

ECE 264 Exam 4

02:30 April 24 - 02:00PM April 27, 2009

1 Binary Search Tree (5 points)

Consider the structure of a binary search tree. Please remember to update all pointers and release memory correctly.

```
#include <stdio.h>
#include <stdlib.h>
typedef struct btreeNode
{
    struct btreeNode * bt_left;
    struct btreeNode * bt_right;
    struct btreeNode * bt_parent;
    int bt_value;
} Node;

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;
}

Node * BTree_search(Node * n, int v)
/* return the node whose value is v */
{
    if (n == 0) { return 0; }
    if ((n -> bt_value) == v) { return n; }
    if ((n -> bt_value) > v)
        { return BTree_search(n -> bt_left, v); }
    return BTree_search(n -> bt_right, v);
}

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
    {
        BTTree_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
    {
        BTTree_insert(& ((*n) -> bt_right), v);
    }
}
}
}

```

1.1 Search (1 point)

Rewrite `BTTree_search` so that it does **not** use recursion.

1.2 Insert (2 points)

Rewrite `BTTree_insert` so that it does **not** use recursion.

1.3 Successor (2 points)

Write a function

```
Node * BTree_findSuccessor(Node * n  
    /* additional arguments if necessary */)
```

It finds the **immediate successor** of n. If n has no successor, return 0. Hint: The function must be able to handle the situation when n's successor is not its right child.

Answer:

```
#include <stdlib.h>  
#include <stdio.h>  
typedef struct btreeNode  
{  
    struct btreeNode * bt_left;  
    struct btreeNode * bt_right;  
    struct btreeNode * bt_parent;  
    int bt_value;  
} Node;  
  
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;  
}  
  
Node * BTree_search(Node * n, int v)  
/* do not use recursion */  
{  
    if (n == 0) { return 0; }  
    if ((n -> bt_value) == v) { return n; }  
    Node * cur = n;  
    while (cur != 0)  
    {  
        if ((cur -> bt_value) == v) { return cur; }  
        if ((cur -> bt_value) > v)  
        {  
            cur = cur -> bt_left;  
        }
```

```

        else
        {
            cur = cur -> bt_right;
        }
    }
    return 0;
}

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

Node * BTREE_findSuccessor(Node * n)

```

```

{
    Node * cur;
    if (n == 0)
        { return 0; }
    if ((n -> bt_right) != 0)
    {
        cur = n -> bt_right;
        while ((cur -> bt_left) != 0)
        {
            cur = cur -> bt_left;
        }
        return cur;
    }
    else
    {
        cur = n -> bt_parent;
        while ((cur != 0) && ((cur -> bt_right) == n))
        {
            n = cur;
            cur = cur -> bt_parent;
        }
        return cur;
    }
}

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

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

void BTTree_destruct(Node * n)
{
    /* to be filled */
}

```

```

}

void findSuccessor(Node * bt, int v)
{
    Node * node;
    Node * succ;
    node = BTree_search(bt, v);
    succ = BTree_findSuccessor(node);
    printf("successor of %d is ", v);
    Node_print(succ, 0);
    printf("\n");
}

int main(int argc, char * argv[])
{
    Node * btree = 0;
    int data [] = {100, 3, 8, 13, 7, 2, 53, 123, 234, 46, 98,
                   64, 5, 6, 999, 4, 85, 27, 102, 105, 146,
                   19, 33};
    int numElem = sizeof(data) / sizeof(int);
    int cnt;
    for (cnt = 0; cnt < numElem; cnt++)
    {
        BTree_insert(& btree, data[cnt]);
        /* printf("inserting %d\n", data[cnt]); */
        /* BTree_printInOrder(btree, 0); */
    }
    BTree_printInOrder(btree, 1);

    findSuccessor(btree, 5);
    findSuccessor(btree, 9);
    findSuccessor(btree, 64);
    findSuccessor(btree, -6);
    findSuccessor(btree, 100);
    findSuccessor(btree, 999);
    findSuccessor(btree, 98);
    findSuccessor(btree, 33);
    findSuccessor(btree, 7);
    BTree_destruct(btree);
    return 0;
}

```

2 Integer Partition (5 points)

Consider how to partition a positive integer n into the sums of positive integers. For example, $n = 4$ can be partitioned to

```
1 + 1 + 1 + 1
1 + 1 + 2
1 + 2 + 1
1 + 3
2 + 1 + 1
2 + 2
3 + 1
4
```

Two partitions are different (and counted **twice**) if they have at least one number different in the two *sequences*. For example, $1 + 2 + 1$ and $1 + 1 + 2$ are considered two different partitions. There are 8 different ways to partition number 4.

