Binary tree

Raimondas Čiegis

Matematinio modeliavimo katedra, e-paštas: rc@vgtu.lt

Spalio 16 d., 2023

Binary tree

First we present a general definition of the binary tree data structure.

Binary tree

First we present a general definition of the binary tree data structure.

Let's assume that a set D of elements is given.

Binary tree

First we present a general definition of the binary tree data structure.

Let's assume that a set D of elements is given.

A set of binary trees is recursively defined as:

- the empty set is a binary tree;

Binary tree

First we present a general definition of the binary tree data structure.

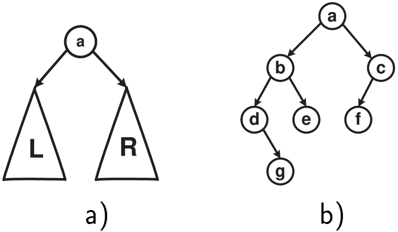
Let's assume that a set D of elements is given.

A set of binary trees is recursively defined as:

- the empty set is a binary tree;
- one vertex $a \in D$ is a binary tree;

- if L and R are binary trees, then denote by $T := L * R$ the binary tree obtained by adding an element $a \in D$ connected to the left to L and to the right to R, by adding edges when these sub-trees are non-empty (see a) part of the figure).

  Then a is the root of a tree, L and R are left and right sub-trees of the tree T.



An example of a binary tree.

We denote vertices of tree T as $v_j \in V$.

We denote vertices of tree T as $v_j \in V$.

If the vertex v_j is connected to v_k by the edge $e_{jk} = (v_j, v_k) \in E$, then the vertex v_k is called a child of v_j, and v_j is called a parent of v_k.

We denote vertices of tree T as $v_j \in V$.

If the vertex v_j is connected to v_k by the edge $e_{jk} = (v_j, v_k) \in E$, then the vertex v_k is called a child of v_j, and v_j is called a parent of v_k.

A vertex of a tree is called a leaf if it has no children.

We denote vertices of tree T as $v_j \in V$.

If the vertex v_j is connected to v_k by the edge $e_{jk} = (v_j, v_k) \in E$, then the vertex v_k is called a child of v_j, and v_j is called a parent of v_k.

A vertex of a tree is called a leaf if it has no children.

The level of a vertex v_j in a tree T is the length of the unique path from the root to this vertex.

We denote vertices of tree T as $v_j \in V$.

If the vertex v_j is connected to v_k by the edge $e_{jk} = (v_j, v_k) \in E$, then the vertex v_k is called a child of v_j, and v_j is called a parent of v_k.

A vertex of a tree is called a leaf if it has no children.

The level of a vertex v_j in a tree T is the length of the unique path from the root to this vertex.

The height of a rooted tree is the maximum of the levels of vertices.

We denote vertices of tree T as $v_j \in V$.

If the vertex v_j is connected to v_k by the edge $e_{jk} = (v_j, v_k) \in E$, then the vertex v_k is called a child of v_j, and v_j is called a parent of v_k.

A vertex of a tree is called a leaf if it has no children.

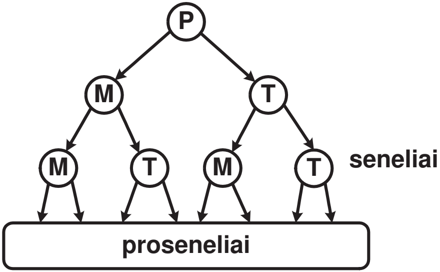
The level of a vertex v_j in a tree T is the length of the unique path from the root to this vertex.

The height of a rooted tree is the maximum of the levels of vertices.

We can use a recursive definition. The level of the root vertex is equal to zero.
Then the level of kth-level vertex's children is equal to $(k+1)$.

Family tree

A family tree, also called a genealogy chart, is a chart representing family relationships in a conventional tree structure (parents, grandparents and great grandparents).



Infix form of mathematical expression saved in a binary tree

Let's consider the following mathematical expression

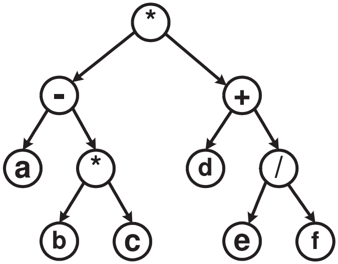
$$(a - b * c) * (d + e/f).$$

Infix form of mathematical expression saved in a binary tree

Let's consider the following mathematical expression

$$(a - b * c) * (d + e/f).$$

We put it into a binary tree

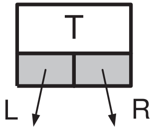


First, we define a basic node (element), it has a value field as well as two links (pointers), they point to the left and right sub-trees.

