DATA STRUCTURES AND ALGORITHMS

LECTURE 12

Lect. PhD. Oneț-Marian Zsuzsanna

Babeș - Bolyai University
Computer Science and Mathematics Faculty

2024 - 2025

In Lecture 11...

- Tree
- Binary tree

Today

- Huffman encoding
- Binary search tree

Huffman coding

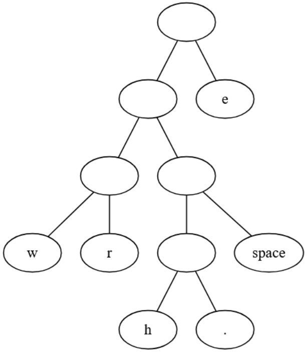
- When building the Huffman encoding, we need to have a message that we want to encode.
- First we have to compute the frequency of every character from the message, because we are going to define the codes based on the frequencies.
- Assume that we have the following message: WE WERE HERE.
- It contains the following characters and frequencies:

w	e	r	h	space	.
2	5	2	1	2	1

Huffman coding

- For defining the Huffman code a binary tree is build in the following way:
    - Start with trees containing only a root node, one for every character. Each tree has a weight, which is the frequency of the character.
    - Get the two trees with the least weight (if there is a tie, choose randomly), combine them into one tree which has as weight the sum of the two weights.
    - Repeat until we have only one tree.
- The implementation can simply be done with a Priority Queue which stores these partially built trees (and considers the weight as priority).

Huffman coding



Huffman coding

- Code for each character can be read from the tree in the following way: start from the root and go towards the corresponding leaf node. Every time we go left add the bit 0 to encoding and when we go right add bit 1.
- Code for the characters:
    - w - 000
    - r - 001
    - h - 0100
    - . - 0101
    - space - 011
    - e - 1
- We can see that the most frequent character (e), indeed has the shortest code, and the least frequent ones have the longest. Also, no code is the prefix of another.
- In order to encode a message, just replace each character with the corresponding code.

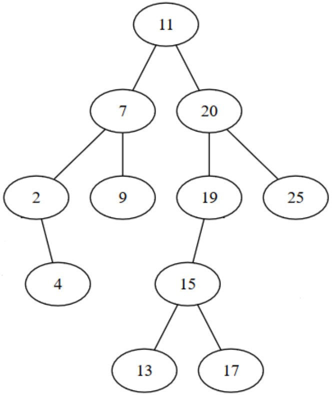
Huffman coding

- Assume we have the following code and we want to decode it: 0110110001000100111001000000
- We do not know where the code of each character ends, but we can use the previously built tree to decode it.
- Start parsing the code and iterate through the tree in the following way:
  - Start from the root
  - If the current bit from the code is 0 go to the left child, otherwise go to the right child
  - If we are at a leaf node we have decoded a character and have to start over from the root
- The decoded message (quotation marks added to see the spaces at the beginning): " wewereerww"

Binary search trees

- A *Binary Search Tree* is a binary tree that satisfies the following property:
  - if x is a node of the binary search tree then:
    - 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 the information from x
- Obviously, the relation used to order the nodes can be considered in an abstract way (instead of having "$\leq$" as in the definition).

Binary Search Tree Example



- If we do an inorder traversal of a binary search tree, we will get the elements in increasing order (according to the relation used).

Binary Search Tree

- The terminology and many properties discussed for binary tree is valid for binary search trees as well:
  - We can have a binary search tree that is full, complete, almost complete, degenerate or balanced
  - 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 - if the tree is degenerate.
  - A binary search tree with N nodes has a height between $log_2 N$ and N (we will denote the height of the tree by h).

Binary Search Tree

- Binary search trees can be used as representation for sorted containers: sorted maps, sorted multimaps, priority queues, sorted sets, etc.
- In order to implement these containers on a binary search tree, we need to define the following basic operations:
  - search for an element
  - insert an element
  - remove an element
- Other operations that can be implemented for binary search trees (and can be used by the containers): get minimum/maximum element, find the successor/predecessor of an element.

Binary Search Tree - Representation

- We will use a linked representation with dynamic allocation (similar to what we used for binary trees)

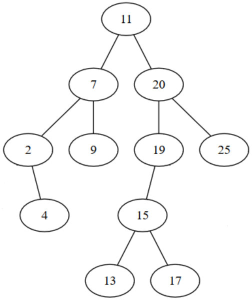
```
BSTNode:
    info: TElem
    left: ↑ BSTNode
    right: ↑ BSTNode
```

```
BinarySearchTree:
    root: ↑ BSTNode
```

- Normally, BST would contain a relation as well. In our examples we will use the $\leq$ relation directly.

Binary Search Tree - search operation

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



- How can we search for element 15? And for element 14?

BST - search operation - recursive implementation

- How can we implement the *search algorithm* recursively?

