ỦY BAN NHÂN DÂN
THÀNH PHỐ HỒ CHÍ MINH
TRƯỜNG ĐẠI HỌC SÀI GÒN

BÁO CÁO MÔN HỌC

CẤU TRÚC DỮ LIỆU VÀ GIẢI THUẬT

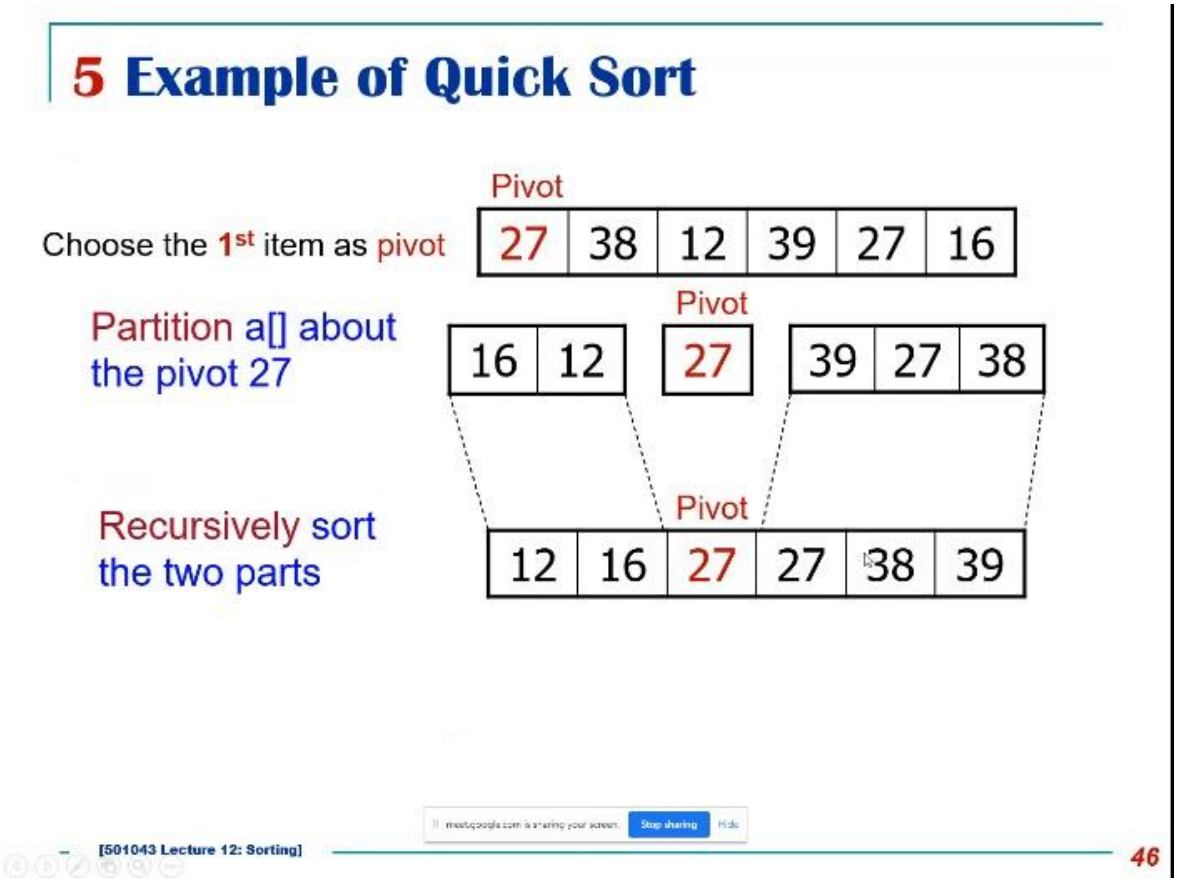
Tên sinh viên: **Trần Tường Minh.**
Mã số sinh viên: **3120411098.**
Lớp: **DCT120C2**

Thành phố Hồ Chí Minh, tháng 12 năm 2021

Câu I: Sắp xếp một mảng gồm n phần tử.

a. Trình bày ý tưởng của thuật toán sắp xếp nhanh (quick sort) sau đó mô tả các ý tưởng thuật toán bằng ví dụ minh họa.

