**CHAPTER 12**

**Advanced Data Structures and Implementation**

**12.3** Incorporate an additional field for each node that indicates the size of its subtree. These fields are easy to update during a splay. This is difficult to do in a skip list.

**12.6** If there are *B* black nodes on the path from the root to all leaves, it is easy to show by induction that there are at most 2*B* leaves. Consequently, the number of black nodes on a path is at most log *N*. Since there can’t be two consecutive red nodes, the height is bounded by 2 log *N*.

**12.7** Color nonroot nodes red if their height is even and their parents height is odd, and black otherwise. Not all red black trees are AVL trees (since the deepest red black tree is deeper than the deepest AVL tree).

**12.8** The problem asks us to compute the suffix array, LCP array, and suffix tree for the two strings ABCABCABC and MISSISSIPPI. The suffix array shows the suffixes in alphabetical order. For ABCABCABC, the possible suffixes are:

C

BC

ABC

CABC

BCABC

ABCABC

CABCABC

BCABCABC

ABCABCABC

And we need to alphabetize them to produce the suffix array. Beside each one we can identify which suffix it is by indicating the index of its first character. (9 – length of suffix)

|  |  |
| --- | --- |
| Index | Suffix |
| 6 | ABC |
| 3 | ABCABC |
| 0 | ABCABCABC |
| 7 | BC |
| 4 | BCABC |
| 1 | BCABCABC |
| 8 | C |
| 5 | CABC |
| 2 | CABCABC |

In the case of MISSISSIPPI, the possible suffixes are:

I

PI

PPI

IPPI

SIPPI

SSIPPI

ISSIPPI

SISSIPPI

SSISSIPPI

ISSISSIPPI

MISSISSIPPI

And we need to alphabetize them to produce the suffix array. Beside each one we can identify which suffix it is by indicating the index of its first character. (11 – length of suffix)

|  |  |
| --- | --- |
| Index | Suffix |
| 10 | I |
| 7 | IPPI |
| 4 | ISSIPPI |
| 1 | ISSISSIPPI |
| 0 | MISSISSIPPI |
| 9 | PI |
| 8 | PPI |
| 6 | SIPPI |
| 3 | SISSIPPI |
| 5 | SSIPPI |
| 2 | SSISSIPPI |

Now for the LCP arrays. The first suffix does not have an LCP (longest common prefix). For each of the others, we determine how many initial characters match between this suffix and the one before it.

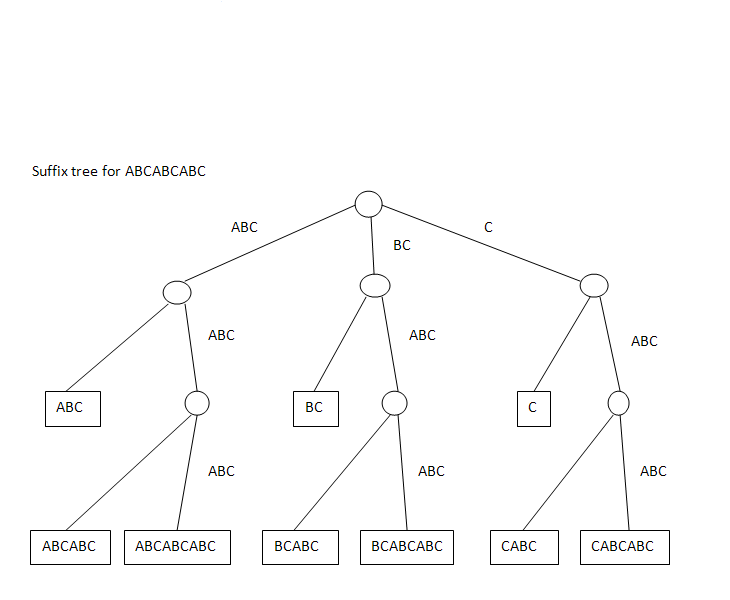
We can complete the tables as follows. For ABCABCABC, we have:

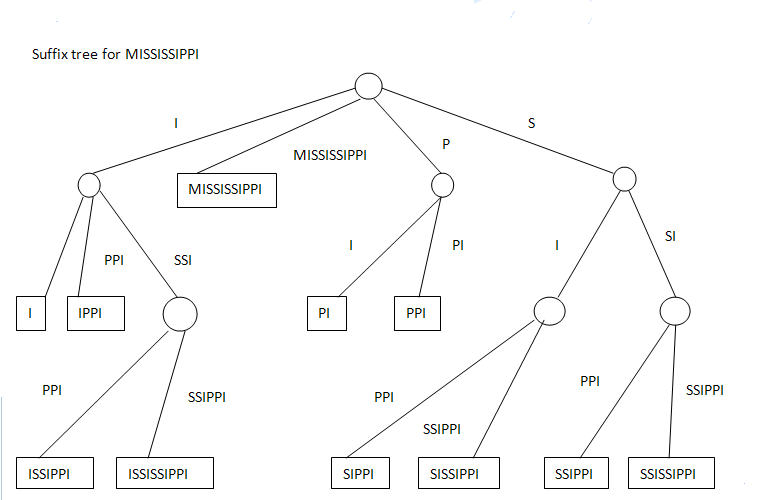
|  |  |  |
| --- | --- | --- |
| Index | LCP | Prefix |
| 6 | - | ABC |
| 3 | 3 | ABCABC |
| 0 | 6 | ABCABCABC |
| 7 | 0 | BC |
| 4 | 2 | BCABC |
| 1 | 5 | BCABCABC |
| 8 | 0 | C |
| 5 | 1 | CABC |
| 2 | 4 | CABCABC |

