Section 8

C++ Operator Overloading & Lambdas, Binary Search Trees (BST), Sets (ADT) and Maps (ADT)

Presentation by *Asem Alaa*

Section 8 Parts:

1. C++ Operator Overloading & Lambdas
   - {`oo.cpp`}{`lambda.cpp`}
2. Binary Search Trees (BST)
   - {`BST.hpp`}{`bst.cpp`}
3. Sets (ADT) & Maps (ADT)
   - {`Set.hpp`}{`set.cpp`}{`Map.hpp`}{`map.cpp`}

Demo:

```
wget -i https://raw.githubusercontent.com/sbme-tutorials/sbme-tutorials.github.io/master/2020/data-structures/snippets/section08/BST/files.txt
```

C++: Operator Overloading (Example 1)

```
struct Vec2
{
    double x;
    double y;
};

int main()
{
    Vec2 u{1,1};
    Vec2 v{4,6};
    Vec2 d = u - v;
    // Compiler Err: error: no match for
    // ‘operator-’ (operand types are ‘Vec2’ and ‘Vec2’)
}
```

C++: Operator Overloading (Example 1)

```cpp
struct Vec2
{
    double x;
    double y;
    Vec2 operator-(const Vec2 &rhs) const
    {
        return Vec2{ x - rhs.x , y - rhs.y };
    }
};

int main()
{
    Vec2 u{1,1};
    Vec2 v{4,6};
    Vec2 d = u - v;
    // Compiler Happy
}
```

C++: Operator Overloading (Example 1)

```
struct Vec2
{
    double x;
    double y;
};
int main()
{
    Vec2 u{1,1};
    Vec2 v{4,6};
    Vec2 d = -(u+v)*(u-v) / (v*u*2);
    std::cout << d;
}
```

C++: Operator Overloading (Example 1)

```
struct Vec2
{
    double x;
    double y;
    Vec2 operator-(const Vec2 &rhs) const{...}
    Vec2 operator-() const {...}
    Vec2 operator+(const Vec2 &rhs) const{...}
    Vec2 operator*(const Vec2 &rhs) const{...}
    Vec2 operator*(double val) const{...}
    Vec2 operator/(const Vec2 &rhs) const{...}

    friend std::ostream &operator<<( std::ostream &output, const Vec2 &v ) {
        output << "(" << v.x << "," << v.y << ")";
        return output;
    }
};
int main()
{
    Vec2 u{1,1};
    Vec2 v{4,6};
    Vec2 d = -(u+v)*(u-v) / (v*u*2);
    std::cout << d; // prints: (1.875,2.91667)
}
```

C++: Operator Overloading (Example 2)

```
struct Image
{
    double *data;
    size_t width;
    size_t height;
    Image( size_t w, size_t h )
    {
        width = w;
        height = h;
        data = new double[ w * h ];
    }
};

int main()
{
    Image img = Image(16, 16); // 16x16 image

    // How to get pixel at position (9,12)?
    double val1 = img.data[ 9 + 12 * img.width ];
}
```

C++: Operator Overloading (Example 2)

```
struct Image
{
    double *data;
    size_t width;
    size_t height;
    Image( size_t w, size_t h ){
        width = w;
        height = h;
        data = new double[ w * h ];
    }

    double &operator()(size_t x, size_t y){
        return data[ x + y * width ];
    }
};
int main()
{
    Image img = Image(16, 16); // 16x16 image
    // How to get pixel at position (9,12)?
    double val1 = img.data[ 9 + 12 * img.width ];
    // Or
    double val2 = img(9, 12);
    // Modify
    img(9, 9) = 1.0;
}
```

C++: Lambda Expressions (Example 1)

Introduced in C++11

Syntax: `[` captures `](` params `){` body `}`

Simplest lambda: `auto l = [](){};`

```
int main()
{
    auto sq = [](double a){ return a*a; };

    std::vector< double > u = { 1.0, -2.0, 3.0, -4.0 };
    for( auto &x : u )
        x = sq( x );

    for( auto x: u ) std::cout << x << " ";
    std::cout << "\n"; // prints: 1.0 4.0 9.0 16.0
}
```

C++: Lambda Expressions (Example 1)

```
int main(){
    // Random number generators
    std::uniform_int_distribution<int> udist(0,100);
    std::mt19937 sampler;

    std::vector< int > v;
    for( int i = 0; i < 10 ; ++i) v.push_back(udist(sampler));

    for( auto x: v ) std::cout << x << " ";
    // Prints: 82 13 91 84 12 97 92 22 63 31
    std::cout << "\n";

    std::sort( v.begin(), v.end());

    for( auto x: v ) std::cout << x << " ";
    // Prints: 12 13 22 31 63 82 84 91 92 97
}
```

C++: Lambda Expressions (Example 2)

```cpp
int main(){
  // Random number generators
  std::uniform_int_distribution<int>udist(0,100); std::mt19937 sampler;

  std::vector< int > v;
  for( int i = 0; i < 10 ; ++i) v.push_back(udist(sampler));

  for( auto x: v ) std::cout << x << " ";
  // Prints: 82 13 91 84 12 97 92 22 63 31
  std::cout << "\n";

  std::sort( v.begin(), v.end());

  for( auto x: v ) std::cout << x << " ";
  // Prints: 12 13 22 31 63 82 84 91 92 97
  std::cout << "\n";

  std::sort( v.begin(), v.end(), [](int a, int b){return a > b;});
  for( auto x: v ) std::cout << x << " ";
  // Prints: 97 92 91 84 82 63 31 22 13 12
}
```

C++: Lambda Expressions (Example 2)

```
struct Student
{
    std::string name;
    int grade;
    std::string toString() const {
        std::stringstream s;
        s << "(" << name << "," << grade << ")";
        return s.str(); // returns: (name,grade)
    }
};
int main(){
    std::vector< Student > students({ {"Mahdy", 86}, {"Ahmed", 70},
                                      {"Samar", 86}, {"Zyad",  70}});
    std::sort( students.begin(), students.end(), []( Student &a, Student &b ){
        return a.grade > b.grade;
    });
    for( auto &stud : students ) std::cout << stud.toString() << " ";
    std::cout << "\n"; // out: (Mahdy,86) (Samar,86) (Ahmed,70) (Zyad,70)

    std::sort( students.begin(), students.end(),[]( Student &a, Student &b ){
        return a.name < b.name;
    });
    for( auto &stud : students ) std::cout << stud.toString() << " ";
    std::cout << "\n"; // out: (Ahmed,70) (Mahdy,86) (Samar,86) (Zyad,70)
}
```

Comparing `std::string`s

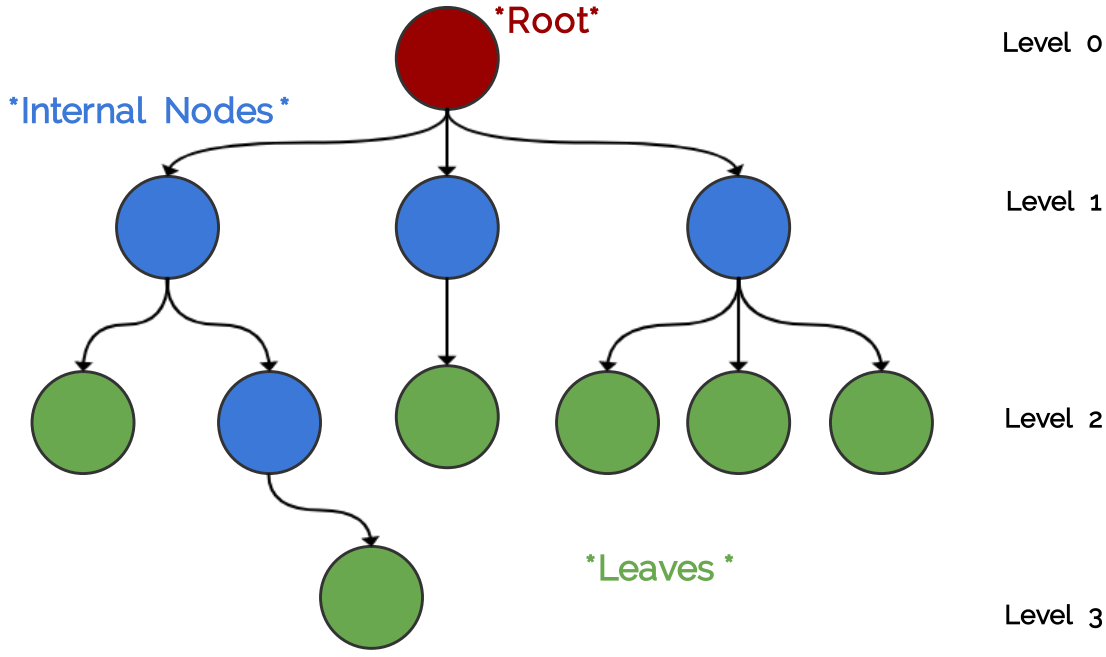
```
#include <string>
int main()
{
    std::string s1 = "batman";
    std::string s2 = "superman";

    int comparison = s1.compare( s2 );
}
```

