Dictionary using a Binary Search Tree

You are required to implement a basic dictionary using a Binary Search Tree (BST). Each entry in the dictionary consists of a word (key) and its meaning (value). The operations that you need to support are inserting a word, deleting a word, and searching for a word. After all operations are performed, the BST should display the words in lexicographical order. Additionally, for every search operation, you must track whether the word was found (1 for success, 0 for failure).

Input Format

The first line contains an integer N, the number of operations.
For each of the next N lines, an integer specifies the operation to be performed followed by the required parameters:
Insert (1): The integer 1, followed by the word (key) and its meaning (value).
Delete (2): The integer 2, followed by the word to be deleted.
Search (3): The integer 3, followed by the word to be searched.

Constraints

Maximum word length: 25 characters.
Maximum meaning length: 100 characters.

Output Format

Inorder Traversal of the BST: After all operations, print the contents of the dictionary in lexicographical order as word,meaning pairs.
Search Results: Print the result of each search operation as a space-separated number of 1 (if the word was found) or 0 (if not found).

Sample Input 0

```
4
1
apple red fruit
1
black a colour
2
apple
3
black
```

Sample Output 0

```
Inorder Traversal:
black, a colour

Search Results:
1
```

Explanation 0

The program should insert the words "apple" and "black" with their meanings into a binary search tree, deletes "apple," and finally searches for "black," resulting in an inorder traversal that displays "black, a colour" and a search result of 1 indicating success.

Sample Input 1

```
3
1
loud high volume
2
louds
3
loud
```

Sample Output 1

```
Inorder Traversal:
loud, high volume

Search Results:
1
```

Explanation 1

The program inserts the word "loud" with its meaning into a binary search tree, deletes the non-existent word "louds," and successfully searches for "loud," resulting in an inorder traversal that shows "loud, high volume" and a search result of 1 indicating success.

Contest ends in 2 hours

Submissions: 27
Max Score: 10
Difficulty: Medium

Rate This Challenge:

More

C

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define MAX1 25
#define MAX2 100

typedef struct node {
  char key[MAX1], value[MAX2];
  struct node *left, *right;
} NODE;

typedef struct tree {
  NODE *root;
} TREE;

// Function to initialize the tree
void init(TREE *pt);
// Function to create a new node
NODE *createNode(char word[MAX1], char meaning[MAX2]);
// Function to recursively insert a node into the BST
NODE *rinsert(NODE *r, NODE *temp);
// Function to insert a node into the tree
void recInsert(TREE *pt, char word[MAX1], char meaning[MAX2]);
// Function for inorder traversal of the BST
void inorder(NODE *r);
// Function to delete a node from the BST
NODE *delNode(NODE *r, char word[MAX1]);
// Function for iterative search in the BST
int search(NODE *r, char word[MAX1]);
// Function to destroy all nodes in the tree
void destroyNode(NODE *r);
// Function to destroy the tree
void destroyTree(TREE *pt);
```

```c

int main() {
  TREE tobj;
  init(&tobj);
  int num_operations;
  char word[MAX1], meaning[MAX2];
  int choice;

  // printf("Enter the number of operations: ");
  scanf("%d", &num_operations);

  // Array to store search results (1 for success, 0 for failure)
  int search_results[num_operations];
  int search_index = 0;

  // Processing all operations
  for (int i = 0; i < num_operations; i++) {
    scanf("%d", &choice);

    switch (choice) {
    case 1: // Insert
      scanf("%s", word);
      fflush(stdin);

      scanf("%[^\n]", meaning);
      recInsert(&tobj, word, meaning);
      break;
    case 2: // Delete
      scanf("%s", word);
      tobj.root = delNode(tobj.root, word);
      break;
    case 3: // Search (results for search only)
      scanf("%s", word);
      if (search(tobj.root, word)) {
        search_results[search_index++] = 1; // Search success
      } else {
        search_results[search_index++] = 0; // Search failure
      }
      break;
    default:
      printf("Invalid operation\n");
    }
  }

  // Inorder traversal after all operations
  printf("Inorder Traversal:\n");
  inorder(tobj.root);

  // Print search results (1 for success, 0 for failure)
  printf("\nSearch Results:\n");
  for (int i = 0; i < search_index; i++) {
    printf("%d ", search_results[i]);
  }
  printf("\n");

  destroyTree(&tobj);
  return 0;
}

void init(TREE *pt) { pt->root = NULL; }

NODE *createNode(char word[MAX1], char meaning[MAX2]) {
  NODE *temp = malloc(sizeof(NODE));
  strcpy(temp->key, word);
  strcpy(temp->value, meaning);
  temp->left = NULL;
  temp->right = NULL;
  return temp;
}

// BST: Recursive Insert
NODE *rinsert(NODE *r, NODE *temp) {
  if (r == NULL) {
```

```
    return temp;
  }
  if (strcmp(temp->key, r->key) < 0) {
    r->left = rinsert(r->left, temp);
  } else if (strcmp(temp->key, r->key) > 0) {
    r->right = rinsert(r->right, temp);
  }
  return r;
}

void recInsert(TREE *pt, char word[MAX1], char meaning[MAX2]) {
  NODE *temp = createNode(word, meaning);
  pt->root = rinsert(pt->root, temp);
}

void inorder(NODE *r) {
  // complete the function for inorder traversal
  if (r != NULL) {
    inorder(r->left);
    printf("%s,%s\n", r->key, r->value);
    inorder(r->right);
  }
}

NODE *delNode(NODE *r, char word[MAX1]) {
  // complete function to delete the node and return the root
  if (r == NULL)
    return r;
  if (strcmp(word, r->key) < 0) {
    r->left = delNode(r->left, word);
  } else if (strcmp(word, r->key) > 0) {
    r->right = delNode(r->right, word);
  } else {
    if (r->left == NULL) {
      NODE *temp = r->right;
      free(r);
      return temp;
    } else if (r->right == NULL) {
      NODE *temp = r->left;
      free(r);
      return temp;
    }
    NODE *p = r->right;
    while (p->left != NULL)
      p = p->left;
    strcpy(r->key, p->key);
    strcpy(r->value, p->value);
    r->right = delNode(r->right, p->key);
  }
  return r;
}

// BST: Iterative search
int search(NODE *r, char word[MAX1]) {
  // complete code to search for the given word
  if (r == NULL) {
    return 0;
  }
  if (strcmp(word, r->key) == 0) {
    return 1;
  } else if (strcmp(word, r->key) < 0) {
    return search(r->left, word);
  } else {
    return search(r->right, word);
  }
}

void destroyNode(NODE *r) {
  if (r != NULL) {
    destroyNode(r->left);
    destroyNode(r->right);
    free(r);
  }
```

```
180 }
181
182 void destroyTree(TREE *pt) {
183   if (pt->root != NULL) {
184     destroyNode(pt->root);
185     pt->root = NULL;
186   }
187 }
```

Line: 1 Col: 1

Upload Code as File Test against custom input Run Code Submit Code