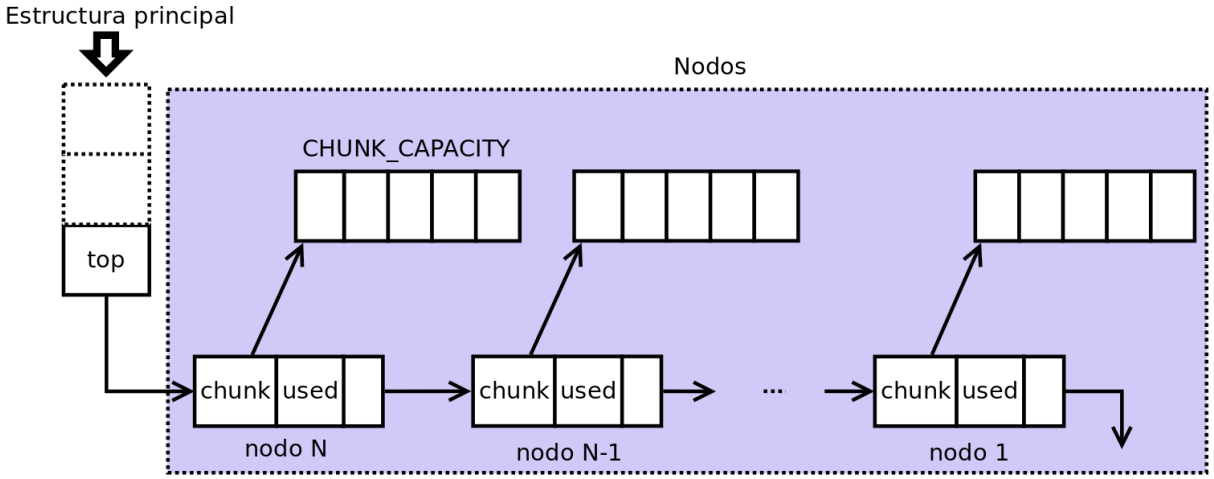
Córdoba, 29 de Julio del 2022

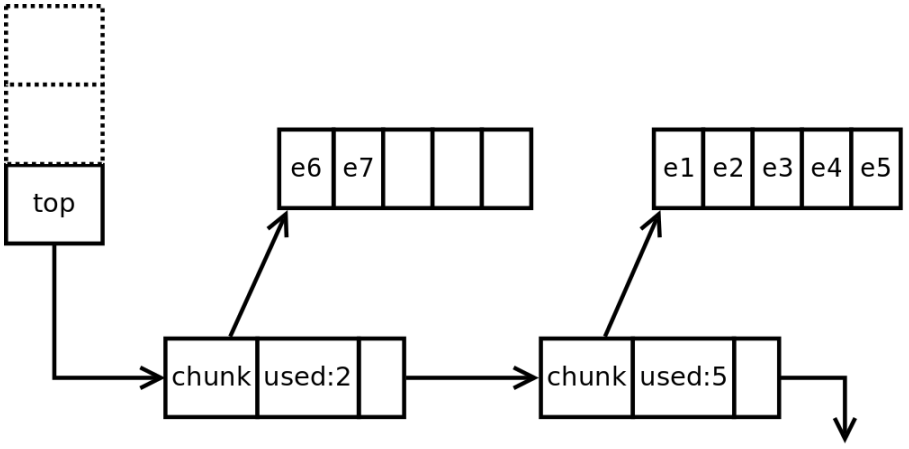
Examen Final

Algoritmos y Estructuras de Datos II - Taller

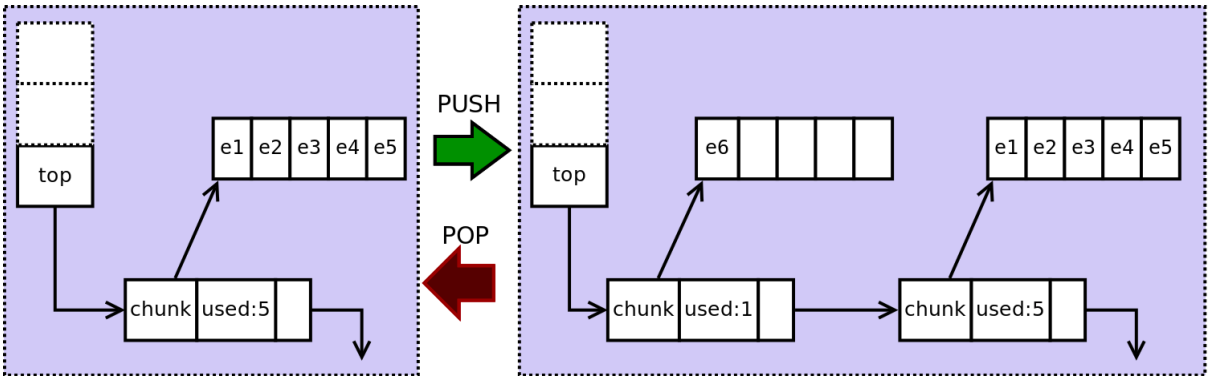
Las listas (simplemente) enlazadas son una estructura de datos muy flexible y las utilizamos para implementar gran cantidad de tipos abstractos. Tienen la desventaja de no ser tan eficientes en cuanto a la velocidad de acceso a los elementos ni al uso de la memoria (esto último lo podrán ver en Sistemas Operativos). En este examen se debe implementar un TAD Pila usando una representación que es un híbrido entre arreglos y listas enlazadas:



Como se puede observar, en esta representación cada nodo guarda varios elementos. De hecho se pueden guardar hasta `CHUNK_CAPACITY` elementos en cada nodo. El campo `chunk` apunta a un arreglo que es donde se guardan los elementos, y el campo `used` indica cuántos elementos tiene almacenado ese nodo. Por último el nodo tiene un campo `next` que apunta al siguiente nodo de la lista. La representación tiene una estructura principal que posee un campo `top` que apunta siempre al nodo que se agregó más recientemente, el cual contiene el tope de la pila. Un ejemplo de una pila con `CHUNK_CAPACITY=5` donde se agregaron 7 elementos `e1`, `e2`, `e3`, …, `e7` (se agregaron en ese orden) se muestra a continuación:



Notar que el elemento que está al tope de la pila es `e7`. Cada vez que se agrega un nuevo elemento a la pila, se lo agrega al final del arreglo `chunk` del nodo apuntado por `top`. Si al intentar agregar el nuevo elemento resulta que el `chunk` del nodo apuntado por `top` está lleno, se debe agregar un nuevo nodo y ubicar el elemento allí. Cuando se saca un elemento de la pila, si resulta que el nodo queda vacío se debe eliminar el nodo. Ambas situaciones se ilustran en la siguiente figura:



Una propiedad de esta representación para el TAD Pila es que los nodos no pueden estar vacíos, siempre tienen elementos guardados. Además, el único nodo que puede tener espacio disponible es el que está apuntado por `top`, los demás deben estar llenos.

Ejercicio 1

Las funciones a implementar del TAD Pila son las siguientes:

Función	Descripción
`stack stack_empty()`	Crea una pila vacía
`stack stack_push(stack s, stack_elem e)`	Inserta el elemento `e` al tope de la pila `s`
`stack stack_pop(stack s)`	Remueve el tope de la pila. Sólo aplica a una pila no vacía.
`unsigned int stack_size(stack s)`	Obtiene el tamaño de la pila. Debe ser de orden constante.
`void stack_top(stack s, stack_elem *top)`	Obtiene el tope de la pila y lo coloca en `*top`. Sólo aplica a una pila no vacía.
`bool stack_is_empty(stack s)`	Verifica si la pila está vacía.
`stack_elem *stack_to_array(stack s)`	Crea un arreglo en memoria dinámica con todos los elementos de la pila. El tope de la pila debe quedar en el último elemento del arreglo. Si la pila está vacía, devuelve `NULL`.
`stack stack_destroy(stack s)`	Libera toda la memoria usada por la pila.

