Names: Tom Lapid, Paz Reingold
User Names: tomlapid, pazreingold
Id's: 203135256, 208209734

Documentation

public class HeapNode

Class HeapNode objects represent binomial trees.
In our implementation, a binomial tree is a heap node that contains a linked list of his children, a degree and a value.
Class HeapNode is an inner class of class BinomialHeap.

Class fields:

private HNLinkedList<HeapNode>children – contains the children of the current HeapNode object in a linked list. Initiated in the Class's constructor.

private int degree - contains the degree (number of children) of the HeapNode. Initiated by the user in the Class's constructor.

private int value - contains the value of the node, a natural number. Initiated by the user in the Class's constructor.

Class methods:

Every single method of the followings has a complexity O(1).

public HeapNode(int value, int degree)– HeapNode class constructor. Initiats the value and degree fields by the given parameters, and the children linked list.

Getters –

public HNLinkedList<HeapNode> getChildren() – returns the HeapNode ' s children .

public int getDegree () – returns the HeapNode ' s degree.

public int getValue() – returns the HeapNode ' s value.

Setters –

public void setChildren(HNLinkedList<HeapNode> children) – sets the new children of the HeapNode to the given children linked list.

public void setDegree(int degree) – sets the new degree of the HeapNode to the given degree.

public void setValue(int value) – sets the new value of the HeapNode to the given value.

public class Cell

The Class Cell's object contains the HeapNode object.
Class Cell is an inner class of class HNLinkedList.

Class fields:

private HeapNode node - contains the HeapNode of the cell.

private Cell next – contains the next cell in the linked list
(null for the last cell in the linked list).

Class methods:

Every single method of the followings has a complexity O(1).

public Cell(HeapNode node, Cell next)– a constructor that initialize a new cell by the given parameters –the relevant heap node and the next cell in the linked list.

public Cell next()– returns the next cell in the linked list.

Getters –

public HeapNode getNode()– returns the HeapNode object of the cell .

Setters –

public void setNext(Cell next)– sets the "next" field of the cell/ the next cell in the linked list, to be the given Cell parameter.

public void setNode(HeapNode node)– sets the HeapNode object that the cell contains to the given parameter HeapNode.

public class HNLinkedList

The class HNLinkedList's object represent a linked list of class Cell's objects that contain the HeapNodes.
Class HNLinkedList is an inner class of class BinomialHeap.

Class fields:

private Cell first - contains the first cell of the linked list.

private Cell last - contains the last cell of the linked list.

private int size - contains the number of cells of the linked list.

Class methods:

public HNLinkedList() – The class's constructor , initializes the linked list's fields.
Complexity - O (1).

public Cell getFirst() - returns the first cell of the linked list.
Complexity - O (1).

public Cell getLast() - returns the last cell of the linked list.
Complexity - O (1).

public boolean isEmpty() – returns true if and only if the linked list is empty else returns false.
Complexity - O (1).

public HeapNode removeFirst() – remove the first cell from the linked list.
Complexity - O (1).

public void remove(HeapNode node) – remove the cell contains the given HeapNode from the linked list.
Complexity - O (**log(n**)), since it iterates over the linked list which in Binomial heap case has upper bound of log(n).

public void addFirst(HeapNode node) – initializes a new cell contains the given HeapNode and insert it as the first cell in the linked list.
Complexity - O (1).

public void addLast(HeapNode node) - initializes a new cell contains the given HeapNode and insert it as the last cell in the linked list.
Complexity - O (1).

public Integer size()– returns the number of cells in the linked list.
Complexity - O (1).

public class BinomialHeap

The Class BinomialHeap object represent a binomial heap using HNLinkedList and HeapNode classes.

Class fields:

private HeapNode heapMin – contains a pointer to the HeapNode's object with the minimal value in the heap.

private int heapSize – contains the size of the heap (i.e. the number of HeapNode's objects in the heap). Initiated in the Class's constructor to be 0.

private HNLinkedList<HeapNode> theHeap – contains the roots of the binomial trees in a linked list. Initiated in the Class's constructor.

Class methods:

Every single method under the getters and the setters has a complexity O (1).

public BinomialHeap() - BinomialHeap class constructor. Initiates the size of the heap and its HeapNode's linked list.

Getters –

public HNLinkedList<HeapNode> getTheHeap() – returns a linked list of the binomial heap's roots.

