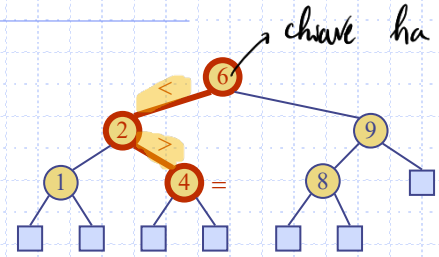
Quando inserisco chiavi e valori, chiavi presenti ordinamento predefinito. La chiave deve essere qualcosa che posso ordinare

Max 2 Nodi

Binary Search Trees



Albero binario di ricerca impone che nodo 6 ha chiave sempre maggiore di tutti i nodi a sinistra e minore di tutti quelli a destra.

1

Se devo inserire una nuova chiave devo trovare il posto preciso dove metterla.

Chiavi hanno ordine ben preciso

Ordered Maps

- Keys are assumed to come from a total order.
- Items are stored in order by their keys
- This allows us to support nearest neighbor queries: Oggetti più vicini a quello che sto cercando? Poi mi sposto
  - Item with largest key less than or equal to k
  - Item with smallest key greater than or equal to k

2

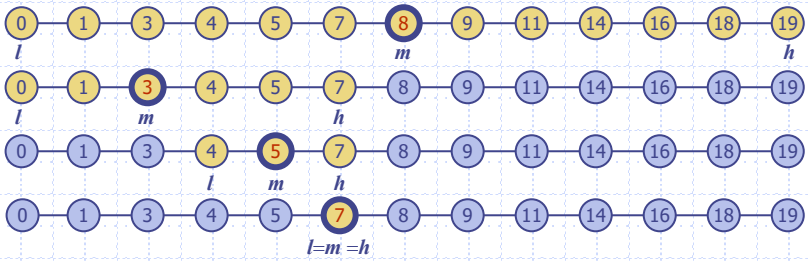
Avremo a che fare con elementi che ordinano le chiavi

Immagina di avere 1000 contatti e cercarne uno. Nel telefono uso una cosa di questo tipo.

Come ottimizzo questa ricerca?

Binary Search

- Binary search can perform nearest neighbor queries on an ordered map that is implemented with an array, sorted by key
  - similar to the high-low children's game
  - at each step, the number of candidate items is halved
  - terminates after O(log n) steps
- Example: find(7)



3

Vettore chiavi

Prendo un elemento. Ciò che cerco si trova a destra o a sinistra? Se a sinistra, non cerco più a destra. Chiave trovata: $m=l=h$

Search Tables

- A search table is an ordered map implemented by means of a sorted sequence
  - We store the items in an array-based sequence, sorted by key
  - We use an external comparator for the keys
- Performance:
  - Searches take $O(\log n)$ time, using binary search
  - Inserting a new item takes $O(n)$ time, since in the worst case we have to shift $n/2$ items to make room for the new item
  - Removing an item takes $O(n)$ time, since in the worst case we have to shift $n/2$ items to compact the items after the removal
- The lookup table is effective only for ordered maps of small size or for maps on which searches are the most common operations, while insertions and removals are rarely performed (e.g., credit card authorizations)

4

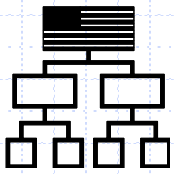
Mi sta bene fino a un certo punto. Tabella di ricerca va bene

Inserimento: Ricerca binaria per vedere dove si dovrebbe trovare elemento e poi spostare tutti elementi a destra o a sinistra per fare spazio. Nel caso peggiore devo spostare $\frac{n}{2}$ elementi. $\Rightarrow O(n)$

Rimozione: Same. Tolgo e risalto, al massimo $\frac{n}{2}$ elementi. $\Rightarrow O(n)$

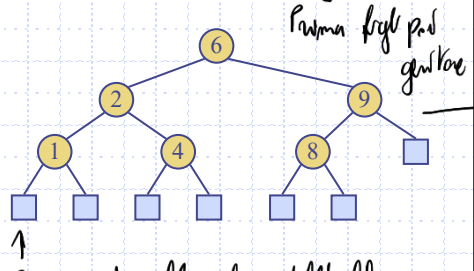
Ma perché se albero binario di ricerca è sempre meglio devo usare questo? Albero binario richiede operazioni estremamente complicate a livello di implementazione

Binary Search Trees



- A binary search tree is a binary tree storing keys (or key-value entries) at its internal nodes and satisfying the following property:
  - Let u, v, and w be three nodes such that u is in the left subtree of v and w is in the right subtree of v. We have $key(u) \leq key(v) \leq key(w)$
- External nodes do not store items
- An inorder traversal of a binary search trees visits the keys in increasing order !

Prima figlio poi genitore



Se arrivato alla fine dell'albero

→ Aggiunta fittizia in modo da rispettare questa proprietà.

