上机操作

张晓平

2016 年 11 月 15 日

1 二叉树

例 1. 自行构造一棵二叉树，实现二叉树的相关操作。

BiTree.h

```h
#include<stdio.h>
#include<stdlib.h>
#define MAX_NODE  50

typedef int ElemType;
typedef struct node {
  struct node * lchild;
  struct node * rchild;
  ElemType data;
} BTNode, * BTree;

BiTree InitBiTree(BTNode * root);
BTNode * MakeNode(ElemType data, BTNode * lchild, BTNode * rchild);
void FreeNode(BTNode * pnode);
void ClearBiTree(BiTree tree);
void DestroyBiTree(BiTree tree);
int IsEmpty(BiTree tree);
int GetDepth(BiTree tree);
ElemType GetItem(BTNode * pnode);
void SetItem(BTNode * pnode, ElemType item);
BiTree SetLChild(BiTree parent, BiTree lchild);
```

```
BiTree SetRChild(BiTree parent, BiTree rchild);
BiTree GetLChild(BiTree tree);
BiTree GetRChild(BiTree tree);
BiTree InsertChild(BiTree parent, int lr, BiTree child);
void DeleteChild(BiTree parent, int lr);
void PreOrderTraverse(BiTree tree, void(* visit)());
void InOrderTraverse(BiTree tree, void(* visit)());
void PostOrderTraverse(BiTree tree, void(* visit)());
void LevelOrderTraverse(BiTree tree, void(* visit)());
void Print(ElemType item);
```

InitBiTree.c

```
#include"BiTree.h"

/* Creat a new bitree */
BiTree InitBiTree(BTNode * root)
{
  BiTree tree = root;
  return tree;
}
```

MakeNode.c

```
#include"BiTree.h"

/* Generate a node */
BTNode * MakeNode(ElemType item, BTNode * lchild, BTNode * rchild)
{
  BTNode * pnode = (BTNode *) malloc(sizeof(BTNode));
  if (pnode)
  {
    pnode->data = item;
    pnode->lchild = lchild;
    pnode->rchild = rchild;
  }
  return pnode;
```

```
}
```

FreeNode.c

```
#include"BiTree.h"

/* Free a node */
void FreeNode(BTNode * pnode)
{
  if(pnode != NULL)
    free(pnode);
}
```

ClearBiTree.c

```
#include"BiTree.h"

/* Clear a bitree */
void ClearBiTree(BiTree tree)
{
  BTNode * pnode = tree;
  if (pnode->lchild != NULL)
    ClearBiTree(pnode->lchild);

  if (pnode->rchild != NULL)
    ClearBiTree(pnode->rchild);

  FreeNode(pnode);
}
```

DestroyBiTree.c

```
#include"BiTree.h"

/* Destroy a BiTree */
void DestroyBiTree(BiTree tree)
{
  if(tree)
```

```
    ClearBiTree(tree);
}
```

IsEmpty.c

```
#include"BiTree.h"

/* Is a BiTree Empty? */
int IsEmpty(BiTree tree)
{
  if (tree == NULL)
    return 0;
  else
    return 1;
}
```

GetDepth.c

```
#include"BiTree.h"

/* Return a BiTree's depth */
int GetDepth(BiTree tree)
{
  int cd, ld, rd;
  cd = ld = rd = 0;
  if (tree)
  {
    ld = GetDepth(tree->lchild);
    rd = GetDepth(tree->rchild);
    cd = (ld > rd ? ld : rd);
    return cd + 1;
  }
  return 0;
}
```

GetRoot.c

```
#include"BiTree.h"
```

```c
/* Return a BiTree's root */
BiTree GetRoot(BiTree tree)
{
  return tree;
}
```

GetItem.c

```c
#include"BiTree.h"

/* Return a node's value */
ElemType GetItem(BTNode * pnode)
{
  return pnode->data;
}
```

SetItem.c

```c
#include"BiTree.h"

/* Set a node's value */
void SetItem(BTNode * pnode, ElemType item)
{
  pnode->data = item;
}
```

SetLChild.c

```c
#include"BiTree.h"

/* Set Left Child */
BiTree SetLChild(BiTree parent, BiTree lchild)
{
  parent->lchild = lchild;
  return lchild;
}
```

SetRChild.c

```c
#include"BiTree.h"

/* Set right Child */
BiTree SetRChild(BiTree parent, BiTree rchild)
{
  parent->rchild = rchild;
  return rchild;
}
```

GetLChild.c

```c
#include"BiTree.h"

/* Return left child */
BiTree GetLChild(BiTree tree)
{
  if (tree)
    return tree->lchild;
  return NULL;
}
```

GetLChild.c

```c
#include"BiTree.h"

/* Return left child */
BiTree GetLChild(BiTree tree)
{
  if (tree)
    return tree->lchild;
  return NULL;
}
```

InsertChild.c

```c
#include"BiTree.h"
```

```c
/* Insert a new SubBiTree */
BiTree InsertChild(BiTree parent, int lr, BiTree child)
{
  if (parent)
  {
    if (lr == 0 && parent->lchild == NULL)
    {
      parent->lchild = child;
      return child;
    }
    if (lr == 1 && parent->rchild == NULL)
    {
      parent->rchild = child;
      return child;
    }
  }
  return NULL;
}
```

DeleteChild.c

```c
#include"BiTree.h"

/* Delete SubBiTree */
void DeleteChild(BiTree parent, int lr)
{
  if (parent)
  {
    if(lr == 0 && parent->lchild != NULL){
      parent->lchild = NULL;
      /* FreeNode(parent->lchild); */
    }

    if(lr == 1 && parent->rchild != NULL){
      parent->rchild = NULL;
      /* FreeNode(parent->rchild); */
```

```
    }
  }
}
```

PreOrderTraverse.c

```
#include"BiTree.h"

/* PreOrder Traverse a BiTree */
void PreOrderTraverse(BiTree tree, void(* visit)())
{
  BTNode * pnode = tree;
  if (pnode)
  {
    visit(pnode->data);
    PreOrderTraverse(pnode->lchild, visit);
    PreOrderTraverse(pnode->rchild, visit);
  }
}
```

InOrderTraverse.c

```
#include"BiTree.h"

/* InOrder Traverse a BiTree */
void InOrderTraverse(BiTree tree, void(* visit)())
{
  BTNode * pnode = tree;
  if (pnode)
  {
    InOrderTraverse(pnode->lchild, visit);
    visit(pnode->data);
    InOrderTraverse(pnode->rchild, visit);
  }
}
```

PostOrderTraverse.c

```c
#include"BiTree.h"

/* PostOrder Traverse a BiTree */
void PostOrderTraverse(BiTree tree, void(* visit)())
{
  BTNode * pnode = tree;
  if (pnode)
  {
    PostOrderTraverse(pnode->lchild, visit);
    PostOrderTraverse(pnode->rchild, visit);
    visit(pnode->data);
  }
}
```

LevelOrderTraverse.c

```c
#include"BiTree.h"

/* LevelOrder Traverse a BiTree */
void LevelOrderTraverse(BiTree tree, void(* visit)())
{
  BTNode * Queue[MAX_NODE], * pnode = tree;
  int front = 0, rear = 0;
  if (pnode != NULL)
  {
    Queue[++rear] = pnode;
    while (front < rear)
    {
      pnode = Queue[++front];
      visit(pnode->data);
      if (pnode->lchild)
        Queue[++rear] = pnode->lchild;
      if (pnode->rchild)
        Queue[++rear] = pnode->rchild;
    }
  }
```

```
}
```

