Binary Trees

L.EIC

Algoritmos e Estruturas de Dados

2023/2024

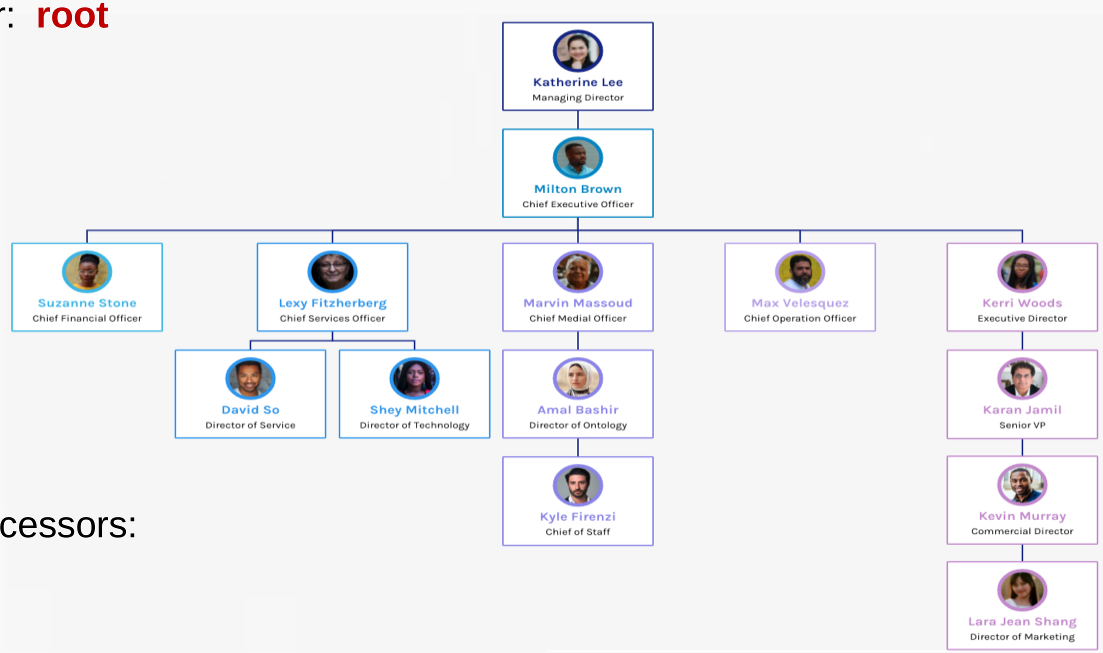
AP Rocha, P Diniz

A Costa, B Leite, F Ramos, J Pires, PH Diniz, V Silva

Tree

- **Tree**: set of nodes and set of edges connecting pairs of nodes
  - each node may have 0 or more successors (children)
  - each node has exactly one predecessor (parent)
    - except the starting node, called the root
  - There is a unique path from the root to each node; the path length for a node is the number of edges to traverse

Node without predecessor: **root**

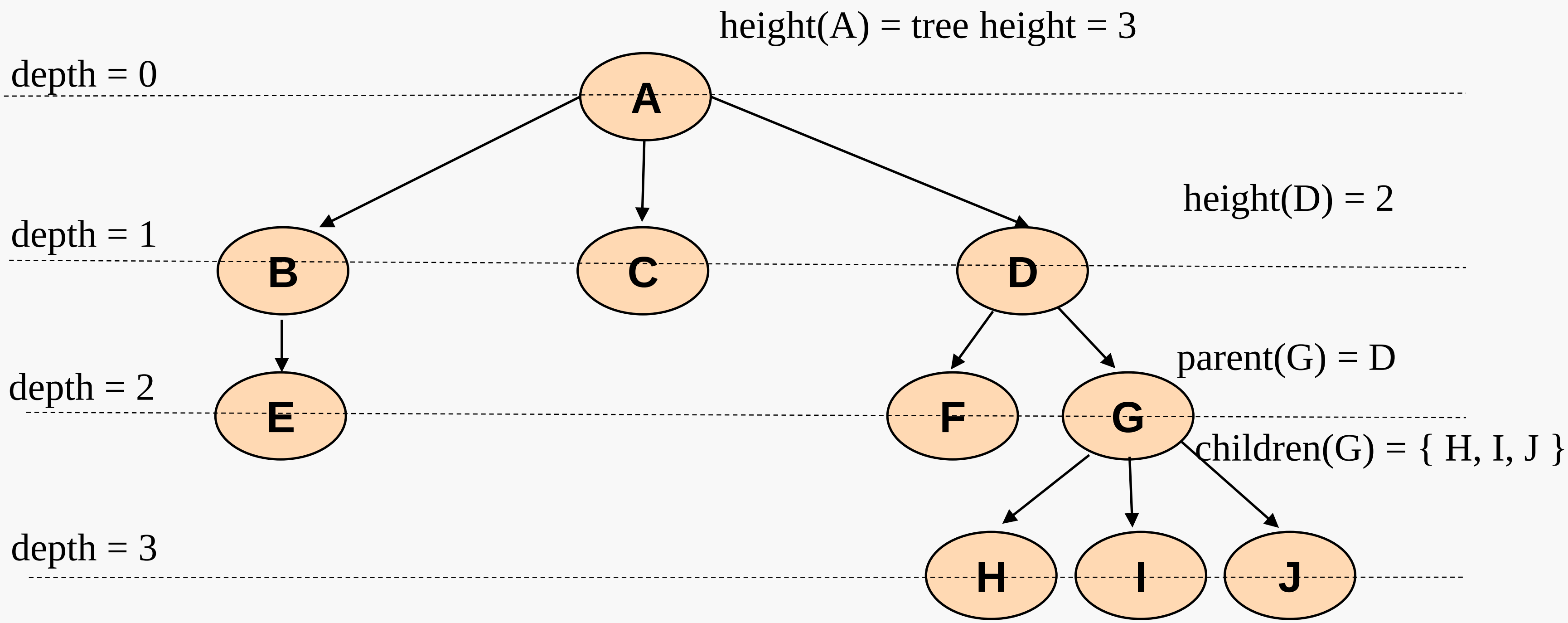


Nodes without successors: **leaves**

Tree terminology

- Tree branches
  - links from nodes to its children: tree of n nodes has $n-1$ branches (max)
- Depth of a node
  - Length of path from root to node
    - depth of root is 0
    - depth of a node = 1 + depth of its parent
- Height of a node
  - Length of path from node to its deepest leaf (descendent)
    - height of a leaf is 0
    - height of a node = 1 + height of children of greatest height
  - Height of the tree: height of the root
- If there is a path from node u to node v
  - u is ancestor of v
  - v is descendent of u
- Size of a node: number of descendents

