Algorithms and datastructures I
Lecture 8: self balancing trees

Jan Hubička

Department of Applied Mathematics
Charles University
Prague

March 24 2020

Set datastructure

We would like to represent a set (or a dictionary) of some elements from an universum.
We expect that elements of the 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
- 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 $O(1)$

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

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 height $\Theta(\log n)$

Insert operation

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

Insert(v,x)

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

Given vertex x we need to to solve the situation where its son s increase height by 1. Assume that y is a left son (for right son the situation is symmetric). Consider three cases:

1. $\delta(x) = +$ (right subtree is higher):

Insert operation

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

Insert(v,x)

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

Given vertex x we need to to solve the situation where its son s increase height by 1. Assume that y is a left son (for right son the situation is symmetric). Consider three cases:

1. $\delta(x) = +$ (right subtree is higher):
   Put $\delta(x) = 0$. We are finished: result is AVL-tree again.

Insert operation

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

Insert(v,x)

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

Given vertex x we need to to solve the situation where its son s increase height by 1. Assume that y is a left son (for right son the situation is symmetric). Consider three cases:

1. $\delta(x) = +$ (right subtree is higher):
   Put $\delta(x) = 0$. We are finished: result is AVL-tree again.
2. $\delta(x) = 0$ (both subtrees are having same height):

Insert operation

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

Insert(v,x)

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

Given vertex x we need to to solve the situation where its son s increase height by 1. Assume that y is a left son (for right son the situation is symmetric). Consider three cases:

1. $\delta(x) = +$ (right subtree is higher):
   Put $\delta(x) = 0$. We are finished: result is AVL-tree again.
2. $\delta(x) = 0$ (both subtrees are having same height):
   Put $\delta(x) = +$ and recursively rebalance in $p(x)$ (subtree of x just got higher).

Insert operation

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

Insert(v,x)

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

Given vertex x we need to to solve the situation where its son s increase height by 1. Assume that y is a left son (for right son the situation is symmetric). Consider three cases:

1. $\delta(x) = +$ (right subtree is higher):
   Put $\delta(x) = 0$. We are finished: result is AVL-tree again.
2. $\delta(x) = 0$ (both subtrees are having same height):
   Put $\delta(x) = +$ and recursively rebalance in $p(x)$ (subtree of x just got higher).
3. $\delta(x) = -$ (left subtree is higher):

Insert operation

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

Insert(v,x)

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

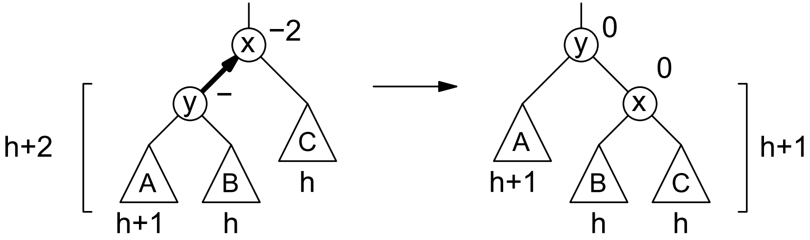
Given vertex x we need to to solve the situation where its son s increase height by 1. Assume that y is a left son (for right son the situation is symmetric). Consider three cases:

1. $\delta(x) = +$ (right subtree is higher):
   Put $\delta(x) = 0$. We are finished: result is AVL-tree again.
2. $\delta(x) = 0$ (both subtrees are having same height):
   Put $\delta(x) = +$ and recursively rebalance in $p(x)$ (subtree of x just got higher).
3. $\delta(x) = -$ (left subtree is higher):
   Subtree of x is not height balanced anymore. We need to use rotations to fix it.

