Algorithm Report2

2015313254 노인호

November 7th 2019

1 Environmnet

코드는 다음과 같은 환경에서 실행되었다.

- Ubuntu 16.04 LTS 64bit
- gcc 5.4.0

만약 코드가 깨져서 잘 안보인다면 다음 url를 가진 사이트를 참고하면 된다.

- https://gist.github.com/nosy0411/e1921c02b7142979c5ce1a4ae254fae9

2 Algo-1

2.1 problem

Construct the open address hash table according to the following description.

(1) Print the contents of the hash table for above three different hash functions

(2) Print the average number of probes for the three different hash functions.

(3) What is size of the largest cluster for each of the three different hash functions.

2.2 source code

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

#define max(X, Y) ((X) > (Y) ? (X) : (Y))
#define randomize() srand(time(NULL))
#define random(n) (rand()%(n)+97)
#define SAMPLE 26
#define DM 37
#define SL 30

int hashtable_l[DM];
int hashtable_q[DM];
int hashtable_d[DM];
int keyarray[SL];


int hash(int key){
  return key % DM;
}

int hash2(int key){
  return (1+(key%(DM-1)));
}

int linear_probing(int key) {
  int i, bucket, count=0;
  for (i=0;i<DM;i++){

    bucket=hash(hash(key)+i);
```

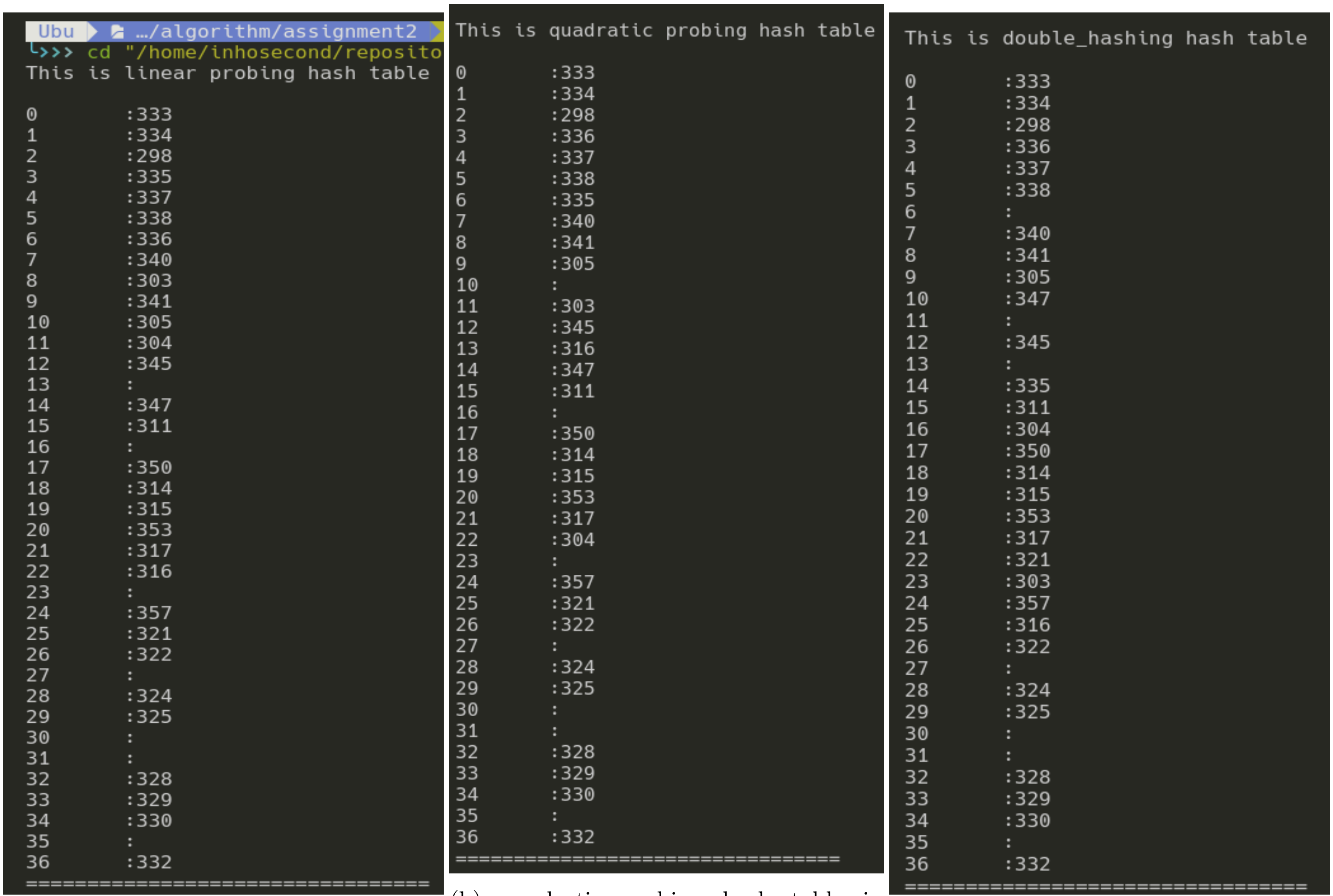
```c
    count++;
    if(hashtable_l[bucket]==0){
      hashtable_l[bucket]=key;
      break;
    }
  }
  return count;
}

int quadratic_probing(int key) {
  int i, bucket, count=0;
  for (i=0;i<DM; i++){

    bucket=hash(hash(key)+1*i+3*i*i);
    count++;
    if(hashtable_q[bucket]==0){
      hashtable_q[bucket]=key;
      break;
    }
  }
  return count;
}

int double_hashing(int key) {
  int i, bucket, count=0;
  for (i=0;i<DM; i++){

    bucket=hash(hash(key)+i*hash2(key));
    count++;
    if(hashtable_d[bucket]==0){
      hashtable_d[bucket]=key;
      break;
    }
  }
  return count;
}

void display(int A[DM]){

  int i;
  for(i=0;i<DM; i++){
    if(A[i]==0){
      printf("%d\t:%c\n",i,' ');
    }
    else{
      printf("%d\t:%d\n",i,A[i]);
    }
  }
}

int clustering(int A[DM]){

  int i,count=0,saved=0;
  for(i=0;i<DM; i++){
    if(A[i]!=0){
      count++;
      if (i==DM-1){
        saved=max(saved,count);
        count=0;
      }
    }
    else{
      saved=max(saved,count);
      count=0;
    }
  }
  return saved;
}

void generate(){
```

```
    int i,check,count=0;

    randomize();
    while(count<SL){

        int sum_key=0;
        int flag=0;

        for(i=0;i<3;i++){
            check = random(SAMPLE);
            sum_key +=check;
        }
        for(i=0;i<count+1;i++){
            if(sum_key==keyarray[i]){
                count--;
                flag=1;
                break;
            }
        }

        if (flag==0){
            keyarray[count]=sum_key;
        }
        count++;
    }
}

int main(){

    int i;

    memset(hashtable_l, 0, sizeof(hashtable_l));
    memset(hashtable_q, 0, sizeof(hashtable_q));
    memset(hashtable_d, 0, sizeof(hashtable_d));
    memset(keyarray, 0, sizeof(keyarray));

    generate();

    float probe_l=0, probe_q=0, probe_d=0;

    for(i=0;i<SL;i++){
        probe_l+=linear_probing(keyarray[i]);
        probe_q+=quadratic_probing(keyarray[i]);
        probe_d+=double_hashing(keyarray[i]);
    }

    printf("This is linear probing hash table\n\n");
    display(hashtable_l);
    printf("==================================\n\n");
    printf("This is quadratic probing hash table\n\n");
    display(hashtable_q);
    printf("==================================\n\n");
    printf("This is double_hashing hash table\n\n");
    display(hashtable_d);
    printf("==================================\n\n");

    printf("The average number of probes for linear probing = %0.3f\n",probe_l/SL);
    printf("The average number of probes for quadratic probing = %0.3f\n",probe_q/SL);
    printf("The average number of probes for double hashing = %0.3f\n",probe_d/SL);
    printf("\n");
    printf("The largest cluster of linear probing = %d\n",clustering(hashtable_l));
    printf("The largest cluster of quadratic probing = %d\n",clustering(hashtable_q));
    printf("The largest cluster of double hashing = %d\n",clustering(hashtable_d));
}
```

Listing 1: algo1.c

2.3 result of hash table

(1) print three hash tables with position and value.



