

TRABAJO PRÁCTICO

Algoritmo y Estructura de Datos II

Tema: Heaps y Prim

Cuatrimestre: 1 – Año: 2022

Universidad Argentina de la Empresa

Facultad de Ingeniería y Ciencias Exactas

Integrantes del Grupo Nro. 2:

Maio, Carolina Belén – LU1139724

Pancetti, Franco

Samokec, Iván – LU1139058

Docentes:

Wehbe, Ricardo Abraham

Barrera, Elizabeth Gabriela Del Valle

Índice

Introducción	3
Árboles binarios y heaps	3
Implementación de Heaps	5
Métodos privados	5
Métodos públicos	6
Link al código	6
Grafos	7
Introducción y tipos	7
Prim	8
Introducción al algoritmo	8
Ejemplo de ejecución	9
Implementación	10
Links a la implementación y GrafoND TDA utilizado	11
Conclusión	12
Bibliografía	13

Introducción

En este documento se explicarán conceptos generales de Árboles Binarios, cómo estos sirven para crear estructuras de tipo *heaps* y su implementación. A su vez, se detallará el funcionamiento del algoritmo *prim* y los fundamentos de grafos para poder comprender el mismo.

Entonces, **¿Qué es un árbol binario?** Un árbol binario es un conjunto finito de nodos, el que puede ser, un conjunto vacío o un conjunto que contiene un nodo raíz y dos árboles binarios disjuntos, llamados subárbol izquierdo y subárbol derecho.

Cada uno de los nodos de un árbol binario puede tener, a lo sumo, dos ramas. Así, un nodo de un árbol binario, puede tener 0, 1 o 2 hijos.

Un **Heap** es un árbol binario el cual además de las mencionadas anteriormente, cumple con las siguientes propiedades:

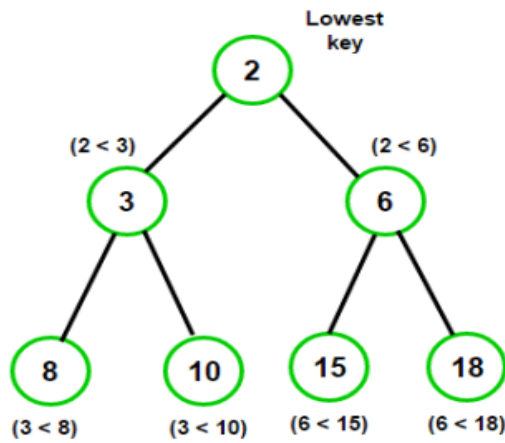
El árbol debe estar *completo*, es decir que todos sus niveles están completamente llenos con excepción del último nivel, el cual puede no haber algún hijo de la raíz del subárbol superior, en este caso el último nivel se almacenan las “claves” lo más a la izquierda posible. Esta propiedad de Heaps permite que estos sean óptimos para ser almacenados en arreglos o Arrays. El mayor elemento (o el menor, dependiendo de la relación de orden escogida) está siempre en el nodo raíz. Por esta razón, los heaps son útiles para implementar colas de prioridad.

Los heaps pueden ser estructurados de dos maneras, Min Heap y Max Heap.

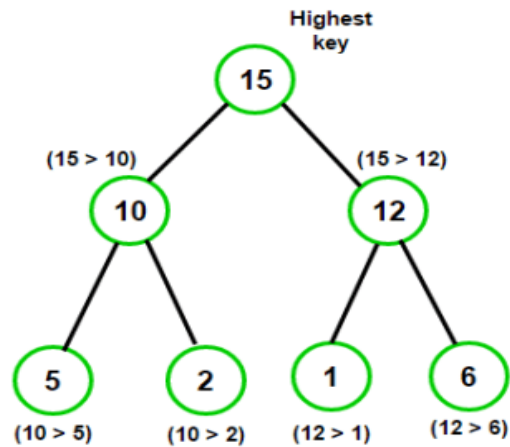
En el primer caso la clave de la raíz debe ser menor que el resto de las claves presentes en el árbol, y esta propiedad debe ser recursivamente verdadera para todos los subárboles que se encuentren por debajo de la raíz principal.