BiTreeTest.c

```
#include"BiTree.h"

int main(void){
  BTNode * n1 = MakeNode(10, NULL, NULL);
  BTNode * n2 = MakeNode(20, NULL, NULL);
  BTNode * n3 = MakeNode(30, n1, n2);
  BTNode * n4 = MakeNode(40, NULL, NULL);
  BTNode * n5 = MakeNode(50, NULL, NULL);
  BTNode * n6 = MakeNode(60, n4, n5);
  BTNode * n7 = MakeNode(70, NULL, NULL);

  BiTree tree = InitBiTree(n7);
  SetLChild(tree, n3);
  SetRChild(tree, n6);

  printf("Depth␣of␣BiTree␣is␣%d\n", GetDepth(tree));

  printf("PreOrder␣Traverse:\n");
  PreOrderTraverse(tree, Print); printf("\n");

  printf("InOrder␣Traverse:\n");
  InOrderTraverse(tree, Print); printf("\n");

  printf("PostOrder␣Traverse:\n");
  PostOrderTraverse(tree, Print); printf("\n");

  printf("LevelOrder␣Traverse:\n");
  LevelOrderTraverse(tree, Print); printf("\n");

  SetItem(tree, 100);
  printf("Root:␣%d\n", GetItem(tree));
```

```
  DeleteChild(tree, 0);
  printf("PreOrder␣Traverse:\n");
  PreOrderTraverse(tree, Print); printf("\n");

  DestroyBiTree(tree);
  if(IsEmpty(tree))
    printf("BiTree␣is␣empty,␣succeed␣to␣destroy!\n");
}
```

```
Depth of BiTree is 3
PreOrder Traverse:
70 30 10 20 60 40 50
InOrder Traverse:
10 30 20 70 40 60 50
PostOrder Traverse:
10 20 30
Root: 100
PreOrder Traverse:
100 60 40 50
BiTree is empty, succeed to destroy!
```

2 Huffman 树

例 2. 输入一串字符 *(a-z)*，求各字符的 *Huffman* 编码。

Huffman.h

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

typedef char ElemType;

/* Huffman tree's node */
typedef struct HuffNode {
  ElemType data;
```

```
  struct HuffNode * rchild;
  struct HuffNode * lchild;
  int weight;
  ElemType code[20];
} HuffNode, * HuffTree;

/* Queue */
typedef struct QueueNode {
  HuffNode * data;
  struct QueueNode * next;
} QueueNode;

typedef struct {
  QueueNode * front;
  QueueNode * rear;
} Queue;

Queue * Create_Empty_Queue();
int EnterQueue(Queue * head, HuffNode * data);
HuffNode * DeleteQueue(Queue * head);
int Is_Empty_Queue(Queue * head);

int Is_Empty_OrderQueue(Queue * head);
int EnterOrderQueue(Queue * head, HuffNode * p);

HuffNode * Create_Huffman_Tree(Queue * head);
int HuffmanCode(HuffNode * root);

HuffNode * MakeNode(ElemType item, HuffNode * lchild, HuffNode * rchild, int weight);
int GetDepth(HuffTree tree);
```

Huffman.c

```
#include "Huffman.h"

Queue * Create_Empty_Queue()
```

```
{
  QueueNode * QNode;
  Queue * HQueue;

  QNode = (QueueNode *) malloc(sizeof(QueueNode));
  QNode->next = NULL;

  HQueue = (Queue *) malloc(sizeof(Queue));
  HQueue->front = HQueue->rear = QNode;

  return HQueue;
}

int EnterQueue(Queue * head, HuffNode * data)
{
  QueueNode * temp;

  temp = (QueueNode *) malloc(sizeof(QueueNode));
  temp->data = data;
  temp->next = NULL;

  head->rear->next = temp;
  head->rear = temp;

  return 0;
}

HuffNode * DeleteQueue(Queue * head)
{
  QueueNode * temp;

  temp = head->front;
  head->front = temp->next;
  free(temp);
  temp = NULL;
```

```
  return head->front->data;
}

int Is_Empty_Queue(Queue * head)
{
  if(head->front == head->rear)
    return 1;
  else
    return 0;
}

int EnterOrderQueue(Queue * head, HuffNode * p)
{
  QueueNode * m = head->front->next;
  QueueNode * n = head->front;
  QueueNode * temp;

  while(m) {
    if (m->data->weight < p->weight) {
      m = m->next;
      n = n->next;
    } else
      break;
  }

  if(m == NULL){
    temp = (QueueNode *) malloc(sizeof(QueueNode));
    temp->data = p;
    temp->next = NULL;

    n->next = temp;
    head->rear = temp;
    return 0;
  }
```

```
  temp = (QueueNode *) malloc(sizeof(QueueNode));
  temp->data = p;
  n->next = temp;
  temp->next = m;
  return 0;
}

int Is_Empty_OrderQueue(Queue * head)
{
  if(head->front->next->next == NULL)
    return 1;
  return 0;
}



