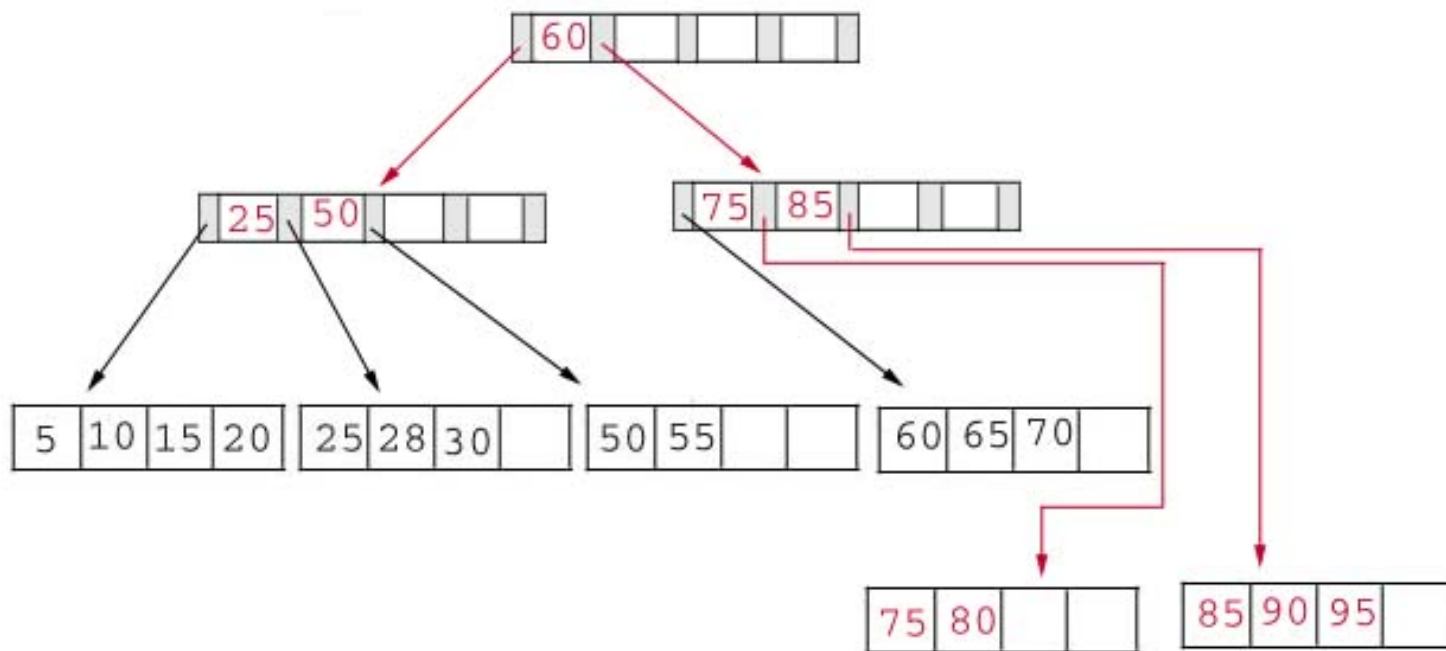


Algorithms and Analysis

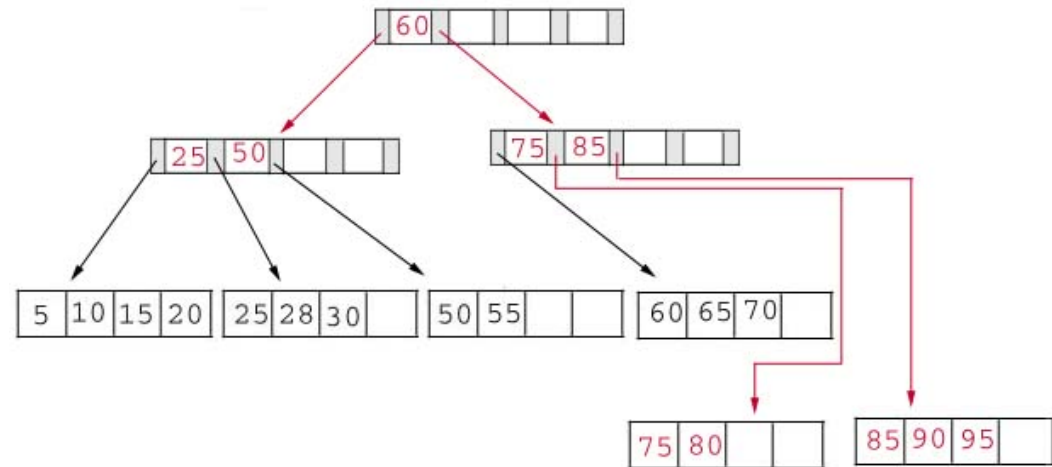
Lesson 9: *Sometimes It Pays Not to Be Binary*



B-Trees, Tries, Suffix Trees

Outline

1. **B-Trees**
2. Tries
3. Suffix Tree



B-Trees

- **B-trees** are balanced trees for fast search, finding successors and predecessors, insert, delete, maximum, minimum, etc.■
- Not to be confused with binary trees■
- They are designed to keep related data close to each other in (disk) memory to minimise retrieval time■
- Important when working with large amount of data that is stored on secondary storage (e.g. disks)■
- Used extensively in databases■

When Big-O Doesn't Work

- An underlying assumption of Big-O is that all elementary operations take roughly the same amount of time■
- This just isn't true of disk look-up■
- The typical time of an elementary operation on a modern processor is 10^{-9} seconds■
- But a typical hard disk might do 7 200 revolutions per minute or 120 revolutions per second■
- The typical time it takes to locate a record is around 10ms or 10^7 times slower than an elementary operation■

Accessing Data from Disk

- When accessing data from disk minimising the number of disk accesses is critical for good performance■
- In database applications we want to store data as large sets■
- Storing data in binary trees is disastrous as we typically need around $\log_2(n)$ disk accesses before we locate our data■
- It is not unusual in databases for $n = 10\,000\,000$ so that $\log_2(n) \approx 24$ ■
- Using binary trees it would often take several seconds to find a record■

Multiway-Trees

- To remedy this we can use M-way trees so that the access time is

$$\log_M(n) = \frac{\log_2(n)}{\log_2(M)}$$

- In practice we might use $M \approx 200 \approx 2^8$ so we can reduce the depth of the tree by around a factor of 8
- The basic data structures for doing this is the B-tree
- There are many variants of B-tree, all trying to squeeze a bit more performances from the basic structure

B⁺ Tree

- A pretty basic implementation would obey the following rules
 1. The data items are stored at leaves■
 2. The non-leaf nodes store up to $M-1$ keys to guide the search:
key i represents the smallest key in subtree $i + 1$ ■
 3. The root is either a leaf or has between 2 and M children■
 4. All non-leaf nodes except the root have between $\lceil M/2 \rceil$ and M children■
 5. All leaves are at the same depth and have between $\lceil L/2 \rceil$ and L data entries■

