Algorithms and Data Structures 1

TIN060

Jan Hric

Lectures, part 2, v. 18.5.2015_c

Dynamic sets

- Data structures for storing some data
- Dynamic structure: changes in time
- An element of a dynamic d.s. is accesible through a pointer and has
  1. A key, usually from some (lineary) ordered set
  2. Pointer(s) to other elements, or parts of d.s.
  3. Other (user) data (!)

Operations

- S is a dynamic set, k is a key, x is a pointer to an element
- Operations
  - **Find(S,k)** – it returns a pointer to an element with the key k (or NIL)
  - **Insert(S,x)** – it inserts an element x into S
  - **Delete(S, x)** – it deletes an element x from S
  - **Min(S)** – it returns a pointer to an element with the minimal key in S
  - **Succ(S,x)** – it returns a pointer to the element next to x (wrt. linear ordering)
  - **Max(S)**, **Predec(S,x)** – analogy to Min, Succ

Binary search trees

- Dynamic d.s. which supports all operations
- A binary tree: each node has 3 pointers:
  - Left child (left)
  - Right child (right)
  - Parent (par)
- A binary search tree (BST) invariant: for each node x: each node in the left subtree of x has a smaller (or equal) key than x, and each node in the right subtree of x has a greater key than x

Operations

- Find(x,k) ; x is a pointer to the root of the tree

```
while (x<> NIL) and (k<>key(x)) do
   if k=<key(x) then x:=left(x)
                else x:=right(x)
return(x)
```

- Time complexity is O(h), where h is the height of the tree
- Min, Succ ...

Operations

- Min(x) ; x is a pointer to the root of the tree

```
while (left(x) <> NIL) do
   x:=left(x)
return(x)
```

- Max(x) is symmetrical to the right
- Time complexity: O(h)

Operations

- Succ(x) ; x is a local pointer (we don't need the Root)

```
if (right(x) <> NIL)
   then return Min(right(x))
   else  ; x doesn't have a right child
      y:= par(x)  ;  go up to the left ancestor
      while y<>NIL and x=right(y) do
         x:=y
         y:=par(y)
      return(y)
```

Modification operations: Insert

- insert(x,z) ; x is a pointer to the root of the tree, z to the new element

```
y:= NIL ; w:=x    ; we suppose left(z)=right(z)=NIL
while (w<>NIL) do ; going down through the tree, with 2 pointers
  y:=w     ; invariant: y=par(w), if w=NIL, insert z under y
  if key(z) =< key(w)
    then w:=left(w)
    else w:=right(w)
par(z):=y
if y<>NIL then
  if key(z) =< key(y)
    then left(y):=z
    else right(y):=z
 else x:=z  ; z is a new root, the tree was empty
```

Operation: delete

- delete(x,z) ; x is a pointer to the root of the tree, z to the being deleted element

```
if left(z)=NIL or right(z)=NIL
  then y:=z  ;  going down through the tree, with 2 pointers
  else y:=Succ(z)  ; y points to the be-deleted node
if left(y)<>NIL
    then w:=left(y)  ; w points to the only child of y or NIL
    else w:=right(y)
if w<>NIL then par(w):=par(y)  ; fixing the parent of w
if parent(y)=NIL
  then x:=w   ;  a new root
  else if y=left(par(y))  ; fixing the left/right down-pointer
         then left(par(y)):=w
         else right(par(y)):=w
if y<>z then key(z):=key(y)  ; moving information
```

Operation: Delete 2

- delete(x,z) ;
- 2 cases: 0-1 child or 2 children
  - 0-1 child: we can delete in place
  - 2 children: we must delete (from the place of) Successor
- Complexity: O(h)
- In case of pointers to the elements from outside of our data structure: repointering of Succ(z)
  - We want to leave a physical copy of Succ(z) - if it exists

Balanced binary trees

- Time complexity of BST: O(h)
  - On each level: Θ(1)
  - For „plain“ BST with n elements:
    - An average case: O(log n)
    - The worst case: O(n)   :-(
- Goal: each operation should be in O(log n) in the worst case
  - By using special *local* transformations
    - Local and global invariants stay valid
  - Local information is added
- Approaches: Red-Black trees, AVL trees

Red-Black trees

- Each node has a colour: red or black
  - Implementation: 1 bit
- Red-Black tree fulfills 4 conditions:
  1. Each node is either red or black
  2. Each leaf (NIL, an external leaf) is black
  3. If a node is red, both its children are black
  4. Each path from a node to any child has the same number of black nodes, so called *black height (bh)*
- From 3., no two red nodes are neighbours on a path → the worst ratio of a path length is 1:2
  - Only black nodes vs. Alternating black and red nodes

Height of R-B trees

- Lemma: R-B tree with n internal nodes has a height h at most 2.log(n+1)
- Proof: by induction on the height

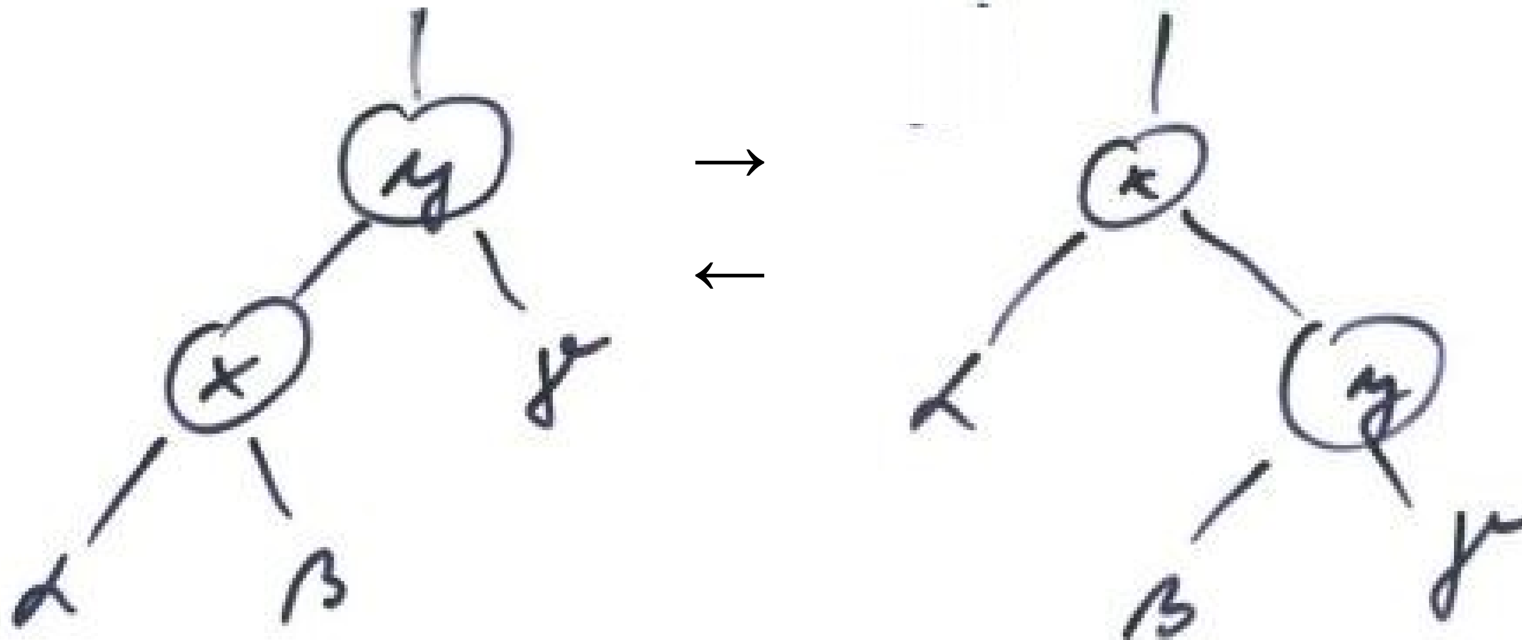
A subtree rooted in x has at least $2^{bh(x)}-1$ internal nodes, by an induction

Apply to the root: $n \geq 2^{h/2}-1$, thus $h \leq 2\log(n+1)$

- Corollary: #operations is O(log(n)), supposing O(1) complexity for each layer

Rotations

- → Right rotation ; ← left rotation
  - A local change, the same ordering
  - Changed edges in a right rot.: parent-y, y-x, x-β
    - We must update pointers in both directions

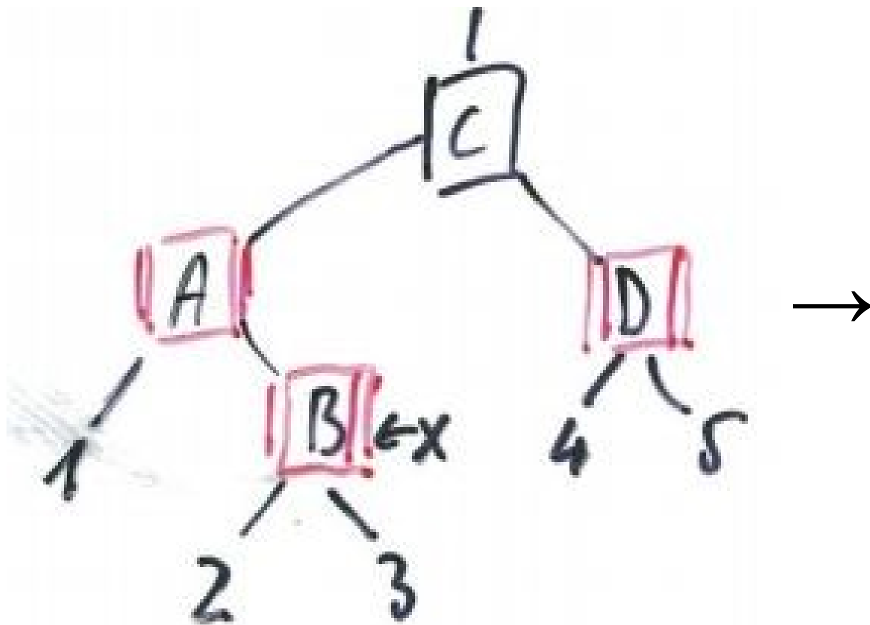


Insert

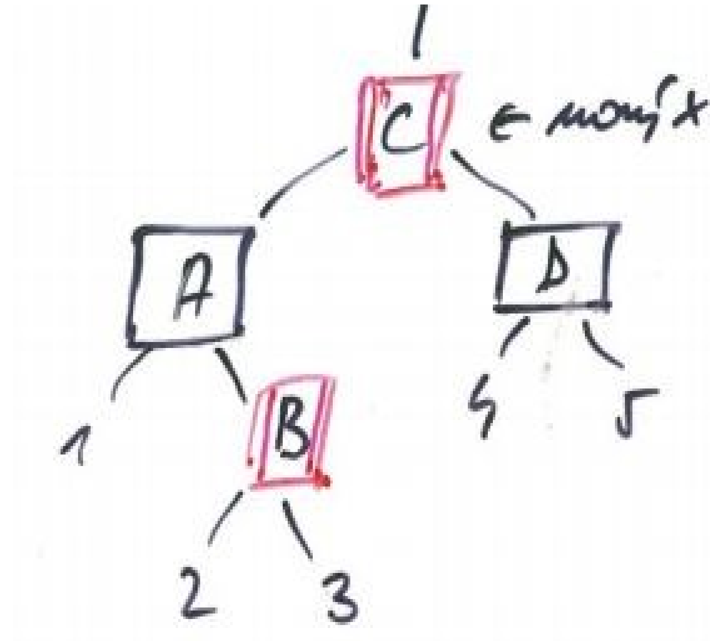
- We can recolour a red root to a black root without a violation of invariants → it is the only way to increase the black height of a tree
  - We maintain the black root as an additional invariant
- We insert a node X to a tree as a leaf of BST and colour it red
  - A possible defect is two red nodes on a path
- Analysis of Insert:
  1. A black node above two red nodes

     → a defect must be propagated up, no reserve
  2. A black node above a black node(s) forms a reserve, we can insert locally
- Analysis of a defect: the uncle of X is 1. red or 2. black
  - Or no defect → the tree is valid