5

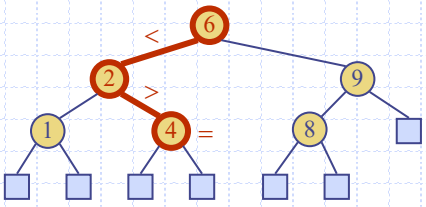
Non ho obbligo su come devo inserire dentro: Posso anche avere albero inverso, ma perdiamo proprietà di ricerca binaria

Search

Più piccolo → Vado a sinistra
Più grande → Vado a destra

- To search for a key k, we trace a downward path starting at the root
- The next node visited depends on the comparison of k with the key of the current node
- If we reach a leaf, the key is not found
- Example: get(4):
  - Call TreeSearch(4,root)
- The algorithms for nearest neighbor queries are similar

```
Algorithm TreeSearch(k, v)
  if T.isExternal (v)
      return v
  if k < key(v)
      return TreeSearch(k, left(v))
  else if k = key(v)
      return v
  else { k > key(v) }
      return TreeSearch(k, right(v))
```

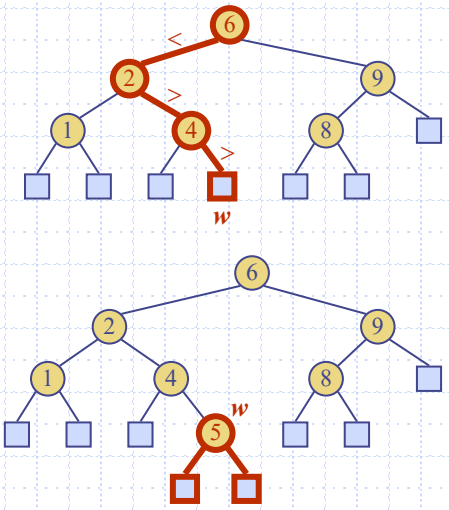


→ Complessità logaritmica!

6

Insertion

- To perform operation put(k, o), we search for key k (using TreeSearch)
- Assume k is not already in the tree, and let w be the leaf reached by the search
- We insert k at node w and expand w into an internal node
- Example: insert 5



7

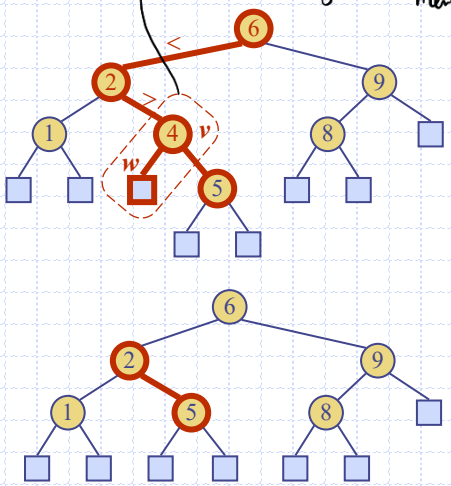
Tempo di inserimento è logaritmico per ora.

⇒ Inserimento è sempre fatto alla frontiera.

Se nodo non ha figli easy!

Deletion

- To perform operation remove(*k*), we search for key *k*
- Assume key *k* is in the tree, and let let *v* be the node storing *k*
- If node *v* has a leaf child *w*, we remove *v* and *w* from the tree with operation removeExternal(*w*), which removes *w* and its parent
- Example: remove 4



8

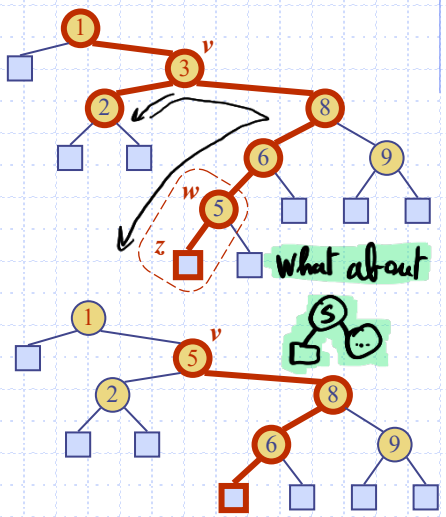
La chiave ha 1 solo figlio e l'altro è un foglia. → Prendo 5 e lo copio su 4.

Ma se ho 2 figli? Nodo 2?

Secondo meccanismo:

Deletion (cont.)

