

Trường đại học sư phạm Hà Nội

Khoa công nghệ thông tin

----&&&----

BÁO CÁO NGHIÊN CỨU KHOA HỌC

Đề tài: CẤU TRÚC DỮ LIỆU STACK VÀ ỨNG DỤNG CỦA STACK
TRONG CÁC GIẢI THUẬT ĐỆ QUI.

Giảng viên hướng dẫn: Thầy Nguyễn Hữu Dung

Sinh viên thực hiện: Nguyễn Thị Kim Oanh

Lớp: ak54

Hà Nội, ngày 15 tháng 4 năm 2008

Cấu trúc dữ liệu Stack và ứng dụng của stack trong các giải thuật đệ qui.

PHẦN 1: MỞ ĐẦU

I. LÍ DO CHỌN ĐỀ TÀI

Các kiểu cấu trúc dữ liệu cơ bản như stack, queue... cùng với các giải thuật đệ qui chiếm một vị trí rất quan trọng trong khoa học máy tính. Ngày nay, với sự phát triển như vũ bão của công nghệ thông tin, các thuật toán mới ra đời để giúp con người giải các bài toán mới, phức tạp. Nhưng vai trò của kiểu cấu trúc dữ liệu stack không hề bị giảm bớt, nó chính là kiểu dữ liệu cơ bản để áp dụng vào giải các bài toán phức tạp. Cũng như stack, đệ qui cũng có tuổi thọ khá cao trong lĩnh vực khoa học máy tính nhưng vị trí, vai trò của nó vẫn rất quan trọng. Nhờ có đệ qui mà một số bài toán phức tạp được giải quyết một cách dễ dàng.

Chính vì vậy mà trong chương trình học môn cấu trúc dữ liệu và giải thuật của các trường cao đẳng, đại học hay trường chuyên, kiểu cấu trúc dữ liệu stack và đệ qui chiếm một vị trí quan trọng, việc học chúng có ý nghĩa làm nền tảng cho việc học các thuật toán khác cũng như viết code để cài đặt một chương trình máy tính nào đó.

Và để cho học sinh, sinh viên có thể tiếp thu những kiến thức đó một cách hiệu quả, tránh rơi vào tình trạng mơ hồ, trừu tượng (hiện tượng hay thường gặp khi học sinh, sinh viên lần đầu tiếp thu kiến thức) thì hướng phát triển lên của đề tài là mô phỏng việc hoạt động của stack, ứng dụng của stack trong hoạt động của các giải thuật đệ qui.

Tuy rằng việc nghiên cứu học tập về stack và đệ qui là một đề tài không còn mới mẻ, thậm chí có nhiều cá nhân cho rằng đã lỗi thời. Nhưng stack và đệ

qui là những mảng kiến thức không thể thiếu trong khoa học máy tính. Chính vì vậy, việc học tập và nghiên cứu chúng luôn cần thiết và mô phỏng hoạt động của stack và đệ qui làm cho công việc đó trở nên hiệu quả và giảm chi phí thời gian cho người học và người dạy.

II. MỤC TIÊU NHIỆM VỤ

- Nghiên cứu để làm rõ tác dụng vai trò của stack trong việc hoạt động của một số giải thuật đệ qui.
- Hướng phát triển là tìm hiểu lí thuyết để mô phỏng hoạt động của stack và ứng dụng của stack trong các giải thuật đệ qui.

III. ĐỐI TƯỢNG NGHIÊN CỨU

- Lí thuyết về cấu trúc dữ liệu trừu tượng Stack
- Hoạt động của Stack và việc áp dụng stack trong một số bài toán cơ bản.
- Đệ qui và một số giải thuật đệ qui.
- Việc ứng dụng stack vào trong các hoạt động của một số giải thuật đệ qui.
- Ngôn ngữ lập trình hướng đối tượng Visual Foxpro dùng để phục vụ cho hướng phát triển là cài đặt mô phỏng.

IV. PHƯƠNG PHÁP NGHIÊN CỨU

Nghiên cứu, học tập chủ yếu thông qua giáo trình môn cấu trúc dữ liệu và giải thuật, tài liệu, bài giảng của giảng viên, sách tham khảo, tài liệu download từ trên mạng.

V. CẤU TRÚC KHOÁ LUẬN

Khoa luận gồm 2 phần:

Phần 1- Mở đầu: là phần nêu lí do chọn đề tài, mục đích nghiên cứu đề tài, đối tượng nghiên cứu và phương pháp nghiên cứu đề tài.

Phần 2- Nội dung: là phần trọng tâm của đề tài, trong phần này gồm có:

- Lí thuyết về cấu trúc dữ liệu stack
- Lí thuyết về đệ qui
- Ứng dụng của stack vào hoạt động của các giải thuật đệ qui.

PHẦN 2: NỘI DUNG

A. LÍ THUYẾT

I. LÍ THUYẾT VỀ CẤU TRÚC DỮ LIỆU STACK

1. ĐỊNH NGHĨA NGĂN XẾP:

Stack là một kiểu danh sách tuyến tính đặc biệt mà phép bổ sung và phép loại bỏ luôn luôn thực hiện ở một đầu gọi là đỉnh.

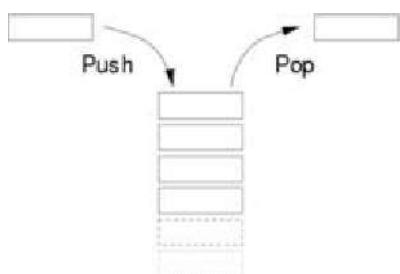
Hay ta còn có thể định nghĩa khác là: ngăn xếp (stack) là một cấu trúc dữ liệu trừu tượng làm việc theo nguyên lý vào sau ra trước (last in first out).

Một ngăn xếp là một cấu trúc dữ liệu dạng thùng chứa (container) của các phần tử (thường gọi là các nút (node)) và có hai phép toán cơ bản : *push* and *pop*.

- *Push* bổ sung một phần tử vào đỉnh (top) của ngăn xếp,nghĩa là sau các phần tử đã có trong ngăn xếp.

- *Pop* giải phóng và trả về phần tử đang đứng ở đỉnh của ngăn xếp.

Trong stack, các đối tượng có thể được thêm vào stack bất kỳ lúc nào nhưng chỉ có đối tượng thêm vào sau cùng mới được phép lấy ra khỏi stack.



