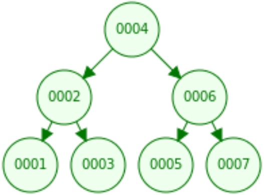
Binary Search Trees

L.EIC

Algorithms and Data Structures

2025/2026



P Ribeiro, AP Tomas

Binary Search Trees - Motivation

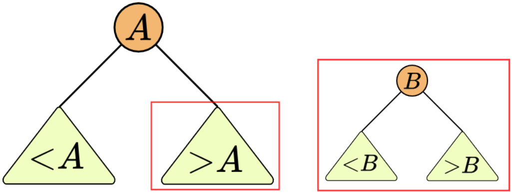
- Let S be a set of **"comparable"** objects/items:
  - Let a and b be two objects.
    They are "comparable" if it is possible to say whether $a < b$, $a = b$, or $a > b$.
  - An example could be numbers, but it could be something else (strins, students with a name and a student number; teams with points, goals scored and conceded, etc.)

- Some possible **problems** of interest:
  - Given a set S, determine if **a given item is in** S
  - Given a **dynamic set** S (which undergoes changes: additions and removals), determine if **a given item is in** S
  - Given a **dynamic set** S, determine the **largest/smallest** item in S
  - **Sort** a set S
  - ...

- **Binary Search Trees!**

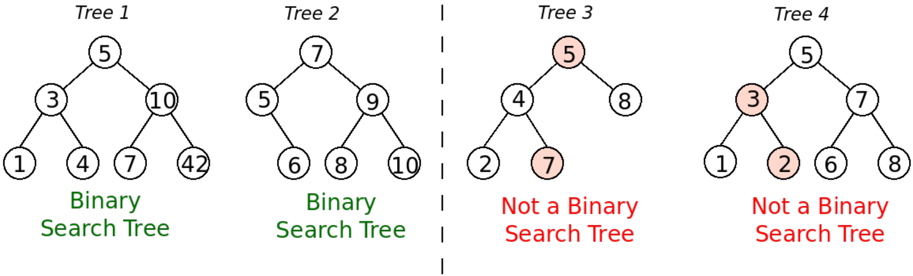
Binary Search Trees - Concept

- For **every** node in the tree, the following must happen:
  the node is greater than all the nodes in its left subtree and smaller than all the nodes in its right subtree



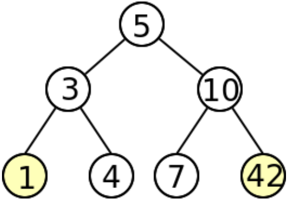
Binary Search Trees - Examples

- For **every** node in the tree, the following must happen: **the node is greater than all the nodes in its left subtree and smaller than all the nodes in its right subtree**



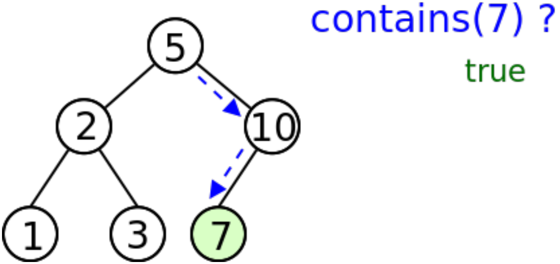
- In trees 1 and 2, the conditions are respected
- In tree 3, node 7 is on the left of node 5, but $7 > 5$
- In tree 4, node 2 is on the right of node 3, but $2 < 3$

Binary Search Trees - Some Consequences



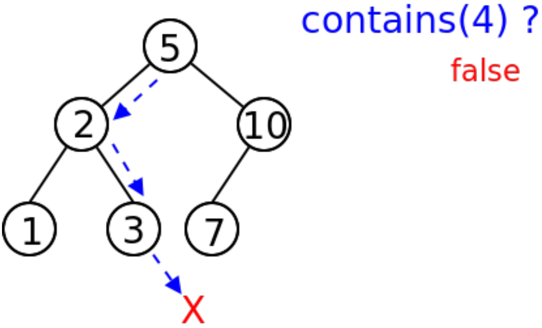
- The **smallest** element is... in the **leftmost node**
- The **largest** element is... in the **rightmost node**

