username1 – pelegchen
id1 - 314953159
name1 - Peleg Chen
username2 - liorbelenkov
id2 - 319125134
name2 - Lior Belenkov

חלק תאורטי

שאלה 1

סעיף א':

פעולת הכנסה לערימת פיבונאצ'י במימוש הינה מסיבוכיות $O(1)$. לכן בהתחלה סכום כל ה-m+1 הכנסות עולות לנו $O(m)$

פעולת delete-min() היחידה במימוש הינה מסיבוכיות: $O(m)$ כ מספר הצמתים הנוכחי בערימה.

פעולת Decrease-key מערימת פיבונאצ'י במימוש הינה מסיבוכיות $O(1)$ ב -*amortized*. לכן סכום כל ה- $logm$ של Decrease-key עולות לנו $O(logm)$
סה"כ עלות הסדרה $O(logm) + 2O(m) = O(m)$

סעיף ב':

m	Run-Time (ms)	totalLinks	totalCuts	Potential
2^5	2	31	5	14
2^{10}	1	1054	15	29
2^{15}	13	33821	30	44
2^{20}	137	1082396	50	59

סעיף ג':

נראה כמה פעולות Link במדויק מתבצעות במהלך סדרת הפעולות כתלות ב-m:

את פעולת ה-Link אנו מבצעים רק כאשר אנו עושים delete-min בשלב ה-consolidate. מכיוון שאנחנו לא מבצעים עדיין פעולות Decrease-key אז נתייחס לעצים בערימה כעצים בינומיים לאורך הניתוח. בהינתן שמספר ההכנסות היה m+1, נספור כמה ביצענו לאחר המחיקה: נשים לב כי מכיוון שמדובר בכמות $m = 2^k$ ערימות בודדות מדרגה 0 אחרי המחיקה אז בעת consolidate אנחנו נאחד את הצמתים 2^k לעץ בינומי אחד.

עץ בינומי מדרגה k מצריך $T(k)$ כמות חיבורים, מתקיים

$T(k) = 2 * T(k-1) + 1$, נוכיח באינדוקציה $T(k) = 2^k - 1$:

בסיס: $T(0) = 0 = 2^0 - 1$

צעד: $T(k) = 2 * T(k-1) = 2 * (2^{k-1} - 1) + 1 = 2^k - 1$

סה"כ הכמות של ה-Link שאנו מבצעים היא $T(k) = 2^k - 1 = m - 1$

נראה כמה פעולות Cut במדויק מתבצעות במהלך סדרת הפעולות כתלות ב-m:

פעולות Cut נעשות רק במקרה של Decrease-key. אנחנו עושים logm פעולות Decrease-key ולכן logm כמות של פעולות Cut. הסדר של הפעולות: תחילה אנו מבצעים Decrease-key על הצומת עם המפתח 1, לאחר מכן על m/2 +1, 3m/4 +1, 7m/8 +1, וכך הלאה עד מפתח m-1 כולל. מכך מתקיים כי לכל מפתח שבו אנחנו עושים Decrease-key אנו לא חותכים מאב אחד 2 בנים (כל אב שונה) ומבצעים Cascading-Cuts. כלומר אנחנו נשארים עם logm כמות של פעולות Cut.

נראה מהו הפוטנציאל כתלות ב-m בסוף סדרת הפעולות:

נזכור כי הפוטנציאל מחושב ע"י: $Potential = \#trees + 2 * \#marked$.
כפי שצוין קודם אנו מבצעים logm פעולות Cut . עם כל פעולה אנחנו מוסיפים עץ לערימה (בערימה כפי שציינו יש עץ אחד) כי עשינו הפחתה של ערך המפתח למספר שלילי קטן מ- 1-. ולכן נוסף לנו logm+1 לפוטנציאל (כולל העץ המקורי).

מכיוון אנו מבצעים logm פעולות Cut לאבות שונים אז אנחנו גם נסמן את ה-logm-1 אבות, מכיוון שאחד האבות, הצומת שערכו *1-* (עשינו Decrease-key לצומת עם ערך מפתח 1) הוא בהכרח שורש לפי סדר ההכנסה ותהליך ה-consolidate.

