DVA104

Data Structures, Algorithms, and Program Development

Gabriele Capannini

gabriele.capannini@mdh.se

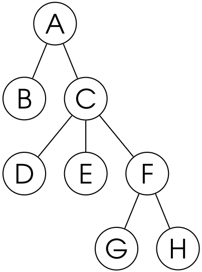
Mälardalen University

October 1, 2018

Trees

Trees are a very common data structure used for:

- representing the file folders structure
- evaluating arithmetical expressions
- compiling a program
- performing effective search
- ...

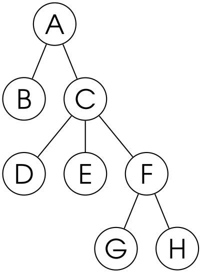


Trees: Terminology

Trees are special type of graphs so that they inherit many terms. Some new ones are listed here:

- Root is the top node and is unique.
- Child is a node directly connected to another one moving downward.
- Parent is the reverse relation of child.
- Siblings are nodes sharing the same parent.
- Leaves is a node having no children.
- An empty tree is a tree without nodes.
- Node depth is the number of edges in its path from the root.
- Level of a node is its depth $+1$.
- A tree level is the set of nodes at the same level.
- Degree of a node is its number of children.

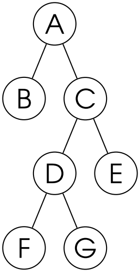
Trees: Terminology (example)



- A is the root.
- B, D, E, G, H are leaves.
- D is child of C hence C is parent of D.
- B and C are siblings.
- Depth of D is 2 since its path from A consists of 2 edges: A → C → D.
- The level 3 of the tree consists of D, E, F.
- Degree of C is 3.

Trees are recursive!

Example:



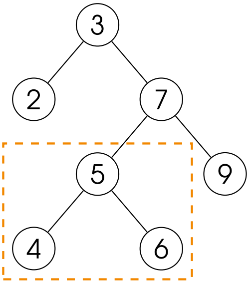
- A is the root of the tree of which nodes are: A, B, C, D, E, F, and G.
- C is the root of the subtree of which nodes are: C, D, E, F, and G.
- D is the root of the subtree of which nodes are: D, F, and G.
- F is the root of the subtree with only one node: F.

A subtree is a tree as well! ✌

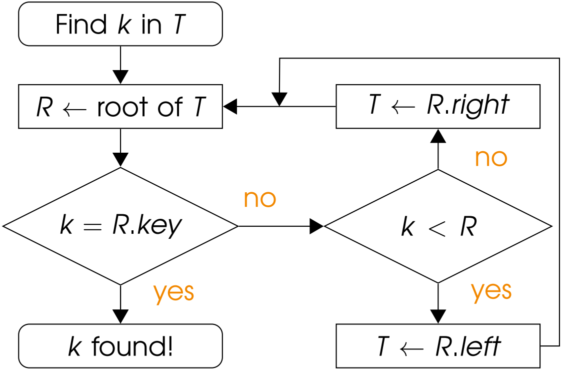
Binary Search Tree (BST)

- A binary tree is a tree of which nodes have a degree ≤ 2.
- So at most two subtrees can be generated by each node of a binary tree (a.k.a. left- and right-subtree).
- A Binary Search Tree (BST) is a Binary Tree (BT) of which nodes are associated to a value (a.k.a. key). Moreover, in a BST, nodes are kept sorted: let k be a node, all keys stored in its left-subtree are less than k, while keys that are equal or greater than k are stored in the right-subtree.

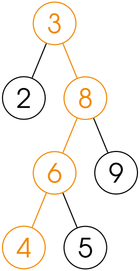
Example: In the BST on the right, all values grater than 3 and less than 7 are stored in the right-subtree of 3 and the left-subtree of 7.



BST: how to search for a value?



