Práctica de ejercicios # 11 - Linked Lists

Estructuras de Datos, Universidad Nacional de Quilmes

18 de junio de 2024

Aclaraciones:

- ***Los ejercicios fueron pensados para ser resueltos en el orden en que son presentados.*** *No se saltee ejercicios sin consultar antes a un docente.*
- ***Recuerde que puede aprovechar en todo momento las funciones que ha definido****, tanto las de esta misma práctica como las de prácticas anteriores.*
- ***Pruebe todas sus implementaciones****, al menos en una consola interactiva.*
- *Es sumamente aconsejable* ***resolver los ejercicios utilizando primordialmente los conceptos y metodologías vistos en videos publicados o clases presenciales****, dado que los exámenes de la materia evaluarán principalmente este aspecto. Si se encuentra utilizando formas alternativas al resolver los ejercicios consulte a los docentes.*

1. Linked List

Ejercicio 1

Dada la siguiente representación de listas, llamada *LinkedList*:

```
struct NodoL {
   int elem; // valor del nodo
   NodoL* siguiente; // puntero al siguiente nodo
}

struct LinkedListSt {
   // INV.REP.: cantidad indica la cantidad de nodos que se pueden recorrer
   //           desde primero por siguiente hasta alcanzar a NULL
   int cantidad;   // cantidad de elementos
   NodoL* primero; // puntero al primer nodo
}

typedef LinkedListSt* LinkedList; // INV.REP.: el puntero NO es NULL

struct IteratorSt {
   NodoL* current;
}

typedef IteratorSt* ListIterator; // INV.REP.: el puntero NO es NULL
```

Definir la siguiente interfaz de este tipo de listas, indicando el costo obtenido (intentar que sea lo más eficiente posible):

- `LinkedList nil()`
  Crea una lista vacía.

- `bool isEmpty(LinkedList xs)`
  Indica si la lista está vacía.
- `int head(LinkedList xs)`
  Devuelve el primer elemento.
- `void Cons(int x, LinkedList xs)`
  Agrega un elemento al principio de la lista.
- `void Tail(LinkedList xs)`
  Quita el primer elemento.
- `int length(LinkedList xs)`
  Devuelve la cantidad de elementos.
- `void Snoc(int x, LinkedList xs)`
  Agrega un elemento al final de la lista.
- `ListIterator getIterator(LinkedList xs)`
  Apunta el recorrido al primer elemento.
- `int current(ListIterator ixs)`
  Devuelve el elemento actual en el recorrido.
- `void SetCurrent(int x, ListIterator ixs)`
  Reemplaza el elemento actual por otro elemento.
- `void Next(ListIterator ixs)`
  Pasa al siguiente elemento.
- `bool atEnd(ListIterator ixs)`
  Indica si el recorrido ha terminado.
- `void DisposeIterator(ListIterator ixs)`
  Libera la memoria ocupada por el iterador.
- `void DestroyL(LinkedList xs)`
  Libera la memoria ocupada por la lista.

Ejercicio 2

Definir las siguientes funciones utilizando la interfaz de *LinkedList*, indicando costos:

1. `int sumatoria(LinkedList xs)`
   Devuelve la suma de todos los elementos.
2. `void Sucesores(LinkedList xs)`
   Incrementa en uno todos los elementos.
3. `bool pertenece(int x, LinkedList xs)`
   Indica si el elemento pertenece a la lista.
4. `int apariciones(int x, LinkedList xs)`
   Indica la cantidad de elementos iguales a x.
5. `int minimo(LinkedList xs)`
   Devuelve el elemento más chico de la lista.
6. `LinkedList copy(LinkedList xs)`
   Dada una lista genera otra con los mismos elementos, en el mismo orden.
   Nota: notar que el costo mejoraría si *Snoc* fuese $O(1)$, ¿cómo podría serlo?

7. `void Append(LinkedList xs, LinkedList ys)`
   Agrega todos los elementos de la segunda lista al final de los de la primera.
   La segunda lista se destruye.
   Nota: notar que el costo mejoraría si *Snoc* fuese $O(1)$, ¿cómo podría serlo?

Ejercicio 3

Agregar la operación de *Append* a la interfaz de *LinkedList*, e implementarla como implementador en $O(1)$.