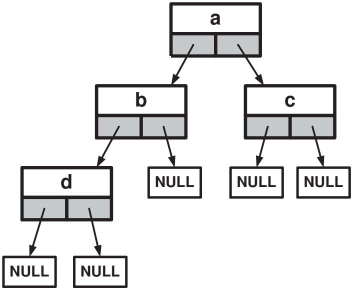
$$
\begin{aligned}
& struct\ node\,\{ \\
& \quad T\ data; \\
& \quad node *\ left; \\
& \quad node *\ right; \\
& \}
\end{aligned}
$$

First, we define a basic node (element), it has a value field as well as two links (pointers), they point to the left and right sub-trees.

$$\begin{aligned}
&struct\ node \{ \\
&\quad T\ data; \\
&\quad node *\ left; \\
&\quad node *\ right; \\
&\}
\end{aligned}$$



A binary tree is a nonlinear structure of nodes, when one node is connected to two or less nodes.



For many important applications it is convenient to add one additional field and one pointer to the structire of the basic node.

This new field is used to store a key.

An additional link points to the parent of the node.

$$
\begin{aligned}
& struct\ node \{ \\
& \quad T\ data; \\
& \quad int\ key; \\
& \quad node *\ left; \\
& \quad node *\ right; \\
& \quad node *\ p; \\
& \}
\end{aligned}
$$

As for all data structures we define the following main methods:

a) Insert a new node;

b) Remove a node from the tree;

c) Check if a node with a given key exists;

As for all data structures we define the following main methods:

a) Insert a new node;

b) Remove a node from the tree;

c) Check if a node with a given key exists;

In the case of binary trees we add two more methods, they are very useful in inplementation of search trees:

d) MINIMUM – finds a node with a smallest key value;

e) SUCCESSOR – finds a next element in a sorted set of elements.

A fully balanced binary tree

It is a general rule that a complexity of most important algorithms depends oh the height of a tree.

Thus our aim is to control/minimize the growth of the height, when the total number of elements is increasing.

A fully balanced binary tree

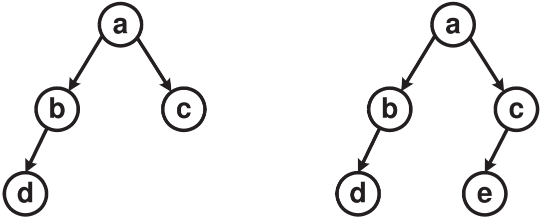
It is a general rule that a complexity of most important algorithms depends oh the height of a tree.

Thus our aim is to control/minimize the growth of the height, when the total number of elements is increasing.

It is important to regulate the construction of left and right subtrees, when new elements are added or removed.

Definition

A tree is called a fully balanced if for any vertex total numbers of elements in the right and left subtrees differ by at most one.



a) b)

Fully balanced binary trees: a) $N = 4$, b) $N = 5$

Next we present a recursive algorithm for construction of a fully balanced binary tree.

$$
\begin{aligned}
& node * \ balancedTree(int\ N)\{ \\
& \quad y = NIL; \\
& \quad if\ (N == 0)\ return\ (NIL); \\
& \quad nL = N/2;\ nR = N - nL - 1; \\
& \quad x = read(); \\
& \quad node * \ Node = new(node); \\
& \quad Node.data = x; \quad Node.key = key; \\
& \quad Node.left = balancedTree(nL); \\
& \quad Node.right = balancedTree(nR); \\
& \quad Node.p = y, \quad y = Node; \\
& \quad return\ (Node); \\
& \}
\end{aligned}
$$

A task to visit all vertices of a binary tree

We consider three important algorithms, they define different orders how vertices of left and right sub-trees are visited.

A task to visit all vertices of a binary tree

We consider three important algorithms, they define different orders how vertices of left and right sub-trees are visited.

Again all three algoritms are recursive.

A task to visit all vertices of a binary tree

We consider three important algorithms, they define different orders how vertices of left and right sub-trees are visited.

Again all three algoritms are recursive.

If a mathematical expression is stored in a tree, then these algorithms can print the prefix, infix and postfix forms of the given mathematical expression.

