Práctica 7

Tipos Abstractos de Datos

Introducción a la Computación

1^{er} cuatrimestre 2019

TIPOS ABSTRACTOS DE DATOS

Ejercicio 1. Considérese el TAD `Fecha`, que tiene las siguientes operaciones:

TAD FECHA

- `F.FechaSiguiente()` $\rightarrow$ `Fecha`:
  Devuelve la fecha siguiente a `F` (ej: al 31/12/1999 le sigue el 1/1/2000).
- `F.Menor`$(f_2) \rightarrow \mathbb{B}$
  Devuelve TRUE si la fecha `F` es anterior a la fecha f_2, y FALSE en caso contrario.

Se pide dar un algoritmo para determinar la cantidad de días que hay entre dos fechas dadas.

Ejercicio 2. Considérese el TAD `Pila(Char)` que define las siguientes operaciones, todas implementadas en $O(1)$:

TAD PILA(CHAR)

- `CrearPila()` $\rightarrow$ `Pila(Char)`: Crea una pila vacía.
- `P.EstáVacía()` $\rightarrow \mathbb{B}$: Devuelve TRUE si P no contiene elementos y FALSE en caso contrario.
- `P.Apilar`(x): Apila x en el tope de P.
- `P.Desapilar()`: Desapila el tope de P. Pre: $\neg$*P.EstáVacía*().
- `P.Tope()` $\rightarrow$ `Char`: Devuelve el elemento que está en el tope de P. Pre: $\neg$*P.EstáVacía*().

Se pide dar algoritmos para los siguientes problemas:

(a) Determinar en si un String dado `S` está *bien balanceado* con respecto a los caracteres { }, [ ], ( ) en $O(|\texttt{S}|)$. Ejemplos: "{a(b)x[()]}" está bien balanceado; "}", "a(b))", "[[})" y "([)]" no están bien balanceados.

(b) Determinar si un String dado `S` que contiene una y sólo una aparición del caracter '`#`' es o no capicúa en $O(|\texttt{S}|)$.

(c) Determinar si un String dado `S` que contiene únicamente ceros y unos tiene la misma cantidad de ceros que de unos en $O(|\texttt{S}|)$. El algoritmo no debe usar variables numéricas.

Ejercicio 3. Considérese el TAD `ÁrbolBinario(`$\mathbb{Z}$`)`, que tiene las siguientes operaciones, todas implementadas en $O(1)$:

TAD ÁRBOLBINARIO($\mathbb{Z}$)

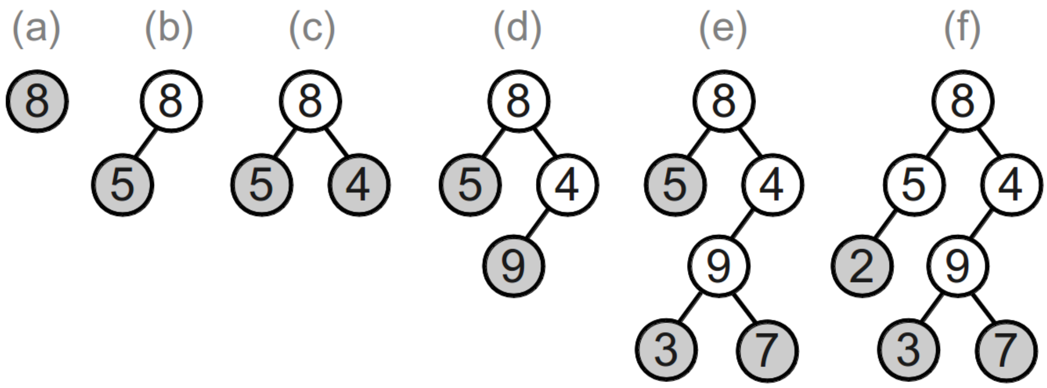
- `A.Raíz()` $\to \mathbb{Z}$: Devuelve el entero almacenado en la raíz del árbol binario A.
- `A.HayIzq?()` $\to \mathbb{B}$: Dice si el árbol binario A tiene subárbol izquierdo.
- `A.HayDer?()` $\to \mathbb{B}$: Dice si el árbol binario A tiene subárbol derecho.
- `A.Izq()` $\to$ ÁrbolBinario($\mathbb{Z}$): Devuelve el subárbol izquierdo de A. (Pre: `A.HayIzq?()`)
- `A.Der()` $\to$ ÁrbolBinario($\mathbb{Z}$): Devuelve el subárbol derecho de A. (Pre: `A.HayDer?()`)

Se pide dar algoritmos *Divide & Conquer* para los siguientes problemas:

(a) Dado un árbol binario A, encontrar el entero más grande almacenado en $O(|A|)$, donde $|A|$ es la cantidad de nodos del árbol.

