1. **Các hàm cơ bản trong C#**
2. **Fibonacci**

Viết phương thức Fibonacci( ) như sau: - Dãy số Fibonacci bậc 1 gồm các số F0, F1, F2, F3, F4, F5, F6, … là dãy 0, 1, 1, 2, 3, 5, 8, … - Nhập vào số nguyên n ≥ 0. - Phương thức trả về một số nguyên là số Fn theo hai cách: dùng giải thuật đệ qui và dùng giải thuật không đệ qui. - Gợi ý: Giải thuật đệ qui: F0 = 0, F1 = 1, Fn = Fn-1 + Fn-2 Giải thuật không đệ qui: dùng ba biến a, b, c để lưu ba số Fibonacci kế tiếp nhau.

static int fibonacci(int n)

{

int f0 = 0, f1 = 1, fn = 1;

if (n==0 || n == 1) return n;

else

{

for (int i = 2; i < n; i++)

{

f0 = f1;

f1 = fn;

fn = f0 + f1;

}

}

return fn;

}

Recursion:

static int fibonacci(int n)

{

if (n == 0 || n ==1) return n;

else return (fibonacci(n-1) + fibonacci(n-2));

}

1. **Neper**

Viết phương thức Neper( ) như sau: - Số e là tổng của các số hạng ak = 1/(k!) với k = 0, 1, 2, … - Nhập vào số nguyên n ≥ 0. - Phương thức trả về tổng của a0 + a1 + … +an. - Gợi ý: Xét sự liên hệ giữa hai số hạng kế tiếp nhau ai và ai+1.

static float gt(float m)

{

float gthua;

if (m == 0) gthua = 1;

else gthua = m\*gt(m-1);

return gthua;

}

static float Neper(float n)

{

float kq;

if (n == 0) kq = 1;

else

kq = Neper(n-1)+(1/gt(n));

return kq;

}

1. **GCD**

Viết phương thức GCD( ) như sau: - Nhập vào hai số nguyên dương m và n. - Phương thức này trả về ước số chung lớn nhất (GCD – Greatest Common Divisor) của m và n theo hai cách: dùng giải thuật đệ qui và dùng giải thuật không đệ qui. - Ví dụ: ước số chung lớn nhất của 372 và 84 là 12. - Gợi ý: Tìm GCD(372, 84): 372 chia 84 dư 36 Tìm GCD(84, 36): 84 chia 36 dư 12 Tìm GCD(36, 12): 36 chia 12 dư 0 Tìm GCD(12, 0): kết thức. Vậy ước số chung lớn nhất của 372 và 84 là 12.

Cách 1:

static int GCD(int n1, int n2)

{

int tmp;

while(n2 != 0) {

tmp = n1 % n2;

n1 = n2;

n2 = tmp;

}

return n1;

}

Recursion:

static int GCD(int n1, int n2)

{

if (n2 == 0) return n1;

return GCD(n2, n1%n2);

}

Cách 2:

static int GCD(int n1, int n2) => (n1 == 0) ? n2 : GCD(n2 % n1, n1);

Cách 3: Chỉ dùng cho Cpp, không đệ quy;.

int GCD(int n1, int n2)

{

while (n1^=n2^=n1^=n2%=n1)

return n2;

1. **Pascal Triangle**

Viết phương thức Pascal( ) như sau: - Nhập vào số nguyên dương n. - Phương thức này sẽ in ra tam giác Pascal ứng với bậc n. - Ví dụ n = 4 thì tam giác Pascal là:

n=0 1

n=1 1 1

n=2 1 2 1

n=3 1 3 3 1

n=4 1 4 6 4 1

static int pas(int hang, int cot)

{

if (cot == 0 || hang == cot) return 1;

return pas(hang-1, cot-1) + pas(hang-1, cot);