   Look at $\delta(y)$ and consider individual cases:

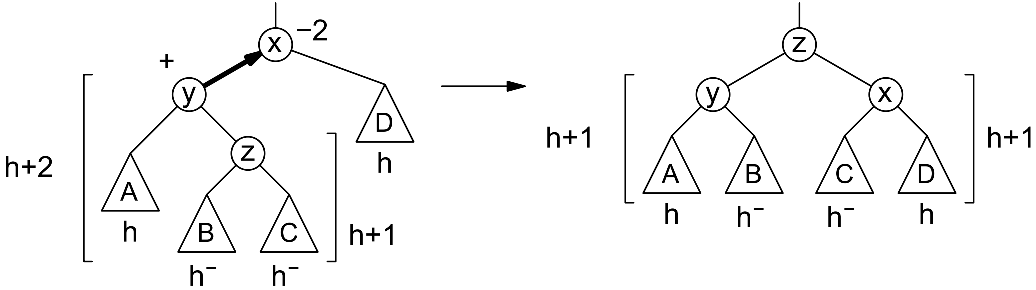
Rebalancing for $\delta(x) = -$ and $\delta(y) = -$

Rebalancing for $\delta(x) = -$ and $\delta(y) = -$



Rebalancing for $\delta(x) = -$ and $\delta(y) = +$

Rebalancing for $\delta(x) = -$ and $\delta(y) = +$



Rebalancing for $\delta(x) = -$ and $\delta(y) = 0$

This case never happens. We only propagate up from vertex with sign $+$ or $-$.

Rebalancing for $\delta(x) = -$ and $\delta(y) = 0$

This case never happens. We only propagate up from vertex with sign $+$ or $-$.

Lemma

Operation INSERT *on AVL-tree can be implemented in* $\Theta(\log n)$ *time.*

Proof.

We know that the height of AVL-tree is $\Theta(\log n)$.
INSERT to binary search tree is done in $\Theta(\log n)$.
Re-balancing may recurse to a father, but number of changes is again limited by the height of tree. $\square$

Rebalancing for $\delta(x) = -$ and $\delta(y) = 0$

This case never happens. We only propagate up from vertex with sign $+$ or $-$.

Lemma

Operation INSERT *on AVL-tree can be implemented in* $\Theta(\log n)$ *time.*

Proof.

We know that the height of AVL-tree is $\Theta(\log n)$.
INSERT to binary search tree is done in $\Theta(\log n)$.
Re-balancing may recurse to a father, but number of changes is again limited by the height of tree. □

See, for example, `https://gist.github.com/Twoody/de8d079842e0dd20cf20d870c73168af`.

Good advice: when implementing AVL-tree write also a verifier that all invariants are maintained correctly.

Delete operation

Delete(v,x)

1. Delete element from a binary search tree
2. Re-balance the tree

Delete operation

Delete(v,x)

1. Delete element from a binary search tree
2. Re-balance the tree

Given vertex x we need to to solve the situation where its son s decreases height by 1. Assume that y is a left son (for right son the situation is symmetric). Consider three cases:

1. $\delta(x) = -$ (left subtree is higher):

Delete operation

Delete(v,x)

1. Delete element from a binary search tree
2. Re-balance the tree

Given vertex x we need to to solve the situation where its son s decreases height by 1. Assume that y is a left son (for right son the situation is symmetric). Consider three cases:

1. $\delta(x) = -$ (left subtree is higher):
   Put $\delta(x) = 0$ and recursively rebalance in $p(x)$ (subtree of x just decreased height)

Delete operation

Delete(v,x)

1. Delete element from a binary search tree
2. Re-balance the tree

Given vertex x we need to to solve the situation where its son s decreases height by 1. Assume that y is a left son (for right son the situation is symmetric). Consider three cases:

1. $\delta(x) = -$ (left subtree is higher):
   Put $\delta(x) = 0$ and recursively rebalance in $p(x)$ (subtree of x just decreased height)
2. $\delta(x) = 0$ (both subtrees are having same heigh):

Delete operation

Delete(v,x)

1. Delete element from a binary search tree
2. Re-balance the tree

Given vertex x we need to to solve the situation where its son s decreases height by 1. Assume that y is a left son (for right son the situation is symmetric). Consider three cases:

1. $\delta(x) = -$ (left subtree is higher):
   Put $\delta(x) = 0$ and recursively rebalance in $p(x)$ (subtree of x just decreased height)
2. $\delta(x) = 0$ (both subtrees are having same heigh):
   Put $\delta(x) = +$. We are finished.

