A short course in data structures analysis

University of Torino, June 2008

Javier Campos

University of Zaragoza (Spain)

Outline

A short course in data structures ***analysis****…*

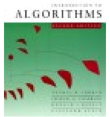
- Worst case analysis:
  - *Red-black trees*
- Average case analysis:
  - *Lexicographical trees*
  - *Skip lists*
- Amortized analysis
  - *Splay trees*
- Data structures for computational biology
  - *Suffix trees*

➡ … it is also an intermediate course in ***dictionary*** **abstract data type**

Basic bibliography

- On algorithms (and data structures):

  [CLRS01]



  Cormen, T.H.; Leiserson, C.E.; Rivest, R.L., Stein, C.:
  Introduction to Algorithms (Second edition),
  The MIT Press, 2001.

- On data structures (and algorithms):

  [MS05]

  Mehta, D.P. and Sahni, S.:
  Handbook of Data Structures and Applications,
  Chapman & Hall/CRC, 2005.

- On… everything:

  [Knu73]

  Knuth, D.E.: *The Art of Computer Programming: Sorting and Searching (vol. 3)*, Addison-Wesley, 1973.

- Additional material (bibliography, papers, applets…):

  http://webdiis.unizar.es/asignaturas/TAP/

Dictionary ADT

- Abstract Data Type (ADT):
  - Definition of a data type independently of concrete implementations
  - Specify the set of data and the set of operations that can be performed on the data
  - Formal definition (algebraic specification)

- Dictionary ADT:
  - An abstract data type storing items, or values. A value is accessed by an associated key. Basic operations are new, insert, find and delete.

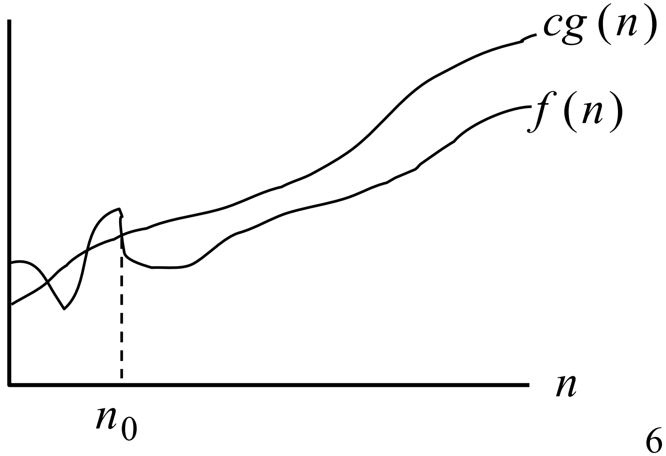
Worst case, best case, average case

- Best, worst and average cases of a given algorithm express what the resource usage is *at least*, *at most* and *on average*, respectively. Usually the resource being considered is **running time**, but it could also be memory or other resource (like number of processors).

- In real-time computing, the worst-case execution time is often of particular concern since it is important to know how much time might be needed *in the worst case* to guarantee that the algorithm would always finish on time.

Asymptotic notation

- Running time is expressed as a funtion in terms of a measure of the problem size (size of input data): $T(n)$.
- In the general case, we have no *a priori* knowledge of the problem size. But, if it can be shown that $T_A(n) \leq T_B(n)$ for all n ≥ 0, then algorithm A is better than algorithm B regardless of the problem size.
- Usually functions $T(n)$ that we obtain are strange and difficult to compare. Then we study their asymptotic behaviour (very large problem sizes), compared with the asymptotic behaviour of well-known functions like linear, logarithmic, quadratic, exponential, etc).
- Big Oh notation:
  - Consider a function $f(n)$ which is non-negative for all integers $n \geq 0$. We say that "$f(n)$ is big oh $g(n)$", which we write $f(n) = O(g(n))$, if there exists an integer n_0 and a constant $c>0$ such that for all integers $n \geq n_0$, $f(n) \leq cg(n)$.



Outline

A short course in data structures analysis…

- Worst case analysis:
  - *Red-black trees*
- Average case analysis:
  - Lexicographical trees: *tries & Patricia*
  - *Skip lists*
- Amortized analysis
  - *Splay trees*
- Data structures for computational biology
  - *Suffix trees*

➡ … and also an intermediate course in *dictionary* abstract data type

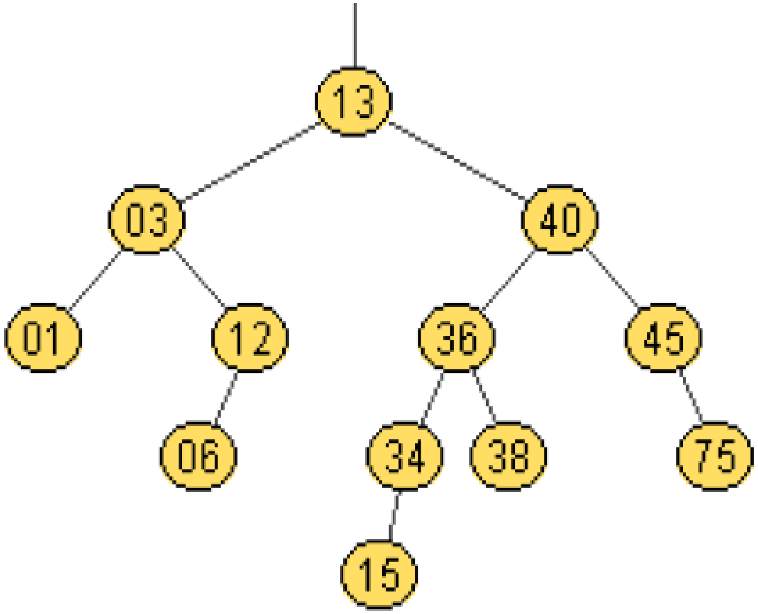
Red-black trees

- What is used for?

They are a type of “balanced” binary search trees (maximum height = logarithmic order on # nodes) to guarantee a worst case cost in $O(\log n)$ for the basic dictionary operations (find, insert, delete).

Remember:

- binary search trees (BST)
- AVL (balanced BST)



Red-black trees

- Definition:
  - Binary Search Tree with an additional bit per node, its colour, that can be either **red** or **black**.
  - Certain **conditions on the node colours** to guarantee that there is no leaf whose depth is more than twice the depth of any other leaf (the tree is "somehow balanced").
  - Each node is a record with: *colour* (red or black), *key* (for finding), and 3 *pointers* to *children* (l, r) and *father* (f).

    We will represent NULL pointers also as (leaves) nodes (to simplify presentation).

Red-black trees

- Red-black conditions:

  RB_1 – Every node is either red or black.

  RB_2 – Every leaf (NULL pointer) is black.

  RB_3 – If a node is red, both children are black.

  (can’t have 2 consecutive reds on a path)

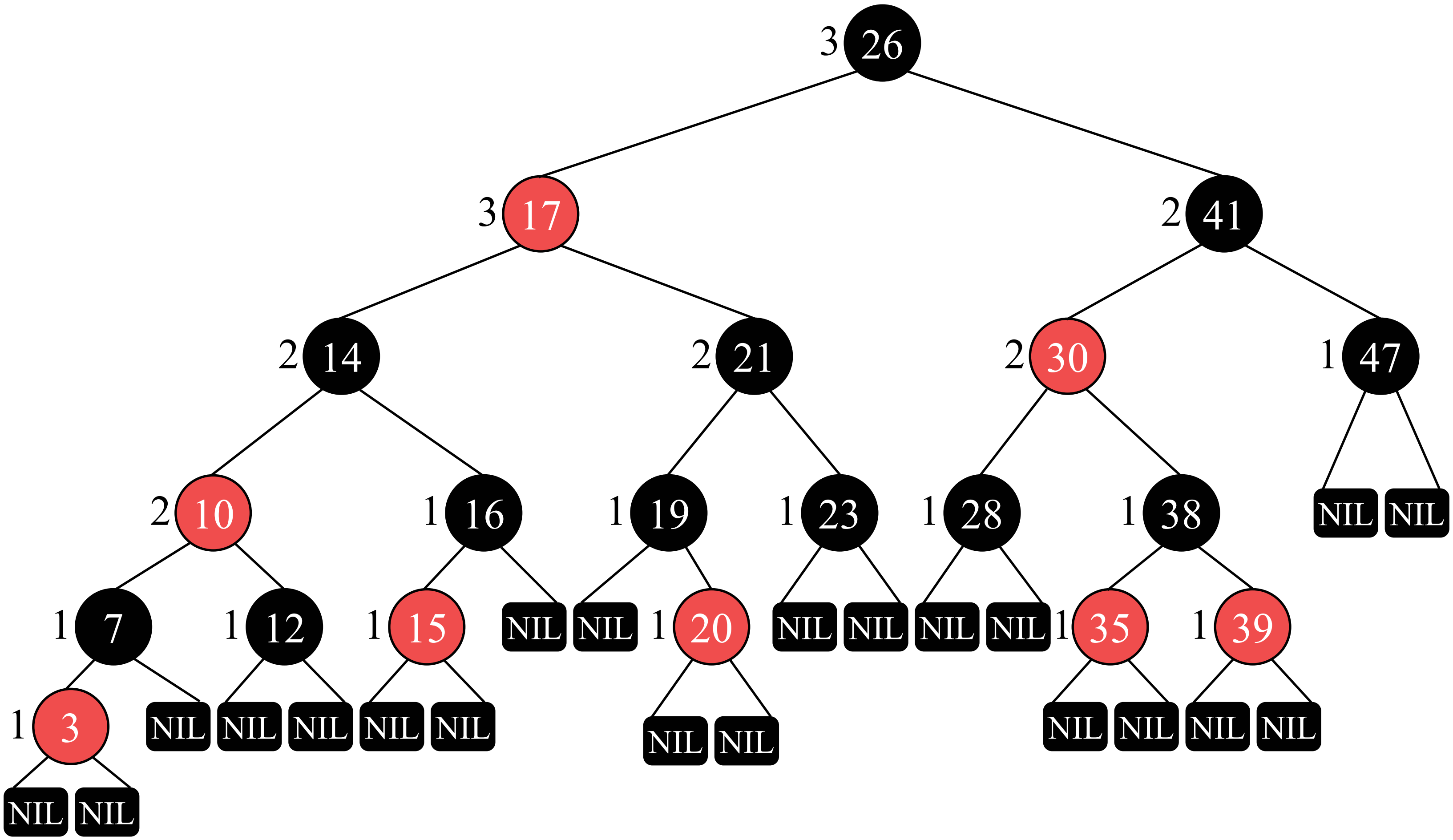
  RB_4 – Every path from node to descendent leaf contains the same number of black nodes.

  (every “real” node has 2 children)

  [RB_5 – The root is always black. → **not necessary]**

- Terminology:
  - *Black-height of node x, bh*(x): # black nodes on path to leaf (excluding x).
  - *Black-height of a tree*: *black-height of its root*.

Red-black trees



Red-black trees

- Lemma: The height of a red-black tree with n internal nodes is less than or equal to 2 log(n+1).

  Proof:

  – The subtree with root x has at least $2^{bh(x)}$-1 internal nodes. By induction on the height of x, $h(x)$:

    - Case $h(x)$=0: x is a leaf (NULL), and its subtree has $2^{bh(x)}-1 = 2^0-1 = 0$ internal nodes.
    - Induction step: consider an internal node x with 2 children; its children have black-height $bh(x)$ ó $bh(x)$-1, depending on its colour (red or black, respectively).

      By induction hypothesis, the 2 children of x have at least $2^{bh(x)-1}$-1 internal nodes. Thus the subtree with root x has at least $(2^{bh(x)-1}-1)+(2^{bh(x)-1}-1)+1= 2^{bh(x)}-1$ internal nodes.

Red-black trees

Proof (cont.):

- [The subtree with root *x* has at least $2^{bh(x)}$-1 internal nodes (already proved).]
- Let *h* be the height of the tree.
  By definition (RB_3: *If a node is red, both children are black*), at least half of the nodes in a path from the root to a leaf (excluding the root) must be black.
  Then, the black-height of the root is at least *h*/2, thus:

$$n \geq 2^{h/2}-1 \quad \Rightarrow \quad \log(n+1) \geq h/2 \quad \Rightarrow \quad h \leq 2 \log(n+1).$$

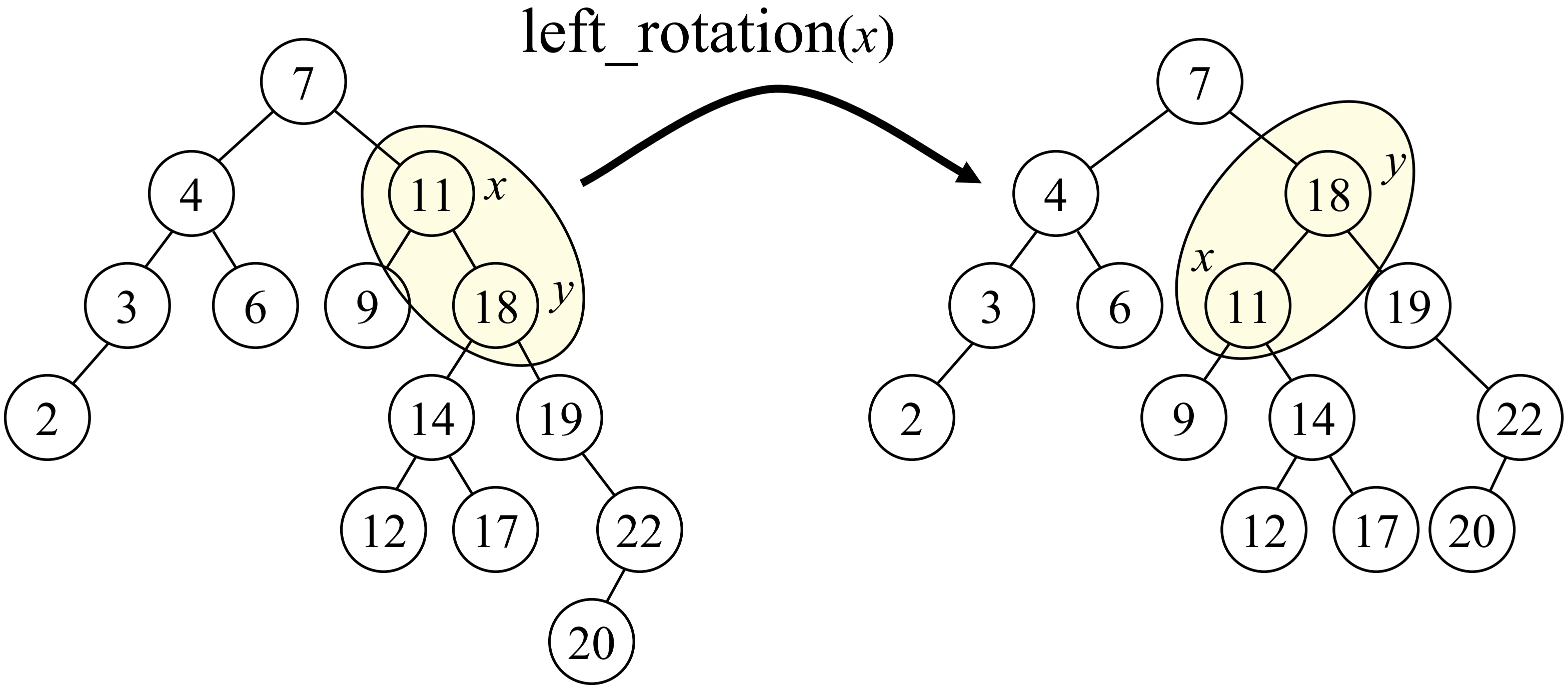
Red-black trees

- Consequences of lemma:
  - “Find” operation (of dictionary ADT) can be implemented to have worst-case execution time in $O(\log n)$ for red-black trees with n nodes.
  - Also “Minimum”, “Maximum”, “Successor”, “Predecessor”…

- And insertion? and deletion?
  - In the sequel we see that insertion and deletion can be also implemented to have worst-case execution time in $O(\log n)$ preserving red-blackness of the tree.

Red-black trees

- Rotations:
  - Local changes of the binary search tree structure that preserve the same data and the “search property”.

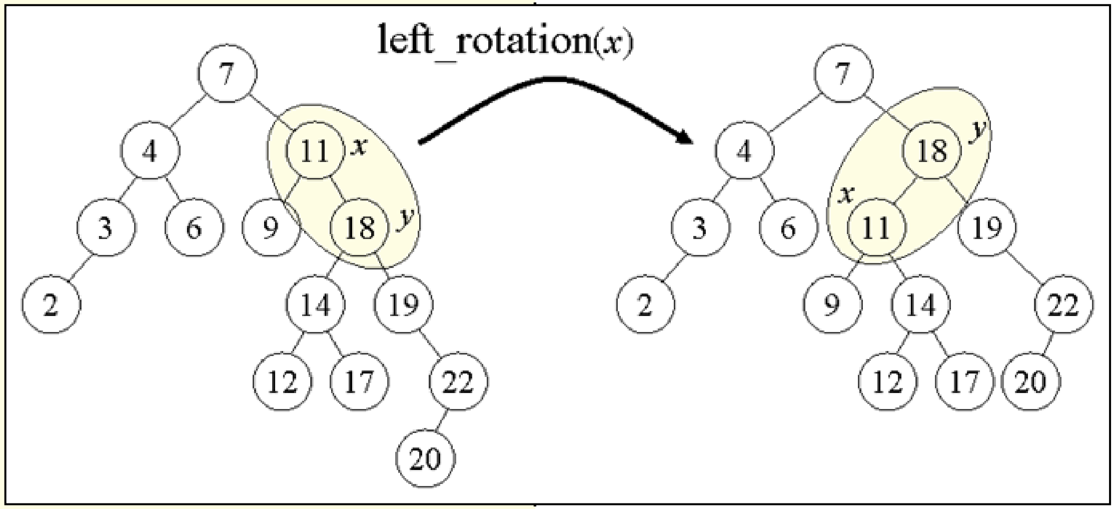


left_rotation(x)

Red-black trees

```
algorithm left_rot(A,x)
begin
  y:=r(x);
  r(x):=l(y);
  if l(y)≠NULL then p(l(y)):=x fi;
  p(y):=p(x);
  if p(x)=NULL then
    root(A):=y
  else
    if x=l(p(x)) then
      l(p(x)):=y
    else
      r(p(x)):=y
    fi
  fi;
  l(y):=x;
  p(x):=y
end
```

(Right rotation is symmetric)



Red-black trees

- Insertion:
  - First a usual insertion in binary search tree is done (ignoring the colour of nodes).
  - If after insertion any of the $RB_{1\text{-}5}$ conditions is violated, the structure of the tree must be modified using rotations changing the colour of nodes.
  - We will not worry about RB_5, it will be trivially preserved.

Insertion

```
   algorithm insert(A,x)
   begin
 1   insert_bst(A,x);
 2   colour(x):=red;
 3   while x≠root(A) and colour(p(x))=red do
 4     if p(x)=l(p(p(x))) then
 5       y:=r(p(p(x)));
 6       if colour(y)=red then
 7         colour(p(x)):=black;
 8         colour(y):=black;          Case 1
 9         colour(p(p(x))):=red;
10         x:=p(p(x))
                                            Case 2
11       else
12         if x=r(p(x)) then x:=p(x); left_rot(A,x) fi;
13         colour(p(x)):=black;
14         colour(p(p(x))):=red;        Case 3
15         right_rot(A,p(p(x)))
16       fi
17     else [the same, changing l/r]
18     fi
19   od;                           Cases 4, 5 & 6
20   colour(root(A)):=black
   end
```

1º) what is broken in red-black tree def. in lines 1-2?

2º) what is the goal of loop 3-19?

3º) what we do in each case?

Insertion

```
   algorithm insert(A,x)
   begin
 1   insert_bst(A,x);
 2   colour(x):=red;
 3   while x≠root(A) and colour(p(x))=red do
 4     if p(x)=l(p(p(x))) then
 5       y:=r(p(p(x)));
 6       if colour(y)=red then
 7         colour(p(x)):=black;
 8         colour(y):=black;
 9         colour(p(p(x))):=red;
10         x:=p(p(x))
11       else
12         if x=r(p(x)) then x:=p(x); left_rot(A,x) fi;
13         colour(p(x)):=black;
14         colour(p(p(x))):=red;
15         right_rot(A,p(p(x)))
16       fi
17     else [the same, changing l/r]
18     fi
19   od;
20   colour(root(A)):=black
   end
```

1º) what is broken in red-black tree def. in lines 1-2?

✓*RB_1: every node is red or black.*

✓*RB_2: every leaf (NULL) is black.*

✘*RB_3: if a node is red, both children are black.*

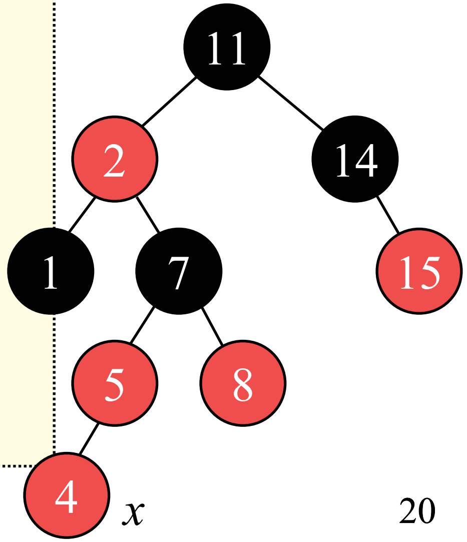
✓*RB_4: every path from node to descendent leaf contains the same number of black nodes.*

```
   algorithm insert(A,x)
   begin
 1   insert_bst(A,x);
 2   colour(x):=red;
 3   while x≠root(A) and colour(p(x))=red do
 4     if p(x)=l(p(p(x))) then
 5       y:=r(p(p(x)));
 6       if colour(y)=red then
 7         colour(p(x)):=black;
 8         colour(y):=black;
 9         colour(p(p(x))):=red;
10         x:=p(p(x))
11       else
12         if x=r(p(x)) then x:=p(x); left_rot(A,x) fi;
13         colour(p(x)):=black;
14         colour(p(p(x))):=red;
15         right_rot(A,p(p(x)))
16       fi
17     else [the same, changing l/r]
18     fi
19   od;
20   colour(root(A)):=black
   end
```

Insertion

2º) what is the goal of loop 3-19?

to solve the above problem, like the case in the figure (insertion of red *x*, as a child of another red), making the red colour move up to the root

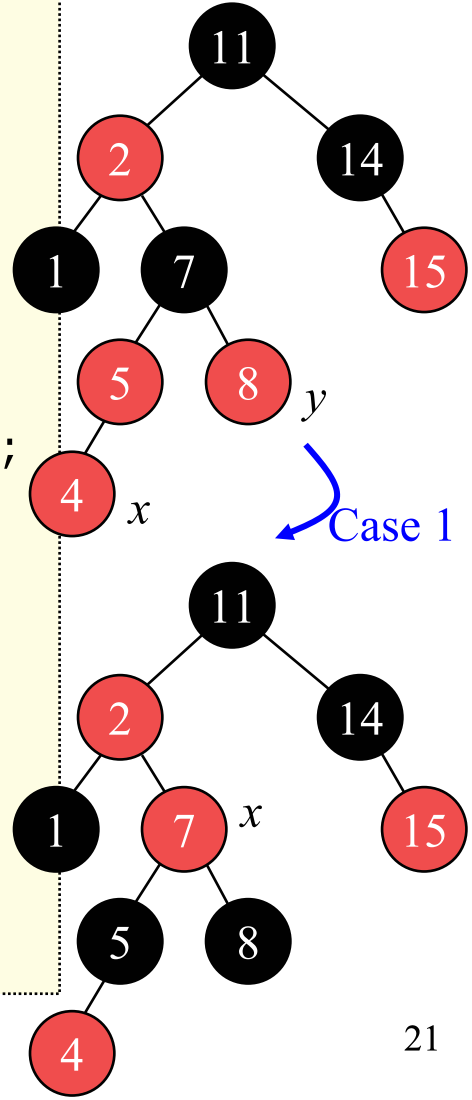


Insertion

3º) what we do in each case?

