Listas Enlazadas:
Aplicaciones

ALGORITMOS Y ESTRUCTURA DE DATOS

Cola de Prioridad

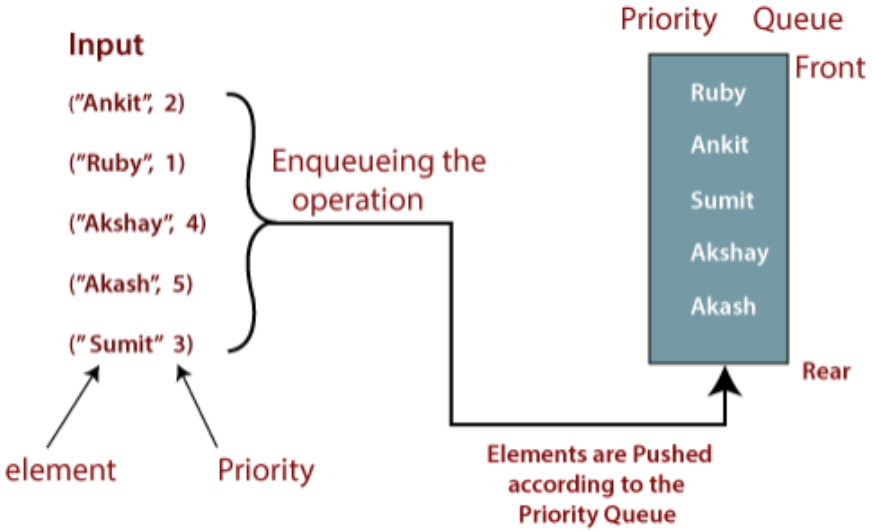
Cola de Prioridad

Es una estructura de datos similar a la cola en donde cada elemento tiene asociado un valor de prioridad. Esto para permitir que los elementos de mayor prioridad sean desencolados primero.

- **push():** añade un elemento y se ubica en la cola según su prioridad.

- **pop():** retira el elemento con mayor prioridad.

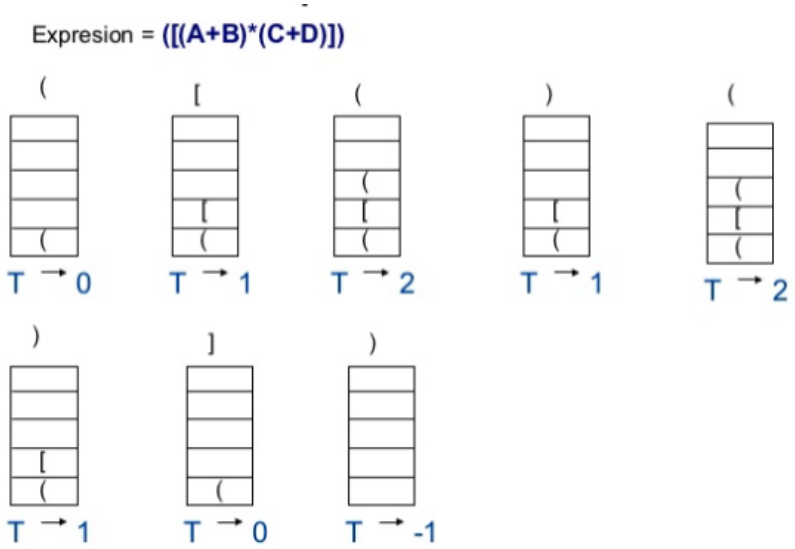
- **top():** retorna el elemento con mayor prioridad.



Evaluación de Paréntesis

Evaluación de Paréntesis

Antes de resolver una expresión matemática, es necesario evaluar la correcta ubicación de los paréntesis. Podemos usar para ello las operaciones de una Pila.



Condiciones para que los paréntesis estén correctos:

- Que la pila quede vacía.

- Que no quede ningún paréntesis con cierre pendiente.

Resolver Expresiones Aritméticas

Resolver Expresiones Aritméticas

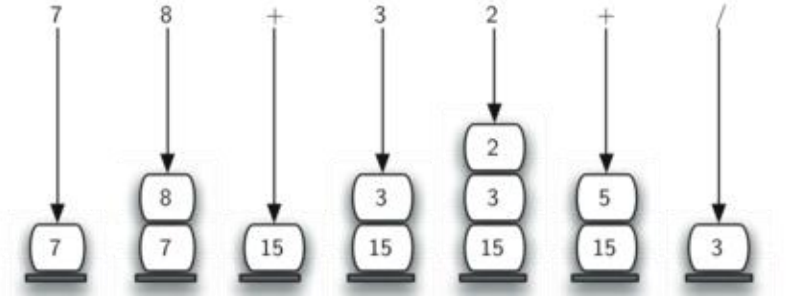
Una expresión aritmética contiene constantes, variables y operaciones con distintos niveles de precedencia. Por ejemplo: **(7 + 8)/(3 + 2)**

¿Cómo podemos diseñar un programa que sea capaz de resolver de forma adecuada dicha expresión?

