בס"ד

מטלה מספר 3

קורס: מבני נתונים

שם: כפיר גולדפרב

ת.ז: 208980359

אימייל: kfir.goldfarb@msmail.ariel.ac.il

פתרון שאלה 1:

סעיף א': $\frac{numOnly}{n} \le \frac{1}{2}$

ראשית נשנה את האי-שיוויון בצורה הבאה:

$$2 \cdot numOnly \le n$$

מכיוון $n > 0$ אז אי-השיוון אינו מתהפך (לא יכול להיות 0 כי לא הגיוני $\frac{numOnly}{n}$ כאשר $n = 0$).

ולכן מספיק להוכיח שלכל $n > 0$ מתקיים $2 \cdot numOnly \le n$:

עבור $n = 1$, לעץ קיים רק השורש, מכיוון שהשורש אינו נחשב בן יחיד, כי אין לו אב אזי $numOnly = 0$ ולכן מתקיים:

$$0 \le 1$$

עבור $n = 2$, לעץ קיים רק שורש ובן אחד ללא שכן ולכן $numOnly = 1$, ולכן מתקיים:

$$\frac{1}{2} \le \frac{1}{2}$$

עבור $n = 3$, מכיוון שעץ AVL הוא עץ מאוזן, אחרי איזון לשורש קיימים שני בנים שכנים זה לזה ולכן $numOnly = 0$, ולכן מתקיים:

$$\frac{0}{3} = 0 \le \frac{1}{2}$$

כלומר ניתן לראות שכאשר n זוג אז $numOnly = 1$, ההסבר לכך הוא שקיים רק עלה יחיד ללא שכנים בעץ AVL מאוזן, וכשאר $n + 1$ כלומר נוסיף לו קודקוד נקבל מספר אי-זוגי של קודקודים אבל אחרי איזון לא יהיה עלה יחיד ללא שכן ולכן כאשר n אי זוגי אז $numOnly = 0$, ראינו שהטענה מתקיימת עבור $n = 1,2,3$ ומכיוון ש-$numOnly$ תמיד יהיה שווה ל-0 או ל-1 ניתן לראות כי $\frac{0}{n}$ תמיד יהיה קטן מחצי לכל $n > 0$, ו-$\frac{1}{n}$ תמיד יהיה קטן מחצי לכל $n > 1$, ולכן האי שיוויון תמיד מתקיים.

■

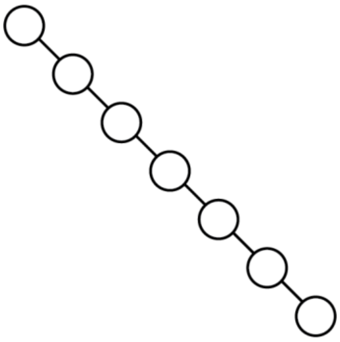
סעיף ב':

הטענה נכונה, הוכחה: ראינו לפי הוכחת סעיף א' שהטענה $\frac{numOnly}{n} \le \frac{1}{2}$ תמיד מקיימת לכל $n > 0$ בעץ AVL, באותה מידה ניתן להגיד כי אם הטענה מתקיימת קל וחומר כאשר העץ הוא עץ AVL או לפחות מקיים את תכונת האיזון, ידוע שבעץ AVL הגובה יהיה לכל היותר $\log_2 n$ כאשר n הוא מספר הקודקודים בעץ, ומכאן ניתן לומר שמתקיים $height(T) = O(\log_2 n)$, כלומר הגובה של עץ T שהוא עץ מאוזן יהיה לכל היותר (חסם עליון)$\log_2 n$, וכאשר T עץ מאוזן הטענה תמיד מתקיימת.

■

סעיף ג':

הטענה אינה נכונה, נראה דוגמה נגדית, יהי עץ בינארי T אשר בנוי בצורה הבאה:



ניתן לראות כי בעץ הנ"ל יש n קודקודים (מקרה פרטי זה יש 7), אשר כולם חוץ מהשורש הם בן יחיד, העץ מקיים את תכונה הבינארי שלכל אב יש לכל היותר 2 בנים, אך ניתן לראות שעץ זה הוא לא מאוזן, אלה בינארי והגובה שלו הוא $n-1$, כלומר $height(T) = O(n+1) = O(n) > O(\log_2 n)$ ולכן זו סתירה לטענה.