Setters –

public void setTheHeap(HNLinkedList<HeapNode> theHeap) – sets the linked list that contains the root nodes of the binomial heap to the given HNLinkedList.

public void setHeapMin(HeapNode heapMin) – set the HeapMin node to the given HeapNode

Additional methods –

public boolean empty() – returns true if and only if the binomial heap is empty, else returns false.
Complexity - O (1).

public void insert(int value) – inserts a HeapNode with the given value to the Binomial Heap.
The method first initiates a new HeapNode that includes the given value and degree 0.
While the HeapNode that we want to insert has the same degree as the first HeapNode on the Heap's linked list (which has the minimal degree by our implementation), the method melds the two of them, removes the first HeapNode from the list and tries again to insert the new melded HeapNode. Once the HeapNode we try to insert does not have the same degree as the first HeapNode on the linked list (or the linked list is empty), the method inserts it as the first HeapNode and update the relevant fields.
Calls to: meldNodes.
Complexity - **O (logn)** - since it iterates over a O (logn) size linked list.

private HeapNode meldNodes (HeapNode tree1, HeapNode tree2) – adds tree2 to the children linked list of tree1 if the value of tree 1 is smaller or equals the value of tree 2, otherwise adds tree 1 to the children linked list of tree2. Later the method updates the relevant tree's degree.
Complexity - O (1).

public void deleteMin() – deletes the HeapNode with the minimum value ("HeapMin") from the Binomial heap. The heap must remain valid (keep its invariants).
The method first creates a new heap from the HeapMin's children, remove HeapMin from the heap and update its size, and then uses the helper method "meld" in order the meld the original heap with the children heap.
Calls to: meld.
Complexity - **O (logn)** - since it has log(n) helper method.

public int findMin() - returns the HeapNode with the smallest value in the heap ("HeapMin") or -1 if the heap is empty.
Complexity - O (1).

public void meld (BinomialHeap heap2) – melding a given heap to the current heap while keeping
All the binomial heap invariants.
The method keep along the process 3 variables - the current cell of each heap and a balance
HeapNode which represents the remainder of the last HeapNodes melding. Now we got 2 main cases:
In the first one, the current cells of the two heaps are not null (we still have cells to iterate over in both heaps), and in that case we got 5 cases to deal with-

A. We got a balance with a smaller degree than the two cells's HeapNodes degrees.in that case the method simply adds the balance HeapNode at the end of the melded heap and sets balance to be empty.

B. The three variable's degrees are equal. In that case the method adds the balance HeapNode at the end of the melded heap and sets the new balance to be the melded HeapNode of The two Cell's HeapNodes.

C. Both Cell's HeapNodes degrees are equal and the balance is empty. In that case the method sets the new balance to be the melded HeapNode of The two Cell's HeapNodes.

D. There is a balance and his degree equals only the degree of the HeapNode with the smaller degree between the two. In the case the method sets the new balance to be the melded HeapNode of the original balance and the HeapNode with the smaller degree.

E. The balance is empty and the two HeapNodes of the Heaps have different degrees. In that case the method adds the HeapNode with the smaller degree at the end of the melded heap.

In the second main case at most one of the current heaps cells is not null (we got at most one heap to finish iterate on) and in that case the method calls to the helper method "meldWithOneEmptyHeap" to finish the melding.

During the method we update the Heap's relevant fields.
Calls to: meldNodes, smallerDegreeNode, meldWithOneEmptyHeap, updateHeapMin.
Complexity - O (logn) since overall, including our helper methods, we iterate over linked lists of heap roots which has O (logn) items**.**

private void updateHeapMin(HNLinkedList<HeapNode> Result) – update the field heapMin to point the HeapNode's object with the minimum value.
Complexity - **O (logn)** - since it iterates over an O (logn) size linked list.

private HNLinkedList<HeapNode> meldWithOneEmptyHeap - The method gets 3 parameters – the current cell of the heap we need to iterate on, the melded heap and the updated balance.
The method iterates over the heap of the given cell and deal with 3 cases:

A. There is a balance and his degree is smaller than the current cell's HeapNode's degree. In that case the method adds the balance HeapNode at the end of the melded heap.
B. There is a balance and his degree is equal to the current cell ' s HeapNode ' s degree. In that case the method sets the balance to be the melded HeapNode of the two.
C. There is no balance. In that case the method adds the current cell's HeapNode at the end of the melded heap.

After finish iterating over the given heap if the balance is not empty the method adds it at the end of the melded heap. Finally it returns the updated melded heap.
Calls to: meldNodes
Complexity - O (logn) since it could iterate over an O (logn) size linked list.

private HeapNode smallerDegreeNode(HeapNode node1,HeapNode node2) – returns the HeapNode with the smaller value.
Complexity - O (1).

public int size() – returns the number of HeapNodes exist in the all heap (not just the roots).
Complexity - O (1).

public int minTreeRank() – returns the root of the heap with the minimal degree (the first root in the linked list according to our implementation).
Complexity - O (1).

public boolean[] binaryRep() - returns an array containing the binary representation of the heap using a boolean array which "true" represent "1" and "false" represent "0" in this array.
Complexity- O (logn) – since it iterates over an O (logn) size linked list.