Delete operation

Delete(v,x)

1. Delete element from a binary search tree
2. Re-balance the tree

Given vertex x we need to to solve the situation where its son s decreases height by 1. Assume that y is a left son (for right son the situation is symmetric). Consider three cases:

1. $\delta(x) = -$ (left subtree is higher):
   Put $\delta(x) = 0$ and recursively rebalance in $p(x)$ (subtree of x just decreased height)
2. $\delta(x) = 0$ (both subtrees are having same heigh):
   Put $\delta(x) = +$. We are finished.
3. $\delta(x) = +$ (right subtree is higher):

Delete operation

Delete(v,x)

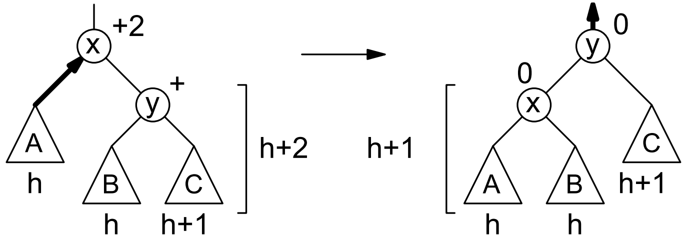
1. Delete element from a binary search tree
2. Re-balance the tree

Given vertex x we need to to solve the situation where its son s decreases height by 1. Assume that y is a left son (for right son the situation is symmetric). Consider three cases:

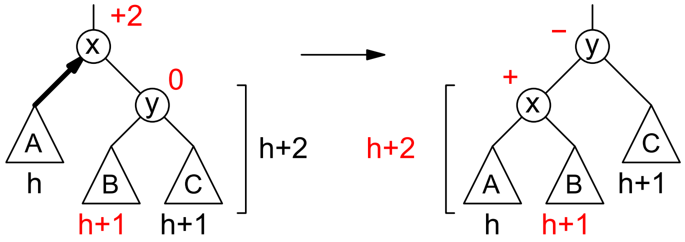
1. $\delta(x) = -$ (left subtree is higher):
   Put $\delta(x) = 0$ and recursively rebalance in $p(x)$ (subtree of x just decreased height)
2. $\delta(x) = 0$ (both subtrees are having same heigh):
   Put $\delta(x) = +$. We are finished.
3. $\delta(x) = +$ (right subtree is higher):
   Subtree of x is not height balanced anymore. We need to use rotations to fix it.
   Look at $\delta(y)$ and consider individual cases

Rebalancing for $\delta(x) = +$ and $\delta(y) = +$

Rebalancing for $\delta(x) = +$ and $\delta(y) = +$

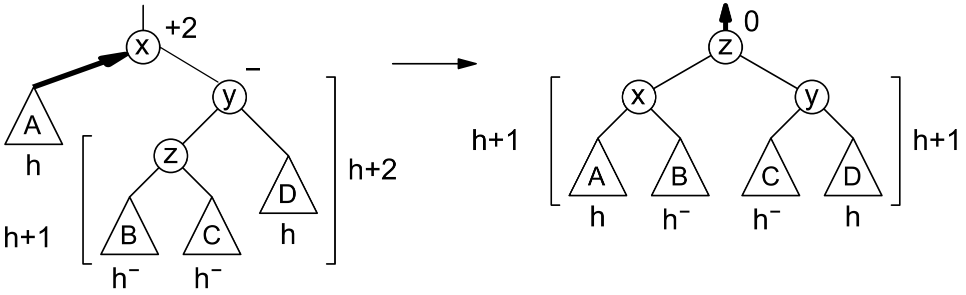


Rebalancing for $\delta(x) = +$ and $\delta(y) = 0$



Rebalancing for $\delta(x) = +$ and $\delta(y) = -$

Rebalancing for $\delta(x) = +$ and $\delta(y) = -$



Theorem

Operations INSERT, DELETE, FIND, MIN, MAX, SUCC, PRED, *on AVL-trees can all be implemented in* $\Theta(\log n)$ *time.*

Theorem

Operations INSERT, DELETE, FIND, MIN, MAX, SUCC, PRED, *on AVL-trees can all be implemented in* $\Theta(\log n)$ *time.*

For INSERT, DELETE, FIND this is best possible.