- The uncle of X (D in pictures) is red
- Recolour, propagate, C is a new X

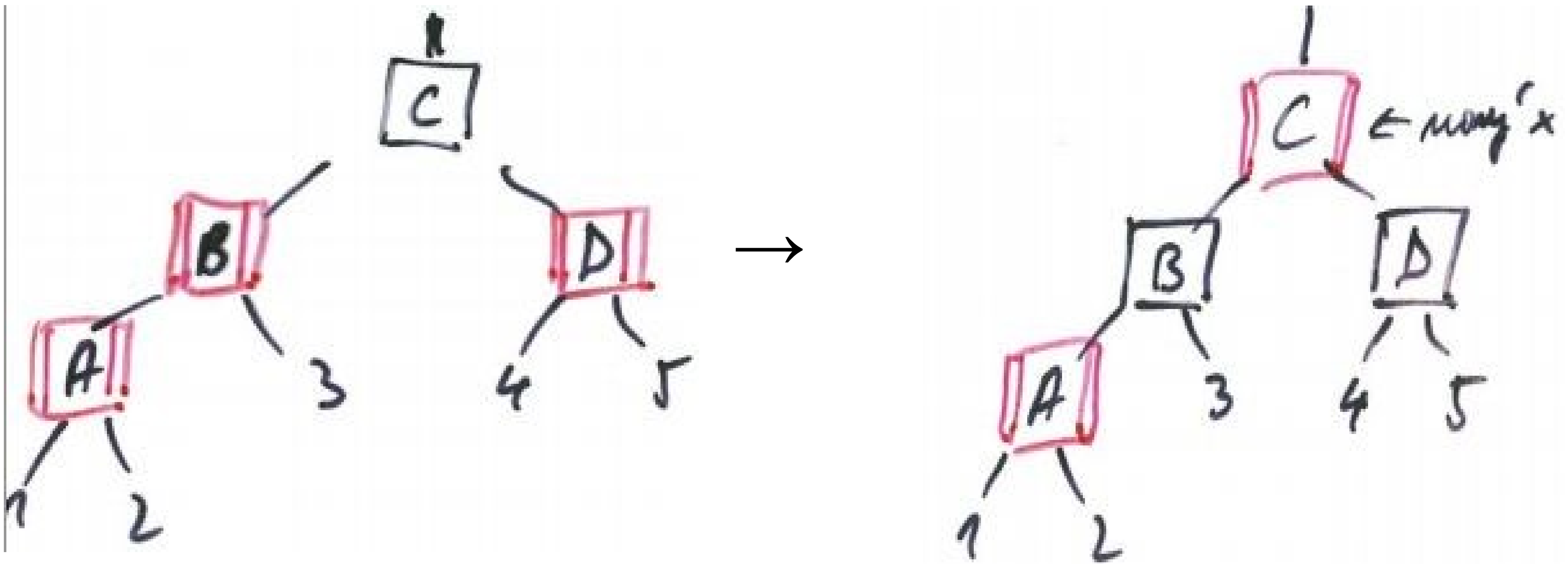


→



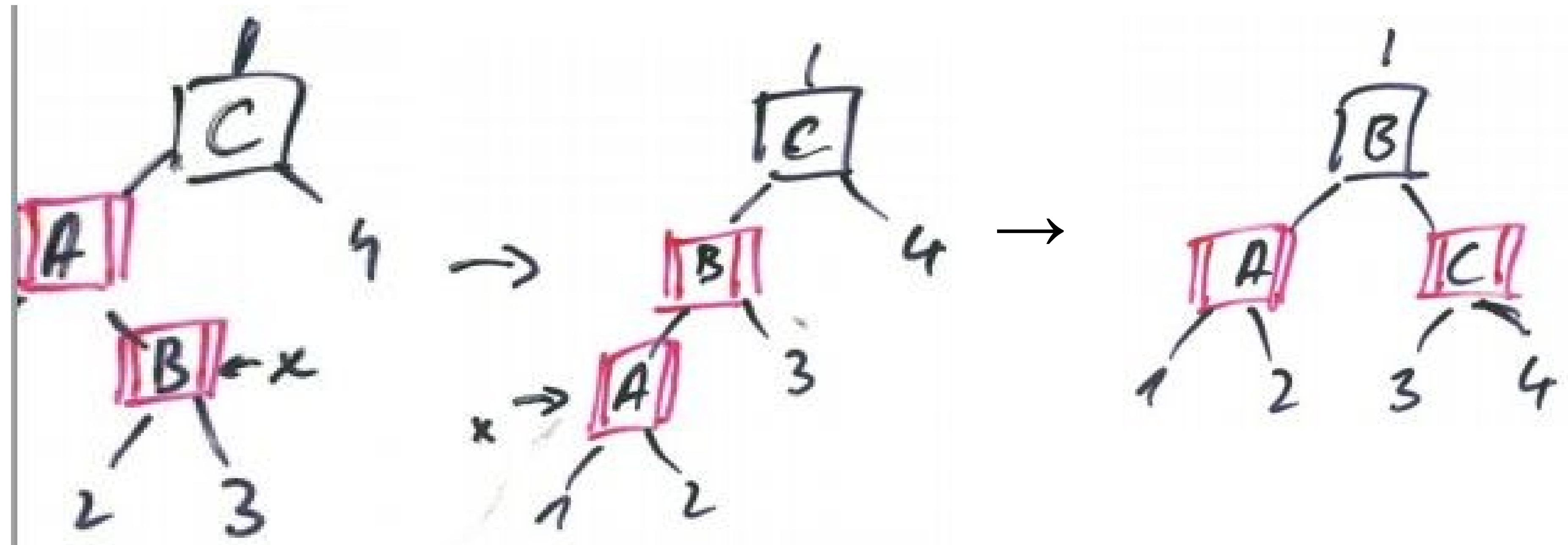
- The parent of C can be red → the only possible defect
  - An ordering is valid, 1-5 are black (are under red nodes)
  - A local black height of 1-5 does not change

- The uncle of X is red, X is an outer node
- Recolour, propagate, C is a new X



- In all cases: 1-5 can be NIL

- The uncle of X (4 in pictures) is black
  - A local elimination of the defect, no propagation



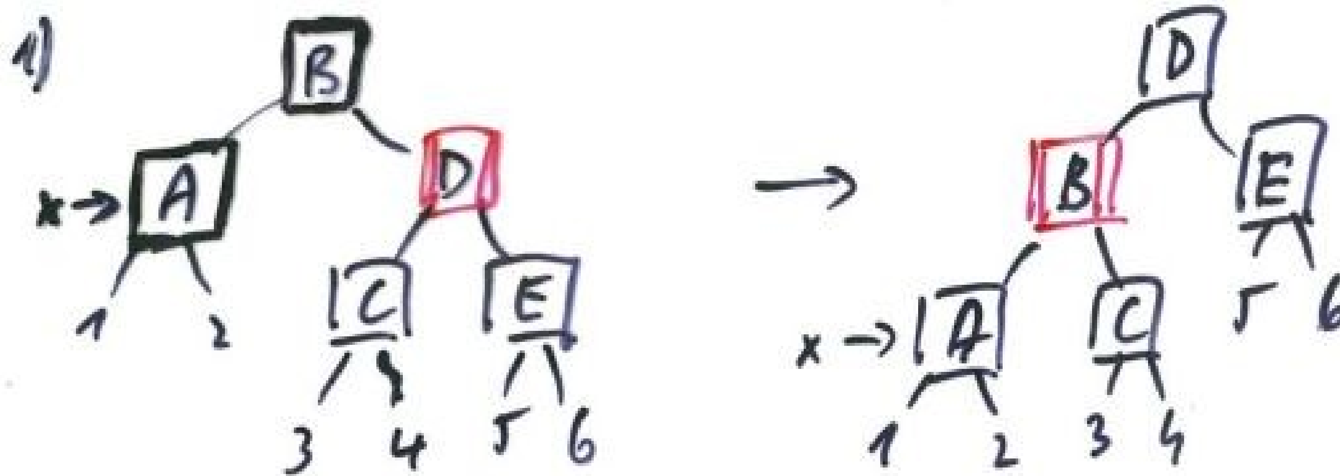
- Edge C-4 is correct, as (the root of) 4 is black

Delete(T,x)

- We delete a leaf, possibly after relinking the successor of x
  - A new defect, if any: a node is *double black*
  - The deleted node has
    1. 0 children → an external node (NIL), double black
    2. 1 child (red) → an internal node, black
    3. 2 children, only during propagation → use transformations
- Transformations: by analysis of cases
  - Only O(1) time on each level

Delete 1

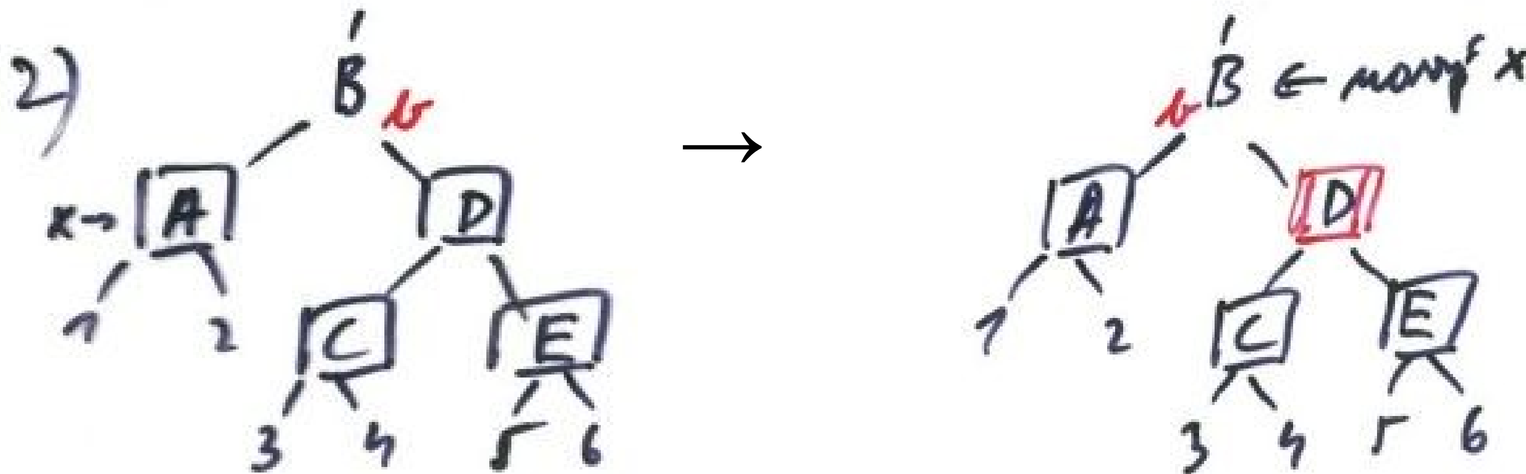
- The parent of X is black and the brother is red
  - Restructuring → The brother is black, continue



