Mục lục

## Câu 1: Trình bày mối quan hệ giữa cấu trúc dữ liệu và giải thuật. Cho ví dụ minh họa

Cấu trúc dữ liệu và giải thuật có mối quan hệ mật thiết.
Giải thuật là một hệ thống chặt chẽ và rõ ràng các qui tắc nhằm xác định 1 dãy các thao tác trên những đối tượng, sao cho sau 1 số bước hữu hạn thực hiện các thao tác đó ta thu được kết quả mong muốn.
Cấu trúc dữ liệu: là cách tổ chức, lưu trữ dữ liệu trong MTDT 1 cách có thứ tự, có hệ thống nhằm sử dụng dữ liệu 1 cách hiệu quả
Ctdl và gt có mối liên hệ chặt chẽ với nhau, chúng luôn tồn tại song song đi kèm nhau theo công thức: ctdl+gt=ctrinh
Bản thân các phần tử của dữ liệu thường có mối quan hệ với nhau, ngoài ra nếu biết tổ chức chúng theo các cấu trúc dữ liệu thích hợp thì việc thực hiện các phép xử lý trên các dữ liệu sẽ càng thuận lợi hơn, đạt hiệu quả cao hơn. Với 1 ctdl đã chọn ta sẽ có giải thuật xử lý tương ứng. Ctdl thay đổi thì giải thuật cũng thay đổi theo. Để có 1 ctrinh tốt, ta cần phải chọn được ctdl phù hợp và chọn được 1 gt đúng đắn
Vd: Giả sử ta có 1 danh sách các trường đại học và cao đẳng trên cả nước mỗi trường có các thông tin sau: Tên trường, địa chỉ, sđt phòng đào tạo. Ta muốn viết một chương trình trên máy tính điện tử để khi cho biết "tên trường" máy sẽ hiện ra màn hình cho ta: "địa chỉ" và "số điện thoại phòng đào tạo" của trường đó.
1 cách đơn giản là cứ duyệt tuần tự các tên trường trong dnah sách cho tới khi tìm thấy trên trường cần tìm thì sẽ đói chiếu ra "địa chỉ" và "số điện thoại phòng đào tạo" của trường đó. Cách tìm tuần tự này rõ ràng chỉ chấp nhận được khi danh sách ngắn còn danh sách dài thì rất mất thời gian.
Nếu ta biết tổ chức lại danh sách bằng cách sắp xếp theo thứ tự từ điển của tên trường, thì có thể áp dụng 1 giải thuật tìm kiếm khác tốt hơn, tương tự như ta vẫn thường làm khi tra từ điển. Cách tìm này nhanh hơn cách trên rất nhiều nhưng không thể áp dụng được với dữ liệu chưa được sắp xếp.
Nếu lại biết tổ chức thêm 1 bảng mục lục chỉ dẫn theo chữ cái đầu tiên của tên trường, thì khi tìm "địa chỉ" và "số điện thoại phòng đào tạo" của Hvktmm ta sẽ bỏ qua được các tên trường mà chữ cái đầu không phải là "H".
Như vậy giữa cấu trúc dl và gt có mqh mật thiết. Có thể coi chúng như hình với bóng, không thể nói gới cái này mà không nhắc tới cái kia.

## Câu 2: Cấu trúc dữ liệu và phép toán

Đối với các bài toán phi số, đi đôi với các cấu trúc dữ liệu mwosi cũng xuất hiện các phép toán mới tác động trên các cấu trúc ấy. Thông thường có các phép toán như : phép tạo lập hoặc hủy bỏ một cấu trúc, phép truy nhập vào từng phần tử của cấu trúc, phép bổ sung hoặc laoij bỏ một phần tử trên cấu trúc…

Các phép toán đó sẽ có những tác dụng khác nhau đối với từng cấu trúc. Có phép toán hữu hiệu đối với cấu trúc này nhưng lại tỏ ra không hữu hiệu trên các cấu trúc khác.
Vì vậy khi chọn một cấu trúc dữ liệu ta phải nghĩ ngay tới các phép toán tác động trên cấu trúc ấy và ngược lại, nói tới phép toán thì lại phải chú ý tới phép đó đk tác động trên cấu trúc dữ liệu nào. Cho nên người ta thường quan niệm : nói tới cấu trúc dữ liệu là bao hàm luôn cả phép toán tác động đến cấu trúc ấy.

**Câu 3: Trình bày sự khác nhau của cấu trúc dữ liệu và cấu trúc lưu trữ, cho vd minh họa?**

Cách biểu diễn một cấu trúc dữ liệu trong bộ nhớ máy tính điện tử đk gọi là cấu trúc lưu trữ . Đó chính là cách cài đặt cấu trúc ấy trên máy tính điện tử và trên cơ sở cấu trúc lưu trữ này mà thực hiện các phép xử lí . Ta cần phân biệt giữa cấu trúc dữ liệu và cấu trúc lưu trữ tương ứng. Có thể có nhiều cấu trúc lưu trữ khác nhau cho cùng một cấu trúc dữ liệu, cũng như có thể có những cấu trúc dữ liệu khác nhau mà đk thể hiện trong bộ nhớ bởi cùng một kiểu cấu trúc lưu trữ .

Vd: cấu trúc lưu trữ kế tiếp ( mảng) và cấu trúc lưu trữ móc nối đều có thể đk dùng để cài đặt cấu trúc dữ liệu ngăn xếp (stack). Mặt khác, các cấu trúc dữ liệu nhue : danh sách, ngân xếp và cây đều có thể cài đặt trên máy thông qua cấu trúc lưu trữ móc nối.

**Câu 4: Trình bày những đặc điểm về cấu trúc trong các ngôn ngữ lt bậc cao, có liên hệ với ngôn ngữ C**

Trong các ngôn ngữ lập trình bậc cao, các dữ liệu được phân nhánh thành các kiểu dữ liệu. kiểu dữ liệu nhận của một biến được xác đinh bởi một tập các giá trị mà biến đó có thể nhận và các phép toán có thể thực hiện trên các giá trị đó.
Mỗi ngôn ngữ lập trình cung cấp cho chúng ta một số kiểu dữ liệu cơ bạn . trong các ngôn ngữ lập trình khác nhau , các kiểu dữ liệu cơ bản có thể khác nhau . Các ngôn ngữ lập trình như C, pascal… có các kiểu dữ liệu cơ bản rất phong phú.
Các kiểu dữ liệu đk tạo thành từ nhiều kiểu dữ liệu khác nhau được gọi la kiểu dữ liệu có cấu trúc. Các dữ liệu thuộc kiểu dữ liệu cấu trúc được gọi là cấu trúc dữ liệu.
Từ các kiểu cơ bản , bằng cách sử dụng các quy tắc ,cú pháp để kiến tạo các kiểu dữ liệu, người lập trình có thể xây dựng nên được gọi là các kiểu dữ liệu xác định bởi người sử dụng.
=> Như vậy: một cấu trúc dữ liệu phức hợp gồm nhiều thành phần dữ liệu, mỗi thành phần hoặc là dữ liệu cơ sở hoặc là cấu trúc dữ liệu đã đk xây dựng. Các thành phần dữ liệu tạo nên một cấu trúc dữ liệu đk liên kết với nhau theo một cách nào đó.
Trong ngôn ngữ lập trình C phương pháp để liên kết dữ liệu :
+) Liên kết dữ liệu cùng kiểu tạo thành mảng dữ liệu.
+) Liên kết các dữ liệu thành mảng cấu trúc trong C.

+) Sử dụng con trỏ để liên kết dữ liệu.

## Câu 5 : Phương pháp thiết kế Top_Down

Ngày nay công nghệ thông tin đã và đang được ứng dụng trong mọi lĩnh vực của cuộc sống, bởi vậy các bài toán giải được trên máy tính điện tử rất đa dạng vào phức tạp, các giải thuật và chương trình để giải chúng cũng có quy mô ngày càng lớn , nên càng khó thì ta càng muốn tìm hiểu và thiết lập chúng.

