DATA STRUCTURES FOR EFFICIENT IMPLEMENTATION OF STICKY POINTERS IN TEXT EDITORS*
(Extended Abstract)

Michael J. Fischer & Richard E. Ladner

Department of Computer Science
University of Washington
Seattle, Washington 98195

Technical Report 79-06-08
June 1979

*Research supported by NSF Grant No. MCS77-02474.

This paper has been submitted to the 20th Annual IEEE Symposium on Foundations of Computer Science to be held in October 1979.

Introduction

Text editing is a fundamental activity in computing, yet there has been very little theoretical research directed toward understanding the data structures and algorithms underlying text editors. The principal example of such research is the literature on the pattern matching problem [KMP 77], [BM 77], [FP 74].

In this paper we investigate the problem of implementing sticky pointers in text strings. If we imagine a text string as a linked list of characters, then a sticky pointer is a pointer to a member of the list. So, despite subsequent insertions and deletions in the text string the sticky pointer remains fastened to the character to which it was originally pointing. If the character is subsequently deleted, two natural conventions are for the sticky pointer not to point to the text at all or for it to point to the next character still in the text.

There are several drawbacks to a linked list implementation of text strings. (i) It increases the storage requirement severalfold over a packed representation. (ii) The text must be copied to a buffer for transfer to or from a disk file. (iii) Text searches are restricted to sequential scans through the text, precluding use of efficient algorithms such as the Boyer-Moore string matching algorithm [BM 77]. (iv) Linked list structures tend not to have good paging performance in a virtual memory environment.

We propose a compromise data structure for implementing a text string. A text string is implemented as a doubly-linked list of segments, where each segment is a node which points to a string of characters packed in an array. (This representation was suggested to the first author by R. S. Boyer.)

In such a structure, the text strings in the array are never copied or moved. To handle an insertion into the middle of a segment, the original segment node is replaced by three new segment nodes. The first points to the portion of the original text up to the point of the insertion, the second points to the inserted text (which can be placed at the free end of the character array), and the third points to the portion of the original text following the point of insertion. Deletion is handled similarly by splitting segments.

We use additional structure to handle sticky pointers. A design goal of our structure is that it not be necessary for the system to keep track of all instances of sticky pointers. This permits such pointers to be passed around freely just like any other data. It also precludes various obvious implementations which require updating the sticky pointers when doing insertions and deletions.

Our implementation of sticky pointers maintains a family of trees whose leaves are the segment nodes representing the text string. The trees encode enough of the history of insertions and deletions to enable an old sticky pointer to be brought up-to-date. The details of the implementation are given later. Some of its properties, however, are that insertions can be done in constant time, independent of the length of the text string, the number of editing operations done so far, and the number of pointers. Similarly, deletion within a segment takes constant time, and an arbitrary deletion takes time proportional to the number of segments containing the deleted text. The time to "find" the place in the text string pointed to by a sticky pointer depends only on the number n of insertion and deletion operations done so far. Although the worst-case time is $\Omega(n)$, two sets of reasonable assumptions for average case time give upper bounds of $O(\log n)$ and $O(1)$ respectively.

The worst-case time can be reduced to O(log n) by employing tree-balancing techniques based on single and double rotations like those used in AVL trees [AVL 62] or the dichromatic trees of Guibas and Sedgewick [GS 78]. Since such rotations do not preserve the conditions needed for the basic sticky pointer structure, we add yet more structure to permit the balancing.

Formal Specification of Sticky Pointers

To formalize our notions, let Σ be a finite alphabet of text characters and let Γ be a (possibly infinite) set of sticky pointer objects. An edit object is a pair (w,γ), where $w \in \Sigma^*$ and $\gamma: \Gamma \to \mathbb{N}$ is a partial function such that $\gamma(p) \leq |w|$ whenever $\gamma(p)$ is defined. Intuitively, w is the text string of the edit object, and $\gamma(p)$ is the position in the text indicated by the pointer object p. We think of p as pointing to the place between the $\gamma(p)^{th}$ and the $\gamma(p) + 1^{st}$ characters of the text.

The basic edit operations are now defined on edit objects. Let (w,γ), (w',γ') be edit objects, $p, q \in \Gamma$, and $x \in \Sigma^*$.

