12	604110853	12	149537
4194368	536936449	271122439	536936450	271122453	536936451	271122471	536936452	271122488
4194400	135266313	604110852	1006702593	874774542	12	604110853	12	151585
4194432	604110857	1006702593	-1943797760	12	153633	-1385562112	-1385693180	737313
4194464	135266313	285212686	-1928724480	604110852	1006702593	874774583	12	604110849
4194496	663585	12	604110852	1006702593	874774540	12	-1928855548	135266313
4194528	604110852	1006702593	874774706	12	135266313	285212685	-1928724480	604110852
4194560	1006702593	874774775	12	604110849	663585	12	604110852	1006702593
4194592	874774540	12	135266313	604110852	1006702593	874774740	12	135266313
4194624	604110858	12	0	0	0	0	0	0
4194656	0	0	0	0	0	0	0	0
4194688	0	0	0	0	0	0	0	0
4194720	0	0	0	0	0	0	0	0

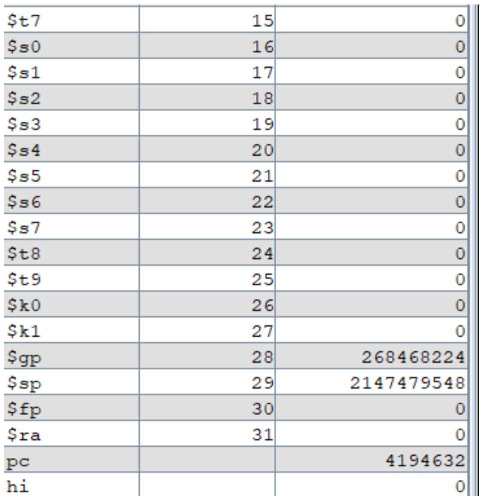
0x00400000 (.text) ☐ Hexadecimal Addresses ☐ Hexadecimal Values ☐ ASCII

➢ Registers:



Registers | Coproc 1 | Coproc 0

Name	Number	Value
$zero	0	0
$at	1	4
$v0	2	10
$v1	3	0
$a0	4	268501062
$a1	5	0
$a2	6	0
$a3	7	0
$t0	8	268697632
$t1	9	4
$t2	10	26
$t3	11	268697632
$t4	12	0
$t5	13	0
$t6	14	0
$t7	15	0



$t7	15	0
$s0	16	0
$s1	17	0
$s2	18	0
$s3	19	0
$s4	20	0
$s5	21	0
$s6	22	0
$s7	23	0
$t8	24	0
$t9	25	0
$k0	26	0
$k1	27	0
$gp	28	268468224
$sp	29	2147479548
$fp	30	0
$ra	31	0
pc		4194632
hi		0

Conclusion:

In conclusion, this report has delved into the concept of recursion and its implementation using a stack based on a linked list data structure. Through our exploration, we have gained a comprehensive understanding of both recursion and the stack data structure, as well as their interplay in algorithmic design.

Recursion, a powerful problem-solving technique, allows functions to call themselves in a recursive manner, simplifying complex problems into smaller, more manageable subproblems. However, it also introduces challenges such as stack overflow and inefficient memory usage. To mitigate these issues, we employed a stack data structure, implemented using a linked list, to manage recursive function calls and their associated memory.

The stack, in conjunction with recursion, facilitates the execution of recursive algorithms by storing intermediate results and function call information. By utilizing a linked list to implement the stack, we ensure dynamic memory allocation and efficient insertion and deletion of elements.

Throughout this report, we have explored various recursive algorithms, including factorial computation, Fibonacci sequence generation, and binary tree traversal, all of which exemplify the versatility and utility of recursion in algorithm design.

In summary, recursion, when coupled with a stack implemented using a linked list, provides a powerful framework for solving a wide range of problems efficiently and elegantly. By understanding the principles and intricacies of recursion and its implementation, we can leverage its full potential in algorithmic problem-solving and software development.

Task assignment:

Nourhan Farag	**Push function in C and Assembly**
Malak Mounir	**Push function in C and Assembly**
Mohannad Mohamed	**Pop function in C and Assembly**
Abdelrahman Afify	**Peek function in C and Assembly**
Abdullah Mohamed	**Main function & Menu in C and Assembly**

All the group participated in the implementation of the code and designing project report and presentation with 20%.