Binary Search: Search method that works on sorted sets of data, typically stored within an array, to find the location matching the search **key**

Basic procedure checks the middle value:

If value matches, return location.
If key is greater than value, we can reduce search space to the upper half
If key is less than value, we can reduce search space to the lower half

Example:

10	11	12	100	200	201	304	500	501	999
0	1	2	3	4 ↑ middle	5	6	7	8	9

key = 12

$12 < 200 \Rightarrow$ value must be located in lower half $(0 \to 3)$

* Process repeats until key is found, or there are no more elements to search

* Need a method to keep track of the elements we are searching.
 ↳ can use a left and right index to keep track of current search space

① Init left → 0 and right → size - 1

② while there are elements to search ≡ while left ≤ right

  a) middle → (left + right) / 2

  b) if data_vals[middle] == key, return middle

  c) if data_vals[middle] < key, left → middle + 1

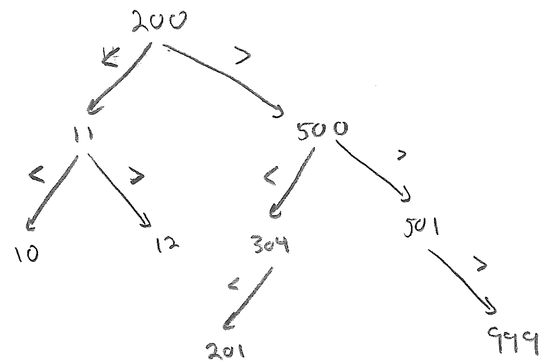
  d) else right = middle - 1

③ return -1

* **What is the complexity?**

Search Pattern Example:

10	11	12	200	201	304	500	501	999



*What if we are using some data that is not stored in an array?

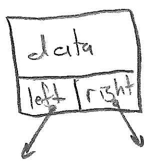
↳ Can we create a data structure to store elements/nodes and support the addition and deletion of nodes?

Binary Trees:

- Hierarchical arrangements of nodes in which each node can have two nodes immediately below it

- Each node consists of data and two pointers to nodes one level below

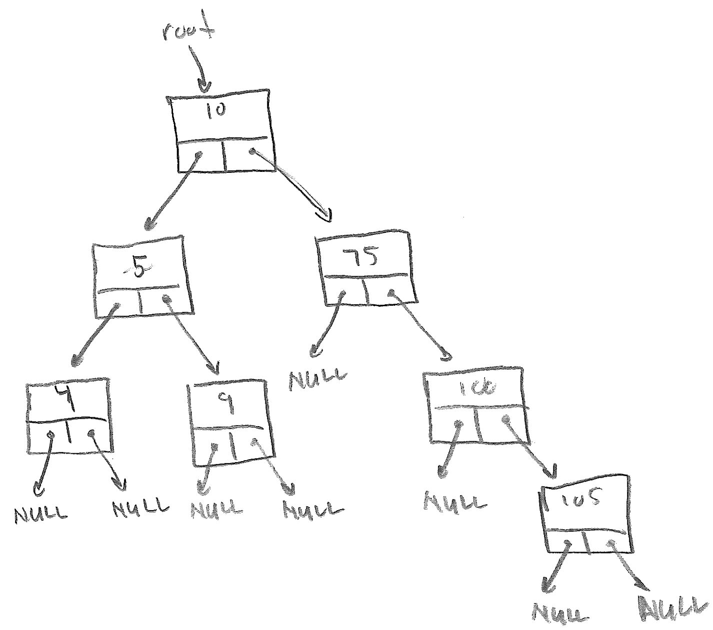
- Nodes one level below current node are called **children/descendents**
- Node one level above current node is called **parent**
- **Left** and **right** pointers typically used to represent pointers within node



- Node with no children is a **leaf** node

- **Root** node is the single node at the top level of hierarchy

Example:



Note: Nodes could also contain pointer to parent node.

