2-INF-237 Vybrané partie z dátových štruktúr

2-INF-237 Selected Topics in Data Structures

- Instructor: Broňa Brejová
- E-mail: brejova@fmph.uniba.sk
- Office: M163

- Course webpage: http://compbio.fmph.uniba.sk/vyuka/vpds/

Literature

- P. Brass. Advanced Data Structures. Cambridge University Press 2008. In library I-INF-B-67
- T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein. Introduction to Algorithms. MIT Press 2001. In library D-INF-C-1.
- D. Gusfield (1997) Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology. Cambridge University Press. In library I-INF-G-8.
- MIT course Advanced Data Structures by Erik Demaine
- Gnarley trees: website by Kuko Kováč
- Research papers
- Class notes, slides

Amortized analysis (amortizovaná analýza)

Usual worst-case analysis: analyze the running time of a single operation on the worst input.

Here: analyze the worst case for a whole sequence of operations

Definition:

If the real cost of ith operation c_i, we can say that amortized cost is $\hat{c}_i$ if $\sum_i c_i \leq \sum_i \hat{c}_i$

Example: vector (dynamic array)

Stores a sequence of elements

Supports operations

– `add(x)` : adds element x to the end of the stored sequence

– `get(i)` : get i-th element of the sequence

– `set(i,x)` : set i-th element of the sequence to value x

Implementation

Automatically growing arrray

– stores its size s and number of elements n

– when $n = s$, allocate a new array of size $2s$ and copy all elements over

Operation add for vector

```
1  void add(vector &a, dataType x) {
2    if (a.n == a.s) {
3      dataType *temp = new dataType[a.s * 2];
4      for (int i = 0; i < a.n; i++) {   // copy elements
5        temp[i] = a.a[i];
6      }
7      delete [] a.a;
8      a.a = temp;
9      a.s *= 2;
10   }
11   a.a[a.n] = x; a.n++;
12 }
```

Analysis of running time

Worst-case

`get` $O(1)$

`add` $O(n)$

Amortized

`get` $O(1)$

`add` $O(1)$

Plan:

– We will analyze amortized complexity by two different methods.

– We will consider only operation `add`.

– We will count how many times we run `temp[i] = a.a[i];`

– If this line is executed t times, running time is $O(1 + t)$.

Accounting method

- Charge $t_j(n)$ for operation o_j (its amortized cost)
- If $t_j(n) >$ cost of operation, distribute the rest to various accounts
- If $t_j(n) <$ cost of operation, charge some accounts
- All accounts have non-negative amounts at all times

For vector:

- Charge 2 for each `add` operation
- Account for each element of the array
- If no copying, store 1 in account $n - 1$, 1 in account $s - n$
- Otherwise use 1 from account i to copy i, then distribute 2 as above

Accounting method for vector

- Charge 2 for each `add` operation
- Account for each element of the array
- If no copying, store 1 in account $n-1$, 1 in account $s-n$
- Otherwise use 1 from account i to copy i, then distribute 2 as above

Prove non-negative account balances

– what is the status of accounts just before and after copying?

Conclusion

- Amortized cost 2 per `add`
- Overall for n `add` operations, line executed at most $2n$ times
- Amortized complexity $O(t_i + 1) = O(2 + 1) = O(1)$

Potential method

Let D_i be data structure after i-th operation

Potential function $\Phi(D_i)$ (analog of total amount in accounts)

Real cost of i-th operation c_i

Amortized cost of i-th operation $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$

Total amortized cost

$$\begin{aligned}\sum_{i=1}^{n} \hat{c}_i &= \sum_{i=1}^{n} (c_i + \Phi(D_i) - \Phi(D_{i-1})) \\ &= \left(\sum_{i=1}^{n} c_i\right) + \Phi(D_n) - \Phi(D_0)\end{aligned}$$

If $\Phi(D_n) \geq \Phi(D_0)$, amortized cost is upper bound on the true cost

Usually $\Phi(D_0) = 0$, so we need $\Phi(D_n)$ non-negative

Potential method for vector

- $\Phi = 2n - s$
- Just before resizing: $\Phi =$
- Just after resizing: $\Phi =$
- Actual cost of resizing: $c =$
- Amortized cost of resizing: $\hat{c} = c + \Phi_{after} - \Phi_{before} =$
- Amortized cost of increasing n: $\hat{c} = c + \Delta\Phi =$
- At the beginning assume $s = n = 0$, $\Phi(D_0) = 0$
- First `add`: increase s and n to 1, $c_i = 0$, $\Delta\Phi =$
- $\Phi(D_i) \geq \Phi(D_0) = 0$, because

Potential method for vector

- $\Phi = 2n - s$
- Just before resizing: $n = s$, $\Phi = n$
- Just after resizing: $2n = s$, $\Phi = 0$
- Actual cost of resizing: $c = n$
- Amortized cost of resizing:
  $\hat{c} = c + \Phi_{\text{after}} - \Phi_{\text{before}} = n + 0 - n = 0$
- Amortized cost of increasing n: $\hat{c} = c + \Delta\Phi = 0 + 2$
- At the beginning assume $s = n = 0$, $\Phi(D_0) = 0$
- First `add`: increase s and n to 1, $c_i = 0$, $\Delta\Phi = 1$
- $\Phi(D_i) \geq \Phi(D_0) = 0$, because $n \geq s/2$

Amortized analysis (amortizovaná analýza)

Analyze the worst case for a whole sequence of operations.

Real cost of ith operation c_i, amortized $\hat{c}_i$

We need $\sum c_i \leq \sum \hat{c}_i$

Accounting method:

Create accounts associated with values, nodes etc.

Operations with $c_i < \hat{c}_i$ store $\hat{c}_i - c_i$ in some accounts

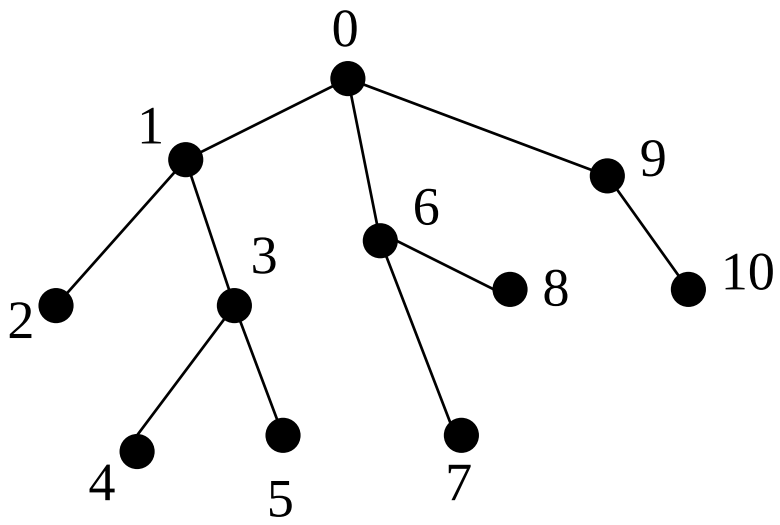
Operations with $c_i > \hat{c}_i$ use money from the accounts

Potential method:

Keep only total in all acounts, potential function $\Phi(D)$

$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$

Lowest common ancestor (LCA), najnižší spoločný predok



v is ancestor of u if it is on the path from u to the root

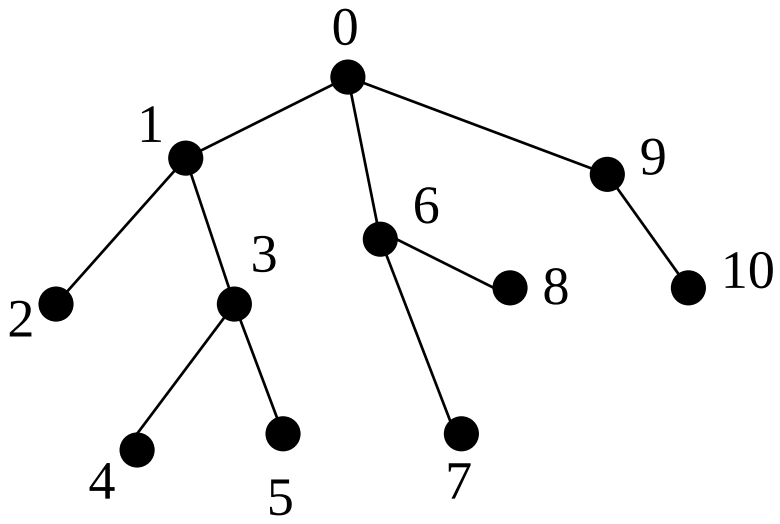
lca(u, v): node of greatest depth in ancestors$(u) \cap$ ancestors(v)

Task: preprocess tree T in $O(n)$, answer lca(u, v) in $O(1)$

Harel and Tarjan 1984, Schieber a Vishkin 1988 (Gusfield book),

Bender and Farach-Colton 2000 (this lecture)

Lowest common ancestor (LCA), najnižší spoločný predok



Task: preprocess tree T in $O(n)$, answer lca(u, v) in $O(1)$

Trivial solutions:

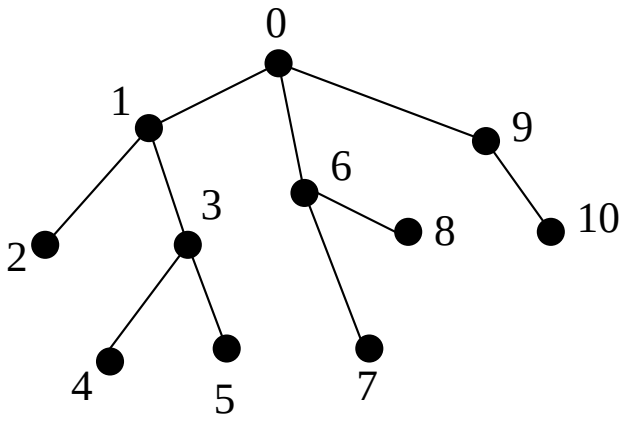
– no preprocessing, $O(n)$ time per lca

– $O(n^3)$ preprocessing, $O(n^2)$ memory, $O(1)$ time per lca

Later in the course:

– $O(\log n)$ amortized on dynamic trees (today static)

Lowest common ancestor (LCA), najnižší spoločný predok



lca(u, v): node of greatest depth which is ancestor of both u and v

Task: preprocess tree T in $O(n)$, answer lca(u, v) in $O(1)$

Range minimum query (RMQ)

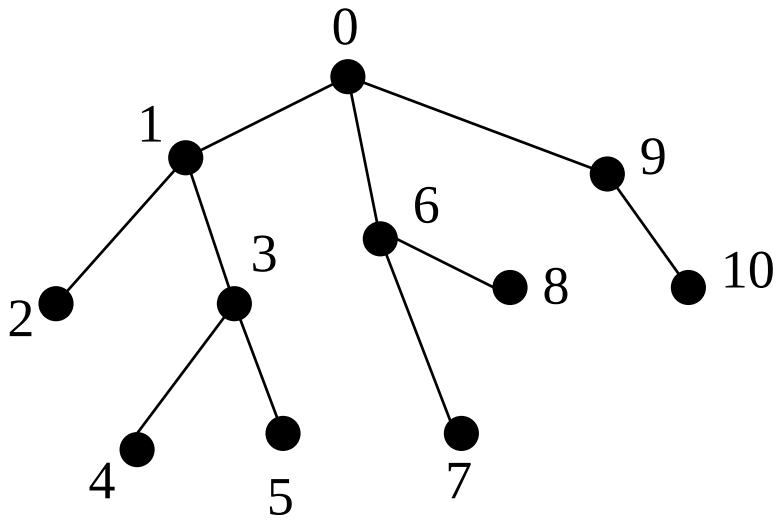
Array A of size n

$RMQ(i, j) = \arg\min_{k \in \{i, i+1, \ldots, j\}} A[k]$

Task: preprocess array A in $O(n)$, then answer RMQ(i, j) in $O(1)$

Lowest common ancestor (LCA)

Preprocess tree to arrays V, D, R



V – visited nodes

D – their depths

R – first occurrence of node in V

i:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
V:	0	1	2	1	3	4	3	5	3	1	0	6	7	6	8	6	0	9	10	9	0
D:	0	1	2	1	2	3	2	3	2	1	0	1	2	1	2	1	0	1	2	1	0

i:	0	1	2	3	4	5	6	7	8	9	10
R:	0	1	2	4	5	7	11	12	14	17	18

Lowest common ancestor (LCA)

```
1  search(root, 0);  // call recursion
2
3  void search(node v, int depth) {
4    R[v] = V.size;
5    V.push_back(v);
6    D.push_back(depth);
7    foreach child u of v {
8      search(u, depth+1);
9      V.push_back(v);
10     D.push_back(depth);
11   }
12 }
```

RMQ algorithm 3

$M[i, k]$: index of minimum in $A[i..i + 2^k - 1]$ for $k = 1, \ldots, \lfloor \lg n \rfloor$

i	0	1	2	3	4	5	6	7	8	9	10	11	12
A[i]	0	1	2	1	2	3	2	1	0	1	0	1	0
k=1	0	1	3	3	4	6	7	8	0	10	10	12	–
k=2	0	1	3	3	7	8	8	8	8	10	–	–	–
k=3	0	8	8	8	8	8	–	–	–	–	–	–	–

RMQ algorithm 3

$M[i, k]$: index of minimum in $A[i..i + 2^k - 1]$ for $k = 1, \ldots, \lfloor \lg n \rfloor$

Preprocessing:

if $A[M[i, k-1]] < A[M[i + 2^{k-1}, k-1]]$ then
$M[i, k] = M[i, k-1]$ else $M[i, k] = M[i + 2^{k-1}, k-1]$

RMQ(i, j):

let $k = \lfloor \lg(j - i + 1) \rfloor$
if $A[M[i, k]] < A[M[j - 2^k + 1, k]]$ return $M[i, k]$ else return
$M[j - 2^k + 1, k]$

Time: $O(n \log n)$ preprocessing, $O(1)$ query

LCA algorithm overview

- Compute arrays V,D,R by depth-first search in the tree
- Enumerate all possible $+1, -1$ blocks of length $m - 1$, precompute answers for all intervals in each type
- Split D into blocks of length $m = \log_2(n)/2$, precompute minimum and its index in each block (A', M'), find type of each block
- Precompute $O(n' \log n')$ data structure for RMQ in A'
- For lca(u, v) a query:
  $i = R[u], j = R[v]$, find position k of minimum in $D[i..j]$ as follows:
  – find block b_i containing i, block b_j containing j
  – compute minimum in $b_i \cap [i, j]$, $b_j \cap [i, j]$
  – compute minimum in $A'[b_{i+1} \dots b_{j-1}]$
  – find minimum of three numbers, let k be its index in D
  return $V[k]$

Lowest common ancestor (LCA)

Range minimum query (RMQ):

Alg.1 no preprocessing, $O(n)$ query

Alg.2 $O(n^2)$ preprocessing, $O(1)$ query

Alg.3 $O(n \log n)$ preprocessing, $O(1)$ query

± 1RMQ: $O(n)$ preprocessing, $O(1)$ time

split to blocks, use alg.2 within blocks, alg.3 between blocks

many blocks repeat in input – save time

LCA: $O(n)$ preprocessing, $O(1)$ time

use ± 1RMQ on array of depths in depth-first search

RMQ: want $O(n)$ preprocessing, $O(1)$ time

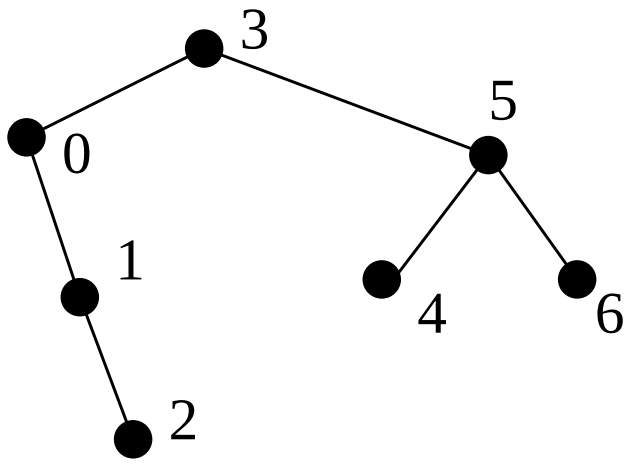
convert back to LCA!

RMQ using LCA

Cartesian tree for A: root: minimum in A (at position k)

left subtree: recursively for $A[1..k-1]$

right subtree: recursively for $A[k+1..n]$



i	0	1	2	3	4	5	6
$A[i]$	1	2	4	0	5	3	6

$A \rightarrow$ Cartesian tree in $O(n)$: add elements from left to right

min $A[i..j] = \text{lca}(i, j)$

Building a Cartesian tree

Use auxiliary value $a[-1] = -\infty$

```
 1  root = new node(-1, null);
 2  r = root;
 3  for(int i=0; i<n; i++) {
 4    while(a[r.id]>a[i]) {
 5      r = r.parent;
 6    }
 7    v = new node(i, r);
 8    v.left = r.right;
 9    r.right = v;
10    r = v;
11  }
```

Lowest common ancestor (LCA)

Range minimum query (RMQ):

Alg.2 $O(n^2)$ preprocessing, $O(1)$ query

Alg.3 $O(n \log n)$ preprocessing, $O(1)$ query

± 1RMQ: $O(n)$ preprocessing, $O(1)$ time

split to blocks, use alg.2 within blocks, alg.3 between blocks

LCA: $O(n)$ preprocessing, $O(1)$ time

use ± 1RMQ on array of depths in depth-first search

RMQ: $O(n)$ preprocessing, $O(1)$ time

use LCA on Cartesian tree

Direct method: split to blocks,

represent blocks by Cartesian trees (Fischer and Heun 2006)

Precomputing values over intervals

Operation $\circ$, compute $R_\circ(i,j) = A[i] \circ A[i+1] \circ \cdots \circ A[j]$

- Precompute all answers: $O(n^2)$ preprocessing, $O(1)$ query
- Precompute prefix "sums" $R_\circ(0,i)$

  good for groups (e.g. $\circ = +$ over Z,Q,R,etc.)

  $R_\circ(i,j) = R_\circ(0,j)) \circ R_\circ(0,i-1))^{-1}$

  Optional HW: what about multiplication?
- Precompute intervals of sizes 2^i

  combine 2 overlapping answers e.g. for min
- Segment trees: precompute non-overlapping intervals of sizes 2^i

  combine several intervals to cover each element exactly once

  good for any associative $\circ$, e.g. matrix multiplication

  also good in case of dynamic updates of the array

Segment tree

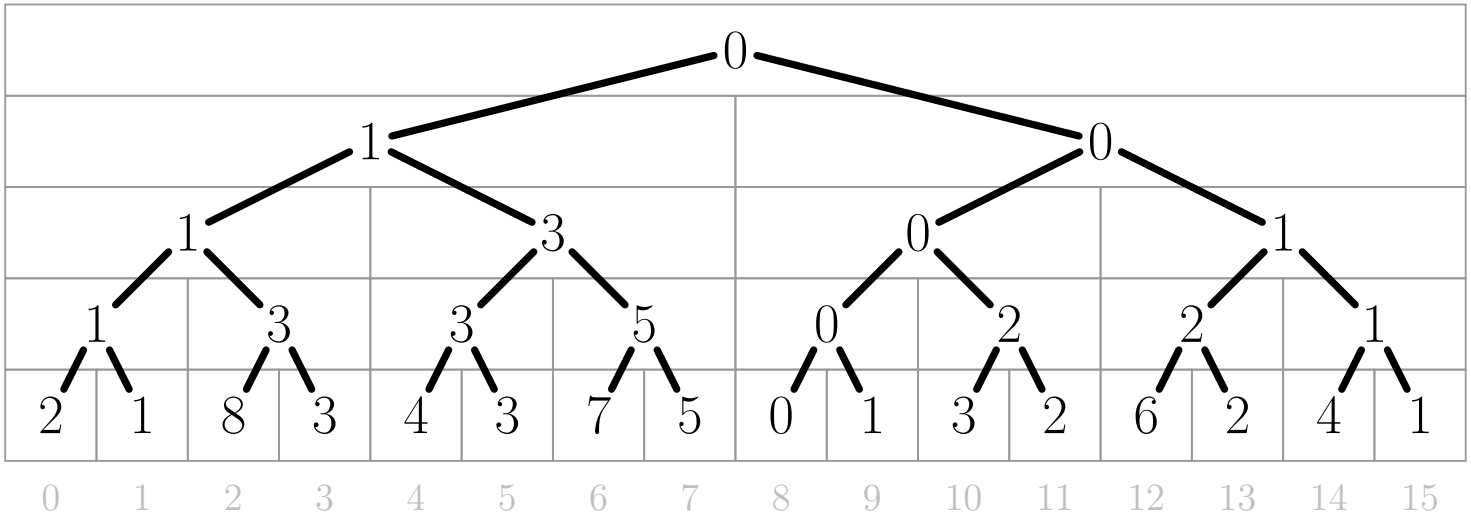
- Root correspods to interval $[0, n)$
- Leafs correspond to intervals $[i, i + 1)$
- If a node corresponds to $[i, j)$

  left child corresponds to $[i, k)$, right child to $[k, j)$

  where $k = \lfloor (i + j)/2 \rfloor$
- For each node $[i, j)$ store $R_\circ(i, j - 1) = A[i] \circ \cdots \circ A[j - 1]$
- Total number of nodes $2n - 1$, height $\lceil \lg n \rceil$

Segment tree

Example for $\circ = \min$

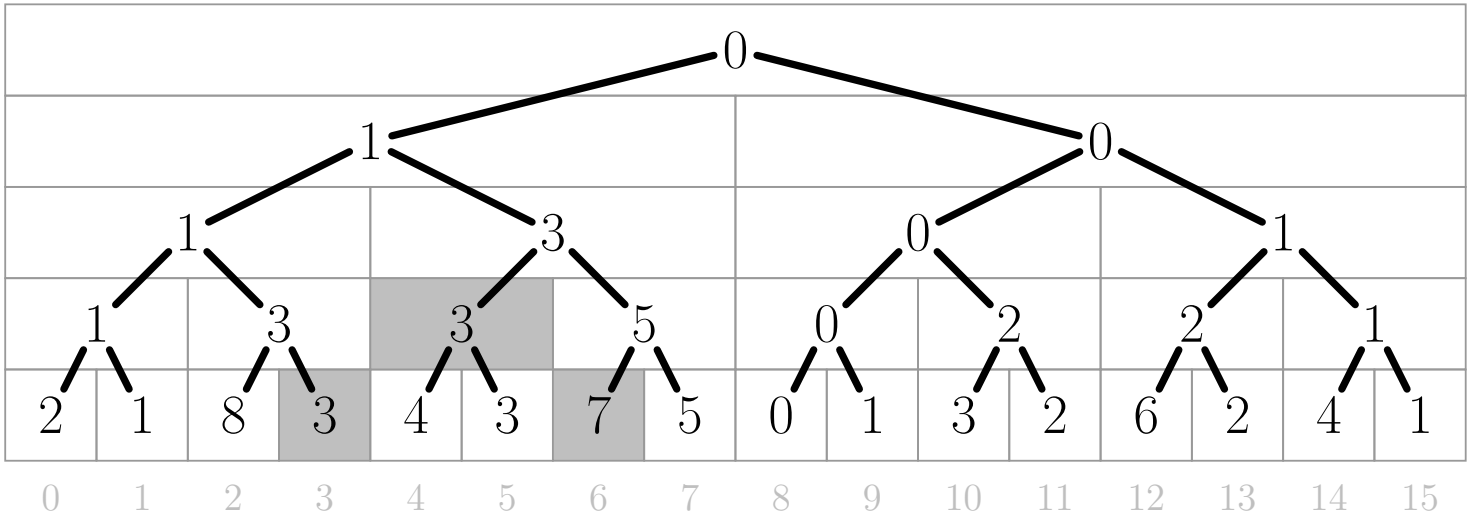


It might be more useful to store position of minimum, rather than value

The structure can be stored in an array similarly as binary heap

Canonical decomposition

Decompose query interval $[x, y)$ to a set of disjoint tree intervals



Canonical decomposition

Decompose query interval $[x, y)$ to a set of disjoint tree intervals

- Current node $[i, j)$, and its left child $[i, k)$
  invariant: $[i, j)$ overlaps with $[x, y)$
- If $[i, j) \subseteq [x, y)$, return $\{[i, j)\}$
- $R = \emptyset$
- If $[i, k)$ overlaps with $[x, y)$, recurse on left child, add to R
- If $[k, j)$ overlaps with $[x, y)$, recurse on right child, add to R
- Return R

Segment tree (summary)

- Tree of intervals, height $O(\log n)$
- Root: entire array; leaves: intervals of length 1
- Each node stores the result of operation $\circ$ on its interval
- Each internal node split into two disjoint intervals, left and right child
- Canonical decomposition: Each query interval can be written as union of $O(\log n)$ disjoint intervals, these can be found in $O(\log n)$ time
- To compute $A[i] \circ \cdots \circ A[j]$, we need $O(\log n)$ time for any associative $\circ$
- Update of an element in A can be done in $O(\log n)$ time

Priority queues

Min-heap property

A tree or a forest has the min-heap property, if the key in each non-root node $\geq$ key in its parent

Running time of heap operations

Operations	Bin. heap worst case	Fib. heap amortized	Leftist heap worst case
MakeHeap()	$O(1)$	$O(1)$	$O(1)$
Insert(H, x)	$O(\log n)$	$O(1)$	$O(\log n)$
Minimum(H)	$O(1)$	$O(1)$	$O(1)$
ExtractMin(H)	$O(\log n)$	$O(\log n)$	$O(\log n)$
Union(H_1, H_2)	$O(n)$	$O(1)$	$O(\log n)$
DecreaseKey(H, x, k)	$O(\log n)$	$O(1)$	$O(\log n)$
Delete(H, x)	$O(\log n)$	$O(\log n)$	$O(\log n)$
BuildHeap($x_1, \ldots, x_n$)	$O(n)$	$O(n)$	$O(n)$

Note: DecreaseKey and Delete require a handle of the element (pointer to node/index to array)

Applications of priority queues

- Job scheduling on a server (Insert, ExtractMin, Delete)
- Event-driven simulation (Insert, ExtractMin)
- Heapsort (BuildHeap, ExtractMin)
- Bentley–Ottmann algorithm for finding intersections of line segments (Insert, ExtractMin)
- Kruskal's and Prim's algorithms for minimum spanning tree (BuildHeap, ExtractMin, DecreaseKey)
- Dijkstra's algorithm for shortest paths (BuildHeap, ExtractMin, DecreaseKey)

Queues and Sorting

Heapsort: BuildHeap or $n \times$ Insert, then $n \times$ ExtractMin

Lower bound: on Insert+ExtractMin $\Omega(\log n)$ in the comparison model

Partial sorting: given n elements, print smallest k in the sorted order

$O(n + k \log k)$ - how?

Incremental sorting: given n elements, build a data structure, then support ExtractMin called k times, k unknown in advance

– with heaps $O(n + k \log n)$

– this is actualy $O(n + k \log k)$: check for $k \leq \sqrt{n}$ and $k \geq \sqrt{n}$

Bentley–Ottmann algorithm
for finding intersections of line segments

n line segments, k intersections

Sweepline algorithm, sweeps from left to right

Maintains priority queue of events: start of a line, end of a line, intersection

Balanced binary search tree of segments at current x-coordinate

$(2n + k)\times$Insert, $(2n + k)\times$ExtractMin

$O((n + k) \log n)$ time

Kruskal's algorithm for minimum spanning trees

Process edges from the smallest weight upwards

If edge connects 2 components, add it

Finish when we have a tree.

Instead of sorting edges:

BuildHeap, at most $m\times$ ExtractMin

Also needs Union-FindSet structure

Total time $O(m \log n)$

Prim's algorithm for minimum spanning trees

Build a tree starting from one node

For each node outside the tree keep the best cost to a node in the tree as a priority in a heap

BuildHeap, $n\times$ Extract-Min, $m\times$ DecreaseKey

$O(m \log n)$ with binary heaps, $O(m + n \log n)$ with Fibonacci heaps

Note: minimum spanning tree can be found in $O(m\alpha(m, n))$

where α is the inverse of Ackermann's function

Using Soft heaps by Chazelle 2000

Constant amortized time for insert, union, findMin, delete

But corrupts (increases) keys of up to ϵn elements

Dijkstra's algorithm for shortest paths

Heap of nodes without final distance

In each step step choose the minimum, set its distance to final, update distances of its neighbors

BuildHeap, $n\times$ Extract-Min, $m\times$ DecreaseKey

$O(m \log n)$ with binary heaps, $O(m + n \log n)$ with Fibonacci heaps

Meldable heaps

The same running times as binary heaps, but Union in $O(\log n)$

– Random meldable heap (expected time)

– Leftist heaps (worst-case time)

– Skew heaps (amortized time)

Overall structure: binary tree with min-heap property

(not always fully balanced)

Key operation is Union

Insert: Union of a single-node heap and a big heap

ExtractMin: Remove root, union its two subtrees

Delete and DecreaseKey can be done as well (more complex)

Random meldable heaps

```
1  Union(H1, H2) {
2      if (H1 == null) return H2;
3      if (H2 == null) return H1;
4      if (H1.key > H2.key) { exchange H1 and H2 }
5      // now we know H1.key <= H2.key
6      if (rand() % 2 == 1) {
7          H1.left = Union(H1.left, H2);
8      } else {
9          H1.right = Union(H1.right, H2);
10     }
11     return H1;
12 }
```

Random meldable heaps: analysis

Lemma: The expected length of a random walk in a any binary tree with n nodes is at most $\lg(n+1)$.

