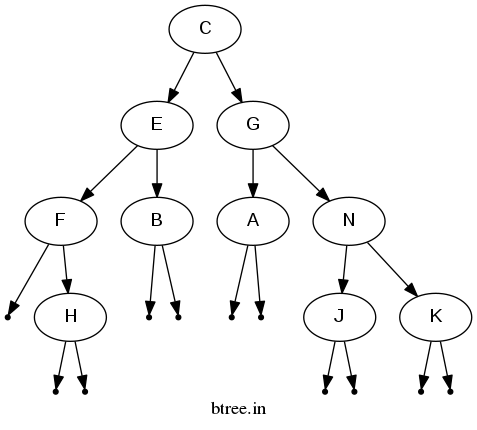
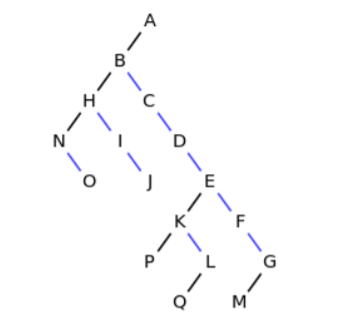
**Visualization of Trees Problem follows…**

Visualization of Trees problem is designed to demonstrate my knowledge of tree creation. It consists of an example tree containing letters of the alphabet as nodes.

****

**Problem 1 follows…**  
**Description:** Problem 1 is designed to demonstrate my knowledge of tree terminology. It consists of an example tree containing letters of the alphabet as nodes.

**Answers to Questions:**



1. What is the root node?

Letter A is the root node since it is the top-most node.

2. What is the degree of each of the nodes?

The degree of A, C, D, F, G, I, L and N nodes is 1 because each of these nodes has one subtree.

The degree of B, E, H and K nodes is 2 because each of these nodes has two subtrees.

The degree of J, M, O, P and Q nodes is 0 because each of these nodes has zero subtrees.

3. What are the parents and children of nodes K and O?

The parent of node K is letter E. The children of node K are letters P and L.

The parent of node O is letter N. Node O has no children.

4. What are the leaf nodes?

The leaf nodes are nodes with no children. Therefore, the J, M, O, P and Q nodes that have zero subtrees are leaf nodes.

5. What is the weight of the tree?

The weight of the tree is 5 because there are 5 leaf nodes, or nodes with no children.

6. What is the depth of nodes J and D

The depth of node J is 4 because it exists on level 4 of the binary tree. The root is considered as level zero. To get to node J, four branches must get traversed.

The depth of node D is 3 because it exists on level 3 of the binary tree. To get to node D, three branches must get traversed.

7. What is the height of the tree?

The height of the tree is 7 because it has 7 subtree levels below its root, letter A.

8. Is the tree a binary tree? Provide a rationale for your answer.

The tree is a binary tree because it is a valid tree where the number of children associated with a parent is a maximum of two.

**Problem 2 follows…**  
**Description:** Problem 2 is designed to demonstrate building a binary tree. It consists of functions and definitions for the binary tree including building a tree, pre-order/in-order/post-order transversal, visit (printing a node’s name) and deleting the binary tree. AllocationCount will tell how many things are currently dynamically allocated.

**Program Code (KAL\_P5\_1.c):**

// Program P5.1 from Kalicharan, modified to make it easier

// to document and understand.

//

// It uses recursive implementations of pre, in and post order

// traversals.

//

// An additonal modification is the addition of the parent linkage

// that can be used to traverse from a node back through its parent.

//

// This program is designed to demonstrate building a binary tree. It

// consists of functions and definitions for the binary tree including

// building a tree, pre-order/in-order/post-order transversal, visit

// (printing a node’s name) and deleting the binary tree. AllocationCount

// will tell how many things are currently dynamically allocated.

// It uses a recursive routine, that is a routine which calls itself.

// It is used in this program while searching a tree to delete all

// of its nodes and free memory.

// printf and reading a FILE support

#include <stdio.h>

// we will use strcpy() and strcmp() from string.h

#include <string.h>

// stdlib provides the definition of NULL and the declarations for

// malloc() and free()

#include <stdlib.h>

// this is an int constant that limits a node/read-in-word's name to a maximum

// length, or number of characters, to this set value

#define MaxWordSize 20

// this is a string constant that sets the name of the file to open and read in

// words from

#define FILENAME "btree.in"

// To make sure we are allocating and deallocating dynamic memory,

// variable AllocationCount is declared and referenced by any other

// code that does dynamic memory allocation and deallocation

int AllocationCount = 0;

// NodeData struct contains one char array element

// called word to allow for a node's name to be set

// to a MaxWordSize number of characters (20) and

// a NULL (zero byte) at the end

typedef struct {

char word[MaxWordSize+1];

} NodeData;

// TreeNode struct contains two elements pertaining

// to characteristics of a node including

// one NodeData struct type to hold a node's name

// and a struct of its own type called treeNode

// to allow for pointers to adjacent nodes such

// as parent and left/right subtrees

typedef struct treeNode {

NodeData data;

struct treeNode \*left, \*right, \*parent;

} TreeNode, \*TreeNodePtr;

// BinaryTree struct contains one TreeNode struct

// point element to be used as a binary tree.

