# UNIT - 3

# NEW INDICES FOR TEXT

## INTRODUCTION

Text searching methods may be classified as lexicographical indices (indices that are sorted), clustering techniques, and indices based on hashing. Two new lexicographical indices for text, called PAT trees and PAT arrays. Our aim is to build an index for the text of size similar to or smaller than the text.

The traditional model of text used in information retrieval is that of a set of documents. Each document is assigned a list of keywords (attributes), with optional relevance weights associated to each keyword (Example: library applications). For more general applications, it has some problems, namely:

- A basic structure is assumed (documents and words). This may be reasonable for many applications, but not for others.
- Keywords must be extracted from the text (this is called "indexing"). This task is not trivial and error prone, whether it is done by a person, or automatically by a computer. ➢ Queries are restricted to keywords.

For some indices, instead of indexing a set of keywords, all words except for those deemed to be too common (called stopwords) are indexed. We prefer a different model. We see the text as one long string. Each position in the text corresponds to a semi-infinite string (sistring), the string that starts at that position and extends arbitrarily far to the right, or to the end of the text.

## PAT TREE STRUCTURE

The PAT tree is a data structure that allows very efficient searching with preprocessing. This section describes the PAT data structure, how to do some text searches and algorithms to build two of its possible implementations.

### Semi-infinite Strings

A semi-infinite string is a subsequence of characters from this array, taken from a given starting point but going on as necessary to the right. In case the semi-infinite string (sistring) is used beyond the end of the actual text, special null characters will be considered to be added at its end, these characters being different than any other in the text. The name semi-infinite is taken from the analogy with geometry where we have semi-infinite lines, lines with one origin, but infinite in one direction. Sistrings are uniquely identified by the position where they start, and for a given, fixed text, this is simply given by an integer.

**Example:**

**Text:** Once upon a time, in a far away land . . .

**sistring 1:** Once upon a time . . .

**sistring 2:** nce upon a time . . .

**sistring 8:** on a time, in a . . .

**sistring 11:** a time, in a far . . .

**sistring 22**: a far away land . . .

Sistrings can be defined formally as an abstract data type and as such present a very useful and important model of text. The most important operation on sistrings is the lexicographical comparison of sistrings and will be the only one defined. This comparison is the one resulting from comparing two sistrings' contents (not their positions). Note that unless we are comparing a sistring to it, the comparison of two sistrings cannot yield equal. (If the sistrings are not the same, sooner or later, by inspecting enough characters, we will have to find a character where they differ, even if we have to start comparing the fictitious null characters at the end of the text).

For example, the above sistrings will compare as follows:

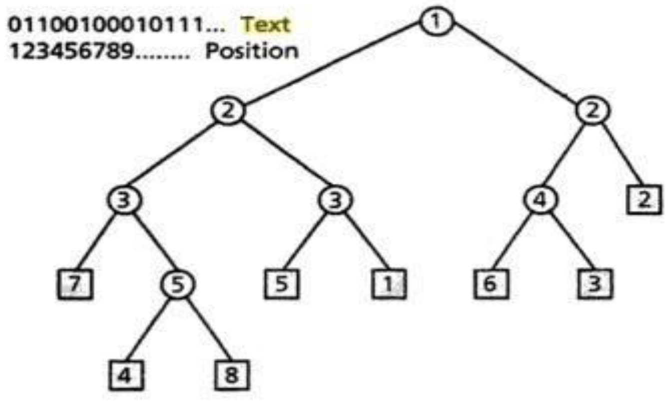
22 < 11 < 2 < 8 < 1

Of the first 22 sistrings (using ASCII ordering) the lowest sistring is "a far away. . ." and the highest is "upon a time. . . ."

**PAT Tree**

A PAT Tree is a Patricia Tree constructed over all the possible sistrings of a text. A Patricia tree is a digital tree where the individual bits of the keys are used to decide on the branching. A zero bit will cause a branch to the left sub tree, a one bit will cause a branch to the right sub tree. Hence Patricia trees are binary digital trees.

