DATA STRUCTURES

Binary Search Trees

Lect. PhD. Diana-Lucia Miholca

2022 - 2023

Babeş - Bolyai University
Faculty of Mathematics and Computer Science

In Lecture 10...

- Binary trees (II)
  - Traversals

Today

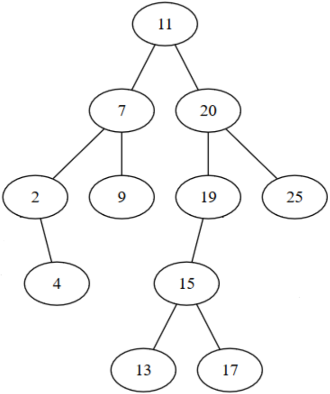
- Binary Search Trees

Binary search trees

A **binary search tree** is a binary tree that satisfies the following property:

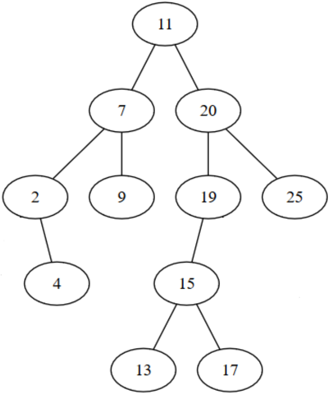
- Let x be a node in a binary search tree.
  - For every node y from the left subtree of x, the information from y is less than or equal to the information from x
  - For every node y from the right subtree of x, the information from y is greater than or equal to the information from x

Binary Search Tree Example



Inorder:

Binary Search Tree Example



- Inorder: 2 4 7 9 11 13 15 17 19 20 25
- An inorder traversal of a binary search tree will visit the elements in increasing order.

Binary Search Tree - terminology

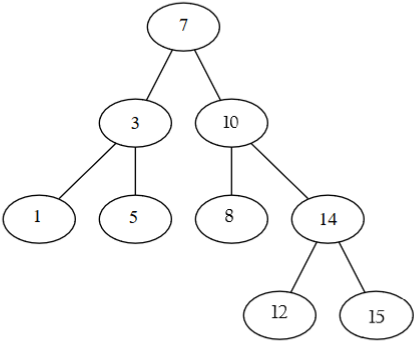
The terminology discussed for binary trees is valid for binary search trees as well:

We can have a binary search trees that are:

- full
- complete
- almost complete
- degenerated
- balanced

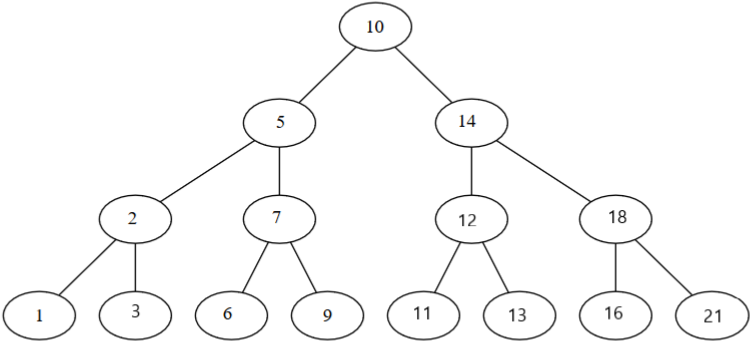
Binary tree - Terminology

A binary search tree is called **full** if every internal node has exactly two children.



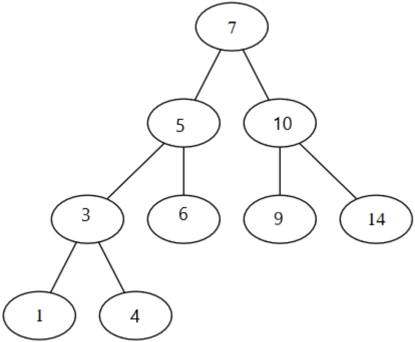
Binary tree - Terminology

A binary search tree is called **complete** if every level of the tree is completely filled.



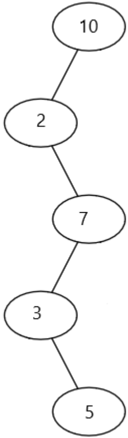
Binary tree - Terminology

A binary search tree is called **almost complete** if every level of the tree is completely filled, except possibly the bottom level, which is filled from left to right.