// The TreeNode struct is important as it will be

// the binary tree's root node and is referenced as

// -> root

typedef struct {

TreeNodePtr root;

} BinaryTree;

// buildTree creates all nodes for a binary tree and assigns each one a word

// with maximum length as its name. The word is grabbed from an opened file

// and this function allocates memory for each newly created

// node. It takes a file name and a binary tree root node as arguments.

// It returns the node which gets created and assigned a name.

TreeNodePtr buildTree (FILE \* in, TreeNodePtr nodeParent);

// Pre-order traversal of a tree.

// First visit the root node, then the left subtree and finally the right subtree.

// Every node may represent a subtree itself.

void preOrder (TreeNodePtr nodeP);

// In-order traversal of a tree.

// First visit the left subtree, then the root and finally the right subtree.

// Every node may represent a subtree itself.

void inOrder (TreeNodePtr nodeP);

// Post-order traversal of a tree.

// First visit the left subtree, then the right subtree and finally the root node.

// Every node may represent a subtree itself.

void postOrder (TreeNodePtr nodeP);

// Visit takes a node to print and display its name, a letter.

// It returns nothing

void visit (TreeNodePtr nodeP);

// deleteTree() deletes the frees the storage that was allocated by the call

// to buildTree()

void deleteTree(TreeNodePtr nodeParent);

// the main takes no arguments, but opens a file to read in

// if the file cannot be opened, it prints a message and

// exits the program. It prints AllocationCount throughout

// its lifetime to provide insight into how many items

// have currently dynamically allocated memory.

// It is responsible for building a binary tree then

// conducting pre-order/in-order/post-order transversals

// while printing out the results as names of nodes in

// the order of the respective transversal. After all

// transversals, the tree is deleted by searching for the lowest

// node using post-order transversal and deleting and freeing

// that node until no nodes exist. Then the binary tree root is

// set to NULL to reflect that the binary tree no longer exists.

// Then the main calls an in-order transversal to prove the

// tree no longer exists and that no node names are printed

// and displayed. The main then

// closes the file which data is being read from.

int main()

{

// try to open a file

FILE \* in = fopen(FILENAME, "r");

// if the file doesn't exists or is empty

// print and display an error message and exit the program

if (in == NULL) {

printf ("Unable to open file %s... exiting\n", FILENAME);

exit (0);

}

// print and display number of items currently dynamically

// allocated

printf("Allocation count starts at %d\n", AllocationCount);

// initialize a new binary tree to prepare for building its nodes

BinaryTree bt;

// build the binary tree using the newly initialized binary tree

// create nodes for each of the words contained in the file

// it takes a NULL argument along with the file name because

// the node parent (in this case the root) is still NULL

// from being newly initialized

bt.root = buildTree(in, NULL);

printf("Allocation count after tree build is %d\n", AllocationCount);

// Dp a pre-order traversal of a tree.

// First visit the root node, then the left subtree and finally the right subtree.

printf("\nThe pre-order traversal is: ");

preOrder(bt.root);

// Do an in-order traversal of a tree.

// First visit the left subtree, then the root and finally the right subtree.

printf("\n\nThe in-order traversal is: ");

inOrder(bt.root);

// Do a post-order traversal of a tree.

// First visit the left subtree, then the right subtree and finally the root node.

printf("\n\nThe post-order traversal is: ");

postOrder(bt.root);

printf("\n\n");

// delete the binary tree and free all node memory

printf("Starting tree deletion...\n");

deleteTree(bt.root);

// set the tree to NULL to reflect the binary tree no longer exists

bt.root = NULL;

// prove the binary tree does not exist by printing and displaying

// AllocationCount to equal zero and conduct a final in-order transversal

// Nothing will print

printf("\nAllocation count after deleting the tree is %d\n", AllocationCount);

printf("\n\nThe in-order traversal is now: \n");

inOrder(bt.root);

printf("back from call to in-order. Nothing should have printed.\n");

// close the file being read

fclose(in);

} // end main

// buildTree takes a file name and a binary tree root node as arguments.

// It is responsible for searching the file for a name, creating a new

// node if the name exists and assigning it to the node's name, or struct

// item called "word."

// if the name is "@" or NULL then the search jumps to the next available

// subtree.

// It uses recursion to call itself and repeat this process for all existing

// nodes within the binary tree.

// Each time a node is created, memory is dynamically allocated

// for it. AllocationCount is increased to reflect the dynamically

// allocated memory. It returns the node which gets created and assigned

// a name.

