Homework 9_1 [Hashing.cpp]

Variable analysis

```
#define KEY_SIZE 10 : the maximum length of string.
#define TABLE_SIZE 13 : the Bucket size in hash table.

ListNode* hash_table[TABLE_SIZE] : hash table.
```

element	
type	**name**
char [KEY_SIZE]	key

key : starting address of char array (=string)

ListNode	
type	**name**
element	item
ListNode *	link

Item : data used in hash key.

link : the address of connected node

Function analysis

```
void hash_chain_delete(element item, ListNode* ht[])
```

```
void hash_chain_delete(element item, ListNode* ht[]) {

	int hash_value = hash_function(item.key); //get hash value

	ListNode* node_before = NULL;
	ListNode* node;

	// find node with item.
	for (node = ht[hash_value]; node; node_before = node, node = node->link) {
		if (equal(node->item, item))
			break;
	}
```

```
        // if node is NULL , there is no element corresponding to given item.
        if (node == NULL) {
                printf("delete key does not exist\n");
                return;
        }

                ListNode* temp = node;

                if (node_before)
                {
                        node_before->link = node->link;
                        printf("not null");
                }
                else //node_before is null
                {
                        ht[hash_value] = node->link;
                        printf("null");
                }
                free(temp); //deallocate memory

}
```

: In order to delete certain key in the hash table, we not only know the node with key `itself`, but also the **before node** of the key. If node is null, which indicates fails to find given element, just return. The only thing left after finding what I mentioned is to connect before node to next node of key node. There are two cases in hash chain deletion.

1. Node_before is NULL

   : It means that the key node is the headmost of the bucket it is included in.

   So substitude ht[hash_vaule] for second head of the bucket ,which is completely same as node->link.

2. Node_before is not NULL

   : It means that the key node has both before and after node. So substitude node_befor->link for next node of key node (=node->link)

[result]

```
Enter the operation to do (0: insert, 1: delete, 2:search, 3: termination): 0
Enter the  key: and
[0] -> null
[1] -> null
[2] -> null
[3] -> null
[4] -> null
[5] -> null
[6] -> null
[7] -> null
[8] -> and -> null
[9] -> null
[10] -> null
[11] -> null
[12] -> null

Enter the operation to do (0: insert, 1: delete, 2:search, 3: termination): 0
Enter the  key: test
[0] -> null
[1] -> null
[2] -> null
[3] -> null
[4] -> null
[5] -> null
[6] -> test -> null
[7] -> null
[8] -> and -> null
[9] -> null
[10] -> null
[11] -> null
[12] -> null

Enter the operation to do (0: insert, 1: delete, 2:search, 3: termination): 0
Enter the  key: cat
[0] -> cat -> null
[1] -> null
[2] -> null
[3] -> null
[4] -> null
[5] -> null
[6] -> test -> null
[7] -> null
[8] -> and -> null
[9] -> null
[10] -> null
[11] -> null
[12] -> null
```

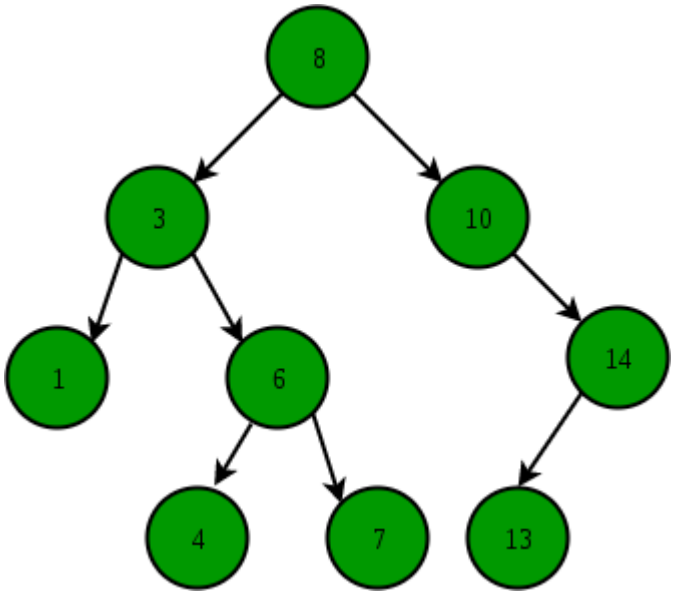
```
Enter the operation to do (0: insert, 1: delete, 2:search, 3: termination): 0
Enter the  key: dna
[0] -> cat -> null
[1] -> null
[2] -> null
[3] -> null
[4] -> null
[5] -> null
[6] -> test -> null
[7] -> null
[8] -> and -> dna -> null
[9] -> null
[10] -> null
[11] -> null
[12] -> null

Enter the operation to do (0: insert, 1: delete, 2:search, 3: termination): 0
Enter the  key: nad
[0] -> cat -> null
[1] -> null
[2] -> null
[3] -> null
[4] -> null
[5] -> null
[6] -> test -> null
[7] -> null
[8] -> and -> dna -> nad -> null
[9] -> null
[10] -> null
[11] -> null
[12] -> null

Enter the operation to do (0: insert, 1: delete, 2:search, 3: termination): 1
Enter the  key: dna
not null[0] -> cat -> null
[1] -> null
[2] -> null
[3] -> null
[4] -> null
[5] -> null
[6] -> test -> null
[7] -> null
[8] -> and -> nad -> null
[9] -> null
[10] -> null
[11] -> null
[12] -> null
```

