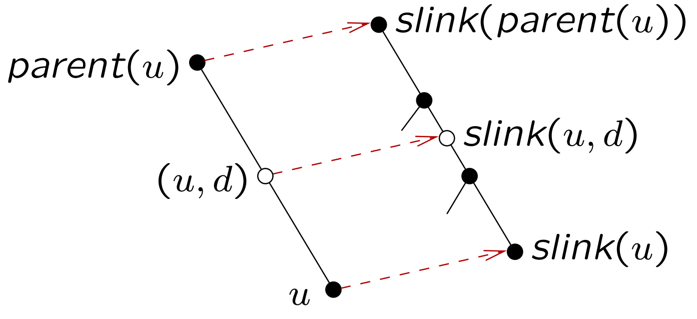
Suffix links are the same as Aho–Corasick failure links but Lemma 4.4 ensures that *depth*(*slink*(u)) = *depth*(u) − 1. This is not the case for an arbitrary trie or a compact trie.

Suffix links are stored for compact trie nodes only, but we can define and compute them for any position represented by a pair (u, d):

```
slink(u, d)
  (1)  $v \leftarrow$ slink(parent($u$))
  (2)  while depth($v$) $< d - 1$ do
  (3)          $v \leftarrow$ child($v, T$[start($u$) + depth($v$) + 1])
  (4)  return $(v, d - 1)$
```



The same idea can be used for computing the suffix links during or after the brute force construction.

```
ComputeSlink($u$)
 (1) $v \leftarrow \mathit{slink}(\mathit{parent}(u))$
 (2) while $\mathit{depth}(v) < \mathit{depth}(u) - 1$ do
 (3)     $v \leftarrow \mathit{child}(v, T[\mathit{start}(u) + \mathit{depth}(v) + 1])$
 (4) if $\mathit{depth}(v) > \mathit{depth}(u) - 1$ then
 (5)     create new node $w$
 (6)     $\mathit{start}(w) \leftarrow \mathit{start}(u) + 1$; $\mathit{depth}(w) \leftarrow \mathit{depth}(u) - 1$; $\mathit{slink}(w) \leftarrow \bot$
 (7)     $p \leftarrow \mathit{parent}(v)$
 (8)     $\mathit{child}(w, T[\mathit{start}(v) + \mathit{depth}(w)]) \leftarrow v$; $\mathit{parent}(v) \leftarrow w$
 (9)     $\mathit{child}(p, T[\mathit{start}(w) + \mathit{depth}(p)]) \leftarrow w$; $\mathit{parent}(w) \leftarrow p$
(10)     $v \leftarrow w$
(11) $\mathit{slink}(u) \leftarrow v$
```

The algorithm uses the suffix link of the parent, which must have been computed before. Otherwise the order of computation does not matter.

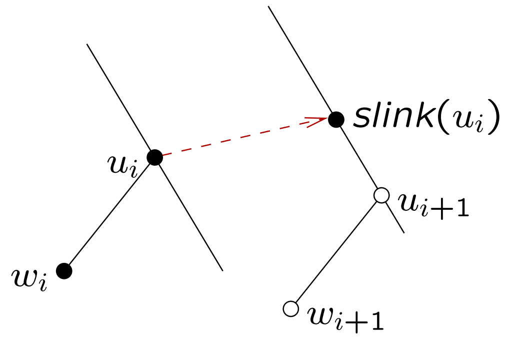
The creation of a new node on lines (4)–(10) is not necessary in a fully constructed suffix tree, but during the brute force algorithm the necessary node may not exist yet:

- If a new internal node u_i was created during the insertion of the suffix T_i, there exists an earlier suffix T_j, $j < i$ that branches at u_i into a different direction than T_i.
- Then *slink*(u_i) represents a prefix of T_{j+1} and thus exists at least as a position on the path labelled T_{j+1}. However, it may be that it does not become a branching node until the insertion of T_{i+1}.
- In such a case, ComputeSlink(u_i) creates *slink*(u_i) a moment before it would otherwise be created by the brute force construction.

