Data Structure and Algorithm
Homework #3

=== Homework Reference Solution ===

***Problem* 1.** Binary Search Tree

We can use the following structure to represent our binary search tree.

```
struct node{
        char word[20+1];            // The maximal length of a word is 20
        struct node *left, *right;  // Point to the left sub-tree and the right sub-tree
};
```

- Compare the two words, *str1* and *str2*
  We can directly use the function strcmp(*str1*, *str2*) in string.h to determine the relationship between *str1* and *str2*. If the returned value is zero, it means that *str1* and *str2* are equal. If the returned value is less than zero, it means that *str1* < *str2*. Otherwise, it means that *str1* > *str2*.

- INSERT *str*
  Start from the root node, if the *node->word* > *str*, go to the left-child node. If the *node->word* < *str*, go to the right-child node. If *node->word* == *str*, print error. If pointer which points to the left-child or the right-child is NULL, set the value of that pointer to point to the new node with *str*.

- DELETE *str*
  To delete the node with the given *str*, first follow the step described in INSERT to find the node with *str*. Then, consider the four different conditions: the node has one child and the child is the left child, the node has one child and the child is the right child, the node has two children, and the node has no child. Use the method described in the lecture note “Tree 1” to delete the node.

- PREORDER, INORDER, and POSTORDER
  Use recursive functions to traverse the tree. For example, we can use the following function to implement INORDER traversal.

```
void inorder (Node *root) {
        if (root) {
                inorder(root->left);
                printf("%s", root->data);
                inorder(root->right);
        }
}
```

***Problem* 2.** Left Child Right Sibling Representation

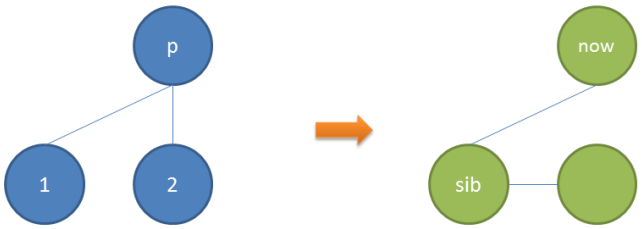
2.1

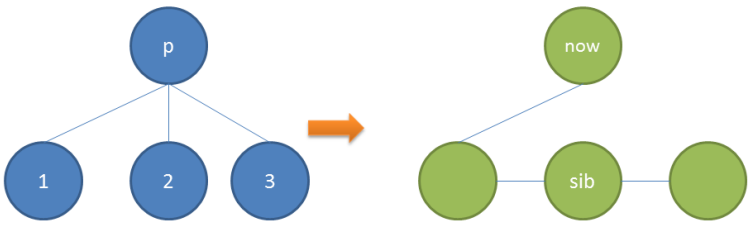
```
LCRSNode* toLCRS(NodeTree* d3treeRoot)
{
	NodeLCRS* lcrsRoot = newLCRSNode();
	toLCRSinner(d3treeRoot, lcrsRoot);
	return lcrsRoot;
}
void toLCRSinner(NodeTree* p, NodeLCRS* now)
{
	NodeLCRS* sib;
	now->data = p->data;

	//left child
	if(p->child1 == NULL)			//no child
		return;
	now->left = newLCRSNode();
	toLCRSinner(p->child1, now->left);	//add the first child of p to the left of lcrs

	//right sibling
	if(p->child2 == NULL)			//only one child
	return;
	sib = now->left;
	sib->right = newLCRSNode();
	toLCRSinner(p->child2, sib->right);


	if(p->child3 == NULL)
		//only two children
		return;
	sib = sib->right;
	sib->right = newLCRSNode();
	toLCRSinner(p->child3, sib->right);
}
LCRSNode* newLCRSNode()
{
   NodeLCRS* p = (NodeLCRS*)malloc(sizeof(NodeLCRS));
   p->left = p->right = NULL;
}
```