- We consider the case where the key k to be removed is stored at a node v whose children are both internal
  - we find the internal node w that follows v in an inorder traversal
  - we copy $key(w)$ into node v
  - we remove node w and its left child z (which must be a leaf) by means of operation removeExternal(z)
- Example: remove 3

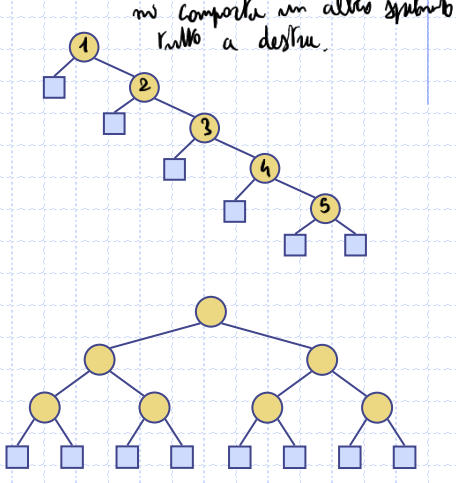


9

Chiave immediatamente successiva in una visita in ordine
⇒ Sono sicuro che mi trovo alla frontiera. Perché non deve avere altri figli.
Scambio nodo trovato con nodo da cancellare e ho fatto la cancellazione → Ordinamento complessivo ordinato

Performance

- Consider an ordered map with n items implemented by means of a binary search tree of height h
  - the space used is $O(n)$
  - methods get, put and remove take $O(h)$ time
- The height h is $O(n)$ in the worst case and $O(\log n)$ in the best case



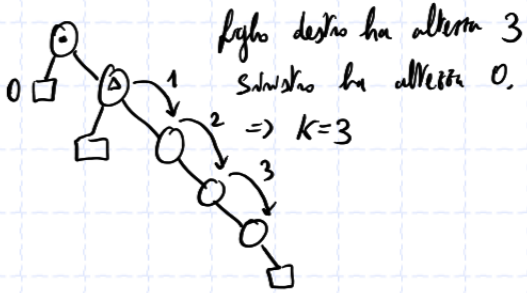
10

Il tempo di ricerca è dato da altezza albero: ricerca diventerebbe ricerca lineare.

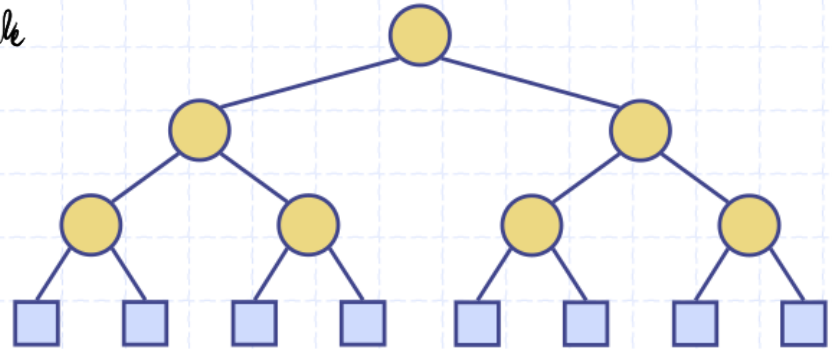
Si cerca di trovare sistema per fare h sia proporzionale a $\log_2 n$.
Questo rispecchierà ulteriore proprietà di ordinamento, ma non proprietà di bilanciamento

Balanced Search Trees

Def: Proprietà di bilanciamento.
Avere differenza tra le altezze dei figli di
ciascun nodo deve essere al più K costante



