I. Cây nhị phân

1. Cấu trúc dữ liệu và thao tác cơ bản

Cấu trúc của 1 Node:

```
typedef struct Node* ref;

struct Node{
    int key;
    ref left;
    ref right;
}
```

Hàm tạo cây nhị phân

```
ref getNode(int k){
    ref p = new Node;
    p->key = k;
    p->left = NULL;
    p->right = NULL;
    return p;
}

ref tree(int n){
    if (n == 0)
        return NULL;
    int nl = n / 2;
    int nr = n - nl - 1;
    ref p = getNode(k);
    p->left = tree(nl);
    p->right = tree(nr);
    return p;
}
```

Hàm duyệt cây nhị phân bằng đệ qui

Duyệt tiền thứ tự (r-L-R)

```
void iterPreOrder(ref root){
    if (root != NULL){
        printf("%d\n", root->key);
        iterPreOrder(root->left);
        iterPreOrder(root->right);
    }
}
```

Duyệt hậu thứ tự (L-R-r)

```
void iterPostOrder(ref root){
    if (root != NULL){
        iterPostOrder(root->left);
        iterPostOrder(root->right);
        printf("%d\n", root->key);
    }
}
```

Duyệt trung thứ tự (L-r-R)

```
void iterInOrder(ref root){
    if (root != NULL){
        iterInOrder(root->left);
        printf("%d\n", root->key);
        iterInOrder(root->right);
    }
}
```

Hàm duyệt cây dùng stack để khử đệ qui

Duyệt tiền thứ tự

```
iterPreOrder(root) {
    p = root;
    if (p) {
        push(S, p);
        while (!isEmpty(S)) {
            p = pop(S);
            cout << p->key << " ";
            if (p->right) push(S, p->right);
            if (p->left)  push(S, p->left);
        }
    }
}
```

Duyệt hậu thứ tự

```
iterPostOrder(root) {
   p = q = root;
   while (p) {
      for ( ; p->left; p = p->left)
         push(S, p);
      while (p->right == NULL || p->right == q) {
         cout << p->key << " ";
         q = p;
         if (isEmpty(S))
            return;
         p = pop(S);
      }
      push(S, p);
      p = p->right;
   }
}
```

Duyệt trung thứ tự

```
void iterInOrder(root) {
   p = root;
   while (p) {
      while (p) {
         if (p->right)       push(S, p->right);
         push(S, p);
         p = p->left;
      }
      p = pop(S);
      while (!isEmpty(S) && p->right == NULL) {
         cout << p->key << " ";
         p = pop(S);
      }
      cout << p->key << " ";
      if (!isEmpty(S)) p = pop(S);
      else             p = NULL;
   }
}
```

Khử đệ qui bằng cách đưa về cây nhị phân suy biến

Duyệt tiền thứ tự

```
MorrisPreOrder(root) {
   p = root;
   while (p)
      if (p->left == NULL) {
         cout << p->key << " ";
         p = p->right;
      }
      else {
         tmp = p->left;
         while (tmp->right != NULL && tmp->right != p)
            tmp = tmp->right;

         if (tmp->right == NULL) {
            cout << p->key << " ";
            tmp->right = p;
            p = p->left;
         }
         else {
            tmp->right = NULL;
            p = p->right;
         }
      }
}
```

Duyệt hậu thứ tự

```
void Reverse(Ref &from, Ref &to){
     if (from == to) return;
     Ref x = from, y = from->right, z;
     while (true){
          z = y->right;
          y->right = x;
          x = y;
          y = z;
          if (x == to) break;
     }
}
```

```
MorrisPostOrder(root) {
   ref   dummy = new node;
   dummy->left = root;
   dummy->right = NULL;

   p = dummy;
   while (p)
      if (p->left == NULL)
         p = p->right;
      else {
         tmp = p->left;
         while (tmp->right != NULL && tmp->right != p)
            tmp = tmp->right;

         if (tmp->right == NULL) {
            tmp->right = p;
            p = p->left;
         }
         else {
            Reverse(p->left, tmp);
            ref   t = tmp;
            while (t != p->left) {
               cout << t->key << " ";
               t = t->right;
            }
            cout << t->key << " ";
            Reverse(tmp, p->left);
            tmp->right = NULL;
            p = p->right;
         }
      }
}
```

Duyệt trung thứ tự

```
MorrisInOrder(root) {
   p = root;
   while (p)
      if (p->left == NULL) {
         cout << p->key << " ";
         p = p->right;
      }
      else {
         tmp = p->left;
         while (tmp->right != NULL && tmp->right != p)
            tmp = tmp->right;

         if (tmp->right == NULL) {
            tmp->right = p;
            p = p->left;
         }
         else {
            cout << p->key << " ";
            tmp->right = NULL;
            p = p->right;
         }
      }
}
```

