Technische Universität München
Fakultät für Informatik
Lehrstuhl für Effiziente Algorithmen
Dmytro Chibisov
Sandeep Sadanandan

Winter Semester 2007/08
Solution Sheet 6
November 30, 2007

Fundamental Algorithms

Problem 1 (10 Points)

A binary tree is $full$ if all of its vertices have either zero or two children. Let B_n denote the number of full binary trees with n vertices.

1. By drawing out all full binary trees with 3, 5, or 7 vertices, determine the exact values of B_3, B_5, and B_7. Why have we left out even numbers of vertices, like B_4?
2. For general n, derive a recurrence relation for B_n.

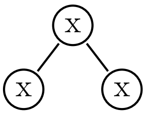
Solution

1. By drawing out all full binary trees with 3, 5, or 7 nodes, determine the exact values of B_3, B_5, and B_7. Why have we left out even numbers of vertices, like B_4?

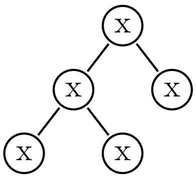
   The figure shows all the full binary trees with 3, 5 or 7 nodes. The the number of trees are 1, 2 and 5 respectively.

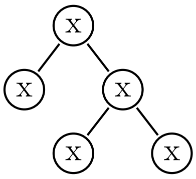
   There are no even number of nodes because, a tree with even number of nodes cannot be a full tree.

$B_3 = 1$

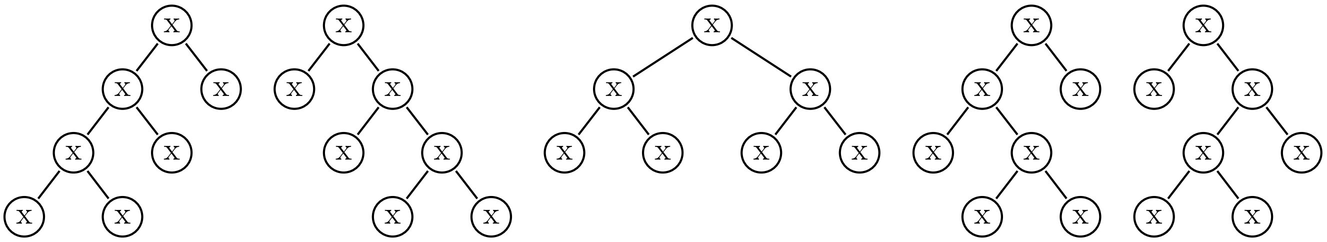


$B_5 = 2$





$B_7 = 5$



2. For general n, derive a recurrence relation for B_n.

$$B_n = \begin{cases} 2\left(B_{n-2} + B_{n-4}B_3 + \ldots + B_{\lceil \frac{n}{2} \rceil} B_{\lfloor \frac{n}{2} \rfloor - 1}\right) & \text{if } n = 4k+1 \\ 2\left(B_{n-2} + B_{n-4}B_3 + \ldots + B_{\lfloor \frac{n}{2} \rfloor} B_{\lfloor \frac{n}{2} \rfloor}\right) - B_{\lfloor \frac{n}{2} \rfloor} B_{\lfloor \frac{n}{2} \rfloor} & \text{if } n = 4k+3 \end{cases}$$

Problem 2 (10 Points)

Review all the sort algorithms taken in the class. Compare their complexities. If possible, try to explain them with day-to-day examples.