Note: $\lg n$ denotes $\log_2 n$

Random walk in each tree from root to a null pointer

In each node choose direction left or right randomly

By Lemma the expected time of Union is $O(\log n)$

Leftist heaps

Add imaginary **external nodes** instead of null pointers

Let $s(x)$ be distance from x to the nearest external node

$s(x) = 0$ for external node x

$s(x) = \min(s(x.left), s(x.right)) + 1$ for internal node x

Height-biased leftist tree is a binary tree in which for every internal node x we have $s(x.left) \geq s(x.right)$

Balancing value $s(x)$ is kept in every node x

Leftist heaps

```
 1  Union(H1, H2) {
 2      if (H1 == null) return H2;
 3      if (H2 == null) return H1;
 4      if (H1.key > H2.key) { exchange H1 and H2 }
 5      // now we know H1.key <= H2.key
 6      HN = Union(H1.right, H2);
 7      if(H1.left != null && HN.s <= H1.left.s) {
 8          return new node(H1.data, H1.left, HN)
 9      } else {
10          return new node(H1.data, HN, H1.left)
11      }
12  }
```

Skew heaps

```
1  Union(H1, H2) {
2        if (H1 == null) return H2;
3        if (H2 == null) return H1;
4        if (H1.key > H2.key) { exchange H1 and H2 }
5        // now we know H1.key <= H2.key
6        HN = Union(H1.right, H2);
7        return new node(H1.data, HN, H1.left)
8  }
```

Heavy-light decomposition

$D(v)$: the number of descendants of node v, including v (size of a node)

Edge from v to its parent p is called **heavy** if $D(v) > D(p)/2$,
otherwise it is **light**.

Observations:

- Each node has at most one child connected to it by a heavy edge
- Each path from v to root at most $\lg n$ light edges
  because after each light edge $D(v) \leq D(p)/2$

Fibonacci heaps

Michael L. Fredman, Robert E. Tarjan. "Fibonacci heaps and their uses in improved network optimization algorithms."Journal of the ACM 1987

Introduction to Algorithms, 3rd edition, Chapter 19

Operations:

MakeHeap()

Insert(H, x)

Minimum(H)

ExtractMin(H)

Union(H_1, H_2)

DecreaseKey(H, x, k)

Delete(H, x)

Fibonacci heaps

Operations	Bin. heap worst case	Fib. heap amortized
MakeHeap()	$O(1)$	$O(1)$
Insert(H, x)	$O(\log n)$	$O(1)$
Minimum(H)	$O(1)$	$O(1)$
ExtractMin(H)	$O(\log n)$	$O(\log n)$
Union(H_1, H_2)	$O(n)$	$O(1)$
DecreaseKey(H, x, k)	$O(\log n)$	$O(1)$
Delete(H, x)	$O(\log n)$	$O(\log n)$

More complex structures with worst-case rather than amortized time:

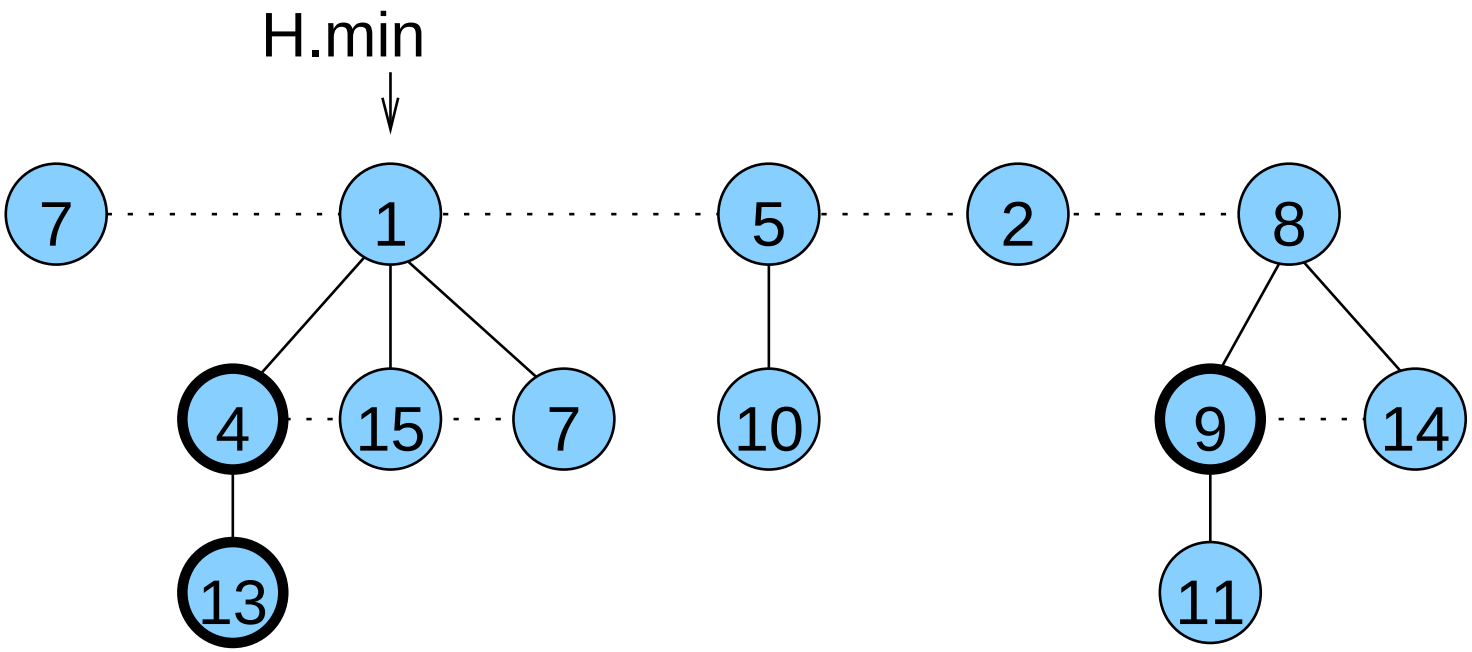
G.S.Brodal: Worst-case efficient priority queues. SODA 1996

Brodal, Lagogiannis, Tarjan: Strict Fibonacci heaps. STOC 2012

Structure of a Fibonacci heap

- Collection of rooted trees, node degrees up to $\mathrm{Deg}(n) = \mathrm{O}(\log n)$
- **Min-heap property:** Key in each non-root node $\geq$ key in its parent
- In each node store: key, other data, parent, first child, next and previous sibling, degree, binary mark
- In each heap: root with min key, number of nodes n, first and last root
- Roots in a heap connected by sibling pointers
- Non-root node is **marked** iff it lost a child since getting a parent
- **Potential function:** number of roots + 2 * number of marked nodes

Example of a Fibonacci heap



Lazy operations with $O(1)$ amortized time

$\Phi =$ number of roots + 2 * number of marked nodes

amortized cost = real cost + $\Phi_{after} - \Phi_{before}$

- MakeHeap: create a new structure
  cost $O(1)$, $\Delta\Phi = 0$
- Insert(H, x): add x as a new root, update $H.n$, $H.min$
  cost $O(1)$, $\Delta\Phi = 1$
- Minimum: return $H.min$
  cost $O(1)$, $\Delta\Phi = 0$
- Union: concatenate root lists, update $H.n$, $H.min$
  cost $O(1)$, $\Delta\Phi = 0$

ExtractMin, $O(\log n)$ amortized

```
1  ExtractMin(H) {
2     z = H.min
3     Add each child of z to root list of H (update its parent)
4     Remove z from root list of H
5     Consolidate(H)
6     return z
7  }
```

Before consolidate, root list may contain up to n roots

Consolidate joins trees until each root has a different degree

Also updates H.min

Let $Deg(n)$ be the maximum degree of a node in a heap of size n

Consolidate

```
1  Consolidate(H) {
2      create array A[0..Deg(H.n)], initialize with null
3      for(each node x in the root list of H) {
4          while(A[x.degree] != null) {
5              y = A[x.degree];
6              A[x.degree] = null;
7              x = HeapLink(H, x, y)
8          }
9          A[x.degree] = x
10     }
11     traverse A, create root list of H, find H.min
12 }
```

HeapLink

```
1  HeapLink(H, x, y) {
2      if(x.key > y key) exchange x and y
3      remove y from root list of H
4      make y a child of x, update x.degree
5      y.mark = false;
6      return x
7  }
```

DecreaseKey, $O(1)$ amortized

```
1  DecreaseKey(H, x, k) {
2      x.key = k
3      y = x.parent
4      if(y!=NULL && x.key < y.key) {
5          Cut(H, x, y)
6          CascadingCut(H, y)
7      }
8      update H.min
9  }
```

Cut: remove x from child list of y, decrement y.degree, add x to the root list, x.parent = null, x.mark = false

Cascading Cut

Node y just lost a child, need to mark it if possible

```
1  CascadingCut(H, y) {
2      z = y.parent
3      if(z != null) {
4          if(y.mark==false) y.mark = true
5          else {
6              Cut(H, y, z);
7              CascadingCut(H, z)
8          }
9      }
10 }
```

Delete $O(\log n)$ amortized

```
1  Delete(H, x) {
2      DecreaseKey(H, x, -infinity)
3      ExtractMin(H)
4  }
```

Fibonacci numbers

$$F_k = \begin{cases} 0 & \text{if } k = 0 \\ 1 & \text{if } k = 1 \\ F_{k-1} + F_{k-2} & \text{if } k \geq 2 \end{cases}$$

Lemma 1: For all $k \geq 0$, we have $F_{k+2} = 1 + \sum_{i=0}^{k} F_i$

Easy proof by induction.

Lemma 2: For all $k \geq 0$, we have $F_{k+2} \geq \phi^k$

where $\phi = (1 + \sqrt{5})/2 = 1.61803\ldots$ is the golden ratio.

Easy proof by induction using the fact that $\phi^2 = \phi + 1$.

Maximum degree of a node

Lemma 3: Let x be a node, and let $y_1, \ldots, y_k$ be its children in the order in which they were linked to x (from earliest). Then $y_j.degree \geq j - 2$ for $j \geq 2$.

Lemma 4: Let x be a node of degree k. Then the size of its subtree is at least $F_{k+2} \geq \phi^k$.

Corollary: The maximum degree $Deg(n)$ of any node in an n-node Fibonacci heap is $O(\log n)$.

Exercise

```
1  Delete2(H, x) {
2     if(x == H.min) ExtractMin(H)
3     else {
4        y = x.parent
5        if(y != null) {
6           Cut(H, x, y)
7           CascadingCut(H, y)
8        }
9        add children of x to the root list of H
10       remove x from the root list of H
11    }
12 }
```

Pairing heaps

- Simple and fast in practice, hard to analyze amortized time
- Fredman, Sedgewick, Sleator, Tarjan 1986
- A tree with arbitrary degrees
  – each node pointer to the first child and next sibling
  – delete and decreaseKey also pointer to previous sibling (or parent if first child)
  – min-heap property
- Linking two trees H_1 and H_2 s.t. H_1 has smaller key:
  – make H_2 the first child of H_1 (original first child is sibling of H_2)
  – subtrees thus ordered by age of linking from youngest to oldest
- ExtractMin most complex, other quite lazy

Recall: binary search trees

- Basic dictionary operations: insert, delete, search
- Keys can be compared with $\leq$ (totally ordered set)
- Every node stores one item and has 0-2 children
- All nodes in the left subtree of a node with key x have value $< \mathrm{x}$
- All nodes in the right subtree of a node with key x have value $> \mathrm{x}$
- Inorder traversal lists keys in increasing order

Binary search trees: running time

- Insert, delete, search: $O(h)$ where h is the height of the tree
- Best case: $h = \Theta(\log n)$
- Worst case: $h = \Theta(n)$ (tree is a path)
- Keys inserted in random order: $h = \Theta(\log n)$ average
- Balanced trees: $h = \Theta(\log n)$
  – examples: AVL, red-black trees
  – keep balancing information in each node
  – complex rules for insert and delete
  – basic step: node rotation (switches parent and child, rearranges their subtrees to maintain correct order of keys)

Today: two tree data structures with $O(\log n)$ amortized time

Scapegoat trees

scapegoat = osoba, na ktorú zhodíme vinu, obetný baránok

- Lazy amortized binary search trees
- Do not require balancing information stored in nodes
- Insert and delete $O(\log n)$ amortized
  search $O(\log n)$ worst-case
- Invariant: keep the height of the tree at most $\log_{3/2} n$
  Note: 3/2 can be changed to $1/\alpha$ for $\alpha \in (1/2, 1)$
- Let $D(v)$ denotes the size of subtree rooted at v

I. Galperin, R.L.Rivest. Scapegoat trees. SODA 1993

Similar idea also A.Andersson 1989

Scapegoat trees, lemma

Lemma 1. If a node v in a tree with n nodes is in depth greater than $\log_{3/2} n$, then on the path from v to the root there is a node u and it parent p such that $D(u)/D(p) > 2/3$.

Let nodes on the path from v to the root be $v_k, v_{k-1}, ...v_0$, where $v_k = v$ and v_0 is the root.

Scapegoat trees, example of use

Scapegoat trees useful when rotations cannot be done fast

(additional information maintained in the nodes)

Goal: maintain a sequence of elements (conceptually a linked list).

Insert gets a pointer to a node, inserts a new node before it, returns pointer to the new node.

Compare gets pointers to two nodes, decided which is earlier in the list.

Idea: store in a scapegoat tree, key is the position in the list.

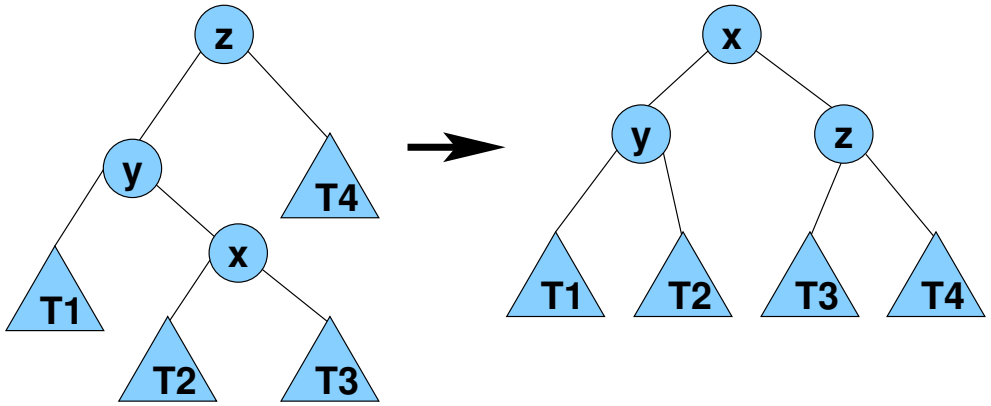
Each node holds the path from the root as a binary number.

Details?

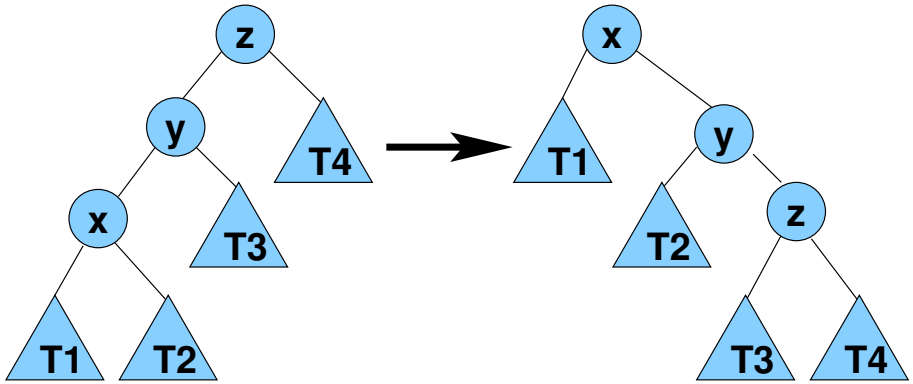
Splay trees

- Binary search tree
- **Amortized time** $O(\log n)$ for each operation
- No balancing information
- The tree can have in principle any shape
- After searching for node x, move node x to root
- This is done by **splaying node** x
- Splaying uses rotations in a prescribed way

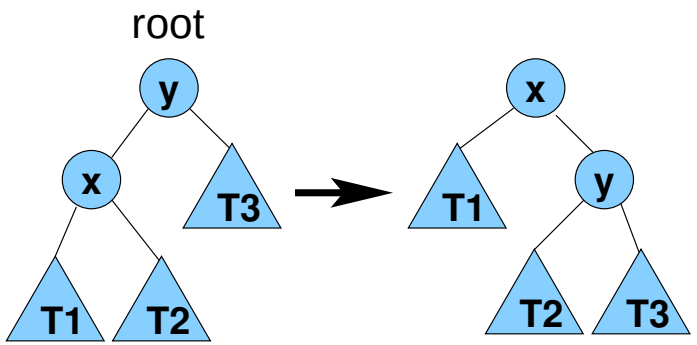
Splay operation for node x: three cases



zig−zag case: same as rotate x twice



zig−zig case: same as rotate y, rotate x



zig case: rotate x

Repeat until x becomes root

Amortized analysis of splaying

real cost: the number of rotations

$D(x)$: the size of the subtree rooted at x

$r(x) = \lg(D(x))$ (rank of node x)

$\Phi(T) = \sum_{x \in T} r(x)$

Lemma 1. Consider one step of splaying x (1 or 2 rotations).

Let $r(x)$ be the rank of x before splaying, $r'(x)$ after splaying.

Amortized cost of one step of splaying is then at most

$3(r'(x) - r(x))$ for zig-zag and zig-zig

$3(r'(x) - r(x)) + 1$ for zig

Lemma 2. Amortized cost of splaying x to the root in a tree with n nodes is $O(\log n)$.

Amortized analysis of splaying

Lemma 1. Consider one step of splaying x (1 or 2 rotations).

Let $r(x)$ be the rank of x before splaying, $r'(x)$ after splaying.

Amortized cost of one step of splaying is then at most

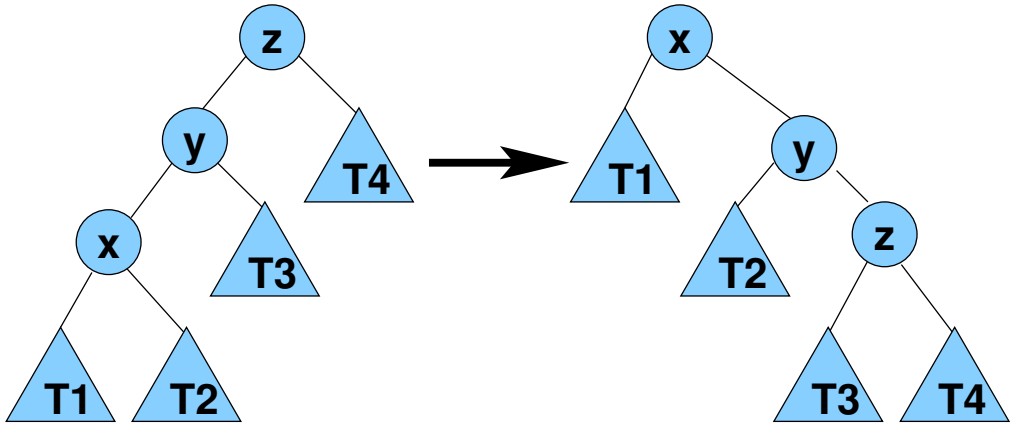
$3(r'(x) - r(x))$ for zig-zag and zig-zig

$3(r'(x) - r(x)) + 1$ for zig

Lemma 2. Amortized cost of splaying x to the root in a tree with n nodes is $O(\log n)$.

Theorem. Amortized cost of insert, search and delete in a splay tree is $O(\log n)$.

Proof of Lemma 1, zig-zig case



zig−zig case: same as rotate y, rotate x

Want: $\hat{c} \leq 3(r'(x) - r(x))$

Have: $\hat{c} = 2 + r'(x) + r'(y) + r'(z) - r(x) - r(y) - r(z)$

Recall: lg is concave and so $\frac{\lg a + \lg b}{2} \leq \lg \frac{a+b}{2}$

and if $a + b \leq 1$, $\lg a + \lg b \leq -2$

Literature

Sleator, Daniel Dominic, and Robert Endre Tarjan. "Self-adjusting binary search trees."Journal of the ACM 32, no. 3 (1985): 652-686.

Collection of splay trees

The following operations can be done in $O(\log n)$ amortized time
(each operation gets pointer to a node)

- findMin(v): find minimum element m in the tree containing v and make it root of that tree.
- join(v, w): all elements in the tree of v must be smaller than all elements in the tree of w. Join these two trees into one.
- splitAfter(v): split tree containing v into 2 trees, one containing keys $\leq v$, one containing keys $> v$, return root of the second tree
- splitBefore(v): split tree containing v into 2 trees, one containing keys $< v$, one containing keys $\geq v$, return root of the first tree

Recall: Union/find

Maintains a collection of disjoint sets, supports operations

- union(v, w): connects sets containing v and w
- find(v): returns representative element of set containing v (can be used to test of v and w are in the same set)

Maintains connected components as we add edges to the graph

Useful in Kruskal's algorithm for minimum spanning tree

Exercise: implement as a collection of splay trees

Union/find implementation

- Each set a tree (non-binary)
- Each node v has a pointer to its parent $v.p$
- find(v) follows parent pointers to the root, returns the root
- union(v, w): use find for v and w and joins one root as a child of other

Improvements:

- Keep track of tree height and always join shorter tree below higher tree
- Path compression in find

Amortized time $O(\alpha(m + n, n))$ where α is inverse Ackermann function, extremely slowly growing

(n is the number of elements, m the number of queries).

Link/cut trees

Maintain a collection of disjoint rooted trees on n nodes

- findRoot(v): find root of the tree containing v
- link(v, w): make w a child of v (w a root, v not in tree of w)
- cut(v) cut edge connecting v to its parent (v not a root)

$O(\log n)$ amortized per operation.

We will show $O(\log^2 n)$ amortized time.

Can be also modified to achieve worst-case $O(\log n)$ time.

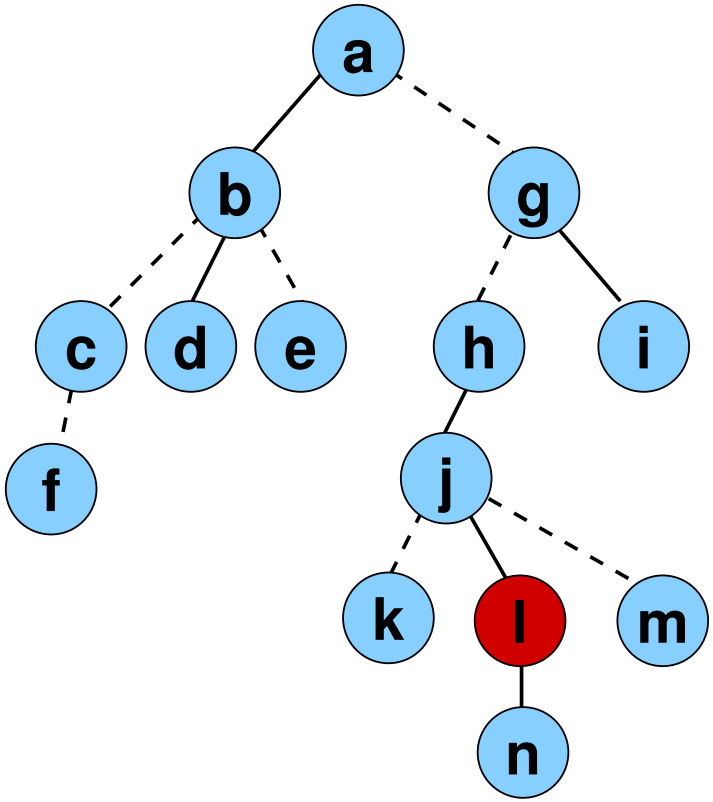
More operations can be added, e.g. weights in nodes.

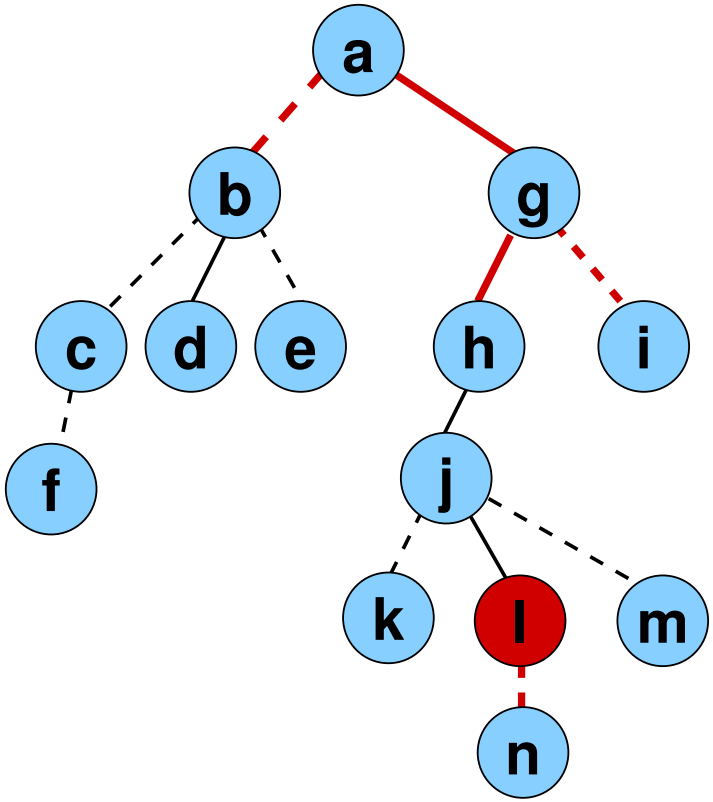
Disjoint paths

- findPathHead(v) - highest element on path containing v
- linkPaths(v, w) - join paths containing v and w (head of v's path will remain head)
- splitPathAbove(v) - remove edge connecting v to its parent p, return some node in the path containing p
- splitPathBelow(v) - remove edge connecting v to its child c, return some node in the path containing c

Can be done in $O(\log n)$ amortized per operation using a collection of splay trees.

Example: expose(l)





expose(v)

```
1   y = cutPathBellow(v);
2   if(y!=NULL) findPathHead(y).dashed = v;
3   while(true) {
4      x = findPathHead(v);
5      w = x.dashed;
6      if(w == NULL) break;
7      x.dashed = NULL;
8      q = splitPathBelow(w);
9      if(q != NULL) {
10        findPathHead(q).dashed = w;
11     }
12     linkPaths(w, x); v = x;
13  }
```

Heavy-light decomposition

$D(v)$: the number of descendants of node v, including v (size of a node)

Edge from v to its parent p is called **heavy** if $D(v) > D(p)/2$,
otherwise it is **light**.

Observations:

- Each node has at most one child connected to it by a heavy edge
- Each path from v to root at most $\lg n$ light edges
  because after each light edge $D(v) \leq D(p)/2$

Amortized analysis of expose

- Potential function Φ: the number of heavy dashed edges

  Cost: the number of splices

- Assume expose creates L new light solid edges

  amortized cost $\hat{c} \leq 2L + 1 = O(\log n)$

- Other operations creating heavy dashed edges:

  – never happens in link, at most $O(\log n)$ times in cut

Literature

Sleator, Daniel D., and Robert Endre Tarjan. “A data structure for dynamic trees.” Journal of computer and system sciences 26, no. 3 (1983): 362-391.

Sleator, Daniel Dominic, and Robert Endre Tarjan. "Self-adjusting binary search trees."Journal of the ACM 32, no. 3 (1985): 652-686.

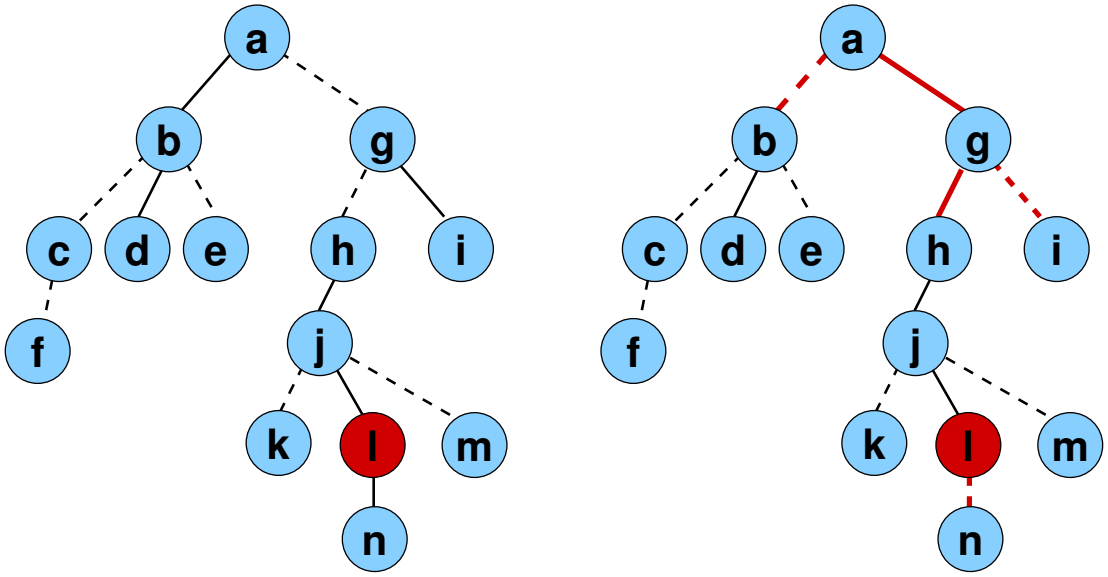
Erik Demaine and his students. "Notes for Lecture 19 Dynamic graphs"MIT course 6.851: Advanced Data Structures, Spring 2012. http://courses.csail.mit.edu/6.851/spring12/scribe/L19.pdf

Mehta, Dinesh P. and Sartaj Sahni eds. "Handbook of data structures and applications."CRC Press, 2004. Section 35.2

Note: At the time of publication, splay trees have improved the best running time for the **maximum flow problem** from $O(nm \log^2 n)$ to $O(nm \log n)$, since then other techniques better.