פתרון שאלה 2:

פונקצית עזר:

```
/**
 * function that return linked list of nodes with the leaves of a tree
 * in every search for a leave the function go throw all the nodes in the path to get to the leave
 * some searches probably will go throw a nodes that visited already
 * in every search for leave the worst case is that the function can throw all node int the tree
 * so the complexity is O(n) in every search for leave
 * @param tree - searching tree leaves
 * @return leaves - linked list of tree leaves
 */
public static LinkedList<Node> getLeaves(BinaryTree tree) {
    LinkedList<Node> paths = new LinkedList<Node>();
    return getLeaves(tree.root, paths);
}

/**
 * recursively function for getLeaves function
 * @param node - starting from the root of the tree and get recursively down to leaves
 * @param leaves - the leaves list that will return
 * @return leaves
 */
// get leaves recursively
private static LinkedList<Node> getLeaves(Node node, LinkedList<Node> leaves) {
    if(node == null) return leaves;
    // if found a leave add to list
    if(node.left == null && node.right == null) leaves.add(node);
    // going throw all left and right nodes
    else {
        getLeaves(node.left, leaves);
        getLeaves(node.right, leaves);
    }
    // return the tree leaves list
    return leaves;
}
```

במחלקת *Node* הוספתי שדה הנקרא *parent* שבו כל צומד שומר על כתובת האב שלו, אם אין לו אב אז ה-parent שלו שווה ל-null כלומר רק לשורש אין אב.

בנוסף במחלקת Node השדה *data* הוא מסוג *Integer*.

כל השדות במחלקת Node עשיתי public.

סעיף א':

```
/**
 * function that return the max sum of a path from the root to leave,
 * in bad case the complexity of every time the function work recursively is O(height(tree)),
 * because the function going throw all paths of the array in every call,
 * @param tree – the tree that the function work on,
 * @return max – the max sum of a path from root to leave.
 */
public static Integer maxSum(BinaryTree tree) {
    // if tree is empty return sum 0
    if(tree.root == null) return 0;

    // number of leaves is equal to number of sums,
    // that why i used numOfLeaves() function to calculate number of leaves = number of paths
    // make an array for all path sum
    int[] emptySums = new int[tree.numOfLeaves()];

    // getting the sums recursively in to the array
    int[] sums = maxSum(tree.root, emptySums, 0, 0);

    // getting and return the max sum path from the array
    int max = sums[0];
    for(int i = 1; i < sums.length; i++) { // O(number of paths)
        if(max < sums[i]) max = sums[i];
    }

    // return max
    return max;
}

/**
 * the recursive function of maxSum function that work recursively on tree paths,
 * and calculate all sums of the paths of the tree and insert the sums to sums array and return the array,
 * @param node – recursively the node that we get his data and add it to the sum of it's path,
 *             – changing by node.left and node.right from the tree root until get null node,
 * @param sums – the array of the sums of the tree paths, recursively adding sum after going throw path,
 * @param i    – the index of sums array,
 * @param sum  – the sum of the path,
 * @return sums – the array with all sums of paths in the tree.
 */
private static int[] maxSum(Node node, int[] sums, int i, int sum) {
    // return sums array when get to nil
    if(node == null) return sums;

    // getting sum of all node data in the path
    sum = sum + node.data;

    // if get to leave insert sum of path to the array
    if(node.left == null && node.right == null) {
        sums[i] = sum;
        sum = 0;
        i++;
    } else {
        // recursively calculating the sums of all paths
        maxSum(node.left, sums, i, sum);
        maxSum(node.right, sums, i, sum);
```

```
  }

  // return paths sum
  return sums;
}
```

