Tatiana Zihindula
C16339923

Algorithm and Program Design

CMPU1001
DT228
Year 1

Class Assignment

Table of Contents

I. Merging the three lists

1. Data structure: Binary Search Tree.

The binary search three used in this project preserves all the characterises of a generic binary search tree, except the fact that it **allows duplicated keys.** This is done **through an additional node pointing to the next student with the same surname**, if any.

Overview:

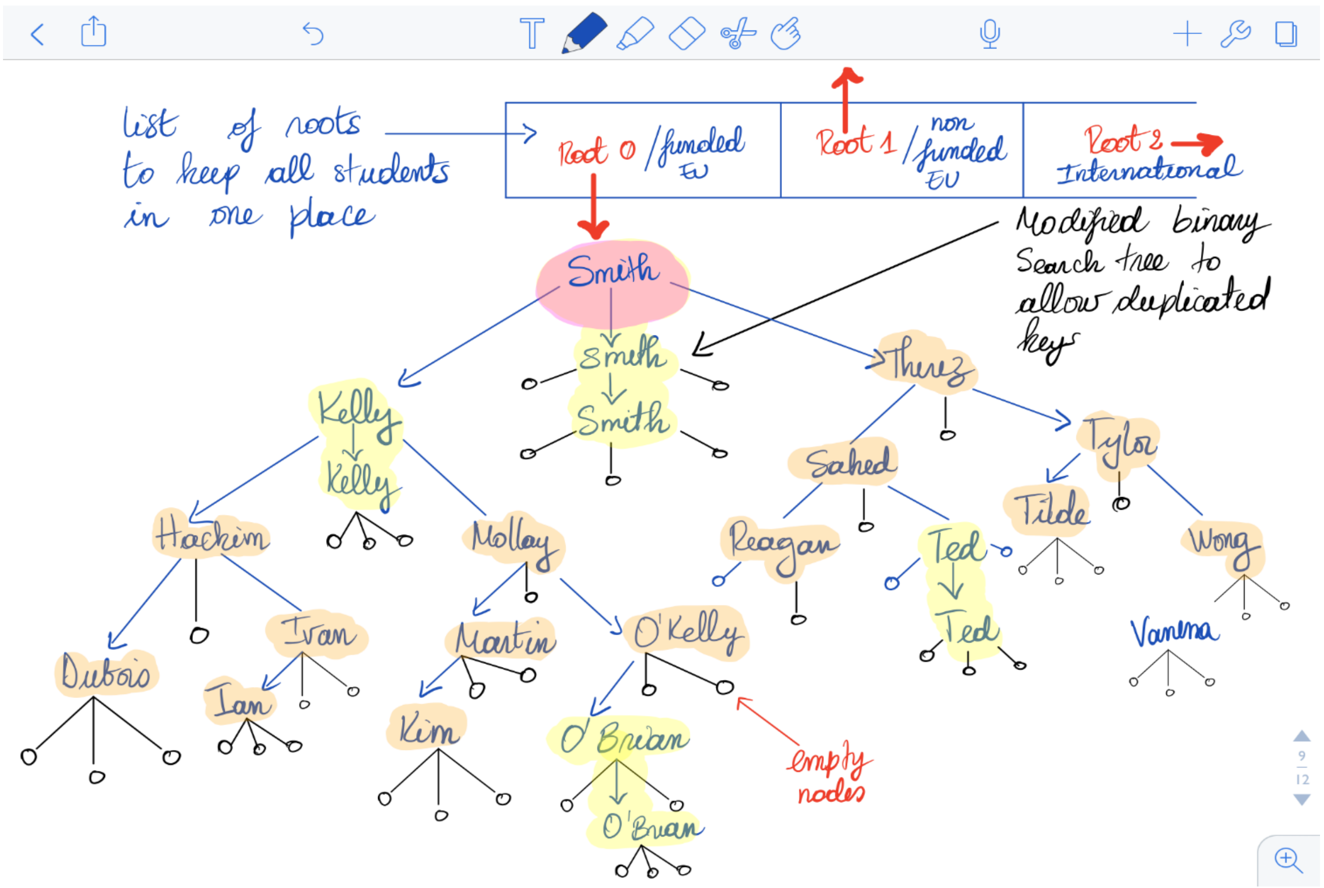


Figure 1: General representation of the Modified binary search tree data structure