Lowest common ancestor in dynamic trees

- Link/cut trees allow us to add/remove edges
- We want to add operation lca using expose
- Expose(v) makes path from v to root solid
- $O(\log n)$ amortized time per update/lca
  $O(\log^2 n)$ in our simplified implementation



Full-text keyword search

Plnotextové vyhľadávanie kľúčových slov

Problem statement

Document: Sequence of words

Goal: Create an index for a static set of documents to answer the following queries efficiently.

Query: Given a word w, find all documents containing w.

Example:

Document 0: `Ema ma mamu.`

Document 1: `Mama ma Emu.`

Document 2: `Mama sa ma. Ema sa ma.`

Query `Mama` returns documents 1,2.

Full-text keyword search

Plnotextové vyhľadávanie kľúčových slov

Problem statement

Document: Sequence of words

Goal: Create an index for a static set of documents to answer the following queries efficiently.

Query: Given a word w, find all documents containing w.

Practical issues

Document: webpage/email/book/chapter/abstract/...

Preprocessing: lower/upper case, stemming (úprava na základný tvar), what is a word/word separator?, synonyms, ...

If there are many documents, how to rank them? (Information/text retrieval)

Preprocessing: divide into words, convert to lowercase, . . .

Document 0: ema, ma, mamu

Document 1: mama, ma, emu

Document 2: mama, sa, ma, ema, sa, ma

Inverted index: for each word a list of occurrences (document IDs)

ema: 0,2

emu: 1

ma: 0,1,2

mama: 1,2

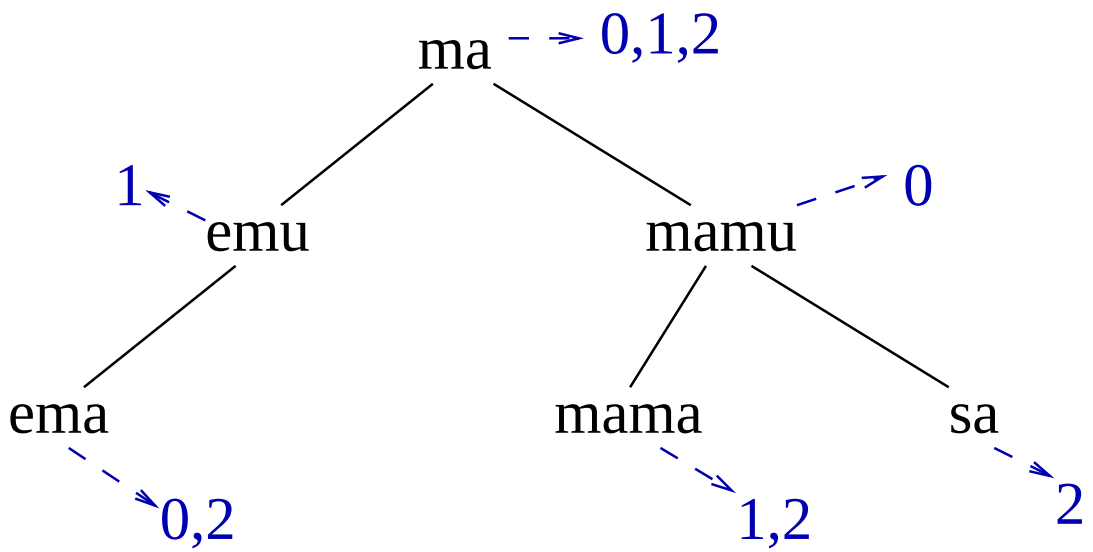
mamu: 0

sa: 2

Implementing inverted index with balanced search trees

Balanced binary search tree, (e.g. red-black tree):

search, insert, delete using $O(\log n)$ comparisons



Trie (lexikografický strom)

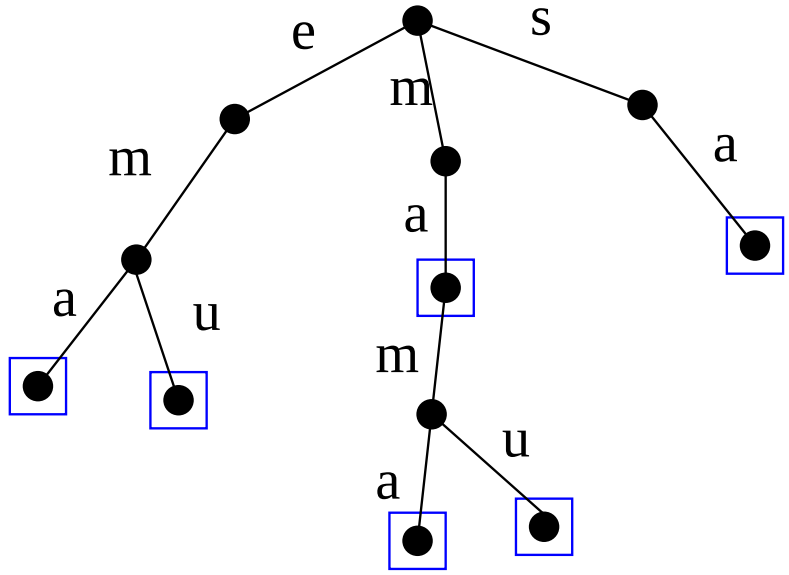
Represents a set of words

Edges labeled with characters

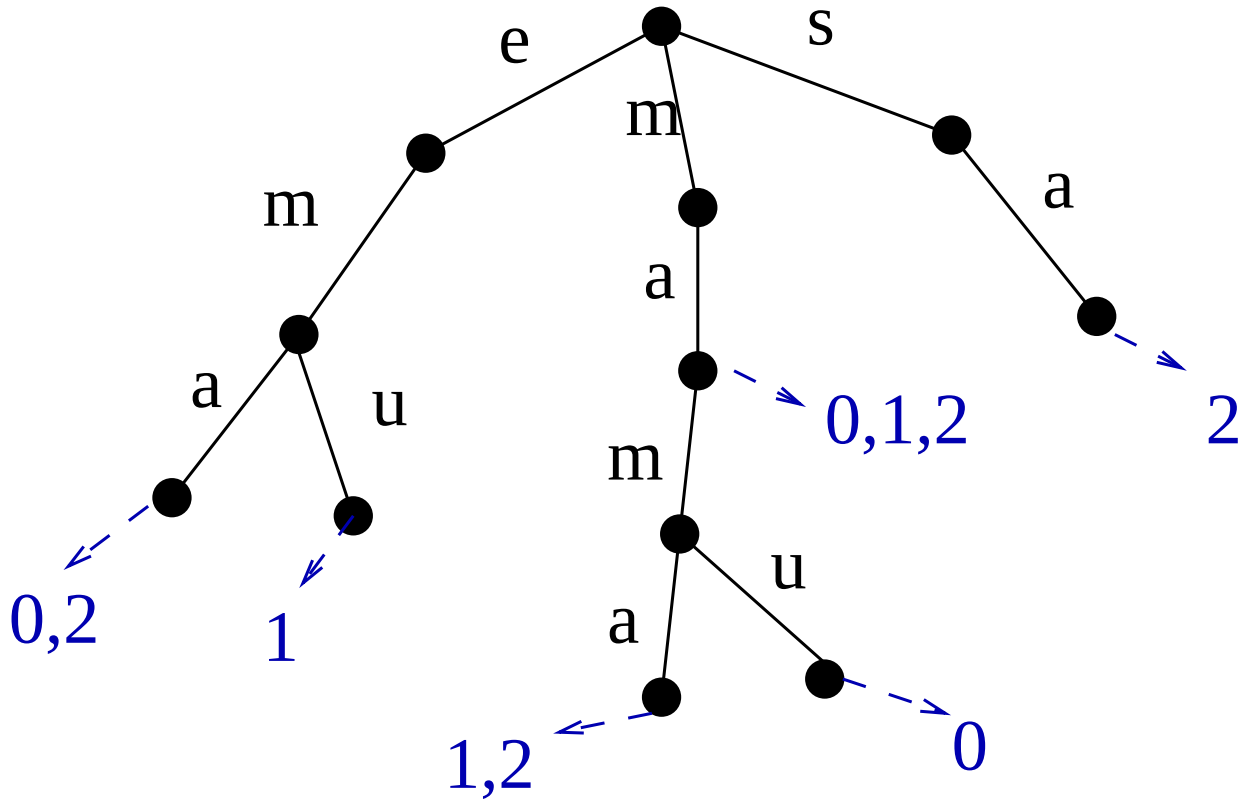
A node represents string read along the path from the root

Root represents empty string

In each node store flag if node in the set, plus other data

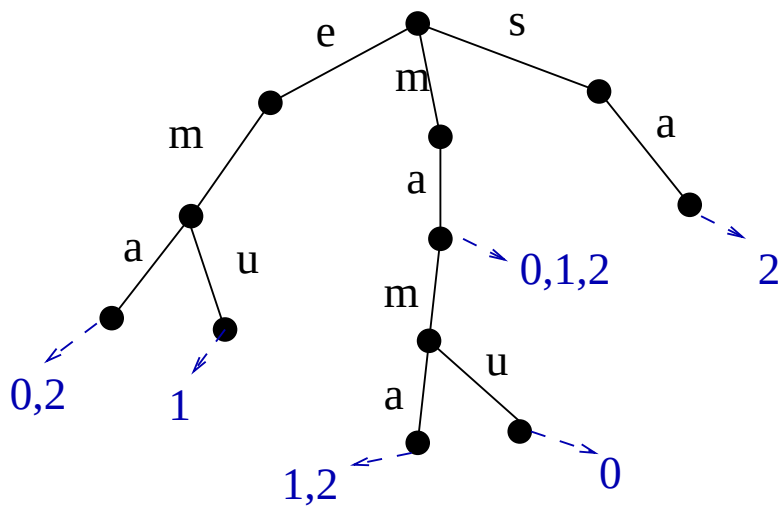


Inverted index implemented as a trie



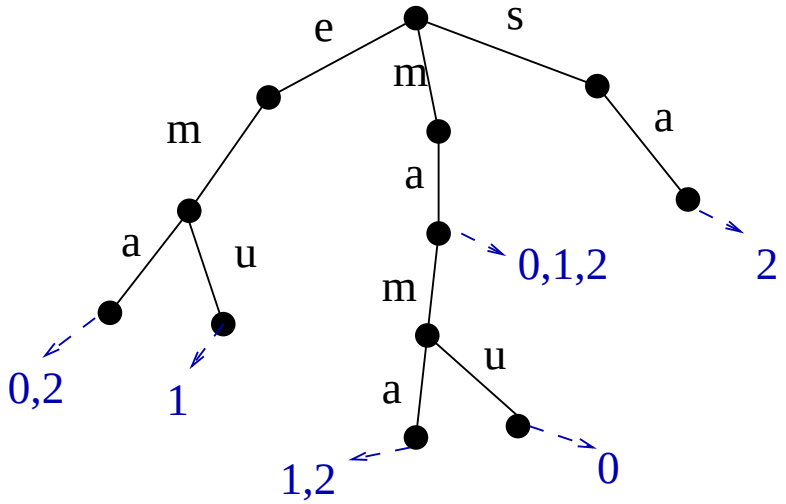
Searching for word w in a trie

```
1  node = root;
2  for(i=0; i<m; i++) {
3      node = node->child[w[i]];
4      if(! node) return empty_list;
5  }
6  return node->list;
```



Inserting word w from document d to a trie

```
1  node = root;
2  for(i=0; i<m; i++) {
3      if(! node->child[w[i]]) {
4          node->child[w[i]] = new node;
5      }
6      node = node->child[w[i]];
7  }
8  node->list.add(d)
```



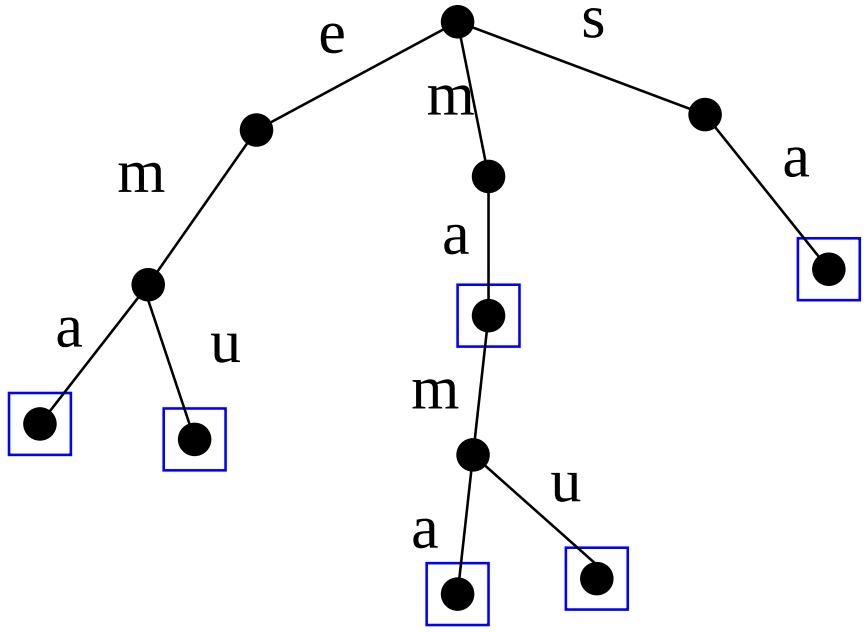
What about delete?

Trie

Assume word of length m, small alphabet

Insert, search, delete in $O(m)$ time if alphabet is small

How to store each node if alphabet large?



Trie

In each node: map from alphabet to pointers to children nodes

Implementation of this map for an alphabet of size σ:

	Search	Insert	Memory
Array of size σ	$O(m)$	$O(m\sigma)$	$O(D\sigma)$
Sorted array	$O(m \log \sigma)$	$O(m \log \sigma + \sigma)$	$O(D)$
Bin. search tree	$O(m \log \sigma)$	$O(m \log \sigma)$	$O(D)$

D – total length of all words

m – length of the word to be searched/inserted

σ – alphabet size

Implementations of inverted index

	Query	Preprocessing
Binary search tree (balanced)	$O(m \log n + p)$	$O(mN \log n)$
Hashing - expected/average case	$O(m + p)$	$O(mN)$
Trie	$O(m \log \sigma + p)$	$O(mN \log \sigma)$

m – max. length of a word

n – the number of distinct words

N – total number of words

σ – alphabet size

p – the number of documents found

Queries with multiple keywords

Searching with 2 keywords (connected by AND)

Intersection of two lists of occurrences

Assume input lists sorted (by some criterion)

Lengths of lists m and n ($\mathrm{m} \leq \mathrm{n}$)

Any ideas?

Queries with multiple keywords

Find intersection of two sorted arrays (lengths $m < n$)

- Linear-time merge $O(m + n)$
- m-times binary search $O(m \log n)$
- Doubling search $O(m \log \frac{n}{m})$

More than two arrays: add one by one, or use a different algorithm

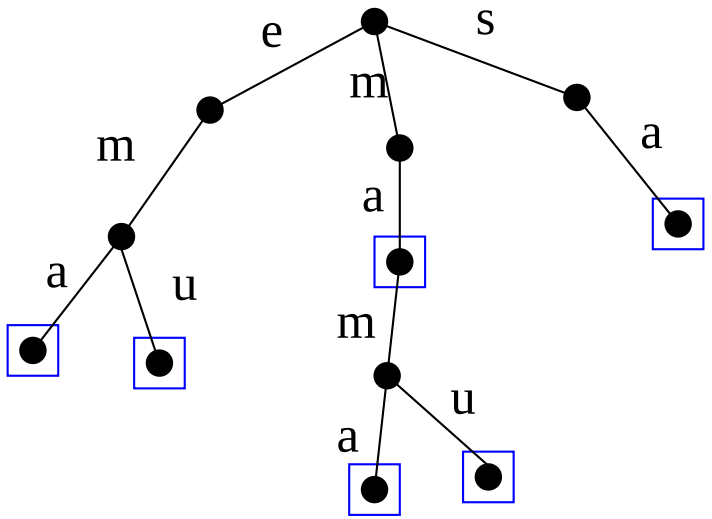
Also possibly preprocess sets for faster answers [Cohen, Porat 2010]

Applications of tries

Work with individual words:

- Keyword search
- Spell-checking
- Counting word frequencies

Also used in multiple pattern search (Aho-Corasick algorithm) and LZW compression



String matching (vyhľadávanie vzorky v texte)

Given: pattern (vzorka) P of length m, text T of length n.

Goal: Find all positions $\{i_1, i_2, \ldots\}$ such that $T[i_j..i_j + m - 1] = P$.

Example:

Input: P =ma, T =Ema ma mamu

Output: 1, 4, 7

Input: P ="a ma", T =Ema ma mamu

Output: 2, 5

Trivial algorithm

```
1  for(i=0; i<=n-m; i++) {
2     j=0;
3     while(j<m && P[j]==T[i+j]) { // (*)
4        j++;
5     }
6     if(j==m) {
7        print(i);
8     }
9  }
```

String matching (vyhľadávanie vzorky v texte)

- Trivial algorithm $O(nm)$
- Knuth-Morris-Pratt algorithm $O(n + m)$ (later)
- What if we want to preprocess T, then search in $O(m + k)$?
  $k =$ the number of occurrences of P in T

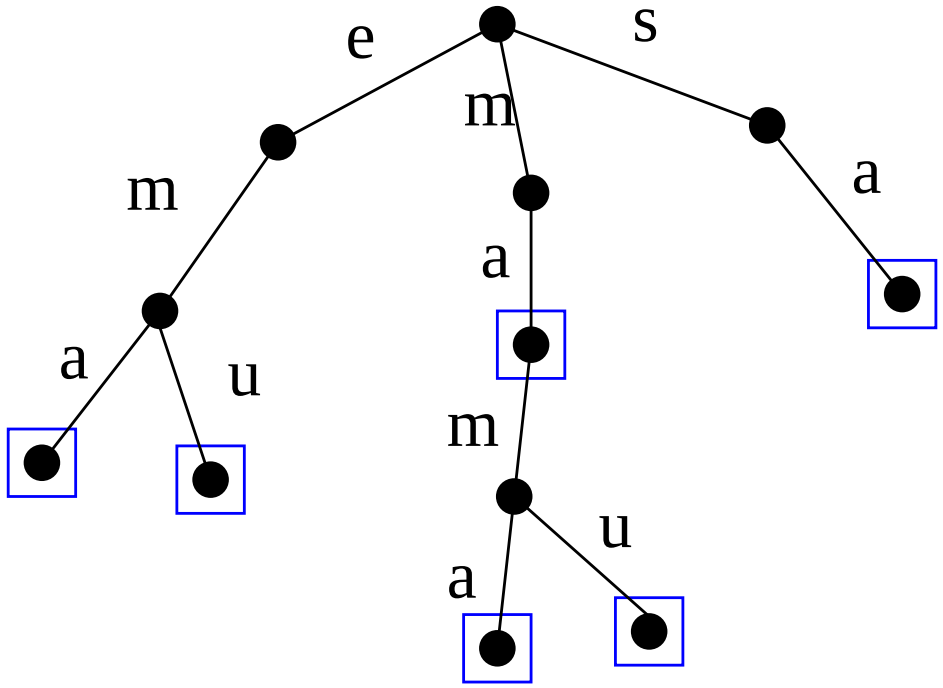
What about the following problems?

Given a set of words $\mathcal{S} = \{S_1, \ldots, S_z\}$:

- Find the longest word w which is a prefix of at least two words in $\mathcal{S}$
- Find the longest word w which is a substring of at least two words in $\mathcal{S}$
- Simpler: Find the longest word w which occurs at least twice in a string T

The longest word which is a prefix of at least two words in $\mathcal{S}$

Example: $\mathcal{S} = \{$ema, ma, mamu, mama, emu$\}$

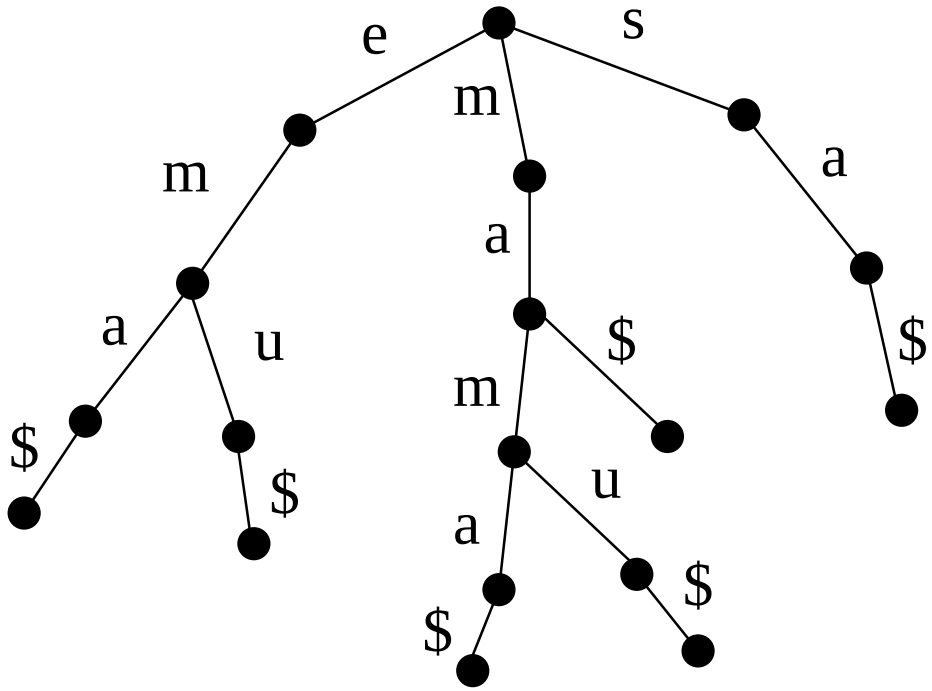


The longest word which is a prefix of at least two words in $\mathcal{S}$

Simplification: add \$ at the end of each word

Words in $\mathcal{S}$ correspond exactly to leaves

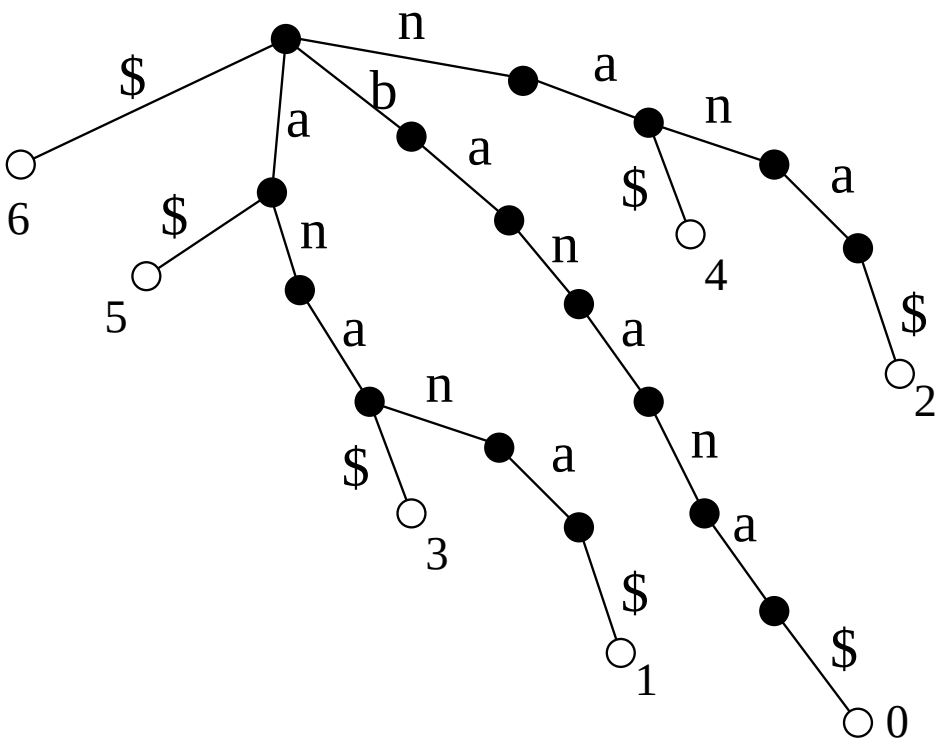
Example: $\mathcal{S} = \{$ema\$, ma\$, mamu\$, mama\$, emu\$$\}$



Trie of all suffixes of a string

Example:

i	0	1	2	3	4	5	6
$T[i]$	b	a	n	a	n	a	$



Nodes correspond to substrings of T

Problem: the number of nodes is $O(n^2)$

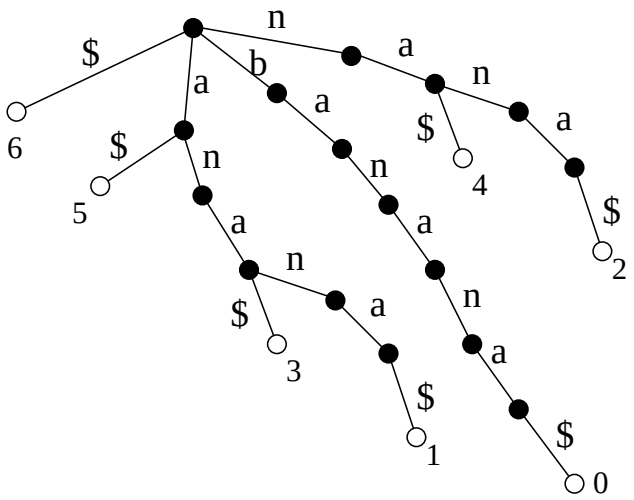
Optional homework

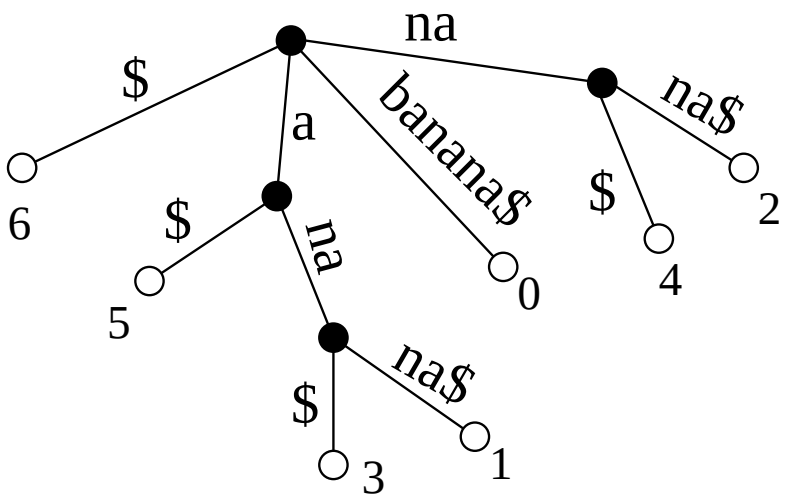
Find a string over binary alphabet which results in a big trie of suffixes

Suffix tree (sufixový strom)

Compact all non-branching paths

T =banana$





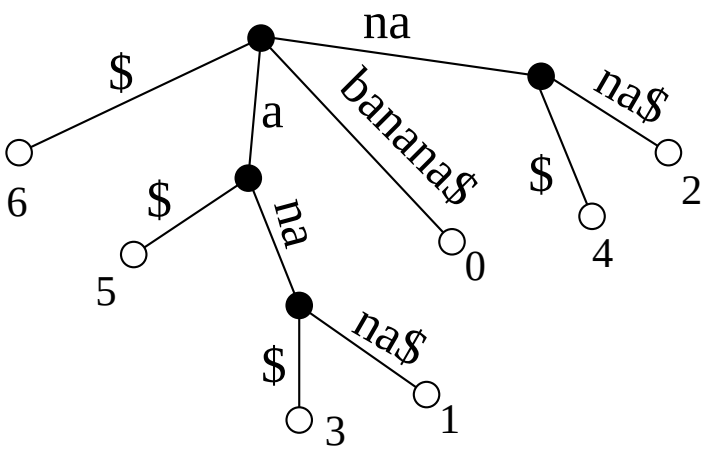
The number of leaves is n

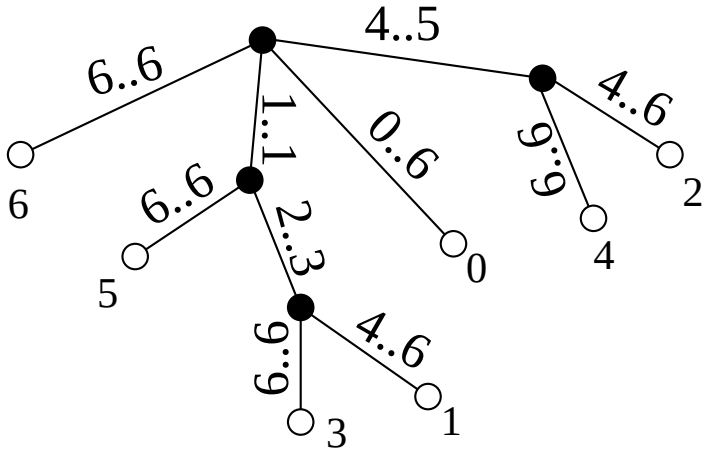
The number of internal nodes is at most $n - 1$

Suffix tree

Store indices to T instead of substrings

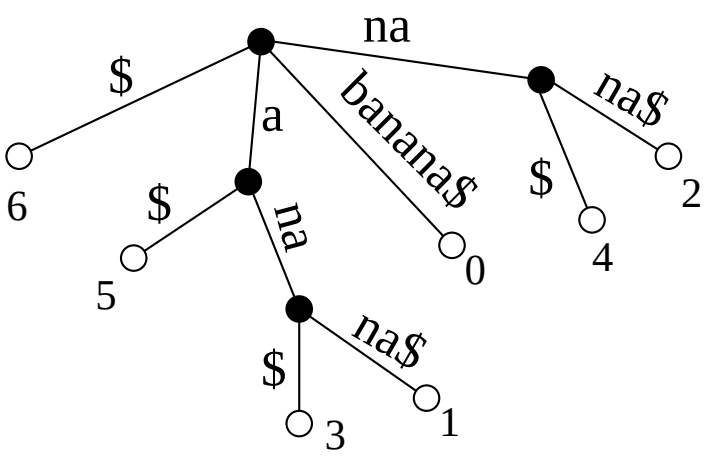
i	0	1	2	3	4	5	6
$T[i]$	b	a	n	a	n	a	$

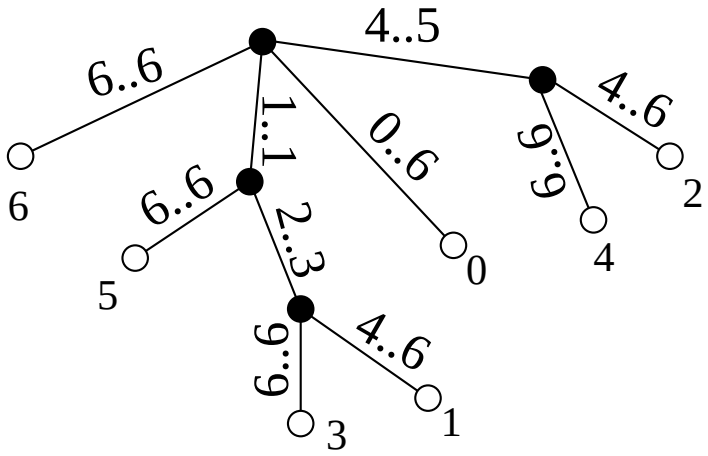




Edges from one node start with different characters.