Patricia trees store key values at external nodes; the internal nodes have no key information, just the skip counter and the pointers to the sub trees. The external nodes in a PAT tree are sistrings, that is, integer displacements. For a text of size n, there are n external nodes in the PAT tree and n - 1 internal node. This makes the tree O(n) in size, with a relatively small asymptotic constant.



**PAT tree when the sistrings 1 through 8 have been inserted**

The below figure shows an example of a PAT tree over a sequence of bits (normally it would be over a sequence of characters), just for the purpose of making the example easier to understand. In this example, we show the Patricia tree for the text "01100100010111..." after the first 8 sistrings have been inserted.

External nodes are indicated by squares, and they contain a reference to a sistring, and internal nodes are indicated by a circle and contain a displacement.

To each external node for the query 00101 we first inspect bit 1 (it is a zero, we go left) then bit 2 (it is zero, we go left), then bit 3 (it is a one, we go right), and then bit 5 (it is a one, we go right).

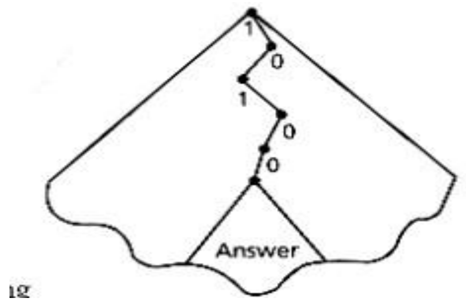
We may skip the inspection of some bits (in this case bit 4). If there is no coincidence at any point then key is not in the tree.

**ALGORITHMS ON THE PAT TREES**

Some of the algorithms for text searching when we have a PAT tree of our text are given as below.

**Prefix Searching**

Notice that every subtree of the PAT tree has all the sistrings with a given prefix, by construction. Then prefix searching in a PAT tree consists of searching the prefix in the tree up to the point where we exhaust the prefix or up to the point where we reach an external node. At this point we need to verify whether we could have skipped bits. This is done with a single comparison of any of the sistrings in the subtree (considering an external node as a subtree of size one). If this comparison is successful, then all the sistrings in the subtree (which share the common prefix) are the answer; otherwise there are no sistrings in the answer.



**Prefix searching**

**Proximity Searching**

We define a proximity search as finding all places where a string S1 is at most a fixed (given by the user) number of characters away from another string S2. The simplest algorithm for this type of query is to search for sl and s2. Then, sort by position the smaller of the two answers. Finally, we traverse the unsorted answer set, searching every position in the sorted set, and checking if the distance between positions (and order, if we always want s1 before s2) satisfies the proximity condition.

**Range Searching**

Searching for all the strings within a certain range of values (lexicographical range) can be done equally efficiently. More precisely, range searching is defined as searching for all strings that lexicographically compare between two given strings. For example, the range “abc” . …. “acc” will contain strings like "abracadabra," "acacia," "aboriginal," but not "abacus" or "acrimonious." To do range searching on a PAT tree, we search each end of the defining intervals and then collect all the subtrees between (and including) them.

**Longest Repetition Searching**

The longest repetition of a text is defined as the match between two different positions of a text where this match is the longest (the most number of characters) in the entire text. For a given text, the longest repetition will be given by the tallest internal node in the PAT tree, that is, the tallest internal node gives a pair of sistrings that match for the greatest number of characters.

**Most Significant or Most Frequent Searching**

This type of search has great practical interest, but is slightly difficult to describe. By “most significant” or “most frequent” we mean the most frequently occurring strings within the text database. For example, finding the “most frequent” trigram is finding a sequence of three letters that appears most often within our text.

**Regular Expression Searching**

The main steps of the algorithm are:

- Convert the regular expression passed as a query into a minimized deterministic finite automation (DFA),
- Next eliminate outgoing transitions from final states (see justification in step (e). This may induce further minimization.
- Convert character DFAs into binary DFAs using any suitable binary encoding of the input alphabet; each state will then have at most two outgoing transitions, one labeled 0 and one labeled 1.

