קובץ תיעוד – פרויקט מעשי 1 מבני נתונים

שם משתמש 1: lb3

שם משתמש 2: royfainaru

תעודת זהות 1: 206913519

תעודת זהות 2 : 207002403

שם מלא 1: ליאור בן אברהם

שם מלא 2: רועי פיינרו

**חלק מעשי**

תיאור המחלקות שמומשו ותיעוד הסיבוכיות:

|  |  |
| --- | --- |
| **Method Name** | **Time Complexity** |
| **isEmpty()** | O(1) |
| **insert(int i)** | O(1) |
| **deleteMin()** | O(n) |
| **findMin()** | O(1) |
| **meld(FibonacciHeap heap2)** | O(1) |
| **size()** | O(1) |
| **countersRep()** | O(n) |
| **delete(HeapNode x)** | O(n) |
| **decreaseKey(HeapNode x, int d)** | O(n) |
| **nonMarked()** | O(1) |
| **potential()** | O(1) |
| **totalLinks()** | O(1) |
| **totalCuts()** | O(1) |
| **kMin()** | O(k\*Deg(H)) |

Fibonacci Heap required mandatory methods:

Full time complexity description of the methods incl. our added methods:

1. **class FibonacciHeap:**

private final NodeFactory nodeFactory:

Initialized node factory class instance, new node creation.

LinkedList rootList:

The linked list object representing the top layer of the heap, contains the roots of the trees in the heap forest.

private static int links:

Counter for total links

private static int cuts:

Counter for total cuts

private int marked:

Counter for number of current marked nodes in heap

public boolean isEmpty():

time complexity: O(1)  
 Returns true if and only if the heap is empty.

public HeapNode insert(int key):

time complexity: O(1) – lazy insert

Creates a node (of type HeapNode) which contains the given key, and inserts it into the heap.

The added key is assumed not to already belong to the heap.

Returns the newly created node.

The node is inserted using the linked list method insertFirst to the root list of the heap (representing the forest) as a tree of rank 0.

public void deleteMin():

time complexity: O(n)

Deletes the node containing the minimum key.

Uses the saved pointer to find the minimum, and deletes it using the root list methods as a linked list object, because the minNode will always be a root node in the forest.

It then calls to reOrganize() that consolidates the heap, and after consolidation, it calls the linked list updateMin() method to find the new minNode at minimum time.

private static double log(double base, double arg):

time complexity: O(1)

Computes log function according to the given base and number, using Math.log10()

private int maxRankUpperBound():

time complexity: O(1)

Calculate the upper bound of the maximum rank in the heap using Fibonacci numbers according to the golden ratio, used in the Fibonacci number formula.

private HeapNode link(HeapNode[] query, HeapNode node):

worst case: O(log(n))

Recursive method that performs 'successive' linking, triggered by a given query.

Since the number of trees is bounded by logn, the max number of links the method will perform is O(logn)

private void consolidate():

time complexity: O(n)

Runs the linking process in the heap after deleteMin() by creating a link query and calling the link method.

public void increaseCuts():

time complexity: O(1)

Increases the static cuts counter by one.

public void increaseMarked():

time complexity: O(1)

Increases the marks counter by one.

public void decreaseMarked():

time complexity: O(1)

Decreases the marks counter by one.

public FibonacciHeap.HeapNode findMin():

time complexity: O(1)

Returns the node of the heap whose key is minimal, or null if the heap is empty, by returning a saved reference to the minNode.

public void meld(FibonacciHeap heap2):

Melds heap2 with the current heap, by concatenating (annexing) the second heap linked list forest, the the current heap linked list forest.

public int size():

time complexity: O(1)

Returns the number of nodes in the heap, by using the linked list size field.

public int[] countersRep():

time complexity: O(n)

Returns an array of counters. The i-th entry contains the number of trees of order i in the heap.

private int getMaxRank():

time complexity: O(n)

Finds the maximum existing tree rank in the heap forest.

public void delete(FibonacciHeap.HeapNode x):

time complexity: O(n)

Deletes the node x from the heap, by using helper methods from the Linked list and heapNode classes.