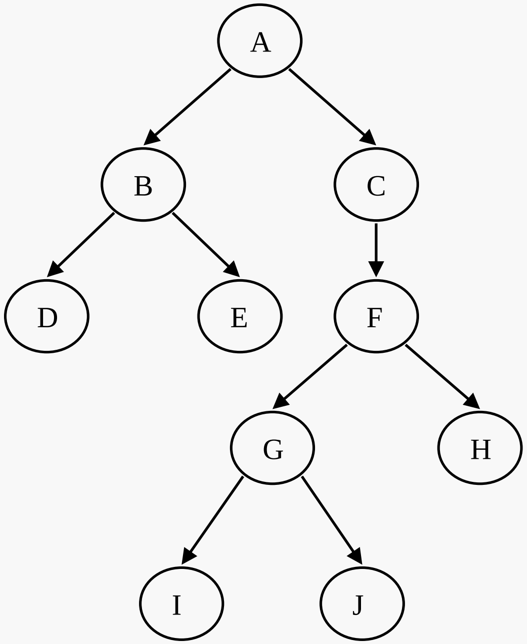
Tree terminology



Binary Tree

A binary tree is a tree where each node has at most two children

- Some properties:
  - A non-empty binary tree with depth h has:
    - at least $h + 1$ nodes
    - at most $2^{h+1} - 1$ nodes
  - The depth of a binary tree with n elements $(n > 0)$ is:
    - at least $log_2 n$
    - at most $n - 1$
  - The average depth of a tree of n nodes is $\sqrt{n}$



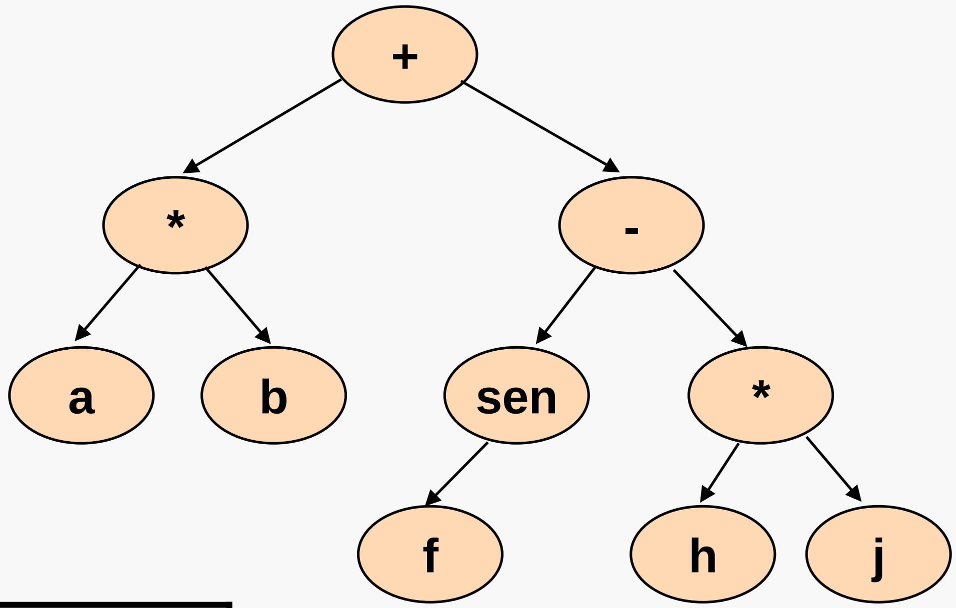
Binary Tree

- Tree traversals
  - To traverse the binary tree is to visit each node in the binary tree exactly once
  - There are four different ways to traverse the binary tree, the first three are defined recursively:

- Preorder: visit the root first, then the left subtree, and finally the right subtree
- Inorder: visit the left subtree first, then the root, and finally the right subtree
- Postorder: visit the left subtree first, then the right subtree, and finally the root
- By level: nodes are visited by increasing level (depth), and within each level, from left to right

Binary Tree

Tree traversal - example



Preorder	+ * a b – sen f * h j
Inorder	a * b + f sen – h * j
Postorder	a b * f sen h j * - +
Level	+ * - a b sen * f h j

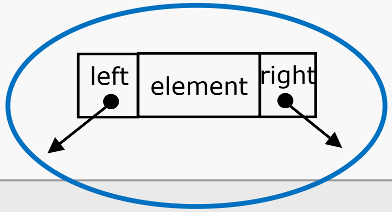
Binary Tree

- **Binary Tree**

  – most usual operations:
    - create an empty tree
    - determine if the tree is empty
    - create a tree from two subtrees
    - remove the elements of the tree (clear the tree)
    - traversal the tree
    - print the tree

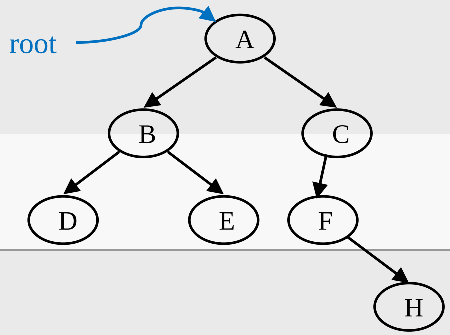
Binary Tree: a possible implementation

Node of the binary tree




```
template <class T> class BTNode {
   T element;
   BTNode<T>* left,* right;
   //…

public:
   BTNode(const T& el, BTNode<T>* l = 0, BTNode<T>* r = 0)
      : element(el), left(l), right(r) {}
};
```

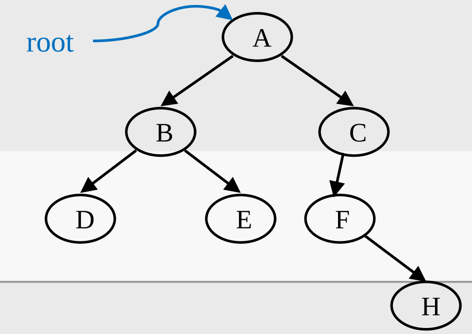
Binary Tree: a possible implementation



Binary tree

```
template <class T> class BinaryTree {
   BTNode<T>* root;
   // …
public:
   BinaryTree() { root = 0; }
   BinaryTree(const BinaryTree & t);
   BinaryTree(const T& el);
   BinaryTree(const T& el, const BinaryTree<T>& l,
                              const BinaryTree<T>& r);
   ~BinaryTree { makeEmpty(); }
   bool isEmpty() const;
   T& getRoot() const;
   void makeEmpty();
   void output(ostream& out) const;
};
```

Binary Tree: a possible implementation



Binary tree: traversal

```
template <class T>
void BinaryTree<T>::output(ostream& out) const {
   output(out, root);
}

template <class T>
void BinaryTree<T>::output(ostream& out,const BTNode<T>* n) const {
   if (n) {
      out << n->element << ' ';
      output(out, n->left);
      output(out, n->right);
   }
}
```

Tree traversal?

Binary Tree: iterators

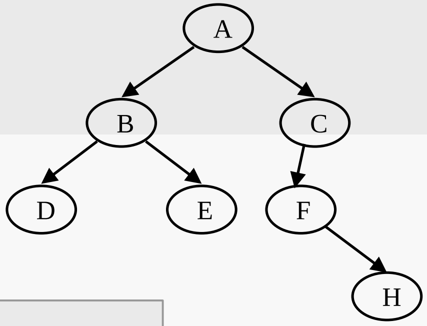
What about *iterators?*

Implementation:

- Constructor has the tree to iterate as a parameter; iterator is referencing the first element.
- methods:
  - void advance()   //go to the next element
  - T& retrieve()   //return the element referenced by the iterator
  - bool isAtEnd() //check if have reached the end