**EXAMPLE**

regex **ab* which means:**

- First character must be a
- Followed by zero or more b's

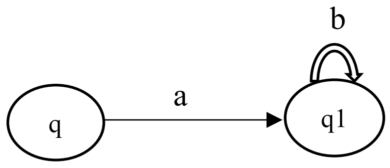
**So valid strings are:**

a, ab, abb, abbb, ...

**DFA Construction**

**States:**

- q0 = start state
- q1 = final (accepting) state



**Encoding rule**

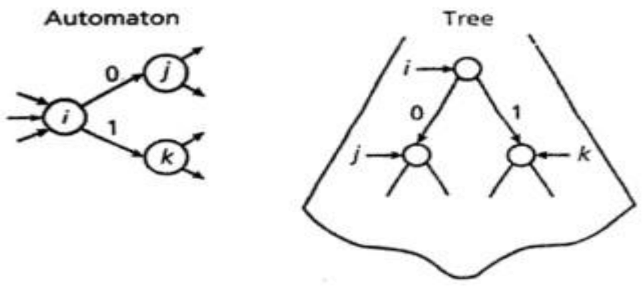
| Character | Binary |
|---|---|
| a | 0 |
| b | 1 |

**Eg: check the string abb is acceptable or not**

So an input string like "abb" becomes binary "011"

| Input (text) | Binary Encoding | DFA Path | Accepted | Reason |
|---|---|---|---|---|
| a | 0 | q0 → q1 | YES | ends in final state |
| ab | 01 | q0 → q1 → q1 | YES | stays in q1 (loop) |
| abb | 011 | q0 → q1 → q1 → q1 | YES | still in q1 |

**The accepted strings are: a, ab, abb**



**Simulating the automation on a binary digital tree**

**BUILDING PAT TREES AS PATRICA TREES**

The implementation of PAT trees using conventional Patricia trees is the obvious one, except for some implementation details which cannot be overlooked as they would increase the size of the index or its accessing time quite dramatically.

The internal nodes will be between 3 and 4 words in size. Each external node could be one word.

The tree organization is such that we can gain from the reading of large records.

The main ideas to solve (or alleviate) both problems are:

- Bucketing of external nodes
- mapping the tree onto the disk using super nodes.

Collecting more than one external node is called bucketing and is an old idea in data processing. A bucket replaces any subtree with size less than a certain constant (say b) and hence saves up to b - l internal nodes.

- Any search that has to be done in the bucket has to be done on all the members of the bucket.
- It increases the number of comparisons for each search.

Organizing the tree in super-nodes has advantages from the point of view of the number of accesses as well as in space.

**Super-Nodes**

- **Idea:** Disk access is slow. Each time we traverse nodes, if they're on different disk pages, we pay a cost.
- So instead of storing **one node per disk page**, we **pack many nodes into a single disk page (super-node)**.
- **One disk page** = ~1000 nodes (internal + external). Each page has **one entry point** (like a local root).

**Inside a page:** we traverse multiple nodes (up to ~10 levels of the tree) without additional disk access.

- If needed, we jump to another page via a pointer.

**Advantage:**

    - If tree height = 100 steps, then normally → 100 disk accesses.
    - With super-nodes (10 steps per page) → only 10 disk accesses.

**Example:**
Suppose tree height = 30.

    - Naive implementation → need up to 30 disk reads.
    - Super-nodes (10 steps/page) → need only 3 disk reads.
    - This makes searches much faster.
- In fact, every disk page has a single entry point, contains as much of the tree as possible, and terminates either in external nodes or in pointers to other disk pages.

Unfortunately, not all disk pages will be 100 percent full. A bottom-up greedy construction guarantees at least 50 percent occupations. Actual experiments indicate an occupation close to 80 percent.

With these constraints, disk pages will contain on the order of 1,000 internal/external nodes. This means that on the average, each disk page will contain about 10 steps of a root-to-leaf path, or in other words that the total number of accesses is a tenth of the height of the tree. Since it is very easy to keep the root page of the tree in memory, is the most efficient in terms of disk accesses for this type of search.

