Programming Part

Problem 3 - Dictionary Set Administration (100 pts)

Problem Description

We have learned some string-matching algorithms such as Rabin-Karp and KMP in class, which efficiently find substrings within larger strings. However, these algorithms are not designed for scenarios where we need to efficiently store and retrieve multiple strings with their common prefixes. For example, when implementing an autocomplete system, searching for strings with a given prefix or organizing a dictionary for efficient lookups becomes crucial. Such tasks require data structures specifically designed for prefix-based operations.

In this problem, we suggest you learn a new data structure called **Trie** to address prefix searching more efficiently. The following description of **Trie** might be helpful for you.

Reference Reading

Definitions. Trie is a specialized tree-based data structure used for storing and retrieving strings, typically in applications where prefix-based operations are crucial. Tries are particularly efficient for tasks such as autocomplete systems, spell checkers, or dictionary management. Let T be a Trie storing a *set* of strings $S = \{s_1, s_2, \ldots, s_n\}$, where each string is composed of characters from a fixed alphabet Σ.

- Each edge represents a character in Σ.
- The root node represents a null string.
- Each node in the Trie represents a prefix of some strings in S. The prefix string can be obtained by concatenating all characters represented by the edges in the path from the root to the node.
- Each string $s \in S$ can be presented by a leaf node in the Trie. Similarly, concatenating the characters represented by the edges forming the path from the root to the leaf node produces the string s.

Figure 3.1 presents an example of Trie. Note that it branches at the common prefix of those in S, i.e., "ca".

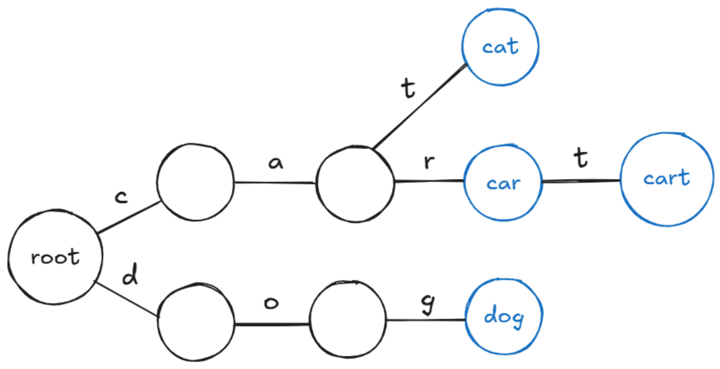

Figure 3.1: A Trie representing $S = \{\text{“cat”}, \text{“car”}, \text{“cart”}, \text{“dog”}\}$

Key properties of a Trie.

- **Prefix representation:** Every prefix of a string $s \in S$ can be presented by a node in T.
- **Search efficiency:** Finding a string or prefix of length m in a Trie takes $O(m)$ time, regardless of the number of strings stored in the Trie.
- **Space efficiency:** Common prefixes among the strings in S are stored only once.

Implementation. A Trie is constructed by inserting a string one character at a time. Each node stores the pointers to the child nodes, each of which corresponds to adding one more character drawn from Σ. In addition, a boolean flag is stored at each node, indicating whether the current node corresponds to a string $s \in S$.

To *insert* a string s into Trie T:

(1) Start from the root node.
(2) For each character c in s: If c does not exist as a child of the current node, create a new node for c.
(3) Move to the child node corresponding to c.
(4) After processing all characters, mark the final node to indicate that the node corresponds to a string in S.

To *search* for a string s in the Trie:

(1) Start from the root node.
(2) Take the next character c in s. Traverse to the child node corresponding to c. If c cannot be found on an edge from the current node, stop the current search and return “not found.”
(3) Repeat the previous step until all characters in s are processed. If all characters are found and the final node is marked as the end of a string, s exists in the Trie.

For *prefix operations*, such as finding all strings of a given prefix p:

(1) Traverse the Trie to find the node corresponding to p using the above *search* algorithm.

(2) Perform a depth-first search (DFS) from this node to collect all strings that share the prefix p.