2. Tiêu chuẩn cân bằng hoàn toàn

Với mỗi một Node gốc (con) bất kì, tổng số Node bên cây con trái và tổng số Node bên cây con phải chênh lệch nhau không quá 1 đơn vị.

```
void BuildCNPCBHT(Ref& root, int*a, int l, int r)
{
    int mid = (l + r )/ 2;
    if (l < r)
    {
        int x = a[mid];
        Add(root, x);
        BuildCNPCBHT(root, a, l, mid);
        BuildCNPCBHT(root, a, mid + 1, r);
    }
}
```

II. Cây nhị phân tìm kiếm (BST)

Tìm kiếm 1 Node

```
void BST(ref &root, int k){
    ref tmp = root;
    while (tmp){
        if (tmp->key == k)
            return tmp;
        if (tmp->key < k)
            tmp = tmp->right;
```

```
        else
            tmp = tmp->left;
    }
    return NULL;
}
```

Thêm 1 Node

Dùng đệ qui

```
void searchAdd(ref &root, int k){
    if (root == NULL)
        root = getNode(k);
    else{
        if (root->key < k)
            searchAdd(root->right, k);
        else
            if (root->key > k)
                searchAdd(root->left, k);
            else
                return;
    }
}
```

Không đệ qui

```
void searchAdd(ref &root, int k){
    if (root == NULL){
        root = getNode(k);
        return;
    }
    ref tmp = root;
    ref pre = NULL;
    while (tmp){
        if (tmp->key == k)
            return;
        else
            if (tmp->key < k){
                pre = tmp;
                tmp = tmp->right;
            }
            else{
                pre = tmp;
                tmp = tmp->left;
            }
    }
    if (pre->key > k)
        pre->left = getNode(k);
    else
        pre->right = getNode(k);
}
```

Xoá 1 Node đệ qui (Node con thế mạng là Node phải nhất của cây con trái)

```
void deleteNode(ref &r, ref &q){
    if (r->right)
        deleteNode(r->right, q);
```

```
        else{
            q->key = r->key;
            q = r;
            r = r->left;
        }
    }

    void searchDelete(ref &root, int k){
        if (root == NULL)
            return;
        if (root->key < k)
            searchDelete(root->right, k);
        else
            if (root->key > k)
                searchDelete(root->left, k);
            else{
                ref q = root;
                if (q->left == NULL)
                    root = q->right;
                else
                    if (q->right == NULL)
                        root = q->left;
                    else
                        deleteNode(root->left, q);
                delete q;
            }
    }
```

III. Cây AVL

Cấu trúc của Node

```
typdef Node* ref;
struct Node{
    int key;
    int count;
    int bal;
    ref left;
    ref right;
}

ref getNode(int k) {
     ref p = new Node;
     p->key = k;
     p->count = 0;
     p->bal = 0;
     p->pLeft = p->pRight = NULL;
     return p;
}
```

Thêm 1 Node

```
void searchAdd(int x, ref &p, int &h) {
    ref p1, p2;
    if (p == NULL) {
        h = 1;
        p = new Node;
        p->key = x;
        p->count = 1;
        p->bal = 0;
        p->pLeft = p->pRight = NULL;
    }
    else
        if (x < p->key) {
            searchAdd(x, p->pLeft, h);
            if (h)
                switch (p->bal) {
                case 1:
                    p->bal = 0;
                    h = 0;
                    break;
                case 0:
                    p->bal = -1;
                    break;
                case -1:
                    p1 = p->pLeft;
                    if (p1->bal == -1) { // LL
                        p->pLeft = p1->pRight;
                        p1->pRight = p;
                        p->bal = 0;
                        p = p1;
                    }
                    else { // LR
                        p2 = p1->pRight;
                        p1->pRight = p2->pLeft;
                        p2->pLeft = p1;
                        p->pLeft = p2->pRight;
                        p2->pRight = p;
                        if (p2->bal == -1) p->bal = 1;
                        else p->bal = 0;
                        if (p2->bal == 1) p1->bal = -1;
                        else p1->bal = 0;
                        p = p2;
                    }
                    p->bal = 0;
                    h = 0;
                }
        }
        else
            if (x > p->key) {
                searchAdd(x, p->pRight, h);
                if (h)
                    switch (p->bal) {
                    case -1:
                        p->bal = 0;
                        h = 0;
                        break;
                    case 0:
                        p->bal = 1;
                        break;
                    case 1:
                        p1 = p->pRight;
                        if (p1->bal == 1) { // RR
                            p->pRight = p1->pLeft;
                            p1->pLeft = p;
                            p->bal = 0;
                            p = p1;
                        }
                        else { // RL
                            p2 = p1->pLeft;
                            p1->pLeft = p2->pRight;
                            p2->pRight = p1;
                            p->pRight = p2->pLeft;
                            p2->pLeft = p;
                            if (p2->bal == 1) p->bal = -1;
                            else p->bal = 0;
                            if (p2->bal == -1) p1->bal = 1;
                            else p1->bal = 0;
                            p = p2;
                        }
                        p->bal = 0;
                        h = 0;
                    }
            }
            else {
                p->count++;
                h = 0;
            }
}
```