}

1. **Number**

Viết phương thức Number( ) như sau: - n là một số nguyên dương và s là tổng các ước số của nó (kể cả số 1). - n là deficient nếu s < n - n là perfect nếu s = n - n là abundant nếu s = n. - Nhập vào hai số nguyên dương x và y với x ≤ y. - Phương thức sẽ in ra phân loại (deficient, perfect, abundant) của các số từ x đến y. - Ví dụ: số 8 là deficient vì 1 + 2 + 4 < 8; số 6 là perfect vì 1 + 2 + 3 = 6; số 12 là abundant vì 1 + 2 + 3 + 4 + 6 > 12.

static int number(int num, int x)

{

int sumcd;

if(x==1)

sumcd = 1;

else if(num%x==0)

sumcd = x + number(num,x-1);

else

sumcd = number(num,x-1);

return sumcd;

}

private static void Main(string[] args)

{

Console.WriteLine("Nhap: ");

int num = Convert.ToInt32(Console.ReadLine());

Console.WriteLine("Tong cua uoc: {0}", number(num, num/2));

if ( number(num, num/2) < num) Console.WriteLine("deficient");

else if ( number(num, num/2) > num) Console.WriteLine("abundant");

else Console.WriteLine("perfect");

}

1. **Đối xứng**

Viết phương thức mang.DoiXung( ) như sau: - mang là một ma trận vuông, mỗi phần tử chứa một số nguyên. - Phương thức trả về true nếu mang là ma trận đối xứng, ngược lại trả về false.

public static bool DoiXung(int[,] mang)

{

int n = mang.GetLength(0);

for (int i = 0; i < n; i++)

{

for (int j = 0; j < n; j++)

{

if (mang[i,j] != mang[j,i])

{

return false;

}

}

}

return true;

}

1. **Matrix tam giác trên**

Viết phương thức mang.TamGiacTren( ) như sau: - mang là một ma trận vuông, mỗi phần tử chứa một số nguyên. - Phương thức trả về true nếu mang là ma trận tam giác trên, ngược lại trả về false.

public static bool TamGiacTren(int[,] mang)

{

int n = mang.GetLength(0);

for (int i = 0; i < n; i++)

{

for (int j = 0; j < i; j++)

{

if (mang[i,j] != 0)

{

return false;

}

}

}

return true; }

1. **Trùng hàng**

Viết phương thức mang.TrungHang( ) như sau: - mang là một ma trận vuông, mỗi phần tử chứa một số nguyên. - Phương thức trả về true nếu mang có ít nhất hai hàng giống nhau, ngược lại trả về false.

public static bool TrungHang(int[,] mang)

{

int n = mang.GetLength(0);

for (int i = 0; i < n; i++)

{

for (int j = i + 1; j < n; j++)

{

bool trung = true;

for (int k = 0; k < n; k++)

{

if (mang[i, k] != mang[j, k])

{

trung = false;

break;

}

}

if (trung)

{

return true;

}

}

}

return false;

}

1. **Nhóm hàng**

. Viết phương thức mang.NhomHang( ) như sau: - mang là một ma trận vuông, mỗi phần tử chứa một số nguyên. - Phương thức in ra nhóm chỉ mục hàng của các hàng giống nhau, mỗi nhóm được bắt đầu in ở đầu hàng màn hình.

public static void NhomHang(int[,] mang)

{

int soHang = mang.GetLength(0); // lấy số hàng của mảng

for (int i = 0; i < soHang; i++)

{

if (i == 0 || !SoSanhHang(mang, i, i - 1)) // Nếu là hàng đầu tiên hoặc không giống hàng trước đó

{

Console.Write("Nhóm hàng " + i + ": ");

}

Console.Write(mang[i, 0] + " ");

// Nếu đã đến hàng cuối cùng hoặc hàng kế tiếp không giống hàng hiện tại

if (i == soHang - 1 || !SoSanhHang(mang, i, i + 1))

{

Console.WriteLine();

}

}

}

// Hàm SoSanhHang để kiểm tra hai hàng có giống nhau hay không

public static bool SoSanhHang(int[,] mang, int hang1, int hang2)

{

int soCot = mang.GetLength(1); // lấy số cột của mảng

for (int j = 0; j < soCot; j++)

