Laboratorio 4

HeapSort

Estructura de Dades

2n Curs GEI
Grup 2

15/12/2020

Ton Llucià Senserrich
47125160T

Eduard Sales Jové
49539818A

Per començar implementarem les funcions: parent, left, right, hasLeft, hasRight i hasParent i el constructor de la classe per a facilitar-nos la feina més tard.

constructor BinaryHeap

En el constructor utilitzem l'array que rebem per paràmetre, reordenant els elements per tal de poder interpretar l'array com un arbre, i per fer tal cosa el que fem és executar la funció heapifyUp per a cada element de l'array i així anirem ascendint els elements en l'arbre tant com sigui necessari per a aconseguir l'estructura desitjada, a cada iteració també augmentarem la mida de l'arbre. Aquest procés també es pot interpretar com que tenim un sufix per ordenar i ordenem cada element d'aquest creant així un prefix ja ordenat, i acabarem quan la mida del sufix sigui 0.

static int parent(int index)

En la funció parent restem 1 a l'índex i ho dividim entre 2 i així aconseguirem l'índex del pare de l'element d'entrada.

static int left(int index)

En tots els casos l'objecte left serà el resultat de multiplicar l'índex del pare per 2 i sumar-li 1.

static int right(int index)

En tots els casos l'objecte right serà el resultat de multiplicar l'índex del pare per 2 i sumar-li 2, o bé sumar-li 1 al pare i després multiplicar-lo per 2.

boolean hasLeft(int index)

La funció retornarà true en cas que l'índex de left sigui més petit que la mida de l'arbre.

boolean hasRight(int index)

La funció retornarà true en cas que el índex de right sigui més petit que la mida de l'arbre.

En les dues funcions, ho comparem amb heapSize, ja que en el cas que ens passessin un índex que fos més gran que la mida de l'arbre, significaria que aquest està fora de l'array i no contaria com un left o un right vàlid.

boolean hasParent(int index)

Farem dues comprovacions, en la primera ens assegurem que no estiguem assenyalant el primer objecte de l'arbre, ja que aquest no te pare.
Després mirarem igual que abans que l'índex del pare sigui més petit que la mida de l'arbre, ja que això vol dir que està dins de l'array

void heapifyDown(int index)

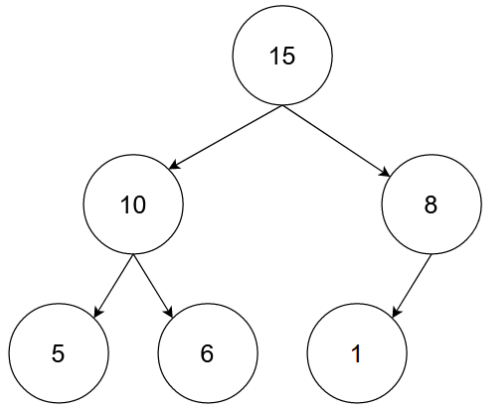
El que aconseguim amb aquesta funció és que la nostra estructura torni a ser un maxHeap, ja que en modificar l'arbre podria deixar de complir els requisits per ser un maxHeap, el que farem en aquesta funció és comprovar si l'element índex és més gran que els seus fills i en cas que sigui més petit canviar-los.
*s'explica amb més detall en la funció remove.

void heapifyUp(int pos)

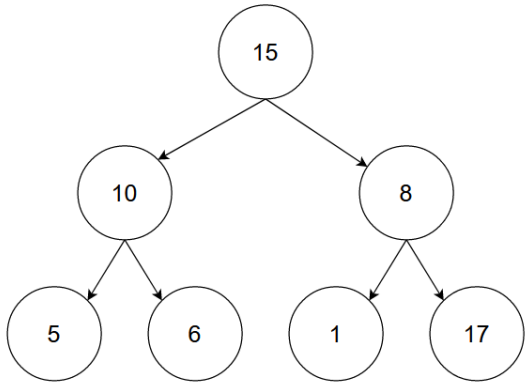
Aquesta funció ordena des de la posició que li passin cap amunt, arribant en el pitjor dels casos, a la posició 0, l'arrel de l'arbre.
Sempre que no ens trobem en la posició 0 i l'objecte pare sigui més petit que l'objecte fill farem un swap entre la posició pare i fill, i farem que pos sigui l'índex del pare per a poder seguir tractant els elements de l'arbre i seguir ordenant.
*s'explica amb més detall en la funció add.

public void add(E elem)

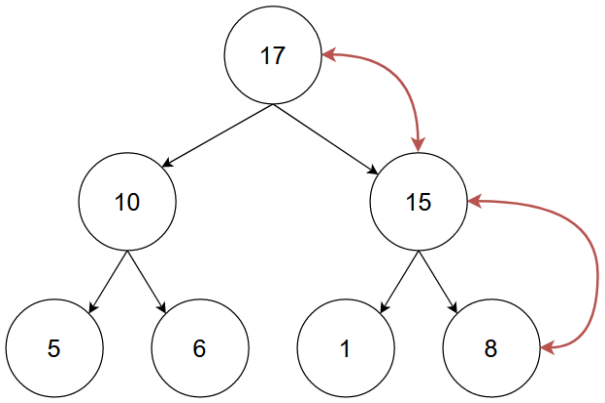
Aquesta funció és utilitzada per afegir un element l'array que representa l'arbre, el plantejament d'aquesta funció consisteix a afegir l'element al final de l'array i cridem a la funció heapifyUp per tal que l'array recuperi la seva estructura de maxHeap, ja que en aquesta inserció la podria haver perdut.



