Data Structures & Algorithm Analysis

Rada Mihalcea

http://www.cs.unt.edu/~rada/CSCE3110

Binary Search Trees

A Taxonomy of Trees

- General Trees – any number of children / node

- Binary Trees – max 2 children / node

  - Heaps – parent < (>) children
  - Binary Search Trees

Binary Trees

- Binary search tree
  - Every element has a unique key.
  - The keys in a nonempty left subtree (right subtree) are smaller (larger) than the key in the root of subtree.
  - The left and right subtrees are also binary search trees.

Binary Search Trees

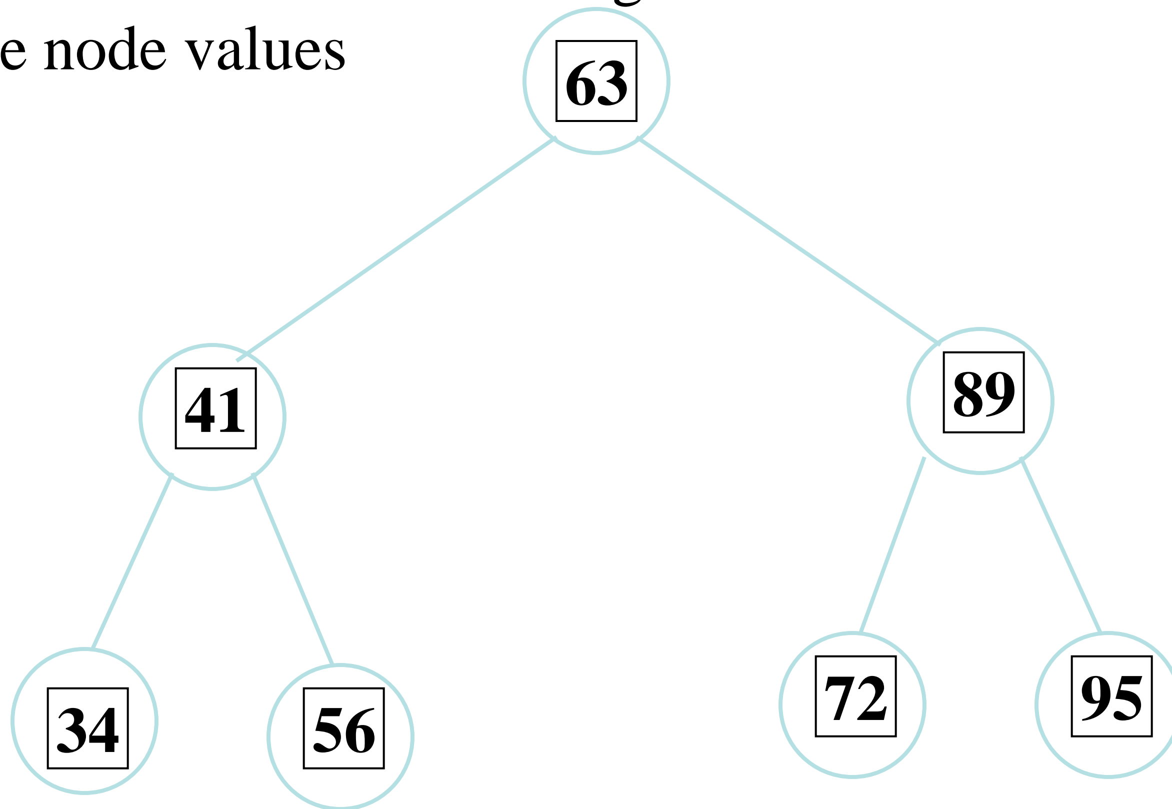
- Binary Search Trees (BST) are a type of Binary Trees with a special organization of data.
- This data organization leads to O(log n) complexity for searches, insertions and deletions in certain types of the BST (balanced trees).
  - O(h) in general

Binary Search Algorithm

Binary Search algorithm of an array of *sorted* items reduces the search space by one half after each comparison

Organization Rule for BST

- the values in all nodes in the left subtree of a node are less than the node value
- the values in all nodes in the right subtree of a node are greater than the node values



Binary Tree

```
typedef struct tnode *ptnode;
    typedef struct node {
                short int  key;
                ptnode right, left;
    } ;
```

BST Operations: Search

Searching in the BST

`function search(key)`

- implements the binary search based on comparison of the items in the tree

- the items in the BST must be comparable (e.g integers, string, etc.)

