---

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, ins-t as root node

Pseudocode: ① Allocate new_node

② Init new_node

a) new_node -> data = data
b) new_node -> left = NULL
c) new_node -> right = NULL

③ if node is NULL

a) if tree is not empty, return error (-1)

b) tree->root = new_node

④ else

a) if node->left is not NULL, return error (-1)

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

~~④ decrease tree size~~

Binary Search Tree

+ can be implemented as a Binary Tree
  - ↳ typedef BiTree BsTree;
  - ↳ typedef BiTreeNode BsTreeNode;

BsTreeNode* bstree_lookup(BsTree *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)

BsTreeNode* lookup(BsTree *tree, BsTreeNode *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 bitree_insert (BiTree *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 (BiTree *tree, BiTreeNode * 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 (BsTree * 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

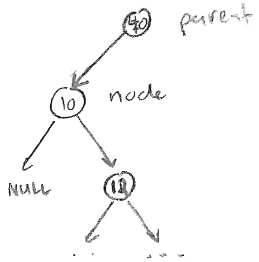
③ parent = parent (tree, node)

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

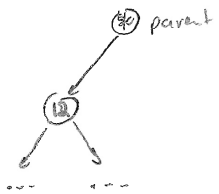
⑤ else if node→right is NULL (i.e. no right child)
- a) if parent is NULL, tree→root = node→left
- b) else if parent→left is node, parent→left = node→left
- c) else parent→right = node→left
- d) free node, decrease tree size, return success

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

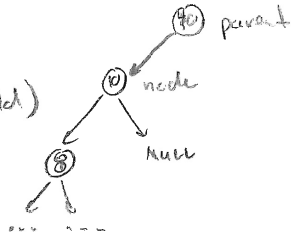
case (a):
(no left child)



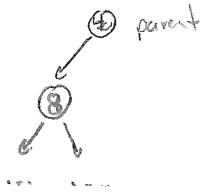
⇒



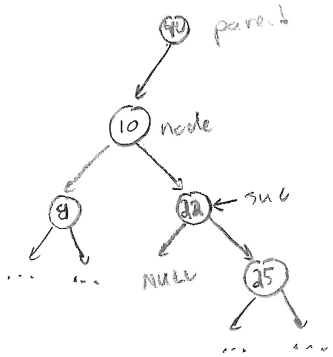
case (b):
(no right child)



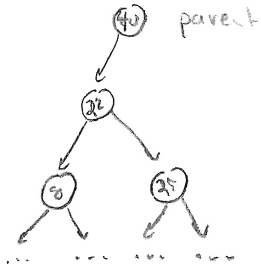
⇒



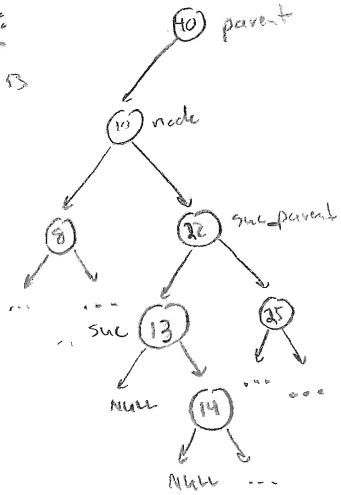
case (c):
(successor is right child)



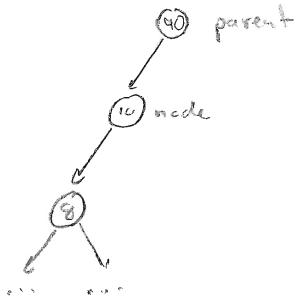
⇒

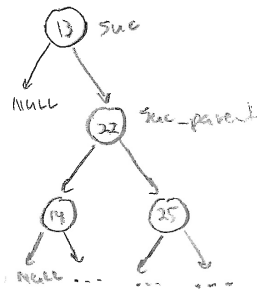


case (d):
(successor is not right child)



⇒ (b.c. I → II)





⇓ (b.c. III → IV)