In this problem, we ask you to implement a dictionary set to support the following six operations. The dictionary set should be implemented with Trie and was initially empty. The six operations are:

- **Insert** operation: Insert the query string into the dictionary set.
- **Check** operation: Check whether the query string is in the dictionary set.
- **Prefix_Search** operation: For a given prefix string q, we have $P = \{p : \ q \sqsubset p \wedge p \in S\}$ representing all strings in the dictionary set S such that q is a prefix of them. The Prefix_Search operation asks you to return the number of these strings, i.e., $|P|$.
- **LCP** operation: Assume that the query string q has the longest common prefix with some strings in the dictionary set S. Let us denote the set S' to have those strings. Note that in the case that the longest common prefix is an empty string, $S' = S$. The LCP operation returns the first string of S' in lexicographical order.
- **Score** operation: The score of the query string q to the dictionary set S is given by

$$\mathcal{S}(q, S) = \sum_{t \in C} |t|, C = \{t : \ t \sqsubset q \ \wedge \ t \sqsubset s \ \wedge \ s \in S\} \tag{3.1}$$

  where C is the set of common prefixes of q and any $s \in S$. The score $\mathcal{S}(q, S)$ is the summation of the lengths of all strings in C. The LCP operation asks you to calculate this score with a query string s and the dictionary set S.
- **Compress** operation: Compress the entire dictionary set by replacing each string in the dictionary set with one of its non-empty prefixes such that all new strings are still unique. Let S_c denote the dictionary set after compression. Find a way to minimize the total length of compressed strings $L = \sum_{s \in S_c} |s|$ and return the value of the minimal L. For example, assume that there are four strings “hsinmu”, “hsin”, “hsuantien”, and “hello” in the dictionary set. After the Compress operation, “hsinmu” can be compressed to “hsi”, “hsin” can be compressed to “hs”, “hsuantien” can be compressed to “h”, “hello” can be compressed to “he”. The four compressed strings cah be “hsi”, “hs”, “h”, and “he”, which are all unique, and $L = 8$. The Compress operation can only be the last in the sequence.

Input

The first line contains one integer Q, representing the total number of operations. Each of the next Q lines is given in one of the following formats:

- `1 str`: Indicating an **Insert** operation with string `str`.
- `2 str`: Indicating a **Check** operation with string `str`.
- `3 prefix`: Indicating a **Prefix_Search** operation with string `prefix`.
- `4 str`: Indicating a **LCP** operation with query string `str`.
- `5 str`: Indicating a **Score** operation with query string `str`.
- `6`: Indicating a **Compress** operation.

Output

- For each **Check** operation, print `YES` in uppercase English letters in a single line if `str` is in the dictionary set. Otherwise, print `NO`.
- For each **Prefix_Search** operation, print a non-negative integer in a single line representing the number of strings in the dictionary set that have `str` as a prefix.
- For each **LCP** operation, print a string in a single line representing the first string in the dictionary set that has the **longest common prefix** with `str` in lexicographical order.
- For each **Score** operation, print a non-negative integer in a single line representing the score of `str`.
- For each **Compress** operation, print a non-negative integer in a single line representing the smallest possible total length of the compressed dictionary set.

Constraints

- $1 \le Q \le 10^4$
- The length of one single input string is not greater than 10^4.
- The total length of input strings is not greater than 3×10^5.
- The inserted strings are unique.
- Each string contains only lowercase English letters.
- The **Compress** operation will only be called once and always be the last one.

Subtasks

Subtask 1 (10 pts)

- Only includes operation 1, 2, 3 (**Insert**, **Check**, **Prefix_Search**).

Subtask 2 (25 pts)

- Only includes operation 1, 2, 3, 4 (**Insert**, **Check**, **Prefix_Search**, **LCP**).

Subtask 3 (25 pts)

- Only includes operation 1, 2, 3, 4, 5 (**Insert**, **Check**, **Prefix_Search**, **LCP**, **Score**).

Subtask 4 (40 pts)

- Includes all operations.

Sample Testcases

Sample Input 1

```
7
1 hsinmu
1 hsuantien
1 dsa
1 csie
2 dsa
2 ntu
3 hs
```

Sample Output 1

```
YES
NO
2
```

Sample Input 2

```
8
1 hsinmu
1 hsin
1 hsuantien
1 hello
4 hsinchiji
4 hsuchihmo
5 hell
6
```

Sample Output 2

```
hsin
hsuantien
7
8
```

Sample Explanation 2

- `4 hsinchiji` will find `hsin` as the longest common prefix with length = 4. Note that even if `hsinmu` has also the longest common prefix with length = 4, it is lexicographically larger than `hsin`, so the answer should be `hsin`.
- `4 hsuchihmo` will find `hsuantien` as the longest common prefix with length = 3.
- `5 hell` will score 1 on `hsinmu`, score 1 on `hsin`, score 1 on `hsuantien`, and score 4 on `hello`. Hence, the total score is 7.
- After the **Compress** operation, `hsinmu` can be compressed to `hsi`, `hsin` can be compressed to `hs`, `hsuantien` can be compressed to `h`, `hello` can be compressed to `he`. The four compressed strings will be `hsi`, `hs`, `h`, and `he`, which are all unique, and the total length is 8. It can be proved that there isn't any compression method to have a shorter total length.