- `comparison > 0`: `s1` comes after `s2` alphabetically.
- `comparison < 0`: `s1` precedes `s2` alphabetically.
- `comparison == 0`: `s1` equals `s2`

Comparing operators `std::string`s

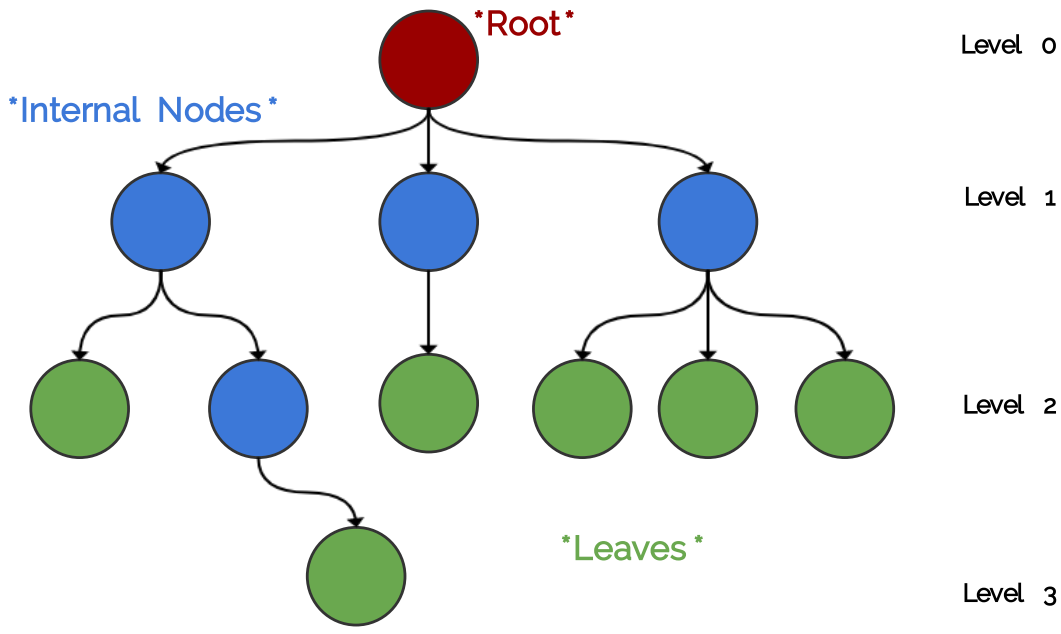
```
#include <string>
#include <iostream>
int main()
{
    std::string s1 = "batman";
    std::string s2 = "superman";

    if( s1 < s2 )
        std::cout << s1 << " precedes " << s2;
    else
        std::cout << s2 << " precedes " << s1;
}
```

Tree



- **Root:** is the top node.
- **Child:** any node that is emerged from an upper node.
- **Parent/Internal Node:** node with at least one child.
- **Siblings:** nodes sharing the same parent.
- **Leaf:** node with no children.
- **Edge:** the link between two nodes.
- **Path:** the sequence of links and nodes to reach from one node to a descedant.
- **Height of node:** the number of links between a node and the furthest leaf.
- **Depth of node:** the number of links between a node and the root.

