

# TP1: Remote Java Virtual Machine


Dado que por el paro de transporte público no pudo ser posible la entrega física del TP, muestro por medio de una foto las correcciones a realizar detalladas por Ezequiel Werner.

### 75:42 - Taller de Programación I

Ejercicio N° 1 Padrón 96453

Alumno Sebastián Ripori

Firma Sebastián D. Ripori

Nota:	Reentrega	Corrige:	WERNER, Ezequiel M. 	Entrega #1
<ul style="list-style-type: none"><li>- Hacer un diagrama</li><li>- Mover los loops de send/recv al TDA socket</li><li>- Usar send en vez de write y pasarle MSG_NOSIGNAL, investigar para qué sirve esa flag.</li><li>- No usar sizeof en la serialización</li></ul>				Fecha de entrega
				11/09/2018
				Fecha de devolución
				18/09/2018

Nota:		Corrige:		Entrega #2
-------	--	----------	--	------------

## TDAs:

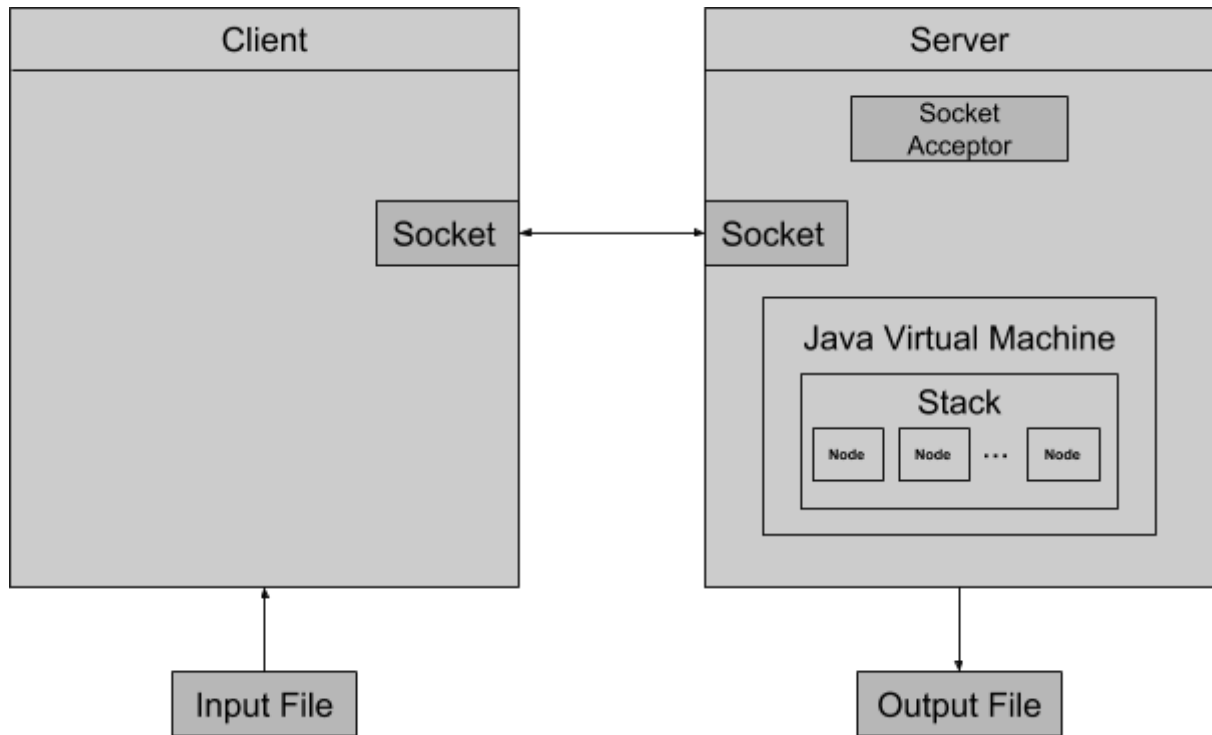
- java\_virtual\_machine
- socket
- stack
- node

## Modularización:

- server
- client

# Esquema de diseño

Es un diagrama para reconocer rápidamente los distintos componentes que integran la solución, y donde están ubicados.



El programa corre bajo un único ejecutable llamado **tp** en el cual cumple dos roles, el de **cliente** o el de **servidor**.