Sample Input 3

```
10
1 cat
1 car
1 cart
1 dog
2 car
3 car
4 california
4 carbon
5 cartoon
6
```

Sample Output 3

```
YES
2
car
car
9
7
```

Problem 4 - DSA Space Administration (100 pts)

Problem Description

The DSA Space Administration, **DSA** in short, manages a modular space station deployed in Galaxy 1126. The stability and operation of this station rely on a sophisticated internal structure implemented using a **Red-Black Tree**. This data structure ensures efficiency and balance when managing the space station's greenhouse modules.

Node Specification

Each Node in this tree represents a Space Greenhouse and has the following attributes:

- t: **A unique ID** used to distinguish different greenhouses, which is also considered the **key** in the Red-Black Tree structure.
- d_t: **The depth** of the greenhouse t within the Red-Black Tree (e.g., the root node has depth 0, its children have depth 1, and so on). The depth can change due to tree operations.
- p_t: **The population**, which is the number of people currently residing in greenhouse t.
- **Node color**: Red or black, maintained according to the rules of the Red-Black Tree as introduced in the lecture.

In addition to these actual greenhouse nodes, the conceptual **NIL nodes** (leaves) are considered part of the structure for connectivity and Red-Black Tree rules. These NIL nodes have the following properties:

- ID (t): They do not have a unique ID.
- Depth (d_t): For distance calculations, a NIL node is considered to have a depth one greater than its parent node.
- Population (p_t): Always 0.
- Node color: Always black.

Connectivity Channels

The station utilizes two types of channels for internal transport and resource allocation:

- **Tree Structure Channels**: These are the standard parent-child links inherent in the Red-Black Tree structure. A direct channel exists between any parent node and its child node.
- **Same-Color Dimensional Channels**: A dimensional channel exists *between any two greenhouses (nodes) u and v that share the same color* (i.e., either both u and v are

black, or both u and v are red). These provide additional shortcuts beyond the basic tree structure.

Distance Definition

- **Channel cost**: The cost or “length” of traversing a single Connectivity Channel (either a tree structure channel between parent/child or a same-color dimensional channel between same-colored nodes u and v) is defined as the *absolute difference in their depths.*
- **Path distance**: A path consists of any sequence of available channels, and its cost is the sum of the costs of those channels.
- **Shortest distance between nodes**: The shortest distance between two nodes u and v is defined as the minimum total cost among all possible paths connecting them.

The parent-child relationships within the Red-Black Tree define the fundamental structural connections between the greenhouses in the station. Michael, the Chief Structural Engineer of DSA, needs to maintain this tree structure. Additionally, he must be able to analyze the distances between greenhouses and evaluate accessibility metrics based on the current structure to improve the quality of life for the residents continuously.

Operations

Your task is to implement a system to assist Michael in managing this Red-Black Tree structure. The system must be able to process a series of operational commands. For any operation that modifies the tree structure (i.e., node insertions or deletions), the system must ensure that all Red-Black Tree properties remain valid after the operation.

The system needs to support the following operations:

1. **Add Greenhouse (Connect)**: 1 x p
   Adds a node representing greenhouse with ID x and population p to the Red-Black Tree structure.
   - Property Maintenance: Adjust the tree (rotations, recoloring) to maintain Red-Black Tree properties after insertion. The specific algorithms for insertion, rotations, and recoloring should follow the methods presented in the **course lecture notes** and the **Appendix** section below.

   Notes:
   - If greenhouse x already exists, this operation is ignored.
2. **Remove Greenhouse (Disconnect)**: 2 x
   Removes the greenhouse node with the specified ID x from the space station.

- Property Maintenance: Adjust the tree to maintain Red-Black Tree properties after deletion. The specific algorithms for deletion, rotations, and recoloring should follow the methods presented in the **course lecture notes** and the **Appendix** section below.