Suffix tree





Each node:

– pointer to parent

– indices of substring for edge to parent

– suffix start (in a leaf)

– pointers to children (in an internal node)

– other data, e.g. string depth

$O(n)$ nodes, construct in $O(n)$ time for constant σ

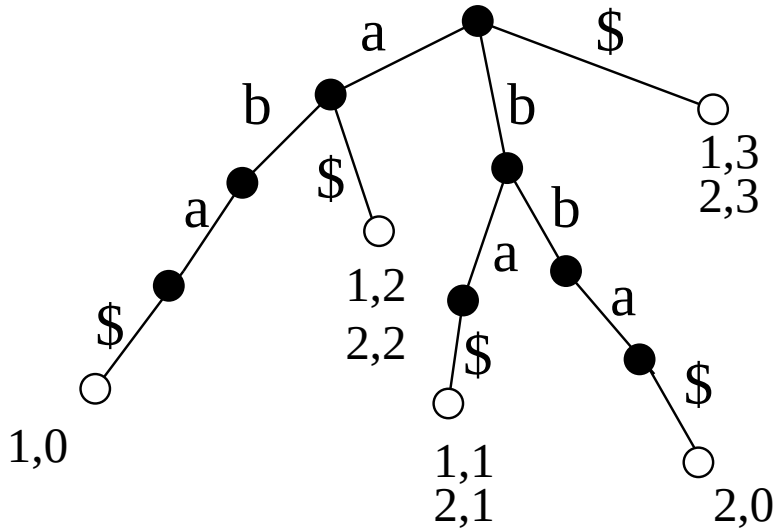
Generalized suffix tree

Store suffixes of several strings $\{S_1, \ldots, S_z\}$

Each leaf a list of suffixes

Each edge i and indices to some S_i

Trie of all suffixes for $S_1 = abaa\$$, $S_2 = bbaa\$$:



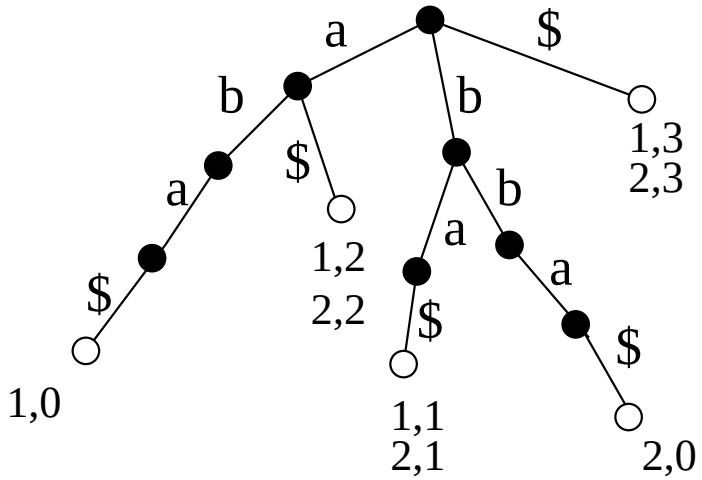
Generalized suffix tree

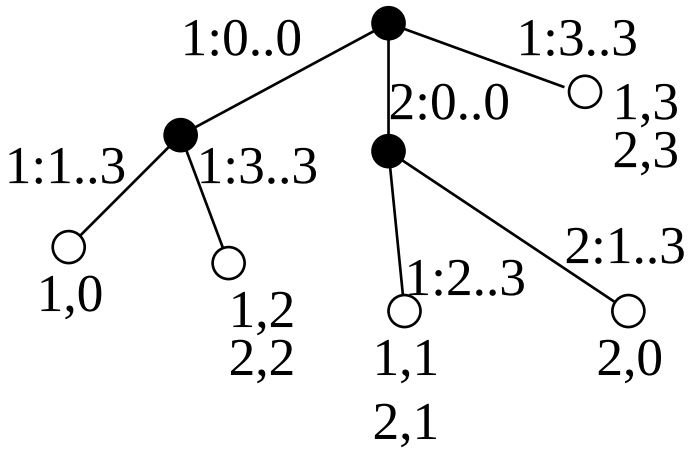
Store suffixes of several strings $\{S_1, \ldots, S_z\}$

Each leaf a list of suffixes

Each edge i and indices to some S_i

Example: $S_1 = abca\$$, $S_2 = bbca\$$:





Applications of suffix trees

- String matching: preprocess text,
  then process each pattern in $O(m + k)$
- Find longest substring with at least two occurrences in S in $O(n)$
  – internal node with highest “string depth”

Generalized suffix trees

For a set of documents $\{S_1, \ldots, S_z\}$

- Find longest substring occuring in at least two input strings in $O(n)$
  – node with highest “string depth” that has at least two different
  document labels in its subtree

Maximal repeats

Maximal pair in S is a pair of substrings $S[i..i+k]$ and $S[j..j+k]$
such that $S[i..i+k] = S[j..j+k]$,
but $S[i-1] \neq S[j-1]$ and $S[i+k+1] \neq S[j+k+1]$

Maximal repeat is a string which is in at least one maximal pair.

Goal: find all maximal repeats in S in $O(n)$ time

Use: poor man's approximate string matching

(e.g. for plagiarism detection)

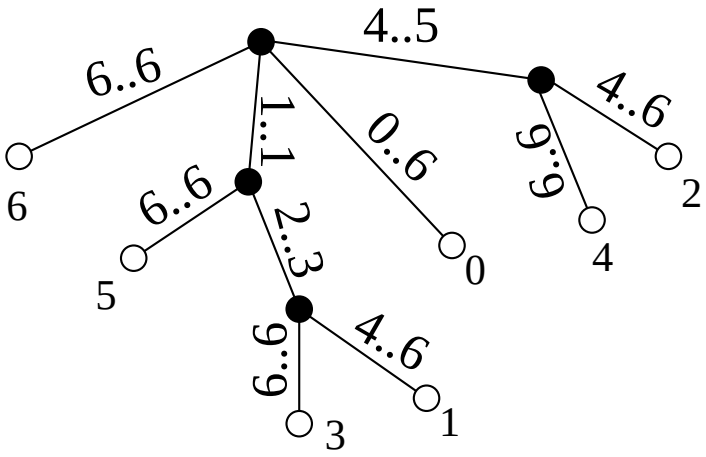
Maximal repeats

Maximal pair in S is a pair of substrings $S[i..i+k]$ and $S[j..j+k]$ such that $S[i..i+k] = S[j..j+k]$,
but $S[i-1] \neq S[j-1]$ and $S[i+k+1] \neq S[j+k+1]$

- Each maximal repeat corresponds to an internal node. Why?
- Def. If v is a leaf for suffix $S[i..n-1]$, its left character is $l(v) = S[i-1]$.
- Def. We will call a node v diverse (rôznorodý) if its subtree contains leaves labeled by at least 2 different values of $l(v)$.
- Thm. Node v corresponds to a maximal repeat iff v is diverse.

Lowest common ancestor in suffix trees

T =banana$



Consider leafs i, j

lca(i, j): longest common prefix of $T[i..n-1]$ and $T[j..n-1]$

String-depth of lca(i, j) gives the length of this prefix

Can be computed in $O(1)$ after $O(n)$ preprocessing for lca

Approximate string matching

Hamming distance $d_H(S_1, S_2)$ between two strings of equal length:

the number of positions where they differ

Task: find approximate occurrences of P in T with Hamming distance $\leq k$

$\{i \mid d_H(P, T[i..i+m-1]) \leq k\}$

Approximate string matching: Trivial algorithm

```
1   for(i=0; i<=n-m; i++) {
2     j=0; err = 0;
3     while(j<m) {
4       if(P[j]!=T[i+j]) err++;
5       if(err>k) { break; }
6       j++;
7     }
8     if(err<=k) {
9       print(i);
10    }
11  }
```

Approximate string matching

Task: find approximate occurrences of P in T with Hamming distance $\leq k$

$\{i \mid d_H(P, T[i..i+m-1]) \leq k\}$

Trivial algorithm $O(nm)$

Algorithm with suffix trees and LCA:

Build generalized suffix tree for P and T in $O(n+m)$

Preprocess for LCA queries in $O(n+m)$

LCA for leaves $T[i..n-1]$ and $P[j..m-1]$ in suffix tree

gives longest common prefix of these two suffixes in $O(1)$

Approximate string matching

```
1  for(int i=0; i<=n−m; i++) {
2    j = 0; err = 0;
3    while(j < m) {
4      q = longest common prefix of T[i+j..n−1], P[j..m−1]
5      if(j+q < m) {   //P[j+q] != T[i+j+q]
6        err++;
7        if(err > k) { break; }
8      }
9      j += q+1;
10   }
11   if(err <= k) { print i; }
12 }
```

Approximate string matching

Hamming distance $d_H(S_1, S_2)$ between two strings of equal length:

the number of positions where they differ

Task: find approximate occurrences of P in T with Hamming distance $\leq k$

$\{i \mid d_H(P, T[i..i+m-1]) \leq k\}$

Trivial algorithm $O(nm)$

Algorithm with suffix trees and LCA: $O(nk)$ [Landau, Vishkin 1986]

More complex version: $O(n\sqrt{k \log k})$ [Amir, Lewenstein, Porat 2000]

Using fast Fourier transform: $O(n\sigma \log m)$ [Fischer and Paterson 1974]

String matching with wildcards

Special character * matches any character from Σ

E.g. aa*b matches aaab, aabb, aacb,...

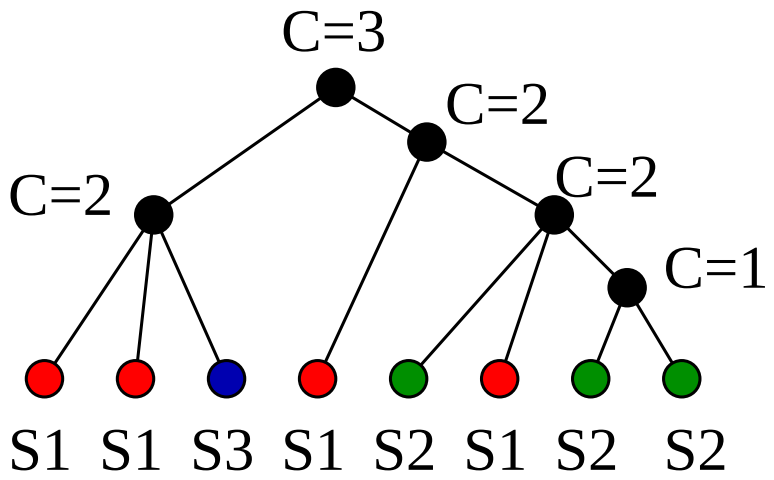
- Trivial algorithm $O(nm)$
- Bit-parallel algorithms for small m: $O(n + m + \sigma)$
- Fast Fourier transform $O(n \log m)$
- Suffix trees $O(nk)$ where k is the number of wildcards

  How?

Counting documents

Generalized suffix tree of $\{S_1, \ldots, S_z\}$

For each node $C(v)$: how many different S_i in its subtree



Use:

– find longest string which is a substring of each S_i

– how many S_i contain pattern P?

Trivial: $O(nz)$; better: $O(n)$ using LCA

Counting documents

- List of leaves in DFS order
- Separate to sublists: L_i = list of suffixes of S_i in DFS order
- Compute lca for each two consecutive members of each L_i

  In each node counter $h(v)$: how many times found as lca
- Compute in each node

  $\ell(v)$: the number of leaves in subtree

  $s(v)$: sum of $h(v)$ in subtree

  $C(v) = \ell(v) - s(v)$

Counting documents

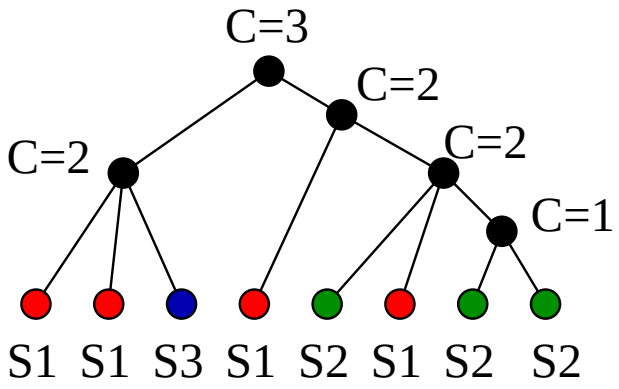
$C(v)$: how many different S_i in its subtree (goal)

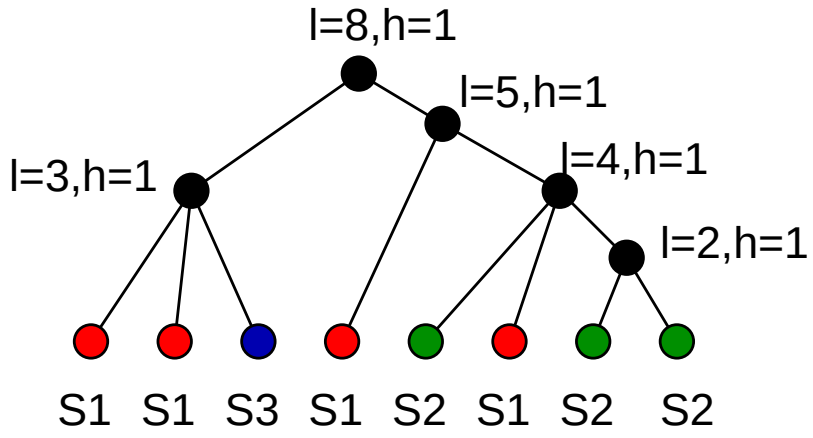
$\ell(v)$: how many leaves in the subtree

$h(v)$: how many times v found as lca

$s(v)$: sum of $h(v)$ in subtree

$C(v) = \ell(v) - s(v)$





Finding all small numbers

We have array A precomputed for RMQ.

For given i, j, x find all indices $k \in \{i, \ldots, j\}$ s.t. $A[k] \leq x$.

```
1  void small(i,j,x) {
2    if(j>i) return;
3    k = rmq(i,j);
4    if(a[k]<=x) {
5      print k;
6      small(i,k-1);
7      small(k+1,j);
8    }
9  }
```

Running time $O(p)$, where p is the number of printed indices

Printing documents

Preprocess texts $\{S_1, \ldots, S_z\}$

Query: which documents contain pattern P?

We can do $O(m + k)$ where k =number of occurrences of P

Want $O(m + p)$ where p =number of documents containing P

Array of leaves L in DFS order

For leaf $L[i]$ let $A[i]$ be the index of previous leaf from the same S_j

Occurrences of P: subtree of corresponding to interval $[i, j]$ in L

Find all $k \in [i, \ldots, j]$ that have $A[k] < i$

Running time? Preprocessing?

Applications of suffix trees

– Index text for string matching

– Find the longest substring with at least 2 occurrences

– Find the longest word which occurs in at least 2 documents

– Find all maximal repeats

With LCA

– Find approximate matches under Hamming distance

– Find pattern with wildcards

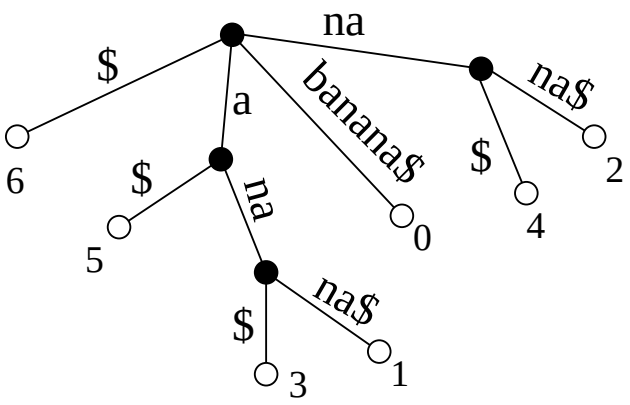
– Count in how many documents a pattern occurs

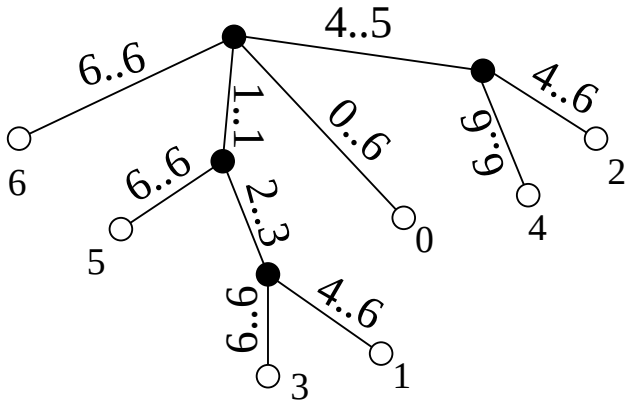
With RMQ

– Print documents containing a pattern

Summary: suffix trees

- Compact representation of all suffixes of a string
- They can be built in $O(n)$ time (proof later)
- They can answer interesting problems related to substring equality
- They need relatively large memory
  (several pointers/integers per character)





Suffix array

Array of suffixes in lexicographic order

(assume $\$ < a \quad \forall a \in \Sigma$)

i	0	1	2	3	4	5	6
$S[i]$	b	a	n	a	n	a	$

i	$SA[i]$	Suffix
0	**6**	$
1	**5**	a$
2	**3**	ana$
3	**1**	anana$
4	**0**	banana$
5	**4**	na$
6	**2**	nana$

i	0	1	2	3	4
$S[i]$	a	a	a	a	$

i	$SA[i]$	Suffix
0	**4**	$
1	**3**	a$
2	**2**	aa$
3	**1**	aaa$
4	**0**	aaaa$

Suffix array

- Array of suffixes in lexicographic order
- Simpler structure, continuous memory
- Less memory: one index per character ($4n$ bytes in total)
- Search for a pattern P by binary search in $O(m \log n)$ time, can be improved to $O(m + \log n)$ with additional memory
- Construction in $O(n)$ even for large alphabets

String matching with suffix arrays

Given suffix array for text T (and possibly other structures), and pattern P, solve these three tasks:

- Task 1: Find out if P occurs in T (yes/no)
- Task 2: Count the number of occurrences of P in T
- Task 3: List all occurrences of P in T

Use binary search in SA:
for string X find maximum i such that $T[SA[i]..n] < X$.

Binary search in suffix array: algorithm 1, $O(m \log n)$

```
1   //find max i such that T[SA[i]..n]< X
2   L = 0; R = n;
3   while(L < R){
4     k = (L + R + 1) / 2;
5     h = 0;
6     while(T[SA[k]+h]==X[h]) { h++; }
7     if(T[SA[k]+h]<X[h]) L = k;
8     else R = k - 1;
9   }
10  return L;
```

Longest common prefix

- $\mathrm{lcp}(A, B) =$ the length of longest common prefix of strings A and B
- $\mathrm{LCP}(i, j) = \mathrm{lcp}(T[SA[i]..n], T[SA[j]..n])$

  i.e. lcp of two suffixes in a suffix array

i	$SA[i]$	Suffix
0	**6**	$
1	**5**	a$
2	**3**	ana$
3	**1**	anana$
4	**0**	banana$
5	**4**	na$
6	**2**	nana$

LCP(2,3) = lcp(ana$,anana$) = 3

LCP(2,5) = lcp(ana$, na$) = 0

Exercise

In one iteration we do $\mathrm{lcp}(X, T[SA[k]..n]) + 1$ comparisons

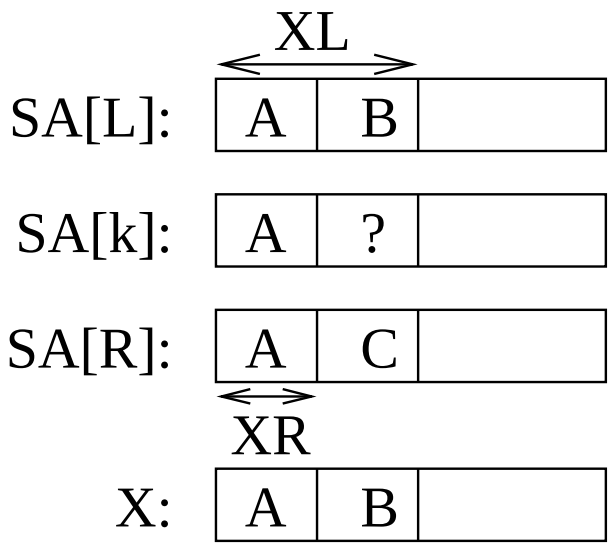
This can be any number between 1 and $\min(m, n + 1 - SA[k])$

Find a bad case (lower bound) for any values $m, n \geq 2$.

Binary search in suffix array: algorithm 2, $O(m \log n)$

```
1   //find max i such that T[SA[i]..n-1]< X
2   L = 0; R = n;
3   XL = lcp(X, T[SA[L]..n]); XR = lcp(X, T[SA[R]..n]);
4   while(R - L > 1){
5     k = (L + R + 1) / 2;
6     h = min(XL, XR);
7     while(T[SA[k]+h]==X[h]) { h++; }
8     if(T[SA[k]+h]<X[h]){ L = k; XL = h; }
9     else { R = k; XR = h; }
10  }
11  sequential search in SA[L..R];
```

Binary search in suffix array: algorithm 2, $O(m \log n)$



Exercise

What is the number of comparisons for $T = ba^{n-1}\$$, $X = a^{n-1}\$$?

What is the number of comparisons for $T = a^{n}\$$, $X = a^{n-1}\$$?

Binary search in suffix array: algorithm 3, $O(m + \log n)$

Recall: $LCP(i, j) = lcp(T[SA[i]..n - 1], T[SA[j]..n - 1])$

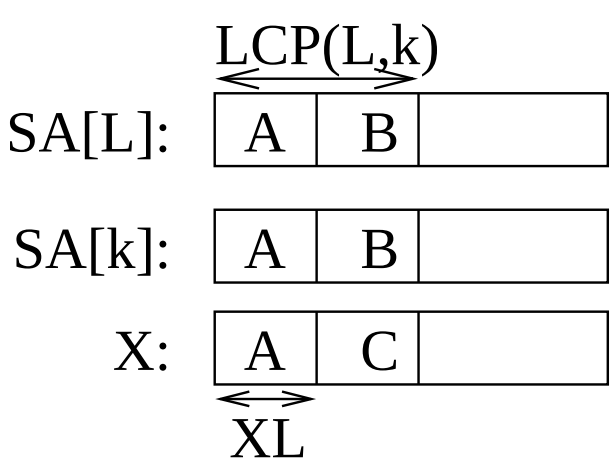
assume we know $LCP(i, j)$ for any i, j (more later)

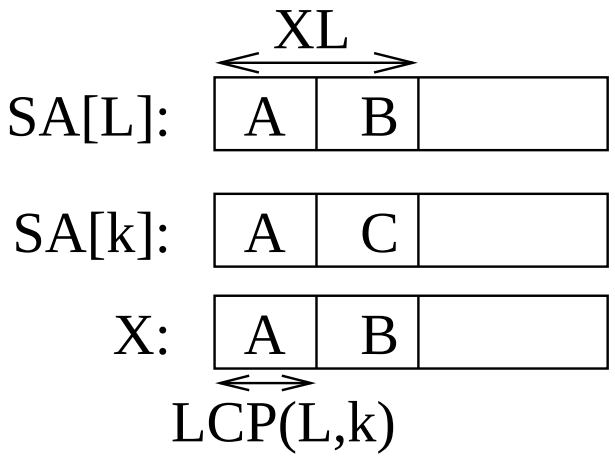
Comparing $T[SA[k]..n]$ and X, assume $XL \geq XR$

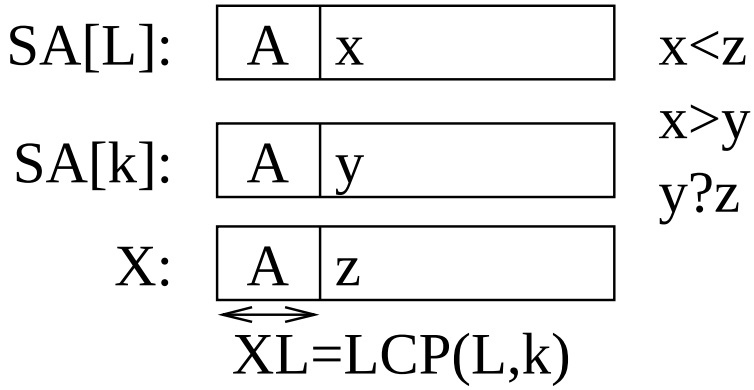
- If $LCP(L, k) > XL$: set $L \leftarrow k$
- If $LCP(L, k) < XL$: set $R \leftarrow k$; $XR \leftarrow LCP(L, k)$;
- If $LCP(L, k) = XL$: start comparing at XL

Case $XL < XR$ symmetrical to $XL \geq XR$

Binary search in suffix array: algorithm 2, $O(m \log n)$







x<z

x>y

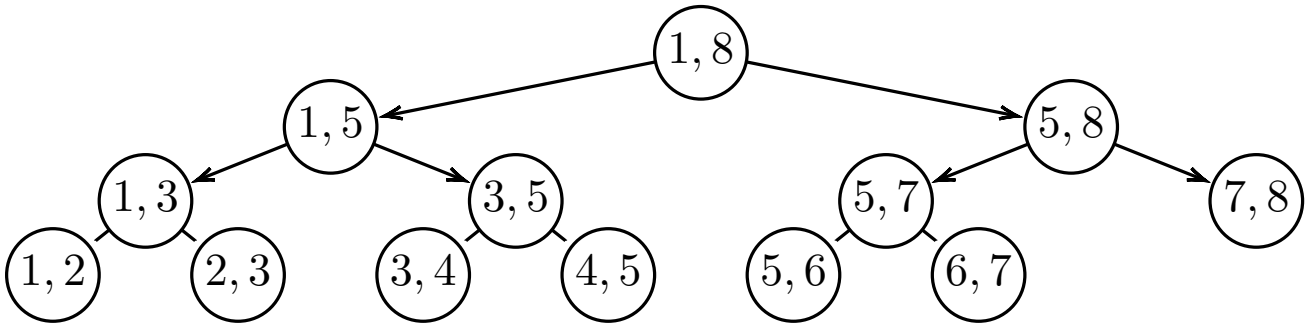
y?z

Binary search in suffix array: algorithm 3, $O(m + \log n)$

```
1  L = 0; R = n; XL = lcp(X, T[SA[L]..n]); XR = lcp(X, T[SA[R]..n]);
2  while(R − L > 1){
3    k = (L + R + 1) / 2;
4    if(XL>=XR && LCP(L,k)>XL) { L = k; }
5    else if(XL>=XR && LCP(L,k)<XL) { R = k; XR = LCP(L,k); }
6    else if(XL<XR && LCP(R,k)>XR) { R = k; }
7    else if(XL<XR && LCP(R,k)<XR) { L = k; XL = LCP(R,k); }
8    else {
9      h = max(XL, XR); while(T[SA[k]+h]==X[h]) { h++; }
10     if(T[SA[k]+h] < X[h]){ L = k; XL = h; } else { R = k; XR = h; }
11   }
12 }
13 sequential search in SA[L..R];
```

LCP values for algorithm 3

Which values are needed? $\mathrm{LCP}(L, k)$ or $\mathrm{LCP}(R, k)$



$2n - 1$ LCP values needed

Let $L[i] = \mathrm{LCP}(i, i+1)$, precompute to an array in $O(n)$ (later)

For $j - i > 1$:

$$\begin{aligned} \mathrm{LCP}(i,j) &= \min\{\mathrm{LCP}(k,k+1) \mid k = i \ldots j-1\} \\ &= \min\{\mathrm{LCP}(i,x), \mathrm{LCP}(x,j)\} \text{ for any } x \in \{i+1, \ldots j-1\} \end{aligned}$$

Suffix trees and arrays - summary

Data structure	Search	# pointers/integers
Suffix tree	$O(m \log \sigma)$	$7n$ or more
Suffix array	$O(m \log n)$	n
Suffix array + LCP	$O(m + \log n)$	$3n$

More memory needed in preprocessing stage.