Example: find 4

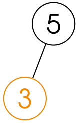


Building a BST

In the beginning we have an empty tree:

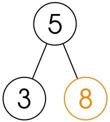
. . . Then we insert 5 which has no children:

. . . Then we insert 3 which is less than 5:

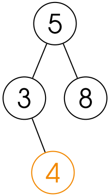


Building a BST (continued)

. . . Then we insert 8 which is greater than 5:

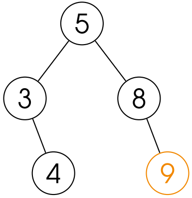


. . . Then we insert 4 which is less than 5 and greater than 3:

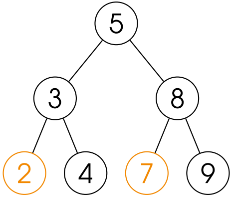


Building a BST (continued)

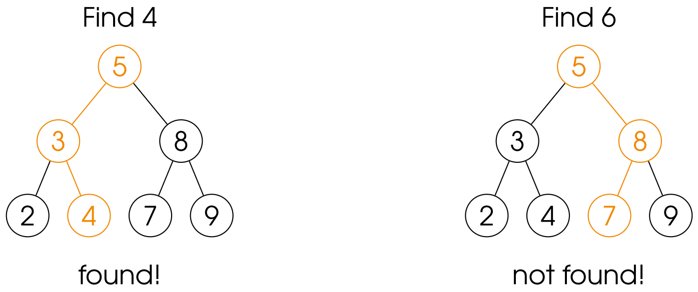
. . . Then we insert 9 which is greater than 5 and 8:



. . . Finally we insert 2 and 7:



Searching in a BST is fast!



- In both cases, we found the answer by visiting 3 nodes while the tree contains 7 nodes.
- We also needed to visit a lot fewer nodes for entering a new item.

Searching in a BST (final version)

Does the tree T contains the value k?

```
function TreeHasValue(T, k):
    if T == ∅ then
        return NO
    end if
    if T.key == k then
        return YES
    end if
    if k < T.key then
        return TreeHasValue(T.left, k)
    else
        return TreeHasValue(T.right, k)
    end if
end function
```

Insertion into a BST

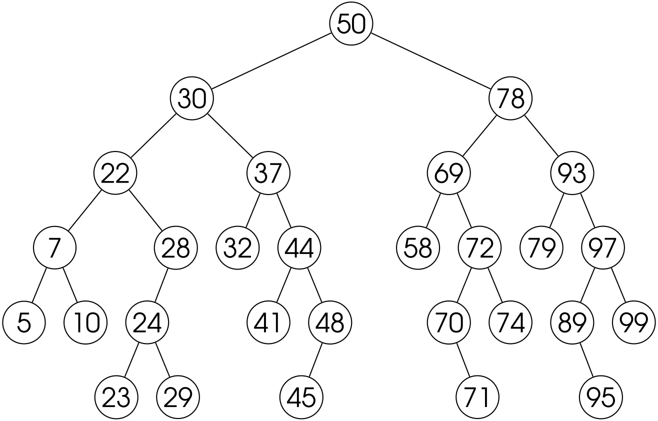
Insert the value k in the tree T.

```
function TREECREATENODE(key k):
    return node such that key=k, left=∅, and right=∅
end function
```

```
function TREEINSERTVALUE(T, k):
    if T == ∅ then
        T ← TREECREATENODE(k)
    end if
    if T.key == k then
        Raise exception
    end if
    if k < T.key then
        TREEINSERTVALUE(T.left, k)
    else
        TREEINSERTVALUE(T.right, k)
    end if
end function
```

BST: test

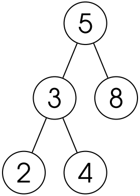
Is the following tree a BST?

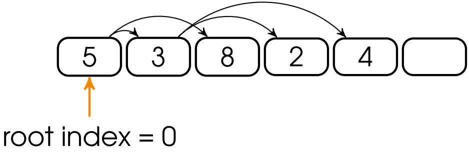


BSTs as Arrays

BSTs (as well as BTs) can be easily implemented by means of an array:

- root is registered in the first position, i.e., 0
- left child of the node stored in position i is in position $2i + 1$
- right child of the node stored in position i is in position $2i + 2$





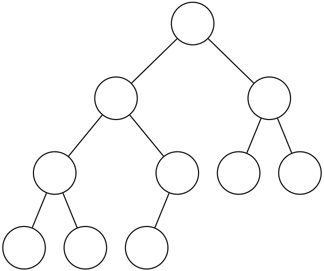
Cons:

- When array is full you need to resize it which can be costly.
- Array has ‘holes’ if the tree is not complete.

About Complete BTs

Definition: a binary tree is complete if all levels are completely filled except possibly the last level that has all keys as left as possible.