סעיף ב':

```
/**
 * function that return string of nodes of the max sum path from leave to root in the tree,
 * the function use the getLeave function to get all tree leaves in the tree,
 * and than use the getPath function to get a path from leave to root,
 * and than use the getSum function to get a sum of a path
 * and than with an index value, calculate who had the max sum in tree paths
 * create a string looking like: "1 -> 2 -> 3 -> 4 -> 5 -> null",
 * and return the string
 * all the complexity of the the other function are detailed below
 * @param tree - the tree that the function work on
 * @return maxPathSumOutput - the string of the max sum path in the tree
 */
public static String maxSumPath(BinaryTree tree) {

  // getting tree leaves
  LinkedList<Node> leaves = getLeaves(tree);

  // calculate the max sum path
  int max = 0, maxIndex = 0;
  for(int i = 0; i < leaves.size(); i++) {
    if(max < getSum(getPath(leaves.get(i), tree))) {
      max = getSum(getPath(leaves.get(i), tree));
      maxIndex = i;
    }
  }

  // create a string maxPathSumOutput of the max sum path from root to leave
  LinkedList<Integer> maxPathSum = getPath(leaves.get(maxIndex), tree);
  String maxPathSumOutput = "";
  for(int i = maxPathSum.size() - 1; i >= 0 ; i--) {
    maxPathSumOutput += maxPathSum.get(i) + " -> ";
  }
  maxPathSumOutput += "null";

  // return the string
  return maxPathSumOutput;
}

/**
 * function that return the sum of a linked list of integers to get paths sum
 * going throw all the nodes in the path from leave to the root
 * so the max complexity is O(height(tree))
 * @param path - a linked list of path nodes data
 * @return sum - the sum of the nodes data in the path
 */
private static int getSum(LinkedList<Integer> path) {
  int sum = 0;
  for(int i = 0; i < path.size(); i++) sum += path.get(i);
  return sum;
}
```

```
/**
 * function that can get the path form leave to root in the tree
 * the function go from leave node to the root node
 * so the complexity of the the function in every call is O(height(tree))
 * @param node - the leave that the function work on to get it's path to root
 * @param tree - the tree
 * @return path - linked list with node path to the root
 */
public static LinkedList<Integer> getPath(Node node, BinaryTree tree) {
  LinkedList<Integer> path = new LinkedList<Integer>();
  while(node != tree.root) {
    // adding to path list all node in the path from leave to root
    path.add(node.data);
    node = node.parent;
  }
  // adding the root
  path.add(tree.root.data);
  return path;
}
```

פתרון שאלה 3:

סעיף א':
מכיוון שעץ ערימה הוא עץ בינארי כמעט שלם, מספר מקסימלי של קודקודים הוא $2^{h+1}-1$ כאשר h הוא הגובה של העץ, ומספר מינימאלי של קודקודים הוא 2^h.

סעיף ב':
מכיוון שהעץ הוא מסוג $min-heap$ כל איבר קטן מצומת האב שלו, ולכן תמיד המספרים הכי גדולי בעץ יהיו העלים, ועל מנת למצוא את המספר המקסימלי ב-$min-heap$ יש להשוות בין כל ערכי העלים של העץ.

סעיף ג':
מכיוון שעץ ערמה הוא עץ בינארי כמעט שלם לפי ההגדרה הגובה של העץ הוא תמיד יהיה $\log_2 n$ כאשר n הוא מספר הקודקודים בעץ.

סעיף ד':
החלטתי לפתור את הבעיה בצורה הבאה: ראשית למזג בין שני מערכי הערימות (כי שניהם ממומשות ע"י מערכים), ולאחר מכן להשתמש באלגוריתם מיון ערימה, ובכך קיבלנו ערימה חדשה המורכבת משני הערימות בצורה ממוינת, להלן פונקצית העזר אשר ממיינת ערימה:

```
// the min heap sort algorithm
public static void heapSort(MinHeap heap){
  heap.buildMinHeap();
  int heapSize = heap.size;
  for (int i = heapSize - 1; i >= 1; i--) {
    heap.swap(0, i);
    heapSize = heapSize - 1;
    heap.minHeapify(0, heapSize);
  }
}
```

*(כל שאר הפונקציות הנ"ל כגון $buildMinHeap$, $swap$, $minHeapify$, כבר ממומשות בעץ ערימה $MinHeap$ אשר למדנו בהרצאה).

הפונקציה $mergeTwoHeaps$:

```
public static MinHeap mergeTwoHeaps(MinHeap h1, MinHeap h2) {
  int[] newHeapArray = mergeArrays(h1.getA(), h2.getA());
  MinHeap newHeap = new MinHeap(newHeapArray);
  MinHeap.heapSort(newHeap);
  return newHeap;
}
```

כאשר השתמשתי בשני פונקציות עזר $getA$ אשר מחזירה את המערך של הערימה וממומשת בתוך המימוש של עץ ערימה בצורה הבאה:

```
public int[] getA() { return a; }
```

כאשר a הוא שדה של מערך מסוג $int[]$ שבו שמורים כל ערכי הערימה, ובנוסף ניתן לראות שהשתמשתי בפונקציה העזר $mergeArrays$ אשר מקבלת שני מערכים, ממזגת אותם למערך חדש ומחזירה אותו, להלן רוד הפונקציה:

```
// merging two arrays into on array
private static int[] mergeArrays(int[] a, int[] b) {
  int[] c = new int[a.length + b.length];
  for(int i = 0; i < a.length; i++) {
    c[i] = a[i];
  }
  int j = a.length;
  for(; j < c.length; j++) {
    c[j] = b[j];
  }
```

