Presentation writer: Tigran Hayrapetyan

Lecturer | Programmer | Researcher

www.linkedin.com/in/tigran-hayrapetyan-cs/

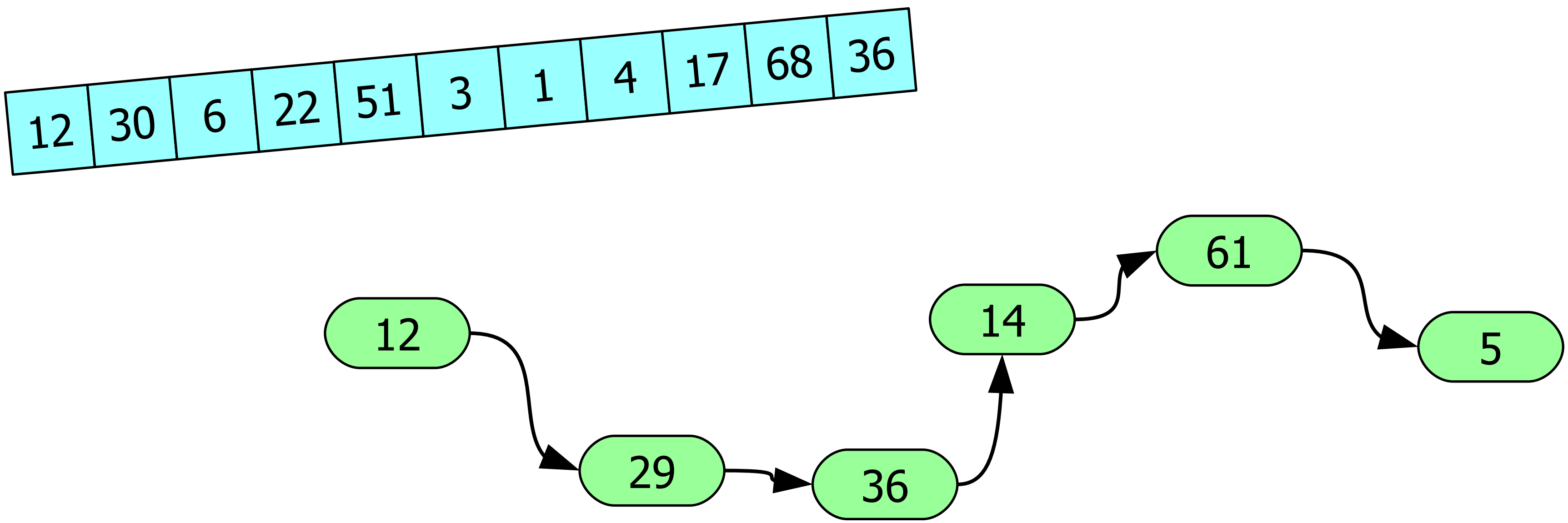
Binary search tree

prerequisites:

- Pointers.

Sequences vs. sets

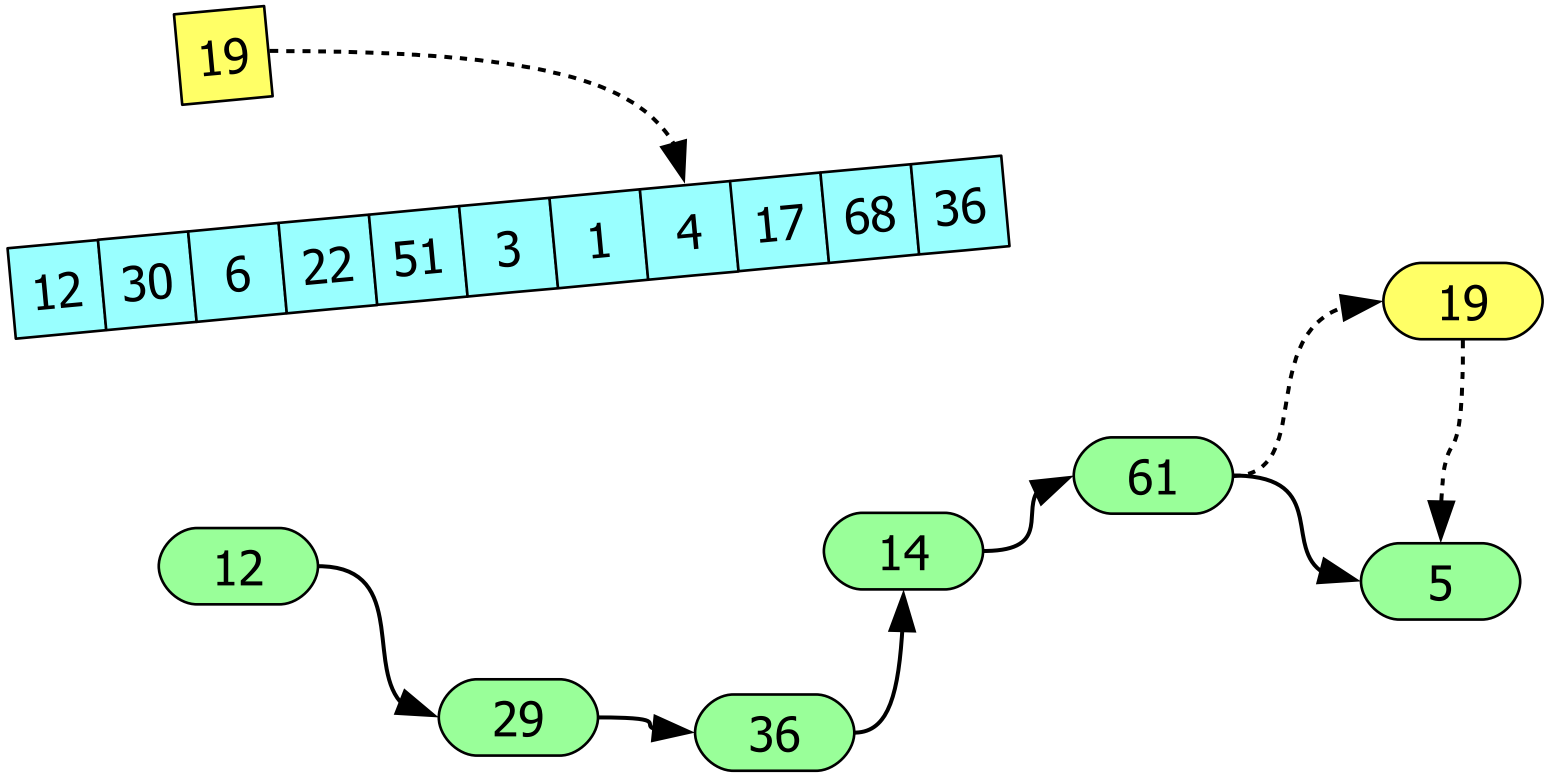
Arrays and linked lists <u>act like sequences</u>:



We can see <u>which value comes after</u> current one, or before it.

Sequences vs. sets

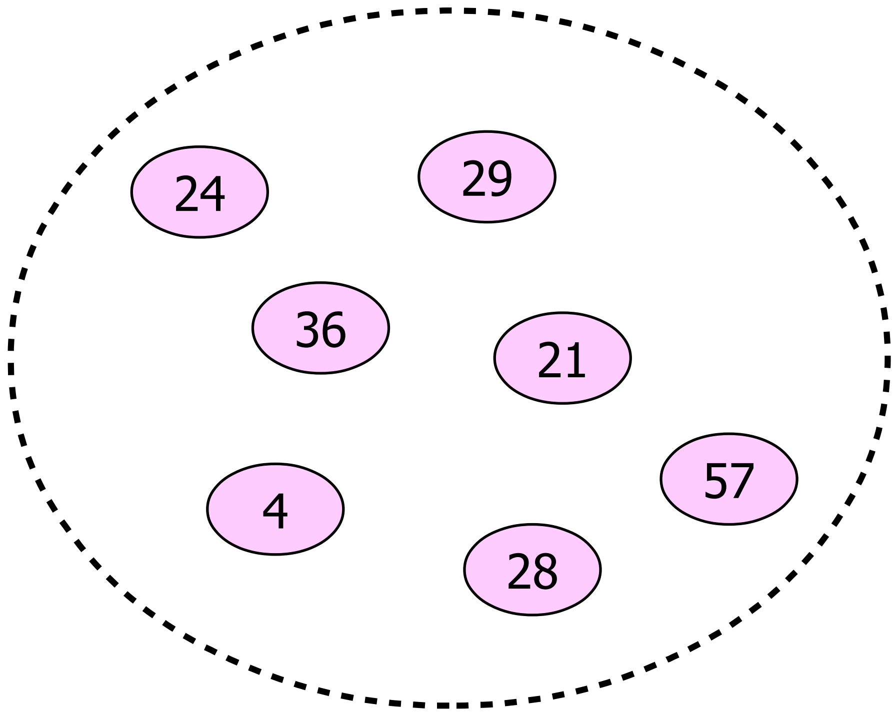
When inserting new value, <u>we specify exact position</u> for it.



Same refers to removal.

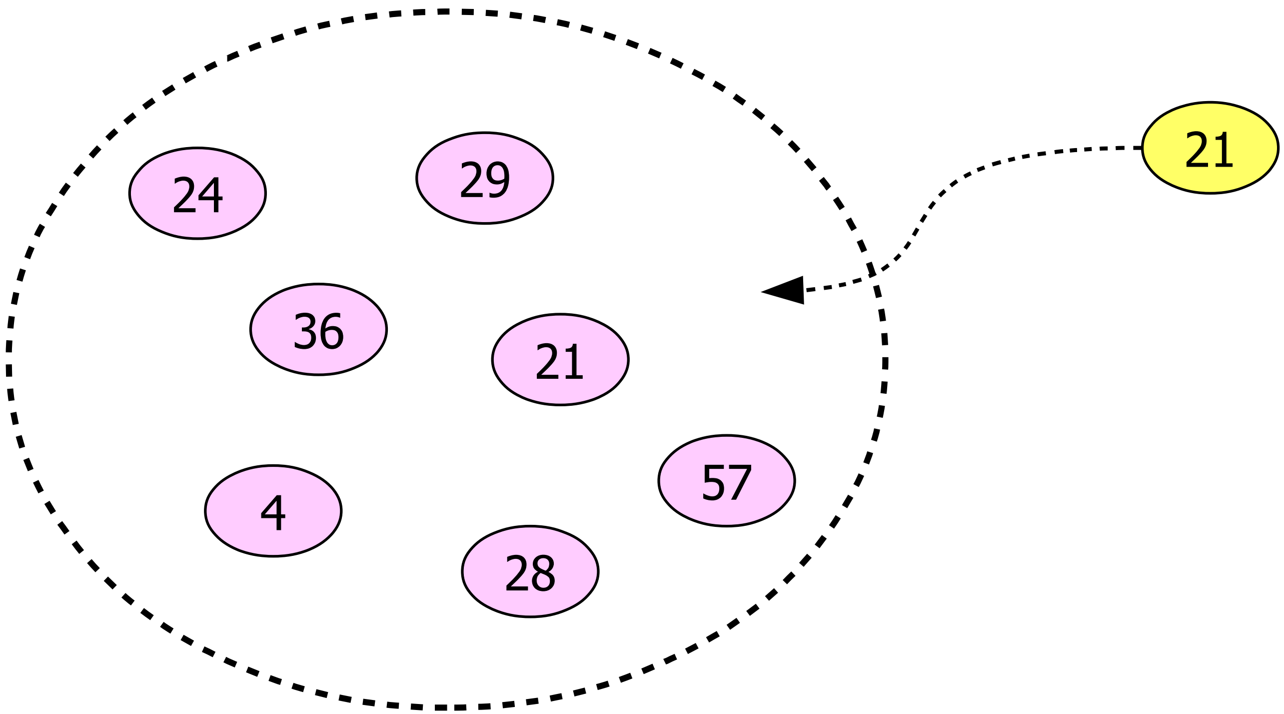
Sequences vs. sets

Binary search trees and hash tables <u>act like sets</u>. We don't specify order of values there.



Sequences vs. sets

Binary search trees and hash tables act like sets. We don't specify order of values there.



When inserting, the structure itself decides where to place the new value.

Sequences vs. sets

This can be easily seen in definitions of their methods:

sequences

```
iterator std::vector::insert(
        const_iterator position,
        const value_type& val );

iterator std::list::insert(
        const_iterator position,
        const value_type& val );
```

sets

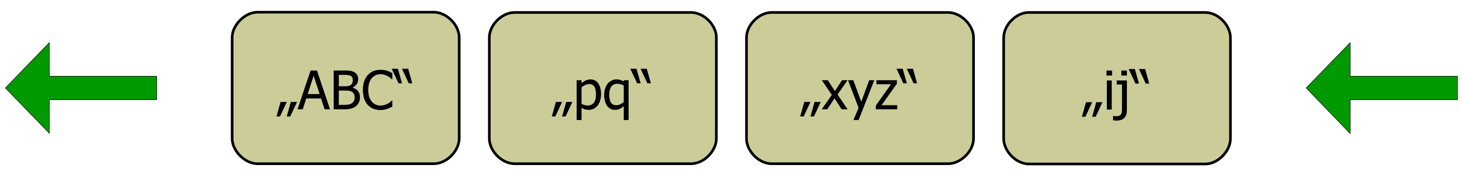
```
pair< iterator, bool > std::set::insert(
        const value_type& val );

pair< iterator, bool > std::unordered_set::insert(
        const value_type& val );
```

Sequences vs. sets

Cases when we need sequences:

1) Queue of some elements:



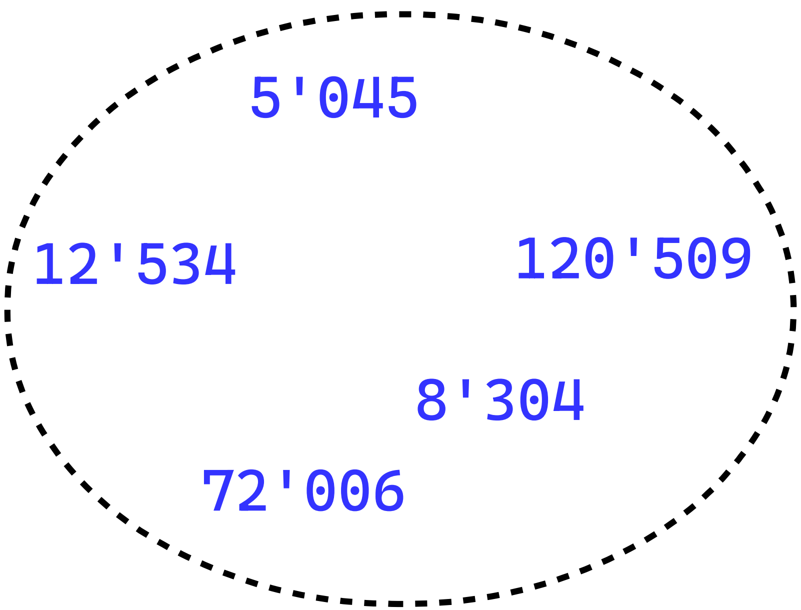
2) Strings:

`„Hello World!“`

Sequences vs. sets

Cases when we need sets:

1) Used IDs:



2) Set of grammatically correct words:

```
...
„dictatorship“
„diction“
„dictionary“
...
```

Sequences vs. sets

Question: Can you bring more examples when we need:

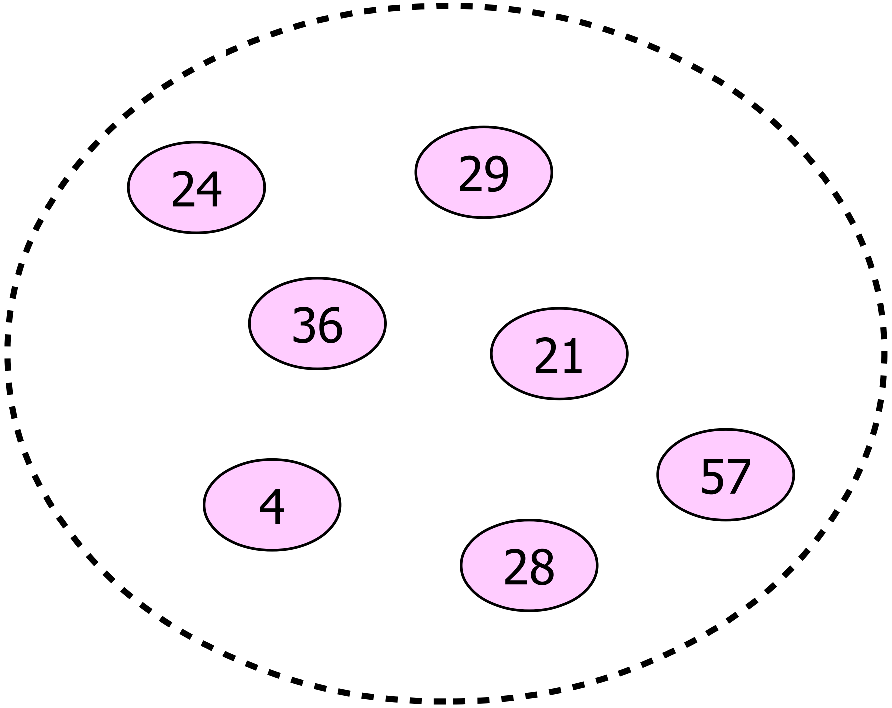
- sequences,
- sets.

Binary tree

Binary search tree is a method for efficient storage of sets.

A BST is composed from nodes,

... each node carries one value of the set.