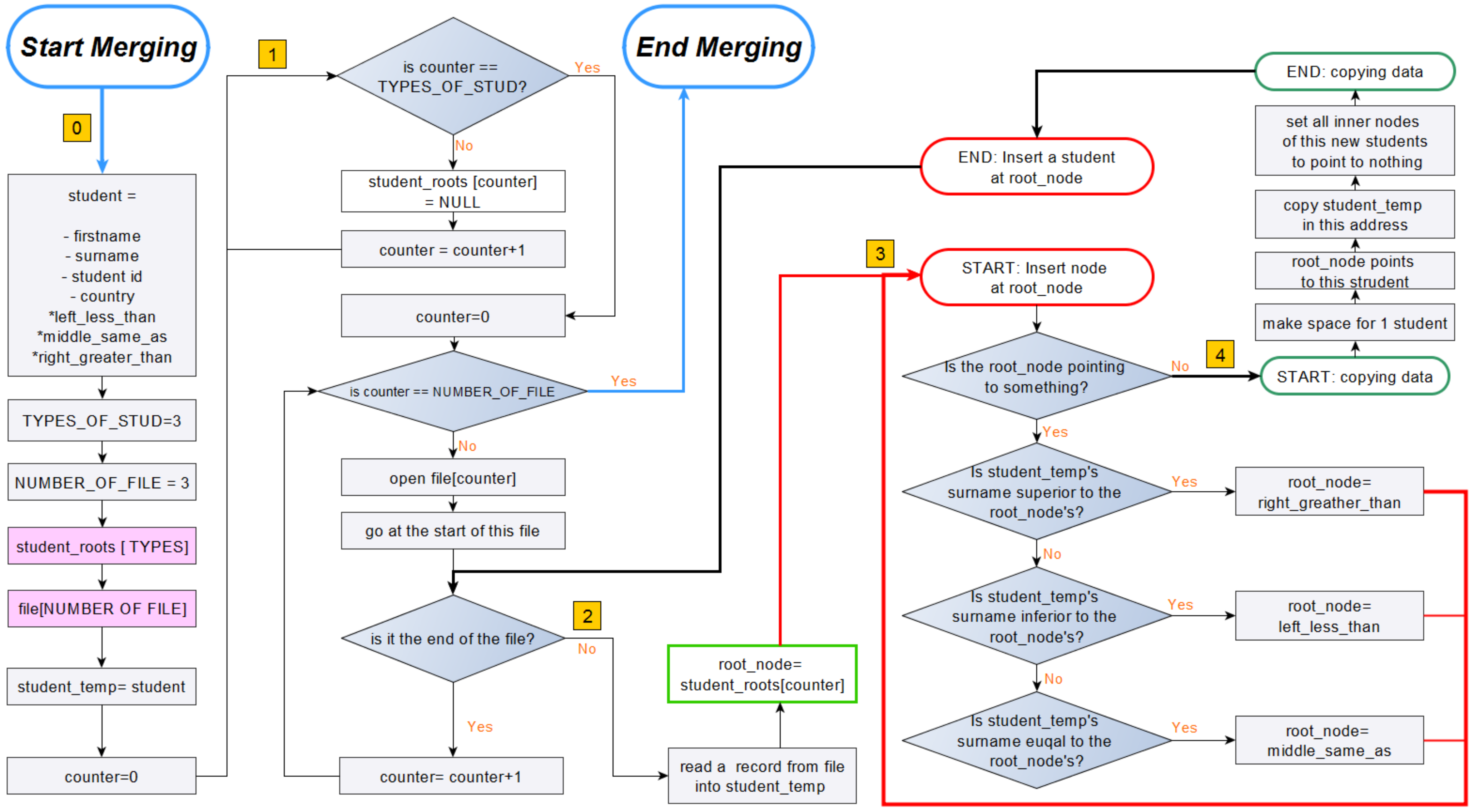
2. Loading, inserting and sorting.

On three separate files are **unsorted data** of every student type. e.g. Eu funded students inside the EU file etc.

The only fields that are read from the files are: the **first name, the surname, the student Id and the country of origin**. The three remaining fields that are the addresses to the student with a surname alphabetically greater, smaller, or equal, are added at insertion time.

Insertion and sorting is done simultaneously, **starting from the root node,** to the final location inside the tree.