TreeNodePtr buildTree (FILE \* in, TreeNodePtr nodeParent)

{

// initialize a string variable to hold MaxWordSize number of

// characters (20) and a NULL (zero byte) at the end

char str[MaxWordSize+1];

// look through the file for names

fscanf(in, "%s", str);

// check if the found string is not a name

// but is an "@" character

// i.e. NULL

if (strcmp(str, "@") == 0) return NULL;

// allocate memory for the node

TreeNodePtr p = (TreeNodePtr) malloc(sizeof(TreeNode));

// increase AllocationCount to reflect the newly created memory

AllocationCount++;

// set the node's name, or struct item called "word"

strcpy(p -> data.word, str);

// set the node's parent, or node which resides above it

p -> parent = nodeParent;

// set the node's left subtree node

p -> left = buildTree(in, p);

// set the node's right subtree node

p -> right = buildTree(in, p);

// return the node who's information was

// just set

return p;

} //end buildTree

// deleteTree() uses recursion to search the tree, delete and free up all of its nodes

// It returns nothing

void deleteTree(TreeNodePtr rootNode)

{

// check if binary tree node is not NULL

if (rootNode != NULL) {

// delete node's left subtree

deleteTree(rootNode->left);

// delete node's right subtree

deleteTree(rootNode->right);

// display which node will be deleted and freed

printf("Node ");

visit(rootNode);

printf("has been freed\n");

// delete and free the selected node

free(rootNode);

// decrease AllocationCount to reflect the freed memory

AllocationCount--;

}

}

// Print and display a node's name, a letter,

// that takes the binary tree's root node as an argument

// It returns nothing

void visit(TreeNodePtr nodeP)

{

// print and display the node's name, in this program it is

// an letter, by accessing its structure's item called word

printf("%s ", nodeP -> data.word);

} //end visit

// Pre-order traversal of a tree that takes the binary tree's root node as an argument

// First visit the root node, then the left subtree and finally the right subtree.

// Every node may represent a subtree itself.

// It returns nothing.

// It uses a recursive routine to call itself to traverse the tree via pre-order method

// and call visit() to print and display each currently

// looked at node's name

void preOrder(TreeNodePtr nodeP)

{

// check that the node exists

if (nodeP != NULL) {

// print and display the existing node's name

visit(nodeP);

// repeat this function's logic at the left subtree's nodes

// using recursion, calling this function again

preOrder(nodeP -> left);

// repeat this function's logic at the right subtree's nodes

// using recursion, calling this function again

preOrder(nodeP -> right);

}

} //end preOrder

// In-order traversal of a tree that takes the binary tree's root node as an argument

// First visit the left subtree, then the root and finally the right subtree.

// Every node may represent a subtree itself.

// It returns nothing.

// It uses a recursive routine to call itself to traverse the tree via in-order method

// and call visit() to print and display each currently

// looked at node's name

void inOrder(TreeNodePtr nodeP)

{

// check that the node exists

if (nodeP != NULL) {

// repeat this function's logic at the left subtree's nodes

// using recursion, calling this function again

inOrder(nodeP -> left);

// print and display the existing node's name

visit(nodeP);

// repeat this function's logic at the right subtree's nodes

// using recursion, calling this function again

inOrder(nodeP -> right);

}

} //end inOrder

// Post-order traversal of a tree that takes the binary tree's root node as an argument

// First visit the left subtree, then the right subtree and finally the root node.

// Every node may represent a subtree itself.

// It returns nothing.

// It uses a recursive routine to call itself to traverse the tree via post-order method

// and call visit() to print and display each currently

// looked at node's name

void postOrder(TreeNodePtr nodeP)

{

// check that the node exists

if (nodeP != NULL) {

// repeat this function's logic at the left subtree's nodes

// using recursion, calling this function again

postOrder(nodeP -> left);

// repeat this function's logic at the right subtree's nodes

// using recursion, calling this function again

postOrder(nodeP -> right);

// print and display the existing node's name

visit(nodeP);

}

} //end postOrder

**Data File:**C E F @ H @ @ B @ @ G A @ @ N J @ @ K @ @

**Program Output:**

Allocation count starts at 0

Allocation count after tree build is 10

The pre-order traversal is: C E F H B G A N J K

The in-order traversal is: F H E B C A G J N K

The post-order traversal is: H F B E A J K N G C

Starting tree deletion...

Node H has been freed

Node F has been freed

Node B has been freed

Node E has been freed

Node A has been freed

Node J has been freed

Node K has been freed

Node N has been freed

Node G has been freed

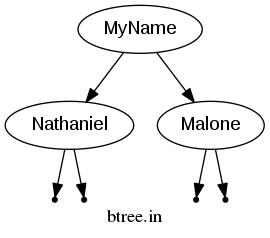
Node C has been freed

Allocation count after deleting the tree is 0

The in-order traversal is now:

back from call to in-order. Nothing should have printed.

**Extra Credit:**



**Problem 3 follows…**  
**Description:** Problem 3 is designed to demonstrate building a binary search tree. It consists of functions and definitions for the binary tree including creating a tree node, reading integers from a file, packaging node data into a struct, in-order transversal, finding or inserting a node, and deleting the binary tree. AllocationCount will tell how many things are currently dynamically allocated.

**Program Code KAL\_P5\_2.c):**

// Program P5.2 from Kalicharan, reformatted and ready for documentation and modification

//

// The program demonstrates how to form a BST and display it. It uses a file containing

// text as the input and the BST is formed from the integers. Each integer has

// an occurrence frequency, which is incremented whenever the identical integer is

// found in the file.

