**UNIVERSITÀ DEGLI STUDI DI SALERNO**



**Project of Advanced Algorithm and Data Structures**

**a.y. 2019/2020**

**Project Documentation**

***Group 21***

***Professors: Realized by:***

prof. Auletta Vincenzo Antoniello Antonia

prof. Ferraioli Diodato Casale Teresa

Cerrone Angelica Maria

Cerino Mario

Palladino Amedeo

**Exercise 1**

For this exercise, the implementation of a B-Tree data structure and its relative Node structure was requested; it must be considered that such choices can affect performance both in terms of computational complexity and I/O complexity.

To solve the proposed problem, we used specific data structures, implemented by some of the classes belonging to the TdP Collection package. In particular, they are: Linked Binary Tree, Red Black Tree, Binary Search Tree, Doubly Linked Base and Positional List. The implementation of the B-tree structure has been carried out by adding a class of the same name, specifically created.

**Choice of B-Tree parameters**

The choice of the parameters of the B-Tree has fallen on: d(degree)=7, a=[(7-1)/2]=3, b=d=7 and is motivated by the following consideration: if a and b are too distant from each other, there is an increase in the complexity of node insertion operations. This complexity reduces the advantages of the data structure, such as balance (derived from the depth decrease).

The size of the node of the B-Tree is increased by the choice of a not low value for b; this leads to a lower complexity in I/O operations, already reduced in B-Tree a O(log n/log B) - with n=total number of elements and B=elements that can be stored in memory. This complexity is computed as the number of blocks transferred in each update/query operation; the advantage obtained depends, therefore, on the smaller amount of operations needed depending on the larger size of the blocks (hypothetically, the d elements belonging to the node and the d-1 references to the children fit compactly to the block, making B and d proportional).

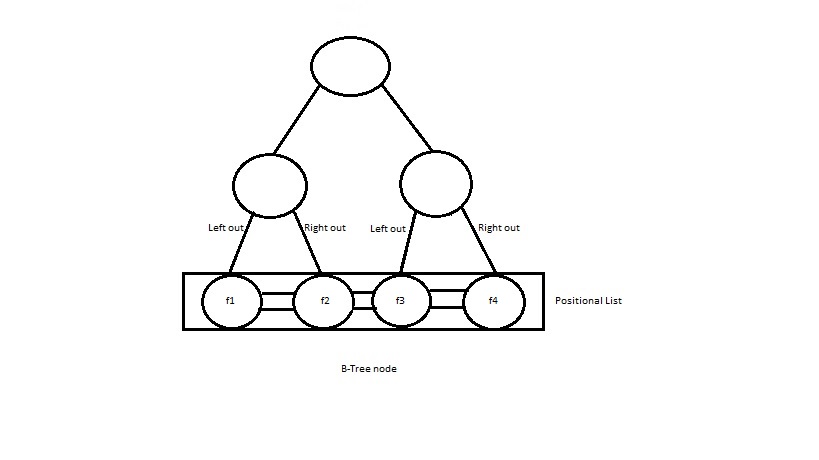
Despite this size would lead to an increase in the complexity of the internal operations of the node, the problem is compensated by the implementation choices made, concerning its structure.

**Node structure**

The choice about node structure fell on the Binary Search Tree, since the computational complexities of search and update operations correspond to O(log b); the operation of the median element search, whose default complexity is O(b), was reduced to O(1) by using a list as an auxiliary data structure.

It has been chosen to avoid the implementation through ordered vectors, since the operations of update would have been executed with computational complexity equal to O(b), in order to guarantee the ordering. In the same way, the option relative to the hash table has been avoided because the identification of the median would have been carried out with complexity equal to O(b) and it would not have been possible to easily manage the ordering of the elements.

The implementation of the auxiliary data structure has been carried out through a Positional List, which extends the Doubly Linked List class; the purpose of this choice is to maintain the references to the child nodes of the current one, facilitating the operations performed on the B-tree. Internal BST nodes, which have at least one None child, maintain a reference to an element of the list. A representation is provided by the following image, where f1, f2, f3 and f4 represent the references to the child nodes of the B-tree.



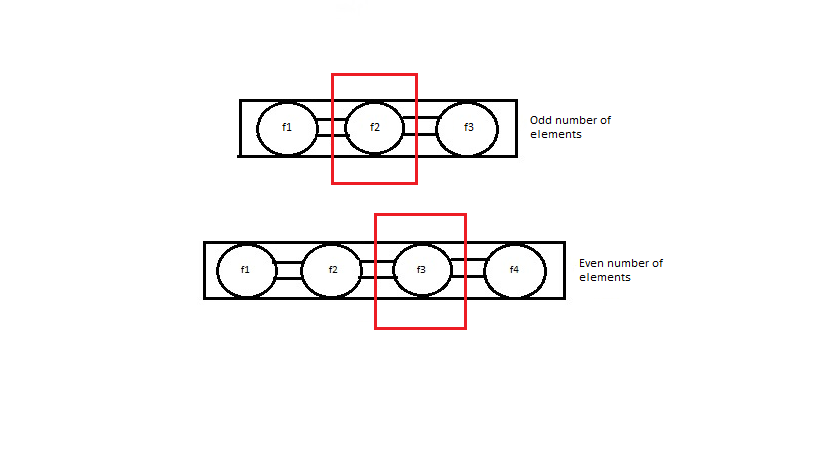
**Doubly Linked List**

Two attributes have been added to the class: median and medianKey, which represent the median of the list and the key of its parent node in the BST. Getter methods have also been provided.

**Half-median detection and update algorithm**

The middle node identifies:

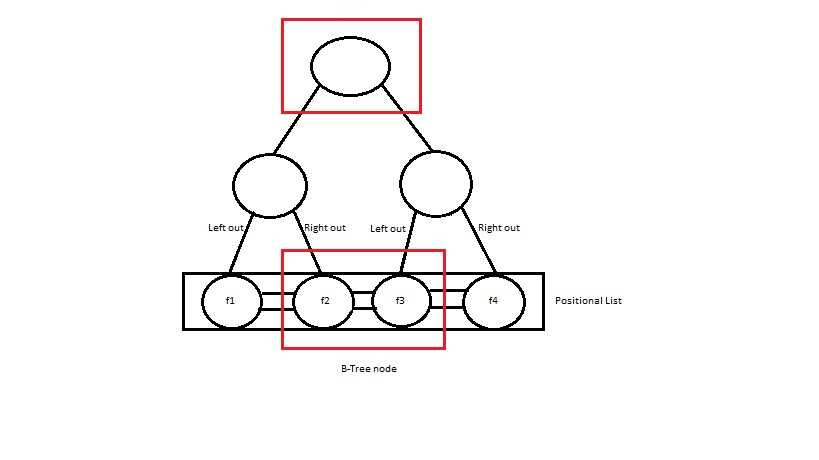
* the central element of the list, if it has an odd number of elements;
* the central node of the list taken in excess, in case the number of elements is even.