Prefix algorithm

First we visit a root, then vertices of a left sub-tree and finally a right-subtree.

```
preOrder (node* v)
begin
  (1)  if  (v ≠ NIL ) then
  (2)        P(v);
  (3)        preOrder(v.left);
  (4)        preOrder(v.right);
       end if
end preOrder
```

Infix algorithm

First we visit vertices of a left sub-tree, then a root, and finally vertices of a right-subtree.

inOrder (node* v)
begin
(1) **if** (v $\neq$ NIL) **then**
(2) inOrder(v.left);
(3) P(v);
(4) inOrder(v.right);
end if
end inOrder

Postfix algorithm

First we visit vertices of a left sub-tree, then a right-subtree and finally a root.

postOrder (node* v)
begin
(1) **if** (v $\neq$ NIL) **then**
(2) postOrder(v.left);
(3) postOrder(v.right);
(4) P(v);
end if
end postOrder

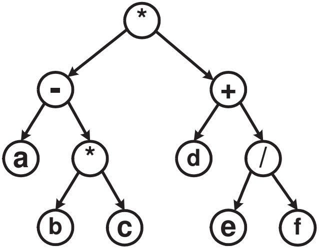
Let us print three different forms of the mathematical expression

$$(a - b * c) * (d + e/f).$$

Let us print three different forms of the mathematical expression

$$(a - b * c) * (d + e/f).$$

First we write it into the binary tree:



$$(a - b * c) * (d + e/f).$$

$$(a - b * c) * (d + e/f).$$

Prefix: $\quad * - a * bc + d/ef,$

$$(a - b * c) * (d + e/f).$$

Prefix: $* - a * bc + d/ef,$

Infix: $a - b * c * d + e/f,$

$$(a - b * c) * (d + e/f).$$

Prefix: $* - a * bc + d/ef,$

Infix: $a - b * c * d + e/f,$

Postfix: $abc * -def / + *.$

Binary search tree

Data sorting and searching are probably the most important operation in data analysis.

Binary search trees are helping to solve these problems efficiently.

Binary search tree

Data sorting and searching are probably the most important operation in data analysis.

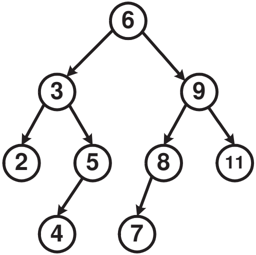
Binary search trees are helping to solve these problems efficiently.

Definition. A binary search tree is a data structure with the key of each internal node being greater than all the keys in the respective node's left subtree and equal or less than the ones in its right subtree.

Data sorting and searching are probably the most important operation in data analysis.

Binary search trees are helping to solve these problems efficiently.

Definition. A binary search tree is a data structure with the key of each internal node being greater than all the keys in the respective node's left subtree and equal or less than the ones in its right subtree.



Let us see how binary search trees solve efficiently some important problems.

Let us see how binary search trees solve efficiently some important problems.

Apply InOrder(T.root) method to visit all vertices of a binary search tree T. What conclusion follows from such an experiment?

Let us see how binary search trees solve efficiently some important problems.

Apply InOrder(T.root) method to visit all vertices of a binary search tree T. What conclusion follows from such an experiment?

Yes, we print elements in a sorted order, starting with the smallest one and moving to the element with the largest key.

Let us see how binary search trees solve efficiently some important problems.

Apply InOrder(T.root) method to visit all vertices of a binary search tree T. What conclusion follows from such an experiment?

Yes, we print elements in a sorted order, starting with the smallest one and moving to the element with the largest key.

Try to prove that this result is valid for any sorted tree.

Next we solve a problem, which is very popular in applications.

In a binary search tree we want to find an element with a key k.

If such element do not exist, then the procedure returns a link to *NIL*.

Next we solve a problem, which is very popular in applications.

In a binary search tree we want to find an element with a key k.

If such element do not exist, then the procedure returns a link to *NIL*.

Our idea

Next we solve a problem, which is very popular in applications.

In a binary search tree we want to find an element with a key k.

If such element do not exist, then the procedure returns a link to *NIL*.

Our idea

Start searching from the root node.

Then if the data is less than the key value, search for the element in the left subtree.

Otherwise, search for the element in the right subtree.