//

// After creating the BST, it is traversed and the output is written into a file.

// The traversal done is an in-order. When done against a BST, it will execute

// visits in a sorted manner! That's why the file produced has the integers in

// increasing order.

// stdio provides access to file processing and console processing

#include <stdio.h>

// stdlib provides malloc and free dynamic memory allocations

#include <stdlib.h>

// ctype gives us the following

// isalpha - looks at a character and returns 0 if it is not an alphanumeric

// and non 0 if it is

// tolower - returns a lower case version of a character (eg. A becomes a)

#include <ctype.h>

// stdbool defines "bool", "true" and "false"

// it is used whenever a boolean is being produced, used, or passed back to a caller

#include <stdbool.h>

// this is a string constant that sets the name of the file to open and read in

// integers from

#define INFILENAME "integers.in"

// this is a string constant that sets the name of the file to open and write in

// integers

#define OUTFILENAME "integersFreq.out"

// To make sure we are allocating and deallocating dynamic memory,

// variable AllocationCount is declared and referenced by any other

// code that does dynamic memory allocation and deallocation

int AllocationCount = 0;

// NodeData struct contains two integer elements, one

// called num to allow for a node's number value to be set

// and the other called freq to count how many times it is

// referenced within a file

typedef struct {

int num;

int freq;

} NodeData;

// TreeNode struct contains two elements pertaining

// to characteristics of a node including

// one NodeData struct type to hold a node's integer

// value and a struct of its own type called treeNode

// to allow for pointers to adjacent nodes such

// as parent and left/right subtrees

typedef struct treeNode {

NodeData data;

struct treeNode \*left, \*right;

} TreeNode, \*TreeNodePtr;

// BinaryTree struct contains one TreeNode struct

// point element to be used as a binary tree.

// The TreeNode struct is important as it will be

// the binary tree's root node and is referenced as

// -> root

typedef struct {

TreeNodePtr root;

} BinaryTree;

// getInt is a utility that reads integers from a file

bool getInt (FILE \*, int \*aNum );

// newTreeNode dynamically allocated a tree node, fills

// in the user data and clears the left and right side links

TreeNodePtr newTreeNode (NodeData nodeInformation);

// newNodeData is a utility that packages an integer and a frequency

// into a user data struct called NodeData

NodeData newNodeData (int aNum, int frequency);

// findOrInsert traverses a binary search tree looking for a node

// that matches the user criteria. In this case, it is an integer.

// It either returns the pointer to the matching node

// or it links the node into the tree at its proper, sorted location

TreeNodePtr findOrInsert(BinaryTree, NodeData nodeInformation);

// inOrder does a traversal of the BST that will write a file

// with the traversal results.

void inOrder (FILE \*, TreeNodePtr);

// deleteTree() deletes the frees the storage that was allocated by the call

// to buildTree()

void deleteTree(TreeNodePtr nodeParent);

// main will do the following:

// 1. It opens both an input and output file. Failure to be able to

// open the files will result in program termination

// 2. It will build a BST from the file integers and maintain a

// count of the number of occurrences of each integer

// 3. It will write the integers and their occurrences in a file

int main()

{

// integer is a scratch array used to get complete integers from the file

int integer;

// open the input file and exit if it does not open

FILE \* in = fopen( INFILENAME, "r");

if (in == NULL) {

printf ("Unable to open file %s... exiting", INFILENAME);

exit (0);

}

// the input file opened, so open the output file.

// it it does not open, close the input file and exit

FILE \* out = fopen( OUTFILENAME, "w");

if (out == NULL)

{

printf ("Unable to open file %s for output... exiting", OUTFILENAME);

fclose (in);

exit(0);

}

// print and display the number of items currently

// dynamically allocated

printf("Allocation count starts at %d\n\n", AllocationCount);

// both files are open, declare the binary search tree (bst)

// and initialize its root as NULL

BinaryTree bst;

bst.root = NULL;

// get integers and insert them into the BST.

// since new nodes will have a frequency of 0 and matching

// nodes will have a current frequency, bump the frequency

// in either case to reflect their occurrence in the text

while (getInt(in, &integer) == true) {

if (bst.root == NULL)

bst.root = newTreeNode(newNodeData(integer, 1));

else {

TreeNodePtr node = findOrInsert(bst, newNodeData(integer, 0));

node -> data.freq++;

}

}

// print and display the number of items currently

// dynamically allocated after building the tree

printf("Allocation count after tree build is %d\n\n", AllocationCount);

// The file has been processed. Write out the results by using

// an in-order traversal and writing as the "visit"

fprintf(out, "\nIntegers Frequency\n\n");

inOrder(out, bst.root);

fprintf(out, "\n\n");

// delete the binary tree and free all node memory

deleteTree(bst.root);

// set the tree to NULL to reflect the binary tree no longer exists

bst.root = NULL;

// prove the binary tree does not exist by printing and displaying

// AllocationCount to equal zero and conduct a final in-order transversal

// Nothing will print

printf("\nAllocation count after deleting the tree is %d\n\n", AllocationCount);

inOrder(out, bst.root);

printf("Back from inOrder.. nothing should have printed.\n");

// close both files and return. We're done!!

fclose(in);

fclose(out);

return 0;

} // end main

// getInt has the task of reading in integers from

// the file, updating the passed-in integer and returning

// a boolean indication if it was able to find/set an integer.

// The routine returns true if there is an integer and false if there is none

bool getInt(FILE \* in, int \*aNum)

{

// initialize a counter to describe

// how many integers are being read in

int cmd;

// read in a line from the file

cmd = fscanf(in, "%d", &\*aNum);

// check that the number of integers being read in

// is at least 1

if (cmd == 1) {

// at least 1 integer is being read in

// so return that an integer was found

return true;

} else {

// at least 1 integer is not being read in

// so return that an integer was not found

return false;

}

} // end getInt

// findOrInsert searches a binary tree looking for nodeInformation

// in this implementation, the nodeInformation field "integer" is being

// looked at to see if there is a match.

//

// Because this is a binary tree, we can keep comparing if the

// value is less than us (indicating to proceed left), is us, or

// greater than us (indicating to proceed right).

// If we hit the end of a left traversal and the value has not been found,

// we add it to our left side and return it

// In a similar manner, if we hit the end of a right traversal and the

// value has not been found, we add it to the right side and return it

// the node we return will either be a new node (zero frequency) or a node

// that we have found. That enables the caller to increment the frequency

// without knowing if this was a new node or a match

//

TreeNodePtr findOrInsert(BinaryTree bt, NodeData nodeInformation)

{

// if the root is empty, just return a node with the nodeInformation in it

if (bt.root == NULL) return newTreeNode(nodeInformation);

// curr is the tree node currently at the top of the tree

TreeNodePtr curr = bt.root;

// loop until we have a match (returns in the loop will happen if

// we hit the end of the tree

while (nodeInformation.num != curr -> data.num) {

// are we less than the node?

if (nodeInformation.num < curr -> data.num) {

// yes, need to go left if we can

// NULL indicates we are at a left leaf, so add the new node

// to the current leaf on the left side and return it

if (curr -> left == NULL)

return curr -> left = newTreeNode(nodeInformation);

else

// keep looking left since we are not at a leaf

curr = curr -> left;

} else {

// we are greater than the current right side

// need to go further right if we can

// NULL indicates we are at a right leaf, so add the new node

// to the current leaf on the right side and return it

if (curr -> right == NULL)

return curr -> right = newTreeNode(nodeInformation);

else

// keep looking right since we are not at a leaf

curr = curr -> right;

}

}

// we get here if the while completed, indicating we found a node

// that matched a current node in the tree

// return pointer to the node

return curr;

} //end findOrInsert

// newTreeNode makes a node through dynamic allocation, initializing

// its left and right children to NULL and inserting the user data provided

// as nodeInformation.

// It returns the pointer to the tree node it just allocated and initialized

TreeNodePtr newTreeNode(NodeData nodeInformation)

{

// allocate a tree node

TreeNodePtr p = (TreeNodePtr) malloc(sizeof(TreeNode));

// update AllocationCount to reflect the newly created memory

AllocationCount++;

// fill in the user data

p -> data = nodeInformation;

// NULL its links

p -> left = p -> right = NULL;

// give it back to the caller

return p;

} //end newTreeNode

// inOrder does a go left, visit, go right traversal through recursively

// calling itself. Instead of calling on a "visit", it accomplishes it

// by writing out the user data integer and occurrence count to the file

// provided

void inOrder(FILE \* out, TreeNodePtr node)

{

// if we are not NULL we cannot go further in the tree

if (node!= NULL) {

// go left

inOrder(out, node -> left);

// "visit" by printing the user information

fprintf(out, "%d %2d\n", node -> data.num, node -> data.freq); /////////////%d = -15s///////////

// go right

inOrder(out, node -> right);

}

return;

} //end inOrder

// deleteTree() uses recursion to search the tree, delete and free up all of its nodes

// It returns nothing

void deleteTree(TreeNodePtr rootNode)

{

// check if binary tree node is not NULL

if (rootNode != NULL) {

// delete node's left subtree

deleteTree(rootNode->left);

// delete node's right subtree

deleteTree(rootNode->right);

// display which node will be deleted and freed

printf("Node %d has been freed\n", rootNode->data);

// delete and free the node

free(rootNode);

// decrease AllocationCount to reflect the freed memory

AllocationCount--;

}

}

// newNodeData is a utility that builds a NodeData structure from

// the user data fields it receives as input. In this case, it is a integer

// that is NULL terminated and an initial frequency of occurrence

NodeData newNodeData(int aNum, int frequency)

{

// make a NodeData and fill it with the integer and frequency provided

NodeData temp;

temp.num = aNum;

temp.freq = frequency;

// return it with the packaging of the user data complete

return temp;

} //end newNodeData

**Data File (integers.in):**21

-33

15

12

-33

12

40

60

77

15

15

60

17

22

-3

**Output File (integersFreq.out)**

Integers Frequency

-33 2

-3 1

12 2

15 3

17 1

21 1

22 1

40 1

60 2

77 1

**Console Output:**

Allocation count starts at 0

Allocation count after tree build is 10

Node -3 has been freed

Node 12 has been freed

Node 17 has been freed

Node 15 has been freed

Node -33 has been freed

Node 22 has been freed

Node 77 has been freed

Node 60 has been freed

Node 40 has been freed

Node 21 has been freed

Allocation count after deleting the tree is 0

Back from inOrder.. nothing should have printed.