Ý tưởng: là phương pháp sắp xếp nhanh. Trước tiên chúng ta chọn 1 phần tử pivot hay còn gọi là phần tử chính, có thể chọn phần tử pivot ở bất kì phần tử nào trong mảng, thông thường chọn đầu hoặc cuối. Giả sử ta chọn phần tử pivot là phần tử đầu, sau đó ta sẽ đem so sánh những phần tử còn lại nếu phần tử nào nhỏ hơn pivot nó sẽ di chuyển sang bên trái, nếu phần tử nào lớn hơn pivot nó sẽ di chuyển sang bên phải của pivot. Sau khi thu được 3 phần là: phần tử bên trái – nhỏ hơn pivot, pivot, phần tử lớn hơn pivot. Phần tử nhỏ hơn pivot sẽ được tiếp tục chọn 1 pivot mới để chia ra phần tử nhỏ hơn pivot mới và phần tử lớn hơn pivot mới. Tương tự như vậy cho phần tử bên phải, ta sẽ chia cho đến khi còn 1 phần tử. Như thế ta có được 1 mảng đã được sắp xếp



Ví dụ: cho 1 mảng gồm 6 phần tử: 3 5 8 2 1 9

Mô tả:

B1: chọn pivot = 3

B2: so sánh các phần tử còn lại với pivot. Nếu giá trị nào < pivot sẽ di chuyển nó sang bên trái. Giá trị nào > pivot sẽ di chuyển nó sang bên phải ➔ 2 1 3 5 8 9 là mảng thu được sau bước 2

B3: Tiếp tục chọn 1 pivot mới cho phần bên trái của pivot để chia để chia ra phần tử nhỏ hơn pivot mới và phần tử lớn hơn pivot mới. ➔ 1 2 3 5 8 9. Tương tự như vậy cho phần tử

bên phải, ta sẽ chia cho đến khi còn 1 phần tử ➔ 1 2 3 5 8 9. Như thế ta có được 1 mảng đã được sắp xếp ➔ 1 2 3 5 8 9

b. Trình bày thuật toán bằng mã giả (pseudocode) hay liệt kê các bước thực hiện.

Thuật toán

Hàm phân đoạn:

i. Truyền vào 3 tham số: mảng a, i, j với i và j lần lượt là 2 phần tử đầu và phần tử cuối của mảng
ii. Lấy pivot = a[i] – phần tử đầu của mảng
iii. Lấy biến m = phần tử đầu của mảng
iv. Tiếp đến tạo vòng lặp for duyệt từ k = i + 1 đến j - cuối mảng
v. Khi thỏa điều kiện a[k] < pivot : tăng m lên 1 đơn vị và đổi vị trí a[k] với a[m]
vi. Nếu không thỏa điều kiện thì tiếp tục tăng k lên 1 đơn vị để xét tiếp phần tử tiếp theo của mảng
vii. Lặp lại công việc từ bước iv đến vi đến cuối mảng
viii. Kết khúc vòng lặp ta đổi vị trí i và m để chuyển pivot về giữa mảng

Hàm quicksort

I. Xét điều kiện i < j để thỏa mãn mảng có tối thiểu 2 phần tử
II. gọi đề quy để sắp xếp mảng bên trái
III. gọi đề quy để sắp xếp mảng bên phải

c. Dùng ngôn ngữ lập trình C++ để thể hiện thuật toán trên. (sinh viên được khuyến khích cho việc giải thích ý nghĩa các biến cũng như các cấu trúc rẽ nhánh, vòng lập của chương trình).

```
#include <iostream>
using namespace std;

//Tạo hàm xuất mảng để xuất mảng sau khi sắp xếp
void xuatmang(int a[], int n)
{
   for (int i = 0;i <n;i++)
  {
      cout << a[i] <<" ";
   }
}
```

```
//Tạo hàm đổi vị trí 2 phần tử
void swap(int &x, int &y)
{
   int temp = x;
   x = y;
   y = temp;
}
int partition(int a[], int i, int j) //tạo 1 hàm có tên là partition trả về 1 biến m kiểu int vàtruyền vào 1 mảng a[] và 1 biến i lưu phần tử đầu của mảng và 1 biến j lưu phần tử cuối của mảng
{
   int pivot = a[i]; // chọn pivot = giá trị phần tử đầu của mảng dùng để làm phần tử chốt so sánh với giá trị các phần tử khác của mảng
   int m = i;    // gán biến m = vị trí phần tử đầu của mảng, mục đích của việc tạo biến m này dùng để chia mảng làm 2 nửa, biên trái là các phần tử có giá trị < giá trị pivot, bên phải các phần tử có giá trị lớn hơn
   for (int k = i + 1; k <= j; k++) // tạo 1 vòng lặp chạy từ vị trí i + 1 đến cuối mảng
   {
      if (a[k] < pivot) // tạo câu lệnh if với điều kiện giá trị phần tử k < pivot, mục đích để kiếm các phần tử < pivot chuyển về phía bên trái của mảng
      {
         m++; // tăng m lên 1 đơn vị
         swap(a[k], a[m]); // đổi giá trị 2 biến k và m
      }
   }
   swap(a[i], a[m]); // đổi giá trị 2 biến i và m
   return m; // trả về vị trí cuối cùng m chuyển pivot về vị trí giữa mảng
}
void quickSort(int a[], int i, int j) //tạo hàm có tên quickSort truyền vào 1 mảng a và biến i lưu phần tử đầu, j lưu phần tử cuối của mảng
{
   if (i < j) // điều kiện để mảng có ít nhất 2 phần tử
   {
      int pivot = partition(a, i, j); // gọi hàm partition và gán cho biến pivot để chia mảng
      quickSort(a, i, pivot - 1); // gọi đề quy để sắp xếp mảng bên trái
      quickSort(a, pivot + 1, j); // gọi đề quy để sắp xếp mảng bên phải
   }
}
```

```
int main ()
{
   int a[100];
   int n;
   cout << "Nhap n: ";
   cin >> n;
   for (int i = 0;i < n;i++)
   {
      cout << "Phan tu thu a["<<i<<"]: ";
      cin >> a[i];
   }
   quickSort(a,0,n-1);
   xuatmang(a,n);
   return 0;
}
```