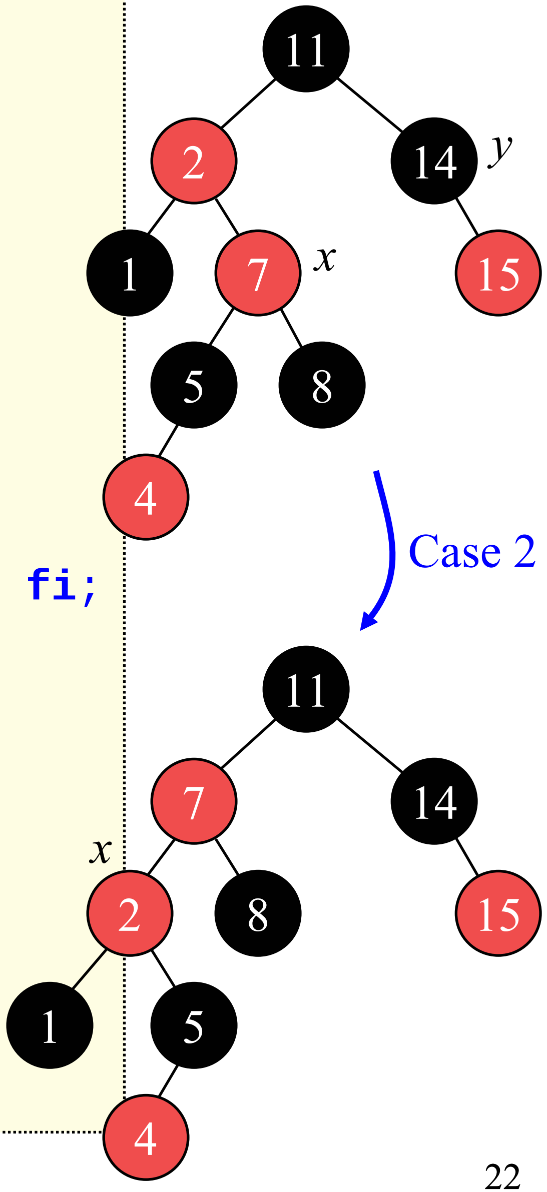
```
algorithm insert(A,x)
begin
  insert_bst(A,x);
  colour(x):=red;
  while x≠root(A) and colour(p(x))=red do
    if p(x)=l(p(p(x))) then
      y:=r(p(p(x)));
      if colour(y)=red then
        colour(p(x)):=black;
        colour(y):=black;
        colour(p(p(x))):=red;
        x:=p(p(x))
      else
        if x=r(p(x)) then x:=p(x); left_rot(A,x) fi;
        colour(p(x)):=black;
        colour(p(p(x))):=red;
        right_rot(A,p(p(x)))
      fi
    else [the same, changing l/r]
    fi
  od;
  colour(root(A)):=black
end
```

Case 1



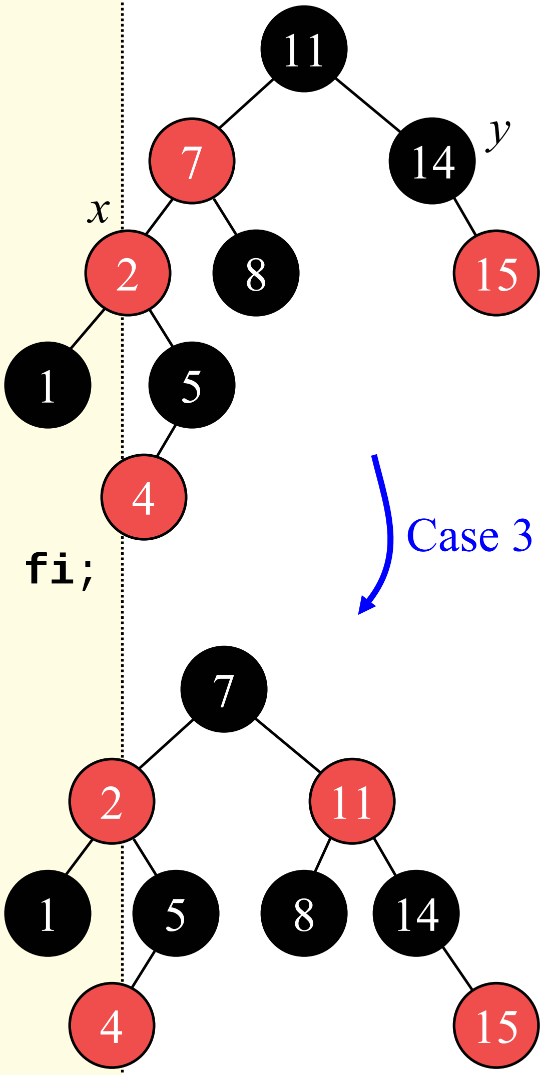
Insertion

```
algorithm insert(A,x)
begin
  insert_bst(A,x);
  colour(x):=red;
  while x≠root(A) and colour(p(x))=red do
    if p(x)=l(p(p(x))) then
      y:=r(p(p(x)));
      if colour(y)=red then
        colour(p(x)):=black;
        colour(y):=black;
        colour(p(p(x))):=red;
        x:=p(p(x))
      else                                 Case 2
        if x=r(p(x)) then x:=p(x); left_rot(A,x) fi;
        colour(p(x)):=black;
        colour(p(p(x))):=red;
        right_rot(A,p(p(x)))
      fi
    else [the same, changing l/r]
    fi
  od;
  colour(root(A)):=black
end
```



Insertion

```
algorithm insert(A,x)
begin
 insert_bst(A,x);
 colour(x):=red;
 while x≠root(A) and colour(p(x))=red do
  if p(x)=l(p(p(x))) then
   y:=r(p(p(x)));
   if colour(y)=red then
    colour(p(x)):=black;
    colour(y):=black;
    colour(p(p(x))):=red;
    x:=p(p(x))
   else
    if x=r(p(x)) then x:=p(x); left_rot(A,x) fi;
    colour(p(x)):=black;     ⎫
    colour(p(p(x))):=red;    ⎬ Case 3
    right_rot(A,p(p(x)))     ⎭
   fi
  else [the same, changing l/r]
  fi
 od;
 colour(root(A)):=black
end
```

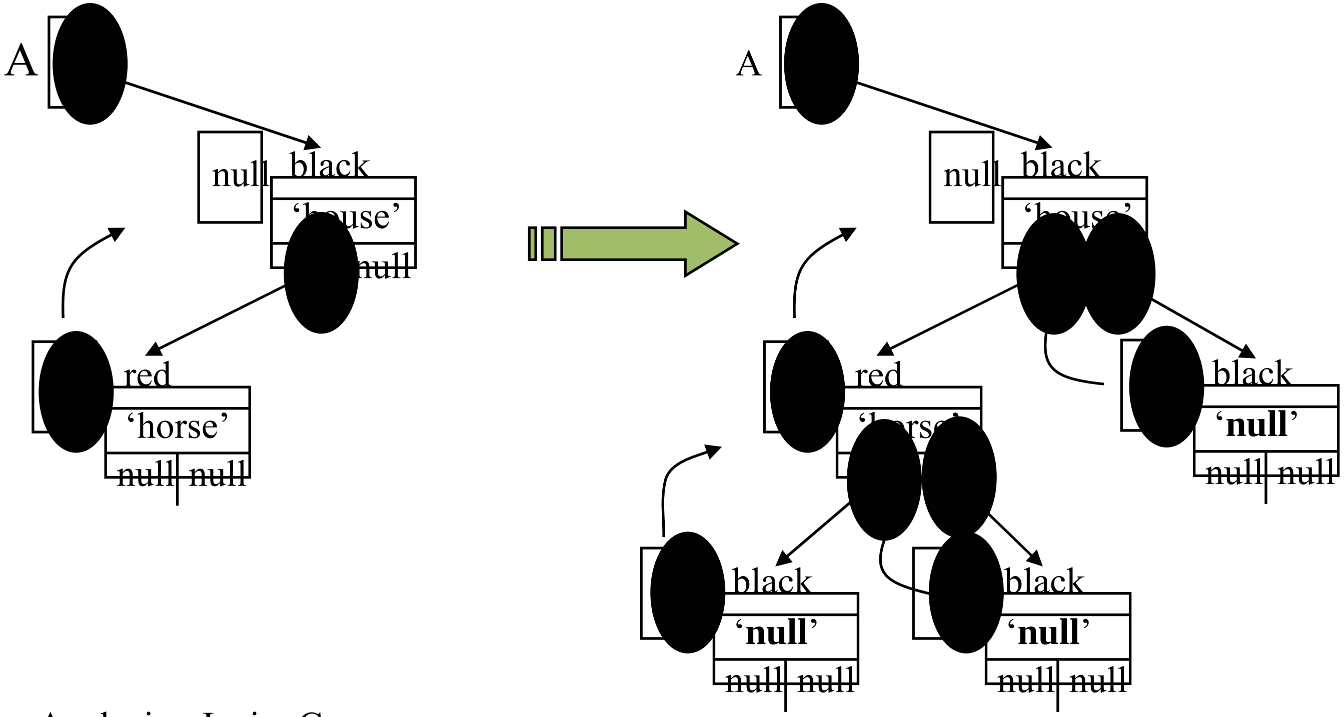


Red-black trees

- Cost of insertion:
  - The height of a red-black tree with *n* nodes is *O*(log *n*), then “insert_bst” is *O*(log *n*).
  - Loop is repeated only if case 1 arises, and in that case pointer *x* goes up in the tree.
    Then, the maximum number of loop iterations is *O*(log *n*).
  - Notice also that more than 2 rotations are never performed
    (the loop ends if case 2 or 3 is run).

Red-black trees

- Deletion:
  - We will see that it can also be done in worst case cost in *O*(log *n*).
  - To simplify the algorithm, we will use the *sentinel* representation for NULL nodes:



Red-black trees

- Deletion: similar to deletion in “bst”

```
algorithm delete(A,z) --z is pointer to node to delete
begin
  if key(l(z))=‘null’ or key(r(z))=‘null’ then
    y:=z
  else
    y:=successor_bst(z)   -- in-order successor of z
  fi;
  if key(l(y))≠‘null’ then x:=l(y) else x:=r(y) fi;
  p(x):=p(y);
  if p(y)=‘null’ then
    root(A):=x
  else
    if y=l(p(y)) then l(p(y)):=x else r(p(y)):=x fi
  fi;
  if y≠z then key(z):=key(y) fi;
  if colour(y)=black then fix_deletion(A,x) fi
end
```

Red-black trees

- Deletion: step by step description

```
 algorithm delete(A,z)  --z is pointer to node to delete
 begin
1  if key(l(z))='null' or key(r(z))='null' then
2    y:=z
3  else
4    y:=successor_bst(z)   -- in-order successor of z
5  fi;
   . . .
```

Lines 1-5: selection of node *y* to put in place of *z*.

Node *y* is:

- The same node *z* (if *z* has at the most 1 child), or
- the successor of *z* (if *z* has 2 children)

Red-black trees

- Deletion: step by step description

```
    . . .
 6  if key(l(y))≠‘null’ then x:=l(y) else x:=r(y) fi;
 7  p(x):=p(y);
 8  if p(y)=‘null’ then
 9    root(A):=x
10  else
11    if y=l(p(y)) then l(p(y)):=x else r(p(y)):=x fi
12  fi;
13  if y≠z then key(z):=key(y) fi;
    . . .
```

Line 6: *x* is child of *y,* or it is NULL if *y* has not children.

Lines 7-12: connection with *y*, modifying pointers in *p*(*y*) and in *x*.

Line 13: if the successor of *z* has been the linked node, the content of *y* is moved to *z*, deleting its previous content (in the algorithm only the key is copied, but if *y* had other fields, they should be also copied).

Red-black trees

- Deletion: step by step description

```
     . . .
14  if colour(y)=black then fix_deletion(A,x) fi
  end
```

Line 14: if *y* is black, “fix_deletion” is used to change the colours and to make rotations needed to restore the properties $RB_{1\text{-}5}$ of red-black tree.

If *y* is red, properties $RB_{1\text{-}5}$ hold (black-height of the nodes did not change and two red nodes were not put adjacent).

Node *x* (argument of the algorithm) was the unique child of *y* before *y* was linked, or it was the sentinel of NULL in the case that *y* had no children.

Red-black trees

- Deletion: fixing properties $RB_{1\text{-}5}$
  - **Problem**: If node *y* in the algorithm was black, after its deletion, all paths passing through it have one black node less, then RB_4 fails

    “Every path from node to descendent leaf contains the same number of black nodes”

    for every ancestor of *y*
  - **Solution**: to interpret that node *x* has an “extra black” colour, then RB_4 “holds”, i.e., when node *y* is deleted, “its blackness is pushed to its child” *x*
  - **New problem**: now *x* can be “twice black”, then RB_1 fails
  - **Solution**: to execute algorithm “fix_deletion” to restore RB_1

Red-black trees

- Deletion: fixing properties $RB_{1\text{-}5}$

```
algorithm fix_deletion(A,x)
begin
  while x≠root(A) and colour(x)=black do
  . . .
  . . .
  od;
  colour(x):=black
end
```

Objective of the loop: to move up the “extra black” until
1) *x* is red, and there is no problem, or
2) *x* is the root, and the “extra black” “desappears” (itself)

Inside the loop, *x* is a black node, different from the root, and with an “extra black”.

Red-black trees

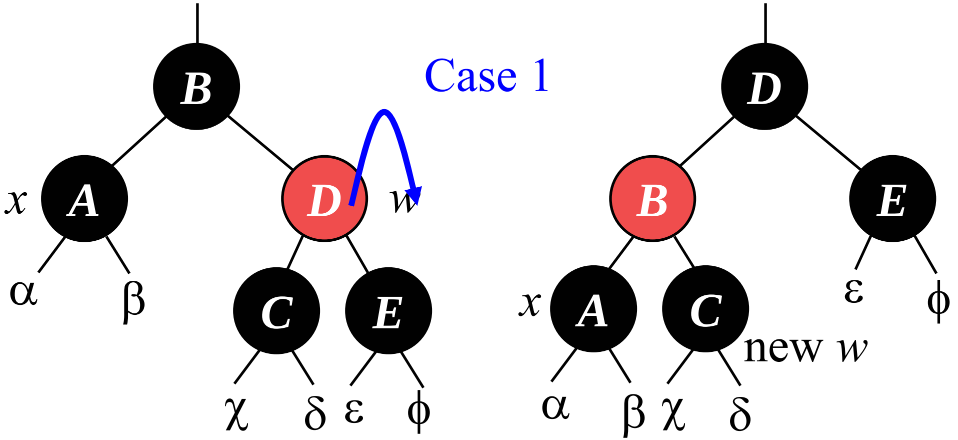
- Inner part of the loop:

```
. . .
 if x=l(p(x)) then
   w:=r(p(x));
   if colour(w)=red then
     colour(w):=black;
     colour(p(x)):=red;
     left_rot(A,p(x));
     w:=r(p(x))
   fi; {goal: to get a black
. . .    brother for x}
```

Case 1

- x is a left child
- w is the brother of x
- x is "double black" ⇒ ⇒ key(w) ≠ NULL (otherwise, # blacks from $p(x)$ to leaf w would be less than # blacks from $p(x)$ to x)

- w must have a black child
- change colour of w and $p(x)$ and exec. "left_rot" with $p(x)$
- the new brother of x, one of the children of w, is black, so we go to another case (2-4)



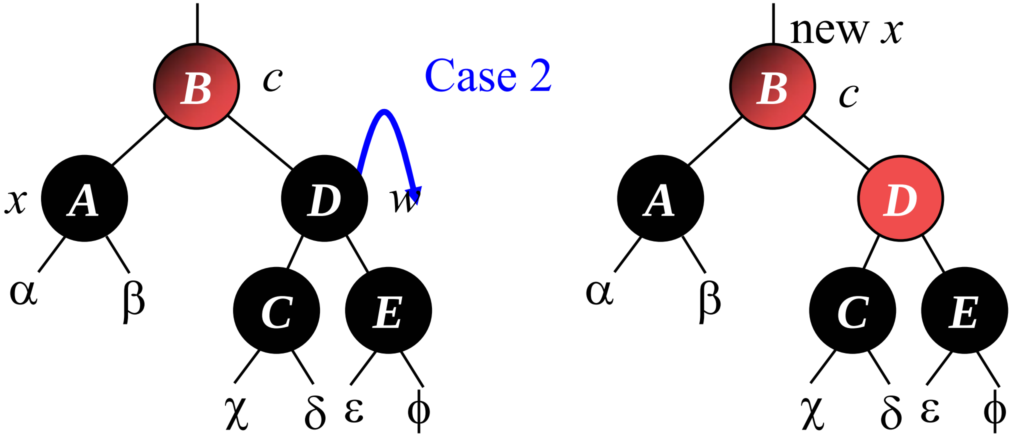
Red-black trees

- Inner part of the loop (cont.):

```
. . .
 if colour(l(w))=black and colour(r(w))=black then
   colour(w):=red;
   x:=p(x)
 else
. . .
```

Case 2

- Node *w* (brother of *x*) and the children of *w* are black.
- We remove a black of *x* and “another” of *w* (*x* is now black and *w* is red).
- We add a black to *p*(*x*).
- Repeat loop with *x*:=*p*(*x*).
- If we entered to this case from case 1, the colour of the new *x* is red and loop ends.

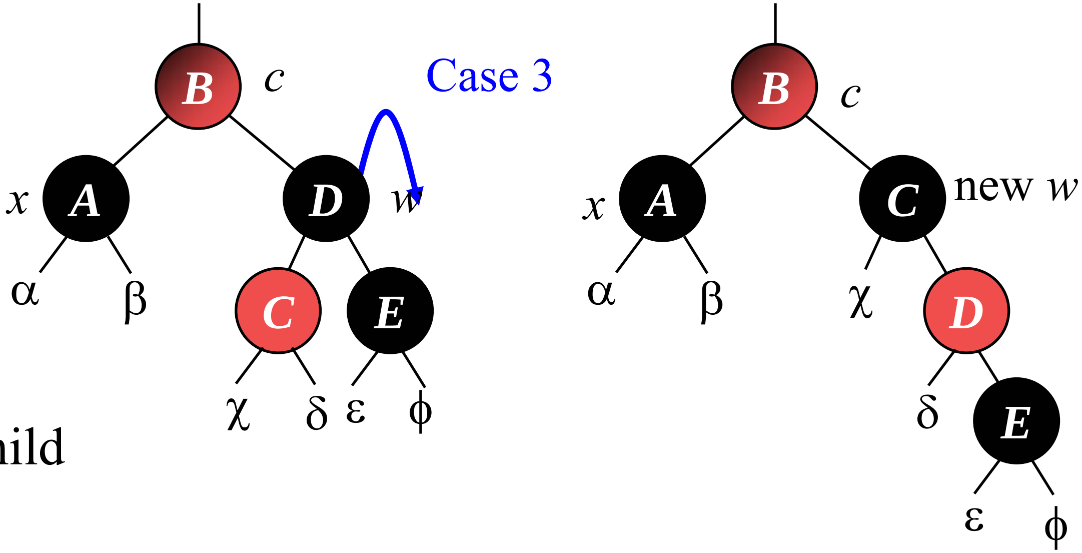


Red-black trees

- Inner part of the loop (cont.):

```
    . . .
        if colour(r(w))=black then
          colour(l(w)):=black;
          colour(w):=red;                  } Case 3
          right_rot(A,w);
          w:=r(p(x))
        fi;
    . . .
```

- *w* and *r*(*w*) are black
- *l*(*w*) is red
- Change the colour of *w* and *l*(*w*) and execute “right_rot” over *w*.
- New brother *w* of *x* is now black with right child red, and this is case 4.



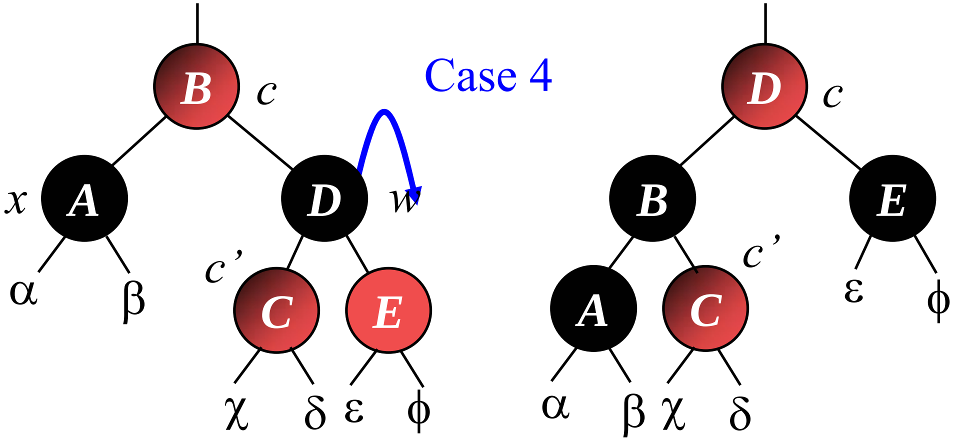
Red-black trees

- Inner part of the loop (cont.):

```
    . . .
20         colour(w):=colour(p(x));
21         colour(p(x)):=black;
22         colour(r(w)):=black;
23         left_rot(A,p(x));
24         x:=root(A)
25       fi
26     else [like 3-25, swapping l/r]
27     fi
   . . .
```

Case 4

- *w* is black and *r*(*w*) is red
- We change some colours and exec. "left_rot" over *p*(*x*), thus the "extra black" of *x* is deleted.



Red-black trees

- Cost of deletion:
  - Cost of algorithm without considering the call to "fix_deletion" is *O*(log *n*), since that is the height of the tree.
  - In "fix_deletion", cases 1, 3 & 4 end after a constant number of colour changes and up to 3 rotations.
  - Case 2 is the only one that can cause reentering the loop, and in that case node *x* is moved up in the tree up to *O*(log *n*) times and without executing rotations.
  - Then, "fix_deletion", and also the complete deletion, are *O*(log *n*) and execute up to 3 rotations.

Outline

A short course in data structures analysis…

- Worst case analysis:
  - *Red-black trees*
- Average case analysis:
  - *Lexicographical trees*
  - *Skip lists*
- Amortized analysis
  - *Splay trees*
- Data structures for computational biology
  - *Suffix trees*

➡ … and also an intermediate course in *dictionary* abstract data type

Lexicographical trees

- *Tries*: motivation…
  - Central letters of the word “retrieval” (*information retrieval* ≈ the science of searching)
  - Pronounced [tri] ("tree"), although some encourage the use of [traɪ] ("try") in order to distinguish it from “tree”.
  - Dictionary of english words in Unix (e.g., *ispell*): 80.000 words and 700.000 characters ➔ more than 8 characters per word in average…

    There is a lot of redundant information:

    bestial bestir bestowal bestseller bestselling
  - To save space:
    group common prefixes
  - To save time:
    shorter words ➔ faster search

```
best
---- i
---- - al
---- - r
---- owal
---- sell
---- ---- er
---- ---- ing
```

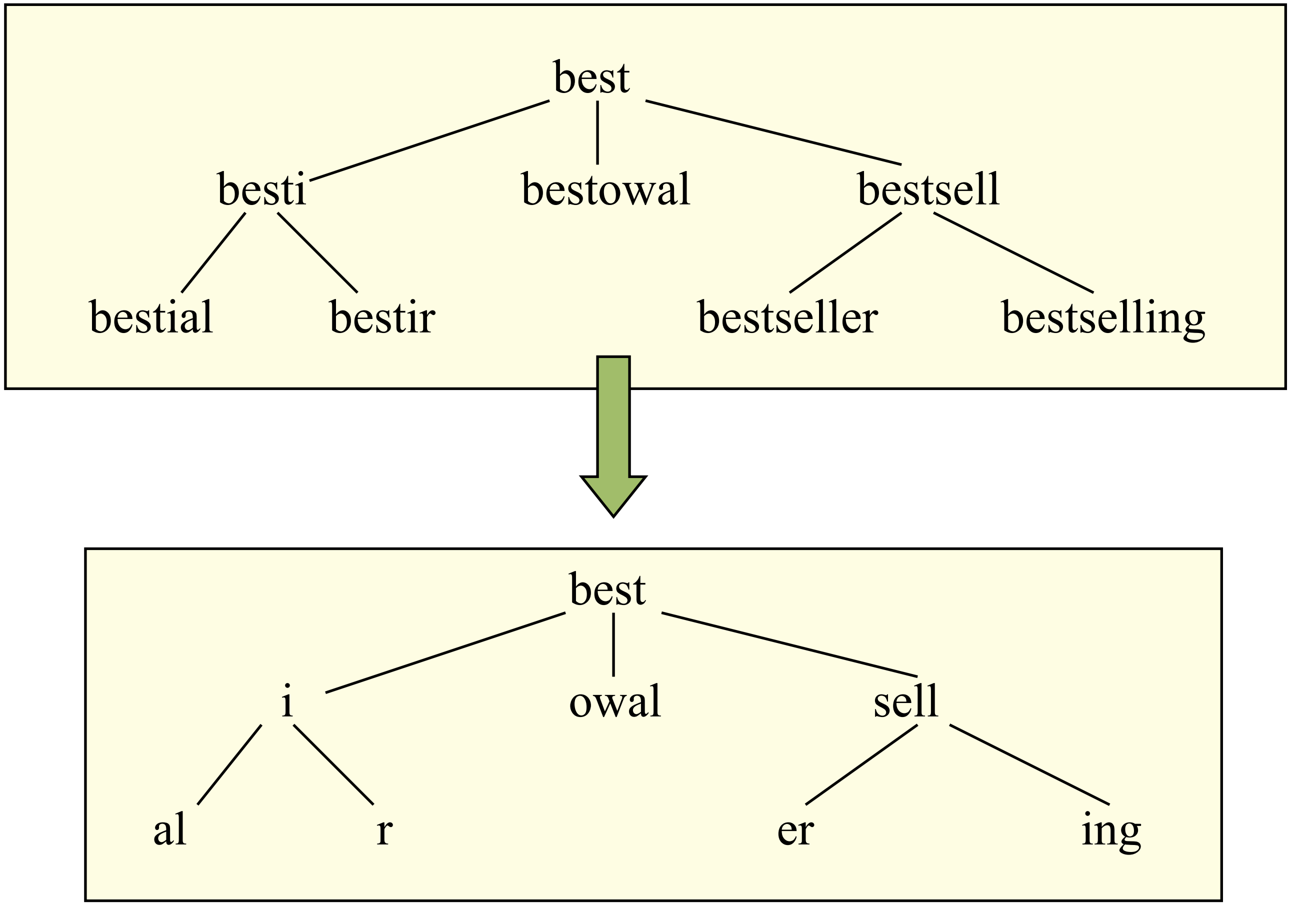
Lexicographical trees

- Trie: formal definition
  - Let $\Sigma=\{\sigma_1, \ldots, \sigma_m\}$ be a finite **alphabet** ($m > 1$).
  - Let Σ^* be the set of **words** (or sequences) of symbols of Σ, and X a subset of Σ^* (i.e., X is a set of words).
  - The **trie** associated to X is:
    - trie(X) = $\varnothing$, if $X = \varnothing$
    - trie(X) = <x>, if $X = \{x\}$
    - trie(X) = <trie($X \setminus \sigma_1$), …, trie($X \setminus \sigma_m$)>, if $|X| > 1$, where $X \setminus \sigma$ represents the subset of all the words in X beginning with σ deleting from them the first letter.
  - If the alphabet includes an order relation of symbols (usual case), the trie is called lexicographical tree.