- Una solución muy utilizada es convertir primero la expresión a su forma postfija:

(7 + 8)/(3 + 2) infija	7 8 + 3 2 + / postfija

- Luego, usando una pila podemos resolver de la siguiente manera:



Implementar el programa completo.
Guía de solución.

De infija (X) a postfija (Y)

1. Push “(“onto Stack, and add “)” to the end of X.
2. Scan X from left to right and repeat Step 3 to 6 for each element of X until the Stack is empty.
3. If an operand is encountered, add it to Y.
4. If a left parenthesis is encountered, push it onto Stack.
5. If an operator is encountered ,then:
    1. Repeatedly pop from Stack and add to Y each operator (on the top of Stack) which has the same precedence as or higher precedence than operator.
    2. Add operator to Stack.
    [End of If]
6. If a right parenthesis is encountered ,then:
    1. Repeatedly pop from Stack and add to Y each operator (on the top of Stack) until a left parenthesis is encountered.
    2. Remove the left Parenthesis.
    [End of If]
    [End of If]
7. END.

Self Organizing List

Para búsquedas frecuentes

Self Organizing List

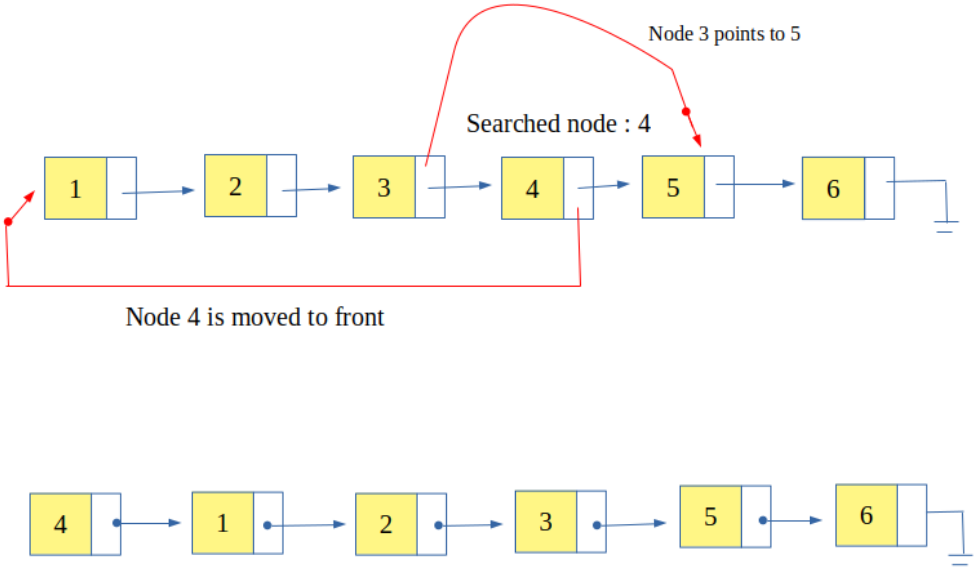
Es un tipo de lista que se va organizando mientras se solicitan elementos

Su peor caso es O(n) en la búsqueda, cuando el elemento es el último

Se reordena basada en 3 estrategias:

- Move-to-Front Method
- Transpose Method
- Count Method

Move to front



Move to front

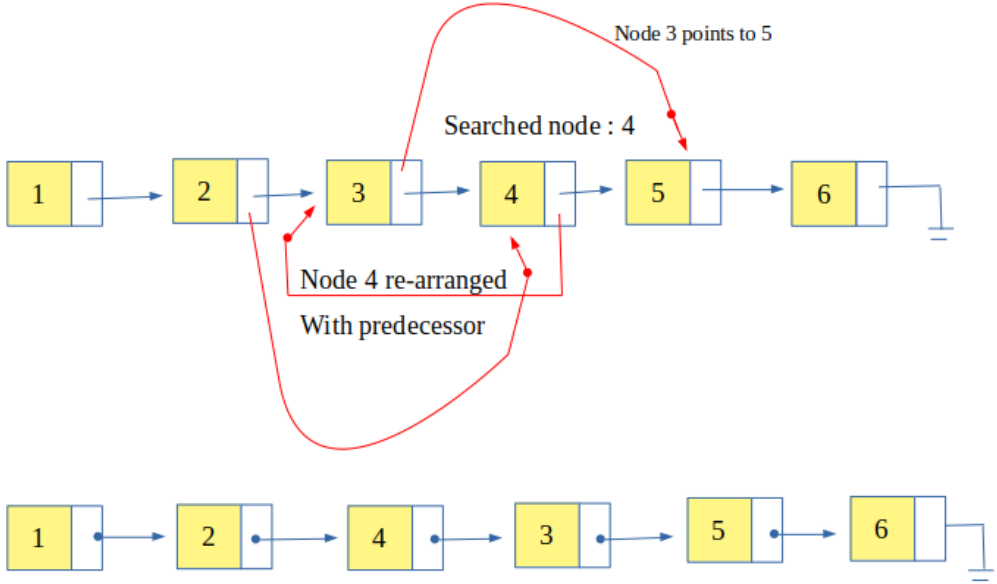
Pros:

Este método se adapta rápido y es fácil de implementar

Cons:

Puede mover al frente (prioriza) elementos que sean poco accedidos

Transpose method



Transpose method

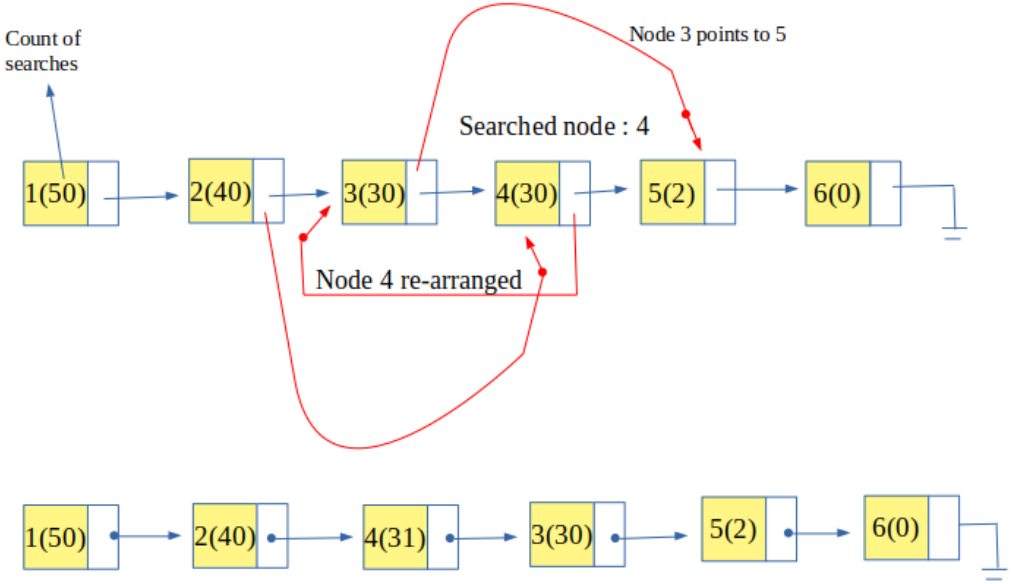
Pros:

- Fácil de implementar y requiere poca memoria
- Mantiene nodos accedidos de manera frecuente al frente

Cons:

- Requiere muchos accesos para enviar un elemento al frente

Count method



Count method

Pros:

Refleja el acceso a elementos de manera más real

Cons:

Requiere un poco más de memoria, y no se adapta rápido a nuevos patrones de acceso

Aplicaciones

Inteligencia artificial

- Redes neuronales

Cache en algoritmos Last Frequently Used (LFU)

Compiladores e interpretadores

Stacks

Queues

Free lists: lo que usa `malloc` por debajo y algunas bases de datos (más de eso en Base de Datos II)

En Linux, cuando uno hace, por ejemplo, `malloc(4 * sizeof(char))`, el puntero que se devuelve es el de la imagen de abajo:

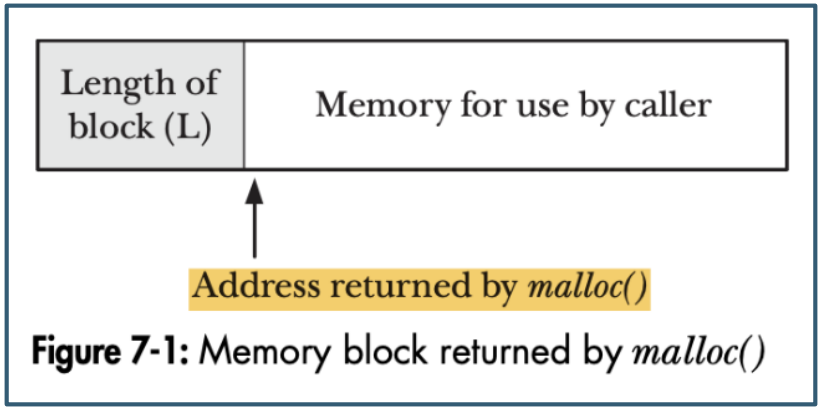


Figure 7-1: Memory block returned by *malloc()*

Es decir, el bloque de memoria retornado, no solo tiene la memoria que pedimos ("Memory for use by caller"), sino también tiene unos bytes reservados *antes* del puntero que indican el tamaño total del bloque.

