

(30)

Write
your name

Georgy
-5

100
100

Excellent

Question 1.1:

$$T(n) = T(n/5) + 5, T(1) = 31$$

$$T(5) = T(5/5) + 5 = T(1) + 5 = 31 + 5 = 36$$

$$T(25) = T(25/5) + 5 = T(5) + 5 = 36 + 5 = 41$$

$$T(125) = T(125/5) + 5 = T(25) + 5 = 41 + 5 = 46$$

$$T(5) = 31 + 5$$

$$T(5^2) = 31 + 5 + 5$$

$$T(5^3) = 31 + 5 + 5 + 5$$

$$T(5^x) = 31 + \sum_{i=0}^{x-1} 5$$

$$5 \sum_{i=0}^{x-1} 1$$

From these three examples, the general expression is: $T(n) = 31 + 5 \log_5(n)$

Let $n = 5^x$, then $T(5^x) = 31 + 5x$

explain how?

Proof:

Base Case ($x = 0$):

$$T(5^0) = 31 + 5(0) = 31 + 0 = 31$$

Inductive step:

Assume that $T(5^x)$ is true, then $T(5^{x+1})$ is true.

The expected answer should be: $T(5^{x+1}) = 31 + 5(x+1)$

$$\text{now for } (x+1): T(5^{x+1}) = T(5^x) + 5 = 31 + 5x + 5 = 31 + 5(x+1)$$

$$\therefore T(5^x) = T(5^{x-1}) + 5 \quad \text{is equivalent to} \quad T(5^x) = 31 + 5x$$

Using mathematical induction. ■

Question 1.2:

$$T(n) = 2T(n/2) + n, T(1) = 1$$

$$T(2) = 2T(2/2) + 2 = 2T(1) + 2 = 2(1) + 2 = 2 + 2 = 4$$

$$T(4) = 2T(4/2) + 4 = 2T(2) + 4 = 2(4) + 4 = 8 + 4 = 12$$

$$T(8) = 2T(8/2) + 8 = 2T(4) + 8 = 2(12) + 8 = 24 + 8 = 32$$

From these three examples, the general expression is: $T(n) = n(\log_2(n) + 1)$

Let $n = 2^x$, then: $T(2^x) = 2^x(x + 1)$

How?

Proof:

Base step ($x = 0$):

$$T(2^0) = 2^0(0 + 1) = 1(1) = 1$$

Inductive step:

Assume that $T(2^x)$ is true, then $T(2^{x+1})$ is true

The expected answer should be: $T(2^{x+1}) = 2^{x+1}(x + 2)$

Now for $(x + 1)$:

$$T(2^{x+1}) = 2T(2^x) + 2^{x+1} = 2(2^x(x + 1)) + 2^{x+1} = 2^{x+1}(x + 1) + 2^{x+1}$$

$$= 2^{x+1}(x + 2)$$

$\therefore T(2^x) = 2T(2^{x-1}) + 2^x$ is equivalent to $T(2^x) = 2^x(x + 1)$

Using mathematical induction. ■

Question 1.3:

Theorem: $\sum_{i=1}^n i^2 = \frac{n(n+1)(2n+1)}{6}$

$$T(n) = T(n-1) + n^2, T(1) = 23$$

$$T(2) = T(1) + 4 = 23 + 4 = 27$$

$$T(3) = T(2) + 9 = 27 + 9 = 36$$

$$T(4) = T(3) + 16 = 36 + 16 = 52$$

From these three examples, the general expression is: $T(n) = 22 + \sum_{i=1}^n i^2$
 $= 22 + \frac{n(n+1)(2n+1)}{6}$

Proof:

Base step ($n = 1$):

$$T(1) = 22 + \frac{1(1+1)(2(1)+1)}{6} = 22 + 1 = 23$$

Inductive step:

Assume that $T(n)$ is true, then $T(n+1)$ is true.