{

if (mang[hang1, j] != mang[hang2, j])

{

return false;

}

}

return true;

}

1. **Cộng**

. Viết phương thức a.Cong(b) như sau: - a và b là các số nguyên không dấu được chứa trong mảng một chiều, mỗi phần tử chứa một ký số. - Phương thức trả về một số nguyên là kết quả của a + b nếu kết quả không bị tràn hoặc trả về mảng gồm các số -1 nếu kết quả bị tràn.

static int[] Cong(int[] a, int[] b)

{

int len = Math.Max(a.Length, b.Length);

int[] res = new int[len];

int carry = 0;

for (int i = 0; i < len; i++)

{

int digitA = (i < a.Length) ? a[a.Length - 1 - i] : 0;

int digitB = (i < b.Length) ? b[b.Length - 1 - i] : 0;

int sum = digitA + digitB + carry;

if (sum >= 10)

{

carry = 1;

sum -= 10;

}

else

{

carry = 0;

}

res[len - 1 - i] = sum;

}

if (carry == 1)

{

return new int[] { -1 };

}

return res;

}

1. **Trừ**

Viết phương thức a.Tru(b) như sau: - a và b là các số nguyên không dấu được chứa trong mảng một chiều, mỗi phần tử chứa một ký số và a ≥ b. - Phương thức trả về một số nguyên là kết quả của a - b.

public static int[] Tru(int[] a, int[] b)

{

int n = a.Length;

int m = b.Length;

// Kiểm tra độ dài của a và b

if (n < m || (n == m && a[0] < b[0]))

{

return new int[] {-1}; // Kết quả không hợp lệ

}

// Khởi tạo mảng kết quả

int[] res = new int[n];

// Thực hiện phép trừ

int borrow = 0;

for (int i = n - 1, j = m - 1; j >= 0; i--, j--)

{

int diff = a[i] - b[j] - borrow;

if (diff < 0)

{

diff += 10;

borrow = 1;

}

else

{

borrow = 0;

}

res[i] = diff;

}

for (int i = n - m - 1; i >= 0; i--)

{

if (a[i] == 0 && borrow == 1)

{

res[i] = 9;

borrow = 1;

}

else

{

int diff = a[i] - borrow;

if (i > 0 || diff > 0)

{

res[i] = diff;

}

borrow = 0;

1. **Nhân**

Viết phương thức a.Nhan(b) như sau: - a và b là các số nguyên không dấu được chứa trong mảng một chiều, mỗi phần tử chứa một ký số. - Phương thức trả về một số nguyên là kết quả của a × b nếu kết quả không bị tràn hoặc trả về mảng gồm các số -1 nếu kết quả bị tràn.

static int[] Nhan(int[] a, int[] b)

{

int m = a.Length;

int n = b.Length;

// Khởi tạo mảng kết quả

int[] res = new int[m + n];

// Đặt tất cả các phần tử của mảng kết quả bằng 0 ban đầu

for (int i = 0; i < m + n; i++)

{

res[i] = 0;

}

// Nhân từng chữ số của a với toàn bộ b, lưu kết quả vào mảng tạm

for (int i = 0; i < m; i++)

{

int carry = 0;

for (int j = 0; j < n; j++)

{

int temp = a[i] \* b[j] + res[i + j] + carry;

carry = temp / 10;

res[i + j] = temp % 10;

}

if (carry > 0)

{

res[i + n] += carry;

}

}

// Loại bỏ các số 0 ở đầu mảng kết quả

int k = m + n - 1;

while (k > 0 && res[k] == 0)

{

k--;

}

// Kiểm tra kết quả có bị tràn hay không

if (k >= 9)

{

// Kết quả bị tràn, trả về mảng gồm các số -1

for (int i = 0; i < m + n; i++)

{

res[i] = -1;

}

return res;

}

// Đảo ngược thứ tự các phần tử trong mảng kết quả

int[] result = new int[k + 1];

for (int i = k; i >= 0; i--)