public void arrayToHeap(**int**[] array) – deletes the previous elements in the heap if it wasn't empty, and then inserts the elements in the array to the heap.
Calls to: insert.
Complexity- O (nlogn) – since it inserts n items using a O (logn) time complexity method ("insert") for each one of them.

public boolean isValid() – returns true if and only if the binomial heap is valid.
The method walks through the heap linked list and checks:
A. Each one of the heap's roots is a valid HeapNode (which suggest that every single HeapNode in the all heap is valid because of the recursive helper method which will be detailed next).
B. The heap doesn't have more than one root with the same rank.
Calls to: isValidBinTree
Complexity - O (n) – because overall, including the helper method, this method checks once each HeapNode of the heap.

public boolean isValidBinTree(HeapNode node) – returns true if and only if the heap node(binomial tree) is valid.
The method uses recursive calls based on the next consumptions about a HeapNode with degree k:
A. It must has k children.
B. The HeapNode ' s value must be smaller the all of his children ' s values.
C. The degree of his children is no more than k-1.
D. It has exactly one child of each rank between 0 to k-1.
Complexity - O (n) since it goes recursively through the all binomial tree and visits every node once.

מדידות -

1. לצורך תיאור סדרת m הפעולות נבחר m כך שמתקיים ש- $m=2^k$ עבור k טבעי כלשהו. הסדרה שלנו תבוצע באופן הבא – ראשית נבצע m-1 הכנסות של איברים לערימה הבינומית שלנו, כשערכם נקבע באופן שרירותי, לדוגמה 1 עד m-1 .
כפי שנלמד בהרצאה, זמן האמורטייזד של פעולת הכנסה לערימה בינומית הוא O (1) ולכן לסדרה זו זמן אמורטייזד קבוע לפעולה.
נציין שמכיוון ש $m-1 = 2^k-1$, כאשר נתבונן בייצוג הבינארי של ערימה זו הוא יראה כך "1....11" באורך k-1, ועל פי מה שנלמד בהרצאה זה בדיוק אומר שבערימה הבינומית שלנו יהיה עץ בינומי אחד מכל דרגה 1 עד k-1.
כעת, כשנכניס את האיבר הm הייצוג הבינארי של הערימה יהפוך ל"0...10" באורך k ביטים, ועל כן כמו שלמדנו בקורס כל העצים הבינומיים יתמזגו לעץ בינומי יחיד מדרגה k. מכיוון שמתבצעים כk מיזוגים עלות ההכנסה ה-m הינה O(k) מכיוון שכל איחוד בודד של שני עצים רץ בזמן קבוע, ומכיוון ש$m=2^k$ אז הפעולה הm בעצם עולה לנו $\Theta \log(m)$.

2.מכיוון שבסעיף א' ביצענו m פעולות insert ואנו יודעים שזמן האמורטייזד של הכנסה לערימה בינומית הינו O(1) אז זמן הריצה הכולל מבחינה אסיפמטוטית של *m* הפעולות הינו O(m).

3.

m	לאחר m-1 פעולות	לאחר m פעולות
1000	"1111100111"	"1111101000 "
2000	"11111001111"	"11111010000"
3000	"101110110111"	"101110111000"

1.**<u>מספר הלינקים :</u>**

m	לאחר m-1 פעולות
1000	991
2000	1990
3000	2990

נשים לב כי עבור כל גודל של ערימה (לדוגמה ערימה בגודל 999) מספר הלינקים הינו בדיוק מספר האיברים בערימה פחות מספר שורשי הערימה (לפי הייצוג הבינארי של 999 - "1111100111" ניתן ללמוד כי יש לה בדיוק 8 שורשים ואכן 991 = 999-8= links #), וזאת מכיוון שאלו האיברים היחידים שאינם מקושרים לאיבר נוסף מעליהם.

2. **<u>מספר הלינקים :</u>**

m	לאחר m פעולות
1000	994
2000	1994
3000	2993

m	לאחר m-1 פעולות	לאחר m פעולות
1000	[true, true, true, false, false, true, true, true, true, true]	[false, false, false, true, false, true, true, true, true, true]
2000	[true, true, true, true, false, false, true, true, true, true]	[false, false, false, false, true, false, true, true, true, true]
3000	[true, true, true, false, true, true, false, true, true, true, false, true]	[false, false, false, true, true, true, false, true, true, true, false, true]

אכן באותו אופן כמו בסעיף 1 קיבלנו בדיוק את מספר הלינקים שציפינו לכל גודל של m בהתאם למספר השורשים שלו שניתן לראות בייצוג הבינארי.

-binaryRep()

ניתן לראות כי במקביל לייצוג הבינארי בו ראינו שכמות ה"1" ים הינה כמות השורשים בערימה, במימוש של המתודה binaryRep() מספר הערכי true הם אלו המסמנים את השורשים .