The search starts at the root. It probes down, comparing the values in each node with the target, till it finds the first item equal to the target. Returns this item or `null` if there is none.

Search in BST - Pseudocode

```
if the tree is empty
        return NULL

else if  the item in the node equals the target
        return the node value

else if  the item in the node is greater than the target
        return the result of searching the left subtree

else if  the item in the node is smaller than the target
        return the result of searching the right subtree
```

Search in a BST: C code

```
Ptnode search(ptnode root,
              int key)
{
/* return a pointer to the node that
   contains key. If there is no such
   node, return NULL */

  if (!root) return NULL;
  if (key == root->key) return root;
  if (key < root->key)
      return search(root->left,key);
  return search(root->right,key);
}
```

BST Operations: Insertion

`function insert(key)`

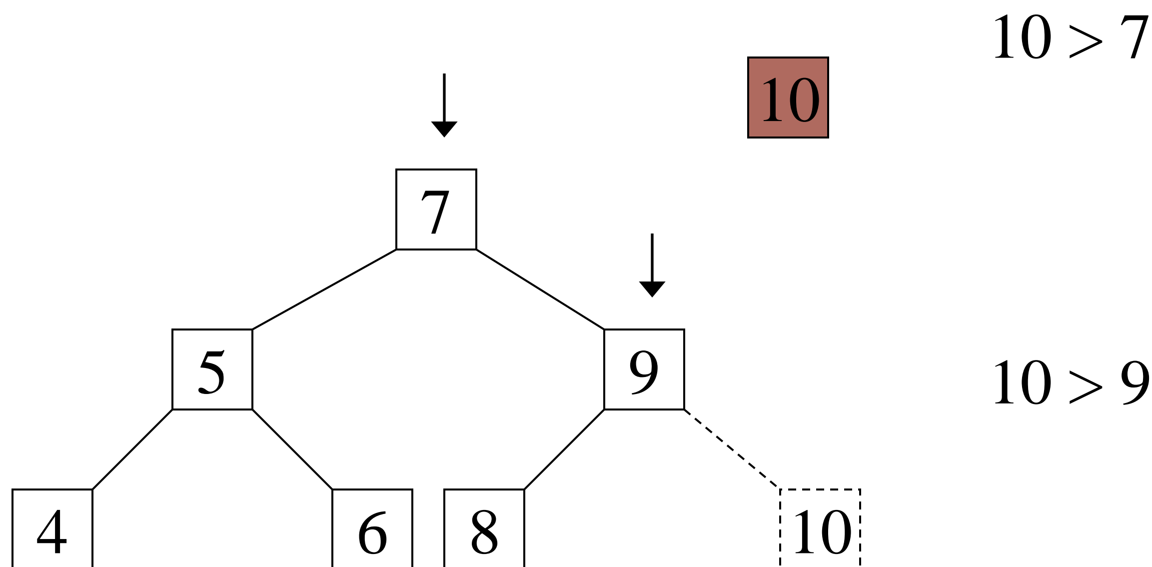
- places a new item near the frontier of the BST while retaining its organization of data:
  - **starting at the root** it probes **down** the tree till it finds a node whose left or right pointer is empty and is a logical place for the new value
  - uses a binary search to locate the insertion point
  - is based on comparisons of the new item and values of nodes in the BST
    - *Elements in nodes must be comparable!*

Case 1: The Tree is Empty

- Set the root to a new node containing the item

Case 2: The Tree is Not Empty

- Call a recursive helper method to insert the item



Insertion in BST - Pseudocode

```
if  tree is empty
      create a root node with the new key
else
      compare key with the top node
      if key =  node key
            replace the node with the new value
      else if  key >  node key
            compare key with the right subtree:
              if  subtree is empty create a leaf node
              else add key  in right subtree
       else  key <  node key
            compare key with the left subtree:
              if the subtree is empty create a leaf node
              else add key to the left subtree
```