סה"כ: $Potential = 2(logm - 1) + (logm + 1) = 3\ logm - 1$

סעיף ד':

נראה כמה פעולות Link במדויק מתבצעות במהלך סדרת הפעולות כתלות ב-m:

זהה לסעיף קודם סה"כ הכמות של ה-Link שאנו מבצעים היא $m-1$

<u>נראה כמה פעולות Cut במדויק מתבצעות במהלך סדרת הפעולות כתלות ב-m:</u>

במקרה זה סדר של הפעולות הוא: תחילה אנו מבצעים Decrease-key על הצומת עם המפתח 0, לאחר מכן על m/2, 3m/4 , 7m/8, וכך הלאה עד מפתח m-2 כולל. נשים לב כי ה- Decrease-key פועל על צמתים שהם צאצאים אחד של שני ("במסלול ישיר") ומתחיל בסדר מהצומת הכי עליון (האב של כולם) עם מפתח 0, לכן אנו לא נצטרך לעשות cut כי כלל הערימה משתמר אחרי כל פעולה. סה"כ 0.

<u>נראה מהו הפוטנציאל כתלות ב-m בסוף סדרת הפעולות:</u>

מתבצע תהליך consolidate ולכן אנחנו נותרים עם עץ אחד. כפי שתיארנו בחלק הקודם - עקב סדר הפעולות לא נצטרך לעשות cut על מנת ליצור עץ חדש – כלומר כמות העצים אינה מתווספת וכך גם כמות המסומנים נשארת 0.

סה"כ $Potential = 2*0+1 = 1$.

<u>סעיף ה':</u>

<u>נראה כמה פעולות Link במדויק מתבצעות במהלך סדרת הפעולות כתלות ב-m:</u>

את פעולת ה-Link אנו מבצעים רק כאשר אנו עושים delete-min ולכן בגרסא הזאת לא מתבצע פעולת Link. סה"כ הכמות של ה-Link שאנו מבצעים היא 0.

<u>נראה כמה פעולות Cut במדויק מתבצעות במהלך סדרת הפעולות כתלות ב-m:</u>

פעולות Cut נעשות רק בביצוע Decrease-key במקרה שהמפתח לא מקיים את כלל הערמה. מכיוון שמדובר בעצים שכולם דרגה 0 אז לא מתבצע פעולת Cut. סה"כ 0.

<u>נראה מהו הפוטנציאל כתלות ב-m בסוף סדרת הפעולות:</u>

לא מתבצע תהליך consolidate ולכן אנחנו נותרים עם m עצים מדרגה 0. מכיוון שהם שורשים אז אין מה לסמן בעת Decrease-key. סה"כ $Potential = m+1$.

<u>סעיף ו':</u>

<u>נראה כמה פעולות Link במדויק מתבצעות במהלך סדרת הפעולות כתלות ב-m:</u>

זהה לסעיפים ג,ד. סה"כ הכמות של ה-Link שאנו מבצעים היא $m-1$

<u>נראה כמה פעולות Cut במדויק מתבצעות במהלך סדרת הפעולות כתלות ב-m:</u>

בלי להתחשב בהוספה האחרונה - זהה לסעיף ג. סה"כ אנחנו מקבלים עם logm כמות של פעולות Cut. עכשיו נחשב התרומה של ההוספה – אנחנו בעצם מבצעים את הפעולה של Decrease-key על צומת-עלה של מסלול (ענף) שלם של אבות מסומנים (שפעלנו עליהם לפני) עד השורש (לא כולל) כלומר כמות $logm-1$ של

צמתים מסומנים, ולכן בעת הפעולה אנחנו נבצע תהליך Cascading-Cuts עד השורש לכן נוסיף עוד $logm - 1$ לסכום. סה"כ $2logm - 1$.

<u>נראה מהו הפוטנציאל כתלות ב-m בסוף סדרת הפעולות:</u>

עקב תהליך ה-Cascading-Cuts שתיארנו לפני, נוצרים לנו $logm - 1$ עצים חדשים שנוספים לכמות העצים שהזכרנו בסעיף ג ולכן

סה"כ $logm - 1 + logm + 1 = 2logm$ עצים. אחרי התהליך של כל Cascading-Cuts, כל האבות המסומנים הפכו לשורשים משמע אין מסומנים.

סה"כ $Potential = 2logm + 2 * 0 = 2logm$

<u>עלות הכי יקרה:</u>

עקב תהליך ה-Cascading-Cuts שתיארנו לפני, העלות הכי יקרה היא $logm - 1$, כאורך המסלול של מסומנים.