The delete operation is implemented by decreasing the node to the minimal int value possible, and calling deleteMin right after.

public void decreaseKey(FibonacciHeap.HeapNode x, int delta):

time complexity: O(n)

Decreases the key of the node x by the given delta value, by using helper methods from the Linked list and heapNode classes.

public int nonMarked():

time complexity: O(1)

This function returns the current number of non-marked items in the heap.

public int potential():

time complexity: O(1)

This function returns the current potential of the heap, which is:

Potential = #trees + 2\*#marked

public static int totalLinks():

time complexity: O(1)

This static function returns the total number of link operations made during the run-time of the program, by using the static counter.

public static int totalCuts():

This static function returns the total number of cut operations made during the run-time of the program, by using the static counter.

private FibonacciHeap.HeapNode find(int k):

time complexity: O(n)

Find a node with key 'k' in the heap, by running recursively on children linked list all over the heap forest linked list.

public static int[] kMin(FibonacciHeap H, int k):

time complexity: O(k \* deg(H))

This static function returns the smallest k elements in a Fibonacci heap that contains a single tree, by utilizing a helper Fibonacci heap, ‘query’, and following the algorithm:

1. copy the key of the minimal node of this heap to the result array,
2. for k-1 times:
   1. copy the children nodes of the recently added key to the result array, into the query heap
   2. remove the minimal node from the query heap, and copy its key into the result array.
3. **class FibonacciHeap.HeapNode:**

public HeapNode nodeDecreaseKey

Time complexity: O(n).

Helper method for heap.DecreaseKey() - works with list.listDecreaseKey().

In case that this node is not the given node, the method searches recursively for the relevant node, and decreases the node’s key with the given delta value, and cuts the node into the heap forest, if necessary. It also takes care to mark the cut node’s parent node, if necessary.

public boolean hasPrev()

Returns true iff this node’s ‘prev’ reference is not null.

Time complexity: O(1)

public boolean hasNext()

Returns true iff this node’s ‘next’ reference is not null.

Time complexity: O(1)

public boolean hasParent()

Returns true iff this node’s ‘parent’ reference is not null.

Time complexity: O(1)

public int getKey()

Returns the key of this node

Time complexity: O(1).

public HeapNode getParent()

Returns the parent of this node, nullable return value is possible.

Time complexity: O(1).

public int rank():

Returns the length of the children linked list of this node.

Time complexity: O(1)

public int getSize()

Returns the size of the children linked list of this node, plus one for the node itself.

Time complexity: O(1).

public boolean getMark()

Returns the mark of this node.

Time complexity: O(1)

public HeapNode findRecursive(int key):

recursively searches for the node with the given key in the tree rooted at this node,

including this node.

Prioritized direct search in children list over deep search to the depth of the tree.

Time complexity: O(n), since the tree is not a search tree.

public void setPrev(HeapNode node):

Sets the ‘prev’ reference to the given node.

Time complexity: O(1).

public void setNext(HeapNode node):

Sets the ‘next’ reference to the given node.

Time complexity: O(1).

public boolean setMark(boolean mark):

Set the mark of this node.

Time complexity: O(1)

public void insertPrev(HeapNode node):

Insert a node as the prev node of this node,

Update both node’s relevant references to each other.

Time complexity: O(1).

public void insertChild(HeapNode node)

Insert a node to the header of the children list of this node.

Time complexity: O(1).

public void plantPrev(LinkedList list):

Plant a linked list previous to this node.

Updates length, size, and relevant node pointers.

Time Complexity: O(1)

public void eject()

Remove this node from the doubly linked list:

set pointers to next, prev and 'siblings' to 'null'

Time Complexity: O(1)

public void cut()

Cut this node from

Time complexity: O(1)

1. **class FibonacciHeap.NodeFactory:**

Node factory which takes the key as an argument and also

sets a reference to the heap of the node.

1. **class LinkedList:**

class LinkedList implements Iterable<FibonacciHeap.HeapNode>:

LinkedList class that stores HeapNodes, also has a pointer to the minNode

and a length and size attributes that update according to the stored node's

size. a size is the total number of nodes stored in the current list, and all their offsprings.

public void deleteMin():

Helper method for heap.deleteMin()

Deletes the given node from the list, and plants the children in his place by order

Time complexity: O(1).

public FibonacciHeap.HeapNode listDecreaseKey(int key, int d, FibonacciHeap heap)

Helper method for heap.DecreaseKey() – based on Node.nodeDecreaseKey()

Time complexity: O(n)

public String toString()

[key\_root, key\_2, ..., key\_tail]

Time Complexity: O(n)

public boolean isEmpty():

Returns true iff root is null

Time complexity: O(1).

public boolean hasParent()

Returns true iff parent is not null.

O(1).

public FibonacciHeap.HeapNode getMin():

Return the minimal node from a designated reference ‘minNode’

Time complexity: O(1).

public FibonacciHeap.HeapNode findRecursive(int key):

recursively search for the node with the provided key within the children list of each node,

only if the requested key is not directly inside the children list.

Returns the node with the corresponding key.

Time complexity: O(n).

private void setSize(int size)

Set the size of this list

O(1).

public void increaseSize(int delta):

Call setSize() based on the delta parameter, which indicates the increase of the list’s size.

Recursively call tha parent’s siblings’ increaseSize() with the given delta parameter.