1. Insertion.

$(w,\gamma) \xrightarrow{\text{insert}(x,p)} (w',\gamma')$ if

(i) $w' = w_1 x w_2$ where $w = w_1 w_2$ and $|w_1| = \gamma(p)$;

(ii) $$\gamma'(r) = \begin{cases} \gamma(r) \text{ if } \gamma(r) \text{ is defined and } \gamma(r) \leq \gamma(p); \\ \gamma(r) + |x| \text{ if } \gamma(r) \text{ is defined and } \gamma(r) > \gamma(p); \\ \text{undefined otherwise.} \end{cases}$$

Intuitively, insert(x,p) means to insert the string x immediately following the $\gamma(p)^{th}$ character of w.

2. Deletion.

$(w,\gamma) \xrightarrow{\text{delete}(p,q)} (w',\gamma')$ if $\gamma(p) \leq \gamma(q)$ and

(i) $w' = w_1w_2$, where $w = w_1yw_2$, $|w_1| = \gamma(p)$, and $|w_1y| = \gamma(q)$;

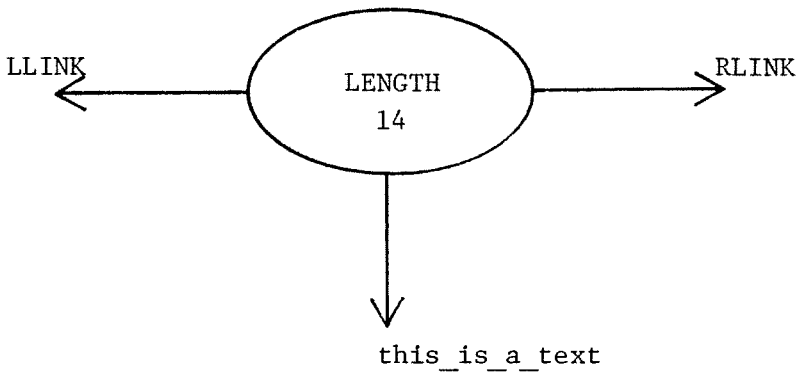
(ii) $$\gamma'(r) = \begin{cases} \gamma(r) \text{ if } \gamma(r) \text{ is defined and } \gamma(r) \leq \gamma(p); \\ \gamma(r) - |y| \text{ if } \gamma(r) \text{ is defined and } \gamma(r) \geq \gamma(q); \\ \text{undefined otherwise.} \end{cases}$$

Intuitively, delete(p,q) means to delete the text between positions p and q.

Other operations like moving a sticky pointer to a new position, searching texts for a pattern, moving text, copying text, finding the i-th character of the text and others could also be defined formally. In most cases the sticky pointers are used to access the text. Any operations that modify the text could be defined using insert and delete so that we concentrate on them and how they affect the ability to access the text using sticky pointers.

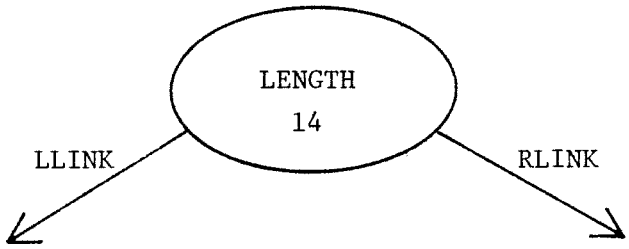
Basic Implementation

A node is a structure with four fields: TEXT, LENGTH, LLINK and RLINK. If TEXT ≠ nil then the node is called a segment node, otherwise the node is called an internal node.