Follow the same algorithm for each node.

$$
\begin{aligned}
&node * \ \ Tree\_Search(node * x, \ int \ k)\{ \\
&\quad if \ (x == NILL \,||\, k == x.key) \ return \ x; \\
&\quad if \ (k < x.key) \ return \ Tree\_Search(x.left, \ k); \\
&\quad else \\
&\qquad return \ Tree\_Search(x.right, \ k); \\
&\}
\end{aligned}
$$

$$
\begin{aligned}
&node * \;\; Tree\_Search(node * x, \;\; int \;\; k)\{ \\
&\quad if \;\; (x == NILL \,||\, k == x.key) \;\; return \;\; x; \\
&\quad if \;\; (k < x.key) \;\; return \;\; Tree\_Search(x.left, \;\; k); \\
&\quad else \\
&\quad\quad return \;\; Tree\_Search(x.right, \;\; k); \\
&\}
\end{aligned}
$$

The structure of this new algorithm is similar to algorithms used to visit all nodes of any binary tree.

In both cases algorithms are recursive.

$$
\begin{array}{l}
node * \ Tree\_Search(node * x, \ int \ k)\{ \\
\quad if \ (x == NILL \ || \ k == x.key) \ return \ x; \\
\quad if \ (k < x.key) \ return \ Tree\_Search(x.left, \ k); \\
\quad else \\
\qquad return \ Tree\_Search(x.right, \ k); \\
\}
\end{array}
$$

The structure of this new algorithm is similar to algorithms used to visit all nodes of any binary tree.

In both cases algorithms are recursive.

Still for the search algorithm at each step we select to visit only one subtree. The second subtree is never visited.

```
node * Tree_Search(node * x, int k){
   if (x == NILL || k == x.key) return x;
   if (k < x.key) return Tree_Search(x.left, k);
   else
      return Tree_Search(x.right, k);
}
```

The structure of this new algorithm is similar to algorithms used to visit all nodes of any binary tree.

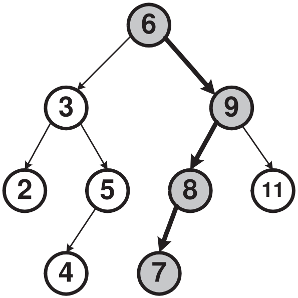
In both cases algorithms are recursive.

Still for the search algorithm at each step we select to visit only one subtree. The second subtree is never visited.

Therefore the complexity of the search algorithm depends only on the height of a binary tree $O(h)$.

We present also an iterative version of the search algorithm. In many cases it can be more efficient than the recursive version.

$$
\begin{array}{l}
node * \ \ Tree\_Iterative\_Search(node * x, \ int \ k)\{ \\
\quad \text{while } (\, x \neq \ \text{NILL} \ \ \text{and} \ \ k \neq x.key \,) \\
\qquad \text{if } (k < x.key) \\
\qquad\quad x = x.left; \\
\qquad \text{else} \\
\qquad\quad x = x.right; \\
\quad return \ x; \\
\}
\end{array}
$$



Searching for a node with a key value 7. Search algorithm traverses the tree "in-depth", choosing an appropriate way (a left or right subtree).

Test Problems

Make iterative algorithms to implement the following methods of binary search trees:

Tree_Minimum_Search(node* T.root),

Tree_Maximum_Search(node* T.root).

Test Problems

Make iterative algorithms to implement the following methods of binary search trees:

Tree_Minimum_Search(node* T.root),

Tree_Maximum_Search(node* T.root).

Do we need to compare keys of elements?

Insert a new node

We want to insert into binary search tree T a new vertex v. The initial values of all fields are the following:

$$v.key = k, \;\; v.left = NIL, \;\; v.right = NIL, \;\; v.p = NIL.$$

We should guarantee that after insertion the tree T remains to be a binary search tree.

Insert a new node

We want to insert into binary search tree T a new vertex v. The initial values of all fields are the following:

$$v.key = k, \;\; v.left = NIL, \;\; v.right = NIL, \;\; v.p = NIL.$$

We should guarantee that after insertion the tree T remains to be a binary search tree.

A given insertion algorithm traverses the tree "in-depth", choosing an appropriate way (a left or right subtree) till a NIL pointer is reached.

Insert a new node

We want to insert into binary search tree T a new vertex v. The initial values of all fields are the following:

$$v.key = k, \quad v.left = NIL, \quad v.right = NIL, \quad v.p = NIL.$$

We should guarantee that after insertion the tree T remains to be a binary search tree.

A given insertion algorithm traverses the tree "in-depth", choosing an appropriate way (a left or right subtree) till a NIL pointer is reached.

Then a new leaf node is added to the tree T at this link.

Insert a new node

We want to insert into binary search tree T a new vertex v. The initial values of all fields are the following:

$$v.key = k, \quad v.left = NIL, \quad v.right = NIL, \quad v.p = NIL.$$

We should guarantee that after insertion the tree T remains to be a binary search tree.