Xoá 1 Node

```
void balance1(ref &p, int &h) {
    ref p1, p2;
    int b1, b2;
    switch (p->bal) {
    case -1:
        p->bal = 0;
        break;
    case 0:
        p->bal = 1;
        h = 0;
        break;
    case 1:
        p1 = p->pRight;
        b1 = p1->bal;
        if (b1 >= 0) { // RR
            p->pRight = p1->pLeft;
            p1->pLeft = p;
            if (b1 == 0) {
                p->bal = 1;
                p1->bal = -1;
                h = 0;
            }
            else {
                p->bal = 0;
                p1->bal = 0;
            }
            p = p1;
        }
        else { // RL
            p2 = p1->pLeft;
            b2 = p2->bal;
            p1->pLeft = p2->pRight;
            p2->pRight = p1;
            p->pRight = p2->pLeft;
            p2->pLeft = p;
            if (b2 == 1) p->bal = -1;
            else p->bal = 0;
            if (b2 == -1) p1->bal = 1;
            else p1->bal = 0;
            p = p2;
            p2->bal = 0;
        }
    }
}
```

```
void balance2(ref &p, int &h) {
    ref p1, p2;
    int b1, b2;
    switch (p->bal) {
    case 1:
        p->bal = 0;
        break;
    case 0:
        p->bal = -1;
        h = 0;
        break;
    case -1:
        p1 = p->pLeft;
        b1 = p1->bal;
        if (b1 <= 0) { // LL
            p->pLeft = p1->pRight;
            p1->pRight = p;
            if (b1 == 0) {
                p->bal = -1;
                p1->bal = 1;
                h = 0;
            }
            else {
                p->bal = 0;
                p1->bal = 0;
            }
            p = p1;
        }
        else { // LR
            p2 = p1->pRight;
            b2 = p2->bal;
            p1->pRight = p2->pLeft;
            p2->pLeft = p1;
            p->pLeft = p2->pRight;
            p2->pRight = p;
            if (b2 == -1) p->bal = 1;
            else p->bal = 0;
            if (b2 == 1) p1->bal = -1;
            else p1->bal = 0;
            p = p2;
            p2->bal = 0;
        }
    }
}
```

```
void del(ref &q, ref &r, int &h) {
    if (r->pRight) {
        del(q, r->pRight, h);
        if (h)
            balance2(r, h);
    }
    else {
        q->key = r->key;
        q->count = r->count;
        q = r;
        r = r->pLeft;
        h = 1;
    }
}
```

```
void searchDelete(int x, ref &p, int &h) {
    ref q;
    if (p == NULL)
        h = 0;
    else
        if (x < p->key) {
            searchDelete(x, p->pLeft, h);
            if (h)
                balance1(p, h);
        }
        else
            if (x > p->key) {
                searchDelete(x, p->pRight, h);
                if (h)
                    balance2(p, h);
            }
            else {
                q = p;
                if (q->pRight == NULL) {
                    p = q->pLeft;
                    h = 1;
                }
                else
                    if (q->pLeft == NULL) {
                        p = q->pRight;
                        h = 1;
                    }
                    else {
                        del(q, p->pLeft, h);
                        if (h)
                            balance1(p, h);
                    }
                    delete q;
            }
}
```

IV. Cây đỏ đen

```
typedef struct Node * ref;
struct Node { int key; int color; ref parent; ref left; ref right;}
ref getNode(int key, int color, ref nil) {
     p = new Node;
     p->key = key;
     p->color = color;
     p->left = p->right = p->parent = nil;
     return p;
}
```