**Problem 4 follows…**  
**Description:** Problem 4 is designed to demonstrate building a binary search tree. It consists of functions and definitions for the binary tree including creating a tree node, reading integers from a file, packaging node data into a struct, in-order transversal, finding or inserting a node, printing a node’s name, parent and left/right children and siblings and deleting the binary tree. AllocationCount will tell how many things are currently dynamically allocated.

**Program Code (KAL\_P5\_1.c):**

// Program P5.1 from Kalicharan, modified to make it easier

// to document and understand.

//

// It uses recursive implementations of pre, in and post order

// traversals.

//

// An additonal modification is the addition of the parent linkage

// that can be used to traverse from a node back through its parent.

//

// This program is designed to demonstrate building a binary tree. It

// consists of functions and definitions for the binary tree including

// building a tree, pre-order/in-order/post-order transversal, visit

// (printing a node’s name) and deleting the binary tree. AllocationCount

// will tell how many things are currently dynamically allocated.

// It uses a recursive routine, that is a routine which calls itself.

// It is used in this program while searching a tree to delete all

// of its nodes and free memory.

// printf and reading a FILE support

#include <stdio.h>

// we will use strcpy() and strcmp() from string.h

#include <string.h>

// stdlib provides the definition of NULL and the declarations for

// malloc() and free()

#include <stdlib.h>

// this is an int constant that limits a node/read-in-word's name to a maximum

// length, or number of characters, to this set value

#define MaxWordSize 20

// this is a string constant that sets the name of the file to open and read in

// words from

#define FILENAME "btree.in"

// To make sure we are allocating and deallocating dynamic memory,

// variable AllocationCount is declared and referenced by any other

// code that does dynamic memory allocation and deallocation

int AllocationCount = 0;

// NodeData struct contains one char array element

// called word to allow for a node's name to be set

// to a MaxWordSize number of characters (20) and

// a NULL (zero byte) at the end

typedef struct {

char word[MaxWordSize+1];

} NodeData;

// TreeNode struct contains two elements pertaining

// to characteristics of a node including

// one NodeData struct type to hold a node's name

// and a struct of its own type called treeNode

// to allow for pointers to adjacent nodes such

// as parent and left/right subtrees

typedef struct treeNode {

NodeData data;

struct treeNode \*left, \*right, \*parent;

} TreeNode, \*TreeNodePtr;

// BinaryTree struct contains one TreeNode struct

// point element to be used as a binary tree.

// The TreeNode struct is important as it will be

// the binary tree's root node and is referenced as

// -> root

typedef struct {

TreeNodePtr root;

} BinaryTree;

// buildTree creates all nodes for a binary tree and assigns each one a word

// with maximum length as its name. The word is grabbed from an opened file

// and this function allocates memory for each newly created

// node. It takes a file name and a binary tree root node as arguments.

// It returns the node which gets created and assigned a name.

TreeNodePtr buildTree (FILE \* in, TreeNodePtr nodeParent);

// Pre-order traversal of a tree.

// First visit the root node, then the left subtree and finally the right subtree.

// Every node may represent a subtree itself.

void preOrder (TreeNodePtr nodeP);

// In-order traversal of a tree.

// First visit the left subtree, then the root and finally the right subtree.

// Every node may represent a subtree itself.

void inOrder (TreeNodePtr nodeP);

// Post-order traversal of a tree.

// First visit the left subtree, then the right subtree and finally the root node.

// Every node may represent a subtree itself.

void postOrder (TreeNodePtr nodeP);

// Visit takes a node to print and display its name, a letter.

// It returns nothing

void visit (TreeNodePtr nodeP);

// postOrderNodeDump is used to print out

// a node's name, parent, left and right children and left and right

// siblings. It takes a node and returns nothing.

void postOrderNodeDump (TreeNodePtr nodeP);

// deleteTree() deletes the frees the storage that was allocated by the call

// to buildTree()

void deleteTree(TreeNodePtr nodeParent);

// the main takes no arguments, but opens a file to read in

// if the file cannot be opened, it prints a message and

// exits the program. It prints AllocationCount throughout

// its lifetime to provide insight into how many items

// have currently dynamically allocated memory.

// It is responsible for building a binary tree then

// conducting pre-order/in-order/post-order transversals

// while printing out the results as names of nodes in

// the order of the respective transversal. After all

// transversals, the tree is deleted by searching for the lowest

// node using post-order transversal and deleting and freeing

// that node until no nodes exist. Then the binary tree root is

// set to NULL to reflect that the binary tree no longer exists.