Time complexity: O(logn).

public void decreaseSize(int delta)

Call increaseSize with -delta

Time Complexiy: O(logn)

public void insertFirst(FibonacciHeap.HeapNode node)

Insert the parameter node as the header of this list,

set the parameter node's siblings list to this list,

update root attribute to the parameter node, and also the tail node if necessary.

updates length and size of this list

updates minNode pointer of this list, if necessary.

Time Complexity: O(logn)

public void annex(LinkedList list2)

Concatenate the param 'list2' to the tail of this list.

updates list pointers as relevant, including tail, root, and minNode

updates length and size of this list

Time Complexity: O(logn)

public void plantBefore(LinkedList list2, FibonacciHeap.HeapNode nodeAfter)

Param ‘list2’ is the list to be planted to this list

Param ‘nodeAfter’ is the node to be after the planted list, null calls annex(list2).

Time complexity: O(logn).

public void cutNode(FibonacciHeap.HeapNode node)

Removes the node with its children from the linked list.

updates length and size of list, also sends an update command upwards in the tree.

updates the minNode reference of this linked list, if necessary.

Time Complexity: O(logn)

public void ripNode(FibonacciHeap.HeapNode node)

Removes the node with its children from the linked list.

updates length and size of list, also sends an update command upwards in the tree.

Time Complexity: O(logn)

public void updateMin()

Go through all the nodes in the list to find the minimal keyed node,

And stores a reference to it in the list.

Time complexity: O(m), where m is the length of this list. (n is the size of the heap).

@Override

public Iterator<FibonacciHeap.HeapNode> iterator()

The node iterator of this linked list.

1. **class LinkedListFactory:**

A factory to create a linked list with a reference to its parent.

1. **class LinkedListIterator implements Iterator<FibonacciHeap.HeapNode>**

A node iterator that goes through the linked list.

**חלק ניסויי / תיאורטי**

**שאלה 1**

1. זמן הריצה האסימפטוטי של סדרת הפעולות במונחי m הוא: O(mlogm)

הסבר:

הפעולה בשורה הראשונה – סדרת ההכנסות, לוקחת O(m) מכיוון שכל הכנסה היא O(1) ומתבצעות בדיוק 1+m הכנסות

הפעולה בשורה השנייה – deleteMin לוקחת במקרה זה (שהוא גם המקרה הגרוע) O(m) מכיוון שמבצעים חיבורים לכולם באיגוד הערימה (כל הצמתים ביער מדרגה 0 מאז ההכנסה)

הפעולה בשורה השלישית – לולאה של decreaseKey, לוקחת O(mlogm).

לפי חוקיות בחירת הצמתים, הצומת שעליו מתבקש לעשות decreaseKey הוא הבן הימני, כלומר האחרון ברשימת הילדים (בניגוד לכיוון הריצה על העץ), ובנוסף לכך, סדר הפעולות מתבצע כך שמתחילים מהבן הימני במיקום הגבוה ביותר בעץ, ומשם ממשיכים לצמתים בתתי העצים הנמוכים יותר, כך שלא קורה מצב של cascading cuts, ומדובר במקרה בו נדרשת ריצה ארוכה על העץ עד למציאת הצומת.

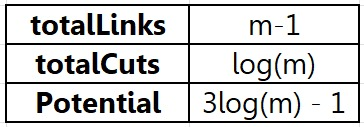
הפעולה של decreaseKey דואגת להפר את כלל הערימה, כך שקורים cutים לצמתים ללא ילדים ,וללא סימונים בצורה כזו שמובילה לחיתוך של צמתים רבים כתוצאה מפעולה יחידה.

הסיבוכיות של decreaseKey במקרה זה תהיה O(m), ומכיוון שהלולאה מבצעת logm איטרציות, הסיבוכיות הכוללת תהיה O(mlogm).

Table

Description automatically generated

1. הביטויים כתלות בm:



**TotalLinks** – לאחר רצף ההכנסות אנחנו מקבלים יער עם m + 1 עצים מדרגה 0.

לאחר המחיקה של הצומת המינימלי, אנחנו מבצעים קונסולידציה של העץ לכדי עץ בינומי אחד מושלם. ולפיכך, בהכרח שאנחנו מבצעים חיבור אחד לכל צומת שנותרה בעץ, חוץ מהשורש. מכאן, קיבלנו m-1 חיבורים.

**totalCuts** – החיתוכים מתבצעים על עץ בינומי מושלם, כאשר אנחנו חותכים את הבן עם דרגה 0 של השורש, ומטיילים לבן עם הדרגה המקסימלית של השורש, וחוזרים על אותו האלגוריתם באופן רקורסיבי, וזה ייתכן מכיוון שידוע לנו שגם הבן השמאלי ביותר של השורש הוא עץ בינומי מושלם.