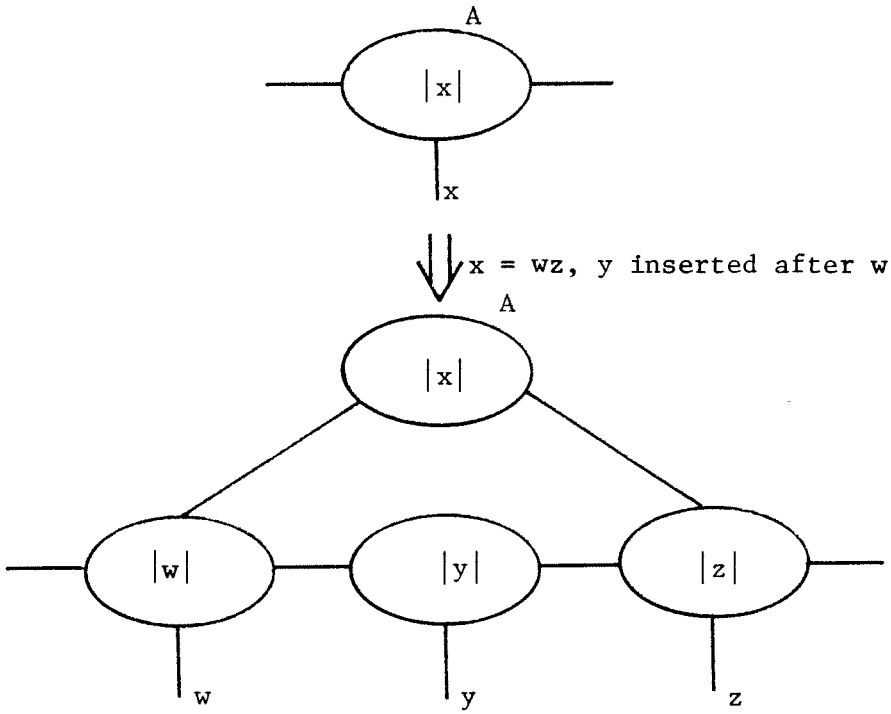
SEGMENT NODE

In a segment node the TEXT field points to a directly addressible text string. The LENGTH field holds the length of the string pointed to by TEXT. LLINK and RLINK point to the previous and next segment nodes, respectively. The complete text string can be read by traversing the segment nodes from left to right.

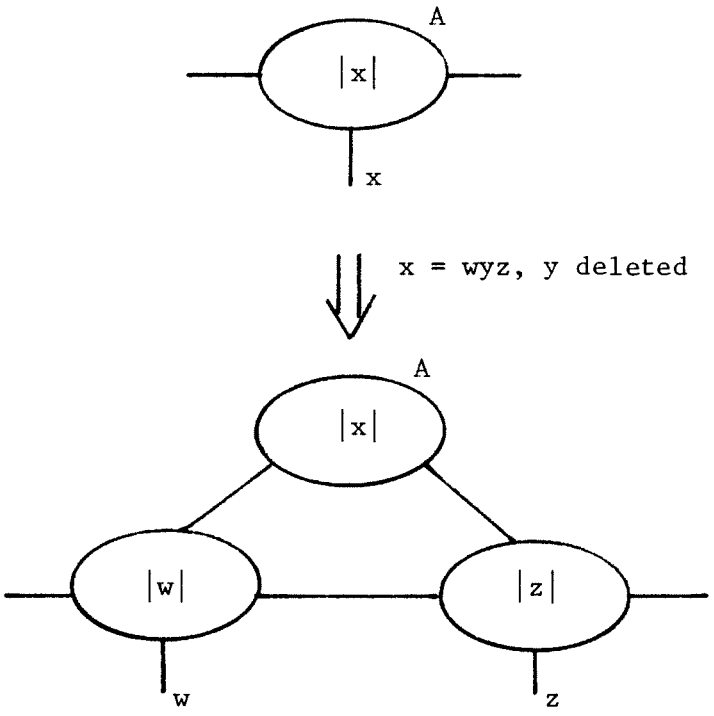


INTERNAL NODE

In an internal node the LLINK and RLINK fields are tree links in a binary tree. Each internal node was originally a segment node. The value it had in its LENGTH field remains when it becomes an internal node. Segment nodes become internal after an insertion or deletion. Iterated insertions and deletions will cause a family of pairwise disjoint trees to be created whose leaves are the segment nodes. An insertion or deletion at the end of a segment will not necessarily force the segment node to be internal. A deletion covering more than one segment is implemented by repeated use of deletion within a segment.

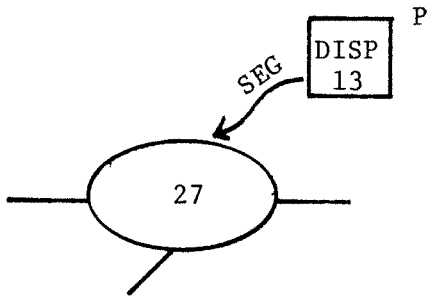


INSERTION



DELETION WITHIN A SEGMENT

A sticky pointer is a structure with two fields SEG and DISP for segment and displacement, respectively. Initially a sticky pointer points directly to a segment node.