Lexicographical trees

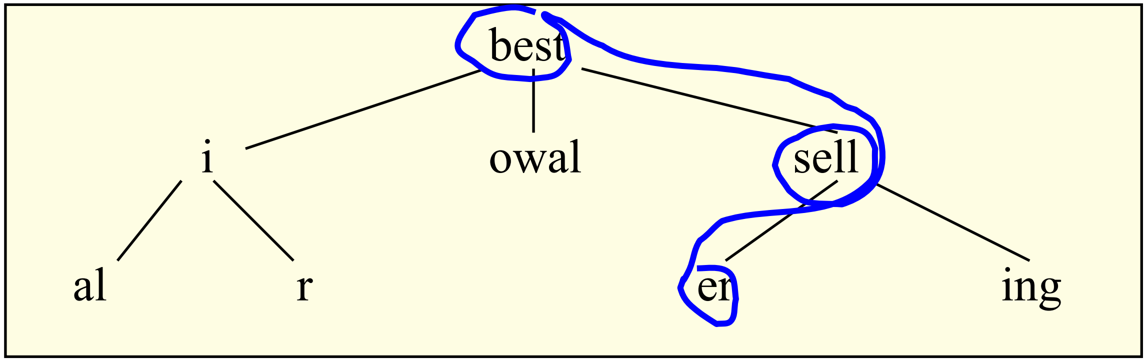
- Then, a trie is *tree of prefixes*:

bestial bestir bestowal bestseller bestselling

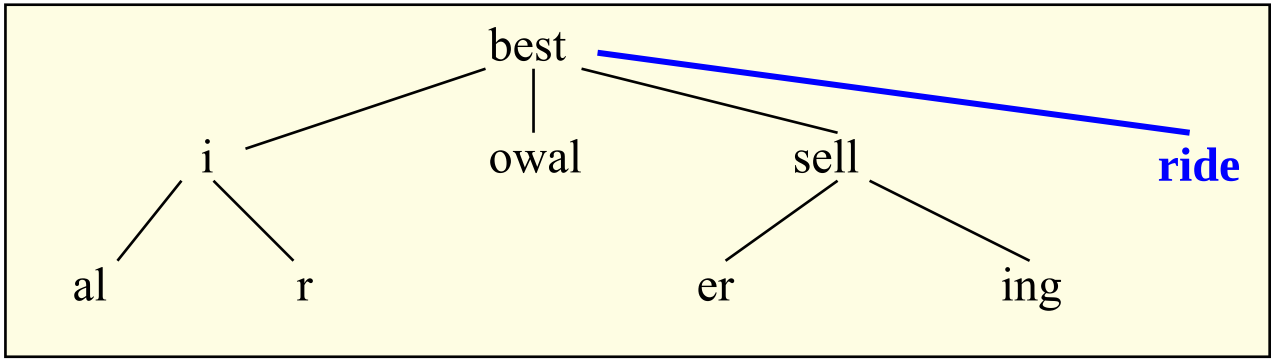


Lexicographical trees

- Utility of trie:
  - Supports operations to search words:



  - Also insertions and deletions can be easily implemented ➔ *dictionary ADT*



Lexicographical trees

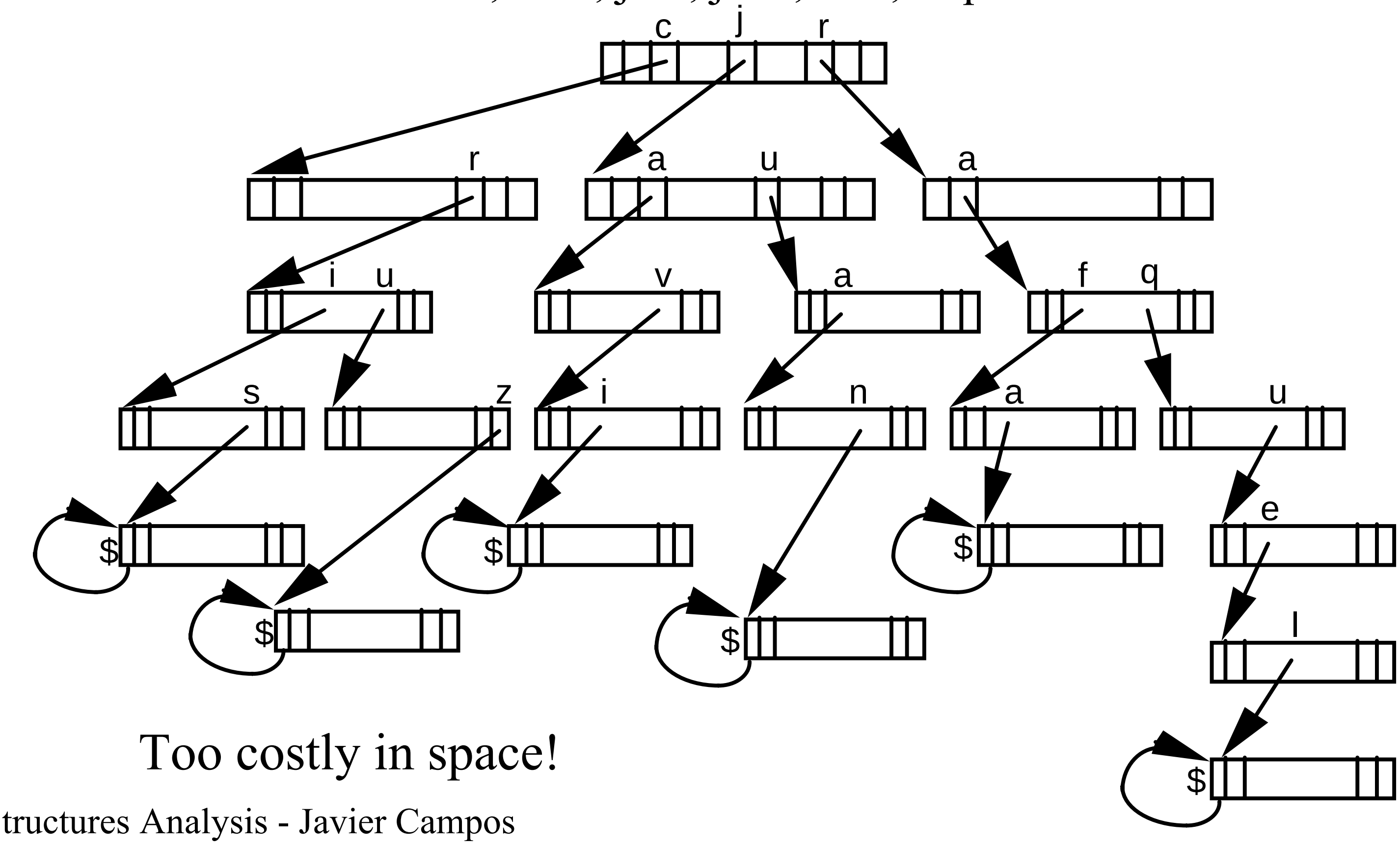
- Also unions and intersections ➔ *set ADT*
- And strings comparisons ➔ texts processing, computational biology…
- A small problem, they cannot contain a word that is a prefix of another contained word… but the problem can be solved if needed by using an ending character

“tries are
one of the most important
general purpose data structures”

Lexicographical trees

- Implementations of tries:
  - *Node-array*: each node is an array of pointers to access to the subtrees

cris, cruz, javi, juan, rafa, raquel

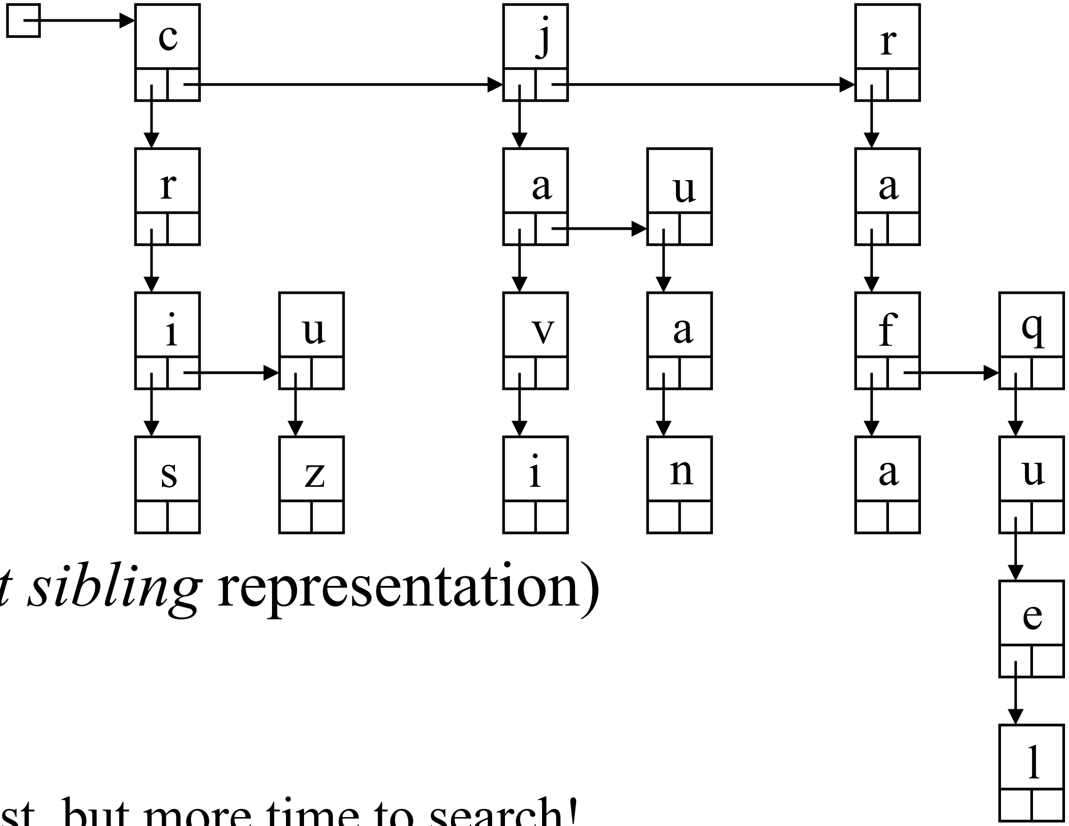


Too costly in space!

Lexicographical trees

– *Node-list*: each node is a linked list (with pointers) containing the roots of subtrees

cris, cruz, javi, juan, rafa, raquel



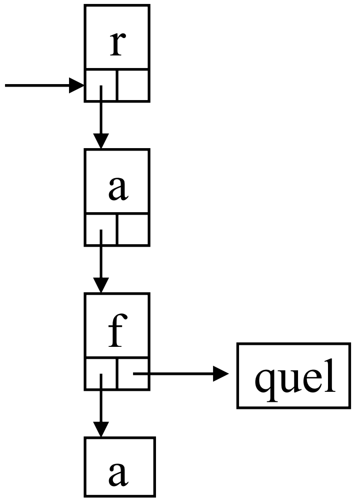
(*child - next sibling* representation)

Less space cost, but more time to search!

Lexicographical trees

– A precision about the previous implementations:

when a certain node is the root of a subtrie containing a single word, that word (suffix) can be directly stored in an external node (thus saving space, even if we are forced to handle pointers to different data types…)



Lexicographical trees

– *Node-BST*: (BST=binary search tree) the structure is also called *ternary search tree*.

Each node contains:

- Two pointers to left and right children (like in BST).
- One pointer, central, to the root of the trie that the node points at.

Goal: to combine the time efficiency of tries with the space efficiency of *BST*'s.

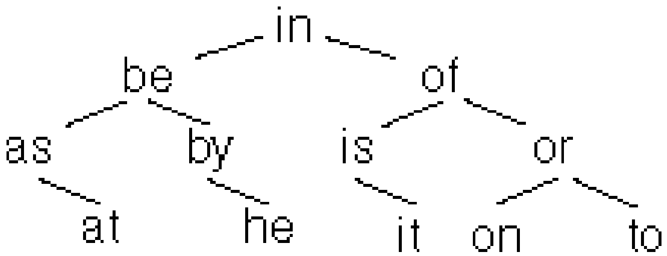
- A search compares the present character in the searched string against the character stored in the node.
- If the searched character is (alphabetically) smaller, the search continues towards the left child.
- If the searched character is bigger, search continues towards the right child.
- If the character is equal, we go towards the central child, and go head searching the next character of the searched string.

Lexicographical trees

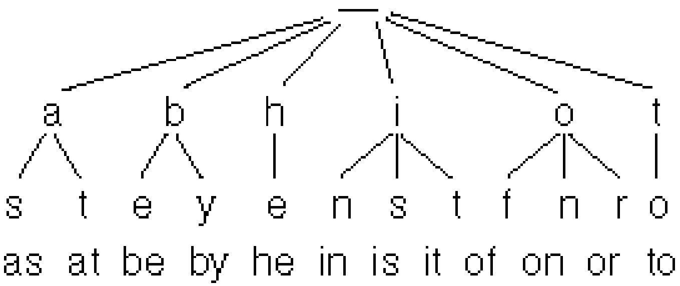
Ternary search tree storing the words…

as at be by he in is it of on or to

In a *BST*:

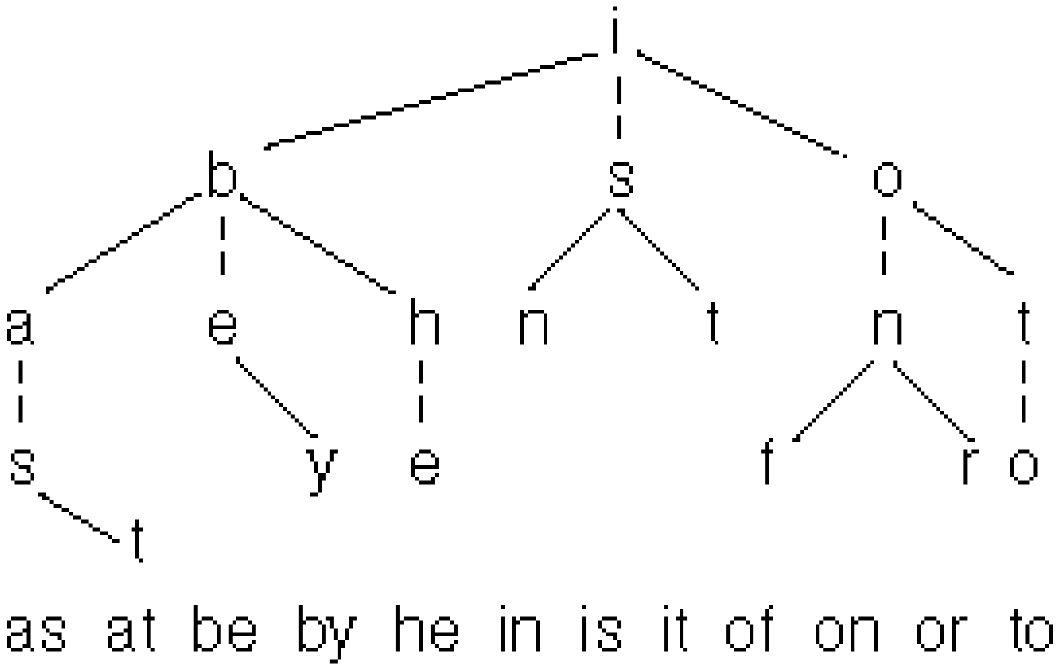


In a trie (representation array or node-list):



Lexicographical trees

In a ternary search tree:



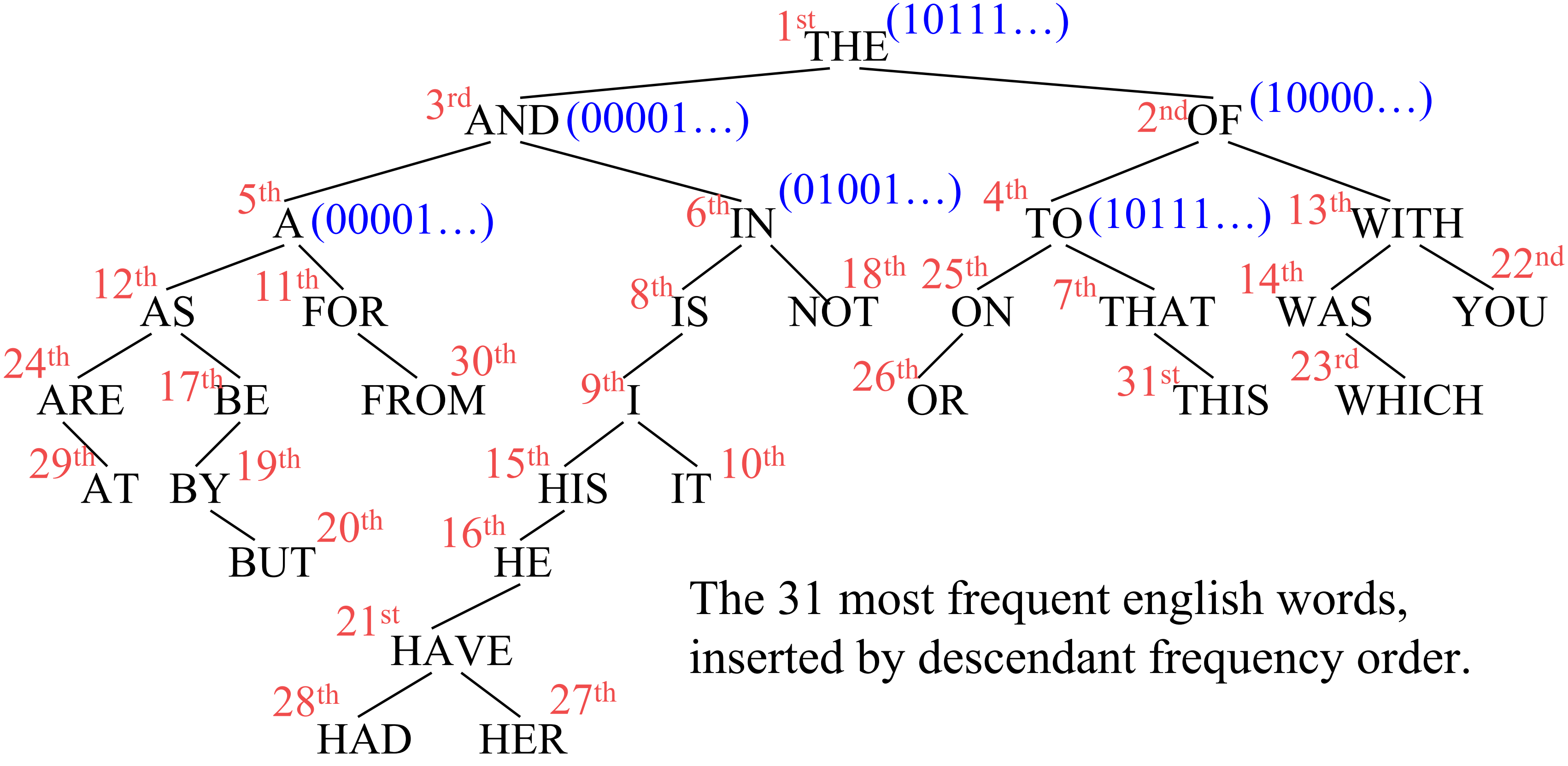
Lexicographical trees

- *Digital search trees*:
  - Binary case (m = 2, i.e., using only 2 symbols):
    - Store complete keys in the nodes, and use their bits to decide following the search towards the left or right subtree.
    - Example, using MIX code (D.E. Knuth)

	0	00000	I	9	01001	R	19	10011
A	1	00001	J	11	01011	S	22	10110
B	2	00010	K	12	01100	T	23	10111
C	3	00011	L	13	01101	U	24	11000
D	4	00100	M	14	01110	V	25	11001
E	5	00101	N	15	01111	W	26	11010
F	6	00110	O	16	10000	X	27	11011
G	7	00111	P	17	10001	Y	28	11100
H	8	01000	Q	18	10010	Z	29	11101

Lexicographical trees

– Digital search trees (binary case):

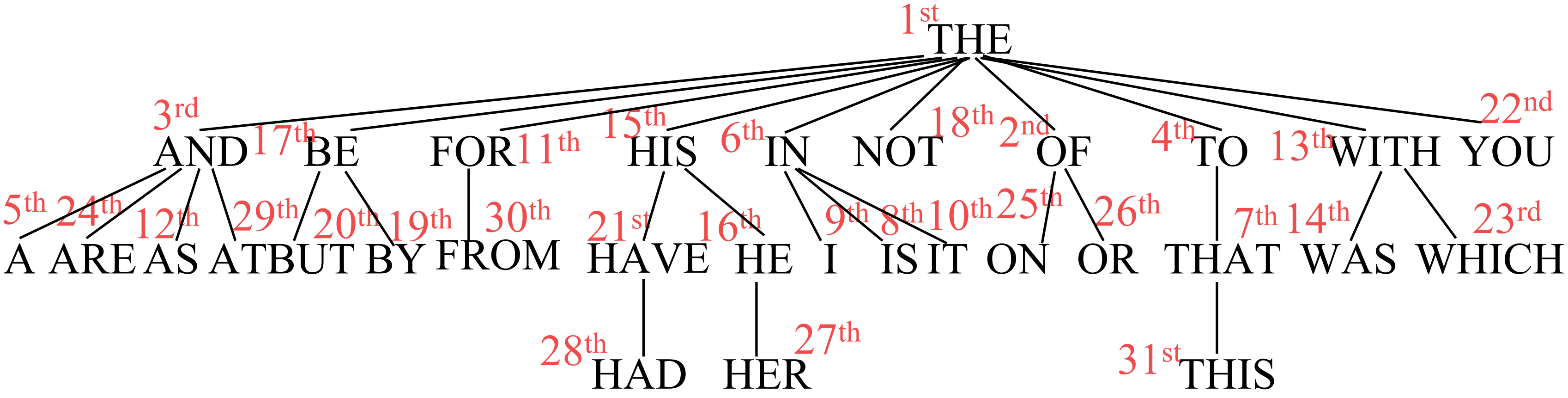


The 31 most frequent english words, inserted by descendant frequency order.

Careful! It is a search tree but considering the binary codification of the keys (MIX code).

Lexicographical trees

- The search in the previous tree is binary, but can be easily extended to *m*-ary (*m* > 2), for an alphabet with *m* symbols.



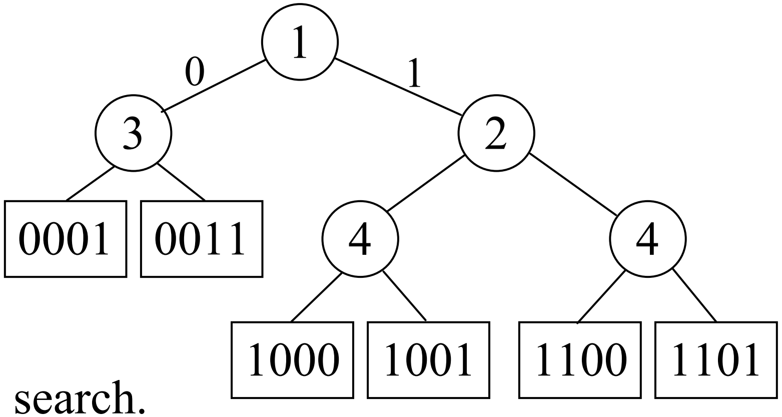
The same keys than before, inserted in the same order, but in a digital search tree of order 26.

Lexicographical trees

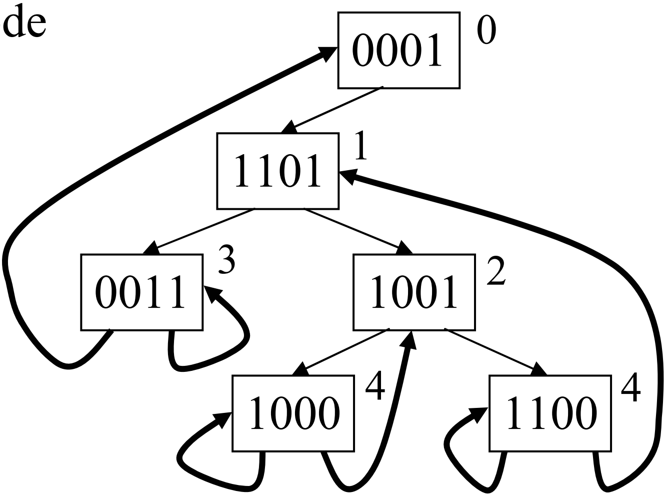
- *Patricia* (*P*ractical *A*lgorithm *T*o *R*etrieve *I*nformation *C*oded *I*n *A*lphanumeric)
  - Problem of tries: if |{keys}| << |{potential keys}|, most of internal nodes in the tree have a single child ➔ space cost grows
  - Idea: binary trie, but avoiding branches with only one direction.
  - *Patricia*: compact representation of a trie where each node with a single child "is joined" with its child.
  - Application example: IP Routing Lookup Algorithms (routing tables in routers, looking for destination address = *longest prefix matching*)

Lexicographical trees

- *Patricia* example:
  - We start from (not a *Patricia*):
    - A binary trie with keys stored in its leaves and compacted (each internal node has 2 children).
    - Label inside internal nodes is the bit used to branch the search.