Echemos un vistazo al main.c:

```
int main(int argc, char* argv[]) {
    if (argv[1] == NULL) return ERROR;

    if (strcmp(argv[1], "client") == SUCCESS) return client(argv);
    else if (strcmp(argv[1], "server") == SUCCESS) return server(argv);
    else return ERROR;
}
```

Aquí se ve como segun el parametro que le pasemos al ejecutable tp, sea server o client, el programa cumplira estos respectivos papeles.

- En el caso de **cliente**: Abrimos el input file, puede ser un file, o puede ser la entrada estándar. Y si todo sale bien procedemos a la parte de sockets. Que consiste en crear un socket que apunte a hostname y puerto especificada como parámetros por el usuario. Esta conexión es con otra instancia del ejecutable tp corrida como servidor. Una vez realizada esta conexión con el server. Procedemos a mandar el entero con signo que especifica la cantidad de variables a crear en la JVM remota. Y luego mediante un bucle while enviamos los bytecodes, byte por byte hasta que llegamos al fin de archivo. Una vez aquí apagamos la escritura del socket y nos quedamos esperando la respuesta del server. Una vez llegada esta respuesta apagamos la lectura. Y liberamos la memoria correspondiente.
- En el caso de **server**: Creamos el socket aceptador que va a esperar que llegue una conexión por parte de un cliente, cuando esto ocurra se creará un socket. Para finalmente estar en contacto con el cliente. Luego procedemos a recibir el entero que dirá cuántas variables crear en la JVM. Y entonces mediante un bucle while procedemos a recibir todos los bytecodes y a ejecutarlos en la JVM. Cuando el server deja de recibir bytecodes, procede a enviar al cliente la respuesta final de como quedaron las variables en la JVM. Y luego de esto realiza los apagados del socket aceptador y del socket server. Por último liberamos los recursos.

## Modularización:

- La idea de haber hecho la separación del cliente en un archivo y el server en otro es para modularizar el código.

## TDAs:

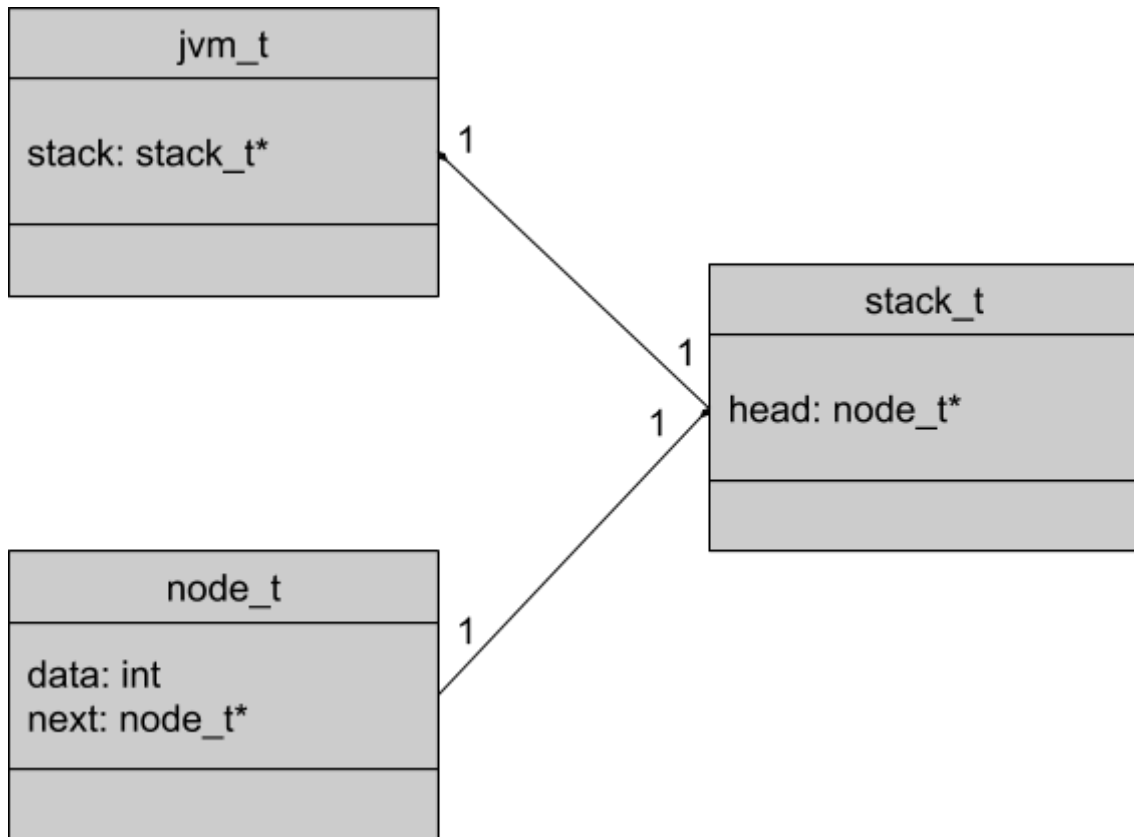
- En cuanto a los TDAs están documentados que realizan cada función. No tiene demasiada complejidad. Son funciones muy simples.
  - El stack está implementado como una linkedlist de nodos.
- 