Insertion into a BST: C code

```
void insert (ptnode *node, int key)
{
  ptnode ptr,
      temp = search(*node, key);
  if (temp || !(*node)) {
   ptr = (ptnode) malloc(sizeof(tnode));
   if (IS_FULL(ptr)) {
     fprintf(stderr, "The memory is full\n");
     exit(1);
   }
   ptr->key = key;
   ptr->left = ptr->right = NULL;
   if (*node)
     if (key<temp->key) temp->left=ptr;
        else temp->right = ptr;
   else *node = ptr;
  }
}
```

BST Shapes

- The order of supplying the data determines where it is placed in the BST , which determines the shape of the BST
- Create BSTs from the same set of data presented each time in a different order:

a) 17 4 14 19 15 7 9 3 16 10

b) 9 10 17 4 3 7 14 16 15 19

c) 19 17 16 15 14 10 9 7 4 3 can you **guess** this shape?

BST Operations: Removal

- **removes** a specified item from the BST and **adjusts** the tree
- uses a binary search to locate the target item:
  - **starting at the root** it probes down the tree till it finds the target or reaches a leaf node (target not in the tree)

- removal of a node must not leave a ‘gap’ in the tree,

Removal in BST - Pseudocode

```
method remove (key)
```

I   if the tree is empty return false

II Attempt to locate the node containing the target using the binary search algorithm

if the target is not found return false

else the target is found, so remove its node:

Case 1: if the node has 2 empty subtrees
replace the link in the parent with null

Case 2: if the node has a left and a right subtree
- replace the node's value with the max value in the left subtree
- delete the max node in the left subtree

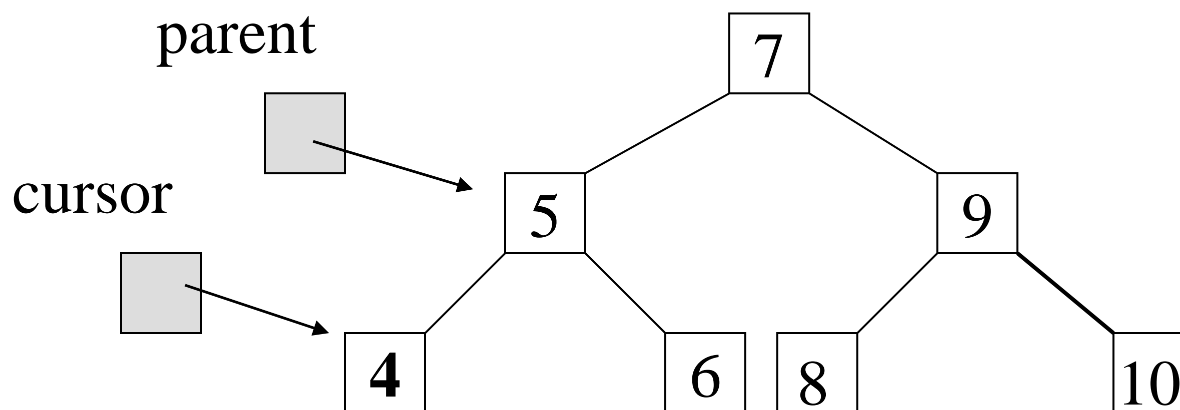
Removal in BST - Pseudocode

Case 3: if the node has no left child
- link the parent of the node
- to the right (non-empty) subtree

Case 4: if the node has no right child
- link the parent of the target
- to the left (non-empty) subtree