Binary Tree Declarations:

```
typedef struct BiTreeNode_ {
    int data;
    struct BiTreeNode_ * left;
    struct BiTreeNode_ * right;
} BiTreeNode;


typedef struct BiTree_ {
    BiTreeNode * root;
    int size;
} BiTree;


void bitree_init (BiTree *tree);
void bitree_destroy (BiTree *tree);
int  bitree_ins_left (BiTree * tree, BiTreeNode *node, int data); // inserts only as leaf node
int  bitree_ins_right (BiTree *tree, BiTreeNode *node, int data); // inserts only as leaf node
int  bitree_rem_left (BiTree *tree, BiTreeNode *node); //removes entire subtree
int  bitree_rem_right (BiTree *tree, BiTreeNode *node); //removes entire subtree
```

Binary Search Tree: Binary tree in which nodes are organized to aid in effecient searching

- An element within the node is used as a key to determine how nodes are organized

- All elements within the left subtree will have a smaller key than the current node

- All elements within the right subtree will have a larger key than the current node

- Duplicate keys are not allowed.

What is the complexity of a binary search tree search?

Binary Search Tree Interface:

```
int bstree_insert(BSTree *tree, int val);
int bstree_remove(BSTree *tree, int val);
BSTreeNode * bstree_lookup(BSTree *tree, int search_key);
```

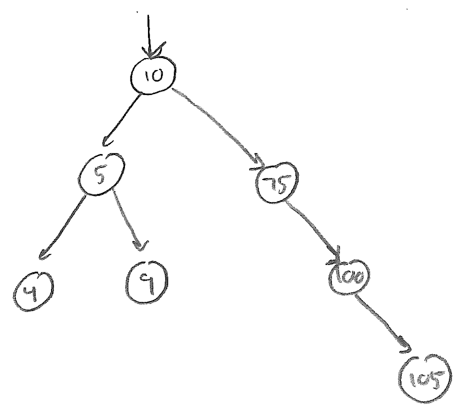
Tree Traversals:

Preorder: Traverse root, Traverse left, Traverse right

Postorder: Traverse left, traverse right, traverse root

Inorder: Traverse left, traverse root, traverse right

Example: Printing Tree



Preorder: 10 5 4 9 75 100 105

Postorder: 4 9 5 105 100 75 10

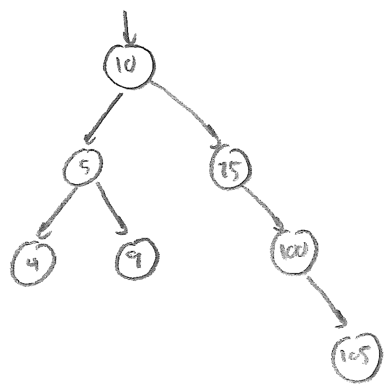
Inorder: 4 5 9 10 75 100 105

Inorder Binary Tree Traversal (Recursion)

```
void  bitree_print_inorder(BiTreeNode *node);


    if node != NULL then
        bitree_print_inorder(node -> left)
        print node -> val
        bitree_print_inorder(node -> right)
    endif
```

Example:



Note: Integer values below used to illustrate current node

```
bitree_print_inorder (10)
 ↳ if 10 != NULL then
      bitree_print_inorder(10→ left ≡ 5)
        ↳ if 5 != NULL then
             bitree_print_inorder (5→ left ≡ 4)
               ↳ if 4 != NULL then
                    bitree_print_inorder (4→ left ≡ NULL)
                      ↳ if NULL != NULL then
                        endif
                      ↵
                      print 4
                      bitree_print_inorder (4→ right ≡ NULL)
                        ↳ if NULL != NULL then
                          endif
                        ↵
                 endif
               ↵
               print 5
               bitree_print_inorder (5→ right ≡ 9)
                 ↳ if 9 != NULL then
                      bitree_print_inorder (9→ left ≡ NULL)
                        ↳ if NULL != NULL then
                          endif
                        ↵
                      print 9
                      bitree_print_inorder (9→ right ≡ NULL)
                        ↳ if NULL != NULL then
                          endif
                        ↵
                   endif
          endif
        ↵
      print 10
      bitree_print_inorder (10→ right ≡ 75)
        ↳ if
```

void bitree_init (BiTree *tree);