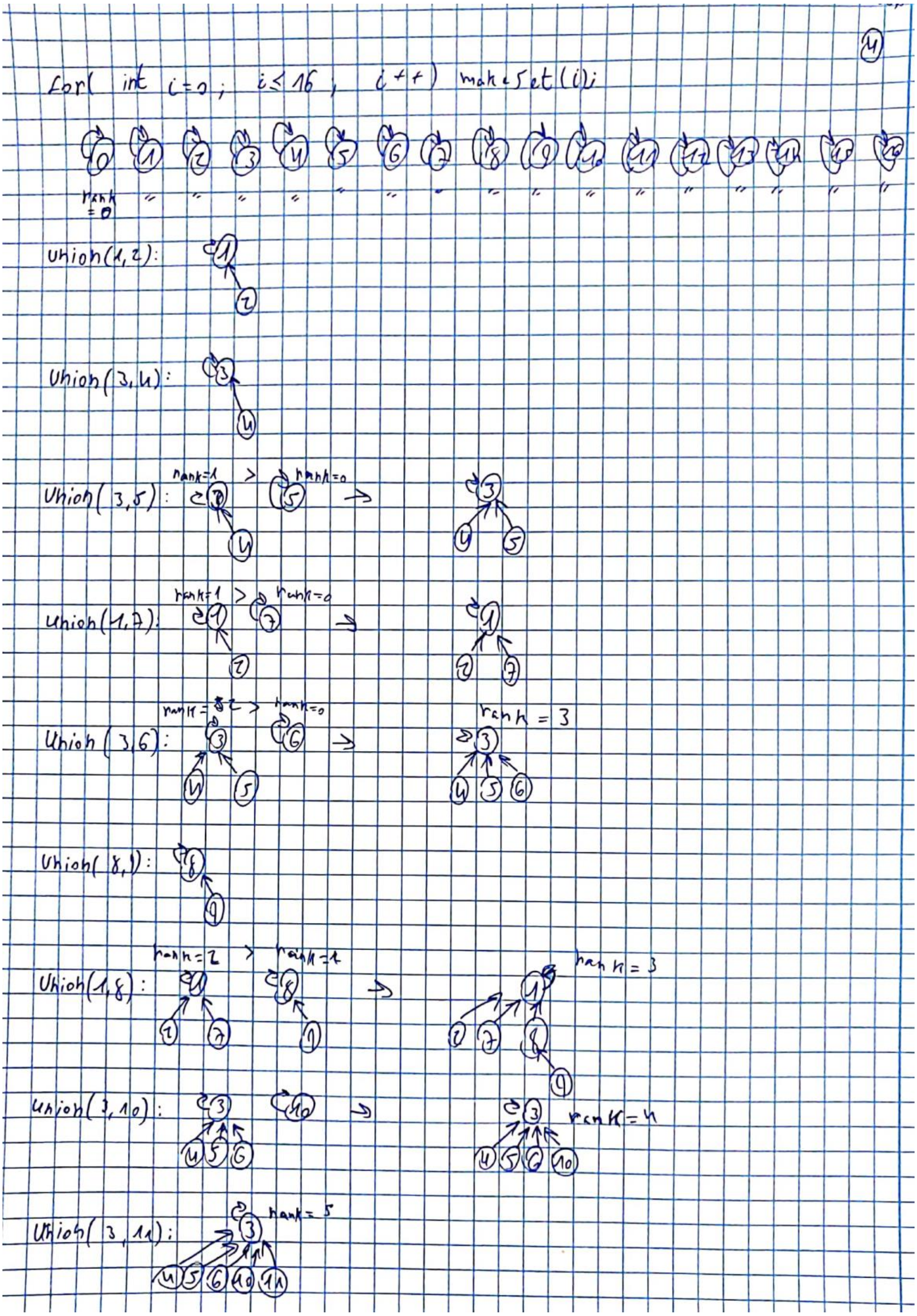
```
  return c;
}
```

סיבוכיות הפונקציה $\boldsymbol{mergeTwoHeaps}$:

ראשית הפונקציה $mergeArrays$ ממזגת שני מערכים בסיבוכיות של $O(a.length + b.length)$ נסמן את $c.length =$ $a.length + b.length = n$ ולכן הסיבוכיות של המיזוג מערכים היא $O(n)$, לאחר מכן נציצור עץ ערימה חדש בעזרת $constructor$ שהסיבוכיות היא גם כן $O(n)$ כי הבנאי לוקח את כל אברי המערך שהוא קיבל ושם את ערכיו מחדש במערך של העץ, לאחר מכן הפונקציה $heapSort$ ממיינת את העץ החדש כשאר הסיבוכיות של הפונקציה $buildMinHeap$ היא $O(n)$ וסיבוכיות הפונקציה $swap$ היא $O(1)$ וסיבוכיות הפונקציה $minHeapify$ היא $O(\log n)$, ולכן הסיבוכיות של המיון עצמו היא $O(nlog\ n)$, וסכך הכל קיבלנו:

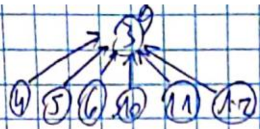
$$O(n) + O(n) + O(n) + O(1) + O(nlog\ n) = \boldsymbol{O(nlog\ n)}$$

פתרון שאלה 4:


4
for(int i=0; i≤16, i++) makeSet(i);
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
rank = 0
Union(1,2):
Union(3,4):
Union(3,5):
rank=1 > rank=0
Union(1,7):
rank=1 > rank=0
Union(3,6):
rank=2 > rank=0
rank = 3
Union(8,9):
Union(1,8):
rank=2 > rank=1
rank = 3
Union(3,10):
rank=4
Union(3,11):
rank=5

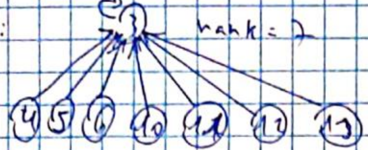
Union(3,12):

rank = 6



Union(3,13):

rank = 7

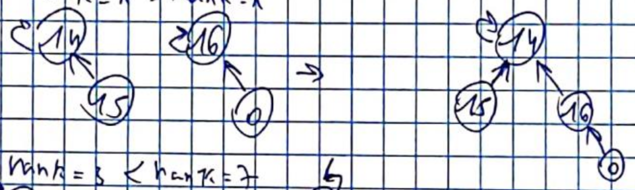


Union(14,15):

Union(16,0):

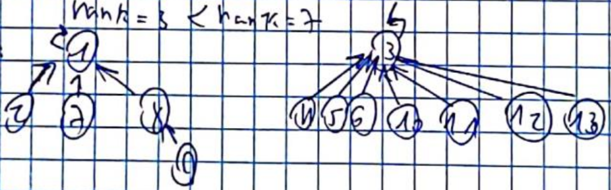
rank = 1 = rank = 1

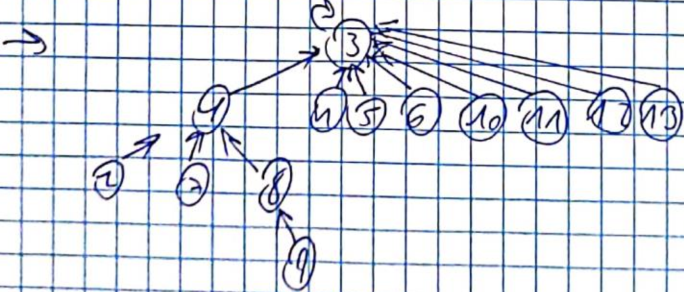
Union(14,16):



rank = 3 < rank = 7

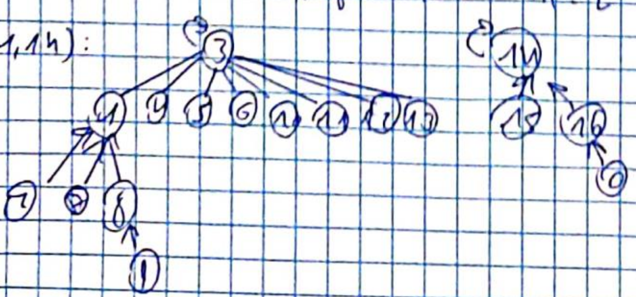
Union(1,3):