- Check an ordering of a tree and a validity of edges

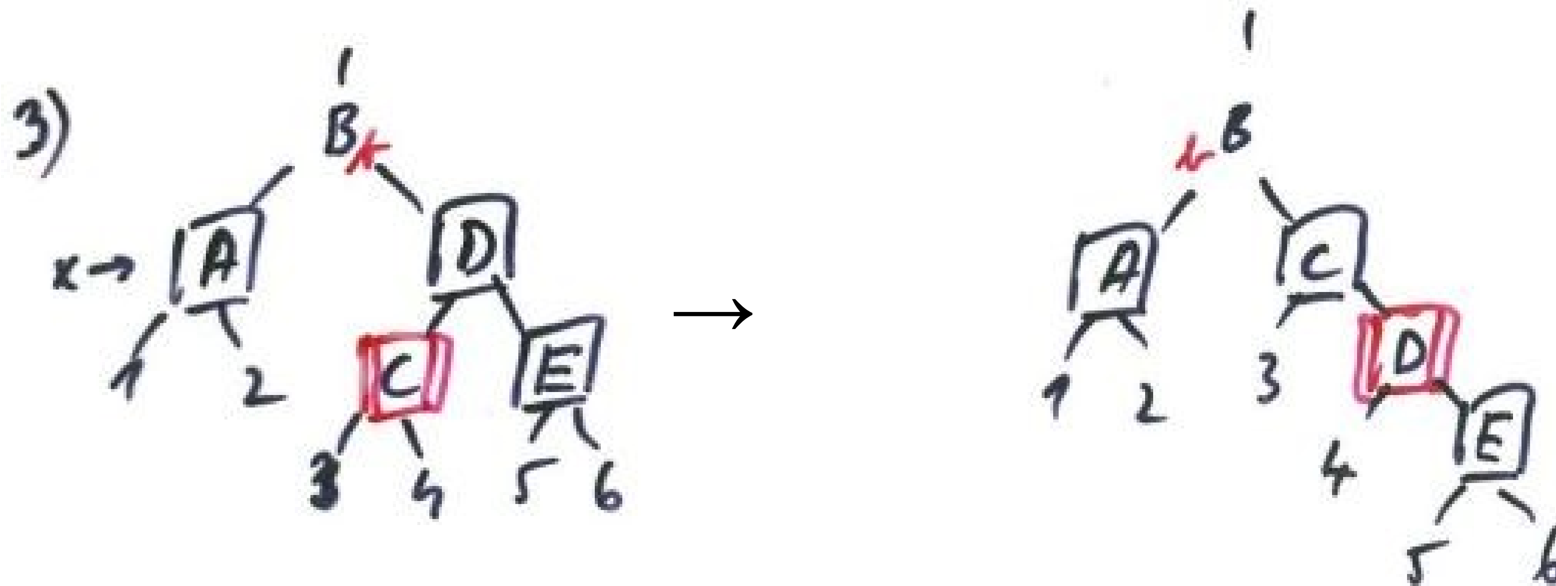
Delete 2

- The brother of X is black and has black children (and a colour of the parent B does not matter)
  - Elimination of a defect (if B was red) or
  - Propagation (if B was black) and B is the new X



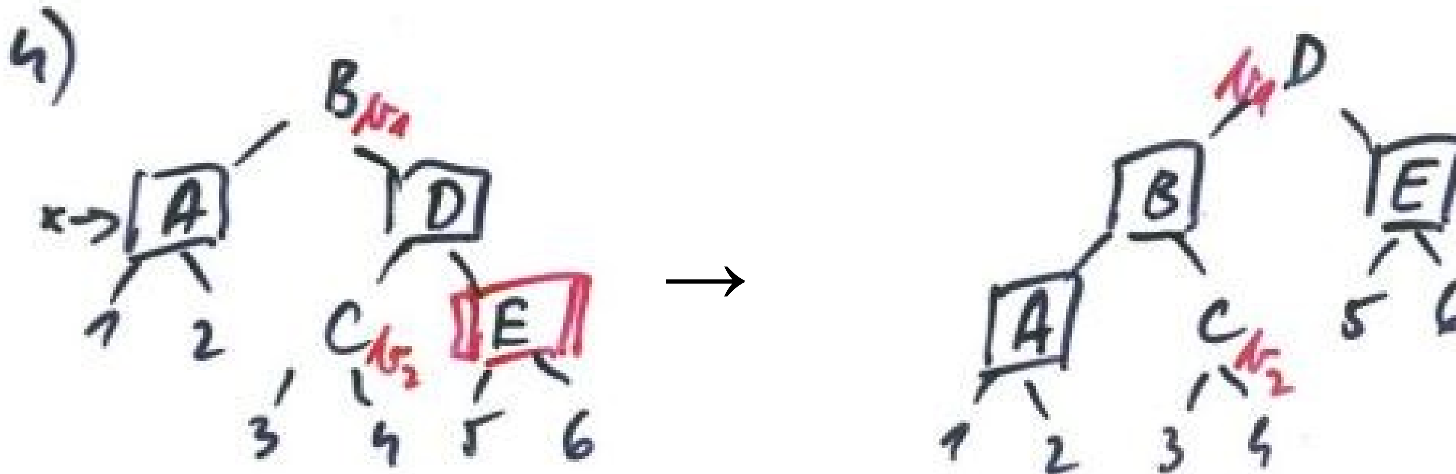
Delete 3

- The brother of X is black and has an outer black child (and an inner red one)
  - Restructuring → continue by 4



Delete 4

- The brother of X is black and has an the outer red child (and a colour of other child does not matter)
  - Local elimination of the defect, we had a reserve



- … and symmetrical cases

AVL trees

- Def: (Adelson-Velskij, Landis) A BST is an AVL tree (AVL balanced) iff for each node x holds:

  |h(left(x)) – h(right(x))| <= 1 ,

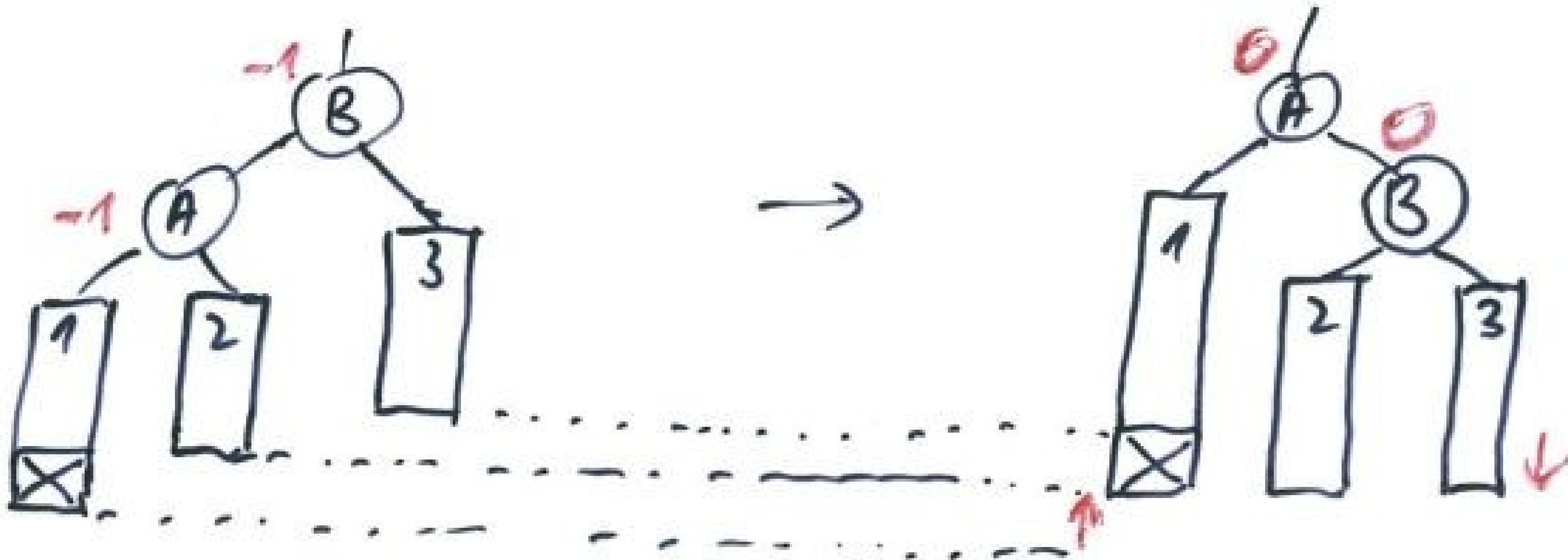
  where h(x) is the height of a tree
- We remember an actual balancing (from {-1, 0, +1}) for an efficiency of operations; (-1 ≈ left is deeper)
- Theorem: The height of AVL tree with n nodes is O(log n)
  - By an induction, we construct a tree having a height h with least nodes
  - $c_h = c_{h-1} + c_{h-2} + 1 = fibbonaci_{h+3} - 1 \, (for\ fib_3 = 2)$

Operations on AVL trees

- Corollary: Nonmodifying operations: in O(log n)
- Modifying operations: Insert, Delete
  - As in BST, but a propagation of a change bottom-up, if needed
  - Locally: fulfill an invariant using rotations (and propagate)
- Properties of rotations:
  - Ordering of keys and subtrees is preserved
  - Height is preserved or propagate a change
  - Double rotations can be implemented by two simple rotations, but for invariant proofs we take it as a single operation.

Insert 1

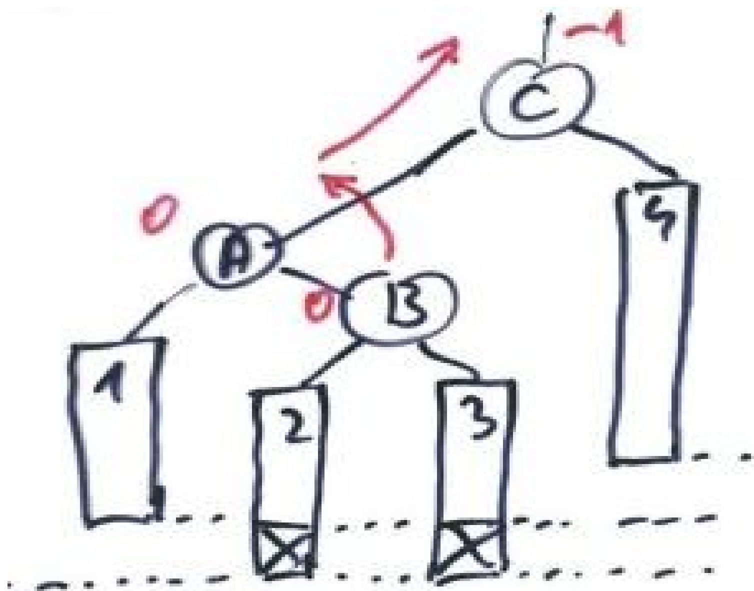
- Check and update balancing bottom up, if needed
- A single rotation: Insert X to an outer subtree

