Algorithms and datastructures I
Lecture 7: tree based data-structures

Jan Hubička

Department of Applied Mathematics
Charles University
Prague

March 24 2020

Kruskal algorithm, 1956

Kruskal algorithm, 1956

Input: Connected graph $G = (V, E)$ and weight function w with unique weights

1. Sort edges by weights; $w(e_1) \leq \cdots \leq w(e_m)$
2. $T \leftarrow (V, \emptyset)$
3. For $i = 1, \ldots m$:
4. $u, v \leftarrow$ vertices in edge e_i
5. If u and v are in different components of T:
6. $T \leftarrow T + e_i$.

Output: Minimum spanning tree T.

Kruskal algorithm, 1956

Kruskal algorithm, 1956

Input: Connected graph $G = (V, E)$ and weight function w with unique weights

1. Sort edges by weights; $w(e_1) \leq \cdots \leq w(e_m)$
2. $T \leftarrow (V, \emptyset)$
3. For $i = 1, \ldots m$:
4. $\quad u, v \leftarrow$ vertices in edge e_i
5. $\quad$ If u and v are in different components of T:
6. $\quad\quad T \leftarrow T + e_i$.

Output: Minimum spanning tree T.

Theorem

Kruskal algorithm finds minimal spanning tree in time $O(m \log n + mT_f(n) + nT_u(n))$ where T_f is time complexity of FIND *and T_u is a time complexity of* UNION *on graph with n vertices.*

Union-find using arrays

Theorem

Kruskal algorithm finds minimal spanning tree in time $O(m \log n + mT_f(n) + nT_u(n))$ where T_f is time complexity of FIND *and T_u is a time complexity of* UNION *on graph with n vertices.*

Union-find using arrays

Theorem

Kruskal algorithm finds minimal spanning tree in time $O(m \log n + mT_f(n) + nT_u(n))$ where T_f is time complexity of FIND *and T_u is a time complexity of* UNION *on graph with n vertices.*

Idea: use array c. For a given vertex v put $c(v)$ to ID of a component it belongs to.

Union-find using arrays

Theorem

Kruskal algorithm finds minimal spanning tree in time $O(m \log n + mT_f(n) + nT_u(n))$ where T_f is time complexity of FIND *and T_u is a time complexity of* UNION *on graph with n vertices.*

Idea: use array c. For a given vertex v put $c(v)$ to ID of a component it belongs to.

Array based union-find

FIND(u,v): $O(1)$ (return true compare if $c(u) = c(v)$)
UNION(u,v): $O(n)$ (search array v and change all occurrences of $c(u)$ to $c(v)$)

Union-find using arrays

Theorem

Kruskal algorithm finds minimal spanning tree in time $O(m \log n + mT_f(n) + nT_u(n))$ where T_f is time complexity of FIND *and T_u is a time complexity of* UNION *on graph with n vertices.*

Idea: use array c. For a given vertex v put $c(v)$ to ID of a component it belongs to.

Array based union-find

FIND(u,v): $O(1)$ (return true compare if $c(u) = c(v)$)
UNION(u,v): $O(n)$ (search array v and change all occurrences of $c(u)$ to $c(v)$)

Theorem

Kruskal algorithm finds minimal spanning tree in time $O(m \log n + mT_f(n) + nT_u(n))$ where T_f is time complexity of FIND *and T_u is a time complexity of* UNION *on graph with n vertices.*

Runtime of complete algorithm: $O(m \log n + m + n^2) = O(m \log n + n^2)$

Union-find using arrays

Theorem

Kruskal algorithm finds minimal spanning tree in time $O(m \log n + mT_f(n) + nT_u(n))$ where T_f is time complexity of FIND *and T_u is a time complexity of* UNION *on graph with n vertices.*

Idea: use array c. For a given vertex v put $c(v)$ to ID of a component it belongs to.