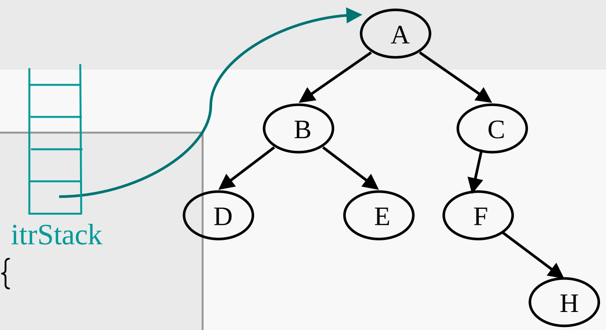
- An implementation for each possible tree traversal:
  - preorder, *BTItrPre*   (let's see this implementation)
  - inorder, *BTItrIn*
  - postorder, *BTItrPos*
  - by level, *BTItrLevel*

Preorder iterator: a possible implementation




```
template <class T> class BTItrPre {
   stack<BTNode<T>*> itrStack;
public:
   BTItrPre(const BinaryTree<T>& t);
   void advance();
   T& retrieve();
   bool isAtEnd() const;
};

template <class T>
BTItrPre<T>::BItrPre(const BinaryTree<T>& t) {
   if ( !t.isEmpty() )
      itrStack.push(t.root);
}
```

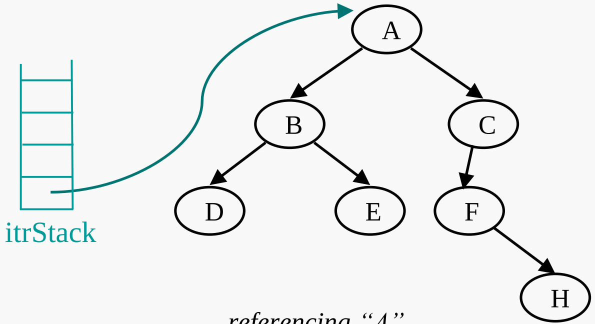
Preorder iterator: a possible implementation



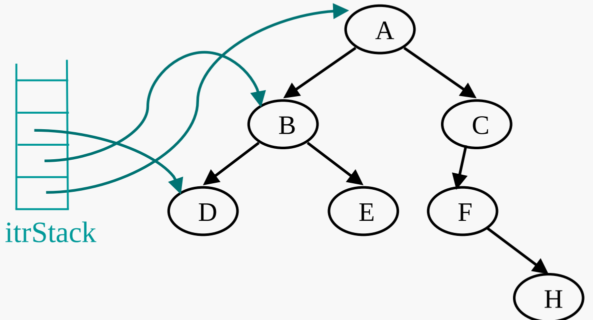

```
template <class T>
BTItrPre<T>::BItrPre(const BinaryTree<T>& t) {
   if ( !t.isEmpty() )
      itrStack.push(t.root);
}

template <class T>
bool BTItrPre<T>::isAtEnd() const {
   return itrStack.empty();
}

template <class T>
T& BTItrPre<T>::retrieve() {
   return itrStack.top()->element;
}
```

Preorder iterator: a possible implementation

```
template <class T>
void BTItrPre<T>::advance() {
   BTNode<T>* actual = itrStack.top();
   BTNode<T>* next = actual->left;
   if (next)
      itrStack.push(next);
   else {
      while ( ! itrStack.empty() ) {
         actual = itrStack.top();
         itrStack.pop();
         next = actual->right;
         if (next) {
            itrStack.push(next);
            break;
         }
      }
   }
}
```



referencing "A"



referencing "D"

Binary Search Trees

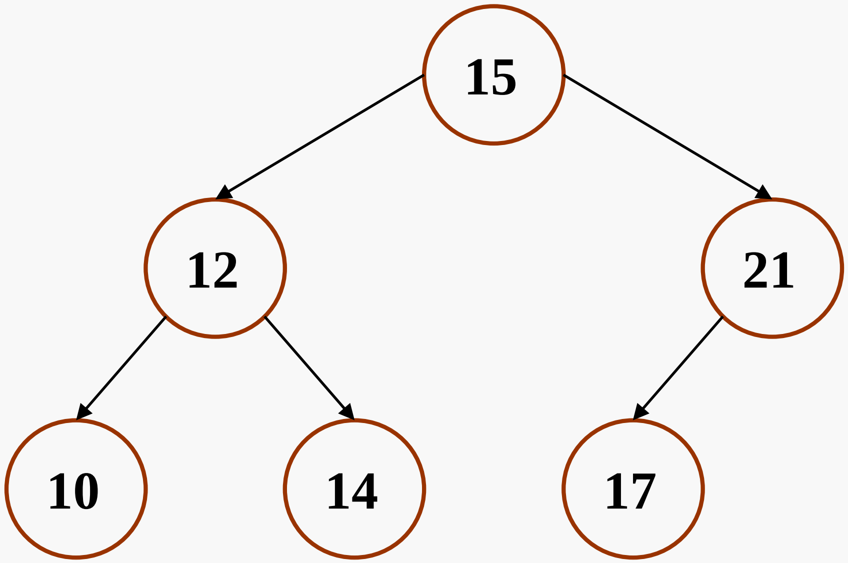
Binary Search Tree (BST)

- **Binary Search Tree**

  Binary tree, without duplicates. For every node (X):

  – The values of all the items in its left subtree are smaller than the item in X

  – The values of all the items in its right subtree are larger than the item in *X*



Binary Search Tree

- In ***Linear Data Structures*** with sorted elements:
  - Search for elements can be achieved in $O(\log n)$
  - ...but not insertion or deletion of elements

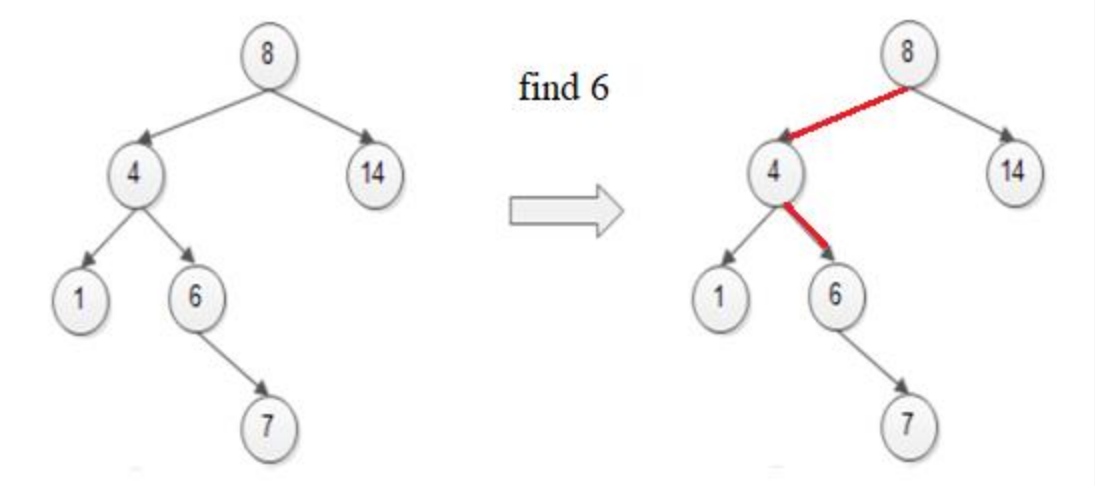
- In a ***Binary Tree*** structure:
  - Logarithmic running time *can be* guaranteed for insertion and deletion operations (if balanced tree)
  - A ***Binary Search Tree*:**
    - Supports more operations compared to the basic Binary Tree structure, namely: search, insertion and deletion
    - Elements in nodes must be comparable (*Comparable*)

Binary Search Tree

- **Search**
  - Takes advantage of the order property of the tree to select a search path.