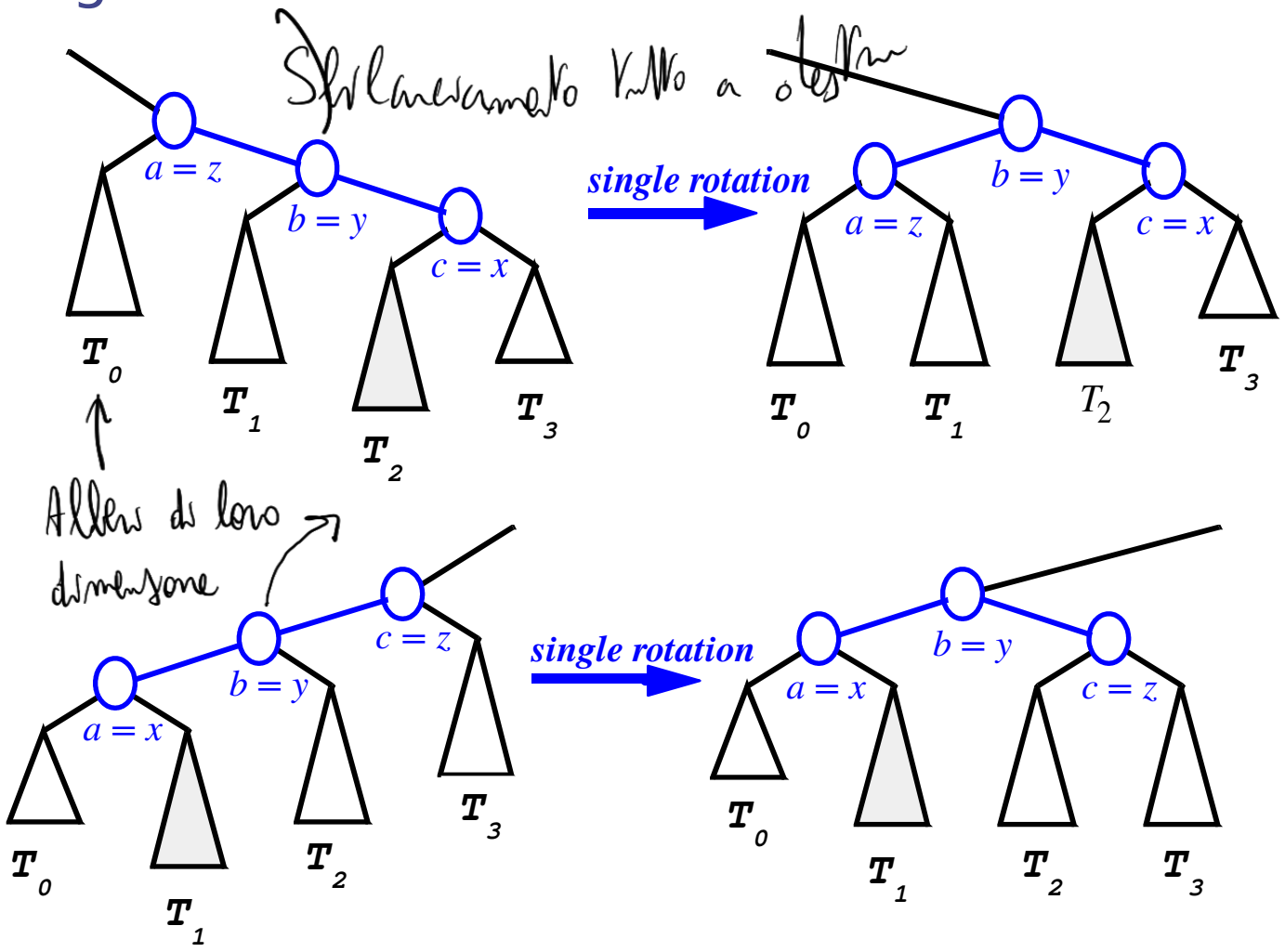
figlio destro ha altezza 3
sinistro ha altezza 0.
=> K=3



Parti da figlio e conta numero di salti da foglie per arrivare
alla frontiera.

Restructuring (as Single Rotations)

- Single Rotations:



Voglio ripristinare il bilanciamento. Come faccio? ROTAZIONI:

ROTAZ. SINGOLA: Sbilanciamento nella chiave figlio sullo stesso ramo.

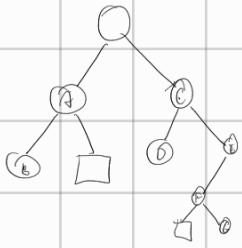
Prendo il nodo più in basso e lo faccio risalire verso l'alto.
Albero nuovo su piano b.

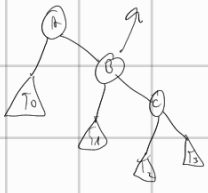
Nodo più in alto come figlio sinistro del nodo centrale (sbilanciamento a destra).
Continuo per sinistra.

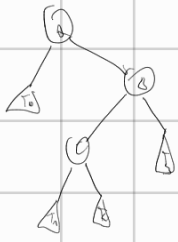
Ma voglio mantenere prop. di ordinamento.
Con quella rotazione, alla fine gli ordini non cambiano.

Ma se albero ora va a zigzag? Altezza di $T_0 = h$, altezza di T_1 sarà almeno $h+1$

Rotaz. doppia = 2 rotaz. Singole. Lo sbilanciamo prima tutto in un verso.

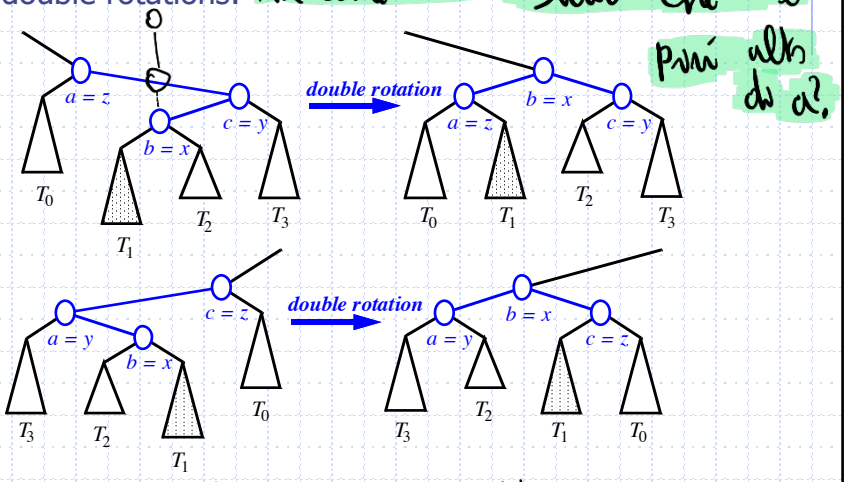






Restructuring (as Double Rotations)

- double rotations: Ma come sono sicuro che b è più alto di a?