Tuy nhiên ta cũng thấy rằng mọi việc  sẽ đơn giản hơn nếu như có thể phân chia bài toán lớn thành các bài toán nhỏ hơn. Điều đó cũng có nghã là nếu coi bài toán của ta như một mô đun chính thì cần chia nó thành các mô đun con, và dĩ nhiên, với tinh thần như thế, đến lượt nó, mỗi mô đun con này lại tiếp tục được chia tiếp cho tới những mô đun ứng với các phần việc cơ bản mà ta đã biết cách giải quyết. Như vậy việc tổ chức lời giải của các bài toán sẽ được thể hiện  theo một cấu trúc nhân cấp có dạng như sau :

Cách giải quyết bài toán theo hình như vậy được gọi là chiến thuật " chia để trị" . Để thể hiện chiến thuật đó, người ta dùng cách thiết kế " đinh_xuống" (top-down design). Đó là cách phân tích tổng quát toàn bộ vấn đề, xuất phát từ dữ kiện và các mục tiêu đặt ra để đề cập đến những công việc chủ yếu trước, rồi sau đó mwosi đi dần vào giải quyết các phần việc cụ thể một cách chi tiết hơn, cũng vì vậy mà người ta còn gọi cách thiết kế này là cách thiết kế từ khái quát đến chi tiết.

Ví dụ: để viết chương trình quản lí bán hàng chạy trên máy tính, với các yêu cầu là : hàng ngày phải nhập các hóa đơn bán hàng, hóa đơn nhập hàng, tìm kiếm các hóa đơn đã nhập để xem hoặc sửa lại. in các hóa đơn cho khách hàng; tính doanh thu, lợi nhuận trong khoảng thời gian bất  kì; tính tổng hợp kho, tính doanh số của từng mặt hàng, từng khách hàng.

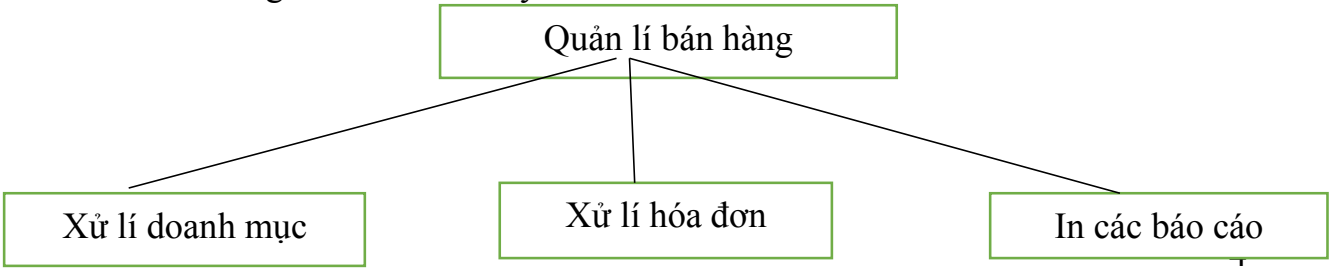
Xuất phát từ những yêu cầu trên ta không thể có ngay giải thuật để xử lí, mà nên chia bài toán thành 3 nhiệm vụ chính cần giải quyết như sau:

Xử lí các danh mục để quản lí và theo dõi các thông tin về hàng hóa và khách hàng.

Xử lí dữ liệu về các hóa đơn bán hàng, hóa đơn nhập hàng.

In các báo cáo về doanh thu, lợi nhuận.

Có thể hình dung cách thiết kế này theo sơ đồ cấu trúc sau:

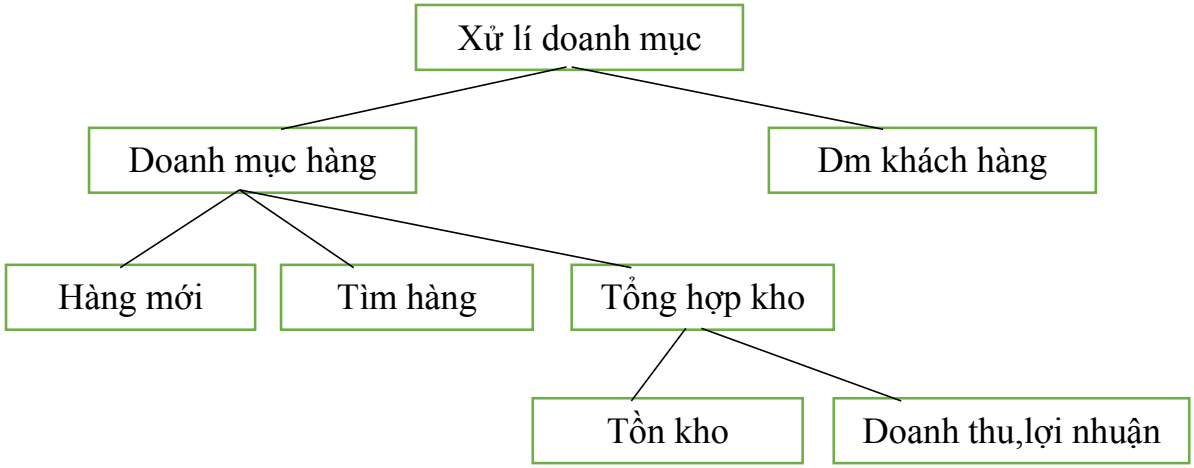


Chia bài toán chính thành 3 bài toán nhỏ

Các nhiệm vụ ở mức đầu này thường vẫn còn tương đối phức tạp, nên cần phải chia tiếp thành các nhiệm vụ con. Chẳng hạn nhiệm vụ " xử lí doanh mục" được chia thành hai là " danh mục hàng hóa" và " danh mục khách hàng".

Trong danh mục hàng hóa lại có thể chia thành các nhiệm vụ nhỏ hơn như:
Thêm hàng mới
Tìm kiếm hàng
Tổng hợp kho

Những nhiệm vụ con này cũng có thể chia thành các nhiêm vụ nhỏ hơn , ta có thể hình dung theo sơ đồ sau:




Cách thiết kế giải thuật theo kiểu top-down như trên giúp cho việc gải quyết bài toán được  định hướng rõ ràng , tránh sa đà ngay vào các chi tiết phụ. Nó cũng là các nền tảng cho việc lập trình có cấu trúc.
Thông thường, đối với các bài toán lớn, việc giải quyết nó phải do nhiều người cùng làm . Chính phương pháp mô đun hóa sẽ cho phép tách bài toán ra thành các phần độc lập, tạo điều kiện cho các nhóm giải quyết phần việc của mình mà không ảnh hưởng gì đến nhóm khác. Với chương trình được xây dựng trên cơ sở của các giải thuật được thiết kế theo cách này , thì việc tìm hiểu cũng như sửa chữa, chỉnh lí sẽ đơn giản hơn.
Trong thực tế, việc phân tích bài toán thành các bài toán con như thế không phải là việc dễ dàng. Chính vì vậy mà có những bài toán, nhiệm vụ phân tích và thiết kế giải thuật giải bài toán còn mất nhiều thời gian và công sức hơn cả nhiệm vụ lập trình.

## Câu 6: Phương pháp tinh chỉnh từng bước ( stepwise refinement)

Tinh chỉnh bước là phương pháp thiết kế giải thuật gắn liền với lập trình. Nó phản ánh tinh thần của quá trình mô đun hóa bài toán và thiết kế kiểu top-down.

Ban đầu chương trình thể hiện giải thuật được trình bày bằng ngôn ngữ tự nhiên, phản ánh ý chính của của công việc cần làm. Từ các bước sau, những lời , những ý đó sẽ được chi tiết hóa dần dần tương ứng với những công việc nhỏ hơn. Ta gọi đó là các bước tinh chỉnh, sự tinh chỉnh này sẽ hướng về phía ngôn ngữ lập trình mà ta đã chọn. Càng ở các bước sau, các lời lẽ đặc tả công việc cần xử lí sẽ được thay thế dần bởi các câu lệnh hướng tới câu lệnh của ngôn ngữ lập trình. Muốn vậy, ở các giai đoạn trung gian người ta thường dùng pha tạp cả ngôn ngữ tự nhiên lẫn ngôn ngữ lập trình, mà người ta gọi là " giả ngôn ngữ" hay " giả mã". Như vậy nghĩa là quá trình thiết kế giải thuật và phát triển chương trình sẽ được thể hiện dần dần, từ dạng ngôn ngữ tự nhiên, qua giả ngôn ngữ, rồi đến ngôn ngữ lập trình, và đi từ mức" làm cái gì" đến mức " làm như thế nào", ngày càng sát với các chức năng ứng với các câu lệnh của ngôn ngữ lập trình đã chọn.

