75.04 Algoritmos y Programación II

Práctica 3: estructuras de datos simples

Notas preliminares

- En esta práctica ejercitaremos el manejo de estructuras de datos lineales: listas, pilas y colas.
- Los ejercicios marcados con el símbolo ♣ constituyen un subconjunto mínimo de ejercitación. No obstante, recomendamos fuertemente realizar todos los ejercicios.

---

1. Listas

Ejercicio 1

(a) Implementar en C++ una clase para representar listas doblemente enlazadas. Usar memoria dinámica y (al menos) las siguientes operaciones:

- `bool vacia()`,
- `bool llena()`,
- `void insertar(const T &)` -inserta un elemento al comienzo de la lista-,
- `void agregar(const T &)` -inserta un elemento en el extremo final-, y
- `T *buscar(const T &)`.

Además, diseñar e implementar un conjunto de pruebas de regresión de de la clase, a fin de hacer validaciones en el momento de hacer cambios en el código.

(b) Repetir el ejercicio anterior para una implementación estática, implementando la clase `lista<N, T>`, en donde "`N`" es el tamaño máximo de la lista y "`T`" el tipo de los elementos contenidos. Sugerencia: usar templates.

Ejercicio 2 ♣

Utilizando las clases del ejercicio anterior,

(a) Programar un método que borre el i-ésimo nodo de la lista, si es que existe.

(b) Programar un método para intercambiar los primeros dos elementos de la lista, teniendo en cuenta **todos** los casos posibles.

Ejercicio 3

Escribir un programa que mezcle dos listas de enteros ordenadas en una única instancia de la clase. La función `mezclar()` debe recibir dos referencias a cada uno de los objetos a mezclarse, y devolver un objeto que contenga la lista mezclada.

Ejercicio 4

(a) Proponer una clase C++ para representar anillos doblemente enlazados (i.e., listas circulares).

(b) A partir de lo anterior, escribir una primitiva `swap` que reciba e intercambie dos nodos de la lista circular. Tener en cuenta **todos** los casos de borde posibles.

Ejercicio 5 ♣

Utilizando las clases desarrolladas en los ejercicios previos, escribir funciones que devuelvan la suma de los números de:

(a) una lista circular, y

(b) una lista doblemente enlazada.

Se sugiere, para ello, implementar iteradores para sendos contenedores.

Ejercicio 6

Escribir un programa para hacer sumas, restas, multiplicaciones, y divisiones de enteros arbitrariamente largos; cada entero puede ser representado por una lista de dígitos. Nota: algunas operaciones requieren desplazamientos hacia adelante y hacia atrás en las listas, por lo que se recomienda usar listas doblemente enlazadas.

Ejercicio 7

Encontrar una representación de listas circulares que permita ser recorrida eficientemente en ambos sentidos, pero usando un único puntero (en realidad, enlace) por nodo. Ayuda: dados dos punteros a nodos consecutivos $\{x[i-1], x[i]\}$, debería ser posible obtener tanto $x[i+1]$ como $x[i-2]$.

2. Pilas y colas

Ejercicio 8

Dada una secuencia S de operaciones push y pop en una pila inicialmente vacía, S es una secuencia válida si se cumple:

- no intenta desapilar elementos en una pila vacía;
- la pila esta vacía al final de la secuencia.

Diseñar un conjunto de reglas para generar una secuencia válida.

Ejercicio 9 ♣

Dados los caracteres (), [] y {}, y una cadena s, decimos que s está *balanceada* si tiene alguno de estos formatos:

- $s =$ "" (string nulo)
- $s = (t)$
- $s = [t]$
- $s = \{t\}$
- $s = tu$

en donde t y u son cadenas balanceadas (en otras palabras, para cada paréntesis, llave o corchete abierto existe un caracter de cierre correspondiente). Por ejemplo, $s =$ "$\{()[()]\}$" está balanceada.

Escribir un programa que use una pila de caracteres para determinar si una cadena está balanceada.

Ejercicio 10

(a) Escribir una función que acepte un *string* como argumento y devuelva un booleano indicando si el string ingresado es un palíndromo. Ejemplos: “Yatay”, “Neuquen”.

(b) Modificar el programa anterior para que acepte frases palíndromas, ignorando caracteres de puntuación. Ejemplos: “A man, a plan, a canal - Panama”, “Dammit, I’m mad”, ”Son robos, no solo son sobornos”.

Ejercicio 11

(a) Diseñar un algoritmo para traducir una expresión *postfix* en *infix*, usando un stack de expresiones *infix*, representadas como instancias de la clase string.

(b) Diseñar un esquema para convertir una expresión *prefix* en *postfix*. *Ayuda*: procesar un símbolo por vez, imprimir los operandos inmediatamente, pero guardar los operadores en un stack hasta que sean requeridos.

Ejercicio 12 ♣

Escribir una implementación de colas donde se use un arreglo para almacenar los elementos, y con las siguientes características adicionales:

- si el arreglo se llena al encolar, se duplica su tamaño;
- el arreglo se acorta a la mitad de la longitud cuando esté a menos de medio llenar.

Mostrar que tanto *encolar* como *desencolar* toman, en promedio, tiempo constante (i.e., que no depende de la cantidad de elementos presentes en la cola).

Hint: considerar el tiempo que lleva encolar $n = 2^k$ elementos en la cola.

Ejercicio 13 ♣

Supongamos que, en cierto programa, sólo se permite usar pilas como estructura de datos. ¿Cómo se puede implementar *eficientemente* una cola con dos pilas?

Ejercicio 14

Una *input-restricted dequeue* es una lista en la que los elementos pueden insertarse sólo desde un extremo pero pueden ser removidos de ambos extremos. Claramente, tal estructura puede comportarse como una pila tanto como una cola, si decidimos remover cada elemento de un extremo fijo. Una *output-restricted dequeue* puede definirse de manera análoga: se permite insertar desde cualquier extremo pero remover sólo de uno. ¿Puede también esta estructura comportarse como una pila y como una cola?

Ejercicio 15

El algoritmo recursivo de ordenamiento QUICKSORT,

```
QUICKSORT_RECURSIVE: procedure expose A.
parse arg L, R
if L < R then do
        Q = PARTITION(L, R)
        call QUICKSORT L, Q
        call QUICKSORT Q + 1, R
end

PARTITION: procedure expose A.
parse arg L, R
X = A.L; I = L - 1; J = R + 1;
do forever
        do until A.J <= X; J = J - 1; end
        do until A.I >= X; I = I + 1; end
        if I < J then do; W = A.I; A.I = A.J; A.J = W; end
        else return J
end
```

puede transformarse en iterativo usando una pila,

```
QUICKSORT_ITERATIVE: procedure expose A.
parse arg L, R
push L R
do while QUEUED() > 0
        pull L R
        if L < R then do
                Q = PARTITION(L, R)
                push L Q; push Q + 1 R
        end
end
```

Implementar ambas versiones (iterativa y recursiva) y hacer corridas de prueba para distintos sets de entrada con el objetivo de analizar cuán distinto es el tiempo de corrida para cada versión. A partir de los resultados, analizar si siempre un algoritmo iterativo se comportará mejor/peor que su contraparte recursiva.