Theorem

Operations INSERT, DELETE, FIND, MIN, MAX, SUCC, PRED, *on AVL-trees can all be implemented in* $\Theta(\log n)$ *time.*

For INSERT, DELETE, FIND this is best possible.

Theorem

Every datastructure for set which only use comparison on the elements of the universum must implement FIND *in* $\Omega(\log n)$ *time.*

Theorem

Operations INSERT, DELETE, FIND, MIN, MAX, SUCC, PRED, *on AVL-trees can all be implemented in* $\Theta(\log n)$ *time.*

For INSERT, DELETE, FIND this is best possible.

Theorem

Every datastructure for set which only use comparison on the elements of the universum must implement FIND *in* $\Omega(\log n)$ *time.*

Proof.

Assume that set contains n elements. Operation FIND(x) has $n + 1$ possible answers. Every comparison has only 3 possible answers. □

(a, b)-trees (Bayer, McCreight)

AVL-trees do few compares, but use a lot of memory.

(a, b)-trees (Bayer, McCreight)

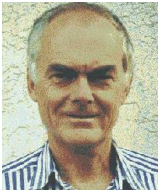
AVL-trees do few compares, but use a lot of memory.
Sorted arrays use less memory, but the INSERT and DELETE operations are slow.

(a, b)-trees (Bayer, McCreight)

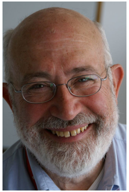
AVL-trees do few compares, but use a lot of memory.
Sorted arrays use less memory, but the Insert and Delete operations are slow.
Can we combine both?

(a, b)-trees (Bayer, McCreight)

AVL-trees do few compares, but use a lot of memory.
Sorted arrays use less memory, but the INSERT and DELETE operations are slow.
Can we combine both?



Rudolf Bayer



Edward M. McCreight

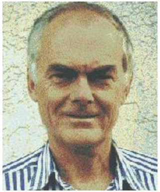
Definition (Generalized search tree)

Generalised search tree is a rooted tree with specified order of sons and two types od vertices:

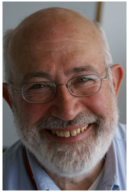
1. Internal vertices contains non-zero number of keys. If internal vertex has keys $x_1 < \cdots < x_n$ then it has $k+1$ sons $s_0, \ldots, s_k$. Keys separate values in sons, so:
   $T(s_0) < x_1 < T(s_1) < x_2 < \cdots < x_{k-1} < T(s_{k-1}) < x_k < T(s_k)$
2. External vertices contain no keys and are leaf.

(a, b)-trees (Bayer, McCreight)

AVL-trees do few compares, but use a lot of memory.
Sorted arrays use less memory, but the INSERT and DELETE operations are slow.
Can we combine both?



Rudolf Bayer



Edward M. McCreight

Definition (Generalized search tree)

Generalised search tree is a rooted tree with specified order of sons and two types od vertices:

1. Internal vertices contains non-zero number of keys. If internal vertex has keys $x_1 < \cdots < x_n$ then it has $k+1$ sons $s_0, \ldots, s_k$. Keys separate values in sons, so:
$T(s_0) < x_1 < T(s_1) < x_2 < \cdots < x_{k-1} < T(s_{k-1}) < x_k < T(s_k)$
2. External vertices contain no keys and are leaf.