Array based union-find

FIND(u,v): $O(1)$ (return true compare if $c(u) = c(v)$)
UNION(u,v): $O(n)$ (search array v and change all occurrences of $c(u)$ to $c(v)$)

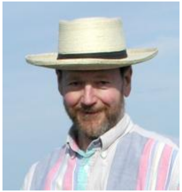
Theorem

Kruskal algorithm finds minimal spanning tree in time $O(m \log n + mT_f(n) + nT_u(n))$ where T_f is time complexity of FIND *and T_u is a time complexity of* UNION *on graph with n vertices.*

Runtime of complete algorithm: $O(m \log n + m + n^2) = O(m \log n + n^2)$

Homework: Try to analyze variant where you always rename the smaller component in time $O(s)$ where s is the size of the component. (it does improve time complexity).

Union-find with “Shrubs” (Gallner, Fisher 1964)

Union-find with “Shrubs” (Gallner, Fisher 1964)

Array P holds predecessor of a vertex (and $\emptyset$ for root).

Union-find with "Shrubs" (Gallner, Fisher 1964)

Array P holds predecessor of a vertex (and $\emptyset$ for root).

Root (v)

1. While $P(v) \neq \emptyset$:
2. $v \leftarrow P(v)$
3. Return v.

Union-find with “Shrubs” (Gallner, Fisher 1964)

Array P holds predecessor of a vertex (and $\emptyset$ for root).

Root (v)

1. While $P(v) \neq \emptyset$:
2. $\quad v \leftarrow P(v)$
3. Return v.

Find (u, v)

1. Return true if Root (u)$=$Root (v).

Union-find with "Shrubs" (Gallner, Fisher 1964)

Array P holds predecessor of a vertex (and $\emptyset$ for root).

Root (v)

1. While $P(v) \neq \emptyset$:
2. $v \leftarrow P(v)$
3. Return v.

Union (u, v)

1. $a \leftarrow$ Root (u), $b \leftarrow$ Root (v)
2. If $a = b$: return
3. $P(b) \leftarrow a$

Find (u, v)

1. Return true if Root (u)$=$Root (v).

Union-find with "Shrubs" (Gallner, Fisher 1964)

Array P holds predecessor of a vertex (and $\emptyset$ for root).

Root (v)

1. While $P(v) \neq \emptyset$:
2. $\quad v \leftarrow P(v)$
3. Return v.

Union (u, v)

1. $a \leftarrow$ Root (u), $b \leftarrow$ Root (v)
2. If $a = b$: return
3. $P(b) \leftarrow a$

Find (u, v)

1. Return true if Root (u)$=$Root (v).

Runtime of Root(v) is bounded by the maximal height of a shrub.

Union-find with "Shrubs" (Gallner, Fisher 1964)

Array P holds predecessor of a vertex (and $\emptyset$ for root).

Root (v)

1. While $P(v) \neq \emptyset$:
2. $\quad v \leftarrow P(v)$
3. Return v.

Union (u, v)

1. $a \leftarrow$ Root (u), $b \leftarrow$ Root (v)
2. If $a = b$: return
3. $P(b) \leftarrow a$

Find (u, v)

1. Return true if Root (u)=Root (v).

Runtime of Root(v) is bounded by the maximal height of a shrub.

Smart optimization: remember height of a tree and always orient new edge from smaller to bigger tree.

Union-find with “Shrubs” (Gallner, Fisher 1964)

Array P holds predecessor of a vertex (and $\emptyset$ for root). H on a root vertex holds the height of its tree.

Root (v)

1. While $P(v) \neq \emptyset$:
2. $\quad v \leftarrow P(v)$
3. Return v.

Union (u, v)

1. $a \leftarrow$ Root (u), $b \leftarrow$ Root (v)
2. If $a = b$: return
3. If $H(a) < H(b)$: $P(a) \leftarrow b$
4. If $H(a) > H(b)$: $P(b) \leftarrow a$
5. If $H(a) = H(b)$: $P(b) \leftarrow a$, $H(a) \leftarrow H(a) + 1$

Find (u, v)

1. Return true if Root (u)=Root (v).

Runtime of Root(v) is bounded by the maximal height of a shrub.

Smart optimization: remember height of a tree and always orient new edge from smaller to bigger tree.