Ordine degli alberi sono sempre rispettati.

13

TreeMap

```
public class TreeMap_asd<K,V> {

    //---------------- nested BalanceableBinaryTree class ----------------
    protected static class BalanceableBinaryTree<K, V>
            extends LinkedBinaryTree<Entry<K, V>> {
        //---------------- nested BSTNode class ----------------
        // this extends the inherited LinkedBinaryTree.Node class
        protected static class BSTNode<E> extends Node<E> {
            int aux = 0;

            BSTNode(E e, Node<E> parent, Node<E> leftChild, Node<E> rightChild) {
                super(e, parent, leftChild, rightChild);
            }

            public int getAux() { return aux; }

            public void setAux(int value) { aux = value; }
        } //---------- end of nested BSTNode class ----------

        // positional-based methods related to aux field
        public int getAux(Position<Entry<K, V>> p) { return ((BSTNode<Entry<K, V>>) p).getAux(); }

        public void setAux(Position<Entry<K, V>> p, int value) { ((BSTNode<Entry<K, V>>) p).setAux(value); }

        // Override node factory function to produce a BSTNode (rather than a Node)
        @Override
        protected Node<Entry<K, V>> createNode(Entry<K, V> e, Node<Entry<K, V>> parent,
                                               Node<Entry<K, V>> left, Node<Entry<K, V>> right) {
            return new BSTNode<>(e, parent, left, right);
        }

        /**
         * Relinks a parent node with its oriented child node.
         */
        private void relink(Node<Entry<K, V>> parent, Node<Entry<K, V>> child,
                            boolean makeLeftChild) {
            child.setParent(parent);
            if (makeLeftChild)
                parent.setLeft(child);
            else
                parent.setRight(child);
        }

    }
}
```

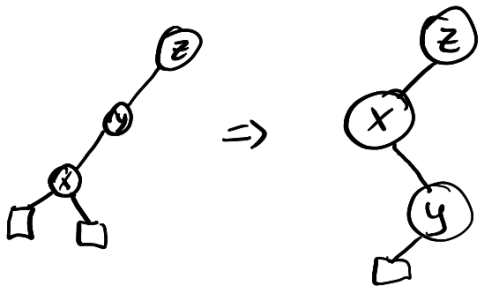
aux?

Devo collegare figlio a genitore superiore

Se figlio destro o sinistro

Con rotazione semplice devo vedere come attaccare figlio

14



Rotation

```
/**
 * Rotates Position p above its parent.  Switches between these
 * configurations, depending on whether p is a or p is b.
 *<pre>
 *          b                 a
 *         / \               / \
 *        a  t2            t0  b
 *       / \                   / \
 *      t0  t1                t1  t2
 *</pre>
 *  Caller should ensure that p is not the root.
 */
public void rotate(Position<Entry<K,V>> p) {
  Node<Entry<K,V>> x = validate(p);
  Node<Entry<K,V>> y = x.getParent();       // we assume this exists
  Node<Entry<K,V>> z = y.getParent();       // grandparent (possibly null)
  if (z == null) {
    root = x;                               // x becomes root of the tree
    x.setParent(null);
  } else
    relink(z, x, makeLeftChild: y == z.getLeft());         // x becomes direct child of z
  // now rotate x and y, including transfer of middle subtree
  if (x == y.getLeft()) {
    relink(y, x.getRight(), makeLeftChild: true);          // x's right child becomes y's left
    relink(x, y, makeLeftChild: false);                    // y becomes x's right child
  } else {
    relink(y, x.getLeft(), makeLeftChild: false);          // x's left child becomes y's right
    relink(x, y, makeLeftChild: true);                     // y becomes left child of x
  }
}
```

Prendo nodo e devo conoscere Padre e nonno.

dammi nodo

DONNO

caso particolare

Rotazione:

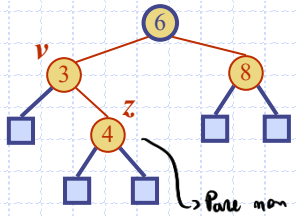
y Primo

15

y deve esistere: Se ho sbilanciamento devo avere per forza un nodo genitore Padre ha h=0.

