Córdoba 5 de Diciembre del 2024

Examen Final de Laboratorio

Algoritmos y Estructura de Datos II

TAD Lista Pivot

Implementar el TAD **plist** que representa una lista de valores numéricos. El TAD almacena números, representados a través del tipo `t_elem`. La lista pivot contiene un elemento especial (que es único), llamado pivot, y que determina el orden en que los demás elementos se agregan o eliminan de la lista. Se debe representar la lista utilizando nodos (simplemente) enlazados, pero la estructura principal se deja a criterio del alumno.

Aclaración: la lista pivot **no está necesariamente** ordenada.

Propiedades principales:

- **plist** se crea seleccionando el valor del elemento pívot. La lista no puede existir sin pivot, y este elemento no puede ser removido.
- **plist** puede tener elementos repetidos, mientras no sean el pivot.
- Cada elemento subsiguiente se agrega junto al elemento pivot, de la siguiente manera:
    - Si el elemento a agregar es menor que el pívot, se agrega inmediatamente a la izquierda del pivot
    - Si el elemento a agregar es mayor que el pívot, se agrega inmediatamente a la derecha de pivot.
    - No pueden agregarse elementos iguales al pivot

Las operaciones del TAD se listan a continuación:

Función	Descripción
`plist plist_create(t_elem pivot)`	Crea una lista “vacía” con un solo elemento: `pivot`
`plist plist_add(plist l, t_elem e)`	Agrega el elemento **e** a la lista.
`bool plist_is_empty(plist l)`	Indica si la lista está vacía, una lista que solo contiene al pivot se considera vacía!
`unsigned int plist_length(plist l)`	Devuelve la cantidad de elementos que hay en la lista (debe contar el pivot)
`t_elem plist_get_pivot(plist l)`	Retorna el valor del pivot
`plist plist_delete_prev(plist l)`	Elimina el elemento inmediato anterior al pivot
`plist plist_delete_next(plist l)`	Elimina el elemento inmediato siguiente al pivot
`unsigned int plist_count(plist l, t_elem e)`	Retorna la cantidad de veces que aparece **e** en la lista
`t_elem * plist_to_array(plist l);`	Devuelve un arreglo en memoria dinámica con el contenido de la lista, en el orden correcto

`void plist_dump(plist l);`	Muestra el contenido de la lista por pantalla
`plist plist_destroy(plist l)`	Destruye la lista y libera toda la memoria usada

Ejemplo de funcionalidad de las operaciones:

```
> plist_create(8)
[ 8 ]
> plist_add(l, 5)
[ 5, 8 ]
> plist_add(l, 7)
[ 5, 7, 8 ]
> plist_add(l, 12)
[ 5, 7, 8, 12]
> plist_delete_prev()
[ 5, 8, 12 ]
> plist_length()
3
> plist_delete_prev()
[ 8, 12 ]
> plist_delete_prev()
[ 8, 12 ]
> plist_add(l, 3)
[ 3, 8, 12 ]
> plist_add(l, 100)
[ 3, 8, 100, 12 ]
> plist_add(l, 3)
[ 3, 3, 8, 100, 12 ]
```

Se debe completar la estructura principal para lograr que la función `plist_length()` sea de orden constante *O(1)*. **El programa resultante no debe dejar *memory leaks ni lecturas/escrituras inválidas*.**

Tomen el tiempo necesario para diseñar una estructura principal correcta, que les permita simplificar el código. Es posible hacerlo de forma que casi todas las funciones de plist sean de orden constante.

Una vez compilado el programa puede probarse usando los archivos de la carpeta **`input`**. El nombre de cada archivo de esta carpeta indica cuántos valores enteros están listados dentro de él. Puesto que pueden haber valores repetidos. Un ejemplo de ejecución:

```
$ ./plistload input/example-easy-005.in
List of elements read in the file: [ 1, 2, 3, 6, 4, 5 ]

The file 'input/example-easy-005.in' has 6 elements
The pivot element is 3
1 [OK], 2 [OK], 3 [OK], 6 [OK], 4 [OK], 5 [OK]

 Removings elements test
List after removing 2 previous element: [ 3, 6, 4, 5 ]
List after removing 2 next element: [ 3, 5 ]
```

Consideraciones:

- **Solo** se debe modificar el archivo **`plist.c`**
- Los elementos del tipo **`t_elem`** deben manipularse utilizando sus funciones propias, dadas en **`elem.h`**
- Una invariante de representación completa debería verificar la propiedad fundamental de la representación.
- Lo mínimo que se pide de la invariante es que verifique consistencia entre los campos de la estructura principal.
- La implementación debe verificar las pre y post condiciones especificadas en **`plist.h`**.
- Se provee el archivo **`Makefile`** para facilitar la compilación.
- Se recomienda usar las herramientas **`valgrind`** y **`gdb`**.
- Usando **`make test`** se ejecuta el programa cargando todos los archivos de input.
- Usando **`make valgrind`** es equivalente a **`make test`** con la diferencia que se ejecuta el programa usando **`valgrind`**.
- Para que se sigan probando los ejemplos aunque falle alguno se puede usar:
  `$ make test -k`
- **Si el programa no compila, no se aprueba el examen.**
- Los *memory leaks* bajan puntos
- Entregar código muy improlijo puede restar puntos
- Si **`plist_length()`** no es de orden constante *O(1)* baja muchísimos puntos
- **No se deben modificar los .h**

Ejercicio para libres:

- Crear un nuevo método `static` (sin modificar plist.h):
  - `plist_remove_duplicates(plist l, t_elem e)`: Quita todos las duplicaciones del elemento e de la lista (solo queda el primer elemento encontrado)
  - Tiene como precondicion: `(e != pivot)`