Y en el segundo caso, de manera similar al anterior, la clave de la raíz debe ser mayor al resto de las claves que contiene el árbol, y recursivamente se debe cumplir para los subárboles inferiores.

Ejemplos de Min y Max Heap:



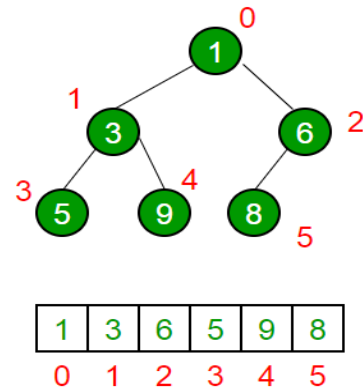
Min Heap
(Parent key is less than or equal to (\leq) the child key)



Max Heap
(Parent key is greater than or equal to (\geq) the child key)

Para lograr representar el árbol, se suele implementar en un arreglo, en donde el ordenamiento se realiza por nivel, siendo así la primer posición del arreglo el nodo padre, y sus hijos la segunda y tercera posición, y así sucesivamente. Entonces, para poder consultar un nodo del árbol podemos utilizar el manejo del arreglo de la siguiente manera:

- Arr[(i-1)/2]** Retorna el nodo padre
- Arr[(2*i)+1]** Retorna el hijo izquierdo
- Arr[(2*i)+2]** Retorna el hijo derecho



Entonces, si estamos en la posición 1 del arreglo que está representando el nodo con clave 3 en el árbol podemos aplicar estas relaciones aritméticas para hallar su nodo padre, y sus hijos... Arr[(2*1)+1] = Arr[3] que sería el nodo con clave 5, es decir su hijo izquierdo.

Implementación

Para este trabajo se realizará una implementación estática, definiendo en primer lugar las siguientes variables globales:

- *isMax*, de tipo booleano, el cual será True si hablamos de un max-heap, o False si hablamos de un min-heap.
- *cant*, de tipo entero, que nos indicará la cantidad de elementos en el árbol.
- *array de enteros*, el cual representará al Heap.

Crearemos cinco métodos privados que nos ayudarán con los métodos restantes:

- *int Padre (int i)*: recibe un índice, y nos devuelve su índice padre.
- *int HijoIzq (int i)*: recibe un índice, y nos devuelve el índice de su hijo izquierdo.
- *int HijoDer (int i)*: recibe un índice, y nos devuelve el índice de su hijo derecho.
- *void SubirNodo (int i)*: Éste método lo usaremos a la hora de insertar. Recibe como parámetro un índice y va a comparar el elemento que se encuentra en dicho índice, llamémoslo X, con su elemento padre. En caso de ser necesario, se intercambiará X con su padre. Esto se repetirá hasta que X quedé en la posición correcta.
- *void BajarNodo(int x, int i)*: A éste método lo vamos a llamar cuando eliminamos un elemento. BajarNodo comparará el elemento X que recibe, con su hijo izquierdo o hijo derecho.

Si hablamos de un max-heap, buscaremos el hijo mayor y, si hablamos de un min-heap, seleccionaremos el hijo menor. Para decidir qué hijo usar, los compararemos y almacenaremos el valor en la variable *hijoSeleccionado*. En caso de ser necesario, se intercambiará al elemento con su *hijoSeleccionado*. Esto se ejecutará hasta que el elemento llegué a su posición correcta para mantener las propiedades de un Heap.

InicializarHeap: Este método recibe como parámetro un booleano, el cual determinará si hablamos de un max-heap (si es True) o de un min-heap (si recibe false).

Aquí se inicializa la variable *cant* en cero, se crea un array (*arr*) de enteros de 100 elementos, y se le otorga el valor a la variable *isMax* con el parámetro recibido.

