Datastructuren

https://git.lumc.nl/j.k.vis/datastructuren

H.J. Hoogeboom
S. de Gouw
J.K. Vis

Universiteit Leiden

najaar 2018

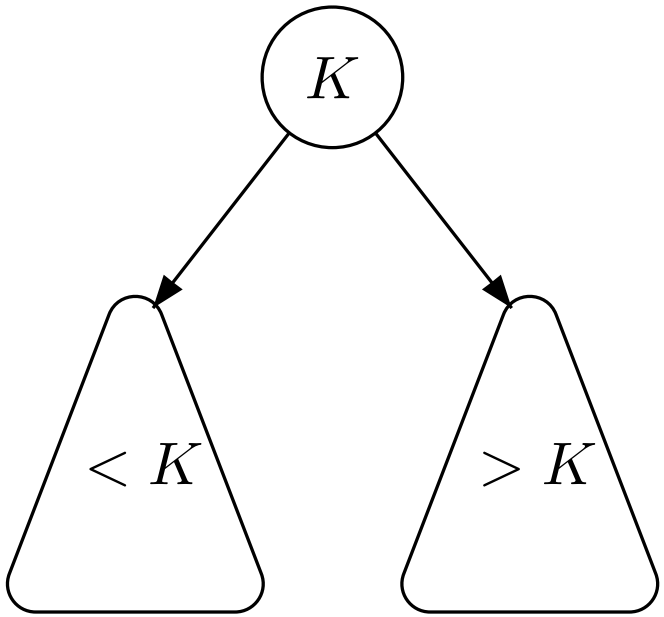
Table of Contents I

1 Binary Search Trees

Contents

1 Binary Search Trees
- Introduction
- BST use cases
- Constructing BSTs
- Analysis of trees
- ADT Set and Dictionary

BST (binary search tree)

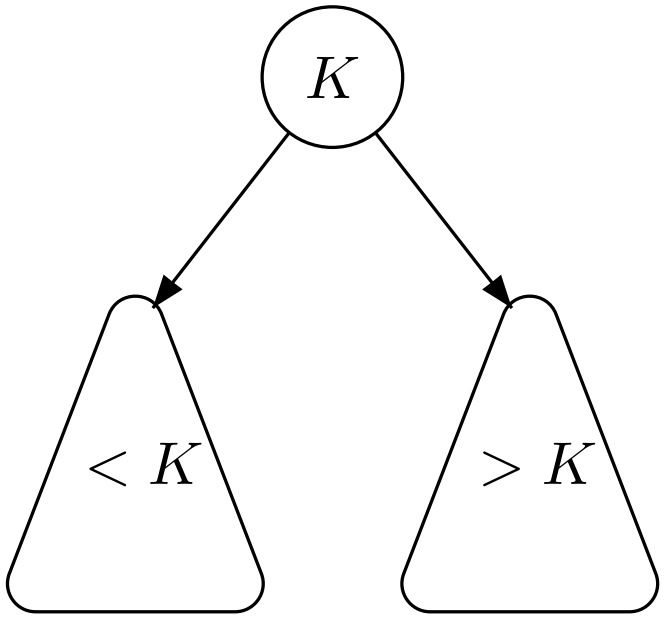


Definition

A binary search tree is a binary tree such that for each node:

- all nodes in its left subtree have smaller values, and
- all nodes in its right subtree have larger values

BST (binary search tree)