Prove that the lower bound for sorting is $n \lg n$

Solution

Sort	Average	Best	Worst	Remarks
Bubble sort	n^2	n^2	n^2	
Selection sort	n^2	n^2	n^2	
Insertion sort	n^2	n	n^2	In best case, insert requires constant time
Merge sort	$n \lg n$	$n \lg n$	$n \lg n$	
Heap sort	$n \lg n$	$n \lg n$	$n \lg n$	
Quick sort	$n \lg n$	$n \lg n$	n^2	

PROOF:

For an input of size n, the decision tree has $n!$ leaves. Which leaves the tree with a height $h \geq \lg(n!)$

$$\begin{aligned} h &\geq \lg(n!) \\ &\geq \lg\left(\left(\frac{n}{2}\right)^{\frac{n}{2}}\right) \\ &= \frac{n}{2}\left(\lg(n) - 1\right) \\ &\geq \left(\frac{n}{4}\right)\lg n \end{aligned}$$

Problem 3

Stacks and Queues.

1. Write pseudo code for `push(x), pop(), add(x), delete()`.
2. How can one simulate a queue with two stacks! (no counting)

What is a circular queue?

Solution

1. Stack

```
#define STACKSIZE 1000

unsigned int stack[STACKSIZE];
int top;

void push(int data)
```

```
{
        if (top < STACKSIZE)
                stack[top++] = data;
        else
                printf("Stack Full");
}

int pop()
{
        if(top != 0)
                return stack[--top];
        else
                print("Stack Empty");

        return -1;
}
```

2. Simulate Queue with Stacks

```
stack Stack1, Stack2;

void add(int data)
{
        Stack1.push(data);
}

int del()
{
        while possible to pop from Stack1
        {
                Stack2.push(Stack1.pop());
        }
        return Stack2.pop();
        while possible to pop from Stack2
        {
                Stack1.push(Stack2.pop());
        }
}
```

3. Circular Queue

   A circular queue is a queue which has a maximum capacity at a givem point of time. It acts as if its head and tail are connected.

   It is usually implemented with a normal array. Once the head/tail reaches the end of the array, the count starts again from the beginning.

Problem 4

Design the functions `insert(data), search(data) and delete(data)` in a binary search tree – RECURSIVELY.

Compare the complexity with the iterative implementations.

Solution

1. `insert(data)`

```
node * insert(node * tree, int data)
{
        if(tree == NULL)
                return newnode(data);
        if (data < tree->data)
                tree->left = insert(tree->left, data);
        if (data > tree->data)
                tree->right = insert(tree->right, data);
        if (data == tree->data)
                tree->count++;
        return tree;
}
```

2. `search(data)` is exactly like `insert(data)` - so, left as exercise.
3. `delte(data)`

```
void delete(node * tree, node * vater, int data)
{
        if (tree == NULL)
                return; // nothing to delete
        if(data < tree->data)
        { // happens to be in the left tree
                delete(tree->left, tree, data);
        }
        if(data > tree->data)
        { // let's delete it from the right subtree.
                delete(tree->right, tree, data);
        }

        // now we are on the tree NODE to be deleted.

        if(tree == vater) //  happens to be the root node.
                if(isleaf(tree)) // the only node in the tree
```

```
                {
                        free(tree);
                        return ;
                }
        else
        {
                if(isleaf(tree))
                {
                        if(vater->left == tree)
                                vater->left = NULL;
                        else // if (vater->right == tree)
                                vater->right = NULL;
                        return;
                }

                // if tree has only one child, we can replace tree by it's kid.
                if((onlykid = single_kid(tree)) != NULL)
                {
                        if(vater->left == tree)
                                vater->left = onlykid;
                        else // if (vater->right == tree)
                                vater->right = onlykid;
                        return;
                }

        }
        // not a leaf, nor the father of only one child -
        // hence replace tree with leftmost child of right child or
        // rightmost child of left child
        // random == 1 --> left child's rightmost child and
        // random == 2 --> right child's leftmost child

        random = replace(tree, vater); // does the random replacement.
        if(random == 1)
                delete(tree->left, tree, data);
        else // (random == 2)
                delete(tree->right, tree, data);
}
```

The number of recursive calls is the same as the number of iterations in the iterative loops. Hence the complexities of both the methods are the same. And it is $O(\lg n)$, where n is the number of nodes in the tree.