this_is_an_exampple_of_text

means

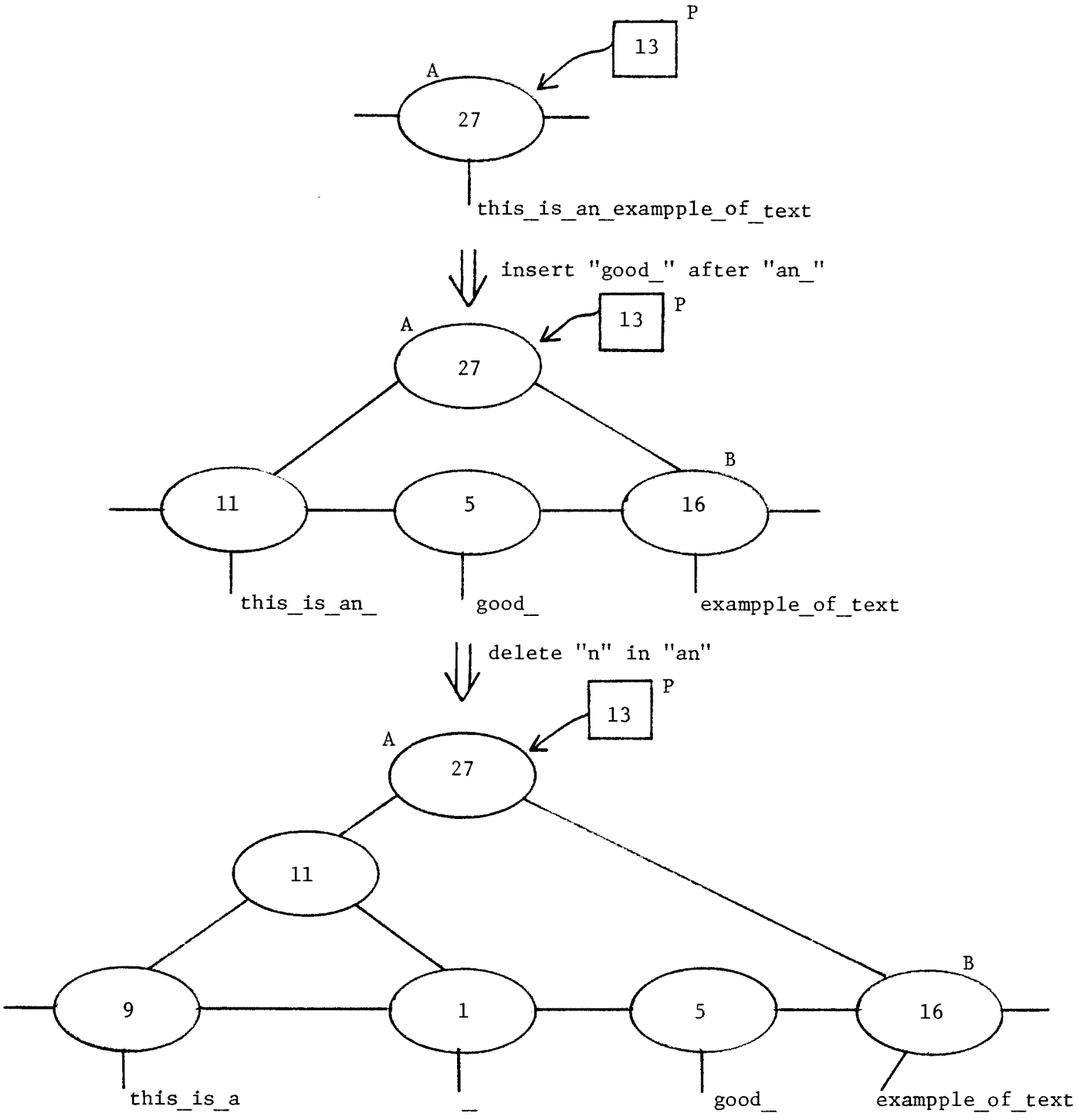
this_is_an_ex ampple_of_text
↑
P

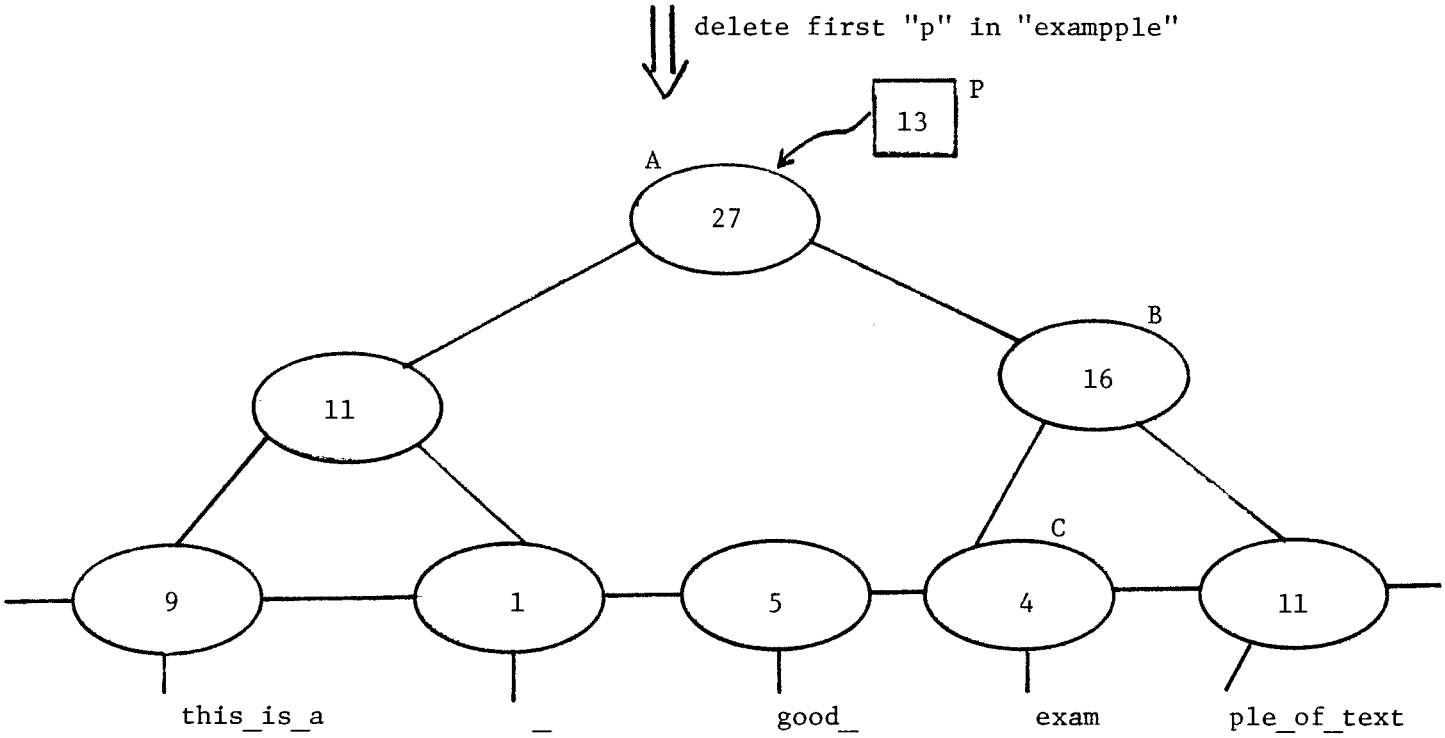
STICKY POINTER

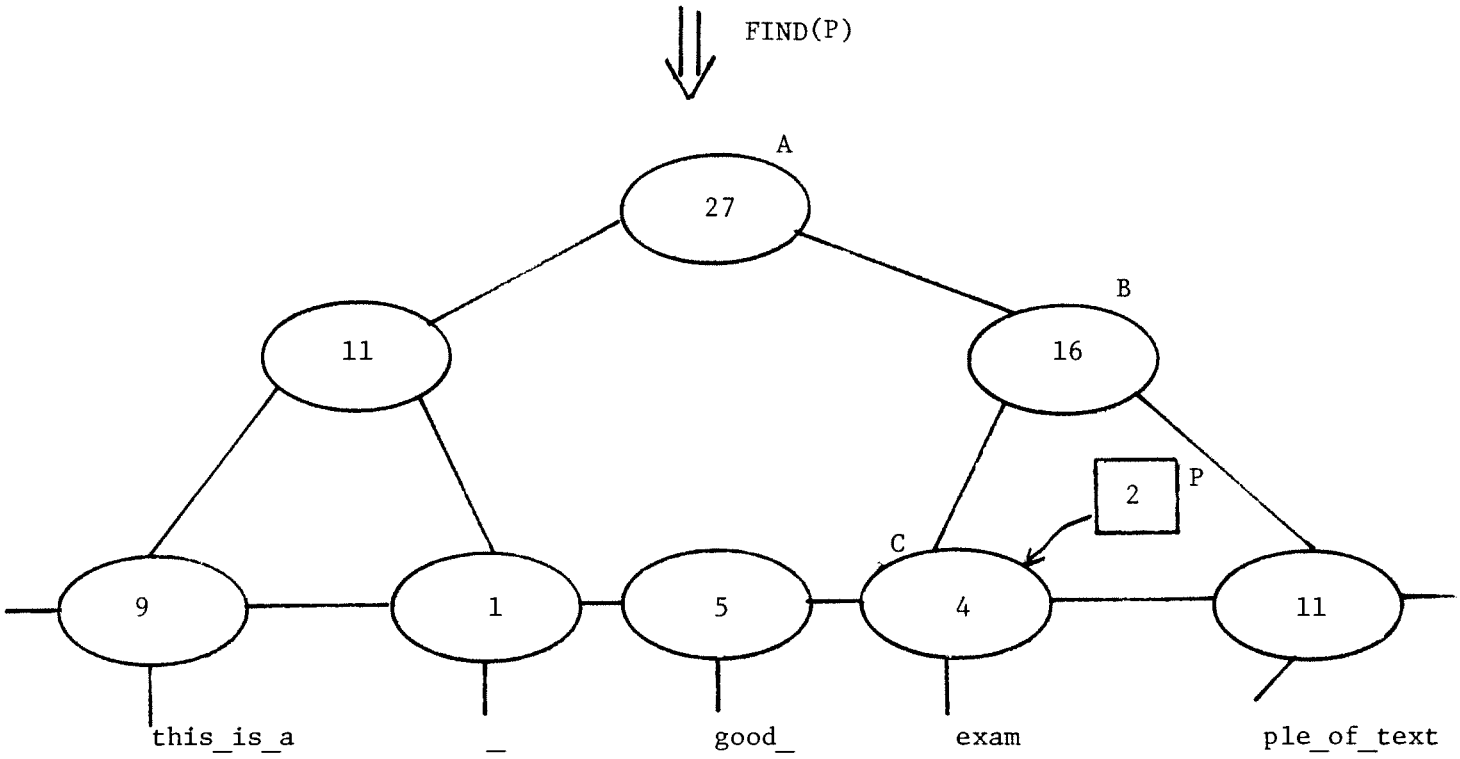
Of course, the segment node originally pointed to by a sticky pointer may become an internal node after a sequence of inserts and deletes. When a sticky pointer is accessed the text associated with it must be found. To do this we define a procedure FIND(p) which given a sticky pointer p updates the SEG and DISP fields so that p points directly to a segment node. Once p points to a segment node, then the text associated with p can be directly accessed through the TEXT field of the segment node.

```
procedure  FIND(p);
    s ← SEG(p);  d ← DISP(p);
    while  TEXT(s) = nil do
        n ← LENGTH(s);
        m1 ← LENGTH(LLINK(s));
        m2 ← n - LENGTH(RLINK(s));
        case
           d ≤ m1 : s ← LLINK(s);
           m1 < d < m2 : p points to deleted text;
           m2 ≥ d : s ← RLINK(s); d ← d - m2
        end
    end;
    if  d > LENGTH(s) then p points to deleted text
        else [SEG(p) ← s;  DISP(p) ← d]
end
```

Example:


P
13
A
27
this_is_an_exampple_of_text
insert "good_" after "an_"
P
13
A
27
B
11
5
16
this_is_an_
good_
exampple_of_text
delete "n" in "an"
P
13
A
27
11
B
9
1
5
16
this_is_a
_
good_
exampple_of_text


delete first "p" in "exampple"
P
13
A
27
B
16
11
C
9
1
5
4
11
this_is_a
_
good_
exam
ple_of_text