Ngoài ra, stack cũng hỗ trợ một số thao tác khác:

- *isEmpty()*: Kiểm tra xem stack có rỗng không.

- Top(): Trả về giá trị của phần tử nằm ở đầu stack mà không hủy nó khỏi stack. Nếu stack rỗng thì lỗi sẽ xảy ra.

2. MÔ TẢ STACK

2.1 Mô tả Stack bằng mảng

Khi mô tả Stack bằng mảng:

- Việc bổ sung một phần tử vào Stack tương đương với việc thêm một phần tử vào cuối mảng.
- Việc loại bỏ một phần tử khỏi Stack tương đương với việc loại bỏ một phần tử ở cuối mảng.
- Stack bị tràn khi bổ sung vào mảng đã đầy.
- Stack là rỗng khi số phần tử thực sự đang chứa trong mảng = 0

```
Program stack_by_array;
const max = 10000;
var
  stack: array[1..max] of integer;
  last: integer;
procedure stack_init;
begin
  last:= 0;
end;

procedure push(v: integer);
begin
  if last = max then writeln('stack is full')
  else
    begin
      inc(last); stack[last]:= v;
    end;
end;
```

```

    end;
end;

function pop: integer;
begin
  if last = 0 then writeln('stack is empty')
  else
    begin
      pop:= stack[last]; Dec(last);
    end;
end;

```

BEGIN

```

  Stack_init;
  <test>;

```

END.

2.2 Mô tả bằng danh sách nối đơn kiểu LIFO

Khi cài đặt Stack bằng danh sách nối đơn kiểu LIFO, thì stack bị tràn khi vùng không gian nhớ dùng cho các biến động không còn đủ để thêm một phần tử mới. Tuy nhiên, việc kiểm tra điều này rất khó bởi nó phụ thuộc vào máy tính và ngôn ngữ lập trình. Ví dụ như đối với Turbo Pascal, khi Heap còn trống 80 bytes thì cũng chỉ đủ chỗ cho 10 biến, mỗi biến 6 bytes mà thôi. Mặt khác, không gian bộ nhớ dùng cho các biến động thường rất lớn nên cài đặt dưới đây ta bỏ qua việc kiểm tra stack tràn.

Program stack_by_linklist;

Type

```

Pnode = ^Tnode;
```

```

Tnode = record
    Value: integer;
    Link: Pnode;
end;
var
  last: Pnode;

Procedure stack_init;
Begin
    Last:= nil;
End;

Procedure push( v: integer);
Var p: Pnode;
Begin
    New(p);
    p^.link:= last; last:= p;
end;
function pop: integer;
var p: Pnode;
begin
    if last = nil then writeln('stack is empty')
    else
        begin
            pop:= last^.value;
            p:= last^.link;
            dispose(last); last:= p;
        end;
end;

```

BEGIN

Stack_init;

<test>;

END.

II. LÝ THUYẾT VỀ ĐỆ QUI

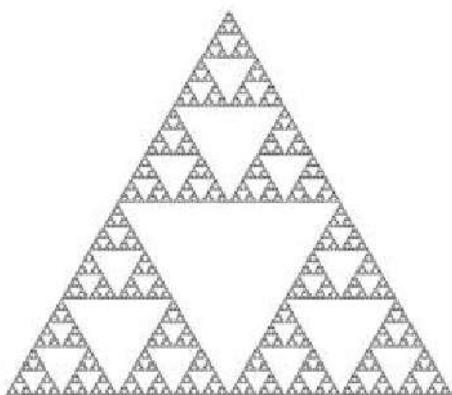
1. KHÁI NIỆM VỀ ĐỆ QUI

Ta nói một đối tượng là đệ qui nếu nó được định nghĩa qua chính nó hoặc một đối tượng khác cùng dạng với chính nó bằng quy nạp.

Ví dụ: Qua 2 chiếc gương cầu đối diện nhau. Trong chiếc gương thứ 1 chứa hình chiếc gương thứ 2. Chiếc gương thứ 2 lại chứa hình chiếc gương thứ 1 nên tất nhiên nó chứa lại hình ảnh của chính nó trong chiếc gương thứ 1. Ở một góc nhìn hợp lí, ta có thể thấy một dãy ảnh vô hạn của cả 2 chiếc gương. Một ví dụ khác là nếu người ta phát hình trực tiếp phát thanh viên ngoài bên máy vô tuyến truyền hình, trên màn hình của máy này lại có chính hình ảnh của phát thanh viên đó ngoài bên máy vô tuyến truyền hình và cứ như thế...

Trong toán học, ta cũng hay gặp các định nghĩa đệ qui:

Giai thừa của n ($n!$): Nếu $n=0$ thì $n! = 1$; nếu $n > 0$ thì $n! = n.(n-1)!$



Tam giác Sierpinski

2. GIẢI THUẬT ĐỆ QUI

Nếu lời giải của một bài toán P được thực hiện bằng lời giải của bài toán P' có dạng giống như P thì đó là một lời giải đệ qui. Giải thuật tương ứng với lời giải như vậy gọi là giải thuật đệ qui. Mới nghe thì có vẻ hơi lạ nhưng điểm mấu chốt cần lưu ý là: P' tuy có dạng giống như P, nhưng theo một nghĩa nào đó, nó phải “nhỏ hơn” P, dễ giải hơn P và việc giải nó không cần dùng đến P.

Trong Pascal, ta đã thấy nhiều ví dụ của các hàm và thủ tục có chứa lời gọi đệ qui tới chính nó, bây giờ, ta tóm tắt lại các phép đệ qui trực tiếp và tương hỗ được viết như thế nào.

Định nghĩa một hàm đệ qui hay thủ tục đệ qui gồm 2 phần:

- Phần neo (anchor): phần này được thực hiện khi mà công việc quá đơn giản, có thể giải trực tiếp chứ không cần phải nhờ đến một bài toán con nào cả.
- Phần đệ qui: Trong trường hợp bài toán chưa thể giải được bằng phần neo, ta xác định những bài toán con và gọi đệ qui giả những bài toán con đó. Khi đã có lời giải của những bài toán con rồi thì phối hợp chúng lại để giải bài toán đang quan tâm.

Phần đệ qui thể hiện tính qui nạp của lời giải. Phần neo cũng rất quan trọng bởi nó quyết định tới tính hữu hạn dùng của lời giải.