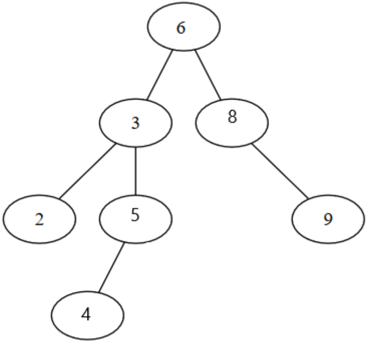
Binary tree - Terminology

A binary search tree is called **degenerated** if every internal node has exactly one child (it is actually a chain of nodes).



Binary tree - Terminology

A binary search tree is called **balanced** if the difference between the height of the left and right subtrees is at most 1 for every node from the tree.



Binary Search Tree

Binary search trees inherit the numerical properties of binary trees:

- The number of nodes in a complete binary search tree of height N is

Binary Search Tree

Binary search trees inherit the numerical properties of binary trees:

- The number of nodes in a complete binary search tree of height N is $2^{N+1} - 1$ (it is $1 + 2 + 4 + 8 + ... + 2^N$)
- The maximum number of nodes in a binary search tree of height N is

Binary Search Tree

Binary search trees inherit the numerical properties of binary trees:

- The number of nodes in a complete binary search tree of height N is $2^{N+1} - 1$ (it is $1 + 2 + 4 + 8 + ... + 2^N$)
- The maximum number of nodes in a binary search tree of height N is $2^{N+1} - 1$ - if the tree is complete.
- The minimum number of nodes in a binary search tree of height N is

Binary Search Tree

Binary search trees inherit the numerical properties of binary trees:

- The number of nodes in a complete binary search tree of height N is $2^{N+1} - 1$ (it is $1 + 2 + 4 + 8 + ... + 2^N$)
- The maximum number of nodes in a binary search tree of height N is $2^{N+1} - 1$ - if the tree is complete.
- The minimum number of nodes in a binary search tree of height N is $N + 1$ - if the tree is degenerated.
- A binary search tree with N nodes has a height between $[log_2(N + 1)]$ and $N - 1$.

Binary Search Tree

Binary search trees can be used as representation for **sorted** containers

- sorted maps
- priority queues
- sorted sets, etc.

Basic operations:

- searching for an element
- inserting an element
- removing an element

Binary Search Tree - other operations

Other operations:

- get the minimum element
- get the maximum element
- find the successor of an element
- find the predecessor of an element

Binary Search Tree - Representation

- A linked representation with dynamic allocation for binary search trees:

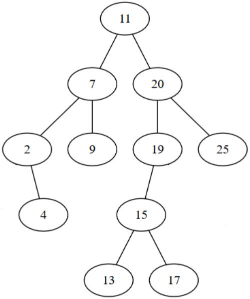
Representation of a node in a Binary Search Tree:

<u>BSTNode:</u>
    info: TComp
    left: ↑ BSTNode
    right: ↑ BSTNode

Representation of a Binary Search Tree:

<u>Binary Search Tree:</u>
    root: ↑ BSTNode

Binary Search Tree - search operation



How can we search for 15? What about the 14?

Binary Search Tree - search operation

How can we **search** for an element in a binary search tree?

The idea of searching for an element *elem*:

- We start at the root of the tree.
- For each node *node* encountered:
  - If $node = NIL \Rightarrow$ unsuccessful search
  - If $[node].info = elem \Rightarrow$ successful search
  - If $[node].info > elem \Rightarrow$ search recursively the left subtree
  - If $[node].info < elem \Rightarrow$ search recursively the right subtree

BST - search operation - recursive implementation

Recursively searching in a Binary Search Tree:

function search_rec (node, elem) **is:**

//pre: node is a BSTNode and elem is the TElem we are searching for

BST - search operation - recursive implementation

Recursively searching in a Binary Search Tree:

```
function search_rec (node, elem) is:
//pre: node is a BSTNode and elem is the TElem we are searching for
    if node = NIL then
        search_rec ← false
    else
        if [node].info = elem then
            search_rec ← true
        else if [node].info < elem then
            search_rec ← search_rec([node].right, elem)
        else
            search_rec ← search_rec([node].left, elem)
    end-if
end-function
```

BST - search operation - recursive implementation - complexity

The time complexity of the *search* operation is

BST - search operation - recursive implementation - complexity

The time complexity of the *search* operation is $O(h)$, where h is the height of the tree.