Binary Search Trees - Searching for a Value



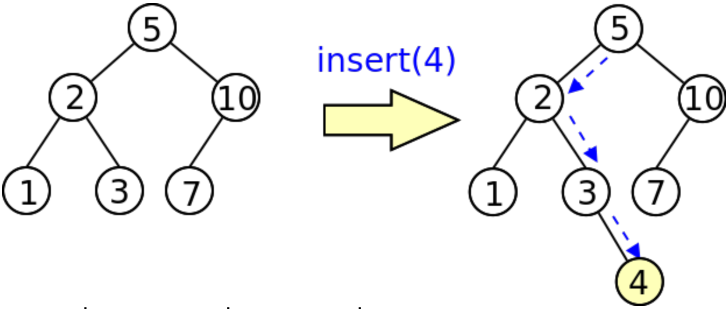
- Start at the root, and traverse the tree
- Choose the left or right branch based on whether the value is smaller or greater than the "current" node

Binary Search Trees - Searching for a Value



- Start at the root, and traverse the tree
- Choose the left or right branch based on whether the value is smaller or greater than the "current" node

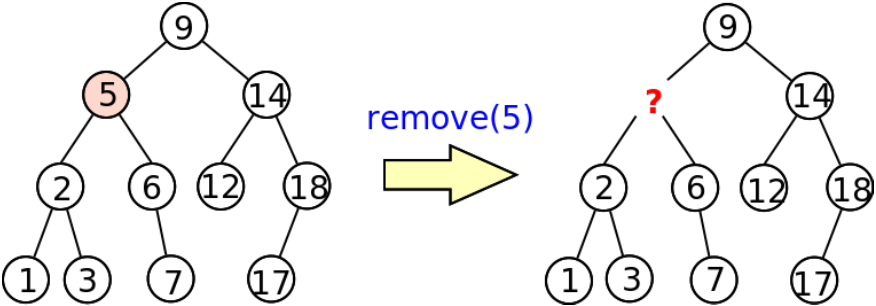
Binary Search Trees - Inserting a Value



- Start at the root, and traverse the tree
- Choose the left or right branch based on whether the value is smaller or greater than the "current" node
- Insert at the corresponding leaf position

Note: If the value is equal to an already existing one, it is typically *not inserted*. If we want to allow repeated values (a *multiset*), we must be consistent and always choose a position (e.g., always insert on the left, where left subtree nodes would be $\leq$ and right subtree nodes would be $>$)

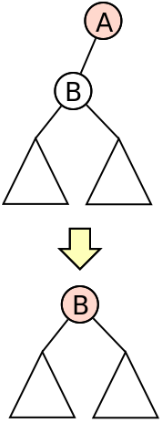
Binary Search Trees - Removing a Value

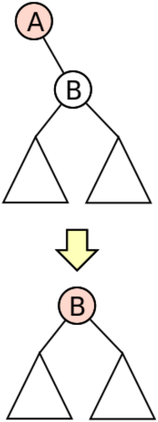


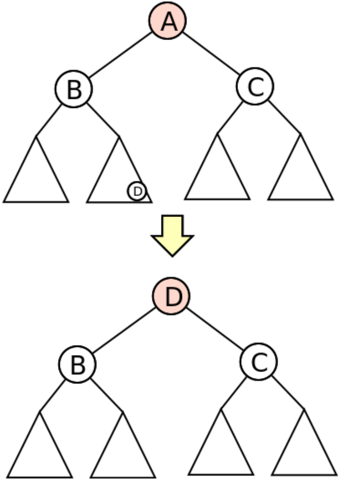
- Start at the root, and traverse the tree until the value is found
- Once the value is found, what should we do next?
  - If the node to be removed has only one child, simply "lift" that child to the corresponding position
  - If it has two children, the candidates to replace it are:
    - The largest node in the left branch, or
    - The smallest node in the right branch