BST - search operation - recursive implementation

- How can we implement the *search algorithm* recursively?

```
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

- Since the *search* algorithm takes as parameter a node, we need a wrapper function to call it with the root of the tree.

```
function search (tree, e) is:
//pre: tree is a BinarySearchTree, e is the elem we are looking for
   search ← search_rec(tree.root, e)
end-function
```

BST - search operation - non-recursive implementation

- How can we define the search operation non-recursively?

BST - search operation - non-recursive implementation

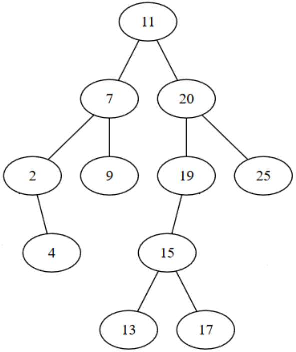
- How can we define the search operation non-recursively?

```
**function** search (tree, elem) **is:**
*//pre: tree is a BinarySearchTree and elem is the TElem we are searching for*
    currentNode $\leftarrow$ tree.root
    found $\leftarrow$ false
    **while** currentNode $\neq$ NIL and not found **execute**
        **if** [currentNode].info $=$ elem **then**
            found $\leftarrow$ true
        **else if** [currentNode].info $<$ elem **then**
            currentNode $\leftarrow$ [currentNode].right
        **else**
            currentNode $\leftarrow$ [currentNode].left
        **end-if**
    **end-while**
    search $\leftarrow$ found
**end-function**
```

BST operation complexities

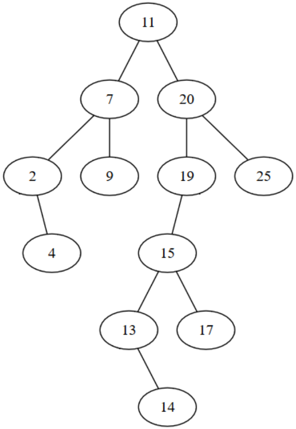
- Most BST operations will follow one single path from the root to a leaf node (or maybe stop somewhere at an internal node, depending on the problem), which means that their complexity depends on the height of the tree, h.
- As discussed, height of the tree can be between $log_2 n$ and n, so the worst case complexities will in general be $\Theta(n)$ for the operations, which means that total complexity will be in many cases O(n).
- Nevertheless, average complexity is $\Theta(log_2 n)$ (assuming that the elements were inserted in a random order).
- Regarding the search algorithm, best case complexity is $\Theta(1)$, average case is $\Theta(log_2 n)$ and worst case is $\Theta(n)$.

BST - insert operation



- How/Where can we insert element 14?

BST - insert operation



BST - insert operation - recursive implementation

- How can we implement the *insert* operation?

BST - insert operation - recursive implementation

- How can we implement the *insert* operation?
- We will start with a function that creates a new node given the information to be stored in it.

```
function initNode(e) is:
//pre: e is a TComp
//post: initNode ← a node with e as information
    allocate(node)
    [node].info ← e
    [node].left ← NIL
    [node].right ← NIL
    initNode ← node
end-function
```

BST - insert operation - recursive implementation

```
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 ← initNode(e)
    else if [node].info ≥ e then
        [node].left ← insert_rec([node].left, e)
    else
        [node].right ← insert_rec([node].right, e)
    end-if
    insert_rec ← node
end-function
```

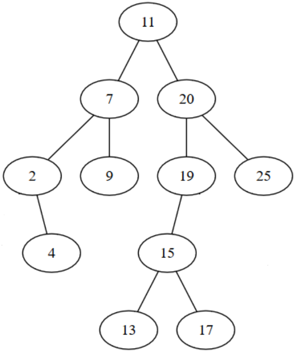
- Complexity:

BST - insert operation - recursive implementation

```
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 ← initNode(e)
    else if [node].info $\geq$ e then
        [node].left ← insert_rec([node].left, e)
    else
        [node].right ← insert_rec([node].right, e)
    end-if
    insert_rec ← node
end-function
```

- Complexity: O(n) ($\Theta(n)$ in worst case, but $\Theta(log_2 n)$ on average)
- Like in case of the *search* operation, we need a wrapper function to call *insert_rec* with the root of the tree.

BST - Finding the minimum element



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

BST - Finding the minimum element