(b) Dado un árbol binario A, devuelva la suma de todos los enteros almacenados en el árbol en $O(|A|)$.

(c) Dado un árbol binario A, devuelva la **distancia desde la raíz de A hasta su hoja más cercana** en $O(|A|)$. Una "hoja" se define como un nodo sin subárboles izquierdo ni derecho (en la figura, las hojas se muestran sombreadas). En particular, esta distancia se define como 0 (cero) para un árbol sin subárboles (en la figura, el ejemplo (a)).



Ejemplos: La distancia de la raíz a la hoja más cercana es, en cada caso:
(a) 0, (b) 1, (c) 1, (d) 1, (e) 1, (f) 2.

Ejercicio 4. Sea `Lista(ℤ)` el TAD que define las siguientes operaciones:

TAD LISTA($\mathbb{Z}$)

- `CrearLista()` $\rightarrow$ `Lista(ℤ)`: Crea una lista vacía.
- `L.Agregar(`x`)`: Inserta x al final de L.
- `L.BorrarTodos(`x`)`: Borra todas las apariciones de x en L.
- `L.Reemplazar(`x, y`)`: Reemplaza en L todas las ocurrencias de x por y.
- `L.Longitud()` $\rightarrow \mathbb{Z}$: Devuelve la cantidad de elementos de la lista.
- `L.I-ésimo(`i`)` $\rightarrow \mathbb{Z}$: Devuelve el i-ésimo elemento de L.
  Precondición: $0 \leq i <$ *Longitud(L)*.
- `L.EstáVacía()` $\rightarrow \mathbb{B}$: Devuelve TRUE si la lista no contiene elementos y FALSE en caso contrario.

donde L : Lista($\mathbb{Z}$) y $i, x, y : \mathbb{Z}$.

Sean `Nodo` y `Lista` las estructuras de representación utilizadas para implementar el TAD:

```
Lista==⟨primero:Ref(TNodo)⟩
Nodo==⟨valor:ℤ, siguiente:Ref(TNodo)⟩
```

El invariante de representación de la estructura propuesta es que $Lista.primero$ apunta al primer elemento de la lista y no hay ciclos entre los nodos. Es decir, no existen nodos $n_1, n_2 \ldots, n_m$ tales que $n_1.siguiente = n2, \ldots, n_{m-1}.siguiente = n_m$ y $n_m.siguiente = n_1$.

a) Dar un algoritmo en pseudocódigo para cada una de las operaciones definidas en el TAD Lista($\mathbb{Z}$) utilizando la representación propuesta.

b) Dar un algoritmo recursivo que imprima la lista en orden inverso (suponer que se cuenta con una función `print(x)`, con $x : \mathbb{Z}$).

c) Calcular el orden de los algoritmos propuestos.

d) Implementar los algoritmos en Python.

e) Modificar la estructura propuesta y los algoritmos para las operaciones *Agregar* y *Longitud* de modo de que ambas pertenezcan a $O(1)$. ¿Qué cambios hay que hacer en el resto de los algoritmos del TAD?

Ejercicio 5. Considérese el TAD `Pila(ℤ)`, que define las mismas operaciones que el TAD del ejercicio 2 con la salvedad de que en este caso es de enteros. Sea también la siguiente estructura de representación:

```
Pila==⟨elementos:Lista(ℤ)⟩
```

donde Lista($\mathbb{Z}$) es el TAD del ejercicio 4.

a) Dar el invariante de representación para la estructura propuesta.

b) Escribir en pseudocódigo los algoritmos de las operaciones definidas en el TAD Pila($\mathbb{Z}$).

c) Implementar en Python el TAD.

d) Implementar en Python los algoritmos del ejercicio 2.