Tree

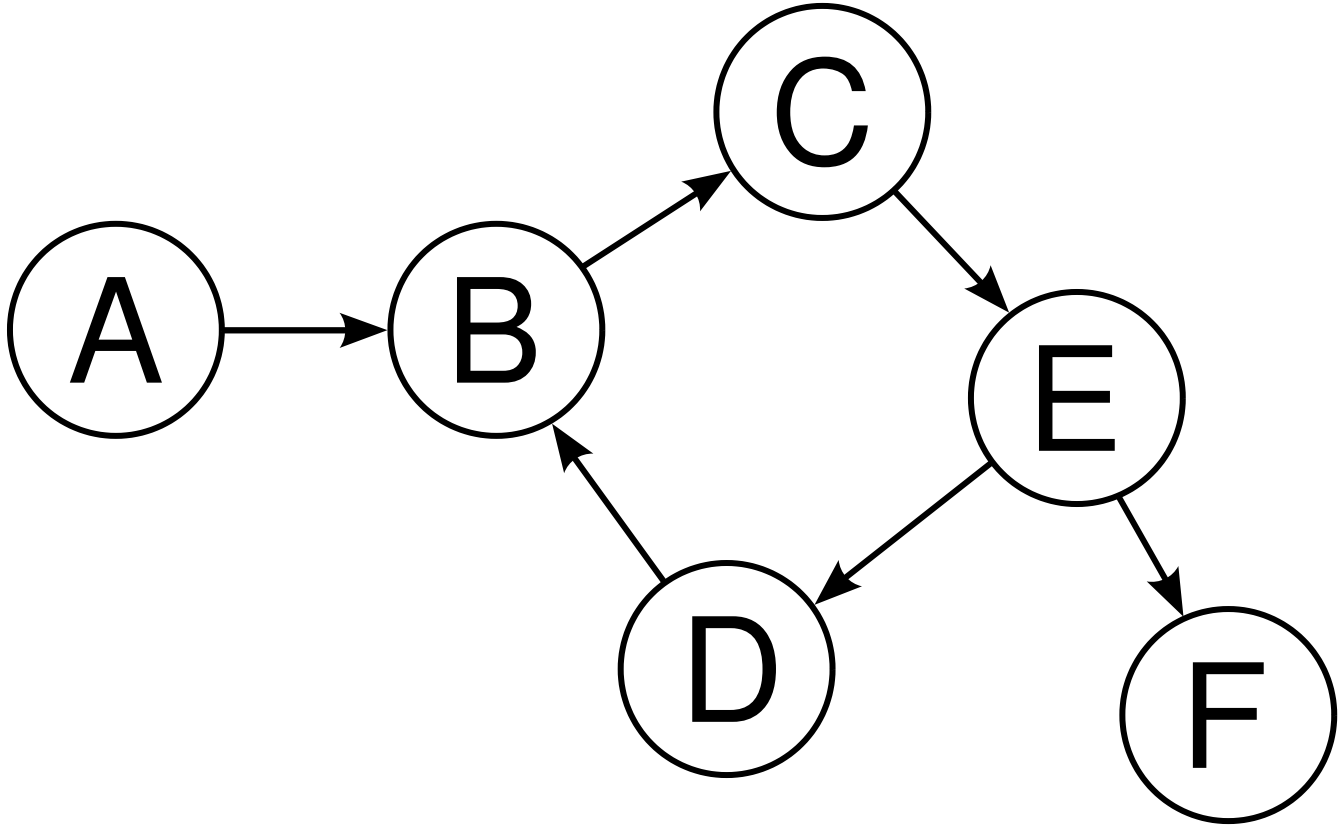


Synonyms

- Node = Vertex = Point
- Edge = Link = Arc

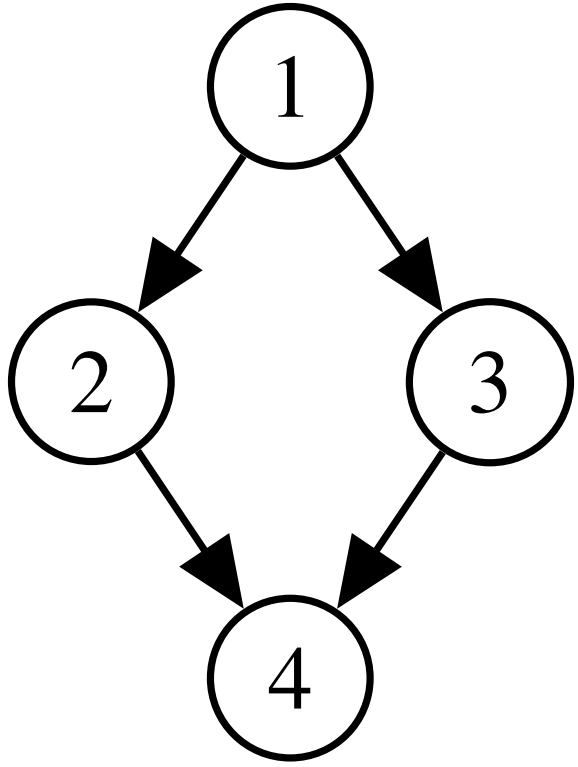
Violating Tree Structure

The following structure **is not tree**



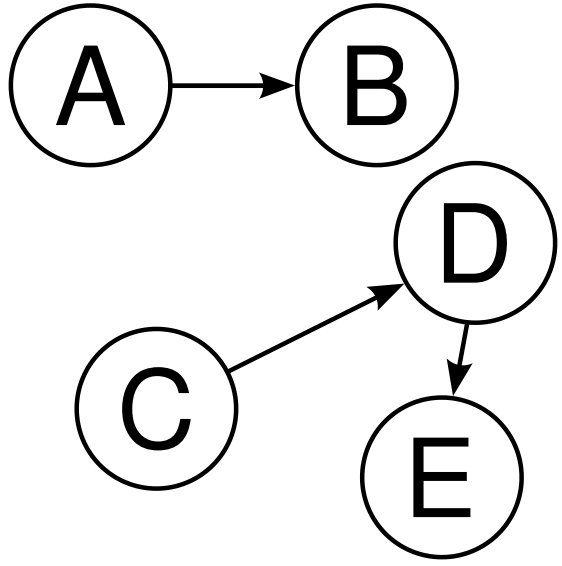
Violating Tree Structure

The following structures **is not tree**



Violating Tree Structure

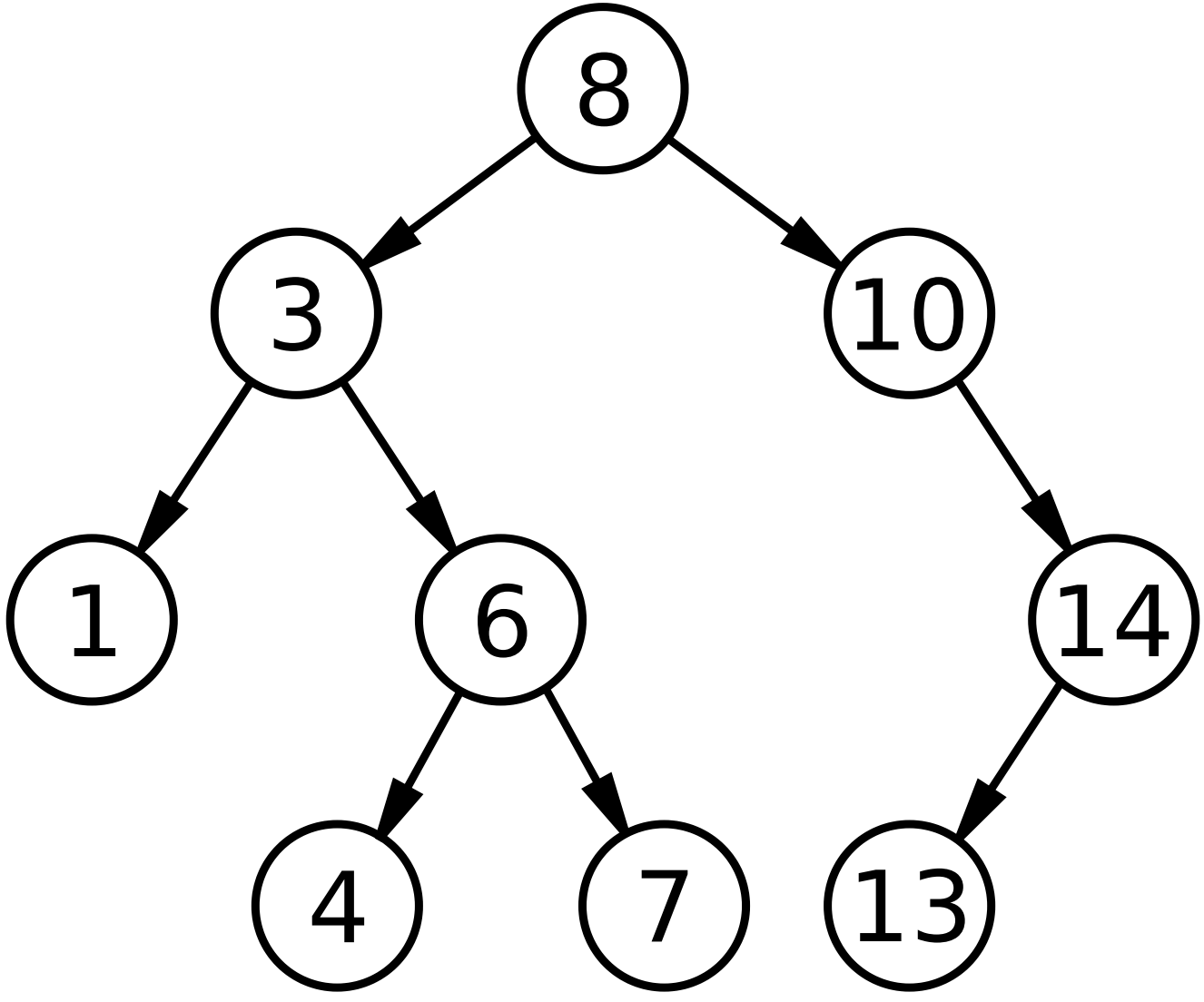
The following structures **is not tree**