Trạng thái ban đầu

```
ref nil, root;
…
nil = new Node; nil->color = BLACK;
nil->left = nil->right = nil->parent = nil;
…
root = nil;
```

```
void leftRotate(ref &root, ref x) {
```

```
     y = x->right;
     x->right = y->left;
     if (y->left != nil) y->left->parent = x;
     y->parent = x->parent;
     if (x->parent == nil) root = y;
     else
          if (x == x->parent->left)     x->parent->left = y;
          else           x->parent->right = y;
     y->left = x;
     x->parent = y;
}


void RBT_Insertion(ref & root, int key) {
     x = getNode(key, RED, nil);
     BST_Insert(root, x);
     Insertion_FixUp(root, x);
}


void BST_Insert(ref &root, ref x) {
     y = nil; z = root;
     while (z != nil) {
          y = z;
          if (x->key < z->key) z = z->left;
          else z = z->right;
     }
     x->parent = y;
     if (y == nil)  root = x;
     else
if (x->key < y->key) y->left = x;
          else y->right = x;
}
void Insertion_FixUp(ref &root, ref x) {
     while (x->parent->color == RED)
          if (x->parent == x->parent->parent->left)
               ins_leftAdjust(root, x);
          else
               ins_rightAdjust(root, x);
     root->color = BLACK;
}
void ins_leftAdjust(ref &root, ref &x) {
     u = x->parent->parent->right;
     if (u->color == RED) {
          x->parent->color = BLACK;
          u->color = BLACK;
          x->parent->parent->color = RED;
          x = x->parent->parent;
     }
     else {
          if (x == x->parent->right) {
               x = x->parent;
```

```
                leftRotate(root, x);
            }
            x->parent->color = BLACK;
            x->parent->parent->color = RED;
            rightRotate(root, x->parent->parent);
        }
    }


    void RBT_Deletion(ref &root, int k) {
        z = searchTree(root, k);
        if (z == nil) return;
        y = (z->left == nil) || (z->right == nil) ?
            z : TreeSuccessor(root, z);
        x = (y->left == nil) ? y->right : y->left;
        x->parent = y->parent;
        if (y->parent == nil) root = x;
        else if (y == y->parent->left)
            y->parent->left = x;
        else
            y->parent->right = x;
        if (y != z) z->key = y->key;
        if (y->color == BLACK)
            Deletion_FixUp(root, x);
        delete y;
    }


    void Deletion_FixUp(ref root, ref x) {
        while ((x->color == BLACK) && (x != root))
            if (x == x->parent->left)
                del_leftAdjust(root, x);
            else
                del_rightAdjust(root, x);
        x->color = BLACK;
    }
    void del_leftAdjust(ref & root, ref & x) {
        w = x->parent->right;
        if (w->color == RED) {
            w->color = BLACK;
            x->parent->color = RED;
            leftRotate(root, x->parent);
            w = x->parent->right;
        }
            if ((w->right->color == BLACK) &&
                (w->left->color == BLACK)) {
                w->color = RED; x = x->parent;
            }
            else {
                if (w->right->color == BLACK) {
                    w->left->color = BLACK;
                    w->color = RED;
```

```
                    rightRotate(root, w);
                    w = x->parent->right;
                }
```

