Dictionary.h

```
/*
*  makeEmpty
*  pre: none
*  post: isEmpty(D)
*/
void makeEmpty(DictionaryRef D);

/*
*  printDictionary
*  pre: none
*  post: prints a text representation of D to file pointed to by out.
*/
void printDictionary(DictionaryRef D, FILE* out);


#endif
```

Dictionary.c

```
/*
*    Dictionary.c
*    Implementation file for Dictionary ADT based on Binary Search Tree
*/

#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#include<assert.h>
#include"Dictionary.h"

/********************************* Private types and functions *********************

/*
*  Node
*/
typedef struct Node{
   int key;
   int value;
   struct Node* left;
   struct Node* right;
} Node;

/*
*  NodeRef
*/
typedef Node* NodeRef;

/*
*  Dictionary
*/
typedef struct Dictionary{
   NodeRef root;
   int numPairs;
} Dictionary;

/*
*  newNode
*  Constructor for the private Node type
*/
NodeRef newNode(int k, int v) {
   NodeRef N = malloc(sizeof(Node));
   assert(N!=NULL);
   N->key = k;
   N->value = v;
   N->left = N->right = NULL;
   return(N);
}

/*
*  freeNode
*  Destructor for the private Node type
*/
void freeNode(NodeRef* pN){
   if( pN!=NULL && *pN!=NULL ){
      free(*pN);
      *pN = NULL;
   }
}
```

Dictionary.c

```
/*
*  findKey
*  returns a reference to the Node containing key k in the subtree rooted at R,
*  or NULL if no such Node exists
*/
NodeRef findKey(NodeRef R, int k){
   if(R==NULL || k==R->key) return R;
   if( k<R->key ) return findKey(R->left, k);
   else return findKey(R->right, k);
}

/*
*  findParent
*  returns a reference to the Node which is the parent of N in the subtree rooted
*  at R, or returns NULL if N is equal to R.
*/
NodeRef findParent(NodeRef N, NodeRef R){
   NodeRef P=NULL;
   if( N!=R ){
      P = R;
      for( ; P->left!=N && P->right!=N; P=(N->key<P->key?P->left:P->right) );
   }
   return P;
}

/*
*  findLeftmost
*  returns the leftmost Node in the subtree rooted at R, or NULL if R is NULL.
*/
NodeRef findLeftmost(NodeRef R){
   NodeRef L = R;
   if( L!=NULL) for( ; L->left!=NULL; L=L->left) ;
   return L;
}

/*
*  printInOrder
*  prints the (key, value) pairs belonging to the subtree rooted at R in order
*  of increasing keys to file pointed to by out.
*/
void printInOrder(NodeRef R, FILE* out){
   if( R!=NULL ){
      printInOrder(R->left, out);
      fprintf(out, "%d %d\n", R->key, R->value);
      printInOrder(R->right, out);
   }
}

/*
*  deleteAll
*  deletes all Nodes in the subtree rooted at N.
*/
void deleteAll(NodeRef N){
   if( N!=NULL ){
      deleteAll(N->left);
      deleteAll(N->right);
      freeNode(&N);
   }
}
```

Dictionary.c

```
/********************************* Public functions *****************************/

/*
*  newDictionary
*  Allocates and initializes a struct representing an empty Dictionary.  Returns a
*  pointer to new memory if successful, terminates if not successful.
*/
DictionaryRef newDictionary(void){
   DictionaryRef D = malloc(sizeof(Dictionary));
   assert(D!=NULL);
   D->root = NULL;
   D->numPairs = 0;
   return D;
}

/*
*  freeDictionary
*  Frees all heap memory associated with *pD and sets the reference *pD to NULL
*/
void freeDictionary(DictionaryRef* pD){
   if( pD!=NULL && *pD!=NULL ){
      makeEmpty(*pD);
      free(*pD);
      *pD = NULL;
   }
}

/*
*  isEmpty
*  pre: none
*  post: returns 1 (true) if D is empty, 0 (false) otherwise
*/
int isEmpty(DictionaryRef D){
   if( D==NULL ){
      fprintf(stderr, "Dictionary Error: calling isEmpty() on NULL DictionaryRef\n")
      exit(EXIT_FAILURE);
   }
   return(D->numPairs==0);
}

/*
*  size
*  pre: none
*  post: returns the number of entries in D
*/
int size(DictionaryRef D){
   if( D==NULL ){
      fprintf(stderr, "Dictionary Error: calling size() on NULL DictionaryRef\n");
      exit(EXIT_FAILURE);
   }
   return(D->numPairs);
}
```

