שפות תכנות - הגשה 1

שאלה 1:

Method 1

```
<data file> ::= ( <field>+ ‘;’ (< field> ) ‘.’ )+
<field>     ::= <digit>+ | “<letter>+” | “<digit>+”
<letter>    ::= a | b | c … | z | A | B … | Z
<digit>     ::= 0 | 1 | 2 | 3 …  | 8 | 9
```

Method 1 - Long way

```
<data file> ::= <list>+
<list>      ::= <field>+ .
<field>     ::= (<integer> | <string>) ;?
<integer>   ::= <digit>+
<string>    ::= “ <digit>+ | <letter>+ “
<letter>    ::= a | b | c … | z | A | B … | Z
<digit>     ::= 0 | 1 | 2 | 3 …  | 8 | 9
```

Method 2 - Recursive

```
<data file> ::= <list> . | <file> <list> .
<list>      ::= <field> | <field> ;
                <field> ; <list> ; |
                <field> ; <list>
<field>     ::= <integer> | “ <string> “
<integer>   ::= <digit> | <digit> <integer>
<string>    ::= <digit> | letter |
                <string> <digit>  |
                <string> <letter>
<letter>    ::= a | b | c … | z | A | B … | Z
<digit>     ::= 0 | 1 | 2 | 3 …  | 8 | 9
```

שאלה 3:

C	**Java**	**Python**	
Compilation Execution speed - Fast Portability - Poor Compilation time - Slow	Compilation - JIT Execution speed - Fast Portability - Good Compilation time - Fast	Interpreter Execution speed - Slow Portability - Good Compilation time - Fast	**Compilation vs Interpreter**

אין מחלקות בשפה, קיים Struct
 	
struct structName
{
   [datatype] var1;
   [datatype] var2;
};
	OOP Language
	
	public class className()
	{
	   [datatype] var1;
	   [datatype] var2;
	
	   public className([datatype] v1, [datatype] v2)
	   {
	      var1 = v1;
	      var2 = v2;
	   }
	
	   // functions …
	}
		OOP Language
		
		class className:
		   var1 = 0
		   def __init__(self, v2):
		      var2 = v2
		   // functions …
			אתחול (מחלקה)

יש להגדיר טיפוס
 	
[datatype] varName;
int/char/float/double …
	יש להגדיר טיפוס
	
	[datatype] varName;
	int/char/float/double …
		אין חשיבות לטיפוס
		
		var1 = “this is String”
		var2 = 0 # this is int
			משתנה באופן כללי

static [datatype] var3 = [value];	static [datatype] var3; public static class staticClass ()	כל המשתנים המוגדרים ברמת המחלקה הם Static	**משתנה מחלקה (Static)**
אין Objects בשפה, קיים Struct struct structName var4 = {v1, v2}	var4 = new className(v1, v2)	class_var = ClassName(v2)	**משתנה אובייקט**

[datatype] func() {
   // doSomething
}
 	
datatype or void
	public [datatype] func() {
	   // doSomething
	}
	
	datatype or void
		אין צורך להגדיר ערך מוחזר
		
		def func():
		   # doSomething
		
		חובה להקפיד על הזחות
			מתודה

static void func() {
   // doSomething
}
	public static void func() {
	   // doSomething
	}
		@staticmethod
		def func():
		   # doSomething
			מתודה סטטית

שאלה 2: (מצורפים קבצי המקור)

