DISSERTATION

A SIMPLE AND DYNAMIC DATA STRUCTURE FOR PATTERN MATCHING IN TEXTS

Submitted by

Sung-Whan Woo

Department of Computer Science

In partial fulfillment of the requirements

For the Degree of Doctor of Philosophy

Colorado State University

Fort Collins, Colorado

Summer 2011

Doctoral Committee:

Advisor: Ross M. McConnell

A. P. Willem Bohm
Tim Penttila
Yashwant K. Malaiya

ABSTRACT

A SIMPLE AND DYNAMIC DATA STRUCTURE FOR PATTERN MATCHING IN TEXTS

The demand for a pattern matching algorithm is currently on the rise from diverse areas such as string search, image matching, voice recognition and bioinformatics. In particular, string search or matching algorithms have been growing in popularity as they have been applied to areas such as text editors, search engines and bioinformatics.

To satisfy these various demands, many string matching methods have been developed to search for substrings (pattern strings) within a text, and several techniques employ the use of tree data structures, deterministic finite automata, and other structures.

The problem of string matching is defined by finding all location of a pattern string P within a text T, where preprocessing of T is allowed in order to facilitate the queries. There has been significant success in finding a pattern string in $O(m+k)$ time, where m is the length of the pattern string and k is the number of occurrences, using data structures that can be constructed in $O(n)$ time, where n is the length of T.

Suffix trees and directed acyclic word graphs are such data structures. All of these data structures index the searched text in $O(m+k)$ time. However, the difficulty of understanding and programming the construction algorithms is rarely mentioned. Also, they have significant space requirements and take $\Theta(n)$ time to update even if one character of T is changed.

To solve these problems, we propose the *augmented position heap*. It can be built in $O(n)$ time, and can be used to search a pattern string in $O(m+k)$ time. Most importantly, when a block of j characters are inserted or deleted, the asymptotic updating it when a text is modified is $O((h(T)+j)h(T))$, where $h(T)$ is the length of the longest substring X of T that occurs at least $||X||$ times in T, where $||X||$ is the length of X. For texts arising from practical applications, $h(T)$ is typically slowly growing function of $||T||$; for a random text T, its expected value is $O(\log n)$.

Another issue in data structures that must be addressed is space requirement. The most space efficient data structure for string search is the suffix array, which uses $2n$ words and supports searches in $O(n\log n+m+k)$. A compact representation of the position heap proposed in this thesis also takes $2n$ words, but can be updated in $O((h(T)+j)h(T))$ time, but takes $O(m^2+k)$ time for a search. The best bound known bound for updating the suffix array or the directed acyclic word graph is $O(n)$, and they both take considerably more space. A compact representation proposed in this thesis for the augmented position heap takes $4n$ words, can be updated just as efficiently as the position heap, and takes $O(m+k)$ time for a search.

ACKNOWLEDGMENTS

Completing a Ph.D. degree is truly good experience, and I would not have been able to complete this journey without the aid and support of countless people over the past years.

First of all, I would like to thank to my advisor, Dr. Ross M. McConnell and must first express my deep and sincere gratitude towards him for his guidance, encouragement and support during four years of my Ph. D. study. Without his encouragement and direction, this work would not have been possible. I would like to thank my committee members Dr. Willem Bohm, Dr. Yashwant K. Malaiya and Dr. Tim Penttila for taking their precious time to review my dissertation and for giving insightful comments on my work.

I would like to thank to Dr. Sandra Schleiffers for her help and continuous encouragement on my teaching assistance during two years, as well as the rest of the Colorado State University Computer Science department for their continued dedication to my education.

I would like to thank my father and mother for all of the love, support, and encouragement. I also would like to thank some of my fellow students Nissa Osheim, Nathan Lindzey and Willam Springer.

Fianlly, many thanks to my patient and loving my wife Jina Lee, and my two children Warren Woo and Elizabeth Woo, who have been a great source of strength all through this work.

DEDICATION

This disseration is dedicated

to my wife Jina,

to my son Warren,

to my daughter Elizabeth

and to my parents.

TABLE OF CONTENTS

LIST OF FIGURES

Chapter 1

Introduction

As time goes by, computational devices are becoming more powerful and the amount of information that is managed and stored by these devices is also rapidly increasing. To filter out useful and valuable information from enormous text, proper algorithms and data structures for pattern matching are demanded. That is, we need an efficient approach, given a (long) text T of length n and a (shorter) pattern P of length m of finding all locations in T where P or an approximation of P occurs as a substring of T.

The pattern matching algorithm provides the fundamental idea for wide areas of science and information processing such as bioinformatics [50, 29, 1, 28], data scanning (e.g. virus scanning [48]), web search engine [10], image searching [4, 5, 2, 32, 27], plagiarism detection (parameterized matching [25]) and data (information) mining technology [51, 49]. Among these applicable areas, this dissertation deals with the string matching problem.

The string matching problems can be classified according to various criteria. The *exact matching problem* is the which is the one have described, where a location in a *text T* is reported if and only if a *pattern P* occurs as a substring at that location. This contrasts with an *approximate matching problem*, where a location is in T reported if some approximation to P occurs there. Searching without preprocessing of T is known as ***online*** string matching; the user is not required to announce anything about the text in advance of a query. This contrasts with the approach of preprocessing the text, which is only interesting if something is known about the text in advance of series of queries. Finally, allowing small edits on T between queries is known as searching on a *dynamic* text, as opposed to searching on a *static* text. The distinction between searching a dynamic and a static text is only of interest when the text is preprocessed.

A naive way to solve the exact online problem is to shift P to each position of T and test whether it is a match. This takes $O(nm)$ time where n denotes the length of T and m denote the length of P, since at each position $\Omega(m)$ characters might need to be compared. This has been improved to $O(n)$, which is the best that can be hoped for when there is no preprocessing of T.

A key insight in improving this $O(nm)$ bound was the observation that m is usually much smaller than n, so preprocessing of the pattern can pay even if it is only used once. This is the approach of the *Knuth-Morris-Pratt algorithm* [40, 47], which was the first to achieve an $O(n)$ bound.

When the text can be preprocessed, a data structure can be produced to facilitate queries. The ***suffix tree*** [18, 45, 59, 61] and the ***suffix array*** [43, 39, 38] are among the most popular. Both of these data structures can be built in $O(n)$. The advantage of the data structure of a preprocessed text is that each query then takes $O(m+k)$ time for the suffix tree and $O(m+\log n)$ time for the suffix array, where k is the number of locations of P in T that must be reported. What is striking about this latter bound that the length n of the text is irrelevant to the time for a query. The ***Directed Acyclic Word Graph (DAWG)*** [44, 8] also can be used in the same time bound as the suffix tree.

The approximate matching problem also looks for a pattern P within T. However, it allows a limited number of mismatches between a pattern string and a text under ***Hamming distance***. Usually, this problem is defined by string matching with k differences. Thus, we need to find all substrings that have different distances, at most given k. The ***dynamic programming*** is a classic solution for this problem since it asks to optimize a relation between two inputs, a pattern string and a text. The longest common subsequence of two strings [31, 42, 7] and a weighted string matching [46, 34] are the sub-categories of this problem.

In many cases, approaches to the approximate string matching problem adopt the scheme of the exact string matching algorithms. Jokenen and Ukkonen [37] propose the first algorithm with a preprocessed data structure for the approximate string matching problem. Their algorithm uses the DAWG and the dynamic programming. However, this algorithm's query time depends on a length of a text. Ukkonen improve their previous algorithm [37] and suggests the first algorithm that depends on m and k [60]. This algorithm applies the dynamic programming over the suffix tree. Trinh *et al.* [33] import the ***suffix array*** for this problem and Russo *et al.* [53] introduce ***compressed suffix trees and array*** that are also applicable for this problem.

There continue to be many issues that relate to string searching problems, such as ***parameterized string matching*** [52, 26], ***multiple pattern searching*** [3], ***multiple approximate string matching*** [14], ***compressed pattern matching problem*** [6, 19].

A big drawback of these data structures is that when the text is edited, we have to rebuild them from scratch, even if the editing operation involves only a single character. This means that we have to spend another $O(n')$ time to correct the data structures after the modification of the text where n' is the length of text after it is edited. The text must therefore be considered to be static. One type of dynamic case where a *set* of texts must be searched for P, and an entire text may be inserted to or deleted from the set; efficient editing within a single text is not supported. More than 80 string matching algorithms have been proposed since 1970 [20]. Most of these algorithms only participate in finding pattern strings within a text. A few algorithms consider the modification of a text. The ***dynamic suffix array*** [22], the ***string B-tree*** [23] and the ***generalized suffix tree*** [24, 21]. However, these concepts are different than the concept that will be discussed in this dissertation since these data structures are for word-based instead of character-based (inserting or deleting one or more words does not affect other words that are stored in the data structure). The ***border tree*** [30] also supports the arbitrary modification of sequential text. However, this algorithm's search time varies depending upon the number of modifications. The border tree takes $O(n)$ for preprocessing and each edit operation takes $O(\log n^i)$, where i is the number of modifications. Finally, this algorithm's search time depends upon the number of modifications which result in $O(m + k\log i + i\log m)$ time.

Salson *et al.* [55, 56, 54] have proposed the dynamic algorithm that modifies the ***Burrows-Wheeler transform*** [9] and the suffix array in $O(n)$ worst-case time. This worst time bound is the same time bound as reconstructing suffix array. However, they claim that their algorithms provide the better time bound in practice.

1.1 Research Overview

Ehrenfeucht and McConnell have proposed the *position heap* [44] which takes $O(n)$ time to build, but requires $O(m^2 + k)$ time for a query. An advantage of the data structure is that it can be updated after insertion or deletion of a block of characters in $O(ch(T) + h^2(T))$ time, where $h(T)$ is the length l of the longest substring of T that occurs at least l times and c is the number of blocks. This is the starting point of the new results in this work.

This dissertation deals with the exact string matching problem on static and dynamic texts by improving on the position heap. We deal with several issues: our improved *augmented position heap*, which allows queries to take $O(m+k)$ time instead of $O(m^2+k)$ time; *linear-time* $(O(n))$ construction of the augmented position heap; how to update the augmented position heap when the text is edited.

Traditionally the data structures for pattern matching problems occupy a large block of memory. If the size of the data structure exceeds the memory limitation, the data must be swapped into the virtual memory [15]. This virtual memory technique delays the response times. The suffix array theoretically requires n integers. Any offline data structure cannot beat this space requirement. But it has a slow query time. we reduce the memory space for the position heap and the augmented position heap with the same construction and query time. We define this data structure as *compact representation* of the position heap using an array, The compact representation of the position heap requires $2n$ integers to represent the position heap and $4n$ integers to represent the augmented position heap.

1.2 Structure Overview

The next section introduces related data structures that are fundamental algorithms for string searching problems. Following sections show the property of the position heap and a query with the position heap Chapter 4 describe the augmented position and how query algorithms works with this data structure. In Chapter 3 and 4, we show two query cases. The first case shows a pattern string is shorter than the height of the position heap, and the second case shows a pattern string is longer than the height of the position heap. with this data structure. Chapter 5 show the linear time algorithms to construct the position heap and the augmented position heap. The space efficient data structure for the position heap and the augmented is illustrated in chapter 6. In Chapter 7, dynamic texts and maintaining the position heap are illustrated. We also represent the space-efficient data structure for position heap. Chapter 8 concludes the dissertation and offers the future work.

Chapter 2

Related Work

The classic data structures for this field are the suffix tree and the suffix array. DAWG also has a competitive time complexity with these data structures. Linear time algorithms to constructing the suffix tree and the DAWG are more complicated than the suffix array's. However, the suffix tree and DAWG provide a better time bound for a query. In this section, classic and fundamental linear time data structures for string searching are illustrated.

2.1 The Suffix Trie

The suffix trie provides a basic idea of the suffix tree. It contains all possible suffixes of a text *T* in a tree data structure. In the suffix trie, each edge is labeled with a character and each path, from the root to a leaf or an internal node, represents a substring of the text.

2.1.1 Properties of the Suffix Trie

The suffix trie is a tree whose edges are labeled with edges. Each suffix of the text is the sequence of labels on some path from the root to a leaf. For each node and each letter *c* of the alphabet, there is at most one edge from the node to a child that is labeled *c*. When two suffixes share a common prefix, the common prefix is the sequence of edge labels on the

Position	Suffix
1	aabcabcaac$
2	abcabcaac$
3	bcabcaac$
4	cabcaac$
5	abcaac$
6	bcaac$
7	caaa$
8	aac$
9	ac$
10	c$

Figure 2.1: All suffixes for the text *“aabcabcaac$”*

shared part of the paths corresponding to both of the suffixes. Figure 2.1 shows all suffixes for text *“aabcabcaac$”* where $ represents the terminator of the text and its corresponding edges and nodes in the suffix trie. The $ symbol is a unique character that only appears once at the end of a text and makes each substring unique. Figure2.2 shows the example of suffix trie for Figure 2.1.

As we can see in Figure 2.2, the suffix trie can have more nodes than the length of a text and contains all of its suffixes. It has exactly n leaves since each path from a root to a leaf represents one of suffixes in a text . To construct the suffix trie, at most n^2 nodes are needed because n nodes are necessary for the first position and at most $n-1$ nodes for the second position and so on. If the text consists entirely of different characters (the length of the text is the size of alphabet), they must be installed in such way that each character of each suffix consumes one unit of time. Thus, the summation of installation time is $\sum_{i=1}^{n} i = \frac{n(n+1)}{2} = O(n^2)$. We also consider the instance in which just one character occurs n times in a text, such as $T = a^n =$ “*aaaa... aaa*”. In this circumstance $O(n^2)$ is also needed to build the suffix trie, since $O(n^2)$ comparisons are needed to install into different

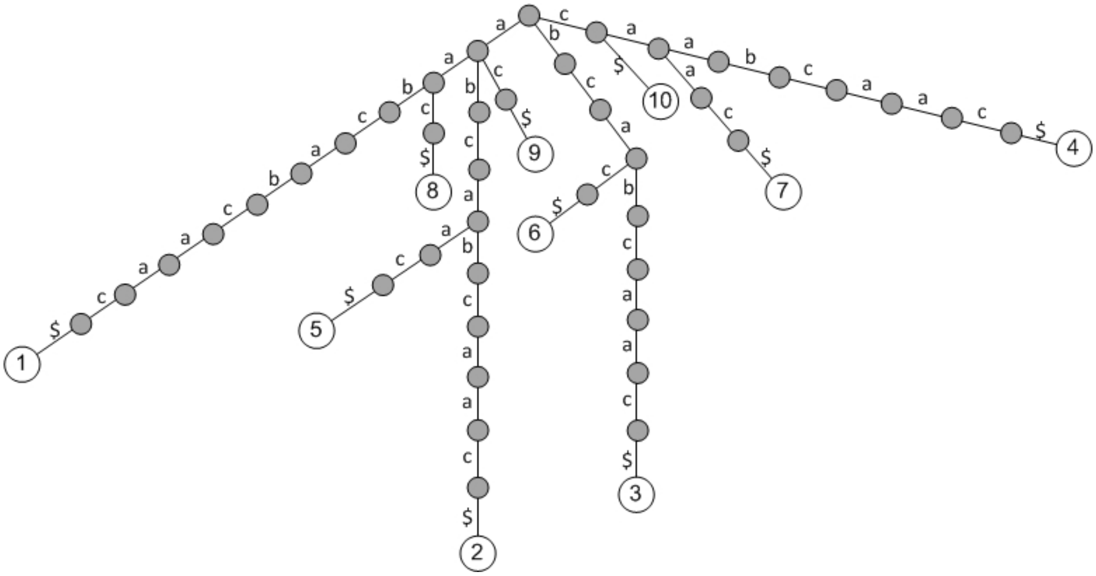


Figure 2.2: The suffix trie

positions.

2.1.2 Query Time with the Suffix Trie

The search for a pattern string in the suffix trie takes $O(m+k(n-m))$ when the alphabet size is constant or $O(m|\sum|+k(n-m))$ when the alphabet size $|\sum|$ is too large to be ignored. $(n-m)$ is the parameter for visiting each leaf node under a certain node. If a pattern string exists in a text, the suffix trie can return position numbers in which the pattern string has occurred. Otherwise, the suffix trie returns null value. For example, looking for the pattern-string *"bca"* in Figure 2.2, start from the root and search the edge which is labeled *'b'* and then transit to the corresponding node. Repeat searching the edges that correspond to a character of the pattern string until there are no more transitions, or until the end of the pattern has been reached string (the depth of the suffix trie matches the index of the pattern string). At the end of the pattern string, simply return the leaves' positions under a node where stopped, since the prefix of these leaves contains the pattern string as the prefix. For instance, the position numbers (3, 6) are leaves and both nodes' prefixes are *"bca"* in Figure 2.2.

If the alphabet size $|\sum|$ is not constant, the search time takes $O(m|\sum| + k(n-m))$, since one transition decision must be made at each node (some nodes must have maximum $|\sum|$ number of transitions). However, we can ignore the $|\sum|$, since the alphabet size is the constant in the real world. If $|\sum|$ cannot be ignore, the hash table can be used to eliminate $|\sum|$ factor.

2.2 The Suffix Tree

A suffix trie is a tree data structure that stores all possible suffixes of a text. A suffix tree is a well-known tree data structure that is a compacted suffix trie. It has a fast pattern-string search algorithm $O(m+k)$ with a linear construction time. Weiner [61] proposed the original algorithm for the suffix tree built in $O(n)$. McCreight [45], Ukkonen [59] and Farach [18]suggested the different ways to construct the suffix tree in $O(n)$.

The next subsections demonstrate how to construct the suffix tree in linear time with the suffix link and the suffix tree's properties.

2.2.1 Constructing The Suffix Tree in Linear Time

The suffix tree represents a compacted suffix trie. The suffix tree's edges concatenate the edges' labels so that a node has only one child in its suffix trie. This also means that every internal node in the suffix tree has more than two children. Reducing redundant nodes in this way saves construction time and space. Before illustrating the linear construction time, the suffix tree that has exactly the same information as the suffix trie is shown. The suffix

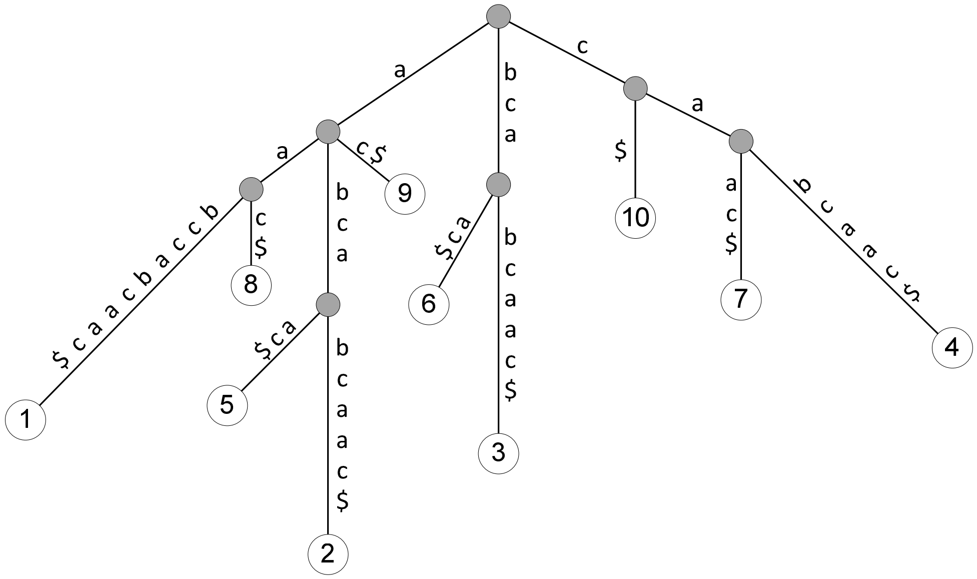


Figure 2.3: The suffix tree

tree shown in Figure 2.3 is derived from Figure 2.2. As we can see in Figure 2.3, the suffix tree has exactly n leaves that are the same as the suffix trie. However, it has at most $2n-1$ total number of nodes. The number of suffix tree's nodes is definitely less than the suffix trie. Like the suffix trie, each leaf corresponds to one of the positions in the text. Each node of the suffix tree has at least two children, except leaves. The concatenation of edges reduces the number of edges and nodes to at most $2(n-1)$ edges and $2n-1$ nodes. Thus, the path from the root to a node is defined as the concatenated edge label. For example, the path label *"caac$"* of the node v is from the root to the leaf node 7 in Figure 2.3. Also, leaves under the node v contain the same prefix. For example, we can reach the node v with the path labeled *"aa"* and leaves 1 and 8 under the node v contain the same prefix *"aa"*. However, the number of characters on the edges are still at most $\frac{n(n+1)}{2}$ because characters of each suffix in the text must be represented in an edge of the suffix tree.

The naive method mentioned in the previous section requires $O(n^2)$ time and space to build the suffix trie since we must install $O(n^2)$ nodes. $O(n^2)$ nodes means that there are $O(n^2)$ characters to be installed. When we reduce the characters (nodes) to be labeled in the edges or to be substituted with other properties to archive $O(n)$ space, the suffix tree can be built in linear time since there are $O(n)$ nodes, edges and labels. Instead of labeling characters on each edge, the labels of each edge are replaced with the pair of positions *(a, b)* where *a* is the beginning of the substring and *b* is the end of the substring. These beginning and ending positions directly point to the location of the text. For example, in Figure 2.4, the edge-label (1,1) represents the substring *'a'* and the edge-label (6,11) represents the substring *"bcaac$*". Also we note that an edge from each leaf to its parent needs only a start position because any path from a leaf to its parent is the suffix of the text (the end position is always the end location of the text). When a leaf is created once, this leaf cannot be turned to a internal node. Finally, the total number of edge-labels are shrunk from $O(n^2)$ to $O(n)$ (the total number of edge-labels are at most two times the number of edges that is $O(n)$).

Now the build time can be speed up to $O(n)$ time since there are $O(n)$ nodes, edges and edge-labels (installing one object per one iteration). Weiner [61], McCreight [45] and Ukkonen [59] used the suffix tree's properties mentioned above. Weiner's algorithm starts to build the suffix tree from the shortest suffix to the longest suffix. McCreight's algorithm builds the suffix tree from the longest suffix to the shortest suffix. And Ukkonen's algorithm incrementally builds the suffix tree. For example, with the text *"aabcabcaac$"*, Weiner's algorithm installs *'$'* first, *"c$"* second, and so on, through *"aabcabcaac$*". McCreight's algorithm installs *"aabcabcaac$"* first, *"abcabcaac$"* second, and so on, through *'$'*. Ukkonen's algorithm installs *'a'* in the first position and then *'a'* in the second position and so on. Even though these three algorithm use different approaches, the resulting suffix

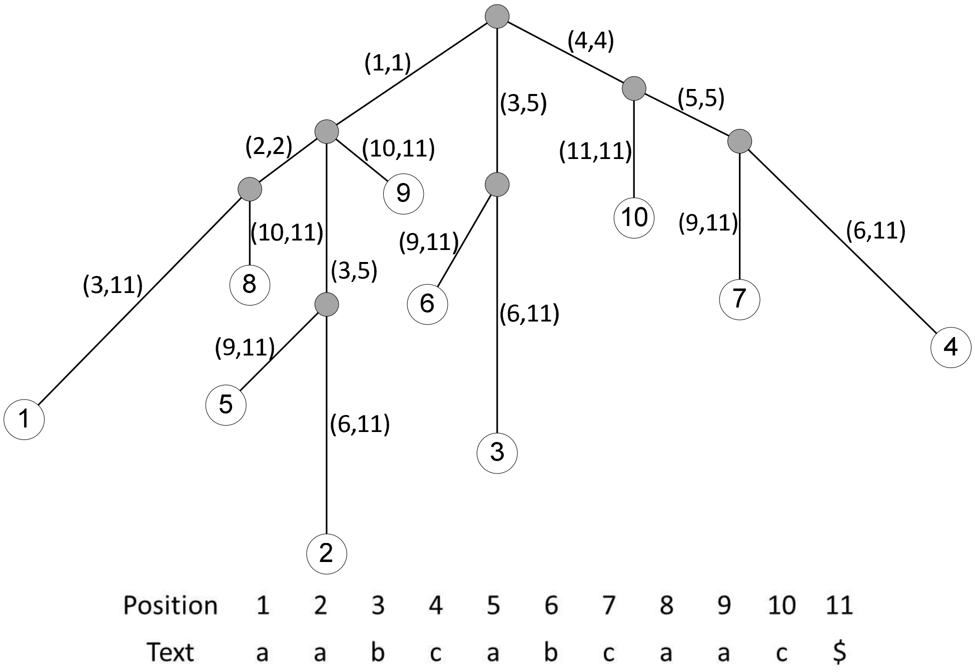


Figure 2.4: The suffix tree with concatenated edge labels

trees are identical.

The *suffix link* is the critical component needed to build the suffix tree in linear time. The three algorithms above use the suffix link (Weiner calls it *"finding* $Head(i)$*"*) and the suffix tree's properties to reduce the construction time bound. The suffix link is the longest common prefix between a suffix to be installed and the current suffix tree, and provides the installation shortcuts. Using the naive method, it is started from the root of the suffix tree. Using the suffix links, however, we can start from a internal node. The suffix link connects one internal node to another internal node such that an internal node v has the path-label $\chi\alpha$ from the root to the node v and an internal node v' has the path-label α that is the longest common prefix of v', thereby creating the suffix link from v to v'. Figure 2.5 shows the suffix links, indicated by dotted lines, of each internal node in the suffix tree. Note that

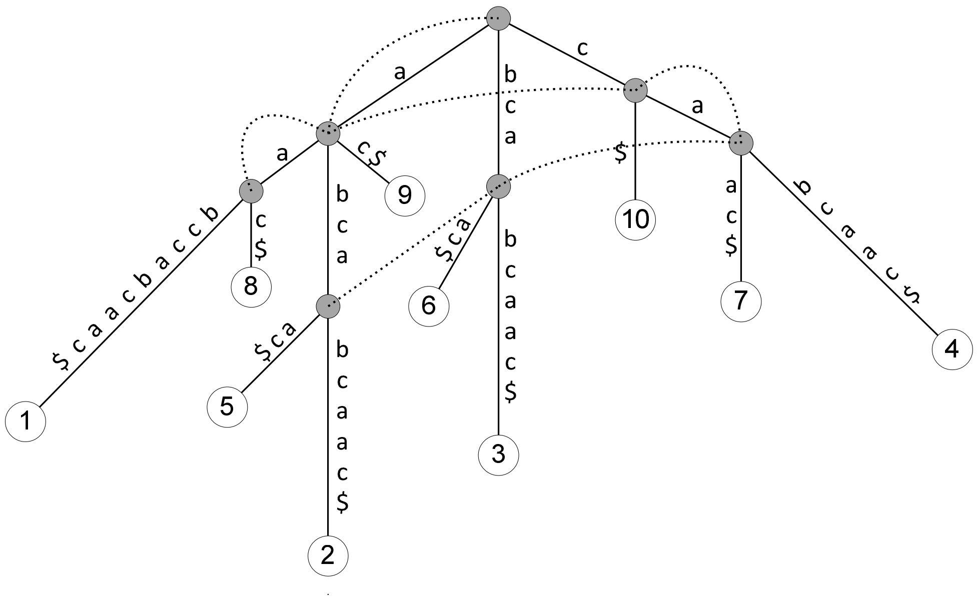


Figure 2.5: The suffix tree with the suffix links
Note: In this figure, the actual characters of the string on the edges in this suffix tree are shown. However, indexes are the same as in Figure 2.4 in its actual implementation.

each algorithm has its own way of creating and maintaining the suffix links: the direction of the suffix links are not assigned in Figure 2.5 because Weiner and McCreight's suffix link directions are opposite of one another.