Example:



BSTs as Linked Lists

BTs can be also implemented by means of a LL.

Example of a tree node with integer keys:

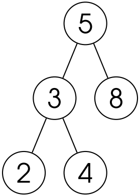
```
struct btreenode {
  int key;
  treeNode* left;
  treeNode* right;
};
typedef struct btreenode BTNode;
```

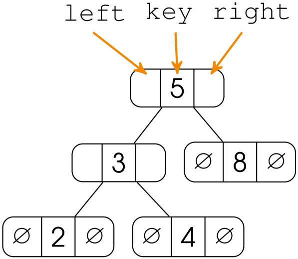
When a new node is created `left` and `right` are `NULL` which means that, initially, a new node has no children:

```
BTNode* createNewBTNode(const int key) {
  BTNode* newnode = (BTNode*)malloc(sizeof(BTNode));
  newnode->left = newnode->right = NULL;
  newnode->key = key;
  return newnode;
}
```

BSTs as Linked Lists (continued)

Example:





∅ = `NULL`

- Each pointer points to a whole `BTNode` not only to `key`.
- Memory management is simplified but, now, each node needs more memory to be stored.

Tree traversal

Just like a LL or an array, we may want to visit every node in a tree.

However, depending on our purpose, it is not obvious in which order you want to visit the nodes in a tree.

For example:

- Should the nodes on top be visited before the bottom?
- Should we visit left-hand tree trees before right-hand tree trees?

Tree traversal (continued)

There are many (recursive) alternatives:

Pre-order (LR):

1. visit the current node,
2. visit the left-subtree,
3. visit the right-subtree.

In-order (LR):

1. visit the left-subtree,
2. visit the current node,
3. visit the right-subtree.

Post-order (LR):

1. visit the left-subtree,
2. visit the right-subtree,
3. visit the current node.

Pre-order (RL):

1. visit the current node,
2. visit the right-subtree,
3. visit the left-subtree.

In-order (RL):

1. visit the right-subtree,
2. visit the current node,
3. visit the left-subtree.

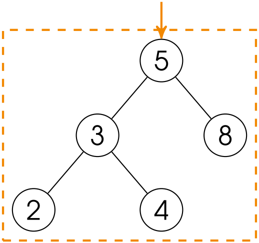
Post-order (RL):

1. visit the right-subtree,
2. visit the left-subtree,
3. visit the current node.

Tree traversal (example)

We want to print the following tree in Pre-order (LR) manner:

```
procedure VISIT(T)
    PRINT(T.key)
    VISIT(T.left)
    VISIT(T.right)
end procedure
```



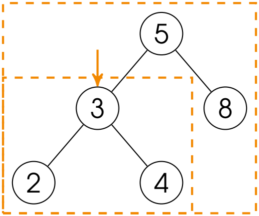
*dashed orange boxes enclose trees on which a visit procedure is active.

Printed nodes: 5

Tree traversal (example)

We want to print the following tree in Pre-order (LR) manner:

```
procedure VISIT(T)
    PRINT(T.key)
    VISIT(T.left)
    VISIT(T.right)
end procedure
```



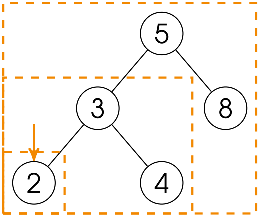
*dashed orange boxes enclose trees on which a visit procedure is active.

Printed nodes: 5, 3

Tree traversal (example)

We want to print the following tree in Pre-order (LR) manner:

```
procedure VISIT(T)
    PRINT(T.key)
    VISIT(T.left)
    VISIT(T.right)
end procedure
```



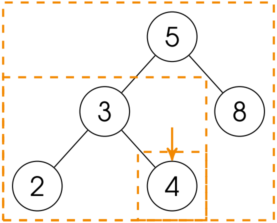
*dashed orange boxes enclose trees on which a visit procedure is active.

Printed nodes: 5, 3, 2

Tree traversal (example)

We want to print the following tree in Pre-order (LR) manner:

```
procedure VISIT(T)
    PRINT(T.key)
    VISIT(T.left)
    VISIT(T.right)
end procedure
```



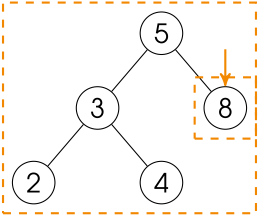
*dashed orange boxes enclose trees on which a visit procedure is active.