Trong quá trình này dữ liệu cũng được " tính chế " dần dần từ dạng cấu trúc dữ liệu đến dạng cấu trúc lưu trữ cụ thể trên máy.

Các bước: diễn đạt gt bằng ngôn ngữ tự nhiên. Thay thế lời lẽ đặc tả công việc bằng các câu lệnh hướng tới câu lệnh của ngôn ngữ ltrinh, dùng giả ngôn ngữ hay giả mã. Viết bằng n2 lập trình

## Câu 7: Trình bày cách phân tích thời gian thực hiện giải thuật

Thời gian thực hiện một giải thuật phụ thuộc vào rất nhiều yếu tố. 1 yếu tố cần chú ý trc tiên đó là kích thước của dữ liệu đưa vào. Chẳng hạn thời gian sắp xếp 1 dãy số phải chịu ảnh hưởng của số lượng các số thuộc dãy số đó. Nếu gọi n là số lượng này thì thời gian thực hiện T của 1 giải thuật phải được biểu diễn như 1 hàm của n: T(n).

Các kiểu lệnh cà tốc độ xử lý của máy tính ngôn ngữ viết chương trình và chương trình dịch ngôn ngữ ấy đều ảnh hưởng tới thời gian thực hiện, nhưng những yếu tố này không đồng đều với mọi loại máy trên đó cài đặt giải thuật, vì vậy không thể dựa vào chúng khi xác lập T(n). Điều đố cũng có nghĩa là T(n) không thể được biểu diễn thành đơn vị thời gian bằng giây, bằng phút.. được. Tuy nhiên không phải vì thế mà không thể so sánh được các giải thuật về mặt tốc độ. Nếu thời gian thực hiện của 1 giải thuật là $T_1(n)=cn^2$ và thời gian thực hiện của 1 giải thuật khác là $T_2(n)=kn$ (với c và k là 1 hằng số nào đó) thì n khá lớn, thời gian thực hiện giải thuật T2 rõ ràng ít hơn so với giải thuật T1. Và như vậy thì nếu nói thời gian thực hiện giải thuật T(n) tỉ lệ với $n^2$ hay tỉ lệ với n cũng cho ta ý niệm về tốc độ thực hiện giải thuật đó khi n khá lớn (với n nhỏ thì việc xét T(n) không có ý nghĩa).

**Câu 8. Trình bày cách Xác định độ phức tạp tính toán của giải thuật, với những nội dung: Qui tắc tổng, phép toán tích cực, thời gian chạy của các câu lệnh lặp, cho ví dụ minh họa.**

Nếu thời gian thực hiện 1 giải thuật là $T(n)=cn^2$ (c là hằng số) thì ta nói : độ phức tạp tính toán của giải thuật này có cấp là $n^2$ và ta ký hiệu:

$T(n) = O(n^2)$ (ký hiệu chữ O lớn)

Một cách tổng quát có thể định nghĩa: 1 hàm f(n) được xác định là O(g(n))

$f(n) = O(g(n))$

và được gọi là có cấp g(n) nếu tồn tại các hằng số c và $n_0$ sao cho:

$f(n) <= cg(n)$ khi $n >= n_0$

nghĩa là khi f(n) bị chặn trên bởi 1 hằng số nhân với g(n) với mọi giá trị của n từ một thời điểm nào đó.

Quy tắc tổng: Giả sử $T_1(n)$ và $T_2(n)$ là thời gian thực hiện của 1 đoạn chương trình $P_1$ và $P_2$ mà $T_1(n) = O(f(n))$; $T_2(n) = O(g(n))$ thì thời gian thực hiện $P_1$ và $P_2$ tiếp theo sẽ là: $T_1(n) + T_2(n) = O(max(f(n),g(n))$.

Ví dụ: trong 1 chương trình có 3 bước thực hiện mà thời gian thực hiện từng bước lần lượt là $O(n^2)$, $O(n^3)$ và $O(n \log_2 n)$ thì thời gian thực hiện 2 bước đầu là $O(max(n^2,n^3))=O(n^3)$ thời gian thực hiện chương trình sẽ là:

$O(max(n^3, n \log_2 n))=O(n^3)$.

Thời gian chạy của các câu lệnh lặp:

Các câu lệnh lặp gồm: for, while, do.. while

Để đánh giá thời gian thực hiện 1 câu lệnh lặp, trước hết ta cần đánh giá số tối đa các lần lặp giả sử đó là L(n). Sau đó đnahs giái thời gian chạy của mỗi lần lặp, chú ý rằng thời gian thực hiện thân của 1 lệnh lặp ở các lần lặp khác nhau có thể khác nhau, giả sử thời gian thực hiện thân lệnh lặp ở lần thứ i(i=1,2,..L(n)) là $T_i(n)$. Mỗi lần lặp, chúng ta cần kiểm tra điều kiện lặp giả sử thời gian lặp kiểm tra là $T_0(n)$. Như vậy thời gian chạy của lệnh lặp là:

$\sum_{i=1}^{L(n)} ( T_0(n)+ T_i(n))$

Công đoạn khó nhất trong đánh giá thời gian chạy của 1 lệnh lặp là đánh giá số lần lặp. Trong nhiều lệnh lặp, đặc biệt là trong các lệnh lặp For, ta có thể thấy ngay số lần lặp tối đa là bao nhiêu. Nhưng cũng không ít các lệnh lặp, từ điều kiện lặp để suy ra số tối đa các lần lặp, ta cần phải tiến hành các suy diễn không đơn giản.

Trường hợp hay gặp là kiểm tra điều kiện lặp chỉ cần thời gian O(1), thời gian thực hiện các lần lặp là như nhau và giả sử ta đánh giá được là O(f(n)); khi đó nếu đánh giá được số lần lặp là O(g(n)) thì thời gian chạy của lệnh lặp là O(g(n)).f(n)

Ví dụ: giải sử có mảng A các số thực, cỡ n và ta cần tìm xem mảng có chứa số thực x không. Điều đó có thể thực hiện bởi giải thuật tìm kiếm tuần tự như sau:

i=0;

while(i<n&& x!=A[i])
i++;
lệnh gán (1) có thời gian chạy là O(1). Lệnh lặp (2)-(3) có số tối đa các lần lặp là n, đó là trường hợp x chỉ xuất hiện ở thành phần cuối cùng của mảng A[n-1] hoặc x không có trong mảng. Thân của lệnh lặp là lệnh (3) có thời gian chạy O(1). Do đó, lệnh lặp có thời gian chạy là O(n). Giải thuật gồm lệnh gán và lệnh lặp với thời gian là O(1) và O(n), nên thời gian chạy của nó là O(n).
phép toán tích cực: Đó là phép toán thuộc giải thuật mà thời gian thực hiện không ít hơn thời gian thực hiện các phép khác hay nói cách khác: số lần thực hiện nó không kém các phép khác.