There are all of the tools needed to create the suffix tree in linear time. The subsections below illustrate how to construct the Weiner, McCreight and Ukkonen's suffix trees.

2.2.2 The Construction Algorithm of Weiner's Suffix Tree

Weiner's algorithm builds the suffix tree from the shortest suffix to the longest suffix. The suffix tree T_i contain‘s suffixes, $a_ia_{i+1} \dots\ a_n$, $a_{i+1}a_{i+2} \dots a_n$, $\dots$, a_n. Weiner's suffix tree T_i is built from the previous iteration suffix tree T_{i+1}. First we find the head using the suffix link and then add the tail $a_{i+|head|-1} \dots\ a_n$ for the suffix tree T_i. By finding the suffix link, it can be started from the node corresponding to this head, instead of beginning from the root. To implement the suffix link, each node (except the leaf nodes) contains additional information about the *indicator vector* and the *link vector*. Each vectors' size is equal to the size of alphabet (also each edge has the indicator vector because the new node inherits the indicator vector from an edge when an edge is split). The indicator vector shows whether a node has a possible suffix link or not for a character a_i. The link vector is the same as the suffix link. These vectors enable Weiner's algorithm to find the suffix link (head) efficiently. Also, the indicator and link vectors in each iteration must be maintained as the suffix tree changes.

In each iteration of Weiner's algorithm searching starts from the leaf that was added in the previous iteration T_{i+1} to extend to T_i and traverse the path from the bottom up until the suffix link (link vector) with a_i is found. For example, in the Appendix, there is an attempt create the suffix tree T_2 for the substring *"abcabcac$"* from T_3. First, traversing toward the root to search the node that has $I_x(a)$ (the indicator vector for a character *"a"* at a node x). However, $L_x(a)$ (the link vector for the character *"a"* at the node x) is null. So, we traverse to the root. $I_{root}(a)$ is true and $L_{root}(a)$ points to the node u. The number of characters between the node x and the root is three *"bca*". Creating the new node y from exactly three characters below the node u. Finally, the tail *"bcaac$"* is added to the suffix tree since the head *"abc*a" exists on the current suffix tree. If all of the nodes' indicator vectors on the path from the leaf to the root for the character a_i are false, the leaf node i is created, since a_i is the new character. So, the indicator and the link vectors are null at the leaf node i.

The last step to complete each iteration of Weiner's algorithm is to update the vectors. There are two cases for updating the vectors. The first trivial case is when the head is empty. The nodes' indicator vector between the root and the leaf node i is updated with the character a_i and it is not necessary to update any link vector since a_i is a new character.

In the second case, the head is not empty. The nodes' indicator vector is between the leaf node i and the node v which contains a link for the character a_i and so is updated. The link vector must be properly updated in this case. The link vector at the node v points to the head which is $a_i\alpha$, where α is the path labeled from the root to the node v.

Finally, in reviewing the time complexity of Weiner's algorithm, as we have demonstrated above, adding one node and one edge takes constant time after finding the head. Finding the head requires traversing up from the last added leaf node. However, in the next iteration, this distance, which climbs upward to find the proper node, will be much shorter than the previous distance since the depth of $a_i\alpha$ is equal to or shorter than the depth of α . At every iteration the suffix tree depth is increased by at most 1 (splitting an edge increases the depth by at most one). It is possible to go many nodes up the tree. However, we go down at most two nodes from the last suffix node. The last leaf node is closer to the root when traversed as many nodes as would be traversed up to find suffix link. Thus, the amortized complexity becomes linear time.

2.2.3 The Construction Algorithm of McCreight's Suffix Tree

McCreight's suffix tree requires less memory than Weiner's suffix tree because it does not use a vector. McCreight's suffix tree is built from the longest suffix (the whole text) to the shortest suffix. The naive way to construct the McCreight's suffix tree takes $O(n^2)$. However, McCreight's algorithm also uses the suffix link to reduce the running time to $O(n)$. The difference is that the direction of the suffix link is the reverse of Weiner's.

2.2.4 The Construction Algorithm of Ukkonen's Suffix Tree

Ukkonen's algorithm builds the suffix tree by adding each character a_i $(i = 1 \ldots n+1)$ at each iteration. Ukkonen's linear time algorithm uses less space than Weiner's algorithm to construct the suffix tree. Ukkonen's algorithm has the advantage of being online. Unlike Weiner's and McCreight's suffix tree, Ukkonen's suffix tree can be extended with a new appended character at the end of the text, by contrast, Weiner's and McCreight's suffix tree must be reconstructed.

Ukkonen's algorithms uses the following three rules to extend the suffix tree with a new character. Thus the suffix extension should obey these three rules.

- Rule 1: *Once a leaf, always a leaf.* If at some point Ukkonen's algorithm creates a leaf node with some position number i, then this leaf node remains the leaf node in all phases. However, the edge label between leaf node and parent must be extended with a new character.

- Rule 2: *Splitting edge.* If no path end of $a_j \ldots a_{i-1}$ starts with character a_i, then we create a leaf with position number i and the edge label is a_i between this leaf node and its parent.

- Rule 3: *Ignoring the existing suffix.* If the path end of $a_j \ldots a_{i-1}$ starts with the character a_i, doing anything, since $a_j \ldots a_i$ is already in the suffix tree.

Figure 2.6 shows an example of extending the suffix tree by applying Rule 1 ~ 3. The upper depiction of Figure 2.6 shows seven leaves with nine characters since a single character *"a"* is in position 9 and the substring *"aa"* for position 8 already exists in the suffix tree. When we append the tenth character 'c', the edges between the seven leaves and their parents are extended with *"c"* by Rule 1, node 8 and 9 are created by Rule 2, and the tenth character *"c"* is ignored by Rule 3. The resulting suffix tree for *'aabcabcaac'* shown in the lower depict of Figure 2.6.

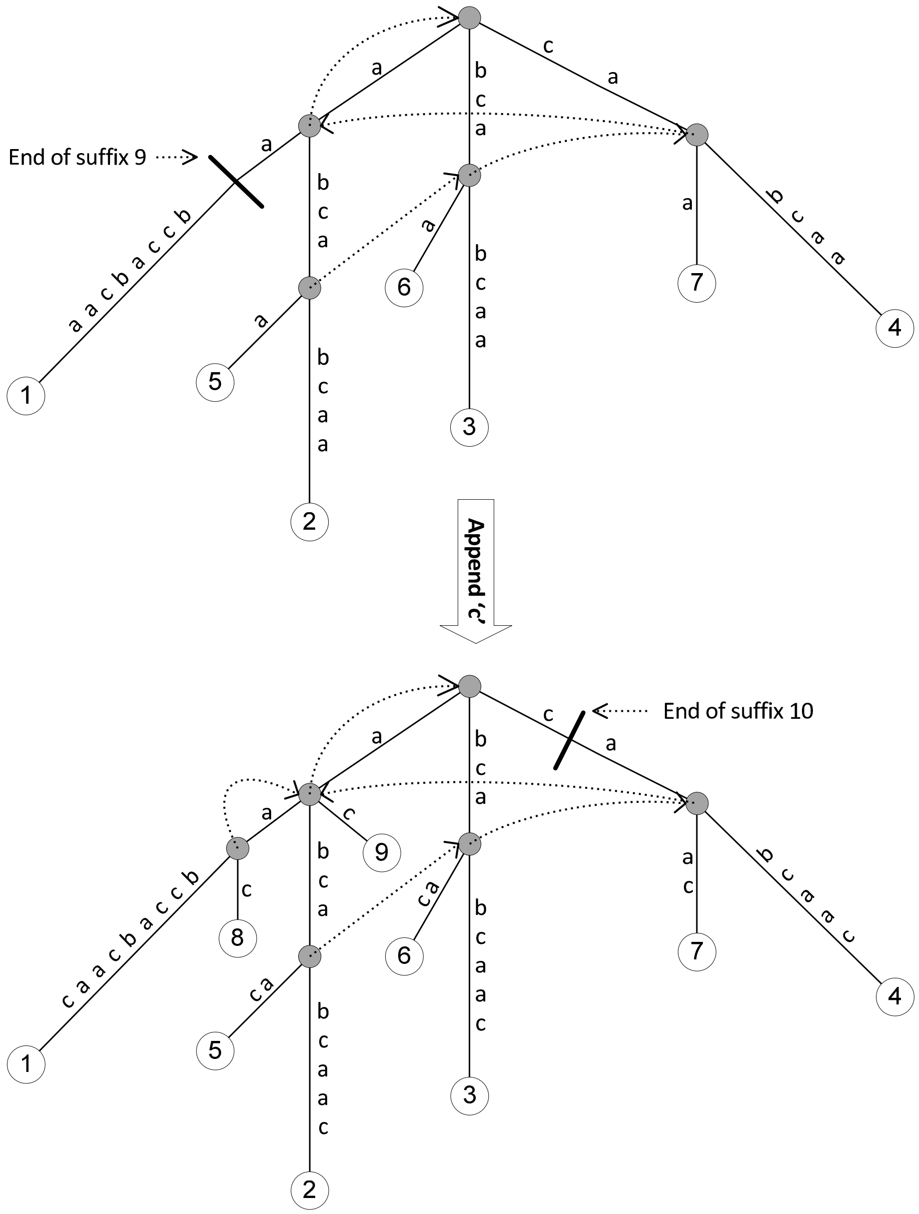


Figure 2.6: Extending the suffix tree from string *"aabcabcaa"* to *"aabcabcaac"*

The naive way to extend a single character needs $O(i^2)$ time in phase i, since there are at most $O(i^2)$ characters and $O(i)$ leaves. To add a new character, $O(i)$ leaves through $O(i^2)$ characters must be visited. However, Rule 1 allows us to skip visiting all of the leaves. Marking only the start position between the leaf node and its parent and putting the end location between the leaf node and its parent when the algorithm is terminated. For example, in Figure 2.6, (2, *end*) is the edge label between leaf 1 and its parent and (6, *end*) is the edge label between leaf 4 and it's parent. When the algorithm is terminated, the parameter *end* is 10. The resulting edge label is (2, 10) and (6, 10). The extension can be done in constant time.

Time bound must be reduced to perform Rule 2 and 3. In each phase, performing Rule 2 and 3 still consumes time since the whole suffix tree is traversed to find a split point or to check existing strings. To efficiently perform Rule 2 and 3, the suffix link is used as stated in the above algorithm states. In phase i, Ukkonen's algorithm locates the suffix $[j \ldots i]$ (in Figure 2.6 the algorithm located at the *end of suffix 9*) and applies Rule 2 (create the leaf node 8 $(i - depthofendof(i-1))$). After creating leaf node 8, the algorithm goes up to the node v to which the suffix link points. (This node v has the suffix link that links to the root.) The algorithm then goes down from this node the number of characters it went up to the node v. (In Figure 2.6 only *"a"* is seen when going up to the node v and to the root using the suffix link then going down one character "a". The resulting node is the node v again). If Rule 2 needs to be applied, the above steps are processed again (in Figure 2.6 Rule 2 was applied and created node 9). Otherwise, Rule 3 is applied (in Figure 2.6 it is stoped at the *end of suffix 10*), stop and then go to the next phase.

The construction time of Ukkones's algorithm is $O(n)$ since at most $2n$ nodes are created (n leaves and at most n internal nodes). If a node is not created, then $O(1)$ time is needed. Splitting an edge also takes $O(1)$. At some point, many nodes may be created in one phase. However, the number of nodes that are created in phase i is not to exceed $2i$ nodes when assumed that is not created any node before phase i. Thus, the construction time of Ukkonen's algorithm is $O(n)$.

2.2.5 Query Time with the Suffix Tree

The three algorithms for building the suffix tree in the previous sections have their own unique way to reduce the construction time. In spite of the three different methods to build a suffix tree, a result of these is the same single suffix tree. All three algorithms produce the same suffix tree like Figure for text *"aabcabcaa$"*. The same query algorithm as suffix trie is also applied to suffix tree. With the suffix tree we can eliminate the $(n-m)$ factor since each leaf node can be reported in $O(1)$.

2.3 The Suffix Array

Another competitive data structure for string searching is a *suffix array* which was introduced by Manber *et al.* [43]. The suffix array stores each suffix of the text in lexicographic order. Manber's suffix array can be built in $O(n\log n)$ for a constant alphabet size. Kärkkäinen *et al.* [39] also proposed a suffix array that works in linear time: its time bound is independent from the alphabet size. The algorithm searches for the substring just like we search a word in the dictionary. The searching time is proportional to the length of text and the length of the pattern string. It is $O(m+\log n)$ time. This search time is longer than the suffix tree's search time $O(m+k)$. A detailed explanation of the searching substring and the construction of suffix array is in next subsections.

2.3.1 Query Time with the Suffix Array

First, searching pattern strings from the text will be shown instead of explaining how to build the suffix array since this search method is used by both Manber's and Kärkkäinen.

Let $T = a_1a_2 \ldots\ a_n$ be a text of length n. The suffix array stores the index of each suffix from the whole text $a_1a_2 \ldots\ a_n$ to a single character a_n lexicographically. All of the suffixes are written, using *pos(i)* , $1 \leq i \leq n$, that represents a suffix starting from position i and ending at position n. Also used is "=" and "<" to denote the lexicographic order. For example, $pos(i) = a_i \ldots\ a_{n-1}$ < $pos(j)$ = a_j . . . a_{n-1} representing the suffix a_i . . . a_{n-1} appears previous to the suffix a_j . . . a_{n-1} in lexicographic order. This means that the $pos(i)$ has a lower number than $pos(j)$ lexicographically. And "=" denotes that both of the substrings are the same; "=" is used only when searching a pattern string in the suffix array (each suffix is unique even though the text is a^n; each suffix length is different). Sort these *pos(1), pos(2), . . . , pos(n-1)* by lexicographic order to provide $O(\log n)$ time in searching for a substring. The search for pattern strings uses the binary search over the sorted suffix array. Each time the search range can be reduced by one half of the previous range. Starting at the middle of the suffix array the pattern string is matched with the suffix of the middle of the suffix array. If they match, then stop and return the *pos(i)*. Otherwise, compare *pos(i)* and the pattern-string. If the pattern-string < *pos(i)*, then the upper half of the suffix array is considered. Otherwise, the candidate suffixes are in the lower half of the suffix array.

Figure 2.7 shows the unsorted and the sorted suffix array for the text "*aabcabcaac*". Figure 2.7 (b) shows the suffix array sorted lexicographically. A search of the pattern-string can be performed via a binary search within the sorted suffix array. Figure 2.7 (b) also shows the process of searching the pattern-string for "*caa*. Numbered arrows show the order of the process. The numbered arrow 1 is the start index and compares *pos(9)* = "*ac*"

Suffix	Position
aabcabcaac	1
abcabcaac	2
bcabcaac	3
cabcaac	4
abcaac	5
bcaac	6
caac	7
aac	8
ac	9
c	10

(a)

SA	Suffix	Position	
SA(0)	aabcabcaac	1	
SA(1)	aac	8	
SA(2)	abcaac	5	
SA(3)	abcabcaac	2	
SA(4)	ac	9	← 1
SA(5)	bcaac	6	
SA(6)	bcabcaac	3	
SA(7)	c	10	← 2
SA(8)	caac	7	← 3
SA(9)	cabcaac	4	

(b)

Figure 2.7: The suffix array: (a) shows the unsorted suffix array. (b) shows the sorted suffix array and the numbered arrows show the order of the process to search for *"caa"*

and "*caa*". This results in *pos(9)* < *"caa*". The next occurrence range is below *pos(9)* in the sorted suffix array. Recursively, it is started at the middle of the next occurrence range. This recursive process will be stopped when the pattern-string matches the substring of the text or the occurrence range is 0. As seen in Figure 2.7, the number of comparisons is reduced by half with each iteration. At most *m* character comparisons are needed to match the pattern-string and *pos(i)* per each iteration. Thus, the running time is $O(m\log n)$.

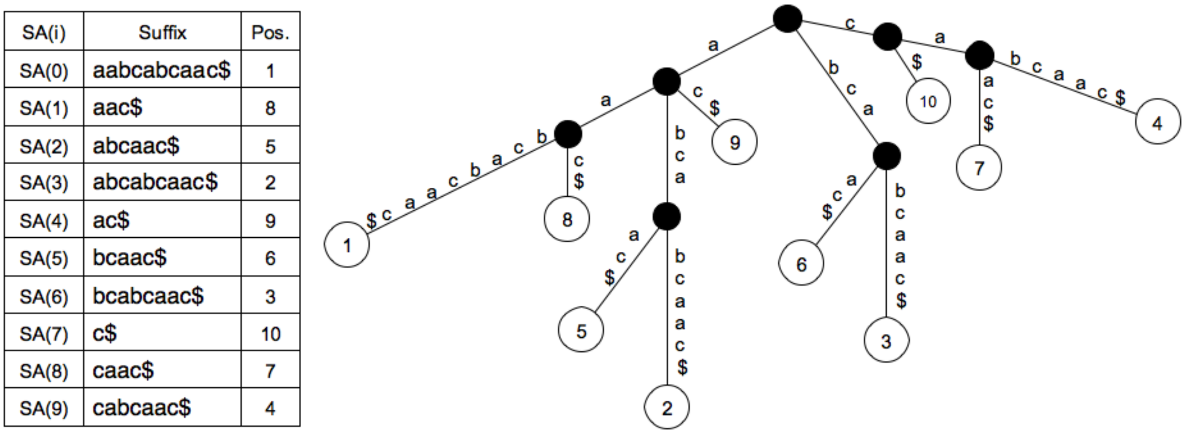
However, such running times can be reduced to $O(m+\log n)$. To search a pattern-string efficiently in a suffix array, LCP (longest common prefix) information is used to reduce the search time to $O(m+\log n)$. To reduce the search time bound to $O(m+\log n)$, there must be an update to $u = lcp(P,U)$ and $l = lcp(P,L)$ where P is a pattern string, U is the upper bound of the range and L is the lower bound of the range. The function $lcp(a,b)$ also returns the length of the LCP of the two strings a and b. In each iteration two cases are evaluated; first we evaluate which one is greater than, equal to, or less than the difference between u and l and $lcp(l \text{ or } u, M)$ where M is the middle point of the range. From this, it can be decided whether the pattern-string is in the upper half or lower half. When u and p or $lcp(l$ or $u,M)$ are equal, compare M to the pattern-string at $lcp+1$. If more than a single character in an iteration is compared, a maximum of u and/or l may grow by 1 for each comparison. Thus, the search time can be reduced by $O(m+\log n)$.

2.3.2 Constructing the Suffix Array

The suffix array stores pointers for sorted positions of suffixes in a given text, as seen in Figure 2.7 (b). The real suffix array has SA(i) and the position column (pointer to a position of text) instead of each suffix of the text since $O(n^2)$ (the length of text is n) spaces are needed to store all of suffixes of a text. Also, this causes the construction time to increase.

Before talking about the construction of a suffix array, similarities between the suffix tree and the suffix array must be examined. This also provides evidence to build the suffix array in $O(n)$. For the most part, the suffix tree and the suffix array contain the same information in lexicographic order. A suffix tree to a suffix array can be induced. The lexicographic order of the suffix array and the suffix tree are as shown in Figure 2.8. When traversing leaves of the suffix tree by in-order traversal, the same lexicographic order as the suffix array is obtained. The suffix tree can be converted in $O(n)$ time with traversing a suffix tree with in-order traversal in $O(n)$.

The algorithm of Manber *et al.* [43] uses the radix sort. The naive way to build the suffix array needs buckets as large as the alphabet size (each character is assigned to a corresponding bucket). Each suffix is classified by the first character of each suffix in its corresponding bucket. Each iteration in this algorithm increases the prefix of each suffix by 2^i where i is the number of the iteration (ex, $2^0, 2^1 \ldots$), that is, a bucket's label size is 2^i for the radix sort. Thus, the iteration is done in $O(\log n)$ iterations since the length of the



SA(i)	Suffix	Pos.
SA(0)	aabcabcaac$	1
SA(1)	aac$	8
SA(2)	abcaac$	5
SA(3)	abcabcaac$	2
SA(4)	ac$	9
SA(5)	bcaac$	6
SA(6)	bcabcaac$	3
SA(7)	c$	10
SA(8)	caac$	7
SA(9)	cabcaac$	4

Figure 2.8: The suffix tree and the suffix array

longest suffix is n. There are n number of suffixes in a text size n. Thus, sorting the suffixes needs $O(n\log n)$ time since the radix sort takes $O(n)$ for a constant alphabet size.

Kärkkäinen *et al.* proposed a truly linear time algorithm for the suffix array that does not depend on the alphabet size. This algorithm uses 2/3-recursion referred to as DC3 (difference cover module 3). This drives the recurrence $T(n) = T(2n/3) + O(n)$ where $T(2n/3)$ stand for the recursion and $O(n)$ stand for merging and the radix sort. Thus, the time bound is $O(n)$ to construct the suffix array. In the paragraph below, the 4 steps to construct the suffix array are illustrated: construct a sample, sort sample suffixes, sort non-sample suffixes and merge.

To execute the first (construct a sample) step we divide the text in 3 parts by using module 3 such that $B_k = \{i = \text{position} \mid i \bmod 3 = k\}$. There are three sets B_0, B_1 and B_2. Among these three sets we concatenate B_1 and B_2 ($\Longrightarrow B_1 \oplus B_2$). This set is called Sc that consists of sample positions like $\{1, 4, 7, \ldots, n \bmod 3 = 1, 2, 5, 8, \ldots, n \bmod 3 = 2\ \}$. Constructed in the second (sort sample suffix) step is the set R of strings with Sc such as $R = \{\ [a_1a_2a_3], [a_4a_5a_6], \ldots, [a_{n-2}a_{n-1}a_n], [a_2a_3a_4], [a_5a_6a_7], \ldots, [a_{n-1}a_na_{n+1}]\}$ where

a_i is a character at position i. Then the radix sort is used that allows $O(n)$ time to sort R. After executing the radix sort, the ranks are assigned to each triple by lexicographic order (if the same triple exists, then the same rank is assigned). If all ranks are different, the order of character is done for this step. Otherwise, the first and second step is recursively performed with *R'* which is it's rank. After the second step, the base case is reached or all ranks are different. Then the third (sort non-sample suffixes) step is performed. In this step B_1 is used that has already been sorted in the previous steps. B_0 stores {3, 6, ..., n mode 3 = 0}. Let $j_a, j_b \in B_0$. Then compare j_a and j_b with rank (j_a + 1) and rank (j_b + 1). For convenience, S_a is defined that is pair (j_a, rank (j_a + 1)). Now S_a and S_b can be easily sorted with radix sort since rank(j_i + 1) is unique even though j_a and j_b are of the same character. When the above steps are finished, two parts, R= $B_1 \oplus B_2$ and B_0 must be merged to obtain completed the suffix array. The fourth (merge) step is same as merging part of the merge sort. The merge operation takes $O(n)$ to arrange two sorted parts R and B_0. Figure 2.9 shows an example of construction of the suffix array in linear time.

2.4 The Directed Acyclic Word Graph

The suffix tree is the DFA (deterministic finite automaton) that represents all possible suffixes of a text and DAWG (directed acyclic word graph) [44] also is the DFA that minimizes the suffix tree. The DAWG provides the same time bound as the suffix array for creating and searching. We create the DAWG runs in $O(n^2)$ because $O(n^2)$ possible substrings in a text size n must be represented in the DAWG. (This is the same as the number of possible ways to choose two, the beginning and ending index, plus each single character from n, $\begin{pmatrix} n \\ 2 \end{pmatrix} + n = \frac{n(n-1)}{2} + n = \frac{n(n+1)}{2}$) . Because the DAWG has $O(n)$ edges and nodes, it performs in linear time.