Printed nodes: 5, 3, 2, 4

Tree traversal (example)

We want to print the following tree in Pre-order (LR) manner:

```
procedure Visit(T)
    Print(T.key)
    Visit(T.left)
    Visit(T.right)
end procedure
```



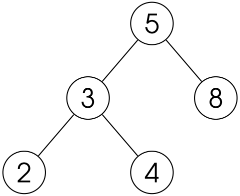
*dashed orange boxes enclose trees on which a visit procedure is active.

Printed nodes: 5, 3, 2, 4, 8

Tree traversal (example)

We want to print the following tree in Pre-order (LR) manner:

```
procedure VISIT(T)
    PRINT(T.key)
    VISIT(T.left)
    VISIT(T.right)
end procedure
```

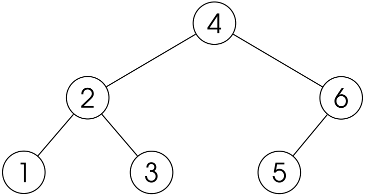


*dashed orange boxes enclose trees on which a visit procedure is active.

Printed nodes: 5, 3, 2, 4, 8

Tree traversal (test)

What's the result in this case? Which kind of traversal are we performing?




```
void printBST(BTNode* tree) {
  if(!root)
    return;
  printBST(tree->left);
  printf("%d\n", tree->key);
  printBST(tree->right);
}
```

BST: removing a tree

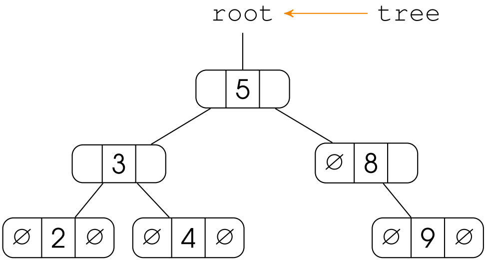
Adding a node into a BST is `easy'. . .

While removing can lead to different cases:

- remove a leaf,
- remove a node with a child,
- remove a node with two children.

Removing a leaf

```
typedef struct btreenode BTNode;
typedef BTNode* BTree;
void remove(BTree* tree /*double pointer*/, int key) {
   ...
}
BTree root;
...
remove(&root, 9); //initially tree points to root
```

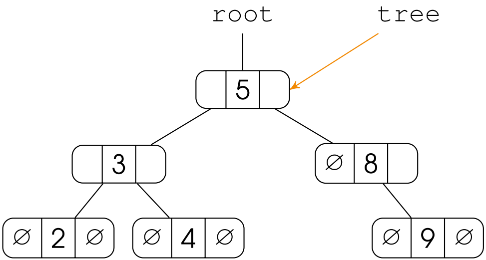


first of all we traverse the tree until tree point to the pointer to 9 ...

Removing a leaf (continued)

We recursively set `tree` with the address of the pointers belonging to the path from (5) to (9). For example, the first recursive call will look like:

```
BTNode* node = *tree;
...
if(key<node->key) // i.e. 9<5 is FALSE, go right
  ...
else
  remove(&node->right, 9); // first recursive call
```

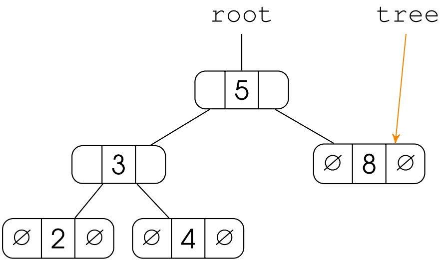


in the second step, `tree` points to the right-subtree of (5)

Removing a leaf (continued)

We reached 9 and `tree` points to the right-subtree of 8 :

```
BTNode* node = *tree;
if(key==node->key) // i.e. 9==9 is TRUE
{
   free(*tree); //free memory
   *tree = NULL; //update pointer
}
```

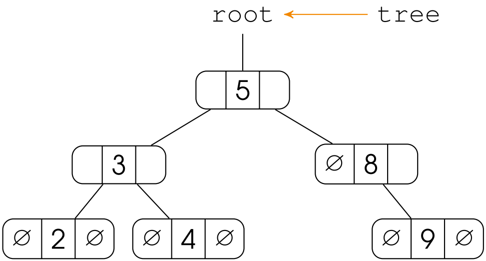


after removing 9 we get a tree like this

Removing nodes with one child

This time we want to remove a node like 8 :