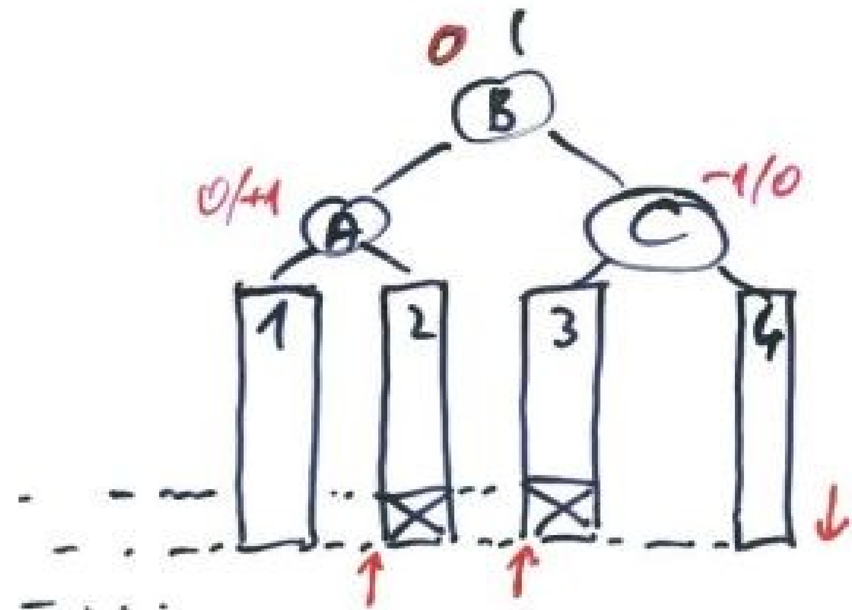


- Local and global heights are the same → no propagation needed

Insert 2

- Double rotation: Insert one of Xs to an inner subtree

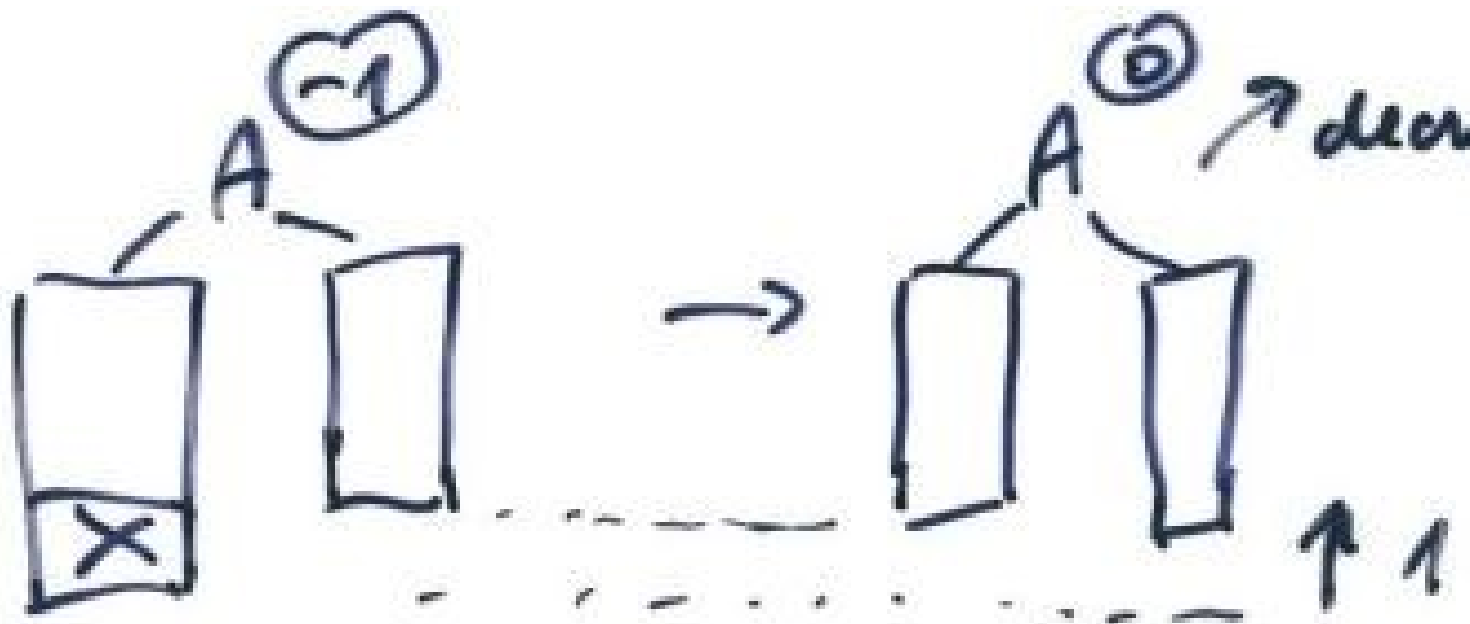




- Local and global heights are the same → no propagation needed

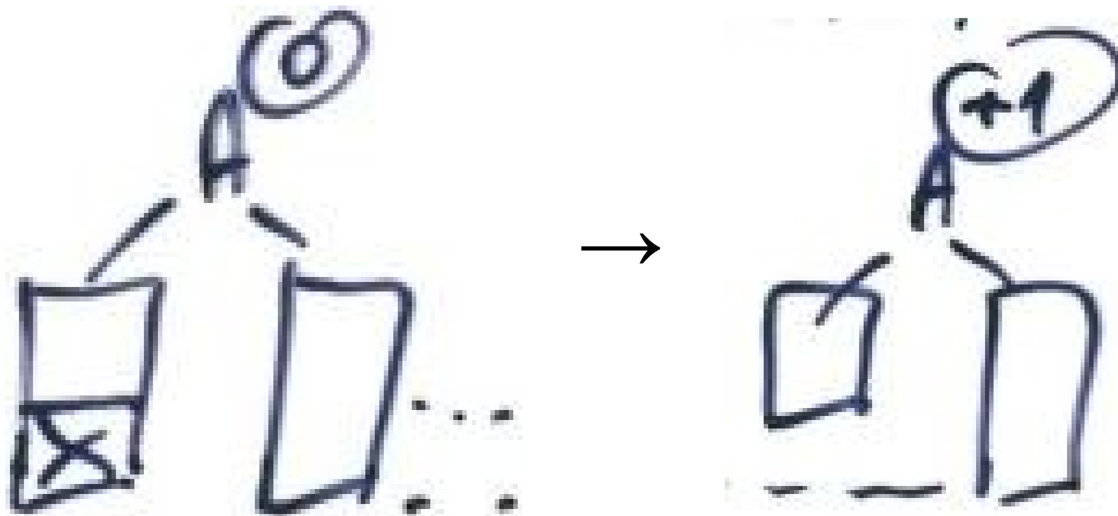
Delete

- Delete X, in the left child of A
- Balancing -1: h(left) > h(right)
  - Propagate a decrement up



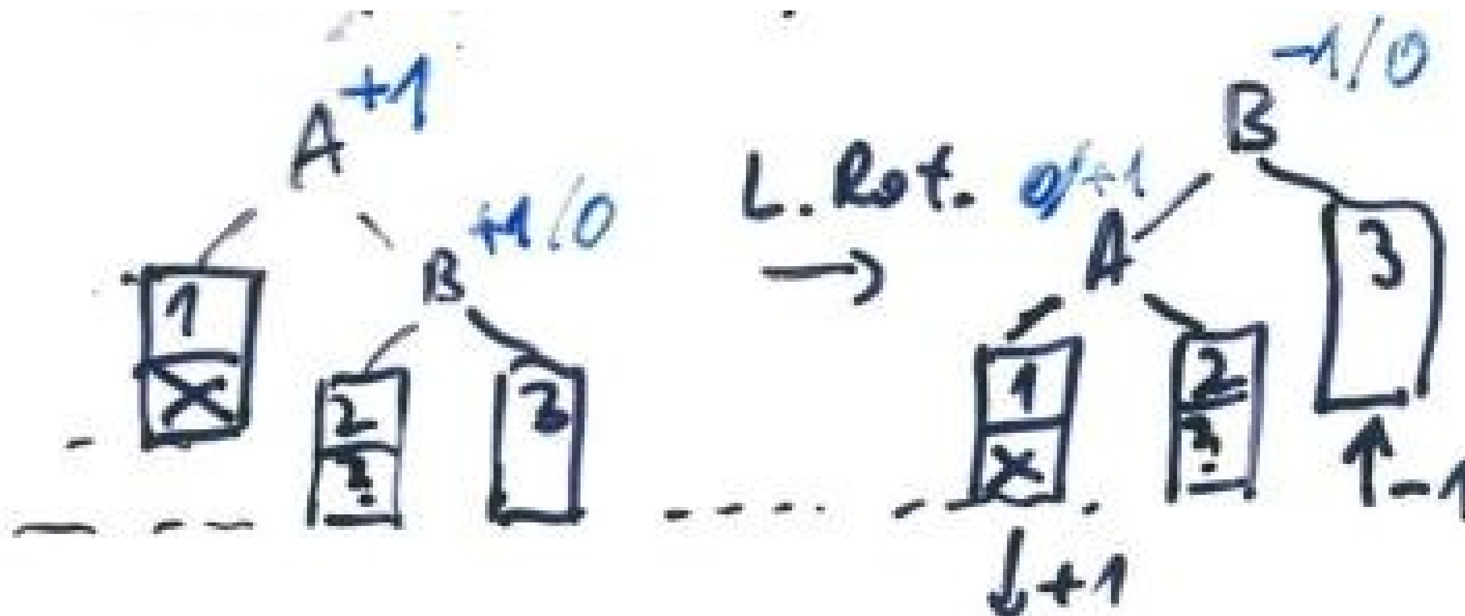
Delete 2

- Delete X, in the left child of A
- Balancing 0: h(left) = h(right)
  - Update balancing, no propagation



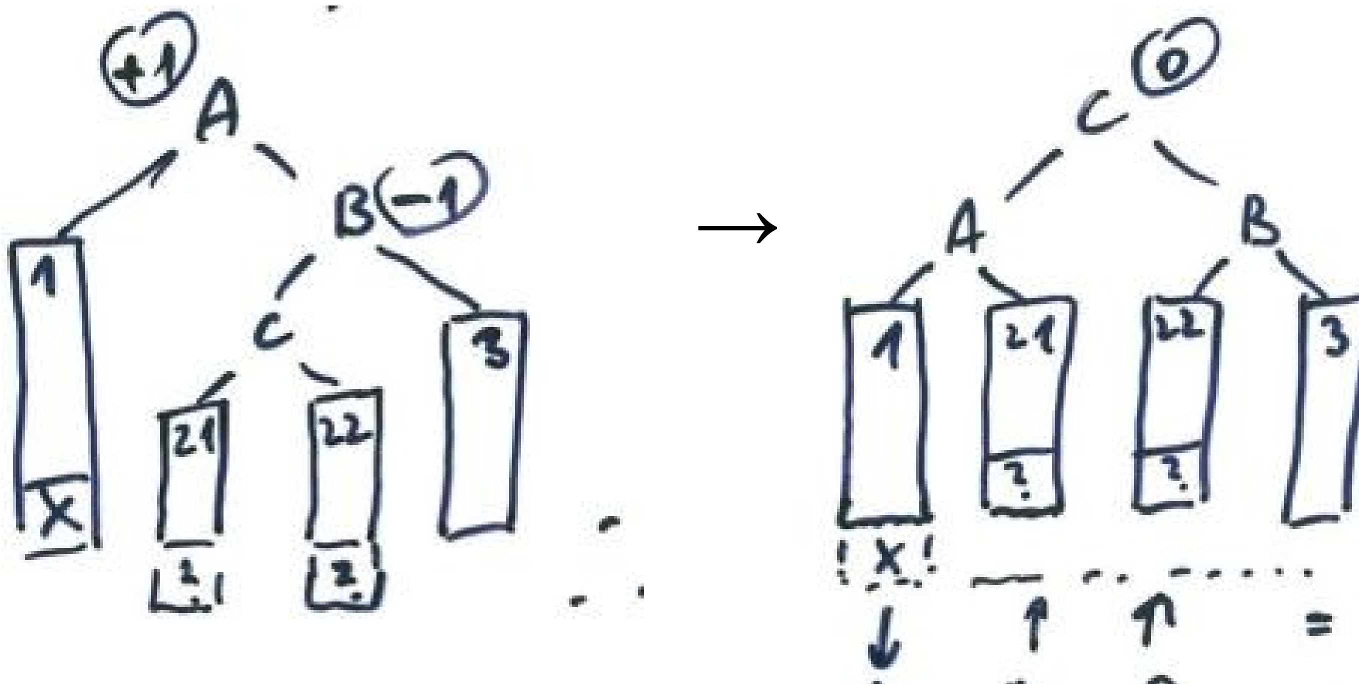
Delete 3a, 3b

- A single rotation, balancing +1 in A
  - Case 3a) h(2) < h(3): w/o „?“: propagate a decrement
  - Case 3b) h(2) = h(3): w/ „?“: without any propagation



Delete 3c

- A double rotation, +1 in A, -1 in B
  - Case 3c) h(C) > h(3), at least one „?“ exists
  - Propagate a decrement



Remarks

- Usual implementation of operations is by a recursion
  - But we can remember the path explicitely: „LLRL“
  - And propagate accoding to data without a recursion
- In some programming languages:
  - A representation of trees by terms: t(Left,X,Right)

```
Rot3c( t(T1,A,t(t(T21,C,T22),B,T3)),
       t(t(T1,A,T21),C,t(T22,B,T3)) ).
```

(B-trees)

- (Temporarily) skipped
  - B-trees are included also elsewhere (Data structures)
- Nonbinary trees
  - Used in database indices, have nodes at disk pages
  - In some sense: a generalization of R-B trees
    - Each leaf has the same depth
    - A black node with red nodes below ~ a node in B-tree
  - A node can have a variable number of keys and children
    - In a B-tree: between n/2 and n → a reserve in space allows spliting and joining nodes (at the same level)

B-trees

- A split of a vertex … (pictures)

Hashing

- Hash tables are suitable for representation of dynamic sets having only the operations Insert, Delete, and Find
  - A time complexity in an average case for 3 ops: $\Theta(1)$
  - Comparing to BST: no interval search (using Succ)
- An idea: a directly addressable table = an array
  - But: keys (=indices) must be different
  - A universum of keys is small
  - There are data or pointers to data in a table
    - Keys are stored explicitely or can be computed

Hashing

- If a universum of keys is big:

  → compute an index to the table from data

  - The hash function $h: U \to \{0..m-1\}$, usually $|U| >> m$
- A hash table size is proportional to a number of actually stored keys
- A problem: collisions
  - Two (or more) keys hash to the same index
  - Collisions are present, if $|U| > m$

Collision solving

- 2 basic types of methods
  1. A chaining of elements
  2. An open addressing
- Ad 1: Elements hashed to the same index are stored in a linked list
- Insert(x): Compute h(key(x)) and store x to the beginning of the relevant list – $\Theta(1)$
- Delete(x): $\Theta(1)$ if a linked list is bidirectional and we have a pointer to x, otherwise as Find(x)

Analysis

- Def: A *load factor* $\alpha = n/m$, m is a table size, n is a count of stored elements
  - A table with linked lists can have $\alpha > 1$
- Preconditions:
  - A value of a hash function is computed in $\Theta(1)$
  - A simple uniform hashing: each key is hashed to the m places with the same probability, independently of other keys
    - A birthday paradox: A probability that among 23 people some couple has the same birthday is above 50 %
- Find: successful and unsuccessful

Analysis

- Theorem 1: An *unsuccessful* search takes $\Theta(1+\alpha)$ in a hash table with linked lists, supposing a simple uniform hashing

  Proof: The key k is hashed to m slots with the same probability. An unsuccessful search passes through a list till its end. An average length of lists is α. An expected number of analysed elements is α and a total time is $\Theta(1+\alpha)$.