Insertar: Este método recibe como parámetro el elemento que se desea insertar y se agrega como el último elemento del arreglo. Luego se incrementa *cant* en 1, por lo que esta variable irá determinando la cantidad de elementos que hay en el árbol.

Por último, luego de cada inserción, se llama al método privado *SubirNodo*, pasándole como parámetro *cant - 1*, ya que es el índice inicial donde se agregó el nuevo elemento. De esta manera nos aseguramos que el árbol heap mantenga sus propiedades.

Eliminar: Con este método eliminamos el nodo raíz, es decir, el mayor elemento si es un max-heap, o el menor elemento si es un min-heap. Para eliminar la raíz (es decir *arr[0]*) la reemplazamos por el último elemento del heap. Es decir, buscaremos el elemento que se encuentre en el último nivel y más a la derecha.

Luego de este reemplazo, debemos asegurarnos que el número que está ahora en la raíz, esté en su posición correcta. Para esto llamamos al método privado *BajarNodo*, y le pasaremos como parámetros el elemento que estamos evaluando, y cero como índice inicial.

Primero: Retorna el nodo raíz. Es decir, retorna el mayor (si es max-heap) o el menor (si es min-heap) elemento del heap. La raíz va a estar siempre en la posición cero del arreglo.

HeapVacio: Retorna True o False dependiendo si el heap está vacío o no. Para esto, evaluamos si la cantidad de elementos (variable *cant*) es igual a cero.

Interfaz: [heapTDA.java](#)

Código de implementación: [heap.java](#)

Grafos

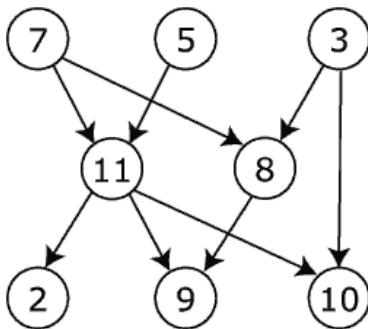
Grafos

Formalmente, un grafo G consiste en dos conjuntos finitos N y A . N es el conjunto de elementos del grafo, también denominados vértices o nodos. A es el conjunto de arcos, que son las conexiones que se encargan de relacionar los nodos para formar el grafo. Los arcos también son llamados aristas o líneas.

Se dice que dos nodos son adyacentes o vecinos si hay una arista que los conecta. Los nodos adyacentes pueden ser representados por pares (a, b) .

Grafos Dirigidos

Hasta ahora hemos supuesto que una arista conecta dos nodos en ambos sentidos igualmente. Un *grafo dirigido* es aquel en el que las aristas tienen un único sentido. En este caso, una arista se dirige desde el nodo origen hasta el nodo destino



Grafos no dirigidos

Un grafo no dirigido es un tipo de grafo en el cual las aristas representan relaciones simétricas y no tienen un sentido definido, a diferencia del grafo dirigido, en el cual las aristas tienen un sentido y por tanto no son necesariamente simétricas.

Algoritmo de Prim

El algoritmo de Prim es un algoritmo perteneciente a la teoría de los grafos para encontrar un árbol recubridor mínimo en un grafo conexo, no dirigido y cuyas aristas están etiquetadas.

En otras palabras, el algoritmo encuentra un subconjunto de aristas que forman un árbol con todos los vértices, donde el peso total de todas las aristas en el árbol es el mínimo posible.

El algoritmo fue diseñado en 1930 por el matemático Vojtech Jarník y luego de manera independiente por el científico computacional Robert C. Prim en 1957 y redescubierto por Dijkstra en 1959. Por esta razón, el algoritmo es también conocido como algoritmo DJP o algoritmo de Jarník.

Descripción conceptual

El algoritmo incrementa continuamente el tamaño de un árbol, comenzando por un vértice inicial al que se le van agregando sucesivamente vértices cuya distancia a los anteriores es mínima. Esto significa que en cada paso, las aristas a considerar son aquellas que inciden en vértices que ya pertenecen al árbol.