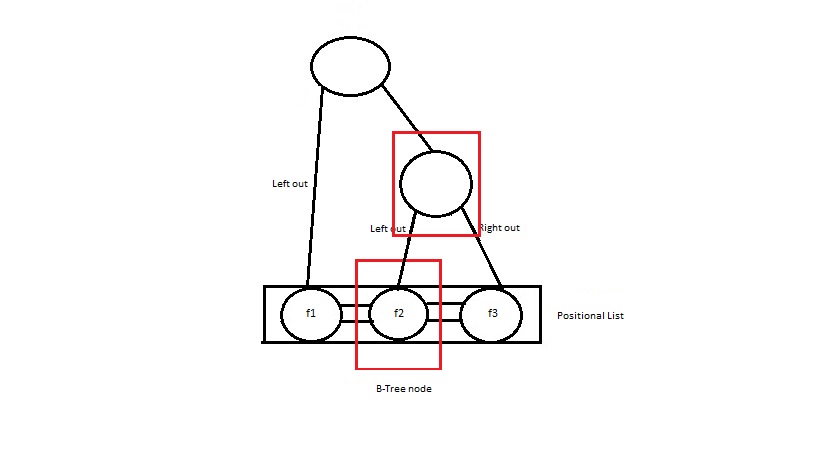
It is necessary to maintain a reference to it, in order to facilitate splitting operations and to identify in a constant time the central element of its own BST node.

This, in fact, corresponds to:

* the predecessor of the parent node of the median taken for excess, in case the two median elements of an even list have different parent nodes and the parent node doesn’t have any left child;

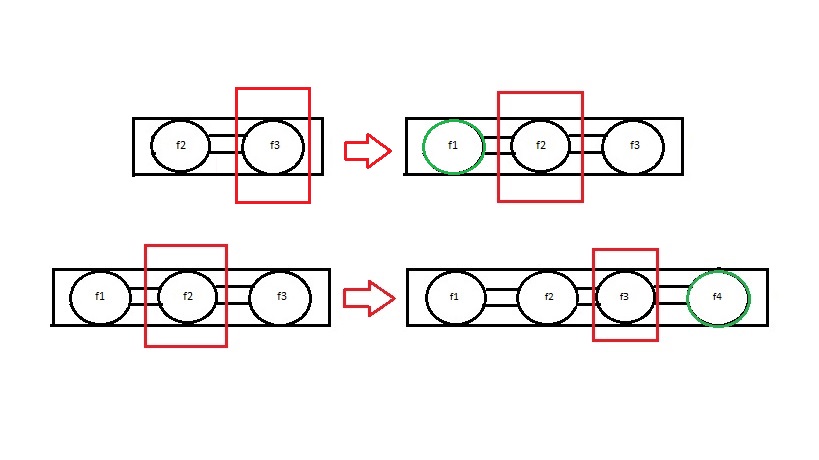


* the parent node of the middle item on the list, otherwise.

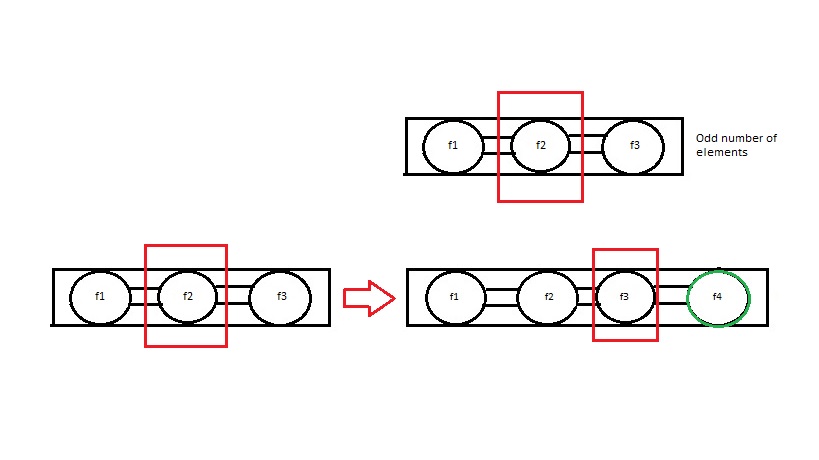


There are specific conditions for which an update is necessary.

1. ADDING AN ITEM TO THE LIST
   1. The middle is moved to the left if the addition is made to the left and the number of items in the list becomes odd.



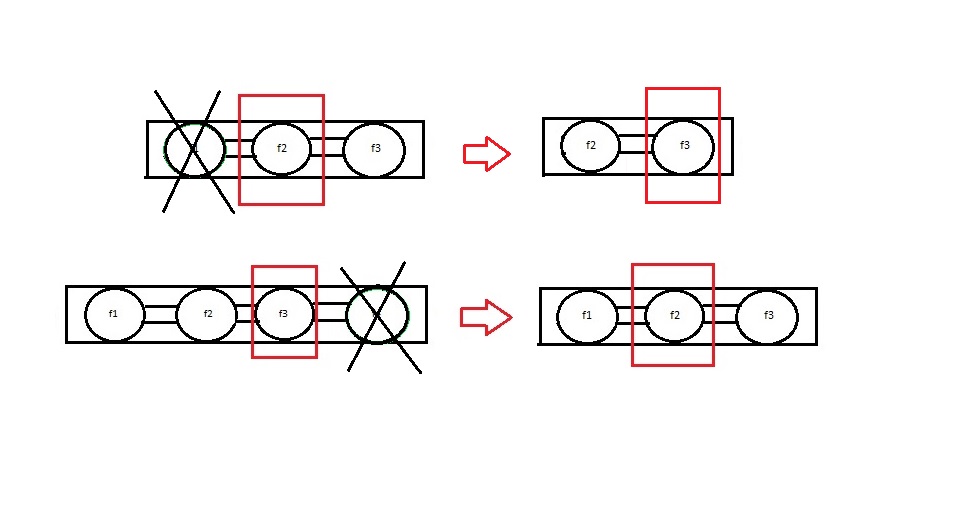
* 1. If the addition is made to the right, the median is moved to the right and the number of items in the list becomes even.



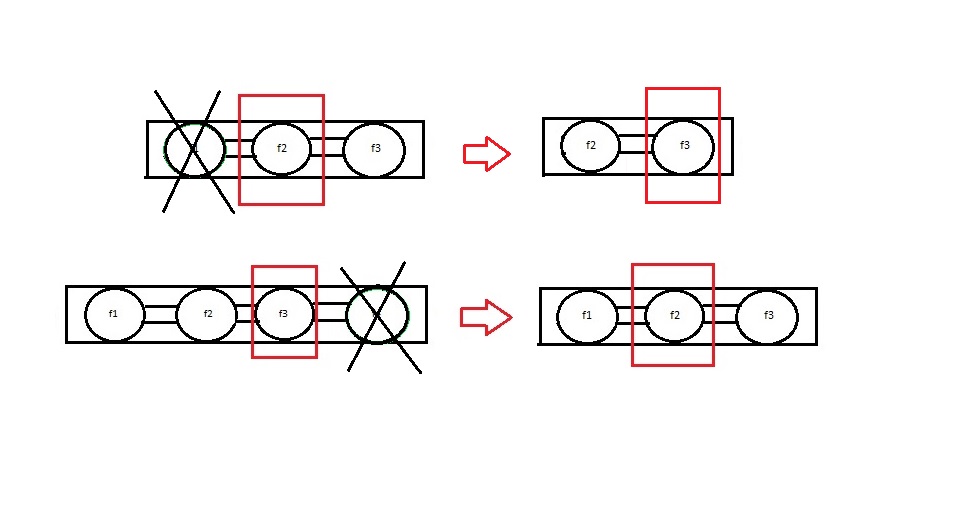
1. DELETION OF AN ITEM FROM THE LIST

2.2 NOT MEDIAN

2.2.1 The median is moved to the right, if the deletion is made to the left and if the number of items in the list is odd.