**Câu 9 : Trình bày ( bằng ngôn ngữ tựa C ) giải thuật bổ sung một nút mới có chứa dữ liệu X vào trước nút con trỏ bởi Q trong danh sách móc nối hai chiều với : Pdau trỏ và phần tử đầu, Pcuoi trỏ vào phần tử cuối, mỗi nút có cấu trúc như sau :**

| | |
|---|---|
| P_L | trỏ tới con trỏ bên trái |
| DATA | chứa dữ liệu |
| P_R | trỏ tới con trỏ bên phải |

```
THEM_NUT ( Pdau, Pcuoi, Q, X)
{
    /*Cho con trỏ L, R lần lượt trỏ tới nút cực trái và nút cực phải của một danh sách móc nối kép, Q là con trỏ trỏ tới một nút trong danh sách này. Giải thuật được thực hiện bổ sung một nút mới, mà dữ liệu chứa ở X, vào trước nút trỏ bởi Q */
P = MALLOC(); // tạo một con trỏ mới
P -> DATA = X;
P -> P_L  = P -> P_R = NULL;
If ( Pcuoi == NULL )
{
      Pdau = Pcuoi = P;
}
Else
      If ( Q ==Pdau )
      {
            Q -> P_L = P;
            P -> P_R = Q;
            Pdau = P;
      }
      Else
      {
```

```
            P -> P_L = Q -> P_L;
            P -> P_R = Q;
            Q -> P_L = P;
            P -> P_L -> P_R = P;
        }
}
```

**Câu 10 : Trình bày ( bằng ngôn ngữ tựa C ) giải thuật loại bỏ một nút trỏ bởi Q trong danh sách móc nối hai chiều với : Pdau chỉ vào phần tử đầu, Pcuoi chỉ vào phần tử cuối, mỗi nút có cấu trúc như sau:**

| | |
|---|---|
| P_L | trỏ tới con trỏ bên trái |
| DATA | chứa dữ liệu |
| P_R | trỏ tới con trỏ bên phải |

```
XOA_NUT (Pdau, Pcuoi, Q )
{
/* L và R là 2 con trỏ trái và phải  của danh sách móc nối kép, Q trỏ tới một nút trong danh sách. Giải thuật thực hiện việc loại bỏ Q khỏi danh sách*/

If ( Pcuoi== NULL )
        Printf("Danh sách rỗng");
Else
        If ( Pdau == Pcuoi )
                Pdau= Pcuoi = NULL;
        Else
                If ( Q == Pdau )
                {
                Pdau = Q-> P_R
                Pdau -> P_L = NULL;
                }
                Else
                If ( Q == Pcuoi )
                {
                Pcuoi = Pcuoi -> P_L;
                Pcuoi -> P_R = NULL;
                }
                Else
                {
                Q -> P_L -> P_R = Q -> P_R;
                Q -> P_R -> P_L = Q -> P_L;
                }
```

```
        Free(Q);
        }
}
```

**Câu 11: Trình bày bằng ngôn ngữ tựa C giải thuật cộng 2 đa thức C = A + B. Các phần tử của mỗi đa thức có cấu trúc như sau**

| | |
|---|---|
| HSO | Ghi hệ số |
| MU | Ghi số mũ |
| NEXT | Ghi địa chỉ đến phần tử tiếp theo |

```
THEM_PHAN_TU ( H, M, D)
{
P = MALLOC();
P -> HSO = H;
P -> MU = M;
If ( C != NULL ) // đã có đuôi
        D -> NEXT = P;//gán P vào đuôi
Else // chưa có đuôi
        C = P;
D = P; // nút mới thêm trở thành đuôi
}

CONG_DA_THUC ( A, B, C)
{
P = A; Q = B; C= NULL;
While ( P == NULL && Q == NULL )
        If ( P -> MU == Q -> MU )
        {
        H = P -> HSO + Q -> HSO;
        If ( H != 0)
                THEM_PHAN_TU(H, P-> MU, D);
        P = P -> NEXT;
        Q = Q -> NEXT;
        }
        Else If( P -> MU > Q -> MU)
        {
        THEM_PHAN_TU ( P-> HSO, P-> MU, D);
        P = P-> NEXT;
        }
        Else
        {
        THEM_PHAN_TU(Q-> HSO; Q-> MU; D);
```

```
        Q = Q-> NEXT;
        }
If ( Q == NULL )//Danh sách ứng với B(x) đã hết
        While ( P != NULL )
        {
        THEM_PHAN_TU ( P-> HSO, P-> MU, D);
        P = P-> NEXT;
        }
Else //Danh sách ứng với A(x) đã hết
        While ( Q != NULL )
        {
        THEM_PHAN_TU ( Q -> HSO, Q -> MU, D);
        Q= Q -> NEXT
        }
D -> NEXT = NULL;
}
```

**Câu 12: Trình bày ( bằng ngôn ngữ tựa C ) giải thuật định giá biểu thực hậu tố bằng cách dùng stack.**

*Ý tưởng*

Ta xem biểu thức hậu tố như một dãy các thành phần, mà mỗi thành phần là toán hạng hoặc toán tử

B1: Khởi tạo 1 stack rỗng

B2: Đọc lần lượt các phần tử của biểu thức từ trái qua phải

Nếu là toán hạng, đẩy vào stack

Nếu là toán tử X, lấy từ stack ra 2 giá trị (Y và Z) sau đó áp dụng toán tử đó vào 2 giá trị vừa lấy ra, đẩy kết quả tìm được (Z X Y) vào stack

B3: sau khi kết thúc B2, thì tất cả biểu thức hậu tố đã đọc xong, trong stack còn duy nhất 1 phần tử là giá trị của biểu thức

*Giải thuật:*

```
DINH_GIA_BIEU_THUC ( )
{
/* Giải thuật này sử dụng một ngăn xếp S, được trỏ bởi con trỏ T, lúc đầu T = -1*/
Do
{
Đọc phần tử X tiếp theo trong biểu thức;
If X là toán hạng
        PUSH( S, T, X);
Else
```

```
{
        Y = POP ( S, T);
        Z = POP ( S, T);
        W = Z X Y;                  // thực hiện phép toán X
        PUSH( S, T, W);
}
}
While ( gặp dấu kết thúc );
R = POP ( S, T);
Printf ( R );
}
```

## Câu 13: chuyển đổi biểu thức trung tố sang hậu tố

*Ý tưởng:*

1. khởi tạo 1 ngăn xếp (stack) rỗng
2. đọc lần lượt các thành phần trong biểu thức

nếu X là toán hạng thì viết nó vào biểu thức hậu tố (in ra)

nếu X là phép toán thì thực hiện:

+ nếu stack không rỗng thì: nếu phần tử ở đỉnh stack là phép toán có độ ưu tiên cao hơn hoặc bằng phép toán hiện thời (X) thì phép toán đó được kéo ra khỏi stack và viết vào biểu thức hậu tố (lặp lại bước này)
+ nếu stack rỗng hoặc phần ử ở đỉnh ngăn xếp là dấu mở ngoặc hoặc phép toán ở đỉnh ngăn xếp có quyền ưu tiên thấp hơn phép toán hiện thời (X) thì phép toán hiện thời được đẩy vào ngăn xếp

nếu X là dấu mở ngoặc thì nó được đẩy vào stack

nếu X là dấu đóng ngoặc thì thực hiện:

+ (bước lặp):loại các phép toán ở đỉnh ngăn xếp và viết vào biểu thức dạng hậu tố cho tới khi đỉnh ngăn xếp là dấu mở ngoặc
+ loại dấu mở ngoặc khỏi ngăn xếp

3. sau khi toàn bộ biểu thức dạng trung tố được đọc, loại lần lượt các phép toán ở đỉnh stack và viết vào biểu thức hậu tố cho tới khi stack rỗng

*Giải thuật:*

```
TRUNGTOHAUTO()
{ //giải thuật này sử dụng 1 stack S, trỏ bởi T, lúc đầu T=-1
do
{
        Đọc thành phần X tiếp theo trong biểu thức;
        if (X là toán hạng)
                printf(X);
```

```
        else if (X là phép toán)
                do
                 {
                if ((T>-1) && (S[T] là phép toán có độ ưu tiên cao hơn X))
                        printf(POP(S,T));
                if ((T==-1) || (S[T]=='(' || (S[T] là phép toán có độ ưu tiên thấp hơn X))
                        PUSH(S,T,X);
                }
                while (phép toán X được đưa vào S)
        else if (X là dấu '(' )
                PUSH(S,T,X);
        else if (X là dấu ')' )
        {
        do
                printf(POP(S,T)); //in ra các phép toán
        while (S[T]==')');
        POP(S,T); //loại dấu ')' ra khỏi stack
        }
}
while (chưa gặp dấu kết thúc biểu thức dạng trung tố)
do
        printf(POP(S,T)); //in ra các phép toán
while(T>-1);
}
```

**Câu 14: Trình bày (nn tựa C) giải thuật duyệt cây theo thứ tự trước, ko đệ quy, dùng stack**

*Ý tưởng:*

1. kiểm tra rỗng
nếu cây rỗng thì kết thúc
nếu không rỗng thì khởi tạo stack
2. thực hiện duyệt
in ra khóa của nút gốc
nếu cây con phải khác rỗng thì lưu địa chỉ gốc cây con phải vào stack
chuyển xuống cây con trái, in ra khóa của nút con trái... (lặp lại)

*Giải thuật:*

T là con trỏ trỏ tới gốc cây đã cho.
S là 1 ngăn xếp (stack) được cài đặt bằng mảng với biến trỏ TOP trỏ tới đỉnh.
Con trỏ P được dùng để trỏ tới nút hiện đang được xét
Có sử dụng các hàm PUSH và POP.
PUSH: Bổ sung 1 phần tử vào ngăn xếp.