    A subtree is ignored with each comparison.

    can be $O(\log n)$

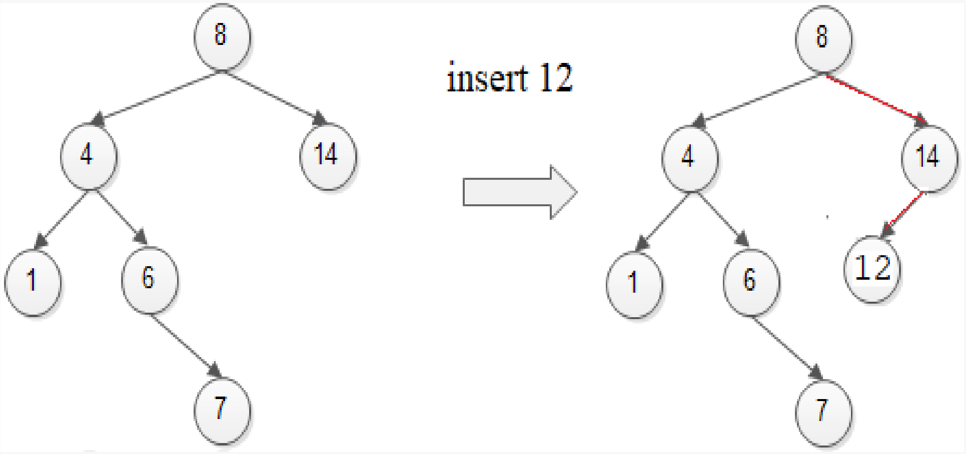


- **Maximum** and **Minimum**
  - When looking for the maximum: consistently choose the right subtree.
  - When looking for the minimum: consistently choose the left subtree.

Binary Search Tree

- **Insertion**
  - Follows the logic behind the search operation. The new node is inserted where the search fails.

    can be $O(\log n)$

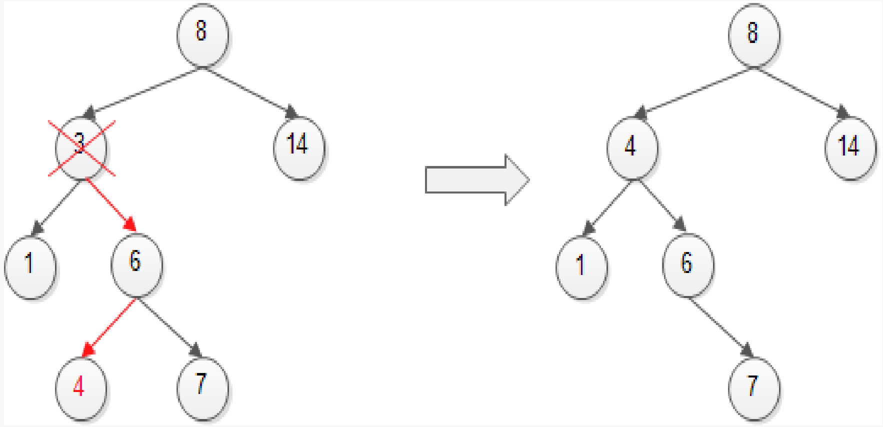


see animation in [VisuAlgo]

Binary Search Tree

- **Deletion**
  - Leaf Node: delete the node
  - Node with only one child: child node replaces the parent node
  - Node with two children: element is replaced by the smallest element in the right subtree (or the largest element in the left subtree); its node has at most one child node that replaces the parent node.

can be $O(\log n)$



see animation in VisuAlgo

Binary Search Tree: a possible implementation

- Declaration of the ***BST*** class (private section)

```
template <class Comparable> class BST {
    BinaryNode<Comparable> *root;
    const Comparable ITEM_NOT_FOUND;

    const Comparable & elementAt(BinaryNode<Comparable> *t) const;
    bool insert(const Comparable & x, BinaryNode<Comparable>* & t);
    bool remove(const Comparable & x, BinaryNode<Comparable>* & t);
    BinaryNode<Comparable> * findMin(BinaryNode<Comparable>*t) const;
    BinaryNode<Comparable> * findMax(BinaryNode<Comparable>*t) const;
    BinaryNode<Comparable> * find( const Comparable & x,
                                              BinaryNode<Comparable>*t) const;
    void makeEmpty( BinaryNode<Comparable>* & t );
    void printTree( BinaryNode<Comparable>*t ) const;
    BinaryNode<Comparable>* copySubTree(BinaryNode<Comparable>* t);


    // continue...
```

Binary Search Tree: a possible implementation

- Declaration of the ***BST*** class (public section)

construtor

```
public:
    explicit BST(const Comparable& notFound) { }
    BST(const BST & t);
    ~BST();
    const Comparable& findMin() const;
    const Comparable& findMax() const;
    const Comparable& find(const Comparable& x) const;
    bool isEmpty() const;
    void printTree() const;
    void makeEmpty();
    bool insert(const Comparable& x);
    bool remove(const Comparable& x);
    const BST& operator =(const BST& rhs);
};
```

Binary Search Tree: a possible implementation

- BST Class: constructor and destructor

```
template <class Comparable>
BST<Comparable>::BST(const Comparable& notFound):
                    root(NULL), ITEM_NOT_FOUND(notFound)
{ }

template <class Comparable>
BST<Comparable>::~BST( ){
   makeEmpty( );
}
```

Binary Search Tree: a possible implementation

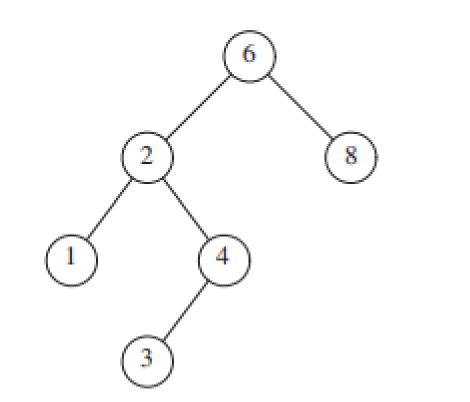
- BST Class: *find*

```
template <class Comparable>
const Comparable & BST<Comparable>::find(const Comparable& x) const {
   return elementAt( find(x, root) );
}


template <class Comparable>
const Comparable&
 BST<Comparable>::elementAt(BinaryNode<Comparable>* t) const{
   if( t == NULL ) return ITEM_NOT_FOUND;
      else return t->element;
}
```

Binary Search Tree: Implementation

- BST Class: *find*

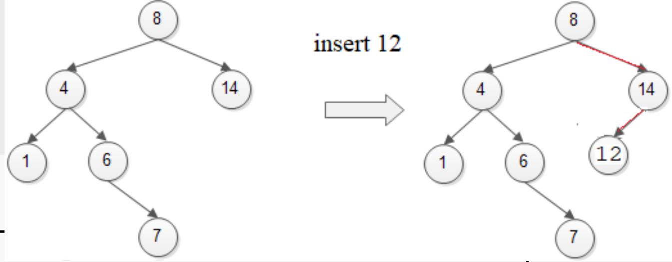
```
template <class Comparable>
BinaryNode<Comparable> * BST<Comparable>::find(const Comparable& x,
                               BinaryNode<Comparable>* t) const
{
   if ( t == NULL )
      return NULL;
   else if ( x < t->element )
      return find(x, t->left);
   else if ( t->element < x )
      return find(x, t->right);
   else return t;
}
```