```
function minimum(tree) is:
//pre: tree is a BinarySearchTree
//post: minimum ← the minimum value from the tree
   currentNode ← tree.root
   if currentNode = NIL then
      @empty tree, no minimum
   else
      while [currentNode].left ≠ NIL execute
         currentNode ← [currentNode].left
      end-while
      minimum ← [currentNode].info
   end-if
end-function
```

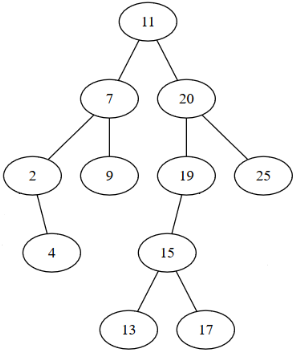
BST - Finding the minimum element

- Complexity of the minimum operation:

BST - Finding the minimum element

- Complexity of the minimum operation: $O(n)$
- We can have an implementation for the minimum, when we want to find the minimum element of a subtree, in this case the parameter to the function would be a node, not a tree.
- We can have an implementation where we return the node containing the minimum element, instead of just the value (depends on what we want to do with the operation)
- Maximum element of the tree can be found similarly.

Finding the parent of a node



- Given a node, how can we find the parent of the node? (assume a representation where the node has no parent field).

Finding the parent of a node

```
function parent(tree, node) is:
//pre: tree is a BinarySearchTree, node is a pointer to a BSTNode, node $\neq$ NIL
//post: returns the parent of node, or NIL if node is the root
    c $\leftarrow$ tree.root
    if c = node then //node is the root
        parent $\leftarrow$ NIL
    else
        while c $\neq$ NIL and [c].left $\neq$ node and [c].right $\neq$ node execute
            if [c].info $\geq$ [node].info then
                c $\leftarrow$ [c].left
            else
                c $\leftarrow$ [c].right
            end-if
        end-while
        parent $\leftarrow$ c
    end-if
end-function
```

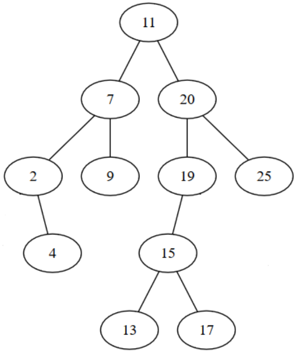
- Complexity:

Finding the parent of a node