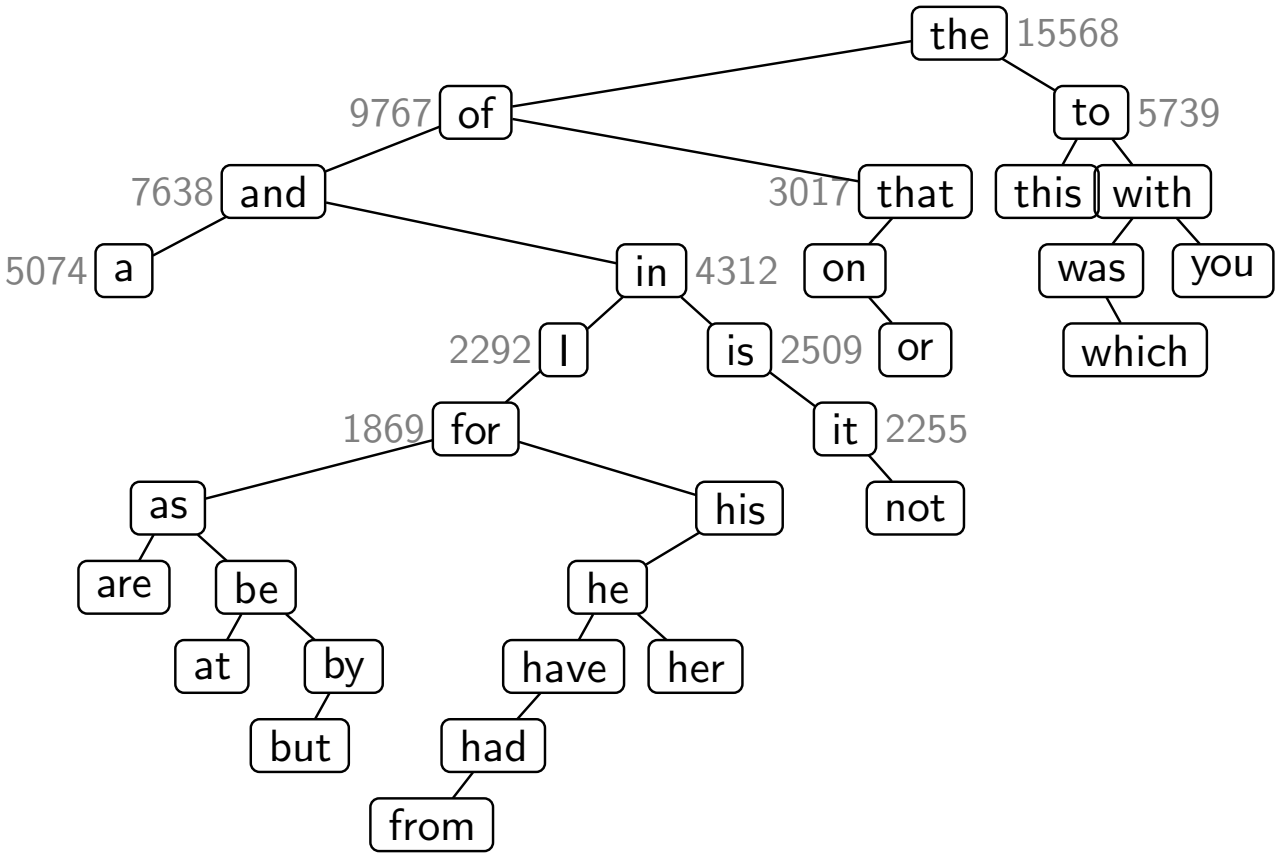


Worst case search complexity: unsuccessful search in

- Worst tree: $O(n)$ (linear tree)
- Best tree: $O(\log_2(n))$ (complete tree)

