מבני נתונים – תרגיל 2 (מעשי)

פעולות על תתי מחרוזת באמצעות עץ סיפות

תאריך פרסום: 4.4.21

תאריך הגשה: 25.4.21

מתרגלת אחראית: ענבל רושנסקי

הנחיות:

- הנחיות מפורטות לגבי הגשת העבודות ינתנו בהמשך באתר הקורס.
- שאלות לגבי העבודה יש לשאול בפורום באתר הקורס או בשעות קבלה של ענבל רושנסקי.
- תיאור המחלקות הנתונות והדרושות למימוש מופיע בסוף המסמך.
- ***נא לקרוא את כל המסמך טרם תחילת העבודה – מומלץ יותר מפעם אחת. חלק מהאלגוריתמים המוצגים כבר סופקו לכם ואין צורך לממשם. מומלץ בנוסף לקרוא מול קבצי הקוד ששופקו לכם לעוד מידע אשר כתוב בהערות הקוד.***

נושאי העבודה: רשימה מקושרת, Compressed Suffix Trie.

מבוא

עבודה זו עוסקת בבעיה מאוד אפליקטיבית בתחומים רבים - "בעיית תת-המחרוזת". כלומר, בהינתן שתי מחרוזות a ו- b, אנחנו מעוניינים לדעת האם המחרוזת a מכילה את המחרוזת b. בתחום הביולוגיה והביו-אינפורמטיקה יש לבעיה מספר שימושים. למשל, בהינתן גנום של אורגניזם חדש שרק נתגלה ורוצף, ואוסף של גנים מאורגניזמים ידועים, יש צורך לבדוק האם הם מוכלים בגנום של האורגניזם החדש. הגנום והגנים הם מקטעי DNA אשר מיוצגים כמחרוזות מא"ב {A,C,T,G}.

פתרון אפשרי, אבל מאוד לא פרקטי מבחינה חישובית הוא לייצג את הגנום וכל אחד מהגנים באמצעות אובייקט מסוג String בשפת Java. במחלקה String קיימת המתודה contains שעונה על צרכים אלו. עבור 2 מחרוזות a,b הקריאה a.contains(b) תחזיר true אם ורק אם b מוכלת ב-a (בפייתון ניתן לכתוב b in a).

המימוש הנאיבי של contains סורק את כל המחרוזת a ומחפש את המחרוזת b, לכן זמן הריצה של הפונקציה הוא $O(|a|)$. אם a מחרוזת מאוד גדולה (כגודל הגנום) ואנו מתכוונים לבדוק האם המון מחרוזות קטנות b (כגון גנים) מוכלות ב-a, זמן הריצה יהיה עצום ולא ריאלי. היינו מעדיפים לבצע חישוב מקדים וחד פעמי על a, כך שלאחר מכן נוכל לבדוק לכל מחרוזת b האם היא מוכלת ב-a בזמן ריצה משמעותית קטן יותר.

בתרגיל זה תממשו מבנה נתונים, שנקרא עץ סיפות (suffix tree), לייצוג מחרוזת a, שיודע לענות על שאילתות הכלה כגון contains(b) ו-numOfOccurrences(b) (מס׳ המופעים של b במחרוזת) ב-$O(|b|)$, שכאמור הוא משמעותית קטן מ-a. בנוסף, נראה איך משתמשים במבנה נתונים זה למציאת תת המחרוזת החוזרת המקסימלית ב-a.

נכיר כעת את מבנה הנתונים:

חלק א׳ (50 נקודות) – בניית עץ סיפות: Suffix Tree

בחלק זה של העבודה, תממשו מבנה נתונים בשם Suffix Tree.

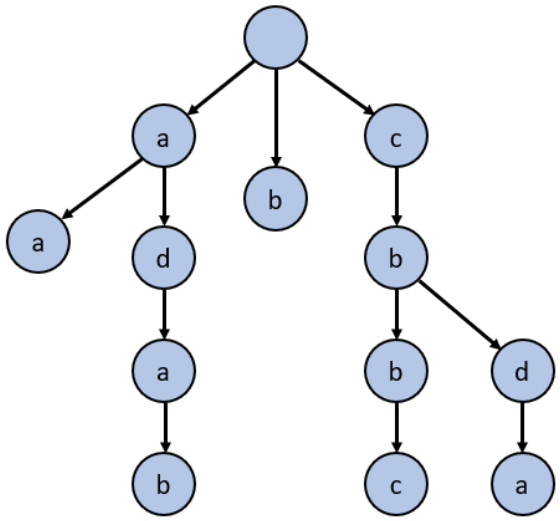
לצורך הגדרת המבנה, נגדיר את המושגים הבאים:

א. רישא (Prefix) – רישא של אובייקט מסויים היא התחילית שלו. מגיע מהמילה "ראש". דוגמאות:
- הרישא בגודל 2 של המערך [1,2,3,4] היא [1,2].
- הרישא בגודל 6 של המחרוזת "HelloWorld" היא "HelloW".

ב. סיפא (Suffix) – הניגוד של רישא. סיפא של אובייקט מסויים היא הסוף שלו. מגיע מהמילה "סוף". דוגמאות:
- הסיפא בגודל 2 של המערך [1,2,3,4] היא [3,4].
- הסיפא בגודל 6 של המחרוזת "HelloWorld" היא "oWorld".