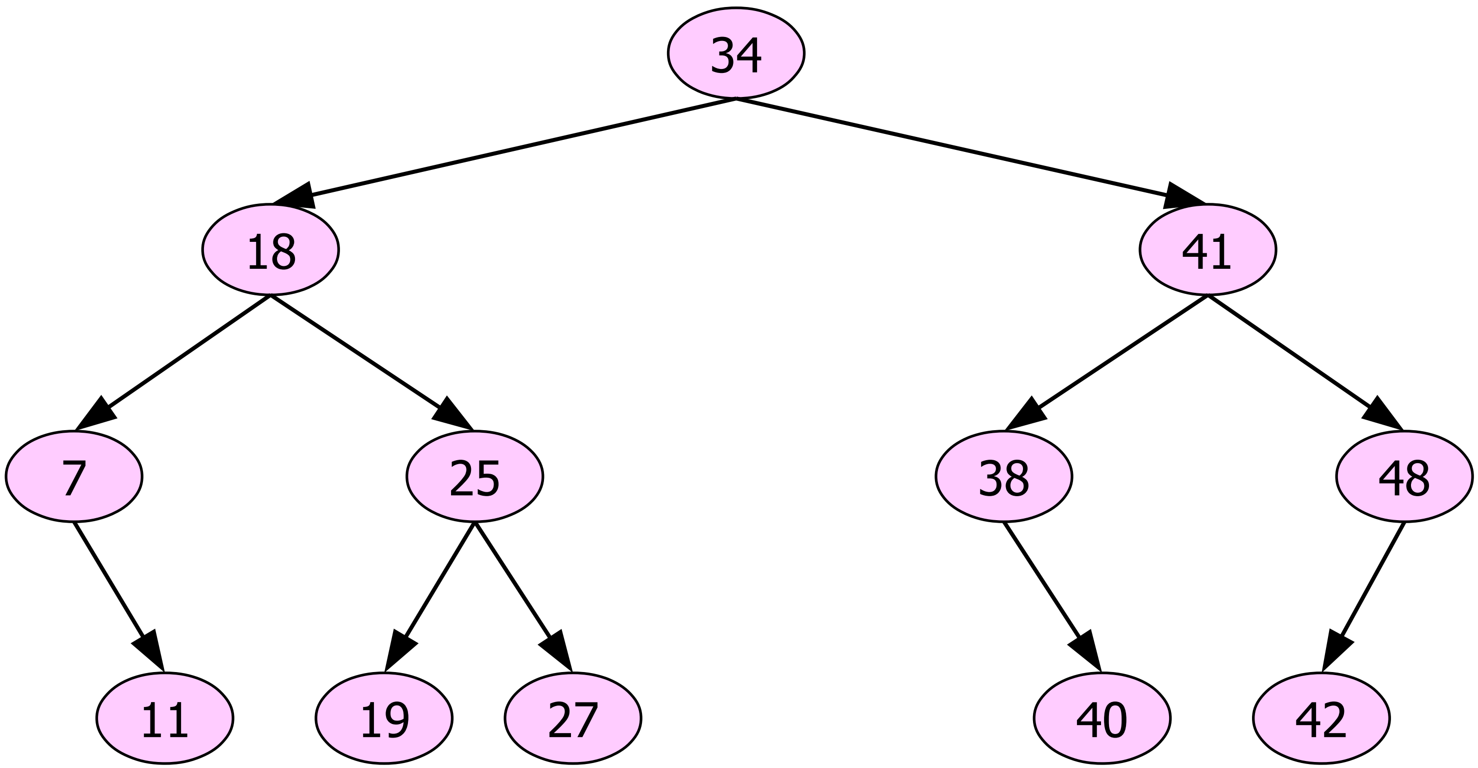
Binary tree

Besides the value, every node has also other fields:

```
structure Node
    value    : Integer,
    left     : pointer to Node,
    right    : pointer to Node.
```

Binary tree

So navigating by "left" and "right" pointers we move down the tree.



Binary tree

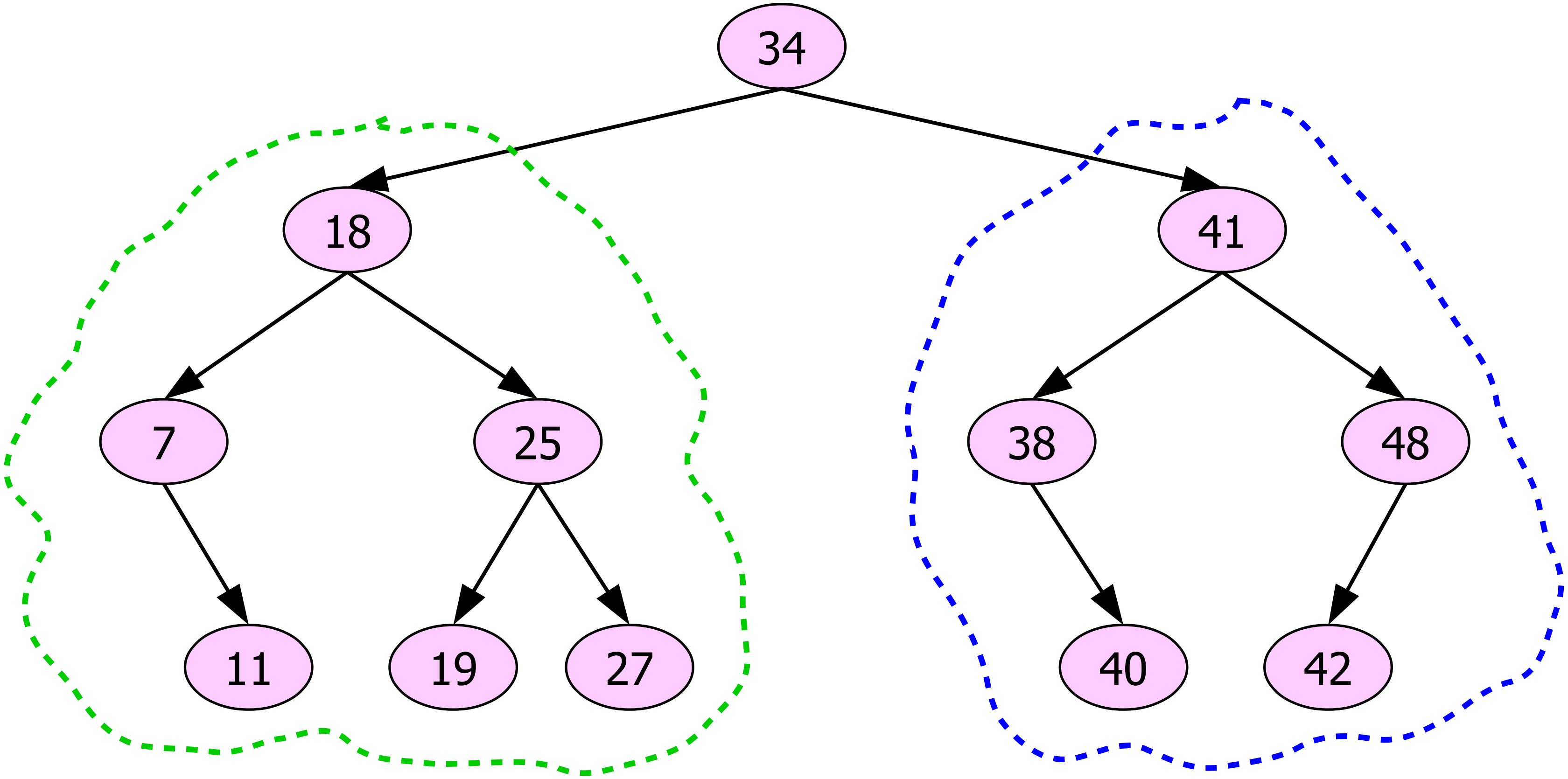
Some properties of the tree:

- Navigating by "left" and "right" pointers we <u>can never fall into a loop</u>,
- Different paths, composed from steps by "left" and "right" <u>can never lead us to the same node</u>.

Binary tree

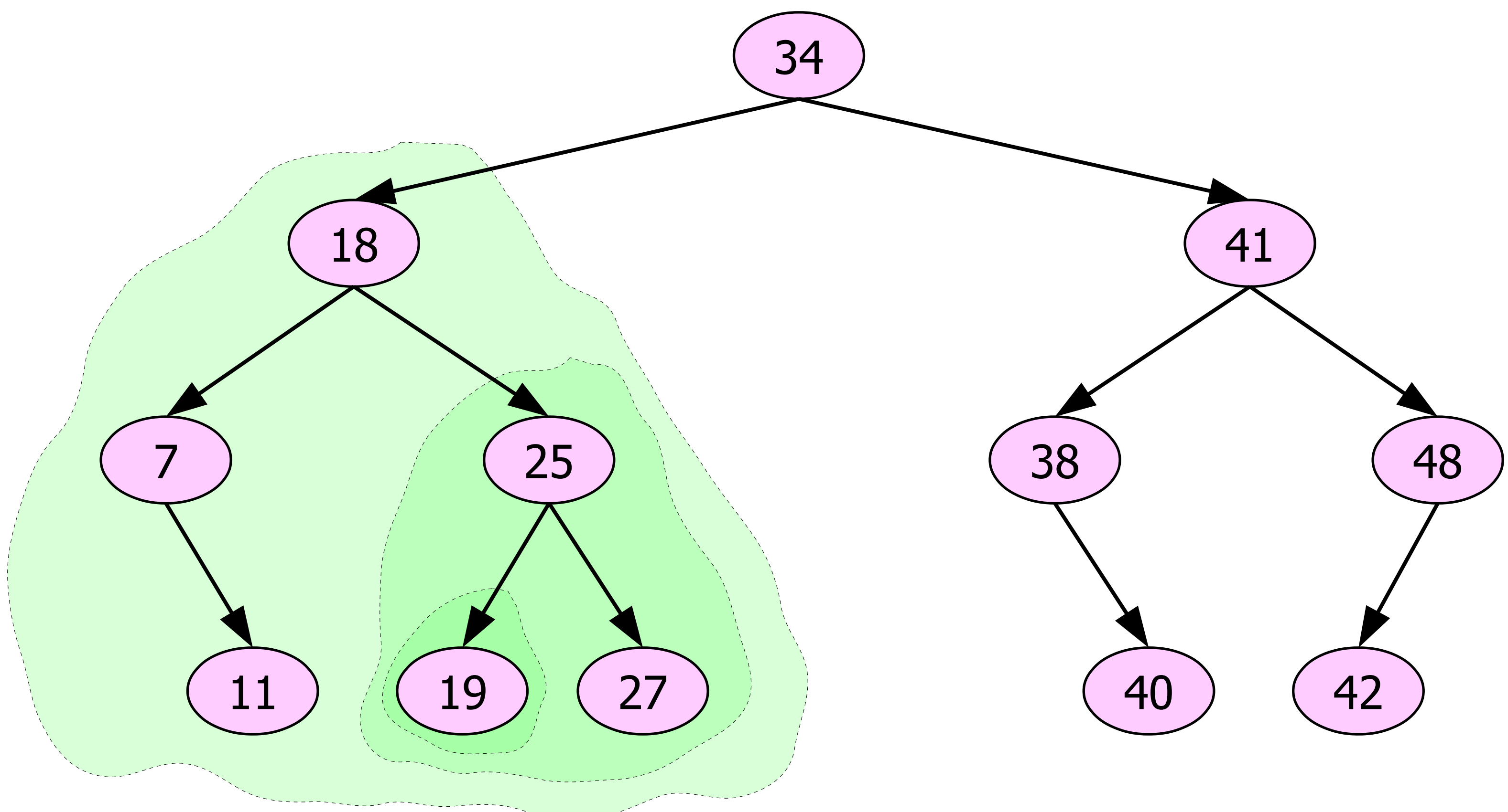
This leads from some other property, that left subtree of a binary tree is also a binary tree,

...as well as its right subtree.



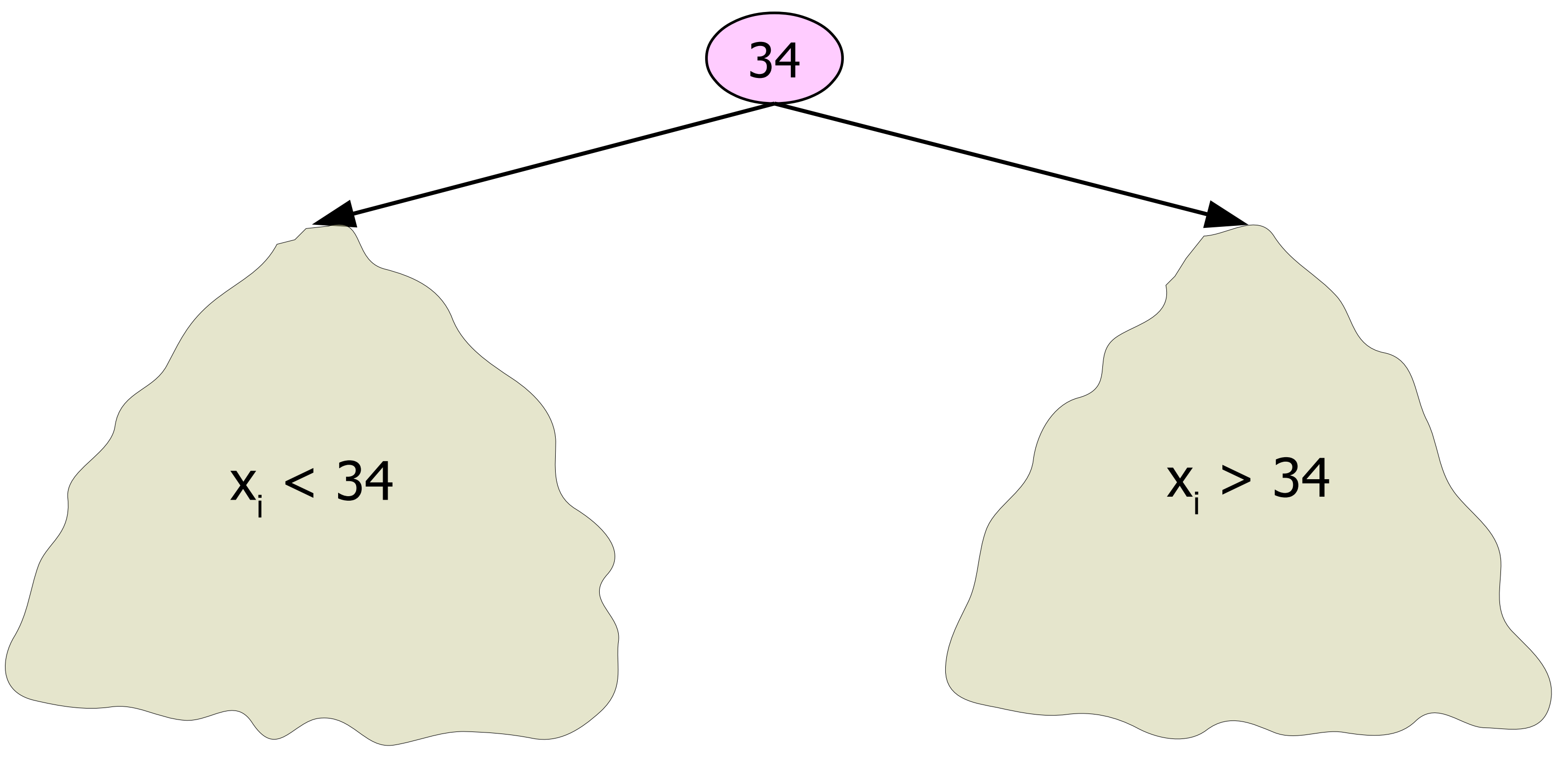
Binary tree

In other words, binary tree is a recursive data structure, as every its subtree can be viewed as another binary tree.



Binary search tree

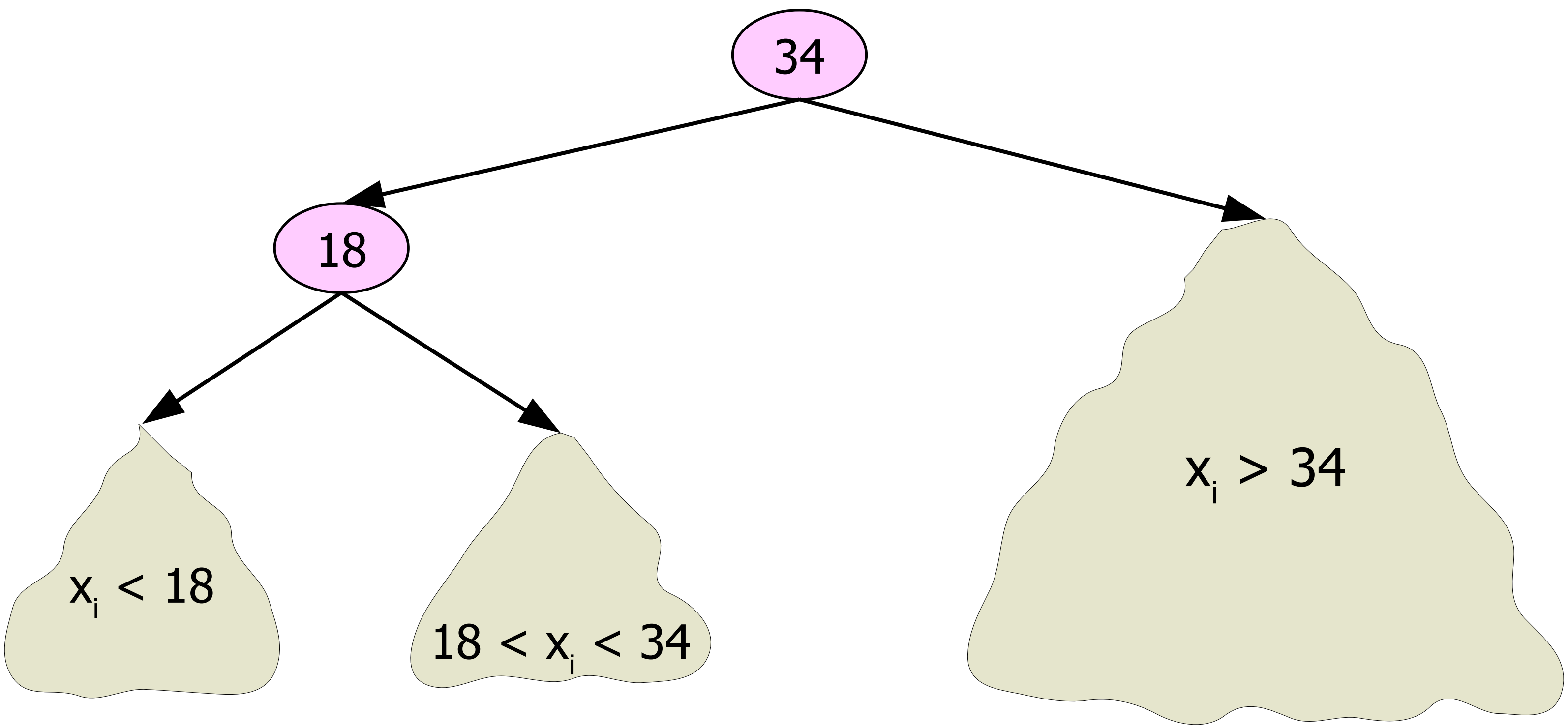
If we speak about binary search tree, every node acts also as a separator:



Binary search tree

If we speak about binary search tree, every node <u>acts also as a separator</u>:

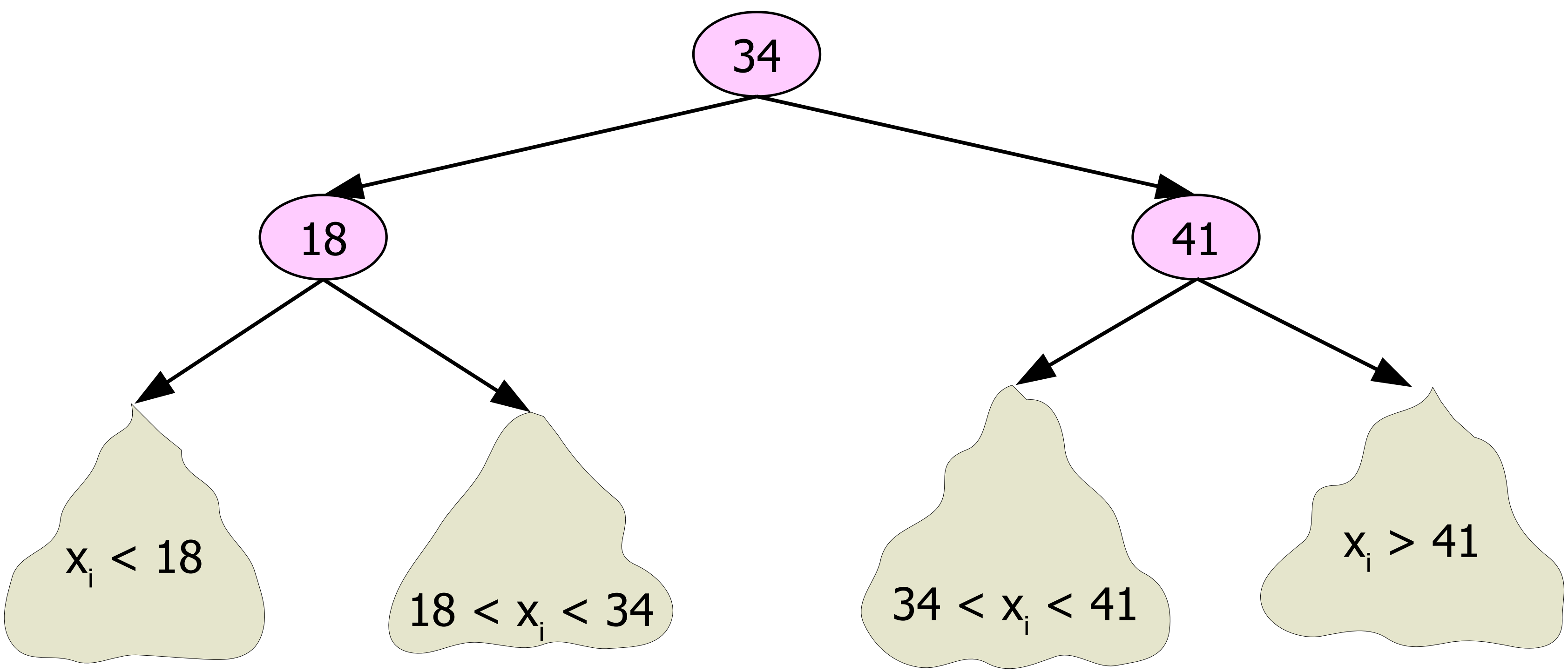
... it refers not only to root node, but to every one.



Binary search tree

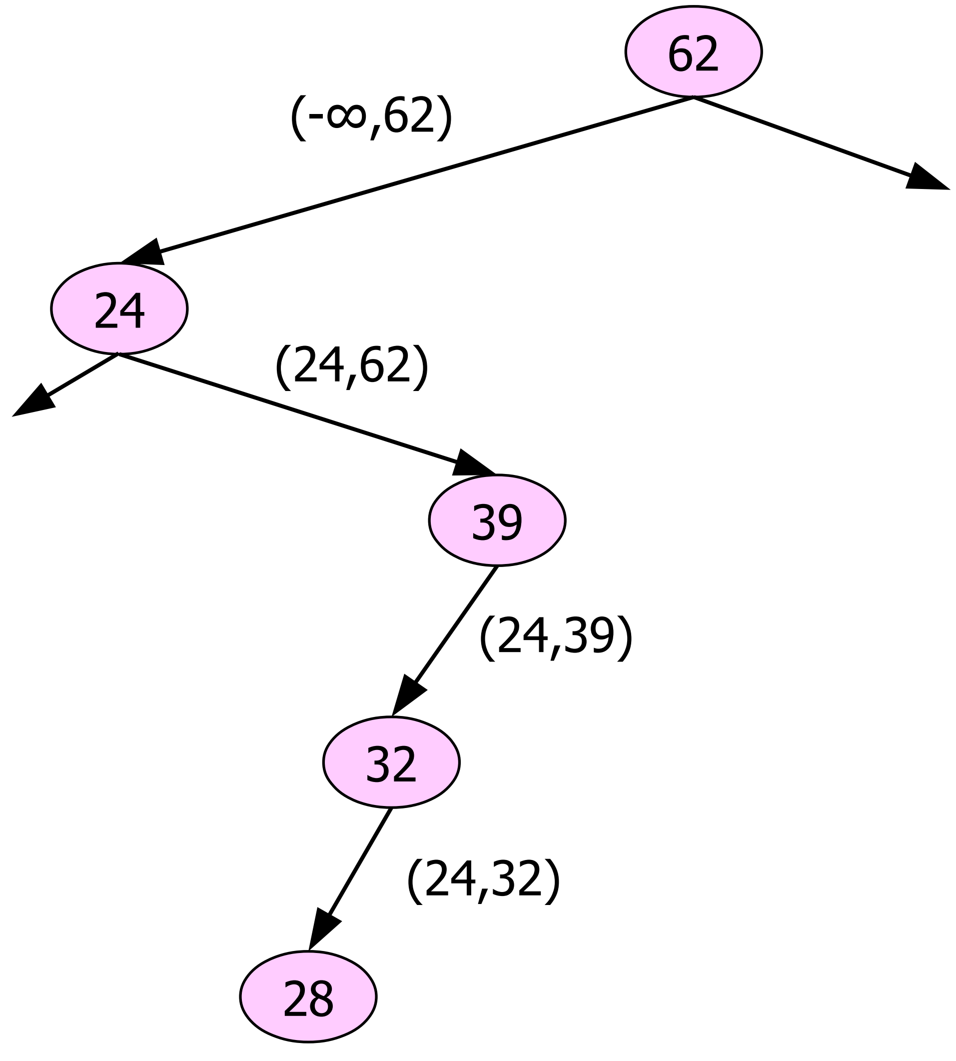
If we speak about binary search tree, every node <u>acts also as a separator</u>:

... it refers not only to root node, but to every one.



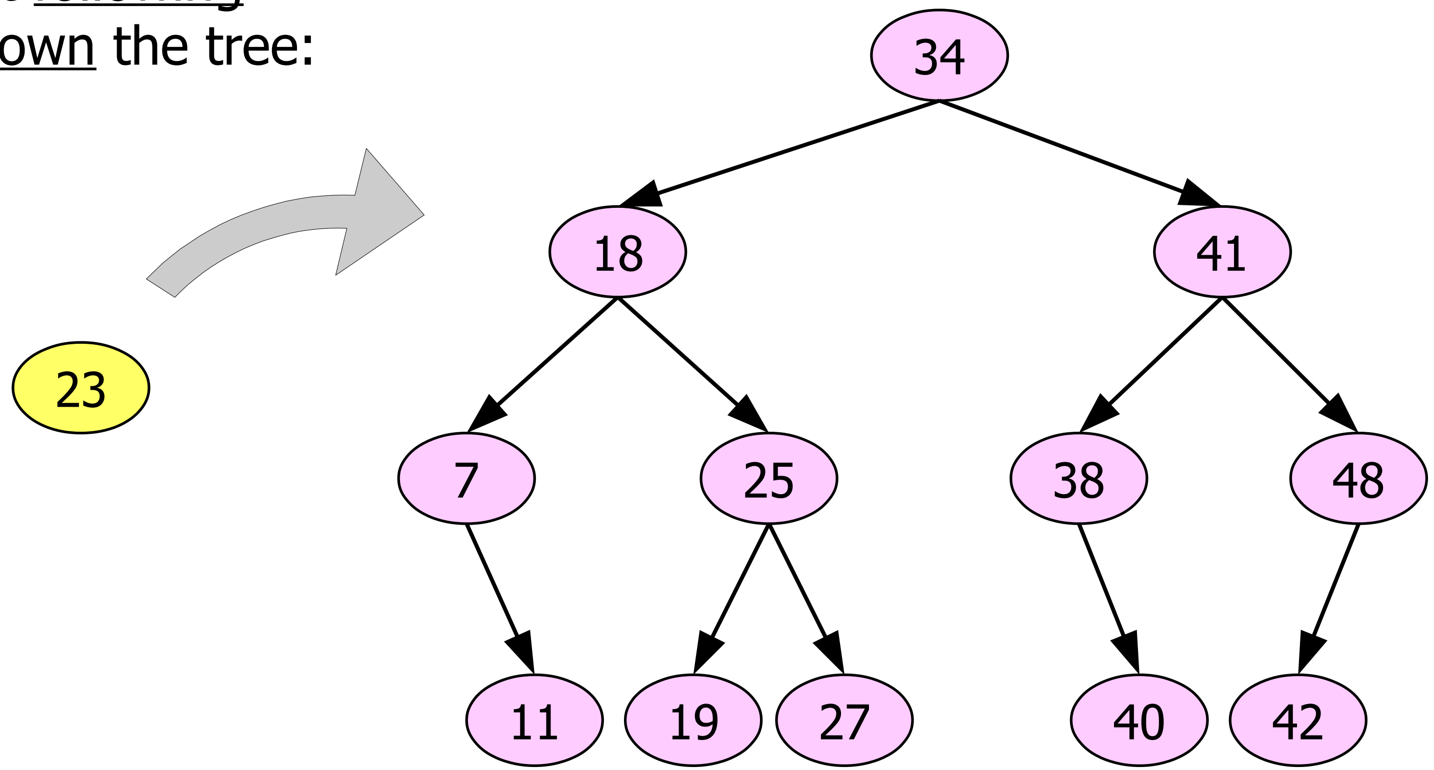
Binary search tree

This means that walking down the tree <u>we limit the possible range of values</u>, that we can meet:



Insertion

Inserting a value '**x**' is performed by just <u>following necessary path down</u> the tree:

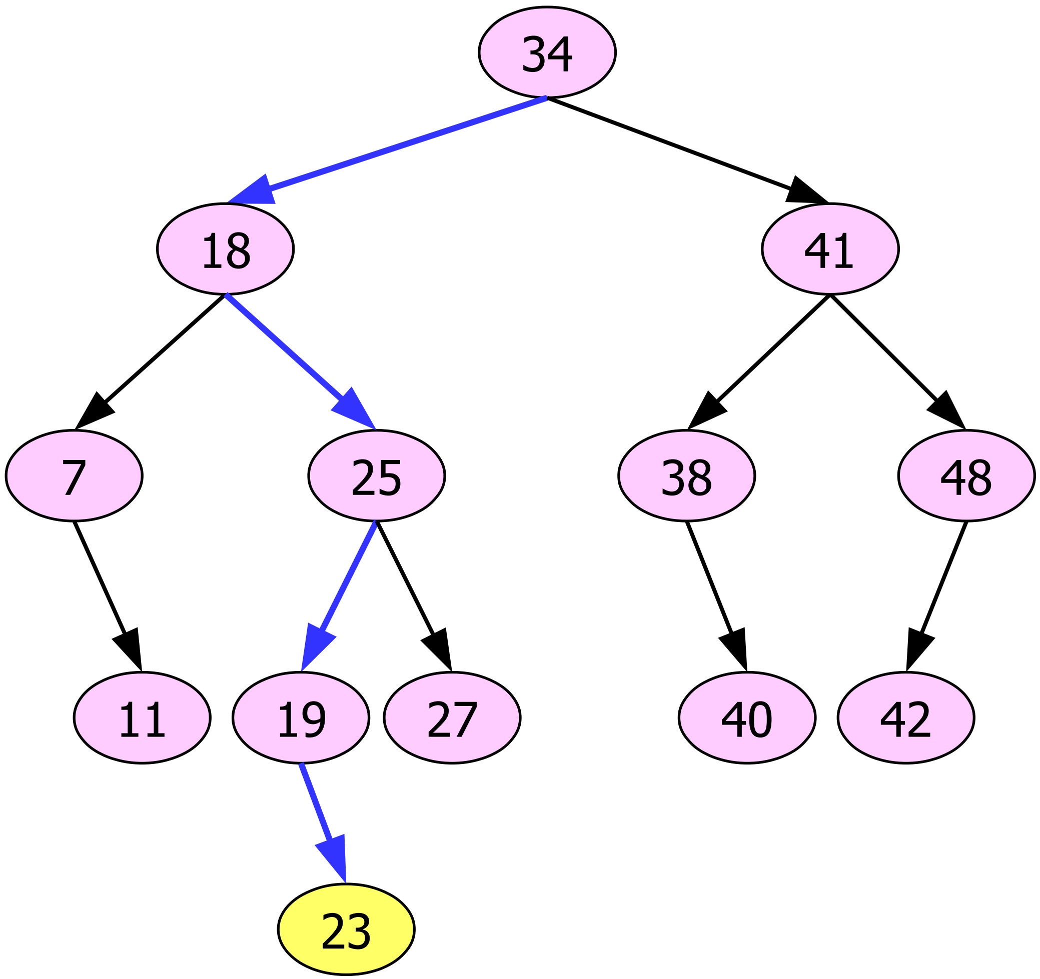


Insertion

Inserting a value '**x**' is performed by just following necessary path down the tree:

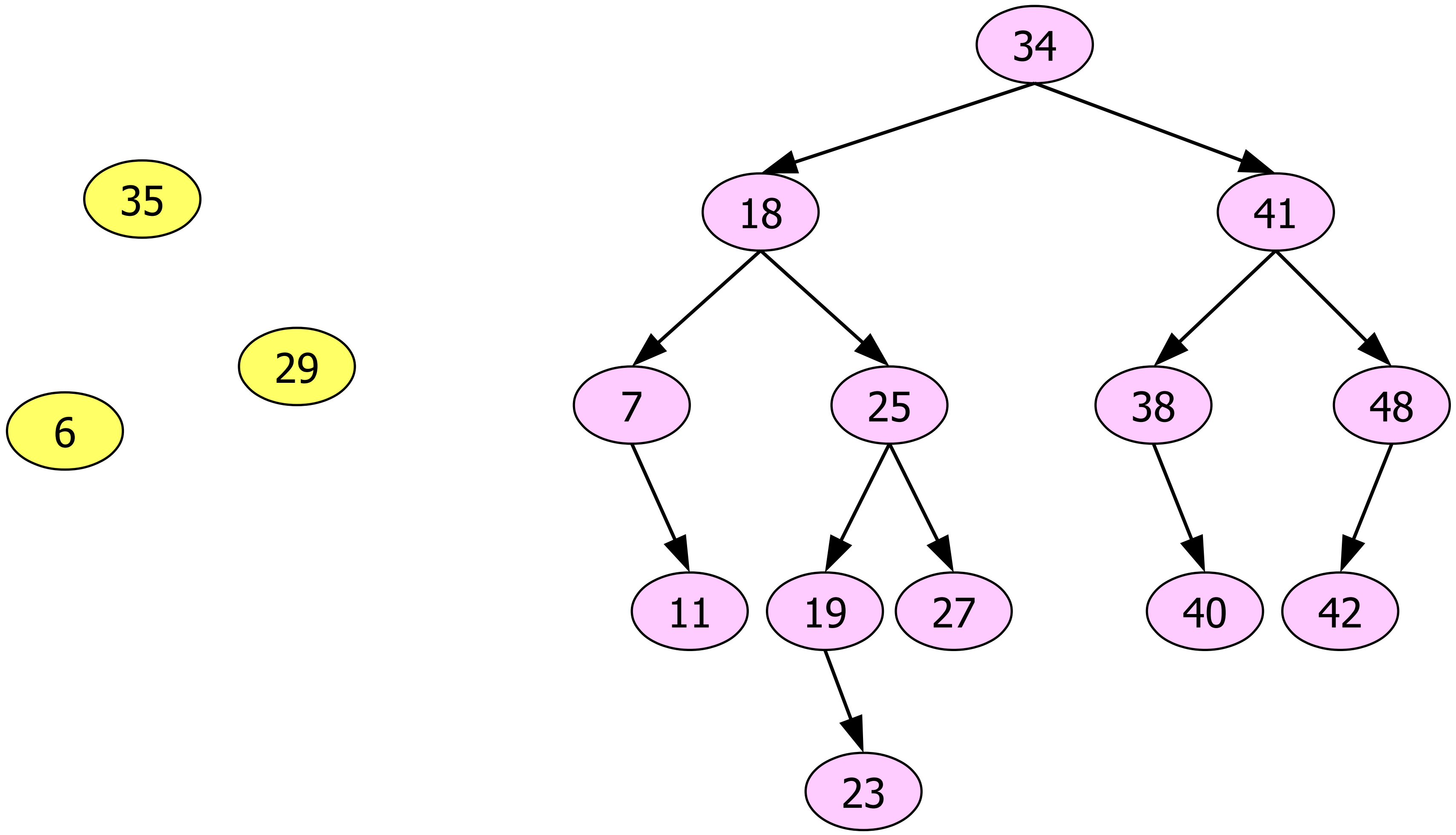
... and placing it as a leaf.

So in binary search tree, new value is always inserted as a leaf.



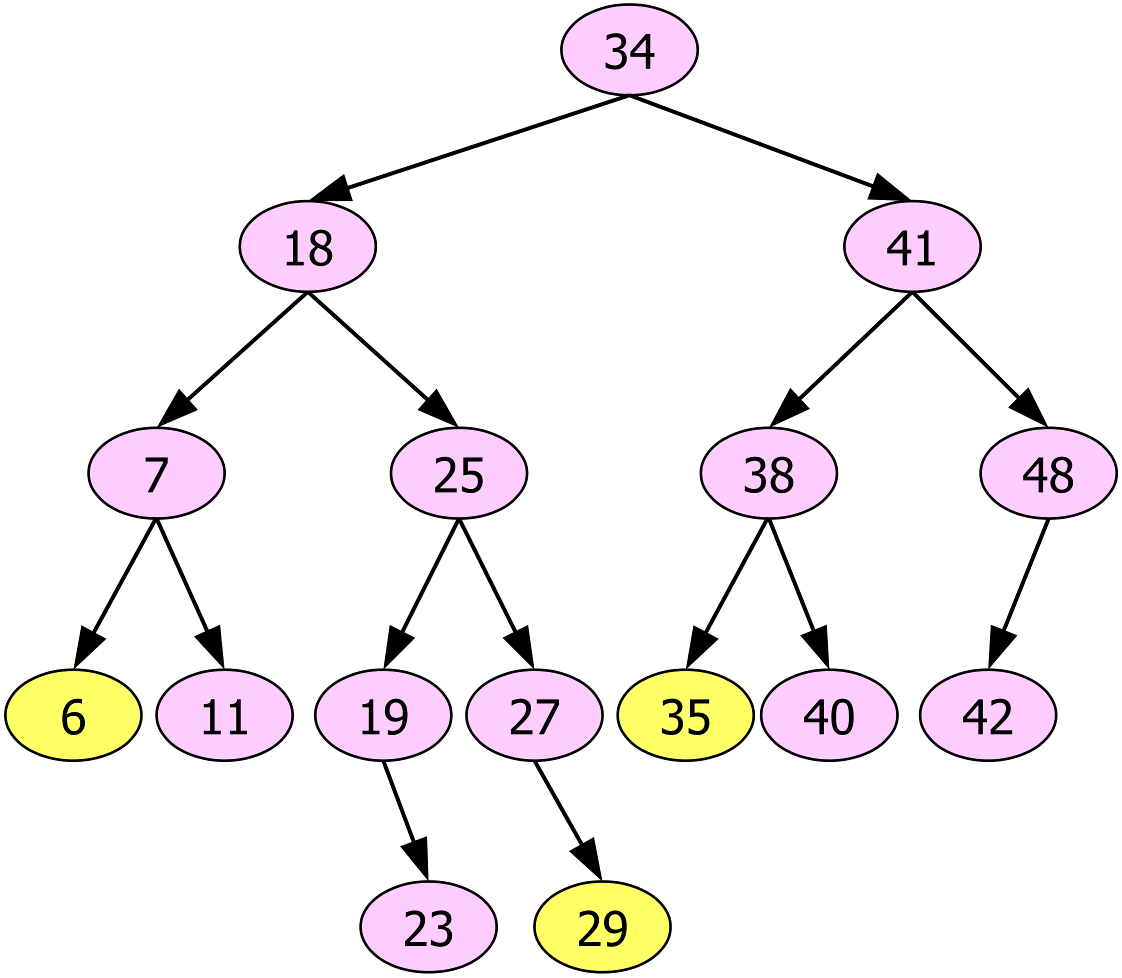
Exercise

Insert the following values into the tree:



Exercise
(solution)

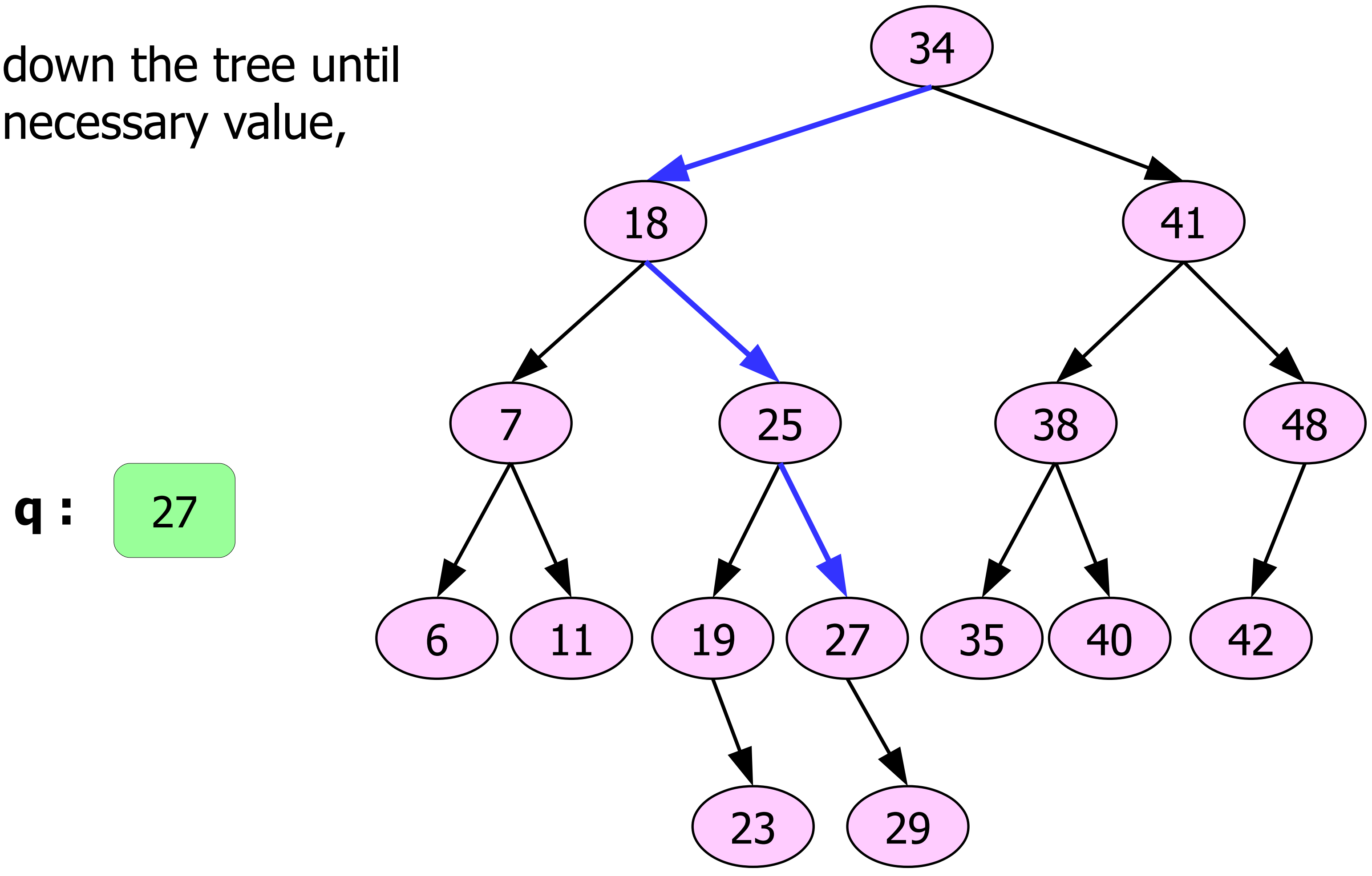
Insert the following values into the tree:



Search

Search is performed similar to insertion:

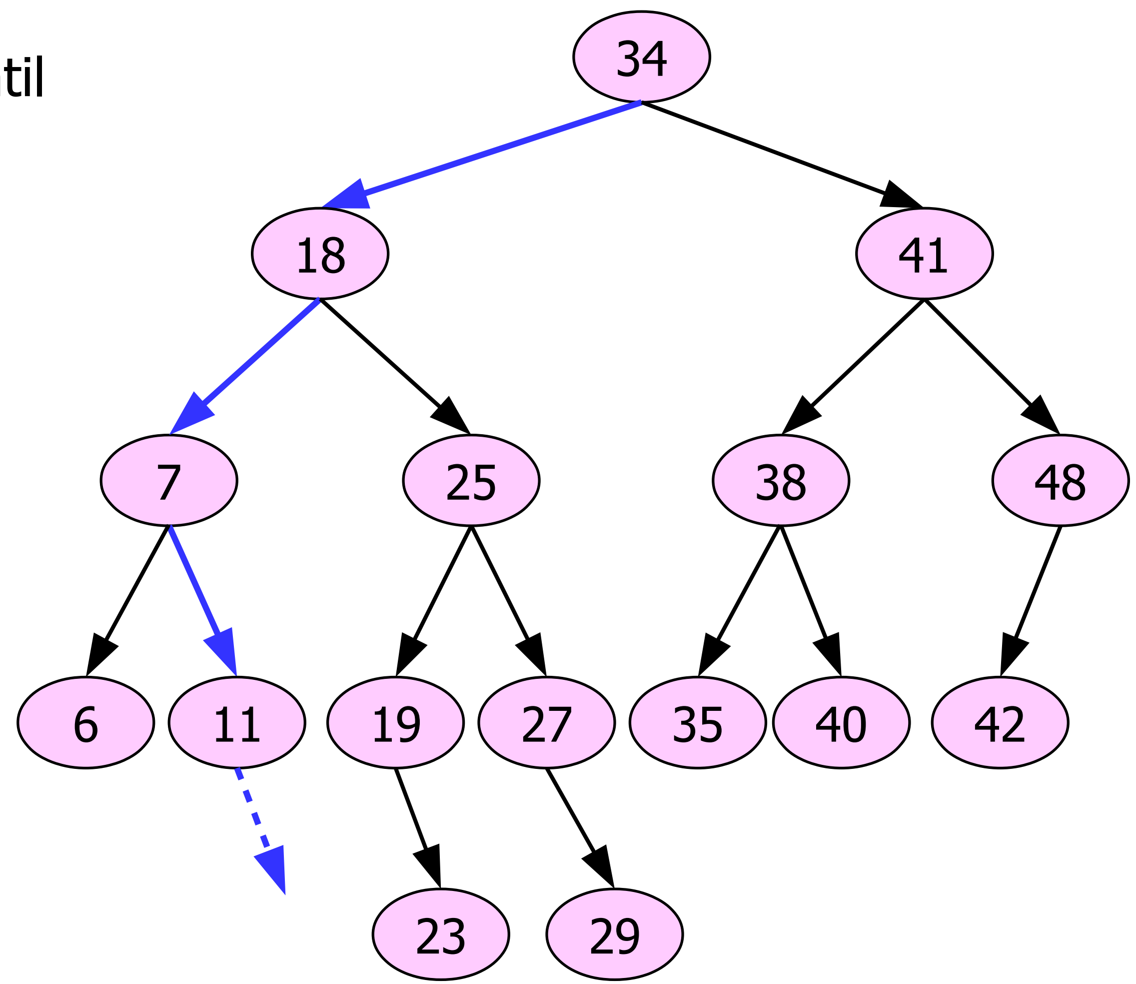
- we walk down the tree until meeting necessary value,



Search

Search is performed similar to insertion:

- we walk down the tree until meeting necessary value,
- or until walking out of the tree.



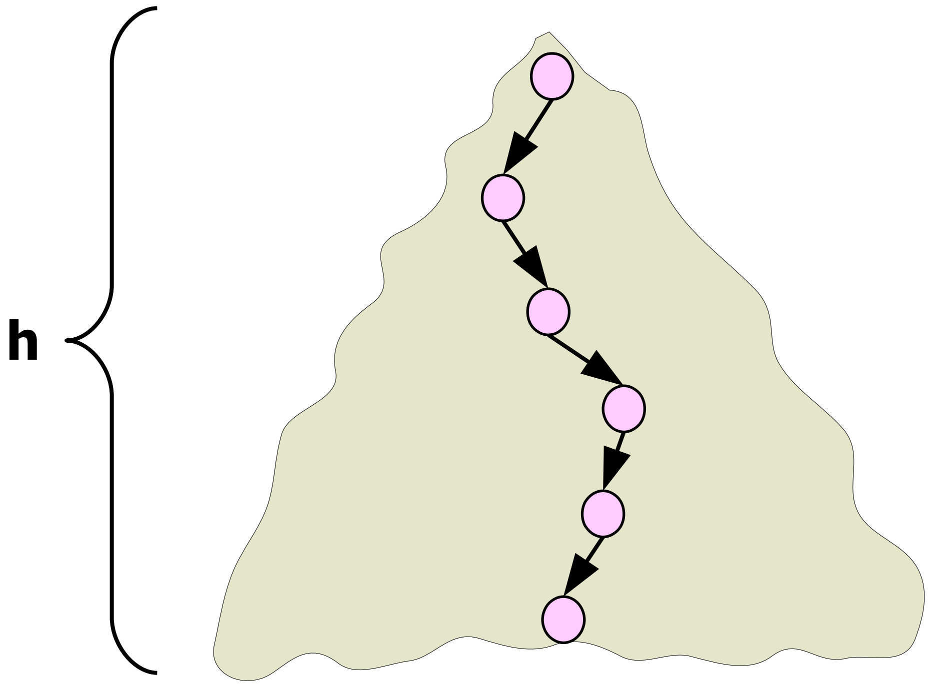
q : 16

Binary search tree

As we have noted, both insertino and search requre **1** walk down the tree,

... which takes time proportional to height of the tree,

... so we aim to keep the height short.



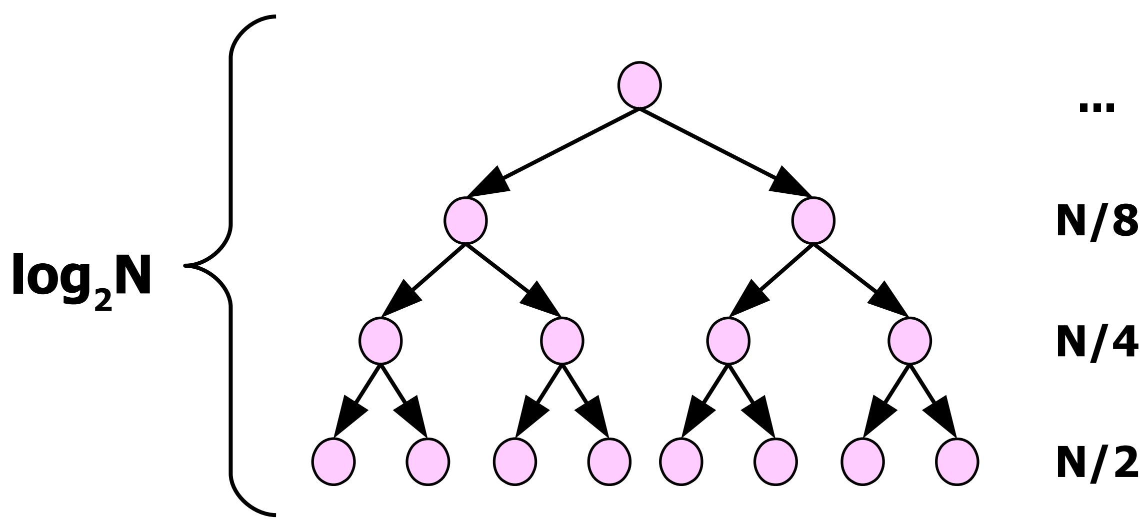
Binary search tree

Question: What is the shortest / longest height, that a Binary search tree with **N** values can have?