Definition ((a, b)-tree)

(a, b)-tree for a given $a \geq 2$, $b \geq 2a - 1$ is a generalised search tree such that:

1. Root has 2 to b sons.
2. Other internal vertices have a to b sons.
3. All external vertices are in the level.

Height of (a, b)-trees

Definition ((a, b)-tree)

(a, b)-tree for given $a \geq 2$, $b \geq 2a - 1$ is generalised search tree such that

1. Root has 2 to b sons.
2. Other internal vertices have a to b sons
3. All external vertices are in the same height

Lemma

Every (a, b)-tree with n keys has depth $\Theta(\log n)$.

Height of (a, b)-trees

Definition ((a, b)-tree)

(a, b)-tree for given $a \geq 2$, $b \geq 2a - 1$ is generalised search tree such that

1. Root has 2 to b sons.
2. Other internal vertices have a to b sons
3. All external vertices are in the same height

Lemma

Every (a, b)-tree with n keys has depth $\Theta(\log n)$.

Proof.

Analyse the minimum number of vertices (a, b)-tree of height h can have. Level 0 has one vertex (root) with at least 2 keys. Level l has at least a times as many keys as level $l - 1$. This grows exponentially fast.

Analogously we can analyse maximum number of vertices. $\square$

Find and insert to (a, b)-tree

Find(v,x)

Find operation can be implemented similarly to one on binary search tree.

1. If v is external vertex return $\emptyset$.
2. Look into keys in v if x is found then return it.
3. If it is not found chose right subtree to recurse into.

Find and insert to (a, b)-tree

Find(v,x)

Find operation can be implemented similarly to one on binary search tree.

1. If v is external vertex return $\emptyset$.
2. Look into keys in v if x is found then return it.
3. If it is not found chose right subtree to recurse into.

Insert(v,x)

Let u be the last internal vertex visited by Find(v,x).

1. If u contains x return.
2. Otherwise add x into u and insert new external vertex
3. If u has more than b sons, split it.

Splitting of a vertex

Splitting of a vertex

Preventive splitting

Useful simplification: If $b \geq 2a$ then we can preventively split every vertex with b sons during the descent to the tree.

Delete from a (a, b)-tree

Delete(v,x)

1. $u \leftarrow$ Find(v,x)
2. If $u = \emptyset$ return

Delete from a (a, b)-tree

Delete(v,x)

1. $u \leftarrow$ Find(v,x)
2. If $u = \emptyset$ return
3. If u is not in the lowest level of the tree:
4. $\quad s \leftarrow$ Succ(u,x)
5. $\quad$ Replace x in u by s
6. $\quad u \leftarrow$ vertex which contains key s.

Delete from a (a, b)-tree

Delete(v,x)

1. $u \leftarrow$ Find(v,x)
2. If $u = \emptyset$ return
3. If u is not in the lowest level of the tree:
4. $s \leftarrow$ Succ(u,x)
5. Replace x in u by s
6. $u \leftarrow$ vertex which contains key s.
7. Remove x from u.

Delete from a (a, b)-tree

Delete(v,x)

1. $u \leftarrow$ Find(v,x)
2. If $u = \emptyset$ return
3. If u is not in the lowest level of the tree:
4. $s \leftarrow$ Succ(u,x)
5. Replace x in u by s
6. $u \leftarrow$ vertex which contains key s.
7. Remove x from u.
8. If u has fewer than a suns see if we can borrow a key from left or right sibling.

Delete from a (a, b)-tree

Delete(v,x)

1. $u \leftarrow$ Find(v,x)
2. If $u = \emptyset$ return
3. If u is not in the lowest level of the tree:
4. $\quad s \leftarrow$ Succ(u,x)
5. $\quad$ Replace x in u by s
6. $\quad u \leftarrow$ vertex which contains key s.
7. Remove x from u.
8. If u has fewer than a suns see if we can borrow a key from left or right sibling.
9. If not merge u with sibling.

Theorem

Operations INSERT, DELETE, FIND, MIN, MAX, SUCC *and* PRED *on* (a, b)*-tree runs in* $\Theta(\log n)$ *time.*