Data Structures and Algorithms

Pierre Collet/ Manal ElZant-Karray

Practical work n°8

Exercise 1: Add to the program of implementation of Binary search tree (doing in lecture session):

1. The function to print the BST in ascending order (What do you notice?)
2. Can you deduce the function to print the BST in descending order?

Exercise 2: Write a function to check if a given Binary tree is a Binary search Tree.

A BST is a BT which has the following properties:
• The left subtree of a node contains only nodes with keys less than the node's key.
• The right subtree of a node contains only nodes with keys greater than the node's key.
• Both the left and right subtrees must also be binary search trees.

First approach: The algorithm is to check for each node whether the maximum of the left subtree is lesser than the node and the minimum of the right subtree is greater than the node. This algorithm works perfect but not efficient in terms of time complexity.

Algorithm:

```
1.  Set prev to INT_MIN.
2.  From main function call checkBST(root, prev)
    //passing prev by reference to update it every time
    checkBST(root, &prev)
3.  if(root==NULL)
        return 1; //null tree is BST
4.  do in-order traversal and checking whether all tree node
    data is sorted or not
    if(!(checkBST(root->left,prev)))  //check left subtree
        return 0;
    //root->data must be greater than prevsince BST results in
    //sorted list after in-order traversal.
5.  if(root->data < prev)
        return 0;
6.  prev=root->data; //update prev value
7.  return checkBST(root->right,prev);//check right subtree
```

Second approach: Can you use the output of the exercise 1 to check if your BT is a BST?

Exercise 3: Write a function to calculate the sum of left and right subtree in a BST.

Exercise 4: Complete the following C Program to convert infix to prefix using stack and evaluate prefix expression.

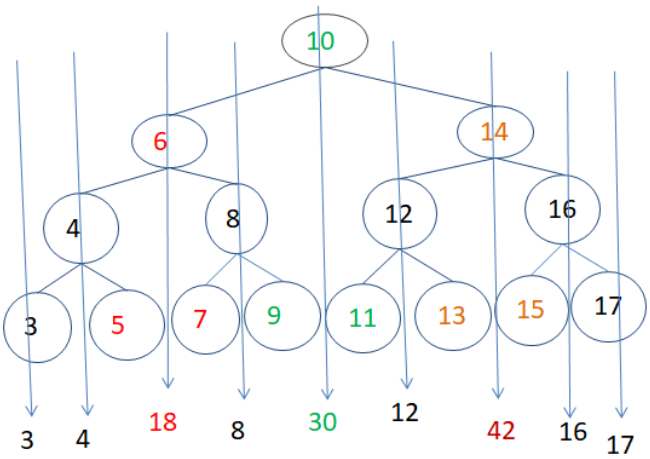
```
#include<stdio.h>
#include<string.h>
#include<math.h>
#include<stdlib.h>
#define BLANK ' '
```

```
#define TAB '\t'
#define MAX 50
long int pop();
long int eval_pre();
char infix[MAX], prefix[MAX];
long int stack[MAX];
int top;
int isempty();
int white_space(char symbol);
void infix_to_prefix();
int priority(char symbol);
void push(long int symbol);
long int pop();
long int eval_pre();

int main(){
   long int value;
   top = -1;
   printf("Enter infix : ");
   gets(infix);
   infix_to_prefix();
   printf("prefix : %s\n",prefix);
   value=eval_pre();
   printf("Value of expression : %ld\n",value);
   return 0;
}
void infix_to_prefix(){
...
}
long int eval_pre(){
...
}
//This function returns the priority of the operator
int priority(char symbol ){
   switch(symbol){
      case ')': return 0;
      case '+':
      case '-': return 1;
      case '*':
      case '/':
      case '%': return 2;
      case '^': return 3;
      default : return 0;
   }
}
void push(long int symbol){
   if(top > MAX){
```

```
      printf("Stack overflow\n");
      exit(1);
    }
    else{
      top=top+1;
      stack[top] = symbol;
    }
}
long int pop(){
    if(top == -1 ) {
      printf("Stack underflow \n");
      exit(2);
    }
    return (stack[top--]);
}
int isempty(){
    if(top==-1) return 1;
    else return 0;
}
int white_space(char symbol){
    if(symbol==BLANK || symbol==TAB || symbol=='\0') return 1;
    else return 0;
}
```

Exercise 5: (to submit!)Write a function to calculate the *vertical sum* in a BST.



Exercise 6: Complete the following C program to Huffman coding using heap data structure:

```
#include <stdio.h>
#include <stdlib.h>
#define MAX_TREE_HT 50
struct MinHNode {
  char item;
  unsigned freq;
  struct MinHNode *left, *right;
```

```
};
struct MinHeap {
  unsigned size;
  unsigned capacity;
  struct MinHNode **array;
};
struct MinHNode *newNode(char item, unsigned freq) {
  struct MinHNode *temp = (struct MinHNode *)malloc(sizeof(struct MinHNode));
  temp->left = temp->right = NULL;
  temp->item = item;
  temp->freq = freq;
  return temp;
}
struct MinHeap *createMinH(unsigned capacity) {// Create min heap
…
}
void swapMinHNode(struct MinHNode **a, struct MinHNode **b) { // Function to swap
  struct MinHNode *t = *a;
  *a = *b;
  *b = t;
}
void printArray(int arr[], int n) { // Print the array
  int i;
  for (i = 0; i < n; ++i) printf("%d", arr[i]);
  printf("\n");
}
void minHeapify(struct MinHeap *minHeap, int idx) { // Heapify
  …
}
int checkSizeOne(struct MinHeap *minHeap) { // Check if size if 1
  return (minHeap->size == 1);
}
struct MinHNode *extractMin(struct MinHeap *minHeap) { // Extract min
  ….
}
void insertMinHeap(struct MinHeap *minHeap, struct MinHNode *minHeapNode) {
… // Insertion function
}
void buildMinHeap(struct MinHeap *minHeap) {
…
}
int isLeaf(struct MinHNode *root) {
  return !(root->left) && !(root->right);
}
struct MinHeap *createAndBuildMinHeap(char item[], int freq[], int size) {
…
}
```

```
struct MinHNode *buildHuffmanTree(char item[], int freq[], int size) {
  .....
}
void printHCodes(struct MinHNode *root, int arr[], int top) {
}
void HuffmanCodes(char item[], int freq[], int size) { // Wrapper function
  ...
}
int main() {
  char arr[] = {'A', 'B', 'C', 'D'};
  int freq[] = {5, 1, 6, 3};
  int size = sizeof(arr) / sizeof(arr[0]);
  printf(" Char | Huffman code ");
  printf("\n--------------------\n");
  HuffmanCodes(arr, freq, size);
}
```