```
Ubu  …/algorithm/assignment2
└>>> cd "/home/inhosecond/reposito
This is linear probing hash table

0       :333
1       :334
2       :298
3       :335
4       :337
5       :338
6       :336
7       :340
8       :303
9       :341
10      :305
11      :304
12      :345
13      :
14      :347
15      :311
16      :
17      :350
18      :314
19      :315
20      :353
21      :317
22      :316
23      :
24      :357
25      :321
26      :322
27      :
28      :324
29      :325
30      :
31      :
32      :328
33      :329
34      :330
35      :
36      :332
=================================
```

(a) linear probing hash table in algo1.c

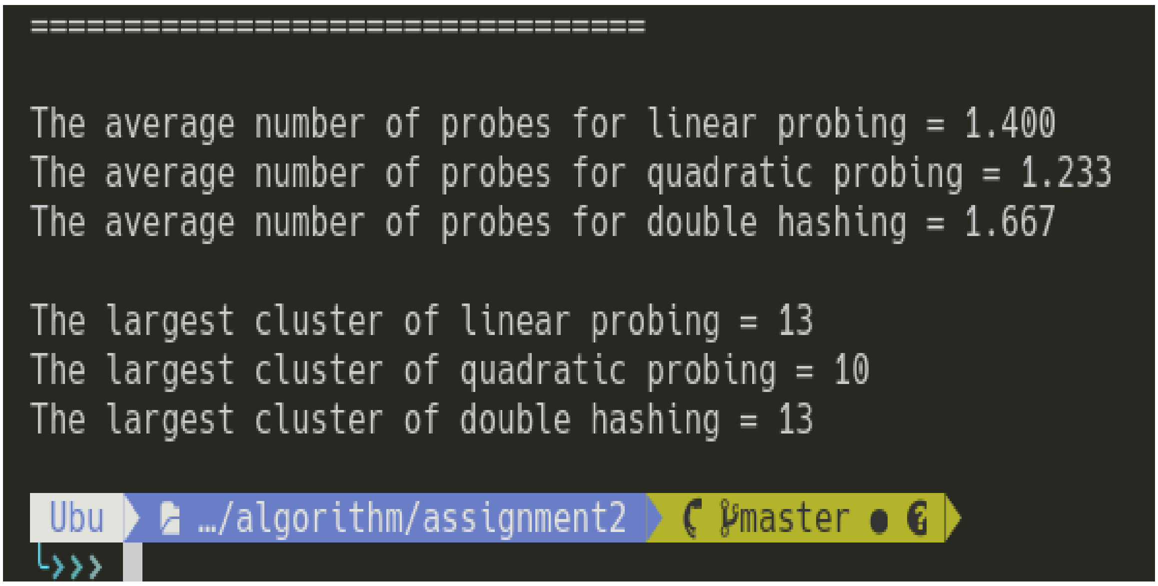
```
This is quadratic probing hash table

0       :333
1       :334
2       :298
3       :336
4       :337
5       :338
6       :335
7       :340
8       :341
9       :305
10      :
11      :303
12      :345
13      :316
14      :347
15      :311
16      :
17      :350
18      :314
19      :315
20      :353
21      :317
22      :304
23      :
24      :357
25      :321
26      :322
27      :
28      :324
29      :325
30      :
31      :
32      :328
33      :329
34      :330
35      :
36      :332
=================================
```

(b) quadratic probing hash table in algo1.c

```
This is double_hashing hash table

0       :333
1       :334
2       :298
3       :336
4       :337
5       :338
6       :
7       :340
8       :341
9       :305
10      :347
11      :
12      :345
13      :
14      :335
15      :311
16      :304
17      :350
18      :314
19      :315
20      :353
21      :317
22      :321
23      :303
24      :357
25      :316
26      :322
27      :
28      :324
29      :325
30      :
31      :
32      :328
33      :329
34      :330
35      :
36      :332
=================================
```

(c) double hashing hash table in algo1.c

(2) the average probe num and (3) size of the largest cluster



```
=====================================

The average number of probes for linear probing = 1.400
The average number of probes for quadratic probing = 1.233
The average number of probes for double hashing = 1.667

The largest cluster of linear probing = 13
The largest cluster of quadratic probing = 10
The largest cluster of double hashing = 13