Cuando llamamos a `free` para liberar el bloque de memoria reservado por `malloc`, este bloque se inserta en el *free list*: un doubly-linked list que tiene todos los bloques de memoria **que han sido reservados por** `malloc` **y que luego fueron liberados con** `free`**.**

When a block is placed on the (doubly linked) free list, *free()* uses the bytes of the block itself in order to add the block to the list, as shown in Figure 7-2.

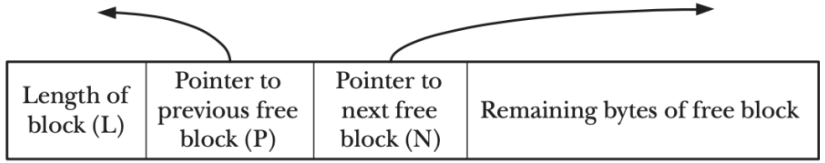
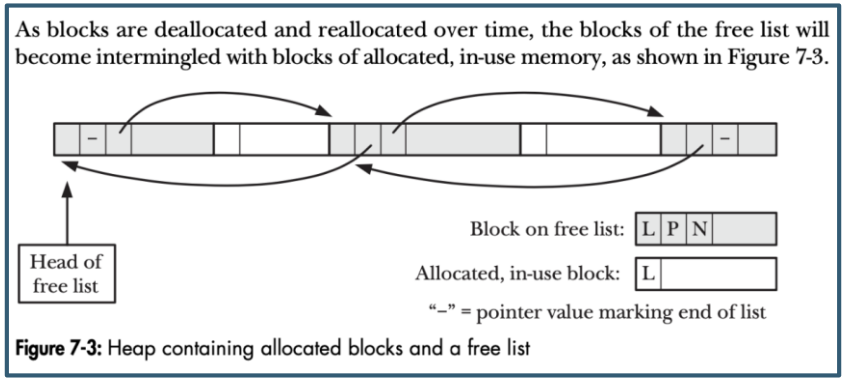


Figure 7-2: A block on the free list

Ese es **uno** de los factores por los cuales acceder a memoria en el heap *puede ser* un poco más lento que en el stack: cuando hacemos `malloc` (al menos en Linux) se tiene que iterar sobre el free list hasta encontrar el bloque que satisfaga el pedido de `malloc` (hay distintos algoritmos para saber qué bloque elegir, algunos son: *first-fit* y *best-fit*). Sin embargo, eso no es razón para nunca más usar el heap, sino una consideración a tomar en cuenta.

Al entrelazar llamadas a `malloc` y a `free`, se podría llegar a la situación de la imagen de abajo.



L: tamaño en bytes del pedazo de memoria reservado/libre
P: puntero al bloque `prev` de la *free list*
N: puntero al bloque `next` de la *free list*
-: `nullptr`

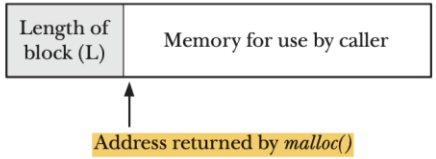


Figure 7-1: Memory block returned by *malloc()*

When a block is placed on the (doubly linked) free list, *free()* uses the bytes of the block itself in order to add the block to the list, as shown in Figure 7-2.

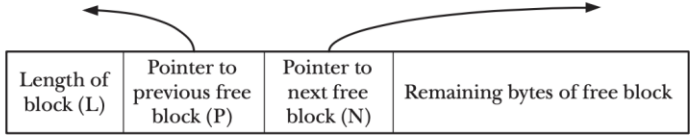


Figure 7-2: A block on the free list

As blocks are deallocated and reallocated over time, the blocks of the free list will become intermingled with blocks of allocated, in-use memory, as shown in Figure 7-3.

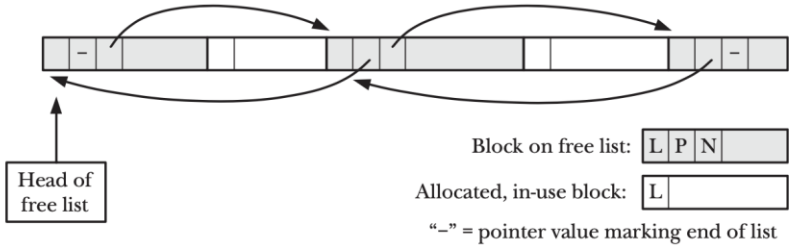


Figure 7-3: Heap containing allocated blocks and a free list

- **¿Por qué creen que en este caso se usa un doubly-linked list y no un forward list?**
- **¿Por qué no un array contiguo de bloques libres?**