POP: Loại 1 phần tử ở đỉnh ngăn xếp đang được trỏ bởi T.

```
TT_TRUOC(T)
{

if(T==NULL)
        {

        Return;
        }
Else
        {
        TOP = -1 ;
        PUSH(S,TOP,T);
        }
While(TOP > -1)
        {
        P = POP(S,TOP);
        While(P!=NULL)
                {
                Printf(P-> DATA);
                If(P -> P_R! = NULL) PUSH(S,TOP, P->P_R);
                P = P -> P_L;
                }
        }
}
```

**Câu 15: Trình bày giải thuật duyệt cây theo thứ tự giữa bằng giải thuật ko đệ quy có sử dụng stack**

*Ý tưởng:*

1. kiểm tra rỗng

nếu cây rỗng thì kết thúc

nếu không rỗng thì khởi tạo stack

2. thực hiện duyệt

lưu địa chỉ của nút gốc vào stack, chuyển xuống cây con trái (lặp lại bước này tới khi cây con trái là rỗng)

lấy phần tử trên cùng ra khỏi stack, trỏ vào vị trí của nút đó trên cây

in ra khóa của nút đang xét

trỏ đến cây con phải

.... (lặp lại cho tới khi stack rỗng)

*Giải thuật:*

T là con trỏ trỏ tới gốc cây đã cho

S là 1 ngăn xếp (stack) được cài đặt bằng mảng với biến trỏ TOP trỏ tới đỉnh
Con trỏ P được dùng để trỏ tới nút hiện đang được xét
Có sử dụng các hàm PUSH và POP.
PUSH: Bổ sung 1 phần tử vào ngăn xếp.
POP: Loại 1 phần tử ở đỉnh ngăn xếp đang được trỏ bởi T.

```
TT_GIUA
{

If(T == NULL)
{

        Return;
}
Else
        {
        TOP = -1;
        P=T;
        }
While(TOP >-1 || P !=NULL)
{
        If(P==NULL)
                {
                P=POP(S,TOP);
                Printf("P->DaTa");
                P=P->R;
                }
        Else
                {
                PUSH(S,TOP,P);
                P = P->L;
                }
}
}
```

---

*Ý tưởng:*

giả sử dãy ban đầu được sắp xếp theo thứ tự tăng dần (K0<K1<...<Kn)
ta chọn khóa ở "giữa" (giả sử Kg) của dãy đang xét để so sánh
+ nếu x = Kg : tìm thấy x trong dãy, dừng quá trình tìm kiếm

+ nếu x < Kg : nếu x có trong dãy thì x nằm ở nửa bên trái của Kg
+ nếu x > Kg : nếu x có trong dãy thì x nằm ở nửa bên phải của Kg
việc tìm kiếm x trên nửa bến trái (hoặc bên phải) của Kg được thực hiện như việc tìm x trên cả dãy ban đầu.

*Giải thuật*

```
TimKiem_đq(K,t,p,x)
{
        If(t<p)
return -1;
        Else
                {
                g=(t+p)/2;
                if( x==K[g] )  return g;
                if( x<K[g]  ) TimKiem_dq(K,t,g-1,x);
                else TimKiem_dq(K,g+1, p,x)
                }
}
```

```
TimKiem_k(K,n,x)
{
        t=0; p = n -1;
        while( t<=p)
                {
                g=( t+p)/2;
                if( x == K[g] ) return g;
                else
                        if( x < K[g] ) p = g-1;
                else t= g+1;
                }
        Return -1;
}
```

*Đánh giá thời gian thực hiện:*

- trường hợp tốt nhất, phần tử giữa mảng ban đầu có giá trị bằng x, lúc này chỉ cần thực hiện 1
phép so sánh

=> Ttốt(n)= O(1)
- trường hợp xấu nhất, phần tử cuối cùng (hoặc đầu tiên) có giá trị bằng x hoặc không có x trong
dãy
=> khi đó dãy liên tiếp được chia đôi và ta phải gọi đệ quy cho tới khi dãy khóa đc xét chỉ
còn 1 phần tử
- giả sử gọi w(n) là hàm biểu thị số lượng các phép so sánh trong trường hợp xấu nhất, ta có
w(n) = 1 + w([n/2])
w(n) = 1 + 1 + w([n/2^2])
w(n) = 1 + 1 + 1 + w([n/2^3])
...
tại bước k ta có:
w(n) = k + w([n/2^n]) (*)
- quá trình gọi đệ quy dừng lại khi dãy chỉ còn 1 phần tử, tức là khi [n/2^k]=1
ta có, w([n/2^k]) = w(1) = 1, và khi [n/2^k]=1 thì suy ra 2^k <= n <= 2^(k+1)
suy ra k <= log(2)n <= k+1, nghĩa là có thể viết: k = [log(2)n]
thay vào (*)
w(n) = [log(2)n] + w(1) = [log(2)n] +1
- như vậy: Txấu(n) = O(log(2)n)
- KẾT LUẬN: Ttb(n) = O(log(2)n)

**Câu 17: kiểm tra xem T có phải là "cây nhị phân tìm kiếm" hay ko**

*Ý tưởng:*

- tạo 1 hàm tìm nút có giá trị lớn nhất của 1 cây (max)
- tạo 1 hàm tìm nút có giá trị nhỏ nhất của 1 cây (min)
- tạo 1 hàm kiểm tra xem cây có phải là cây tìm kiếm nhị phân hay ko
    + nếu cây rỗng thì nó là cây nhị phân tìm kiếm (return 0)
    + đầu tiên kiểm tra cây con trái (Left) có phải cây nhị phân tìm kiếm hay ko
        * nếu đúng thì chuyển xuống bước tiếp theo
        * sai thì return 1 (cây nhị phân đang xét không phải cây nhị phân tìm kiếm)
    + kiểm tra cây nhị phân đang xét
        * trường hợp 1: cây đang xét có cả 2 cây con trái và phải
        = tìm max cây con trái(MaxL), min cây con phải(MinR) sau đó so sánh
        với khóa tại nút gốc
        = nếu không thỏa mãn MaxL<key && key<MinR thì cây đó ko phải cây nhị phân t.kiếm
        * trường hợp 2: cây đang xét chỉ có cây con phải

= tìm min cây con phải, so sánh với khóa tại nút
= nếu không thỏa mãn key < MinR thì cây đó không phải cây nhị phân
t.kiếm
* trường hợp 3: cây đang xét chỉ có cây con trái
= tìm max cây con cái, so sánh với khóa tại nút
= nếu không thỏa mãn MaxL < key thì cây đó không phải cây nhị phân
t.kiếm

+ tiếp tục kiểm tra đối với cây con phải

*Giải thuật:*

```
TimMax(T, Max) // Tìm giá trị khoá Max của cây T
        {
        if (T==NULL)
                return;
        if (T->P_L != NULL)
                Max = (Max > T->P_L->KEY)? Max : T->P_L->KEY;
        if (T->P_R != NULL)
                Max = (Max > T->P_R->KEY)? Max : T->P_R->KEY;
        Max = (Max > T->KEY) ? Max : T->KEY;
        TimMax(T->P_L, Max);
        TimMax(T->P_R, Max);
        }
TimMin(T, Min) // Tìm giá trị khoá Min của cây T
        {
        if (T==NULL)
                return;
        if (T->P_L != NULL)
                Min = (Min < T->P_L->KEY)? Min : T->P_L->KEY;
        if (T->P_R != NULL)
                Min = (Min < T->P_R->KEY)? Min : T->P_R->KEY;
        Min = (Min < T->KEY) ? Min : T->KEY;
        TimMin(T->P_L, Min);
        TimMin(T->P_R, Min);
        }
KiemTra(T)// Nếu kết quả là 0 thì T là cây nhị phân tìm kiếm
        {
        if (T==NULL)
                return 0;
        Left = KiemTra(T->P_L);
        If  (Left) // Cây con trái không là cây nhị phân tìm kiếm
                return 1;
```

```
	if (T->P_L != NULL && T->P_R != NULL) // T Có 2 con
		{
		TimMax(T->P_L, MaxL);
		TimMin(T->P_R, MinR);
		if (!(MaxL < T->KEY && T->KEY < MinR))
			return 1;
		}
	else if (T->P_L == NULL && T->P_R != NULL)// Chỉ có con phải
		{
		TimMin(T->P_R, MinR);
		if (!(T->KEY < MinR))
			return 1;
		}
	else if (T->P_L != NULL && T->P_R == NULL)//Chỉ có con trái
		{
		TimMax(T->P_L, MaxL);
		if (!(MaxL < T->KEY))
			return 1;
		}
	Right = KiemTra(T->P_R);
	return Left + Right;
}
```