```
function parent(tree, node) is:
//pre: tree is a BinarySearchTree, node is a pointer to a BSTNode, node $\neq$ NIL
//post: returns the parent of node, or NIL if node is the root
    c $\leftarrow$ tree.root
    if c = node then //node is the root
        parent $\leftarrow$ NIL
    else
        while c $\neq$ NIL and [c].left $\neq$ node and [c].right $\neq$ node execute
            if [c].info $\geq$ [node].info then
                c $\leftarrow$ [c].left
            else
                c $\leftarrow$ [c].right
            end-if
        end-while
        parent $\leftarrow$ c
    end-if
end-function
```

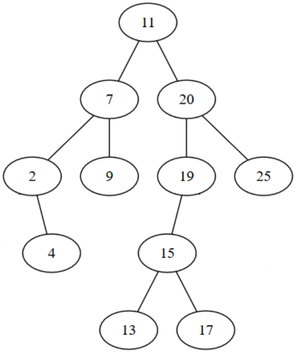
- Complexity: $O(n)$

BST - Finding the successor of a node



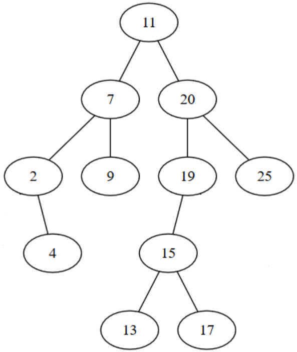
- Given a node, how can we find the node containing the next value (considering the relation used for ordering the elements)?
- How can we find the next after 11?

BST - Finding the successor of a node



- Given a node, how can we find the node containing the next value (considering the relation used for ordering the elements)?
- How can we find the next after 11? After 13?

BST - Finding the successor of a node



- Given a node, how can we find the node containing the next value (considering the relation used for ordering the elements)?
- How can we find the next after 11? After 13? After 17?

BST - Finding the successor of a node

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

BST - Finding the successor of a node

- Complexity of successor:

BST - Finding the successor of a node

- Complexity of successor: depends on parent function:
    - If *parent* is $\Theta(1)$, complexity of successor is $O(n)$
    - If *parent* is $O(n)$, complexity of successor is $O(n^2)$
- What if, instead of receiving a node, successor algorithm receives as parameter a value from a node (assume unique values in the nodes)? How can we find the successor then?
- Similar to successor, we can define a predecessor function as well.

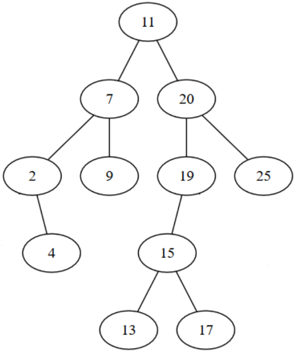
BST - Finding successor of a node

- If we do not have direct access to the parent, finding the successor of a node is $O(n^2)$.
- Can we reduce this complexity?

BST - Finding successor of a node

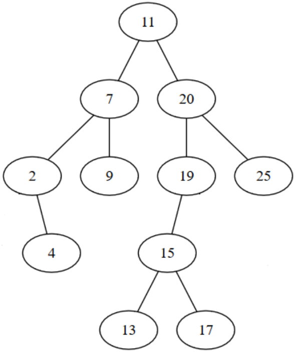
- If we do not have direct access to the parent, finding the successor of a node is $O(n^2)$.
- Can we reduce this complexity?
- $O(n^2)$ is given by the potentially repeated calls for the *parent* function. But we only need the *last* parent, where we went left. We can do one single traversal from the root to our node, and every time we continue left (i.e. current node is greater than the one we are looking for) we memorize that node in a variable (and change the variable when we find a new such node). When the current node is at the one we are looking for, this variable contains its successor.

BST - Remove a node



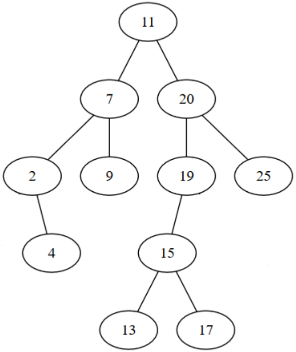
- How can we remove the value 25?

BST - Remove a node



- How can we remove the value 25? And value 2?

BST - Remove a node



- How can we remove the value 25? And value 2? And value 11?

BST - Remove a node

- When we want to remove a value (a node containing the value) from a binary search tree we have three cases:
  - The node to be removed has no descendant
    - Set the corresponding child of the parent to NIL
  - The node to be removed has one descendant
    - Set the corresponding child of the parent to the descendant
  - The node to be removed has two descendants
    - Find the maximum of the left subtree, move it to the node to be deleted, and delete the maximum
      OR
    - Find the minimum of the right subtree, move it to the node to be deleted, and delete the minimum

Binary Search Tree

- Think about it:
    - Can we define a Sorted List on a Binary Search Tree? If not, why not? If yes, how exactly? What would be the most complicated part?

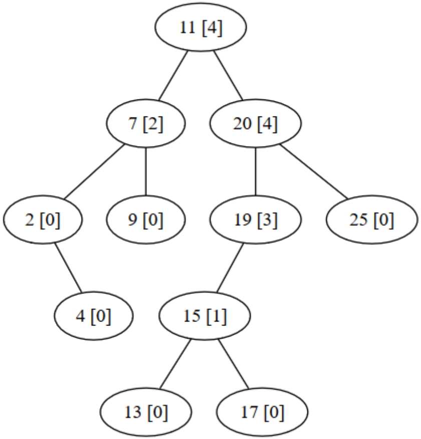
Binary Search Tree

- Think about it:
  - Can we define a Sorted List on a Binary Search Tree? If not, why not? If yes, how exactly? What would be the most complicated part?
  - Lists have positions and operations based on positions. In case of a SortedList we have an operation to remove an element from a position and to return an element from a position (but no insert to position). How can we keep track of positions in a binary search tree?

Binary Search Tree

- Think about it:
    - Can we define a Sorted List on a Binary Search Tree? If not, why not? If yes, how exactly? What would be the most complicated part?
    - Lists have positions and operations based on positions. In case of a SortedList we have an operation to remove an element from a position and to return an element from a position (but no insert to position). How can we keep track of positions in a binary search tree?
    - We can keep in every node, besides the information, the number of nodes in the left subtree. This gives us automatically the "position" of the root in the SortedList. When we have operations that are based on positions, we use these values to decide if we go left or right.

Binary Search Tree



Binary Search Tree with duplicate values

- Starting from an initially empty Binary Search Tree and the relation $\leq$, insert into it, in the given order, the following values: 10, 20, 5, 7, 15, 5, 30, 3, 5, 5, 1, 9, 29, 2.

Binary Search Tree with duplicate values

- Starting from an initially empty Binary Search Tree and the relation $\leq$, insert into it, in the given order, the following values: 10, 20, 5, 7, 15, 5, 30, 3, 5, 5, 1, 9, 29, 2.
- How would you count how many times the value 5 is in the tree?

Binary Search Tree with duplicate values

- Starting from an initially empty Binary Search Tree and the relation $\leq$, insert into it, in the given order, the following values: 10, 20, 5, 7, 15, 5, 30, 3, 5, 5, 1, 9, 29, 2.
- How would you count how many times the value 5 is in the tree?
- Remove 3 (show both options)

Binary Search Tree with duplicate values

- Starting from an initially empty Binary Search Tree and the relation $\leq$, insert into it, in the given order, the following values: 10, 20, 5, 7, 15, 5, 30, 3, 5, 5, 1, 9, 29, 2.
- How would you count how many times the value 5 is in the tree?
- Remove 3 (show both options)
- How would you count now how many times the value 5 is in the tree now?

Today

- Huffman encoding
- Binary search tree