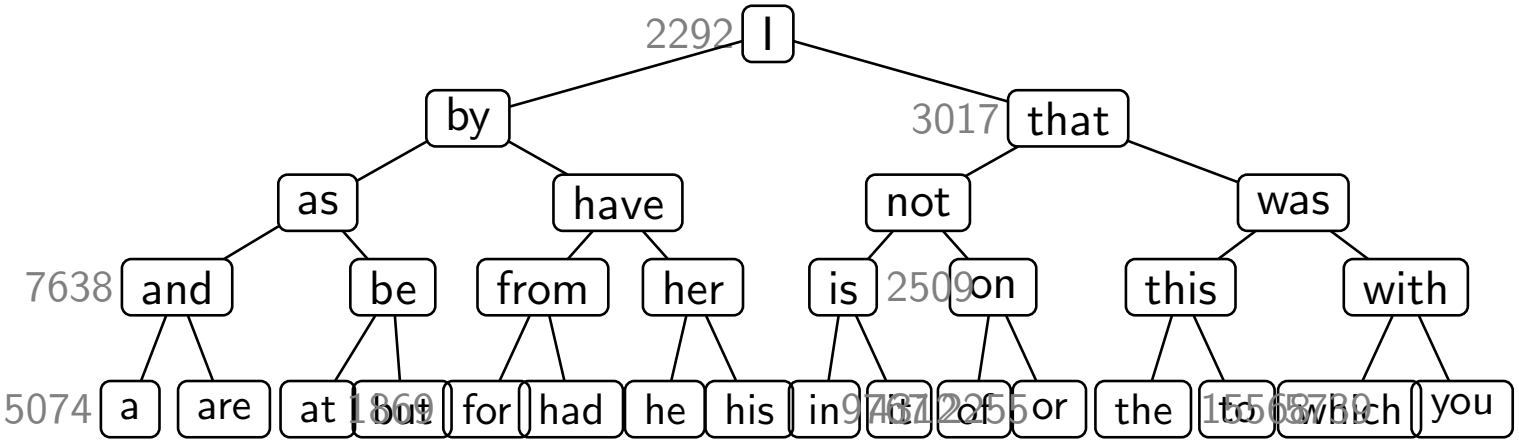
Average case behavior: see later

BST with 31 most common English words



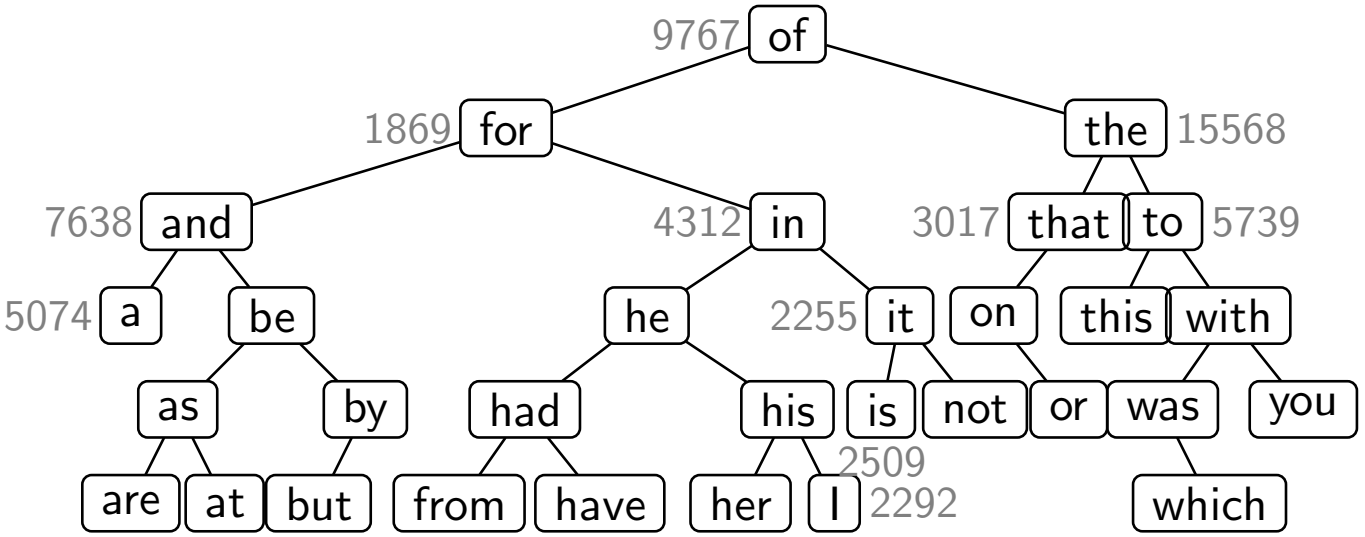
- Words (values) ordered alphabetically
- Inserted in BST by decreasing order of frequency
- Successful search of BST requires 4.042 comparisons (on avg.)

Balanced BST with 31 most common English words



- Perfectly balanced BST
- Successful search requires 4.042 comparisons (on avg.)

Optimal BST with 31 most common English words



- Optimal tree taking frequencies into account
- Successful search requires 3.437 comparisons (on avg.)

Search value

```
bool contains( const Comparable & x, Node *t ) const {
    if( t == nullptr )
        return false;
    else if( x < t->element )
        return contains( x, t->left );
    else if( t->element < x )
        return contains( x, t->right );
    else
        return true; // found
}
```

