Advanced Algorithm Design and Analysis (Lecture 4)

SW5 fall 2004

Simonas Šaltenis
E1-215b
simas@cs.aau.dk

Text-Search Data Structures

- Goals of the lecture:
  - ***Dictionary ADT*** *for strings:*
    - *to understand the principles of **tries**, compact tries, Patricia tries*
  - *Text-searching data structures:*
    - *to understand and be able to analyze text searching algorithm using the **suffix tree** and Pat tree*
  - *Full-text indices in external memory:*
    - *to understand the main principles of String B-trees.*

Dictionary ADT for Strings

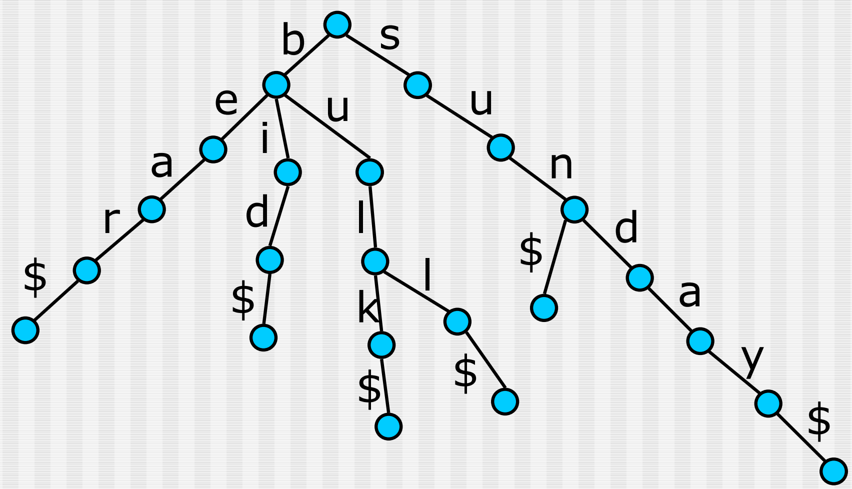
- *Dictionary* ADT for strings – stores a set of text strings:
  - *search*(x) – checks if string x is in the set
  - *insert*(x) – inserts a new string x into the set
  - *delete*(x) – deletes the string equal to x from the set of strings
- Assumptions, notation:
  - n strings, N characters in total
  - m – length of x
  - Size of the alphabet $d = |\Sigma|$

BST of Strings

- We can, of course, use binary search trees. Some issues:
  - Keys are of varying length
  - A lot of strings share similar prefixes (beginnings) – potential for saving space
  - Let's count comparisons of characters.
    - What is the worst-case running time of searching for a string of length m?

Tries

- *Trie* – a data structure for storing a set of strings (name from the word "re*trie*val"):
  - Let's assume, all strings end with "$" (not in Σ)



Set of strings: {**bear, bid, bulk, bull, sun, sunday**}

Tries II

- Properties of a ***trie***:
  - A multi-way tree.
  - Each ***node*** has from 1 to *d* children.
  - Each ***edge*** of the tree is labeled with a character.
  - Each ***leaf*** node corresponds to the stored string, which is a concatenation of characters on a path from the root to this node.

Search and Insertion in Tries

```
Trie-Search(t, P[k..m])  //inserts string P into t
if t is leaf then return true
else if t.child(P[k])=nil then return false
     else return Trie-Search(t.child(P[k]), P[k+1..m])
```

- The search algorithm just follows the path down the tree (starting with Trie-Search(*root*, *P*[0..*m*]))

```
Trie-Insert(t, P[k..m])
if t is not leaf then  //otherwise P is already present
   if t.child(P[k])=nil then
        Create a new child of t and a "branch" starting
        with that chlid and storing P[k..m]
   else Trie-Insert(t.child(P[k]), P[k+1..m])
```

- How would the delete work?