Notes:

- If greenhouse x does not exist, this operation is ignored.
- Populations within greenhouses that have been removed are excluded from further consideration.

3. **Relocate Population (Relocate)**: 3 u v p

   Moves p people from greenhouse u to greenhouse v.

   - The actual number of people moved will be $\min(p_u, p)$, where p_u is the current population of greenhouse u.
   - The population of greenhouse u is decreased by the actual number of people moved.
   - The population of greenhouse v is increased by the actual number of people moved.

   Notes:

   - If $u = v$, this operation is ignored as it results in no net change.
   - If greenhouse u or greenhouse v (or both) do not exist, this operation is ignored.

4. **Emergency Evacuation (Evacuate)**: 4 x

   Moves all people from greenhouse x to its parent and its non-NIL descendant greenhouses (its subtree excluding itself).

   Distribution Logic:

   Let S be the set of descendant greenhouses of x, excluding NIL nodes. Let $|S|$ denote the number of these descendant greenhouses, and p_x be the population of greenhouse x before applying this operation.

   - Each descendant greenhouse $v \in S$ receives $\lfloor \frac{p_x}{|S|+1} \rfloor$ people from greenhouse x.
   - The total number of people distributed to children is

   $$D = \sum_{v \in S} \lfloor \frac{p_x}{|S| + 1} \rfloor = \lfloor \frac{p_x}{|S| + 1} \rfloor \times |S|$$

   - The remaining people, $R = p_x - D$, are handled as follows:
     - If greenhouse x has a parent, the parent greenhouse's population is increased by R.
     - If greenhouse x is the root (i.e., has no parent), these R people are considered to have departed the space station and are lost.
   - Consequently, the population of greenhouse x is set to 0.

   Notes:

   - If greenhouse x does not exist, this operation is ignored.

5. **Calculate Shortest Distance (Distance)**: 5 u v
   Queries the **Path Distance** (as previously defined) between nodes u and v in the Red-Black Tree.
   Notes:
   - If greenhouse u or greenhouse v (or both) do not exist in the Red-Black Tree, the output for this query is -1.
   - If $u = v$ the distance is considered 0.
6. **Calculate Accessibility Index (Access Index)**: 6 x
   Queries the Accessibility Index $\mathcal{A}$ for a given target greenhouse x. Let G denote the set of greenhouses. The accessibility index is given by

$$\mathcal{A}(x) = \sum_{w \in G} p_w \times [\text{dist}(w, x)]^2 \tag{3.2}$$

   where
   - The sum is over all existing greenhouse nodes $w \in G$;
   - p_w is the population of greenhouse node w;
   - $\text{dist}(w, x)$ is the shortest distance between greenhouse node w and target greenhouse node x (as calculated by Operation 5).

   Notes:
   - If the target greenhouse x does not exist, the output for this query is -1.

Constraints

- $1 \leq Q \leq 2 \times 10^5$: Number of operations
- $1 \leq x, u, v \leq 10^9$: The unique ID of the greenhouse
- $1 \leq p \leq 10^9$: The population

Subtasks

Subtask 1 (20 pts)

- $1 \leq Q \leq 1000$
- Only includes operations 1, 2, 5

Subtask 2 (20 pts)

- $1 \leq Q \leq 1000$
- Include all operations

Subtask 3 (20 pts)

- $1 \leq Q \leq 2 \times 10^5$
- Only includes operations 1, 3, 5, 6

Subtask 4 (20 pts)

- $1 \leq Q \leq 2 \times 10^5$
- Only includes operations 1, 2, 3, 5, 6

Subtask 5 (20 pts)

- $1 \leq Q \leq 2 \times 10^5$
- Include all operations

Input Format

The first line contains a single integer Q, representing the total number of operations to process. Each of the next Q lines contains one operation, formatted exactly as described above in the "Operations" section.

Output Format

For each operation of type 5 (Distance) and type 6 (Access Index), output a line with a single integer, representing the desired answer for the given query.

Hints

- The initial subtasks are designed to help you verify the core Red-Black Tree operations (insertion, deletion, property maintenance) before tackling more complex logic.
- Review the Red-Black Tree properties taught in class; you may find it helpful for this problem! When calculating distances between nodes of different colors (e.g., from a Black node to a Red node, or vice-versa), think about Red-Black Tree properties. How can these property, combined with the Same-Color Dimensional Channels, help you construct an efficient path?
- For the Accessibility Index, instead of summing for each greenhouse node w individually, you can simplify the calculation. Try to keep track of the populations of the nodes that share some properties ensuring they will all yield the same value from the distance formula you derived.