כך יוצא שאנחנו מטיילים לעומק העץ, עד לעלה, וידוע לנו שהגובה של עץ בינומי מושלם הוא log(m).

**Potential** – הפוטנציאל שלנו הוא #roots + 2\*#marked.

ידוע לנו שביצענו log(m) חיתוכים, ועל כן לאחר רצף הפעולות תהיה לנו ערימה עם log(m) + 1 שורשים.

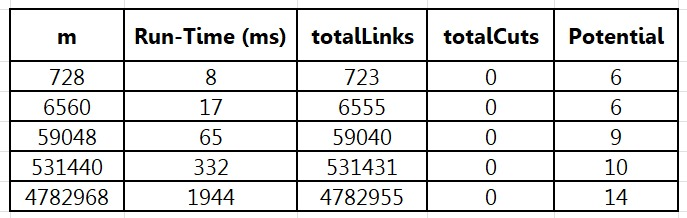
בנוסף, על כל חיתוך שבוצע, סימנו את האב בתור marked, למעט השורש של העץ המקורי, מכיוון שהוא שורש בערימה, וצמתים שהם שורשים אינם מסומנים לעולם. לכן, יש לנו log(m) – 1 צמתים מסומנים, ומתקבל potential = log(m) + 1 + 2 \* (log(m) – 1) = 3log(m) – 1.

1. ה. ו.

Table

Description automatically generated

**שאלה 2**



1. סיבוכיות הזמן הכוללת של סדרת הפעולות עבור m נתון היא . מכיוון שמדובר ברצף של פעולות הכנסה, וברצף של פעולות מחיקת מינימום, ניתן לבצע ניתוח amortized עבור כל אחת מהפעולות. ידוע לנו כי זמן הריצה של פעולת הinsert היא וכי זמן הריצה של פעולת deleteMin היא באמורטייזד. ולכן, עבור סדרת פעולות הכנסה באורך וסדרת פעולות מחיקת מינימום באורך , זמן הריצה הכולל יהיה .
2. פעולות link:

ניתן לחשב מפורשות את מספר פעולות החיבור שיתבצעו במהלך מחיקת איבר מינימלי אחד מתוך הסדרה, באופן הבא:

1. יש לנו m איברים בעץ. קיים מספר טבעי מקסימלי כך ש, כך ש הוא דרגת העץ הגדולה ביותר שיש לנו בערימה, ו- היא שארית מספר הצמתים שיש לנו בערימה, מלבד בעץ הראשי.
2. כאשר אנחנו מבצעים deleteMin, אנחנו מוחקים את השורש של העץ הגדול ביותר, שדרגתו היא .
3. כאשר אנחנו מבצעים את המחיקה, אנחנו "זורקים" את כל בניו של השורש לתוך הערימה, ומבצעים חיבורים עד לקבלת ערימת פיבונאצ'י חוקית.
4. נטען: מספר החיבורים שמתבצעים בעת התהליך היא הדרגה המקסימלית של בניו של הצומת שמחקנו, ומכיוון שמדובר היה בעץ בינומי מושלם, דרגה זו היא בהכרח , אולם תהליך החיבורים מתחיל רק מדרגת העץ המינימלית שקיימת בשארית, לפני האיחוד בין הערימה המקורית לבין רשימת בניו של הדרגה שנמחקה. סה"כ ניתן לראות מהי הדרגה המינימלית, נקרא לה , לפי המרה של מספר הצמתים בשארית למספר בינארי, ולקיחת החזקה הנמוכה ביותר של 2 במספר זה.
5. נחזור על החישוב עבור כל אחד מפעולות המחיקה, נסכום עד לקבלת מספר החיבורים הכולל שבוצעו מתוך התהליך.
6. ניתן לראות מהטבלה כי מספר החיבורים שבוצעו הוא מספר הצמתים שהוכנו לערימה מלכתחילה (), פחות הפוטנציאל (מספר העצים ביער בסוף התהליך):

פעולות cut:

נשים לב שכל פעולות המחיקה שאנחנו מבצעים הן על שורש, ולכן לא ייתכן כי אנחנו בכלל מבצעים פעולות cut בסדרת הפעולות שתוארה.

פוטנציאל:

מכיוון שידוע לנו כי מספר הcut הוא 0, הרי ברור שמספר הצמתים המסומנים הוא 0 גם,

ולכן, .

בנוסף, ידוע לנו שהערימה שלנו מוכרבת ממספר עצים בינומיים מושלמים בסוף התהליך (מכיוון שלא בוצעו אף cuts במהלך סדרת הפעולות), ולכן נתייחס לעצים כאל כאלה, וניווכח לדעת כי מספר העצים ברשימה הוא פשוט מספר הביטים בהמרה של מספר הצמתים ברשימה למספר בינארי: