Department of Computer Science & Engineering, Faculty of Engineering

CSC2100B/S: Data Structures

Spring 2011

Written Assignment 3

Due: *8 Apr, 2011*

Name: _____

SID: _____

(Rev. 1.1 @ 28/03)

Section I: Revision Questions (10 points)

Put down your answers to the following simple revision questions.

1. The height of a binary tree with 10 nodes is at most **9** . (1 pt)

2. If the postorder traversal of a binary tree T is $3, 5, 2, 1, 4$, the root of T should be numbered as **4** . (1 pt)

3. The *average* time complexity for inserting a new node in a binary search tree with N nodes is $O(\lg N)$. (1 pt)

4. Usually an empty tree is represented by **NULL pointer** in C. (1 pt)

5. A binary heap of 12 nodes has height **3** . (1 pt)

6. The most time-efficient operation of a *max-heap* is **findmax** . (1 pt)

7. Binary heap can be used to sort N items in $O(N \lg N)$ time. (1 pt)

8. Changing the priority of an element in a binary heap requires a fixing up followed by a **fixing down** . (1 pt)

9. An AVL tree of height 3 contains at least **7** nodes. (1 pt)

10. AVL trees use the technique **rotation** to restore balance of tree heights. (1 pt)

Section II: Short Questions (45 points)
Unless otherwise stated, you should answer the following questions in plain English, diagrams or math expressions instead of code snippets.

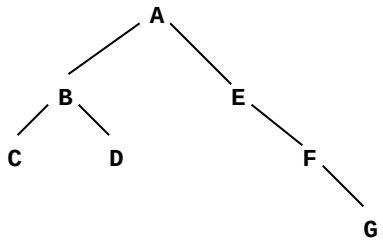
1. Suppose the postorder and inorder of a binary tree T is

```
C D B G F E A
C B D A E F G
```

respectively. Restore and draw T.
Is T a binary search tree?

5 pts

Solution:

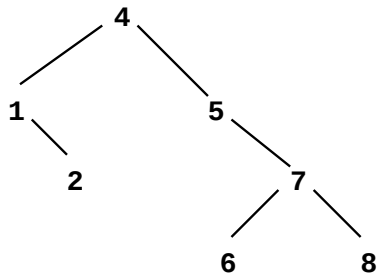


It is **NOT** a binary search tree.

2. Show the final binary search tree after inserting 3, 1, 5, 4, 2, 7, 8, 6 into an empty one and then deleting the root.

4 pts

Solution:



3. Suppose keys stored at nodes are integers, design a simple algorithm that finds the median among all keys stored in a *binary search tree* with N nodes. What is the time complexity of your algorithm?

4 pts

Solution: Keep a global counter.
Traverse the BST in inorder.
Increment the counter during the visit.
The median is found when the counter reaches $\lfloor N/2 \rfloor$.
Time complexity $= O(N/2) = O(N)$

4. Write a recursive function that determines the height of the binary (search) tree rooted at the node pointed by `t`. Use the function prototype: `int height(bst_t *t)`.
(*Hint*: let the height of an empty tree be -1)

4 pts

Solution:

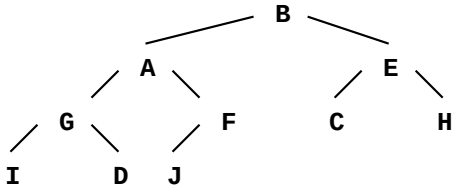
```
int height(bst_t *t){
    int u, v;
    if (t == NULL) return -1;
    u = height(t->l);
    v = height(t->r);
    return (u > v ? u + 1 : v + 1);
}
```

5. You are given an array of 10 characters (`A` to `J`), the 0-th entry contains `Z` which is intentionally inserted for easy heap building and should be excluded from the actual data.

