Estructuras de datos compuestas: Quash (*min heap* y *hash table*.

Alvarado Contreras Rebeca.
Universidad Autónoma de San Luis Potosí, Facultad de Ciencias.
Ingeniería Física, Cómputo de alto rendimiento.
Proyecto 1: Quash.

***Objetivos*—Implementar una estructura de datos compuesta: *Quash* (*min-heap+hash table*) con las intrucciones: `lookup(x)`, `insert(x)`, `deleteMin`, `delete(x)`, asumiendo que x son enteros con signo. Para el *min-heap*, emplear un arreglo.**

***Palabras clave*—Tablas Hash, Min Heap, Quash.**

I. Introducción

Aunque las tablas hash son estructuras muy útiles y eficientes, estas no permiten búsquedas que dependan del orden de los elementos, por ejemplo, el elemento mínimo. Para esta tarea, las estructuras heap son apropiadas, pero a su vez, estas no permiten una búsqueda general de cualquier elemento, solo el mínimo/máximo (según sea un min heap o max heap). Por esto, una estructura combinada con una tabla hash y un heap es conveniente, esto es, un *Quash* (priority queue + hash table).

De esta manera podemos tener las siguientes funciones:

- `insert(x)`: $O(\log n)$
- `find(x)`: $O(1)$ - dominado por Tablas Hash
- `deleteMin`: $O(\log n)$ - dominado por min heap
- `delete(x)`: $O(\log n)$

Ya se han visto las tablas hash, se descrbirá brevement la estructura heap.

I-A. Heap

Un heap (binario) es una estructura de datos usados para la implementación de colas de prioridad (priority queues). Es un árbol binario con dos propiedades especiales: [1]

- Propiedad de forma: El árbol se llena de izquierda a derecha, de tal manera que todos los niveles estén llenos; exceptuando el último nivel.
- Propiedad de heap: Si es un *max-heap*, el valor de cada nodo es mayor o igual que el de sus hijos; y vice versa. Si se trata de un *min-heap*, el valor de cada nodo es menor o igual que le de sus hijos.

Los heaps binarios se suelen implementar con un arreglo, usando posiciones relativas para representar las relaciones padre-hijo. [1]

Para las cuestiones de insertar y eliminar, es necesario usar un algoritmo de *heapify-up* y *heapify-down* respectivamente, para conservar las propiedades del heap; de esto se hablará un poco más a detalle en la siguiente sección.

II. Metodología

En este aparatado se habla sobre consideraciones generales, y una breve descripción del programa implementado.

Para la tabla hash se empleó *open addressing* con sondeo cuadrático; esto, debido a su simplicidad en cuanto a estructura con respecto a *separate chaining*, y su rapidez con respecto otros tipos de sondeo.

Uno de las primeras cuestiones a solucionar es el cómo los número enteros (negativos incluidos) se pasan por la función hash, dado que el módulo de número negativos es "complicado", por esto, se optó por convertir los números a cadenas de caracteres y generar una llave.

Para guardar los elementos a insertar se creó la estructura `element` en donde se guarda, su valor, número de copias, posición en el heap y en la tabla hash. Esta estructura es compartida tanto por el heap, como por la tabla hash, es decir, ambas apuntan a la misma estructura. En específico, al insertar en el heap, este devuelve un puntero al elemento insertado, el cual será la entrada para la tabla hash, en donde el hash item tiene una propiedad de puntero tipo `element`, aunque era suficiente con solo guardar en la hash table la posición en el heap y el número de copias o contador.

Esto se hizo con el motivo de que los cambios que una estructura hiciera, se vieran reflejados en la otra. Aunque esto es muy útil al momento de insertar, se tuvieron unos cuantos problemas dado que el heap es dinámico al mover las posiciones en `heapify down` y `heapify up`, por lo que había que estar actualizando las direcciones en la tabla hash en cada iteración.

Ahora, se habla de manera breve las funciones principales, enfocándose al heap.

II-A. Creación de min heap

El heap se implementó en forma de arreglo, donde se puede saber las posiciones del padre, hijo derecho e izquierdo por medio de las siguientes expresiones, dada la posición i:

$$parent(i) = child >> 1$$

$$left\ child(i) = parent << 1$$

$$right\ child(i) = (parent << 1) + 1$$

El tamaño del arreglo, el usuario dando un aproximado del número de nodos, se calcula para obtener todo los niveles del árbol completo. Recordando que el número de nodos, dado el número de niveles n es: $2^n - 1$, los niveles requqeridos para satisfacer el número de nodos pedidos por el usuario es $\lceil \log_2 nodos \rceil$. También hay que recordar reservar un espacio extra para el centinela inicial (en posición 0 del arreglo).