El árbol recubridor mínimo está completamente construido cuando no quedan más vértices por agregar.

Funcionamiento del algoritmo de Prim

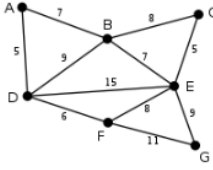
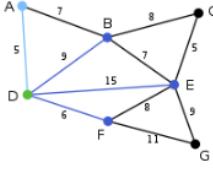
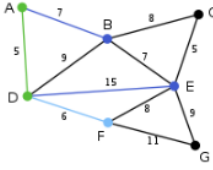
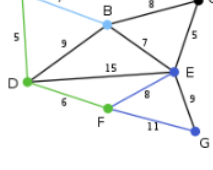
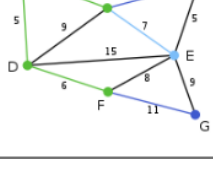
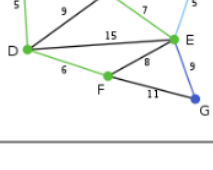
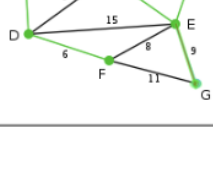
Se marca un vértice cualquiera. Será el vértice de partida.

Se selecciona la arista de menor peso incidente en el vértice seleccionado anteriormente y se selecciona el otro vértice en el que incide dicha arista.

Repetir el paso 2 siempre que la arista elegida enlace un vértice seleccionado y otro que no lo esté. Es decir, siempre que la arista elegida no cree ningún ciclo.

El árbol de expansión mínima será encontrado cuando hayan sido seleccionados todos los vértices del grafo.

Ejemplo de ejecución

Grafo	Descripción
	<p>Este es el grafo inicial. Los números indican el peso de las aristas. Se elige de manera aleatoria uno de los vértices que será el vértice de partida. En este caso se ha elegido el vértice D.</p>
	<p>Se selecciona la arista de menor peso de entre todas las incidentes en el vértice D, siempre que la arista seleccionada no cree ningún ciclo. En este caso es la arista AD.</p>
	<p>Ahora se selecciona la arista de menor peso de entre todas las incidentes en los vértices D y A, siempre que la arista seleccionada no cree ningún ciclo. En este caso es la arista DF.</p>
	<p>Se selecciona la arista de menor peso de entre todas las incidentes en los vértices D, A y F, siempre que la arista seleccionada no cree ningún ciclo. En este caso es la arista AB. Llegado a este punto, la arista DB no podrá ser seleccionada, ya que formaría el ciclo ABD.</p>
	<p>Se selecciona la arista de menor peso de entre todas las incidentes en los vértices D, A, F y B, siempre que la arista seleccionada no cree ningún ciclo. En este caso es la arista BE. Llegado a este punto, las aristas DE y EF no podrán ser seleccionadas, ya que formarían los ciclos ABED y ABEFD respectivamente.</p>
	<p>Se selecciona la arista de menor peso de entre todas las incidentes en los vértices D, A, F, B y E, siempre que la arista seleccionada no cree ningún ciclo. En este caso es la arista EC. Llegado a este punto, la arista BC no podrá ser seleccionada, ya que formaría el ciclo BEC.</p>
	<p>Solo que disponible el vértice G, por lo tanto se selecciona la arista de menor peso que incide en dicho vértice. Es la arista EG. Como todos los vértices ya han sido seleccionados el proceso ha terminado. Se ha obtenido el árbol de expansión mínima con un peso de 39.</p>

Implementación Prim

Para esta implementación, en primer lugar se crea un método *Adyacentes*, el cual va a recibir como parámetros un grafo y un vértice (llamémosle v). Este método nos retornará un conjunto con los vértices adyacentes a dicho vértice v (es decir, los que comparten aristas).

Para la función *prim*, en primer lugar inicializamos el Grafo no dirigido *resultado* que retornaremos al final de la función.