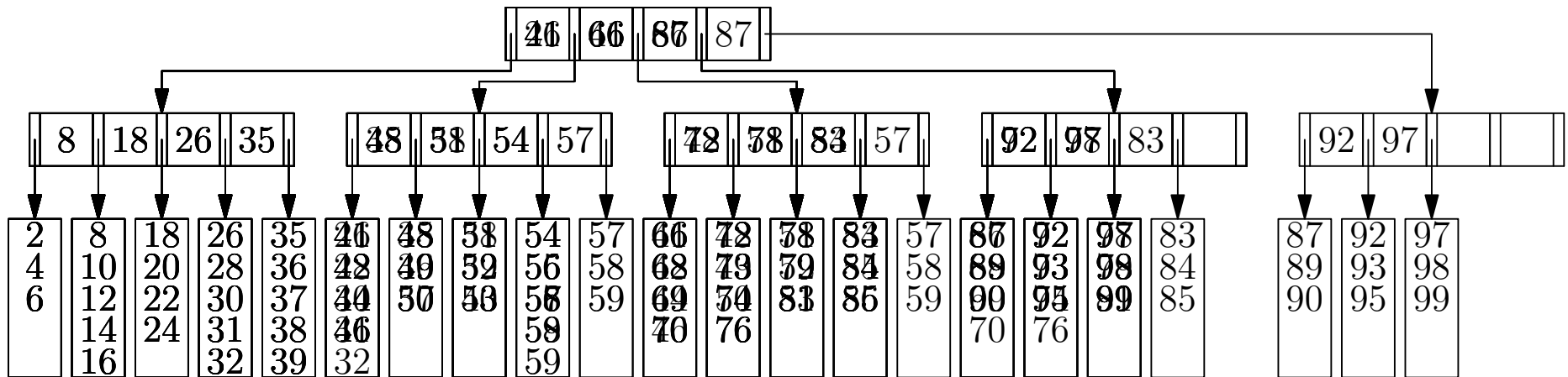
Choosing M and L

- The choice of M and L depends on the block size (the information read in one go from disk)■
- It also depends on the type of data that is being stored (integer, reals, strings, etc.)■
- M and L might be in the hundreds or thousands■
- In the examples below we consider tiny $M = L = 5$ which is unrealistic, but drawable■

B-Tree Example

- $M = 5, L = 5$

insert(80)



Other Changes

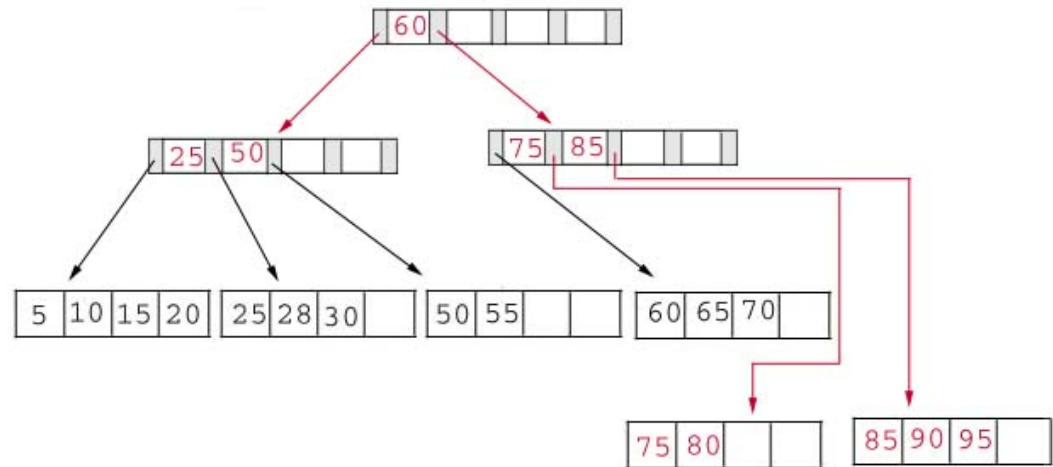
- If the root is full then it can be split into two and a new root created■
- B-trees also have to allow the removal of records without losing its structure■
- There are a number of variant strategies (e.g. neighbouring nodes can adopt a child if the current node cannot expand any more)■
- The actual implementation of B-trees is tricky because there are many special cases■

B-Tree Summary

- B-trees are an important data structure for databases where reducing the number of disk searches is vital■
- They tend to be much more complex than the other data structures we have seen■
- The problem of disk access can be improved by replacing disk memory with solid-state drives (still slow compared to memory)■
- For massive databases new data structures have been developed to allow faster (although less flexible) information access (e.g. NOSQL, MongoDB, Neo4j)■