Câu II.

a. Trình bày danh sách liên kết đơn là gì? Trình bày và giải thích các thao tác cơ bản trên danh sách liên kết đơn.

-Định nghĩa: Danh sách liên kết đơn bao gồm các node liên kết với nhau mỗi node gồm 2 thành phần

- Data (key): là dữ liệu của node
- Next: là con trỏ chỉ địa chỉ của node tiếp theo

Để quản lí danh sách liên kết, ta sử dụng con trỏ head trỏ tới node đầu tiên của danh sách liên kết.

Head là node đầu tiên của danh sách liên kết

Node cuối cùng là node trỏ đến NULL

-Các thao tác cơ bản:

1. Khởi tạo node
2. Thêm 1 node vào cuối danh sách liên kết: đầu tiên cần tạo 1 node mới để thêm vào cuối dslk, sau đó cho node cuối của dslk liên kết với node vừa tạo, rồi cập nhật lại dslk
3. Thêm 1 node vào đầu danh sách liên kết: đầu tiên cần tạo 1 node mới để thêm vào đầu dslk, sau đó cho node vừa tạo liên kết với node đầu của dslk, rồi cập nhật lại dslk
4. Tìm 1 node có khóa x trong danh sách liên kết: duyệt từ đầu đến cuối dslk node nào có giá trị = giá trị node cần tìm thì tức là có node cần tìm trong dslk
5. Xóa 1 node trong danh sách liên kết: có 3 trường hợp là xóa node đầu, giữa, cuối