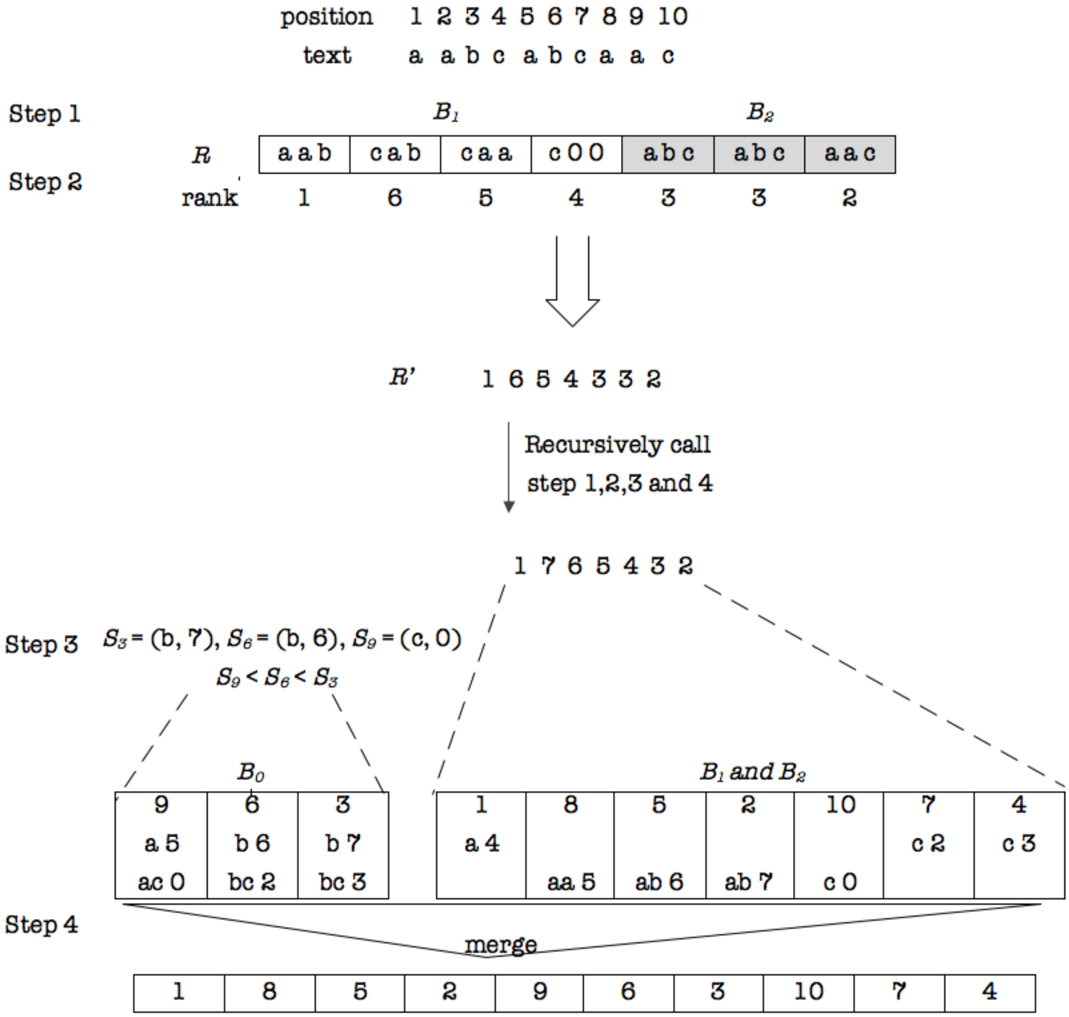


Figure 2.9: Constructing suffix array in linear time

First illustrated are the properties of the DAWG. As mentioned above, the number of nodes and edges of the DAWG of the text size n is $O(n)$. To understand $O(n)$ nodes and edges in the DAWG, all possible substrings are shown that can be set into *equivalence classes*. This means that two or more strings are in the same set if they have the same set of ending positions. For example,

1	2	3	4	5	6	7	8	9	10	11	12
a	a	b	c	a	a	b	c	a	a	b	c

when looking for the end positions of *"aabc", "abc"* and *"bc",*

End-Positions (aabc) = {4, 8, 12}

End-Positions (abc) = {4, 8, 12}

End-Positions (bc) = {4, 8, 12}

The three strings fall into the same set and these strings have an equivalence relation (reflexive; End-Positions (abc) = End-Positions (abc), symmetric; End-Positions (abc) = End-Positions (bc) $\Longleftrightarrow$ End-Positions (bc) = End-Positions (abc), and transitive; End-Positions (aabc) = End-Positions (abc) and End-Positions (abc) = End-Positions (bc) then End-Positions (aabc) = End-Positions (bc)). These strings are *right-equivalence class* by definition {*y*|*End-Position (y) = End-Position (x)*}. Also, note that the number of elements in a set (cardinality) is the number of occurrences. The substrings: "aabc", "abc" and "bc" are occur three times in the text. Thus, each substring in a text ends in one and only one right-equivalence class. This right-equivalence class is a node in the DAWG.

The size of the DAWG has $O(n)$ nodes and edges and this can be created in $O(n)$ time. At most $2n$ right-equivalence classes (nodes) are in the DAWG and the maximum number of edges is $3n-4$. First, it is proven that the DAWG has at most $2n-1$ nodes. One property of the right-equivalence class is the transitive reduction in the Hash diagram. Therefore, sets of ending position of two right-equivalence classes relation is one of these; $X \subseteq Y$, $Y \subseteq X$ or $X \cap Y = \emptyset$. This also shows that the cardinality of a set containing any i is unique. Now, the tree rooted with a degenerate class is created. Nodes are sorted containing $\{i\}$ for any i by cardinality and each sorted node is attached to the lowest cardinality set. The leaves are now at most n leaves and each internal node has at least 2 children. So, there are at most $2n-1$ nodes. To show that $3n-3$ edges are in the DAWG, it is assumed that the DAWG has $2n-2$ nodes plus one sink node and makes a directed spanning tree rooted at the degenerate class. This spanning tree has $2n-2$ edges. Think about every edge that is in the DAWG and not in the spanning tree. Theses edges are denoted by e. These edges connect two paths, one path from the root to e and another path from e to the sink (note that we must select an e closer to the sink). This connected path yield a suffix of a text. The number of edges e is at most the number of suffixes in a text. There are $n-1$ e's (when we make a spanning tree, one of the suffixes is in the tree). Thus, the total number of edges is $3n-3$.

The easy way to create the DAWG is from the degenerate node $\{1, 2, \ldots, n\}$ then partition the members of this node into the same character and attach this set to the root recursively running this process on every node. The running time for this process is $O(n^2)$ since there are $O(n^2)$ substrings. Figure 2.10 illustrates the naive method for creating the DAWG.

There is no advantage in using the quadratic time algorithms for string search. The DAWG can be incrementally built to reduce the time bound. Similar to the algorithms discussed above, the DAWG also uses a shortcut to finish one phase in the constant time.

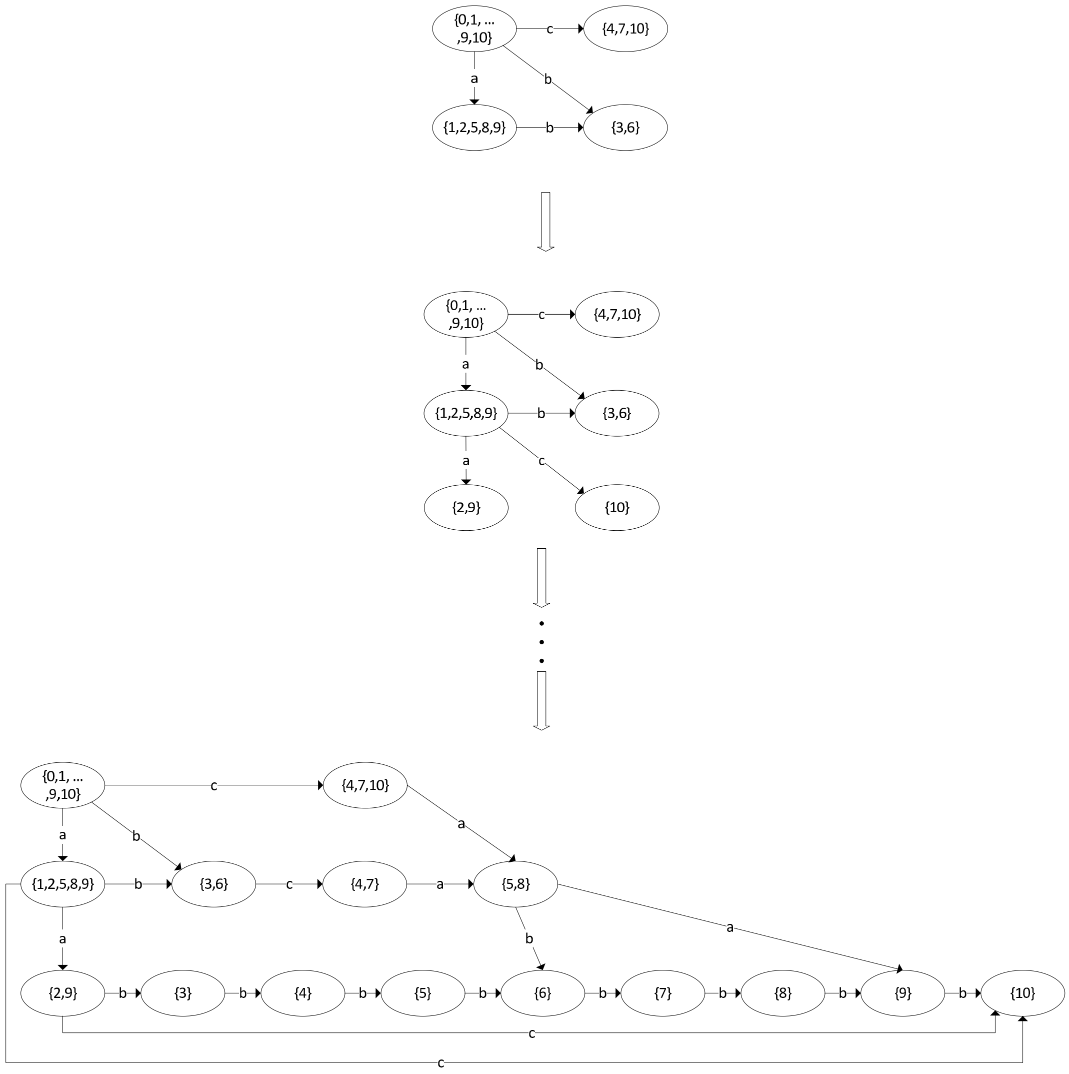


Figure 2.10: The naive method to create the DAWG

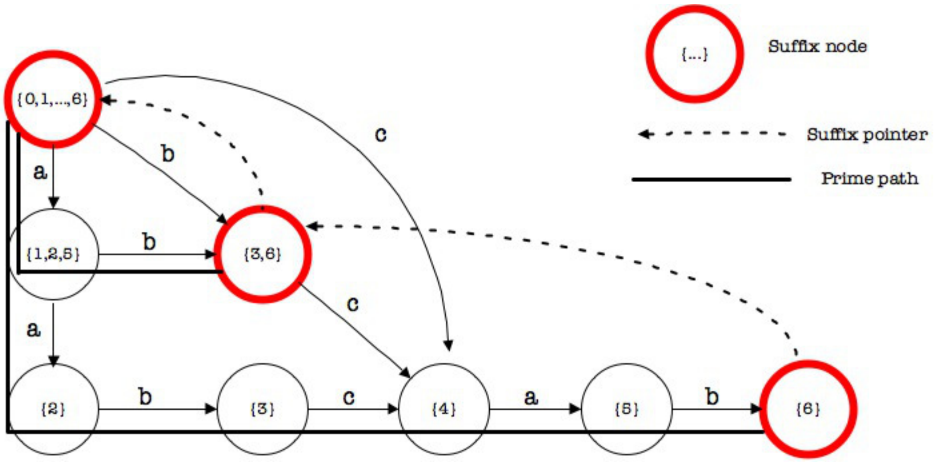


Figure 2.11: Suffix pointers for the DAWG

This shortcut is called the *suffix point*. The suffix point connects right-equivalence classes that were created in a previous phase. These right-equivalence class are called the *suffix node*s. The longest path from the root (the degenerate class) to a suffix node is the *primal path*. Figure 2.11 shows an example of the suffix node, the suffix pointer and the primal path.

Each phase starts from the DAWG of t_i to obtain the DAWG of t_{i+1}. A new sink node must be created and this new sink has an incoming edge form the previous sink node with label t_{i+1}. In Figure 2.11, the node 6 is a newly created sink node with edge *'b'* and node 5 is the previous sink node. Then we follow the suffix pointer to call other suffix nodes. If this suffix node does not have an edge with label t_{i+1}, an edge between this suffix node and the sink node is created with the label t_{i+1}. Otherwise, it must be checked whether a node reached from the suffix node with edge label t_{i+1} must be split. If this node is not split, stop and then add the next character. Let node Z be the neighbor of this suffix node on t_{i+1} and compare the length of primal path to the suffix node + 1 and the length of primal path to the node Z. If the primal path to the node Z is longer, the node Z must be split. Otherwise, just

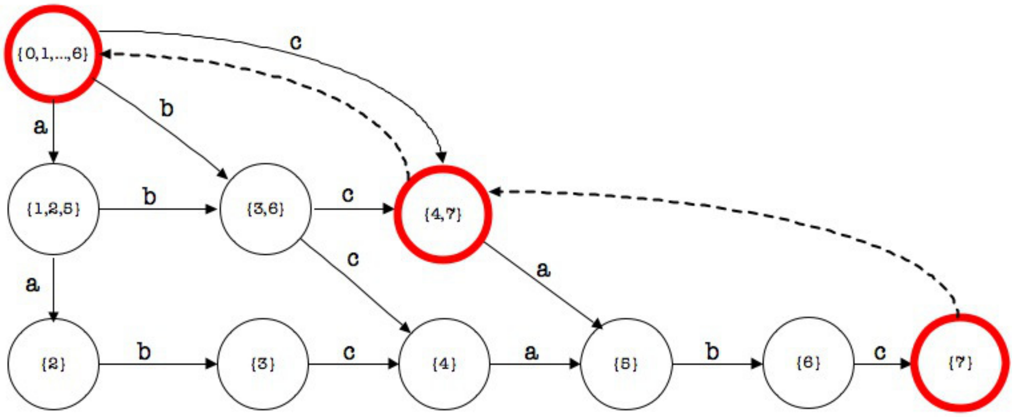


Figure 2.12: Extending the DAWG from *"aabcab"* to *"aabcabc"*

add the position t_{i+1} to the node Z. When splitting the node Z, the new node Y is created, which inherits the outgoing edge and members from Z (we also include the position of t_{i+1}) and installed incoming edge from remained suffix nodes with label t_{i+1}. This step is repeated until the last suffix node is reached (the degenerate node). Figures 2.11 and 2.12 shows examples of incremental processing from DAWG of *"aabcab"* to the DAWG of *"aabcabc*". The suffix node {3, 6} already has the edge labeled *'c'* and the length of the primal path to the node {4} is longer. This node is split and incoming edges are installed from remaining suffix nodes {0, 1, 2 . . ., 6} and {3, 6}.

Installing a new node and edge and splitting can be done in constant time. Each phase creates at most two nodes and each newly created node has one suffix pointer. In some phases, many suffix nodes are traversed, but a much smaller number of suffix nodes are traversed in the next phase. Thus the DAWG is created in linear time.

2.4.1 The compact DAWG

Now consider a more memory efficient DAWG called the compact DAWG. Redundant information (substrings) in a text requires more spaces. For example, if a text consists of a^n, there are $n+1$ right-equivalence classes and n edges in the DAWG. The single character “a” falls into the set $\{1, 2, \ldots$, n$\}$ and “aa” falls into the set $\{2, 3, \ldots, n\} \ldots$, a^n falls into the set $\{n\}$. So, $n+1$ nodes and n edges are needed to represent a^n in the DAWG (this is a linked list and the initial node is the root). In another extreme condition, when the length of a text is same as the alphabet size, $n+1$ nodes and $2n-1$ edges are also created. If an additional $O(n)$ time is spent, a compact DAWG can be obtained that occupies much less space than the DAWG.

To create the compact DAWG, first the non-primal node is defined that has exactly one outgoing edge in the DAWG. Only two nodes can be represented and one edge for the text a^n since all nodes are non-primal nodes except the root and sink. In a second extreme case, two nodes and n edges can be created. As in Figure 2.13, the compact DAWG consists only of primal nodes except the sink node. Each edge is represented by the length of the substring and the smallest member represents each node. Consider the text $a^n b$, this case forces us to create the same number of nodes and edges as the DAWG since every node has two outgoing edges. The DAWG in this case cannot be compressed. Converting the DAWG to the compact DAWG is performed in depth-first traversals. Each class can be labeled with a single position from its set of ending positions. When retreating from a node, the primal node and the distance are calculated. Finally, The DAWG edges are removed. Also, Crochemore *et al.* [35, 36, 13]suggest the compact DAWG is directly constructed from a text. Their approach modified the linear time DAWG algorithm.

Finally, the search pattern string runs in $O(p+k)$ time. This is the same as finding a

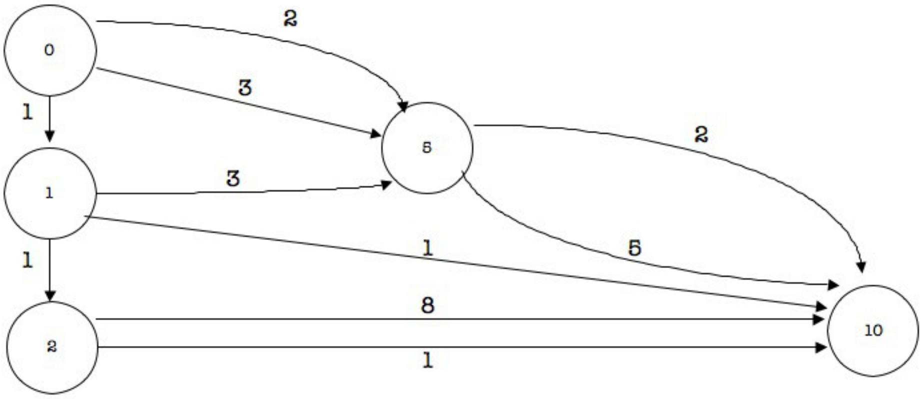


Figure 2.13: The compact DAWG

pattern string in the suffix tree. However, a search of a pattern string in the compact DAWG is different than in the DAWG. Looking at an example with the string *"abc*", starting from the root, node 1 is selected in which the number of elements can be calculated. First, there are five paths to the sink meaning that node 1 has five elements. Second, the edge label for each path is summed up and subtracted from the sink position to obtain positions (1 = 10-9, 2 = 10-8, 5 = 10-5, 8 = 10-2, 9 = 10-1). In node 1, it is possible to transition to node 5 with the character *"b*". On this node, positions {5, 8} can be calculated as above. However, one character is exceeded since the edge consists of three characters, therefore, one is subtract from {5, 8}. The resulting position is then {4, 8}.

Chapter 3

The Position Heap

The **sequence hash trees** of Coffman and Eve [11] and the data compression algorithm of **Lempel-Ziv** [41] provide the basic idea for the position heap. The sequence hash tree cannot be used directly for string searching problems. Ehrenfeucht and McConnell [44]. tailored the algorithm of Lempel-Ziv to string searching and proposed the position heap. First, we explain the sequence hash tree before demonstrating the position heap.

3.1 Sequence Hash Trees

A data structure of Coffman and Eve [11], called a **sequence hash tree**, was designed for the problem of implementing hash tables (dictionaries) whose keys are strings. It consists of a trie for indexing into the table. The structure of the tree depends on the order in which the strings are inserted. We describe a minor variant that is easier to adapt to the substring matching problem, below.

Let $\mathcal{S} = (S_1, S_2, \ldots S_n)$ be a given ordering of the strings. Without loss of generality for our purposes, we may assume that no string is $\mathcal{S}$ is a prefix of any other. The trie that they construct is defined by induction, as follows. If $i = 1$, the trie H_1 is just a root node with a pointer to S_1. If $i > 1$, then H_i is obtained from H_{i-1} by finding the shortest prefix Xb of

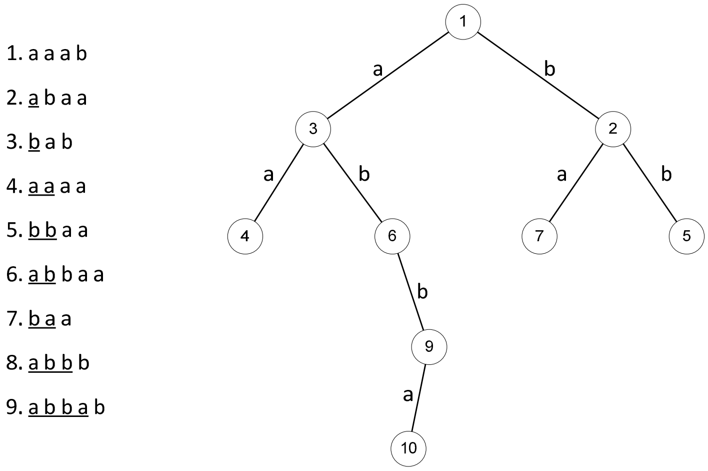


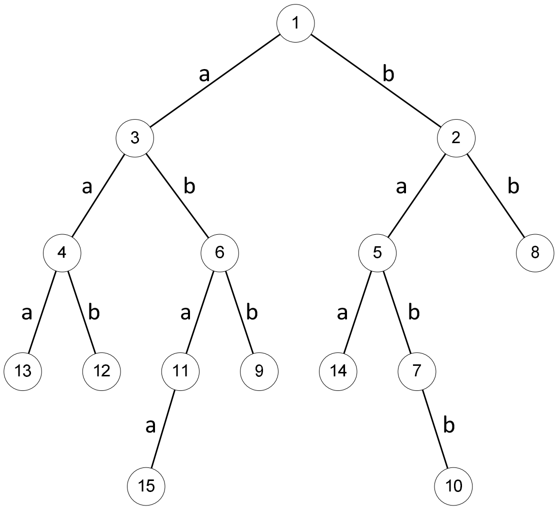
Figure 3.1: The sequence hash tree of a sequence of strings. We refer to each node by the string of letter labels on the path from the root to the node. For example, the node labeled 6 can be thought of as synonymous with the string *ab*. Each string in the sequence is installed at a new node that is the shortest prefix of the string that isn't already a node of the sequence hash tree. These prefixes are underlined. For example, when string 9 is inserted, its prefix *abb* is already a node of the tree, but its prefix *abba* is not, so a pointer to string 9 is inserted at a new node, *abba*.

S_i that is not already a node of the trie. A new node Xb is added as the child of node X on edge labeled b, and a pointer is installed from it to S_i.

Figure 3.1 gives an example. Coffman and Eve's paper has received little attention since it was published in 1970, due, in no doubt, to the existence of superior ways of implementing a hash table. In the present paper, we show that this data structure is much richer when considered in the context of the new problem. The structure of the set of suffixes of a text T allows us to derive interesting and algorithmically useful properties that do not apply in the general case addressed by Coffman and Eve.

3.2 The Position Heap

The position heap can be best defined through a simple constructive algorithm.



15	14	13	12	11	10	9	8	7	6	5	4	3	2	1
a	b	a	a	a	b	a	b	b	a	b	a	a	b	a

Figure 3.2: Incremental construction of the position heap. Suffixes $t_1, t_2, \ldots, t_n$ are inserted in ascending order of length. The figure depicts the insertion of t_i when $i = 15$. Indexing into the heap on t_i identifies the longest prefix (aba) of t_i that is already a node Y of the heap. The shortest prefix of t_i that is not already a node of the heap $(abaa)$ is inserted as a child of Y and labeled with position i.

To construct the position heap, we processes a text from right to left. We therefore adopt the convention of indexing the positions of T in descending order, that is, $T = t_n t_{n-1} ... t_1$ (See Figure 3.2). To build the position heap, starting from the right most character t_1, position 1 is the root of the position heap.

Following the example of Figure 3.2, after the root is situated, the next position, position 2, is processed. Since the root is occupied, we look at the character, b, that occurs at position 2. We make a node labeled 2 as a child of the root on edge labeled b.

We then process position 3. Since the root is occupied, we look at the character, a, that occurs at position 3, and make a new node labeled 3 as a child of the root on edge labeled a.

A key issue first arises in processing the next position, position 4. The root is occupied by 1. We look at the character, a, at position 4, but we cannot place the 4 in a child of the root labeled a, since that node is already occupied by position 3. Therefore, we must look at the next longer prefix, aa, of the suffix that begins at position 4. We place a new node labeled 4 at the end of a path whose edge labels are aa.

Similarly, ba is the shortest prefix of the suffix beginning at position 5 that does not already occur as the sequence of edge labels on a path from the root, so we place a new node labeled 5 at the end of a path whose edge labels are ba.

Definition 3.2.1 *The* **position heap** $H(T)$ *of a text* T *is obtained by iteratively inserting the suffixes* $(t_1, t_2, \ldots, t_n)$ *of* T*, in ascending order of length, into Coffman and Eve's data structure using their insertion operation. That is,* t_i *is inserted by creating a new node that is the shortest prefix of* t_i *that is not already a node of the tree, and labeling it with position* i*.*

Let us call the algorithm implied by this constructive definition the **naive construction**

algorithm. Figure 3.2 gives an illustration. Coffman and Eve assume that each inserted string ends with a special character \$ to ensure that if all inserted strings are distinct, no inserted string is already a node of the tree when it is inserted. This is unnecessary here, since each string t_i is longer than any string inserted before it, and each node previously inserted is a prefix of some t_j for $j < i$.

The construction can be executed for any text T, and, since it is deterministic, the position heap $H(T)$ for a text is unique.

3.2.1 A Time Bound for Constructing the Position Heap

We can obtain a time bound for constructing the position heap by analyzing the running time of the naive algorithm.

Let $h(T)$ denote the length $||X||$ of the longest substring X of T that occurs at least $||X||$ times in T.

Lemma 3.2.2 *The height of the position heap of a text T is at most* $2h(T)$*.*

Proof: Let $X = x_k x_{k-1} \ldots x_1$ be a deepest leaf of the tree. Let X_i denote the prefix $x_k x_{i-1} \ldots x_i$ of X. For each i from 1 through k, X_i occurs at least i times in T because it has at least i descendants, $\{X_i, X_{i-1}, \ldots X_1\}$, and each of these contains an occurrence of a substring of which X_i is a prefix. Therefore, $X_{\lceil k/2 \rceil}$ has length $\lfloor k/2 \rfloor$ and occurs at least $\lceil k/2 \rceil$ times in T. It must be that $\lfloor k/2 \rfloor$ is a lower bound on $h(T)$, so the height k is bounded by $2h(T)$. Figure 3.3 depicts that the height of the position heap is $O(h(T))$.