3. VÍ DỤ VỀ GIẢI THUẬT ĐỆ QUI

3.1 Hàm tính giai thừa

Function Factorial (n: integer): integer;

Begin

if n= 0 then Factorial:= 1

*else Factorial:= n * Factorial(n-1);*

End;

Ở đây, phần neo định nghĩa kết quả hàm tại $n=0$, còn phần đệ qui (ứng với $n>0$) sẽ định nghĩa kết quả hàm qua giá trị của n và giai thừa của $n-1$.

Ví dụ: dùng hàm này để tính $3!$, trước hết nó phải tính $2!$ bởi $3!$ được tính bằng tích của $3*2!$. Tương tự, để tính $2!$, nó lại phải tính $1!$ bởi $2!$ được tính bằng $2*1!$. Áp dụng bước qui nạp này thêm một lần nữa $1!=1*0!$, và ta đạt được tới trường hợp của phần neo, đến đây từ giá trị 1 của $0!$, nó tính được $1!=1*1$; từ giá trị của $1!$ tính được $2!$; sau đó tính được $3!$; cuối cùng cho kết quả là 6

$$3!=3*2! \rightarrow 3*2*1! \rightarrow 3*2*1*0! 3*2*1*1 = 6$$

3.2 Dãy số Fibonacci

Dãy số Fibonacci bắt nguồn từ bài toán cổ về việc sinh sản của các cặp thỏ. Bài toán đặt ra như sau:

1. Các con thỏ không bao giờ chết.
2. Hai tháng sau khi ra đời, mỗi cặp thỏ mới sẽ sinh ra một cặp thỏ con (một đực, một cái).
3. Khi đã sinh con rồi thì cứ mỗi tháng tiếp theo chúng lại sinh được một cặp thỏ mới.

Gia sử từ đầu tháng 1 có một cặp mới ra đời thì đến giữa tháng thứ n sẽ có bao nhiêu cặp?

Ví dụ, $n=5$ ta thấy:

Giữa tháng 1: 1 cặp ab ban đầu

Giữa tháng 2: 1 cặp ab ban đầu chưa đẻ

Giữa tháng 3: 2 cặp ab ban đầu và cặp cd

Giữa tháng 4: 3 cặp ab, cd, ef, cặp ab ban đầu tiếp tục đẻ.

Giữa tháng 5: 5 cặp ab, cd, ef, gh, ik; cặp ab ban đầu đẻ thêm và cặp cd bắt đầu đẻ.

Bây giờ, ta xét tới việc tính số cặp thỏ ở tháng thứ n: $F(n)$

Nếu mỗi cặp thỏ ở tháng thứ $n-1$ đều sinh ra một cặp thỏ con thì số cặp thỏ ở tháng thứ n sẽ là:

$$F(n) = 2 * F(n-1)$$

Nhưng vấn đề không phải như vậy, trong các cặp thỏ ở tháng thứ $n-1$, chỉ có những cặp thỏ đã ở tháng thứ $n-2$ mới sinh ra con ở tháng thứ n được thôi. Do đó $F(n) = F(n-1) + F(n-2)$ (= số cũ + số sinh ra). Vậy có thể tính được $F(n)$ theo công thức sau:

- $F(n) = 1$ nếu $n \leq 2$
- $F(n) = F(n-1) + F(n-2)$ nếu $n > 2$

function $F(n: integer): integer;$

begin

```
if  $n \leq 2$  then  $F := 1$ 
else  $F := F(n-1) + F(n-2);$ 
```

end;

3.3 Giả thuyết của Collatz.

Collatz đưa ra giả thuyết rằng: với một số nguyên dương X , nếu X chẵn thì ta gán $X := X \text{ div } 2$; nếu X lẻ thì ta gán $X := X*3+1$. Thì sau một số hữu hạn bước, ta sẽ có $X = 1$.

Ví dụ $X=10$, các bước tiến hành như sau:

1. $x = 10$ chẵn $\rightarrow x := 10 \text{ div } 2$ ($x := 5$)
2. $x = 5$ lẻ $\rightarrow x := 5*3+1$; ($x := 16$)
3. $x = 16$ chẵn $\rightarrow x := 16 \text{ div } 2$; ($x := 8$)
4. $x = 8$ chẵn $\rightarrow x := 8 \text{ div } 2$; ($x := 4$)
5. $x = 4$ chẵn $\rightarrow x := 4 \text{ div } 2$; ($x := 2$)
6. $x = 2$ chẵn $\rightarrow x := 2 \text{ div } 2$; ($x := 1$)

Cứ cho là giả thuyết Collatz là đúng đắn, vẫn đề đặt ra là: cho trước số 1 cùng với hai phép toán $*2$ và $\text{div } 3$, hãy sử dụng một cách hợp lí hai phép toán đó để biến số 1 thành giá trị nguyên dương X cho trước.

Ví dụ: $X=10$ ta có $1*2*2*2*2 \text{ div } 3 = 10$

Dễ thấy rằng lời giải của bài toán gần như thứ tự ngược của phép biến đổi Collatz: để biểu diễn số $X > 1$ bằng một biểu thức bắt đầu bằng số 1 và hai phép toán “ $*2$ ”, “ $\text{div } 3$ ”. Ta chia hai trường hợp:

- Nếu X chẵn, thì ta tìm cách biểu diễn số $X \text{ div } 2$ và viết thêm phép toán $*2$ vào cuối.
- Nếu X lẻ, thì ta tìm cách biểu diễn số $X*3+1$ và viết thêm phép toán $\text{div } 3$ vào cuối.

Procedure solve(x: integer);

begin

if x = 1 then write(x)

else

if x mod 2 = 0 then

begin

solve(x div 2);

*write(' *2');*

end

else

begin

*solve(x*3 +1);*

write(' div 3');

end;

end;

Trên đây là cách viết đệ qui trực tiếp, còn có một cách viết đệ qui tương hỗ như sau:

Procedure solve(x: integer); forward;

Procedure solveodd(x: integer);

Begin

*Solve(x*3+1);*

*Write (' *2');*

End;

Procedure solveeven(x: integer);

Begin

Solve(x div 2);

*Write(' *2');*

End;

Procedure solve(x: integer);

Begin

If x =1 then write(x)

Else

If x mod 2 = 1 then solveodd(x)

Else solveeven(x);

End;

Trong cả hai cách viết, để tìm biểu diễn số x theo yêu cầu chỉ cần gọi solve(x) là xong. Tuy nhiên trong cách viết đệ qui trực tiếp, thủ tục solve có lời gọi tới chính nó, còn trong cách viết đệ qui tương hỗ, thủ tục solve chưa lời gọi tới thủ tục solveodd và solveeven, hai thủ tục này lại chứa trong nó lời gọi ngược về thủ tục solve.