3. Flowchart



Start Merging
0
student =
- firstname
- surname
- student id
- country
*left_less_than
*middle_same_as
*right_greater_than
TYPES_OF_STUD=3
NUMBER_OF_FILE = 3
student_roots [TYPES]
file[NUMBER OF FILE]
student_temp= student
counter=0
1
is counter == TYPES_OF_STUD?
Yes
No
student_roots [counter] = NULL
counter = counter+1
counter=0
is counter == NUMBER_OF_FILE
Yes
No
open file[counter]
go at the start of this file
is it the end of the file?
2
No
Yes
counter= counter+1
End Merging
root_node= student_roots[counter]
read a record from file into student_temp
END: Insert a student at root_node
3
START: Insert node at root_node
Is the root_node pointing to something?
No
4
Yes
Is student_temp's surname superior to the root_node's?
Yes
root_node= right_greather_than
No
Is student_temp's surname inferior to the root_node's?
Yes
root_node= left_less_than
No
Is student_temp's surname euqal to the root_node's?
Yes
root_node= middle_same_as
END: copying data
set all inner nodes of this new students to point to nothing
copy student_temp in this address
root_node points to this strudent
make space for 1 student
START: copying data

Figure 2: Flowchart representation of the insertion in the Modified Binary Search Tree

Flowchart textual description

Step 0: <u>**The merging program starts**</u>

- The student data type (**student**) is defined as shown in the flowchart
- The three type of students (**TYPES_OF_STD**) are: Eu funded students, Non-funded EU students and international students.
- The student's records are read from three different files (**NUMBER_OF_FILE**), one for each type respectively.
- A list (array **student_roots**) of three (TYPES_OF_STUD) root nodes is created to store addresses of the roots to each student type.
- The temporary student (**student_temp)**, will be used to store the data read from each file temporarily.

Step 1:

- All the addresses inside the array of roots are initialised to point to nothing (**NULL**).
- The counter is initialised to count the number of file that have been opened.
- When a file is open, it's first initialised at its start, then checked for emptiness.
- If the file is empty, or if the end of file has been reached, the next file is opened. If the number of file to open (NUMBER_OF_FILE) is reached, <u>**the merging program ends.**</u>

Step 2:

- Until the end of the file is reached, a student field is read from the file into the temporary student, (student_temp).
- The root from which the temporary student will be stored is initialised to the root of the array index of the current opened file (**student_roots [counter]**).
  <u>*Note*</u>: The files and the student types are correlated. i.e. data in file 0 is stored in root0 Modified Binary Search Tree, file 1 into root1 etc.
- Go to step 3 to Insert the temporary student's data inside the tree with initial address the current root node.

Step 3: **This step is done recursively until a leaf node is found.**
IF the current root is not empty,

- If the temporary student's surname is greater than the one pointed to by the current root node, the right pointer of the student pointed to by this current root node gets initialised as the root node: (root_node=right_greater_than)
- Else If the comparison results in less than, the left pointer of the student of this current root node gets initialised as the root node: (root_node=left_less_than)
- Otherwise, then the names must be equal, therefore the middle pointer gets initialised as the root node: (root_node=middle_same_as).
  Restart step 3.

Step 4: **The current root node is empty (step 3's base case).**

- A new student type is initialised. (In the implemented C program, a memory address will be allocated for one student).
- The node currently initialised as the root node is set to point at this address.

  - The data inside the temporary student is copied inside this address
  - The inner nodes for this student are marked as leaf (all initialised to NULL).

- Finish step 4 : adding the data
- Finish step 3: inserting at root node

Go check if it's the end of the file (Gate to step 2).

4. Big O

Inserting : (Modified) Binary Search Tree	
Worst case	**Average case**
O(n)	O (n log n)

Insertion in this Modified binary search tree data structure has the Big O (worst case scenario) of O(n).

This can happen when all the inserted nodes were **already sorted in ascending order**, with the root node being the smallest of them all.
Illustration:

Example: Considering inserting the following list into the tree:

List= *Alice, Bob, Cadet, Rebecca, Ted, Zoe*.
Size of the list = 6

Steps taken equal 6, which is the number of elements in the list.