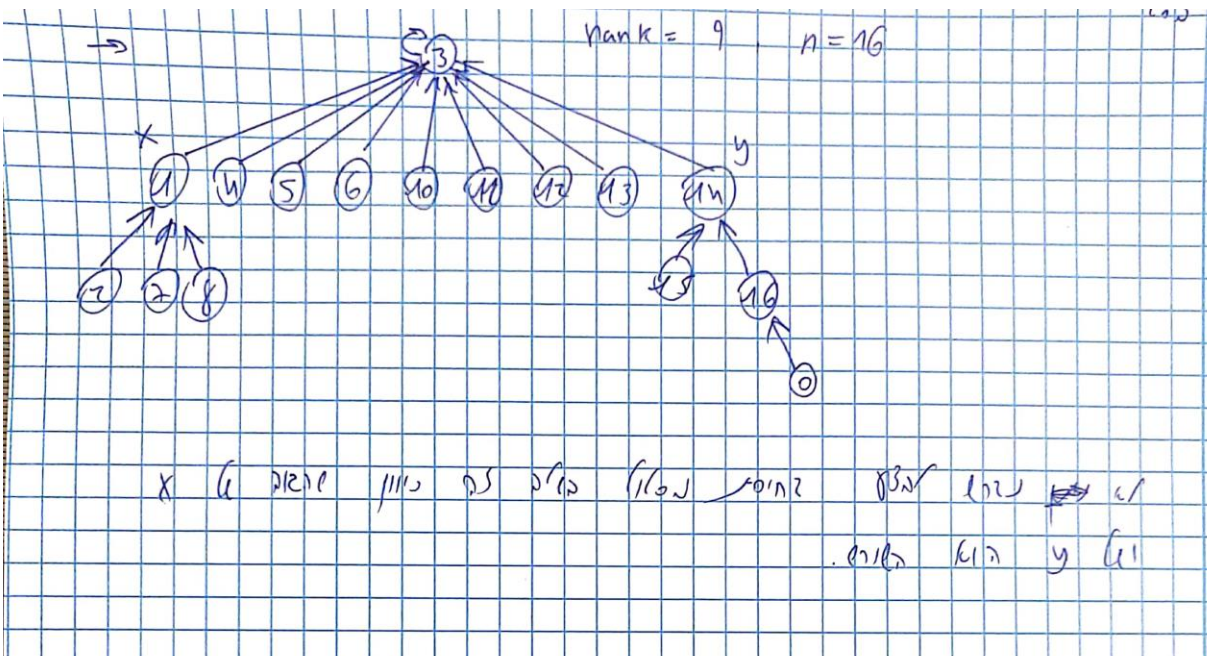


rank = 8 > rank = 2

Union(1,14):



תשובה לסעיף בעמוד הבא:



סיבוכיות שאלה 4:

הפונקציה $makeSet$ במקרה הזה מייצרת 16 קבוצות נפרדות (עצים בעלי שורשים יחידים המצביעים על עצמם), סיבוכיות הפונקציה היא $O(1)$, ולכן נחשב: $16 \cdot O(1) = \boldsymbol{O(1)}$.

כל פונקציית איחוד $union$ מבצעת פעולות יחידות כמו להגדיר $parent$ לשורש של עץ אחר, מכיוון שהפעולות מסוג זה הן בסיבוכיות $O(1)$ אז הפונקציה $union$ היא בסיבוכיות $O(1)$, מכיוון שבכל התהליך הנ"ל בשאלה לא היה צורך בשימוש בדחיסת מסלול, הסיבוכיות עדיין $O(1)$, הפונקציה $union$ מתבצעת 16 ולכן נחשב $16 \cdot O(1) = \boldsymbol{O(1)}$.

סה"כ:

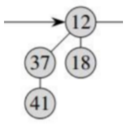
$$16 \cdot O(1) + 16 \cdot O(1) = \boldsymbol{O(1)}.$$

פתרון שאלה 5:

סעיף א':

- <u>כאשר x הוא שורש העץ:</u>
העץ הבינומי היחידי בערימה בינומית שאח השורש שלו הוא $null$ הוא העץ הכי ימני בערימה, ולכן לפי התנאי $x.silbing \neq null$, נתיחס לכל העצים האחרים (בלי העץ האחרון), מכיוון שערימה בינומית תמיד ממוינת לפי דרגות שורש העצים הבינומים מצד שמאל לימין (כלומר שתמיד עץ בינומי בערימה בינומית, דרגת השורש שלו יותר גדולה מדרגת השורש של העץ השמאלי לו), ולכן לפי תכונה זו תמיד $x.degree$ יהיה קטן מ-$x.silbing.degree$.