Ubu  …/algorithm/assignment2  master
└>>>
```

Figure 2: average probe num and size of largest cluster in algo1.c

3 Algo-2

3.1 problem

Problem about Binary Search Tree

3.2 source code

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <math.h>
#include <limits.h>

#define min(X, Y) ((X) < (Y) ? (X) : (Y))
#define randomize() srand(time(NULL))
#define random(n) (rand()%(n))
#define SAMPLE 50
#define INTERVAL 5
#define NUM 20

int A[NUM]={0};
int node_exist=1;

typedef enum { false, true } bool;

typedef int Key;

typedef struct _BSTNode {
  Key key;
  struct _BSTNode* left;
  struct _BSTNode* right;
  struct _BSTNode* parent;
} BSTNode;

// Create a new node.
BSTNode* CreateNode(Key key) {
  BSTNode* node = (BSTNode*)malloc(sizeof(BSTNode));
  node->key = key;
  node->left = NULL;
  node->right = NULL;
  node->parent = NULL;

  return node;
}
// Destroy a node.
  void DestroyNode(BSTNode* node) {
  free(node);
}
// Verify if the tree is a binary search tree.
// Initialize the minimum and maximum as INT_MIN and INT_MAX
bool Verify(BSTNode* root, int min, int max) {
  if (root != NULL) {
    // Return false if this node violates the min/max constraints.
    if (root->key < min || root->key > max)
      return false;
    else
      // Check the subtrees recursively tightening the min/max constraints.
      return Verify(root->left, min, root->key) && Verify(root->right, root->key, max);
  } else
    return true;  // an empty tree is BST.
}

// Search and display an item in BST.
BSTNode* TREE_SEARCH(BSTNode* root, Key key) {
  BSTNode* cur = root;
  while (cur != NULL) {
    printf("%d ",cur->key);
```

```
    if (cur->key == key){
      break;
    }
    else if (cur->key > key)
      cur = cur->left;
    else
      cur = cur->right;
  }

  return cur;
}
// Search an item in BST.
BSTNode* TREE_FILTER(BSTNode* root, Key key) {
  BSTNode* cur = root;
  while (cur != NULL) {
    if (cur->key == key){
      break;
    }
    else if (cur->key > key)
      cur = cur->left;
    else
      cur = cur->right;
    }

  return cur;
}
// Check an item
BSTNode* TREE_CHECK(BSTNode* root, Key key) {
  BSTNode* cur = root;
  BSTNode* temp=NULL;
  int check=0;
  while (cur != NULL) {
    if (cur->key == key){
      check=1;
      break;
    }
    else if (cur->key > key){
      temp=cur;
      cur = cur->left;
    }
    else{
      temp=cur;
      cur = cur->right;
    }
  }
  if(check==1)
    return cur;
  else{
    node_exist=0;
    return temp;
  }
}

// Insert an item to BST.
void TREE_INSERT(BSTNode* root, Key key) {
  BSTNode *check = TREE_FILTER(root,key);
  if(check==NULL){
    BSTNode *y=NULL;
    BSTNode *x=root;
    BSTNode *z=CreateNode(key);
    BSTNode *temp=NULL;

    while(x!=NULL){
      x->parent=temp;
      y=x;

      if((z->key)<x->key){
        temp=x;
        x=x->left;
      }
```

```
      else{
        temp=x;
        x=x->right;
      }
    }
    z->parent=y;
    if(y==NULL)
      root=z;
    else if ((z->key)<(y->key))
      y->left=z;
    else
      y->right=z;
  }
}

BSTNode *TREE_MIN(BSTNode* node){
  BSTNode *cur = node;
  while(cur->left!=NULL){
    cur=cur->left;
  }

  return cur;
}

BSTNode *TREE_MAX(BSTNode* node){
  BSTNode *cur = node;
  while(cur->right!=NULL){
    cur=cur->right;
  }

  return cur;
}

void TRANSPLANT(BSTNode* root, BSTNode* u, BSTNode* v){
  if(u->parent==NULL){
    root=v;
  }
  else if(u==u->parent->left){
    u->parent->left=v;
  }
  else{
    u->parent->right=v;
  }
  if(v!=NULL){
    v->parent=u->parent;
  }
}

// Remove an item from BST.
void TREE_DELETE(BSTNode* root, BSTNode* z ) {

  if(z->left==NULL){
    TRANSPLANT(root, z,z->right);
  }
  else if (z->right==NULL){
    TRANSPLANT(root, z,z->left);
  }
  else {
    BSTNode *y = TREE_MIN(z->right);
    if(y->parent!=z){
      TRANSPLANT(root,y,y->right);
      y->right=z->right;
      y->right->parent=y;
    }
    TRANSPLANT(root,z,y);
    y->left=z->left;
    y->left->parent=y;
  }
  free(z);
```

```
}

//Display a tree.
void PRINT_BST(BSTNode* root, int dist) {
  // Base case
  if (root == NULL)
    return;

  // Increase distance between levels
  dist += INTERVAL;

  // Process right child first
  PRINT_BST(root->right, dist);

  // Print current node after space
  // count
  printf("\n");
  for (int i = INTERVAL; i < dist; i++)
    printf(" ");
  printf("%d\n", root->key);

  // Process left child
  PRINT_BST(root->left, dist);
}

void generate(){

  randomize();
  for(int i=0;i<NUM;i++){
    while(1){
      int check=0;
      A[i]=random(SAMPLE);
      for(int j=0;j<i;j++){
        if(A[j]==A[i]){
          check=1;
          break;
        }
      }
      if(check==1){
        continue;
      }
      else{
        break;
      }
   }

  }
}

BSTNode *TREE_SUCCESSOR(BSTNode *node, BSTNode *root){
  BSTNode *x=node;
  if (x->right!=NULL){
    return TREE_MIN(x->right);
  }
  if(x==TREE_MAX(root)){
    return x;
  }
  BSTNode *y=x->parent;

  while(y!=NULL && x==y->right){
    x=y;
    y=y->parent;
  }
  return y;
}

BSTNode *TREE_PREDECESSOR(BSTNode *node, BSTNode *root){
  BSTNode *x=node;
  if (x->left!=NULL){
    return TREE_MAX(x->left);
```

```c
    }
    if(x==TREE_MIN(root)){
        return x;
    }
    BSTNode *y=x->parent;
    while(y!=NULL && x==y->left){
        x=y;
        y=y->parent;
    }
    return y;
}

BSTNode* TREE_NEAREST_NEIGHBOR(BSTNode* root, int key){
    BSTNode* y=NULL;
    BSTNode* ys=NULL;
    BSTNode* yp=NULL;
    y=TREE_CHECK(root,key);
    ys=TREE_SUCCESSOR(y,root);
    yp=TREE_PREDECESSOR(y,root);
    int miny, minys, minyp;
    miny=abs((y->key)-key);
    minys=abs((ys->key)-key);
    minyp=abs((yp->key)-key);

    if(min(miny,minys)==miny){
        if(min(miny,minyp)==miny)
            return y;
        else
            return yp;
    }
    else{
        if(min(minys,minyp)==minys)
            return ys;
        else
            return yp;
    }

}

// Clear a tree.
void ClearTree(BSTNode* root) {
    if (root != NULL) {
        ClearTree(root->left);
        ClearTree(root->right);
        free(root);
    }
}

int main(){

    int i;

    generate();
    BSTNode *T = CreateNode(A[0]);
    for(i=0;i<NUM;i++){
        printf("%d ",A[i]);
    }
    printf("\n==================================================\n");
    for(i=1;i<NUM;i++){
       TREE_INSERT(T,A[i]);
    }
    PRINT_BST(T,0);
    printf("==================================================\n\n");
    BSTNode *node = TREE_SEARCH(T,10);
    printf("\n");
    if (node==NULL){
        printf("Search 10 but not exist\n\n");
    }
    else{
        printf("Successfully searched 10\n\n");
```