Los elementos fueron inicializados con un valor de INF, exceptuando por el centinela, con calor -INF.

II-B. Encontrar elemento

Esta función fue la más sencilla, con la tabla hash se busca el elemento y se le devuelve al usuario el número de copias si fue encontrado. Cabe recordar que la función de búsqueda de la tabla hash devuelve un puntero hacia el elemento encontrado.

En esta parte, también se le avisa al usuario en caso de encontrar alguna llave duplicada.

II-C. Insertar elemento

Se empieza por buscar la existencia del elemento, si es encontrado, se aumenta el contador o el número de copias. Ahora, si no fue hallado, primero se inserta en el heap y después con el puntero regresado por el heap se inserta en la tabla hash.

II-C1. Heap: Insertar

El valor a insertar, se inserta en la primera posición disponible (en la posición correspondiente al número de elementos + 1), y después se le hace un `heapify up` (véase Código 1).

Este se trata de que el elemento insertado al final, quede en el lugar correcto para no violar la propiedad heap. Se compara este con su padre, y si este elemento insertado es menor, se intercambia (el contenido de toda la estructura), hasta que su padre sea menor que el elemento.

```
int heapify_up(quash *qh, int pos){
    heap *h=qh->hp;
    int new_pos=pos;
    int ht_pos, ht_parent;
    while(h->elem[pos].value <
    ↪  h->elem[parent(pos)].value){
        //el hijo es más chico
        new_pos=parent(pos);
        element elem_aux=(h->elem[pos]);
        ↪  //intercambiar
        (h->elem[pos])=(h->elem[parent(pos)]);
        (h->elem[parent(pos)])=(elem_aux);//elem
        ↪  auz ya es una direccion

        //actualizar posiciones, recordar que los
        ↪  elementos ya estan actualizados
        h->elem[pos].pos_heap=pos;
        h->elem[parent(pos)].pos_heap=new_pos;


        //aactualizar las direcciones en la ht
        ht_pos=h->elem[pos].pos_hash;
        ht_par=parent(pos)].pos_hash;

        qh->ht->item_list[ht_pos].elem = ...
                        &(h->elem[pos]);

        qh->ht->item_list[h->elem[ht_par].elem=...
                        &(h->elem[parent(pos)]);

        pos=parent(pos);
    }
    return new_pos;
}
```

Código 1: Código para algoritmo `heapify up`.

II-D. Eliminar elemento

El porceso para eliminar el elemento mínimo (raiz) y otro arbitrario es prácticamente igual, exceptuando por la hoja, en la cual solo se reinicia la celda/elemento.

Se comienza por checar que la estructura no esté vacía y la existencia del elemento con la tabla hash, y se obtiene la posición en el heap, y se revisa el número de copias, si las copias son mayores a 1, solo se disminuye el contador.

Por otro lado, si solo hay una copia, se elimina de la siguiente manera: La hoja del heap, se cambia a esta posición (reiniciando la hoja y decrementando el número de elemento), y a esta posición se le hace un `heapify down` (véase Código 2); en donde el elemento es comparado con sus hijos, y se intercambia con el menor de ellos. La eliminación en la tabla hash es la usual.

```
void heapify_down(quash *qh, int pos){
    heap *h = qh->hp;
    int last_leave = h->num_elem;
    int child, ht_pos, ht_child;

    for( ; pos < last_leave; pos=child){
        child = left_child(pos);

        if(child > last_leave)//todo bien, aun no
        ↪  llegamos al final
```

```
            break;
            //que pasaa si era >=?

        //queremos intercambiarlo por el menor
        if( h->elem[child].value >
        ↪  h->elem[child+1].value)
           child++;//el right child es el menor

        //el child es menor que el padre?
        if(h->elem[child].value <
        ↪  h->elem[pos].value){

            element elem_aux = h->elem[pos];
            h->elem[pos] = h->elem[child];
            h->elem[child] = elem_aux;

            h->elem[pos].pos_heap=pos;
            h->elem[child].pos_heap=child;

            ht_pos=h->elem[pos].pos_hash;
            ht_child=h->elem[child].pos_hash;
            //actualizar en la ht
            qh->ht->item_list[ht_pos].elem =...
            &(h->elem[pos]);
            qh->ht->item_list[ht_child].elem =...
            &(h->elem[child]);
        }

        else
            break;
    }
}
```

Código 2: Código para algoritmo `heapify down`.

