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 per a facilitar-nos la feina més tard.

static int parent(int index)

Gràcies al fet que la forma en la qual representem l'array està en forma d'arbre binari, veiem que en el cas que sol acceptem nombres enters si li restem 1 a l'índex i després el dividim entre 2 sempre ens donarà l'índex del parent

static int left(int index)

En tots els casos l'objecte left serà el resultat de multiplicar el 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 heap

boolean hasRight(int index)

La funció retornarà true en cas que el índex de right sigui més petit que la mida de heap

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

void heapifyDown(int index)

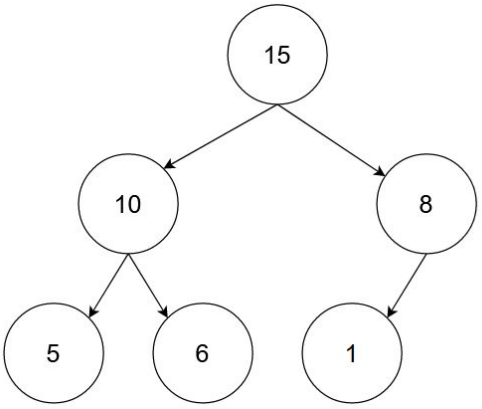
El que aconseguim amb aquesta funció és que la nostra estructura torni a ser un maxHeap,ja que en modificarla podria deixar de cumplir els requisits per ser un maxHeap, el que farem en aquesta funció és comprovar si l'element índex es més gran que els seus fills i en cas que sigui mes petit canviar-los

void heapifyUp(int pos)

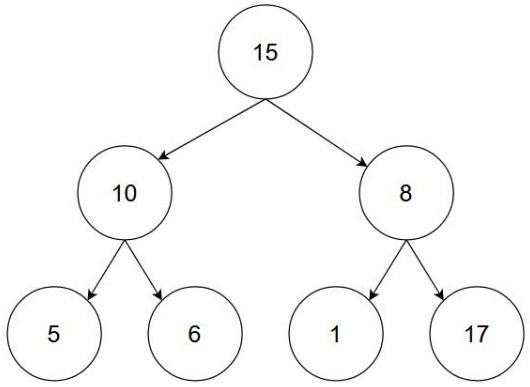
Aquesta funció ordena des de la posició que li passin cap amunt, arribant en el pitjor dels casos, a la posició 0, l'arrel de l'arbre.
Sempre que no ens trobem en la posició 0 i l'objecte pare sigui més petit que l'objecte fill farem un swap entre la posició pare i fill, i farem que pos ara sigui el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

public void add(E elem)

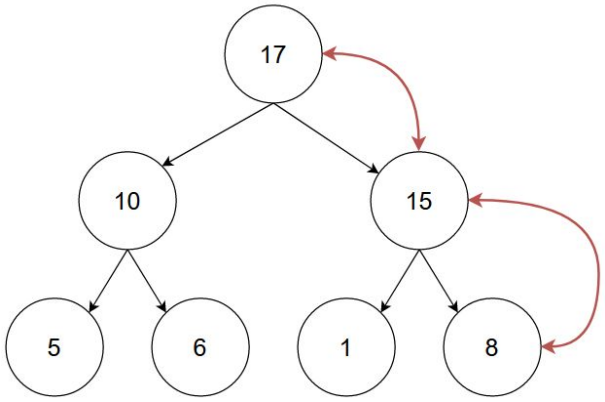
aquesta funció és utilitzada per afegir un element l'array i per tant al binary heap, el plantejament d'aquesta funció consisteix a afegir l'element al final de l'array i cridem a la funció heapifyUp per tal que l'array recuperi la seva estructura de maxHeap, ja que amb aquesta inserció la podria haver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



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



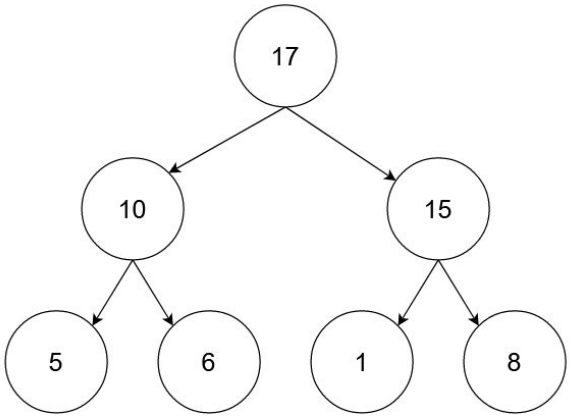
Afegim el nou node al final de la llista i procedim a reordenar-la.



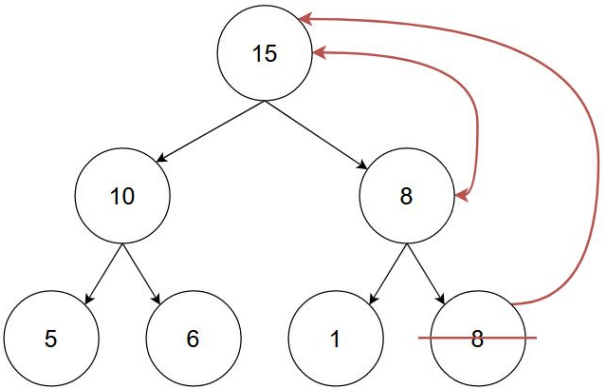
Comprovem si el nou node és més gran que el seu pare i en tal cas fem un swap, fins a arribar a l'arrel o fins a trobar un pare més gran que el seu fill.(les línies vermelles representen els swaps).

public E remove()

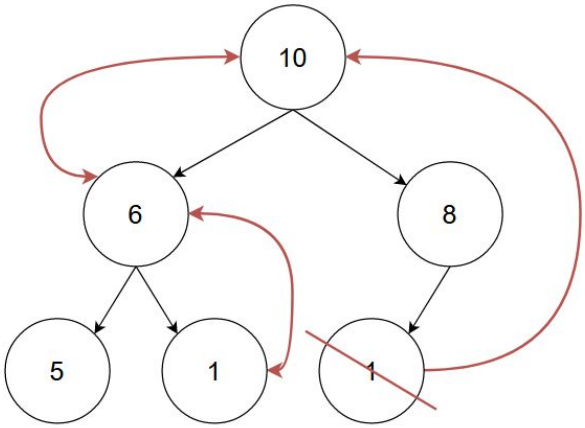
en aquest mètode l'únic que fem és eliminar el primer element(l'arrel) de la llista i retornar-lo, però després d'eliminar l'element hem de tenir en compte que el binaryHeap ja no serà un maxHeap per tant cal reordenar-lo per tant el que fem abans d'eliminar l'element corresponent posem en la seva posició, la 0, 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 el últim element de l'array a la posició 0, i suprimim el últim element de l'array.
Seguidament apliquem sobre l'element 0 el mètode heapifyDown, el qual reordenarà, en aquest cas canviant el 16 pel 8.



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, eliminarà l'últim element i seguidament aplicarà la funció heapifyDown en la posició 0, la qual intercanviarà l'1 pel 10 i l'1 pel 6, aquest esquema 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 per a crear una BinaryHeap
Anirem agafant els elements des de l'última posició fins la numero 0, i mentre els anem tractant els fiquem al binaryHeap que hem creat, els col·loquem a la posició 0 i l'esborrarem de la llista, els ficarem a la posició 0, ja que d'aquesta forma quan tractem el següent i també el fiquem a la posició 0, la funció add per si sola ja mourà l'anterior a la posició 1, de tal manera que no ho hem de fer nosaltres

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