נגדיר את מבני הנתונים הבאים:

1. Trie: זהו עץ **(לא בינארי)** בעל המאפיינים הבאים:



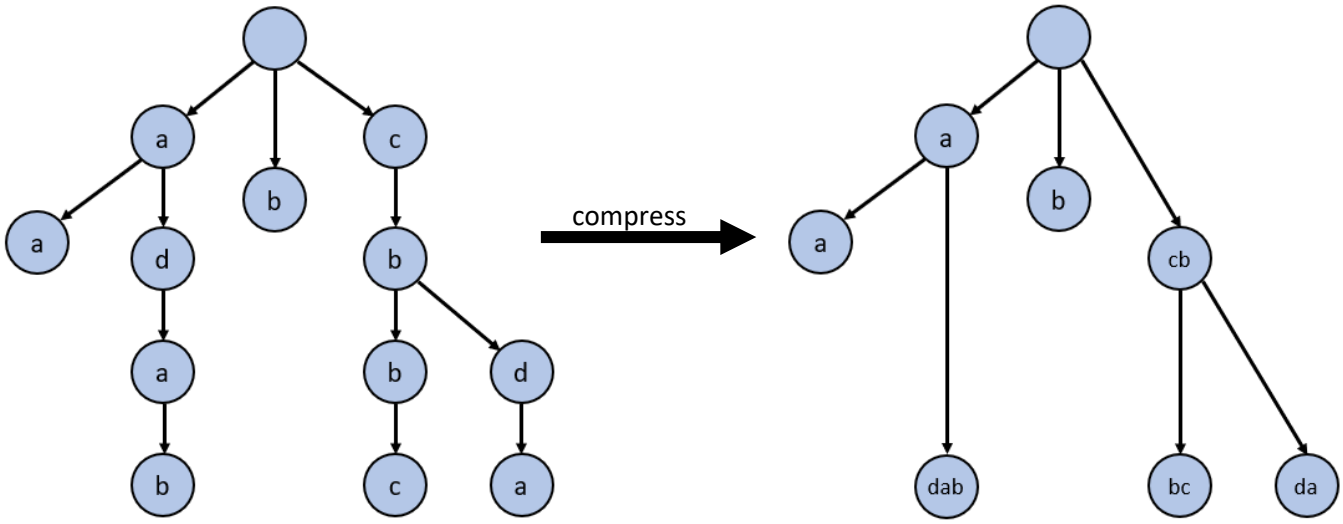
- כל צומת פרט לשורש מייצג אות. בדוגמא כאן, הבן השמאלי של השורש מייצג את האות a. יכולים להיות כמה צמתים בעץ שמייצגים את אותה האות.
- כל מסלול שמתחיל בשורש ומתקדם מטה מייצג מילה.

בדוגמא כאן, עבור המילים aa, adab, b, cbbc, cbda התקבל העץ הזה. ניתן לראות שלכל מילה קיים מסלול מהשורש לכיוון אחד העלים שמייצג את המילה. כפי שאתם רואים בדוגמא זו, כל מסלולי המילים הנ״ל נגמרים בדיוק בעלה. זה קורה כאשר אף מילה מתוך סט המילים שלנו אינה רישא (prefix) של השניה. כלומר, במקרה כזה יש יחס של אחד לאחד בין עלי העץ למילים אותן הוא מייצג.

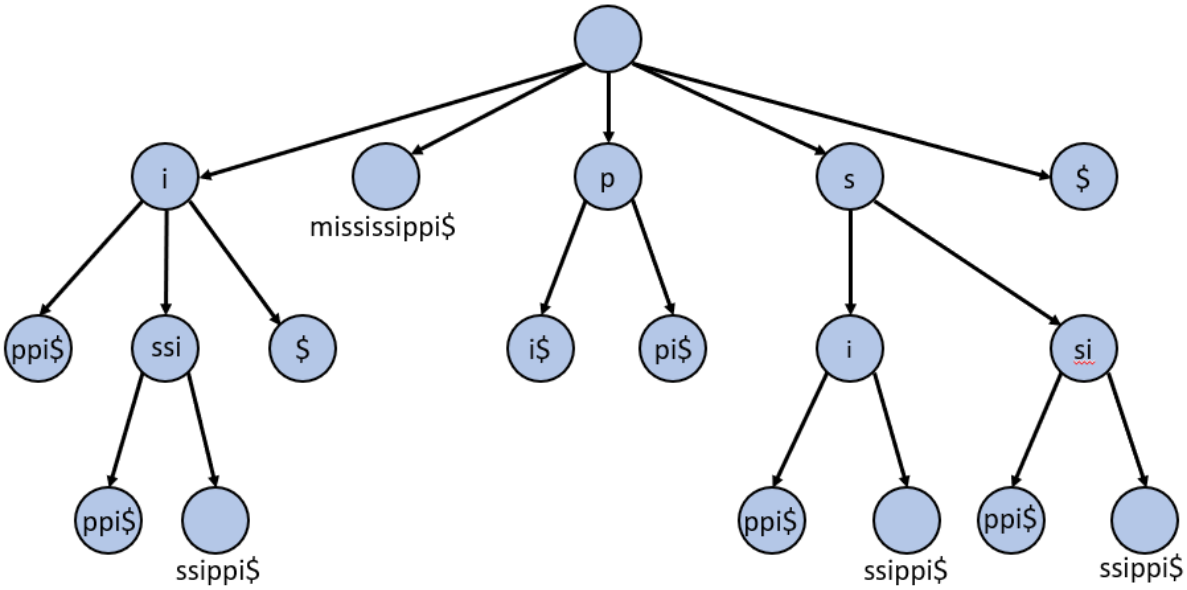
נשים לב שאם ל-2 מילים יש רישא משותפת באורך i, אז גם קיים להם מסלול משותף באורך i מהשורש ולאחריו פיצול (כלומר האב הקדמון הקטן ביותר המשותף ל-2 העלים הוא בעומק i).