Union-find with “Shrubs” (Gallner, Fisher 1964)

Array P holds predecessor of a vertex (and $\emptyset$ for root). H on a root vertex holds the height of its tree.

Root (v)

1. While $P(v) \neq \emptyset$:
2. $v \leftarrow P(v)$
3. Return v.

Union (u, v)

1. $a \leftarrow$ Root (u), $b \leftarrow$ Root (v)
2. If $a = b$: return
3. If $H(a) < H(b)$: $P(a) \leftarrow b$
4. If $H(a) > H(b)$: $P(b) \leftarrow a$
5. If $H(a) = H(b)$: $P(b) \leftarrow a$, $H(a) \leftarrow H(a) + 1$

Find (u, v)

1. Return true if Root (u)=Root (v).

Runtime of Root(v) is bounded by the maximal height of a shrub.

Smart optimization: remember height of a tree and always orient new edge from smaller to bigger tree.

Invariant

Shrub of height h has at least 2^h vertices

Union-find with “Shrubs” (Gallner, Fisher 1964)

Array P holds predecessor of a vertex (and $\emptyset$ for root). H on a root vertex holds the height of its tree.

Root (v)

1. While $P(v) \neq \emptyset$:
2. $v \leftarrow P(v)$
3. Return v.

Union (u, v)

1. $a \leftarrow$ Root (u), $b \leftarrow$ Root (v)
2. If $a = b$: return
3. If $H(a) < H(b)$: $P(a) \leftarrow b$
4. If $H(a) > H(b)$: $P(b) \leftarrow a$
5. If $H(a) = H(b)$: $P(b) \leftarrow a$, $H(a) \leftarrow H(a) + 1$

Find (u, v)

1. Return true if Root (u)=Root (v).

Runtime of Root(v) is bounded by the maximal height of a shrub.

Smart optimization: remember height of a tree and always orient new edge from smaller to bigger tree.

Invariant

Shrub of height h has at least 2^h vertices

Theorem

Time complexity of UNION *and* FIND *is* $O(\log n)$.

Union-find with path compression

Root (v) with path compression variant 1

1. While $P(v) \neq \emptyset$:
2. $\quad u \leftarrow v$
3. $\quad v \leftarrow P(v)$
4. $\quad$ if $P(v) \neq \emptyset$ then:
5. $\quad\quad P(u) \leftarrow P(v)$
6. Return v.

Root (v) with path compression variant 2

1. $u \leftarrow v$
2. While $P(v) \neq \emptyset$:
3. $\quad v = P(v)$
4. While $P(u) \neq \emptyset$:
5. $\quad w \leftarrow P(u)$
6. $\quad P(u) \leftarrow v$
7. $\quad u \leftarrow w$
8. Return v.

Union-find with path compression

Root (v) with path compression variant 1

1. While $P(v) \neq \emptyset$:
2. $\quad u \leftarrow v$
3. $\quad v \leftarrow P(v)$
4. $\quad$ if $P(v) \neq \emptyset$ then:
5. $\qquad P(u) \leftarrow P(v)$
6. Return v.

Root (v) with path compression variant 2

1. $u \leftarrow v$
2. While $P(v) \neq \emptyset$:
3. $\quad v = P(v)$
4. While $P(u) \neq \emptyset$:
5. $\quad w \leftarrow P(u)$
6. $\quad P(u) \leftarrow v$
7. $\quad u \leftarrow w$
8. Return v.

In 1975 Robert Tarjan shown that adding the path compression reduces the time to $O(\alpha(n))$ where α is the inverse of Ackerman function.

Union-find with path compression

Root (v) with path compression variant 1

1. While $P(v) \neq \emptyset$:
2. $\quad u \leftarrow v$
3. $\quad v \leftarrow P(v)$
4. $\quad$ if $P(v) \neq \emptyset$ then:
5. $\quad\quad P(u) \leftarrow P(v)$
6. Return v.

Root (v) with path compression variant 2

1. $u \leftarrow v$
2. While $P(v) \neq \emptyset$:
3. $\quad v = P(v)$
4. While $P(u) \neq \emptyset$:
5. $\quad w \leftarrow P(u)$
6. $\quad P(u) \leftarrow v$
7. $\quad u \leftarrow w$
8. Return v.