---

*Ý tưởng*

- Tìm kiếm

Trước hết, khoá tìm kiếm X được so sánh với khoá ở gốc cây, và 4 tình huống có thể xảy ra:

- Không có gốc (cây rỗng): X không có trên cây, phép tìm kiếm thất bại
- X trùng với khoá ở gốc: Phép tìm kiếm thành công
- X nhỏ hơn khoá ở gốc, phép tìm kiếm được tiếp tục trong cây con trái của gốc với cách làm tương tự
- X lớn hơn khoá ở gốc, phép tìm kiếm được tiếp tục trong cây con phải của gốc với cách làm tương tự

- Bổ sung

Ta chèn khoá vào cây, trước khi chèn, ta tìm xem khoá đó đã có trong cây hay chưa, nếu đã có rồi thì bỏ qua, nếu nó chưa có thì ta thêm nút mới chứa khoá cần chèn và nối nút đó vào cây nhị phân tìm kiếm tại mối liên kết vừa rẽ sang khiến quá trình tìm kiếm thất bại.

*Giải thuật*
BTS(T,x,Q)

```
P == NULL; Q=T;
While( Q! = NULL)// tìm kiếm, con trỏ P trỏ vào nút cha của Q
        {
        if( x == Q-> KEY) return;
        if( x < Q -> KEY)
                {
                P=Q;
                Q = Q-> P_L;
                }
        Else
                {
                P=Q;
                Q=Q-> P_R;
                }
        }
Q = maloc(); //chưa có khóa x, thực hiện bổ sung
Q->KEY =x;
Q->P_L = Q-> P_R = NULL;
if( T==NULL) //cây rỗng, nút mới chính là gốc
         T=Q;
else
        if(x < P-> KEY)
                 P-> P_L=Q;
        Else
                 P-> P_R =Q;

}
```

**Câu 19: loại bỏ 1 nút có giá trị X trên cây nhị phân tìm kiếm.**

*Ý tưởng*

Tìm kiếm xem khóa x có trên cây không. Tìm thấy, nút P trỏ vào nút cần xóa

- Nút P là nút lá, trường hợp này ta chỉ việc đem mối nối cũ trỏ tới nút P (từ nút cha của P) trỏ về NULL, rồi giải phóng P
- Nếu P là nút "nửa lá" ( nghĩa là nó chỉ có 1 nhánh con ( là cây con trái hoặc cây con phải)) thì nút thay thế P chính là nút gốc của nhánh con đó. Khi đó

ta điều chỉnh mối nối như sau: cho mối nối trỏ tới P( từ nút cha của P) trỏ vào nút gốc nhánh con của P. sau đó giải phóng P

- Nếu P là nút có đầy đủ 2 nhánh con thì nút thay thế P là nút lớn nhất thuộc cây con trái hoặc nút bé nhất của cây con phải. Khi đó thay vì xóa nút P, ta lấy giá trị của nút thay thế thay cho giá trị nút P rồi xóa nút thay thế

*Giải thuật*

```
BST_Delete(T,x)
{
P=T;Q=NULL;//về sau, P trỏ sang nút khác, luôn giữ Q luôn là cha của P
while(P!=NULL) //tìm xem có khóa x trên cây hay không
        {
        if (P -> Key == x) //tìm thấy x
                break;
        Q=P;
        if (x<P-> KEY )
                P=P-> P_L
        else
                P=P-> P_R;
        }
if( P=NULL ) //x không có trên cây nên không xóa được
        return;
if( P-> P_L != NULL && P -> P_R != NULL) //P có đầy đủ 2 con
        { //sẽ tìm nút cực phải của cây con trái làm nút thay thế
        Node= P;//ghi nhớ nút chứa khóa x
        Q=P; P= P ->P_L;//chuyển sang cây con trái để tìm nút cực phải
        while (P-> P_R; != NULL) //tìm đến nút cực phải
                {
                Q=P; P= P-> P_R;
                }
        //chuyển giá trị trong nút thay thế lên nút cần xóa
        Node ->KEY= P-> KEY;
        }
/* Nút cần xóa bây giờ là nút P, nó chỉ có thể là nút lá hoặc có một nhánh con: nếu P có một nhánh con thì dùng con trỏ Child trỏ vào nút gốc của nhánh con đó; còn nếu P là nút là thì cho Child =NULL*/
if( P-> P_L != NULL )
        Child = P->P_L;
else
        Child=P->P_R;
if (P==T ) //nếu nút cần xóa là gốc của cây
```

```
        T= Child;
Else // nút bị xóa không phải là gốc của cây thì lấy mối nối từ cha của nó là Q nối thẳng tới Child
        if( Q-> P_L ==P )
                Q->P_L = Child;
        else
                Q-> P_R=Child;
free(P); // giải phóng P
}
```

## Câu 20: sắp xếp nhanh ( Phân đoạn) Quick sort

*Ý tưởng*

Chọn một khoá ngẫu nhiên nào đó của đoạn làm "chốt" (Pivot). Mọi khoá nhỏ hơn khoá chốt được xếp vào vị trí đứng trước chốt, mọi khoá lớn hơn khoá chốt được xếp vào vịtrí đứng sau chốt. Sau phép hoán chuyển như vậy thì đoạn đang xét được chia làm hai đoạn khác rỗng mà mọi khoá trong đoạn đầu đều ≤ chốt và mọi khoá trong đoạn sau đều ≥ chốt. Hay nói cách khác: Mỗi khoá trong đoạn đầu đều ≤ mọi khoá trong đoạn sau. Và vấn đề trởthành sắp xếp hai đoạn mới tạo ra (có độ dài ngắn hơn đoạn ban đầu) bằng phương pháp tương tự.

*Giải thuật*

Chọn số đầu tiên làm khóa, cho mốt biến j chạy từ số thứ 2 đến hết dãy, biến j chạy từ cuối dãy đến đầu dãy, biến I chạy đến khi gặp vị trí có giá trị lớn hơn khóa thì dừng lại, j chạy đến khi gặp vị trí có giá trị nhỏ hơn thif dừng lại

- Nếu i<j thì đổi vị tri 2 giá trị tại i và j, tiếp tục thực hiện cho I và j chạy
- Nếu i>j thì đổi chỗ vị trí giá trị tại j với khóa, tiếp tục cho I và j chạy

Thực hiện đến khi ta phân chia được dãy làm 2 mảng, 1 mảng alf những số nhỏ hơn khóa, 1 mảng là những số lớn hơn khóa, tiếp tục thực hiện quy trình trên với các mảng đã phân chia

Thời gian thực hiện giải thuật $Txau(n)=O(n^2)$ : $Ttot(n)=O(n\log_2 n)$

```
DOICHO(a,b)
{
        x=a;
        a=b;
        b=x;
}
PHANDOAN(K,t,p,j)
{

        i= t+1; j = p;
        do
                {
```

```
            While((i<=j) && (K[i]<K[t])) i=i+1;
            While((i<=j) && (K[i]>K[t])) j=j+1;
            if(i<j)
                  {
                  DOICHO(K[i],K[j];
                  i=i+1;
                  j=j-1;
                  }

            }
      While(i<=j);
            DOICHO(K[t],K[j]);
}

SAPXEP(K, t,p)
{
      If(t<p)
            {
            PHANDOAN(K,t,p,j);
            SAPXEP(K, t,j-1);
            SAPXEP(K, j+1,p);
            }

}
```

## Câu 21: sắp xếp vun đống (Heapsort)

*Ý tưởng*

Để chọn ra số lớn nhất, ta dựa vào cấu trúc đống và để sắp xếp theo thứ tự tăng dần của các giá trị khóa là số, thì khóa lớn nhất sẽ được sắp xếp vào cuối dãy, nghĩa là nó được đổi chỗ với khóa đang ở "đáy đống", và sau phép đổi chỗ này một khóa trong dãy đã vào đúng vị trí của nó trong sắp xếp. Nếu không kể tới khóa này thì phần còn lại của dãy khóa ứng với 1 cây nhị phân hoàn chỉnh, vơi số lượng khóa nhỏ hơn 1, sẽ không còn là đống nữa, ta lại vun đống và thực hiện tiếp phép đổi chỗ giữa khóa ở đỉnh đống và khóa ở đáy đống tương tự như đã làm. Cho tới khi chỉ còn 1 nút thì các khóa đã được sắp xếp vào đúng vị trí của nó trong săp xếp.