**PAT TREES REPRESENTATION AS ARRAYS**

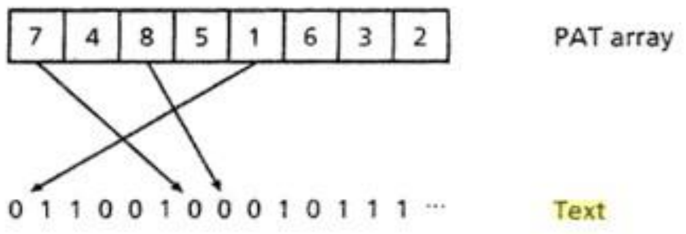
When a search reaches a bucket, we have to scan all the external nodes in the bucket to determine which if any satisfy the search. If the bucket is too large, these costs become prohibitive.

If the external nodes in the bucket are kept in the same relative order as they would be in the tree, then we do not need to do a sequential search, we could do an indirect binary search (i.e., compare the sistrings referred to by the external node) to find the nodes that satisfy the search. Cost of searching a bucket becomes 2 log b - 1 instead of b. here b means external node bucket size.

It is not significant for small buckets; it is a crucial observation that allows us to develop another implementation of PAT trees. This is simply to let the bucket grow well beyond normal bucket sizes, including the option of letting it be equal to n, that is, the whole index degenerates into a single array of external nodes ordered lexicographically by sistrings.

It shows that these arrays contain most of the information we had in the Patricia tree at the cost of a factor of log2 n. The argument simply says that for any interval in the array which contains all the external nodes that would be in a subtree, in log2 n comparisons in the worst case we can divide the interval according to the next bit which is different.

The sizes of the subtrees are trivially obtained from the limits of any portion of the array, so the only information that is missing is the longest-repetition bit, which is not possible to represent without an additional structure. Any operation on a Patricia tree can be simulated in O(log n) accesses.



**PAT Trees Represented as Arrays Searching PAT Trees as Arrays**

To simulate the Patricia tree for prefix and range searching, and we obtain an algorithm which is O(log n) instead of O(log2 n) for these operations. Actually, prefix searching and range searching become more uniform. Both can be implemented by doing an indirect binary search over the array with the results of the comparisons being less than, equal (or included in the case of range searching), and greater than. In this way, the searching takes at most 2 log2 n - 1 comparison and 4log2 n disk accesses.

In prefix searching and range searching can be done in time O(log2 n) with a storage cost that is exactly one word (a sistring pointer) per index point (sistring).

**Building PAT Trees as Arrays**

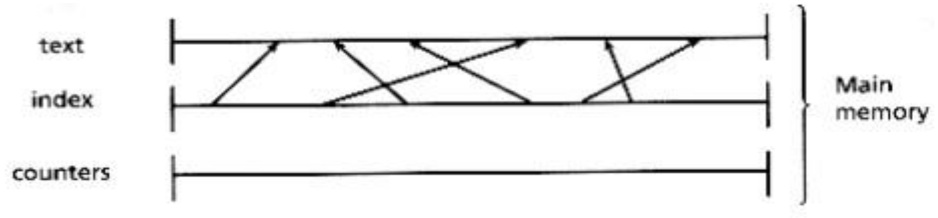
A historical note is worth entering at this point. Most of the research on this structure was done within the Centre for the New Oxford English Dictionary at the University of Waterloo, which was in charge of researching and implementing different aspects of the computerization of the OED from 1985. Hence, there was considerable interest in indexing the OED to provide fast searching of its 600Mb of text. A standard building of a Patricia tree in any of its forms would have required about n log2(n) t hours, where n is the number of index points and t is the time for a random access to disk.

