# 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/

## 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 $< x$
- All nodes in the right subtree of a node with key $x$ have value $> x$
- 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 its 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, decides 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.

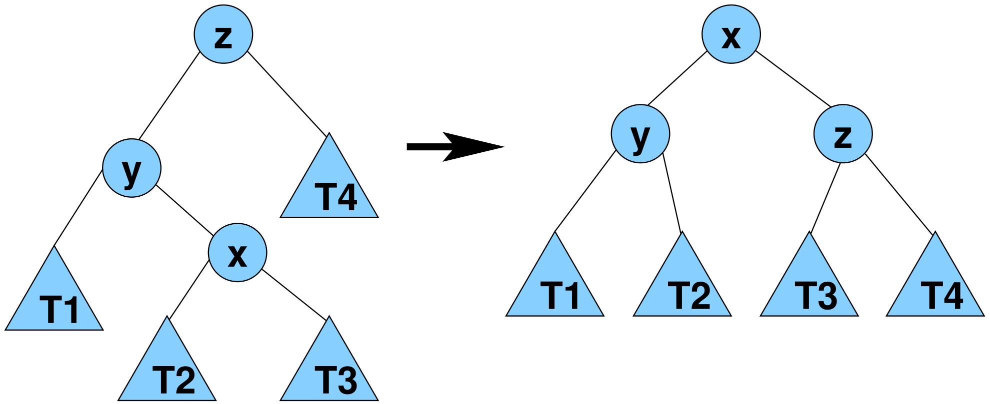
**Details?**

## Splay trees

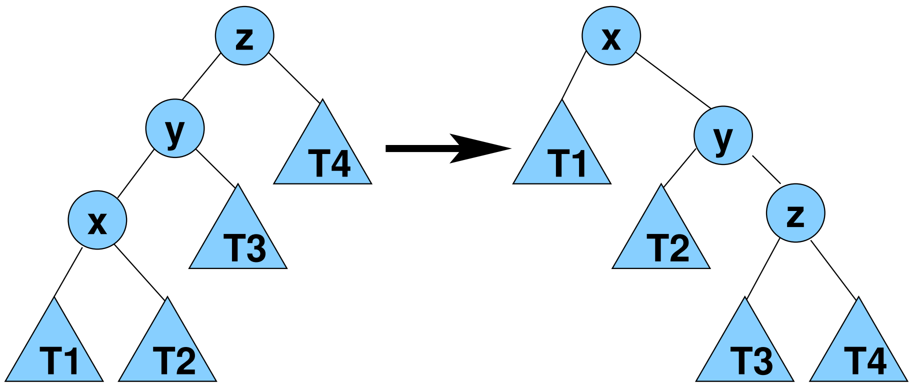
Sleator and Tarjan 1985

- 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

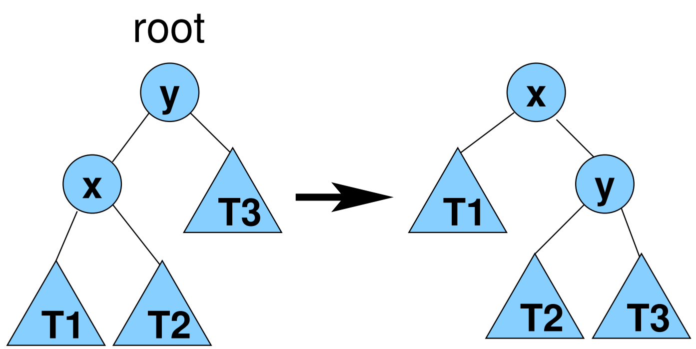
# Splay operation for node $x$: three cases




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 $x$)

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

## Amortized analysis of splaying: exercise

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

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

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

### Questions:

What is the potential of a path with $n$ nodes?

What is the potential of a complete binary tree of height $m$ and $n = 2^{m+1} - 1$?

(asymptotic answers in $\Theta$ notation)

## 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

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 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 $O(\log n)$.

## Splay tree operations (proof of Theorem)

Real cost $c$ in amortized analysis: the number of rotations

In each operation keep running time $O(1 + c)$

**Search(x):** walk down to $x$,

then splay $x$ or the last visited node if $x$ not found

**Insert(x):** insert $x$ as in unbalanced BST, then splay $x$

Inserting increases potential!