The expected answer should be: $T(n+1) = 22 + \frac{(n+1)(n+2)(2n+3)}{6}$

Now for $(n+1)$:

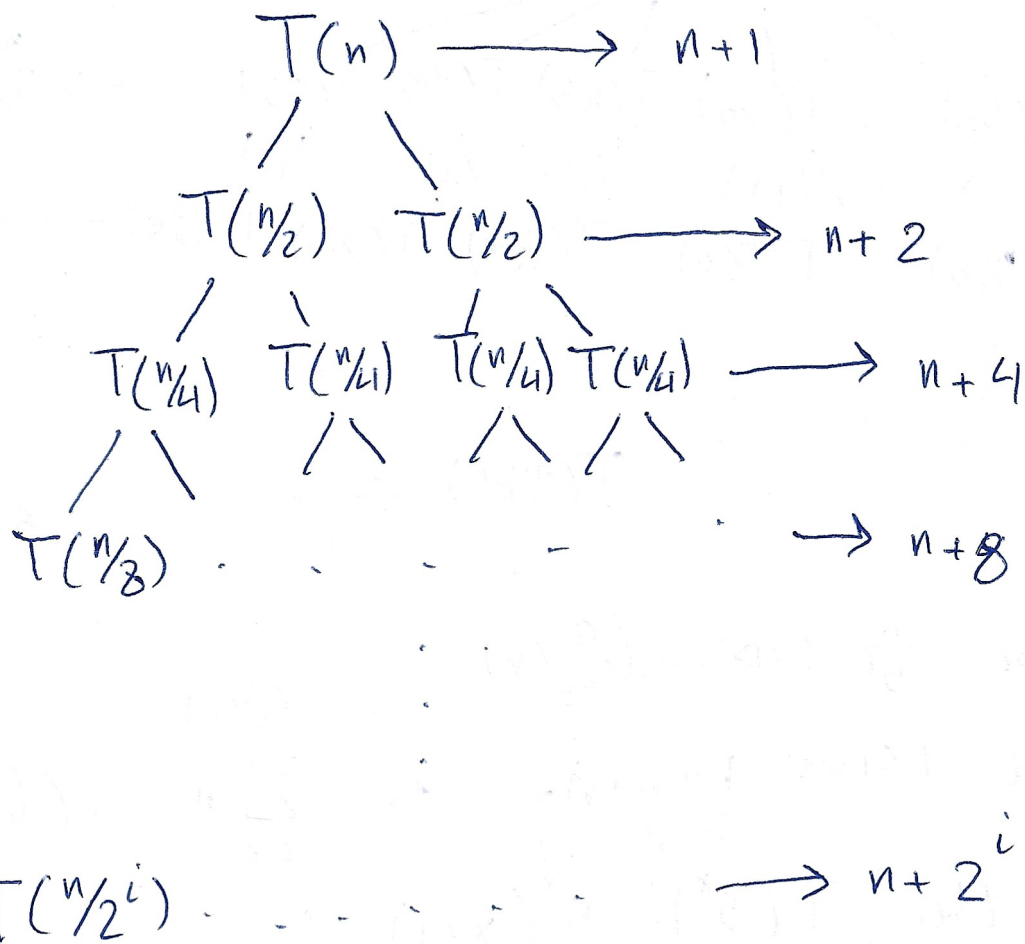
$$\begin{aligned} T(n+1) &= T(n) + (n+1)^2 = 22 + \frac{(n+1)(n+2)(2n+3)}{6} + (n+1)^2 \\ &= 22 + \frac{(n+1)(n+2)(2n+3) + 6(n+1)^2}{6} = 22 + \frac{(n+1)(n+2)(2n+3)}{6} \end{aligned}$$

$\therefore T(n) = T(n-1) + n^2$ is equivalent to $T(n) = 22 + \frac{n(n+1)(2n+1)}{6}$