### **Las siguientes funciones pertenecientes al TDA socket:**

socket\_send\_int  
socket\_send\_char  
socket\_rcv\_int  
socket\_rcv\_char  
socket\_rcv\_unsigned\_char

Cuando detectan un error, proceden a apagar el socket y a liberar sus recursos. Esto se hizo con el fin de que en el código del cliente y del server no se tenga que estar llamando a shutdown y destroy del socket en caso de error, por todos lados. Y así evitar código duplicado.

## Diagrama de Clases



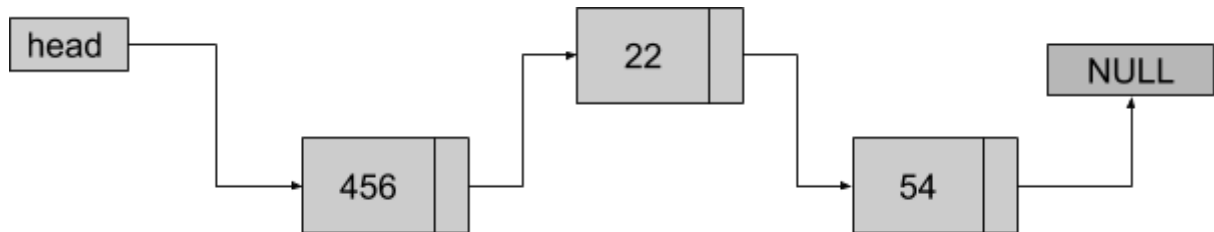
Es interesante ver cómo se relacionan estos tres TDAs, la `java_virtual_machine` posee un `stack` y este a su vez posee un nodo (raíz), y este nodo posee un puntero al siguiente nodo.

# Stack

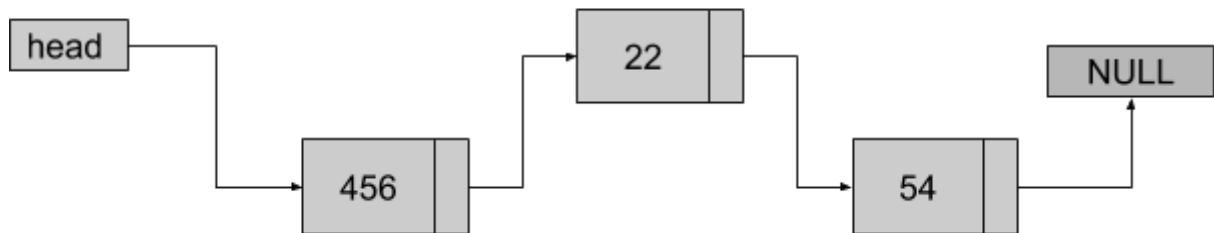
A continuación se detalla mediante dibujos como funcionan los dos métodos más complejos del TDA Stack, estos son `stack_push` y `stack_pop`.

## `stack_push(stack, 93)`

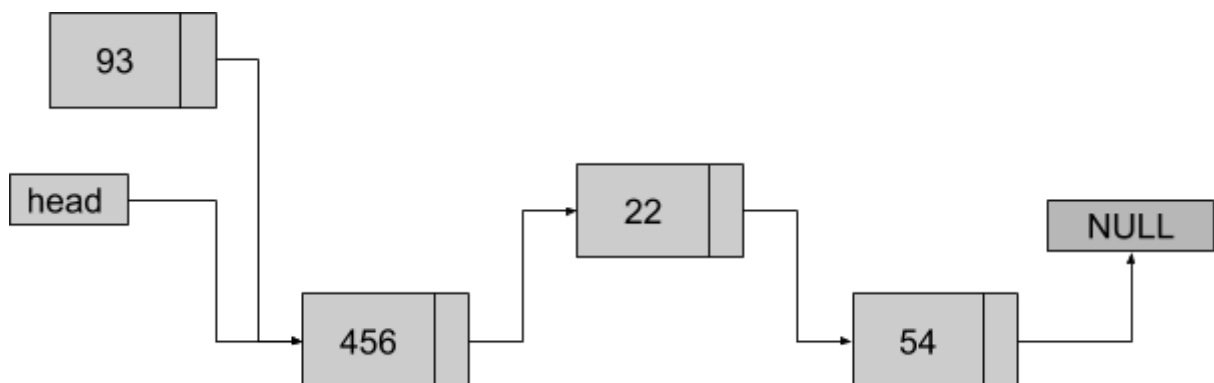
Supongamos que tenemos el siguiente stack:



Y se llama a la función `stack_push` con el parámetro 93. En primera instancia esta función crea un nuevo nodo le pasa el valor 93.