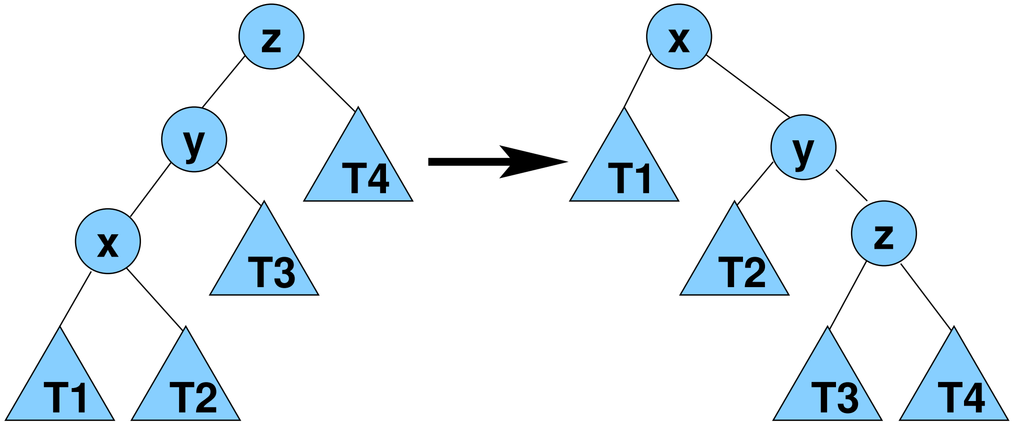
**Delete(x):** delete $x$ as in unbalanced BST

this may in fact delete node for the successor of $x$ if $x$ has 2 children

splay the parent of the deleted node

Deleting does not increase potential

## Proof of Lemma 1, zig-zig case




$r'(x) = r(z)$

$r'(y) < r'(x)$

$r(y) > r(x)$

$D'(x) > D(x)+D'(z)$

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

## Weighted amortized analysis of splaying

Assign each node $x$ fixed weight $w(x) > 0$

$D(x)$: the sum of weights in the subtree rooted at $x$

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

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

### Weighted version of Lemma 2:

Amortized cost of splaying $x$ to the root is

$1 + 3(r(t) - r(x)) = O\left(1 + \log \frac{D(t)}{D(x)}\right)$, where $t$ is the original root before splaying.

## Static optimality lemma

**Weighted version of Lemma 2:** Amortized cost of splaying $x$ to the root is $1 + 3(r(t) - r(x)) = O\left(1 + \log \frac{D(t)}{D(x)}\right)$ where $t$ is the original root before splaying.

**Static optimality theorem:** Starting with a tree with $n$ nodes, execute $m$ find operations where find($x$) is done $q(x) \geq 1$ times. The total access time is

$$O\left(m + \sum_x q(x) \log \frac{m}{q(x)}\right) = O(m(1 + H)),$$

where $H$ is the entropy of the sequence of operations.

**Note:** Lower bound for a static tree is $\Omega(mH)$.

## Sequential access theorem

**Sequential access theorem:** (Tarjan 1985, Elmasry 2004)

Starting from any tree with $n$ nodes, splaying each node to the root once in the increasing order of the keys has total time $O(n)$.

