Binary Search Trees

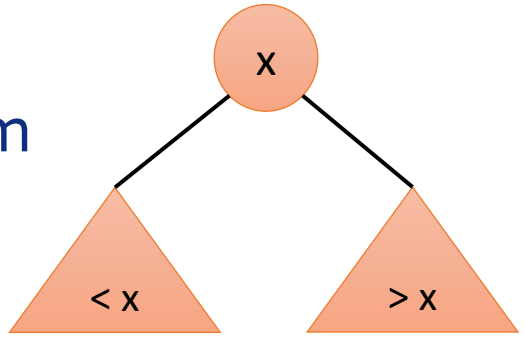
CSE 2020 Computer Science II

Learning Objectives

- define binary search tree ADT
- describe the process of inserting elements to a binary search tree, deleting elements from a binary search tree, and searching elements in a binary search tree.
- design, implement and analyze insertion, deletion, and search operations of binary search trees

Binary Search Tree ADT

- A binary search tree is a binary tree, in which
  1. each node contains distinct value
  2. for any node,
     - the node values in its left subtree are less than the node value
     - the node values in its right subtree are greater than the node value
- Inorder traversal sequence is in ascending order
- The most left node is the minimum
- The most right node is the maximum



Operations

- const C& **findMin**() const: returns the smallest
- const C& **findMax**() const: returns the greatest
- bool **contains**(const C& x) const:

  returns true if x is in this tree
- bool **isEmpty**() const:

  returns true if the tree is empty
- void **insert**(const C& x): inserts x
- void **remove**(const C& x): removes x
- void **makeEmpty**(): make the tree to empty state
- void **printBST**() const: print the elements in inorder seq

BST Example 1

- Create a BST from a sequence of data values. Insertion order:

  5, 9, 7, 3, 8, 11, 6, 4, 12, 2, 1

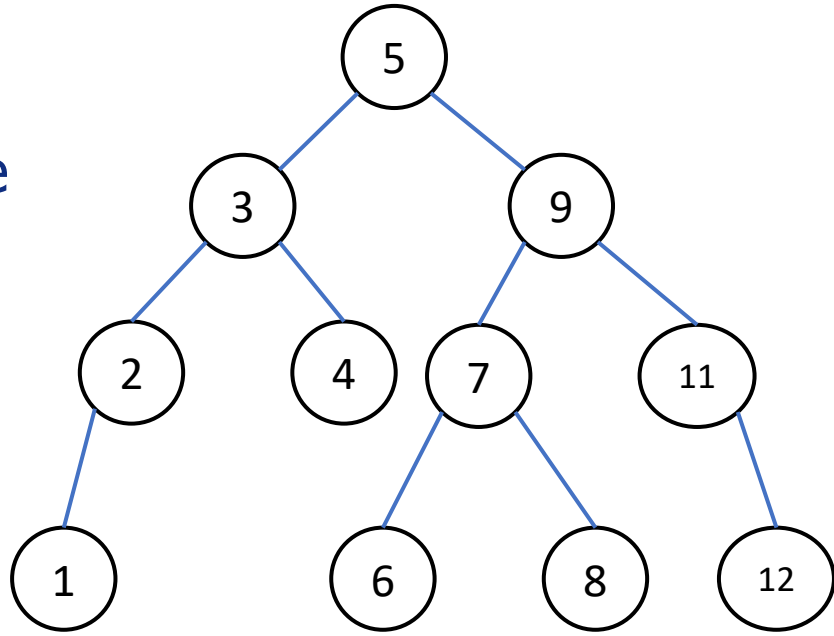
- Insertion order determines the shape of BST

  for example, 1, 2, 3 and 2, 1, 3

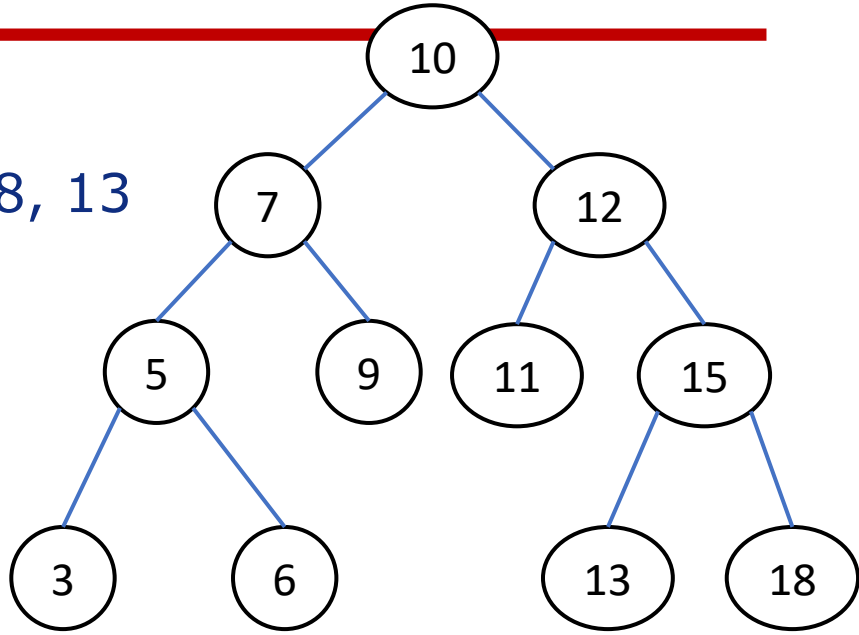
- printBST

  1, 2, 3, 4, 5, 6, 7, 8, 9, 11, 12

- min 1, max 12



BST Example 2



- Insertion order:

  10, 12, 15, 7, 5, 6, 9, 3, 11, 18, 13
- min 3
- max 18
- contain 8? no
- contain 18? yes
- printBST 3, 5, 6, 7, 9, 10, 11, 12, 13, 15, 18
- remove 10
  - use 11 replace 10, remove 11 from right subtree
  - **OR** use 9 replace 10, remove 9 from right subtree

Implementation of BST ADT

- Private attribute *root*, a pointer to the root of the binary search tree

```
template <typename C>
class BinarySearchTree {
private:
   struct BinaryNode {
        C element;  // element is of type C which is comparable
        BinaryNode* left;
        BinaryNode* right;
        BinaryNode(const C& theElement, BinaryNode* lt, BinaryNode* rt)
          : element( theElement ), left( lt ), right( rt ) { }
    };
    BinaryNode* root;
    ……
};
```

- In main(), BinarySearchTree<int> intbst; intbst.printBST();

Implementation

- Recursive definition of BST → recursive functions, for example, *printBST(root)* will recursively call itself to access nodes in inorder traversal. But root is a private attribute, it is not visible or accessible in main().
- Declare recursive functions as private
- The public member functions call private recursive member functions, such as public *void printBST() const* calls private *void printBST(root) const*.

Implementation - printBTS

- printBST()

```
public
   void printBST( ) const
   {  printBST( root );  }
private
   void printBST( BinaryNode* t) const;
    //inorder traversal
   if t is not empty
    {
         printBST(t->left);
         cout << t->element;
         printBST(t->right);
    }
```

Implementation – findMin

- findMin()

```
public:
   const C & findMin() const
   {  return findMin( root )->element; }
private:
    BinaryNode* findMin(BinaryNode* t) const;
```

  - if t is empty, return nullptr
  - if t->left is nullptr, return t
  - else find min in t->left

Implementation – findMax

- findMax()

```
public:
    const C & findMax() const
    {return findMax( root )->element; }
private:
    BinaryNode* findMax(BinaryNode* t) const;
```

  - if t is empty, return nullptr
  - if t->right is nullptr, return t
  - else find max in t->right

Implementation - contains

- contains()

```
public
   bool contains( const C & x ) const
   {return contains( x, root );}
private
   bool contains( const C & x, BinaryNode* t ) const;
```

- contains x ?
  - if t is empty, return false
  - else if x < t->element, search x in t->left
  - else if x > t->element, search x in t->right
  - else find the matched element, return true

Implementation - insert

- insert()

```
public
    void insert( const C & x )
    { insert( x, root ); }
private
    void insert( const C & x, BinaryNode* & t);
```

- insert x in t
  - proceed down the binary search tree until reach the last spot on the path traversed, then insert the new node as a leaf there.
  - if t is empty, insert the new node as a leaf
  - else if x <t->element, insert x in t->left
  - else if x > t->element, insert x in t->right
  - else find the matched element, do nothing

Implementation - remove

- remove()

```
public
    void remove( const C & x )
    { remove( x, root ); }
private
    void remove( const C & x, BinaryNode* & t );
```

  - if t is empty, not find matched, do nothing
  - else if x <t->element, remove x in t->left
  - else if x > t->element, remove x in t->right
  - else find the matched element, see next page

Implementation – remove (cont.)

- find x in the tree, the node containing x is
  - a leaf node – remove the node
  - a node has one child – bypass the node
  - a node has two children
    - replace the node element with the smallest value of its right subtree, remove the node with this smallest value recursively
    - **or** replace the node element with the largest value of its left subtree, remove the node with this largest value recursively

Implementation - makeEmpty

- makeEmpty()

```
public
    void makeEmpty( )
    {  makeEmpty( root ); }
private
    void makeEmpty( BinaryNode* & t );
    // postorder
    if t is not empty
    {
        make empty t->left
        make empty t->right
        delete t;
    }
```

Implementation files bst.cpp

- The implementation of private recursive functions
- The file bst.txt (contains bst.cpp and TestBST.cpp) is posted on Canvas

Time Complexity

- The time complexity of operations are determined by the height of a tree.
- If the input comes into a binary search tree presorted, the binary search tree becomes a linked list, contains, insertion and deletion are all $O(n)$
- A balanced binary search tree such as AVL tree (for every node, the height of the left and right subtrees can differ by at most 1), can ensure the height of the tree is O(logn), so **contains, insertion and deletion are all $O(logn)$**