A given insertion algorithm traverses the tree "in-depth", choosing an appropriate way (a left or right subtree) till a NIL pointer is reached.

Then a new leaf node is added to the tree T at this link.

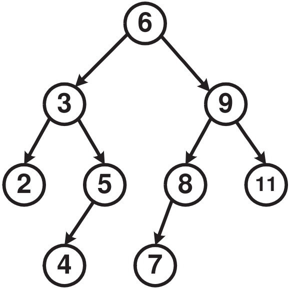
If the tree T is empty, then this new vertex bacomes a root of T.

```
insert (tree T, node* v)
    y = NIL,   x = T.root
    while  ( x ≠ NIL )
               y = x
               if ( v.key < x.key )
                  x = x.left
               else   x = x.right
    v.p = y
    if (y == NIL)
               T.root = v
    elseif (v.key < y.key)
               y.left = v
    else   y.right = v
```

Example 1

Insert the given elements into a new binary search tree:

6, 9, 3, 5, 11, 4, 8, 2, 7.



Delete a vertx

It is more difficult to delete a vertex from T, since the obtained tree should remain a binary search tree.

We consider three separate cases:

We consider three separate cases:

1. A leaf node v should be deleted. Then an appropriate link of v parent $w = v.p$ gets the value NIL:

$$
\begin{aligned}
& w = v.p \\
& \text{if } (T.root == v)\ T.root = NIL \\
& \text{else} \\
& \quad \text{if } (v.key < w.key)\ w.left = NIL \\
& \quad \text{else } w.right = NIL \\
& delete(v)
\end{aligned}
$$

2. We want to delete a vertex v, which has one child. Then an appropriate link of the parent $w = v.p$ points to the child of v:

$$w = v.p$$
$$\text{if } (v.left == NIL) \ c = v.right$$
$$\text{else } c = v.left$$
$$\text{if } (T.root == v) \ T.root = c$$
$$\text{else}$$
$$\quad c.p = w$$
$$\quad \text{if } (v.key < w.key) \ w.left = c$$
$$\quad \text{else } w.right = c$$
$$delete(v)$$

3. We want to delete a vertex v, which has two children.

3. We want to delete a vertex v, which has two children.

The first algorithm finds the smallest element in the right sub-tree of v:

$$node * \; Tree\_Right\_Minimum(node * x)\{$$
$$y = x.right$$
$$s = y$$
$$\text{while } (y \neq \text{ NILL })$$
$$s = y$$
$$y = y.left$$
$$return\ s$$
$$\}$$

Then a vertex v is deleted by applying the following algorithm (first we check if the smallest element defines the right side child of v):

$$s = Tree\_Right\_Minimum(v)$$

$$w = v.p, \; z = s.p$$

$$\text{if } (z \neq v)$$

$$\quad z.left = s.right$$

$$\quad s.right = v.right$$

$$s.left = v.left$$

$$\text{if } (v.key < w.key)$$

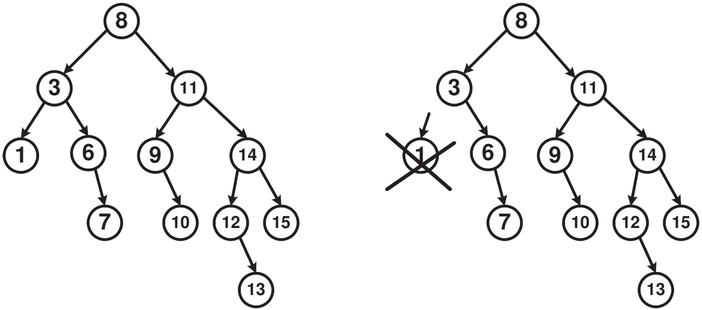
$$\quad w.left = s$$

$$\text{else}$$

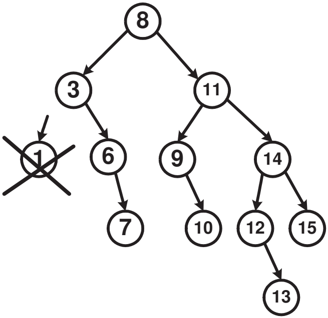
$$\quad w.right = s$$

$$delete(v)$$

Let us delete a leaf vertex $v.key = 1$.



Next we delete a vertex which has one child $v.key = 6$.



Finally we delete a vertex with two children $v.key = 11$.