בדוגמא זו למילים cbbc, cbda יש רישא משותפת באורך 2 ולכן גם המסלול ההתחלתי באורך 2 שלהם משותף.

2. Compressed Trie : זהו העץ המתקבל ע"י מיזוג כל הצמתים שיש להם בן אחד בדיוק, עם הבן שלהם, פרט לשורש. נעדיף לעבוד על עץ כזה לצורך חיסכון בזיכרון.



3. Suffix Tree : עבור מחרוזת S, Suffix Tree זהו Compressed Trie שבו סט המילים של העץ הוא כל הסיפות של S. כדי להשיג יחס של אחד לאחד בין עלי העץ למילים, נרצה שאף מילה לא תהיה רישא של השניה. כדי להשיג זאת, נוסיף לסוף המילה S תו מיוחד שאינו שייך לאלפבית של המילים שלנו, כמו למשל $.
בדוגמה למטה המילה S היא mississippi ולכן הסיפות שהוכנסו לעץ הן :



1 mississippi$
2 ississippi$
3 ssissippi$
4 sissippi$
5 issippi$
6 ssippi$
7 sippi$
8 ippi$
9 ppi$
10 pi$
11 i$
12 $

$S = \text{mississippi\$}$

בדוגמא זו ניתן לראות שאם לא היינו מוסיפים את התו המיוחד $, המילה i שהיא סיפא חוקית, הייתה נגמרת בצומת פנימית ולא בעלה.

מספר אבחנות לגבי עץ הסיפות:

1. מספר הבנים בכל צומת הוא לכל היותר כגודל הא״ב ממנו בנויות המחרוזת (1+ בגלל התו המיוחד).
2. אורך המסלול הארוך ביותר בעץ הוא לכל היותר כאורך המילה (בפועל בגלל הכיווץ נקבל מסלולים קצרים יותר).

מימוש העץ-

במימוש שלנו, לכל צומת בעץ קיימים השדות הבאים:

- **children** - מערך של הבנים הישירים של הצומת. המערך בגודל קבוע וניתן להניח שהוא גדול מספיק בשביל כל האלפבית שלנו. המערך ממויין לפי סדר לקסיקוגרפי של המילים בתוך הבנים. בנוסף כל איברי המערך מיושרים לשמאלו, כלומר כל התאים הריקים במערך (null) מיושרים לימין. לדוגמא, מערך הבנים של השורש בדוגמא למעלה הוא [i, mississippi$, p, s, $, null, null, null, …]. אם כעת לצורך ההדגמה נרצה להוסיף בן עם המילה ״ra״ נצטרך להזיז (לעשות shift) לחלק מאיברי המערך כדי לשמור על המיון, ונקבל את המערך [i, mississippi$, p, ra, s, $, null, null, …].
  שימו לב: כדי להשוות בין 2 אחים, מספיק להשוות רק את התו הראשון שלהם, ולא את כל המילה עצמה. זה נכון בגלל התכונות של compressed trie.
  שימו לב 2: לצורך הנוחות, הבנים בדוגמא מעלה מתוארים כמחרוזות, אבל כמובן שהבנים הם אובייקט מסוג צומת עץ, בדיוק כמו האב.
- **numOfChildren** – מספר הבנים בפועל שיש לצומת (≤ גודל המערך children).
- **chars** – ייצוג של מחרוזת הקודקוד. במקום להשתמש במחלקה String, בחרנו כאן להשתמש ברשימה מקושרת של תווים `CharLinkedList`, כדי שיהיה יותר קל לעבור על התווים אחד אחד. כחלק מהעבודה תצטרכו לממש את הרשימה המקושרת. לאחר המימוש, לנוחיותכם מצורפת מתודת מחלקה (סטטית) `CharLinkedList.from(char c)` שיוצרת רשימה מקושרת עם איבר בודד c.
- **descendantLeaves** – מספר העלים בתת העץ של הצומת (הערך שווה 1 עבור צומת שהוא עלה).
- **parent** – מצביע (פוינטר) לצומת האב (null אם מדובר בשורש).
- עוד שדות שיהיו רלוונטיים בהמשך.

תיאור אלגוריתם לייצור עץ סיפות בהינתן מחרוזת S:

1. שרשר ל-S את התו $.
2. לכל $i \in \{0, \dots, |S|\}$ הוסף את הסיפא של S , המתחילה מהמקום ה-i במחרוזת, לשורש העץ.