Analysis

- Theorem 2: A *successful* search takes $\Theta(1+\alpha)$ in a hash table with linked lists, supposing a simple uniform hashing

  Proof: Suppose that each stored element is searched with the same probability. Suppose new elements are stored at the end of lists. Expected number of processed elements is 1+number of elements in a list during an insertion of a searched element.

  The expected length of a list is $(i-1)/m$ during an insertion of the i-th element.

Analysis

$$C_{11} = M_1 \oplus M_2 \ominus M_4 \oplus M_6$$
$$C_{12} = M_4 \oplus M_5$$
$$C_{21} = M_6 \oplus M_7$$
$$C_{22} = M_2 \ominus M_3 \oplus M_5 \ominus M_7$$

- A conclusion: if n=O(m), then α=n/m=O(1)
- A note: inserting to the end vs. the beginning
  - Frequent keys vs. a locality principle

Hash functions

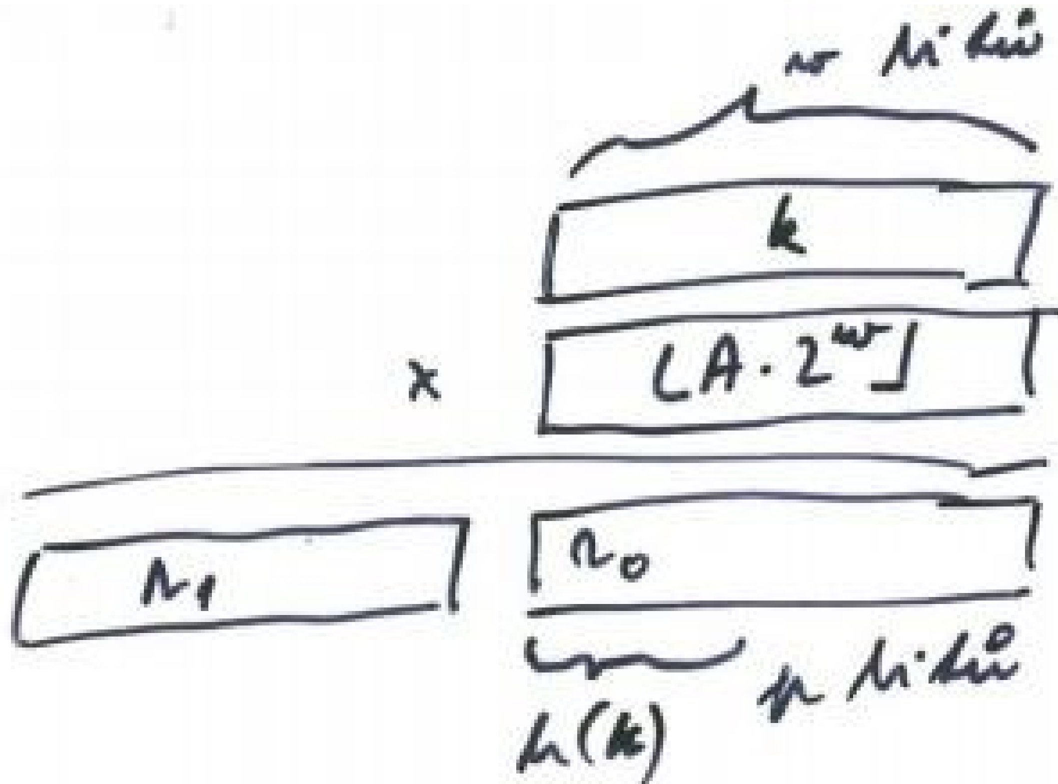
- Applications of hash functions (with different demands)
  1. A hash table
  2. A signature of data, (a fingerprint ...)
  3. In cryptography
- A construction of hash functions (for 1., 2.)
  1. Hashing by division
  2. Hashing by multiplication
  3. Universal hashing (later)
- Note: We aim at a hash function which distributes keys uniformly (and is quick)

Hash functions

- A precondition: keys are numbers, otherwise transform them to numbers
- Hashing by division
  - $h(k) = k \, mod \, m$
  - Not suitable for m=2^p, 10^p, 2^p-1 (Q: why?)
  - Suitable, if m is a prime number far from 2^p
- Hashing by multiplication
  - $h(k) = \lfloor m \cdot (k \cdot A \, mod \, 1) \rfloor$, where 0<A<1
  - If m is a power of 2, h(k) computes easily
  - Knuth recommends A = (√ 5 -1)/2, a golden ratio φ

Hashing by multiplication

- $h(k) = \lfloor m \cdot (k \cdot A \bmod 1) \rfloor$
  - An idea, for m=2^p ; k as a floating point: 1/2=<k<1
  - A word length: w bits



Open addressing

- All elements are in a hash table, so a load factor is $\alpha<1$
- We <u>compute</u> indices to the table instead of having explicit pointers in lists

  → a bigger table in the same memory

- A sequence of trials depends on a key and on the order of a trial

  $h: U \times \{0..m-1\} \rightarrow \{0..m-1\}$

  – We look at positions h(k,0), h(k,1), … h(k,m-1), which should be a permutation of all positions

- (A rule of thumb: $\alpha \approx 70\% - 90\%$)

Open addressing

- An open addressing supports Find, Insert. An implementation of Delete is nontrivial or impossible
  - Linked lists or Pseudodelete (with a rehashing)
- We want to approximate a uniform hashing:
  - All m! sequences of trials are equiprobable
- Methods (only approximations of a uniform hashing)
  - A linear probing
  - A quadratic probing
  - A double hashing

Open addresing - Methods

- A linear probing
  - $h(k,i) = (h'(k)+i) \bmod m$
- Disadvantages:
  - only m different sequences of trials
  - Primary clusters are created: long sequences of filled slots
- Ex: $\alpha=0.5$, filled positions are:
  1. Even
  2. In the first half of a table
- HW: implement Delete for a linear probing

Open addresing - Methods

- A quadratic probing

  $h(k,i) = (h'(k)+c.i+d.i^2) \mod m$; where $c \neq 0$, $d \neq 0$

  - Parameters c and d must be appropriately chosen to search through the whole table
  - Only m different sequences, but without primary clusters. Only secondary clusters for elements with the same initial position

Double hashing

- Using auxiliary fuctions h1 and h2

  $h(k,i) = (h1(k)+i.h2(k)) \bmod m$

  - If h2(k) is not commensurable with m (no common factor), then a trial sequence goes through the whole table
  - A number of sequences is m^2.
- Examples of possible choices:
  - $m=2^p$; h2(k) is odd
  - m is a prime number, $0<h2(k)<m$ (~a pair of prime numbers)

Open addressing - analysis

- Theorem 1: An expected number of trials in a table with open addressing with a load factor α is $1/(1-\alpha)$ for an <u>unsuccessful</u> search (supposing an uniform hashing)
- Theorem 2: An expected number of trials in table with open addressing with a load factor α is $1/\alpha \ln(1/(1-\alpha))+1/\alpha$ for a <u>successful</u> search (supposing an uniform hashing and an equiprobable search of keys)

Universal hashing

- A problem: Some n keys can be chosen so that they are mapped to a single slot (if |U| > n^2) for each fixed hash function

  → use a randomisation

  - Idea: We choose a hash function randomly and independently of keys from some suitable set of functions
    - A function is selected dynamically (in a run time), but then it is fixed and used for a hash table
- Properties:
  - No particular input (set of keys) is a priori bad (But ...)
  - A repeated use on the same input calls (almost surely) different functions → „an average case“ for any data distribution (!an advantage of univ. hashing)

Universal hashing

- Df: Let H be a finite set of hash functions from U to {0..m-1}. The set H is called *universal* if for all pairs of different keys x,y from U the number of hash functions with a property h(x)=h(y) is
  |H|/m.
- An observation: for a randomly chosen function $h \in H$ is a probability of a collision for two different random elements x, y exactly 1/m. It is the same probability as if values h(x) and h(y) are chosen randomly and independently from {0..m-1}.

Universal hashing

- Theorem: let h be randomly chosen from a universal set of hash functions and it is used to hash n keys to a table with the size m, n=<m. Then expected number of collisions of a random key x is less than 1.
- Remark: a precondition n=<m means, that an average number of keys in one slot is less than 1

Construction

- We choose a prime number m and we split each key x to (r+1) parts. We write x= <x_0,..,x_r>. The r is chosen by the way that each x_i is (strictly) less than m.
- Let a = <a_0,...,a_r> be a sequence of (r+1) independent random numbers from {0..m-1}.
- Let $h_a(k) = (\sum_{i=0}^{r} a_i x_i) \bmod m$, and $H = \cup_a \{h_a\}$.
- It holds: $|H| = m^{r+1}$ (the number of different as)
- A theorem: H is a universal set of hash functions

Example

- Ex: For x with a fixed size description: choose a length of each x_i as 1 bit and select corresponding a_i.
  - Usually: a bit is set for some property/attribute of x
  - An advantage: a quick update, incrementally
    - (A hash is sometimes computed using bitwise XOR instead of modulo)
    - (Board) positions in games

(Dynamic hash tables)

- Disadvantages of hash tables
  - A fixed size m
  - Only a pseudodelete operation
- An idea: a periodic reconstruction of data struct's
  - Measurements by an amortised complexity: a worst case in a seq. of operations
  - Rehashing (eager or lazy), with a new function
  - Increasing the size: 2 times (or d times), if a table is full (wrt. α); also delete pseudodeleted elements
  - Decreasing the size: 2 times, if the table has $\alpha.m/8$ elements
    - At least O(m) operations from previous reconstruction

Graphs

- Representations of graphs: $G=(V,E)$
  - Vertices V, $|V| = n$; Edges E, $|E| = m$

1. A neighbourhood matrix:
   - $A=(a_{ij})$ with a size $|V|x|V|$, a used memory $\Theta(n^2)$
   - $a_{ij} = 1$ iff $(v_i,v_j) \in E$, and $a_{ij} = 0$ otherwise
2. A list of neighbours (a sparse representation):
   - An array of size |V|: pointers to lists (or arrays)
   - A memory: $\Theta(n+m)$
3. By a computation: isEdge(i,j), getNeighbours(i)

- For undirected and directed graphs, without and with weights (on edges)

Searching of graphs

- Two basic methods
  - A depth first search (DFS)
    - Using a stack
  - A breadth first search (BFS)
    - Using a queue
- A „three colours“ notation
  - White for unvisited nodes, grey for processed (open), and black for finished (closed) ones
  - An invariant: no edge from any black node to a white one