Operation: Init tree to empty tree

Pseudocode: ① tree→size = 0

② tree→root = NULL

void bitree_destroy (BiTree *tree);

Operation: Destroy tree by removing all elements

Pseudocode: ① remove subtree rooted at root

↳ bitree_rem_left (tree, NULL)

int bitree_ins_left (BiTree *tree, BiTreeNode *node, int data);

Operation: Insert new node as left child of specified node. Only allowed as insertion as leave node. If node is NULL and tree is empty, insert as root node

Pseudocode: ① Allocate new_node. //requires call to malloc

② Init new_node.

a) new_node → data = data.

b) new_node → left = NULL.

c) new_node → right = NULL.

] new node is always leaf node

③ if node is NULL //insert root node

a) if tree is not empty, return error (-1) //and free new_node

b) tree→root = new_node

④ else

a) if node→left is not NULL, return error (-1) //and free node

b) node→left = new_node

⑤ increment size

⑥ return success (0)

void bitree_rem_left(BiTree *tree, BiTreeNode *node);

Operation: Removes subtree rooted at left child of specified node. Nodes removed using a postorder traversal. If node is NULL, the root of tree (i.e. entire tree) will be removed.

Pseudocode: ① if tree is empty, return

② if node is NULL

a) bitree_rem_left(tree, tree->root)

b) bitree_rem_right(tree, tree->root)

c) free root

d) root = NULL

e) decrease tree size

③ else if node->left != NULL

a) bitree_rem_left(tree, node->left)

b) bitree_rem_right(tree, node->left)

c) free node->left

d) node->left = NULL

e) decrease tree size

Binary Search Tree

+ can be implemented as a Binary Tree

↳ typedef BiTree BiSTree;

↳ typedef BiTreeNode BiSTreeNode;

BiSTreeNode* bistree_lookup (BiSTree *tree, int key);

Operation: Calls recursive lookup function to find node that matches specified key. lookup() function is required as pointer to current node being searched is necessary for tree traversal

Pseudocode: ① return lookup (tree, tree->root, key)

BiSTreeNode* lookup (BiSTree *tree, BiSTreeNode *node, int key

---

Operation: Performs preorder traversal of tree rooted at node, returning pointer to node matching key if it exists. Returns NULL otherwise

Pseudocode: ① if node is NULL, return NULL

② compval = compare (key, node->data)

③ if compval is 0 (match found), return node

④ else if compval < 0 (key < node)

a) return lookup (tree, node->left, key)

⑤ else (key > node)

a) return lookup (tree, node->right, key) int bistree_insert (BisTree *tree, int data);

Operation: Insert new node as leaf node into binary tree by calling recursive insert() function.

Pseudocode: ① return insert(tree, tree→root, data)

int insert (BisTree *tree, BisTreeNode * node, int data)

Operation: Inserts new node as leaf within subtree rooted at node. If node is NULL, inserts as root of tree

Pseudocode: ① if node is NULL, return bitree_ins_left (tree, NULL, data)

② compval = compare (key, node→data) *key is the same as data in this example

③ if node with same key found (compval == 0)

a) return -1

④ if key < node→data (compval < 0)

a) if node→left == NULL, return bitree_ins_left (tree, node, data)

b) else return insert (tree, node→left, data)

⑤ else (key > node)

a) if node→right == NULL, return bitree_ins_right (tree, node, data)

b) else return insert (tree, node→right, data) int bstree_remove (BisTree * tree, int key);