Outline

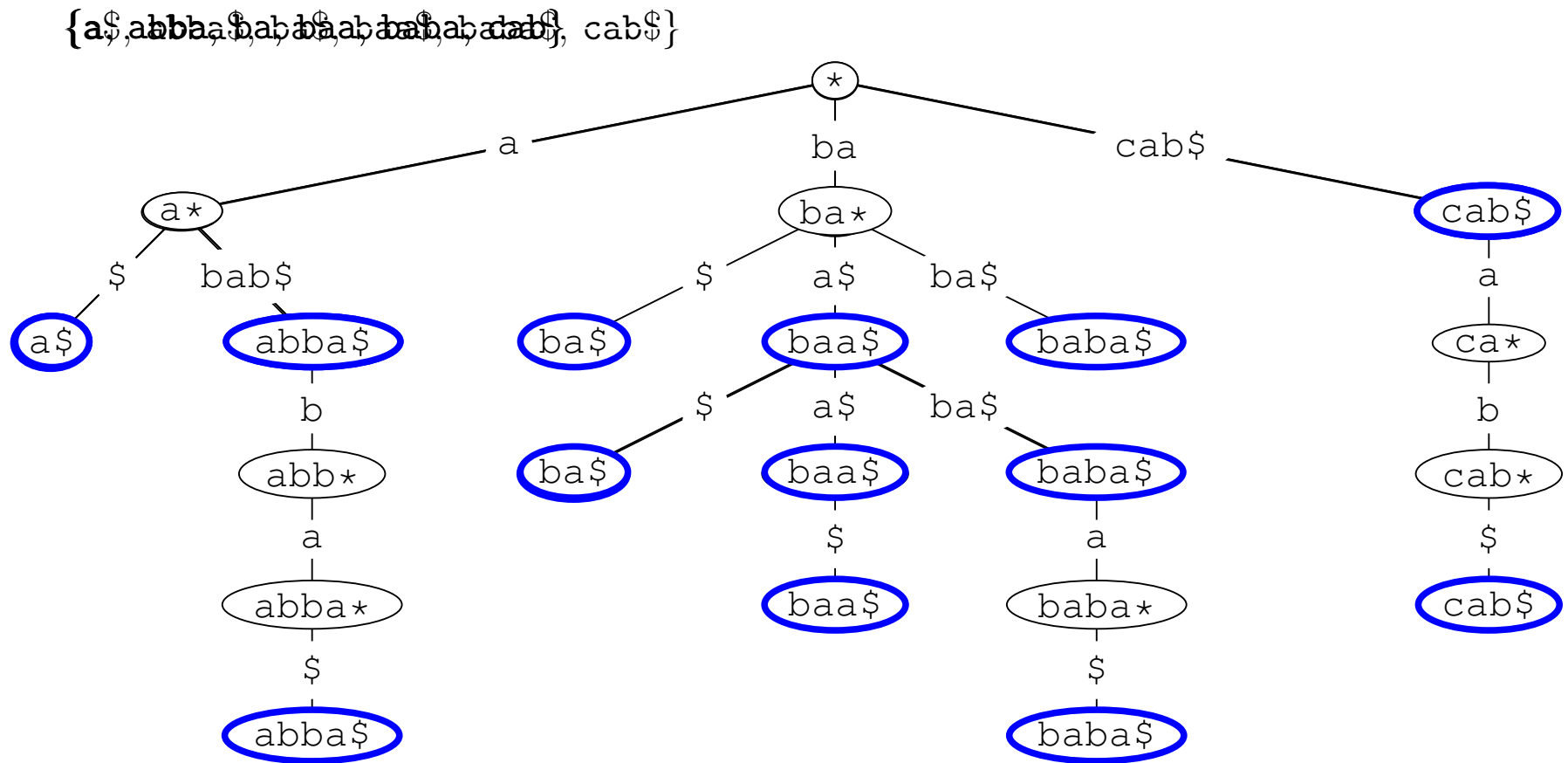
1. B-Trees
2. **Tries**
3. Suffix Tree



Tries

- A **Trie** (pron. 'try') or **digital tree** is a multiway tree often used for storing large sets of words■
- They are trees with a possible branch for every letter of an alphabet■
- Their names comes from the word *re**trie**val*■
- Tries usually compactify the edges in the tree■
- All words end with a special letter “\$”■