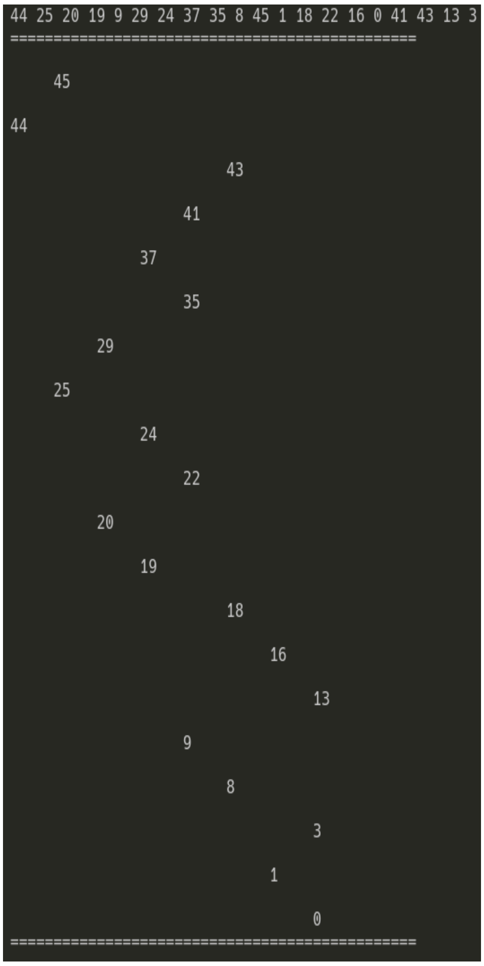
```
}
node = TREE_SEARCH(T,9);
printf("\n");
if (node==NULL){
  printf("Search 9 but not exist\n\n");
}
else{
  printf("Successfully searched 9\n\n");
}
node = TREE_SEARCH(T,15);
  printf("\n");
if (node==NULL){
  printf("Search 15 but not exist\n\n");
}
else{
  printf("Successfully searched 15\n\n");
}
printf("==================================================\n");
printf("Nearest neighbor of 5 is %d\n",TREE_NEAREST_NEIGHBOR(T,5)->key);
printf("Nearest neighbor of 9 is %d\n",TREE_NEAREST_NEIGHBOR(T,9)->key);
printf("Nearest neighbor of 17 is %d\n",TREE_NEAREST_NEIGHBOR(T,17)->key);
printf("==================================================\n");


node = TREE_CHECK(T,6);
if(node_exist==1){
  printf("Try to insert 6 but there is already node 6 in tree\n");
}
else{
  TREE_INSERT(T,6);
  if(Verify(T,TREE_MIN(T)->key,TREE_MAX(T)->key)==true){
    PRINT_BST(T,0);
    printf("\n==================================================\n");
    printf("Inserted 6\n");
  }
  else{
    printf("This is not Binary Search Tree\n");
  }
}
printf("==================================================\n");
node_exist=1;
node = TREE_CHECK(T,29);
if(node_exist==1){
  printf("Try to insert 29 but there is already node 29 in tree\n");
}
else{
  TREE_INSERT(T,29);
  if(Verify(T,TREE_MIN(T)->key,TREE_MAX(T)->key)==true){
    PRINT_BST(T,0);
    printf("\n==================================================\n");
    printf("Inserted 29\n");
  }
  else{
    printf("This is not Binary Search Tree\n");
  }
}

printf("==================================================\n");
node_exist=1;
node = TREE_CHECK(T,17);
if(node_exist==1){
  printf("Try to insert 17 but there is already node 17 in tree\n");
}
else{
  TREE_INSERT(T,17);
  if(Verify(T,TREE_MIN(T)->key,TREE_MAX(T)->key)==true){
    PRINT_BST(T,0);
    printf("\n==================================================\n");
    printf("Inserted 17\n");
  }
```

```
    else{
      printf("This is not Binary Search Tree\n");
    }
  }
  printf("================================================\n");
  node_exist=1;
  node = TREE_CHECK(T,21);
  if(node_exist==1){
    printf("Try to insert 21 but there is already node 21 in tree\n");
  }
  else{
    TREE_INSERT(T,21);
    if(Verify(T,TREE_MIN(T)->key,TREE_MAX(T)->key)==true){
      PRINT_BST(T,0);
      printf("\n================================================\n");
      printf("Inserted 21\n");
    }
    else{
      printf("This is not Binary Search Tree\n");
    }
  }
  node_exist=1;
  printf("================================================\n\n\n\n");

  node = TREE_CHECK(T,6);
  if (node_exist==1){
    TREE_DELETE(T,node);
    if(Verify(T,TREE_MIN(T)->key,TREE_MAX(T)->key)==true){
      PRINT_BST(T,0);
      printf("\n================================================\n");
      printf("Deleted 6\n");
    }
    else{
      printf("This is not Binary Search Tree\n");
    }
    printf("================================================\n");
  }
  else{
    printf("Try to delete 6 but node 6 does not exist in tree\n");
  }
  printf("================================================\n");

  node = TREE_CHECK(T,17);
  if (node_exist==1){
    TREE_DELETE(T,node);
    if(Verify(T,TREE_MIN(T)->key,TREE_MAX(T)->key)==true){
      PRINT_BST(T,0);
      printf("\n================================================\n");
      printf("Deleted 17\n");
    }
    else{
      printf("This is not Binary Search Tree\n");
    }
    printf("================================================\n");
  }
  else{
    printf("Try to delete 17 but node 17 does not exist in tree\n");
  }

  node = TREE_CHECK(T,21);
  if (node_exist==1){
    TREE_DELETE(T,node);
    if(Verify(T,TREE_MIN(T)->key,TREE_MAX(T)->key)==true){
      PRINT_BST(T,0);
      printf("\n================================================\n");
      printf("Deleted 21\n");
    }
    else{
      printf("This is not Binary Search Tree\n");
    }
```

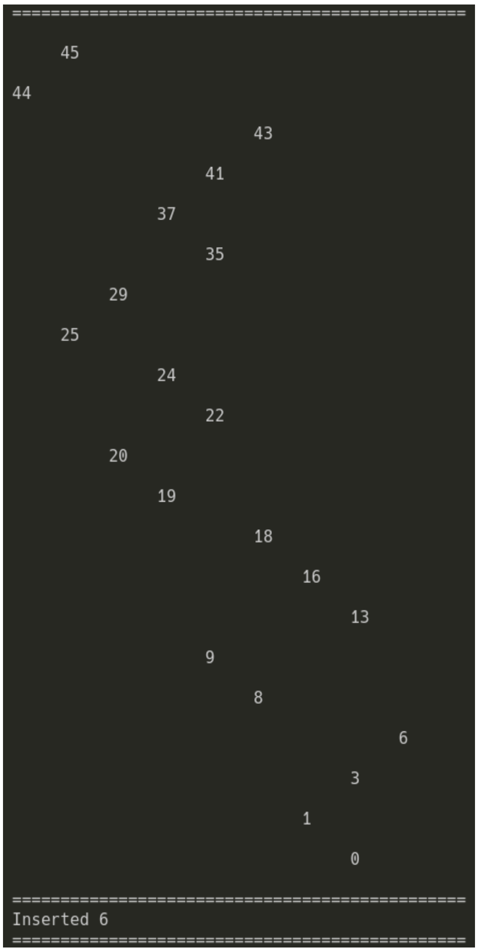
```
    printf("==================================================\n");
  }
  else{
    printf("Try to delete 21 but node 21 does not exist in tree\n");
  }

  node = TREE_CHECK(T,7);
  if (node_exist==1){
    TREE_DELETE(T,node);
    if(Verify(T,TREE_MIN(T)->key,TREE_MAX(T)->key)==true){
      PRINT_BST(T,0);
      printf("\n==================================================\n");
      printf("Deleted 7\n");
    }
    else{
      printf("This is not Binary Search Tree\n");
    }
    printf("==================================================\n");
  }
  else{
    printf("Try to delete 7 but node 7 does not exist in tree\n");
  }
  ClearTree(T);
}
```