תיאור אלגוריתם להוספת סיפא של S מהמקום ה-i לצומת node בעץ:

1. אם $i == S.length$ עצור.
2. סמן ב-c את התו במקום ה-i של S.
3. אם ל-node אין בן עם מחרוזת שמתחילה ב-c, צור בן כזה והוסף למערך הבנים (לא לשכוח לשמור על המיון).
4. אחרת, מצא את הבן הזה (בעזרת חיפוש בינארי על המערך).
5. כך או כך, הוסף לבן שנמצא (או נוצר) בצורה רקורסיבית את הסיפא של S מהמקום ה-i+1.

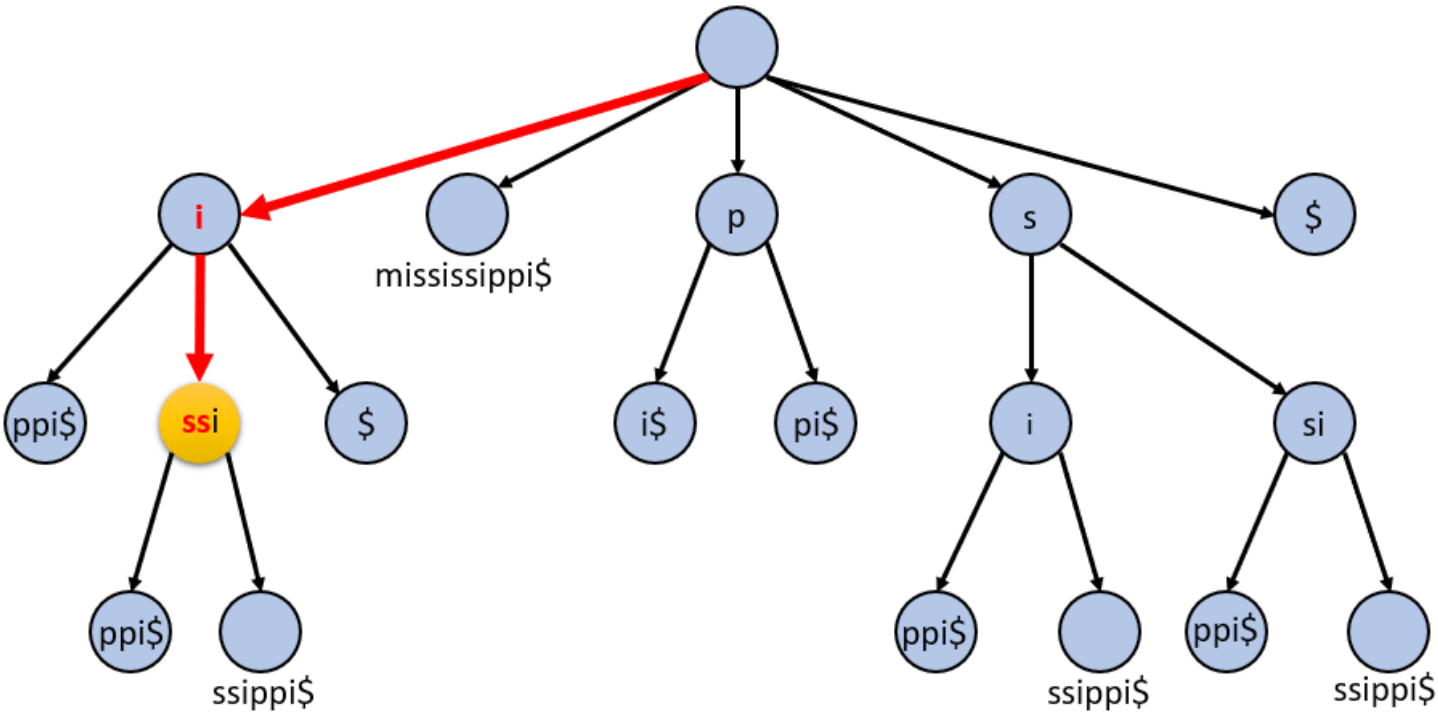
לאחר הוספת כל הסיפות לעץ, כל שנדרש כעת הוא לכווץ את העץ ע"י שימוש בחוק של Compressed Trie.