- For visualization, tools like red-black-tree-dataviz can be helpful. However, be aware that their implementation might differ from textbook algorithms, so relying solely on their behavior may not guarantee a correct solution.

Sample Testcases

Sample Input 1

```
6
1 10 100
1 20 200
1 30 300
5 10 30
1 5 50
5 5 30
```

Sample Output 1

```
0
1
```

Sample Input 2

```
13
1 10 100
1 5 50
1 15 150
5 5 15
1 20 200
5 5 20
6 10
1 2 20
3 10 5 30
2 5
5 2 20
5 100 200
6 5
```

Sample Output 2

```
0
1
1000
1
-1
-1
```

Sample Input 3

```
17
1 100 1000
1 50 500
1 150 1500
1 25 250
1 75 750
6 25
6 50
3 100 150 300
6 25
6 50
4 50
6 25
6 50
1 30 300
6 50
2 100
6 30
```

Sample Output 3

```
6000
2000
5100
1700
5272
2200
10600
2216
```

Appendix: Red-Black Tree Pseudocode

DISCLAIMER: The pseudocode presented on the following pages is taken directly from the course textbook and is included herein strictly for educational purposes. This material is provided on an ”as-is” basis. While efforts are made to ensure accurate reproduction, no warranty (express or implied) is given as to its absolute correctness, completeness, or fitness for any specific application beyond the educational scope intended. Viewers are responsible for their own interpretation and use of this information.

Please note: If a function used in the provided pseudocode is not fully implemented in the textbook's version, you are encouraged to implement it yourself. This is intended as part of the learning exercise and, hopefully, should be straightforward :) Furthermore, it is crucial that your own implementation closely follows the logic and structure presented in the textbook's pseudocode. Deviations from the textbook's approach may result in an inability to ensure your solution can pass all provided test cases.

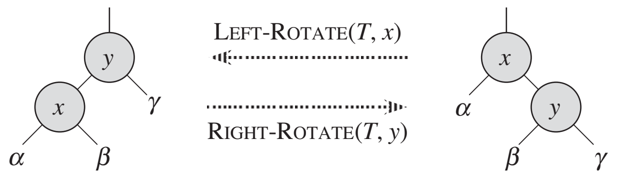

Figure 13.2 The rotation operations on a binary search tree. The operation LEFT-ROTATE(T, x) transforms the configuration of the two nodes on the right into the configuration on the left by changing a constant number of pointers. The inverse operation RIGHT-ROTATE(T, y) transforms the configuration on the left into the configuration on the right. The letters α, β, and γ represent arbitrary subtrees. A rotation operation preserves the binary-search-tree property: the keys in α precede $x.key$, which precedes the keys in β, which precede $y.key$, which precedes the keys in γ.

```
LEFT-ROTATE(T, x)
 1  $y = x.right$                  // set $y$
 2  $x.right = y.left$             // turn $y$'s left subtree into $x$'s right subtree
 3  if $y.left \neq T.nil$
 4      $y.left.p = x$
 5  $y.p = x.p$                    // link $x$'s parent to $y$
 6  if $x.p == T.nil$
 7      $T.root = y$
 8  elseif $x == x.p.left$
 9      $x.p.left = y$
10  else $x.p.right = y$
11  $y.left = x$                   // put $x$ on $y$'s left
12  $x.p = y$
```

Figure 13.3 shows an example of how LEFT-ROTATE modifies a binary search tree. The code for RIGHT-ROTATE is symmetric. Both LEFT-ROTATE and RIGHT-ROTATE run in $O(1)$ time. Only pointers are changed by a rotation; all other attributes in a node remain the same.

Exercises

13.2-1
Write pseudocode for RIGHT-ROTATE.

13.2-2
Argue that in every n-node binary search tree, there are exactly $n - 1$ possible rotations.