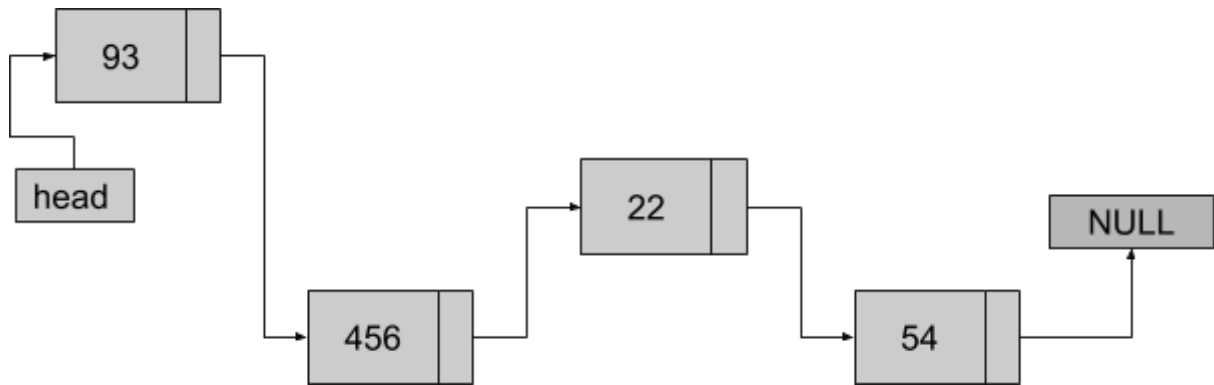


Luego al nuevo nodo se le setea como “nodo siguiente” el primer nodo de la pila.



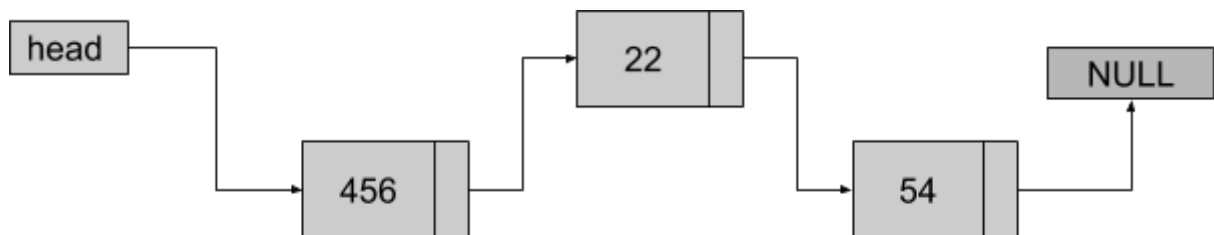


Y por último se actualiza la variable head, para que apunte al nuevo nodo. Notar que todo tiene sentido, la variable head del TDA Stack apunta al último nodo que se agregó, y este apunta a que era el head.