V. B Tree

```
#define M 3

struct node {
     int n; /* n < M No. of keys in node will always less than order of
B tree */
     int keys[M - 1]; /*array of keys*/
     struct node *p[M]; /* (n+1 pointers will be in use) */
}*root = NULL;

enum KeyStatus { Duplicate, SearchFailure, Success, InsertIt, LessKeys
};

void insert(int key);
void display(struct node *root, int);
void DelNode(int x);
void search(int x);
enum KeyStatus ins(struct node *r, int x, int* y, struct node** u);
int searchPos(int x, int *key_arr, int n);
enum KeyStatus del(struct node *r, int x);
void eatline(void);
void inorder(struct node *ptr);
int totalKeys(struct node *ptr);
void printTotal(struct node *ptr);
int getMin(struct node *ptr);
int getMax(struct node *ptr);
void getMinMax(struct node *ptr);
int max(int first, int second, int third);
int maxLevel(struct node *ptr);
void printMaxLevel(struct node *ptr);


void insert(int key)
{
     struct node *newnode;
     int upKey;
     enum KeyStatus value;
     value = ins(root, key, &upKey, &newnode);
     if (value == Duplicate)
          printf("Key already available\n");
     if (value == InsertIt)
     {
          struct node *uproot = root;
          root = malloc(sizeof(struct node));
          root->n = 1;
          root->keys[0] = upKey;
          root->p[0] = uproot;
          root->p[1] = newnode;
     }/*End of if */
```

```
}/*End of insert()*/

enum KeyStatus ins(struct node *ptr, int key, int *upKey, struct node
**newnode)
{
     struct node *newPtr, *lastPtr;
     int pos, i, n, splitPos;
     int newKey, lastKey;
     enum KeyStatus value;
     if (ptr == NULL)
     {
          *newnode = NULL;
          *upKey = key;
          return InsertIt;
     }
     n = ptr->n;
     pos = searchPos(key, ptr->keys, n);
     if (pos < n && key == ptr->keys[pos])
          return Duplicate;
     value = ins(ptr->p[pos], key, &newKey, &newPtr);
     if (value != InsertIt)
          return value;
     /*If keys in node is less than M-1 where M is order of B tree*/
     if (n < M - 1)
     {
          pos = searchPos(newKey, ptr->keys, n);
          /*Shifting the key and pointer right for inserting the new
key*/
          for (i = n; i>pos; i--)
          {
               ptr->keys[i] = ptr->keys[i - 1];
               ptr->p[i + 1] = ptr->p[i];
          }
          /*Key is inserted at exact location*/
          ptr->keys[pos] = newKey;
          ptr->p[pos + 1] = newPtr;
          ++ptr->n; /*incrementing the number of keys in node*/
          return Success;
     }/*End of if */
     /*If keys in nodes are maximum and position of node to be inserted
is last*/
     if (pos == M - 1)
     {
          lastKey = newKey;
          lastPtr = newPtr;
     }
     else /*If keys in node are maximum and position of node to be
inserted is not last*/
     {
          lastKey = ptr->keys[M - 2];
          lastPtr = ptr->p[M - 1];
          for (i = M - 2; i>pos; i--)
```

```
            {
                ptr->keys[i] = ptr->keys[i - 1];
                ptr->p[i + 1] = ptr->p[i];
            }
            ptr->keys[pos] = newKey;
            ptr->p[pos + 1] = newPtr;
        }
        splitPos = (M - 1) / 2;
        (*upKey) = ptr->keys[splitPos];

        (*newnode) = malloc(sizeof(struct node));/*Right node after
split*/
        ptr->n = splitPos; /*No. of keys for left splitted node*/
        (*newnode)->n = M - 1 - splitPos;/*No. of keys for right splitted
node*/
        for (i = 0; i < (*newnode)->n; i++)
        {
            (*newnode)->p[i] = ptr->p[i + splitPos + 1];
            if (i < (*newnode)->n - 1)
                (*newnode)->keys[i] = ptr->keys[i + splitPos + 1];
            else
                (*newnode)->keys[i] = lastKey;
        }
        (*newnode)->p[(*newnode)->n] = lastPtr;
        return InsertIt;
}/*End of ins()*/

void display(struct node *ptr, int blanks)
{
        if (ptr)
        {
            int i;
            for (i = 1; i <= blanks; i++)
                printf(" ");
            for (i = 0; i < ptr->n; i++)
                printf("%d ", ptr->keys[i]);
            printf("\n");
            for (i = 0; i <= ptr->n; i++)
                display(ptr->p[i], blanks + 10);
        }/*End of if*/
}/*End of display()*/

void search(int key)
{
        int pos, i, n;
        struct node *ptr = root;
        printf("Search path:\n");
        while (ptr)
        {
            n = ptr->n;
            for (i = 0; i < ptr->n; i++)
                printf(" %d", ptr->keys[i]);
```

```
            printf("\n");
            pos = searchPos(key, ptr->keys, n);
            if (pos < n && key == ptr->keys[pos])
            {
                 printf("Key %d found in position %d of last dispalyed
node\n", key, i);
                 return;
            }
            ptr = ptr->p[pos];
      }
      printf("Key %d is not available\n", key);
}/*End of search()*/

int searchPos(int key, int *key_arr, int n)
{
      int pos = 0;
      while (pos < n && key > key_arr[pos])
            pos++;
      return pos;
}/*End of searchPos()*/

void DelNode(int key)
{
      struct node *uproot;
      enum KeyStatus value;
      value = del(root, key);
      switch (value)
      {
      case SearchFailure:
            printf("Key %d is not available\n", key);
            break;
      case LessKeys:
            uproot = root;
            root = root->p[0];
            free(uproot);
            break;
      }/*End of switch*/
}/*End of delnode()*/

enum KeyStatus del(struct node *ptr, int key)
{
      int pos, i, pivot, n, min;
      int *key_arr;
      enum KeyStatus value;
      struct node **p, *lptr, *rptr;

      if (ptr == NULL)
            return SearchFailure;
      /*Assigns values of node*/
      n = ptr->n;
      key_arr = ptr->keys;
      p = ptr->p;
```

```
     min = (M - 1) / 2;/*Minimum number of keys*/

     //Search for key to delete
     pos = searchPos(key, key_arr, n);
     // p is a leaf
     if (p[0] == NULL)
     {
          if (pos == n || key < key_arr[pos])
               return SearchFailure;
          /*Shift keys and pointers left*/
          for (i = pos + 1; i < n; i++)
          {
               key_arr[i - 1] = key_arr[i];
               p[i] = p[i + 1];
          }
          return --ptr->n >= (ptr == root ? 1 : min) ? Success :