The maximum height of a tree with n nodes is

BST - search operation - recursive implementation - complexity

The time complexity of the *search* operation is $O(h)$, where h is the height of the tree.

The maximum height of a tree with n nodes is $n - 1$ (if the tree is degenerated).

Therefore, the time complexity of the *search* operation can also be expressed as $O(n)$.

.

BST - search operation - recursive implementation

We need a wrapper to call the recursive function with the root of the tree:

Recursive searching function - initial call

function search (tree, e) **is:**
//pre: tree is a Binary Search Tree, e is the elem we are looking for
search $\leftarrow$ search_rec(tree.root, e)
end-function

BST - search operation - non-recursive implementation

- The iterative implementation of the *search* operation:

Iteratively searching in a Binary Search Tree:

function search (tree, elem) **is:**
//pre: tree is a Binary Search Tree and elem is the TElem we are searching for

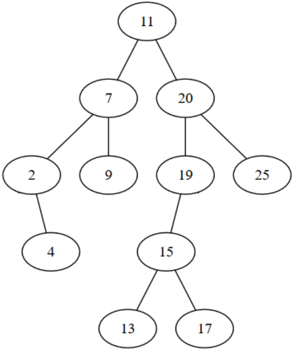
BST - search operation - non-recursive implementation

- The iterative implementation of the *search* operation:

Iteratively searching in a Binary Search Tree:

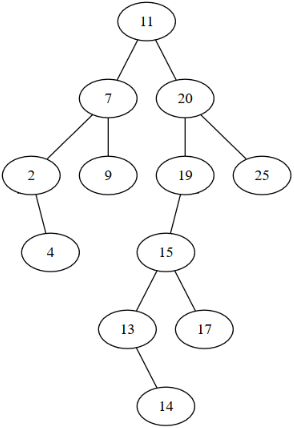
function search (tree, elem) **is:**
//pre: tree is a Binary Search Tree and elem is the TElem we are searching for
    currentNode ← tree.root
    found ← false
    while currentNode ≠ NIL and not found **execute**
        if [currentNode].info = elem **then**
            found ← true
        else if [currentNode].info < elem **then**
            currentNode ← [currentNode].right
        else
            currentNode ← [currentNode].left
        end-if
    end-while
    search ← found
end-function

BST - insert operation



Where can we insert element 14?

BST - insert operation



BST - insert operation - recursive implementation

- An function that creates a new node with a given element:

Creating a node with a given element:

function initNode(e) **is:**
//pre: e is a TComp
//post: initNode: $\uparrow$ BSTNode $\leftarrow$ a node with e as information
    allocate(newNode)
    [newNode].info $\leftarrow$ e
    [newNode].left $\leftarrow$ NIL
    [newNode].right $\leftarrow$ NIL
    initNode $\leftarrow$ newNode
end-function

BST - insert operation - recursive implementation

Recursively inserting in a Binary Search Tree::

function insert_rec(node, e) **is:**

//pre: node is a BSTNode, e is TComp

//post: a node containing e was added in the tree starting from node

BST - insert operation - recursive implementation

Recursively inserting in a Binary Search Tree::

function insert_rec(node, e) **is:**
//pre: node is a BSTNode, e is TComp
//post: a node containing e was added in the tree starting from node
    if node = NIL **then**
        node $\leftarrow$ initNode(e)
    else if [node].info $\geq$ e **then**
        [node].left $\leftarrow$ insert_rec([node].left, e)
    else
        [node].right $\leftarrow$ insert_rec([node].right, e)
    end-if
    insert_rec $\leftarrow$ node
end-function

BST - insert operation - recursive implementation

- The time complexity of the *insert* operation is

BST - insert operation - recursive implementation

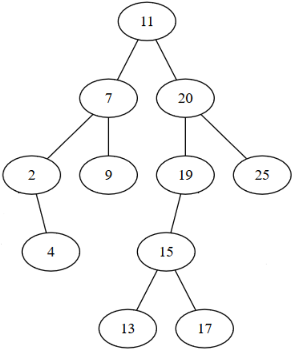
The time complexity of the *insert* operation is $O(h)$ (or $O(n)$)

We need a wrapper function to call *insert_rec* with the root of the tree:

Recursive insertion function - initial call:

function insert (tree, e) **is:**
//pre: tree is a Binary Search Tree, e is the elem to be inserted
    tree.root $\leftarrow$ insert_rec(tree.root, e)
end-function

BST - Finding the minimum element



How can we find the minimum element of the binary search tree?