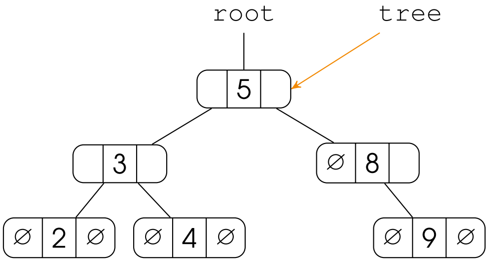
```
BTree root;
...
remove(&root, 8); //initially tree points to root
```



first of all we traverse the tree until we reach 8 . . .

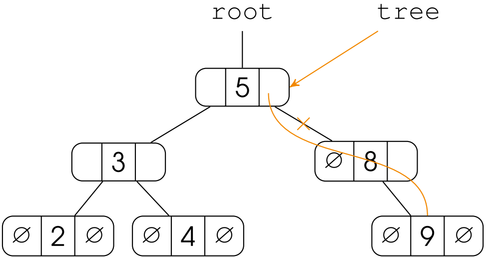
Removing nodes with one child (continued)

As in the previous case, we need to reach the node of which child points to the target value: in this case it is 8. In this case, however, 8 has a child:



Removing nodes with one child (continued)

We replace the child-pointer pointed by `tree` with the child of the node to remove:



That is:

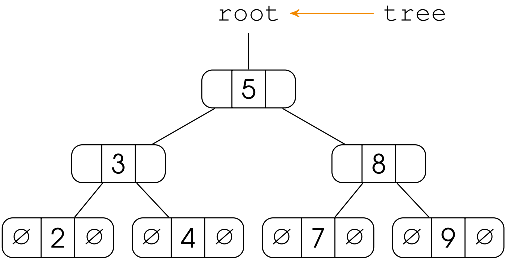
```
BTNode* tmp = *tree;
*tree = *tree->right;
free(tmp);
```

Notice that 'removing a leaf' is just a 'simplified' case of this since in that case `*tree->right` is `NULL`.

Removing nodes with two children

This time we want to remove a node like 5 :

```
BTree root;
...
remove(&root, 5); //initially tree points to root
```



In this case we need to find a good candidate to replace 5 : who?

Hint: remember the definition of BST!

Removing nodes with two children (continued)

Removing (5) :

we can alternatively choose:

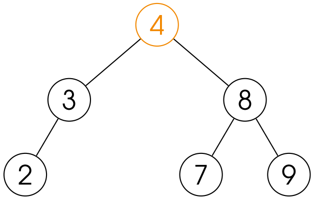
- the max value stored in the left-subtree, that is 4
- the min value stored in the right-subtree, that is 7

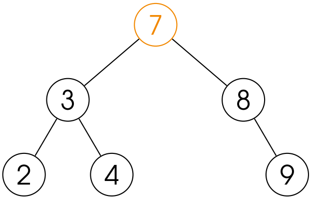
By the way, how to find the min and max value in a BST?

Removing nodes with two children (continued)

Removing 5 :

Once the node has been replaced, we proceed by removing the copied one:





Note that removing such a node must fall in one of the two previous cases: `removing a leaf' or `removing a node with one child'.

Tree balancing

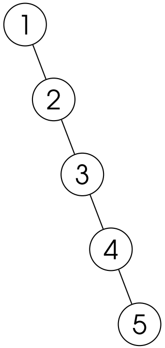
Questions:

- How does a BST look like if we insert the sequence 1,2,3,4,5?
- How many nodes are there in the tree?
- Is it still quick to search in this tree?
- Is it faster than a linked list?
- How many levels have the tree?

Tree balancing (continued)

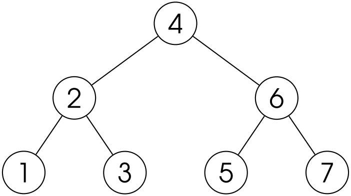
Answers:

- It looks like the next tree.
- There are 5 nodes.
- Searching is not faster than a LL.
- The tree has 5 levels.
- Such a tree is called unbalanced and it is not effective in searching.
- If the number of nodes equals the number of levels then the tree is a LL! (degenerate tree)



So, what is the min number of levels for a tree with 5 nodes?

