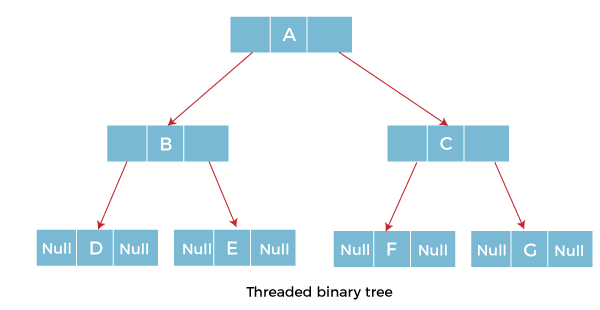
**What do you mean by Threaded Binary Tree?**

In the linked representation of binary trees, more than one half of the link fields contain NULL values which results in wastage of storage space. If a binary tree consists of **n** nodes then **n+1** link fields contain NULL values. So in order to effectively manage the space, a method was devised by Perlis and Thornton in which the NULL links are replaced with special links known as threads. Such binary trees with threads are known as **threaded binary trees**. Each node in a threaded binary tree either contains a link to its child node or thread to other nodes in the tree.

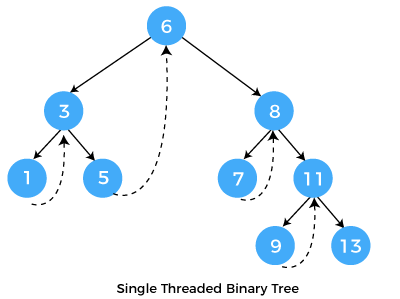


### **Types of Threaded Binary Tree**

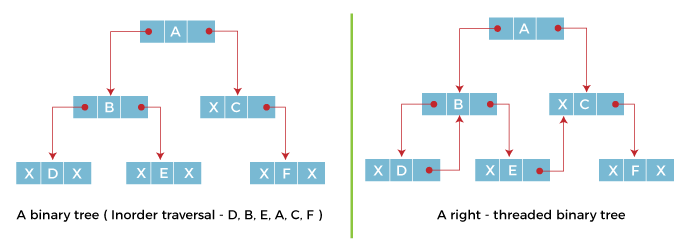
There are two types of threaded Binary Tree:

* One-way threaded Binary Tree
* Two-way threaded Binary Tree

**One-way threaded Binary trees:**

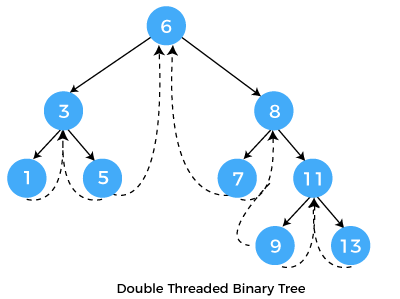


In one-way threaded binary trees, a thread will appear either in the right or left link field of a node. If it appears in the right link field of a node then it will point to the next node that will appear on performing in order traversal. Such trees are called **Right threaded binary trees**. If thread appears in the left field of a node then it will point to the nodes inorder predecessor. Such trees are called **Left threaded binary trees.** Left threaded binary trees are used less often as they don't yield the last advantages of right threaded binary trees. In one-way threaded binary trees, the right link field of last node and left link field of first node contains a NULL. In order to distinguish threads from normal links they are represented by dotted lines.

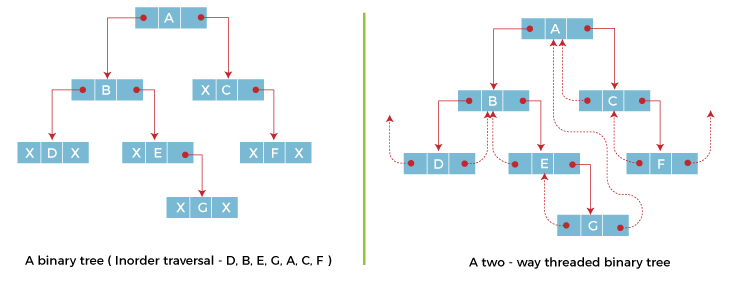


The above figure shows the inorder traversal of this binary tree yields D, B, E, A, C, F. When this tree is represented as a right threaded binary tree, the right link field of leaf node D which contains a NULL value is replaced with a thread that points to node B which is the inorder successor of a node D. In the same way other nodes containing values in the right link field will contain NULL value.