Đối với những bài toán nêu trên, việc thiết kế các giải thuật đệ qui tương ứng với khá thuận lợi vì cả hai đều thuộc dạng tính giá trị hàm mà định nghĩa qui nạp của hàm đó được xác định dễ dàng.

Nhưng không phải lúc nào phép đếm qui cũng có thể nhìn nhận và thiết kế dễ dàng như vậy. Thế thì vấn đề gì cần lưu tâm trong phép giải đếm qui? Có thể nhìn thấy câu trả lời qua việc giải đáp các câu hỏi:

1. Có thể định nghĩa được bài toán dưới dạng phối hợp của những bài toán cùng loại nhưng nhỏ hơn hay không? Khái niệm “nhỏ hơn” là thế nào?
2. Trường hợp đặc biệt nào của bài toán sẽ được coi là trường hợp tầm thường và có thể giải ngay được để đưa vào phần neo của phép giải đếm qui.

3.4 Bài toán tháp Hà Nội.

Đây là một bài toán mang tính chất một trò chơi, nội dung như sau: có n đĩa đường kính hoàn toàn phân biệt, đặt chồng lên nhau, các đĩa được xếp theo thứ tự giảm dần của đường kính hoàn toàn phân biệt, đặt chồng lên nhau, các đĩa được xếp theo thứ tự giảm dần của đường kính tính từ dưới lên, đĩa to nhất được đặt sát đất. Có ba vị trí có thể đặt các đĩa đánh số 1, 2, 3. Chồng đĩa ban đầu được đặt ở vị trí 1:



Người ta muốn chuyển cả chồng đĩa từ vị trí 1 sang vị trí 2, theo những điều kiện:

- Khi di chuyển một đĩa, phải đặt nó vào một trong 3 vị trí đã cho
- Mỗi lần chỉ có thể chuyển một đĩa và phải là đĩa ở trên cùng
- Tại một vị trí, đĩa nào mới chuyển đến sẽ phải đặt lên trên cùng
- Đĩa lớn hơn không bao giờ được phép đặt lên trên đĩa nhỏ hơn (hay nói cách khác: một đĩa chỉ được đặt trên mặt đất hoặc trên một đĩa lớn hơn)

Trong trường hợp có 2 đĩa, cách làm có thể mô tả như sau:

Chuyển đĩa nhỏ hơn sang vị trí 3, đĩa lớn hơn sang vị trí 2 rồi chuyển đĩa nhỏ từ vị trí 3 sang vị trí 2.

Những người mới bắt đầu có thể giải quyết bài toán một cách dễ dàng khi số đĩa là ít, nhưng họ sẽ gặp rất nhiều khó khăn khi số các đĩa nhiều hơn. Tuy nhiên, với tư duy qui nạp toán học và một máy tính thì công việc trở nên khá dễ dàng:

Có n đĩa

- Nếu $n=1$ thì ta chuyển đĩa duy nhất đó từ vị trí 1 sang vị trí 2 là xong.
- Giả sử rằng ta có phương pháp chuyển được $n-1$ đĩa từ vị trí 1 sang vị trí 2, thì cách chuyển $n-1$ đĩa từ vị trí x sang vị trí y ($1 \leq x, y \geq 3$) cũng tương tự
- Giả sử rằng ta có phương pháp chuyển được $n-1$ đĩa giữa hai vị trí bất kỳ. Để chuyển n đĩa từ vị trí x sang vị trí y , ta gọi vị trí còn lại là z ($= 6 - x - y$). Coi đĩa to nhất là mặt đất, chuyển $n-1$ đĩa còn lại từ vị trí x sang vị trí z , sau đó chuyển đĩa to nhất sang vị trí y và cuối cùng lại coi đĩa to nhất đó là mặt đất, chuyển $n-1$ đĩa còn lại đang ở vị trí z sang vị trí y chồng lên đĩa to nhất đó.

Cách làm đó được thể hiện trong thủ tục đệ qui dưới đây:

```

Procedure move(n, x, y: integer);
Begin
  If n= 1 then writeln('chuyển 1 đĩa từ', x, 'sang',y)
  Else
    Begin
      Move(n- 1, x, 6-x-y);
      Move(1, x, y);
      Move(n-1, 6- x – y, y);
    End;
End;

```

Chương trình chính rất đơn giản, chỉ gồm có 2 việc: nhập vào số n và gọi Move(n, 1, 2)

4. HIỆU LỰC CỦA ĐỀ QUI

Qua các ví dụ trên, ta có thể thấy đề qui là một công cụ mạnh để giải quyết các bài toán. Có những bài toán mà bên cạnh giải thuật đề qui vẫn có những giải thuật lặp khá đơn giản và hữu hiệu. Chẳng hạn bài toán tính giai thừa hay tính số Fibonaci. Tuy vậy, đề qui vẫn có vai trò xứng đáng của nó, có nhiều bài toán mà việc thiết kế giải thuật đề qui đơn giản hơn nhiều so với lời giải lặp và trong một số trường hợp chương trình đề qui hoạt động nhanh hơn chương trình viết không có đề qui. Giải thuật cho bài toán Tháp Hà Nội và thuật toán sắp xếp kiểu phân đoạn (Quick Sort) mà ta sẽ nói tới trong các bài sau là ví dụ.