Se deben completar además las definiciones para la estructura principal y para la estructura de los nodos. En la estructura principal se pueden incluir los campos que se necesiten para que `stack_size()` pueda calcularse en orden constante.

Para probar la implementación se incluye el archivo **`test-stack.c`** que construye una pila de caracteres a partir del parámetro que se pasa al ejecutar el programa. Para compilar el test se puede usar el **`Makefile`** incluído haciendo:

```
$ make test
```

y luego, para ejecutar el programa de prueba:

```
$ ./test "hola"
```

Como resultado debería obtenerse:

```
[stack_size():  0, stack_is_empty(): true ] pushing: 'h' [stack_size():  1, stack_is_empty(): false]
[stack_size():  1, stack_is_empty(): false] pushing: 'o' [stack_size():  2, stack_is_empty(): false]
[stack_size():  2, stack_is_empty(): false] pushing: 'l' [stack_size():  3, stack_is_empty(): false]
[stack_size():  3, stack_is_empty(): false] pushing: 'a' [stack_size():  4, stack_is_empty(): false]

stack_to_array(): |hola|

[stack_size():  4, stack_is_empty(): false] popping [stack_size():  3, stack_is_empty(): false]
[stack_size():  3, stack_is_empty(): false] popping [stack_size():  2, stack_is_empty(): false]
[stack_size():  2, stack_is_empty(): false] popping [stack_size():  1, stack_is_empty(): false]
[stack_size():  1, stack_is_empty(): false] popping [stack_size():  0, stack_is_empty(): true ]
```

A la izquierda se muestra el estado de la pila antes de realizar la acción (*push* o *pop*) y a la derecha el estado resultante. Se puede modificar a gusto el archivo **`test-stack.c`** pudiendo mostrar más información que puede ser útil para *debug* (por ejemplo el tope de la pila en caso que nos sea vacía). Quizás en una primera instancia convenga comentar la ejecución de `stack_to_array()` hasta que se la implemente de manera confiable.

Ejercicio 2

Completar **`check_balance.c`** y utilizar la pila para verificar si una expresión tiene los paréntesis balanceados. El algoritmo consiste en:

1. Ignorar los caracteres distintos a **`'('`** y **`')'`**
2. Cada vez que se encuentra un caracter **`'('`** se lo debe agregar a la pila
3. Cuando se encuentre un caracter **`')'`** se debe verificar que en el tope de la pila se encuentre un caracter **`'('`**. Si ese es el caso, se debe quitar el elemento de la pila y seguir la verificación. Si la pila está vacía o en el tope no está el caracter **`'('`** la expresión no está balanceada y se termina la verificación.
4. Si al finalizar la verificación la pila se encuentra vacía, entonces se concluye que la expresión está balanceada. Si al finalizar quedan elementos en la pila, entonces la expresión no está balanceada.

Para compilar el ejercicio se puede usar el **`Makefile`** incluido de la siguiente manera:

```
$ make
```

Luego el programa se ejecuta como sigue:

```
$ ./check_balance "((x + y)*(x - y))"
```

Para este caso se debería obtener la salida

```
The given expression is balanced
```

Otros ejemplos:

```
$ ./check_balance "()()"
The given expression is balanced

$ ./check_balance ""
The given expression is balanced

$ ./check_balance "(hola)(  )()"
The given expression is balanced

$ ./check_balance "(()"
The given expression is not balanced

$ ./check_balance ")("
The given expression is not balanced

$ ./check_balance "(()))"
The given expression is not balanced
```

Consideraciones

- Si **`stack.c`** no compila, no se aprueba el examen.
- Si **`check_balance.c`** no compila muy difícilmente se apruebe el examen.
- Si **`stack_size()`** no es de orden constante baja muchísimos puntos
- No implementar la invariante baja puntos
- No chequear pre y post condiciones baja puntos
- Los *memory leaks* bajan puntos
- Entregar código muy improlijo puede restar puntos
- Para sacar **P** en el examen **se debe** hacer una invariante no trivial.
- Se provee el archivo **`Makefile`** para facilitar la compilación.
- Se recomienda usar las herramientas **`valgrind`** y **`gdb`**.