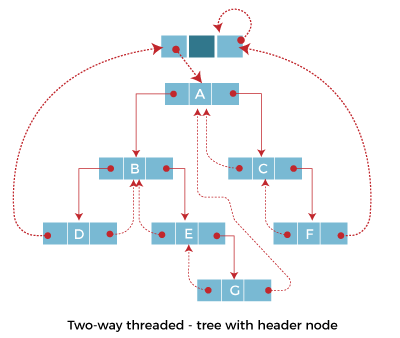
**Two-way threaded Binary Trees:**



In two-way threaded Binary trees, the right link field of a node containing NULL values is replaced by a thread that points to nodes inorder successor and left field of a node containing NULL values is replaced by a thread that points to nodes inorder predecessor.



The above figure shows the inorder traversal of this binary tree yields D, B, E, G, A, C, F. If we consider the two-way threaded Binary tree, the node E whose left field contains NULL is replaced by a thread pointing to its inorder predecessor i.e. node B. Similarly, for node G whose right and left linked fields contain NULL values are replaced by threads such that right link field points to its inorder successor and left link field points to its inorder predecessor. In the same way, other nodes containing NULL values in their link fields are filled with threads.



In the above figure of two-way threaded Binary tree, we noticed that no left thread is possible for the first node and no right thread is possible for the last node. This is because they don't have any inorder predecessor and successor respectively. This is indicated by threads pointing nowhere. So in order to maintain the uniformity of threads, we maintain a special node called the **header node**. The header node does not contain any data part and its left link field points to the root node and its right link field points to itself. If this header node is included in the two-way threaded Binary tree then this node becomes the inorder predecessor of the first node and inorder successor of the last node. Now threads of left link fields of the first node and right link fields of the last node will point to the header node.

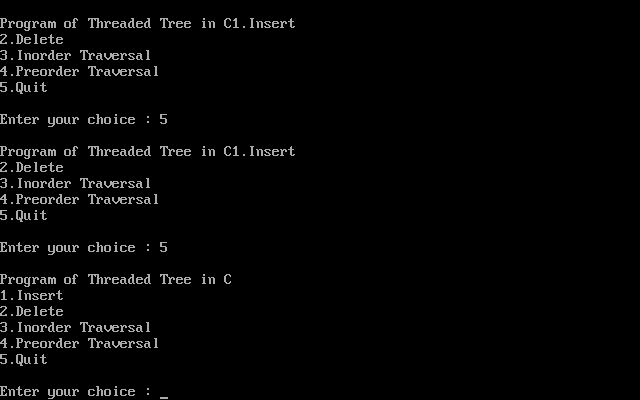
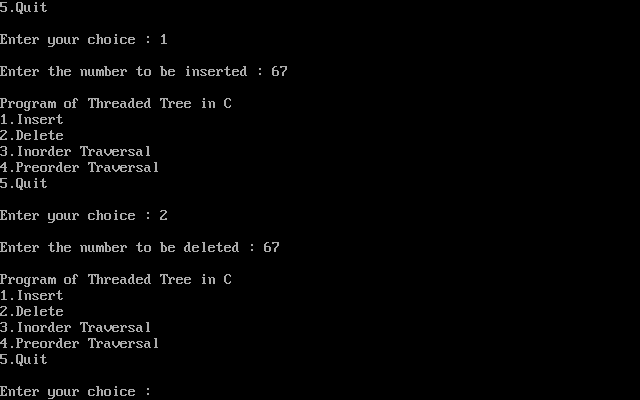
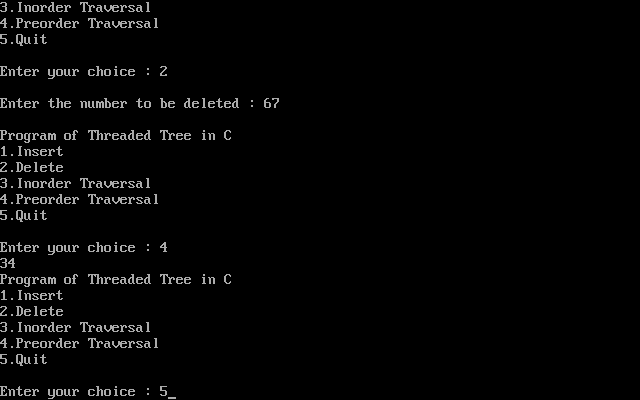
## **Algorithm for Inorder Traversal of Threaded Binary Tree:**