2. Set

Ejercicio 4

Dada la siguiente representación de conjuntos:

```
struct NodoS {
   int elem; // valor del nodo
   NodoS* siguiente; // puntero al siguiente nodo
}

struct SetSt {
   int cantidad; // cantidad de elementos diferentes
   NodoS* primero; // puntero al primer nodo
}

typedef SetSt* Set;
```

Definir la siguiente interfaz de este tipo de conjuntos, indicando el costo obtenido (intentar que sea lo más eficiente posible):

- `Set emptyS()`
  Crea un conjunto vacío.
- `bool isEmptyS(Set s)`
  Indica si el conjunto está vacío.
- `bool belongsS(int x, Set s)`
  Indica si el elemento pertenece al conjunto.
- `void AddS(int x, Set s)`
  Agrega un elemento al conjunto.
- `void RemoveS(int x, Set s)`
  Quita un elemento dado.
- `int sizeS(Set s)`
  Devuelve la cantidad de elementos.
- `LinkedList setToList(Set s)`
  Devuelve una lista con los lementos del conjunto.
- `void DestroyS(Set s)`
  Libera la memoria ocupada por el conjunto.

3. Queue

Ejercicio 5

Dada la siguiente representación de colas:

```
struct NodoQ {
   int elem; // valor del nodo
   NodoQ* siguiente; // puntero al siguiente nodo
}

struct QueueSt {
   int cantidad; // cantidad de elementos
   NodoQ* primero; // puntero al primer nodo
   NodoQ* ultimo; // puntero al ultimo nodo
}

typedef QueueSt* Queue;
```

Definir la siguiente interfaz de este tipo de colas, respetando el costo de las operaciones:

- `Queue emptyQ()`
  Crea una cola vacía.
  Costo: $O(1)$.
- `bool isEmptyQ(Queue q)`
  Indica si la cola está vacía.
  Costo: $O(1)$.
- `int firstQ(Queue q)`
  Devuelve el primer elemento.
  Costo: $O(1)$.
- `void Enqueue(int x, Queue q)`
  Agrega un elemento al final de la cola.
  Costo: $O(1)$.
- `void Dequeue(Queue q)`
  Quita el primer elemento de la cola.
  Costo: $O(1)$.
- `int lengthQ(Queue q)`
  Devuelve la cantidad de elementos de la cola.
  Costo: $O(1)$.
- `void MergeQ(Queue q1, Queue q2)`
  Anexa $q2$ al final de $q1$, liberando la memoria inservible de $q2$ en el proceso.
  Nota: Si bien se libera memoria de $q2$, no necesariamente la de sus nodos.
  Costo: $O(1)$.
- `void DestroyQ(Queue q)`
  Libera la memoria ocupada por la cola.
  Costo: $O(n)$.

4. Árboles binarios

Ejercicio 6

Dada esta definición para árboles binarios

```
struct NodeT {
   int elem;
   NodeT* left;
   NodeT* right;
}

typedef NodeT* Tree;
```

definir la siguiente interfaz:

- `Tree emptyT()`
- `Tree nodeT(int elem, Tree left, Tree right)`
- `bool isEmptyT(Tree t)`
- `int rootT(Tree t)`
- `Tree left(Tree t)`
- `Tree right(Tree t)`

Ejercicio 7

Definir las siguientes funciones utilizando la interfaz de árbol y recursión:

1. `int sumarT(Tree t)`
   Dado un árbol binario de enteros devuelve la suma entre sus elementos.
2. `int sizeT(Tree t)`
   Dado un árbol binario devuelve su cantidad de elementos, es decir, el tamaño del árbol (size en inglés).
3. `bool perteneceT(int e, Tree t)`
   Dados un elemento y un árbol binario devuelve True si existe un elemento igual a ese en el árbol.
4. `int aparicionesT(int e, Tree t)`
   Dados un elemento e y un árbol binario devuelve la cantidad de elementos del árbol que son iguales a e.
5. `int heightT(Tree t)`
   Dado un árbol devuelve su altura.
6. `ArrayList toList(Tree t)`
   Dado un árbol devuelve una lista con todos sus elementos.
7. `ArrayList leaves(Tree t)`
   Dado un árbol devuelve los elementos que se encuentran en sus hojas.
8. `ArrayList levelN(int n, Tree t)`
   Dados un número n y un árbol devuelve una lista con los nodos de nivel n.

Ejercicio 8

Definir las funciones del punto anterior utilizando BFS (recorrido iterativo a lo ancho), a excepción de `heightT`, `leaves` y `levelN`. Para esto, utilizar una `Queue` de `Tree`.

5. Heaps

Ejercicio 9

Implementar heaps binarias según el código de la teórica, y probarlas con ejemplos.