<u>טבלה מסכמת:</u>

case	totalLinks	totalCuts	Potential	decreaseKey max cost
ג	$m - 1$	$logm$	$3\ logm - 1$	
ד	$m - 1$	0	1	
ה	0	0	$m + 1$	
ו	$m - 1$	$2logm - 1$	$2logm$	$logm - 1$

שאלה 2

<u>סעיף א':</u>

m	Run-Time (ms)	totalLinks	totalCuts	Potential
728	3	723	0	6
6560	3	7278	0	6
59048	41	66318	0	9
531440	226	597749	0	10
4782968	2759	5380704	0	14

<u>סעיף ב':</u>

פעולת הכנסה לערימת פיבונאצ'י במימוש הינה מסיבוכיות: $O(1)$ ולכן סה"כ $O(m)$. בפעולת delete-min הראשונה היא בעלות $O(m)$ ככמות העצים.

אחרי הפעולה אנחנו נשארים עם כמות $O(logm)$ עצים ולכן שאר הפעולות delete-min (בכמות 3m/4 – 1) עולות לנו $O(logm)$ במקום (כמות העצים, פחות העץ שמימנו מחקנו + כמות התתי עצים שלו כשאר הינה $O(logm)$)

$$\sum_{k=0}^{m} O(1) + O(m) + \sum_{i=0}^{\frac{3m}{4}-1} O(\log(i)) = O(mlogm)$$

סעיף ג':

נראה כמה פעולות Link במדויק מתבצעות במהלך סדרת הפעולות כתלות ב-m:

את פעולת ה-Link אנו מבצעים כאשר אנו עושים delete-min בשלב ה-consolidate. מכיוון שאנחנו לא מבצעים פעולת Decrease-key אז נתייחס לעצים בערימה כעצים בינומיים לאורך הניתוח. בהינתן שמספר ההכנסות היה m+1, נספור כמה ביצענו לאחר המחיקה הראשונה:

עץ בינומי מדרגה k מצריך כמות חיבורים שנסמנה $\mathrm{T}(k)$,

הראנו בסעיף קודם $\mathrm{T}(k) = 2^k - 1$. על מנת לחשב מאילו דרגות של עצים הערימה מורכבת נשתמש בשיטה דומה לחישוב המספר בייצוג הבינארי של m:

$$\sum_{i=0}^{\lceil \log(m) \rceil} \left(\left\lfloor \frac{m}{2^i} \right\rfloor mod2 \right) * (2^i - 1)$$

נותרנו עם כמות עצים כמספר הביטים 1 בייצוג הבינארי של מספר m, בנוסף העצים שנותרנו איתם הם עצים בינומיים מדרגות שונות. נשים לב לתכונה הבאה, נסמנה *A* - לפי סדר ההכנסה של 0 עד *m* לערימה, אחרי המחיקה עם delete-min ואחרי תהליך ה-*consolidate* נקבל כי לכל 2 עצים בערימה מדרגות *i,j* כך ש-*i<j* המפתחות בעץ מדרגה *j* גדולים יותר מהמפתחות בעץ מדרגה *i*.

לכן נסמן ב-d את העץ בעל הדרגה הכי קטנה, נשים לב ששורשו הוא המפתח המינימלי (1).

עכשיו אחרי מחיקה נוספת, לפי תכונות של עצים בינומים אנחנו נמחק את שורש של העץ d ונוסיף את כל תתי העצים שלו לערימה שהינם מדרגה קטנה יותר מהעץ המקורי. כל העצים הם מדרגה שונה ולכן בתהליך ה-consolidate עבור מחיקה זאת אנחנו לא מבצעים Links.

העץ הבעל הדרגה הקטנה ביותר הינו שוב עם מפתח מינימלי כשורש. לפי תכונה זו ותכונה A שמשתמרת אז התהליך שתיארנו חוזר על עצמו בכל מחיקה ולכן גם במחיקות הבאות אנחנו לא מבצעים Links.

לכן סה"כ הפעולות Links שביצענו הם הכמות שביצענו לאחר המחיקה הראשונה:

$$\sum_{i=0}^{\lceil \log(m) \rceil} (2^i - 1) \left(\left\lfloor \frac{m}{2^i} \right\rfloor mod2 \right)$$

נראה כמה פעולות Cut במדויק מתבצעות במהלך סדרת הפעולות כתלות ב-m:

פעולות Cut נעשות רק במקרה של Decrease-key ולא נעשות בפעולות אילו כלל לכן מספר הפעולות הינו 0.