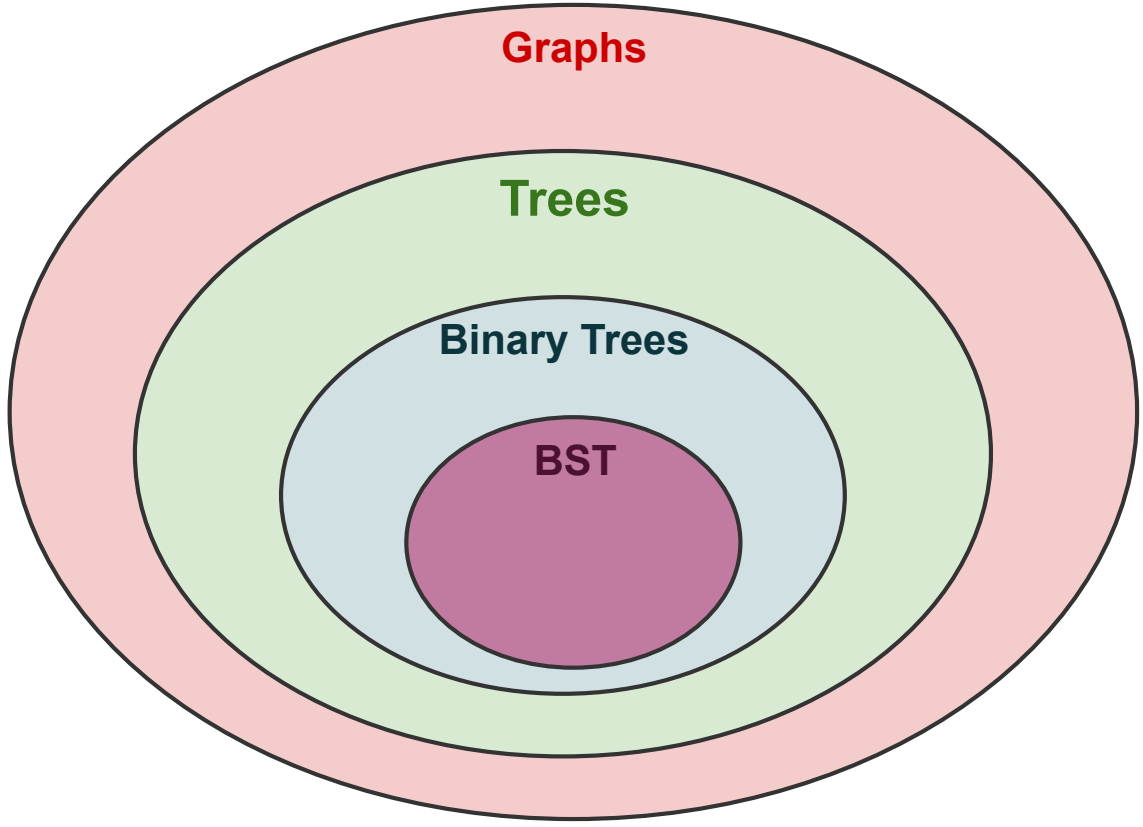
Binary Search Trees (BST)

- **Binary trees** is a special case of trees where each node can have at most 2 children.
- Also, these children are named: **left child** or **right child**.
- A very useful specialization of **binary trees** is **binary search tree (BST)**
- left children < parent < right children,
- and this rule applies recursively across the tree.

Binary Search Trees (BST)



Binary Search Trees (BST)



Binary Search Trees (BST)

Motivation

Efficient search/insertion/deletion in *logarithmic* time $O(\log(n))$

- Arrays:
  - efficient search on sorted arrays $O(\log(n))$,
  - ineffiecient insertion/deletion $O(n)$.
- Linked lists:
  - inefficient search $O(n)$,
  - efficient insertion/deletion $O(1)$.

Binary Search Trees (BST)

Intuition

- Tree combines the advantages of arrays and linked lists.
- Properties of **BST** (e.g being ordered) makes it potential for many applications.

Binary Search Trees (BST)

Implementation Using Linked Structures (Pointers)

- Trees can be stored in arrays (like Heaps) or stored as linked nodes (i.e using pointers).
- We will implement the **BST** using linked nodes.
- Recursion: Think of each node in a tree as a separate standalone tree.

Node structure

```
template< typename T >
class BST
{
    struct BSTNode
    {
        T data;
        BSTNode *left;
        BSTNode *right;
    };
};
```

Operations (isEmpty)

```
template< typename T >
class BST{ //...
    static bool isEmpty( const BSTNode *t )
    {
        return t == nullptr;
    }
};
```

Operations (isLeaf)

```
template< typename T >
class BST{ //...
    static bool isLeaf( const BSTNode *t )
    {
        return !isEmpty(t)
                && isEmpty( t->left )
                && isEmpty( t->right );
    }
};
```

Operations (size)

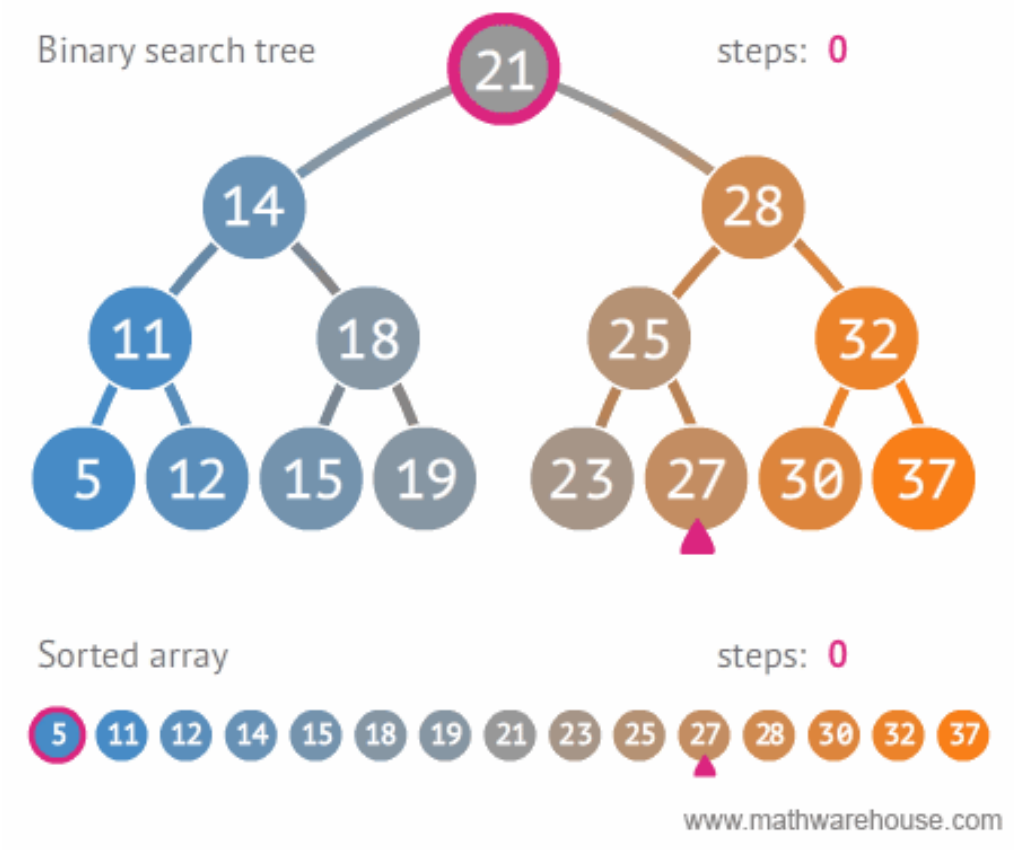
```
template< typename T >
class BST{ //...
    static size_t size( const BSTNode *t )
    {
        if ( !isEmpty( t ) )
            return 1 + size( t->left ) + size( t->right );
        else return 0;
    }
};
```

Operations (insert)

Operations (insert)

```
template< typename T >
class BST{ //...
    static BSTNode * insert( BSTNode *t, T data )
    {
        if ( isEmpty( t ))
            return new BSTNode{ data , nullptr , nullptr };
        else
        {
            if ( data < t->data )
                t->left = insert( t->left, data );
            else t->right = insert( t->right, data );
            return t;
        }
    }
};
```

Operations (find)