A breadth first search

BFS(G,s) ; c: colour, d: distance, p: predecessor

```
1 forall u in V do c[u]:=white; d[u]:=Maxint;
2                  p[u]:=NIL
3 c[s]:=grey; d[s]:=0; Queue:={s};
4 white Queue not empty do
5   u:=getFrom(Queue);
6   forall v in neighbours(u) do
7     if c[v]=white then   ; d[v]=Maxint
8       c[v]:=grey; d[v]:=d[u]+1; p[v]:=u
9       addTo(v,Queue)
10  c[u]:=black; deleteFrom(u,Queue)
```

BFS

- Notes:
  - Searches a graph in levels according to a shortest path (i.e. a number of edges from s)
  - Visits all accessible nodes and creates a tree of shortest paths (p[] in alg.)
    - A reconstruction of a tree backwards (a „design pattern“)
  - Works also for directed graphs (without changes)
  - Is a base for other algoritms (a shortest path, a min. spanning tree)
  - Has a running time $\Theta(n+m)$, if a list-of-neighbours repr. is used

BFS: applications

- A test of connectedness of G
  - Choose any vertex s and run BFS(G,s)
  - If any vertex remains white, the graph is not connected
- Counting of connected components of G
  - Run repeatedly BFS from any white node till some white node exists
- A test for a bipartite graph
- In $\Theta(n+m)$ time

Depth first search

- Active (grey) nodes are stored on a stack
  1. Recursive calls: an implicit stack
  2. An explicit stack, in a data structure
- For directed graphs
- We use global time events: opening a node (d[]) and closing a node (f[])
- A representation of G: lists of neighbours

A depth first search

DFS(G) ; c: colour, d: entry time, f leaving time, (p: predecessor)

```
1 forall u in V do c[u]:=white;
2 time:=0
3 forall u in V do if c[u]=white then VISIT(u)
4 procedure VISIT(u)   ;recursive version
5   c[u] = grey;
6   time++ ; d[u]:=time
7   forall v in neighbour(u)
8     if c[v]=white then VISIT(v);
9   c[u]:=black
10  time++ ; f[u]:=time
```

DFS applications

- As BFS
- A test for a cycle in G

- (A serialisation of a memory)
- (A garbage collection)
- (An implementation for implicit graphs: an iterative deepening)

Classification of edges

- For DFS in a directed graph

1. (i,j) is <u>a tree edge</u> iff j was searched from i; white j during a visit of (i,j)
2. (i,j) is <u>a backward edge</u> iff j is an ancestor of i in a DFS tree; grey j during a visit of (i,j)
3. (i,j) is <u>a forward edge</u> iff i is an (indirect) ancestor of j in a DFS tree; black j during a visit of (i,j) and d[i]<d[j]
4. (i,j) is <u>a crossing edge</u> otherwise; black j during a visit of (i,j) and d[i]>d[j]

- For undirected graphs: only tree and backward edges

DFS

- Properties:
  - Tree edges create a directed forest
    - a DFS forest: a set of DFS trees
  - Intervals $[d(i),f(i)]$ create a „good parenthesization“: for each $i \neq j$ one of the following statements is valid:
    - $[d(i),f(i)] \cap [d(j),f(j)] = \emptyset$
    - $[d(i),f(i)] \subset [d(j),f(j)]$ – i is a successor of j in a DFS tree
    - $[d(i),f(i)] \supset [d(j),f(j)]$ – j is a successor of i in a DFS tree
  - Corollary: j is a successor of i in a DFS tree iff an interval for j is included in an interval for i
  - Time: $\Theta(n+m)$

Topological sorting

- Df: A function $t: V \rightarrow \{1..n\}$ is a topological numbering of V if for each edge (i,j) holds $t(i) < t(j)$.
  - Another view: A topological ordering is a sequence of vertices, where all edges go from the left to the right
- An observation: A topological numbering exists only for acyclic graphs
  - No cycle ↔ no backward edge during DFS
  - DAG – a Directed Acyclic Graph

Algorithms

- Naive:

1. Find a vertex with no outgoing edge and assign a last free number to it.
2. Delete the numbered vertex from a graph and if the graph is not empty, goto 1.

- Time complexity: $\Theta(n.(n+m))$

Algorithms

- A topological sorting(G), in time Θ(n+m)

1. Compute DFS(G)
2. If a backward edge exists then
3.   Return „impossible – the graph is not DAG“
4. Store all closed vertices to a head of a list S during DFS ; no additional sorting
5. Return S

- Theorem: A numbering of vertices of DAG according to decreasing values of closing times f(i) is topological.

Transitive closure

- Df: A graph G'=(V,E') is a transitive closure of a directed graph G=(V,E) if for all pairs of vertices i, j, where i≠j, holds:

  if there is a directed path from i to j in G, then $(i, j) \in E'$

- A transitive closure G' represented by a neighbourhood matrix is a matrix of accessibility of G
  - The matrix can be computed in Θ(n.(n+m)) using DFS n times

Strongly Connected Components

- Df: Let G=(V,E) be a directed graph. A set $K \subset V$ is a <u>strongly connected component</u> if it holds:

1. For each i,j from K a directed path from i to j and a directed path from j to i exist in G.
2. K is a maximal set fulfilling the condition 1.

- Ad 2: There is no strict superset L of K fulfilling 1.
- Corollary: each node belongs to a single SCC and all SCCs create a decomposition of V
- A naive alg.: it uses a transitive closure, then it reads SCCs from the matrix in Θ(n^2)

SCC algorithm

- Input : G=(V,E)

1. Find all closure times d(u) of vertices using DFS, return them in a linked list in a decreasing order
2. Make G' , the transposition of G
3. DFS(G'), where its main cycle selects vertices in an ordering from the step 1.

- An output: DFS trees from 3. are SCCs of G
- Df: Let G = (V,E). A graph G'=(V,E'), where

  $(i,j) \in E' \Leftrightarrow (j,i) \in E$, is *a transposition* of G

  - A usual notation: G^T .

Properties of SCC alg

- A transposition G' can be constructed in time $\Theta(n+m)$

  $\rightarrow$ An SCC alg. runs in time $\Theta(n+m)$

- Lemma: Let G=(V,E) be a directed graph and K is SCC in G. It holds after an SCC algorithm:

  1. K is a subset of vertices of a single DFS tree
  2. K creates a subtree in the constructed DFS tree

- Ideas: 1. from def. of SCC, a whole comp. is visited
- 2. Accessible nodes outside a current component K were closed before visiting K during the step 3.

Minimal Path Problem in G

- We use:
  - A directed graph G=(V,E)
  - A weight function w: E → R
  - A weight of a path $P = \langle v_{0,} v_1 \ldots v_k \rangle$ is
    $$w(P) = \sum_{i=1}^{k} w(v_{i-1}, v_i)$$
- Df: A weight of a shortest path from u to v is
  $$d'(u,v) = min\{w(P); P \text{ is a path from } u \text{ to } v\}$$
  - if there is no path, we set $d'(u,v) = \infty$

Variants

- A shortest path from u to v is any path P from u to v with w(P) = d'(u,v)
- Usually: a minimal path, also maximal/extremal (DAG)
- Variants of the minimal path algorithm:

  1. From a fixed vertex s to a fixed vertex v
  2. From a fixed s to all x $\in V$
  3. From x to y, for all x,y $\in V$, later

- An overview of methods for a single source s:
  - An acyclic graph (and any weights) → alg. DAG
  - Nonnegative weights (in any graph) → Dijkstra alg.
  - No restrictions → Bellman-Ford alg.

Observations

- 1. Any subpath from u to v of a shortest path P is a shortest path from u to v
- 2. Let P be a shortest path from s to v and (u,v) is its last edge. Then d'(s,v) = d'(s,u) + w(u,v)
- 3. For all edges (u,v): d'(s,v) <= d'(s,u) + w(u,v)
- An idea: each vertex v has a value d(v) and it holds: d(v) >= d'(s,v) … an invariant
  - d(v) represents a lenght of some path
- A reconstruction of a path: (again) using a predecessor array p

Relaxation

- A relaxation – an improving of estimates:

```
1 Relax(u,v,w):  ; a relaxation of the edge (u,v)
2   if d(v)>d(u)+w(u,v) then
3     d(v):=d(u)+w(u,v)
4     p(v):=u ; store the previous vertex on a shortest path to v
5 end;
```

- A relaxation is used (in some order) repeatedly
- An initialisation: set d(s):=0, for other v: d(v):=∞

Relaxation

4. If (u,v) is an edge, then after relaxation d(v)<=d(u)+w(u,v) is valid.

5. The formula d(v)>=d'(s,v) is valid after initialization and remains valid after any relaxation steps. If d(v) reaches the (unknown) value d'(s,v), then it stops changing.

6. Let $P=\langle v_{0,} v_{1,} \dots , v_n \rangle$ be a shortest path from $s=v_0 \text{ to } v_n=v$. Then after relaxing of edges $(v_{0,} v_1), (v_{1,} v_2) \dots , (v_{n-1}, v_n)$ in this order, it is true that d(v) = d'(s,v)

- Algorithms must process any possible shortest path

Algorithm DAG

- Also: An algorithm of a Critical Path (CP Method)

1. DAG(G,w,s):
2. Sort V(G) topologically
3. Initialization(G,s)
4. For each vertex u in a(n increasing) topological ordering do
5. For each edge (u,v) do Relax(u,v,w)

- After DAG() stops, for all $s,v \in V$: $d(v)=d'(s,v)$

Alg. DAG

- A time complexity: $\Theta(n+m)$
- Applications: Edges are processes, weights represent durations of processes. A graph expresses dependencies in a control flow of a project. We look for a (maximal) *critical path.* A delay of a process on any critical path delays the whole project.
  - Looking for a maximal path: 1. weights are negative, or 2. an initialization with $-\infty$ and Relax with a reversed comparison

Dijkstra alg.

- Assumption: All weights are nonnegative.
  - → No negative cycles
- Vertices are divided to the sets: S and Q = V \ S
  1. v is in S: its shortest path from s is correctly computed: d(v) = d'(s,v), and outgoing edges from v are relaxed
  2. Otherwise v in Q: Q is a data structure supporting search for v with a minimal value d(v)
- We start with: Q=V, S=Ø (after an initialization)
- A data structure for Q: a heap = a priority queue

Dijkstra alg.

1. Dijkstra(G,w,s)
2. Initialization(G,s)
3. S:=Ø, Q:=V
4. while Q≠Ø do
5. u:=Extract-Min(Q)
6. S:=S $\cup$ {u}
7. for each v$\in$V with (u,v)$\in$E do Relax(u,v,w)

Correctness

- Let $G=(V,E)$ be a directed weighted graph with nonnegative weights and s is any vertex of G.