Binary search tree

Question: What is the shortest / longest height, that a Binary search tree with **N** values can have?

Answer: Shortest possible height is $\mathbf{log_2N}$.

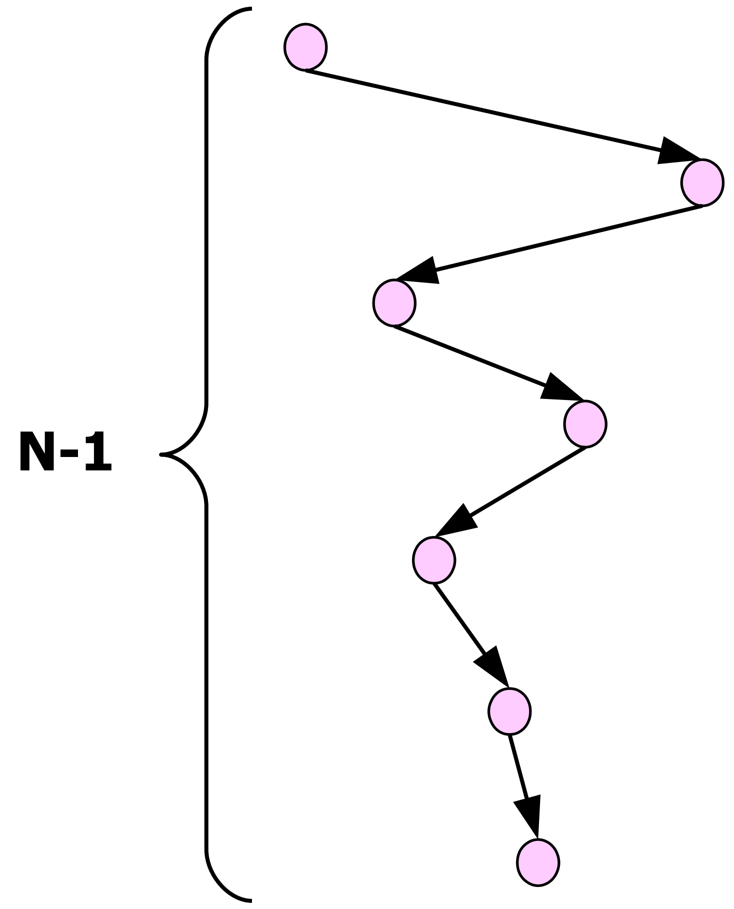


Binary search tree

Question: What is the shortest / longest height, that a Binary search tree with **N** values can have?

Answer: Shortest possible height is $\mathbf{log_2N}$.

... while longest possible height is **N-1**.



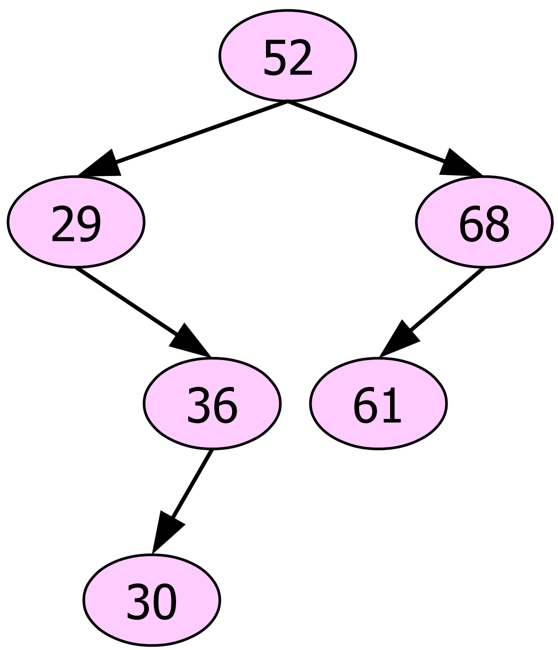
Exercises

Implement:

- structure of node of Binary search tree,
- inserting values into it,
- searching for values,
- checking if given tree is a Binary search tree.

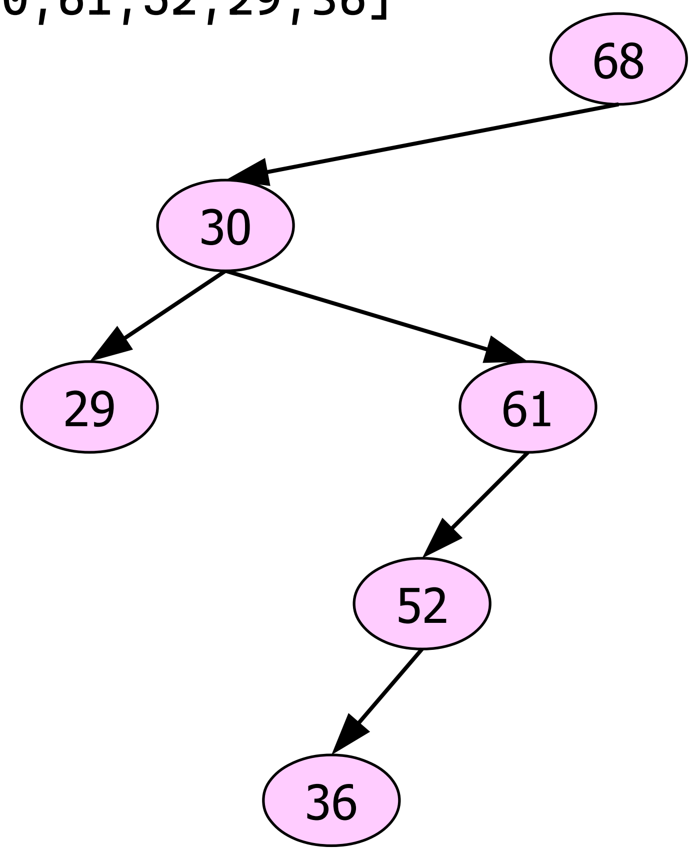
Properties

An important property for binary search tree is that <u>order of insertion matters</u>:



[52,29,36,68,30,61]

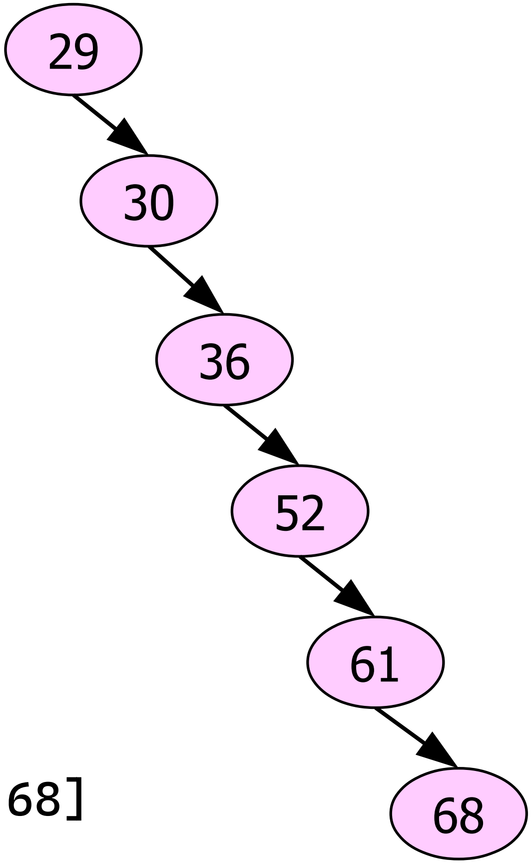
[68,30,61,52,29,36]



Properties

If inserting all values in increasing order, the tree will <u>become degraded</u>,

... its height will be maximal.



`[29,30,36,52,61,68]`

Properties

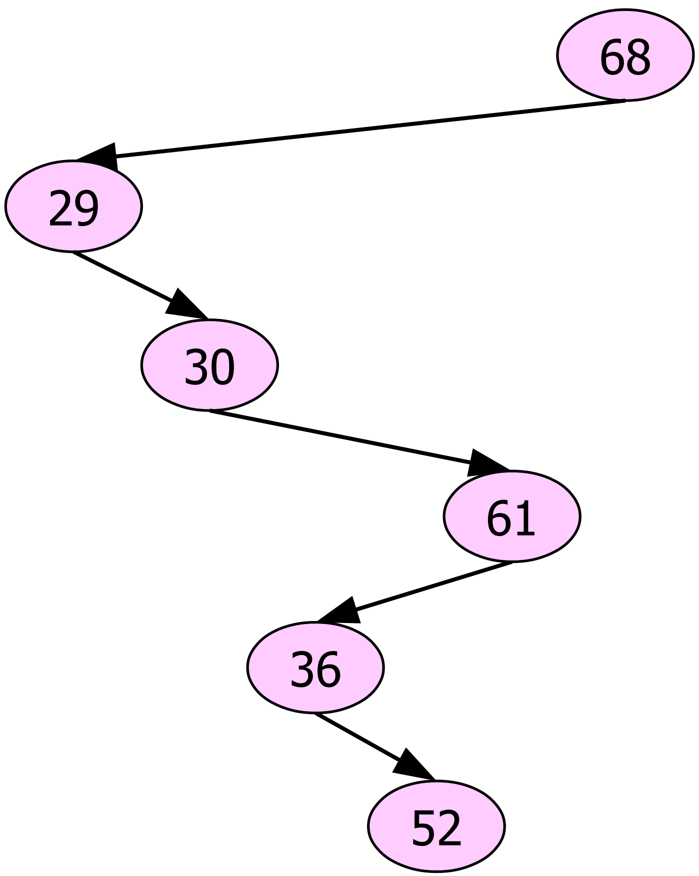
Question: What other sequences of insertion will bring to a degraded tree?

Properties

Question: <u>What other sequences</u> of insertion will bring to a degraded tree?

Answer: So called "narrowing" sequence,

... where every next value <u>is either minimal or maximal of the remaining</u> ones.

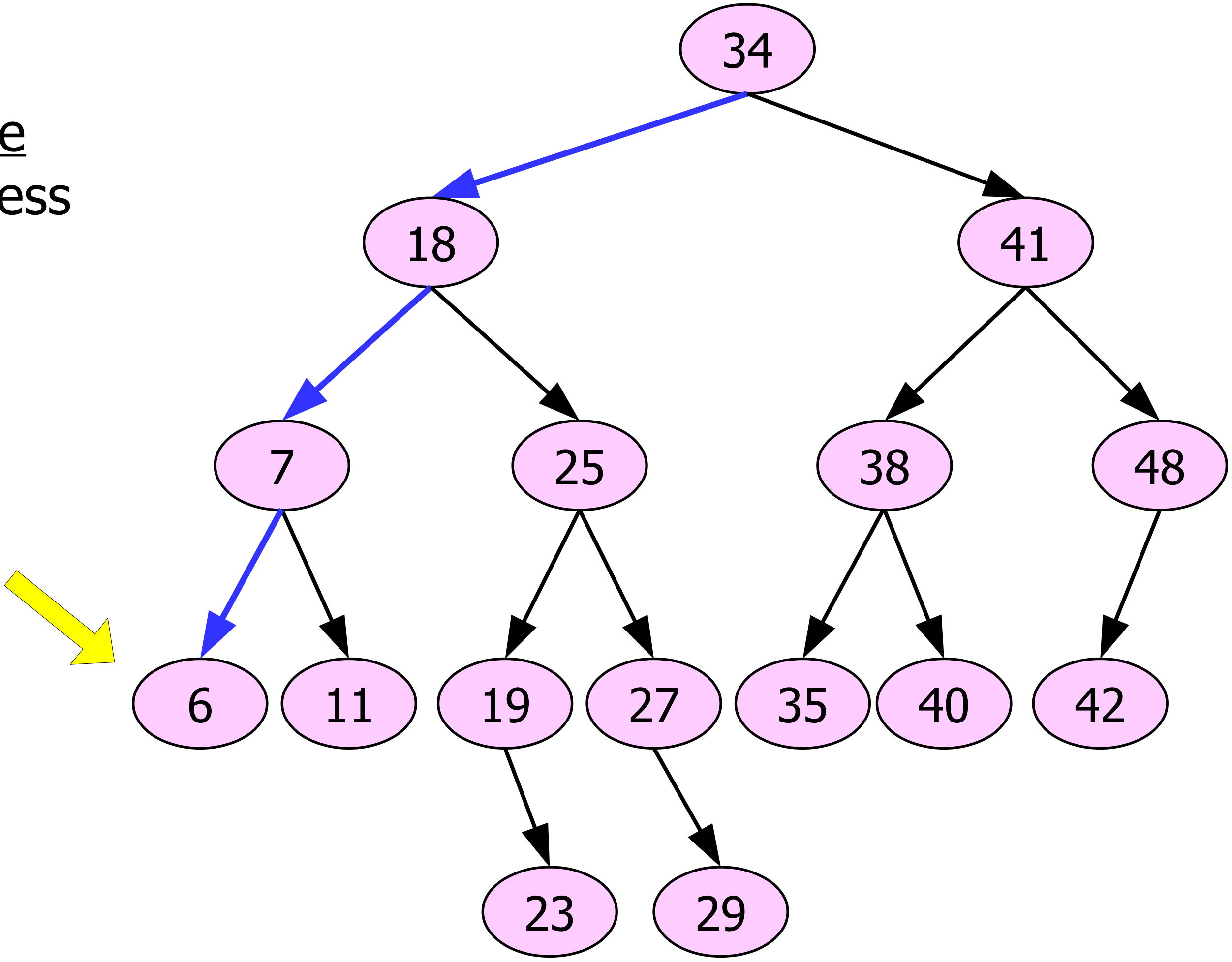


`[68,29,30,61,36,52]`

Properties

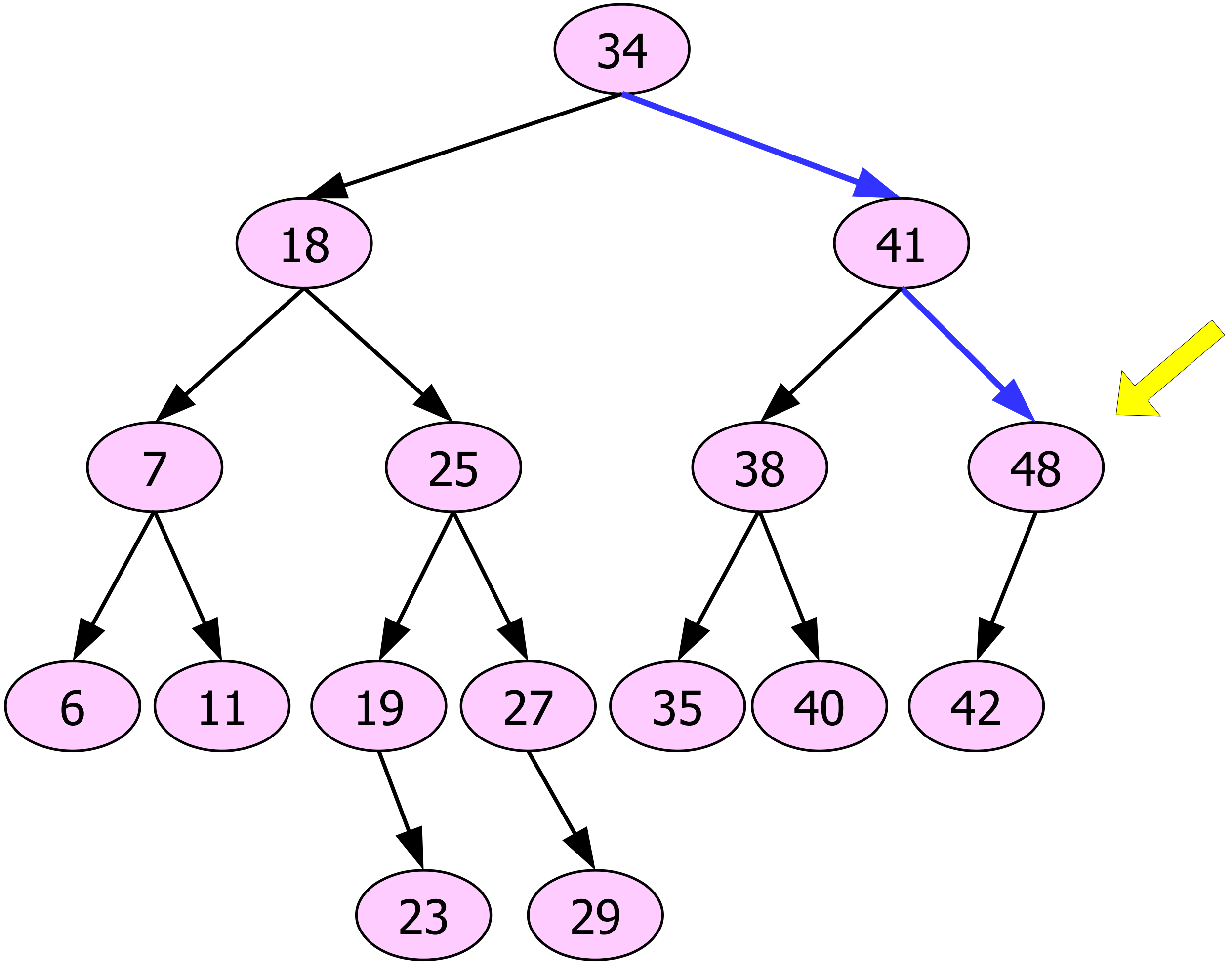
To find the minimal value we must <u>go left, as long as</u> that is possible.