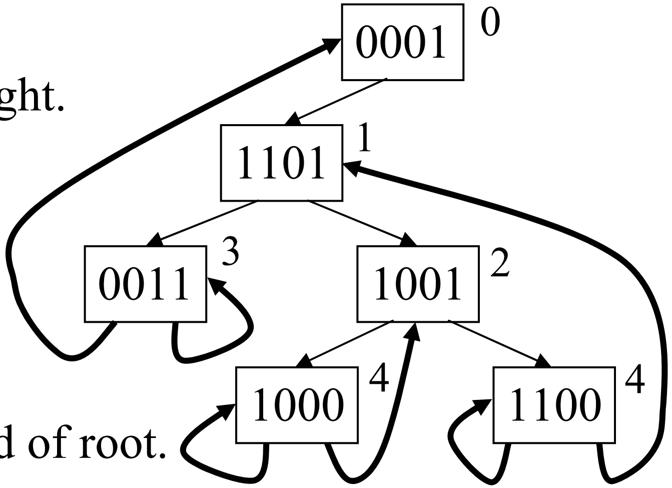


  - In *Patricia*, keys are stored in internal nodes.
    - Since there is one less internal node than # keys, 1 more node is added (the root).
    - Each node still stores the number of bit used to branch.
    - That number distinguishes if pointer goes up or down (if > parent's, down).



Lexicographical trees

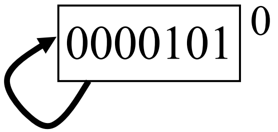
- Searching a key in *Patricia*:
  - Bits of the key are used, from left to right. Going down in the tree.
  - When the followed pointer goes up, the searched key is compared with the key in the node.
  - Example, searching key 1101:
    - We always start going to left child of root.
    - The pointer goes down (we know that because the bit labelling the node 1101, 1, is greater than parent's, 0).
    - Search according to the value of bit 1 of the key, since it is 1, we go to the right child (1001).
    - The number of bit in the reached node is 2, we go down according to that bit, since the 2nd bit of the searched key is 1, we go to the right child (1100).
    - Now the 4th bit of the key is used, since it is 1, we follow the pointer to right child and arrive to key 1101.
    - Since the number of bit is 1 (<4) we compare its key with the searched key, since they are equal, we end with success.



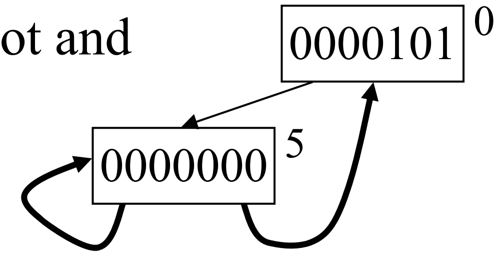
Lexicographical trees

– Inserting a key in *Patricia*:

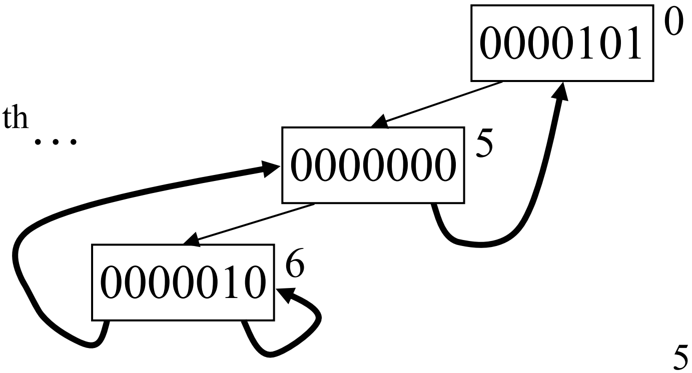
- We start from empty tree (no key).
- We insert key 0000101.



- Now, we insert key 0000000. Searching that key we arrive to the root and see that it is different. We see that the first bit where they differ is the 5th. We create left child labelled with bit 5 and store the key in it. Since 5th bit of inserted key is 0, left pointer of that node points to itself. And right pointer points to root node.

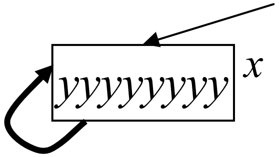


- We insert now 0000010. Search ends at 0000000. First bit where they differ is 6th…



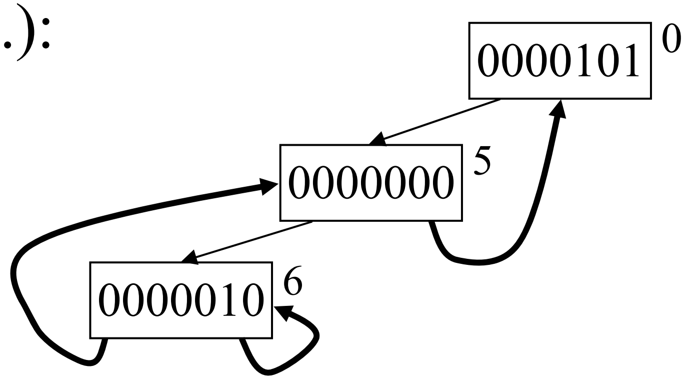
Lexicographical trees

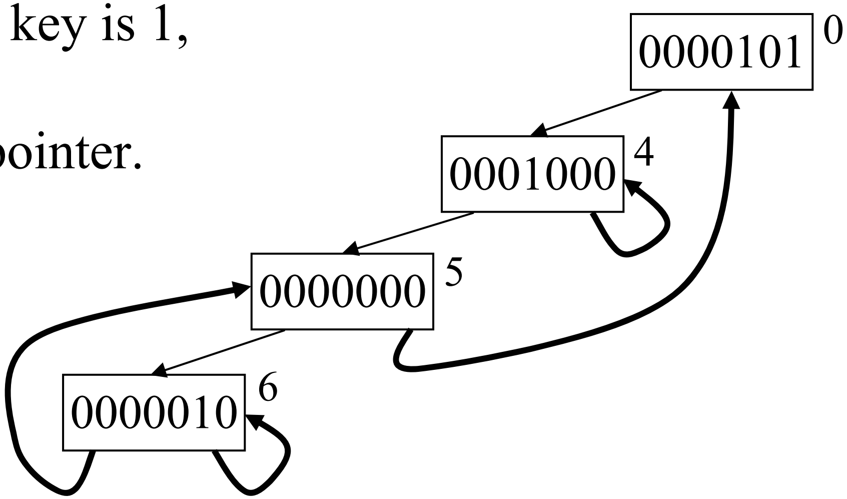
- General strategy to insert (from the 2nd key on):
  - Search the key *C* to insert; search ends at a node with key *C'*.
  - Compute number *b* = the leftmost bit where *C* and *C'* differ.
  - Create a new node with new key inside, labelled with previous bit number, and insert it in the path from the root to node with *C'* in such a way that the labels with bit number are in ascending order in the path.
  - That insertion has broken a pointer from node *p* to node *q*.
  - Now the pointer goes from *p* to the new inserted node.
  - If bit number *b* of *C* is 1, the right child of the new node will point to the same node, otherwise, the left child will be the "self-pointer".
  - The other child will point to *q*.



Lexicographical trees

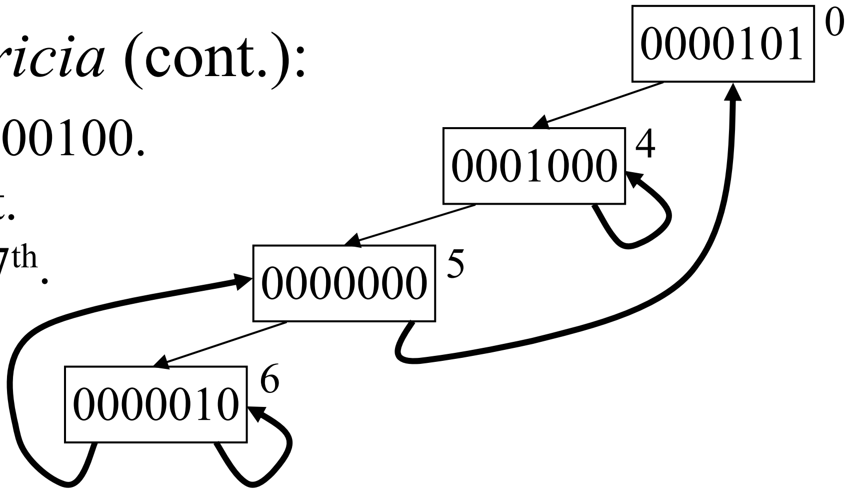
- Inserting a key in *Patricia* (cont.):
  - We insert key 0001000.
    Search ends at 0000000.
    The first bit where they differ is 4.
    Create a new node with label 4
    and put the new key inside.
    Insert the new node in the path from root to node 0000000
    in such a way that bit number labels are in ascending order,
    i.e., we put it as the left child of the root.
    Since 4th bit of inserted key is 1,
    the right child of
    the new node is a self-pointer.



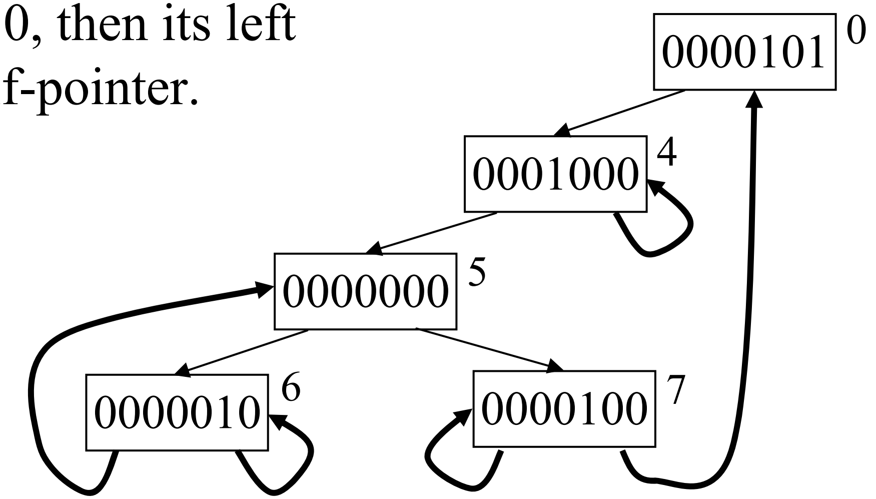


Lexicographical trees

- Inserting a key in *Patricia* (cont.):
  - Now, we insert key 0000100.
    Search ends at the root.
    First bit they differ is 7th.
    Create a new node
    with label 7.

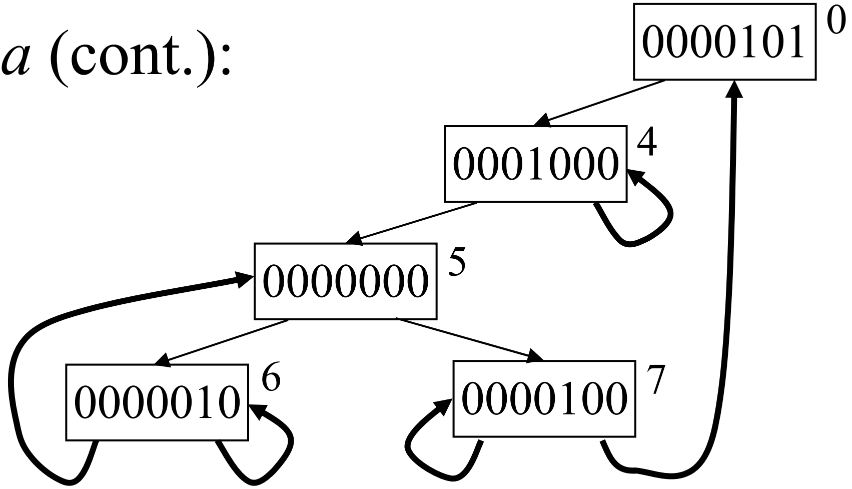


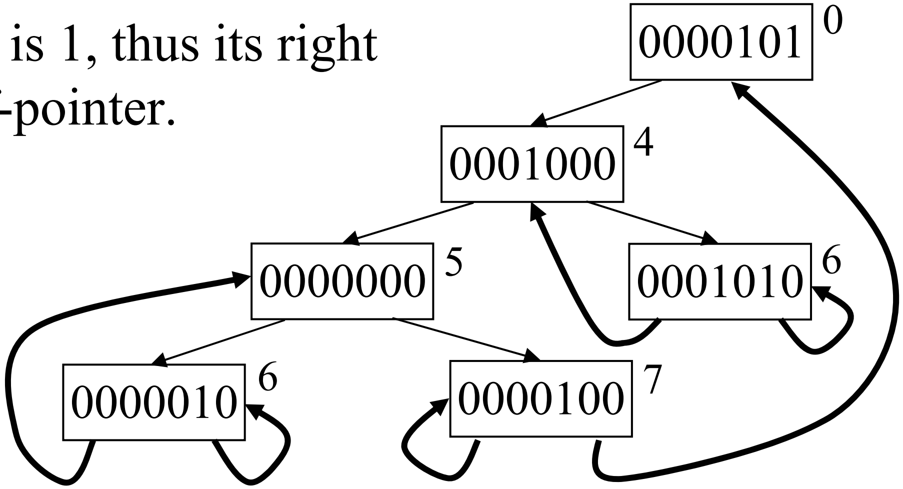
Insert the new node in the search path in such a way that bit number labels are in ascending: right child of 0000000.
Bit 7 of new key is 0, then its left child becomes a self-pointer.



Lexicographical trees

- Inserting a key in *Patricia* (cont.):
  - Insert key 0001010.
    Search ends at 0001000.
    First bit they differ is 6th.
    Create a new node
    with label 6.
    Insert the new node
    in the search path in such a way that bit number labels are in ascending order: right child of 0001000.
    Bit 6 of the new key is 1, thus its right child becomes a self-pointer.





Lexicographical trees

- Deletion of a key:
  - Let *p* be the node with key to delete; two cases:
    - *p* has a self-pointer:
      - » if *p* = root, it is the unique node, delete it → empty tree
      - » if *p* is not the root, we change the pointer going from *p*'s parent to *p* and now it points to the child of *p* (the one that is not a self-pointer.)
    - *p* has not a self-pointer :
      - » search node *q* that has a pointer going up to *p*
(the node from where we arrived to *p* in the search of the key to delete)
      - » move the key in *q* to *p* and delete node *q*
      - » to delete *q*, search node *r* having a pointer going up to *q* (just searching the key in *q*)
      - » change the pointer from *r* to *q* and make it point to *p*
      - » pointer going down from *q*'s parent to *q* is changed to point to the *q*'s child that was used to locate *r*

Lexicographical trees

- Analysis of algorithms:
  - It is obvious that the cost of *search*, *insert* and *delete* operations is in linear order in the height of the tree, but… how much is that?
  - Case of binary tries (m = 2, i.e., alphabet with 2 symbols)…
  - There is a curious relation between this kind of trees and a sorting algorithm, "*radix*-exchange", so let us quickly review *radix* sorting methods

Lexicographical trees

- Post office method, also known as *bucket sort*
  - If we need to sort letters by provinces, we put a box or *bucket* per each province and we perform a sequential traverse of all the letters storing them at the correxponding box ➔ very efficient!
  - If letters must be sorted by postal code (like 10149), we need 100.000 boxes (and a big office) ➔ the method is only (very) useful if the number of potential different elements is small.
  - In general, if *n* elements must be sorted and they can take values in a set of *m* different values (boxes), the cost in time (and space) is $O(m+n)$.

Lexicographical trees

- First idea to naturally extend *bucket sort* method: in the case of sorting letters by postal code…
  - First phase: we use 10 boxes to sort letters by the first digit of its code
    - each box may contain now 10.000 different codes
    - the cost in time for this phase is $O(n)$
  - Second phase: proceed in a similar way for each box (using the next digit)
  - There are five phases…

Lexicographical trees

- A more elaborated extension: *radix sort*

> “Once upon a time, computer programs were written in Fortran and entered on punched cards, about 2000 to a tray. Fortran code was typed in columns 1 to 72, but columns 73-80 could be used for a card number. If you ever dropped a large deck of cards you were really in the poo, unless the cards had been numbered in columns 73-80. If they had been numbered you were saved; they could be fed into a card sorting machine and restored to their original order.
>
> The card sorting machine was the size of three or four large filing cabinets. It had a card input hopper and ten output bins, numbered 0 to 9. It read each card and placed it in a bin according to the digit in a particular column, e.g. column 80. This gave ten stacks of cards, one stack in each bin. The ten stacks were removed and concatenated, in order: stack 0, then 1, 2, and so on up to stack 9. The whole process was then repeated on column 79, and again on column 78, 77, etc., down to column 73, at which time your deck was back in its original order!
>
> The card sorting machine was a physical realisation of the radix sort algorithm.”
>
> http://www.csse.monash.edu.au/~lloyd/tildeAlgDS/Sort/Radix.html

Lexicographical trees

- *Radix sort*:
  - A (FIFO) queue of keys is used to implement each "box" (as many of them as the used numeration base, any base is valid).
  - Keys are classified according to its rightmost digit (the less significant), i.e., each key is placed in the queue corresponding to its rightmost digit.
  - All the queues are concatenated (ordered according to the rightmost digit).
  - Repeat the process, classifying according to the second from the right digit.
  - Repeat the process for all digits.

Lexicographical trees

```
algorithm radix(X,n,k)  {pre: X=array[1..n] of keys,
             each one with k digits;  post: X sorted}
begin
  put the elements of X in queue GQ {X can be used};
  for i:=1 to d do {d = used numeration base)
    emptyqueue(Q[i])
  od;
  for i:=k downto 1 do
    while not isempty(GQ) do
      x:=front(GQ); dequeue(GQ);
      d:=digit(i,x); enqueue(x,Q[d])
    od;
    for t:=1 to d do insertQueue(Q[t],GQ) od
  od;
  for i:=1 to n do
    X[i]:=front(GQ); dequeue(GQ)
  od
end
```

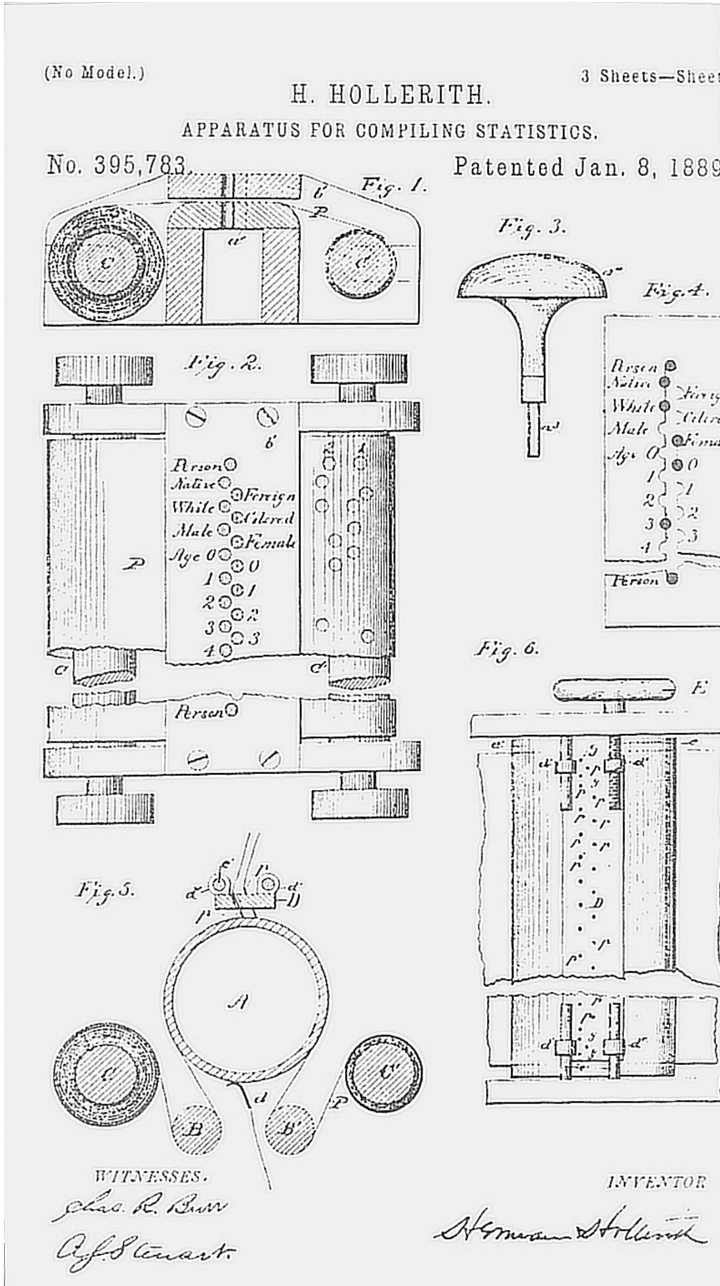
Lexicographical trees

- Analysis of *radix sort* method:
  - In time: $O(kn)$, i.e., considering that the number of bits (k) of each key is a constant, it is a linear time method on the number (n) of keys
  - Notice that the bigger numeration base, the lower cost
  - In space: $O(n)$ additional space

    (It is possible to do it in situ, with additional space on $O(\log n)$ to store array indices)

Lexicographical trees

- Some few history: origins of *radix sort*
  - USA, 1880: census of previous decade cannot be finished (specifically, problem to count the number of single inhabitants)
  - Herman Hollerith (under contract for the US Census Bureau) invents an electric tabulating machine to solve the problem; essentially, an implementation of *radix sort*

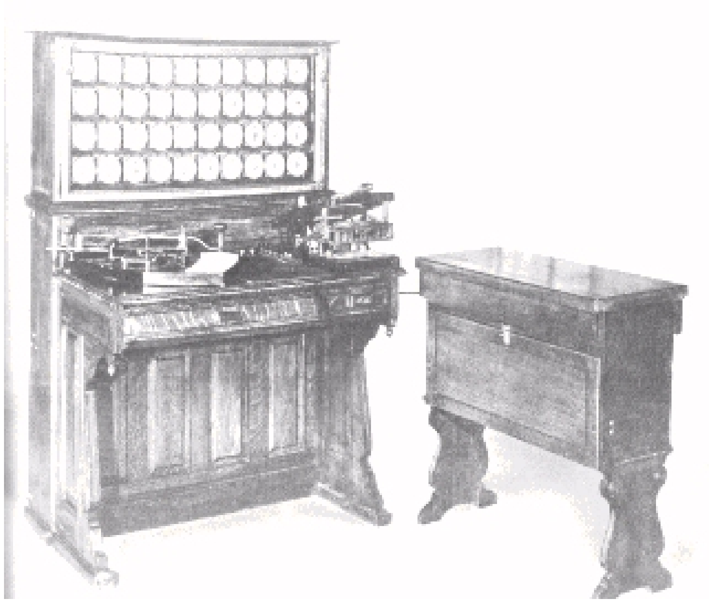


Lexicographical trees

- Some few history: origins of *radix sort* (cont.)
  - 1890: about 100 Hollerith machines are used to tabulate the census of the decade (an expert operator processed 19.071 cards in a working journey of 6’5 hours, about 49 cards by minute)
  - 1896: Hollerith starts the *Tabulating Machine Company*

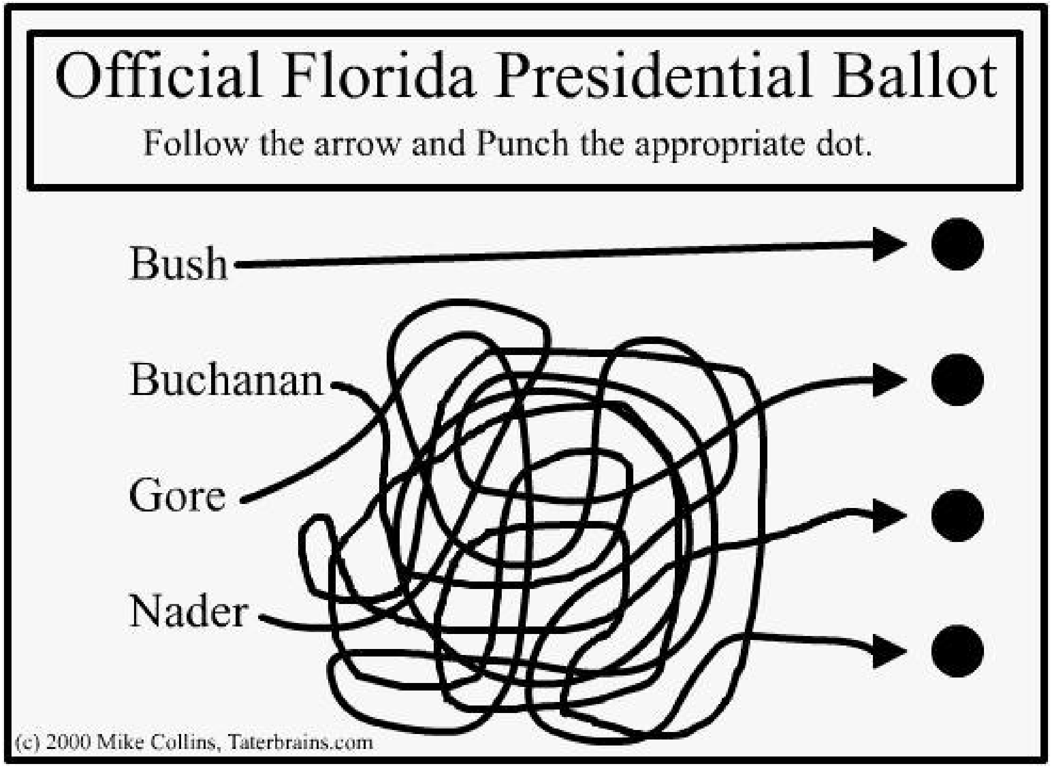
Lexicographical trees

- Some few history: origins of *radix sort* (cont.)
  - 1900: Hollerith solves another federal crisis inventing a new machine with automatic card-feed mechanism (in use, with a few variations, until 1960)
  - 1911: Hollerith's firm merged with others to create *Calculating-Tabulating-Recording Corporation* (CTR)
  - 1924: Thomas Watson renames CTR to *International Business Machines* (IBM)