Listing 2: algo2.c

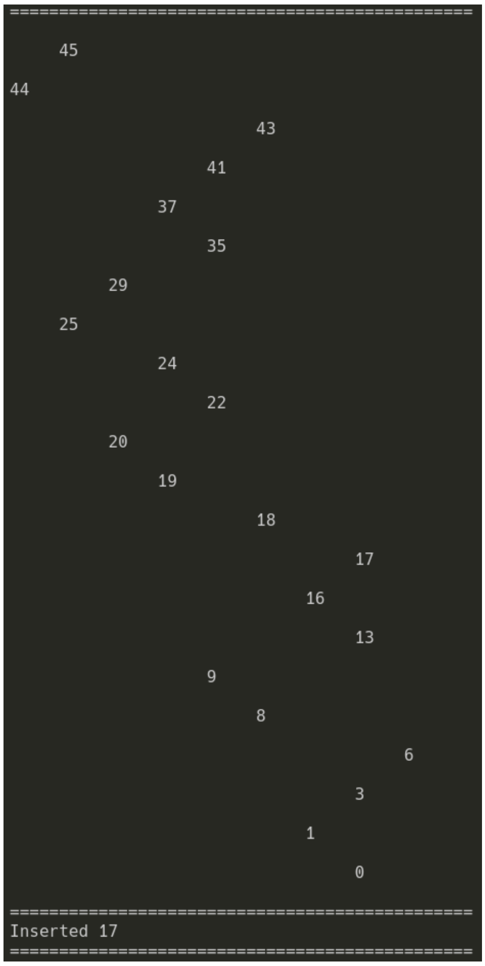
3.3 result of BST



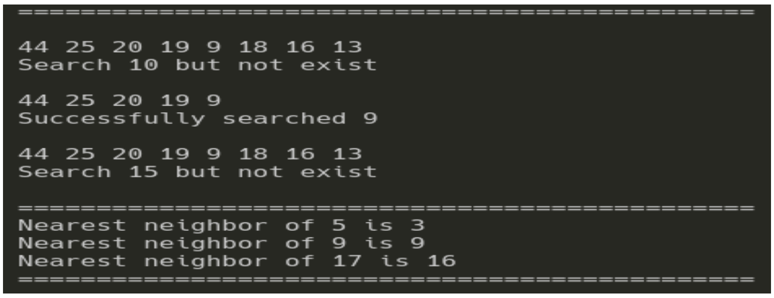
(a) BST constructed from A[20] in algo2.c



(b) 6 inserted in BST in algo2.c

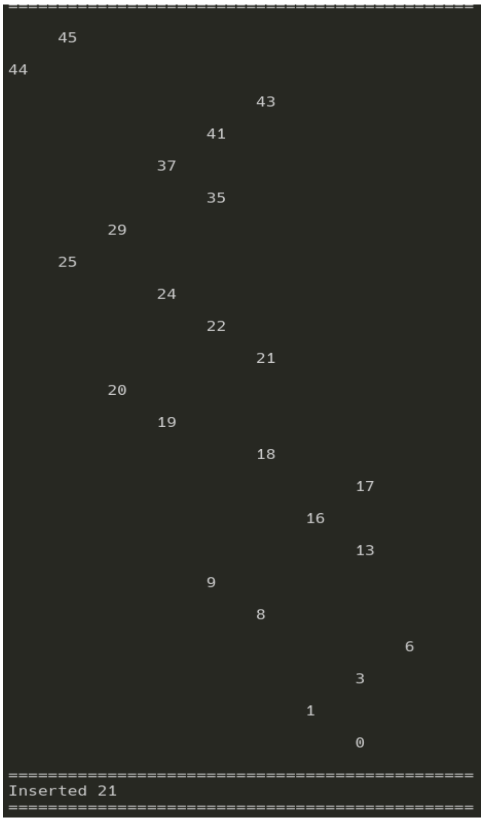


(c) 17 inserted in BST in algo2.c

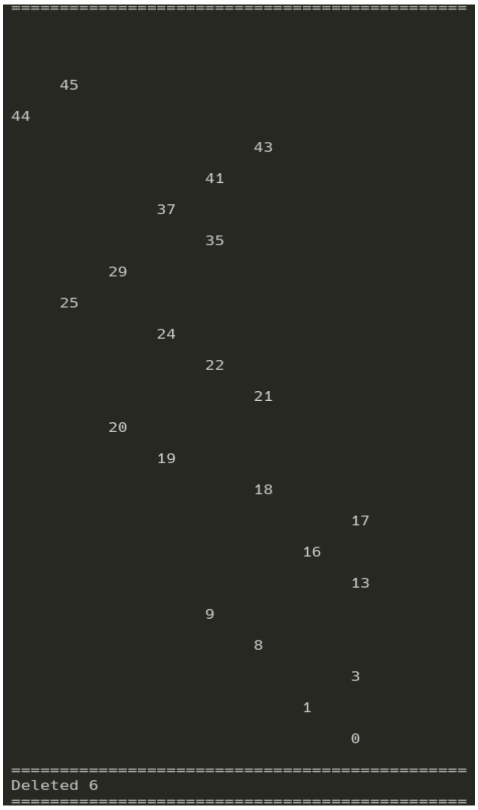


(d) Search and Nearest Neighbors in algo2.c

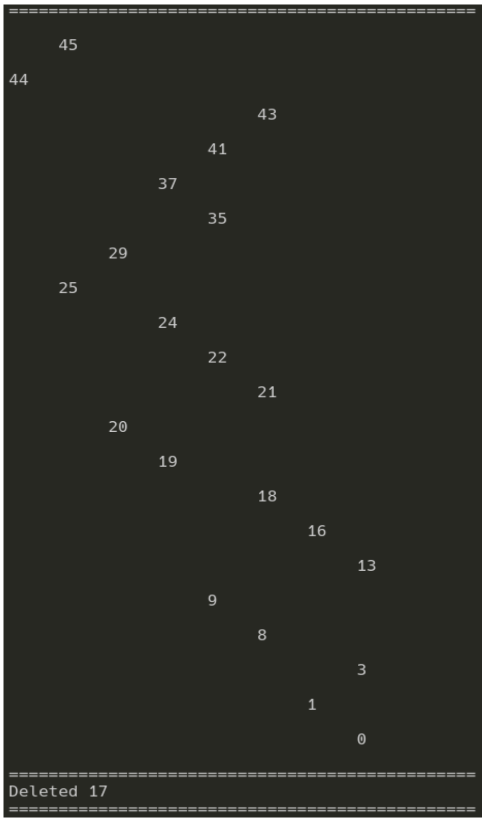
(e) 29 already exist in BST in algo2.c



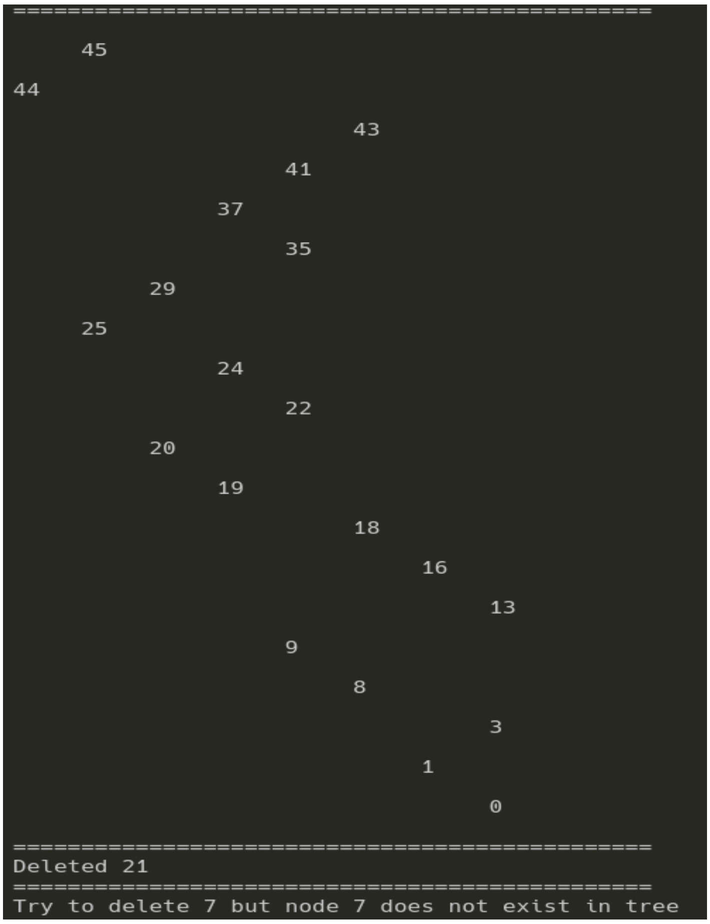
(a) 21 inserted in BST in algo2.c



(b) 6 deleted in BST in algo2.c



(c) 17 deleted in BST in algo2.c



(a) 21 deleted and 7 does not exist in BST in algo2.c

4 Algo-3

4.1 problem

Problem about Red black tree

4.2 source code

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

#define max(X, Y) ((X) > (Y) ? (X) : (Y))