McCreight's Algorithm

McCreight's suffix tree construction is a simple modification of the brute force algorithm that computes the suffix links during the construction and uses them as short cuts:

- Consider the situation, where we have just added a leaf w_i representing the suffix T_i as a child to a node u_i. The next step is to add w_{i+1} as a child to a node u_{i+1}.
- The brute force algorithm finds u_{i+1} by traversing from the root. McCreight's algorithm takes a short cut to *slink*(u_i).



- This is safe because *slink*(u_i) represents a prefix of T_{i+1}.

Algorithm 5.5: McCreight
Input: text $T[0..n]$ ($T[n] = \$$)
Output: suffix tree of T: $root$, *child*, *parent*, *depth*, *start*, *slink*

```
(1)  create new node $root$; $depth(root) \leftarrow 0$; $slink(root) \leftarrow root$
(2)  $u \leftarrow root$; $d \leftarrow 0$
(3)  for $i \leftarrow 0$ to $n$ do        // insert suffix $T_i$
(4)      while $d = depth(u)$ and $child(u, T[i+d]) \neq \perp$ do
(5)          $u \leftarrow child(u, T[i+d])$; $d \leftarrow d+1$
(6)          while $d < depth(u)$ and $T[start(u)+d] = T[i+d]$ do $d \leftarrow d+1$
(7)      if $d < depth(u)$ then        // we are in the middle of an edge
(8)          create new node $v$
(9)          $start(v) \leftarrow i$; $depth(v) \leftarrow d$; $slink(v) \leftarrow \perp$
(10)         $p \leftarrow parent(u)$
(11)         $child(v, T[start(u)+d]) \leftarrow u$; $parent(u) \leftarrow v$
(12)         $child(p, T[i + depth(p)]) \leftarrow v$; $parent(v) \leftarrow p$
(13)         $u \leftarrow v$
(14)     create new leaf $w$        // $w$ represents suffix $T_i$
(15)     $start(w) \leftarrow i$; $depth(w) \leftarrow n - i + 1$
(16)     $child(u, T[i+d]) \leftarrow w$; $parent(w) \leftarrow u$
(17)     if $slink(u) = \perp$ then ComputeSlink($u$)
(18)     $u \leftarrow slink(u)$; $d \leftarrow d - 1$
```

Theorem 5.6: Let T be a string of length n over an alphabet of constant size. McCreight's algorithm computes the suffix tree of T in $\mathcal{O}(n)$ time.

Proof. Insertion of a suffix T_i takes constant time except in two points:

- The while loops on lines (4)–(6) traverse from the node *slink*(u_i) to u_{i+1}. Every round in these loops increments d. The only place where d decreases is on line (18) and even then by one. Since d can never exceed n, the total time on lines (4)–(6) is $\mathcal{O}(n)$.

- The while loop on lines (2)–(3) during a call to ComputeSlink(u_i) traverses from the node *slink*(*parent*(u_i)) to *slink*(u_i). Let d'_i be the depth of *parent*(u_i). Clearly, $d'_{i+1} \geq d'_i - 1$, and every round in the while loop during ComputeSlink(u_i) increases d'_{i+1}. Since d'_i can never be larger than n, the total time in the loop on lines (2)–(3) in ComputeSlink is $\mathcal{O}(n)$.

$\square$

There are other linear time algorithms for suffix tree construction:

- Weiner's algorithm was the first. It inserts the suffixes into the tree in the opposite order: $T_n, T_{n-1}, \ldots, T_0$.

- Ukkonen's algorithm constructs suffix tree first for $T[0..1)$ then for $T[0..2)$, etc.. The algorithm is structured differently, but performs essentially the same tree traversal as McCreight's algorithm.

- All of the above are linear time only for constant alphabet size. Farach's algorithm achieves linear time for an integer alphabet of polynomial size. The algorithm is complicated and unpractical.

Applications of Suffix Tree

Let us have a glimpse of the numerous applications of suffix trees.

Exact String Matching

As already mentioned earlier, given the suffix tree of the text, all occ occurrences of a pattern P can be found in time $\mathcal{O}(|P| + occ)$.

Even if we take into account the time for constructing the suffix tree, this is asymptotically as fast as Knuth–Morris–Pratt for a single pattern and Aho–Corasick for multiple patterns.

However, the primary use of suffix trees is in indexed string matching, where we can afford to spend a lot of time in preprocessing the text, but must then answer queries very quickly.