{

result[k - i] = res[i];

}

return result;

}

1. **Cấu trúc dữ liệu:**
2. **Linked List:**

Linked List sử dụng 1 số hàm để thực hiện các chức năng sau:

* Phương thức Node: xác định một nút (node) trong danh sách liên kết, bao gồm hai thành phần: giá trị của nút (element) và nút tiếp theo trong danh sách (next).
* Phương thức LinkedList: khởi tạo danh sách liên kết rỗng.
* Phương thức length: trả về số lượng phần tử trong danh sách liên kết.
* Phương thức isempty: kiểm tra xem danh sách liên kết có rỗng hay không.
* Phương thức addFirst: thêm một phần tử vào đầu danh sách liên kết.
* Phương thức addLast: thêm một phần tử vào cuối danh sách liên kết.
* Phương thức addany: thêm một phần tử vào vị trí chỉ định trong danh sách liên kết.
* Phương thức removeFirst: xóa phần tử đầu tiên của danh sách liên kết.
* Phương thức removeLast: xóa phần tử cuối cùng của danh sách liên kết.
* Phương thức removeAny: xóa phần tử ở vị trí chỉ định trong danh sách liên kết.
* Phương thức search: tìm kiếm phần tử có giá trị bằng key trong danh sách liên kết và trả về chỉ số đầu tiên nếu tìm thấy, hoặc -1 nếu không tìm thấy.
* Phương thức display: hiển thị tất cả các phần tử trong danh sách liên kết.

Code:

*//Tạo đối tượng Node cho DSLK đơn, Định nghĩa thuộc tính element và next của lớp Node. Thuộc tính element được sử dụng để lưu trữ giá trị của phần tử trong danh sách liên kết. Thuộc tính next là một tham chiếu đến Node tiếp theo trong danh sách liên kết.*

public class Node

{

public int element;

public Node next;

public Node(int e, Node n)

{

element = e;

next = n;

}

}

*//định nghĩa class LinkedList bao gồm thuộc tính Head, tail, size để* lưu trữ nút đầu tiên, nút cuối cùng và kích thước của LL. Thuộc tính *next tham chiếu đến Node tiếp theo của DSLK.*

class LinkedList

{

private Node head;

private Node tail;

private int size;

//tạo một phương thức LinkedList() để khởi tạo giá trị Node rỗng cho DSLK

public LinkedList()

{

head = null;

tail = null;

size = 0;

}

//Trả về kích thước của danh sách liên kết

public int length()

{

return size;

}

//phương thức kiểm tra xem DSLK có rỗng không, phương thức dạng bool, nếu size khác 0, hàm cho ra false và ngược lại

public bool isempty()

{

return size == 0;

}

*//phương thức thêm vào đầu liên kết*

public void addFirst(int e)

{

Node newest *);//khởi tạo một node có giá trị là element và next = null* = new Node(e, null

*//kiểm tra xem DSLK có rỗng hay không, nếu rỗng thì sẽ khởi tạo node này là phần đầu cũng như là phần đuôi của dslk*

if (isempty())

{

head = newest;

tail = newest;

}

//*ngược lại, tiến hành trỏ thành phần cần liên kết vào head và gán head bằng node mới.*

else

{

newest.next = head;

head = newest;

}

size++;

}

*//Hàm addLast thêm một phần tử liên kết vào cuối danh sách liên kết, ban đầu, kiểm tra xem danh sách liên kết có rỗng hay không, nếu rỗng, phần tử mới nhất sẽ là head, ngược lại, ta trỏ phần đuôi vào node mới, sau đó gán tail thành node mới, từ đó, tail chính là node mới và kích thước tăng thêm 1.*

public void addLast(int e)

{

Node newest = new Node(e, null);

if (isempty())

head = newest;

else

tail.next = newest;

tail = newest;

size = size +1;

}

*//Hàm addany thêm phần tử e vào vị trí tại position, đầu tiên ta kiểm tra giá trị có hợp lệ hay không (<0, >size), ngược lại là có danh sách, ta dịch con trỏ p đến vị trí position, sau đó*