2.2

```
printData(NodeLCRS* p)
{
	if(p == NULL)
		return;
```

```
        print(p->data);
        printData(p->left);
        printData(p->right);
}
```

2.3
Preorder traversal. The result corresponds to the depth-first traversal of the original tree.

2.4
A B E K L F C G D H M I J

***Problem* 3.** Binary Tree and Binary Search Tree

3.1

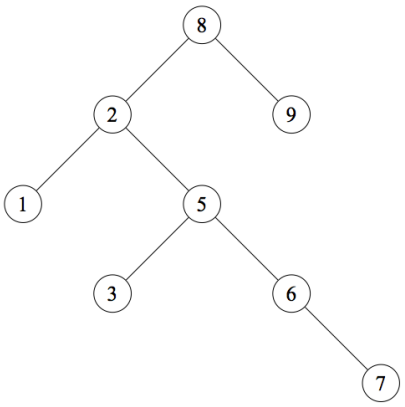
```
list find(T, a, b)
    list ans = []  // here ans is a list
    if a<T.value
        ans += find(T.left, a, b)  //"+=" will append the returned list from find()
to the end of "ans"
    if a<=T.value<=b
        ans += T.value
    if T.value<b
        ans += find(T.right, a, b)
    return ans
```

3.2

(i) Suppose a node in a binary search tree has two children, and its predecessor in the inorder traversal has the right child. As discussed in the class, the inorder predecessor node P of a node X is "the node with the maximum value in its left sub-tree". If the predecessor node P of node X has the right child, it is NOT the node with the maximum value in the left sub-tree of X, which leads to contradiction of the initial assumption. Therefore, the original statement in the problem is true.

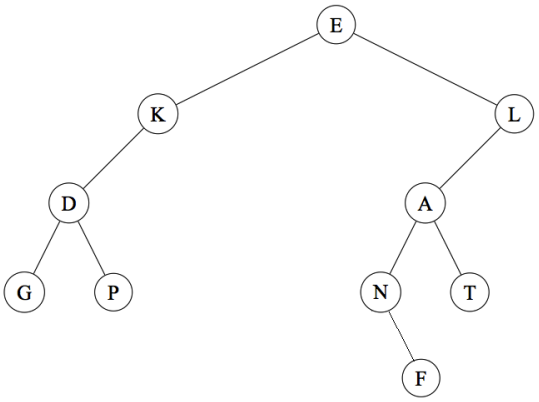
(ii) Suppose a node in a Binary Search Tree has two children, and its successor in the inorder traversal has the left child. The inorder successor node S of a node X is "the node with the minimum value in its right sub-tree". If the successor node S of node X has the left child, it is NOT the node with the minimum value in the right sub-tree of X, which leads to contradiction of the initial assumption. Therefore, the original statement in the problem is true.

3.3



3.4

The last node in post-order traversal will be the root of tree, i.e. E is the root. We can therefore separate the nodes into left subtree (i.e. GDPK) and right subtree (i.e. NFATL) according to the in-order traversal. The same procedure can be applied to the subtrees until no nodes can be separated into subtrees. The final result is shown in the figure below.


E
K
L
D
A
G
P
N
T
F

***Problem* 4.** Max Heap

4.1
The optimal algorithm to solve this question is as follows.
Assume the number of elements in the given heap array is *N*.
Elements from $A[\lfloor \frac{N}{2} \rfloor + 1]$ to $A[N]$ are leaves, so we start from node $A[\lfloor \frac{N}{2} \rfloor]$ down to node $A[1]$, and for each node we heapify the subtree of the current node.

Time compexity: In the worst case, each node will travel to the leaf level, assumed it's *H*. Hence the total comparison times is $H + 2(H-1) + 2^2(H-2) + \cdots + 2^{H-1} \cdot 1$.