Lemma 3.2.3 *The naive algorithm takes* $O(nh(T))$ *time to build a position heap for a text* T*.*

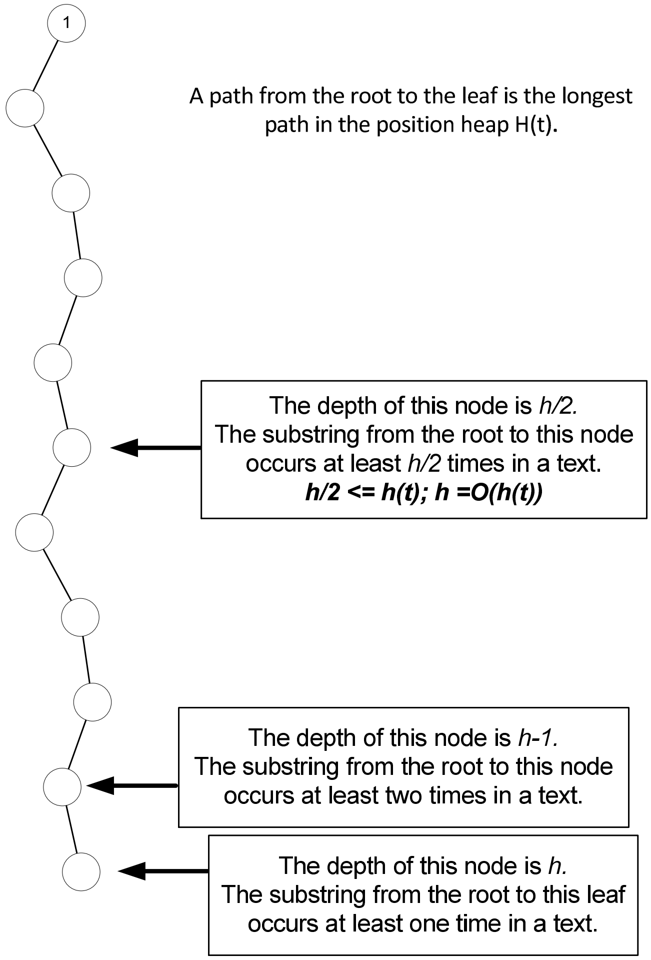

1
A path from the root to the leaf is the longest path in the position heap H(t).
The depth of this node is h/2.
The substring from the root to this node occurs at least h/2 times in a text.
h/2 <= h(t); h =O(h(t))
The depth of this node is h-1.
The substring from the root to this node occurs at least two times in a text.
The depth of this node is h.
The substring from the root to this leaf occurs at least one time in a text.

Figure 3.3: The longest path in the position heap is selected. A path from a root to a leaf is a substring in a text and this path length is h. This substring occurs at least one time in the text. The depth of the parent node of this leaf node is $h-1$. This substring with the length $h-1$ appears at least two time in the text. When choosing a node that is the middle node of this path, this middle node's depth is $h/2$ and the substring with length $h/2$ appears at least $h/2$ times in the text.

Proof: Indexing into the heap to find the parent of the new node to be inserted for position i takes time that is bounded by the height of the heap, hence $O(h(T))$ time. Adding the new child takes $O(1)$ time. Summing this over all positions i gives an $O(nh(T))$ bound.

3.3 The Query Algorithm with the Position Heap

We now give a time bound for querying the position heap. We improve the time bound in the next chapter, at the expense of adding elements to the data structure.

Definition 3.3.1 *The* **naive query algorithm** *for finding all occurrences of a pattern string P in T consists of the following steps.*

- *Index into the position heap to find the longest prefix X of P that is a node of $H(T)$. For each ancestor X' of X (including X), look up the position i stored in X'. Position i is an occurrence of X'. Determine whether this occurrence is followed by $P - X'$. If it is, report i as an occurrence of P.*

- *If $X = P$, also report all positions stored at descendants of X.*

For example, when looking for the pattern string *"aba"* in Figure 3.4, start from the root and search the edge which is labeled *'a'* and then transition to the corresponding node 3. Repeat searching for the edges that correspond to characters of the pattern string until there are no more transitions, . After reaching the end of the pattern string that is node 11, simply return the position numbers of the subtree rooted at the last node 11 that is {11, 15}, because the descendant nodes under the subtree have a prefix that is the same as the pattern string. Then check all nodes on the path. After finding the path for the pattern string in the position heap, verify all nodes on the path that are {1, 3, 6}, since these nodes' path label is

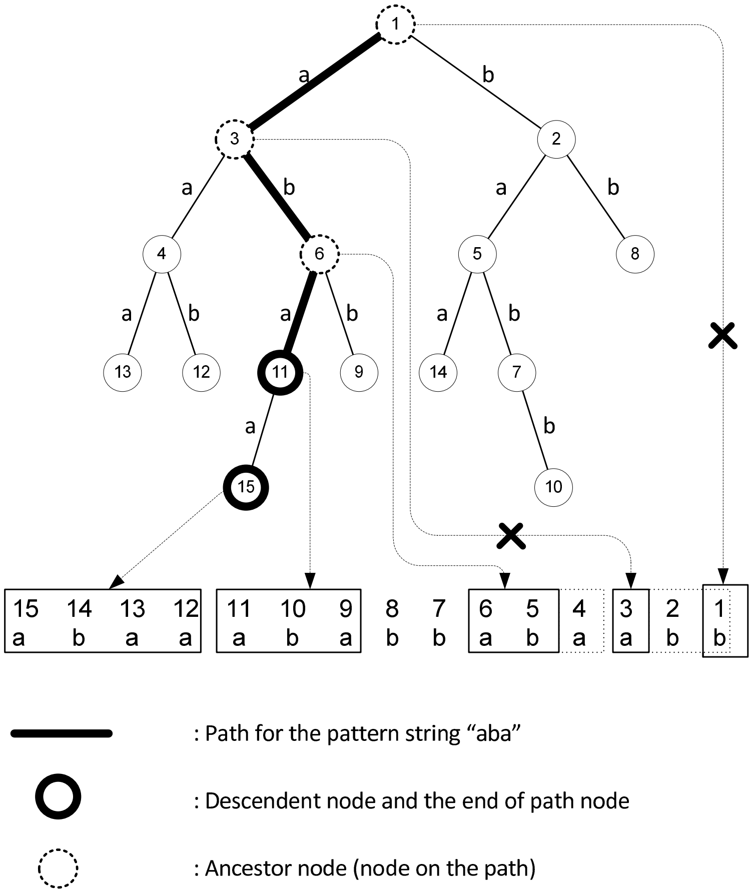

1
a
b
3
2
a
b
a
b
4
6
5
8
a
b
a
b
a
b
13
12
11
9
14
7
a
b
15
10
15 14 13 12
a b a a
11 10 9
a b a
8 7
b b
6 5
a b
4
a
3
a
2
b
1
b
: Path for the pattern string "aba"
: Descendent node and the end of path node
: Ancestor node (node on the path)

Figure 3.4: The example of the naive query algorithm

a prefix of the pattern string. The verification of these nodes on the path delays the position heap's search time. With this example, position 6 has the same prefix as the pattern string.

Lemma 3.3.2 *The naive query algorithm is correct.*

Proof: A node X' contains a position i where X' occurs in T.

If X' is a prefix of P, then it is an ancestor of X, and i may or may not be an occurrence of P in T, depending on whether the occurrence of X' at i is followed by $P - X'$. The test for this condition returns i if and only if it is an occurrence of P.

If P is a prefix of X', $X = P$, and since all prefixes of X' occur at position i, so does P. This is reported during the traversal of the subtree rooted at P.

If the longest common prefix Y of P and X' is neither P nor X', then the occurrence of Y at i is followed by the first letter of $X' - Y$, which is not the first letter of $P - Y$. Therefore, i is not an occurrence of P. The query does not report i in this case.

Lemma 3.3.3 *The naive query algorithm runs in $O(min(m^2, mh(T)) + k)$ time, where m is the length of the query string, and k is the number of occurrences of it in T.*

Proof: If X is the longest prefix of P that is a node of the heap, it takes $O(||X||)$ time to find X by indexing into the heap on P. For each of the $||X|| + 1$ ancestors of X, we must look up the position i stored in the ancestor, and determine, in $O(m)$ time whether P occurs at position i. Since $||X|| \leq m$, this gives an $O(m^2)$ bound for this step. Since $||X||$ is $O(h(T))$, this also gives an $O(mh(T))$ bound for this part.

If $X = P$, that is, if P is a node of the position heap, it also takes $O(1)$ time to return each of the positions in the subtree rooted in its subtree for a total of $O(m^2 + k)$ and $O(mh(T) + k)$.

Lemma 3.3.4 *If T is a randomly constructed string and the construction of P does not depend on T, or if P is a randomly constructed string and construction of T does not depend on P, then the naive query algorithm takes $O(m + k)$ expected time, where m and k are as in Lemma 3.3.3.*

Proof: The $mh(T)$ term comes from the fact that at each of $O(h(T))$ nodes X', we must check whether the occurrence of X' at the position i that it stores is followed by $P - X$. This requires checking whether $||P - X||$ letters of P match at $||P - X||$ positions of T. The check halts when a mismatch is detected. The probability of any of positions matching is $1/|\Sigma|$, so the expected number of checks before halting is $(\Sigma-1)/\Sigma\sum_{i=1}^{||P-X||} 1/|\Sigma|^i = O(1)$.

Chapter 4

The Augmented Position Heap

The position heap provides the better space requirement and the equal time bound for construction time with the suffix tree, suffix array and DAWG except searching time; we will show the linear time algorithm to construct the position heap and the augmented position heap in the next chapter. As shown in Table 4.1, the position heap's and suffix array's searching time is slower than other competitive data structure. This is a disadvantage in practical interest.

To overcome this obstacle, additional pointers are installed into the position heap. This heap is called the augmented position heap. This work is appeared in [17, 16]. The construction time bound is the same as the position heap. It can be built in $O(n)$ amortized time, and spend $O(m+k)$ time to find the k occurrences of the pattern string P. However, it need additional space than the position heap because of extra pointers.

	Preprocessing Time	Searching Time
Augmented Position Heap	$O(n)$	$O(m+k)$
Position Heap	$O(n)$	$O(m^2+k)$
DAWG	$O(n)$	$O(m+k)$
Suffix Array	$O(n)$	$O(m+\log n+k)$
Suffix Tree	$O(n)$	$O(m+k)$

Table 4.1: Construction and Searching time comparison

These extra pointers are defined as ***maximal reach pointers***. A maximal-reach pointer is the longest prefix of starting position at t_i that is a node in the position heap. A node has a maximal-reach pointer that points to a descendant or does not have a maximal-reach pointer because a longest existing prefix is itself in the position heap. In Figure 4.1, node 1, 2, 3, 4, 5 and 6 have a maximal-reach pointer. However, node 7, 8, 9, 10, 11, 12, 13, 14 and 15 do not have a maximal-reach pointer (a maximal reach pointer points itself). Also, dot lines can be compared under the text and bold lines above the text in Figure 4.1. Dot lines indicates paths for the maximal-reach pointers and bold lines represent paths from the root to each node. A length of dot and bold line is identical when a node does not have a maximal-reach pointer. If a node has a maximal-reach pointer, a dot line is longer than a bold line. Since the maximal-reach pointer must reach deeper than its path.

For example, with Figure 4.1, node 3 can reach node 11 with substring *"aba"*. Substring *"aba"*, $t[3:1]$ is the longest prefix that can be represented in the augmented position heap. So, the augmented position heap contains information that the longest prefix for starting at each position.

Definition 4.0.5 *Let i be the position stored at node X in $H(T)$, and let Y be the largest prefix of T_i that is a node of $H(T)$. The* **maximal reach pointer for** *X is a pointer from node X to node Y. The* **augmented position heap** *for T is obtained by labeling each node X of $H(T)$ with its maximal-reach pointer and X's preorder and postorder number in a traversal of $H(T)$. We also associate with the heap an array* `Nodes[]` *such that* `Nodes[i]` *contains a pointer to the node of the heap that contains position i. Let $H'(T)$ denote the augmented position heap.*

Create the position heap for T. The pointers can be installed in `Nodes[]` and the preorder and postorder numbers can be assigned to nodes of the heap during a depth-first

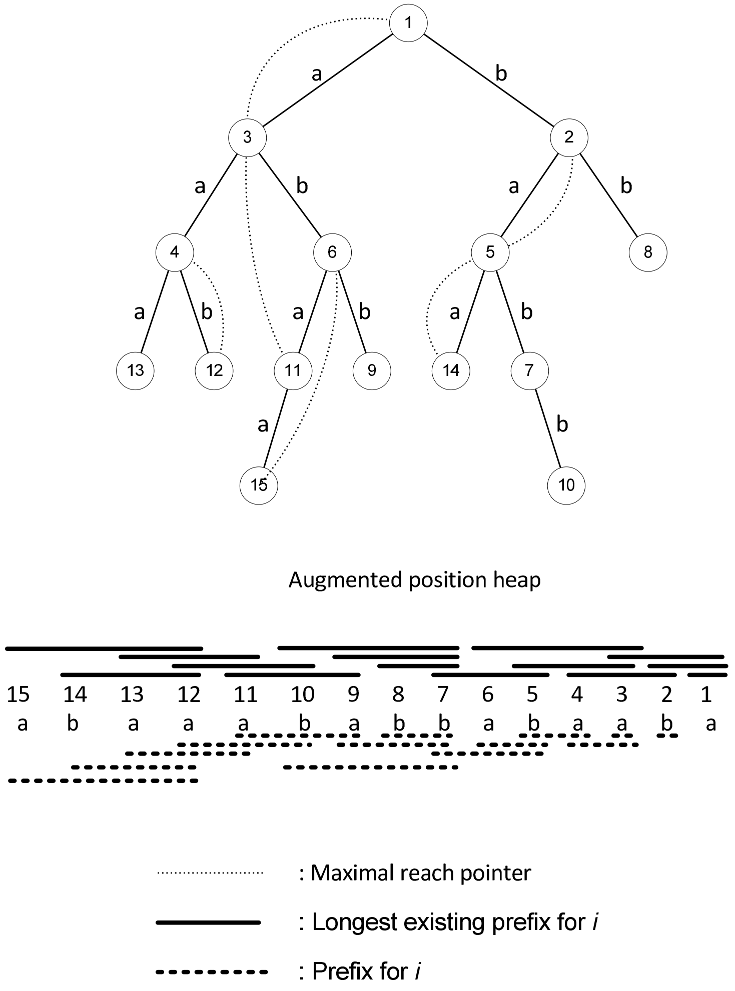


Figure 4.1: Augmented position heap

traversal of T. Then, the naive method to point the longest existing path for each each suffix T_i of T intuitively visits down though a path until a substring is matched with a path or the end of path. To point each node's existing longest path, start from each node and look for an edge's label that is a character of $node's\,poistion - node's\,depth$. These steps are processed until a leaf is reached or a edge does not exist. In the worst case, every node's existing longest path can reach leaves. The time bound relates with $O(h(T))$ of the position heap. Thus, the running time for installing maximal-reach pointers is $O(nh(T))$.

4.1 An $O(m+k)$ Time Bound for a Search with the Augmented Position Heap

A pattern string can be found in $O(m^2+k)$ within the traditional position heap. This is the critical disadvantage when comparing with other data structures for the string searching. The fastest algorithms for string searching are the suffix tree and DAWG. These data structures allow us to find a pattern string in $O(m+k)$. This is the best search time bound so far. Most of the query algorithms in this section are similar with the naive query algorithms. It recommends that the readers focus more on how to utilize the maximal-reach pointers to acquire the improved query time bound.

As shown in the previous chapter, what consumes time is the intermediate nodes that are path nodes between the root and the end of the pattern string since a pattern string must be compared with each substring that starts from the intermediate node's position. The worst case for this comparing process is $O(m^2)$. If we improve this comparing process from $O(m)$ to $O(1)$ for each intermediate node, this time bound for the search can be reduced to $O(m+k)$.

The linear query algorithm with the augmented position heap is similar with the naive query algorithm except when comparing a substring with a pattern string. Starting from the root and the first character p_1 of the pattern string, follow the path with the pattern string. After stoping at node v, which is the end of the path, report all nodes of the subtree rooted this node v. So far, this scheme is the same as the naive query algorithm. All nodes on the path must be checked with the naive query algorithm to obtain all occurrences, since there is limited information of the ancestors of node v within the position heap. To obtain complete information for each ancestors' prefix, $O(m^2)$ time must be spent. Instead of this slow process, nodes are reported that are linked with this subtree by a maximal-reach pointer. The maximal-reach pointer verifies that these nodes start with the same prefix as the pattern string. If node i's maximal-reach pointer points to a subtree that roots node v of the end of the path, position i begins with the prefix that is the same as the pattern string. It takes $O(1)$ whether it points to the proper subtree.

To verify the descents, perorder and postorder numbers are utilized. It is well-known that node x of a rooted tree is an ancestor of node y if and only if the preorder number of x is less than the preorder number of y and the postorder number of x is larger than the postorder number of y. This gives the following:

Lemma 4.1.1 *Given pointers to two nodes X and Y of $H'(T)$, it takes $O(1)$ time to determine whether X is an ancestor of Y.*

Lemma 4.1.2 *Given a pointer to a node X of $H'(T)$ and a position i, it takes $O(1)$ time to determine whether i is an occurrence of string X in T.*

Proof: It takes $O(1)$ time to find the node Y that contains i. By Lemma 4.1.1, it takes $O(1)$ time to determine whether Y is a descendant of X. If so, then since i is an occurrence of Y and X is a prefix of Y, i is an occurrence of X.

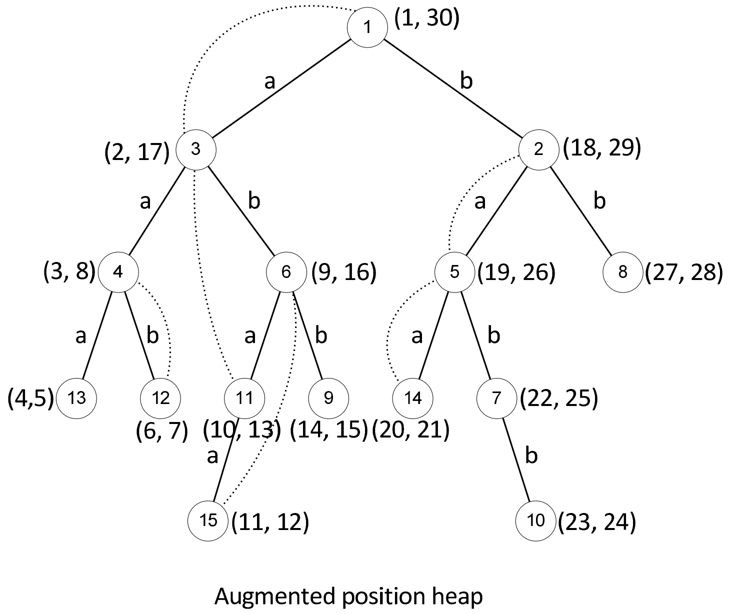


Figure 4.2: Depth-first discovery and finishing times with the augmented position heap. Any descendant's discovery and finishing time is bounded its ancestor's discovery and finishing time.

If not, it takes $O(1)$ time to determine whether Y is an ancestor of X, by Lemma 4.1.1. If it is, then let Z be the node pointed to by the maximal reach pointer of Y. Position i is an occurrence of X if and only if X is a prefix of T_i. Z is the maximal prefix of T_i that is a node of the heap. Therefore, X occurs at position i if and only if it is a (not-necessarily proper) prefix of Z, that is, if and only if Z is a descendant of X. This takes $O(1)$ time to determine, by Lemma 4.1.1.

For example with Figure 4.2, given a pointer to the node *ab* (the one labeled 6), we can tell that 11 is a descendant by looking in `Nodes[11]` to find a pointer to its node *aba*, and using the preorder and postorder numbers of *ab* and *aba* to determine that *aba* is a descendant. Therefore, it is an occurrence. We can tell that 3 is an occurrence by looking in `Nodes[3]` to find its node *a*, using the preorder/postorder numbers to find that it is an ancestor of node *ba*, using its maximal reach pointer to find the node *aba*, and using the preorder/postorder numbers of *ba* and *bab* to determine that *aba* is a descendant of *ab*. We can tell that 1 is not an occurrence, because its maximal-reach pointer doesn't point to a descendant of *ab*.

Corollary 4.1.3 *Let Xc be a string such that X is a node of the tree and Xc is not. Given a pointer to X and a position j, it takes $O(1)$ time to determine whether j is an occurrence of Xc.*

Proof: By Lemma 4.1.2, it takes $O(1)$ time to determine whether j is an occurrence of X. If it is, then it is an occurrence of Xc if c occurs at position $j - ||X||$, which takes $O(1)$ time to check when T is stored in an array.

Before giving pseudocode for the linear-time query algorithm, we illustrate the main ideas in Figure 4.3. There are two cases: Case 1, where the search string is a node of the position heap, and Case 2, where it is not.

Case 1 is illustrated by *ba*, which is the node labeled 6. By Lemma 4.1.2, we can now check in $O(1)$ time apiece which of the positions $\{1,3\}$ at proper ancestors are occurrences of *ba*. Only 3 is; its node is the only proper ancestor with a maximal-reach pointer into *ba*'s subtree. That is $O(m)$ time so far. In addition, we report the labels of descendants $\{6,12,9\}$ in $O(1)$ time apiece, as before, in $O(k)$ time, for a total of $O(m+k)$ time.

Case 2 is illustrated by *babbabbab*, which is not a node of the heap. Our strategy is to partition the string into segments *babb*, *abba*, and *b*, which can be handled efficiently by Corollary 4.1.3 and Lemma 4.1.2. We use the corollary to find the occurrences of *babb*, discard those that are not followed by *abba*. This gives the occurrences of *babbabba*. We then use the lemma to discard from these the occurrences those that are not followed by *b*.

To apply the corollary, we want all the segments except the last to be of the form Xc, where X is a node of the tree and Xc is not. The first such segment is *babb*. This is our *current substring*. As in the naive query algorithm, only ancestors of $X = bab$ can be positions of *babb*. These are labeled $\{1,3,6,9\}$. By Corollary 4.1.3, we can determine which are occurrences of the current substring *babb* in $O(1)$ time apiece, for a total of $O(||Xc||)$ time. This leaves positions $\{6,9\}$. The string *babb* becomes the *finished prefix*, its positions $\{6,9\}$ are known, and the rest of the query string, *abbab* is the *remaining suffix*.

We now look for the prefix of the remaining suffix *abbab* of the form Xc, where X is a node of the heap and Xc is not. This is *abba*. We want to find which occurrences of Xc follow occurrences of the finished prefix *babb*. To do this, we subtract the length of the finished prefix from each of the positions of the finished prefix and determine in $O(1)$ time whether it is an occurrence of Xc, by Corollary 4.1.3. In the example, subtracting $||babb|| = 4$ from 9 gives 5, and we determine that 5 is an occurrence of *abba*. Therefore, 9 is an occurrence of *babbabba*. Subtracting 4 from 6 gives 2, and we determine that 2 is not an occurrence of *abba*. The finished prefix is now *babbabba*, the positions where it occurs are known to be $\{9\}$, and the remaining suffix is *b*.

When the remaining suffix is short enough to be a node of the tree, let us denote it Y. (In the example, $Y = b$.) We subtract the length of the finished prefix from each of its occurrences ($\{9\}$ for this example), and check whether each of these positions ($\{1\}$ in this example) is an occurrence of Y, using Lemma 4.1.2.

Generalizing from these examples, we get the algorithm of Table 4.2.

Lemma 4.1.4 *The linear query algorithm is correct.*

Proof: For Case 1, the procedure is the same as the naive algorithm, except that at each

Table 4.2: The **linear query algorithm for use with the augmented position heap**

- **Case 1:** *P is a node of $H'(T)$.* This is detected by indexing into $H'(T)$ on P, and gives node P. For each proper ancestor X' of P, look up the position i stored at X', and determine whether it is an occurrence of P. In addition, report all positions recorded in the subtree rooted at P.

- **Case 2:** *P is not a node of $H'(T)$.*

```
// Find an initial set of candidate positions
Let CurrentSubstring be the shortest prefix of P that is not a node of H'(T)
Let I be the set of positions where CurrentSubstring occurs

// Invariants: FinishedPrefix + RemainingSuffix = P;
// I is the set of positions where FinishedPrefix occurs in T

FinishedPrefix = CurrentSubstring
RemainingSuffix = P - CurrentSubstring
while RemainingSuffix is not a node of H'(T)
    Let CurrentSubstring be the shortest prefix of RemainingSuffix
        that is not a node of H'(T)
    $I := \{j | j \in I$ and the occurrence of FinishedPrefix at $j$ in $T$
            is followed by an occurrence of CurrentSubstring$\}$
    RemainingSuffix = RemainingSuffix - CurrentSubstring
    FinishedPrefix = FinishedPrefix + CurrentSubstring
CurrentSubstring = RemainingSuffix
Let $I := \{j | j \in I$ and the occurrence of CurrentPrefix at $i$
    is followed by CurrentSubstring$\}$
```

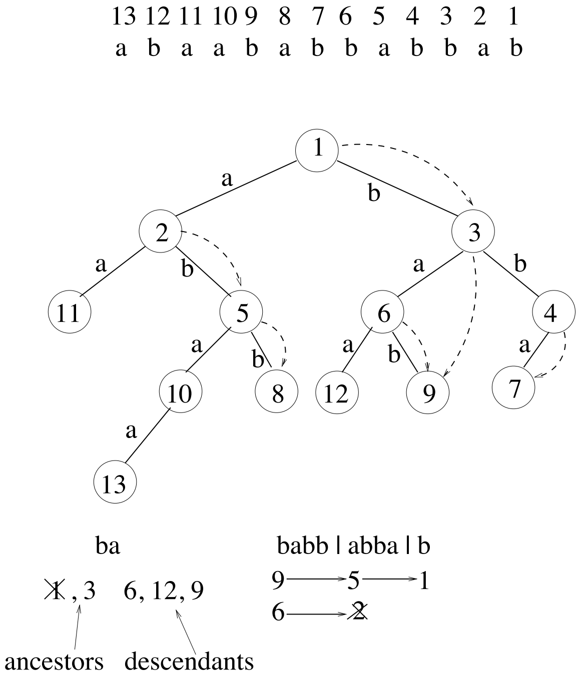


Figure 4.3: The linear query algorithm on strings *ba* and *babbabbab* on the augmented position heap. Maximal-reach pointers that are loops are omitted from the diagram.

ancestor X' of P, we determine whether i is an occurrence of P in $O(1)$ time, instead of $O(||P||)$ time, using Lemma 4.1.2.

For Case 2, by induction on the number of times `FinishedPrefix` is assigned, I is the set of positions where `FinishedPrefix` occurs in T. In the final line, $P = FinishedPrefix + RemainingSuffix$, and I is assigned to be those positions of $FinishedPrefix$. After the final step, $FinishedPrefix = P$, hence I is the set of positions in T where P occurs.

Lemma 4.1.5 *The linear query algorithm can be implemented in $O(m+k)$ time using the augmented position heap.*

Proof: Case 1 differs from the naive approach only in that it uses Lemma 4.1.2 to determine which ancestors of P contain the position of an occurrence of P, reducing each of these tests from $O(m)$ to $O(1)$. Since there are $m+1$ ancestors of P, this takes $O(m)$ time. As in the naive query algorithm, all other occurrences of P are found in $O(1)$ time apiece during a traversal of the subtree rooted at P, for a total of $O(m+k)$ time.