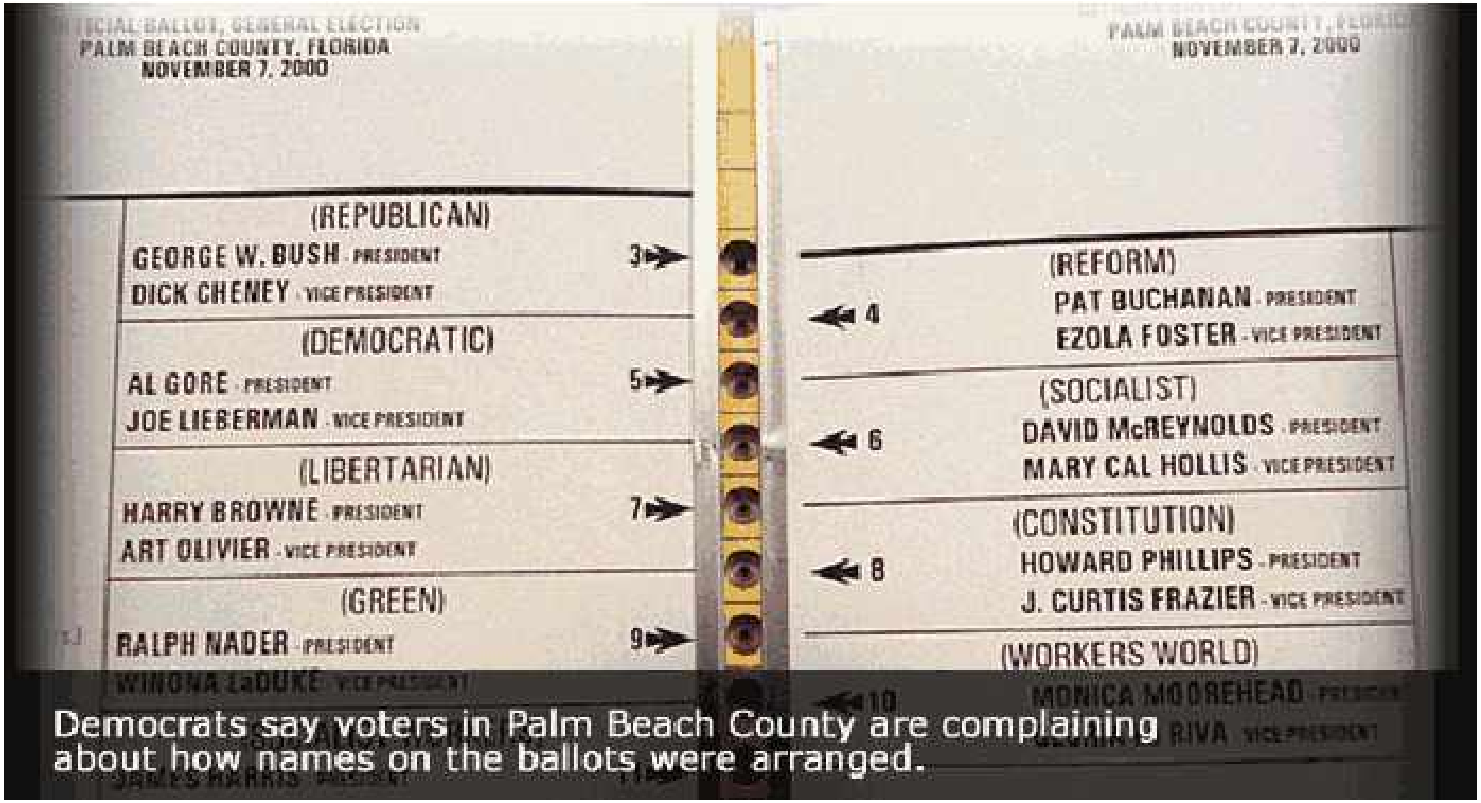
Lexicographical trees

- Some few history: origins of *radix sort* (cont.)
  - The rest of history is well known… until:
  - 2000: USA Presidential Election Crisis


Official Florida Presidential Ballot
Follow the arrow and Punch the appropriate dot.
Bush
Buchanan
Gore
Nader
(c) 2000 Mike Collins, Taterbrains.com

Lexicographical trees

- Some few history: origins of *radix sort* (cont.)
  - That was a joke. This is the real one:



ICIAL BALLOT, GENERAL ELECTION
PALM BEACH COUNTY, FLORIDA
NOVEMBER 7, 2000

(REPUBLICAN)
GEORGE W. BUSH - PRESIDENT
DICK CHENEY - VICE PRESIDENT 3

(DEMOCRATIC)
AL GORE - PRESIDENT
JOE LIEBERMAN - VICE PRESIDENT 5

(LIBERTARIAN)
HARRY BROWNE - PRESIDENT
ART OLIVIER - VICE PRESIDENT 7

(GREEN)
RALPH NADER - PRESIDENT 9
WINONA LaDUKE - VICE PRESIDENT

PALM BEACH COUNTY, FLORIDA
NOVEMBER 7, 2000

4 (REFORM)
PAT BUCHANAN - PRESIDENT
EZOLA FOSTER - VICE PRESIDENT

6 (SOCIALIST)
DAVID McREYNOLDS - PRESIDENT
MARY CAL HOLLIS - VICE PRESIDENT

8 (CONSTITUTION)
HOWARD PHILLIPS - PRESIDENT
J. CURTIS FRAZIER - VICE PRESIDENT

10 (WORKERS WORLD)
MONICA MOOREHEAD - PRESIDENT
GLORIA LA RIVA - VICE PRESIDENT

Democrats say voters in Palm Beach County are complaining about how names on the ballots were arranged.

Lexicographical trees

- *Radix*-exchange sorting method:

  (version by D. Knuth in his Book)

  - Supose keys are stored in its binary representation
  - Instead of comparing keys, we compare bits
    - Step 1: keys are sorted according to its most significative bit
      - Find the leftmost key k_i with its first bit equal to 1 and the rightmost key k_j with its first bit equal to 0, exchange both keys and repeat the process until $i > j$
    - Step 2: sequence of keys is splitted into two parts and step 1 is applied to each part recursively
      - Sequence of keys has been splitted into two: those starting with 0 and the rest staring with 1; previous step is applied recursively to both subsequences of keys, but now taking into consideration the second most significative bit, etcetera.

Lexicographical trees

- *Radix*-exchange and *quicksort* are very similar:
  - Both are based in the *partition* idea.
  - Keys are exchanged until sequence is splitted intto two parts:
    - Left subsequence, where all keys are less than or equal to a given key *K* and right subsequence where all keys are greater than or equal to *K*.
    - *Quicksort* takes as key *K* an existing key in the sequence while *radix*-exchange takes an artificial key based on the binary representation of keys.
  - Historically, *radix*-exchange was published one year before *quicksort* (in 1959).

Lexicographical trees

- Analysis of *radix*-exchange:
  - The asymptotic analysis of radix-exchange is… let us say… a non-trivial matter!

According to Knuth[(*)], the **average** sorting time is

$$U_n = \underline{n \log n} + n\left(\frac{\gamma - 1}{\ln 2} - \frac{1}{2} + f(n)\right) + O(1),$$

with $\gamma = 0{,}577215...$ the Euler constant

and $f(n)$ a "quite strange" function such that $|f(n)| < 173 * 10^{-9}$

(*) Requires infinite series manipulation and their approximation, complex-variable mathematical analysis (complex integrals, Gamma function)…

Lexicographical trees

- Relation with cost analysis of tries (binary case):
  - The number of internal nodes in a binary trie that stores a set of keys is equal to the number of partitions achieved with *radix*-exchange sorting method to sort the same set of keys.
  - The **average** number of bit queries to find a key in a binary trie with *n* keys is 1/*n* times the number of bit queries needed to sort those *n* keys using *radix*-exchange.

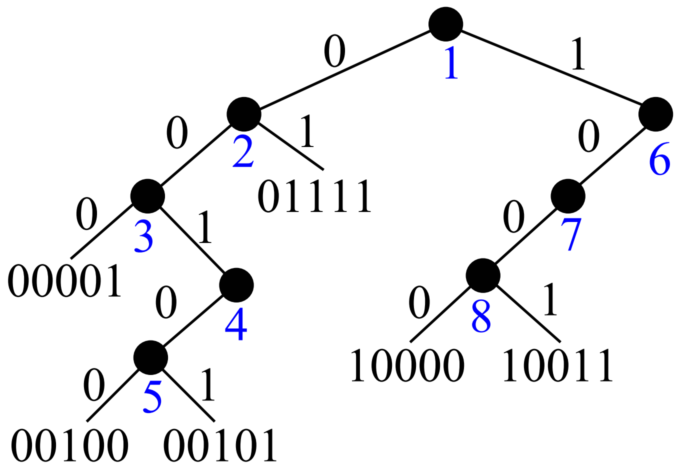
Lexicographical trees

– Example: with 6 keys, the letters in ‘ORDENA’

(keys coded using MIX code, pág. 49)

							l	r	bit
1	10000	10011	00100	00101	01111	00001	1	6	1
2	00001	01111	00100	00101	¦ 10011	10000	1	4	2
3	00001	00101	00100	¦ 01111	¦ 10011	10000	1	3	3
4	00001	¦ 00101	00100	¦ 01111	¦ 10011	10000	2	3	4
5	00001	¦ 00101	00100	¦ 01111	¦ 10011	10000	2	3	5
6	00001	¦ 00100	¦ 00101	¦ 01111	¦ 10011	10000	5	6	2
7	00001	¦ 00100	¦ 00101	¦ 01111	¦ 10011	10000	5	6	3
8	00001	¦ 00100	¦ 00101	¦ 01111	¦ 10011	10000	5	6	4
9	00001	¦ 00100	¦ 00101	¦ 01111	¦ 10000	¦ 10011			_

8 partitions: internal nodes of the tree correspond to partitions (the *k*-th node in a pre-order traversal of the tree corresponds to the *k*-th partition).



The number of bit queries at a partition level is equal to the number of keys in the subtree of the corresponding node.

Lexicographical trees

- Then, the **average** cost of searching in a binary trie with *n* keys is:

$$U_n = \underline{\log n} + \frac{\gamma - 1}{\ln 2} - \frac{1}{2} + f(n) + O(n^{-1}),$$

with $\gamma = 0{,}577215\ldots$ the Euler constant

and $f(n)$ a "quite strange" function such that $|f(n)| < 173 * 10^{-9}$

- The **average** number of nodes of a binary trie with *n* keys is:

$$\boxed{\frac{n}{\ln 2}} + ng(n) + O(1),$$

with $g(n)$ a negligible function, like $f(n)$

Lexicographical trees

- Analysis of *m*-ary tries:
  - The analysis is as difficult (or even more) as the binary case…, leading to:
    - The **average** number of nodes needed to randomly store n keys in a m-ary trie is approximately $n/\ln m$
    - The number of digits or characters examined in an **average** key search operation is approximately $\underline{\log_m n}$
- The analysis of digital search trees and *Patricia* leads to very similar results
- According to Knuth, the analysis of *Patricia* includes…

  "... possibly the hardest asymptotic nut
  we have yet had to crack…"

Average cost of a search operation

	Successful search	Unsuccessful search
Search in a trie	log n + 1,33275	log n – 0,10995
Search in a digital tree	log n – 1,71665	log n – 0,27395
Search in Patricia	log n + 0,33275	log n – 0,31875

Outline

A short course in data structures analysis…

- Worst case analysis:
  - *Red-black trees*
- Average case analysis:
  - *Lexicographical trees*
  - *Skip lists*
- Amortized analysis
  - *Splay trees*
- Data structures for computational biology
  - *Suffix trees*

… and also an intermediate course in *dictionary* abstract data type

Skip lists

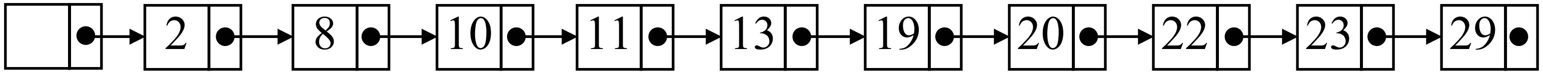
- They are "*probabilistic* data structures".

- They are a good alternative for balanced search trees (AVL, 2-3, red-black,…) to store dictionaries with n keys with an **average** basic operations cost in $O(\log n)$.

- They are much more easy to implement than, for instance, AVL or red-black trees.

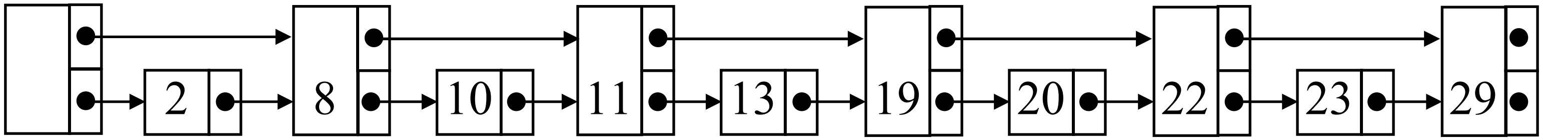
Skip lists

- Linked list (sorted):



  – The worst-case cost for searching a key is linear on the number of nodes.

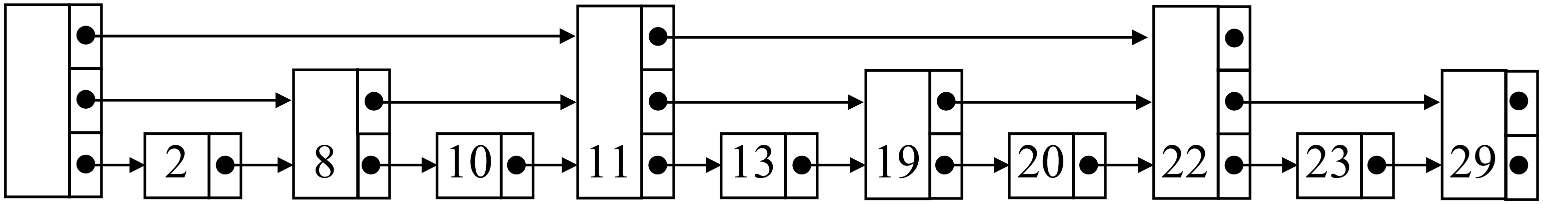
- But, adding a pointer to each even node…



  – Now, the number of nodes examinated during a search is, at most, $\lceil n/2 \rceil + 1$.

Skip lists

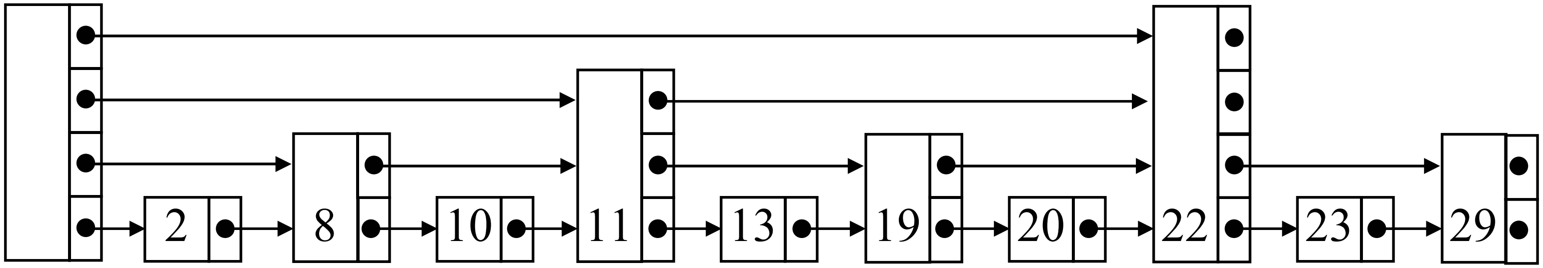
- And adding another pointer to each node in a multiple of 4 position …



  – Now, the number of nodes examinated during a search is, at most, $\lceil n/4 \rceil + 2$.

Skip lists

- And finally, the **limit case**: each node in a multiple of 2^i position, points to the node sited 2^i places ahead (for all $i \geq 0$):



  – The total number of pointers is doubled (with respect to the initial linked list).

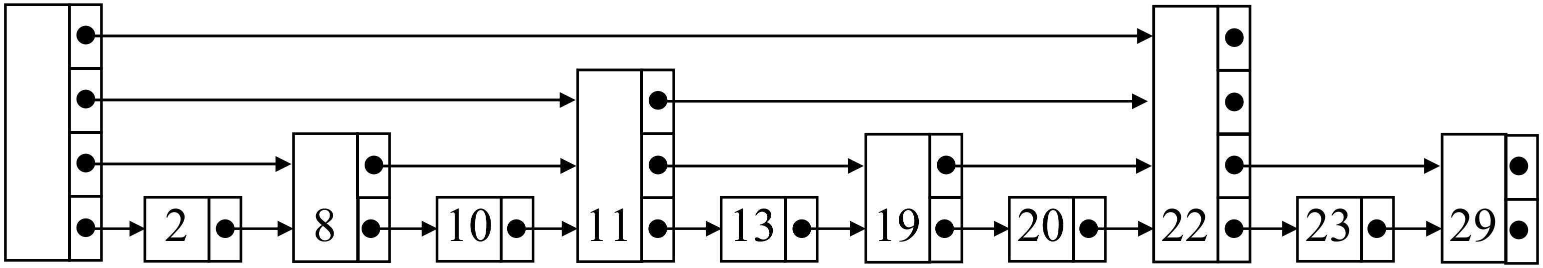
  – Now, the time for a search is bounded above by $\lceil \log_2 n \rceil$, because searching consists on going ahead to the next node (using the highest pointers) or going down to next level of pointers and go ahead…

  – Essentially, it is a binary search.

  – Problem: insertion and deletion are too difficult!

Skip lists

- In particular…



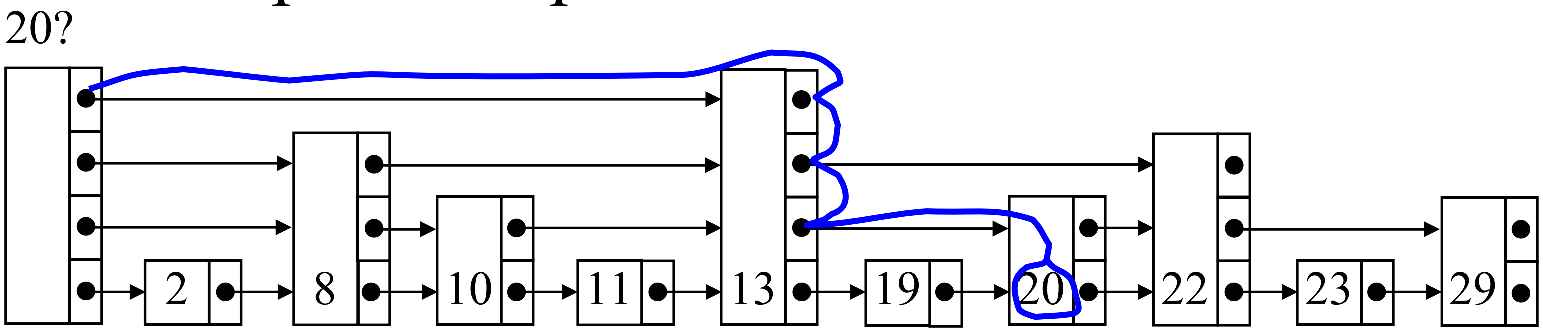
  - If we call "height *k*-node" to that with *k* pointers, the following property holds (thus, it is a weaker property than the "limit case" definition):

> The *i*-th pointer of any height *k*-node ($k \geq i$) points to the next height *i*-node or higher (i.e. height *j*-node with $j > i$).

  - We adopt this as definition of skip list (together with a random selection of height assigned to a new node).

Skip lists

- An example of skip list:



- How to implement the search of a key?
  - We start from the higest pointer in the head node.
  - Go ahead in the same level until a key greater than the searched key is found (or NULL), then go down 1 level and go ahead in the new level.
  - When advance stops at level 1, either we are in front of searched key or it does not exist in the list.

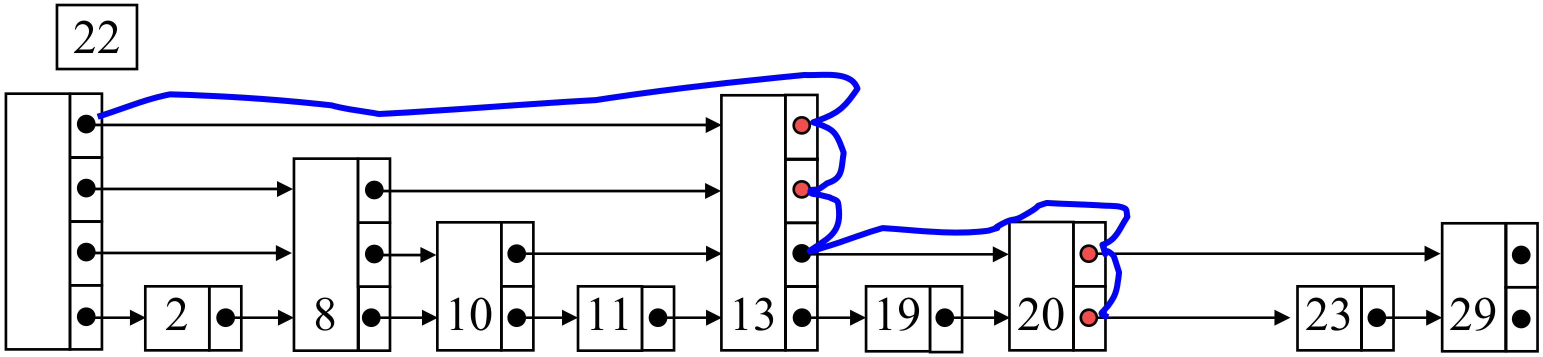
Skip lists

- And insertion? (deletion is similar)
  - First: to insert a new element, the height of its corresponding node must be decided.
    - In a "limit case" list, half of the nodes have height 1, 1/4 of nodes have height 2 and, in general, $1/2^i$ nodes have heigth i.
    - Height of a new node is selected according to those probabilities: throw a coin until you get heads, the total number of needed throwings is selected as heigth of the node.
      - Geometric distribution with parameter $p = 1/2$.

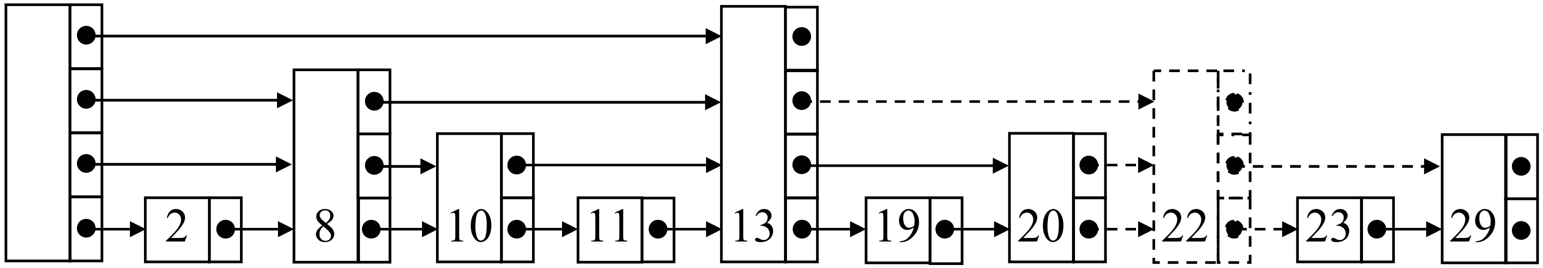
        (In fact, an arbitrary parameter can be selected, p, then we could select the more suitable value for p)

Skip lists

- Second: we need to know where to insert
  - Proceed as in a search, keeping the trace of nodes where we go one level down.



  - Decide randomly the height of the new node and insert it, linking the pointers conveniently.



Skip lists

- An a priori estimation of the length of the list is needed (as in hash tables) to determine the maximum heigth of the nodes.
- If such estimation is not available, a “big” number may be assumed or a *rehashing*–like technique can be used (reconstruction).
- Experimental results seem to show that skip lists are so efficient as many balanced search tree implementations, and they are easier to implement.

Skip lists

```
function search(list:skip_list; searched_key:keytype)
         return valuetype
begin
  x:=list.head;
  {invariant: x↑.key<searched_key}
  for i:=list.heigth downto 1 do
    while x↑.next[i]↑.key<searched_key do
      x:=x↑.next[i]
    od
  od;
  {x↑.key<searched_key≤x↑.next[1]↑.key}
  x:=x↑.next[1];
  if x↑.key=searched_key then
    return x↑.value
  else
    failure of the search
  fi
end
```

Skip lists

```
algorithm random_height return natural
begin
  height:=1;
  {random function returns a uniform value [0,1)}
  while random<p and height<MaxHeight do
    height:= height+1
  od;
  return height
end
```

MaxHeight is selected as $\log_{1/p} N$, where N = upper bound of list length. For instance, if $p = 1/2$, MaxHeight = 16 is perfect for lists with up to 2^{16} (= 65.536) elements.

