Práctica 3

Pilas

1. Implemente pilas utilizando arreglos. Utilice la siguiente estructura:

```
typedef struct _AStack {
  int data[MAX_STACK];
  int back;
} AStack;
```

Procure que la interfaz provea las siguientes operaciones:

a) `AStack *astack_create()` : crea una pila vacía.

b) `int astack_top(AStack *)`: toma una pila y devuelve el elemento en la cima.

c) `void astack_push(AStack *, int)`: toma una pila y un elemento y agrega el elemento a la pila.

d) `void *astack_pop(AStack *)`: toma una pila y quita el elemento de la cima.

e) `void *astack_reverse(AStack *)`: toma una pila y la invierte.

f) `void astack_print(AStack *)`: toma una pila e imprime sus elementos en orden.

g) `void astack_destroy(AStack *)`: toma una pila y la destruye.

2. Modifique la estructura recién utilizada para poder almacenar cualquier cantidad de elementos (modifique las funciones necesarias para que, en caso de quedarse sin lugar, se solicite más memoria automaticamente).

3. Implemente pilas enlazadas. Utilice la siguiente estructura nodo para guardar cada uno de los datos de su pila:

```
typedef struct _SLStackNode {
  int data;
  struct _SLStackNode *next;
} SLStackNode;

typedef SLStackNode *SLStack;
```

Procure que la interfaz provea las mismas operaciones que en el ejercicio anterior.

4. Considere las listas simplemente enlazadas implementadas en la práctica 1. Implemente la función `SList * slist_reverse(SList *)` que tome una lista simplemente enlazada y la invierta (Ayuda: puede utilizar una pila).

5. En la notación matemática usual, una operación aritmética se escribe de manera infija, por ej. el $+$ en $3+4$ va entre los operadores 3 y 4. En la notación polaca inversa, primero se escriben los operadores y luego se escribe la operación, por lo que la expresión anterior quedaría `3 4 +`. Esta notación evita la necesidad de paréntesis y es utilizada por algunas calculadoras.
Escriba una calculadora que tome una expresión aritmética en notación polaca inversa y calcule su valor[1]. Por ej., la expresión matemática $5+((1+2)*4)-3$ en notación polaca inversa se escribe `5 1 2 + 4 * + 3 -` y debe dar como resultado el valor 14.

[1] El artículo "Notación polaca inversa" de Wikipedia puede ser de ayuda

Colas

6. Implemente colas utilizando arreglos circulares. Utilice la siguiente estructura:

```
typedef struct _AQueue {
  int data[MAX_QUEUE];
  int front, back;
} AQueue;
```

Procure que la interfaz provea las siguientes operaciones:

a) `AQueue *queue_create()`: crea una cola.

b) `int aqueue_front(AQueue *)`: toma una cola y devuelve el elemento en la primera posición.

c) `void aqueue_enqueue(AQueue *, int)`: toma una cola y un elemento y agrega el elemento al fin de la cola.

d) `void aqueue_dequeue(AQueue *)`: toma una cola y le quita su primer elemento.

e) `void aqueue_print(AQueue *)`: toma una cola y la imprime en orden.

f) `void aqueue_destroy(AQueue *)`: toma una cola y la destruye.

7. Implemente colas enlazadas. Utilice la siguiente estructura nodo para guardar cada uno de los datos de su cola:

```
typedef struct _SLQueueNode {
  int data;
  struct _SLQueueNode *next;
} SLQueueNode;

typedef SLQueueNode *SLQueue;
```

Procure que la interfaz provea las mismas operaciones que en el ejercicio anterior.

8. Considere los árboles binarios implementados en la práctica anterior. Implemente la función `btree_foreach_level(BTree *list, VisitorFuncInt visit, void *extra_data)` que utilice el recorrido 'Primero por Extensión' (Ayuda: puede utilizar una cola para guardar los nodos a visitar).

Heaps Binarios

9. Implemente heaps binarios utilizando arreglos para representar árboles binarios completos parcialmente ordenados. Utilice la siguiente estructura:

```
typedef struct _BHeap {
  int data[MAX_HEAP];
  int nelems;
} BHeap;
```

Procure que la interfaz provea las siguientes operaciones:

a) `BHeap *bheap_create()`: crea un heap.

b) `int bheap_minimum(BHeap *)`: toma un heap y devuelve el menor elemento.

c) `void bheap_erase_minimum(BHeap *)`: toma un heap y borra su menor elemento.

d) `void bheap_insert(BHeap * , int)`: toma un heap y agrega un elemento.

e) `void bheap_print(BHeap *)`: toma un heap e imprime sus elementos utilizando el orden "Primero por Extensión".

f) `void bheap_destroy(BHeap *)`: toma un heap y lo destruye.

10. Una *cola de prioridades* es una estructura de datos en la que los elementos se atienden en el orden indicado por una prioridad asociada a cada uno. Si varios elementos tienen la misma prioridad, se atenderán de modo convencional según la posición que ocupen.

Ejemplo: Supongamos contamos con una cola vacía $q = \langle\rangle$ e insertamos 5 elementos en el siguiente orden:

- $Elemento_1$ con $priority = 10$. En este caso la cola de prioridad resultante será: $q = \langle Elemento_1 \rangle$;
- $Elemento_2$ con $priority = 15$, la cola de prioridad resultante será: $q = \langle Elemento_2, Elemento_1 \rangle$;
- $Elemento_3$ con $priority = 10$, la cola de prioridad resultante será: $q = \langle Elemento_2, Elemento_1, Elemento_3 \rangle$;
- $Elemento_4$ con $priority = 5$, la cola de prioridad resultante será: $q = \langle Elemento_2, Elemento_1, Elemento_3, Elemento_4 \rangle$;
- $Elemento_5$ con $priority = 40$, la cola de prioridad resultante será: $q = \langle Elemento_5, Elemento_2, Elemento_1, Elemento_3, Elemento_4 \rangle$.

Utilice un heap para implementar una cola de prioridad. Resuelva los siguientes ítems:

a) Implemente la función `void pqueue_enqueue(PQueue *q, int data, int priority)` que toma una cola y un valor data con su prioridad asociada y agrega el elemento a la cola teniendo en cuenta el valor de su prioridad (ver ejemplo).

b) Implemente la función `pqueue_dequeue` que extraiga un elemento que tenga la mayor prioridad disponible en la cola.

c) Implemente la función `pqueue_front` que obtenga un elemento que tenga la mayor prioridad disponible.

d) ¿Podría implementar esta estructura usando un arreglo o una lista enlazada como una cola normal? ¿Qué ventaja puede tener utilizar heaps?