// Then the main calls an in-order transversal to prove the

// tree no longer exists and that no node names are printed

// and displayed. The main then

// closes the file which data is being read from.

int main()

{

// try to open a file

FILE \* in = fopen(FILENAME, "r");

// if the file doesn't exists or is empty

// print and display an error message and exit the program

if (in == NULL) {

printf ("Unable to open file %s... exiting\n", FILENAME);

exit (0);

}

// print and display number of items currently dynamically

// allocated

printf("Allocation count starts at %d\n", AllocationCount);

// initialize a new binary tree to prepare for building its nodes

BinaryTree bt;

// build the binary tree using the newly initialized binary tree

// create nodes for each of the words contained in the file

// it takes a NULL argument along with the file name because

// the node parent (in this case the root) is still NULL

// from being newly initialized

bt.root = buildTree(in, NULL);

printf("Allocation count after tree build is %d\n", AllocationCount);

// Dp a pre-order traversal of a tree.

// First visit the root node, then the left subtree and finally the right subtree.

printf("\nThe pre-order traversal is: ");

preOrder(bt.root);

// Do an in-order traversal of a tree.

// First visit the left subtree, then the root and finally the right subtree.

printf("\n\nThe in-order traversal is: ");

inOrder(bt.root);

// Do a post-order traversal of a tree.

// First visit the left subtree, then the right subtree and finally the root node.

printf("\n\nThe post-order traversal is: ");

postOrder(bt.root);

printf("\n\n");

// print postOrderNodeDump headers

printf("node\tparent\tlchild\trchild\tlsib\trsib\n");

// print node, parent, left/right children and siblings

postOrderNodeDump (bt.root);

// delete the binary tree and free all node memory

printf("\nStarting tree deletion...\n");

deleteTree(bt.root);

// set the tree to NULL to reflect the binary tree no longer exists

bt.root = NULL;

// prove the binary tree does not exist by printing and displaying

// AllocationCount to equal zero and conduct a final in-order transversal

// Nothing will print

printf("\nAllocation count after deleting the tree is %d\n", AllocationCount);

printf("\n\nThe in-order traversal is now: \n");

inOrder(bt.root);

printf("back from call to in-order. Nothing should have printed.\n");

// close the file being read

fclose(in);

} // end main

// buildTree takes a file name and a binary tree root node as arguments.

// It is responsible for searching the file for a name, creating a new

// node if the name exists and assigning it to the node's name, or struct

// item called "word."

// if the name is "@" or NULL then the search jumps to the next available

// subtree.

// It uses recursion to call itself and repeat this process for all existing

// nodes within the binary tree.

// Each time a node is created, memory is dynamically allocated

// for it. AllocationCount is increased to reflect the dynamically

// allocated memory. It returns the node which gets created and assigned

// a name.

TreeNodePtr buildTree (FILE \* in, TreeNodePtr nodeParent)

{

// initialize a string variable to hold MaxWordSize number of

// characters (20) and a NULL (zero byte) at the end

char str[MaxWordSize+1];

// look through the file for names

fscanf(in, "%s", str);

// check if the found string is not a name

// but is an "@" character

// i.e. NULL

if (strcmp(str, "@") == 0) return NULL;

// allocate memory for the node

TreeNodePtr p = (TreeNodePtr) malloc(sizeof(TreeNode));

// increase AllocationCount to reflect the newly created memory

AllocationCount++;

// set the node's name, or struct item called "word"

strcpy(p -> data.word, str);

// set the node's parent, or node which resides above it

p -> parent = nodeParent;

// set the node's left subtree node

p -> left = buildTree(in, p);

// set the node's right subtree node

p -> right = buildTree(in, p);

// return the node who's information was

// just set

return p;

} //end buildTree

// deleteTree() uses recursion to search the tree, delete and free up all of its nodes

// It returns nothing

void deleteTree(TreeNodePtr rootNode)

{

// check if binary tree node is not NULL

if (rootNode != NULL) {

// delete node's left subtree

deleteTree(rootNode->left);

// delete node's right subtree

deleteTree(rootNode->right);

// display which node will be deleted and freed

printf("Node ");

visit(rootNode);

printf("has been freed\n");

// delete and free the selected node

free(rootNode);

// decrease AllocationCount to reflect the freed memory

AllocationCount--;

}

}

// Print and display a node's name, a letter,

// that takes the binary tree's root node as an argument

// It returns nothing

void visit(TreeNodePtr nodeP)

{

// print and display the node's name, in this program it is

// an letter, by accessing its structure's item called word

printf("%s ", nodeP -> data.word);

} //end visit

// Pre-order traversal of a tree that takes the binary tree's root node as an argument

// First visit the root node, then the left subtree and finally the right subtree.

// Every node may represent a subtree itself.

// It returns nothing.

// It uses a recursive routine to call itself to traverse the tree via pre-order method

// and call visit() to print and display each currently

// looked at node's name

void preOrder(TreeNodePtr nodeP)

{

// check that the node exists

if (nodeP != NULL) {

// print and display the existing node's name

visit(nodeP);

// repeat this function's logic at the left subtree's nodes

// using recursion, calling this function again

preOrder(nodeP -> left);

// repeat this function's logic at the right subtree's nodes

// using recursion, calling this function again

preOrder(nodeP -> right);

}

} //end preOrder

// In-order traversal of a tree that takes the binary tree's root node as an argument

// First visit the left subtree, then the root and finally the right subtree.