Approximate String Matching

Several approximate string matching algorithms achieving $\mathcal{O}(kn)$ worst case time complexity are based on suffix trees.

Filtering algorithms that reduce approximate string matching to exact string matching such as partitioning the pattern into $k+1$ factors, can use suffix trees in the filtering phase.

Another approach is to generate all strings in the k-neighborhood of the pattern, i.e., all strings within edit distance k from the pattern and search for them in the suffix tree.

The best practical algorithms for indexed approximate string matching are hybrids of the last two approaches.

Text Statistics

Suffix tree is useful for computing all kinds of statistics on the text. For example:

- The number of distinct factors in the text is exactly the number of nodes in the (uncompact) trie. Using the suffix tree, this number can be computed as the total length of the edges plus one (root/empty string). The time complexity is $\mathcal{O}(n)$ even though the resulting value is typically $\Theta(n^2)$.

- The longest repeating factor of the text is the longest string that occurs at least twice in the text. It is represented by the deepest internal node in the suffix tree.

Generalized Suffix Tree

A generalized suffix tree of two strings S and T is the suffix three of the string $S£T\$$, where £ and \$ are symbols that do not occur elsewhere in S and T.

Each leaf is marked as an S-leaf or a T-leaf according to the starting position of the suffix it represents. Using a depth first traversal, we determine for each internal node if its subtree contains only S-leafs, only T-leafs, or both. The deepest node that contains both represents the longest common factor of S and T. It can be computed in linear time.

The generalized suffix tree can also be defined for more than two strings.

AC Automaton for the Set of Suffixes

As already mentioned, a suffix tree with suffix links is essentially an Aho–Corasick automaton for the set of all suffixes.

- We saw that it is possible to follow suffix link / failure transition from any position, not just from suffix tree nodes.
- Following such an implicit suffix link may take more than a constant time, but the total time during the scanning of a string with the automaton is linear in the length of the string. This can be shown with a similar argument as in the construction algorithm.

Thus suffix tree is asymptotically as fast to operate as the AC automaton, but needs much less space.

Matching Statistics

The matching statistics of a string $T[0..n)$ with respect to a string S is an array $MS[0..n)$, where $MS[i]$ is a pair (ℓ_i, p_i) such that

1. $T[i..i + \ell_i)$ is the longest prefix of T_i that is a factor of S, and
2. $S[p_i..p_i + \ell_i) = T[i..i + \ell_i)$.

Matching statistics can be computed by using the suffix tree of S as an AC-automaton and scanning T with it.

- If before reading $T[i]$ we are at the node v in the automaton, then $T[i - d..i) = S[j..j + d)$, where $j =$ *start*(v) and $d =$ *depth*(v).
- If reading $T[i]$ causes a failure transition, then $MS[i - d] = (d, j)$. Following the failure transtion decrements d and thus increments $i - d$ by one.
- Following a normal transition, increments both i and d by one, and thus $i - d$ stays the same.

From the matching statistics, we can easily compute the longest common factor of S and T. Because we need the suffix tree only for S, this saves space compared to a generalized suffix tree.

Matching statistics are also used in some approximate string matching algorithms.

LCA Preprocessing

The lowest common ancestor (LCA) of two nodes u and v is the deepest node that is an ancestor of both u and v. Any tree can be preprocessed in linear time so that the LCA of any two nodes can be computed in constant time. The details are omitted here.

- Let w_i and w_j be the leaves of the suffix tree of T that represent the suffixes T_i and T_j. The lowest common ancestor of w_i and w_j represents the longest common prefix of T_i and T_j. Thus the lcp of any two suffixes can be computed in constant time using the suffix tree with LCA preprocessing.

- The longest common prefix of two suffixes S_i and T_j from two different strings S and T is called the longest common extension. Using the generalized suffix tree with LCA preprocessing, the longest common extension for any pair of suffixes can be computed in constant time.

Some $\mathcal{O}(kn)$ worst case time approximate string matching algorithms use longest common extension data structures.

Longest Palindrome

A palindrome is a string that is its own reverse. For example, `saippuakauppias` is a palindrome.

We can use the LCA preprocessed generalized suffix tree of a string T and its reverse T^R to find the longest palindrome in T in linear time.