In 1975 Robert Tarjan shown that adding the path compression reduces the time to $O(\alpha(n))$ where α is the inverse of Ackerman function.
Ackerman function is very fast growing function. $A(4)$ is approximately

$$2^{2^{2^{2^{16}}}}$$

Thus we can think of it as an $O(1)$ implementation.

Set datastructure

We would like to represent a set (or a dictionary) of some elements from an universum.
We expect that elements of universum in set can be assigned and compared in $O(1)$

- INSERT(v): Insert v to the set
- DELETE(v): Delete v from the set
- FIND(v): Find v in the set
- SHOW: Print whole set
- MIN: Return minimum
- MAX: Return maximum
- SUCC(v): Find successor
- PRED(v): Find predecessor

Basic implementations

	INSERT	DELETE	FIND	MIN/MAX	SUCC/PRED
Linked list	$O(n)$ or $O(1)$	$O(n)$ or $O(1)$	$O(n)$	$O(n)$	$O(n)$

Set datastructure

We would like to represent a set (or a dictionary) of some elements from an universum.
We expect that elements of universum in set can be assigned and compared in $O(1)$

- INSERT(v): Insert v to the set
- DELETE(v): Delete v from the set
- FIND(v): Find v in the set
- SHOW: Print whole set
- MIN: Return minimum
- MAX: Return maximum
- SUCC(v): Find successor
- PRED(v): Find predecessor

Basic implementations

	INSERT	DELETE	FIND	MIN/MAX	SUCC/PRED
Linked list	$O(n)$ or $O(1)$	$O(n)$ or $O(1)$	$O(n)$	$O(n)$	$O(n)$
Array	$O(n)$ or $O(1)$	$O(n)$ or $O(1)$	$O(n)$	$O(n)$	$O(n)$

Set datastructure

We would like to represent a set (or a dictionary) of some elements from an universum.
We expect that elements of universum in set can be assigned and compared in $O(1)$

- INSERT(v): Insert v to the set
- DELETE(v): Delete v from the set
- FIND(v): Find v in the set
- SHOW: Print whole set
- MIN: Return minimum
- MAX: Return maximum
- SUCC(v): Find successor
- PRED(v): Find predecessor

Basic implementations

	INSERT	DELETE	FIND	MIN/MAX	SUCC/PRED
Linked list	$O(n)$ or $O(1)$	$O(n)$ or $O(1)$	$O(n)$	$O(n)$	$O(n)$
Array	$O(n)$ or $O(1)$	$O(n)$ or $O(1)$	$O(n)$	$O(n)$	$O(n)$
Sorted array	$O(n)$	$O(n)$	$O(\log n)$	$O(1)$	$O(\log n)$ or $O(1)$

Set datastructure

We would like to represent a set (or a dictionary) of some elements from an universum.
We expect that elements of universum in set can be assigned and compared in $O(1)$

- INSERT(v): Insert v to the set
- DELETE(v): Delete v from the set
- FIND(v): Find v in the set
- SHOW: Print whole set
- MIN: Return minimum
- MAX: Return maximum
- SUCC(v): Find successor
- PRED(v): Find predecessor

Basic implementations

	INSERT	DELETE	FIND	MIN/MAX	SUCC/PRED
Linked list	$O(n)$ or $O(1)$	$O(n)$ or $O(1)$	$O(n)$	$O(n)$	$O(n)$
Array	$O(n)$ or $O(1)$	$O(n)$ or $O(1)$	$O(n)$	$O(n)$	$O(n)$
Sorted array	$O(n)$	$O(n)$	$O(\log n)$	$O(1)$	$O(\log n)$ or $O(1)$

Today: We design datastructure that does all in an logarithm.