Skip lists

```
algorithm insert(list:skip_list; k:keytype; v:valuetype)
variable trace:array[1..MaxHeight] of pointer
begin
  x:=list.head;
  for i:=list.heigth downto 1 do
    while x↑.next[i]↑.key<k do
      x:=x↑.next[i]
    od;
    {x↑.key<k≤x↑.next[i]↑.key}
    trace[i]:=x
  od;
  x:=x↑.next[1];
  if x↑.key=k then
    x↑.value:=v
  else {insertion}
  . . .
```

Skip lists

```
  . . .
  else {insertion}
    height:=random_height;
    if height>list.height then
      for i:=list.height+1 to height do
        trace[i]:=list.head
      od;
      list.height:=height
    if;
    x:=new(height,k,v);
    for i:=1 to height do
      x↑.next[i]:=trace[i]↑.next[i];
      trace[i]↑.next[i]:=x
    od
  fi
end
```

Skip lists

```
algorithm delete(list:skip_list; k:keytype)
variable trace:array[1..MaxHeight] of pointer
begin
  x:=list.head;
  for i:=list.height downto 1 do
    while x↑.next[i]↑.key<k do
      x:=x↑.next[i]
    od;
    {x↑.key<k≤x↑.next[i]↑.key}
    trace[i]:=x
  od;
  x:=x↑.next[1];
  if x↑.key=k then {deletion}
  . . .
```

Skip lists

```
  . . .
  if x↑.key=k then {deletion}
    for i:=1 to list.height do
      if trace[i]↑.next[i]≠x then break fi;
      trace[i]↑.next[i]:=x↑.next[i]
    od;
    dispose(x);
    while list.height>1 and
           list.head↑.next[list.height]=NULL do
      list.height:=list.height-1
    od
  fi
end
```

Skip lists

- Analysis of cost in time:
  - The time needed for searching, insertion and deletion is dominated by the time to search an element.
  - To insert/delete, there is an additional cost that is proportional to the height of the node inserted/deleted.
  - The time needed for searching is proportional to the length of the search path.
  - The length of the search path is determined by the list structure, i.e., by the pattern of heights of nodes in the list.
  - The list structure is determined by the number of nodes and by the results in the random generation of the heights of nodes.

Skip lists

- Analysis of cost in time (details):
  - We analyze the length of the search path from right to left, i.e., starting from the position immediately before the searched element
    - First: how many pointers do we need to visit to move upward from height 1 (of the element just before the searched one) to height $L(n) = \log_{1/p} n$?
      - we assume that we reach the height $L(n)$; that hypothesis is like assuming that the list is infinitely large to the left
      - suppose that, during the climb, we are at i-th pointer of a certain node x
      - the height of x must be at least i, and the probability for the height of x to be greater than i is p

Skip lists

- Climb from height 1 to height $L(n)$…
  - Climb to the height $L(n)$ can be interpreted as a series of independent Bernoulli trials, calling “success” to an upwards movement and “failure” to a leftward movement.
  - Then, the number of leftward movements in the climb up to height $L(n)$ is the number of failures until the $(L(n)\text{-}1)$-th success of the experiment series, i.e., it is a negative binomial random variable $NB(L(n)-1,p)$.
  - The number of movements upward is exactly $L(n)-1$, then:

    the cost of climbing to the height $L(n)$ in a list with infinite length is $=_{\text{prob}} (L(n)-1) + NB(L(n)-1,p)$

    Note: $X =_{\text{prob}} Y$ if $\Pr\{X>t\} = \Pr\{Y>t\}$, for all t

    and also, $X \leq_{\text{prob}} Y$ if $\Pr\{X>t\} \leq \Pr\{Y>t\}$, for all t.
  - The infinity hypothesis for the length of the list is pessimistic, i.e.: the cost of climbing to the height $L(n)$ in a list of length n $\leq_{\text{prob}} (L(n)-1) + NB(L(n)-1,p)$

Skip lists

- Second: once in the height $L(n)$, how many leftwards movements are needed to reach the head of the list?
  - It is bounded by the number of elements of heigth $L(n)$ or higher in the list. This number is a binomial random variable $B(n,1/np)$.
- Third: once in the head of the list, we need to climb to the highest heigth.
  - M = random variable “max. heigth in a list with n elements”

    $\Pr\{\text{height of a node}>k\} = p^k \Rightarrow \Pr\{M>k\} = 1-(1-p^k)^n < np^k$
  - Then we have: $M \leq_{\text{prob}} L(n) + NB(1,1-p) +1$

    Proof: $\Pr\{NB(1,1-p)+1>i\} = p^i \Rightarrow$
    $\Pr\{L(n)+NB(1,1-p)+1>k\} = \Pr\{NB(1,1-p)+1>k-L(n)\} = 1/2^{k-L(n)} = np^k$.
    Then: $\Pr\{M>k\} < \Pr\{L(n)+NB(1,1-p)+1>k\}$ for all k.

Skip lists

– Putting all together:

Number of comparisons in the search =

= length of the search path + 1 $\leq_{\text{prob}}$

$\leq_{\text{prob}}$ $L(n) + NB(L(n)\text{-}1,p) + B(n,1/np) + NB(1,1\text{-}p) + 1$

And its average value is

$$L(n)/p + 1/(1\text{-}p) + 1 = \underline{O(\log n)}$$

Selecting p…

p	searching time (normalized for 1/2)	Average number of pointers by node
1/2	1	2
1/e	0,94…	1,58…
1/4	1	1,33…
1/8	1,33…	1,14…
1/16	2	1,07…

Skip lists

- Comparison with other data structures:
  - The cost of the operations is in the same order of magnitude than for balanced search trees (AVL) and for self-organizing trees (we will see them…).
  - Operations are easier to implement than for balanced or self-organizing trees.
  - Constant factors make the difference:
    - these factors are fundamental, especially for sub-linear algorithms (like here):
      if A and B solve the same problem in $O(\log n)$ but B is twice faster than A, then during the time in which A solves a problem of size n, B solves another problem of size n^2.

Skip lists

- “Complexity” (in the sense of difficulty to implement) inherent to an algorithm usually fixes a lower bound to the constant factor of any implementation of that algorithm.
  - For instance, self-organizing trees are continuously arranging while a search is done, while the innermost loop in the deletion procedure for skip lists is compiled into only 6 instructions in a 68020 CPU.
- If a given algorithm is “difficult”, programmers will postpone (… or they will never do) possible optimizations in the implementation.
  - For instance, insertion and deletion algorithms for balanced search trees are usually implemented recursively, with the additional cost that this fact means (in each recursive call…). However, due to the inner difficulty of those algorithms, iterative solutions are not usually implemented

Skip lists

implementatiun	search	insertion	deletion
Skip list	0,051 ms (1,0)	0,065 ms (1,0)	0,059 ms (1,0)
AVL non-recurs.	0,046 ms (0,91)	0,10 ms (1,55)	0,085 ms (1,46)
2-3 tree recurs.	0,054 ms (1,05)	0,21 ms (3,2)	0,21 ms (3,65)
self-organ. tree:			
downward adjust.	0,15 ms (3,0)	0,16 ms (2,5)	0,18 ms (3,1)
upward adjust.	0,49 ms (9,6)	0,51 ms (7,8)	0,53 ms (9,0)

- All the implementations were optimized.
- Refer to CPU time in a Sun-3/60 and using a data structure with 2^{16} (= 65.536) integer keys.
- Values in brackets are relative, normalized for skip lists.
- Insertion and deletion times do NOT include the time required to manage dynamic memory (“new” and “dispose”).

Outline

A short course in data structures analysis…

- Worst case analysis:
  - *Red-black trees*
- Average case analysis:
  - *Lexicographical trees*
  - *Skip lists*
- Amortized analysis
  - *Splay trees*
- Data structures for computational biology
  - *Suffix trees*

… and also an intermediate course in *dictionary* abstract data type

Amortized analysis

- Amortized analysis
  - Computation of average cost of an operation, obtained by dividing the worst-case cost for the execution of a sequence of operations (not necessarily of the same type) divided by the number of operations
- Utility:
  - It is possible that the worst-case cost for the isolated execution of an operation is very high while if the operation is considered in a complete sequence of operations the average cost reduces drastically
- Note:
  - It is NOT an average-cost analysis like the usual one, e.g. that computed for skip lists (probability space for input data does not appear now)

Amortized analysis

- Actually, amortized cost of an operation is an "**accounting trick**" that has no relation with the actual cost of the operation.
  - Amortized cost of an operation can be defined as **anything** with the only condition that considering a sequence of *n* operations:

$$\sum_{i=1}^{n} A(i) \geq \sum_{i=1}^{n} C(i)$$

where $A(i)$ and $C(i)$ are the amortized cost and the exact cost, respectively, of the i-th operation in the sequence.

Amortized analysis

Aggregated analysis:

- Consists in computing the total worst-case cost $T(n)$ for a sequence of n operations, not necessarily of the same type, and computing the average cost or *amortized cost* of a single operation in the sequence as $T(n)/n$.
- The next method that we will consider (accounting method / potential method) computes an amortized cost specific for each type of operation.

Amortized analysis

- Example: *stack* with *multiPop* operation
  - Consider a stack represented with a linked list (using pointers) of records with the typical operations of *createEmpty*, *push*, *pop* and *isEmpty*.
  - The real cost of all these operations is Θ(1), thus, the cost of a sequence of n operations of *push* and *pop* is Θ(n).
  - Now we add the *multiPop*(s,k) operation, that deletes the k top elements in stack s, if there are so many, or empties the stack otherwise.

```
algorithm multiPop(s,k)
begin
  while not isEmpty(s) and k≠0 do
    pop(s); k:=k-1
  od
end
```

Amortized analysis

- The exact cost of *multiPop* is, obviously, $\Theta(\min(h,k))$, where h is the height of the stack before the operation.
- What is the cost of a sequence of n operations of *push*, *pop* or *multiPop*?
  - The maximum height of the stack can be in $O(n)$, then the maximum cost of a *multiPop* operation in that sequence can be in $O(n)$.
  - Then, the maximum cost of a sequence of n operations is bounded by $O(n^2)$.
  - The above computation is correct, but the bound $O(n^2)$, obtained *considering the worst case for each operation in the secquence*, is not tight.
  - Aggregated analysis considers the worst case for the execution of the sequence as a whole…

Amortized analysis

- Aggregated analysis for the sequence of operations:
  - Each element present in the stack can be popped at the most only once in the whole sequence of operations.
  - Then, the maximum number of times that *pop* operation can be executed in a sequence of n operations (including the calls in *multiPop*) is equal to the maximum number of times that *push* operation can be executed, and that is n.
  - Therefore, the total cost of any sequence of n operations of *push*, *pop* or *multiPop* is $O(n)$.
  - And the *amortized cost* of each operation is the average: $O(n)/n = O(1)$.

Amortized analysis

Accounting method (and potential method)

- Imagine the amortized cost **of each operation** as a *prize* asigned to the operation and that can be greater, equal or less than the real cost of the operation.
- When the prize of an operation exceeds its real coste, the resulting *credit* can be later used to pay other operations whose prize is less than its real cost.
- A *potential function*, $P(i)$, [the *balance*] can be defined for each operation in the sequence:

  $$P(i) = A(i) - C(i) + P(i-1), \quad i = 1, 2, \ldots, n$$

  where $A(i)$ and $C(i)$ are the amortized cost and the exact cost, respectively, of the i-th operation.
- The potential for each operation is interpreted as the credit available to pay the rest of the sequence.

Amortized analysis

- By adding the potential of all the operations:

$$\sum_{i=1}^{n} P(i) = \sum_{i=1}^{n}\big(A(i)-C(i)+P(i-1)\big)$$

(by definition of amortized cost)

$$\sum_{i=1}^{n} A(i) \geq \sum_{i=1}^{n} C(i)$$

$$\Rightarrow \sum_{i=1}^{n}\big(P(i)-P(i-1)\big) = \sum_{i=1}^{n}\big(A(i)-C(i)\big)$$

$$\Rightarrow P(n)-P(0) = \sum_{i=1}^{n}\big(A(i)-C(i)\big) \Rightarrow \underline{P(n)-P(0) \geq 0}$$

- Therefore, a prize must be assigned to each operation such that the available credit is always non negative.
- Potential method: similar to this one (define first a positive potential function and then derive the prizes $A(i)$, $i=1,\ldots,n$).

Amortized analysis

- Comming back to the stack example with *multiPop* operation
  - Remember the real cost:
    - $C(push) = 1$
    - $C(pop) = 1$
    - $C(multiPop) = \Theta(\min(h,k))$
  - We assign (arbitrarily) the amortized cost (*prize*) of each operation as:
    - $A(push) = 2$
    - $A(pop) = 0$
    - $A(multiPop) = 0$
  - To see if the above definition of amortized cost is correct we only have to prove that the credit (potential function) is always non negative.

Amortized analysis

- That is, we need to prove that $P(n) - P(0) \geq 0, \forall n$.
- When each element is *pushed*, with prize 2, we pay the real cost of one unit corresponding to *push* operation and another unit is left over as credit.
  - At each time instant we have 1 credit unit per each element stored in the stack.
  - That unit is the *pre-pay* to *pop* that element later.
- When *popping*, the prize of the operation is 0 and the real cost of operation is paid with the credit associated to the *popped* element.
- In this way, paying a few more per *pushing* (2 instead of 1) we do not need to pay per *popping* neither per *multiPopping*.

Splay trees

- Basic ideas:
  - Binary search tree for storing a set of elements of a domain with an order relation defined, with usual operations of searching, inserting, deleting…
  - Renounce to a "strict" balancing after each operation as in AVL, in 2-3, in B, or in red-black trees.
  - They are NOT balanced; height can reach $n - 1$
    - The fact that the worst-case cost of an operation with a binary search tree is $O(n)$ is not so bad…

      … as long as that happens rarely!
  - After the execution of each operation, the tree *tends to* improve its balance, in such a way that the amortized cost of each operation is $O(\log n)$.

Splay trees

- Observe that:
  - If an operation, e.g. searching, can have a worst-case cost in *O*(*n*), and we want to get an amortized cost in *O*(log *n*) for all the operations, then it is essential that the nodes visited during the search operation will move after the searching.
    - Otherwise, we could repeat the same search operation *m* times and we would get a total amortized cost in *O*(*mn*), then we would NOT get and amortized in *O*(log *n*) per each operation.
  - Therefore, after visiting a node, we push it upward the root using rotations (similars to those used for AVL or red-black trees)
    - The restructuration depends only on the path traversed in the access to the searched node
    - With that restructuration, the subsequent accesses (to the same node) will be faster

→ Specially usefull for the case of **temporal locality of reference**

Splay trees

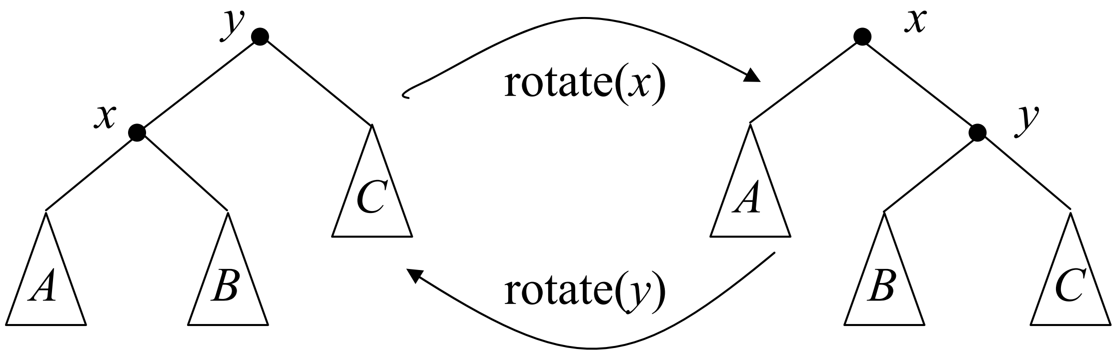
- Basic operation: *splay*
  - splay(i,S): reorganizes the tree S in such a way that the new root is:
    - either the element i if i belongs to S, or
    - $\max\{k \in S \mid k < i\}$ or $\min\{k \in S \mid k > i\}$, if i does not belong to S
- The other operations:
  - search(i,S): tell us if i belong to S
  - insert(i,S): inserts i in S (if it not was in)
  - delete(i,S): deletes i from S (it it was)
  - append(S,S'): joins S and S' in a single tree, assuming that (precondition) $x < y$ for all $x \in S$ and all $y \in S'$
  - split(i,S): splits S into S' and S'' such that $x \leq i \leq y$ for all $x \in S'$ and for all $y \in S''$

Splay trees

- All the previous operations can be implemented with a constant number of *splays* plus a constant number of “atomic” operations (comparisons, pointers assignments…)
  - Example: append(*S*,*S’*) can be implemented with splay(+∞,*S*) that puts the maximum of *S* in its root and the rest in its left subtree, and then putting *S’* as the right subtree of the root of *S*.
  - Another example: delete(*i*,*S*) can be implemented with splay(*i*,*S*) to put *i* in the root, then we delete *i* and execute append to join left and right subtrees.

Splay trees

- Implementation of *splay* operation
  - Remember rotations:



- Preserves the "search structure" of the search tree

Splay trees

- A first possibility to move upward a node *i* to the root is rotating it repeatedly
  - Not useful: it may cause other nodes go down too much
  - We con prove that there is a sequence of *m* operations requireing $\Omega(mn)$:
    - create a tree inserting keys 1, 2, 3, …, *n* in an empty tree, then move the last inserted node to the root by means of rotations
      - » the nodes in the obtained tree only have left child
      - » the total cost till here is $O(n)$
    - searching key 1 takes *n* comparisons, then key 1 is moved to the root with rotations ($n - 1$ rotations)
    - search key 2 takes *n* comparisons and then $n - 1$ rotations
    - iterating the process, a sorted search of all the keys takes

$$\sum_{i=1}^{n-1} i = \Omega(n^2)$$

    - after that, the tree goes back to its original shape, so repeating the sorted search sequence, we get $\Omega(mn)$ for *m* operations.

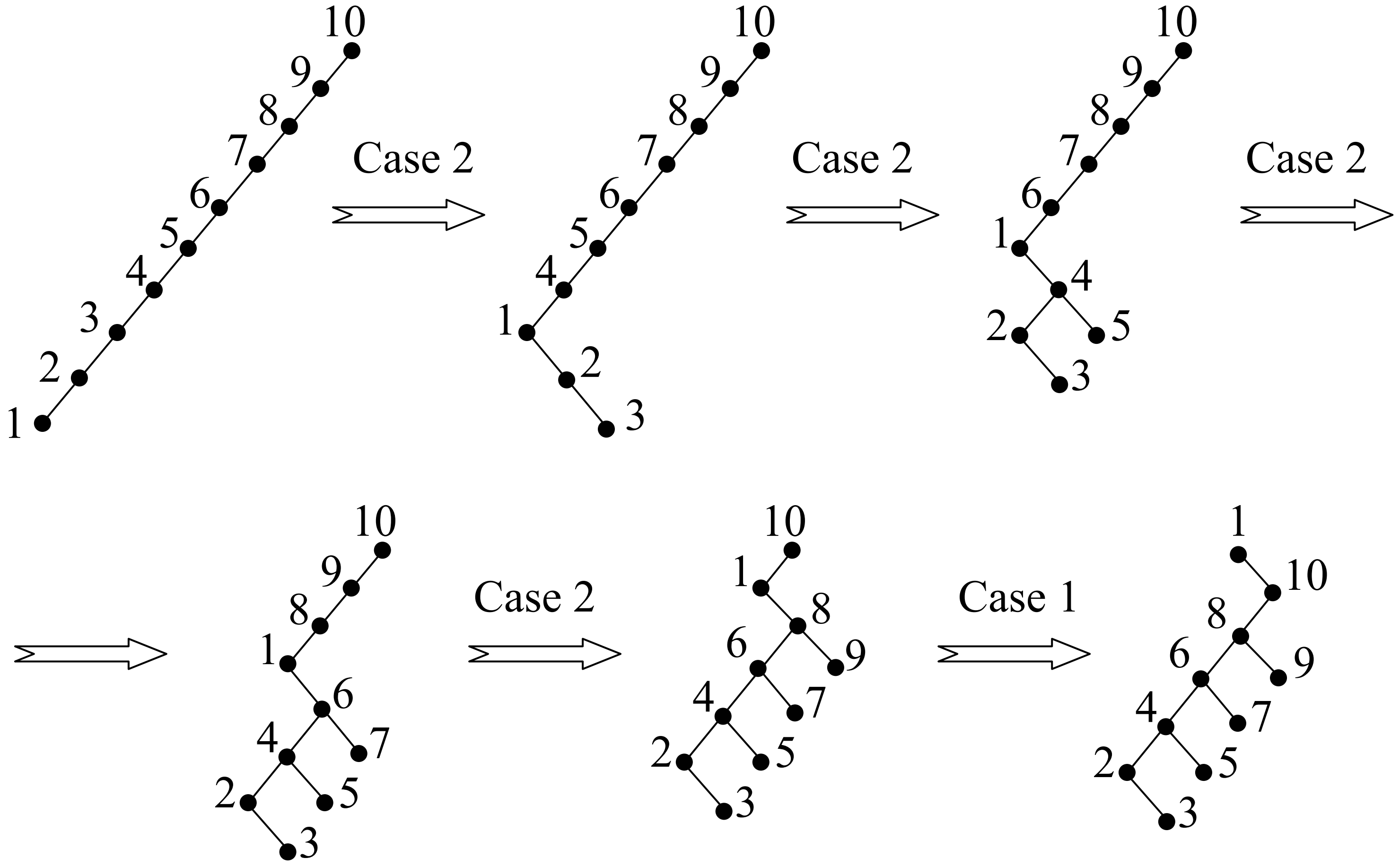
Splay trees

– The good solution (to move *x* upward to the root) is:

- **Case 1:** if *x* has parent but has not grandparent, execute rotate(*x*) (with its parent)
- **Case 2:** if *x* has parent (*y*) and also grandparent, and *x* and *y* are both left child or both right child, then se execute rotate(*y*) and then rotate(*x*) (with its parent in both cases)
- **Case 3:** if *x* has parent (*y)* and also grandparent, and *x* and *y* are one right child and the other left child, then execute rotate(*x*) and then rotate(*x*) again (with its parent in both cases)

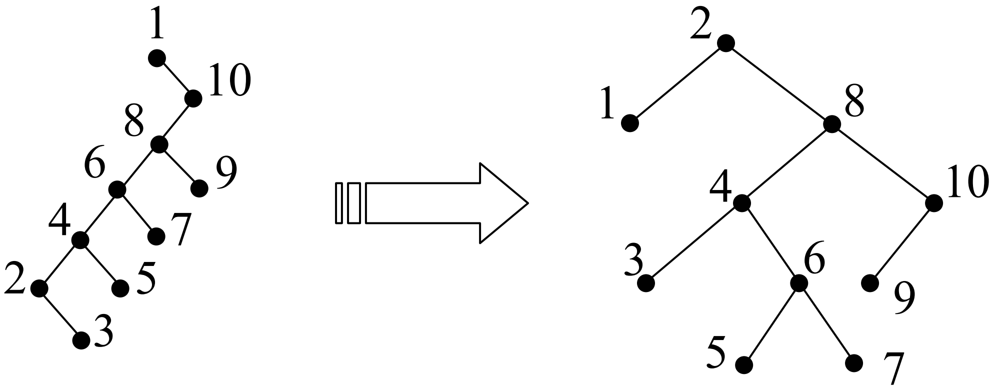
Splay trees

– Example: splay(1,*S*), with *S* a degenerate tree



Splay trees

– Example (cont.): splay(2,*S*) to the previous result…



- Note that the tree is more and more balanced after each *splay*

Splay trees

- Analysis of the cost of *splay* using accounting method:
  - Each node x has a credit associated with it
  - When node x is created, the prize of that creation is associated with x as credit, and will be used for subsequent restructurations
  - Let $S(x)$ be the subtree with x as root and $|S|$ its number of nodes
  - Let $\mu(S) = \lfloor(\log |S|)\rfloor$ and $\mu(x) = \mu(S(x))$
  - We require the following *credit invariant*:

  Node x has always a credit greater than or equal to $\mu(x)$.

Splay trees

– **Lemma:** each *splay* operation requires at most

$$3(\mu(S) - \mu(x)) + 1$$

units of credit to be executed and preserve the credit invariant.