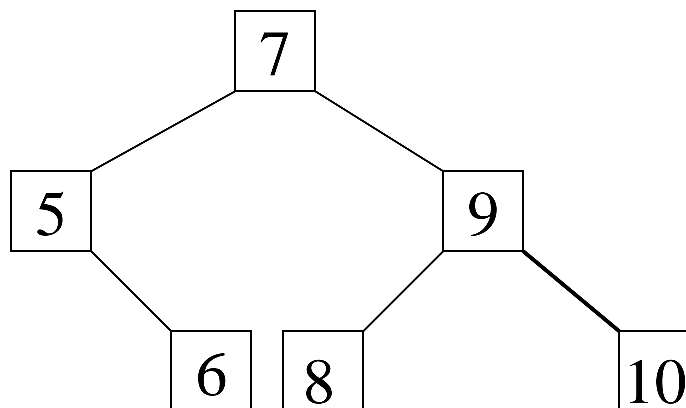
Removal in BST: Example

Case 1: removing a node with 2 EMPTY SUBTREES



Removing 4

replace the link in the parent with `null`



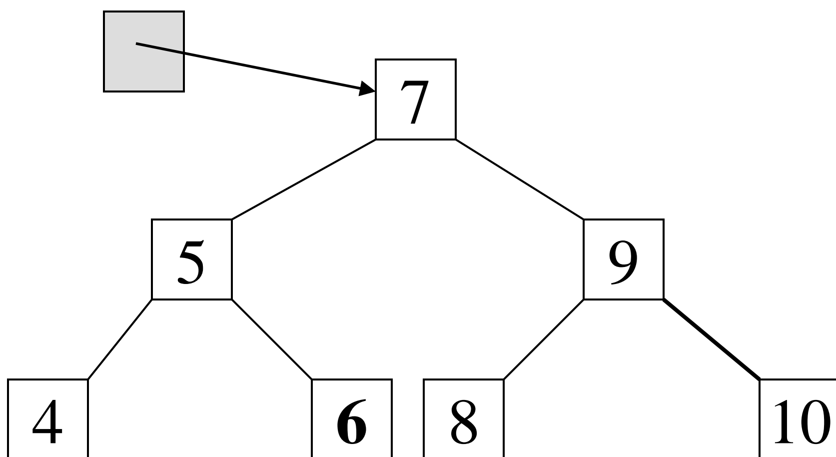
Removal in BST: Example

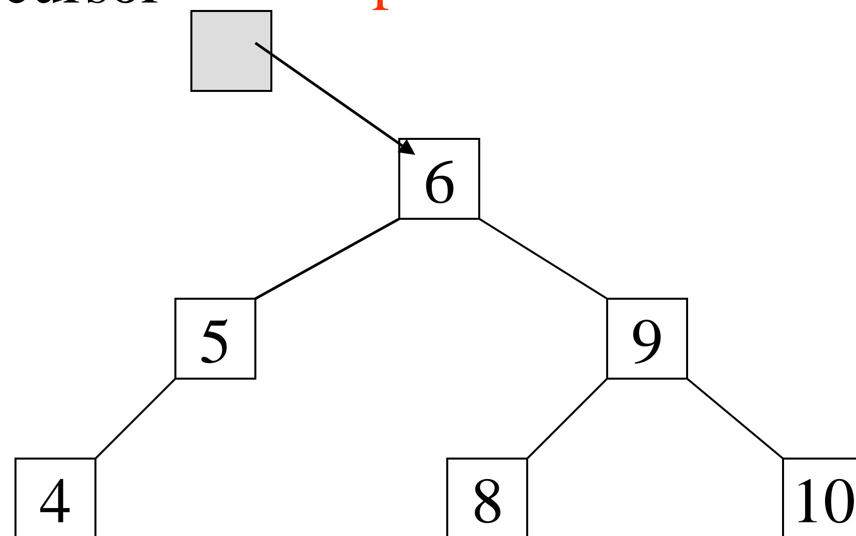
Case 2: removing a node with 2 SUBTREES

- replace the node's value with the max value in the left subtree
- delete the max node in the left subtree

What other element can be used as replacement?