AVL Trees = Alberi Rudi che proprietà di bilanciamento

$k = 1$

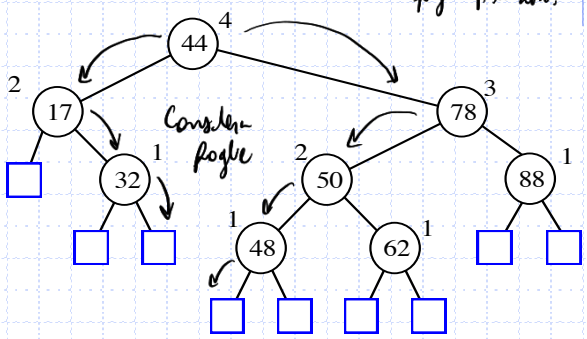


Padre non bilanciato ma $k=1$, Va bene

16

AVL Tree Definition

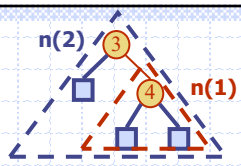
- AVL trees are balanced
- An AVL Tree is a binary search tree such that for every internal node v of T, the heights of the children of v can differ by at most 1



An example of an AVL tree where the heights are shown next to the nodes

17

Height of an AVL Tree



Fact: The height of an AVL tree storing n keys is O(log n).

Proof (by induction): Let us bound n(h): the minimum number of internal nodes of an AVL tree of height h.

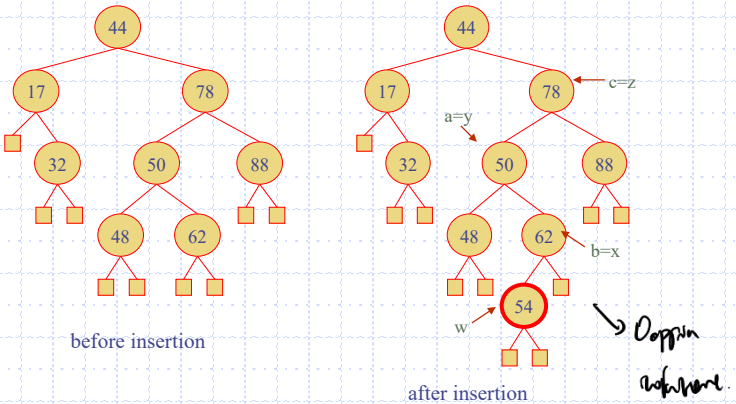
- We easily see that n(1) = 1 and n(2) = 2
- For n > 2, an AVL tree of height h contains the root node, one AVL subtree of height n-1 and another of height n-2.
- That is, n(h) = 1 + n(h-1) + n(h-2)
- Knowing n(h-1) > n(h-2), we get n(h) > 2n(h-2). So n(h) > 2n(h-2), n(h) > 4n(h-4), n(h) > 8n(n-6), … (by induction), n(h) > 2^in(h-2i)
- Solving the base case we get: n(h) > $2^{h/2-1}$
- Taking logarithms: h < 2log n(h) +2
- Thus the height of an AVL tree is O(log n)

18

Qui devo inserire nodi sulla frontiera. Ma dopo inserimento è sbilanciato.

Insertion

- Insertion is as in a binary search tree
- Always done by expanding an external node.
- Example:



Doppia rotazione.

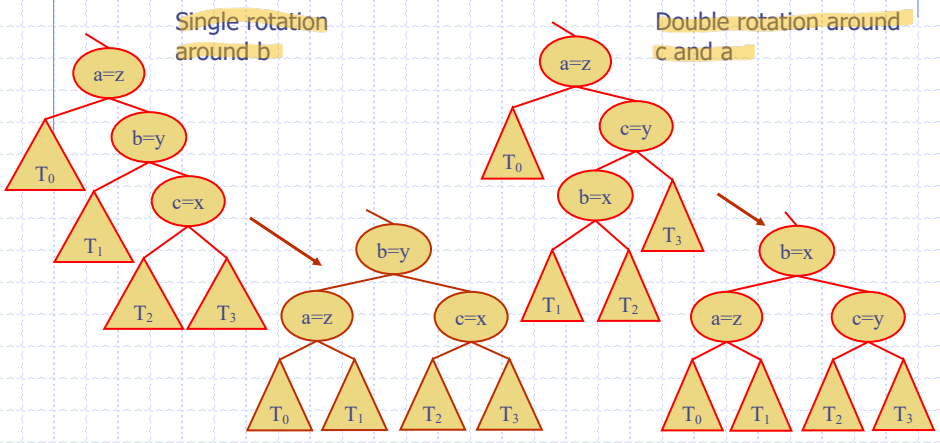
19

Nel ribilanciare 3 nodi potrei sbilanciare nodi superiori.

Devo ricalcolo per il genitore dei nodi le altezze per vedere se ho causato sbilanciamento. Se sì, rifaccio ristrutturazione. Fino alla radice.

Trinode Restructuring → Ristrutturazione trinodo.