13.3 Insertion

We can insert a node into an n-node red-black tree in $O(\lg n)$ time. To do so, we use a slightly modified version of the TREE-INSERT procedure (Section 12.3) to insert node z into the tree T as if it were an ordinary binary search tree, and then we color z red. (Exercise 13.3-1 asks you to explain why we choose to make node z red rather than black.) To guarantee that the red-black properties are preserved, we then call an auxiliary procedure RB-INSERT-FIXUP to recolor nodes and perform rotations. The call RB-INSERT(T, z) inserts node z, whose *key* is assumed to have already been filled in, into the red-black tree T.

```
RB-INSERT(T, z)
 1  $y = T.nil$
 2  $x = T.root$
 3  while $x \neq T.nil$
 4      $y = x$
 5      if $z.key < x.key$
 6          $x = x.left$
 7      else $x = x.right$
 8  $z.p = y$
 9  if $y == T.nil$
10      $T.root = z$
11  elseif $z.key < y.key$
12      $y.left = z$
13  else $y.right = z$
14  $z.left = T.nil$
15  $z.right = T.nil$
16  $z.color = $ RED
17  RB-INSERT-FIXUP(T, z)
```

The procedures TREE-INSERT and RB-INSERT differ in four ways. First, all instances of NIL in TREE-INSERT are replaced by $T.nil$. Second, we set $z.left$ and $z.right$ to $T.nil$ in lines 14–15 of RB-INSERT, in order to maintain the proper tree structure. Third, we color z red in line 16. Fourth, because coloring z red may cause a violation of one of the red-black properties, we call RB-INSERT-FIXUP(T, z) in line 17 of RB-INSERT to restore the red-black properties.

```
RB-INSERT-FIXUP(T, z)
 1  while z.p.color == RED
 2      if z.p == z.p.p.left
 3          y = z.p.p.right
 4          if y.color == RED
 5              z.p.color = BLACK                       // case 1
 6              y.color = BLACK                         // case 1
 7              z.p.p.color = RED                       // case 1
 8              z = z.p.p                               // case 1
 9          else if z == z.p.right
10                  z = z.p                             // case 2
11                  LEFT-ROTATE(T, z)                   // case 2
12              z.p.color = BLACK                       // case 3
13              z.p.p.color = RED                       // case 3
14              RIGHT-ROTATE(T, z.p.p)                  // case 3
15      else (same as then clause
                with "right" and "left" exchanged)
16  T.root.color = BLACK
```

To understand how RB-INSERT-FIXUP works, we shall break our examination of the code into three major steps. First, we shall determine what violations of the red-black properties are introduced in RB-INSERT when node z is inserted and colored red. Second, we shall examine the overall goal of the **while** loop in lines 1–15. Finally, we shall explore each of the three cases[1] within the **while** loop's body and see how they accomplish the goal. Figure 13.4 shows how RB-INSERT-FIXUP operates on a sample red-black tree.

Which of the red-black properties might be violated upon the call to RB-INSERT-FIXUP? Property 1 certainly continues to hold, as does property 3, since both children of the newly inserted red node are the sentinel $T.nil$. Property 5, which says that the number of black nodes is the same on every simple path from a given node, is satisfied as well, because node z replaces the (black) sentinel, and node z is red with sentinel children. Thus, the only properties that might be violated are property 2, which requires the root to be black, and property 4, which says that a red node cannot have a red child. Both possible violations are due to z being colored red. Property 2 is violated if z is the root, and property 4 is violated if z's parent is red. Figure 13.4(a) shows a violation of property 4 after the node z has been inserted.

[1] Case 2 falls through into case 3, and so these two cases are not mutually exclusive.

13.4 Deletion

Like the other basic operations on an n-node red-black tree, deletion of a node takes time $O(\lg n)$. Deleting a node from a red-black tree is a bit more complicated than inserting a node.

The procedure for deleting a node from a red-black tree is based on the TREE-DELETE procedure (Section 12.3). First, we need to customize the TRANSPLANT subroutine that TREE-DELETE calls so that it applies to a red-black tree:

```
RB-TRANSPLANT(T, u, v)
1  if u.p == T.nil
2      T.root = v
3  elseif u == u.p.left
4      u.p.left = v
5  else u.p.right = v
6  v.p = u.p
```

The procedure RB-TRANSPLANT differs from TRANSPLANT in two ways. First, line 1 references the sentinel $T.nil$ instead of NIL. Second, the assignment to $v.p$ in line 6 occurs unconditionally: we can assign to $v.p$ even if v points to the sentinel. In fact, we shall exploit the ability to assign to $v.p$ when $v = T.nil$.