Binary search trees

Binary search trees

Definition (Binary tree)

Binary tree is:

1. a rooted tree where
2. every vertex has at most 2 sons and
3. we where distinguish left and right son of every vertex

Binary search trees

Definition (Binary tree)

Binary tree is:

1. a rooted tree where
2. every vertex has at most 2 sons and
3. we where distinguish left and right son of every vertex

Notation: for a vertex v in a binary tree we denote by

$l(v)$ and $r(v)$ the left and right son of v,

$p(v)$ the parent of v.

$T(v)$ the subtree rooted in v,

$L(v)$ and $R(v)$ the subtree rooted in left and right son of v,

$h(v)$ the height of $T(v)$.

Binary search trees

Definition (Binary tree)

Binary tree is:

1. a rooted tree where
2. every vertex has at most 2 sons and
3. we where distinguish left and right son of every vertex

Notation: for a vertex v in a binary tree we denote by

$l(v)$ and $r(v)$ the left and right son of v,
$p(v)$ the parent of v.
$T(v)$ the subtree rooted in v,
$L(v)$ and $R(v)$ the subtree rooted in left and right son of v,
$h(v)$ the height of $T(v)$.

Definition (Binary search tree)

Binary search tree is a binary tree where every vertex v has unique key $k(v)$ and for every vertex v it holds:

1. $\forall_{x \in L(v)} : x < v$ and
2. $\forall_{y \in R(v)} : y > v$.

Operations on binary search trees

Show(v): Print all values in a tree with root v

1. If $v = \emptyset$: return
2. Show ($l(v)$)
3. Print v
4. Show ($r(v)$)

Operations on binary search trees

Show(v): Print all values in a tree with root v

1. If $v = \emptyset$: return
2. Show ($l(v)$)
3. Print v
4. Show ($r(v)$)

Find(v,x): Find key x in a tree with root v

1. If $v = \emptyset$: return $\emptyset$
2. If $x = k(v)$: return v
3. If $x < k(v)$: return Find($l(v)$,x)
4. If $x > k(v)$: return Find($r(v)$,x)

Operations on binary search trees

Show(v): Print all values in a tree with root v

1. If $v = \emptyset$: return
2. Show ($l(v)$)
3. Print v
4. Show ($r(v)$)

Min(v): Return minimum of a tree with root v

1. If $v = \emptyset$: return $\emptyset$
2. If $l(v) = \emptyset$: return v
3. Return Min($l(v)$)

Find(v,x): Find key x in a tree with root v

1. If $v = \emptyset$: return $\emptyset$
2. If $x = k(v)$: return v
3. If $x < k(v)$: return Find($l(v)$,x)
4. If $x > k(v)$: return Find($r(v)$,x)

Operations on binary search trees

Show(v): Print all values in a tree with root v

1. If $v = \emptyset$: return
2. Show ($l(v)$)
3. Print v
4. Show ($r(v)$)

Min(v): Return minimum of a tree with root v

1. If $v = \emptyset$: return $\emptyset$
2. If $l(v) = \emptyset$: return v
3. Return Min($l(v)$)

Find(v,x): Find key x in a tree with root v

1. If $v = \emptyset$: return $\emptyset$
2. If $x = k(v)$: return v
3. If $x < k(v)$: return Find($l(v)$,x)
4. If $x > k(v)$: return Find($r(v)$,x)

Insert(v,x): Insert x to a tree with root v

1. If $v = \emptyset$: create new vertex v with key x and return it
2. If $x < k(v)$: $l(v) \leftarrow$ Insert ($l(v)$,x)
3. If $x > k(v)$: $r(v) \leftarrow$ Insert ($r(v)$,x)
4. If $x = k(v)$: then x already exists in the tree and there is nothing to do.

Operations on binary search trees

Show(v): Print all values in a tree with root v

1. If $v = \emptyset$: return
2. Show ($l(v)$)
3. Print v
4. Show ($r(v)$)

Find(v,x): Find key x in a tree with root v

1. If $v = \emptyset$: return $\emptyset$
2. If $x = k(v)$: return v
3. If $x < k(v)$: return Find($l(v)$,x)
4. If $x > k(v)$: return Find($r(v)$,x)

Min(v): Return minimum of a tree with root v

1. If $v = \emptyset$: return $\emptyset$
2. If $l(v) = \emptyset$: return v
3. Return Min($l(v)$)

Insert(v,x): Insert x to a tree with root v

1. If $v = \emptyset$: create new vertex v with key x and return it
2. If $x < k(v)$: $l(v) \leftarrow$ Insert ($l(v)$,x)
3. If $x > k(v)$: $r(v) \leftarrow$ Insert ($r(v)$,x)
4. If $x = k(v)$: then x already exists in the tree and there is nothing to do.