For Case 2, let $(P_1\ P_2, \ldots, P_l)$ be the values taken on by $CurrentSubstring$, and let $(I_1, I_2, \ldots, I_l)$ be the values taken on by I.

To find the i^{th} value P_i of $CurrentSubstring$, index as far as possible on $RemainingSuffix$ into $H'(T)$, yielding node X_i, and let b be the next character of $RemainingSuffix$ following prefix X. $P_i = X_i b$. Over all iterations, this takes time proportional to $\sum_{i=1}^{l} ||P_i|| = O(||P||)$. For $2 \leq i \leq l$, and each $j \in I_{i-1}$, it takes $O(1)$ time to determine whether the instance of $FinishedPrefix$ at position j is followed by X_i; this is determined by finding whether $j - ||FinishedPrefix||$ is an occurrence of X_i, using Lemma 4.1.2. It then takes $O(1)$ time to determine whether this occurrence of X_i is followed by an occurrence of b at position $j - ||FinishedPrefix|| - ||X_i||$. This determines whether the occurrence of $FinishedPrefix$

at position j is followed by $X_i b = P_i$. Therefore, it takes $O(1)$ time to determine, for each element of I_{i-1}, whether it remains in i.

By the naive algorithm, each P_i has $O(||P_i||)$ occurrences, because P_i is not a node of the tree, hence its occurrences can only be recorded in ancestors (prefixes) of X_i. Therefore, $||I_i|| = O(||P_i||)$. Determining I_l therefore takes $O(\sum_{i=1}^{l-1} ||I_i||) = O(\sum_{i=1}^{l-1} ||P_i||) = O(||P||)$ time.

4.1.1 Returning Positions One-by-One in Left-to-Right Order

It is sometimes claimed that the suffix array returns all k occurrences of P in $O(m + \log n)$ time, even though k can be superlinear in this bound. The reason is that it gives a pointer to a list of the positions. This time bound captures the fact that if the user wants to examine the first k' positions, this takes $O(k')$ rather than $\Theta(k)$ time. One way to view this is that it returns an iterator in $O(m + \log n)$ time that then takes $O(1)$ time per position to return the positions.

The position heap can be implemented to have this property also, using a depth-first search that maintains a stack of active calls that have not yet made a recursive call on their last child. One use of such an iterator, however, is to examine the *first k'* positions *in left-to-right order*. This is a common operation in text editors, for example. This can be implemented in $O(\log k)$ worst-case time per element, due to the fact that the node labels have the heap property.

We illustrate how to produce an iterator that returns them in right-to-left order; left-to-right order can be obtained by building the augmented position heap for the reverse of the text. The positions of nodes on the indexing path X take $O(1)$ time to check. If $P = X$, then the descendants of X might also have to be returned in left-to-right order. Keep a priority queue on the topmost nodes of the subtree of X whose positions have not yet been returned. Because the positions have the heap property, the minimum position is among these nodes. Initially the priority queue has X in it. Each time a new position is asked for, the minimum index i in the priority queue is returned, and the positions in the children of the node containing i are inserted to the priority queue. Since $\Sigma = O(1)$ and the size of the subtree is $O(k)$, the size of the priority queue is $O(k)$, and extracting i and inserting its children takes $O(\log k)$ time.

Chapter 5

Linear Time Algorithm to Build the Position Heap and the Augmented Position Heap

The above naive algorithms' preprocessing time cannot compete with other construction time. However, the time bound can be improved with the position heap, the dual heap and the depth of each node. In this chapter, we show how to build the position heap and the augmented position heap in $O(n)$ time.

5.1 Building the Position Heap in $O(n)$ Time

Each time a node is added to the position heap, its parent must be located so that it can be added as a child. The reason the above algorithm for constructing the position heap from the root does not take $O(n)$ time is that indexing from the root to find this parent at each iteration is not an $O(1)$ operation.

5.1.1 The Strategy

Indexing into the heap from the root is not the only way to find the parent of the new node at step i. Let X_{i-1} be the node added at step $i-1$, let the first letter of T_i be b, that is, let $T_i = bT_{i-1}$, and let X_i be the node added at step i. Since X_{i-1} is a prefix of T_{i-1}, $X_i = bY$, where Y is a prefix of T_{i-1}, hence a (not necessarily proper) ancestor of X_{i-1}. By Lemma 5.1.3, below, $||X_i|| = ||bY|| \leq ||X_{i-1}|| + 1$. In other words, the depth of the added node can increase by at most one at each iteration. This suggests the idea building a separate structure, which we will call the *dual heap*, for finding the parent of $X_i = bY$, given Y. We may then search upward from X_{i-1} in the position heap, instead of downward from the root, in order to find Y. This is not an $O(1)$ operation, because we may have to search upward through a lot of ancestors of X_{i-1} to find Y, but each ancestor that we traverse decreases the depth of the next node, $X_i = bY$ by 1. Since the depth can build back up at the rate of at most one per iteration, this will give an $O(1)$ amortized bound per iteration.

Since Y can be much shorter than X_{i-1}, the upward search might have to proceed through a large number of nodes on the path from X_{i-1} toward the root before Y is reached. However, the new node at step i, bY is then much shorter than the node, X_{i-1}, inserted at the previous iteration. The cost of the operation is proportional to the decrease in depth from one iteration to the next. What makes the approach more efficient than the above approach is that depth of the new node inserted at successive iterations can grow by at most 1 from one iteration to the next, by Lemma 5.1.3. This allows us to amortize occasional large costs incurred in iterations where the depth decreases by a large amount over many iterations where the depth slowly builds up again at the rate of one per iteration.

The argument is the same as that for a stack with a *multipop* operation described in the chapter *Amortized Analysis* in the textbook [12].

5.1.2 Implementation

The following lemma is the basis of the claim that the depth in the tree at which the algorithm works must build up again slowly if there is a sudden large and costly decrease in the depth.

Lemma 5.1.1 *If P is not a node of $H(T)$, it has fewer than $||P||$ occurrences in T.*

Proof: Every suffix of T that has P as a prefix results in a new node of the tree that is either a proper prefix of P or that has P as a prefix. Since P does not occur in the tree, it is not a prefix of any node in the tree. Therefore, the number of suffixes of T that have P as a prefix, hence the number of occurrences of P, is bounded by the number of proper prefixes of P.

Let us say that a set of $\mathcal{S}$ of strings is **hereditary** if, whenever $X \in \mathcal{S}$, every substring of X is also in $\mathcal{S}$.

Lemma 5.1.2 *The nodes of the position heap are a hereditary set of strings.*

For example, in the final tree, node *abaa* is labeled with position 13 of Figure 4.3. Its substrings *aba*, *baa*, *ab*, *ba*, *aa*, *a*, *b*, and the empty string are all nodes of the position heap; they are labeled with positions $10, 12, 5, 6, 11, 2, 3, 1$, respectively.

Proof: Let us show this by induction on the length of $T_i = t_i t_{i-1} ... t_1$. The lemma is trivially true for $H(T_1)$, which has only one node, the empty string. Otherwise, we adopt as the induction hypothesis that the nodes of $H(T_{i-1})$ have the hereditary property. Since $H(T_i)$ differs from $H(T_{i-1})$ only by the addition of a node X, $H(T_i)$ can only fail to have the hereditary property if some proper substring of X fails to be a node of T_i.

This can't be the case if $||X|| < 2$, since λ is a node of $H(T_i)$. Suppose $||X|| \geq 2$. We can then write X as $aX'b$. The parent of $aX'b$ is aX', hence it is a node of $H(T_{i-1})$. Since

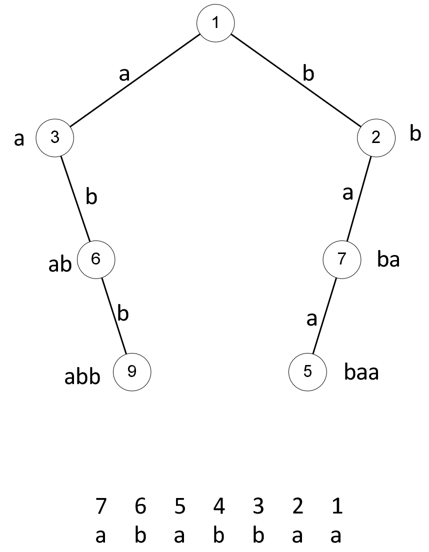


Figure 5.1: The hereditary property doesn't necessarily apply when the suffixes are not inserted in order of ascending length. The figure depicts the Coffman and Eve structure where the insertion order of the suffixes is $(T_1, T_4, T_2, T_7, T_5, T_6, T_3)$. String *abb* is a node, but its substring *bb* is not a node of the tree.

aX' is longer than X', X' is a node in $H(T_{i-2})$ by the induction hypothesis. Also, $X'b$ is a prefix of T_{i-1}, and since X' is a node of T_{i-2}, $X'b$ is either added at step $i-1$ or is already a node of T_{i-2}. In either case, it is a node of $H(T_{i-1})$. We conclude that aX' and $X'b$ are nodes of T_{i-1}. By the induction hypothesis, every substring of aX' and $X'b$ is a node of T_{i-1}, hence of T_i, and these are every proper substring of the new node $X = aX'b$.

This hereditary property is not shared by arbitrary instances of Coffman and Eve's data structure, as the node labeled 9 in Figure 3.1 is the string *abba*, but its substring *bba* is not a node of the tree. It is not even true when the keys are the suffixes of a text T when they are not inserted in ascending order of length. Figure 5.1 gives an example.

Lemma 5.1.3 *For* $1 < i \leq ||T||$, *if* X_{i-1} *is the node inserted at step* $i-1$ *and* X_i *is the node inserted at step i, then* $||X_i|| \leq ||X_{i-1}|| + 1$.

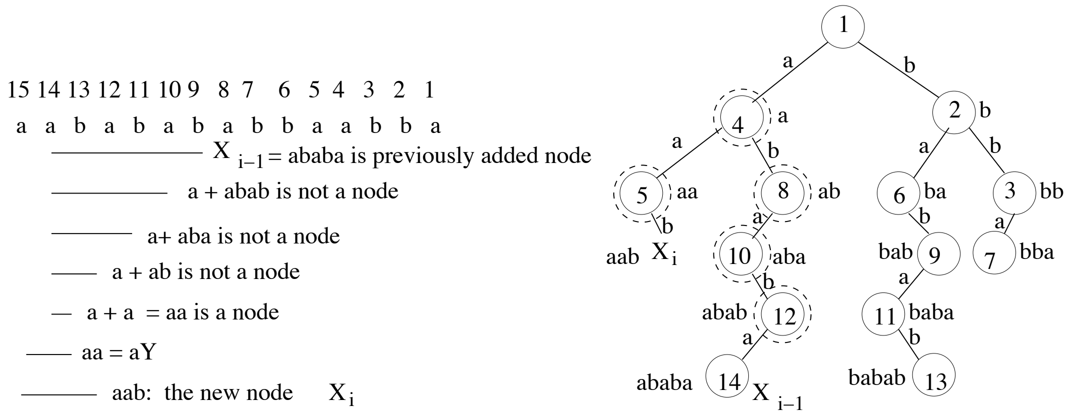


Figure 5.2: Given the node X_{i-1} added at step $i-1$, find the parent of the node X_i added at step i.

Proof: Let a denote the first letter of T_i. X_{i-1} is the shortest prefix of T_{i-1} that is not already a node of $H(T_{i-2})$ and X_i is the shortest such prefix of $aT_{i-1} = T_i$. Let b denote the last letter of X_i. Then X_i can be written as aYb for some string Y.

Suppose $||X_i|| \geq ||X_{i-1}|| + 2$. Then X_{i-1} is a *proper* prefix of Yb. Since X_{i-1} is the longest prefix of T_{i-1} that is not a node of $H(T_{i-2})$, Yb is not a node of $H(T_i)$. By the hereditary property, Yb is a node of $H(T_i)$, since it is a substring of X_i, which is a node of $H(T_i)$. The only new node added to $H(T_{i-1})$ to get $H(T_i)$ is aYb, so Yb was already a node of $H(T_{i-1})$, a contradiction.

To insert a node to the position heap, we must find the parent. Since inserting the node after the parent is found takes $O(1)$ time, the only obstacle to getting a linear time bound is repeated indexing into the position heap to find the parent of each node to be added. We must use an alternative method to find this parent.

The idea of our $O(n)$ alternative method is given in Figure 5.2. At step $i-1 = 14$, we add $X_{i-1} = ababa$ as a new node. At step $i = 15$, we must add the shortest prefix of $H(T_i)$ that is not already a node of the position heap. Let a denote the first letter of T_i.

If the string a does not already occur as a node of the position heap, then it can be added as a child of the root in $O(1)$ time.

Otherwise, as in the proof of Lemma 5.1.3, the new node must be aYb for some prefix Yb of the node X_{i-1} added in step $i-1$, where b is the character occurring $||aY||+1$ positions into T_i.

Below, we show how to find, for each such prefix Y of X_{i-1}, whether aY is already a node of the position heap, and if so, to return a pointer to it, in $O(1)$ time apiece. We try this on all proper prefixes of X_{i-1} in descending order of length until we find the first. In the figure, we let Y take on the sequence of values $(abab, aba, ab, a)$, whereupon it is discovered that $aY = aa$ is already a node of the position heap, and since the concatenation of a and ab is not, aa is the longest prefix of T_i that is already a node of the position heap. We have found the desired parent of the new node. The new node, $X_i = aYb$, is added as its child of aY on an edge labeled with letter b.

This does not give an $O(1)$ bound to add each node of the tree. However, we can amortize the variable costs, showing that they sum to $O(n)$ over all iterations.

The reason the cost of step i is not $O(1)$ is that we might have to try many prefixes Y of T_{i-1} before we find the one such that aY is already a node of the heap. Let the *decrease in depth* denote the difference $||X_i|| - ||X_{i-1}||$ of the depth of the node added at position $i-1$ and the depth of the node added at position i. If this is negative, call it an *increase in depth*. If at step i, we try k_i prefixes before finding Y such that aY is already a node of the tree, then we spent $O(k_i)$ time on the step, and $||X_i|| = ||aYb|| = ||X_{i-1}|| - (k_i - 2)$. The decrease in depth is $k_i - 2$. The first two prefixes take $O(1)$ time, so the time spent at step i is $O(1)$ plus the decrease in depth. By Lemma 5.1.3, the depth can increase by at most 1 at each iteration, so the total increase in depth is $O(n)$ over all iterations. The

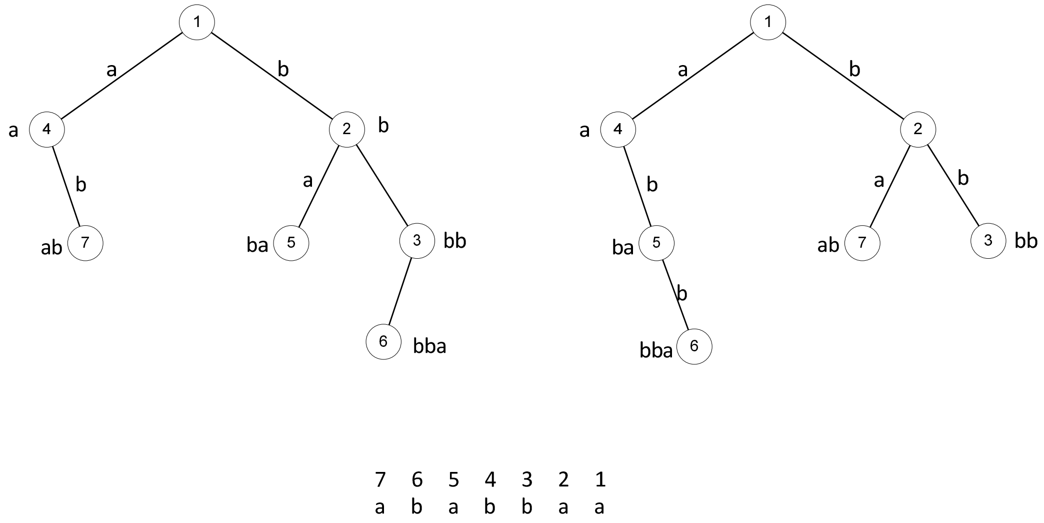


Figure 5.3: The position heap and its dual for the text *abbabbb*. The labels of the path leading to a node in the dual is the reverse of the labels of the path leading to it in the position heap.

total decrease in depth can't exceed the total increase in depth, which means that over all iterations, the total decrease in depth is $O(n)$. Therefore, the total time spent by the algorithm is $nO(1)+O(n)=O(n)$.

It remains to describe how to get an $O(1)$ bound for finding, for each prefix Y of X_{i-1}, whether aY is already a node of the heap.

Definition 5.1.4 *Let the* **dual** $D(T)$ *of the position heap* $H(T)$ *be the trie where for each node* X *of* $H(T)$*, the* **reverse** X^R *of* X *is a node of* $D(T)$*. (see Figure 5.3).*

We continue to refer to each node by its path label X in the position heap, even when considering it as a node of the dual. Equivalently, each node of $D(T)$ is denoted by the sequence X of labels on edges from the node to the root of $D(T)$.

It is tempting to think that the dual is just the position heap of the reverse of the text, but it is easily verified that this is not the case.

Lemma 5.1.5 *The set of nodes of $D(T)$ is the same as the set of nodes of $H(T)$.*

Proof: Because for every node X of $H(T)$, there is a node X in $D(T)$, where X is the string of labels from the node to the root in $D(T)$, every node of $H(T)$ is a node of $D(T)$. It remains to show that every node of $D(T)$ is a node of $H(T)$. Let X be an arbitrary node of $H(T)$. By Lemma 5.1.2, not only is every prefix of a node X of $H(T)$ a node of $H(T)$, but so is every suffix. This implies that every ancestor of X in $D(T)$ is a node of $H(T)$. There are no nodes on any path of $D(T)$ that fail to be a node of $H(T)$.

We implement the position heap and its dual on the same set of nodes, so that each node has both a parent in the position heap and a parent in the dual.

We concurrently construct the position heap and its dual. Suppose that at step i we already have $H(T_{i-1})$ and $D(T_{i-1})$. We show how to update both to get $H(T_i)$ and $D(T_i)$ in $O(k_i)$ time.

When going from $H(T_{i-1})$ to $H(T_i)$, let a be the first letter of T_i and X_{i-1} the node added at step $i-1$. (Refer to Figure 5.4.) The prefixes of Y in descending order of length are the ancestors encountered on the path from X_{i-1} to the root of the position heap. For each such ancestor Y, we can find whether aY is already a node of the heap by determining whether Y has a child on an edge labeled a in the dual. This takes $O(1)$ time, since Y is both a node of the heap and of the dual. We stop when we encounter the first one. By the above algorithm, this takes care of adding node aXb to $H(T_{i-1})$, yielding $H(T_i)$ in $O(k_i)$ time.

However, we must also add this node to the dual, which requires locating its parent, Xb, and adding it as a child on edge labeled a. Fortunately, Xb was just the last prefix of Y considered before X was discovered. We already found Xb in the position heap, and since it is also a node of the dual, we have it in the dual. aXb can be added as a child of Xb on edge labeled a in $O(1)$ additional time over what we have accounted for in adding it in the

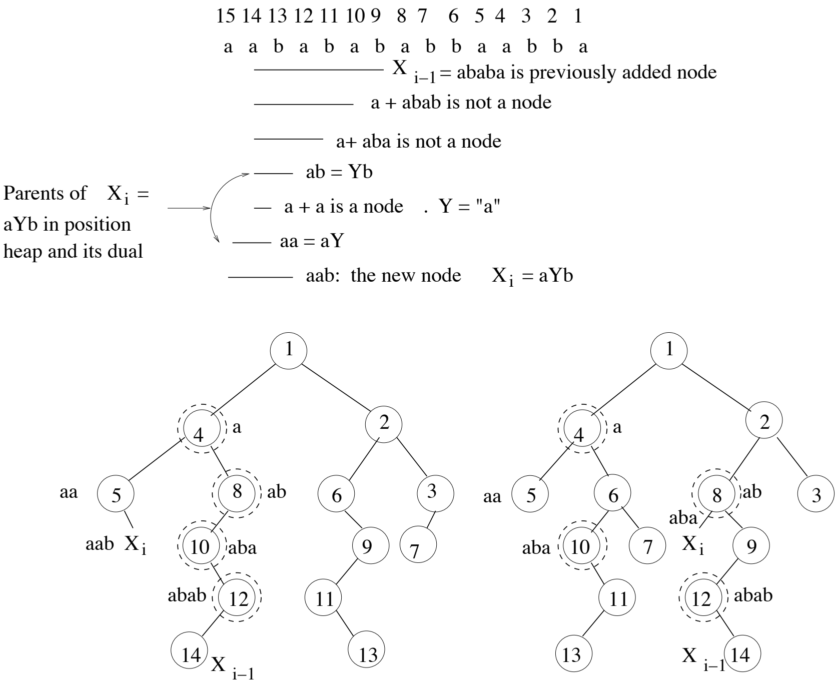


Figure 5.4: Implementing the algorithm of Figure 5.2 using the position heap and its dual. Starting at the previously-added node X_{i-1}, we find the lowest ancestor Y such that aY is already a node. This is accomplished by traversing ancestors in the position heap, and seeing if they have a child on edge labeled a in the dual. In this case Y is the node labeled 4. Its child on edge labeled a in the dual is aY, the node labeled 5. It is the parent of the new node $X_i = aab$ in the position heap. The last prefix ab tried before Y was found is the longest node of the dual heap that is a prefix of X and has no child labeled a. It is the parent of X_i in the dual.

position heap. This gives the following:

Lemma 5.1.6 *It takes linear time to construct the position heap of a text T.*

5.2 Constructing the Augmented Position Heap in $O(n)$ Time

The augmented position heap differs from the position heap in that the nodes are labeled with depth-first discovery and finishing times and with maximal-reach pointers. Depth-first search on a tree with n nodes takes $O(n)$ time, so it only remains to describe how to compute the maximal-reach pointers in $O(n)$ time.

Once again, the strategy is to amortize the cost. The approach is virtually the same as it is for adding new nodes: instead of searching downward from the root at each iteration, we search upward in the tree, starting at the node pointed to by a maximal-reach pointer at the previous iteration. Even though this is not an $O(1)$ operation, the cost is proportional to the decrease in depth of the node pointed to by the maximal-reach pointer. This depth can increase by at most 1 from one iteration to the next, allowing to amortize large decreases in depth over many small increases in depth.

Lemma 5.2.1 *For $1 < i \leq ||T||$, if X_{i-1} is the node pointed to by the maximal-reach pointer of node $i-1$ and X_i by the maximal-reach pointer of node i, then $||X_i|| \leq ||X_{i-1}|| + 1$.*

Proof: Let a denote the first letter of T_i. X_{i-1} is the longest prefix of T_{i-1} that is a node of $H(T)$, and X_i is the longest prefix of $aT_{i-1} = T_i$ that is a node of $H(T)$. Let b denote the last letter of X_i. Then X_i can be written as aYb for some string Y.

Suppose $||X_i|| \geq ||X_{i-1}|| + 2$. Then X_{i-1} is a *proper* prefix of Yb and Yb is not a node of $H(T_{i-1})$. By the hereditary property, Yb is a node of $H(T_i)$, since it is a substring of X_i, which is a node of $H(T_i)$. The only new node added to $H(T_{i-1})$ to get $H(T_i)$ is aYb, so Yb was already a node of $H(T_{i-1})$, a contradiction.

To construct the *augmented* position heap in $O(n)$ time, our strategy is first to construct the position heap in $O(n)$ time using the algorithm from the previous section. As before, we create the array `N[]`, where `N[i]` points to the node that contains position *i*, and this takes $O(n)$ time by trivial methods. We then add the discovery and finishing times and the maximal-reach pointers on a second pass, in $O(n)$ time.

We find and test each prefix by starting at X_{i-1} in the position heap and ascending through ancestors until we find the first one, Y, that has a child on edge labeled a in the dual heap. This child, aY, in the dual is the node to which node i must point.

The analysis of the linear running time is the same as it is for linear-time construction of the position heap. The *current depth* is the depth of node X_i in the position heap. The first two prefixes of X_{i-1} take $O(1)$ time to check for a child on edge labeled a in the dual heap. Each additional prefix takes $O(1)$ time to check, and decreases the current depth in the position heap. Call this the *variable part* of the time spent at position i. By Lemma 5.2.1, the current depth can increase by at most one per iteration. The initial depth is at most 1, since T_1 has length 1. The total decrease in depth can therefore be at most one greater than the total increase in depth, which is $O(1)$ per iteration, hence $O(n)$ overall. The sum of the variable parts of the times spent at the different iterations is therefore $O(n)$. We therefore get the following:

Lemma 5.2.2 *It takes $O(n)$ time to construct the augmented position heap for a text T of length n.*

Chapter 6

Space-Efficient Representation for the Position Heap and the Augmented Position Heap

The position heap needs a less space than the suffix tree and the DAWG. However, it still requires more space than suffix array since the suffix array satisfies with one array. Like a deterministic automaton data structure, each node of the position heap contains its position number and a hash table that is a useful for reducing the time to find a successor or child on a given letter to $O(1)$ expected time. A hash table carries a significant overhead of unused space.

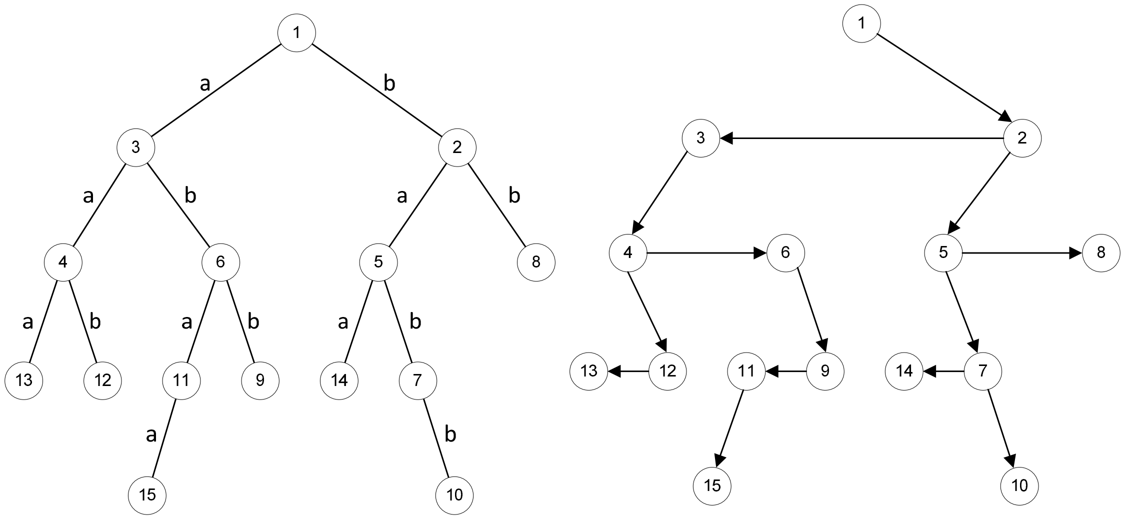
There is a way to eliminate a position number and a hash table from each node, so that only $2n$ integers are required to represent it, and so that only $4n$ integers are required to represent the augmented position heap. The trick is to represent the set of nodes with an array of nodes, and to use array indices in place of pointers to nodes. Our analysis of the time bounds will assume that the alphabet size does not grow as the size of the text grows.