HuffNode * Create_Huffman_Tree(Queue *head)
{
  HuffNode * right, * left, * current;

  while (!Is_Empty_OrderQueue(head)) {
    left  = DeleteQueue(head);
    right = DeleteQueue(head);
    current = (HuffNode *) malloc(sizeof(HuffNode));
    current->weight = left->weight + right->weight;
    current->rchild = right;
    current->lchild = left;
    EnterOrderQueue(head, current);
  }

  return head->front->next->data;
}

//Huffman Code
int HuffmanCode(HuffNode * root)
{
```

```
  HuffNode * current = NULL;
  Queue * queue = Create_Empty_Queue();
  EnterQueue(queue, root);

  while(!Is_Empty_Queue(queue)){
    current = DeleteQueue(queue);

    if(current->rchild == NULL && current->lchild == NULL)
      printf("%c:%d␣%s\n", current->data, current->weight, current->code);

    if(current->lchild){
      strcpy(current->lchild->code, current->code);
      strcat(current->lchild->code, "0");
      EnterQueue(queue, current->lchild);
    }

    if(current->rchild){
      strcpy(current->rchild->code, current->code);
      strcat(current->rchild->code, "1");
      EnterQueue(queue, current->rchild);
    }
  }
  return 0;
}

/* Generate a node */
HuffNode * MakeNode(ElemType item, HuffNode * lchild, HuffNode * rchild, int weight)
{
  HuffNode * pnode = (HuffNode *) malloc(sizeof(HuffNode));
  if (pnode){
    pnode->data = item;
    pnode->lchild = lchild;
    pnode->rchild = rchild;
    pnode->weight = weight;
    /* pnode->code = code; */
```

```
  }
  return pnode;
}

/* Return a BiTree's depth */
int GetDepth(HuffTree tree)
{
  int cd, ld, rd;
  cd = ld = rd = 0;
  if(tree) {
    ld = GetDepth(tree->lchild);
    rd = GetDepth(tree->rchild);
    cd = (ld > rd ? ld : rd);
    return cd+1;
  }else
    return 0;
}
```

HuffmanTest.c

```
// input characters a-g
#include "Huffman.h"
int main(void){
  Queue * head;
  HuffNode * root;
  HuffNode * node[100];
  ElemType ch, cc[100];
  int weight[100] = {0};
  int i, k = 0;
  printf("input character:\n");
  while(1) {
    scanf("%c", &ch);
    if(ch == '\n'){
      break;
    }
    else {
```

```
      cc[k++] = ch;
    }
  }

  for(i = 0; i < k; i++)
      weight[cc[i]-'a']++;

  k = 0;
  for(i = 0; i < 7; i++){
    if(weight[i] > 0) {
      node[k++] = MakeNode('a'+i, NULL, NULL, weight[i]);
    }
  }

  head = Create_Empty_Queue();
  for(i = 0; i < k; i++)
    EnterOrderQueue(head, node[i]);

  root = Create_Huffman_Tree(head);
  printf("\nDepth␣of␣Huffman␣Tree␣is␣%d\n", GetDepth(root));

  printf("\nHuffman␣Codes␣are:\n");
  HuffmanCode(root);
}
```

运行结果

```
input character:
aaaabbbcccddddddeeeeeffffffgggggggg

Depth of Huffman Tree is 5

Huffman Codes are:
d:6 00
g:7 01
a:4 100
```

```
e:5 101
f:6 111
c:3 1100
b:3 1101
```