  Then after Dijkstra(G,w,s) it is true that

$d(v)=d'(s,v)$ for all v from G

- A time complexity:
  - n times Extract-Min, m times Decrease-key (in Relax)

  1. $\Theta(n^2)$: Q as an array
  2. $\Theta((n+m) \log n)$: Q as a binary heap
  3. ($\Theta(n \log n + m)$: Q as a fibonacci heap)

(Min) Heaps

- Heap operations:

1. Insert(H,k) – it inserts a key k into the heap H
2. Extract-Min(H) – it returns a minimal key in H
3. Decrease-key(H,ptr_k,val) – it decreases k to val

  - No operation Find(H,k)

- Implementations (of a binary heap):
  1. In an array: a[i] has children at a[2*i] and a[2*i+1]
     - if a[] starts at 1
  2. In a (balanced) binary tree

  - A heap invariant: children are greater than their parent

Example

- An example of using heaps: The algorithm Heapsort for n elements in increasing ordering:

1. Insert n elements into a heap
   - Or create a heap in a „batch mode“ in O(n)
2. Use Extract-Min(H) n times
   - For in-place sorting in an array:
   - Use Max-Heap and put/exchange max elements to the end of an unsorted part

Alg. Bellman-Ford

- Slower than Dijkstra, but it works also with negative edges (but no negative cycles)
  - A note: negative cycles and a maximal path
    - The algorithm does not check if a path is a simple path, so results can be wrong in case of negative cycles
- An output of the algorithm:
  - FALSE, if G contains a negative cycle accessible from an initial vertex s
  - TRUE, otherwise, with d[], p[]

Implementation

- Bellman-Ford(G,w,s):

1. Initialization(G,s)
2. for i:=1 to |V|-1 do ; (n-1) is a length of max. path
3.   for each (u,v) in E(G) do Relax(u,v,w)
4. for each (u,v) in E(G) do ; a search of a neg. cycle
5.   if d(v) > d(u) + w(u,v) return FALSE
6. return TRUE

- A time complexity: O(nm)

Properties

- We have a graph G, a weight function w and a start vertex s
- If a negative cycle is reachable from s, then the Bellman-Ford alg. returns FALSE
- Otherwise, the alg. returns TRUE and for all vertices v it is true that d(v)=d'(s,v)
  - The alg. relaxed all paths up to the length n-1
- A note: a triangle inequality is not true in a graph with a negative cycle

All shortest paths

- A goal: compute all shortest paths d'(u,v)
- Prepare a neighborhood matrix W:
  1. $w_{uv} = 0 \; if \; u = v$
  2. $w_{uv} = w(u,v) \; if \; (u,v) \in E$
  3. $w_{uv} = \infty \; if \; (u,v) \notin E$
- We allow negative edges, but do not allow negative cycles.
- We can use Critical Path, Dijkstra, and Bellman-Ford n times with a time complexity O(n(n+m)), O(n^3), and O(n^4), respectively

„Matrix multiplication“ algorithm

- We use an induction on a number of vertices on a shortest path
- Define: d_{uv}^{k}= a minimal path from u to v with at most k edges

1. $k=1: d_{uv}^{1}=w_{uv}$
2. $a\ step: k-1 \rightarrow k: d_{uv}^{k}=min(d_{uv}^{k-1}, min_{1 \leq l \leq n}(d_{ul}^{k-1}+w(l,v)))=$
$min_{1 \leq l \leq n}(d_{ul}^{k-1}+w(l,v))$

  - w(v,v)=0 enable to shorten expr. in the last equality

- We use matrices: $D^{k}=(d^{k})_{uv}; W=w_{uv}$

„Matrix multiplication“ algorithm

- We want: $D^{(n-1)}$, where $D^{(k+1)} = D^{(k)} \otimes W$, $D^{(1)} = W$
- $\otimes$ uses a special operation instead of a dot product. The special operation uses
  1. a summation instead of a multiplication
  2. a minimisation instead of a summation
- If G has no negative cycles, then any shortest path is simple, without cycles → a shortest path has at most n-1 edges
- A slow algorithm computes the result with n-2 matrix multiplications → time compl. O(n^4)

„Matrix multiplication“ algorithm

- A quick alg. computes only powers → a time complexity $O(n^3 \log_2(n))$
- We need to test for negative cycles
  1. A negative number on a diagonal
  2. A computation of D did not stabilise: D^2 ≠ D
  3. A final test for a relaxation as in Bellman-Ford alg.
- An implementation note: The matrix D can be computed „in-place“
  - Because each number in a matrix corresponds to some path

Floyd-Warshall algorithm

- An algorithm has similar idea as a matrix multiplication algorithm: it builds a final result from smaller optimal parts („dynamic programming“).
- d_{uv}^{k} = a minimal path from u to v through (internal) vertices {1..k}
- $d_{uv}^{0} = w(u, v)$
- $d_{uv}^{k} = min(d_{uv}^{k-1}, d_{uk}^{k-1} + d_{kv}^{k-1})\ for\ k > 0$

Floyd - Warshall alg.

- Floyd-Warshall(G,w)

```
1. D^0 := W
2. for k:=1..n do
3.   for u:=1..n do
4.     for v:=1..n do
5.       d[u,v]:=min(d[u,v], d[u,k]+d[k,v])
6. return D
```

- A time complexity: O(n^3)
- Correctness: any vertex can be an internal vertex → we have tried all paths

Minimal Spanning Tree, MST

- An input: A connected graph G=(V,E) with a weight function w: $E \rightarrow R$
- A goal: to find a minimal spanning tree G'=(V,T) of G
  - A spanning tree: a connected acyclic subgraph
  - A weight of a tree: a sum of edge weights
- As $|T|=|V|-1$ in a tree, we can suppose that $w(e) >= 0$
- Idea: we add edges to a set of edges A which is permanently a subset of some MST
  - It is a greedy algorithm

MST

- Def: Let A be a set of edges, which is a subset of a minimal spanning tree. An edge e is *safe* for A, if $A \cup \{e\}$ is also a subset of some MST.
- generic_MST(G,w):

1. $A := \emptyset$
2. for i := 1 to n-1 do
3. find safe edge $(u,v) \in E$
4. $A := A \cup \{(u,v)\}$
5. return A

MST

- A *cut* (of a graph) is a partition of vertices to two parts $(S, V \setminus S)$.
- An edge (u,v) *crosses* a cut $(S, V \setminus S)$, if $|\{u,v\} \cap S| = 1$
- A cut *respects* a set of edges A, if no edge from A crosses the cut.
- An edge is *light* for a cut, if its weight is the smallest weight among all edges that cross the cut

MST

- Theorem: Let $G=(V,E)$ be a connected graph with a weight function $w: E \to R$, a set of edges A is a subset of a MST, and $(S, V \setminus S)$ is any cut, which respects A.

  Then, if $(u,v) \in E$ is a light edge, then it is a safe edge
- Idea of a proof: an exchange property
- Corollary: if C is a component of a graph given by A, then any minimal edge (u,v) between C and other components is safe for A.

MST, strategies

1. Algorithm Borůvka 1926, Kruskal 1956
    1. selects a minimal edge between two components of A
    2. The set A is a forest (a set of trees)
    3. Two trees are connected to a single tree in each step
2. Algorithm Jarník 1930, Prim 1957
    1. The set A is a single tree
    2. The alg. selects a minimal edge between A and some other component (i.e. a vertex)

Borůvka – Kruskal alg.

- Borůvka_Kruskal(G,w)

```
1 sort all edges to nondecreasing order according
  to their weights
2 A := Ø
3 foreach v in V do Make_Set(v)
4 foreach (u,v) in E in precomputed order do
5   if Find_Set(u) ≠ Find_Set(v) then
6     A := A ∪ {(u,v)}
7     Union(u,v)
8 return A
```


- Alg. uses a Union-Find data structure

A data structure Union-Find

- Time complexity:
  - Edges represented using linked lists: $\Theta(m \log m)$
    - dominated by sorting of edges

- Used operations:
  - Make_set: n times
  - Union: n-1 times
  - Find-Set: at most 2m times

A data structure Union-Find

- An implementation of a Union-Find structure
  1. In an array: Each vertex points to a representant
     - Union: $\Theta(n)$
     - Find-Set: $\Theta(1)$
     - total: $\Theta(n^2+m)$
  2. Using pointers (in an array or in a „tree“ structure)
     - Union: $\Theta(1)$ (A) – an implementation „trick“
     - Find-Set: $\Theta(\log n)$, using (A)
     - total: $\Theta(n+m \log n)$
- (A): A root of the smaller tree will point to a root of the bigger tree → a depth is $\Theta(\log n)$

Jarník – Prim alg.

- Jarník_Prim(G,w,r)

```
1 Q := V; A := Ø
2 foreach v in V do key(v) = ∞
3 key(r) := 0; p(r) := NIL
4 while Q ≠ Ø do
5   u := Extract_Min(Q); add (p(u),u) to A
6   foreach v in V s.t. (u,v) in E do
7     if v in Q and key(v) > w(u,v) then
8       key(v):=w(u,v); p(v):=u
9 return A
```

- The algorithm uses a heap for vertices

Jarník – Prim alg.

- A time complexity
  - A heap as an array: $\Theta(n^2)$
  - A heap as a binary heap: $\Theta(m \log n)$
- Used operations:
  - Insert: n times, for vertices
  - Extract-Min: n times, for vertices
  - Decrease-key: m times, for edges

Divide et Impera

- A method for design of algorithms
- An algorithm of this type has usually 3 steps:
  1. Divide a problem to some smaller subproblems of the same type
  2. Solve subproblems
     1. recursively using another division, if they are big enough
     2. directly for small subproblems (often trivial)
  3. Combine solutions of subproblems to get a solution of an original problem
- Examples: Mergesort, Binary searching

Complexity analysis

- $T(n)$: a time for solving a problem of a size n
  - We suppose: if $n<k$, then $T(n) = \Theta(1)$
- $D(n)$: a time for a division of a problem to #a subproblems of a size n/c, and for combining solutions of subproblems

→ a recurrent equation:

  - $T(n) = a.T(n/c) + D(n)$, for $n>=k$
  - $T(n) = \Theta(1)$, for $n<k$

Methods of solving

1. A substitution method
2. A Master Theorem

- A simplification:
  1. An assumption T(n) = Θ(1) is not written explicitely
  2. We ignore rational parts and use only n/2 instead of $\lceil n/2 \rceil$ and $\lfloor n/2 \rfloor$
  3. We use an asymptotic notation in equations, as we are interested only in an asymptotic solution
- Ex: Mergesort: T(n) = 2 T(n/2) + Θ(n)
- BinSearch: T(n) = T(n/2) + Θ(1)

Substitution method

- To guess an asymptotically correct solution
- To check correctness using an induction
  - separately for an upper and lower bound
- A pitfall:
  - Constants in an Induction hypothesis and Induction conclusion must be the same; a proof is for a fixed c
- Ex: Mergesort: $T(n) = 2\ T(n/2) + \Theta(n)$,
  - A solution $T(n) = \Theta(n \log n)$
  - An induction: $T(n) < \underline{c}.n \log n - d.n$ for $T(n)=O(n \log n)$

Quick multiplication

- A (slow) basic-school multiplication of long numbers with n bits has a time complexity O(n^2)
- A quick multiplication: T(n) = 3.T(n/2)+b.n
  - A solution: $T(n)=O(n^{\log_2 3})$ (using a master theorem)
- An induction: $T(n)<c.n^{\log_2 3}-d.n$
  - The part „-d.n“ creates a reserve for an overhead
- An alg. computes A, C, B recursively using 3 multiplications „*“:

$$(x_1\cdot p+x_2)*(y_1\cdot p+y_2)=A\cdot p^2+B\cdot p+C\,,\, where\; p=2^{n/2}$$

$$A=x_1*y_1\,;\, C=x_2*y_2$$

$$B=(x_1+x_2)*(y_1+y_2)-A-C=x_1*y_2+x_2*y_1$$

Examples

- T(n) = 2 T(n/2) + Θ(n), Mergesort, Fast Fourier Transform
- T(n) = 4 T(n/2) + Θ(n), the classical multiplication
  - T(n) = Θ(n^2)
- T(n) = 3 T(n/2) + Θ(n), a quick multiplication
  - $T(n) = \Theta(n^{\log_2 3})$
- T(n) = T(n/5)+T(7n/10)+Θ(n), Median/k-th elem.
  - T(n) = Θ(n) … only a substitution method; using 1/5+7/10<1
- T(n) = 4 T(n/3) + Θ(1), (a „fractal“ drawing)
  - $T(n) = \Theta(n^{\log_3 4})$
- T(n) = 8 T(n/2) + Θ(n^2), Matrix multiplication
  - T(n) = Θ(n^3)

Master Theorem

- Let a≥1, c>1, d≥0 are real numbers and T: N → N is a nondecreasing function, such that for all n expressed as c^k, k∈N, holds:

  T(n) = a.T(n/c) + F(n)

  where F: N → N fullfils $F(n)=\Theta(n^d)$. Let $x=\log_c a$.
- Then

a) If x<d, then $T(n)=\Theta(n^d)$

b) If x=d, then $T(n)=\Theta(n^d \log_c n)=\Theta(n^x \log_c n)$

c) If x>d, then $T(n)=\Theta(n^x)$

Matrix multiplication

- An input: matrices A and B of an order n x n
- An output: C = A $\otimes$ B, also of an order n x n
- If $n=2^k$, we can reformulate an alg. using Divide et impera method

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \qquad B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} \qquad C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}$$

$$C_{11} = (A_{11} \otimes B_{11}) \oplus (A_{12} \otimes B_{21})$$
$$C_{12} = (A_{11} \otimes B_{12}) \oplus (A_{12} \otimes B_{22})$$
$$C_{21} = (A_{21} \otimes B_{11}) \oplus (A_{22} \otimes B_{21})$$
$$C_{22} = (A_{21} \otimes B_{12}) \oplus (A_{22} \otimes B_{22})$$

From classical to Strassen Alg.