Operation: Removes node matching key from tree if node is found

Pseudocode:
① node = lookup (tree, key)

② if node is NULL, return error

③ parent = parent (tree, node)

④ if node→left is NULL (i.e. no left child)

a) if parent is NULL, tree→root = node→right (i.e. remove root)

b) else if parent→left is node, parent→left = node→right

c) else parent→right = node→right

d) free node, decrease tree size, return success

⑤ else if node→right is NULL (i.e. no right child)

a) if parent is NULL, tree→root = node→left

b) else if parent→left is node, parent→left = node→left

c) else parent→right = node→left

d) free node, decrease tree size, return success

⑥ else (i.e. left and right child)

a) suc = node→right

b) while suc→left is not NULL, suc = suc→left (i.e. find successor)

c) suc_parent = parent (suc)

d) if suc is node→right (i.e. successor is node's right child)

I> suc→left = node→left

II> if parent == NULL, tree→root = suc

III> if parent→left is node, parent→left = suc

IV> else parent→right = suc

V> free node, decrease tree size, return success

e) else (i.e. the tricky case)

I> suc_parent→left = suc→right

II> suc→right → node→right

III> suc→left → node→left

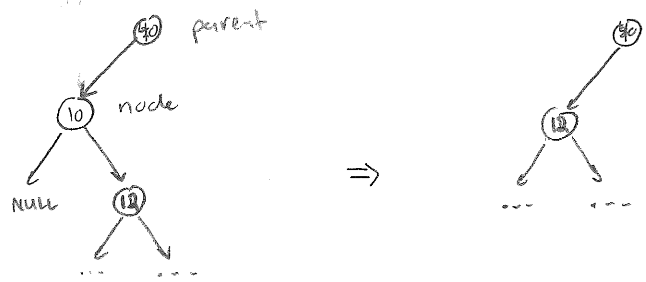
IV> if parent = null, tree→root = suc

V> else if parent→left is node, parent→left = suc

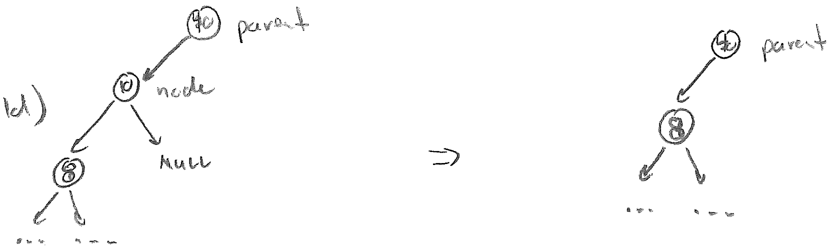
VI> else parent→right = suc

VII> free node, decrease tree size, return success

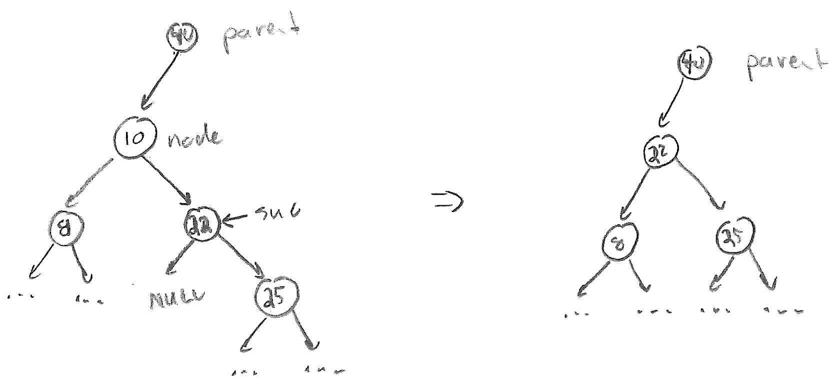
case (a):
(no left child)



case (b):
(no right child)



case (c):
(successor is right child)



case (d):
(successor is not right child)