call with: `contains(v,root);`

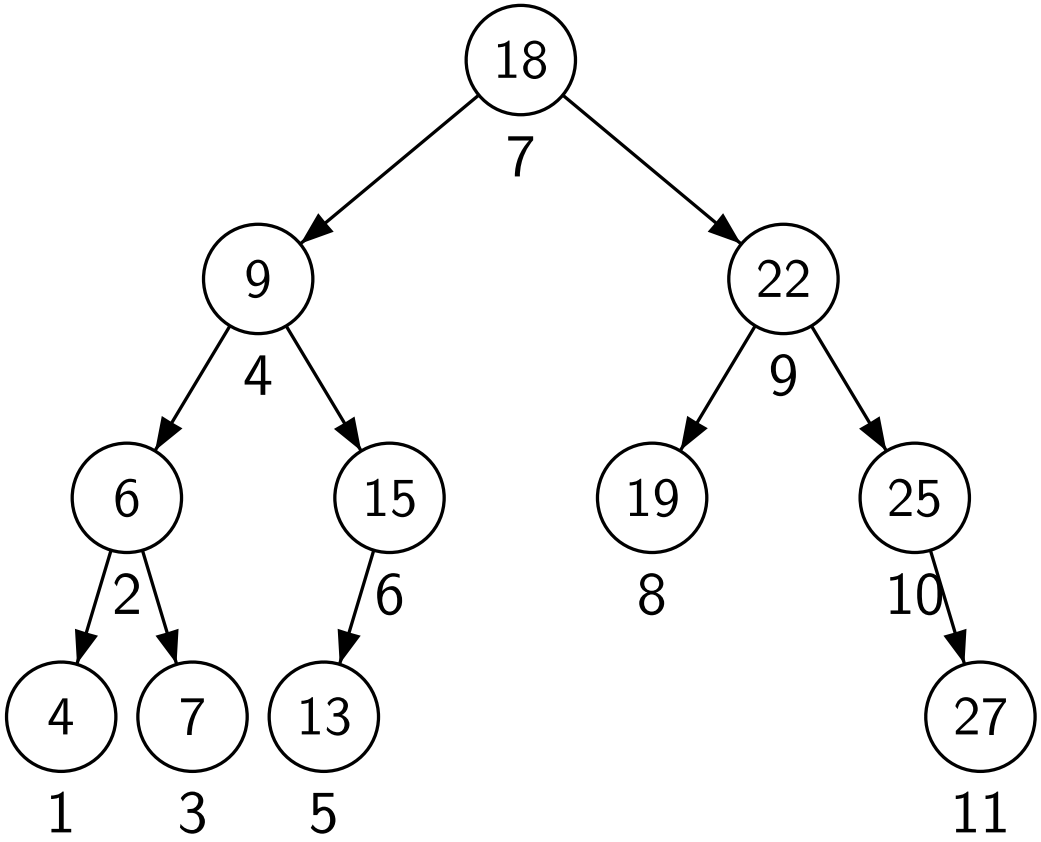
Find min/max value

```
BinaryNode * findMin( BinaryNode *t ) const {
    if( t == nullptr )
        return nullptr;
    if( t->left == nullptr )
        return t;
    return findMin( t->left );
}

BinaryNode * findMax( BinaryNode *t ) const {
    if( t != nullptr )
        while( t->right != nullptr )
            t = t->right;
    return t;
}
```

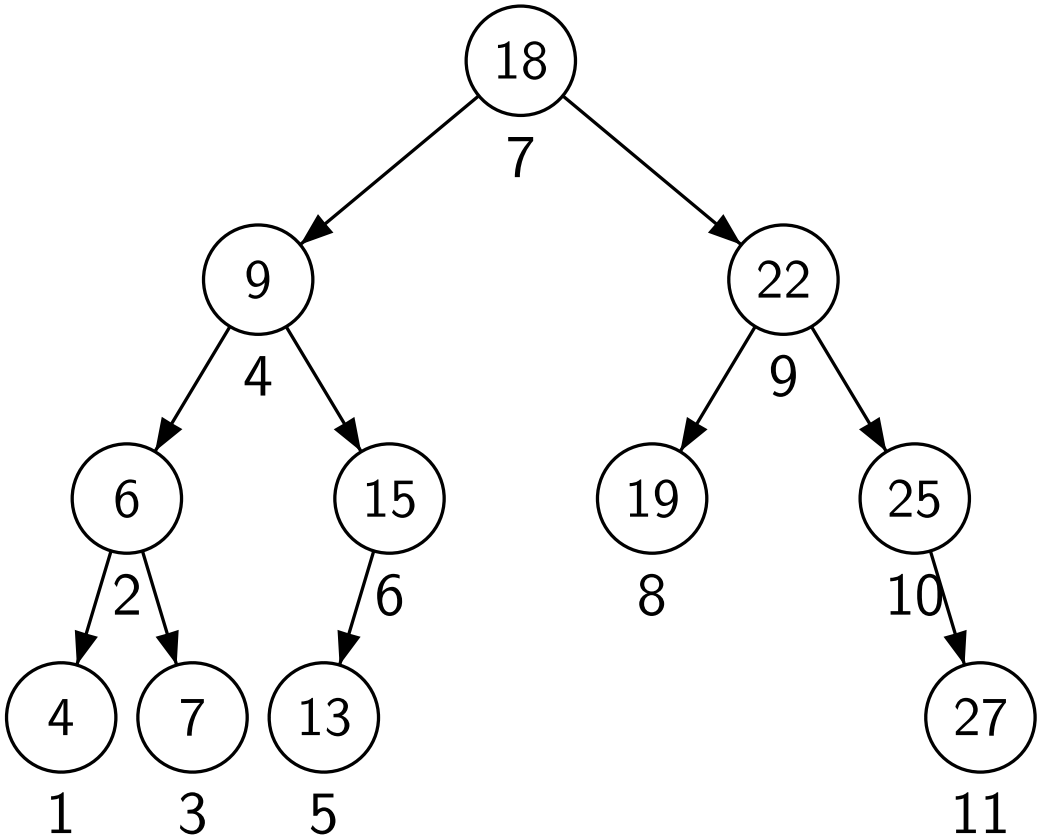
call with: `findMin(root);` and `findMax(root);`