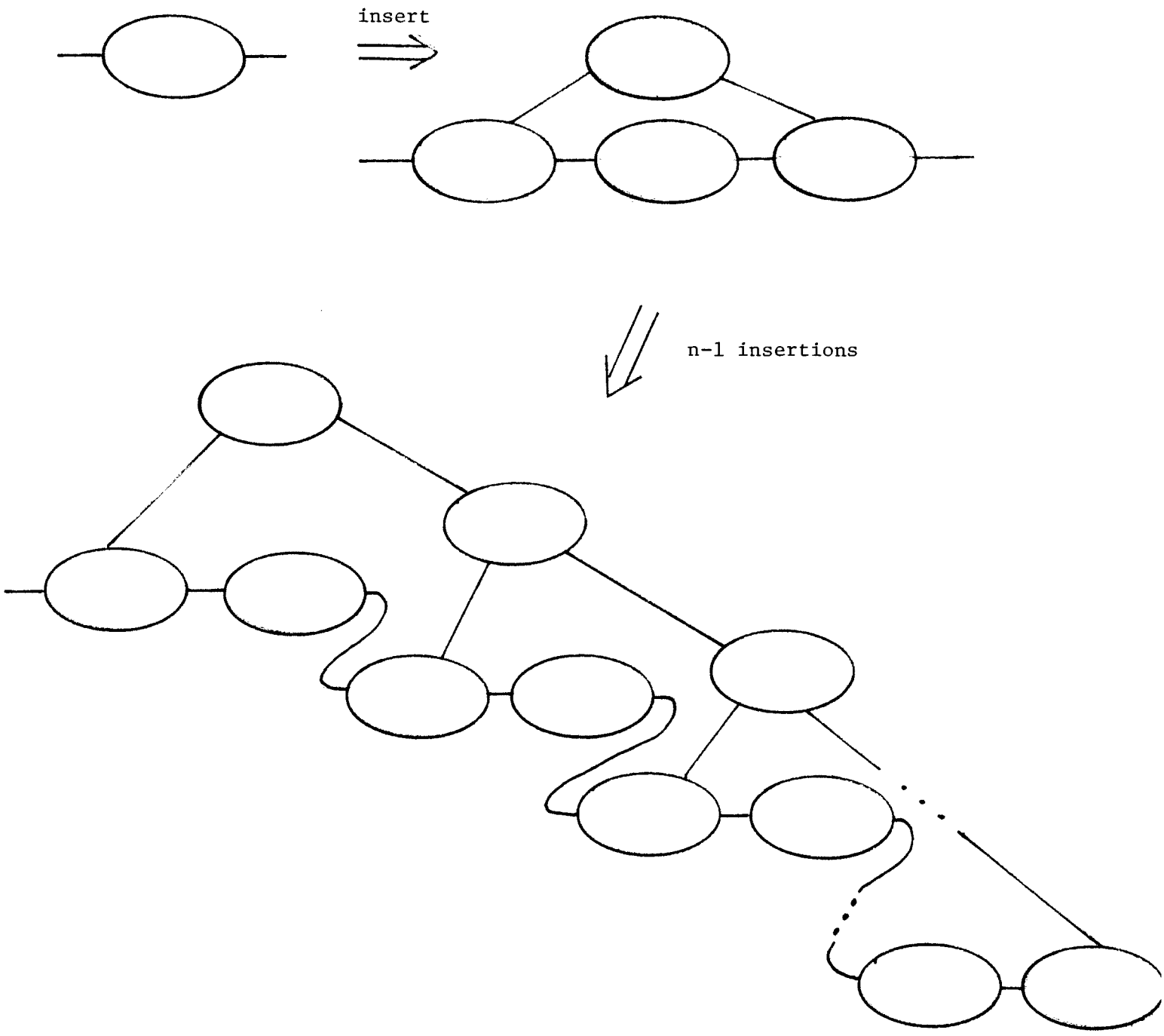

FIND(P)
A
27
B
16
11
P
2
C
9
1
5
4
11
this_is_a
_
good_
exam
ple_of_text

It is important to notice that the sticky pointer P does not need to be touched during the insertions and deletions. It remains attached to the node A until it is needed in some operation. Before it is actually used it is "pushed" down to a segment node using FIND. Should P be the only sticky pointer attached to A, then after FIND(P) is executed, A becomes garbage which could be collected by the storage manager.

Performance of the Basic Implementation

Since we have FIND we can assume that insertion and deletion always occur directly at a segment node and consequently have constant cost. We investigate the cost of FIND after n insertion and deletion operations.

In the worst case FIND can cost $\Omega(n)$ for consider for consider the case of n insertions into an initial text where each insertion occurs in the right-most segment.



n insertions into right-most segment.

A FIND in the right-most segment requires a traversal of n internal nodes.

The average performance of the basic implementation is difficult to assess because of the indeterminability of a random editing sequence. To gain some understanding of average behavior we examine two extreme situations. 1) Small-length insertions and deletions are made randomly in a very long segment and 2) Moderate to long-length insertions are made randomly in an initially empty text.

To idealize the first situation we assume that all insertions and deletions are made in the original text, that the probability of an insertion or deletion in a particular segment is proportional to the length of the segment, and that the sticky pointer we are accessing after n inserts and deletes is in the original text.

Theorem 1. The expected cost of FIND after n small insertions and deletions in a very long text is $O(\log n)$.

Although Theorem 1 is intuitively true because we are forming a random binary tree, we give a careful analysis of the expected external path length of a binary tree with weighted leaves which is formed by n splitting operations. The probability of a leaf being split is proportional to its weight and splitting it consists of giving it two children with random weights which add up to the weight of the split node. The expected path length is exactly $2(H_n-1)$, where H_n is the n-th harmonic number $= 1 + 1/2 + \ldots + 1/n = O(\log n)$.

To idealize the second situation we again assume all the insertions are the same size and that the probability that an insertion is made in a segment is proportional to the length of the segment. The difference here is that the probability of inserting into a previously inserted segment is no longer negligible.

Theorem 2. The expected cost of FIND after n approximately equal size insertions in an initially empty text is O(1).