Next:

- Construction of suffix arrays
- Computation of lcp values
- Construction of suffix trees from suffix arrays

Inverse of a suffix array

Array rank such that $rank[i] = x \iff SA[x] = i$

Can be computed in $O(n)$ from SA:

```
1  for(i=0;i<=n,i++){
2     rank[SA[i]] = i;
3  }
```

Go from suffix to its position in the suffix array, its neighbors, etc.

Longest common prefix

- $\mathrm{lcp}(A, B) =$ the length of longest common prefix of strings A and B
- $\mathrm{LCP}(i, j) = \mathrm{lcp}(T[SA[i]..n-1], T[SA[j]..n-1])$
- $L[i] = \mathrm{LCP}(i, i+1)$

  i.e. lcp of two consecutive suffixes in a suffix array

Lemma: If $SA[x+1]+1 = SA[y+1]$, then $L[y] \geq L[x]-1$.

Suffix	i	$SA[i]$	$L[i]$
$	0	5	0
aabab$	1	0	1
ab$	2	3	2
abab$	3	1	0
b$	4	4	1
bab$	5	2	-

Computation of the LCP array

```
1   h = 0;
2   for(i=0; i<=n; i++) {
3     if(rank[i]>0) {
4       k = SA[rank[i]−1];
5       // compare suffixes S[i..n−1] a S[k..n−1]
6       // assuming they have at least h characters in common
7       while (S[i+h]==S[k+h]) { h++; }
8       // we have found first mismatch
9       L[rank[i]−1] = h;
10      if(h>0) { h−−; }
11    }
12  }
```

Similar but quadratic-time algorithm

```
1  for(i=0; i<=n; i++){
2    if(rank[i]>0){
3      k = SA[rank[i]-1];
4      // compare suffixes starting at i a k
5      h = 0;
6      while (T[i+h]==T[k+h]) { h++; }
7      // we have found the first difference
8      L[rank[i]-1] = h;
9    }
10 }
```

Recall: RadixSort

Bucket sort:

$O(n + d)$ to sort n numbers from $\{0, \ldots, d - 1\}$

Stable (does not change order of equal keys)

Radix sort:

$O(k(n + d))$ to sort n k-digit numbers with digits from $\{0, \ldots, d - 1\}$

Use Bucket sort on each digit, starting from the least significant

How to create a suffix array

Goal: sort all suffixes lexicographically

Not so good options:

- Use MergeSort $O(n^2 \log n)$
- Use RadixSort $O(n^2)$
- Create a suffix tree, then convert to array by DFS. $O(n)$, but linear construction of suffix trees complicated.

Instead use $O(n)$ algorithm, e.g. Karkkainen and Sanders 2003

$O(n)$ algorithm for suffix array construction

Assume alphabet $\{1, \ldots, n\}$

Replace \$ by several zeroes

i	0	1	2	3	4	5	6	7	8			
$S_p[i]$	f	a	b	b	c	a	b	b	d			
$S[i]$	5	1	2	2	3	1	2	2	4	0	...	0

Step 1: sort suffixes $S[3i + k..n]$ for $k = 1, 2$ to $SA_{1,2}$

$S_p = \mathtt{fabbcabbd}$

$S = 5, 1, 2, 2, 3, 1, 2, 2, 4, 0$

$S' = [\mathtt{abb}][\mathtt{cab}][\mathtt{bd0}][\mathtt{bbc}][\mathtt{abb}][\mathtt{d00}]$

$= [1, 2, 2][3, 1, 2][2, 4, 0][2, 2, 3][1, 2, 2][4, 0, 0]$

$= 1, 4, 3, 2, 1, 5, 0$

S'	1	4	3	2	1	5	0
index in S'	0	1	2	3	4	5	6
index in S	1	4	7	2	5	8	
SA'	6	0	4	3	2	1	5
$SA_{1,2}$	–	1	5	2	7	4	8

Step 1: sort suffixes $S[3i + k..n]$ for $k = 1, 2$ to $SA_{1,2}$

$$\text{rank}[i] = \begin{cases} 0 & i \geq n \\ - & i \bmod 3 = 0 \\ j & SA_{1,2}[j] = i \end{cases}$$

j	0	1	2	3	4	5	6
$SA_{1,2}$	–	1	5	2	7	4	8

i	0	1	2	3	4	5	6	7	8	9
$S_p[i]$	f	a	b	b	c	a	b	b	d	
$S[i]$	5	1	2	2	3	1	2	2	4	0
$\text{rank}[i]$	–	1	3	–	5	2	–	4	6	0

Step 2: sort suffixes $S[3i..n]$ to SA_0

$S[3i..n]$ represent as $(S[3i], rank[3i+1])$

$S[3i..n] < S[3j..n]$

$\iff (S[3i], rank[3i+1]) \leq (S[3j], rank[3j+1])$

i	0	1	2	3	4	5	6	7	8	9
$S_p[i]$	f	a	b	b	c	a	b	b	d	
$rank[i]$	–	1	3	–	5	2	–	4	6	0
	f, 1	–	–	b, 5	–	–	b, 4	–	–	

$SA_0 = (6, 3, 0)$.

Step 3: merge SA_0 and $SA_{1,2}$ to SA.

To compare $S[i..n]$ from $SA_{1,2}$ and $S[j..n]$ from SA_0:

$$\text{if } i \bmod 3 = 1\text{:} \quad S[i..n] \leq S[j..n]$$
$$\iff \quad (S[i], rank[i+1]) \leq (S[j], rank[j+1])$$
$$\text{if } i \bmod 3 = 2\text{:} \quad S[i..n] \leq S[j..n]$$
$$\iff \quad (S[i], S[i+1], rank[i+2]) \leq (S[j], S[j+1], rank[j+2])$$

i	0	1	2	3	4	5	6	7	8	9
$S_p[i]$	f	a	b	b	c	a	b	b	d	
$rank[i]$	–	1	3	–	5	2	–	4	6	0

$SA_{1,2} = (-, 1, 5, 2, 7, 4, 8)$

$SA_0 = (6, 3, 0)$

Algorithm overview

- Step 1: sort suffixes $S[3i + k..n]$ for $k = 1, 2$ to $SA_{1,2}$

  (create triples, radixsort and rename, call recursion, renumber)

  $T(2n/3) + O(n)$

- Step 2: sort suffixes $S[3i..n]$ to SA_0

  (create pairs, use radixsort)

  $O(n)$

- Step 3: merge SA_0 and $SA_{1,2}$ to SA.

  (compare as triples or pairs)

  $O(n)$

Overall running time $T(n) = T(\frac{2}{3}n) + O(n)$

Master theorem

One of the three cases from the theorem:

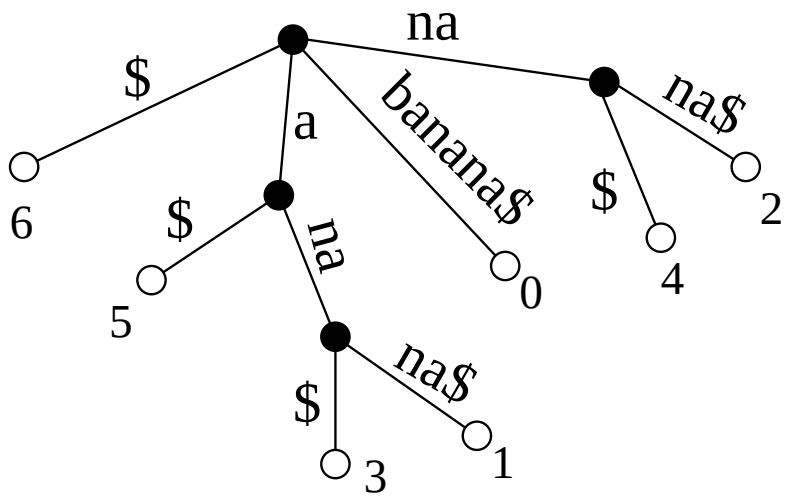
If $T(n) = aT(\frac{n}{b}) + f(n)$
where $f(n) = \Omega(n^{\log_b(a)+\varepsilon})$
and $a \cdot f(\frac{n}{b}) \leq c \cdot f(n)$ for some $c < 1$,
then $T(n) = \Theta(f(n))$.

In our algorithm: $T(n) = T(\frac{2}{3}n) + O(n)$
$a = 1, b = \frac{3}{2}$
$\log_{\frac{3}{2}}(1) = 0$
$a \cdot f(\frac{n}{b}) = \frac{2}{3}n$
Therefore $T(n) = \Theta(n)$.

From suffix array to suffix tree

T =banana$

i	SA[i]	L[i]	suffix
0	**6**	0	$
1	**5**	1	a$
2	**3**	2	ana$
3	**1**	0	anana$
4	**0**	0	banana$
5	**4**	2	na$
6	**2**	-	nana$



From suffix array to suffix tree

```
1  create roof and leaf w corresponding to SA[0]
2  v = w;
3  for(int i=1; i<=n; i++) {
4     while(v.parent.string_depth>L[i-1]) {
5       v = v.parent;
6     }
7     if(v.parent.string_depth<L[i-1]) {
8       split edge from v.parent to v with a new vertex
9       at string depth L[i-1]
10    }
11    attach new leaf w for SA[i] from v.parent;
12    v = w;
13 }
```

Suffix trees and arrays - summary

Data structure	Search	# pointers/integers
Suffix tree	$O(m \log \sigma)$	$7n$ or more
Suffix array	$O(m \log n)$	n
Suffix array + LCP	$O(m + \log n)$	$3n$

More memory needed in preprocessing stage.

- Construction of suffix arrays in $O(n)$
- Computation of lcp values in $O(n)$
- Construction of suffix trees from suffix arrays in $O(n)$

String matching (vyhľadávanie vzorky v texte)

Given: pattern (vzorka) P of length m, text T of length n.

Goal: Find all positions $\{i_1, i_2, \ldots\}$ such that $T[i_j..i_j + m - 1] = P$.

Example:

Input: P =ma, T =Ema ma mamu

Output: 1, 4, 7

Input: P ="a ma", T =Ema ma mamu

Output: 2, 5

Trivial algorithm

```
1  for(i=0; i<=n-m; i++) {
2     j=0;
3     while(j<m && P[j]==T[i+j]) { // (*)
4        j++;
5     }
6     if(j==m) {
7        print(i);
8     }
9  }
```

String matching

- Trivial algorithm $O(nm)$

  Worst case $T = a^n$, $P = a^m$

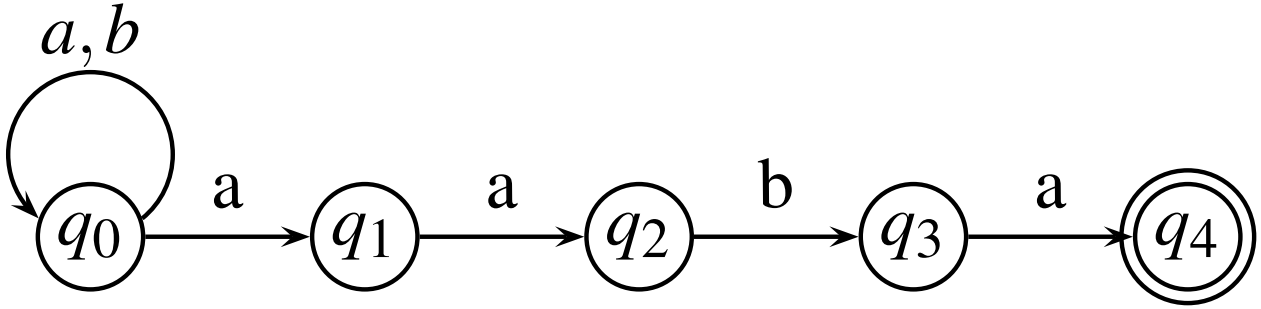
  Average case on random strings $O(n + m)$

- Build suffix tree for T in $O(n)$, then search for P in $O(m)$

- Knuth-Morris-Pratt algorithm $O(n + m)$

Nondeterministic finite automaton for $\{xP|x \in \Sigma^*\}$

$P =$aaba



The automaton can reach state q_i after reading T

iff suffix of T of length i is a prefix of P.

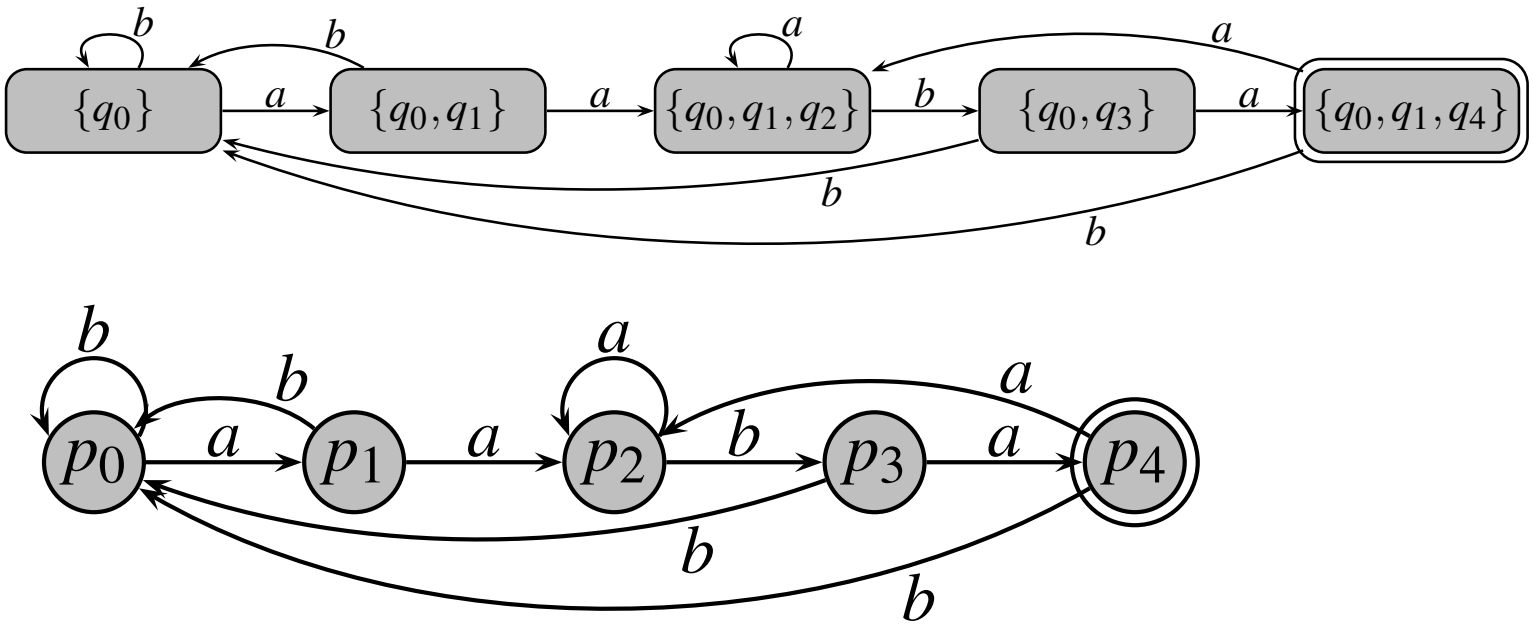
Number of states: $m + 1$

Simulate on a string T in $O(mn)$ time.

Simulation of nondeterministic finite automaton on text T

```
1  S = {0};
2  for(i=0; i<n; i++){
3     S1 = empty_set;
4     foreach state j in S {
5        add delta(j, T[i]) to S1;
6     }
7     if (m in S1){
8        print i-m+1;
9     }
10    S = S1;
11 }
```

Deterministic finite automaton for $\{xP|x \in \Sigma^*\}$



The automaton reaches state p_i after reading T

iff i is the length of the longest suffix of T which is a prefix of P

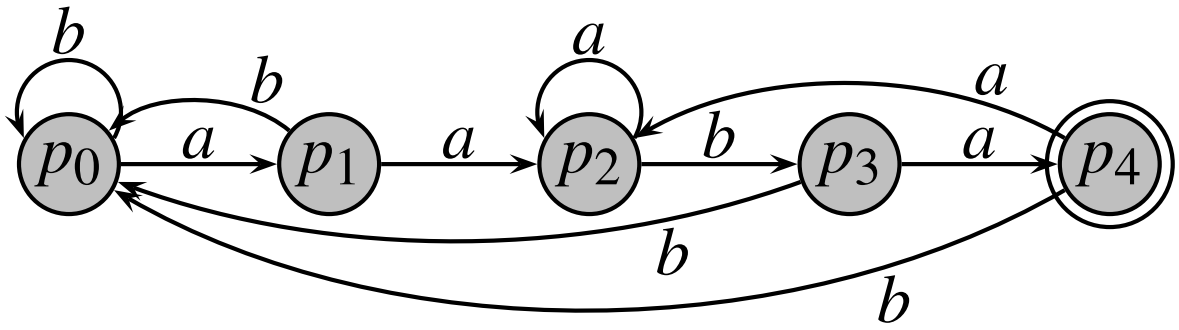
$\Rightarrow$ automaton finishes in state p_m iff T ends with P

Read T letter by letter, update state, print occurrence if state is p_m

Simulation of deterministic finite automaton on text T

```
1  state = 0;
2  for(i=0; i<n; i++){
3     state= delta(state,T[i]);
4     if (state==m){
5        print i-m+1;
6     }
7  }
```

Deterministic finite automaton for $\{xP|x \in \Sigma^*\}$



Number of states: $m + 1$

Simulate on string T in $O(n)$ time.

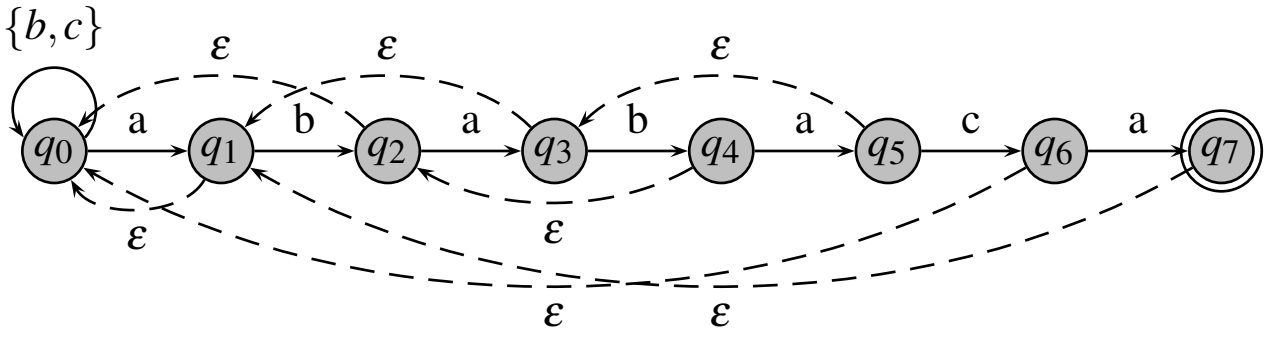
Memory $O(m\sigma)$

Building automaton $O(m\sigma)$ time – optional homework

Overall $O(n + m\sigma)$ time

Morris-Pratt algorithm 1970 $O(m+n)$

P =ababaca



Epsilon transition used when no other option.

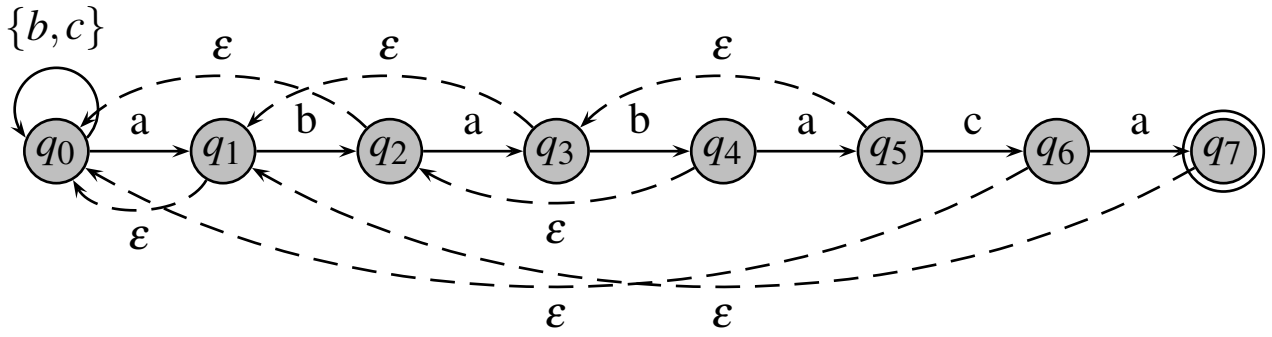
Epsilon transition from state i to $sp[i]$ ($sp[i] < i$).

$sp[i]$ = length of the longest proper suffix of $P[0..i-1]$ which is also a prefix of P.

The automaton reaches state q_i after reading T

iff i is the length of the longest suffix of T which is a prefix of P

Morris-Pratt algorithm



$i, sp[i], sp[sp[i]], \ldots, 0$ form a linked list of all suffixes of $P[0 \ldots i-1]$ which are prefixes of P.

E.g. $i = 5$, $P[0..4] =$ ababa

Linked list q_5, q_3, q_1, q_0

Suffixes/prefixes ababa, aba, a, ε

Next letter x arrives: find longest one that can be followed by x in P

$x =$a, state $q_0 \to q_1$; $x =$b, state $q_3 \to q_4$; $x =$c, state $q_5 \to q_6$

Morris-Pratt algorithm

```
1   state=0;
2   for(i=0; i<n; i++){
3     while(state>0 && T[i]!=P[state]){  //epsilon transitions
4       state=sp[state];
5     }
6     if(T[i]==P[state]){   //move to next state
7       state++;
8     }
9     if(state==m){   //accepting state — print occurrence
10      print i-m+1; state=sp[state];
11    }
12  }
```

Morris-Pratt algorithm preprocessing

```
1   sp[0]=sp[1]=0;
2   j=0;
3   for(i=2; i<=m; i++){
4     // invariant: j=sp[i-1];
5     while(j>0 && P[i-1]!=P[j]) {
6       j=sp[j];
7     }
8     if(P[i-1]==P[j]) {
9       j++;
10    }
11    sp[i]=j;
12  }
```

$sp[i]$ = length of the longest proper suffix of $P[0..i-1]$ which is also a prefix of P.

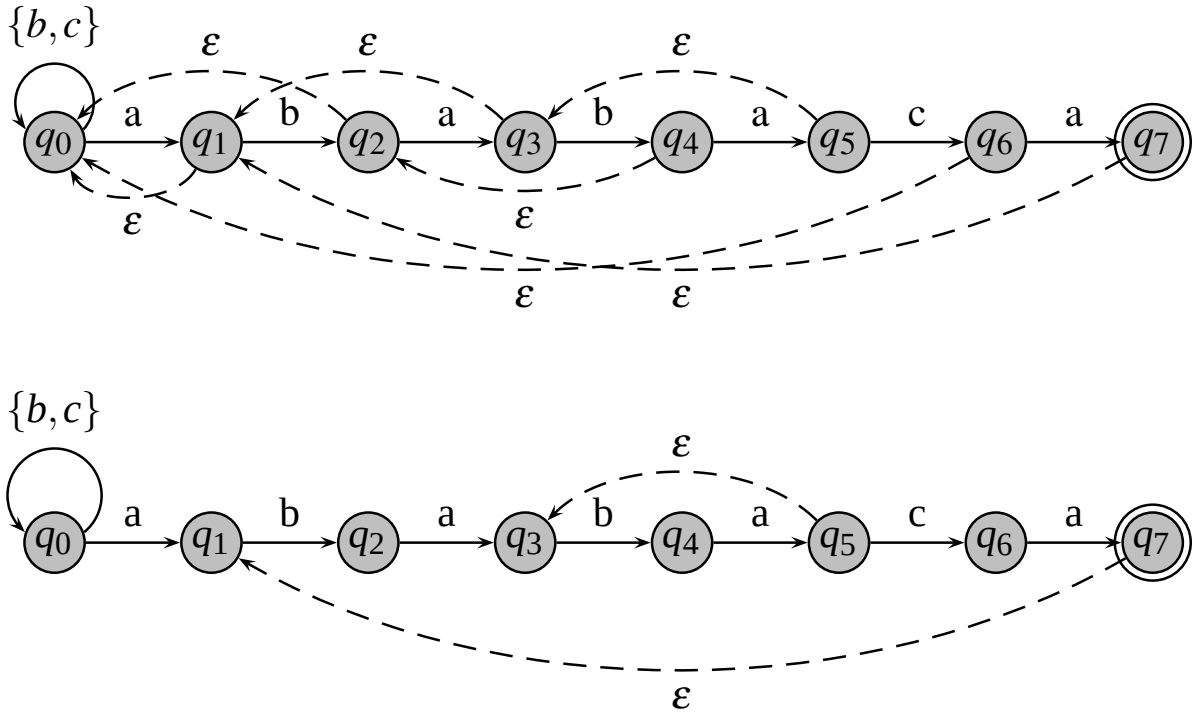
Morris Pratt algorithm 1970

- Build sp table in $O(m)$ time, $O(m)$ memory
- Simulate automaton on T in $O(n)$ time
- Works well for large alphabets

  characters used only for equality testing
- Works well for streaming T,

  but can have $O(m)$ delay between characters

Knuth-Morris-Pratt algorithm, 1977



Longer epsilon transitions:

from state i to $sp_2[i]$, where $sp_2[i]$ is the length of the longest proper sufix $P[0..i-1]$, which is a prefix of P and $P[j] \neq P[i]$

Knuth-Morris-Pratt algorithm, preprocessing

Given values $sp[i]$ from MP algorithm, compute $sp_2[i]$ from KMP

```
1  sp2[0]=0;
2  for(i=1; i<=m; i++) {
3     if(i==m || P[sp[i]]!=P[i]) {
4        sp2[i]=sp[i];
5     }
6     else {
7        sp2[i]=sp2[sp[i]];
8     }
9  }
```

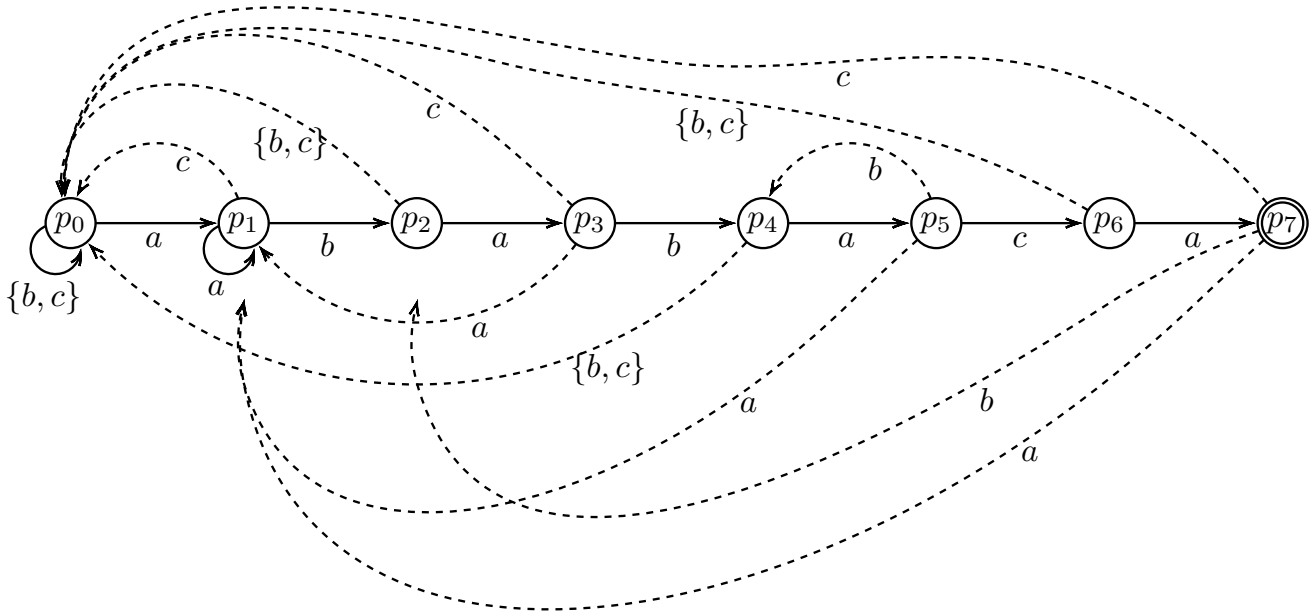
Knuth-Morris-Pratt algorithm, analysis

Theorem: The number of ε transitions in one iteration of KMP algorithm is at most $\log_\phi(m+1)$, where $\phi = \frac{1+\sqrt{5}}{2}$ is the golden ratio.