<u>נראה מהו הפוטנציאל כתלות ב-m בסוף סדרת הפעולות:</u>

נזכור כי הפוטנציאל מחושב ע"י: $Potential = \#trees + 2 * \#marked$.
נשים לב כי אנו לא מבצעים פעולות Cut לכן אף צומת לא תהיה מסומנת.
נחשב את מספר העצים ונסיים.

מספר העצים בערימה יהיה כמספר ה-"1"-ים בייצוג הבינארי של $\frac{m}{4} + 1$.

נסביר תחילה מדוע $\frac{m}{4} + 1$:

התחלנו עם $m + 1$ צמתים לאחר מכן מחקנו $\frac{3m}{4}$ לכן נותרו כ-$\frac{m}{4} + 1$ צמתים.
נסביר כעת למה מספר ה-"1"-ים בייצוג הבינארי:

לאחר המחיקה הראשונה נשארנו עם מספר עצים בהתאם לייצוג הבינארי של m, בכל מחיקה נשאר עם צומת אחת פחות על כן מספר העצים בערימה יהיה כמספר ביטי 1 בייצוג הבינארי של מספר הצמתים.

אזי: $$Potential = \sum_{i=0}^{\left\lceil \log(\frac{m}{4}+1) \right\rceil} \left(\left\lfloor \frac{\frac{m}{4}+1}{2^i} \right\rfloor mod2 \right)$$

תיעוד חיצוני

FibonacciHeap:

public HeapNode getFirst()

Return the First (most left and newest) node in the heap.

Complexity: O(1)

public int getSize()

Return the number of nodes in the heap

Complexity: O(1)

public int getNumTrees()

Return the number of trees in the heap.

Complexity: O(1)

public int getCountMarkNodes()

Return the number of mark nodes.

Complexity: O(1)

public boolean isEmpty()

Returns true if and only if the heap is empty.

Complexity: O(1)

private replaceMin(HeapNode node)

If argument node has minimal key, update min attr in Heap.

@pre: node is in heap, node.getParent() == null (node is root node).

Returns true if min got replaced, false otherwise.

public HeapNode insert(int key)

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.

Help functions: replaceMin
Complexity: O(1)

Returns the newly created node.

private void removeMinNode()

Only used for deleteMin. Remove the node with minimal key, add and connect its children as heap roots by nullifying parent connection and connecting them to the main root chain; reset their marked status.

@pre: there is more than 1 node in Heap.
Help functions: HeapNode.resetMarkedInChain, HeapNode.nulifyParentInChain, HeapNode.insertBefore
Complexity: O(m), m - number of subtrees of root with minimum key (m = O(logn))

private HeapNode consolidateConnect(HeapNode node1, HeapNode node2)

Only used for consolidate. Add the node with the bigger key as left-most child of node with the smaller key.

Complexity: O(1)

Returns the HeapNode with the smaller key

public void consolidate()

Consolidate trees by linking them with so that we will have O(log(n)) trees, each with a different rank, using the "buckets" method we saw in lecture. First, we create an array of buckets that each will hold a tree with a different rank. We go over every tree and insert it by it's rank to the appropriate bucket, if there is a tree in that bucket we connect them using consolidateConnect and move it to the next bucket. After consolidation

process is finished, we go over the bucket array, collect and connect the remaining trees while determining which root has the minimum value.

Help functions: HeapNode.resetMarkedInChain, HeapNode.nulifyParentInChain, HeapNode.insertBefore
Complexity: O(k-1+m), k - number of trees in heap (before deletion of min used prior) / m - number of subtrees of root with minimum key (m = O(logn))

<u>public void deleteMin()</u>

Deletes the node containing the minimum key.
First, we remove the minimum key with removeMinNode;
Then we consolidate and connect all trees using the method we saw in lecture with consolidate function, while determining which key has the minimum value.

Help functions: removeMinNode, consolidate
Complexity: O(k-1+m), k - number of trees in heap / m - number of subtrees of root with minimum key (m = O(logn))

<u>public HeapNode findMin()</u>

Returns the node of the heap whose key is minimal, or null if the heap is empty.

Complexity: O(1)

<u>public void meld (FibonacciHeap heap2)</u>

Melds heap2 with the current heap.

Help functions: FibonacciHeap.insertBefore, replaceMin
Complexity: O(1)

<u>public int size()</u>

Returns the number of elements in the heap.