Homework: Figure out implementation of SUCC and PRED

Delete in binary search tree

Delete(v,x): Insert x to a tree with root v

1. If $v = \emptyset$: return $\emptyset$
2. If $x < k(v)$: $l(v)$ ←Delete($l(v)$,x)
3. If $x > k(v)$: $r(v)$ ←Delete($r(v)$,x)
4. If $x = k(v)$:
5. If $l(v) = r(v) = \emptyset$: return $\emptyset$

Delete in binary search tree

Delete(v,x): Insert x to a tree with root v

1. If $v = \emptyset$: return $\emptyset$
2. If $x < k(v)$: $l(v)$ $\leftarrow$Delete($l(v)$,x)
3. If $x > k(v)$: $r(v)$ $\leftarrow$Delete($r(v)$,x)
4. If $x = k(v)$:
5. If $l(v) = r(v) = \emptyset$: return $\emptyset$
6. If $l(v) = \emptyset$: return $r(v)$

Delete in binary search tree

Delete(v,x): Insert x to a tree with root v

1. If $v = \emptyset$: return $\emptyset$
2. If $x < k(v)$: $l(v) \leftarrow$Delete($l(v)$,x)
3. If $x > k(v)$: $r(v) \leftarrow$Delete($r(v)$,x)
4. If $x = k(v)$:
5. If $l(v) = r(v) = \emptyset$: return $\emptyset$
6. If $l(v) = \emptyset$: return $r(v)$
7. If $r(v) = \emptyset$: return $l(v)$

Delete in binary search tree

Delete(v,x): Insert x to a tree with root v

1. If $v = \emptyset$: return $\emptyset$
2. If $x < k(v)$: $l(v) \leftarrow$Delete($l(v)$,x)
3. If $x > k(v)$: $r(v) \leftarrow$Delete($r(v)$,x)
4. If $x = k(v)$:
5.   If $l(v) = r(v) = \emptyset$: return $\emptyset$
6.   If $l(v) = \emptyset$: return $r(v)$
7.   If $r(v) = \emptyset$: return $l(v)$
8. $s \leftarrow$Min(v)
9. $k(v) \leftarrow k(s)$
10. $r(v) \leftarrow$Delete($r(v)$,s)

Time complexity

Theorem

Operations Insert, Delete, Find, Min, Max, Succ *and* Pred *on binary search tree runs in time* $O(h)$ *where* h *is a height of the tree.*

Sadly the height of a binary search tree can be n.

Time complexity

Theorem

Operations INSERT, DELETE, FIND, MIN, MAX, SUCC *and* PRED *on binary search tree runs in time* $O(h)$ *where* h *is a height of the tree.*

Definition (Perfectly ballanced tree)

Binary search tree is perfectly balanced if $\forall_v : \big| |L(v)| - |R(v)| \big| \leq 1$.

Depth of perfectly balanced tree is $\lfloor \log n \rfloor$.

Time complexity

Theorem

Operations INSERT*,* DELETE*,* FIND*,* MIN*,* MAX*,* SUCC *and* PRED *on binary search tree runs in time $O(h)$ where h is a height of the tree.*

Definition (Perfectly ballanced tree)

Binary search tree is perfectly balanced if $\forall_v : \big| |L(v)| - |R(v)| \big| \le 1$.

Depth of perfectly balanced tree is $\lfloor \log n \rfloor$.

Theorem

The time complexity of insert on perfectly balanced tree is $\Omega(n)$.

Put $n = 2^k - 1$ and then perform Insert(1), Insert (2),. . . , Insert(n).
Continue by Delete(1), Insert($n+1$), Delete(2), Insert($n+2$), . . .