Note: Overall running time still $O(n+m)$.

Optional homework

Compute DFA in $O(m\sigma)$ time using $sp[i]$ table from MP alg.



DFA reaches state p_i after reading T

iff i is the length of the longest suffix of T which is a prefix of P

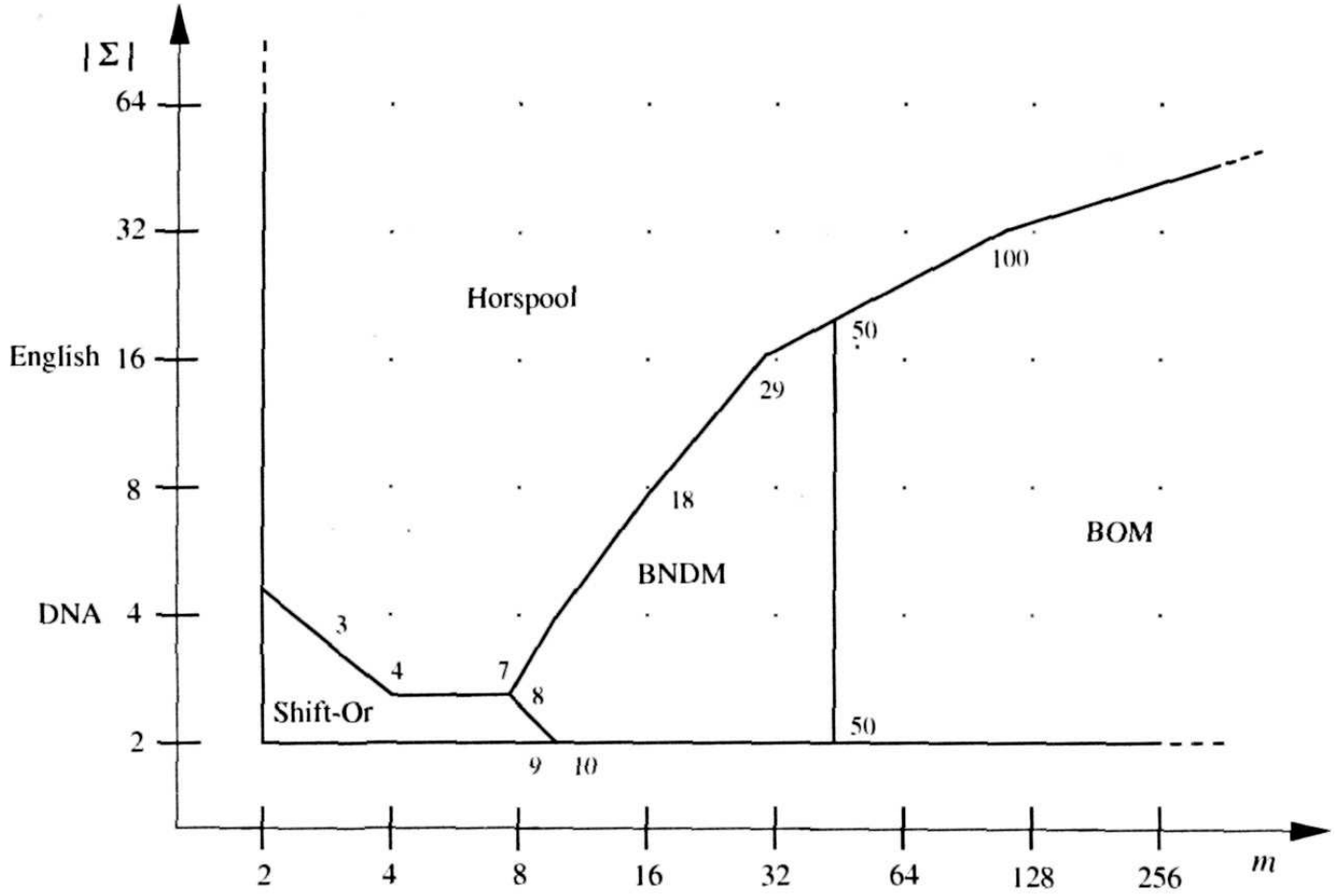
Overview of string matching algorithms

Algorithm	Worst case	Average case	Note
Trivial	$O(nm)$	$O(n+m)$	simple
DFA	$O(n+\sigma m)$	$O(n+\sigma m)$	real-time
(K)MP	$O(n+m)$	$O(n+m)$	
Boyer Moore	$O(nm), O(n+m)$	$O(\frac{n}{\sigma}+m)$	
Shift-and	$O(n+m+\sigma)$	$O(n+m+\sigma)$	m-bit register
BNDM	$O(nm+\sigma)$	$O(n\frac{\log_\sigma m}{m}+m+\sigma)$	m-bit register

Multiple patterns: Aho Corasick $O((n+m)\log\sigma+k)$

2D patterns: Baker Bird $O((n+m)\log\sigma)$

Fastest algorithm for various m and σ



Navarro, Raffinot (2002) Flexible Pattern Matching in Strings.

Random texts, 32-bit numbers

Approximate string matching

Hamming distance $d_H(S_1, S_2)$ between two strings of equal length:

the number of positions where they differ

Task: find approximate occurrences of P in T with Hamming distance $\leq k$

$\{i \mid d_H(P, T[i..i+m-1]) \leq k\}$

Trivial algorithm $O(nm)$

Algorithm with suffix trees and LCA: $O(nk)$ [Landau, Vishkin 1986]

More complex version: $O(n\sqrt{k \log k})$ [Amir, Lewenstein, Porat 2000]

Using fast Fourier transform: $O(n\sigma \log m)$ [Fischer and Paterson 1974]

Edit distance, Levenshtein distance (editačná vzdialenosť)

Edit operations: $(u, v \in \Sigma^*, a, b \in \Sigma)$

– insertion (inzercia) $uv \rightarrow uav$

– deletion (delécia) $uav \rightarrow uv$

– substitution (substitúcia) $uav \rightarrow ubv$

Edit distance $d_E(S, T) =$

shortest sequence of edit operations that changes S to T

Example:

$S =$ema ma mamu, $T =$mama sa ma, $d_E(S, T) = 5$

ema␣ma␣mamu (delete e) ma␣ma␣mamu (delete space)

mama␣mamu (substitute m->s) mama␣samu (insert space)

mama␣sa␣mu (substitute u-a) mama␣sa␣ma

Edit distance as an alignment

Sequence alignment (zarovnanie):

insert gaps ($-$) to S and T to get matrix with 2 rows

— column with a gap (insertion or deletion): cost 1

— column with two different symbols (substitution): cost 1

— column with equal symbols: cost 0

Example:

```
ema␣ma␣ma-mu
-ma-ma␣sa␣ma
100100010101
```

Problem: compute $d_E(S, T)$ for two input strings S and T

(note $d_H(S, T)$ trivially in $O(n)$ time)

Dynamic programming for $d_E(S, T)$

Let $m = |S|$, $n = |T|$, indexing from 1: $S[1..m]$, $T[1..n]$

Let $A[i, j] = d_E(S[1..i], T[1..j])$

Compute $A[i, j]$ for $0 \leq i \leq m$, $0 \leq j \leq n$

Example:

`12345678901`

`ema␣ma␣mamu`

`mama␣sa␣ma`

$A[3, 4] = 2$

Dynamic programming for $d_E(S, T)$

Let $m = |S|$, $n = |T|$, indexing from 1: $S[1..m]$, $T[1..n]$

Let $A[i, j] = d_E(S[1..i], T[1..j])$

Compute $A[i, j]$ for $0 \leq i \leq m$, $0 \leq j \leq n$:

$$A[0, i] = A[i, 0] = i$$

$$A[i, j] = \min \begin{cases} A[i-1, j-1] + c(S[i], T[j]) \\ A[i-1, j] + 1 \\ A[i, j-1] + 1 \end{cases}$$

$$c(a, b) = \begin{cases} 0 & \text{if } a = b \\ 1 & \text{if } a \neq b \end{cases}$$

		m	a	m	a		s	a		m	a
	0	1	2	3	4	5	6	7	8	9	10
e	1	1	2	3	4	5	6	7	8	9	10
m	2	1	2	2	3	4	5	6	7	8	9
a	3	2	1	2	2	3	4	5	6	7	8
	4	3	2	2	3	2	3	4	5	6	7
m	5	4	3	2	3	3	3	4	5		
a											

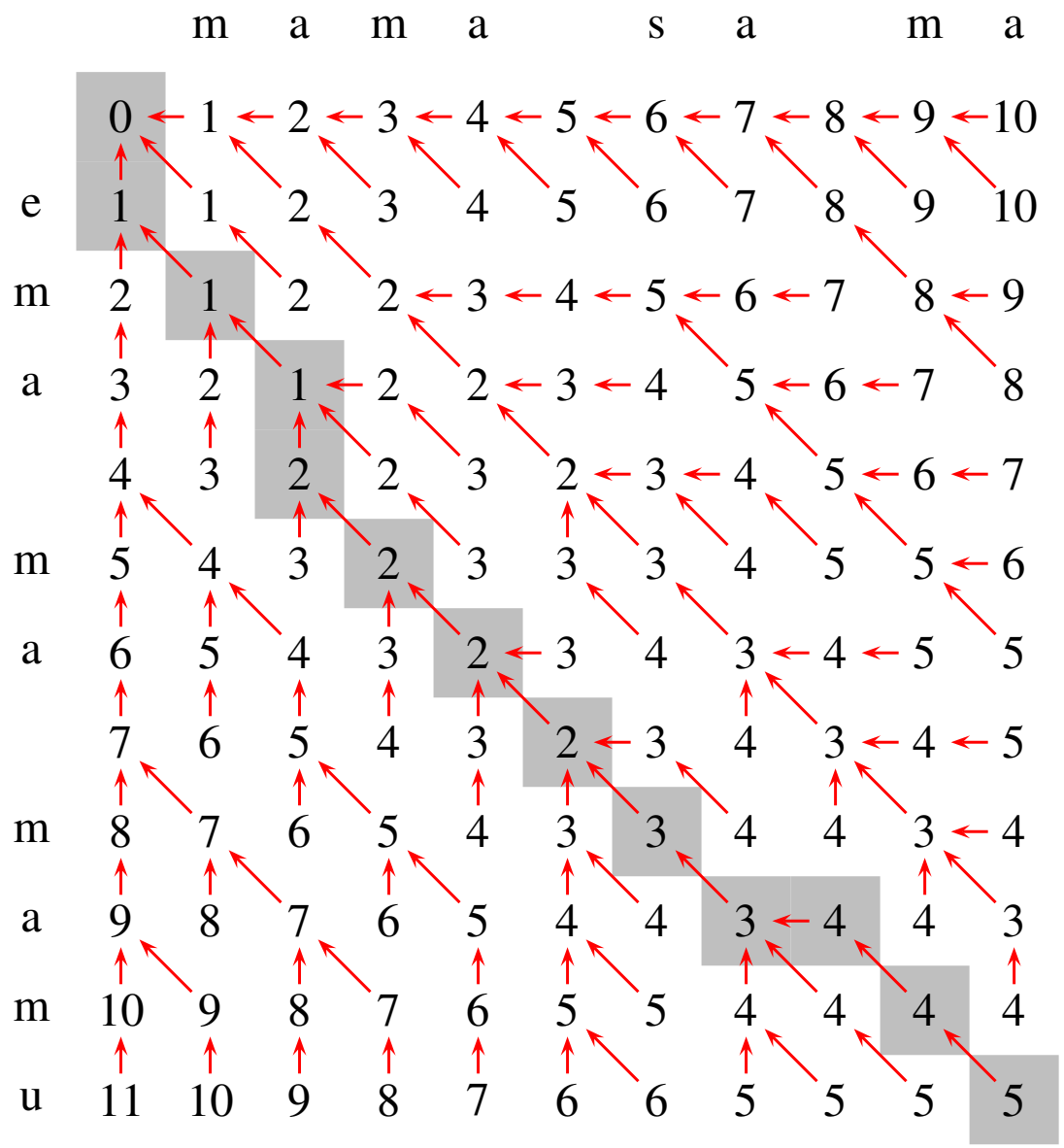
$$A[i,j] = \min \begin{cases} A[i-1,j-1] \\ \quad +c(S[i],T[j]) \\ A[i-1,j]+1 \\ A[i,j-1]+1 \end{cases}$$

m

a

m

u



Alignment:

```
ema␣ma␣ma-mu
-ma-ma␣sa␣ma
```

Generalized edit distance

(zovšeobecnená editačná vzdialenosť)

Table of weights $w(a, b)$ for $a, b \in \Sigma \cup \{-\}$

For $a, b \in \Sigma$:

$w(a, -)$ cost of deletion, $w(-, b)$ cost of insertion

$w(a, b)$ cost of substitution from a to b, or cost of identity if $a = b$

Dynamic programming to find the alignment with lowest cost

– for some strange weights not the lowest sequence of operations

– not always a distance function

$$A[i, j] = \min \begin{cases} A[i-1, j-1] + w(S[i], T[j]) \\ A[i-1, j] + w(S[i], -) \\ A[i, j-1] + w(-, T[j]) \end{cases}$$

Longest common subsequence lcs

Najdlhšia spoločná podpostupnosť

Def: subsequence of a sequence $a_1, \ldots, a_n$ is sequence $a_{i_1}, a_{i_2}, \ldots, a_{i_k}$ such that $1 \leq i_1 < i_2 < \cdots < i_k \leq n$

$lcs(S, T) =$ length of the longest sequence which is a common subsequence of both S and T

Example:

$S =$ema ma mamu, $T =$mama sa ma, $lcs(S, T) = 7$

Also alignment (disallow substitutions):

```
ema␣ma␣---mamu
-ma-ma␣sa␣ma--
.**.***...**..
```

How to find using DP for generalized edit distance?

Program diff

Compare two files line by line, e.g. two versions of a source code.

Find (approximation of) lcs(S,T) where symbols are lines of the files.

```
1522a1540
>     my $last_line = undef;
1525,1526c1543,1544
<     foreach my $gtf_line (@$transcript) {
<       # printf STDERR Dumper($gtf_line);
---
>     for(my $line_num = 0; $line_num<@$transcript; $line_num++)
>       my $gtf_line = $transcript->[$line_num];
```

Running time of dynamic programming

$O(nm)$ on strings of lengths n and m

How long does it takes?

(straightforward implementation, random sequences of length n, ordinary desktop couple of years ago)

n	time
100	0.0008s
1,000	0.08s
10,000	8s
100,000	13 minutes (*)
1,000,000	22 hours (*)
10,000,000	3 months (*)
100,000,000	25 years (*)

Improvements of dynamic programming

Hunt-Szymanski algorithm for LCS: $O(m + n + r \log r)$

where $0 \leq r \leq mn$, $r = |\{(i, j) : S[i] = T[j]\}|$

Uses $O(r \log r)$ algorithm to find the longest increasing subsequence

Four Russians technique: $O(mn/\log m)$ for a constant σ

(precompute small squares, save time)

Hirschberg's algorithm: $O(mn)$ time but $O(m + n)$ memory

(compute where the paths crosses middle of the matrix,

divide and conquer)

Ukkonen's algorithm: $O(md)$ time where $d = d_E(S, T)$

Guess d, verify in a band of diagonals

Theory of edit distance computation

Compute d_E of two strings of length n

Exact algorithm: $O(n^2/\log n)$ time

Approximation algorithm: $(\log n)^{O(1/\epsilon)}$ approximation in $n^{1+\epsilon}$ time [Andoni, Krauthgamer, Onak 2010]

Conditional lower bound: If edit distance can be computed in $O(n^{2-\delta})$ for some constant $\delta > 0$, then SAT for formula with N variables and M clauses can be solved in $M^{O(1)}2^{(1-\epsilon)N}$ for a constant $\epsilon > 0$, which would violate Strong Exponential Time Hypothesis (SETH). [Backurs, Indyk 2015]

Approximate string matching (približné výskyty vzorky)

Find substrings of T s.t. $d_E(T[i..j], P) \leq k$

How to modify dynamic programming for edit distance?

$$A[0, i] = A[i, 0] = i$$

$$A[i, j] = \min \begin{cases} A[i-1, j-1] + c(S[i], T[j]) \\ A[i-1, j] + 1 \\ A[i, j-1] + 1 \end{cases}$$

Simple dynamic programming $O(mn)$

Subproblem: $A[i,j] = \min_\ell d_E(P[0..i], T[\ell..j])$

Recurrence:

$A[0,j] = 0$

$A[i,0] = i$

$$A[i,j] = \min \begin{cases} A[i-1,j-1] + c(P[i], T[j]) \\ A[i-1,j] + 1 \\ A[i,j-1] + 1 \end{cases}$$

Print all j s.t. $A[m,j] \leq k$ (ends of occurrences)

We need only 2 columns from A

What if we want the best start (with smallest d_E) for each end?

Simple improvement, faster on average

Compute correct values only for active cells

We call $A[i, j]$ active if $A[i, j] \leq k$

Let $L[j]$ be largest i s.t. $A[i, j]$ is active

Lemma: Adjacent values in a row or column of A differ by at most 1

Lemma: $L[j] \leq L[j-1] + 1$

Simple improvement, faster on average

```
1   Compute A[*,0];
2   L = k;
3   for(j=1; j<=n; j++) {
4     L2 = 0;
5     for(i=0; i<=L+1; i++) {
6       compute A[i,j];   // if unknown A[i,j-1], assume k+1
7       if(A[i,j]<=k) L2 = i;
8     }
9     L = L2;
10  }
```

Average-case running time $O(kn)$, worst-case $O(mn)$

Landau-Vishkin 1986 $O(kn)$

Diagonal number d : all values $A[i, j]$ where $j - i = d$

$L[d, e]$: maximum row i on diagonal d such that $A[i, i + d] \leq e$

```
1  // L[d,−1] = −1 (+ plus other boundary cases)
2  for(e=0; e<=k; e++) {
3     for(d= −e; d<=n; d++) {
4        i = min(m, max(L[d, e−1]+1, L[d−1,e−1], L[d+1,e−1]+1))
5        while(i<m && i+d<n && P[i]==T[i+d]) { i++; }
6        L[d,e] = i;
7        if(L[d,e]==m) { print occurrence ending at d+m }
8     }
9  }
```

Landau-Vishkin 1986 $O(kn)$

```
1  for(e=0; e<=k; e++) {
2    for(d=-e; d<=n; d++) {
3      i = min(m, max(L[d, e-1]+1, L[d-1,e-1], L[d+1,e-1]+1))
4      while(i<m && i+d<n && P[i]==T[i+d]) { i++; } // (*)
5      L[d,e] = i;
6      if(L[d,e]==m) { print occurrence ending at d+m }
7    }
8  }
```

$L[d, e]$: maximum row i on diagonal d such that $A[i, i + d] \leq e$

Clearly algorithm computes lower bound of $L[d, e]$

But correctness needs more proof

Line (*) replaced with LCA on suffix tree for P and T in $O(1)$

$i+=$ longest common prefix of $P[i..m]$ and $T[i + d..n]$

Baeza-Yates, Perleberg 1992

Divide P to $k+1$ substrings of size $r = \lfloor \frac{m}{k+1} \rfloor$ (last one possibly longer). Denote the set of these substrings $\mathcal{P}$.

Lemma: Let T' be a substring of T. If $d_E(T', P) \leq k$, at least one of the strings in $\mathcal{P}$ is a substring of T'.

Find occurrences of strings in $\mathcal{P}$ in T, search around each.

Average-case running time $O(n + n(k+1)m^2/\sigma^r)$

Filtering in general: Similar exact or heuristic techniques widely used

Burrows-Wheeler transform (BWT) 1994

T =banana$

Sort all rotations of the word lexicographically:

b	a	n	a	n	a	$
a	n	a	n	a	$	b
n	a	n	a	$	b	a
a	n	a	$	b	a	n
n	a	$	b	a	n	a
a	$	b	a	n	a	n
$	b	a	n	a	n	a

$	b	a	n	a	n	**a**
a	$	b	a	n	a	**n**
a	n	a	$	b	a	**n**
a	n	a	n	a	$	**b**
b	a	n	a	n	a	**$**
n	a	$	b	a	n	**a**
n	a	n	a	$	b	**a**

BWT = annb$aa

Can be computed using suffix array: BWT[i] = T[SA[i]-1] (or T[n] if SA[i]=0)

Reverse transformation

Get first column ($\mathbb{F}$) by sorting the last ($\mathbb{L}$):

$\mathbb{F}$		$\mathbb{L}$
\$	...	a
a	...	n
a	...	n
a	...	b
b	...	\$
n	...	a
n	...	a

Reverse transformation

Observations:

– $F[i]$ follows $L[i]$ in T

– jth occurrence of x in F is the same as jth occurrence of x in L

0:	**$**$_6$	b	a	n	a	n	**a**$_5$
1:	**a**$_5$	$	b	a	n	a	**n**$_4$
2:	**a**$_3$	n	a	$	b	a	**n**$_2$
3:	**a**$_1$	n	a	n	a	$	**b**$_0$
4:	**b**$_0$	a	n	a	n	a	**$**$_6$
5:	**n**$_4$	a	$	b	a	n	**a**$_3$
6:	**n**$_2$	a	n	a	$	b	**a**$_1$

Reverse transformation

– $F[i]$ follows $L[i]$ in T

– jth occurrence of x in F is the same as jth occurrence of x in L

\$$_6$	b	a	n	a	n	**a**$_5$
a$_5$	\$	b	a	n	a	**n**$_4$
a$_3$	n	a	\$	b	a	**n**$_2$
a$_1$	n	a	n	a	\$	**b**$_0$
b$_0$	a	n	a	n	a	**\$**$_6$
n$_4$	a	\$	b	a	n	**a**$_3$
n$_2$	a	n	a	\$	b	**a**$_1$

Find \$ in L, get $T[0]$ in F ($T[0] = b$, 1st)

Find 1st b in L, get $T[1]$ in F ($T[1] = a$, 3rd)

Find 3rd a in L, get $T[2]$ in F ($T[2] = n$, 2nd)

In general: If $T[i]$ in $L[j]$, get $T[i+1]$ in $F[j]$

Reverse transformation

0:	$\$_6$	a_5
1:	a_5	n_4
2:	a_3	n_2
3:	a_1	b_0
4:	b_0	$\$_6$
5:	n_4	a_3
6:	n_2	a_1

Sort L to get F, then build the following data structures:

Representation of F :

$: 0

a: 1

b: 4

n: 5

0: $

1: a

2: a

3: a

4: b

5: n

6: n

Representation of L:

$: 4

a: 0, 5, 6

b: 3

n: 1, 2

Use of Burrows-Wheeler transform in compression

T =ema.ma.mamu.mama.ma.emu.ema.sa.ma$

BWT =auaaaauaammsmmmmmm$....ae.e..ea.mm

In a given region of SA common prefixes

Preceded by similar letters

In our case '.' preceded by u,a

Regions with the same letter repeated or few letters mixed

Use of Burrows-Wheeler transform in compression

T =ema.ma.mamu.mama.ma.emu.ema.sa.ma$

BWT =auaaaauaammsmmmmmm$....ae.e..ea.mm

Regions with the same letter repeated or few letters mixed

Move-to-front recording: replace $T[i]$ by the number of distinct letters since last occurrence of $T[i]$ in $T[0..i-1]$

$.aemsu|auaaaauaammsmmmmmm$....ae.e..ea.mm

4, 1, 1, 0, 0, 0, 1, 1, 0, 3, 0, 3, 1, 0, 0, 0, 0, 0, 6, 6, 0, 0, 0, 4, 6, 2, 1, 1, 0, 1, 2, 2, 4, 0

Small numbers, many zeroes, in English text 50% zeroes

Use of Burrows-Wheeler transform in compression

Encode MTF of BWT by Huffman or arithmetic encoding

$T \rightarrow BWT \rightarrow MTF \rightarrow$ Huffman/arithmetic encoding $\rightarrow$ compressed T

e.g. bzip2 (with further details)

Entropy

Let $\Sigma = \{x_1, \ldots, x_\sigma\}$, probability of x_i is p_i

Entropy of this distribution is: $-\sum_{i=1}^{\sigma} p_i \lg p_i$

Entropy gives lower bound on the number of bits needed per character.

Huffman encoding assigns shorter code sequences to more frequent words, uses approximately $-\lg p_i$ bits to encode x_i

Arithmetic encoding has fewer rounding errors

Example: Entropy of T is 2.37, entropy of MTF of BWT is 2.18

A different reverse transformation (backwards)

$LF[i]$: row j in which $F[j]$ corresponds to $L[i]$

Example: $LF[2] = 6$

If i corresponds to $T[k..n]$, then $LF[i]$ corresponds to $T[k-1..n]$

0:	**$**$_6$	b	a	n	a	n	**a**$_5$
1:	**a**$_5$	$	b	a	n	a	**n**$_4$
2:	**a**$_3$	n	a	$	b	a	**n**$_2$
3:	**a**$_1$	n	a	n	a	$	**b**$_0$
4:	**b**$_0$	a	n	a	n	a	**$**$_6$
5:	**n**$_4$	a	$	b	a	n	**a**$_3$
6:	**n**$_2$	a	n	a	$	b	**a**$_1$

```
1  T[n] = $; s = 0;
2  for(i=n-1; i>=0; i--) { T[i] = L[s]; s = LF[s]; }
```

A different reverse transformation (backwards)

$LF[i]$: row j in which $F[j]$ corresponds to $L[i]$

If i corresponds to $T[k..n]$, then $LF[i]$ corresponds to $T[k-1..n]$

$C[x]$: the index of first occurrence of x in F

$rank[x, i]$: the number of occurrences of x in $L[0..i]$

$LF[i] = C[L[i]] + rank[L[i], i-1]$

i	$F[i]$	$L[i]$	$r[\$, i]$	$r[a, i]$	$r[b, i]$	$r[n, i]$
0:	$\$_6$	a_5	0	1	0	0
1:	a_5	n_4	0	1	0	1
2:	a_3	n_2	0	1	0	2
3:	a_1	b_0	0	1	1	2
4:	b_0	$\$_6$	1	1	1	2
5:	n_4	a_3	1	2	1	2
6:	n_2	a_1	1	3	1	2

Note: less practical decompression (more memory, result in reverse order)

Use of Burrows-Wheeler transform for string matching

FM index [Ferragina and Manzini 2000]

Occurrences of pattern P in T form an interval in $\mathcal{SA}$

Consider suffixes of P from shortest, update interval

Example: search for $\mathrm{P} = \mathtt{nan}$

$banana**a**	$banana**a**	$banana**a**	$banana**a**
a$banan**n**	a$banan**n**	a$banan**n**	a$banan**n**
ana$ba**n**	ana$ba**n**	**an**a$ba**n**	ana$ba**n**
anana$**b**	anana$**b**	**an**ana$**b**	anana$**b**
banana**$**	banana**$**	banana**$**	banana**$**
na$ban**a**	**n**a$ban**a**	na$ban**a**	na$ban**a**
nana$b**a**	**n**ana$b**a**	nana$b**a**	**na n**a$b**a**

Counting occurrences of P using FM index

FM index:

$C[x]$: the index of first occurrence of x in F

$rank[x, i]$: the number of occurrences of x in $L[0..i]$

```
1  l = 0; r = n;
2  for(i = m-1; i >= 0; i--) {
3    a = P[i];
4    l = C[a] + rank[a, l-1];
5    r = C[a] + rank[a, r] - 1;
6    if(l > r) return 0; // no occurrences
7  }
8  return r - l + 1;
```

To report positions, we also need suffix array SA

Counting occurrences of P using FM index

$C[x]$: the index of first occurrence of x in F

$rank[x, i]$: the number of occurrences of x in $L[0..i]$

Update of ℓ: $\ell = C[a] + rank[a, \ell - 1]$

Update of r: $r = C[a] + rank[a, r] - 1$

	$X\ldots$	a
	$aX\ldots$	
	$aY\ldots$	
ℓ'	$aP\ldots$	
r'	$aP\ldots$	
	$Y\ldots$	a
ℓ	$P\ldots$	
	$P\ldots$	
r	$P\ldots$	

FM index

$C[x]$: the index of first occurrence of x in F

$rank[x, i]$: the number of occurrences of x in $L[0..i]$

Example:

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
T[i]	e	m	a	.	m	a	.	m	a	m	u	.	m	a	m	a	.	m	a	.	e	m	u	$
SA[i]	23	19	16	3	11	6	18	15	2	5	13	8	0	20	17	14	1	4	12	7	21	9	22	10
L[i]	u	a	a	a	u	a	m	m	m	m	m	m	$	.	.	a	e	.	.	.	e	a	m	m
r$(i)	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1
r.(i)	0	0	0	0	0	0	0	0	0	0	0	0	0	1	2	2	2	3	4	5	5	5	5	5
ra(i)	0	1	2	3	3	4	4	4	4	4	4	4	4	4	4	5	5	5	5	5	5	6	6	6
re(i)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	2	2	2	2
rm(i)	0	0	0	0	0	0	1	2	3	4	5	6	6	6	6	6	6	6	6	6	6	6	7	8
ru(i)	1	1	1	1	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2

x	$	.	a	e	m	u
C(x)	0	1	6	12	14	22

```
 0 23 $
 1 19 .emu$
 2 16 .ma.emu$
 3  3 .ma.mamu.mama.ma.emu$
 4 11 .mama.ma.emu$
 5  6 .mamu.mama.ma.emu$
 6 18 a.emu$
 7 15 a.ma.emu$
 8  2 a.ma.mamu.mama.ma.emu$
 9  5 a.mamu.mama.ma.emu$
10 13 ama.ma.emu$
11  8 amu.mama.ma.emu
12  0 ema.ma.mamu.mama.ma.emu$
13 20 emu$
14 17 ma.emu$
15 14 ma.ma.emu$
16  1 ma.ma.mamu.mama.ma.emu$
17  4 ma.mamu.mama.ma.emu$
18 12 mama.ma.emu$
19  7 mamu.mama.ma.emu$
20 21 mu$
21  9 mu.mama.ma.emu$
22 22 u$
23 10 u.mama.ma.emu$
```

Summary: Burrows-Wheeler transform

- BWT can be computed in $O(n)$ time via suffix array
- Reverse transformation also in $O(n)$
- Can be used in compression, groups the same letters together
- Can be used in FM index for string matching
- We need to store ranks in a smaller memory - next topic

Succint data structures

We usually count memory in **words** of some size $w \geq \lg n$

each word can hold pointer, index, count, symbol etc.

