Práctica 3

Pilas

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

```
struct _Stack {
  int data[MAX_STACK];
  int back;
};

typedef struct _Stack Stack;
```

Procure que la interfaz provea las siguientes operaciones:

a) `Stack *stack_create()` : crea una pila.

b) `int stack_top(Stack)`: toma una pila y devuelve el elemento en la cima.

c) `Stack *stack_push(Stack *, int)`: toma una pila y un elemento y agrega el elemento a la pila.

d) `Stack *stack_pop(Stack *)`: toma una pila y devuelve la pila sin el elemento de la cima.

e) `Stack *stack_reverse(Stack *)`: toma una pila y devuelve la pila invertida.

f) `void stack_print(Stack)`: toma una pila y la imprime en orden.

g) `void stack_destroy(Stack *)`: 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 _node {
  int data;
  struct _node *next;
}Node;

typedef Node sNode;
```

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

a) `pNodo *stack_create();`

b) `int stack_top(sNode *);`

c) `sNode *stack_push(sNode *, int);`

d) `sNode *stack_pop(sNode *);`

e) `sNode *stack_reverse(sNode *);`

f) `void stack_print(sNode *);`

g) `void stack_destroy(sNode *)`.

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).

Colas

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

```
struct _Queue {
  int data[MAX_QUEUE];
  int front, back;
};

typedef struct _Queue Queue;
```

Procure que la interfaz provea las siguientes operaciones:

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

b) `int queue_front(Queue)`: toma una cola y devuelve el elemento en la primera posición.

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

d) `Queue *queue_dequeue(Queue *)`: toma una cola y devuelve la cola sin su primer elemento.

e) `void queue_print(Queue)`: toma una cola y la imprime en orden.

f) `void queue_destroy(Queue *)`: toma una cola y la destruye.

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

```
typedef struct _node {
  int data;
  struct _node *next;
}Node;

typedef Node qNode;
```

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

a) `qNode *queue_create();`

b) `int queue_front(qNode *);`

c) `qNode *queue_enqueue(qNode *, int);`

d) `qNode *queue_dequeue(qNode *);`

e) `void queue_print(qNode *);`

f) `void queue_destroy(qNode *)`.

7. 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

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

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

typedef struct _BHeap 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) `BHeap *bheap_erase_minimum(BHeap *)`: toma un heap y borra su menor elemento.

d) `BHeap *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.