#define randomize() srand(time(NULL))
#define random(n) (rand()%(n))
#define SAMPLE 50
#define COUNT 5
#define NUM 20

int A[NUM]={0};

typedef int Key;
typedef struct _node {
  // RED-true, BLACK-false
  bool color;
  Key key;
  struct _node *left;
  struct _node *right;
  struct _node *parent;
} node;


node *CreateNode(Key key) {
  node *n = (node *)malloc(sizeof(node));
  n->left = NULL;
  n->right = NULL;
  n->parent = NULL;
  n->key = key;

  n->color = true;
  return n;
}

node *TREE_MIN(node* n){
  node *cur = n;
  while(cur->left!=NULL){
    cur=cur->left;
  }

  return cur;
}

// Search an item in BST.
node* RB_SEARCH(node* root, Key key) {
  node* cur = root;
  while (cur != NULL) {
    if (cur->key == key){
      break;
    }
    else if (cur->key > key)
      cur = cur->left;
    else
      cur = cur->right;
  }
```

```
  return cur;
}

void *rotateLeft(node** root, node** x) {
  // printf("left rotation!!\n");
  node *y = (*x)->right;
  if(y==NULL){
    return;
  }

  (*x)->right = y->left;
  if((*x)->right!=NULL){
    (*x)->right->parent=(*x);
  }

  if(y->left!=NULL){
    y->left->parent=(*x);
  }
  y->parent=(*x)->parent;

  if((*x)->parent==NULL){
    *root=y;
  }
  else if((*x)==(*x)->parent->left){
    (*x)->parent->left=y;
  }
  else{
    (*x)->parent->right=y;
  }
  y->left = (*x);
    (*x)->parent = y;
  }

void *rotateRight(node** root, node** x) {
  // printf("right rotation!!\n");
  node *y = (*x)->left;
  if(y==NULL){
    return;
  }

  (*x)->left = y->right;
  if((*x)->left!=NULL){
    (*x)->left->parent=(*x);
  }

  if(y->right!=NULL){
    y->right->parent=(*x);
  }
  y->parent=(*x)->parent;

  if((*x)->parent==NULL){
    *root=y;
  }
  else if((*x)==(*x)->parent->right){
    (*x)->parent->right=y;
  }
  else{
    (*x)->parent->left=y;
  }
  y->right = (*x);
  (*x)->parent = y;
}

void RB_INSERT_FIXUP(node** root, node** z){
  node *y=NULL;
  node *p=NULL, *gp=NULL;

  while(((*z)!=*root) && ((*z)->color!= false) && (*z)->parent->color==true){
    p=(*z)->parent;
    gp=(*z)->parent->parent;
```

```
    if(p==gp->left){

      y=gp->right;

      if(y!=NULL && y->color==true){
        p->color=false;
        y->color=false;
        gp->color=true;
        (*z)=gp;
      }

      else if((*z)==p->right){
        rotateLeft(root,&p);
        (*z)=p;
        p=(*z)->parent;
      }
      else{
        rotateRight(root,&gp);
        p->color=false;
        gp->color=true;
        (*z)=p;
      }
    }

    else{

      y=gp->left;
      if(y!=NULL && y->color==true){
        p->color=false;
        y->color=false;
        gp->color=true;
        (*z)=gp;
      }

      else if((*z)==(*z)->parent->left){
        rotateRight(root,&p);
        (*z)=p;
        p=(*z)->parent;
      }
      else{
        rotateLeft(root,&gp);
        p->color=false;
        gp->color=true;
        (*z)=p;
      }
    }
  }
  (*root)->color=false;
}

node* RB_INSERT(node* root, Key key){
  node *z=CreateNode(key);
  node *y=NULL;
  node *x=root;

  while(x!=NULL){
    y=x;
    if ((z->key)<(x->key)){
      x=x->left;
    }
    else{
      x=x->right;
    }
  }
  z->parent=y;
  if (y==NULL){
    root=z;
  }
  else if ((z->key)<(y->key)){
    y->left=z;
```

```
    }
    else{
        y->right=z;
    }

    RB_INSERT_FIXUP(&root,&z);

    return root;
}

void RB_TRANSPLANT(node** root, node** u, node** v){
    if((*u)->parent==NULL){
        *root=*v;
    }
    else if((*u)==(*u)->parent->left){
        (*u)->parent->left=*v;
    }
    else{
        (*u)->parent->right=*v;
    }
    if((*v)!=NULL){
        (*v)->parent=(*u)->parent;
    }
}

void RB_DELETE_FIXUP(node** root, node** x){
    node *w;
    while(((*x)!=(*root)) && (((*x)->color)==false)){
        if ((*x)==(*x)->parent->left){
            w=(*x)->parent->right;

            if(w->color==true){
                w->color=false;
                (*x)->parent->color=true;
                rotateLeft(root,&((*x)->parent));
                w=(*x)->parent->right;
            }

            if(w->left->color==false && w->right->color==false){
                w->color=true;
                (*x)=(*x)->parent;
            }

            else if (w->right->color==false && w->left->color==true){
                w->left->color=false;
                w->color=true;
                rotateRight(root,&w);
                w=(*x)->parent->right;
            }
            else {
                w->color=(*x)->parent->color;
                (*x)->parent->color=false;
                w->right->color=false;
                rotateLeft(root,&((*x)->parent));
                (*x)=*root;
            }
        }

        else{
            w=(*x)->parent->left;

            if(w->color==true){
                w->color=false;
                (*x)->parent->color=true;
                rotateRight(root,&((*x)->parent));
                w=(*x)->parent->left;
            }

            if(w->left->color==false && w->right->color==false){
                w->color=true;
```

```
        (*x)=(*x)->parent;
      }

      else if (w->left->color==false && w->right->color==true){
        w->right->color=false;
        w->color=true;
        rotateLeft(root,&w);
        w=(*x)->parent->left;
      }
      else {
        w->color=(*x)->parent->color;
        (*x)->parent->color=false;
        w->left->color=false;
        rotateRight(root,&((*x)->parent));
        (*x)=*root;
      }
    }
  }
  (*x)->color=false;
}

node* RB_DELETE(node* root, Key key){
  node *z=RB_SEARCH(root,key);
  node *x;
  node *y=z;
  bool yo;
  yo=y->color;

  if(z->left==NULL){
    x=z->right;
    RB_TRANSPLANT(&root,&z,&(z->right));
  }
  else if(z->right==NULL){
    x=z->left;
    RB_TRANSPLANT(&root,&z,&(z->left));
  }
  else{
    y=TREE_MIN(z->right);
    yo=y->color;
    x=y->right;

    if(y->parent==z){
      x->parent=y;
    }
    else{
      RB_TRANSPLANT(&root,&y,&(y->right));
      y->right=z->right;
      y->right->parent=y;
    }
    RB_TRANSPLANT(&root, &z, &y);
    y->left=z->left;
    y->left->parent=y;
    y->color=z->color;
  }
  if(yo==false){
    RB_DELETE_FIXUP(&root,&x);
  }
  free(z);
  return root;
}