6. Duyệt danh sách liên kết

b. Dùng ngôn ngữ lập trình C++ viết hàm thực hiện các thao tác trên. (sinh viên được khuyến khích cho việc giải thích ý nghĩa các biến cũng như các cấu trúc rẽ nhánh, vòng lặp của chương trình)

```
#include <iostream>
using namespace std;

struct Node {
   int data;
   Node *next;
};
//khoi tao
struct Node *initNode (int n) //tạo 1 hàm có tên initNode truyền vào 1 biến n, hàm trả về 1 node
{
   struct Node *newNode = new Node; //khai báo và tạo 1 node mới có tên newNode
   newNode->data = n; //giá trị của newNode = giá trị biến n truyền vào hàm
   newNode->next = NULL; //vừa khởi tạo nên node->next = NULL
   return newNode; //trả về 1 Node
}
//them node vao cuoi
void addNode (Node *head,int n) //khai báo 1 hàm có tên addNode để thêm 1 node vào cuối dslk, truyền vào hàm 1 Node head và 1 biến n là giá trị của node ta sẽ thêm vào dslk
{
   Node *newNode = new Node; //khai báo và tạo 1 Node mới có tên newNode
   newNode->data = n; //Node vừa tạo có giá trị = n ta truyền vào hàm
   newNode->next = NULL; //node vừa tạo nên ->next = NULL

   Node *cur = head; //khai báo 1 Node có tên cur trỏ đến con trỏ head dùng để duyệt dslk
   while(cur) //điều kiện cur != NULL
   {
      if (cur->next == NULL) //lúc này đang ở node cuối của dslk
      {
         cur->next = newNode; //tạo mối liên kết giữa node cuối và node mới vừa tạo
```

```
            return; //thoát vòng lặp
        }
        cur = cur->next; //nếu không thỏa mãn điều kiện if thì thực duyệt tiếp để kiểm tra Node khác
    }
}
//them vao dau
void InsertFront (Node **head, int n) //tạo 1 hàm có tên InsertFront truyền vào 1 Node có tên head do là khi thêm vào con trỏ head sẽ thay đổi nên ta cần truyền tham chiều và 1 biến n là giá của node ta sẽ thêm vào
{
    Node *newNode = new Node; //khai báo và tạo 1 node mới có tên newNode
    newNode->data = n; //giá trị của node mới = giá trị của biến ta truyền vào
    newNode->next = *head; //tạo mối liên kết giữa Node vừa tạo với node đầu tiên của dslk vì cần thêm vào đầu dslk
    *head = newNode; //cập nhật lại con trỏ head, vì lúc này node vừa tạo đã là node đầu tiên của dslk
}
// tim node co khoa
struct Node *searchNode (struct Node *head, int n) //tạo hàm có tên searchNode trả về 1 node, truyền vào hàm 1 Node tên head và 1 biến n là giá trị của node cần tìm
{
    Node *cur = head; //khai báo 1 Node có tên cur trỏ đến con trỏ head dùng để duyệt dslk
    while (cur) //điều kiện để thực hiện vòng lặp
    {
        if (cur->data == n) //đã tìm ra node cần tìm
        {
            return cur; //thoát vòng lặp
        }
        cur = cur->next; //nếu chưa tìm thấy duyệt sang node kế tiếp đến hết dslk
    }
    cout << " Khong tim thay.\n ";  //nếu duyệt xong dslk mà không tìm thấy node xuất ra dòng Khong tim thay.
}
// xoa node
```

```
void deleteNode (Node **head, int n) //tạo hàm có tên deleteNode truyền vào 1 Node có tên head do con trỏ head có thể thay đổi khi xóa nên ta cần truyền vào tham chiều và 1 biến n là giá trị của node ta cần xóa
{
   Node *cur = *head; //khai báo 1 Node có tên cur trỏ đến con trỏ head dùng để duyệt dslk
   Node *pre = NULL; //khai báo 1 Node tên pre cho trỏ đến NULL mục đích dùng đề xóa node khác node đầu dslk
   if (cur->data == n) //trường hợp tìm thấy giá trị node cần xóa ngay node đầu dslk
   {
      *head = cur->next; //di chuyển con trỏ head sang node kế tiếp
      delete cur; //xóa node cần xóa
      return; //thoát vòng lặp
   }
   else //trường hợp node cần xóa khác node đầu tiên
      while (cur != NULL && cur->data != n) //điều kiện duyệt vòng lặp đề tìm node cần xóa
      {
         pre = cur; //cho con trỏ pre trỏ đến cur
         cur = cur->next; //cho con trỏ cur di chuyển đến node kế tiếp
      }
   if (cur == NULL) //trường hợp duyệt hết dslk tức là node cần xóa ko có
      return; //thoát điều kiện if
   pre->next = cur->next; //trường hợp tìm được node cần xóa thì ta cho con trỏ pre trỏ đến node kế tiếp của node cur để tạo mối liên kết
   delete cur; //xóa node cur cũng chính là node cần xóa
}
//in danh sach
void Print (Node *head) //tạo hàm có tên Print truyền vào 1 Node tên head dùng để in ra dslk
{
   Node *cur = head; //khai báo và tạo 1 Node tên cur trỏ đến head dùng để duyệt dslk
   while (cur) //điều kiện để thực hiện vòng lặp
   {
      cout << cur->data << " "; //xuất ra giá trị data của node
      cur = cur->next; //di chuyển đến node tiếp theo
```

```
    }
    cout << endl;
}
int main ()
{
    Node *head;
    head = initNode(1);
    addNode (head, 2);
    addNode (head, 3);
    addNode (head, 4);
    addNode (head, 5);
    InsertFront (&head,0);
    Print (head);

    head = searchNode (head,1);
    if (head != 0)
    {
        cout<<"Co node trong List.\n";
    }

    deleteNode(&head,0);
    cout <<"List is: ";
    Print(head);
    return 0;
}
```

Câu III

a. **Trình bày ngăn xếp (stack) là gì? Cấu trúc và cơ chế hoạt động ra sao?**

Ngăn xếp stack là 1 cái cấu trúc dữ liệu dùng để lưu trữ dữ liệu theo nguyên tắc LAST IN FIRST OUT

Để cài 1 ngăn xếp ta có thể dùng mảng hoặc danh sách liên kết

b. **Trình bày và giải thích các thao tác cơ bản của ngăn xếp.**

Các thao tác cơ bản;

- Khởi tạo stack dùng để khởi tạo stack rỗng
- Thêm dữ liệu vào top được gọi là push(data) để thêm phần tử data vào stack
- Xóa dữ liệu trên top được gọi là pop() dùng để xóa 1 phần tử ở đầu ngăn xếp