*giá trị tiếp theo của giá trị cần gán là giá trị tiếp theo của p (newest.next = p.next), và sau đó gán giá trị của p tiếp theo là phân tử cần gán.*

public void addany(int e, int position)

{

Node newest = new Node(e,null);

Node p = head;

int i = 1;

if (position <= 0 || position >=size)

{

Console.WriteLine("Invalid");

return;

}

while (i < (position-1))

{

p = p.next;

i++;

}

newest.next = p.next;

p.next = newest;

size++;

}

*//Hàm removeFirst() xóa bỏ phần tử đầu của dslk, ban đầu ktra xem có dslk hay không, nếu không thì ta có giá trị e là giá trị đầu danh sách, gán head vào phần tử tiếp theo của đầu danh sách hiện tại và tiến hành giảm size đi 1 vì đã xóa đi 1 phần tử. Lệnh kiểm tra xem có rỗng hay không ở bước cuối có hay không thì không quan trọng.*

public int removeFirst()

{

if ((isempty()))

{

Console.WriteLine("List is Empty");

return -1;

}

else

{

int e = head.element;

head = head.next;

size = size - 1;

if (isempty())

tail = null;

return e;

}

}

*//Phương thức removeLast() xóa phần tử cuối cùng trong dslk, ban đầu vẫn là bước kiểm tra xem đang có dslk hay không. Nếu có, tạo node p là head của dslk và tiến hành dịch con trỏ p giống như addAny nhưng ở đây position thay bằng kích cỡ của dslk hiện tại, sau đó gán đuôi là p, sau đó gán p là vị trí p tiếp theo và cuối cùng gán e là giá trị mà con trỏ p trỏ đến, xóa phần tử tiếp theo và khi này kích thước giảm đi 1 và gán vào size.*

public int removeLast()

{

if (isempty())

{

Console.WriteLine("List is Empty");

return -1;

}

Node p = head;

int i = 1;

while (i < size - 1)

{

p = p.next;

i = i + 1;

}

tail = p;

p = p.next;

int e = p.element;

tail.next = null;

size--;

return e;

}

*//Phương thức removeAny xóa phần tử ở 1 vị trí bất kì, ban đầu vẫn là kiểm tra xem có phần tử hay không, sau đó tạo 1 node p và gán là header, sau đó dịch con trỏ p đến vị trí position và gán giá trị tiếp theo cho e và gán giá trị của con trỏ tiếp theo là giá trị của con trỏ tiếp theo nữa (để bỏ qua giá trị bị xóa).*

public int removeAny(int position)

{

if (position <=0 || position >= size - 1)

{

Console.WriteLine("Invalid");

return -1;

}

Node p = head;

int i = 1;

while (i < position)

{

p = p.next;

i++;

}

int e = p.next.element;

p.next = p.next.next;

size--;

return e;

}

*//Hàm search đơn giản là tìm phần tử key, nếu không phải key thì cộng index, ngược lại xuất ra giá trị index hiện tại, nếu không có thì báo lỗi và xuất ra phương thức.*

public int search(int key)

{

Node p = head;

int index = 0;

while(p != null)

{

if (p.element == key)

{

return index;

}

p = p.next;

index++;

}

return -1;

}

*//Hàm này quá dễ, k giải thích*

public void display()

{

Node p = head;

while (p != null )

{

Console.Write(p.element + "--->");

p = p.next;

}

Console.WriteLine();

}

}

Đối với Doubly LinkedList, giải thích tương tự dslk đơn nhưng chỉ có thêm prev ở mỗi node thì cấu trúc này, mỗi node mang tính 2 chiều

Code:

class DoublyLinkedList

{

private Node head;

private Node tail;

private int size;

public DoublyLinkedList()

{

head = null;

tail = null;

size = 0;

}

public int length()

{

return size;

}

public bool isEmpty()

{

return size == 0;

}

public void addLast(int e)