**Note:** trivial upper bound $O(n \log n)$

## 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 $\alpha$ 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.

## 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

## 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)

### Representation:

Each edge solid or dashed

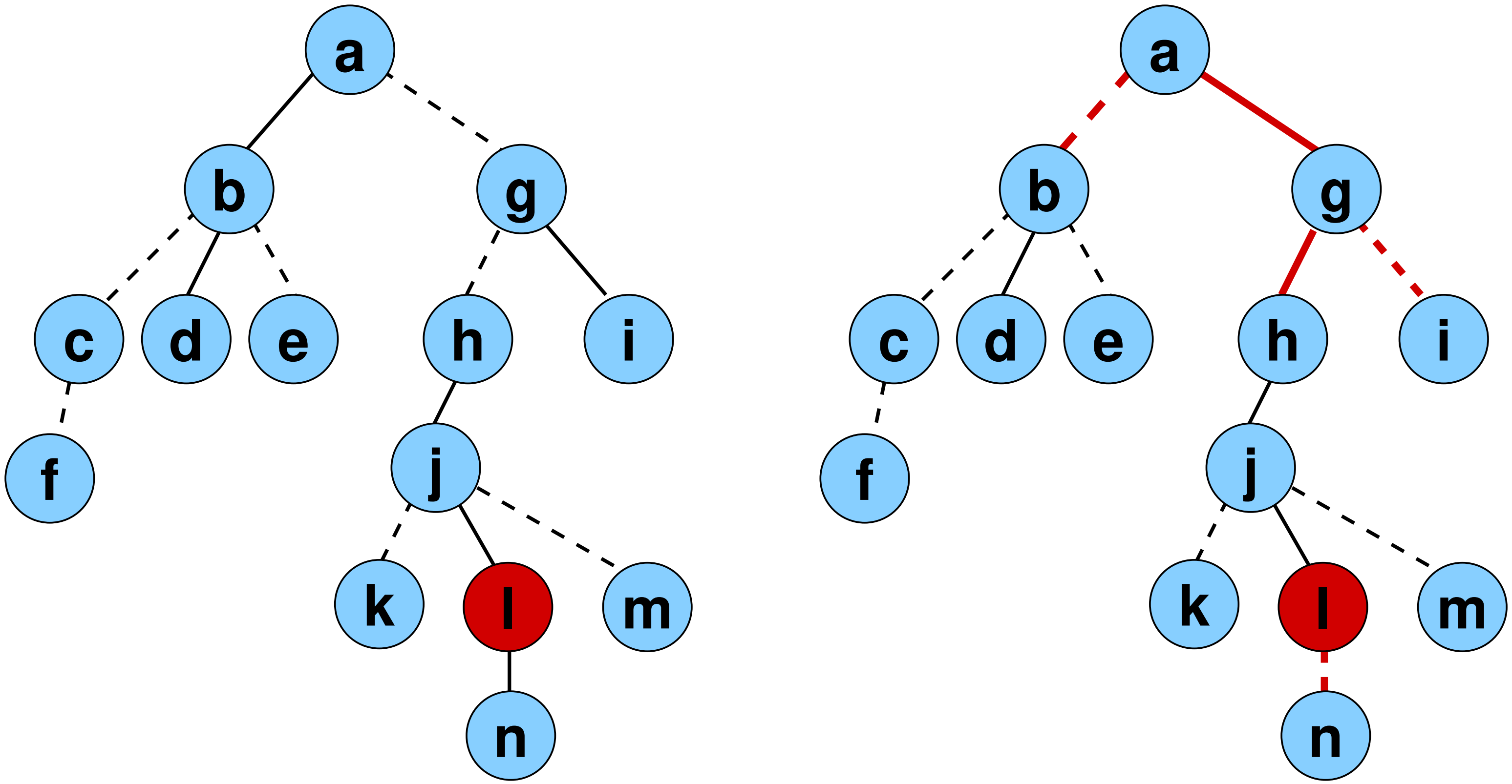
Each node at most one child connected by solid edge

Disjoint solid paths kept in the dynamic path structure (splay trees)

Each head of a path keeps pointer to its parent (dashed edges)

## Operation expose$(\ell)$

Make $\ell$ the lower end of a solid path to root



## Link/cut tree operation via dynamic paths and expose

- findRoot($v$): find root of the tree containing $v$

  expose($v$); findPathHead($v$)

- link($v, w$): make root $w$ a child of $v$

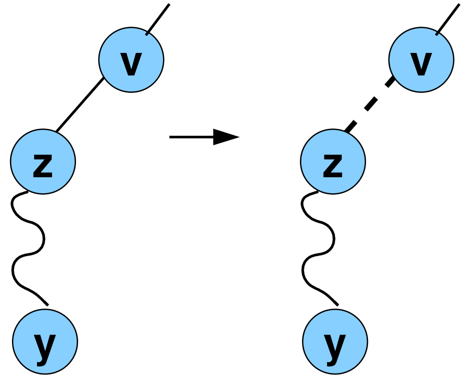
  expose($v$); linkPaths($v, w$)

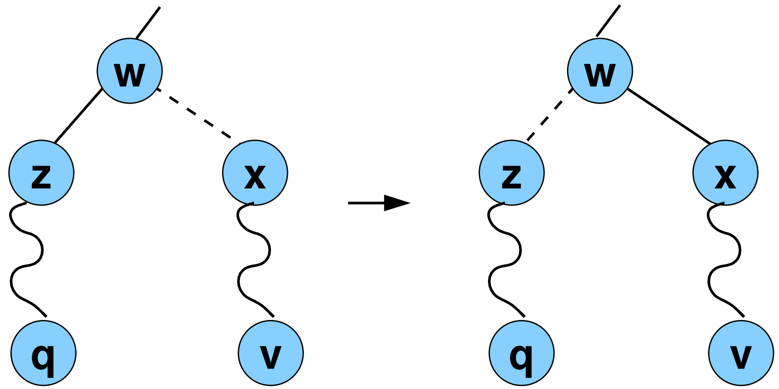
- cut($v$): cut edge connecting $v$ to its parent

  expose($v$); splitPathAbove($v$);

## expose(v)

```
y = splitPathBellow(v);
if(y != NULL) findPathHead(y).dashed = v;
while(true) {
   x = findPathHead(v);
   w = x.dashed;
   if(w == NULL) break;
   x.dashed = NULL;
   q = splitPathBelow(w);
   if(q != NULL) { findPathHead(q).dashed = w; }
   linkPaths(w, x);
}
```



## 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 $\Phi$: 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

## Application: maximum flow problem

In 1983, a different version of link-cut trees has 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.