Let

$$S = H + 2(H-1) + 2^2(H-2) + \cdots + 2^{H-1} \cdot 1$$
$$2S = 2H + 2^2(H-1) + 2^3(H-2) + \cdots + 2^H \cdot 1$$
$$2S - S = -H + 2(1) + 2^2(1) + \cdots + 2^{H-1} + 2^H$$
$$\therefore S = 2^{H+1} - H - 2$$
$$\text{since } H = \lfloor \log_2(N+1) \rfloor - 1$$
$$\therefore S = 2^{\lfloor \log_2(N+1) \rfloor} - (\lfloor \log_2(N+1) \rfloor - 1) - 2$$
$$\therefore O(S) = O(N)$$

In the best case, the time complexity is $\Omega(N)$.
$\therefore$ the time complexity of this algorithm is $\theta(N)$.

4.2
Initialization: **50, 25**
Insert 66: **50, 25, 66**
Heapify the array: **66, 25, 50**
Insert 33: **66, 25, 50, 33**
Heapify the array: **66, 33, 50, 25**
Insert 34: **66, 33, 50, 25, 34**
Heapify the array: **66, 34, 50, 25, 33**
Insert 74: **66, 34, 50, 25, 33, 74**
Heapify the array (swap 50 and 74): **66, 34, 74, 25, 33, 50**
Heapify the array (swao 66 and 74): **74, 34, 66, 25, 33, 50**

4.3
Initialization: **74, 34, 66, 25, 33, 50**
Delete 33: **74, 34, 66, 25, 50**
Heapify the array: **74, 50, 66, 25, 34**
Delete 66: **74, 50, 34, 25** → It is already heapified.

***Problem* 5.** Threaded Binary Tree

5.1

(a) If the newly inserted leaf node is the left-child of its parent, then it lies between its parent's inorder predecessor and its parent in the inorder traversal. Therefore, the left-thread of the newly inserted node will point to where the original left-thread of its parent points, and the right-thread of the newly inserted node will be a thread pointing to its parent.

(b) If the newly inserted leaf node is the right-child of its parent, then it lies between its parent and its parent's inorder succesor in inorder traversal. Therefore, the right-thread of the newly inserted node will point to where the original right-thread of the parent points, and the left-thread of the newly inserted node will be a thread pointing to its parent.

5.2

If the given node is the right child of its parent, then we will keep following the left pointer, until the traversal reaches a leaf node and follow one left thread to go back to the given node's parent. This is what the function inorderPredecessor does. The only exception is that if the given node is the left child; in this case the traversal will eventually reach the root, and the function will return NULL.

Similarly, if the given node is the left child of its parent, then we will keep following the right pointer, until the traversal reaches the leaf node and follow one right thread to go to the given node's parent. This is what the function inorderSucessor does. The only exception is that if the given node is the right child; in this case the traversal will eventually reach the root, and the function will return NULL.

The function *find_parent()* just combines the two functions described above.

```
TBT_Node *  inorderPredecessor (TBT_Node *  now, TBT_Node * given_node) {
      if(now->RTag == 1 && now -> RPtr == given_node)
      //Rtag==0 is a thread. Check if the current node is the given node's parent
            return now;
      if (now -> RPtr == now)  //root of a binary tree
            return NULL;
      return inorderPredecessor (now->LPtr , given_node);
}

TBT_Node* inorderSuccesor(TBT_Node* now, TBT_Node* given_node){
      if(now->LTag == 1 && now -> LPtr == given_node)
      //Ltag==0 is a thread. Check if the current node is the given node's parent
            return now;
      if (now -> RPtr == now) //root of binary tree
            return NULL;
      return      inorderSuccesor (now->RPtr, given_node) ;
}
TBT_Node* find_parent(TBT_Node * given_node){
      TBT_Node * parent ;
      parent = inorderPredecessor(given_node , given_node);
      if(parent ==NULL)
            parent = inorderSuccessor (given_node , given_node);
       return     parent ;
}
```