6.1 An Array Representation of the Position Heap

An array can represent compact information for the position heap. Each node has at most two integers. The first integer is the index of the *first child* in a list of children of the node. The second integer is the index of the *next sibling* in a list of siblings of the node; this list is the list of children of the parent of the node. This data structure supports the operation of finding all of the children of a node; go to the first child using the first-child index, and then follow next sibling pointers (indices) until a null index is encountered. The null index can be represented with an integer that cannot be an array index, such as -1.

The array in Figure 6.1 illustrates this for text *"abaaababbabaaba"*. The size of the array is the same as the size of the text and one index in the array stores two integers; the first integer indicates one of its children's index and the second number denotes one of its sibling's index.

This data structure is a well-known one for representing trees. However, a significant advantage of it is that it eliminates the need to label a node with a position in the text T; this is represented implicitly with the node's index in the array. The node corresponding to position i of the text is the one sitting at position i of the array. Whenever a node is accessed, its index is known, so there is no need to label the node explicitly with a position number. A second advantage is that it eliminates the need to explicitly label edges of the tree with letters. As before, if the node with index i has depth d, then the letter implicitly labeling its parent edge is the one sitting at position $i-d+1$ of T. We implement the algorithms so that d is known whenever we access a node, so the label of the edge can be found via a lookup in T, rather than looking for an edge label.



Position heap

A compact representation of the position heap

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1
0,0	0,0	0,0	0,13	15,0	0,0	0.11	0,0	10,14	9,0	7,8	12,6	4,0	5,3	2,0

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1
a	b	a	a	a	b	a	b	b	a	b	a	a	b	a

Figure 6.1: The position heap and its compact representation

6.2 Construction of the Array Representation in $O(nh(T))$

In order to achieve this time bound for constructing the array representation, it suffices to show that we can run the naive construction algorithm, and that whenever we are at a node, we know its index and depth.

To index into the heap, we start at the root, as before. The root is the node sitting at position 0 of the array. Each time we pass from parent to child, we increment a variable d that keeps track of the current depth. The parent gives the index of the child, not just its address, so we have both the depth and the index of each node we pass through, as required. Each time we pass from a parent to a child, we seek the child whose parent edge is labeled with a letter c of a suffix of T. We have given a mechanism for finding the implicit label of each edge in $O(1)$ time, given the depth of its child and the child's index. This child can be therefore found by traversing the list of children until the one is found whose parent edge is implicitly labeled c, in time proportional to the number of children. Since there is at most one child for each letter of the alphabet and the alphabet has size $O(1)$, it takes $O(1)$ time to go from parent to child during indexing. This is the same as the time bound for this operation given earlier, so it has no effect on the running time of the naive algorithm.

Note that the naive algorithm creates the nodes in ascending order of position number. In the array representation, this corresponds to creating the nodes in ascending order of index in the array. Since the size n of the text is known in advance, we allocate an array of n nodes before we begin. On the i^{th} iteration, we must link the i^{th} element of the array into the tree structure. This is accomplished by making the first child of the parent be the i^{th} node's next sibling, and then making the i^{th} node be the new first child of the parent. Since the naive algorithm finds the parent of node i, the operation of linking the node in takes $O(1)$ time, and does not affect the asymptotic bound of the naive algorithm. Let is call this the *child-linking* operation.

Since none of the asymptotic time bounds of any of the operations of the naive construction algorithm are affected, the running time is again $O(nh(T))$

6.2.1 Searching with the Compact Representation in $O(m^2+k)$ Time

In the description of the naive construction algorithm, we have shown how to index into the tree on a given string while spending $O(1)$ time on each node of the indexing path. The $O(m^2+k)$ algorithm requires indexing into the tree on the pattern string, and the time for this operation is unaffected.

Using a depth-first search to visit the subtree of the last node reached takes time proportional to the size of this subtree. We access each node by index number, rather than by address, and this index number is one of the positions of the pattern string. All positions corresponding to descendants of the last node reached on the indexing path takes $O(1)$ time per node, and since they are all reported as occurrences of the pattern string, this takes $O(k)$ time.

In addition, we access all nodes on the indexing path by array index. These indices are the text positions corresponding to, hence the remaining candidates to be occurrences of the pattern. We check each of them to see if it is a true occurrence of the pattern string in $O(m)$ time as before. Since there are $O(m)$ of them, these checks take $O(m^2)$ time.

6.3 Constructing the Compact Representation in $O(n)$ Time

To construct the compact representation of the position heap in $O(n)$ time, rather than in $O(nh(T))$ time, we will require $3n$, rather than $2n$ integers.

During the $O(n)$ construction algorithm, we need the following two operations:

1. Given a node and a letter c, find the child of the node on the edge labeled c in the dual heap.

2. Given a node, find its parent in the position heap.

We therefore implement the dual heap using the data structure given above for the position heap, using two integers per node. During construction of the position heap, since we only need to support the operation of finding the parent, we use a simple array of n integers for the position heap, where n is the length of T. The integer at index i is the index of the parent of the node corresponding to position i. This requires a total of two integers per position of T for the dual heap and one integer per position of T for the position heap, for a total of $3n$ integers. Figure 6.2 demonstrates the array representation of the position and dual heap.

We must also justify that we can know the depth of each node we visit during the construction algorithm. This is a little bit trickier than it was in the naive construction algorithm, since indexing does not start at the root, but rather at the previous node linked into the tree. However, if we know the depth of a node of the dual, we increment this depth by 1 to get the depth of the child given by Operation 1 above. If we know the depth of a node in the position heap, we decrement this depth by 1 to get the depth of the parent given by operation 2, above.

All traversal of the tree is accomplished by Operations 1 and 2 during the $O(n)$ construction algorithm. By incrementing and decrementing the depth, depending on whether we are performing operation 1 or 2, we know the current depth at all times.

When a new node is linked in, it must be linked into both the position heap and its dual. The $O(n)$ algorithm links them in in ascending order of position number, which corresponds to ascending order of index in the arrays. For the position heap array, when we add node *i*, we let the i^{th} element store the index of the parent in the position heap. In the dual heap, we link it in as a child of its parent in the dual, using the *child-linking* operation described above.

A final issue is changing the representation of the position heap where each node has a pointer to its parent into one where each node has a pointer to its list of children. Let us call the first representation and *upward-pointing representation* and the second a *downward-pointing representation*. This will require changing its representation from an array with one integer per element (the parent) to a representation with an array with two integers per element (the first child and the next sibling).

By the time we need to perform this conversion, the dual heap is no longer needed. Therefore, we can reinitialize the array for holding the downward-pointing representation of the dual heap, and reuse it for storing the downward-pointing representation of the position heap. For each node *i* in ascending order of index number, we find the parent *j* of *i* using the upward-pointing representation. In the downward-pointing representation, we link *i* in as a child of *j*, using the *child-link* operation. Note that we do not know the depth of *j*, so we do not know the character labeling the new downward-directed tree edge from *j* to *i*. However, this is not required to link it in; the label of the edge is an implicit letter that we do not know at this point. That will be known whenever we use the edge during the searching operation, as described above, since the depth of the edge will be known at that time.

When we are done, we have both an upward-pointing and a downward-pointing representation of the position heap, and at no point have we used more than $3n$ integers. The upward-pointing one is not needed for searching, so it may be discarded, leaving a space requirement of $2n$ integers thereafter. The key bottleneck for the space requirement is the

$3n$ integers required for the construction.

6.4 Constructing a Compact Representation of Augmented Position Heap

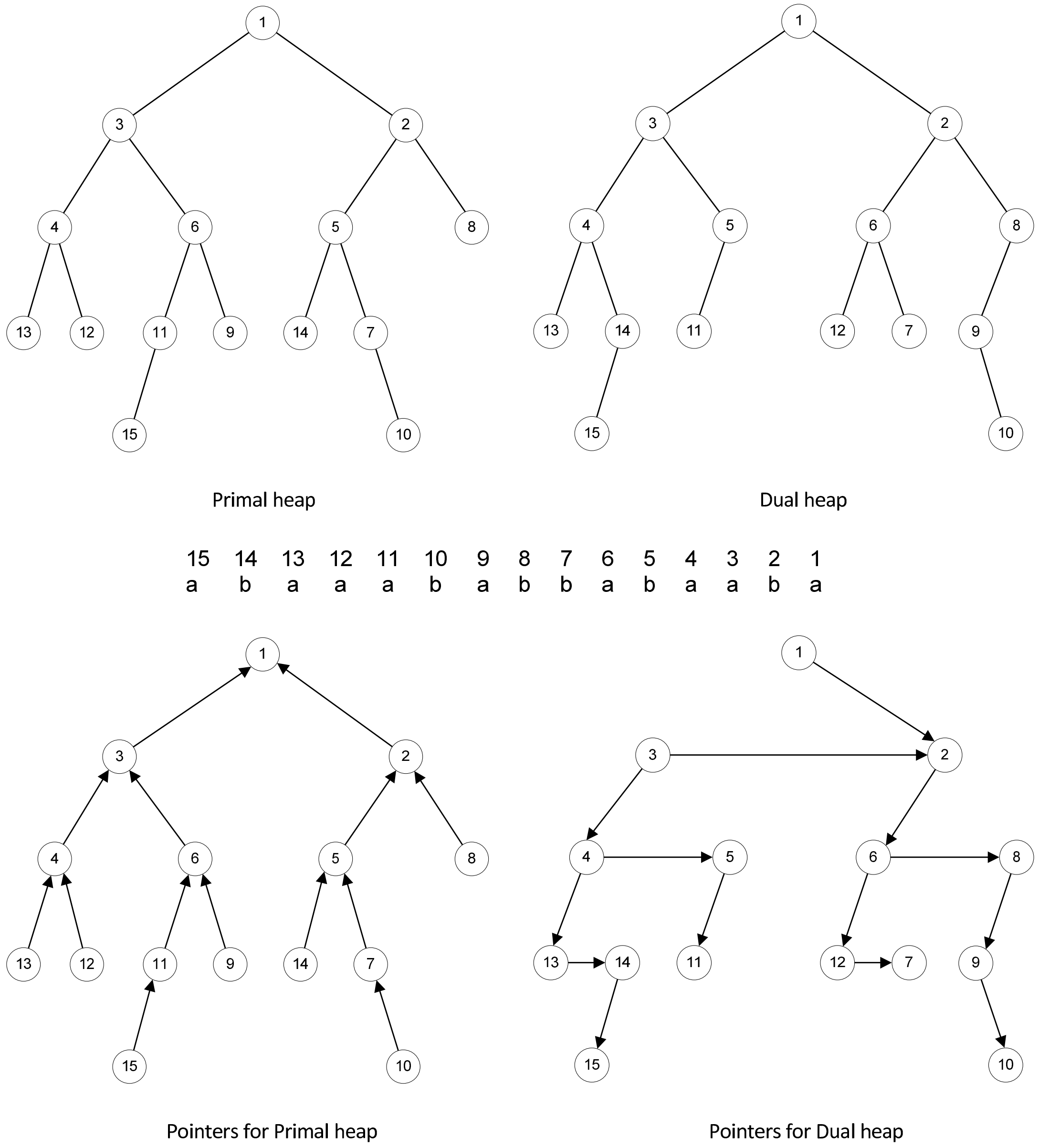
To construct a compact representation of the augmented position in $O(n)$ time, we will need $4n$ integers, rather than $3n$ for constructing the simple position heap in $O(n)$ time, or $2n$ for constructing the simple position heap in $O(nh(T))$ time.

Since pointers are replaced with array indices, the maximal-reach pointers are turned into integers, giving array indices of the nodes they point to, rather than their addresses in memory. Rather than directly labeling nodes of the position heap with these maximal-reach pointers, we use a separate array $M[]$, where $M[i]$ stores the index of the node pointed to by the maximal-reach pointer of node i. This will be convenient for keeping the space requirement down during the construction phase.

Similarly, we keep the discovery and finishing times in two arrays, $D[]$ and $F[]$, where $D[i]$ and $F[i]$ give the discovery and finishing times of node i. To keep the bottleneck of the space requirement down, we don't allocate space for these arrays until other elements required for construction of the position heap and installation of the maximal-reach pointers can be deallocated.

6.4.1 An implementation that uses $5n$ integers

First, we describe how to construct it using $5n$ integers. The trick is to use the dual heap during construction of the upward-pointing position heap, as described in the $O(n)$ algorithm above. This takes n integers for the position heap and $2n$ for the dual. We then use the dual to install the maximal-reach pointers on the nodes of the upward-pointing position



Primal heap

Dual heap

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1
a	b	a	a	a	b	a	b	b	a	b	a	a	b	a

Pointers for Primal heap

Pointers for Dual heap

Figure 6.2: The compact representation for position and dual heap

heap. Since each node of the position heap is now labeled with a parent and a maximal-reach pointer (index), the position heap also takes $2n$ integers, for a total of $4n$ for the two heaps.

At this point, the dual heap is no longer needed. We reuse its space to create a downward-pointed representation of the position heap, as described above. The array $M[]$ of maximal-reach pointers does not need to be touched during this conversion. The total space requirement so far is still $4n$ integers. We can now deallocate the upward-pointed representation of the position heap, since it is no longer needed. At this point, space for $3n$ integers is allocated. At this point, we can allocate space for the discovery and finishing time arrays, $D[]$ and $F[]$, and fill them in with a depth-first search on the downward-pointed representation of the position heap. At this point, space for $5n$ integers is allocated, which is the bottleneck.

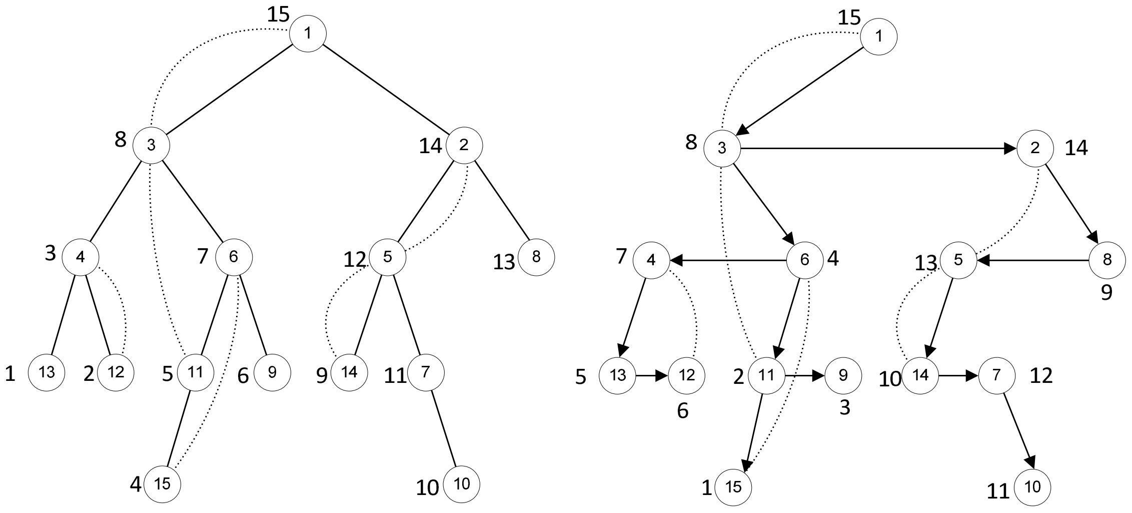
6.4.2 An implementation that uses $4n$ integers

Getting the space requirement down to four integers depends on a result I have recently obtained and that have not yet been published: it suffices to label all nodes only with finishing times, omitting the discovery times, and still get the $O(m+k)$ bound for queries. Figure 6.3 shows the array representation with $4n$ integers.

The reason for labeling nodes both with discovery and finishing times is to accomplish the following:

- Result (*): Given pointers to two nodes v_1 and v_2, determine whether v_2 is a descendant of v_1 in $O(1)$ time.

Together with the maximal-reach pointers, this allowed us to reduce the time to check, for each candidate position on the indexing path, whether it is an occurrence of the whole



Augmented position heap

Compact representation for Augmented position heap

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1
a	b	a	a	a	b	a	b	b	a	b	a	a	b	a

Implementation with the array node of the augmented position heap	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1
	0,0, 1,0	0,7, 10,0	0,12, 5,0	0,0, 6,0	15,9, 2,0	0,0, 11,0	0,0, 3,0	0,5, 9,0	10,0, 12,11	11,4, 4,1	14,0, 13,10	13,0, 7,6	6,2, 8,2	8,0, 14,13	3,0, 15,8

Figure 6.3: Augmented position heap and its compact representation

string of edge labels on the indexing path. This was the case if and only if the maximal-reach pointer pointed to a descendant of the last node on the indexing path, but the above result was needed to determine this in $O(1)$ time for each of the $O(m)$ nodes on the indexing path.

Let us now observe that we can accomplish Result (*) in $O(1)$ time if we know the earliest finishing time i of any node in the subtree rooted at v_1. Then v_2 is a descendant of v_1 if and only if the finishing time of v_2 lies between i and the finishing time j of v_1. Figure 6.4 shows the example.

The possibility of using this was ignored in our earlier work because it seemed impossible to find i in $O(1)$ time. The obvious way would be to start a depth-first search at v_1 until the first node is marked with a finishing time, but this might require traversing $\Theta(h(T))$ nodes to find this node.

Instead, let v_3 be the the last node that finished before the depth-first search discovered v_1. The first key observation is that i is one greater than the finishing time of v_3 if v_3 exists, and $i = 1$ otherwise. Therefore, if we apply only finishing times and we know v_3, we can achieve Result (*). However, this then reduces the problem to finding v_3, which is not an $O(1)$ operation. The second key observation is that v_3 is a child of a node on the path from the root to v_1. The only place where we apply Result(*) is when v_1 is the end of the indexing path, which we must traverse in order to find v_1. Therefore, even though finding v_3 is not an $O(1)$ operation, its cost is subsumed by the cost of finding v_3; once we know the finishing time of v_3, we can traverse the path again, finding the latest finishing time of any child of a node on the path that has an earlier finishing time than v_1's finishing time.

To prove the correctness, it suffices to prove the following:

Lemma 6.4.1 *Let v_1 be a node of a rooted tree, and let v_3 be the last node that finishes*

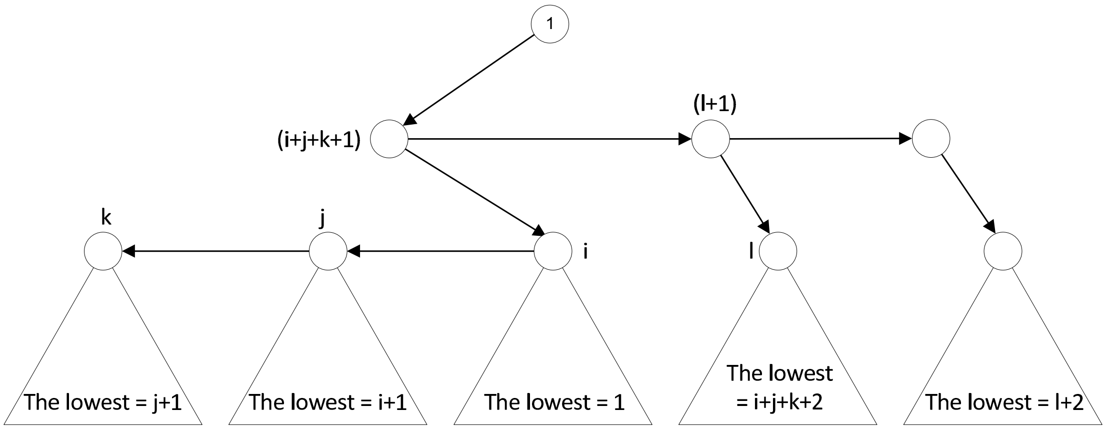


Figure 6.4: Finishing times with the compact representation. A letter beside of each node indicates a finishing time. The first subtree contains the lowest finishing time among siblings.

before v_1 is discovered during a depth-first search. The location of v_3 is at child of a proper ancestor of v_1.

Proof: Let us say that a node is *white* if it has not been discovered yet, *gray* if it has been labeled with a discovery time but not with a finishing time, and *black* if it has been labeled with a finishing time. When a node v is discovered, its entire subtree is blackened before any node outside of its tree is labeled, and v gets the latest finishing time of any node in the tree. When v_3 is discovered, its ancestors are the set of gray nodes, so the maximal blackened nodes are children of these ancestors, each of them has the latest finishing time of the subtree rooted at it, and one of them must have the latest discovery time of all black nodes.

6.4.3 Searching with the Compact Representation in $O(m+k)$ Time

The m^2 factor can be removed from the naive query time bound. As seen in Chapter 4.1, the time bound of testing each node on the path is reduced $O(1)$ with the augmented position heap from $O(m)$ with the position heap.

We check all nodes on the indexing path by array index. We check each of them in $O(1)$ time with maximal-reach pointers and finishing times. Since there are $O(m)$ of them, these checks take $O(m)$ time.

Chapter 7

Updating the Position Heap and the Augmented Position Heap when the Text is Edited

When a block of characters is inserted to or deleted from a text T, the position heap must pass through a series of steps in which it is a trie, but has some things wrong with it that must be repaired in order for it to be the position heap of the new text. The goal of this section is to give algorithms for `Delete` and `Insert`, which update the position heap when a block of text is deleted from or inserted to the text T.

Since the text is no longer static, it is no longer convenient to label a node of the position heap with its position *number* in the text; when a position is deleted, the position numbers of all letters to its left decrease by one. To avoid having to update the position-number labels of all those nodes, we instead label the nodes with *position pointers* to the positions of the text. This requires us to define the analog of the heap property when pointers, rather than integers, are used.

Definition 7.0.2 *If p is a pointer to a position in T, let T_p denote the suffix of T that begins at p. If X is a node in the trie with a pointer to a position of T, let $p(X)$ denote this pointer.*

The trie has the **heap property** *if whenever Y is a child of X, $p(Y)$ is to the left of $p(X)$ in T. The pointer $p(X)$ is* **correctly placed** *if X has an occurrence at position $p(X)$, that is, if X is a prefix of $T_{p(X)}$.*

The constructive definition of the position heap (3.2.1) remains unchanged, except that each time a position is inserted, the new node is labeled with a pointer to the position, rather than its position number.

It will be convenient to look up the corresponding position-heap node given a pointer to a position in the text T. This is accomplished by labeling each position p of the text with a pointer $N(p)$ to the node of the position heap that points to it. This serves the same function as the array $N[]$ in the static case. To avoid the need to mention this pointer each time we move a pointer in the position heap, we will define the operation of moving a position p from one node to another in the position heap as including the operation of making the pointer $N(p)$ point to the new node.

The following lemma is useful for establishing that a procedure for updating the position heap after an edit operation on T has correctly produced the position heap for the modified text.

Lemma 7.0.3 *A trie H where each node is labeled with a pointer to a letter of a text T is the position heap for T if and only if it satisfies the following properties:*

1. *H has the heap property;*
2. *Every position of T is pointed to by* **at most** *one pointer $p(X)$ for some node X in the trie;*
3. *Every position of T is pointed to by* **at least** *one pointer $p(X)$ for some node X of the trie;*

4. *For every node X, $p(X)$ is correctly placed.*

Proof: By induction on the number of positions inserted by the naive construction algorithm.

7.1 Deleting or Inserting a Block of Text in T

The workhorses of the algorithm for updating the position heap after insertion or removal of a block of text are `Remove` and `Add`. Below, we explain how they work, but for now, we define the problems in terms of their preconditions and postconditions so that, for the time being, we can make calls to them in our implementation of `Delete` and `Insert`.

Definition 7.1.1 The problems solved by `Remove` and `Add`

An input to `Remove` *or* `Add` *is a trie that satisfies properties properties 1 and 2 of Lemma 7.0.3, but might not satisfy properties 3 and 4.*

- *An additional input to* `Remove` *is a node X that contains a position pointer to be removed from the set of position pointers in the trie. It removes the pointer without disrupting the heap property, without otherwise changing the set of position pointers in the tree,* **and without creating any new violations of property 4 at any position pointers.**

- *An additional input to* `Add` *is a position pointer to be inserted to the trie. The position pointer must not already occur in the trie. It correctly places the pointer to without disrupting the heap property, without otherwise changing the set of position pointers in the tree,* **and without creating any new violations of property 4 at any position pointers.**

A call to Remove *or* Add *must update a variable h that gives the current height of the trie.*

Implementation requires shuffling position pointers in the tree in a way that is familiar to anyone who has studied heaps. Details are given below. In the meantime, given the problems solved by Remove and Add, we can now explain the main procedures of the section, Delete and Insert, in terms of calls to Remove and Add.

The Delete procedure updates the position heap when a block of characters is deleted from the text so that it is the position heap of the new text.

Definition 7.1.2 *An algorithm for* Delete

- *Let h be the height of the input position heap.*
- *Call* Remove *and* Add*, using the modified text, on the $h-1$ characters that lie to the left of the deleted block.*
- *Using* Remove*, remove the position pointers to the deleted characters.*

The reason for the second step is illustrated in Figure 7.1, where, for ease of understanding, the positions are identified with their position numbers in the original text, rather than with position pointers. The figure depicts the situation that arises when Delete is performed on just a single character, the one at position 10.