LessKeys;
     }/*End of if */

     //if found key but p is not a leaf
     if (pos < n && key == key_arr[pos])
     {
          struct node *qp = p[pos], *qp1;
          int nkey;
          while (1)
          {
               nkey = qp->n;
               qp1 = qp->p[nkey];
               if (qp1 == NULL)
                    break;
               qp = qp1;
          }/*End of while*/
          key_arr[pos] = qp->keys[nkey - 1];
          qp->keys[nkey - 1] = key;
     }/*End of if */
     value = del(p[pos], key);
     if (value != LessKeys)
          return value;

     if (pos > 0 && p[pos - 1]->n > min)
     {
          pivot = pos - 1; /*pivot for left and right node*/
          lptr = p[pivot];
          rptr = p[pos];
          /*Assigns values for right node*/
          rptr->p[rptr->n + 1] = rptr->p[rptr->n];
          for (i = rptr->n; i>0; i--)
          {
               rptr->keys[i] = rptr->keys[i - 1];
               rptr->p[i] = rptr->p[i - 1];
          }
          rptr->n++;
```

```
            rptr->keys[0] = key_arr[pivot];
            rptr->p[0] = lptr->p[lptr->n];
            key_arr[pivot] = lptr->keys[--lptr->n];
            return Success;
        }/*End of if */
        //if (posn > min)
        if (pos < n && p[pos + 1]->n > min)
        {
            pivot = pos; /*pivot for left and right node*/
            lptr = p[pivot];
            rptr = p[pivot + 1];
            /*Assigns values for left node*/
            lptr->keys[lptr->n] = key_arr[pivot];
            lptr->p[lptr->n + 1] = rptr->p[0];
            key_arr[pivot] = rptr->keys[0];
            lptr->n++;
            rptr->n--;
            for (i = 0; i < rptr->n; i++)
            {
                rptr->keys[i] = rptr->keys[i + 1];
                rptr->p[i] = rptr->p[i + 1];
            }/*End of for*/
            rptr->p[rptr->n] = rptr->p[rptr->n + 1];
            return Success;
        }/*End of if */

        if (pos == n)
            pivot = pos - 1;
        else
            pivot = pos;

        lptr = p[pivot];
        rptr = p[pivot + 1];
        /*merge right node with left node*/
        lptr->keys[lptr->n] = key_arr[pivot];
        lptr->p[lptr->n + 1] = rptr->p[0];
        for (i = 0; i < rptr->n; i++)
        {
            lptr->keys[lptr->n + 1 + i] = rptr->keys[i];
            lptr->p[lptr->n + 2 + i] = rptr->p[i + 1];
        }
        lptr->n = lptr->n + rptr->n + 1;
        free(rptr); /*Remove right node*/
        for (i = pos + 1; i < n; i++)
        {
            key_arr[i - 1] = key_arr[i];
            p[i] = p[i + 1];
        }
        return --ptr->n >= (ptr == root ? 1 : min) ? Success : LessKeys;
    }/*End of del()*/

    void eatline(void) {
```

```
     char c;
     printf("");
     while (c = getchar() != '\n');
}

/* Function to display each key in the tree in sorted order (in-order
traversal)
@param struct node *ptr, the pointer to the node you are currently
working with
*/
void inorder(struct node *ptr) {
     if (ptr) {
          if (ptr->n >= 1) {
               inorder(ptr->p[0]);
               printf("%d ", ptr->keys[0]);
               inorder(ptr->p[1]);
               if (ptr->n == 2) {
                    printf("%d ", ptr->keys[1]);
                    inorder(ptr->p[2]);
               }
          }
     }
}

/* Function that returns the total number of keys in the tree.
@param struct node *ptr, the pointer to the node you are currently
working with
*/
int totalKeys(struct node *ptr) {
     if (ptr) {
          int count = 1;
          if (ptr->n >= 1) {
               count += totalKeys(ptr->p[0]);
               count += totalKeys(ptr->p[1]);
               if (ptr->n == 2) count += totalKeys(ptr->p[2]) + 1;
          }
          return count;
     }
     return 0;
}

/* Function that prints the total number of keys in the tree.
@param struct node *ptr, the pointer to the node you are currently
working with
*/
void printTotal(struct node *ptr) {
     printf("%d\n", totalKeys(ptr));
}

/* Function that returns the smallest key found in the tree.
@param struct node *ptr, the pointer to the node you are currently
working with
```

```
*/
int getMin(struct node *ptr) {
     if (ptr) {
          int min;
          if (ptr->p[0] != NULL) min = getMin(ptr->p[0]);
          else min = ptr->keys[0];
          return min;
     }
     return 0;
}

/* Function that returns the largest key found in the tree.
@param struct node *ptr, the pointer to the node you are currently
working with
*/
int getMax(struct node *ptr) {
     if (ptr) {
          int max;
          if (ptr->n == 1) {
               if (ptr->p[1] != NULL) max = getMax(ptr->p[1]);
               else max = ptr->keys[0];
          }
          if (ptr->n == 2) {
               if (ptr->p[2] != NULL) max = getMax(ptr->p[2]);
               else max = ptr->keys[1];
          }
          return max;
     }
     return 0;
}

/* Function that prints the smallest and largest keys found in the tree.
@param struct node *ptr, the pointer to the node you are currently
working with
*/
void getMinMax(struct node *ptr) {
     printf("%d %d\n", getMin(ptr), getMax(ptr));
}

/* Function that determines the largest number.
@param int, integer to compare.
@param int, integer to compare.
@param int, integer to compare.
*/
int max(int first, int second, int third) {
     int max = first;
     if (second > max) max = second;
     if (third > max) max = third;
     return max;
}
```

```
/*Function that finds the maximum level in the node and returns it as
an integer.
@param struct node *ptr, the node to find the maximum level for.
*/
int maxLevel(struct node *ptr) {
     if (ptr) {
          int l = 0, mr = 0, r = 0, max_depth;
          if (ptr->p[0] != NULL) l = maxLevel(ptr->p[0]);
          if (ptr->p[1] != NULL) mr = maxLevel(ptr->p[1]);
          if (ptr->n == 2) {
               if (ptr->p[2] != NULL) r = maxLevel(ptr->p[2]);
          }
          max_depth = max(l, mr, r) + 1;
          return max_depth;
     }
     return 0;
}

/*Function that prints the maximum level in the tree.
@param struct node *ptr, the tree to find the maximum level for.
*/
void printMaxLevel(struct node *ptr) {
     int max = maxLevel(ptr) - 1;
     if (max == -1) printf("tree is empty\n");
     else printf("%d\n", max);
}
```