Có một mối quan hệ khăng khít giữa đề qui và qui nạp toán học. Cách giải đề qui cho một bài toán dựa trên việc định rõ lời giải cho trường hợp suy biến (neo) rồi thiết kế làm sao để lời giải của bài toán được suy ra từ lời giải của bài toán nhỏ hơn cùng loại như thế. Tương tự như vậy, qui nạp toán học chứng minh một tính chất nào đó ứng với số tự nhiên cũng bằng cách chứng

minh tính chất đó đúng với một số trường hợp cơ sở (thường người ta chứng minh nó đúng với 0 hay đúng với 1) và sau đó chứng minh tính chất đó sẽ đúng với n bất kỳ nếu nó đã đúng với mọi số tự nhiên nhỏ hơn n. Do đó ta không mấy làm ngạc nhiên khi thấy qui nạp toán học được dùng để chứng minh các tính chất có liên quan tới giải thuật đệ qui. Chẳng hạn: chứng minh số phép chuyển đĩa để giải bài toán tháp Hà Nội với n đĩa là $2^n - 1$:

- Rõ ràng là tính chất này đúng với n=1, bởi ta cần $2^1 - 1 = 1$ chuyển đĩa để thực hiện yêu cầu.
- Với $n > 1$; giả sử rằng để chuyển $n-1$ đĩa giữa hai vị trí ta cần $2^{n-1} - 1$ phép chuyển đĩa, khi đó để chuyển n đĩa từ vị trí x sang vị trí y, nhìn vào giải thuật qui ta có thể thấy rằng trong trường hợp này nó cần $(2^{n-1} - 1) + 1 + (2^{n-1} - 1) = 2^n - 1$ phép chuyển đĩa. Tính chất được chứng minh đúng với n.

Vậy thì công thức này sẽ đúng với mọi n.

Thật đáng tiếc nếu như chúng ta phải lập trình với một công cụ không cho phép đệ qui, nhưng như vậy không có nghĩa là ta bó tay trước một bài toán mang tính đệ qui. Mọi giải thuật đệ qui đều có cách thay thế bằng một giải thuật không đệ qui (khử đệ qui), có thể nói được như vậy bởi tất cả các chương trình con đệ qui sẽ đều được trình dịch chuyển thành những mã lệnh không đệ qui trước khi giao cho máy tính thực hiện.

Việc tìm hiểu cách khử đệ qui một cách “máy móc” như các chương trình dịch thì chỉ cần hiểu rõ cơ chế xếp chồng của các thủ tục trong một dây chuyền gọi đệ qui là có thể làm được. Nhưng muốn khử đệ qui một cách tinh tế phải tuỳ thuộc vào từng bài toán mà khử đệ qui cho khéo. Không phải tìm đâu xa, những kỹ thuật giải công thức truy hồi bằng qui hoạch động là ví dụ cho thấy tính nghệ thuật trong những cách tiếp cận bài toán mang bản chất đệ qui để tìm ra một giải thuật không đệ qui đầy hiệu quả.

III. STACK VÀ VIỆC CÀI ĐẶT THỦ TỤC ĐỆ QUI

Khi một thủ tục đệ qui được gọi tới từ chương trình chính, ta nói: thủ tục được thực hiện ở mức 1 hay độ sâu 1 của tính đệ qui. Nhưng khi thực hiện ở mức 1 lại gấp lại lời gọi chính nó, nghĩa là phải đi sâu vào mức 2 và cứ như thế cho tới một mức nào đó. Rõ ràng là mức k phải được hoàn thành xong thì mức (k-1) mới được thực hiện. Lúc đó ta nói: việc thực hiện được quay về mức (k-1).

Khi từ một mức I, đi sâu vào mức (i+1) thì có thể có một số tham số, biến cục bộ hay địa chỉ (gọi là địa chỉ quay lui) ứng với mức i cần phải được bảo lưu để khi quay về tiếp tục sử dụng.

Như vậy trong quá trình thực hiện, những tham số, biến cục bộ hay địa chỉ bảo lưu sau lại được khôi phục trước. Tính chất “vào sau ra trước” này dẫn tới việc sử dụng stack trong cài đặt thủ tục đệ qui. Mỗi khi có lời gọi tới chính nó thì stack sẽ được nạp để bảo lưu các giá trị cần thiết. Còn mỗi khi thoát ra khỏi một mức thì phần tử ở đỉnh stack sẽ được “móc” ra để khôi phục lại các giá trị cần thiết cho mức tiếp theo.

Ta có thể tóm tắt các bước này như sau:

1. MỞ ĐẦU

Bảo lưu tham số, biến cục và địa chỉ quay lui.

2. THÂN

Nếu tiêu chuẩn cơ sở (base criterion) ứng với trường hợp suy biến đã đạt được thì thực hiện được ở phần tính kết thúc (final computation) và chuyển sang bước 3.

3. KẾT THÚC

Khôi phục lại tham số, biến cục bộ và địa chỉ quay lui và chuyển tới địa chỉ quay lui này.

1. BÀI TOÁN TÍNH GIAI THỪA

Sau đây là chương trình thể hiện các cài đặt thủ tục đệ qui, có thể dùng stack cho bài toán tính n! và “tháp Hà Nội”.

Program factorial

{Cho số nguyên n, giải thuật này thực hiện tính n! Ở đây sử dụng một stack A mà đỉnh được trả bởi T. Mỗi phần tử của A là một bản ghi gồm có hai trường:

Trường N ghi giá trị động của n ở mức hiện hành.

Trường RETADD ghi địa chỉ quay lui.

Lúc đầu stack A rỗng: T = 0.

Một bản ghi TEMREC được dùng làm bản ghi trung chuyển, nó cũng có 2 trường:

PARA ứng với N

ADDRESS ứng với RETADD

Ở đây đặt giả thuyết: lúc đầu TEMREC đã chứa các giá trị cần thiết, nghĩa là PARA chứa giá trị n đã cho, ADDRESS chứa địa chỉ ứng với lời gọi trong chương trình chính mà ta gọi là ĐCC(viết tắt của địa chỉ chính).

Các giải thuật PUSH và POS sẽ được sử dụng ở đây. Trong giải thuật này ta viết N(T) thì điều đó có nghĩa là giá trị ở trường N của phần tử đang trả tới bởi T (phần tử ở đỉnh stack A)}

1. {Bảo lưu giá trị của N và địa chỉ quay lui}

call PUSH(A,T, TEMREC)

2. {Tiêu chuẩn cơ sở đã đạt chưa?}

if N(T)=0 then

begin

factorial:=1;

```

    goto bước 4;
end
else begin
    PARA := N(T) - 1
    ADDRESS:= bước 3
end;
goto bước 1
3. {tính N!}
factorial:= N(T)* factorial;
4. {Khôi phục giá trị trước của N và địa chỉ quay lui}
call POP(A, T, TEMREC);
goto ADDRESS
5 end.