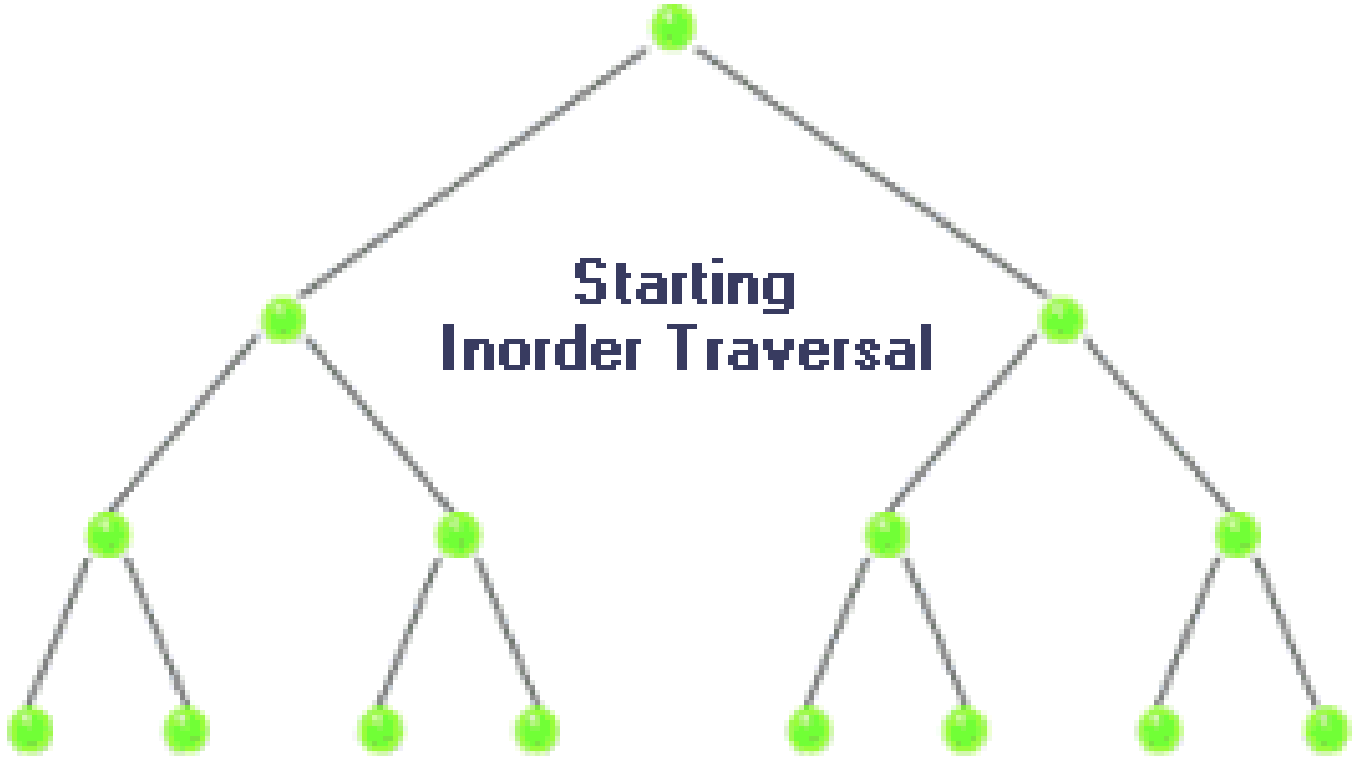


Operations (find)

```
template< typename T >
class BST{ //...
    static bool find( const BSTNode *t, T data )
    {
        if ( isEmpty( t ))
            return false;
        else
        {
            if ( data == t->data )
                return true;
            else if ( data < t->data )
                return find( t->left , data );
            else return find( t->right , data );
        }
    }
};
```

Traversal Operations

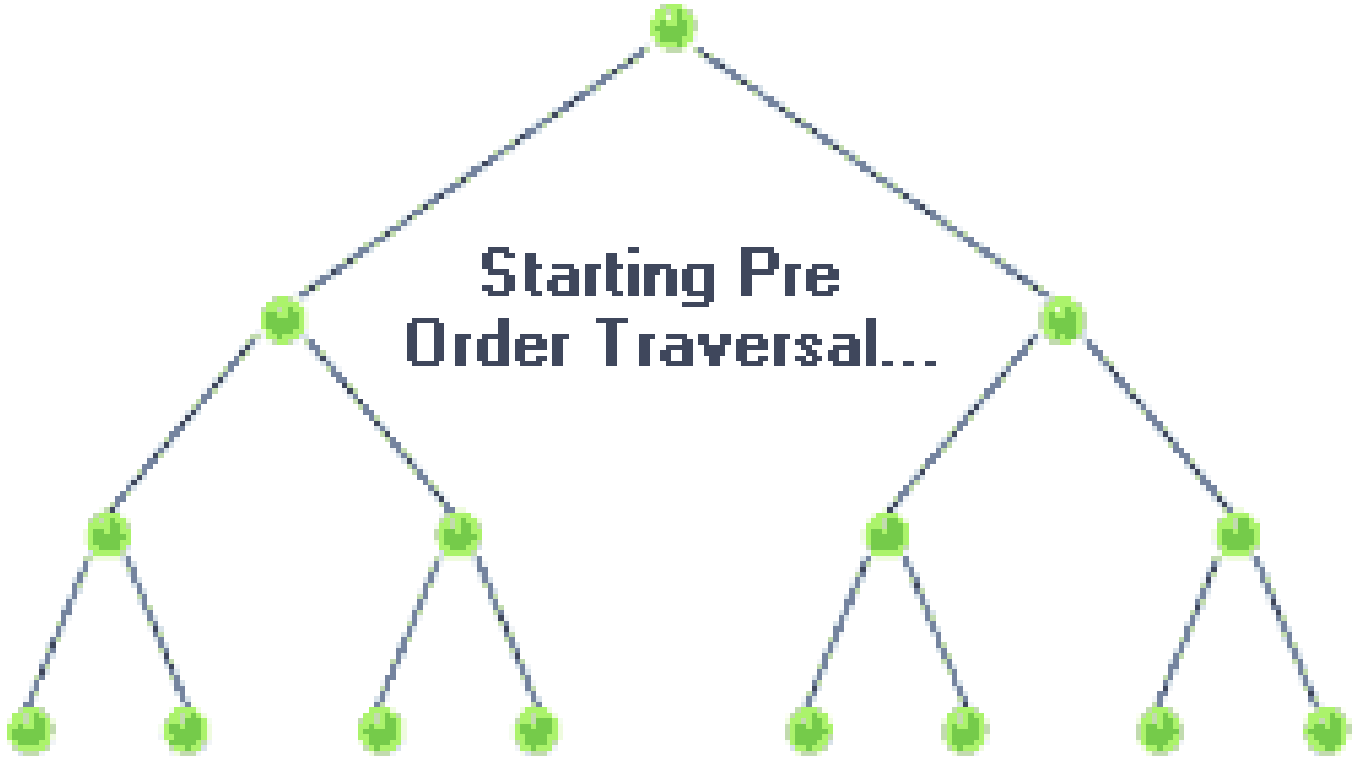
Traversal Operations: In-order



Traversal Operations: In-order

```
template< typename T >
class BST{ //...
    static void inorder( const BSTNode *t )
    {
        if( !isEmpty( t ))
        {
            inorder( t->left );
            std::cout << t->data << " ";
            inorder( t->right );
        }
    }
};
```

Traversal Operations: Pre-order



Traversal Operations: Pre-order

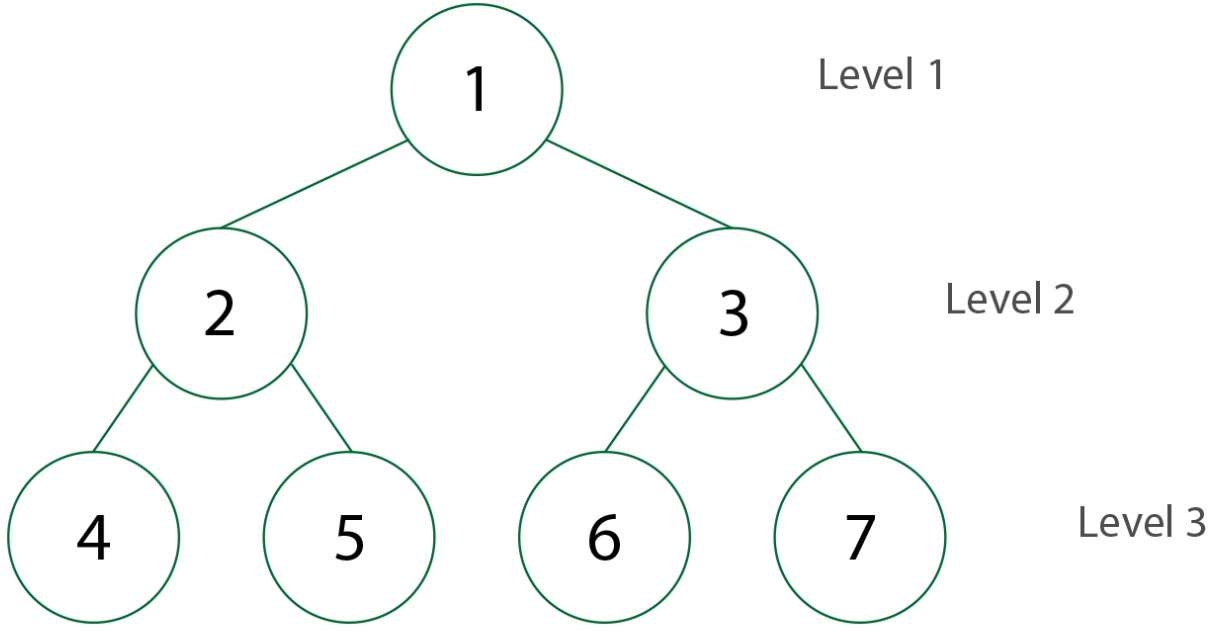
```
template< typename T >
class BST{ //...
    static void preorder( const BSTNode *t )
    {
        if( !isEmpty( t ))
        {
            std::cout << t->data << " ";
            preorder( t->left );
            preorder( t->right );
        }
    }
};
```