III. Resultados y discusión

El uso de punteros para que ambas estructuras apuntaran hacia el mismo elemento (y no solo copias) fue muy útil para la función de insertar, ya que cada estructura modificaba directamente el lugar en el heap y la hash table, y a primera impresión es más eficiente en el uso de memoria. Sin embargo, gracias a que el heap está moviendo los datos constantemente, había que tener cuidado para actualizar la dirección a la que apuntaba la hash table. Además, el uso de punteros, a veces puede hacer más lenta la aplicación debido a los saltos en memoria para accesar a ellos. Aunado a lo anterior, es posible que guardar un puntero requiera más memoria que guardar una copia.

Otro asunto que podría ser mejorado es la forma en la que se insertan los valores en la tabla hash, ya que se convirtió a cadena de caracteres (por el problema de números negativos), y después a este se le genera una llave. Además de correr un pequeño riesgo de tener llaves duplicadas, también el proceso de conversión puede ser un poco costoso.

Las instrucciones: `insert`, `lookup`, `delete`, `deleteMin`, `print`; se leen de un archivo de texto (véase Figura 1a). La salida obtenida se muestra en la Figura 1b.

```
insert 5
insert -25
insert 34
lookup 20
insert 20
lookup 20
insert 20
lookup 20
print
deleteMin
delete -50
print
delete 5
print
deleteMin
print
delete 20
deleteMin
print
deleteMin
```

(a) Entrada.

```
Insertando 5. #copias: 1
Insertando -25. #copias: 1
Insertando 34. #copias: 1
No se encontro 20
Insertando 20. #copias: 1
Elemento 20 encontrado. #copias: 1
Ya existe 20. #copias: 2
Elemento 20 encontrado. #copias: 2
HEAP: [  -25  5  34  20  ]
Elemento -25 eliminado.
No existe el elemento -50.
HEAP: [  5  20  34  ]
Elemento 5 eliminado.
HEAP: [  20  34  ]
Contador de 20 decrementado. #copias: 1
HEAP: [  20  34  ]
Elemento 20 eliminado.
Elemento 34 eliminado.
HEAP: [  ]
Estructura vacia.
```

(b) Salida.

Figura 1: Resultados de aplicar operaciones de Quash.

Se puede ver que se implementó satisfactoriamente la funcionalidad de la estructura de datos compuesta *Quash*. El usuario, en general solo ve el heap, donde la tabla hash es una mera estructura auxiliar invisible para el usuario.

También podría ser incluida la función de expandir el heap según sea necesario, teniendo el debido cuidado de actualizar las direcciones en la tabla hash.

IV. Conclusión

Es posible usar estructuras de datos compuestas para complementar la funcionalidad de diversas estructuras; de esta manera se abren nuevas oportunidades, teniendo el mejor rendimiento de cada estructura, o que sirva como auxiliar (aunque para el usuario parezca invisible); aunque puede ser un tanto ineficiente en cuestión de memoria.

En este caso, se implementó satisfactoriamente la estructura compuesta *Quash* combinando un *min-heap* y una *tabla hash*. Así, se usó la propiedad de la tabla hash de búsqueda en tiempo constante para mejorar la falta de búsqueda de un elemento en el min-heap; y vice versa, con el min-heap se usó como cola de prioridad, superando de esta manera, la limitación de la hash table por su falta de orden en los elementos.

Al usar este tipo de estructuras compuestas, es necesario ser muy cuidadosos en las cuestiones de transacción, es decir, que el cambio que haga una estructura se vea reflejado en la otra.

También, estructuras de datos, aparentemente con una forma compleja, es posible representarla con otras estructuras más sencillas, tales como arreglos o listas ligadas; usando expresiones para relacionar las respectivas posiciones.

Referencias

[1] *Binary heap*, en *Wikipedia*, Page Version ID: 1001780295, 21 de ene. de 2021. dirección: https://en.wikipedia.org/w/index.php?title=Binary_heap&oldid=1001780295 (visitado 17-04-2021).