trie.py

```
def getIndex(ch):
    return (ord(ch) - ord('A') + 1) if ord(ch) != 0 else 0


class TrieNode:
    sigma = 26  # N - Size of the alphabet

    # init node
    def __init__(self):
        self.children = [None] * (self.sigma + 1)

    # Get children by index
    def getChildren(self, index):
        return self.children[index]

    # Set children by index
    def setChildren(self, index, node):
        self.children[index] = node

    # Check if node is leaf
    def isLeaf(self):
        if self.children[0] is None:
            return False
        for i in range(1, self.sigma + 1):
            if self.children[i] is not None:
                return False
        return True


class Trie:
    # init Trie
    def __init__(self):
        self.root = TrieNode()

    # insert to Trie
    def insert(self, str):
        curr = self.root

        for ch in str:
            index = getIndex(ch)
            if curr is not None and curr.getChildren(index) is None:
                curr.setChildren(index, TrieNode())

            curr = curr.getChildren(index)

        curr.setChildren(0, TrieNode())

    # Search in Trie
    def search(self, str):
        curr = self.root
        for ch in str:
            index = getIndex(ch)
            if curr.getChildren(index) is None:
                return 0
            curr = curr.getChildren(index)
        return 1 if curr is not None and curr.getChildren(0) is not
None else 0

    # Recursive function to remove a string from a Trie
    def remove(self, node, str):
        str_len = len(str)

        if node is None:
            return False

        if str_len > 0:
            index = getIndex(str[0])
            next_node = node.getChildren(index)
            if next_node is None:
                return False

            self.remove(next_node, str[1:])

            if next_node.isLeaf():
                node.setChildren(index, None)
                return False

        return True


# Test
strings = ["ROMANE", "ROMANUS", "ROMULUS", "RUBENS", "RUBER", "RUB",
"RUBICON", "RUBICUNDUS", "RUBY", "ROAD"]
found_msg = ["Not found", "Found"]

root = Trie()

for s in strings:
    root.insert(s)

s = "RUBER"  # String: s

print("--- Before Remove ---")
print("Search %s in Trie:\t%s" % (s, found_msg[root.search(s)]))
print("Search %s in Trie:\t%s" % ("RUBY",
found_msg[root.search("RUBY")]))
print("Search %s in Trie:\t%s" % ("RUBI",
found_msg[root.search("RUBI")]))

print("\nDeletes %s..." % s)
if root.remove(root.root, s):
    print(s, "Successfully deleted!")


print("\n--- After Remove ---")
print("Search %s in Trie:\t%s" % (s, found_msg[root.search(s)]))
print("Search %s in Trie:\t%s" % ("RUBY",
found_msg[root.search("RUBY")]))
print("Search %s in Trie:\t%s" % ("RUBI",
found_msg[root.search("RUBI")]))
```

trie.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define SIGMA 26 // N - Size of the alphabet

typedef struct node {
    struct node *children[SIGMA + 1]; // +1 for \0
} TrieNode;

// Get index by subtracting the ASCII value of ch at the value of 'A'
int getIndex(char ch) {
    return (ch == '\0') ? 0 : ((int)ch - (int)'A' + 1);
}

// Check if node is leaf
int isLeaf(TrieNode* pNode) {
    if (pNode->children[0] == NULL)
        return 0;
    for (int i = 1; i < SIGMA + 1; i++)
        if (pNode->children[i])
            return 0;
    return 1;
}

// init new Trie node
TrieNode* init(void) {
    TrieNode *pNode = NULL;

    pNode = (TrieNode*) malloc(sizeof (TrieNode));

    if (pNode) {
        for (int i = 0; i < SIGMA + 1; i++)
            pNode->children[i] = NULL;
    }

    return pNode;
}

// Insert to Trie
void insert(TrieNode* pNode, const char* str) {
    int index;
    int len = strlen(str);

    TrieNode* pCurrNode = pNode;

    for (int i = 0; i < len; i++) {
```

```c
        index = getIndex(str[i]);
        if (!pCurrNode->children[index])
            pCurrNode->children[index] = init();

        pCurrNode = pCurrNode->children[index];
    }

    if (!pCurrNode->children[0])
        pCurrNode->children[0] = init();
}

// Recursive function to remove a string from a Trie
int removeString(TrieNode *pNode, char* str) {
    int index;
    TrieNode* pNextNode;

    if (pNode == NULL || !pNode)
        return 0;

    if (*str) {
        index = getIndex(*str);
        pNextNode = pNode->children[index];
        if (!pNextNode)
            return 0;

        removeString(pNextNode, str + 1);

        if (isLeaf(pNextNode)) {
            free(pNextNode);
            pNode->children[index] = NULL;
            return 1;
        }
    }

    return 1;
}

// Search in Trie
int search(TrieNode* pNode, const char* str) {
    int index;
    int len = strlen(str);
    TrieNode* pCurrNode = pNode;

    for (int i = 0; i < len; i++) {
        index = getIndex(str[i]);

        if (!pCurrNode->children[index])
            return 0;

        pCurrNode = pCurrNode->children[index];
    }

    return (pCurrNode != NULL && pCurrNode->children[0]) ? 1 : 0;
}

int main() {
    // #define CHAR_MAX 127 // Default
    char strings[][CHAR_MAX] = { "ROMANE", "ROMANUS", "ROMULUS",
"RUBENS", "RUBER", "RUB", "RUBICON", "RUBICUNDUS", "RUBY", "ROAD" }; //
s - String
    char found_msg[][CHAR_MAX] = { "Not found", "Found" };