Trie



Uses of Tries

- Tries are yet another way of implementing sets■
- They provide quick insertion, deletion and find■
- Typically considerably quicker than binary trees and hash tables■
- They are particularly good for spell checkers, completion algorithms, longest-prefix matching, hyphenation■
- Each search finds the longest match between the words in the set and the query■

Trie for 31 Most Common English Words

search(0,0,0,0,0,0,0,0,0,0,0,0) Failure

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|----|---------|------|---------|--------|---------|-------|--------|-------|-------|------|--------|
| \$ | | I\$ | | | | A\$ | | HE\$ | | | |
| A | AA(\$) | | HAI(\$) | | WAS\$ | | THAT\$ | | | | |
| B | BU(\$) | | | | | | | | | | |
| C | | | | | | | | | | | |
| D | | | | HAD\$ | | | | | | | |
| E | | | HEE(\$) | | | | THE\$ | | BE\$ | | |
| F | FO(\$) | | | | | | | | | OF\$ | |
| G | | | | | | | | | | | |
| H | HA(\$) | | | | WHICH\$ | | | | | | |
| I | IN(\$) | | HIS\$ | | WITH\$ | | THIS\$ | | | | |
| J | | | | | | | | | | | |
| K | | | | | | | | | | | |
| L | | | | | | | | | | | |
| M | | | | | | | | | | | |
| N | NOT\$ | IN\$ | | | | AND\$ | | | | ON\$ | |
| O | ON(\$) | | | | | | | | | | FOR\$ |
| P | | | | | | | | | | | |
| Q | | | | | | | | | | | |
| R | | | | | | ARE\$ | | HER\$ | | OR\$ | FROM\$ |
| S | | IS\$ | | | | AS\$ | | | | | |
| T | THA(\$) | IT\$ | | | | AT\$ | | | | | |
| U | | | | | | | | | BUT\$ | | |
| V | | | | HAVE\$ | | | | | | | |
| W | WI(\$) | | | | | | | | | | |
| X | | | | | | | | | | | |
| Y | YOU\$ | | | | | | | | BY\$ | | |
| Z | | | | | | | | | | | |

Disadvantage of Tries

- Table-based tries typically waste large amounts of memory■
- Often table-based tries are used for the first few layers, while lower levels use a less memory intensive data structure■
- These days memory is less of a problem so table-based tries are acceptable for some applications■
- There are many implementations of tries each suited to a particular task■

Binary Tries

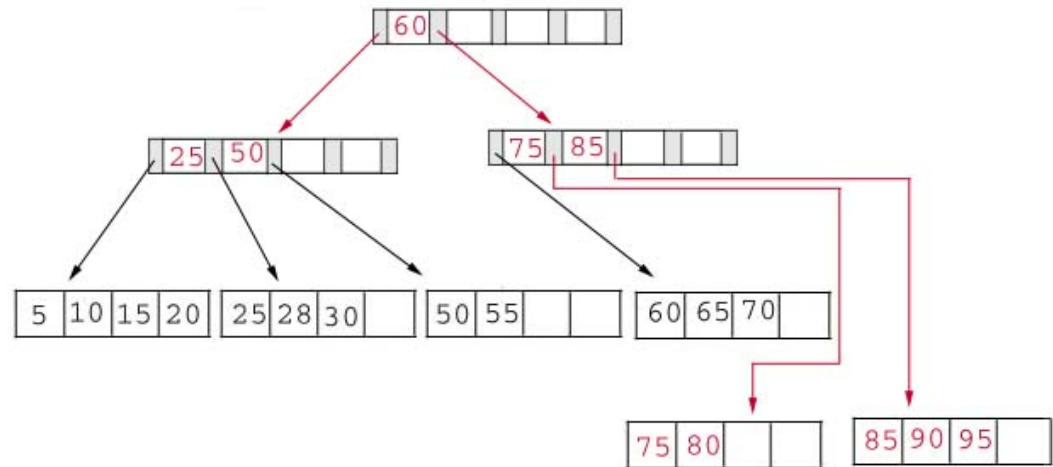
- One extreme (though not uncommon) solution to address memory issues is to build a bit-level trie so the data-structure is a binary tree■
- It differs from a binary tree in that the decisions to go left or right depends on the current bit■
- Although you lose the advantage of a multiway tree (of reducing the depth) it does find the longest match and it speeds up finds which fail■

Why Tries?

