Estrucutras Abstractas en Mips ensamblador.

Daniel Pinto 15-11138
Pedro Rodriguez 15-11264

February 2020

1 TAD Lista

Nuesta implementacion de lista es una estructura de tipo registro la cual definimos como:

```
struct Lista {
    list_header:: *ListHeader
    list_contents:: *ListContents
}
```

En donde: *ListHeader y *ListContents son apuntadores a estructuras de tipo ListHeader y ListContents definidas como:

```
struct ListHeader {
    list_header_add:: address (word)
    list_compare_pointer:: address (word)
    list_print_pointer:: address (word)
    pointer2First :: address (word)
}


struct ListContents {
    list_contents_add:: address (word)
    list_contents_value:: address (word)
    pointer2Prev:: address (word)
    pointer2Next :: address (word)
}
```

Consideramos dividir la lista en estos dos tipos para emular un objeto y tener un metadata al que podamos no solo referirnos de manera sencilla, sino que ademas sea escalable para futuras aplicaciones, como lo es: construir un hashcode tomando la direccion del campo (la cual forma una correspondencia biunivoca con cada instancia del objeto), o incluso verificacion de tipo (sumando alguna

constante a la direccion del objeto.) Sacrificando solo un minimo de espacio que es el que se lleva la direccion de la estrucutra.

1.1 list_crear

Para crear la lista vacia solo necesitamos crear el objeto header y definir el campo pointer2Next a un valor nulo definido como: 0xFFFFFFFF.

1.2 list_insertar

La insercion se basa en una busqueda lineal sobre los elementos del campo list_contents hasta encontrar uno que sea mayor.

1.3 list_obtener

La busqueda se basa en iterar sobre la lista hasta llegar al argumento n, y retornar el campo list_contents_value .

1.4 list_imprimir

La impresion se basa en iterar sobre la lista y aplicar la funcion imprimir al campo list_contents_value .

1.5 list_destruir

Para destruir la lista basta con desvincular y poner en 0 cada elemento de la lista.

2 TAD_TablaHash

Nuesta implementacion de tabla de hash es una estructura de tipo registro la cual definimos como:

```
struct TAD_TablaHash {
    hashTable_add:: *Hashtable
    hashTable_header:: *HashHeader
    hashTable_contents:: *HashContents
}
```

En el cual, hashTable_add es un apuntador al propio objeto, y hashTable_header, hashTable_contents son apuntadores a las estructuras:

```
struct HashHeader {
    Hash_head:: address (word)
    HHPoint2Comp:: address (word)
```

```
    HHPoint2Hash:: address (word)
    HHNumber_Classes:: address (word)
    HHPoint2Print:: address (word)
}

struct HashContent {
    hash_content_table :: Array[Lists[Pair]]
}
```

En donde el HashHeader es el metadata de la tabla de hash, y el HashContent es el contenido per se. Replicando la misma metodologia que el Tad Lista.

2.1 tab_crear

De forma analoga al list_crear almacenamos el metadata en el campo HashHeader, teniendo en cuenta la salvedad que la funcion de comparacion va a comparar los pares mediante la clave (visto en la seccion miscelanea) y luego reservamos el espacio equivalente a 4 veces el numero de particiones (suficiente como para almacenar un apuntador a la lista que maneja las colisiones correspondiente).

2.2 tab_insertar

Para insertar creamos un tipo Par: elementoxclave, y luego insertamos en la lista ubicada en el indice dado por la funcion de hash aplicada a la clave para manejar las colisiones.

2.3 tab_buscar

De forma similar al tab_insertar, hashseamos la clave, y creamos un par dummy: Par: 0xclave, y realizamos una busqueda lineal sobre la clave, hasta encontrar el otro par, y devolver el elemento, o devolver -1 en caso de que la clave no se encuentre en la lista.

2.4 tab_rehash

Para realizar rehashing sobre la tabla de hash, basto con crear una nueva tabla con el nuevo numero de particiones, e iterar sobre cada elemento de la tabla dehash antigua, para luego insertarlos con la funcion tab_insertar.

2.5 tab_destruir

El tab_destruir aplica list_destruir a cada una de las posiciones del arreglo, para luego finalmente setear en 0 tales posiciones.

3 Pair

Pair es una estructura tipo registro definida como:

```
struct Pairr {
    Pair_fst:: address (word)
    Pair_snd:: address (word)
}
```

La cual posee solo 3 metodos: su constructor y dos getters para ottener cada componente del par. Usamos esta estructura auxiliar para modelar los tpos: elementoxclave.

4 Miscelanea

Aqui señalamos una salvedad que posee el codigo para funcionar con cierto nivel de abstraccion, la cual se se encuentra en la funcion HC_compare la cual recibe dos elementos de tipo: elementoxclave, saca las 2das componentes de cada par, y las compara mediante una funcion guardada en una etiqueta globar llamada Point2Comp, simulando asi una especie de currying para poder guardar la nueva funcion de comparacion en las listas creadas,