Using mathematical induction. ■

$$T(n) = 2T(n/2) + n + 1, \quad T(1) = 1$$

P2
10/10



Stopping case: $\frac{n}{2^i} = 1 \Rightarrow i = \log_2(n)$

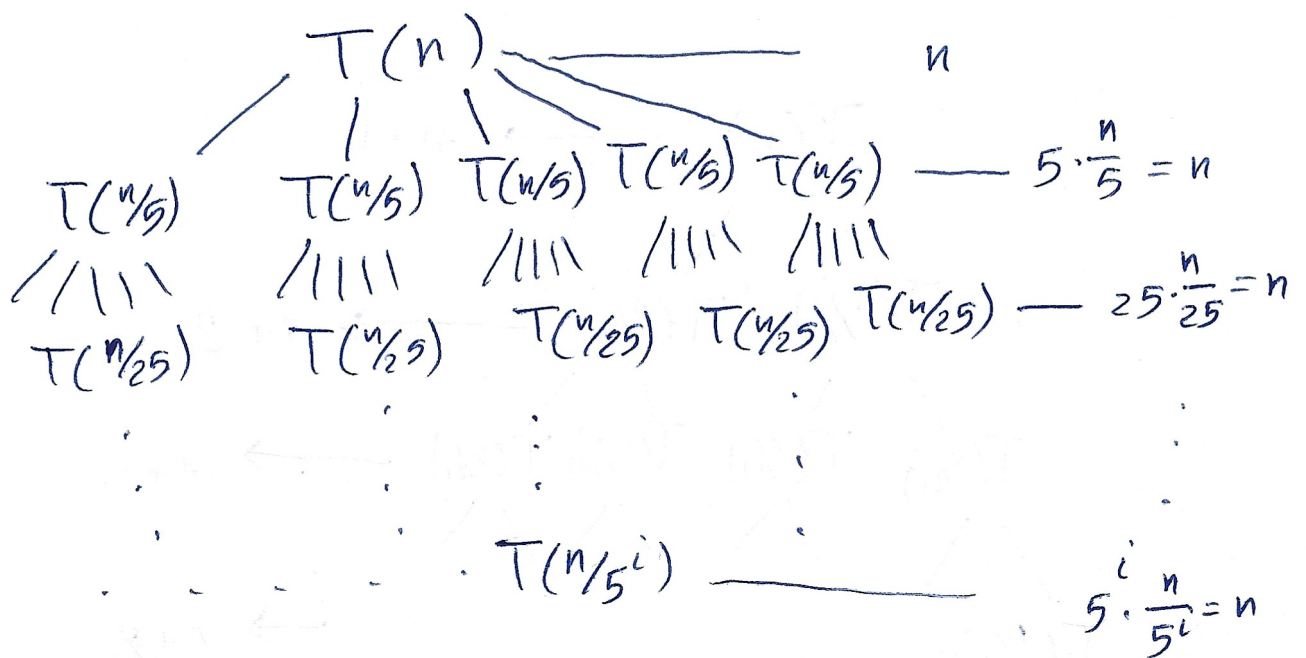
\therefore The sum of terms: $(n+1) + (n+2) + \dots + (n+2^i) = \sum_{i=0}^{\log_2(n)} (n+2^i)$

$$\Rightarrow \sum_{i=0}^{\log_2(n)} (n+2^i) = \sum_{i=0}^{\log_2(n)} n + \sum_{i=0}^{\log_2(n)} 2^i = n(\log_2(n) + 1) + 2^{\log_2(n)+1} - 1$$

Let $n = 2^x$, then: $T(2^x) = 2^x(x+2) - 1$

So, what is the complexity?

$$T(n) = 5T(n/5) + n, \quad T(1) = 1$$



Stopping case: $\frac{n}{5^i} = 1 \Rightarrow i = \log_5(n)$

Sum of the terms: $n + n + n + \dots + n = \sum_{i=0}^{\log_5(n)} n = n(\log_5(n) + 1)$

