

ECE 264 Advanced C Programming

Contents

1 Binary Search Tree

1

1 Binary Search Tree

In this example, we will see how to create a *binary tree*, in particular a *binary search tree*.

```
/* btreenode.h */
#ifndef BTREENODE_H
#define BTREENODE_H
typedef struct btreenode
{
    struct btreenode * bt_left;
    struct btreenode * bt_right;
    struct btreenode * bt_parent;
    int bt_value;
} Node;

Node * BTree_copy(Node * n);
void BTree_assign(Node * * n1, Node * n2);
void BTree_insert(Node * * n, int v);
int BTree_delete(Node * * btree, int v);
void BTree_printInOrder(Node * n, int level);
void BTree_printPreOrder(Node * n, int level);
void BTree_printPostOrder(Node * n, int level);
void BTree_destruct(Node * n);
int BTree_search(Node * btree, int v);
#endif

/* btreenode.c */
#include "btreenode.h"
#include <stdio.h>
#include <stdlib.h>
static Node * Node_construct(int v)
{
    Node * n = malloc(sizeof(Node));
```

```

    n -> bt_value = v;
    n -> bt_right = 0;
    n -> bt_left = 0;
    n -> bt_parent = 0;
    return n;
}

void BTree_insert(Node * * n, int v)
{
    if ((*n) == 0) /* first node */
    {
        Node * p = Node_construct(v);
        *n = p;
        return;
    }
    if (((*n) -> bt_value) == v)
    { return; } /* already in the tree */
    if (((*n) -> bt_value) > v)
    {
        if (((*n) -> bt_left) == 0)
        {
            Node * p = Node_construct(v);
            (*n) -> bt_left = p;
            p -> bt_parent = (*n);
        }
        else
        {
            BTree_insert(& ((*n) -> bt_left), v);
        }
    }
    else
    {
        if (((*n) -> bt_right) == 0)
        {
            Node * p = Node_construct(v);
            (*n) -> bt_right = p;
            p -> bt_parent = (*n);
        }
        else
        {
            BTree_insert(& ((*n) -> bt_right), v);
        }
    }
}

```

```

    }
}

static void Node_print(Node * n, int level)
{
    int indcnt; /* indentation */
    for (indcnt = 0; indcnt < level; indcnt++)
        { printf("  "); }
    printf("%d\n", n -> bt_value);
}

void BTree_printInOrder(Node * n, int level)
{
    if (n == 0) { return; }
    BTree_printInOrder(n -> bt_left, level + 1);
    Node_print(n, level);
    BTree_printInOrder(n -> bt_right, level + 1);
}

void BTree_printPreOrder(Node * n, int level)
{
    if (n == 0) { return; }
    Node_print(n, level);
    BTree_printPreOrder(n -> bt_left, level + 1);
    BTree_printPreOrder(n -> bt_right, level + 1);
}

void BTree_printPostOrder(Node * n, int level)
{
    if (n == 0) { return; }
    BTree_printPostOrder(n -> bt_left, level + 1);
    BTree_printPostOrder(n -> bt_right, level + 1);
    Node_print(n, level);
}

int BTree_search(Node * btree, int v)
{
    if (btree == 0) { return 0; }
    if ((btree -> bt_value) == v) { return 1; }
    if ((btree -> bt_value) > v)

```

```

    { return BTree_search(btree -> bt_left, v); }
    return BTree_search(btree -> bt_right, v);
}

/* main.c */
#include "btreenode.h"
#include <stdio.h>
void testFunc()
{
    Node * btree1 = 0;
    int data [] = {1, 100, 3, 64, 5, -6, 999, 4, -85, 7};
    int numElem = sizeof(data) / sizeof(int);
    int cnt;
    for (cnt = 0; cnt < numElem; cnt ++)
    {
        printf("inserting %d\n", data[cnt]);
        BTree_insert(& btree1, data[cnt]);
        BTree_printInOrder(btree1, 1);
    }
    printf("search %d = %d\n", 6, BTree_search(btree1, 6));
    BTree_printInOrder(btree1, 1);
    printf("search %d = %d\n", 5, BTree_search(btree1, 5));
    BTree_printInOrder(btree1, 1);

    printf("delete %d = %d\n", 3, BTree_delete(& btree1, 3));
    BTree_printInOrder(btree1, 1);

    printf("delete %d = %d\n", 13, BTree_delete(& btree1, 13));
    BTree_printInOrder(btree1, 1);

    printf("delete %d = %d\n", 10, BTree_delete(& btree1, 10));
    BTree_printInOrder(btree1, 1);

    printf("delete %d = %d\n", 19, BTree_delete(& btree1, 19));
    BTree_printInOrder(btree1, 1);

    printf("delete %d = %d\n", -85, BTree_delete(& btree1, -85));
    BTree_printInOrder(btree1, 1);

    printf("delete %d = %d\n", 999, BTree_delete(& btree1, 999));
    BTree_printInOrder(btree1, 1);

    printf("delete %d = %d\n", -6, BTree_delete(& btree1, -6));
}