VI. Tìm kiếm chuỗi

Brute force

```
void BruteForce(string T, string P) {
     int count = 0;  int n = T.length();
     int m = P.length();  int j = 0;
     for (int i = 0;i < n;i++)
          if (T[i] == P[0])  {
               j = 0;
               while (j < m && (i + j) < n && T[i + j] == P[j] )
                    j++;
               if (j == m) { //ghi nhận;   count++; }
          }
}
```

DFA

```
void BuildDFA(string P, string T, map<char,int>* dfa)
{
     //giả sử bảng chữ cái là ABCD
     int m = P.length();
     for (int q=0; q<=m; q++)
          for (char a = 'A'; a <= 'D'; a++)
          {
               int k = (m + 1 < q + 2) ? m + 1 : q + 2;
               string Pk;
               string Pqa;
```

```
                string hautoPqa;
                do
                {
                     k--;
                     Pk = P.substr(0, k);
                     Pqa = P.substr(0, q) + a;
                     hautoPqa = Pqa.substr(Pqa.length() - k, k);
                } while (Pk != hautoPqa);
                dfa[q][a] = k;
           }
}

void DFASearch(string P, string T)
{
     int m = P.length();
     int n = T.length();
     int count = 0;
     map<char, int>* dfa= new map<char, int>[m + 1];
     BuildDFA(P, T, dfa);
     int q = 0;
     for (int i = 0;i < n;i++)
     {
          q = dfa[q][T[i]];
          if (q == m)
          {
               //ghi nhận kết quả
               //-> vị trí i-m+1
               count++;
          }
     }
     delete[] dfa;
}
KMP
void ComputePi(string P, int *pi) {
     int m = P.length();   int k;  pi[1] = k = 0;
     for (int q = 2; q <= m; q++) {
          while ((k > 0) && (P[k] != P[q - 1]))   k = pi[k];
          if (P[k] == P[q - 1])    k++;
          pi[q] = k;
     }
}

void KMP(string T, string P) {
     int m = P.length();
     int *pi = new int[m + 1];
     int count = 0;
     ComputePi(P, pi);
     int q = 0;
     for (int i = 0;i < m;i++) {
          while ((q > 0) && (P[q] != T[i]))
               q = pi[q];
          if (P[q] == T[i])
```

```
                q++;
            if (q == m) {
                //ghi nhận kết quả -> i-m+1;
                count++;
                q = pi[q];
            }
        }
    }

    Horspool
    void ComputeArray(map<char,int> D, string P, string Z) //Z là bảng chữ
    cái, giả sử là "ABCD"
    {
        int m = P.length();
        for (int i = 0;i < Z.length(); i++)     D[Z[i]] = m;
        for (int i = 0; i < m - 1; i++)         D[P[i]] = m - 1 - i;
    }

    void Horspool(string P, string T) {
        int m = P.length();
        int n = T.length();
        map<char, int> D;
        ComputeArray(D, P, "ABCD");
        int i = m - 1;
        while (i < n)
        {
            int k = 0;
            while ((k < m) && (P[m - 1 - k] == T[i - k]))
                k++;
            if (k == m)
            {
                //ghi nhận kết quả
                //-> i-m+1
            }
            i += D[T[i]];
        }
    }
```