BST - Finding the minimum element

Finding the minimum in a Binary Search Tree:

function minimum(tree) **is:**

//pre: tree is a Binary Search Tree

//post: minimum = the minimum value from the tree

BST - Finding the minimum element

Finding the minimum in a Binary Search Tree:

function minimum(tree) **is:**
//pre: tree is a Binary Search Tree
//post: minimum = the minimum value from the tree
  currentNode $\leftarrow$ tree.root
  if currentNode = NIL **then**
    @empty tree, no minimum
  else
    while [currentNode].left $\neq$ NIL **execute**
      currentNode $\leftarrow$ [currentNode].left
    end-while
    minimum $\leftarrow$ [currentNode].info
  end-if
end-function

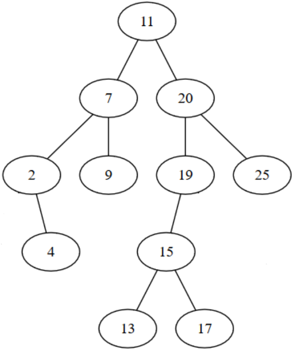
BST - Finding the minimum element

The time complexity of the *minimum* operation is

BST - Finding the minimum element

- The time complexity of the *minimum* operation is $O(h)$ (or $O(n)$)
- We can adapt the *minimum* function for finding the minimum of a subtree
  - The parameter would be a pointer to the root of the subtree
- Maximum can be found symmetrically
  - It is in the rightmost node of the BST

Finding the parent of a node



How can we find the parent of the node?

Finding the parent of a node

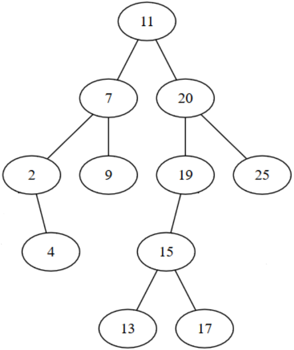
Finding the parent of a node in a Binary Search Tree:

function parent(tree, node) **is:**
//pre: tree is a Binary Search Tree, node is a pointer to a BSTNode, node $\neq$ NIL
//post: returns the parent of node, or NIL if node is the root

```
    c ← tree.root
    if c = node then //node is the root
        parent ← NIL
    else
        while c ≠ NIL and [c].left ≠ node and [c].right ≠ node execute
            if [c].info ≥ [node].info then
                c ← [c].left
            else
                c ← [c].right
            end-if
        end-while
        parent ← c
    end-if
end-function
```

Complexity: $O(h)$ (or $O(n)$)

BST - Finding the successor



How can we find the successor of a node?

How can we find the successor of 11? What about the successor of 17?

BST - Finding the successor of a node

How can we find the **successor** of a node *x* in a binary search tree?

The idea is the following:

- If *x* has a non-empty right subtree:
  - Its successor is just the leftmost node in *x*'s right subtree
- If *x* does not have a right subtree:

  The successor of *x* is the lowest ancestor of the node whose left child is also an ancestor of *x*

  - We go up the tree from *x* until we encounter a node that is the left child of its parent. The parent of that node is the succesor.

BST - Finding the successor of a node

Finding the successor of a node in a Binary Search Tree:

function successor(tree, x) **is:**
//pre: tree is a Binary Search Tree, x is a pointer to a BSTNode, x $\neq$ NIL
//post: returns the node with the next value after the value from x
//or NIL if x is the maximum
    if [x].right $\neq$ NIL **then**
        c $\leftarrow$ [x].right
        while [c].left $\neq$ NIL **execute**
            c $\leftarrow$ [c].left
        end-while
        successor $\leftarrow$ c
    else
        p $\leftarrow$ parent(tree, x)
        while p $\neq$ NIL **and** [p].left $\neq$ x **execute**
            x $\leftarrow$ p
            p $\leftarrow$ parent(tree, p)
        end-while
        successor $\leftarrow$ p
    end-if
end-function

