Práctica de ejercicios # 10 - Linked Lists

Estructuras de Datos, Universidad Nacional de Quilmes

18 de noviembre de 2020

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 {
   int cantidad; // cantidad de elementos
   NodoL* primero; // puntero al primer nodo
   NodoL* actual; // puntero al nodo actual (para recorridos)
}

typedef LinkedListSt* LinkedList;
```

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.
- `void initialize(LinkedList xs)`
  Apunta el recorrido al primer elemento.
- `int current(LinkedList xs)`
  Devuelve el elemento actual en el recorrido.
- `void setCurrent(int x, LinkedList xs)`
  Reemplaza el elemento actual por otro elemento.
- `void next(LinkedList xs)`
  Pasa al siguiente elemento.
- `void finished(LinkedList xs)`
  Indica si el recorrido ha terminado.
- `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.
   Nota: notar que el costo mejoraría si *snoc* fuese $O(1)$, ¿cómo podría serlo?

2. Set

Ejercicio 3

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.
- `void 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 4

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 lista vacía.
  Costo: $O(1)$.
- `bool isEmptyQ(Queue q)`
  Indica si la lista 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 lista.
  Costo: $O(n)$.