Trie Node Structure

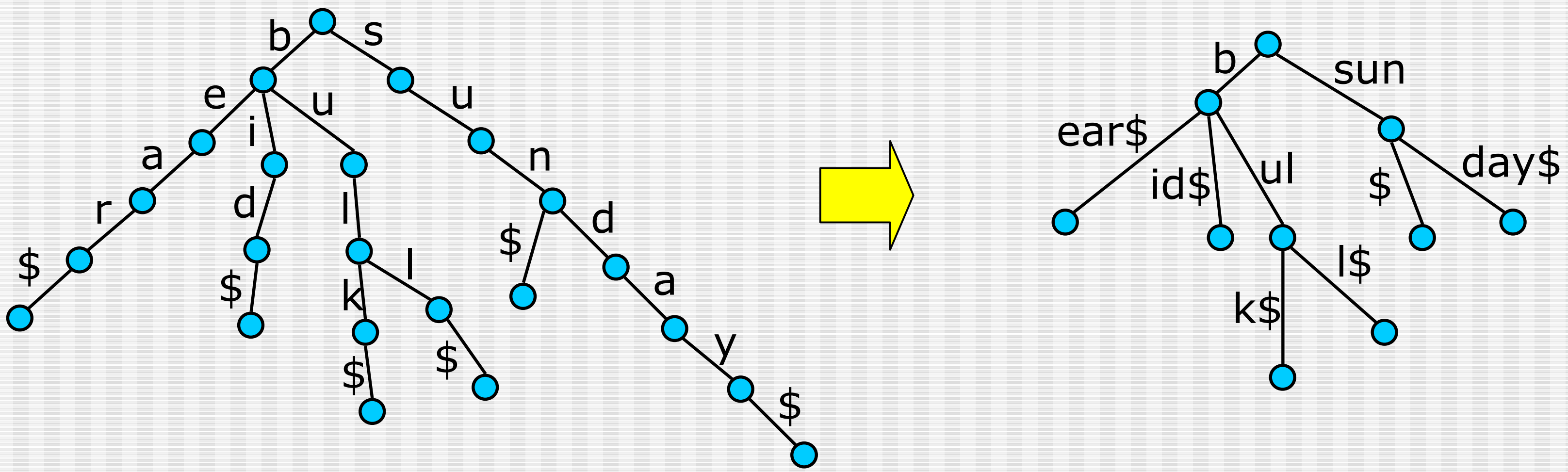
- "Implementation detail"
  - What is the node structure? = What is the complexity of the t.*child*(c) operation?:
    - An **array** of child pointers of size *d*: waist of space, but *child*(c) is $O(1)$
    - A **hash table** of child pointers: less waist of space, *child*(c) is expected $O(1)$
    - A **list** of child pointers: compact, but *child*(c) is $O(d)$ in the worst-case
    - A **binary search tree** of child pointers: compact and *child*(c) is $O(\lg d)$ in the worst-case

Analysis of the Trie

- Size:
  - $O(N)$ in the worst-case
- Search, insertion, and deletion (string of length m):
  - depending on the node structure: $O(dm)$, $O(m \lg d)$, $O(m)$
  - Compare with the string BST
- Observation:
  - Having chains of one-child nodes is wasteful

Compact Tries

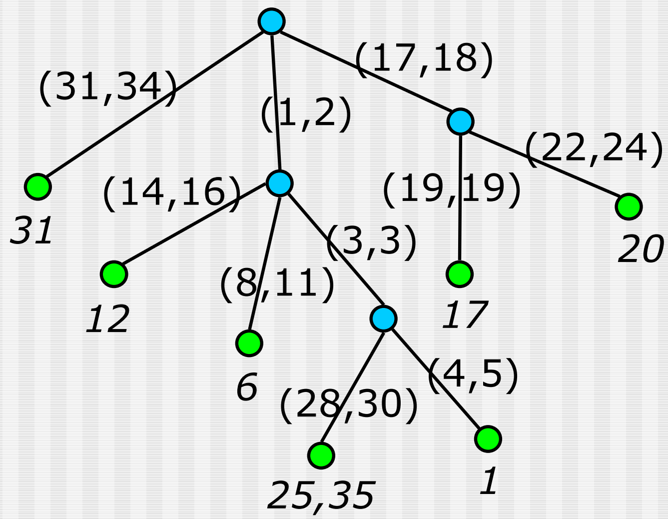
- *Compact Trie*:
  - Replace a *chain* of one-child nodes with an edge labeled with a string
  - Each non-leaf node (except root) has at least two children



Compact Tries II

- Implementation:
  - Strings are external to the structure in one array, edges are labeled with indices in the array (*from*, *to*)
- Can be used to do *word matching*: find where the given word appears in the text.
  - Use the compact trie to “store” all words in the text
  - Each child in the compact trie has a list of indices in the text where the corresponding word appears.