Each position p that is correctly placed is stored at a node X that has an occurrence at position p in the original text. Position 11 is not involved in the edit, but it is no longer correctly placed because it is stored at *abb*. The occurrence of *abb* previously at position 11 no longer occurs, because the first *b* of it has been deleted at position 10. Therefore, position 11 is no longer correctly placed. Therefore, Delete calls Remove and Add on position 11 so that it is correctly placed, without creating any new incorrect placements. All such un-edited positions that become incorrectly placed as a result of an edit lie within $h-1$ positions to the left of the edited position, because $h-1$, being the height of the tree, is the maximum length of the string that must occur at a position in order for it to be correctly placed. Let us call these $h-1$ positions the **affected positions**; they are not edited by the edit operation, but their placement in the heap is nevertheless affected by the edit. [1]

Lemma 7.1.3 `Delete` *is correct.*

Proof: Initially, no pointer to the right of the edited position is incorrectly placed, since the naive algorithm inserts these in the same way, whether it is operating on the unedited or edited text. As explained above, an incorrectly placed pointer to the left of the edited positions must be among the $h-1$ affected positions. The pointers to the block of positions that have been removed are also incorrect.

Each call to `Remove` followed by `Add` correctly places an affected position pointer p without changing the set of position pointers, disrupting the heap property, or creating new violations of property 4 at any other position pointers. The net effect of this is therefore to reduce the number of incorrectly placed position pointers by one, while maintaining the other properties. There are at most $h-1$ affected positions immediately to the left of the edited position, so when the second step has finished, there are no incorrectly placed position pointers to the left of the edited positions.

[1] We have defined `Delete` so that it performs a `Remove` and `Add` on positions on the four positions 11 through 14, since the height of the tree is 5. However, it is unnecessary to perform this operation on position 13. It is easy to see by Lemma 5.1.3, that once such a correctly-placed position is found to the left of the edited position, all positions to the left of the position are correctly placed. This observation can make the algorithm run somewhat faster, but since it does not affect the asymptotic bound, we omit the proof of it here.

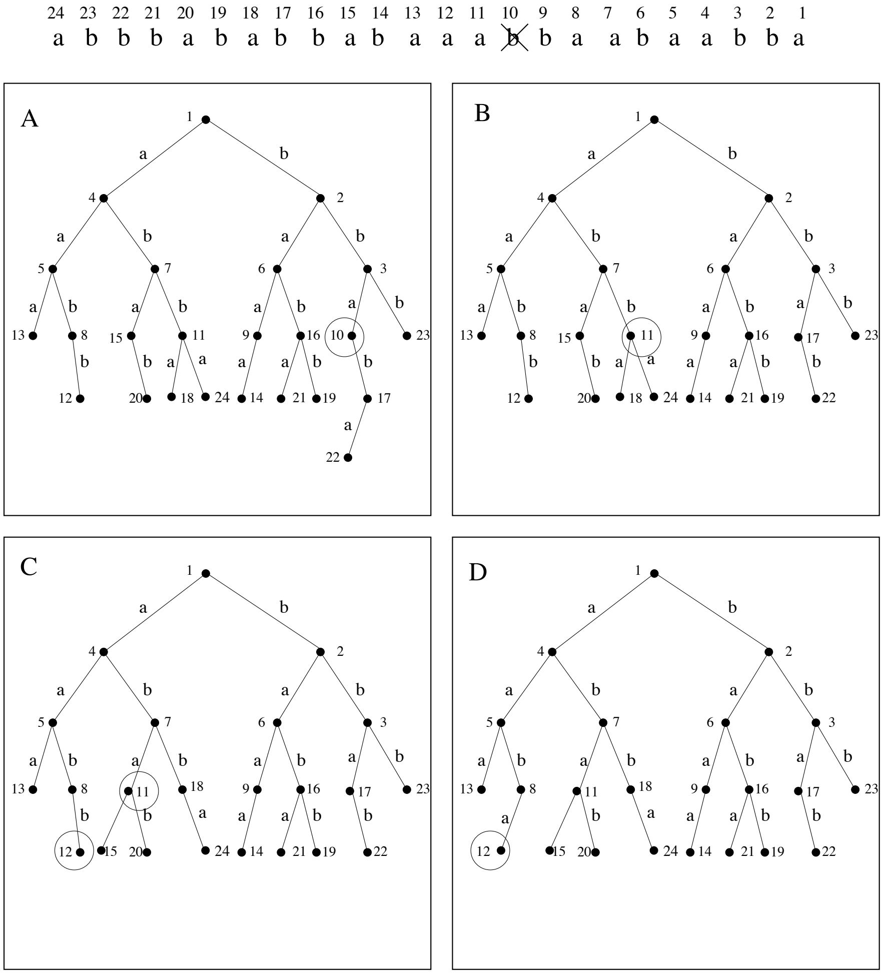


Figure 7.1: Deletion of the *b* at position 10. First, position 10 is removed from the trie with `Remove` (Figure B). Since position 11 resides at node *abb*, position 11 is supposed to be an occurrence of *abb*. This is no longer the case, because the deletion of position 10 removes the first *b* from this occurrence. Position 11 is no longer correctly placed, and this is fixed with a call to `Remove`, followed by a call to `Add` that correctly places it using the new string (Figure C). Similarly, position 12 is no longer correctly placed and this must be repaired in the same way (Figure D). If no node is a string that is longer than *h*, there can be no more than $h-1$ positions to the left of the edited position that are affected in this way.

Because `Remove` removes the position pointers to the block of b defunct positions, does not otherwise change the set of position pointers, and does not cause any new position pointers to be incorrectly placed. It follows that after this step, the trie adheres to all four properties of Lemma 7.0.3, so the modified trie is the position heap of the modified text.

The `Insert` procedure updates the position heap when a character is inserted to the text.

Definition 7.1.4 Implementation of *An algorithm for* `Insert`

- *Using* `Add`*, insert position pointers to the b new characters into the trie.*
- *Let h be the current height of the trie.*
- *Call* `Remove` *and* `Add` *on the $h-1$ characters that lie to the left of the inserted block.*

The proof of correctness of `Insert` is identical to that of `Delete`; it is essentially the inverses of the sequence of calls to `Add` and `Remove` in `Delete`.

7.2 Algorithms for `Remove` and `Add`

We now give algorithms for the `Remove` and `Add` operations, which remove or add a single position pointer to one of the intermediate tries during a call to `Delete` or `Add`. The `Remove` operation is illustrated in Figure 7.2.1

Definition 7.2.1 `Remove` **on position** p**.** *The position p must be removed from the node X that contains it in the current trie. Though a pointer to X is given by $N(p)$, it is convenient to find the path from the root to X by indexing into T_p, where T is the text before the edit operation. This text is known from the new text and the edited block, which are both known* *by the call to* `Delete` *or* `Insert` *that calls* `Remove`*. This gives the depth of X if X is not the root.*

Removing p leaves X with an empty position pointer. This must be filled without violating the heap property. This can be accomplished by finding the child Y of X whose position pointer into T is rightmost, and promoting (moving) that position pointer to the parent. This, in turn, leaves an empty position pointer at Y , which may be filled recursively, ending at a base case where the node is a leaf, which is deleted.

To update the record h of the height of the trie, we assume that `Remove` *and* `Add` *jointly maintain a list that gives, for each depth, a count of the number of nodes at that depth. The counter at the depth of the removed leaf is decremented, and if this counter goes to 0, it is removed from the end of this list and the variable h is decremented.*

The `Add` operations is illustrated in Figure 7.2.

Definition 7.2.2 `Add` **position** p**.** *If p is the position pointer to be inserted, we index on T_p until we can't index any farther, or else a position q to the left of p is found.*

If we cannot index any farther, let X be the last node on the indexing path. We create a new child Y of X reachable on letter $||X||+1$ of T_p and store p in it.

Otherwise q must be **pushed down** *to preserve the heap property. Let X be the node that contains q. The push-down operation is accomplished as follows. As a base case, if X has no child Y reachable on character $||X||+1$ of T_q, then such a child is created and q is stored in it. If Y exists, the position pointer r in Y is pushed down recursively, and q is stored in Y.*

To update the record h of the height, increment the counter for the number of nodes at the depth of the new leaf, and increment h if the new leaf is the first node at that depth.

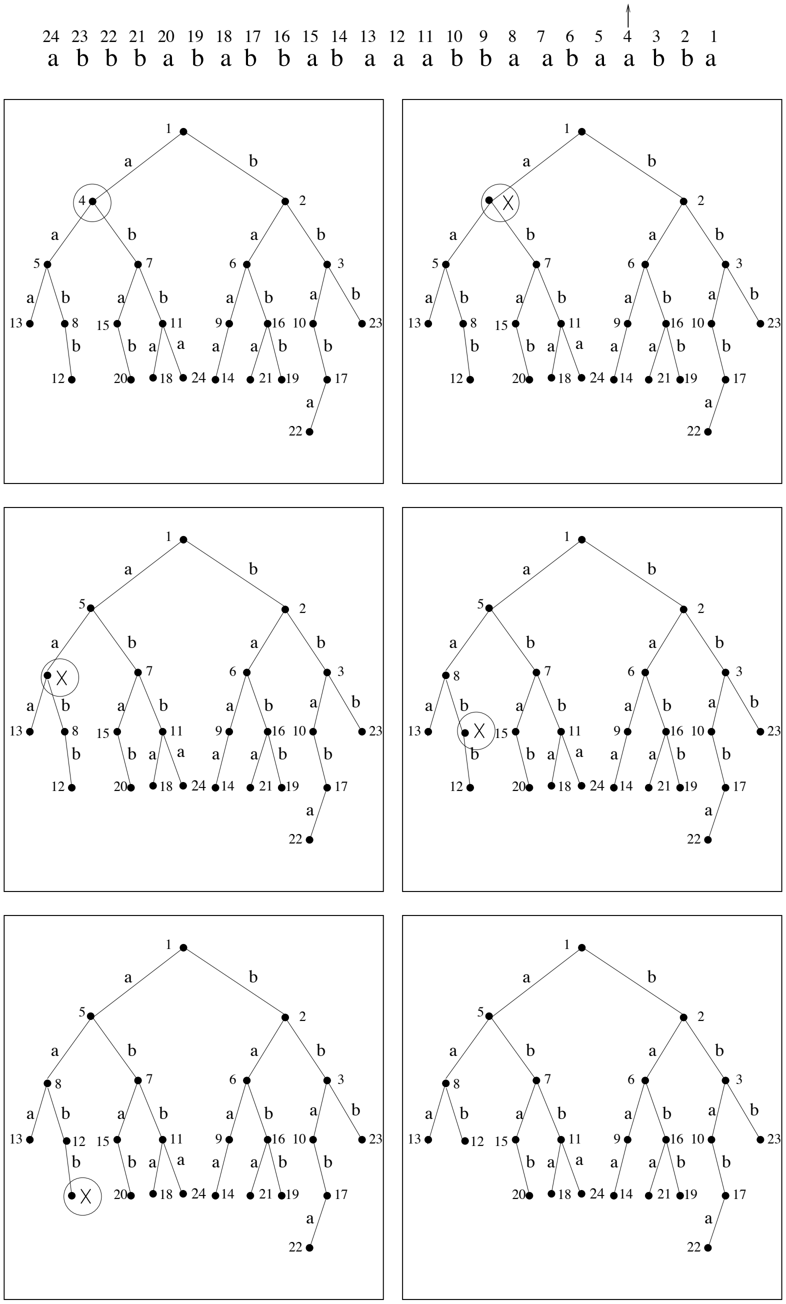


Figure 7.2: Using the `Remove` operation to remove the pointer to position 4 from a position heap. Removing the pointer from its position-heap node leaves the node with an empty position pointer. This is filled by promoting the position pointer of the child whose position in T is smallest (rightmost), in this case the pointer to position 5, to the empty parent. The child is now empty, and it is filled recursively. As a base case, the empty node is a leaf, and it is deleted. The only change to the shape of the tree is the deletion of a leaf.

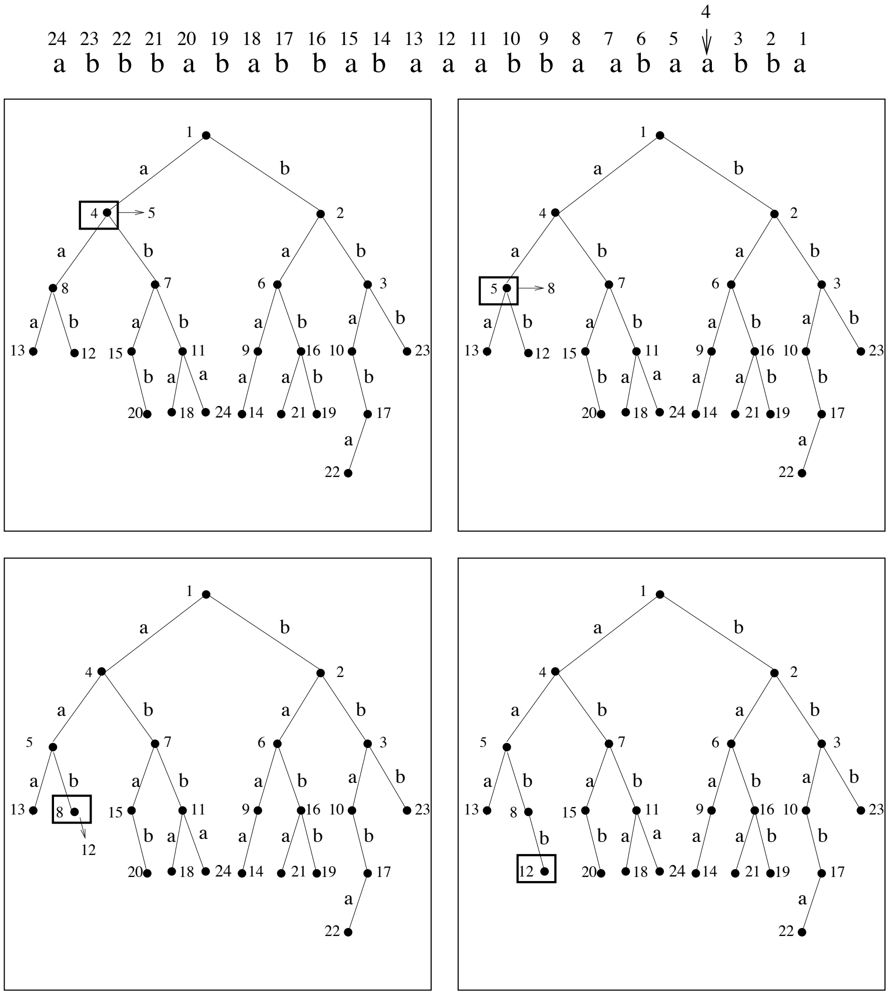


Figure 7.3: The `Add` operation performs the inverse of `Remove`. To add the pointer to position 4 back in, index into the tree on $T_4 = abba$ until a node X is encountered that has a larger position label. In this case, X is the node with a pointer to position 5 in it. This pointer is pushed down to the child reachable on the letter $||X|| + 1$ of T_5. Since $||X|| = 1$, this is the letter a at position 2 of $T_5 = aaba$. This makes it necessary to push down the pointer to position 8 recursively. As a base case, the pointer (to position 12) is pushed down to a new leaf. The only change to the shape of the tree is the addition of a new leaf.

7.3 Use of Splay Trees for Representing Dynamic Texts and Other Lists

When T changes dynamically, we can no longer assume that the characters of T are in an array. For completeness, we specify a straightforward way to represent a dynamic text in a way that supports array-like operations with a small extra cost. This allows us to generalize what we have developed above without defining too many new operations.

In particular, we use what we could call a `dynamic array abstract data type`, which supports the following operations on ordered lists.

- $makelist(v)$: Create a new dynamic array of length 1 containing a single element, v;
- $join(L_1, L_2)$: Join dynamic arrays L_1 and L_2, returning the concatenation L_1L_2;
- $split(L, v)$: Given element v in dynamic array L, split L into two dynamic arrays, L_1 consisting of the part of the array up through v, and L_2 consisting of the part of the array following v;
- $L[i]$ Return the element currently in position i of L;
- $i(L, v)$: Given a pointer to element v of L, return its current position number i in L;

Sleator and Tarjan show how to implement `makelist` in $O(1)$ time, and *join* and *split* in $O(\log n)$ amortized time [57, 58]. The data structure is based on *splay trees*, which are binary trees whose represented list is given by their inorder traversal. To obtain the bounds, every time they traverse a path in the tree, they perform a *splay* on the path. This consists of a set of rotations that alter the shape of the tree without altering the represented list. Their operations traverse the path from v to the root, without affecting the $O(\log n)$ amortized bound.

The textbook [12] shows how to implement the last two operations of the abstract data type in time proportional to the path from the root to v in their chapter, *Augmenting Data Structures*. The idea is to label each node x with $size(x)$, which is the number of nodes in subtree rooted at x. Let l and r be the left and right children of the root. To look up $L[i]$, if $i \leq size(l)$, recursively find $L[i]$ in the left subtree, if $i = size(l) + 1$, then $L[i]$ is at the root, and if $i > size(l) + 1$, recursively find $L[i - size(l) - 1]$ in the right subtree. As always, the path traversed by this operation is splayed. During a rotation, it is easy to update the size labels of the nodes involved in the rotation in $O(1)$ time, using the size labels of their new children after the rotation.

This gives the following:

Lemma 7.3.1 *On the dynamic array abstract data type, the* `makelist` *operation takes* $O(1)$ *time, and each of the other operations takes* $O(\log n)$ *amortized time.*

Corollary 7.3.2 *Given two elements v and w of an instance L of the dynamic array abstract data type, and another instance* L_2 *of the dynamic array abstract data type, the following take* $O(\log n)$ *amortized time:*

- *Determine which of v and w is earlier in L by looking up* $i(L,v)$ *and* $i(L,w)$*;*
- *Remove the segment of L between v and w, yielding two dynamic arrays, what remains of L and a dynamic array consisting of the removed section, by splitting L before v and after w, and joining the two dynamic arrays on either side of this section;*
- *Insert* L_2 *in L starting at position i, by splitting L before position i and concatenating the three dynamic arrays;*

The data structure for the dynamic version of the position heap is identical to that of the static version, except that the dynamic array data structure is used to represent the text, rather than an array of characters, and instead of a position number, a node of the position heap has a pointer to the node of the dynamic array data structure. The position in memory of a splay-tree node never changes, even though the shape of the splay tree that represents it may, so the position pointers from the position heap do not have to be updated during a splay operation on a path. Above, we say that there must be a pointer from each position p of the text to the position-heap node that contains p. This pointer can be stored as the label $N(p)$ on the corresponding splay-tree node in the representation of T.

To obtain amortized bounds, we assume that the text T starts out as the empty string, and evolves through a series of insertions and deletions. The time bound is amortized beginning at this starting point.

7.4 A Time Bound for the Naive Query Algorithm on the Dynamic Position Heap

In this section, we examine the effect of the new implementation on the time bound for the naive query algorithm given by Definition 3.3.1.

- Index into the position heap to find the longest prefix X of P that is a node of $H(T)$. For each ancestor X' of X (including X), look up the pointer p into T stored in X'. This is a pointer to an occurrence of X' in T. Determine whether this occurrence is followed by $P-X'$ by calling $T[i(p)+||X'||]$, $T[i(p)+||X'||+1]$, $\ldots$, $T[i(p)+||X||-1]$ to find the substring of T that follows this instance of X', and comparing this with $P-X'$.

- If $X=P$, also report all positions stored at descendants of X.

Recall from Lemma 3.3.3 that, in the static case, the position heap takes $O(min(m^2, mh(T)) + k)$ time.

Lemma 7.4.1 *The naive query algorithm on the dynamic position heap takes $O(min(m^2, mh(T))\log n + k)$ amortized time, where m is the length of the query string, and k is the number of occurrences of it in T.*

Proof: For each ancestor X' of X, it takes $O(||X||\log n)$ amortized time to traverse the first $||X||$ characters of $T_{p(X)}$, comparing them with X. Since the height of the tree is $O(h(T))$, there are $O(min(m, h(T))$ ancestors of X. Multiplying the number of ancestors by the time spent at each ancestor gives $O(min(m^2, mh(T))\log n)$ time for these steps.

If $X = P$, that is, if P is a node of the position heap, it also takes $O(k')$ time to return the k' pointers in the subtree rooted in its subtree, each of which points to an occurrence of P. There are $O(k)$ of these, so they take $O(k)$ time to return.

7.5 Time Bounds for `Delete` and `Insert`

Lemma 7.5.1 *The time for a call to* `Add` *or* `Remove` *on the dynamic implementation of a trie of height h is $O(h\log n)$, amortized.*

Proof: For `Remove`, we must descend along a recursive path, identifying at each node the child that can be promoted without violating the heap property. This is the child whose pointer into the text T is rightmost in T. Comparing two pointers to see which is rightmost requires $O(\log n)$ amortized time by Corollary 7.3.2, and it takes at most $|\Sigma| - 1$ such comparisons, which is $O(1)$ comparisons, since we have assumed that Σ is $O(1)$. Promoting a pointer p takes $O(1)$ time to move it to the parent, and $O(1)$ time to change the pointer $N(p)$ at position p of the text so that it points to the new node where p resides. The height of the tree is $O(h)$, so the total time is $O(h \log n)$, amortized. It takes $O(h)$ time to update the counts of the number of nodes at each level, and $O(1)$ to update the pointer $N(p)$ to the node that contains p.

For Add, we must find the node where the new position p must be added. This requires indexing in on T_p until we cannot index further, or else a node is found that contains a pointer q that lies to the left of p in T. Indexing requires looking up each successive character of T_p. This takes $O(\log n)$ amortized time per character. At each node X on the indexing path, we must determine whether the position q at the node lies to the left of p, which takes $O(\log n)$ amortized time by Corollary 7.3.2. Since the height of the tree is $O(h)$, finding the node where p must be added takes $O(h \log n)$ amortized time.

We must now analyze the time for the push-down operation. At each node X, we know $||X||$ because we found it by starting at the root, and each child is one character longer than its parent. We look up the pointer q in X in $O(1)$ time, find the character $||X|| + 1$ in T_q in $O(\log n)$ amortized time by using the $L[]$ operator on the dynamic array implementation of T, and find the child of X reachable on that character in $O(1)$ time. Moving a pointer p from parent to child and updating $N(p)$ to reflect the move takes $O(1)$ time. The total time for push-down is $O(\log n)$, amortized, at each position on the push-down path, which has length $O(h)$.

Lemma 7.5.2 Delete *on a block of b characters takes $O((h(T) + b)h(T) \log n)$ amortized time, where T is the text before it is edited.*

Proof: We perform a Remove and an Add on $h - 1$ positions to the left of the edited block, where $h = O(h(T))$ is the initial height of the tree. Since Remove does not increase the height of the tree, and Add increases it by at most 1, the height can grow by at most $h - 1$

during these operations, so it remains $O(h(T))$ throughout these steps. By Lemma 7.5.1, these $h-1$ operations take $O(h(T)^2 \log n)$ amortized time.

To remove the defunct positions in `Delete` takes b calls to `Remove`. Since a call to `Remove` never increases the height of the tree, the tree has height $O(h(T))$ throughout this step. The b calls to `Remove` on these positions takes $O(bh(T) \log n)$ amortized time.

Lemma 7.5.3 `Insert` *on a block of b characters takes* $O((h(T')+b)h(T') \log n)$ *amortized time, where* T' *is the text after it is edited.*

Proof: Suppose a block of b characters of a text T' is removed, yielding text T. (We have switched the roles of T and T'.) A call to `Delete` on text T', resulting in the position heap of T, takes $O((h(T')+b)h(T') \log n)$ time by Lemma 7.5.2. This operation can be inverted by reinserting the block of b deleted positions and calling `Insert`, yielding the position heap for T. This replaces the b calls to `Remove` with b calls to `Add`, which may be performed in reverse order, thereby stepping through the same sequence of tries in reverse order. Each call to `Remove` in `Delete` takes the same asymptotic bound as the inverse call to `Add`, so these operations take $O((h(T')+b)h(T') \log n)$, just as the call to `Delete` does.

As shown in the proof of the time bound for `Delete` the height h of the tree after the calls to `Remove` and `Add` on the affected positions is $O(h(T'))$, so the final $h-1$ calls to `Remove` and `Add` take $O(h(T')^2 \log n)$ amortized time.

7.6 A Dynamic Implementation of the Augmented Position Heap

Recall that the additional features of the augmented position heap are a labeling of the nodes with maximal-reach pointers and discovery and finishing times.

7.6.1 Discovery and Finishing Times

In the static case, the discovery and finishing times are integer labels from 1 to $2n$, where no two discovery/finishing labels are equal. The only purpose of these is so that we can compare two discovery times or two finishing times to find which is earlier; this is used in the test of whether one node is an ancestor of another. To support this operation on a dynamic structure, we consider discovery and finishing of nodes to be **events**, and create an **event list** using the dynamic array abstract data type described above. Instead of being labeled with integers to represent the discovery and finishing time, a node is labeled with two pointers into this list, one to its discovery event and one to its finishing event. This provides all of the functionality of the discovery- and finishing-time labels of the (static) augmented position heap, but at a cost of $\Theta(\log n)$ amortized time per comparison, rather than $O(1)$.

We must update the event list when the topology of the tree changes. Recall that the only effect of `Remove` or `Add` on the topology of the trie is the removal or addition of a leaf. If the leaf is removed, we remove its discovery and finishing events from the event list in $O(\log n)$ amortized time, using the leaf's pointers to these two events. If a new leaf Z is added to the tree, we must find where its discovery and finishing events must be inserted to the event list. Since Z is a leaf, its discovery and finishing events must be consecutive in the event list. If Z is leftmost among its siblings, they are added immediately following the discovery event of its parent. Similarly, if the new leaf is rightmost among its siblings, they are added immediately preceding the finishing event of its parent. Otherwise, they are added immediately following the finishing event of the sibling to the left. The correctness of this placement follows from the order in which nodes are visited in a depth-first search. When Z is inserted, `Add` has found the path from the root to Z, so the parent is known. It therefore takes $O(1)$ time to find the parent or left sibling of Z, and then $O(\log n)$ amortized time to insert the new events.

7.7 `Remove` and `Add` on the Augmented Dynamic Position Heap

Let us say that a trie is an **augmented trie** for a text if it satisfies properties1 and 2 of Lemma 7.0.3 and its discovery- and finishing-event list is correct. It might not satisfy properties 3 and 4, and some of its maximal-reach pointers might not be correct.

If X is a node of an augmented trie, let us denote its maximal-reach pointer by $m(X)$. The position pointer $p(X)$ of X is the position pointer **corresponding** to $m(X)$ and *vice versa*. Recall that $m(X)$ is supposed to point to the node of the trie that is the maximal prefix of $T_{p(X)}$ that is a node of the trie. Let us say that $m(X)$ is **correct** if it satisfies this property.