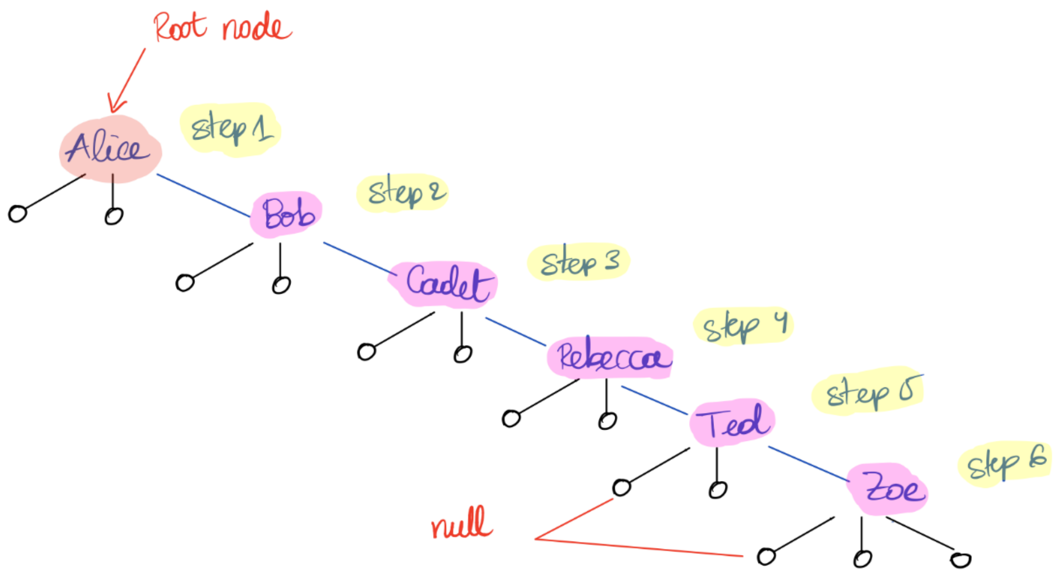


Figure 3: Worst case scenario Representation

Findings:

Initially unordered is most likely to work best for this Modified Binary Search tree, resulting in the average case scenario of **O(n log n).** This is because surnames that are only greater or lower than the current root node's, will cut the insertion time down to half at each recursive call, while nodes sharing the **same surname will still have to be inserted linearly from their respective parent node**. Thus O (n log n), where n is the number of students with the duplicated surnames.

Illustration: see *Figure 1*, page 3.

II. Searching for all international students.

1. Algorithm: Mapping.

To **search for all international student**, The first letter of the word tag given is mapped to the respective root node of all students of that type.
e.g In the student tags: **E**u, **N**on-eu and **I**nternational, The first letter of each type is used to redirect the printing to the respective node.

This is similar to what an **hash table (hash map) data structure** would do with a hash function. The main difference is the fact that this mapping is hardoded and not calculated beforehand, therefore it might not always be consistent on larger problem set especially if the keys have the same tags, in this a full hash table with a consistent hash function may be required to minimise the chances of collisions.

Overview:

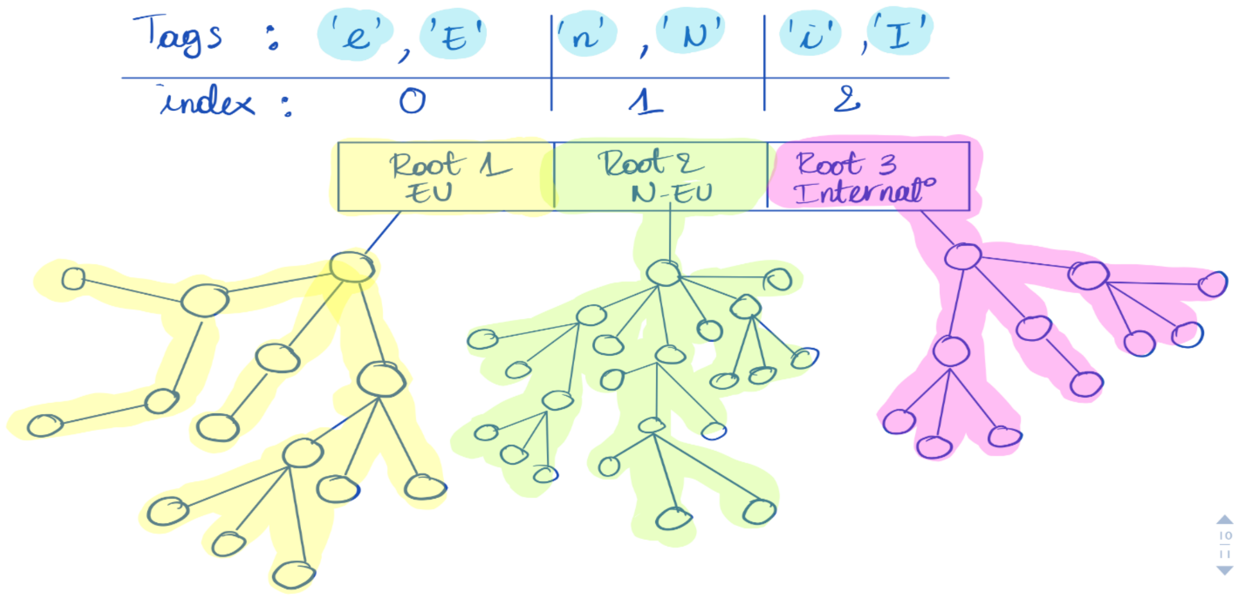


Figure 4 Mapping algorithm to find student type by tag.

2. Pseudocode

```
Search_all (key)

    IF the key's first letter is 'i', or the key's first letter is 'I' THEN
        return 2
    ELSE IF the key's first letter is 'n', or the key's first element is 'N' THEN
        return 1
    ELSE
        return 0
    End IF

End Search_all
```

Following this mapping algorithm, the program will return 2, every time the key is either **international, or International.**

3. Big O

Mapping algorithm		
Worst case	**Average case**	**Best case**
O (1)	Θ (1)	Ω (1)

This mapping algorithm does not have a Big O of O(n) as would a generic hash table in its worst case, this is due to the fact that the mapping is set and not calculated, which manually handles collision.

Moreover, considering this problem that only require **three mappings** this algorithm might work best compared to a hash table data structure which might have required further calculations and complexity inside the hash function.

Illustration: see *figure 4* page 7.

III. Searching for a student by surname.

4. Algorithm: (generic) Binary Search Tree.

To search for a student by surname, the modification done to the Binary search three matter less.
Only the first occurrence of the key's address is returned. All duplicates are dealt with on output.

Overview:

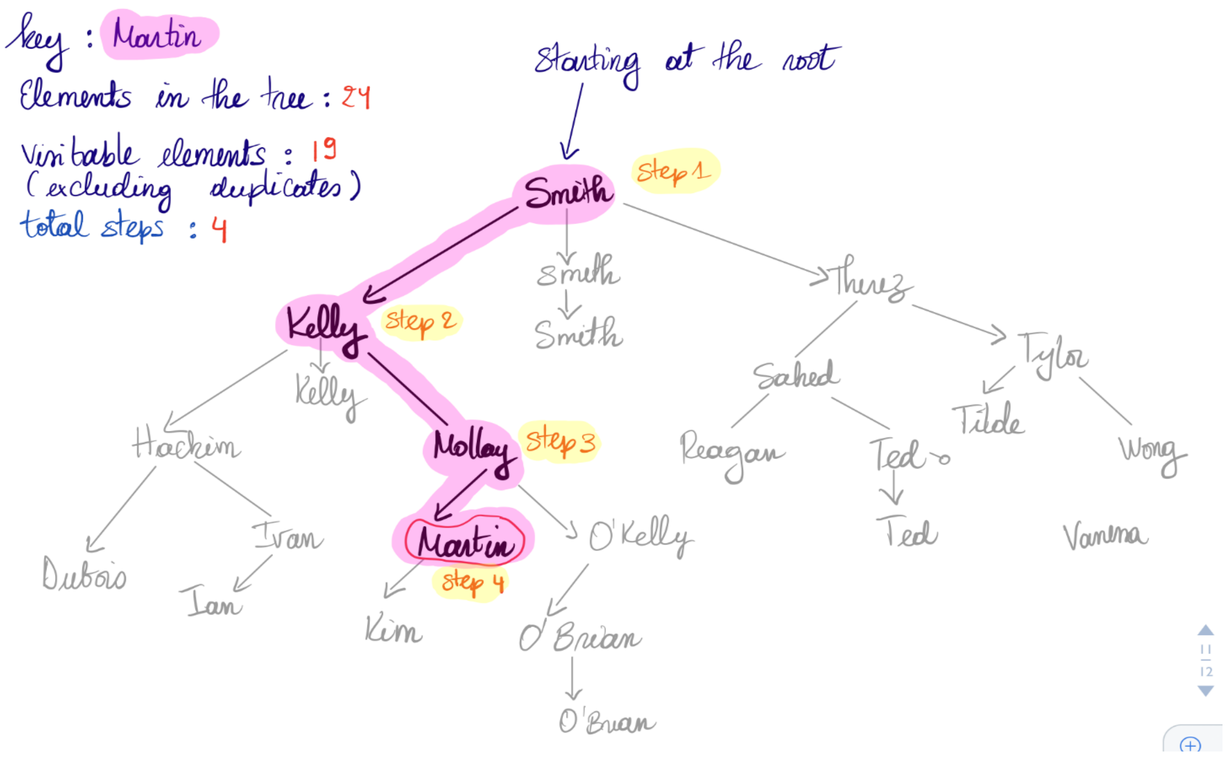


Figure 5: searching for a student by surname

2. Pseudocode

```
Search_student (surname, RootNode)

        IF (RootNode == NULL) THEN
                return NULL

        ELSE IF (RootNode's surname == surname) THEN
                return RootNode

        ELSE IF (RootNode's surname > surname) THEN
                return Search_student (surname, LowLeftRootNode)

        ELSE
                return Search_student (surname, GreaterRighttRootNode)

        End IF

End Search_ student
```

If a NULL node is encounted before finding the key, then the element is not present inside the three and the function return NULL.

Else if it is present at the current root node, this current root node is returned.

If it is **lower** than the present root node, the same key gets **searched to the left**, but this time with the root node being the one at the **left** of the current node.

If the key is **greater** than the current root node's tree, the same key gets searched to the right but this time with the root node being the one at the **right** of the current.

1. Big O

Searching: Binary Search Tree		
Worst case	**Average case**	**Best case**
O (n)	O (log n)	O(1)

Just like inserting, searching a key in a binary search tree can result in the big O of O(n) at if the tree has evolved into a linked-list like data structure at insertion time.

On average (data perfectly balanced or somewhat balanced in the tree), will result in the size of the elements to search being cut down to half on every next recursive call. See *Figure 5*, page 8. Where 5 steps are done when searching among 19 names.

The best-case scenario would be to search for **Smith** in *figure 5* on page 8. As it would only take one step, (hit the base case immediately).

IV. C implementation.

The C implementation contain the following:

1. **file_handling.h**: contains the implementation of the flowchart.
2. **find_and_print.h**: contains the implementation in part 2 and 3.
3. **student_truct_template.h** : contain the struct template of the student.
4. **main.c** : file to be compiled.

5. **student_function.h**: contains utility function that error checks the arguments passed on the command line, defines certain symbolic names etc.

6. Three files: ***funded_eu.csv, non_funded_eu.csv, international_students.csv***

Note : this program uses command line arguments. The option '**--help**' passed after the executable name instructs on how to enter them. The function display_help() is also called by default every time a wrong command is passed.

The third argument is **also designed not to be case sensitive** to facilitate the look up.

Commands	***Output***
Executable_name find Smith	*Will search and print Smith if found*
Executable_name find smith	*Will also search and Smith if found*
Executable_name find-all international	*Will print all the international student in sorted order*
Executable_name hello	*Will give out an error*
Executable_name	*Will also give out an error*
Executable_name --help	*Will Displays help*

References

Binary Search Tree [Online] / auth. Programing Paul. - 6 November 2013. - 10 April 2017. - https://youtu.be/sf_9w653xdE?list=PLTxllHdfUq4d-DE16EDkpeb8Z68DU7Z_Q.
C How To Program [Book] / auth. Paul Deitel Harvey Deitel. - New Jersey : Pearson, 2012. - 7th.
C IN A NUTSHELL [Book] / auth. Crawford Peter Printz and Tony. - Sebastopol : O'Reilly, 2005. - 1st.
Data Structure and algorithms with Python [Book] / auth. Kent D.Lee Steve Hubbar. - New York : Springer, 2015.
Datastrycture and Algoritms in Java [Book] / auth. Michael T. Goodrich Roberto Tamassia, Michael H. Goldwassier. - USA : Wiley, 2014. -