- We get: T(n) = 8.T(n/2) + O(n^2)
  - A solution from a master theorem: $a = 8, c = 2, \log_c a = 3, d = 2$
  - T(n) = O(n^3), the same as the classic alg.
- To get a lower time complexity, it is necessary to decrease a=8 and preserve (or slightly increase) d=2
- The Strassen algorithm uses 7 multiplication of n/2-submatrices (instead of 8 multiplications classically)

Strassen Alg. 1

- A preparatory computation: 7 multiplications

$$M_1 = (A_{12} \ominus A_{22}) \otimes (B_{21} \oplus B_{22})$$
$$M_2 = (A_{11} \oplus A_{22}) \otimes (B_{11} \oplus B_{22})$$
$$M_3 = (A_{11} \ominus A_{21}) \otimes (B_{11} \oplus B_{12})$$
$$M_4 = (A_{11} \oplus A_{12}) \otimes B_{22}$$
$$M_5 = A_{11} \otimes (B_{12} \ominus B_{22})$$
$$M_6 = A_{22} \otimes (B_{21} \ominus B_{11})$$
$$M_7 = (A_{21} \oplus A_{22}) \otimes B_{11}$$

Strassen Alg. 2

- A final computation of C

$$C_{11} = M_1 \oplus M_2 \ominus M_4 \oplus M_6$$
$$C_{12} = M_4 \oplus M_5$$
$$C_{21} = M_6 \oplus M_7$$
$$C_{22} = M_2 \ominus M_3 \oplus M_5 \ominus M_7$$

- A time complexity: T(n) = 7.T(n/2)+O(n^2)
- The Master Theorem: $a=7, c=2, \log_c a = \log_2 7 = x, d=2$

$$T(n) = O(n^x) \approx O(n^{2.81})$$

- Note: Numbers in submatrices get bigger (by 1 bit)

Strassen alg.

- Note: The Strassen algorithm needs a „minus“ operation (an inversion op. to „plus“) → it works over a ring (an algebraic structure with „+“, „-“ and „*“, without „/“)
  - It cannot be used directly for a boolean multiplication (e.g. for a transitive closure in graphs), but can be used for the 0-1 representation of boolean matrices
- Schema for pictures:
  - A: 1st row first
  - B: 1st column first

$$
\begin{array}{c|cccc}
 & B_{11} & B_{21} & B_{12} & B_{22} \\
A_{11} & + & \cdot & \cdot & \cdot \\
A_{12} & \cdot & + & \cdot & \cdot \\
A_{21} & \cdot & \cdot & \cdot & \cdot \\
A_{22} & \cdot & \cdot & \cdot & \cdot
\end{array}
$$

Strassen alg.

- We want 4 submatrices of C:

$$
\begin{array}{cc}
 & B_{11} \quad B_{21} \quad B_{12} \quad B_{22} \\
C_{11} = \begin{array}{c} A_{11} \\ A_{12} \\ A_{21} \\ A_{22} \end{array} & \begin{pmatrix} + & . & . & . \\ . & + & . & . \\ . & . & . & . \\ . & . & . & . \end{pmatrix}
\end{array}
$$

$$
C_{12} = \begin{pmatrix} . & . & + & . \\ . & . & . & + \\ . & . & . & . \\ . & . & . & . \end{pmatrix} C_{21} = \begin{pmatrix} . & . & . & . \\ . & . & . & . \\ + & . & . & . \\ . & + & . & . \end{pmatrix} C_{22} = \begin{pmatrix} . & . & . & . \\ . & . & . & . \\ . & . & + & . \\ . & . & . & + \end{pmatrix}
$$

Strassen alg.

- 7 preparatory computations:

$$M_1 = \begin{pmatrix} \cdot & \cdot & \cdot & \cdot \\ \cdot & + & \cdot & + \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & - & \cdot & - \end{pmatrix} \quad M_2 = \begin{pmatrix} + & \cdot & \cdot & + \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ + & \cdot & \cdot & + \end{pmatrix} \quad M_3 = \begin{pmatrix} + & \cdot & + & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ - & \cdot & - & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{pmatrix}$$

$$M_4 = \begin{pmatrix} \cdot & \cdot & \cdot & + \\ \cdot & \cdot & \cdot & + \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{pmatrix} \quad M_5 = \begin{pmatrix} \cdot & \cdot & + & - \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{pmatrix} \quad M_6 = \begin{pmatrix} \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ - & + & \cdot & \cdot \end{pmatrix}$$

$$M_7 = \begin{pmatrix} \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ + & \cdot & \cdot & \cdot \\ + & \cdot & \cdot & \cdot \end{pmatrix}$$

Strassen alg.

- A final computation of 4 submatrices of C:

$$C_{12} = M_4 \oplus M_5$$
$$C_{21} = M_6 \oplus M_7$$
$$C_{11} = M_1 \oplus (M_2 \ominus M_4 \oplus M_6);$$
$$M_2 \ominus M_4 \oplus M_6 = \begin{pmatrix} + & . & . & \pm \\ . & . & . & - \\ . & . & . & . \\ \pm & + & . & + \end{pmatrix}$$
$$C_{22} = (M_2 \oplus M_5 \ominus M_7) \ominus M_3:$$
$$(M_2 \oplus M_5 \ominus M_7) = \begin{pmatrix} + & . & + & \pm \\ . & . & . & . \\ - & . & . & . \\ \pm & . & . & + \end{pmatrix}$$

A lower bound of Sorting

- The problem of Sorting: To sort an input sequence of the length *n*
  - Many algorithms for the same problem P (also undiscovered, …) → new: a complexity of a problem P
- The asymptotical complexity of Sorting: the complexity of the best algorithm (using the worst case complexity)
- Simple: an upper bound for a complexity of the problem P: Any algorithm for P gives an upper bound for a complexity of the problem P
- Asymptotical lower bound for the Sorting problem: An arbitrary alg. must fulfill the lower bound
  - An idea of an approach: To prove some common characteristic of all algorithms for a problem

Decision tree for Sorting 1

- A sorting algorithm based on comparisons: Branching of a program flow is based only on comparisons (branching in general, not only „if“)
  - Especially not allowed: indirect addressing on a key
- Any deterministic sorting algorithm based on comparisons can be represented by a decision tree – a binary tree with
  1. internal nodes representing a comparison: a test $x \leq y$
     - Left branch if TRUE
     - Right branch if FALSE
  2. leaves representing output permutations of an input

Decision tree for Sorting 2

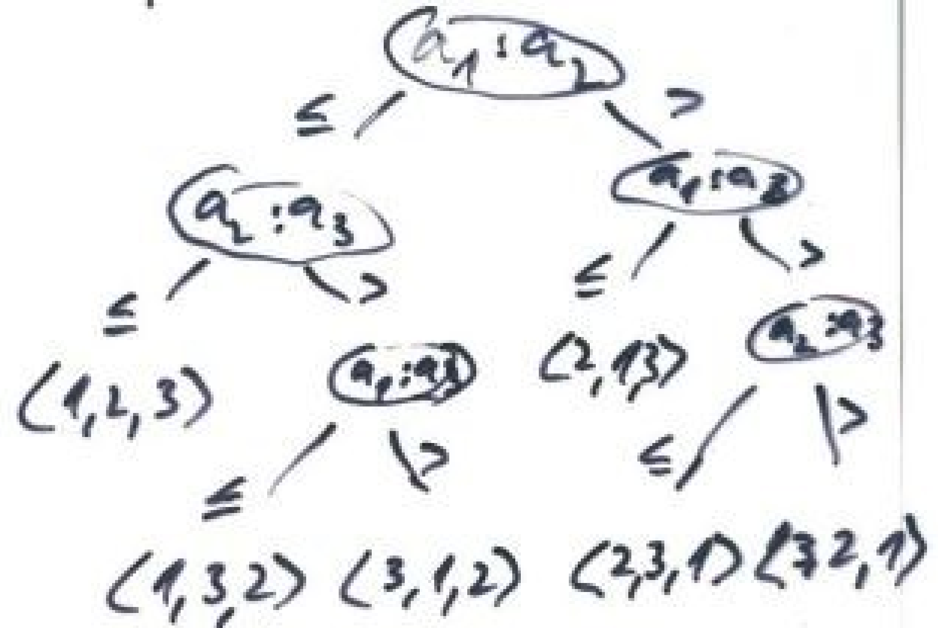
- A tree for a <u>correct</u> sorting algorithm must have all n! permutations (= orderings) in leaves, (A).

  → #leaves >= n!

  - HW: is it possible that #leaves > n! ?

- Ex: alg. Insertsort, n=3

  – (A): If the permutation P is not presented in a tree, then the inverted permutation to P taken as an input cannot be sorted



Decision trees (for sorting)

- A (worst) time complexity of the algorithm = a longest branch in a tree = a depth of the tree
- Th: A binary tree with n! leaves has a depth d∈ Ω(n log n).

$$2^d \geq n! = \prod_{i=1}^{n} i \geq \prod_{i=n/2}^{n} i \geq (n/2)^{n/2} \rightarrow$$

$$d \geq \log(n!) \geq \log((n/2)^{n/2}) \geq n/2 . \log(n/2) \rightarrow$$

$$d \in \Omega(n \log n)$$

Linear time sorting

- Algorithms in this section are not based on comparisons
  - Countingsort
  - Radixsort
- They use keys for adressing (usually in an array)

Counting sort

- Input: n numbers from the interval 1..k
  - We suppose k=O(n)
  - This condition on bounded keys is not present in general sorting algorithms.
- Data structures:
  1. I[1..n] – an input array
  2. O[1..n] – an output array
  3. C[1..k] – a counting array

Counting sort algorithm

```
1.for i:=1 to k do C[i]:=0     ; initialization
2.for i:=1 to n do C[I[i]]++   ; C[i] is # of i in I[]
3.for i:=2 to k do C[i]+=C[i-1]  ; C[i] is # of j, j<=i
4.for i:=n to 1 do             ; put I[i] in a correct place
5.   O[ C[ I[i] ]] := I[i]     ; C[j] points to the last
6.   C[ I[i] ]--               ; ... empty place for j
```

- A time complexity: O(n+k)

Counting sort: properties

- A time complexity: O(n+k)
- A stability of sorting: Equal elements from an input have the same order in an output array
- Impl.: We must copy data in the last pass, as we are interested also in data associated with I[i]. (vs. generate k C[k]-times)

HW

- Change the algorithm, so that the last pass is a forward pass instead of a backward one.
  - You still want a stable sorting
  - The forward pass is more appropiate for streamed data returned from a compresion, from a serialization, or from a magnetic tape :-)

Radixsort

- A historical use: sorting of punch cards
- An observation: if we sort numbers according to the most significant order, we use a stable sorting and an input sequence was sorted according to a less significant orders, than we have a sorted sequence as output

  → Radixsort: sort according to the lowest order, put groups immediately in a sequence and continue sorting according to higher orders.

  - An advantage: we operate with a single sequence

Radixsort

- Countingsort algorithm can be used as a stable algorithm for a single pass
- A current use
  - Sorting of compound keys (e.g. a year, a month, a day)
  - Sorting of alphanumeric keys (words)
- A time complexity: O(d.(n+k)) = O(n), if k=O(n) and d is a constant (d is #digits)
- Notes about padding:
  - Numbers with a different number of digits are padded with zeros on the <u>left</u> side
  - Words with different lengths are padded with blanks on the <u>right</u> side

Randomization of Quicksort

- A problem of a fixed choice of pivot: Some input sequences are bad.
  - We must suppose a uniform distribution of input sequences for an average case
- Randomization: We choose a random element as a pivot instead of a fixed one.
- An average complexity is over all possible choices of pivots (for any input sequence) → we do not need a uniform distribution of inputs
  - But: A particular run (for an input sequence and a choice of pivots) can still be O(n^2)

To do / skipped

-
- Algebraic alg. (LUP decomposition)
- B-trees
-

$\in$ $\cup$