... because <u>left subtree means</u> there are values less than the current.



Properties

Similarly the maximal value should be searched.



Implementation in STL

The C++ standard library has functions for binary search on a sorted array:

```
template< typename FwdIt, typename T >
FwdIt lower_bound(FwdIt first, FwdIt last, const T& val);

template< typename FwdIt, typename T >
FwdIt upper_bound(FwdIt first, FwdIt last, const T& val);
```

But for binary search trees there are custom implementations:

```
iterator std::set<T>::lower_bound(const T& val);

iterator std::set<T>::upper_bound(const T& val);
```

Exercises

Continue implementation:

- finding minimal / maximal values,
- write insertion / search in recursive way (without loop).

Comparators

During every operation, when navigating the tree, we must know if '**x**' is "less than" '**y**', or not.

... in other words, if '**x**' comes before '**y**'.

11 **<** 38

This is quite easy to do for numbers, or even strings,

... strings are being compared in lexicographical order then.

Comparators

But if working with complex data types, comparisng them is not quite easy.

From: Abovyan
To: Sevan
Quantity: 2
Cost: 15

X: 130.5
Y: 52.6
Z: -1.5

Comparators

When comparing data types, the following must hold:

1) $A \nless A$

2) $A < B \;\Rightarrow\; B \nless A$

3) $A < B$ **&&** $B < C$

$\Rightarrow A < C$

... which is quite expected, if it should express 'x' <u>coming before</u> 'y'.

Comparators

Common solution is to sequentially compare corresponding fields, until finding a difference.

... let's see how it is done in C++ STL library:

Comparators

Given "Delivery" structure as the following:

```
struct Delivery {
    std::string _from;
    std::string _to;
    int _quantity;
    double _cost;
};
```

Comparators

The comparator can be written as:

```
struct DeliveryComparator {
    bool operator()( const Delivery& x,
            const Delivery& y ) const {
        if ( x._from != y._from )  // Compare „_from“
            return x._from < y._from;
        if ( x._to != y._to )      // Compare „_to“
            return x._to < y._to;
        if ( x._quantity != y._quantity )  // Compare „_qty“
            return x._quantity < y._quantity;
        if ( x._cost != y._cost )  // Compare „_cost“
            return x._cost < y._cost;
        return false;              // The entries are equal
    }
};
```

Comparators

Question 1: Is this comparator proper?

Question 2: Is this comparator good enough?

Comparators

Question 1: Is this comparator proper?

Answer 1: Yes, it is.

Question 2: Is this comparator good enough?

Answer 2: No.

At first we compare "`_from`",

... which is of type '`std::string`',

... comparison of strings is done much slower than comparison of numbers.

... if deliveries differ by quantity or cost (which is quite probable), it will be faster to arrange them in that way.

Comparators

So such comparator will be better:

```
struct DeliveryComparator {
    bool operator()( const Delivery& x,
            const Delivery& y ) const {
        if ( x._quantity != y._quantity )  // Compare „_qty“
            return x._quantity < y._quantity;
        if ( x._cost != y._cost )  // Compare „_cost“
            return x._cost < y._cost;
        if ( x._from != y._from )  // Compare „_from“
            return x._from < y._from;
        if ( x._to != y._to )      // Compare „_to“
            return x._to < y._to;
        return false;              // The entries are equal
    }
};
```

Comparators

Question 3: Is this comparator written in the best posible way?

Comparators

Question 3: Is this comparator written in the best posible way?

Answer: No.

- A lot of deliveries might have "`quantity = 1`",
- Comparing such quantities at the very beginning will result in waste of time.
- So it will be better to comare "`cost`" at first.

Comparators

This comparator will be <u>even better</u>:

```
struct DeliveryComparator {
    bool operator()( const Delivery& x,
            const Delivery& y ) const {
        if ( x._cost != y._cost )  // Compare „_cost“
            return x._cost < y._cost;
        if ( x._quantity != y._quantity )  // Compare „_qty“
            return x._quantity < y._quantity;
        if ( x._from != y._from )  // Compare „_from“
            return x._from < y._from;
        if ( x._to != y._to )      // Compare „_to“
            return x._to < y._to;
        return false;              // The entries are equal
    }
};
```

Comparators

So summarizing,

- both "`quantity`" and "`cost`" are integers, and comparing them is fast,
- but the probability of "`quantity`"-es to be equal is higher,
- that's why it is better to compare "`cost`"-s at first.

Comparators

Custom comparators can be implemented <u>for primitive types too</u>:

- comparing integers only by last digit,
- comparing strings in a different way,
- etc...

Comparators

Question: In C++ comparators can be written both as classes and as functions. Is one approach better than the other?

```
struct DeliveryComparator {
    bool operator()( const Delivery& x,
            const Delivery& y ) const {
        ...
    }
};
```

```
bool compareDeliveries( const Delivery& x,
        const Delivery& y ) {
    ...
}
```

Comparators

Question: In C++ comparators can be written both as classes and as functions. Is one approach better than the other?

Answer: If writing in a class, we can:

- customize the comparator, and even
- change its behavior over time.

... done with help of member variables.

Exercises

Write comparators for:

Name: <string>
Surname: <string>
Age: <integer>
Salary: <integer>

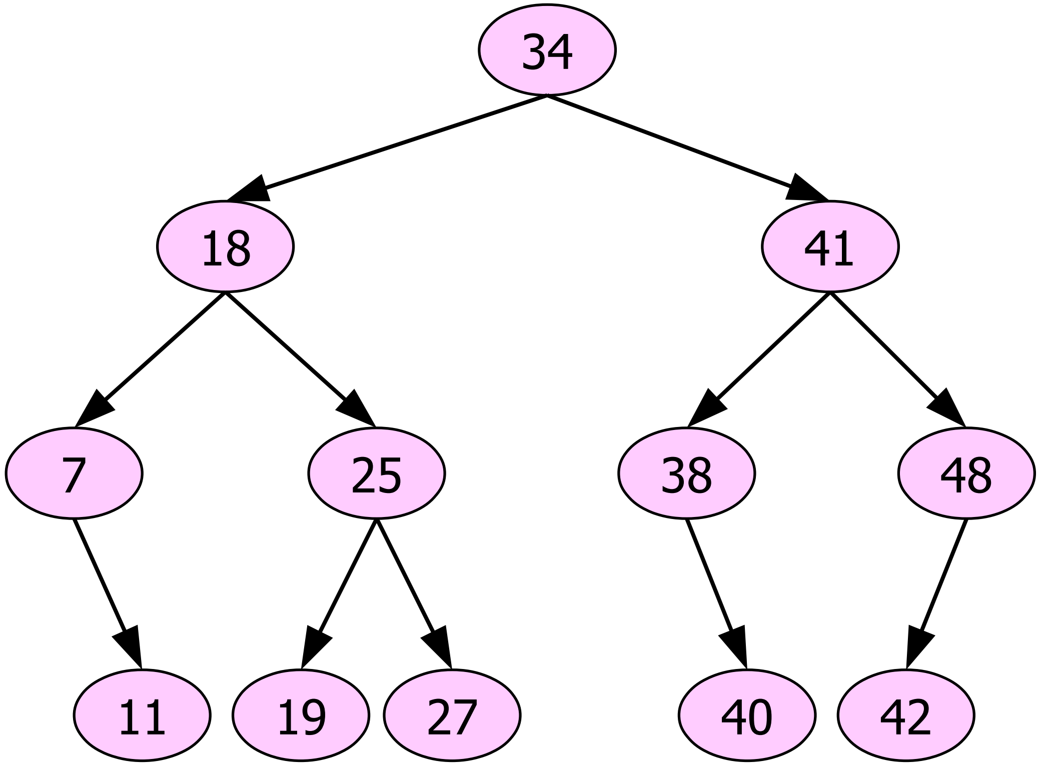
from_id: <integer>
to_id: <integer>
quantity: <integer>

X: <real>
Y: <real>
Title: <string>
Bonus: <integer>

Traversals

Traversing Binary search trees is <u>not as simple as</u> traversing:

- arrays, or
- linked lists.



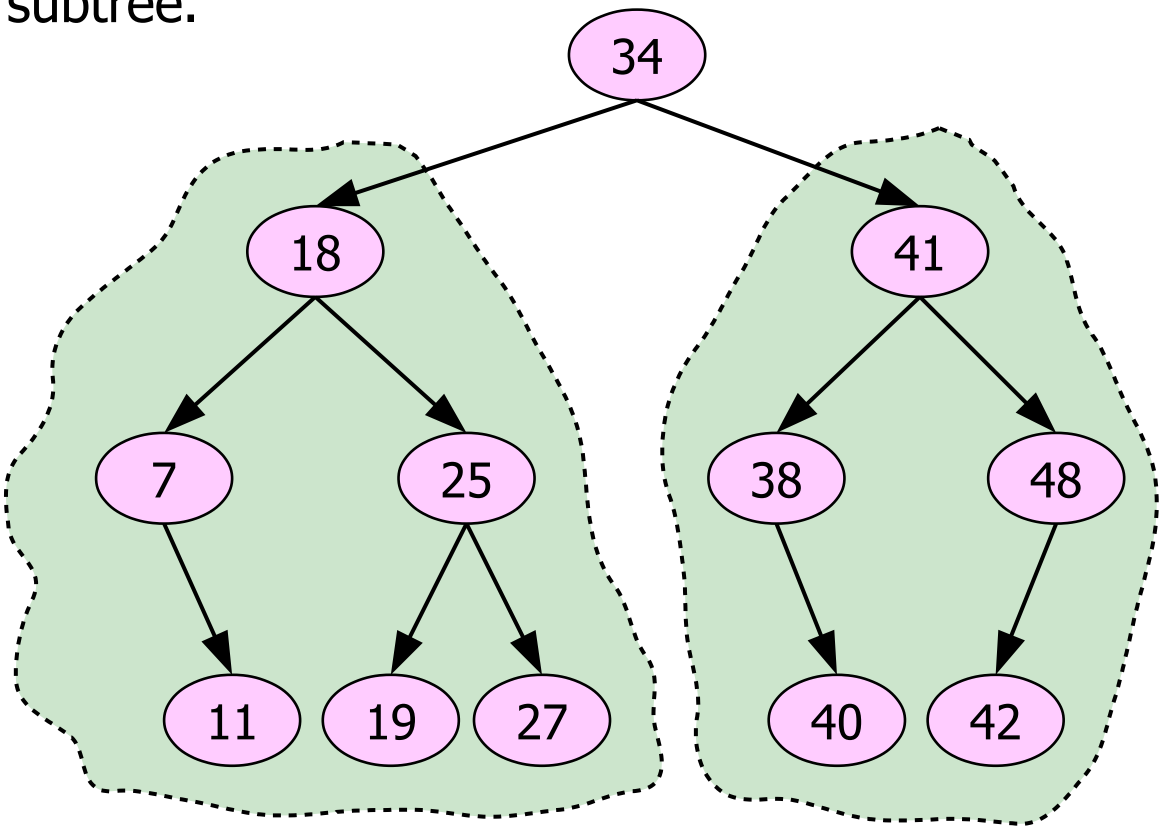
Traversals

A native traversal method is "in-order" traversal, which:

1) recursively traverses left subtree,

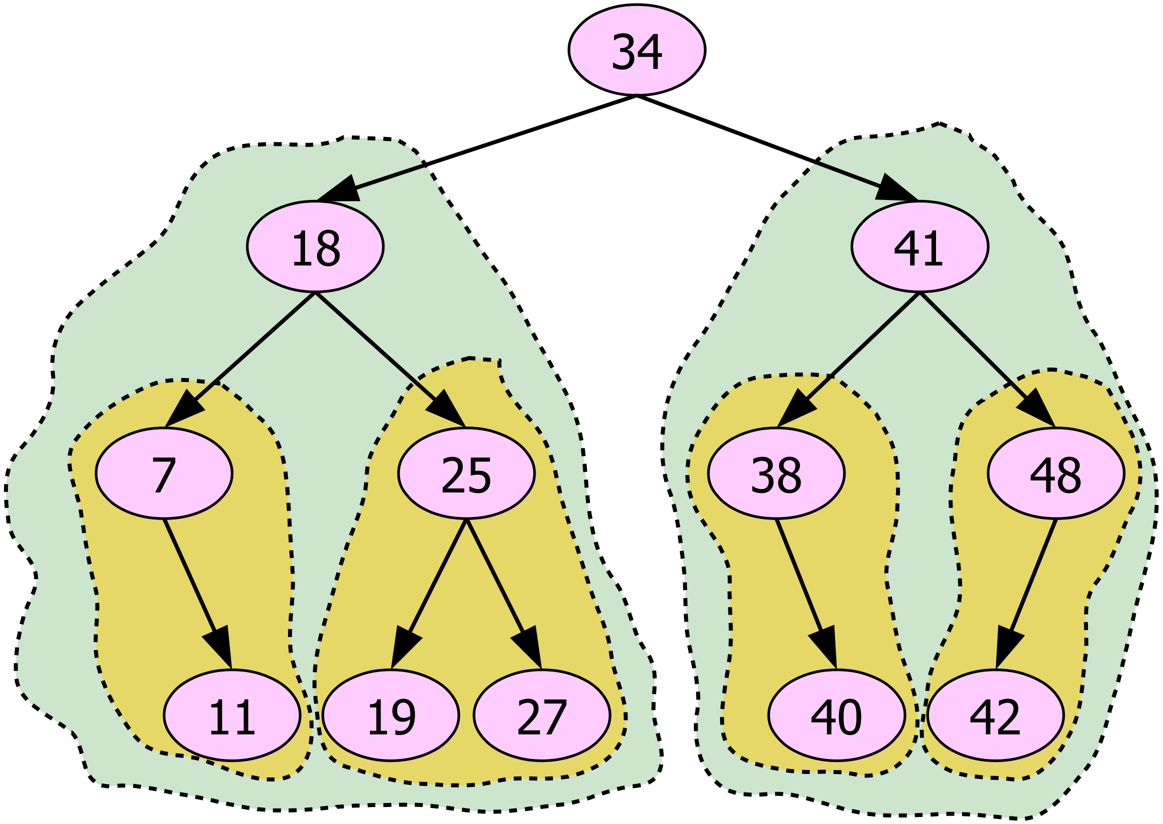
2) mentions current value,

3) recursively traverses right subtree.



Traversals

Applying this algorithm will <u>result in increasing sequence</u> of values.



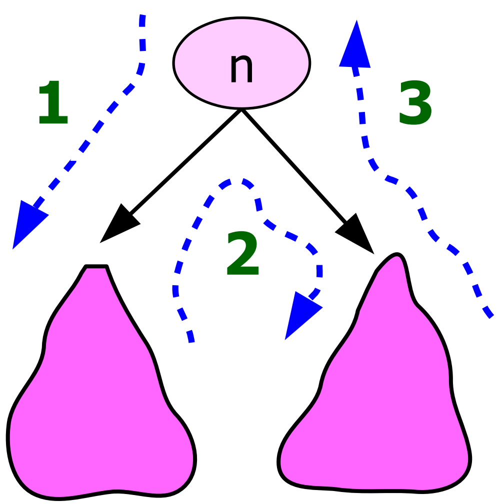
Traversals

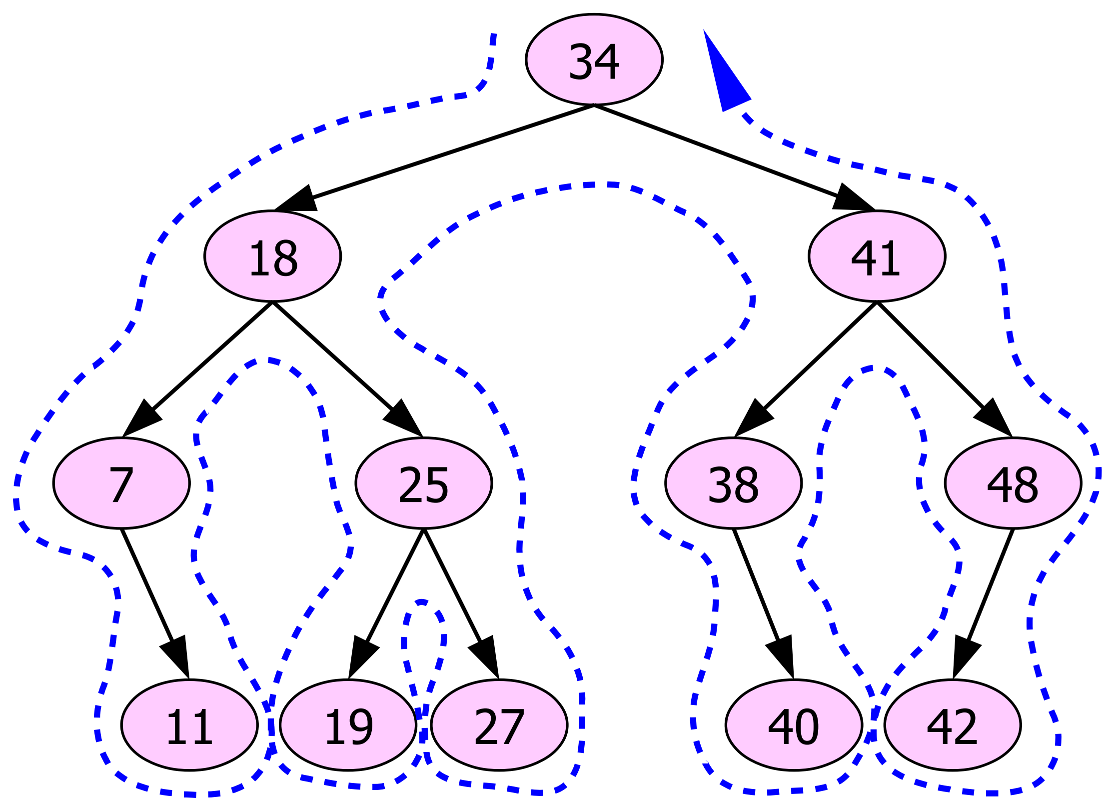
Recursive implementation turns out quite short:

```
procedure inOrderTraversal( n: Node )
    if n->left != null
        inOrderTraversal( n->left )
    print n->value
    if n->right != null
        inOrderTraversal( n->right )
```