- Thao tác lấy dữ liệu peek() trả về giá trị phần tử ở đỉnh stack. Số phần tử không thay đổi
- Kiểm tra stack có rỗng không isEmpty()

c. Dùng ngôn ngữ lập trình C++ viết hàm thực hiện các thao tác trên. (sinh viên được khuyến khích cho việc giải thích ý nghĩa các biến cũng như các cấu trúc rẽ nhánh, vòng lặp của chương trình).

```
#include <iostream>
using namespace std;

struct StackNode {
    int data;
    struct StackNode* next;
};

struct StackNode* newNode (int data) //khai báo 1 hàm có tên là newNode truyền vào 1 biến data và trả về 1 Stacknode dùng để tạo mới 1 stackNode
{
    struct StackNode* stackNode = (struct StackNode*)malloc(sizeof(struct StackNode)); //khai báo và sử dụng malloc cấp phát bộ nhớ động để tạo một StackNode mới
    stackNode->data = data; //giá trị của stackNode = giá trị data truyền vào
    stackNode->next = NULL; //do vừa khởi tạo nên stackNode ->next = NULL
    return stackNode; //trả về stackNode
}

int isEmpty (struct StackNode* root) //tạo 1 hàm có tên isEmpty truyền vào 1 StackNode để kiểm tra danh sách có rỗng không
{
    return !root; //
}

void push (struct StackNode** root, int data) //tạo hàm có tên push để thêm giá trị data vào stack, truyền vào 1 StackNode do khi thêm phần tử vào, StackNode sẽ thay đổi nên ta truyền tham chiếu và 1 biến data là giá trị sẽ thêm vào stack
{
    struct StackNode* stackNode = newNode(data); //khai báo và tạo 1 StackNode mới có tên stackNode
    stackNode->next = *root; //phần tử mới thêm vào sẽ lên kết với các phần tử cũ
```

```
    *root = stackNode; //cập nhật lại stack sau khi đã thêm data vào
    printf("%d pushed to Stack\n", data); //xuất ra giá trị vừa thêm vào stack
}

int pop (struct StackNode** root) //tạo 1 hàm có tên pop dùng để xóa phần tử ở đỉnh stack, truyền vào 1 StackNode do khi xóa phần tử ở đỉnh stack sẽ thay đổi nên ta truyền tham chiếu, hàm trả về giá trị đã xóa
{
    if (isEmpty(*root)) //kiểm tra stack rỗng hay không
        return INT_MIN; //nếu rỗng trả về min
    struct StackNode* temp = *root; //khai báo và tạo 1 biến kiểu StackNode tên temp trỏ đến root dùng để lưu lại node hiện tại để tiếp tục thao tác công việc xóa
    *root = (*root)->next; //di chuyển trỏ root sang node kế tiếp
    int popped = temp->data; //khai báo biến kiểu int tên popped gán = temp->data để lưu giá trị của phần tử mình sắp xóa
    free(temp); //xóa phần tử
    return popped; //trả về giá trị vừa xóa
}

int peek (struct StackNode* root) //khai báo hàm có tên peek kiểu int truyền vào 1 StackNode, hàm trả về giá trị phần tử ở đỉnh stack
{
    if (isEmpty(root)) //kiểm tra stack có rỗng không
        return INT_MIN; //trả về min nếu rỗng
    return root->data; //trả về phần tử ở đỉnh stack
}

int main ()
{
    struct StackNode* root = NULL;
    push (&root, 1);
    push (&root, 2);
    push (&root, 3);
    printf ("%d popped from stack\n", pop(&root));
    printf ("Top element: %d\n", peek(root));
```

d. Giải thích và viết chương trình ứng dụng ngăn sếp (stack) để thực hiện việc chuyển đổi một số từ hệ cơ số 10 sang một số mới với cơ số b bất kỳ (2 <= b <= 9).

```
 int n, b; //khai báo 2 biến n và b là một số từ hệ cơ số 10 và cơ số b bất kì (2 <= b <=9)
   cout<<"Nhap vao 1 so n : ";
   cin>>n; //nhập vào số từ hệ cơ số 10
   cout<<"Nhap vao co so b: ";
   cin>>b; //nhập cơ số b bất kì (2 <= b <=9)
   int temp = n; //khai báo và tạo 1 biến temp gán = số hệ cơ số 10
   while(temp > 0) //điều kiện số cần chuyển đổi là số nguyên dương > 0
   {
      int le = temp % b; //khai báo vào tạo 1 biến le kiểu int gán = kết quả phép chia lấy dư của 2 số là số ta cần chuyển đổi và cơ số b do rằng muốn chuyển đổi cơ số ta thực hiện phép chia lấy dư
      push(&root, le); //ta bỏ kết quả phép chia trên vào stack
      temp /= b; //lấy biến temp chia cho cơ số b để tiếp tục việc chuyển đổi cơ số cho đến khi không còn thỏa mãn điều kiện
   }
      while(!isEmpty(root)) //nếu stack khác rỗng
      cout<<pop(&root); //xuất ra kết quả là số được chuyển đổi từ hệ cơ số 10

   return 0;
}
```