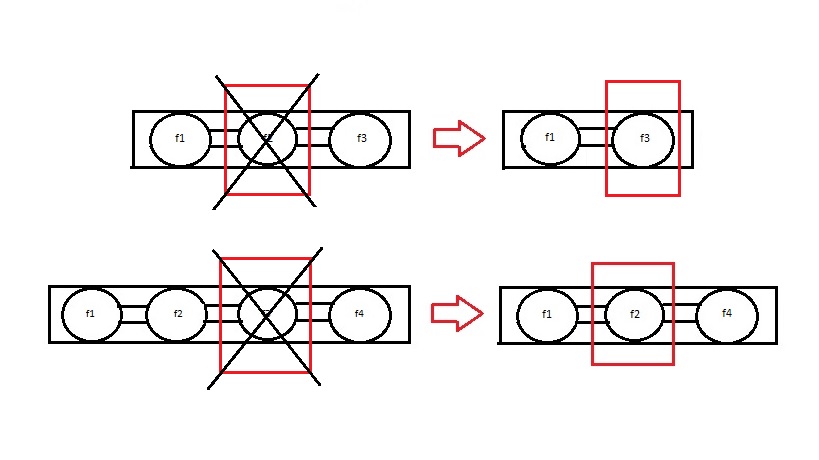


2.2.2 The middle is moved to the left, if the deletion is made to the right and if the number of items in the list is even.

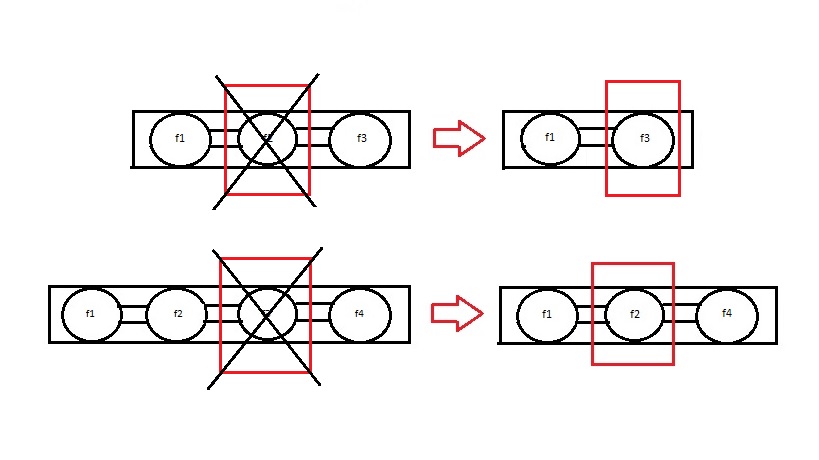


2.3 MEDIAN

2.3.1 The successor of the eliminated node becomes the new median, if the number of list items is odd.



2.3.2 The predecessor of the eliminated node becomes the new median, if the number of the items on the list is even.



The methods that have been specifically implemented in the DoublyLinkedList class are as follows:

* computeMedianAdd(self,nodeParent,newest)

The purpose of this method is to update the median element when a new node is added, based on the above conditions. The input parameters are: *newest*, which represents the node to be added to the list and *nodeParent*, which identifies the parent node (belonging to the BST) of *newest*.

The function initializes the median to *newest* when the length of the list is equal to 2 (and the first node has been added to the relative BST).

In other cases, the \_*medianKey* attribute is compared with *nodeParent* to see if the addition was made to the right or left of the median and len(self) is called to see if the list length is odd or even.

When the check is successful, \_*median* is set to the predecessor (or successor) of the current median and \_*medianKey* is set to the key of the predecessor (or successor) of its parent node in the BST.

* computeMedianRemove(self,nodeParent)

The operations performed by this method are the same as those described above, with the difference that:

* + this method is called when an item is removed from the list;
  + \_*median* and \_*medianKey* are set to None, in case of length equal to 2 (which would correspond to 0 after the update);
  + the checks and settings, made for updating \_*median*, are the same as the previous function but the conditions to be checked are different;
  + the checks are also made in case of deletion of the median element of the tree (condition in which the update of \_*median* and \_*medianKey* on the right or left depends on the length of the list).

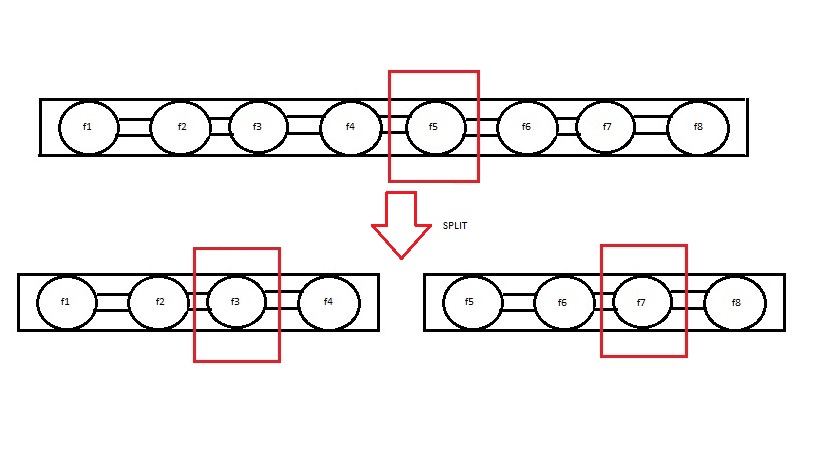
**Positional List**

This class extends the previous one and it identifies the data structure used to implement the auxiliary list to the BST.

The added methods are:

* \_computeMedianAdd and \_computeMedianRemove, which concern the computation of the median element and, therefore, they recall the homonymous methods of the extended class.
* \_splitMedian, which splits the auxiliary list, as a result of the split operation applied on the BST.

1. The first obtained list is set in this way: \_*header* coincides with the \_*header* of the initial list, \_*trailer* with the predecessor to the median element, and \_*median* with the predecessor to \_*trailer*. This last assignment has been made considering the specification b=7; in this case, in fact, the overflow generated by the addition of the eighth element in the list causes the sublist to have a length equal to 4 and the median can be set to the third node of the list.
2. In the second list: \_*header* coincides with \_*median* successor, \_*trailer* with the \_*trailer* of the original list and \_*median* with the predecessor of the last element for the same hypothesis mentioned above.



**Linked Binary Tree**

In order to implement the B-Tree efficiently, it was necessary to modify the implementation of the LinkedBinaryTree class, which represents the basic structure of the B-Tree node. In particular, a Positional Doubly Linked List has been added as a property, necessary to maintain the children of a B-Tree node. In addition, the implementation of the Node class has been changed so that each node keeps the *left\_out* and the *right\_out*, respectively the external left child (represented by a positional list node pointing to a B-Tree node, which will be the child of the linked binary tree node with the *left\_out* node) and the external right child. The tree has been constructed so that each leaf (node with left and right child None) has both *left\_out* and *right\_out*, while nodes with only one child have only *left\_out* (if the left child is None), or *right\_out* (if the right child is None).