Inicializamos dos conjuntos: *pendientes* y *aux_pendientes*. Estos nos servirán para ir almacenando aquellos vértices que aún no son parte del Grafo *resultado*.

Elegimos un vértice para comenzar el nuevo grafo. Solicitamos sus adyacentes y los recorremos, quedándonos solamente con la arista de menor valor y el vértice correspondiente, el cuál se almacenará en la variable *vertice_seleccionado*.

Para este punto ya tendremos:

- El primer vértice (elegido al azar del conjunto *pendientes*)
- Su arista de menor valor, y
- Vértice correspondiente a dicha arista.

Agregaremos dichas variables al grafo *resultado*, los eliminamos de *pendientes* y *aux_pendientes*, y continuamos con el algoritmo.

Para el siguiente ciclo instanciamos otra variable, llamada *vertices_restantes*.

La estrategia consiste en que, mientras *aux_pendientes* aún tenga elementos, es decir, mientras aún haya vértices que no fueron agregados al Grafo Resultado, recorreremos el conjunto *pendientes*.

Por cada *pendiente* (llamémosle p) se buscarán sus adyacentes.

Recorremos sus adyacentes. Si el adyacente se encuentra dentro de los vértices del resultado, quiere decir que éste *pendiente* p puede ser una posibilidad de próximo vértice a incorporar. El próximo paso es evaluar el peso de la arista entre p y el adyacente.

A lo largo del ciclo de *pendientes*, compararemos los distintos pesos de las aristas con sus distintas posibilidades de adyacentes.

Al final de este ciclo, nos quedaremos con la arista de menor valor, el vértice *p* correspondiente, y el adyacente al cuál hay que unirlo, almacenados en las variables *menor_arista*, *vertice_seleccionado*, y *mejor_adyacente*.

Es recién en este punto cuando eliminamos el vértice seleccionado (el cual agregaremos al resultado) del conjunto *aux_pendientes*.

El proceso se vuelve a repetir hasta que ya no haya elementos sin agregar.

El conjunto de pendientes, se vacía y se vuelve a rellenar para el final de cada iteración del ciclo *aux_pendientes*, para esto es que utilizamos el conjunto auxiliar *vertices_restantes*.

Para cuando termine este ciclo y el conjunto *aux_pendientes* esté vacío, el Grafo no direccionado *resultado* estará completo.

[Implementacion Prim](#)

Grafo no dirigido TDA: [Interfaz](#) e [Implementación](#)

Conclusión

A lo largo de este trabajo se profundizó sobre la estructura de datos heaps, los cuales vimos que pueden ser tanto *max* como *min* heaps.

Una de las ventajas de este tipo de datos es que, al ser un árbol completo, puede representarse como un array, por lo cual también simplifica las operaciones en relación a este. De acuerdo a sus características, podemos concluir que si buscamos tener el elemento más grande a mano (o más chico, dependiendo del orden que se eligió), un heap resultaría ideal.

En cuanto al algoritmo de Prim, vimos que es un algoritmo para encontrar el árbol recubridor mínimo partiendo un grafo no dirigido. Es decir, que el algoritmo va a devolvernos un nuevo grafo donde las aristas sean del menor valor posible, y donde todos los vértices están conectados.

Bibliografía

- ❖ [Heap Data Structure](#)
- ❖ [Wikipedia](#)
- ❖ [DCC UChile](#)
- ❖ [ANIEI](#)
- ❖ [UDB - heaps](#)
- ❖ [UDB - Algoritmo de Prim - **prim**](#)
- ❖ [Algoritmo de Prim | Estructura de Datos II](#)
- ❖ **[6 GRAFOS](#)**
- ❖ [HEAPS - MAX HEAP Y MIN HEAP ¿Como funcionan? Código en Java](#)
- ❖ [Heap binario](#)
- ❖ *Sznajdleder, Pablo Algoritmos a fondo : con implementaciones en C y Java . - 1a ed.*