1. ALgorithm Inorder(I)
2. {
3. ThreadedTreeNode \*Header;
4. Header=I;
5. **while**(1)
6. {
7. I=fnFindInorder\_Successor(H);
8. **if**(I==Header)
9. **return**;
10. **else**
11. print(I->info);
12. }
13. }
14. In the above algorithm, we have discussed the Inorder traversal of Threaded binary trees.
15. Pseudocode of Threaded Tree in C:
16. #include <stdio.h>
17. #include <stdlib.h>
18. typedef **enum** {**false**,**true**} **boolean**;
19. struct node \*in\_succ(struct node \*p);
20. struct node \*in\_pred(struct node \*p);
21. struct node \*insert(struct node \*root, **int** ikey);
22. struct node \*del(struct node \*root, **int** dkey);
23. struct node \*case\_a(struct node \*root, struct node \*par,struct node \*ptr);
24. struct node \*case\_b(struct node \*root,struct node \*par,struct node \*ptr);
25. struct node \*case\_c(struct node \*root, struct node \*par,struct node \*ptr);
26. **void** inorder( struct node \*root);
27. **void** preorder( struct node \*root);
29. struct node
30. {
31. struct node \*left;
32. **boolean** lthread;
33. **int** info;
34. **boolean** rthread;
35. struct node \*right;
36. };
37. **int** main( )
38. {
39. clrscr();
40. **int** choice,num;
41. struct node \*root=NULL;
42. **while**(1)
43. {
44. printf("\Program of Threaded Tree in C\n");
45. printf("1.Insert\n");
46. printf("2.Delete\n");
47. printf("3.Inorder Traversal\n");
48. printf("4.Preorder Traversal\n");
49. printf("5.Quit\n");
50. printf("\nEnter your choice : ");
51. scanf("%d",&choice);
52. **switch**(choice)
53. {
54. **case** 1:
55. printf("\nEnter the number to be inserted : ");
56. scanf("%d",&num);
57. root = insert(root,num);
58. **break**;
59. **case** 2:
60. printf("\nEnter the number to be deleted : ");
61. scanf("%d",&num);
62. root = del(root,num);
63. **break**;
64. **case** 3:
65. inorder(root);
66. **break**;
67. **case** 4:
68. preorder(root);
69. **break**;
70. **case** 5:
71. exit(1);
73. **default**:
74. printf("\nWrong choice\n");
75. }
76. }
77. **return** 0;
78. }
79. struct node \*insert(struct node \*root, **int** ikey)
80. {
81. struct node \*tmp,\*par,\*ptr;
82. **int** found=0;
83. ptr = root;
84. par = NULL;
85. **while**( ptr!=NULL )
86. {
87. **if**( ikey == ptr->info)
88. {
89. found =1;
90. **break**;
91. }
92. par = ptr;
93. **if**(ikey < ptr->info)
94. {
95. **if**(ptr->lthread == **false**)
96. ptr = ptr->left;
97. **else**
98. **break**;
99. }
100. **else**
101. {
102. **if**(ptr->rthread == **false**)
103. ptr = ptr->right;
104. **else**
105. **break**;
106. }
107. }
108. **if**(found)
109. printf("\nDuplicate key");
110. **else**
111. {
112. tmp=(struct node \*)malloc(sizeof(struct node));
113. tmp->info=ikey;
114. tmp->lthread = **true**;
115. tmp->rthread = **true**;
116. **if**(par==NULL)
117. {
118. root=tmp;
119. tmp->left=NULL;
120. tmp->right=NULL;
121. }
122. **else** **if**( ikey < par->info )
123. {
124. tmp->left=par->left;
125. tmp->right=par;
126. par->lthread=**false**;
127. par->left=tmp;
128. }
129. **else**
130. {
131. tmp->left=par;
132. tmp->right=par->right;
133. par->rthread=**false**;
134. par->right=tmp;
135. }
136. }
137. **return** root;
138. }
140. struct node \*del(struct node \*root, **int** dkey)
141. {
142. struct node \*par,\*ptr;
143. **int** found=0;
144. ptr = root;
145. par = NULL;
146. **while**( ptr!=NULL)
147. {
148. **if**( dkey == ptr->info)
149. {
150. found =1;
151. **break**;
152. }
153. par = ptr;
154. **if**(dkey < ptr->info)
155. {
156. **if**(ptr->lthread == **false**)
157. ptr = ptr->left;
158. **else**
159. **break**;
160. }
161. **else**
162. {
163. **if**(ptr->rthread == **false**)
164. ptr = ptr->right;
165. **else**
166. **break**;
167. }
168. }
169. **if**(found==0)
170. printf("\ndkey not present in tree");
171. **else** **if**(ptr->lthread==**false** && ptr->rthread==**false**)/\*2 children\*/
172. root = case\_c(root,par,ptr);
173. **else** **if**(ptr->lthread==**false** )