Sorting



Inorder: 4,6,7,9,13,15,18,19,22,25,27

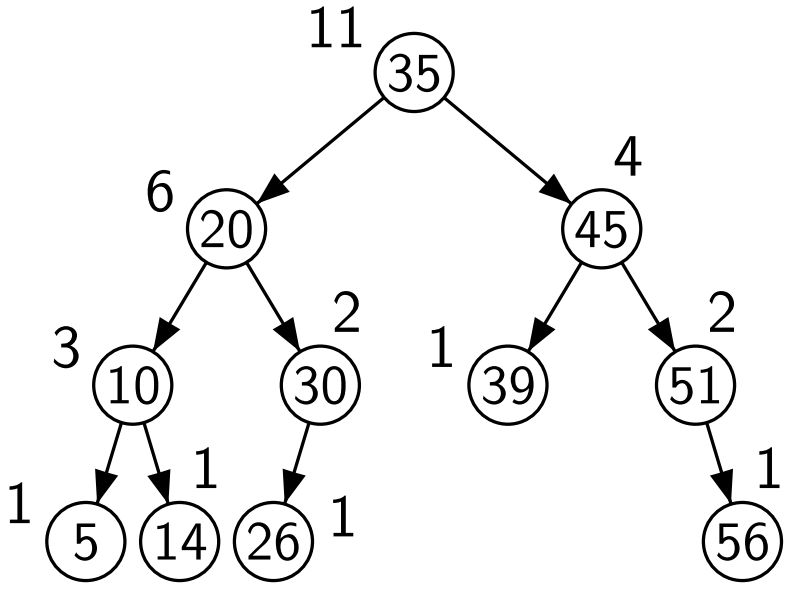
Sorting



Lemma

Inorder traversal of BST yields sorted values

Find k-th element

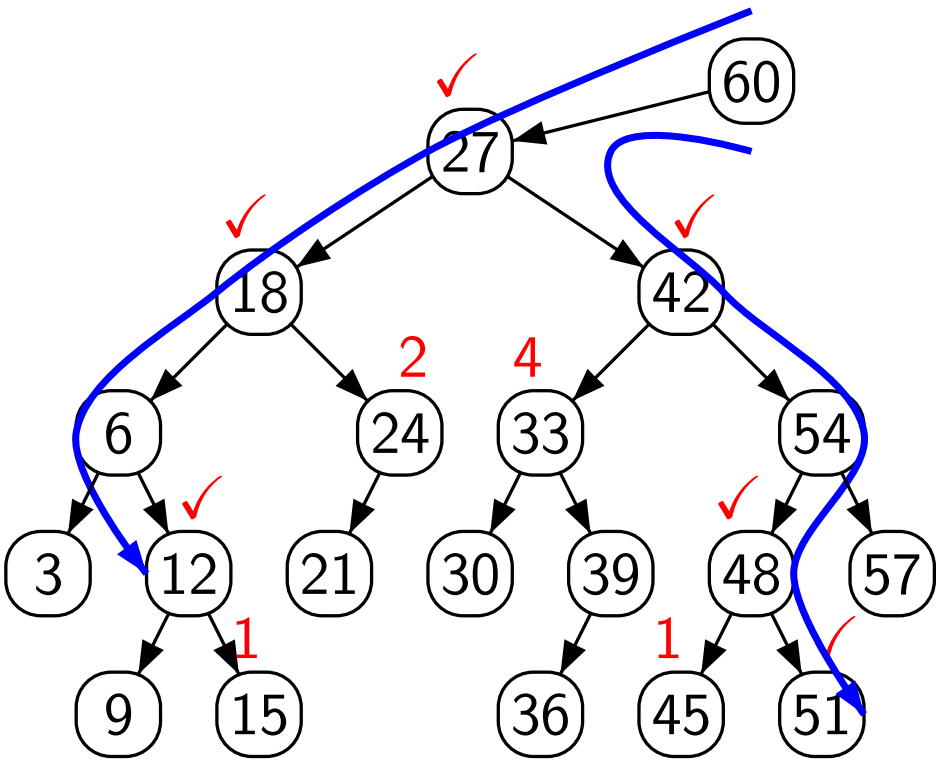


Augment each node with the size of its subtree

- Let r be `left->size + 1`
- If $k = r$: stop! This node has k^{th} item
- If $k < r$: search k^{th} item in left subtree