//Display a tree.
void PRINT_RBT(node *root, int space) {
  // Base case
  if (root == NULL)
    return;

  // Increase distance between levels
  space += COUNT;
```

```
  // Process right child first
  PRINT_RBT(root->right, space);

  // Print current node after space
  // count
  printf("\n");
  for (int i = COUNT; i < space; i++)
    printf(" ");
  if (root->color == true) {
    printf("%d[R]\n", root->key);
  } else {
    printf("%d[B]\n", root->key);
  }
  // Process left child
  PRINT_RBT(root->left, space);
}

void generate(){

  randomize();
  for(int i=0;i<NUM;i++){
    while(1){
      int check=0;
      A[i]=random(SAMPLE);
      for(int j=0;j<i;j++){
        if(A[j]==A[i]){
          check=1;
          break;
        }
      }
      if(check==1){
        continue;
      }
      else{
        break;
      }
    }

  }
}

// Insert an item to BST.
void TREE_INSERT(node* root, Key key) {
  node *y=NULL;
  node *x=root;
  node *z=CreateNode(key);
  node *temp=NULL;

  while(x!=NULL){
    x->parent=temp;
    y=x;

    if(x->key==key){
      return 0;
    }

    if((z->key)<x->key){
      temp=x;
      x=x->left;
    }

    else{
      temp=x;
      x=x->right;
    }
  }
  z->parent=y;
  if(y==NULL)
    root=z;
  else if ((z->key)<(y->key))
```

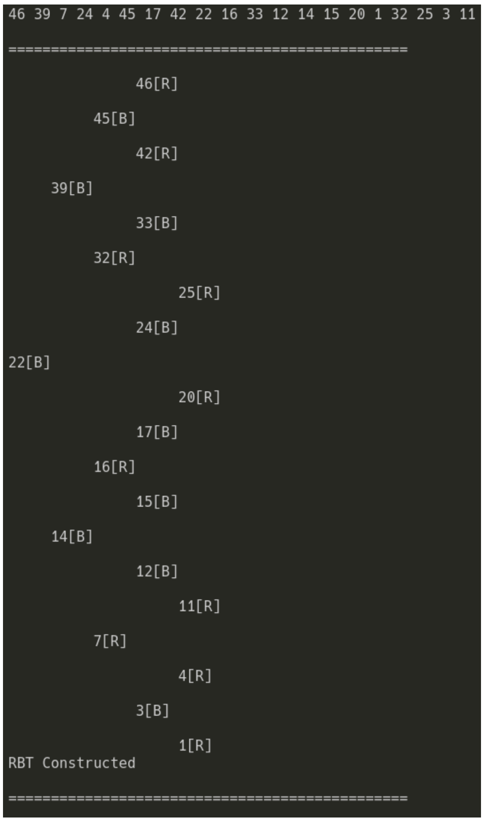
```
    y->left=z;
  else
    y->right=z;
}

//calculate the height of a binary treefree(del1);
int height(node* node) {
  int r = 0, l = 0;
  if (node->right != NULL)
    r = height(node->right);
  if (node->left != NULL)
    l =height(node->left);

  return 1 + max(r, l);
}

void ClearTree(node* root) {
  if (root != NULL) {
    ClearTree(root->left);
    ClearTree(root->right);
    free(root);
  }
}

int main(){
  int i;

  generate();

  for(i=0;i<NUM;i++){
    printf("%d ",A[i]);
  }
  printf("\n");

  printf("\n================================================\n");
  node *root = NULL;
  for(i=0;i<NUM;i++){
    root=RB_INSERT(root,A[i]);
  }
  PRINT_RBT(root,0);
  printf("RBT Constructed\n");
  printf("\n================================================\n");
  node *check=RB_SEARCH(root,6);
  if(check==NULL){
    root=RB_INSERT(root,6);
    PRINT_RBT(root,0);
    printf("Inserted node 6 in red black tree\n");
    printf("\n================================================\n");
  }
  else{
    printf("\n================================================\n");
    printf("Already exist node 6 in red black tree\n");
    printf("================================================\n");
  }
  check=RB_SEARCH(root,29);
  if(check==NULL){
    root=RB_INSERT(root,29);
    PRINT_RBT(root,0);
    printf("Inserted node 29 in red black tree\n");
    printf("\n================================================\n");
  }
  else{
    printf("\n================================================\n");
    printf("Already exist node 29 in red black tree\n");
    printf("================================================\n");
  }
  check=RB_SEARCH(root,17);
  if(check==NULL){
    root=RB_INSERT(root,17);
    PRINT_RBT(root,0);
```

```
    printf("Inserted node 17 in red black tree\n");
    printf("\n==================================================\n");
  }
  else{
    printf("\n==================================================\n");
    printf("Already exist node 17 in red black tree\n");
    printf("==================================================\n");
  }
  check=RB_SEARCH(root,21);
  if(check==NULL){
    root=RB_INSERT(root,21);
    PRINT_RBT(root,0);
    printf("Inserted node 21 in red black tree\n");
    printf("\n==================================================\n");
  }
  else{
    printf("\n==================================================\n");
    printf("Already exist node 21 in red black tree\n");
    printf("==================================================\n");
  }
  printf("\n==================================================\n");

  check = RB_SEARCH(root,6);
  if (check==NULL){
    printf("Not exist node 6 in red black tree\n");
    printf("==================================================\n");
  }
  else{
    node *r =RB_DELETE(root,6);
    PRINT_RBT(root,0);
    printf("deleted node 6 in red black tree\n");
    printf("\n==================================================\n");
  }

  check = RB_SEARCH(root,17);
  if (check==NULL){
    printf("Not exist node 17 in red black tree\n");
    printf("==================================================\n");
  }
  else{
    node *r=RB_DELETE(root,17);
    PRINT_RBT(root,0);
    printf("deleted node 17 in red black tree\n");
    printf("\n==================================================\n");
  }

  check = RB_SEARCH(root,21);
  if (check==NULL){
    printf("Not exist node 21 in red black tree\n");
    printf("==================================================\n");
  }
  else{
    node *r =RB_DELETE(root,21);
    PRINT_RBT(root,0);
    printf("deleted node 21 in red black tree\n");
    printf("\n==================================================\n");
  }

  check = RB_SEARCH(root,7);
  if (check==NULL){
    printf("Not exist node 7 in red black tree\n");
    printf("==================================================\n");
  }
  else{
    node *r=RB_DELETE(root,7);
    PRINT_RBT(root,0);
    printf("deleted node 7 in red black tree\n");
    printf("\n==================================================\n");
  }
  ClearTree(root);
```

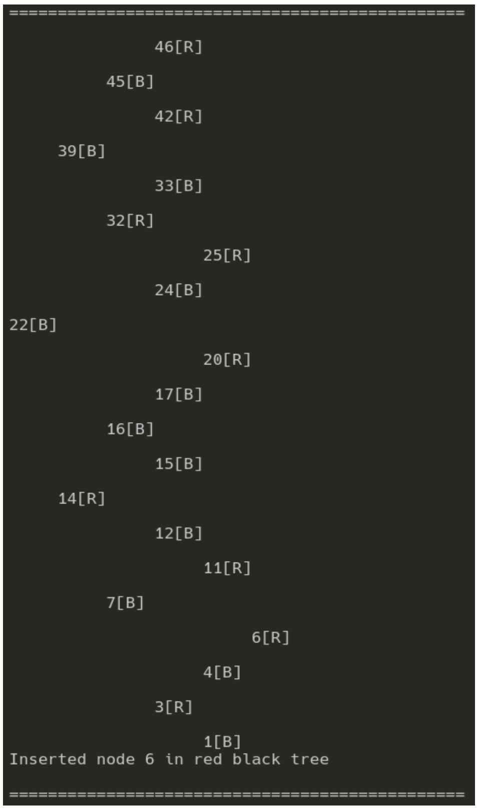
```
    node *BST = CreateNode(A[0]);
    for(i=1;i<NUM;i++){
        TREE_INSERT(BST,A[i]);
    }
    node *RBT = NULL;
    for(i=0;i<NUM;i++){
        RBT=RB_INSERT(RBT,A[i]);
    }
    printf("\n");
    printf("The height of BST is %d\n",height(BST));
    printf("The height of RBT is %d\n",height(RBT));