174. root = case\_b(root, par,ptr);
175. **else** **if**(ptr->rthread==**false**)
176. root = case\_b(root, par,ptr);
177. **else**
178. root = case\_a(root,par,ptr);
179. **return** root;
180. }
182. struct node \*case\_a(struct node \*root, struct node \*par,struct node \*ptr )
183. {
184. **if**(par==NULL)
185. root=NULL;
186. **else** **if**(ptr==par->left)
187. {
188. par->lthread=**true**;
189. par->left=ptr->left;
190. }
191. **else**
192. {
193. par->rthread=**true**;
194. par->right=ptr->right;
195. }
196. free(ptr);
197. **return** root;
198. }
199. struct node \*case\_b(struct node \*root,struct node \*par,struct node \*ptr)
200. {
201. struct node \*child,\*s,\*p;
202. **if**(ptr->lthread==**false**)
203. child=ptr->left;
204. **else**
205. child=ptr->right;
206. **if**(par==NULL )
207. root=child;
208. **else** **if**( ptr==par->left)
209. par->left=child;
210. **else**
211. par->right=child;
212. s=in\_succ(ptr);
213. p=in\_pred(ptr);
214. **if**(ptr->lthread==**false**)
215. p->right=s;
216. **else**
217. {
218. **if**(ptr->rthread==**false**)
219. s->left=p;
220. }
221. free(ptr);
222. **return** root;
223. }
224. struct node \*case\_c(struct node \*root, struct node \*par,struct node \*ptr)
225. {
226. struct node \*succ,\*parsucc;
227. parsucc = ptr;
228. succ = ptr->right;
229. **while**(succ->left!=NULL)
230. {
231. parsucc = succ;
232. succ = succ->left;
233. }
234. ptr->info = succ->info;
235. **if**(succ->lthread==**true** && succ->rthread==**true**)
236. root = case\_a(root, parsucc,succ);
237. **else**
238. root = case\_b(root, parsucc,succ);
239. **return** root;
240. }
241. struct node \*in\_succ(struct node \*ptr)
242. {
243. **if**(ptr->rthread==**true**)
244. **return** ptr->right;
245. **else**
246. {
247. ptr=ptr->right;
248. **while**(ptr->lthread==**false**)
249. ptr=ptr->left;
250. **return** ptr;
251. }
252. }
253. struct node \*in\_pred(struct node \*ptr)
254. {
255. **if**(ptr->lthread==**true**)
256. **return** ptr->left;
257. **else**
258. {
259. ptr=ptr->left;
260. **while**(ptr->rthread==**false**)
261. ptr=ptr->right;
262. **return** ptr;
263. }
264. }
265. **void** inorder( struct node \*root)
266. {
267. struct node \*ptr;
268. **if**(root == NULL )
269. {
270. printf("Tree is empty");
271. **return**;
272. }
273. ptr=root;
274. **while**(ptr->lthread==**false**)
275. ptr=ptr->left;
276. **while**( ptr!=NULL )
277. {
278. printf("%d ",ptr->info);
279. ptr=in\_succ(ptr);
280. }
281. }
282. **void** preorder(struct node \*root )
283. {
284. struct node \*ptr;
285. **if**(root==NULL)
286. {
287. printf("Tree is empty");
288. **return**;
289. }
290. ptr=root;
291. **while**(ptr!=NULL)
292. {
293. printf("%d ",ptr->info);
294. **if**(ptr->lthread==**false**)
295. ptr=ptr->left;
296. **else** **if**(ptr->rthread==**false**)
297. ptr=ptr->right;
298. **else**
299. {
300. **while**(ptr!=NULL && ptr->rthread==**true**)
301. ptr=ptr->right;
302. **if**(ptr!=NULL)
303. ptr=ptr->right;
304. }
305. }
306. }

**Explanation:**

In the above example, we have created a threaded binary tree for various operations.

**Output:**

**Advantages of Threaded Binary Tree:**

* In threaded binary tree, linear and fast traversal of nodes in the tree so there is no requirement of stack. If the stack is used then it consumes a lot of memory and time.
* It is more general as one can efficiently determine the successor and predecessor of any node by simply following the thread and links. It almost behaves like a circular linked list.

**Disadvantages of Threaded Binary Tree:**

* When implemented, the threaded binary tree needs to maintain the extra information for each node to indicate whether the link field of each node points to an ordinary node or the node's successor and predecessor.
* Insertion into and deletion from a threaded binary tree are more time consuming since both threads and ordinary links need to be maintained.