VII. Topological Sort

```
    struct leader;
    struct trailer;
    struct leader
    {
        int key;
        int count;
        leader* next;
        trailer* trail;
    };

    struct trailer
    {
        leader* id;
        trailer* next;
```

```
};



leader* addList(leader* &head, leader* &tail, int k)
{
     leader* h = head;
     tail->key = k;
     while (h->key != k)
          h = h->next;
     if (h == tail)
     {
          tail = new leader;
          h->count = 0;
          h->next = tail;
          h->trail = NULL;
     }
     return h;
}

void main()
{
     leader* head = new leader;
     leader* tail = head;
     int x;
     int y;
     ifstream fin("test.txt");
     fin >> x;
     while (x)
     {
  fin >> y;
  //leader key x
  leader* p = addList(head, tail, x);
     leader* q = addList(head, tail, y);//leader k y
          //tạo trailer mới
          trailer* t = new trailer;
          t->id = q;
          t->next = p->trail;
          p->trail = t;
          //tăng count
          q->count++;
          fin >> x;
     }
     fin.close();
     //bỏ các phần tử có count=0 vào head;
     leader* p = head;
     head = NULL;
     while (p != tail)
     {
```

```
            leader* temp = p;
            p = p->next;
            if (temp->count == 0)
            {
                  temp->next = head;
                  head = temp;
            }
      }
      //duyệt
      leader* q = head;
      while (q)
      {
            cout << q->key << " ";
            trailer* t = q->trail;
            q = q->next;
            while (t)
            {
                  leader* tid = t->id;
                  (tid->count)--;
                  if (tid->count == 0)
                  {
                        tid->next = q;
                        q = tid;
                  }
                  t = t->next;
            }
      }
}
```

VIII. Một số bài toán QHĐ kinh điển

Dãy con tăng dài nhất

```
      vector<int> longestIncreasingSubsequence(int *a, int n){
      int *f = new int[n];
      int t = 0;
      memset(f, 0, sizeof(f));
      for(int i = 1; i < n; ++i){
            f[i] = 1;
            for(int j = 0; j < n; ++j)
             if(a[i] > a[j])
                 f[i] = max(f[i], f[j] + 1);
            if(f[t] < f[i]) t = i;
      }
      vector<int> ret;
      ret.push_back(a[t]);
      for(int i = t - 1; ~i; --i)
        if(a[i] < a[t] && f[i] + 1 == f[t]) {
             t = i;
             ret.push_back(a[t]);
      }
      reverse(ret.begin(), ret.end());
      return ret;
}
```

```
int main(){
     int a[] = {1, 7, 2, 4, 3, 3, 4, 5, 10, 6, 7, 6, 8};
     vector<int> v  = longestIncreasingSubsequence(a, 13);
     for(int i = 0; i < v.size(); ++i)
       cout << v[i] << ' ';
    cout << endl;
}
```

Chuỗi con chung dài nhất

```
const int N=1003;

int f[N][N];

string longestCommonSubsequence(string a, string b){
     int m = a.length();
     int n = b.length();
     int sz = 0;
     string res = "";
     f[0][0] = 0;
     for(int i = 1; i <= m; ++i){
          for(int j = 1; j <= n; ++j){
               f[i][j] = max(f[i][j - 1], f[i - 1][j]);
               if(a[i - 1] == b[j - 1])
                f[i][j] = max(f[i][j], f[i - 1][j - 1] + 1);
               sz = max(sz, f[i][j]);
          }
     }
     for(int i = m; i; --i){
          for(int j = n; j; --j){
               if(a[i - 1] == b[j - 1] && f[i][j] == sz){
                    --sz;
                    res += a[i - 1];
                    n = j - 1;
                    break;
               }
          }
     }
     reverse(res.begin(), res.end());
     return res;
}

int main(){
     string a = "abcefghk", b = "bxfgkl";
     cout << longestCommonSubsequence(a,b)<<endl;
}
```

```
Dãy con có Tổng bằng k cho trước
struct Node{
    int key;
    Node* next;
    bool check;
};

Node* getNode(int k){
    Node* p = new Node;
    p->key = k;
    p->next = NULL;
    p->check = false;
    return p;
}

void AddNode(Node *&head, int k){
    if (head == NULL){
        head = getNode(k);
        return;
    }
    Node* tmp = head;
    while (tmp->next)
        tmp = tmp->next;
    tmp->next = getNode(k);
}

int getSum(Node* head){
    int res = 0;
    for (Node *tmp = head; tmp; tmp = tmp->next)
        if (tmp->check)
            res += tmp->key;
    return res;
}

void printList(Node* head){
    for (Node *tmp = head; tmp; tmp = tmp->next)
        if (tmp->check)
            cout << tmp->key << "  ";
    cout << endl;
}

void cal(Node* head, Node* p, int k){
    if (p == NULL){
        if (getSum(head) == k)
            printList(head);
        return;
    }
```

```
    p->check = true;
    cal(head, p->next, k);
    p->check = false;
    cal(head, p->next, k);
}

int main(int argc, const char * argv[]) {
    int a[] = {3, 5, 1, 7, 10, 9, 4, 2, 8, 6};
    Node* head = NULL;
    for (int i = 0; i < 10; ++i)
        AddNode(head, a[i]);
    cal(head, head, 10);

    return 0;
}
```