```

Sau đây là hình ảnh minh họa tình trạng của stack A trong quá trình thực hiện giải thuật, ứng với $n = 3$.

Số mức	Các bước thực hiện
Vào mức 1(lời gọi chính)	Bước 1 PUSH(A,O,(3,ĐCC)) bước 2: $N <> 0$ PARA:= 2; ADDRESS:= bước 3
Vào mức 2 (gọi đệ qui lần 1)	bước 1: PUSH(A,1,(2,bước 3))

	<p>bước 2: N<>0 PARA:=1; ADDRESS:= bước 3</p>
Vào mức 3 (gọi đệ qui lần 2)	<p>bước 1: PUSH(A,2,(1, bước 3)) bước 2: N<>0; PARA:=0; ADDRESS:= bước 3;</p>
Vào mức 4 (gọi đệ qui lần 3)	<p>bước 1: PUSH(A,3,(0,bước 3)) bước 2: N= 0; Factorial:=1; bước 4: POP(A, 4, TEMREC); goto bước 3;</p>
Quay lại mức 3	<p>bước 3: factorial:=1*1 bước 4: POP(A, 3, TEMREC); Goto bước 3;</p>
Quay lại mức 2	<p>bước 3: factorial:= 2*1 POP(A, 2, TEMREC);</p>

	goto bước 3
Quay lại mức 1	bước 3: factorial:= 3*2 bước 4: POP(A,1, TEMREC); goto ĐCC

2.BÀI TOÁN THÁP HÀ NỘI

N: là số lượng đĩa.

SN: chỉ cọc xuất phát

IN: chỉ cọc trung chuyển

DN: chỉ cọc đích.

Giải thuật này thực hiện chuyển chồng N đĩa từ cọ SN sang cọ DN.

Ở đây sử dụng một stack ST mỗi phần tử của nó là một bản ghi gồm có 5 trường tương ứng với N, SN, IN, DN và RETADD để chứa các giá trị của N, SN, IN, DN và địa chỉ quay lui.

Một bản ghi TEMREC cũng có các trường tương ứng với các trường nêu trên, lần lượt gọi là NVAL, SNVAL, DNVAL và ADDRESS. TEMREC được dùng làm bản ghi trung chuyển. Thoạt đầu nó ghi nhận các giá trị ban đầu của N, SN, IN, DN và RETADD.

Stack ST có định được trả bởi T, lúc đầu stack rỗng thì T=0.

1 {Bảo lưu tham số và địa chỉ quay lui}

call PUSH(ST, T, TEMREC)

2. {Kiểm tra giá trị dừng của N, nếu chưa đạt thì chuyển N- 1 đĩa từ cọc xuất phát sang cọc trung chuyển}

If N(T)= 0 then goto RETADD(T)

else begin

```
NVAL:= N(T)- 1;  
SNVAL:=SN(T);  
INVAL:=DN(T);  
DNVAL:=IN(T);  
ADDRESS:= bước 3;  
goto bước 1;
```

3. {Chuyển đĩa thứ N từ cọc xuất phát sang cọc đích và N-1 đĩa từ cọc trung chuyển sang cọc đích}

```
call POS(ST,T, TEMREC);  
write('dia', N, 'tu coc', 'SN', 'den coc', 'DN')  
NVAL:= N(T)-1;  
SNVAL:= IN(T);  
INVAL:= SN(T);  
DNVAL:=DN(T);  
ADDRESS:= bước 4;  
goto bước 1;
```

4. {Quay lại mức trước}

```
call POP(ST,T, TEMREC);  
goto RETADD(T)
```

5. end

3. TÌM KIẾM QUICK SORT

3.1 Giới thiệu phương pháp

Sắp xếp kiểu phân đoạn (quick sort) là một cải tiến của phương pháp sắp xếp kiểu đổi chỗ. Đây là phương pháp khá tốt, do đó người sáng lập ra nó C.A.R.Hoare, đã mạnh dạn đặt tên cho nó là sắp xếp NHANH.

Procedure Quick-sort(K , LB , UB);

If $LB < UB$ then

Begin

Call PART(K , LB , UB , j);

Call Quick-sort(K , LB , $j-1$);

Call Quick-sort(K , $j+1$, UB);

End;

Return;

Procedure PART(K , LB , UB , j);

1. $i := LB + 1$; $j := UB$;

2. while $i \leq j$ do begin

3. while $K[i] < K[LB]$ do $i := i + 1$;

4. while $K[j] > K[LB]$ do $j := j - 1$;

5. if $i < j$ then begin

$K[i]$ đổi chỗ $K[j]$;

$i := i + 1$;

$j := j - 1$;

end;

end;

6. $K[LB]$ đổi chỗ $K[j]$

7. return

Và chúng ta cũng có thể cài đặt Quick-sort mà có sử dụng một stack như sau:

bước 1: khởi tạo một stack rỗng

bước 2: mảng a ta đang xét từ phần tử thứ 1 đến phần tử thứ n, gán L:=1, và R:= n. Đẩy 2 giá trị 1 và n vào stack.

bước 3: lấy lại L, R từ stack ra

bước 4: phân hoạch dãy $a[L]..a[R]$ thành 2 dãy $a[L]..a[j-1]$ và dãy $a[j+1]...a[R]$

bước 5: nếu $(j+1) < R$ thì đẩy $(j+1)$, R vào stack

bước 6: nếu $L < (j-1)$ quay lại bước 4 để phân hoạch dãy $a[L]...a[j-1]$ nếu không chuyển tới bước 7

bước 7: nếu stack khác rỗng thì quay lại bước 3 để phân hoạch tiếp, nếu stack rỗng thì kết thúc.

4.BÀI TOÁN DUYỆT HẬU THỨ TỰ

Bài toán này áp dụng để duyệt cây nhị phân.

Phép xử lí các nút trên cây mà ta gọi chung là phép thăm (visit) các nút một cách hệ thống sao cho mỗi nút được thăm một lần gọi là phép duyệt cây.

Giả sử rằng nếu như một nút không có nút con trái hay nút con phải thì liên kết left (right) của nút đó được liên kết thẳng tới một nút đặc biệt là NIL hay NULL, nếu cây rỗng thì nút gốc cũng được gán bằng NIL.