AVL-trees (1962)

Georgy Adelson-Velsky

Evgenii Landis

Definition (AVL tree)

Binary search tree is height balanced (or AVL-tree) if

$$\forall_v : \left| h(l(v)) - h(r(v)) \right| \leq 1.$$

AVL-trees (1962)

Georgy Adelson-Velsky

Evgenii Landis

Definition (AVL tree)

Binary search tree is height balanced (or AVL-tree) if

$$\forall_v : \left| h(l(v)) - h(r(v)) \right| \leq 1.$$

Lemma

Every AVL-tree with n vertices has depth $\Theta(\log n)$

AVL-trees (1962)

Georgy Adelson-Velsky

Evgenii Landis

Definition (AVL tree)

Binary search tree is height balanced (or AVL-tree) if

$$\forall_v : \left|h(l(v)) - h(r(v))\right| \leq 1.$$

Lemma

Every AVL-tree with n vertices has depth $\Theta(\log n)$

Proof.

Denote by A_n the minimal number of vertices of an AVL-tree.
Show that $A_0 = 0$, $A_1 = 1$, $A_n = A_{n-1} + A_{n-2} + 1$

AVL-trees (1962)

Georgy Adelson-Velsky

Evgenii Landis

Definition (AVL tree)

Binary search tree is height balanced (or AVL-tree) if

$$\forall_v : \left|h(l(v)) - h(r(v))\right| \leq 1.$$

Lemma

Every AVL-tree with n vertices has depth $\Theta(\log n)$

Proof.

Denote by A_n the minimal number of vertices of an AVL-tree.
Show that $A_0 = 0$, $A_1 = 1$, $A_n = A_{n-1} + A_{n-2} + 1$
Observe $A_n \geq 2^{\frac{n}{2}}$:

$$A_n = A_{n-1} + A_{n-2} + 1 \geq 2^{\frac{n-1}{2}} + 2^{\frac{n-2}{2}}$$

AVL-trees (1962)

Georgy Adelson-Velsky

Evgenii Landis

Definition (AVL tree)

Binary search tree is height balanced (or AVL-tree) if

$$\forall_v : \left|h(l(v)) - h(r(v))\right| \le 1.$$

Lemma

Every AVL-tree with n vertices has depth $\Theta(\log n)$

Proof.

Denote by A_n the minimal number of vertices of an AVL-tree.
Show that $A_0 = 0$, $A_1 = 1$, $A_n = A_{n-1} + A_{n-2} + 1$
Observe $A_n \ge 2^{\frac{n}{2}}$:

$$A_n = A_{n-1} + A_{n-2} + 1 \ge 2^{\frac{n-1}{2}} + 2^{\frac{n-2}{2}} = 2^{\frac{n}{2}} \left(2^{-\frac{1}{2}} + 2^{-1}\right)$$

AVL-trees (1962)

Georgy Adelson-Velsky

Evgenii Landis

Definition (AVL tree)

Binary search tree is height balanced (or AVL-tree) if

$$\forall_v : \left|h(l(v)) - h(r(v))\right| \leq 1.$$

Lemma

Every AVL-tree with n vertices has depth $\Theta(\log n)$

Proof.

Denote by A_n the minimal number of vertices of an AVL-tree.
Show that $A_0 = 0$, $A_1 = 1$, $A_n = A_{n-1} + A_{n-2} + 1$
Observe $A_n \geq 2^{\frac{n}{2}}$:

$$A_n = A_{n-1} + A_{n-2} + 1 \geq 2^{\frac{n-1}{2}} + 2^{\frac{n-2}{2}} = 2^{\frac{n}{2}} \left(2^{-\frac{1}{2}} + 2^{-1}\right) > 2^n(0.707 + 0.5) > 2^{\frac{n}{2}}.$$

□

Insert operation

Remember for every vertex a sign $\delta(v) = h(l(v)) - h(r(v))$

Insert(v,x)

1. Insert element to a binary search tree
2. Re-balance the tree

Insert case — —

Insert case −+