13 pts

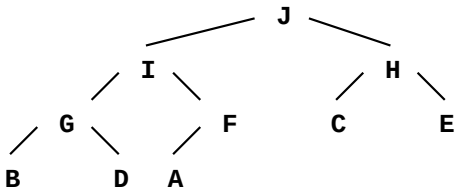
Z	B	A	E	G	F	C	H	I	D	J
0	1									10

(a) (3 pts) Draw the complete binary tree represented by the array.
Solution:



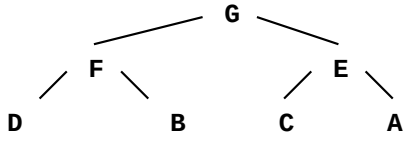
(b) (5 pts) Apply the linear-time *heapify* procedure to the array to convert it into a binary max-heap. Show *both* the array and the heap.
Solution:

Z	J	I	H	G	F	C	E	B	D	A
0	1									10



(c) (5 pts) Suppose the heap in (b) has its maximums deleted *three* times, show *both* the final array and heap.
Solution:

Z	G	F	E	D	B	C	A	H	I	J
0	1									10

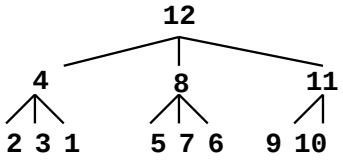


6. Sometimes it is more efficient to implement a 3-heap instead binary heap since the height of the tree would be smaller. In the 3-heap structure, every node (except the last one) has 3 children. If a 3-heap is stored as an array, for a node located at position k, where should be the parent and the children of the node?
Illustrate your answer with a non-trivial example (either min-heap or max-heap).

5 pts

Solution: The parent is at $\lfloor \frac{k+1}{3} \rfloor$.
The children are at $3k-1, 3k, 3k+1$.

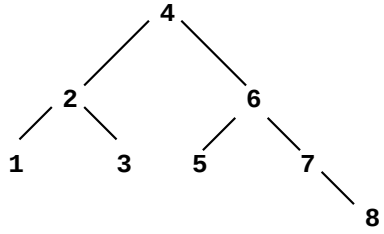
	12	4	8	11	2	3	1	5	7	6	9	10
0	1	2	3	4	5	6	7	8	9	10	11	12



7. Show the AVL tree constructed by inserting keys $3, 2, 1, 4, 5, 7, 6, 8$ into an empty one.

5 pts

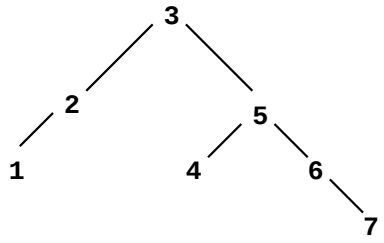
Solution:



8. In what order will the set of keys from 1 to 7 from an AVL tree of height 3? Suggest one possible sequence and draw the resultant AVL tree.

5 pts

Solution: One possible sequence: 3, 2, 5, 1, 4, 6, 7



Section III: Long Question / Code Study (30 points)
Unless otherwise stated, you should answer the following questions in plain English instead of code snippets.

15 pts

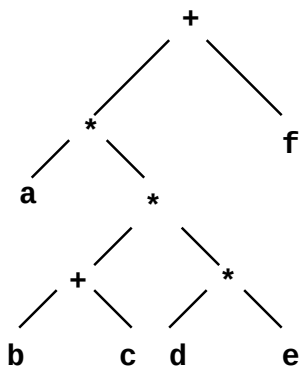
1. In this question, we are going to build a parse tree from a prefix expression. We begin with the basic tree node declaration in C:

```
1  typedef struct node_s node;
2  struct node_s {
3      char token;
4      node *l; /* left subtree */
5      node *r; /* right subtree */
6  };
```

Suppose a prefix expression (with only additions and multiplications) is read into a C string `s` with integer `i` initialized to 0 (assume both `s` and `i` are global variables), the following function `parse()` will construct a parse tree representing the expression:

```
1  node *parse(void){
2      char t = s[i++];
3      node *x = malloc(sizeof *x);
4      x->token = t;
5      x->l = x->r = NULL;
6      if ((t == '+' || t == '*')){
7          x->l = parse();
8          x->r = parse();
9      }
10     return x;
11 }
```

(a) (3 pts) Suppose `s` = `+*a*+bc*def`, draw the parse tree generated by the function.
Solution:



(b) (2 pts) What is the time complexity of the algorithm in terms of the number of nodes in the parse tree? Explain briefly.
Solution: Complexity = $O(N)$. This is exactly a preorder traversal.