Trong phép duyệt hậu thứ tự thì giá trị mỗi nút bất kỳ sẽ được liệt kê sau giá trị lưu ở 2 nút con của nút đó, có thể mô tả bằng thủ tục đệ qui như sau:

Procedure visit(N)

Begin

If N ≠ nil then

Begin

visit(nút con trái của N);

visit(nút con phải của N);

<output trường info của nút N>;

end;

end;

Và chúng ta có thể cài đặt thủ tục duyệt hậu tự như trên có sử dụng stack như sau:

Procedure visit(N)

Begin

If $N \neq \text{nil}$;

Push N vào stack;

Begin

$\text{Visit}(\text{nút con trái của } N);$

Visit(nút con phải của N);

Pos N ra khỏi stack;

<output trường info của nút N>;

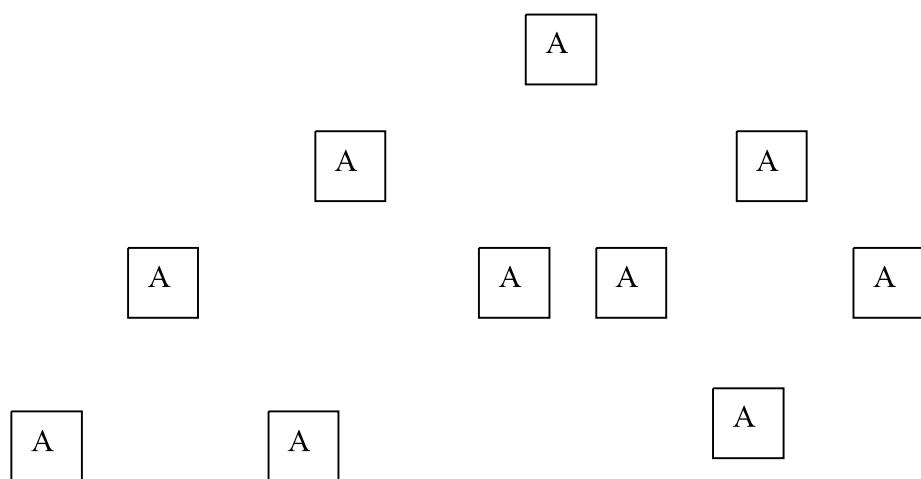
End;

End;

Xét một ví dụ như sau:

Ta có một cây nhị phân như sau:

Trường info của mỗi nút lưu tên của nút đó.



Hoạt động của thuật toán duyệt hậu tự và tình trạng của stack sẽ được biểu diễn như sau:

- Nút gốc A khác nil, đẩy A vào stack
- xuống nút B là con trái của A, đẩy B vào stack.
- xuống nút D là con trái của B, đẩy D vào stack.
- xuống nút H là con trái của D, H là nút lá thông báo trường info của H
- Thăm nút I là con phải của D. I là nút lá thông báo trường info của I.
- đẩy D ra khỏi stack thông báo trường info của D
- thăm nút E là con phải của B. E là nút lá thông báo trường info của E.
- đẩy B ra khỏi stack, thông báo trường info của B
- thăm nút C là con phải của A. C có con trái và con phải, đẩy C vào stack
- thăm F là con trái của C. F có con phải, đẩy F vào stack.
- Thăm J là con phải của F. J là nút lá thông báo trường info của J
- đẩy F ra khỏi stack, thông báo trường info của F.
- thăm G là con phải của C, G là nút lá thông báo trường info của G
- đẩy C ra khỏi stack, thông báo trường info của C.
- đẩy A ra khỏi stack thông báo trường info của A.

Vậy sau khi duyệt thì danh sách các trường info của các nút được liệt kê là:

H I D E B J F G C A

5 BÀI TOÁN TÌM KIẾM CHIỀU SÂU

5.1 Cài đặt đệ qui:

Dữ liệu vào: đơn đồ thị vô hướng $G = (V, E)$ gồm n đỉnh, m cạnh. Các đỉnh được đánh số từ 1 đến n . Đỉnh xuất phát S , đỉnh đích F .

Dữ liệu ra:

a, Tất cả các đỉnh có thể đến được từ S.

b, Một đường đi đơn (nếu có) từ S đến F.

Tư tưởng của thuật toán: trước hết, mọi đỉnh x kè với S tất nhiên sẽ đến được từ S. Với mỗi đỉnh x kè với S đó thì tất nhiên những đỉnh y kè với x cũng đến được từ S... Điều đó gợi ý cho ta viết một thủ tục đệ qui DFS(v) mô tả việc duyệt từ đỉnh u bằng cách thông báo thăm đỉnh u và tiếp tục duyệt DFS(v) với v là một đỉnh chưa thăm kè với u.

- Để không một đỉnh nào bị liệt kê tới 2 lần, ta sử dụng kĩ thuật đánh dấu, mỗi lần thăm một đỉnh, ta đánh dấu đỉnh đó lại để các bước duyệt đệ qui kế tiếp không duyệt lại đỉnh đó nữa.
- Để lưu lại đường đi từ đỉnh xuất phát S, trong thủ tục DFS(u), trước khi gọi đệ qui DFS(v) với v là một đỉnh kè với u mà chưa đánh dấu, ta lưu lại vết đường đi từ u tới v bằng cách đặt TRACE[v]:=u, tức là TRACE[v] lưu lại đỉnh liền trước v trong đường đi từ S tới v. Khi quá trình tìm kiếm theo chiều sâu kết thúc, đường đi từ S tới F sẽ là:

$$F \leftarrow p_1 = \text{Trace}[F] \leftarrow p_2 = \text{Trace}[p_1] \leftarrow \dots \leftarrow S.$$

Procedure DFS($u \in V$);

Begin

1. <Thông báo tới được u>;
2. <Đánh dấu u là đã thăm (có thể đến được từ S)>;
3. <Xét mọi đỉnh v kè với u mà chưa thăm, với mỗi đỉnh v đó>;

Begin

 TRACE[v]:= u;

 DFS(v);

end;

5.2 Cài đặt bằng Stack

Ta thấy với quá trình thể hiện đệ qui ở trên, ta thấy nếu dây chuyền đệ qui là: $\text{DFS}(S) \rightarrow \text{DFS}(u_1) \rightarrow \text{DFS}(u_2) \dots$ Thì thủ tục đệ qui DFS nào gọi cuối dây chuyền sẽ được thoát ra đầu tiên, thủ tục DFS(S) gọi đầu dây chuyền sẽ thoát ra cuối cùng. Vậy nên chẳng, ta có thể mô tả dây chuyền đệ qui bằng một ngăn xếp.