The procedure RB-DELETE is like the TREE-DELETE procedure, but with additional lines of pseudocode. Some of the additional lines keep track of a node y that might cause violations of the red-black properties. When we want to delete node z and z has fewer than two children, then z is removed from the tree, and we want y to be z. When z has two children, then y should be z's successor, and y moves into z's position in the tree. We also remember y's color before it is removed from or moved within the tree, and we keep track of the node x that moves into y's original position in the tree, because node x might also cause violations of the red-black properties. After deleting node z, RB-DELETE calls an auxiliary procedure RB-DELETE-FIXUP, which changes colors and performs rotations to restore the red-black properties.

```
RB-DELETE(T, z)
 1  y = z
 2  y-original-color = y.color
 3  if z.left == T.nil
 4      x = z.right
 5      RB-TRANSPLANT(T, z, z.right)
 6  elseif z.right == T.nil
 7      x = z.left
 8      RB-TRANSPLANT(T, z, z.left)
 9  else y = TREE-MINIMUM(z.right)
10      y-original-color = y.color
11      x = y.right
12      if y.p == z
13          x.p = y
14      else RB-TRANSPLANT(T, y, y.right)
15          y.right = z.right
16          y.right.p = y
17      RB-TRANSPLANT(T, z, y)
18      y.left = z.left
19      y.left.p = y
20      y.color = z.color
21  if y-original-color == BLACK
22      RB-DELETE-FIXUP(T, x)
```

Although RB-DELETE contains almost twice as many lines of pseudocode as TREE-DELETE, the two procedures have the same basic structure. You can find each line of TREE-DELETE within RB-DELETE (with the changes of replacing NIL by *T.nil* and replacing calls to TRANSPLANT by calls to RB-TRANSPLANT), executed under the same conditions.

Here are the other differences between the two procedures:

- We maintain node y as the node either removed from the tree or moved within the tree. Line 1 sets y to point to node z when z has fewer than two children and is therefore removed. When z has two children, line 9 sets y to point to z's successor, just as in TREE-DELETE, and y will move into z's position in the tree.
- Because node y's color might change, the variable *y-original-color* stores y's color before any changes occur. Lines 2 and 10 set this variable immediately after assignments to y. When z has two children, then $y \neq z$ and node y moves into node z's original position in the red-black tree; line 20 gives y the same color as z. We need to save y's original color in order to test it at the

node x is either "doubly black" or "red-and-black," and it contributes either 2 or 1, respectively, to the count of black nodes on simple paths containing x. The *color* attribute of x will still be either RED (if x is red-and-black) or BLACK (if x is doubly black). In other words, the extra black on a node is reflected in x's pointing to the node rather than in the *color* attribute.

We can now see the procedure RB-DELETE-FIXUP and examine how it restores the red-black properties to the search tree.

```
RB-DELETE-FIXUP(T, x)
 1  while $x \neq T.root$ and $x.color$ == BLACK
 2      if $x$ == $x.p.left$
 3          $w = x.p.right$
 4          if $w.color$ == RED
 5              $w.color$ = BLACK                              // case 1
 6              $x.p.color$ = RED                              // case 1
 7              LEFT-ROTATE($T, x.p$)                          // case 1
 8              $w = x.p.right$                                // case 1
 9          if $w.left.color$ == BLACK and $w.right.color$ == BLACK
10              $w.color$ = RED                                // case 2
11              $x = x.p$                                      // case 2
12          else if $w.right.color$ == BLACK
13                  $w.left.color$ = BLACK                     // case 3
14                  $w.color$ = RED                            // case 3
15                  RIGHT-ROTATE($T, w$)                       // case 3
16                  $w = x.p.right$                            // case 3
17              $w.color = x.p.color$                          // case 4
18              $x.p.color$ = BLACK                            // case 4
19              $w.right.color$ = BLACK                        // case 4
20              LEFT-ROTATE($T, x.p$)                          // case 4
21              $x = T.root$                                   // case 4
22      else (same as then clause with "right" and "left" exchanged)
23  $x.color$ = BLACK
```

The procedure RB-DELETE-FIXUP restores properties 1, 2, and 4. Exercises 13.4-1 and 13.4-2 ask you to show that the procedure restores properties 2 and 4, and so in the remainder of this section, we shall focus on property 1. The goal of the **while** loop in lines 1–22 is to move the extra black up the tree until

1. x points to a red-and-black node, in which case we color x (singly) black in line 23;
2. x points to the root, in which case we simply "remove" the extra black; or
3. having performed suitable rotations and recolorings, we exit the loop.