Volem afegir un nou node, per la qual cosa utilitzarem la funció add explicarem com opera la funció add, i en conseqüència la funció heapifyUp.



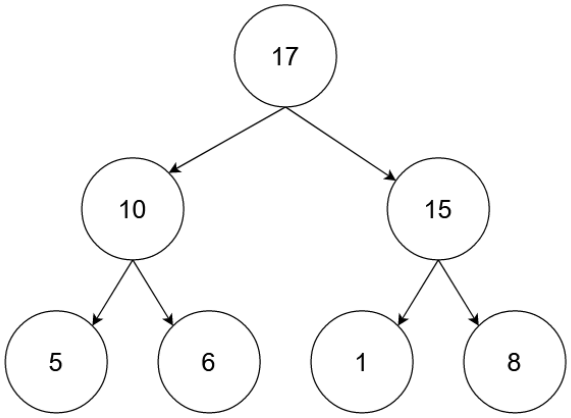
Afegim el nou node al final de la llista i procedim a reordenar-la cridant a la funció heapifyUp.



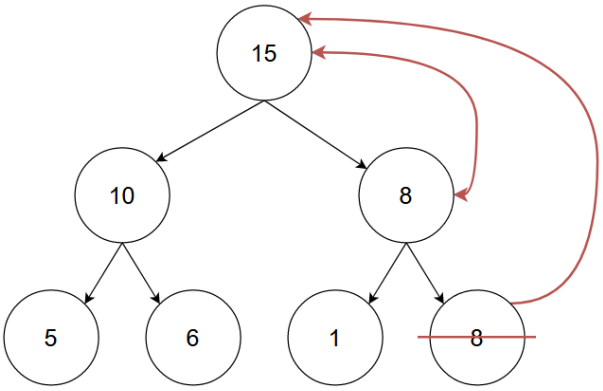
Comprovem si el nou node és més gran que el seu pare i en tal cas fem un swap, fins a arribar a l'arrel o fins a trobar un pare més gran que el seu fill.(les línies vermelles representen els swaps).

public E remove()

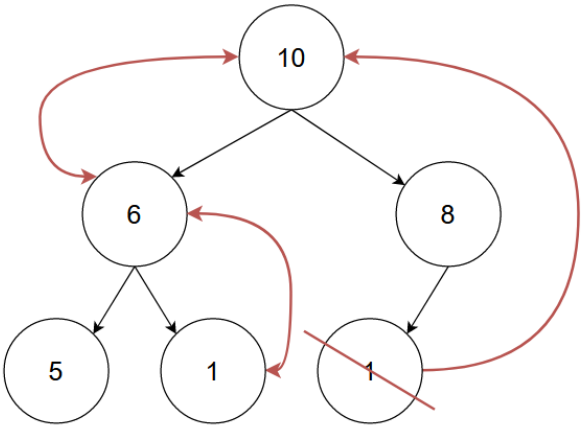
En aquest mètode l'únic que fem és eliminar el primer element(l'arrel) de l'arbre i retornar-lo, però després d'eliminar l'element hem de tenir en compte que el binaryHeap ja no serà un maxHeap i cal reordenar-lo per tant el que fem abans d'eliminar l'element corresponent posem en la seva posició, l'últim element de la llista, per tal de poder-la ordenar novament cridant a la funció hepifyDown.



Suposem que volem eliminar el primer element (l'arrel), que serà el més gran.



En el mètode remove, copiem l'últim element de l'array a la posició 0, excloent així l'element de l'arbre, seguidament apliquem sobre l'element 0 el mètode heapifyDown, el qual reordenarà l'arbre, en aquest cas canviant el 15 pel 8 i finalment suprimim definitivament l'últim element de l'array.



Suposarem que volem eliminar un altre cop l'element en la posició 0 per tal d'observar el comportament de l'algorisme, el mètode remove copiarà l'últim element de l'array a la posició 0, excloent així l'últim element de l'arbre, seguidament aplicarà la funció heapifyDown en la posició 0, la qual intercanviarà l'1 pel 10 i l'1 pel 6, i finalment eliminarà definitivament l'últim element, aquest funcionament es repetiria tants cops com elements volguéssim eliminar, adapten-se a la situació per tal de garantir una estructura maxHeap.

public static void sort(ArrayList list,Comparator<? super E> cmp)
Cridem al constructor amb la llista que rebem com a paràmetre per tal que aquest ordeni adequadament la llista per perquè aquesta es pugui interpretar com un arbre, un cop tenim això el que fem és afegir a l'esquerra de la llista l'element arrel de l'arbre i l'eliminem de l'arbre, si repetim aquesta operació, decrement-ant l'índex on inserim l'element, fins que no quedin elements a l'arbre haurem aconseguit ordenar la llista de petit a gran.
Hem de tenir en compte que totes aquestes operacions es realitzen sobre la mateixa llista que hem rebut per paràmetre i podem interpretar-la com si tinguéssim un prefix que té forma d'arbre i un sufix que té forma de la llista que necessitem.

public static <E extends Comparable<? super E>> void sort(ArrayList<E> list)
En aquesta funció cridarem a la funció sort però ho farem amb un comparador que hem creat nosaltres, ja que la E extends comparable