    TrieNode* trieRoot = init(); // T - Rooted tree

    int arr_size = sizeof(strings) / sizeof(strings[0]);
//    int arr_size = 10;

    for (int i = 0; i < arr_size; i++)
        insert(trieRoot, strings[i]);

    char* s = "RUBER"; // String: s

    printf("--- Before Remove ---\n");
    printf("Search %s in Trie:\t%s\n", s, found_msg[search(trieRoot,
s)] );
    printf("Search %s in Trie:\t%s\n", "RUBY",
found_msg[search(trieRoot, "RUBY")] );
    printf("Search %s in Trie:\t%s\n", "RUBI",
found_msg[search(trieRoot, "RUBI")] );

    printf("\nDeletes %s...", s);
    if (removeString(trieRoot, s))
        printf("\n%s Successfully deleted!\n", s);


    printf("\n--- After Remove ---\n");
    printf("Search %s in Trie:\t%s\n", s, found_msg[search(trieRoot,
s)] );
    printf("Search %s in Trie:\t%s\n", "RUBY",
found_msg[search(trieRoot, "RUBY")] );
    printf("Search %s in Trie:\t%s\n", "RUBI",
found_msg[search(trieRoot, "RUBI")] );
    return 0;
}
```

RunTrie.java

```
public class RunTrie {
    public static void main(String[] args) {
        String strings[] = {"ROMANE", "ROMANUS", "ROMULUS", "RUBENS",
"RUBER", "RUB", "RUBICON", "RUBICUNDUS", "RUBY", "ROAD"}; // s - String
        String found_msg[] = {"Not found", "Found"};

        Trie root = new Trie();

        for (int i = 0; i < strings.length; i++)
            root.insert(strings[i]);

        String s = "RUBER"; // String: s

        System.out.printf("--- Before Remove ---\n");
        System.out.printf("Search %s in Trie:\t%s\n", s,
found_msg[root.search(s)]);
        System.out.printf("Search %s in Trie:\t%s\n", "RUBY",
found_msg[root.search("RUBY")]);
        System.out.printf("Search %s in Trie:\t%s\n", "RUBI",
found_msg[root.search("RUBI")]);


        System.out.printf("\nDeletes %s...", s);
        if (root.remove(root.getRoot(), s))
        System.out.printf("\n%s Successfully deleted!\n", s);

        System.out.printf("\n--- After Remove ---\n");
        System.out.printf("Search %s in Trie:\t%s\n", s,
found_msg[root.search(s)]);
        System.out.printf("Search %s in Trie:\t%s\n", "RUBY",
found_msg[root.search("RUBY")]);
        System.out.printf("Search %s in Trie:\t%s\n", "RUBI",
found_msg[root.search("RUBI")]);
    }
}

class Trie {
    private TrieNode root;

    public Trie() {
        init();
    }

    // init Trie
    public void init() {
        root = new TrieNode();
    }

    // insert to Trie
    public void insert(String str) {
        int index;
        int len = str.length();

        TrieNode currNode = root;

        for (int i = 0; i < len; i++) {
            index = getIndex(str.charAt(i));
            if (currNode != null && currNode.getChildren(index) ==
null)
                currNode.setChildren(index, new TrieNode());

            currNode = currNode.getChildren(index);
        }

        currNode.setChildren(0, new TrieNode());
    }

    // Search in Trie
    public int search(String str) {
        int index;
        int len = str.length();
```

```java
        TrieNode currNode = root;

        for (int i = 0; i < len; i++) {
            index = getIndex(str.charAt(i));

            if (currNode.getChildren(index) == null)
                return 0;

            currNode = currNode.getChildren(index);
        }

        return (currNode != null && currNode.getChildren(0) != null) ?
1 : 0;
    }

    // Recursive function to remove a string from a Trie
    public boolean remove(TrieNode node, String str) {
        int index;
        TrieNode nextNode;

        if (node == null)
            return false;

        if (str.length() > 0) {
            index = getIndex(str.charAt(0));
            nextNode = node.getChildren(index);
            if (nextNode == null)
                return false;

            remove(nextNode, str.substring(1));

            if (nextNode.isLeaf()) {
                node.setChildren(index, null);
                return false;
            }
        }

        return true;
    }

    // Get index by subtracting the ASCII value of ch at the value of
'A'
    private int getIndex(char ch) {
        return (ch == '\0') ? 0 : ((int) ch - (int) 'A' + 1);
    }

    // Get root node
    public TrieNode getRoot() {
        return root;
    }
}

// TrieNode class
class TrieNode {
    private final int SIGMA = 26; // N - Size of the alphabet
    private TrieNode[] children;
    TrieNode() {
        children = new TrieNode[SIGMA + 1];
        init();
    }

    // init node
    public void init() {
        for (int i = 0; i < SIGMA + 1; i++)
            children[i] = null;
    }

    // Get children by index
    public TrieNode getChildren(int index) {
        return children[index];
    }

    // Set children by index
    public void setChildren(int index, TrieNode trieNode) {
        children[index] = trieNode;
    }

    // Check if node is leaf
    public boolean isLeaf() {
        if (children[0] == null)
            return false;
        for (int i = 1; i < SIGMA + 1; i++)
            if (children[i] != null)
                return false;
        return true;
    }
}
```