let $n = 5^x$, then: $T(5^x) = 5^x(x+1)$

Complexity?

```

1: //Name: Ghazi Najeeb AL-Abbar
2: //ID: 2181148914
3: //Problems: 3 - 4 - 5 - 6
4: //problem-3.cpp
5:
6: //Let n be the number of nodes in the binary tree
7:
8: #include <iostream>
9: #include <queue>
10: using namespace std;
11:
12: //Node class
13: struct Node{
14:
15:     //Constructor
16:     //Makes every child of a node point to null
17:     //Assigns the user input as data
18:     Node(int Data): data(Data)
19:     {
20:         this->right = nullptr;
21:         this->left = nullptr;
22:     }
23:
24:     Node* right;
25:     Node* left;
26:     int data;
27: };
28:
29: //Binary Search Tree Class
30: class BST{
31:
32: public:
33:
34:     //Constructor
35:     //Makes the root point to null when instantiating a new BST object
36:     BST()
37:     {
38:         this->root = nullptr;
39:     }
40:
41: //Problem 3
42: /*****
43:
44:     //Returns true if the input data is found within the tree
45:     //Time Complexity: O(Log(n)) because searching goes through the nodes by cutting down
46:     bool Search(int key){
47:
48:         Node* temp = root; //temp node to hold the root (copy of the root, not reference)
49:
50:         return searchRec(key, temp); //Calls searchRec to return true or false
51:     }
52:
53:     //Adds a node with user input data to the tree
54:     //Time Complexity: O(Log(n)) because going through the nodes means cutting down the L
55:     void add(int key){addRec(key, root);} //Calls addRec to add a node to the tree. The root

```

```

56:
57: //Deletes a node from the tree
58: //Time Complexity:  $O(\log(n))$  because going through the nodes means cutting down the L
59: void Delete(int key){deleteRec(root, key);} //Calls deleteRec to remove a node from the t
60:
61: //Prints the tree in order
62: //Time Complexity:  $O(n)$  since it goes through all the whole tree
63: void printInorder(){
64:
65:     Node* temp = root; //temp node to hold the root (copy of the root, not reference)
66:
67:     Inorder(temp); //Calls the Inorder function to print the tree in order
68:
69:     cout << endl;
70: }
71:
72: //Prints the tree post order
73: //Time Complexity:  $O(n)$  since it goes through all the whole tree
74: void printPostorder(){
75:
76:     Node* temp = root; //temp node to hold the root (copy of the root, not reference)
77:
78:     Postorder(temp); //Calls the Postorder function to print the tree post order
79:
80:     cout << endl;
81: }
82:
83: //Prints the tree pre-order
84: //Time Complexity:  $O(n)$  since it goes through all the whole tree
85: void printPreorder(){
86:
87:     Node* temp = root; //temp node to hold the root (copy of the root, not reference)
88:
89:     Preorder(temp); //Calls the Preorder function to print the tree pre-order
90:
91:     cout << endl;
92: }
93:
94: /*****
95:
96: //problem 4
97: //Counts and returns the number of Leaves in the binary tree
98: //Time Complexity:  $O(n)$  since it goes through all the whole tree
99: int LeafCount(){
100:
101:     int Count = 0; //Counter that counts the number of Leaves
102:
103:     Node* temp = root; //temp node to hold the root (copy of the root, not reference)
104:
105:     LeafCountRec(temp, Count); //Calls the function LeafCountRec to count the number of L
106:
107:     return Count; //Returns the number of Leaves
108: }
109:
110: //problem 5

```

111: //Finds and returns the kth smallest element from the binary tree.
 112: //Time Complexity: O(n) since it goes through all the whole tree
 113: int KthSmallest(int k){
 114:
 115: //Checks if the root is null. If it was, the function would end and returns nothing
 116: if (root == nullptr)
 117: cout << "The tree is Empty!\n";
 118:
 119: //The case where the tree is not empty.
 120: else{
 121:
 122: //Queue to hold all the data from the tree in (in order) sequence
 123: queue<int> Q; //The data is positioned from the smallest to the largest
 124: Node* temp = root; //temp node to hold the root (copy of the root, not reference)
 125:
 126: traverse(temp, Q); //Calls the traverse function to store the data in the queue.
 127:
 128: //goes through the queue and pops the front k - 1 times
 129: for (int i = 1; i < k; i++)
 130: Q.pop();
 131:
 132: return Q.front(); //Returns the front (the kth element)
 133: }
 134: }
 135:
 136: //problem 6
 137: //Checks if the binary tree is complete. Returns true if it is. Otherwise, returns fa
 138: //Time Complexity: O(n) since it goes through all the whole tree
 139: bool isComplete(){
 140:
 141: //If the tree is empty, then by definition, it is complete. Returns true
 142: if (root == nullptr)
 143: return true;
 144:
 145: bool isNotComplete = false; //Boolean value that checks if the tree is empty or not.
 146:
 147: queue<Node*> Q; //Queue to store the nodes in descending order
 148: Q.push(root); //enqueues the root
 149:
 150: //Goes through the tree and checks whether it is complete or not
 151: while (!Q.empty()){
 152:
 153: Node* temp = Q.front(); //Temporary node to hold the front of the queue
 154: Q.pop(); //The queue is popped (handles the descending movement across the tree)
 155:
 156: /* Since isNotComplete starts out as false, then the queue just enqueues the left
 157: If it happens to find a null pointer between two non-null pointing nod
 158: isNotComplete switches to true and the function returns returns false.
 159:
 160: If the queue reached a null pointer and no other non-null pointing nod
 161: */
 162:
 163: //checks whether the temp node is null
 164: if (temp == nullptr)
 165: isNotComplete = true; //isNotComplete is switched to true