To keep these properties up to date, it was necessary to change some methods in the LinkedBinaryTree class and a node with one of some methods:

* \_add\_left(p, e, left\_out=None)

Adds a new position to the left of position *p* with the *e* element. The new feature is the *left\_out* parameter, which will update the *left\_out* of the new tree node, while its *right\_out* will be the *left\_out* of the parent, which is set to None.

* \_add\_right(p, e, right\_out=None)

Similar behavior to \_add\_left()

* \_delete(p, left=True)

When deleting a node, depending on whether you delete (left or right), it must pass the *right\_out* or *left\_out* of the node to the parent, and delete the other list reference by deleting the node from the list.

**Binary Search Tree**

If the internal node structure of a B-Tree is a LinkedBinaryTree, we decided to use a BinarySearchTree to extend its functionality, in order to obtain methods for the elements research in the tree. The BinarySearchTree in the TdPCollection package is implemented to be used as a TreeMap. So if you search for a *k* key, and it is not present in the tree, a KeyError exception raises. What we want is different, because if a key is not present in a BST in a B-Tree node, we would like to search for it in the other B-Tree nodes, which are also BST. To do this, we changed the get\_item implementation to return the predecessor or successor of the element to search for, so that we can continue to search for it in external nodes by following the *left\_out* and right*\_out* references of the node returned by the get\_item. An add method has also been implemented to properly set the *left\_out* and *right\_out* references of the new nodes added to the tree as references of its relative list.

**Red Black Tree**

A RB-Tree implementation for the internal nodes of the B-Tree structure has been chosen. This choice depends on the necessity to keep the tree balanced (to represent the internal elements of a B-Tree node and to ensure the update and search operations in O(log b). The RB-Tree structure is more efficient than its AVLTree ancestor because it does not need to go up to the root in some situations (to guarantee the balancing property), as it is necessary in some situations in an AVLTree. However, the difficulty of the balanced tree use depends on splitting and merging operations, which need an algorithm to split or merge two balanced trees and keep them balanced. This algorithm should be efficient, otherwise we would lose all the computational advantages we have achieved. In particular we have decided to use two algorithms (split and catenate) that are executed in constant amortized time, because they only change links between the nodes of two trees. These algorithms were taken from an article by Ron Wein, Efficient Implementation of Red-Black Trees with Split and Catenate Operations.

**Implementing the Catenation Operation[[1]](#footnote-2)**

Tarjan [Tar83a], who gives the details of the catenate and split procedures for red-black trees, uses a variant of the data structure that stores data elements only in the tree leaves, where internal nodes only contain keys. Using this representation, it is possible to give a simple representation of the catenate operation. We are given two sets S1 and S2, represented as red-black trees T1 and T2, respectively, such that for each x ∈ S1 and y ∈ S2 we have x ≤ y. Assume, without loss of generality, that black height(T1) ≤ black height(T2). We perform the following procedure (see Figure 2 for an illustration).

1. Go down from root(T2) along the leftmost path in the tree, until reaching a black node ρ such that black height(ρ) = black height(T1).
2. Let π ←− parent(ρ).
3. Make root(T1) and ρ siblings by creating a new node ν, and setting left(ν) ←− root(T1), right(ν) ←− ρ and color(ν) ←− Red.
4. If π = Null, set root(T1) ←− ν. Otherwise, set parent(ν) ←− π and root(T1) ←− root(T2).
5. Perform the insertion fix-up procedure from the node ν. 6. Mark T2 as empty by setting root(T2) ←− Null.

The result of this procedure is a consolidated set S which is represented by the catenated red-black tree T1. The running time of the catenation procedure is clearly dominated by Step (1) and is therefore O(log n2 − log n1) = O(log n2 n2 ), where n1 = |S1| and n2 = |S2|. Note however that if the node ρ is known in advance, the rest of the steps can be carried out in constant amortized time.

**Implementing the Split Operation**

1. Let T1 ←− ∅ and T2←− ∅.
2. Let κ ←− root(T).
3. While κ 6= Null, do:
   * If x < element(κ), then: Catenate the subtree rooted at right(κ) to T2 using the pivot ν2. Let ν2 ←− κ. Let κ ←− left(κ).
   * Else if x > element(κ), then: Catenate the subtree rooted at left(κ) to T1 using the pivot ν1. Let ν1 ←− κ. Let κ ←− right(κ)
   * Otherwise (x = element(κ)): Catenate the subtree rooted at left(κ) to T1 using the pivot ν1, and set ν1 ←− Null Catenate the subtree rooted at right(κ) to T2 using the pivot ν2, and set ν2 ←− Null.

Insert κ as the minimum node of T2. End the split procedure.

1. If ν1 6= Null, insert it as the maximum node of T1. If ν2 6= Null, insert it as the minimum node of T2.

To run these algorithms, we need to maintain the black height of the tree nodes; this is done efficiently because the property is added to the nodes which modify their own black\_height and the ancestors' one, at re-colouring time. These operations' computational cost corresponds to O(logb), with the update\_black\_height(p) method.

Then we used two methods, split and chain, which perform the algorithms shown above.

**External structure – B-Tree**

The class was created to implement the required external data structure. In \_\_init\_\_, the default parameters (*a,b*) have been set, depending on the respective degree. The implemented methods are associated with search and update operations. The innested Node class is initialized to a RedBlackTree, the structure with which the internal nodes have been implemented, and the getter methods (related to attributes and internal elements) are provided.

* search(tree\_node,k)

The search for the *k*-key node proceeds for all the elements of the B-Tree, leading back to a search within a Red Black Tree. The variable *p*, of type Position, is initialized at the root of the tree currently examined. Until the *k* key is found, the search proceeds for the right or left subtree, based on the comparison between *k* and *p.key()*. Recursive calls to the method are made on the child nodes of *p*, until a leaf node is reached.

* delete(k)

The elimination of the *k*-key node is carried out under different conditions:

* + if the node is a leaf, the call to the function \_predecessor\_external\_subtree allows to find the position of the *k*-key node in the outer subtree to *p*;
  + if there is a left child of *p* in the B-Tree, the search proceeds in that direction;
  + if there is a left or right child of *p* within the same node, the Position of the last element of the right/left subtree is returned.

After properly setting the *new\_p* and *new\_tree\_node* variables, the references of the node to be deleted are transferred to *p* and, after updating the tree size, the verification of a possible underflow is done.

* \_predecessor\_external\_subtree(p,tree\_node,k)

The method detects in the outer *p* sub-tree the position of the node with a *k* key. In particular, if *p* is not a leaf or has external nodes in the B-Tree, this search is carried out in the right or left subtree (depending on the position of its children). The latter check depends on the setting of the *\_right\_out* and *\_left\_out* attributes, specifically designed for operations between nodes in the B-Tree.

* add(k)

The addition of a node within the B-Tree proceeds as follows:

* + if the tree has no element, a node and its inner tree are created and the *k*-key element is added;
  + otherwise, if the search for a *k*-key node fails (the element is not already present), it is added to *tree\_node* (obtained in output from the search function call).
* check\_overflow(tree\_node)/check\_underflow(tree\_node)

They verify the presence or absence of an overflow/underflow case, by a simple comparison between the size of *tree\_node* and *b* (maximum expected value, before the verification of an overflow) and *a* (minimum expected value, before the verification of an underflow). If the necessary conditions are not verified, the split or resolve\_underflow functions are called.