*Giải thuật*

Như vậy sắp xếp kiểu vun đống (Heap sort) gồm 2 giai đoạn:

1. Giai đoạn tạo đóng ban đầu:

- Giải thuật thực hiện việc chỉnh lý 1 cây nhị phân với gốc root thỏa mãn điều kiện của

đống.
Cây con trái (gốc 2i+1) và cây con phải (gốc 2i+2) đều thỏa điều kiện của đống.
- Cây lưu trữ trên mảng K có n phần tử được đánh số từ 0.

```
ADJUST(root, n)
{
 Key = K[root]; // Key nhận giá trị khóa ở nút gốc
 While(root*2 <= n-1) // chừng nào root chưa phải là lá
 {
c = root*2 + 1; // Xét nút con trái của root
if((c < n-1) && (K[c] < K[c+1])) // nếu con phải lớn hơn con trái
        c = c + 1; // chọn ra nút có giá trị lớn nhất
if(K[c] < Key) // cả 2 nút con của root đều có giá trị nhỏ hơn Key
        break; // dừng
K[root] = K[c]; // chuyển giá trị từ nút con c lên nút cha root
root = c; // đi xuống xét nút con c
 }
 K[c] = Key; // Đặt giá trị Key vào nút con c
}
```

2. Giai đoạn sắp xếp, gồm 2 bước:
- Đổi chỗ + Vun đống được thực hiện (n-1) lần.

Hàm sắp xếp vun đống được thực hiện bởi giải thuật sau

```
HEAP_SORT(K,n)
{
for(i= |_n/2_| ; i<=0; r--) // Tạo đống ban đầu
ADJUST(i,n);
for(i = (n-1); i>=0; i--) // Sắp xếp
        {
        x = K[0];
        K[0] = K[i];
        K[i] = x;
        ADJUST(0,i);
        }
}
```

* Đánh giá giải thuật: với trường hợp xấu nhất có thể thấy rằng
- gian đoạn 1 (tạo đống): [n/2] lần gọi ADJUST(i,n)
- giai đoạn 2 (sắp xếp): n-1 lần gọi ADJUST(0,i)

=> có khoảng 3n/2 thực hiện ADJUST
- với cây nhị phân hoàn chỉnh có n nút thì chiều cao của cây lớn nhất cũng chỉ xấp xỉ

log(2)n-1

- số lượng so sánh giá trị khóa khi thực hiện hàm ADJUST xấp xỉ: 3n/2 * log(2)n
từ đó suy ra: Txấu(n)= O(log(2)n)
Ttb(n) = O(n*log(2)n)

## Câu 22: Sắp xếp hòa nhập (Merge-sort)

*Ý tưởng*

Hòa nhập 2 đường là phép hợp nhất 2 dãy khóa đã sắp xếp ddeerr ghép lại thành 1 dãy khóa có kích thước bằng tổng kích thước 2 dãy khóa ban đầu và dsayx khóa tạo thành cũng có thứ tự sắp xếp. Nguyên tắc thực hiện: so sánh 2 khóa đứng đầu 2 dãy, chọn ra khóa nhỏ nhất đưa vào miền sắp xếp (1 dãy khóa phụ có kích thước bằng tổng kích thước 2 dãy khóa đầu ở vị trí thích hợp) sau đó khóa này bị loại ra khỏi dãy khóa chứa nó. Quá trình tiếp tục đến khii một trong 2 dãy khóa đã cạn. Khi đó chỉ cần chuyển toàn bộ dãy khóa còn lại vào miền sắp xếp là xong

*Giải thuật*

```
MERGE(K,b,m,n,X)//hàm hòa nhập 2 mảng con thành mảng mới X được sx
        {
        i=t=b; j=m+1;
        /*t chạy trong miền sắp xếp, i chạy theo dãy khóa thứ nhất, j chạy theo dãy
        khóa thứ 2*/
        while(i<=m) and (j<=n) //chừng nào cả 2 dãy khóa
                If( K[i]<K[j] ) //so sánh 2 khóa tìm khóa nhỏ hơn rồi cho khóa đó vào
        miền sắp xếp
                        X[t++]=K[i++];
                else
                        X[t++]=K[j++];
        if (i>m) //mảng 1 hết khóa trước
                for(; j<=n;)
                        X[t++]=K[j++];
        Else //mảng 2 hết khóa trước
                for(; i<=m;)
                        X[t++]=K[i++];
        }
MPASS(K,X,n,h)
        {/* hàm thực hiện hòa nhập từng cặp mạch kế cận nhau, có độ dài h từ mảng
        K sang mảng X, n là số lượng khóa có trong K*/
        i=0;
        while(i<=n-2*h) //hòa nhập cặp mạch có độ dài h
                {
                MERGE(K, i, i+h-1, i+2*h-1, X);
                i= i+2*h;
                }
```

```
        if (i+h<n) //hòa nhập cặp mạch còn lại với tổng độ dài <2h
                MERGE(K, i, i+h-1, n-1, X);
        Else //chỉ còn 1 mạch
                for(;i<n;i++)
                        X[i]=K[i];
        }
STRAIGHT_MSORT (K,n)
        {/*để thực hiện lưu trữ mạch mới, dùng mảng phụ X[n], K và X sẽ được
        dùng luân phiên, h ghi nhận kích thước của các mạch, sau mỗi lợt h đc tăng
        gấp đôi*/
        h=1;
        do
                {
                MPASS(K,X,n,h);//Hòa nhập và chuyển các khóa vào X
                MPASS(X,K,n,2*h);//Hòa nhập và chuyển các khóa vào X
                h=4*h;
                }
        while( h<n-1 )
        }
```

## Câu 23: Quân hậu

Xét bàn cờ vua có kích thước 8x8, ta cần đặt 8 quân hậu vào bàn cờ vua sao cho k quân nào ăn quân nào, 8 quân hậu k ăn nhau tức chúng nằm ở các hàng, các cột, các dường chéo khác nhau. Giả sử, ta đặt 8 con hậu vào 8 cột: con hậu j nằm ở cột j, hàng i, hai đường chéo là (i+j) và (i-j)

Quy tắc chọn hàng: để hàng I được chấp nhận thì hàng i và 2 đường chéo i+j và i-j được tự do hay là k có con hậu nào trên hàng đó.