Removing 7

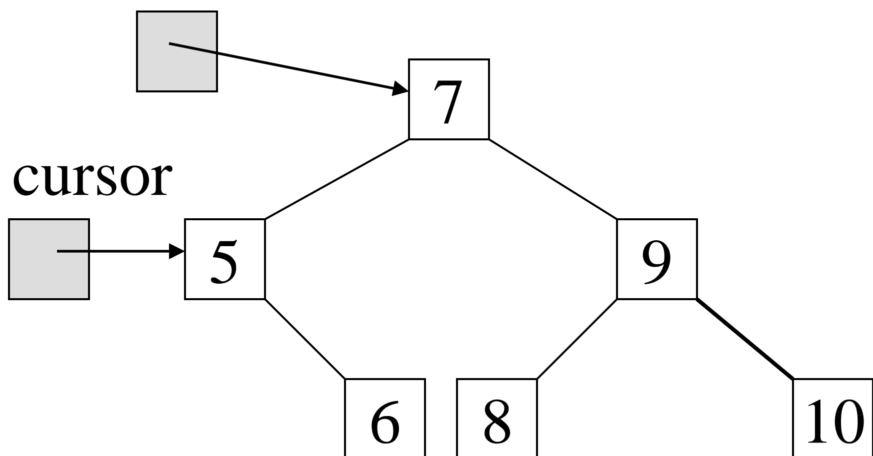


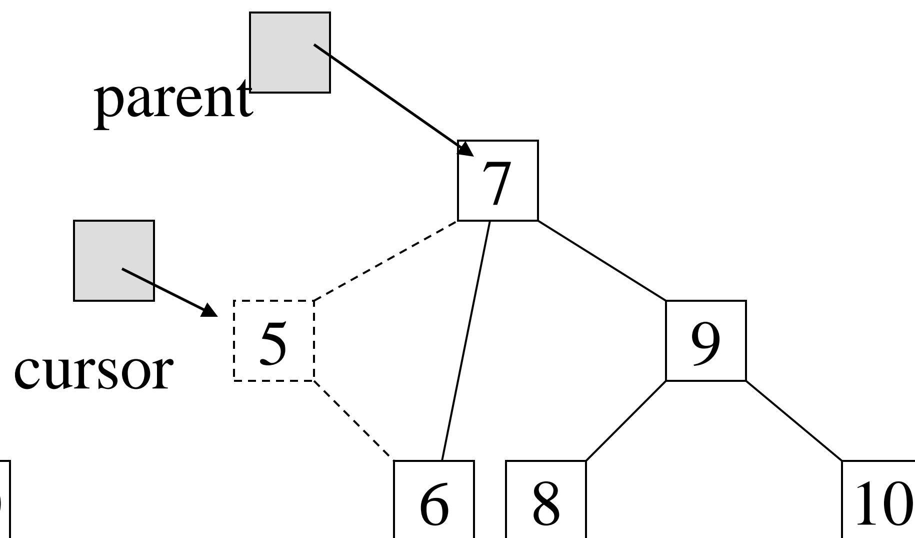


Removal in BST: Example

Case 3: removing a node with 1 EMPTY SUBTREE

the node has no left child:
link the parent of the node to the right (non-empty) subtree





Removal in BST: Example

Case 4: removing a node with 1 EMPTY SUBTREE

the node has no right child:
link the parent of the node to the left (non-empty) subtree

Removing 5

Analysis of BST Operations

- The complexity of operations **`get`**, **`insert`** and **`remove`** in BST is O(h) , where h is the height.
- O(log n) when the tree is balanced. The updating operations cause the tree to become unbalanced.
- The tree can degenerate to a linear shape and the operations will become O (n)

Best Case

```
BST tree = new BST();

tree.insert ("E");
tree.insert ("C");
tree.insert ("D");
tree.insert ("A");
tree.insert ("H");
tree.insert ("F");
tree.insert ("K");
```

Output:

```
>>>> Items in advantageous order:
            K
        H
            F
E
            D
        C
            A
```

Worst Case

```
BST tree = new BST();
for (int i = 1; i <= 8; i++)
   tree.insert (i);
```

Output:

```
>>>> Items in worst order:
        8
       7
      6
     5
    4
   3
  2
 1
```

Random Case

```
tree = new BST ();
for (int i = 1; i <= 8; i++)
   tree.insert(random());
```

Output:

```
>>>> Items in random order:
     X
          U
P
          O
     H
          F
                    B
```

Applications for BST

- Sorting with binary search trees
  - Input: unsorted array
  - Output: sorted array

- Algorithm ?
- Running time ?

Better Search Trees

Prevent the degeneration of the BST :

- A BST can be set up to maintain balance during updating operations (insertions and removals)
- Types of ST which maintain the optimal performance:
  - splay trees
  - AVL trees
  - 2-4 Trees
  - Red-Black trees
  - B-trees