Tree balancing (continued)



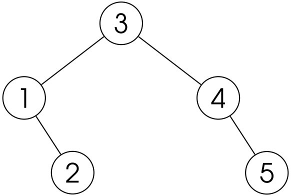
- The capacity of a BST with 1 level is 1 (just the root)
- The capacity of a BST with 2 levels is 3 (root+2ch, i.e., 1+2)
- The capacity of a BST with 3 levels is 7 (i.e., 3+4)
- The capacity of a BST with 4 levels is 15 (i.e., 7+8)

In general, the capacity of a BST with L levels is $2^L - 1$

Tree balancing (continued)

Hence the minimum number of levels for a BST with 5 nodes is 3!

For example it could be:



- Is that BST sorted? Yes
- How many nodes? 5
- How many levels? 3
- Is it effective for searching? Yes, max 3 steps

Tree balancing (continued)

If we can not store more than $2^L - 1$ nodes in a BST with L levels, how many levels do we need to store n nodes?

$$n \leq 2^L - 1$$
$$n + 1 \leq 2^L$$
$$\log_2(n+1) \leq \log_2(2^L) = L$$
$$L \geq \log_2(n+1)$$

Answer: we need at least $\log_2(n+1)$.

Since $\log_2(n+1)$ may contain decimals and L should be an integer, we must round up the result: $\lceil \log_2(n+1) \rceil$.

In C, round-up can be calculated using the `ceil` function:
`http://www.cplusplus.com/reference/cmath/ceil/`

Tree balancing (continued)

- A BST with n nodes on $\lceil \log_2(n+1) \rceil$ levels is said to be balanced (definition comes later).
- In the worst case, we need to visit $\lceil \log_2(n+1) \rceil$ nodes to find a given value, while in a LL this requires to visit n nodes:

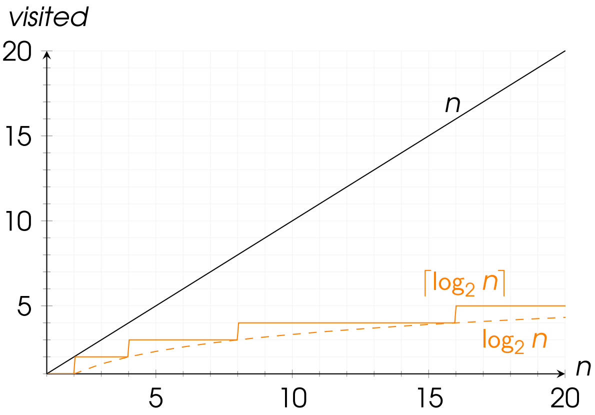
For example, assume that we have stored about one million values, e.g., $n = 1048576 = 2^{20}$.

LL in the worst case, we need to test one million (i.e., 1048576) nodes to find a specific one.

BST we need to test $\lceil \log_2(n+1) \rceil$ that is 20 nodes (at most).

Tree balancing (continued)

The following plot shows the number of *visited* nodes w.r.t. the number n of nodes stored in the data structure both for the LL and the balanced BST cases:



Tree balancing (continued)

Application:

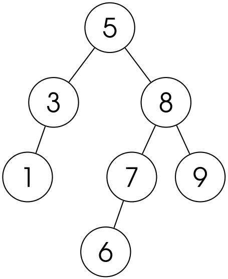
Do you remember ADTs Set?

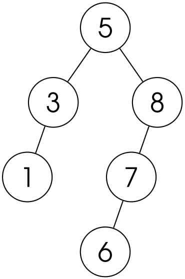
```
addElement()
removeElement()
isInSet()
```

If we implemented this as a sorted balanced BST instead of an Array or LL then the `isInSet()` function would be much faster!

Tree balancing (continued)

Definition: balanced tree means that the difference between the height of left- and right-subtree of each node is max 1, where the height of a (sub)tree is defined as the number of edges on the longest path between the root and a leaf.





According to the definition only the left one is balanced. Why?

Tree balancing (continued)

Now, the question is: how do we get a tree balanced?

There are several variants of self-balanced BSTs:

- AVL trees
- Red/Black Trees

Otherwise we can:

- First, enter all values from the tree into a sorted array (in-order LR visit).
- Then, balance the tree (rebuild the tree with the same values).