* \_immediate\_siblings(tree\_node)

Returns the siblings of the node identified by tree\_node. If the node is not the first element in the parent node's child list, *t\_node\_before* is set to all adjacent elements to its left; opposite reasoning for *t\_node\_after*.

* resolve\_underflow (tree,node)

The method solves an underflow condition and it performs a check on elements adjacent to tree*\_node* by having the transfer method called on the right or left subtree if the number of elements in the relevant list is more than *a*. Otherwise, the fusion function is called on one of these sub-trees.

* transfer(tree\_node, tree\_transfer\_node, before=True)

The transfer function is called up in order to resolve an underflow condition by exploiting the adjacent sub-trees. In the case of a left subtree, the last node is called by setting the *p\_transfer* variable. For the right subtree, the variable is set to its first element. Finally, the parent node *p\_parent* is added to the underflowed subtree, the Position *p\_transfer* is replaced by *p\_parent* and deleted from the subtree adjacent to tree\_node.

* fusion(tree\_node,tree\_fusion\_node,left=True)

The method performs fusion between adjacent nodes in order to resolve underflow conditions. After creating a Red Black Tree as a support structure, the catenate function is called between the node that caused the overflow and the adjacent left subtree (if *left*=True). Otherwise, the simple addition of the parent node to the adjacent subtree is implemented; finally, this node is removed from the *u* tree.

* split(tree\_node)

Based on the median set in the *tree\_node* list, the split function of the Red Black Tree class is recalled. The latter returns the trees obtained from the operation and the method proceeds by making a distinction between 'root node' cases or ‘not root node’ of the B-Tree. In the first case, in fact, an auxiliary tree is adopted in order to create a new structure with the median as root; in both cases, however, the method proceeds to set the references:

* + to the child nodes (and, automatically, to their respective sub-trees), with an update of *left/right\_out*, *parent* and *list\_parent* attributes;
  + to the new root by calling the function update\_children on its child nodes.
* \_update\_children(node)

This method supports the split method and performs the update of the parent attribute of the children of node.

**Exercise 2**

The exercise proposes to develop a scheduler for a CPU, so that it could run a bunch of *jobs* within a **waiting line**. Each *job* is characterized by its *priority*, its *name*, its *length* and the *waiting time* that defines the number of time slice elapsed since its insertion before it is executed. These elements are defined by a range of values, and some elements of a job in the **waiting line**, once an interval of time equal to x time slice has elapsed, may change (in particular, the *priority* is increased and the *waiting time* of each *job* is reset to zero).

Before talking about the resolution of the exercise, some assumptions have been made: as suggested by the exercise, each iteration of the cycle that allows the execution of the operations corresponds to a time slice for the CPU, and it is assumed that, once a job is taken to be executed, it cannot be interrupted to execute other jobs (therefore the CPU is considered as a non-preemptive resource); moreover, it has been assumed to consider 5 time slice as the time interval after which the operations of increasing the priority and resetting the waiting line jobs will take place.

To solve the proposed problem, we’ve started from the definition of the Priority Queue Base, Heap Priority Queue and Adaptable Heap Priority Queue classes, made available by the text book “Data Structures and Algorithms in Python” by the authors Goodrich, Tamassia and Goldwasser, and to these classes the following modifications have been made:

* in the code that defines the Priority Queue Base, the class nested Item has been renamed to \_Job, its constructor method has been modified (with the addition of some specific attributes, such as priority, length and waiting\_time), the method for comparing the various objects belonging to the class, and the method for displaying the object’s attributes;
* no changes have been made to the code defining the Heap Priority Queue, except that the modified Priority Queue Base class has been imported;
* in the code that defines the Adaptable Heap Priority Queue, the reset() method has been added, which allows to perform the operations of incrementing and resetting the respective priority and waiting\_time attributes of each *job*, and it’s invoked once the previously assumed 5 time slice interval has ended.

Since several changes have been made, it was preferred to modify the known codes and define the files for the individual classes:

* The file ‘waiting\_line\_base’ contains the nested class \_Job and the structure of a Priority Queue Base, renamed Waiting Line Base;
* The file ‘heap\_waiting\_line’ contains the structure of a Heap Priority Queue, renamed Heap Waiting Line;
* The file ‘adaptable\_waiting\_line’ includes the code of the Adaptable Heap Priority Queue class, specifically modified and renamed Adaptable Heap Waiting Line.

**Introduction to the algorithm:** The algorithm is divided into the following *phases*: it starts with the request to the user to insert the first *job* inside the *waiting line*; then, since there is at least one job in the waiting line, this *job* is chosen by the *scheduler* to be executed by the CPU. After that, a new request is made to insert a new *job* and, consequently, to execute a new *job*. As a prerequisite for a correct execution of the algorithm, it has been established to ask the user to necessarily enter the first *job* he wants to execute and, only once the user has correctly entered the first *job*, the algorithm goes to the next step. Every time a *job* is inserted, it will be placed inside a *waiting line*, which will have the structure of the Adaptable Heap Waiting Line mentioned before. As required by the exercise specifications, *jobs* are entered only if the *priority* and *length* attribute values are within the specified ranges (i.e. if the *priority* is between -20 and 19 inclusive, and if the *length* is between 1 and 100 inclusive). Once the insertion and execution phases of the first *job* have been performed, the algorithm is iterative, so it enters in a cycle that allows to perform the same operations until the algorithm execution is finished. The algorithm has been created in order to understand whether or not a *job* is executed in a given time slice, communicated to the user by displaying a message on the screen.

**Initialization:** The algorithm starts with the definition of the *waiting line* object, which will assume the structure of the Adaptable Heap Waiting Line object, and the time slice variable which will be initialized at 0. Once a *job* has been inserted, a *\_Job* object is initialized with the values inserted in input by the user and then, when the first *job* is executed, the *scheduler* variable is initialized assuming the structure of an array containing the information of the *job* chosen to be executed by the CPU, and this array will be modified every time a *job* is chosen to be executed. As said before, every 5 time slice is called the reset() method, belonging to the *waiting line* object class, (so to initialize the *waiting time* to 0 and increase the *priority* of each *job* by 1) and only afterwards the time slice variable is initialized again at 0, which is essential to call the reset() method correctly again.

**Visit of the adaptable heap priority queue:** The execution of the developed algorithm allows to insert new jobs into an adaptable heap priority queue, the structure of the *waiting line* object, which considers the increasing order of the *priority* attribute as a sorting criterion within the queue. Once the first *job* is inserted, it’s determined that the inserted *job* is the *job* to execute by checking that the *waiting line* is not empty. Once the *scheduler* has chosen the first *job* to execute, the algorithm asks to the user if he wants to insert a new *job* or not. If he says ‘yes’, the user will enter a new *job* if the information of the new *job* complies with the reference ranges, otherwise, the user has to try again the insertion.

The algorithm goes on to the next step, which checks and selects a new *job* to run, if the user doesn’t want to enter a new *job* or immediately after entering a new *job*. This operation will be performed if there are jobs in the *waiting line* and if the CPU has finished running the current *job*, determining if the *length* of the current job is 0. The choice of the new *job* to execute is defined according to the following comparison criteria (defined in the \_\_lt\_\_() method of the *\_Job* class, which is plugged into the Waiting Line Base class): a first comparison is made between the priorities of all the *jobs* in the *waiting line* and the job with the *highest priority* is chosen (i.e. the job with *the lowest value of priority* of all the jobs); with the same *priority* between two or more jobs, a comparison is made between the *lengths* of each job and the one with the *greatest length* is chosen and, with the same *lengths* between the jobs, a comparison is made between the *waiting time* values of each job and the one with the *greatest waiting time* is chosen.