Khi mô tả quá trình đệ quy bằng một ngăn xếp, ta luôn luôn có thể cho ngăn xếp lưu lại dây chuyền duyệt sâu từ nút gốc (đỉnh xuất phát S).

<Thăm S, đánh dấu S đã thăm>;

<Đẩy S vào ngăn xếp>;

repeat

<lấy u khỏi ngăn xếp>;

if <u có đỉnh kè chưa thăm> then

begin

<chỉ chọn lấy một đỉnh v, là đỉnh đầu tiên kè u mà chưa được thăm>

<thông báo thăm v>;

<đẩy u trở lại ngăn xếp>;

<đẩy tiếp v vào ngăn xếp>;

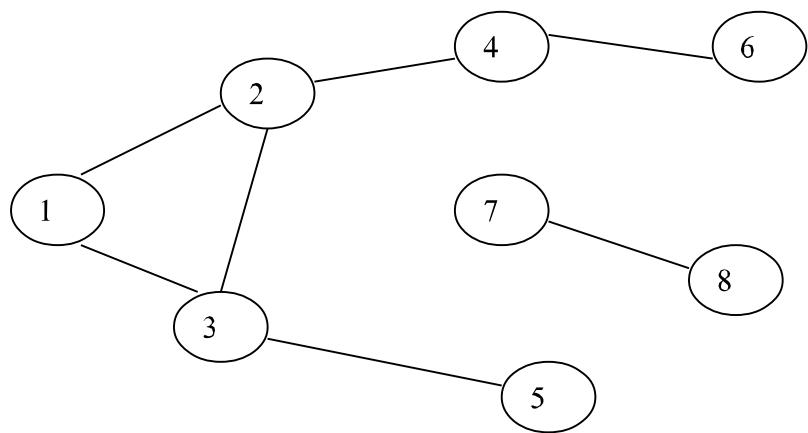
end;

{Còn nếu u không có đỉnh kè chưa thăm thì ngăn xếp sẽ ngắn lại, tương ứng với quá trình lùi về của dây chuyền DFS}

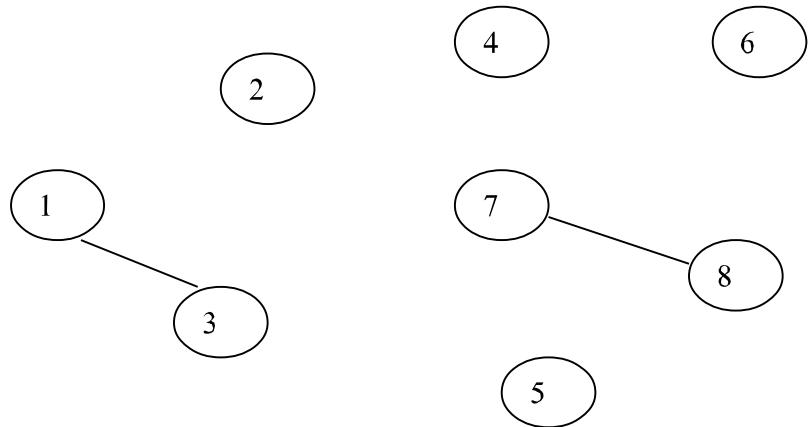
until <ngăn xếp rỗng>;

Ta xét một ví dụ:

Ta có một đồ thị như hình ở dưới đây:



Với đồ thị trên, đỉnh xuất phát $S = 1$; quá trình duyệt đệ quy có thể vẽ trên cây tìm kiếm DFS sau (mũi tên $u \rightarrow v$ chỉ thao tác đệ qui: $\text{DFS}(u)$ gọi $\text{DFS}(v)$).



Thứ tự duyệt là $\text{DFS}(1) \rightarrow \text{DFS}(2) \rightarrow \text{DFS}(3) \rightarrow \text{DFS}(5) \rightarrow \text{DFS}(4) \rightarrow \text{DFS}(6)$.

Cũng với đồ thị trên, ta thử theo dõi quá trình thực hiện tìm kiếm theo chiều sâu dùng ngăn xếp và đối sánh thứ tự duyệt các đỉnh với thứ tự như ta dùng đệ qui.

Trước hết, ta thăm đỉnh 1 và đẩy nó vào ngăn xếp:

Bước lặp	Ngăn xếp	u	v	Ngăn xếp sau mỗi bước	Giải thích
1	1	1	2	1, 2	xuống thăm 2
2	1, 2	2	3	1, 2, 3	xuống thăm 3
3	1, 2, 3	3	5	1, 2, 3, 5	xuống thăm 5
4	1, 2, 3, 5	5	Không có	1, 2, 3	Lùi lại
5	1, 2, 3	3	Không có	1, 2	Lùi lại
6	1, 2	2	4	1, 2, 4	xuống thăm 4
7	1, 2, 4	4	6	1, 2, 4, 6	xuống thăm 6
8	1, 2, 4, 6	6	Không có	1, 2, 4	Lùi lại
9	1, 2, 4	4	Không có	1, 2	Lùi lại
10	1, 2	2	Không có	1	Lùi lại
11	1	1	Không có	rỗng	Lùi hết dây chuyền, xong.

Thứ tự duyệt là 1, 2, 3, 5, 4, 6.

TẠM KẾT

Như vậy, thông qua các ví dụ trên, tuy chưa là hoàn đầy đủ, nhưng ta đã có thể thấy rõ được tác dụng, vai trò của stack trong mỗi giải thuật đệ qui trên. Báo cáo vẫn còn chưa đầy đủ về nội dung và có thể còn sai sót đâu đó. Em mong các thầy cô đóng góp ý kiến cho em, để em có thể hoàn thiện bổ sung cho bài luận văn sắp tới. Phương hướng phát triển của báo cáo là không dừng lại ở nghiên cứu lý thuyết mà sẽ hướng tới cài đặt chương trình mô phỏng hoạt động của stack và ứng dụng của stack trong các giải thuật đệ qui, vì vậy những đóng góp của các thầy cô sẽ là những đóng góp quý báu cho em. Em xin chân thành cảm ơn thầy Nguyễn Hữu Dung đã chỉ bảo giúp đỡ em trong thời gian vừa qua!