Câu IV

a. Trình bày cây nhị phân tìm kiếm là gì? Trình bày và giải thích các thao tác cơ bản trên cây nhị phân tìm kiếm.

Cây nhị phân tìm kiếm là tập hợp các node mỗi node có nhiều nhất là 2 nhánh con, các giá trị của các node bên phía trái của 1 node nó sẽ nhỏ hơn giá trị của node đó, còn các giá trị node ở phía bên phải của node đó sẽ có giá trị lớn hơn node đó.

Các thao tác cơ bản:

1. Khởi tạo: dùng để tạo một node mới của cây nhị phân tìm kiếm
2. Tìm kiếm: để tìm kiếm 1 node có giá trị n bất kì, ta cần so sánh giá trị cần tìm đó với node gốc. Sẽ có 3 trường hợp xảy ra:
    - Node gốc = NULL thì node cần tìm không có
    - Node gốc khác n và node gốc khác rỗng thì:
    - Tìm node cần tìm theo nhánh con bên trái

- Tìm node cần tìm theo nhánh con bên phải

3. Chèn: công dụng dùng để thêm 1 node có giá trị n bất kì vào cây. Có 3 trường hợp:
    - Nếu node gốc = NULL thì tức là cây rỗng nên ta chỉ cần gọi hàm tạo mới 1 node với giá trị n truyền vào để thêm vào cây
    - Nếu n < node gốc thì thêm node ở cây con bên trái
    - Nếu n > node gốc thì thêm node ở cây con bên phải
4. Phép duyệt: có 3 cách duyệt cây nhị phân:
    - NLR: duyệt nút gốc, duyệt NLR con trái rồi đến duyệt NLR con phải
    - LNR: duyệt LNR con trái, duyệt nút gốc, rồi duyệt LNR con phải
    - LRN: duyệt LRN con trái, duyệt LRN con phải, nút gốc
5. Xóa: đầu tiên cần tìm node cần xóa có tồn tại trong cây hay không. Nếu Node cần xóa có tồn tại thì cần xác định giá trị node cần xóa > hay < node gốc
    - Nếu giá trị node cần xóa < node gốc thì duyệt bên con trái để tìm ra node cần xóa
    - Nếu giá trị node cần xóa > node gốc thì duyệt bên con phải để tìm ra node cần xóa

    Sau khi tìm ra được node cần xóa có 3 trường hợp:
    - Node cần xóa là node lá thì chỉ cần hủy node cần xóa là xong
    - Node cần xóa có 1 node con thì cho cái gốc chỉ đến node con của node cần xóa sau đó hủy node cần xóa
    - Nếu node cần xóa có 2 node con thì cần tìm node con nhỏ nhất bên trái (hoặc bên phải đều được) để thay thế giá trị node cần xóa = giá trị node con nhỏ nhất, sau đó xóa node con nhỏ nhất
6. Hủy cây: để thực hiện việc hủy cây nhị phân tìm kiếm, đầu tiên phải thực hiện duyệt LRN hay có thể hiểu rằng ta xóa từ nút lá lên đến gốc