For MISSISSIPPI, we have:

|  |  |  |
| --- | --- | --- |
| Index | LCP | Prefix |
| 10 | - | I |
| 7 | 1 | IPPI |
| 4 | 1 | ISSIPPI |
| 1 | 4 | ISSISSIPPI |
| 0 | 0 | MISSISSIPPI |
| 9 | 0 | PI |
| 8 | 1 | PPI |
| 6 | 0 | SIPPI |
| 3 | 2 | SISSIPPI |
| 5 | 1 | SSIPPI |
| 2 | 3 | SSISSIPPI |

Finally, we draw the suffix trees for each string.





**12.9 (a)** We see that for all i, rank[sa[i]] = i. Thus, rank and sa are inverses of each other. The sa array gives the list of suffixes in alphabetical order. In other words, sa[4] tells you what the 5th suffix is in alphabetical order, and the value being returned is the index of the first letter of the suffix. So, sa[4] = 0 means that if you sort the suffixes in alphabetical order, the 5th one is the suffix that begins with the [0] letter in the word, i.e. the entire word.

y = sa[x] means: x = ordinal position; y = index of first letter of the suffix at that position.

y = rank[x] means: x = index of first letter in some suffix, and y = where this suffix resides in alphabetical order of suffixes

**(b)** Need to show that LCP[rank[i]] == h implies LCP[rank[i+1]] >= h-1.

The reason why we have the big IF statement: if (rank[i] > 0) is because we want to avoid computing LCP[0]. If rank[i] is 0, the 0 means it refers to the first prefix in alpha order.

When we compare substrings at positions i+h and j+h (e.g. h could be 0), i and j point to cells of the string that are likely to be the same letter. This is because j was found to be the next subsequent alpha order prefix.

The reason why we decrement h is that we are going to perform the same substring matching as on the previous iteration except we omit the first character. For example, we may compare "ss" with "ss", not "iss" with "iss".

So, when we compare LCP[rank[i]] and LCP[rank[i+1]], there are two cases. First, if LCP[rank[i]] > 0, then this means that the substring in s starting at i matches another substring in the word. When we advance from i to i+1, the value of LCP[rank[i+1]] will be 1 less than LCP[rank[i]] because the substrings we are matching skip the first corresponding character.

In the other case, we could have that LCP[rank[i]] = 0 and LCP[rank[i+1]] > 0 which could occur when we discover that rank[i+1] matches a substring and rank[i] does not.

**(c)** By definition, the LCP of a suffix is the number of initial characters that match the suffix that immediately precedes it in the alphabetical listing of suffixes. For example, the [1] and [2] suffixes in the suffix array of MISSISSIPPI are IPPI and ISSIPPI. The LCP of ISSIPPI is 1, because only its initial "I" matches the previous suffix.

Inside the loop, we define j to be the starting position of the previous suffix in the list: j = sa[rank[i] - 1]. Then, h counts the number of matching letters, and the result is stored in the LCP array.

**(d)** The algorithm consists of a loop that runs for N iterations. The only doubt that the algorithm might execute in more than O(N) time is the fact that there is an inner while loop. However, the inner loop does not iterate unless h starts out at 0 and we have found new matching substrings.

The total number of times we execute the body of the inner loop is also bounded by N. Therefore, the inner loop contributes O(N) complexity. The rest of the code executes once per iteration, so the entire algorithm executes in O(N) time.

**12.13** See H. N. Gabow, J. L. Bentley, and R. E. Tarjan, “Scaling and Related Techniques for Computational Geometry,” *Proceedings of the Sixteenth Annual ACM Symposium on Theory of Computing* (1984), 135 – 143, or C. Levcopoulos and O. Petersson, “Heapsort Adapted for Presorted Files,” *Journal of Algorithms* 14 (1993), 395 – 413.

**12.23** A linked structure is unnecessary; we can store everything in an array. This is discussed in reference [12]. The bounds become *O*(*k* log *N*) for insertion, *O*(*k*2 log *N*) for deletion of a minimum, and *O*(*k*2 *N*) for creation (an improvement over the bound in [12]).

**12.29** Consider the pairing heap with 1 as the root and children 2, 3, . . . *N*. A *deleteMin* removes 1, and the resulting pairing heap is 2 as the root with children 3, 4, . . . *N*; the cost of this operation is *N* units. A subsequent *deleteMin* sequence of 2, 3, 4, . . . will take total time **Ω**(*N*2)

**12.33** a. No