In this case we give a careful analysis of the expected external path length in a forest with weighted leaves which would correspond to those in our data structure after n approximately equal size insertions into an initially empty text. The expected path length (EPL) is given by

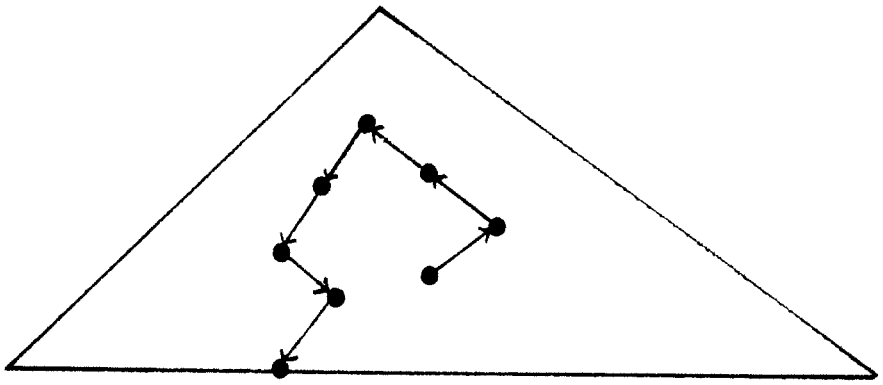
$$EPL(n) = \sum_{m=0}^{n-1} 2(H_{m+1}-1)\ g(m,n)$$

where g(m,n) is the probability that a tree in the forest has exactly m+1 leaves. It can be shown that EPL(n) $\leq$ 1 for all n.

We examined these two situations because they were interesting and analyzable. We suspect that the actual behavior of the algorithm lies somewhere between these extremes.

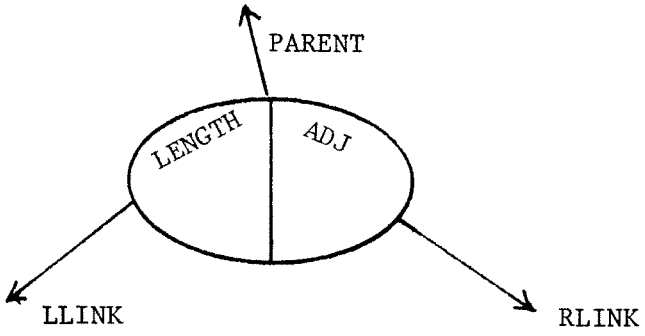
Balanced Tree Implementation

A scheme for balancing binary trees based on single and double rotations [AVL 62] [GS 78] can be used to keep the trees in the data structure balanced. However, the rotations destroy the properties needed by the simple mechanism for "finding" the segment associated with a sticky pointer so additional data fields and a more complicated algorithm are needed. A find now will traverse up the tree to some node, then down the tree to the "correct" segment.



FIND in a balanced tree.

We need extra fields PARENT, ADJ (adjustment) as well as any additional fields needed by the balancing algorithm.



NODE IN BALANCED TREE IMPLEMENTATION

The PARENT field is used to move up the tree and the ADJ field is used to help decide which way to go, up, left or right. Surprisingly, this local information is enough to figure out where to go yet not so much that it cannot be updated when rebalancing. The details of traversal and updating are left to the full paper. The balancing unfortunately adds to the cost of insertion and deletion so that they can no longer be done in constant time. Since inserts and deletes are always preceded by finds in either of our implementations the loss of constant time for these operations may not be significant.

Theorem 3. The worst case cost for FIND, INSERT, and DELETE after n insertions and deletions in the balanced tree implementation is O(log n).

REFERENCES

[AVL 62] Adel'son-Vel'skii, G.M., and Y.M. Landis. "An algorithm for organization of information." Dokl. Akad. Navk SSSR 146, 263-266 (in Russian). English translation in Soviet Math. Dokl. 3 (1962) 1259-1262.

[BM 77] Boyer, R.S., and J.S. Moore. "A fast string searching algorithm." Communications of the ACM 20, No. 10, (1977) 762-772.

[FP 74] Fischer, M.J., and M.S. Paterson. "String-matching and other products." in Complexity of Computation, volume VII of SIAM-AMS Proceedings, American Math. Soc., Providence, R.I., 1974, 113-126.

[GS 78] Guibas, L.J., and R. Sedgewick. "A dichromatic framework for balanced trees." Proceedings of 19th Annual Symposium on Foundations of Computer Science. (1978), 8-21.

[KMP 77] Knuth, D.E., J.H. Morris, Jr., and V.R. Pratt. "Fast pattern matching in strings." SIAM Journal on Computing 6, No. 2, (1977) 240-267.