Today …

Binary Trees

Binary Search Trees
  definition
  search
Binary Trees
Binary Trees

A k-ary tree where \( k = 2 \)
Linked Structure for Binary Trees

Every node has:

- data
- parent
- left child
- right child
## Computational Cost

<table>
<thead>
<tr>
<th>Operation</th>
<th>Unordered Sequence</th>
<th>Ordered Sequence</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>search</strong></td>
<td>search for a key</td>
<td></td>
</tr>
<tr>
<td><strong>insert</strong></td>
<td>insert a key</td>
<td></td>
</tr>
<tr>
<td><strong>delete</strong></td>
<td>delete a key</td>
<td></td>
</tr>
<tr>
<td><strong>min/max</strong></td>
<td>smallest/largest key</td>
<td></td>
</tr>
<tr>
<td><strong>floor/ceiling</strong></td>
<td>predecessor/successor</td>
<td></td>
</tr>
<tr>
<td><strong>rank</strong></td>
<td>number of keys less than key</td>
<td></td>
</tr>
</tbody>
</table>
## Computational Cost

<table>
<thead>
<tr>
<th></th>
<th>sequential search (unordered sequence)</th>
<th>binary search (ordered sequence)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>search</strong></td>
<td>search for a key</td>
<td>O(n)</td>
</tr>
<tr>
<td><strong>insert</strong></td>
<td>insert a key</td>
<td>O(n)</td>
</tr>
<tr>
<td><strong>delete</strong></td>
<td>delete a key</td>
<td>O(n)</td>
</tr>
<tr>
<td><strong>min/max</strong></td>
<td>smallest/largest key</td>
<td>O(n)</td>
</tr>
<tr>
<td><strong>floor/ceiling</strong></td>
<td>predecessor/successor</td>
<td>O(n)</td>
</tr>
<tr>
<td><strong>rank</strong></td>
<td>number of keys less than key</td>
<td>O(n)</td>
</tr>
</tbody>
</table>
# Computational Cost

<table>
<thead>
<tr>
<th>what?</th>
<th>sequential search (unordered sequence)</th>
<th>binary search (ordered sequence)</th>
</tr>
</thead>
<tbody>
<tr>
<td>search</td>
<td>search for a key</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>insert</td>
<td>insert a key</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>delete</td>
<td>delete a key</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>min/max</td>
<td>smallest/largest key</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>floor/ceiling</td>
<td>predecessor/successor</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>rank</td>
<td>number of keys less than key</td>
<td>$O(n)$</td>
</tr>
</tbody>
</table>
Binary Search Trees
Binary Search Tree

A BST is a **binary tree**
Binary Search Tree

A BST is a **binary tree**

A BST has **symmetric order**

each node $x$ in a BST has a key $\text{key}(x)$

for all nodes $y$ in the left subtree of $x$, $\text{key}(y) < \text{key}(x)$ **

for all nodes $y$ in the right subtree of $x$, $\text{key}(y) > \text{key}(x)$ **

(**) assume that the keys of a BST are pairwise distinct
The image contains a graph with nodes labeled with numbers from 20 to 80. The graph starts with a node labeled "< 50," which branches into two paths: one leading to 30 and the other to 25, both of which further branch into nodes 21, 25, 20, 40, 60, 70, 75, and 80. The structure forms a hierarchical diagram with connections indicating a flow from lower to higher values.
class BSTNode {

private:
    int data;
    BSTNode *left;
    BSTNode *right;

public:
    BSTNode(int d) {
        data = d;
        left = right = NULL;
    }
    ~BSTNode() {}

    friend class BSTree;
};
class BSTree {

private:
    BSTNode *root;

    BSTNode *search(BSTNode *p, int d);
    void insert(BSTNode **p, int d);
    void destroy(BSTNode *p);

    void preorder(BSTNode *p);
    void inorder(BSTNode *p);
    void postorder(BSTNode *p);

public:
    BSTree();
    ~BSTree();

    void insert(int d);
    bool remove(int d);
    BSTNode *search(int d);
    void traversal(int type);
};
class BSTree {

private:
    BSTNode *root;

    BSTNode *search(BSTNode *p, int d);
    void insert(BSTNode **p, int d);
    void destroy(BSTNode *p);

    void preorder(BSTNode *p);
    void inorder(BSTNode *p);
    void postorder(BSTNode *p);

public:
    BSTree();
    ~BSTree();

    void insert(int d);
    bool remove(int d);
    BSTNode *search(int d);
    void traversal(int type);
};
class BSTree {

private:
    BSTNode *root;

    BSTNode *search(BSTNode *p, int d);
    void insert(BSTNode **p, int d);
    void destroy(BSTNode *p);
    void preorder(BSTNode *p);
    void inorder(BSTNode *p);
    void postorder(BSTNode *p);

public:
    constructor & destructor
        BSTree();
        ~BSTree();
    }

        basic operations
    void insert(int d);
    bool remove(int d);
    BSTNode *search(int d);
    void traversal(int type);

};
class BSTree {

private:
    BSTNode *root;

    BSTNode *search(BSTNode *p, int d);
    void insert(BSTNode **p, int d);
    void destroy(BSTNode *p);

    void preorder(BSTNode *p);
    void inorder(BSTNode *p);
    void postorder(BSTNode *p);

public:
    BSTree();
    ~BSTree();

    void insert(int d);
    bool remove(int d);
    BSTNode *search(int d);
    void traversal(int type);

};
Basic Operations
Search

Start at root node

If the search key matches the current node’s key then **found**

If search key is greater than current node’s key
  search recursively on right child

If search key is less than current node’s key
  search recursively on left child

Stop recursion when current node is NULL (**not found**)
Search for key = 25
Search for key = 25
Search for key = 25

key < 50
Search for key = 25

key < 30

key < 50

20

21

25

40

50

60

70

80

75
Search for key = 25
Search for key = 25
Search for key = 25

key < 50

key < 30

key < 30

key > 20

key > 20

key > 20
Search for key = 25

key < 50

key < 30

key < 30

key > 20

key found!

Search for key = 25
Search for key = 77
Search for key = 77
Search for key = 77
Search for key = 77

50

30

20  40

25

21

70

60  80

75

key > 50
Search for key = 77

key > 50

key > 70
Search for key = 77

key > 50

key > 70
Search for key = 77
Search for key = 77
Search for key = 77

- key > 50
- key > 70
- key < 80
- key > 75, not found!
BSTNode *BSTTree::search(BSTNode *p, int d) {
    if (p) {
        if (p->data == d) return p;
        else if (p->data < d) return search(p->right, d);
        else return search(p->left, d);
    }
    return NULL;
}

BSTNode *BSTree::search(int d) {
    return search(root, d);
}