- Let (*a*,*b*,*c*) be the inorder listing of *x*, *y*, *z*
- Perform the rotations needed to make *b* the topmost node of the three

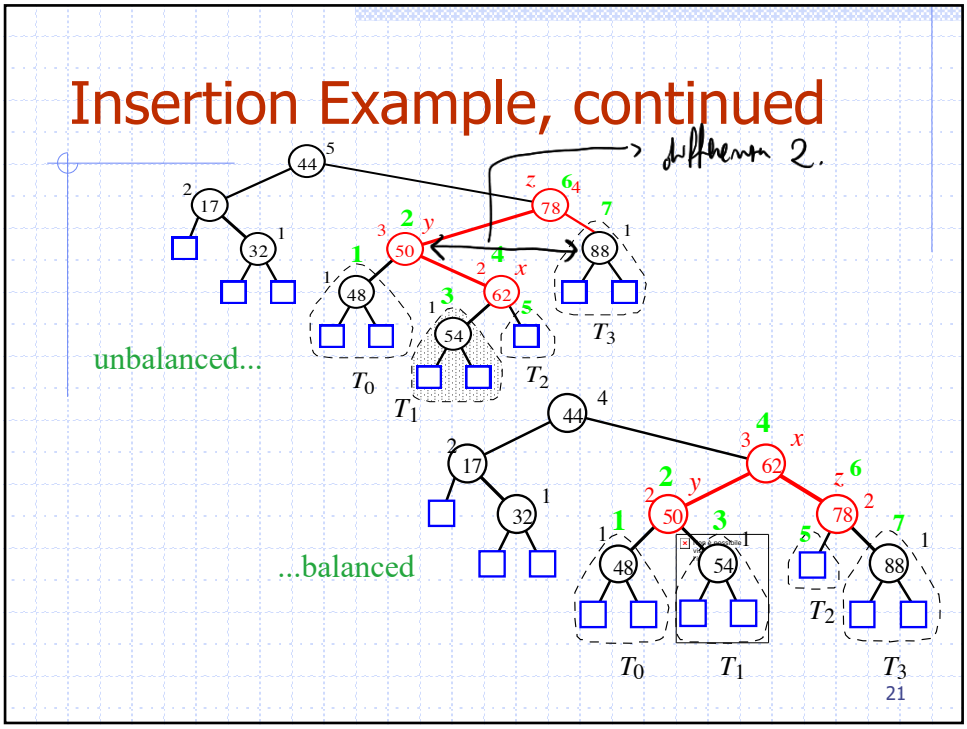


20

Combinaz. di ristrutturaz. singole e doppie su alberi AVL che vengono applicate fino a quello ottenuto dopo aggiungimento di nodo.

Da dove parte la ristrutturazione se aggiungo nodo?

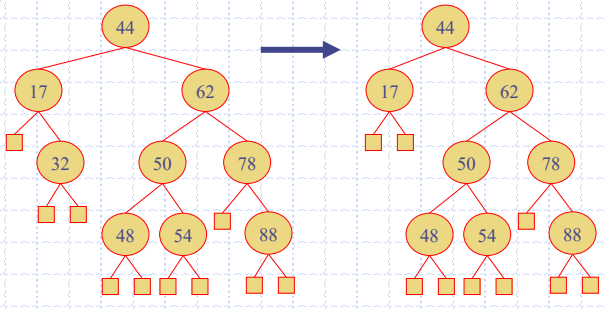
Insertion Example, continued



21

Removal

- Removal begins as in a binary search tree, which means the node removed will become an empty external node. Its parent, w, may cause an imbalance.
- Example:



22

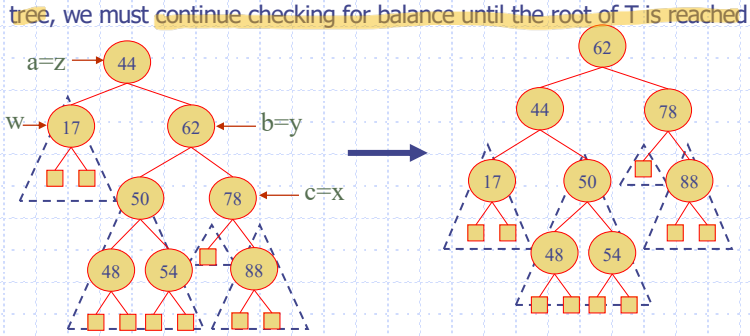
Ci spostiamo su w in che modo?

w da dove è è stabilito?

Ma allora è travelling up o ricorsivo?

Rebalancing after a Removal

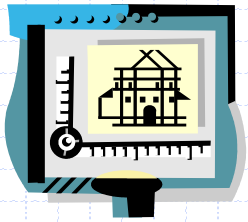
- Let z be the first unbalanced node encountered while travelling up the tree from w. Also, let y be the child of z with the larger height, and let x be the child of y with the larger height
- We perform a trinode restructuring to restore balance at z
- As this restructuring may upset the balance of another node higher in the tree, we must continue checking for balance until the root of T is reached



23

Nodi su cui al massimo devo lavorare → proporz. ad h. ⇒ log n ristrutturazioni.