Proof:

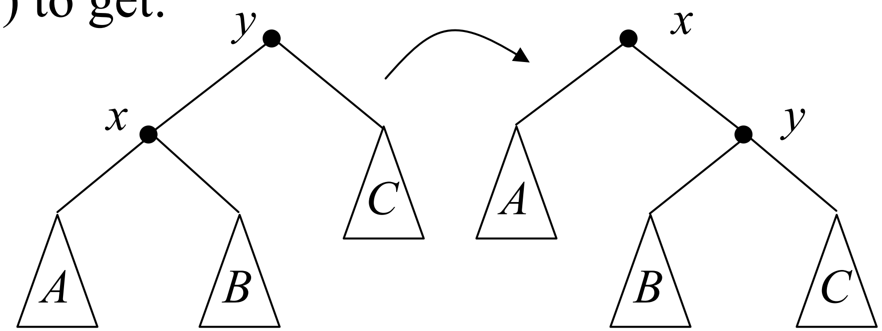
- Let y be the parent of x and z, if it exists, the parent of y.
- Let μ and μ' be the values of μ before and after *splay*.
- There are three cases:
  (a) z does not exist
  (b) x and y are both left child or both right child
  (c) x and y are one right child and the other left child

Splay trees

- **Case 1:** z does not exist
  - That is the last rotation in the *splay* operation
  - We execute rotate(x) to get:

    $\mu'(x) = \mu(y)$

    $\mu'(y) \leq \mu'(x)$



  - To preserve the credit invariant we spend:

    $\mu'(x) + \mu'(y) - \mu(x) - \mu(y) = \mu'(y) - \mu(x)$

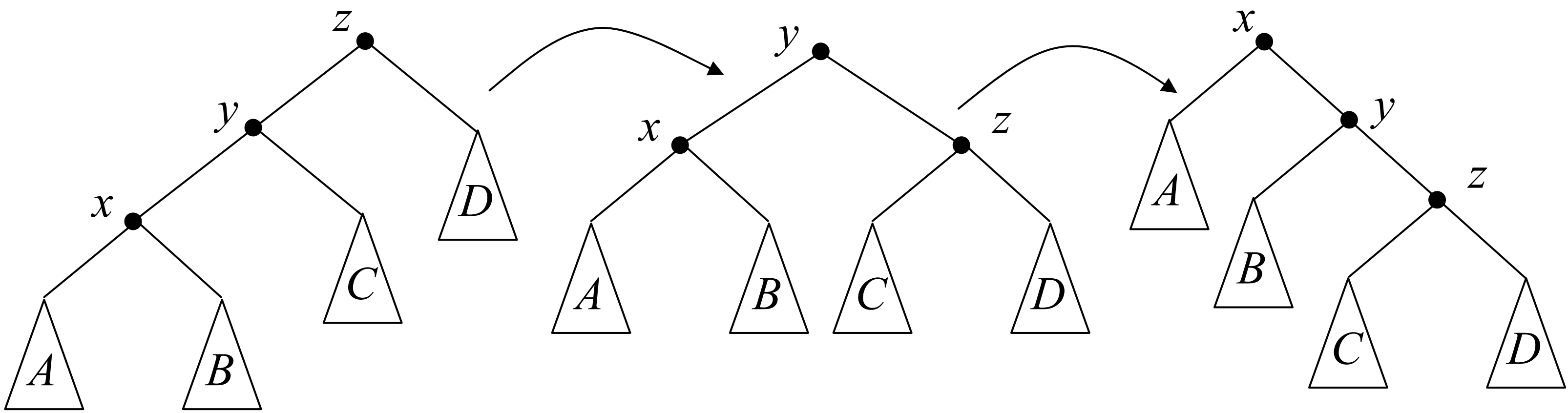
    $\leq \mu'(x) - \mu(x) \leq 3(\mu'(x) - \mu(x)) = 3(\mu(S) - \mu(x))$
  - We add a credit unit due to the constant cost operations (comparisons and pointers manipulation)
  - Therefore, we pay at most

$$3(\mu'(x) - \mu(x)) + 1$$

credit units per rotation

Splay trees

- **Case 2:** *x* and *y* are both left child or both right child
  - Execute rotate(*y*) and then rotate(*x*)



  - First, we will see that the prize of these 2 rotations to preserve the invariant is not greater than 3(μ'(*x*) – μ(*x*))
  - After, if a sequence of rotations is done to move *x* up to the root, a telescopic sum arises, and the total cost is not greater than 3(μ(*S*) – μ(*x*)) +1 (the '+1' comes from the last rotation)

Splay trees

- Prize of the 2 rotations:
  - To preserve the invariant we need

    $\mu'(x) + \mu'(y) + \mu'(z) - \mu(x) - \mu(y) - \mu(z)$ units (*)

  - Since $\mu'(x) = \mu(z)$, we have:

    $$\begin{aligned} &\mu'(x) + \mu'(y) + \mu'(z) - \mu(x) - \mu(y) - \mu(z) \\ &\quad = \mu'(y) + \mu'(z) - \mu(x) - \mu(y) \\ &\quad = (\mu'(y) - \mu(x)) + (\mu'(z) - \mu(y)) \\ &\quad \leq (\mu'(x) - \mu(x)) + (\mu'(x) - \mu(x)) \\ &\quad = 2(\mu'(x) - \mu(x)) \end{aligned}$$

- Even $\mu'(x) - \mu(x)$ units are left in order to pay the elementary operations of both rotations (comparisons and pointers manipulation), unless $\mu'(x) = \mu(x)$ …

Splay trees

– We see that in this case ($\mu'(x) = \mu(x)$) the amount needed to preserve the invariant $^{(*)}$ is negative, therefore we preserve the invariant “for free”:

$$\left.\begin{array}{l}\mu'(x) = \mu(x) \\ \mu'(x) + \mu'(y) + \mu'(z) \geq \mu(x) + \mu(y) + \mu(z)\end{array}\right\} \Longrightarrow \text{contradiction}$$

In effect: $\mu(z) = \mu'(x) = \mu(x)$

$$\Rightarrow \quad \mu(x) = \mu(y) = \mu(z)$$

Then: $\mu'(x) + \mu'(y) + \mu'(z) \geq 3\mu(z) = 3\mu'(x)$

$$\Rightarrow \mu'(y) + \mu'(z) \geq 2\mu'(x)$$

And since $\mu'(y) \leq \mu'(x)$ y $\mu'(z) \leq \mu'(x)$ then

$$\mu'(x) = \mu'(y) = \mu'(z)$$

And since $\mu(z) = \mu'(x)$ then

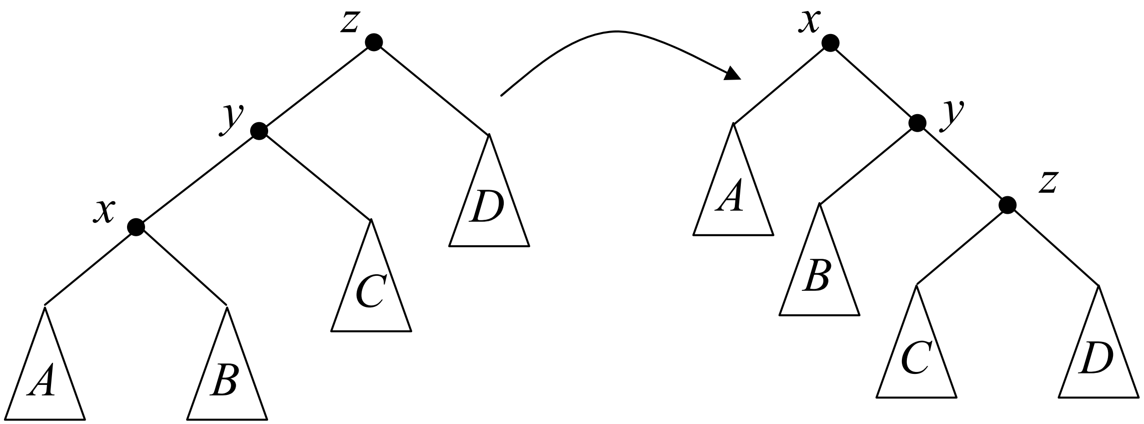
$$\mu(x) = \mu(y) = \mu(z) = \mu'(x) = \mu'(y) = \mu'(z)$$

Splay trees

But $\mu(x) = \mu(y) = \mu(z) = \mu'(x) = \mu'(y) = \mu'(z)$ is impossible by definition of μ and μ'…

- » In effect, if $a = |S(x)|$ and $b = |S'(z)|$, then we would get

  $\lfloor \log a \rfloor = \lfloor \log (a + b + 1) \rfloor = \lfloor \log b \rfloor$



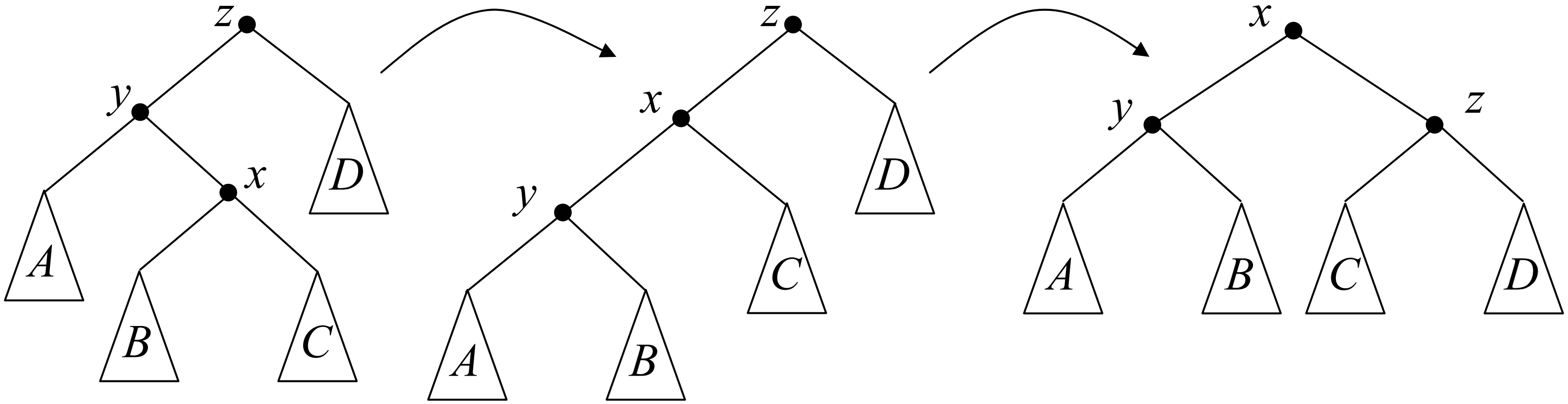
  and assuming that $a \leq b$ (the other case is similar),

  $\lfloor \log (a + b + 1) \rfloor \geq \lfloor \log 2a \rfloor = 1 + \lfloor \log a \rfloor > \lfloor \log a \rfloor$

  Therefore we reach a contradiction.

Splay trees

- **Case 3:** *x* and *y* are one right child and the other left child
  - Execute rotate(*x*) twice



  - As in the previous case, the prize of these 2 rotations to preserve the invariant is not greater than 3(μ'(*x*) – μ(*x*))
  - The proof is similar

Splay trees

- In summary:
  - The prize of each *splay* operation is bounded by $3\lfloor \log n \rfloor + 1$, with n the number of keys in the tree
  - The reminder operations can be implemented with a constant number of *splays* plus a constante number of elementary operations (comparisons and pointers assignments)

  - Therefore: the total time required to execute a sequence of m operations with a self-organizing tree is in $O(m \log n)$, with n the number of insertion and append operations

Splay trees

- Implementation details
  - Even if *splay* was described as a bottom-up operation, the most convenient implementation achieves it top-down
    - While we are searching a key, or the position to insert it, we can proceed doing *splay* along the path
    - That implementation is called *top-down splay trees*
    - Can be found for instance in M.A. Weiss book (*Data Structures and Algorithm Analysis in Java*, 2nd Ed., Addison-Wesley, 2007)

Outline

A short course in data structures analysis…

- Worst case analysis:
  - *Red-black trees*
- Average case analysis:
  - *Lexicographical trees*
  - *Skip lists*
- Amortized analysis
  - *Splay trees*
- Data structures for computational biology
  - *Suffix trees*

➡ … and also an intermediate course in *dictionary* abstract data type

Data structures for computational biology

- Basic problem: exact string matching
  - Given a ***text*** or string with m characters

    $$S = `t_1\ t_2\ \ldots\ t_m'$$

    and a ***pattern*** with n characters ($m \geq n$)

    $$P = `p_1\ p_2\ \ldots\ p_n'$$

    finding one or, more generally, all the occurrences of P in S (i.e., finding the position(s) of P in S)

  - Critical statement to measure the efficiency of possible solutions:
    - number of comparisons between pairs of characters

Data structures for computational biology

- Importance of the problem:
  - Indispensable in a large number of applicacions:
    - text editors and processors,
    - utilities like *grep* in unix,
    - information retrieval,
    - searching in catalogs of digital libraries,
    - internet searching,
    - news readers in internet,
    - electronic journals,
    - telephone directories,
    - electronic encyclopedias,
    - searching in DNA / RNA sequences databases,
    - …
  - Well-solved problem in some particular cases but…

Data structures for computational biology

"We used GCG
(a very popular interface to search DNA and proteins in data banks;
http://www-biocomp.doit.wisc.edu/gcg.shtml)
to search in Genbank
(the largest database of DNA in USA;
http://www.ncbi.nlm.nih.gov/Genbank/)
a string (pattern) with 30 characters
(small size problem for that application domain)
and it took 4 hours using a local copy of the database to determine that the pattern did not appear in the database…"

"We repeted later the same test using Boyer-Moore algorithm and the search took less than 10 minutes (and most of time was due to the movement of data from disk to memory, since the actual search took less than 1 minute)…"

D. Gusfield's book:

Algorithms on Strings, Trees, and Sequences. Computer Science and Computational Biology, Cambridge University Press, 1997.

Data structures for computational biology

- Importance of the problem (cont.)
  - String matching problems have not yet a so *efficient* and *universal* solution making the researchers lose the interest in the problem.
    - Databases’ size will continue growing up and string matching will always be a necessary subtask in many applications.
  - In addition, the problem and the known or new solutions are interesting in itself.
    - Solutions for the problem can give new ideas to solve more difficult problems related with strings and with important practical interest.

Data structures for computational biology

- Direct (naïve) method

```
function substring(S,P:string; m,n:nat) ret natural
{Returns r if the 1st occurrence of P in S starts in
 position r (i.e. r is the smallest integ. such that $S_{r+i-1}=P_i$
 for i=1,2,…,n), and returns 0 if P is not substring of S}
variables ok:boolean; i,j:natural
begin
  ok:=false; i:=0;
  while not ok and i≤m-n do
    ok:=true; j:=1;
    while ok and j≤n do
      if P[j]≠S[i+j] then ok:=false else j:=j+1 fi
    od;
    i:=i+1;
  od;
  if ok then return i else return 0 fi
end
```

Data structures for computational biology

- Analysis of direct method:
  - Tries to find pattern P at each position of S.
  - Worst case:
    - Makes n comparisons at each position to test if it has found the pattern.

      E.g.: S = ‘aaaaaaab’ P = ‘aaab’

    - Total number of comparisons is

$$\Omega(n(m-n))$$

i.e., $\Omega(nm)$ if m is substantially greater than n.

Data structures for computational biology

- A pleiade of methods that significatively improve the cost of the previous direct method
  - Knuth-Morris-Pratt (1977), $O(m)$ (with m = length of text)
  - Boyer-Moore (1977), $O(m)$ (with m = length of text), in practice, sublinear
  - Etc.
  - For instance, the book (& applets) by Charras and Lecroq

    Handbook of extact string-matching algorithms

    http://www-igm.univ-mlv.fr/%7Elecroq/string/string.pdf

    the applets:

    http://www-igm.univ-mlv.fr/%7Elecroq/string/

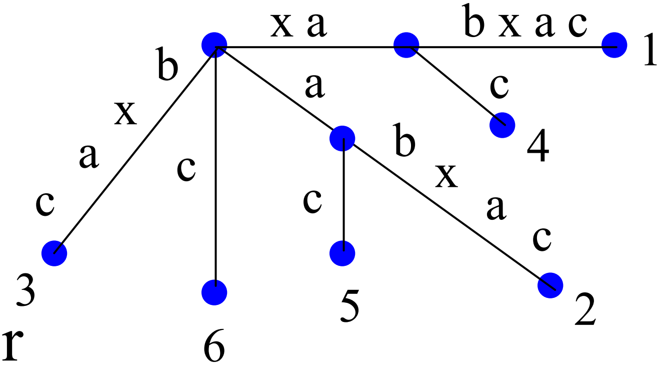
Suffix trees

- What is a suffix tree?
  - A data structure useful to store a string with important pre-processed information about its internal structure.
  - That information is useful, for instance, to solve the the exact string matching problem in linear time on the pattern length:
    - Let S be a text with length m
    - It is pre-processed (a suffix tree is built) in time $O(m)$
    - To search a pattern P of length n takes $O(n)$.
      This bound is not reached by KMP neither by BM ($O(m)$)
  - It is also useful for other more complex problems, like for instance:
    - Given a set of texts $\{S_i\}$, find if P is substring of any S_i
    - Inexact string matching problems…

Suffix trees

- Definition: suffix tree for a string *S* of length *m*
  - Rooted tree with *m* leaves numbered from 1 to *m*
  - Each internal node (except the root) has at least 2 children
  - Each edge is labelled with a non-empty substring of *S*
  - 2 edges hanged from the same node cannot have labels starting with the same character
  - For each leaf *i*, the concatenation of the labels in the path from the root to *i* gives the suffix of *S* that starts in position *i* of *S*.

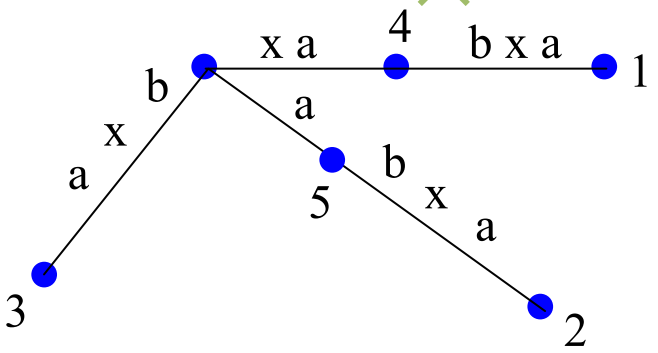
S = xabxac



Suffix trees

- A problem…
  - The definition does not guarantee that a suffix tree exists for any string *S*.
  - If a suffix of *S* is also the preffix of another suffix of *S*, then the path in the tree for the first suffix would not end in a leaf.
  - Example:

S = xabxa~~€~~



  - Solution: to add a terminator character, *S* = xabxa€

Suffix trees

- Terminology:
  - *Label of a node*: sorted concatenation of the labels in the edges of the path from the root to that node
  - *Depth in the string* of a node: number of characters in the *label of the node*

Suffix trees

- Solution to the substring problem:

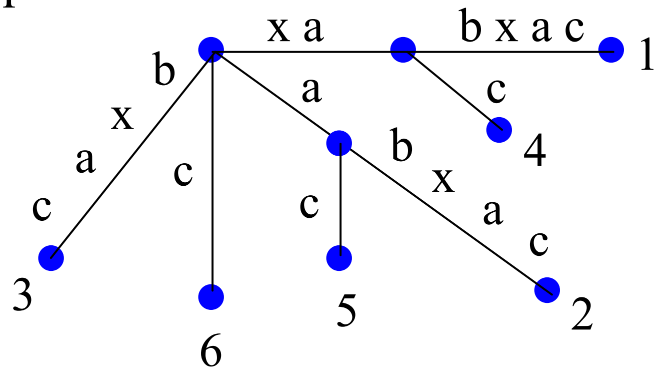
  To find all occurrences of *P*, with length *n*, in a string *S*, with length *m*, in *O*(*n* + *m*) time:

  - Build a suffix tree for the string *S*, in *O*(*m*) time.
  - Match the characters of *P* along the unique path in the suffix tree of *S* until

    a) either *P* is exhausted or

    b) no more matches are possible.

S = xabxac



  - In case (b), *P* is not in *S*.
  - In case (a), every leaf in the subtree below the point of the last match is numbered with a starting location of *P* in *S*, and there are no more.

Suffix trees

- Explanation of case (a):
  - *P* occurs in *S* starting at location *j* if and only if *P* occurs as a prefix of *S*[*j*..*m*].
  - But this happens if and only if string *P* labels an initial part of the path from the root to leaf *j*.
  - It is the initial path that will be followed by the matching algorithm.
  - The matching path is unique because no two edges out of a common node can have edge-labels beginning with the same character.
- And, because we have assumed a finite alphabet, the work at each node takes constant time and so the time to match *P* to a path is proportional to the length of *P*, $O(n)$.

Suffix trees

- Cost of locating all the occurrences of the pattern in the string, in case (a):
  - If P fully matches some path in the tree, just traverse the subtree below the end of the matching path, collecting position numbers written at the leaves.
  - Since every internal node has at least 2 children, the number of leaves encountered is proportional to the number of edges traversed, so the time for the traversal is $O(k)$, if k is the # of occurrences of P in S.
  - Thus the cost to find all locations of the pattern is $O(n + m)$, i.e., $O(m)$ to build the suffix tree and $O(n + k)$ for the search.

Suffix trees

- Cost (continuation):
  - That is the same cost of previous algorithms (for instance, KMP), but:
    - In that case (KMP) we needed a $O(n)$ time for preprocessing P and then a $O(m)$ time for the search.
    - Now, we spend $O(m)$ time in the preprocessing and then $O(n + k)$ in the search, where k is the number of occurrences of P in S.
  - If only one location of P in S is needed, the search can be reduced from $O(n + k)$ to $O(n)$ by adding to every node (in the preprocessing) the number of one of the leaves of its subtree.
  - There exists another algorithm that uses suffix trees to solve the same problem, that needs $O(n)$ time for preprocessing and $O(m)$ time for the search (i.e., exactly like KMP).

Suffix trees

- How to build a suffix tree:
  - We see the ideas behind one of the three linear (on the text length) methods known to build the tree
    - Weiner, 1973: “the algorithm of 1973”, according to Knuth
    - McCreight, 1976: more in space
    - Ukkonen, 1995: as efficient as McCreight’s and “easier” to understand (14 pages in D. Gusfield’s book…)
  - But first we see a *naïve* method, with quadratic cost on the string length (and really easy, i.e., a single slide).

Suffix trees

- Building suffix tree for string *S* (with length *m*) in quadratic time:
  - Create tree N_1 with a single edge from root to a leaf numbered with 1, and label *S*€, i.e., *S*[1..*m*]€.
  - Build tree N_{i+1} (up to N_m) by adding suffix *S*[*i*+1..*m*]€ to N_i:
    - Starting from root of N_i, find the longest path whose label matches a prefix of *S*[*i*+1..*m*]€ (as when we search a pattern in a tree), and then:
      - either the path finishes at a node, *w*, or
      - we are "in the middle" of the label of edge (u,v); in this case, split the edge in 2 (through that point) inserting a new node, *w*
    - Create a new edge (*w*,*i*+1) from *w* to a new leaf and label it wih the final part of *S*[*i*+1..*m*]€ (that was not found in N_i).

Suffix trees

- A linear algorithm to build a suffix tree (Ukkonen, 1995)
- About the way of presenting it:
  - First, we present it in the simplest way, even very inefficient.
  - Then, its efficiency can be improved with several tricks and some common sense.
- The method: build a sequence of *implicit suffix trees*, and finally transform the last of them in the suffix tree of S
  - *Implicit suffix tree*, I_m, for string S: it is obtained from the suffix tree for S€ by removing every copy of the terminal symbol € from the edge labels, then removing any edge that has no label, and then removing any node that does not have at least 2 children.
  - The *implicit suffix tree*, I_i, for a prefix $S[1..i]$ is similarly defined by taking the suffix tree for $S[1..i]$€ and deleting things as above.

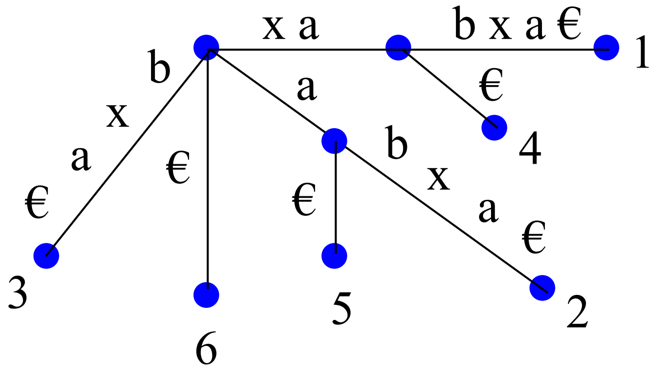
Suffix trees

- The *implicit suffix tree* for a string *S*:
  - Has less leaves than the suffix tree for *S*€ if and only if at least a suffix of *S* is prefix of another suffix of *S* (symbol € was added just to avoid that situation).

S = xabxa€

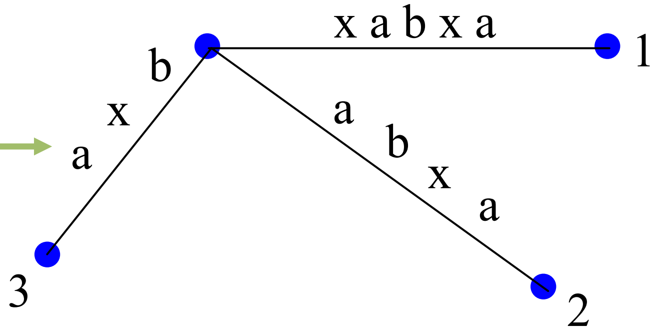
Suffix xa is prefix of suffix xabxa, and also suffix a is prefix of abxa.

Then, in the suffix tree of *S*, the edges leading to leaves 4 and 5 are labelled only with €.



If we delete those edges we get 2 nodes with only 1 child, they are also deleted.

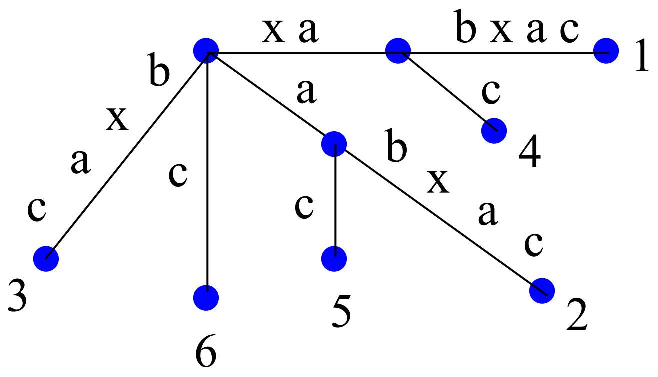
In the implicit tree, it may happen that there is not a leaf for each suffix of *S*, thus it has less information than the suffix tree.



