Estructura de Datos (2023-24, Grupo A)

Montículos Maxifóbicos

Un montículo (heap) es un árbol que verifica la **Propiedad de Orden de los Montículos** (**HOP**): para cualquier elemento en el árbol, excepto la raíz, el valor del elemento es mayor o igual que el de su padre. Este *invariante* debe ser preservado por todas las operaciones que modifiquen el montículo. Gracias a esta propiedad, el elemento mínimo de un montículo siempre está en la raíz del árbol, por lo que la operación minElem es muy eficiente (es O(1)).

Con objeto de lograr un buen rendimiento para las operaciones insert y delMin, se suele impone una restricción estructural adicional al montículo. En el caso de las **Montículos Binarios Completos** estudiados en clase, el montículo debe ser siempre un **árbol binario completo**. Además de la propiedad HOP, todas las operaciones de manipulación del montículo deberán preservar este segundo invariante. Una propiedad importante de los árboles binarios completos es que la altura del árbol es logarítmica con respecto al número total de elementos almacenados en el árbol. Debido a esta razón, la altura de cualquier Montículo Binario Completo es logarítmica con respecto al número de elementos almacenados en el montículo (la altura es muy pequeña en comparación con el número de elementos almacenados en la estructura de datos). Las operaciones insert y delMin para estos montículos se pueden diseñar de manera que el número de pasos realizados sea proporcional a la altura del montículo, y, por tanto, estas dos operaciones se ejecutan en tiempo O(log n).

Otra forma de implementar un montículo de manera eficiente que también hemos estudiado es usando **Montículos Zurdos** (**WBLH**). Un árbol binario es zurdo si, para cualquier nodo en el árbol, el peso de su hijo izquierdo es mayor o igual al peso de su hijo derecho (el peso de un árbol es el número total de elementos almacenados en ese árbol). Un Montículo Zurdo es un árbol binario zurdo que, además, verifica HOP. Todas las operaciones de manipulación del montículo deben preservar estos dos invariantes. Debido a HOP, el elemento mínimo está en la raíz del árbol y minElem es una operación O(1). La espina derecha de un árbol binario es el camino obtenido si partimos desde la raíz del árbol y descendemos siempre a lo largo del hijo derecho de cada nodo, hasta que se alcance una hoja del árbol. Obsérvese que, debido a la propiedad HOP, los elementos en la espina derecha de un montículo estarán ordenados ascendentemente. Otra propiedad importante es que, para cualquier árbol zurdo, la longitud de su espina derecha es logarítmica con respecto al número total de elementos almacenados en el árbol (la longitud de la espina derecha es muy pequeña en comparación con el número de elementos almacenado en la estructura de datos). Una operación que mezcle, o combine, dos WBLHs y devuelva otro WBLH (merge) puede ser diseñada de manera que sólo sea necesario comparar los elementos en espinas derechas de los montículos de entrada, y, por lo tanto, la operación merge se ejecutará en tiempo logarítmico. Finalmente, las operaciones insert y delMin para WBLHs se pueden implementar utilizando merge, y por esta razón, las eficiencias de estas operaciones son también O (log n).

En esta sesión de laboratorio tendrás que implementar otra clase de montículos: los **Montículos Maxifóbicos** (**MXH**) (véase el ejercicio 8 de la cuarta relación de ejercicios). Un Montículo Maxifóbico es un árbol binario que verifica la HOP. El siguiente tipo de datos aumentado será usado para representar un MXH:

data Heap a = Empty | Node a Int (Heap a) (Heap a) **deriving** Show

donde el valor Int en el Node almacenará el peso del nodo (el número total de elementos en el árbol cuya raíz es dicho nodo).

Para estos montículos no se impone ninguna otra condición estructural, pero la operación de combinación (merge) está diseñada de una manera muy específica, de modo que se pueda garantizar una buena eficiencia. Para esta estructura de datos, merge debe tomar dos MXHs y debe volver otro MXH (su fusión o combinación). Como los montículos de entrada verifican la HOP, podemos determinar la raíz del montículo fusionado comparando sólo las raíces de montículos de entrada (la raíz más pequeña de las dos entradas sobrevive como la raíz del resultado combinado). Ahora, nos quedan tres montículos:

- El montículo de entrada cuya raíz no fue seleccionada,
- El hijo izquierdo del montículo de entrada cuya raíz fue seleccionada,
- El hijo derecho del montículo de entrada cuya raíz fue seleccionada.

Merge debe seleccionar el montículo más grande (con respecto a su peso o el número total de elementos) de entre los tres anteriores y usarlo como uno de los hijos (no importa si derecho o izquierdo) del nodo correspondiente al montículo mezcla. Además, debe mezclar de forma recursiva los otros dos montículos (aquellos con pesos más pequeños) con objeto de obtener el otro hijo del nodo resultado. Hay que tener en cuenta que merge siempre fusiona los dos montículos más pequeños y evita considerar para la mezcla el más grande, y de ahí el nombre de esta estructura de datos (Maxifóbico = evita la fusión del más grande de los tres montículos). Las operaciones insert y delMin para MXHs se pueden implementar de forma trivial usando merge, de forma similar a como se hace en el caso de los Montículos Zurdos.

En esta sesión de laboratorio**,** tendrás que:

- Implementar un módulo Haskell (DataStructures.Heap.MaxiphobicHeap) que proporcione un tipo para montículos maxifóbicos junto con las operaciones empty, isEmpty, merge, insert, minElem y delMin. La operación merge se usará para implementar insert y delMin, de forma similar a como se ha hecho en clase para los Montículos Zurdos.
- Analizar la complejidad de merge y las demás operaciones (incluye tu análisis como comentarios en el fichero).
- Utilizar los módulos Haskell proporcionados DataStructures.PriorityQueue.MaxiphobicPriorityQueue (que implementa una cola de prioridad por delegación sobre tu implementación de MXH) y DataStructures.PriorityQueue.PriorityQueueAxioms para comprobar con QuickCheck que tu implementación de MXH verifica las propiedades de las colas de prioridad vistas en clase (sólo tienes que ejecutar la función priorityQueueAxioms ya definida en último módulo).
- Escribir una implementación en Java de Montículos Maxifóbicos.

Puedes descargar los códigos Haskell y Java en el enlace que aparece al final de esta página. Sólo tendrás que editar los ficheros DataStructures.Heap.MaxiphobicHeap.hs y dataStructures.heap.MaxiphobicHeap.java para implementar tus soluciones,.

Descarga de códigos Haskell y Java para esta práctica.

Última modificación: domingo, 19 de noviembre de 2023, 18:15

◄ Código para conjuntos lineales ordenados en Java

Saltar a...

Enunciado ►