Complexity: O(1)

public int[] countersRep()

Return an array of counters. The i-th entry contains the number of trees of order i in the heap.
(Note: The size of of the array depends on the maximum order of a tree.)

Help functions: findMaxRank()
Complexity: O(n)

public int findMaxRank()

Return the max rank of the all trees in the heap

Complexity: O(n)

public void delete(HeapNode x)

Deletes the node x from the heap.
It is assumed that x indeed belongs to the heap.

Complexity: O(n)

public void decreaseKey(HeapNode x, int delta)

Decreases the key of the node x by a non-negative value delta. The structure of the heap should be updated to reflect this change (for example, the cascading cuts procedure should be applied if needed).

Help functions: cascadingCut()

Complexity: O(log(n))

public int nonMarked()

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

Complexity: O(1)

public int potential()

This function returns the current potential of the heap, which is:
Potential = #trees + 2#marked

In words: The potential equals to the number of trees in the heap
plus twice the number of marked nodes in the heap.

Complexity: O(1)

public static int totalLinks()

This static function returns the total number of link operations made during the
run-time of the program. A link operation is the operation which gets as input two
trees of the same rank, and generates a tree of rank bigger by one, by hanging the
tree which has larger value in its root under the other tree.

Complexity: O(log(n))

public static int totalCuts()

This static function returns the total number of cut operations made during the
run-time of the program. A cut operation is the operation which disconnects a subtree
from its parent (during decreaseKey/delete methods).

Complexity: O(1)

public void cascadingCut(HeapNode x,HeapNode xParent)

This is recursive function!

This function continues to make cascading cuts as long as the parent tree is marked.

Help functions: cut()

public void cut(HeapNode x,HeapNode xParent)

Cuts node x from xParent , xParent its x's parent and adds it as a new tree.

Help functions: replaceMin()

Complexity: O(1)

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

This static function returns the k smallest elements in a Fibonacci heap that contains a single tree.
The function should run in O(kdeg(H)). (deg(H) is the degree of the only tree in H.)

Help functions: isEmpty(),deleteMin(),findMin(),insert(),

Complexity: O(kdeg(H))

HeapNode

public HeapNode(int key)

Initializing heap node

Complexity: O(1)

public int getKey()

Return key of the node

Complexity: O(1)

public void setKey(int k)

Set k as the node's key

Complexity: O(1)

public int getRank()

Return the rank of the node

Complexity: O(1)

public void setRank(int k)

Set k as the node's rank

Complexity: O(1)

public boolean getMarked()

Return true if the current node is marked else return false.

Complexity: O(1)

public void setMarked(boolean TF)

Set true if the node is marked else set false.

Complexity: O(1)

public HeapNode getChild()

Return the child of the node.

Complexity: O(1)

public void setChild(HeapNode node)

Set node as the child of the current node.

Complexity: O(1)

public HeapNode getParent()

Return the parent of the node.

Complexity: O(1)

public void setParent(HeapNode node)

Set node as the parent of the current node.

Complexity: O(1)

public HeapNode getNext()

Return the next node of the current.

Complexity: O(1)

public HeapNode getFirst()

Set node as the next node of the current.

Complexity: O(1)

public HeapNode getPrev()

Return the previous node of the current.

Complexity: O(1)

public void setPrev(HeapNode node)

Set node as the next node of the current.

Complexity: O(1)

public HeapNode getKMinPointer()

Return the KMinPointer. We use this in the function kMin(FibonacciHeap H, int k).

Complexity: O(1)

public void setKMinPointer(HeapNode node)

Set node as the KMinPointer of the current.

Complexity: O(1)

private void insertBefore(HeapNode node)

Add node x as a left sibling to instance node.

Complexity: O(1)

private void insertBefore(HeapNode node1, HeapNode node2)

Add a chain of nodes and connect them to instance node. The last node in the chain (node2) will be the left sibling of instance node.

Complexity: O(1)

private int nulifyParentInChain()

Go over instance node and all it's sibling and change attribute 'parent' to null.

Returns the number of nodes in chain (k+1).

Complexity: O(k+1), k - number of siblings instance node has.

private int ResetMarkedInChain()

Go over instance node and all it's sibling and change attribute 'mark' to 'false'.

Returns the number of nodes in chain in which 'mark' was set to 'true'.

Complexity: O(k+1), k - number of siblings instance node has.