Traversal Operations: Post-order

Traversal Operations: Post-order

```
template< typename T >
class BST{ //...
    static void postorder( const BSTNode *t )
    {
        if( !isEmpty( t ))
        {
            postorder( t->left );
            postorder( t->right );
            std::cout << t->data << " ";
        }
    }
};
```

Traversal Operations: Breadth-first



Traversal Operations: Breadth-first

```
template< typename T >
class BST{ //...
    static void breadthfirst( const BSTNode *tree )
    {
        std::queue< const BSTNode * > q;
        q.push( tree );
        while( !q.empty())
        {
            auto t = q.front();
            q.pop();
            if( !isEmpty( t->left )) q.push( t->left );
            if( !isEmpty( t->right )) q.push( t->right );
            std::cout << t->data << " ";
        }
    }
};
```

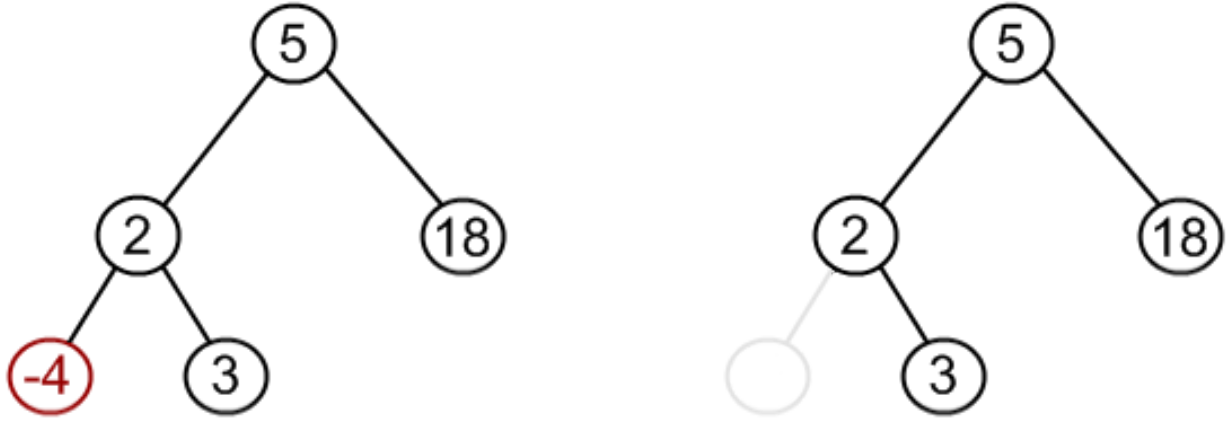
Operations (clear)

```
template< typename T >
class BST{ //...
    static void clear( BSTNode *t )
    {
        if ( !isEmpty( t )) {
            clear( t->left );
            clear( t->right );
            delete t;
        }
    }
};
```

Operations (remove)

Case I: Node to be removed **has no children**

Example: `remove( tree , -4 )`

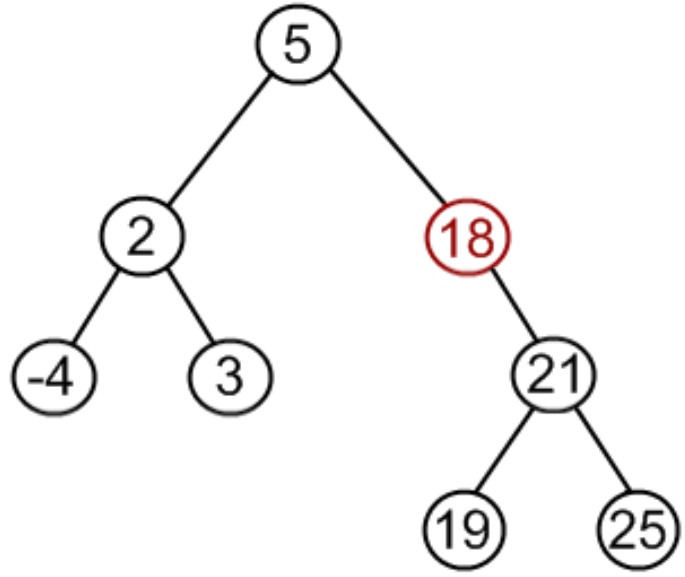


Operations (remove)

Case II: Node to be removed **has one child**

Example: `remove( tree , 18 )`

Case II: Node to be removed **has one child**

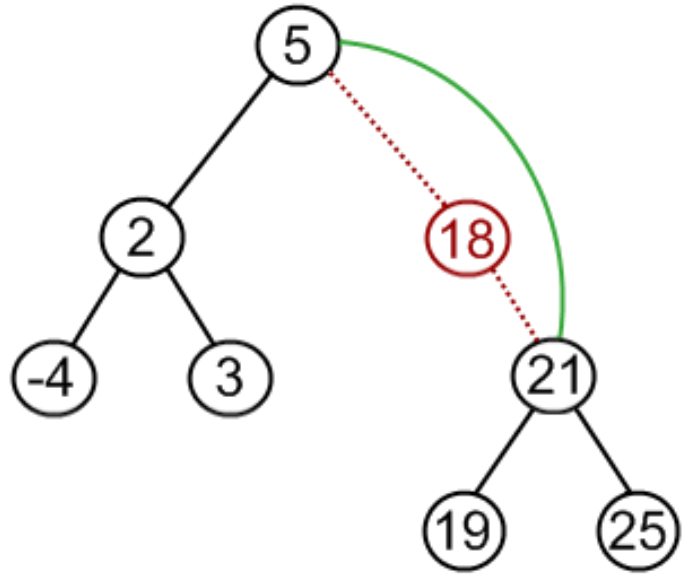


Operations (remove)

Case II: Node to be removed **has one child**

Example: `remove( tree , 18 )`

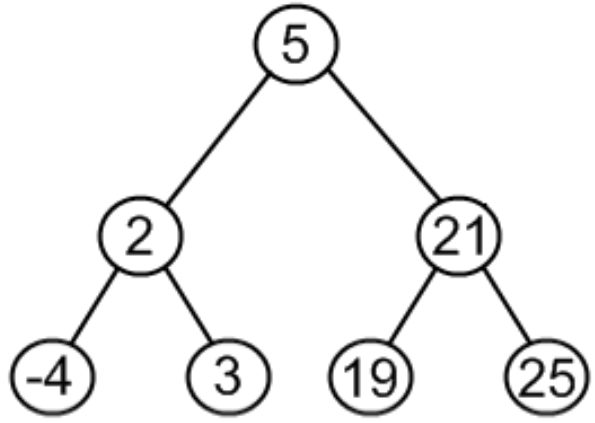
Case II: Node to be removed **has one child**



Operations (remove)