(c) (6 pts) Write a C function `void infix(node *t)` that prints to `stdout` the fully parenthesized infix expression for the parse tree rooted at `t`.

Solution:

```
1  void infix(node *t){
2      if (!t) return;
3      if (t->token == '+' || t->token =='*') putchar('(');
4      infix(t->l);
5      putchar(t->token);
6      infix(t->r);
7      if (t->token == '+' || t->token == '*') putchar(')');
8  }
```

(d) (4 pts) Briefly describe how to evaluate the expression based on the parse tree, assuming that each value for the variables has been substituted in the leaf nodes properly (e.g. $a \rightarrow 3$).

Solution: Traverse the parse tree in postorder.
Define the visitor function as follows:
if it is an operator (+, *), retrieve operands from the left and right subtree (leaves) and calculate.
Return the computed result.

15 pts

2. *fix_up* and *fix_down* are two common operations used to restore the heap-order property after modifications of items or keys. Study the following implementations of the operations:

```
1  int fix_up(int n, int *a, int i){
2      int t, x = 0;
3      for ( ; a[i / 2] < a[i]; i /= 2){
4          t = a[i / 2];
5          a[i / 2] = a[i];
6          a[i] = t;
7          x++;
8      }
9      return x;
10 }
11
12 int fix_down(int n, int *a, int i){
13     int t, j, x = 0;
14     for ( ; i * 2 <= n; i = j){
15         j = i * 2;
16         if (______________________) j++;
17         if (a[i] > a[j])
18             break;
19         t = a[i];
20         a[i] = a[j];
21         a[j] = t;
22         x++;
23     }
24     return x;
25 }
```

(a) (1 pt) Fill in the missing condition on line 16.

Solution: `j < n && a[j + 1] > a[j]`

(b) (2 pts) What are the return values of the functions `fix_up()` and `fix_down()`?

Solution: In both functions, `x` is the no. of swaps happen during the operations.

(c) (2 pts) Are the functions building *min-heap* or *max-heap*? Explain briefly.

Solution: They are building max-heap, because the routines try to make the parents larger than its children. (lines 3, 16 and 17)

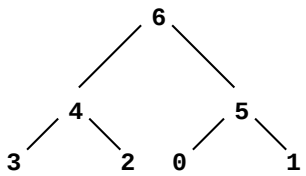
(d) (4 pts) The following function heapifies an array. Study it carefully:

```
1  int heapify1(int n, int *a){
2      int i, x = 0;
3      for (i = 2; i <= n; i++)
4          x += fix_up(n, a, i);
5      return x;
6  }
```

Suppose it is called on an array `a[]` $= \{7, 5, 3, 1, 4, 2, 0, 6\}$, write down the resultant array and the heap represented.

What is the return value in the above execution?

Solution: `a[]` $= \{7, 6, 4, 5, 3, 2, 0, 1\}$



$x = 3$

(e) (6 pts) Write a *better* C function `heapify2(int n, int *a)` that also heapifies and returns `x`. Propose an example to illustrate that your function is better.

Solution:

```
1  int heapify2(int n, int *a){
2      int i, x = 0;
3      for (i = n / 2; i > 0; i--)
4          x += fix_down(n, a, i);
5      return x;
6  }
```

`a[]` $= \{100, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$
`heapify1()` returns 19. `heapify2()` returns 8.