חלק ב' (20 נקודות) – שימוש בעץ סיפות למימוש שאילתות contains ו-numOfOccurrences

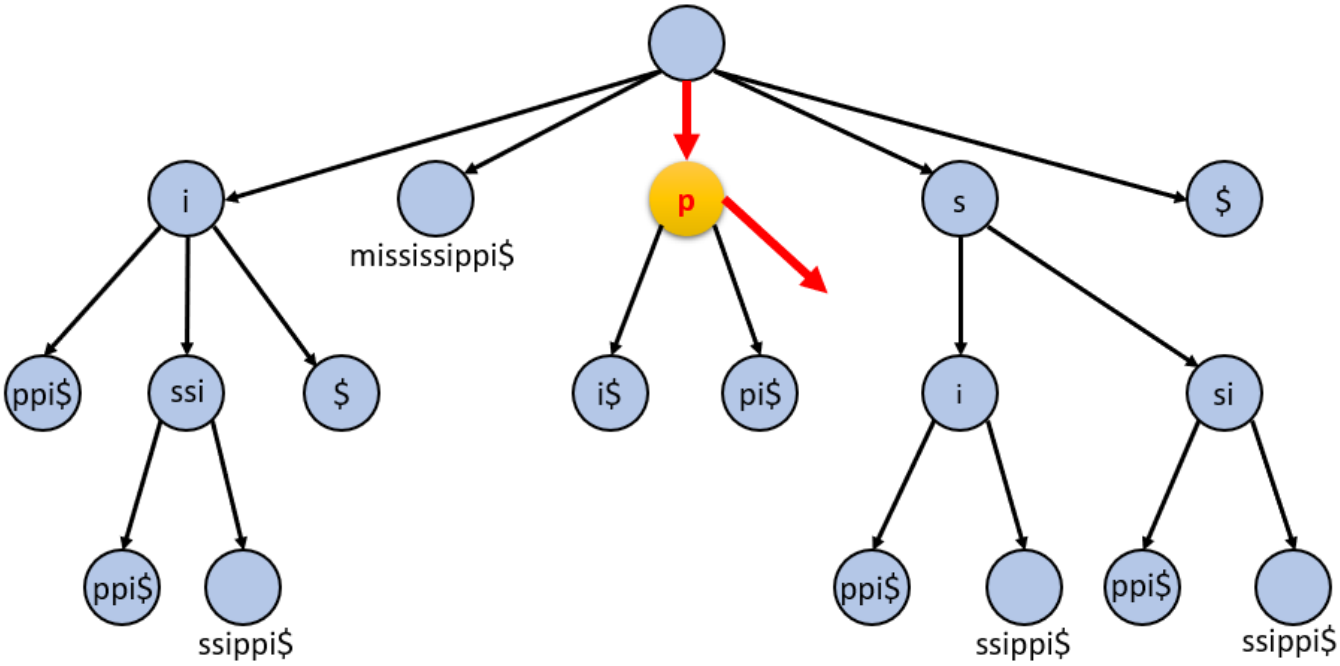
כעת יש לנו עץ סיפות של מחרוזת S מלא ומוכן לשימוש. נדגים כיצד בעזרתו ניתן לבדוק כמה פעמים מחרוזת b מוכלת ב-S בזמן ריצה של $O(|b|)$:

בהינתן מחרוזת b, "נרוץ" על העץ מהשורש לאורך המסלול שמכתיבים תווי b. אם "עפנו" מהעץ באמצע, כלומר לא קיים צומת בן מתאים, אז b לא מוכל ב-S. אחרת, סיימנו לסרוק את b והגענו לקודקוד כלשהו בעץ. כיוון שקיים יחס של אחד לאחד בין עלי העץ למס' המילים בעץ, נקבל שכמות העלים שהם צאצאים של הקודקוד היא כמות המופעים של b בתוך S. נראה דוגמאות על עץ הסיפות של המילה mississippi שראינו קודם:

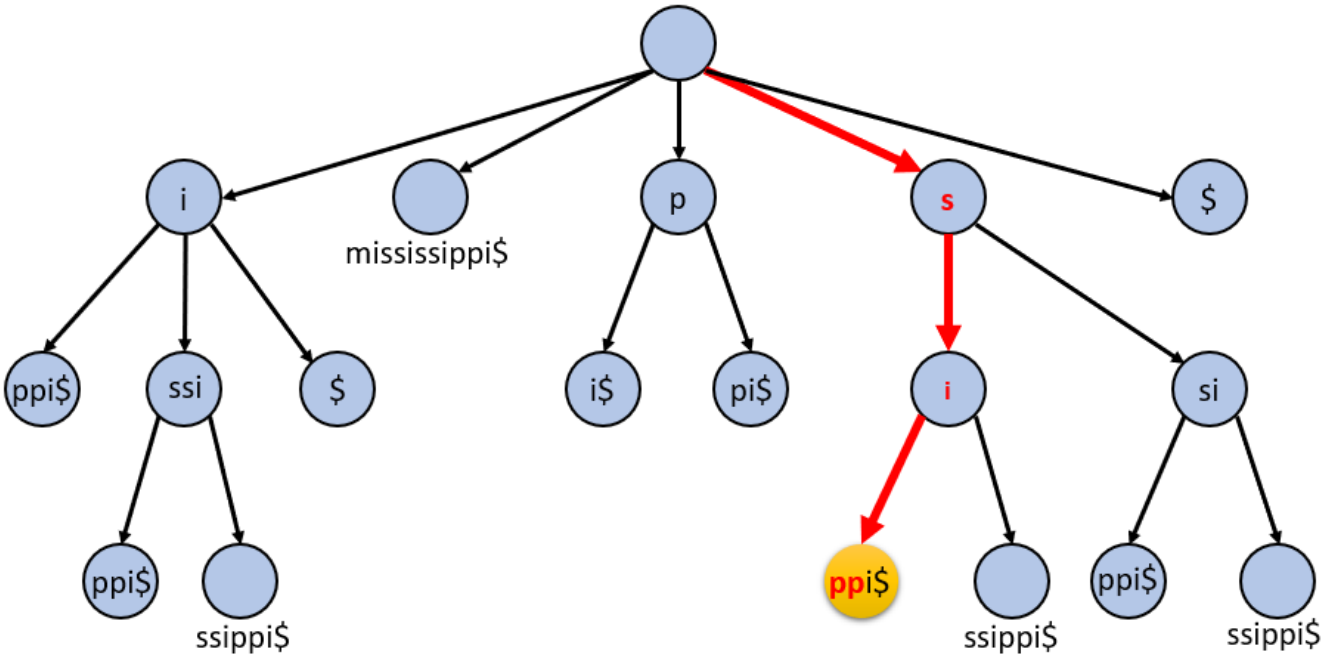
עבור הקלט iss, נסרוק את העץ, נגיע לצומת הצהוב דרך החצים האדומים בדוגמה מטה ונעצור. לצומת זה יש 2 עלים שהם צאצאים שלו (במקרה זה גם בנים ישירים), ואכן המחרוזת iss מופיעה פעמיים ב-mississippi.