Traversals

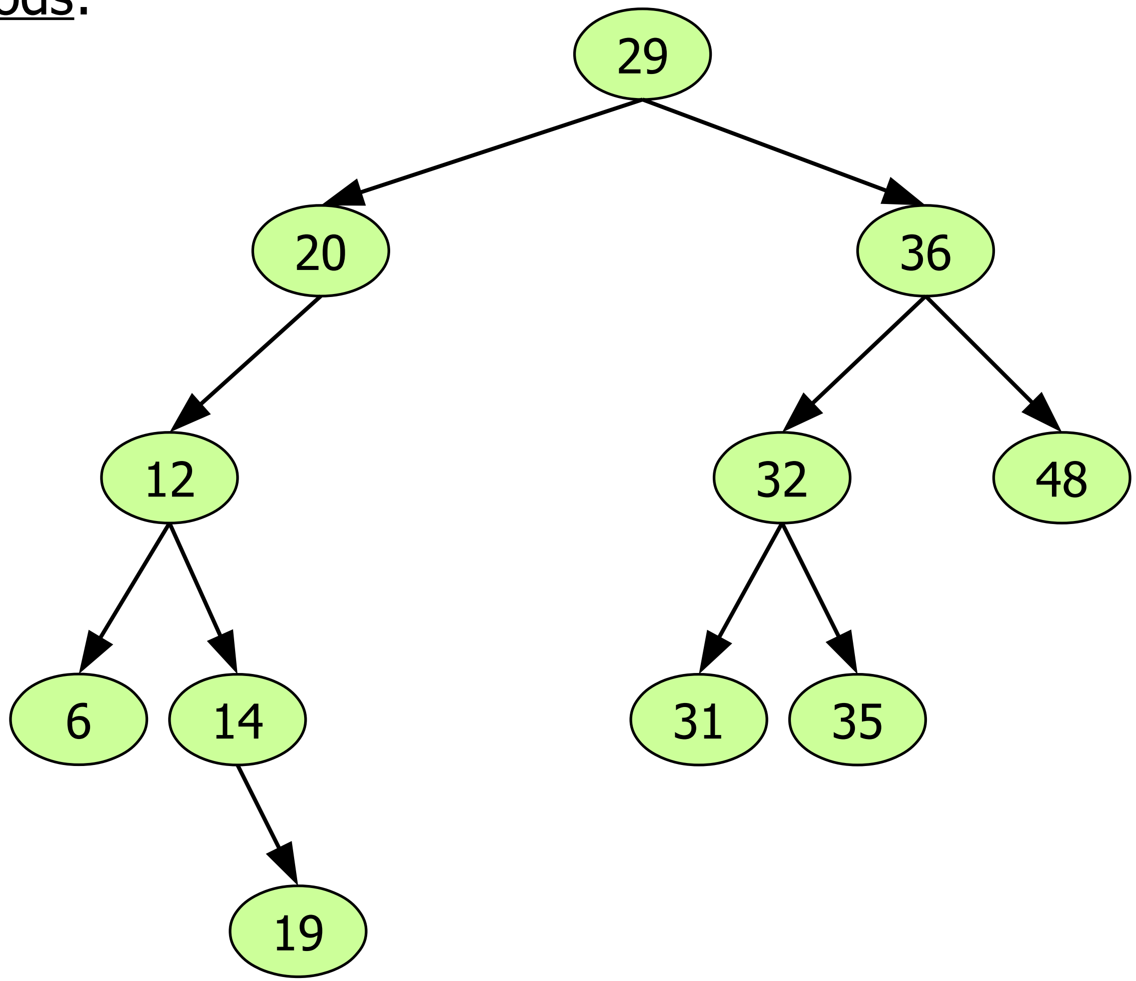
Another way to interpret the inorder-traversal is <u>with help of the following curve</u>:





Exercise

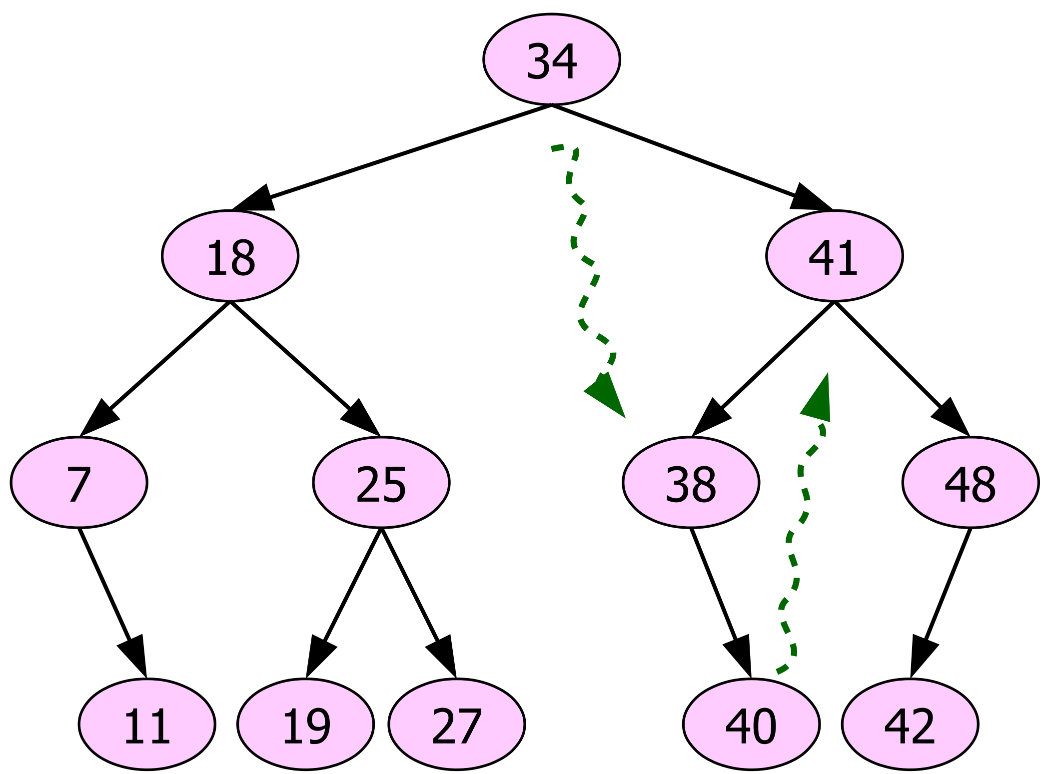
Perform in-order traversal on the following tree, by <u>both methods</u>:



Traversals

But recursive implementation is not always convenient to use. Maybe we want to:

- move just **1** step forward,
- move only few steps forward.



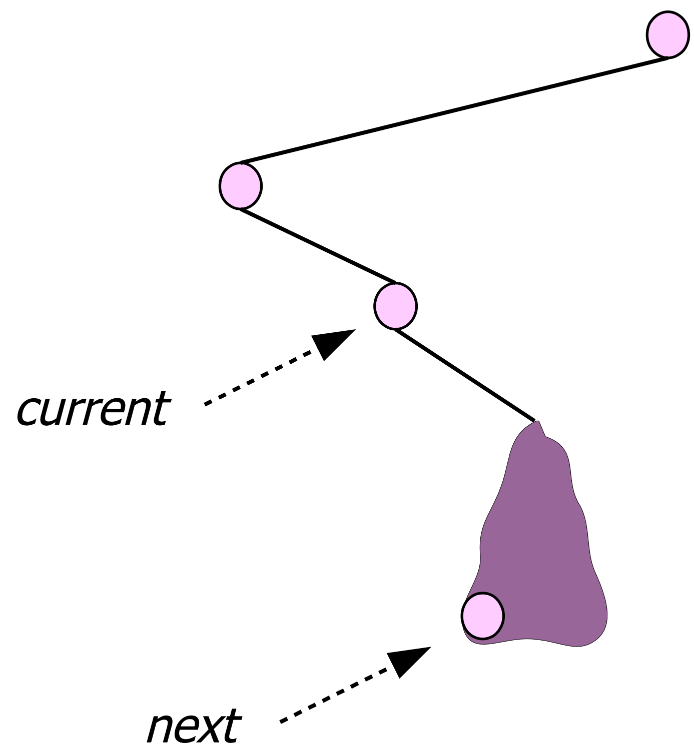
Traversals

This is when we need the iterative implementation.

So we want a method which will <u>move to the in-order next node</u>.

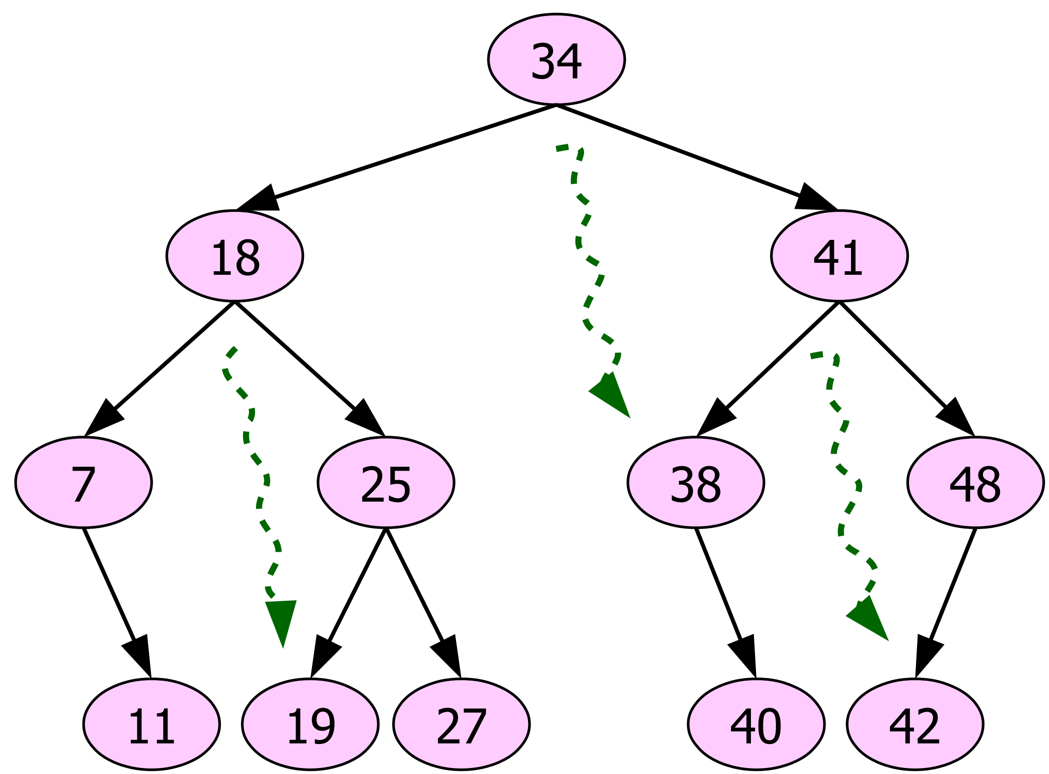
Case 1) Current node has right subtree,

... we move to minimal value of that subtree.



Traversals

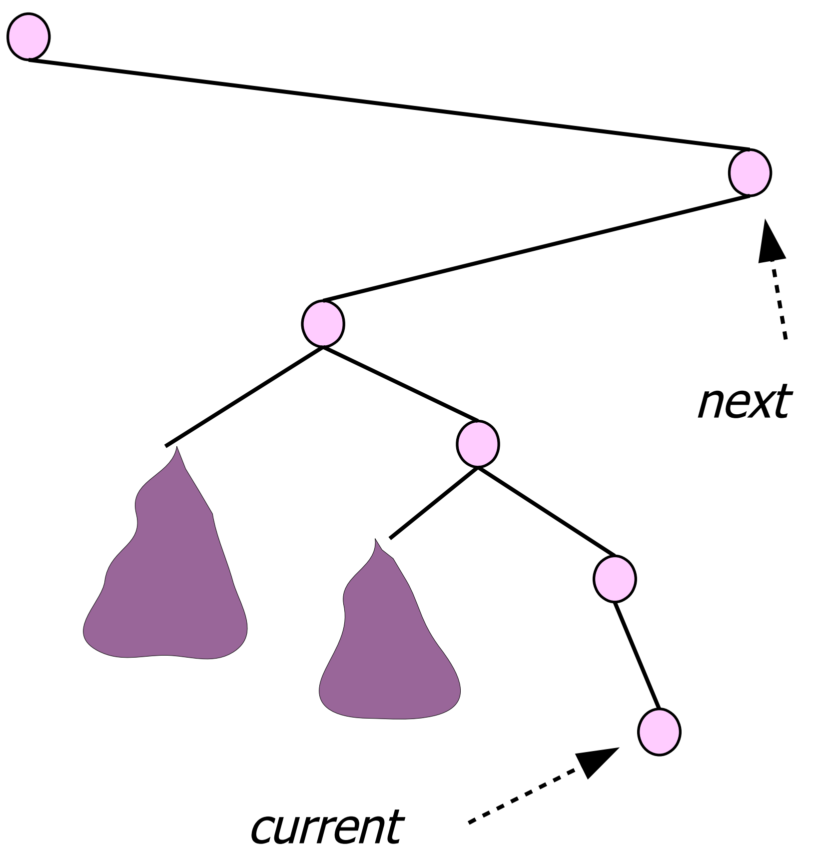
We can easily check it on the real example.



Traversals

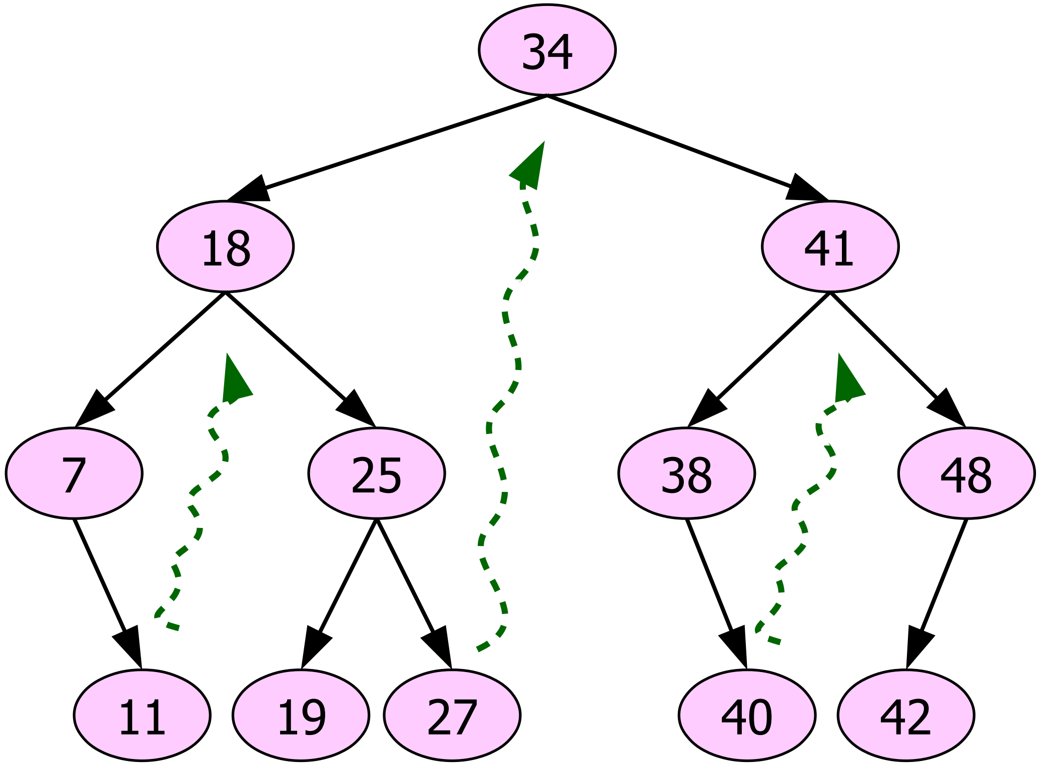
Case 2) Current node has no right subtree,

... we move up as long as we return from right subtree.



Traversals

Again, let's check it on the real example.