    ClearTree(BST);
    ClearTree(RBT);

    return 0;
}
```

Listing 3: algo3.c

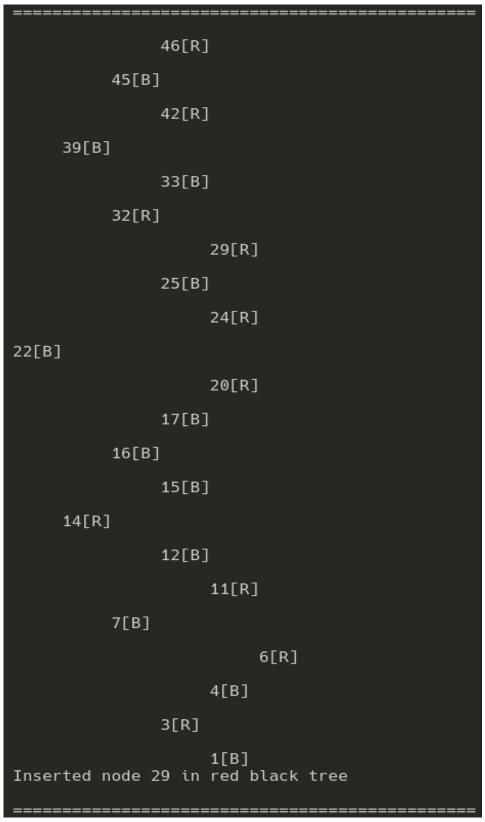
4.3 result of red black tree



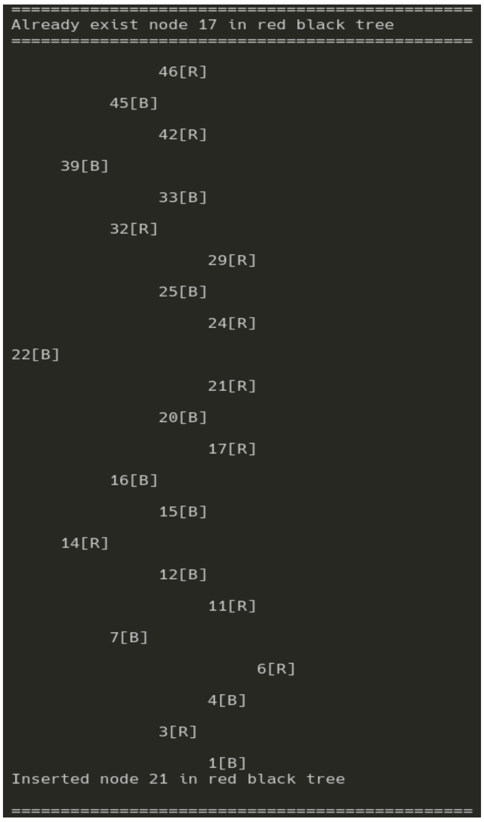
(a) RBT constructed in algo3.c



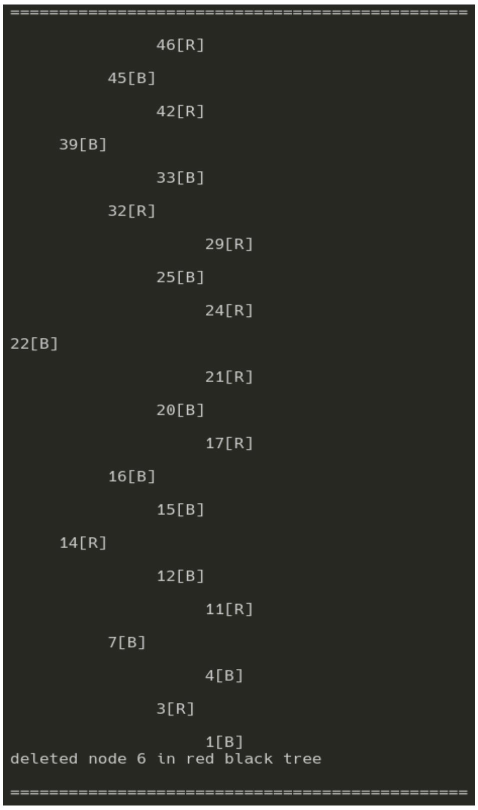
(b) 6 inserted in RBT in algo3.c



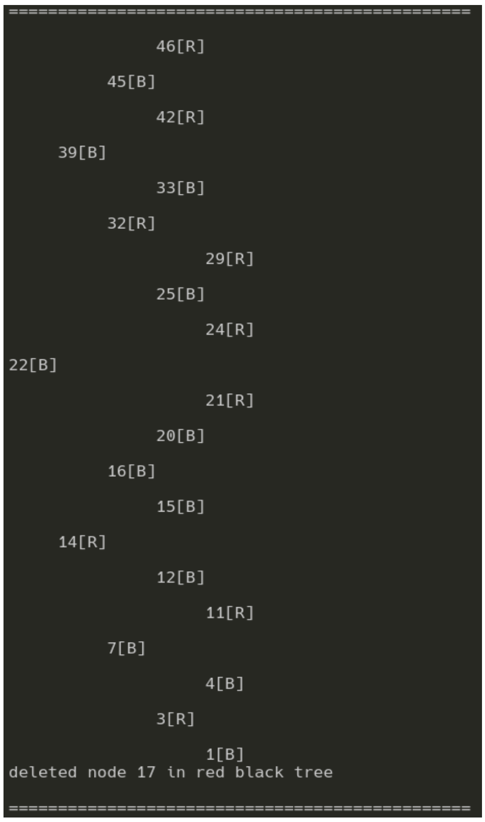
(c) 29 inserted in RBT in algo3.c



(a) 17 already exist and 21 inserted in RBT in algo3.c



(b) 6 deleted in RBT in algo3.c



(c) 17 deleted in RBT in algo3.c

```
===============================================

               46[R]

          45[B]

               42[R]

     39[B]

               33[B]

          32[R]

                    29[R]

               25[B]

                    24[R]

22[B]

               20[B]

          16[B]

               15[B]

     14[R]

               12[B]

                    11[R]

          7[B]

                    4[B]

               3[R]

                    1[B]
deleted node 21 in red black tree

===============================================
```

(a) 21 deleted in RBT in algo3.c

```
===============================================

               46[R]

          45[B]

               42[R]

     39[B]

               33[B]

          32[R]

                    29[R]

               25[B]

                    24[R]

22[B]

               20[B]

          16[B]

               15[B]

     14[R]

               12[B]

          11[B]

                    4[B]

               3[R]

                    1[B]
deleted node 7 in red black tree

===============================================

The height of BST is 9
The height of RBT is 5
```

(b) 7 deleted in RBT and height of RBT and BST in algo3.c