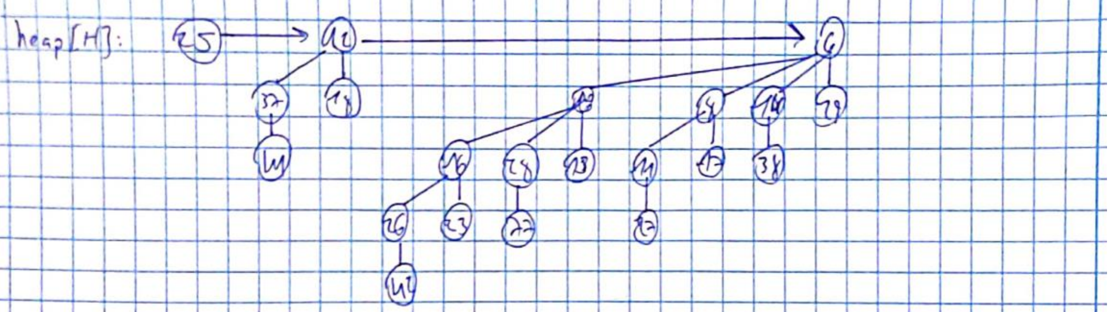
- <u>כאשר x הוא לא שורש העץ:</u>
הקודקודים אשר מקיימים $x.silbing \neq null$ הם כל הקודקודים חוץ מהקודקודים הכי ימניים, בעץ בינומי תמיד הקודקודים של רמות ההמשך נמצאים מצד שמאל, כלומר אם בעץ בינומי יש כמה רמות, אז הבנים של הרמה המסויימת הם בנים של הקודקודים השמאליים, להמחשה:



כלומר כאן העלה 41 הוא בן של 37 שהוא הבן השמאלי של השורש, וניתן לשים לב שה-$silbing$ שלו הוא לא $null$, ומכיוון שעצים בינומים בנויים בצורה רקורסיבית, תכונה זו תמיד תתקיים ולכן $x.degree$ יהיה תמיד גדול או שווה ל-$x.silbing.degree$.

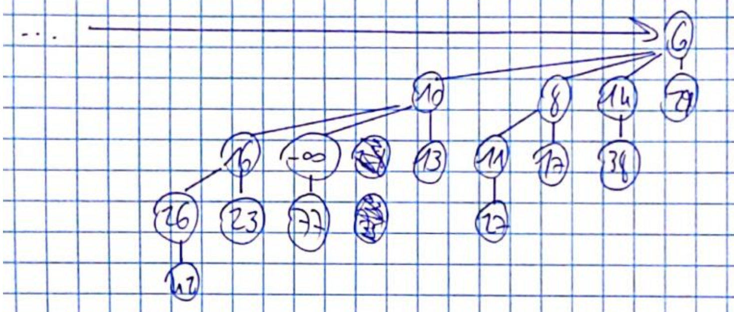
סעיף ב:

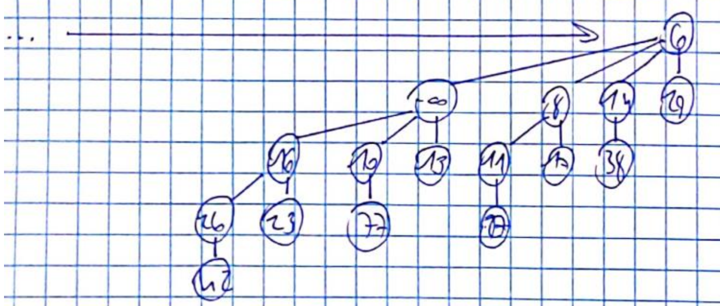
5 ב:

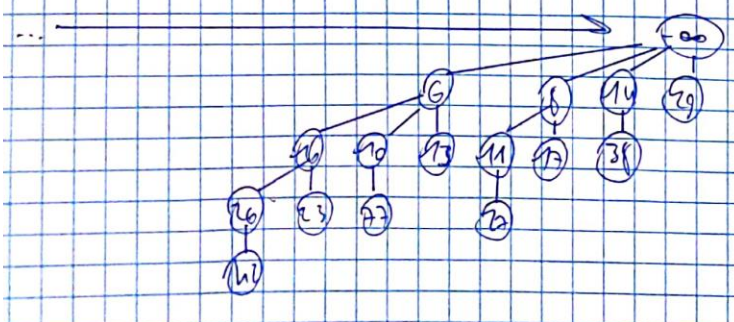


נהפוך את 28 לערך -∞, ונבצע decreasekey

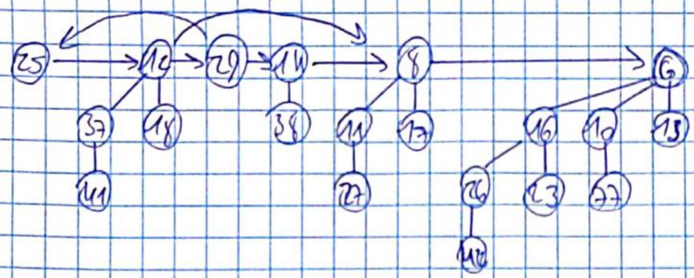
נמשיך עד לשורש:

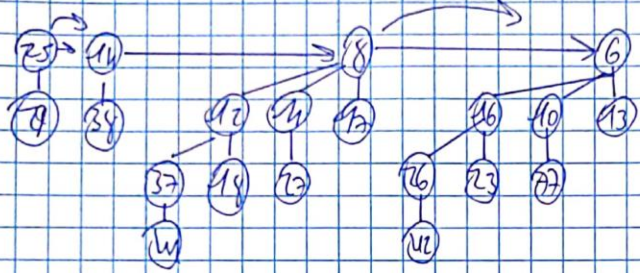


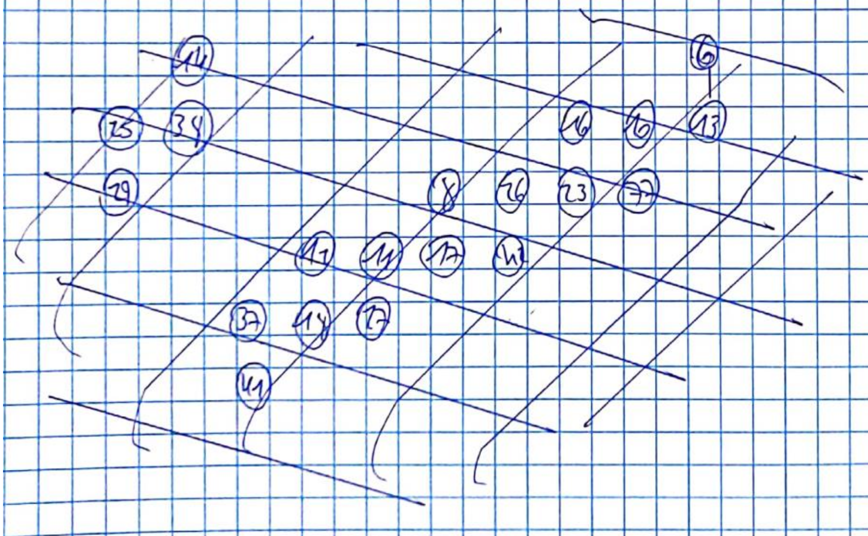




עכשיו נבצע מחיקה ל- ∞ - (כלומר 48), נהפוך את הבנים
לסדר עולה הפוך ונבצע מיזוג בינום לפי סדר המחרוזת בשורה:

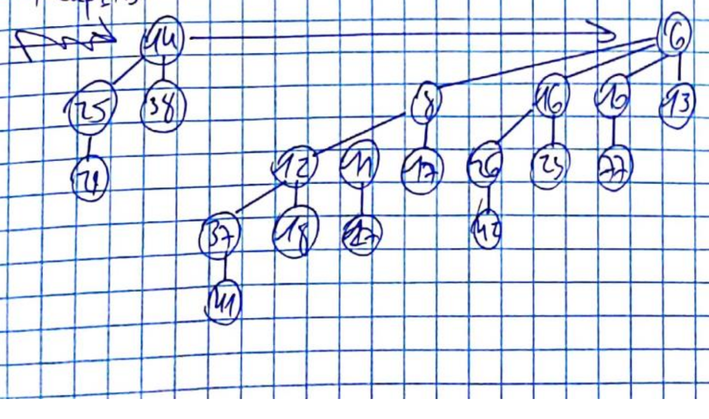






תוצאה סופית:

Heap[14]:



סיבוכיות שאלה 5 סעיף ב':

סיבוכיות של מחיקת איבר בערימה בינומית הוא $O(\log_2 n)$, בפירוט:

תחילה אנחנו מבצעים את פונקצית $decreaseKey$ אשר מסדרת את תכונה ה-$minHeap$ עם הקודקוד ששינינו (ל-$-\infty$), ביצוע פעולה זו היא בסיבוכיות של $O(\log_2 n)$,

לאחר מכן אנחנו מבצעים את פונקצית $extractMin$, שבעצם מוחקת את הקודקוד ששינינו ($-\infty$), הפונקציה של המחיקה כבר מבצעת סידור של כל הערימה כולל מיזוג בין העצים, סיבוכיות הפונקציה היא $O(\log_2 n)$,

וסה"כ קיבלנו:

$$O(\log_2 n) + O(\log_2 n) = \boldsymbol{O(\log_2 n)}$$