166:
 167: *//if temp is not empty*
 168: else{
 169:
 170: *//If isNotComplete is true while temp is not empty, then the tree is not comp*
 171: if (isNotComplete)
 172: return false;
 173:
 174: Q.push(temp->left); *//enqueues the left child first*
 175: Q.push(temp->right); *//enqueues the second child next*
 176: }
 177: }
 178:
 179: *//If the end of the loop has been reached, then the tree is a complete tree. Returns*
 180: return true;
 181: }
 182:
 183: private: *//Encapsulated members and functions*
 184:
 185: Node* root; *//Holds the root node for the tree*
 186:
 187: *//adds a node with data equal to key to the tree by searching for its destination rec*
 188: *//Time Complexity: O(Log(n)) because going through the nodes means cutting down the l*
 189: void addRec(int key, Node*& node){
 190:
 191: *//Checks if the current node points to null*
 192: if (node == nullptr){
 193:
 194: Node* NewNode = new Node(key); *//creates new node with data equal to key*
 195:
 196: node = NewNode; *//The current node is assigned to the new node*
 197:
 198: cout << "Node with data " << key << " is added!\n";
 199:
 200: return; *//Leaves the function*
 201: }
 202:
 203: *//If the current node data is greater than or equal to key*
 204: if (node->data >= key)
 205: return addRec(key, node->left); *//Moves to the left node recursively*
 206:
 207: *//If the current node data is less than key*
 208: else
 209: return addRec(key, node->right); *//Moves to the right node recursively*
 210:
 211: }
 212:
 213: *//Checks if key is a data for a node within the tree. Returns true if the node exists*
 214: *//Time Complexity: O(Log(n)) because searching goes through the nodes by cutting down*
 215: bool searchRec(int key, Node* node){
 216:
 217: *//if the current node is null, then the node does not exist. Returns false*
 218: if (node == nullptr)
 219: return false;
 220:

P3
 Q16

```

221:         //If the current node is equal to the key value, then it returns true
222:         if (key == node->data)
223:             return true;
224:
225:         //If the current node data is greater than or equal to the key value
226:         if ( node->data >= key)
227:             return searchRec(key, node->left); //Moves to the left child recursively
228:
229:         //If the current node data is less than the key value ✓
230:         else
231:             return searchRec(key, node->right); //Moves to the right child recursively
232:     }
233:
234:     //Prints the tree in order
235:     //Time Complexity: O(n) Since it goes through the whole tree
236:     void Inorder(Node* node){
237:
238:         //if the current node is pointing to null
239:         if (node == nullptr)
240:             return; //Leaves function or recursive call ✓
241:
242:         Inorder(node->left); //recursively move to the left child
243:
244:         cout << node->data << " "; //prints the data of the current node
245:
246:         Inorder(node->right); //recursively move to the right child
247:     }
248:
249:     //Prints the tree in post order
250:     //Time Complexity: O(n) Since it goes through the whole tree
251:     void Postorder(Node* node){
252:
253:         //if the current node is pointing to null
254:         if (node == nullptr)
255:             return; //Leaves function or recursive call ✓
256:
257:         Postorder(node->left); //recursively move to the left child
258:
259:         Postorder(node->right); //recursively move to the right child
260:
261:         cout << node->data << " "; //prints the data of the current node
262:     }
263:
264:     //Prints the tree in pre order
265:     //Time Complexity: O(n) Since it goes through the whole tree
266:     void Preorder(Node* node){
267:
268:         //if the current node is pointing to null
269:         if (node == nullptr)
270:             return; //Leaves function or recursive call
271:
272:         cout << node->data << " "; //prints the data of the current node
273:
274:         Preorder(node->left); //recursively move to the left child
275:



```



```

276:     Preorder(node->right); //recursivly move to the right child
277: }
278:
279: //Finds the node data with the smallest value after a certain node
280: //Time Complexity: O(Log(n)) since it goes left using a fraction of n steps
281: int MinValue(Node* node){
282:
283:     Node* temp = node; //temp node to hold the root (copy of the root, not reference)
284:
285:     //Moves Left until it reaches the node that points to null
286:     while (temp->left != nullptr)
287:         temp = temp->left; //temp is temp's left child
288:
289:     return temp->data; //Returns the node data of the far left child
290: }
291:
292: //Deletes a node from the tree
293: //Time Complexity: O(Log(n)) because going through the nodes means cutting down the L
294: void deleteRec(Node*& node, int key){
295:
296:     //if the node with data equal to key is not found
297:     if (node == nullptr){
298:
299:         cout << "No such Node exists with data " << key << endl; //prints message
300:
301:         return; //Leaves function
302:     }
303:
304:     //If the current node data is equal to the key value, then the deletion process begin
305:     if (node->data == key){
306:
307:         //If the current node has no children
308:         if (node->left == nullptr && node->right == nullptr){
309:
310:             cout << "node with data " << key << " succesfully deleted\n";
311:
312:             Node* temp = node; //Stores the current node address in temp
313:             delete temp; //deletes temp (frees up the memory)
314:
315:             node = nullptr; //current node points to null
316:
317:             return; //Leaves function
318:         }
319:
320:
321:         //If the current node only has a left child
322:         if (node->left != nullptr && node->right == nullptr){
323:
324:             cout << "node with data " << key << " succesfully deleted\n";
325:
326:             Node* temp = node->left; //stores the current node's left child's address in
327:
328:             node = node->left; //Current is assigned to its left child
329:
330:             delete temp; //deletes temp (frees up the memory)

```

```

331:
332:         return; //Leaves the function
333:     }
334:
335:     //If the current node only has a right child
336:     if (node->left == nullptr && node->right != nullptr){
337:
338:         cout << "node with data " << key << " succesfully deleted\n";
339:
340:         Node* temp = node->right; //stores the current node's right child's address i
341:
342:         node = node->right; //Current is assigned to its right child ✓
343:
344:         delete temp; //Current is assigned to its left child ✓
345:
346:         return; //Leaves the function
347:     }
348:
349:
350:     //If the node has both a right and a left child
351:     if (node->left != nullptr && node->right != nullptr){
352:
353:         node->data = MinValue(node->right); //Current node data is the smallest node
354:
355:         deleteRec(node->right, MinValue(node->right)); //deleteRec is called to remov
356:
357:         cout << "node with data " << key << " succesfully deleted\n"; ✓
358:
359:         return; //Leaves function ✓
360:     }
361: }
362:
363: //If the node data is greater than the key value
364: if (node->data > key)
365:     return deleteRec(node->left, key); //Recursivly move to the left child
366:
367: //If the node data is less than the key value
368: if (node->data < key)
369:     return deleteRec(node->right, key); //Recursivly move to the right child
370: }
371:
372: //problem 4
373: //Counts and returns the number of leaves in the binary tree
374: //Time Complexity: O(n) since it goes through all the whole tree
375: void LeafCountRec(Node* node, int& Count){
376:
377:     //if the current node points to null
378:     if (node == nullptr)
379:         return; //Leaves the function or recursive call
380:
381:     //If the node's left child and right child are both empty, then it is a leaf
382:     if (node->right == nullptr && node->left == nullptr){
383:
384:         Count += 1; //increment the counter by one ✓
385:

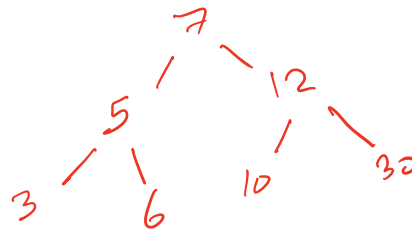
```