Binary Search Trees - Removing a Value

- After finding the node, we need to decide **how to remove it**
  - 3 possible cases:

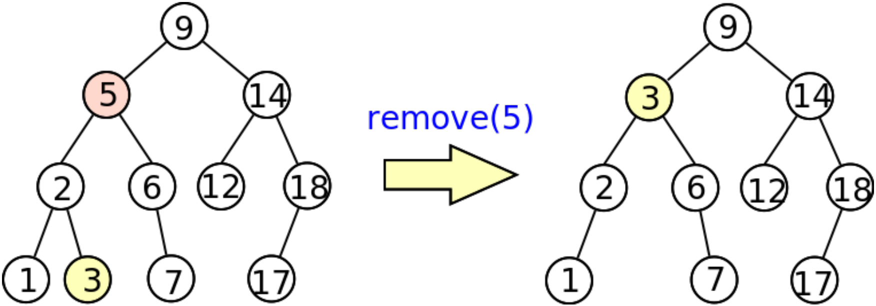






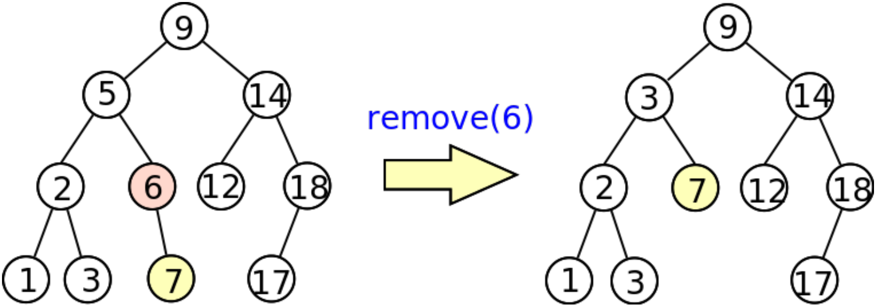
Binary Search Trees - Removing a Value

Example with two children



Binary Search Trees - Removing a Value

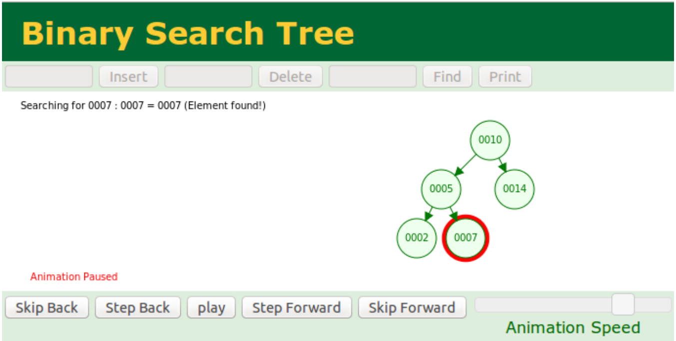
Example with only one child



Binary Search Trees - Visualization

- You can visualize search, insertion, and removal (try the provided URL):

  https://www.cs.usfca.edu/~galles/visualization/BST.html



Binary Search Trees - Complexity

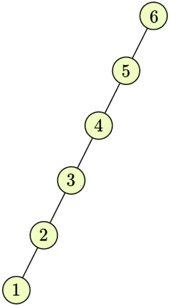
- How to characterize the **time each operation takes**?
  - All operations search for a node by traversing the **height** of the tree

Complexity of operations in a binary search tree

Let h be the height of a binary search tree T. The complexity of finding the minimum, maximum, or performing a search, insertion, or removal in T is $\mathcal{O}(h)$.