Note: we only need the < operator

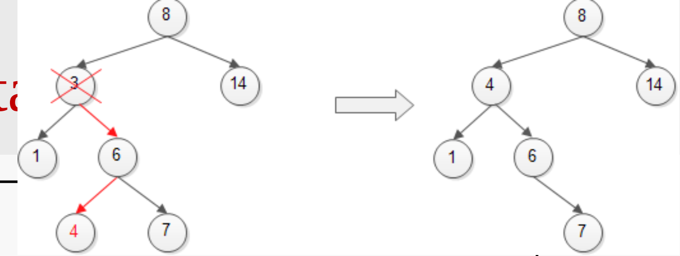
Binary Search Tree: Implementation

- BST Class: *insert*




```
template <class Comparable>
bool BST<Comparable>::insert(const Comparable& x) {
   return insert (x,root);
}

template <class Comparable>
bool BST<Comparable>::insert(const Comparable& x,
                                  BinaryNode<Comparable>* & t) {

   if ( t == NULL ) {
      t = new BinaryNode<Comparable>(x, NULL, NULL);
      return true;
   }
   else if ( x < t->element)
      return insert(x, t->left);
   else if (t->element < x)
      return insert(x, t->right);
   else
      return false;    // duplicate; do nothing
}
```

Binary Search Tree: a possible implementa



- BST Class: *remove*

```
template <class Comparable>
bool BST<Comparable>:: remove(const Comparable & x,
                              BinaryNode<Comparable> * & t) {
   if ( t == NULL )
      return false;  // item not found; do nothing
   if ( x < t->element )
      return remove(x, t->left);
   else if ( t->element < x )
      return remove(x, t->right);
   else if ( t->left !=  NULL && t->right != NULL ) {
      t->element = findMin(t->right)->element;
      return remove(t->element, t->right);
   }
   else {
      BinaryNode<Comparable>* oldNode = t;
      t = ( t->left != NULL ) ? t->left : t->right;
      delete oldNode;
      return true;
  }
}
```

Class *set* (STL)

Class ***set*** in STL:

template<Key,Compare> class set

- *Key*: type of the item in the set
- *Compare*: determines the ordering of the keys; by default is the operator <

- distinct, ordered data stored in a *balanced** binary tree
- implemented as a *red black ** tree
- inorder iterator

en.cppreference.com/w/cpp/container/set

* we will see this later

Class *set* (STL)

Some methods of *set* (STL)

- iterator **begin**()
- iterator **end**()
- size_type **size**() const
- bool **empty**() const
- void **clear()**
- std::pair<iterator,bool> **insert**(const value_type& value)
- iterator **erase**(const_iterator position);
- iterator **find**(const Key& key)
- …

Binary Search Tree: example 1

- Counting word occurrences

We want to write a program that reads a text file and outputs an ordered listing of the words it contains, as well as their number of occurrences.

- Store the words and associated counters in a binary search tree.
- Use alphabetical order to compare nodes.

Binary Search Tree: example 1

WordFreq class : representation of the words and respective frequencies

wordFreq.h

```
class WordFreq {
    string word;
    int frequency;
public:
    WordFreq();
    WordFreq(string w);
    bool operator < (const WordFreq & w) const;
    bool operator == (const WordFreq & w) const;
    void incFrequency();
    friend ostream& operator <<(ostream& out,const WordFreq& w);
};
```

Binary Search Tree: example 1

wordFreq.cpp

```
WordFreq::WordFreq(): word(""), frequency(0) {};

WordFreq::WordFreq(string w): word(w), frequency(1) {};


bool WordFreq::operator < (const WordFreq & w) const  {
    return word < w.word;
}


bool WordFreq::operator == (const WordFreq & w) const {
    return word == w.word;
}


void WordFreq::incFrequency() {
    frequency ++;
}
```

Binary Search Tree: example 1

```
int main() {
   set<WordFreq> words;
   string word1 = getWord();
   while ( word1 != "" ) {
      set<WordFreq>::iterator it  = words.find(WordFreq(word1));
      if ( it == words.end() )
         words.insert(WordFreq(word1));
      else {
         WordFreq found = *it;
         found.incFrequency();
         words.erase(it);
         words.insert(found);
      }
      word1 = getWord();
   }
   set<WordFreq>::iterator it = words.begin();
   while ( it!=words.end() ) {
      cout << *it;
      it++;
   }
}
```

Binary Search Tree: example 2

In a library, information about existing books is stored in a binary search tree (books) sorted by author and, for the same author, by title.

```
class Book {
   string title;
   string author;
public:
   // …
};

class Library {
  set<Book> books;
public:
  Library();
  // …
};
```

Binary Search Tree: example 2

- Implement in the **Library** class the member function:

  void addBooks(const vector<Book> & books1)

  This function inserts into the BST *books* the books included in the *books1* vector.

```
void Library::addBooks(const vector<Book>& books1) {
    for (int i=0; i< books1.size(); i++)
        books.insert(books[i]);
}
```

```
bool Book::operator < (const Book& l1) const {
    if (author == l1.author)
        return (title < l1.title);
    else
        return (author < l1.author);
}
```

Binary Search Tree: example 2

- Implement in the **Library** class the member function:

```
vector<string> getTitles(string author1, int & nVisits)
```

The function returns a vector with the titles, in alphabetical order, of all the books of author *author1*. And returns in *nVisits* the number of nodes that were visited in this search.

```
vector<string> Library::getTitles(string author1, int& nVisits) {
    vector<string> vres;
    nVisits=0;
    set<Book>::iterator it = books.begin();
    for(; it!=books.end(); it++) {
        nVisits++;
        if (it->getAuthor() > author1)
            break;
        if (it->getAuthor() == author1)
            vres.push_back(it->getTitle());
    }
    return vres;
}
```

Binary Search Tree: example 2

- Implement in the **Library** class the member function:

```
string removeBook(string author1, string title1)
```

The function removes the book of author *author1* and title *title1* from BST books and returns the string "*removed*".

If the book does not exist, it returns the title of the first book (alphabetically) of that author.

If no book of that author exists, it returns the string "*author nonexistent*".

Binary Search Tree: example 2

```
string Library::removeBook(string author1, string title1){
   Book l1(title1, author1);
   set<Book>::iterator it = books.find(l1);
   if (it!=books.end()) {
       books.erase(it);
       return "removed";
    }

    set<Book>::iterator it = books.begin();
          for (;it!=books.end(); it++) {
       if (it->getAuthor() == author1)
          return it->getTitle();
       if (it->getAuthor() > author1)
             return "author nonexistent";
    }
    return "author nonexistent";
}
```

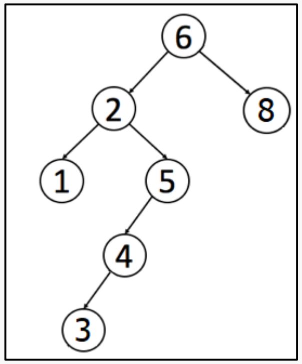
Balanced Binary Search Trees

Binary Search Tree

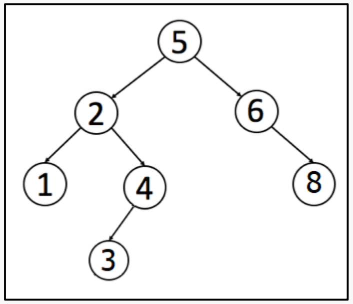
- ***Binary Search Tree***
  - tree can be unbalanced
  - insertion, removal and searching operations can be $O(n)$ in the worst case, when tree degenerates into list