Dictionary.c

```
/*
*  lookup
*  pre: none
*  post: returns value associated key k, or UNDEF if no such key exists
*/
int lookup(DictionaryRef D, int k){
   NodeRef N;
   if( D==NULL ){
      fprintf(stderr, "Dictionary Error: calling lookup() on NULL DictionaryRef\n");
      exit(EXIT_FAILURE);
   }
   N = findKey(D->root, k);
   if( N!=NULL ) return N->value;
   else return UNDEF;
}

/*
*  insert
*  inserts new (key,value) pair into D
*  pre: key k currently does not exist in D, i.e. lookup(D, k)==UNDEF
*  post: !isEmpty(D), size(D) is increased by one
*/
void insert(DictionaryRef D, int k, int v){
   NodeRef N, A, B;
   if( D==NULL ){
      fprintf(stderr, "Dictionary Error: calling insert() on NULL DictionaryRef\n");
      exit(EXIT_FAILURE);
   }
   if( findKey(D->root, k)!=NULL ){
      fprintf(stderr, "Dictionary Error: cannot insert() duplicate keys\n");
      exit(EXIT_FAILURE);
   }
   N = newNode(k, v);
   B = NULL;
   A = D->root;
   while( A!=NULL ){
      B = A;
      if( k<A->key ) A = A->left;
      else A = A->right;
   }
   if( B==NULL ) D->root = N;
   else if( k<B->key ) B->left = N;
   else B->right = N;
   D->numPairs++;
}
```

Dictionary.c

```
/*
*  delete
*  deletes pair with the key k
*  pre: key k currently exists in D, i.e. lookup(D, k)!=UNDEF
*  post: size(D) is decreased by one
*/
void delete(DictionaryRef D, int k){
   NodeRef N, P, S;
   if( D==NULL ){
      fprintf(stderr, "Dictionary Error: calling delete() on NULL DictionaryRef\n");
      exit(EXIT_FAILURE);
   }
   N = findKey(D->root, k);
   if( N==NULL ){
      fprintf(stderr, "Dictionary Error: cannot delete() non-existent key\n");
      exit(EXIT_FAILURE);
   }
   if( N->left==NULL && N->right==NULL ){
      if( N==D->root ){
         D->root = NULL;
         freeNode(&N);
      }else{
         P = findParent(N, D->root);
         if( P->right==N ) P->right = NULL;
         else P->left = NULL;
         freeNode(&N);
      }
   }else if( N->right==NULL ){
      if( N==D->root ){
         D->root = N->left;
         freeNode(&N);
      }else{
         P = findParent(N, D->root);
         if( P->right==N ) P->right = N->left;
         else P->left = N->left;
         freeNode(&N);
      }
   }else if( N->left==NULL ){
      if( N==D->root ){
         D->root = N->right;
         freeNode(&N);
      }else{
         P = findParent(N, D->root);
         if( P->right==N ) P->right = N->right;
         else P->left = N->right;
         freeNode(&N);
      }
   }else{
      S = findLeftmost(N->right);
      N->key = S->key;
      N->value = S->value;
      P = findParent(S, N);
      if( P->right==S ) P->right = S->right;
      else P->left = S->right;
      freeNode(&S);
   }
   D->numPairs--;
}
```

1

2.1

2.2

3

Dictionary.c

```
/*
*  makeEmpty
*  pre: none
*  post: isEmpty(D)
*/
void makeEmpty(DictionaryRef D){
   deleteAll(D->root);
   D->root = NULL;
   D->numPairs = 0;
}

/*
*  printDictionary
*  pre: none
*  post: prints a text representation of D to file pointed to by out.
*/
void printDictionary(DictionaryRef D, FILE* out){
   if( D==NULL ){
      fprintf(stderr, "Dictionary Error: calling printDictionary() on NULL Dictionar
      exit(EXIT_FAILURE);
   }
   printInOrder(D->root, out);
}
```