A corresponding position pointer and maximal-reach pointer must reside at the same node, so when a position pointer is promoted (moved from child to parent), or pushed down (moved from parent to child), so must the corresponding maximal-reach pointer. When a position pointer is removed from the trie, so must the corresponding maximal-reach pointer. The asymptotic cost of performing these operations on the maximal-reach pointer is subsumed by the cost of performing them on the position pointer. When a call to `Add` inserts a new position p to a node X, it must insert the maximal-reach pointer to X. The node that it must point to is found by indexing as far as possible into the trie on $T_{p(X)}$; it must point to the last node on this path. This takes $O(h\log n)$ amortized time, where h is the current height of the trie. This is subsumed by the $O(h\log n)$ amortized cost of `Add`, as described above, for the un-augmented dynamic position heap.

An new issue arises in managing the maximal-reach pointers during a call to `Remove` that does not arise in managing position pointers. A call to `Remove` changes the topology of the trie by removing exactly one leaf Z. Let P be the parent of Z. Let q and m be corresponding position and maximal-reach pointers, respectively. If m points to Z, then Z ceases to be the maximal prefix of T_q that is a node of the trie; the next smaller prefix, P, is now this node. Since Z is a descendant of the node occupied by m, this issue only affects maximal-reach pointers at ancestors of Z. Since `Remove` finds the path from the root to Z, we add a step that checks each node Y on this path to see if $m(Y)$ points to Z, and, if so, changes it to point to P. This requires traversing $O(h)$ nodes, and at each, performing an $O(1)$ operation. This $O(h)$ cost is subsumed by the cost of `Remove`, as described above, for the unaugmented heap.

The issue also affects a call to `Add`, which changes the topology of the trie by adding Z as a child of P. Let c be the letter that labels the edge from P to Z. If $m(X)$ initially points to P, then P is a prefix of $T_{p(X)}$. If $Z = Pc$ is also a prefix of $T_{p(X)}$, then $m(X)$ must point to Z, not P, in order to be correct in the new trie. Since P is a prefix of $T_{p(X)}$, this only happens at ancestors of P. We add a step to `Add` that traverses each node X on the path from the root to P, determines whether $m(X)$ points to P, and, if so, whether the character $||P||$ positions to the right of $p(X)$ is c. If it satisfies all of these conditions, then $m(X)$ is changed to point to Z. Looking up the character $||P||$ positions to the right of p takes $O(\log n)$ amortized time using the dynamic array implementation of the text, for a total of $O(h \log n)$ amortized time, which is subsumed by the cost of `Add` given above. Finally, $m(Z)$ is set to point to Z.

Let us call the operations of `Add` and `Remove` that carry these additional steps `Add2` and `Remove2`. Since the cost of all additional operations performed in `Add2` and `Remove2` are subsumed by the cost of operations in `Add` and `Remove`, we get the following:

Lemma 7.7.1 *The time for a call to* Add2 *or* Remove2 *on a trie of height h is $O(h \log n)$, amortized.*

The additional operations in Add2 and Delete2 allow us to add the following preconditions and postconditions to them, in addition to the ones that apply to Add and Remove (Definition 7.1.1.)

1. For Remove2, the additional preconditions are that the event list is correct for input trie but that some maximal-reach pointers may be incorrect. The additional postconditions are that the event list is correct for the modified trie, that the maximal-reach pointer corresponding to the removed position has also been removed, *and that no other maximal-reach pointers have been made incorrect by the operation.*

2. The additional preconditions for Add2 are also that the event list is correct for the input trie, and some maximal-reach pointers may be incorrect. The additional postconditions are that the event list is correct for the modified trie, that the maximal-reach pointer corresponding to the inserted position is correct, *and that no other maximal-reach pointers have been made incorrect by the operation.*

7.8 Delete and Insert on the Augmented Position Heap

To modify Delete and Insert for the augmented position heap, we make them call Remove2 and Add2 in place of Remove and Add. Like the position pointers, the maximal-reach pointers corresponding to the $h-1$ positions to the left of the edited position can become incorrect, even though they correspond to positions that were not edited. The reason is exactly the same as it is for the position pointers. The maximal-reach pointer associated with a pointer q is supposed to point to the longest node Y that is a prefix of T_q.

This node can be a string of length as long as h. Since q is within $h-1$ positions of where T is edited, Y may no longer be a prefix of T_q after the edit.

However, by the preconditions and postconditions of `Remove2` and `Add2`, calling `Remove2` followed by `Add2` on the $h-1$ positions preceding the edited position suffices to repair this problem at each of these positions without creating any new problems with position pointers or maximal-reach pointers at other nodes.

Since the asymptotic time bounds of `Remove2` and `Add2` are the same as those of `Remove` and `Add`, the analysis of the running time of `Delete` and `Insert` is unchanged when it operates on the augmented position heap. This gives the following, by Lemmas 7.5.2 and 7.5.3.

Lemma 7.8.1 `Delete` *on a block of b characters takes $O((h(T)+b)h(T)\log n)$ amortized time on the augmented position heap, where T is the text before it is edited.*

Lemma 7.8.2 `Insert` *on a block of b characters takes $O((h(T')+b)h(T')\log n)$ amortized time, where T' is the text after it is edited.*

7.9 Time Bound for Queries on the Dynamic Augmented Position Heap

Lemma 7.9.1 *Using the augmented dynamic position heap, it takes $O(m\log n+k)$ amortized time to find the k occurrences of a string P in T.*

Proof: This is obtained by reexamining the proof of Lemma 4.1.5 in light of the fact that the use of dynamic arrays causes some operations that previously took $O(1)$ time to take $O(\log n)$ amortized time. These are determining whether a node Y is a descendant of a node X, which takes $O(\log n)$ amortized time: we look up whether X's discovery event precedes Y's and X's finishing even follows Y's in the dynamic-array implementation of the event list. This is accomplished with the $i()$ operator on dynamic arrays. This contrasts with the $O(1)$ time these comparisons take when the discovery and finishing times are implemented with integers. We must also determine whether a known occurrence of a string X_i is followed by a letter b. The position of the occurrence and the length of X_i gives the position of the next letter. Looking up this letter is accomplished with the $L[]$ operator on dynamic arrays, which takes $O(\log n)$ amortized time instead of the $O(1)$ time it takes on an array implementation of the text. There are $O(m)$ of these operations, for an $O(m \log n)$ amortized bound for them. Any remaining occurrences of the query string are reported by traversing the subtree rooted at the query string, which takes time proportional to the number of positions in this subtree.

Chapter 8

Conclusion

In the deluge of information, the world has been changed consistently by the dramatic information flooding through our society. To filter out useful and valuable information, string searching or matching is one of methods used. In this research, we have studied pattern matching algorithms and data structures. Many data structures represent substrings. Of these, we have briefly presented the suffix tree, the suffix array, the DAWG and the position heap. Each of these data structures can achieve a construction time in $O(n)$ time and provide a reasonable time bound for the search. The implementation of the suffix tree is more complex than the implementation of the suffix array although the search time of the suffix tree is faster.

In this dissertation, we describe the position heap, augmented position heap, the compact position heap and the augmented position heap and its modification. The strong point of the position heap is ease of use. Most people who have fundamental knowledge of data structure can easily understand the naive construction algorithm. The improved data structure of position heap is the augmented position heap. It reduces the search time from $O(m^2+k)$ to $O(m+k)$. Thus, it provides the same construction and search time bound as the suffix array or DAWG and is simpler to implement.

The compact version of position heap has been shown. It can compete with suffix array's space requirement. The position heap and the augmented position heap with tree data structure require more space than suffix array because of the hash table, the maximal-reach pointers and postorder. The proposed compacted representation of the position heap now competes against the suffix array. The position heap only uses two integers for each index of an array and the augmented position heap is also compressed with four integers for each index of an array. The additional two integers of the augmented position heap give an advantage for the search time.

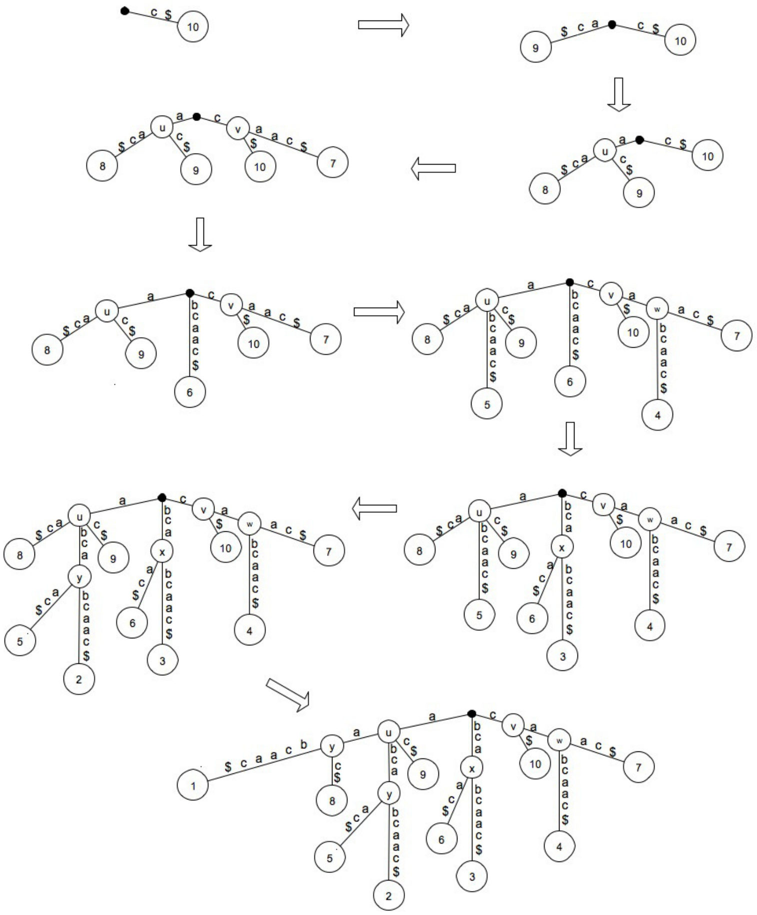
The most advanced of the position heap or the augmented position heap are dynamic. When a text is edited, we can reflect modification to the data structure that costs less than reconstruction. Even though there are many great and popular algorithms for string searching or matching data structures, most of them do not support the modification of characters or substrings. The data structure must be rebuilt even for small modifications of text. This needs at least another $O(n)$ time. In this research, we have proposed the idea of the simple and dynamic data structure, the position heap and the augmented position heap, which supports inserting and deleting a single character in $O(h(T)^2 \log n)$. The maintenance of the position heap only takes $O(h(T)^2)$. We also need $O(\log n)$ amortized time for a dynamic array abstract data type, which is proposed by Sleator and Tarjan [57, 58]. This time bound provides a great advantage over rebuilding the data structure. Also demonstrated was the dynamic texting for a block of string such as substitution of substring of a text. We can easily perform the substitution operation when we know the insertion and deletion process, since we process deletion operation for substituted substrings and insert these substrings into the position heap. The substitution operation spends $O((h(T+b)h(T)\log n)$ for b lengths of substrings.

Future works include developing an approximate pattern matching algorithm for the position heap or the augmented position heap. The area of bioinformatics requires a more approximate pattern matching algorithm than pattern matching exactly.

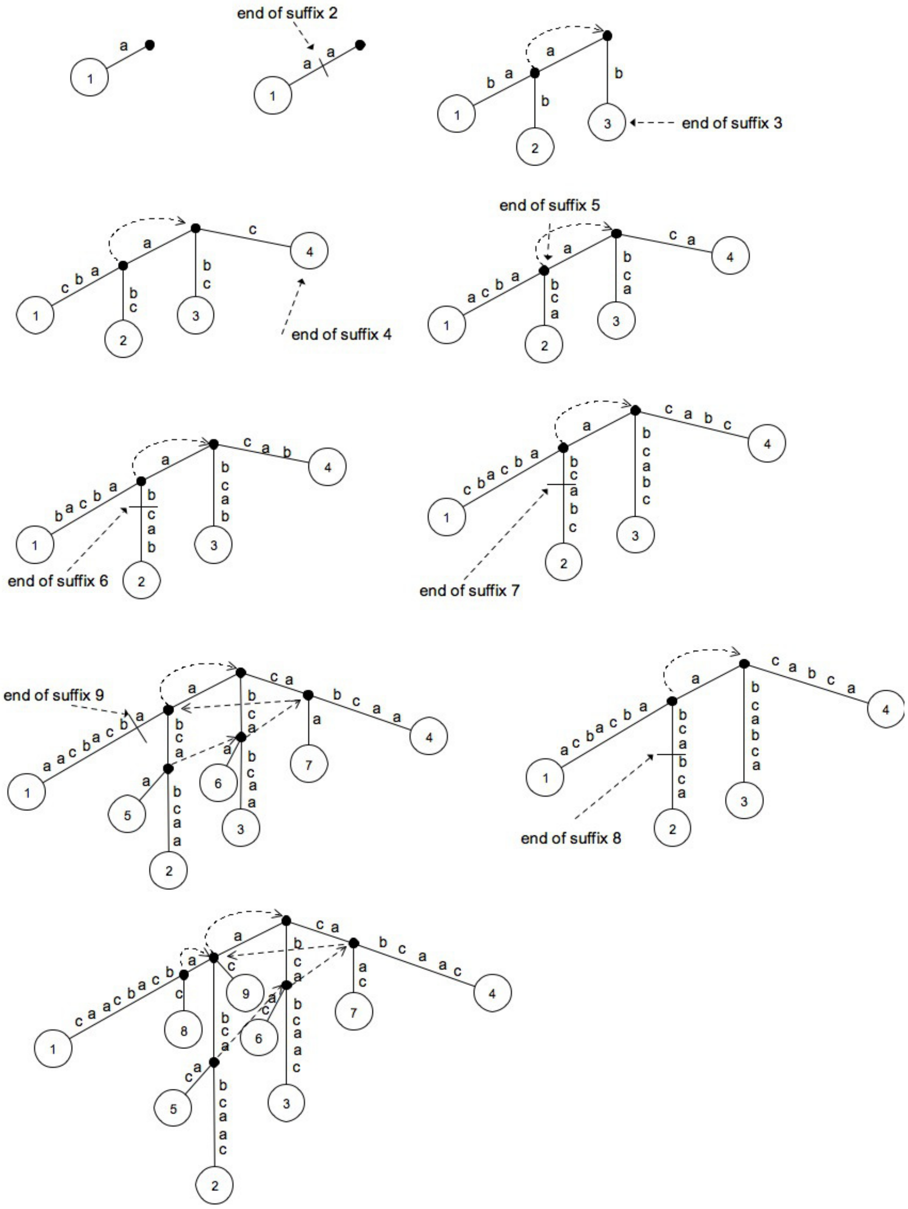
Appendix A

Suffix Tree

Weiner's suffix tree with "aabcabcaac"



Ukkonen's suffix tree with "aabcabcaac"



REFERENCES

[1] Stephen Altschul and Bruce Erickson. Optimal sequence alignment using affine gap costs. *Bulletin of Mathematical Biology*, 48(5):603–616, September 1986.

[2] Amihood Amir, Oren Kapah, and Dekel Tsur. Faster two-dimensional pattern matching with rotations. *Theor. Comput. Sci.*, 368:196–204, December 2006.

[3] Ricardo Baeza-Yates and Gonzalo Navarro. Multiple approximate string matching. In Frank Dehne, Andrew Rau-Chaplin, Jörg-Rüdiger Sack, and Roberto Tamassia, editors, *Algorithms and Data Structures*, volume 1272 of *Lecture Notes in Computer Science*, pages 174–184. Springer Berlin / Heidelberg, 1997.

[4] Ricardo Baeza-Yates and Mireille Régnier. Fast two-dimensional pattern matching. *Inf. Process. Lett.*, 45:51–57, January 1993.

[5] Theodore P. Baker. A technique for extending rapid exact-match string matching to arrays of more than one dimension. *SIAM J. Comput.*, 7(4):533–541, 1978.

[6] Pawe Baturo and Wojciech Rytter. Compressed string-matching in standard sturmian words. *Theor. Comput. Sci.*, 410:2804–2810, August 2009.

[7] L. Bergroth, H. Hakonen, and T. Raita. A survey of longest common subsequence algorithms. In *Proceedings of the Seventh International Symposium on String Processing Information Retrieval (SPIRE'00)*, pages 39–, Washington, DC, USA, 2000. IEEE Computer Society.

[8] Anselm Blumer, J. Blumer, David Haussler, Ross M. McConnell, and Andrzej Ehrenfeucht. Complete inverted files for efficient text retrieval and analysis. *J. ACM*, 34(3):578–595, 1987.

[9] M. Burrows and D. J. Wheeler. A block-sorting lossless data compression algorithm. Technical Report 124, 1994.

[10] Charles L. A. Clarke, Maheedhar Kolla, Gordon V. Cormack, Olga Vechtomova, Azin Ashkan, Stefan Buttcher, and Ian MacKinnon. Novelty and diversity in information retrieval evaluation. In *SIGIR'08*, pages 659–666. SIGIR'08, 2008.

[11] E. Coffman and J. Eve. File structures using hashing functions. *Communications of the ACM*, 13(7):427–432, July 1970.

[12] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press and McGraw-Hill, 2nd edition, 2001.

[13] Maxime Crochemore and Renaud Vérin. Direct construction of compact directed acyclic word graphs. In *CPM*, pages 116–129, 1997.

[14] Liuling Dai. An aggressive algorithm for multiple string matching. *Inf. Process. Lett.*, 109:553–559, May 2009.

[15] Peter J. Denning. *Before Memory was Virtual*. IEEE Press, 1997.

[16] A. Ehrenfeucht, Ross M. Mcconnell, Nissa Osheim, and Sung-Whan Woo. Position heaps: A simple and dynamic text indexing data structure. *Journal of Discrete Algorithms*, in press.

[17] Andrzej Ehrenfeucht, Ross M. Mcconnell, and Sung-Whan Woo. Contracted suffix trees: A simple and dynamic text indexing data structure. In *CPM '09: Proceedings of the 20th Annual Symposium on Combinatorial Pattern Matching*, pages 41–53, Berlin, Heidelberg, 2009. Springer-Verlag.

[18] M. Farach. Optimal suffix tree construction with large alphabets. *Proceedings of the 38th Annual Symposium on the Foundations of Computer Science*, pages 137–143, 1997.

[19] Martin Farach and Mikkel Thorup. String matching in lempel-ziv compressed strings. In *Proceedings of the twenty-seventh annual ACM symposium on Theory of computing*, STOC '95, pages 703–712, New York, NY, USA, 1995. ACM.

[20] Simone Faro and Thierry Lecroq. The exact string matching problem: a comprehensive experimental evaluation. *CoRR*, abs/1012.2547, 2010.

[21] P. Ferragina and G. Manzini. Opportunistic data structures with applications. In *Proceedings of the 41st Annual Symposium on Foundations of Computer Science*, pages 390–, Washington, DC, USA, 2000. IEEE Computer Society.

[22] Paolo Ferragina and Roberto Grossi. Fast incremental text editing. In *SODA*, pages 531–540, 1995.

[23] Paolo Ferragina and Roberto Grossi. The string b-tree: A new data structure for string search in external memory and its applications. *Journal of the ACM*, 46:236–280, 1998.

[24] Paolo Ferragina, Roberto Grossi, and Manuela Montangero. On updating suffix tree labels. *Theor. Comput. Sci.*, 201(1-2):249–262, 1998.

[25] Kimmo Fredriksson and Maxim Mozgovoy. Efficient parameterized string matching. *Inf. Process. Lett.*, 100:91–96, November 2006.

[26] Kimmo Fredriksson and Maxim Mozgovoy. Efficient parameterized string matching. *Inf. Process. Lett.*, 100:91–96, November 2006.

[27] Z. Galil and K. Park. Truly alphabet-independent two-dimensional pattern matching. *Foundations of Computer Science, Annual IEEE Symposium on*, 0:247–256, 1992.

[28] Gaston H. Gonnet. Some string matching problems from bioinformatics which still need better solutions. *Journal of Discrete Algorithms*, 2:3–15, March 2004.

[29] O. Gotoh. An improved algorithm for matching biological sequences. *Journal of molecular biology*, 162(3):705–708, December 1982.

[30] Ming Gu, Martin Farach, Richard Beigel, and Yale Dimacs Yale. An efficient algorithm for dynamic text indexing (extended abstract), 1993.

[31] D. S. Hirschberg. A linear space algorithm for computing maximal common subsequences. *Commun. ACM*, 18:341–343, June 1975.

[32] Christian Hundt and Maciej Liśkiewicz. Two-dimensional pattern matching with combined scaling and rotation. In *Proceedings of the 19th annual symposium on Combinatorial Pattern Matching*, CPM '08, pages 5–17, Berlin, Heidelberg, 2008. Springer-Verlag.

[33] Trinh N.D. Huynh, Wing-Kai Hon, Tak-Wah Lam, and Wing-Kin Sung. Approximate string matching using compressed suffix arrays. *Theoretical Computer Science*, 352(1-3):240 – 249, 2006.

[34] Costas S. Iliopoulos, Christos Makris, Yannis Panagis, Katerina Perdikuri, Evangelos Theodoridis, and Athanasios Tsakalidis. The weighted suffix tree: An efficient data structure for handling molecular weighted sequences and its applications. *Fundam. Inf.*, 71:259–277, February 2006.

[35] S. Inenaga, H. Hoshino, A. Shinohara, and M. Takeda. On-line construction of compact directed acyclic word graphs. *Proceedings of the 12th Annual Symposium on Combinatorial Pattern Matching*, 2089:169–180, Jan 2001.

[36] S. Inenaga, M. Takeda, A. Shinohara, and H. Hoshino. The minimum dawg for all suffixes of a string and its applications. *Combinatorial Pattern Matching: 13th Annual Symposium*, pages 153–167, 2002.

[37] Petteri Jokinen and Esko Ukkonen. Two algorithms for approxmate string matching in static texts. In Andrzej Tarlecki, editor, *Mathematical Foundations of Computer Science 1991*, volume 520 of *Lecture Notes in Computer Science*, pages 240–248. Springer Berlin / Heidelberg, 1991.

[38] J. Karkkainen and P. Sanders. Simple linear work suffix array construction. *Proc. 13th International Conference on Automata, Languages and Programming*, 2003.

[39] J Karkkainen, P Sanders, and S Burkhardt. Linear work suffix array construction. *Journal of the ACM*, 53(6):918–936, 2006.

[40] Donald E. Knuth, James H. Morris, and Vaughan R. Pratt. Fast Pattern Matching in Strings. *SIAM Journal on Computing*, 6(2):323–350, March 1977.

[41] A. Lempel and J. Ziv. Compression of two-dimensional data. *IEEE Trans.Inform.Theory*, 32(1), January 1986.

[42] David Maier. The complexity of some problems on subsequences and supersequences. *J. ACM*, 25:322–336, April 1978.

[43] U. Manber and G. Myers. Suffix array: A new method for on-line string searches. *SIAM J. Compt.*, pages 319–327, 1990.

[44] R. McConnell and A. Ehrenfeucht. String searching. *Book chapter*, Oct 2005.

[45] E. M. McCreight. A space-economical suffix tree construction algorithm. *Journal of the ACM*, 23(2):262–272, 1976.

[46] Webb Miller and Eugene Myers. Sequence comparison with concave weighting functions. *Bulletin of Mathematical Biology*, 50:97–120, 1988.

[47] J. H. Morris and V. R. Pratt. A linear pattern-matching algorithm. *Technical Report 40, University of California, Berkeley*, 1970.

[48] Carey Nachenberg. Computer virus-antivirus coevolution. *Commun. ACM*, 40:46–51, January 1997.

[49] G. Niklas Norén, Andrew Bate, Johan Hopstadius, Kristina Star, and I. Ralph Edwards. Temporal pattern discovery for trends and transient effects: its application to patient records. In *Proceeding of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining*, KDD '08, pages 963–971, New York, NY, USA, 2008. ACM.

[50] W. R. Pearson and D. J. Lipman. Improved tools for biological sequence comparison. *Proceedings of the National Academy of Sciences of the United States of America*, 85(8):2444–2448, April 1988.

[51] Yi Peng, Gang Kou, Yong Shi, and Zhengxin Chen. A descriptive framework for the field of data mining and knowledge discovery. *International Journal of Information Technology and Decision Making*, 7(4):639–682, 2008.

[52] Rajesh Prasad and Suneeta Agarwal. Parameterized string matching: an application to software maintenance. *SIGSOFT Softw. Eng. Notes*, 35:1–5, May 2010.

[53] Luís M. S. Russo, Gonzalo Navarro, Arlindo L. Oliveira, and Pedro Morales. Approximate string matching with compressed indexes. *Algorithms*, 2(3):1105–1136, 2009.

[54] M. Salson, T. Lecroq, M. Léonard, and L. Mouchard. A four-stage algorithm for updating a burrows-wheeler transform. *Theor. Comput. Sci.*, 410:4350–4359, October 2009.

[55] M. Salson, T. Lecroq, M. Léonard, and L. Mouchard. Dynamic extended suffix arrays. *J. of Discrete Algorithms*, 8:241–257, June 2010.

[56] Mikaël Salson, Thierry Lecroq, Martine Léonard, and Laurent Mouchard. Dynamic Burrows-Wheeler transform. In Jan Holub and Jan Žd'árek, editors, *Proceedings of the Prague Stringology Conference 2008*, pages 13–25, Czech Technical University in Prague, Czech Republic, 2008.

[57] Daniel D. Sleator and Robert Endre Tarjan. A data structure for dynamic trees. In *STOC '81: Proceedings of the thirteenth annual ACM symposium on Theory of computing*, pages 114–122, New York, NY, USA, 1981. ACM.

[58] R. E. Tarjan. *Data structures and network algorithms*. Society for Industrial and Applied Math., Philadelphia, 1983.

[59] E. Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995.

[60] Esko Ukkonen. Approximate string-matching over suffix trees. In Alberto Apostolico, Maxime Crochemore, Zvi Galil, and Udi Manber, editors, *Combinatorial Pattern Matching*, volume 684 of *Lecture Notes in Computer Science*, pages 228–242. Springer Berlin / Heidelberg, 1993.

[61] P. Weiner. Linear pattern matching algorithms. *IEEE 14th Annual Symposium on Switching and Automata Theory*, pages 1–11, 1973.