// Every node may represent a subtree itself.

// It returns nothing.

// It uses a recursive routine to call itself to traverse the tree via in-order method

// and call visit() to print and display each currently

// looked at node's name

void inOrder(TreeNodePtr nodeP)

{

// check that the node exists

if (nodeP != NULL) {

// repeat this function's logic at the left subtree's nodes

// using recursion, calling this function again

inOrder(nodeP -> left);

// print and display the existing node's name

visit(nodeP);

// repeat this function's logic at the right subtree's nodes

// using recursion, calling this function again

inOrder(nodeP -> right);

}

} //end inOrder

// Post-order traversal of a tree that takes the binary tree's root node as an argument

// First visit the left subtree, then the right subtree and finally the root node.

// Every node may represent a subtree itself.

// It returns nothing.

// It uses a recursive routine to call itself to traverse the tree via post-order method

// and call visit() to print and display each currently

// looked at node's name

void postOrder(TreeNodePtr nodeP)

{

// check that the node exists

if (nodeP != NULL) {

// repeat this function's logic at the left subtree's nodes

// using recursion, calling this function again

postOrder(nodeP -> left);

// repeat this function's logic at the right subtree's nodes

// using recursion, calling this function again

postOrder(nodeP -> right);

// print and display the existing node's name

visit(nodeP);

}

} //end postOrder

// postOrderNodeDump is used to print out

// a node's name, parent, left and right children and left and right

// siblings. It takes a node and returns nothing.

void postOrderNodeDump(TreeNodePtr nodeP){

// check that a node exists to print out relevant data

if (nodeP != NULL){

// repeat this function's logic at the left subtree's nodes

// using recursion, calling this function again

postOrderNodeDump(nodeP->left);

// repeat this function's logic at the right subtree's nodes

// using recursion, calling this function again

postOrderNodeDump(nodeP->right);

// print and display the node's name

printf("%s \t", nodeP->data.word);

// print and display the node's parent's name, or \*

// A node has a parent if its parent link is not NULL. The parent node name can be

// accessed by going through the parent link

if (nodeP->parent != NULL) {

printf("%s \t", nodeP->parent->data.word);

} else

printf("\* \t");

// print and display the node's left child's name, or \*

// A node has a left child if its left link is not NULL. The left child node name can be

// accessed by going through the left link

if (nodeP->left != NULL) {

printf("%s \t", nodeP->left->data.word);

} else

printf("\* \t");

// print and display the node's right child's name, or \*

// A node has a right child if its right link is not NULL. The right child node name

// can be accessed by going through the right link

if (nodeP->right != NULL) {

printf("%s \t", nodeP->right->data.word);

} else

printf("\* \t");

// print and display the node's left sibling's name, or \*

// A node has a left sibling if the node (a) has a parent and (b) the parent's left child

// exists and (c) the left child is not the same link as our node

if ((nodeP->parent != NULL) && (nodeP->parent->left != NULL) &&

(nodeP->parent->left != nodeP)) {

printf("%s \t", nodeP->parent->left->data.word);

} else

printf("\* \t");

// print and display the node's right sibling's name, or \*

// A node has a right sibling if the node (a) has a parent and (b) the parent's right

// child exists and (c) the right child is not the same link as our node

if ((nodeP->parent != NULL) && (nodeP->parent->right != NULL) &&

(nodeP->parent->right != nodeP)) {

printf("%s \n", nodeP->parent->right->data.word);

} else

printf("\* \n");

}

}

**Data File:**  
C E F @ H @ @ B @ @ G A @ @ N J @ @ K @ @

**Program Output:**

Allocation count starts at 0

Allocation count after tree build is 10

The pre-order traversal is: C E F H B G A N J K

The in-order traversal is: F H E B C A G J N K

The post-order traversal is: H F B E A J K N G C

node parent lchild rchild lsib rsib

H F \* \* \* \*

F E \* H \* B

B E \* \* F \*

E C F B \* G

A G \* \* \* N

J N \* \* \* K

K N \* \* J \*

N G J K A \*

G C A N E \*

C \* E G \* \*

Starting tree deletion...

Node H has been freed

Node F has been freed

Node B has been freed

Node E has been freed

Node A has been freed

Node J has been freed

Node K has been freed

Node N has been freed

Node G has been freed

Node C has been freed

Allocation count after deleting the tree is 0

The in-order traversal is now:

back from call to in-order. Nothing should have printed.

**Problem 4 follows…**  
**Description:** Problem 5 is designed to demonstrate how an array can implicitly define where parent, sibling and child nodes can be found relative to any one node within a binary tree.

**Solution:**

node parent lchild rchild lsib rsib

D \* A G \* \*

A D H F \* G

G D \* B A \*

H A \* \* \* F

F A \* M H \*

B G C \* \* \*

M F \* \* \* \*

C E \* \* \* \*

E \* \* \* \* \*

**File listing:**

D A H @ @ F @ M @ @ G @ B C E