Xét trường hợp tổng quát:Ta tìm cách dặt quân hậu j vào hàng i cột j bằng cách cho i chạy từ 1 đến 8 nếu tìm được 1 vị trí an toàn thì:

- Đặt con hậu j vào hàng đấy. Vị trí I,j không an toàn nữa
- Nếu là con hậu cuối cũng thì thu được 1 kết quả
- Nếu không phải con hậu cuối cùng, ta đặt caccs con hậu còn lại ở các cột đến khi j=8 thì kết thúc
- Sau khi tìm được vị trí của con hậu j ta lại đặt vị trí i,j trở thành an toàn để tìm các cách khác nhau

Quy ước:

- x[j]=i: con hậu thứ j được đặt ở hành thứ i
- a[j]=1: hàng i an toàn
- b[i+j]=1: đường chéo i+j an toàn
- c[i-j]=1: đường chéo i-j an toàn

## Câu 24: giai thừa

TINHGIAITHUA()
{/*cho số nguyên n, giải thuật này thực hiện tính n!. ở đây sử dụng 1 ngăn xếp A mà đinhr đc trỏ bởi T. Mỗi phần tử của A là một cấu trúc gồm 2 tp:

- N: ghi gtri của n ở mức hiện hành
- RETADD: ghi địa chỉ quay lui

Lúc đầu ngăn xếp A rỗng: T=-1
Một biến cấu trúc TEMREC được dùng làm biến trung chuyển, nó cũng có 2 tp:

- PARA: tương ứng với n
- ADRESS: tương ứng với RETADD

ở đây giả thiết: lúc đầu TEMREC đã chứ các giá trị cần thiết, nghĩa là PARA chứ giá trịn n!, ADRESS chứa địa chỉ ứng với lời gọi trong chương trình mà ta gọi là DCC (địa chỉ chính)
các giải thuật PUSH,POP đc sd ở đây*/

```
1.//bào lưu giá trị của N và địa chỉ quay lui
        PUSH(A,T,TEMREC);
2.//tiêu chuẩn cơ sở đã đạt chưa?
        if(T.N==0)
        {
                GIAI_THUA =1;
                Goto Bước 4;
        }
        Else
        {
                PARA = T.N – 1;
                ADDRESS=Bước 3;
        }
Goto Bước 1;
3// tính N!
GIAI_THUA = T.N*GIAI_THUA;
4.//khôi phục giá trị N và địa chỉ quay lui
TEMREC = POP (A,T);
Goto ADDRESS;
5,//kết thúc
Return GIAI_THUA;
}
```

## Câu 25: Duyệt cây thứ tự sau

```
TTSAU_S(T)//hàm không đệ quy duyệt cây thứ tự sau
{//trong phép duyệt này, nút gốc chỉ được thăm sau khi đã duyệt xong 2 con của nó. Như vậy, chỉ từ khi con phải đi lên gặp gốc thì gốc mới được thăm, chứ k phải ở lần gặp gốc khi từ con trái đi lên. Do đó, để pphaan biệt, người ta đưa thêm dấu âm vào địa chỉ ( địa chỉ âm) để đánh dấu cho lần đi lên từ con phải */
If(T==NULL)//khởi đầu
        {
        Printf('cây rỗng');
        Returm;
        }
Else
        {
        TOP=-1;
        P=T;
        }
While (1) //2. Thực hiện duyệt
        {
        While(P!=NULL)//lưu địa chỉ gốc và xuống con trái
                {
                PUSH(S,TOP,P);
                P=P->P_L;
                }
        While(S[TOP]<0)//thăm nút mà con trái và con phải đã duyệt
                {
                P=POP(S,TOP);
                Printf(P->DATA);
                If(TOP==0)
                        Return;
                }
        P=S[TOP]->P_R;//xuống con phải và đánh dấu
        S[TOP]=-S[TOP];
        }
}
```

## Câu 26: ưu nhược các phương pháp sắp xếp

- sắp xếp nhanh ( QS )

- Ưu điểm : Có hiệu suất tốt nhất O(nlog$_2$n), xấu nhất n$^{2.}$ là thuật toán chạy nhanh nhất ở trường hợp trung bình . k cần bộ nhớ phụ. dùng tốt cho danh sách liên kết
- Nhược điểm: trường hợp xấu nhất chiêm O (n$^2$). code khá phức tạp. không ổn định. tùy vào cách chọn pivot mà tốc độ của thuật toán nhanh hay chậm. cần thêm không gian nhớ cho ngăn xếp, để bảo lưu thông tin về các phân đoạn sẽ được xử lý tiếp theo
- Hiệu suất :tốt nhất : log tệ nhất : O( )
- Nhận xét: hiệu quả thực hiện của giải thuật Quick sort phj thuộc vào viễ chọn giá trị mốc
  + tường hợp tốt nhất xảy ra nếu mỗi lần phân hoạch đoạn đều chọn được phần tử median( phần tử lớn hơn hay bằng nửa số phần tử, nhỏ hơn hay bằng nửa số phần tử còn lại làm mốc, khi đó dãy đc phân chia thành 2 phần bằng nhau, và cần log2n lúc phân hoạch đoạn thì sắp xếp xong.
  + nhưng nếu mỗi bước phân hoạch đoạn phhaanf tử được chọn có giá trị cực đại hay cực tiểu làm mốc, dãy sẽ bị phân chia thành 2 phần k đều nhau: 1 phần chỉ có 1 phần tử, phần còn lại gồm n-1 phần tử, do vậy cần thực hiện n bước phân hoạch đoạn mới sắp xếp xong nên có tổng kết:
  Độ phức tạp th tốt nhất: O(nlog$_2$n), xấu nhất O(n$^2$)

- Sắp xếp vun đống ( HS )
- Ưu điểm :hiệu suất của thuật toán cao : tốt nhất nlog tệ nhất : nlog
- Nhược điểm :dù bất cứ dữ liệu nào trong mọi trường hợp nó đều tiêu tốn nlog2n. code phức tạp. cần 1 nút nhớ phụ để thực hiện đổi chỗ. Khi dãy số đã sắp xếp có thứ tự thì giải thuật này k hiệu quả
- Nhận xét: đối với HS 1 cây nhị phân hoàn chỉnh có n nút thì chiều cao cây đó là log2(n+1). Khi tạo đống cũng như vun đống trong giai đoạn sắp xếp, th xấu nhất thì số lượng phép so sánh cũng chỉ tỷ lệ với chiều cao của cây. Do đó có thể suy ra trong trường hợp xấu nhất của thời gian thực hiện chỉ là O(nlog2n). việc đánh giá tgian thực hiện trung bình phức tạp hơn, cấp dộ lớn của tgian thực hiện tb gthuat này là O(nlog2n)

- Sắp xếp hòa nhập ( MS )
- Ưu điểm : hiệu suất của MS rất cao , tgian O(nlog2n)
- Nhược điểm : khi cài đặt thuật toán đòi hỏi thêm không gian bộ nhớ để lưu các dãy phụ. Chi phí không gian khá lớn (đòi hỏi tới 2n phần tử nhớ,gấp đôi phương pháp thông thường) Hạn chế này khó chấp nhận vì trong thực tế các dãy đc sắp xếp thường có kích thước lớn , vì vậy thuật toán trộn thường đc dụng đề sắp xếp các cấu trúc dữ liệu khác phù hợp hơn như danh sách liên kết or file. code khá phức tạp
- Nhận xét: độ phức tạp là nlogn bất chấp trạng thái dữ liệu vào

- Nhận xét khi sử dụng phương pháp sắp xếp :

Cùng một mục đích sắp xếp như nhau mà có rất nhiều phương pháp và kỹ thuật giải quyết khác nhau. Cấu trúc dữ liệu được lựa chọn để hình dung đối tượng của sắp xếp đã ảnh hưởng rất sát tới giải thuật xử lý

Các phương pháp sắp xếp đơn giản đã thể hiện 3 kỹ thuật cơ sở của sắp xếp , cấp độ lớn của thời gian thực hiện chung là O ($n^2$ ) vì vậy chỉ nên sử dụng chúng khi n nhỏ . các giải thuật cải tiến như QS, HS đã đạt được hiệu quả cao nên đc sử dụng khi n lớn. MS cũng k kém hiệu lực về tgian thực hiện nhưng nhưng về không gian thì đòi hỏi của nó k thích nghi vs sắp xếp trong . nếu bảng sắp xếp vốn có khuynh hướng hầu như đã đc sắp sẵn thì QS lại k nên dùng . nhưng nếu ban đầu bảng có khuynh hướng ít nhiều có thứ tự ngược vs thứ tự sắp xếp thì HS lại tỏ ra thuận lợi. Việc khẳng định 1 kỹ thuật sắp xếp nào vừa nói trên luôn tốt nhất vs mọi kỹ thuật khác là điều k nên. Việc chọn 1 phương pháp thích hợp thường tùy thuộc vào từng yêu cầu , từng đkiện cụ thể

Chúc ace thi tốt nhá nhá

_Nắng_