Balanced tree: for each node, the heights of the left and right subtrees differ by at most one unit

- ***Balanced Binary Search Tree***
  - avoid degenerate cases
  - guarantees $O(\log n)$ for insertion, removal and searching

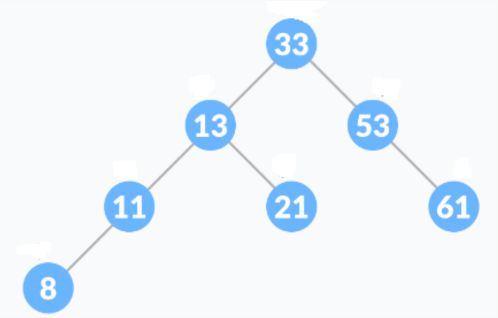


unbalanced BST



balanced BST

AVL Tree



- AVL Tree
  - Balanced binary search tree

- Insertion and removal identical to BST
  - But the tree might become unbalanced

Apply *rotations* along the path on the unbalanced nodes in order to keep it balanced

- Searching identical to BST

AVL: insertion

- **Insertion**

  – After an insertion, only nodes on the path from the root to have the balance condition changed.

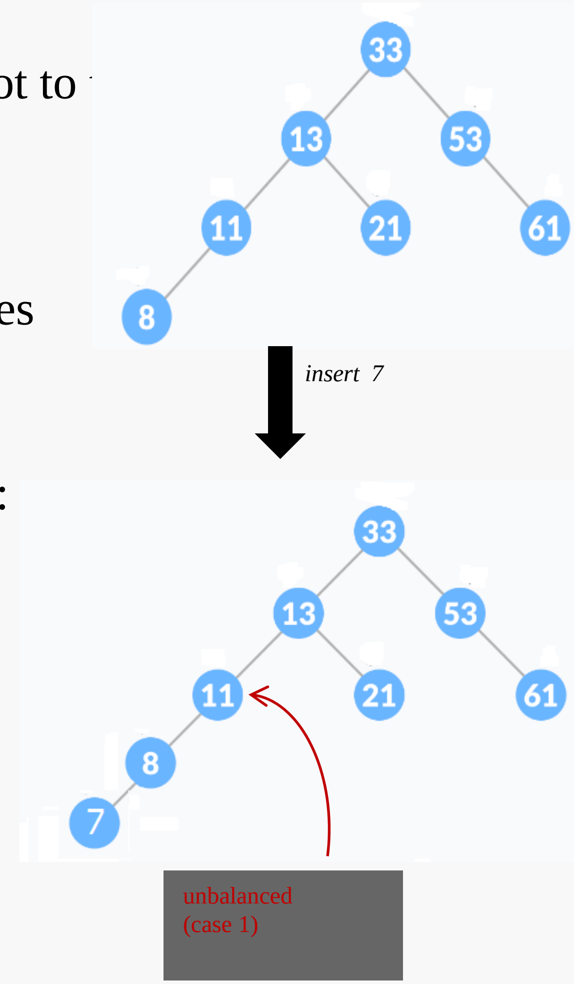
  – If so, it is necessary to rebalance

    - rebalance the deepest node where unbalance arises
    - the whole tree is balanced

  – Let K be the node to be rebalanced due to insertion in:

    1. left tree of the left child of K
    2. right tree of the left child of K
    3. left tree of the right child of K
    4. right tree of the right child of K

  *cases 1 and 4 are symmetrical; * cases 2 and 3 are symmetrical;



AVL: insertion

- **Single rotation** (for cases 1 and 4)

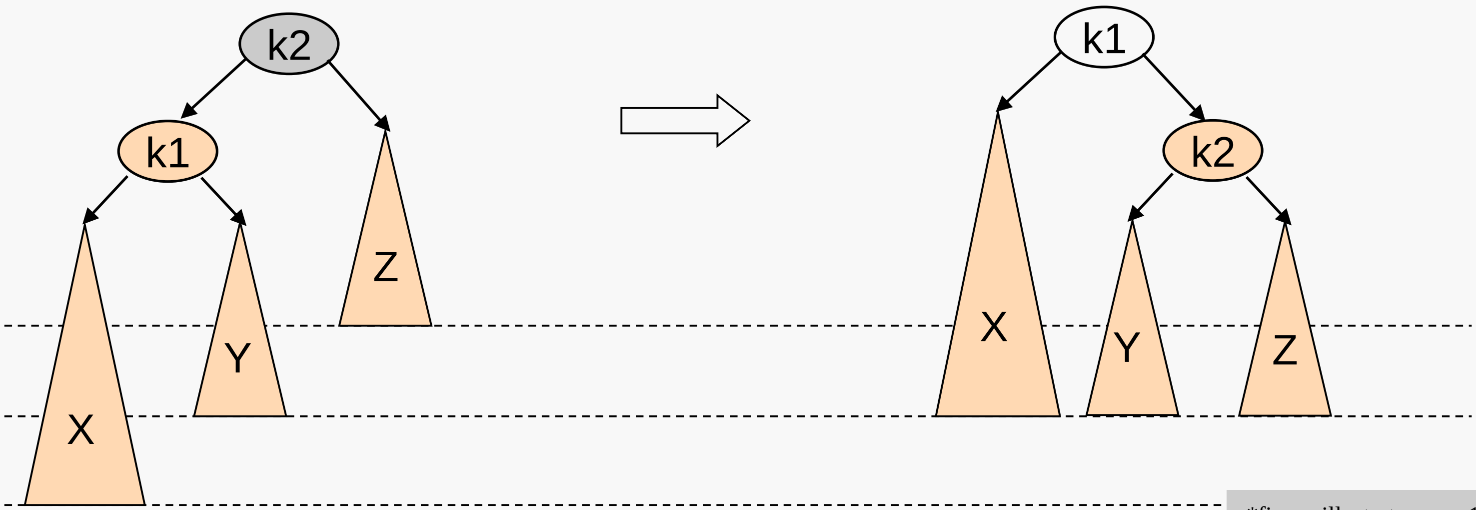
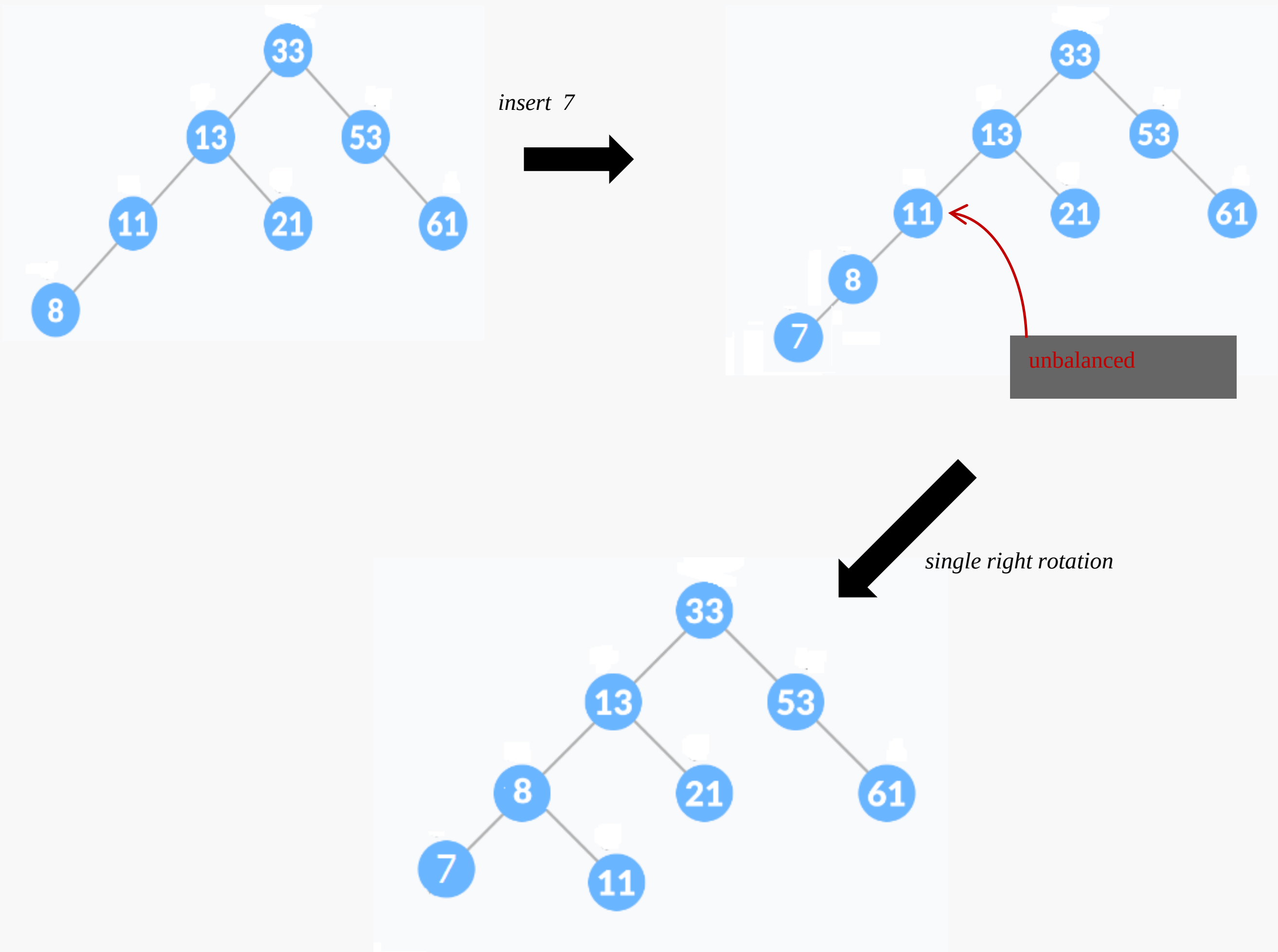