Imbalance in a Binary Search Tree

- The **problem** with the previous methods:

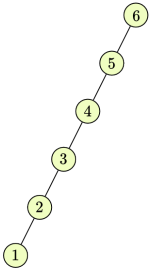


The height of the tree can be of the order $\mathcal{O}(n)$ (n, number of elements)

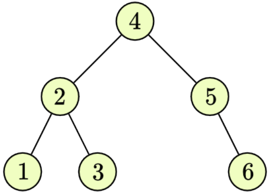
(The height depends on the order of insertion, and there are "bad" orders)

Balanced Trees

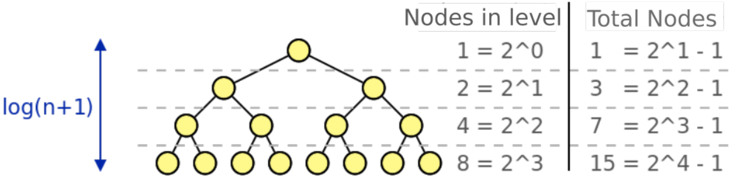
- We want trees... **balanced**



vs



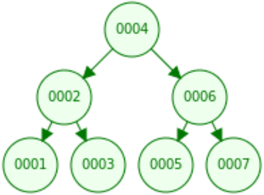
- In a balanced tree with n nodes, the height is ... $\mathcal{O}(\log n)$



Balanced Trees

For a given set of numbers, **which order to insert** in a binary search tree so that it becomes as balanced as possible?

Answer: *"binary search"* - if the numbers are sorted, insert the middle element, split the remaining list at that element, and insert the remaining elements from each half in the same manner.



Balancing Strategies

- What if we don't know all the elements at the beginning and we need to **dynamically** insert and remove elements?
- There are strategies to ensure that the complexity of operations like searching, inserting, and removing is better than $\mathcal{O}(n)$

Balanced trees:
(height $\mathcal{O}(\log n)$)

- AVL Trees (*)
- Red-Black Trees (*)
- Splay Trees
- B-Trees
- Treaps
- ...

Other data structures:

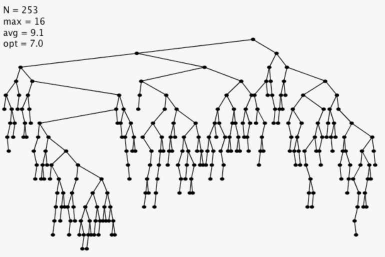
- Skip List
- Hash Table (*)
- Bloom Filter

- We will discuss some of these strategies at this course (the ones with *)

Height for Random Order

Height of a tree with random elements

If we insert n elements in a completely random order into a binary search tree, its *expected height* is $\mathcal{O}(\log n)$



- If you're curious about a proof, check out the excellent book *"Introduction to Algorithms"* (this is not required for the exam)
- For purely *random* data, the average height is $\mathcal{O}(\log n)$ as we insert and remove nodes

Binary Search Trees - An Implementation

- We are not going to analyze in detail during the lecture, but here is a possible implementation of an (unbalanced) binary search tree
- The core of the class is exactly the same as before.

```
template <class T> class BSTree {
private:
  struct Node {
    T value;
    Node *left, *right;
  };

  Node *root;

public:
  BSTree() {root = nullptr;}

  // ...
}
```

Binary Search Trees - Implementation

- What changes are the other methods, such as **contains**, **insert**, and **remove**, which can take advantage of the fact that it is a binary search tree.
- These are methods of the BSTree class. For a visual explanation of what they do, you can refer to previous slides (slides 38 to 45).
- Let's start with **contains**.

```cpp
// Is the value contained in the tree?
bool contains(const T & val) {
  return contains(root, val);
}

bool contains(Node *n, const T & val) {
  if (n == nullptr) return false;
  if (val < n->value) return contains(n->left, val);
  if (val > n->value) return contains(n->right, val);
  return true;
```

Binary Search Trees - Implementation

- For **insert**, we take advantage of the return value of a recursive function.

```
// Add an element to a search tree
// Returns true if insertion succeeded, false otherwise
bool insert(const T & val) {
  if (contains(val)) return false;
  root = insert(root, val);
  return true;
}

Node *insert(Node *n, const T & val) {
  if (n == nullptr) {
    Node *aux = new Node;
    aux->value = val;
    aux->left = aux->right = nullptr;
    return aux;
  } else if (val < n->value) {
    n->left = insert(n->left, val);
  } else if (val > n->value) {
    n->right = insert(n->right, val);
  }
  return n;
}
```