{

Node newest = new Node(e, null, null);

if (isEmpty())

{

head = newest;

tail = newest;

}

else

{

tail.next = newest;

newest.prev = tail;

tail = newest;

}

size++;

}

public void addFirst(int e)

{

Node newest = new Node(e,null,null);

if (isEmpty())

{

head = newest;

tail = newest;

}

else

{

newest.next = head;

head.prev = newest;

head = newest;

}

size++;

}

public void addany(int e, int position)

{

if(position <= 0 || position >= size)

{

Console.WriteLine("Invalid");

return;

}

Node newest = new Node(e,null,null);

Node p = head;

int i = 1;

while (i < position-1)

{

p = p.next;

i = i+1;

}

newest.next = p.next;

p.next.prev = newest;

p.next = newest;

newest.prev = p;

size++;

}

public int removeFirst()

{

if(isEmpty())

{

Console.WriteLine("Empty");

return -1;

}

else

{

int e = head.element;

head = head.next;

head.prev = null;

size--;

return e;

if (isEmpty())

{

tail = null;

}

else

{

head.prev = null;

}

return e;

}

}

public int removeLast()

{

if (isEmpty())

{

Console.WriteLine("Empty");

return -1;

}

int e = tail.element;

tail = tail.prev;

tail.next = tail;

size--;

return e;

}

public int removeany(int position)

{

if (position <= 0||position>=size-1)

{

Console.WriteLine("Invalid");

return -1;

}

p = head;

int i = 1;

while(i<position-1)

{

p = p.next;

i++;

}

int e = p.next.element;

p.next = p.next.next;

p.next.prev = position;

size--;

return e;

}

1. **Stack**

Các hàm sử dụng trong Stack:

* Đối với Stack chỉ thao tác được trên top, khi top được xử lý mới đến cái phần tử ở dưới nên mỗi phần tử chỉ có 2 thuộc tính Top và data (khác với Stack Linked List) nên Stack sẽ có tính chất LIFO.

*//Tạo thuộc tính cho stack có data và top (LIFO)*

class StacksArray

{

int[] data;

int top;

*//Phương thức khởi tạo*

public StacksArray(int n)

{

data = new int[n]; // create Array which has n elements

top = 0;

}

public int Length()

{

return top;

}

public bool IsEmpty()

{

return top == 0;

}

*//Kiểm tra xem stack có full hay chưa (top = chiều dài stack)*

public bool IsFull()

{

return top == data.Length;

}

*// Chèn 1 giá trị vào stack, gán phần tử trên top = e, sau đó top thêm 1*

public void Push(int e)

{

if(IsFull())

{

Console.WriteLine("Stack is Full/Overflow");

}

else

{

data[top] = e;

top++;

}

}

*// Phương thức lấy ra phần tử trên top, cho e bằng giá trị top, và xóa top đó.*

public int Pop()

{

int e;

if (IsEmpty())

{

Console.WriteLine("Stack is Empty");

e = -1;

}

else

{

e = data[top - 1];

top--;

}

return e;

}

*//Lấy giá trị ở top mà không làm mất top*

public int Peek()

{

int e;

if (IsEmpty())

{

Console.WriteLine("Stack is empty");

e = -1;

}

else

{

e = data[top-1];

}

return e;

}

public void Display()

{

for (int i=0; i < top; i++)

{

Console.Write(data[i] + " -- ");

}

Console.WriteLine();

}

1. **Queue**

Khác với Stack, Queue có dạng FIFO, do đó ta cần có các thuộc tính: trước (front), sau (rear) và data

Code:

class QueuesArray

{

int[] data;

int front;

int rear;

int size;

public QueuesArray(int n)

{

data = new int[n];

front = 0;

rear = 0;

size = 0;

}

public int len()

{

return size;

}

public bool IsEmpty()

{

return size == 0;

}

public bool IsFull()

{

return size == data.Length;

}

*// Phương thức chèn phần tử vào Queue, cho giá trị e nằm ở rear của Queue, sau đó tăng giá trị rear và kích thước của Queue lên 1.*

public void EnQueue(int e)