AVL Tree Performance



- AVL tree storing n items
  - The data structure uses O(n) space
  - A single restructuring takes O(1) time → Ogni volta che faccio ristrutturazione devo fare 3 o 4 operazioni.
    - using a linked-structure binary tree
  - Searching takes O(log n) time
    - height of tree is O(log n), no restructures needed
  - Insertion takes O(log n) time
    - initial find is O(log n)
    - restructuring up the tree, maintaining heights is O(log n)
  - Removal takes O(log n) time
    - initial find is O(log n)
    - restructuring up the tree, maintaining heights is O(log n)

24

Caso peggiore devo ristrutturare tutto l'albero nella sua altezza.
Al più log n perché legato ad altezza.

Implementazione è un casino

Restructuring

```
/**
 *
 * Returns the Position that becomes the root of the restructured subtree.
 *
 * Assumes the nodes are in one of the following configurations:
 *<pre>
 *     z=a                 z=c           z=a               z=c
 *    /  \                /  \          /  \              /  \
 *   t0  y=b             y=b  t3       t0   y=c          y=a  t3
 *      /  \            /  \              /  \          /  \
 *     t1  x=c         x=a  t2           x=b  t3       t0   x=b
 *        /  \        /  \              /  \              /  \
 *       t2  t3      t0  t1            t1  t2            t1  t2
 *</pre>
 * The subtree will be restructured so that the node with key b becomes its root.
 *<pre>
 *          b
 *         /  \
 *        a    c
 *       / \  / \
 *      t0 t1 t2 t3
 *</pre>
 * Caller should ensure that x has a grandparent.
 */
public Position<Entry<K,V>> restructure(Position<Entry<K,V>> x) {
  Position<Entry<K,V>> y = parent(x);
  Position<Entry<K,V>> z = parent(y);
  if ((x == right(y)) == (y == right(z))) {  // matching alignments
    rotate(y);                               // single rotation (of y)
    return y;                                // y is new subtree root
  } else {                                   // opposite alignments
    rotate(x);                               // double rotation (of x)
    rotate(x);
    return x;                                // x is new subtree root
  }
}
```

25

Java Implementation

```
1  /** An implementation of a sorted map using an AVL tree. */
2  public class AVLTreeMap<K,V> extends TreeMap<K,V> {
3    /** Constructs an empty map using the natural ordering of keys. */
4    public AVLTreeMap( ) { super( ); }
5    /** Constructs an empty map using the given comparator to order keys. */
6    public AVLTreeMap(Comparator<K> comp) { super(comp); }
7    /** Returns the height of the given tree position. */
8    protected int height(Position<Entry<K,V>> p) {
9      return tree.getAux(p);
10   }
11   /** Recomputes the height of the given position based on its children's heights. */
12   protected void recomputeHeight(Position<Entry<K,V>> p) {
13     tree.setAux(p, 1 + Math.max(height(left(p)), height(right(p))));
14   }
15   /** Returns whether a position has balance factor between −1 and 1 inclusive. */
16   protected boolean isBalanced(Position<Entry<K,V>> p) {
17     return Math.abs(height(left(p)) − height(right(p))) <= 1;
18   }
```

26

Dove devo ribilanciare?

Java Implementation, 2

```
19   /** Returns a child of p with height no smaller than that of the other child. */
20   protected Position<Entry<K,V>> tallerChild(Position<Entry<K,V>> p) {
21     if (height(left(p)) > height(right(p))) return left(p);      // clear winner
22     if (height(left(p)) < height(right(p))) return right(p);     // clear winner
23     // equal height children; break tie while matching parent's orientation
24     if (isRoot(p)) return left(p);                               // choice is irrelevant
25     if (p == left(parent(p))) return left(p);                    // return aligned child
26     else return right(p);
27   }
```

è il guaio che crea problemi per ribilanciamento

27

Java Implementation, 3

```
33   protected void rebalance(Position<Entry<K,V>> p) {
34     int oldHeight, newHeight;
35     do {
36       oldHeight = height(p);                    // not yet recalculated if internal
37       if (!isBalanced(p)) {                     // imbalance detected
38         // perform trinode restructuring, setting p to resulting root,
39         // and recompute new local heights after the restructuring
40         p = restructure(tallerChild(tallerChild(p)));
41         recomputeHeight(left(p));
42         recomputeHeight(right(p));
43       }
44       recomputeHeight(p);
45       newHeight = height(p);
46       p = parent(p);
47     } while (oldHeight != newHeight && p != null);
48   }
49   /** Overrides the TreeMap rebalancing hook that is called after an insertion. */
50   protected void rebalanceInsert(Position<Entry<K,V>> p) {
51     rebalance(p);
52   }
53   /** Overrides the TreeMap rebalancing hook that is called after a deletion. */
54   protected void rebalanceDelete(Position<Entry<K,V>> p) {
55     if (!isRoot(p))
56       rebalance(parent(p));
57   }
58 }
```

→ p sbilanciato, ribilancio sul più alto.

↳ Per fare ristrutturazione mi serve me, mio padre e mio nonno

Perché?

28