UNR - FCEIA

LCC - Estructuras de Datos y Algoritmia I

Trabajo Práctico N^o 3 - Tabla Hash y operaciones de intervalos

Espíndola Ignacio

Agosto 2020

Profesores:
Federico Severino Guimpel
Mauro Lucci
Martín Ceresa
Emilio Lopez
Valentina Bini

1 Intervalo

El primer avance en el trabajo fue pensar la estructura con la que representar a los elementos de los conjuntos, ya que estos tenían la característica de que podían ser o bien números o intervalos. Para agrupar estas dos posibilidades, lo que hice fue crear una estructura en la que los números se guardasen como intervalos donde inicio=final:

Estructura intervalo

```
typedef struct {
  int inicio;
  int final;
} Intervalo;
```

Luego, cuando imprimo estos valores, simplemente chequeo si inicio == final, y los imprimo como corresponda.
Las funciones que manejan los intervalos son bastante simples y guardan gran similitud con las utilizadas para el TP2 de la materia.
Lo que vale la pena destacar es la función intervalo_interseccion.

intervalo_interseccion

```
    Intervalo *intervalo_interseccion(Intervalo *intervaloA, Intervalo
        *intervaloB){
  int inicio, final;
  if(intervaloA->inicio < intervaloB->inicio){
    inicio = intervaloB->inicio;
  } else {
    inicio = intervaloA->inicio;
  }
  if(intervaloA->final < intervaloB->final){
    final = intervaloA->final;
  } else {
    final = intervaloB->final;
  }
  Intervalo *interseccion = intervalo_crear(inicio, final);
  return interseccion;
}
```

Esta función es llamada solamente si ya se sabe (mediante la función intervalo_interseca) que los dos intervalos tienen intersección, entonces no hace falta chequear ese caso.
Básicamente, devuelve un nuevo intervalo, donde el inicio es el mayor de los dos inicios de los intervalos, y el final es el mínimo de los dos finales de los intervalos.
entro del árbol.

2 Conjunto

2.1 Estructura

Una vez terminados los intervalos, continué con los conjuntos, y consideré varias alternativas para la estructura a utilizar.
Llegue a la conclusión de que trabajar con una Linked List (doblemente enlazada), no suponía ningún mayor problema comparada con otras estructuras como Arboles.

Estructura conjunto

```
   typedef struct _GNodo {
  Intervalo *dato;
  struct _GNodo *sig;
  struct _GNodo *ant;
} GNodo;

typedef struct {
  GNodo *primero;
  GNodo *ultimo;
  int cantidad;
} Extremos;

typedef Extremos *Conjunto;
```

2.2 Funciones

Luego, la mayoría de las funciones son iguales o muy similares a las del trabajo practico anterior. Por lo tanto voy a explicar las funciones nuevas

2.2.1 Colapsar un conjunto

Esta función existe para chequear que los conjuntos no ocupen mas espacio del necesario, y acomodarlos en caso de que no fuese así
. Por ejemplo, si un conjunto es [1:5, 4:10, 11:20], puede ser reescrito como un conjunto de un solo intervalo [1:20].

2.2.2 Unión

Para la unión, lo que hago es agregar todos los intervalos de ambos conjuntos, y luego los ordeno utilizando el algoritmo mergeSort.
Finalmente, colapso el conjunto para optimizar el espacio utilizado.

2.2.3 Intersección

Para la intersección, el algoritmo es este:
Primero, chequeo si los primeros intervalo de cada conjunto se intersecan entre si.
De ser así, agrego esa intersección al conjunto que devolveré Luego, avanzo al siguiente intervalo en el conjunto donde encontré al menor de los dos intervalos recién comparados.
Cabe destacar que con menor me refiero al que tiene un menor extremo final.
Con este método, me aseguro de que voy a encontrar todas las intersecciones entre los intervalos de los conjuntos.

2.2.4 Resta

Para la resta, el procedimiento es el siguiente.
Primero que nada, clono el conjunto del cual quiero restar.
Luego, recorro cada intervalo del segundo conjunto y lo voy restando de cada elemento del primer conjunto.
Para esto, tuve que subdividir en 6 casos (utilizare letras para explicar, considerar que a<b<c<d):

[a:b] - [b:c], entonces [a:b-1]

[b:c] - [a:b], entonces [a+1:b]

[b:c] - [a:d], entonces NULL

[a:c] - [b:d], entonces [a:b-1]

[b:d] - [a:c], entonces [b+1:d]

[a:d] - [b:c], entonces [a:b-1] y [c+1:d]

2.2.5 Complemento

El complemento simplemente es la resta entre el universo[INT_MIN:INT_MAX] y el conjunto dado.
Para poder utilizar este método tuve que incluir la librería limits.h.

3 Tree

Lo siguiente fue implementar los arboles, que servirán para manejar las colisiones en la tabla. Es decir, cuando dos conjuntos tengan la misma key de la Tabla Hash, serán almacenados en un AVL de forma de que se pueda seguir accediendo a ellos
La estructura que utilizo guarda semejanzas con la del trabajo practico anterior, con la diferencia de que ya no debo guardar el mayor extremo derecho de todos los hijos del nodo, y ahora almaceno el alias del conjunto.

Estructura Tree

```
typedef struct _Nodo {
 char *alias;
 Conjunto conj;
 int altura;
 struct _Nodo *izq;
 struct _Nodo *der;
} Nodo;

typedef Nodo *Tree;
```

4 Hash

Para la Tabla Hash, la estructura fue simplemente un arreglo de Tree, de largo MAXSIZE.
La función de Hash la diseñe para que fuese equitativa de forma que las colisiones se minimizasen.

Estructura Tree

```
int hash_key(char* alias){
   long long int key = 1, index = 0;
   while(alias[index] != '\0'){
       key *= alias[index];
       key = key % INT_MAX;
       index++;
   }
   key *= (MAXSIZE - strlen(alias));
   key = key % INT_MAX;
   key = key % MAXSIZE;
   return key;
}
```

Lo que hace es multiplicar todos los valores ASCII de los caracteres del alias, mientras simultáneamente realiza el modulo contra INT_MAX, de forma de no tener un overflow.
Luego, para aportar mas variabilidad, también utilizo el largo del alias.
Notese que el termino es (MAXSIZE - strlen(alias)), para asegurarme de que el numero sea suficientemente grande como para llegar a las keys mas altas.
Finalmente, realizo el modulo con el tamaño de la tabla, para obtener la key final.

5 Interprete

Los alias deben tener al menos dos caracteres, y no pueden empezar con un '{'.

Bibliografía

Wikipedia, Función Hash
Wikipedia, Tabla Hash