- Let k_i be the length of the longest common extension of T_i and T^R_{n-i-1}, which can be computed in constant time. Then $T[i-k_i..i+k_i]$ is the longest odd length palindrome with the middle at i.

- We can find the longest odd length palindrome by computing k_i for all $i \in [0..n)$ in $\mathcal{O}(n)$ time.

- The longest even length palindrome can be found similarly in $\mathcal{O}(n)$ time.

Suffix Array

The suffix array of a text T is a lexicographically ordered array of the set $T_{[0..n]}$ of all suffixes of T. More precisely, the suffix array is an array $SA[0..n]$ of integers containing a permutation of the set $[0..n]$ such that $T_{SA[0]} < T_{SA[1]} < \cdots < T_{SA[n]}$.

A related array is the inverse suffix array SA^{-1} which is the inverse permutation, i.e., $SA^{-1}[SA[i]] = i$ for all $i \in [0..n]$.

As with suffix trees, it is common to add the end symbol $T[n] = \$$. It has no effect on the suffix array assuming $ is smaller than any other symbol.

Example 5.7: The suffix array and the inverse suffix array of the text $T =$ banana$.

i	$SA[i]$	$T_{SA[i]}$
0	6	$
1	5	a$
2	3	ana$
3	1	anana$
4	0	banana$
5	4	na$
6	2	nana$

j	$SA^{-1}[j]$	
0	4	banana$
1	3	anana$
2	6	nana$
3	2	ana$
4	5	na$
5	1	a$
6	0	$

Suffix array is much simpler data structure than suffix tree. In particular, the type and the size of the alphabet are usually not a concern.

- The size on the suffix array is $\mathcal{O}(n)$ on any alphabet.

- We will see that the suffix array can be constructed in the same asymptotic time it takes to sort the characters of the text.

As with suffix trees, exact string matching in T can be performed by prefix search on the suffix array. The answer can be conveniently given as a contiguous range in the suffix array containing the suffixes. The range can be found using string binary search.

- If we have the additional arrays $LLCP$ and $RLCP$, the result range can be computed in $\mathcal{O}(|P| + \log n)$ time.

- Without the additional arrays, we have the same time complexity on average but the worst case time complexity is $\mathcal{O}(|P| \log n)$.

- We can then count the number of occurrences in $\mathcal{O}(1)$ time, list all occ occurrences in $\mathcal{O}(occ)$ time, or list a sample of k occurrences in $\mathcal{O}(k)$ time.

LCP Array

Efficient string binary search uses the arrays $LLCP$ and $RLCP$. For many applications, the suffix array is augmented with a different lcp array $LCP[1..n]$. For all i,

$$LCP[i] = lcp(T_{SA[i]}, T_{SA[i-1]})$$

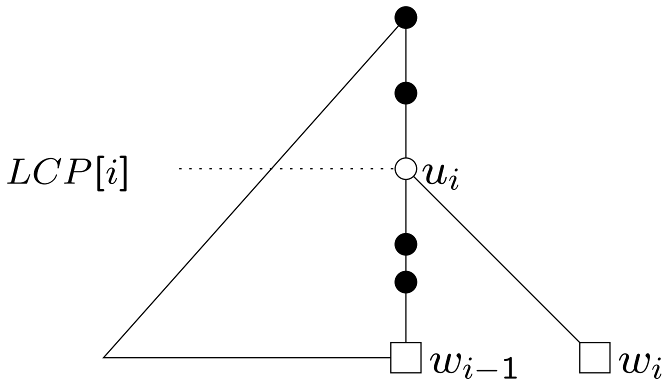
This is the same as the lcp information in the output of StringMergesort.

Example 5.8: The LCP array for $T =$ `banana$`.

i	$SA[i]$	$LCP[i]$	$T_{SA[i]}$
0	6		`$`
1	5	0	`a$`
2	3	1	`ana$`
3	1	3	`anana$`
4	0	0	`banana$`
5	4	0	`na$`
6	2	2	`nana$`

The suffix tree can be easily constructed from the suffix and LCP arrays in linear time.

- Insert the suffixes into the tree in lexicographical order.
- The leaf w_i representing the suffix T_i is inserted as the rightmost leaf. The parent u_i of w_i is along the rightmost path in the tree, and the depth of u_i is $LCP[i]$. If there is no node at that depth, a new node is inserted.