Now we will count memory in **bits**

Lower bound: to store any $x \in \mathcal{U}$, we need at least $\mathrm{OPT} = \lg |\mathcal{U}|$ bits

Implicit data structure uses $\mathrm{OPT} + \mathrm{O}(1)$ bits

Example: binary heap, sorted array

Succint data structure uses $\mathrm{OPT} + \mathrm{o}(\mathrm{OPT})$ bits

Leading constant 1 plus some lower-order terms

Compact data structure uses $\mathrm{O}(\mathrm{OPT})$ bits

Succint structure for binary rank and select

Bit vector A[0..n-1]

rank(i) = number of bits set to 1 in A[0..i]

select(i) = position of the i-th bit set to 1

Example:

i	0	1	2	3	4	5	6	7
$A[i]$	0	1	1	0	1	0	0	1

rank(3) = 2, rank(4) = 3

select(1) = 1, select(3) = 4

Goal:

rank, select in $O(1)$ time

structure needs $n + o(n)$ bits of memory

we will concentrate on rank

Succint structure for rank (Jacobson 1989)

- Divide bit vector to super blocks of size $t_1 = \lg^2 n$
- Divide each super block to blocks of size $t_2 = \frac{1}{2} \lg n$
- Keep rank at each super block boundary

  $O(\frac{n}{t_1} \cdot \log n) = O(n/\log n) = o(n)$ bits
- Keep rank within super block at each block boundary

  $O(\frac{n}{t_2} \cdot \log t_1) = O(n \log\log n/\log n) = o(n)$ bits
- Each block stored as a binary number using t_2 bits

  n bits
- For each of 2^{t_2} possible blocks and each query keep the answer

  $O(2^{t_2} \cdot t_2 \cdot \log t_2) = O(\sqrt{n} \log n \log\log n) = o(n)$ bits

Succint structure for rank (Jacobson 1989)

R1: array of ranks at superblock boundaries

R2: array of ranks at block boundaries within superblocks

R3: precomputed rank for each block type and each position

B: bit array

```
1  rank(i) {
2     superblock = i/t1;   //integer division
3     block = i/t2;
4     index = block*t2;
5     type = B[index..index+t2-1];
6     return R1[superblock]+R2[block]+R3[type, i%t2]
7  }
```

Succint structure for select

- Let $t_1 = \lg n \lg\lg n$, $t_2 = (\lg\lg n)^2$.
- Store select($t_1 \cdot i$) for $i = 0, \ldots, n/t1$;
  this divides bit vector into super-blocks of unequal size.
- Large super-blocks of size $\geq t_1^2$: store array of indices of 1 bits.
- Small super-block of size $\leq t_1^2$: repeat with t_1:
  store select($t_2 \cdot i$) within super-block for $i = 0, \ldots, n/t2$;
  this divides small super-blocks into blocks of unequal size.
- Large blocks of size $\geq t_2^2$: store relative indices of all 1 bits.
- Small blocks of size $< t_2^2$: store as t_2^2-bit integer,
  plus a lookup table of all answers.

Wavelet tree (Grossi, Gupta, Vitter 2003)

$\Sigma_0 = \{\$, ., a\}$ $\Sigma_1 = \{e, m, u\}$

```
i    0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22
S[i] e  m  a  .  m  a  .  m  a  m  u  .  m  a  m  a  .  m  a  .  e  m  u
B[i] 1  1  0  0  1  0  0  1  0  1  1  0  1  0  1  0  0  1  0  0  1  1  1
S0   a.a.a.aa.a.$      S1   emmmmummmemu
```

$\Sigma_{00} = \{\$\}, \Sigma_{01} = \{., a\}$, $\Sigma_{010} = \{.\}, \Sigma_{011} = \{a\}$
$\Sigma_{10} = \{e\}, \Sigma_{11} = \{m, u\}$, $\Sigma_{110} = \{m\}, \Sigma_{111} = \{u\}$

```
i     0  1  2  3  4  5  6  7  8  9 10 11
S0[i] a  .  a  .  a  .  a  a  .  a  .  $
B0[i] 1  1  1  1  1  1  1  1  1  1  1  0
S00  a.a.a.aa.a.  S01 $
```

Store

B[i] =110010010110101001001110

B0[i] = 111111111110 B1[i] = 011111111011

B01[i] = 10101011010 B11[i] = 000100001

Dynamic texts (Navarro, Nekrich 2014)

Access, rank, select

Insert/delete character

All in $\mathrm{O}(\log n / \log\log n)$ amortized

Additional memory $\mathrm{o}(n \lg \sigma) + \mathrm{O}(\sigma \lg n)$

FM index with wavelet trees

- Find exact occurrences of pattern in $O(m + k)$ time
- Requires arrays C and $rank$, $rank$ has size n log σ words
- Wavelet trees store $rank$ in $n \lg \sigma + o(n\sigma) + O(\sigma \lg n)$ bits
  time increases by lg σ factor
- Printing occurrences also needs array SA
  store only some values, recompute the rest using LF

Succint data structures

- Data structure uses $\mathrm{OPT} + \mathrm{o}(\mathrm{OPT})$ bits of memory
  and supports fast operations
- Rank and select on a binary vector of length n in $\mathrm{O}(1)$ time
- Wavelet tree supports rank over larger alphabet in $\mathrm{O}(\log \sigma)$ time,
  uses binary rank
- FM index counts occurrences of pattern P in T in $\mathrm{O}(\mathrm{m} \log \sigma)$ time,
  uses BWT and wavelet tree

Next:

- Compressed data structures (for rank)
- Succint data structure for binary trees

Compressed structure for rank (Raman, Raman, Rao 2002)

- Compressed size of bit vector + $o(n)$ bits
- Need to reduce the following part:

  Each block stored as a binary number using t_2 bits
- Blocks with many 0s or many 1s stored using fewer bits
- For each block store the number of 1s (class)

  $O(\frac{n}{t_2} \log t_2) = O(n \log\log n / \log n) = o(n)$ bits
- For a block with x 1s store its signature: index in lexicographic order of all binary strings of size t_2 with x 1s

  $\lceil \lg \binom{t_2}{x} \rceil \leq \lg 2^{t_2} = t_2$ bits (overall at most $\frac{n}{t_2} t_2 = n$ bits)
- Rearrange the table with answers for all possible blocks of size t_2

  Add signature boundaries in compressed bit vector $o(n)$

Compressed structure for rank (RRR)

$t_2 = 3$

Cl	Sig	Length	Block	Answers
0	0	0	000	0 0 0
1	0	2	001	0 0 1
	1		010	0 1 1
	2		100	1 1 1
2	0	2	011	0 1 2
	1		101	1 1 2
	2		110	1 2 2
3	0	0	111	1 2 3

Original bits: 000|101|001|111|111

Number of 1s in each block: 00|10|01|11|11

Index of block: $\epsilon|01|00|\epsilon|\epsilon$

Where each block starts (within superblock): 0000|0000|0010|0100|0100

Compressed structure for rank (RRR)

rank(i):

- superblock = i/t1 (integer division)
- block = i/t2
- index = S1[superblock]+S2[block]
- class = C[block]
- length = L[class]
- signature = B[index..index+length-1]
- return R1[superblock]+R2[block]+R3[class, signature, i%t2]

Analysis of RRR structure

- Let S be a string in which $a \in \Sigma$ occurs n_a times
- Its **entropy** is $H(S) = \sum_a \frac{n_a}{n} \lg \frac{n}{n_a}$
- $H(S) \leq \lg \sigma$ (lower if some characters more frequent than others)
- RRR structure for bit vector B uses $nH(B) + o(n)$ **bits**

Stirling's approximation of $n!$

$$n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n (1 + O(1/n))$$

$$\ln(n!) = n\ln(n) - n + O(\ln(n))$$

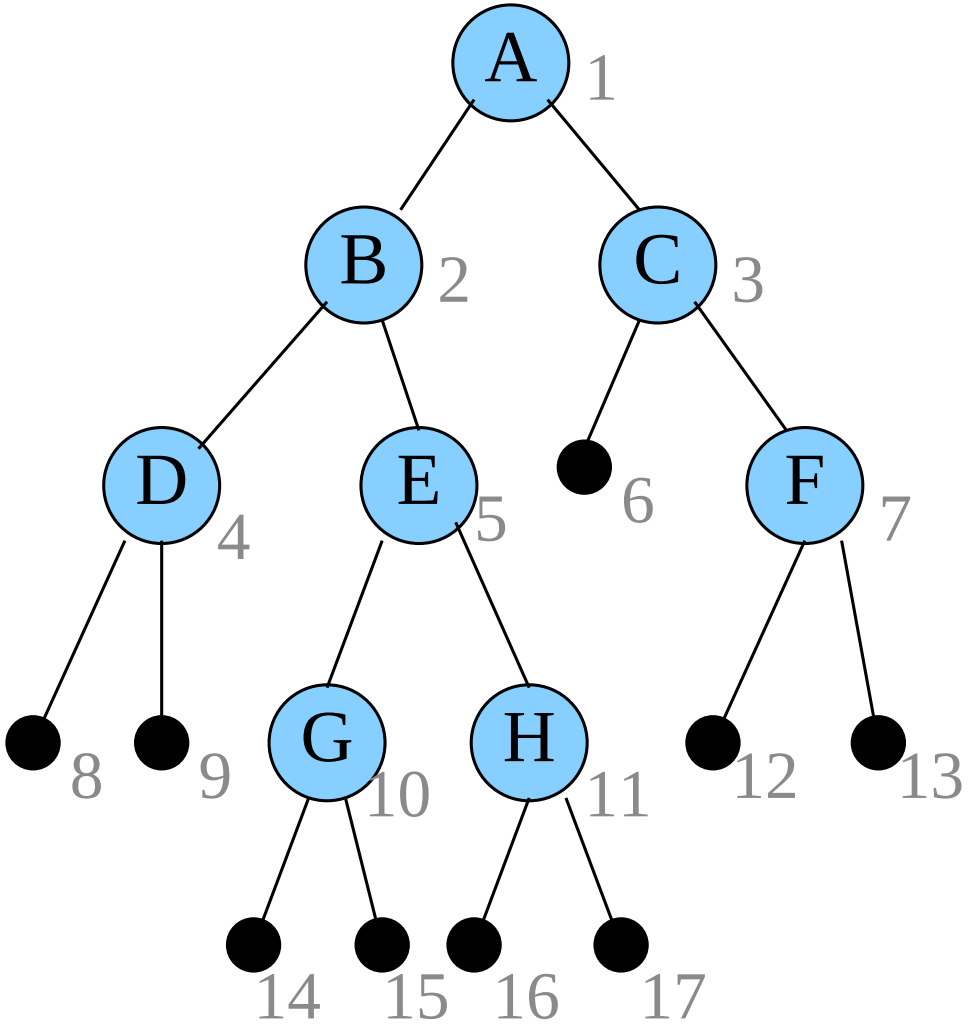
Uses of RRR structure

- Store binary rank structures in the wavelet tree for text T
  overall $nH(T) + o(nH(T))$ bits
- Instead of wavelet tree, store indicator vector for each $a \in \Sigma$
  overall $nH(T) + O(n) + o(\sigma n)$ bits
  $O(1)$ per rank query
- FM index needs rank in BWT of T, which might be even better compressible than T

Succint binary trees

- Consider all binary trees with n nodes
- Classical trees with pointers use $\Omega(n \log n)$ bits
- OPT is cca $2n$ bits (proof later)
- **Goal:** Use $2n + o(n)$ memory,
  support operations left child, right child, parent in $O(1)$
- Add $n + 1$ auxiliary leaves
- Nodes are numbers $\{1, \ldots, 2n + 1\}$ in level order (BFS)

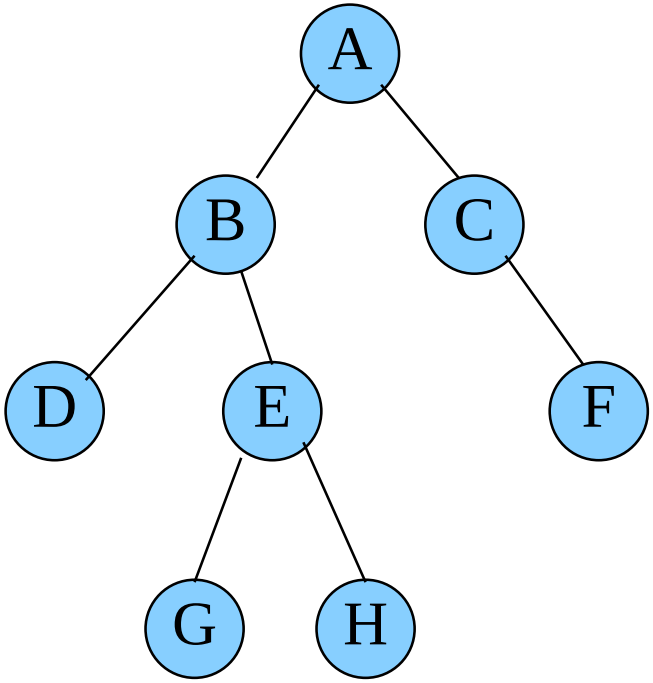
Succint binary trees: level order representation

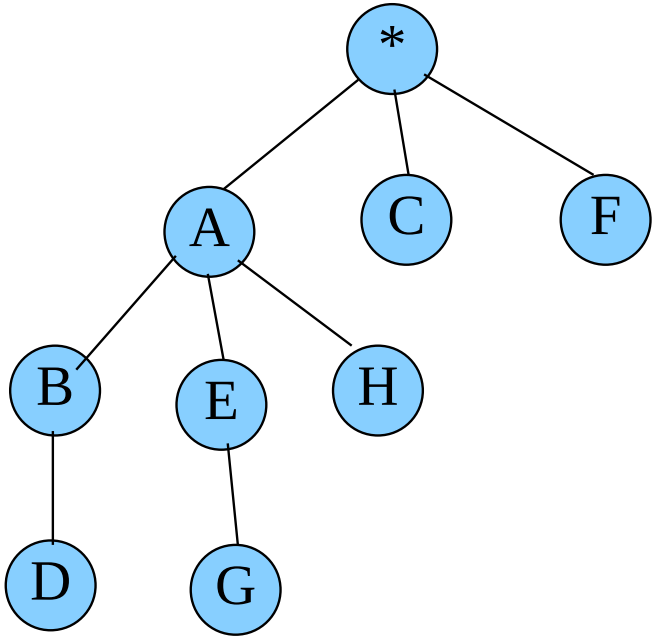


Succint binary trees

- Consider all binary trees with n nodes
- **Goal:** Use $2n + o(n)$ memory,
  support operations left child, right child, parent in $O(1)$
- Add $n + 1$ auxiliary leaves
- Nodes are numbers from $\{1, \ldots, 2n + 1\}$
  Using rank can be mapped to $\{1, \ldots, n\}$
  These can be then used as indices to arrays with additional data
- Can be part of binary tries, binary suffix trees
  Extensions to larger alphabets exist
- Static trees only, construction requires more memory

Equivalence of binary trees and rooted ordered trees





Rooted ordered tree as a well-parenthesized expression:

((( )) (( )) ( )) ( ) ( )

ABDDBEGGEHHACCFF

Counting well-parenthesized expresions

$X(n, m, k)$: set of all sequences containing n times 1, m times -1 with all prefix sums $\geq k$

Easy: $|X(n, m, -\infty)|$

Want: $|X(n, n, 0)|$

Prove: $|X(n, n, -\infty) \setminus X(n, n, 0)| = |X(n-1, n+1, -\infty)|$

Then: $|X(n, n, 0)| = \binom{2n}{n} - \binom{2n}{n-1} = \frac{(2n)!}{n!n!} - \frac{(2n)!}{(n-1)!(n+1)!}$

$= \frac{(2n)!(n+1-n)}{n!(n+1)!} = \binom{2n}{n}/(n+1)$

Example:

$|X(3, 3, -\infty) \setminus X(3, 3, 0)| = |X(2, 4, -\infty)| = 15$

$|X(3, 3, -\infty)| = 20$

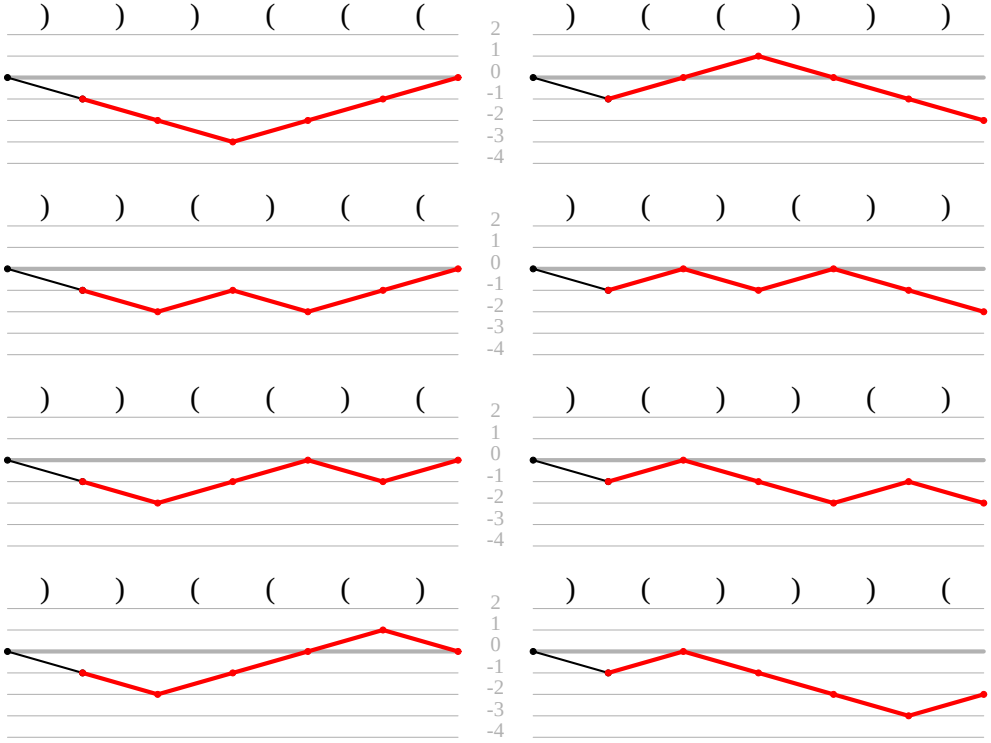
$|X(3, 3, 0)| = 20 - 15 = 5 = C_3$

Counting well-parenthesized expresions

$X(n, m, k)$: set of all sequences containing n times 1, m times -1 with all prefix sums $\geq k$

$$|X(n, n, -\infty) \setminus X(n, n, 0)| = |X(n-1, n+1, -\infty)|$$

Consider $n = 3$, first four examples out of 15:



Introduction to hashing

- Universe $U = \{0, ..., u - 1\}$
- Table of size m
- Hash function $h : U \to \{0, ..., m - 1\}$
- Let X be the set of elements currently in the hash table, $n = |X|$
- We will assume $n = \Theta(m)$

Totally random hash function

(a.k.a uniform hashing model)

- select $h(x)$ for each $x \in U$ uniformy independently
- not practical - storing this function requires $u \lg m$ bits
- used for simplified analysis of hashing in ideal case

Universal family of hash functions

- set of hash functions H
- $\exists c$ such that for any distinct $x, y \in U$ we have
  $\Pr(h(x) = h(y)) \leq c/m$
- Example: choose prime $p \geq u$
  $H_p = \{h_a \mid h_a(x) = (ax \bmod p) \bmod m), 1 \leq a \leq p-1\}$

Hashing with chaining and universal family of h.s.

- Linked list (or vector) for each hash table slot
- Let c_i be the length of chain i
- For a fixed $x \in U$, $i = h(x)$ we have
  $E_{h \in H}[c_i] \leq 1 + cn/m = O(1)$
- However $E_{h \in H}[\max_i c_i] = O(\sqrt{n})$
- For a totally random function this is $O(\log n / \log \log n)$

Perfect hashing

- Fredman, Komlos, Szemeredi 1984
- Top level: universal hash function to table of size $\Theta(n)$
- Second level: bucket i with c_i elements hashed to a table of size αc_i^2
- If any collision at second table, choose a new hash function
- If overall size too big, start from scratch
- Overall $O(1)$ deterministic search, $O(n)$ space, $O(n)$ expected preprocessing
- Dynamic version resizes similarly as vector, $O(1)$ amortized expected update, but still $O(1)$ deterministic search

Bloom filter (Bloom 1970)

Supports insert x, test if x is in the set

– may give false positives, e.g. claim that x is in the set when it is not

– false negatives do not occur

Algorithm

a bit vector $B[0, \ldots, m-1]$,

k hash functions $h_1, \ldots, h_k : U \to \{0, \ldots, m-1\}$

insert(x): set $B[h_1(x)], \ldots, B[h_k(x)]$ to 1

contains(x): check if $B[h_1(x)], \ldots B[h_k(x)]$ are all 1

– if yes, claim x is in the set, but possibility of error

– otherwise answer no, surely true

Analysis: If all h_i are totally random and independent, the probability of error is approximately $(1 - e^{-nk/m})^k$.

Bloom filter analysis

If all h_i are totally random and independent, the probability of error is approximately $(1 - e^{-nk/m})^k$.

- Insert n elements, each with k hash functions
- $\Pr(B[i] = 1) = 1 - (1 - 1/m)^{nk} \approx 1 - e^{-nk/m}$, denote p
- Expected number of 1's is pm, but individual $B[i]$ not independent
- False positive: k randomly chosen slots all 1
- If the hash table exactly pm 1's and $(1 - p)m$ 0's
  probability of false positive would be p^k as claimed
- But we do not know exact count of 1's
  therefore individual samples not independent
- With high prob. the count close to pm (modified Chernoff bound)

Why independence not true?

Insertion of n elements creates unknown number of 1's.

Each hash function in search tells us something about this count.

This would happen even if $B[i]$ independent.

Simplified toy example

Hash table with 2 slots,

each set to 1 or 0 uniformly independently.

Twice sample a slot (indep., uniformly),

results X_1, X_2

$\Pr(X_2 = 1) = 1/2$

$\Pr(X_2 = 1 \mid X_1 = 1) = 3/4$

B_1	B_2	X_1	X_2
0	1	B_2	$B_1 = 0$
0	1	B_2	$B_2 = 1$
1	0	B_1	$B_1 = 1$
1	0	B_1	$B_2 = 0$
1	1	B_1	$B_1 = 1$
1	1	B_1	$B_2 = 1$
1	1	B_2	$B_1 = 1$
1	1	B_2	$B_2 = 1$

Bloom filter analysis

- If hash functions are totally random and independent, the probability of error is approximately $(1 - e^{-nk/m})^k$
- For $k = \ln(2)m/n$, get error $c^{-m/n}$, where $c = 2^{\ln(2)} \approx 1.62$
- To get error rate p for some n, we need $m = n \lg(1/p) \lg(e)$
- For 1% error, we need about $m = 10n$ bits of space and $k = 7$
- Memory and error rate are independent of the universe size
- Hash table needs at least to store data (e.g. in $n \lg u$ bits)
- If we used $k = 1$ (one hash function), we need $99.5n$ for 1% error

Nearest neighbor

- Preprocess set X
- Query(x):

  find $y \in X$ minimizing $d(x, y)$ for a given distance measure d

Approximate near neighbour (r, c)-NN

- Preprocess a set X for given $r > 0$, $c > 1$
- Query(x):

  if there is $y \in X$ such that $d(y, x) \leq r$,

  return $z \in X$ such that $d(z, x) \leq c \cdot r$
- Locality sensitive hashing returns such z with probability $\geq f$

Algorithm

For Hamming distance on strings of length d

- Hash each string using L hashing functions,
  each using k randomly chosen positions (random projections)
- Then rehash to obtain total space $O(nd + nL)$
- Given x, find collisions in each hash function,
  report if any at distance at most cr
- If checked more than $3L$ collisions, stop
- Query time $O(Lk + Ld)$

$k = O(\log n)$, $L = O(n^{1/c})$

Probability of failure less than some constant $f < 1$

Can be boosted by repeating independently

A LSH family

A family of hash functions H is (r_1, r_2, p_1, p_2)-sensitive if for any $x, y \in U$ we have

If $d(x, y) \leq r_1$ then $\Pr_{h \in H}(h(x) = h(y)) \geq p_1$

If $d(x, y) \geq r_2$ then $\Pr_{h \in H}(h(x) = h(y)) \leq p_2$

We want $p_1 > p_2$, $r_1 = r$, $r_2 = cr$

Example:

Consider binary strings of length d under Hamming distance

Let $h_i(x)$ be the i-th bit of x

$(r, cr, 1 - \frac{r}{d}, 1 - \frac{cr}{d})$-sensitive family of size d

Amplification of (r_1, r_2, p_1, p_2)-sensitive family H

AND amplification

Randomly choose $h_1, \ldots, h_k \in H$

Let $g(x) = (h_1(x), \ldots, h_k(x))$

$g(x) = g(y)$ iff $h_i(x) = h_i(y)$ for **all** $i \in \{1, \ldots, k\}$

Such g form (r_1, r_2, p_1^k, p_2^k)-sensitive family

OR amplification

Randomly choose $h_1, \ldots, h_L \in H$

Create L hash tables, one for each h_i, take union of hits from all tables

Consider $g(x) = g(y)$ iff $h_i(x) = h_i(y)$ for **some** $i \in \{1, \ldots, L\}$

Similar to $(r_1, r_2, 1 - (1 - p_1)^L, 1 - (1 - p_2)^L)$-sensitive family

Combine to get $(r_1, r_2, 1 - (1 - p_1^k)^L, 1 - (1 - p_2^k)^L)$-sensitive family

Outline of analysis

Consider $(r_1, r_2, 1-(1-p_1^k)^L, 1-(1-p_2^k)^L)$-sensitive family

$k = \lceil \log_{1/p_2} n \rceil$, $L = n^\rho / p_1$

$\rho = \log(1/p_1)/\log(1/p_2)$; for Hamming dist. $\rho \leq 1/c$

Probability of collision of x and $y \in X$ s.t. $d(x,y) \leq r_1$ is

$\geq 1-(1-p_1^k)^L \geq 1 - 1/e \approx 0.63$

Probability of collision x and $y \in X$ s.t. $d(x,y) \geq r_2$ in one hash table is

$\leq p_2^k = 1/n$

Expected number of collisions in all tables is $\leq L$

By Markov inequality $\geq 3L$ collisions with probability $\leq 1/3$

Prob. of getting a good collision and not too many collisions

$\geq (1-1/3) + (1-1/e) - 1 = 1 - 1/3 - 1/e \approx 0.3 > 0$

Minimum hashing

Jaccard index

similarity of two sets $J(A, B) = |A \cap B|/|A \cup B|$,

distance d(A,B) = 1-J(A,B)

used e.g. for approximate duplicate document detection based on set of words in the document

Minimum hash of set A

$\text{minhash}_h(A) = \min_{x \in A} h(x)$

Assume totally random hash function h and no collisions in $A \cup B$

$\Pr(\text{minhash}_h(A) = \text{minhash}_h(B)) = J(A, B) = 1 - d(A, B)$

$(r_1, r_2, 1 - r_1, 1 - r_2)$-sensitive family

Again can we apply amplification

External memory model, I/O model

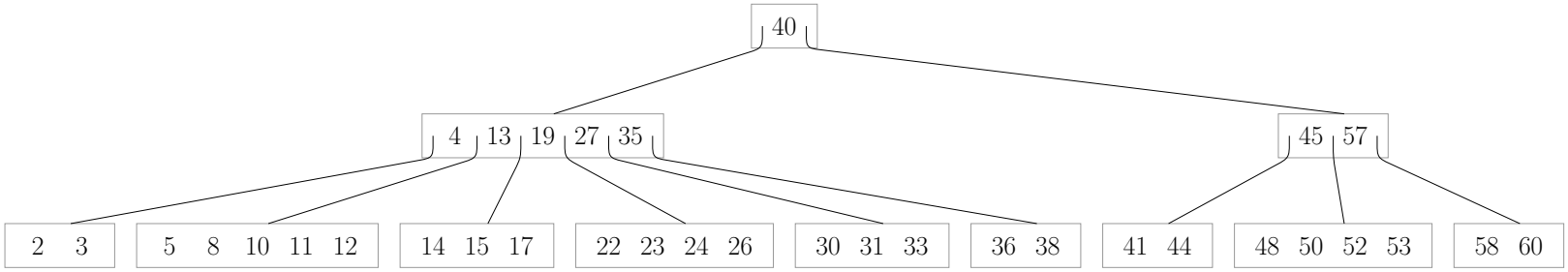
- big and slow disk, fast memory of a limited size M words
- disk reads/writes in blocks of size B words
- when analyzing algorithms, count how many blocks are read or written (**memory transfers**)