- If $k > r$: search $(k - r)^{\text{th}}$ item in right subtree

counting items in $[12, 52]$

Annotate nodes with size of its subtree, count items while searching upper (right path) and lower bound (left path)



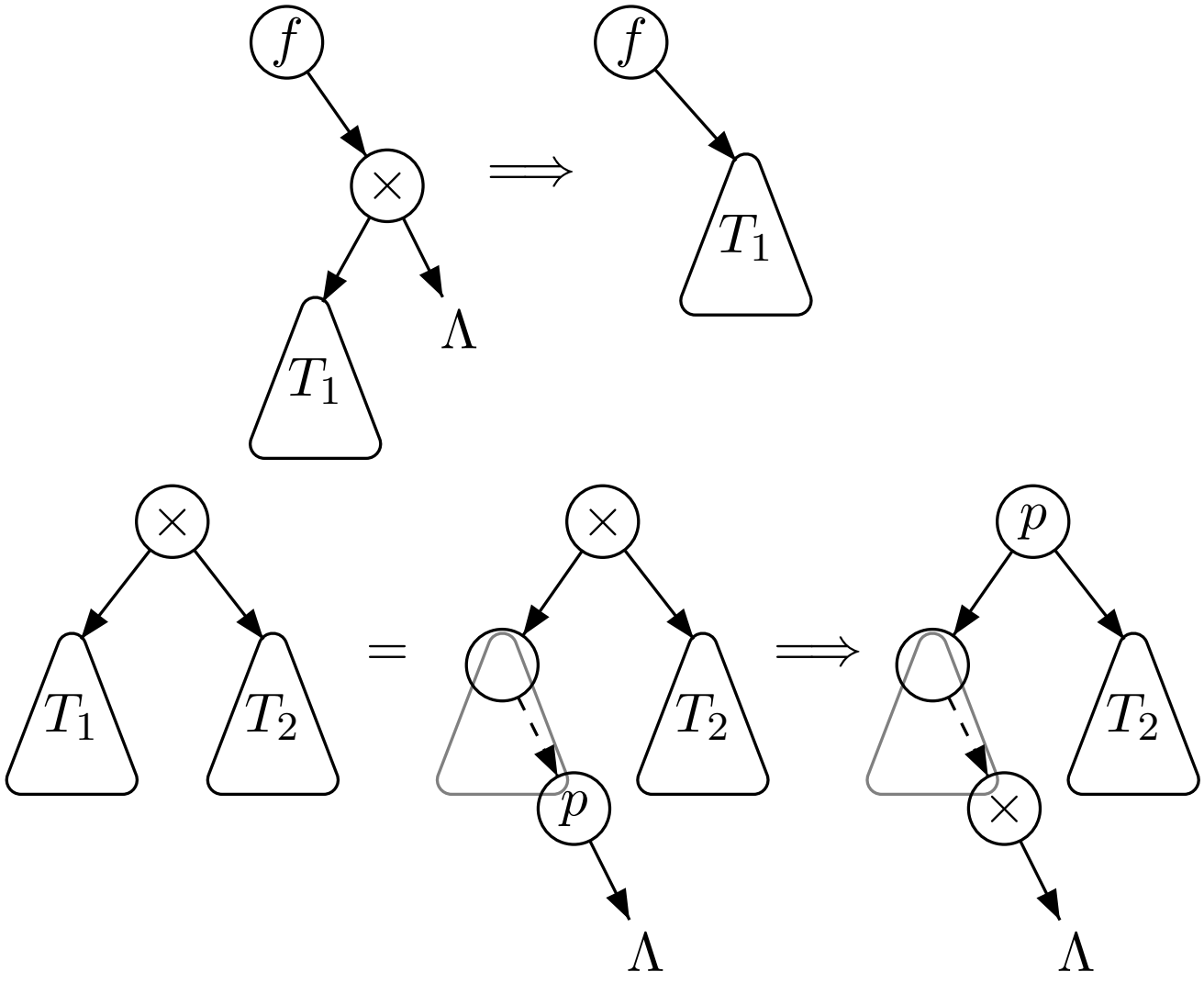
total nodes: split node + #left_subtree + #right_subtree =
$1 + (1+2+0+1+1) + (1+4+0+1+1+1) = 14$