If a job is chosen to execute, the user will display the message “Add new job *name* with priority *priority* and length *length*”, otherwise if no new job has been chosen because the CPU is already executing one, then the user will display the message “Current job: *name* with priority *priority* and length *length*”, while if no job has been chosen because the waiting line is empty, the user will display the message “No new *job* in this time slice”.

Once this is done, the time slice is increased and, before the request to insert a new job in the waiting line, the *length* of the running job is decreased and the number of elapsed time slice is checked in order to call the reset() function as said before.

**Exercise 3**

For this exercise, it was required to design a DFS visit to a graph. It was also requested that the implementation of this visit should be iterative and without the use of any auxiliary data structure.

In order to solve the proposed problem, changes have been made to the **Vertex** class, the only one it was allowed to be modified. Starting from the already defined **Graph** class with adjacency map, made available by the codes in the text book “Data Structures and Algorithms in Python” by the authors Goodrich, Tamassia and Goldwasser, the constructor method within the nested class **Vertex** has been modified, with the inclusion of some attributes, and the insertion of get and set methods for these attributes. In the original class, only the **\_element** attribute existed within the constructor. For the purposes of the algorithm design, three more attributes have been added: **\_visited**, **\_discoverer** e **\_adj\_vertices**.

* The **\_visited** attribute is a Boolean variable, that is set to **False** by default, and indicates whether or not the vertex has been visited during the execution of the designed DFS algorithm.
* The **\_discoverer** attribute is used to store the vertex that discovered it during the execution of the designed DFS algorithm.
* The **\_adj\_vertices** attribute is an integer variable used to store the number of adjacent vertices.

Since the attributes are declared private within the class, it has therefore been necessary to add getters and setters to make these attributes accessible, as the information stored by them is manipulated during the execution of the designed DFS algorithm.

**Introduction to the method:** The designed method takes as input the graph on which the DFS visit should be performed and the vertex from which it should start. As required by the specifications of the exercise, the algorithm is iterative and does not use any auxiliary data structure (unlike the original one, where the execution is assisted by the use of a stack for the storage of the vertices that will be visited). The output of the method consists of a dictionary, in which are included the edges that are traversed for the exploration of the vertices of the graph under examination, in particular the vertices that are explored for the first time. The dictionary returned by the method represents a forest in which the starting vertex is the root.

**Initialization:** The designed method starts with the initialization of the dictionary, that will be the return value, and all the vertices of the graph. The latter, required by the fact that the algorithm can be called more than once, involves setting the **\_visited**, **\_discoverer** and **\_adj\_vertices** attributes (mentioned and described above):

* the **\_visited** attribute is set to **False**, to make all the vertices of the graph visitable;
* the **\_discoverer** attribute is set to **None**. In particular for the vertex **vertex**, where the DFS visit of the graph starts, the value of this attribute will not be modified;
* the **\_adj\_vertices** attribute is set to the value returned by the **degree** method of the **Graph** class, called on the graph for that particular vertex being examined. In the case of a directed graph, this method returns the number of adjacent edges of the vertex, hence the number of adjacent vertices that can be reached and visited. In the case of an undirected graph, instead, this method returns the number of all the incident edges of the vertex (there is no distinction between outgoing and incoming edges), hence the number of adjacent vertices that can be reached and visited.

**Visit of the graph:** Once the initialization phase is completed, the actual execution of the designed DFS algorithm can start. As a first step, the **\_visited** attribute of the vertex **vertex** from which the visit starts, is set to **True**. A Boolean variable called **stop** is also set to **False**, for the handling of the **while** loop that’s used. In the loop takes place the exploration of the vertices. The **\_adj\_vertices** attribute is verified to make sure that the starting vertex has incident edges (for a directed graph, the check is about the outgoing edges): if the checking is successful, the exploration of the vertexes continues; if not, other operations are performed, described in details below.

* When the **\_adj\_vertices** attribute testing is positive, the **incident\_edges** method of the **Graph** class is then called to obtain the list of incident edges of the vertex **vertex** under consideration. The idea behind the designed algorithm is to examine the first edge returned by the **incident\_edges** method and determine the opposite vertex to it (the opposite vertex is obtained thanks to the **opposite** method of the **Graph** class):
* if this vertex has not yet been visited (this is where the **\_visited** attribute of the vertex is needed), then the following operations are performed:
* the vertex will be marked as visited (from project requirements, it is not possible to use any auxiliary data structure, so to allow the correct execution of the algorithm the code relies on the storage of this information directly inside the vertex);
* the edge that allowed the discovery of the vertex is stored in the dictionary that will be returned by the DFS method;
* the number of adjacent vertices that needs to be examine/visit by the vertex **vertex** is re-set again by invoking the **degree** method of the **Graph** class (this operation is fundamental to allow the correct execution of the algorithm and therefore the visit of all the reachable vertices);
* the vertex **vertex**, which allowed the exploration of its unvisited opposite, is stored within the **\_discoverer** attribute of the opposite vertex;
* the “new” vertex **vertex** will be the one that has been determined as the opposite of the edge under examination.

At this point, the break command is used to interrupt the scrolling of the list of edges obtained from the call to the **incident\_edges** method, made on the “old” vertex **vertex**. Since the while loop is still running, then the call to the **incident\_edges** method will be repeated, but this time for the “new” vertex **vertex**, as the check on the **\_visited** attribute of the opposite vertex.

* When the checking on the **\_visited** attribute of the vertex fails (and so it turns out that the opposite vertex has already been visited), then the number of adjacent vertices is decreased using the set and get methods of the **Vertex** class for the **\_adj\_vertices** attribute: if with this decrease, the number will be zero then it is necessary to go back to the vertex that discovered the one under examination and this “string rolling” is done until meeting a discoverer vertex that still has adjacent vertices to visit. If the **\_discoverer** attribute is **None** (because the starting vertex has been reached), then the while loop is terminated.
* During the algorithm design, there have been taken into account some peculiar cases (even extreme ones) in which the vertex under examination could not have incident edges or outgoing edges, the latter always in the case of directed graphs. If the reached vertex or the starting vertex has no adjacent vertices, then the new vertex **vertex** that will be examined will be its discoverer, when this attribute is not **None**; otherwise, the while loop is stopped by setting to **True** the Boolean variable **stop**.

**Exercise 4**

For this exercise, it was required the design and implementation of an algorithm to install the BaceFook software on the minimum number of users of its social network, so that in each pair of friends, at least one of them has the software. The specification is that this algorithm is implemented using Dynamic Programming, assuming the network is a tree.

To solve the proposed problem, a **GeneralTree** class has been created by extending the **Graph** class, located in the “TdP Collection” folder that contains some of the codes in the text book “Data Structures and Algorithms in Python” by the authors Goodrich, Tamassia and Goldwasser. For the purposes of the algorithm design, two attributes have been added, within the constructor, and several methods to adapt the **Graph** class to manage the relationships between nodes in a tree. In the **GeneralTree** class constructor, the **Graph** class constructor is called and the **\_root** and **\_parent** attributes are added.