{

if (IsFull())

{

Console.WriteLine("Queue is overflow");

}

else

{

data[rear] = e;

rear++;

size++;

}

}

*// Lấy giá trị của Queue, ở đây ta lấy đằng trước nên ta gán element là giá trị của front, sau đó front cộng thêm 1 vì front đầu đã lấy ra, kích thước Queue giảm.*

public int DeQueue()

{

int e;

if (IsEmpty())

{

Console.WriteLine("Queue is empty");

e = -1;

}

else

{

e = data[front];

front++;

size--;

}

return e;

}

*// Hàm này tương tự Dequeue nhưng k lấy giá trị ra*

public int First()

{

int e;

if (IsEmpty())

{

Console.WriteLine("Queue is empty");

e = -1;

}

else

{

e = data[front];

}

return e;

}

public void Display()

{

for(int i=0; i<rear; i++)

Console.Write(data[i] + " -- ");

Console.WriteLine(); }

**4. Trees**

Concept:

-Trees là dạng cấu trúc dữ liệu không tuyến tính (Ngược với stack và queue).

-Là 1 bộ của Node

- Có quan hệ Cha-Con qua các cạnh

-Trees có thể trống

Root:

-Là mức cao nhất trong Trees

Siblings: Là Node con của Node trước

External Node/ Leaf Node: là các node đứng 1 mình, không rẽ nhánh tiếp.

Internal Node/ Non-Leaf Node : ngược lại External Node/ Leaf Node.

Edge: cạnh

Path: Đường dẫn

Subtree: cây phụ

Forest: là khi gỡ Root và chỉ còn tập hợp các cây phụ

Height và Level:

-Tính từ Root: Height 0, Level 1.

-Mỗi hàng sẽ tăng thêm 1 bậc.

Cây nhị phân: Mỗi Node tối đa 2 nhánh, mỗi node sẽ có node trái và node phải

Degree: số lượng con của node đó

Số lượng Node trong cây nhị phân 2^(h+1) - 1 với h là chiều cao đếm từ 0

Proper Binary Tree: Mỗi node chỉ có 0 hoặc là 2 con

Full Binary Tree: mỗi node có chính xác 2 con

Complete Binary Tree: Mỗi mức đều được đánh số từ trái sang phải mà nó không gặp gap (lỗ hỏng, vd trái->phải->"phải"->trái...) nào khác (vd: trái-> phải->trái->phải->trái->...)

-----

-Trong cây nhị phân, mỗi node xem như 1 key.

-Keys ở bên trái có trọng số nhỏ hơn bên còn lại và ngược lại.

-Keys bên trái và cây bên phải đều gọi là Binary Search Trees.

-Trong Binary Search Treesthif không có phần tử hoặc node trùng nhau.

-Khi duyệt sẽ có list của các phần tử được sắp xếp.

-Cây tìm kiếm nhị phần có thể biểu diễn dạng mảng hoặc là danh sách liên kết.

----

function:

function search(key)

troot = root

while troot then

if key == troot.element then

return True

else if key < troot.element then

troot = troot.left

else if key > troot.element then

troot = troot.right

return false

Sử dụng đệ quy:

function rsearch(troot, key)

if root then

if key == troot.element then

return true

else if key < troot.element then

return rsearch(troot.left, key)

else if key > troot.element then

return rsearch(troot.right, key)

else

return false

BST - Iterative Insert

function insert(troot, e)

temp = null

while troot then

temp = troot

if e == troot.element then

return

else if e < troot.element then

troot = troot.left

else if e > troot.element then

troot = troot.right

n = Node(e)

if root

if e < temp.element then

temp.left = n

else

temp.right = n

else

root = n

function rinsert(troot, e)

if root then

if e < troot.element then

troot.left = rinsert(troot.left, e)

else if e > troot.element then

troot.right = rinsert(troot.right, e)

else

n = Node(e)