*figure illustrates case 1

- *before rotation*:
  - k2 is the deepest node where balance fails
  - left subtree is 2 levels below the right
- *after rotation*:
  - k1 and k2 now have subtrees of the same height
  - problem is solved with a single operation

AVL: insertion



see animation in [VisuAlgo]()

AVL: insertion

- **Double rotation** (for cases 2 and 3)

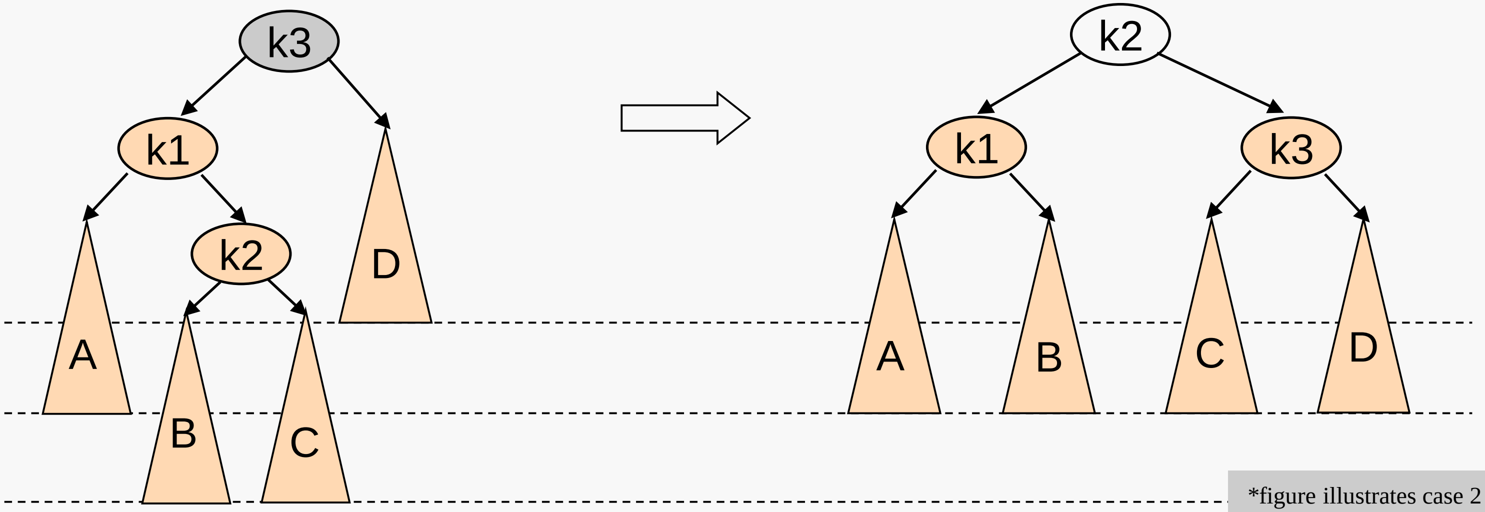


*figure illustrates case 2

- *before rotation*:
  - one (and only one) of sub-trees B or C is 2 levels apart from D
- *double rotation* can be seen as a sequence of 2 single rotations
  - rotation between the node's child and grandchild
  - rotation between the node and its new child

AVL: a possible implementation

AVL Tree **Node**

```
template <class Comparable>
class AVLNode {
      Comparable element;
      AVLNode *left, *right;
      int height;
public:
      AVLNode(const Comparable &e, AVLNode *esq = 0,
              AVLNode *dir = 0, int h = 0);
      //…
};
```

AVL Tree

```
template <class Comparabe>
class AVLTree {
      AVLNode<Comparable> *root;
      //…
};
```

AVL

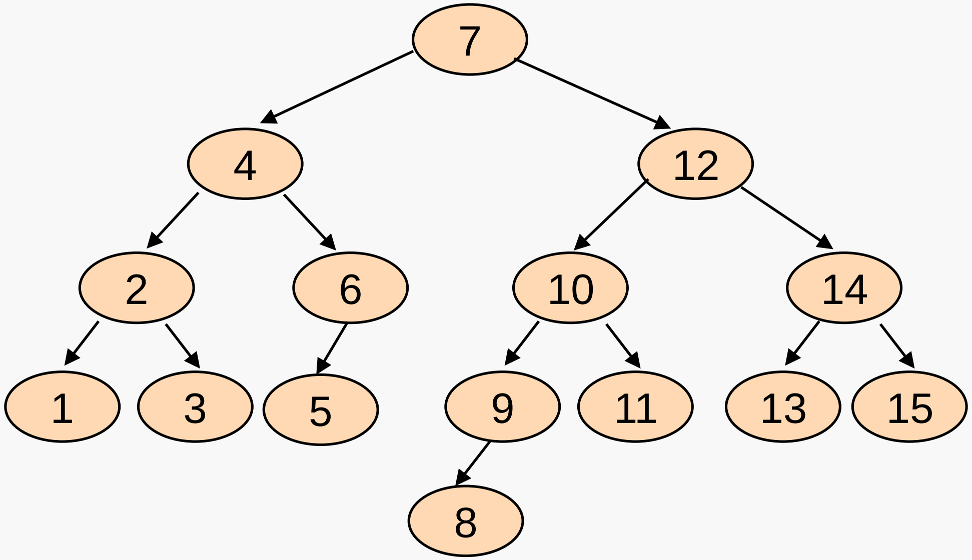
Insertion in a AVL tree has complexity $O(\log n)$:

- $O(\log n)$ to get the insertion position
- $O(1)$ to eventually rebalance the tree

Exercise: build the AVL tree that results from inserting the following sequence of values:

1, 2, 3, 4, 5, 6, 7, 15, 14, 13, 12, 11, 10, 9, 8

result:



AVL: removal

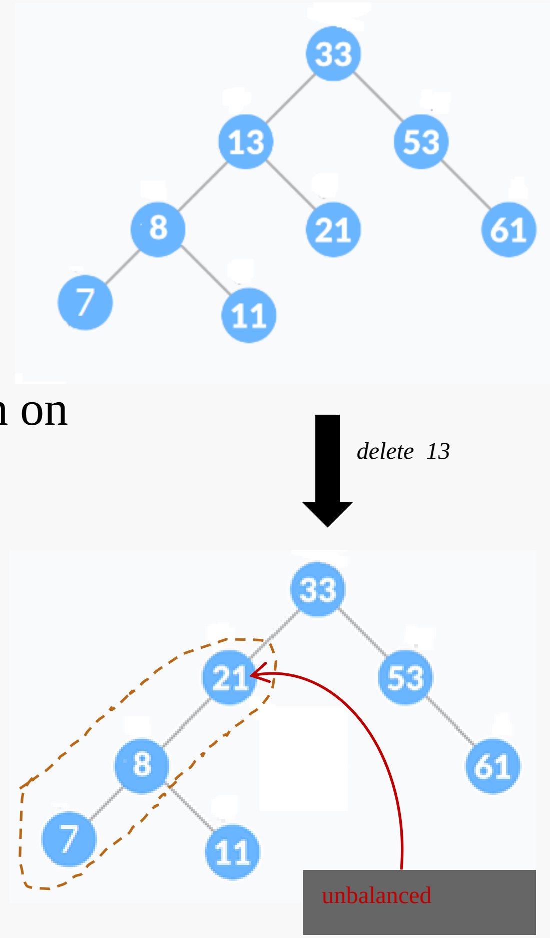
- **Removal**

  Same idea as insertion:

  – find the node to remove

  – remove the node (same process as in BST)

  – apply rotations, as already described, along the path on all unbalanced nodes until reach the root

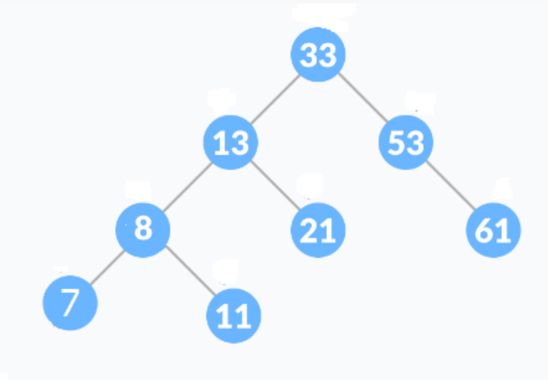


Removal in a AVL tree has complexity $O(\log n)$:

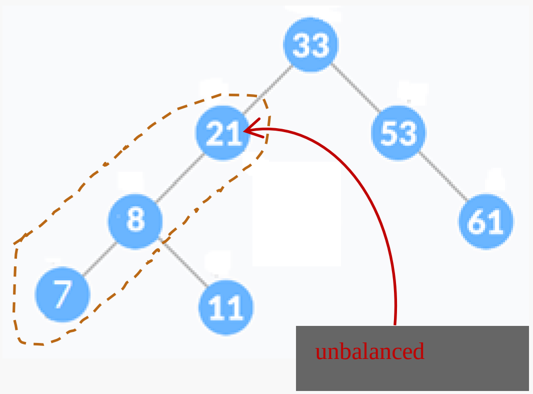
- $O(\log n)$ to find the node to be deleted
- $O(1)$ to eventually rebalance the tree

0

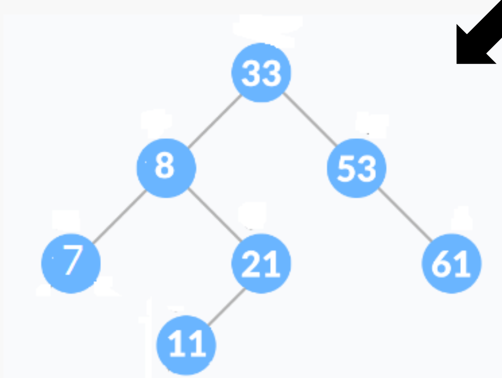
AVL: removal



delete 13



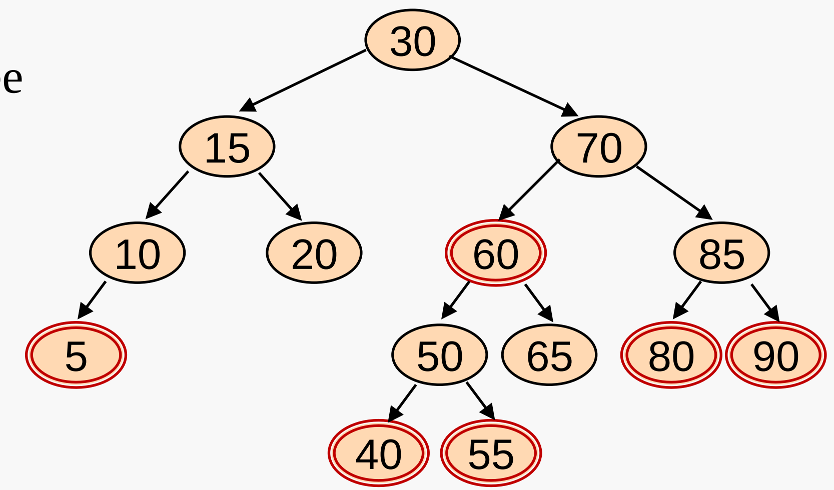
single right rotation



Red-Black Trees

Red-Black tree

- another type of balanced binary search tree
- operations have complexity $O(\log n)$



- ***Properties*** of a Red-Black tree
  - each node is colored as red or black
  - the root is black
  - if a node is red, its children are black
  - any path from a node to an empty subtree contains the same number of black nodes

  - height of a Red Black tree is at most = $2 \times \log(n + 1)$
    - ensures that search operation is of logarithmic order.

Class *set* (STL)

Class ***set*** in STL:

template<Key,Compare> class set

- *Key*: type of the item in the set
- *Compare*: determines the ordering of the keys; by default is the operator <

- implemented as a **red-black tree**