* The \_root attribute is a variable that is set to None by default, and specifies which node represents the tree root.
* The \_parent attribute is a dictionary used to store parent nodes within the tree, for each of the nodes in the tree.

Since attributes are declared private within the class, it was therefore necessary to add methods to allow access to these attributes. In particular:

* The root(self) method returns the tree node representing the root.
* The parent(self,v) method returns the parent node of node v passed as a parameter.

The other methods that have been added are the following:

* The is\_root(self,v) method returns a Boolean value, which is set to True if the node v passed as a parameter represents the tree root; otherwise is set to False.
* The is\_leaf(self,v) method returns a Boolean value, which is set to True if the node v passed as a parameter has no nodes as child, and is therefore a leaf; otherwise is set to False.
* The add\_root(self,x) method allows to insert the root node inside the tree by calling the insert\_vertex(x) method of the extended Graph class. The \_root attribute of the tree is set to root, i.e. the value returned by the insert\_vertex method, while the \_parent dictionary, at the key root, is set to None. When it’s done, the method returns the inserted root node.
* The add\_child(self,v,x) method allows to insert a generic node inside the tree, again by calling the insert\_vertex(x) method of the extended Graph class. The link between the newly inserted node and the node v, passed as a parameter, is added by calling the insert\_edge(v,child) method of the extended Graph class, where child is the value returned by the insert\_vertex method. The \_parent dictionary, at the key child, is set to node v. At the end, the method returns the inserted child node.
* The children(self,v) method allows to obtain the children of a certain node v passed as a parameter. In this method, calls are made to the incident\_edges(v) and opposite(v) methods of the extended Graph class: the first allows to obtain the edges, and so the links, between the node v and its children; the second allows to obtain the child of a certain link. The method then returns the children, one at a time using the yield command.

The algorithm designed and implemented involves the use of two methods: min\_nodes\_install and find\_solution. The design has been done taking into account the specification given by the exercise to reach a computation complexity of O(n).

**min\_nodes\_install method:** The method requires a tree tree as a parameter. As a preliminary operation, the recursive DFS algorithm is called using the DFS\_complete method, which returns a dictionary forest with all the nodes of the connected component of the tree tree on which the algorithm was called. The keys of this dictionary, composed of the nodes of the returned connected component, are added to a list whose order is reversed and stored within the reversed\_l list. This reversal is justified by the fact that, in order to compute the solution, it is necessary to start from the leaves and climb the tree up to the root. Two other dictionaries are also used, called dp and mark. The dp dictionary is used as a supporting structure for the choice that’ll be made for a particular node about the installation of the software and taking into account its parent: in particular, with the expression dp[v][parent\_taken] we are examining a certain node v and its minimum number of nodes where the software is installed, in the subtree that has as root v, whether the parent is “taken” or “not taken” (by “taken” we intend the parent on which the software is installed). The mark dictionary is used to store the status of each node and indicate if the software will be installed on that node. In particular, there can be 3 states: S, N or S/N.

* The state S means that the software will be installed on the node.
* The state N means that the software will not be installed on the node.
* The state S/N means that further verifications are needed to decide if the software will be installed on it or not.

A for loop is then used for the nodes in the reversed\_l list and the leaf condition is checked for each one of them by calling the is\_leaf(v) method of the GeneralTree class.

* (Basic case) In this case the solution to the sub-problem is defined, which is represented by the fact that the node v under examination is a leaf. This node is marked with the state S/N and the number of nodes where the software is installed is determined, in the subtree that has v as root, both in case the parent node is “taken” and “not taken”: when the parent node is “not taken”, the corresponding value of the nodes is listed in the dictionary in dp[v][0]; when the parent node is “taken”, the corresponding value of the nodes is listed in the dictionary in dp[v][1]. For the leaf v under examination, the corresponding values listed in dp[v][0] and dp[v][1] are 1 and 0 respectively. Since v is a leaf, it means that there is no subtree that has this node as root, so it is considered the only pair composed by node v and its parent: when the parent is not the node on which the software is installed (the index 0 in dp[v][0]), then the node v itself is the one on which the software is installed (the value of dp[v][0] equal to 1); vice versa, when the parent has the software (the index 1 in dp[v][1]), then it is not the node v on which the software is installed (the value of dp[v][1] equal to 0).
* In the second condition fall all the cases in which the node v under examination is not a leaf (we are therefore merging the sub-problems, combining the computed solutions). For the node v under examination, the values in dp[v][0] and dp[v][1] are both set to None, because the next operations will determine these values. Two support variables are then used, called taken\_value and not\_taken\_value.
  + When the software is installed on the node v under examination, the taken\_value variable is defined as the minimum number of nodes on which the software is installed for all the children of node v. The related calculation is the value 1 (software installed on node v) summed to the minimum number of nodes of the children of v.
  + Instead, when the software is not installed on the node v under examination, but on the parent of v, the variable not\_taken\_value is defined as the minimum number of nodes on which the software is not installed for all the children of v.

At this point, for the leaf v under examination, the corresponding values in dp[v][0] and dp[v][1] are the value in taken\_value variable and the minimum value between taken\_value and not\_taken\_value respectively. All that is left is the definition of the node state, based on the comparison between the values computed in the two variables mentioned above.

* + If the value of taken\_value and the one of not\_taken\_value are equal, then the node is marked with the state S/N.
  + If the value of taken\_value is less than not\_taken\_value, then the node is marked with the state S.
  + If the value of taken\_value is greater than not\_taken\_value, then the node is marked with the state N.

At the end of the method, the find\_solution method is called to find the optimal solution.

**find\_solution method:** The parameters passed are the tree tree on which the solution must be searched upon, the dp and mark dictionaries, previously defined, and the list l of all the nodes of the tree that are part of its connected component. A dictionary, called installed, is used to store the nodes on which the software will be installed, and it’s returned as a result of the method. A for loop is then used starting from the nodes of the list l, passed as a parameter: the nodes are then examined in this order, from the root node down to the leaves, a choice justified by the fact that the optimum solution is being calculated. For each node in the list, the parent(self,v) method is called to obtain the node parent of node v under examination and two conditions are verified, namely that the parent obtained is not None and the value referred to the key parent within the installed dictionary is equal to False:

* If these conditions are not tested positively, it means that the node parent under examination is either None (so the node v under examination is the root) or it turns out that the node parent is a node on which the software must be installed. In this case, it is necessary to further examine node v.
  + if the state of node v is equal to the state S, then in the installed dictionary, at the key v, the value is set to True (so on the node v the software will be installed);
  + if the state of node v is equal to the state N, then in the installed dictionary, at the key v, the value is set to False (so on the node v the software will not be installed);
  + if the state of node v is equal to the state S/N, then in the installed dictionary, at the key v, the value is set to False (so on the node v the software will not be installed), but a further condition is verified which represents a peculiar case: if the node v under examination is the root, its children are analyzed and if there is even a single one that has the state N, then inside the installed dictionary, at the key v, the value is set to True.
* If these conditions are tested positively, it means that the node parent under examination is None and is not a node on which the software will be installed. So, in this case it is the node v that will be the node on which the software will be installed (by setting to True the value corresponding to its key, inside the installed dictionary).

For each node in the list, at the end of the verifications of the various conditions, will appear a message on the screen to know if the node is the one on which the software will be installed.