Case II: Node to be removed **has one child**

Example: `remove( tree , 18 )`

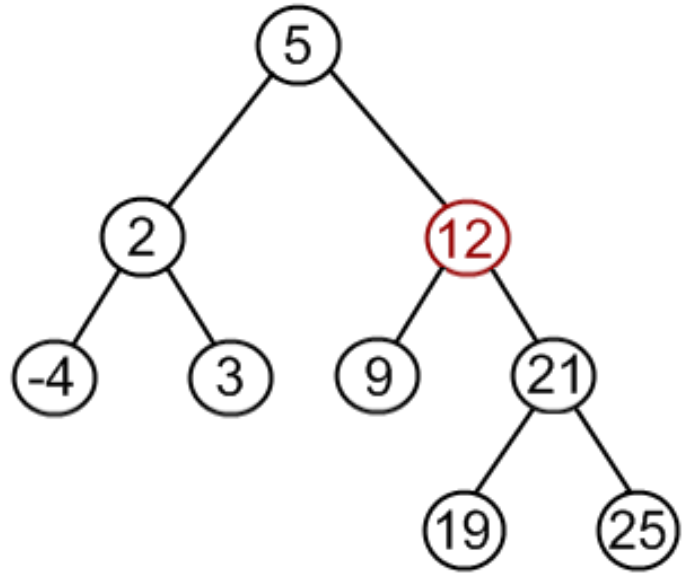


Operations (remove)

Case III: Node to be removed **has two children**

Example: `remove( tree , 18 )`

Case II: Node to be removed **has one child**

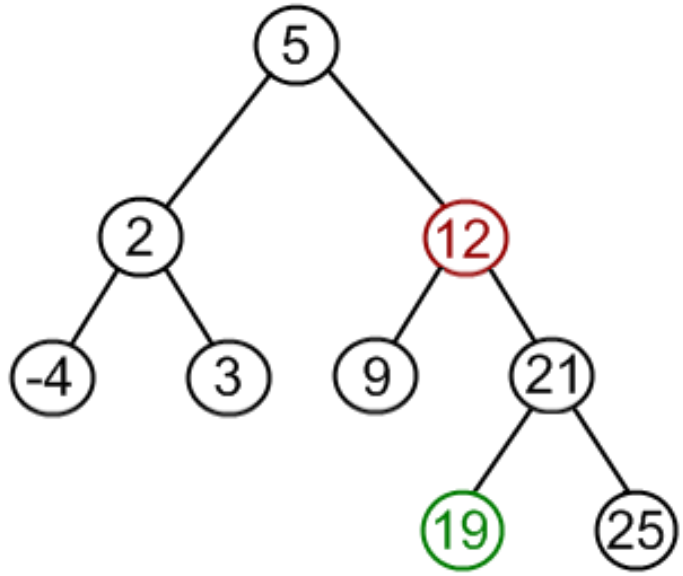


Operations (remove)

Case III: Node to be removed **has two children**

Example: `remove( tree , 18 )`

Case II: Node to be removed **has one child**

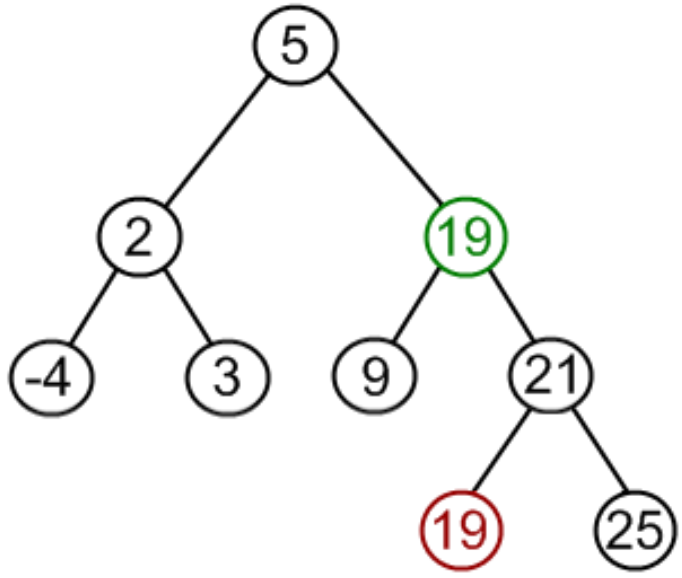


Operations (remove)

Case III: Node to be removed **has two children**

Example: `remove( tree , 18 )`

Case II: Node to be removed **has one child**

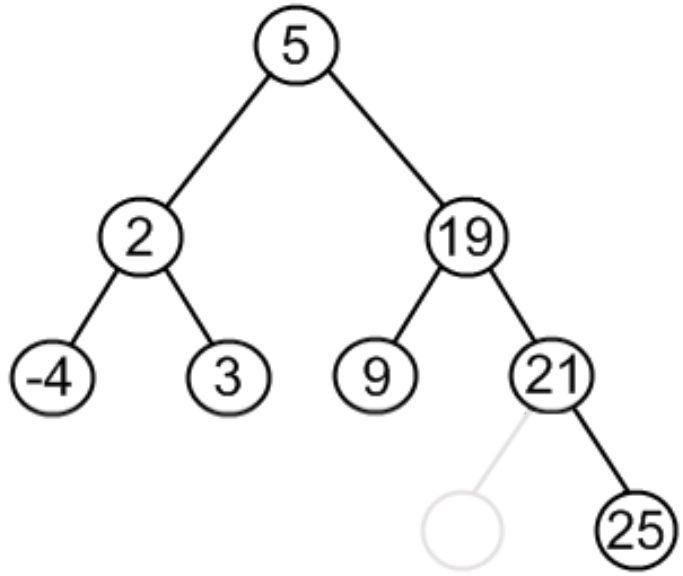


Operations (remove)

Case III: Node to be removed **has two children**

Example: `remove( tree , 18 )`

Case II: Node to be removed **has one child**



Operations (remove)

```
template< typename T >
class BST
{ //...
    static BSTNode * remove( BSTNode *t, T data )
    {
        if ( isEmpty( t )) return nullptr;
        if ( data == t->data ) t = removeNode( t );
        else if ( data < t->data ) t->left = remove( t->left, data );
        else t->right = remove( t->right, data );
        return t;
    }
};
```

Operations (remove)

```
template< typename T >
class BST{ //...
    static BSTNode * minNode( BSTNode *t ){
        auto min = t;
        while( min->left ) min = min->left;
        return min;
    }
    static BSTNode * removeNode( BSTNode *t ){
        if ( !isEmpty( t->left ) && !isEmpty( t->right )){
            BSTNode *minRight = minNode( t->right );
            t->data = minRight->data;
            t->right = remove( t->right, t->data );
        } else {
            BSTNode *discard = t;
            if ( isLeaf( t )) t = nullptr;
            else if ( !isEmpty( t->left )) t = t->left;
            else t = t->right;
            delete discard;
        }
        return t;
    }
};
```

Abstract Data Types (ADT) based on BST

Set

- **BST**: efficient insertions and removals.
- **modification**: in `insert` function, only insert unique values,

```
int main()
{
    std::mt19937 sampler; // random number sampler
    std::uniform_int_distribution<int> udist(0,100); // distribution
    std::set< int > s;
    for( int i = 0; i < 100 ; ++i)
        s.insert( udist(sampler) );
    for( auto x : s2 )
        std::cout << x << " ";
}
```

Set: Add

- Slight modification of `BST::insert`,
- Insertion is done only when the key doesn't exist.

Set: Add

- Slight modification of `BST::insert`,
- Insertion is done only when the key doesn't exist.