עבור הקלט psi, נגיע לצומת הצהוב המסומן מטה ולאחר מכן ניזרק מהעץ כיוון שאין לו בן שמתחיל באות s, ואכן המחרוזת psi לא מוכלת ב-mississippi.



עבור המחרוזת sipp נגיע לצומת הצהוב המסומן מטה ונעצור. לצומת זה יש עלה אחד שהוא צאצא שלו (הוא עצמו) ואכן המחרוזת sipp מופיעה פעם אחת ב-mississippi.



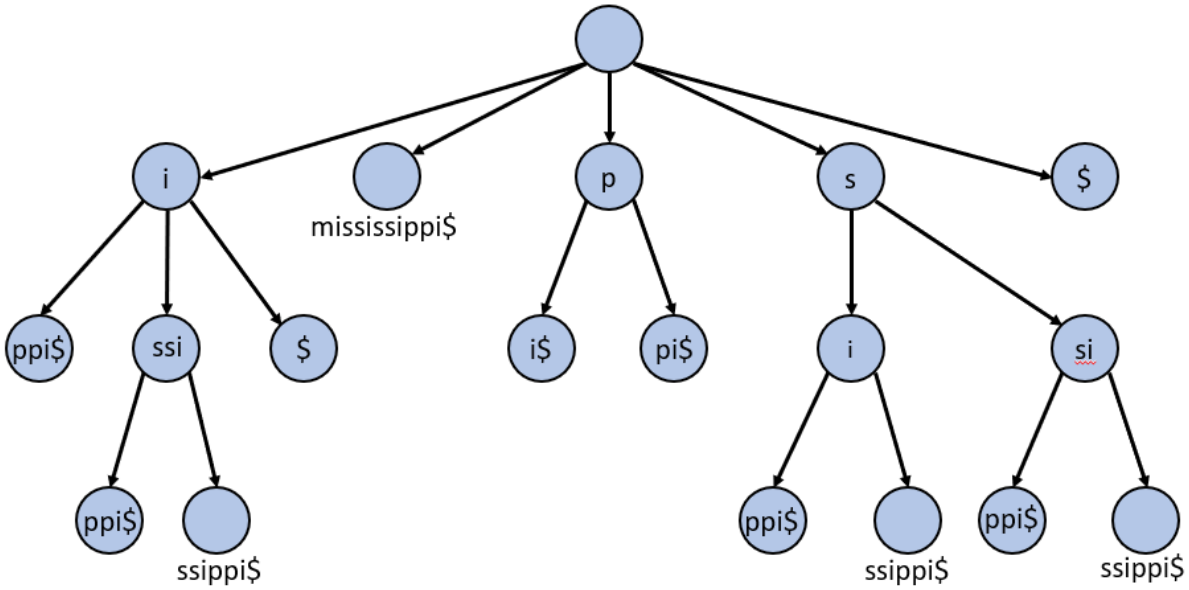
חלק ג' (20 נקודות) – שימוש בעץ סיפות למציאת תת המחרוזת החוזרת המקסימלית

נראה כעת שימוש נוסף ל-Compressed Trie:

בהינתן מחרוזת S, נרצה למצוא את תת המחרוזת המקסימלית שחוזרת על עצמה בתוך המחרוזת שהתקבלה.

- Longest repeated substring: בהינתן מילה נרצה למצוא מתוך כל תתי המחרוזות אשר חוזרות על עצמן יותר מפעם אחת, את תת המחרוזת הארוכה ביותר מבחינת מספר תווים.

נבנה Compressed Trie שיכיל את כל הסיפות של S משורשר ל-$, כפי שעשינו בעץ הסיפות הרגיל. בנוסף לכך, נחזיק בעץ שדה בשם maxLength – שהוא גודל תת המחרוזת המקסימלית, וsubstringStartNode שמכיל את הקודקוד שממנו ניתן לשחזר את תת המחרוזת המקסימלית. בהינתן המילה, נחשב באופן מיידי את שני שדות אלו, ובקריאה לפונקציות המתאימות נחזיר את המידע המבוקש. נקרא לעץ זה longest Repeated Suffix Tree. דוגמא עבור המילה mississippi:



מהאופן שבו בנוי עץ הסיפות, אם יש תת מחרוזת אשר חוזרת ונשנת בטקסט מספר פעמים, כל אותם הפעמים יעברו באותו נתיב בעץ בשלב מסויים. כלומר, שתי תתי מחרוזות חוזרות על עצמם בעץ במידה ויש במסלול בעץ שהם עוברים בו לפחות צומת אחד שהוא פנימי ולא עלה (וכמובן לא השורש).

עבור הדוגמא של Mississippi, ישנם מספר תתי מחרוזות אשר חוזרות על עצמן בעץ. ניתן לזהות אותן באופן הבא:

במסלול של תת המחרוזת "i" יש צומת פנימי אחד במורד העץ.