Regarding the computational complexity, we should consider that in the designed min\_nodes\_install method, a call to the recursive DFS algorithm is made using the DFS\_complete method to derive nodes belonging to the connected component of the tree. The complexity of the DFS algorithm is but since we are considering the DFS visit of a tree, where n is greater than m, then the complexity is . The algorithm that searches for the minimum number of nodes where to install the software always has a complexity equal to because even if the tree is visited both in ascending (from the leaves to the root) and vice versa, the single nodes are not visited twice as there are checking conditions that allow to switch to the next node if the one under examination is already equipped with the software.

**Exercise 5**

The exercise requires the design of a Greedy algorithm that installs on the computer of some users a software capable of recognizing and blocking fake news; in particular, the installation must be performed on at least one device for each pair of friends. According to the specifications of the exercise, the social network can be any graph, which implementation has been done using a dedicated class.

Starting from the definition of the class Graph with adjacency map, made available by the text book "Data Structures and Algorithms in Python" by the authors Goodrich, Tamassia and Goldwasser, for the design of the algorithm, the class Graph itself and its nested class Vertex have been modified. In the nested class Vertex, the constructor method has been modified, with the addition of an attribute, and get and set methods have been inserted. In the original class, within the constructor only the \_element attribute was present. For the purposes of algorithm design, the \_installed attribute has been added, a Boolean variable that is set to False by default and indicates whether or not the software is installed onto the vertex during the execution of the BaceFookAntifake algorithm. This attribute has been defined as private and therefore it was necessary to add the get and set methods to make it accessible, as it is manipulated during the algorithm execution.

Regarding the class Graph, the constructor method has been modified, with the addition of an attribute. In the original class, within the constructor there were two maps, \_outgoing and \_incoming. For the purposes of algorithm design, the \_notInstalled attribute has been added, a dictionary used to store the number of adjacent uninstalled vertices (value) for each vertex (key). The following methods have also been modified and added.

* insert\_vertex(self,x=None): when a new vertex is inserted, in addition to its creation, it is added to the dictionary \_notInstalled, at key vertex and with value 0.
* insert\_edge(self,u,v,x=None): in addition to the creation of the edge, if the graph is not direct, the number of adjacent vertices that do not have the software installed is increased by one, both for the origin vertex and destination vertex; otherwise it’s increased only for the origin vertex.
* get\_not\_installed\_adj(self,v): this method, which does not exist in the original class, takes one vertex and returns the related number of adjacent vertices that do not have the software installed, using the \_notInstalled dictionary, created within the constructor.
* decrease\_not\_installed\_adj(self,v,adjacent\_installation=False): this method, which does not exist in the original class, takes a vertex and an optional adjacent\_installation parameter (set to False if it is not passed by the caller) and provides no return value. If the adjacent\_installation parameter is:
  + True, the number of adjacent vertices of v (the incoming vertex), which do not have the software installed, is decreased by 1;
  + False, for each adjacent vertex of v, the number of adjacent vertexes which do not have the software installed is decreased by 1. In particular, in the case of directed graphs, this decrease is performed only for adjacent vertices which have an edge entering v.

**Introduction to the designed method:** The designed method takes in input a graph and returns in output a list, which contains the vertices on which the software is installed to recognize and block fake news. The size of the list represents the minimum number of vertices for the software to work properly. As required by the specifications of the exercise, the algorithm installs the software on at least one vertex in each pair of vertices.

**Initialization**: The designed method starts with initializing the dictionary which will be its return value.

**Execution of the algorithm:** Once the initialization phase is complete, the actual execution of the designed Greedy algorithm can start. The first step is to initialize a cycle that will scroll through all the vertices of the graph. For the current vertex u two conditions are checked: if the \_installed attribute is set to False and if the number of adjacent vertices that do not have the software installed is other than zero.

* If the checking is not successful, then the cycle starts again moving to the next vertex of the graph.
* If the checking is successful, all the adjacent vertices of the current vertex u are examined. For each adjacent vertex v, its \_installed attribute is checked, and in this case two situation can occur:
  + positive check, the cycle starts again, switching to the next adjacent vertex;
  + failing check, the current cycle goes on and the number of adjacent vertices of v, which do not have the software, is tested to see if it’s greater than or equal to the one of vertex u. If the control returns:
    - False, then the software is installed on vertex u by setting its \_installed attribute to True, and this vertex is inserted in the list that will be returned at the end of the algorithm. The method decrease\_not\_installed\_adj is called which will decrease the number of adjacent vertices that don’t have the software installed. Lastly, the break command is invoked to stop the cycle of the list of incident edges to vertex u in order to switch to the next vertex u of the graph;
    - True, the same operations are performed as in the previous case, but on vertex v, with the exception of the break command because since they’re performed on one of the vertices adjacent to u, then it is necessary to keep scrolling the list of these vertices and switching to the next one.

**Greedy algorithm performance:** For our greedy algorithm performance evaluation, we have considered the article ‘Approximation Algorithms: Vertex Cover’[[2]](#footnote-3). As this article says, many practical significant optimization problems are NP-Hard, because it is hard to find an algorithm that solves them in polynomial time. The problem proposed in this exercise belongs to this category; in particular, we can associate it to the ‘Vertex cover’ problem. The latter is formally described in this way: ‘find minimum set of vertices that covers all the edges in the graph’. The algorithm proposed for the problem resolution runs in polynomial time and outputs a solution close to the optimal one; thus, it belongs to “approximation algorithms” and it is evaluable through a performance parameter α.

Generally, let P be a minimization problem and I be an instance of P. Let A be an algorithm that finds feasible solution to instances of P. A(I) is the cost of the solution returned by the algorithm A for instance I, and OPT(I) is the cost of the optimal solution (minimum) for I. Then, A is said to be an α-approximation algorithm for the P if:

The article proofs that 2-approximation is a tight bound for the ‘vertex cover’ problem, so we can consider 2 as the upper bound for α value.

**Greedy algorithm solution:** Our greedy solution can be formalized by the Integer Programming formulation. Let G = (V, E) be a graph and a variable such that:

then the solution of the algorithm will be in the form:

Thus, we can conclude that if we minimize the sum of , we have also found the minimum vertex cover, or in other words:

**Optimal solution:** Considering the abovementioned article, the ‘Vertex cover’ problem can be formalized through the following algorithm:

* find a maximal matching M;
* return the set of end-points of all edges ∈ M.

M represents the graph maximal matching. The matching is a set of [edges](https://en.wikipedia.org/wiki/Edge_(graph_theory)) without common [vertices](https://en.wikipedia.org/wiki/Vertex_(graph_theory)); it is maximal if every edge in G has a non-empty intersection with at least one edge in M. The optimum vertex cover must cover every edge in M. So, it must include at least one of the endpoints of each edge ∈ M, where no 2 edges in M share an endpoint. Hence, optimum vertex cover must have size:

**Algorithm approximation ratio:** The computation of our algorithm approximation ratio has been executed through experimental proofs applied on randomly generated graphs. For each graph, we compared the algorithm result and the optimal solution, obtaining α values included in the interval [1.3607, 2]. The lower bound depends on Dinur-Safra (2001) Theorem, demonstrated for this kind of problems.

1. <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.109.4875&rep=rep1&type=pdf> [↑](#footnote-ref-2)
2. <http://tandy.cs.illinois.edu/dartmouth-cs-approx.pdf> [↑](#footnote-ref-3)