py
9/1/24

```

386:         return; //leaves the function or recursive call
387:     }
388:
389:     //if the current node only has a right child
390:     if (node->right != nullptr && node->left == nullptr)
391:         LeafCountRec(node->right, Count); //Recusrivly move to the right child
392:
393:     //if the current node only has a left child
394:     if (node->right == nullptr && node->left != nullptr)
395:         LeafCountRec(node->left, Count); //Recusrivly move to the right child
396:
397:     //if the current node has both left and right children
398:     if (node->right != nullptr && node->left != nullptr){
399:
400:         LeafCountRec(node->left, Count); //Recusrivly move to the left child
401:
402:         LeafCountRec(node->right, Count); //Recusrivly move to the right child
403:
404:         return; //leaves the function or recursive call
405:     }
406: }
407:
408: //Function that traverses through the tree in order, and enqueues all node data into
409: //By the end of the function, the queue has all the tree's node data in ascending ord
410: //Time Complexity: O(n) Since it goes through the whole tree
411: void traverse(Node* node, queue<int>& Q){
412:
413:     //If the current node is empty
414:     if (node == nullptr)
415:         return; //leaves the function or recursive call
416:
417:     traverse(node->left, Q); //Recusrivly move to the left child
418:     Q.push(node->data); //enqueues the current node data into the queue
419:     traverse(node->right, Q); //Recusrivly move to the right child
420: }
421: };
422:
423: //Beginning of program
424: int main(){
425:
426:     BST tree;
427:
428:     tree.add(7);
429:     tree.add(12);
430:     tree.add(5);
431:     tree.add(3);
432:     tree.add(6);
433:     tree.add(10);
434:     tree.add(30);
435:
436:     cout << "Is the there a node with data 12 in the tree? The answer is: ";
437:
438:     if (tree.Search(12))
439:         cout << "True\n";
440:     else

```



```

441:         cout << "False\n";
442:
443:     cout << "Is the there a node with data 100 in the tree? The answer is: ";
444:
445:     if (tree.Search(100))
446:         cout << "True\n";
447:     else
448:         cout << "False\n";
449:
450:     cout << "Printing in order: ";
451:     tree.printInorder();
452:
453:     cout << "Printing post order: ";
454:     tree.printPostorder();
455:
456:     cout << "Printing pre-order: ";
457:     tree.printPreorder();
458:
459:     cout << "The number of leaves is " << tree.LeafCount() << endl;
460:
461:     cout << "The 4th smallest element is " << tree.KthSmallest(4) << endl;
462:
463:     cout << "Is the tree complete? The answer is: ";
464:
465:     if (tree.isComplete())
466:         cout << "True\n";
467:     else
468:         cout << "False\n";
469:
470:     tree.Delete(7);
471:
472:     cout << "Is the there a node with data 7 in the tree? The answer is: ";
473:
474:     if (tree.Search(7))
475:         cout << "True\n";
476:     else
477:         cout << "False\n";
478:
479:     cout << "Is the there a node with data 30 in the tree? The answer is: ";
480:
481:     if (tree.Search(30))
482:         cout << "True\n";
483:     else
484:         cout << "False\n";
485:
486:     cout << "Printing in order: ";
487:     tree.printInorder();
488:
489:     cout << "Printing post order: ";
490:     tree.printPostorder();
491:
492:     cout << "Printing pre-order: ";
493:     tree.printPreorder();
494:
495:     cout << "The number of leaves is " << tree.LeafCount() << endl;

```

```
496:
497:     cout << "The 4th smallest element is " << tree.KthSmallest(4) << endl;
498:
499:     cout << "Is the tree complete? The answer is: ";
500:     if (tree.isComplete())
501:         cout << "True\n";
502:     else
503:         cout << "False\n";
504:
505:     return 0;
506: }
507: //End of program
```