Insertion (implementation)

```
template<class T>
void Node<T>::insert(const T& el, Node<T> * & p) {
    if( p == nullptr ) {
        t = new Node{el, nullptr, nullptr};
    } else if (el < p->data) {
        insert(el, p->left);
    } else if (el > p->data) {
        insert(el, p->right);
    } else {
        ; // Duplicate; do nothing
    }
}
```

call with: `insert(el,root);`

Deletion "by copying"



Deletion (implementation)

```
void remove( const Comparable & x, Node * & t ) {
    if( t == nullptr )      return;
    if( x < t->data )       remove( x, t->left );
    else if( x > t->data) remove( x, t->right );
    else if( t->left != nullptr && t->right != nullptr ) {
        Node *pred = findMax( t->left );
        t->element = pred->element;
        remove( t->element, t->left );
    }
    else {
        BinaryNode *oldNode = t;
        if(t->left != nullptr ) t = t->left
        else                    t = t->right;
        delete oldNode;
    }
```

Counting trees

Unlabeled n-node binary trees

$B_n = \sum_{i=0}^{n-1} B_{i-1} * B_{n-i}$ with $B_0 = 1$

- B_{i-1} left subtrees and B_{n-i} right subtrees
- B_n is n^{th} Catalan number: $B_n = \frac{1}{n+1}\binom{2n}{n} = \frac{(2n)!}{(n+1)!*n!}$

Counting trees

Unlabeled n-node binary trees

$B_n = \sum_{i=0}^{n-1} B_{i-1} * B_{n-i}$ with $B_0 = 1$

- B_{i-1} left subtrees and B_{n-i} right subtrees
- B_n is n^{th} Catalan number: $B_n = \frac{1}{n+1}\binom{2n}{n} = \frac{(2n)!}{(n+1)!*n!}$

Labeled n-node binary trees

$L_n = n! * B_n = \frac{(2n)!}{(n+1)!}$

- Can label nodes in unlabeled tree in $n!$ ways

Counting trees

Unlabeled n-node binary trees

$B_n = \sum_{i=0}^{n-1} B_{i-1} * B_{n-i}$ with $B_0 = 1$

- B_{i-1} left subtrees and B_{n-i} right subtrees
- B_n is n^{th} Catalan number: $B_n = \frac{1}{n+1}\binom{2n}{n} = \frac{(2n)!}{(n+1)!*n!}$

Labeled n-node binary trees

$L_n = n! * B_n = \frac{(2n)!}{(n+1)!}$

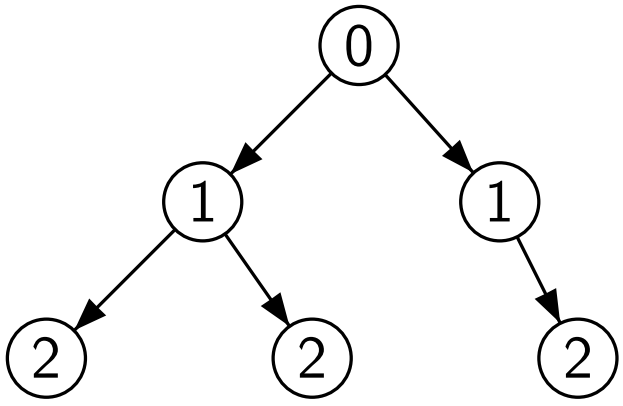
- Can label nodes in unlabeled tree in $n!$ ways

n-node binary search trees

$\text{BST}_n = \sum_{i=0}^{n-1} \text{BST}_{i-1} * \text{BST}_{n-i}$ with $BST_0 = 1$