- Keep the nodes on the rightmost path on a stack with the deepest node on top. The node u_i or the edge, where u_i is inserted, is found by removing nodes from the stack until the right depth has been reached. Note that the removed nodes are no more on the rightmost path after the insertion of w_i.

The suffix tree can be replaced by the suffix and LCP arrays in many applications. For example:

- The longest repeating factor is marked by the maximum value in the LCP array.

- The number of distinct factors can be compute by the formula

$$\frac{n(n+1)}{2} + 1 - \sum_{i=1}^{n} LCP[i]$$

  This follows from the suffix tree construction on the previous slide and the formula we saw earlier for the suffix tree.

- Matching statistics of T with respect to S can be computed in linear time using the generalized suffix array of S and T (i.e., suffix array of $S£T\$$) and its LCP array.

RMQ Preprocessing

The range minimum query (RMQ) asks for the smallest value in a given range in an array. Any array can be preprocessed in linear time so that RMQ for any range can be answered in constant time.

In the LCP array, RMQ can be used for computing the lcp of any two suffixes.

Lemma 5.9: The length of the longest common prefix of two suffixes $T_i < T_j$ is $lcp(T_i, T_j) = \min\{LCP[k] \mid k \in [SA^{-1}[i] + 1..SA^{-1}[j]]\}$.

The proof is left as an exercise.

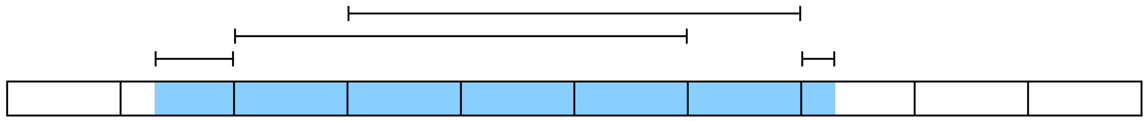
- The RMQ preprocessing of the LCP array supports the same kind of applications as the LCA preprocessing of the suffix tree. RMQ preprocessing is much simpler than LCA preprocessing.

- The RMQ preprocessed LCP array can also replace the LLCP and RLCP arrays.

We will next describe the RMQ data structure for an arbitrary array $L[1..n]$ of integers.

- We precompute and store the minimum values for the following collection of ranges:
  - Divide $L[1..n]$ into blocks of size $\log n$.
  - For all $0 \leq \ell \leq \log(n/\log n))$, include all ranges that consist of 2^ℓ blocks. There are $\mathcal{O}(n)$ such ranges.
  - Include all prefixes and suffixes of blocks. There are $\mathcal{O}(n)$ such ranges.

- Now any range $L[i..j]$ that overlaps or touches a block boundary can be exactly covered by at most four ranges in the collection.



  The minimum value in $L[i..j]$ is the minimum of the minimums of the covering ranges and can be computed in constant time.

Ranges $L[i..j]$ that are completely inside one block are handled differently.

- Let $P(i) = \min\{k > i \mid L[k] < L[i]\}$. Then the position of the minimum value in the range $L[i..j]$ is the last position in the sequence $i, P(i), P(P(i)), \ldots$ that is in the range.

- Store the positions $i, P(i), P(P(i)), \ldots$ up to the end of the block containing i as a bit vector $B(i)$ of at most $\log n$ bits. We assume that $\log n$ bits fits in a single machine word. Thus we need $\mathcal{O}(n)$ words to store $B(i)$ for all i.

- The position of the minimum in $L[i..j]$ is found as follows:
  - Turn all bits in $B(i)$ after position j into zeros. This can be done in constant time using bitwise shift- and and-operations.
  - The right-most 1-bit indicates the position of the minimum. It can be found in constant time using a lookup table of size $\mathcal{O}(n)$.

All the data structures can be constructed in $\mathcal{O}(n)$ time (exercise).

Enhanced Suffix Array

The enhanced suffix array adds two more arrays to the suffix and LCP arrays to make the data structure fully equivalent to suffix tree.

- The idea is to represent a suffix tree node v by a suffix array range corresponding to the suffixes that are in the subtree rooted at v.

- The additional arrays support navigation in the suffix tree using this representation: one array along the regular edges, the other along suffix links.