במסלול של תת המחרוזת " issi " יש שני צמתים פנימיים במורד העץ.

במסלול של תת המחרוזת "p" יש צומת פנימי אחד במורד העץ.

במסלול של תת המחרוזת "s" יש צומת פנימי אחד במורד העץ.

במסלול של תת המחרוזת "si" יש שני צמתים פנימיים במורד העץ.

במסלול של תת המחרוזת "ssi" יש שני צמתים פנימיים במורד העץ.

תת המחרוזת המקסימלית תהיה בצומת בעומק הגדול ביותר והרחוק ביותר מן השורש. כלומר נרצה למצוא את הצומת בעל המרחק המקסימלי מן השורש לפני ביצוע ה Compress של העץ (**בעל מספר התווים המקסימלי**) ולשמור אותו במשתנה substringStartNode. במקרה של הדוגמא שלנו תת המחרוזת המקסימלית הינה issi.

טיפ: השתמשו בפונקציה רקורסיבית פרטית שתצרו, שתעזור לכם לטייל בעץ על מנת למצוא את הצומת המתאים לדרישה.

שימו לב – במימוש שלנו, במידה ויש מחרוזת כמו aaa שמכילה את תת המחרוזת aa פעמיים עם חפיפה, התוצאה שתצטרך לחזור היא aa. בנוסף, במידה וישנם שני תתי מחרוזות מקסימליות, יש להחזיר את הראשונה לפי סדר לקסיקוגרפי! כלומר עבור המחרוזת "ioiosbdbd", התת מחרוזת המקסימלית החוזרת הינה bd.

לצורך מימוש הפונקציונליות, הוספנו לאובייקט הצומת בעץ שדה נוסף:

- totalWordLength – אורך המילה שנוצרת מהמסלול מהשורש עד לצומת זה כולל.

חלק ד' (10 נקודות)- בדיקת הקוד

כחלק מהתרגיל, אתם תידרשו לכתוב בדיקות (tests) עבור הקוד שלכם. פירוט של חלק זה נמצא תחת המחלקה Tester בחלק המפרט את המחלקות בתרגיל.

פירוט המחלקות בתרגיל: מומלץ לקרוא את התיעוד של הפונקציות בממשקים ובמחלקות האבסטרקטיות

המחלקות שאנו מספקים לכם- אין לשנות מחלקות אלה:

המחלקות שסופקו לכם הן אבסטרקטיות וממומשות חלקית. יש להרחיב את המחלקות ולממש את הפונקציות האבסטרקטיות. שם המחלקה שתיצרו יהיה ***ClassNameImpl*** כאשר ***ClassName*** זהו שם המחלקה האבסטרקטית.

Interface *ICharLinkedListNode -* The interface `ICharLinkedListNode` (in the ICharLinkedListNode.java file) specifies the desired functions we want for a single node in the char linked list.

Class *CharLinkedList -* The abstract class `CharLinkedList` is provided with the needed fields in order to create an effective linked list, and a constructor. It also contains:

- A static function `from(char c)`, which you can use to easily create a linked list with a single character in it, once the `CharLinkedListImpl` is implemented.
- An iterator implementation to be able to iterate the list using foreach (i.e. `for (char c : list)`)
- A convenient `toString()` method for debugging purposes.
- Getters and setters.

Class *SuffixTreeNode* – An instance of the abstract class `SuffixTreeNode` is a single node in a suffix tree. The class is provided with the needed fields in order to handle all nodes' desired functions and a constructor. It also contains:

- A method `restoreStringAlongPath()` to restore and retrieve the string written in the path from the root to this node.
- A convenient `toString()` method for debug purposes.
- Getters and setters.

Class *CompressedTrie -* This class is fully implemented and provides the functionality to build a compressed trie.

Class *SuffixTree* – This abstract class extends `CompressedTrie` and is provided with a constructor to build a suffix tree from a given word.

Class longestRepeated*SuffixTree* – This abstract class extends `CompressedTrie` and is provided with a constructor to build the longest repeated suffix tree from a given word.

המחלקות שאתם צריכים לממש או שניתנו באופן חלקי:

Class *CharLinkedListNodeImpl* (Included in part 1 grade):

This class should implement the `ICharLinkedListNode` interface. All Methods should be implemented in $O(1)$ running time.

Class *CharLinkedListImpl* (Included in part 1 grade):

This class extends the `CharLinkedList` abstract class. All methods should be implemented in $O(1)$ running time except method size `()` which should be implemented in $O(n)$ running time, with n being the number of characters in the list. Adding new fields to this class or the abstract class is not allowed.