```

```

BTree_printInOrder(btrees1, 1);

printf("delete %d = %d\n", 100, BTree_delete(& btrees1, 100));
BTree_printInOrder(btrees1, 1);

for (cnt = 0; cnt < numElem; cnt ++)
{
    printf("inserting %d\n", data[cnt] * cnt);
    BTree_insert(& btrees1, data[cnt] * cnt);
    BTree_printInOrder(btrees1, 1);
}
printf("search %d = %d\n", 6, BTree_search(btrees1, 6));
BTree_destruct(btrees1);
}

int main(int argc, char * argv[])
{
    testFunc();
    return 0;
}
/*
inserting 1
    1
inserting 100
    1
        100
inserting 3
    1
        3
            100
inserting 64
    1
        3
            64
                100
inserting 5
    1
        3
            5
                64
                    100
inserting -6

```

```
    -6
  1
    3
      5
        64
          100
inserting 999
```

```
    -6
  1
    3
      5
        64
          100
          999
```

```
inserting 4
    -6
  1
    3
      4
        5
          64
            100
              999
```

```
inserting -85
    -85
  1
    3
      4
        5
          64
            100
              999
```

```
inserting 7
    -85
  1
    3
      4
        5
          7
            64
```

```

    100
      999
search 6 = 0
      -85
        -6
          1
            3
              4
                5
                  7
                    64

```

```

    100
      999
search 5 = 1
      -85
        -6
          1
            3
              4
                5
                  7
                    64

```

```

    100
      999
delete 3 = 0
      -85
        -6
          1
            3
              4
                5
                  7
                    64

```

```

    100
      999
delete 13 = 0
      -85
        -6
          1
            3
              4
                5

```

```

        64      7
    100
      999
delete 10 = 0
      -85
      -6
    1
      3

```

```

    4
    5
    7

```

```

        64
    100
      999
delete 19 = 0
      -85
      -6
    1
      3

```

```

    4
    5
    7

```

```

        64
    100
      999
delete -85 = 0
      -85
      -6
    1
      3

```

```

    4
    5
    7

```

```

        64
    100
      999
delete 999 = 0
      -85
      -6
    1
      3

```



```

        4
      5
    7
  64
100
  999
delete -6 = 0
      -85
      -6
1
    3
      4
    5
  7
64
100
  999
delete 100 = 0
      -85
      -6
1
    3
      4
    5
  7
64
100
  999
inserting 0
      -85
      -6
      0
1
    3
      4
    5
  7
64
100
  999
inserting 100
      -85

```

```

    -6
      0
1     3
      4
      5
      7
      64
     100
    999
inserting 6
    -85
     -6
      0
1     3
      4
      5
      6
      7
      64
     100
    999
inserting 192
    -85
     -6
      0
1     3
      4
      5
      6
      7
      64
     100
    192
    999
inserting 20
    -85
     -6
      0
1

```

```

      3
        4
          5
            6
              7
                20
                  64
                    100
                      192
                        999
inserting -30
              -85
                -30
                  -6
                    0
                      1
                        3
                          4
                            5
                              6
                                7
                                  20
                                    64
                                      100
                                        192
                                          999
inserting 5994
              -85
                -30
                  -6
                    0
                      1
                        3
                          4
                            5
                              6
                                7
                                  20
                                    64
                                      100
                                        192
                                          999

```

```

                    5994
inserting 28
                -85
                -30
                -6
                0
1
                3
                4
                5
                6
                7
                20
                28
                64
100
                192
                999
                    5994
inserting -680
                -680
                -85
                -30
                -6
                0
1
                3
                4
                5
                6
                7
                20
                28
                64
100
                192
                999
                    5994
inserting 63
                -680
                -85
                -30

```

```

      -6
     0
    1
     3
      4
     5
      6
     7
    20
     28
      63
     64
    100
     192
    999
     5994
search 6 = 1
*/

```

Page 477 in ABoC has a figure illustrating a binary tree. Each node in the binary tree has three pointers, left, right, and parent. If a node is pointed by another node's left or right pointer, we call the former a *child* of the latter. `BTree_insert` looks complicated but the concept is straightforward. If the new value is already in the tree, do nothing. If the new value is smaller than the current value, insert the new value on the left *subtree*. Otherwise, insert the new value on the right subtree. We are using recursion here because it provides a natural way of thinking. Let's consider inserting in the left subtree. If the current node has no left subtree, this node is called a *leaf node* or *leaf* for simplicity. In this case, insert the new value as the left node of this node. If the current node has a left node, insert the new value into the left subtree. Let's trace the program by inserting a few values.

After inserting some values, we can print the tree. We have three choices:

- *pre order*: printing the value of the node first, then the left subtree, then the right subtree.
- *in order*: printing the left subtree first, then the value of the node, then the right subtree.
- *post order*: printing the left subtree first, then the right subtree, then the value of the node.

We usually visit the left subtree before the right subtree. This is a convention widely used. Consider rotating the three clockwise 90 degrees and then mirror vertically. This is our

binary tree. Moreover, if you ignore the indentations and read the values from top, you should notice that the values are **sorted**. This is the reason we call it a binary “search” tree. When we search a value, we need to go to either the left subtree or the right subtree. We do not have to check the other side. This can dramatically reduce the time spent on checking.