Ejercicio 6. Sea Cola($\mathbb{Z}$) el TAD que define las siguientes operaciones:

- `CrearCola`() $\rightarrow$ Cola($\mathbb{Z}$): Crea una cola vacía.
- `C.EstáVacía`() $\rightarrow \mathbb{B}$: Devuelve TRUE si C no contiene elementos y FALSE en caso contrario.
- `C.Encolar`(x): Encola x al final de la cola C.
- `C.SacarPrimero`() $\rightarrow \mathbb{Z}$: Saca de C el primer elemento y lo devuelve.
  Precondición: $\neg$*EstáVacía*(C).

donde C : Cola($\mathbb{Z}$) y $x : \mathbb{Z}$.

Sea la siguiente estructura de representación:

`Cola==`$\langle$`elementos:Lista(`$\mathbb{Z}$`)`$\rangle$

donde Lista($\mathbb{Z}$) es el TAD del ejercicio 4.

a) Dar el invariante de representación para la estructura propuesta.

b) Escribir en pseudocódigo los algoritmos de las operaciones definidas en el TAD Cola($\mathbb{Z}$).

c) Implementar en Python.

Ejercicio 7. Sea Conjunto($\mathbb{Z}$) el TAD que define las siguientes operaciones:

- `CrearConjunto`() $\rightarrow$ Conjunto($\mathbb{Z}$): Crea un conjunto vacío.
- `C.Agregar`(x): Agrega x a C.
- `C.Pertenece`(x) $\rightarrow \mathbb{B}$: Devuelve TRUE si x pertenece a C y FALSE en caso contrario.
- `C.Tamaño`() $\rightarrow \mathbb{Z}$: Devuelve el cardinal del conjunto.
- `C.EstáVacío`() $\rightarrow \mathbb{B}$: Devuelve TRUE si C no contiene elementos y FALSE en caso contrario.
- `C.ListarElementos`() $\rightarrow Lista(\mathbb{Z})$: Devuelve una lista con todos los elementos del conjunto.

donde C : Conjunto($\mathbb{Z}$) y $x : \mathbb{Z}$.

Sea la siguiente estructura de representación:

`Conjunto==`$\langle$`elementos:Lista(`$\mathbb{Z}$`)`$\rangle$

donde Lista($\mathbb{Z}$) es el TAD del ejercicio 4. Suponer que el invariante de representación para esta estructura es True. Es decir, cualquier lista de enteros es una representación válida de algún conjunto de enteros.

a) Escribir en pseudocódigo los algoritmos de las operaciones definidas en el TAD Conjunto($\mathbb{Z}$).

b) Implementar en Python.

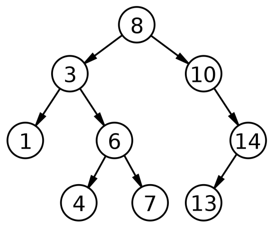
Ejercicio 8. Una estructura de representación alternativa para implementar el TAD Conjunto es usando un *Árbol binario de búsqueda* (ABB), y puede representarse usando los siguientes tipos:

```
Conjunto==⟨raiz:Ref(NodoBinario)⟩
NodoBinario==
      ⟨valor:ℤ, hijoIzquierdo:Ref(NodoBinario), hijoDerecho:Ref(NodoBinario)⟩
```

El invariante de representación de la estructura propuesta es que $Conjunto.raiz$ apunta a la raiz del arbol, no hay ciclos entre los nodos y para todo nodo n_1, n_2 sucede que

- si n_2 es *alcanzable* desde $n_1.hijoIzquierdo$, entonces $n_1.valor > n_2.valor$;
- si n_2 es *alcanzable* desde $n_1.hijoDerecho$, entonces $n_1.valor < n_2.valor$.

Por ejemplo, el siguiente dibujo muestra un ABB válido:



a) Escribir en pseudocódigo los algoritmos de las operaciones definidas en el TAD Conjunto($\mathbb{Z}$) usando la estructura de representación descripta.

b) Suponiendo una distribución uniforme de los enteros que se agregan al conjunto, estimar el orden, en promedio, de las operaciones *Agregar* y *Pertenece*.

c) Escribir en pseudocódigo un algoritmo que imprima en orden ascendente los elementos del conjunto (suponer que cuenta con una función `print(x)`, con $x : \mathbb{Z}$). ¿Qué cambio debería hacer para imprimirlos en order descendente?

d) Implementar en Python el TAD Conjunto($\mathbb{Z}$) usando como estructura un ABB.