Building a balanced BST

Assume we have a sorted array with the values to store in the tree, how can we build it balanced?

For example, we want to store the values from 1 to 10 in a BST.

1	2	3	4	5	6	···	10

- How can we build it balanced?
- Which value should be the root?

Building a balanced BST (continued)

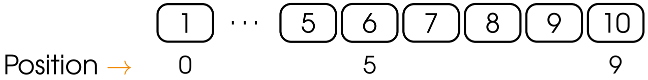
	1	2	3	4	5	6	$\cdots$	10
Position $\rightarrow$	0				4			9

The array has 10 entries (from position 0 to 9), '5' (in position 4) is almost in the middle: $4 = \lfloor (0 + 9)/2 \rfloor$.

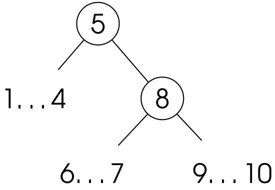
$$
\begin{array}{ccc}
 & 5 & \\
 \swarrow & & \searrow \\
1 \ldots 4 & & 6 \ldots 10
\end{array}
$$

Now, what should be the root of the right-subtree of 5 ?

Building a balanced BST (continued)

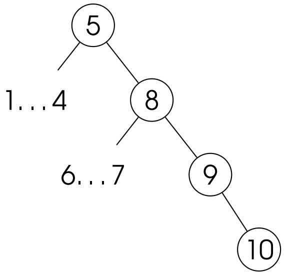


The right-subtree has 5 entries (from position 5 to 9), '8' (in position 7) is in the middle: $7 = \lfloor (5+9)/2 \rfloor$.



Building a balanced BST (continued)

We follow the same schema for adding '9' and '10' to the tree:



...Even for the left-subtree. For example, what should be the root of the left-subtree of 8? The left-subtree covers the array positions from 5 to 6, hence we select the value in position 5 (since $\lfloor (5+6)/2 \rfloor = 5$).

Building a balanced BST (continued)

We are building a balanced BST recursively. At each step:

- First, we choose the root from the given (sub)array.
- Then, we recursively apply the same procedure on the subarrays on the left and on the right of the chosen element for the left- and the right-subtree, respectively.

function BUILDBST(*node*, *array*, *first*, *last*):
  . . .
  $m \leftarrow \lfloor (first + last)/2 \rfloor$
  $node.key \leftarrow array[m]$
  BUILDBST(*node.left*, *array*, *first*, $m-1$)
  BUILDBST(*node.right*, *array*, $m+1$, *last*)
end function

What is the 'base-case' for this recursive function?

Building a balanced BST (continued)

When `10' is added, *first* and *last* equal 9 so that even *m* is equal to 9. Hence , the recursive calls look like:

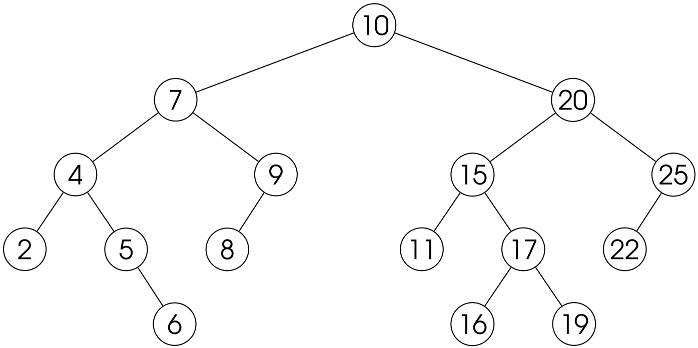
function BuildBST(*node*, *array*, *first*=9, *last*=9):
  ...
  $m \leftarrow \lfloor (first + last)/2 \rfloor = \lfloor (9+9)/2 \rfloor = 9$
  $node.key \leftarrow array[m]$
  BuildBST(*node.left*, *array*, *first*, $m-1$)
  BuildBST(*node.right*, *array*, $m+1$, *last*)
end function

The function is called twice on empty portions of *array* since $first > m-1$ and $m+1 > last$, which makes no sense. Hence the function should preliminary check the values of *first* and *last*:

function BuildBST(*node*, *array*, *first*, *last*):
  If $first > last$ **then** exit
  ...
end function

Test

Look at the following tree:



- Is it a BT?
- Is it a BST?
- Is it balanced?

The end. . . Questions?