b. Dùng ngôn ngữ lập trình C++ để thực hiện các thao tác trên. (sinh viên được khuyến khích cho việc giải thích ý nghĩa các biến cũng như các cấu trúc rẽ nhánh, vòng lặp của chương trình).

```
#include <iostream>
#include <math.h>
using namespace std;

struct Node {
   int indexdata;
   Node *left;
```

```
    Node *right;
};
//Hàm khởi tạo 1 node
struct Node *createNode (int n) // khai báo 1 hàm có tên là createNode trả về 1 node, truyền vào 1 biến n là giá trị của node đó
{
    Node *newNode = new Node;  // khai báo và tạo 1 node mới có tên là newNode
    newNode->left = NULL;  // khi vừa khởi tạo nên cây con trái = NULL
    newNode->right = NULL;  // khi vừa khởi tạo nên cây con phải = NULL
    newNode->indexdata = n;  // giá trị của newNode = giá trị n ta truyền vào hàm
    return newNode; // trả về 1 node
}

struct Node *InsertNode (Node *root, int n) //1 hàm có tên là InsertNode dùng để chèn 1 node mới vào cây có giá trị n, đầu vào ta truyền vào hàm 1 cái gốc và 1 biến n. Hàm trả về 1 gốc
{
    if (root == NULL) // nếu gốc = NULL
        root = createNode(n); // ta gọi hàm tạo mới để chèn 1 node với giá trị n
    if (n < root->indexdata) // nếu giá trị của node mới ta muốn thêm vào < giá trị node gốc
        root->left = InsertNode (root->left, n); // thì ta gọi đệ quy để thêm node vào bên trái
    if (n > root->indexdata) // nếu giá trị của node mới ta muốn thêm vào > giá trị node gốc
        root->right = InsertNode (root->right, n); // thì ta gọi đệ quy để thêm node vào bên phải
    return root; // trả về 1 cái gốc
}

struct Node *minValue(struct Node *root) //khai báo hàm có tên minValue truyền vào 1 cái gốc và trả về 1 node dùng để tìm node con nhỏ nhất bên trái phục vụ cho việc xóa Node có 2 con
{
    Node *cur = root; //khai báo và tạo 1 Node cur và gán = root dùng để duyệt để tìm node con nhỏ nhất bên trái
```

```
    while(cur != NULL && cur->left != NULL) //vòng lặp với điều kiện cur != NULL và cur->left != NULL vì cần tìm node con nhỏ nhất bên trái nên cur->left != NULL nếu cur->left = NULL tức là đã tìm đc node cần tìm nên sẽ dừng vòng lặp
    {
        cur = cur->left; // nếu thỏa điều kiện thì duyệt đến node kế tiếp
    }
    return cur; // trả về cái node cần tìm
}
struct Node *deleteNode(struct Node *root, int data) //khai báo hàm có tên deleteNode và truyền vào 1 gốc và 1 biến data là giá trị của node cần xóa, hàm trả về 1 cái gốc
{
    if(root == NULL) //nếu root = NULL đồng nghĩa không có node cần xóa trong cây chỉ cần trả về gốc
        return root; // trả về gốc


    if(data < root->indexdata) // nếu giá trị của node cần xóa < node gốc thì ta đi tìm node cần xóa đó bên cây con trái
        root->left = deleteNode(root->left,data); // gọi đệ qui tìm node cần xóa bên cây con trái
    else if (data > root->indexdata) // nếu giá trị của node cần xóa < node gốc thì ta đi tìm node cần xóa đó bên cây con phải
        root->right = deleteNode(root->right,data); // gọi đệ qui tìm node cần xóa bên cây con trái
    else //lúc này nếu node cần xóa có tồn tại trong cây thì ta sẽ thi hành các câu lệnh trong hàm else tức là hiện tại ta đã tìm ra node cần xóa (data == root->indexdata)
    {
        //1 child or leaf
        if(root->left == NULL) // nếu node con của node cần xóa = NULL thì đây là cây con phải
        {
            Node *temp = root->right; // khai báo và tạo một Node temp và gán = giá trị của cây con phải của node cần xóa để lưu lại giá trị cây con phải node cần xóa
            delete root; // sau đó xóa node cần xóa
            return temp; // return temp để cập nhật lại mối liên kết của node gốc với temp
        }
        else if(root->right == NULL) // nếu node con của node cần xóa = NULL thì đây là cây con trái
```

```
        {
            Node *temp = root->left; // khai báo và tạo một Node temp và gán = giá trị của cây con trái của node cần xóa để lưu lại giá trị của cây con trái node cần xóa
            delete root; // sau đó xóa node cần xóa
            return temp; ; // return temp để cập nhật lại mối liên kết của node gốc với temp
        }
        //2 child
        Node *temp = minValue(root->right); // khai báo và tạo 1 node temp và gán = giá trị của node con nhỏ nhất bên trái
        root->indexdata = temp->indexdata; // gán node cần xóa = giá trị của node con nhỏ nhất bên trái
        root->right = deleteNode(root->right, temp->indexdata); // gọi đệ quy để xóa node con nhỏ nhất bên trái
    }
      return root; // trả về gốc
}

void DestroyTree (Node *root)  //tạo hàm có tên là DestroyTree truyền vào 1 cái gốc
{
    if (root != NULL) //điều kiện nếu root != NULL
    {
        DestroyTree(root->left); //gọi đệ quy hủy bên trái
        DestroyTree(root->right); //gọi đệ quy hủy bên phải
        delete root; // hủy node gốc
    }
}

Node *findNode (Node *root, int n) //tạo hàm có tên finNode truyền vào 1 cái gốc và 1 biến n là giá trị của node cần tìm, hàm trả về 1 node
{
    if (root == NULL) //nếu root == NULL tức là không có node cần tìm
        return NULL; //nên trả về NULL
    if (n < root->indexdata) //nếu giá trị cần tìm < node gốc
        return findNode (root->left, n); //thì gọi đề quy tìm node bên cây con trái
    if (n > root->indexdata) //nếu giá trị cần tìm > node gốc
```

```
        return findNode (root->right, n); //thì gọi đề quy tìm node bên cây con phải
    return root; //trả về node cần tìm = NULL nếu không tìm được
}

void printLNR (Node *root) //tạo hàm có tên là printLNR truyền vào 1 gốc để duyệt cây
{
    if (root != NULL) //nếu cây khác rỗng
    {
        printLNR(root->left); //duyệt bên trái
        cout << root->indexdata << " ";  //duyệt node gốc
        printLNR(root->right); //duyệt bên phải
    }
}

void printNLR (Node *root) //tạo hàm có tên là printNLR truyền vào 1 gốc để duyệt cây
{
        if (root != NULL) //nếu cây khác rỗng
       {
                cout << root->indexdata <<" "; //duyệt node gốc
                printNLR(root->left); //duyệt bên trái
                printNLR(root->right); //duyệt bên trái
       }
}

void printRNL (Node *root) //tạo hàm có tên là printRNL truyền vào 1 gốc để duyệt cây
{
                if (root != NULL) //nếu cây khác rỗng
    {
                printRNL(root->right); //duyệt bên trái
                cout << root->indexdata << " "; //duyệt node gốc
                printRNL(root->left); //duyệt bên trái
    }
}
```