Binary Search Trees - Implementation

- **remove**: replace removed value with largest value from left subtree.

```
// Remove an element from a search tree
// Returns true if removal succeeded, false otherwise
bool remove(const T & val) {
  if (!contains(val)) return false;
  root = remove(root, val);
  return true;
}

Node *remove(Node *n, const T & val) {
  if (val < n->value) n->left = remove(n->left, val);
  if (val > n->value) n->right = remove(n->right, val);
  else if (n->left == nullptr) {
    Node * tmp = n->right; delete n; return tmp; // "garbage collection
  } else if (n->right == nullptr) {
    Node *tmp = n->left; delete n; return tmp;   // "garbage collection
  } else {
    Node *max = n->left;
    while (max->right != nullptr) max = max->right;
    n->value = max->value;
    n->left = remove(n->left, max->value);
  }
  return n;
}
```

Binary Search Trees - Test

- An example of usage (assuming the other methods were implemented as in normal Binary Trees):

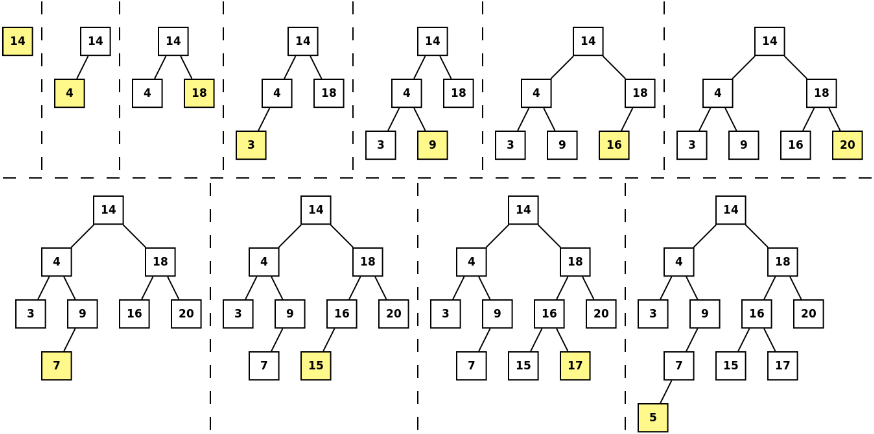
```
#include "binarySearchTree.h"

int main() {
  // Tree creation
  BSTree<int> t;
  // Inserting 11 elements in the binary search tree
  int data[] = {14, 4, 18, 3, 9, 16, 20, 7, 15, 17, 5};
  for (int i=0; i<11; i++) t.insert(data[i]);

  // Writing the result of calling some methods
  std::cout << "numberNodes = " << t.numberNodes() << std::endl;
  std::cout << "depth = " << t.depth() << std::endl;
  std::cout << "contains(2) = " << t.contains(2) << std::endl;
  std::cout << "contains(3) = " << t.contains(3) << std::endl;
  // Writing the nodes of the tree following several possible orders
  t.printPreOrder(); t.printInOrder(); t.printPostOrder();
  // Trying node removal
  t.remove(14);
  t.printPreOrder(); t.printInOrder(); t.printPostOrder();
}
```

Binary Search Trees (BST) - Test

- The code from the previous slide creates the following BST:



- Inorder, this tree becomes: `3 4 5 7 9 14 15 16 17 18 20`

When printed *inOrder*, the values of a BST are sorted in ascending order.

What's next?

- On the next chapter we will cover:
  - Balanced BSTs: **AVL Trees** and **Red-Black Trees**
  - BSTs in STL: **set** and **map** (and also **multiset** and **multimap**)
  - **Applications** of BSTs