- Tries are a classic example of a trade-off between memory and computational complexity■
- Tries are slightly specialist and tend to get used in very particular applications
 - ★ Finding longest matches
 - ★ Completion, spell checking, etc.■
- A basic trie is not too complicated, however, . . . ■
- There are many implementation which try to overcome the difficulty of wasting too much memory■

Outline

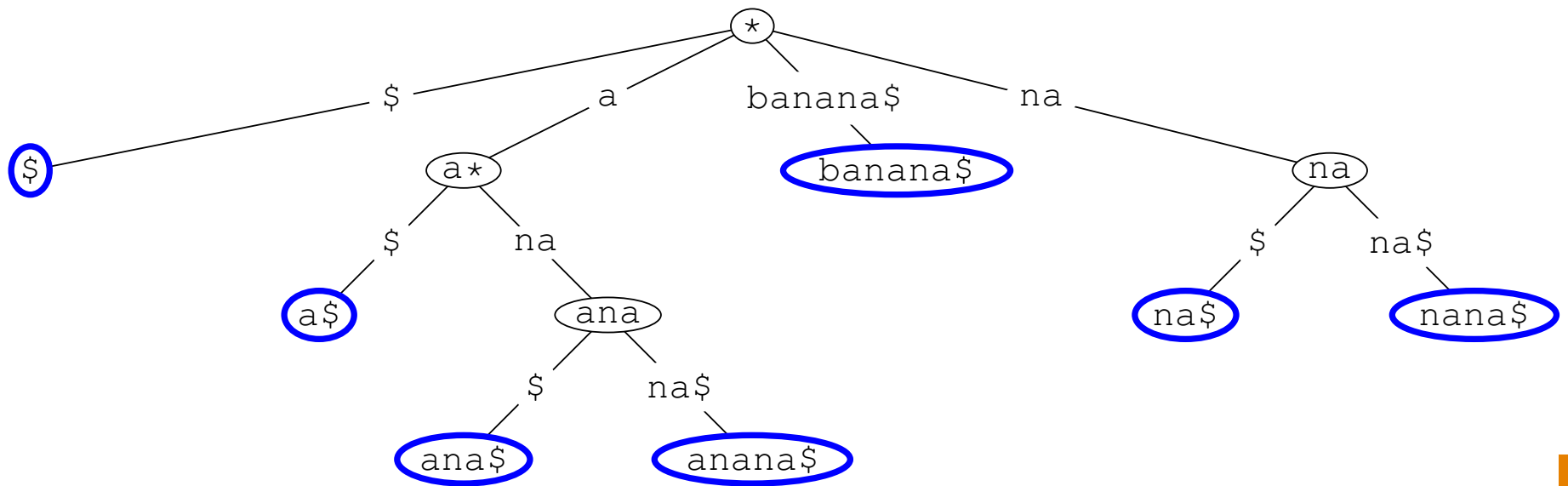
1. B-Trees
2. Tries
3. **Suffix Tree**



Suffix Tree

- Suffix tree is a trie of all suffixes of a string
- E.g. banana

{\$, a\$, na\$, ana\$, nana\$, anana\$, banana\$}

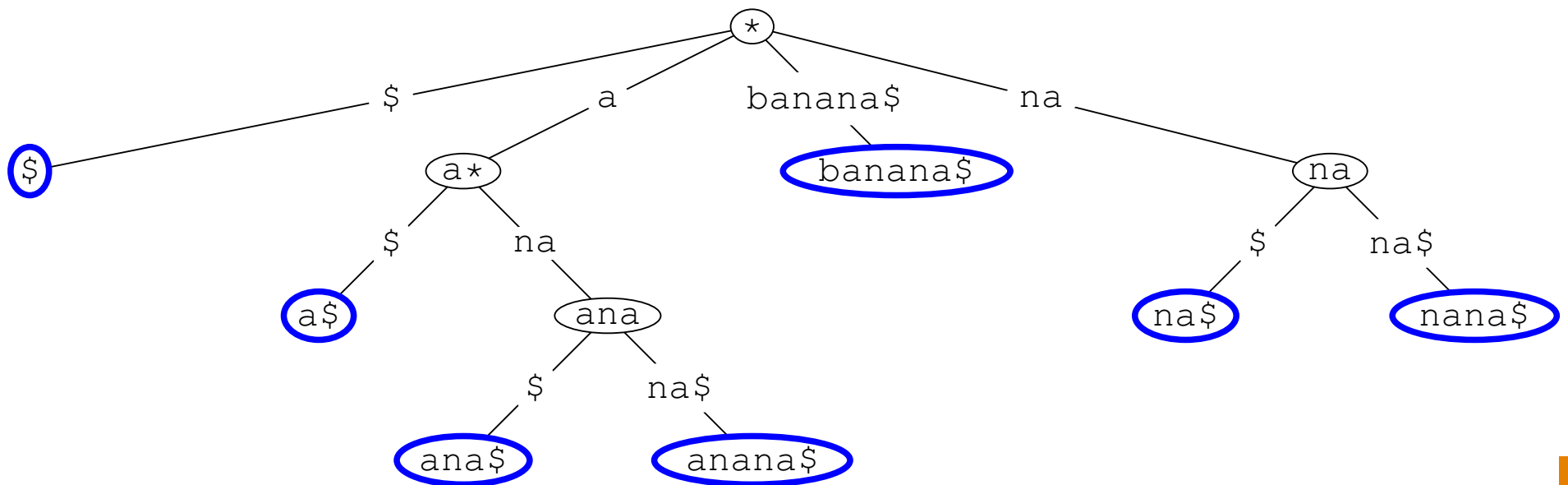


Importance of Suffix Tree

- The first linear-time algorithm for computing suffix trees was proposed by Peter Weiner in 1973, a more space efficient algorithm was proposed by Edward M. McCreight in 1976■
- Esko Ukkonen in 1995 proposed a variant of McCreight's algorithm, but in a way that was much easier to understand■
- It really only got implemented after this■
- They are very important for string-based algorithms■
- The classic application is in finding a match for a query string, Q , in a text, T ■

String Matching

- To find a match of a query string, Q , in a text, T , we can first construct the suffix tree of the string T we then simple look up the query, Q , using the trie

$$\{\$, \text{ a}\$, \text{ na}\$, \text{ ana}\$, \text{ nana}\$, \text{ anana}\$, \text{ banana}\$\}$$


Complexity of Suffix Trees

- Using a regular trie for a suffix tree would typically use far too much memory to be useful■
- However, by using pointers to the original text it is possible to build a suffix tree using $O(n)$ memory where n is the length of the text■
- Furthermore (and rather incredibly) there is a linear time ($O(n)$) algorithm to construct the trie■
- The algorithm is not however trivial to understand■

Uses of Suffix Trees

- Suffix trees are efficient whenever it is likely that you will do multiple searches■
- Exact word matching is in itself a very important application■
- Suffix trees in combination with dynamic programming (which we will eventually get to) can be used to do inexact matching (finding the match with the smallest edit distance)■
- Suffix trees get used in bioinformatics, advanced machine learning algorithms, . . . ■

Lessons

- Multiway trees can considerably speed up search over binary trees■
- They are very important in some specialised applications (e.g. databases, spell-checking, completion, suffix trees)■
- They are not as general purpose as binary search trees and are more complicated to implement■
- But they can give the best performance■—sometimes performance matters enough to make it worthwhile implementing multiway trees■