c. Viết và giải thích chương trình đếm số nút trong cây có giá trị là số nguyên tố

```
bool SNT (int n) //tạo hàm có tên là SNT kiểu bool dùng để kiểm tra giá trị truyền vào có phải là 1 số nguyên tố hay không, truyền vào hàm 1 biến n
{
	if (n < 2)  //nếu giá trị truyền vào nhỏ hơn 2
	{
		return false; //thì số đó không phải là số nguyên tố
	}
	else  //nếu số đó > 2
	{
		for (int i = 2;i <= sqrt(n);i++) //thì ta cần duyệt từ 2 đến căn bậc 2 của số đó
		{
			if (n % i == 0) //xét từ 2 đến căn bậc 2 của số đó có bất kì số nào mà số đang cần xét chia hết không. Nếu có
				return false; //thì số đó không phải là số nguyên tố
		}
	}
	return true; //không rơi vào các trường hợp trên kia thì số này là số nguyên tố
}

int demsonguyento (Node *root, int &count) //tạo 1 hàm có tên demsonguyento dùng để đếm số lượng số nguyên tố có trong cây. Truyền vào 1 cái gốc và 1 biến count do biến count sẽ thay đổi giá trị sau mỗi lần đếm nên ta cần truyền vào tham chiếu để lưu lại giá trị. Hàm trả về biến count cũng chính số lượng số nguyên tố có trong cây
{
	if (root != NULL) //nếu cây khác rỗng
	{
		if (SNT(root->indexdata) == true) //gọi lại hàm kiểm tra số nguyên tố nếu kết quả = true tức là giá trị truyền vào đó là số nguyên tố
		{
			count++; //nếu là số nguyên tố thì tăng biến count lên 1
		}
		demsonguyento(root->left, count); //gọi đệ quy duyệt sang bên trái để đếm số nguyên tố
		demsonguyento(root->right, count); //gọi đệ quy duyệt sang bên phải để đếm số nguyên tố
```

```
            }
        return count; //trả về số lượng số nguyên tố có trong cây
}
int main ()
{
    Node *t = NULL;
    t = createNode (5);
    t = InsertNode (t, -5);
    t = InsertNode (t, 8);
    t = InsertNode (t, 7);
    t = InsertNode (t, 15);
    t = InsertNode (t, 14);
    t = InsertNode (t, 13);
    t = InsertNode (t, 20);
    printLNR (t);
    cout <<endl;

    deleteNode (t, 5);
    printLNR (t);
    //Node *f = findNode(t, 1);
    //if (f == NULL)
    {
        //cout <<"Khong co Node trong cay.\n";
    }
    //else
    {
        //cout <<"Co node trong cay.\n";
    }

    //cout <<"Destroy Tree";
    //DestroyTree (t);
    //printLNR (t);

    //cout <<"NLR: ";
    //printNLR(t);
```

```
    //cout<<"RNL: ";
    //printRNL(t);

    int count = 0;
    cout<<"So luong so nguyen to: " <<demsonguyento(t, count);
}
```

----Hết---