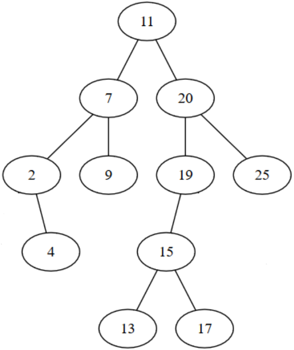
BST - Finding the successor of a node

Time complexity of *successor* depends on *parent* function:

- If *parent* runs in $\Theta(1) \Rightarrow$ complexity of successor is $O(h)$ (or $O(n)$)
- If *parent* runs in $O(n) \Rightarrow$ complexity of successor is $O(h^2)$ (or $O(n^2)$)

Similar to *successor*, we can define a *predecessor* function as well.

BST - Remove a node



How can we remove the value 25? What about removing 2? What about removing 11?

BST - Remove a node

When we want to remove a value (a node x containing the given value) from a BST we have 3 cases:

- x has no children
  - Set the corresponding child of the parent to NIL
- x has one descendant
  - Set the corresponding child of the parent to the child of x
- x has two children
  - Find its predecessor p, let it take the position of x in the tree and delete the node containing p

  OR

  - Find its successor s and splice s out of its current location and have it replace x in the tree.

BST - Removing an element

Recursively removing an element from a BST:

```
function remove_rec(p, e) is:
//pre: p: ↑ BSTNode, e: TComp
//post: e is removed from the BST rooted by p; return the root of the new (sub)tree
    if p = NIL then
        remove_rec ← p
    else
        if e < [p].info then
            [p].left ← remove_rec([p].left,e)
            remove_rec ← p
        else if e > [p].info then
            [p].right ← remove_rec([p].right,e)
            remove_rec ← p
        else
            if [p].left ≠ NIL and [p].right ≠ NIL then
                temp ← minimum([p].dr)
                [p].info ← [temp].info
                [p].right ← remove_rec([p].right, [p].info)
                remove_rec ← p
//continued on the next slide
```

BST - Removing an element

Recursively removing an element from a BST:

```
        else if [p].left = NIL then
            remove_rec← [p].right
        else
            remove_rec ← [p].left
        end-if
    end-if
  end-if
end-function
```

The time complexity of the *remove* operation is $O(h)$ (or $O(n)$)

BST - Removing an element

We need a wrapper function to call *remove_rec* with the root of the tree:

Recursively deletion an from a BST - - initial call:

function remove (tree, e) **is:**
//pre: tree is a Binary Search Tree, e is the elem to be removed
//post: tree' is a Binary Search Tree obtained by removing e from tree
    tree.root $\leftarrow$ remove_rec(tree.root, e)
end-function

Others

- Traversals from previous course

Containers represented using BSTs

BSTs are used for representing the following containers:

- ADT Set
  - set in C++ STL, TreeSet in Guava (Google Core Libraries for Java) (implemented using balanced BST)
- ADT Map (Sorted Map)
  - map in C++ STL, TreeMap in Guava (for Java)
- ADT MultiMap (Sorted MultiMap)
  - multimap in C++ STL, TreeMultimap in Guava (for Java)
- ADT Bag
  - TreeMultiset in Guava (for Java)

Binary Search Trees- Applications

Real-word applications of binary search tree data structure:

Unix Kernel

- Managing Virtual Memory Areas (VMAs)

Compilers

- For implementing Syntax Tress

Routing tables

- A routing table is used to link routers in a network. It is usually implemented with a variation of a binary search tree.

Data compression

- Huffman coding

Bibliography

- David M. Mount, *Lecture notes for the course Data Structures* (CMSC 420), at the Dept. of Computer Science, University of Maryland, College Park, 2001

- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein, *Introduction to Algorithms*, Third Edition, The MIT Press, 2009

- Narasimha Karumanchi, *Data Structures and Algorithms Made Easy: Data Structures and Algorithmic Puzzles*, Fifth Edition, 2016

- Clifford A. Shaffer, *A Practical Introduction to Data Structures and Algorithm Analysis*, Third Edition, 2010

Thank you