BST Insertion

```
template< typename T >
class BST{ //...
    static BSTNode * insert( BSTNode *t, T data ){
        if ( isEmpty( t ))
            return new BSTNode{ data , nullptr , nullptr };
        else
        {
            if ( data < t->data )
                t->left = insert( t->left, data );
            else t->right = insert( t->right, data );
            return t;
        }
    }
};
```

Set Insertion (solution 1)

```
template< typename T >
class Set
{ //...
    static SetNode * insert( SetNode *t, T data ){
        if ( isEmpty( t ))
            return new SetNode{ data , nullptr , nullptr };
        else if ( data != t->data )
        {
            if ( data < t->data )
                t->left = insert( t->left, data );
            else t->right = insert( t->right, data );
        }
        return t;
    }
};
```

Set: Insertion (solution 2)

1. use `find` to check if the element doesn't already exist,
2. if so, use `insert`.

```
template< typename T >
class Set{ //...
public:
    void add( T data )
    {
        if ( !find( data ))
            insert( data );
    }
};
```

Excercises: Set Union

- `union`: given two sets S_1 and S_2 make a new data structure $S_3 = S_1 \cup S_2$

possible implementation:

1. make an empty set `S3`,
2. iterate over elements of `S1` inserting each element to `S3`, and similarly for `S2`.

Excercises: Set Intersection

- `intersect`: given two sets S_1 and S_2 make a new data structure $S_3 = S_1 \cap S_2$

possible implementation:

1. make an empty set `S3`,
2. iterate over elements of `S1` inserting each element that also exists in `S2` into `S3`.

Excercises: Set Equality

- `equals`: given two sets S_1 and S_2, check the equality of the two sets,

possible implementation:

1. first, check that S_1 and S_2 sizes are equal,
2. then what?

Map

Synonyms: Associative containers, dictionary, symbol table.

A **map** is a collection of searchable key-value pairs, where each key has a value.

Map: Example Application 1

We can have a **map** (aka **dictionary**) to represent the count of words in a page or textbook, such that:

1. the **key** here is the *word* (`std::string`)
2. the **value** is the count of this word (`int`).

Map: Example Application 2

for the function that counts characters in **DNA**:

```
int countCharacter( std::string dna, char query ){
    int count = 0;
    for ( int i = 0; i < dna.size(); ++i)
        if ( query == dna[i] ) ++count;
    return count;
}
int main( int argc, char **argv ){
    std::string dna = readStream();
    int countA = countCharacter( dna , 'A');
    int countC = countCharacter( dna , 'C');
    int countG = countCharacter( dna , 'G');
    int countT = countCharacter( dna , 'T');
}
```

Map: Example Application 2

Map Elegant solution

- `countCharacter` was called four times (i.e to count **A**, **C**, **G**, and **T**).
- However, by using **map** data structure we can run this function to count all characters in a single run!

```
#include <map>
int main( int argc, char **argv ){
    std::string dna = readStream();
    std::map< char, int > dnaCounter;
    for( int i = 0 ; i < dna.size() ; ++i )
        dnaCounter[ dna[i] ]++;
    return 0;
}
```

Implementing a Dictionary (i.e Map) Using BST

Map implementation using **BST** would be as easy as implementing a **set**.

Dictionary Node Structure

```
template< typename K, typename V >
class Map
{
    struct MapNode
    {
        K key;
        V value;
        MapNode *left;
        MapNode *right;
    };
};
```

Implementing a Dictionary (i.e Map) Using BST

```
template< typename K, typename V >
class Map{ //...
    static bool isEmpty( const MapNode *m ){ return m == nullptr; }
    static bool isLeaf( const MapNode *m ){
        return !isEmpty(m) && isEmpty(m->left) && isEmpty(m->right);
    }
    static size_t size( const MapNode *m ){
        if ( !isEmpty( m ))
            return 1 + size( m->left ) + size( m->right );
        else return 0;
    }
    static void clear( MapNode *m ){
        if ( !isEmpty( m )){
            clear( m->left );
            clear( m->right );
            delete m;
        }
    }
};
```

Implementing a Dictionary (i.e Map) Using BST

```
template< typename K, typename V >
class Map { //...
    static bool find( const MapNode *m, K key )
    {
        // Same as BST
    }

    static MapNode * insert( MapNode *m, K key, V value ){
        if ( isEmpty( m ))
            return new MapNode{ key, value , nullptr , nullptr };
        else if ( key != m->key ){
            if ( key < m->key )
                m->left = insert( m->left, key , value );
            else m->right = insert( m->right, key , value );
        }
        return m;
    }
};
```

Implementing a Dictionary (i.e Map) Using BST

```
template< typename K, typename V > class Map{ //...
    static MapNode * remove( MapNode *m, K data )
    {
        // Same as BST
    }
    static MapNode * minNode( MapNode *m )
    {
        // Same as BST
    }

    static MapNode * removeNode( MapNode *m )
    {
        // Same as BST
    }
};
```

Implementing a Dictionary (i.e Map) Using BST

```
template< typename K, typename V >
class Map { //...
    static V &at( MapNode *m , K key ){
        if ( isEmpty( m )){
            std::cout << "Key not found!\n";
            exit( 1 );
        }
        else {
            if ( key == m->key ) return m->value;
            else if ( key < m->key ) return at( m->left , key );
            else return at( m->right , key );
        }
    }
};
```

Implementing a Dictionary (i.e Map) Using BST

```
template< typename K, typename V >
class Map{ //...
    template< typename Function >
    static void forEach( MapNode *m , Function fn )
    {
        if( !isEmpty( m ))
        {
            forEach( m->left, fn );
            fn( m->key , m->value );
            forEach( m->right, fn );
        }
    }
};
```

Implementing a Dictionary (i.e Map) Using BST

```
template< typename K, typename V >
class Map{ //...
private:
    MapNode *root = nullptr;
public:
    bool isEmpty() const  { return isEmpty( root );}
    size_t size() const  { return size( root ); }

    void insert( K key, V value )
    { root = insert( root , key , value );}

    bool find( K key ) const{ return find( root , key );}

    void clear()
    {
        clear( root );
        root = nullptr;
    }
    void remove( K key ){ root = remove( root , key );}
};
```

Implementing a Dictionary (i.e Map) Using BST

```
template< typename K, typename V >
class Map{ //...
    template< typename Function >
    void forEach( Function fn ) const
    { forEach( root , fn ); }

    void print () const
    {
        forEach( []( K k, V v ){
           std::cout << k << ":" << v << "\n";
        });
    }
};
```

Implementing a Dictionary (i.e Map) Using BST

```
template< typename K, typename V >
class Map{ //...
    V &at( const K &k )
    {
        return at( root, k );
    }

    V &operator[]( const K &k )
    {
        if( !find( k ))
            insert( k , V() );
        return at( k );
    }
};
```

Exercise: Word Count

Consider the following text for Carl Sagan

Look again at that dot. That's here. That's home. That's us. On it everyone you love, everyone you know, everyone you ever heard of, every human being who ever was, lived out their lives. The aggregate of our joy and suffering, thousands of confident religions, ideologies, and economic doctrines, every hunter and forager, every hero and coward, every creator and destroyer of civilization, every king and peasant, every young couple in love, every mother and father, hopeful child, inventor and explorer, every teacher of morals, every corrupt politician, every "superstar," every "supreme leader," every saint and sinner in the history of our species lived there on a mote of dust suspended in a sunbeam. The Earth is a very small stage in a vast cosmic arena. Think of the rivers of blood spilled by all those generals and emperors so that, in glory and triumph, they could become the momentary masters of a fraction of a dot. Think of the endless cruelties visited by the inhabitants of one corner of this pixel on the scarcely distinguishable inhabitants of some other corner, how frequent their misunderstandings, how eager they are to kill one another, how fervent their hatreds. Our posturings, our imagined self-importance, the delusion that we have some privileged position in the Universe, are challenged by this point of pale light. Our planet is a lonely speck in the great enveloping cosmic dark. In our obscurity, in all this vastness, there is no hint that help will come from elsewhere to save us from ourselves. The Earth is the only world known so far to harbor life. There is nowhere else, at least in the near future, to which our species could migrate. Visit, yes. Settle, not yet. Like it or not, for the moment the Earth is where we make our stand. It has been said that astronomy is a humbling and character-building experience. There is perhaps no better demonstration of the folly of human conceits than this distant image of our tiny world. To me, it underscores our responsibility to deal more kindly with one another, and to preserve and cherish the pale blue dot, the only home we've ever known.

Prelimenary Statistics

Total count of words	count of words after removing duplicates (i.e word set)
362	205