Word Matching with Tries



1 2 3 4 5 6 7 8 9 10 12 14 16 18 20 22 24 26 28 30 32 34 36 38 40

T: `they think that we were there and there`

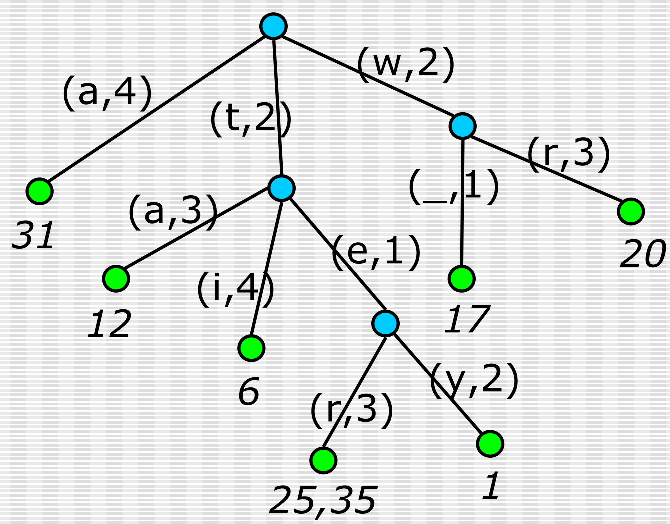
- To find a word *P*:
  - At each node, follow edge (*i*,*j*), such that *P*[*i*..*j*] = T[i..j]
  - If there is no such edge, there is no *P* in *T*, otherwise, find all starting indices of *P* when a leaf is reached

Word Matching with Tries II

- Building of a compact trie for a given text:
  - How do you do that? Describe the compact trie insertion procedure
  - Running time: $O(N)$
- Complexity of word matching: $O(m)$
- What if the text is in external memory?
  - In the worst-case we do $O(m)$ I/O operations just to access single characters in the text – not efficient

Patricia trie

- *Patricia trie*:
  - a compact trie where each edge's label (*from*, *to*) is replaced by (*T*[*from*], *to* – *from* + *1*)



1 2 3 4 5 6 7 8 9 10 12 14 16 18 20 22 24 26 28 30 32 34 36 38 40

T: `they think that we were there and there `

Querying Patricia Trie

- *Word prefix* query: find all words in *T*, which start with *P*[0..*m*-1]

```
Patricia-Search(t, P, k)      // inserts P into t
01 if t is leaf then
02    j ← the first index in the t.list
03    if T[j..j+m-1] = P[0..m-1] then
04       return t.list    // exact match
05 else if there is a child-edge (P[k],s) then
06         if k + s < m then
07            return Patricia-Search(t.child(P[k]), P, k+s)
08         else go to any descendent leaf of t and do the
               check of line 03, if it is true, return
               lists of all descendent leafs of t,
               otherwise return nil
09      else return nil     // nothing is found
```

Analysis of the Patricia Trie

- Idea of patricia trie – postpone the actual comparison with the text to the end:
  - If the text is in external memory only O(1) I/O are performed (if the trie fits in main-memory)
- Build a Patricia Trie for word matching:

1 2 3 4 5 6 7 8 9 10 12 14 16 18 20 22 24 26 28 30 32 34 36 38 40

T: **føtex har haft en fødselsdag**

- Usually binary patricia tries are used:
  - Consider binary encoding of text (and queries)
  - Each node in the tree has two children (left for 0, right for 1)
  - Edges are labeled just with skip values (in bits)