Class *SuffixTreeNodeImpl*:

This class extends the `SuffixTreeNode` abstract class and should implement the following functions:

`SuffixTreeNode search(char c)`	פונקציית מעטפת ל-`binarySearch`. שייך לניקוד חלק 1.
`SuffixTreeNode binarySearch(char target, int left, int right)`	השיטה מחפשת בשיטת חיפוש בינארי את הבן הישיר של הצומת שהתו הראשון שלו הוא target. להזכירכם: כדי להשוות בין 2 אחים מספיק להשוות את התו הראשון שלהם כי הוא בהכרח שונה. על השיטה להחזיר null אם לא נמצא בן כזה. זמן ריצה: $O(logn)$ כאשר n הוא מס' הבנים בפועל של הצומת. שייך לניקוד חלק 1.
`void shiftChildren(int until)`	השיטה מזיזה את כל התאים במערך הילדים תא אחד ימינה, עד התא באינדקס *until* כולל. ניתן להניח שמערך הילדים מספיק גדול. בסיום השיטה, זה לא משנה מה נמצא במערך באינדקס *until* (יכול להיות null או הערך הקודם). זמן ריצה: $O(n)$ כאשר n הוא מס' הבנים בפועל של הצומת. שייך לניקוד חלק 1.
`void addChild(SuffixTreeNode node)`	השיטה מוסיפה בן חדש למערך הבנים של הצומת. ניתן להניח שהילד לא היה שייך למערך קודם. יש לשמור על המיון הלקסיקוגרפי. זמן ריצה: $O(n)$ כאשר n הוא מס' הבנים בפועל של הצומת. שייך לניקוד חלק 1.
`void addSuffix(char[] word, int from)`	השיטה מוסיפה לצומת את הסיפא של *word* מאינדקס *from* כולל, בצורה רקורסיבית, לפי האלגוריתם שתואר מעלה. המחרוזת *word* מיוצגת כמערך תווים כדי שיהיה יותר קל ומהיר לגשת לתו באינדקס מסוים. כיוון שבעץ סיפות אנחנו קודם נוסיף את כל הסיפות ורק אחר כך נכווץ את העץ, ניתן להניח שבקריאה ל-`addSuffix` לכל הצמתים יש

תו אחד בדיוק (פרט לשורש). זמן ריצה: $O(\|word\|)$. שייך לניקוד חלק 1.	
השיטה מכווצת את העץ ע"י הסרת כל הצמתים שיש להם בן אחד בדיוק, ומיזוג התוכן שלהם עם התוכן של הבן (היחיד) שלהם. דוגמא גרפית לפלט נמצאת בתיאור של CompressedTrie. זמן ריצה: $O(T)$ כאשר T הוא גודל תת העץ המושרש ע"י הצומת (כלומר מס' הצאצאים של הצומת). שייך לניקוד חלק 1.	`void compress()`
השיטה מחזירה את מס' המופעים של הסיפא של *subword* מאינדקס *from* כולל. רצוי לממש בצורה רקורסיבית. מומלץ לראות דוגמאות לקריאה לפונקציה בתוך תיעוד הפונקציה בקוד. ניתן להניח שהמחרוזת לא ריקה. זמן ריצה: $O(\|subword\|)$. שייך לניקוד חלק 2.	`int numOfOccurrences(char[] subword, int from)`

<u>על כל השיטות לעדכן את שדות המחלקה במידת הצורך! אין להוסיף שדות נוספים!</u>

Class *SuffixTreeImpl* (Included in part 2 grade):

This class should extend the `SuffixTree` abstract class. Implement The abstract functions. Don't overthink it too much, each method can be implemented in one line of code. For both `contains` and `numOfOccurrences` methods you can assume the given string is not empty.

Class *longestRepeatedSuffixTreeImpl* (Included in part 3 grade):

This class should extend the `longestRepeatedSuffixTree` abstract class. Implement The abstract function `createLongestRepeatedSubstring()` and `getLongestRepeatedSubstring()` as mentioned earlier.

Class *Tester*

The class `Tester` will help you test your code. You should add tests for every public method. Your tests should include extreme (״boundary״) cases, as well as the obvious ״middle of the road״ cases. Remember to choose **diverse** test cases.

For example, testing `new SuffixTree("mississippi").contains(String subword)` with strings ״sis״, ״sip״ and ״ipp״ is redundant, since they all test the exact same thing – in this case, multi chars substrings that are contained in ״mississippi״ once and are not in the edge of the word.

We suggest the following approach:

- Test basic methods before complicated ones.
- Test simple objects before objects that include or contain the simple objects.
- Do not test too many things in a single test case.

We also provide the file Tester.java. This file includes a main function that will be in-charge of running all the tests for your code. A helper function that you will use is the following:

```
private static void test (boolean exp, String msg)
```

This function receives a Boolean expression, and an error message. If the Boolean expression is evaluated to false, the function will print the error message to the screen. The function will also "count" for you the number of successful tests that you executed.

Please read carefully the code in this file. As you can see, we already provided a few tests to your code. You are required to add your own tests to check Part A, B and C of the assignment, so the output of running the `Tester`, will be:

```
All <i> tests passed!
```

Where `i` (the total number of tests) should be at least **50**.

הנחיות הגשה:

עליכם להגיש את כל המחלקות שצוינו לעיל (מלבד המחלקות שניתנו לכם באופן מלא).

אופן ההגשה יפורט בהמשך.

באופן עקרוני הקובץ שעליכם להגיש הוא קובץ **zip** ובתוכו תיקיה אחת בשם **hw2** שמכילה את הקבצים:

CharLinkedListImpl.java, CharLinkedListNodeImpl.java, longestRepeatedSuffixTreeImpl.java, SuffixTreeImpl.java, SuffixTreeNodeImpl.java, Tester.java

כל סטייה מהוראות אלו יגמרו לכך שעבודתכם לא תרוץ ולא תעבור את הטסטים. נא לבדוק עצמכם פעמיים!

בהצלחה ועבודה מהנה!