Traversals

Putting this into code:

```
function inorderNext( n: Node ) : Node
    if n->right != null  // We have right subtree
        n := n->right
        while n->left != null
            n := n->left
        return n
    else  // We have no right subtree
        while n->parent != null and n == n->parent->right
            n := n->parent
        return n->parent
```

Traversals

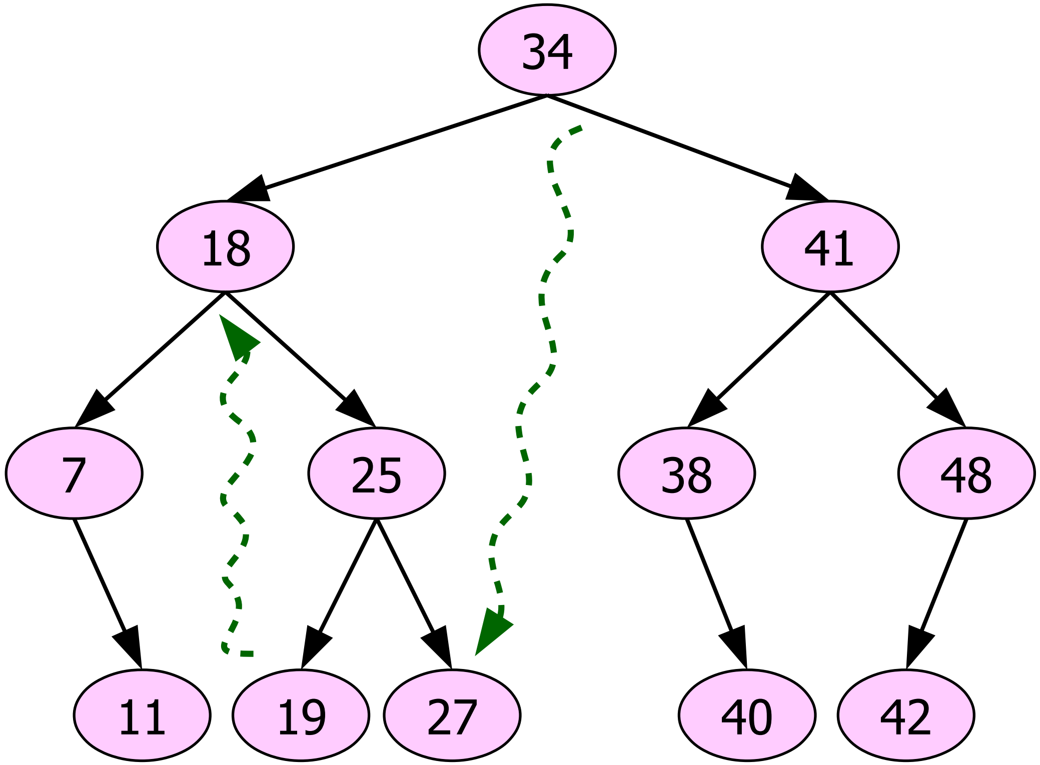
That is the way how iterators of trees are implemented in STL:

```
std::set< int > s;
...
std::set< int >::iterator it = s.begin();
++it;
++it;
std::cout << *it;
--it;
...
```

Exercises

1) Describe the logic of finding <u>in-order predecessor</u>.

2) Write it's code.

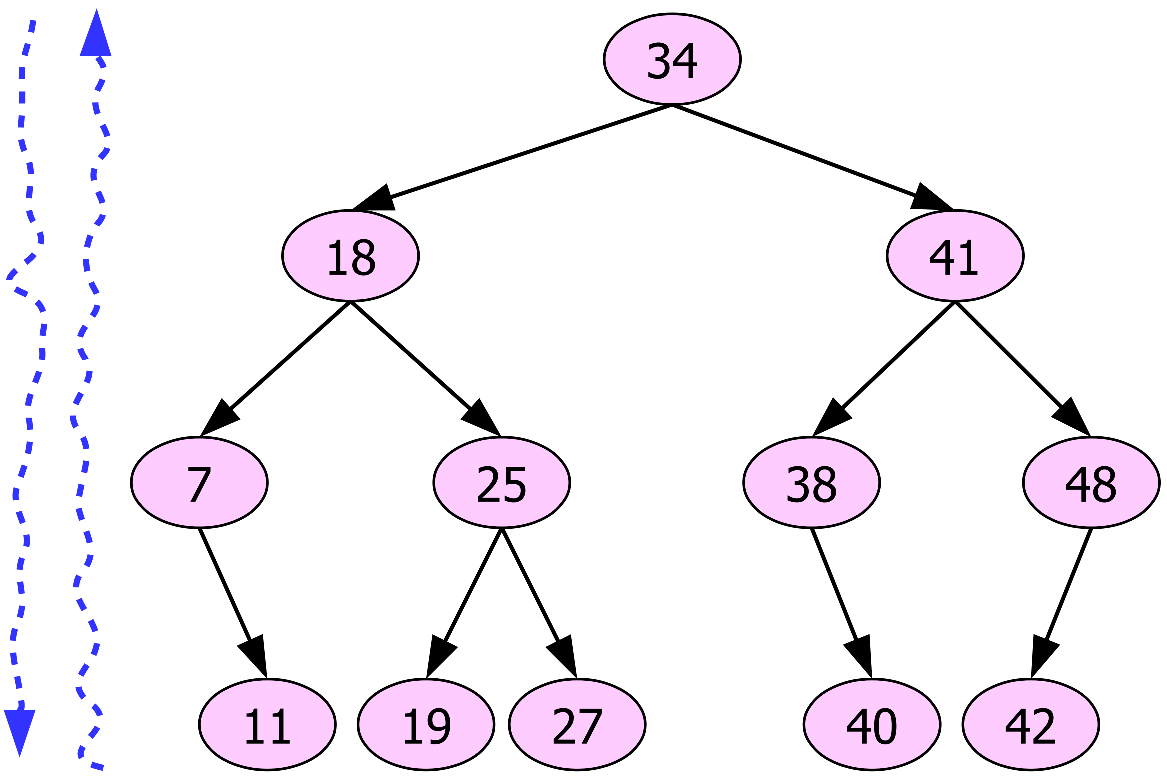


Traversals

<u>Worst case time complexity</u> of finding inorder next/previous is:

O(h) = O(logN)

... because we either move down or up the tree.



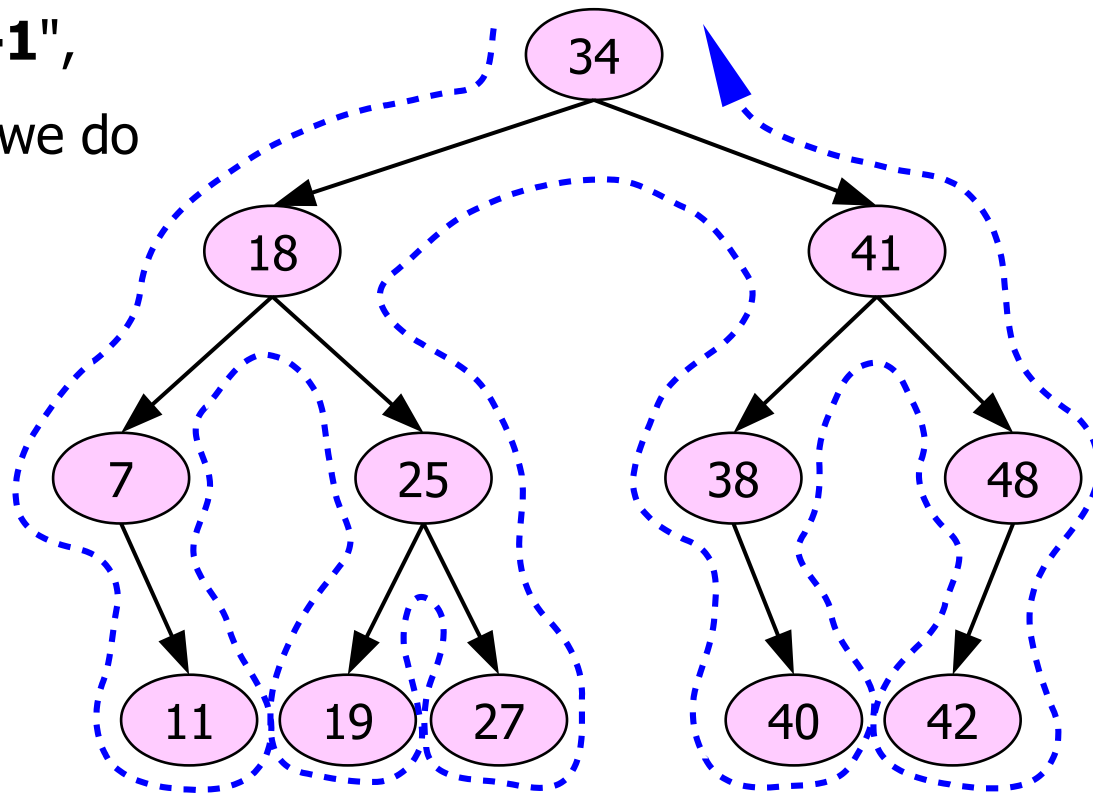
Traversals

Question: What is the average time complexity of that operations?

Traversals

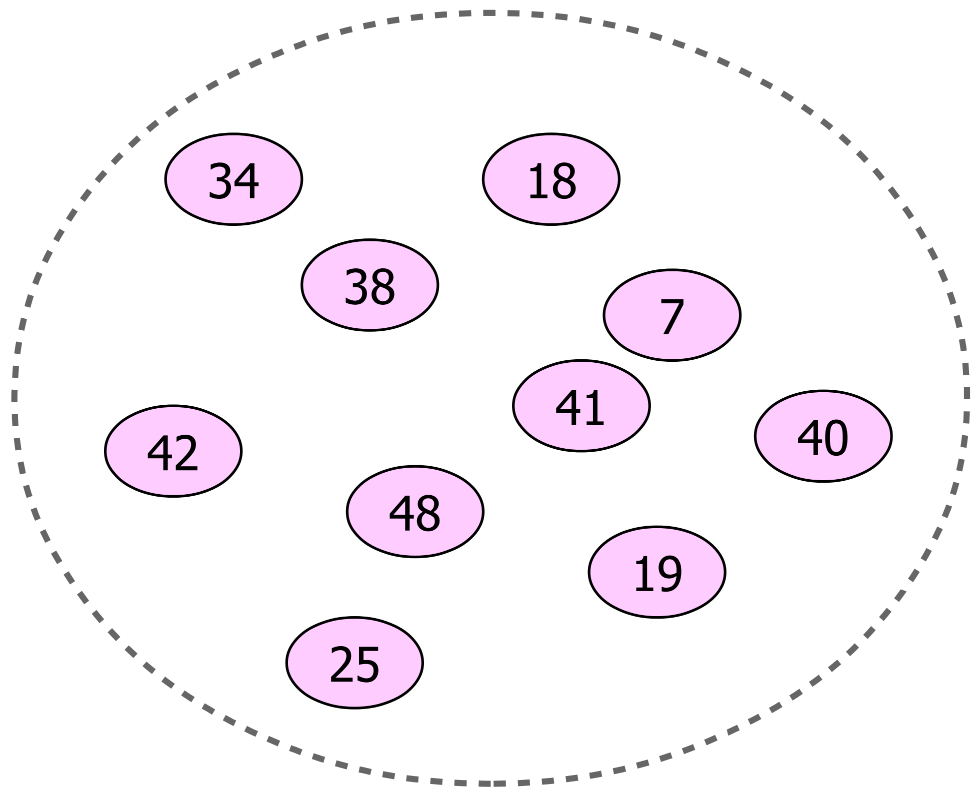
Answer: Let's consider we do a complete traversal.

- So all **N** values of the tree will be visited,
- Every edge will be addressed twice then,
- And number of edges is "**N-1**",
- So to travel along **N** values we do "**2(N-1)**" elementary steps,
- Which means that traveling to one next values will take **2 = O(1)** elementary steps on average.



Batch construction

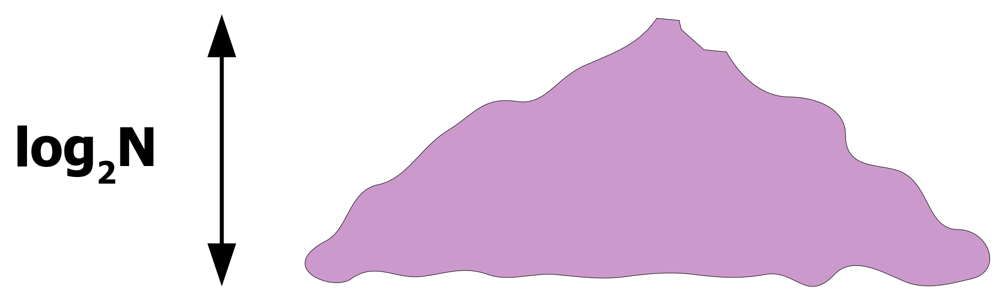
Batch construction means constructing <u>optimal data structure</u> from given in advance set of values.



Batch construction

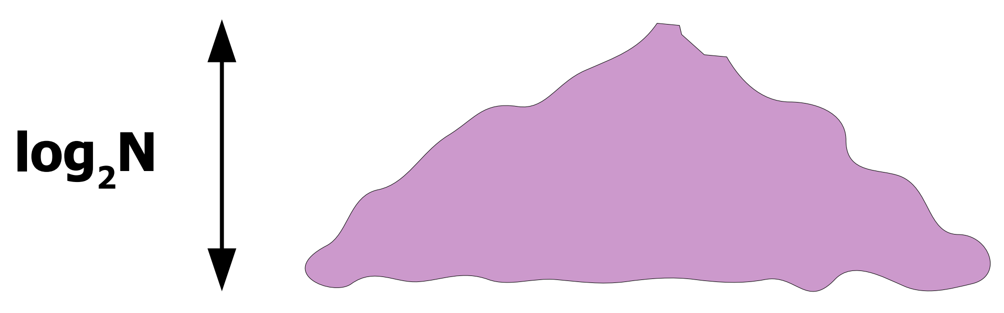
When it comes to batch constructing Binary search tree, we want to construct a tree with minimal height:

$h \sim \log_2 N$



Batch construction

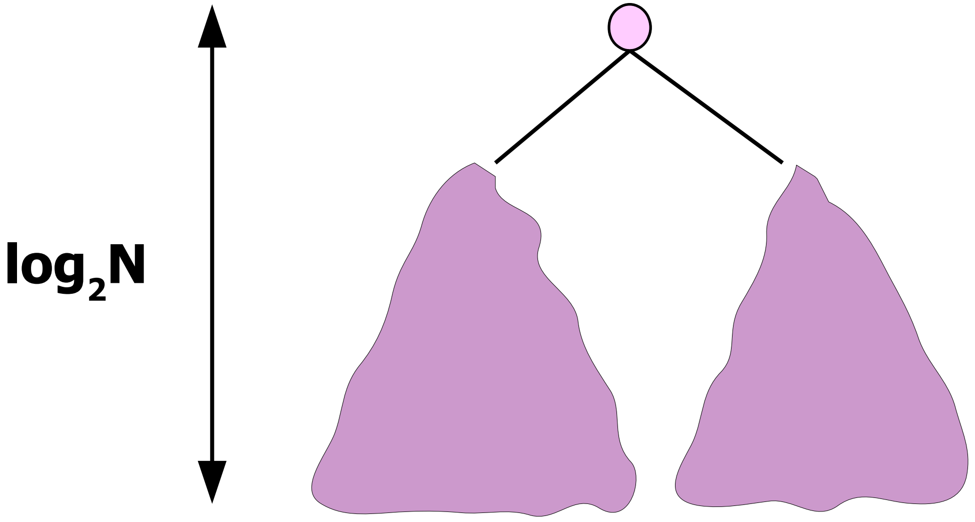
Question: How can we implement batch construction for Binary search tree?



Batch construction

Answer: Balanced Binary search tree means that its left and right subtrees are of almost equal sizes.

... which means that median should be choosen as root.

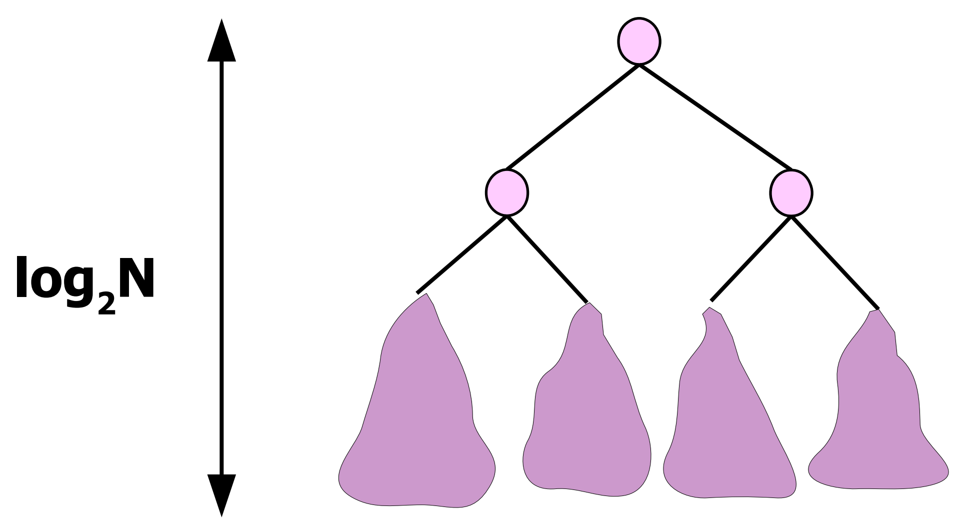


Batch construction

Answer: Balanced Binary search tree means that its left and right subtrees are of almost equal sizes.

... which means that median should be choosen as root.

... this refers to subdividing any subtree.



Exercises

1) Implement batch construction algorithm.
2) Derive its time complexity.

Presentation writer: Tigran Hayrapetyan

Lecturer | Programmer | Researcher

www.linkedin.com/in/tigran-hayrapetyan-cs/

Thank you!