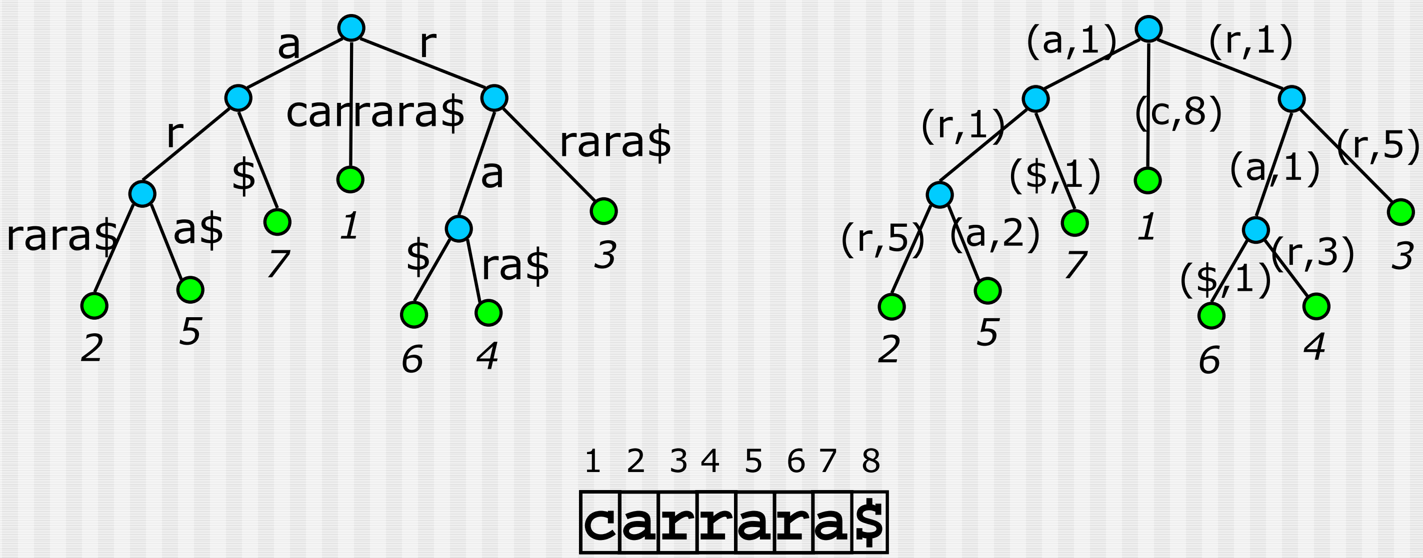
Text-Search Problem

- Input:
  - *Text T* = "**`carrara`**"
  - *Pattern P* = "**`ar`**"
- Output:
  - All occurrences of *P* in *T*
- Reformulate the problem:
  - *Find all suffixes of T that has P as a prefix!*
  - We already saw how to do a *word prefix* query.

```
carrara
 arrara
  rrara
   rara
    ara
     ra
      a
```

Suffix Trees

- *Suffix tree* – a compact trie (or similar structure) of all suffixes of the text
  - Patricia trie of suffixes is sometimes called a *Pat tree*



Pat Trees: Analysis

- Text search for P is then a prefix query.
  - Running time: $O(m+z)$, where z is the number of answers
  - Just $O(1)$ I/Os if the text is in external-memory (independent of z)!
- The size of the Pat tree: $O(N)$
  - Why?
  - Advantage of compression: the size of the simple trie of suffixes would be in the worst-case $N + (N-1) + (N-2) + \ldots 1 = O(N^2)$

Constructing Suffix Trees

- The naïve algorithm
  - Insert all suffixes one after another: $O(N^2)$
- Clever algorithms: $O(N)$
  - McCreight, Ukkonen
  - Scan the text from left to right, use additional suffix links in the tree
- *Question*: *How does the the Pat tree looks like after inserting the first five prefixes using the naïve algorithm*?

1	2	3	4	5	6	7	8	9
H	o	n	o	l	u	l	u	$

Full-Text Indices

- What if the Pat tree does not fit in main memory?
- A number of external-memory data structures were proposed:
  - SPat arrays
  - String B-trees
- String B-tree:
  - A B-tree for strings, i.e., supports dictionary operations
  - Can be used for text-searching if all suffixes are stored in it

String B-tree

- Rough idea:
  - Text is external to the tree, strings are represented in the B^+-tree by the indices of where they begin in the text
    - This would mean doing $O(lg\ B)$ I/Os when visiting each node – too much!
  - Idea – organize all keys in each node into a Patricia trie. When searching this trie (without any I/Os):
    - We reach a leaf. What then?
    - We stop in the middle. What then?
  - The total running time of text search:
    - $O((m+z)/B + \log_B N)$