**Building PAT arrays in memory**

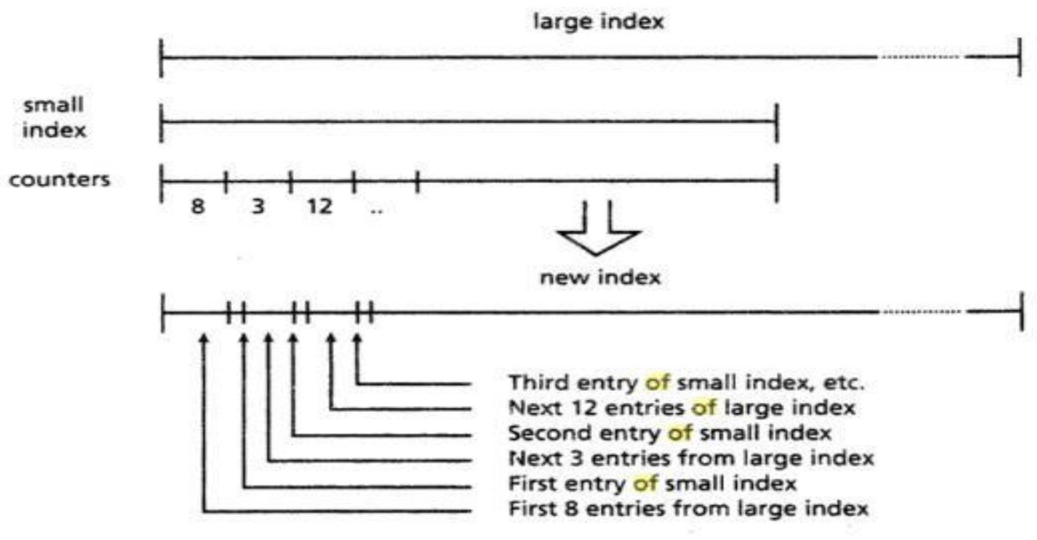
If a portion of the text is small enough to fit in main memory together with its PAT array, then this process can be done very efficiently as it is equivalent to string sorting. Note that here we are talking about main memory; if paging is used to simulate a larger memory, the random access patterns over the text will certainly cause severe memory thrashing.

**Merging small against large PAT arrays**

A second case that can be solved efficiently is the case of merging two indices (to produce a single one) when the text plus twice the index of one of them fits in main memory. This algorithm is not trivial and deserves a short explanation. The text of the small file together with a PAT array for the small file (of size n1) plus an integer array of size n1 + 1 are kept in main memory. The integer array is used to count how many sistrings of the big file fall between each pair of index points in the small file.



**Small index in main memory**


large index
small index
counters
8
3
12
..
new index
Third entry of small index, etc.
Next 12 entries of large index
Second entry of small index
Next 3 entries from large index
First entry of small index
First 8 entries from large index

**Merging the small and the large index**

**Delayed Reading Paradigm**

Comparing sistrings through random access to disk is so expensive is a consequence of two phenomena. First, the reading itself requires some relatively slow) physical movement. Second, the amount of text actually used is a small fraction of the text made available by an I/O operation.

A programming technique that tries to alleviate the above problem without altering the underlying algorithms. The technique is simply to suspend execution of the program every time a random input is required, store these requests in a request pool, and when convenient/necessary satisfy all requests in the best ordering possible.

Some of the following modules are:

- The index building program
- a list of blocked requests
- a list of satisfied requests

**Merging Large Files**

To merge two or more large indices. We can do this by reading the text associated with each key and comparing. If we have n keys and m files, and use a heap to organize the current keys for each file being merged, this gives us O(n log m) comparisons.

An improvement can be made by reducing the number of random disk accesses and increasing the amount of sequential disk I/O. The sistring pointers in the PAT array are ordered lexicographically according to the text that they reference and so will be more or less randomly ordered by location in the text.

If we can read a sufficiently large block of pointers at one time and indirectly sort them by location in the text, then we can read a large number of keys from the text in a sequential pass. The larger the number of keys read in a sequential pass, the greater the improvement in performance.

The greatest improvement is achieved when the entire memory is allocated to this key reading for each merge file in turn. The keys are then written out to temporary disk space. These keys are then merged by making a sequential pass over all the temporary files.

Thus, there are two constraints affecting this algorithm: the size of memory and the amount of temporary disk space available. They must be balanced by the relationship

**memory X number of files = temporary disk space**