troot = n

return troot

Các cách duyệt BinSearchTrees

- Inorder

- Preorder

- Postorder

- Level

Túm lại, Trees có 1 root, khi bỏ root, sẽ tạo ra nhiều trees gọi là forest, với mỗi tree có nhánh trái và nhánh phải, và nhánh level thấp hơn sẽ là con, và level cao hơn sẽ là cha, và level cao nhất sẽ là root, chiều từ root đếm xuống sẽ đếm height = 0 rồi đếm lên, và mỗi node sẽ có nhánh trái và nhánh phải, nhánh trái sẽ nhỏ hơn node cha và nhánh phải sẽ lớn hơn node cha, và trees không bị lủng lỗ (node trái->phải->trái->phải …) là Conplete Binary Tree; Mọi Node có chính xác 0 hoặc 2 con thì là Proper Binary Trees, còn Full Binary Trees có chính xác 2 con, tóm lại là nhớ thế thôi.

Code:

*//Tạo thuộc tính Node và phương thức khởi tạo Node*

public class Node

{

public int element;

public Node left;

public Node right;

public Node(int e, Node l, Node r)

{

element = e;

left = l;

right = r;

}

}

*//Chèn vào Trees, nếu là root thì thoát khỏi phương thức, còn nếu giá trị nhỏ hơn root thì chèn bên trái, lớn hơn thì bên phải.*

public void insert(Node temproot, int e)

{

Node temp = null;

while(temproot != null)

{

temp = temproot;

if (e == temproot.element)

return;

else if (e < temproot.element)

temproot = temproot.left;

else if (e > temproot.element)

temproot = temproot.right;

}

Node n = new Node(e, null, null);

if (root != null)

{

if (e < temp.element)

temp.left = n;

else

temp.right = n;

}

else

root = n;

}

//Tương tự nhưng theo đệ qui

public Node recursiveInsert (Node temproot, int e)

{

if(temproot != null)

{

if (e < temproot.element)

temproot.left = recursiveInsert(temproot.left, e);

else if (e > temproot.element)

temproot.right = recursiveInsert(temproot.right, e);

}

else

{

Node n = new Node(e, null, null);

temproot = n;

}

return temproot;

}

*//Duyệt Trees theo phương thức Inorder, nếu trees cò giá trị để duyệt, duyệt từ nhánh trái rồi đến root con rồi đến nhánh phải đến khi đến root lớn và qua nhánh phải và tiếp tục cho đến khi hết Trees*

public void inorder(Node temproot)

{

if (temproot != null)

{

inorder(temproot.left);

Console.Write(temproot.element + " ");

inorder(temproot.right);

}

}

*//Duyệt cây trừ root đến hết nhánh bên phải rồi duyệt sang nhánh bên trái (không duyệt root gốc lần nữa)*

public void preorder(Node temproot)

{

if (temproot != null)

{

Console.Write(temproot.element + " ");

preorder(temproot.left);

preorder(temproot.right);

}

}

*//Duyệt từ dưới lên, trái rồi phải sang gốc cha, hết nhánh thì sang nhánh tiếp theo, hết nhánh rồi mới đến gốc.*

public void postorder(Node temproot)

{

if (temproot!= null)

{

postorder(temproot.left);

postorder(temproot.right);

Console.Write(temproot.element + " ");

}

}

*//Duyệt theo cấp bậc, từ root xuống hàng tiếp theo, hết hàng rồi đến hàng tiếp theo cho đến khi hết cây. Ở đây cần sử dụng Queue để ốc thể duyệt theo cách này, nói chung duyệt đến khi hết hàng thì sẽ đến hàng tiếp theo.*

public void levelorder(Node temproot)

{

if (temproot!= null)

{

Queue<Node> queue = new Queue<Node>();

queue.Enqueue(temproot);

while(queue.Count > 0)

{

Node n = queue.Dequeue();

Console.Write(n.element + " ");

if (n.left != null)

queue.Enqueue(n.left);

if (n.right != null)

queue.Enqueue(n.right);

}

}

}