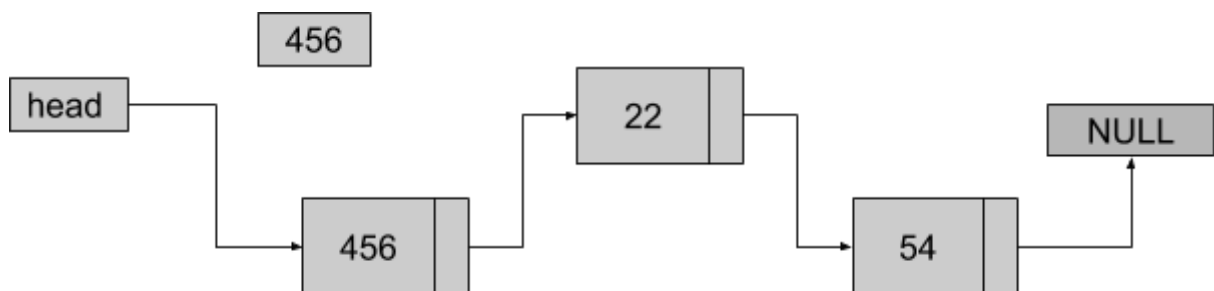


## stack\_pop(stack)

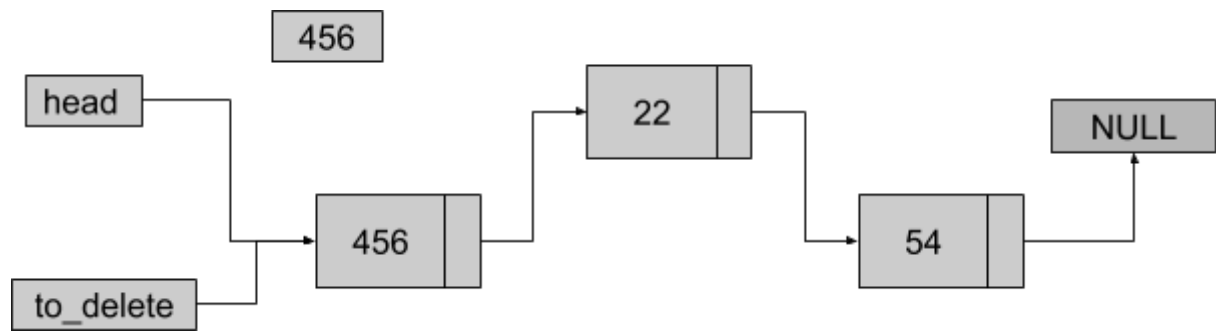
Suponiendo que tenemos el mismo stack que antes:



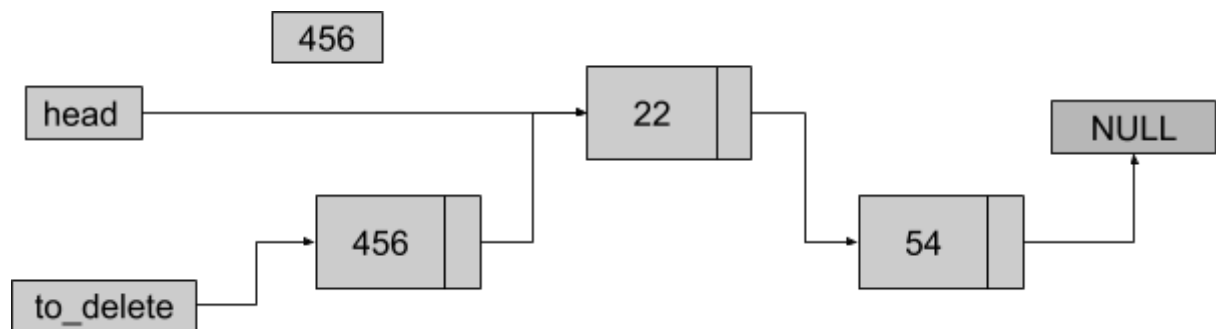
Nos guardamos en una variable el dato a retornar.



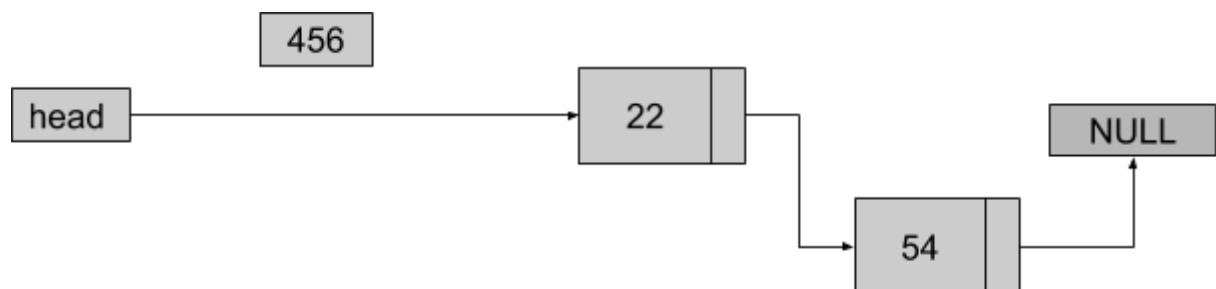
Creamos un puntero que apunte al nodo a eliminar.



Hacemos que la variable head ahora apunte al siguiente del que antes era el head.



Y por último borramos el nodo apuntado por la variable to\_delete



La función retorna 456 y ahora el nuevo head de la pila es el nodo que guarda el valor 22.