- Hence: $\text{BST}_n = B_n$

Internal path length



$$\text{ipl} = 0 + 1 + 1 + 2 + 2 + 2 = 8$$

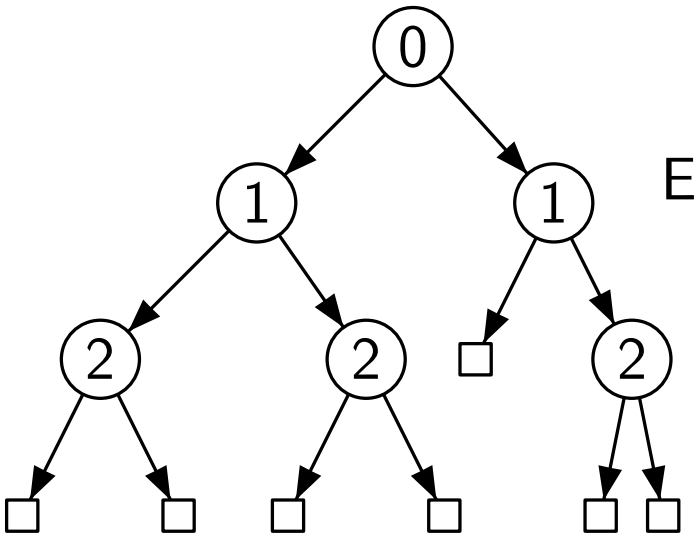
Path length of node: # edges from root to node

Definition (Internal path length)

ipl = sum of all path lengths to all nodes

Avg # comparisons in successful search: $\frac{\text{ipl}}{n} + 1$

External path length



$\mathsf{E} = 3 + 3 + 3 + 3 + 2 + 3 + 3 = 20$

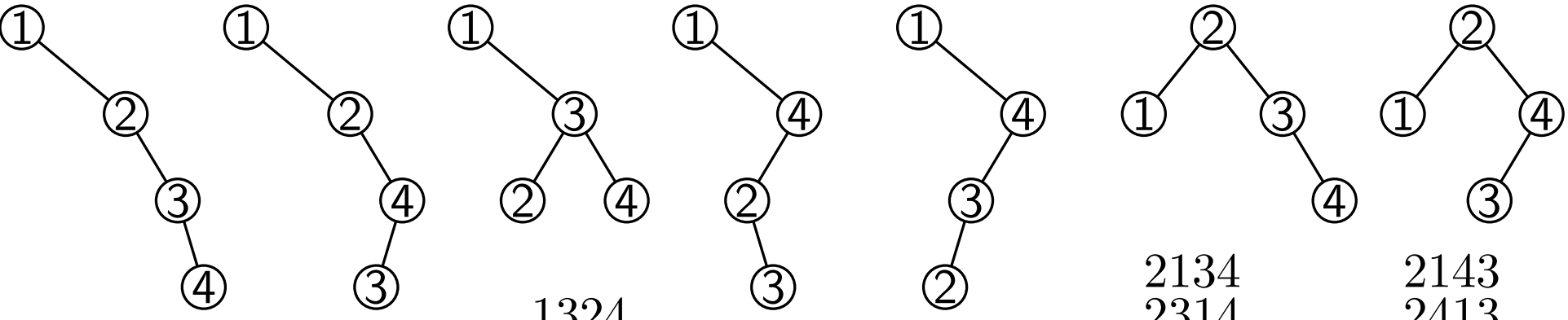
Create full tree by attaching 1 or 2 new leaves to nodes

Definition (External path length)

E = sum of all path lengths to the (new) leaves

Avg # comparisons in *un*successful search: $\frac{E}{n+1}$ ($n+1$ leaves)
Relation to ipl: $E = \text{ipl} + 2n$

Example: 4 node BSTs



4 nodes: 14 BSTs (7 symmetric to above), $4! = 24$ permutations

$$\frac{1}{24}(12 \times 4 + 4 \times 5 + 8 \times 6) = \frac{116}{24} = \frac{29}{6}$$

ADT Set

- INITIALIZE: construct an empty set.
- ISEMPTY: check whether there the set is empty ($\varnothing$, contains no elements).
- SIZE: return the number of elements, the cardinality of the set.
- ISELEMENT(a): returns whether a given object from the domain belongs to the set, $a \in A$.
- INSERT(a): add an element to the set (if it is not present, $A \cup \{a\}$)
- DELETE(a): removes an element from the set (if it is present, $A \setminus \{a\}$).

Efficient implementation of ADT Set possible with BST