2.1 Counting Partitions (1 point)

Consider the following program. The program computes how many ways n can be partitioned. Fill in the code and **explain**.

```
#include <stdlib.h>
#include <stdio.h>

void partition(int n, int * counter)
{
    int i;
    if (n == 0)
    {
        (* counter)++;
    }
    for (i = 1; i <= n; i++)
    {
        /* fill in the code and explain */
    }
}

int main(int argc, char * argv[])
{
    int n = 0;
    int counter = 0;
```

```

if (argc < 2)
{
    printf("need a value\n");
    return -1;
}
n = (int)strtol(argv[1], (char **)NULL, 10);
if (n < 0)
{
    printf("need a positive value\n");
    return -1;
}
partition(n, & counter);
printf("There are %d ways to partition %d.\n", counter, n);
return 0;
}

```

2.2 Recursive Relation (2 points)

Let $f(n)$ be the number of ways to partition value n . $f(1) = 1$ since there is only one way to partition 1. $f(2) = 2$ since there are two ways to partition 2: 1 + 1 or 2. $f(3) = 4$ since there are four ways to partition 3: 1 + 1 + 1 or 1 + 2 or 2 + 1 or 3.

How many ways can we partition n ($n > 3$) so that the first value is 1? Express your answer using $f(1)$ or $f(2)$ or $f(3)$... or $f(n - 1)$ or a combination of them.

How many ways can we partition n so that the first value is 2? Express your answer using $f(1)$ or $f(2)$ or $f(3)$... or $f(n - 1)$ or a combination of them.

Explain your answers.

2.3 Recursive Relation (1 point)

How many ways can we partition n ? The first number can be 1, 2, 3, ..., $n - 1$, or n . Express $f(n)$ using $f(1)$ or $f(2)$ or $f(3)$... or $f(n - 1)$ or a combination of them.

2.4 General Formula (1 point)

Express $f(n)$ using n **without** using any of $f(1)$ or $f(2)$ or $f(3)$... or $f(n - 1)$.

Answer:

2.1

```
partition(n - i, counter);
```

The value of i is used for the first number.

2.2

If the first number is 1, there are $f(n - 1)$ ways to partition the value $n - 1$. If the first number is 2, there are $f(n - 2)$ ways to partition the value $n - 2$.

2.3

The first number can be

- If the first value is 1, there are $f(n - 1)$ ways to partition $n - 1$
- If the first value is 2, there are $f(n - 2)$ ways to partition $n - 2$
- If the first value is 3, there are $f(n - 3)$ ways to partition $n - 3$
- ...
- If the first value is $n - 1$, there is $f(1)$ way to partition 1
- If the first value is n , there is only one way

$$f(n) = f(n - 1) + f(n - 2) + \dots + f(1) + 1$$

2.4

$$f(1) = 1 \text{ so } f(n) = 2^{n-1}.$$

It is **invalid** to use the following explanation. "We know $f(n) = 2^{n-1}$; therefore, $f(n - 1) = 2^{n-2}$." You have to explain why $f(n) = 2^{n-1}$ first. It is also invalid to say " $f(n) = 2^{n-1}$ so $f(n) = f(n - 1) + f(n - 2) + \dots + f(1) + 1$." because $f(n) = 2^{n-1}$ is your hypothesis. You cannot use a hypothesis to prove itself.

Another invalid explanation is " $f(1) = 1, f(2) = 2, f(3) = 4, f(4) = 8$. Therefore, $f(n)$ must be 2^{n-1} ." There is no guarantee that $f(5)$ is 16. Even if $f(5)$ is 16, there is no guarantee that $f(6)$ is 32.