Example:

scanning through n elements: $O(\lceil n/B \rceil)$ transfers

B-tree

- parameter T
- node v has $v.n$ keys and 0 or $v.n + 1$ children
- keys in a node are sorted, each subtree contains only values between two successive keys in the parent
- all leaves are in the same depth
- for each node v except root satisfies $T - 1 \leq v.n \leq 2T - 1$
- in root, $1 \leq v.n \leq 2T - 1$



B-tree insert

- Find the leaf where the new key belongs
- If leaf has $2T - 1$ keys:

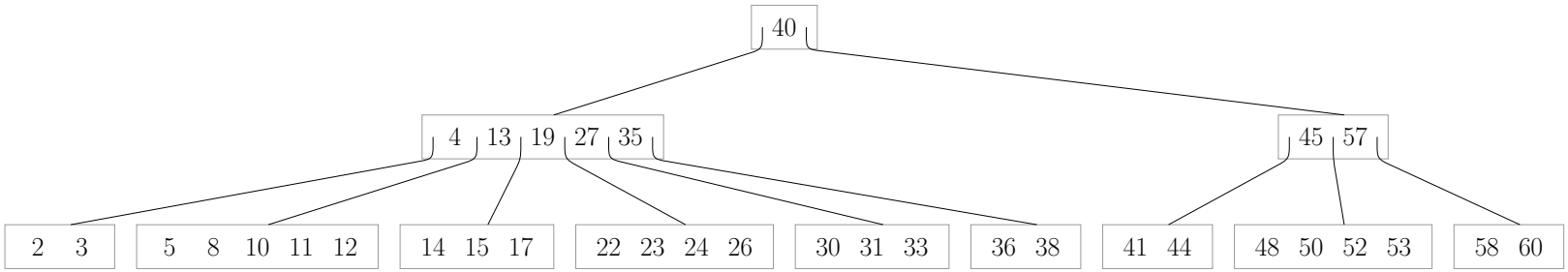
  Split it into two leaves, each with $T - 1$ keys

  Insert original median to the parent (recursively)

  If the recursion reaches the root and root is full,

  create a new root with 2 children

- New element can be now inserted into its leaf



External MergeSort

- Create sorted runs of size M
- Repeatedly merge $M/B - 1$ runs into one:

  read one block from each run, use one block for output

  when output block gets full, write it to disk

  when some input gets exhausted, read next block from the run (if any)
- Overall $\log_{M/B-1}(n/M)$ merging passes through data

External memory model: summary

- B-trees can do search tree operations (insert, delete, search/predecessor) in $O(\log_{B+1} n) = O(\log n / \log(B+1))$ memory transfers
- Sorting $O((n/B) \log_{M/B}(n/M))$ memory transfers

Cache oblivious model

- Algorithm in external memory model:
  - explicitly requests block transfers,
  - knows B,
  - controls memory allocation
- Algorithm in cache oblivious model:
  - does not know B or M,
  - algorithm requests reading/writing from disk,
  - automated caching,
  - memory M/B slots, each holding one block from disk

Cache operation

Algorithm requests reading/writing from disk

- cache M/B slots, each holding one block
- if the block containing request in cache, no transfer
- else replace one slot with block holding requested item, write original block if needed (1 or 2 transfers)
- which one to replace: classical on-line problem of paging

Paging

- Cache with k slots, each holds one page
- Sequence of page requests
- If requested page not in cache, bring it in and replace some other page (**page fault**)
- **Goal:** minimize the number of page faults
- **Offline optimum:**

  At a page fault remove the page that will not be used longest
- Example of an **on-line algorithm: FIFO**

  At a page fault remove the page that is longest time in cache

Paging

- On-line paging algorithm is **k-competitive**, if it always uses at most $k \cdot$ OPT page faults, where OPT is off-line optimum for the same input
- On-line algorithm is **conservative** if in a segment of requests containing at most k distinct pages, it needs $\leq k$ page faults
- Each conservative algoritjm is k-competitive
- FIFO is conservative and thus also k-competitive
- No deterministic algorithm can be better than k-competitive
- Conservative paging algorithm on memory with k slots uses at most $k/(k-h)\text{OPT}_h$ page faults where OPT_h is the off-line optimum for h slots $(h \leq k)$
- In fact it is possible to improve ratio to $k/(k-h+1)$

Cache oblivious model

- Algorithm in cache oblivious model:
  - does not know B or M,
  - algorithm requests reading/writing from disk,
  - automated caching,
  - memory M/B slots, each holding one block from disk,
  - assumption: paging done by offline optimum,
  - usually asymptotially equivalent to FIFO on memory 2M
- Advantages of cache oblivious algorithms:
  - no need to know B
  - may adapt to changing M or B
  - also good for memory hierachy (multiple caches, disk, network)
  - often the same complexity as in external memory model

Static cache-oblivious search trees, Prokop 1999

- Perfectly balanced binary search tree with nodes stored on disk in van Emde Boas order
- Search by the usual method, $O(\log_{B+1} n)$ block transfers
  (the same as B-trees for known B)
- For comparison: how many transfers needed for binary search?
  What about tree with nodes in pre-order or level-order?
  How to store the tree if we know B?

van Emde Boas order

- Split tree of height $\lg n$ into top and bottom,
  each of height $\frac{1}{2} \lg n$
- Top: a small tree with about $\sqrt{n}$ nodes
- Bottom: about $\sqrt{n}$ small trees, each about $\sqrt{n}$ nodes
- Print each of these small trees recursively, concatenate results

Example: A tree with 4 levels is split into 5 trees with 2 levels.

Resulting ordering:

```
              1
       2                 3
   4       7       10        13
5   6   8   9  11  12  14  15
```

Ordered file maintenance

- Maintain n items in an array of size $O(n)$ with gaps of size $O(1)$
- Updates: delete item, insert item after a given items

  (similar to linked list)
- Update rewrites interval of size $O(\log^2 n)$ amortized, in $O(1)$ scans
- Done by keeping appropriate density in a hierarchy of intervals
- We will not cover details

Dynamic cache oblivious trees

- Keep elements in ordered file in a sorted order
- Build a full binary tree on top of array (segment tree)
- Each node stores maximum in its subtree
- Tree stored in vEB order
- When array gets too full, double the size, rebuild everything

Search: check max in left child and decide to move left or right.

Follows a path from root, uses $O(\log_B n)$ transfers

Update: search to find the leaf, update ordered file,

then update all ancestors of changed values by postorder traversal

Improvement of update time by bucketing

Model of computation: word RAM

- Consider universe $\mathcal{U} = \{0, \ldots, 2^w - 1\}$
- Word: w-bit unsigned integer
- Input, output, queries, etc. are given in words
- Memory is an array of $\mathfrak{m}$ cells of size w
- Words can serve as pointers (indices)
- We need $w \geq \lg(\mathfrak{m})$, otherwise we cannot index whole memory
- If $\mathfrak{n}$ is the problem size, this implies $w \geq \lg(\mathfrak{n})$
- Program may use "C-style" operations in $O(\mathfrak{n})$ on words, such as $+, -, *, /, \%, \&, |, \gg, \ll, <, >$

Predecessor problem

- Maintain a set words over universe U
- Operations insert, delete, predecessor, successor
- Predecessor of $x \in U$: $\max\{y \in S \mid y < x\}$

n: the number of elements in the set

w: size of word

$u = 2^w$: size of universe

Binary search trees: $O(\log n)$ time, $O(n)$ words

Rank/select (static only): $O(1)$ time, $O(2^w/w)$ words

Today vEB: $O(\log w)$ time, $\Omega(2^w)$ words

Improvements: $O(\log w)$ time, $O(n)$ words

Note: if $u = n^{O(1)}$, we have $O(\log w) = O(\log \log n)$

van Emde Boas tree (vEB) (1977)

Goal: running time governed by recurrence

$$T(w) = T(w/2) + O(1)$$

$$T(u) = T(\sqrt{u}) + O(1)$$

Result: $T(w) = O(\log w)$, $T(u) = O(\log \log u)$

Approach: Split U into blocks of size $\sqrt{u}$ or split words into halves

Hierarchical coordinates: $x = \langle b, i \rangle$

where b is block ID, i ID within block

$$x = b\sqrt{u} + i$$

$$b = x/\sqrt{u}$$

$$i = x\%\sqrt{u}$$

van Emde Boas data structure

Structure V for universe size u contains:

- Array V.blocks of size $\sqrt{u}$
  – each element pointer to vEB for universe of size $\sqrt{u}$
- Another vEB V.summary for universe of size $\sqrt{u}$
  – contains as elements IDs of non-empty blocks
- Integer V.max contains maximum element in V
- Integer V.min contains minimum element in V
  – this minimum not stored elsewhere (blocks/summary)
  – this is important for achieving running time

Example of a van Emde Boas tree

$w = 4, \{1 \mid 4, 5, 6, 7 \mid 9, 10 \mid 12, 13, 14\}$

Root structure

min 1, max 14

summary $\{1, 2, 3\}$

blocks $[\emptyset, \{0, 1, 2, 3\}, \{1, 2\}, \{0, 1, 2\}]$

Subtrees for $w = 2$

For set $\{1 \mid 2, 3\}$

min 1, max 3, summary $\{1\}$ blocks $[\emptyset, \{0, 1\}]$

For set $\emptyset$

min None, max None, summary $\emptyset$ blocks $[\emptyset, \emptyset]$

For set $\{0, 1 \mid 2, 3\}$

min 0, max 3, summary $\{0, 1\}$ blocks $[\{1\}, \{0, 1\}]$

...

Predecessor query

```
1  Predecessor(V, x = <b,i>) {
2      if (V.blocks[b].min != None && i > V.blocks[b].min) {
3          return <b, Predecessor(V.blocks[b], i)>;
4      } else {
5          bb = Predecessor(V.summary, b);
6          if(bb == None) {
7              if(x > V.min) return V.min;
8              else return None;
9          }
10         return <bb, V.blocks[bb].max>;
11     }
12 }
```

Insert operation

```
1   Insert(V, x = <b,i>) {
2       if (V.min == None) {
3           V.min = V.max = x;
4           return;
5       }
6       if (x < V.min) swap(x, V.min);
7       if (x > V.max) V.max = x;
8       if (V.blocks[b].min == None) {
9           Insert(V.summary, b);
10      }
11      Insert(V.blocks[b], i);
12  }
```

vEB trees via hash tables

Improving memory:

– do not store empty vEB structures

– array V.blocks replaced by a hash table with dynamic perfect hashing

The memory is proportional to the number of vEB structures:

– hash table proportional to the number of children

– charge space in hash table to children, the rest $O(1)$ per node

The number of vEB structures $O(n \log w)$:

– charge each vEB to the minimum element

– in summary represent each block by minimum

– each element minimum at most twice on one level

Total memory $O(n \log w)$ words,

can be improved to $O(n)$ by careful packing of shorter numbers

x-fast trie, Willard 1983

Each element of S a binary string of length w

Store all prefixes of all elements of S in a hash table

Overall $O(nw)$ words of space

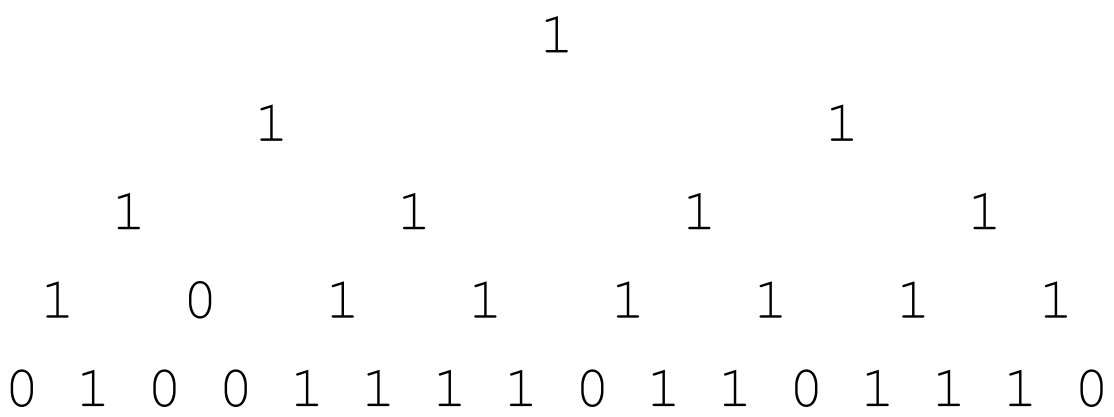
For each prefix store minimum and maximum element with this prefix

For each element of the set store its predecessor and successor

Example of x-fast trie

$w = 4, \{1, 4, 5, 6, 7, 9, 10, 12, 13, 14\}$

View as trie:



Prefixes:

$\epsilon, 0, 1, 00, 01, 10, 11, 000, 010, 011, 100, 101, 110, 111,$
$0001, 0100, 0101, 0110, 0111, 1001, 1010, 1100, 1101, 1110$

Predecessor in x-fast trie

```
1  Predecessor(H, x) {
2    if(x in H) { return H[x].predecessor }
3    y = longest prefix of x in H   // by binary search
4    if(x == y1u) { return H[y].max }
5    else return H[H[y].min].predecessor
6  }
```

Time $O(\log w)$ with perfect hashing

Insert to x-fast trie

- Need to insert/update all $w + 1$ prefixes of x
- For each prefix update min and max
- Update predecessor and successor for x and its neighbors
- $O(w)$ expected time with perfect hashing

y-fast trie, Willard 1983

Disadvantages of x-fast trie:

space $O(nw)$ rather than $O(n)$

insert time $O(w)$ rather than $O(\log w)$

Improve by bucketing:

Maintain buckets, each containing $\Theta(w)$ successive elements of the set

Within bucket store data in a balanced BST

Minimum of each bucket inserted to a x-fast trie or improved vEB

Size of this structure is $O(nw/w) = O(n)$

Combined sizes of trees $O(n)$

y-fast trie operations

Maintain buckets, each containing $\Theta(w)$ successive elements of the set

Within bucket store data in a balanced BST

Minimum of each bucket inserted to a x-fast trie or improved vEB

Predecessor query:

find correct bucket, search in BST, both $O(\log w)$

Insert:

find correct bucket, if enough space, add in $O(\log w)$,

otherwise split bucket into two, insert another element to main structure

Second case costs $O(w)$ but happens infrequently, amortize

Pointer machine model of computation

- No arrays, no pointer arithmetic
- Constant-size nodes, holding data (numbers, characters, etc) and pointers to other nodes
- Pointers can be assigned:
  – address of a newly created node
  – copy of another pointer
  – null
- Root node with all global variables
  all variables of the form root.current.left.x

Persistent data structures

- partial persistence: update only current version, query any old version, linearly ordered versions
- full persistence: update any version, versions form a tree
- confluent persistence: combine versions together, versions form a DAG
- fully functional: never modify nodes, only create new
- backtracking: query and update only current version, revert to an older version
- retroactive: insert updates to the past, delete past updates, query at any past time relative to current set of updates

General transformation for partial persistency

Arbitrary data structure

with $f(n)$ update, $g(n)$ query, $O(n)$ space

Partially persistent version

$O(n/k + f(n))$ update, $O(kf(n) + g(n))$ query

Store whole structure after k updates

Simulate updates from nearest checkpoint

Often balanced at $k \approx \sqrt{n}$

General transformation with fat nodes

Assumes pointer machine model, adds $O(\log n)$ factor overhead

Fat node: binary search tree with time as keys

Each BST node holds a node of the original d.s.

Query of original node at time t: predecessor search in BST

Update of original node: insert a new maximum to BST

Some BSTs allow $O(1)$ maximum inserts and maximum queries

General transformation with fat nodes

Arbitrary pointer machine data structure

with $f(n)$ update, $g(n)$ query

Partially persistent version

$O(f(n))$ update, $O(g(n))$ current query, $O(g(n) \log n)$ past query

General transformation for backtracking

Arbitrary pointer machine data structure

with $f(n)$ update, $g(n)$ query

Version with backtracking

$O(f(n))$ update, $O(g(n))$ query, $O(1)$ amortized backtrack

Use fat nodes with stacks instead of BST

Update adds a new version on the stack, prepays its removal

Query accesses top of the stack

Backtrack pops all stacks until it finds correct time stamp

Each item popped only once, prepaid

What about stacks that do not need change?

General transformation with node copying

Assumes pointer machine and

each node of original structure has **in-degree** $O(1)$

True e.g. for BSTs, but not for union-find

Better partial persisence: remove $O(\log n)$ overhead

Main idea:

avoid searching for time t separately in each fat node

link together versions belonging to the same time

but not all time stamps present in each node

General transformation with node copying

Arbitrary pointer machine data structure

with at most $p = O(1)$ incoming pointers per node

and $f(n)$ update, $g(n)$ query

Partially persistent version

$O(f(n))$ amortized update, $O(g(n))$ query

General transformation with node copying

New node:

– original node

– p reverse pointers for current version only

– $2p$ mods (version, field, value)

multiple such nodes for an original node

Version:

– time t and original root node at time t

– array of roots (not pointer machine) or BST for roots or user supplies root

Read node at time t:

apply all mods with version$< t$

$O(1)$ overhead

General transformation with node copying

Update node, i.e. change $n.x$ from z to y

If node not full, add a new mod

Otherwise add a new node n' with latest version of n

Other nodes may have back pointers to n, change to n'

– to do so, follow forward pointers from n'

Recursively change pointers to n to point to n' in the newest version

– keep pointer to n in the old version

– found using back pointers

Add back pointer from y to n'

Remove back pointer from z to n

Φ: total number of mods in the latest versions of nodes

Retroactive data structures

- Insert(t,updateOp(args))
- Delete(t)
- Query(t,queryOp(args))

  partially retroactive allows query only at present

  fully retroactive query at any time

Commutative and invertible updates

Easy case:

- Order of updates can be permuted
- Each update has an inverse operation
- e.g. search: maintain a set under insert, delete, find (hashing)
- e.g. maintain array, update $A[i] += x$, query $A[i]$
- but not heap insert, delete_min
- Partial retroactivity simply applies updates in the present

Decomposable search problems

Search problem:

maintain a set S with insert and delete

support query(x, S)

Decomposable search problem

query$(x, A \cup B) =$ query$(x, A) \square$ query(x, B)

– operation $\square$ computable in $O(1)$

– possibly require that A and B are disjoint

Examples:

– exact set membership, nearest neighbor, predecessor

– for disjoint sets also rank

Full retroactivity for decomposable search problems

- Item a present in set S during interval (b_a, e_a)

  assume each item inserted only once

- Binary search tree, keys are items, values (b_a, e_a)

- Segment tree, leaves are times when operation inserted

  – each node: data str. for decomposable search for a subset of S

  – element a in nodes in canonical decomposition of (b_a, e_a)

- Retroactive update:

  – find interval (b_a, e_a) in BST and update to (b'_a, e'_a)

  – delete interval (b_a, e_a) from segment tree

  – insert new interval (b'_a, e'_a) to segment tree

  – segment tree with leaf insert/deletes, needs rebalancing (how?)

Full retroactivity for decomposable search problems

- Query at time t:
  - find a leaf for predecessor of t
  - search in each node on the path to root
  - combine results using $\square$
- Overall $\log n$ factor overhead for each update and query as well as for space

Arbitrary data structure

$f(n)$ update, $g(n)$ query

Totally retroactive version

$O(f(n) \log n)$ amortized update, $O(g(n) \log n)$ query

Review: Scapegoat trees

- Lazy amortized binary search trees
- Do not require balancing information stored in nodes
- Insert and delete $O(\log n)$ amortized
  search $O(\log n)$ worst-case
- Invariant: keep the height of the tree at most $\log_{3/2} n$
- When invariant not satisfied, completely rebuild a subtree

Partially persistent data structures

Update only current version, query any old version

versions linearly ordered

Arbitrary pointer machine data structure

with at most $O(1)$ incoming pointers per node

and $f(n)$ update, $g(n)$ query

Partially persistent version with node copying

$O(f(n))$ amortized update, $O(g(n))$ query

Retroactive data structures

Insert updates to the past, delete past updates, query at any past time relative to the current set of updates

Search problem:

maintain a set S with insert and delete

support query(x, S)

Decomposable search problem

query$(x, A \cup B) =$ query$(x, A) \square$ query(x, B)

Arbitrary data structure

$f(n)$ update, $g(n)$ query

Totally retroactive version

$O(f(n) \log n)$ amortized update, $O(g(n) \log n)$ query

Planar point location

- Plane subdivided into regions by non-intersecting straight lines (planar graph)
- Given point (x, y), which face contains it?
- Examples: regions in a map, GUI elements, ...
- Static version: preprocess a fixed graph
- Dynamic version: edge added/removed

Vertical ray shooting

- Given set of non-intersecting line segments
- Query: which edge first intersects a vertical ray starting in (x, y)?
- In static case implies planar point location

  (each edge keeps face ID)
- First assume that all line segments horizontal

Vertical ray shooting for horizontal line segments (static)

- Sweep with partially persistent balanced BST

  Left segment endpoint (x_1, y): insert y at time x_1

  Right segment endpoint (x_2, y): delete y at time x_2

- $O(n \log n)$ time preprocessing

- Given ray from (x, y), search for successor of y at time x

- $O(\log n)$ query

Vertical ray shooting for horizontal line segments (dynamic)

- Use retroactive binary search tree
- $O(\log^2 n)$ queries last time, $O(\log n)$ version also exists
- Insert line segment $(x_1, y), (x_2, y)$:

  Insert(x_1,insert(y))

  Insert(x_2,delete(y))
- Delete line segment $(x_1, y), (x_2, y)$:

  Delete(x_1,insert(y))

  Delete(x_2,delete(y))

Vertical ray shooting with arbitrary segments

- Segments do not cross, but any direction
- Static version still with partially persistent BST
- When searching successor of y at time x,
  use comparison function which depends on x
- Dynamic version does not work in $O(\log n)$

Also interesting is ray shooting in arbitrary direction

– no poly-log algorithms known

– motivated by ray tracing

Orthogonal range searching

- Maintain a set of points in R^d
- Query: find points in box $[a_1, b_1] \times \cdots \times [a_d, b_d]$
  existence / count / report k
- Static / dynamic case
- E.g. database queries combining d columns
- Also related to nearest neighbour
- Range trees $O(\lg^d n + k)$ query
- Layered range trees $O(\lg^{d-1} n + k)$ query
- Updates in range trees $O(\log^d n)$ amortized
- Further improvements exist

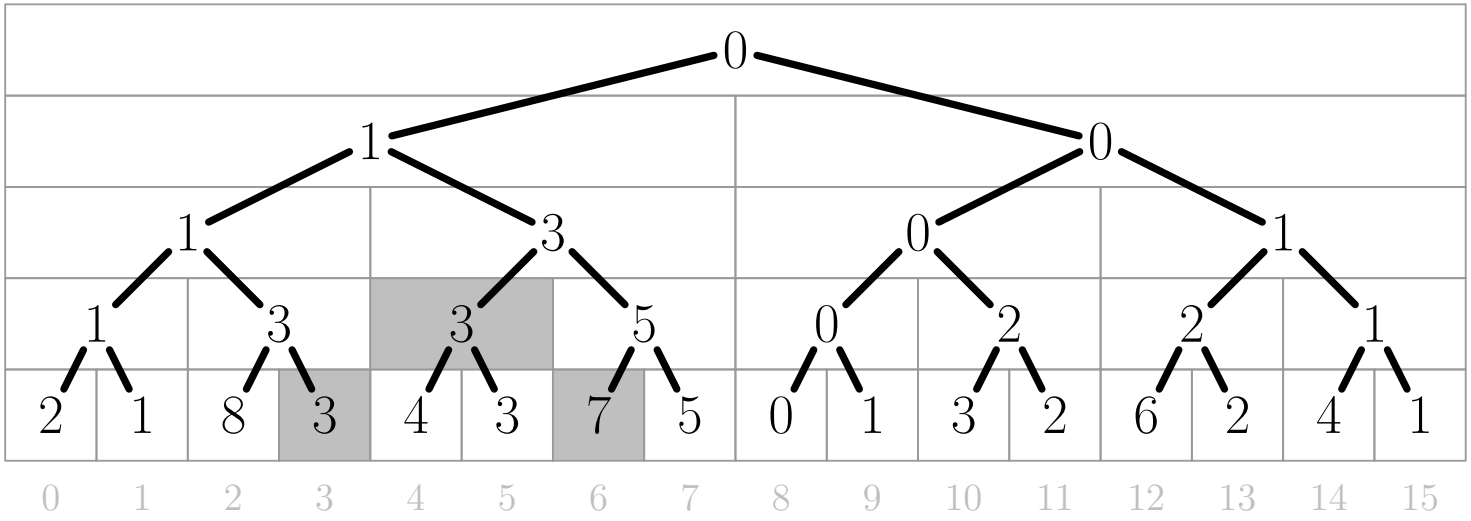
Range trees: 1D case

Report/count points in an interval $[a, b]$

- Balanced binary search tree/segment tree
- Points in the leaves
- Internal nodes store maximum in the left subtree and subtree size for fast counting
- Find predecessor of a, successor of b
- Leaves between form the answer
  $O(\log n)$ subtrees (canonical decomposition)

Canonical decomposition

Decompose query interval $[x, y)$ to a set of disjoint tree intervals



Canonical decomposition

Decompose query interval $[x, y)$ to a set of disjoint tree intervals

- Current node $[i, j)$, and its left child $[i, k)$
  invariant: $[i, j)$ overlaps with $[x, y)$
- If $[i, j) \subseteq [x, y)$, return $\{[i, j)\}$
- $R = \emptyset$
- If $[i, k)$ overlaps with $[x, y)$, recurse on left child, add to R
- If $[k, j)$ overlaps with $[x, y)$, recurse on right child, add to R
- Return R

Range trees: 2D case

- Build BST for x-coordinate
- Consider internal node v
- Build BST tree for subtree rooted at v in y-coordinate
- Each point in $O(\log n)$ y-coord trees
- Search for $[a_1, b_1] \times [a_2, b_2]$:
  – find $O(\log n)$ subtrees for $[a_1, b_1]$ according to x
  – search in each according to y

d dimensions:

- Every node in dimension i has a range tree for remaining dimensions
- Query $O(\log^d n)$, space and preprocessing $O(n \log^{d-1} n)$

Layered range trees: $O(\log^{d-1} n)$ for $d \geq 2$

a.k.a fractional cascading

- Replace y BSTs by sorted arrays
- Root of x BSTs has all points sorted by y in array
- Array in a child a subset of parent's

  link from parent array to successors in child array
- At the root find $[a_2, b_2]$ in the array
- Follow array links as traversing the tree
- In higher dimensions use this at the last dimension
- Saves $\log n$ factor

Dynamic range trees: outline

- Use scapegoat trees
- Rebalancing: rebuild an entire subtree
- If rebuild linear, $O(\log n)$ amortized updates
  – pay $O(1)$ on each level towards future rebuilds
  – linearly many updates between 2 rebuilds of the same node
- If rebuild $O(n \log n)$, $O(\log^2 n)$ amortized updates
  – pay $O(\log n)$ on each level towards future rebuilds
- In layered range trees: $O(\log^d n)$ amortized update

Wavelet tree (Grossi, Gupta, Vitter 2003)

$\Sigma_0 = \{\$, ., a\}$ $\Sigma_1 = \{e, m, u\}$

```
i    0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22
S[i] e  m  a  .  m  a  .  m  a  m  u  .  m  a  m  a  .  m  a  .  e  m  u
B[i] 1  1  0  0  1  0  0  1  0  1  1  0  1  0  1  0  0  1  0  0  1  1  1
S0   a.a.a.aa.a.$      S1   emmmmummmemu
```

$\Sigma_{00} = \{\$\}, \Sigma_{01} = \{., a\}$, $\Sigma_{010} = \{.\}, \Sigma_{011} = \{a\}$
$\Sigma_{10} = \{e\}, \Sigma_{11} = \{m, u\}$, $\Sigma_{110} = \{m\}, \Sigma_{111} = \{u\}$

```
i     0  1  2  3  4  5  6  7  8  9 10 11
S0[i] a  .  a  .  a  .  a  a  .  a  .  $
B0[i] 1  1  1  1  1  1  1  1  1  1  1  0
S00  a.a.a.aa.a.  S01 $
```

Store

B[i] =110010010110101001001110

B0[i] = 111111111110 B1[i] = 011111111011

B01[i] = 10101011010 B11[i] = 000100001

2D range searching via wavelet trees

- Points $(x_0, y_0), \ldots, (x_{n-1}, y_{n-1})$ s.t. $y_i < y_{i+1}$
- Represent as "string" $T = x_0, \ldots, x_{n-1}$
- Build a wavelet tree
- Similar to BST/segment tree for x-coordinate: each node an interval
- Canonical decomposition of $[a_1, b_1]$ in $O(\log n)$ intervals
- Find substring corresponding to $[a_2, b_2]$ in T
- Track down to canonical decomposition intervals
- Good for counting points in $O(\log n)$, small memory
- Reporting points takes $O(\log n)$ per point, can be improved

Exercise

- Consider a static set of points in 2D, each with a cost

  (for example hotels...)

- Find the lowest-cost point in a given rectangle

- How to add to layered range trees and to wavelet trees?

Exercise

- Preprocess text T (e.g. to suffix array plus other structures)
- Query: (P, i, j): find/count occurrences of P in $T[i..j]$
- How can we use range searching for this?