Homework 9_2 [Bst_sort.cpp]

Binary search tree



Binary Search Tree is a node-based binary tree data structure which has the following properties:

- The left subtree of a node contains only nodes with keys lesser than the node's key.
- The right subtree of a node contains only nodes with keys greater than the node's key.
- The left and right subtree each must also be a binary search tree.

To use Binary Search Tree for sorting data, we have to apply proper tree traversal considering characteristics of BST. In BST left child <= vertex <= right is always true. So traversing with `Left -> vertex -> right` is needed, which is inorder traversal.
For example, if we apply inoder traversal to above graph, the visiting sequence is 1 -> 3->4->6->7->8->10->13->14. It is the desired result with acending order(sorted).

Variable analysis

Node	
type	**name**
int	data
Node *	left,right

data : the key used in bst insertion

left : the left child node

right : the right child node

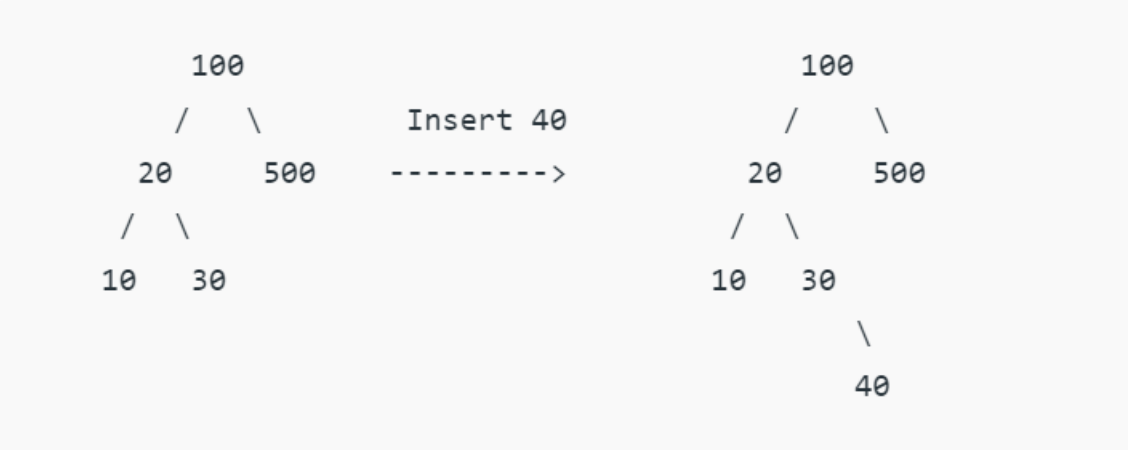
function analysis

1) int random(int data_maxval)

```
// Integer random number generation function between 1 and n
int random(int data_maxval) {
        return rand() % data_maxval + 1;
}
```

: Returns the generated random number not overring maximum value.

2) Node* bst_insert(Node* root, int key)




```
node* bst_insert(node* root, int key) {
        if (root == NULL) { // if find proper place to be inserted

                // generate new node.
                root = (node*)malloc(sizeof(node));
                root->data = key;
                root->left = root->right = NULL;
        }
        else if (root->data >= key) // if key is the smaller
                root->left = bst_insert(root->left, key); // move to left tree
```

```
        else // if key is the larger
                root->right = bst_insert(root->right, key); // move to right tree
        return root;
}
```

: A new key is always inserted at the leaf. We start searching a key from the root until we hit a leaf node. Once a leaf node is found, the new node is added as a child of the leaf node.

3) `void in_order(Node* root)`

```
// l -> v - > r
void in_order(node* root) {
        if (root == NULL)
                return;
        in_order(root->left); // left
        printf("%d\n", root->data); // vertex
        in_order(root->right); // right

}
```

: Using inorder(left->vertex->right)traversal, it prints out all nodes in the BST in acending order.

[result]

```
7
20
24
29
30
31
37
43
67
68
76
89
122
151
175
180
182
207
230
254
257
275
275
277
287
288
297
301
328
341
347
354
357
365
369
379
387
387
398
407
411
416
436
440
445
450
453
462
464
```