Suffix trees

– On the other hand, if *S* ends with a character that does not appear in the rest of *S*, then the implicit suffix tree of *S* has a leaf for every suffix and therefore it is in fact a suffix tree.

S = xabxac€



Suffix trees

- The algorithm (in its unefficient version, $O(m^3)$)

```
algorithm Ukkonen_high_level(S:string; m:nat)
begin
  build tree I_1;
  for i:=1 to m-1 do
    for j:=1 to i+1 do
      find the end of the
        path from the root
        labelled with S[j..i]
        in the present tree;
      if needed then
        extend that path with
        character S(i+1) to
        assure that string
        S[j..i+1] is in the tree
    od
  od
end
```

→ It is an edge labelled with $S(1)$

Extension j: string S[j..i+1] is added, for j=1..i.

Finally (ext. j+1), string S(i+1) is added

Phase i+1: computation of tree I_{i+1} from tree I_i.

Phase i+1 has i+1 extensions

Remember that I_i is the implicit suffix tree for the prefix S[1..i].

Suffix trees

- Details on extension *j* of phase *i*+1:
  - Find *S*[*j*..*i*] = β in the tree and when the end of β is reached, try that suffix β*S*(*i*+1) is also in the tree; to do that, three cases can arise:

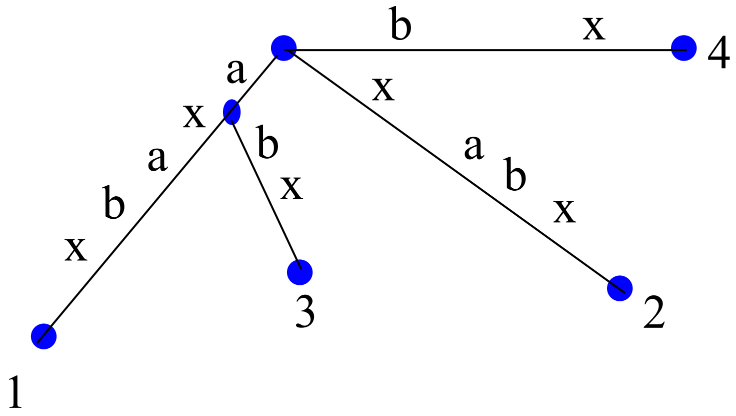
    [Rule 1] If β ends in a leaf: add *S*(*i*+1) at the end of the label of the edge from which hangs that leaf.

    [Rule 2] If no path from the end of string β starts with *S*(*i*+1), but there is at least a labelled path starting at the end of β, a new edge to a new leaf (labelled with *j*) is created hanging from that point and labelled with *S*(*i*+1).
    If β ends in the middle of a label of an edge, a new node must be inserted, splitting the edge, and hanging as a new leaf. The new leaf is labelled with *j*.

    [Rule 3] If there is a path at the end of β starting with *S*(*i*+1), the string β*S*(*i*+1) was already in the tree. Nothing to do.

Suffix trees

- Example: S = axabx

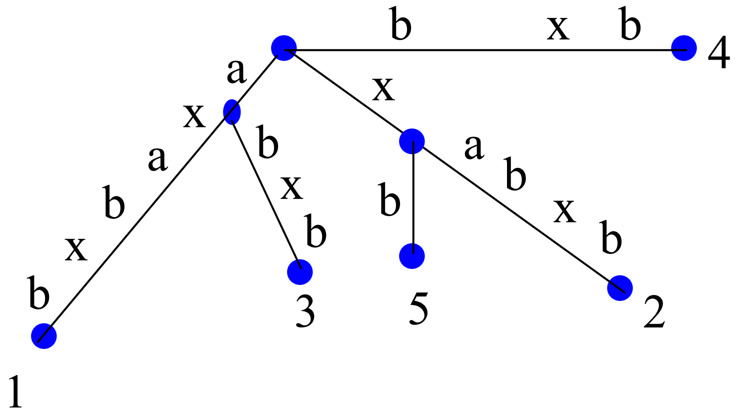


It is an implicit suffix tree for S.

The first 4 suffixes end in a leaf.

The last one, x, ends in the middle of an edge.

If we add b as the sixth char. of S, the first 4 suffixes are extended using rule 1, the 5th one using rule 2, and the 6th with rule 3.

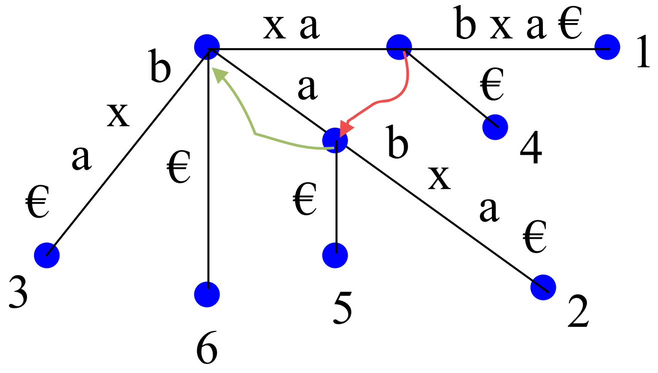


Suffix trees

- Cost of this first version:
  - Once we reach the end of suffix β of $S[1..i]$ we need a constant order time for the extension (to assure that suffix $\beta S(i+1)$ is in the tree).
  - Then, the key point is to find the ends of all the $i+1$ suffixes of $S[1..i]$.
  - The easiest way:
    - To find the end of β in $O(|\beta|)$ time, going down from the root;
    - thus, extension j in phase $i+1$ takes $O(\mathrm{i}+1-j)$ time;
    - Then, tree I_{i+1} can be generated from I_i in $O(i^2)$, then I_m is created in $O(m^3)$

Suffix trees

- Reducing the cost…: *pointers to suffixes*
  - Let *v* be an internal node with label (from the root) *x*α (i.e., a character *x* followed by a string α) and another node *s*(*v*) with label α, then a pointer from *v* to *s*(*v*) is called a pointer to suffix



  - Special case: if α is the empty string then the pointer to suffix from an internal node with label *x*α points to the root node
  - Root node is not consider as “internal”, then no pointer to suffix goes out of the root

Suffix trees

- **Lemma 1**: if we add (in the algorithm) in extension j of phase i+1 to the present tree an internal node v with label $x\alpha$, then, either:
  - the path labelled with α already ends at an internal node, or
  - due to extension rules, an internal node will be created at the end of string α in extensión j+1 of phase i+1
- **Corollary 1**: every internal node created in Ukkonen's algorithm will be the origin of a pointer to suffix at the end of the next extension
- **Corollary 2**: for every implicit suffix tree I_i (in the algorithm), if an internal node v is labelled with $x\alpha$, then there is a node $s(v)$ in I_i labelled with α

(proofs: D. Gusfield's book)

Suffix trees

- Phase i+1, extension j (for j=1..i+1): find suffix $S[j..i]$ of $S[1..i]$
  - the naïve version traverses a path from the root,
  - it can be simplified by using pointers to suffixes…
  - first extension (j=1):
    - the end of string $S[1..i]$ must be at a leaf of I_i because it is the longest string of the tree;
    - it is enough with storing a pointer to the leaf that corresponds with the whole string $S[1..i]$;
    - in that way, we have a direct access to the end of suffix $S[1..i]$, and the addition of character $S(i+1)$ is solved using rule 1, with constant order cost in time.

Suffix trees

- Second extension (j=2): find the end of $S[2..i]$ in order to append $S(i+1)$.
  - Let $S[1..i] = x\alpha$ (with α empty or not) and let (v,1) be the edge that arrives to the leaf 1.
  - We need to find the end of α in the tree.
  - Node v is either the root or an internal node of I_i
    - If v is the root, to reach the end of α we need to go down the tree following the path α (as in the naïve algorithm).
    - If v is internal, by Corollary 2, there is pointer to suffix from v to $s(v)$.

      Moreover, since $s(v)$ has a label that is prefix of α, the end of α must be in the subtree of $s(v)$.

      Then, in the search of the end of α, we do not need to traverse the whole string, but we can start the path at $s(v)$ (using the pointer to suffix).

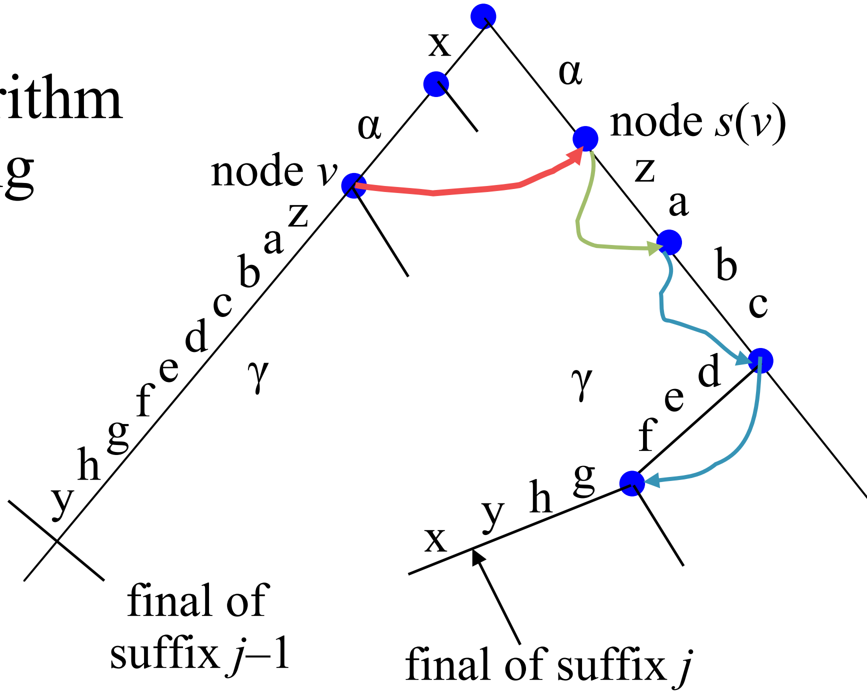
Suffix trees

- The other extensions from $S[j..i]$ to $S[j..i+1]$ with $j > 2$:
    - Starting from the end of $S[j–1..i]$ (where we arrived in previous extension) move upward a node to reach either the root or an internal node v, where a pointer to suffix starts that goes to $s(v)$.
    - If v is not the root, follow the pointer to suffix and go down in the subtree of $s(v)$ until the end of $S[j..i]$, and make the extension with $S(i+1)$ according with the extension rules.

- Obtained cost: for the moment… the same one, but a trick is possible to reduce it to $O(m^2)$

Suffix trees

- **Trick no. 1**:
  - In extension j+1 the algorithm goes down from $s(v)$ along the substring, let be γ, until the end of $S[j..i]$
  - Direct implementation: $O(|\gamma|)$
  - It can be reduced to O(# nodes in the path):



    - Let $g = |\gamma|$
    - The 1st character of γ must appear in 1 (and only 1) edge of those starting from $s(v)$; let g' be the # characters of that edge
    - If $g' < g$ then we can jump to the final node of the edge
    - Make $g=g-g'$, assign the position of g'+1 to a variable h and continue going down

Suffix trees

- Terminology: *depth* of a node is the number of nodes in the path from the root to that node
- **Lemma 2**: Let $(v,s(v))$ be a pointer to suffix followed in Ukkonen's algorithm. At that point, the depth of v is, at most, one unit more than the depth of $s(v)$.
- **Theorem 1**: using trick 1, the time cost of every phase of Ukkonen's algorithm is in $O(m)$.
- **Corollary 3**: Ukkonen's algorithm can be implemented using pointers to suffixes to get a cost in $O(m^2)$ time.

(proofs: D. Gusfield's book)

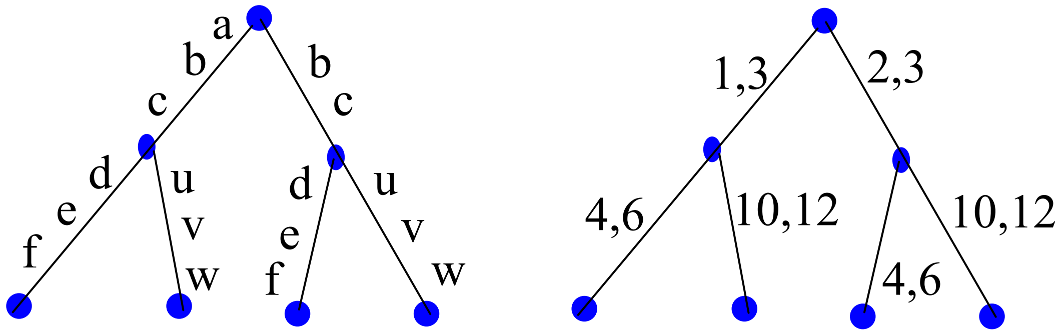
Suffix trees

- Problem to reduce the cost below $O(m^2)$:
  - Storing the tree can require $\Theta(m^2)$ in space:
    - Labels of edges can include $\Theta(m^2)$ characters
    - Example: S = abcdefghijklmnopqrstuvwxyz
      Every suffix starts with a different letter, then 26 edges hang from the root, and each one is labelled with a complete suffix, then we require 26 × 27/2 characters in total
  - We need a different way to store the labels…

Suffix trees

- Label compression
  - Store a *pair of indexes*: start and end of the substring in the string *S*
  - The cost to find the characters using the positions is constant (store a copy of *S* in a direct access structure)

S = abcdefabcuvw



  - Maximum number of edges: 2m – 1, then the cost in space to store the tree is in $O(m)$

Suffix trees

- Observation 1: remember Rule 3

> Details about extension *j* in phase *i*+1:
>
> Search *S*[*j*..*i*] = β in the tree and when reached the end of β we need to include suffix β*S*(*i*+1) in the tree, three cases can arise:
>
> …
>
> **[Rule 3]** If there is a path at the end of β starting with *S*(*i*+1), the string β*S*(*i*+1) was already in the tree. Nothing to do.

- If Rule 3 is applied in any extension *j*, rule 3 will also be applied in the other extensions since *j*+1 to *i*+1.
- Moreover, a pointer to suffix is added only after applying Rule 2…

> **[Regla 2]** If no path from the end of string β starts with *S*(*i*+1), but there is at least a labelled path starting at the end of β, a new edge to a new leaf (labelled with *j*) is created hanging from that point and labelled with *S*(*i*+1). If β ends in the middle of a label of an edge, a new node must be inserted, splitting the edge, and hanging as a new leaf. The new leaf is labelled with *j*.

Suffix trees

- **Trick no. 2**:
  - Phase *i*+1 must finish just after the first time that Rule 3 is applied for an extension.
  - The "extensions" in phase *i*+1 after the first application of Rule 3 will be called *implicit* (there is nothing to do in them, then we do not execute them).
  - On the contrary, an extension *j* in which we explicitly find the end of *S*[*j*..*i*] (then, either Rule 1 or Rule 2 is applied), is called *explicit*.

Suffix trees

- Observation 2:
  - Once a leaf is created and labelled with *j* (due to the suffix of *S* that starts in position *j*), it remains as a leaf during all the execution of the algorithm.
  - In effect, if there is a leaf labelled with *j*, then Rule 1 of extension will apply in all the subsequent phases to the extension *j*.

[Regla 1] If β ends in a leaf: add $S(i+1)$ at the end of the label of the edge from which hangs that leaf.

  - Then, after creating leaf 1 in phase 1, in every phase *i* there is an initial sequence of consecutive extensions (starting from extension 1) where either Rule 1 or Rule 2 are applied.
  - Let j_i be the last extension in the above sequence.

Suffix trees

- Since in each application of Rule 2 a new leaf is created, from observation 2 follows that $j_i \leq j_{i+1}$,
  - i.e., the length of the initial sequence of extensions where rules 1 or 2 are applied does not reduce in the next phases;
  - then, we can apply the following trick in the implementation:
    - in phase i+1 skip all explicit extensions from 1 to j_i (then it takes a constant time to perform implicitly all these extensions);
    - we see it in detail now…
      (remember that the label of an edge is represented with 2 indexes, p, q, specifying the substring $S[p..q]$, and that the edge to a leaf in tree I_i will have $q = i$, then in phase i+1 q will be incremented to i+1, indicating the character $S(i+1)$ appended at the end of each suffix)

Suffix trees

- **Trick no. 3**:
  - In phase i +1, when an edge to a leaf is created and should be labelled with indexes (p, i+1) representing the string $S[p..i+1]$, instead of doing that, do label it with (p, e), where e is a symbol denoting the "present end"
  - e is a global index that takes the value i+1 once in each phase
  - In phase i+1, since we know that Rule 1 will be applied at least in the extensions from 1 to j_i, it is not needed additional work to implement these j_i extensions; instead of that, with a constant cost, we increment variable e and later we perform the needed work for extensions after j_i+1

Suffix trees

- Cost: Ukkonen's algorithm, using pointers to suffixes and implementing tricks 1, 2 and 3, builds the implicit suffix trees I_1 to I_m in $O(m)$ time.

  (details: D. Gusfield's book)

- The implicit suffix tree I_m can be transformed in the final suffix tree in $O(m)$ time:
  - Add the terminal symbol € at the end of *S* and continue the execution of Ukkonen's algorithm with that character
  - Since no suffix is now a prefix of another suffix, the algorithm creates an implicit suffix tree such that every suffix ends at a leaf (therefore, it is also a suffix tree)
  - Finally, change the value of index *e* (in edges to leaves) into *m*

Suffix trees

First applications of suffix trees:

- [P1] Exact string matching
  - If the pattern, $P(1..n)$, and the text, $S(1..m)$, are known at the same time, the cost using suffix trees is equal to using KMP or BM: $O(n+m)$
  - If we need to search several patterns in the same text, after a pre-processing (creation of the suffix tree of the text) with cost $O(m)$, each search of the k occurrences of a pattern P in S takes $O(n+k)$; on the other hand, algorithms based on pre-processing the pattern (as KMP) take $O(n+m)$ for every search
  - This problem was the origin of suffix trees

Suffix trees

- [P2] Exact matching of a set of patterns
  - There exists a (non trivial, section 3.4 in Gusfield's book) algorithm by Aho and Corasick for searching all (k) occurrences of a set of patterns with total length n in a text with length m with cost $O(n+m+k)$ in time
  - Using a suffix tree we get exactly the same bound

Suffix trees

- [P3] Searching a substring in a set of texts
  - Example: identification of mortal remains (used for instance with U.S. military personnel)
    - A small mitochondrial DNA interval of each person (database of military personnel) is stored (“sequenced”)
    - The selected interval is such that:
      - It can be easily and reliably isolated using a PCR (“polymerase chain reaction”: process used to duplicate a DNA interval; it is to genes what Gutenberg's printing press was to the written word)
      - it is a highly variable string (thus it is a “nearly unique” identifier of a person)
    - To identify a person, mitochondrial DNA is extracted from the remains of persons who have been killed (or in many cases a substring, if the conditions of the remains do not allow the complete extraction) and matched against the database of strings of personnel (in fact, the longest common substring of the extracted interval and the DB intervals)

Suffix trees

- Solution to [P3]: a *generalized suffix tree*
  - It is used to store the suffixes of a set of texts
  - Conceptually simple way to build it:
    - add a different ending character (not belonging to the alphabet of the texts) to each text in the set,
    - concatenate all the resulting strings, and
    - generate the suffix tree for the whole resulting string.
  - Consequences:
    - The obtained tree has a leaf per each suffix of the concatenated string and it is built in linear time on the sum of the lengths of the concatenated texts
    - The numbers (labels) of the leaves can be substituted for a pair of numbers: one identifying the text to which it belongs and the other one the initial position in that text

Suffix trees

- Problem with the method:
  - the tree stores suffixes that cover more than one of the original texts, thus they are not valid suffixes
- Solution:
  - Since the ending character of each text is different and does not appear in the original texts, the label of all the path from the root to any internal node is a substring of one of the original texts
  - Then, the suffixes that are not valid are always in paths from the root to the leaves, thus reducing the second index (that one that is marking the end of the substring) from the label of the edge that goes to each leaf, all the non-valid suffixes can be deleted (those covering more than a text of the set)

Suffix trees

– Smart implementation:
  - Simulate the above method without making the concatenation of the strings
  - With 2 different string S_1 and S_2:

    1st build the suffix tree of S_1 (assuming it has an ending character)

    2nd starting from the root of the previous tree, make the matching of S_2 with a path in the tree until a difference appears

    » suppose that the first i characters of S_2 match

    » then the tree stores all the suffixes of S_1 and $S_2[1..i]$

    » essentially, we executed the first i phases of Ukkonen's algorithm for S_2 using the tree of S_1

    3rd continue Ukkonen's algorithm with that tree for S_2 from phase i+1

    » at the end, the tree stores all the suffixes of S_1 and S_2 but without "synthetic" suffixes

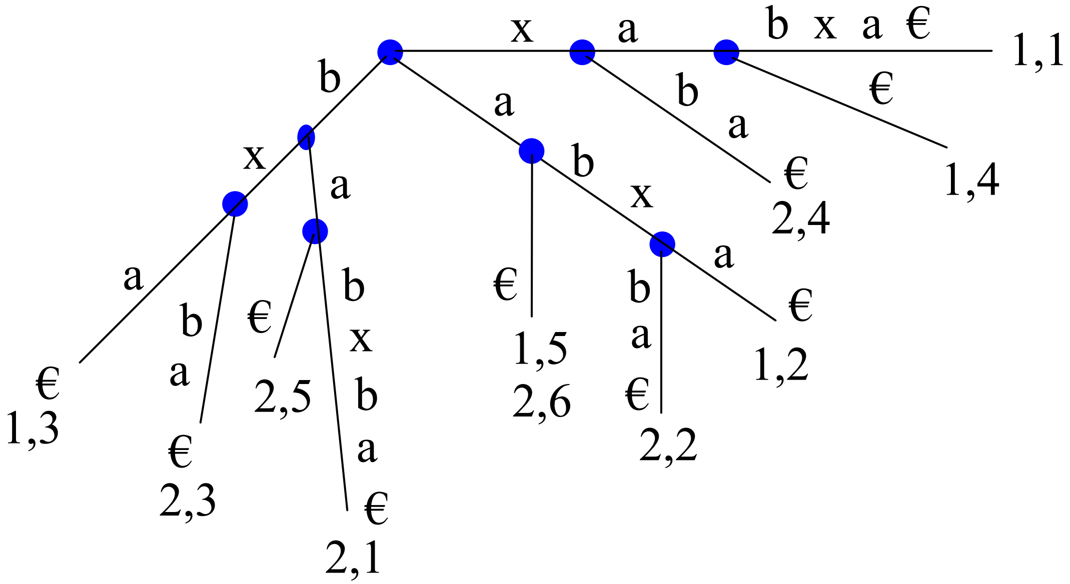
Suffix trees

- For more than 2 strings: repeat the process for all them
  - The generalized suffix tree for all of them is created in linear time on the sum of the lengths of all the strings

- Small problems:
  - The compressed labels of edges can refer to several strings ➔ we must add a symbol to every edge (now they are 3)
  - It can be identical suffixes in 2 (or more) strings, in that case a leaf will represent all the strings and starting positions of the corresponding suffix

Suffix trees

– Example: result of adding *babxba* to the tree of *xabxa*



Suffix trees

- Summarizing, to solve [P3]:
  - Build the generalized suffix tree of strings S_i in the DB in $O(m)$ time, with m the sum of lengths of all the strings, and also in $O(m)$ in space
  - A string S with length n can be found (or its absence proved) in the DB in $O(n)$ time
    - It is done just matching that substring with a path in the tree
    - The path ends at a leaf when looking the last character of the string if and only if the string appears complete in the DB
    - If S is substring of one or several strings of the DB, the algorithm finds all the strings of the DB containing S in time $O(n+k)$ where k is the number of occurrences of the substring (it is done by traversing the subtree hanging below the path followed when matching S with the tree)
    - If S does not match with a path in the tree then neither S is in the DB nor is substring of a string in the DB; in this case, the matched path defines the longest prefix of S that is substring of a string in the DB

Suffix trees

- [P4] Longest common substring of two strings
  - Build the generalized suffix tree of S_1 and S_2
  - Every leaf of the tree represents either a suffix of one of the strings or a suffix of both strings
  - Mark every internal node *v* with 1 (respectively, 2) if there is a leaf in the subtree of *v* that represents a suffix of S_1 (respectively, S_2)
  - The label of the path of every internal node marked simultaneously with 1 and 2 is a common subtring of S_1 and S_2 and the longest among them is the longest common substring
  - Then, the algorithm has to search the node marked with both 1 and 2 that has a longest path label
  - The cost to build the tree and to search is linear

Suffix trees

- Therefore: “Theorem. The longest common substring of two strings can be computed in linear time by using a generalized suffix tree.”
  - Even though nowadays the above result seems to be easy, D. Knuth, in the 70’s, conjectured that it would not be possible to find a linear algorithm to solve this problem…

- The identification of mortal remains problem [P3], reduced to find the longest common substring of a given string being also substring of any of the strings in a database can be easily solved by extending the solution to problem [P4].

Suffix trees

- Et cetera, et cetera:
  - [P5] Recognizing DNA contamination

    Given a string S_1 and another S_2 (that can be contaminated), the problem of finding all the substrings of S_2 that occur in S_1 and that have a length greater than l.
  - [P6] Common substring to more than two strings
  - [P7] Compression of a suffix tree, using a directed acyclic graph of words
  - [P8] Inverse use: suffix tree of the searched pattern
  - … (look at D. Gusfield’s book)
  - More concrete applications in genoma projects
  - Minimal length codification of DNA…
  - Inexact pattern matching…
  - Multiple comparison of strings…